



# Jeu du tarot Africain

Nicolas DIAS  
Thomas PRÉVOST

# Table des matières

I.	Présentation du projet . . . . .	2
1.	Règles du jeu et objectifs du projet . . . . .	2
2.	Pistes envisagées pour l'implémentation . . . . .	3
3.	Choix techniques . . . . .	3
II.	Le programme sur papier . . . . .	4
III.	Le programme en fonctionnement . . . . .	4
1.	Figures imposées . . . . .	4
2.	Tests effectués . . . . .	5
IV.	Détail du fonctionnement par module . . . . .	6
1.	Module <b>Tarot_Africain</b> . . . . .	6
2.	Module <b>Joueurs</b> . . . . .	7
3.	Module <b>TA_Bots</b> . . . . .	7
V.	Limitations & perspectives . . . . .	9
1.	Limitations observées . . . . .	9
2.	Perspectives et améliorations . . . . .	9

*Le code source complet du projet est disponible sur GitHub :*

<https://github.com/thomas40510/ProjetInfoS2>

---

## I. Présentation du projet

Nous avons choisi, pour ce projet d'informatique, d'implémenter en Python le jeu du Tarot Africain, qui est un jeu de cartes opposant de deux à quatre joueurs, et demandant stratégie, réflexion mais aussi une part de chance.

### 1. Règles du jeu et objectifs du projet

Une partie de Tarot Africain se déroule assez simplement. Pour commencer, il faut se munir d'un jeu de tarot, que l'on trie. On place alors d'un côté les atouts (du 1 au 21 ainsi que l'excuse) et d'un autre le reste des cartes séparé et trié par couleur<sup>1</sup>.

#### Début de partie

Donner, à chaque joueur, un paquet trié de cartes de couleur : elles représentent les vies du joueur et sont placées devant lui, visibles de tous les autres joueurs. Les cartes de tarot seront mélangées entre chaque partie.

#### Déroulement d'une partie

Au début de chaque manche, un joueur désigné distribue 5 cartes d'atout à chaque joueur. Chaque joueur, en commençant par le celui ayant distribué, placera alors un pari sur le nombre de tours qu'il va gagner après avoir consulté ses cartes. Chaque joueur est libre de proposer un nombre de son choix, à l'exception du dernier à parler devant proposer un nombre tel que l'ensemble des paris soit différent du nombre de cartes que les joueurs ont en main (5 au premier tour).

Les joueurs jouent alors à tour de rôle une carte de leur main, en commençant toujours par celui ayant distribué. Gagne le tour le joueur ayant placé la carte la plus forte (à noter que le joueur ayant placé l'excuse choisit sa valeur en fonction de sa stratégie : soit elle est la plus forte, soit elle est la plus faible). Ce joueur débutera le tour suivant, et ainsi de suite jusqu'à ce qu'il ne reste aucune carte en main.

En fin de manche, les joueurs perdent autant de vies qu'il leur manque de plis pour remplir leur pari (par exemple, un joueur ayant placé un pari de 3 et n'ayant gagné qu'un pli perd  $3 - 1 = 2$  vies).

Commence alors la manche suivante, au cours de laquelle seront distribuées 4 cartes (puis 3 à la suivante, puis 2, puis une seule).

---

1. pique, carreau...

### **Tour à une seule carte**

Le tour à une seule carte est particulier : chaque joueur ne regarde pas sa carte, mais la pose sur son front. Il est donc capable de voir uniquement les cartes des autres joueurs et estime s'il est capable de remporter ce pli.

Là encore, s'applique la même condition sur le dernier joueur à parier (la somme totale ne peut pas être égale à 1).

### **Règle de la « remontée »**

Une règle supplémentaire existe : si, à l'issue d'une manche, un joueur est le seul à ne pas perdre de point, il en regagne un.

### **Fin de partie**

Une partie se finit à l'issue de la manche d'une seule carte. À ce moment là, le jeu peut continuer, chaque joueur gardant le nombre de vies qu'il avait à l'issue de la partie précédente. Le joueur ayant distribué transmettant ce rôle à celui à sa gauche.

Le jeu continue alors de partie en partie jusqu'à ce que tous les joueurs sauf un soient éliminés.

