IFT232

Projet d'intégration

Partie 1

Ce projet comporte plusieurs exercices difficiles visant à vous faire réviser la plupart des notions du cours. Les premiers numéros concernent une amélioration de la qualité du code, en vue d'implanter des fonctionnalités plus intéressantes en se dotant d'une version du code de base facile à étendre et tester. Les numéros finaux concernent l'intégration de nouvelles fonctionnalités et touchent les patrons de conception.

Mise en contexte

Ce projet concerne l'écriture d'un jeu d'échecs rudimentaire.

Pour l'instant, le jeu est capable de présenter une planche, de l'initialiser avec des pièces et de vous laisse déplacer les pièces presque n'importe où.

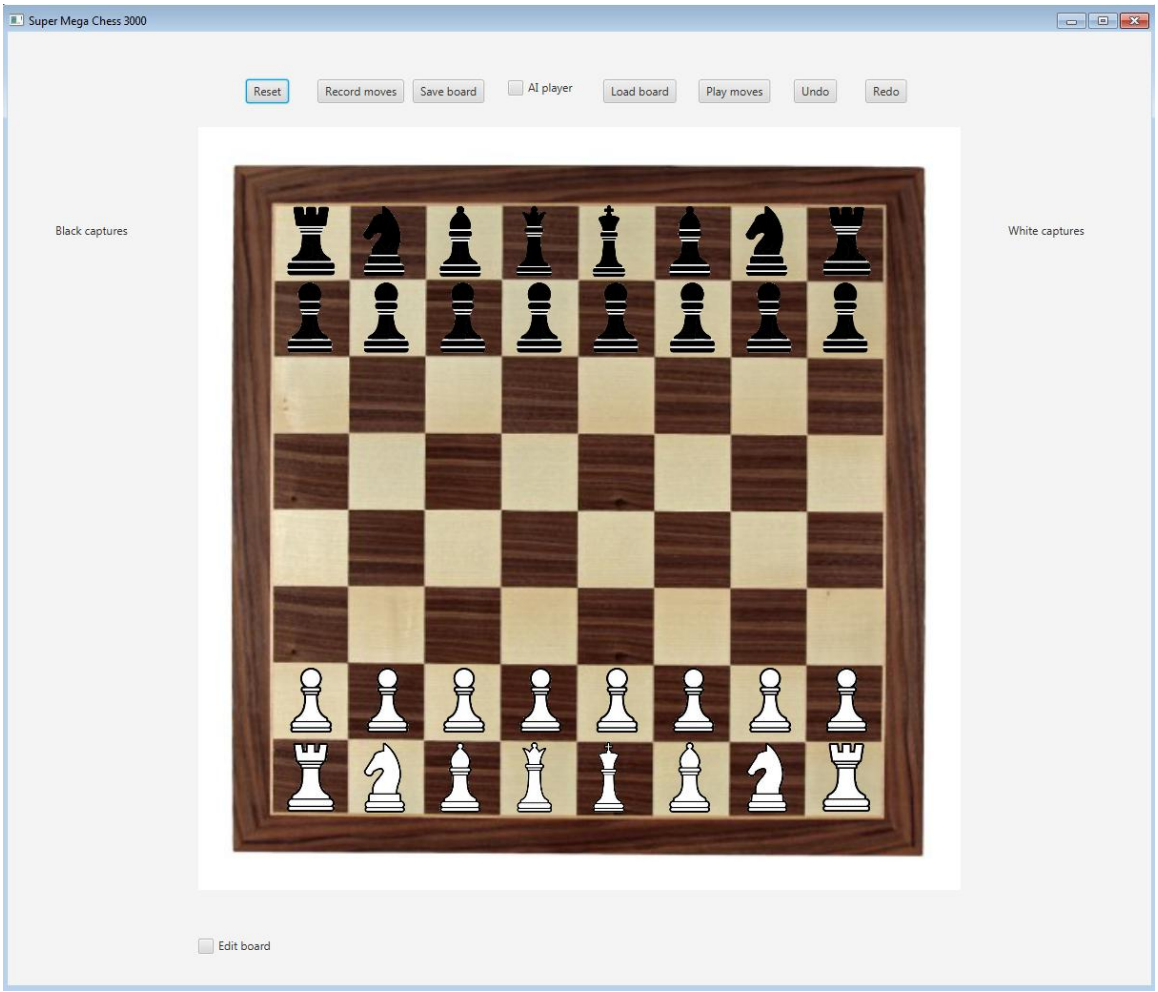
Le code a été écrit jusqu'à l'atteinte d'un point critique : les règles de déplacement de pièces sont sur le point d'être implantées, mais l'auteur a soudainement été saisi d'une atroce migraine!

