[II.1202] Algorithmique et programmation

Livrable
Jeu des six couleurs

Tristan Muratore

Aurélien Schiltz

2015 - 2016

Table des matières

Introduction

Le module *[II.1202] - Algorithmique et programmation* dispensé au second semestre de notre première année de cycle ingénieur à l'ISEP nous amène à approfondir nos connaissances en algorithmique, et à implémenter des algorithmes dans le langage Java.

Le projet final de ce module consistait en la conception et l'implémentation complète d'un jeu de stratégie où les joueurs ont pour objectif de conquérir la totalité d'un plateau de jeu constitué de cases de couleurs. Les cases peuvent prendre une des six couleurs différentes. Lorsque c'est son tour, un joueur choisit une couleur parmi les couleurs disponibles, c'est-à-dire n'étant pas déjà possédées par un joueur de la partie (autrement dit, il ne peut y avoir plus de 4 joueurs par partie, sans quoi il n'existe plus qu'une couleur disponible). Toutes les cases possédées par ce joueur deviennent alors de la couleur qu'il a choisi, et toutes les cases adjacentes à une case possédée par ce joueur *et* de la couleur qu'il a choisie entrent en sa possession. Le joueur gagne la partie quand il obtient plus de la moitié des cases ou quand il ne reste plus de cases libres et qu'il est le joueur possédant le plus de cases.

Lien vers le repository :

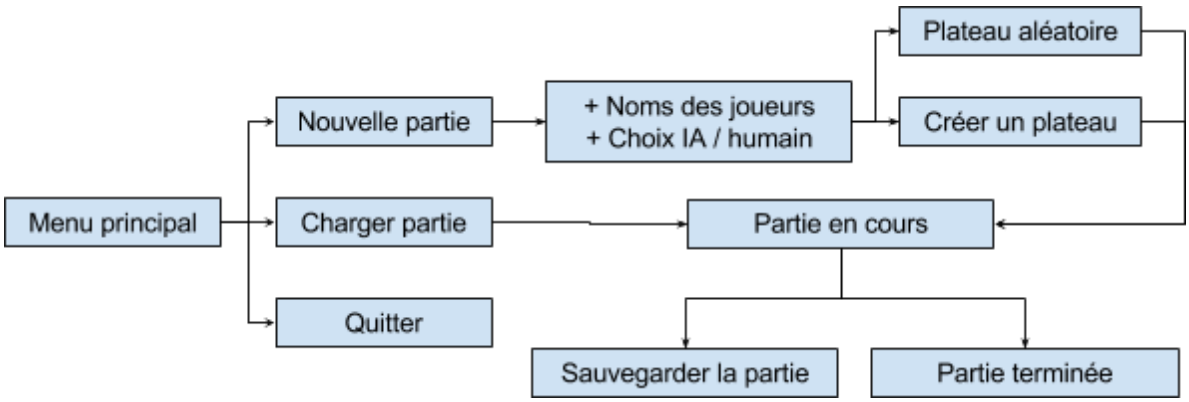
http://github.com/ohhopi/six-couleurs/

Lien ver le JavaDoc associé :

