Rapport final
Projet Splendor Duel

LO21 - A23

Présenté par :

Aubin Vert
Léopold Chappuis
Gaspard Petri
Alexandru Ghitu
Quentin Fouinat-Beal

Sommaire

Remerciements

Nous souhaitons commencer ce rapport en remerciant Antoine Jouglet ainsi que toute l'équipe pédagogique de LO21 pour leur accompagnement et leurs conseils qui nous auront été précieux.

Introduction

Dans le cadre de notre cours de LO21, nous avons été missionnés pour créer une version numérique du jeu Splendor Duel, créé par Marc André et Bruno Cathala, et édité par Space Cowboy. Pour cela, nous devions utiliser le langage C++ et employer une approche orientée objet.

Pour mener à bien ce projet, nous avons dû mobiliser les différentes connaissances acquises en cours ainsi qu'en TD. De plus, un tel projet nous a permis d'apprendre à gérer un projet du début jusqu'à la fin et ce en travaillant en équipe. Ainsi, au cours de cette expérience, nous devions faire preuve d'esprit d'équipe, de rigueur, d'anticipation, d'adaptabilité et de polyvalence face aux différentes tâches que nous devrons traiter.

Lien du gitlab : https://gitlab.utc.fr/lo21-td3-a23/lo21-splendor-duel

Possibilités de l'application

- Notre application possède plusieurs modes de jeu, grâce à deux types de joueurs : IA ou humain. L'application permet des parties aussi bien en humain contre humain, IA contre humain ou IA contre IA. Au début d'une partie, les noms des joueurs (humain/IA) sont paramétrables. Cependant faute de temps, nous avons simplement implémenté une IA aléatoire, il n'y a donc pas de paramétrage possible pour le niveau d'adversité de l'IA. A noter qu'il est tout à fait possible et simple d'implémenter une nouvelle IA plus développée grâce au Design Pattern Strategy_Player (détaillé plus loin dans ce rapport) et donc de paramétrer un niveau d'adversité de l'IA au début du jeu.

- Il est possible de procéder à l'abandon ou à la reprise d'une éventuelle partie en cours. La reprise d'une partie est proposée au début du jeu et permet de reprendre la dernière partie jouée si elle n'a pas été finie. Pour abandonner une partie en cours, il suffit de stopper le programme.

- Durant la partie, il est demandé à l'utilisateur (si c'est un humain) de valider ou non son choix d'action, et ce à chaque étape du tour. Si le choix n'est pas validé, l'action s'annule.

- Un historique des scores est disponible et permet de sauvegarder pour chaque partie les caractéristiques suivantes : le score des deux joueurs, le joueur perdant et le joueur gagnant avec pour chacun le type de joueur, son nom, son nombre de parties jouées et son nombre de parties gagnées. Le nombre de matchs disputés est également sauvegardé.

- L'application permet également de jouer avec ou sans interface graphique, en fonction du main lancé. En effet le projet a été configuré de sorte à ce que les deux "versions" du projet (console et Qt) puissent coexister.

Architecture

Contexte

L'architecture (UML en annexe) a été pensée, dans un premier temps, après une analyse générale du jeu Splendor Duel. Nous nous sommes mis d'accord sur les classes, puis nous avons attaqué le développement de ces dernières après une répartition des tâches. Cependant, nous nous sommes vite rendu compte de certains problèmes. Par exemple, au début, nous avons pensé de faire une classe mère Jeton puis de faire hériter chaque type de jeton de cette classe, avec sa couleur. Après le début du développement nous avons décidé de créer juste une seule classe qui aura comme attribut le type de jeton (une énumération), cela a rendu le développement beaucoup plus simple. De plus, après avoir été familiarisé sur la partie du cours sur le transtypage, nous avons compris que rester avec la première idée aurait imposé à un moment donné l'utilisation des dynamic_cast afin de faire la différence entre les jetons. Ce problème a été évité avec la nouvelle solution. Un autre aspect est la maintenabilité du code: nous avons essayé de faire le maximum afin de le rendre le plus maintenable possible. Notamment, aux endroits où cela a été possible, nous avons ajouté des attributs de type 'max', comme par exemple le nombre maximal de jetons ou de cartes afin de rendre possible d' éventuelles mises à jour dans le futur.

