BOUSSIDAN Aaron, PERRIN Jérémie et REY Simon

Projet de Programmation 2 - Ksparov 1.0

Année 2016-2017, Semestre 2

Depuis les début de l'informatique moderne et le développement grand public des micro-processeur de nombreuses applications ont vu le jour pour permettre à tout le monde de jouer aux échecs.

En effet qui ne rêve pas de pouvoir s'entraîner sans relâche pour atteindre le niveau d'un Bobby Fischer, ou de pouvoir revivre des parties de légende comme la finale de 1985 opposant Kasparov à Karpov.

Vous faire vivre cela, et bien plus encore, a été notre leitmotiv durant le développement de cette application. Voici la première version, la version 1.0, du meilleur jeu d'échec jamais créé, voici Ksparov !

Plus sérieusement ce document a pour vocation de décrire les choix d'implémentation fait et de présenter les fonctionnalités de notre application.

Le plus grand jeu de l'esprit jamais inventé, plus vous l'apprenez, plus vous y prenez du plaisir.
Garry Kasparov

1 L'organisation générale du code

Le code de notre projet a été divisé en quatre parties (chacune dans un fichier) qui correspondent chacune à une fonction particulière du code. On retrouve donc :
- **draw.scala** qui comprend tout ce qui concerne l'affichage de l'interface du jeu
- **rules.scala** qui implémente toutes les règles du jeu pour chaque pièce
- **game.scala** qui gère le lancement du jeu et qui fait l'interface entre les règles et l'application
- **solve.scala** qui définit ce qui a trait à la résolution par l'intelligence artificielle

Nous allons donc à présent étudier en détail chacun de ces fichiers.

2 draw.scala ou l'interface utilisateur

Le code a ici été séparé en plusieurs objets, chaque objet comprend tout ce qui concerne l'affichage d'une fenêtre en particulier. Nous retrouvons donc un objet pour le menu initial, un pour le réglage des paramètres, un pour le plateau de jeu...

Les fonctionnalités de l'interface : L'interface a été créé de manière à pouvoir implémenter tout le projet, les trois parties, sans avoir à la modifier à chaque phase. Ainsi le jeu s'ouvre sur un menu général permettant pour le moment de lancer une partie, de modifier les paramètres et de quitter le jeu. Des boutons existent déjà pour le chargement d'une partie, les scores des joueurs... mais ne sont pas fonctionnels.

Le lancement d'une partie fait appel à un autre menu, celui qui permet de choisir le mode de jeu pour la partie : Humain/Humain, Humain/IA ou IA/IA. Après tout ces choix, l'échiquier apparaît enfin !

L'organisation générale d'un objet DrawQuelquechose : Chaque objet du fichier *draw.scala* a un nom construit comme *DrawUnechose* où *Unechose* est l'élément affiché par cet objet (*DrawParameters*, *DrawMenu*, *DrawBoard*...). Tous les objets sont construits de la même manière : tout les éléments d'une fenêtre sont des extensions de *GridPanel* qui sont ensuite organisés via un *BorderPanel* dans l'interface. Au final, toute fenêtre est découpée en carré de 80 pixel.

Pour avoir un code plus propre une classe commune à tous les objets a été créé : *BackgroundCase* qui étend un *GridPanel* et qui permet selon ses paramètres de dessiner un certain nombre de cases de fond.

Les paramètres : Tous les visuels des textures des cases et des pièces de jeu peuvent être modifiées dans les paramètres. La plupart des textures ne sont pas très esthétiques mais ceci est un autre problème. Les paramètres et leurs modifications sont enregistrés dans un fichier : *src/main/resources/Parameters* ce qui permet de conserver les choix précédent à chaque ouverture de l'application.

Le plateau de jeu : Le plateau de jeu est, heureusement, la fenêtre la plus complète, on y retrouve plusieurs éléments.

Tout d'abord le plateau, qui est constitué de case sur lesquelles les pièces se rajoutent via une icône. La particularité étant que le plateau ne dessine pasles pièces lorsque l'on en créé une nouvelle instance,

il faut appeler **DrawActions.draw_board** pour cela.

Sur chaque coté du plateau on retrouve un compte des pièces mortes pour chaque joueur. Ceci est fait par le dessin des pièces via *DrawActions.draw_board* pour lequel les pièces ayant des positions négatives sont mortes.