http://ohhopi.github.io/six-couleurs/

I. Fonctionnalités implémentées

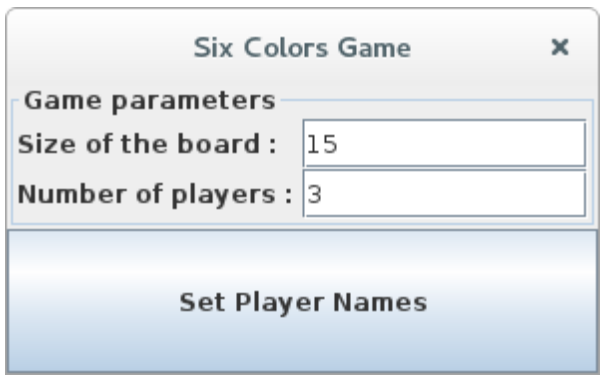
Arborescence des fonctionnalités



Parcours standard des fonctionnalités

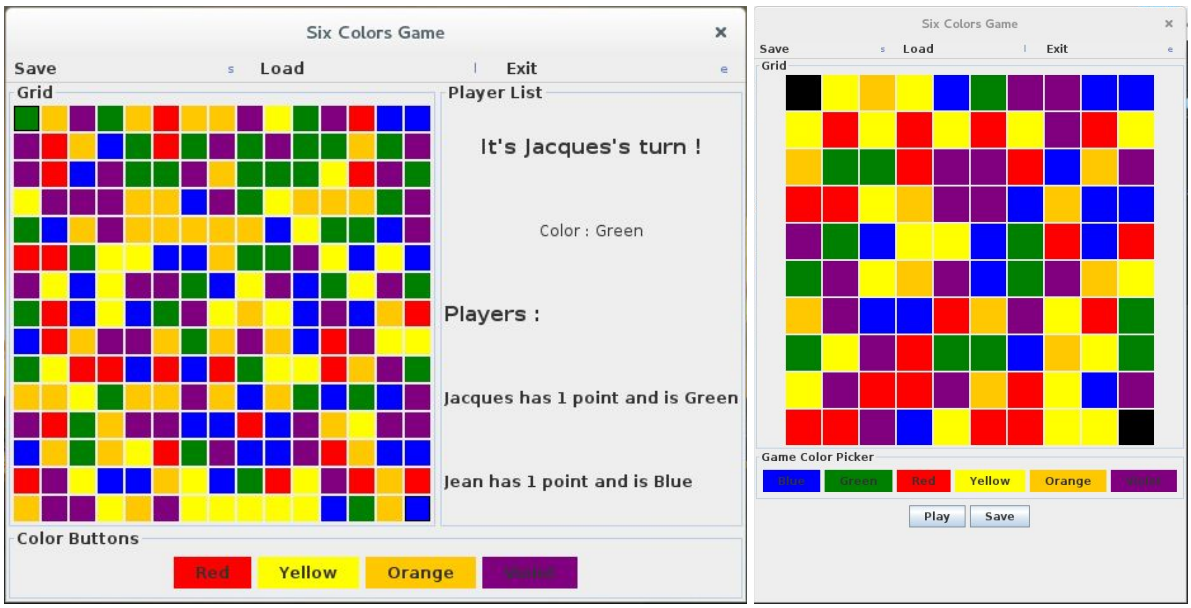
Ce menu invite le joueur à choisir entre créer une nouvelle partie locale, charger une partie sauvegardée, et quitter.

Cliquons sur “New local game” pour faire apparaître l’interface ci-dessous, qui invite le joueur à choisir une taille de grille et un nombre de joueurs pour cette partie.



Enfin, le joueur est invité à saisir les noms des joueurs, et à décider si ceux-ci seront incarnée par des Intelligences Artificielles (il faut alors en choisir le niveau) ou par des joueurs humains.

Le joueur a alors le choix entre jouer sur un plateau généré aléatoirement, ou construire son propre plateau de jeu (à droite dans l'image ci-dessous). Un clic sur "*Random Board*" permet de lancer le jeu sans plus attendre (directement l'écran à gauche ci-dessous).

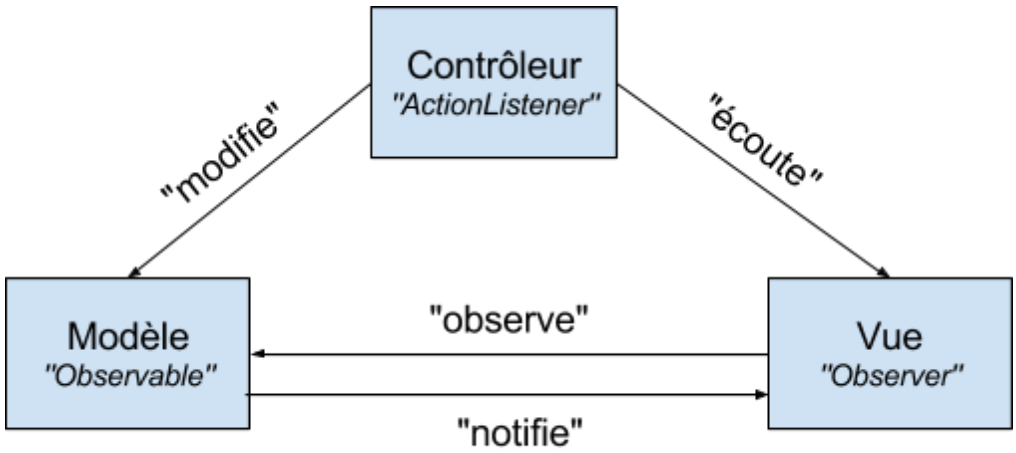


II. Implémentation des fonctionnalités

Cette partie a pour objectif de présenter la manière dont nous avons implémenté les fonctionnalités du jeu des six couleurs

