PROJET Splendor
Compte Rendu Final

Réalisé au sein de l'unité de valeur LO21

Programmation et conception orientée objet

BORDEAU Cléa

Habert Thomas

LEFEBVRE Philippe

VALTY Eugène

Sommaire

Introduction :

Ceci est le rapport final de notre projet de réalisation du jeu Splendor. Il a été réalisé dans le cadre de L'UV LO21 à l'UTC par Thomas HABERT, Philippe LEFEBVRE, Eugène VALTY et Cléa BORDEAU. Ce rapport accompagne l'ensemble du code source du projet, une documentation générée en Doxygen ainsi qu'une vidéo de présentation de l'application.

Ce rapport sera sujet de descriptions détaillées de sa conception tant bien dans le développement du code que dans la répartition du travail et l'organisation au sein du groupe.

Notre application :

Notre application permet de réaliser des parties de 2 à 4 joueurs humain ou virtuel. On propose également au début de la partie d'utiliser une extension ou non.

L'interface est séparée en deux parties: La partie plateau et la partie consacrée aux joueurs.

Le plateau est composé de cartes nobles (situées en haut à droite) et les cartes ressources (en dessous des cartes nobles) qui sont triées par niveaux (niveau 1 puis 2 puis 3).

Nous pouvons aussi observer les jetons que possède la banque à gauche du plateau. Sur chaque jeton est écrit un numéro qui correspond au nombre de jetons de ce type disponible dans la banque.

Dans la partie gauche de l'interface nous avons représentées les données des joueurs: Cartes, jetons, cartes réservées si on appuie sur le bouton "Montrer les cartes réservées", score actuel, nom, type de joueur Humain ou IA.

Il est aussi possible de passer à une version du jeu avec l'extension Cités d'activée, qui remplace les nobles par des cités, et change la condition de fin de partie et de victoire.

Tâches non accomplies:

Nous n'avons malheureusement pas eu le temps d'implémenter les différents objectifs optionnels proposés par le sujet, tels que la sauvegarde des parties, ou le log des actions de joueurs.

Nous pensons cependant que leur implémentation ne devrait pas être très difficile avec notre architecture actuelle, comme par exemple avec l'ajout du design pattern Visitor pour la création de sauvegardes.

Description de l'architecture :

Vous trouverez sur la page 7 l'UML du projet. Celui-ci a subi de nombreuses modifications tout au long de notre projet comme vous pourrez le voir grâce aux rapports 1,2,3. Ceci nous à fait prendre conscience qu'il est très difficile d'aboutir directement à une architecture parfaite, et qu'il faut se confronter aux problèmes pour finir par obtenir quelque chose de qualitatif.

Design patterns utilisés :

Pour ce projet nous utilisons divers design patterns:

Nous utilisons un iterator pour parcourir DrawPile. L'itérateur permet l'encapsulation du détail du fonctionnement de la structure et passe plusieurs méthodes simples qui permettent d'accéder aux éléments de la collection.

Nous avons également utilisé le design pattern Vue/Controller qui s'occupe de la gestion des différentes phases du jeu, pour l'interface utilisateur. Enfin nous avons utilisé le design pattern singleton pour la classe Game afin de restreindre l'instanciation de la classe à un seul objet.

Enfin, nous avons utilisé à plusieurs reprises les design patterns Strategy afin de faciliter l'héritage de certaines classes et de faciliter le changement des implémentations.

Souplesse de l'architecture:

Toutes les classes utilisées dans l'extension que nous avons décidé de réaliser (Cités) dérivent d'une classe existante de l'architecture. De cette façon, nous avons pu remplacer que le strict nécessaire dans l'architecture pour obtenir ce résultat.

Chaque classe est donc remplaçable par une version modifiée qui hérite de la classe d'origine, et grâce au polymorphisme et au down/upcasting, le reste de l'architecture peut rester identique et s'adapter à ce changement avec seulement peu de modifications.

Chaque élément de l'architecture est responsable de son fonctionnement et rien de plus, ce qui permet de remplacer d'être précis lors de l'extension des classes pour la nouvelle tâche que l'on souhaite exécuter.

