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

LO21 **Projet Splendor - Rapport 1**

Application de jeu de société en C++

P21

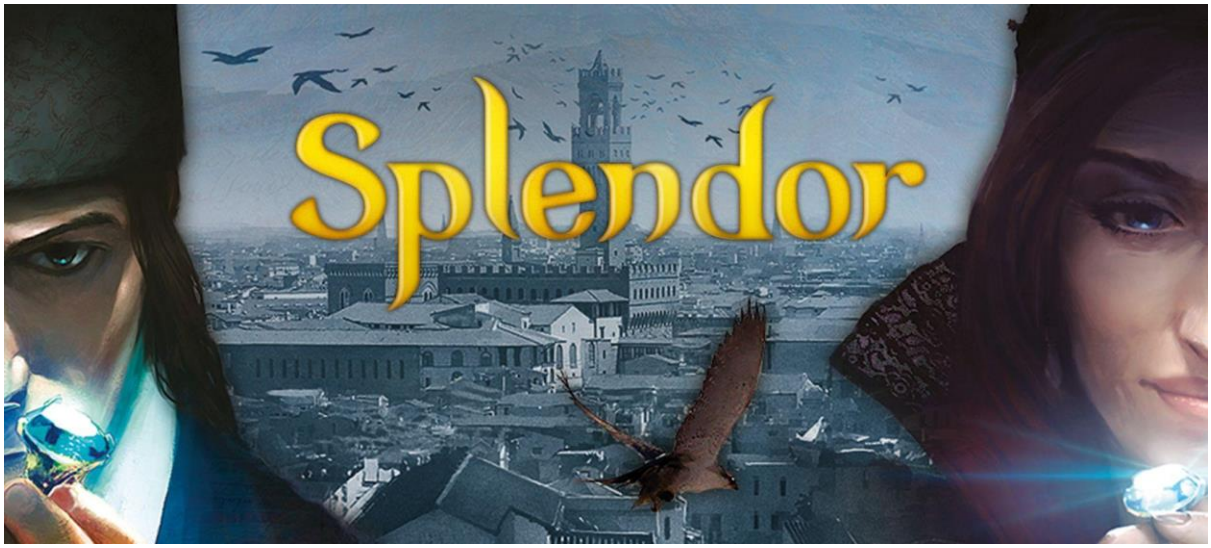

TABLE DES MATIÈRES

1. INTRODUCTION

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 permettra de jouer des parties de 2 à 4 joueurs, opposant des humains ou des IA. À terme, l'application permettra de jouer avec des extensions, et de sauvegarder et reprendre une partie en cours.

2. JEU

2.1 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

2.2 PHASES D'UN TOUR

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

3. DÉROULEMENT DU JEU

Le déroulement d'une partie se résume en trois points :

1) Initialisation du jeu (mise en place du plateau)
2) Tours de jeu (voir le point 2.2)
3) Un joueur dépasse le seuil de points de prestige, fin du tour et désignation du vainqueur.

3.1 UML

Différents choix de conception ont abouti à la conception du diagramme UML suivant, schématisant les différentes entités et les associations entre elles tout au long d'une partie. Tout d'abord, le découpage en différentes classes est primordial pour représenter au mieux les différentes entités et leurs relations, permettre une plus grande souplesse et une gestion plus claire de notre programme.

L'utilisation de design patterns est incontournable, ces patrons de conception permettent de répondre à des problèmes courants de conception logiciels d'une bonne façon.

Nous avons déjà décidé d'utiliser donc une architecture MVC (Model View Controller) pour notre application, et le design pattern Singleton pour la classe Game.

Nous avons réalisé 2 versions d'UML : une version complète, assez lourde à lire et comprendre, et une version "simplifiée", où nous avons retiré les méthodes de getters, setters et de display, pour se concentrer et faciliter la lecture des associations entre les classes.

