# niwa_rapport_ED2b

- Cahier des charges minimal :

  Pour ce projet, nous avons implémenter un plateau de jeu qui permet à deux joueurs de faire une partie de Niwa. En particulier, nous permettons aux joueurs de :

  - Créer un plateau de forme arbitraire en connectant des tuiles/hexagones.
  - Bouger les pions et donner des perles selon les règles du jeu.
  - Arrêter le jeu quand un des deux joueurs a gagné.

  Tout en facilitant l’expèrience utilisateur avec une interface graphique réalisée à l'aide de la librairie Swing.

- Généralisations :
  - Éditer le plateau en donnant la possibilité de changer de taille
  - Créer des joueurs IA
  - Sauvegarder le jeu

  ▼ Possibilité de changer plusieurs paramètres du jeu

    - Choisir le nombre de joueurs (2 ou 4)
    - Choisir le nombre de pion par joueur (3 à 6)
    - Choisir le nombre de perles maximale qu’un pion peut porter (3 à 6)
    - Choisir le nombre de perles de départ par pion (2 à 4)
    - Choisir une création manuelle ou automatique du plateau
    - Choisir le nombre de tuiles lors de la création d’un plateau par l’utilisateur
    - Choisir le nombre de bots

# 1. Méthodologie: la structure orienté objet et le schéma UML

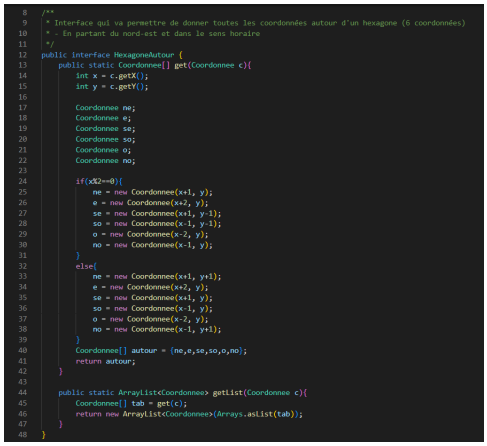
▼ Le schéma UML ( dernière page )

- Afin de créer ce schéma UML, nous avons réunit toutes les classes constituant notre projet. C’est à dire que nous les avons représenter par leurs attributs et les fonctions qu’elles contiennent. Vous pourrez y trouver toutes les liaisons, tel que les classes étendant une autre ou quels sont les classes implémentant une interface, sous forme de flèches. Vous trouverez dans le coin du document ci-dessus, une légende afin de détailler certaines notations (vous trouverez le schéma dans le fichier contenant l’entièreté du projet).

▼ **Les éléments orientés objet que nous avons vus en cours** sont autant d’éléments conceptuels qui ont enrichi notre style et nous permit d’écrire des programmes plus clairs. Voici quelques exemples :

▼ Interfaces

### ▼ Interface “HexagoneAutour”

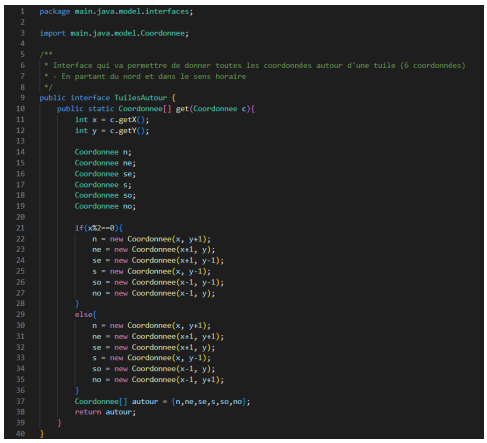


```java
/**
 * Interface qui va permettre de donner toutes les coordonnées autour d'un hexagone (6 coordonnées)
 * - En partant du nord-est et dans le sens horaire
 */
public interface HexagoneAutour {
    public static Coordonnee[] get(Coordonnee c){
        int x = c.getX();
        int y = c.getY();

        Coordonnee ne;
        Coordonnee e;
        Coordonnee se;
        Coordonnee so;
        Coordonnee o;
        Coordonnee no;

        if(x%2==0){
            ne = new Coordonnee(x+1, y);
            e = new Coordonnee(x+2, y);
            se = new Coordonnee(x+1, y-1);
            so = new Coordonnee(x-1, y-1);
            o = new Coordonnee(x-2, y);
            no = new Coordonnee(x-1, y);
        }
        else{
            ne = new Coordonnee(x+1, y+1);
            e = new Coordonnee(x+2, y);
            se = new Coordonnee(x+1, y);
            so = new Coordonnee(x-1, y);
            o = new Coordonnee(x-2, y);
            no = new Coordonnee(x-1, y+1);
        }
        Coordonnee[] autour = {ne,e,se,so,o,no};
        return autour;
    }

    public static ArrayList<Coordonnee> getList(Coordonnee c){
        Coordonnee[] tab = get(c);
        return new ArrayList<Coordonnee>(Arrays.asList(tab));
    }
}
```

Cette interface permet de récuperer la liste de coordonnées des hexagones autour d’un hexagone.

Utilisée dans plusieurs cas, exemple d’un cas basique : nous avons un pion sur un hexagone, nous voulons savoir s’il peut se déplacer sur les hexagones autour de lui. Pour récupérer les coordonnées, on se sert de l’interface.

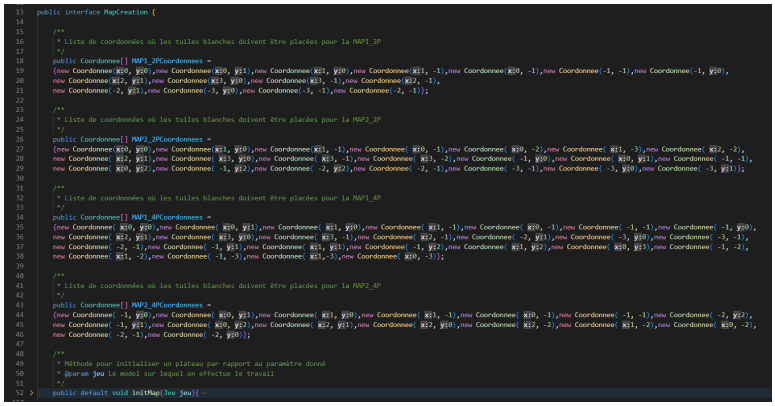
### ▼ Interface “TuileAutour”



```java
package main.java.model.interfaces;

import main.java.model.Coordonnee;

/**
 * Interface qui va permettre de donner toutes les coordonnées autour d'une tuile (6 coordonnées)
 * - En partant du nord et dans le sens horaire
 */
public interface TuilesAutour {
    public static Coordonnee[] get(Coordonnee c){
        int x = c.getX();
        int y = c.getY();

        Coordonnee n;
        Coordonnee ne;
        Coordonnee se;
        Coordonnee s;
        Coordonnee so;
        Coordonnee no;

        if(x%2==0){
            n = new Coordonnee(x, y+1);
            ne = new Coordonnee(x+1, y);
            se = new Coordonnee(x+1, y-1);
            s = new Coordonnee(x, y-1);
            so = new Coordonnee(x-1, y-1);
            no = new Coordonnee(x-1, y);
        }
        else{
            n = new Coordonnee(x, y+1);
            ne = new Coordonnee(x+1, y+1);
            se = new Coordonnee(x+1, y);
            s = new Coordonnee(x, y-1);
            so = new Coordonnee(x-1, y);
            no = new Coordonnee(x-1, y+1);
        }
        Coordonnee[] autour = {n,ne,se,s,so,no};
        return autour;
    }
}
```

Même fonctionnement que l'interface "HexagoneAutour", mais cette fois-ci pour les tuiles.