La création d'une classe dédiée à chaque objet dans notre projet, accompagnée de l'utilisation intensive de vecteurs pour stocker les données, a été une décision

stratégique visant à simplifier le développement et à rendre le code plus robuste. En exploitant les fonctionnalités intégrées des vecteurs, telles que les itérateurs, l'ajout et la suppression d'éléments, ainsi que la recherche, nous avons pu bénéficier d'une flexibilité accrue dans la manipulation des données au sein de chaque classe.

Cependant, la création de classes individuelles ne représentait qu'une partie du défi. La véritable difficulté résidait dans l'établissement de liens cohérents entre ces classes pour assurer un fonctionnement harmonieux du jeu. Cette interconnexion a exigé une réflexion approfondie sur les dépendances entre les objets du jeu, garantissant que chaque action effectuée dans une classe impacte de manière appropriée les autres composants du système. La coordination de ces interactions a demandé une attention particulière pour éviter les erreurs potentielles et garantir la cohérence globale du jeu.

Nous allons à présent détailler, pour chaque classe, certaines décisions d'architecture.

Modules (classes)

Jeu

La classe Jeu joue un rôle central dans notre développement, agissant en tant que contrôleur principal du jeu. Elle assume la responsabilité de l'initialisation et de la destruction de tous les éléments du jeu, tels que les joueurs, les cartes, les jetons, et autres. En conséquence, nous avons pris la décision d'appliquer le Design Pattern Singleton à cette classe, garantissant ainsi qu'il n'existe qu'une seule instance de cet objet dans le jeu.
Cette classe englobe également des méthodes essentielles pour exécuter les "grandes actions" du jeu, telles que le remplissage du plateau. De plus, elle est chargée de contrôler l'état global du jeu, coordonnant les tours des joueurs et mettant fin à la partie si l'un d'entre eux remporte la victoire.

Remarque:

La responsabilité du jeu s'étend à la construction et à la destruction de la plupart des objets, mais une particularité intervient lors de la destruction des instances de joueur. Dans ce contexte, seuls les privilèges et les cartes royales sont également détruits, simultanément avec la libération de la main du joueur. Cette décision découle du fait qu'une fois qu'un joueur a acquis une carte ou un privilège,

celui-ci est retiré du jeu pour éviter toute duplication et préserver l'intégrité de la composition globale.

Strategy_player (Joueur) : l'humain et l'IA

Le module Strategy_player représente les participants au jeu. Chaque instance de cette classe encapsule les informations spécifiques à un participant, telles que son nom, le nombre de points accumulés, le nombre de couronnes, le nombre de cartes de joaillerie et royales acquises, le nombre de jetons, et le nombre de privilèges détenus. Certains de ces attributs sont soumis à des limites maximales définies par des constantes statiques, reflétant les contraintes du jeu.

Les collections de cartes de joaillerie achetées, réservées et royales sont gérées à l'aide de std::vector. De même, les jetons, privilèges et autres agrégats de données sont également stockés et gérés de la même manière. L'utilisation des vecteurs nous a fait gagner beaucoup de temps et nous a permis de garantir une certaine robustesse de notre code.

La classe Strategy_player expose des méthodes communes aux joueurs et IA: accès et modification d'attributs, remplissage du plateau, etc.

Les classes filles donnent les méthodes nécessitant une adaptation en fonction du cas d'utilisation (joueur ou IA). En effet, la plupart de ces méthodes sont sensiblement équivalentes. Cependant, les entrées au clavier d'un humain sont remplacées par des choix aléatoires pour l'IA.

L'utilisation du Design Pattern strategy permet de garantir un ajout ultérieur d'autres IA de façon simple et en ne remettant pas en cause le fonctionnement des types joueur et IA. L'ajout d'un nouveau type d'IA ou de joueur ne nécessiterait qu'un développement d'une classe fille de Strategy_player qui serait adapté à l'utilisation voulue pour ce nouveau type de participant.

Au début de la partie, nous avons choisi d'empêcher les joueurs de rentrer un nom vide. Cela permet non seulement d'éviter un manque de clarté d'affichage, mais surtout des statistiques erronées dans l'historique avec un nom par défaut qui apparaît bien plus souvent.

