Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2021 – 2022
DEV2
-

DEV2 – Laboratoire Java

Projet - Les échecs

Partie 1

Table des matières

Version du document
25 février 2022

Ne paniquez pas ! Si ce document est si long c'est parce que l'on vous aide beaucoup.

Présentation du projet

Avant de commencer à coder, veuillez prendre connaissance des informations suivantes.

1 Modalités pratiques

Échéances

Nous sommes strict · es sur les échéances. Prenez la précaution de vérifier auprès de votre enseignant · e des modalités spécifiques de remises.

Date limite de remise : le vendredi 1 avril à 18h

Dépôt git

Nous vous avons créé le dépôt `dev2-etd/2022/projet/chess-xxxxx` pour votre projet. C'est avec **ce dépôt** que vous travaillerez et remettrez votre travail aux différentes échéances.

L'utilisation de git tout au long du développement du projet est obligatoire (voir la section suivante).

Évaluation

Nous vous demandons de respecter les consignes suivantes. Le non respect de celles-ci **entrainera inévitablement une perte de points** lors de l'évaluation de votre projet.

Vous devez impérativement :

- suivre à la lettre l'analyse qui vous sera fournie dans l'énoncé qui va suivre, à savoir : la découpe en classes de l'application et la structure du code selon le design-pattern MVC. Il vous est **interdit de créer d'autres méthodes publiques dans les classes** que celles que nous vous demanderons d'écrire. Vous pouvez par contre ajouter autant de méthodes privées que vous le voulez,
- utiliser git (comme précisé ci-dessus). Vous devez faire des **commits** réguliers et au minimum **à chaque fois que c'est demandé** dans l'énoncé faute de quoi **votre travail ne sera pas évalué**,
- utiliser l'environnement de développement NetBeans.

Veuillez de plus bien faire attention à :

- documenter tout votre code. Documenter signifie écrire la Javadoc de vos méthodes, mais aussi inclure des commentaires au sein du code des méthodes lorsque c'est nécessaire. Une méthode qui n'est pas documentée ne **sera pas évaluée**,
- écrire des tests JUnit lorsque c'est demandé. Une méthode qui n'est pas accompagnée de tests unitaires alors que c'est explicitement demandé ne **sera pas évaluée**,
- utiliser les éléments du langage Java les plus adaptés à la situation (un *for-each* quand c'est possible par exemple),

- ▷ soigner la lisibilité. Tout problème de lisibilité (mauvais choix de noms, mauvaise indentation...) ou d'utilisation inadéquate des fonctionnalités du langage sera **sanctionné**. Vous êtes de même invité à **décomposer** vos méthodes afin de faciliter votre écriture, réutiliser du code et le rendre plus lisible,
- ▷ pour toutes les classes du projet : juger s'il est utile de redéfinir les méthodes `toString`, `equals` et `hashCode`.

Pondération

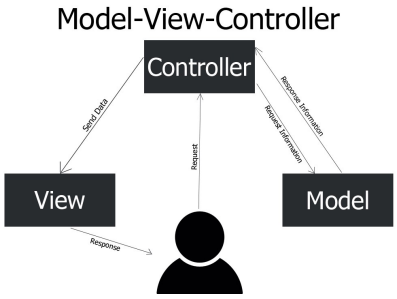
Cette première partie compte pour la moitié dans la cote du projet. Toutefois, une cote définitive ne vous sera attribuée qu'après la **défense orale individuelle** du projet.

⚠ Coopération vs tricherie

Le projet est un **travail individuel**. Il vous est interdit de copier en tout ou en partie le travail d'un autre étudiant. En cas de copie manifeste, le **copieur et le copié seront sanctionnés**. Toutefois, nous encourageons la collaboration : des échanges sur la compréhension de l'énoncé, sur les algorithmes à mettre en œuvre, la comparaison de pratiques, l'aide au débogage.

2 Architecture de l'application

Le patron de conception (*design pattern*) Modèle-Vue-Contrôleur (**MVC**) est une manière de structurer le code adaptée pour la programmation d'applications avec interaction utilisateur. On y distingue la partie métier (modèle et contrôleur) de la partie de présentation (vue) de l'application. Le patron modèle-vue-contrôleur (en abrégé MVC, de l'anglais *model-view-controller*) est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective (voir Wikipedia [1]).



Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- ▷ un modèle : modèle de données et code métier ;
- ▷ une vue : présentation et interaction avec l'utilisateur ;
- ▷ un contrôleur : logique de contrôle, gestion des événements.

