

Mémoire
Réalisation d'un jeu d'Othello

Morgane Badré
Benjamin Letourneau,
Vincent Wilmet,
Nicolas Yvon

Client : M. Zeitoun
Chargé de TD : M. Frey

8 avril 2014

Remerciements

- M. Narbel, responsable de l'UE PDP pour les cours dispensés.
- M. Zeitoun, notre client, qui a pris sur son temps pour effectuer des réunions hebdomadaires et pour nous avoir donné de nombreux conseils.
- M. Frey, notre chargé de TD, pour nous avoir aidé et donné des conseils pour le PDP.
- M. le rapporteur inconnu qui a contribué de manière anonyme à notre évaluation.

Résumé

L'objectif de ce projet est de réaliser un jeu Othello permettant de jouer Humain opposé à une intelligence artificielle (IA) ou IA contre IA. Une IA est un logiciel qui a pour but de reproduire “intelligemment” un comportement spécifique, ici gagner une partie Othello. Le jeu, avec une interface intuitive, doit permettre à l'utilisateur de pouvoir choisir le niveau (difficulté) de l'IA en tant qu'aide et joueur.

Le projet s'est déroulé en différentes étapes. Tout d'abord, des recherches ont été effectuées afin de découvrir le jeu et sa communauté, mais également ses spécificités scientifiques. Puis, une analyse des besoins nous a permis de définir clairement les attentes du client. Dans un troisième temps, nous avons mis en place l'architecture du jeu. Le développement a ensuite commencé et pour terminer, une série de tests va être présentée.

Table des matières

Introduction	4
1 Analyse de l'existant	5
1.1 Éléments bibliographiques	5
1.1.1 Généralités sur les jeux d'accessibilité	5
1.1.2 Heuristiques et Algorithmes de force-brute	5
1.1.3 Algorithmes évolutionnistes et réseaux neuraux	6
1.2 Étude de l'existant	6
2 Analyse des besoins	7
2.1 Besoins fonctionnels	7
2.1.1 Jouer une partie / mettre en pause une partie	7
2.1.2 Réaliser une Intelligence Artificielle (IA)	7
2.1.3 Enregistrer une partie / Relancer une partie	8
2.1.4 Recommencer une partie à une certaine position	8
2.1.5 Possibilité de revenir un coup en arrière	8
2.1.6 Fournir le manuel utilisateur du jeu (interface et règles)	9
2.1.7 Proposer des suggestions de coups par l'IA	9
2.1.8 Chargement d'un fichier pré-configuré	9
2.1.9 Possibilité de choisir la taille de la grille de jeu	10
2.1.10 Menu pour choisir le temps de réflexion de l'IA	10
2.1.11 Possibilité d'échanger les joueurs	10
2.2 Besoins non fonctionnels	10
2.2.1 Besoins comportementaux	11
2.2.2 Besoins organisationnels	12
2.2.3 Besoins externes	15
3 Architecture et description du logiciel	16
3.1 Architecture du projet	16
3.1.1 Logiciel	17
3.1.2 Intelligences Artificielles	17
3.1.3 Gestionnaire de Fichier	18
3.1.4 Editeur de Plateau	19

3.1.5	BenchMark	20
3.1.6	Gestionnaire de Temps	21
3.2	Design patterns utilisés	22
3.2.1	State	22
3.2.2	Abstract Factory	23
3.2.3	Observer	23
3.2.4	Model-View-Controller	23
3.2.5	Singleton	24
3.2.6	Strategy	24
3.3	Heuristiques et Algorithmes de recherches de solution	25
3.3.1	Stratégie positionnelle	25
3.3.2	Mobilité	26
3.3.3	Maximisation	26
4	Fonctionnement et tests	29
4.1	Etat du projet	29
4.1.1	Logiciel	29
4.1.2	Editeur de plateau	32
4.1.3	Aide du jeu	33
4.2	Améliorations possibles sur le projet	33
4.2.1	Interface graphique	33
4.2.2	Fichier de sauvegarde	34
4.2.3	Module éditeur de plateau	34
4.2.4	Améliorations diverses	34
4.3	Protocoles de test	35
4.3.1	Tests unitaires	35
4.3.2	Tests d'intégration	36
4.3.3	Tests fonctionnels	36
4.3.4	Tests systèmes	36
4.3.5	Tests non réalisés	37
Conclusion		38
Annexes		42
A Prototypes papier de l'interface		42
A.1	Prototype papier du plateau de jeu	42
A.2	Prototype papier du retour en arrière pour les coups	43
A.3	Prototype papier des boutons et menus	43
B Diagrammes UML		44
B.1	Diagramme de machine à états	44
B.2	Diagramme de déploiement	45

B.3	Diagramme des cas d'utilisation	46
B.4	Diagramme de classes du Logiciel	47
B.4.1	Model du logiciel	47
B.4.2	Vue du logiciel	53
B.4.3	Contrôleur du logiciel	56
B.4.4	Utils du logiciel	58
B.5	Diagramme de classes Intelligence Artificielle	59
B.6	Diagramme de classes du gestionnaire de fichiers	60
B.7	Diagramme de classes de l'Editeur de plateaux	61
B.8	Diagramme de classes du Gestionnaire de temps	62
C	Pseudo-codes d'algorithmes	63
C.1	Algorithme MiniMax	63
C.2	Algorithme Alpha-Beta	64
D	Site Web	65
E	Poster	66

Introduction

Le travail que nous présentons dans ce rapport correspond à un projet informatique de programmation que nous avons réalisé en quadrinôme dans le cadre de notre première année de Master informatique de l'Université de Bordeaux.

Le thème de notre projet est de gérer la conduite d'un projet : développer un jeu Othello. Pour cela, nous sommes encadrés par un client (M. Zeitoun) qui nous exprime ses besoins et un chargé de TD (M. Frey) qui nous supervise et nous dirige au fur et à mesure de l'avancement du projet et de ses différentes étapes. Nous bénéficions également des techniques et conseils vus lors des cours dispensés par M. Narbel.

Les objectifs principaux attendus pour ce projet étaient : de définir les éléments nécessaires pour représenter les différentes configurations du jeu, sur plateaux de taille variable, typiquement 6x6 ou 8x8 cases, de programmer un joueur automatique naïf, suivant une heuristique très simple en début de partie, et explorant tout l'arbre des parties lorsqu'il reste peu de coups à jouer ; et d'utiliser une ou plusieurs heuristiques pour améliorer le joueur naïf (Minimax, Alpha-Beta, ...).

Othello est un jeu de stratégie à deux joueurs : Noir et Blanc. Il se joue sur un plateau unicolore de 64 cases, 8 sur 8 appelé othellier. Ces joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. Le but du jeu est d'avoir plus de pions de sa couleur que l'adversaire à la fin de la partie. Celle-ci s'achève lorsque aucun des deux joueurs n'a de coup légal à jouer. Cela intervient généralement lorsque les 64 cases sont occupées (définition de la Fédération Française d'Othello).

Il a été sujet à de nombreuses utilisations par des scientifiques lors de tournois. Ces rencontres permettaient de mesurer la performance de leur Intelligence Artificielle, les unes par rapport aux autres.

Ainsi ce projet est centré sur la réalisation du logiciel et plus particulièrement sur une IA compétitive. De plus, les différents besoins de notre client nous ont amenés à réaliser un logiciel Othello amélioré proposant des fonctionnalités supplémentaires au jeu Othello générique.

Notre rapport se décompose en quatre parties dont les premières correspondent aux recherches effectuées avec une étude de l'existant, une bibliographie et une analyse des besoins. Puis, nous présentons notre projet avec nos choix de conception, l'architecture logicielle, les spécificités du jeu, nos choix d'implémentation et les algorithmes utilisés. Enfin, nous évoquerons les fonctionnalités existantes, améliorations que nous pourrions apporter dans le futur ainsi que les différents tests que nous avons réalisés.

Chapitre 1

Analyse de l'existant

1.1 Éléments bibliographiques

Pour plus de lisibilité, nous avons partagé notre bibliographie en trois catégories. Une première partie contient les généralités sur les jeux d'accessibilité. La deuxième regroupe les algorithmes de force brute : IA basique et la dernière partie présente les algorithmes évolutionnistes et les réseaux neuraux.

1.1.1 Généralités sur les jeux d'accessibilité

Les jeux d'accessibilité, le plus souvent en tour par tour et entre deux joueurs, consistent à déplacer des pièces sur un plateau contenant un nombre fini de positions. Le but étant de capturer les positions du plateau ou bien les pièces de l'adversaire afin de gagner (Jeu d'Echecs, Dames, Othello, ...) (voir [Tey12]). Il existe de nombreux programmes (historiques [dLB96] et présentation [Bur03]) implémentant les règles d'Othello ainsi que des intelligences artificielles permettant de jouer contre une machine.

1.1.2 Heuristiques et Algorithmes de force-brute

De nombreuses heuristiques ou stratégies permettent de jouer dans le but de gagner. La stratégie positionnelle (voir [Mac06]), la mobilité (voir [SA04]), la maximisation (voir [CDN98]) par le MiniMax (voir [Tor13]) ou l'AlphaBeta sont des heuristiques et algorithmes différents permettant de chercher des solutions. Ces algorithmes ont été comparés (voir [Che10]) entre eux ce qui nous a permis de choisir l'algorithme AlphaBeta.

Il existe cependant une version alternative de l'algorithme AlphaBeta appelé le ProbCut (voir [Bur95] [Bur97]) que nous n'avons pas implémentée mais qui semble intéressante à explorer.

1.1.3 Algorithmes évolutionnistes et réseaux neuraux

Les algorithmes évolutionnistes consistent à faire évoluer un ensemble de solutions par itération d'un nouvel ensemble de solutions améliorées. Ils sont utilisés dans le but d'optimiser la résolution de problèmes. Les réseaux neuraux sont des modèles de calculs basés sur les réseaux de neurones biologiques de l'être humain. Ils sont représentés par des automates. Nous n'avons pas implémenté ce genre d'algorithme (voir [Leo95]) par manque de temps.

1.2 Étude de l'existant

Depuis les années 80 (voir [d'O02a, dB96] pour un historique), de nombreux programmes Othello français (voir [d'O02c]) et étrangers (voir [d'O02b]) ont été créés et se sont défiés lors de tournois afin de tester leurs performances. De Iago (voir [Ros81]), Bill (voir [LM86]), Hannibal (voir [GP98]) à Logistello (voir [Bur02]), tous ces programmes utilisent des heuristiques et algorithmes qui ont été développés et améliorés au fil des recherches afin de trouver des solutions de manière rapide et efficace. Nos recherches nous ont amenés à trouver des projets autour d'Othello tel que le Jacothellon (voir [CN06]), un Othello en java (voir [SHM05]) et un autre en C++ avec la librairie SFML (voir [Mer10]).

Chapitre 2

Analyse des besoins

Cette partie présente l'analyse des besoins pour le projet. Deux types de besoins seront étudiés, à savoir les besoins fonctionnels et les besoins non fonctionnels.

2.1 Besoins fonctionnels

Les besoins fonctionnels sont des besoins en rapport direct avec les demandes de l'utilisateur. Des diagrammes UML accompagnent ces spécifications afin de simplifier l'analyse (annexes [B.1](#), [B.2](#), [B.3](#)). Les fonctionnalités sont classées par priorité de développement.

2.1.1 Jouer une partie / mettre en pause une partie

L'utilisateur doit pouvoir lancer à tout moment une partie avec l'affichage d'un plateau contenant l'environnement initial, ainsi que faire une pause dans le déroulement de la partie. Le lancement d'une partie ne doit pas prendre plus de 2 secondes. Cette fonctionnalité ne sera pas difficile à implémenter.

Test système Appuyer sur le bouton lance bien le mode Jeu du logiciel.

2.1.2 Réaliser une Intelligence Artificielle (IA)

L'utilisateur doit pouvoir jouer contre une Intelligence Artificielle tout d'abord naïve puis plus tard, si possible, évolutive. L'IA doit pouvoir apporter une solution en un temps imparti par l'utilisateur. Le risque de cette approche est que l'IA peut être très longue à répondre selon l'avancée de la partie. Dans ce cas, l'algorithme le plus adapté sera choisi et un compteur de temps sera ajouté pour empêcher tout débordement sur le temps maximal. Cette fonctionnalité est certainement la plus difficile du projet.

Test comportemental Vérifier que le résultat de la réflexion de l'IA est correct.

Test de performance Exécuter le logiciel en même temps qu'un autre (gourmand en ressources) afin de voir si les résultats obtenus sont identiques sur une même partie. (savoir si l'IA est compétente).

2.1.3 Enregistrer une partie / Relancer une partie

L'utilisateur doit pouvoir enregistrer sa partie et la relancer pour pouvoir la continuer. Cette action est réalisée à travers l'actionnement d'un bouton. Le fichier de sauvegarde doit être créé en moins de 5 secondes et ne doit pas dépasser une taille de 10 Ko. Il sera nécessaire d'implémenter un petit analyseur syntaxique et lexical pour la lecture et l'écriture des fichiers de sauvegarde. Les risques de cette fonctionnalité sont qu'il est possible que la sauvegarde soit interrompue et donc qu'il y ait une perte de données, le fichier étant alors corrompu. Ce risque peut être paré par une protection des données du fichier par l'utilisation de balises ouvrantes et fermantes afin de vérifier l'intégrité des données du fichier. Cette fonctionnalité sera facile à implémenter.

Test système Le fichier de sauvegarde est bien créé et valide. Le plateau affiché correspond bien au fichier en entrée.

Test aléatoire Lancer aléatoirement des enregistrements de la partie.

2.1.4 Recommencer une partie à une certaine position

L'utilisateur doit pouvoir recommencer à jouer à une partie, préalablement lancée et enregistrée, en ayant la possibilité de choisir à partir de quel état du plateau recommencer. C'est-à-dire que si 10 coups ont été joués alors l'utilisateur peut choisir de recommencer à partir du deuxième ou bien du huitième. Ainsi, le fichier de sauvegarde doit contenir un historique complet de la partie en l'état. Le temps de chargement d'un fichier de sauvegarde et la mise en place du plateau à la position souhaitée ne doit pas prendre plus de 10 secondes. En cas de crash de la sauvegarde, le système récupérera l'avant-dernière sauvegarde. Celle-ci a une difficulté d'implémentation normale.

Test unitaire La position choisie correspond bien à celle sauvegardée.

Test négatif Choisir une position hors des bornes de la partie (< 0 ou > au dernier coup).

2.1.5 Possibilité de revenir un coup en arrière

L'utilisateur doit pouvoir, s'il a effectué un déplacement non désiré, revenir en arrière dans le jeu afin de rejouer les coups qu'il désire (mécanisme Défaire/Refaire). L'utilisateur aura le choix parmi tous les coups qu'il avait joués et pourra ainsi revenir en arrière pour tester une stratégie différente. Le temps de modification de l'affichage et de la structure doit être quasi-instantané, une demi-seconde. Une corruption éventuelle d'historique conduit à utiliser l'historique du coup précédent. Il faudra donc garder en mémoire tous les coups

joués et permettre de “revenir en avant”, c'est-à-dire, annuler un retour en arrière. Cette fonctionnalité a une difficulté d'implémentation normale.

Test unitaire La position choisie correspond bien à celle sauvegardée.

Test aux limites Cette fonctionnalité doit être désactivée lorsque qu'il n'y a plus aucun coup précédent le coup actuel.

Test aléatoire Simulation d'un nombre aléatoire de retours en arrière.

2.1.6 Fournir le manuel utilisateur du jeu (interface et règles)

L'utilisateur doit pouvoir accéder à une aide générale sur le fonctionnement et les règles du jeu. Cette aide sera introduite par l'ouverture d'un petit site web disponible en local. L'explication sera lisible et assimilable en moins de 10-15 minutes. Il faudra alors résumer pour mettre l'essentiel et vulgariser les explications afin qu'elle soit compréhensible par tous. Cette fonctionnalité sera facile à réaliser.

Test unitaire Le site web de l'aide s'affiche bien lors de l'appui sur le bouton.

2.1.7 Proposer des suggestions de coups par l'IA

L'utilisateur doit pouvoir demander au logiciel de l'aide pour visualiser le meilleur coup à jouer après analyse par l'IA qui proposera plusieurs niveaux d'aide comme par exemple, le choix entre les deux meilleurs coups à jouer. L'IA doit répondre en un temps acceptable dans l'ordre de 2 à 3 secondes. L'algorithme utilisé dépendra de l'avancement de la partie. Pour que l'algorithme choisi ne prenne pas trop de temps, on ajoutera un compteur de temps. La difficulté de cette fonctionnalité dépendra de la difficulté de l'intégration de l'IA (voir fonctionnalité n°[2.1.2](#)).

Test unitaire Cette fonctionnalité doit toujours nous renvoyer une position correspondant à un coup valide.

Test aux limites Ce bouton n'est activé que lorsque le joueur peut jouer.

2.1.8 Chargement d'un fichier pré-configuré

L'utilisateur doit pouvoir sélectionner un fichier qu'il a composé afin de jouer sur un plateau de sa configuration. Ce fichier doit être écrit de façon interprétable par le logiciel. Le fichier doit être vérifié et chargé puis le plateau qui en résulte doit être affiché en 5 secondes maximum. Le fichier peut-être corrompu donc une vérification de jalons de début et de fin obligatoire ainsi que la présence des informations requises permettra de parer à ce risque. S'il y a corruption du fichier ou une erreur lors de la position des jetons dans le fichier, un message d'erreur s'affichera. Cette fonctionnalité n'est pas dure à implémenter.

Test système Le plateau correspond bien au fichier entré si celui-ci est valide.

2.1.9 Possibilité de choisir la taille de la grille de jeu

L'utilisateur doit pouvoir choisir la taille de sa grille de jeu. C'est-à-dire une grille 5*5, 6*6, ... ou plus. Mais aussi 6*10, ou autre grille rectangulaire. La grille pourra être choisie dans une limite d'au minimum 4x4 jusqu'au maximum 50x50. Le risque de cette fonctionnalité est que l'utilisateur choisisse une grille trop grande ou trop petite pour que le logiciel soit jouable. Il suffira de limiter la taille de la grille et d'obliger l'utilisateur à entrer les bonnes dimensions. Cette fonctionnalité est simple à implémenter.

Test unitaire La grille doit correspondre à la taille demandée.

Test négatif La grille ne doit pas être acceptée si la taille demandée est trop grande ou trop petite.

Test aléatoire Lancer différentes configurations de grilles qui soient dans l'intervalle autorisé.

2.1.10 Menu pour choisir le temps de réflexion de l'IA

Le joueur doit pouvoir choisir si l'IA prendra plus ou moins beaucoup de temps pour le calcul des coups. Ce temps ne devra pas pouvoir dépasser 2 jours. Cette fonctionnalité ne devrait pas poser de problème particulier.

Test système Le temps de réflexion de l'IA doit correspondre à celui donné dans le menu de la partie.

