Rapport projet programmation 3

Par Andrea Randriamalazavola et Quentin Menini.

Table des matières

1. Introduction

Notre groupe, composé de Quentin Menini et Andrea Randriamalazavola, a codé un jeu d'échec mono-utilisateur dans un navigateur, en utilisant le langage java. Ce rapport sera enclin à suivre l'ordre chronologique de l'avancement de notre projet, et un exemplaire de l'ensemble du code est disponible à la fin du rapport.

Nous souhaitons remercier nos mamans respectives ainsi que Samuel Thibault et Jeremy Frey de nous avoir soutenus tout au long de ce projet.

2. Déroulement du projet

a. Coups de bases

Nous avons commencé par implémenter la base du jeu d'échec, à savoir le plateau (Board.java), les pièces du jeu, et leurs déplacements respectifs (Piece.java et les classes portant le même nom anglais de chaque pièce).

Nous avons, dans Board.java, créé un tableau de pièces initialisées à leur position de début de partie et avons implémenté leurs déplacements potentiels dans leur classes respectives. Certains déplacements sont néanmoins factorisables dans Piece.java : `possibleMovesDiagonales()` utilisé par le fou et la reine, `possibleMovesDroit()` utilisé par la tour et la reine, ou encore `possibleMovesSE()` utilisable par toutes les pièces du jeu, (cette dernière correspond à la liste de coup possible privé de celle mettant son roi en échec, nous en parlerons plus en détails dans les chapitres suivant).

Evidemment les fonctions `possibleMoves()` de chaques pièces sont implémentées de façon à ce qu'elle puisse bouger suivant son déplacement sauf en cas d'obstacles. Le cavalier n'est évidemment pas concerné par le problème d'obstacles.

b. Affichage en console

Avant d'attaquer la partie réseau induisant un affichage plus clair du plateau dans un navigateur, nous avons dû tester nos fonctions dans la console.

Pour cela nous avons implémenté la classe TestFonctions.java qui formate un plateau dans la console : les pièces y sont représentées par un couple de lettres dont la première correspond à la couleur de la pièce (« B » pour une pièce noire « W » pour une pièce blanche), et la deuxième le type de la pièce (première lettre du type de la pièce, « P » pour pion, « T » pour le tour etc…).

Il faut noter que le programme implémenté par TestFonctions.java n'est pas interactif et que les coups doivent être entrés en ajoutant les instructions correspondantes dans le `main()`. Il est fonctionnel pour visualiser l'initialisation du plateau et les déplacements des pièces, mais n'est en aucun cas adapté aux coups spéciaux ou aux options comme « undo/redo » implémentés par la suite. Cette fonction n'est qu'une solution alternative avant que la partie réseau ne soit opérationnelle.

c. Envoi de code HTML par le serveur

Pour pouvoir afficher le jeu dans le navigateur, il fallait lui envoyer du code HTML. Pour ce faire nous avons commencé par lui envoyer la page fixe donnée dans le sujet.

Nous avons créé une fonction qui permet de lire un fichier et renvoie un string contenant le code du fichier. Pour rendre ce code lisible par le navigateur, nous avons implémenté `getContentType(Object)`, qui renvoie un string contenant le code HTML de la description du type du fichier. Celui-ci sera insérer dans l'en-tête de la requête que nous envoyons.

Au début on envoyait le string entier puis nous avons décidé de l'envoyer bit par bit d'où la fonction `getBytes(String)`, dont nous parlerons dans le chapitre des problèmes rencontrés.

L'en-tête obtenu est le suivant :

```
HTTP/1.1 200 OK
Server: WebChess localhost:7777
Content-Length: content.getBytes("UTF-8").length
Connection: close
Content-Type: getContentType(fichier);
charset=UTF-8
```

Après cet en-tête, nous envoyons le contenu du fichier à afficher dans le navigateur, l'envoi vers le navigateur se fait à l'aide de la fonction `write(byte[])` appliquée à notre OutputStream.

d. Génération de code HTML

La page du jeu devant être actualisée à chaque déplacement, nous devions implémenter une classe générant du code HTML à envoyer au navigateur : la classe « HTMLGen.java ». Ce code se base sur un même modèle étant donné que le fond et le plateau ne change jamais.

Lors de l'instanciation de cette classe, le tableau passé en paramètres est scanné, et le tableau de jeu est rempli case par case, nous avons donc implémenté la fonction `printPiece()` qui renvoie le code HTML correspondant à la pièce à afficher suivant si la case est vide, si la pièce est noire ou blanche, si c'est un roi, un cavalier ou n'importe quelle autre pièce, et si la case est cliquable ou non. Pour qu'une case soit cliquable il faut qu'elle soit prise par une pièce de la couleur du joueur actuel (ce qui correspond à `currentPlayer` dans la classe Board) ou que ce soit une case vide jouable par la pièce actuellement selectionnée (ce qui correspond à `selectedCase` dans la classe Board).

Pour éviter qu'un joueur ne triche en changeant les paramètres du `get` et selectionne une case non cliquable par exemple, nous avons à chaque fois vérifié dans notre `main()` que le paramètre était valide, nous voulions créer une page HTML sur laquelle le joueur aurait été renvoyé à chaque tentative de triche (avec une bonne grosse image traitant le joueur de tricheur et lui rappelant que tricher c'est mal), mais nous avons plus tard préféré ne rien faire. Le tricheur peut donc tout essayer pour faire un déplacement illégal, rien ne se passera.

Notre `main()` fonctionne comme ceci : s'il y a des paramètres, il fait l'action correspondante et affiche le tableau, sinon il affiche simplement le tableau. A la fin de notre projet, les paramètres possibles sont (dans l'ordre où nous les gérons) :

- `Save` (+ nom de sauvegarde) qui, si le nom n'est pas déjà pris, créé un fichier de sauvegarde pour la partie actuelle.

- `Load` (+ nom de sauvegarde) qui, si le nom existe, charge la partie sauvegardée.
- `Suppr` (+ nom de sauvegarde) qui supprime le fichier de sauvegarde.
- `Delete` (+ nom de sauvegarde) qui demande confirmation pour supprimer le fichier de sauvegarde.
- `Rook`, `Knight`, `Bishop` ou `Queen` qui, si le coup en cours est une promotion, choisi la pièce nommée pour la promotion.
- `NewGame` qui réinitialise le tableau pour une nouvelle partie.
- `Undo` qui annule le coup précédent si un coup a déjà été joué.
- `Redo` qui refait le coup annulé si un coup a été annulé.
- `to` (+ nom de case), qui bouge la pièce selectionnée au nom de case en paramètre si une case est selectionnée et si le coup est valide.
- nom de case, qui selectionne la case en paramètre si elle n'est pas vide et si la pièce sur cette case appartient au joueur à qui c'est le tour de jouer.

3. Extensions réalisées

a. Coups spéciaux

1) Roque

Pour implémenter les différents types ded roque, nous avons dû créer la variable de classe `hasMovedOnce`, qui passe à true dès que la pièce bouge et qui empêche donc le roque. Dans les coups possibles du roi, s'il n'a pas bougé, que la tour correspondante n'a pas bougé, et qu'aucune des positions de déplacement du roi n'est prise par une pièce où peut être prise par une pièce adverse, le roque est possible.

2) Prise en passant

Pour implémenter la prise en passant, nous avons dû créer la variable de classe `mangeableEnPrisePassant`, qui contient le numéro de coup dans lequel nous pouvons prendre la pièce en prise en passant (en effet ce coup doit être fait immédiatement après le mouvement de la pièce. Si un pion est à côté d'un pion adverse qui vient d'effectuer son déplacement de deux cases, le premier peut manger l'autre en se déplaçant en diagonale sur la case qui aurait été prise si le déplacement n'avait été que d'une case.

3) Promotion

Lorsqu'un pion atteint la dernière case du plateau, il se transforme au choix en Reine, Fou, Cavalier ou Tour. L'implémentation du choix n'a pas été compliquée (si le jeu est dans l'état de choix d'une pièce, HTMLgen affiche les images des pièces à selectionner avec un lien qui met le nom de la pièce choisie en paramètre), mais nous avons eu des problèmes avec l'annulation de ce coup (lors du choix de la pièce, nous sommes dans un état au milieu du coup, nous avons donc choisi de ne pas pouvoir annuler pendant cet état, néanmoins elle est de nouveau possible après le choix de la pièce).

b. Enregistrement des coups (undo/redo)

Nous avons créé la classe Coup.java, qui contient toutes les informations nécessaires sur un coup : la pièce déplacée (`movedPiece`), l'ancienne pièce s'il s'agit d'une promotion (`oldPiece`), le fait qu'une pièce ait été mangée ou non (`hasEaten`), si le coup est un petit roque (`isPetitRoque`) ou un grand roque (`isGrandRoque`), la pièce mangée s'il y en a une (`eatenPiece`), la case de départ (`caseDepart`), la case d'arrivée (`caseArrivee`), le numéro du coup (`numeroCoup`), si c'est une prise en passant (`isPriseEnPassant`) et/ou si c'est une promotion (`isPromotion`).

Nous avons aussi créé des variables de classe dans Board.java : `listeCoups` qui est la liste de tous les coups joués, même ceux annulés, `indexMove` qui est le numéro du coup à jouer et `indexMoveMax` qui est le dernier coup qu'on puisse rétablir avec redo.

Grâce à cela, nous avons pu créer la fonction `annulerCoup()` qui remet le plateau dans l'état où il était avant le coup, et la fonction `retablirCoup()` qui rejoue le coup qui a été annulé. Si après un coup annulé on joue un coup de façon normale (sans faire de redo), `indexMoveMax` prend la valeur de `indexMove` et le redo n'est plus possible.

c. Notation de partie

Nous nous sommes servis du tableau de coups implémenté pour le undo/redo pour créer un affichage de la liste des coups joués. Pour ce faire, il suffit de parcourir le tableau de coup et d'écrire la notation du coup grâce à la fonction `afficherCoup()`.

Lorsque l'indice du coup à afficher dépasse l'indice du coup actuel, les coups sont affichés en grisé, ce qui permet de voir les coups qui ont été annulés et peuvent encore être rejoués (si un coup est joué de manière normale, ces coups ne seront plus affichés car il n'est plus possible de les rejouer).

d. Prise de pièces

Nous avons simplement créé deux tableaux de pièces (`whiteEatenPieces` et `blackEatenPieces`) dans Board.java, qui contiennent respectivement toutes les pièces mangées blanches et noires. A chaque annulation de coup, ce tableau est parcouru et toutes les pièces non mangées en sont supprimées.

Pour l'affichage, nous avons choisis d'afficher l'image correspondant à la défunte pièce, pour commémorer son souvenir.

e. If-modified-since

Nous avons remarqué que le chargement des images était un peu long sur certains ordinateurs, nous avons donc décidé de renvoyer le code « 304 not modified » pour les fichiers qui ne changent pas au cours du temps (les images). Pour ce faire nous avons d'abord envoyé dans l'en-tête la date de dernière modification, avec `Last-Modified`. Pour un fichier de type image, nous avons mis une date arbitraire (Tue, 19 Nov 2013 20:35:08 GMT car c'est la date et l'heure à laquelle nous avons codé ceci), alors que pour le reste (la page HTML et le CSS), c'est la

date actuelle qui est renvoyée, ces deux fichiers sont donc automatiquement rechargés. Après avoir fait ceci, nous avons remarqué que, dans la plupart des navigateurs, les images n'étaient pas rechargées, mais le code 304 n'était jamais renvoyé. Nous avons eu quelques problèmes pour récupérer la date qui permettait de savoir si l'on devait renvoyer ce code (nous en parlerons dans la prochaine partie). Une fois cette date récupérée, une simple comparaison permet de savoir si l'on doit recharger le fichier ou non, et si la date de dernière modification est antérieure à la date du `if-modified-since` demandée par le navigateur, il suffit de renvoyer `HTTP/1.1 304 Not Modified` au lieu de `HTTP/1.1 200 OK`.

f. Système de sauvegarde

Nous avons choisi de sauvegarder les parties localement sur le serveur, pour cela nous avons cherché comment écrire un fichier texte en java, et par chance nous sommes tombé sur ce site : http://www.tomsguide.fr/forum/id-553824/ecrire-lire-fichier-txt-java.html qui expliquait comment sauvegarder toute une classe dans un fichier grâce à la serialisation. Ceci était une aubaine car si l'on sauvegardait la classe Board, toute la partie était sauvegardée. Il suffisait juste que toutes les classes utilisées implémentent Serializable. Nous avons donc modifié le code HTML et le code du `main()` pour pouvoir sauvegarder les parties en entrant un nom de sauvegarde, charger une partie et en supprimer une grâce à une liste de sauvegardes. Pour avoir cette liste, nous avons créé la classe SavedGamesList qui contient une ArrayList de String contenant les noms des sauvegardes. Cette classe est elle-même sauvegardée dans le fichier ListeSauvegardes.txt. Lorsque nous lançons le serveur, ce fichier est lu (si il n'existe pas, une exception est générée mais cela ne gène en rien le déroulement), lorsque l'on sauvegarde une partie, si le nom n'est pas déjà utilisé par une autre sauvegarde et si ce nom n'est pas ListeSauvegardes, la partie est sauvegardée dans un fichier correspondant au nom donné et ce nom est ajouté à SavedGamesList, sinon rien ne se passe. A chaque modification de SavedGamesList (ajout ou suppression de sauvegarde) le fichier est réécrit pour pouvoir être accessible même après avoir quitté et relancé le serveur. De cette manière si le fichier n'existait pas déjà, il est créé dès la première sauvegarde.