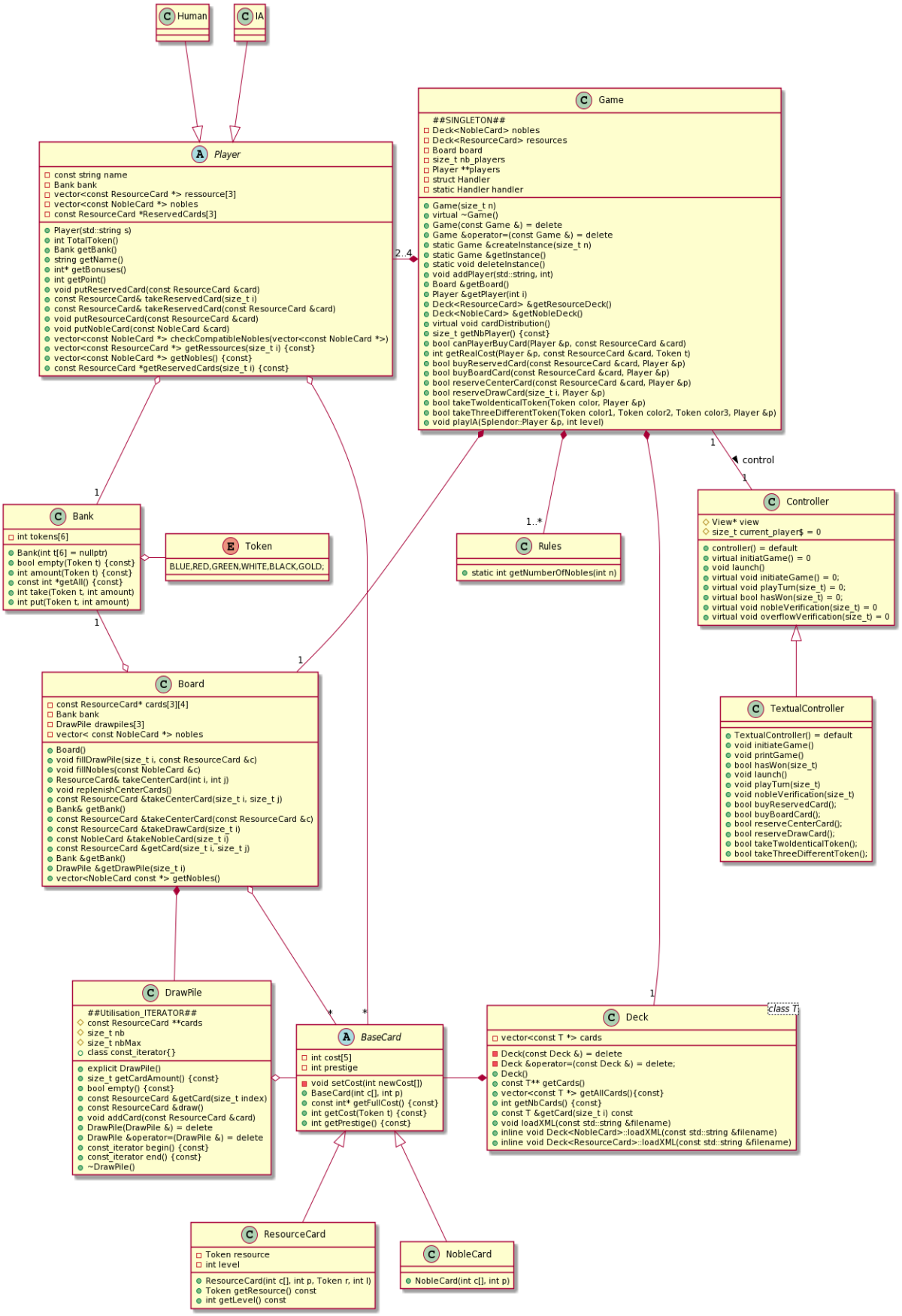
De plus, nous avons la possibilité grâce au design pattern Modèle/Contrôleur/Vue de changer de Modèle sans nécessairement affecter les autres parties de manière disruptive.

L'UML suivant n'inclut pas les classes réalisées pour l'extension , la raison étant que ces classes ne sont que de simples héritages des classes de base, ce qui n'a pas grand intérêt pour l'architecture globale.

