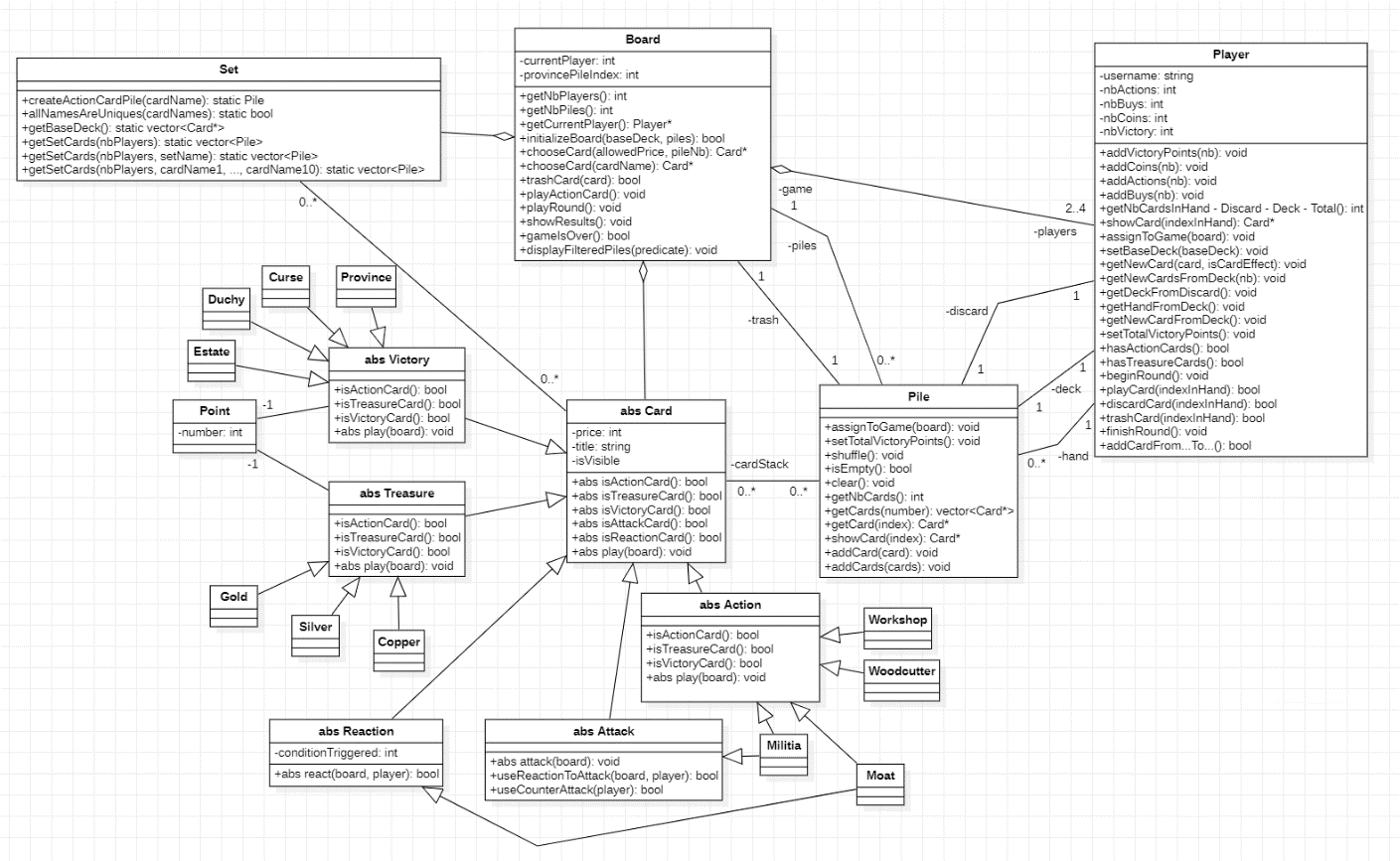
**Rapport de projet – Dominion**

1. **Cahier des charges**
2. Diagramme de classes



1. Commentaires sur le diagramme de classes :

* Board : cette classe représente le plateau de jeu, sur lequel les cartes (stocks et rebut) et les joueurs vont être placés. Les différentes méthodes sont liées aux actions générales du jeu, à savoir la mise en place du plateau, les tours de jeu et les différentes phases, ainsi que l’attente de la fin de partie et l’annonce des résultats.
* Set : cette classe statique permet d’initialiser les piles de cartes d’une partie en sélectionnant les cartes utilisées, soit en donnant le nom d’un set de base, soit en donnant le nom de 10 cartes, soit aléatoirement. Les noms disponibles ainsi que leurs significations seront déclarés dans des énumérations, afin de ne pas être perdu lors de l’appel à la classe Set.
* Pile : cette classe représente un paquet de cartes. Au départ une stack était utilisée, mais comme il est possible de choisir des cartes dans différentes piles ou bien de les mélanger, il a finalement été plus utile d’utiliser directement un vector. Ainsi, la main du joueur qui n’était pas une Pile au départ a pu être inclue comme instance de cette même classe. Différentes actions de base sont disponibles, comme le mélange, la suppression et l’ajout de carte(s), ou encore le compte du nombre de cartes.
* Player : cette classe représente un joueur et les différentes statistiques utiles durant la partie (nbActions, nbBuys, nbCoins, nbVictory). Il possède aussi plusieurs variables de classe Pile correspondant au deck, à la main et à la défausse. Les différentes actions disponibles sont liées au jeu avec les différentes cartes disponibles en main, qui peuvent aussi donner lieu à des actions directement sur le deck ou la défausse, comme la pioche de cartes par exemple.
* Point : cette classe est une classe mère de certaines classes de type de carte (Victory et Treasure), qui ont un certain nombre de points attribués. D’autre part, les cartes de type Action n’ont pas de points donc n’ont pas besoin de l’implémenter. Une carte Malédiction est une carte spécifique de type Victoire avec un nombre de points de -1.
* Card : la classe abstraite de base de toute sous-classe de type de carte ou de carte spécifique. Elle se compose d’un prix d’achat, d’un titre et d’un booléen pour savoir si la carte est visible (utile par la suite pour la partie graphique). Elle contient les méthodes virtuelles pures de base qui permettent de savoir de quel type de carte il s’agit (isActionCard(), isVictoryCard(), isTreasureCard()), ainsi que de la méthode play().
