

Rapport Java

Jeu d'échecs



Corentin Prigent
Félix Baubriaud

Année 2023/2024

Table des matières

Table des matières.....	2
Introduction.....	3
1. Présentation des règles du jeu.....	4
2. Présentation du jeu reproduit.....	6
2.1. Description générale.....	6
2.2. Partie Git et base de données.....	7
2.3. Exemples d'utilisation.....	9
3. Développement du jeu.....	18
3.1. Description des classes.....	20
Classe "Case"	20
Classe "Echiquier"	20
Classe "Pièce mobile"	20
Classe "Tour"	21
Classe "Cavalier"	21
Classe "Fou"	22
Classe "Dame"	22
Classe "Roi"	22
Classe "Pion"	22
Classe "Partie"	23
Classe "Fenêtre principale"	24
Classe "Fenêtre dialogue"	24
Classe "Connexion BDD"	25
Classe "Mon Panel"	26
3.2. Description du programme principal.....	26
3.3. Description de certaines méthodes.....	28
Méthode manger.....	28
Méthode dessiner.....	28
Méthode deplaPossib.....	29
Méthode déplacer.....	31
Conclusion.....	32

Introduction

Dans le cadre de notre formation d'ingénieur à Polytech Clermont, nous avons entrepris un projet universitaire durant le premier semestre de notre 5ème année en Ingénierie Mathématiques et Data Science. L'objectif principal de ce projet était de développer un jeu d'échecs en utilisant le langage Java, avec l'environnement de développement IntelliJ.

Ce rapport détaillera les différentes phases de notre projet, mettant en lumière les choix de conception, la structure du code, ainsi que les défis auxquels nous avons été confrontés et les solutions que nous avons apportées tout au long du processus de développement.

Dans la première partie, nous présenterons les règles classiques du jeu d'échecs. La deuxième partie sera consacrée à la présentation du jeu que nous avons implémenté, avec une introduction à Git, soulignant son importance et expliquant son fonctionnement, et un exemple détaillé du déroulement d'une partie. Par la suite, nous examinerons en détail le code du jeu, en mettant en évidence le rôle des différentes classes, comment est programmé le jeu et on se concentrera sur certaines méthodes importantes. En conclusion, nous dresserons un bilan de notre expérience de développement du jeu d'échecs.

1. Présentation des règles du jeu

Les échecs se jouent sur un échiquier carré divisé en 64 cases de couleur claire et sombre. Chaque joueur commence avec 16 pièces : un roi, une dame, deux tours, deux cavaliers, deux fous et huit pions.

L'objectif ultime est de mettre le roi adverse en échec et mat, c'est-à-dire de le cerner de telle sorte qu'aucun déplacement des pions adverses ne puisse le sauver.

Pièces et leur mouvement :

- **Pion** : Avance d'une case vers l'avant, capture en diagonale, et a une option de double avancement au premier coup.
- **Tour** : Se déplace horizontalement ou verticalement sur n'importe quelle case disponible, non cachée par d'autres pièces.
- **Cavalier** : Se déplace en L, en sautant par-dessus d'autres pièces.
- **Fou** : Se déplace en diagonale sur n'importe quelle case disponible, non cachée par d'autres pièces.
- **Reine** : Combine les mouvements de la tour et du fou.
- **Roi** : Se déplace d'une case dans n'importe quelle direction.

Début de la partie: Les pièces sont disposées sur les deux premières rangées (voir exemples d'utilisation). Les blancs commencent à chaque fois la partie (règle universelle).

Il existe d'autres mouvements spéciaux:

- **Roque** : Mouvement spécial impliquant le roi et l'une des tours sous certaines conditions (pas de pions entre le roi et la tour, tous deux n'ayant pas bougé depuis le début de la partie et roi qui n'est pas en échec). Le roi avance alors de deux cases vers sa tour, et celle-ci passe derrière son roi.

- **En Passant** : Possibilité pour un pion simple de capturer un autre pion simple qui a avancé de deux cases comme s'il n'avait avancé que d'une case.

Promotion du Pion : Un pion atteignant la rangée la plus éloignée est promu en une pièce de son choix entre Cavalier, Fou, Tour et Reine. Il peut y avoir plus de 2 Reines dans un même camp.

Fin de la partie:

- **Déclarer forfait**
- **Pat** (*stalemate* en anglais) : Si un joueur n'a aucun coup légal et que son roi n'est pas en échec, la partie est déclarée *nulle*.
- **Échec et mat** : le roi est mis en échec et ne peut ni être sauvé par une autre pièce ni bouger.

2. Présentation du jeu reproduit

2.1. Description générale

Le jeu dispose de toutes les règles générales des échecs, promotion, roque et “en passant” inclus. Il n’est pas possible de bouger un pion sur une case qui ne lui est pas accessible, ou si cela met en échec son roi. Lorsque son roi est en échec, seuls les déplacements “sauvant” son roi sont autorisés. Si plus aucun déplacement n’est possible, la partie s’arrête, avec un gagnant par échec et mat s’il y a échec, et égalité (“pat”) sinon.

Un score s’affiche également à la fin de la partie. Ce score est calculé de la manière suivante :

- 30 points par joueur au début de la partie.
- 1 point de moins par déplacement. Ainsi plus la partie se termine rapidement plus le score est élevé.
- 2 points de plus par pion adverse mangé.
- 1 point de plus par promotion effectuée.
- 10 points de plus pour le gagnant à la fin de la partie, aucun s’il y a égalité.

Plusieurs options sont possibles lors de la partie en plus du jeu de base. Il est possible de déclarer forfait, ce qui met fin à la partie et déclare le joueur adverse vainqueur. Il est aussi possible de recommencer la partie ou de la quitter. Enfin, c’est possible d’accéder aux paramètres du jeu, afin de configurer la position et la taille de la fenêtre ainsi que de l’échiquier, et également d’indiquer si on veut enregistrer les scores des joueurs dans la base de données, dont on va parler ci-dessous.

2.2. Partie Git

Le développement logiciel est un processus dynamique et évolutif, nécessitant une coordination entre les membres d'une même équipe. Au fil du temps, la nécessité de suivre et de contrôler les modifications apportées au code source a conduit à l'émergence de solutions de gestion de versions. Parmi ces solutions, Git a révolutionné la manière dont les projets logiciels sont gérés et collaborés.

Principales caractéristiques de Git:

- **Distribution** : Git est un système distribué, ce qui signifie que chaque développeur a une copie complète de l'historique du projet sur sa machine locale.
- **Historique en arbre** : Les changements dans le code source sont enregistrés dans un arbre de commits. Chaque commit représente une version spécifique du code
- **Branching et merging** : Git facilite la création de branches (ramifications) pour le développement parallèle. Ces branches peuvent ensuite être fusionnées pour intégrer les changements dans une branche principale.
- **Staging Area** : Avant de valider un changement, il doit être ajouté à la "Staging Area". Cela permet de sélectionner les modifications spécifiques à inclure dans le prochain commit.

Principaux concepts de Git:

1. **Repository (Dépôt)** : C'est la base de données Git qui stocke toutes les versions du code, ainsi que les métadonnées nécessaires.
2. **Commit** : Un commit est une capture instantanée d'une version particulière du code. Chaque commit a un identifiant unique et contient les changements apportés par rapport à la version précédente.
3. **Branch (Branche)** : Une branche est une ligne de développement indépendante. Elle permet aux développeurs de travailler sur des fonctionnalités ou des corrections de bugs sans affecter la branche principale (généralement appelée "master" ou "main").

4. **Merge (Fusion)** : La fusion est le processus d'intégration des modifications d'une branche dans une autre. Git gère les fusions automatiques lorsque possible, mais parfois une intervention manuelle est nécessaire.
5. **Pull Request (Demande de tirage)** : Une demande de tirage est une fonctionnalité courante sur les plateformes d'hébergement de code telles que GitHub. C'est un moyen pour les développeurs de proposer des changements et de demander à l'équipe principale de fusionner ces changements dans la branche principale.

Principales commandes Git:

