Rapport 3

Machi Koro

CHAOUCHI Nasser & DUBUS Théo & GAJAN Antoine & SZENDROVICS Sacha & TAUPIN Nicolas

Responsable du livrable 3 : CHAOUCHI Nasser

Sommaire

Introduction

Dans le cadre de l'UV LO21, nous utilisons les connaissances apportées par les cours magistraux et les travaux dirigés afin de mener à bien un projet de groupe. Celui-ci répond au sujet disponible ici. Le but de ce dernier est de nous familiariser avec la programmation orientée objet en produisant une version numérique du jeu *Machi Koro*. Nous mettrons en application dans ce projet les différentes notions vues en cours de LO21.

Dans ce troisième rapport, l'implémentation de l'architecture, la modification de notre vision et les difficultés rencontrées lors de la phase de développement seront présentées.

Nous évoquerons également la répartition des tâches, les premières images du jeu en mode console ainsi que nos idées pour la phase applicative avec Qt.

1 Evolution de notre architecture

Lors de l'implémentation en code de notre jeu, nous avons constaté de nouvelles évolutions et changements par rapport à notre modèle UML théorique précédent. En effet, c'est en se confrontant à la "réalité" du code que nous nous sommes rendus compte de modifications nécessaires telles que des passages par référence, l'ajout subtil de paramètres ou encore la création de fonctions de service. Ainsi, certaines différences entre l'architecture proposée dans le deuxième rapport et celle actuelle ont pu apparaître et nous allons les résumer ci-dessous.

1.1 Joueur

Comme à peu près toutes les classes et méthodes de notre jeu, la classe Joueur n'a pas été épargnée par les modifications lors de l'implémentation en code de celle-ci.

1.1.1 Ajouts

Des contraintes que nous n'avions pas réalisées lors de la réalisation de l'UML ont notamment contribué à apporter de nouveaux attributs. C'est le cas du vecteur de bâtiment liste_bâtiment_fermes() qui recense les bâtiments fermés et ainsi permet de gérer cette mécanique de fermeture des bâtiments apportée par l'extension Green Valley. Nous avons aussi apporté en plus les attributs est_ia et stratégie permettant de faciliter la manipulation des IA dans notre jeu. En effet, dans l'UML, nous n'avions pas du tout pris en compte tout l'aspect IA du jeu et des joueurs. Nous avons donc décidé d'implémenter l'information qu'un joueur est une IA dans la classe joueur. Cependant, toute la mécanique de prise de décision de l'IA se trouve là où se font les décisions, c'est-à-dire dans les effets de cartes. Nous aborderons cet aspect ultérieurement. Pour ce qui est des méthodes, nous avons pu ajouter beaucoup de nouvelles méthodes. En plus des getters et setters, nous avons ajouté des fonctions d'affichage. Ces fonctions basiques parcourent seulement les différents vecteurs du joueur et les affichent, mais seront amenées à changer lors de l'implémentation en Qt. Nous avons aussi amené des fonctions plus "utilitaires", possede_bâtiment() ou selectionner_bâtiment() par exemple. Ces fonctions réalisent uniquement des actions de vérification d'appartenance ou de présence. Ces fonctions sont pratiques pour nous car nous effectuons souvent ces actions dans le code.

1.1.2 Modifications

Nous avons aussi été amenés à modifier la vision que nous avions des fonctions déjà présentes dans l'UML. Tout d'abord, nous avons décidé de modifier l'attribut liste_bâtiments du joueur. En effet, nous avons décidé de complexifier le type afin de pouvoir nous faciliter la tâche plus tard dans le jeu (notamment dans la recherche et l'affichage des cartes). Ainsi, nous avons décidé dans ce map de regrouper les bâtiments du joueur par couleur de bâtiment. En effet, malgré cette structure multi-dimensionnelle plus complexe, cette dernière nous permettra de faciliter la recherche et l'affichage par couleur. Nous n'aurons pas à itérer sur toute la liste de bâtiments, mais seulement faire une recherche par couleur. Nous avons aussi modifié certains prototypes. Nous avons modifié la fonction init_joueur() de l'UML qui selon n'avait pas de sens car a le même rôle qu'un constructeur. Nous l'avons donc remplacé par le

constructeur qui, après réflexions, prend beaucoup plus qu'un simple nom en paramètres. En effet, nous avons décidé de construire nos joueurs lors de la construction de Partie après la récupération des monuments et bâtiments contenus dans Edition de Jeu. Il faut pouvoir initialiser un joueur avec des bâtiments dits starters (boulangerie et champ de blé). La méthode la plus pratique que nous avons trouvé afin d'intégrer cette mécanique est d'initialiser la liste de bâtiments du joueur avec une liste de starters qui sera donné par l'instance de partie qui va créer le joueur. De même, la liste de monuments du joueur sera initialisée avec une liste de monuments fournie par partie à partir de ce qu'elle a pu récupérer de Edition de Jeu. On mettra l'état activé à faux pour tous les monuments le nécessitant lors de l'initialisation.

