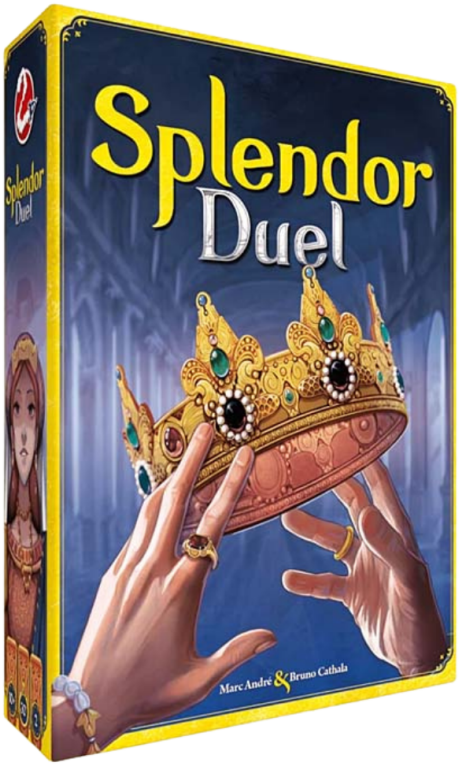
Celine Ozkaya
Remy Théolier
Lise Leclef
Sacha Malaterre
Raphaël Chauvier

LO21
Projet Splendor Duel

-

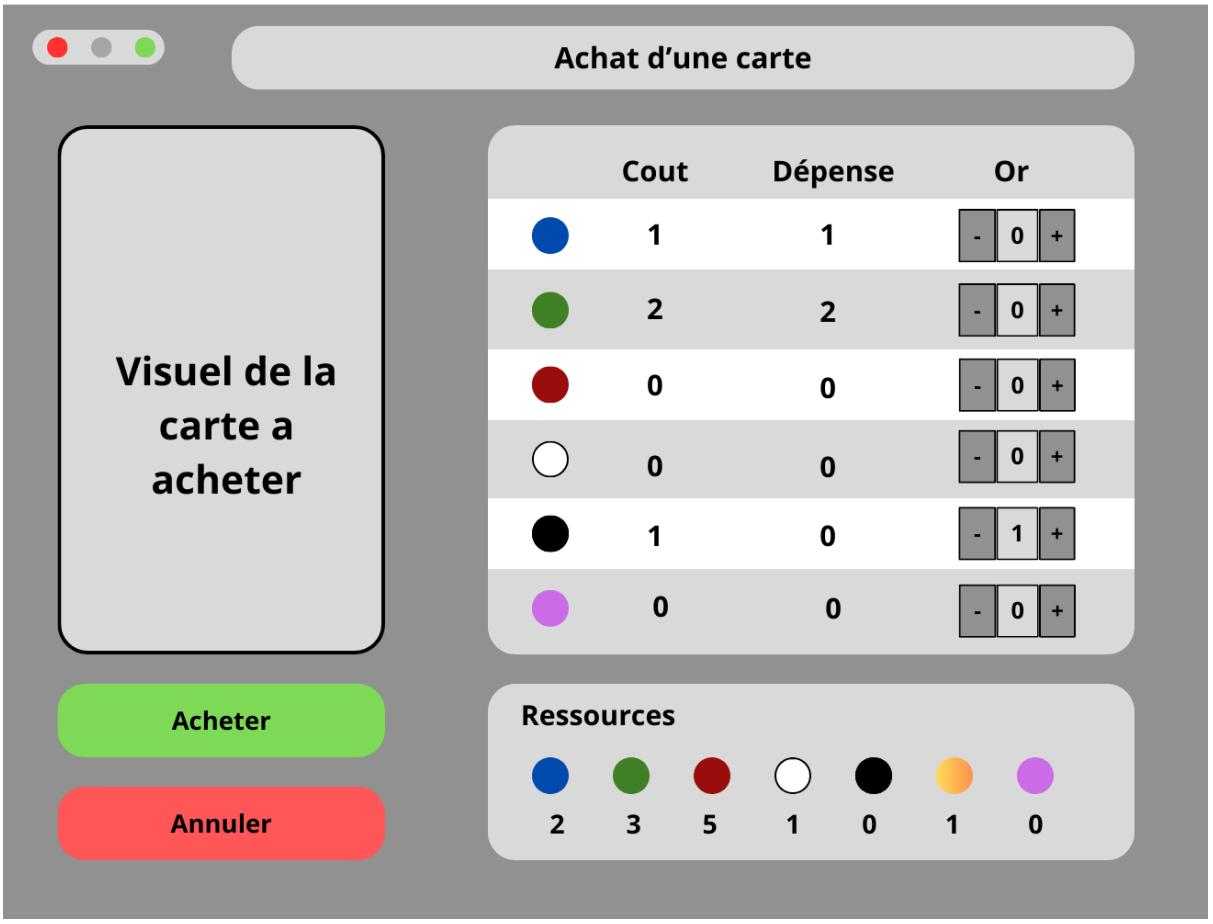
Compte Rendu 3



I. Visuels

a) Joueurs

Page “Acheter une carte” (Celine)



