

Livrable final

Machi Koro



Sommaire

Introduction	2
1 Présentation du jeu	3
1.1 Menu	3
1.2 Partie principale	3
1.3 Fin de partie	4
2 Architecture du jeu dans sa version définitive	5
2.1 Préambule	5
2.2 La classe Partie	6
2.3 Les Vues	6
2.3.1 VueInfo	7
2.3.2 VuePioche	7
2.3.3 VueShop	8
2.3.4 VueCarte	8
2.3.5 VueJoueur	9
2.3.6 VuePartie	10
2.4 Les éléments centraux	11
2.4.1 Cartes	11
2.4.2 Shop	12
2.4.3 Joueur	12
2.4.4 Pioche	12
2.5 Elements supplémentaires	13
3 Programmation de la version graphique avec Qt	14
3.1 Menu pour instancier une partie	14
3.2 Vue de la partie	15
3.3 Vue de cartes	15
3.4 Vue du joueur	16
3.5 Vue de la pioche	17
3.6 Vue du Shop	17
3.7 Affichage de l'entête	18
3.8 Vue avec les informations	18
4 Évolutions et limites de notre projet	19
4.1 Ambitions graphiques	19
4.2 Autres ambitions	19
5 Liste des tâches réalisées par chacun	21
Conclusion	25

Introduction

Dans le cadre de l'UV LO21, nous avons travaillé ce semestre sur la conception du jeu Machi Koro, traduit par Minivilles en français. Machi Koro est un jeu de stratégie et de gestion de ressources dans lequel les joueurs incarnent des chefs d'entreprises qui essaient de développer leur ville. Le but du jeu est de devenir le premier à construire tous les bâtiments de sa liste de cartes. Pour cela, les joueurs doivent collecter des ressources en achetant et en développant de nouveaux bâtiments et en utilisant habilement leurs cartes de développement. Avec une mécanique de jeu simple et des éléments de chance aléatoires, Machi Koro est un jeu accessible pour les joueurs de tous âges et de tous niveaux de compétence.

Nous avons pu réaliser ce projet dans le but de nous familiariser avec la programmation orientée objet et les interfaces graphiques. Au cours de ce projet, nous avons mis en application les différentes notions que nous avons apprises en cours de LO21, telles que les classes, les attributs, le polymorphisme et l'interface graphique. Nous avons également eu l'opportunité de mettre en pratique nos compétences et de les améliorer au fil du temps. Ce projet nous a offert une solide base de connaissances théoriques qui nous sera utile dans notre parcours professionnel, que nous souhaitons travailler dans le développement de logiciels, la création de jeux vidéo ou tout autre domaine de l'informatique.

Dans ce dernier rapport, nous allons partager notre utilisation de l'UML pour concevoir et planifier le développement du jeu. Nous allons également détailler l'architecture de notre jeu, notre processus de programmation en utilisant le framework Qt, en expliquant les choix de conception et les défis que nous avons rencontrés en cours de route. Ce rapport vous donnera un aperçu de notre travail et de notre approche de la programmation orientée objet, et vous permettra de mieux comprendre comment nous avons réussi à créer un jeu fonctionnel.

En plus de présenter les réalisations et les enseignements de ce projet, ce rapport s'efforcera par ailleurs de mettre en lumière les difficultés et les limites rencontrées au cours de sa réalisation. Nous allons discuter des défis auxquels nous avons été confrontés et de la manière dont nous avons réussi à surmonter ces obstacles. Cela permet de mieux comprendre les processus de développement de logiciels et de voir comment nous avons géré les imprévus et les difficultés inévitables qui se sont présentés. En explorant ces problèmes de manière ouverte et honnête, nous espérons que ce rapport sera non seulement informatif, mais aussi inspirant et instructif.

1 Présentation du jeu

1.1 Menu

Le menu de notre jeu comprend deux fenêtres de formulaires : une pour le choix des extensions et éditions, et une autre pour le choix du nombre de joueurs, de leur comportement ainsi que les caractéristiques du shop.

Le premier formulaire nous permet de choisir parmi les éditions "Standard", "Deluxe" et "Custom" du jeu. "Standard" et "Deluxe" possèdent le même contenu que les éditions du même nom que nous pouvons trouver dans le commerce. L'édition "Custom" est une édition personnalisée que nous avons créée afin d'inclure des cartes spéciales qui ne sont disponibles dans aucune autre édition, comme "la fabrique du père Noël". Ces cartes ont été pour la plupart créées par les éditeurs du jeu à l'occasion d'événement spéciaux. Nous avons à cœur de rendre le jeu le plus complet possible, nous avons alors implémenté toutes les cartes dont nous avons réussi à trouver l'existence. L'édition "Custom" contient donc toutes les cartes du jeu, cette dernière nous sert à donner une utilité à chacune de nos cartes. Si nous choisissons dans le menu l'édition "Standard", nous avons la possibilité d'ajouter les extensions "Marina" et/ou "GreenValley".

Dans le deuxième formulaire, l'utilisateur est invité à sélectionner le nombre de joueurs et à entrer le nom et le type (Humain, IA agressive, IA défensive, IA aléatoire) de chaque joueur. Il est important de noter que les noms des joueurs ne peuvent pas être une chaîne vide et que deux joueurs ne peuvent pas avoir le même nom, sinon un message d'erreur sera affiché sur la fenêtre. L'utilisateur doit également sélectionner le type de shop qu'il souhaite utiliser : "standard" (avec une pioche) ou "extended" (avec toutes les cartes disponibles dès le début). Si l'utilisateur choisit le type "standard", il peut par ailleurs choisir le nombre de cartes disponibles dans le shop. Ce deuxième formulaire dépend en partie du premier, car chaque édition/extension autorise un nombre de joueurs maximum différent et la taille maximale du shop ne peut pas dépasser le nombre de cartes différentes disponibles, ce nombre variant également selon les éditions/extensions. Suite à la validation du premier formulaire, ces limites seront enregistrées et prises en compte dans le second formulaire.

1.2 Partie principale

Une fois le menu validé, la partie commence et une fenêtre de jeu s'ouvre, comprenant trois parties :

Entête

Il comprend un résumé du profil de la partie (édition, nombre de joueurs, nombre de monuments nécessaires pour gagner), le nom du joueur actuel, le numéro du tour actuel, un bandeau décoratif et le résultat du score des dés. Il y a aussi un bouton "ne rien faire" permettant au joueur de passer son tour sans effectuer d'actions.

Partie centrale

Il contient une représentation de la pioche avec son taux de remplissage en pourcentage, un shop avec les cartes affichées sous forme de grille et avec un petit numéro sur chaque carte indiquant le nombre de cartes empilées. Il y a par ailleurs une info box déroulante qui fournit des informations utiles à l'utilisateur, comme des notifications lors du déclenchement d'un effet ou des instructions sur les actions que peut effectuer le joueur actuel.

Partie basse

Il comprend monuments et bâtiments possédés par le joueur, le nom du joueur, son argent et un bouton pour afficher ses bâtiments fermés. Il y a également des boutons permettant de voir les vues des autres joueurs.

Popups interactives

Les popups interactives sont des fenêtres contextuelles qui s'ouvrent lorsque l'utilisateur doit faire un choix. Elles peuvent par exemple lui demander de sélectionner un joueur, un bâtiment ou un monument appartenant à un autre joueur ou parmi ses propres cartes. Ces popups sont généralement utilisées dans le cadre du déclenchement d'effet de certaines cartes nécessitant un choix du joueur en question. Il y a de plus des popups qui se déclenchent lors d'un clic sur une carte dans la vue d'un joueur ou sur le shop. Cette popup contient la carte agrandie afin de pouvoir l'examiner de plus près. Dans le cas d'un clic sur un des monuments du joueur actuel ou une des cartes du shop, un bouton "acheter" est ajouté sur la carte, permettant au joueur en question de procéder à l'achat de la carte. Si le joueur ne peut pas acheter la carte (monument déjà possédé ou ressources en argent insuffisantes), le bouton d'achat sera désactivé.

1.3 Fin de partie

À la fin de la partie, le jeu se ferme et une fenêtre indiquant le nom du gagnant apparaît.

2 Architecture du jeu dans sa version définitive

Dans cette partie, nous allons vous présenter l'architecture du logiciel final en détaillant la partie récemment modifiée graphique. Pour une vue d'ensemble de l'architecture complète, nous vous invitons à consulter le diagramme UML en [annexe](#).