1.1.3 Suppressions

Nous avons aussi été amenés à repenser des fonctions. Par exemple, nous avons enlevé la fonction get_repartition_argent(). En effet, après réflexion collective, nous avons décidé que ce serait via la classe Partie que la répartition sera réalisée. Dans la même optique, nous avons déplacé la fonction acheter_carte() de Joueur vers Partie. Partie ayant un rôle central dans notre modélisation, elle s'occupera de l'achat d'une carte par un joueur au moment opportun.

1.2 Cartes

Les cartes, principale composante physique du jeu, ont, elles aussi, été sujet à évolution.

1.2.1 Ajouts

Tout d'abord, notre vision a été amenée évoluer suite aux cartes des nouvelles éditions : Green Valley et Marina. En effet, il est évoqué dans les effets de certaines cartes comme l'entreprise de rénovation un concept de bâtiment ouvert ou fermé. Plusieurs possibilités ont alors émergé pour représenter cette notion. La première consistait en l'ajout d'un attribut est_ouvert dans la classe Carte. Toutefois, après réflexion, cela n'est pas possible dans le sens où nous avons décidé de travailler avec une unique instance de chaque bâtiment. Cette décision avait été prise lors du précédent rapport dans le souhait d'améliorer la complexité spatiale de notre programme. C'est pourquoi nous avons préféré considérer un nouvel attribut propre aux joueurs : une liste de bâtiments fermés. Ainsi, lorsqu'un effet se déclenche avec la notion d'ouvert et fermé, il faudra uniquement ajouter ou supprimer le bâtiment de cette liste.

Ensuite, lors de la phase de développement du jeu en mode console, nous avons fait face à des questionnements concernant le lien entre éditions de jeu et cartes. En effet, lors de notre première esquisse de code, nous souhaitions créer les cartes depuis le constructeur d'édition de jeu. Le problème rencontré concernait alors les bâtiments de l'édition standard. En effet, certaines extensions ajoutent une carte physique pour un certain type de bâtiment comme la boulangerie. Or, avec notre modélisation précédente, nous allions créer un premier pointeur sur boulangerie avec l'édition standard, et une autre instance avec une extension ajoutant une carte boulangerie. Notre solution a alors consisté en la création d'une fonction clone() qui a pour but

de créer une nouvelle instance de la carte souhaitée. Ainsi, nous pouvons créer des copies des cartes, ce qui permettra de pallier le problème. Cette problématique sera abordée plus amplement dans le paragraphe relatif à l'édition de jeu.

1.2.2 Modifications

Nous avons été amenés à modifier notre vision des effets. Après de longues discussions, des interrogations avaient émergé en ce qui concerne les paramètres de la méthode. En effet, nous souhaitions dans le précédent rapport fournir uniquement le joueur actuel. Cependant, nous avons remarqué certaines limites à cette modélisation. En effet, le joueur actuel peut être récupéré depuis la classe Partie. Sa présence est donc inutile. En revanche, il nous semblait important de considérer le détenteur de la carte, afin que ce soit bien l'effet de la carte de la bonne personne qui se déclenche. De plus, certains monuments comme le centre commercial permettent un bonus dans l'effet des cartes rouges et vertes. Nous avons donc fait le choix d'ajouter un deuxième paramètre représentant le bonus. Ainsi, notre méthode virtuelle pure declencher_effet() prendra 2 paramètres : le détenteur de la carte et le potentiel bonus.

1.2.3 Suppressions

De nombreuses interrogations ont été soulevées quant aux paramètres des constructeurs. En effet, lors de notre précédent rapport, nous souhaitions implémenter une spécialisation de bâtiment (par exemple, une boulangerie) à l'aide d'un attribut, le path de l'image (qui sera important pour le rendu visuel avec Qt). Néanmoins, nous avons considéré que cet attribut n'était plus nécessaire. Le chemin d'accès de l'image est maintenant fourni directement dans le constructeur de la carte.

1.3 Partie

1.3.1 Ajouts

En ce qui concerne la classe Partie, nous avons fait différents ajouts, nous pouvons noter ainsi :

— static Partie* Singleton : L'ajout d'un singleton afin d'instancier la classe qu'une seule fois. De ce fait, l'unicité de la partie se fera et il n'y aura pas plusieurs parties en même temps. Le design pattern **Singleton** amène donc des méthodes supplémentaires telles que la méthode get_instance() et liberer_instance() qui instancie une seule instance de Partie, et qui permet de détruire cette même partie respectivement. Ces méthodes seront ainsi des méthodes **static** car elles seront globales dans tout le programme (l'instance est une variable globale qui sera liée à toutes les classes/méthodes).