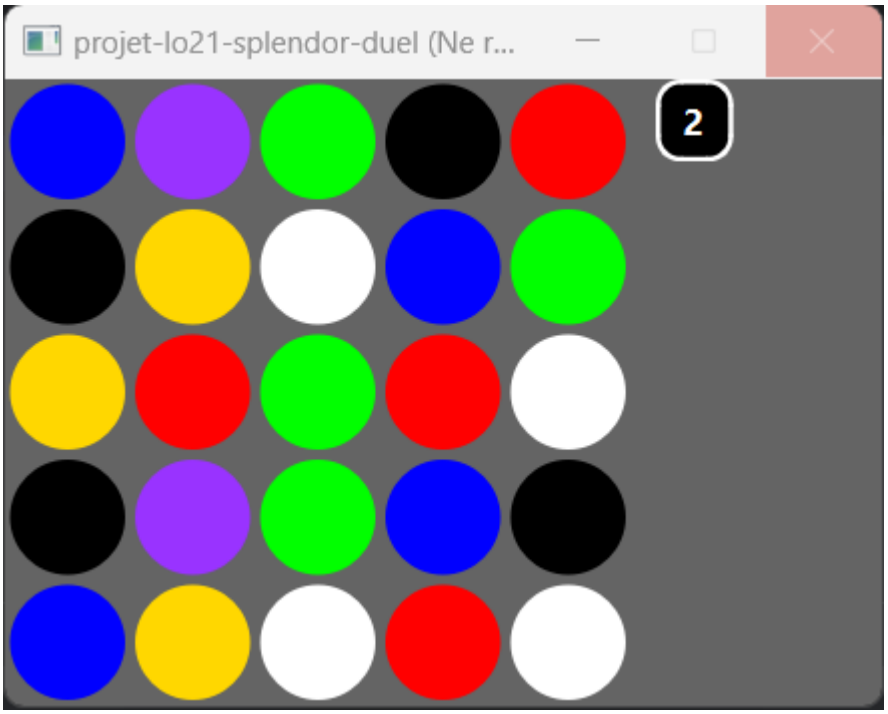
Le tableau Ressource représente les jetons que possède le joueur.

La colonne Coût représente le coût de la carte par jeton (ici, la carte coûte 1 bleu, 2 vert et 1 noir). La colonne Dépense représente le nombre de jetons de la couleur demandés que le joueur doit dépenser. La colonne Or permet de sélectionner le nombre de jetons Or que l’on veut dépenser pour remplacer un jeton. Ainsi, Coût = Dépense + Or, étant donné que Dépense décroît quand Or augmente (Cout Noir = 1, Or pour remplacer Noir = 1 donc pas besoin de Dépense Noir).

b) Plateau de Jetons

La plateau de Jetons est visuellement opérationnel, il suffira juste de remplacer les dessins de cercles par des boutons cliquables.
Dans le widget, on a une grille de jetons (5*5) représentés par un cercle de la couleur des jetons pour chacun d'entre eux. On retrouve juste à côté le compteur de privilèges.
Aussi, le plateau se met bien à jour quand on retire/place des jetons ou des privilèges, mais il faut encore faire en sorte que la fenêtre puisse fonctionner en même temps que le mode console (on se concentrera plutôt à rendre interactifs les widgets et à directement les faire interagir avec des fonctions).

Voici un aperçu du plateau de jetons une fois la partie chargée, à l'état initial:



La case noire représentant les privilèges encore sur le plateau.

Pour créer cette fenêtre, un Qlabel à été utilisé pour mettre en forme le nombre de privilèges dans une classe, Qwidget a été utilisé quant à lui pour générer les cercles représentants les jetons dans une autre classe, et aussi pour générer le plateau avec les 25 jetons, dans une dernière classe qui utilise la précédente.

Une classe temporaire MainWindow est finalement utilisée pour mettre ensemble les deux premiers widgets créés pour ce projet, qui sera complétée au fur et à mesure.

Les mises à jour du plateau sont effectuées à l'aide du design pattern observer sur le plateau (contenant les jetons et les privilèges).

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. Il a été légèrement modifié depuis le dernier rendu. Les éléments modifiés sont en **gras**.

Variable	Signification
Classe Player	
string name	nom du joueur
int nbPrivilege	nombre de privilèges
Type type	type du joueur : humain ou IA
int nbCrown	nombre de couronnes total parmis les cartes du joueur
int prestigePoints	points de prestige totaux
int nbTokens	nombre de tokens du joueur, permet de vérifier si le joueur a moins de 11 jetons
vector<JewelryCards*> jewelryCards	tableau représentant les cartes joailleries qu'un joueur a acheté
vector<RoyalCards*> royalCards	tableau représentant les cartes royales (0, 1 ou 2) qu'un joueur a récupéré
vector<JewelryCards*> reserve	tableau représentant les cartes réservées par un joueur
unordered_map<TokenColor, vector<Token>> tokens	Dictionnaire de type {TokenColor, vector<Token>} Il représente les instances de jetons (possédées par un joueur) par couleur
unordered_map<TokenColor, int> tokenSummary	Dictionnaire représentant le nombre de jetons par couleur du joueur
SummaryCard 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
Fichier Partie (.h et .cpp)	
Game	Instance de jeu
GameTable	Plateau de jeu
round	Tour en cours
Fichier Controller (.h et .cpp)	
Controller	Controlleur de jeu
currentplayer	Joueur en cours