2.1 Préambule

L'architecture de notre jeu se divise en 3 grands modules que l'on peut à chaque fois associer aux principaux acteurs d'une partie physique de Machi-Koro. Ces modules contiennent eux même des classes permettant de faire tourner une partie de Machi-Koro. Nous allons ici brièvement présenter les modules avant de rentrer dans le détail des classes incluses dans ces modules.

Tout d'abord, le module Carte va gérer l'ensemble des éléments relatifs aux cartes du jeu et la manipulation de celles-ci vis-à-vis des autres modules (ajout ou retrait de cartes d'un joueur, achat d'une carte dans une partie, etc...). Comme le veut la définition du jeu, les cartes sont toutes différentes. Elles peuvent être des bâtiments ou des monuments, elles ont un nom, des effets différents. Nous avons donc mis en place un jeu d'héritage au sein de cette classe afin d'implémenter ce phénomène. Ainsi, chaque carte du jeu possède sa propre classe. Cela permet de gérer toutes les spécificités liées aux cartes et leurs effets. Les effets seront d'ailleurs contenus dans ces classes-là. Ces classes spécifiques héritent d'une classe propre à leur nature (elles héritent de Bâtiment ou Monument selon qu'elles sont l'un ou l'autre). Cela permet notamment de gérer les subtilités et distinctions entre bâtiments et monuments. Enfin, toutes ces cartes héritent de la classe mère Carte qui définit de manière primaire ce qu'est un objet carte. Ce choix d'architecture pour représenter les cartes permet la modularité du jeu. En effet, si une nouvelle carte est ajoutée au jeu, il suffit de créer une nouvelle classe spécifique à la carte puis la faire hériter selon sa nature (bâtiment ou monument). Une classe d'affichage, VueCarte, permettant aux cartes de bien s'afficher de manière graphique, est associée à la classe Carte.

Viens ensuite le module Joueur. Ce module-ci s'occupe de tout ce qui tourne autour de la gestion des joueurs, qu'ils soient réels ou IA. Contrairement à la classe Carte, nous n'avons ici pas décidé de fractionner le module en beaucoup de classes. En effet, selon nous, on ne peut pas intégrer plus de types de joueur qu'humain et IA. Nous reviendrons sur ce point dans la partie dédiée à la classe joueur et en fin de rapport. Nous avons donc stocké tout le module dans la classe Joueur correspondante.

Enfin, le module Contrôleur. Ce dernier est très important dans l'architecture. C'est lui qui fait le principal des liens entre les éléments de l'architecture, qui séquence, qui crée et gère une partie de jeu de Machi-Koro. Les sous-modules de Contrôleur vont être expliqués en détail dans la suite de cette partie. Nous allons donc commencer avec la classe Partie de Contrôleur.

2.2 La classe Partie

La classe `Partie` revêt un rôle capital dans la gestion du jeu. En effet, lors de ce projet, nous avons décidé d'utiliser le design pattern Singleton afin de ne pouvoir instancier qu'une et une seule partie par ouverture du programme. Et c'est cette classe qui va venir se charger de cette tâche. Ainsi, la création d'une partie, qui implique la création de joueurs, d'un ensemble de cartes, et de tous les éléments qui gravitent autour d'une partie, devront être instanciés par `Partie`. Cette classe permet de gérer deux fonctionnalités capitales : pouvoir créer et pouvoir jouer une partie.

Tout d'abord, nous avons décidé d'intégrer le design pattern Singleton à `Partie`. Ainsi, une classe `Handler` est liée à `Partie` et, via une méthode d'instanciation, permet d'avoir une instance de `Partie` (nous n'avons jamais accès de manière publique au constructeur de `Partie`). Instancier une partie signifie créer une partie. Ainsi, lorsque l'on en crée une, on va créer de nouveaux joueurs, une nouvelle pioche, un nouveau shop, et des paramètres pour la partie. La pioche va être remplie à d'une liste de carte prédéfinie grâce au constructeur de l'Édition de jeu sélectionné dans le constructeur d'instances de `Partie`. Le fait que l'attribut du handler soit en static permet un accès unique et global de la partie au programme et donc ne permet de n'instancier qu'une seule fois un shop, une pioche, les cartes... Cela affecte non seulement la facilité d'accès à l'instance de la partie au sein du programme, mais aussi à terme les performances du jeu.

L'autre objectif central de la classe `Partie` est sa capacité à pouvoir faire fonctionner une partie de *Machi-Koro*. Conceptuellement, une partie est une suite de tours. C'est de cette manière que la classe gère le lancement de parties. Tant qu'il n'y a pas de gagnants, on joue des tours de jeu. Et de la même manière, on peut décomposer un tour en plusieurs phases de jeu. L'enjeu a donc été d'identifier puis implémenter ces différentes phases de jeu à la fonction permettant de jouer un tour. Comme l'instance de la partie est l'objet "omnipotent" du programme, il aura accès à tous les éléments de celle-ci et c'est lui qui ordonnera à tous les autres éléments du jeu ce qu'il faut faire. Par exemple, au cours d'un tour, lorsqu'il faudra piocher ou déclencher les effets d'une carte donnée, c'est la partie en cours (et plus précisément la fonction `jouer_tour`) qui donnera l'ordre à la pioche de dépiler, l'ordre à la carte de déclencher son effet.

Étant donné que la classe `Partie` est celle qui gère le déroulement d'une partie, elle concentre de ce fait beaucoup de fonctions dites "utilitaires" qui accomplissent des tâches redondantes lors du déroulement d'une partie. Typiquement, le fait de choisir un joueur ou une carte parmi une liste donnée, le fait d'ajouter de l'argent au pécule d'un joueur, etc...

2.3 Les Vues

La création de classes de vues a été une étape importante dans le développement de la version graphique du logiciel. En effet, cela a permis de séparer la logique du traitement des données de l'affichage de ces données à l'écran. Grâce à ces nouvelles classes, il est désormais possible de personnaliser l'apparence de chaque élément

affiché dans l'interface graphique, en utilisant par exemple des images ou des couleurs différentes. La classe "VueJoueur" par exemple, a été spécifiquement conçue pour représenter toutes les informations et les éléments graphiques associés à un joueur dans l'interface utilisateur. Grâce à cette classe, il est possible de gérer l'affichage de toutes les données liées à un joueur, telles que son nom, son score ou sa progression dans le jeu. En créant des classes de vues pour chaque type d'objet à afficher, il est possible de rendre modulaire le code et de rendre le développement plus flexible. Si l'on souhaite par exemple ajouter de nouvelles informations à afficher pour un joueur, il suffira de modifier la classe "VueJoueur" sans avoir à toucher au reste du code. Ces classes de vues permettent également de centraliser toutes les instructions d'affichage, ce qui rend le code plus lisible et plus facile à maintenir.

2.3.1 VueInfo

La classe `VueInfo` hérite de `QVBoxLayout` et a pour but de créer une info box scrollable permettant d'afficher des indications à l'utilisateur, par exemple indiquer que l'effet d'une carte a été activé ou encore donner des instructions au joueur pour qu'il sache quelle action il peut effectuer.

Elle possède deux méthodes publiques :

- `VueInfo` : constructeur de la classe
- `add_info` : ajoute une indication à la liste des indications affichées dans l'info box. Prend en paramètre une référence constante vers une chaîne de caractères contenant l'indication à ajouter.

La classe possède également plusieurs attributs privés :

- `info_permanent` : correspond à un label situé au-dessus de la zone scrollable
- `widget_layout_info` : correspond au widget contenant le vertical layout dans lequel seront les informations
- `scroll_info` : correspond à la zone scrollable dans laquelle est contenue `widget_layout_info`
- `info_layout` : correspond au layout vertical dans lequel sont les informations
- `liste_info` : correspond à un vector contenant toutes les informations de l'infobox (sous forme de `QLabel`)

À noter que lorsqu'on ajoute une nouvelle information dans l'info box, celle-ci apparaît en haut, les anciens messages étant décalés vers le bas. Ainsi, le message le plus en haut correspond toujours à l'information la plus récente.

La liste des informations contenue dans l'info box sont uniquement propres au tour en cours, cette dernière étant réinitialisée après chaque tour d'un joueur en même temps que le reste de la vue de la partie.

2.3.2 VuePioche

Cette classe possède deux attributs : `"pioche_exception"`, qui est un pointeur sur un objet `"QLabel"` permettant d'afficher une exception liée à la pioche lorsqu'il n'y a pas de carte dans la pioche, et `"barre_pioche"`, qui est un pointeur sur un

objet "QProgressBar" indiquant le niveau de remplissage de la pioche.