4. Problèmes rencontrés

a. Annulation de la promotion

Lorsque nous avons codé l'annulation des coups, nous avons eu des problèmes avec l'annulation de la promotion. Certaines fois, la pièce choisie était dédoublée, d'autres fois, ce n'étais pas au bon joueur de jouer. Pour régler le problème du joueur, nous avons choisi de ne pas pouvoir annuler le coup pendant le choix de la pièce, car avant que la pièce ne soit choisie, on n'est pas encore passé au joueur suivant et l'annulation d'un coup passe au joueur précédent. Pour le problème de la pièce qui se dédouble, nous nous sommes rendu compte que cela se produisait si le pion mangeait une pièce au moment de sa promotion, en effet nous n'avions pas pensé que ceci était possible et nous avons utilisé la liste `eatenpiece` de la classe Coup pour sauvegarder le pion, la piece mangée n'était donc pas sauvegardée. Nous avons donc créé la variable de classe `oldpiece` qui sauvegarde le pion et modifié le code pour que la pièce mangée soit bel et bien sauvegardée.

b. Affichage des coups possibles sans les cases mettant notre roi en echec

Il est interdit de jouer un coup mettant son roi en Echec, nous voulions donc que ces coups ne soient pas affichés avec les coups possibles, nous avons donc appelé la fonction `echec()` dans les fonctions `possibleMoves()`, or la fonction `echec()` appelle les `possibleMoves()` de toutes les pièces adverses, ce qui nous a ammené à une boucle infinie et un stack overflow (on n'a pas fait brûler tout le cremi mais pas loin). Nous avons donc abandonné l'idée jusqu'à ce que l'on pense à créer une fonction annexe, `possibleMovesSE()` qui simule tous les coups possible d'une pièce grâce à un clone de `board` et regarde si ce coup met le roi en echec, si c'est le cas, elle enlève le coup de la liste des coups possibles qu'elle renvoit. Cette fonction est appelée dans HTMLGen pour ne pas afficher les coups qui mettent le roi en echec, ainsi que dans `isEchecEtMat()` et `isPat()` mais pas dans `echec()` pour eviter des appels récursifs infinis.

c. Echec et Mat

Lorsque nous avons codé la fonction `isEchecEtMat()` pour la première fois, nous n'avions pas la fonction `possibleMovesSE()`, et nous avions eu du mal à coder la fonction (il fallait regarder tous les coups possibles de toutes les pièces et voir si le roi était toujours en echec) et la fonction ne marchait pas bien, en effet beaucoup de EchecException étaient levées car si un coup testé laissait notre roi en echec, il ne pouvait pas être joué. Cette fonction avait un comportement étrange et nous avons donc décidé de la coder plus tard. Lorsque nous avons codé la fonction `possibleMovesSE()`, il est devenu beaucoup plus facile de coder `isEchecEtMat()`, en effet, il suffisait de regarder la liste des coups possibles renvoyée par `possibleMovesSE()` pour chaque pièce et si au moins un coup était possible, il n'y avait pas Echec et Mat.

d. ConcurrentModificationException

Lorsqu'un coup était annulé, nous devions parcourir les listes de pièces mangées pour en supprimer celles qui n'avaient pas été mangées, le problème est que lorsque l'on supprimait un élément en parcourant la liste, nous avions une exception de type CurrentModificationException, nous avons trouvé la solution sur ce site : http://www.developpez.net/forums/d763054/java/general-java/debuter/probleme-type-java-util-concurrentmodificationexception-lors-suppression/. La solution consiste à parcourir la liste dans l'autre sens, pour ne pas avoir à accéder à l'élément suivant d'un élément supprimé.

e. If-modified-since

Pour pouvoir renvoyer le code « 304 not-modified », il fallait pouvoir comparer la date de modification et la date demandée par le navigateur, or nous avons eu un peu de mal à accéder à la date demandée par le navigateur. Nous avons finalement trouvé la solution sur http://stackoverflow.com/questions/1930158/how-to-parse-date-from-http-last-modified-header.

f. Fin de code HTML

Nous avons remarqué que le code HTML n'était jamais complètement envoyé au navigateur, nous pensions que le problème était le mode d'envoi de la page (la page était envoyée int par int en castant chaque caractère en int) Nous avons donc converti le `String` du code en tableau de `Byte` et l'avons envoyé avec la fonction `write()` de la classe OutputStream qui prend comme paramètre un tableau de `Byte` et l'envoit dans son intégralité. Le problème n'était pas réglé, nous avons donc comparé la taille du tableau de Byte et la taille du String et remarqué qu'elles étaient différentes (en effet l'encodage est différent), nous avons donc changé l'en-tête en remplaçant la taille du `String` du contenu par la taille du contenu converti en tableau de `Byte` et le problème était réglé.

5. Conclusion

Nous avons pris beaucoup de plaisir à réaliser ce projet, chaque nouvelle extension à rajouter était un petit défi et nous étions ravis à chaque fois qu'une nouvelle extension était fonctionnelle. Nous n'avons pas pu réaliser toutes les extensions proposées mais toutes celles que nous avons réalisées sont fonctionnelles et notre jeu d'échecs est complet et suit toutes les règles d'un vrai jeu d'échec (nous n'avons pas géré les fins de parties n'étant pas le pat ni l'echec et mat car nous considèrions qu'étant donné que nous n'avons pas fait de tableau de score, les joueurs pouvaient eux-même décider de la fin de la partie).

Le code du jeu en java n'a pas été très dur (toutes les fonctions de déplacement basique ont été codées en une après midi) mais la partie réseau nous a posé quelques problèmes au début (nous ne savions pas trop comment partir pour créer le serveur). Une fois la partie réseau codé, nous avons créé les extensions que nous avions jugées les plus utiles (les coups spéciaux sont importants car sans eux, ce n'est pas une vraie partie d'échec qui est jouée) et les plus simples à mettre en place (la liste des coups joués ou le cimetière de pièce ne nous ont pas pris beaucoup de temps à mettre en place).

Nous avons aussi beaucoup joué avec le CSS pour avoir une page de jeu à notre goût, et que nous trouvons particulièrement réussie, avec les options dans un bandeau déroulable sur la gauche et un petit Easter Egg en bas de page (avouez-le, vous ne l'auriez pas découvert si on ne vous l'avait pas dit).

Au final nous sommes assez contents de ce que nous avons fait, même si nous aurions aimé implémenter plus d'extensions. Le principal pour nous est que le jeu fonctionne bien.

