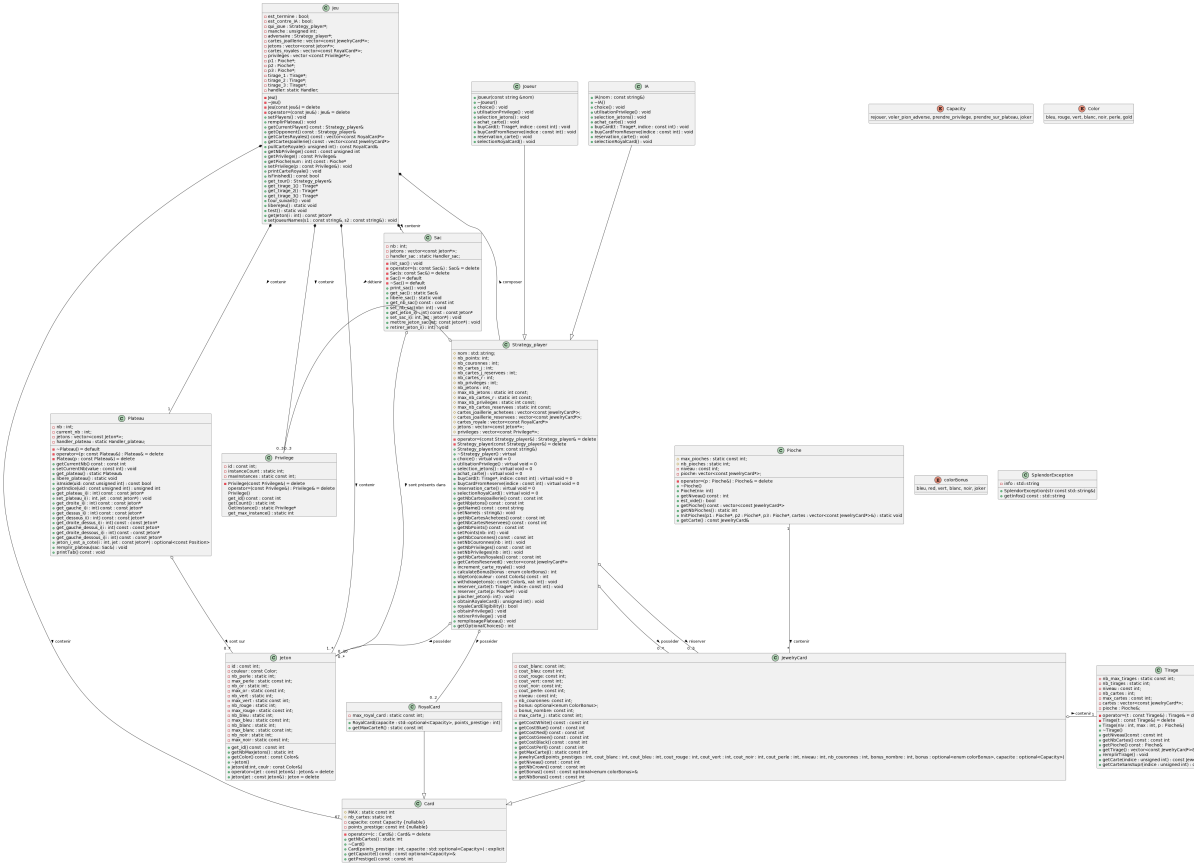

Architecture : remarques générales et difficultés

L’architecture 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 dernères 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 un 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 des 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 détailler, pour chaque classe, certaines décisions des architectures et les éventuelles difficultés dans la prochaine partie.

Les 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.

Difficultés rencontrées:

La difficulté principale dans le développement de cette classe a été bien évidemment la conservation de la cohérence avec les autres classes. De plus, une autre difficulté

a été de bien développer le constructeur et destructeur du jeu afin de garantir la bonne initialisation et destruction de tous les éléments du jeu.