Elle comprend également un constructeur qui prend en paramètre la pioche du jeu et un pointeur sur un objet "QWidget" parent (optionnel).

La classe "VuePioche" a été ajoutée pour gérer l'affichage de la pioche dans la version graphique du logiciel, en utilisant des objets de la bibliothèque Qt pour l'interface graphique. Cette dernière est inspirée de l'exercice que l'on a pu faire en TD représentant le remplissage d'une pioche grâce à une barre de progression.

2.3.3 VueShop

La classe VueShop hérite de QGridLayout et a pour but de représenter graphiquement le "shop" de cartes dans une interface utilisateur.

Elle possède plusieurs attributs :

- `tab_vue_shop` : un vecteur de pointeurs vers des objets VueCarte correspondant aux vues de chaque carte dans le shop.
- `carte_choisie` : un pointeur vers un objet VueCarte correspondant à la vue de la carte sélectionnée par l'utilisateur (lorsqu'il clique sur une des cartes du shop).
- `largeur` : un entier non signé utilisé pour déterminer combien de cartes mettre sur chaque ligne du shop.
- `bouton_acheter` : un pointeur vers un objet QPushButton correspondant au bouton "acheter" sur la popup de la carte sélectionnée.

La classe possède également plusieurs méthodes publiques et slots :

- `VueShop` : constructeur de la classe.
- `void clicked_acheter_event` : slot déclenché lorsque l'utilisateur clique sur le bouton "acheter" sur la popup de la carte sélectionnée. Il déclenche l'achat de la carte.
- `void batiment_clique` : slot déclenché lorsque l'utilisateur clique sur une vue de carte dans le shop. Il ouvre une popup de la carte cliquée.

2.3.4 VueCarte

La classe VueCarte permet de gérer l'affichage dans plusieurs autres vues telles que la VueShop et la VueJoueur. Elle hérite d'un QPushButton car l'image dans le jeu est cliquable.

La classe VueCarte possède plusieurs attributs privés :

- `carte` : un attribut `const Carte*` permettant de retrouver les informations (path) de la carte pour pouvoir l'afficher par la suite
- `pixmap` : un attribut `QPixmap` qui sert à avoir l'image que l'on va affilier QIcon pour l'afficher
- `ButtonIcon` : c'est le QIcon permettant d'afficher la carte en settant l'Icon sur le QPushButton
- `path_carte` : le path de la carte est important, car celui-ci est le chemin de la console jusqu'à l'image correspondante pour la mettre dans la VueCarte

- `est_actif` : c'est un booléen qui sera mit à true dans le constructeur quand il s'agira d'un monument actif, les monuments possèdent deux paths (inactif, actif) donc cela permettra d'afficher la bonne image du monument au cours du jeu.

Elle possède aussi des méthodes publiques ainsi qu'un slot et un signal :

- `VueCarte` : constructeur de la classe prenant en compte en paramètre, la carte voulue, son état (qui change sa taille), si la carte est active ou non (qui est initialisée par défaut à false) et son `QWidget*` parent.
- `VueCarte` : à recheck
- `get_pixmap` et `set_pixmap` : getter et setter du pixmap
- `set_icon` : setter de `QIcon` de la Vue
- `get_est_actif` : getter de l'information active ou non de la carte

2.3.5 VueJoueur

La classe `VueJoueur` est la vue permettant d'afficher les informations principales du joueur, comme son nom, son argent, ses bâtiments, ses monuments (actifs ou non) et si l'extension le permet, ses bâtiments fermés.

La `VueJoueur` possède beaucoup d'attributs afin de la gérer au mieux durant la partie :

- `est_joueur_actuel` : c'est un booléen permettant de savoir si la vue du joueur est celle du joueur actuel (permettant l'achat de monument)
- `text_bat` : `QLabel*` qui sert à afficher le texte "Bâtiments"
- `nom_joueur` : `QLabel*` affichant le nom du joueur en vert si c'est le joueur actuel, sinon en rouge
- `argent` : `QLCDNumber*` permettant d'afficher l'argent du joueur
- `layout_informations` : `QHBoxLayout*` permettant de lier la `layout_informations_gauche` et `layout_droit`
- `layout_informations_gauche` et `layout_droit` : `QVBoxLayout*` gérant les layouts gauche et droite respectivement de la vue
- `layout_haut_gauche` et `layout_haut_droit` : `QHBoxLayout*` qui gère les éléments en haut de la `VueJoueur` donc le nom, l'argent du joueur d'un côté et, le `text_bat` et le bouton des bâtiments fermés de l'autre.
- `layout_batiments` et `layout_monuments` : `QGridLayout*` permettant respectivement de contenir les bâtiments du joueur et les monuments de la partie (actif ou non suivant le joueur)
- `vue_batiments`, `vue_batiments_ferme` et `vue_monuments` : `vector*` de `VueCarte` permettant de les stocker et de les ajouter dans les `QGridLayout`.
- `joueur` : cet attribut est prévu afin de garder le joueur en mémoire afin d'avoir son attribut plus facilement grâce à ses getters.
- `bat_ferme` : Ce bouton est actif seulement quand les extensions permettent des bâtiments fermés, il permet d'ouvrir une fenêtre de bâtiments fermés.
- `bouton_achat` : `QPushButton` qui sera affiché sur le visuel des cartes monuments quand ils sont désactivés et sur le joueur actuel, il sera cliquable si le joueur actuel a assez d'argent.
- `carte_choisie` : Cet attribut sert à exécuter l'achat d'une carte pour le faire parvenir à la fonction `acheter_mon(VueCarte *vc)` de la classe `Partie`

- `scroll_bat` et `scroll_mon` : correspondent aux zones scrollables des bâtiments et des monuments dans la vue.
- `parent` : correspond au parent de la `VueJoueur`
- `fenetre_bat_fermes` : `QWidget*` qui correspond à la fenêtre des bâtiments fermés qui s'ouvrira lorsqu'on appuiera sur `bat_ferme`
- `widget_scroll_bat` et `widget_scroll_mon` : qui correspondent au `QWidget*` qui contiendront les `QScrollArea` `scroll_bat` et `scroll_mon`

La classe `VueJoueur` possède également quelques méthodes et slots publics.

- `VueJoueur` : Constructeur de la classe `VueJoueur` qui prend en argument d'entrée, le joueur que l'on veut afficher, un booléen `e_j_a` qui est `true` si le joueur est le joueur actuel et son parent.
- `get_est_joueur_actuel` : qui retourne `est_joueur_actuel`
- `get_carte_choisie` : qui retourne la `VueCarte` choisie par le joueur
- `batimentClique` : slot qui permet de gérer l'affichage du bâtiment cliqué dans une nouvelle fenêtre
- `affichage_bat_ferme` : slot qui ouvre une nouvelle fenêtre avec les bâtiments fermés
- `clicked_acheter_event` : slot qui permet de lancer l'achat du monument par le joueur actuel
- `monumentClique` : slot qui permet de gérer l'affichage du monument cliqué dans une nouvelle fenêtre

2.3.6 VuePartie

La classe `VuePartie` est l'affichage général de la partie à travers les vue grâce à l'implication de la classe `Partie`.