6. Code

Board.java

```
1.  import java.io.Serializable;
2.  import java.util.ArrayList;
3.  import java.util.HashMap;
4.  import java.util.Iterator;
5.
6.  /**
7.   * Classe instanciant une partie, à savoir le plateau, l'état et la position de
8.   * l'ensemble des pièces de la partie.
9.   */
10. public class Board implements Serializable {
11.
12.         /**
13.          * default serial version ID
14.          */
15.         private static final long serialVersionUID = 1L;
16.
17.         /**
18.          * Tableau de l'ensemble des pièces de la partie.
19.          */
20.         private Piece[] pieces;
21.
22.         /**
23.          * Liste des pièces blanches mangées.
24.          */
25.         private ArrayList<Piece> whiteEatenPieces;
26.
27.         /**
28.          * Liste des pièces noires mangées.
29.          */
30.         private ArrayList<Piece> blackEatenPieces;
31.
32.         /**
33.          * Instance du Roi noir du joueur.
34.          */
35.         private King blackKing;
36.
37.         /**
38.          * Instance du roi blanc du joueur.
39.          */
```

```
40.         private King whiteKing;
41.
42.         /**
43.          * Chaîne de caractère indiquant à qui est la main. Ses valeurs possibles
44.          * sont "Black" ou "White".
45.          */
46.         private String currentPlayer;
47.
48.         /**
49.          * Chaîne de caractère décrivant la case en surbrillance, de "A" à "H" pour
50.          * les abscisses et de 1 à 8 pour les ordonées.
51.          */
52.         private String selectedCase;
53.
54.         /**
55.          * Indice indiquant l'indice du coup courant dans la liste des coup.
56.          */
57.         private int indexMove;
58.
59.         /**
60.          * Indice indiquant l'indice maximum des coups (différent de "indexMove"
61.          * après un ou plusieurs annulerCoup).
62.          */
63.         private int indexMoveMax;
64.
65.         /**
66.          * Table de hashage contenantla liste des coups jouées.
67.          */
68.         private HashMap<Integer, Coup> listeCoups;
69.
70.         /**
71.          * Getteur de la liste des pièces blanches prises.
72.          *
73.          * @return La liste des pièces blanches prises.
74.          */
75.         public ArrayList<Piece> getWhiteEatenPieces() {
76.                 return whiteEatenPieces;
77.         }
78.
79.         /**
80.          * Getteur de la liste des pièces noires prises.
81.          *
82.          * @return La liste des pièces noires prises.
83.          */
84.         public ArrayList<Piece> getBlackEatenPieces() {
85.                 return blackEatenPieces;
86.         }
87.
```

```
88.         /**
89.          * Ajoute une pièce à la liste des pièces prises.
90.          *
91.          * @param p
92.          *            Pièce à ajouter.
93.          * @param color
94.          *            Couleur de la pièce à ajouter.
95.          */
96.         public void estMangee(Piece p, String color) {
97.                 if (color.equals("white"))
98.                         this.whiteEatenPieces.add(p);
99.                 else
100.                         this.blackEatenPieces.add(p);
101.         }
102.
103.         /**
104.          * Getteur de la liste des coups joués au cours de la partie.
105.          *
106.          * @return La liste des coups joués au cours de la partie.
107.          */
108.         public HashMap<Integer, Coup> getListeCoups() {
109.                 return listeCoups;
110.         }
111.
112.         /**
113.          * Ajoute un coup dans la variable "listeCoups".
114.          *
115.          * @param coup
116.          *            Coup à ajouter.
117.          */
118.         public void ajouterCoup(Coup coup) {
119.                 for (int i = this.indexMove; i < this.indexMoveMax; ++i)
120.                         listeCoups.remove(i);
121.                 listeCoups.put(this.indexMove, coup);
122.                 this.coupSuivant();
123.         }
124.
125.         /**
126.          * Met à jour le tableau des pièces mangées.
127.          */
128.         public void piecesNonMangees() {
129.                 for (int i = whiteEatenPieces.size() - 1; i >= 0; i--)
130.                         if (!whiteEatenPieces.get(i).isDead())
131.                                 whiteEatenPieces.remove(whiteEatenPieces.get(i));
132.                 for (int i = blackEatenPieces.size() - 1; i >= 0; i--)
133.                         if (!blackEatenPieces.get(i).isDead())
134.                                 blackEatenPieces.remove(blackEatenPieces.get(i));
135.         }
```

```
136.
137.         /**
138.          * Annule le dernier coup joué dans board.
139.          */
140.         public void annulerCoup() {
141.                 this.indexMove--;
142.                 this.annulerCoup(this.listeCoups.get(this.indexMove));
143.                 this.selectedCase = "00";
144.                 this.nextPlayer();
145.                 /*
146.                  * Iterator<Piece> whiteIt = whiteEatenPieces.iterator(); while
147.                  * (whiteIt.hasNext()){ Piece p = whiteIt.next(); if (!p.isDead()){
148.                  * whiteEatenPieces.remove(p); } } Ceci est impossible Exception in
149.                  * thread "main" java.util.ConcurrentModificationException solution
150.                  * trouvée sur http://www.developpez.net/
151.                  * forums/d763054/java/general-java/debuter/
152.                  * probleme-type-java-util-concurrentmodificationexception
153.                  * -lors-suppression/
154.                  */
155.                 this.piecesNonMangees();
156.         }
157.
158.         /**
159.          * Annule le dernier coup joué par rapprot à un coup.
160.          *
161.          * @param coup
162.          *                                  dernier coup joué.
163.          */
164.         private void annulerCoup(Coup coup) {
165.                 if (coup.getIsPromotion()) {
166.                         int pion = 0;
167.                         for (int i = 0; i < 32; ++i) {
168.                                 if (this.pieces[i].equals(coup.getMovedPiece())) {
169.                                         pion = i;
170.                                         break;
171.                                 }
172.                         }
173.                         this.pieces[pion] = coup.getOldPiece();
174.                         if (coup.getHasEaten()) {
175.                                 coup.getEatenPiece().setColumn(
176.                                                 coup.getCaseArrivee().getColumn());
177.                                 coup.getEatenPiece().setRow(coup.getCaseArrivee().get
    Row());
178.                         }
179.                         this.pieces[pion].setRow(coup.getCaseDepart().getRow());
180.                         this.pieces[pion].setColumn(coup.getCaseDepart().getColumn())
    ;
181.                 } else {
```

```
182.                         if (coup.getIsGrandRoque() || coup.getIsPetitRoque()) {
183.                                 if (coup.getIsGrandRoque())
184.                                         coup.getEatenPiece().setColumn(1);
185.                                 else if (coup.getIsPetitRoque())
186.                                         coup.getEatenPiece().setColumn(8);
187.                                 coup.getEatenPiece().moveOnce(false);
188.                                 coup.getMovedPiece().moveOnce(false);
189.                         } else if (coup.getHasEaten()) {
190.                                 if (coup.getIsPriseEnPassant())
191.                                         coup.getEatenPiece().setRow(coup.getCaseDepar
    t().getRow());
192.                                 else
193.                                         coup.getEatenPiece().setRow(coup.getCaseArriv
    ee().getRow());
194.                                 coup.getEatenPiece().setColumn(
195.                                                 coup.getCaseArrivee().getColumn());
196.                         }
197.                         coup.getMovedPiece().setRow(coup.getCaseDepart().getRow());
198.                         coup.getMovedPiece().setColumn(coup.getCaseDepart().getColumn
    ());
199.                 }
200.         }
201.
202.         /**
203.          * Incrémente indexMove lors d'un coup d'un des joueurs.
204.          */
205.         public void coupSuivant() {
206.                 this.indexMove += 1;
207.                 this.indexMoveMax = this.indexMove;
208.         }
209.
210.         /**
211.          * Décrémente indexMove.
212.          */
213.         public void coupPrecedent() {
214.                 this.indexMove -= 1;
215.         }
216.
217.         /**
218.          * Getteur de indexMove.
219.          *
220.          * @return l'indice du coup courant dans la liste des coup.
221.          */
222.         public int getNumeroCoup() {
223.                 return indexMove;
224.         }
225.
226.         /**
```

```
227.          * Getteur de indexMoveMax.
228.          *
229.          * @return l'indice maximum des coups.
230.          */
231.         public int getNumeroCoupMax() {
232.                 return indexMoveMax;
233.         }
234.
235.         /**
236.          * Affecte la valeur numeroCoup à indexMove.
237.          *
238.          * @param numeroCoup
239.          *                                  valeur à affecter à indexMove.
240.          */
241.         public void setNumeroCoup(int numeroCoup) {
242.                 this.indexMove = numeroCoup;
243.         }
244.
245.         /**
246.          * Getteur indiquant à qui est la main.
247.          *
248.          * @return "Black" ou "White" selon à qui le tour.
249.          */
250.         public String getCurrentPlayer() {
251.                 return currentPlayer;
252.         }
253.
254.         /**
255.          * Getteur indiquant la case en surbrillance.
256.          *
257.          * @return Chaine de caractère de la case à évaluer, de "A" à "H" pour les
258.          *         abscisses et de 1 à 8 pour les ordonnées.
259.          */
260.         public String getSelectedCase() {
261.                 return selectedCase;
262.         }
263.
264.         /**
265.          * Setteur indiquant la case en surbrillance.
266.          *
267.          * @param Chaine
268.          *            de caractère de la case, de "A" à "H" pour les abscisses et de
269.          *            1 à 8 pour les ordonnées.
270.          */
271.         public void setSelectedCase(String selectedCase) {
272.                 this.selectedCase = selectedCase;
273.         }
274.
```

```
275.         /**
276.          * Getteur de la liste des pièces présentes dans board.
277.          *
278.          * @return La liste des pièces présentes dans board.
279.          *
280.          */
281.         public Piece[] getPieces() {
282.                 return pieces;
283.         }
284.
285.         /**
286.          * Getteur du roi noir.
287.          *
288.          * @return le roi du joueur en noir.
289.          */
290.         public King getBlackKing() {
291.                 return blackKing;
292.         }
293.
294.         /**
295.          * setteur du roi noir.
296.          *
297.          * @param blackKing
298.          *                                  roi noir à initialiser.
299.          */
300.         public void setBlackKing(King blackKing) {
301.                 this.blackKing = blackKing;
302.         }
303.
304.         /**
305.          * Getteur du roi blanc.
306.          *
307.          * @return el roi du joueur en blanc.
308.          */
309.         public King getWhiteKing() {
310.                 return whiteKing;
311.         }
312.
313.         /**
314.          * setteur du roi blanc.
315.          *
316.          * @param whiteKing
317.          *                                  roi blanc à initialiser.
318.          */
319.         public void setWhiteKing(King whiteKing) {
320.                 this.whiteKing = whiteKing;
321.         }
322.
```

```
323.         /**
324.          * Fait passer le tour à l'adversaire (lors d'un coup joué).
325.          */
326.         public void nextPlayer() {
327.                 if (this.currentPlayer.equals("black"))
328.                         this.currentPlayer = "white";
329.                 else
330.                         this.currentPlayer = "black";
331.         }
332.
333.         /**
334.          * Duplique une liste de pièces.
335.          *
336.          * @param liste
337.          *                                  la liste des pièces à cloner.
338.          * @return la liste dupliquée.
339.          */
340.         public ArrayList<Piece> cloneListe(ArrayList<Piece> liste) {
341.                 ArrayList<Piece> clone = new ArrayList<Piece>();
342.                 for (int i = 0; i < liste.size(); ++i)
343.                         clone.add(liste.get(i));
344.                 return clone;
345.         }
346.
347.         /**
348.          * Constructeur de board. Initialise un début de partie
349.          * si aucun paramètre n'est donné.
350.          */
351.         public Board() {
352.                 this.whiteEatenPieces = new ArrayList<Piece>();
353.                 this.blackEatenPieces = new ArrayList<Piece>();
354.                 this.listeCoups = new HashMap<Integer, Coup>();
355.                 this.indexMove = 1;
356.                 this.selectedCase = "00";
357.                 this.currentPlayer = "white";
358.                 this.blackKing = new King("black", 8, 5);
359.                 this.whiteKing = new King("white", 1, 5);
360.                 this.pieces = new Piece[32];
361.                 this.pieces[0] = new Rook("black", 8, 1);
362.                 this.pieces[1] = new Knight("black", 8, 2);
363.                 this.pieces[2] = new Bishop("black", 8, 3);
364.                 this.pieces[3] = new Queen("black", 8, 4);
365.                 this.pieces[4] = this.blackKing;
366.                 this.pieces[5] = new Bishop("black", 8, 6);
367.                 this.pieces[6] = new Knight("black", 8, 7);
368.                 this.pieces[7] = new Rook("black", 8, 8);
369.                 for (int i = 0; i < 8; ++i) {
370.                         this.pieces[i + 8] = new Pawn("black", 7, i + 1);
```

```
371.                         this.pieces[i + 16] = new Pawn("white", 2, i + 1);
372.                 }
373.                 this.pieces[24] = new Rook("white", 1, 1);
374.                 this.pieces[25] = new Knight("white", 1, 2);
375.                 this.pieces[26] = new Bishop("white", 1, 3);
376.                 this.pieces[27] = new Queen("white", 1, 4);
377.                 this.pieces[28] = this.whiteKing;
378.                 this.pieces[29] = new Bishop("white", 1, 6);
379.                 this.pieces[30] = new Knight("white", 1, 7);
380.                 this.pieces[31] = new Rook("white", 1, 8);
381.         }
382.
383.         /**
384.          * Constructeur d'un board initialisé avec certain paramètre.
385.          * Peut être initialiser avec tous les paramètres, ou aucun.
386.          *
387.          * @param pieces
388.          *                                  listes des pièces en jeu.
389.          * @param blackKing
390.          *                                  roi du joueur en noir.
391.          * @param whiteKing
392.          *                                  roi du joueur en blanc.
393.          */
394.         public Board(Piece[] pieces, King blackKing, King whiteKing,
395.                         ArrayList<Piece> whiteEatenPieces, ArrayList<Piece>
    blackEatenPieces) {
396.                 this.listeCoups = new HashMap<Integer, Coup>();
397.                 this.indexMove = 1;
398.                 this.selectedCase = "00";
399.                 this.currentPlayer = "white";
400.                 this.whiteEatenPieces = this.cloneListe(whiteEatenPieces);
401.                 this.blackEatenPieces = this.cloneListe(blackEatenPieces);
402.                 this.pieces = new Piece[32];
403.                 for (int i = 0; i < 32; ++i) {
404.                         this.pieces[i] = pieces[i].clone();
405.                         if (this.pieces[i] instanceof King)
406.                                 if (this.pieces[i].getColor().equals("black"))
407.                                         this.blackKing = (King) this.pieces[i];
408.                                 else
409.                                         this.whiteKing = (King) this.pieces[i];
410.                 }
411.         }
412.
413.         /**
414.          * Constructeur d'un board initialisé avec tous les paramètres.
415.          * Peut être initialiser avec certains paramètres, ou aucun.
416.          *
417.          * @param pieces
```

```
418.          *                                  listes des pièces en jeu.
419.          * @param blackKing
420.          *                                  roi du joueur en noir.
421.          * @param whiteKing
422.          *                                  roi du joueur en blanc.
423.          * @param whiteEatenPieces
424.          *                                  liste des pièces blanche prises.
425.          * @param blackEatenPieces
426.          *                                  liste des pièces noires prises.
427.          * @param currentPlayer
428.          *                                  joueur font c'est le tour.
429.          * @param selectedCase
430.          *                                  case en surbrillance.
431.          * @param numeroCoup
432.          *                                  indice du coup courant dans la liste des
    coups.
433.          * @param numeroCoupMax
434.          *                                  indice maximum d'un coup dans la liste des
    coups.
435.          * @param lC
436.          *                                  liste des coups joués au cours de la
    partie.
437.          */
438.         @SuppressWarnings("unchecked")
439.         public Board(Piece[] pieces, King blackKing, King whiteKing,
440.                         ArrayList<Piece> whiteEatenPieces,
441.                         ArrayList<Piece> blackEatenPieces, String currentPlayer,
442.                         String selectedCase, int numeroCoup, int numeroCoupMax,
443.                         HashMap<Integer, Coup> lC) {
444.                 this.listeCoups = (HashMap<Integer, Coup>) lC.clone();
445.                 this.indexMove = numeroCoup;
446.                 this.indexMoveMax = numeroCoupMax;
447.                 this.selectedCase = selectedCase;
448.                 this.currentPlayer = currentPlayer;
449.                 this.whiteEatenPieces = this.cloneListe(whiteEatenPieces);
450.                 this.blackEatenPieces = this.cloneListe(blackEatenPieces);
451.                 this.pieces = new Piece[32];
452.                 for (int i = 0; i < 32; ++i) {
453.                         this.pieces[i] = pieces[i].clone();
454.                         if (this.pieces[i] instanceof King)
455.                                 if (this.pieces[i].getColor().equals("black"))
456.                                         this.blackKing = (King) this.pieces[i];
457.                                 else
458.                                         this.whiteKing = (King) this.pieces[i];
459.                 }
460.         }
461.
462.         /**
```

```
463.          * retourne un clone de board.
464.          */
465.         public Board clone() {
466.                 return new Board(this.pieces, this.blackKing, this.whiteKing,
467.                                 this.whiteEatenPieces, this.blackEatenPieces,
468.                                 this.currentPlayer, this.selectedCase,
