PROJET SOKOBAN IMDS4A
Année 2021/2022

Manuel Développeur

Réalisé par :

Simon AMBLARD
Alexandre CANCEL

POLYTECH
CLERMONT-FERRAND

Table des Matières :

Table des figures :

1. Analyse et conception du logiciel

1.1 Présentation du jeu

Le Sokoban est un jeu vidéo de réflexion inventé au Japon. Le jeu original a été écrit par Hiroyuki Imabayashi et comportait 50 niveaux. Il remporte en 1980 un concours de jeu vidéo pour ordinateur. Plus tard Hiroyuki Imabayashi est devenu président de la compagnie japonaise Thinking Rabbit qui détient aujourd'hui les droits sur le jeu depuis 1982.

Le but du Sokoban est que le joueur doit ranger des caisses sur des cases cibles. Il peut se déplacer dans les quatre directions à l'aide de différentes touches (ici haut, bas, gauche et droite), et pousser les caisses. Une fois toutes les caisses rangées c'est-à-dire que chaque case marquée est occupée par une caisse alors le niveau est réussi et le joueur peut passer au niveau suivant. Chaque niveau a une architecture différente généralement plus difficile au fur et à mesure que l'on avance dans les niveaux. L'idéal est de réussir avec le moins de coups possibles.

Cependant, le joueur a des actions restreintes avec l'environnement du jeu comme :

- On ne peut pousser qu'une seule caisse à la fois.
- On ne peut pas traverser les murs.
- On ne peut pas passer par-dessus des caisses.
- On peut seulement pousser les caisses (on ne peut donc pas tirer les caisses).

1.2 Cas d'utilisation

En premier lieu, nous avons élaboré le diagramme de cas d'utilisation. Ce dernier permet de faire ressortir les acteurs et les fonctions offertes par le système. C'est pourquoi, nous avons choisi de réaliser ce type de diagramme afin de voir à quoi l'utilisateur sera confronté lors du lancement du jeu.

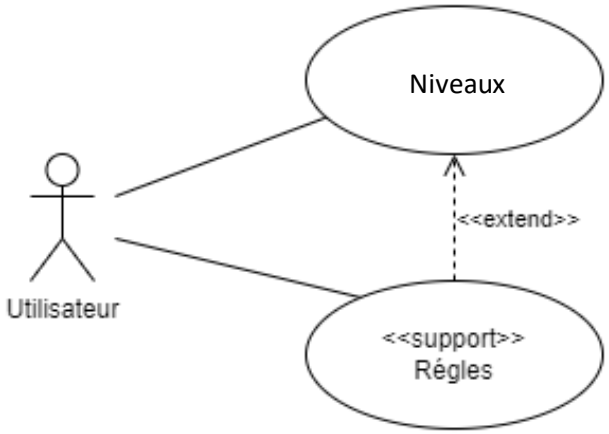

Figure 1 : Diagramme de cas d'utilisation

On voit que l'utilisateur a deux choix : **Niveaux** et **Règles**. D'abord l'option **Niveaux** va permettre au joueur de choisir quel niveau il veut lancer et ainsi commencer une partie. Et enfin, l'option Règles affichera clairement à l'utilisateur les différentes règles du jeu ainsi que les commandes pour jouer.

1.3 Analyse

La phase d'analyse du jeu permet d'élaborer le diagramme de séquence système, le diagramme classe d'analyse et diagramme d'état/transition.

1.3.1 Diagramme de séquence du système

Le diagramme de séquence va nous permettre de décrire les interactions entre l'utilisateur et le jeu en insistant sur l'aspect temporel.