Sous l'échiquier on retrouve un simple encart qui permet d'afficher des messages, c'est ici que les échecs et échecs et mat s'affichent.

Finalement on retrouve en haut de la fenêtre un menu rapide qui permet pour le moment simplement de revenir au menu.

Les actions dynamiques de l'interface : Le plateau de jeu est plus ou moins dynamique pour afin en temps réel les mouvements des pièces, les messages et le nombre de pièce morte. Tout cela est défini dans l'objet *DrawActions*.

L'affichage des pièces passe par l'appel de la méthode *DrawActions.draw_board*. Cette méthode prend chaque pièce et définit l'icône correspondante sur la case des coordonnées de la pièces. Les pièces ayant des coordonnées négatives sont des pièces mortes et donc *draw_board* actualise le compte des pièces mortes à chaque fois qu'il en trouve une. La méthode fini par une actualisation de la fenêtre ce qui permet d'afficher les pièces sur le plateau et le compte des pièces mortes.

A chaque fin de mouvement, la méthode *Ksparov.play_move*, qui applique le déplacement, vérifie si un évènement s'est produit suite à ce mouvement (échec, échec et mat ou pat). Si elle détecte un évènement elle affiche le message correspondant, sinon et affiche un message indiquant que la main est pour le joueur suivant.

3 rules.scala ou l'implémentation des règles des échecs

Les fonctionnalités : Toutes les règles de base des échecs sont présentes, à l'exception de la prise en passant, de la promotion, et des règles conduisant à une égalité (l'égalité est partiellement codée pour les IA). Notamment, il est possible de roquer dans le jeu, qui est un des mouvements du roi !

Structure de rules : Toutes les pièces héritent d'une classe abstraite *Piece*, qui contient principalement une méthode utile pour le reste du jeu : la fonction *move*, qui effectue un mouvement.
Chaque pièce particulière (pion, tours,..) hérite ensuite de *Piece* et se distingue par un "pattern" particulier, qui traduit l'ensemble de ses mouvements possibles.

Des méthodes plus générales sont présentes dans *Aux*, et ce qui concerne la vérification de l'échec et mat appartient à *Checkmate*.

La fonction *move*, et ses fonctions auxiliaires : La fonction *move* sert à vérifier si un mouvement est valable, à l'appliquer si cela est le cas, et à modifier la plateau en conséquence (suppression de pièces, échec...).

Elle appelle pour cela une fonction *pre_move*, qui vérifie la faisabilité du mouvement sans modifier le plateau (avoir une telle fonction se révèle utile pour, par exemple, afficher l'ensemble des cases atteignables par une pièce). *pre_move* vérifie plusieurs critères : si le mouvement est conforme aux déplacements de la pièce et à quelques règles basiques (ceci est effectué par *Aux.checks_pre_move*), si le chemin que veut suivre la pièce est libre et s'il y a une pièce adverse à l'arrivée (effectué par *clear_path*), et si ce mouvement met un ou deux rois en échec (effectué par *check_check*).

Spécificité de certaines pièces : Le pion, qui est la seule pièce à avoir un déplacement non symétrique, possède un pattern un peu différent, qui est obligé de référer au joueur le contrôlant. Le cavalier, quant à lui, peut passer au dessus des autres pièces, et nécessite donc une modification de *clear_path*. Le roi, enfin, possède en attribut la liste des pièces le mettant en échec, des booléens indiquant s'il est en échec et s'il a bougé, et a une version de *move* différente pour gérer le cas du roque.

***check_ mate*, vérification de l'échec et mat :** Plutôt que de vérifier si tous les mouvements possibles laissent le roi en échec (ce qui aurait une complexité assez grande), la fonction *Checkmate.check_ mate* vérifie si le roi peut bouger, et si une autre pièce peut s'interposer sur le chemin de son attaquant, voir le prendre.

4 game.scala ou l'on peut enfin jouer

L'objet *Constants* : L'un des objets du fichier game.scala est *Constants*. C'est un objet qui contient toutes les variables qui sont utilisés par les différentes parties du code. On y retrouve donc la plupart des booléens nécessaires à l'exécution de la partie, mais aussi des constants comme les chemin vers les icônes, la couleur du text, la taille des cases... L'idée a été ici de regrouper tout ce qui est en commun avec plusieurs objets dans un seul et même objet.

