Rapport Final

Création du jeu :

PRÉPARÉ PAR :

- Lahmidi Oussama
- Hasnaoui Inas
- Fattas Amine
- AHBANE Abdellah
- Bouziti Nouhaila

ENCADRANTE :
Mme. Loubna BENCHIKHI

RESPONSABLES :
Mme. Maria ZRIKEM
M. Nabil EL MARZOUQI

Sommaire :

1. Remerciement :

Nous tenons tout d’abord à remercier Dieu le tout puissant et miséricordieux, qui nous a donné la force d’accomplir ce modeste travail.

En second lieu, nous tenons à remercier notre encadrante pour ses précieux conseils et son aide durant toute la période du travail.

Nos sincères remerciements vont également aux membres de jury pour l’intérêt qu’ils ont porté à notre recherche en acceptant d’examiner notre travail et de l’enrichir par leurs propositions.

2. Introduction :

Ce rapport présente le travail effectué par notre Groupe pour le module Réalisation de projet Informatique. Ce dernier était pour nous l'occasion de découvrir comment un projet se crée et comment la recherche en informatique se déroule.

Ce projet a pour sujet le jeu classique Mahjong Tower, qu'on a développé autrement avec de nouvelles règles et avec un thème Marocain(Mahjong++). Dans le cadre de notre projet, on a été engagé de mettre en place chaque fin de mois un rapport d'avancement de notre projet, ça fait 2 rapports intermédiaires et une séance avec l'encadrante chaque semaine.

Notre rapport sera composé de trois volets principaux :

- La première partie décrira le concept de jeu, c'est-à-dire sa définition et son fonctionnement ainsi que ses règles. Elle comporte aussi, une présentation sur comment l'interface de jeu fonctionne.
- Ensuite, la seconde partie concerne la Conception de projet, dans cette partie on va traiter les côtés interface de jeu et sa logique, en parlant des outils utilisés pour développer notre jeu et en décrivant les classes et les méthodes principales de notre code ainsi que leurs explications et définition d'une modélisation du jeu sous forme de graphe. Ensuite, on va décrire les algorithmes qu'on a implémentés.

- Enfin, la dernière partie présentera les difficultés qu'on a trouvées durant la réalisation de ce projet.

3. Guide de l'utilisateur :

3.1 Présentation de jeu :

Mahjong Tower aussi connu comme le solitaire de Shanghai, est un jeu individuel qui utilise des tuiles de Mahjong.
Les tuiles sont arrangées dans des dispositions spéciales avec leurs faces tournées vers le haut.
Une tuile qui peut être déplacée à gauche ou à droite sans déranger les autres tuiles est dite exposée. Des paires exposées ou des tuiles identiques (les tuiles à Fleurs et Saisons dans le même groupe étant considérées comme identiques) sont retirées de la disposition une à la fois, en exposant progressivement les couches inférieures du jeu. Le but du jeu est de vider la disposition en associant toutes les tuiles. Le jeu se termine lorsque la disposition est totalement vide, ou lorsqu'il n'y a plus de paires exposées restantes.

Problème :
La contrainte de Mahjong Tower est qu'il est un jeu 'Solo', il n'y a pas d'adversaire.

Solution proposée :
La solution proposée a été d'introduire un compte-à-rebours pour chaque joueur. Et ainsi on a introduit de nouvelles règles avancées, d'où le nom "Mahjong++"; c'est à dire une version avancée.

3.2 Règles du jeu :

Les règles du jeu : Mahjong++

- Sélectionner deux pièces identiques pour les faire disparaitre il faut que :
- Aucune pièce ne doit être collée à gauche ou à droite
- Il ne doit pas y avoir une pièce au-dessus
- Chaque joueur a 15 secondes pour le niveau facile, 20 secondes pour le niveau moyen et 25 secondes pour le niveau difficile pour jouer avant de passer le tour à son adversaire.
- Le score se calcule selon le tableau suivant:
- Le 1er pair = 10
- Le 2e pair = 20
- Le 3e pair = 40
- Etc...

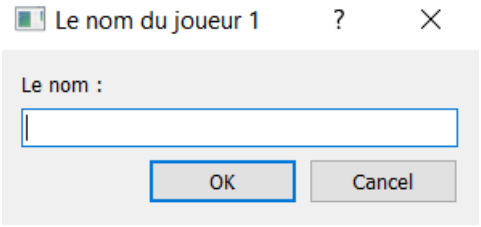
3.3 Comment le jeu fonctionne

La page d'accueil du jeu nous permet de choisir le mode du jeu (Joueur VS joueur ou joueur VS ordinateur) :

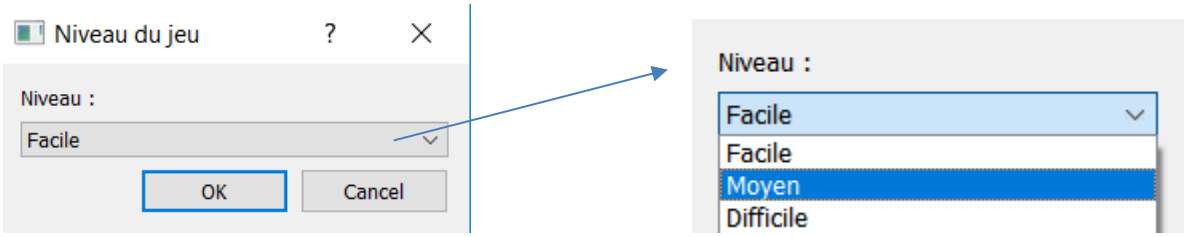
Page d'accueil

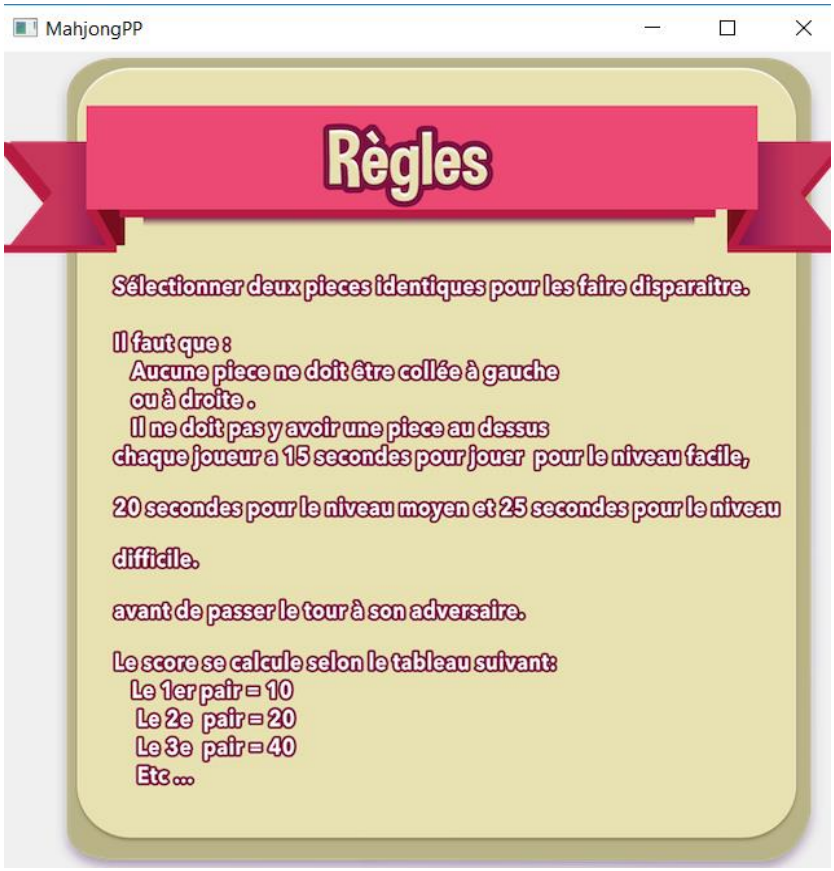
Puis nous entrons notre nom et le niveau auquel nous souhaitons démarrer notre partie :

Boite de Dialogue : Pour saisir les noms



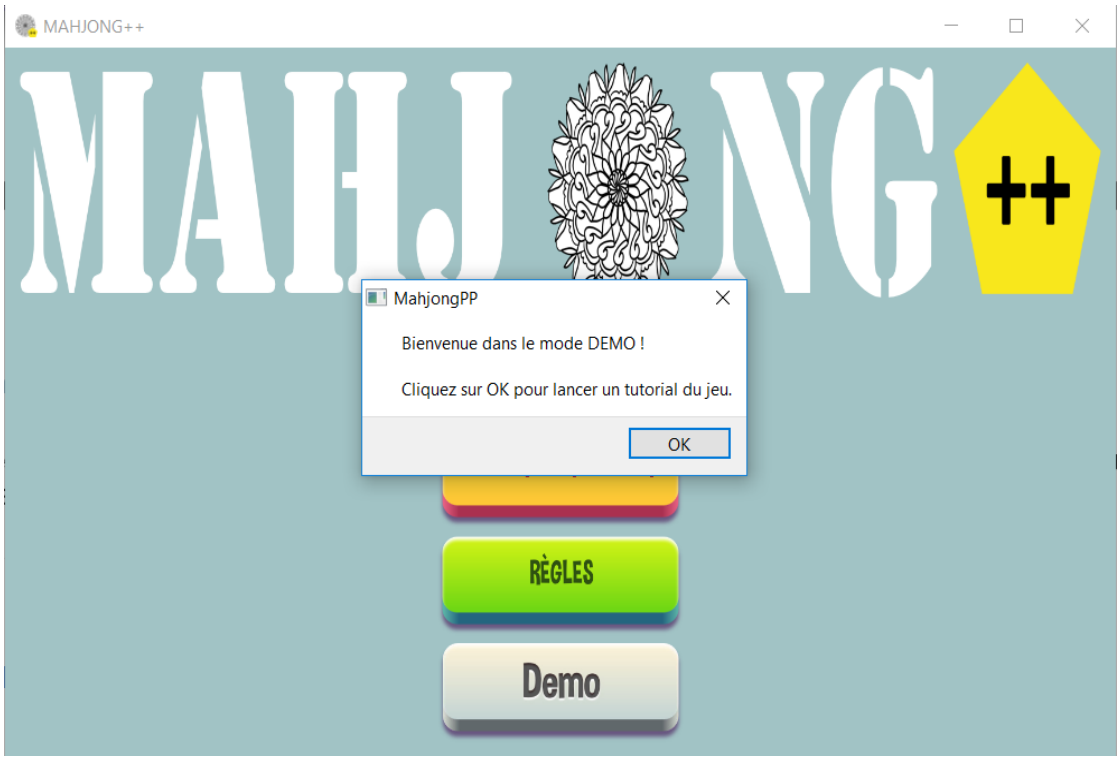
Boite de Dialogue : Pour Choisir le niveau