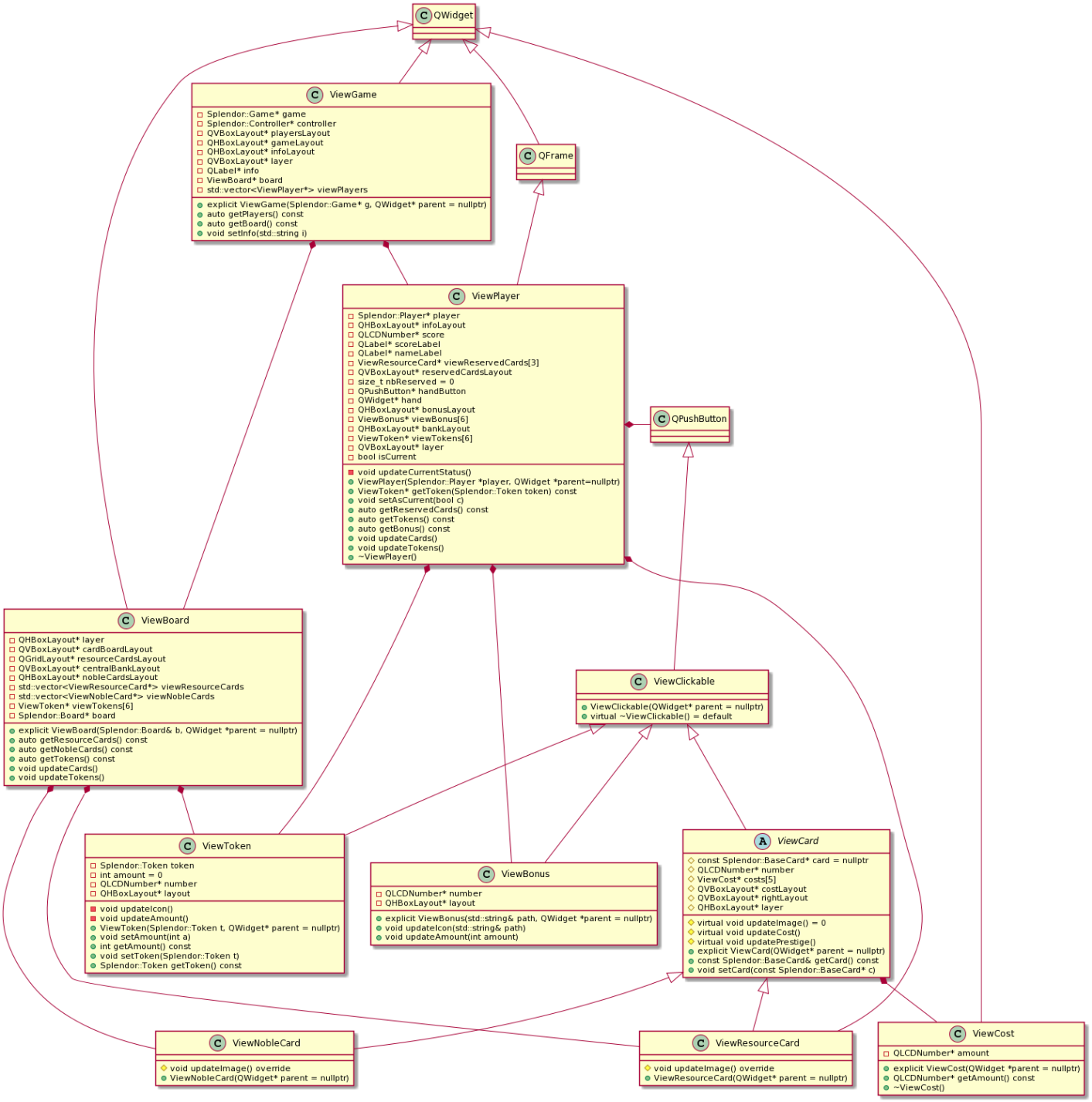
Le choix de difficulté des IA:

Il était précisé dans le sujet du projet que les IA devaient pouvoir avoir plusieurs niveaux de difficulté. Nous avons pris en compte cette consigne dans la fonction *playIA()* du controller mais nous n'avons pas pu l'ajouter à la vue finale. Ainsi le choix de la difficulté des IA est bien présent dans le code mais n'est pas utilisable en jeu.

