

Projet LO21 : Splendor Duel

Rapport Final



Groupe 4 : Theo Guegan, Samuel Manchajm, Robert Antaluca, Samuel Beziat, Maxime Gautrot

Table des matières

I. Introduction.....	3
II. Résumé du projet.....	4
Liste des tâches.....	4
Contribution des membres.....	6
Retour sur les fonctionnalités attendues.....	6
Console.....	6
Interface.....	6
Utilisation de l'application.....	6
III. Architecture.....	7
UML.....	7
UML Général Final.....	7
UML Simplifié des interactions.....	8
Entités Fonctionnelles :.....	9
Choix du design pattern MVC.....	9
IA avec Strategy.....	9
Pattern builder pour la partie.....	10
Boucle de jeu.....	12
Sauvegarde.....	13
Graphisme.....	14
Objectifs :.....	14
Plateau et ses jetons :.....	14
Pyramide et Cartes.....	15
Page du menu principal, page de création de partie et page de jeu.....	16
Relier Front et Back.....	17
IV. Évolution de l'application.....	18
IA évolutive.....	18
Graphisme évolutif.....	18
Attributs évolutifs.....	19
Règles évolutives.....	19
Résumé.....	19
V. Cohésion de groupe.....	20
VI. Conclusion et analyse Critique.....	20
Bilan Général (améliorations).....	20
Bilans personnels.....	21

I. Introduction

Dans le cadre de l'UV LO21, Programmation et conception orientées objet, nous avons conçu et développé une application permettant de jouer au jeu de société Splendor Duel créé par Marc André et Bruno Cathala, et édité par SPACE Cowboys.

À tour de rôle, les joueurs prennent des jetons sur le plateau ou s'en servent pour acheter des cartes Joaillerie qui leur donnent les points de prestige et les couronnes nécessaires à la victoire. Les cartes offrent aussi des capacités spéciales et des bonus qui réduisent le coût des cartes Joaillerie suivantes. Pour l'emporter, il faudra remplir l'une des trois conditions indiquées sur la tuile Victoire.

Ce projet a pour but pendant tout un semestre de se familiariser à la réalisation d'un projet de génie logiciel. Que cela passe par la phase de conception de l'architecture, de développement du back-end ou la réalisation de l'interface. L'ensemble de ce projet est réalisé en C++ et à l'aide de QT pour la partie graphique.

Nous tenons à remercier Antoine Jouglet Responsable de l'UV LO21 pour ses enseignements et tout particulièrement Loïc Adam, notre chargé de TD et de projet, pour ses précieux conseils tout au long de la réalisation de l'application. Nous espérons que notre résultat vous plaira !

II. Résumé du projet

Liste des tâches

Tâche	État	Fichiers associés	Responsable	Temps	
Liste des attentes	Terminé		Samuel B	1h	Période 1
Liste des contraintes	Terminé		Robert	1h	
Mécanique de Jeu	Terminé	regle.docx	Théo, Samuel B	2h	
Recherche Design Pattern	Terminé	design pattern.docx	Théo, Samuel M	2h	
UML V1	Terminé	splendorduel.pl antuml	Samuel M	1h	
UML V2	Terminé	splendorduel.pl antuml	Samuel M	1h	
Sac de Jetons	Terminé	jeton.hpp, jeton.cpp	Théo	4h	Période 2
Plateau	Terminé	jeton.cpp, jeton.hpp	Théo	6h	
Gestion des Cartes	Terminé	carte.hpp, carte.cpp	Samuel B	10h	
Gestion des Joueurs	Terminé	joueur.hpp joueur.cpp	Samuel M	8h	
Implémentation des actions d'un joueur	Terminé	joueur.hpp joueur.cpp	Samuel M	8h	
Classe Partie	Terminé	partie.cpp partie.hpp	Robert	4h	
Classe Espace de Jeux	Terminé	espacedejeux.c pp espacedejeux.h	Robert	4h	
Classe Pyramide	Terminé	espacedejeux.c pp espacedejeux.h	Robert	4h	
Graphisme	Terminé		Maxime	+25h	

Tâche	État	Fichiers associés	Responsable	Temps	
Sauvegarde et restitution de partie	Terminé		Theo, Samuel B	10h	Période 3
Builder partie	Terminé	partie.h partie.cpp	Samuel B	4h	
Strategy / IA	Terminé	strategy.cpp strategy.hpp	Samuel M	4h	
Capacite	Terminé	controller.cpp controller.hpp	Samuel M	3h	
Modifs Espace de jeux	Terminé	espacejeux.cpp espacejeux.hpp	Robert	3h	
Mécanique de tour	Terminé	partie.cpp, partie.hpp	Samuel M, Robert	4h	
Boucle de Jeux	Terminé	main.cpp -> controller.cpp	Robert	3h	
Implémentation MVC	Terminé	controller.cpp controller.hpp	Samuel B, Théo	10h	
Refactoring Strategy	Terminé	controller.cpp	Théo	5h	
Modification du back pour le front	Terminé		Tout le monde	10h	
Gestion des affichages console	Terminé		Théo	4h	
Débogage	Terminé		Tout le monde (merci Samuel B et Théo)	45h	
Journée debug	Terminé	8h chacun		40h	
Relier Back Front	Terminé		Tout le monde	25h	
Nuit debug front	Terminé	10h chacun	Ct sympa	50h	
Réunions	Terminé			25h	
Rush final	Terminé	Chacun a fait	de son mieux		
Rapport final	Terminé		Samuel M	5h	

Contribution des membres

	Samuel B	Théo	Robert	Maxime	Samuel M
Nombre d'heures de travail	~80h	~75h	~65h	~75h	~65h

Retour sur les fonctionnalités attendues

Cette partie reprend les fonctionnalités attendues que nous avons dégagées du sujet et expliquées dans le premier rapport. Ceci permet d'avoir une vue d'ensemble sur le travail réalisé.