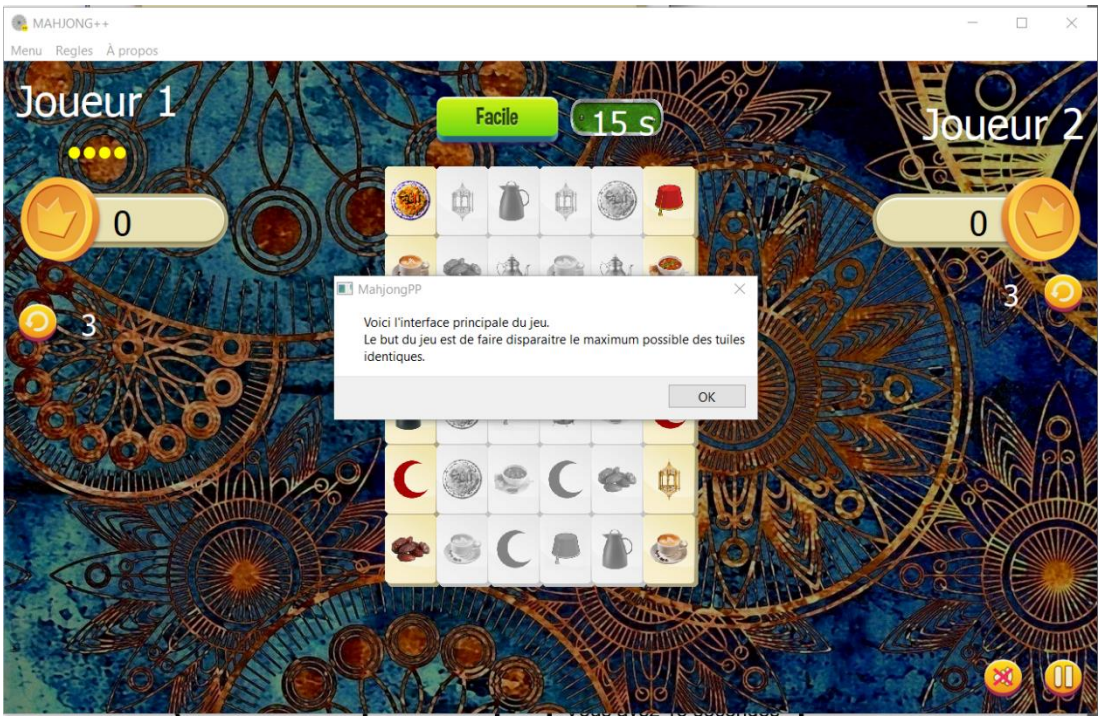




Fenêtre : des règles

Pour le mode Demo, c'est un mode qui va guider l'utilisateur pour comprendre les règles et l'interface du jeu en lui affichant à chaque fois des boites de dialogues.

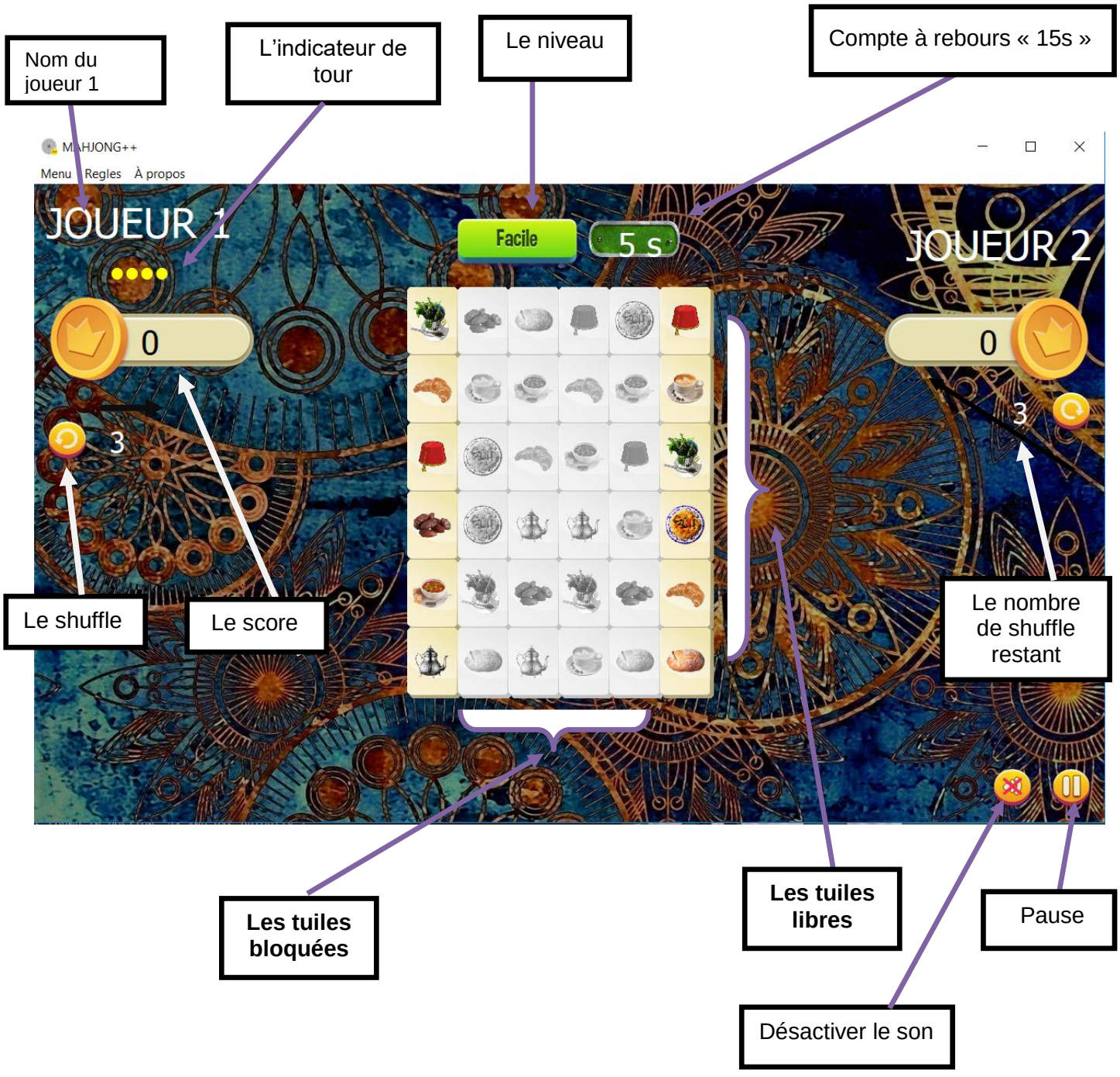




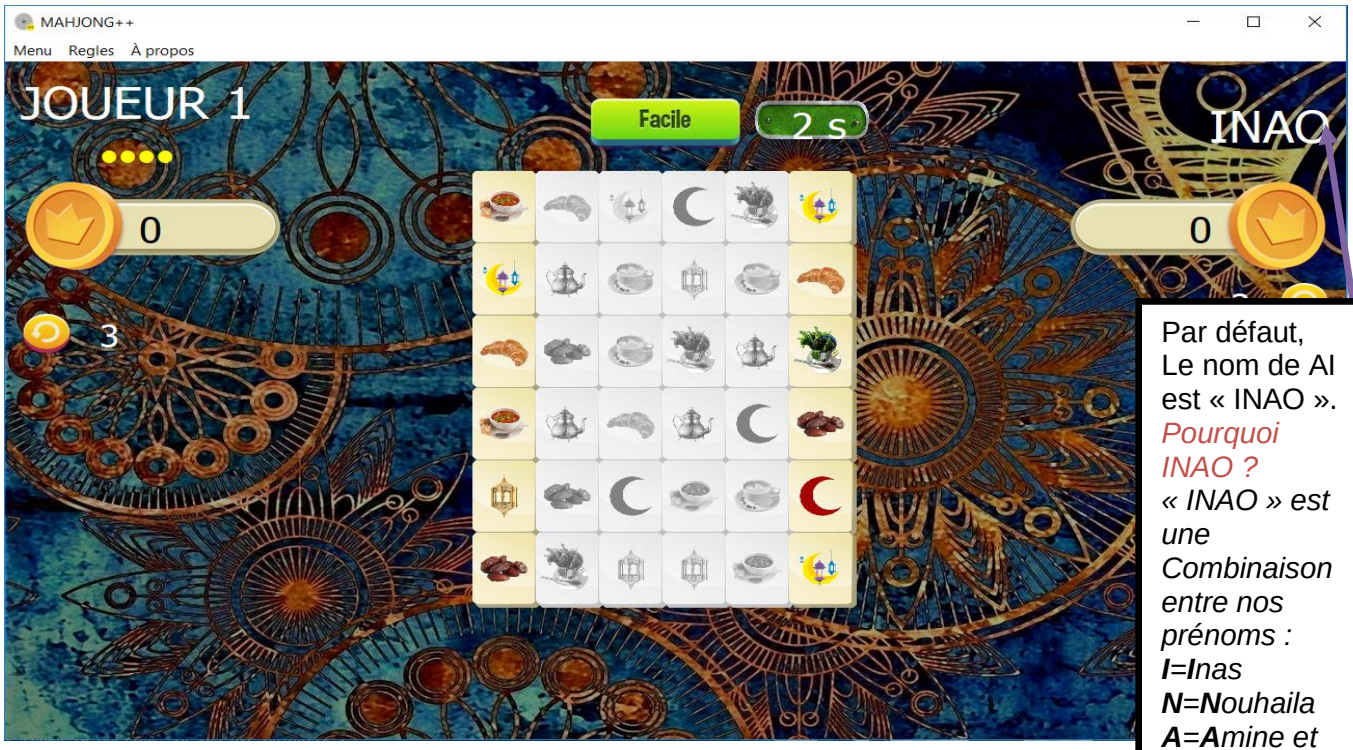
Après avoir choisi un mode de jeu et rempli les boites de dialogues, nous sommes prêts à démarrer la partie en cliquant sur le bouton "OK".