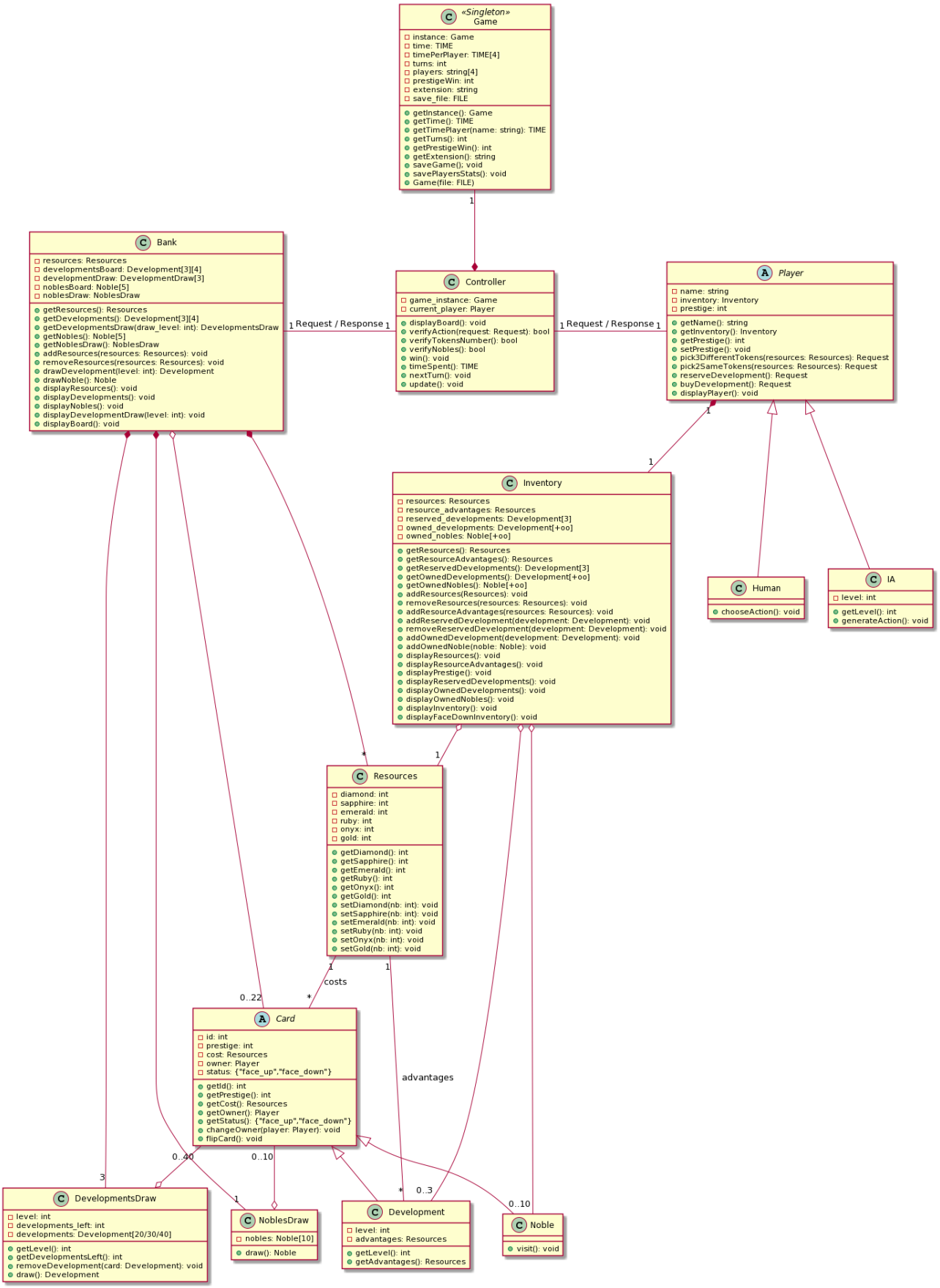

Fig 3 : UML version 2.0

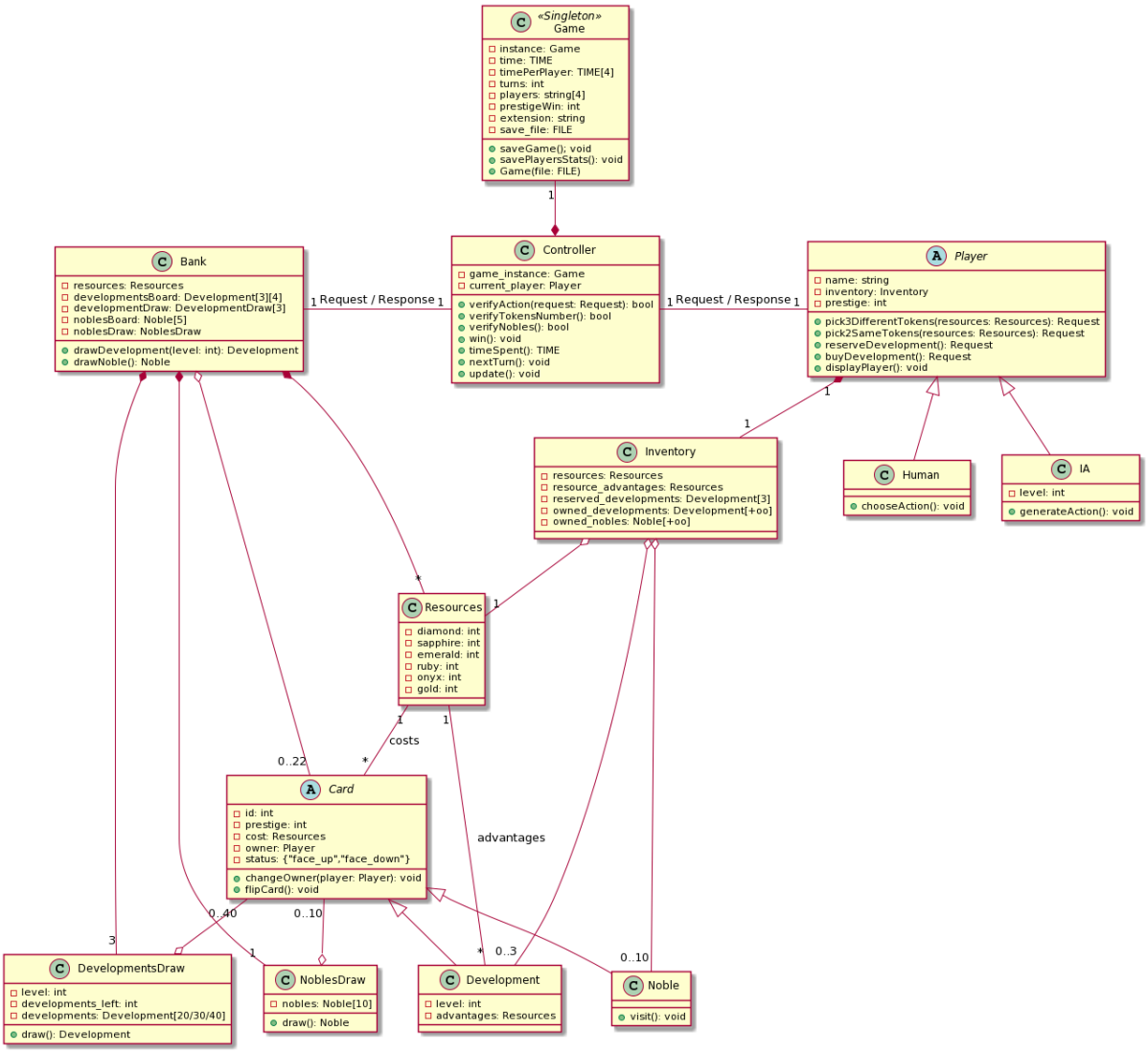

Fig 4 : UML version 2.0 - simplifié pour faciliter la lecture des associations (retrait des getters / setters / displays)

3.2 ARCHITECTURE GLOBALE

Dans cette partie, nous allons tenter d'expliquer pourquoi nous avons pensé concevoir notre architecture représentée par le diagramme UML précédent, et comment cela fonctionne.

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)

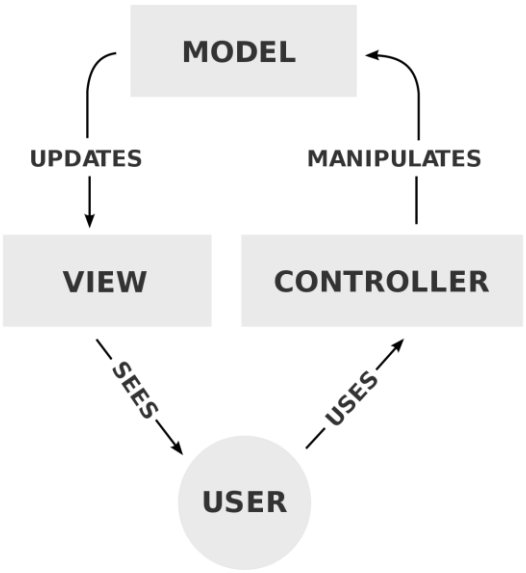

Fig 5 : Schéma d'une architecture MVC

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 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**.