    this.indexMove,
469.                                 this.indexMoveMax, this.listeCoups);
470.         }
471.
472.         /**
473.          * vérifie si la case (row,column) est vide.
474.          *
475.          * @param row
476.          *                                 position en abscisse de la case.
477.          * @param column
478.          *                                 position en ordonnée de la case.
479.          * @return         vrai si la case est vide, faux si la case est
480.          *                         occupée par une pièce.
481.          */
482.         public boolean isEmpty(int row, int column) {
483.                 for (int i = 0; i < 32; ++i)
484.                         if (this.pieces[i].getRow() == row)
485.                                 if (this.pieces[i].getColumn() == column)
486.                                         return false;
487.                 return true;
488.         }
489.
490.         /**
491.          * vérifie si la case (caseJeu) est vide.
492.          *
493.          * @param caseJeu
494.          *                                         coordonnees en chaine de
    caractères
495.          *                                         de la case.
496.          * @return vrai si la case est vide, faux si la case est
497.          *                         occupée par une pièce.
498.          */
499.         public boolean isEmpty(String caseJeu) {
500.                 assert (!(caseJeu.charAt(0) < 'A') && !(caseJeu.charAt(0) > 'H')
501.                                 && !(caseJeu.charAt(1) < '1') && !(caseJeu.charAt(1)
    > '8'));
502.                 int column = caseJeu.charAt(0) - 'A' + '1' - 48;
503.                 int row = caseJeu.charAt(1) - 48;
504.                 return this.isEmpty(row, column);
505.         }
506.
507.         /**
```

```
508.          * vérifie si la case (row,column) contient une pièce blanche.
509.          *
510.          * @param row
511.          *                                  position en abscisse de la case.
512.          * @param column
513.          *                                  position en ordonnée de la case.
514.          * @return         vrai si la case contient une pièce blanche,
515.          *                         faux si elle est noire ou si elle est vide.
516.          */
517.         public boolean isWhite(int row, int column) {
518.                 for (int i = 0; i < 32; ++i)
519.                         if (this.pieces[i].getRow() == row
520.                                         && this.pieces[i].getColumn() == column)
521.                                 return this.pieces[i].isWhite();
522.                 return false;
523.         }
524.
525.         /**
526.          * vérifie si la case (row,column) contient une pièce noire.
527.          *
528.          * @param row
529.          *                                  position en abscisse de la case.
530.          * @param column
531.          *                                  position en ordonnée de la case.
532.          * @return        vrai si la case contient une pièce noire,
533.          *                         faux si elle est blanche ou si elle est vide.
534.          */
535.         public boolean isBlack(int row, int column) {
536.                 for (int i = 0; i < 32; ++i)
537.                         if (this.pieces[i].getRow() == row
538.                                         && this.pieces[i].getColumn() == column)
539.                                 return this.pieces[i].isBlack();
540.                 return false;
541.         }
542.
543.         /**
544.          * Ajoute dans une liste toutes les cases contigües à une pièce.
545.          *
546.          * @param p
547.          *                                  pièce à évaluer
548.          * @return la liste des cases contigües à la pièce.
549.          */
550.         public ArrayList<Square> arroundSquares(Piece p) {
551.                 ArrayList<Square> arround = new ArrayList<Square>();
552.                 Square haut = new Square(p.getRow() + 1, p.getColumn());
553.                 if (haut.isRealSquare())
554.                         arround.add(haut);
555.                 Square bas = new Square(p.getRow() - 1, p.getColumn());
```

```
556.                 if (bas.isRealSquare())
557.                         arround.add(bas);
558.                 Square droite = new Square(p.getRow(), p.getColumn() + 1);
559.                 if (droite.isRealSquare())
560.                         arround.add(droite);
561.                 Square gauche = new Square(p.getRow(), p.getColumn() - 1);
562.                 if (gauche.isRealSquare())
563.                         arround.add(gauche);
564.                 Square hautdroite = new Square(p.getRow() + 1, p.getColumn() + 1);
565.                 if (hautdroite.isRealSquare())
566.                         arround.add(hautdroite);
567.                 Square hautgauche = new Square(p.getRow() + 1, p.getColumn() - 1);
568.                 if (hautgauche.isRealSquare())
569.                         arround.add(hautgauche);
570.                 Square basdroite = new Square(p.getRow() - 1, p.getColumn() + 1);
571.                 if (basdroite.isRealSquare())
572.                         arround.add(basdroite);
573.                 Square basgauche = new Square(p.getRow() - 1, p.getColumn() - 1);
574.                 if (basgauche.isRealSquare())
575.                         arround.add(basgauche);
576.                 return arround;
577.         }
578.
579.         /**
580.          * Retourne la liste des cases où l'adversaire peut prendre au
581.          * prochain coup, en d'autres termes les cases qui mettrait le
582.          * roi de la couleur indiqué en echec.
583.          *
584.          * @param color
585.          *                                  couleur du roi qui serait en echec.
586.          * @return la liste des cases.
587.          */
588.         public ArrayList<Square> echec(String color) {
589.                 ArrayList<Square> echecList = new ArrayList<Square>();
590.                 if (color.equals("black"))
591.                         for (int i = 0; i < 32; ++i)
592.                                 if (this.pieces[i].isWhite())
593.                                         if (this.pieces[i] instanceof King)
594.                                                 echecList.addAll(this.arroundSquares(
    this.pieces[i]));
595.                                         else
596.                                                 echecList.addAll(this.pieces[i].possi
    bleMoves(this));
597.                 if (color.equals("white"))
598.                         for (int i = 0; i < 32; ++i)
599.                                 if (this.pieces[i].isBlack())
600.                                         if (this.pieces[i] instanceof King)
```

```
601.                                                 echecList.addAll(this.arroundSquares(
    this.pieces[i]));
602.                                         else
603.                                                 echecList.addAll(this.pieces[i].possi
    bleMoves(this));
604.                 return echecList;
605.         }
606.
607.         /**
608.          * Détermine s'il y a echec et mat du joueur jouant color .
609.          *
610.          * @param color
611.          *                                  couleur du joueur à évaluer
612.          * @return vrai si le joueur color est en echec et mat, faux sinon.
613.          */
614.         public boolean isEchecEtMat(String color) {
615.                 for (int i = 0; i < 32; ++i) {
616.                         if (this.pieces[i].getColor().equals(this.currentPlayer)) {
617.                                 ArrayList<Square> coups =
    this.pieces[i].possibleMovesSE(this);
618.                                 if (coups.size() > 0)
619.                                         return false;
620.                         }
621.                 }
622.                 return true;
623.         }
624.
625.         /**
626.          * Détermine si la case (row,column) est prenable par
627.          * l'adversaire au coup suivant. (le joueur considéré
628.          * correspond à la couleur passée en paramètres).
629.          *
630.          * @param color
631.          *                                  couleur du joueur
632.          * @param row
633.          *                                  coordonnée en abscisse de la case.
634.          * @param column
635.          *                                  coordonnée en ordonnée de la case.
636.          * @return vrai si la case met le joueur en echec, faux sinon
637.          */
638.         public boolean isEchec(String color, int row, int column) {
639.                 ArrayList<Square> echecList = this.echec(color);
640.                 return this.isEchec(echecList, row, column);
641.         }
642.
643.         /**
644.          * Détermine si la case (row,column) est prenable par
645.          * l'adversaire au coup suivant. (le joueur considéré
```

```
646.          * dépend de la liste des case passée).
647.          *
648.          * @param echecList
649.          *                                  liste des cases mettant le joueur en
    echec.
650.          * @param row
651.          *                                  coordonnée en abscisse de la case.
652.          * @param column
653.          *                                  coordonnée en ordonnée de la case.
654.          * @return vrai si la case met le joueur en echec, faux sinon
655.          */
656.         public boolean isEchec(ArrayList<Square> echecList, int row, int column) {
657.                 Iterator<Square> it = echecList.iterator();
658.                 while (it.hasNext()) {
659.                         Square s = it.next();
660.                         if (s.isThisSquare(row, column))
661.                                 return true;
662.                 }
663.                 return false;
664.         }
665.
666.         /**
667.          * getteur de la pièce contenu dans la case (row,column).
668.          *
669.          * @param row
670.          *                                  coordonnée en abscisse de la case.
671.          * @param column
672.          *                                  coordonnée en ordonnée de la case.
673.          * @return la pièce contenu dans la case.
674.          */
675.         public Piece getPiece(int row, int column) {
676.                 assert (!this.isEmpty(row, column));
677.                 int i = 0;
678.                 while (i < 32
679.                                 && !(this.getPieces()[i].getRow() == row &&
    this.getPieces()[i]
680.                                                 .getColumn() == column))
681.                         i++;
682.                 return this.getPieces()[i];
683.         }
684.
685.         /**
686.          * getteur de la pièce contenu dans la case (caseJeu).
687.          *
688.          * @param caseJeu
689.          *                                  coordonnée en chaîne de caractère de la
    case.
690.          * @return la pièce contenu dans la case.
```

```
691.          * @throws OutOfBoardException
692.          * @throws NonPossibleMoveException
693.          */
694.         public Piece getPiece(String caseJeu) throws OutOfBoardException,
695.                         NonPossibleMoveException {
696.                 assert (!(caseJeu.charAt(0) < 'A') && !(caseJeu.charAt(0) > 'H')
697.                                 && !(caseJeu.charAt(1) < '1') && !(caseJeu.charAt(1)
    > '8'));
698.                 int column = caseJeu.charAt(0) - 'A' + '1' - 48;
699.                 int row = caseJeu.charAt(1) - 48;
700.                 return this.getPiece(row, column);
701.         }
702.
703.         /**
704.          * Déplace la pièce contenu dans la case (row1,column1) vers
705.          * la case (row2,column2).
706.          *
707.          * @param row1
708.          *                                  coordonnée en abscisse de la case
    initiale.
709.          * @param column1
710.          *                                  coordonnée en ordonnée de la case
    initiale.
711.          * @param row2
712.          *                                  coordonnée en abscisse de la case de
    destination.
713.          * @param column2
714.          *                                  coordonnée en ordonnée de la case de
    destination.
715.          * @throws OutOfBoardException
716.          *                                  exception renvoyée si les coordonnées
    explicitées
717.          *                                  ne corresondent pas à une case du plateau.
718.          * @throws NonPossibleMoveException
719.          *                                  exception renvoyée si le coup n'est pas
    jouable.
720.          */
721.         public void deplacerPiece(int row1, int column1, int row2, int column2)
722.                         throws OutOfBoardException, NonPossibleMoveException {
723.                 this.getPiece(row1, column1).deplacerPiece(this, row2, column2);
724.         }
725.
726.         /**
727.          * Déplace la pièce contenu dans la case (caseDepart) vers
728.          * la case (caseArrivee).
729.          *
730.          * @param caseDepart
```

```
731.          *                                          coordonnees en chaine de
    caractères
732.          *                                          de la case de départ.
733.          * @param caseArrivee
734.          *                                          coordonnees en chaine de
    caractères
735.          *                                          de la case d'arrivée.
736.          * @throws OutOfBoardException
737.          *                                  exception renvoyée si les coordonnées
    explicitées
738.          *                                  ne corresondent pas à une case du plateau.
739.          * @throws NonPossibleMoveException
740.          *                                  exception renvoyée si le coup n'est pas
    jouable.
741.          */
742.         public void deplacerPiece(String caseDepart, String caseArrivee)
743.                         throws OutOfBoardException, NonPossibleMoveException {
744.                 assert (!(caseDepart.charAt(0) < 'A') && !(caseDepart.charAt(0) >
    'H')
745.                                 && !(caseDepart.charAt(1) < '1')
746.                                 && !(caseDepart.charAt(1) > '8')
747.                                 && !(caseArrivee.charAt(0) < 'A')
748.                                 && !(caseArrivee.charAt(0) > 'H')
749.                                 && !(caseArrivee.charAt(1) < '1') &&
    !(caseArrivee.charAt(1) > '8'));
750.                 int column1 = caseDepart.charAt(0) - 'A' + '1' - 48;
751.                 int column2 = caseArrivee.charAt(0) - 'A' + '1' - 48;
752.                 int row1 = caseDepart.charAt(1) - 48;
753.                 int row2 = caseArrivee.charAt(1) - 48;
754.                 this.deplacerPiece(row1, column1, row2, column2);
755.         }
756.
757.         /**
758.          * initialise une promotion
759.          *
760.          * @param piece
761.          *                                  pièce à remplacer lors de la promotion
762.          *                                  les choix possibles sont "Rook", "Knight",
763.          *                                  "Bishop" ou "Queen".
764.          */
765.         public void setPromotion(String piece) {
766.                 Boolean promotion = false;
767.                 Coup c = this.getListeCoups().get(this.getNumeroCoup());
768.                 if (c.getIsPromotion() && c.getMovedPiece() instanceof Pawn)
769.                         promotion = true;
770.                 assert (promotion);
771.                 Piece pion = c.getMovedPiece();
772.                 Piece p = null;
```

```
773.                 if (piece.equals("Rook"))
774.                         p = new Rook(pion.getColor(), pion.getRow(),
    pion.getColumn());
775.                 else if (piece.equals("Knight"))
776.                         p = new Knight(pion.getColor(), pion.getRow(),
    pion.getColumn());
777.                 else if (piece.equals("Bishop"))
778.                         p = new Bishop(pion.getColor(), pion.getRow(),
    pion.getColumn());
779.                 else if (piece.equals("Queen"))
780.                         p = new Queen(pion.getColor(), pion.getRow(),
    pion.getColumn());
781.                 int numPiece = 0;
782.                 for (int i = 0; i < 32; ++i) {
783.                         if (this.pieces[i].equals(c.getMovedPiece())) {
784.                                 numPiece = i;
785.                                 break;
786.                         }
787.                 }
788.                 this.pieces[numPiece] = p;
789.                 c.setMovedPiece(p);
790.                 this.coupSuivant();
791.         }
792.
793.         /**
794.          * Rétablit un coup, (possible après un ou plusieurs annulerCoup)
795.          *
796.          * @throws OutOfBoardException
797.          * @throws NonPossibleMoveException
798.          */
799.         public void retablirCoup() throws OutOfBoardException,
800.                         NonPossibleMoveException {
801.                 int max = this.indexMoveMax;
802.                 HashMap<Integer, Coup> hash = new HashMap<Integer, Coup>();
803.                 for (int i = this.indexMove; i < this.indexMoveMax; ++i)
804.                         hash.put(i, this.listeCoups.get(i));
805.                 Coup c = this.listeCoups.get(this.indexMove);
806.                 Piece saveNewPiece = c.getMovedPiece();
807.                 this.deplacerPiece(c.getCaseDepart().getNomCase(), c.getCaseArrivee()
808.                                 .getNomCase());
809.                 if (c.getIsPromotion()) {
810.                         if (saveNewPiece instanceof Rook)
811.                                 this.setPromotion("Rook");
812.                         else if (saveNewPiece instanceof Knight)
813.                                 this.setPromotion("Knight");
814.                         else if (saveNewPiece instanceof Bishop)
815.                                 this.setPromotion("Bishop");
816.                         else if (saveNewPiece instanceof Queen)
```

```
817.                                    this.setPromotion("Queen");
818.                    }
819.                    this.indexMoveMax = max;
820.                    for (int i = this.indexMove; i < this.indexMoveMax; ++i)
821.                            this.listeCoups.put(i, hash.get(i));
822.            }
823.
824.            /**
825.             * Evalue s'il y a un cas de pat sur le plateau.
826.             *
827.             * @return
828.             */
829.            public Boolean isPat() {
830.                    for (int i = 0; i < 32; ++i) {
831.                            if (this.pieces[i].getColor().equals(this.currentPlayer)) {
832.                                    if (this.pieces[i] instanceof King)
833.                                            if (this.isEchec(this.pieces[i].getColor(),
834.                                                            this.pieces[i].getRow(),
    this.pieces[i].getColumn()))
835.                                                    return false;
836.                                    ArrayList<Square> coups =
    this.pieces[i].possibleMovesSE(this);
837.                                    if (coups.size() > 0)
838.                                            return false;
839.                            }
840.                    }
841.                    return true;
842.            }
843.    }
```