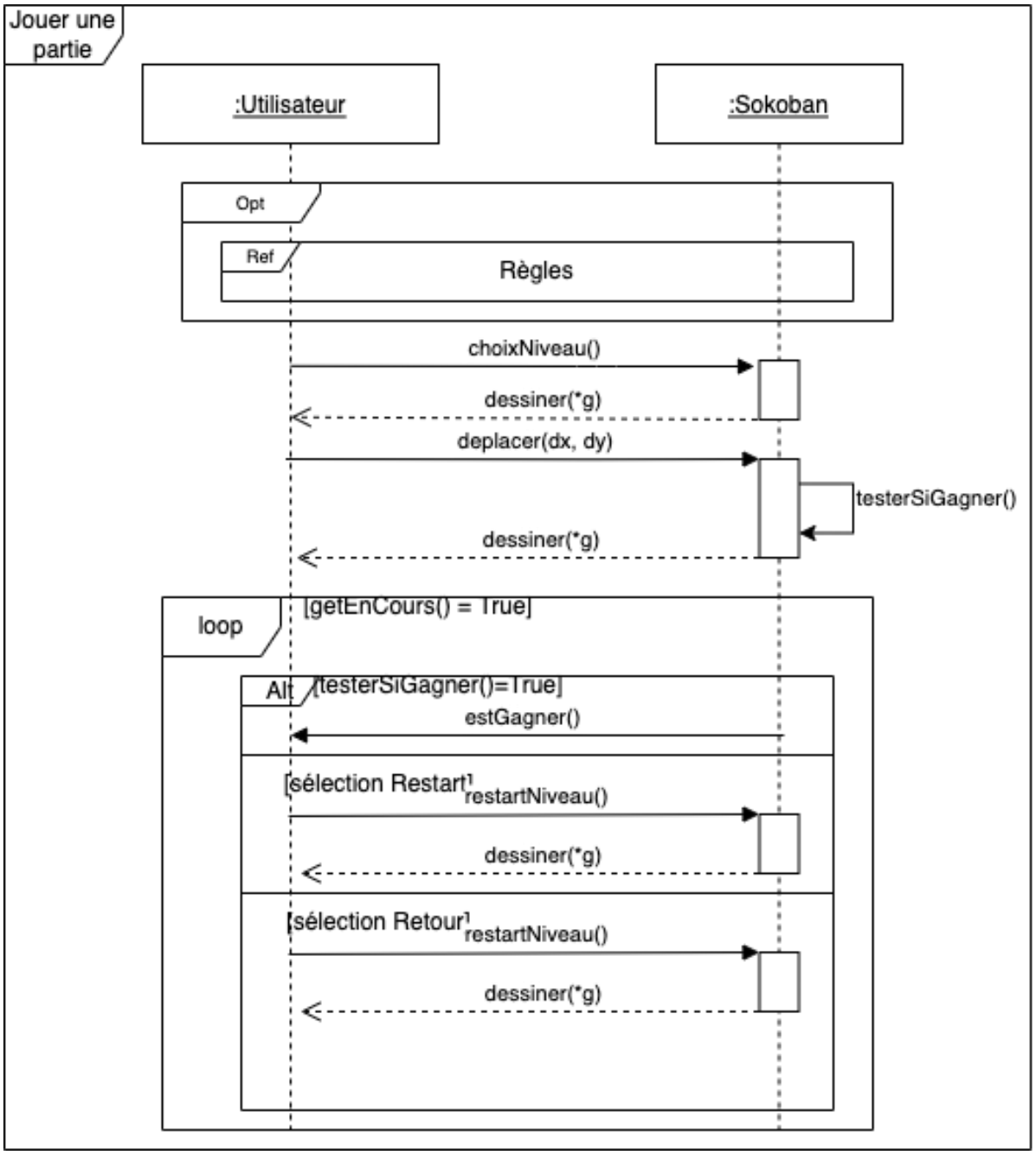

Figure 2 : Diagramme de séquence système

Sur ce diagramme, nous avons décrit le fonctionnement du système du jeu au cours du temps. L'utilisateur peut d'abord choisir d'ouvrir les Règles. Ensuite, il va choisir un niveau (**choixNiveau**()) ce qui correspond dans notre programmation à cliquer sur l'un des boutons correspondant à un niveau précis. Une fois le niveau choisi, la partie commence et on affiche la grille correspondante (**dessiner**(*g)).

Tant que le niveau est en cours l'utilisateur va pouvoir déplacer le personnage (**deplacer**(dx, dy)).A chaque déplacement, le jeu va tester si l'utilisateur a gagné (**testerSiGagner**()). Si oui, le jeu affiche que le niveau est gagné et va débloquer le niveau suivant. L'utilisateur aura accès, s'il le souhaite, à relancer la partie (**on_pRestart_clicked**()) retourner au début du niveau ou de la quitter (**on_pRetour_clicked**()) il sera redirigé vers la page d'accueil avec des boutons présents sur l'interface.

1.3.2 Diagramme de Classe d'analyse

Ce diagramme doit définir les classes qui modélisent les entités ou les concepts présents dans le jeu "Sokoban". Nous avons identifié les entités ou concepts du jeu et on a ajouté les associations et les attributs.

Pour décrire la structure du système du jeu Sokoban, un diagramme de classe permet de modéliser les classes, les attributs, les opérations et les relations entre les objets.