Nous avons donc la fenêtre suivante :

- ✓ **Niveau facile**
  - a. Joueur VS Joueur



b. Joueur VS Ordinateur

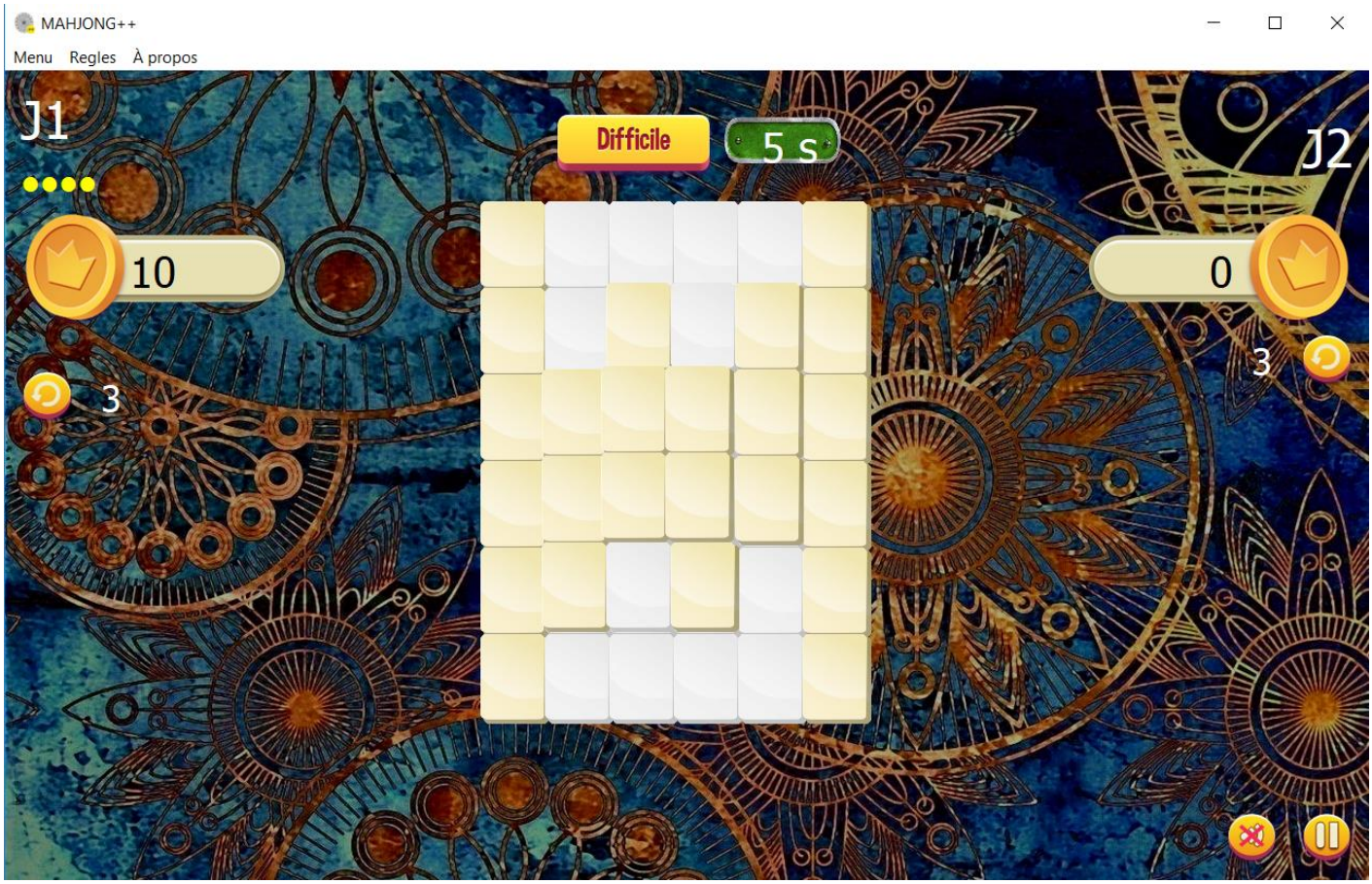


Par défaut, Le nom de AI est « INAO ».
Pourquoi INAO ?
« INAO » est une Combinaison entre nos prénoms :
***I*=*I*nas**
***N*=*N*ouhaila**
***A*=*A*mine et *A*bdellah**
***O*=*O*ussama**

✓ Niveau moyen

✓ Niveau difficile

Dans ce niveau, au début de chaque tour, les icones des familles sont visibles tant que le joueur n'a pas encore retiré un pair. Ce niveau dispose aussi de 3 étages.



Remarque :

Les deux modes de jeu, JVJ et JVO supportent les trois niveaux à savoir Facile, Moyen et Difficile.

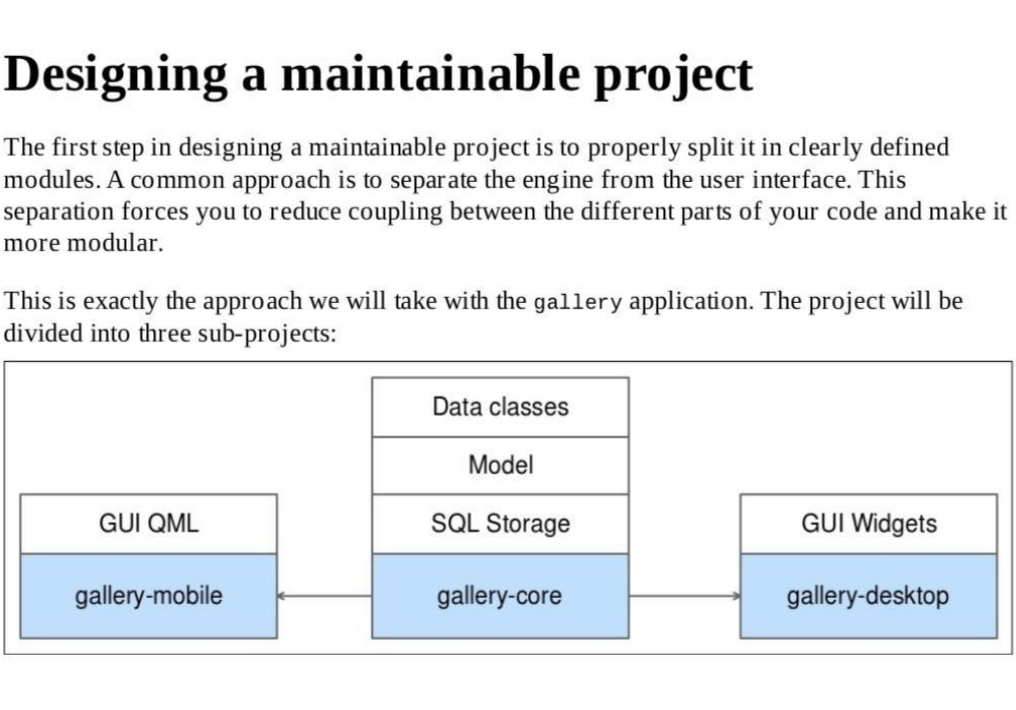
4. Guide de développeur :

4.1 Interfaces de jeu :

4.1.1 QML et Pourquoi on l'a abandonné :

Au cours d'apprentissage du monde de Qt, on a découvert un type de projets qui s'appelle QT Quick, qui permet de créer des applications légères au niveau de l'interface et l'usage des ressources. La base de Qt Quick est QML, ce dernier permet de créer les interfaces dynamiques d'une façon très simple. **MAIS le problème qu'on a trouvé c'est que le code QML ne peut pas communiquer directement avec le code C++.** Comment? QML est basé sur le langage JavaScript, les données présentées dans l'interface sont en fait des vecteurs / tableaux dans JavaScript et que le langage C++ ne peut pas les comprendre. Ainsi l'orientation QML contredit l'objectif du projet: appliquer ce qu'on a étudié dans le langage C++.

C'est pourquoi on a abandonné QML et on a décidé d'utiliser Qt Widgets. Les éléments de l'interface (comme tuile, boutons.) sont présentés par des objets dans Qt Widgets et C++ : tel que chaque objet envoie des SIGNAUX et reçoit des SLOTS. De cette façon, ils seront liés et il y aura des interactions logiques entre eux.