Remarque:

Après avoir développé l'IA, nous avons pu tester le programme principal (main) en faisant jouer une IA contre une autre IA. L'intérêt de cette manipulation a été de vérifier le bon fonctionnement d'une partie dans son intégralité et d'assurer que le programme ne tombe pas dans des états particuliers que nous n'aurions pas suspecté en ayant joué nous mêmes. Par exemple, nous nous sommes rendus compte que nous n'avions pas géré le cas dans lequel un joueur ou une IA voulait piocher un jeton sur le plateau alors qu'il ne restait plus que des jetons or et que le joueur ne pouvait plus réserver de carte (donc ne peut pas piocher de jetons or), ce qui provoquait une boucle infinie. De nombreux cas similaires à celui-ci ont pu être identifiés grâce à cette 'simulation'. Ainsi, le développement de l'IA nous a permis d'apporter plus de robustesse à notre programme principal.

Carte

Pour représenter les cartes du jeu, nous avons une classe mère Card et 2 classes filles RoyalCard et JewelryCard. L'idée de l'héritage permet de conserver les propriétés communes aux 2 classes filles afin de ne pas avoir à dupliquer le code et à respecter le principe SOLID.
Pour être sûrs d'éviter tout problème de duplication des cartes, nous avons décidé d'implémenter un attribut statique qui limite le nombre maximum d'objets Carte créés.
Pour ce qui est des capacités, nous avons choisi de les implémenter au sein d'un *enum class*, pour s'assurer que seules les capacités définies dans le jeu puissent exister.

JewelryCard

La classe JewelryCard représente les cartes joaillerie du jeu. Ces cartes sont précisément au nombre de 67 dans le jeu, c'est pourquoi nous avons encore une fois créé un attribut statique qui empêche d'en créer plus.
En plus de la structure héritée de Carte, celles-ci possèdent un coût en jetons, des couronnes ainsi qu'un potentiel bonus sur une couleur donnée. Le type de l'attribut représentant ce bonus est défini dans un *enum class* de la même manière que les capacités, afin d'éviter l'apparition dans le jeu de bonus de couleurs inexistantes. De plus, cet attribut, que l'on avait décidé const au début du projet, a dû finalement être défini comme mutable à cause de la capacité joker, qui permet de choisir le bonus couleur de la carte.

Pour les cartes royales, représentées par la classe RoyalCard, nous avons également choisi d'ajouter un attribut statique pour en limiter le nombre, puisque seuls 4 exemplaires existent sur le plateau.
Pour le reste, les cartes royales héritent simplement de leur classe mère Carte.

Tirage

Cette classe constitue une représentation de chacun des trois tirages disponibles dans le jeu. Chaque tirage est essentiellement une collection de cartes, et pour simplifier la gestion de ces cartes, nous avons opté pour l'utilisation de vecteurs. Cette approche offre une souplesse significative, notamment lors du remplissage du tirage, où il suffit de piocher une carte depuis la pioche et de l'ajouter au tirage en utilisant la fonction "push" associée aux vecteurs.

L'affichage du contenu complet d'un tirage est également rendu aisé grâce à l'opérateur "<<", permettant ainsi une visualisation rapide pendant le déroulement du jeu. De plus, une fonction clé, "getCarte()", a été implémentée pour faciliter le transfert d'une carte du tirage vers la main du joueur.

Étant donné qu'il n'y a que trois tirages dans le jeu, nous avons introduit un attribut statique de classe pour limiter le nombre maximal de tirages. Cette approche assure la présence constante de seulement trois instances de la classe de tirage, garantissant ainsi une gestion efficace et cohérente de cette composante du jeu.

En résumé, la classe tirage permet non seulement la gestion des cartes, mais également l'affichage et le transfert fluide de ces cartes entre le tirage et la main du joueur. L'utilisation judicieuse des vecteurs et l'utilisation des membres statiques contribuent à la robustesse et à l'efficacité de cette composante.

Pioche

La classe Pioche est conçue pour représenter une collection de cartes de joaillerie, principalement implémentée à l'aide d'un vecteur. À l'initialisation, chaque pioche est mélangée pour introduire une composante aléatoire dans le jeu. Comme dans le cas des tirages, un attribut statique est utilisé pour restreindre le nombre de pioches créées à un maximum de 3.

