Projet Application Conception

Tommy ANFOSSO

December 16, 2020

1 Fonctionnalités attendues

L'objectif de l'application est de réaliser un jeu quixo en utilisant un langage de programmation objet doté d'une intelligence artificielle (algorithme: min-max ou alpha-bêta). Pour ce faire, les méthodes liées à la programmation orientée objet et modélisation du logiciel devront être appliquées. Nous devons prendre en compte toutes les subtilités liées au jeu quixo (condition de déplacement des pions, de gain de la partie...). Le jeu devra être basé sur une modélisation UML proposée.

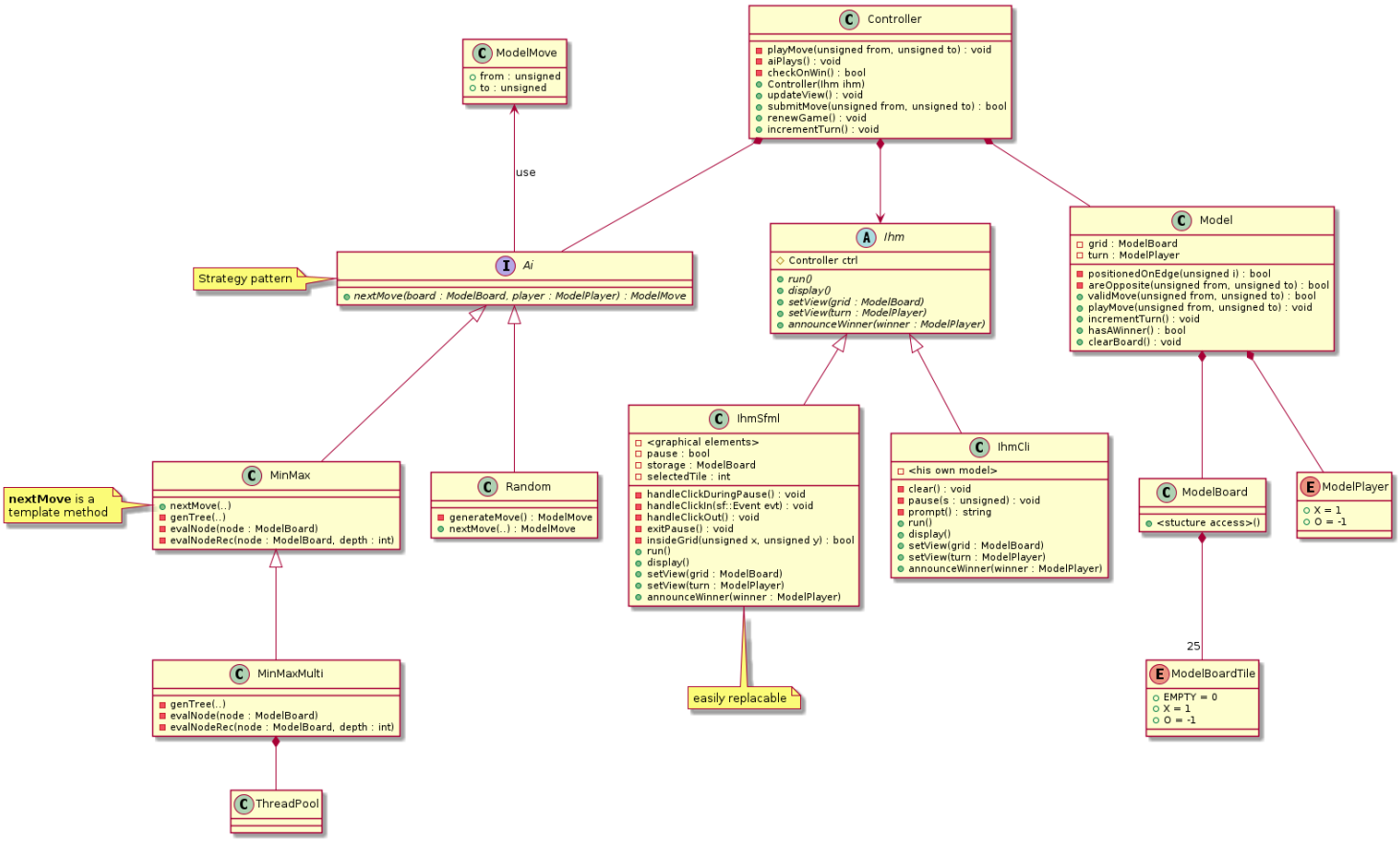
Les fonctionnalités suivantes sont attendues :

- Concevoir le moteur du jeu, c'est-à-dire l'échiquier, les pions (et leurs déplacements).
- Mettre en place 3 différents designs patterns vu en modélisation du logiciel
- Vérifier que le jeu se termine bien (partie gagnée ou perdu).
- Un affichage du jeu doit être prévu, et doit permettre l'interaction. Cela peut se traduire par un mode console (a minima) ou par une interface graphique (très fortement recommandé).
- Une intelligence artificielle basée sur un algorithme min-max ou alpha-bêta. A noter que l'algorithme min-max ou alpha-bêta devra être multi-threadé.