Le design pattern Singleton est utilisé pour **assurer l'unicité de l'instance de la partie en cours**. C'est dans la classe Game que nous allons stocker l'état actuel du jeu (mis à jour à la fin de chaque tour). Pour cela nous avons pensé utiliser 2 variables parallèles:

- **"Etat n-1"** : Etat du jeu au début du tour (qu'on ne modifiera qu'au moment où le joueur passe son tour) : sert de sauvegarde pour pouvoir permettre au joueur de "undo" une action durant son tour. C'est **cette version de la sauvegarde qui sera gardée si les joueurs veulent arrêter une partie en cours**.
- **"Etat n"** : Etat du jeu actuel, modifié en temps réel : **si un joueur veut "undo" une action, Etat n est remis à jour avec l'Etat n-1**.

Nous pensons également implémenter un système de sauvegarde et de "undo" via le design pattern Memento.

3.3 DIAGRAMMES DE SÉQUENCE

Afin de nous faciliter l'assimilation des intéractions 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 :

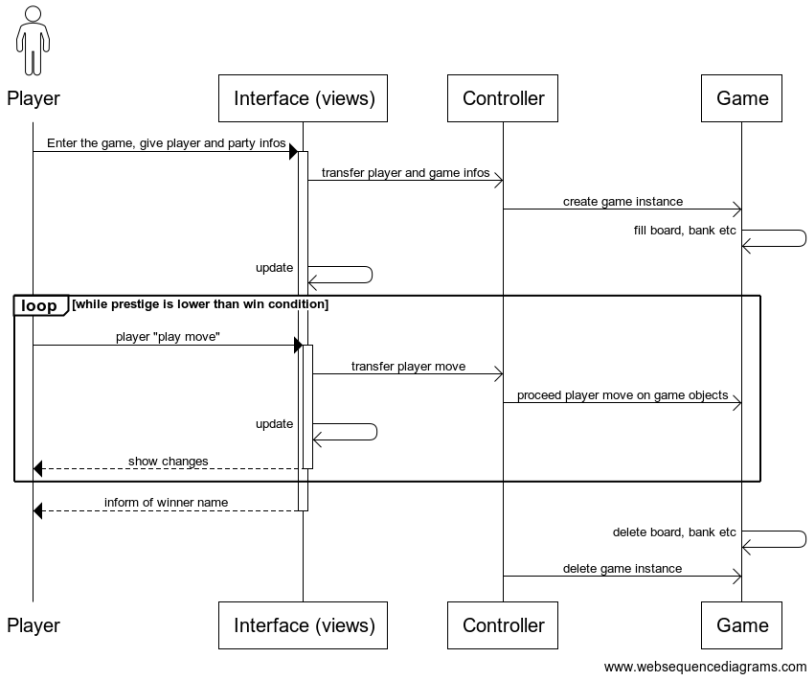

Fig 6 : Diagramme de séquence global

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

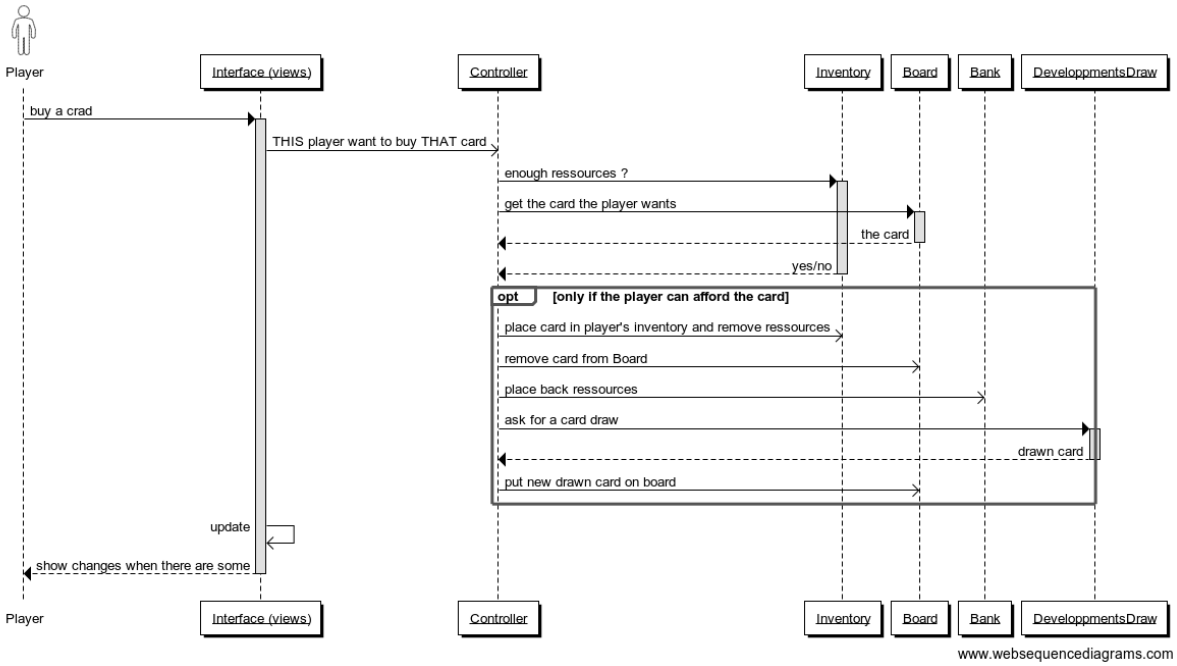

Fig 7 : Diagramme de séquence d'une action type

4. TÂCHES EFFECTUÉES ET EN COURS

Un peu naïvement, nous avons décidé d'un agenda commun, qui sera très certainement amené à être modifié :

I. Définition des classes
II. UML (portées, méthodes, associations, cardinalités)
III. Recherches designs patterns (répartis selon les membres de l'équipe)

DEADLINE : 15 novembre

La **mise en place de l'architecture MVC** est, selon nous, le plus grand défi de ce projet, car une fois celle-ci fonctionnelle, la base de notre projet sera solide.

En parallèle, nous réalisons un vrai travail de **recherche sur les design patterns** suivants, pour les utiliser par la suite dans le développement de notre application. Si ce n'est pas tous, beaucoup nous sont inconnus, alors nous prenons le temps qui nous est nécessaire pour comprendre ces nouveaux concepts, avant de les inclure à notre projet. Peut-être ne sont-ils pas tous utiles, cela enrichira dans tous les cas notre culture informatique:

- Architecture MVC (expliquée précédemment)