Test aléatoire Entrer des temps aléatoires raisonnables, < 3 secondes, pour le temps de réflexion de l'IA et vérifier que l'algorithme le respecte.

2.1.11 Possibilité d'échanger les joueurs

Le logiciel doit permettre au joueur d'échanger sa place avec l'IA et de pouvoir continuer la partie ainsi. Le temps de réponse du logiciel pour échanger les joueurs ne doit pas dépasser plus de 2 secondes. Il ne sera possible d'échanger les joueurs que lorsque la partie aura été commencée et non terminée. Cette fonctionnalité n'est pas dure à implémenter.

Test unitaire Il est visuellement possible de voir que nos jetons sont échangés avec ceux de notre adversaire.

Test aléatoire Tester pendant une partie l'échange des joueurs à des intervalles aléatoires.

2.2 Besoins non fonctionnels

Les besoins non fonctionnels sont des besoins en rapport direct avec le comportement du logiciel et la gestion du projet.

2.2.1 Besoins comportementaux

Performances

Le logiciel doit pouvoir s'exécuter en 2-3 secondes maximum. L'algorithme force-brute (recherche exhaustive) doit pouvoir calculer une stratégie maximale de 4x4 en 15 minutes maximum (moyenne). Le temps de calcul de l'algorithme force-brute pour une grille 6x6 ou 8x8 étant considérablement plus élevé, l'utilisation d'algorithmes plus évolués devra être étudiée et expérimentée. Le logiciel ne doit pas dépasser une utilisation de la mémoire supérieure à 2Go.

Test système Exécuter le logiciel sur différents types de machines de puissances différentes : petites machines (netBook, notebook), machines de bureau (PC de salon, PC portable moyen, c'est-à-dire avec un dual core + 4go de ram maxi), machines de jeu (Core i7, 16GO de ram, ...).

Fiabilité

Le logiciel doit implémenter des sauvegardes régulières pour permettre à l'utilisateur de pouvoir reprendre une partie coupée. Ces sauvegardes doivent être intègres. Le logiciel doit enregistrer au fur et à mesure les coups joués en tant que position (i,j) dans un fichier annexe afin de garder une trace des parties réalisées. Afin de permettre à l'utilisateur de récupérer une partie en cours avant un crash quelconque de l'ordinateur ou une fermeture de l'application, la partie doit être régulièrement enregistrée afin de pouvoir être relancée.

Test système Fermer l'application en plein milieu d'une partie, pendant le chargement d'une partie, pendant la réflexion de l'IA.

Test aléatoire Ouvrir les fichiers de sauvegarde afin de vérifier qu'ils ne soient pas corrompus et intègres.

Facilité d'utilisation

L'interface graphique sera composée de menus, de boutons et d'une grille représentant le plateau de jeu (figure A.1 de l'annexe A). Elle sera simple et intuitive (autres prototypes de l'interface dans l'annexe A).

Test d'utilisation Faire jouer des personnes inconnues au projet, des personnes ne connaissant pas les règles du jeu Othello, des enfants aux personnes âgées.

Domaine d'action

Il y a deux sortes d'utilisateurs qui seront concernées par le jeu :

- l'utilisateur “tout public”, qui a pour but de simplement jouer au jeu.

- l'utilisateur chercheur / développeur qui sera intéressé pour tester l'efficacité de l'IA, trouver des stratégies gagnantes.

Le logiciel peut prendre des fichiers en paramètre qui permettront de commencer une partie sauvegardée ou de créer une situation initiale différente. Un fichier en sortie sera créé à la suite d'une partie remportée pour récupérer l'historique.

Test d'utilisation Complément du test de Facilité d'utilisation. Faire jouer des scientifiques pouvant utiliser les résultats du programme à des fins professionnelles.

Portabilité

Le logiciel sera fonctionnel pour les plates-formes supportant Java permettant une portabilité sur les machines Unix et Windows.

Test système Ce test se fait en complément du test de performance, tester le logiciel sur différents types de machines supportant la JVM (Windows, Mac Os, Linux).

2.2.2 Besoins organisationnels

Standards et processus de développement

Méthode d'organisation : Une réunion par semaine avec le client ainsi qu'une réunion hebdomadaire avec le chargé de TD (en concordance avec le planning imposé par le responsable PdP). Nous avons prévu des créneaux horaires pour travailler en groupe en plus du travail personnel effectué par chacun.

Outils de spécification : Pour la réalisation de ce projet nous avons utilisé plusieurs outils orientés travail en équipe, gestion de projet et développement.

- Pour optimiser le travail de groupe nous avons choisi d'utiliser **Git**. C'est un service web qui permet de gérer le versionnement de fichier. On va ainsi pouvoir suivre l'évolution d'un projet et surtout travailler à plusieurs sur les mêmes fichiers. Ce type de service enregistre au fur et à mesure les différentes versions des fichiers, ce qui donne la possibilité de pouvoir revenir à une version antérieure. Nous avons préféré **GIT** à **SVN** car il repose sur le principe du "fork". Ce principe permet à n'importe quel utilisateur de reprendre un projet et de le gérer après l'avoir "forké" (création d'un embranchement du projet qui est une copie de l'originale). Cela permet à un projet de "vivre" en pouvant être repris par n'importe qui de la communauté Open Source.

- Cependant, nous avons quand même utilisé **SVN**, autre logiciel de gestion de versions. Pourquoi avons nous, en plus de Git, utilisé SVN ? Car nous avons comme directive de mettre sur le serveur SVN de l'université (serveur Savane) l'intégralité de nos fichiers sources (latex pour les documents, et java pour le code) afin qu'ils soient disponibles pour les futurs étudiants.
- Pour une meilleure segmentation du code, chaque module a été implémenté dans un nouveau projet qui lui est spécifique. Ainsi, nous avons pu séparer les différents modules facilitant le partage du travail dans le groupe. Le module du jeu est lié à tous les autres modules par l'intermédiaire de JAR (Java ARchive qui contient le code source et certains fichiers utiles). A chaque version stable d'un module, nous exportons un jar contenant les fichiers sources et les commentaires. Ils sont regroupés dans un dossier, ce qui nous permet d'avoir à tout moment une archive fonctionnelle pour chaque module. Pour réaliser cette gestion des modules et le développement du code, nous avons choisi d'utiliser l'IDE **Eclipse**. Il permet également de lier les projets directement plutôt que de passer par la génération de JAR. Nous avons choisi de ne pas utiliser cette fonctionnalité car si un des modules ne fonctionne pas alors la répercussion est immédiate et empêche le fonctionnement de tous les projets qui lui sont liés.
- **Microsoft Project** est un logiciel de gestion de projet édité par Microsoft. Il permet de réaliser un planning de travail prévisionnel, ainsi que de vérifier le bon déroulement du projet en fonction de ce planning. Il possède de nombreuses autres fonctionnalités que nous n'avons pas utilisées telles que l'aide à la gestion des ressources et du budget alloué au projet. Ce logiciel nous a servi notamment pour établir un planning de travail ainsi qu'un diagramme de Gantt que vous pouvez retrouver dans la partie suivante de ce rapport.
- **UML Designer** est un logiciel permettant de modéliser graphiquement les diagrammes UML 2.4. Il est développé par la société Obeo (<http://www.obeo.fr/>) sur une base d'Eclipse 4.3. Nous avons choisi ce logiciel plutôt qu'un autre (type Umbrello, Bouml,...) car il est relativement simple d'utilisation, stable, génère des diagrammes lisibles, et permet aussi de générer les fichiers sources correspondant aux diagrammes.
- Nous avons choisi d'utiliser le langage de programmation **Java** car les programmes Java sont portables sur plusieurs systèmes d'exploitation (**UNIX**, **Windows**, **OS X**, **Linux**). Pour structurer les fichiers des différentes sauvegardes, nous avons choisi d'utiliser le langage **XML**. Ainsi nous avons architecturé avec des balises les structures de données à conserver.

Environnements et systèmes d'exploitation : Les systèmes d'exploitation qui seront utilisés pendant le développement sont : Windows, Linux et OS X. Le logiciel sera développé en Java 1.7 avec l'IDE Eclipse. Développer sur différents environnements nous permettra d'effectuer des tests constants sur la portabilité du logiciel.

Gestion du temps

Nous avons décidé de représenter notre gestion du temps de travail grâce à un diagramme de Gantt (figure 2.1) et un tableau descriptif de nos tâches (tableau 2.1).

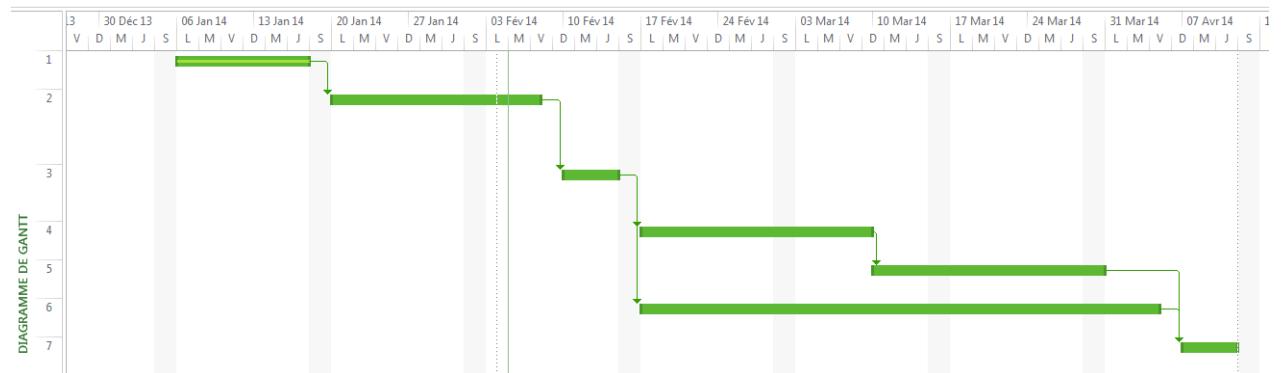


FIGURE 2.1 – Diagramme de GANTT représentant la gestion du temps pour le projet.

Numéro de Tâche	Nom de la tâche	Durée	Début	Fin	Prédécesseurs
1	Bibliographie + Analyse de l'existant	10 jours	Lun 06/01/14	Ven 17/01/14	
2	Rédaction du Cahier des Besoins + Etablissement des Tests	15 jours	Lun 20/01/14	Ven 07/02/14	1
3	Mise en place de l'architecture Logicielle	5 jours	Lun 10/02/14	Ven 14/02/14	2
4	Développement logiciel + Tests	16 jours	Lun 17/02/14	Dim 09/03/14	3
5	Tests (Débug logiciel)	16 jours	Lun 10/02/14	Dim 30/03/14	4
6	Rédaction du mémoire de projet	35 jours	Lun 17/02/14	Ven 04/04/14	3
7	Préparation et entraînement pour la soutenance	5 jours	Lun 07/04/14	Ven 11/04/14	5;6

Tableau 2.1 – Tableau descriptif des tâches

1. L'analyse de l'existant permet de commencer le projet, et de réaliser une ébauche de la bibliographie qui sera complétée (modifiée) lors de la rédaction du mémoire final de projet.
2. Rédaction du cahier des Besoins (Fonctionnels et Non-fonctionnels). Etablissement des tests à réaliser pendant et après la phase de développement.
3. Réalisation de l'architecture logicielle. Modélisation à l'aide du logiciel UML Designer.
4. Le développement logiciel contient les tâches suivantes :

- Réalisation de l'IHM
- Réalisation de la gestion des fichiers (pour la sauvegarde d'une partie)
- Réalisation de l'Intelligence Artificielle du jeu

Durant toute la durée de développement logiciel, les tests sont réalisés.

5. Réalisation des tests sur le logiciel afin de traquer les bugs existants.
6. La rédaction du mémoire de projet est répartie sur la durée du développement et des tests. La durée estimée est d'une semaine pour les quatre membres du projet.
7. Cette partie du projet consiste à la réalisation d'un support de présentation (Power Point). Nous allons aussi réaliser des entraînements de soutenance.

2.2.3 Besoins externes

Contraintes d'interopérabilité

Aucune contrainte.

Contraintes légales

Aucune contrainte.

Contraintes éthiques

Aucune contrainte.

Chapitre 3

Architecture et description du logiciel

Après l'analyse de l'existant, nous avons décidé de développer le logiciel entièrement sans réutiliser un logiciel pré-existant. En effet, la reprise d'un code réalisé par une autre personne est aussi longue que de le développer soi-même. De plus, le logiciel doit répondre à de nombreux critères tel que la maintenabilité du code mais également avoir une architecture optimale, qui n'auraient pas forcément été présents dans le code repris. Cependant, nous nous sommes inspirés des différents articles portant sur des heuristiques pour l'IA.

3.1 Architecture du projet

La création d'un jeu comme Othello est considérable, pour faciliter le développement nous avons décidé de segmenter en briques fonctionnelles le projet. Le principe de la segmentation (voir figure 3.1) en module est qu'ils sont tous indépendants et testables séparément. Nous comptons six segments : le logiciel, l'intelligence artificielle, le gestionnaire de fichiers, l'éditeur de plateau, le BenchMark et le gestionnaire de temps.

Les blocs IA et logiciel sont les plus importants et conséquents. Le second intérêt de cette architecture est que chaque module peut être remplacé par un autre plus ou moins équivalent sans compromettre l'intégrité et le bon fonctionnement du logiciel. Par exemple, pour le bloc de l'IA, si une personne de la communauté souhaite l'améliorer, il peut tout simplement coder le module IA et l'intégrer au logiciel pour le tester.

Nous avons également ajouté un module supplémentaire pour nous faciliter le développement et la compréhension d'erreurs pour un utilisateur "scientifique". Le module "ErrorLog" permet d'écrire dans le même fichier ("log.txt") les différentes erreurs rencontrées pendant l'utilisation actuelle du logiciel.

Développant en JAVA, les différents modules sont liés par le biais d'interfaces. Ainsi, chaque brique présentera une interface définie et dûment commentée qui représentera le comportement du module (sauf pour le BenchMark et ErrorLog qui ne possèdent qu'une à deux méthodes).

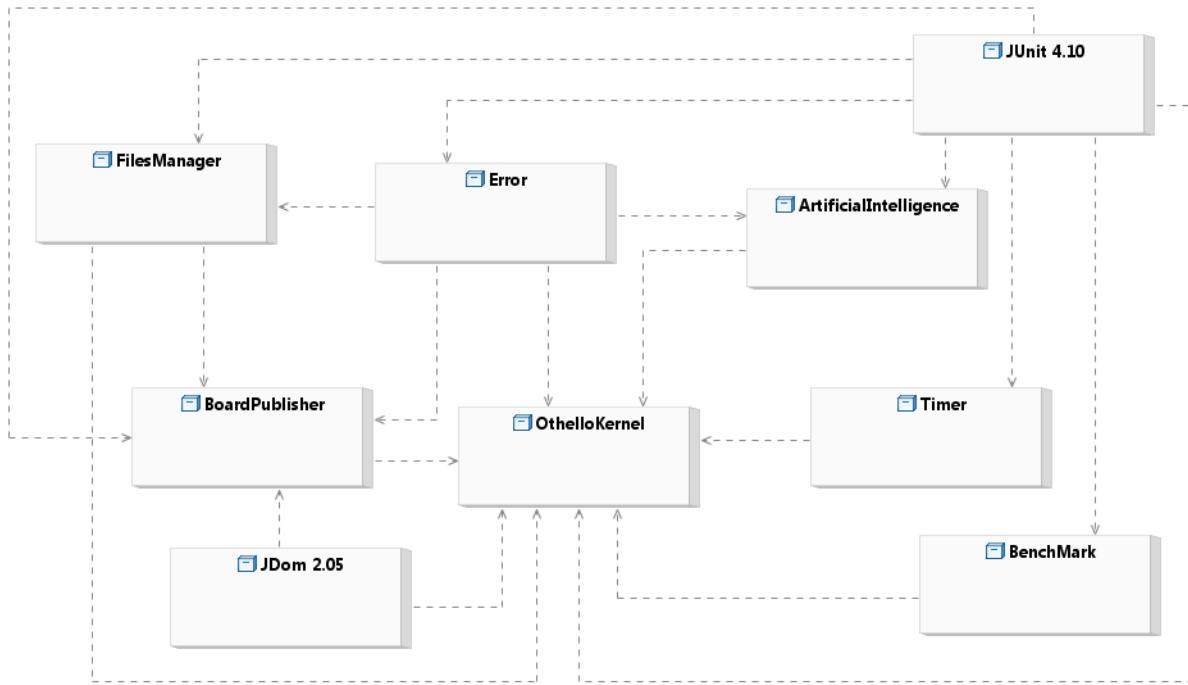


FIGURE 3.1 – Diagramme de dépendances

3.1.1 Logiciel

C'est le module principal qui va gérer le fonctionnement complet du jeu. Il va intégrer tous les autres modules et lancer l'interface graphique permettant à l'utilisateur de jouer. Il gère l'accès et l'utilisation de ces derniers lors du déroulement du jeu.

Le logiciel (voir figures B.4) est lui-même composé de trois modules principaux : l'interface, les données et le contrôleur. Ce dernier gère l'intégralité du projet. Il exploite tous les modules au bon moment pour chaque fonctionnalité le nécessitant.

De plus, il était prévu à la base de faire une vue en mode "shell" (sur le terminal) en plus de la vue graphique. Ainsi, nous avons géré l'intégralité du comportement général du jeu dans une classe abstraite. Pour chaque affichage différent du jeu, il suffit de créer un contrôleur qui spécialise celui-ci. C'est le cas de notre interface graphique qui est gérée par la classe "GameControllerGraphical" implémentant la classe abstraite "GameController". Ainsi pour gérer un affichage type "Shell", il suffirait de créer un fichier "GameController-Shell", en suivant l'exemple de "GameControllerGraphical", qui afficherait le jeu dans le terminal.

3.1.2 Intelligences Artificielles

Ce module (voir figure 3.2) est la partie la plus compliquée et la plus importante à réaliser dans notre projet. En effet, il se compose de différentes IA qui ont des algorithmes distincts, chacune avec sa propre stratégie correspondant à un niveau de difficulté.