* Action / Treasure / Victory / Attack / Reaction : les différentes classes abstraites de type de cartes existants. Chaque carte spécifique doit être une classe (Gold, Province, Witch, Workshop, etc.) qui hérite d’une un plusieurs classes de type de carte. Par exemple la carte Milice qui est une carte Action et Attaque en même temps, doit hériter d’Action et d’Attack pour avoir toutes les fonctionnalités nécessaires. Les cartes ayant des comportements à résoudre via la console et dont un choix d’agir est possible peuvent être passée en entrant -1 dans une question qui demande d’entrer un nombre.

1. **Choix de conception et difficultés rencontrées :**
2. La base, le nombre de joueurs et le choix des 10 cartes Royaume

Pour réaliser la base, à savoir le jeu à 2 joueurs sur une console avec les cartes Royaume suggérées, je me suis contenté de suivre le cahier des charges décrit en amont. En plus de cela, il m’a paru très facile d’implémenter le choix du nombre de joueurs lors du début de partie, et cette extension fait donc partie intégrante du cahier des charges. Enfin, l’ajout d’une classe statique Set pour gérer la création des piles de cartes m’a permis d’élargir très facilement au choix des 10 cartes Royaume.

La principale difficulté a été de lier les classes Board, Player et Pile ensemble, car elles dépendent toutes l’une de l’autre. Le concept de « forward declaration » m’a permis de remédier à ce problème. Cette difficulté était liée à une autre, à savoir l’instanciation des cartes. D’une part, j’ai dû m’assurer que les classes Card, Action, Victory et Treasure étaient bien abstraites, car ce ne sont pas des cartes à proprement parler. Ce n’était évidemment pas le cas au départ, et j’ai dû pour cela modifier les vecteurs de cartes en vecteurs de pointeurs de cartes. La fonction play() a aussi été une cause de difficulté, car je n’étais pas sûr des paramètres qu’elle devrait prendre. En effet, chaque carte a un effet différent qui agit sur des paramètres différents du jeu. Afin de pouvoir l’utiliser pour tout type de carte, j’ai opté comme paramètre une référence à l’objet Board. On joue la carte sur le plateau de jeu, et la carte applique ainsi son effet aux différentes entités de ce plateau de jeu. La méthode play() peut être utilisée par un joueur possesseur de la carte ou par une pile de cartes (pour le compte des points de victoire). Ainsi, cela a nécessité d’assigner chaque pile et joueur à un plateau de jeu spécifique à l’initialisation de la partie, afin de passer ce plateau de jeu à la fonction play() des cartes.