Console

- Deux joueurs doivent pouvoir s'affronter → Implémenté
- Joueur humain ou IA → Implémenté avec le Design Pattern Strategy
- Possibilité de reprendre la dernière partie → Implémenté avec les sauvegardes en SQLite et le design pattern Builder pour construire une ancienne partie.
- Vérificateurs de victoire et de validité des coups → Implémenté avec la boucle de jeu et le Controller
- Jeu fonctionnel → Implémenté, que cela soit en console ou en graphique

Interface

- Paramétrer une partie (nom joueur, niveau IA, règles spécifiques ? etc...) → Implémenté
- Demande de confirmation des coups → Implémenté
- Bibliothèques de joueurs (humain/IA) + statistiques pour chacun → Implémenté

Utilisation de l'application

Pour pouvoir utiliser l'application émulant le jeu Splendor Duel, il suffit de :

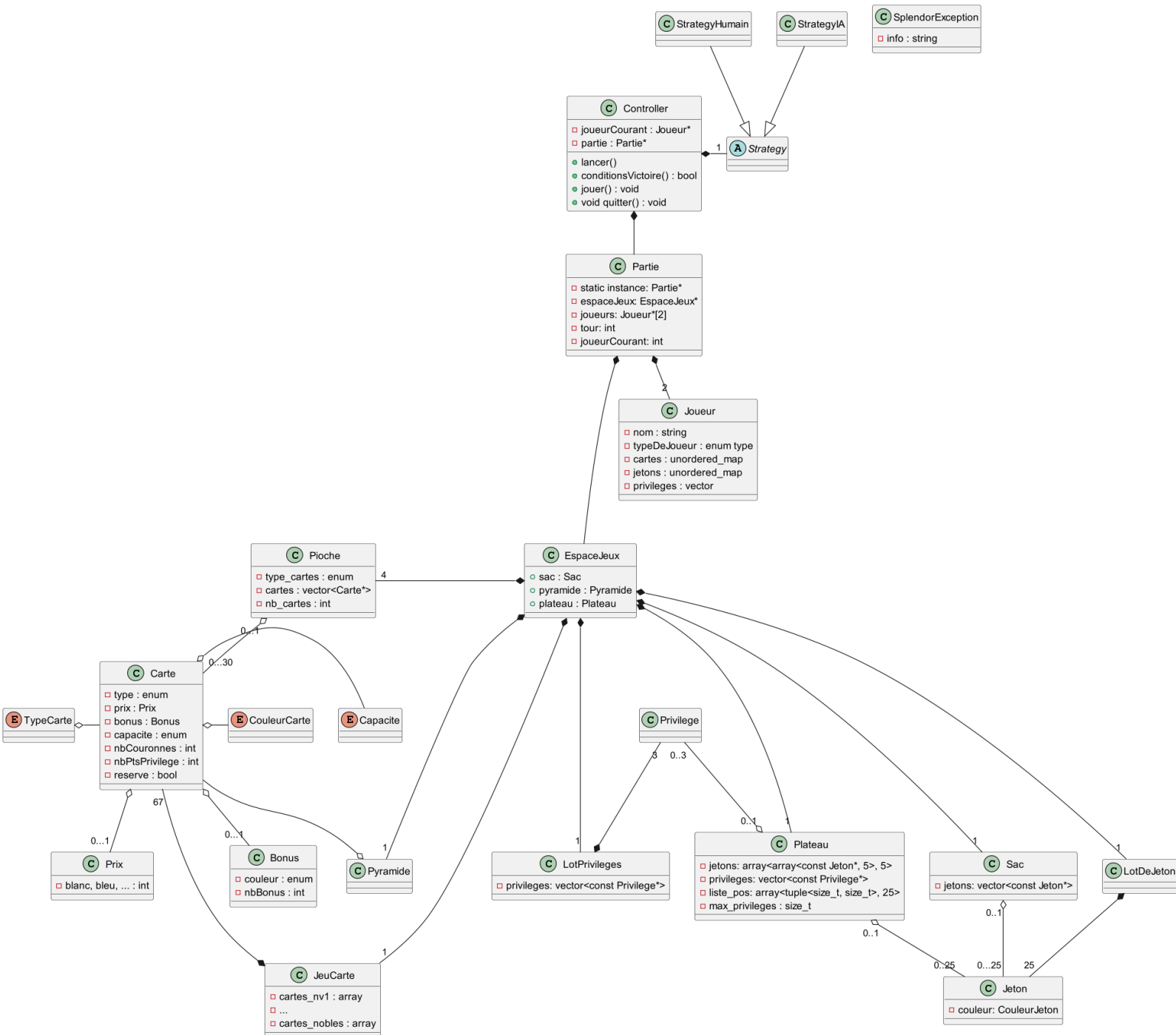
1. Cloner le repository : https://github.com/thequega/Projet_LO21.git
2. Ouvrir le projet dans QtCreator en désactivant le Shadow Build (Projets->Compiler->Décocher Shadow Build)
3. Compiler (Pour mac, définir le répertoire de travail dans "source")
4. Dans le terminal, choisir si l'on veut jouer en mode console ou en mode graphique
5. Bonne partie !