Quelques vérifications sont faites dans la méthode ***move*** de la classe **ChessBoard**, pour empêcher la superposition de pièces, mais les règles concernant les déplacements des pièces seront difficiles à implanter et à tester avec le code dans cet état.

Il faudra donc d'abord réorganiser le code pour rendre plus facile l'intégration des règles. Pour vérifier leur bon fonctionnement, il faudra faciliter l'écriture et la génération de scénarios de tests. Une fois l'ensemble des règles correctement intégrées et testées, il deviendra possible d'écrire une intelligence artificielle rudimentaire et de la tester correctement.

Quelques contrôles ont été prévus dans l'interface de base :

- **Reset** : replace les pièces à la position de départ.
- **Record moves** : permettra l'enregistrement des séquences de mouvements, sera utilisé pour générer des scénarios de tests.
- **Save board** : permettra l'enregistrement de l'état de la planche de jeu, sera utilisé pour générer des scénarios de tests.
- **AI Player** : sera utilisé pour activer le joueur artficiel.
- **Load board** : sera utilisé pour charger une planche de jeu d'un fichier.
- **Play Moves** : sera utilisé pour exécuter une suite de mouvements à partir d'un fichier.
- **Undo et Redo** : sera utilisé pour défaire et refaire des mouvements.
- **Edit board** : sera utilisé pour ajouter des pièces sur la planche de jeu.



De plus, il est présentement possible de faire glisser les pièces à l'aide de la souris pour réaliser des déplacements.

Exercice 0 : Visite dans le code

Les classes sont organisées plutôt simplement pour implanter les fonctionnalités de base du jeu. C'est à peine orienté-objet.

ChessUtils : boîte à outils, sert à convertir la notation algébrique pour les noms de pièces et les coordonnées en caractéristiques de pièces et positions sur une planche de jeu.

ChessGame : application principale, contient la fenêtre d'interface de base avec ses contrôles et une planche de jeu.

ChessBoard : représente la planche de jeu, avec une grille de pièces et des règles de mouvement fondamentales.

ChessPiece: une pièce de jeu. Identifiée par un type et une couleur, connaît sa position dans la grille de jeu. Sa représentation visuelle sait détecter sont déplacement dans la grille de jeu.

Exercice 1 : Découplage de l'interface et du modèle

L'état actuel du code est analogue à ceci :

Présentement, les classes **ChessPiece** et **ChessBoard** représentent des parties critiques du jeu en plus d'éléments d'interface graphique. Ceci implique qu'on ne peut pas facilement manipuler le jeu sans manipuler l'interface graphique, et vice-versa. Avant d'ajouter et de tester de

nouvelles fonctionnalités, nous allons découpler l'interface graphique et le modèle du jeu.

Pour ce faire, il faut créer un nouveau package (chess.ui) ainsi que deux classes : ***BoardView*** et ***PieceView***, qui représenteront uniquement les éléments d'interface graphique.

Plusieurs éléments de **ChessBoard** doivent être déplacés vers BoardView pour réaliser le découplage. Entre autres :

- les constantes de taille
- les propriétés *startX*,*startY*, *boardPane* et *boardView*
- les méthodes de conversion de coordonnées (*gridToPane* et *paneToGrid*)
- la presque totalité du constructeur

Le ChessBoard contiendra maintenant un BoardView, et redirigera les appels concernant des méthodes de BoardView vers celle-ci. Entre autres, l'obtention du panneau d'interface (getPane()) devrait maintenant s'appeler getUI() et rediriger l'appel vers BoardView.