— map<bâtiment*, unsigned int> list_bâtiments : Cet attribut nous sert à avoir la liste des cartes de type **bâtiment** ainsi que leur nombre afin d'avoir leur nombre tout au long de la partie.

— de_1 et de_2 : Ces attributs de la classe "théorique" Dé de notre UML se rerouvent dans Partie car nous n'avons pas besoin d'instancier de nouveaux objets, le mieux ici est d'avoir une fonction qui choisit aléatoirement le chiffre de chacun des dés. Pour cela, nous avons ajoutés des méthodes get_de_1(), get_de_2(), set_de_1(), set_de_2() qui permettront d'avoir accès à la valeur des deux dés respectifs et pouvoir modifier la valeur de ces attributs. Ces valeurs seront choisies aléatoirement grâce à la méthode lancer_de().

— unsigned int nb_monuments_win : Cet attribut nous sert à avoir le nombre de monument souhaité afin de gagner une partie, elle sera définie grâce à la classe **EditionDeJeu**.

— Shop* shop et Pioche* pioche : les attributs Shop et Pioche serviront à avoir accès à la pioche et au shop de la partie comme la partie a pour fonction de contrôleur.

— bool est_gagnant(unsigned int j) const : cette méthode a pour argument d'entrée un unsigned int qui fait référence au joueur_actuel du tour et vérifie si a la fin de son tour celui-ci a gagné ou non.

— acheter_carte(), acheter_monu(), acheter_bat() : Ces méthodes permettent au joueur_actuel d'acheter une carte, la fonction acheter_carte() fait appel à acheter_monu() ou acheter_bat() suivant le choix de l'utilisateur. Pour le cas des joueurs non humain, une partie leur est dédiée plus tard dans ce rapport.

— transfert_argent() : afin de gérer les nombreuses transactions d'argent entre les joueurs, nous avons codé cette méthode qui prend en argument le joueur a débité, le joueur a crédité ainsi que le montant, ceux sont tous des variables de type unsigned int. Elle va prendre tous les cas possibles pour le joueur à débiter. Par exemple, si la somme à créditer est supérieure à l'argent possédé par le joueur qui va être débité, alors nous faisons en sorte que le joueur donne toutes ses pièces et se retrouve à 0 pièce.

— ajout_bâtiment() : la fonction va permettre d'ajouter un bâtiment à la liste de bâtiment de la partie.

— selectionner_joueur() qui va prendre en compte le tableau de joueur et le joueur_actuel afin de retourner le joueur que l'on souhaite affecter suivant les effets de la carte.

— rejouer_tour() qui permet au joueur_actuel de rejouer un tour.

— vector<bâtiment *> get_starter() : Cette méthode va créer le starter de base de chaque joueur en lui donnant un champ de blé et une boulangerie.

1.3.2 Modifications

Nous avons pu modifier certains attributs et méthodes qui nous semblaient plus complexes à implémenter lors de la phase de programmation :

— joueur_actuel : nous avons passé l'attribut en unsigned int car nous préférons raisonner dans le projet avec des indices de joueur plutôt qu'avec des pointeurs sur Joueur*. Nous avons donc une méthode unsigned int qui renvoie le joueur_actuel, qui est get_joueur_actuel(). Toutefois, nous avons l'attribut tab_joueurs qui est un vecteur de Joueur*, couplé à l'indice des joueurs, nous pouvons avoir accès à chacun des joueurs d'une manière plus simple et efficace.

— Le constructeur de la classe : Dans notre ancienne version de l'UML, nous avions init_partie() qui agissait comme constructeur de la partie, toutefois nous avons décidé de lui donner comme argument d'entrée, une édition de jeu et des extensions afin d'initialiser la partie de la meilleure des manières.

— jouer_tour() : Cette méthode ne prend plus de Joueur* en argument d'entrée car nous avons le joueur_actuel à chaque tour et celui-ci sera incrémenté de 1 à chaque tour jusqu'à boucler au premier joueur de la liste.

1.4 Shop

1.4.1 Ajouts

Dans la classe **Shop**, nous avons réalisé peu de modifications. Nous pouvons noter l'ajout de la méthode acheter_bâtiment() (anciennement retirer() mais avec une fonction différente) qui permet de retirer une seule carte bâtiment du Shop, si celle-ci arrive à un nombre de 0 alors la carte ne sera plus présente dans le shop. Le bâtiment sera lui renvoyé avec cette méthode afin de l'ajouter au joueur. De plus, nous avons ajouté un attribut nb_tas_reel qui contrairement nb_tas_max (anciennement nb_cartes_possibles dans notre UML) indique le nombre de tas réel du shop. On pourra alors avoir un nombre de tas réel différent du nombre de tas max. Dans le constructeur du **Shop**, nous avons utilisé un shuffle qui mélangera les cartes.
En ce qui concerne, l'affichage du shop, nus avons conçu deux versions : un affichage simple avec les cartes, leur nombre et leur effet et une avec tout simplement leur nombre et leur nom.