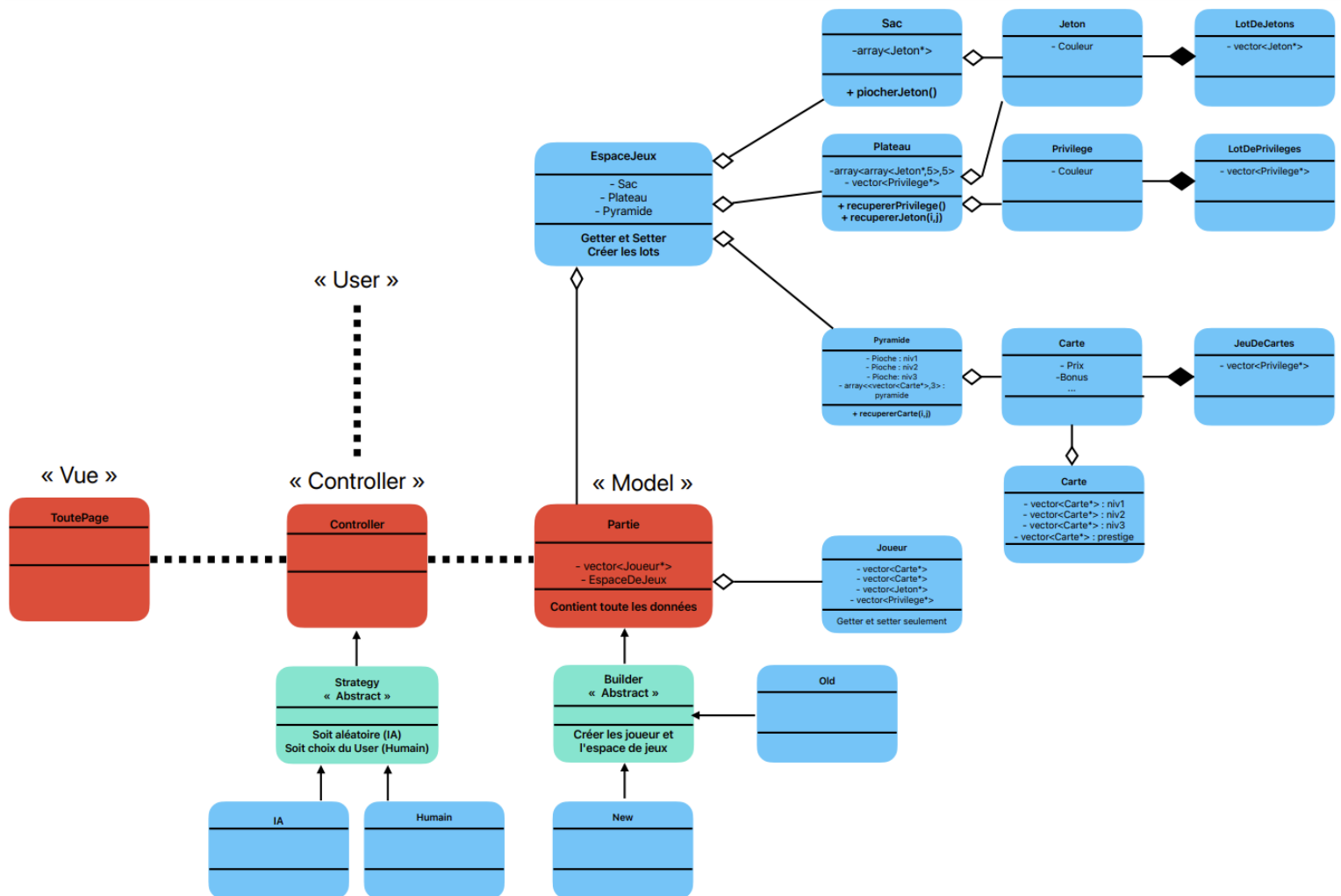
III. Architecture

UML

UML Général Final

Voici la dernière version de l'architecture de notre application. Les choix principaux d'architecture sont expliqués dans les parties suivantes. Cette UML ne prend pas en compte la partie graphique du projet.



UML Simplifié des interactions

Dans ce schéma simplifié de notre architecture, nous pouvons voir comment le **controller** sert de "hub" entre toutes les parties de notre application et permet donc de respecter SOLID.

Entités Fonctionnelles :

Cette partie explique les éléments et les choix d'architecture principaux. Il est impossible de formuler l'ensemble des choix d'architecture en restant succinct. Nous avons donc choisi de seulement parler des Design Patterns principaux utilisés pour rendre notre architecture stable et extensible.

Choix du design pattern MVC

Comme on peut le voir sur le schéma précédent, il a été choisi d'utiliser le pattern MVC (Modèle-Vue-Contrôleur) pour notre architecture finale. C'est au moment de devoir lier toutes les parties du jeu pour gérer les actions et les entrées utilisateurs que nous avons réfléchi et sommes tombés d'accord sur l'utilisation du design pattern MVC. En effet, celui-ci est couramment utilisé pour la modélisation d'application, notamment lorsqu'il y a besoin de traiter des demandes de l'utilisateur et de réagir en conséquence. Il permet de séparer en 3 couches distinctes l'application :

- La Vue : ce sont toutes les composantes graphiques, avec lesquelles l'utilisateur va interagir, et qui va donc transférer des informations au **Controller**.
- Le Modèle : c'est la couche des données de notre application. Données ici ne veut pas seulement dire base de données. Par exemple, dans le cadre du jeu, les instances de la classe '**Joueur**' font partie du modèle, car elles ne sont que des entités qui possèdent des cartes, des jetons, etc. C'est une brique de base du jeu, mais qui ne prend pas de décision. Idem pour le plateau, les cartes, etc. (il faut donc bien différencier l'utilisateur, de notre classe '**Joueur**', qui elle, n'est qu'un support qui contient certaines données). Les classes de cette couche devraient donc seulement avoir des méthodes getters, setters et des méthodes de mise-à-jour.
- Le **Controller** : c'est le chef d'orchestre qui va gérer les entrées utilisateur. Il va vérifier la validité de celles-ci, et en fonction va soit renvoyer une exception, ou bien appeler les méthodes du modèle pour mettre à jour celui-ci. C'est donc dans cette partie que nous avons finalement choisi de mettre la gestion des actions des joueurs (ou des utilisateurs plutôt). La classe Joueur ne prend donc en charge aucune action.

IA avec Strategy

Pour implémenter les joueurs humains et les IA, nous avons choisi le design pattern Strategy qui permet à partir de la classe abstraite **Strategy** de définir différents comportements en fonction du type de joueur. Cela nous permet de définir autant de types de joueurs que voulut sans modifier la classe **Joueur**. Ainsi, si nous voulons rajouter un niveau d'IA, nous avons juste besoin de rajouter une Classe fille de la classe Strategy.

En implémentant le MVC il nous a paru logique de déplacer l'ensemble des actions dans le **controller**. En effet, comme énoncé précédemment, le contrôleur va gérer les entrées utilisateurs, il va vérifier la validité de celles-ci, et en fonction va soit renvoyer une exception, ou bien appeler les méthodes du modèle pour mettre à jour celui-ci.