III. Nouvelle décomposition des tâches et tâches déjà effectuées

A. Tâches effectuées:

En plus des tâches du dernier compte-rendu, nous avons pu:
-Corriger et affiner les différentes classes principales du jeu (Cartes, Joueur,...)
-Utilisation de design pattern pour répondre à de nombreux problèmes
-Implémenter le système de création, et celui de gestion d'une partie complète, rendant le jeu intégralement fonctionnel en mode console.
-Entamer l'implémentation de la version GUI du jeu
-Entamer l'implémentation de la gestion d'une sauvegarde

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

- Continuer l'implémentation de chaque sous module QT, pour ensuite rassembler chaque partie dans une même console. A associer avec des fenêtres de gestion de partie. ~15h
- Créer l'Intelligence Artificielle permettant de simuler un joueur ~5h
- Finalisation du système de sauvegarde ~3h
- Relecture, nettoyage, correction et ajout de commentaires au code (Noms de variables plus claires, utilisations des instances des singletons, sans avoir à les mettre en argument, etc…) ~3h pp.

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 :

a) Back-end

- getName(), getNbPrivilege(), getCrowns(), getPrestige(), getNbTokens() 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 addToken(Token &token) 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() ajoute et retire respectivement un privilège
- 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) vérifie qu'un joueur à la moyen d'acheter une carte
- void spendRessources(const Jewerlry &card) effectue le paiement de la carte
- **void reserveOneCard(Jewerlry &card) permet d'ajouter une carte à l'inventaire de cartes réservées**

- **bool canReserveCard() permet de vérifier qu'un joueur peut réserver une carte. Elle est appelée au moment où un joueur choisit d'effectuer une action de réservation afin de vérifier sa capacité à effectuer ladite action.**
- **getBlueSummary(), ainsi que tous les getters de cartes résumées retourne la carte résumé de la couleur demandé.**
- **getMaxPrestigeColor() permet d'accéder au nombre de prestiges maximal au sein des couleurs que possède un joueur.**
- **getBonusSummary() crée un vector donnant les bonus totaux par couleur**
- **void removeToken(TokenColor color) permet de retirer un jeton de couleur donnée de l'inventaire du joueur, et retourne l'instance du jeton en vue de l'ajouter au sac de jeton/ le placer dans l'inventaire du joueur adverse selon les actions en cours.**
- **void actionBuyReservedCard(JewelryCard& card) permet d'acheter une carte réservée.**

A ce stade, les méthodes de la classe joueurs sont implémentées.

b) Front-end

Pour ce qui est du QT, un widget PlayerQT a été créé qui représente un joueur et ses différentes caractéristiques. Ce widget n'est pour l'instant pas achevé, mais voici une représentation de ce qu'il ressemblera (normalement) à la fin :

Les cartes résumés sont représentées par un rectangle de la couleur concernée et de deux compteurs : celui en haut pour le total de points de prestige des cartes de cette couleur, et en bas le total des bonus de la couleur. La carte de couleur Or servira à représenter les cartes royales.

Pareil pour les jetons résumés, ils sont représentés par un cercle de leur couleur et d'un compteur. Le compteur est le nombre de jetons de cette couleur possédés par le joueur.

Pour simplifier la structure du layout global, des widgets CardWidget et TokenWidget ont été créés.

CardWidget a comme méthodes updateBonus et updatePrestige permettant de mettre à jour ces deux attributs par rapport aux attributs du joueur concerné.

TokenWidget a comme méthode updateNumToken permettant de mettre à jour le nombre de tokens correspondant

Toujours dans la même logique, PlayerQT possède pour l'instant les méthodes updatePrivilege, updateCrown et updateTotalPrestige.

Pour ce qui est de la réserve de carte, nous avons fait le choix de la représenter par un bouton qui lance une popup, de cette manière la réserve est uniquement visible lorsque l'utilisateur le choisit. Ceci évite donc que le joueur adverse puisse voir la réserve de l'autre. Ce bouton sera actif uniquement lors du tour du joueur concerné. La popup contiendra donc 3 emplacements, et un bouton retour. Les emplacements seront par défaut grisés et barrés, mais quand ils seront remplis ils contiendront l'affiche normal des cartes concernées.

Concernant la progression de cette représentation QT du joueur, il manque encore la réserve qui n'est pour l'instant pas fonctionnelle, et il y a encore des problèmes dans le remplissage des gridlayout des jetons et des cartes. Il faudra donc encore faire du débuggage. Il reste probablement 3-4 heures de travail sur ce widget avant qu'il soit fini.