Elle possède beaucoup d'attributs comme la classe `VueJoueur` :

- `nb_joueurs` : nombre de joueurs de la partie
- `joueur_affiche` : indice du joueur à afficher dans la `VueJoueur`
- `tab_vue_shop`
- `label_edj` : `QLabel*` servant à l'affichage de l'édition de jeu (et extensions) de la partie
- `label_joueur_actuel` : `QLabel*` servant à l'affichage du nom du joueur actuel
- `lcd_de1` et `lcd_de2` : `QLCDNumber*` qui permet l'affichage des résultats du dé ou des dés.
- `image_entete` : qui sert à afficher l'image d'entête du jeu
- `affichage_de_1` et `affichage_de_2` : sert à afficher un texte "Dé 1 :" et "Dé 2 :" respectivement
- `layout_de_1` et `layout_de_2` : `Layout` pour l'affichage du LCD du dé et son texte
- `display_des` : `layout` permettant de mettre verticalement les deux `layout` de dés.
- `structure` : `QVBoxLayout*` qui définira la structure général de la page du jeu
- `layout` : `QHBoxLayout*` qui affiche les boutons de afin d'accéder au joueur précédent et suivant, également sur ce `layout` il y a la vue `Joueur`
- `entete_gauche` : `QVBoxLayout` qui contient les infos (nombre de tours, édition de jeu, joueur actuel)

- entete : QHBoxLayout* qui contient l'entete_gauche, l'image d'entête ainsi que les dés.
- body : layout ayant la VuePioche(et sa progress bar), la VueShop et la VueInfo
- vue_joueur : VueJoueur affiché à l'instant t
- parent_fenetre : Widget Parent de la fenêtre
- fenetre_carte : Widget des fenêtres popup lors d'un clic sur une carte
- pioche_exception : QLabel* qui affiche quand il n'y a pas de cartes dans la pioche
- bouton_rien_faire : QPushButton* qui permettra de passer son tour sans jouer
- pioche : layout avec VuePioche et sa progress bar
- fenetre_pioche : Widget qui permettra d'ajouter pioche au layout body
- view_pioche : VuePioche* de la partie
- view_shop : VueShop* de la partie
- scroll_shop : QScrollArea* qui rend scrollable view_shop
- widget_shop : qui permet d'ajouter le scroll_shop (ainsi que view_shop) à body
- infos : VueInfo* de la partie
- widget_infos : qui permet d'ajouter infos à body

La classe possède également, comme les autres vues, des méthodes et des slots :

- VuePartie : constructeur de la VuePartie, il prendra ses paramètres avec son parent et l'instance de la partie qui est statique
- update_vue_joueur, update_vue_partie, update_vue_shop, update_vue_pioche, update_vue_info, update_des, update_nom_joueur : méthodes permettant de l'update des vues à chaque fois qu'elles sont appelées
- get_vue_joueur : qui renvoie vue_joueur
- get_vue_infos : qui renvoie infos
- get_vue_carte : qui renvoie fenetre_carte
- set_vue_carte : qui set fenetre_carte
- d_click et g_click : slot permettant de changer de vue de joueur, le bouton de gauche sera connecté au slot g_click et le bouton de droite au slot d_click
- ne_rien_faire_bouton : slot qui ouvre une fenêtre pour valider son choix de passer son tour ou non

2.4 Les éléments centraux

2.4.1 Cartes

Les cartes représentent l'élément central et majeur de notre jeu. En effet, Machi Koro est un jeu de cartes, et sans ces dernières, le jeu ne pourrait exister. Dans notre architecture, les cartes sont séparées en deux sous-catégories via un héritage public. Il existe les bâtiments et les monuments. Les bâtiments forment une partie centrale de notre architecture. Elles sont en effet reliées aux autres éléments centraux : la pioche, au shop, les joueurs. En effet, le joueur possède des bâtiments qui lui permettront d'évoluer au cours de la partie. Les bâtiments que l'on ne retrouve pas dans le jeu des joueurs sont stockés dans le shop ou dans la pioche. Afin de rendre notre architecture amovible, les différents bâtiments et monuments qui existent sont modélisées par des classes héritant respectivement de bâtiments ou monuments. Quant

aux effets, nous avons privilégié leur représentation avec une méthode virtuelle pure dans la classe Carte. En effet, pour respecter le principe de substitution et pouvoir déclencher l'effet de la carte depuis un pointeur sur Carte ou Batiment, la mise en place d'une méthode virtuelle pure était indispensable.

Afin d'optimiser les ressources mémoires, nous avons choisi de considérer les cartes comme des pointeurs sur les instances cartes. En effet, cela nous permet de ne créer qu'une instance de la carte pour l'ensemble des joueurs et la partie, tout en permettant aux joueurs d'avoir des interactions différentes et indépendantes avec chacune d'entre elles. Manipuler des pointeurs nous a permis d'éviter de créer autant d'instances de cartes qu'il n'y a dans le jeu de cartes physique.

2.4.2 Shop

Nous avons choisi d'implémenter une classe spécifique au shop permettant de faciliter l'affichage, la manipulation et l'achat des cartes bâtiment. Elle contient deux entiers non signés permettant d'obtenir le nombre de tas actuel et le nombre de tas maximum. L'attribut contenu permet quant à lui de stocker à la fois le pointeur de bâtiment et sa quantité, d'où la nécessité d'utiliser le type map.

Nous avons également la méthode `completer_shop` qui permet d'ajouter une carte dans le shop, cette méthode est appelée dans la partie. Le bâtiment passé en paramètre est celui récupéré depuis la pioche. La méthode `acheter_bat` permet quant à elle de récupérer un bâtiment du shop, de décrémenter sa quantité ou de décrémenter le nombre de tas réel. Cela permet ensuite de compléter si nécessaire le shop.

2.4.3 Joueur

Les joueurs représentent également un élément central de notre jeu. Les joueurs sont en relation avec la Partie, qui aura informatiquement créé les instances de joueurs. Ils sont aussi en relation avec les Bâtiments et les Monuments puisque chaque joueur possède ses propres cartes.

Pour gérer les différences entre joueurs humains et intelligences artificielles, nous avons choisi de privilégier un héritage par classe mère. Ce terme doit être compris dans le sens où joueur humain et intelligence artificielle sont regroupées au sein de la classe Joueur, sous la forme d'un attribut `est_IA`. Nous avons conçu 3 types d'intelligence artificielle : une agressive (achète en priorité des cartes rouges), une défensive (achète en priorité des cartes bleues) et une aléatoire (achète des bâtiments de différentes couleurs de manière aléatoire). Pour permettre l'achat de cartes, nous avons défini au sein de la classe Joueur une méthode d'achat.

2.4.4 Pioche

Dans notre architecture, nous avons décidé de considérer notre pioche comme une pile. Ainsi, le contenu de la pioche peut être interprété comme un tableau de pointeurs sur Bâtiment. Ainsi, pour obtenir la carte du dessus de la pile, il faudra

simplement dépiler la pile. En parallèle, il est indispensable de pouvoir alerter l'utilisateur quand la pioche est vide, d'où la mise en place d'une méthode dédiée à cet effet. Ainsi, la pioche interagit avec le Shop puisque c'est la pioche qui remplit le Shop.

Pour faciliter le mélange de la Pioche, nous avons choisi de mélanger préalablement les bâtiments, que nous transmettons en paramètre via un tableau au constructeur de Pioche.

2.5 Elements supplémentaires

Dans notre architecture, nous avons intégré deux classes qui, bien qu'elles ne soient pas au cœur du projet, présentent néanmoins une certaine importance.

GameException

La classe "gameException" hérite de la classe "exception" de la bibliothèque standard de C++ et sert à gérer les erreurs survenant dans le projet de manière structurée et efficace. Elle possède une méthode publique "what()" qui retourne un message d'erreur sous forme de chaîne de caractères constante et un constructeur qui prend en paramètre une chaîne de caractères et qui l'utilise pour initialiser l'attribut "info" de la classe, qui stockera le message d'erreur. En utilisant cette classe, nous pouvons facilement gérer les erreurs dans notre projet et fournir des informations significatives aux utilisateurs en cas de problèmes.

EditionDeJeu

La seconde classe est "EditionDeJeu", qui aurait également pu être gérée au sein de la classe "partie", mais qui a été séparée afin d'améliorer la modularité du projet. Elle s'occupe de la gestion de l'édition du jeu, ce qui permet de séparer cette partie du reste du code et de la rendre plus facilement modifiable en cas de besoin. En séparant cette classe de "partie", nous avons créé une structure de code plus claire et amélioré sa lisibilité.

La classe "EditionDeJeu" est utilisée pour gérer les différentes éditions (standard, custom, deluxe) et extensions (marina et greenvalley) du jeu. Elle possède plusieurs attributs protégés pour stocker les informations sur l'édition en cours, comme le nom de l'édition, le nombre maximum de joueurs et le nombre de monuments nécessaires pour gagner. Elle possède également un vecteur de pointeurs sur la classe "Monument" et un map associant des pointeurs sur la classe "Batiment" à des entiers, permettant de connaître la quantité de chaque carte. Elle a un booléen pour indiquer si l'édition est une edition (true) ou une extension (false). La classe possède un constructeur et un destructeur pour initialiser et nettoyer ces attributs, ainsi que plusieurs méthodes "getters" pour accéder à ces informations. Elle permet également de récupérer les monuments et les bâtiments de l'édition en cours.

