Léna BAILLET - Alice DEMONFAUCON - Louis GREINER - Myriem KHAL - Lucas STALLKNECHT

LO21 **Projet Splendor - Rapport Final**

Application de jeu de société en C++

P21

TABLE DES MATIÈRES

1. RAPPEL DES RÈGLES DU JEU

Le but du jeu est d'atteindre un certain seuil de points de prestige en cumulant les cartes Développement et Nobles.

Fig 1 : Exemple de carte de développement (cercle vert représentant le nombre de points de prestige que la carte donne, encadré bleu représentant le coût de la carte, et la pierre précieuse en haut à droite représentant l'avantage en ressource octroyé)

Le jeu est composé de :

- 40 jetons de pierres précieuses :
  - 7 diamant (5 pour une partie à 3 joueurs, 4 pour 2 joueurs)
  - 7 saphir (5 pour une partie à 3 joueurs, 4 pour 2 joueurs)
  - 7 émeraude (5 pour une partie à 3 joueurs, 4 pour 2 joueurs)
  - 7 rubis (5 pour une partie à 3 joueurs, 4 pour 2 joueurs)
  - 7 onyx (5 pour une partie à 3 joueurs, 4 pour 2 joueurs)
  - 5 or (équivaut à un joker de re, cela peut remplacer n'importe quelle ressource)
- 90 cartes de développement :
  - 40 de niveau 1
  - 30 de niveau 2
  - 20 de niveau 3
  - 10 cartes nobles

Le plateau est composé de :

- 3 pioches de cartes de développement (faces cachées) :
  - niveau 3
  - niveau 2
  - niveau 1
- une grille de 4 cartes de chaque niveau (4x3, faces visibles)
- n+1 cartes noble (soit n le nombre de joueurs, faces visibles, qui visitent les joueurs si leurs avantages de ressources sont suffisants, les autres cartes noble ne serviront pas pour cette partie)

La main d'un joueur est composée de :

- son nombre de points de prestige (visible de tous)
- son stock de pierres précieuses (total maximum de 10 à la fin de son tour, sinon il doit défausser jusqu'à revenir à 10) et ses avantages (visible de tous)
- ses cartes réservées (maximum de 3, visibles pour le joueur, faces cachées pour les autres)
- ses nobles (visibles de tous)

Les mains des autres joueurs sont affichées, version faces cachées.

Lorsqu'un joueur atteint le nombre de points de prestige nécessaire (généralement 15), le tour de jeu se termine puis le vainqueur est celui qui a le plus de points de prestige à la fin de ce tour. En cas d'égalité, le joueur ayant acheté le moins de cartes l'emporte. Si il y a encore égalité, alors c'est un match nul.

Fig 2 : Interface graphique de la version officielle jouable sur la plateforme Steam

A chaque fois qu'un joueur peut jouer, il parcourt les points suivants :

1) Choix d'une action parmi les 4 :

- Prendre 3 jetons de couleurs différentes
- Prendre 2 jetons de la même couleur (seulement s'il reste 4 jetons de cette couleur ou plus)
- Réserver une carte développement (de la grille visible, ou depuis une pioche), et prendre un jeton d'or (s'il en reste sur le plateau, sinon rien)

- Acheter une carte développement (préalablement réservée ou depuis la grille visible) si le joueur possède les ressources nécessaires (addition des ressources possédées et des avantages du joueur
2) Défausser les jetons si le joueur a plus de 10 jetons
3) Si ses avantages en ressources le lui permettent, visite d'un noble (et octroiement de points de prestige), attention, si un joueur peut recevoir la visite de plusieurs nobles, il doit alors choisir celui qu'il reçoit

2. CE QUE PERMET L'APPLICATION

L'objectif de ce projet est de se familiariser avec les différents concepts de la programmation orientée objet, à travers la conception et le développement d'une application permettant de jouer au jeu de société Splendor. L'application permet donc de jouer des parties de 2 à 4 joueurs, opposant des humains ou des IA. À terme, l'application permet de jouer avec des extensions, et de sauvegarder et reprendre une partie en cours.

Opérations attendues implémentées :

- Paramètres des joueurs : nombre de joueurs modulable entre 2 et 4, nom, IA (un seul niveau de difficulté) ou humain
- Respect des règles du jeu original
- Activation d'une extension
- Interface graphique intuitive et simple
- Du fun pendant toute la partie

Opérations attendues non implémentées :

- sauvegarder une partie en cours
- reprendre une partie depuis une sauvegarde
- cartes réservées des joueurs dont ce n'est pas le tour face cachée (actuellement, tous les joueurs voient les cartes réservées des autres joueurs)
- choisir le noble que le joueur souhaite obtenir dans le cas où il est éligible à la visite de plusieurs nobles (actuellement, c'est le premier noble dans la liste des nobles qui visite le joueur)

3. ARCHITECTURE
3.1 DESCRIPTION DU FONCTIONNEMENT

Nous avons décidé d'utiliser une architecture **MVC (Model View Controller)** pour notre application, et les différents design patterns Singleton ou encore Abstract Factory ont été utilisés (en plus de ceux que nous avons utilisés sans faire exprès).

Singleton: garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance. Utilisé dans notre projet pour la classe "Game".