2. **Fichiers Jeton (Raphaël)**

Le classe jeton été déjà implémentée de façon complète et utilisable au précédent compte-rendu. Pour avoir un code plus clair et plus pratique au sein du projet, j'ai implémenté le design pattern singleton pour chaque classe unique lors d'une partie. C'est le cas de la classe TotalToken, Board, Bag, TotalPrivilege. Un design pattern iterator a été implémenté dans la classe Board pour permettre de parcourir tous les jetons de celui-ci.
Quelques méthodes supplémentaires ont aussi été implémentées: En plus des différents design pattern (pour le jeu et l'affichage graphique), les méthodes suivantes ont été implémentées:
-bool CellColor(size_t i, size_t j, TokenColor color) const permettant de vérifier la couleur d'une cellule.
-bool hasTokenOfColor(TokenColor color) const permettant de vérifier si le plateau a bien au moins un jeton d'une certaine couleur.
-bool containsOnly(TokenColor color) const permettant de vérifier si le plateau contient uniquement une certaine couleur.
-bool isCellEmpty(size_t i, size_t j) const
-unsigned int getNbTokens() const permettant de récupérer le nombre de jetons.

Finalement, une nouvelle couleur à été ajoutée à TokenColor: None. Permettant de pouvoir retourner des jetons vides.

Le temps de travail ajouté estimé pour ce fichier est de 5h.

3. Fichiers Cartes

Les modifications apportées aux différentes classes et méthodes des fichiers Cards.h et Cards.cpp, par Sacha et Rémy, ont été nombreuses.

En premier lieu nous avons implémenté les constructeurs des 3 différents decks de niveau des classes respectives : Deck_Level_one, Deck_Level_two, Deck_Level_three.
Ces constructeurs impliquent d'aller chercher les données des cartes dans une base de donnée, pour ce faire nous avons inclus des fichiers sqlite3. Donc chacun de ces constructeurs vient itérer uniquement sur les cartes de son niveau dans la database et créer carte par carte le deck avec les bons attributs.

Pour permettre de correctement récupérer les éléments comme les Abilities, TokenColor etc. Nous avons dû implémenter des fonctions rangées dans un namespace Utility qui servent à faire des conversion entre string et les enum. Les attributs des cartes joailleries étant complexes nous avons dû apprendre à manipuler des éléments nouveaux comme des unordered_map.

En ce qui concerne l'arrangement aléatoire des decks, plusieurs options s'offraient à nous. Nous avons décidé de rajouter le mélange à la fin de chacun des constructeurs des decks. Nous utilisons une seed générée à partir de l'horloge interne de l'ordinateur pour s'assurer que le mélange soit unique. Notre vecteur "pioche" de cartes joailleries est donc créé puis mélangé.

Pour tester ce code nous avons créé un main temporaire avec quelques fonctions d'affichages console dans un projet annexe. Nous rencontrions d'énormes problèmes au début, des données qui n'avaient aucun sens (ex : level = 91263921873). Le souci était le fait que nous supprimions les objets temporaires après les avoir ajoutés à la pioche dans les constructeurs des decks.

Après s'être occupés des cartes joailleries, nous devions construire le deck des cartes royales. Cela suit à peu près le même chemin que pour les constructeurs de decks des cartes joailleries. On vient chercher les données de la database dans une autre table dédiée.

Il nous fallait maintenant revenir sur la classe Pyramid_Cards et ses méthodes. Le constructeur ainsi que la méthode draw ne sont pas bien différents, simplement l'un remplit tout le jeu et l'autre ne remplit qu'une seule case manquante. Pour le moment la méthode takeCard ne change.

Les instances des différents decks ainsi que la pyramide sont uniques. Nous avons donc implémenté des singletons pour chacune de ces classes. Des modifications dans les différentes méthodes des classes s'en sont suivies car nous traitons maintenant avec des pointeurs vers l'instance unique et non directement l'instance. (passage du . vers la ->).

Nous voyons déjà des voies d'améliorations pour notre code. Notamment pouvoir faire une seule fonction pour les constructeurs des decks au lieu de copier quasiment 3 fois la même chose, grâce au polymorphisme.

Globalement après avoir implémenté ce code, notre partie Carte fonctionne en console. Nous pouvions générer les decks, print les cartes ainsi qu'une pyramide. Nous sommes donc passé à la partie graphique.

Au début nous pensions générer les cartes avec une multitude de widgets pour chaque information (niveau, prestige, coût etc). Concrètement, nous avons par exemple créé une méthode "toHexaDecimal" prenant en paramètre la couleur indiquée en toutes lettres dans la base de données pour l'objet étudié, et permettant ainsi d'obtenir un code couleur en hexadécimal interprétable par QT pour créer une carte avec un fond de la couleur correspondante. Toutefois, nous nous sommes rendus compte que cela complexifiait la tâche et nous n'avions que peu de temps pour nous familiariser correctement avec Qt. Nous avons donc penché sur l'option d'importer des visuels pour chacune des cartes. Nous avons rajouté un id dans les attributs des cartes pour bien importer la bonne image à la bonne carte.

Concernant les images, au moment où nous écrivons ces lignes, nous utilisons des images de cartes à jouer classiques en attendant de savoir comment nous allons récupérer les visuels des cartes du jeu. Deux options sont envisageables : scanner l'entièreté du deck réel ou bien générer les cartes via un logiciel de génération d'images par Intelligence Artificielle. Quoi qu'il en soit, nous avons déjà implémenté les classes graphique des cartes, du plateau ainsi que des différentes pioches.