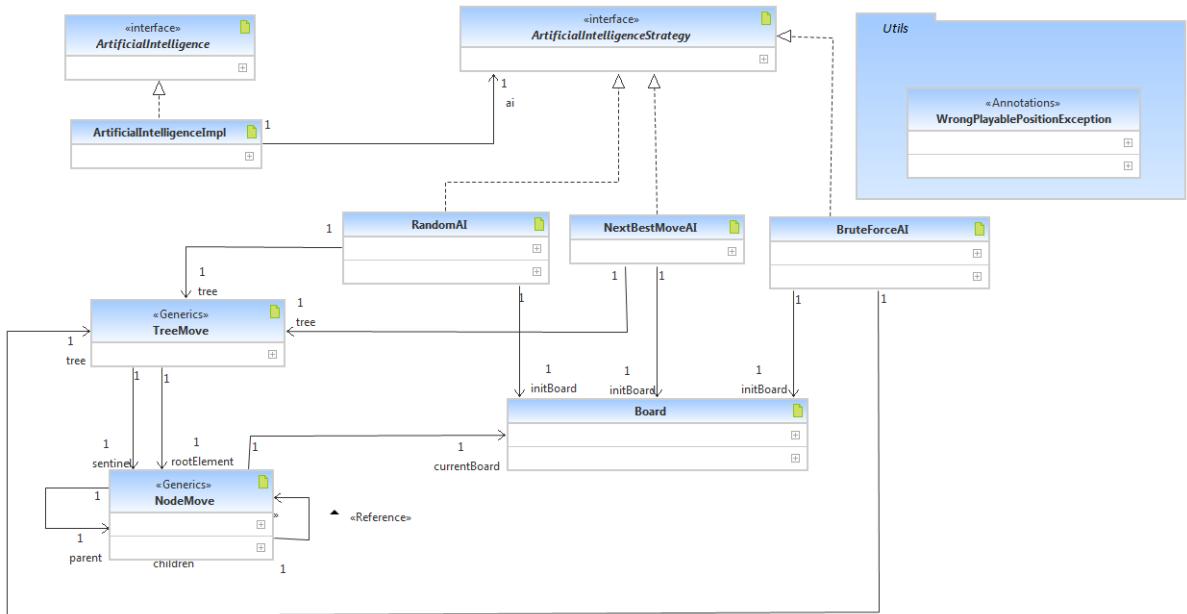


FIGURE 3.2 – Diagramme UML de classe du module d’intelligence artificielle

Pour les différents calculs des algorithmes, un pattern state avait été mis en place afin de représenter le plateau. Cependant, ce pattern rend les structures lourdes à allouer et à désallouer par la JVM ce qui va entraîner des surcharges de piles, des erreurs de désallocation du Garbage Collector de la JVM, ... Nous avons donc décidé de rendre la structure plus simple mais tout aussi efficace en utilisant simplement une énumération de pion pour représenter une case.

De plus, nous avons créé deux classes nous permettant de manipuler des arbres génériques. La généricité de ces arbres permet la réutilisation de ces classes par d’autres développeurs.

3.1.3 Gestionnaire de Fichier

Il permet de gérer et surtout de sécuriser les actions de lecture et d’écriture pour les différents accès aux fichiers. Il est essentiellement utilisé pour les actions de sauvegarde et de chargement des parties.

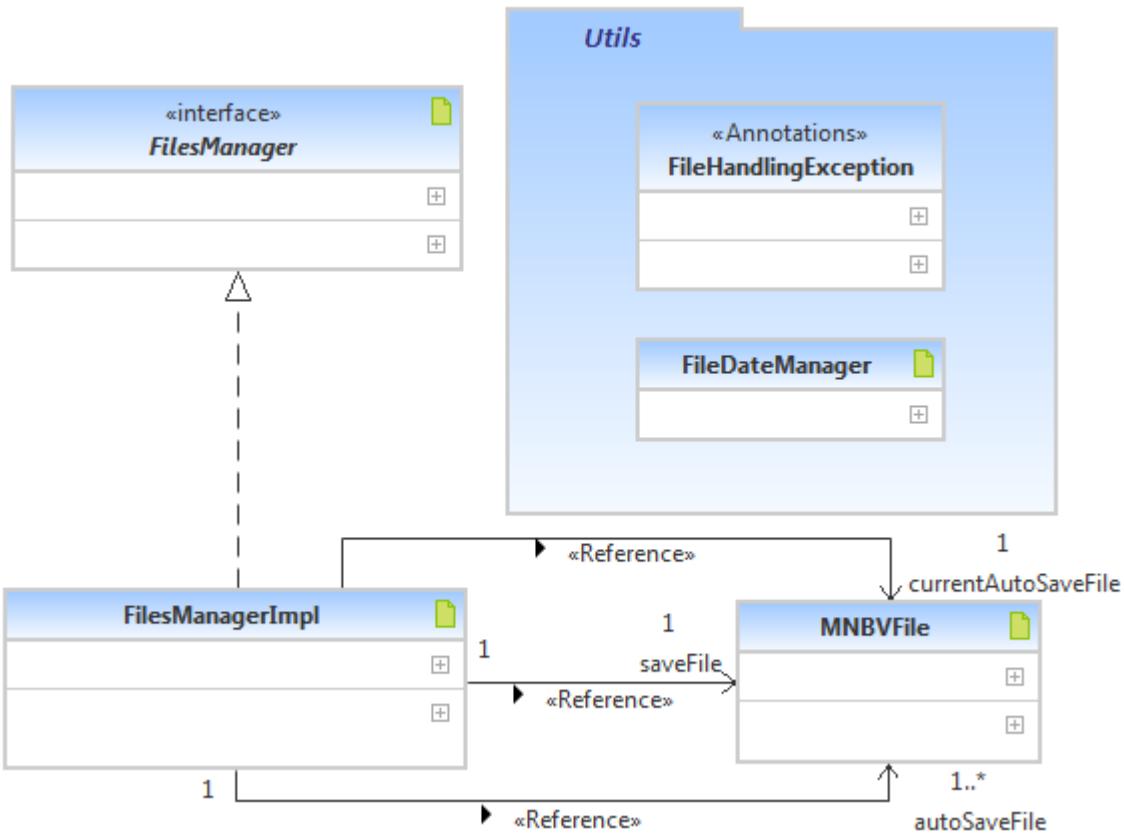


FIGURE 3.3 – Diagramme UML de classe du module de gestion de fichier

3.1.4 Editeur de Plateau

Ce module permet à l'utilisateur de générer un plateau qu'il pourra ensuite charger dans le logiciel pour jouer. Le plateau est personnalisable : la taille de la grille, la position et le nombre de pions de départ. Cette fonctionnalité est lancée dans la console système à partir du menu du jeu.

L'éditeur possède une partie correspondant à un modèle à savoir les classes Board et Player. Ces classes servent à récupérer les données utilisateur. Celles-ci sont ensuite utilisées par les classes LoadBoardFile et GenerateXML. La classe LoadBoardFile permet de récupérer une grille préconfigurée (fichier.grd). GenerateXML permet grâce à la bibliothèque JDOM 2.05 (<http://www.jdom.org/>) de générer un document xml bien formaté. Ce document correspond à notre fichier de sauvegarde.

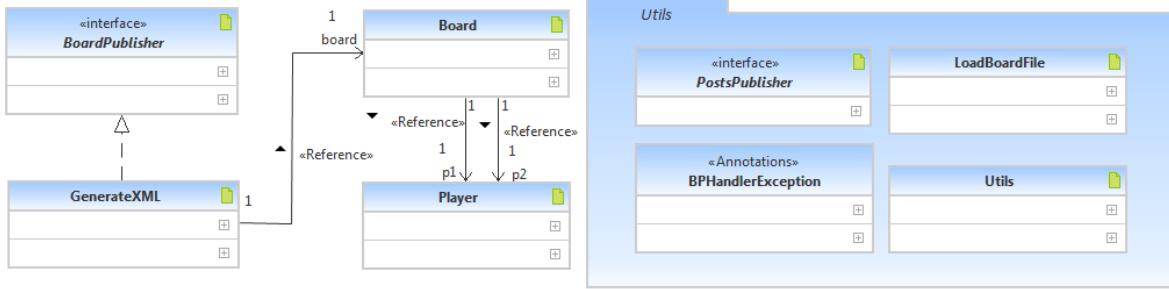


FIGURE 3.4 – Diagramme de classe du module éditeur de plateau

3.1.5 BenchMark

Cette brique permet de lancer une série d’algorithmes compliqués et gourmands en ressources qui prendra un temps plus ou moins long selon la machine. A partir de ce temps, on peut connaître la puissance de l’ordinateur qui exécute le programme. Par manque de temps, nous n’avons pas pu implémenter l’utilisation de ce module dans le logiciel. Cela nous aurait permis de modifier l’utilisation de certains algorithmes de l’IA en fonction des capacités disponibles.

Ce module est basé sur un travail réalisé par les scientifiques Roldan Pozo et Bruce Miller : SciMark 2.0. Ce logiciel, libre de droit, réalise plusieurs calculs et établit un score selon les résultats. Par manque de temps, nous n’avons pas pu nous pencher d’avantage sur les calculs utilisés.

Ainsi, notre brique reprend ce module et permet de faire une liaison entre celui-ci et le logiciel qui va l’utiliser. Pour cela, nous utilisons une interface qui va informer le logiciel de l’état d’avancement du BenchMark, celui-ci durant environ 30 secondes.

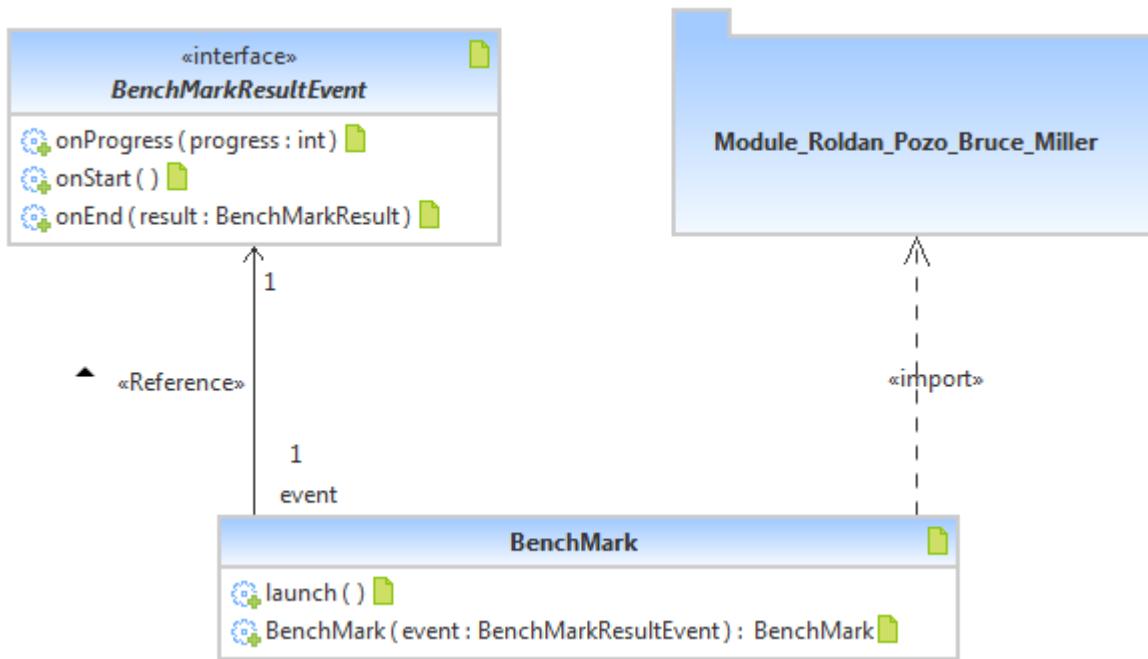


FIGURE 3.5 – Diagramme UML de classe du module de BenchMark

3.1.6 Gestionnaire de Temps

Cette partie 3.6 propose les fonctionnalités de chronomètre et de minuteur. Celles-ci sont utilisées pour limiter le temps de calcul de l'IA et minuter le jeu.

Le minuteur déclenche une action dès que le temps passé en paramètre de la méthode est écoulé. Cet évènement est récupéré par le logiciel qui peut ensuite réaliser le traitement associé.

Le chronomètre fonctionne à partir de l'horloge système, plus particulièrement sur le temps écoulé en millisecondes. Après déclenchement et jusqu'à sa réinitialisation, le gestionnaire renvoie la différence du temps écoulé par rapport au temps du lancement.

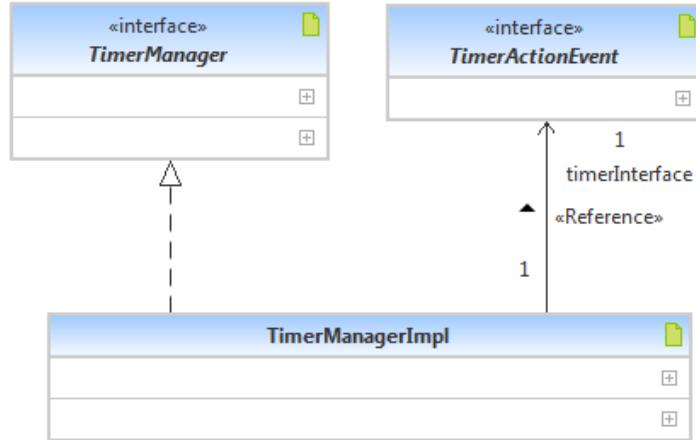


FIGURE 3.6 – Diagramme de classe du module de gestion de temps

3.2 Design patterns utilisés

Les design patterns (patron de conception, en français) ont été créés afin de remédier à certains problèmes de conception récurrents. Chacun d'eux répond à un problème précis et permet d'améliorer la maintenabilité et la réutilisation de notre code.

Notre formation à l'approche objet en Java nous a permis de voir plusieurs de ces patterns. Nous les avons implémentés dans nos différents modules dès que l'architecture si prêtait.

3.2.1 State

Le principe de ce pattern est de pouvoir changer le comportement d'un objet sans modifier son instance, ainsi que de faciliter l'ajout de nouveaux comportements. Nous l'avons utilisé afin de gérer les pièces de l'othellier.

En effet, une pièce peut être de différentes couleurs : noire, blanche ou vide (équivalent à aucune pièce). Le principal avantage de ce pattern, pour les pièces, est de pouvoir facilement changer sa couleur ou d'ajouter une nouvelle couleur si on le souhaite.

Cependant, cette implémentation a ses désavantages. Chaque changement de couleur demande une nouvelle instanciation de comportement et donc, une partie de la mémoire. C'est pour cela que pour les calculs lourds des intelligences artificielles, ce pattern n'a pas été utilisé.

Nous avons également utilisé ce pattern sur la classe **Joueur**. Celle-ci contient deux comportements : **Machine** et **Humain** correspondant à un joueur humain décidant son coup et à l'**IA** qui décide son propre coup. Nous avons utilisé ce pattern dans le cas d'un comportement différent qui pourrait être demandé par le client pour améliorer l'expérience utilisateur du jeu ou dans une démarche scientifique de recherches. Prenons, par exemple, une fonctionnalité où à partir d'une certaine position dans la partie, un joueur **Humain**

déciderait de laisser l'IA jouer à sa place. Ainsi, en changeant simplement le type de joueur, l'IA prendrait la main. Ceci est possible car à chaque pièce posée, le contrôleur général gère l'évènement associé (automatique pour l'IA et sur clique d'une case pour l'Humain) selon le type du joueur.

3.2.2 Abstract Factory

Le pattern Abstract Factory (Fabrique Abstraite) permet de fournir une interface unique et simple d'utilisation pour créer les objets d'une même famille sans avoir à connaître la classe à instancier. Nous avons utilisé ce pattern dans le module Othello Kernel notamment sur les objets “package Model”.

Notre fabrique abstraite (classe `AbstractFactory`) permet de gérer plusieurs fabriques, à savoir celles de “Piece”, de “Player”, “Board”, “GameSettings”, et de “Save/RestoreGame”. Chacune d'elle permet de gérer un type d'objet précis. Ainsi, nous avons plusieurs “factory” ; une pour chaque classe citée ci-dessus. Toutes ces “factory” sont elles mêmes interfacées par une classe abstraite “`AbstractFactory`” et récupérables par l'intermédiaire de la classe “`FactoryProducer`”.

Ce pattern présente, malgré tout, plusieurs inconvénients. Premièrement, il est compliqué à mettre en place car chaque fabrique d'objets à besoin d'une classe et d'une interface. De plus, toutes les “factory”, quel que soit leur comportement, doivent implémenter les mêmes méthodes car elles interfacent toute la même classe abstraite. Ainsi, nous avons sécurisé chaque comportement inapproprié pour chaque “factory” par un système d'exceptions. Finalement, chaque fabrique étant liée à la même classe abstraite, la modification d'une signature de méthode se répercute sur toutes les “factory” entraînant des modifications profondes.

Malgré cette mise en place relativement lourde, ce pattern est pratique notamment lors de mises en place d'évolutions du logiciel, de maintenance, pour sa facilité d'ajout de constructeurs et également pour segmenter le code.

3.2.3 Observer

Le pattern Observer permet de réduire les dépendances d'objet mais également de gérer des évènements déclenchés par un objet observé et suivi par un observateur. Il est indispensable pour nous, pour la gestion de l'affichage de l'othellier.

En effet, à chaque modification de l'othellier celle-ci doit être représentée visuellement sur la fenêtre utilisateur. Ainsi, la zone de dessin (`GameCanvas`), représentant l'othellier sous forme de grille, espionne le modèle contenant l'othellier et affiche la grille à chaque modification : ajout d'une pièce, calcul des pièces jouables ...

3.2.4 Model-View-Controller

Modèle-Vue-Contrôleur permet de séparer dans une application interactive les différents composants. La vue permet l'interaction avec les utilisateurs, le modèle contient les données

et le contrôleur gère les évènements utilisateurs et la modification de données.

Ce pattern utilise également le patron “Observer”. En effet, les données du modèle sont liées à la vue afin que celles-ci reflètent les changements à chaque modification.

Ce patron a été utilisé pour gérer le jeu Othello. Nous avons séparé le jeu en trois packages correspondant aux modules MVC. Les accès entre les modules Vue-Contrôleur et Vue-Modèle sont entièrement interfacés pour bien segmenter le code.

3.2.5 Singleton

Ce design pattern permet de restreindre l’instanciation d’une classe à un unique objet. Nous l’avons utilisé à plusieurs reprises dans le projet.

Toutes les diverses classes implémentant les “factory” suivent ce pattern afin de limiter l’espace mémoire utilisé. De plus, il n’est pas nécessaire d’avoir plusieurs objets pour une même factory dans notre projet.

Ce pattern a également été utilisé pour la classe Application afin d’empêcher la multi-instanciation. En effet, le but de cette classe est d’être unique/”constante” pour tout le logiciel afin que chaque classe ayant besoin de ces informations puissent y accéder sans dupliquer de données.

Les contrôleurs du logiciel se doivent d’être uniques pour ne pas avoir de répétition d’action pour un évènement donné. Du coup, nous les avons implementés en tant que Singleton par mesure de sécurité.

Lors des tests de performance, nous nous sommes rendus compte que nous avions un important problème d’optimisation mémoire. En effet, à chaque changement d’état d’une pièce, nous faisons une nouvelle instanciation correspondant à la nouvelle couleur : blanche ou noire. Afin d’améliorer nos performances mémoires, la couleur noire/blanche étant la même pour toutes les pièces, nous avons changé les classes codant les couleurs ”Blanche” et ”Noire” pour qu’elles suivent le pattern Singleton. Ainsi, avec une unique intanciation des couleurs nous utilisons beaucoup moins de mémoire.