Abstract factory: Utilisé pour les constructeurs de Player et Card notamment. Permet de créer des familles d'objets apparentés sans préciser leur classe concrète et de déclarer explicitement des interfaces pour chaque produit de la famille de produits. Pour chaque variante d'une famille de produits, nous créons une classe fabrique qui implémente l'interface de la classe mère abstraite.

Une architecture MVC est composée de trois modules :

- le **modèle** qui contient les données à afficher (= la base de données)
- la **vue** qui contient la présentation de l'interface graphique (= fonctions d'affichage)
- le **contrôleur** qui contient la logique concernant les actions effectuées par l'utilisateur (= gestion des requêtes, interface de communication entre l'utilisateur, le modèle et les vues)

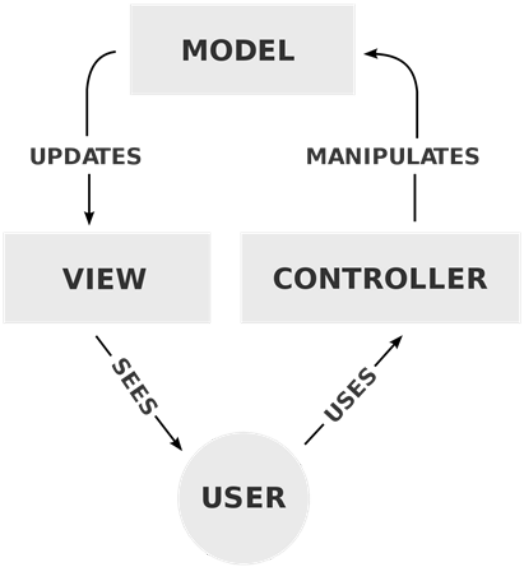


Fig 3 : Schéma d'une architecture MVC

Le principe d'une architecture MVC repose sur le fait que l'utilisateur (le joueur) ne peut qu'envoyer des requêtes (qui n'altèrent donc pas directement l'état du jeu) au Controller. Le controller traite cette requête, et en sortie modifie ou non l'état du jeu. Si l'état du jeu est modifié, la vue est modifiée en conséquence, et la nouvelle vue est envoyée à l'utilisateur.

Prenons un exemple :

- Un joueur envoie une requête au contrôleur “pick2SameTokens(Resources)” où Resources équivaut à 2 diamants.
- Le contrôleur vérifie que l’action est possible en allant chercher dans Bank (qui fait partie du modèle) le nombre de jetons de diamant actuel avec la méthode getResources().
- Si le nombre de diamant est >= 4 :
    - le contrôleur retire 2 jetons de Bank à l’aide de removeResources(Resources) et les ajoute à l’inventaire du joueur à l’aide de addResources(Resources)
    - La vue est modifiée en conséquence dans chaque classe
- Sinon, renvoie un message d’erreur (et l’état du jeu n’a pas été altéré)

Le but est d’interdire à l’utilisateur de modifier directement la structure du jeu, si une action est interdite par le contrôleur, rien ne se passe (juste un message d’erreur en retour pour l’utilisateur). Dans le cadre de notre programme, il y a même une double vérification des erreurs, une au niveau du Controller comme indiqué ci-dessus, et une au niveau de l’interface graphique, pour indiquer à l’utilisateur qu’il n’a pas le droit de faire telle ou telle action.

Dans notre jeu, seront représentés :

- le modèle par : `Bank` et `Player` (et donc indirectement Inventory)
- la vue par : méthodes display de `Bank` et `Player` (symbolise la main du joueur, et des autres joueurs)
- le contrôleur par : `Controller`

L’élément qui compose presque toutes les classes, est l’objet **`Resources`**, que l’on pourrait grossièrement apparenter à un tableau d’entier de ressources `[nb_diamant, nb_saphir, nb_émeraude, nb_rubis, nb_onyx, nb_or]`. Ainsi, ce tuple pourra représenter aussi bien le stock de pierres précieuses d’un joueur, comme son avantage en ressources, ou encore le coût d’une carte développement, **uniformisant donc les échanges de ressources à travers le jeu**.

Afin de nous faciliter l'assimilation des interactions entre les différents modules/classes de notre projet, nous avons décidé de réaliser deux diagrammes de séquence :

- un diagramme représentant la globalité de la partie, de façon peu détaillé mais illustrant les liens entre les modules de notre architecture :