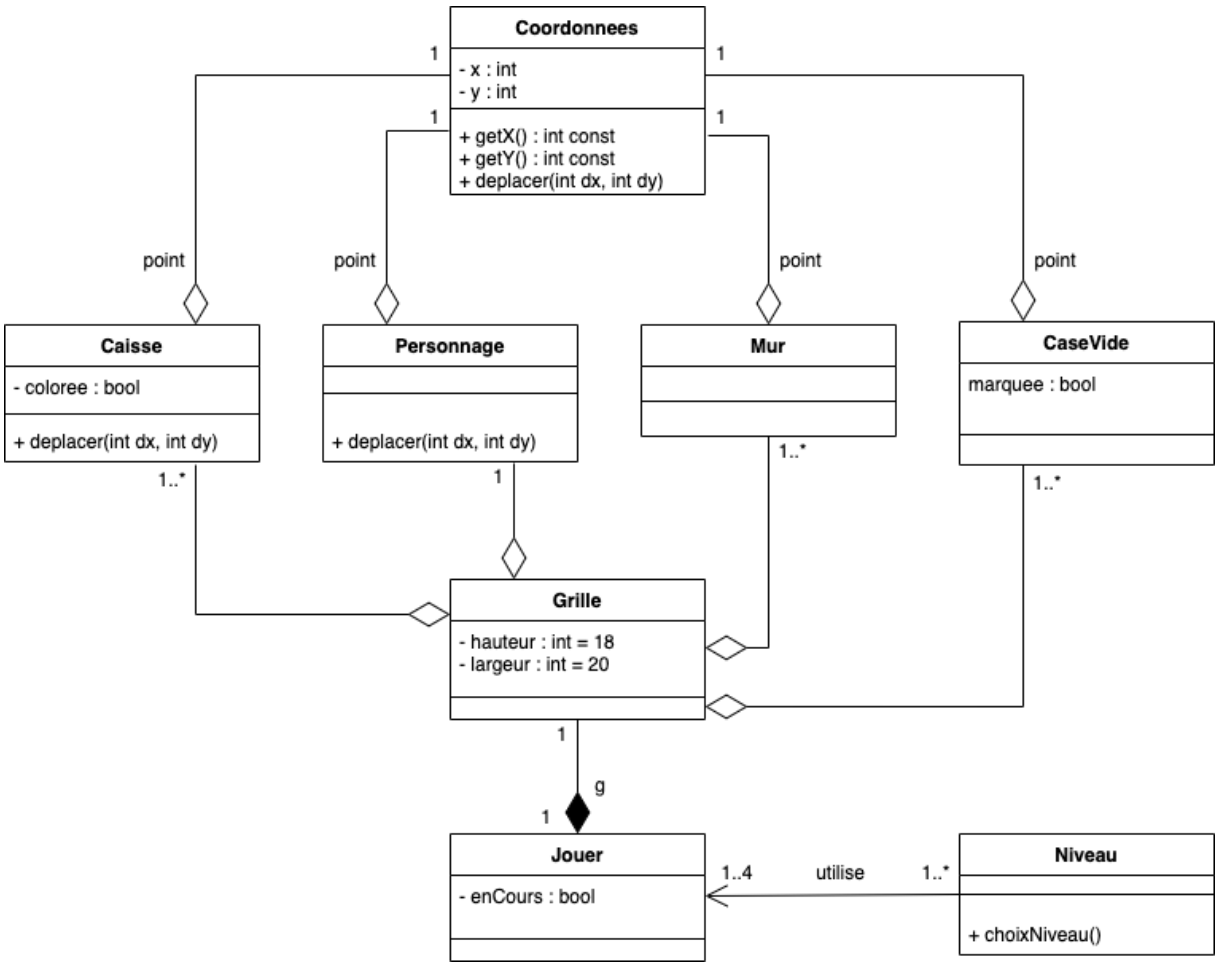

Figure 3 : Diagramme de classe d'analyse

Nous avons représenté le jeu à travers différentes classes. Le jeu se déroule dans la classe **Jouer** et est composé d'une **Grille** de **Case** correspondant à un objet du jeu. De plus chaque case possède un attribut de la classe **Coordonnées** permettant de les placer dans l'interface pour l'affichage.

1.4 Conception

1.4.1 Diagramme de Séquence de déplacer joueur

Nous avons décidé de décrire le cheminement des méthodes, pour déplacer le joueur, à travers un diagramme de séquence spécifique à cette action en montrant clairement les interactions avec les autres classes.