1.5 Pioche

1.5.1 Ajouts

Pour ce qui est de la classe Pioche, nous avons ajouté une amitié avec la classe Partie, chose non précisée dans l'UML. Celle-ci permettra à la classe partie d'accéder aux éléments de la classe pioche, ce qui facilitera son instanciation. En effet, comme Partie est le contrôleur du jeu, c'est elle qui va créer le shop, la pioche, les joueurs...

1.5.2 Modifications

Nous avons modifié le constructeur de Pioche, suite à une modification de paramètres. Ce nouveau constructeur prend en paramètres un vector de pointeurs de bâtiments, chose que nous n'avions pas précédemment. Cela vient du fait que c'est la partie qui instancie la pioche. En d'autres termes, la Partie envoie dans la pioche une liste de bâtiments qu'elle a "importée" depuis une édition de jeu puis mélangée avant de l'injecter dans la pioche.
Nous avons aussi choisi de modifier le type de contenu d'un tableau vers un stack (une pile) car une pioche est par définition une pile. De ce fait, les fonctions de base associées à un stack, comme dépiler ou empiler, sont déjà implémentées et nous n'aurons pas à les coder.

1.6 Edition de Jeu

Notre vision de l'architecture de l'édition de jeu a aussi été amenée à évoluer.

Comme évoqué dans le paragraphe concernant les cartes, les éditions de jeu nous ont amené à réfléchir sur la manière dont elle devait exister. En effet, nous souhaitons depuis le premier rapport que Partie soit la classe qui gère le cycle de vie des cartes. Cependant, le problème de certaines extensions qui rajoutent une instance d'une carte déjà présente dans l'édition standard méritait réflexion. C'est pourquoi nous avons décidé de distinguer les éditions de jeu des extensions. A l'aide d'une méthode get_starter(), nous pouvons à présent avoir les instances des bâtiments nous posant soucis dans le passé, à savoir la boulangerie et le champ de blé sont les deux cas particuliers. Ainsi, nous pouvons gérer aisément cette difficulté.

Notre classe EditionDeJeu est donc définie par un nom, un nombre de joueurs minimal et maximal, une liste de monuments et de bâtiments. Enfin, pour différencier les éditions des extensions, nous avons ajouté un attribut est_edition. En effet, pour démarrer une partie, il faut choisir une unique édition de jeu et d'éventuelles extensions.

Nous avons implémenté des accesseurs pour chacun des attributs afin de permettre l'accès en lecture et en écriture selon les cas aux attributs privés.

1.7 Implémentation d'une IA

La réalisation d'une intelligence artificielle (IA) est à double enjeu. En effet, en plus de permettre à un joueur d'affronter l'ordinateur, la conception d'une IA nous permet avant tout de gagner un temps précieux dans le débogage du programme. Pour cela, il nous suffit de faire jouer IA contre IA un grand nombre de fois, et vérifier que tous les effets sont opérationnels.

Nous avons développé trois stratégies pour l'IA.

L'IA va tendre à d'abord acheter un monument avec une probabilité de 0.8 et 0.2 d'acheter un bâtiment. Si l'achat d'un des deux types ne peut se faire, alors l'IA va tenter d'acheter l'autre. Ainsi, si l'IA ne peut acheter aucun monument, elle va

tenter d'acheter un bâtiment. Nous avons fait ce choix, dans le but que l'IA puisse finir plus facilement une partie sans acheter toutes les cartes disponibles. Le choix des IA lors du choix de joueur, de bâtiment ou de monument se fait de manière aléatoire peu importe la stratégie de l'IA. Seule la phase d'achat de bâtiment va diverger selon la stratégie.

En effet, la couleur du bâtiment va influer lors de la phase d'achat. Nous avons développé trois stratégies pour l'IA :

- Agressive : l'IA voudra acheter en priorité des bâtiments de couleur rouge, si elle ne peut pas, elle achètera de manière aléatoire
- Défensive : l'IA voudra acheter en priorité des bâtiments de couleur bleue, si elle ne peut pas, elle achètera de manière aléatoire
- Neutre : l'IA aura un comportement totalement aléatoire

1.8 Classe exception

Afin de gérer les exceptions, nous avons choisi de gérer les exceptions à l'aide d'une classe. Pour cela, nous l'avons fait hériter de la classe de la STL nommée exception. Celle-ci nous permet d'obtenir plus d'informations et d'appliquer la méthodologie vue en cours magistral.

2 Notre jeu en images

FIGURE 1 – Accueil de notre jeu

L'accueil du jeu se fait par le choix des paramètres de la partie. C'est ici que le jeu va construire l'instance de Partie avec les paramètres rentrés.