Sac

La classe Sac représente une collection de jetons, structurée au moyen d'un vecteur. Contrairement à d'autres composants du jeu, aucune composante aléatoire n'est intégrée dans cette classe. L'aspect aléatoire est introduit lors du remplissage du plateau, où les jetons sont attribués de manière aléatoire. La classe Sac adopte également le design pattern 'Singleton', reflétant la singularité de cette entité dans l'ensemble du jeu, car il n'y a qu'un seul sac utilisé. Le Sac n'a pas de responsabilité pour les jetons, qui sont créés et libérés avec le Jeu.

Plateau

La classe Plateau représente une collection de jetons organisée à l'aide d'un tableau à une dimension (vecteur) de 25 pointeurs vers des jetons constants. Pour simuler la structure matricielle 5x5 d'un véritable plateau, des opérations modulo sont employées pour naviguer dans le vecteur. Par exemple, le jeton d'indice 5 correspond au dessous du jeton d'indice 0. Le recours au Design Pattern 'Singleton' s'impose également, étant donné qu'il existe une seule instance de plateau dans le jeu.
L'aspect aléatoire du placement des jetons prend place au moment du remplissage du plateau, ajoutant une dimension de variabilité au jeu.

Jeton

La classe Jeton représente les différents jetons du jeu, qui sont au nombre limité de 25. Pour représenter cela nous avons mis en place, cette fois-ci, un attribut statique pour chacune des couleurs de jetons, puisque chaque couleur comptabilise 4 jetons, à l'exception des couleurs or et perle qui en comptent respectivement 3 et 2. Ces couleurs sont par ailleurs définies de la même façon et pour les mêmes raisons que les couleurs des bonus des cartes joailleries, grâce à un *enum class*.

Privilège

La classe Privilege représente l'une des classes les plus simples de notre projet. Son design a été pensé de manière à restreindre le nombre d'instances de privilèges à trois, grâce à l'utilisation d'un membre statique. Chaque privilège est identifié par un identificateur distinct, simplifiant ainsi leur manipulation au sein du

système. Cette approche minimaliste contribue à la clarté et à l'efficacité de la gestion des privilèges.

Match

La classe Match représente le résultat d'une partie jouée par deux joueurs, IA ou non. Un match contient les informations relatives à la partie que l'on souhaite sauvegarder, telles que les noms du gagnant et du perdant, leur score ou encore le fait qu'ils soient IA ou non. Le principal intérêt de cette classe est qu'elle permet d'enregistrer ces informations dans un fichier json.

History

La classe History représente l'historique des parties jouées sur notre jeu. Un objet de la classe History est un objet pouvant regrouper tous les matchs de tous les joueurs du jeu (on parle ici des instances de Strategy_player, et pas uniquement de la classe fille Joueur). De ce fait, nous avons convenu que la classe serait implémentée à l'aide du design pattern Singleton, afin d'en garantir l'unicité. Cette classe assure l'enregistrement des matchs au sein d'un fichier json, ainsi que la récupération des données de ce fichier pour permettre l'affichage de l'historique aux joueurs. De plus, nous avons choisi pour cette classe qu'un joueur entrant un nom déjà contenu dans l'historique sera alors considéré comme le même joueur. Non seulement cette classe permet cela, mais elle permettra en plus au jeu de repartir du même pointeur de joueur.

Développement du Main

Le programme principal (main) assure le bon déroulement d'une partie. C'est le développement de celui-ci qui nous a causé le plus de problèmes. En effet, le main doit garantir la bonne interaction entre l'ensemble des classes. De plus, après avoir fait le choix d'utiliser le design pattern 'Strategy' pour que le main soit adapté autant pour un joueur humain qu'une IA, nous avons dû reprendre le main dans son intégralité car celui-ci n'était pas adapté à l'utilisation de ce design pattern.

Conclusions sur l'architecture du projet

Pour conclure sur cette partie, nous avons fait notre possible pour créer une architecture optimisée et robuste, notamment en utilisant au maximum les types de

la bibliothèque standard C++ comme les vector. Mais nous pouvons tout de même soulever certains compromis que nous avons dû faire, entre robustesse, complexité et temps.
On peut par exemple parler des *enum class* que l'on a utilisés en lieu et place de véritables classes, qui ont l'avantage de nous avoir grandement simplifié le travail sur certains points, mais aussi quelques désavantages, dont certains seront mentionnés dans la partie suivante.