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.

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: 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.

Difficultés rencontrées:

Lors du choix de l'implémentation du DP Strategy, nous avions déjà developpé en partie notre programme principal (main). Cependant, celui-ci n'était pas du tout adapté à l'utilisation de ce DP et cela nous à demandé plus de 13h de travail de refactoring et d'adaptation.

Carte

Nous avons une classe mère Card et 2 classes filles Royal_Card 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.

Difficultés rencontrées:

Les difficultés liées à la classe Cartes ont principalement émergé lors de l'intégration de la bibliothèque Nlohmann, essentielle pour permettre l'indexation de chaque 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.

Nous n'avons pas rencontré de difficultés pendant la conception de cette classe.

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.

La conception de la classe Pioche a été réalisée sans difficulté particulière.

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 crées et libérés avec le Jeu.

La conception de la classe Sac a été réalisée sans difficulté particulière.

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 des jetons prend place au moment du remplissage du plateau, ajoutant une dimension de variabilité au jeu.

Difficultés rencontrées :

Malgré la conception globalement réussie de cette classe, la gestion précise de l’alignement des jetons lors de la pioche a présenté une difficulté significative, demandant un investissement d’environ 15 heures pour résoudre cette problématique spécifique.

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.

Nous n’avons pas rencontré de difficultés pendant la conception de cette classe.

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 du reprendre le main dans son intégralité car celui-ci n’était pas adapté à l’utilisation de ce design pattern. Cela représente plus de 15 heures de travail.

Les tâches déjà réalisées

Le développement des classes:

- Jeu
- Joueur
- Carte
- Tirage
- Pioche
- Sac

- Plateau
- Privilège

Le développement du main

Le développement de l’IA

Les tâches à venir (par ordre de priorité) et repartition du travail

- Sauvegarde d’une partie (Design Pattern ‘Builder’) : Léopold
- Fonction ‘undo’ (Design Pattern ‘Momento’) : Aubin
- Interface Qt, interaction entre front end et back end : Quentin, Gaspard, Alexandru
- Rapport final : Tout le monde
- Démo du code en vidéo : Tout le monde
- Refactor du code : Tout le monde

Temps passé pour la réalisation depuis le dernier rapport

Jeu : Alexandru (7h)

Refactor Sac : Alexandru (2h)

Joueur : Alexandru (2h)

Refactor Joueur et Jeu : Léopold (20h)

Refactor Joueur et Jeu : Quentin (5h)

Gestion de la prise des jetons (allignement) : Aubin (15h)

Développement du main : Léopold (6h)

Refactor du projet dans son intégralité (adaptation au DP ‘Strategy’) : Aubin (13h)

Développement de l’IA : Aubin (8h)

Rapport : Alexandru (7h), Aubin (2h)

UML : Alexandru (2h)

Prise en main de Qt : Quentin (1.5h), Gaspard (5h)

Capacités cartes : Gaspard (3h)

Conclusion et bilan

En conclusion, le rapport met en évidence les différentes composantes essentielles du projet, en se concentrant sur l’architecture, le développement et les défis rencontrés. À l’étape actuelle, la fonction principale du programme (main) est pleinement opérationnelle, offrant une expérience de jeu interactive dans divers modes tels que joueur contre joueur, joueur contre IA, et même IA contre IA. Nous orienterons désormais nos efforts vers les dernières étapes du projet, en mettant en œuvre les fonctionnalités spécifiques mentionnées précédemment. Cette phase de développement final vise à consolider l’ensemble du jeu Splendor Duel, assurant une réalisation complète et fonctionnelle de notre vision initiale.

En ce qui concerne la cohésion du groupe, tout le monde s’entend bien et s’entraide. Nous adoptons une approche flexible, travaillant à la fois à distance et en présentiel, ce qui nous permet de résoudre rapidement les éventuels problèmes et de maintenir une dynamique positive au sein de l’équipe.