## ▼ Interface "MapCreation"



```java
public interface MapCreation {

    /**
     * Liste de coordonnées où les tuiles blanches doivent être placées pour la MAP1_2P
     */
    public Coordonnee[] MAP1_2PCoordonnees =
    {new Coordonnee(x20, y20),new Coordonnee(x20, y21),new Coordonnee(x21, y20),new Coordonnee(x21, -1),new Coordonnee(x20, -1),new Coordonnee(-1, -1),new Coordonnee(-1, y20),
    new Coordonnee(x22, y21),new Coordonnee(x21, y20),new Coordonnee(x21, -1),new Coordonnee(x22, -1),
    new Coordonnee(-2, y21),new Coordonnee(-3, y20),new Coordonnee(-3, -1),new Coordonnee(-2, -1)};

    /**
     * Liste de coordonnées où les tuiles blanches doivent être placées pour la MAP2_2P
     */
    public Coordonnee[] MAP2_2PCoordonnees =
    {new Coordonnee(x20, y20),new Coordonnee(x21, y20),new Coordonnee(x21, -1),new Coordonnee( x20, -1),new Coordonnee( x20, -2),new Coordonnee( x21, -3),new Coordonnee( x22, -2),
    new Coordonnee( x22, y21),new Coordonnee( x21, y20),new Coordonnee( x21, -1),new Coordonnee( x21, -2),new Coordonnee( -1, y20),new Coordonnee( x20, y21),new Coordonnee( -1, -1),
    new Coordonnee( x20, y22),new Coordonnee( -1, y22),new Coordonnee( -2, y22),new Coordonnee( -2, -1),new Coordonnee( -3, -1),new Coordonnee( -3, y20),new Coordonnee( -3, y21)};

    /**
     * Liste de coordonnées où les tuiles blanches doivent être placées pour la MAP1_4P
     */
    public Coordonnee[] MAP1_4PCoordonnees =
    {new Coordonnee( x20, y20),new Coordonnee( x20, y21),new Coordonnee( x21, y20),new Coordonnee( x21, -1),new Coordonnee( x20, -1),new Coordonnee( -1, -1),new Coordonnee( -1, y20),
    new Coordonnee( x22, y21),new Coordonnee( x21, y20),new Coordonnee( x21, -1),new Coordonnee( x22, -1),new Coordonnee( -2, y21),new Coordonnee( -3, y20),new Coordonnee( -3, -1),
    new Coordonnee( -2, -1),new Coordonnee( -1, y21),new Coordonnee( x21, y21),new Coordonnee( -1, y22),new Coordonnee( x21, y22),new Coordonnee( x20, y23),new Coordonnee( -1, -2),
    new Coordonnee( x21, -2),new Coordonnee( -1, -3),new Coordonnee( x21,-3),new Coordonnee( x20, -3)};

    /**
     * Liste de coordonnées où les tuiles blanches doivent être placées pour la MAP2_4P
     */
    public Coordonnee[] MAP2_4PCoordonnees =
    {new Coordonnee( -1, y20),new Coordonnee( x20, y21),new Coordonnee( x21, y20),new Coordonnee( x21, -1),new Coordonnee( x20, -1),new Coordonnee( -1, -1),new Coordonnee( -2, y22),
    new Coordonnee( -1, y21),new Coordonnee( x20, y22),new Coordonnee( x22, y21),new Coordonnee( x22, y20),new Coordonnee( x22, -2),new Coordonnee( x21, -2),new Coordonnee( x20, -2),
    new Coordonnee( -2, -1),new Coordonnee( -2, y20)};

    /**
     * Méthode pour initialiser un plateau par rapport au paramètre donné
     * @param jeu Le model sur lequel on effectue le travail
     */
    public default void initMap(Jeu jeu){
```

Cette interface, en lui donnant un objet "Jeu", pourra initialiser son plateau en fonction de la demande de l'utilisateur. Par exemple s'il veut créer son propre plateau, l'interface placera seulement le premier temple de départ. S'il veut une map préfabriquée, l'interface placera toutes les tuiles qu'il faut pour cela.

Pour les créations automatiques, il y a 2 maps pour le mode 2 joueurs et mode 4 joueurs :

- MAP1_2P : 1ère map du mode 2 joueurs
- MAP2_2P : 2ème map du mode 2 joueurs
- MAP1_4P : 1ère map du mode 4 joueurs

- MAP2_4P : 2ème map du mode 4 joueurs

## ▼ Interface “DeplacementPion”

```java
package main.java.model.interfaces;

import java.util.ArrayList;

import main.java.model.Coordonnee;
import main.java.model.Pion;

/**
 * Interface qui renvoie une liste de tous les choix possibles de déplacement qu'un pion peut avoir (par rapport à sa position + portes des hexagones)
 */
public interface DeplacementPion {
    public ArrayList<Coordonnee> canMoveLocations(Pion p);
}
```

Interface contenant une unique fonction, qui doit renvoyer une liste de tous les choix possibles de déplacement qu'un pion peut avoir (par rapport à sa position + portes des hexagones).

## ▼ Interface “ColorsSwitcher”

```java
package main.java.model.interfaces;

import java.awt.Color;

import main.java.model.Couleurs;

public interface ColorsSwitcher {
    public static Couleurs toCouleurs(Color c){
        if(c == Color.RED){return Couleurs.ROUGE;}
        if(c == Color.ORANGE){return Couleurs.ORANGE;}
        if(c == Color.GREEN){return Couleurs.VERT;}
        else{return null;}
    }

    public static Color toColor(Couleurs c){
        if(c == Couleurs.ROUGE){return Color.RED;}
        if(c == Couleurs.ORANGE){return Color.ORANGE;}
        if(c == Couleurs.VERT){return Color.GREEN;}
        else{return null;}
    }
}
```

Dans certaines situations, nous devons dessiner graphiquement des formes en fonction de la couleur des portes / perles. Il faut donc pouvoir passer d’un objet Couleurs → Color et vice-versa.