A. Patron de conception

La conception du jeu repose sur le modèle MVC (Modèle Vue Contrôleur). Le terme Vue désigne l'ensemble des fonctionnalités prenant en charge l'affichage, le modèle, celles représentant les objets qui stockent les informations pertinentes au jeu, et le contrôleur, celles qui reçoivent, interprètent et valident les données saisies dans la vue et les modifient dans le modèle.



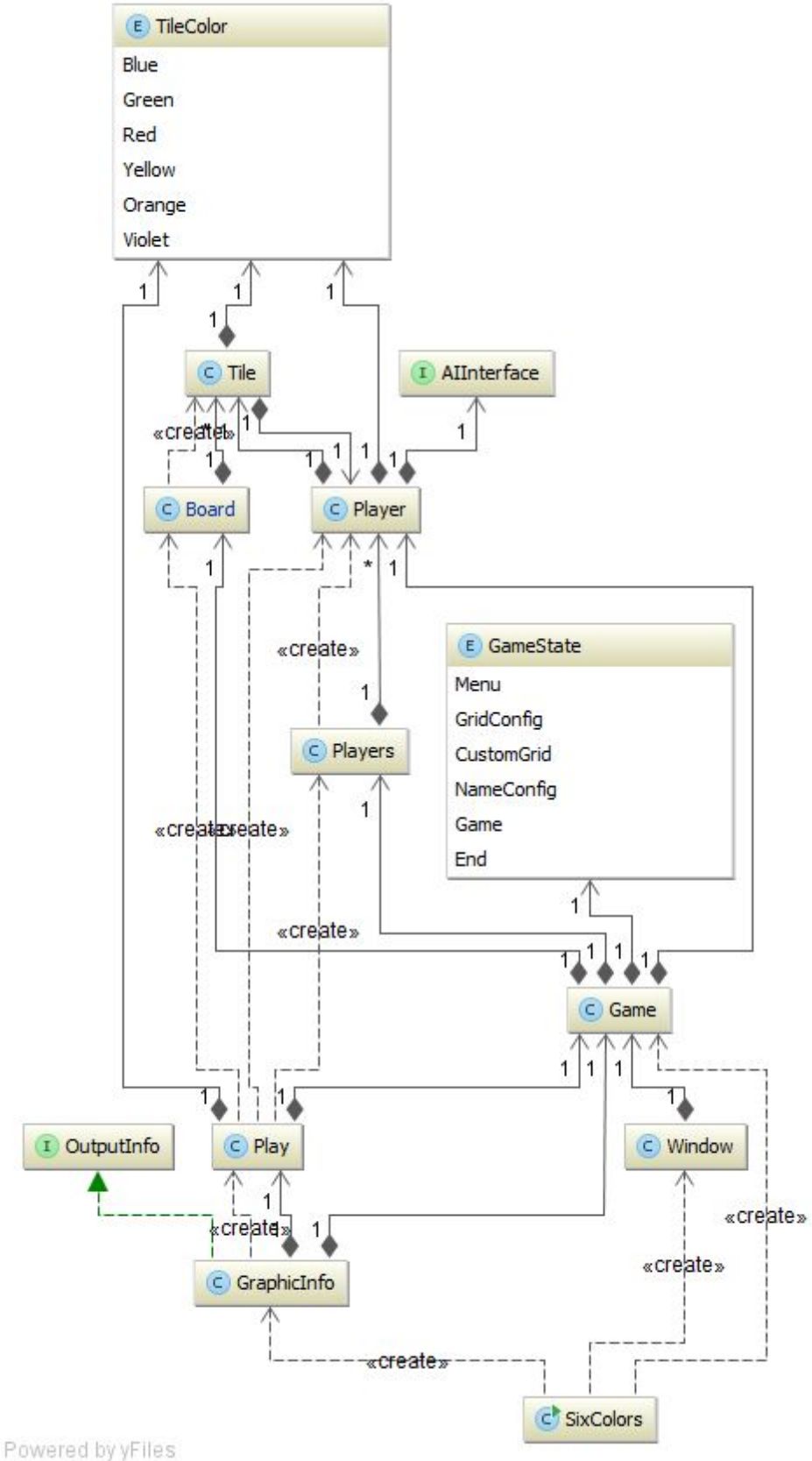
.