Évoluabilité

Tout au long de la conception du projet, il nous a tenu à cœur de faire attention à ce que notre structure soit facilement modifiable, et ce dès le commencement du projet, lors des premières étapes de réflexion.

Ajout de nouvelles IA joueurs

Grâce à l'utilisation du design pattern Strategy pour les classes de joueurs, il suffit de rajouter une nouvelle classe qui hérite de *Strategy_player* pour ajouter un nouveau type de joueur ou d'IA, par exemple une IA avec un niveau de difficulté plus élevé.

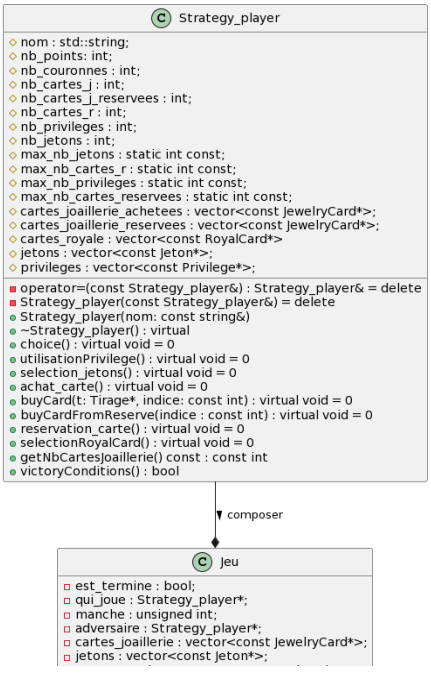
En utilisant le principe de polymorphisme, les classes filles (Joueur et IA) peuvent hériter du comportement de la classe mère, et grâce aux méthodes virtuelles, on peut facilement définir ces dernières dans chaque classe fille et réutiliser les méthodes de la classe mère sur un objet d'une classe fille.

Ajout ou remplacement d'une règle

Les classes à modifier pour l'ajout d'une règle dépendent du type de cette dernière. Prenons par exemple une règle de condition de victoire : "Le joueur gagne s'il possède 5 couronnes et 10 points". Ici, il suffirait de rajouter cette condition dans la méthode *victoryConditions()* de la classe Strategie_player :

```
bool Strategy_player::victoryConditions() {
    if(nb_courones >= 10)
        return true;
    if(nb_points >= 20)
        return true;
    //ajout de règle
    if(nb_courones >= 5 && nb_points >= 10)
        return true
```

victoryConditions() est appelée dans la méthode *tour_suivant()* de la classe Jeu. Ainsi dans le main à chaque fois qu'on appelle *tour_suivant()* (pour changer de tour), les conditions de victoires sont vérifiées simplement et efficacement. Cela est permis grâce au lien de composition entre les deux classes visible sur l'UML simplifié ci-dessous. En effet, l'attribut booléen *est_termine* de Jeu est assigné à false par défaut et se voit possiblement attribuer la valeur true par *tour_suivant()*.



Ainsi le changement ou l'ajout d'une règle de ce type ne remet pas en cause le bon déroulement du jeu.

Pour ajouter de nouvelles règles de types différents, il faudrait modifier les méthodes de classe concernées, mais le fait que toutes les classes principales composent le jeu permet une certaine simplicité d'implémentation tout en limitant l'impact sur le reste du programme.

Néanmoins, il aurait été plus efficace après réflexion d'ajouter une classe *Regle* à part entière en lui appliquant le design pattern Singleton. Ainsi nous aurions accès à ses attributs représentants toutes les règles du jeu, ce qui aurait permis de modifier cette classe et donc les règles du jeu plus efficacement.

Ajout d'un nouveau type de capacité

Les types de capacité sont contenus dans un *enum class* nommé *Capacity*.
Puis chaque instance de la classe *Card* possède possiblement un attribut *capacite* de type *const optional<Capacity>*.
Ajoutons à titre d'exemple une capacité permettant de voler une carte à l'adversaire (capacité très puissante cela dit en passant). Il faudrait d'abord l'ajouter dans l'*enum class* :