▼ Enumérations

## ▼ Enumeration “JeuEtat”

```java
package main.java.model;

public enum JeuEtat {
    CHOOSING_TUILE_LOCATION,
    ROTATING_TUILE,

    PLACING_START_PION,

    CHOOSING_PION,
    PLACING_PION,
    CHOOSING_PEARL_DESTINATION,

    CONTINUE,

    GAME_OVER,

    CHANGING_VIEW,

    GAME_INTERRUPT;
}
```

Cette enumération permet le bon fonctionnement de la boucle qui gère le déroulement du jeu. Elle contient chaque “Etat” dans lequel le jeu peut être. Exemple : savoir si c’est le moment de placer une tuile ? Le moment de tourner une tuile ? Le moment de placer un pion ? etc…

## ▼ Enumeration “MapEtat”

```java
package main.java.model;

public enum MapEtat {

    MANUEL,

    MAP1_2P,
    MAP2_2P,

    MAP1_4P,
    MAP2_4P,

}
```

Cette enumération contient simplement la manière dont le plateau doit être créer. Il y deux catégorie :

- MANUEL : Dans ce cas, le plateau devra être crée à la main par l’utilisateur
- Le reste (MAP1_2P, etc…) : Dans ces cas, le plateau sera initialisé automatiquement, en fonction de la map choisie.

## ▼ Enumeration “Couleurs”

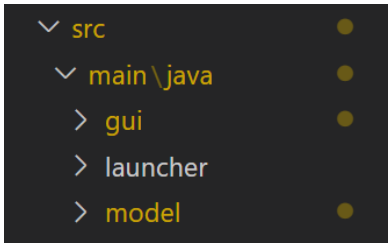
src > main > java > model > Couleurs.java > Couleurs

```java
package main.java.model;

public enum Couleurs {ROUGE, VERT, ORANGE}
```

Cette enumération permet simplement les 3 couleurs que le jeu a besoin pour :

- Les perles des pions
- Les portes des tuiles

▼ Design Pattern




Nous avons une séparation du modèle et la vue (accessoirement le launcher également). Cela nous permet d'avoir :

- Meilleure organisation
- Meilleur lisibilité

▼ Expression lambda

💡 **Les problèmes rencontrés**

- La conception du plateau : il fallait savoir comment implémenter un plateau proprement. Surtout que dans le cas de Niwa, on aurait pu l'implémenter de plusieurs façons puisque les tuiles sont formées d'hexagones. Il fallait donc faire un choix et bien l'appliquer.
- Toujours au niveau du plateau, mais cette fois-ci graphiquement : Une fois le model fini, il faut pouvoir afficher le plateau graphiquement. Sauf que comme vu précédemment, les tuiles sont formées d'hexagones. Il fallait donc trouver une solution pour dessiner proprement des tuiles à l'écran.

▼ Déroulement de l'implémentation

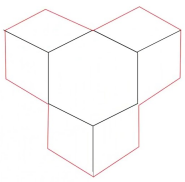
- Avant de commencer, il est important de prendre le temps de comprendre le sujet et les consignes qui sont demandées. Notre première grande étape a donc consisté à faire des choix quant à la façon dont nous souhaitions implémenter le jeu.
- Ensuite, nous avons commencé à implémenter les premières bases du jeu en respectant ses règles et contraintes. Notre objectif n'était pas de créer un squelette parfait et finalisé, mais plutôt de produire un squelette contenant les éléments essentiels du jeu. Cela nous a permis de répartir le travail sur la partie graphique, qui est étroitement liée à ce squelette.
- Ainsi, nous avons travaillé simultanément sur deux parties distinctes : le modèle et le graphique. Tandis que le modèle se peaufinait et s'enrichissait de fonctionnalités supplémentaires, les premières étapes de la partie graphique étaient mises en place.
- Il va de soi que la meilleure façon de tester un modèle est de le tester graphiquement en tant que joueur. Plus nous testions notre jeu avec l'interface graphique, plus les problèmes liés au modèle apparaissaient. La partie graphique est ainsi devenue un outil essentiel pour vérifier le bon fonctionnement de notre modèle.
- Une fois le modèle entièrement fixé et sans bugs, notre mission consistait simplement à peaufiner la partie graphique afin d'obtenir un rendu visuel plus soigné.

# 2. Dual-graph et systèmes de coordonnées : personnaliser la plateau et déplacer les pions

Voici un lien Youtube, expliquant le fonctionnement du plateau :

Niwa - Fonctionnement du plateau

.



https://youtu.be/y_tiLyty6oA

# 3. Ergonomies: l’interface Graphique

## Mécanismes de jeu

Nous avons fait de notre mieux pour développer une interface graphique intuitive à manier et agréable à naviguer, tout en veillant à soigner l’esthétique du design de celle-ci. Nous avons pris comme direction artistique d’avoir un thème plutôt minimaliste, tout en nuances pastels, ce fut dans l’optique de cultiver l’atmosphère de zénitude que nous inspire le jeu Niwa, ce dernier prenant place dans la version plateau dans un jardin japonais.

## Sauvegarde

Nous avons implémenter la possibilité de sauvegarder une partie à la demande du joueur. Quand le joueur, en pleine partie, choisira de cliquez sur le bouton menu: il aura 4 possibilités dont celle de sauvegarder sa partie puis quitter sa partie. Nous avons donc permis la Sérialisation de toutes les classes le nécessitant (et seulement celles dont il est nécessaire de l’implémenter afin d’éviter certains problèmes de performances). Nous sérialisons donc la partie, c’est-à-dire la classe InterfaceDeJeu gérant touts les aspects (graphiques et déroulement de la partie).

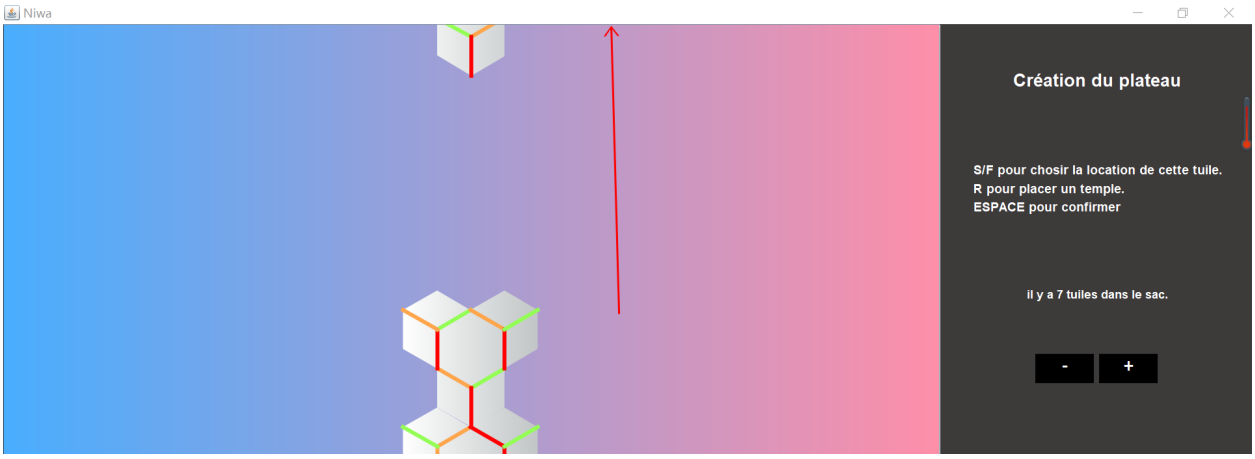
- Création de la sauvegarde : On commence par vérifier que l’état du jeu est acceptable : c’est-à-dire que si le joueur s’est déplacer, on vérifie que la perle du pion a été déplacer ou que le nom n’est pas une chaîne de caractères vide ou ne dépasse pas 12 caractères, ne contient que des caractères alpha-numériques ou encore que le nom n’est pas déjà utiliser par une autre sauvegarde. Quand le nom entrez ne fera plus parti de ces cas-là, on créera la sauvegarde pour ensuite interrompre le jeu et aller directement au menu principal.
- Ouverture de la sauvegarde : On propose d’abord au.x joueur.s (si une sauvegarde existe bien sûre) d’ouvrir la sauvegarde de leurs choix, sauvegardes qui s’afficheront dans une liste. Quand le.s joueur.s auront choisit la partie souhaitée, le JOptionPane comportant la liste renverra un fichier (qui est un objet InterfaceDeJeu sérialisé) qu’on désérialisera pour ensuite lancer la partie demandée.