Plus concrètement, la manière dont nous avons implémenté ce patron de conception au sein de notre application est la suivante :

- Le **Modèle** est un **Observable,** c'est-à-dire qu'il peut être écouté. Il possède une liste d'objets qui l'écoutent et quand il change il peut les notifier.
- La **Vue** est un **Observer,** c'est-à-dire qu'il observe un objet qui le notifie quand il le faut et elle exécute sa méthode **update()**, qui rafraîchit l'affichage avec les nouvelles données du jeu.
- Le **Contrôleur** est un **ActionListener**, c'est-à-dire qu'il écoute et attends qu'une action vienne d'un des actionneurs qui l'ont enregistré pour exécuter sa méthode **ActionPerformed()**, qui récupère les données de la vue pour les traiter et si elles sont valides de les passer au modèle.

Dans le diagramme **UML** simplifié ci-dessous on peut notamment distinguer :

- **Game**, le modèle parent qui englobe tous les autres modèles,
- **Play**, le contrôleur principal du jeu qui s'interface sur un,
- **OutputInfo**, un contrôleur de récupération de données.
- **Window**, la vue graphique du jeu
- **SixColors**, la méthode **main** du jeu qui instancie notre Game, OutputInfo et Window.
- **Player**, le modèle représentatif d'un joueur qui s'interface sur,
- **AIInterface,** l'interface sur laquelle se branchent les IA.



Vous trouverez en dernière page un graphique UML complet du jeu.

Le contrôleur principal : Play.

Le coeur de cette classe repose sur un *switch* sur les différents états *(GameState)* du jeu dans une méthode nommée **control()** qui est appelée par l'**OutputInfo** quand celui-ci à récupéré les données.

Cette méthode appelle ensuite des méthodes spécifiques à chaque état qui valident ou non les données saisies et donc, passent le jeu à l'état suivant ou non.

La vue : Window.

L'implémentation de Swing est principalement dans cette classe qui étends **JFrame**, la fenêtre de jeu. Elle génère toutes les vues du jeu. A l'instar de **Play**, sa fonction pivot **update()** (appelée quand le modèle est modifié), est un *switch* sur l'état du jeu qui appelle des fonctions spécifiques à l'état pour afficher la vue associée à l'état tout en récupérant les données du modèle tout juste modifié.

Chaque méthode d'affichage se crée un objet **JPanel** qu'elle remplit puis passe en *contentPane*, l'objet JPanel affiché dans le JFrame.

B. Implémentation des règles du jeu

Comme présenté dans la partie précédente, lorsqu'un joueur clique sur un bouton pour choisir une couleur dans l'écran de partie en cours, le contrôleur frontal voit sa méthode *ActionPerformed* appelée. Celle-ci se charge de détecter quelle couleur a été choisie, et positionne la couleur courante du joueur pour refléter ce choix. La méthode *updateBoard* est alors invoquée, qui invoque la méthode *update* de l'objet B*oard* attaché à ce contrôleur. Cette méthode attend deux arguments :

- le joueur en cours ;
- la case de départ du joueur, qui sera le point de départ de propagation de la récursion.

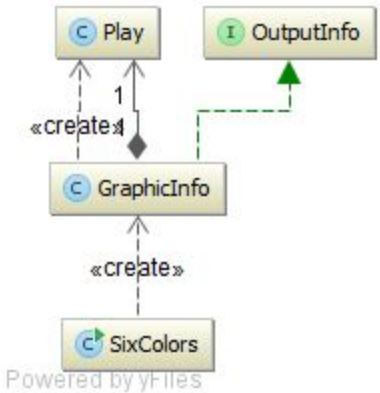
La méthode se charge alors de détecter les voisins de la case passée en second paramètre, de s'appeler récursivement sur lesdits voisins, puis de définir leur possesseur comme étant le joueur courant si elles sont de la couleur choisie par celui-ci. Ceci nous évite de devoir construire et mémoriser des groupes de cases adjacentes de même couleur dans notre représentation des données. La condition de sortie de la fonction est l'appel sur une case n'ayant pas lieu d'appartenir au "territoire" du joueur, c'est-à-dire n'étant ni possédée, ni de la couleur choisie par celui-ci.