Ensuite, plusieurs des éléments de **ChessPiece** doivent être déplacés vers PieceView. Entre autres :

- la constante *pieceSize*
- les constantes *names* et *prefixes*
- l'attribut *board*
- l'attribut *piecePane*
- la méthode *enableDragging*(), contenant toute la gestion des actions de souris
- une grande partie du constructeur

Une ChessPiece contiendra maintenant une PieceView, qu'elle est responsable d'initialiser. Des redirections seront nécessaires vers les méthodes de PieceView, comme remplacer *getPane*() par *getUI*() (idem qu'avec ChessBoard).

Un problème de couplage grave demeure cependant, il se trouve dans enableDragging().

Le code associé aux événement MousePressed et MouseReleased doit être modifié, parce qu'il a besoin de trop d'information provenant de ChessBoard.

En réalité, comme le ChessBoard contient des références aux pièces se trouvant dans chaque case, vous pouvez laisser le soin au ChessBoard de déduire quel mouvement a été tenté lorsque la pièce est relâchée, si vous disposez des coordonées originales en pixels ainsi que des coordonnées où la pièce a été relâchée, également en pixels.

À l'instar de la variable *mouseAnchor*, vous pouvez enregistrer la position originale de la pièce lors de l'événement *MousePressed*, puis demander au ChessBoard de tenter le mouvement lors du *MouseReleased*, en lui passant les coordonnées initiales et finales en pixels. Le ChessBoard et sa BoardView correspondante peuvent déduire avec quel endroit de la grille on interagit et donc, quelles sont les pièces concernées.

Ajoutez donc la méthode ChessBoard.move() qui prend des Point2D en paramètre, servant de coquille qui effectue les conversions de coordonnées pour appeler la méthode move originale, puis basculez-lui la responsabilité de mettre à jour la pièce déplacée, s'il y a lieu. Ceci devrait grandement simplifier le code de gestion des déplacements dans l'interface.

La partie de déplacement concernant l'image de la pièce devrait être déléguée à ChessPiece, puis à PieceView.

Après ces modifications, le jeu devrait fonctionner comme avant. Déplacez quelques pièces et tentez de les superposer, pour voir si les mêmes règles qu'avant s'appliquent.

Exercice 2 : Chargement et sauvegarde de planches de jeu

Avant d'intégrer les règles du jeu à notre programme, nous allons devoir nous doter d'une façon de tester les planches de jeu facilement.

Si on peut écrire des descriptions de planches de jeu dans des fichiers et les charger dans le jeu, alors on peut demander à un collègue de créer du matériel de test en éditant les fichiers!

Cependant, afin que personne ne sente le besoin de jouer les singes saisisseurs de données, il serait préférable de permettre la sauvegarde de grille, afin de pouvoir facilement générer des mises en situation pour tester les règles du jeu, directement à l'aide de l'interface du jeu.

Pour commencer, vous pouvez ouvrir le fichier **normalStart** du dossier **boards** dans l'éditeur de texte d'Eclipse pour voir comment une planche de jeu est décrite dans un fichier. Les pièces sont placées sur la planche à l'aide de la notation dite algébrique, qui fonctionne comme ceci :

- La colonne se dénote avec une lettre (**a** à **h** inclusivement);
- La rangée se dénote avec un chiffre (1 à 8, 1 étant en bas de la planche);
- Une pièce se dénote avec une lettre pour la couleur et une pour le type;
- Les couleurs sont **w** pour white (blanc) et **b** pour black (noir).
- Les types sont **p** pour **p**awn (pion), **n** pour k**n**ight (cavalier), **b** pour **b**ishop (fou), **r** pour **r**ook (tour), **q** pour **q**ueen (reine), **k** pour **k**ing (roi).