Piece.java

```java
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Classe instanciant une piece. Elle comprend en variable de classe sa couleur,
 * sa position, et son type (en chaine de caractère).
 */
abstract public class Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Couleur de la piece ("black" ou "white").
	 */
	private String color;

	/**
	 * Position de la pièce en hauteur (de 1 à 8 et 0 si pièce perdue).
	 */
	private int row;

	/**
	 * Position de la pièce en largeur (de 1 à 8 et 0 si pièce perdue).
	 */
	private int column;

	/**
	 * Chaine de caractère du type la pièce.
	 */
	private String nom;

	/**
	 * Caractère identifiant la pièce.
	 */
	private String shortcut;

	/**
	 * Booleen indiquant si la piece s'est deplacée au moins une fois dans la
	 * partie.
	 */
	private boolean hasMovedOnce;
```

```
46.
47.         /**
48.          * Numero du coup pour lequel un pion peut être mangé en prise en passant
49.          */
50.         private int mangeableEnPrisePassant;
51.
52.         /**
53.          * getter mangeableEnPrisePassant
54.          *
55.          * @return mangeableEnPrisePassant
56.          */
57.         public int getMangeableEnPrisePassant() {
58.                 return mangeableEnPrisePassant;
59.         }
60.
61.         /**
62.          * setter mangeableEnPrisePassant
63.          *
64.          * @param mangeableEnPrisePassant
65.          */
66.         public void setMangeableEnPrisePassant(int mangeableEnPrisePassant) {
67.                 this.mangeableEnPrisePassant = mangeableEnPrisePassant;
68.         }
69.
70.         /**
71.          * Renvoie un clone de la pièce.
72.          */
73.         protected abstract Piece clone();
74.
75.         /**
76.          * Retourne la liste des cases où la liste peut jouer, (sans prendre en
77.          * compte en compte les cases mettant se mettant en échec).
78.          *
79.          * @param board
80.          *            La partie en cours.
81.          * @return Un tableau de cases.
82.          */
83.         abstract ArrayList<Square> possibleMoves(Board board);
84.
85.         /**
86.          * Setteur du nom de la pièce.
87.          *
88.          * @param nom
89.          *            Nom de la pièce.
90.          */
91.         public void setNom(String nom) {
92.                 this.nom = nom;
93.         }
```

```
94.
95.         /**
96.          * Setteur du caractère identifiant la pièce.
97.          *
98.          * @param shortcut
99.          *            Caractère identifiant la pièce.
100.          */
101.         public void setShortcut(String shortcut) {
102.                 this.shortcut = shortcut;
103.         }
104.
105.         /**
106.          * Getteur du nom de la pièce.
107.          *
108.          * @return Le nom de la pièce
109.          */
110.         public String getNom() {
111.                 return nom;
112.         }
113.
114.         /**
115.          * Getteur du caractère identifiant la pièce.
116.          *
117.          * @return Le caractère identifiant la pièce.
118.          */
119.         public String getShortcut() {
120.                 return shortcut;
121.         }
122.
123.         /**
124.          * Retourne vrai si la pièce est noire, faux sinon.
125.          *
126.          * @return un booleen indiquant si la pièce est noire.
127.          */
128.         public boolean isBlack() {
129.                 return (this.color.equals("black"));
130.         }
131.
132.         /**
133.          * Retourne vrai si la pièce est blanche, faux sinon.
134.          *
135.          * @return un booleen indiquant si la pièce est blanche.
136.          */
137.         public boolean isWhite() {
138.                 return (this.color.equals("white"));
139.         }
140.
141.         /**
```

```
142.          * Getteur de la couleur de la pièce.
143.          *
144.          * @return La couleur de la pièce.
145.          */
146.         public String getColor() {
147.                 return color;
148.         }
149.
150.         /**
151.          * Setteur de la couleur de la pièce.
152.          *
153.          * @param color
154.          *            La couleur de la pièce.
155.          */
156.         public void setColor(String color) {
157.                 this.color = color;
158.         }
159.
160.         /**
161.          * Getteur de la valeur en ordonnée de la pièce.
162.          *
163.          * @return La valeur en ordonnée de la pièce.
164.          */
165.         public int getRow() {
166.                 return row;
167.         }
168.
169.         /**
170.          * Setteur de la valeur en ordonnée de la pièce.
171.          *
172.          * @param row
173.          *            La valeur en ordonnée de la pièce.
174.          */
175.         public void setRow(int row) {
176.                 this.row = row;
177.         }
178.
179.         /**
180.          * Getteur de la valeur en abscisse de la pièce.
181.          *
182.          * @return La valeur en abscisse de la pièce.
183.          */
184.         public int getColumn() {
185.                 return column;
186.         }
187.
188.         /**
189.          * Setteur de la valeur en abscisse de la pièce.
```

```
190.          *
191.          * @param column
192.          *            La valeur en absisse de la pièce.
193.          */
194.         public void setColumn(int column) {
195.                 this.column = column;
196.         }
197.
198.         /**
199.          * Setteur du booleen indiquant si la pièce a bougé au moins une fois durant
200.          * la partie.
201.          *
202.          * @param b
203.          *            Le booleen true ou false.
204.          */
205.         public void moveOnce(boolean b) {
206.                 this.hasMovedOnce = b;
207.         }
208.
209.         /**
210.          * Retourne vrai si la pièce a bougé au moins une fois durant la partie,
211.          * faux sinon.
212.          *
213.          * @return Le booleen indiquant si la pièce a bougé au moins une fois durant
214.          *         la partie.
215.          */
216.         public boolean hasMovedOnce() {
217.                 return this.hasMovedOnce;
218.         }
219.
220.         /**
221.          * Retourne vrai si la pièce est prise faux sinon.
222.          *
223.          * @return Le booleen indiquant si la pièce est prise.
224.          */
225.         public boolean isDead() {
226.                 return ((this.getRow() == 0) && (this.getColumn() == 0));
227.         }
228.
229.         /**
230.          * Retourne vrai si la case décrite par les paramètres "row" et "column
231.          * contient une pièce adverse.
232.          *
233.          * @param board
234.          *            La partie en cours.
235.          * @param row
236.          *            La valeur en ordonnée de la case à évaluer.
237.          * @param column
```

```
238.          *            La valeur en abscisse de la case à évaluer.
239.          * @return Le booleen indiquant si la case décrite contient une adverse.
240.          */
241.         public boolean isOpponent(Board board, int row, int column) {
242.                 if (this.isBlack())
243.                         return board.isWhite(row, column);
244.                 else
245.                         return board.isBlack(row, column);
246.         }
247.
248.         /**
249.          * Retourne vrai si la case décrite par les paramètres "row" et "column
250.          * contient une pièce alliée.
251.          *
252.          * @param board
253.          *            La partie en cours.
254.          * @param row
255.          *            La valeur en ordonnée de la case à évaluer.
256.          * @param column
257.          *            La valeur en abscisse de la case à évaluer.
258.          * @return Le booleen indiquant si la case décrite contient une pièce
259.          *         alliée.
260.          */
261.         public boolean isSameColor(Board board, int row, int column) {
262.                 if (this.isBlack())
263.                         return board.isBlack(row, column);
264.                 else
265.                         return board.isWhite(row, column);
266.         }
267.
268.         /**
269.          * Retourne la liste des cases jouables en diagonales jusqu'à la rencontre
270.          * d'un obstacle.
271.          *
272.          * @param board
273.          *            La partie en cours.
274.          * @return La liste des cases jouables en diagonales.
275.          */
276.         public ArrayList<Square> possibleMovesDiagonale(Board board) {
277.                 ArrayList<Square> movesList = new ArrayList<Square>();
278.                 int i = 1;
279.                 // Mouvement en haut à droite
280.                 while ((this.getRow() + i < 9) && (this.getColumn() + i < 9)
281.                                 && board.isEmpty(this.getRow() + i, this.getColumn()
    + i)) {
282.                         movesList.add(new Square(this.getRow() + i, this.getColumn()
    + i));
283.                         i++;
```

```
284.                 }
285.                 if (this.isOpponent(board, this.getRow() + i, this.getColumn() + i))
286.                         movesList.add(new Square(this.getRow() + i, this.getColumn()
    + i));
287.                 i = 1;
288.                 // Mouvement en haut à gauche
289.                 while ((this.getRow() + i < 9) && (this.getColumn() - i > 0)
290.                                 && (board.isEmpty(this.getRow() + i, this.getColumn()
    - i))) {
291.                         movesList.add(new Square(this.getRow() + i, this.getColumn()
    - i));
292.                         i++;
293.                 }
294.                 if (this.isOpponent(board, this.getRow() + i, this.getColumn() - i))
295.                         movesList.add(new Square(this.getRow() + i, this.getColumn()
    - i));
296.                 i = 1;
297.                 // Mouvement en bas à gauche
298.                 while ((this.getRow() - i > 0) && (this.getColumn() - i > 0)
299.                                 && (board.isEmpty(this.getRow() - i, this.getColumn()
    - i))) {
300.                         movesList.add(new Square(this.getRow() - i, this.getColumn()
    - i));
301.                         i++;
302.                 }
303.                 if (this.isOpponent(board, this.getRow() - i, this.getColumn() - i))
304.                         movesList.add(new Square(this.getRow() - i, this.getColumn()
    - i));
305.                 i = 1;
306.                 // Mouvement en bas à droite
307.                 while ((this.getRow() - i > 0) && (this.getColumn() + i < 9)
308.                                 && (board.isEmpty(this.getRow() - i, this.getColumn()
    + i))) {
309.                         movesList.add(new Square(this.getRow() - i, this.getColumn()
    + i));
310.                         i++;
311.                 }
312.                 if (this.isOpponent(board, this.getRow() - i, this.getColumn() + i))
313.                         movesList.add(new Square(this.getRow() - i, this.getColumn()
    + i));
314.                 return movesList;
315.         }
316.
317.         /**
318.          * Retourne la liste des cases jouables en lignes droites jusqu'à la
319.          * rencontre d'un obstacle.
320.          *
321.          * @param board
```

```
322.          *            La partie en cours.
323.          * @return La liste des cases jouables en lignes droites.
324.          */
325.         public ArrayList<Square> possibleMovesDroit(Board board) {
326.                 ArrayList<Square> movesList = new ArrayList<Square>();
327.                 int i = 1;
328.                 // Mouvement en haut
329.                 while ((this.getRow() + i < 9)
330.                                 && (board.isEmpty(this.getRow() + i,
    this.getColumn()))) {
331.                         movesList.add(new Square(this.getRow() + i,
    this.getColumn()));
332.                         i++;
333.                 }
334.                 if (this.isOpponent(board, this.getRow() + i, this.getColumn()))
335.                         movesList.add(new Square(this.getRow() + i,
    this.getColumn()));
336.                 i = 1;
337.                 // Mouvement en bas
338.                 while ((this.getRow() - i > 0)
339.                                 && (board.isEmpty(this.getRow() - i,
    this.getColumn()))) {
340.                         movesList.add(new Square(this.getRow() - i,
    this.getColumn()));
341.                         i++;
342.                 }
343.                 if (this.isOpponent(board, this.getRow() - i, this.getColumn()))
344.                         movesList.add(new Square(this.getRow() - i,
    this.getColumn()));
345.                 i = 1;
346.                 // Mouvement à gauche
347.                 while ((this.getColumn() - i > 0)
348.                                 && (board.isEmpty(this.getRow(), this.getColumn() -
    i))) {
349.                         movesList.add(new Square(this.getRow(), this.getColumn() -
    i));
350.                         i++;
351.                 }
352.                 if (this.isOpponent(board, this.getRow(), this.getColumn() - i))
353.                         movesList.add(new Square(this.getRow(), this.getColumn() -
    i));
354.                 i = 1;
355.                 // Mouvement à droite
356.                 while ((this.getColumn() + i < 9)
357.                                 && (board.isEmpty(this.getRow(), this.getColumn() +
    i))) {
358.                         movesList.add(new Square(this.getRow(), this.getColumn() +
    i));
```

```
359.                        i++;
360.                }
361.                if (this.isOpponent(board, this.getRow(), this.getColumn() + i))
362.                        movesList.add(new Square(this.getRow(), this.getColumn() +
    i));
363.                return movesList;
364.        }
365.
366.        /**
367.         * Retourne vrai si la pièce est de soustype "Roi".
368.         *
369.         * @return un booleen indiquant si la pièce est de sous-type "Roi".
370.         */
371.        public boolean isKing() {
372.                if (this instanceof King)
373.                        return true;
374.                return false;
375.        }
376.
377.        /**
378.         * Affecte la valeur 0 aux coordonnées de la pièce, indiquant qu'elle est
379.         * mangé.
380.         */
381.        private void isBeingEaten() {
382.                this.row = 0;
383.                this.column = 0;
384.        }
385.
386.        /**
387.         * Déplace la pièce de la partie "board", depuis ses coordonnées initiales
388.         * (de ses variables de classes) vers les coordonnées "row" et "column".
389.         *
390.         * @param board
391.         *            La partie en cours.
392.         * @param row
393.         *            Valeur en ordonnées de la case vers laquelle déplacer la
394.         *            pièce.
395.         * @param column
396.         *            Valeur en abscisse de la case vers laquelle déplacer la pièce.
397.         * @throws OutOfBoardException
398.         * @throws NonPossibleMoveException
399.         * @throws EchecException
400.         */
401.        public void deplacerPiece(Board board, int row, int column)
402.                        throws OutOfBoardException, NonPossibleMoveException,
403.                        EchecException {
404.                Boolean priseEnPassant = false;
405.                if (this instanceof Pawn)
```

```
406.                         if (board.isEmpty(row, column))
407.                                 if (this.column != column)
408.                                         priseEnPassant = true;
409.                 Coup coup = new Coup();
410.                 coup.setIsPriseEnPassant(priseEnPassant);
411.                 coup.setMovedPiece(this);
412.                 coup.setNumeroCoup(board.getNumeroCoup());
413.                 Square caseDepart = new Square(this.row, this.column);
414.                 coup.setCaseDepart(caseDepart);
415.                 Square caseArrivee = new Square(row, column);
416.                 coup.setCaseArrivee(caseArrivee);
417.                 int oldRow = this.row;
418.                 int oldColumn = this.column;
419.                 boolean mange = false;
420.                 board.setSelectedCase("00");
421.                 Piece pieceMange = new Pawn("blue", 0, 0);
422.                 if ((row < 1) || (row > 8) || (column < 1) || (column > 8))
423.                         throw new OutOfBoardException("jeu hors des limites");
424.                 ArrayList<Square> listeCoups = this.possibleMoves(board);
425.                 boolean peutJouer = false;
426.                 Iterator<Square> it = listeCoups.iterator();
427.                 while (it.hasNext()) {
428.                         Square s = it.next();
429.                         if (s.isThisSquare(row, column))
430.                                 peutJouer = true;
431.                 }
432.                 if (!peutJouer)
433.                         throw new NonPossibleMoveException("coup non possible");
434.                 if (board.isEmpty(row, column) && !priseEnPassant) {
435.                         this.row = row;
436.                         this.column = column;
437.                 } else {
438.                         mange = true;
439.                         coup.setHasEaten(mange);
440.                         if (priseEnPassant)
441.                                 pieceMange = board.getPiece(oldRow, column);
442.                         else
443.                                 pieceMange = board.getPiece(row, column);
444.                         coup.setEatenPiece(pieceMange);
445.                         pieceMange.isBeingEaten();
446.                         board.estMangee(pieceMange, pieceMange.getColor());
447.                         this.row = row;
448.                         this.column = column;
449.                 }
450.                 if (this.color.equals("black")) {
451.                         if (board.isEchec("black", board.getBlackKing().getRow(),
    board
452.                                         .getBlackKing().getColumn())) {
```

```
453.                                    this.row = oldRow;
454.                                    this.column = oldColumn;
455.                                    if (mange) {
456.                                            if (priseEnPassant) {
457.                                                    pieceMange.setRow(oldRow);
458.                                                    pieceMange.setColumn(column);
459.                                            } else {
460.                                                    pieceMange.setRow(row);
461.                                                    pieceMange.setColumn(column);
462.                                            }
463.                                    }
464.                                    board.piecesNonMangees();
465.                                    throw new EchecException("Ce mouvement met votre roi
    en echec");
466.                            }
467.                    } else {
468.                            if (board.isEchec("white", board.getWhiteKing().getRow(),
    board
469.                                            .getWhiteKing().getColumn())) {
470.                                    this.row = oldRow;
471.                                    this.column = oldColumn;
472.                                    if (mange) {
473.                                            if (priseEnPassant) {
474.                                                    pieceMange.setRow(oldRow);
475.                                                    pieceMange.setColumn(column);
476.                                            } else {
477.                                                    pieceMange.setRow(row);
478.                                                    pieceMange.setColumn(column);
479.                                            }
480.                                    }
481.                                    board.piecesNonMangees();
482.                                    throw new EchecException("Ce mouvement met votre roi
    en echec");
483.                            }
484.                    }
485.                    if (this instanceof Pawn) {
486.                            if (oldRow - row == 2 || row - oldRow == 2) {
487.                                    this.setMangeableEnPrisePassant(board.getNumeroCoup()
    + 1);
488.                            } else if (row == 1 || row == 8) {
489.                                    coup.setIsPromotion(true);
490.                                    coup.setOldPiece(this);
491.                            }
492.                    }
493.                    board.ajouterCoup(coup);
494.                    if (coup.getIsPromotion())
495.                            board.setNumeroCoup(board.getNumeroCoup() - 1);
496.                    if ((this instanceof Rook) || (this instanceof King)) {
```

```
497.                        if (this instanceof King) {
498.                                if (!this.hasMovedOnce() && row == 1 && column == 7
499.                                                && !board.isEmpty(1, 8)) {
500.                                        Piece rook = board.getPiece(1, 8);
501.                                        coup.setEatenPiece(rook);
502.                                        rook.setColumn(6);
503.                                        coup.setIsPetitRoque(true);
504.                                }
505.                                if (!this.hasMovedOnce() && row == 8 && column == 7
506.                                                && !board.isEmpty(8, 8)) {
507.                                        Piece rook = board.getPiece(8, 8);
508.                                        coup.setEatenPiece(rook);
509.                                        rook.setColumn(6);
510.                                        coup.setIsPetitRoque(true);
511.                                }
512.                                if (!this.hasMovedOnce() && row == 1 && column == 3
513.                                                && !board.isEmpty(1, 1)) {
514.                                        Piece rook = board.getPiece(1, 1);
515.                                        coup.setEatenPiece(rook);
516.                                        rook.setColumn(4);
517.                                        coup.setIsGrandRoque(true);
518.                                }
519.                                if (!this.hasMovedOnce() && row == 8 && column == 3
520.                                                && !board.isEmpty(8, 1)) {
521.                                        Piece rook = board.getPiece(8, 1);
522.                                        coup.setEatenPiece(rook);
523.                                        rook.setColumn(4);
524.                                        coup.setIsGrandRoque(true);
525.                                }
526.                        }
527.                        this.hasMovedOnce = true;
528.                }
529.        }
530.
531.        /**
532.         * Retourne vrai si la case de la partie "b" décrite par les coordonnées
533.         * "column" et "row" est jouable par la pièce.
534.         *
535.         * @param row
536.         *            La valeur en ordonnées de la case à évaluer.
537.         * @param column
538.         *            La valuer en abscisse de la case à évaluer.
539.         * @param b
540.         *            La partie en cours.
541.         * @return Le booleen indiquant si la cas décrite est jouable par la pièce.
542.         */
543.        public boolean isPlayable(int row, int column, Board b) {
544.                ArrayList<Square> possibleMoves = possibleMoves(b);
```

```
545.                 Iterator<Square> it = possibleMoves.iterator();
546.                 while (it.hasNext()) {
547.                         Square s = it.next();
548.                         if (s.isThisSquare(row, column))
549.                                 return true;
550.                 }
551.                 return false;
552.         }
553.
554.         /**
555.          * Retourne vrai si la case de la partie "b" décrite par la chaine de
556.          * caractère "caseJeu" est jouable par la pièce.
557.          *
558.          * @param row
559.          *            Chaine de caractère de la case à évaluer, de "A" à "H" pour
560.          *            les abscisses et de 1 à 8 pour les ordonées.
561.          * @param b
562.          *            La partie en cours.
563.          * @return Le booleen indiquant si la cas décrite est jouable par la pièce.
564.          */
565.         public boolean isPlayable(String caseJeu, Board b) {
566.                 assert (!(caseJeu.charAt(0) < 'A') && !(caseJeu.charAt(0) > 'H')
567.                                 && !(caseJeu.charAt(1) < '1') && !(caseJeu.charAt(1)
    > '8'));
568.                 int column = caseJeu.charAt(0) - 'A' + '1' - 48;
569.                 int row = caseJeu.charAt(1) - 48;
570.                 return this.isPlayable(row, column, b);
571.         }
572.
573.         /**
574.          * Retourne la liste des coups possibles de la pièce (en prenant
575.          * en considération les coups qui peuvent mettre son roi en échec).
576.          *
577.          * @param board
578.          * @return liste des coups possibles de la pièce
579.          */
580.         public ArrayList<Square> possibleMovesSE(Board board) {
581.                 Board boardSimu;
582.                 ArrayList<Square> movesList = new ArrayList<>();
583.                 for (int i = 1; i <= 8; i++) {
584.                         for (int j = 1; j <= 8; j++) {
585.                                 if (this.isPlayable(i, j, board)) {
586.                                         try {
587.                                                 boardSimu = board.clone();
588.                                                 boardSimu.deplacerPiece(this.getRow()
    ,
589.                                                                 this.getColumn(), i,
    j);
```

```
590.                                                    if (this.color == "white") {
591.                                                            ArrayList<Square> echecList =
    boardSimu
592.                                                                                    .echec("white
    ");
593.                                                            int row =
    boardSimu.getWhiteKing().getRow();
594.                                                            int column =
    boardSimu.getWhiteKing().getColumn();
595.                                                            boolean isEchec = false;
596.                                                            Iterator<Square> it =
    echecList.iterator();
597.                                                            while (it.hasNext()) {
598.                                                                    Square s = it.next();
599.                                                                    if
    (s.isThisSquare(row, column))
600.                                                                                    isEchec =
    true;
601.                                                            }
602.                                                            if (!isEchec)
603.                                                                    movesList.add(new
    Square(i, j));
604.                                                    } else {
605.                                                            ArrayList<Square> echecList =
    boardSimu
606.                                                                                    .echec("black
    ");
607.                                                            int row =
    boardSimu.getBlackKing().getRow();
608.                                                            int column =
    boardSimu.getBlackKing().getColumn();
609.                                                            boolean isEchec = false;
610.                                                            Iterator<Square> it =
    echecList.iterator();
611.                                                            while (it.hasNext()) {
612.                                                                    Square s = it.next();
613.                                                                    if
    (s.isThisSquare(row, column))
614.                                                                                    isEchec =
    true;
615.                                                            }
616.                                                            if (!isEchec)
617.                                                                    movesList.add(new
    Square(i, j));
618.                                                    }
619.                                            } catch (OutOfBoardException e) {
620.                                                    e.printStackTrace();
621.
```

```
622.                                        } catch (NonPossibleMoveException e) {
623.                                                e.printStackTrace();
624.                                        }
625.                                }
626.
627.                        }
628.                }
629.                return movesList;
630.        }
631.
632. }
```