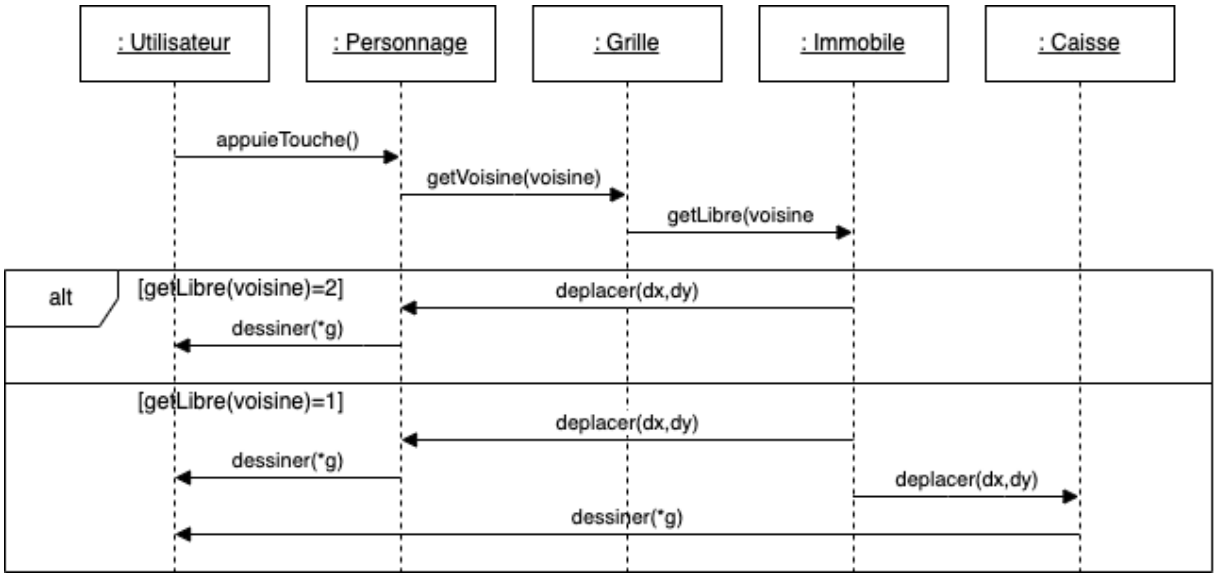

Figure 4 : Diagramme de séquence de la méthode déplacer personnage

Sur ce diagramme, nous avons traité en particulier l'action « déplacer joueur ». Une fois la partie lancée, l'utilisateur aura le contrôle sur le joueur. En effet, il a la possibilité de le diriger pour jouer. L'utilisateur appuie sur une touche pour déplacer le joueur vers une direction, cela va engendrer un traitement des cas suivants :

On identifie la case voisine du personnage (**getVoisine**(voisine)) et on vérifie si elle est libre c'est-à-dire si **getLibre**(voisine) renvoie 2.

- Si c'est le cas, on déplace le personnage et on affiche son nouvel emplacement,

- Si la case voisine est une caisse c'est-à-dire si **getLibre**(voisine) renvoie 1, alors on déplace le personnage ainsi que la caisse et on affiche leur nouvel emplacement.

- Dans tout autre cas, le déplacement ne sera pas possible et le personnage ne bougera pas.

1.4.2 Diagramme de classe de conception

Le diagramme de classe technique a été réalisé pour décrire plus précisément les classes et les objets à utiliser, leurs différentes relations et enfin, leurs méthodes, dans une approche plus orientée programmation. Nous avons ainsi repris le diagramme d'analyse en ajoutant des informations. Nous avons alors précisé les méthodes et leurs types d'entrée/sortie et si elles sont publiques, privées ou protégées.