La partie modèle (*model*) contiendra pour nous les classes qui définissent les éléments ainsi que la logique principale de l'application. Ces classes seront regroupées dans un package spécifique : `g12345.chess.model` [2].

La partie vue (*view*) concerne les classes qui s'occupent de la présentation et de l'interaction avec l'utilisateur. Elles seront également regroupées dans leur propre package : `g12345.chess.view`.

1. http://fr.wikipedia.org/wiki/Modèle-vue-contrôleur *consulté le 30 janvier 20020*
2. Durant ce travail, `g12345` représente (évidemment) votre matricule.

La partie dynamique du jeu (*controller*) sera contenue dans le package : `g12345.chess.controller`.

Pour finir, **la classe principale** chargée de lancer le jeu sera contenue dans le package `g12345.chess`.

Itération 1

Pour cette première itération, nous allons coder une version fortement simplifiée du jeu, détaillée ci-dessous :

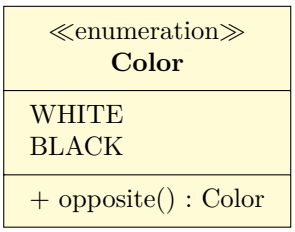
1. Nous ne **prendrons en compte que les pions**. L'ajout des autres pièces du jeu (tour, cavalier, fou, roi et reine) sera entreprise lors de l'itération 2. Lors du lancement d'une partie, les positions du plateau normalement occupées par ces pièces seront laissées vides,
2. Avec la simplification ci-dessus, nous considérerons qu'un joueur est déclaré perdant lorsque, au moment de jouer un coup, il se retrouve bloqué (il ne peut plus jouer aucun coup respectant les règles). Comme les pions ne peuvent se déplacer en arrière (et ne peuvent sortir du plateau !), ceci finira par arriver pour un des deux joueurs. L'autre joueur est alors déclaré vainqueur.

3 Éléments de base du jeu

Commençons par définir les éléments de base qui seront nécessaires pour notre jeu d'échecs, avec en premier lieu l'élément visuel le plus évident : les couleurs.

Toutes les classes de cette section font partie du modèle ; elles doivent donc être placées dans le package `g12345.chess.model`.

3.1 Couleurs



Veuillez tout d'abord créer une énumération `Color` dans votre projet, qui représentera la couleur d'un joueur, d'une case, ou d'une pièce de notre jeu d'échec.

Relire le cours

Vous ne savez plus ce qu'est une énumération ? Si c'est le cas, n'hésitez pas à relire le cours (ou à consulter la documentation Java en ligne). Le but du projet est notamment de vous faire pratiquer les notions de Java, il est donc tout à fait normal que vous éprouviez le besoin de revoir certaines notions avant de vous mettre à coder ! Au besoin, votre professeur de labo peut également vous aider.

Les couleurs admissibles pour une case ou une pièce du plateau de jeu sont `BLACK` et `WHITE`.

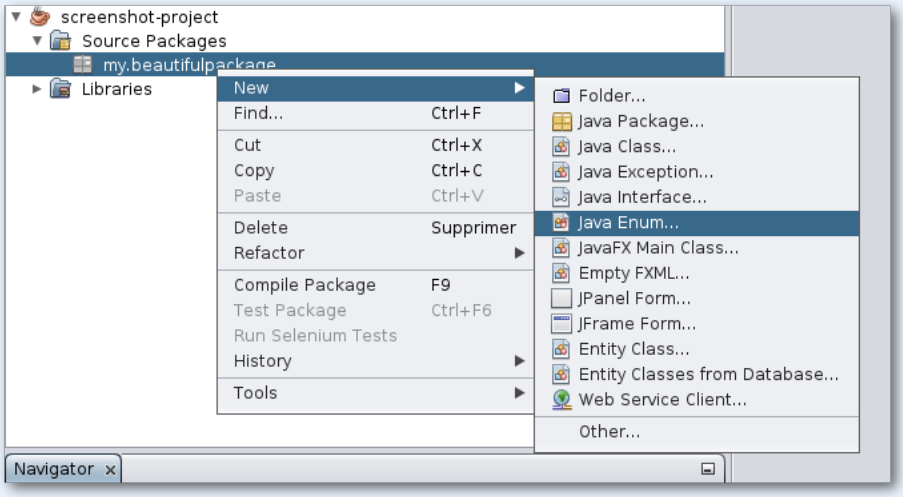
Méthode. Vous ajouterez de plus une méthode `opposite()` renvoyant la couleur opposée d'une couleur donnée (`WHITE` pour `BLACK` et `BLACK` pour `WHITE`). Cette méthode peut sembler étrange à ce stade du développement, mais nous verrons qu'elle nous sera bien utile lorsque nous implémenterons le déplacement des pièces sur le plateau de jeu.