L'image au-dessus montre l'utilisation de QML et QT Widgets :
(**Source** : Lazar G., Penea R. - Mastering Qt 5- 2016)

4.1.2 Qt et C++ :

La bibliothèque Qt et le langage C++ :

Au début du projet, nous nous sommes retrouvés dans différents choix de bibliothèques graphiques à utiliser, parce que nous n'avons jamais appris à programmer des interfaces graphiques, et nous avons choisi Qt parce qu'il est purement basé sur le langage C ++, et nous aurons l'occasion d'appliquer ce que nous avons étudié en programmation orientée objet C++. Qt propose des composants de widgets, un accès aux données, des connexions réseau, la gestion des threads, l'analyse XML, etc. Comment utiliser les interfaces graphiques ou l'architecture? Chaque élément de l'interface est un objet en C ++. Qt est un Framework complet et multiplateforme avec lequel tous types d'applications pourront être réalisées, notamment le développement du jeu (dans différents systèmes d'exploitation).
L'environnement de développement comprend:

1. Des bibliothèques pour les objets graphiques (boutons, zones de texte, etc.), les structures de données (QList, QVector, QStack, etc.) et d'autres encore utilisés pour la compilation et le débogage.

2. QMake (un outil de gestion de compilation pour les projets Qt)

3. Qt Designer (un outil graphique de type glisser-déposer)

4. Une riche documentation intégrée.

Qt peut être utilisé pour réaliser des programmes pour des systèmes embarqués, des programmes de gestion de bases de données, des programmes utilisant le réseau et des programmes avec des graphiques 2D comme notre cas.
Puisque le projet sera réalisé par C++, on a choisi ce Framework car il supporte le C++ et pour sa richesse.

4.2 La logique de jeu :

4.2.1 Les classes et leurs méthodes importantes :

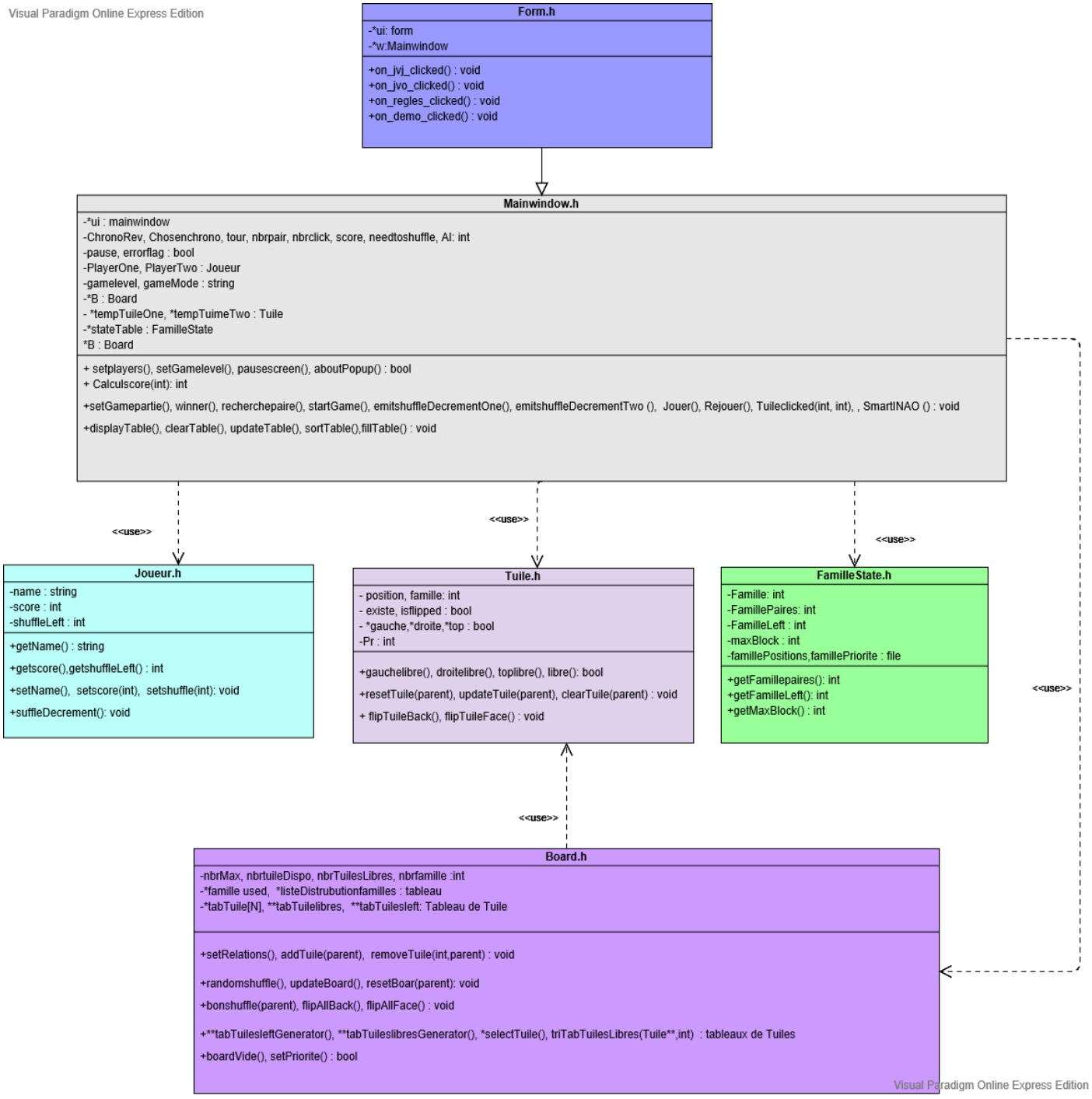


Diagramme de classe pour les classes utilisées

➢ Classe Tuile :

Définit chaque Tuile utilisée dans le jeu, elle la représente sous forme d'un objet dont les attributs et les méthodes sont toutes les informations caractéristiques de chaque Tuile, ainsi que les fonctions qui la manipulent selon le Board (tableau des Tuiles). Cette classe se compose de :

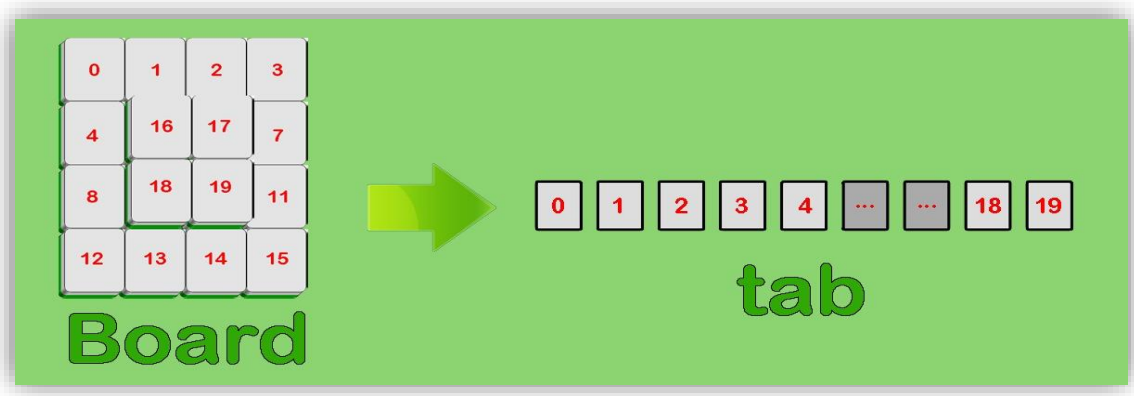
- ✓ **Position** : l'indice (un nombre) unique de la Tuile par rapport au tableau de la Classe Board (à voir après).
- ✓ **Famille** : l'icône de la Tuile (Œuf, Soupe, dattes, Total de 15 Familles) représentée par un entier compris entre 1 et 15 (Une valeur nulle indique une tuile sans icone).
- ✓ **Existe** : Egale à "Vrai" si la Tuile sélectionnée est présentée dans le Board, "Faux" sinon.
- ✓ **Gauche, droite, top** : Des pointeurs qui pointent sur la Tuile dont la direction est respective au nom du pointeur.
- ✓ **Pr :** c'est un nombre qui indique la priorité de chaque tuile dans le Board.
- ✓ **Tuile (** QWidget *parent = 0, int position = 0, int famille = 0,bool existe = 0 **)** : Constructeur de la tuile
- ✓ **~Tuile () :** Destructeur de la tuile.
- ✓ **ClearTuile ():** Supprime la tuile de l'interface et la déconnecte du signal de Clicked ().
- ✓ **libre ()** : retourne "Vrai" si la tuile est libre, "Faux" sinon.
- ✓ **UpdateTuile**() : Fonction pour mettre à jour la tuile après le shuffle.
- ✓ **resetTuile() :** Fonction pour réinitialiser la tuile (signal, existe)
- ✓ ***mousePressEvent***() : évènement de click avec (position, famille) au parent envoyé par le signal Clicked ()
- ✓ **Clicked** () : Un signal qui envoie (position, famille) au parent.

Pour le niveau Difficile :

- ✓ **isFlipped :** attribut booléen, = 1 si la famille de la tuile est cachée, = 0 sinon.
- ✓ **flipTuileBack () :** Cacher la forme de la tuile.
- ✓ **flipTuileFace () :** Montrer la forme de la tuile.

➢ Classe Board :

C'est la table qui représente le terrain du jeu. Dans notre code, elle est représentée par un tableau d'une seule dimension qui contient toutes les Tuiles existant au début de la partie. Chacune de ces tuiles est identifiée par un indice unique dans le tableau "position". Le Choix de tableau comme solution est pris pour faciliter le parcours et la recherche des paires. De plus, enlever une Tuile va tout juste marquer sa case dans le tableau comme "Vide".



La photo ci-dessus est la numération des tuiles du Board (un tableau de 1 dimension).