2 Modélisation

Faire une modélisation basée sur une architecture MVC nous permet, en tant que développeur, de pouvoir remplacer facilement la partie IHM (contenant les vues). On pourrait par exemple commencer par une interface minimaliste en ligne de commandes puis, s'il nous reste suffisamment de temps, faire une interface graphique sans avoir à adapter le code côté Contrôleur ou Modèle. Cela offre également la possibilité d'avoir plusieurs interfaces utilisant chacune une librairie graphique différente, et de choisir celle que l'on veut au lancement de l'application.

2.1 Diagramme de classes



Hiérarchie des classes:

- Les classes MVC : **Controller**, **Model**, **Ihm**
- **ModelBoard** et **ModelPlayer** sont les entités
- L'interface **Ai** pour générer des coups selon un algorithme
- Les classes **Random**, **MinMax** et **MinMaxMulti** sont des implémentations d'IA différentes

2.2 Design patterns

MVC La classe Controller joue le rôle de 'médiateur' entre le modèle et la vue. Les actions utilisateurs côté interface font appel aux use cases du contrôleur qui interagissent avec le modèle. En fin de use case, le contrôleur met automatiquement à jour la vue de l'interface. Par conséquent le modèle et l'interface sont indépendants et peuvent être remplacé individuellement sans casser l'application.

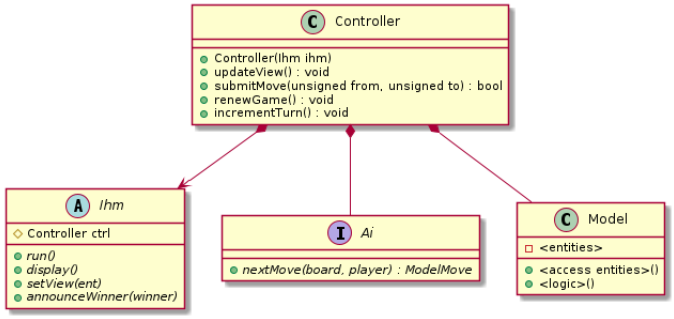


Figure 1: MVC

Strategy Le contrôleur fait appel à la méthode **nextMove(..)** de l'IA pour générer un coup, selon le type d'IA utilisé l'algorithme pour y parvenir est différent. Il est alors possible d'ajouter autant d'algorithmes que l'on veut sans modifier le code existant.

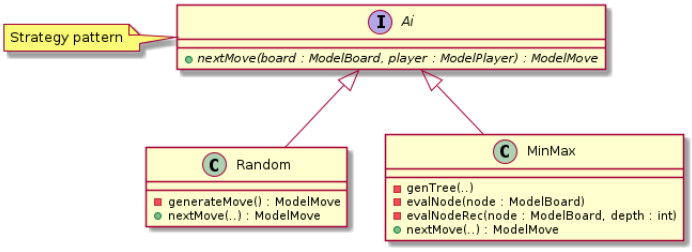


Figure 2: Strategy

Template Method Les stratégies MinMax et MinMaxMulti décrivent un même algorithme, seule l'implémentation de certaines étapes se voient changer. Le fait de découper l'algorithme en plusieurs étapes permet aux sous-classes d'implémenter leur version de l'algorithme tout en conservant sa forme. De ce fait, nous n'avons pas dupliquer de code et avons permis l'ajout d'implémentations supplémentaires de l'algorithme MinMax sans modification du code existant.

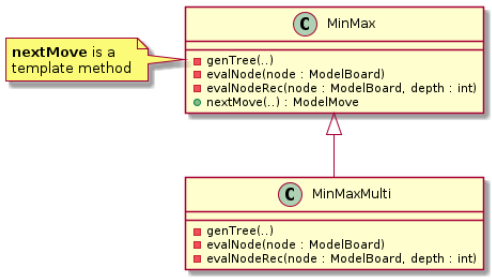


Figure 3: Template Method

State L'interface doit pouvoir annoncer le nom du gagnant en fin de partie. Le fait de mettre en pause l'interface gèle complètement celle-ci ce qui n'est pas le comportement attendu. Nous définissons alors un état de pause après l'annonce, dont l'on peut sortir via un événement spécifique.