Dans l’extension du choix des 10 cartes Royaume, la difficulté était de choisir la carte de manière optimisée. Pour cela, j’ai utilisé des énumérations des différentes cartes et sets disponibles, ainsi que des fonctions qui retournent le nom de la carte. Cela permet donc de donner une appellation générique à chaque carte et ainsi éviter des erreurs d’inattention lors de l’instanciation ou bien des transtypages trop fréquents.

1. L’interface graphique (captures d’écran en annexe)

Cette extension était la plus enrichissante selon moi, il était donc indispensable de la réaliser. J’ai pour cela utilisé la bibliothèque recommandée (SFML) afin de créer une première base graphique de l’interface de jeu. Pour des raisons de temps, elle reste très succincte mais est fonctionnelle. Plusieurs classes qui ne sont pas initialement dans le cahier des charges ont été ajoutées, pour que l’utilisateur puisse naviguer dans cette interface, à savoir les classes Button, TextButton, ImageButton, ButtonGroup, TextInputField et Text.

Comme le fonctionnement se base ici sur des boucles while et non plus sur des cout/cin, il m’a été très difficile de traduire l’effet de toutes les cartes vers l’interface graphique. Pour les cartes qui n’attendent rien de l’utilisateur, aucun problème. En revanche, pour toutes celles qui demandent un choix de l‘utilisateur, comment s’assurer que ce choix sur l’interface soit transmis à la classe de la carte afin que la fonction associée en retourne un résultat final ? Cette question reste encore en suspens, et tous les choix dus à des cartes doivent être résolus directement depuis la console. Un message en informe l’utilisateur sur l’interface lors de la partie.

De plus, la fenêtre de jeu ne peut pas être redimensionnée, par manque de temps pour développer l’aspect responsive des différents éléments. Cependant, contrairement à l’affichage simple en console, il est possible de voir les cartes de manière détaillées avec leur image, d’une part sur le plateau de jeu et d’autre part en mode plein écran en effectuant un clic droit sur une des piles de réserve du plateau. Dans les 2 modes de jeu, les cartes ne sont pas cachées, car on joue sur un seul écran.

1. La première extension et autres extensions du jeu

En ce qui concerne les cartes du set de base, elles ont été implémentées au fur et à mesure du développement des autres extensions (sauf les 10 premières créées dès le développement du jeu en mode console). Pour créer une nouvelle carte, il suffit de d’ajouter les fichier hpp et cpp de la nouvelle carte, redéfinir la méthode play() pour implémenter l’effet de la carte, la méthode attack() et react() selon le type, et éventuellement la fonction useEffect() pour le passage futur à l’interface graphique. Ensuite, on ajoute la carte à l’énumération qui convient, à la logique de création de pile dans la classe Set, on ajoute son image dans le dossier assets, et enfin on ajoute son nom dans le Makefile. A la fin de chaque implémentation, je me suis assuré lors de tests aux limites que l’effet voulu était bien obtenu, sans effets de bords, et que la carte apparaissait correctement dans l’interface graphique.

Plus le nombre de cartes augmentait, et plus leur effet a été compliqué à implémenter. En effet, certaines cartes demandent vraiment beaucoup de réflexions pour bien prendre en compte les choix des joueurs. Cela a aussi abouti à la création de nouvelles méthodes dans les classes Pile et Player afin de prendre en compte des comportements très spécifiques, comme le passage de cartes depuis le deck directement vers la défausse, ou bien l’acquisition d’une carte directement dans la main d’un joueur par exemple.