Pawn.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant le Pion.
 */
public class Pawn extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public Pawn(String color, int row, int column) {
		this.setNom("Pawn");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("p");
		else
			this.setShortcut("P");
		this.setMangeableEnPrisePassant(0);
	}

	/**
	 * Retourne la liste des coups possibles du pion (sans prendre en
	 * considération les coups qui peuvent mettre son roi en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		if (this.isDead())
			return movesList;
		if (this.isBlack()) {
```

```
46.                         if (this.getRow() == 7) {
47.                                 if (board.isEmpty(6, this.getColumn())) {
48.                                         movesList.add(new Square(6, this.getColumn()));
49.                                         if (board.isEmpty(5, this.getColumn()))
50.                                                 movesList.add(new Square(5,
    this.getColumn()));
51.                                 }
52.                                 if (this.getColumn() != 1
53.                                                 && board.isWhite(6, this.getColumn() -
    1))
54.                                         movesList.add(new Square(6, this.getColumn() -
    1));
55.                                 if (this.getColumn() != 8
56.                                                 && board.isWhite(6, this.getColumn() +
    1))
57.                                         movesList.add(new Square(6, this.getColumn() +
    1));
58.                         } else {
59.                                 if (this.getRow() != 1
60.                                                 && board.isEmpty(this.getRow() - 1,
    this.getColumn()))
61.                                         movesList.add(new Square(this.getRow() - 1,
    this
62.                                                         .getColumn()));
63.                                 if (this.getRow() != 1
64.                                                 && this.getColumn() != 1
65.                                                 && board.isWhite(this.getRow() - 1,
66.                                                                 this.getColumn() - 1))
67.                                         movesList.add(new Square(this.getRow() - 1,
    this
68.                                                         .getColumn() - 1));
69.                                 if (this.getRow() != 1
70.                                                 && this.getColumn() != 8
71.                                                 && board.isWhite(this.getRow() - 1,
72.                                                                 this.getColumn() + 1))
73.                                         movesList.add(new Square(this.getRow() - 1,
    this
74.                                                         .getColumn() + 1));
75.                         }
76.                 } else {
77.                         if (this.getRow() == 2) {
78.                                 if (board.isEmpty(3, this.getColumn())) {
79.                                         movesList.add(new Square(3, this.getColumn()));
80.                                         if (board.isEmpty(4, this.getColumn()))
81.                                                 movesList.add(new Square(4,
    this.getColumn()));
82.                                 }
83.                                 if (this.getColumn() != 1
```

```
84.                                         && board.isBlack(3, this.getColumn() -
    1))
85.                                     movesList.add(new Square(3, this.getColumn() -
    1));
86.                             if (this.getColumn() != 8
87.                                         && board.isBlack(3, this.getColumn() +
    1))
88.                                     movesList.add(new Square(3, this.getColumn() +
    1));
89.                         } else {
90.                                 if (this.getRow() != 8
91.                                         && board.isEmpty(this.getRow() + 1,
    this.getColumn()))
92.                                     movesList.add(new Square(this.getRow() + 1,
    this
93.                                                     .getColumn()));
94.                                 if (this.getRow() != 8
95.                                         && this.getColumn() != 1
96.                                         && board.isBlack(this.getRow() + 1,
97.                                                         this.getColumn() - 1))
98.                                     movesList.add(new Square(this.getRow() + 1,
    this
99.                                                     .getColumn() - 1));
100.                                if (this.getRow() != 8
101.                                        && this.getColumn() != 8
102.                                        && board.isBlack(this.getRow() + 1,
103.                                                        this.getColumn() +
    1))
104.                                    movesList.add(new Square(this.getRow() + 1,
    this
105.                                                    .getColumn() + 1));
106.                        }
107.                }
108.                // Prise en passant
109.                if ((this.getColor().equals("black") && this.getRow() == 4)
110.                                || (this.getColor().equals("white") && this.getRow()
    == 5)) {
111.
112.                        if (this.getColumn() != 1) {
113.                                if (!board.isEmpty(this.getRow(), this.getColumn() -
    1)) {
114.                                    Piece p = board.getPiece(this.getRow(),
115.                                                    this.getColumn() - 1);
116.                                    if (p instanceof Pawn) {
117.                                        if (p.getMangeableEnPrisePassant() ==
    board
118.                                                        .getNumeroCoup()) {
```

```
119.                                                     if
    (this.getColor().equals("black")
120.                                                             &&
    board.isEmpty(3, this.getColumn() - 1))
121.                                                         movesList.add(new
    Square(3,
122.                                                                 this.
    getColumn() - 1));
123.                                                     else if
    (this.getColor().equals("white")
124.                                                             &&
    board.isEmpty(6, this.getColumn() - 1))
125.                                                         movesList.add(new
    Square(6,
126.                                                                 this.
    getColumn() - 1));
127.                                                 }
128.                                             }
129.                                         }
130.                         }
131.                         if (this.getColumn() != 8) {
132.                                 if (!board.isEmpty(this.getRow(), this.getColumn() +
    1)) {
133.                                         Piece p = board.getPiece(this.getRow(),
134.                                                 this.getColumn() + 1);
135.                                         if (p instanceof Pawn) {
136.                                                 if (p.getMangeableEnPrisePassant() ==
    board
137.                                                                 .getNumeroCoup()) {
138.                                                     if
    (this.getColor().equals("black")
139.                                                             &&
    board.isEmpty(3, this.getColumn() + 1))
140.                                                         movesList.add(new
    Square(3,
141.                                                                 this.
    getColumn() + 1));
142.                                                     else if
    (this.getColor().equals("white")
143.                                                             &&
    board.isEmpty(6, this.getColumn() + 1))
144.                                                         movesList.add(new
    Square(6,
145.                                                                 this.
    getColumn() + 1));
146.                                                 }
147.                                         }
148.                                 }
```

```
149.                         }
150.                 }
151.                 return movesList;
152.         }
153.
154.         /**
155.          * Retourne un clone de la pièce.
156.          */
157.         protected Pawn clone() {
158.                 Pawn p = new Pawn(this.getColor(), this.getRow(), this.getColumn());
159.                 p.setMangeableEnPrisePassant(this.getMangeableEnPrisePassant());
160.                 return p;
161.         }
162.
163. }
```

King.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant le Roi.
 */
public class King extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public King(String color, int row, int column) {
		this.setNom("King");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("r");
		else
			this.setShortcut("R");
		this.moveOnce(false);
	}

	/**
	 * Retourne la liste des coups possibles du roi (sans prendre en
	 * considération les coups qui peuvent le mettre en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		// haut
		if (this.getRow() != 8)
```

```
45.                         if (!this.isSameColor(board, this.getRow() + 1,
    this.getColumn()))
46.                                 movesList.add(new Square(this.getRow() + 1,
    this.getColumn()));
47.                 // bas
48.                 if (this.getRow() != 1)
49.                         if (!this.isSameColor(board, this.getRow() - 1,
    this.getColumn()))
50.                                 movesList.add(new Square(this.getRow() - 1,
    this.getColumn()));
51.                 // gauche
52.                 if (this.getColumn() != 1)
53.                         if (!this.isSameColor(board, this.getRow(), this.getColumn() -
    1))
54.                                 movesList.add(new Square(this.getRow(),
    this.getColumn() - 1));
55.                 // droite
56.                 if (this.getColumn() != 8)
57.                         if (!this.isSameColor(board, this.getRow(), this.getColumn() +
    1))
58.                                 movesList.add(new Square(this.getRow(),
    this.getColumn() + 1));
59.                 // haut droite
60.                 if (this.getRow() != 8 || this.getColumn() != 8)
61.                         if (!this.isSameColor(board, this.getRow() + 1,
62.                                         this.getColumn() + 1))
63.                                 movesList.add(new Square(this.getRow() + 1,
64.                                                 this.getColumn() + 1));
65.                 // haut gauche
66.                 if (this.getRow() != 8 || this.getColumn() != 1)
67.                         if (!this.isSameColor(board, this.getRow() + 1,
68.                                         this.getColumn() - 1))
69.                                 movesList.add(new Square(this.getRow() + 1,
70.                                                 this.getColumn() - 1));
71.                 // bas droite
72.                 if (this.getRow() != 1 || this.getColumn() != 8)
73.                         if (!this.isSameColor(board, this.getRow() - 1,
74.                                         this.getColumn() + 1))
75.                                 movesList.add(new Square(this.getRow() - 1,
76.                                                 this.getColumn() + 1));
77.                 // bas gauche
78.                 if (this.getRow() != 1 || this.getColumn() != 1)
79.                         if (!this.isSameColor(board, this.getRow() - 1,
80.                                         this.getColumn() - 1))
81.                                 movesList.add(new Square(this.getRow() - 1,
82.                                                 this.getColumn() - 1));
83.                 // petit roque
84.                 if (!this.hasMovedOnce()) {
```

```
85.                         if (this.canShortCastle(board, 1))
86.                                 movesList.add(new Square(1, 7));
87.                         if (this.canShortCastle(board, 8))
88.                                 movesList.add(new Square(8, 7));
89.                 }
90.                 // grand roque
91.                 if (!this.hasMovedOnce()) {
92.                         if (this.canLongCastle(board, 1))
93.                                 movesList.add(new Square(1, 3));
94.                         if (this.canLongCastle(board, 8))
95.                                 movesList.add(new Square(8, 3));
96.                 }
97.                 return movesList;
98.         }
99.
100.          /**
101.           * Retourne le booleen remplissant les conditions nécessaires pour effectuer
102.           * le petit roque.
103.           *
104.           * @param board
105.           *            La partie en cours.
106.           * @param row
107.           *            Prend la valeur 1 ou 8.
108.           * @return Le booleen remplissant les conditions nécessaires pour effectuer
109.           *         le petit roque.
110.           */
111.          private boolean canShortCastle(Board board, int row) {
112.                  assert (row == 1 || row == 8);
113.                  return this.isSameColor(board, row, 8)
114.                                  && !board.getPiece(row, 8).hasMovedOnce()
115.                                  && board.isEmpty(row, 6) && board.isEmpty(row, 7)
116.                                  && !board.isEchec(this.getColor(), row, 5)
117.                                  && !board.isEchec(this.getColor(), row, 6)
118.                                  && !board.isEchec(this.getColor(), row, 7);
119.          }
120.
121.          /**
122.           * Retourne le booleen remplissant les conditions nécessaires pour effectuer
123.           * le grand roque.
124.           *
125.           * @param board
126.           *            La partie en cours.
127.           * @param row
128.           *            Prend la valeur 1 ou 8.
129.           * @return Le booleen remplissant les conditions nécessaires pour effectuer
130.           *         le grand roque.
131.           */
132.          private boolean canLongCastle(Board board, int row) {
```

```
133.                 assert (row == 1 || row == 8);
134.                 return this.isSameColor(board, row, 1)
135.                                 && !board.getPiece(row, 1).hasMovedOnce()
136.                                 && board.isEmpty(row, 4) && board.isEmpty(row, 3)
137.                                 && board.isEmpty(row, 2)
138.                                 && !board.isEchec(this.getColor(), row, 5)
139.                                 && !board.isEchec(this.getColor(), row, 4)
140.                                 && !board.isEchec(this.getColor(), row, 3);
141.         }
142.
143.         /**
144.          * Retourne un clone de la pièce.
145.          */
146.         protected King clone() {
147.                 return new King(this.getColor(), this.getRow(), this.getColumn());
148.         }
149.
150. }
```