# Effets sonores

- Ajout de musiques d'ambiance pour le jeu
- Ajout de son lorsque l'on clique sur les boutons

# Effets visuels

## Le plateau s'agrandit en fonction du déplacement de la tuile



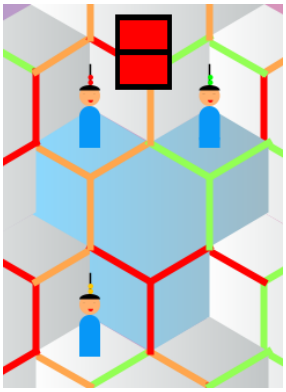

## Quand on ne peut pas placer une tuile à cause d'une contrainte, le plateau "tremble”

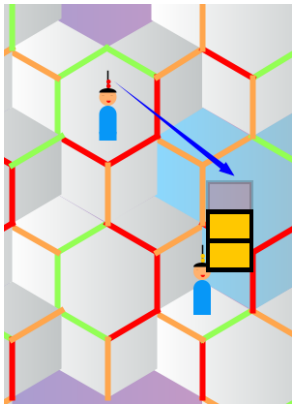
## Zoom, dezoom et déplacement du plateau

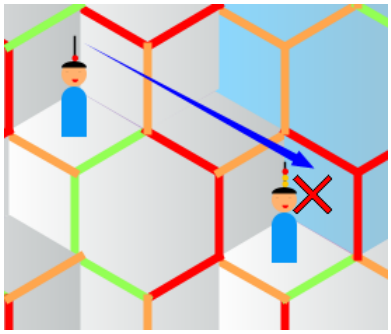
pour une meilleure visualisation des déplacements


## Amélioration du visuel des perles

On permet aux joueurs d'observer plus facilement les déplacements des perles quand le curseur est sur le pion



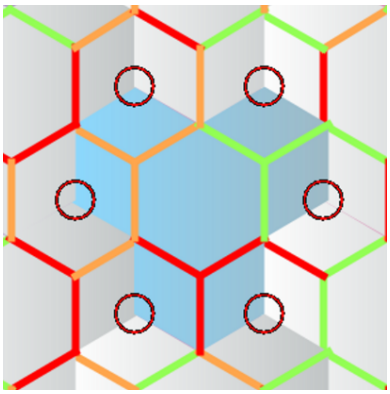
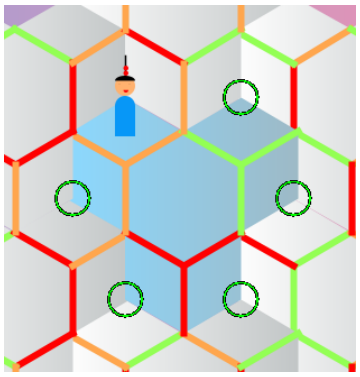
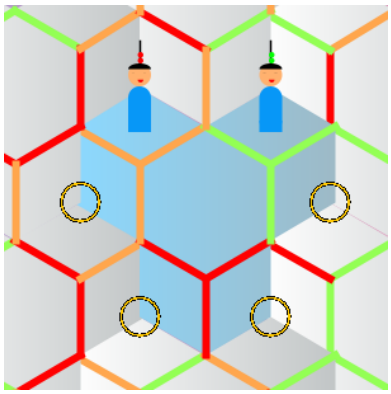




Un pion ne peut pas avoir plus de 3 perles dans ce cas-là

## Indication visuelle de l’ordre de placement des pions de départ

en changeant de la couleur des cercles, qui permettre aux joueurs de faire des bons choix

💡 **Les problèmes rencontrés**

- Problème assez fréquent lorsqu'il s'agit d'importer des images, sons, etc… les chemins des fichiers peuvent causer des soucis (le programme ne trouve pas l'emplacement des ressources). Parfois, les chemins pouvaient fonctionner chez certains, chez d'autres non. Il fallait donc savoir bien gérer les ouvertures des fichiers ainsi que les chemins d'accès.

# 4. Principes de l'algorithme pour la conception du joueur IA

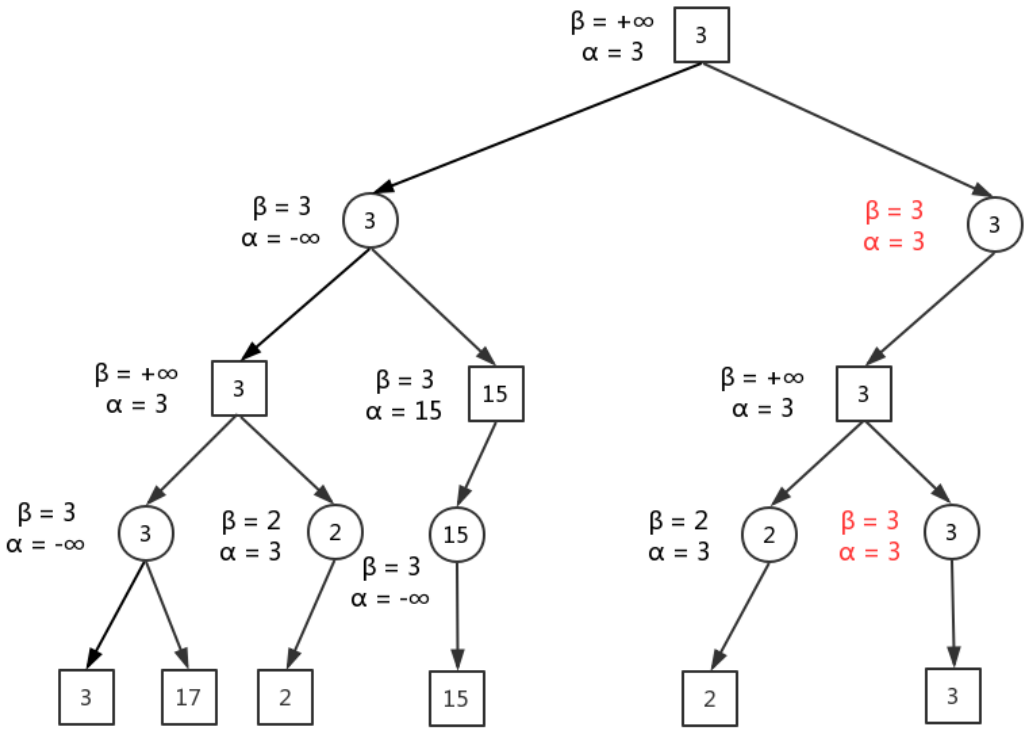
## Minimax?