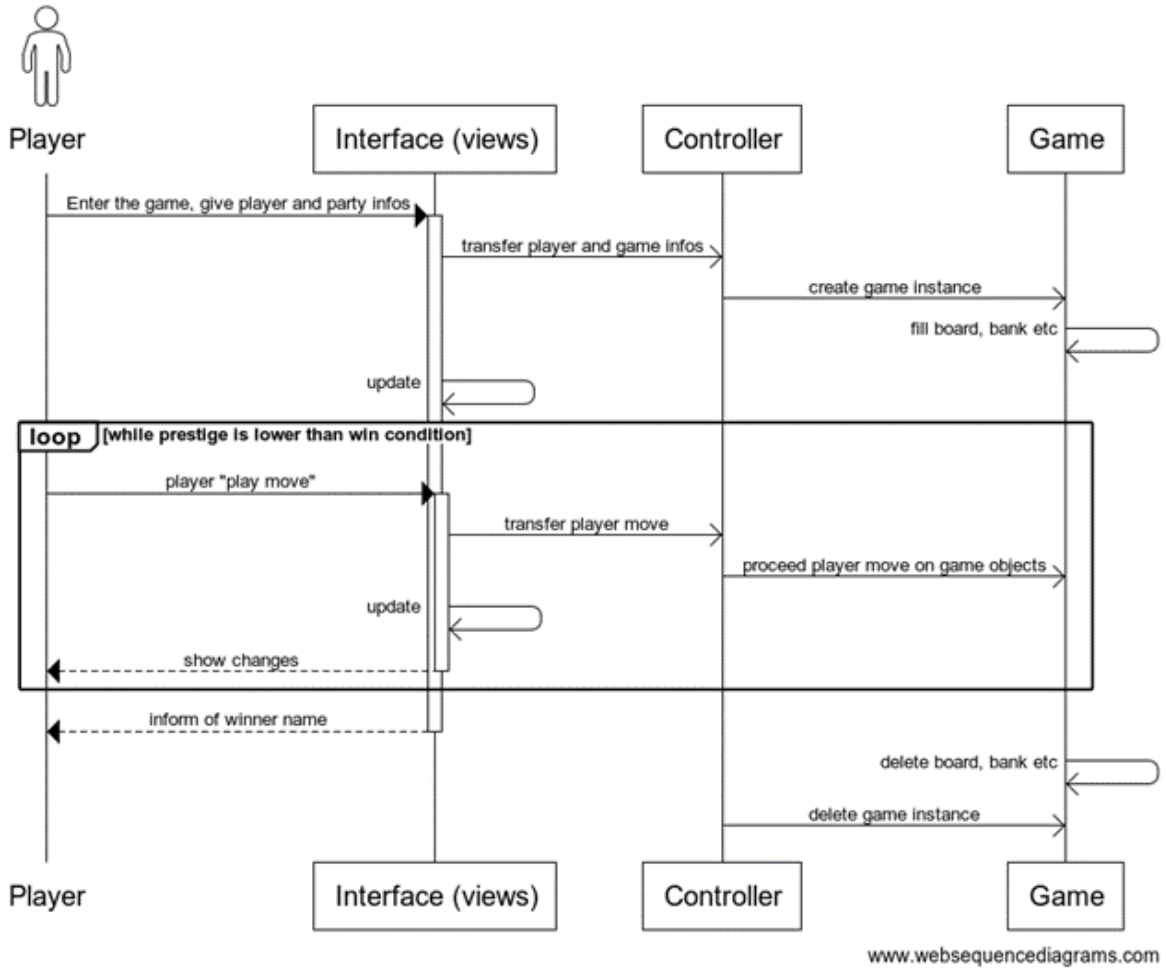


Fig 4 : Diagramme de séquence global

- un exemple d'action que le joueur est amené à effectuer au cours de la partie :