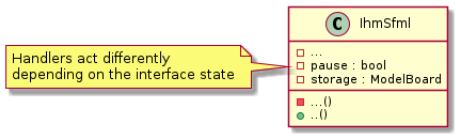


Figure 4: State

2.3 Diagramme séquence MVC

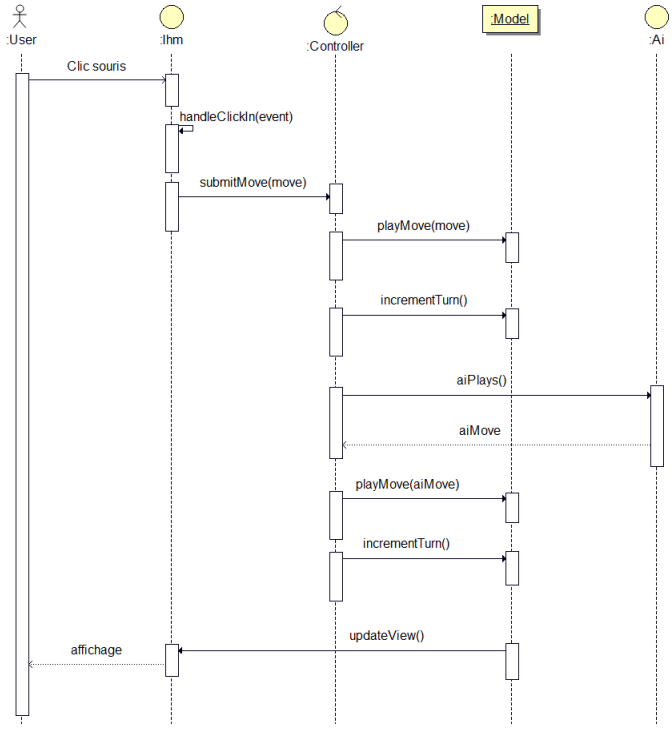


Figure 5: Diagramme séquence MVC

3 Implémentation

3.1 Choix techniques

Pour le développement de l'interface graphique, j'ai choisi d'utiliser la librairie multimédia SFML qui impose peu de contraintes vis à vis du code. La librairie est également multi-plateforme. Je n'ai utilisé aucune autre classe ou librarie externe à la STD de C++11. L'utilisation des pointeurs intelligents m'a permis de ne pas avoir à gérer moi même l'allocation et désallocation des objets, au risque d'oublis ou de causer des fuites. Pour ce qui est de l'algorithme, j'ai implémenté le MinMax plutôt que l'alpha-bêta sans vrai raison particulière. Ma fonction d'évaluation consiste à calculer la différence entre le nombre max de symboles alignés côté X et côté O.

3.2 Algorithme MinMax

Ci-dessous la méthode permettant de calculer récursivement la valeur d'un état du plateau de jeu :

```
int MinMax::evalFuncRec(Model::Board* node, unsigned depth)
{
    // générer les noeuds fils du plateau courant
    std::vector<Model::Board> children= this->genChildren(*node);

    // maximiser la valeur
    if(depth == 0)
    {
        // -infini
        int value = -9999;
        for(Model::Board& child : children)
            value= std::max(value, this->evalFuncRec(&child, 1));
        return value;
    }
    // minimiser la valeur
    else if(depth == 1)
    {
        // +infini
        int value = 9999;
        for(Model::Board& child : children)
            value= std::min(value, this->evalFuncRec(&child, 2));
        return value;
    }

     // si le noeud est une feuille de l'arbre, on évalue sa
    valeur
    else /* depth == 2 */
    {
        int eval = this->evalFunc(*node);
        if(eval > this->bestOutcome.second)
        {
            this->bestOutcome.first = node;
            this->bestOutcome.second = eval;
        }
        return this->evalFunc(*node);
    }
}
```

Listing 1: methode d'evaluation recursive

4 Améliorations possibles

Je n'ai pas eu le temps de me pencher davantage sur la partie algorithme de l'IA, il aurait fallut augmenter la profondeur de l'arbre généré dans l'algorithme MinMax pour voir le réel intérêt du multithread.