- ✓ **nbrMax:** Attribut de nombre initial de tuiles dans le Board.
- ✓ **nbrTuilesDispo:** Attribut de nombre de tuiles restantes dans le Board.
- ✓ **nbrFamille:** Attribut de nombre de familles utilisées dans le Board.
- ✓ **nbrTuilesLibres:** Attribut de nombre de tuiles libres dans le Board.
- ✓ ***famillesUsed:** Tableau de familles utilisées dans le Board.
- ✓ ***listDistributionfamilles :** Tableau de distribution de familles dans le Board généré aléatoirement selon le nombre initial de tuiles.
- ✓ ***tabTuiles[N] :** Tableau de tuiles.
- ✓ ****tabTuilesLeft:** Tableau de tuiles restantes dans le Board.
- ✓ ****tabTuilesLibres:** Tableau de tuiles libres dans le Board.

- ✓ **Board (**QWidget *parent = nullptr, int nbrMax = 0**) :** Constructeur de Board.
- ✓ ~***Board*** () : Destructeur de Board.

- ✓ **addTuiles(QWidget*):** Fonction de remplissage du Board.

- ✓ **setRelations() :** Établissement de liaisons entre les tuiles collées (Création du d'un graphe).
- ✓ **setPriorite ():** distributeur de priorités au tuiles.
- ✓ **UpdateBoard ():** Fonction pour mettre à jour le Board.

- ✓ **triTabTuilesLibres (Tuile**, int) :** Fonction qui trie le tableau des tuiles libres selon leurs priorités.

- ✓ **tabTuilesLeftGenerator ():** Générateur de tableau de tuiles restantes dans le Board.
- ✓
- ✓ **tabTuilesLibresGenerator ():** Générateur de tuiles libres dans le Board.

- ✓ **selectTuile ():** Fonction utilisée par l'ordinateur pour sélectionner une position d'une tuile libre aléatoirement.

- ✓ **boardVide ():** = 1 si le Board est vide, 0 sinon.

- ✓ **removeTuile (int, QWidget*):** Fonction pour supprimer une tuile**.**

- ✓ **resetBoard (QWidget*) :** Fonction pour réinitialiser le Board (cas de rejouer ())**.**

Pour niveau difficile:

- ✓ **flipAllBack() :** Cacher tous les tuiles dans le Board.
- ✓ **flipAllFace() :** Montrer tous les tuiles dans le Board.

Les fonctions du shuffle:

- ✓ **void Randomshuffle ():** Mélanger les tuiles aléatoirement (Pour le niveau facile)
- ✓ **bool Bonshuffle (QWidget*) :** Mélanger les tuiles selon une stratégie (Pour moyen et difficile. Voir plus de détails sur son implémentation dans la rubrique des Algorithmes).

*L'Algorithme de **distribution initiale des familles** suit les étapes suivantes :*

➢ Classe Joueur :

Cette Classe représente les caractéristiques de chaque Joueur à savoir:

`Name:` Le nom.
`Score:` Le score.
`ShuffleLeft:` Le nombre de shuffles restants.

- ✓ **Joueur (**QString, int = 0, int = 3**) :** Constructeur du Joueur avec shuffle = 3 par défaut**.**
- ✓ **setName** (QString) **:** Setter de nom.
- ✓ **getName**(): Getter de nom.
- ✓ **setScore** (int) **:** Setter de score.
- ✓ **getScore** () **:** Getter de score.
- ✓ **setShuffle**(int): Setter de Shuffle.
- ✓ **getShuffleLeft** () **:** Getter de Shuffle.
- ✓ **shuffleDecrement** () **:** Fonction pour décrémenter le shuffle par un signal.
- ✓ **~Joueur() :** destructeur.

Rappel :
Option shuffle : Dans les règles du jeu, chaque Joueur a le droit de redistribuer les Tuiles restantes 3 fois. Cette « 3 fois » est représentée par l'entier "shuffleLeft".

➢ Classe FamilleState :

FamilleState est une classe qui peut être assimilée à une structure de données, qui contient (famille, famillePositions, famillePriorite, famillleleft, famillePaires, maxBlock). Son rôle est d'enregistrer l'état de tuiles libres durant le jeu et qui est utilisée par l'ordinateur dans les niveaux avancés pour trouver le meilleur choix de tuiles à sélectionner en utilisant les algorithmes de **Multiple-First** et **Max-Block** (Voir la rubrique 'les algorithmes). La structure de FamilleState ressemble au figure suivant:

ID de la famille	Tableau de positions	Tableau de priorités	Nombre de tuiles dispo	Nombre de paires	Indice de Max-Block
5	[0, 36, 12]	[1, 1, 0]	3	1	2

- **FamilleState (**QObject *parent = nullptr**) :** Constructeur de la classe
- **int famille :** Chaque famille dans le Board a un indice ou ID, compris entre 1 et 15 maximal de familles.
- **int famillePaires** : Le nombre de paires restantes et libres dans le Board de la famille correspondante, généralement il est égal au nombre de tuiles dispo / 2.
- **int familleLeft :** Nombre de tuiles restantes, généralement il est égal à la taille de **famillePositions**.
- **int maxBlock :** est l'indice le plus important dans la structure, c'est lui qui aide l'ordinateur à sélectionner la meilleure tuile pour converger vers une solution optimale de Board (Board ne se bloque pas à la fin du jeu). La valeur de **Max-Block** est égale à la somme des priorités de la famille correspondante.

- **QList<int> famillePositions :** Est une liste qui prend au maximum 4 éléments, elle contient les positions de tuiles libres et restantes dans le Board. Par exemple : La famille 5 se trouve dans les positions 0, 32 et 12.
- **QList<int> famillePriorite : :** Est une liste de même taille que famillePositions, et qui contient respectivement les priorités correspondantes à chaque position. Par exemple : la tuile qui se trouve dans la position 0 est de priorité 1, et celle qui se trouve dans la position 12 est de priorité 0, etc…

Les getters:

- **Int getFamillePaires ()**
- **Int getFamilleLeft ()**
- **Int getMaxBlock ()**

Les Opérateurs:

La classe possède aussi des opérateurs de comparaison qui vont être utilisées plus tard pour l'application des algorithmes de **Max-Block** et **Multiple First** :

- **bool operator< (**const FamilleState& a, const FamilleState& b**)**
- **bool operator<= (**const FamilleState& a, const FamilleState& b**)**

➢ Classe Mainwindow :

C'est la classe la plus importante du jeu, elle utilise les classes vues précédemment pour gérer le jeu. Elle comporte de :

- **MainWindow**(QString, QWidget *parent = nullptr) : Constructeur de fenêtre principale.
- bool **errorFlag** : Indicateur de démarrage du jeu (le jeu ne sera pas lancé si errorFlag =1)
- bool **setPlayers**() : Setter des joueurs.
- bool **setGameLevel**() : Setter du niveau.
- void **setGamePartie**() : Setter de design de la partie (Interface de la partie) et construction du Board et remplissage de la table stateTable .
- bool **pauseScreen**() : Fonction pour suspendre le jeu.
- int **chronoRev** : Le chrono du jeu.
- int **chosenChrono** : Le chrono initial du jeu selon le niveau.
- QTimer ***t** : Gestionnaire du temps.
- int **tour:** Attribut de tour pour distinguer entre le tour de joueur 1 et le tour du joueur 2.
- int **nbrpair** : Attribut de nombre de paires trouvées durant un tour.
- int **nbrClick** : Attribut de nombre de cliques de tuiles durant un tour.
- int **score** : Un score temporaire pour aider la fonction qui calcule le score.
- bool **pause** = false : Flag de pause (=1 si le jeu est suspendu, 0 sinon).

- QString **gameLevel** : Attribut de niveau choisi (chaine de caractères).
- QString **gameMode** : Attribut de mode du jeu choisi (chaine de caractères).
- Joueur **playerOne** : Objet du joueur1.
- Joueur **playerTwo** : Objet du joueur2.
- int **calculScore**(int) : Fonction pour calculer le score pour le joueur courant selon le nombre de paires sélectionnées durant un tour .
- void **winner**() : Fonction appelée dès que le Board soit vide et elle affiche le joueur gagnant.
- bool **aboutPopup**() : Boite de dialogue pour afficher « A propos »
- void **recherchePaire**() : Fonction utilisée par l'ordinateur pour rechercher la tuile correspondante dans le niveau facile.
- void **fillTable**() : Fonction qui remplit la table stateTable
- void **sortTable**(); Fonction qui trie la table selon les maxBlock des familles
- void **updateTable** () : Fonction qui met à jour la table
- void **displayTable**() : Fonction qui affiche les données de la table dans le console (pour le débogage)

- void **startGame**() : Fonction pour déclencher le jeu.
- bool **on_pause_clicked**() : Fonction du bouton 'PAUSE'.
- void **emitShuffleDecrementOne**() : Fonction pour décrémenter le shuffle au joueur 1 et mélanger les tuiles.
- void **emitShuffleDecrementTwo**() : Fonction pour décrémenter le shuffle au joueur 2 et mélanger les tuiles.

- void **jouer():** Fonction qui gère la distribution des tours selon le cycle du compte à rebours.

- void **Rejouer**() : Fonction pour réinitialiser tous les données pour relancer une nouvelle partie.
- void **tuileClicked**(int, int) : Fonction pour comparer 2 tuiles.
- int **AI** : Nombre aléatoire utilisé par l'ordinateur pour simuler le temps de réflexion et de recherche.
- int **needToShuffle** : indice utilisé par l'ordinateur lorsqu'il est besoin de faire le Shuffle.
- Board***** **B** : Objet de type Board.
- FamilleState ***stateTable**;

- Tuile ***tempTuileOne** : Tuile temporaire utilisée pour enfiler la 1ere tuile sélectionnée.
- Tuile ***tempTuileTwo** : Tuile temporaire utilisée pour enfiler la 2eme tuile sélectionnée.
- **~MainWindow**() : Destructeur de Board.