En effet, une pièce donnée ne peut capturer une autre pièce sur le plateau que si cette autre pièce est dans l'autre camp (autrement dit, si la couleur d'une pièce est l'opposée de l'autre).

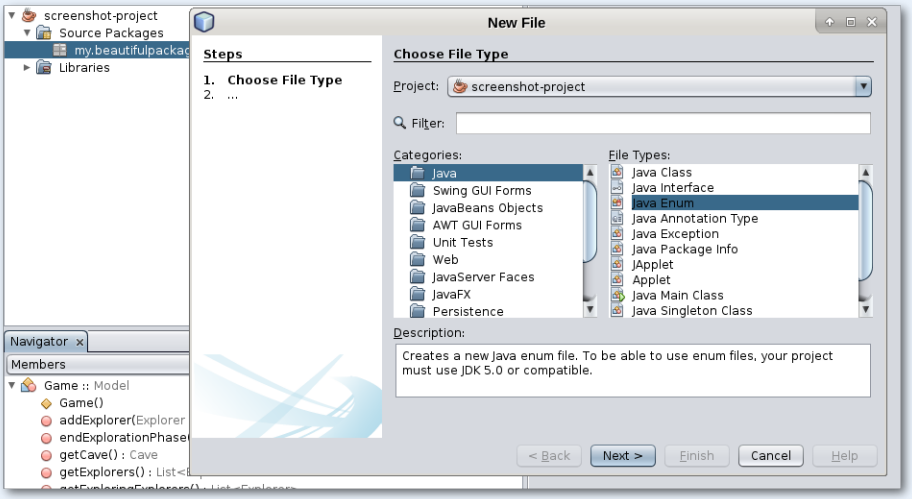
Astuce Netbeans

Pour créer une **énumération** avec Netbeans :

- ▷ clic droit sur le package et glisser la souris sur `new` ;
- ▷ si `Java Enum` apparait, cliquer dessus et se laisser guider ;



- ▷ sinon, cliquer sur `Other`, choisir dans le menu `Java Enum` et se laisser guider.



Lorsque cette énumération est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer l'énumération Color"` et un push sur le dépôt.
Rappel : Les commits demandés **sont obligatoires** sans quoi nous n'évaluerons pas votre travail

3.2 Positions

Position
- row : integer - column : integer
+ Position(row : integer, column : integer) + getRow() : integer + getColumn() : integer

Pour indiquer une position sur un plateau de jeu, nous n'allons pas nous contenter de deux valeurs entières représentant la ligne et la co-

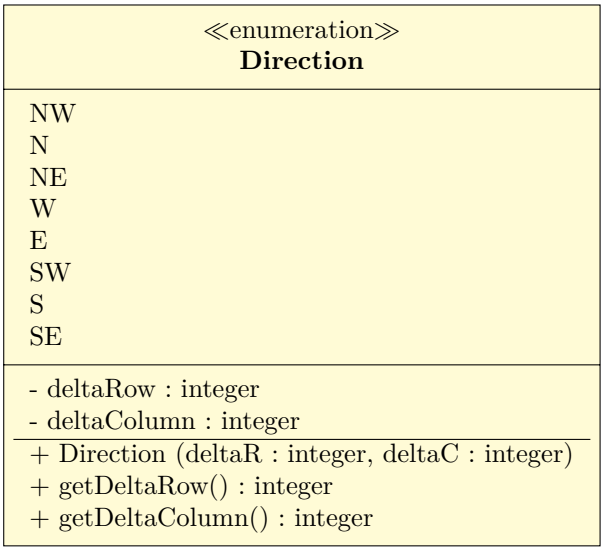
lonne ; Nous allons les rassembler pour former une *position*.

Attributs. Une position est caractérisée par une ligne et une colonne.

Méthodes. On se contentera d'un constructeur et des accesseurs mais aucun mutateur.

Lorsque cette classe est terminée (y compris sa Javadoc), faites un commit `"Créer la classe Position"` et un push sur le dépôt.

3.3 Directions



Une pièce se déplace toujours dans une direction donnée sur le plateau : le nord-ouest (`NW`), le nord (`N`), le nord-est (`NE`), l'ouest (`W`), l'est (`E`), le sud-ouest (`SW`), le sud (`S`) ou le sud-est (`SE`). Comme nous ne considérons que les pions dans cette première itération, les directions `W` et `E` ne seront pas utilisées (un pion ne se déplace jamais horizontalement). Nous les définirons tout de même vu qu'elles seront utiles plus tard, lorsque nous considérerons les déplacements des tours, rois et reines.