Queen.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant la Dame.
 */
public class Queen extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public Queen(String color, int row, int column) {
		this.setNom("Queen");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("d");
		else
			this.setShortcut("D");
	}

	/**
	 * Retourne la liste des coups possibles de la Dame (sans prendre en
	 * considération les coups qui peuvent mettre son roi en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		if (this.isDead())
			return movesList;
		ArrayList<Square> movesList1 = this.possibleMovesDiagonale(board);
		ArrayList<Square> movesList2 = this.possibleMovesDroit(board);
```

```
46.                 movesList1.addAll(movesList2);
47.                 return movesList1;
48.         }
49.
50.         /**
51.          * Retourne un clone de la pièce.
52.          */
53.         protected Queen clone() {
54.                 return new Queen(this.getColor(), this.getRow(), this.getColumn());
55.         }
56.
57. }
```

Rook.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant la Tour.
 */
public class Rook extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public Rook(String color, int row, int column) {
		this.setNom("Rook");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("t");
		else
			this.setShortcut("T");
		this.moveOnce(false);
	}

	/**
	 * Retourne la liste des coups possibles de la Tour (sans prendre en
	 * considération les coups qui peuvent mettre son roi en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		if (this.isDead()) {
			return movesList;
		}
```

```
46.                 return this.possibleMovesDroit(board);
47.         }
48.
49.         /**
50.          * Retourne un clone de la pièce.
51.          */
52.         protected Rook clone() {
53.                 return new Rook(this.getColor(), this.getRow(), this.getColumn());
54.         }
55.
56. }
```

Knght.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant le Cavalier.
 */
public class Knight extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public Knight(String color, int row, int column) {
		this.setNom("Knight");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("c");
		else
			this.setShortcut("C");
	}

	/**
	 * Retourne la liste des coups possibles du cavalier (sans prendre en
	 * considération les coups qui peuvent mettre son roi en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		if (this.isDead())
			return movesList;
		// haut haut droit
		if (this.getRow() < 7 || this.getColumn() < 8)
```

```
46.                     if (!this.isSameColor(board, this.getRow() + 2,
47.                                     this.getColumn() + 1))
48.                             movesList.add(new Square(this.getRow() + 2,
49.                                             this.getColumn() + 1));
50.             // haut haut gauche
51.             if (this.getRow() < 7 || this.getColumn() > 1)
52.                     if (!this.isSameColor(board, this.getRow() + 2,
53.                                     this.getColumn() - 1))
54.                             movesList.add(new Square(this.getRow() + 2,
55.                                             this.getColumn() - 1));
56.             // bas bas gauche
57.             if (this.getRow() > 2 || this.getColumn() > 1)
58.                     if (!this.isSameColor(board, this.getRow() - 2,
59.                                     this.getColumn() - 1))
60.                             movesList.add(new Square(this.getRow() - 2,
61.                                             this.getColumn() - 1));
62.             // bas bas droit
63.             if (this.getRow() > 2 || this.getColumn() < 8)
64.                     if (!this.isSameColor(board, this.getRow() - 2,
65.                                     this.getColumn() + 1))
66.                             movesList.add(new Square(this.getRow() - 2,
67.                                             this.getColumn() + 1));
68.             // droit droit haut
69.             if (this.getRow() < 8 || this.getColumn() < 7)
70.                     if (!this.isSameColor(board, this.getRow() + 1,
71.                                     this.getColumn() + 2))
72.                             movesList.add(new Square(this.getRow() + 1,
73.                                             this.getColumn() + 2));
74.             // droit droit bas
75.             if (this.getRow() > 1 || this.getColumn() < 7)
76.                     if (!this.isSameColor(board, this.getRow() - 1,
77.                                     this.getColumn() + 2))
78.                             movesList.add(new Square(this.getRow() - 1,
79.                                             this.getColumn() + 2));
80.             // gauche gauche haut
81.             if (this.getRow() < 8 || this.getColumn() > 2)
82.                     if (!this.isSameColor(board, this.getRow() + 1,
83.                                     this.getColumn() - 2))
84.                             movesList.add(new Square(this.getRow() + 1,
85.                                             this.getColumn() - 2));
86.             // gauche gauche bas
87.             if (this.getRow() > 1 || this.getColumn() > 2)
88.                     if (!this.isSameColor(board, this.getRow() - 1,
89.                                     this.getColumn() - 2))
90.                             movesList.add(new Square(this.getRow() - 1,
91.                                             this.getColumn() - 2));
92.             return movesList;
93.     }
```

```
94.
95.         /**
96.          * Retourne un clone de la pièce.
97.          */
98.         protected Knight clone() {
99.                 return new Knight(this.getColor(), this.getRow(), this.getColumn());
100.          }
101.
102.  }
```

Bishop.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Instance de la pièce représentant le Fou.
 */
public class Bishop extends Piece implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Constructeur de la pièce. Il s'initialise avec la couleur de la pièce, et
	 * doit être positionner aux coordonnées "row" et "column".
	 *
	 * @param color
	 *            Couleur de la pièce à instancier "white" ou "black".
	 * @param row
	 *            Coordonnée en abscisse.
	 * @param column
	 *            Coordonnée en ordonnée.
	 */
	public Bishop(String color, int row, int column) {
		this.setNom("Bishop");
		this.setColor(color);
		this.setRow(row);
		this.setColumn(column);
		if (color.equals("black"))
			this.setShortcut("f");
		else
			this.setShortcut("F");
	}

	/**
	 * Retourne la liste des coups possibles du fou (sans prendre en
	 * considération les coups qui peuvent mettre son roi en échec).
	 */
	public ArrayList<Square> possibleMoves(Board board) {
		ArrayList<Square> movesList = new ArrayList<Square>();
		if (this.isDead())
			return movesList;
		return this.possibleMovesDiagonale(board);
	}
```

```
46.
47.         /**
48.          * Retourne un clone de la pièce.
49.          */
50.         protected Bishop clone() {
51.                 return new Bishop(this.getColor(), this.getRow(), this.getColumn());
52.         }
53.
54. }
```

Square.java

```java
import java.io.Serializable;

/**
 * Implementation d'une case (utile pour faire une fonction qui renvoie les 2 coordonnées d'une case)
 *
 */
public class Square implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * ordonnée de la case
	 */
	private int row;
	/**
	 * abscisse de la case
	 */
	private int column;

	/**
	 * Getter de l'abscisse
	 *
	 * @return
	 */
	public int getRow() {
		return row;
	}

	/**
	 * Getter de l'ordonnée
	 *
	 * @return
	 */
	public int getColumn() {
		return column;
	}

	/**
	 * Constructeur de square
	 *
	 * @param row
```

```
45.          * @param column
46.          */
47.         public Square(int row, int column) {
48.                 this.row = row;
49.                 this.column = column;
50.         }
51.
52.         /**
53.          * Verification que la case est dans le tableau
54.          *
55.          * @return true si la case appartient au tableau
56.          */
57.         public Boolean isRealSquare() {
58.                 return (this.row > 0 && this.row < 9 && this.column > 0 && this.column
    < 9);
59.         }
60.
61.         /**
62.          * Verifie si la case instanciée a les coordonnées passées en paramètre
63.          *
64.          * @param row
65.          * @param column
66.          * @return
67.          */
68.         public boolean isThisSquare(int row, int column) {
69.                 return (column == this.getColumn() && row == this.row);
70.         }
71.
72.         /**
73.          * Nom de la case instanciée
74.          *
75.          * @return chaine de caractères correspondant au nom de la case
76.          */
77.         public String getNomCase() {
78.                 String nom = new String();
79.                 if (this.getColumn() == 1)
80.                         nom += "A";
81.                 else if (this.getColumn() == 2)
82.                         nom += "B";
83.                 else if (this.getColumn() == 3)
84.                         nom += "C";
85.                 else if (this.getColumn() == 4)
86.                         nom += "D";
87.                 else if (this.getColumn() == 5)
88.                         nom += "E";
89.                 else if (this.getColumn() == 6)
90.                         nom += "F";
91.                 else if (this.getColumn() == 7)
```

```
92.                         nom += "G";
93.                 else if (this.getColumn() == 8)
94.                         nom += "H";
95.                 nom += Integer.toString(this.getRow());
96.                 return nom;
97.         }
98.
99. }
```

TestFonctions.java

```java
//import java.util.ArrayList;

/**
 * Classe contenant un main qui teste nos fonctions de jeu en mode console
 *
 */
public class TestFonctions {

	/**
	 * Affichage du plateau de jeu en console
	 *
	 * @param b
	 * @return
	 */
	public static String affichageBoard(Board b) {
		String s = "";
		Piece p;
		for (int i = 8; i > 0; --i) {
			s += "|";
			for (int j = 1; j < 9; ++j) {
				if (b.isEmpty(i, j))
					s += "  ";
				else {
					p = b.getPiece(i, j);
					if (p.isWhite()) {
						s += "W";
					} else {
						s += "B";
					}
					if (p instanceof Pawn)
						s += "P";
					else if (p instanceof Rook)
						s += "T";
					else if (p instanceof Knight)
						s += "C";
					else if (p instanceof Bishop)
						s += "F";
					else if (p instanceof Queen)
						s += "Q";
					else if (p instanceof King)
						s += "K";
				}
				s += "|";
			}
			s += "\n";
```

```
46.                 }
47.                 return s;
48.         }
49.
50.         /**
51.          * main testant des fonctions en mode console
52.          *
53.          * @param argv
54.          * @throws NonPossibleMoveException
55.          * @throws OutOfBoardException
56.          */
57.         public static void main(String argv[]) throws OutOfBoardException,
58.                         NonPossibleMoveException {
59.
60.                 Board b = new Board();
61.                 // HTMLGen html = new HTMLGen(b);
62.
63.                 // System.out.println(html.getPage());
64.
65.                 System.out.println(affichageBoard(b));
66.                 try {
67.                         b.deplacerPiece("E2", "E4");
68.                 } catch (Exception e) {
69.                         e.printStackTrace();
70.                 }
71.                 System.out.println(affichageBoard(b));
72.                 try {
73.                         b.deplacerPiece("A7", "A5");
74.                 } catch (Exception e) {
75.                         e.printStackTrace();
76.                 }
77.                 System.out.println(affichageBoard(b));
78.                 try {
79.                         b.deplacerPiece("F1", "B5");
80.                 } catch (Exception e) {
81.                         e.printStackTrace();
82.                 }
83.                 System.out.println(affichageBoard(b));
84.                 try {
85.                         b.deplacerPiece("D7", "D5");
86.                 } catch (Exception e) {
87.                         e.printStackTrace();
88.                 }
89.                 System.out.println(affichageBoard(b));
90.                 try {
91.                         b.deplacerPiece("C7", "C6");
92.                 } catch (Exception e) {
93.                         e.printStackTrace();
```

```
94.                 }
95.                 System.out.println(affichageBoard(b));
96.
97.         }
98.
99. }
```

Web.java

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.text.ParseException;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Date;
6.  import java.util.Locale;
7.
8.  /**
9.   * Classe contenant le main servant à lancer le serveur
10.  *
11.  */
12. public class Web {
13.
14.         /**
15.          * Lis un fichier du package
16.          *
17.          * @param fichier
18.          * @return le contenu du fichier
19.          * @throws IOException
20.          */
21.         static String lireFichier(String fichier) throws IOException {
22.                 BufferedReader lecteurAvecBuffer = null;
23.                 String ligne;
24.                 String fileContent = "";
25.                 try {
26.                         lecteurAvecBuffer = new BufferedReader(new
    FileReader(fichier));
27.                 } catch (FileNotFoundException exc) {
28.                         System.out.println("Erreur d'ouverture");
29.                         return lireFichier("pageNotFound.html");
30.                 }
31.                 while ((ligne = lecteurAvecBuffer.readLine()) != null)
32.                         fileContent += ligne + "\n";
33.                 lecteurAvecBuffer.close();
34.                 // System.out.println("::: Fichier lu :::");
35.                 return fileContent;
36.         }
37.
38.         /**
39.          * permet de remplir automatiquement le content-type de l'en tête
40.          *
41.          * @param objet
42.          *            (nom du fichier à lire)
43.          * @return le type du fichier
44.          */
```

```
45.         static String getContentType(String objet) {
46.                 if (objet.contains(".html"))
47.                         return "text/html";
48.                 if (objet.contains(".css"))
49.                         return "text/css";
50.                 if (objet.contains(".svg"))
51.                         return "image/svg+xml";
52.                 if (objet.contains(".ico"))
53.                         return "image/ico";
54.                 if (objet.contains(".jpg"))
55.                         return "image";
56.                 return "text/html";
57.         }
58.
59.         /**
60.          * runner du serveur
61.          *
62.          * @param args (ne s'en sert pas)
63.          */
64.         public static void main(String[] args) {
65.                 ServerSocket socketserver;
66.                 try {
67.                         socketserver = new ServerSocket(7777);
68.                         Socket socket;
69.                         InputStream istream;
70.                         OutputStream ostream;
71.                         // trouv sur
72.                         // http://stackoverflow.com/questions/1930158/how-to-parse-
    date-from-http-last-modified-header
73.                         SimpleDateFormat format = new SimpleDateFormat(
74.                                         "EEE MMM dd HH:mm:ss z yyyy", Locale.ENGLISH);
75.                         SimpleDateFormat format2 = new SimpleDateFormat(
76.                                         "EEE, dd MMM yyyy HH:mm:ss z", Locale.ENGLISH);
77.                         Board b = new Board();
78.                         SavedGamesList save = new SavedGamesList();
79.                         ObjectInputStream ois1;
80.                         try {
81.                                 ois1 = new ObjectInputStream(new BufferedInputStream(
82.                                                 new FileInputStream(new
    File("ListeSauvegardes.txt"))));
83.                                 save = (SavedGamesList) ois1.readObject();
84.                                 ois1.close();
85.                         } catch (IOException e) {
86.                                 e.printStackTrace();
87.                         } catch (ClassNotFoundException e2) {
88.                                 e2.printStackTrace();
89.                         }
90.                         while (true) {
```