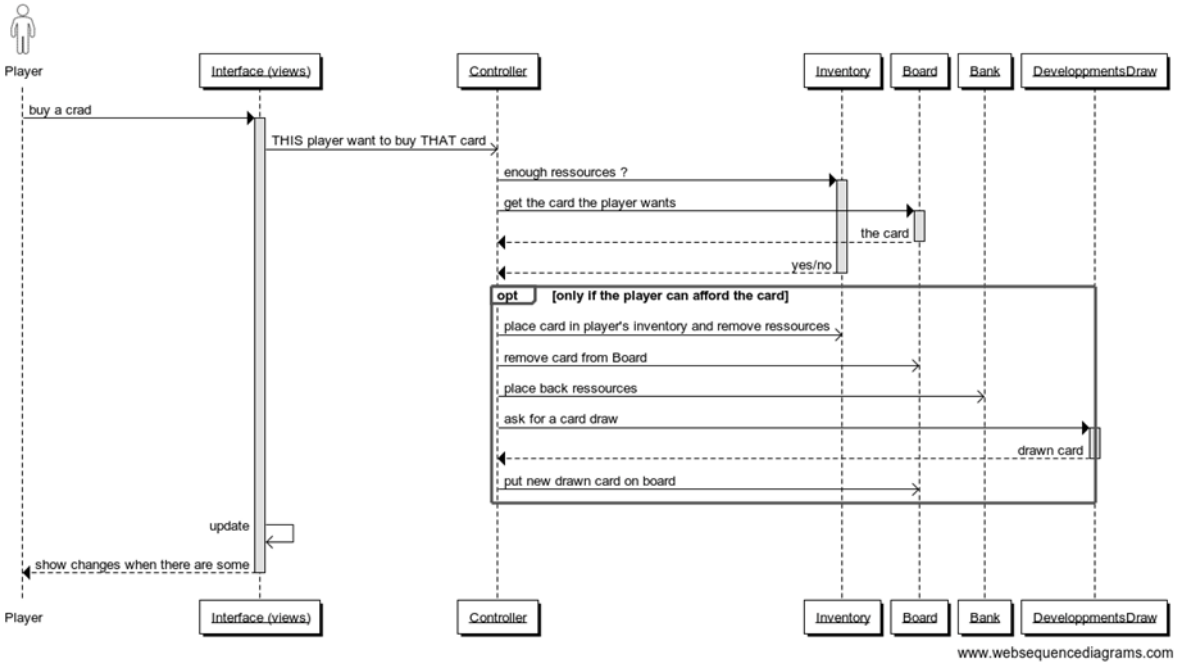
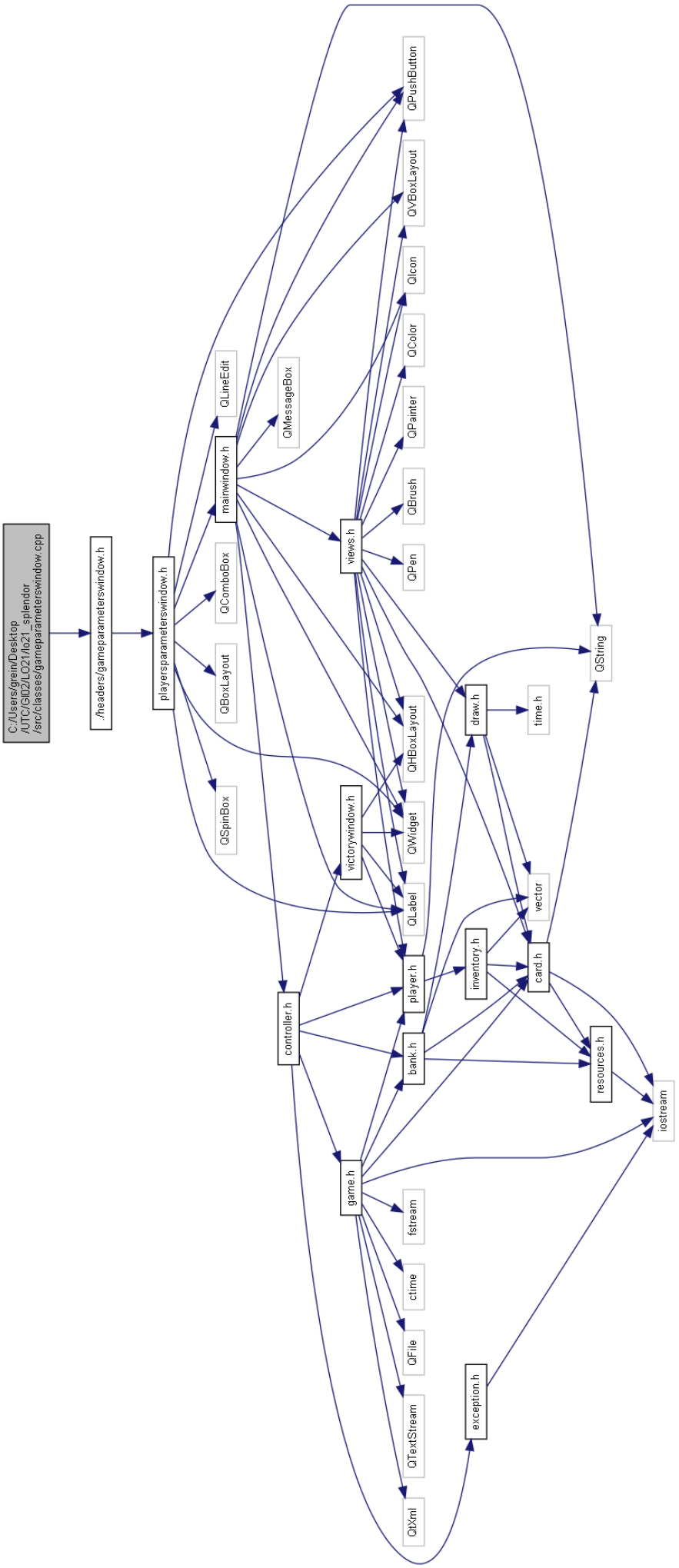


Fig 5 : Diagramme de séquence d'une action type

L'UML suivant n'est pas forcément utile, il peut servir de support pour la compréhension de la suite.

Fig 6 : Graphique mettant en lumière les différentes inclusions dans les classes (donne une idée de l'importante interconnectivité des classes du programme)


C:/Users/grein/Desktop
/UTC/GI02/LO21/lo21_splendor
/src/classes/gameparameterswindow.cpp
./headers/gameparameterswindow.h
parameterswindow.h
QLineEdit
QMessageBox
mainwindow.h
QComboBox
QBoxLayout
QSpinBox
views.h
QPushButton
QVBoxLayout
QIcon
QColor
QPainter
QBrush
QPen
QHBoxLayout
QWidget
QLabel
QString
draw.h
time.h
victorywindow.h
vector
player.h
inventory.h
card.h
controller.h
bank.h
resources.h
iostream
game.h
fstream
ctime
QFile
QTextStream
QtXml
exception.h

Cet affreux graphique (généré par Doxygen) essaie de montrer comment notre architecture fonctionne, avec les dépendances entre les différentes classes. Il faut voir le fonctionnement de notre programme, comme un arbre qui a pour racine, après les différentes fenêtres de configuration de la partie, la classe MainWindow. La classe MainWindow a :

- accès à la classe Controller (controller.h) en lecture seulement , qui contrôle tous les éléments du jeu, et fait avancer la partie
- le contrôle des vues (views.h) des différentes classes de notre programme.