C. Interface utilisateur

Nous avons choisi d'implémenter les interfaces graphiques en utilisant la bibliothèque **Swing**. Ce choix a été motivé par :

- l'aisance avec laquelle cette bibliothèque permet de construire des interfaces graphiques complexes, comportant des boutons, des champs de textes, des menus déroulants ;
- la performance de cette bibliothèque en terme de rapidité d'affichage, par comparaison notamment avec la bibliothèque StdDraw.

Nous avons à de maintes reprises réfléchi à la manière dont nous implémentions l'interface graphique au sein de notre patron de conception. En effet, un de nos objectifs était de rendre cette implémentation suffisamment générique pour conserver la possbilité de ré-implémenter une sortie console. Autrement dit, nous avons cherché à découpler la vue du contrôleur. Nous n'avons pas implémenté de sortie console, mais nous avons conçu une couche d'abstraction, une Interface nommée **OutputInfo.**



Le but d'OutputInfo est uniquement de récupérer les données pertinentes de la vue et de les passer au contrôleur principal **Play,** qui contient tous les tests relatifs au jeu. **Play** ne contient donc aucune spécificité à la vue utilisée. Pour l'interface graphique on passe par **GraphicInfo.**

Pour implémenter le support console, il suffirait de créer une classe implémentant l'interface **OutputInfo**, que nous pourrions par exemple nommer **ConsoleInfo.**

D. Intelligences artificielles

Les intelligences artificielles sont définies par des classes, instanciées lorsque nécessaire par le contrôleur, et stockée dans les objets Player correspondants. Elles sont accessibles à l'aide des méthodes getAiInstance de la classe Player. Elles implémentent l'interface AIInterface, comportant une seule méthode dont voici la signature :

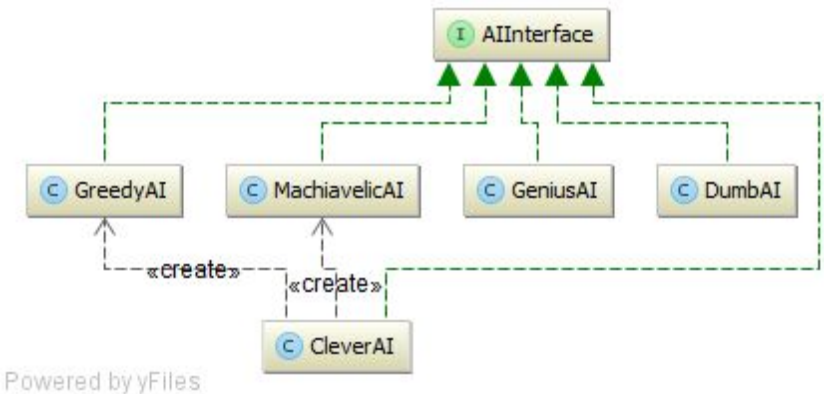
```
TileColor colorChoice(Game game);
```

Cette méthode est appelée par le contrôleur, qui lui passe en paramètre l'objet représentant la partie en cours. Les IA se servent de cet objet pour étudier la partie en cours (notamment grâce à sa méthode getAvailableTileColors, qui retourne un ArrayList contenant les couleurs disponibles), et retournent le choix de couleur qu'elles ont effectué.

Il est à noter que certaines d'entre elles, lorsqu'il est nécessaire de réaliser des tests d'hypothèses, réalisent une copie profonde de cet objet pour pouvoir y réaliser leurs tests sans l'altérer. Cette copie profonde est implémentée au sein de la méthode Game.deepCopy(), et fonctionne par sérialisation - désérialisation. Ce choix technique est potentiellement moins optimal en termes de performance qu'une implémentation de méthodes de tests d'hypothèses au sein de ces IA, mais permettait une meilleure maintenabilité et une mise en oeuvre plus simple : nous avons préféré consacrer notre énergie à développer les algorithmes de ces IA en tant que tels.

Nous avons implémenté 5 algorithmes d'intelligence artificielle différents, dont les surnoms sont :

- Dumb ;
- Greedy ;
- Machiavelic ;
- Clever ;
- Genius.