FIGURE 2 – Exemple d'un tour pour un joueur

Chaque tour, nous affichons toutes les informations pratiques au tour : le joueur actuel et la valeur des dés qui viennent d'être lancé. Pour les informations sur le joueur actuel, un tableau coché ou non représente la liste de ses monuments (activés ou non). Les bâtiments sont aussi représentés sous forme d'une liste avec leur quantité à gauche.

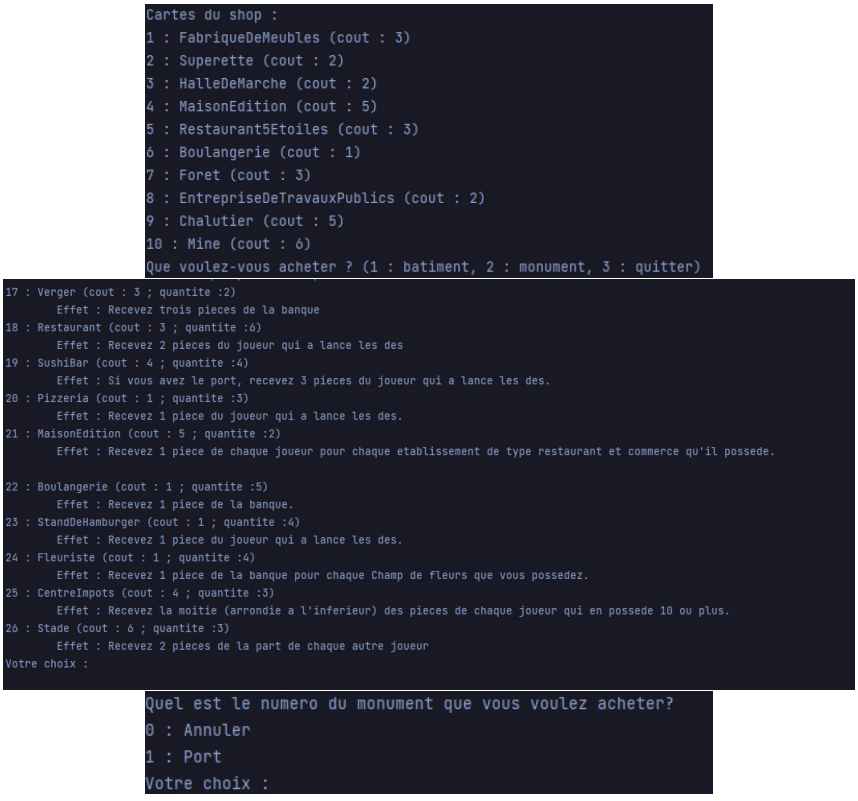

FIGURE 3 – Interactions joueur/jeu

Pour chaque tour, à chaque joueur, on présente le contenu du shop actuel et on propose au joueur de faire 3 actions : soit activer un de ses monuments, soit acheter un nouveau bâtiment, soit ne rien faire du tout. Lorsqu'on choisit d'acheter un nouveau bâtiment, pour chaque carte du shop, un descriptif de la carte est faite, il ne reste qu'à choisir celui qu'on veut. Lorsque l'on veut activer un monument, seuls les monuments que l'on peut se permettre (donc prix monument $<$ argent joueur actuel) apparaissent.

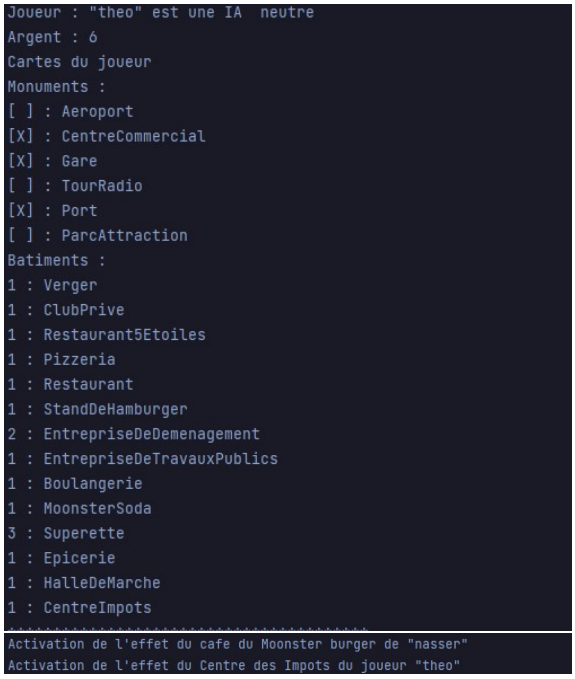

FIGURE 4 – Evolution du set d'un joueur au cours de la partie joueur/jeu

On peut remarquer que chaque fois qu'une carte prend effet (active son effet suite à un évènement dans la partie), un message est mis dans la console, afin de pouvoir suivre l'enchaînement des actions.