Le controller, de son côté, a pour attribut une instance de la classe Game : c'est le controller qui est le maître du jeu. La classe Game contient toutes les cartes du jeu, mais c'est dans son attribut Bank que les données du jeu y sont stockées (pioches, plateau, ressources).

Ainsi, l'utilisateur n'a accès qu'aux vues, qui sont modifiées par lecture de l'état du jeu que donne le controller.

Remarque : dans le dossier /doc/html figure la documentation générée par Doxygen, mais aussi les graphes de chaque classe générés par Graphviz, si vous en avez le courage, vous pouvez y jeter un œil.

En cascade, un certain nombre de compositions permettent une certaine souplesse et cohérence à notre architecture. L'UML *très simplifié* suivant permet de voir les grandes lignes de cette architecture :

UML generated by PlantUML (code and graph) (aussi disponible ci-après)

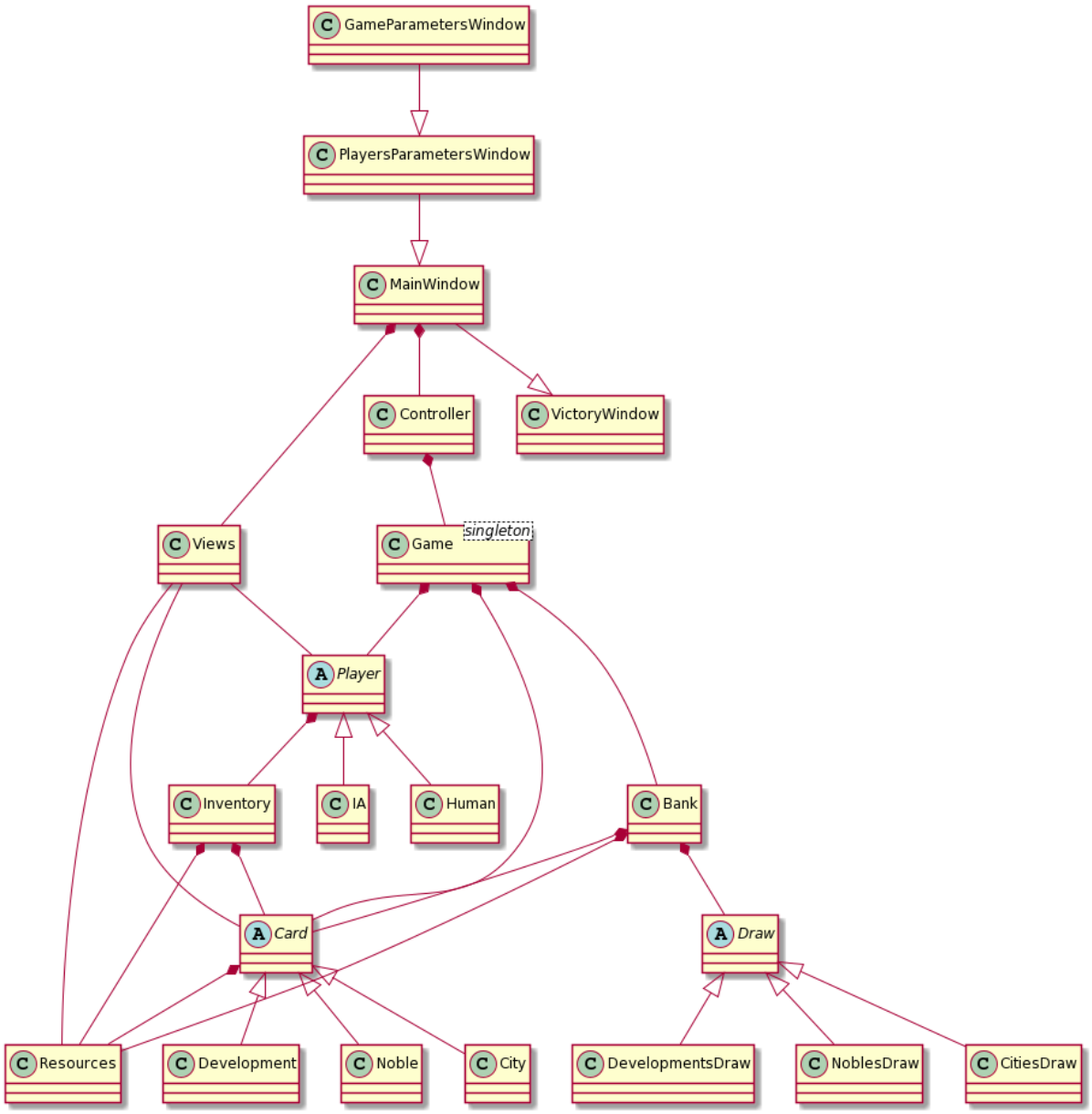


Fig 7 : Diagramme UML très (très) simplifié

Des diagrammes UML plus complets (très légèrement différents de l'implémentation finale, car des choix d'implémentation ont modifié les plans initiaux) figurent dans les rapport précédents, mais nous avons décidé de ne pas les inclure dans ce rapport, pour ne pas le surcharger.