En ajoutant la carte Douves, je me suis rendu compte qu’il me manquait des types de cartes (Attack et Reaction). J’ai donc ajouté ces classes, ce qui a engendré un problème en diamant pour les classes héritant de 2 types à la fois. J’ai alors ajouté l’héritage virtuel sur les classes de types de cartes, ce qui a abouti pour les cartes spécifiques à devoir appeler le constructeur de Card en plus des constructeurs de types de cartes. Ainsi, j’ai dû modifier le constructeur des cartes spécifiques pour appeler directement le constructeur de Card en plus. D’autre part, j’ai séparé la logique d’attaque des cartes Attack dans une fonction spécifique appelée attack() et j’ai ajouté une fonction react() dans les cartes de type Reaction. C’est aussi à ce moment-là que j’ai remarqué que les cartes Trésor et Victoire ne devait pas hériter de la classe Point, mais finalement contenir une donnée membre de ce type. Ainsi pour des cartes qui sont à la fois Trésor et Victoire, il suffira d’appeler la méthode setVictoryPoints() pour récupérer les points de victoire, et d’utiliser play() pour augmenter son nombre de pièces, en précisant quel type de carte on utilise pour récupérer le nombre de points directement dans la classe de carte spécifique (par exemple Harem).

Pour certaines cartes spécifiques, le comportement n’a pas pu être implémenté en totalité. En particulier la carte Marchand (Merchant), qui n’est pas une carte de type Duration, mais dont le comportement y ressemble en partie. Le joueur gagne 1 pièce s’il joue une carte Argent lors de son tour. Pour l’instant, la carte se contente d’observer les cartes de la main du joueur pour voir s’il possède un Argent. Il est cependant possible de jouer d’autres cartes Action par la suite et piocher de nouvelles cartes dans sa main, alors que l’effet du Marchand est terminé. Ce type de carte nécessiterait donc une légère modification de la conception actuelle, en ajoutant une liste de carte jouées à ce tour (dans la classe Board ou dans une classe spécifique Round), qui seraient ajoutées à la défausse seulement en fin de tour du joueur, ou alors une classe Mat (« player mat », utilisé notamment dans l’extension Seaside).

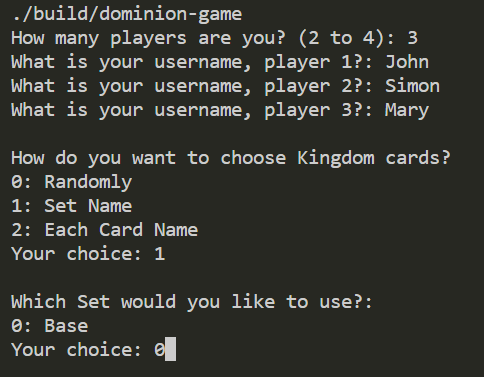
1. La sauvegarde de partie

Cette extension n’est disponible que dans la version graphique du jeu. En effet, il est facile d’ajouter un bouton « sauvegarder et quitter » à l’interface et de garder les variables en suspens jusqu’à ce que l’utilisateur reprenne la partie. Dans le cas où il relance une nouvelle partie, il suffira de supprimer les données de la partie précédente.

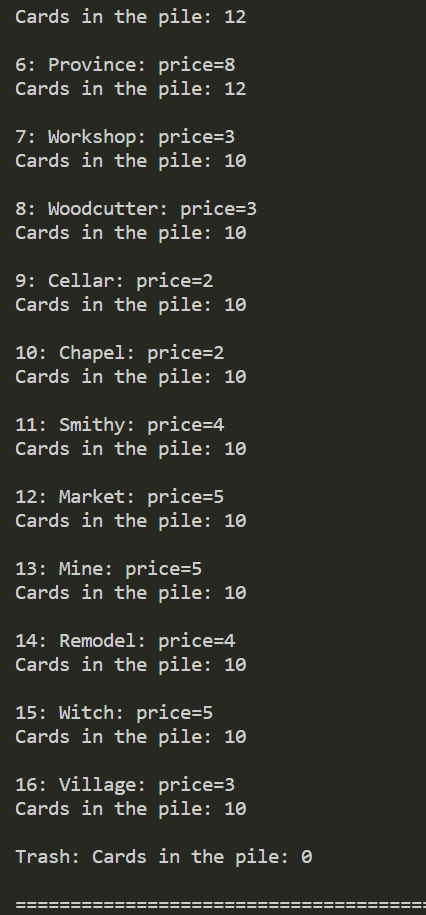
Dans la console, une fois la partie finie ou quittée, l’exécution est totalement terminée. Il serait possible de sauvegarder dans un fichier (texte, JSON, …) les valeurs des variables, afin de les réutiliser lors du lancement du programme la fois suivante. Cela se ferait, soit en continu (traitement lourd), soit lors de l’arrêt du processus (avec CRTL + C), soit avec un indice spécifique entré par l’utilisateur (-5 ou quit par exemple).