Les différentes directions sont illustrées sur la figure ci-dessous.

NorthWest	North	NorthEast
West	**Position de départ**	East
SouthWest	South	SouthEast

Mais évidemment, pour savoir comment déterminer la position obtenue en se déplaçant dans une direction donnée à partir d'une position de départ, il faut d'abord se mettre d'accord sur l'ordre dans lequel nous allons numéroter nos lignes et nos colonnes sur le plateau de jeu.

Ici, conformément à la numérotation utilisée aux échecs (et contrairement à l'habitude en algorithmique et en Java), nous numéroterons nos lignes du bas en haut (du sud vers le nord) et nos colonnes de gauche à droite (de l'ouest à l'est).

Exemple : Si la position de départ est $(2, 2)$ alors celle située au nord est la position $(3, 2)$, celle à l'est est la position $(2, 3)$.

Attributs. Pour mémoriser le déplacement à effectuer (qui est toujours de +1 ou -1 sur une ligne et/ou une colonne) nous allons ajouter deux attributs à cette énumération : l'un représentant le déplacement pour les lignes et l'autre pour les colonnes. Le tableau ci-dessous reprend les valeurs de deltaRow et deltaColumn pour chacun des déplacements :

	deltaRow	deltaColumn
NW	1	-1
N	1	0
NE	1	1
W	0	-1
E	0	1
SW	-1	-1
S	-1	0
SE	-1	1

Méthodes. On se contentera des getters pour les deux attributs deltaRow et deltaColumn ainsi qu'un constructeur. La constructeur se chargera d'initialiser les valeurs de deltaRow et deltaColumn.

Lorsque cette classe est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer l'énumération Direction"` et un push sur le dépôt.

3.4 Compléter la classe Position

Ajoutez à présent une méthode

```
public Position next(Direction dir)
```

à votre classe `Position`. Cette méthode va renvoyer la nouvelle position obtenue en se déplaçant dans la direction donnée, comme illustré dans la section 3.3.

Exemple : La position au nord-est de la position (4,4) est la position (5,5). Ainsi, l'appel à la méthode next sur sur une position d'attributs (4,4) renvoie une position d'attributs (5,5) :

```
Position position1 = new Position(4,4);
Position position2 = position1.next(Direction.NE);
//  position2.getRow() vaut 5 et position2.getColumn() vaut 5
```

Tester le code. Nous aimerions à présent tester la méthode `next` de la classe Position. Veuillez proposer un plan de test et l'implémenter au moyen de tests unitaires JUnit.

⚠ Aucun test ne se lance

Si le rapport de test vous dit qu'aucun test n'a été exécuté ou mentionne que tous les tests ont échoués à cause d'un `NullPointerException` c'est que vous devez mettre à jour votre SUREFIRE – un composant MAVEN utilisé pour les tests.
Pour ce faire, votre `pom.xml` doit contenir :