## **2. Pistes envisagées pour l'implémentation**

Il a alors été initialement choisi, pour implémenter le jeu de Tarot Africain, le découpage en plusieurs classes : une classe **Joueur**, une classe **Carte**, une classe **Manche** et une classe **Partie**. Structure permettant de manipuler ces différents objets afin de dérouler une partie complète.

Une première structure du programme a alors été schématisée à partir de ces pistes techniques, soulevant alors plusieurs questions au sein du binôme, impliquant finalement de reconsidérer le découpage qui avait été réalisé initialement.

## **3. Choix techniques**

Notre choix, après l'étude préalable ayant été menée, s'est donc porté sur un découpage plus simple que celui initialement prévu : la classe **Carte**, bien trop simple, n'ayant pas de raison d'exister.

Le projet s'articule donc, à ce stade, autour de 3 classes principales :

- La classe **Joueur**, de laquelle sont héritées les classes **JoueurHumain** et **JoueurBot** permettant de faire jouer un humain ou la machine ;
- La classe **Manche** permettant de dérouler une manche de la partie ;
- La classe **Tarot** permettant de dérouler la partie.

En particulier, la classe **JoueurHumain**, permet de faire jouer l'utilisateur en lui demandant de rentrer chaque coup, tandis que la classe **JoueurBot** permet de faire jouer la machine en calculant chacun de ses coups, généralement de manière réursive, grâce au module `TA_Bots.py`. Le fonctionnement de ce dernier est explicité en IV.3.

## II. Le programme sur papier

Le programme construit autour d'un fichier principal contenant le jeu en lui-même, auquel viennent se greffer deux modules, respectivement pour les joueurs et le calcul des coups de la machine.

Le fonctionnement du programme, dans son ensemble, est donc plutôt simple. On pourra se référer entre autres au diagramme de classes et au diagramme d'actions présentés en annexe, mais aussi au détail par module proposé en IV..

- L'utilisateur lance une partie par l'exécution de la classe `Tarot`. Il spécifie son nom.
- Est alors initialisée la partie, avec le nom des joueurs ; une manche est créée ainsi que les joueurs (humain ou machine) qui sont en lice.
- À chaque tour, un leader est désigné (soit le premier joueur en début de manche, soit le vainqueur du tour précédent), et chaque joueur place ses paris et joue. Si le joueur est la machine, son coup est calculé en fonction des coups des autres joueurs et leurs paris ; si le joueur est humain, le programme lui demande son pari et la carte qu'il choisit de poser, connaissant les paris déjà placés et les cartes posées.
- À l'issue de chaque tour est identifié le vainqueur du pli, et les joueurs perdent le nombre de point correspondant à l'écart entre leur pari et leur nombre de plis.

## III. Le programme en fonctionnement

### 1. Figures imposées

Les figures imposées suivantes sont vérifiées par le programme :

1. **Factorisation du code** : le programme est composé de trois modules différents (`Tarot_Africain.py`, `TA_Bots.py` et `Joueurs.py`) responsables de diverses fonctionnalités bien spécifiques, et d'un module de tests, et est donc factorisé ;
2. **Documentation et commentaire du code** : tout naturellement, le programme est intégralement documenté et commenté ;
3. **Tests unitaires** : des tests unitaires ont été écrits pour chaque partie du programme ;
4. **Création d'un type d'objet (classe)** : cinq objets ont été créés dans le cadre de ce programme, à savoir `Joueur`, `Manche` et `Tarot`, mais aussi `JoueurHumain` et `JoueurBot` ;
5. **Récurtivité** : la récursivité est exploitée dans le jeu de la machine (module `TA_Bots.py`), permettant d'explorer les différentes situations possibles afin d'identifier le coup optimal ;
6. **Héritage entre deux types créés** : les classes `JoueurHumain` et `JoueurBot` héritent toutes deux de la classe `Joueur` ;

- 7. Héritage depuis un type intégré :** nous avons implémenté une « mémoire », héritée du type `list` de Python, prenant en charge les logs de toutes des différentes manches jouées au cours d'une partie, permettant non seulement d'effectuer des traitements statistiques sur les parties afin de permettre l'amélioration du fonctionnement des bots, mais aussi de conduire des tests unitaires. Une voie envisageable serait également de s'en servir pour proposer au joueur des statistiques sur sa partie.