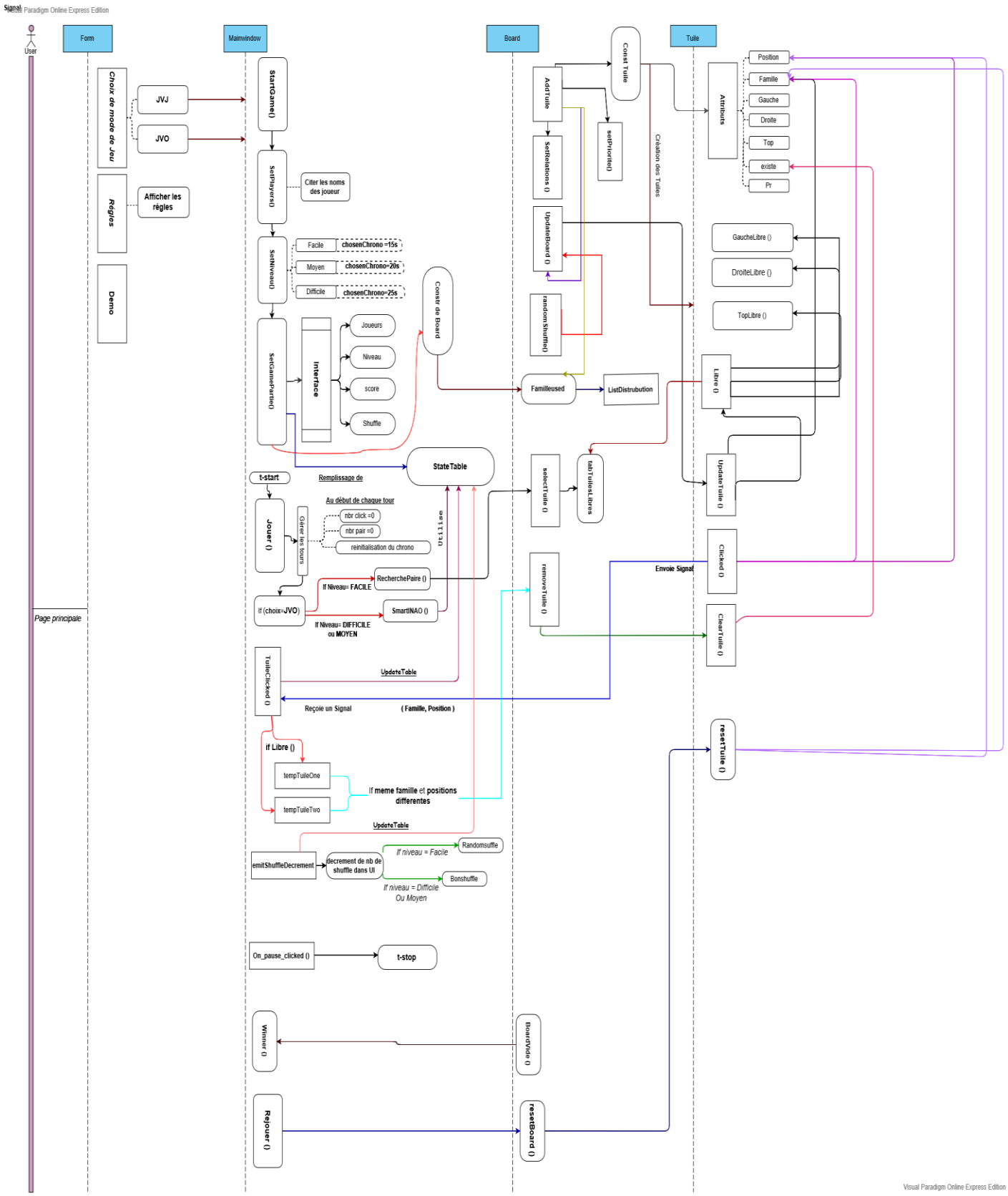
➢ Classe Form :

Cette classe gère la fenêtre d'initialisation de jeu. Elle contient :

- **Form**(QWidget *parent = nullptr);
- void **on_jvj_clicked()** : Un Button pour continuer dans le mode Joueur vs Joueur.
- void **on_jvo_clicked() :** Un Button pour continuer dans le mode Joueur vs Ordinateur.
- **MainWindow** *w**;**
- void **on_regles_clicked() :** Un Button pour découvrir les règles de jeu.
- void **on_demo_clicked() :** Un Button pour suivre une Démo de jeu
- **~Form();**

4.2.2 Les relations entre eux :

Le Schéma suivant explique les étapes de fonctionnement de jeu et les relations entre les classes :


Signal
Visual Paradigm Online Express Edition
User
Form
Mainwindow
Board
Const Tuile
Tuile
Choix de mode de Jeu
JVJ
JVO
Règles
Afficher les règles
Demo
StartGame()
SetPlayers()
Citer les noms des joueur
SetNiveau()
Facile
chosenChrono =15s
Moyen
chosenChrono=20s
Difficile
chosenChrono=25s
SetGamePartie()
Interface
Joueurs
Niveau
score
Shuffle
Const de Board
AddTuile
SetRelations ()
UpdateBoard ()
randomShuffle()
setPriorite()
Création des Tuiles
Attributs
Position
Famille
Gauche
Droite
Top
existe
Pr
GaucheLibre ()
DroiteLibre ()
TopLibre ()
Libre ()
Familleused
ListDistrubution
StateTable
t-start
Remplissage de
Jouer ()
Au début de chaque tour
nbr click =0
nbr pair =0
reinitialisation du chrono
Gérer les tours
If (choix=JVO)
If Niveau= FACILE
RecherchePaire ()
If Niveau= DIFFICILE ou MOYEN
SmartINAO ()
selectTuile ()
tabTuilesLibres
UpdateTuile ()
removeTuile ()
Clicked ()
ClearTuile ()
Envoie Signal
TuileClicked ()
UpdateTable
Reçoie un Signal
(Famille, Position)
if Libre ()
tempTuileOne
tempTuileTwo
If meme famille et positions differentes
resetTuile ()
emitShuffleDecrement
decrement de nb de shuffle dans UI
If niveau = Facile
Randomsuffle
Bonshuffle
If niveau = Difficile Ou Moyen
On_pause_clicked ()
t-stop
Winner ()
BoardVide ()
Rejouer ()
resetBoard ()
Page principale
Visual Paradigm Online Express Edition

Au démarrage de jeu, le Constructeur de Form lance la page d'accueil qui contient les choix des modes et les règles, ainsi qu'une Démo pour le jeu. Après, Un objet Mainwindow va être crée, son constructeur va appeler la méthode **StartGame ()** pour faire appel à **SetPlayers ()** et **setGamelevel** qui vont lancer des boites de dialogues pour setter les noms et le choix de niveau, Chaque niveau a son chronomètre (par exemple pour facile c'est 15 secondes). Cet objet a un booléen *errorFlag* qui retourne false si les boites de dialogues ont été bien remplies et lance la page de Jeu, sinon s'il est true le jeu reste toujours dans la page d'accueil. Par suite, la méthode **SetGamePartie ()** va être appelée pour créer l'Interface de jeu après avoir stocker les informations nécessaires à partir des boites, à savoir les noms des joueurs, le niveau, le chrono, le score et le shuffle. Puis, le constructeur de Board va être appelé par la méthode **SetGame partie** () pour créer l'objet **Board B**, la méthode va créer aussi un objet **stateTable** de type **FamilleState**.

La construction d'objet Board prend comme argument le nbrMax de tuile à utiliser, c'est la taille de Tableau tabTuiles qui représente tout le Board. **La création du Board** passe par les étapes suivantes:

1. Calcul de nombre de familles à utiliser (Généralement il est égal au nombre total de tuiles / 4)

2. Création de tableau de familles à utiliser tel que chaque famille est représentée par un indice compris entre 1 et 15. Les indices sont générés d'une façon aléatoire tel que chaque indice doit apparaitre une seule fois dans le tableau.

3. Création de tableau de distribution de familles, qui a la meme dimension que tableau de tuiles totales et 4 fois la taille de tableau de familles pour que chaque tuile prendra une famille selon son indice.

Le nombre de tuiles dépend du niveau choisi (36 pour le niveau facile et 56 pour les 2 autres niveaux). Quand les tuiles sont bien ajoutés la méthode **setRelations ()** va initialiser les pointeurs de chaque tuile: gauche, droite et top dans le tableau, pour lier les tuiles entre eux selon leurs dispositions. Par exemple: les tuiles dans l'extrémité possède des tuiles dans l'une de ses côtés. Les **pointeurs gauches**, **droite** et **top** sont de type booléens qui pointe sur la valeur de l'attribut '**existe**' de la tuile à côté.

Le Graphe :

Modélisations du jeu à l'aide d'un graphe pondéré :

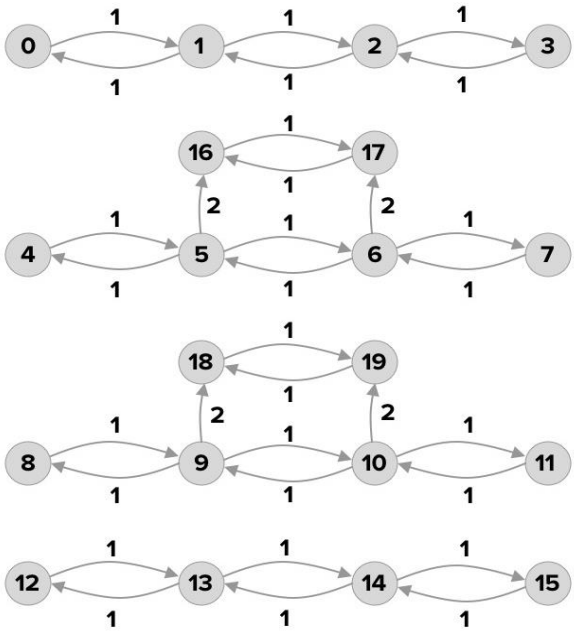
Les sommets : Attribut existe de chaque Tuile

Les arrêts :

 : Pointeur de droite.