Le dernier point à expliciter dans notre architecture, est le comportement de l'IA, pour l'instant à un seul niveau de difficulté, et au comportement basique (mais redoutable croyez-moi). Le comportement est simple, et a été pensé pour fonctionner peu importe l'extension choisie (par volonté de souplesse majoritairement). Le comportement ne prend

donc pas en compte les cartes spéciales, mais ça n'importe que peu, car ces dernières interviennent sans action nécessaire de la part du joueur.

Le jeu de l'IA suit l'algorithme suivant :

- Acheter une carte (celle qui donne le plus de prestige si il y en a plusieurs) dès que l'IA peut, en achetant en priorité une carte déjà préalablement réservée par l'IA.
- Si 0 carte réservée :
    - Réserver une carte de niveau 2 (celle qui donne le plus de points de prestige)
- Sinon si 1 carte réservée :
    - Réserver une carte quelconque dans le jeu (ce point-là est à améliorer)
- Sinon si 2 cartes réservées :
    - Si somme des ressources est inférieure à 5 :
        - Piocher 3 ressources différentes (soit resource1, resource2 et resource3 les ressources les plus présentes dans la somme des ressources du coût des 2 cartes réservées en déduisant les ressources actuelles de l'IA).
            - Si il n'y a plus de ressource du type voulu, décaler les prédominances de ressources d'un "cran"
                - Si il n'y vraiment plus rien, resourceX est une ressource aléatoire

En ayant un peu d'imagination, on voit clairement quelques failles, qui pourrait faire crasher le programme, car l'IA se trouverait dans une impasse, ou une condition qui n'a pas d'action. C'est pourquoi, si aucune action n'est possible à la fin de n'importe quelle branche, l'IA passe son tour. Ce n'est pas très élégant, mais ça évite le crash, et il faut aussi avoir en tête la très très faible probabilité que l'IA atteigne un des points critiques. Cependant, c'est un point sur lequel on devra se repencher à la reprise du projet.

3.2 SOUPLESSE ET AVANTAGES

L'architecture mise en place permet facilement des évolutions, notamment grâce aux différentes classes abstraites dont le programme est composé (comme Card ou Draw). Cela permet de décliner facilement, et sans risque pour les fonctionnalités déjà implémentées, les différents objets dont le jeu est composé. Nous avons choisi d'implémenter la première extension, "Cités", car c'est celle qui était conseillée dans les règles du jeu pour débuter avec les extensions. Les cartes Cities remplacent les cartes Nobles. Ainsi, par exemple, en créant une classe fille City qui hérite de Card (avec un attribut en plus qui correspond aux points de prestige requis en plus des ressources nécessaires), de la même manière que Noble, on obtient rapidement un nouvel objet fonctionnel. Il suffit alors de créer de la même façon la pioche, la vue, l'attribut de l'inventaire pour stocker les cités, et nous avons presque fini d'implémenter une nouvelle extension. Il faut alors rajouter des méthodes qui

correspondent aux nouvelles règles (par exemple dans le win). L'implémentation de l'extension a pu se faire sans modifier le reste du code, ce qui montre son adaptabilité.

4. PLANNING CONSTATÉ

Plutôt qu'un planning, comme le développement du projet a été très linéaire, nous allons faire une liste des tâches tout au long du projet, avec la date de début et de fin de chaque. Les différents rapports jalons étaient :

- Rapport 1 : 15/11/2021
- Rapport 2 : 13/12/2021
- Rapport 3 : 24/12/2021

Début du projet (~ 15/10/2021) jusqu'au 15/11/2021 :

Conception de l'architecture (UML, diagrammes de séquence), test du jeu sur des plateformes en ligne pour avoir en tête les règles et confirmer la cohérence de notre architecture.

15/11/2021 jusqu'au 13/12/2021 :

Mise en place du répertoire de travail sur GitLab, implémentation des classes, en suivant l'UML relatif au projet.

En parallèle, tests depuis la console des classes implémentées, donc ajout de multiples fonctions de test et d'affichage console.

13/12/2021 jusqu'au 29/12/2021 :

Une fois que le jeu est devenu suffisamment complexe à tester depuis la console (à ce stade, il est possible de lancer une partie, de générer le plateau de jeu, et d'effectuer les différentes actions des joueurs sur les cartes et ressources de la partie, donc le quand le jeu fonctionne déjà grossièrement), nous avons commencé à réfléchir à l'interface graphique. En reprenant le système de BoxLayout de Qt, nous avons donc imaginé l'interface suivante :