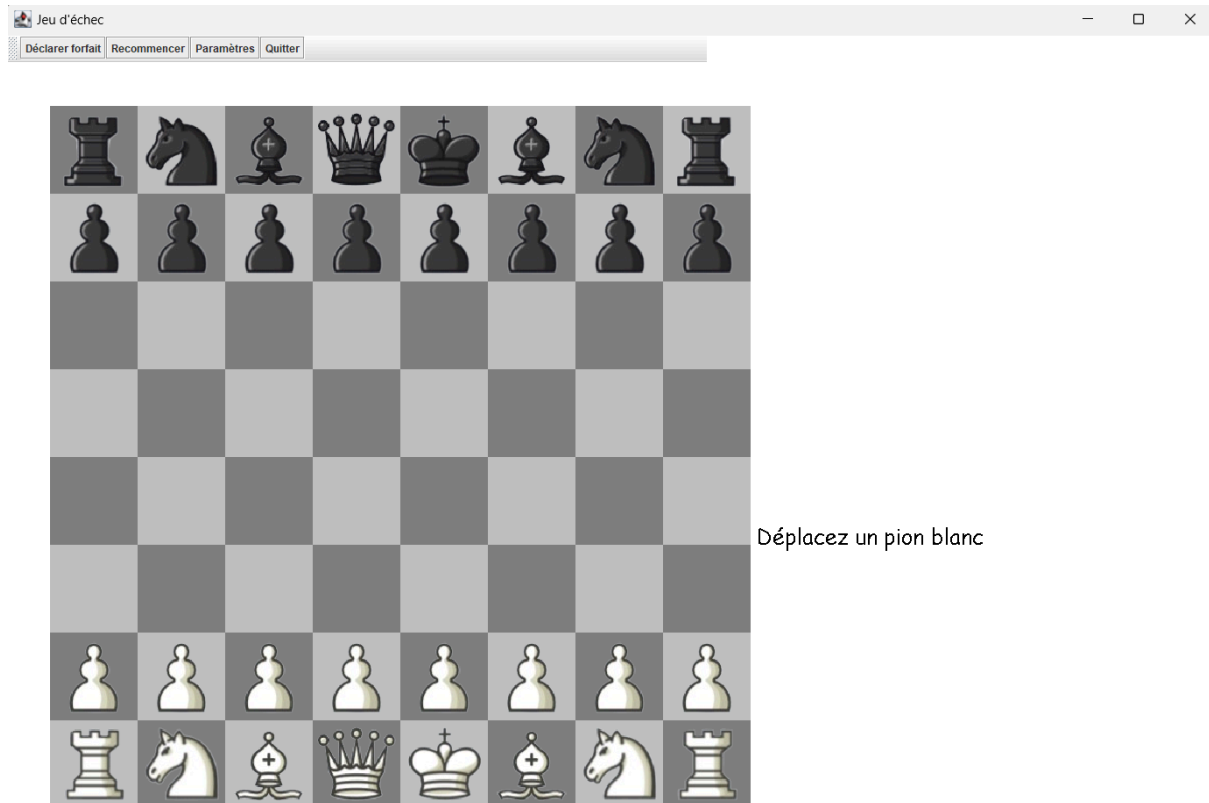
- **'git init'** : Initialise un nouveau dépôt Git.
- **'git clone'** : Clone un dépôt existant sur votre machine.
- **'git add'** : Ajoute des modifications à la Staging Area.
- **'git commit'** : Crée un nouveau commit avec les changements de la Staging Area.
- **'git push'** : Envoie les commits vers un dépôt distant.
- **'git pull'** : Récupère les dernières modifications depuis un dépôt distant.
- **'git branch'** : Gère les branches du dépôt.
- **'git merge'** : Fusionne les changements de deux branches.
- **'git log'** : Affiche l'historique des commits.
- **'git status'** : Affiche l'état actuel du dépôt.

Cycle de travail avec Git:

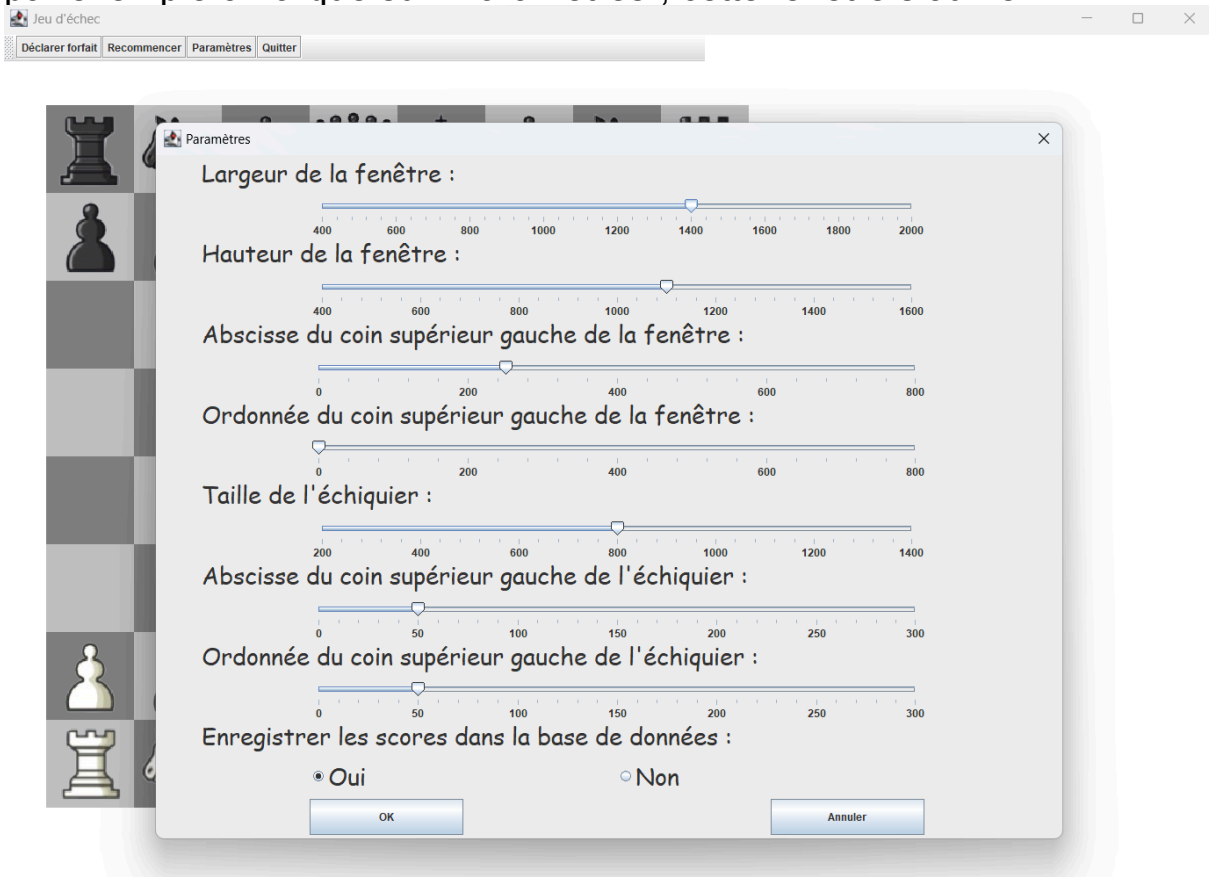
1. **Initialisation** : 'git init' pour créer un nouveau dépôt ou 'git clone' pour copier un dépôt existant.
2. **Modification du code** : Modifier les fichiers du projet.
3. **Staging** : Utiliser 'git add' pour ajouter les modifications à la Staging Area.
4. **Commit** : Utiliser 'git commit' pour créer un commit avec les modifications ajoutées.
5. **Push** : Utiliser 'git push' pour envoyer les commits vers un dépôt distant.

2.3. Exemples d'utilisation

Au lancement d'une partie, on obtient cette fenêtre :

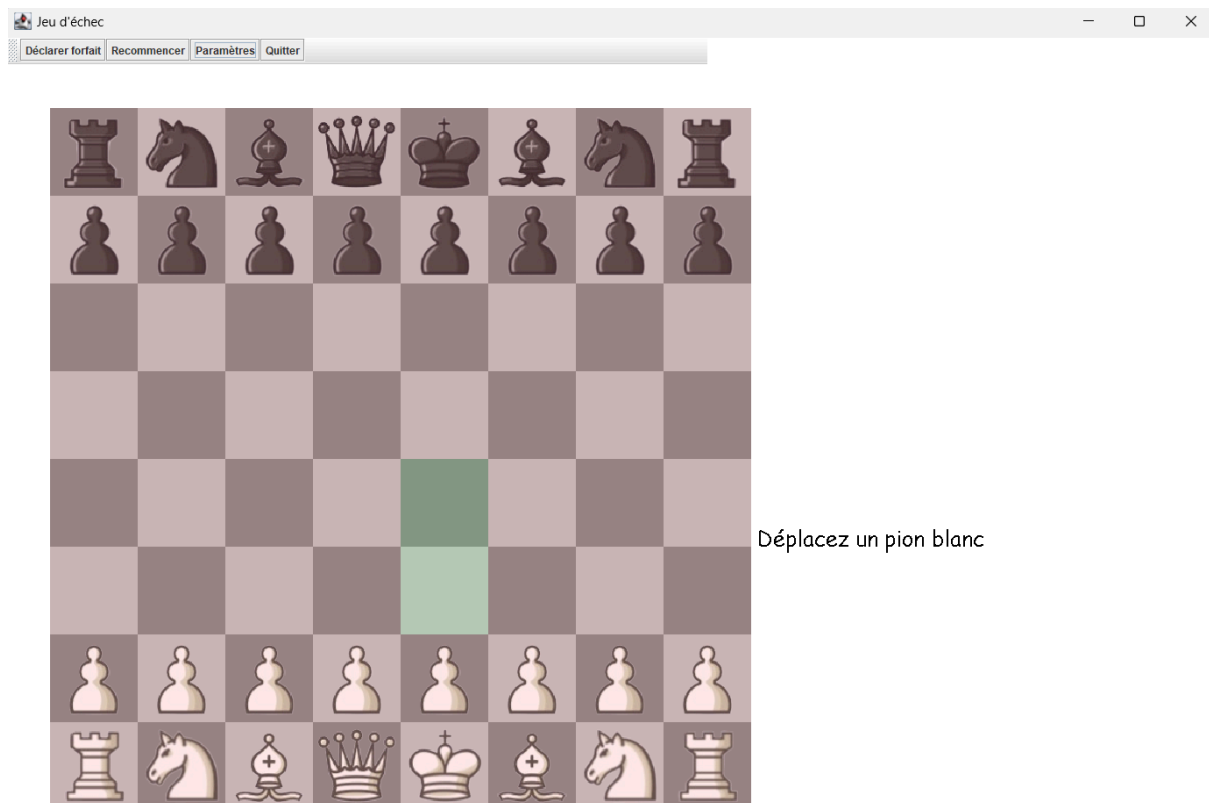


Les différentes options possibles sont situées dans la barre d'outils. Si par exemple on clique sur "Paramètres", cette fenêtre s'ouvre :



On peut régler la position et la largeur de la fenêtre et de l'échiquier avec les sliders, et choisir si on veut enregistrer les scores dans la base de données en cochant "Oui" ou "Non". Ces paramètres peuvent être ouverts n'importe quand lorsque la partie est en cours et le réglage est pris en compte à l'appui du bouton "OK".