3.2.6 Strategy

Ce patron de conception permet de modifier dynamiquement les algorithmes utilisés. Le but étant de changer de comportement facilement selon l’environnement. Le principe repose sur l’encapsulation de ces algorithmes pour les rendre interchangeables selon les besoins.

Celui-ci a été utilisé pour la gestion des difficultés de l’IA. En effet, selon la difficulté, l’IA n’utilise pas les mêmes algorithmes : elle change de stratégie selon la difficulté choisie.

Par exemple, en mode facile le calcul de position est basé sur un algorithme aléatoire alors qu’en mode difficile celui-ci est basé sur un algorithme brute force.

3.3 Heuristiques et Algorithmes de recherches de solution

Il existe beaucoup de stratégies (voir [Mac06], [SA04]) permettant de jouer à Othello. Chacune a ses propres spécificités comme par exemple, maximiser le score des pions, empêcher le joueur d'encercler l'autre joueur, donner le moins de coups possibles à l'adversaire,... Nous avons implémenté la stratégie positionnelle, la mobilité, et la maximisation.

3.3.1 Stratégie positionnelle

Le principe de la stratégie positionnelle (voir [Mac06]) est de donner des scores aux cases vides du plateau en fonction de leur stabilité, semi-stabilité et de leur instabilité. Une case est dite stable ou définitive si elle ne peut être reprise par l'adversaire lorsqu'on l'aura jouée. Une case est dite instable lorsqu'on est sûr qu'elle pourra être reprise par le joueur adverse. Et enfin, une case est dite semi-stable lorsque la probabilité que le joueur adverse puisse la reprendre est moyenne. Ainsi, on peut déjà classer quelques cases comme par exemple les coins. Les coins sont stables puisque une fois pris, ils ne peuvent être repris par l'adversaire. La case en diagonale directe de chaque coin est appelée la case X. Les cases X sont instables. De même que les cases C qui sont les 2 cases adjacentes au coin sur les bords du plateau.

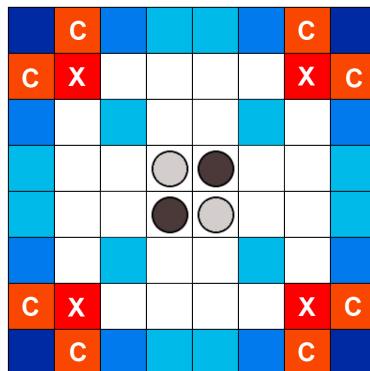


FIGURE 3.7 – Grille classant les cases.

Sur le schéma 3.7, plus la case vire vers le rouge, plus il faut éviter de la jouer. Plus la case est bleu, plus il est intéressant de la jouer. Ainsi, on peut créer une matrice évaluant tous les types de cases.

30	-20	10	5	5	10	-20	30
-20	-25					-25	-20
10		5			5		10
5			○	●			5
5			●	○			5
10		5			5		10
-20	-25					-25	-20
30	-20	10	5	5	10	-20	30

FIGURE 3.8 – Grille évaluant une partie de toutes les cases de l’othellier.

Cette matrice (figure 3.8) est alors adaptée à la taille de la grille de jeu.

Lorsque certaines configurations permettent de capturer sans crainte une position instable, celle-ci se transforme en position stable après recalcule de la matrice.

3.3.2 Mobilité

Le principe de la mobilité (voir [SA04]) est de faire en sorte que l’adversaire ait le moins de coups possibles. Si l’adversaire a beaucoup de coups possibles, il aura certainement une plus grande probabilité de jouer un bon coup. Si l’adversaire a peu de coups possibles, il n’aura pas vraiment le choix et peut se retrouver avec des coups qui le forceront à donner facilement des pions à son adversaire. Cette stratégie est très efficace en début et en milieu de partie car elle permet de contrer toutes stratégies d’encerclement. De plus, le nombre de pièces appartenant à chaque joueur n’est pas une indication fiable pour vaincre en début et en milieu de partie donc, il faut se concentrer sur le fait de diminuer le nombre de coup du joueur afin d’avoir un net avantage lorsqu’on passera à une stratégie de maximisation...

3.3.3 Maximisation

La maximisation est une stratégie dont le principe est de capturer le plus de pièces possibles afin d’avoir le meilleur score. Cette stratégie est la meilleure en apparence puisqu’elle représente l’objectif final du jeu, à savoir obtenir plus de pions que son adversaire. Ceci est vrai si on calcule l’arbre entier de partie représentant toutes les parties possibles du jeu à partir de tous les coups jouables de toutes les positions de plateau. C’est ce qui est plus communément appelé un algorithme "brute force" ou force brute. Cet arbre donne alors le déroulement d’une partie coup après coup jusqu’à atteindre la fin, les feuilles de l’arbre.

MiniMax : parcours exhaustif de l’arbre des parties

L’algorithme MiniMax (voir [CDN98]) permet de parcourir exhaustivement l’arbre de partie du jeu et de décrire des chemins gagnants pour les deux joueurs. Pour cela, il calcule

tous les enchaînements de coup possible à partir de la position de base du jeu. Une fois arrivée à une racine, il prend le score du joueur Max et le remonte. S'il y a plusieurs branches au niveau d'un noeud, le noeud prendra le score maximal des branches si c'est le tour du joueur Max (joueur qui maximise le score) ou prendra le score minimal si c'est le tour du joueur Min (joueur qui minimise le score).

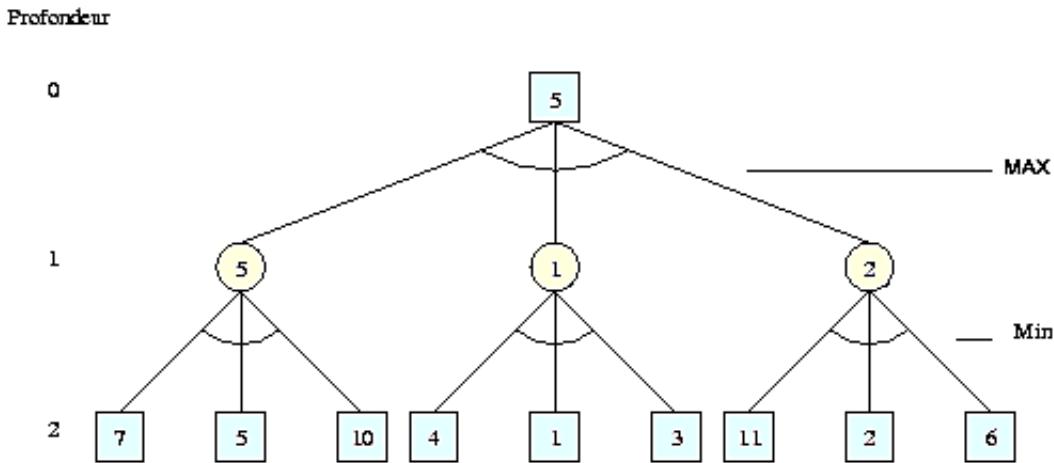


FIGURE 3.9 – Arbre MiniMax de profondeur 2.

L'arbre de partie pouvant être tellement grand qu'il est impossible pour un ordinateur à l'heure actuelle de calculer et de sauvegarder un arbre de partie entier pour une grille d'Othello 8x8 en position initiale de début de jeu. C'est pour cela qu'il est nécessaire de fixer une profondeur à ne pas dépasser. Ainsi, il est d'usage de lancer l'algorithme en fin de partie afin d'obtenir un arbre plus petit et surtout de calculer une stratégie potentiellement gagnante pour le joueur ayant le dernier coup. Si cette stratégie est utilisée en début de partie ou en milieu, elle sera facilement battue par une stratégie positionnelle ou de mobilité puisque l'arbre ne pourra en aucun cas assurer une victoire du joueur sur le long terme.

L'algorithme en pseudo-code est consultable en annexe [C.1](#).

Il existe cependant plusieurs améliorations de l'algorithme MiniMax permettant d'élaguer des branches de l'arbre, l'allégeant quelque peu.

L'élagage Alpha-Beta

L'élagage Alpha-Beta (voir [\[CDN98\]](#)) reprend le principe de l'algorithme MiniMax en ajoutant des bornes alpha et beta afin d'éviter l'exploration de branches inutiles de l'arbre. En effet, il suffit de passer aux noeuds les valeurs actuelles de ces bornes qui changent de valeur au fur et à mesure de la recherche. Alpha représente pour un noeud MIN n, la plus grande valeur connue pour un noeud MAX ancêtre de n. Beta représente pour un noeud MAX n, la plus grande valeur connue pour un noeud MIN ancêtre de n.

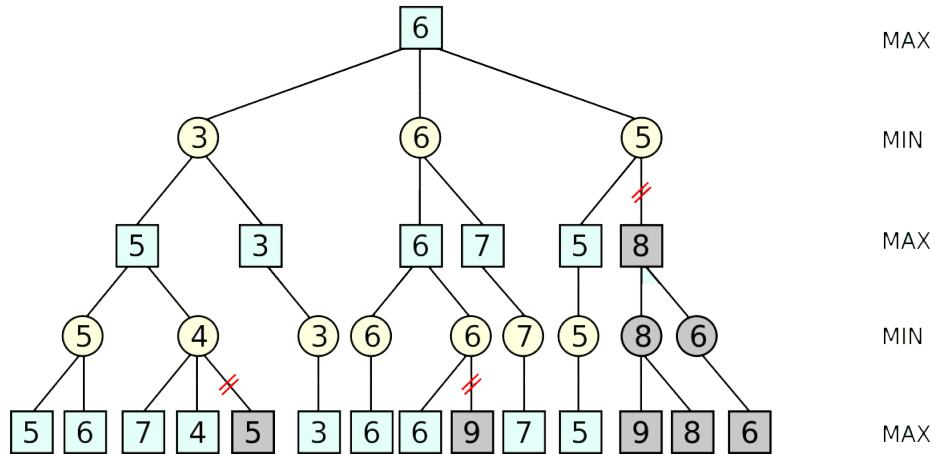


FIGURE 3.10 – Arbre Alpha-Beta de profondeur 4.

L'algorithme utilise le début de la recherche de solution afin de borner les valeurs possibles de la racine. On élague ainsi des sous-arbres entiers de manière non risquée.

L'algorithme en pseudo-code est consultable en annexe [C.2](#).

Nous avons implémenté trois autres versions un peu plus efficace pour cette algorithme :

- La version NegaMax est une version conventionnant l'utilisation de l'algorithme. Elle permet d'éviter la distinction entre les deux joueurs Max et Min et simplifie ainsi le code.
- La version fail soft alpha-beta utilise un nouveau pivot dans la recherche afin de rajouter une borne current permettant d'accélérer la recherche.
- La version NegaScout ou appelé également Principal Variation Search effectue des pré-recherches sur l'arbre en utilisant un intervalle plus petit (α , $\alpha + 1$ à la place d' α , $\alpha + 1$, β en début d'algorithme) afin d'élaguer davantage de branches inutile.

Chapitre 4

Fonctionnement et tests

A ce jour, nous avons développé l'intégralité des besoins fonctionnels. Néanmoins, nous aurions pu, si nous avions eu plus de temps, ajouter des optimisations, des améliorations, supprimer davantage de bugs.

4.1 Etat du projet

Comme décrit dans la partie 3 de ce rapport (Architecture et description du logiciel), nous avons découpé le projet en différents modules que nous allons décrire.

4.1.1 Logiciel

Au lancement du jeu (figure 4.1), quelques réglages sont à faire avant de pouvoir jouer. Dans un premier temps, il faut indiquer la taille de la grille (le maximum est 50*50). Puis, lancer le BenchMark (explication ci-dessous) et indiquer la difficulté souhaitée. Enfin, préciser si l'on joue entre joueurs ou un joueur contre l'IA.

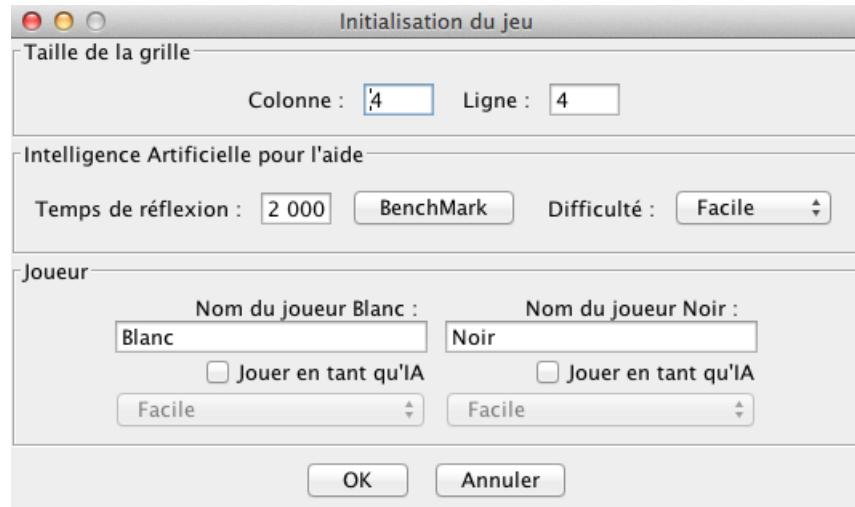


FIGURE 4.1 – Initialisation du jeu

Le Benchmark (figure 4.2) permet d'évaluer les performances de la machine. En fonction du temps de réflexion obtenu, on en déduit le temps qu'il faudra à l'IA pour jouer. Celui-ci ne permet pas, actuellement, de récupérer le temps qu'il faudrait associé à l'IA mais uniquement l'indice de performance de la machine.

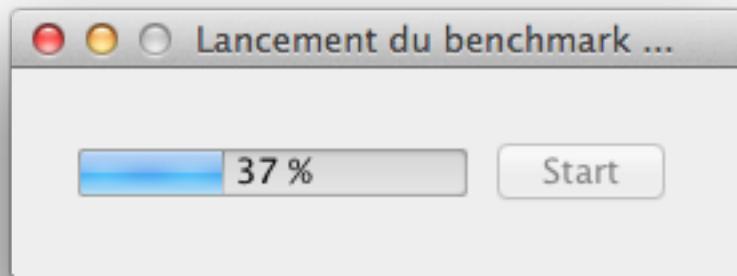


FIGURE 4.2 – Lancement du BenchMark

Après avoir paramétré le jeu (figure 4.3), les jetons sont placés au centre du plateau et par défaut c'est le joueur blanc qui commence.

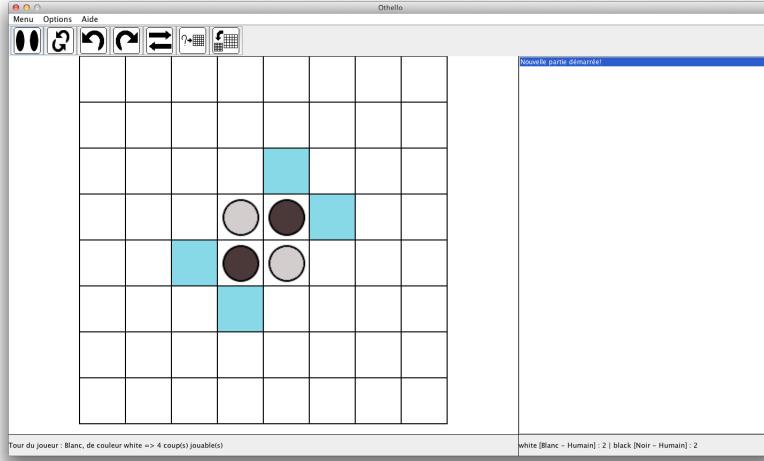


FIGURE 4.3 – Jeu de base

Le logiciel, au cours d'une partie (figure 4.4), possède plusieurs zones afin d'informer l'utilisateur : le plateau, l'historique (liste à droite), menu général en haut à gauche ainsi qu'une aide et enfin, en bas, à gauche une barre de notification et à droite, une barre de statistiques sur la partie actuelle.

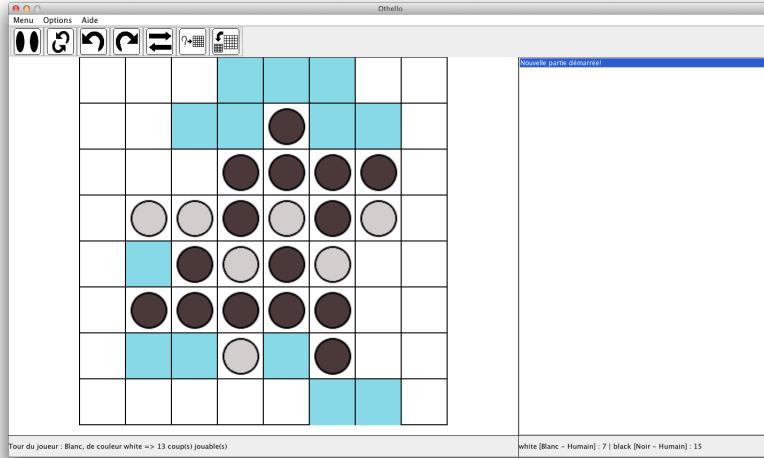


FIGURE 4.4 – Partie en cours

Si l'utilisateur le désire, il peut choisir de revenir à n'importe quelle position jouée (figure 4.5) en cliquant sur le bouton "Revenir à une position". Ainsi, il peut recommencer une partie à une position donnée.

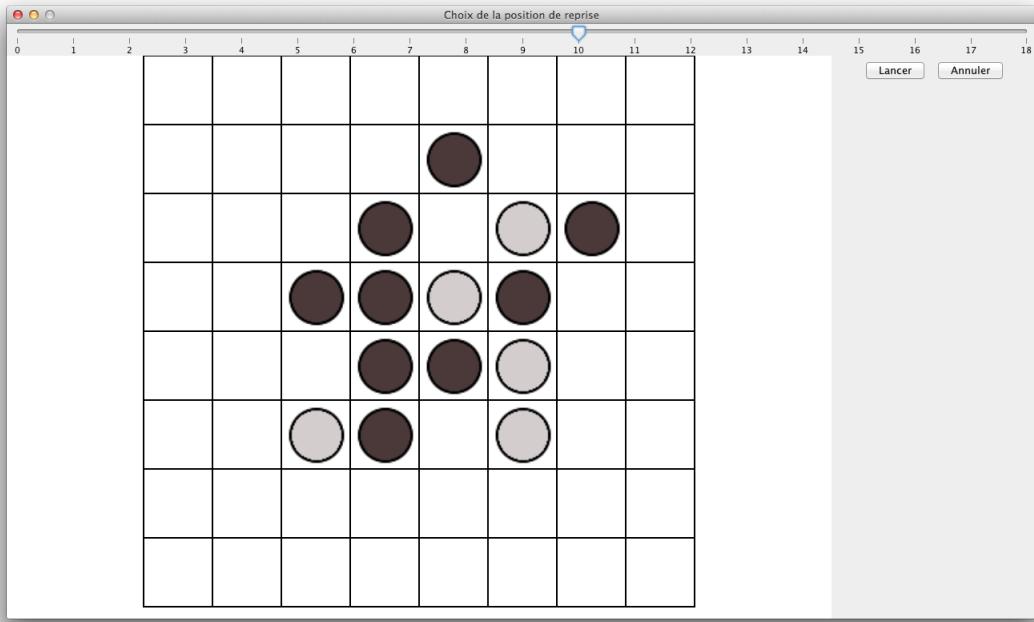


FIGURE 4.5 – Changement de position jouée grâce au slider

4.1.2 Editeur de plateau

Afin d'accéder à ce module, vous devez aller dans le menu "Menu" puis "Configurer une partie".