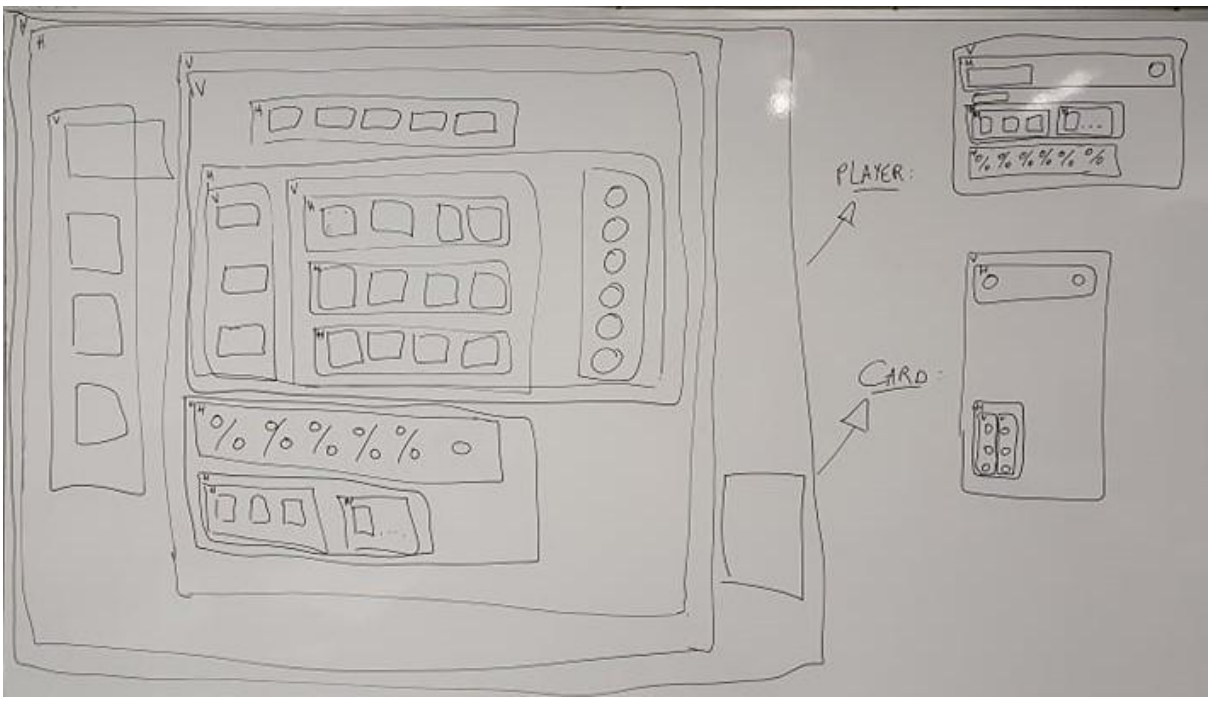


Fig X : Schéma de l'interface pensée autour des systèmes de BoxLayout

Le développement de l'interface graphique a permis de tester une nouvelle fois toutes les fonctionnalités, et de mettre en évidence certains bugs qui nous avaient échappés jusque-là. Cela impliquait naturellement aussi de penser au développement des nouvelles classes liées au module Vue de l'architecture MVC, et au développement des autres fenêtres de paramétrisation de la partie.

Le développement d'une IA a été très rapide, car nous avons manqué de temps, mais nous avons déjà un peu travaillé sur l'amélioration d'une IA suivant des stratégies différentes, qui figure dans la reprise de projet.

Le développement et ajout d'une extension a été très facile, il a fallu ajouter quelques classes, mais grâce à notre architecture, en moins d'un jour nous avons pu implémenter les Cités de Splendor.

29/12/2021 jusqu'au 02/01/2022 :

Rédaction du rapport, de la documentation, nettoyage du code et ajout de commentaires pour la reprise du projet.

5. CONTRIBUTIONS PERSONNELLES

Louis :

- en moyenne 4h par jour (en comptant les jours où je ne me consacre pas au projet, comme les jours où je travaille 12h dessus), à compter du 10 novembre, donc 50 jours, -> 200 heures environ? estimation difficile
- 100% du projet, sauf l'écriture des fichiers XML
- j'ai kiffé, projet très formateur, mais c'est frustrant de ne pas avoir plus de temps à consacrer au projet, car j'ai l'impression d'avoir bâclé le travail par moments
- Note personnelle : je pense qu'il serait pertinent, au vu de l'échec de certains groupes (il n'y a pas du tout que notre groupe dans ce cas) à parvenir à travailler réellement en équipe, de faire un sondage ou proposer à ceux qu'ils le souhaitent, un retour d'expérience sur le projet, et des différentes améliorations possibles pour rendre le projet plus attrayant / d'inciter plus le travail en groupe / proposer une découpe différente du projet selon le différents groupe. Car selon moi, le problème principal que nous avons rencontré en tant que groupe, c'est la montagne de tâches à réaliser, qui peut faire peur aux premiers abords, et qui est difficilement rattrapable si on n'a pas participé et vu tous les aspects du projet, et que la découpe en tâches individuelles est très compliquée à mettre en place, car en parallèle, nous devons nous former à tout un tas de chose : la communication, la gestion d'un projet de cette envergure, apprendre l'utilisation de Git, ou soit simplement juste les éléments abordés en cours de LO21, le tout, en parallèle de nos autres UVs respectives, qui nous donnent des plages de disponibilités dans le semestre tous différentes. J'exprimerai tout cela en détail dans l'évaluation des enseignements à la fin du semestre bien sûr, mais je profite de la fraîcheur ici.