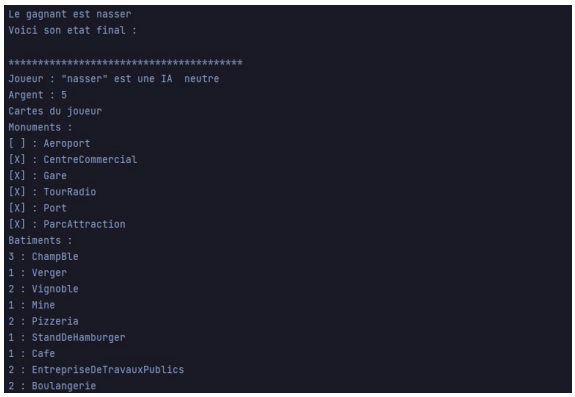

FIGURE 5 – Lorsqu'un joueur à gagné

Voici le lien vers une démonstration vidéo de notre jeu en mode console en IA vs IA avec 3 joueurs.

3 Interface graphique

Notre jeu étant opérationnel en mode console, nous devons à présent développer une architecture graphique. Pour cela, nous utiliserons le framework Qt.

3.1 Premières esquisses

Au cours de nos réunions, nous avons pu discuter de la future interface graphique. Afin de mieux comprendre le travail à réaliser et répartir le travail de manière optimale, nous avons émis des idées quant à notre vision de l'interface.

Tout d'abord, on demandera à l'utilisateur de choisir l'édition de jeu et les extensions comme suit :

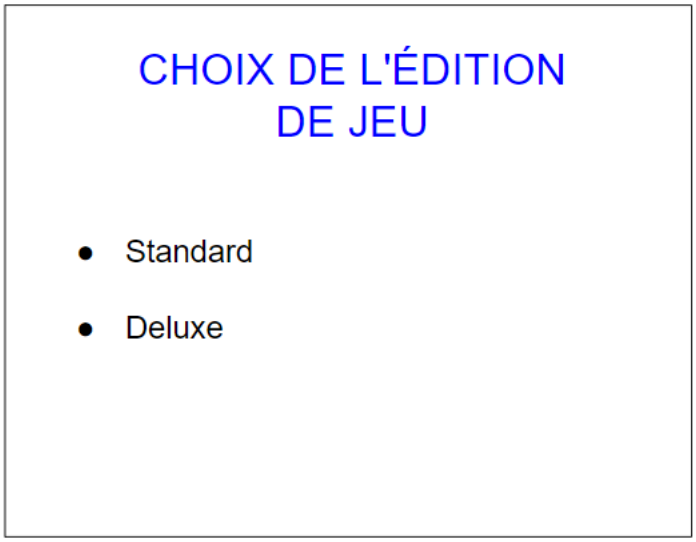

FIGURE 6 – Choix de l'édition de jeu

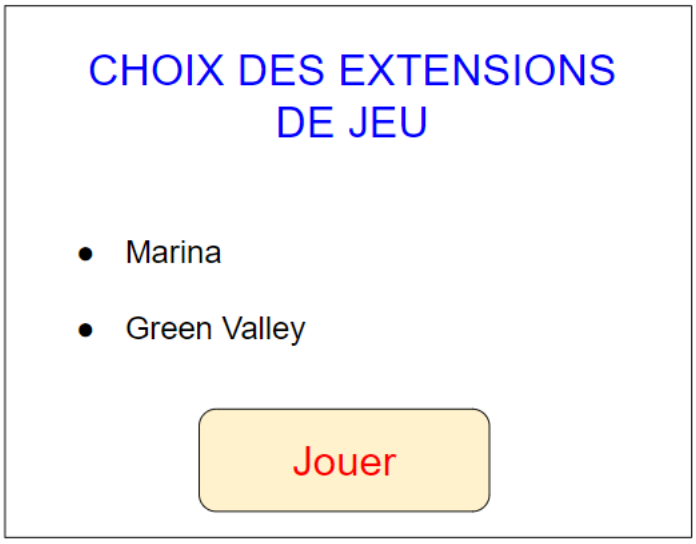

FIGURE 7 – Choix des extensions de jeu

Ensuite, on demandera à l'utilisateur de saisir les joueurs et s'ils sont des IA ou non. Nous avons alors la représentation suivante :