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>2.22.2</version>
      </plugin>
   </plugins>
</build>
```

Lorsque cette fonctionnalité est terminée (y compris sa Javadoc et les tests), et que les tests JUnit passent, faites un commit avec le message `"Adapter la classe Position"` et un push sur le dépôt.

3.5 Pièces

Comme nous ne considérons que des pions dans cette première itération, la classe `Piece` représentera toujours un pion et sera codée de la sorte :

Piece
- color : Color
+ Piece(color : Color) + getColor() : Color

Constructeur. Le constructeur initialise la pièce avec la couleur donnée.

Attributs. Uniquement la couleur de la pièce.

Méthodes. `getColor` : le getter de l'attribut `color`. Une pièce ne peut pas changer de couleur en cours de partie, il n'y a donc pas besoin de setter pour cet attribut.

Lorsque cette fonctionnalité est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer la classe Piece"` et un push sur le dépôt.

3.6 Joueurs

Player
- color : Color
+ Player(color :Color) + getColor() : Color

Attributs. Pour le moment, nous nous contenterons de la couleur du joueur (correspondant à la couleur de ses pièces).

Constructeur. Le constructeur initialisera un nouveau joueur de couleur donnée.

Méthodes. Uniquement le getteur de l'attribut `color`.

Lorsque cette classe est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer la classe Player"` et un push sur le dépôt.

3.7 Cases

Cette classe représentera une des 64 cases sur le plateau de jeu.

Square
- piece : Piece
+ Square(piece : Piece) + getPiece() : Piece + setPiece(piece : Piece) + isFree() : boolean

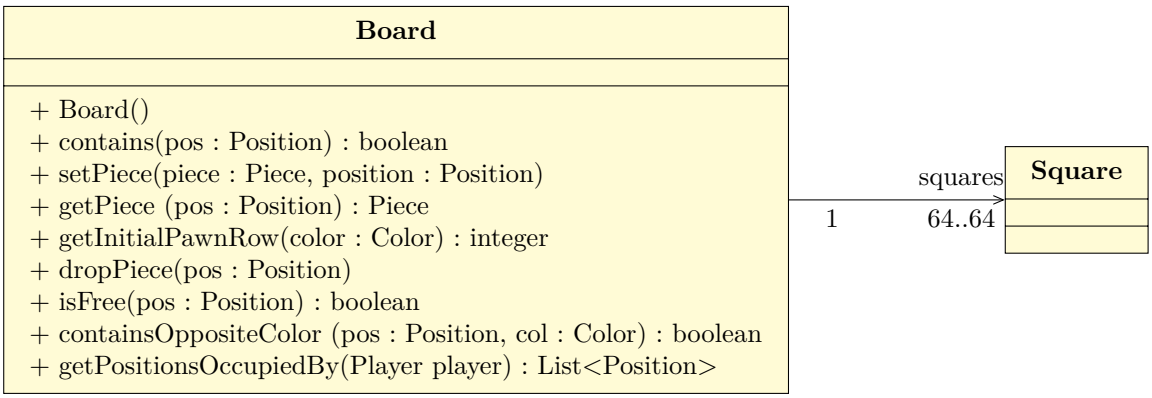
Attributs. Nous ne définirons pour le moment qu'un attribut : la pièce contenue sur la case de jeu (qui sera égale à `null` si la case ne contient aucune pièce). Nous ne définirons pas la couleur de la case pour le moment (nous pourrons toujours le faire plus tard, si nécessaire).

Méthodes. Implémentera les habituels getters et setters pour la pièce. La méthode isFree() renvoie true si la case est libre (ne contient pas de pièce) et false sinon.

Lorsque cette fonctionnalité est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer la classe Square"` et un push sur le dépôt.

3.8 Le plateau de jeu

Cette classe représentera le plateau de jeu (l'échiquier).



Attributs. Uniquement un tableau à deux dimensions de cases.

Constructeur. Le constructeur initialise un nouveau plateau de 8 fois 8 cases (ne contenant aucune pièce).

Méthodes.

- `contains` : renvoie `true` si la position est sur le plateau (ligne et colonne entre 0 et 7) et `false` sinon,
- `getInitialPawnRow` : renvoie 6 pour la couleur `BLACK` et 1 pour la couleur `WHITE`. Cette méthode nous sera utile pour positionner les pièces sur le plateau, ainsi que pour déterminer si un pion est toujours sur sa position initiale ou non.
- `setPiece` : place la pièce passée en paramètre sur le case correspondante du plateau. Déclenche une `IllegalArgumentException` si la position donnée n'est pas sur le plateau.
- `getPiece` : renvoie la pièce située sur la case dont la position est passée en paramètre. Déclenche une `IllegalArgumentException` si la position donnée n'est pas sur le plateau.
- `dropPiece` : supprime la pièce de la case dont la position est passée en paramètre. Déclenche une `IllegalArgumentException` si la position donnée n'est pas sur le plateau.
- `isFree` : renvoie `true` si la case de position donnée est libre (il n'y pas de pièce dessus) et `false` sinon. Déclenche une `IllegalArgumentException` si la position donnée n'est pas sur le plateau.