## 2. Tests effectués

De nombreux tests ont été effectués afin de vérifier le bon fonctionnement et la cohérence du programme, ces derniers étant réunis dans le fichier `test_Tarot_Africain.py`. Nous détaillons donc ici les quatre jeux de tests unitaires ayant été écrits et exécutés.

Il est à noter cependant que seuls les tests pertinents ont été écrits : les classes n'ayant pas été testées l'ont été volontairement, car consistant uniquement en des instanciations et des fonctions d'affichage, ou des fonctions triviales (classes `Tarot` et `Log` essentiellement).

### Cohérence de la distribution

Un élément essentiel au bon déroulement de la partie est le fait qu'une même carte ne puisse être distribuée deux fois dans une même manche.

Cette condition est donc vérifiée pour chaque nombre de joueurs et chaque nombre de cartes possibles, pour un total de  $20 \times 4 \times 3 = 240$  tests. Ce nombre, correspondant à 20 tests par nombre de joueurs pour chaque nombre de cartes en main, permet en effet de vérifier statistiquement la cohérence de la distribution des cartes malgré que cette dernière soit aléatoire. Nous estimons que ce nombre de tests, s'ils reviennent tous positifs, est suffisant pour vérifier cette condition.

En l'occurrence, le résultat est bien revenu positif à chaque exécution.

### Cohérence des paris

De la même manière, une partie ne peut se dérouler correctement s'il est possible de placer des paris faux ou incohérents. Il était donc essentiel d'écrire des tests unitaires vérifiant cette cohérence.

Ainsi, pour chaque nombre possible de joueurs et chaque nombre possible de cartes en main, les paris des joueurs, aussi bien humains que machines, sont recueillis et testés afin de vérifier leur cohérence. En particulier, il est vérifié le fait que la somme des paris ne soit pas égale au nombre de cartes en main (du fait de la condition sur le pari du dernier joueur à parier), mais aussi que chaque pari est positif et inférieur au nombre total de plis possibles.

Sont donc, de la même manière que pour la cohérence de la distribution, exécutés  $20 \times 4 \times 3 = 240$  tests. Tous sont revenus positifs, la cohérence des paris est donc bonne.

## Cohérence de la partie

Au-delà de la cohérence des paris, est également essentielle la cohérence du choix des cartes, que ce soit par un joueur humain ou par la machine. En particulier, il est nécessaire de vérifier que ne sera jamais posée par un joueur une carte qu'il ne possède pas, et qu'il ne puisse pas poser deux fois la même carte.

Ce jeu de tests vérifie donc, pour chaque nombre de joueurs et chaque nombre de cartes en main possibles, si chaque carte posée était bien possédée par le joueur avant d'être jouée et qu'il ne l'a pas jouée deux fois.

Ce qui constitue donc un total de 1 800 tests.

Nous remarquerons que ce jeu de test, combiné aux tests sur la cohérence des paris, permet de tester le bon fonctionnement des différentes classes du module **Joueurs**, et ce quand bien même appel est fait à la classe **Manche** : en effet, les joueurs sont instanciés en fonction des paramètres de la manche (notamment pour la distribution des cartes, ayant été préalablement testée).

## Perte de points

La dernière chose à vérifier est le fait que les joueurs perdent bien des vies.

Pour cela, toujours pour chaque nombre de joueurs et chaque nombre de cartes en main possibles, il est vérifié pour vingt manches différentes qu'il n'en existe aucune au cours de laquelle aucun joueur n'a perdu de vie.

# IV. Détail du fonctionnement par module

Nous détaillons ici le fonctionnement des différents modules du jeu et de leurs classes et fonctions principales. Il est cependant porté à l'attention du lecteur que toutes les fonctions ne sont pas nécessairement explicitées ici : nous avons fait le choix de ne parler que des fonctions principales, les autres étant triviales avec comme seule utilité de faciliter le traitement ou les calculs<sup>2</sup>.