La planche décrite par le fichier normalStart est une planche de jeu habituelle pour démarrer la partie. Une grille n'a pas besoin de contenir toutes les pièces, et peut contenir un nombre incorrect de pièces également.

a) Implantez la méthode readFromFile dans la classe ChessBoard. Cette méthode devrait produire une planche de jeu à partir d'un fichier. Elle est présentement appelée par une utilisation du bouton « Load board», mais ne fait rien. Cette méthode devrait créer un ChessBoard et y placer les pièces lues dans le fichier, aux coordonnées spécifiées. Vous devriez utiliser un objet **Scanner** pour lire le fichier, sa méthode **next**() lit des chaînes de caractères, détectant les séparations avec des espaces ou des sauts de ligne.

b) Implantez la méthode ***readFromStream****()* de la classe ChessPiece. Cette méthode servira à convertir une ligne du fichier en ChessPiece. Vous pouvez facilement convertir une ligne du fichier en données utiles en séparant les coordonnées de la description de la pièce à l'aide de le méthode ***substring***().

c) Une des méthodes de ChessPiece peut maintenant être éliminée (regardez dans ChessBoard.resetGame()…).

d) Implantez la méthode ***saveToFile*** de la classe ChessBoard. Pour ce faire, il est recommandé d'utiliser un objet **FileWriter**. Cette méthode devrait procéder à l'enregistrement de chaque pièce présente sur la grille dans le fichier.

e) Implantez la méthode ***saveToStream*** de ChessPiece. Cette méthode devrait convertir la situation actuelle de la pièce en notation algébrique et l'écrire dans le fichier. N'oubliez pas les méthodes de *ChessUtils* pour réaliser cela.

N.B. Vos méthodes des parties **a**,**b**,**d** et **e** peuvent lancer des exceptions, celles-ci seront récupérées par l'interface graphique, qui affichera un dialogue d'erreur le cas échéant.

Exercice 3 : Premier scénario de test

C'est maintenant le bon moment pour écrire un premier test. On peut démarrer avec une planche initiale, déplacer une pièce, enregistrer la planche et ensuite, écrire un test qui compare la planche initiale et la planche finale.

a) En guise de préparation, il faut maintenant pouvoir créer un jeu d'échecs sans interface graphique, pour pouvoir rouler des tests. Pour ce faire, il faut terminer notre découplage entre l'interface et le modèle, le dernier nœud étant situé au niveau de **ChessGame**, qui mélange éléments d'interface et fonctionnalités du jeu.

Faites une opération semblable à celle du premier exercice dans la classe ChessGame pour créer une classe **GameView**, qui sera la partie interface graphique du jeu. Cependant, le programme devra tout de même démarrer

avec la classe contenant l'interface (maintenant GameView), celle-ci doit demeurer le programme principal.

Ceci implique que GameView contient un ChessGame, et non le contraire. ChessGame devrait contenir relativement peu de code pour l'instant, mais permettra à des classes de test de créer un jeu d'échecs et de charger des planches de jeu sans passer par l'interface graphique.

Présentement, la mise à jour de l'interface est faite dans la méthode **move (Point2D, Point2D)**. Il serait préférable de séparer l'exécution du mouvement de la mise à jour de l'interface. Déplacez donc le code qui ne concerne pas l'interface graphique dans une méthode **move(Point,Point)**, qui recevra les coordonnées converties. L'ancienne méthode ne contiendra plus que la conversion des coordonnées, une redirection à la nouvelle méthode, et les lignes de code qui mettent à jour l'interface graphique après.

b) Écrivez une première classe de test à l'aide de JUnit, se nommant **MoveTest**, que vous devriez placer dans le package **chess**.**tests**.

Une première méthode de test pourrait prendre la forme suivante :

```
@Test
public void testBasicCollision() throws Exception {

	ChessGame game=new ChessGame();
	game.loadBoard("boards/normalStart");

	ChessBoard result= ChessBoard.readFromFile("boards/normalStart");
	//Move tower over a pawn of the same color
	game.movePiece("a1-a2");

	assertTrue(game.compareBoard(result));

}
```

Ici, on a tenté de déplacer un pion sur la case occuppée par une tour de la même couleur, ce mouvement devrait échouer avec les règles actuellement implantées.