- `containsOppositeColor` : renvoie `true` si la case dont la position passée en paramètre contient une pièce de la couleur opposée à celle passée en paramètre, et `false` sinon. Cette méthode permettra donc de vérifier qu'une case est occupée par une pièce adverse. Déclenche une `IllegalArgumentException` si la position donnée n'est pas sur le plateau.
- `getPositionsOccupiedBy` : renvoie la liste de toutes les positions occupées par le joueur donné.

Tester le code. Nous aimerions à présent tester les méthodes de la classe Board au moyen de tests unitaires JUnit. Nous vous fournissons les tests, vérifiez qu'ils passent tous. Attention : il vous est interdit de modifier les tests en question.

Lorsque cette classe est terminée (y compris sa Javadoc) et que tous les tests passent, faites un commit avec le message `"Créer la classe Board"` et un push sur le dépôt.

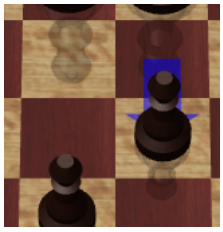
3.9 Compléter la classe Piece

Nous pouvons à présent compléter la classe `Piece` avec une méthode

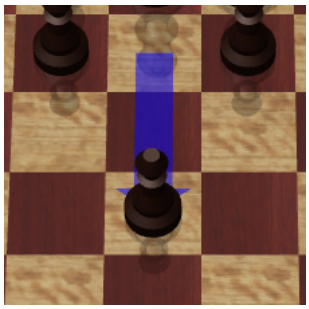
```
public List<Position> getPossibleMoves(Position position, Board board)
```

fournissant la liste des positions possibles pour une pièce située en position donnée sur le board. Nous vous rappelons pour cela les déplacements possibles d'un pion :

1. Un pion peut se déplacer d'**une** case vers le sud (pour les noirs) ou vers le nord (pour les blancs) à condition que cette case soit libre.

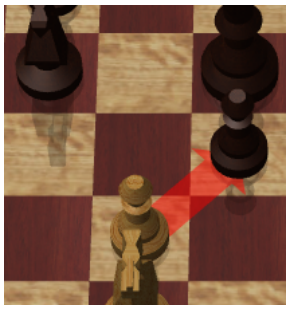


2. Un pion peut se déplacer de **deux** cases vers le sud/nord à condition que ces deux cases soient libre et qu'il s'agit du premier mouvement de ce pion lors de la partie.

   Pour déterminer s'il s'agit de la première fois que le pion se déplace, comme un pion se déplace toujours en avant, il suffit de déterminer si le pion est toujours sur la ligne de départ (en utilisant la méthode `getInitialPawnRow` de la classe `board`)
3. Un pion ne peut se déplacer en diagonale (vers le sud-est/sud-ouest pour un pion noir et vers le nord-est/nord-ouest pour un pion blanc) que s'il capture une pièce adverse.

   
4. Lorsque vous avez pris connaissance des règles du jeu d'échecs, vous avez sans doute entendu parler de la « prise en passant ». Nous laisserons cette règle de côté lors de cette première itération.

Vous êtes désormais prêt à implémenter cette méthode.

Décomposer et commenter son code

Rappel : Nous vous demandons de soigner la lisibilité de votre code. La méthode `getPossibleMoves` que l'on vous demande d'écrire est un peu plus longue que celles écrites jusqu'à présent, elle doit donc naturellement être commentée et, si nécessaire, découpée en sous-méthodes privées.

Tester le code. La méthode `getPossibleMoves` que vous venez d'implémenter est évidemment fondamentale pour le bon fonctionnement du jeu, et il importe donc de

bien la tester. Nous vous conseillons ici de bien réfléchir à un plan de test complet, permettant de tester tous les cas livrant une liste différente de positions : pièce située sur les bords, pièce située au milieu de plateau, couleur de la pièce, etc.

Un premier cas à tester serait par exemple : un pion blanc, se trouvant sur sa case de départ, et dont les deux cases situées devant lui sont libres. Nous vous fournissons un fichier implémentant ce premier cas. Vérifiez que le test passe bien. À vous d'implémenter les autres ! Faites bien attention à **avoir un plan de test aussi complet que possible**.

Attention : Comme le test en question compare des listes de positions, il est indispensable d'avoir correctement redéfini la méthode `equals` de la classe position (pourquoi ?). Si ce n'est déjà fait, faites le maintenant (avant d'écrire les tests).

Lorsque cette fonctionnalité est terminée (y compris sa Javadoc et les tests JUnit), faites un commit avec le message `"Créer et tester la méthode getPossibleMoves"` et un push sur le dépôt.