Finalement, ces entrées utilisateur sont surtout présentes dans la boucle de jeu et dans les actions d'un joueur, donc il paraît cohérent de placer les actions dans **Controller**.

Dans un même temps, nous avons décidé de complètement modifier notre conception de **strategy**, pour rendre le débogage bien plus facile. Au lieu de décliner des affichages différents et d'autres spécificités dans les différentes classes de **Strategy**. Il a été décidé de tout rassembler en une fois dans le **controller** avec les mêmes affichages pour une IA ou un Humain. Les seules parties variables de ces actions sont donc les entrées utilisateurs qui sont soit un cin ou le renvoi d'un nombre aléatoire. Grâce à cette implémentation, nous n'avons plus besoin de créer de nouvelles méthodes de strategy pour chaque action, mais seulement en fonction des types de retour que nous voulons. Par exemple, dans notre implémentation, nous avons uniquement une méthode *unsigned int choix_min_max(unsigned int min, unsigned int j)* qui permet de renvoyer soit un nombre aléatoire dans cet intervalle, soit de demander un nombre à l'utilisateur dans cet intervalle.

Pour la partie graphique, la strategy du joueur a petit à petit disparu pour laisser place à la totale liberté des actions du joueur en interagissant avec l'interface graphique. Cependant, la strategy de l'IA est quant à elle toujours présente et inchangée par rapport au mode console. On peut donc définir une nouvelle IA dans Strategy qui améliorerait à la fois l'IA de la partie console, mais également de la partie graphique en même temps.

Pattern builder pour la partie

Le problème qui s'est posé à nous était qu'en fonction de la nature de la partie (dernière partie sauvegardée ou nouvelle partie), la construction était assez différente. En effet, lorsque l'on crée une nouvelle partie, il y a juste à tout mettre "à vide" ou à zéro, la construction n'est pas très compliquée. En revanche, quand l'on veut créer une partie issue de la dernière sauvegarde, cela est bien plus compliqué et lourd (en termes de code), car il faut récupérer des données, et tout remettre en place et à la bonne place. Surcharger le constructeur n'aurait pas été une bonne option, puisque la construction d'une ancienne partie est composée de plusieurs étapes toutes conséquentes.

En cherchant un peu, nous avons décidé d'appliquer le pattern builder à notre classe **partie**. Ce design pattern nous permet alors de bien segmenter les différentes étapes de la construction d'une partie, et de réaliser des constructeurs différents en fonction de la nature de la partie. Voici donc l'architecture qui a été retenue :



On récupère ainsi une instance de partie selon le choix du **builder** approprié, et ce, via la classe **Director**.

L'implémentation de ce pattern a eu pour conséquence d'écarter l'implémentation de **Partie** en tant que singleton. Ce n'est pas très dommageable, mais si à l'avenir, on tient vraiment à ce qu'une partie soit unique, il faudra y réfléchir pour mêler Singleton et Builder.

Boucle de jeu

Pour la version du jeu en terminal, la boucle de jeu se base sur une boucle while(1) qui tourne indéfiniment jusqu'à un return 0.

On décompose un tour en 4 états représentés par la variable *etat_tour* contenues dans un switch: les actions optionnelles, les actions obligatoires, les vérifications de fin de tour (victoire, jetons > 10 ...) et le changement de joueur ou la sortie de boucle.

Les actions sont quant à elles aussi organisées à l'aide de blocs switch, avec chaque case de ces blocs qui correspond à une action spécifique que le joueur peut choisir d'effectuer.

À chaque appel d'une méthode du controller, les try-catch sont utilisés pour intercepter les erreurs qui peuvent survenir lors des actions des joueurs, comme des mouvements illégaux ou des choix invalides, et lui permettre de recommencer.

Les variables *etat_action* et *etat_tour* contrôlent où le joueur en est dans ses actions et dans son tour, elles peuvent être mises à jour par un choix de l'utilisateur, un chiffre généré par l'IA ou lorsqu'une action s'est bien déroulée.

Pour ce qui est de la partie console, la boucle de jeu en tant que telle disparaît pour laisser place à la vue, en effet ici les actions sont totalement libres pour l'utilisateur et ce dernier doit pouvoir choisir par lui-même de faire une action optionnelle ou obligatoire. Le comportement de l'application doit ici s'adapter entièrement aux interactions du joueur pour permettre la meilleure expérience de jeu possible.

On retrouve cependant pour la partie IA une boucle de jeu très similaire à la boucle de jeux du monde console, en effet contrairement à un joueur une IA n'a pas d'interaction directe avec la partie graphique, c'est pourquoi nous avons définie une méthode *tour_IA()* dans notre contrôleur en reprenant la structure de la boucle de jeux en l'adaptant pour permettre à l'IA de jouer 1 tour.

Sauvegarde

Pour la gestion des données liée à la partie et aux scores, nous avons décidé d'utiliser SQLite de la même manière que nous stockons nos cartes. Dans notre **controller**, deux méthodes, à savoir *sauvegarderpartie()* et *enregisterscore()* nous permettent de push toutes les informations nécessaires à une restitution dans notre base de donnée.







Pour ce qui est de l'enregistrement des scores, nous stockons à chaque fin de partie dans une base de donnée score.sqlite les données suivantes :

id	pseudo	nbVictoire	nbDefaite
1	Alain telligence	3	1
2	AL Gorythme	0	3
3	WorsTayls	1	0

Pour cela, nous avons utilisé la bibliothèque SQL proposés par Qt, voici l'exemple d'une requête SQL :