**Algorithme MiniMax**

L'algorithme MiniMax, également appelé algorithme de minimax, est un algorithme qui cherche la valeur minimale de la plus grande probabilité de défaite.

L'algorithme MiniMax est souvent utilisé dans les jeux et programmes qui mettent en concurrence deux parties. Cet algorithme est un algorithme à somme nulle, c'est-à-dire que l'une des parties choisit une option qui maximise son avantage parmi les options disponibles, tandis que l'autre partie choisit une méthode pour minimiser l'avantage de l'adversaire. Au début, la somme totale est de 0.

**Élagage alpha-bêta**






L'Élagage alpha-bêta tire son nom des deux limites qui sont transmises lors du calcul et qui limitent l'ensemble des solutions possibles en fonction de la partie de l'arbre de recherche déjà vue. Alpha (α) représente l'intervalle inférieur maximal actuellement observé parmi toutes les solutions possibles et Beta (β) représente l'intervalle supérieur minimal actuellement observé parmi toutes les solutions possibles.

Ainsi, si un nœud de l'arbre de recherche est considéré comme un nœud sur la voie de la meilleure solution (ou comme un nœud jugé nécessaire à la recherche), il satisfait la condition suivante (N est la valeur d'estimation du nœud courant) :
$\alpha \leq N \leq \beta$

Au cours de la résolution, α et β se rapprochent progressivement. Si pour un nœud, α > β se produit, cela signifie que ce point ne produira certainement pas de meilleure solution et n'a donc plus besoin d'être étendu (c'est-à-dire qu'il n'a plus besoin de générer de nœuds enfants), ce qui permet de couper l'arbre de jeu.

## Fonction d'évaluation

La fonction d'évaluation est une fonction utilisée pour évaluer l'état d'une partie à un moment donné. Elle est généralement utilisée dans l'algorithme MiniMax pour décider du meilleur coup à jouer. Les trois fonctions d'évaluation ci-dessus sont des exemples de fonctions d'évaluation couramment utilisées dans les jeux de plateau tels que les échecs ou le reversi.

(1) **f (état)** :elle calcule la distance entre le pion et la base adverse
(2) **g (état)** :elle calcule le nombre de perles que le pion possède

Il est important de noter que ces fonctions d'évaluation ne sont que des exemples et qu'il existe de nombreuses autres façons d'évaluer l'état d'une partie. Il est également possible de combiner plusieurs de ces fonctions d'évaluation avec des coefficients en une seule fonction d'évaluation plus complète.

Par exemple, on définit une nouvelle fonction d'évaluation:

> eval = **opponentBaseDistanceWeight**·f(**état**) + **pearlsWeight**·g(**état**)

Cette fonction d'évaluation estime le score de l'état du jeu en fonction de la distance des personnages du joueur par rapport à la base adverse et du nombre de perles que les personnages possèdent. Plus le score est élevé, plus l'état du jeu est favorable au joueur ; plus le score est faible, moins l'état du jeu est favorable au joueur.

💡 **les problèmes rencontrés**

1. L'algorithme Minimax est un algorithme de recherche adapté aux scénarios de jeu finis et d'information complète. Dans ce projet, l'IA ne peut pas utiliser l'algorithme Minimax en raison de l'espace d'états trop vaste, la profondeur et la largeur de l'arbre de recherche sont très importantes, rendant le processus de recherche très complexe et long.
2. De plus, le résultat du jeu peut être influencé par de nombreux facteurs, tels que la disposition du plateau de jeu, les stratégies des joueurs, etc., ce qui rend difficile la conception de la fonction d'évaluation.
    a. Je n'ai pas pris en compte deux situations :
        1. Lorsque deux personnages se trouvent sur des cases adjacentes, ils rencontrent. Si l'un des personnages n'a pas de perle sur sa tête, il peut sauter par-dessus l'autre personnage et atterrir sur l'une des 3 cases situées derrière lui. S'il n'y a pas assez de places, il peut se déplacer jusqu'à l'une des 2 cases suivantes (en sautant deux cases).
        1. Si les deux personnages ont des perles, ils se bloquent mutuellement le passage sur leur case respective.
    b. De plus, le fait d'avoir beaucoup de perles ne sert à rien si le nombre de perles sur la tête du personnage ne correspond pas à la couleur de la ligne correspondante.
3. Par conséquent, l'IA de ce projet n'est pas adaptée pour utiliser l'algorithme Minimax.

## Q-learning naif…

Le Q-learning est un algorithme d'apprentissage par renforcement basé sur la valeur, apprenant continuellement la stratégie optimale en interagissant avec l'environnement. Dans ce projet, l'IA utilise le Q-learning pour apprendre la stratégie optimale.

1. Initialement, la table Q est initialisée à zéro. Ensuite, à chaque étape, l'IA sélectionne une action en fonction de l'état actuel et des actions possibles, observe

les commentaires de l'environnement et le nouvel état généré.À l'aide de ces informations, l'IA met à jour la table Q et apprend progressivement la stratégie optimale. Plus précisément :

2. Principe Q-learning : Le Q-learning est un algorithme d'apprentissage par renforcement qui apprend la fonction optimale de valeur état-action (Q) pour guider l'agent dans ses décisions.