4 La classe Game

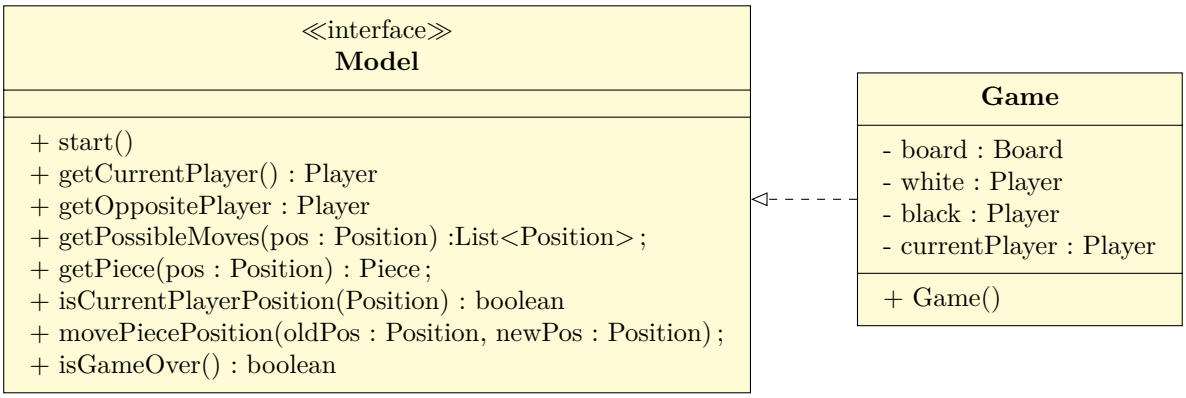
Nous sommes à présent prêts à écrire la classe `Game`. Celle-ci rassemble les éléments nécessaires au jeu, et implémente les différentes étapes de celui-ci. Elle est le point d'accès privilégié pour la vue et le contrôleur.

Cette classe (ainsi que l'interface `Model` associée, voir ci-dessous) doivent se trouver dans le package `g12345.chess.model`.

4.1 L'interface `Model`

Nous introduisons une interface `Model` qui définit les méthodes que doit implémenter la classe `Game`. Certaines ont une action sur le jeu (elles le font avancer) ; d'autres interrogent le jeu sur son état.

Nous vous fournissons cette interface. Voyez la documentation pour comprendre précisément ce que doit faire chaque méthode. La section suivante vous y aidera aussi.



4.2 La classe `Game`

Attributs.

- ▷ `board` le plateau de jeu,
- ▷ `white` le joueur blanc,
- ▷ `black` le joueur noir,
- ▷ `currentPlayer` le joueur courant (soit black, soit white),

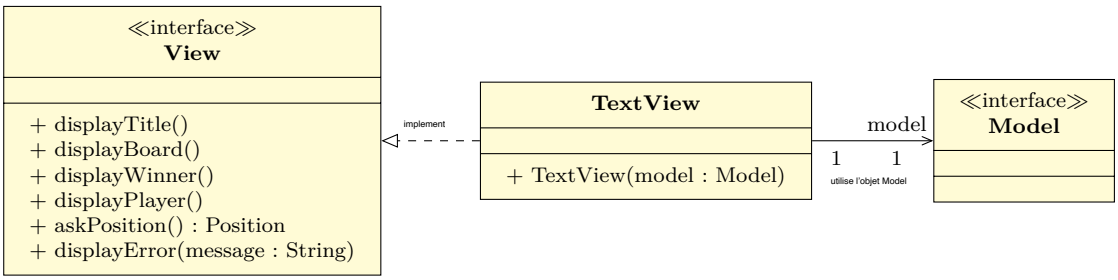
Constructeur. Il se contente de créer un nouveau joueur blanc, un nouveau joueur noir et un nouveau plateau de jeu vide. Attention : la création des pièces n'est pas effectuée dans le constructeur. C'est le rôle de la méthode `start` (voyez la javadoc fournie avec l'interface), qui se chargera aussi d'initialiser le joueur courant à `white` (c'est toujours le joueur blanc qui commence dans une partie d'échecs).

Méthodes. Ce sont celles de l'interface `Model`. L'implémentation de la plupart des méthodes suit immédiatement de la Javadoc (fournie avec l'interface `Model`).

Lorsque cette fonctionnalité est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer et tester la classe Game"` et un push sur le dépôt.

5 La vue

La bonne pratique demande à faire tous les affichages (et les lectures) dans une classe dédiée du package `view`. On vous propose à nouveau d'écrire une interface, et puis une classe implémentant cette interface.



Méthodes.

- ▷ `displayTitle()` : affiche un titre et un message de bienvenue aux joueurs.
- ▷ `displayWinner()` : affiche le joueur gagnant.
- ▷ `displayBoard()` : affiche le plateau de jeu. Voyez la remarque importante à ce propos ci-dessous.
- ▷ `displayPlayer()` : affiche un message invitant le joueur courant (blanc ou noir) à jouer,
- ▷ `askPosition()` : demande une position valide sur le plateau de jeu à l'utilisateur. Voyez la remarque importante à ce propos ci-dessous.
- ▷ `displayError` : affiche le message d'erreur passé en paramètre.

Remarque importante : convivialité du jeu

Lors de l'implémentation des méthodes décrites ci-dessus, vous ferez naturellement attention à avoir une présentation aussi soignée que possible : pas de fautes d'orthographe, robustesse des méthodes de lecture, affichage correct des cases du plateau, numérotation des lignes et colonnes du plateau suivant la convention habituelle des échecs (rappelée ci-dessous) etc. Le soin apporté aux interactions avec l'utilisateur fera partie de nos critères d'évaluation !

Pour rappel, **la numérotation conventionnelle aux échecs** est la suivante : les colonnes sont numérotées de gauche à droite au moyen de lettres minuscules (de **a** à **h**)

et les lignes de bas en haut au moyen des nombres de 1 à 8.Vous êtes par contre **libres de choisir une représentation des pions** qui vous semble la plus adaptée, mais la distinction entre les pièces blanches et noires doit évidemment apparaître clairement (ainsi que les différentes cases du plateau) pour que les joueurs puissent s'y retrouver.

Voici un exemple d'affichage possible : la numérotation est bien respectée, on a choisi de représenter les pions blancs par les lettres `PB` et les pions noirs par les lettres `PN`.