IA “Dumb”

Cette intelligence artificielle se contente de choisir une couleur au hasard, parmi celles disponibles.

IA “Greedy”

Cette IA itère sur les couleurs disponibles. Pour chaque couleur disponible, elle crée une copie profonde de l’objet Game passé en paramètre, et teste le gain de points relatif à chacun des choix possibles. Elle retourne la couleur offrant le gain le plus important.

IA “Machiavelic”

L’objectif de cette IA est exactement l’inverse de la précédente : elle réalise les tests dans le but de choisir la couleur qui aurait maximisé les points du ou des adversaires, afin de les empêcher de la choisir.

En théorie, son objectif est d’handicaper au maximum son ou ses adversaires, mais dans la pratique, la situation dans laquelle elle se trouve lui est toujours défavorable :

- face à un seul joueur, l’adversaire peut choisir parmi quatre couleurs, et n’est donc pas très handicapé par le choix de sa couleur la plus favorable ;
- face à plusieurs joueurs, cette IA sélectionne le joueur qu’elle peut le plus handicaper mais n’interfère pas avec le(s) autre(s) joueurs, et se retrouve rapidement en difficulté face à un jeu agressif tel que celui pratiqué par l’IA “Greedy”.

IA “Clever”

L’objectif de cette IA était de réaliser un compromis entre les choix de “Machiavelic” et de “Greedy” : elle joue le choix de Machiavelic si et seulement si celui-ci permet d’éviter à l’adversaire de gagner plus de `f * p` points, où f est un facteur défini empiriquement, et p est le nombre de points que le choix de “Greedy” permet de gagner. Autrement dit, elle compare les choix de “Greedy” et “Machiavelic”, avec une forte pondération pour la proposition de “Greedy” supérieure à celle de “Machiavelic”.

Dans la pratique, “Clever” n’est pas si efficace, pour exactement les mêmes raisons que celles avancées pour justifier l’inefficacité de Machiavelic : elle n’est performante que pour des valeurs de f basses, c’est-à-dire des situations où le joueur adverse pourrait gagner énormément de cases par comparaison avec notre choix “Greedy”. Quand bien même nous l’handicapons au tour actuel, il suffira à ce dernier d’attendre un tour, que la couleur de cette zone soit libre, avant de la conquérir, et nous aurons perdu un tour sans avoir beaucoup gagné de points.

IA “Genius”

Cette IA implémente l’algorithme Min/Max.

Pour cela, elle construit un arbre représentant toutes les possibilités de jeu pour ce tour, puis celles du/des adversaire(s) pour le(s) tour(s) suivant(s). Elle réduit ensuite cet arbre récursivement : à chaque niveau, elle choisit l’option qui minimise les points gagnés par le(s) adversaires et qui maximise les points qu’elle gagne en réalisant le choix représenté par le noeud courant.