## Schéma UML des classes relatives à cette implémentation

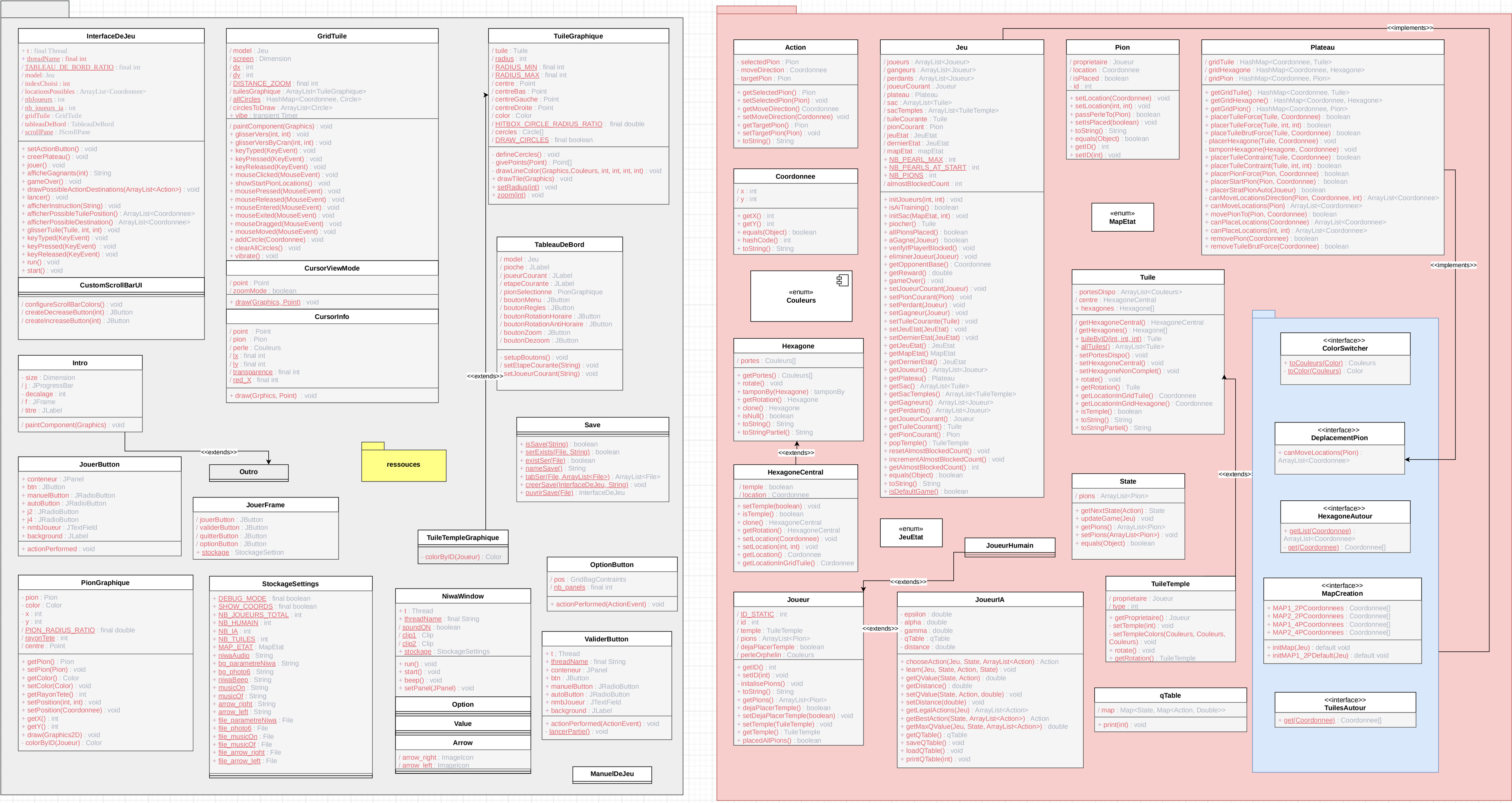
3. Conception de l'état : Dans ce projet, la classe State représente l'état du jeu, y compris la position du joueur, les combinaisons de couleurs, etc.
4. Conception de l'action : La classe Action représente les actions que l'agent peut prendre, y compris la sélection des pions, la direction du mouvement et le pion cible pour accepter la perle.
5. Maintenance de la table Q : Nous utilisons une HashMap pour stocker les paires état-action et leurs valeurs Q correspondantes. Au cours de l'entraînement, nous mettons à jour les valeurs de la table Q en fonction de la règle de mise à jour du Q-learning.

```
newQValue = currentQValue + alpha * (reward + gamma * maxNextQValue - currentQValue)
```

6. Stratégie epsilon-greedy :
    a. Au début de l'entraînement, l'agent explore avec une probabilité plus élevée (choisissant des actions aléatoires) pour mieux comprendre l'environnement ( epsilon=1.0）.
    b. Au fur et à mesure de l'entraînement, la probabilité d'exploration diminue et l'agent commence à exploiter les connaissances acquises (choisissant l'action avec la valeur Q la plus élevée) ( epsilon=0.1）.

💡 **les problèmes rencontrés**

1. Espace d'état trop grand : L'espace d'état du jeu est assez grand, ce qui rend la dimension de la table Q importante et nécessite davantage de ressources de stockage et de calcul.
2. Vitesse d'apprentissage lente : Pendant le processus d'apprentissage, la mise à jour de la table Q nécessite de nombreuses itérations et calculs, ce qui ralentit la vitesse d'apprentissage.
3. Fichiers .ser trop volumineux : Les fichiers .ser utilisés pour enregistrer la table Q sont volumineux, ce qui rallonge le temps de chargement des fichiers et ralentit davantage la vitesse d'apprentissage.