UML de la partie fonctionnelle



UML des vues


QWidget
ViewGame
- Splendor::Game* game
- Splendor::Controller* controller
- QVBoxLayout* playersLayout
- QHBoxLayout* gameLayout
- QHBoxLayout* infoLayout
- QVBoxLayout* layer
- QLabel* info
- ViewBoard* board
- std::vector<ViewPlayer*> viewPlayers
+ explicit ViewGame(Splendor::Game* g, QWidget* parent = nullptr)
+ auto getPlayers() const
+ auto getBoard() const
+ void setInfo(std::string i)
QFrame
ViewPlayer
- Splendor::Player* player
- QHBoxLayout* infoLayout
- QLCDNumber* score
- QLabel* scoreLabel
- QLabel* nameLabel
- ViewResourceCard* viewReservedCards[3]
- QVBoxLayout* reservedCardsLayout
- size_t nbReserved = 0
- QPushButton* handButton
- QWidget* hand
- QHBoxLayout* bonusLayout
- ViewBonus* viewBonus[6]
- QHBoxLayout* bankLayout
- ViewToken* viewTokens[6]
- QVBoxLayout* layer
- bool isCurrent
- void updateCurrentStatus()
+ ViewPlayer(Splendor::Player *player, QWidget *parent=nullptr)
+ ViewToken* getToken(Splendor::Token token) const
+ void setAsCurrent(bool c)
+ auto getReservedCards() const
+ auto getTokens() const
+ auto getBonus() const
+ void updateCards()
+ void updateTokens()
+ ~ViewPlayer()
QPushButton
ViewBoard
- QHBoxLayout* layer
- QVBoxLayout* cardBoardLayout
- QGridLayout* resourceCardsLayout
- QVBoxLayout* centralBankLayout
- QHBoxLayout* nobleCardsLayout
- std::vector<ViewResourceCard*> viewResourceCards
- std::vector<ViewNobleCard*> viewNobleCards
- ViewToken* viewTokens[6]
- Splendor::Board* board
+ explicit ViewBoard(Splendor::Board& b, QWidget *parent = nullptr)
+ auto getResourceCards() const
+ auto getNobleCards() const
+ auto getTokens() const
+ void updateCards()
+ void updateTokens()
ViewClickable
+ ViewClickable(QWidget* parent = nullptr)
+ virtual ~ViewClickable() = default
ViewToken
- Splendor::Token token
- int amount = 0
- QLCDNumber* number
- QHBoxLayout* layout
- void updateIcon()
- void updateAmount()
+ ViewToken(Splendor::Token t, QWidget* parent = nullptr)
+ void setAmount(int a)
+ int getAmount() const
+ void setToken(Splendor::Token t)
+ Splendor::Token getToken() const
ViewBonus
- QLCDNumber* number
- QHBoxLayout* layout
+ explicit ViewBonus(std::string& path, QWidget *parent = nullptr)
+ void updateIcon(std::string& path)
+ void updateAmount(int amount)
ViewCard
const Splendor::BaseCard* card = nullptr
QLCDNumber* number
ViewCost* costs[5]
QVBoxLayout* costLayout
QVBoxLayout* rightLayout
QHBoxLayout* layer
virtual void updateImage() = 0
virtual void updateCost()
virtual void updatePrestige()
+ explicit ViewCard(QWidget* parent = nullptr)
+ const Splendor::BaseCard& getCard() const
+ void setCard(const Splendor::BaseCard* c)
ViewNobleCard
void updateImage() override
+ ViewNobleCard(QWidget* parent = nullptr)
ViewResourceCard
void updateImage() override
+ ViewResourceCard(QWidget* parent = nullptr)
ViewCost
- QLCDNumber* amount
+ explicit ViewCost(QWidget *parent = nullptr)
+ QLCDNumber* getAmount() const
+ ~ViewCost()

Description du planning et contributions:

Description du planning constaté du projet

La réalisation de notre projet s'est faite en trois grandes étapes.

1. Tout d'abord la compréhension du problème et la conceptualisation de l'architecture et donc de l'UML, des classes et des méthodes.

2. Ensuite la réalisation du code basé sur l'architecture précédemment définit. Lors de cette étape, il a été inévitable de revoir l'UML et de le mettre à jour en fonction des problèmes que l'on rencontrait au fur et à mesure du développement de notre code (exemple : suppression des classes actions) .