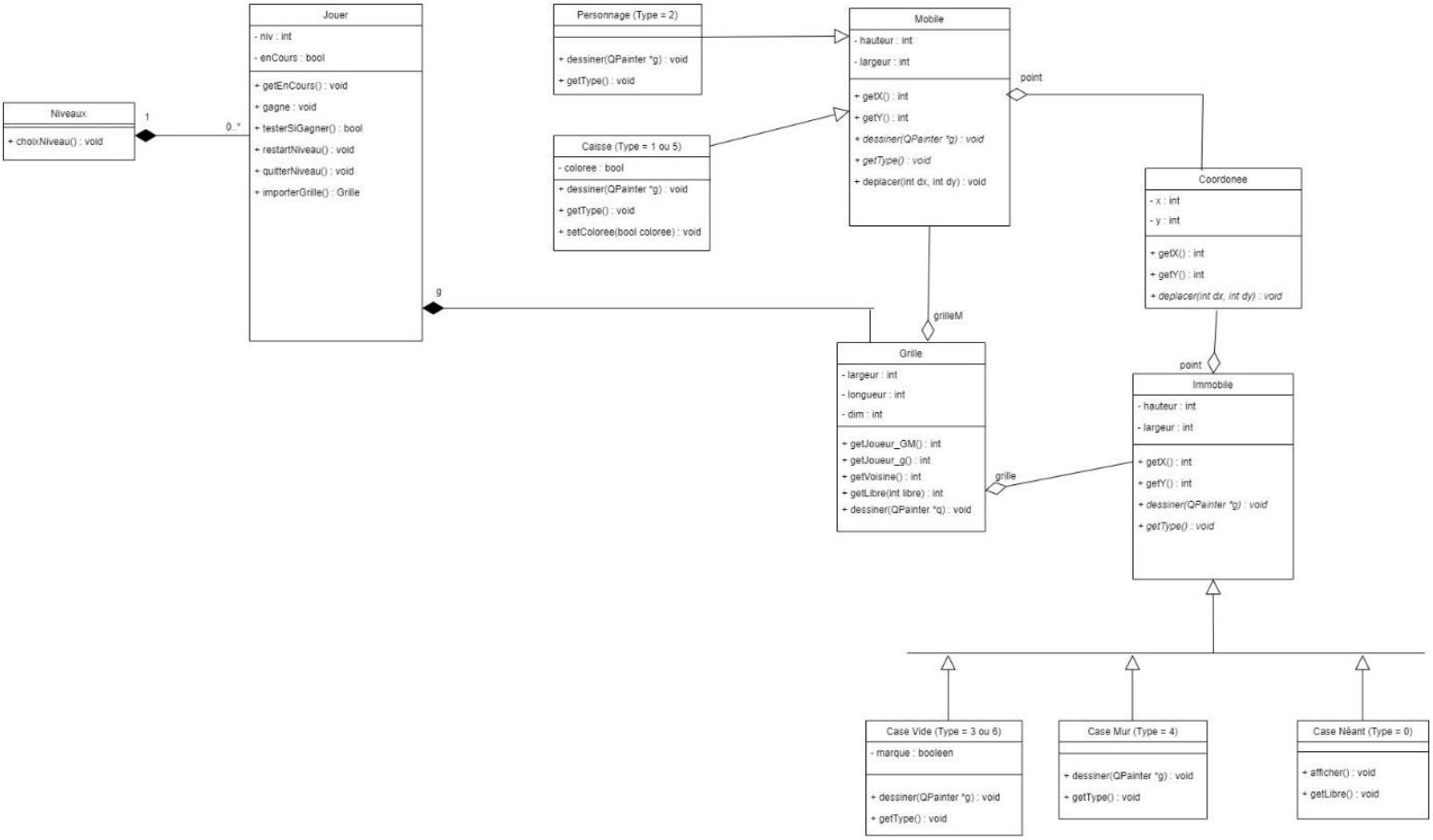

Figure 5 : Diagramme de classe

On retrouve les méthodes utilisées dans les différents diagrammes, en particulier ceux de séquences. En effet, par exemple dans la classe "Partie", il y a la méthode « **choixNiveau**() »utilisée dans le diagramme séquence système, correspondant au choix du niveau par le joueur.

- La classe **Niveaux** contient :

La méthode **choixNiveau**() : Elle permet de choisir un niveau à jouer et de charger les données de celui-ci en appuyant sur le bouton correspondant.

- La classe **Jouer** contient :

L'attribut g : il correspond à la grille correspondant au niveau.

L'attribut niveau : il correspond à l'entier du niveau par exemple niveau = 1 si l'utilisateur a sélectionné le niveau 1.

La méthode **on_pRestart_clicked**() : Elle permet de recommencer le niveau si l'utilisateur appuie sur le bouton correspondant.

La méthode **on_pRetour_clicked**() : Elle permet d'ouvrir l'interface de la classe **Niveau** si l'utilisateur sur le bouton correspondant.

La méthode **testerSiGagner**() : Elle renvoie un booléen vrai si les caisses sont toutes placées au bon endroit et faux sinon.

La méthode **Gagne**() : Elle permet d'ouvrir une fenêtre de dialogue **Résultat** pour indiquer à l'utilisateur qu'il a terminé le niveau.

- La classe **Grille** contient :

L'attribut hauteur: il correspond au nombre de cases en hauteur de la grille.

L'attribut largeur: il correspond au nombre de cases en largeur de la grille.

L'attribut dim: il correspond au nombre d'éléments mobiles sur la grille (nombre de caisses plus le joueur).

L'attribut grilleM: c'est un vecteur contenant les caisses et le joueur avec leurs coordonnées sur la grille. Nous avons utilisé la bibliothèque `vector` de C++ (Microsoft, s.d.).

L'attribut grille: c'est un vecteur contenant les éléments de la grille (les cases néants, vides et les murs) avec leurs coordonnées. Nous avons utilisé la bibliothèque `vector` de C++ (Microsoft, s.d.).

La méthode **getJoueur_gM**() : Elle renvoie l'emplacement du joueur dans grilleM.

La méthode **getJoueur_g**() : Elle renvoie l'emplacement du joueur dans grille.

La méthode **getVoisine**(int voisine) : Elle renvoie l'emplacement d'une caisse dans grilleM en voisine du joueur sinon elle renvoie -1.