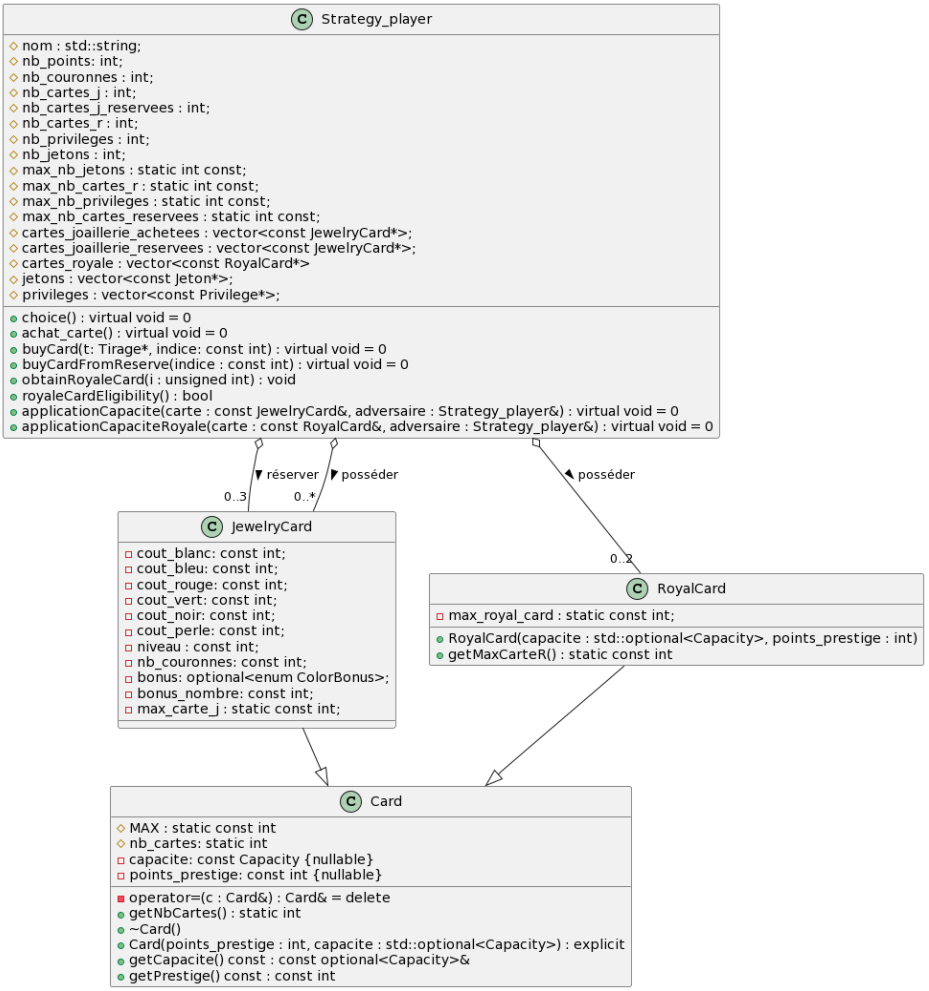
```
enum  class  Capacity {rejouer, voler_pion_adverse,
        prendre_privilege,prendre_sur_plateau, joker, //voler_carte};
```

Puis l'effet que procure cette capacité sera appliqué dans les méthodes virtuelles de Strategie_player *applicationCapacite(const JewelryCard& carte, Strategy_player& adversaire)* et *applicationCapaciteRoyale(const RoyalCard& carte, Strategy_player& adversaire)* lors de l'achat de la carte par le joueur selon que la carte soit une carte Joaillerie ou une carte Royale. Il faudra simplement ajouter le code correspondant à l'effet de la capacité dans l'espace commenté sur la photo ci-dessous :

```
else if (capa == Capacity::prendre_privilege){
    Jeu::getJeu().getCurrentPlayer().obtainPrivilege();
}
/*
else if (capa == Capacity::voler_carte){
    ...
}
 */
else if (capa == Capacity::prendre_sur_plateau){
```

Enfin, il faudra implémenter ce code à quelques modifications près dans la définition des méthodes associés dans chaque classe fille de Strategy_player en raison de la nature virtuelle de ces méthodes.

L'UML simplifié ci-dessous représente les relations entre *Strategie_player* et *Card* avec ses classes dérivées :



Ce sont les agrégations des classes *JeweleryCard* et *RoyalCard* dans *Strategy_player* qui permettent de gérer facilement d'éventuelles modifications et ajouts de caractéristiques de cartes via des méthodes virtuelles de *Strategy_player*.