Pour lancer le programme, il suffit de suivre les instructions données dans le README.

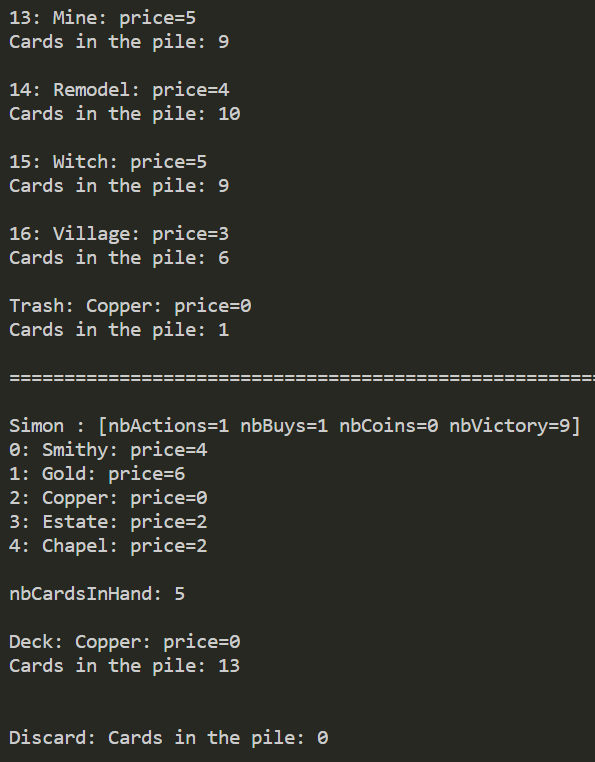
1. **Annexes**
2. Initialisation de la partie (CLI)



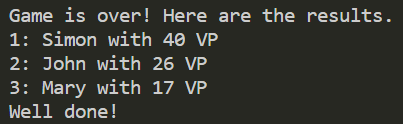
1. Affichage des piles de cartes du plateau (CLI)



1. Tour de jeu d’un joueur avec affichage de son profil (CLI)



1. Fin de partie et affichage des résultats (CLI)



1. Choix du nombre de joueurs et du mode de choix des 10 cartes Royaume (bouton Resume visible seulement si la partie a été interrompue en cours de route) (GUI)

Une image contenant texte, peinture, art, arbre

Description générée automatiquement

1. Entrée du nom des joueurs par l’utilisateur (GUI)

Une image contenant habits, Personnage de fiction, Jeu PC, Art numérique

Description générée automatiquement

1. Choix du set prédéfini de 10 cartes Royaume (GUI)

Une image contenant Jeu PC, Jeu vidéo de stratégie, Jeu d’aventure, Jeu d’action-aventure

Description générée automatiquement

1. Choix par nom des 10 cartes Royaume (les noms s’enlèvent lorsqu’on les choisit) (GUI)

Une image contenant texte, embarcation, transport, bateau

Description générée automatiquement

1. Ecran de chargement (GUI)



1. Plateau de jeu lors d’une partie (GUI)

Une image contenant texte, capture d’écran, plein air, art

Description générée automatiquement

Une image contenant texte, art

Description générée automatiquement

1. Détails sur une carte spécifique (GUI)

Une image contenant texte, dessin humoristique, capture d’écran, Jeu PC

Description générée automatiquement

1. Fin de partie et affichage des résultats (GUI)

Une image contenant texte, embarcation, transport, bateau

Description générée automatiquement