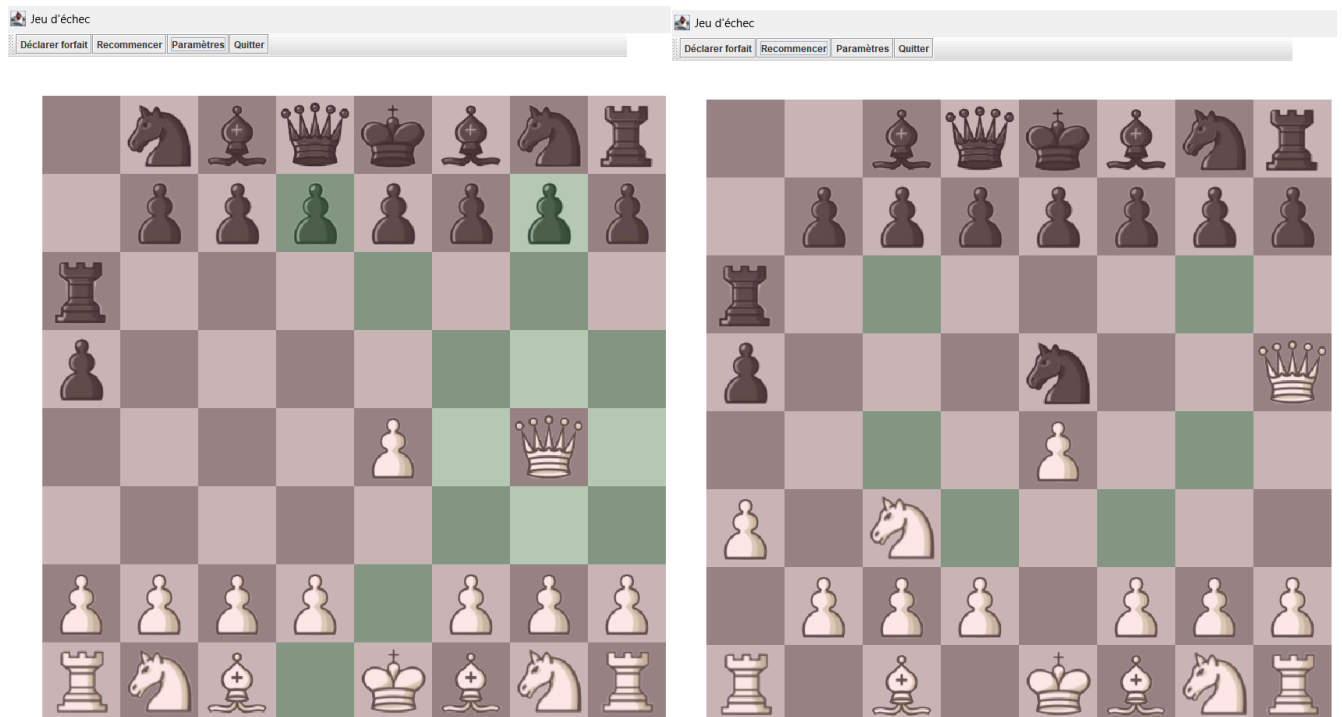
Ensuite, lorsqu'on clique sur un pion (ou n'importe où sur la case du pion) de la bonne couleur, on peut voir ses possibilités de déplacement en vert, le reste des déplacements sont colorés en rouge.



Si on clique en dehors du plateau ou sur une case rouge, on revient au choix du pion à déplacer. En revanche si on clique sur une case verte (n'importe où sur la case), le pion se déplace sur cette case et le tour se termine.



La partie continue. Pour chaque type de pion, les déplacements possibles sont différents. Par exemple, voici ce qu'on obtient lorsqu'on clique sur une dame ou un cavalier :

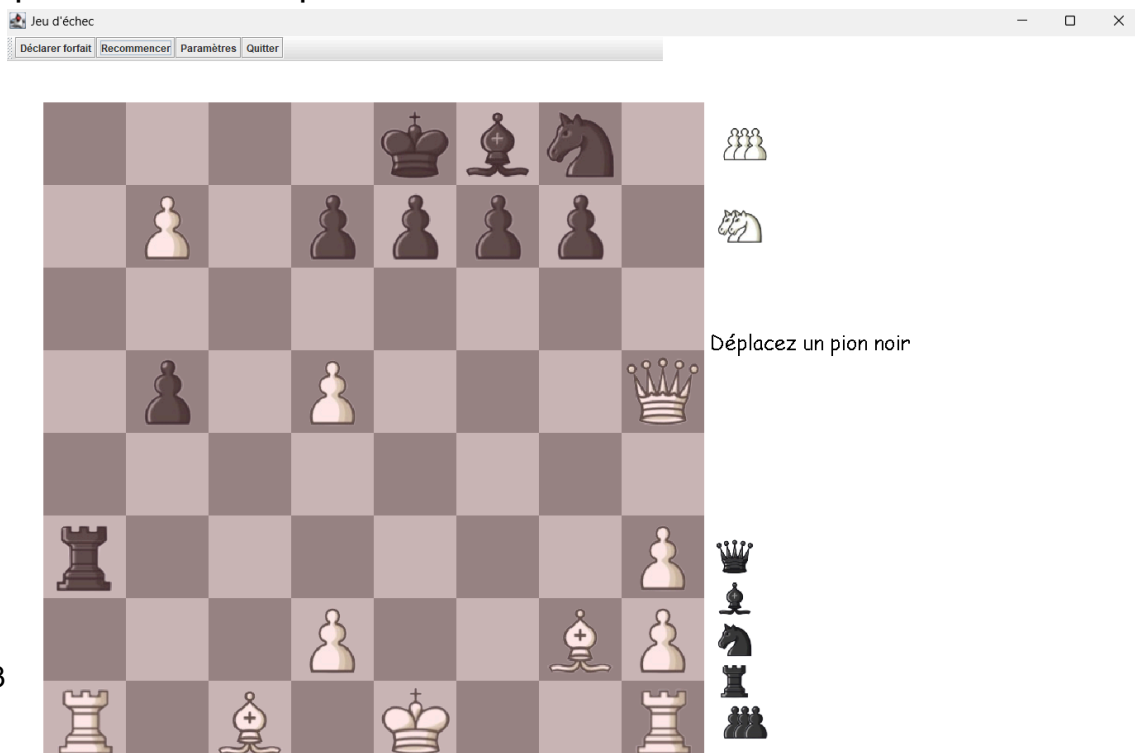


Lorsque l'on choisit de se déplacer sur une case où il y a déjà un pion ennemi, le pion sélectionné mange le pion ennemi et prend sa place. Les pions mangés apparaissent alors à droite de l'échiquier :



Comme on peut le voir, les pions mangés apparaissent en plus petit et ceux du même type et de la même couleur s'empilent.

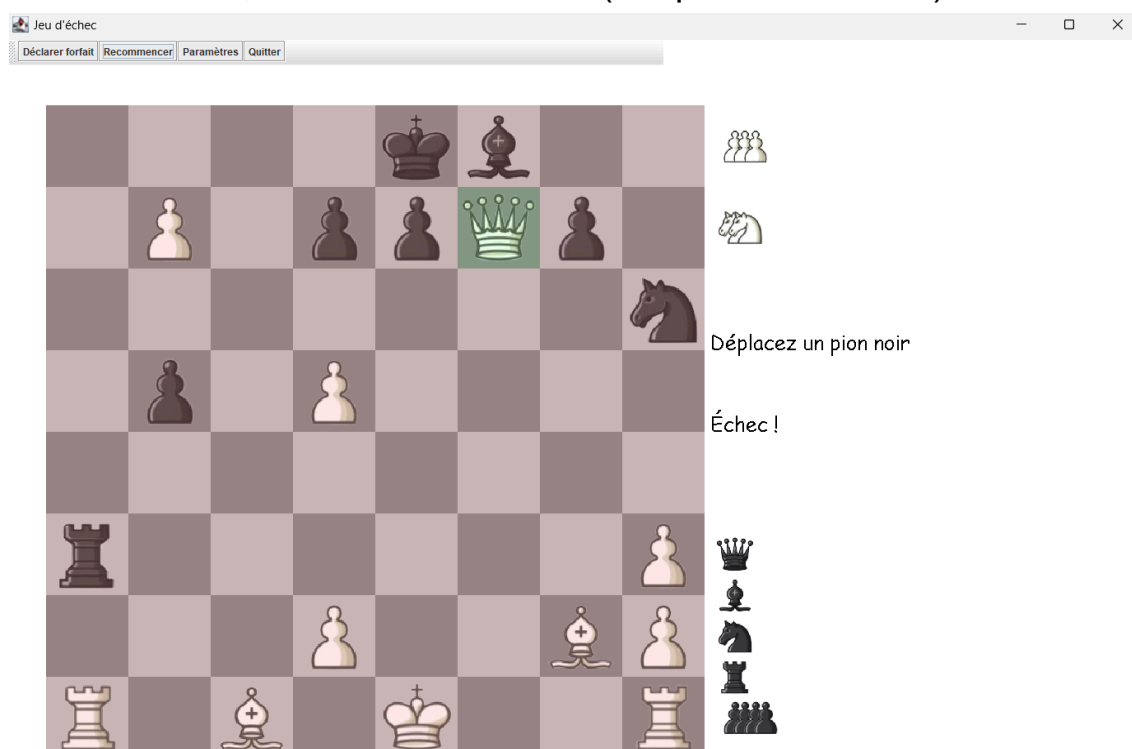
Ici lorsqu'on essaie de déplacer le pion noir situé entre le roi noir et la dame blanche, toute les case apparaissent en rouge car un déplacement de ce pion mettrait en échec son roi.