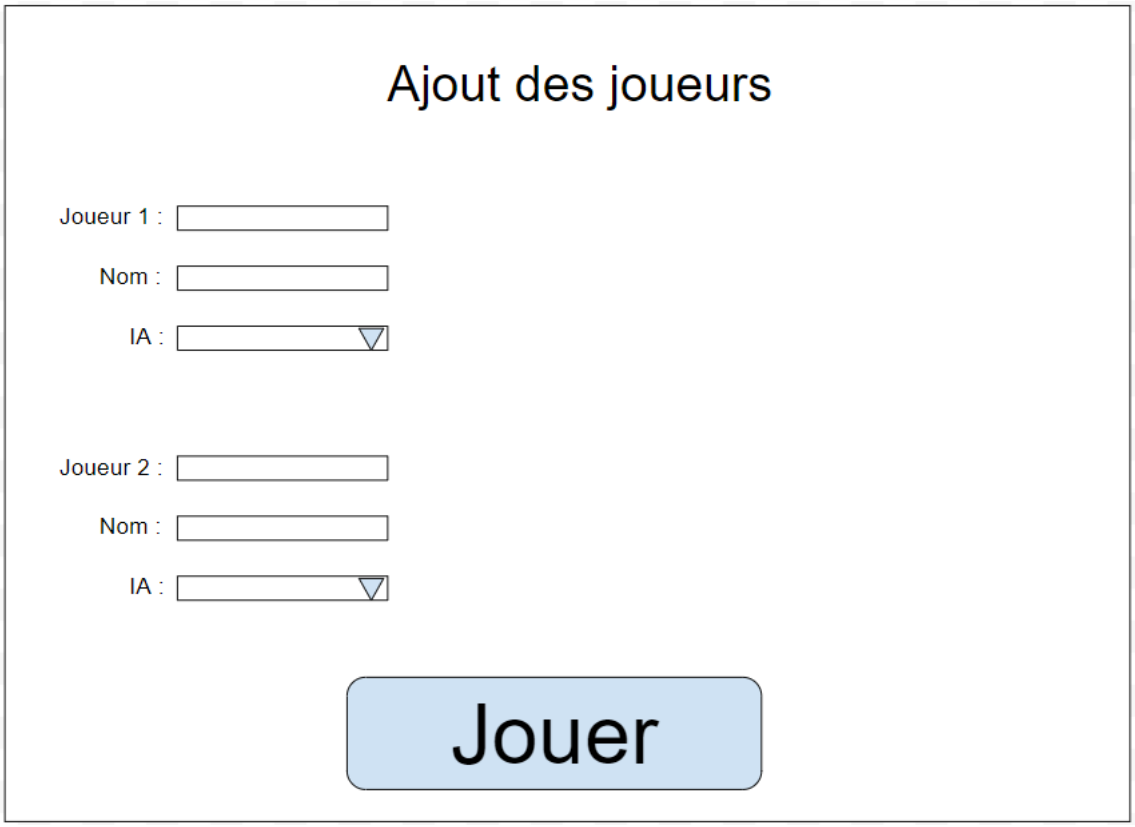

FIGURE 8 – Ajout des joueurs

Enfin, nous avons émis une ébauche de la modélisation d'un tour de jeu. Pour cela, en haut de la page, il sera indiqué le nom du joueur qui doit jouer, ainsi que la valeur des dés. Pour le plateau, située au-dessous, nous ferons apparaître la pioche à gauche avec les affichages pour comprendre le jeu (ex : le bâtiment a déclenché son effet, un joueur a reçu tel somme). Le shop sera représentée comme une succession de cartes différentes. Quant au bas de la page, il permettra de voir les cartes d'un joueur et de naviguer à l'aide des flèches sur les côtes. Entre autres, cela permettre de voir l'argent du joueur, son nom, ses bâtiments et monuments. Par défaut, on veillera à ce que le joueur affiché soit le joueur qui doit jouer.