Concernant la méthode de travail utilisée, nous nous sommes améliorés. En effet, si précédemment, nous nous retrouvions pour travailler à deux sur le même ordinateur et partager nos idées, des divergences d'emploi du temps et l'envie d'accroître notre efficacité nous ont poussés à utiliser le module Live Share de Visual Studio. Celui-ci nous a permis de travailler à distance l'un de l'autre, sur le même fichier via une connexion internet. La grande force de cet outil est que nous pouvions travailler avec un IDE différent du moment que les fichiers de travail étaient ouverts via Visual Studio. Avec le travail personnel à la maison en plus nous pouvons estimer notre temps de travail à 4h par semaine par personne.

4. Fichiers Partie (Raphael)

Deux fichiers ont été ajoutés: Partie et Controller.
Le fichier partie contient une classe GameException, qui sert à soumettre les erreurs liées à une partie. Une classe GameTable qui contient le plateau de jetons, le sac, les joueurs et les cartes, et permet de récupérer chacune de ces instances. Une classe Game, qui permet d'obtenir le nombre de tours en cours, de définir les joueurs, etc.
Finalement, le design pattern builder a été implémenté pour construire le jeu de zéro, de façon sécurisée, mais aussi plus tard grâce à une sauvegarde.

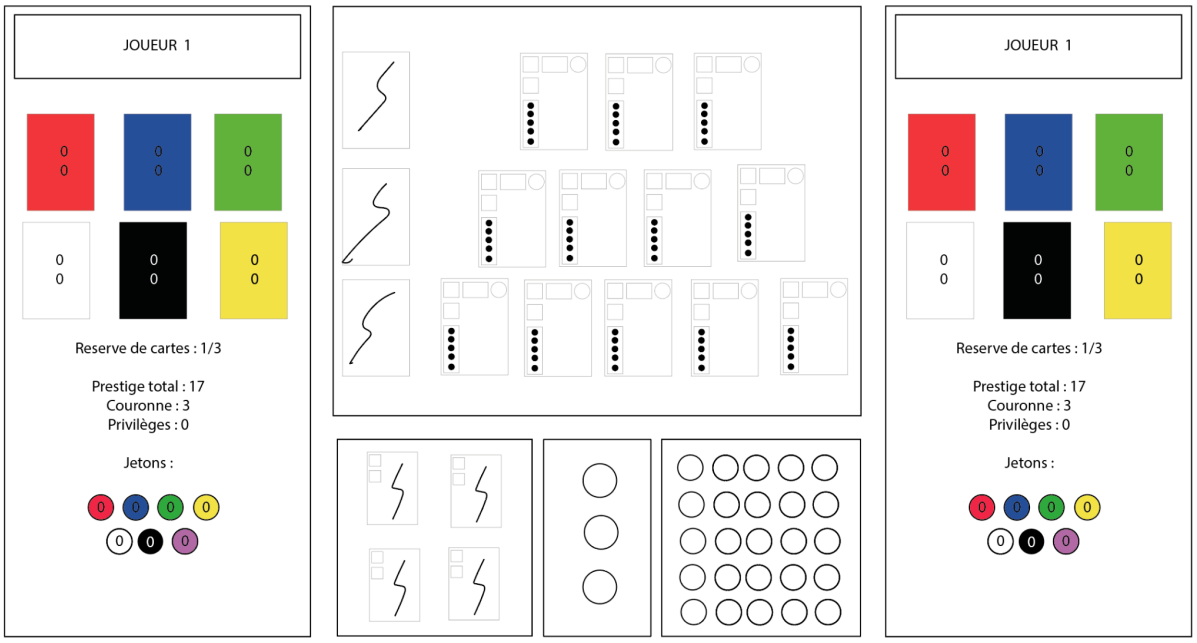
Le fichier controller contient une classe controller permettant de gérer le déroulement d'une partie, et ses règles, de façon complète. Il contient une instance de Game, ainsi que le joueur courant. De nombreuses méthodes ont été implémentés pour respecter les règles du jeu et permettre son bon déroulement.

Pour plus de détails sur l'implémentation de ces deux fichiers, se référer au code source.

Le temps qui a été nécessaire à l'implémentation de la partie en mode console est de 20h.

Grâce à l'implémentation de ces deux fichiers, on peut exécuter le jeu en mode console complètement.

Globalement, l'interface jouable d'une partie ressemblera à cela :



Nous nous réservons toutefois la possibilité de changer ce patron créé par Rémy en fonction de l'avancement du projet.

5. Fichier cards.db

Ce fichier est le fichier contenant la base de données des cartes. Modifié depuis le dernier rendu, voici sa nouvelle structure :

Table : JewelryCards
Champs - TYPE :
prestige - INTEGER,
crown - INTEGER,
cost_w - INTEGER,
cost_v - INTEGER,
cost_n - INTEGER,
cost_p - INTEGER,
cost_r - INTEGER,
color - TEXT,
level - INTEGER,
ability1 - TEXT,
ability2 - TEXT,
bonus_nb - INTEGER,
jewelryId - INTEGER

Table : RoyalCards
Champs - TYPE :
royalid- INTEGER,
capacity - TEXT,
prestige - INTEGER

Ces tables ont été modifiées via la solution logiciel open source DB Browser for SQLite permettant de visualiser très simplement la base de données elle-même.