Une fois le module lancé, il suffit juste de suivre les instructions à l'écran.

Vous devrez tout d'abord saisir la taille du plateau de jeu, puis préciser la grille de jeu initiale (pion par pion ou à l'aide d'un fichier préconfiguré par vos soins). Il ne vous restera plus qu'à renseigner les valeurs concernant les joueurs et l'intelligence artificielle.

Si vous voulez créer un fichier de jeu à partir d'un plateau préconfiguré, vous devrez créer un fichier avec l'extension ".grd" puis le remplir de façon à représenter la grille de jeu. Pour un emplacement vide de la grille, vous devrez mettre un tiret (" - "), pour un pion blanc, vous devrez saisir la lettre o en minuscule ("o") et pour un pion noir, vous devrez saisir l'astérisque ("*"). Pour finir avec la création de ce fichier, afin de revenir à la ligne, vous devrez utiliser la touche entrée de votre clavier ("enter").

Voici un petit exemple ce que que pourrait représenter un plateau préconfiguré sur une grille de jeu de dimension (20x15).

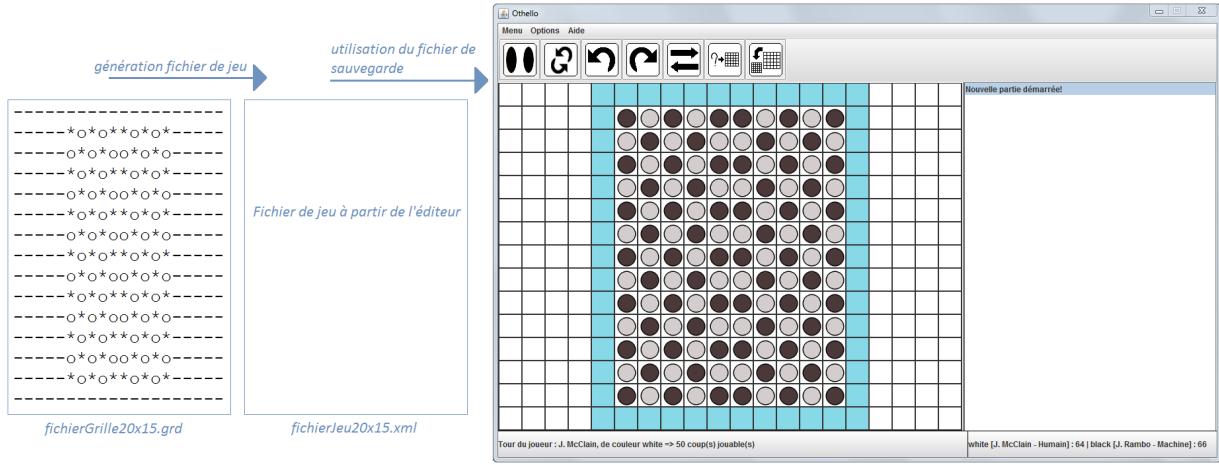


FIGURE 4.6 – Exemple d'utilisation de l'éditeur de plateau

Attention, lors de la création du fichier de jeu, vous devez absolument passer un fichier ".grd" contenant une grille correspondant aux dimensions du plateau de jeu que vous êtes en train de créer, sinon votre grille de jeu sera standard. Elle sera aussi standard si vous saisissez un autre caractère que ceux énoncés ci dessus.

4.1.3 Aide du jeu

Dans un premier temps, nous avions choisi de faire un fichier PDF contenant les règles et l'aide du jeu. Finalement, nous avons décidé de développer un petit site internet en HTML5 et CSS3 qui est visible en local.

Le site présente le projet, le jeu avec ses règles, les documents relatifs au développement, la documentation et l'équipe. Depuis le jeu, nous avons le bouton "Aide" qui permet d'y accéder directement.

En annexe D.1, nous avons une capture ainsi que l'adresse du site hébergé sur le serveur du Cremi.

4.2 Améliorations possibles sur le projet

4.2.1 Interface graphique

Notre version étant une version Beta, il existe quelques bugs graphiques. Notamment au niveau de l'affichage du plateau. Lorsque la grille de jeu s'affiche on peut voir les cases apparaître les unes après les autres. Cela peut gêner l'utilisateur. Concernant le menu, il s'affiche parfois derrière la grille de jeu, il faut donc, afin de pouvoir le voir dans son intégralité, réduire puis restaurer la fenêtre de jeu.

Une fonctionnalité intéressante à modifier serait l'interface d'affichage. En effet, celle-ci se rafraîchit plusieurs fois à la suite d'une pièce posée car l'ajout d'un pion entraîne

plusieurs calculs qui rafraîchissent tout l'interface (exemples : action de poser un pion, calcul des pions qui se retournent, calcul des pions jouables pour l'adversaire, ...). Cette modification est très facile à implémenter grâce au pattern Observer.

4.2.2 Fichier de sauvegarde

L'une des optimisations que nous aurions pu apporter au projet concerne le module de sauvegarde (notamment l'auto-sauvegarde). En effet, lorsque l'on fait appel à la sauvegarde automatique, un nouveau fichier est créé à chaque coup joué (que l'utilisateur soit un humain ou l'intelligence artificielle). Il serait intéressant qu'à chaque fin de partie, les fichiers créés soient détruits afin de ne pas avoir occupé un espace disque trop grand au bout d'un certain nombre de parties jouées.

Comme indiqué dans le paragraphe précédent, le jeu possède un module d'édition de plateau et réalise régulièrement des sauvegardes. Le problème est que lorsque l'on veut charger une partie, le temps de chargement peut être assez long. Effectivement, il est très rapide sur des grilles de taille standard (inférieure à 10×10), mais sur des grilles de taille 50×50 par exemple, il peut durer plusieurs secondes. La solution est donc d'exécuter le chargement d'un fichier de sauvegarde sur son propre thread afin que le reste du logiciel ne se bloque pas.

4.2.3 Module éditeur de plateau

Nous avons développé un module de création de plateau de jeu en lignes de commande (cf. partie Etat du projet). Il serait intéressant de réaliser un éditeur de plateau de jeu graphique basé sur l'interface graphique du jeu elle-même. Du fait d'avoir réalisé une architecture du logiciel modulaire, il serait très simple de l'intégrer dans le projet.

De plus, lorsque l'on exécute le module d'édition de plateau de jeu, si l'utilisateur ne souhaite pas terminer la création de son plateau, le seul moyen pour quitter l'éditeur est de fermer complètement l'application. La solution à ce problème serait d'ajouter à l'éditeur, une fonctionnalité permettant de quitter proprement le module, et ainsi redonner la main au jeu.

4.2.4 Améliorations diverses

Lorsque nous avons testé notre logiciel sur les ordinateurs du Cremi, nous nous sommes compte que le module de gestion de fichiers ne gère pas les chemins de type /net/cremi/... mais nécessite un chemin C :\... Ainsi, il faudrait utiliser des chemins relatifs, ./chemin/..., plutôt qu'un chemin absolu. A cause de ces inconvénients, la sauvegarde utilisateur et le chargement ne fonctionnent pas au Cremi.

Grâce à l'architecture choisie, l'interface peu travaillé, est facilement modifiable et de nombreuses autres fonctionnalités peuvent être facilement implémentées. Parmi ces fonctionnalités, on pourrait avoir une amélioration de l'IA qui serait évolutive, c'est-à-dire qui

apprend en fonction des coups joués par l'utilisateur, s'améliorant donc au fur et à mesure de son utilisation.

Dans l'optique de rendre le jeu plus fluide et agréable à jouer nous avons créé le module BenchMark. Celui-ci attribue un chiffre à la puissance de la machine qui le lance. Nous n'avons pas eu le temps de le faire mais cela aurait été une amélioration permettant de gérer le déclenchement de certains algorithmes "gourmands" de l'IA. En effet, une machine peu puissante lancerait ces algorithmes au dernier moment pour que l'interface soit toujours réactive à l'utilisateur. Une machine puissante pourrait les lancer plus tôt dans la partie et ainsi optimiser les choix de l'IA.

Dans l'idée de rendre le jeu plus accessible nous avons codé la gestion des textes de l'interface utilisateur, pour chaque module, dans une interface. Ainsi, si l'on souhaite internationaliser le logiciel il suffit de créer une classe qui récupère le bon texte dans la langue désirée dans l'interface.

4.3 Protocoles de test

Depuis le début de nos études, c'est la première fois que nous avons à réaliser un projet aussi complet (étude bibliographique, étude des besoins, gestion d'un client, ...). Ainsi, nous n'avons pas réussi à respecter notre emploi du temps comme nous aurions dû. De ce fait, nous n'avons pas pu réaliser autant de tests que voulu.

Pour s'assurer du bon fonctionnement de notre architecture et de notre implémentation, nous avons réalisé différents types de tests.

4.3.1 Tests unitaires

Le **test unitaire** est un procédé permettant de vérifier le bon fonctionnement d'une partie spécifique du programme. Pour se faire, nous nous sommes aidés de l'outil JUnit pour tester certaines de nos méthodes.

Parmi les tests prévus, nous en avons réalisé la plus grande partie que nous présentons ci-dessous :

- L'othellier choisi au chargement correspond bien à celui de la sauvegarde
- La position choisie ne doit pas être hors des bornes
- Lorsque l'IA suggère un coup, la position de celui-ci doit correspondre à un coup valide
- La grille générée doit avoir la taille demandée
- Le temps de réflexion alloué à la réflexion de l'IA doit être respecté
- Tester visuellement que les jetons sont correctement échangés avec ceux de l'adversaire lors du changement de joueur

- Changer de joueur fréquemment
- Les fichiers de sauvegardes contiennent les informations correspondant à la partie sauvée.

4.3.2 Tests d'intégration

Le **test d'intégration** permet de vérifier le bon fonctionnement du regroupement de différents modules validés. Pour l'évolution de chaque module, à l'ajout ou la modification de fonctionnalités, entraînant la création d'un .jar, tous les tests d'intégration sont refaits afin de vérifier que les changements ne sont pas sources de bugs.

4.3.3 Tests fonctionnels

Le **test fonctionnel** sert à vérifier qu'une fonctionnalité est correctement exécutée. Il se rapproche plus d'une représentation accessible au client, "personne lambda", qu'à un expert en informatique.

Voici une liste des tests effectués :

- Vérifier que le résultat de la réflexion de l'IA est correct.
- Le fichier de sauvegarde est bien créé et valide. De plus, le plateau affiché après chargement correspond bien au fichier en entrée.
- La fonctionnalité de retour en arrière est désactivée lorsqu'il n'y a plus aucun coup précédent le coup actuel.

4.3.4 Tests systèmes

Les **tests systèmes** sont un ensemble de tests effectués pour évaluer la correspondance avec les exigences spécifiées. Ils appartiennent à la classe des tests de type boîte noire, et donc, ne nécessitent aucune connaissance au niveau développement. Voici une liste des test effectués :

- Exécution du jeu simultanément à un autre logiciel (gourmand en ressources) pour vérifier que le comportement du logiciel reste inchangé.
- Le temps de réflexion associé à l'IA doit correspondre à celui donné lors du choix utilisateur
- Exécution du logiciel sur différentes machines de puissances différentes (netBook, ultraBook, PC de salon, PC de jeu)
- Fermer l'application en plein milieu d'une partie, pendant la réflexion de l'IA.
- Tester le logiciel sur différents systèmes d'exploitation supportant la JVM : Windows, Mac OS, Linux

4.3.5 Tests non réalisés

Voici quelques tests que nous n'avons pas eu le temps de faire :

- Test d'utilisation : Faire jouer des personnes inconnues au projet, des personnes ne connaissant pas les règles du jeu Othello, des enfants aux personnes âgées.
- Test d'utilisation : Complément du test de Facilité d'utilisation. Faire jouer des scientifiques pouvant utiliser les résultats du programme à des fins professionnelles.

Conclusion

L'objectif de ce projet est de mettre en application les principes et techniques liés à la programmation vus durant notre formation telles que la gestion de projet, la programmation en Java et la documentation de code.

Travailler en quadrinôme sur un projet de 4 mois nous a permis de développer notre travail d'équipe mais essentiellement de comprendre l'intérêt d'un bon cahier des charges et d'une bonne répartition des tâches afin d'optimiser le travail de chaque personne.

Nous nous sommes rendus compte que la gestion du temps est très importante afin de tenir le planning établi.

L'ensemble des fonctionnalités demandées par le client a été implémenté. Nous avons un jeu fonctionnel où un humain peut affronter une IA en choisissant une difficulté. Deux IA peuvent également se rencontrer. Le joueur peut également créer son propre plateau ou revenir à une ancienne position jouée.

Bibliographie

- [Bur95] Michael Buro. Probcub : An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2) :71–76, 1995.
- [Bur97] Michael Buro. How machines have learned to play othello. *ICCA Journal*, 20(2) :71–76, 1997.
- [Bur02] Michael Buro. Logistello, 2002. URL : <https://skatgame.net/mburo/log.html>.
- [Bur03] Michael Buro. The evolution of strong othello programs. *Entertainment Computing - Technology and Applications*, pages 81–88, 2003.
- [CDN98] B. Causse, R. Delorme, and S. Nicolet, 1998. URL : <http://www.ffothello.org/info/algos.php>.
- [Che10] Jack Chen. Applications of articial intelligence and machine learning in othello. Technical report, tjhsst Computer Systems Lab, 2009-2010.
- [CN06] Audrey Colbrant and Elodie Nouguier. Jacothellon, 2005/2006. URL : <http://alasea.free.fr/Othello.pdf>.
- [Del12] Richard Delorme. Edax, 2012. URL : <http://abulmo.perso.neuf.fr/edax/4.3/index.htm>.
- [dlB96] Bruno de la Boissarie. Anthologie des programmes d'othello, 1996. URL : http://brunodlb.pagesperso-orange.fr/ot_story.htm.
- [d'O02a] Fédération Française d'Othello. Anthologie des programmes d'othello-reversi, 2002. URL : <https://skatgame.net/mburo/tour.html>.
- [d'O02b] Fédération Française d'Othello. Programmes étrangers d'othello, 2002. URL : <http://www.ffothello.org/info/programmeurs2.php>.
- [d'O02c] Fédération Française d'Othello. Programmes français d'othello, 2002. URL : <http://www.ffothello.org/info/programmeurs1.php>.
- [GP98] Louis Geoffroy and Martin Piotte. Hannibal, 1998. URL : <http://satirist.org/learn-game/systems/othello/hannibal.html>.

- [Leo95] Anton Leouski. Learning of position evaluation in the game of othello. Technical report, Department of Computer Science University of Massachusetts, 1995.
- [LM86] Kai-Fu Lee and Sanjoy Mahajan. Bill : A table-based, knowledge-intensive othello program. Technical report, Computer Science Department of Carnegie Mellon University, 1986.
- [Mac06] Steve MacGuire. Strategy guide for reverso and reversed reversi, 2006. URL : <http://www.samsoft.org.uk/reversi/strategy.htm>.
- [Mer10] Pierre-Emmanuel Mercier. Othello en c++ et sfml, 2010. URL : <http://blog.ace-art.fr/2010/04/26/othello/>.
- [Ros81] Paul S. Rosenbloom. A world-championship-level othello programm. Technical report, Computer Science Department of Carnegie Mellon University, 1981.
- [SA04] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello. Technical report, Department of Computer Science and Engineering, Paul G. Allen Center, University of Washington, Seattle, WA-98195, 2004.
- [SHM05] Guillaume Sauveur, Maxime Harnois, and Charles Melin. Jeu d'othello en java, 2005. URL : <http://www.taussane.com/backup/projet2.php>.
- [Tey12] Olivier Teytaud. Programmation des échecs et d'autres jeux. *Interstices*, 2012.
- [Tor13] Fabien Torre, septembre 2013. URL : <http://www.grappa.univ-lille3.fr/~torre/Enseignement/Cours/Intelligence-Artificielle/jeux.php>.