La méthode **getLibre**(int libre) : Elle renvoie 1 si la case en libre du joueur est une caisse , 2 si la case en libre à côté du joueur est vide sinon elle renvoie 0.

La méthode **dessiner**(QPainter * g) : Elle affiche dans l'interface **Jouer** la grille avec les caisses et le joueur par-dessus celle-ci.

- La classe **Immobile** contient :

L'attribut point : il correspond aux coordonnées de l'élément immobile.
L'attribut hauteur : il correspond à la hauteur en pixels de l'image de l'élément immobile.

L'attribut largeur : il correspond à la largeur en pixels de l'image de l'élément immobile.

La méthode **getX**() : il renvoie l'abscisse de l'élément immobile.

La méthode **getY**() : il renvoie l'ordonnée de l'élément immobile.

La méthode ***getType***() : méthode abstraite qui renvoie un chiffre (3 pour les cases vides, 6 pour les cases vides marquées, 4 pour les murs et 0 pour les cases néants).

La méthode ***dessiner***(QPainter * imm) : méthode abstraite qui permet d'afficher un élément immobile.

- La classe **Mobile** contient :
  L'attribut point : il correspond aux coordonnées de l'élément mobile.
  L'attribut hauteur : il correspond à la hauteur en pixels de l'image de l'élément mobile.
  L'attribut largeur : il correspond à la largeur en pixels de l'image de l'élément mobile.

La méthode **deplacer**(int dx, int dy) : qui permet de déplacer la caisse ou le joueur de dx, dy sur la grille.

La méthode **getX**() : il renvoie l'abscisse de l'élément mobile.

La méthode **getY**() : il renvoie l'ordonnée de l'élément mobile.

La méthode ***getType***() : méthode abstraite qui renvoie un chiffre (1 pour les caisses, 5 pour les caisses marquées et 2 pour le personnage).

La méthode ***setColoree***(bool coloree) : méthode abstraite qui renvoie vrai ou faux en fonction de si le mobile est marqué ou non.

La méthode ***dessiner***(QPainter * mob) : méthode abstraite.

- Les classes **Caisse, Personnage** contiennent :

  L'attribut coloree (uniquement pour la classe Caisse, si le personnage est sur une case marquée, nous ne changeons rien) : c'est un booléen qui est vrai lorsque l'élément caisse est sur une case marquée faux sinon.

La méthode ***getType***() : renvoie un entier (1 pour les caisses, 5 pour les caisses marquées et 2 pour le personnage).

La méthode ***setColoree***(bool coloree) : met la valeur du booléen coloree dans l'attribut coloree.

La méthode ***dessiner***(QPainter * p) : Elle permet d'afficher l'élément mobile.

- Les classes **Mur, CaseVide** et **Neant** contiennent :

  L'attribut marquee (uniquement pour la classe CaseVide, inutile dans les autres classes) :

  c'est un booléen qui est vrai lorsque la case est marquée, faux sinon.

La méthode ***getType***() : Elle renvoie un entier (3 pour les cases vides, 6 pour les cases vides marquées, 4 pour les murs et 0 pour les cases néants).

La méthode ***dessiner***(QPainter * cv) : Elle permet d'afficher l'élément immobile.

2. Programmation du logiciel

2.1 Diagramme de composants

Un diagramme de composants a pour objectif d'illustrer la relation entre les différents composants d'un système. Le diagramme de composant permet au responsable de la planification d'identifier les différents composants afin que l'ensemble du système fonctionne selon le cahier des charges. Il permet d'aider à imaginer la structure physique du système, à faire ressortir les composants du système, leurs relations et à mettre en évidence le comportement de chaque service vis-à-vis de l'interface.