- 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 "Game"

- Decorator: patron de conception structurel qui permet d'**affecter dynamiquement de nouveaux comportements à des objets** en les plaçant dans des decorators (emballeurs) qui implémentent ces comportements. Au lieu de créer une classe pour une fonctionnalité bien précise, on ajoute plutôt des fonctionnalités à une classe "de base".

- Factory: permet d'**instancier des objets dont le type est dérivé d'un type abstrait** et de séparer la création d'objets dérivant d'une classe mère de leur utilisation.

- Abstract factory: Utile 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 FabriqueAbstraite

- Builder: patron de conception de création qui permet de construire des objets complexes étape par étape. Il permet de se débarrasser d'un constructeur avec trop de paramètres facultatifs.

- Bridge: patron de conception structurel qui **permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies** (abstraction et implémentation) qui peuvent évoluer indépendamment l'une de l'autre. Utilisé lors d'héritage : on crée un pont entre deux classes qui contiennent des classes filles.

- Composite: patron de conception structurel qui **permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels**. Réservée aux applications dont la structure principale peut être représentée sous la forme d'une arborescence (forte hiérarchie)

- Iterator: patron de conception comportemental qui **permet de parcourir les éléments d'une collection sans révéler sa représentation interne** (liste, pile, arbre, etc.)

- Template method: patron de conception comportemental qui **permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure**. Utile pour gérer par exemple différents formats très similaires mais légèrement différents, et qui donc nécessitent des codes légèrement différents. On découpe alors l'algo en une série d'étapes, qui seront des méthodes regroupées dans une grande méthode "template method".

- Adapter: patron de conception structurel qui **permet de faire collaborer des objets ayant des interfaces normalement incompatibles**. C'est un objet spécial qui convertit l'interface d'un objet afin qu'un autre objet puisse le comprendre.