Annexes

Annexe A

Prototypes papier de l'interface

A.1 Prototype papier du plateau de jeu

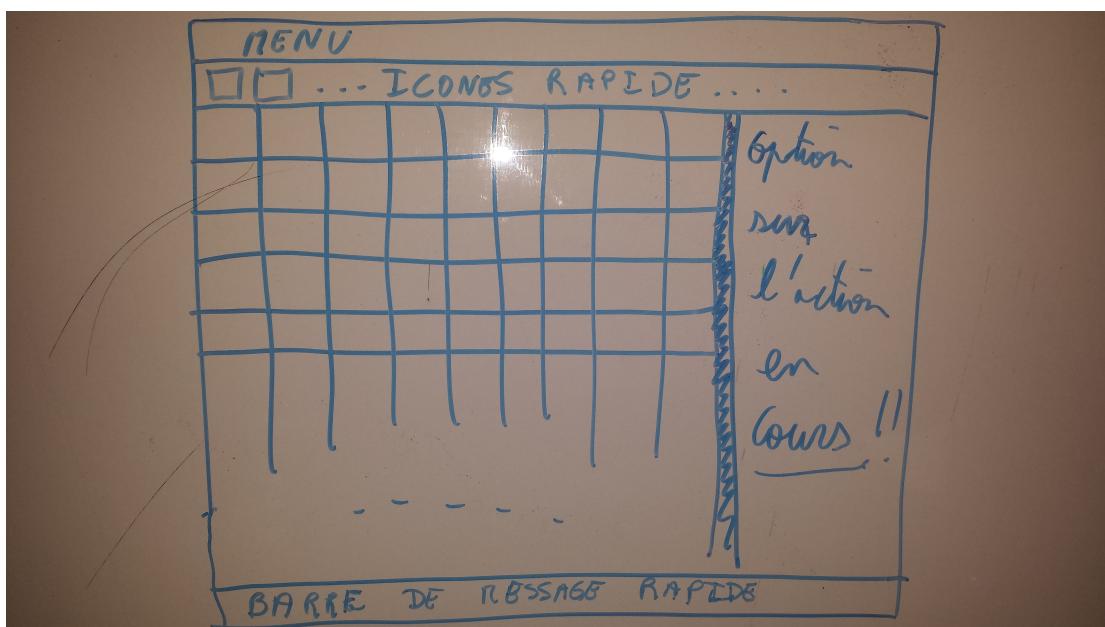


FIGURE A.1 – Plateau de jeu comprenant la grille, les menus et les boutons

A.2 Prototype papier du retour en arrière pour les coups

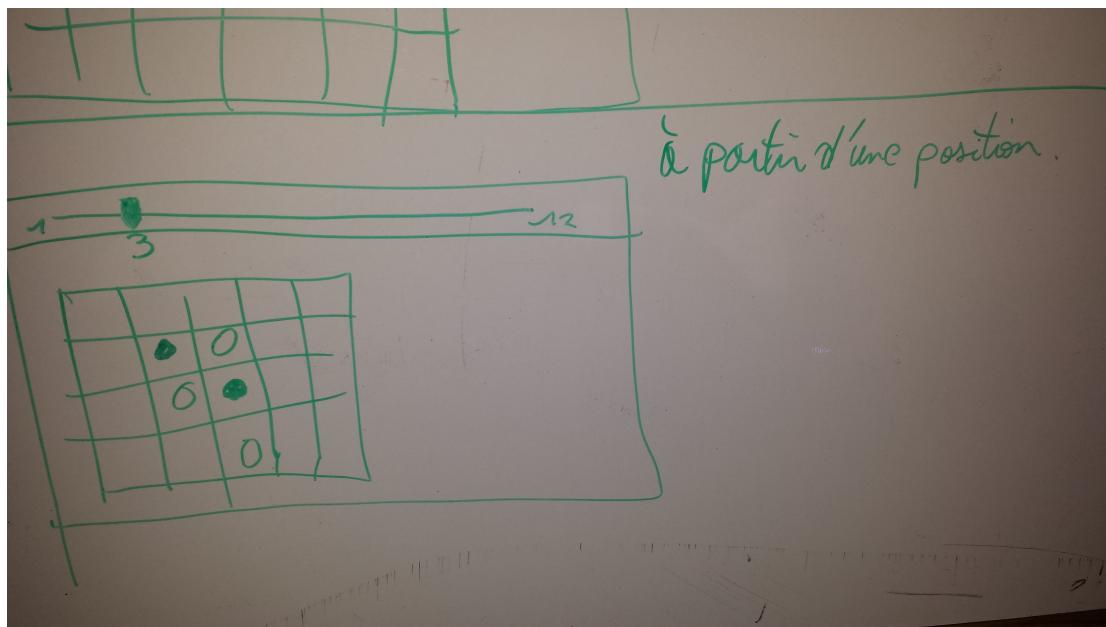


FIGURE A.2 – Prototype de la fonctionnalité de retour en arrière pour des coups

A.3 Prototype papier des boutons et menus



FIGURE A.3 – Prototype des boutons et des menus pour l'interface

Annexe B

Diagrammes UML

B.1 Diagramme de machine à états

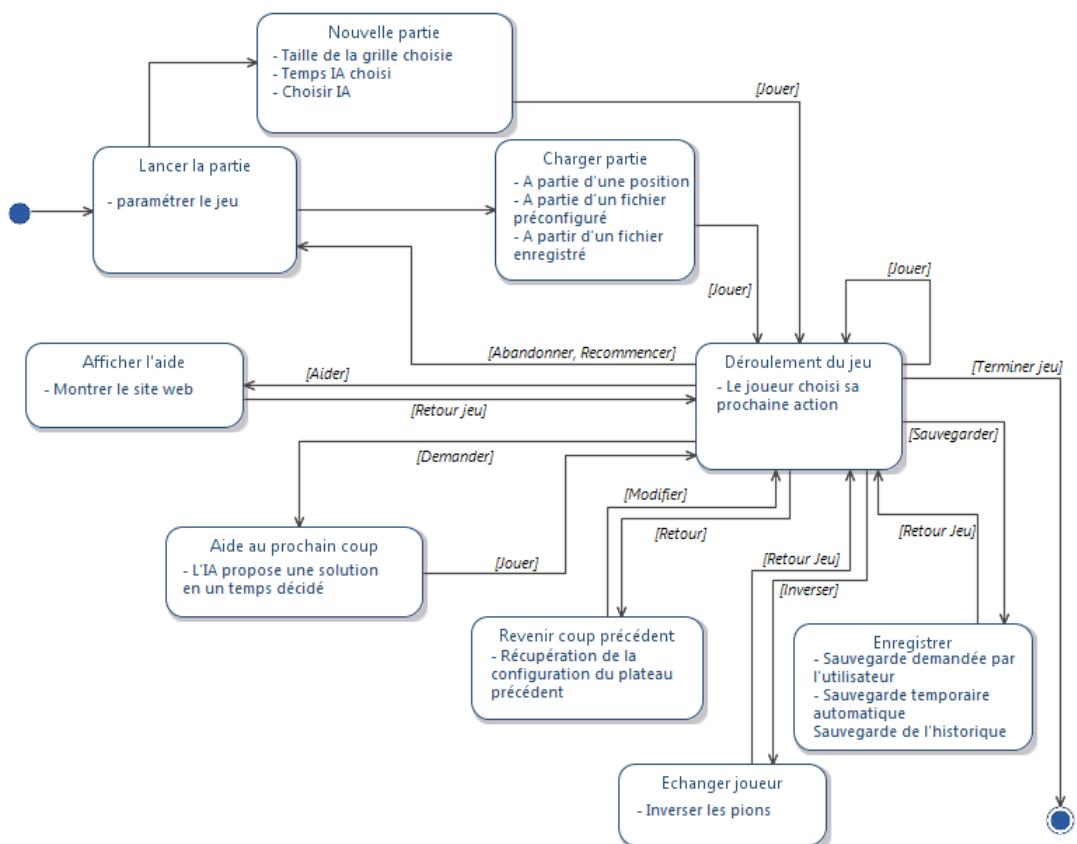


FIGURE B.1 – Diagramme UML de machine à états

B.2 Diagramme de déploiement

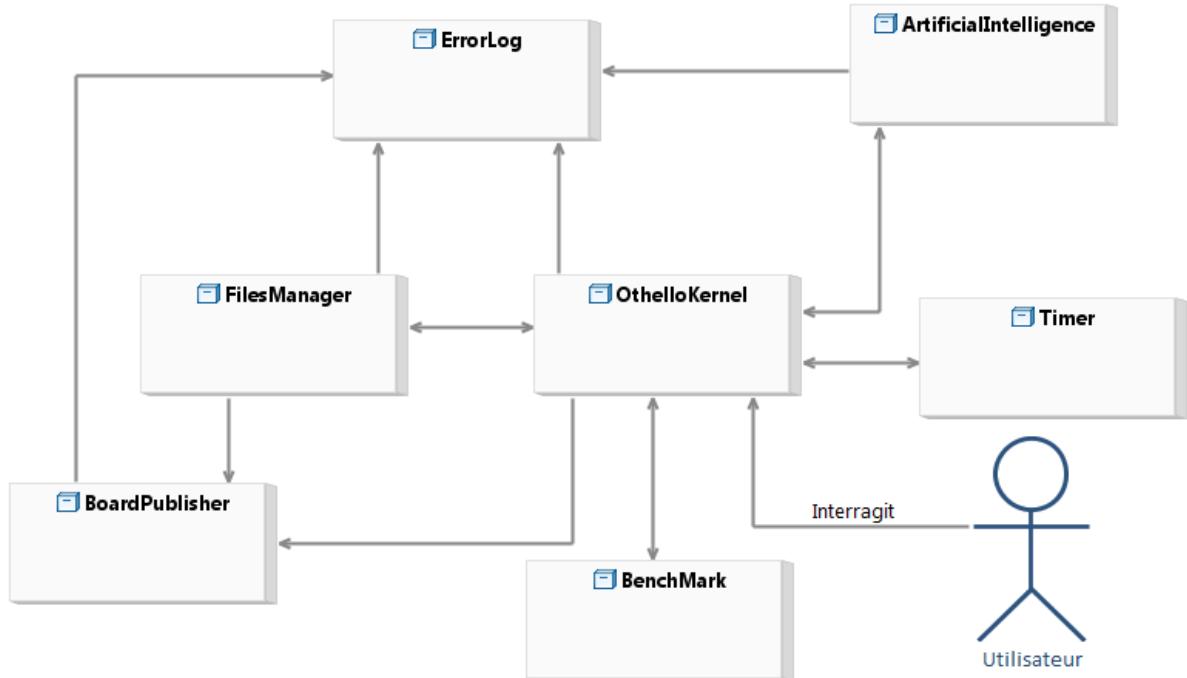


FIGURE B.2 – Diagramme UML de déploiement

B.3 Diagramme des cas d'utilisation

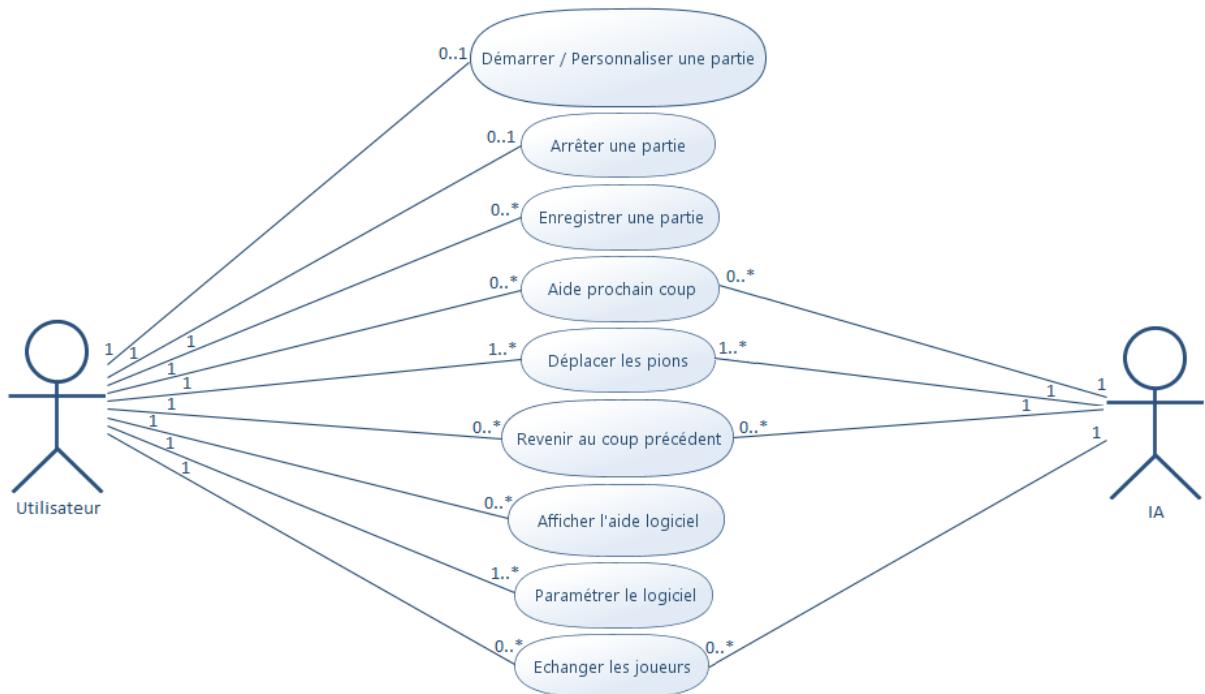


FIGURE B.3 – Diagramme UML des cas d'utilisation du logiciel

B.4 Diagramme de classes du Logiciel

B.4.1 Model du logiciel

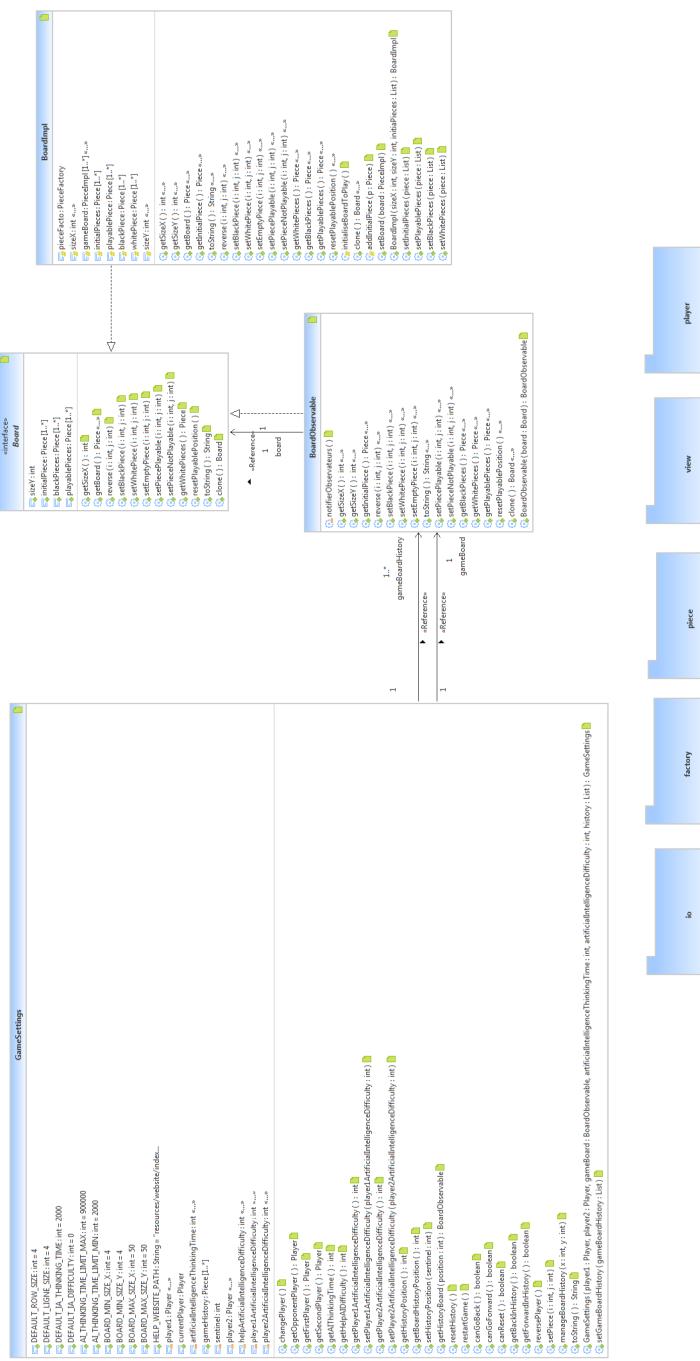


FIGURE B.4 – Diagramme UML de classes du Package Model du Logiciel

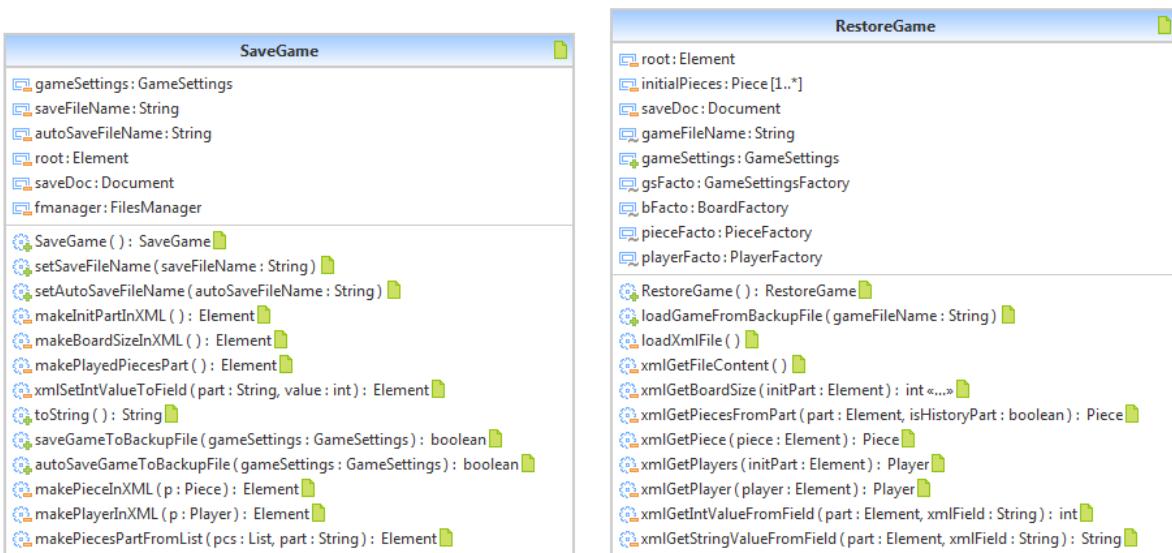


FIGURE B.5 – Diagramme UML de classes du Package Model.IO du Logiciel

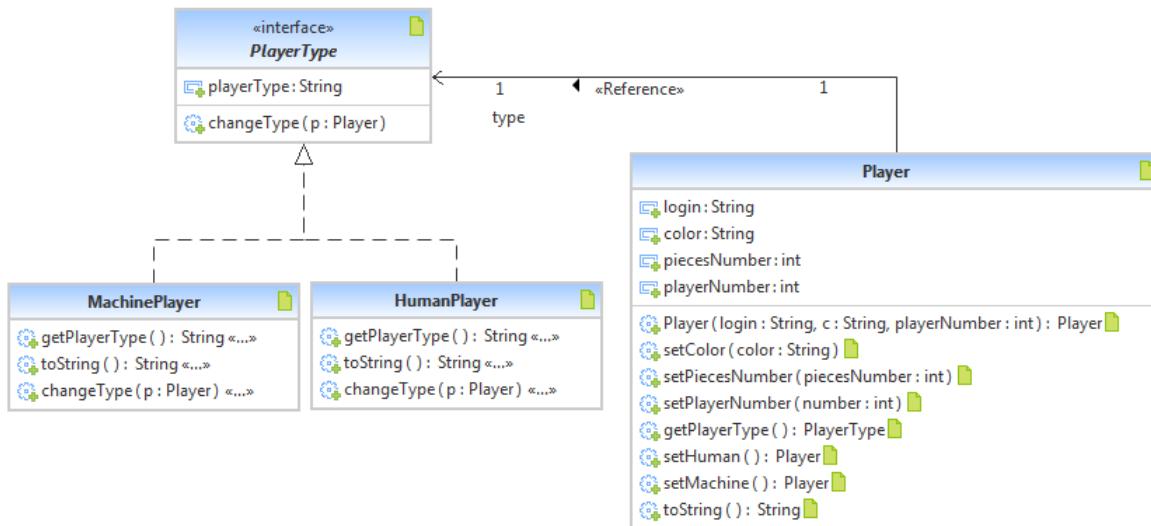


FIGURE B.6 – Diagramme UML de classes du Package Model.Player du Logiciel