3 Programmation de la version graphique avec Qt

Le passage du mode console au mode graphique a été difficile à appréhender pour notre groupe. En effet, notre jeu en mode console est ce qu'on pourrait qualifier de "séquentiel". Le jeu est une suite de tours, d'achat, de déclenchement d'effets... Le mode console est basé sur l'attente de l'input d'un joueur, via l'instruction `cin`. D'un autre côté, Qt est basé sur la programmation événementielle. Tout est régi par des signaux et des réponses à ceux-ci, ce qui n'est pas adapté pour faire un jeu de manière séquentielle. Il a donc été particulièrement difficile pour nous de traduire le premier mode de fonctionnement vers le deuxième. La suite de cette partie et du rapport est là pour expliciter et préciser les implémentations que nous avons dû faire afin de parvenir à cette tâche. Nous nous sommes inspirés de l'exemple réalisé en TD.

3.1 Menu pour instancier une partie

La création du menu de l'application graphique a été l'une des premières étapes de développement de notre projet. Nous avons décidé de mettre en place différentes options afin que les utilisateurs puissent personnaliser leur expérience de jeu.

Tout d'abord, nous avons ajouté la possibilité de choisir entre différentes éditions et/ou extensions du jeu. Cela permet aux utilisateurs de personnaliser leur partie afin de pouvoir jouer avec la version qu'ils préfèrent.

Une fois la fenêtre permettant ce choix validé, une seconde fenêtre apparaît : nous incluons le choix du nombre de joueurs et du type de chaque joueur (humain ou plusieurs types d'IA). Cela permet aux utilisateurs de jouer à plus de joueurs qu'ils ne sont réellement et également de définir le comportement de ces joueurs additionnels. Dans cette même fenêtre, nous avons ajouté le choix du type de shop (standard ou extended), et dans le cas "standard", le choix de la taille du shop.

Le menu se fait en deux fenêtres, car certains choix vont dépendre des extensions/éditions (nombre de joueurs max, taille du shop max, etc). Ainsi, on a d'abord besoin de récupérer ses informations pour ensuite construire de façon dynamique le second formulaire.

À la suite du menu, toutes les informations nécessaires à la création d'une partie sont récupérés, mais il reste encore à créer l'instance de `Partie`. Ainsi, le constructeur de `Partie` et la méthode `get_instance()` ont été re-codés pour pouvoir être appelés par le menu une fois que ce dernier avait récupéré toutes les informations nécessaires.

Il nous a ensuite fallu lier ce menu avec la fenêtre principale de la partie. En effet, nous avons à ce moment une fenêtre pour instancier la partie et une fenêtre avec le plateau de jeu, sans pour autant parvenir à les lier. Afin de pallier cette difficulté, nous avons décidé de créer le plateau de jeu depuis la fonction `jouer_partie()`.

3.2 Vue de la partie

La vue de la partie est formée de plusieurs sous vues, qui sont des instances de différentes classes. Ainsi, la classe `VuePartie` contient des attributs `VueShop`, `VueJoueur`, `VueInformation`, `VuePioche`. Il y a également un layout pour l'entête.

Afin d'améliorer l'expérience utilisateur et d'éviter que de multiples fenêtres pop-up se créent si l'utilisateur clique sur plusieurs cartes, nous avons décidé de créer une mécanique qui permet de n'avoir qu'une seule fenêtre pop-up au maximum.

3.3 Vue de cartes

Nous avons considéré les cartes comme une instance d'une classe héritant de `QPushButton`. En particulier, nous avons dû jouer sur la dimension des cartes afin d'avoir un affichage propre, qui convient non seulement aux cartes du joueur, mais également aux cartes du shop. Nous avons par ailleurs redéfini le slot pour gérer le comportement des vues de cartes lors d'un clic. En effet, dans notre conception, nous avons choisi que le clic d'une carte puisse permettre d'ouvrir l'image de la carte en agrandi dans une fenêtre pop up afin de pouvoir lire la description de l'effet de la carte et les autres informations. En outre, nous avons eu la nécessité de trouver un moyen d'afficher une image. Dans notre précédente version, les images étaient au format jpg. Néanmoins, après recherche sur internet, le format le plus facile à manipuler avec Qt est le format png. A l'aide des `Pixmap`, nous avons trouvé le moyen de manipuler facilement les images.

De plus, nous nous sommes aperçus que les chemins d'accès aux images étaient incorrects. Nous les avons donc modifiés afin de pouvoir afficher les images des cartes. En parallèle, pour gérer les images des monuments (actifs ou non), nous avons implémenté dans le constructeur de la vue de carte une succession de conditions pour établir s'il s'agit d'un bâtiment ou d'un monument, et s'il est actif ou non.

Ainsi, nous avons le rendu suivant dans la fenêtre principale :

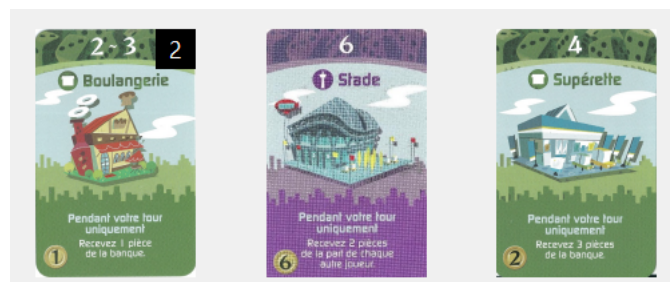


FIGURE 1 – Affichage des cartes dans la VueJoueur

Lorsque la carte est ouverte dans une fenêtre pop up, on a l'affichage suivant :

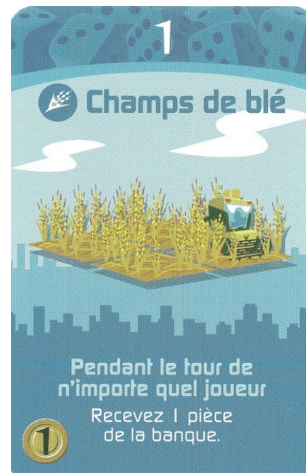


FIGURE 2 – Affichage de la carte dans une fenêtre pop-up

Enfin, en interne dans notre jeu, nous devons trouver le moyen de connaître la carte manipulée en permanence. Ainsi, pour faciliter la gestion, nous avons créé des accesseurs permettant de récupérer la carte en cours de manipulation. Ces dernières nous permettront de simplifier le code des fonctions de sélection et d'achat de cartes.

3.4 Vue du joueur

Suite à nos ébauches de rendu visuel dans le précédent rapport, nous avons le besoin d'afficher une vue de joueur en bas de la fenêtre principale. En manipulant les layouts comme vus en TD, ainsi que les différents types de widget, nous sommes parvenus à obtenir l'affichage suivant :

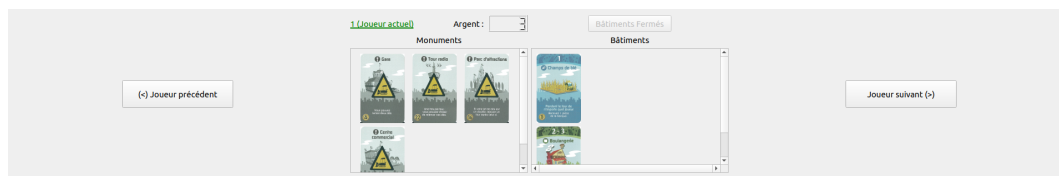


FIGURE 3 – Vue du Joueur

Nous avons appelé le constructeur de VueCarte dans le but d'afficher les cartes comme nous le souhaitions. Afin d'améliorer l'expérience utilisateur, nous avons manipulé des QScrollBar pour que les cartes prennent qu'un certain pourcentage de place sur l'écran.

Le rendu visuel du joueur étant terminé, il nous reste à gérer les différentes possibilités. En effet, lorsque le joueur clique sur un monument, nous souhaitons que la carte puisse s'afficher de manière agrandie et qu'il puisse l'acheter s'il ne l'a pas déjà fait. Nous avons donc défini un slot monumentClique() permettant d'afficher la carte en agrandi avec un bouton d'achat si toutes les conditions pour qu'il l'achète sont réunies (argent suffisant, monument non acheté, etc ...). Également, nous avons conçu un deuxième slot pour gérer l'achat lors du clic sur le bouton acheter.

Ensuite, nous avons manipulé les différents types de widgets pour afficher les informations sur le joueur. Enfin, pour être en accord avec les différentes extensions

du jeu qui font appel à la mécanique des bâtiments ouverts et fermés, nous avons ajouté un bouton pour accéder à ces derniers, dans une fenêtre pop-up.