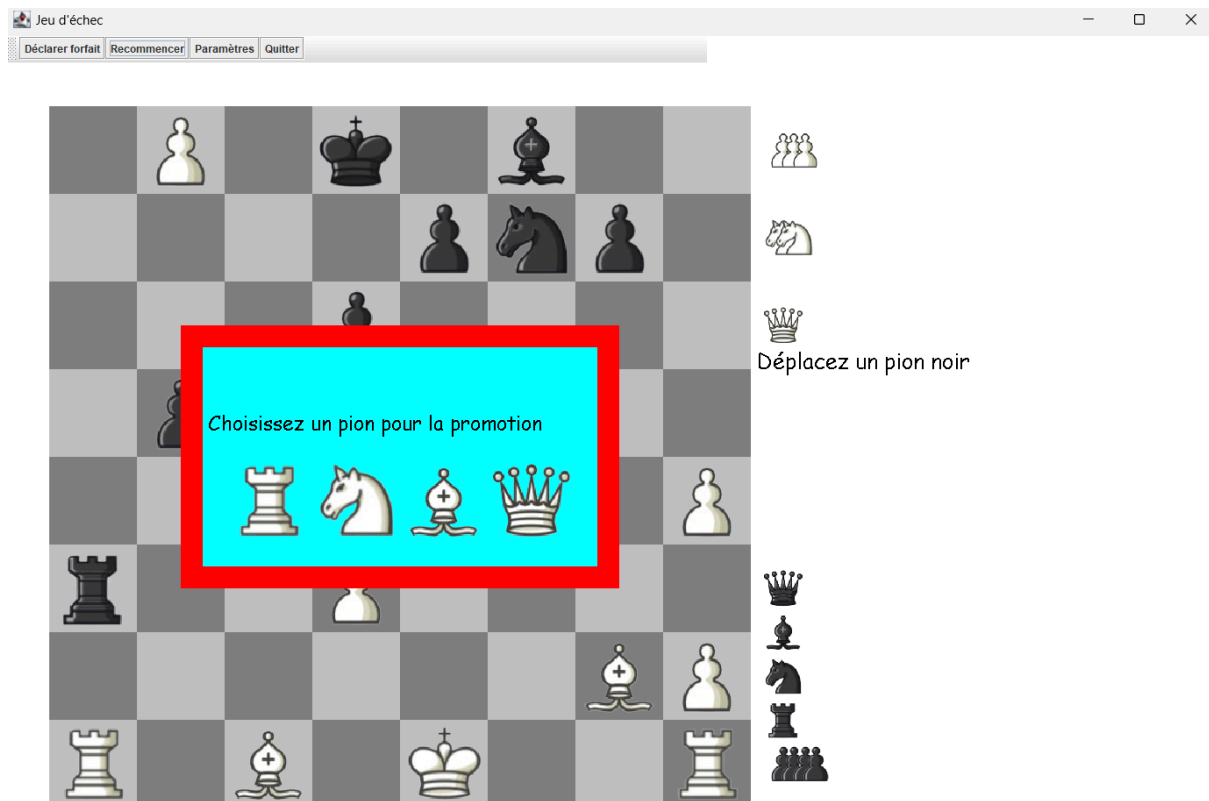
Lorsqu'il y a échec, on nous prévient à droite de l'échiquier :



Seuls les déplacements "sauvant" le roi allié sont alors possibles, tous les autres sont en rouge. Dans notre cas, les seules possibilités seraient de déplacer le roi noir vers la gauche ou de manger la dame blanche soit avec le roi, soit avec le cavalier (ce qu'on va faire ici).



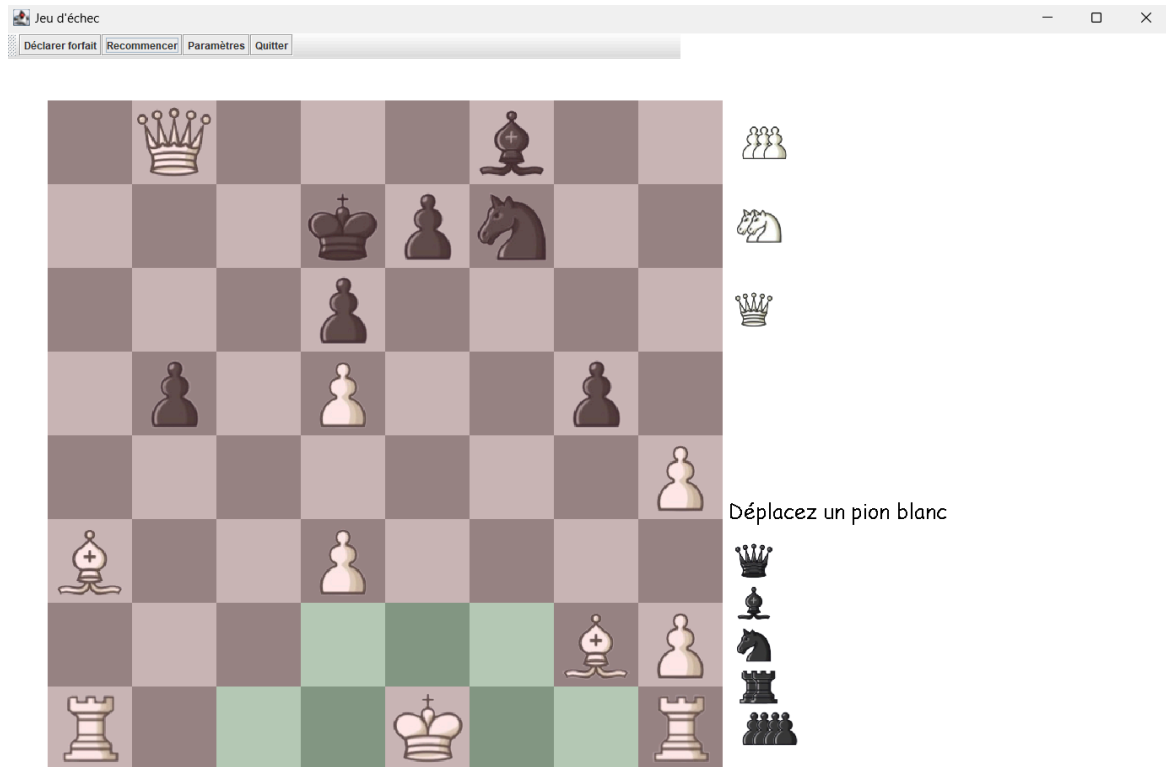
Lorsqu'un pion simple traverse tout le plateau et atteint l'autre bout de l'échiquier, on nous demande de choisir un pion pour sa promotion :



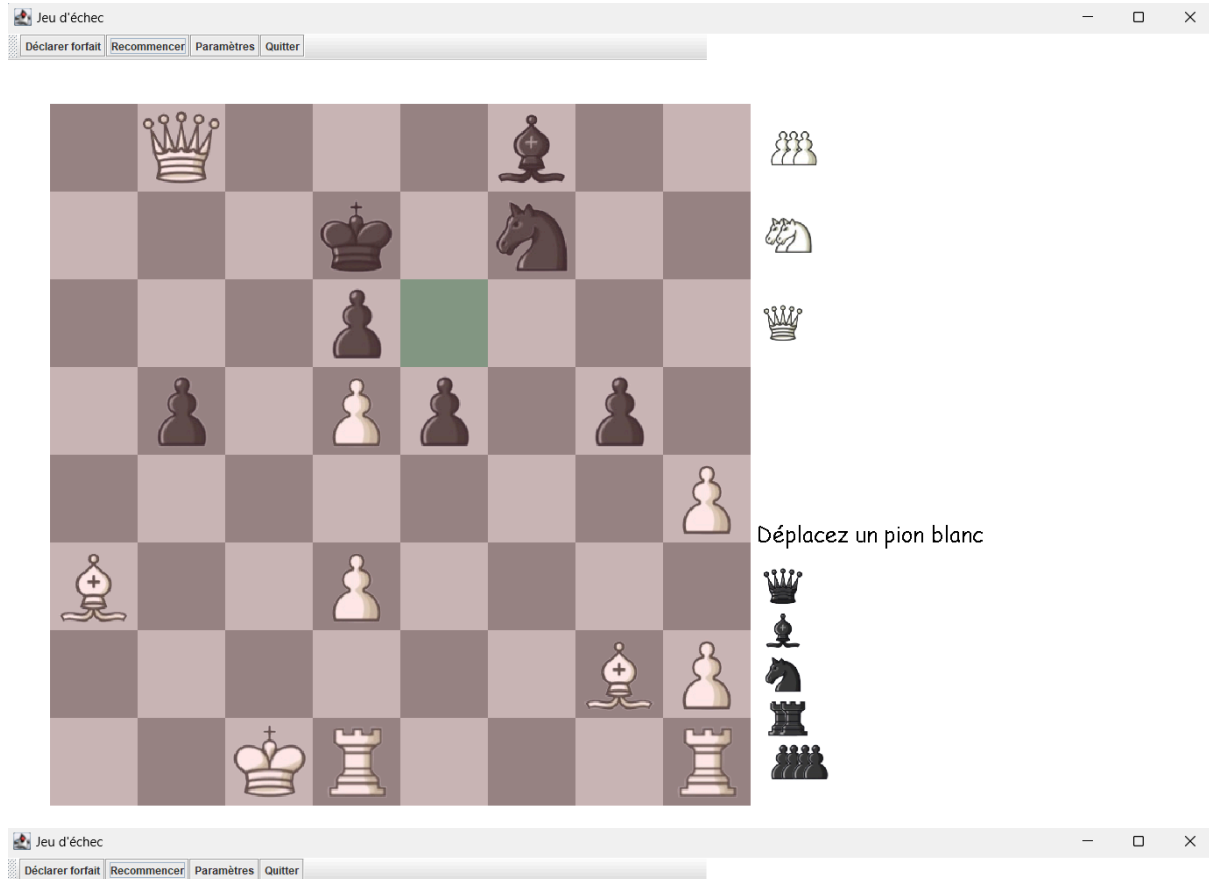
Le pion est alors remplacé par le pion sur lequel on clique, ici la dame (ce qui met le roi ennemi en échec).