```
  ------------------------
8 |  |  |  |  |  |  |  |  |
  ------------------------
7 |PN|PN|PN|PN|PN|PN|PN|PN|
  ------------------------
6 |  |  |  |  |  |  |  |  |
  ------------------------
5 |  |  |  |  |  |  |  |  |
  ------------------------
4 |  |  |  |  |  |  |  |  |
  ------------------------
3 |  |  |  |  |  |  |  |  |
  ------------------------
2 |PB|PB|PB|PB|PB|PB|PB|PB|
  ------------------------
1 |  |  |  |  |  |  |  |  |
  ------------------------
    a  b  c  d  e  f  g  h
```

Vous devez également prendre garde à demander les positions à l'utilisateur sous la forme conventionnelle : un numéro de ligne de 1 à 8 et une lettre pour la colonne. À vous de transcrire ce numéro de ligne et cette lettre en une position valide sur le plateau.

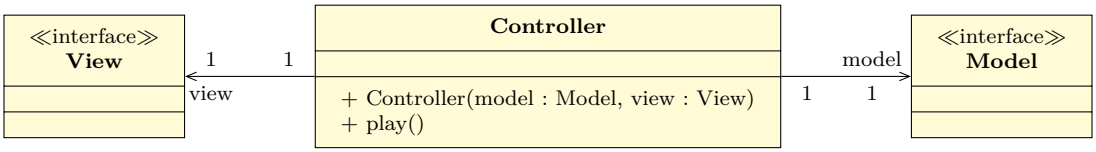
⚠ Tout est dans la vue

Maintenant que vous disposez d'une vue, **absolument tout** l'affichage doit se faire dans cette vue. Aucun affichage ne peut être effectué dans le contrôleur ou dans une classe du modèle par exemple.
Cette bonne pratique permet de facilement changer de vue ; il suffit de remplacer les classes de la vue !

Lorsque la vue est terminée (y compris sa Javadoc), faites un commit avec le message `"Créer la vue"` et un push sur le dépôt.

6 Le contrôleur

La classe `Controller` (située dans le package du même nom) va nous permettre de faire le lien entre la vue et le modèle.



Consructeur. Celui-ci initialise le contrôleur avec la vue et le modèle passés en paramètres.

Méthodes. La méthode `play` pilote le jeu. Voici le canevas de cette méthode (à vous de le compléter) :

```
public void play(){

```

```
    boolean gameIsOver = false;

    view.displayTitle();
    game.start();


     while(!gameIsOver){
       /*
          1) Afficher le plateau de jeu et inviter le joueur courant à jouer,
          2) Demander une position de départ et d arrivée,
          3) Jouer le coup (après avoir vérifier sa validité)
          4) Vérifier si le jeu est terminé, et mettre gameIsOver à jour
        */

      }
     view.displayWinner();

}
```

Faites un commit avec le message `"Ecrire le contrôleur"` et un push sur le dépôt.

La méthode principale de la classe `g12345.chess`, elle, se réduit maintenant au strict minimum.

```
public static void main(String[] args) {
    Model game = new Game();
    Controller controller = new Controller(game, new TextView(game));
    controller.play();
}
```

7 La remise

Félicitations ! Vous venez de terminer l'itération 1 du projet. Avant la remise et le push final sur le dépôt, veuillez vérifier une dernière fois que toutes les consignes (voir la section 1) sont bien respectées.