- Visitor: patron de conception comportemental qui **permet de séparer les algorithmes des objets sur lesquels ils opèrent**. Permet de placer un nouveau comportement dans une classe séparée que l'on appelle visiteur, plutôt que de l'intégrer dans des classes existantes. Il adapte son comportement en fonction de la classe qu'il est en train de visiter.

- Strategy: patron de conception comportemental qui **permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables**. Permet de prendre une classe dotée d'un comportement spécifique mais qui l'exécute de différentes façons, et de décomposer ses algorithmes en classes séparées appelées stratégies. Plutôt que de s'occuper de la tâche, la classe

originale (le contexte) la délègue à l'objet stratégie associé. **On peut alors modifier les stratégies sans modifier la classe de base**.

- Façade: patron de conception structurel qui **procure une interface offrant un accès simplifié à une librairie**, un framework ou à n'importe quel ensemble complexe de classes. Une façade se révèle très pratique si votre application n'a besoin que d'une partie des fonctionnalités d'une librairie sophistiquée parmi les nombreuses qu'elle propose.

- Memento: patron de conception comportemental qui **permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation**. Avant d'effectuer une action, l'application prend un instantané (snapshot) du dernier état des objets. Il peut être utilisé plus tard pour rétablir les objets dans leur ancien état. Plutôt que d'essayer de copier l'état de l'élément depuis « l'extérieur », la classe de l'élément peut prendre la photo elle-même, car elle a accès à son propre état. **Memento permet de stocker la copie de l'état de l'objet dans un objet spécial appelé mémento**. Son contenu n'est accessible que pour l'objet qui l'a créé. Les autres objets peuvent communiquer avec les mémentos via une interface limitée qui leur permet de récupérer certaines métadonnées de la photo (date de création, nom de l'action effectuée, etc.), mais pas l'état de l'objet original contenu dans la photo. Nous réfléchissons à utiliser ce design pattern pour restaurer les données du joueur ("undo" une action).

5. TÂCHES FUTURES

Une fois l'UML, et l'architecture globale de l'application terminée, notre carnet de route est défini comme suivant (amené à changer) :

IV. Finir l'UML et l'architecture globale, ce qui comprend:
   1. Prise en compte des remarques suite au premier rendu et modifications
   2. Détermination des variables CONST et STATIC: pour cela bien revoir ces mots clés pour pouvoir les placer de manière pertinente et utile

V. Implémentation des classes
   1. Commencer dès la semaine prochaine au moins les classes et la structure de base, même si la partie "théorique" n'est pas finie, et inclure les dernières modifications que l'on aura faites sur l'UML

VI. Implémentation des nouveaux design patterns sélectionnés:

1. Comprendre quels design pattern sont utiles pour notre implémentation
2. Les implémenter

VII. Début de l'interface graphique

VIII. Jouable + premiers tests en human only

IX. Sauvegarde (persistance) + Statistiques joueurs

X. Développement IA

XI. Extensions

DEADLINE : 6 décembre

XII. Correction bugs

DEADLINE : 20 décembre

XIII. Rendu final

DEADLINE : 3 janvier

6. CONVENTIONS ET LOGICIELS UTILISÉS

Nous avons décidé, pour que notre **code soit uniforme**, et faciliter ainsi sa compréhension, que le langage, de programmation, des différents diagrammes et schémas, est **l'anglais**.

De plus, nous avons décidé de l'ensemble des conventions suivantes pour la syntaxe :

- format des :
  - variables : `nom_variable`
  - fonctions : `nomFonction(....)`
  - classes : `class Maclasse{ … }`
  - fichiers : `nom_fichier.extension`
  - constantes : `#define CONSTANTE 3.14159`
- pas d'accent
- les variables qui ont vocation à être plurielles ont un nom au PLURIEL, i.e, `owned_cardS` (même si au début on n'en a pas)

Les différents logiciels et applications utilisés jusqu'à présent sont :

- Gitlab: logiciel de gestion de versions
- PlantUML : permet de créer des diagrammes UML via un pseudo langage.
- draw.io (ou aussi websequencediagrams.com, diagrams.net) : permet de créer des diagrammes de séquence via une interface graphique
- Qt : framework utilisé pour concevoir des interfaces graphiques, et l'architecture d'une application en utilisant des mécanismes de signaux et slots
- Visual Studio Code : éditeur de texte, compilation "à la main" du C++

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