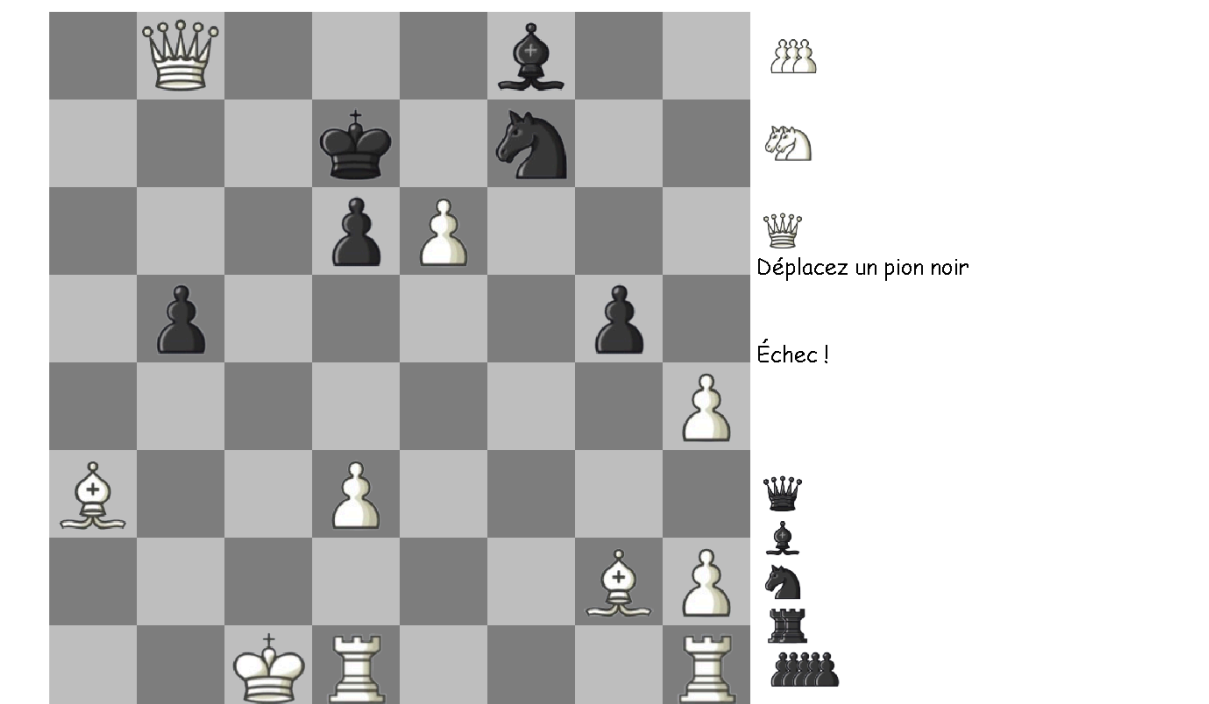
On va maintenant tester les coups spéciaux. Dans cette partie, les tours blanches et le roi n'ont pas encore été déplacés. Alors il est possible de faire un roque, c'est-à-dire de déplacer le roi de deux cases vers une tour et de passer la tour de l'autre côté du roi. On peut le faire soit à droite, soit à gauche, à condition que le roi ne soit pas en échec.



Si le joueur suivant décide d'avancer le pion simple noir (qui n'a pas encore bougé de la partie) de deux cases, le joueur blanc peut manger ce dernier au tour suivant comme si le pion noir avait avancé d'une seule case. Il s'agit du coup "en passant".



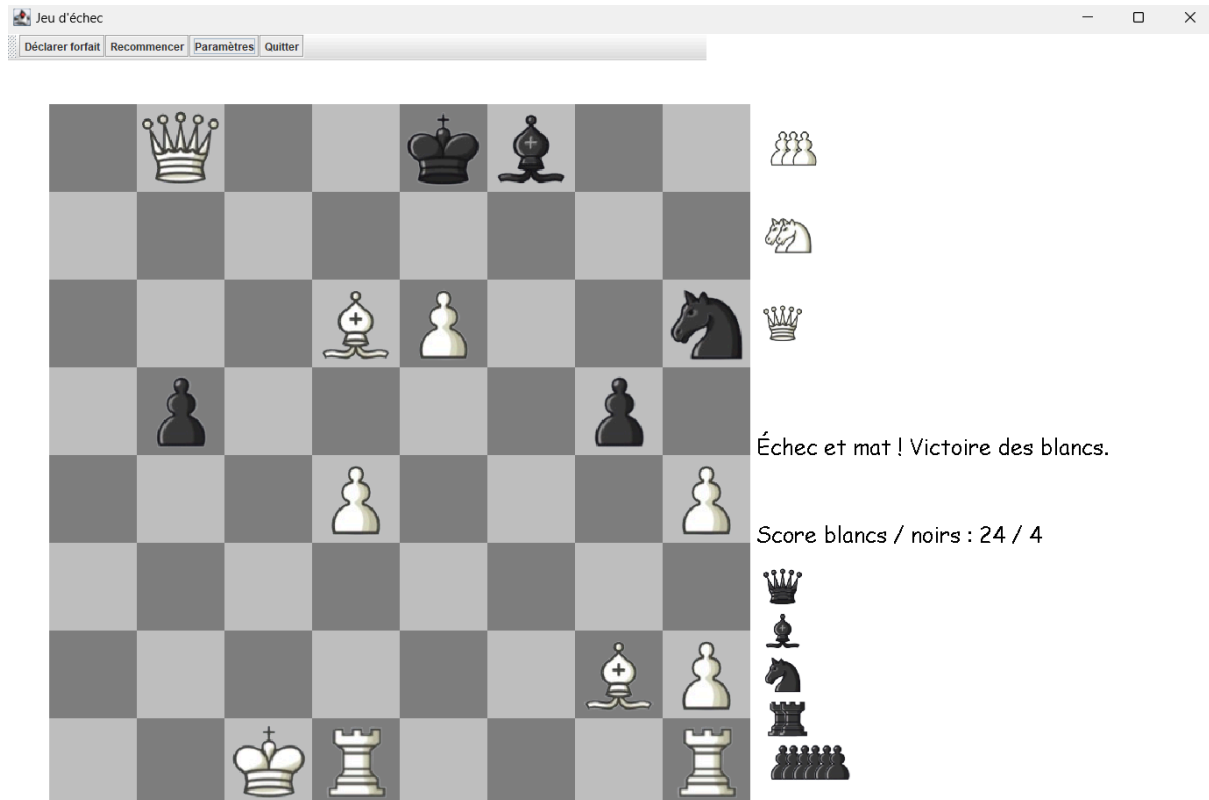
Déplacez un pion blanc



Déplacez un pion noir

Échec !

Nous arrivons à la fin de la partie. Si un joueur ne peut plus effectuer aucun déplacement sans mettre en échec son roi, la partie se termine. Si de plus le roi est en échec, la victoire revient au joueur adverse par échec et mat. Sinon, on dit qu'il y a "pat" et personne ne gagne.



Ici, le roi noir est en échec, ne peut se déplacer nulle part sans se mettre en échec et aucun pion noir ne peut le sauver. Il y a donc échec et mat et la partie est remportée par les pions blancs. Les scores s'affichent et s'enregistrent ou pas dans la base de données selon le choix fait dans les paramètres.

3. Développement du jeu

Dans cette section, nous allons présenter comment on a procédé pour coder le jeu d'échecs en java sur le logiciel IntelliJ.