3.5 Vue de la pioche

Pour gérer l’affichage de la pioche dans notre jeu, nous avons décidé de créer une classe spécifique appelée `VuePioche`. Cette classe est responsable de tout ce qui concerne l’affichage de la pioche et permet de rendre le code plus clair en fractionnant `VuePartie` en plusieurs classes d’affichage. En utilisant cette approche, nous avons pu alléger la classe `VuePartie` et simplifier le processus de modification de l’affichage de la pioche si nécessaire. La classe `VuePioche` est basée sur un layout vertical et inclut une barre de progression qui indique le nombre de cartes restantes dans la pioche. La classe `VuePartie` joue donc un rôle central dans l’affichage de la partie en cours et gère l’actualisation de l’affichage de la pioche via `VuePioche`.

La valeur de la barre de progression sera mise à jour chaque fois qu’une carte est piochée en utilisant la classe `VuePartie`. Pour ce faire, la classe `VuePartie` appellera une de ses fonctions qui met à jour la `VuePioche` en appelant le constructeur de `VuePioche` avec la pioche en tant que paramètre. La pioche contient des informations sur sa taille initiale et sa taille actuelle

3.6 Vue du Shop

Pour gérer l’affichage du shop dans la vue de la partie, nous avons également décidé de créer une classe dédiée appelée `VueShop`. Cette classe hérite du layout `QGridLayout`, car nous souhaitons afficher le shop sous forme de grille de cartes, comme nous avons pu le faire en TD. L’un des défis de l’affichage graphique du shop est que le nombre de cartes peut varier en fonction des éditions, ce qui implique par ailleurs des changements de format de la grille. Pour gérer cela, nous avons ajouté un attribut `largeur` à la classe `VueShop` qui permet de déterminer la largeur de la grille en fonction du nombre total de cartes à afficher dans le shop. Pour offrir un visuel de cartes plutôt qu’une simple chaîne de caractères, `VueShop` est une matrice de `VueCarte`. Ainsi, lors de la construction graphique du shop, le contenu du shop est copié dans un vector de `VueCarte` dans `VueShop`, puis ces cartes sont ajoutées à leur bon emplacement dans la grille de `VueShop` grâce à l’attribut `largeur`. Cela permet d’avoir un affichage visuellement agréable et facilement compréhensible pour l’utilisateur. En utilisant cette approche, nous avons pu créer une classe `VueShop` flexible et adaptée à différents formats de shop selon le nombre de cartes. Nous avons également pu alléger la classe `VuePartie` en séparant cet affichage en une classe dédiée. Cela nous a permis de simplifier le processus de modification de l’affichage du shop.

Pour gérer l’empilement de cartes dans le shop (lorsqu’il y a plusieurs cartes identiques empilées les unes sur les autres), nous avons dû réviser notre approche initiale. Nous avons d’abord essayé d’utiliser le concept de `QPaintEvent`, mais cela s’est avéré infructueux. Nous avons finalement opté pour une solution plus simple et efficace en utilisant un `QLabel` avec un arrière-plan transparent qui affiche le nombre de cartes empilées. Cette approche a été simple à mettre en œuvre et a permis de

gérer l'empilement de cartes de manière claire et efficace dans l'interface utilisateur.

Pour améliorer l'expérience de jeu, nous avons souhaité permettre à l'utilisateur d'acheter une carte directement à partir de l'interface graphique du shop, sans avoir à utiliser une fenêtre distincte. Pour ce faire, nous avons créé un slot qui réagit au signal `carteClicked` émis par la classe `VueCarte` et qui affiche une grande version de la carte sélectionnée dans une nouvelle fenêtre lorsque l'utilisateur clique dessus. Dans cette nouvelle fenêtre, nous avons ajouté un bouton "acheter le bâtiment" qui est connecté à un deuxième slot de la classe `VueShop`. Lorsque ce bouton est cliqué, ce slot envoie une notification à la partie indiquant que le joueur a acheté le bâtiment correspondant (en appelant la méthode `acheter_batiment` de `Partie`) et déclenche la suite du tour (en appelant la fonction `suite_tour` de `Partie`). Pour éviter les bugs, nous avons "sécurisé" ce bouton en le rendant cliquable uniquement durant lorsque le joueur est un joueur "physique", que l'on se situe dans la phase d'achat et quand le joueur a les moyens de s'offrir la carte sélectionnée. Ainsi, l'utilisateur peut acheter une carte de manière simple et intuitive à partir de l'interface graphique du shop sans risque de faire planter le programme.

3.7 Affichage de l'entête

Afin de rendre l'interface utilisateur la plus conviviale possible, nous avons intégré un "entête" d'informations en haut de la page qui affiche les informations de base nécessaires au bon déroulement de la partie, comme la valeur des dés, le joueur actuel et les éditions de jeu choisies. Cet entête est composé d'éléments de base de Qt, tels que `QLabel` et `QLCDNumber`. Nous avons également ajouté un bouton qui permet de gérer un cas particulier de la mécanique d'achat. En effet, lorsque le joueur est en phase d'achat, il a le choix entre acheter un bâtiment, un monument, ou ne rien faire. Dans ce dernier cas, il suffit de cliquer sur ce bouton pour envoyer une notification à la partie indiquant que le joueur n'a rien acheté. Pour éviter les bugs, ce bouton n'est actif que lorsque le joueur est en phase d'achat. Grâce à ces éléments, l'interface utilisateur est facile à comprendre et à utiliser pour l'utilisateur.

3.8 Vue avec les informations

Nous avons développé la classe `VueInfo` qui est un `vbox` contenant tous les éléments nécessaires à l'affichage des informations destinées à l'utilisateur. Cette classe comprend un `QLabel` permanent qui affiche des informations importantes en permanence, ainsi qu'un `QWidget` avec un layout vertical qui contient un `QScrollArea` pour permettre à l'utilisateur de parcourir toutes les informations affichées. Le layout vertical contient également une liste de `QLabel` qui sont utilisés pour afficher les informations de manière temporaire. La classe dispose d'un constructeur permettant de créer l'objet avec un `QWidget` parent optionnel, ainsi que d'une méthode `add_info` qui permet d'ajouter facilement des messages à l'affichage principal de l'utilisateur. Grâce à cette classe, il est facile d'ajouter des informations destinées à l'utilisateur et de s'assurer qu'elles sont affichées de manière claire et visible.

4 Évolutions et limites de notre projet

Au cours de notre projet, nous avons dû réduire certaines de nos ambitions initiales en raison de limites techniques ou de contraintes de temps. Bien que nous aurions aimé inclure ces éléments dans la version définitive, ils n'ont pas pu être implémentés. Voici quelques exemples de fonctionnalités que nous aurions aimé ajouter, mais qui ont finalement été abandonnées :

4.1 Ambitions graphiques

Ajouter des avatars aux joueurs

Nous aurions aimé intégrer des avatars pour les joueurs afin de rendre le jeu plus immersif et dynamique. Pour les joueurs contrôlés par l'IA, nous aurions pu sélectionner aléatoirement une image à partir d'un dossier prédéfini. Pour les joueurs humains, nous aurions proposé la possibilité de choisir leur propre avatar, ce qui aurait ajouté une touche personnelle au jeu. Cette fonctionnalité aurait également permis de mieux distinguer les joueurs entre eux durant les parties.

Ajouter le visuel des pièces

L'idée de représenter visuellement les pièces au lieu de simplement afficher un montant en argent a été présente dès le début du projet, nous en avons même parlé dans un de nos rapports. Nous avons commencé à implémenter cette fonctionnalité en préparant les images des pièces et en créant une méthode permettant de déterminer la distribution de pièces pour un joueur en fonction de sa quantité d'argent. Cependant, nous avons dû mettre cette caractéristique de côté pour nous concentrer sur le développement des autres vues, qui nécessitaient une grande quantité de travail.

Ajouter le visuel des dés

Nous avons eu l'idée de créer une représentation visuelle pour les dés afin de rendre le jeu plus immersif et interactif. Nous avons donc commencé à chercher des modules permettant d'afficher des images animées de dés sous forme de "gifs". Cependant, la mise en place de cette fonctionnalité peut s'avérer complexe et n'est pas essentielle pour le fonctionnement du jeu. Nous avons donc décidé de la mettre de côté pour nous concentrer sur les autres aspects du projet.

4.2 Autres ambitions

Ajout d'une musique

Nous aurions souhaité ajouter une bande-son en fond sonore au jeu afin de lui donner plus de caractère et de profondeur. Cependant, la gestion de la lecture de la musique en parallèle du jeu peut s'avérer complexe et n'est pas essentielle au bon fonctionnement du logiciel. La lecture d'une musique nécessite l'installation du module Multimédia de Qt, entraînant des problèmes de portabilités pour les utilisateurs n'ayant pas installé ce module spécifique.