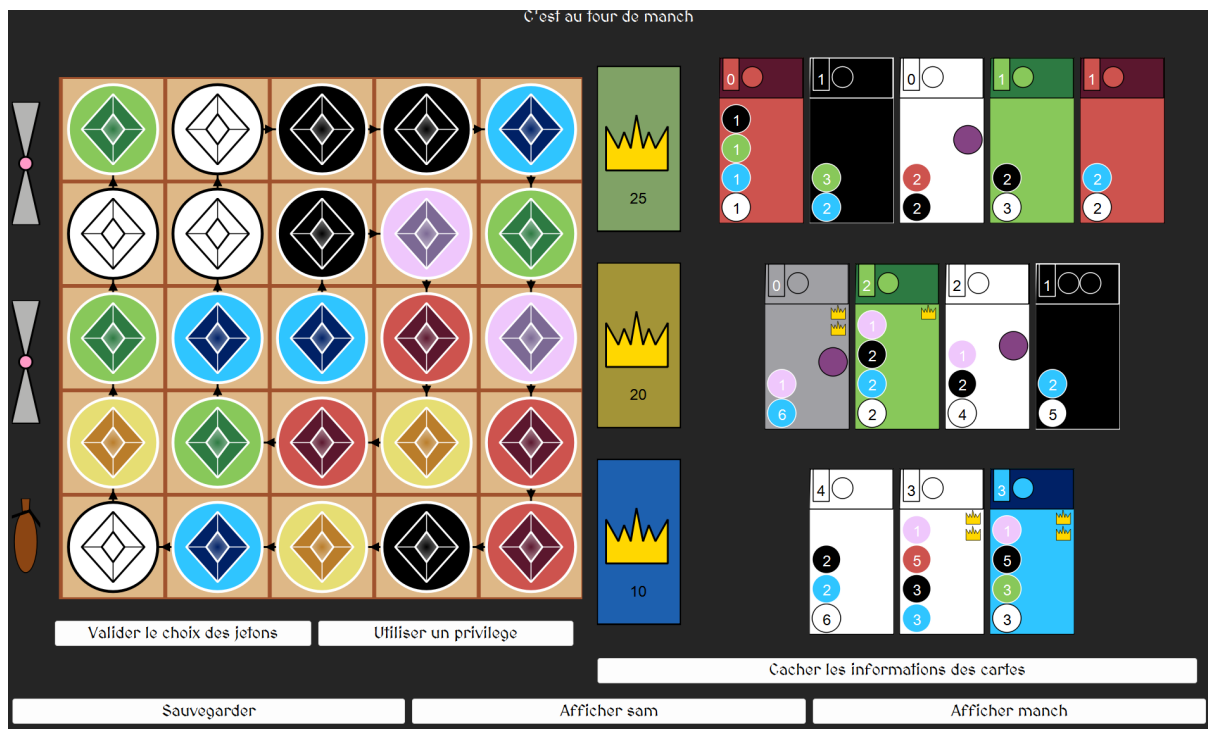
```
QString sql = "SELECT * FROM score WHERE pseudo = " +  
QString::fromStdString(getPartie().getJoueur(i)->getPseudo()) + "';";  
if (!query.exec(sql)) {  
    std::cerr << "Erreur lors de la recherche du joueur dans la base de donnee" << std::endl;  
    db.close();  
    return;  
}
```

Pour la gestion de la sauvegarde, nous utilisons différentes tables dans save.sqlite :

```
>  cartes_joueur  
>  infopartie  
>  jetons_joueur  
>  joueur  
>  plateau  
>  pyramide
```

Ces tables, nous permettant de stocker les cartes des joueurs avec une jointure sur l'id du joueur, mais également les infos de la partie actuelle, le pseudo des joueurs, les jetons du plateau, etc.

Graphisme



Objectifs :

L'objectif de la partie graphique est d'avoir quelque chose de fonctionnel, simple d'utilisation, modulable le plus possible, et si possible, esthétiquement jolie.

Pour ces différentes raisons, certains choix ont été faits :

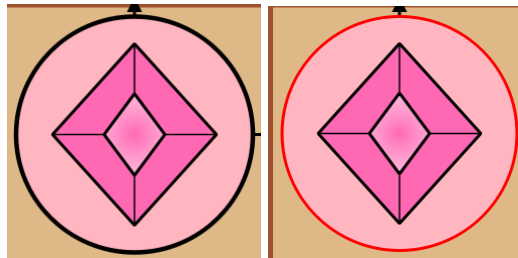
- Les objets appartenant aux joueurs (cartes achetées, jetons..) sont dans des widgets séparés que l'on peut afficher grâce à des boutons sur la page de jeu. Cela a pour but de simplifier au maximum la page de jeu pour ne pas la surcharger d'informations, en plus de pouvoir gérer de montrer ou non les cartes réservées selon quel joueur joue.
- Tous les objets seront dessinés vectoriellement, car il est bien plus simple de les modifier, d'en ajouter ou d'en retirer. Par exemple, dans le cas du plateau, le fait de le dessiner vectoriellement permet d'augmenter le nombre de Jetons via un simple entier et d'avoir l'affichage graphique qui suit.
- Tous les objets sont aussi adaptés par rapport à la taille de l'écran de l'utilisateur, pour éviter tout problème d'affichage selon les différentes machines qui pourraient lancer le jeu. (responsive)
- Les cartes seront retournables afin de pouvoir voir de façon plus simple les caractéristiques d'une carte.

Plateau et ses jetons :

Les Jetons (classe **vueJeton**) héritent de la classe **QPushButton**, cela permet de facilement connecter le fait de cliquer sur le jeton et les actions qui en résultent. On utilise aussi la classe **position** pour définir une position pour les jetons et ainsi simplifier le lien avec le back-end, mais aussi pouvoir bien définir où ils sont placés visuellement sur le plateau. Pour ce qui est de leur dessin, nous dessinons un cercle, ensuite, deux losanges avec un plus petit que l'autre que nous faisons rejoindre avec des traits, ensuite, nous ajoutons un gradient pour un effet lumineux.

Enfin, au départ, nous souhaitions mettre une croix sur les boutons validés (inspiré du TD sur le jeu Set). Cependant, Théo a eu l'excellente idée de plutôt changer la couleur du contour en rouge, ce qui est bien plus esthétique, mais aussi plus lisible.

À gauche un bouton non cliqué et à droite un bouton cliqué :



Pour faire le lien avec le back-end, **vueJeton** contient un pointeur de **Jeton** qui est le jeton auquel il est relié.