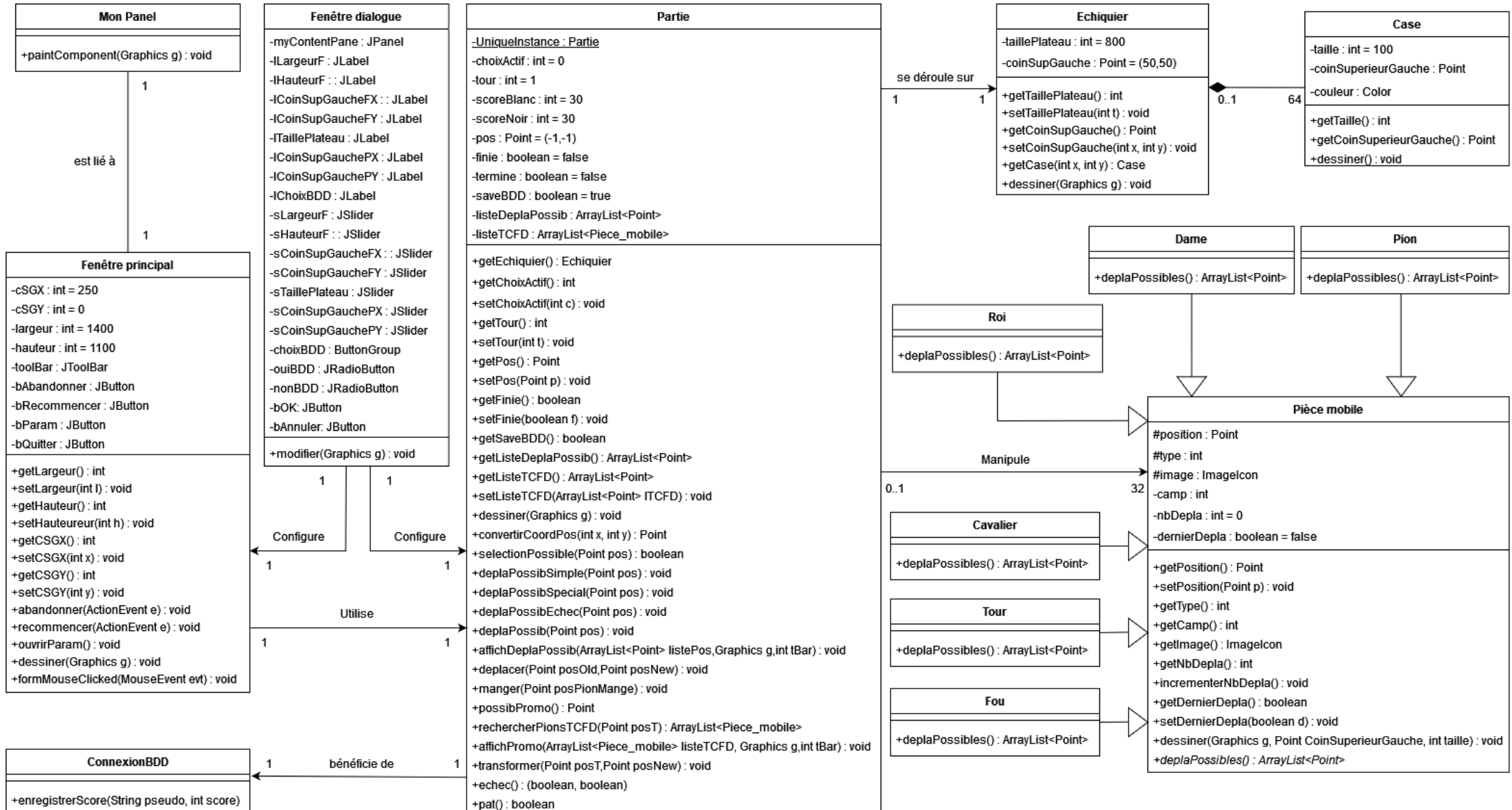
Au début, nous avons simplement repris les codes des TP 1 et 2 et l'avons adapté pour représenter notre échiquier, composé de cases. Ensuite, nous nous sommes intéressés à la classe d'un pion particulier (la tour) que l'on a définie comme sous-classe de "Pièce mobile". Une fois que ces classes avaient la base nécessaire pour pouvoir s'afficher correctement, nous avons implémenté la classe "Partie" qui gère à la fois les pions et l'échiquier. C'est dans cette classe que nous définissons principalement les règles du jeu. Nous avons construit le code de la fenêtre principale au fur et à mesure de l'avancement du projet, et avons créé les autres pions une fois que tout fonctionnait pour le premier type de pion. Enfin, nous avons corrigé certaines erreurs que nous avons pu rencontrer et ajouté des fonctionnalités supplémentaires (enregistrement de score dans la base de données, affichage de la partie, création de boutons spécifiques, paramètres...)

14 classes au total ont été créées :

- la classe "Case"
- la classe "Echiquier"
- la classe "Pièce mobile" et ses 6 sous-classes correspondant chacune à un type de pion
- la classe "Partie"
- la classe "Fenêtre principale"
- la classe "Fenêtre dialogue" (fenêtre qui s'ouvre lorsqu'on accède au paramètres de la partie)
- la classe "ConnexionBDD"
- la classe "Mon Panel"

Ci-dessous se trouve le diagramme de classe correspondant :

Diagramme de classe



3.1. Description des classes

Classe “Case”

Cette classe permet de dessiner un carré qui correspondra à une case de l'échiquier. Elle est destinée à être instanciée plusieurs fois dans la classe échiquier (64 fois en tout) pour constituer le plateau.

Son constructeur prend comme paramètre la position de la case repérée par les coordonnées de son coin supérieur gauche, la couleur de la case et la taille d'un côté de la case. La méthode *dessiner* dessine simplement un carré suivant les paramètres renseignés.

Classe “Echiquier”

La classe “Echiquier” permet de construire le plateau du jeu.

Son constructeur prend en paramètre la taille d'un côté du plateau (tous les côtés égaux) et les coordonnées du coin supérieur gauche de l'échiquier dans la fenêtre. Ces paramètres ainsi que les mutateurs permettent la modification de ces 2 attributs, que l'on fait via la fenêtre des paramètres.

On construit l'échiquier case par case avec une double boucle *for*, en colorant différemment les cases paires et impaires dont la taille est donnée par la taille de l'échiquier divisé par 8.

Classe “Pièce mobile”

Cette classe comprend tous les attributs et méthodes que les pions ont en commun.

La position du pion ne correspond pas ici aux coordonnées dans la fenêtre mais à la position du pion dans la matrice constituant l'échiquier. C'est un point (x,y) entre (0,0) et (11,7). Les positions entre (0,0) et (7,7) indiquent la position du pion dans l'échiquier, la première coordonnée correspondant à la ligne et la deuxième à la colonne, tandis que les positions entre (8,0) et (11,7) sont des emplacements réservés aux pions “mangés” en dehors du plateau. Il y a donc en tout $12 \times 8 = 96$ emplacements possibles pour les pions, dont 64 vides car il y a 32 pions.

Le type du pion est un chiffre entre 0 et 5 permettant de savoir quel type de pion il s'agit : 0 pour tour, 1 pour cavalier, 2 pour fou, 3 pour dame, 4 pour roi et 5 pour pion simple.

Le camp correspond à la couleur du pion : 1 pour blanc et 0 pour noir.

Nous avons ajouté le nombre de déplacements des pions et un booléen *dernierDepla* indiquant si le pion vient d'être déplacé ou non. Ces deux attributs nous seront utiles notamment pour les règles comme "en passant" et le "roque" ou encore pour le déplacement des pions simples. Le constructeur prend en paramètre la position et le camp pour les initialiser. Les attributs *image* et *type* dépendent du pion en question donc ils sont initialisés dans les classes des pions. Il y a également un constructeur de copie pour cette classe et les sous-classes "Tour", "Cavalier", "Fou" et "Dame", qu'on utilisera lors de la promotion d'un pion.

La méthode *dessiner* prend en paramètre la taille et la position du pion dans la fenêtre, que l'on trouvera à partir de la position du pion dans la matrice de l'échiquier, et dessine l'image à l'emplacement correspondant.

La méthode *deplaPossibles* est abstraite car dépend du type des pions.

Classe "Tour"

Dans chaque sous-classe de "Pièce mobile", on doit redéfinir la méthode *deplaPossibles*. Ici, nous n'avons pas accès aux autres pions donc cette méthode renverra une liste de déplacements possibles du pion dans la matrice de l'échiquier sous formes de coordonnées (x,y) comme s'il n'y avait aucun pion sur l'échiquier. Le reste (prise en compte des échecs, des autres pions et déplacements exceptionnels) est géré dans la classe "Partie" qui a accès à tous les pions.

La tour se déplace en ligne droite donc sa méthode *deplaPossibles* est assez simple à implémenter. On fixe la colonne à la position y du pion et on parcourt les lignes puis on fixe la ligne à la position x du pion et on parcourt les colonnes, le tout en une boucle *for*.

Classe "Cavalier"

Le cavalier se déplace d'une case horizontalement et de deux cases verticalement ou d'une case verticalement et de deux cases horizontalement. Pour obtenir les coordonnées d'une case où le cavalier

peut se déplacer, on augmente ou diminue de 1 une coordonnée de la position du pion et on augmente ou diminue de 2 l'autre coordonnée. On implémente cela à l'aide de deux petites boucles pour ajouter les déplacements possibles à la liste en sortie en veillant à ce qu'on reste toujours dans le plateau (coordonnées toujours comprises entre 0 et 7 inclus).

Classe “Fou”

Même principe pour le fou : ses déplacements diagonaux sont obtenues en augmentant ou diminuant une coordonnée de la position du pion et en augmentant ou diminuant autant l'autre coordonnée, sans dépasser de l'échiquier.

Classe “Dame”

Pour les déplacements de la dame, on combine les déplacements de la tour et du fou. On peut remplir la liste des déplacements possibles en une seule boucle, avec des conditions pour ne pas sortir de l'échiquier.

Classe “Roi”

Le roi ne se déplace que d'une case (en dehors du roque qui est géré dans la classe “Partie”). On ajoute donc à la liste des déplacements la position du roi (x,y) additionnée de toutes les combinaisons (i,j) pour i et j entre -1 et 1.