Contrairement à la première base de données, celle-ci comporte donc deux tables : l'une pour les cartes royales et l'autre pour les cartes joailleries. Une fusion est bien évidemment possible en ajoutant un champ "type" ayant pour valeur "royal" ou "jewerly" mais par soucis de lisibilité, nous avons choisi de créer deux tables distinctes. Le champ correspondant à l'identifiant de la carte permet de la retrouver très simplement dans notre programme QT et de lier l'instance de chaque carte de notre programme à ses données et son image correspondante (cf fichier Cartes).

2h ont été nécessaires pour la création de ce fichier, de la réflexion jusqu'à sa finalisation.

6. Fichier save.db

Lorsqu'une partie est lancée, il doit être possible de la mettre "en pause" et de la reprendre plus tard. Pour cela, Rémy va mettre en place un système de sauvegarde au tour par tour. Actuellement, seul le fichier de sauvegarde a été créé (1h30 ont été nécessaires) et le code est en train d'être développé.

La première étape était de réfléchir à propos de chaque élément en jeu dans une partie et du choix de le sauvegarder ou non dans une base de données plus ou moins temporaire. Le fichier save.db sert donc à sauvegarder ces données.

Lesdites données sont :

Table : bag → *sert à sauvegarder le contenu du sac de jetons soit le nombre de jetons de chaque couleur.*

"champ" - TYPE :

```
"tokenBlue"    INTEGER,
"tokenWhite"   INTEGER,
"tokenGreen"   INTEGER,
"tokenBlack"   INTEGER,
"tokenRed"     INTEGER,
"tokenPerle"   INTEGER,
"tokenGold"    INTEGER,
"tokenNone"    INTEGER
```

Table : binCard → *sert à sauvegarder les cartes déjà achetées pour ne pas les remettre en jeu (nous ne conservons pas les cartes mais seulement un résumé (summary) des possessions de chaque joueur.*

"champs" - TYPE :

```
"idCard"       INTEGER,
"typeCard"     INTEGER
```

Table : boardPrivilege → *sert à sauvegarder le nombre de privilèges sur la table pouvant être pris par les joueurs.*

"champs" - TYPE :

```
"privilege"    INTEGER
```

Table : boardRoyal → *sert à sauvegarder les cartes de type Royal encore en jeu.*

“champs” - TYPE :

"idCard" INTEGER,
"xCard" INTEGER

Table : boardToken → *sert à sauvegarder la position et la couleur de chaque jeton encore sur le plateau.*

“champs” - TYPE :

"xToken" INTEGER,
"yToken" INTEGER,
"colorToken" TEXT

Table : infopartie → *sert à sauvegarder les informations de la parties telles que le tour et le joueur venant de jouer/à jouer (à définir plus tard dans le code).*

“champs” - TYPE :

"turn" INTEGER,
"currentPlayer" INTEGER

Table : player → *sert à sauvegarder le “summary” de chaque joueur, à savoir son identifiant, son pseudo, son type (humain ou IA), son nombre de privilèges, ses jetons et ses couronnes.*

“champs” - TYPE :

"id" INTEGER,
"pseudo" TEXT,
"typePlayer" TEXT,
"privileges" INTEGER,
"tokenBlue" INTEGER,
"tokenWhite" INTEGER,
"tokenGreen" INTEGER,
"tokenBlack" INTEGER,
"tokenRed" INTEGER,
"tokenPerle" INTEGER,
"tokenGold" INTEGER,
"crown" INTEGER