Pour le plateau (classe **vuePlateau**), nous souhaitions utiliser un **QGridLayout** pour placer les jetons sur le plateau. Celui-ci avait l'avantage de pouvoir augmenter ou réduire sa taille afin de contenir tous les jetons que l'on voulait et de pouvoir les placer et replacer simplement. Cependant, lors de la définition du *paintEvent()*, nous avons rencontré un problème selon les ordinateurs. En effet, lorsque nous utilisons des valeurs pour placer correctement les cases autour des jetons, le **QGridLayout** faisait un espacement entre les différentes colonnes, espacement qui ne pouvait être réglé par le *setSpacing()* et la redéfinition de la marge.

La solution a donc été de créer une nouvelle classe héritant de **QWidget**, **grilleJetons**. Elle se comporte presque comme un **QGridLayout** et place les jetons grâce à sa méthode *placerJetons()* en fonction de la position que contient chaque Jeton. De plus, **grilleJetons** contient des **QRect** ce qui permet de placer très simplement les jetons et ensuite de les colorier très simplement avec *drawRect()*. De plus, le plateau a aussi un bouton qui permet de valider les jetons sélectionnés.

Enfin, pour le dessin du plateau, on dessine en réalité **grilleJetons**. Pour ce faire, on va colorier chaque carré puis on ajoute, via le même algorithme que la matrice spirale, des traits qui relient chaque carré entre eux avec en leur centre un triangle isocèle dans la direction du trait que l'on vient de faire.

Pyramide et Cartes

Les Cartes (**vueCarte**) sont créées avec héritage de `QStackedWidget`. Cela permet de pouvoir passer de la classe **visuelCarte** et la carte **infoCarte**, afin d'avoir le visuel d'un côté et d'avoir les informations de celle-ci quand on passe dessus. Elles seront sûrement liées au back-end comme pour les jetons via un pointeur qui pointera sur son homologue dans le back.

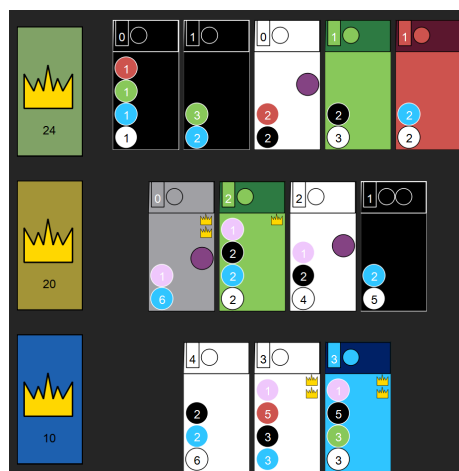
De plus, les actions de clique droit et gauche permettront de respectivement les acheter et les réserver. Elles ont aussi une position, pour permettre de bien les afficher au bon emplacement. Le `paintEvent()` est actuellement basique, car il ne contient qu'une couleur avec un triangle et un cercle, mais pourra à terme être fait très simplement en fonction de la carte pointée. Enfin, bien que pas encore implémenté, il y aura aussi les paquets sur la gauche et les cartes nobles en bas.

La pyramide, elle, marche comme un grand widget permettant d'accueillir toutes les cartes. Pour ce faire, on a un layout principal et on crée des layouts horizontaux que l'on inclut à l'intérieur des premiers et qui permettent ainsi de définir les différents niveaux. Enfin, en bas, nous ajoutons un bouton permettant de cacher les informations des cartes.

Page du menu principal, page de création de partie et page de jeu

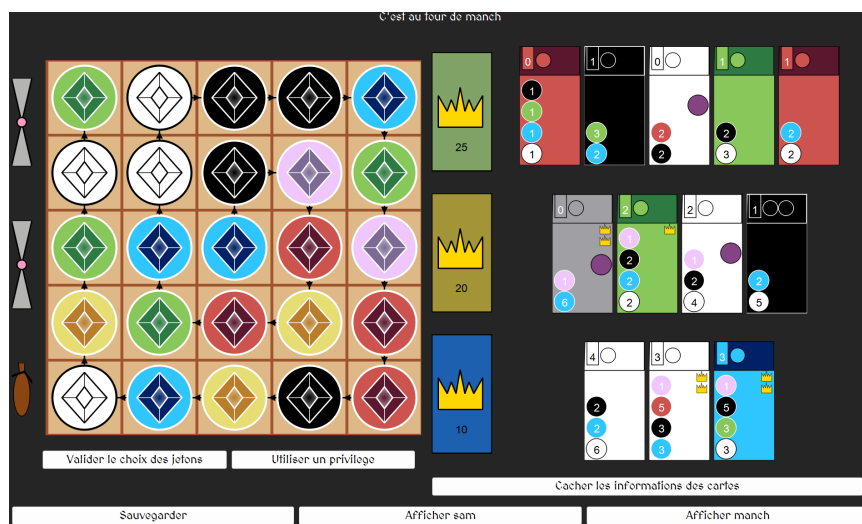
La page du menu est très simple, on a quatre boutons permettant respectivement de créer une nouvelle partie, accéder à la bibliothèque de joueur, accéder à la partie, sauvegarder et enfin quitter l'application.

La page de création est une page permettant de lier le menu principal avec la page de jeu. On y récupère des informations cruciales comme quel type de joueur et les pseudos des joueurs. Pour les récupérer, on utilise un mélange de `QComboBox` qui permettent de choisir des éléments dans un Scroll et de `QLineEdit` qui permettent de récupérer des choses qui sont écrites dedans. Tout ça est fait de façon que les `QLineEdit` disparaissent si on choisit une IA ou un joueur qui existe déjà.



Enfin, la page de jeu est un widget qui contient tous les éléments implémentés précédemment, donc le plateau (**vuePlateau**), la pyramide (**vuePyramide**), mais aussi les pages des joueurs (qui ne sont pour l'instant qu'un ensemble de layout pour placer les futurs éléments) et les boutons permettant d'afficher les pages.