Classe “Pion”

Le déplacement des pions simples est plus complexe (avance d'une case ou deux s'il s'agit du premier déplacement, ne peuvent manger qu'en diagonale, règle “en passant”) donc il est traité totalement dans la classe “Partie” dans une méthode à part appelée *deplaPossibSimple*. Comme c'est une sous-classe de “Pièce mobile” qui contient la méthode abstraite *deplaPossibles*, on est obligé de définir une méthode *deplaPossibles* pour “Pion” mais celle-ci ne sera jamais utilisée.

Classe “Partie”

Cette classe contient à la fois l'échiquier et une matrice de pions 12×8 où les 8 premières colonnes représentent l'échiquier et les 4 dernières des emplacements pour les pions mangés. Voici comment est initialisé cette matrice dans le constructeur :

tn	cn	fn	dn	rn	fn	cn	tn				
pn	pn	pn	pn	pn	pn	pn	pn				
tb	cb	fb	db	rb	fb	cb	tb				
pb	pb	pb	pb	pb	pb	pb	pb				
échiquier								pions mangés			

La classe “Partie” contient également des attributs supplémentaires :

- choixActif, un entier entre 0 et 2 qui indique ce que doit faire le joueur. 0 s'il doit sélectionner un pion, 1 s'il doit choisir un déplacement et 2 s'il doit choisir un pion pour la promotion. Cet attribut est modifié dans la fenêtre principale.
- tour = 1 si c'est au tour du joueur jouant les pions blancs, 0 si c'est au tour du joueur jouant les pions noirs.
- les score des 2 joueurs initialisé à 30, qu'on modifiera dans les méthodes *deplacer*, *manger* et *transformer* et à la fin de la partie.
- la position *pos* du pion sélectionné par le joueur, utile pour avoir accès à cette position lors du prochain click déterminant son déplacement.
- les booléens *finie* qui indique si la partie est finie, *temine* qui permet d'éviter de changer et d'enregistrer le score plusieurs fois à la fin de la partie, et *saveBDD* qui indique si les scores doivent être enregistrés dans la base de données.

- l'attribut *listeDeplaPossib*, contenant les déplacements possibles d'un pion sélectionné, utile pour avoir accès à cette liste lors du prochain click.
- la liste *listeTCFD*, qui contiendra dans l'ordre une tour, un cavalier, un fou et une dame lors de la promotion, utile pour avoir accès à cette liste lors du prochain click.

Les méthodes de cette classe permettent de jouer au jeu en suivant les règles et peuvent être complexes car il y a souvent plusieurs cas à gérer. Certaines seront décrites dans la partie 2.3.

Classe “Fenêtre principale”

Cette classe permet de construire la fenêtre principale du jeu et de définir ce qui se passe lorsqu'on clique à un endroit dans cette fenêtre (dans la méthode *formMouseClicked*). Elle ne contient que les attributs spécifiques à la fenêtre : coordonnées du coin supérieur gauche de la fenêtre (*cSGX* et *cSGY*), hauteur et largeur, barre d'outil avec les boutons pour déclarer forfait, recommencer, paramétrer ou quitter la partie.

La classe contient la partie et utilise ses méthodes pour que le jeu se déroule selon les règles des échecs. On détaillera son fonctionnement dans la partie 3.2.

Classe “Fenêtre dialogue”

Cette fenêtre s'ouvre lorsque l'on clique sur le bouton “Paramètres” dans la barre d'outil de la fenêtre principale. La classe a besoin d'être liée à la fenêtre principale pour pouvoir configurer sa position et ses dimensions, et aussi à la partie pour pouvoir modifier l'attribut *saveBDD* et changer la taille et la position de l'échiquier via l'accessor *getEchiquier* de “Partie”. Elle contient comme attributs tous les labels, boutons et sliders. Le bouton “OK” appelle une procédure *modifier* qui met à jour la fenêtre et le plateau selon les valeurs des sliders/boutons, avant de redessiner la fenêtre principale et de fermer la fenêtre dialogue.


Classe “Connexion BDD”

Cette classe un peu particulière contient seulement la méthode *enregistrerScore*, qui ajoute à la base de données les pseudos renseignées à l'appel de la fonction (dans la méthode *dessiner* de “Partie”) suivi des scores.

Description détaillée de la classe:

1. Nous importons (`'java.sql.*'`) qui est une classe permettant d'interagir avec les bases de données.
2. Méthode *enregistrerScore* : Cette méthode prend en paramètres un pseudonyme et un score. Elle est utilisée pour enregistrer ces informations dans la base de données.
3. Connexion à la base de données : Le code établit une connexion à une base de données MySQL en spécifiant l'URL de connexion (`"jdbc:mysql://172.16.23.115/tpjavaimds5A"`) ('172.16.23.115' l'adresse IP du serveur, 'tpjavaimds5A' le nom de la base de données), le nom d'utilisateur ('IMDS5A') et le mot de passe ('Polytech').
4. Chargement du Pilote JDBC et Connexion : Le pilote JDBC spécifique à MySQL est chargé, et une connexion à la base de données est établie en utilisant les informations de connexion fournies.
5. Création d'une Déclaration : Un objet 'Statement' est créé pour permettre l'exécution de requêtes SQL sur la base de données.
6. Insertion d'un Nouveau Score : Un nouveau score est inséré dans la table "Scores" de la base de données avec les valeurs du pseudonyme et du score fournies en paramètres.
7. Exécution d'une Requête de Sélection ('SELECT') : Une requête 'SELECT' est exécutée pour récupérer tous les enregistrements de la table "Scores". Les résultats sont ensuite affichés dans la console.
8. Gestion des Exceptions: En cas d'erreur lors de la connexion à la base de données ou lors de l'exécution des requêtes, les détails de l'exception sont affichés, et le programme se termine.

Exemple affichage score avec heidiSQL:

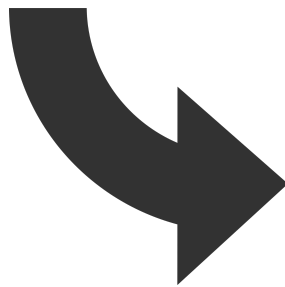


Échec et mat ! Victoire des blancs.

Score blancs / noirs : 37 / 28

```

Main >
nom: 15
id: 5481661
nom: null
nom: 0
id: 5481662
nom: null
nom: 15
id: 5481663
nom: player1F_C
nom: 37
id: 5481664
nom: player2F_C
nom: 28
  
```



	idScores	pseudo	score
67	189 759	player2F_C	29
68	4 465 465	ps_blancs	0
69	5 481 651	ps_noirs	0
70	5 481 652	Theo	4
71	5 481 653	null	15
72	5 481 654	null	0
73	5 481 655	null	15
74	5 481 656	null	0
75	5 481 657	null	15
76	5 481 658	null	0
77	5 481 659	null	0
78	5 481 660	null	15
79	5 481 661	null	0
80	5 481 662	null	15
81	5 481 663	player1F_C	37
82	5 481 664	player2F_C	28

Les scores de Player1F_C (Player 1 Felix Corentin) et Player2F_C (Player 1 Felix Corentin) sont ajoutés à la table score. Étant donné que les index sont auto-incrémentés, nous avons juste eu besoin de mettre les scores et les pseudos dans le *insert into*, les index s'incrémentent directement à la suite.

Classe “Mon Panel”

Cette classe permet de réafficher la fenêtre principale lorsqu'on la rétrécit puis l'agrandit avec la méthode *paintComponent*.

3.2. Description du programme principal

Le programme principale du jeu est contenu dans la méthode *formMouseClicked* de la fenêtre principale, qui permet de récupérer les coordonnées de l'endroit où l'on clique dans la fenêtre. En effet, les utilisateurs n'ont besoin que de la souris pour jouer à ce jeu donc c'est cette méthode qui est appelée à chaque fois au cours de la partie.

Quel que soit l'état du jeu au moment où l'on clique, la méthode commence par convertir grâce à la fonction *convertirCoordPos* les coordonnées dans la fenêtre en position dans l'échiquier entre (0,0) et (7,7), (-1,-1) si on a cliqué en dehors du plateau.

Quand la partie commence, l'attribut *choixActif* est égale à 0 : le joueur doit choisir un pion à déplacer. Lorsqu'il clique, la méthode *formMouseClicked* est enclenchée. On vérifie premièrement qu'il a cliqué sur un pion de la bonne couleur avec la fonction *selectionPossible*. Si c'est le cas on stocke la position du pion choisi dans l'attribut *pos* de partie via son mutateurs. Cela permettra d'y avoir accès au prochain appel de *formMouseClicked*. Ensuite on appelle la procédure *deplaPossib* pour ce pion, ce qui va remplir l'attribut *listeDeplaPossib* des coordonnées des cases où le pion peut se déplacer. Puis on affiche ces déplacements avec la méthode *affichDeplaPossib*. Enfin, l'attribut *choixActif* passe à 1 et on sort de la procédure *formMouseClicked*. Voilà ce qui se passe au premier clic sur un pion.

Lorsque *choixActif* = 1, le joueur doit choisir un déplacement. Au moment du clic, on commence par vérifier que la liste des déplacements possibles du pion sélectionné au préalable contient la position pointée par la souris en utilisant la méthode *contains* existant par défaut en java. Si ce n'est pas le cas (case rouge sélectionnée ou clic en dehors du plateau), on redessine la fenêtre sans l'affichage des déplacements possibles et on revient au choix du pion (*choixActif* = 0). Sinon (case verte sélectionnée), on appelle la procédure *déplacer* pour déplacer le pion et éventuellement manger un autre pion, puis on vérifie si on est dans le cas d'une promotion avec *possibPromo*. Cette fonction renvoie la position du pion s'il est possible d'effectuer une promotion, et (-1,-1)

sinon. Cette position est alors stockée dans l'attribut *Pos* afin d'y avoir accès lors du prochain clic déterminant la promotion du pion (si elle est effectivement possible).