InterfaceDeJeu
+ t : final Thread
+ threadName : final int
/ TABLEAU_DE_BORD_RATIO : final int
/ model: Jeu
/ indexChoisi : int
/ locationsPossibles : ArrayList<Coordonnee>
/ nbJoueurs : int
/ nb_joueurs_ia : int
/ gridTuile : GridTuile
/ tableauDeBord : TableauDeBord
/ scrollPane : JScrollPane
+ setActionButton() : void
+ creerPlateau() : void
+ jouer() : void
+ afficheGagnants(int) : String
+ gameOver() : void
+ drawPossibleActionDestinations(ArrayList<Action>) : void
+ lancer() : void
+ afficherInstruction(String) : void
+ afficherPossibleTuilePosition() : ArrayList<Coordonnee>
+ afficherPossibleDestination() : ArrayList<Coordonnee>
+ glisserTuile(Tuile, int, int) : void
+ keyTyped(KeyEvent) : void
+ keyPressed(KeyEvent) : void
+ keyReleased(KeyEvent) : void
+ run() : void
+ start() : void
CustomScrollBarUI
/ configureScrollBarColors() : void
/ createDecreaseButton(int) : JButton
/ createIncreaseButton(int) : JButton
GridTuile
/ model : Jeu
/ screen : Dimension
/ dx : int
/ dy : int
/ DISTANCE_ZOOM : final int
/ tuilesGraphique : ArrayList<TuileGraphique>
/ allCircles : HashMap<Coordonnee, Circle>
/ circlesToDraw : ArrayList<Circle>
+ vibe : transient Timer
/ paintComponent(Graphics) : void
+ glisserVers(int, int) : void
+ glisserVersByCran(int, int) : void
+ keyTyped(KeyEvent) : void
+ keyPressed(KeyEvent) : void
+ keyReleased(KeyEvent) : void
+ mouseClicked(MouseEvent) : void
+ showStartPionLocations() : void
+ mousePressed(MouseEvent) : void
+ mouseReleased(MouseEvent) : void
+ mouseEntered(MouseEvent) : void
+ mouseExited(MouseEvent) : void
+ mouseDragged(MouseEvent) : void
+ mouseMoved(MouseEvent) : void
+ addCircle(Coordonnee) : void
+ clearAllCircles() : void
+ vibrate() : void
CursorViewMode
/ point : Point
/ zoomMode : boolean
+ draw(Graphics, Point) : void
CursorInfo
/ point : Point
/ pion : Pion
/ perle : Couleurs
/ lx : final int
/ ly : final int
/ transparence : final int
/ red_X : final int
+ draw(Grphics, Point) : void
TuileGraphique
/ tuile : Tuile
/ radius : int
/ RADIUS_MIN : final int
/ RADIUS_MAX : final int
/ centre : Point
/ centreBas : Point
/ centreGauche : Point
/ centreDroite : Point
/ color : Color
/ HITBOX_CIRCLE_RADIUS_RATIO : final double
/ cercles : Circle[]
/ DRAW_CIRCLES : final boolean
- defineCercles() : void
- givePoints(Point) : Point[]
- drawLineColor(Graphics,Couleurs, int, int, int, int) : void
+ drawTile(Graphics) : void
+ setRadius(int) : void
+ zoom(int) : void
TableauDeBord
/ model : Jeu
/ pioche : JLabel
/ joueurCourant : JLabel
/ etapeCourante : JLabel
/ pionSelectionne : PionGraphique
/ boutonMenu : JButton
/ boutonRegles : JButton
/ boutonRotationHoraire : JButton
/ boutonRotationAntiHoraire : JButton
/ boutonZoom : JButton
/ boutonDezoom : JButton
- setupBoutons() : void
/ setEtapeCourante(String) : void
/ setJoueurCourant(String) : void
<<extends>>
Intro
- size : Dimension
/ j : JProgressBar
- decalage : int
/ f : JFrame
/ titre : JLabel
/ paintComponent(Graphics) : void
<<extends>>
Outro
ressouces
Save
+ isSave(String) : boolean
+ serExists(File, String) : boolean
+ existSer(File) : boolean
+ nameSave() : String
+ tabSer(File, ArrayList<File>) : ArrayList<File>
+ creerSave(InterfaceDeJeu, String) : void
+ ouvrirSave(File) : InterfaceDeJeu
JouerButton
+ conteneur : JPanel
+ btn : JButton
+ manuelButton : JRadioButton
+ autoButton : JRadioButton
+ j2 : JRadioButton
+ j4 : JRadioButton
+ nmbJoueur : JTextField
+ background : JLabel
+ actionPerformed : void
JouerFrame
/ jouerButton : JButton
/ validerButton : JButton
/ quitterButton : JButton
/ optionButton : JButton
+ stockage : StockageSettiion
TuileTempleGraphique
- colorByID(Joueur) : Color
OptionButton
/ pos : GridBagContraints
/ nb_panels : final int
+ actionPerformed(ActionEvent) : void
ValiderButton
+ t : Thread
+ threadName : final String
+ conteneur : JPanel
+ btn : JButton
+ manuelButton : JRadioButton
+ autoButton : JRadioButton
+ nmbJoueur : JTextField
+ background : JLabel
+ actionPerformed(ActionEvent) : void
- lancerPartie() : void
PionGraphique
- pion : Pion
- color : Color
- x : int
- y : int
/ PION_RADIUS_RATIO : final double
/ rayonTete : int
/ centre : Point
+ getPion() : Pion
+ setPion(Pion) : void
+ getColor() : Color
+ setColor(Color) : void
+ getRayonTete() : int
+ setPosition(int, int) : void
+ setPosition(Coordonnee) : void
+ getX() : int
+ getY() : int
+ draw(Graphics2D) : void
- colorByID(Joueur) : Color
StockageSettings
+ DEBUG_MODE : final boolean
+ SHOW_COORDS : final boolean
+ NB_JOUEURS_TOTAL : int
+ NB_HUMAIN : int
+ NB_IA : int
+ NB_TUILES : int
+ MAP_ETAT : MapEtat
+ niwaAudio : String
+ bg_parametreNiwa : String
+ bg_photo6 : String
+ niwaBeep : String
+ musicOn : String
+ musicOf : String
+ arrow_right : String
+ arrow_left : String
+ file_parametreNiwa : File
+ file_photo6 : File
+ file_musicOn : File
+ file_musicOf : File
+ file_arrow_right : File
+ file_arrow_left : File
NiwaWindow
+ t : Thread
+ threadName : final String
/ soundON : boolean
/ clip1 : Clip
/ clip2 : Clip
+ stockage : StockageSettings
+ run() : void
+ start() : void
+ beep() : void
+ setPanel(JPanel) : void
Option
Value
Arrow
/ arrow_right : ImageIcon
/ arrow_left : ImageIcon
ManuelDeJeu
Action
- selectedPion : Pion
- moveDirection : Coordonnee
- targetPion : Pion
+ getSelectedPion() : Pion
+ setSelectedPion(Pion) : void
+ getMoveDirection() Coordonnee
+ setMoveDirection(Cordonnee) void
+ getTargetPion() : Pion
+ setTargetPion(Pion) : void
+ toString() : String
Coordonnee
/ x : int
/ y : int
+ getX() : int
+ getY() : int
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
«enum»
Couleurs
Hexagone
/ portes : Couleurs[]
+ getPortes() : Couleurs[]
+ rotate() : void
+ tamponBy(Hexagone) : tamponBy
+ getRotation() : Hexagone
+ clone() : Hexagone
+ isNull() : boolean
+ toString() : String
+ toStringPartiel() : String
<<extends>>
HexagoneCentral
/ temple : boolean
/ location : Coordonnee
+ setTemple(boolean) : void
+ isTemple() : boolean
+ clone() : HexagoneCentral