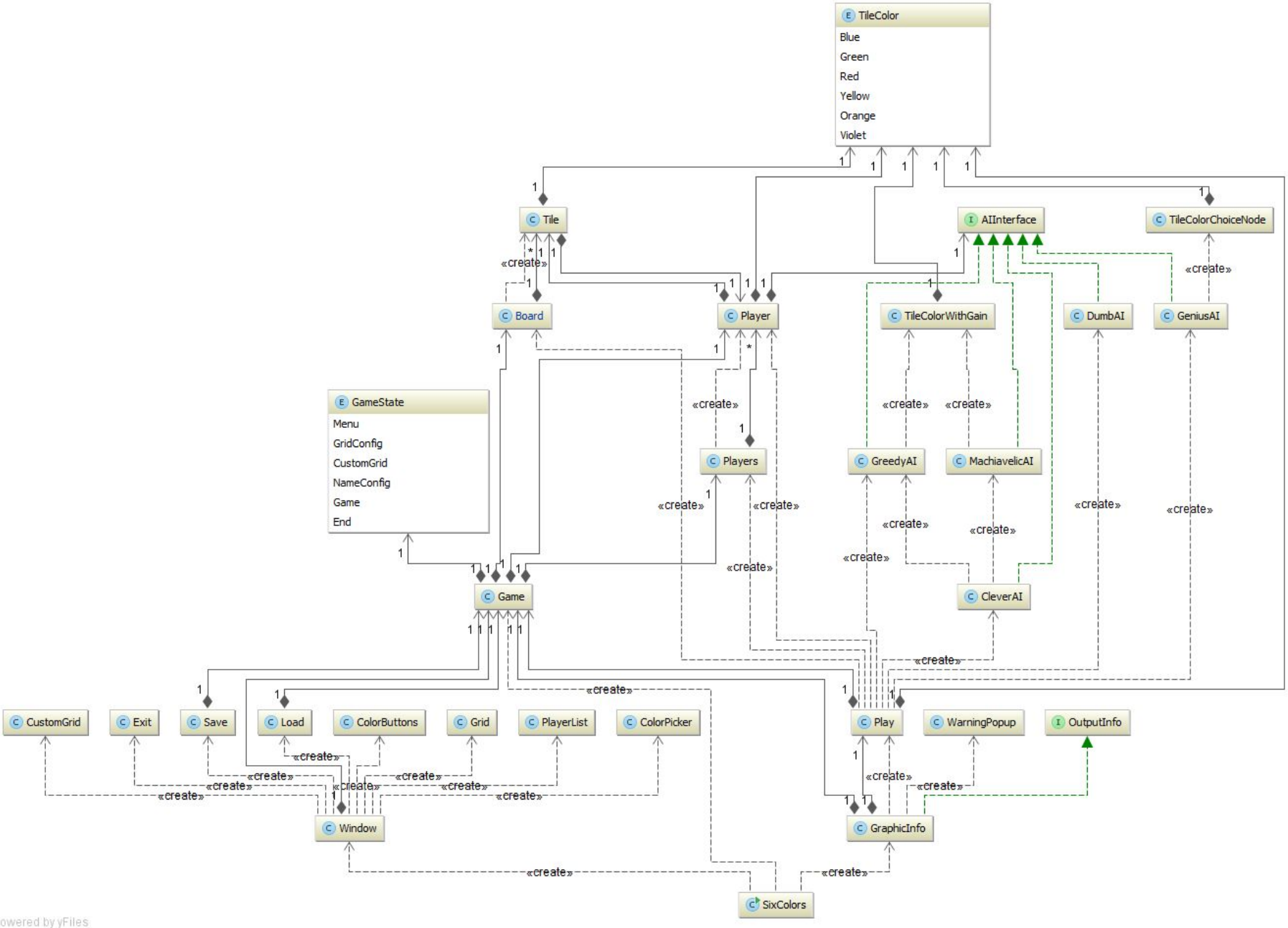
La profondeur de cette arbre, et donc la puissance de cette IA, est définie par la constante de classe DEPTH.

Conclusion

Avec presque 2500 lignes de code écrites, ce projet s'est montré de grande envergure. Nous avons pu travailler en collaboration sans problème à l'aide de l'outil de contrôle de versions git. Ce projet s'est montré très formateur, en nous imposant de concevoir en partant de zéro un projet, son architecture, et les algorithmes associés. Nous avons pour cela adopté le patron de conception MVC dès le début, ce qui nous a grandement facilité la tâche.

C'était également une bonne occasion de nous former de manière plus approfondie sur le langage orienté objet qu'est Java, ainsi que sur la bibliothèque Swing. Cela a nécessité de remettre régulièrement en question notre travail, et de communiquer de manière très suivie sur les implémentations que nous réalisions.

Nous avons également eu l'occasion d'expérimenter deux méthodes de développement en équipe, que sont la relecture de code croisée, et la programmation en binôme. Si ces méthodes ne sont pas aussi productives que lorsque chacun travaille de son côté, nous avons réalisé qu'elles nous permettait de produire du code de meilleure qualité, plus facilement.


TileColor
Blue
Green
Red
Yellow
Orange
Violet
Tile
AIInterface
TileColorChoiceNode
Board
Player
TileColorWithGain
DumbAI
GeniusAI
GameState
Menu
GridConfig
CustomGrid
NameConfig
Game
End
Players
GreedyAI
MachiavelicAI
CleverAI
Game
CustomGrid
Exit
Save
Load
ColorButtons
Grid
PlayerList
ColorPicker
Play
WarningPopup
OutputInfo
Window
GraphicInfo
SixColors
«create»
Powered by yFiles