Dans le cas d'une promotion, on redessine le plateau avec le pion déplacé, puis on va appeler la fonction *rechercherPionsTCFD*. Cette fonction renvoie une tour, un cavalier, un fou et une dame présents dans l'échiquier (ou mangés). On les stocke dans l'attribut *listeTCFD* via son mutateur pour ensuite afficher les images de chacun de ces pions avec la procédure *affichPromo*. Enfin on passe l'attribut *choixActif* à 2 pour que le prochain clic serve à choisir la promotion. S'il n'y a pas de promotion possible, on change de joueur (avec une petite fonction simple qui renvoie 1 si $t=0$ et 0 si $t=1$: $f(t) = -1 \times (t - 1)$) et on remet *choixActif* à 0.

Pour finir, lorsque *choixActif* = 2, c'est que l'on doit choisir un pion pour la promotion. Il ne se passe rien tant que le joueur ne clique pas sur un des 4 pions proposés qui s'affiche aux positions (2,4) à (5,4). Une fois qu'il a choisi, on appelle la procédure *transformer* qui va utiliser la position du pion *Pos* et *listeTCFD* pour remplacer le pion simple par une copie du pion choisi dans la liste. On termine en changeant de joueur, en remettant *choixActif* à 0 et en redessinant la fenêtre avec le pion maintenant promu.

Dans la méthode *dessiner* de "Partie", on donnera la valeur *false* à l'attribut *finie* lorsqu'on aura un "pat". Ainsi, cliquer sur la fenêtre ne fera plus rien car on exécute les instructions de la méthode *formMouseClicked* seulement si *finie* = *false*.

3.3. Description de certaines méthodes

On va ici décrire le fonctionnement de quelques méthodes importantes de la classe “Partie”.

Méthode *manger*

Cette procédure est appelée dans la méthode *deplacer* lorsqu’un pion se déplace sur une case qui n’est pas vide ou si on est dans le cas “en passant”.

La position du pion mangé est passée en paramètre. On parcourt la matrice de pions à partir de la 9ème colonne (emplacements des pions mangés) jusqu’à ce qu’il y ait une place vide, et on y ajoute le pion mangé avant de libérer son ancienne place dans l’échiquier. À la fin, on augmente le score du joueur de 2 points.

Méthode *dessiner*

Dans cette méthode, on commence par appeler la méthode *dessiner* de l’échiquier puis, par dessus, celle des pions présents sur l’échiquier (contenus dans les 8 premières colonnes de *matricePions*) sans oublier de convertir les positions des pions sur l’échiquier en coordonnées dans la fenêtre.

On va ensuite dessiner à droite de l’échiquier les pions mangés. Pour cela on parcourt la matrice des pions à partir de la 9ème colonne. Afin de les trier par type et couleur, on va, selon le type des pions, puis selon leur couleur, les dessiner à une ligne différente. On utilise des compteurs pour chaque type de pions de couleur différente afin d’obtenir la colonne où chaque pion sera dessiné, à une taille deux fois plus petite qu’une case.

Ensuite, on traite l’affichage du texte selon les différents cas, dont la taille et la position dépendent de la taille de l’échiquier.

On vérifie si la partie est finie avec la fonction *pat*, qui renvoie *true* si aucun déplacement n’est possible, ou en évaluant l’attribut *finie* qui prend la valeur *true* dans le cas d’un abandon. Dans ce cas, on affiche

qu'il y a "pat", "échec et mat" (s'il y a en plus échec) ou encore "victoire par abandon" selon la situation, et on enregistre, après avoir éventuellement augmenter le score du vainqueur de 10 points, les deux scores. Cette dernière instruction est exécutée seulement si l'attribut *termine* vaut *false*, signifiant que ce bloc n'a pas encore été exécuté. Enfin, on affiche les scores et on met *termine* à *true* pour les prochain appel de *dessiner()* (lorsqu'on réduit la fenêtre ou active un bouton). Dans le cas standard (dernier *else*), on affiche simplement quel pion (noir ou blanc) il faut déplacer et on appelle la fonction *echec* pour également afficher s'il y a échec.

Méthode *deplaPossib*

Cette méthode remplit *listeDeplaPossib* à partir de la position d'un pion, de tous ses déplacements possibles dans l'échiquier, prenant en compte les autres pions et les échecs.

Dans un premier temps, on ne prend pas en compte les échecs. Si le pion est un pion simple, on appelle la méthode *deplaPossibSimple*. Cette dernière vérifie d'abord si le pion peut se déplacer (vers le haut ou vers le bas selon sa couleur) sans sortir du plateau. On vérifie ensuite :

- s'il peut se déplacer d'une case en regardant si la prochaine est vide.
- s'il peut se déplacer de deux cases en regardant si le nombre de déplacement du pion est 0 et si les 2 prochaines cases sont vides.
- s'il peut se manger un pion en diagonale en regardant s'il y a un pion ennemi dans la diagonale.
- s'il peut manger un pion "en passant" en regardant si le pion est positionné à la bonne ligne pour effectuer ce coup, s'il y a un pion simple ennemi à droite du pion, qui vient de se déplacer (*dernierDepla = true*) et dont le nombre de déplacement est 1.

On remplit alors la liste des déplacements possibles en fonction de la validité de ces conditions.

S'il s'agit d'un autre type de pion, on appelle *deplaPossibSpecial*. Dans cette procédure, on commence par appeler la méthode *deplaPossible* du pion. On parcourt ensuite la liste des déplacements possibles de ce pion

et on ajoute les déplacements menant sur une case occupée à une liste temporaire *posOccupee*. On en profite pour enlever de la liste des déplacements possibles les cases occupées par des pions alliés.

Ensuite, on doit retirer les cases “cachées” par des pions. Pour cela, on parcourt la liste des déplacements possibles. Pour chaque déplacement, on parcourt ensuite *posOccupee* et on regarde à chaque fois si la case occupée se situe entre la case du pion et la case du déplacement que ce soit sur la colonne, la ligne ou la diagonale. Dans ce cas, on ajoute le déplacement à une nouvelle liste *posCache*, pour ensuite enlever ces déplacements de la liste de déplacements possible.

Après cette étape, on appelle la méthode *deplaPossibEchec* pour enlever les déplacements menant son roi en échec. Dans cette procédure, on procède au déplacement du pion sur une case où le déplacement est possible mais sans l’afficher. On regarde alors s’il y a échec en appelant la méthode *echec*, et dans ce cas on enlève le déplacement de la liste. On remet ensuite le pion à sa place, ainsi que celui qu’il a éventuellement mangé et qu’on a gardé dans une variable temporaire.

Enfin, on vérifie dans *deplaPossib* si le déplacement est un “roque”. Pour cela, on doit d’abord regarder si c’est le roi qui se déplace pour la première fois, qu’il n’y a pas échec, puis pour chaque tour vérifier qu’elle ne s’est pas non plus déplacée et que les cases séparant le roi et la tour sont vides, et enfin que l’emplacement entre le pion et la case est bien présent dans la liste des déplacements possibles actuelle (elle ne met pas le roi en échec). Dans ce cas, on ajoute le déplacement de deux cases à la liste, et on rappelle *deplaPossibEchec* pour l’enlever si elle mène à un échec. C’est dans la méthode *deplacer* qu’on effectuera le déplacement de la tour en même temps que celui du roi.

Méthode *deplacer*

Cette procédure permet de déplacer le pion situé en position *posOld* sur la case en position *posNew*, où *posOld* et *posNew* sont deux paramètres de la méthode.

Si la case en position *posNew* n'est pas vide, on appelle la méthode *manger* sur *posNew*. Si la case est vide mais qu'il s'agit d'un déplacement diagonal par un pion simple, c'est qu'on est dans le cas "en passant". On appelle alors *manger* sur la position située sur la même ligne que *posOld* et la même colonne que *posNew*.

Ensuite on vérifie si on est dans le cas d'un roque en regardant si le pion est un roi qui se déplace de deux cases. Dans ce cas, selon la tour concernée, on la change de place.

C'est à ce moment qu'on effectue le changement de place du pion.

Il faut ensuite parcourir la matrice des pions et trouver celui ayant *dernierDepla = true* (il s'agit de celui qui s'est déplacé en dernier) pour remettre cet attribut à *false*, puis mettre celui du pion déplacé à *true*.

On incrémente également le nombre de déplacement de ce pion et on baisse le score du joueur effectuant le déplacement d'un point.

Conclusion

En conclusion, la réalisation de ce jeu d'échecs en Java a constitué une opportunité pour approfondir notre connaissance du logiciel IntelliJ et affiner notre maîtrise de Java. Chaque étape de l'implémentation a été l'occasion d'une réflexion approfondie pour élaborer un code optimisé.

L'acquisition de compétences pratiques dans la manipulation de bases de données via Java, ainsi que la mise en œuvre de scores personnalisés, ont élargi notre compréhension des applications concrètes de l'Ingénierie Mathématiques et Data Science.

En résumé, ce projet a vraiment boosté notre apprentissage. Il nous a fourni des compétences techniques, une meilleure vision du développement de logiciels, et a renforcé notre savoir pour affronter des projets plus complexes au cours de notre formation d'ingénieurs.