3. Enfin, nous avons réuni les codes de chacun des membres du groupe afin d'obtenir un code fonctionnel. Même si nous nous étions mis d'accord sur des normes à respecter, il a été inévitable de passer un temps considérable à débugger les codes. Une fois que nous avons obtenu un code fonctionnel, nous avons réalisé l'UML pour les vues puis l'interface utilisateur, afin d'obtenir notre application.

Contribution personnelle de chacun des membres du groupe

Thomas : Thomas a concentré ses efforts sur les classes Player ainsi que le controller codés en C++. Ces classes représentent selon nous des classes complexes de la partie fonctionnelle, d'où le fait qu'elles ont pris du temps à être réalisées et que nous les avons réalisées à plusieurs. Il a également travaillé sur la mise en place des extensions ("Cités" mais aussi "Bastion" que nous n'avons pas pu achever). Enfin Thomas a rédigé un rapport.

% de contribution estimé :	20%
nombre d'heures de travail estimé :	35h

Philippe : Philippe a réalisé plusieurs classes dans la partie fonctionnelle telles que Deck, ResourceCard, NobleCard, BaseCard. Il a concentré une grande partie de son travail sur la réalisation des vues, donc de l'interface utilisateur et de la conception de l'UML des vues. Il a aussi participé à la conception du Controller QT, notamment sur certaines actions comme la prise de jetons, leur retour dans le cas où un joueur en a plus de 10, et la réception des nobles. Il a apporté son aide à l'implémentation de l'extension et à sa réalisation. Enfin, il a participé à des tests de la fonctionnalité du programme et de plusieurs parties de l'architecture.

% de contribution estimé :	30%
nombre d'heures de travail estimé :	40h

Cléa : Cléa a concentré son travail sur la conception de l'UML et donc sur l'architecture du projet. Elle a également codée les actions que les joueurs peuvent effectuer. Nous avions initialement décidé de représenter les actions par des classes puis avons décidé que ce seraient de simples fonctions dans la classe Game (voir rapport 2 et 3). Elle a également réalisé la partie IA. Une partie de son travail est ainsi concentré sur la classe Game et TextualController. Elle a également rédigé 2 rapports dont le rapport final.

% de contribution estimé :	20%
nombre d'heures de travail estimé :	35h

Eugène : Eugene a participé à la réalisation des classes Controller, Player, DrawPile, Board. Il a également réalisé des vues et participé à la conception de l'interface utilisateur. Il a merge les codes de tous les membres afin d'obtenir un code fonctionnel et à permis de débugger une grande partie du code. Il a participé à la finalisation de l'implémentation du jeu de base et à la réalisation de l'extension , en particulier son implémentation technique. Enfin Eugene a rédigé 1 rapport.

% de contribution estimé :	30%
nombre d'heures de travail estimé :	40.1h

Conclusion :

En définitive, ce projet a été l'occasion pour nous d'appliquer toutes les connaissances accumulées au cours du semestre ainsi que d'approfondir certaines notions en effectuant nos propres recherches. La réalisation d'un projet dans son entièreté nous a forcés à être au maximum complets et exhaustifs.

Ce projet a également été l'organisation d'entraîner notre organisation dans le cadre d'un travail informatique : il a fallu répartir les tâches, que ce soit pour le développement du projet ou pour la rédaction des comptes rendus, ou encore mettre en place un système d'information pour la mise en commun de nos travaux personnels (GitHub). L'hétérogénéité des compétences au préalable du projet a pu entraîner certaines difficultés d'organisation, en effet le temps de réalisation des tâches pouvait fortement varier en fonction de la personne qui s'en occupait. Le travail en groupe et en commun a permis de pallier ce problème tout en favorisant une transmission de connaissances entre les membres du groupe.

Comme tout projet, nous avons dû effectuer des compromis et parfois nous limiter afin de privilégier la qualité globale du projet. Par exemple, nous avons parlé dans le rapport du cas de la difficulté des IA. Nous aurions également pu ajouter d'autres extensions car notre code a été pensé dans cette optique mais nous avons dû nous limiter afin de respecter les délais de l'UV. D'ailleurs, après avoir développé la première extension "Cités", nous avions commencé à en développer une seconde "Bastion" mais malheureusement nous n'avons pas pu l'achever.