Ajout d'une nouvelle carte

L'expression "ajout d'une nouvelle carte" est plutôt ambigu. En effet si l'on parle d'ajouter une carte d'un type (Joaillerie ou Royale) déjà implémenté, il nous suffit d'ajouter les caractéristiques de cette carte dans le json :

Puis il faudrait incrémenter l'attribut static *MAX* de *Card* pour que le constructeur soit en mesure de construire la carte (car nous limitons le nombre d'instances de *Card* à *MAX*).

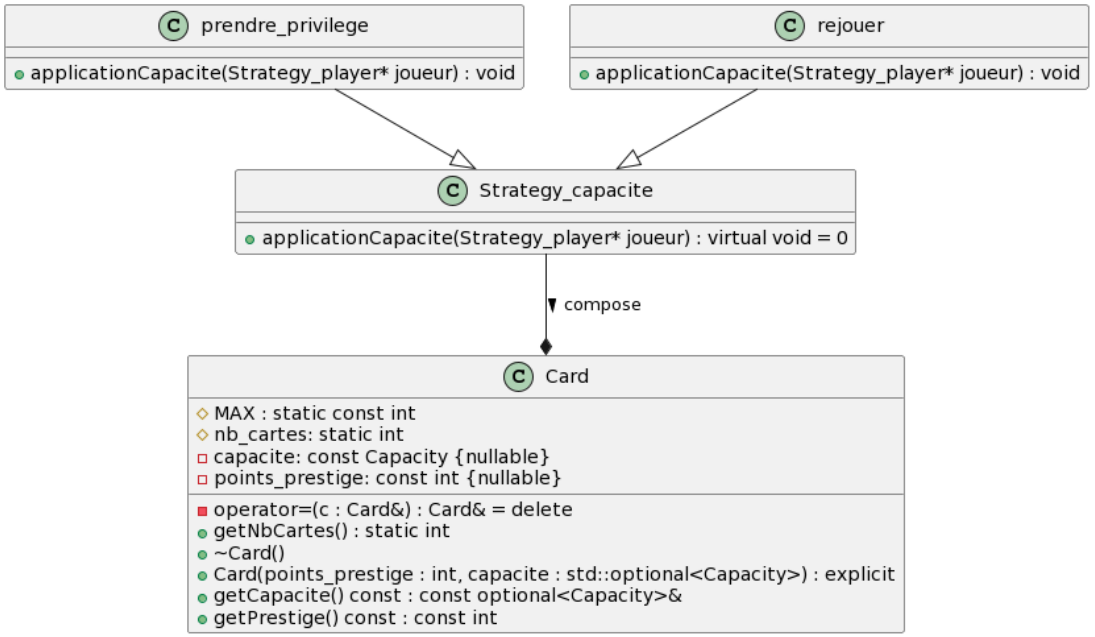
Autrement si l'on parle d'ajouter un nouveau type de carte, alors il faudrait créer une nouvelle classe héritant de *Card*. Enfin comme expliqué précédemment, les caractéristiques et le comportement de ce nouveau type de carte seront gérés dans la classe *Strategy_player* et ses classes filles.

Pistes d'améliorations sur l'évoluabilité du jeu

Bien que notre application soit en partie évoluable, certaines idées de choix architecturaux nous sont venues trop tard, les changements étant impossibles à faire faute de temps. Nous avons conscience de cela notamment pour les règles qui constituent notre principale point faible en matière d'évoluabilité.

- Par exemple pour l'évoluabilité des règles, il aurait été plus efficace après réflexion d'ajouter une classe *Regle* à part entière en lui appliquant le design pattern Singleton. Ainsi nous aurions accès à ses attributs représentants toutes les règles du jeu, ce qui aurait permis de modifier cette classe et donc les règles du jeu plus efficacement.

- Pour l'évoluabilité concernant l'ajout de capacité, nous aurions pu appliquer le design pattern *Strategy* sur une classe Strategy_*Capacite* en ajoutant les classes filles correspondant chacune à une capacité. L'UML simplifié ci-dessous permet d'illustrer l'idée :



Le choix d'architecture expliqué dans la section précédente du rapport permet de cibler assez précisément les parties de code à modifier et/ou à implémenter suite à l'ajout de nouvelles fonctionnalités, bien que certaines restent malgré tout plus complexes.