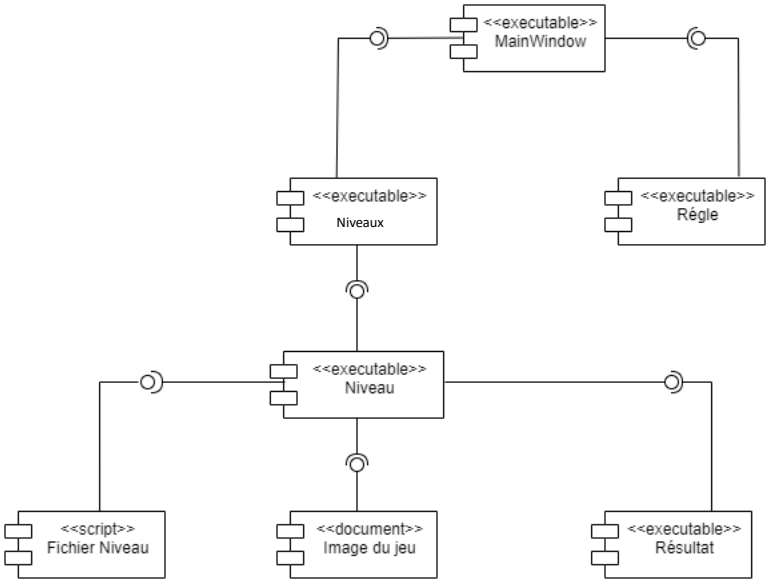

Figure 6 : Diagramme de composants

Dans notre cas, l'interface niveau a besoin de récupérer les scripts et les images dans des sous fichiers pour son affichage. De plus, l'interface **Résultat** n'est accessible que par l'interface Jouers. Même raisonnement pour les interfaces **Niveau** et **Règle** qui sont uniquement accessible par la **MainWindow**.

2.2 Architecture du logiciel

Ci-dessous l'organisation des fichiers et des dossiers de notre projet Sokoban :

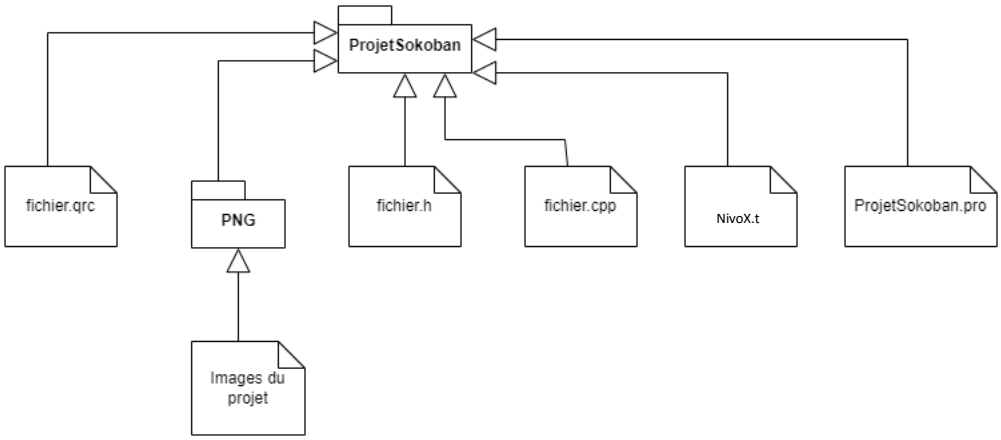

Figure 7 : Schéma de l'architecture du logiciel

2.3 Présentations des écrans

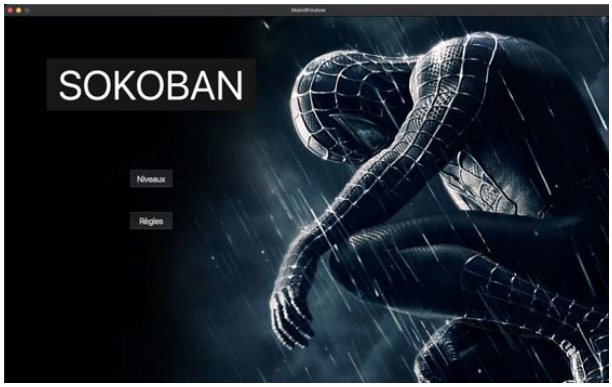

Figure 8 : Ecran de Menu

Écran de lancement que l'on obtient après avoir lancé l'exécution du code sur Qt. Il est généré par la classe MainWindow avec deux boutons :

- Niveaux : void MainWindow::**on_pNiveaux_clicked**();
- Règles : void MainWindow::**on_pRegle_clicked**();

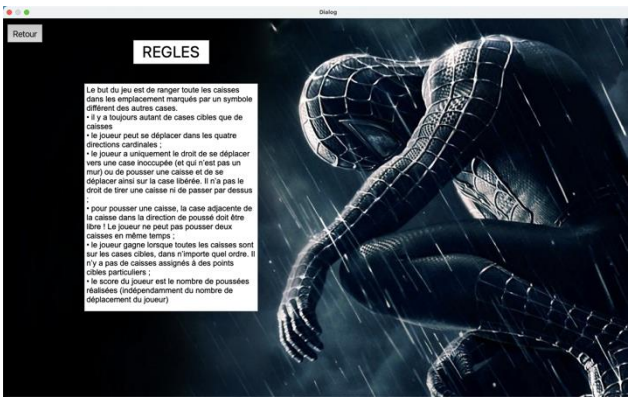

Figure 9 : Ecran des Règles

Une fois après avoir appuyé sur le bouton Règle de la fenêtre précédente, l'utilisateur est redirigé vers cette fenêtre où les règles du jeu lui sont expliquées. Cette fenêtre est gérée par la classe Règle avec un bouton :

- Retour : void Regle::**on_pushButton_clicked**();

Ce bouton permet de revenir à la fenêtre précédente c'est-à-dire la fenêtre MainWindow et ferme la fenêtre Règle.