## 1. Module **Tarot\_Africain**

Ce module, qui constitue le fichier principal du programme, est responsable de la partie en elle-même, de sa création et son déroulement.

### Classe **Tarot**

La classe **Tarot** est la classe principale du programme. C'est elle qui, l'utilisateur ayant entré son nom, débute la partie et crée les manches successives de jeu. C'est également la classe ayant la responsabilité de gérer les vies des joueurs et de s'occuper des éventuels joueurs ayant perdu (fonction **enleve**).

Sans oublier le fait qu'elle désigne également le leader de la prochaine manche, tout en vérifiant qu'il peut prendre ce rôle (donc qu'il n'est pas mort).

Le tout étant réalisé par sa fonction **exe**.

---

2. Elles sont néanmoins toutes commentées

### Classe **Manche**

La classe **Manche** est la classe qui, comme son nom l'indique constitue une manche de jeu. Elle est instanciée à partir des informations concernant les joueurs, leur nombre de vies et le nombre de cartes à leur distribuer.

Fort de ces informations, elle créera joueurs et leur distribuera leurs cartes. Ceci fait, elle fera jouer les joueurs (fonction **jeu**), en ayant préalablement collecté leurs paris (appel à la fonction **paris**). Chaque joueur ayant posé sa carte, le vainqueur du pli est désigné.

Finalement, la fonction **jeu** renverra le vainqueur de la manche.

### Classe **Log**

Permettant de garder une trace de l'ensemble des manches d'une partie, la classe **Log** permet entre autres d'accéder à des statistiques calculées sur une partie ou un ensemble spécifique de manches, mais aussi à rendre possibles certains tests unitaires sur le bon déroulé d'une partie.

Cette classe ayant vocation à permettre le traitement des données d'une partie, nous proposons au lecteur des méthodes qui, sans la moindre prétention d'exhaustivité, montrent l'intérêt de cette classe pour faciliter l'analyse des parties *a posteriori*.

## 2. Module **Joueurs**

Le module **Joueurs** est responsable de la création des joueurs, qu'ils soient humain ou machine, et de leurs actions.

### Classe **JoueurHumain**

Il s'agit de la classe permettant à l'utilisateur de jouer. Elle l'interrogera pour chaque décision à prendre, aussi bien au niveau du pari (fonction **JoueurHumain.pari2**) ou de la carte à jouer (fonction **JoueurHumain.choixcartes2**).

En particulier, la cohérence de chaque entrée de l'utilisateur sera vérifiée par ces fonctions pour assurer le bon fonctionnement du jeu.

### Classe **JoueurBot**

Cette classe est celle représentant un joueur contrôlé par la machine. Chacune de ses décisions, que ce soit son pari (fonction **JoueurBot.pari2**) ou sa carte à jouer (fonction **JoueurBot.choixcartes2**), est déterminée comme étant la plus optimale grâce aux fonctions apportées par le module **TA\_Bots**.

## 3. Module **TA\_Bots**

Le module **TA\_Bots** contient toutes les fonctions utilisées pour le calcul des décisions des joueurs contrôlés par la machine.

### Fonction **pari1Carte**

Cette fonction permet calcule le pari de la machine lors de la manche à une seule carte. Elle fonctionne *grosso modo* selon le même raisonnement qu'un joueur



humain : si elle repère une carte forte chez les autres joueurs, le pari sera 0. Sinon, le pari sera de 1. La seule exception est le cas où le bot est le dernier à jouer ; dans ce cas le pari est entièrement déterminé par les autres paris.

### **Fonction `pariMCartes`**

Cette fonction permet de calculer le pari de la machine lors d'une manche à plusieurs cartes.

Le bot connaissant ses cartes et les paris placés par les joueurs avant lui, il calcule son pari par un rapport risque / gain selon le fonctionnement suivant :

- À partir des paris déjà placés, il évalue le risque en prenant en compte la somme des plis potentiellement gagnés par les parieurs, mais aussi le ratio joueurs ayant parié / nombre total de joueurs. Le risque vaut soit  $-1$  (peu de risque) soit  $1$  (risqué).
- Est alors parcouru carte par carte le jeu du bot, permettant de calculer, en fonction de la valeur de la carte et du risque, si un pli est envisageable avec cette carte sur une échelle de 0 (aucune chance) à 1 (très probable).
- Ces résultats sont ensuite sommés et sont calculés les trois entiers les plus proches de cette somme. L'entier qui constituera le pari sera déterminé par le respect des règles (condition sur le pari si le bot joue en dernier) mais sera préférentiellement le plus proche des trois.