Notons par ailleurs que bien qu'il s'appelle *Constants* une bonne partie des variables de cet objet sont variables.

Les joueurs : La classe *Player* est une classe abstraite qui contient l'identifiant du joueur (0 pour le noir et 1 pour le blanc), des booléens utiles pour le déplacement (*ai* et *move*) ainsi que la méthode *get_ move* qui permet le déplacement des pièces.

Cette classe est étendue en deux : *AI* pour les joueurs de type intelligence artificielle et *Human* pour les joueurs humains. La classe *AI* est présenté en section 5 sur le fichier solve.scala.

La méthode *get_ move* de la classe *Human* : Pour un joueur humain le déplacement se fait en deux étapes : sélection de la pièce et sélection de la case où se déplacer. La méthode *get_ move* s'applique pour une case sur laquelle le joueur a cliqué, elle commence par vérifier que la pièce sélectionnée appartient bien au joueur (méthode *isHis*). Si tel est le cas, le booléen *Constants.first_ choice_ done* passe à *true* ce qui ouvre la sélection de la case, et les cases possibles du déplacement se colorie en rouge par appel de la méthode *DrawActions.draw_ possiblie_ moves*.

Lorsque le joueur clique sur une deuxième case *get_ move* est à nouveau appelée, puisque l'on ne peut pas se déplacer sur une case contenant une de nos pièce, elle passe la première sélection (*isHis* renvoi false) et si le premier choix a été fait elle teste si la case est atteignable par la pièce dans ce cas elle applique le mouvement, sinon elle repasse *Constants.first_ choice_ done* à *false*.

L'objet *Ksparov* : C'est l'objet central de notre application, c'est lui qui définit tout et qui fait appel au reste du code. Il comprend plusieurs éléments différents.

L'interface graphique : La variable *frame* définie dans *Ksparov* est la variable de base de l'interface graphique, c'est elle dont le contenu change pour changer de fenêtre. La fonction *main* fait simplement appel à la méthode *application.main(Array())* qui lance l'application en elle même avec la première frame, à savoir le menu principal.

L'initialisation des pièces : On retrouve dans *Ksparov* la variable *board* qui est un tableau de 32 pièces contenant toutes les pièces du jeu, les 16 premières cases étant les pièces du joueur blanc et les 16 dernières celle du joueur noir. La méthode *Ksparov.init_ board* permet de définir le tableau initial avec toutes les pièces à leur position initiale.

L'initialisation du jeu : La méthode *Ksparov.init_ game* est appelée après le choix du type de jeu dans le second menu de sélection, elle initialise les variables nécessaires pour commencer une partie. Elle commence par initialiser les pièces (via *Ksparov.init_ board*) ensuite selon le type de jeu choisi elle définit les variables des joueurs correspondantes, enfin elle modifie les booléens aux bonnes valeurs. On peut noter ici que dans le cas d'une partie opposant une IA et un humain le choix de la couleur des joueurs est aléatoire.

Jouer un mouvement : Lorsqu'un joueur clique sur une case la méthode *Ksparov.play_ move* est appelée, c'est elle qui gère tous les déplacements, son exécution est une longue série de conditions imbriquées. Tout d'abord elle vérifie si la partie n'a pas déjà été gagnée par un des joueurs (le booléen *Constants.game_ won*), dans ce cas elle ne fait rien. En effet dans ce cas les cases ne doivent pas répondre

aux clics des joueurs.

Dans un second temps elle vérifie si l'IA n'est pas en pat à travers le booléen *Constants.game_ nulle*). En effet étant aléatoire l'IA finit souvent à n'avoir plus aucuns mouvements possibles, le pat a donc été implémenté pour l'IA, de manière à ne pas boucler à l'infini dans le choix du mouvement de l'IA.

Si ces deux conditions ne sont pas vérifiées *Ksparov.play_ move* appelle la méthode *get_ move* du joueur actif sur la case sélectionnée. Si à l'issu de *get_ move* le joueur s'est déplacé (booléen *moved* du joueur), il faut vérifier l'état de la partie après ce déplacement.