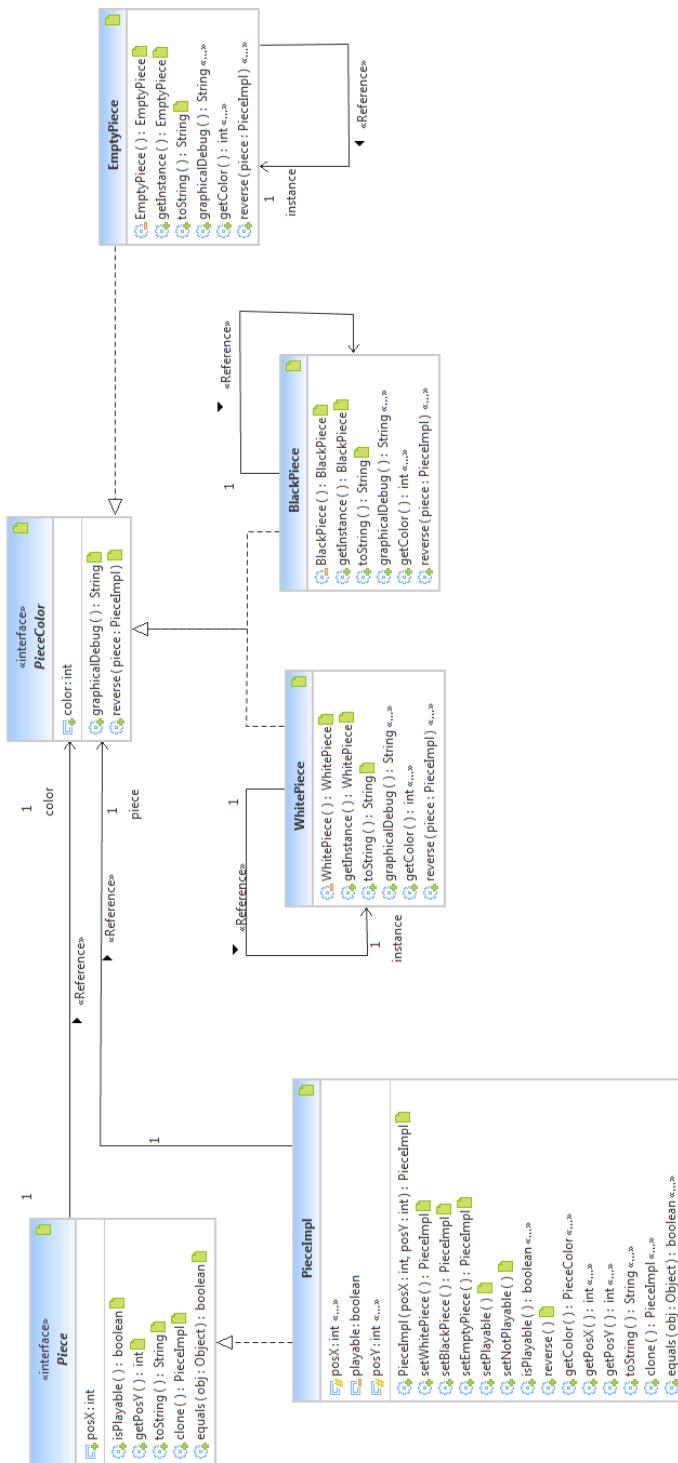


FIGURE B.7 – Diagramme UML de classes du Package Model.Piece du Logiciel

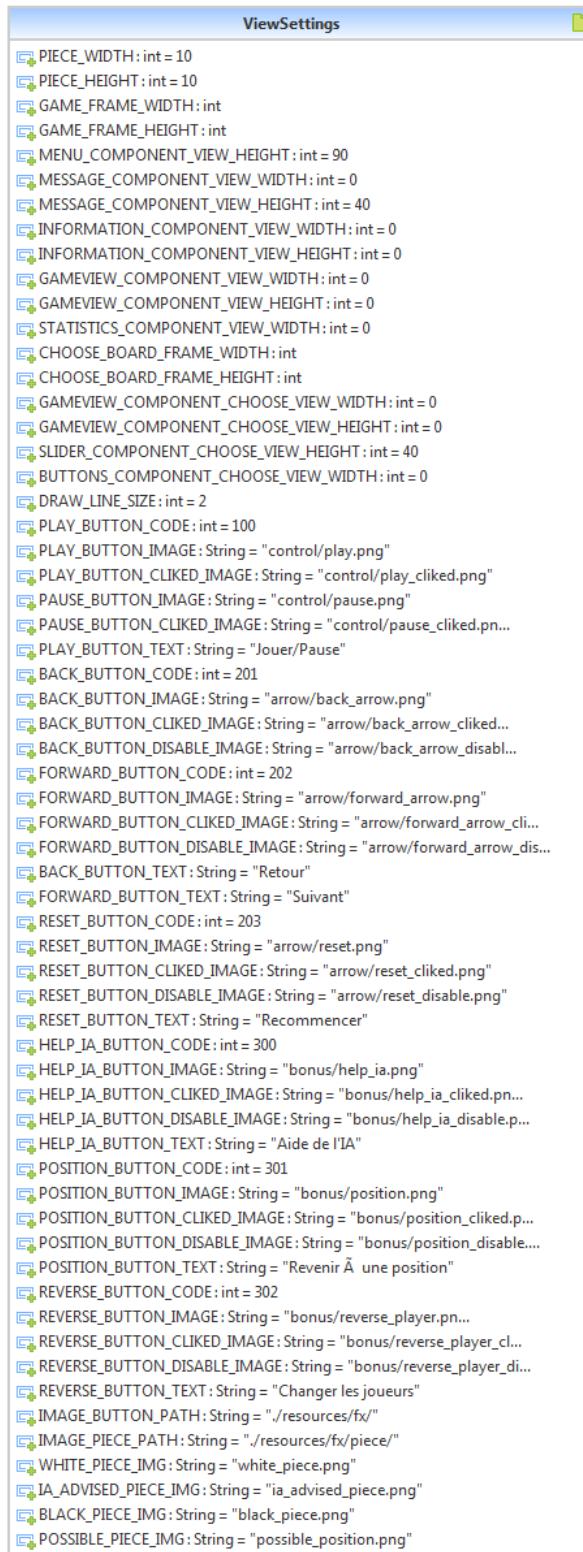


FIGURE B.8 – Diagramme UML de classes du Package Model.View du Logiciel

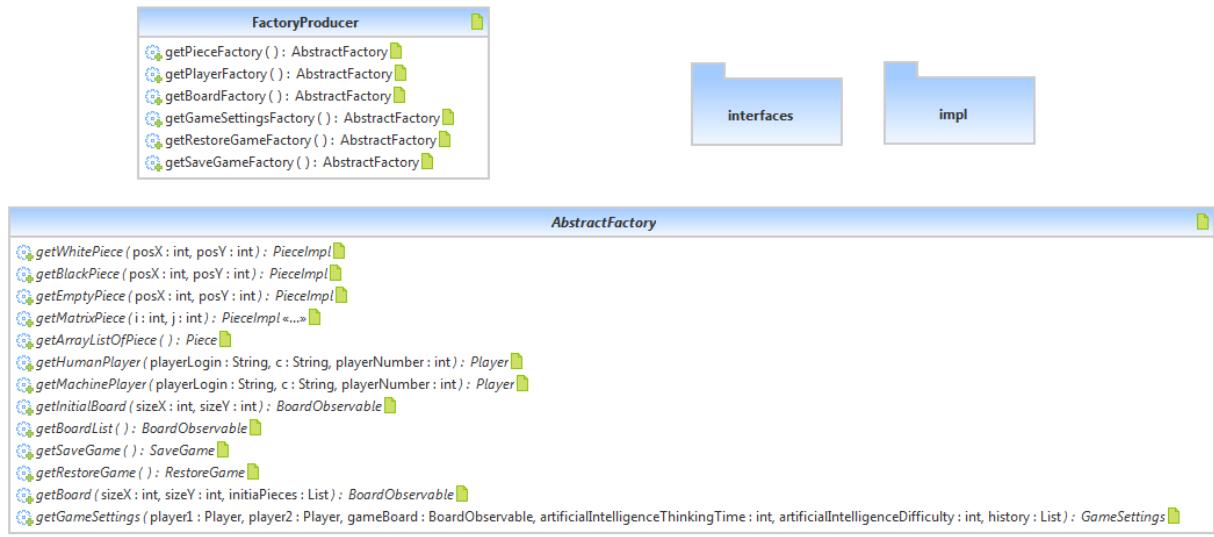


FIGURE B.9 – Diagramme UML de classes du Package Model.Factory du Logiciel

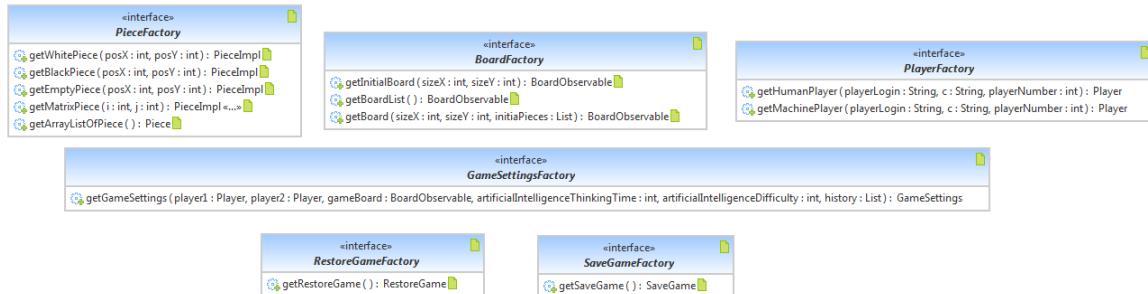


FIGURE B.10 – Diagramme UML de classes du Package Model.Factory.Interfaces du Logiciel



FIGURE B.11 – Diagramme UML de classes du Package Model.FactoryImpl du Logiciel

B.4.2 Vue du logiciel

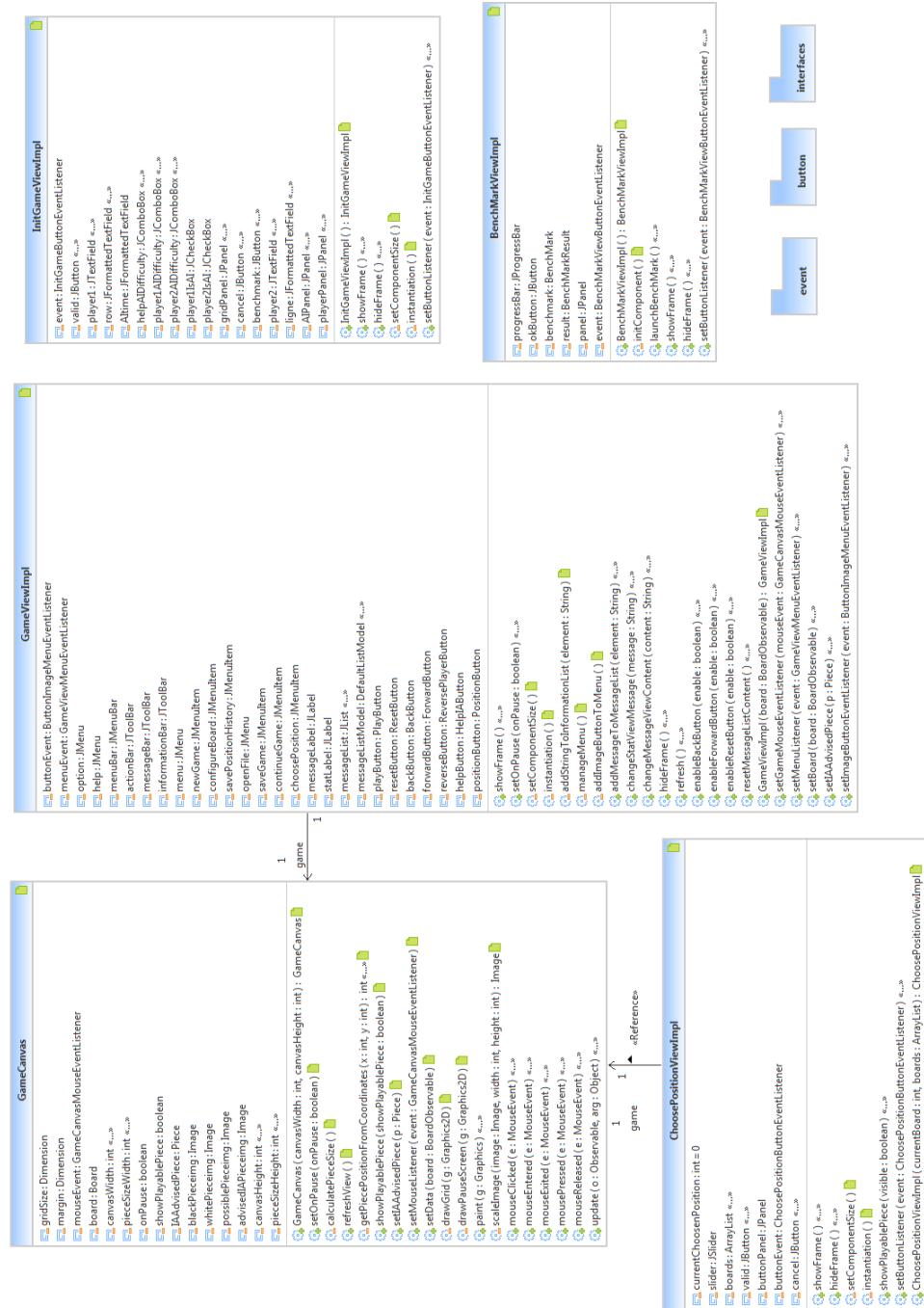


FIGURE B.12 – Diagramme UML de classes du Package View du Logiciel

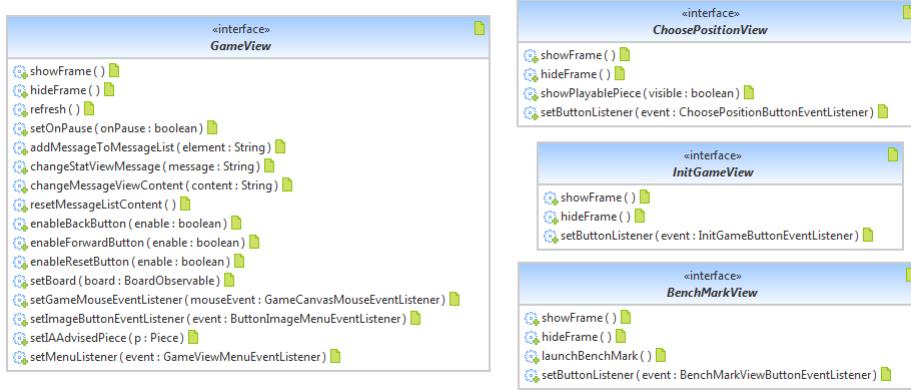


FIGURE B.13 – Diagramme UML de classes du Package View.Interfaces du Logiciel

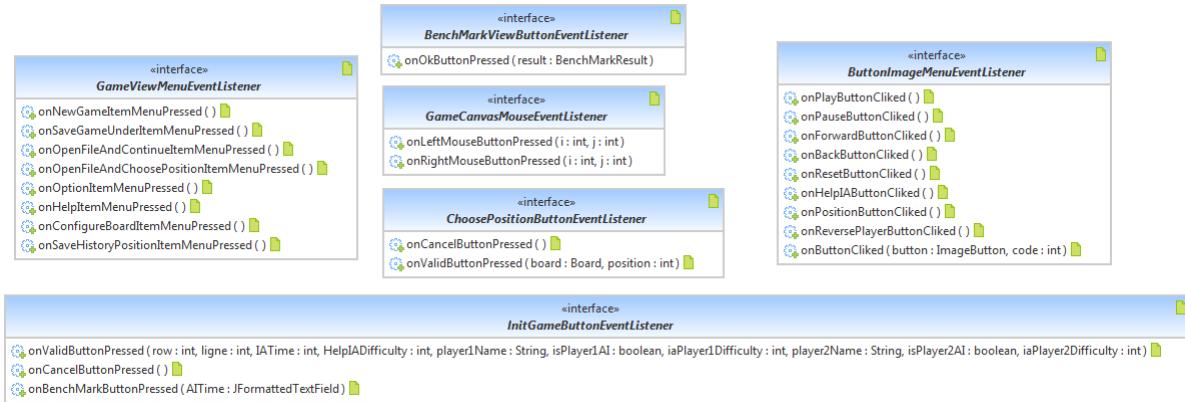


FIGURE B.14 – Diagramme UML de classes du Package View.Event du Logiciel



FIGURE B.15 – Diagramme UML de classes du Package `View.Button` du Logiciel

B.4.3 Contrôleur du logiciel



FIGURE B.16 – Diagramme UML de classes du Package Contrôleur du Logiciel

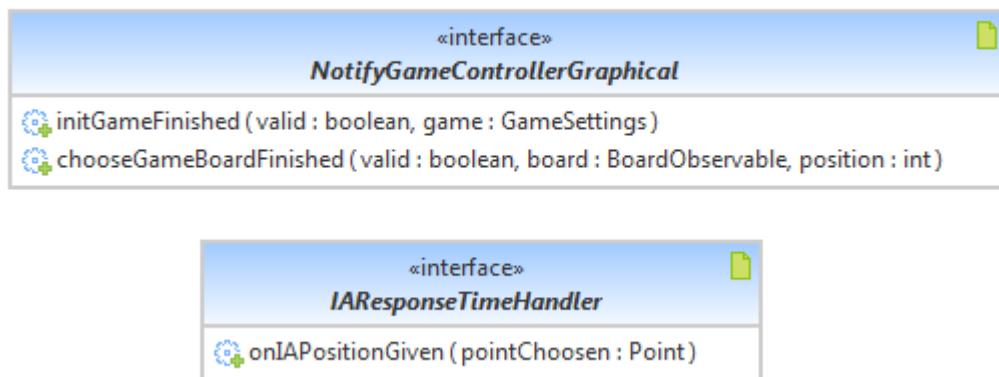


FIGURE B.17 – Diagramme UML de classes du Package Controller.Interfaces du Logiciel