Alice:

- première partie avec conception de l'UML, recherche des design pattern, compréhension du jeu (environ 15 heures)
- extension (fichier XML, nouvelles classes, nouvelles fonctionnalités etc…) (environ 18 heures)
- 2 premières fenêtres Qt (environ 20 heures pour pouvoir prendre en main QT et les rendre fonctionnelles)
- chercher solution pour le chemin absolu/ relatif (test avec DESTDIR= $$PWD mais pas abouti…) (2 heures)
- difficulté d'organisation et de communication au sein du groupe qui ont rendu le projet assez difficile à gérer (difficile de s'impliquer plus, je n'ai pas participé autant que ce que j'aurais voulu) et plutôt stressant. J'estime avoir participé à seulement 20/25% du projet.

Léna:

- première partie avec conception de l'UML, recherche des design pattern, analyse des besoins du jeu (environ 15h)
- fichier XML jeu de base + recherches sur l'interface graphique (mes codes pour l'interface graphique n'ont pas été pris mais j'ai travaillé environ 40h dessus, formation à QT comprise)
- recherches sur l'IA (environ 12h)
- Les problèmes de communication et d'organisation du groupe ont fait que je n'ai pas progressé autant que j'aurais voulu en POO et ont rendu le projet anxiogène. C'est néanmoins très formateur et cela nous permettra de ne pas reproduire les mêmes erreurs. Je pense avoir participé à 20% du projet.

6. REPRISE DU PROJET

IMPORTANT : pour pouvoir tester le programme, il faut au préalable **modifier le chemin d'accès aux fichiers XML** dans le fichier **src/classes/game.cpp**, lignes 34 et 40, en remplaçant le chemin d'accès sur ma machine à votre chemin d'accès.

GitLab: https://gitlab.utc.fr/lstallkn/lo21_splendor

Les tâches qui restent à faire sont :

- Mettre un chemin relatif et non absolu pour les fichiers XML
- Développement d'un niveau supérieur d'IA :
  Une IA qui est capable de calculer l'intérêt d'une carte en fonction de son coût (nombre et sortes de pierres) et de ses avantages (sortes de pierres et nombre de prestiges) par rapport aux autres cartes présentes et qui adapte sa stratégie de jeu aux cartes présentes sur le plateau, nous aurions voulu développer deux stratégies :
  - Stratégie du rush : il y a des pierres qui prédominent sur le plateau (l'émeraude et le saphir sont favorisés car ils s'engendrent eux-mêmes). L'objectif de cette stratégie est de récupérer les cartes développement de niveau 3 en réservant des cartes de niveau 2 qui permettent avec leur avantage d'acheter les cartes niveau 3. L'IA préfèrera prendre 3 jetons plutôt que de prendre une carte niveau 1.
  - Stratégie avec un plateau homogène : prendre des cartes de niveau 1 afin d'avoir des bonus de toutes les sortes de pierres et continuer jusqu'à pouvoir acheter des cartes de niveau 2. Acheter des cartes de niveau 2 tant que possible sinon prendre des cartes de niveau 1.
    Algorithme envisagé :
    - TANT QUE on a avantage pierres (saphir ou rubis ou…) = 0
      - POUR chaque sorte de pierre

            - identifier la carte niveau 1 : avec comme avantage cette pierre et la plus intéressante
            - TANT QU'on ne peut pas l'acheter ET qu'elle est toujours présente sur le plateau
                - piocher les jetons permettant d'acheter cette carte
    - TANT QUE le jeu n'est pas fini
        - TANT QU'on ne peut pas acheter carte niveau 2
            - identifier carte niveau 2 la plus intéressante en fonction de nos ressources
            - analyser les ressources manquante à son achat
            - SI les ressources manquantes sont disponibles dans la pioche et piochables en 1 tour
                - piocher les jetons
            - SINON
                - identifier la pierre dont la différence entre notre avantage et le coût de la carte niveau 2 est la plus grande
                - identifier la carte niveau 1 : avec comme avantage cette pierre et la plus intéressante en fonction de nos ressources
                - TANT QU'on ne peut pas l'acheter ET qu'elle est toujours présente sur le plateau
                    - piocher les jetons permettant d'acheter cette carte

- Ajout d'une autre extension
- Amélioration de l'interface graphique
- Cacher les cartes réservées des autres joueurs quand c'est pas leur tour
- Permettre de choisir de quels jetons le joueur souhaite se défausser s'il a plus de 10 jetons
- Permettre de choisir quel noble accepter si un joueur est éligible à la visite de plusieurs nobles (de même pour la cité avec l'extension)
- Faire un peu de ménage dans le code, notamment des attributs de classe qui sont inutiles
- Système de chronométrage de chaque tour de chaque joueur, ainsi que la durée de la partie
- Système de sauvegarde, et reprise de partie en cours
- Système de statistiques des joueurs

Excellent site avec des exemples simples pour comprendre les design pattern : https://refactoring.guru/fr. On y trouve également des exemples d'implémentation de ces design pattern en C++.

7. CONTACTS

Pour tout question relative au projet, contacter :

- Léna BAILLET - lena.baillet@etu.utc.fr
- Alice DEMONFAUCON - alice.demonfaucon@etu.utc.fr
- Louis GREINER - louis.greiner@etu.utc.fr
- Myriem KHAL - myriem.khal@etu.utc.fr
- Lucas STALLKNECHT- lucas.stallknecht@etu.utc.fr