Il y a alors une série de test qui vérifient, dans l'ordre d'exécution, s'il y a échec et mat, si l'IA peut toujours se déplacer ou s'il y a échec. Dans chacun de ces cas un message particulier est affiché donnant l'évènement en question et les booléens correspondant sont mis à jour.

Si aucun de ces tests n'est validés, alors la main passe au joueur suivant et le booléen *Constants.first_ choice_ done* est remis à *false*.

Enfin si l'on dans le cas d'une partie entre un joueur et une IA et que le joueur courant (donc après déplacement du joueur qui vient de cliquer) est l'IA, on appelle directement la méthode *get_ move* de l'IA.

Il est important de noter que dans le cas d'une partie entre deux IA il faut cliquer sur une case pour obtenir le prochain mouvement, ce qui n'est pas le cas lors d'un matche entre une IA et un humain.

5 solve.scala ou l'intelligence limitée

L'intelligence artificielle : Pour le moment l'intelligence artificielle est encore pas très au point puisqu'elle choisit aléatoirement ses coups à chaque mouvement, mais elle permet de jouer contre quelqu'un. Son code a malgré tout été relativement optimisé.

La méthode *get_ move* de l'IA : L'IA choisit son coup aléatoirement parmi les différents mouvements possible. Elle commence donc par choisir une pièce aléatoirement entre ses pièces et elle choisit ensuite lequel de ses mouvements elle applique.

Plus précisément, la méthode maintient un tableau de booléen à jour définissant si la pièce sélectionnée à déjà été testée. Ce tableau est initialisée en retirant les pièces mortes pour ne pas perdre de temps. Un nombre aléatoire en 1 et 16 est tiré, si la pièce n'a pas été testée (case du tableau *already_ check* à *false*) alors la méthode cherche à savoir si un mouvement est possible.

Si un mouvement est possible, elle choisit alors une des cases possibles aléatoirement, sinon, si aucun mouvement n'a été appliqué, on vérifie s'il reste des mouvement possibles (s'il existe une case à *false* dans *already_ check*), si tel est le cas on boucle, sinon l'IA est en pat.

6 parse.scala ou les dangers de la PGN

L'objet *Save* : La sauvegarde de la partie en cours se fait via cet objet. Les méthodes sont internes pour la plupart. Il faut faire particulièrement attention à comment on récupère un mouvement. En effet, les deux méthodes à utiliser sont *add_ move1* et *add_ move2* et doivent être inserées dans la méthode *get_ move* de toutes les sous-classes de player afin de récuperer les mouvements les uns aprés les autres. La première doit être placé juste avant l'application du mouvement, la deuxième après si le mouvement a bien eu lieux. La méthode à appeler pour sauvegarder une partie est *write_ to_ file*.

L'objet *Load* : Cet objet a pour fonction de lire un fichier au format PGN (Portable Game Notation). Pour cela une méthode *get_ list_ move_ from_ file* permet de récuperer la liste des mouvements (non encore parsés). Une fois cette liste établie, une partie peut être lancée avec comme joueur des *Reproducers*. *Reproducer* est une sous-classe de player et est un robot, en utilisant les règles du jeu définies dans Rules, il comprend le PGN. Il commence son tour en lisant la tête de l'attribut *list_ of_ moves*, qui est un coup au format PGN. S'il y a ambiguïté il décide, en faisant appel a la méthode *pre_ move* des piéces, quelle piéce doit bouger et applique le mouvement.

7 Les défauts et erreurs connues

L'implémentation telle qu'elle est actuellement connaît certaines lourdeurs dans le code. En effet, pour les pièces le code utilise deux variables : une liste de 32 pièces et une liste de 64 cases qui sert à positionner chaque pièce de la première liste. Le code pourrait être modifier pour n'utiliser plus que la liste des 32 pièces.

De même pour chaque nouvelle fenêtre, chaque nouveau mouvement un grand nombre de classes sont instanciées, principalement dans la partie graphique, ce qui peut être lourd en mémoire.

En ce qui concerne rules, l'implémentation du roque est quelque peu lourde, et demande quelques répétitions de code.

Les pions promus ne comptent pas comme pions morts mais c'est logique connard.