Nous avons également redéfini la méthode *mousePressEvent()* qui s'active quand on clique sur la page et qui ferme tous les widgets autres que la page de jeu. En effet, nous avions des problèmes pour les widgets que l'on ouvrait depuis la page de jeu puisque lorsque l'on cliquait sur la page de jeu, elle passait en premier plan et comme nous n'avions pas fermé les autres widgets, on ne pouvait plus les faire réapparaître. Ainsi, maintenant, lorsqu'on clique tout se ferme, on peut donc les faire réapparaître. De plus, nous avons aussi redéfini la méthode pour quitter la page. Maintenant, lorsque l'on clique sur la croix pour quitter la page, un pop-up de validation apparaît pour être sûr que l'on veuille bien la fermer.



IV. Évolution de l'application

L'architecture mise en place permet facilement des évolutions. Durant toute la conception de cette application, nous nous sommes toujours demandés si nous utilisions la méthode d'implémentation qui serait la plus extensible par la suite. C'est donc pour cela que nous avons utilisé certains Designs Patterns ou utilisé certains types. Cette partie, sans revenir en détail sur l'architecture, va résumer les éléments principaux permettant l'évolutivité de l'application.

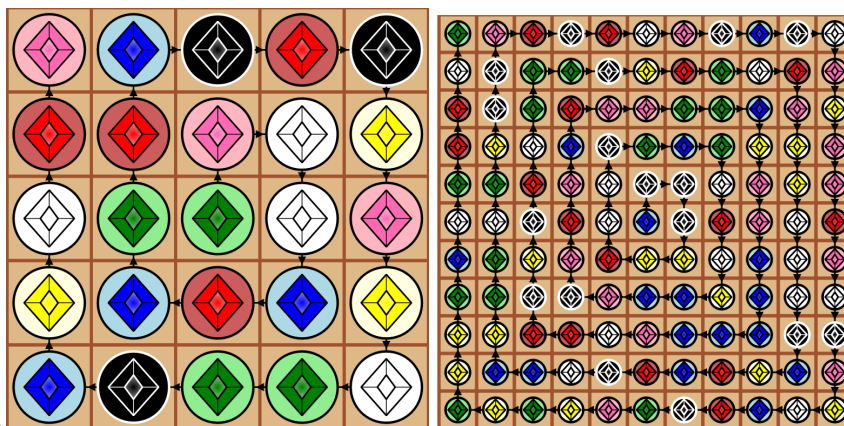
IA évolutive

Comme mentionné précédemment, l'utilisation du Design Pattern Strategy rend l'IA évolutive à souhait. En effet, par la suite, sans modifier notre premier niveau d'IA (aléatoire), il serait possible de créer une nouvelle classe fille correspondant à un deuxième niveau d'IA, tout cela sans changer les autres classes. Nous avons, par exemple, pensé à implémenter un Système expert d'ordre 0+ (coucou IA01).

Graphisme évolutif

Grâce à l'implémentation de tous les éléments graphiques en vectoriel, il est possible de modifier, ajouter tous les éléments en quelques lignes. Même si cela a pris plus de temps à réaliser, ne pas prendre d'éléments graphiques fixes (images), nous assure une évolutivité optimale.

Ci-dessous un exemple du plateau avec 25 Jetons et un exemple avec 121 Jetons :



Attributs évolutifs

Pour rendre l'ajout de nouvelles cartes et de nouveaux types de jetons le plus facile possible, nous avons décidé de stocker tous ces éléments dans des map. En effet, les map de jetons et de cartes sont construits de manière automatique dans le constructeur de **Joueur**, avec des couleurs qui s'ajoutent automatiquement. Grâce à ceci, si nous rajoutons un nouveau jeton dans la BDD avec une nouvelle couleur, alors celle-ci sera automatiquement prise en compte. De plus, cette nouvelle couleur sera également très simple à mettre en place dans l'interface avec l'évolutivité de celle-ci.

La plupart de nos conteneurs sont fait à partir de vecteur permettant une modification du nombre total très simple. Enfin, tous les objets SplendorDuel ont leur propre classe, même les privilèges, ce qui permet l'ajout d'attributs et de comportement très simple.

Règles évolutives

Un des objectifs principaux en termes d'évolutivité est de pouvoir ajouter de nouvelles règles et de nouvelles conditions de terminaisons. Cela est normalement assez simple. Par exemple, pour ajouter une nouvelle condition de victoire, il suffit d'ajouter un cas dans la méthode gagnant du Controller. Nous pourrions facilement rajouter qu'un joueur gagne s'il obtient 10 points de prestiges de plus que son adversaire.

L'implémentation du MVC favorise pleinement la facilité d'ajout de nouvelles règles en centralisant toutes les conditions dans le **Controller**.

Résumé

Voici un bref résumé des possibilités d'évolution attendues et comment elles sont respectées.

- ajout de nouvelles IA joueurs
 - Possible grâce au design Pattern *Strategy*
- ajout d'un nouveau type de capacité
 - possible simplement en ajoutant une valeur à l'enum *Capacite* et en ajoutant un cas dans la méthode *appliquerCapacite*
- ajout d'une nouvelle carte
 - Automatique dans le back avec la génération à partir du SGBD
- ajout ou remplacement d'une règle (et la vérification de son respect par l'application)
 - Possible grâce au MVC
- ajout d'une nouvelle condition de terminaison
 - Possible en modifiant la méthode gagnant du Controller
- éventuellement l'ajout d'éléments d'IHM de paramétrage liés aux nouveaux ajouts.

V. Cohésion de groupe

Après maintenant plusieurs mois à se supporter, la cohésion du groupe a toujours été très bonne au cours de ce projet. Nous avons, tout au long du projet, réalisé des réunions de deux heures hebdomadaires qui nous ont permis de mettre en commun le travail réalisé et de chacun apporter un regard critique sur notre travail personnel.

La dernière période, plus axée sur le debug est la partie graphique, fut challengeante en termes de coopération, car nous avons dû tous travailler en collaboration sur les mêmes classes. Nous avons donc dû être très rigoureux dans notre communication, mais aussi au niveau de git pour créer le moins de conflits possible.