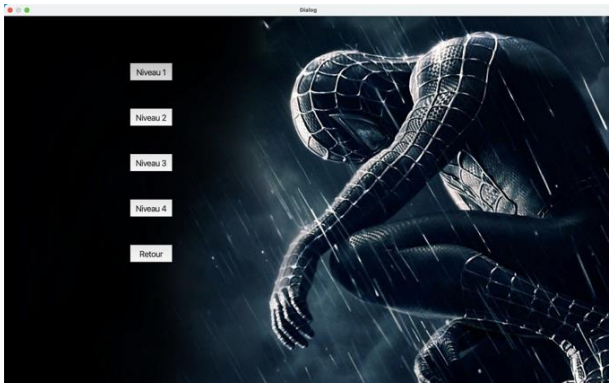

Figure 10 : Ecran des Niveaux

Une fois après avoir appuyé sur le bouton Jouer de la fenêtre MainWindow, l'utilisateur est redirigé vers cette fenêtre où il a le choix entre 4 niveaux. Cette fenêtre est gérée par la classe Niveaux avec comme boutons :

- Niveau 1 : void Niveaux::**on_pNiveau1_clicked**() ;
- Niveau 2 : void Niveaux::**on_pNiveau2_clicked**() ;
- Niveau 3 : void Niveaux::**on_pNiveau3_clicked**() ;
- Niveau 4 : void Niveaux::**on_pNiveau4_clicked**() ;
- Retour : void Niveaux::**on_pushButton_clicked**() ;

Les 4 premiers boutons permettent de lancer les niveaux correspondant puis le dernier bouton permet de revenir à la fenêtre précédente c'est-à-dire la fenêtre MainWindow et ferme la fenêtre Niveau.

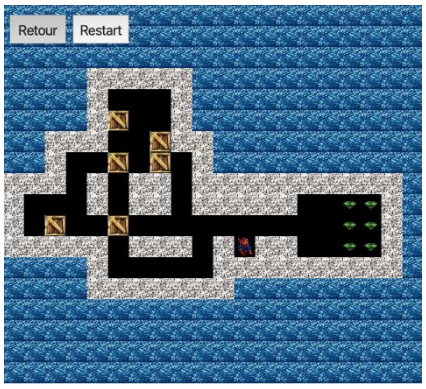

Figure 11 : Ecran du niveau 1

Une fois après avoir appuyé sur l'un des boutons Niveau de la fenêtre Niveau, l'utilisateur est redirigé vers une fenêtre comme celle-ci (ici nous avons la fenêtre du niveau 1). Cette fenêtre est gérée par la classe Jouer avec comme boutons :

- Retour : void Jouer::**on_pRetour_clicked**() ;
- Restart : void Jouer::**on_pRestart_clicked**() ;

Le premier bouton permet de revenir à la fenêtre précédente, c'est-à-dire la fenêtre Niveau et ferme la fenêtre du niveau en cours. Le second bouton permet de relancer le niveau en cours depuis sa grille de départ.

Figure 12 : Ecran de victoire

Une fois après avoir réussi l'un des niveaux, cette fenêtre va s'ouvrir automatiquement et en même temps la fenêtre de notre niveau va se fermer. Cette fenêtre est gérée par la classe Resultats avec comme boutons :

- Retour : void Resultat::**on_pRetour_clicked**() ;
- Restart : void Resultat::**on_pRestart_clicked**();
- Niveau Suivant : void Resultat::**on_pushButton_clicked**() ;

Le premier bouton permet de revenir à la fenêtre précédente, c'est-à-dire la fenêtre Niveau et ferme la fenêtre de Résultat. Le second bouton permet de relancer le niveau réussi depuis sa grille de départ. Le dernier bouton permet de directement passer au niveau suivant, si l'utilisateur termine

le niveau 4 qui est notre dernier niveau alors en cliquant sur Niveau Suivant, il retournera directement sur la grille du niveau 1.

4. Bibliographie

Documentation QT, Disponible sur : https://doc.qt.io/

Documentation C++, Disponible sur : https://docs.microsoft.com/fr-fr/cpp/cpp/?view=msvc-170

Initiation à doxygen C++, Disponible sur : https://franckh.developpez.com/tutoriels/outils/doxygen/

Le jeu Sokoban, Disponible sur : https://fr.wikipedia.org/wiki/Sokoban

Le diagramme de Gantt, Disponible sur : https://www.gantt.com/fr/