Qt

Pour la partie Qt, nous avons dû reprendre toutes les fonctions du projet nécessitant une forme d'interaction avec le joueur pour les rendre compatibles avec l'interface graphique. Pour cela, il a dans un premier temps fallu lier les visuels du jeu aux objets auxquels ils correspondent, puis s'accommoder des signaux de Qt pour faire interagir les actions des utilisateurs avec le back end.

Beaucoup d'autres fonctions ont également dû être créées pour la partie Qt, notamment pour gérer l'état des objets de l'interface graphique, comme l'état cliquable des cartes ou des jetons.

Tout comme la version console, cette partie est jouable aussi bien en mode humain-humain, IA-humain que IA-IA. Pour que l'affichage des actions se fasse correctement dans ce dernier mode, nous avons utilisé la fonction *QCoreApplication::processevents()*, ce qui ralentit le jeu mais augmente la lisibilité.

Un autre concept de cours a été utilisé pour le Qt, le design pattern Template Method. Nous l'avons utilisé pour créer un bouton permettant de voir les statistiques des joueurs calculées grâce aux attributs de nombre de victoires des joueurs, qui transitent entre les parties grâce à l'historique.

Remarque :

Par manque de temps, la validation de chacune des actions des joueurs (validation avec le popup qui demande la confirmation) n'a pas pu être entièrement finie avant la date du rendu.

Planning effectif

Jusqu'au premier rapport, les premières semaines de travail ont été consacrées à la réflexion autour du sujet. Cette étape de réflexion nous a amené à réaliser une analyse des concepts du jeu ainsi que le tout premier diagramme UML de notre projet, présents dans le rapport 1.

Les semaines suivantes ont été consacrées, jusqu'au second rapport, à l'implémentation quasi complète de toutes les classes, à l'exception de Jeu et Joueur, qui n'ont pu être terminées qu'après puisqu'elles dépendent de toutes les classes, il fallait que les autres soient finies pour être en capacité de terminer le développement de Jeu et Joueur.

Après le second rapport, nous avons commencé par nous occuper de terminer les classes Jeu et Joueur, en corrigeant tous les bugs, pour pouvoir enchaîner avec le développement du main. En a découlé le développement de l'IA aléatoire, et pour cela l'implémentation du design pattern Strategy. Enfin, une mise à jour de l'UML a été réalisée, une première (re)prise en main de Qt a été faite et la gestion des capacités des cartes a été débutée.

Durant les deux dernières semaines, sur la dernière ligne droite, la sauvegarde de la partie en cours et l'historique des parties ont été implémentés en parallèle de la réalisation du front avec Qt.

Pistes d'amélioration

Nous avons durant le projet réfléchi à sécuriser l'accès aux fichiers json de backup et d'historique grâce à une signature numérique que le seul le jeu pourrait créer, et qui empêcherait l'utilisation par les joueurs de fichiers qu'ils auraient modifiés. Cependant, faute de temps, nous n'avons pas pu implémenter cette sécurité supplémentaire.

Contribution effective

Pourcentage de contribution sur l'ensemble du projet :

- Alexandru : 25%
  Jeu, Joueur, Qt (60%)
- Aubin : 25%
  Sac, Plateau, Jeton, Strategy_player, IA, Joueur, Qt (40%), main, scans des cartes, json des cartes
- Gaspard : 15%
  Pioche, capacités, rendu final
- Léopold : 20%
  main, Cartes, historique, stockage, intégration nlohmann
- Quentin : 15%
  Tirage, ajout des liens vers les visuels du jeu, rendu final

Tous les membres ont également participé au fix des bugs (principalement Léopold) et aux rapports.

Temps consacré au projet :

- Alexandru : ~ 110h
- Aubin : ~ 110h
- Gaspard : ~ 60h
- Léopold : ~ 80h
- Quentin : ~ 60h

Conclusion

Pour conclure, ce projet nous a permis d'apprendre à gérer un projet conséquent en groupe. Cette expérience a été très enrichissante pour l'ensemble des membres du groupe, en mettant parfois à rude épreuve nos compétences techniques et notre patience. Par ailleurs, l'ambiance entre les membres est restée bonne constamment durant le projet, ce qui a rendu cette expérience d'autant plus agréable. En définitive, nous sortons grandis de cette expérience.

Annexe