Le groupe s'entend donc toujours aussi bien et nous savons également bien communiquer pour se répartir les tâches en fonction du travail de chacun. De plus, le groupe a vraiment un niveau homogène, ce qui facilite vraiment la réalisation de ce beau projet !

VI. Conclusion et analyse Critique

Bilan Général (améliorations)

Actuellement l'application ne couvre pas 100% des fonctionnalités du Jeu. En effet, faute de plus de temps, nous n'avons pas pu implémenter quelques détails du fonctionnement du jeu et certains aspects de l'implémentation pourraient être revus. Voici une liste de ce qui resterait à faire :

- Gestion de la deuxième capacité des cartes (en mode graphique seulement) : la deuxième capacité n'existe que pour une seule carte dans tout le jeu, et c'est un peu embêtant à implémenter.
- En mode graphique, on suppose que quand le joueur est censé faire une action "forcée" (ex : récupérer une carte noble), il ne ferme pas la fenêtre qui apparaît et ne passe pas à autre chose. Ainsi, nous ne couvrons pas toutes les possibles actions de l'utilisateur sur l'application.
- Il peut bien sûr rester des bugs qui nous ont échappé.
- Au niveau du code, certaines relations de friends (entre le controller et le joueur par exemple) ne sont pas forcément justifiées et mériteraient donc d'être changées.

Excepté cela, nous avons implémenté 99% des fonctionnalités présentes dans SplendorDuel et de manière a priori fonctionnelle.

Nous sommes fortement satisfaits de l'ensemble des fonctionnalités que nous avons implémentées. Au cours de ce projet, le but a toujours été de prioriser la modularité de notre application, nous avons toujours essayé d'implémenter les fonctionnalités de la

meilleure des manières. Cela peut par exemple se voir sur notre interface où nous aurions pu faire quelque chose de bien plus simple. Néanmoins, nous nous sommes toujours questionnés sur la facilité d'utilisation de notre jeu en prenant la place des utilisateurs.

Nous sommes donc fiers de finalement avoir une application 100% fonctionnelle est agréable d'utilisation pour tous les utilisateurs !

Bilans personnels

Samuel B

C'est le meilleur projet qu'il m'ait été donné de réaliser durant tout mon cursus UTC. Pour moi qui n'avait jamais réalisé de vrai développement informatique un petit peu conséquent, ça n'a été que du positif, allant ainsi de l'assimilation des concepts de POO abordés dans le cours, à la familiarisation avec des outils basiques comme git. Mais mon ressenti serait sans doute différent si je n'avais pas été dans cet excellent groupe avec qui j'ai travaillé. Ce fut un plaisir jusqu'à la dernière heure, et à ce titre, je les remercie tous pour leur investissement, et ce sera avec plaisir que je retravaillerai avec eux par la suite.

Ce projet m'a certes pris énormément de temps, mais je pense que cela en valait la peine. Enfin, merci à Loïc Adam pour son suivi et la bienveillance apportée durant les différents jalons.

Samuel M

Pour ma part, les attentes sur ce projet de LO21 ont toutes été remplies. En effet, comme en pâtit sa réputation, ce projet est très chronophage et demande beaucoup d'investissement. Néanmoins, celui-ci fut, pour l'instant, le projet le plus formateur de ma scolarité. En effet, que cela soit bien-sûr sur le côté programmation et technique. Mais celui-ci fut surtout instructif par son axe gestion de projet, où nous avons dû apprendre en autonomie à mener un conséquent projet jusqu'au bout. J'aimerais par ailleurs remercier l'ensemble des membres du groupe, avec lesquels nous nous sommes tous extrêmement bien entendus.

Théo

Le projet est sans aucun doute le projet le plus chronophage qui m'a été demandé dans toute ma scolarité, lié au fait que j'ai personnellement décidé d'y accorder énormément de temps, poussé par un réel souhait de rendre une application propre. Ce genre de projet n'est jamais réellement terminé, on a toujours envie de compléter, d'améliorer, d'ajouter de nouvelles fonctionnalités. Je ne regrette cependant en rien le temps passé sur ce projet, car c'est selon moi également le projet le plus formateur que j'ai eu. Au début du semestre, je n'avais aucune idée de comment conduire un projet informatique, comment collaborer en utilisant git, comment définir une architecture et en voyant tout le travail accompli après un semestre, je trouve que les progrès sont impressionnants. Je suis donc très heureux d'avoir pu réaliser ce projet, surtout dans un groupe dans lequel la bonne humeur a toujours été au rendez-vous.

Maxime

Ce projet était mon premier projet de groupe. Et quel projet ! Bien que cela fut très compliqué de s'habituer au début et de trouver le temps de travailler en groupe, j'ai adoré l'expérience. Que ce soit la compréhension de nouveaux outils comme github, Qt, le C++ de manière générale, je sens que ce projet a bien amélioré mes compétences en tant qu'informaticien. Je me suis aussi découvert une passion pour le front avec le fait de faire des interfaces graphiques. La création de signaux, ainsi que l'organisation des différents widgets et surtout le dessin vectoriel qui, à force, commençait à être un peu chronophage. Finalement, je pense que ce que j'ai le plus aimé durant ce projet, c'est le fait d'avoir changé de groupe en début de semestre, car j'ai le groupe dans lequel je suis est vraiment super. En commençant, je n'étais sûrement pas le meilleur programmeur, et je pense que je ne le suis toujours pas, mais les personnes du groupe m'ont super bien intégré et m'ont vraiment aidé durant ce semestre pour LO21. Merci les gars !

Robert

Ce projet de LO21 a été une expérience incroyablement enrichissante. Malgré le temps et l'investissement considérables nécessaires, il a largement dépassé mes attentes. C'était sans aucun doute l'un des projets les plus concrets et intéressants auquel j'ai pu participer, de par le défi de programmation et technique, sa durée, la variété des missions mais aussi la gestion de projet en autonomie.

Un grand merci à tous les membres du groupe pour notre collaboration harmonieuse et efficace et pour les amitiés créées. Je tiens aussi à remercier Mr. Loïc Adam pour ses conseils et son encadrement.