B.4.4 Utils du logiciel

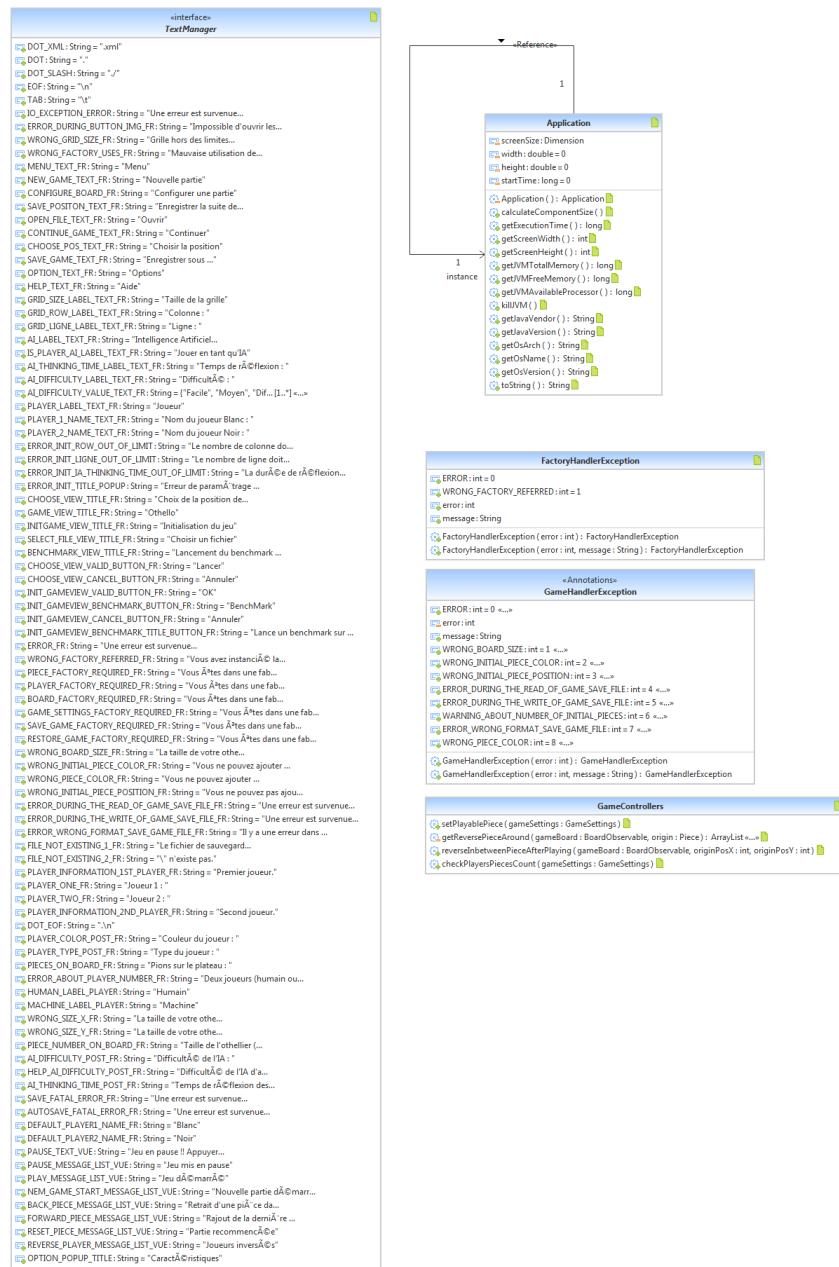


FIGURE B.18 – Diagramme UML de classes du Package Utils du Logiciel

B.5 Diagramme de classes Intelligence Artificielle

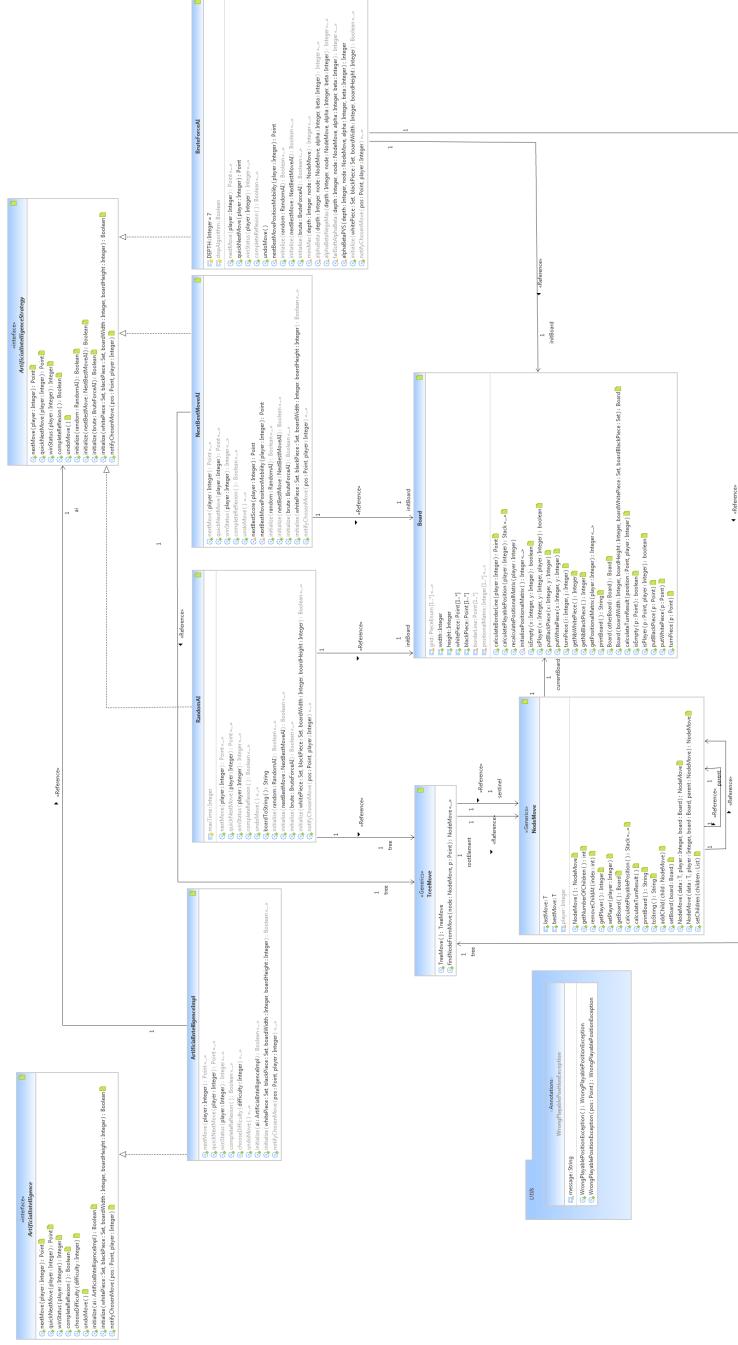


FIGURE B.19 – Diagramme UML de classes de l'intelligence Artificielle

B.6 Diagramme de classes du gestionnaire de fichiers

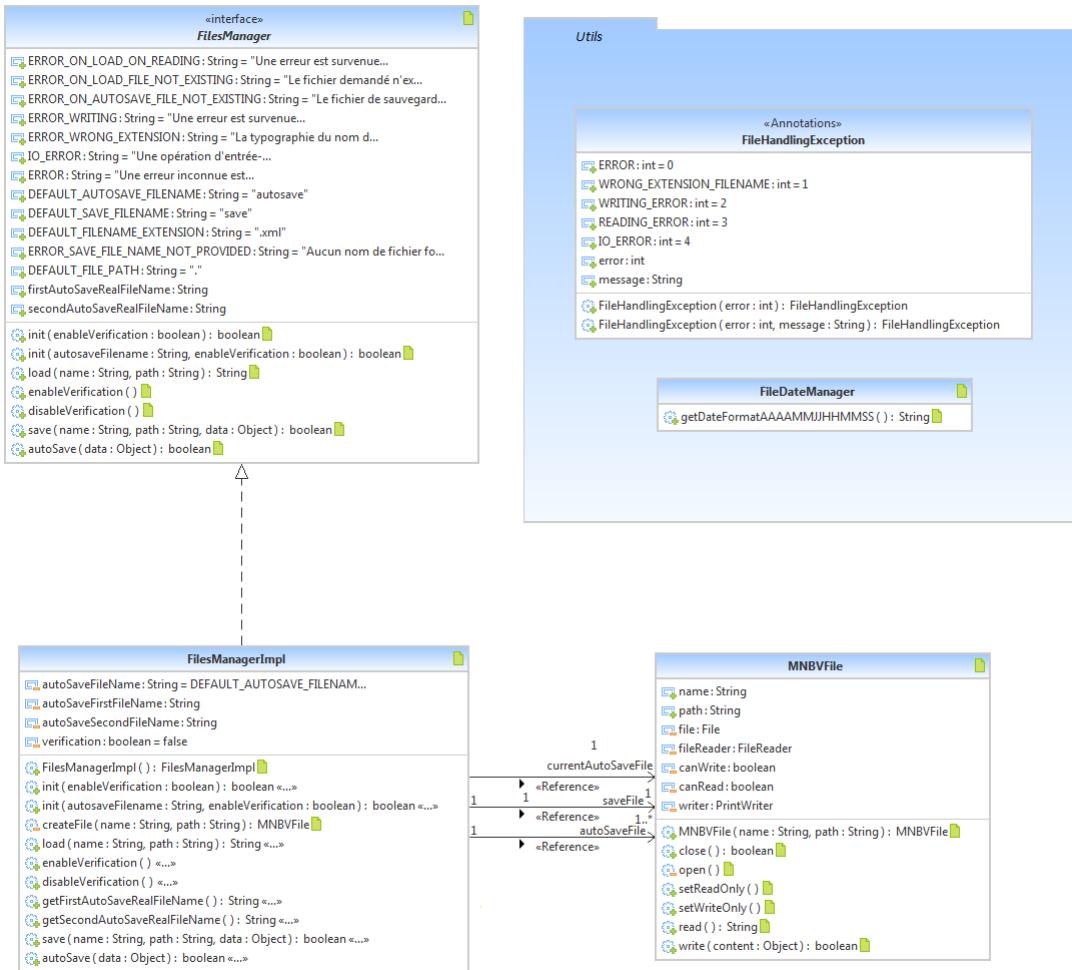


FIGURE B.20 – Diagramme UML de classes du Gestionnaire de fichiers

B.7 Diagramme de classes de l'Editeur de plateaux



FIGURE B.21 – Diagramme UML de classes de l’Editeur de plateaux

B.8 Diagramme de classes du Gestionnaire de temps

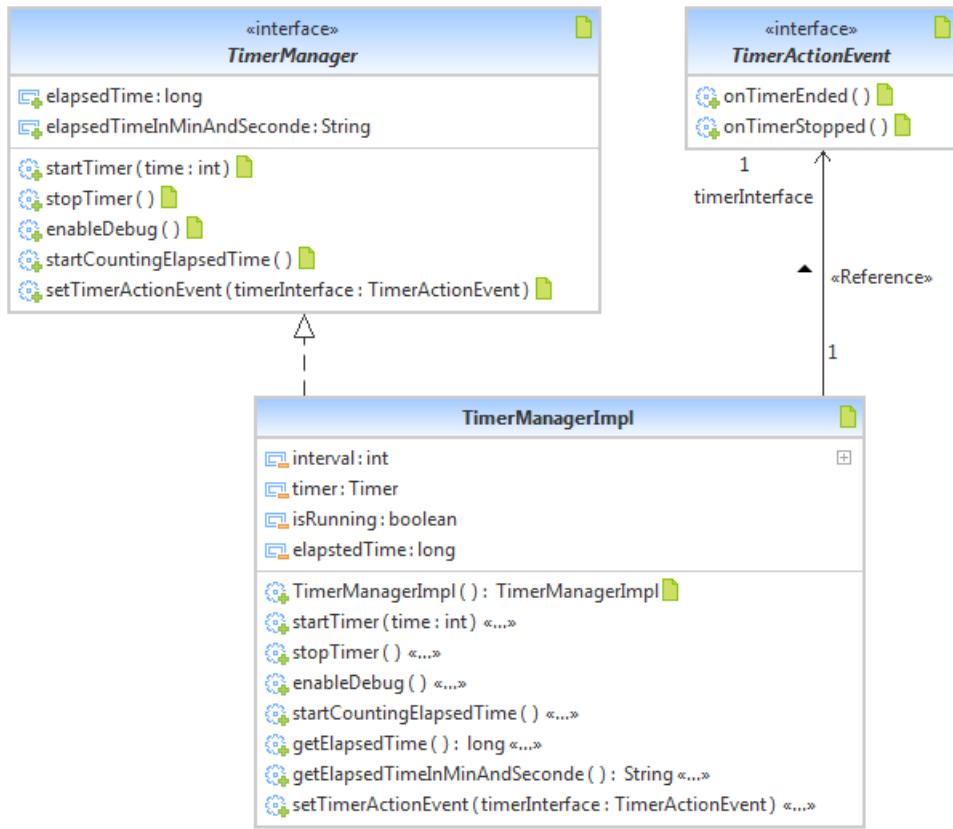


FIGURE B.22 – Diagramme UML de classes du Gestionnaire de temps

Annexe C

Pseudo-codes d'algorithmes

C.1 Algorithme MiniMax

```
int fonction minimax (int depth){  
    if (game over or depth = 0)  
        return winning score or eval();  
    int bestScore;  
    move bestMove;  
    if (noeud == MAX) { //=Programme  
        bestScore = -INFINITY;  
        for (each possible move m) {  
            make move m;  
            int score = minimax (depth - 1)  
            unmake move m;  
            if (score > bestScore) {  
                bestScore = score;  
                bestMove = m ;  
            }  
        }  
    }  
    else { //type MIN = adversaire  
        bestScore = +INFINITY;  
        for (each possible move m) {  
            make move m;  
            int score = minimax (depth - 1)  
            unmake move m;  
            if (score < bestScore) {  
                bestScore = score;  
                bestMove = m ;  
            }  
        }  
    }  
    return bestscore ;  
}
```

FIGURE C.1 – Pseudo-code de l'algorithme MiniMax [CDN98]

C.2 Algorithme Alpha-Beta

```
int alphabeta(int depth, int alpha, int beta)
{
    if (game over or depth <= 0)
        return winning score or eval();
    move bestMove;
    if (noeud == MAX) { //Programme
        for (each possible move m) {
            make move m;
            int score = alphabeta(depth - 1, alpha, beta)
            unmake move m;
            if (score > alpha) {
                alpha = score;
                bestMove = m ;
                if (alpha >= beta)
                    break;
            }
        }
        return alpha ;
    }
    else { //type MIN = adversaire
        for (each possible move m) {
            make move m;
            int score = alphabeta(depth - 1, alpha, beta)
            unmake move m;
            if (score < beta) {
                beta = score;
                bestMove = m ;
                if (alpha >= beta)
                    break;
            }
        }
        return beta;
    }
}
```

FIGURE C.2 – Pseudo-code de l'algorithme Alpha-Beta [CDN98]

Annexe D

Site Web

Nous avons choisi de développer un site web en local pour plusieurs raisons. La principale raison est qu'il nous sert de support pour l'aide du jeu. Il nous permet également de présenter le projet, l'équipe et d'avoir accès à la documentation.

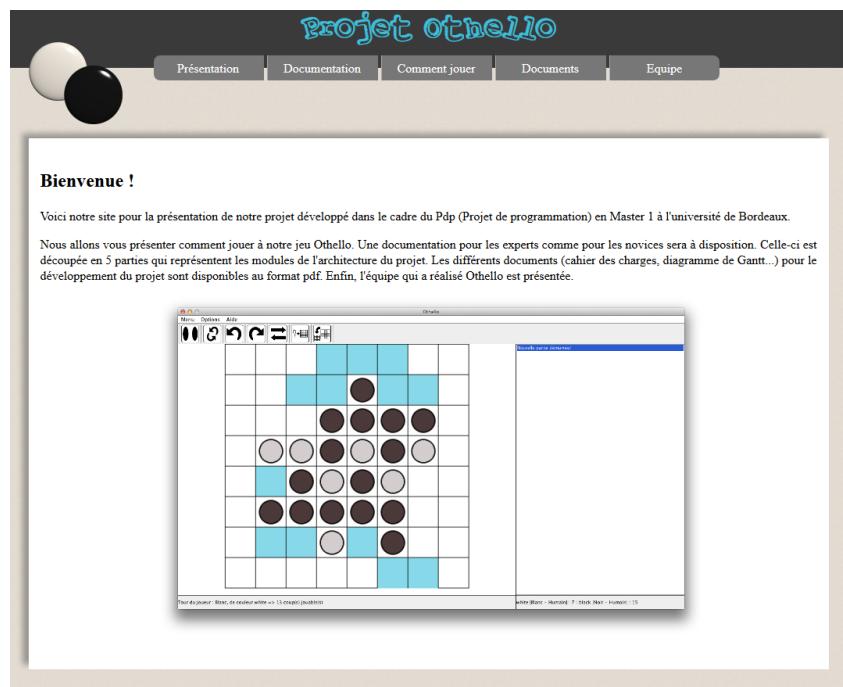


FIGURE D.1 – Capture du site internet

<http://morgane.badre.1.emi.u-bordeaux1.fr/Othello/>

Annexe E

Poster

Nous avons eu l'opportunité dans le cadre du cours de communication de réaliser un poster. C'était un moyen d'agrémenter et de présenter notre projet.

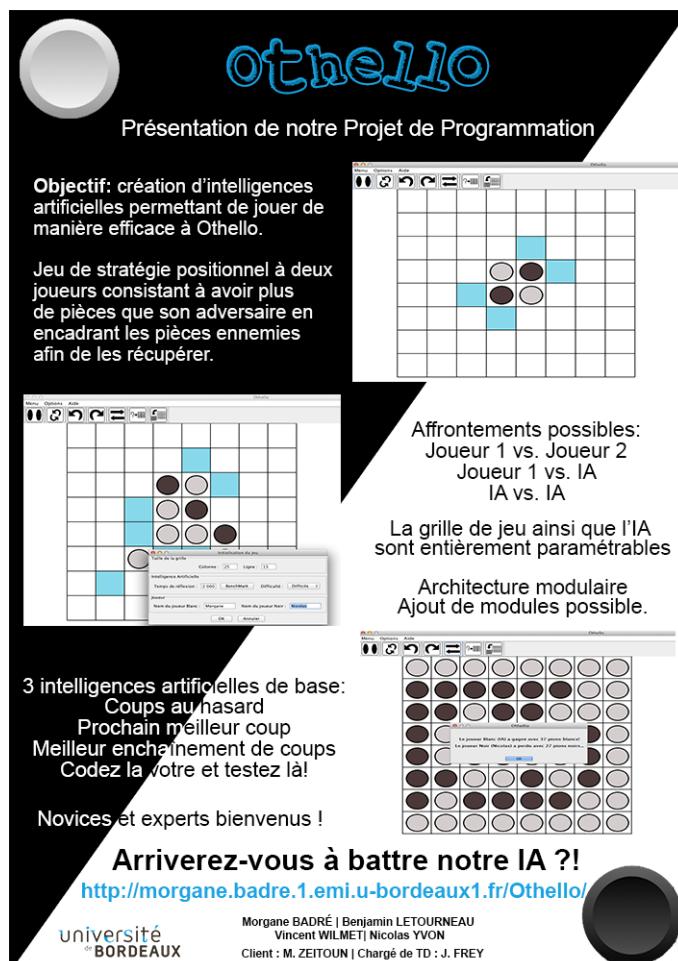


FIGURE E.1 – Poster du projet