Ceci permet donc à la fonction de renvoyer le pari optimal pour le bot, connaissant l'état de la manche et les cartes qu'il en main.

### **Fonction `choix1Carte`**

Cette fonction, qui est la plus simple, prend en charge le jeu du bot lors d'une manche à une seule carte.

Si ce dernier a l'excuse, la fonction choisira sa valeur (min ou max) en fonction du pari du bot. Sinon, la fonction renverra simplement la carte du bot.

### **Fonction `choixMCartes`**

Cette fonction, prenant en charge le choix de carte pour les manches à plusieurs cartes, commence par évaluer le risque en fonction des paris placés par les joueurs : si leur somme est supérieure au nombre de cartes, ils vont se battre pour gagner un maximum de plis (jeu au supérieur) ; sinon ils vont essayer de laisser le plus de plis possible aux autres (jeu à l'inférieur).

La fonction va donc traiter différemment ces deux cas :

- Si la manche joue au supérieur, il va falloir se battre pour gagner le bon nombre de plis, et donc choisir une carte qui maximise les chances de remporter un pli voulu (il est donc évident que l'ordre de jeu importe, le dernier à jouer étant sûr de son coup). Sera donc calculée, en fonction de la position du bot dans la manche, soit la meilleure carte s'il joue en dernier, soit, pour chaque joueur, la probabilité qu'il remporte le pli en fonction de son pari et des cartes précédemment posée (par un appel à la fonction `Proba`) ;
- Si la manche joue à l'inférieur, le but sera de forcer les autres joueurs à remporter des plis qu'ils ne veulent pas. Sera donc calculé, en fonction du

pari du bot, de ses cartes et de sa position dans la manche, la carte optimale permettant de gagner les plis voulus et laisser les autres aux adversaires.

### **Fonction Proba**

La fonction **Proba** évalue, de manière récursive, si la situation est favorable ou non pour le bot en connaissant ses cartes, le nombre de plis restant à gagner pour chaque joueur, la position du joueur et le joueur débutant la manche.

## **V. Limitations & perspectives**

### **1. Limitations observées**

Plusieurs limitations ont été identifiées, majoritairement en lien avec le fonctionnement du programme dans la console, sans interface utilisateur. Cette restriction étant à l'origine d'un affichage pas toujours intuitif et clair pour l'utilisateur, qui doit, pour jouer, se contenter d'un affichage textuel.

Malgré tout, en termes d'implémentation uniquement technique, aucune limitation majeure n'a pu être identifiée, mise à part une optimisation éventuelle des décisions du bot, qui dans le cadre de ce projet ne nous semble pas pertinente.