Historique et tableau des scores des anciennes parties

Il aurait été intéressant d'enregistrer et de pouvoir accéder aux scores de parties précédentes afin de pouvoir s'y référer et de créer un sentiment de compétition entre les joueurs. Cela aurait permis aux utilisateurs de suivre leur progression au fil du temps et de se mesurer à leurs propres performances passées, ainsi que celles de leurs amis ou adversaires. Cette fonctionnalité aurait également ajouté une dimension supplémentaire de défi et de motivation pour les joueurs.

Possibilité d'enchaîner plusieurs parties

Nous avons envisagé de proposer à l'utilisateur la possibilité de recommencer une partie une fois qu'elle est terminée. Cette fonctionnalité aurait permis aux joueurs de s'amuser à nouveau avec le même jeu, de tester de nouvelles stratégies ou de simplement passer un bon moment en redécouvrant le jeu. Malheureusement, nous n'avons pas eu le temps d'implémenter cette caractéristique dans la version définitive du logiciel.

Comportement des IA

Nous avons implémenté trois intelligences artificielles (IA) dotées de comportements uniques. Leur comportement est toutefois directement intégré dans le code des différentes méthodes de notre application, ce qui rend la création de nouvelles IA ou la modification des comportements des IA existantes plus complexe à mettre en œuvre. Nous aurions aimé proposer une solution plus flexible et modulable pour gérer le comportement des IA, mais nous n'avons pas réussi à trouver une solution satisfaisante dans les limites de temps et de ressources dont nous disposions.

Adaptabilité du code

A l'avenir, nous pourrions faire en sorte de rendre le jeu opérationnel sur d'autres plateformes telles que les tablettes ou les téléphones portables. Pour cela, il nous faudra être en mesure de gérer les différents cas possibles de plateformes et de taille d'écran. Ceci permettra d'améliorer l'expérience utilisateur.

5 Liste des tâches réalisées par chacun

Cette partie fait suite aux explications des rôles de chacun. Elle est dédiée au listage des tâches réalisées par chaque membre du groupe depuis le début du semestre. Ainsi, les durées indiquées sur chaque tâche sont propres à une personne et correspondent au temps passé dessus. Les durées pour les tâches faites ensemble correspondent à une estimation du temps que nous avons passé en cumulé.

Répartition du travail sous forme de camembert

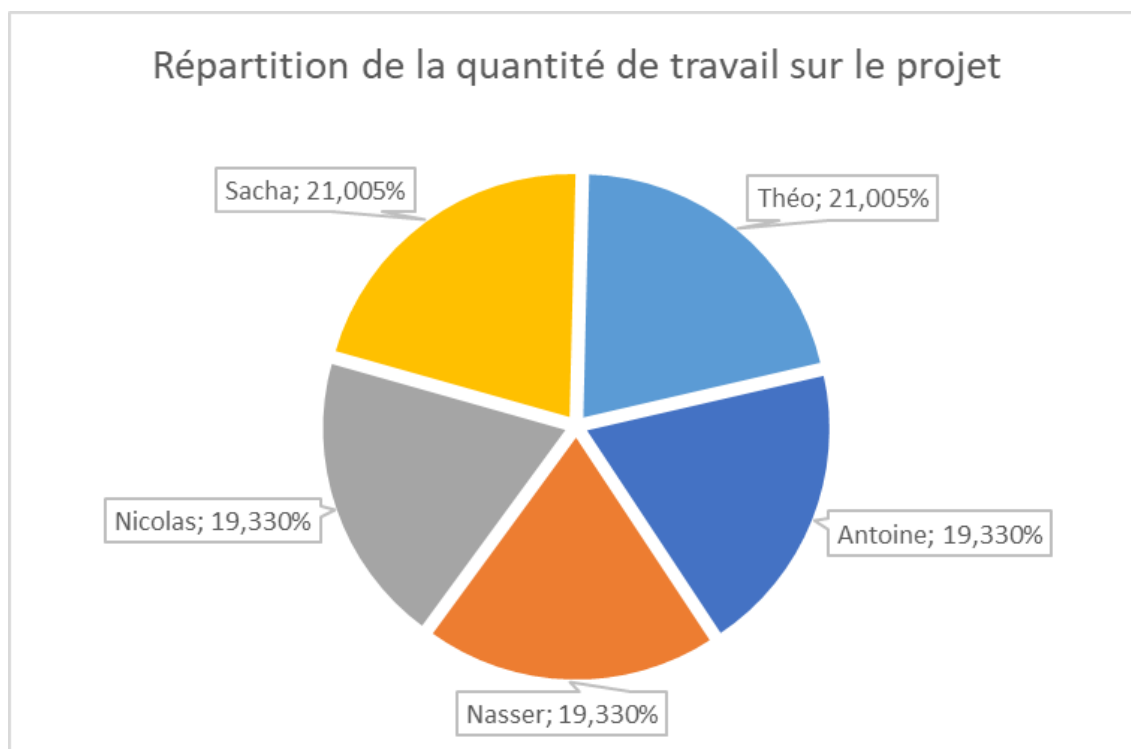



















FIGURE 4 – Répartition du travail version console

Fait en groupe :













-  : Architecture principale, avec toutes les réflexions qui y sont liées (35 heures)
-  : Partie console de l'application (100 heures)
-  : Évolution de l'architecture vers une architecture Qt (15 heures)
-  : Évolution de l'application console vers sa version Qt (130 heures)
-  : Débogage de la version console, session avant les vacances de Noël (30 heures)
-  : Débogage de la version Qt, fait pendant l'évolution (40 heures)

Fait par Antoine :







-  : Création des cartes violettes
-  : Création de fonctions de services dans la version console







-  : Optimisation des fonctions et de leurs ressources mémoires (passage par références, pointeurs,...)
-  : Implication dans la création et la maintenance de la classe Pioche
-  : Implication dans la création et la maintenance de la classe EditionDeJeu
-  : Implication dans la création, l'amélioration et dans la maintenance de VueCarte
-  : Implication dans la création, l'amélioration et dans la maintenance dans VueJoueur
-  : Création du lien entre le menu et la VuePartie
-  : Mise à jour des fonctions de service de VuePartie et VueJoueur en version Qt
-  : Implication dans la copie des images des cartes du jeu
-  : Correctifs de coquilles et débogage dans le code de la version Qt

Fait par Nasser :


















-  : Création des cartes rouges
-  : Optimisation des attributs de la classe Monument afin de pouvoir d'optimiser leur affichage pour la version Qt
-  : Implication dans la création, l'amélioration de la classe Shop
-  : Implication dans la création et dans l'amélioration de VueCarte
-  : Implication dans la création, l'amélioration et révision dans VueJoueur
-  : Implication dans la maintenance de la classe VueInfo
-  : Création du singleton dans la Partie
-  : Nettoyage et révision des inclusions pour la version graphique
-  : Création du défilement des VueJoueur à travers des boutons (Update de la VueJoueur)
-  : Implication dans la gestion des IA pour la version Qt
-  : Implication dans la suppression des fonctions et des attributs inutilisés du projet version Qt
-  : Correctifs de coquilles et débogage dans le code de la version Qt

Fait par Nicolas :






-  : Création des cartes bleues
-  : Création de fonctions usuelles pour faciliter les sélections
-  : Ajout du scan des cartes manquantes à la version physique Deluxe
-  : Adaptation de jouer_tour à la version Qt
-  : Implication dans la création et amélioration de la classe Pioche
-  : Implication dans l'amélioration et la révision de la classe EditionDeJeu













-  : Implication dans la création et l'amélioration de la classe VueShop
-  : Implication dans la création et l'amélioration de la classe VuePioche
-  : Création de la section entête dans VuePartie
-  : Nettoyage et révision des inclusions pour la version graphique
-  : Implication dans l'amélioration et le débogage de VueJoueur
-  : Correctifs de coquilles et débogage dans le code de la version Qt

Fait par Sacha :