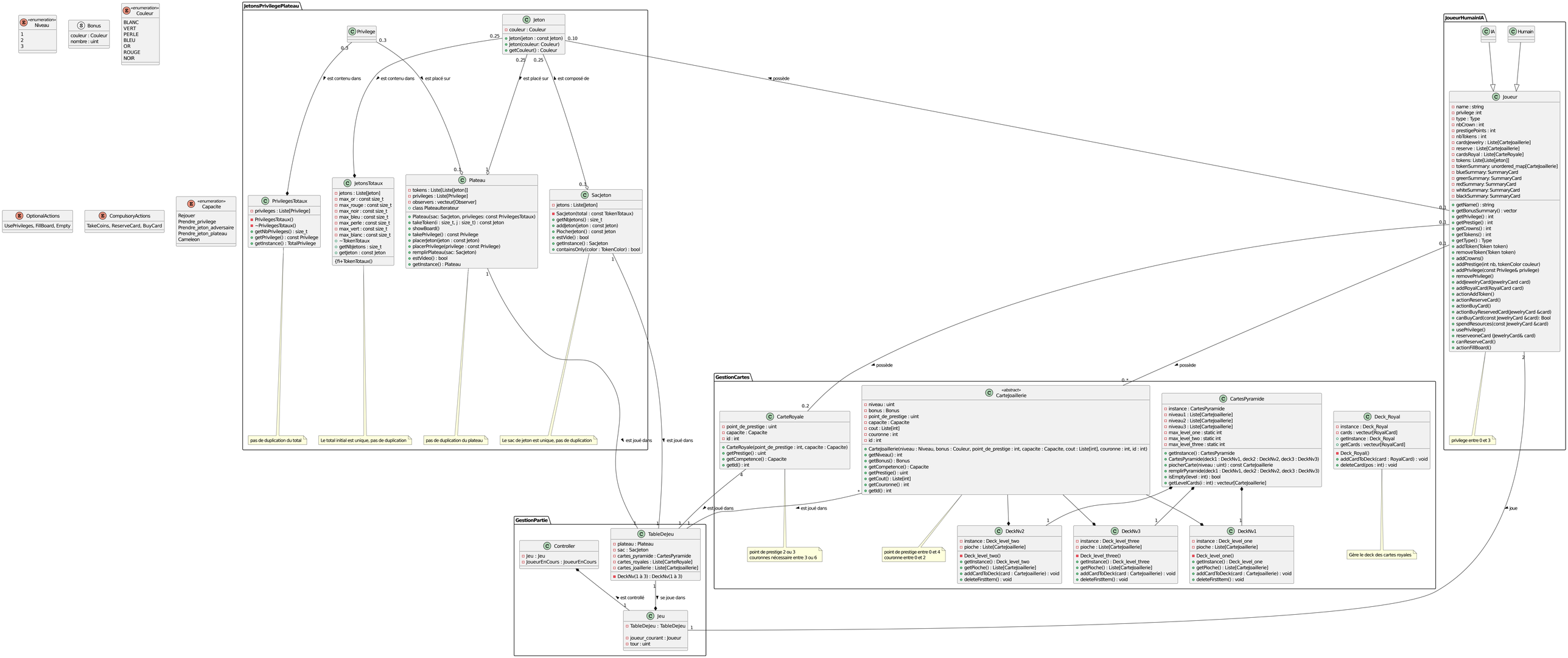
Table : pyramid → *sert à sauvegarder l’état de la pyramide (position et identifiant de chaque carte présente dans la pyramide.*

“champs” - TYPE :

"xCard" INTEGER,
"yCard" INTEGER,
"idCard" INTEGER

Note : Rémy a, pour le moment, choisi de ne pas sauvegarder les cartes présentes dans les pioches, ce qui signifie que le système de sauvegarde devra être capable de différencier les cartes non jouées et de n'ajouter que celles-ci aux pioches correspondantes lorsque la partie sera reprise.