c) Pour que ce test fonctionne, vous devez améliorer l'interface que ChessGame fournit au monde extérieur. Tout d'abord, il faudrait être capable de commander des mouvements en format algébrique directement à la classe ChessGame (méthode ***movePiece***()). Ensuite, il faudrait être capable de comparer deux planches de jeu pour voir si elles sont identiques. La méthode ***compareBoard***() de ChessGame devrait faire cette vérification, cependant, il en découle qu'on devra implanter une méthode ***equals***() dans ChessBoard ainsi que dans ChessPiece.

d) Rédigez des fichiers contenant une seule pièce, placée au centre de l'échiquier, pour chaque type de pièce. Ces fichiers serviront de base pour tester les mouvements des pièces. ***Note: déjà fait pour quelques pièces.***

e) Pour chaque pièce, chargez le fichier correspondant dans l'interface, déplacez la pièce à un emplacement légal, puis sauvegardez la planche de jeu. Répétez le processus pour plusieurs mouvements différents, qui offriraient une couverture intéressante du comportement de la pièce. ***Note: déjà fait pour quelques pièces.***

f) Écrivez une méthode de test pour chaque pièce, visant à tester les mouvements. Utilisez les planches de jeu de départ de chaque pièce et les planches de jeu sauvegardées pour vérifier que la pièce peut être correctement déplacée à cette position. ***Note: déjà fait pour quelques pièces.***

g) Finalement, dans les méthodes de test pour chaque pièce, écrivez des tests avec des mouvements qui devraient être illégaux pour la pièce en question. Le test devrait vérifier que la planche de jeu finale devrait être la même que la planche initiale, mais elle ne le sera pas, parce que le jeu ne vérifie pas encore les règles de mouvement des pièces. ***Note: déjà fait pour quelques pièces.***

Exercice 4 : Règles de déplacement de pièces simples

À l'aide des tests que vous avez pris le temps de générer, vous pouvez maintenant écrire le code vérifiant les règles de base pour les pièces.

L'endroit de prédilection pour ajouter le code de vérification semble être la méthode **move** de ChessBoard…

La méthode actuelle effectue un certain nombre de vérifications, puis exécute le mouvement si le mouvement est légal. Avant de procéder à l'implantation des règles spécifiques, il est impératif de nettoyer ce code.

Tout d'abord, il faudrait encapsuler les actions sur l'attibut ***grid*** dans un certain nombre de fonctions , représentant plus clairement les diverses actions qu'on fait :

- *assignSquare(Point,ChessPiece)* place une pièce à une position dans la grille
- *clearSquare(Point)* place une pièce vide à une position dans la grille
- *removePiece(Point)* retire la pièce se trouvant à une position et la retire également de l'interface graphique.
- *getPiece(Point)* obtient la pièce se trouvant à la position donnée

Une fois ces méthodes écrites, la méthode move(Point,Point) peut être réécrite plus proprement, et on peut lui ajouter un appel ressemblant à ceci :

```
ChessPiece toMove = getPiece(startPos);
if (!toMove.verifyMove(startPos, endPos)) {
     return false;
}
```

La méthode verifyMove déterminerait si la pièce accepte de faire le mouvement en question. L'implantation des règles se ferait donc dans cette méthode.

Pour l'instant, un switch-case sur le type de pièce fera parfaitement l'affaire. Les règles de base, pour chaque pièce, de façon sommaire, sont principalement sensibles à la distance parcourue en x (***deltax***) et la distance parcourue en y (***deltay***) :

- Pion : ***deltax*** est nulle et ***deltay*** est +1 (pour les noirs) et -1 (pour les blancs)
- Chevalier : des deux distances, ***deltax*** et ***deltay***, une doit être 2 et l'autre 1.
- Fou : ***deltax*** et ***deltay*** doivent être égales.
- Tour : des deux distances, ***deltax*** et ***deltay***, une doit être 0 et l'autre 1 à 7.
- Reine : Si le fou ou la tour peut le faire, la reine aussi.
- Roi : toute combinaison de ***deltax*** et ***deltay*** valant 0 ou 1.

Maintenant, tous les tests que vous aviez préparés à l'étape précédente devraient fonctionner correctement.