### **2. Perspectives et améliorations**

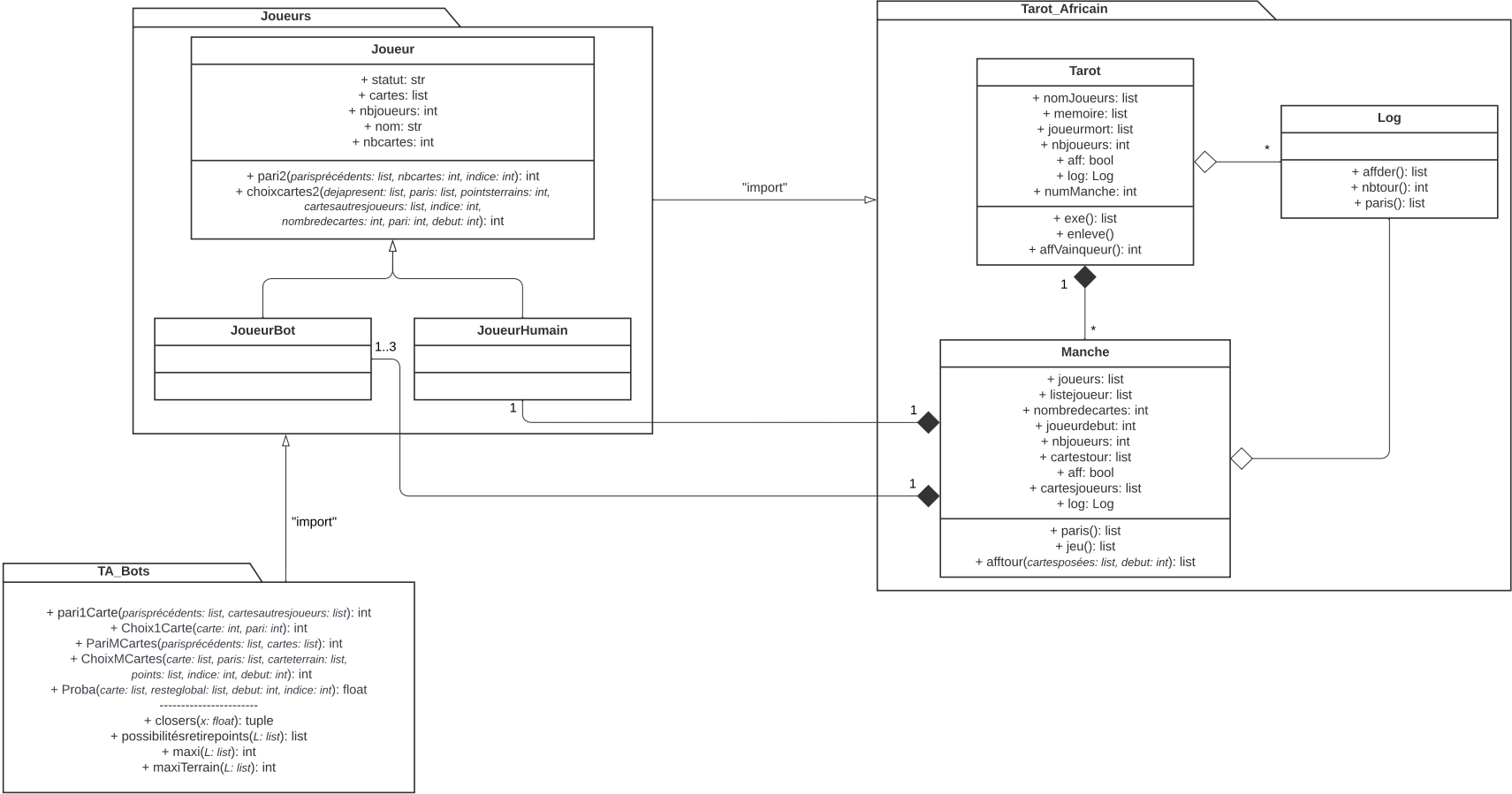
Ces constatations orientent donc tout naturellement la suite du projet vers l'intégration d'une interface utilisateur permettant d'apporter une solution aux différentes limitations induites par le mode de jeu en textuel, que nous avons pu identifier lors des tests du jeu.

Aucun choix technique n'a encore été fait concernant l'implémentation de l'UI, même si PyQt5 semble être une solution envisageable.

Une autre amélioration envisageable serait également de permettre à l'utilisateur de se créer un profil stocké sur une base de données, lui permettant d'accéder à différentes statistiques concernant ses parties précédentes. Cette piste ne sera pas nécessairement traitée immédiatement, la priorité étant donnée à l'implémentation de l'UI, mais reste ouverte comme perspective intéressante du projet.

Une dernière perspective d'amélioration, tenant uniquement de l'expérience utilisateur, serait d'ajouter la possibilité pour l'utilisateur de choisir la difficulté de la machine avant de commencer la partie. Ceci lui permettrait de jouer contre une machine plus faible s'il ne parvient pas à gagner contre une machine plus forte. À noter cependant que cette amélioration nécessiterait une refonte conséquente des algorithmes de décision, et serait hors de propos dans le cadre de ce projet, bien qu'intéressante.

# Annexe I. Diagramme de classes



# Annexe II. Fonctionnement schématique d'une partie