-  : Création des cartes monument
-  : Création des scans d'images du jeu et leur redimensionnement
-  : Création et maintenance de la classe Pioche
-  : Débogage de la version console
-  : Implication dans le débogage de la version Qt
-  : Révision des inclusions des fichiers présents dans les dossiers controleur et Joueur
-  : Implication dans l'amélioration et le débogage des différentes cartes (monuments, bleues, rouges, vertes, violettes)
-  : Création et gestion des IA pour les versions console et Qt
-  : Révision de la classe Partie de la version console
-  : Adaptation de jouer_tour et jouer_partie à la version Qt
-  : Implication dans la création et la maintenance de la classe Shop
-  : Implication dans la création et la maintenance de la classe EditionDeJeu
-  : Implication dans la création et l'amélioration de la classe VueShop
-  : Implication dans la création et l'amélioration de la classe VuePioche
-  : Création de la classe VueInfo et inclusion dans VuePartie
-  : Implication dans la suppression des fonctions et des attributs inutilisés du projet versions console et Qt
-  : Gestion des derniers bugs de la version graphique

Fait par Théo :

-  : Création des cartes vertes
-  : Création de la classe d'exception et son application
-  : Maintenance et modification du fonctionnement du singleton de Partie
-  : Implication dans l'amélioration et le débogage/recodage des cartes (vertes, bleues, rouges, violettes, monuments)
-  : Nettoyage et révision des inclusions des cartes et fichiers présents dans controleur pour la version console

-  : Implication dans la création et la gestion des IA pour la version console
-  : Révision de la classe Partie de la version console
-  : Débogage de la version console
-  : Création, scan et recadrage des images de jeu
-  : Conversions de toutes les images pour les rendre utilisables
-  : Implication dans la création et amélioration de la classe Pioche
-  : Création, maintenance et amélioration des formulaires de menu et du code de traitement associé pour la version Qt
-  : Recodage de l'initialisation de la partie pour la version graphique (`get_instance()` et constructeur de partie)
-  : Implication dans la maintenance et débogage de VueInfo
-  : Implication dans l'amélioration et le débogage de VueShop
-  : Implication dans l'amélioration et le débogage de VueJoueur
-  : Gestion des derniers bugs de la version graphique

Conclusion

Au cours de cette dernière phase du projet Machi Koro, nous avons développé une interface graphique en utilisant le framework Qt, qui était une première pour nous. Cette interface graphique vient compléter la version console du jeu que nous avons déjà conçue et développée au cours des deux derniers mois. Bien que la version console était déjà fonctionnelle et avait été testée avec succès, nous avons découvert quelques bugs mineurs lors de notre passage à Qt. Nous avons pris le temps de les corriger afin de garantir une expérience de jeu fluide et agréable pour les utilisateurs. En tout, ce projet a duré environ quatre mois, avec un mois consacré à la création de l'interface graphique Qt et deux mois sur la version console. Nous sommes fiers de ce que nous avons accompli et nous espérons que l'ajout de cette interface graphique apportera une valeur supplémentaire au jeu et suscitera l'intérêt des utilisateurs.

Pendant cette phase du projet, nous avons dû consacrer beaucoup de temps et d'énergie à la modélisation et à la compréhension des concepts clés de Qt et de leur fonctionnement. Ce fut une étape cruciale pour réussir à mettre en place une interface graphique solide et intuitive pour notre jeu. Nous avons dû apprendre de nouvelles compétences et mettre en pratique nos connaissances de la programmation orientée objet pour réussir à maîtriser cet outil de développement. Grâce à cet effort important de modélisation et de compréhension, nous avons réussi à créer une interface graphique de qualité qui offre une expérience de jeu fluide et agréable aux utilisateurs.

Ce projet nous a offert l'opportunité de mettre en pratique les connaissances théoriques que nous avons acquises au cours de nos études. Nous avons pu appliquer les concepts de programmation orientée objet que nous avons appris lors de nos cours magistraux et les mettre en œuvre dans ce projet. Cette expérience a été très enrichissante pour nous et nous a permis de mieux comprendre comment ces concepts sont mis en œuvre dans la pratique. Nous sommes fiers de ce que nous avons accompli et espérons que ce projet sera une référence pour nos futures carrières dans l'informatique.

Ce projet nous a permis de développer de solides compétences en programmation orientée objet et en création d'interfaces graphiques. Si nous avons besoin de travailler dans une équipe pour concevoir un logiciel avec une interface graphique à l'avenir, nous serons en mesure de le faire de manière efficace et professionnelle. Nous avons également appris à prêter attention aux problématiques de conception logicielle et à les prendre en compte dans notre travail, afin de garantir un logiciel fonctionnel et de qualité.

En résumé, grâce à ce projet, nous avons acquis les compétences nécessaires pour travailler efficacement en équipe pour concevoir un logiciel avec une interface graphique tout en prenant en compte les problématiques de conception logicielle. Nous sommes prêts à mettre ces compétences en pratique dans un environnement professionnel.

ANNEXES

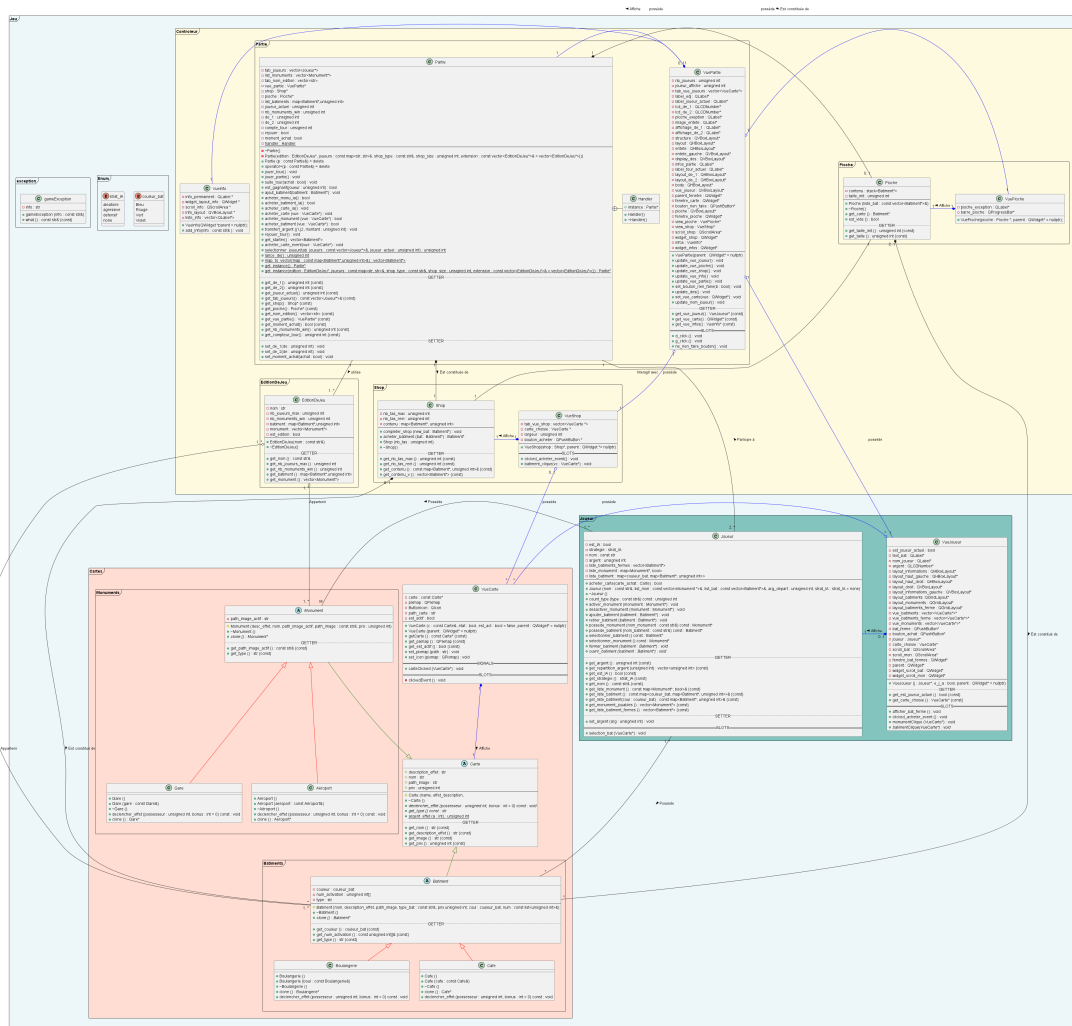


FIGURE 5 – Architecture finale

Table des figures

1	Affichage des cartes dans la VueJoueur	15
2	Affichage de la carte dans une fenêtre pop-up	16
3	Vue du Joueur	16
4	Répartition du travail version console	21
5	Architecture finale	26