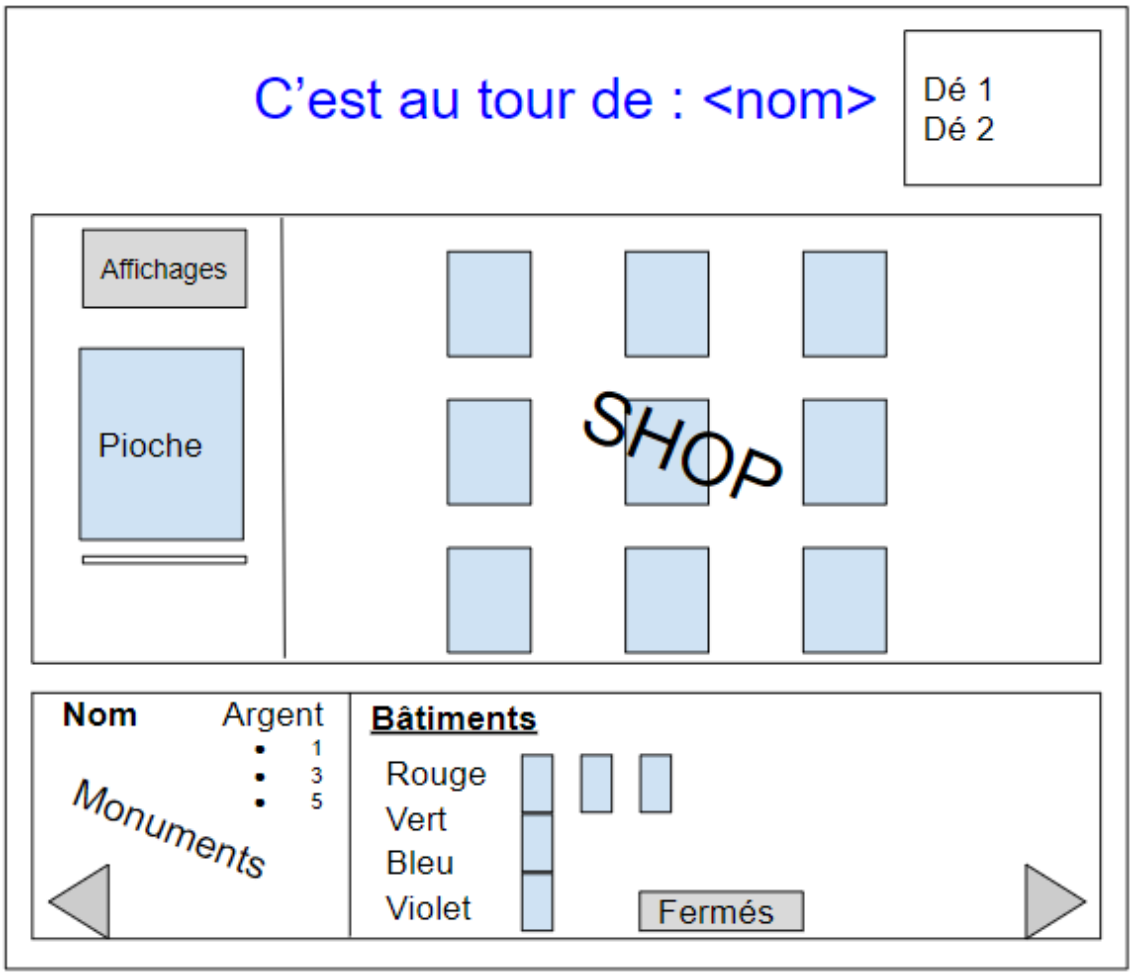

FIGURE 9 – Modélisation d'un tour de jeu

Lorsqu'il s'agira de demander à l'utilisateur de faire des choix, nous ferons apparaître des fenêtres pop-up.

Enfin, en fonction du temps disponible, nous adapterons l'interface graphique aux contraintes temporelles que nous avons.

Conclusion

Au cours de cette troisième partie du projet, nous avons construit l'architecture de notre jeu Machi Koro en mode console. Plus particulièrement, cette phase a nécessité une remise en question de notre vision de l'architecture, afin de l'optimiser.

Ce projet est pour nous l'occasion de mettre à profit notre esprit d'équipe pour appliquer les connaissances théoriques acquises lors des cours magistraux et travaux dirigés en les appliquant à un cas concret : design patterns (iterator, singleton), classes, méthodes, pointeurs et STL.
Nous devons à présent nous concentrer sur la réalisation d'une interface graphique, avec Qt.

Liste des tâches

Cette section est dédiée à l'avancée des tâches du projet. En particulier, on y trouvera les tâches à faire, celles en cours, et celles terminées. De semaine en semaine, nous reprendrons les différentes tâches et les feront changer de partie en fonction de leur état d'avancement.

Le format de chaque tâche est le suivant :

À faire :

- : Correction de l'architecture si besoin ; tous ; 2 heures
- : Créer l'interface graphique avec QtCreator et l'implémenter dans notre projet ; tous ; Durée Indéterminée

En cours :

- : Début de modélisation avec Qt ; tous ; 1 heure
- : Réflexion sur l'architecture graphique du jeu ; tous ; 3 heures

Fait :

- : Répartition des tâches ; tous ; 30 minutes
- : Ajout des IA ; Théo, Sacha ; 1 heure
- : Création des effets des cartes violettes ; Antoine ; 2 heures
- : Création des effets des cartes rouges ; Nasser ; 2 heures
- : Création des effets des cartes bleues ; Nicolas ; 2 heures
- : Création des effets des cartes vertes ; Théo ; 2 heures
- : Création des effets des cartes monuments ; Sacha ; 2 heures
- : Ajout des attributs et méthodes nécessaires au bon fonctionnement du jeu ; tous ; 3 heures
- : Code du contrôleur (Partie, Pioche, Shop, EditionDeJeu) ; tous ; 4 heures
- : Rédaction du rapport ; Nasser, Antoine, Nicolas ; 5 heures
- : Relecture du code : Théo, Sacha ; 12 heures

Table des figures