+ getRotation() : HexagoneCentral
+ setLocation(Coordonnee) : void
+ setLocation(int, int) : void
+ getLocation() : Cordonnee
+ getLocationInGridTuile() : Cordonnee
Jeu
/ joueurs : ArrayList<Joueur>
/ gangeurs : ArrayList<Joueur>
/ perdants : ArrayList<Joueur>
/ joueurCourant : Joueur
/ plateau : Plateau
/ sac : ArrayList<Tuile>
/ sacTemples : ArrayList<TuileTemple>
/ tuileCourante : Tuile
/ pionCourant : Pion
/ jeuEtat : JeuEtat
/ dernierEtat : JeuEtat
/ mapEtat : mapEtat
+ NB_PEARL_MAX : int
+ NB_PEARLS_AT_START : int
+ NB_PIONS : int
/ almostBlockedCount : int
+ initJoueurs(int, int) : void
+ isAiTraining() : boolean
+ initSac(MapEtat, int) : void
+ piocher() : Tuile
+ allPionsPlaced() : boolean
+ aGagne(Joueur) : boolean
+ verifyIfPlayerBlocked() : void
+ eliminerJoueur(Joueur) : void
+ getOpponentBase() : Coordonnee
+ getReward() : double
+ gameOver() : void
+ setJoueurCourant(Joueur) : void
+ setPionCourant(Pion) : void
+ setPerdant(Joueur) : void
+ setGagneur(Joueur) : void
+ setTuileCourante(Tuile) : void
+ setJeuEtat(JeuEtat) : void
+ setDernierEtat(JeuEtat) : void
+ getJeuEtat() : JeuEtat
+ getMapEtat() MapEtat
+ getDernierEtat() : JeuEtat
+ getJoueurs() : ArrayList<Joueur>
+ getPlateau() : Plateau
+ getSac() : ArrayList<Tuile>
+ getSacTemples() : ArrayList<TuileTemple>
+ getGagneurs() : ArrayList<Joueur>
+ getPerdants() : ArrayList<Joueur>
+ getJoueurCourant() : Joueur
+ getTuileCourant() : Tuile
+ getPionCourant() : Pion
+ popTemple() : TuileTemple
+ resetAlmostBlockedCount() : void
+ incrementAlmostBlockedCount() : void
+ getAlmostBlockedCount() : int
+ equals(Object) : boolean
+ toString() : String
+ isDefaultGame() : boolean
«enum»
JeuEtat
JoueurHumain
<<extends>>
Joueur
/ ID_STATIC : int
/ id : int
/ temple : TuileTemple
/ pions : ArrayList<Pion>
/ dejaPlacerTemple : boolean
/ perleOrphelin : Couleurs
+ getID() : int
+ setID(int) : void
- initalisePions() : void
+ toString() : String
+ getPions() : ArrayList<Pion>
+ dejaPlacerTemple() : boolean
+ setDejaPlacerTemple(boolean) : void
+ setTemple(TuileTemple) : void
+ getTemple() : TuileTemple
+ placedAllPions() : boolean
<<extends>>
JoueurIA
- epsilon : double
- alpha : double
- gamma : double
- qTable : qTable
- distance : double
+ chooseAction(Jeu, State, ArrayList<Action>) : Action
+ learn(Jeu, State, Action, State) : void
+ getQValue(State, Action) : double
+ getDistance() : double
+ setQValue(State, Action, double) : void
+ setDistance(double) : void
+ getLegalActions(Jeu) : ArrayList<Action>
+ getBestAction(State, ArrayList<Action>) : Action
+ getMaxQValue(Jeu, State, ArrayList<Action>) : double
+ getQTable() : qTable
+ saveQTable() : void
+ loadQTable() : void
+ printQTable(int) : void
Pion
/ proprietaire : Joueur
/ location : Coordonnee
/ isPlaced : boolean
- id : int
+ setLocation(Coordonnee) : void
+ setLocation(int, int) : void
+ passPerleTo(Pion) : boolean
+ setIsPlaced(boolean) : void
+ toString() : String
+ equals(Object) : boolean
+ getID() : int
+ setID(int) : void
«enum»
MapEtat
Plateau
/ gridTuile : HashMap<Coordonnee, Tuile>
/ gridHexagone : HashMap<Coordonnee, Hexagone>
/ gridPion : HashMap<Coordonnee, Pion>
+ getGridTuile() : HashMap<Coordonnee, Tuile>
+ getGridHexagone() : HashMap<Coordonnee, Hexagone>
+ getGridPion() : HashMap<Coordonnee, Pion>
+ placerTuileForce(Tuile, Coordonnee) : boolean
+ placerTuileForce(Tuile, int, int) : boolean
+ placeTuileBrutForce(Tuile, Coordonnee) : boolean
- placerHexagone(Tuile, Coordonnee) : void
- tamponHexagone(Hexagone, Coordonnee) : void
+ placerTuileContraint(Tuile, Coordonnee) : boolean
+ placerTuileContraint(Tuile, int, int) : boolean
+ placerPionForce(Pion, Coordonnee) : boolean
+ placerStartPion(Pion, Coordonnee) : boolean
+ placerStratPionAuto(Joueur) : boolean
- canMoveLocationsDirection(Pion, Coordonnee, int) : ArrayList<Coordonnee>
+ canMoveLocations(Pion) : ArrayList<Coordonnee>
+ movePionTo(Pion, Coordonnee) : boolean
+ canPlaceLocations(Coordonnee) : ArrayList<Coordonnee>
+ canPlaceLocations(int, int) : ArrayList<Coordonnee>
+ removePion(Coordonnee) : boolean
+ removeTuileBrutForce(Coordonnee) : boolean
<<implements>>
<<implements>>
Tuile
- portesDispo : ArrayList<Couleurs>
/ centre : HexagoneCentral
+ hexagones : Hexagone[]
/ getHexagoneCentral() : HexagoneCentral
/ getHexagones() : Hexagone[]
+ tuileByID(int, int, int) : Tuile
+ allTuiles() : ArrayList<Tuile>
- setPortesDispo() : void
- setHexagoneCentral() : void
- setHexagoneNonComplet() : void
+ rotate() : void
+ getRotation() : Tuile
+ getLocationInGridTuile() : Coordonnee
+ getLocationInGridHexagone() : Coordonnee
+ isTemple() : boolean
+ toString() : String
+ toStringPartiel() : String
<<extends>>
State
/ pions : ArrayList<Pion>
+ getNextState(Action) : State
+ updateGame(Jeu) : void
+ getPions() : ArrayList<Pion>
+ setPions(ArrayList<Pion>) : void
+ equals(Object) : boolean
TuileTemple
/ proprietaire : Joueur
/ type : int
+ getProprietaire() : Joueur
- setTemple(int) : void
- setTempleColors(Couleurs, Couleurs, Couleurs) : void
+ rotate() : void
+ getRotation() : TuileTemple
qTable
/ map : Map<State, Map<Action, Double>>
+ print(int) : void
<<interface>>
ColorSwitcher
+ toCouleurs(Color) : Couleurs
- toColor(Couleurs) : Color
<<interface>>
DeplacementPion
+ canMoveLocations(Pion) : ArrayList<Coordonnee>
<<interface>>
HexagoneAutour
+ getList(Coordonnee) : ArrayList<Coordonnee>
- get(Coordonnee) : Coordonnee[]
<<interface>>
MapCreation
+ MAP1_2PCoordonnees : Coordonnee[]
+ MAP2_2PCoordonnees : Coordonnee[]
+ MAP1_4PCoordonnees : Coordonnee[]
+ MAP2_4PCoordonnees : Coordonnee[]
+ initMap(Jeu) : default void
+ initMAP1_2PDefault(Jeu) : default void
<<interface>>
TuilesAutour
+ get(Coordonnee) : Coordonnee[]
Launcher
+ main(String[]) : void
Légende :
+ : public
- : private
/ : proctected
int : type de variable
variable : variable statique
function(type_arg1, ..., type_argn) : fonction statique
function(type_arg1, ..., type_argn) : fonction
void : type de retour
A
B
<<...>>
: lien de type L de A vers B