 : Pointeur de gauche.

 **:** Pointeur de haut.



Remarque :

Une Tuile est libre si seulement si la somme des poids d'arêtes qui la relient avec ses successeurs est inferieur strictement à 2.

Ensuite le rôle de **setPriorité ()** vient après pour définir les priorités des Tuiles.

SetPriorite () :

Rappel : Priorité est un indice qui concerne seulement les tuiles libres et il est utilisé par l'ordinateur pour faire disparaitre les tuiles qui possèdent les priorités les plus élevés en premier. Par définition, haute priorité implique la libération d'un nombre maximum de tuiles. L'attribut de priorité est initialisé par défaut avec la valeur -1 pour toute les tuiles, 0 pour les tuiles libres et elle sera incrémentée selon le cas et selon l'étage, tel que les tuiles qui se trouvent au top auront toujours une valeur de priorité = 2.

La méthode **setPriorite ()** est appelée après la distribution de familles et établissement de relations. L'algorithme passe par les étapes suivantes:

1. On vérifie si la tuile à côté est libre, si c'est le cas on décrémente la valeur de priorité par 1, sinon elle ne change pas.

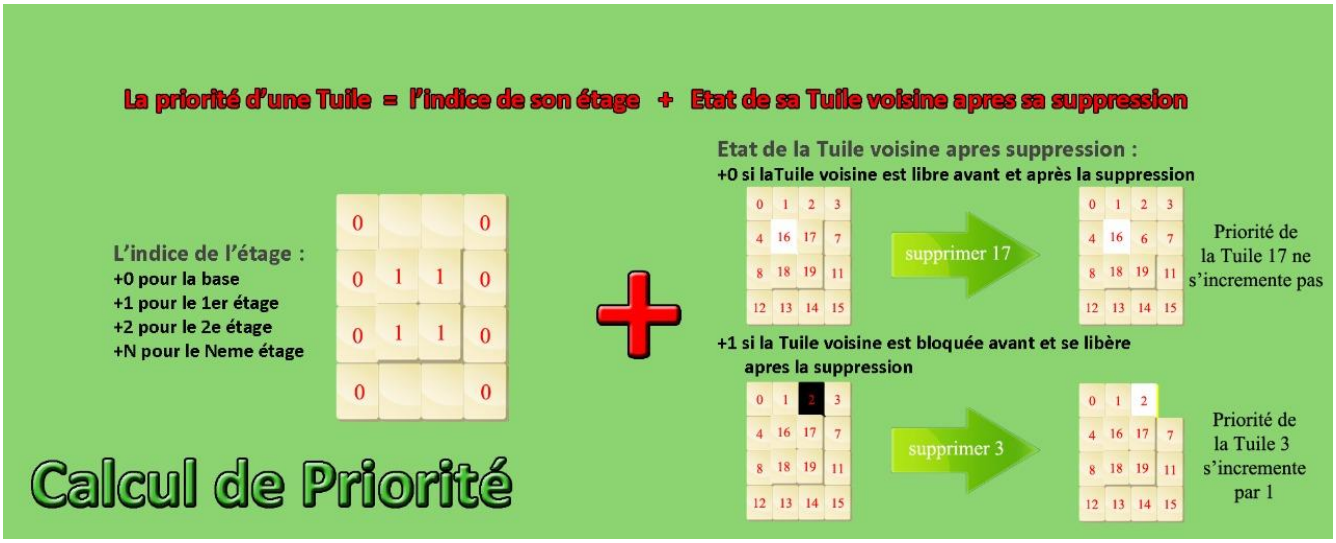
2. On simule la suppression de la tuile courante en affectant la valeur 0 ou 'false' à son attribut 'existe'.

3. On teste la tuile à côté si elle a été libérée.

4. La valeur de priorité dépend de la valeur retournée par la fonction libre () de la tuile à côté, tel que on incrémente la valeur de priorité par 1 si la tuile à côté a été libérée, sinon elle ne se change pas.

5. On applique l'algorithme sur tous les tuiles libres dans le Board.

Le schéma suivant explique le calcul de setPriorite () :



Le Board contient aussi les méthodes **Randomshuffle ()** et **Bonshuffle (),** ces deux méthodes vont être rappelées lorsque l'utilisateur clique sur l'icône de shuffle dans l'interface avec **Randomshuffle ()** et il est utilisé dans le niveau facile et **Bonshuffle ()** pour les niveaux difficile et moyen (en va voir comment ils marchent dans la rubrique des algorithmes).

Pour **stateTable** c'est une structure qui contient (*famille*, *familllePositions*, *familllePriorite*, *famillleleft*, *familllePaires*, *maxBlock*), utilisée par Mainwindow pour générer un tableau qui contient toutes les informations sur les familles existantes.

Après avoir, crée le Board et la remplie avec des Tuiles, maintenant l'espace est prêt, Le Chrono va déclencher la méthode **Jouer ()** qui va gérer les tours et le chrono ; *à chaque début de tour* il met *nbrClick=0*, *nbrpair=0* et *chrono à sa valeur initiale* selon le niveau choisi. Quand l'utilisateur cliquez sur une tuile un signal de click va donner la famille et la position au tuile **Clicked ()** ce slot faire prend comme argument les informations nécessaires pour les stocker dans tuile temporaire (**tempTuileOne** et **tempTuileTwo**) en même temps incrément le nombre de click, si

elle est libre et le nbrClick=1, les valeurs de famille et position seront stockés dans **tempTuileOne**. Quand l'utilisateur clique sur une 2e Tuile le même processus va être exécuté et ensuite s'il est libre elle va être stocker dans **tempTuileTwo** et puis une comparaison va être effectuée sur les deux Tuiles, s'ils ont la même famille et des positions différentes **removetuile ()** va être rappelée et puis **clearTuile ()** ils vont faire dispparaitre les deux Tuiles (existe = 0). Sinon, La position de tempTuileTwo et sa famille vont être passés à tempTuileOne, pour que le prochain click considère la dernière Tuile Cliqué comme la première dont il va chercher sa paire. Après chaque changement dans le Board la méthode **UpdateBoard ()** sera rappelée avec **UpdateTuile ()** pour faire des changements nécessaires pour chaque tuile. Le score est calculé selon l'indice de tours, pour attribuer le score au joueur concerné.

4.2.3 Bonshuffle

Remarque importante :

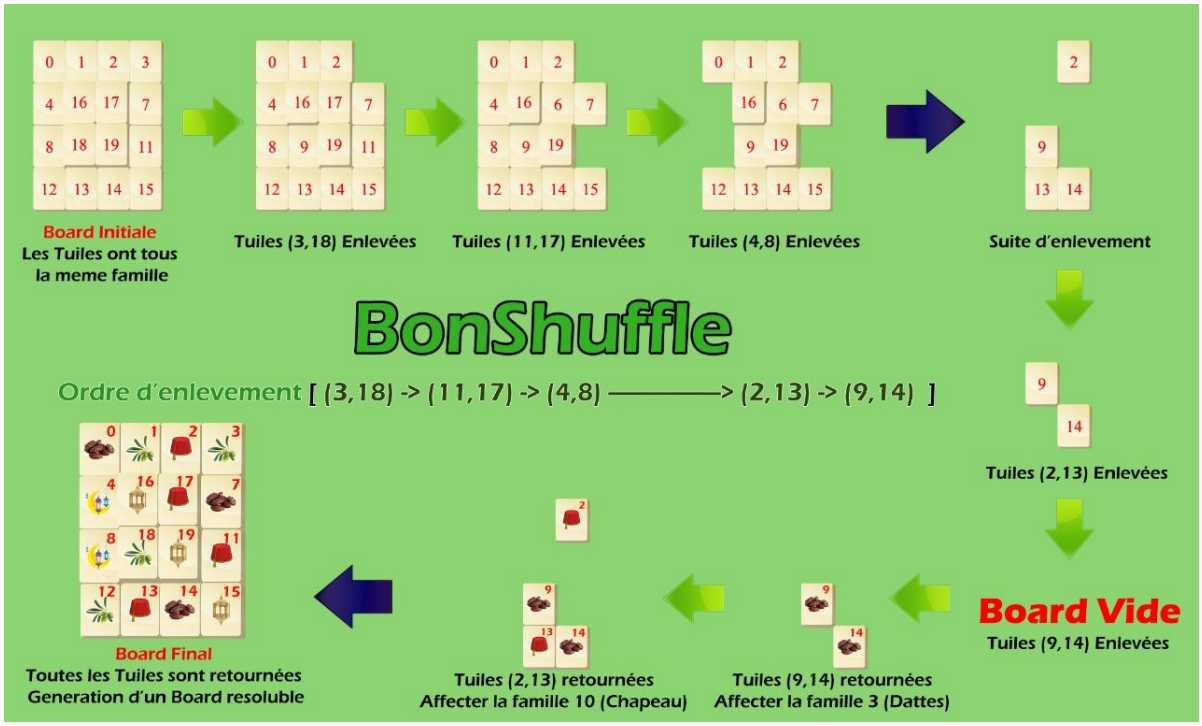
Au cours du jeu on peut parfois se bloquer face à des situations impossibles à résoudre.