V. UML


«enumeration»
Niveau
1
2
3
Bonus
couleur : Couleur
nombre : uint
«enumeration»
Couleur
BLANC
VERT
PERLE
BLEU
OR
ROUGE
NOIR
JetonsPrivilegePlateau
Privilege
Jeton
couleur : Couleur
jeton(jeton : const Jeton)
jeton(couleur: Couleur)
getCouleur() : Couleur
0..3
0..3
0..25
0..10
0..25
0..25
est contenu dans
est contenu dans
est placé sur
est placé sur
est composé de
possède
PrivilegesTotaux
privileges : Liste[Privilege]
PrivilegesTotaux()
~PrivilegesTotaux()
getNbPrivileges() : size_t
getPrivilege() : const Privilege
getInstance() : TotalPrivilege
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()}
Plateau
tokens : Liste[Liste[Jeton]]
privileges : Liste[Privilege]
observers : vecteur[Observer]
class PlateauIterateur
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)
estVide() : bool
getInstance() : Plateau
SacJeton
jetons : Liste[Jeton]
SacJeton(total : const TokenTotaux)
getNbJetons() : size_t
addJeton(jeton : const Jeton)
PiocherJeton() : const Jeton
estVide() : bool
getInstance() : SacJeton
containsOnly(color : TokenColor) : bool
0..1
1
0..1
1
1
est joué dans
est joué dans
«enumeration»
Capacite
Rejouer
Prendre_privilege
Prendre_jeton_adversaire
Prendre_jeton_plateau
Cameleon
OptionalActions
UsePrivileges, FillBoard, Empty
CompulsoryActions
TakeCoins, ReserveCard, BuyCard
pas de duplication du total
Le total initial est unique, pas de duplication
pas de duplication du plateau
Le sac de jeton est unique, pas de duplication
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
getBonusSummary() : vector
getPrivilege() : int
getPrestige() : int
getCrowns() : int
getTokens() : int
getType() : Type
addToken(Token token)
removeToken(Token token)
addCrowns()
addPrestige(int nb, tokenColor couleur)
addPrivilege(const Privilege& privilege)
removePrivilege()
addJewelryCard(JewelryCard card)
addRoyalCard(RoyalCard card)
actionAddToken()
actionReserveCard()
actionBuyCard()
actionBuyReservedCard(JewelryCard &card)
canBuyCard(const JewelryCard &card): Bool
spendResources(const JewelryCard &card)
usePrivilege()
reserveoneCard (JewelryCard& card)
canReserveCard()
actionFillBoard()
0..1
0..1
0..1
2
privilege entre 0 et 3
possède
possède
0..2
0..*
joue
GestionCartes
CarteRoyale
point_de_prestige : uint
capacite : Capacite
id : int
CarteRoyale(point_de_prestige : int, capacite : Capacite)
getPrestige() : uint
getCompetence() : Capacite
getId() : int
4
est joué dans
point de prestige 2 ou 3
couronnes nécessaire entre 3 ou 6
«abstract»
CarteJoaillerie
niveau : uint
bonus : Bonus
point_de_prestige : uint
capacite : Capacite
cout : Liste[int]
couronne : int
id : int
CarteJoaillerie(niveau : Niveau, bonus : Couleur, point_de_prestige : int, capacite : Capacite, cout : Liste[int], couronne : int, id : int)
getNiveau() : int
getBonus() : Bonus
getCompetence() : Capacite
getPrestige() : uint
getCout() : Liste[int]
getCouronne() : int
getId() : int
*
est joué dans
point de prestige entre 0 et 4
couronne entre 0 et 2
CartesPyramide
instance : CartesPyramide
niveau1 : Liste[CarteJoaillerie]
niveau2 : Liste[CarteJoaillerie]
niveau3 : Liste[CarteJoaillerie]
max_level_one : static int
max_level_two : static int
max_level_three : static int
getInstance() : CartesPyramide
CartesPyramide(deck1 : DeckNv1, deck2 : DeckNv2, deck3 : DeckNv3)
piocherCarte(niveau : uint) : const CarteJoaillerie
remplirPyramide(deck1 : DeckNv1, deck2 : DeckNv2, deck3 : DeckNv3)
isEmpty(level : int) : bool
getLevelCards(i : int) : vecteur[CarteJoaillerie]
1
1
1
Deck_Royal
instance : Deck_Royal
cards : vecteur[RoyalCard]
getInstance : Deck_Royal
getCards : vecteur[RoyalCard]
Deck_Royal()
addCardToDeck(card : RoyalCard) : void
deleteCard(pos : int) : void
Gère le deck des cartes royales
DeckNv2
instance : Deck_level_two
pioche : Liste[CarteJoaillerie]
Deck_level_two()
getInstance() : Deck_level_two
getPioche() : Liste[CarteJoaillerie]
addCardToDeck(card : CarteJoaillerie) : void
deleteFirstItem() : void
DeckNv3
instance : Deck_level_three
pioche : Liste[CarteJoaillerie]
Deck_level_three()
getInstance() : Deck_level_three
getPioche() : Liste[CarteJoaillerie]
addCardToDeck(card : CarteJoaillerie) : void
deleteFirstItem() : void
DeckNv1
instance : Deck_level_one
pioche : Liste[CarteJoaillerie]
Deck_level_one()
getInstance() : Deck_level_one
getPioche() : Liste[CarteJoaillerie]
addCardToDeck(card : CarteJoaillerie) : void
deleteFirstItem() : void
GestionPartie
Controller
jeu : Jeu
joueurEnCours : JoueurEnCours
TableDeJeu
plateau : Plateau
sac : SacJeton
cartes_pyramide : CartesPyramide
cartes_royales : Liste[CarteRoyale]
cartes_joaillerie : Liste[CarteJoaillerie]
DeckNv(1 à 3) : DeckNv(1 à 3)
1
1
est controllé
se joue dans
Jeu
TableDeJeu : TableDeJeu
joueur_courant : Joueur
tour : uint
1

Projet LO21

9 déc

Tâches

Nom	Date de début	Date de fin
Étude du Sujet *Analyser en profondeur les exigences du jeu Splendor Duel, les règles du jeu, et les fonctionnalités nécessaires.*	14/09/2023	06/10/2023
Shéma prévisionnel des classes et méthodes *Documenter les objets méthodes essentiels du projet, les fonctionnalités clés.*	21/09/2023	06/10/2023
Conception Initiale UML *Créer un diagramme UML initial pour représenter la structure et les relations des classes.*	28/09/2023	06/10/2023
UML 2 *Réviser et améliorer le diagramme UML en tenant compte des commentaires de la première version.*	09/10/2023	19/10/2023
Architecture approfondie *Réfléchir à l'architecture globale du logiciel.*	09/10/2023	19/10/2023
Implémentation initial *Commencer/continuer à coder les classes et les fonctionnalités de base du jeu Splendor Duel en C++.*	20/10/2023	02/11/2023
Finalisation de la première implémentation *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.*	03/11/2023	01/12/2023
Conception des images *Créer les graphismes et les images nécessaires pour le plateau, les cartes et les jetons du jeu.*	01/12/2023	12/12/2023
Conception interface graphique avec Qt *Créer une conception graphique pour l'interface utilisateur du jeu Splendor Duel.*	25/11/2023	16/12/2023
Sauvegarde *Création du système de sauvegarde d'une partie en cours.*	05/12/2023	20/12/2023
Implémentation interface *Intégrer l'interface utilisateur avec le code du jeu et s'assurer que tout fonctionne de manière fluide avec Qt, en C++.*	10/12/2023	20/12/2023
Finalisation *Finalisation du projet : batteries de tests en plus des tests existants, étude finale des fuites de mémoire, etc...*	07/12/2023	23/12/2023