Par exemple : lorsqu'elles ne restent dans notre Board que deux Tuiles mais placées l'une sur l'autre. La Tuile d'au-dessous n'est pas libre d'où on ne peut jamais l'enlever. Et on peut rapidement remarquer que même le Shuffle est inutile à ce point, en bref cette situation n'a pas solution.

Normalement, un bon joueur du Mahjong suit une stratégie de jeu pour ne pas tomber dans ce piège. Pourtant, la nature du jeu ne peut garantir qu'il ne va jamais se bloquer un jour.

Donc pour diminuer les chances de rencontrer ce problème, on a optimisé la distribution aléatoire en une meilleure fonction que l'on a appelé le **Bonshuffle**.

- Bool **Bonshuffle(QWidget*)** : Mélange les tuiles aléatoirement d'une façon optimale à fournir un Board résoluble. Son algorithme est le suivant :



Remarque :

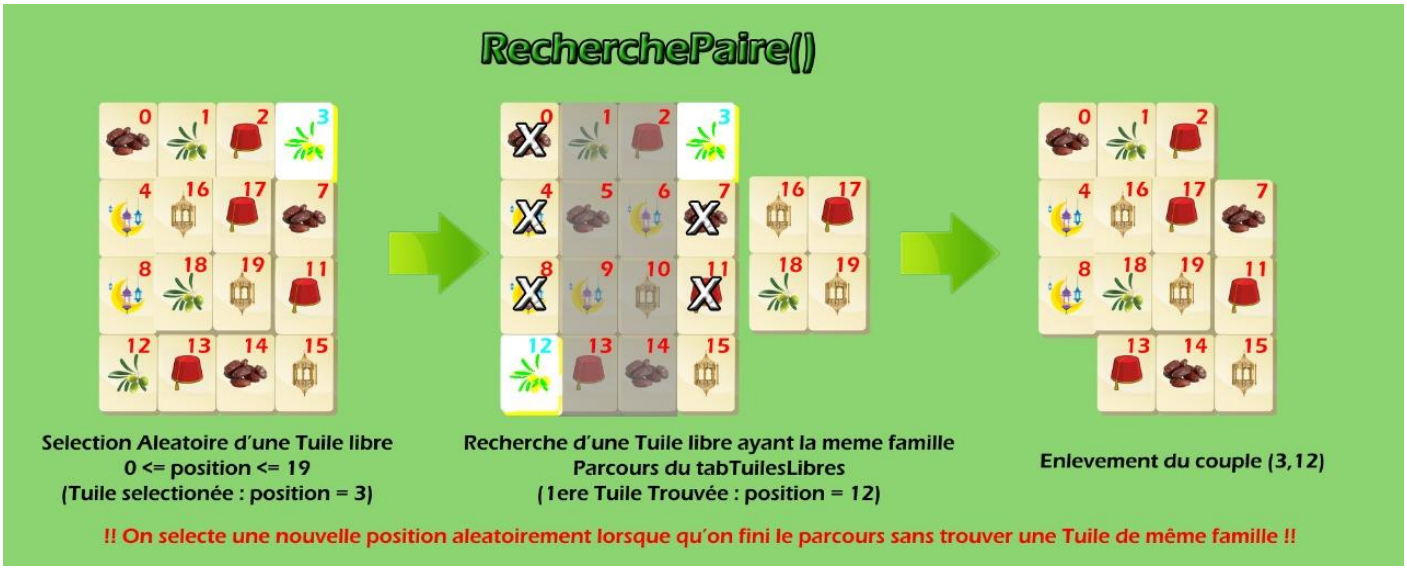
Même cet algorithme est juste une heuristique qui diminue les chances de block, son grand avantage est que Board qu'il génère contient sûrement au moins une solution, et c'est au joueur de la trouver ou pas.

4.3 L'Ordinateur (INAO)

4.3.1 Les algorithmes

a. RecherchePaire () :

Cet algorithme représente la méthode la plus classique qu'un joueur débutant utilise intuitivement pour jouer. D'abord son Œil tombe aléatoirement sur une tuile quelconque puis il commence par chercher dans tout le Board jusqu'à ce qu'il tombe la première fois sur sa sœur.

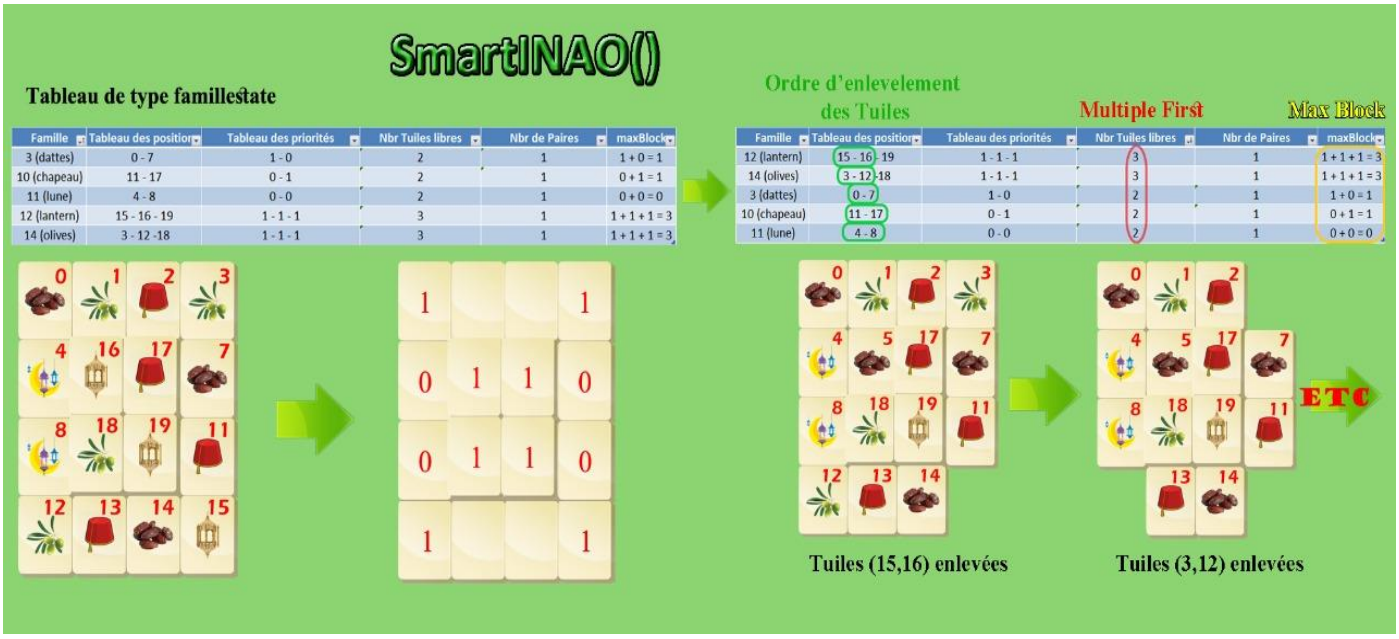


b. SmartINAO () :

Cette méthode est la fonction qui met en œuvre l'utilité de la classe « FamilleState », il s'occupe de trier le tableau de cette classe selon la priorité de chaque Tuile. Ensuite l'enlèvement des Tuiles va suivre l'ordre décroissant de la priorité de chaque paire de Tuiles. Notre ordinateur INAO est capable d'effectuer ce tri grâce à deux algorithmes :

- **Multiple-First Heuristique** : priorise les Tuiles dont la famille apparaisse le maximum de fois. ça correspond à la longueur du tableau de positions dans la classe FamilleState.
- ***Max-Block Heuristique*** : priorise la Tuile qui libère lors de sa suppression le maximum de Tuiles possibles. Ça correspond à l'indice « MaxBlock » dans la classe FamilleState, qui est la somme des priorités des tuiles de même famille (Voir le calcul des priorités dans la méthodes setpriorite () de la classe Board page 24)

Les informations fournis par ces deux algorithmes sont la clé pour la stratégie que l'ordinateur a l'intention de suivre. Dans la suite la fonction SmartINAO va s'occuper du reste, comme dans l'exemple suivant :



5. Les Difficultés qu’on a trouvées

Plusieurs difficultés ont été rencontrées au cours de ce projet, nous les décrirons dans cette partie en mettant en avant les solutions qui ont pu être mises en place :

Le premier problème rencontré était le retard. Nous avons avancé sans être certains d’appliquer la bonne méthode ; l'abandonnement de QML par exemple occasionna un gros retard qui ne nous permis de démarrer réellement le projet qu’au mois de mars. Pour faire face à ce problème, nous avons dû prioriser certaines fonctions et en mettre d’autres de côté afin d’obtenir un produit fonctionnel et satisfaisant.

Le second problème du projet, qui n’en est pas réellement un, concerne la technique. En travaillant sur des technologies que nous ne maîtrisions pas, nous avons donc passé du temps à nous former et à lire des livres de références, suivre plusieurs tutoriels.
Nous avons eu également beaucoup de mal à implémenter le joueur ordinateur, En effet, adapter les algorithmes vus en cours à un cas concret n’est pas si évidant.

6. Conclusion :

Pour notre premier projet professionnel, programmer ce jeu était un grand challenge. C'était bel et bien la première fois qu'on donne une forme au savoir qu'on a acquis jusqu'à ce point. Travailler en équipe avec un langage non maitrisé sur une plateforme dont on ignorait l'existence just qu'il y a quelques semaines, c'est ce qu'on appelle une aventure. On était obligé d'étudier son langage à zéro en parallèle avec nos études, tout en pensant aux solutions, aux algorithmes, aux stratégies, jour et nuit. Et franchement le fait que ce projet soit un "Jeu" était un sacré facteur qui nous a motivé à aller si loin. S'elle y a une leçon qu'on apprise de ce module, c'est qu'en aimant notre travail on peut créer des miracles !

Merci pour cette expérience.