```
91.                                 String partieASupprimer = "";
92.                                 Boolean notModified = false;
93.                                 socket = socketserver.accept();
94.                                 istream = socket.getInputStream();
95.                                 ostream = socket.getOutputStream();
96.                                 byte[] buffer = new byte[1024];
97.                                 int nb = -1;
98.                                 String input = "";
99.                                 // System.out.println("====================== Nouvelle
    requete ====================\n");
100.                                do {
101.                                        nb = istream.read(buffer);
102.                                        String tmp = new String(buffer, "UTF-8");
103.                                        input += tmp;
104.                                } while (nb == 1024);
105.                                // System.out.println(input);
106.                                if (input.startsWith("GET ")) {
107.                                        String objet = "";
108.                                        int j = 5;
109.                                        while (input.charAt(j) != ' ') {
110.                                                objet += input.charAt(j);
111.                                                ++j;
112.                                        }
113.                                        String fichier = "";
114.                                        String parametres = "";
115.                                        boolean passe = false;
116.                                        for (int i = 0; i < objet.length(); ++i)
117.                                                if (passe)
118.                                                        parametres +=
    objet.charAt(i);
119.                                                else if (objet.charAt(i) == '?')
120.                                                        passe = true;
121.                                                else
122.                                                        fichier += objet.charAt(i);
123.                                        // System.out.println("::: Lecture de " +
    fichier +
124.                                        // " avec comme parametres : " + parametres
    +" :::");
125.                                        try {
126.                                                String content = "";
127.                                                if (fichier.length() == 0
128.                                                                ||
    fichier.equals("index.html")) {
129.                                                        // avec parametres
130.                                                        if (parametres.length() > 0)
    {
131.                                                                // trouvé sur
```

```
132.                                                                    //
    http://www.tomsguide.fr/forum/id-553824/ecrire-lire-fichier-txt-java.html
133.                                                                    if
    (parametres.startsWith("Save")) {
134.                                                                            ObjectOutputS
    tream oos;
135.                                                                            String file =
    parametres.substring(5);
136.                                                                            if
    (file.equals("ListeSauvegardes")
137.
      || save.isSave(file)) {
138.                                                                                    // ne
    fait rien si le nom est déjà pris
139.                                                                            } else {
140.                                                                                    try {
141.
      oos = new ObjectOutputStream(
142.
                    new BufferedOutputStream(
143.
                                    new FileOutputStream(
144.
                                                    new File(
145.
                                                                    file
146.
                                                                                    +
    ".txt"))));
147.
      oos.writeObject(b);
148.
      oos.close();
149.
      save.newSave(file);
150.
      ObjectOutputStream oos2;
151.
      oos2 = new ObjectOutputStream(
152.
                    new BufferedOutputStream(
153.
                                    new FileOutputStream(
154.
                                                    new File(
155.
                                                                    "ListeSauvegardes.txt
    ")));
```

```
156.
      oos2.writeObject(save);
157.
      oos2.close();
158.                                                                    }
    catch (java.io.IOException e) {
159.
      e.printStackTrace();
160.                                                                    }
161.                                                                }
162.                                                            } else if
    (parametres.startsWith("Load")) {
163.                                                                ObjectInputSt
    ream ois;
164.                                                                String file =
    parametres.substring(5);
165.                                                                if
    (save.isSave(file)) {
166.                                                                    try {
167.
      ois = new ObjectInputStream(
168.
                      new BufferedInputStream(
169.
                                          new FileInputStream(
170.
                                                      new File(
171.
                                                              file
172.
                                                                      +
    ".txt"))));
173.
      b = (Board) ois.readObject();
174.
      ois.close();
175.                                                                    }
    catch (IOException e) {
176.
      e.printStackTrace();
177.                                                                    }
    catch (ClassNotFoundException e2) {
178.
      e2.printStackTrace();
179.                                                                    }
180.                                                                } else {
181.                                                                    // ne
    fait rien si la sauvegarde est inexistante
```

```
182.                                                                    }
183.                                                                } else if
    (parametres.startsWith("Suppr")) {
184.                                                                    String partie
    = parametres.substring(6);
185.                                                                    save.getSaved
    Games().remove(partie);
186.                                                                    File
    partieFile = new File(partie + ".txt");
187.                                                                    partieFile.de
    lete();
188.                                                                    ObjectOutputS
    tream oos2;
189.                                                                    oos2 = new
    ObjectOutputStream(
190.
      new BufferedOutputStream(
191.
                    new FileOutputStream(
192.
                                    new File(
193.
                                                        "ListeSauvegardes.txt"))));
194.                                                                    oos2.writeObj
    ect(save);
195.                                                                    oos2.close();
196.                                                                } else if
    (parametres.startsWith("delete")){
197.                                                                    partieASuppri
    mer = parametres.substring(7);
198.
199.                                                                } else if
    (parametres.equals("Rook")
200.                                                                            ||
    parametres.equals("Knight")
201.                                                                            ||
    parametres.equals("Bishop")
202.                                                                            ||
    parametres.equals("Queen")) {
203.                                                                    Boolean
    promotion = false;
204.                                                                    if
    (b.getNumeroCoupMax() > b
205.
      .getNumeroCoup()) {
206.                                                                            Coup
    c = b.getListeCoups().get(
```

```
207.
            b.getNumeroCoup());
208.                                                                    if
    (c.getIsPromotion()
209.
            && c.getMovedPiece() instanceof Pawn) {
210.
      promotion = true;
211.                                                                    }
212.                                                                }
213.                                                                if
    (promotion) {
214.                                                                    b.set
    Promotion(parametres);
215.                                                                } else {
216.                                                                    // Ne
    fait rien si une piece est en parametre et que le coup n'est pas une promotion
217.                                                                }
218.                                                            } else if
    (parametres.equals("NewGame"))
219.                                                                b = new
    Board();
220.                                                            else if
    (parametres.equals("Undo"))
221.                                                                if
    (b.getNumeroCoup() > 1) {
222.                                                                    b.ann
    ulerCoup();
223.                                                                } else {
224.                                                                    //
    annulation Impossible car aucun coup n'a été joué
225.                                                                }
226.                                                            else if
    (parametres.equals("Redo"))
227.                                                                if
    (b.getNumeroCoup() < b
228.
      .getNumeroCoupMax()) {
229.                                                                    try {
230.
      b.retablirCoup();
231.
      b.nextPlayer();
232.                                                                    }
    catch (Exception e) {
233.
      e.printStackTrace();
234.                                                                }
```

```
235.                                                                    } else {
236.                                                                            //
    Redo Impossible car aucun coup n'a été annulé
237.                                                                    }
238.                                                            // case ou l'on veut
    aller
239.                                                            else if
    (parametres.startsWith("to")) {
240.                                                                    if
    ((parametres.charAt(2) < 'A')
241.
      || (parametres.charAt(2) > 'H')
242.
      || (parametres.charAt(3) < '1')
243.
      || (parametres.charAt(3) > '8')) {
244.                                                                            // Ne
    fait rien si les parametres sont hors du tableau
245.                                                                    } else {
246.                                                                            Strin
    g nomCase = "";
247.                                                                            nomCa
    se += parametres.charAt(2);
248.                                                                            nomCa
    se += parametres.charAt(3);
249.                                                                            try {
250.
      if (b.getPiece(b.getSelectedCase())
251.
                      .isPlayable(nomCase, b)) {
252.
              try {
253.
                      b.deplacerPiece(
254.
                                      b.getSelectedCase(),
255.
                                      nomCase);
256.
                      b.setSelectedCase("00");
257.
              } catch (NonPossibleMoveException e) {
258.
                      e.printStackTrace();
259.
                      b.nextPlayer();
260.
              }
```

```
261.
            b.nextPlayer();
262.
     } else {
263.
            // ne fait rien si la case n'est pas jouable
264.
     }
265.                                                                    }
   catch (Exception e) {
266.
     e.printStackTrace();
267.                                                                    }
268.                                                                }
269.                                                            }
270.                                                            // pion que l'on
   selectionne
271.                                                            else {
272.                                                                if
   ((parametres.charAt(0) < 'A')
273.
     || (parametres.charAt(0) > 'H')
274.
     || (parametres.charAt(1) < '1')
275.
     || (parametres.charAt(1) > '8')) {
276.                                                                    // Ne
   fait rien si les parametres sont hors du tableau
277.                                                                } else {
278.                                                                    Strin
   g nomCase = "";
279.                                                                    nomCa
   se += parametres.charAt(0);
280.                                                                    nomCa
   se += parametres.charAt(1);
281.                                                                    if
   (b.isEmpty(nomCase)) {
282.
     // Ne fait rien si on selectionne une case vide
283.                                                                    }
   else {
284.
     try {
285.
            if (b.getPiece(nomCase)
286.
                        .getColor()
```

```
287.
                            .equals(b
288.
                                                .getCurrentPlayer()))
289.
                    b.setSelectedCase(nomCase);
290.
            else {
291.
                    // Ne fait rien si la piece de la case en parametre n'est pas de la
    bonne couleur
292.
            }
293.
     } catch (Exception e) {
294.
            e.printStackTrace();
295.
     }
296.                                                                                    }
297.                                                                                }
298.                                                                            }
299.                                                                        }
300.                                                                        // géneration de la page html
301.                                                                        try {
302.                                                                                HTMLGen html = new
    HTMLGen(b,
303.                                                                                                save.
    getSavedGames(),
304.                                                                                                parti
    eASupprimer);
305.                                                                                content +=
    html.getPage();
306.                                                                        } catch (Exception e) {
307.                                                                                e.printStackTrace();
308.                                                                        }
309.                                                                    } else {
310.                                                                        // gestion code 304
311.                                                                        if (input.contains("If-
    Modified-Since")) {
312.                                                                                String input2 = new
    String(input);
313.                                                                                while
    (!input2.startsWith("If-Modified-Since")) {
314.                                                                                        input2 =
    input2.substring(1);
315.                                                                                }
```

```
316.                                                    input2 =
    input2.substring(19, 48);
317.                                                    {
318.                                                            if
    (!input2.contains(",")) {
319.                                                                    input
    2 = input2.substring(0,
320.
                input2.length() - 1);
321.                                                            }
322.                                                    }
323.                                                    //
    System.out.println("\n-----------------\n" +
324.                                                    // input2 + "\n------
    ------------\n");
325.                                                    try {
326.                                                            Date d = new
    Date();
327.                                                            if
    (input2.contains(",")) {
328.                                                                    d =
    format2.parse(input2);
329.                                                            } else {
330.                                                                    d =
    format.parse(input2);
331.                                                            }
332.                                                            Date fileDate
    = new Date();
333.                                                            if
    (getContentType(fichier).contains(
334.
      "image")) {
335.                                                                    Strin
    g dateImage = "Tue Nov 19 20:35:08 CET 2013";
336.                                                                    fileD
    ate = format.parse(dateImage);
337.                                                            }
338.                                                            if
    (fileDate.before(d)) {
339.                                                                    notMo
    dified = true;
340.                                                            }
341.                                                    } catch
    (ParseException e) {
342.                                                            e.printStackT
    race();
343.                                                    }
344.                                            }
```

```
345.                                                    if (!notModified) {
346.                                                            content =
    lireFichier(fichier);
347.                                                    }
348.                                            }
349.                                            String header = "";
350.                                            String output = "";
351.                                            Date date = new Date();
352.                                            if (notModified) {
353.                                                    header = "HTTP/1.1 304 Not
    Modified"
354.                                                                    + "\nServer:
    WebChess localhost:7777"
355.                                                                    + "\nDate: "
    + format2.format(date)
356.                                                                    + "\n\n";
357.                                                    output = header;
358.                                            } else {
359.                                                    // envoi de la page html au
    navigateur
360.                                                    header = "HTTP/1.1 200 OK"
361.                                                                    + "\nServer:
    WebChess localhost:7777"
362.                                                                    // Différente
    longueur de string et byte avec les accents en UTF-8
363.                                                                    + "\nContent-
    Length: " + content.getBytes("UTF-8").length
364.                                                                    +
    "\nConnection: close"
365.                                                                    + "\nContent-
    Type: "
366.                                                                    +
    getContentType(fichier)
367.                                                                    + ";
    charset=UTF-8";
368.                                                    header += "\nLast-Modified:
    ";
369.                                                    if
    (getContentType(fichier).contains("image")) {
370.                                                            header += "Tue, 19
    Nov 2013 20:35:08 GMT";
371.                                                    } else {
372.                                                            header += date;
373.                                                    }
374.                                                    header += "\n\n";
375.
376.                                                    output = header + content;
377.                                            }
```

```
378.                                                    // System.out.println(header);
379.                                                    byte[] temp = output.getBytes("UTF-
    8");
380.                                                    ostream.write(temp);
381.                                                    ostream.flush();
382.                                            } catch (IOException e) {
383.                                                    //  Auto-generated catch block
384.                                                    e.printStackTrace();
385.                                            }
386.                                    }
387.                                    // socket.close();
388.                                    // socketserver.close();
389.                            }
390.                            // socketserver.close();
391.                            // System.out.println("::: Deconnexion");
392.
393.                    } catch (IOException e1) {
394.                            //  Auto-generated catch block
395.                            e1.printStackTrace();
396.                    }
397.            }
398.
399.    }
```

HTMLGen.java

```
1.  import java.util.ArrayList;
2.  import java.util.Iterator;
3.
4.  /**
5.   * Classe servant à générer le code HTML de la page de jeu
6.   *
7.   */
8.  public class HTMLGen {
9.
10.         /**
11.          * Liste des coups possibles
12.          */
13.         private ArrayList<Square> possibleMoves;
14.
15.         /**
16.          * Header statique
17.          */
18.         private static String head = "<html>\n"
19.                         + "<head>\n"
20.                         + "         <meta http-equiv=\"content-type\"
    content=\"text/html;charset=utf-8\" />\n"
21.                         + "         <link rel=\"stylesheet\" href=\"stylesheet.css\"
    type=\"text/css\" />\n"
22.                         + "         <link rel=\"shortcut icon\" type=\"image/x-icon\"
    href=\"favicon.ico\">\n"
23.                         + "         <title>WebChess Kasparov</title>\n" + "</head>\n" +
    "<body>\n";
24.
25.         /**
26.          * Chaine de caractère contenant le corps de la page
27.          */
28.         private String body;
29.
30.         /**
31.          * Chaine de caractère contenant la banière gauche de la page
32.          */
33.         private String left = "";
34.
35.         /**
36.          * partie du HTML contenant les boutons UNDO et REDO
37.          */
38.         private String boutonsUndoRedo = "";
39.
40.         /**
41.          * partie du HTML contenant la liste des coups
```

```
42.          */
43.         private String listeCoups = "";
44.
45.         /**
46.          * partie du HTML contenant le cimetière
47.          */
48.         private String eatenPieces = "";
49.
50.         /**
51.          * Partie du HTML contenant les messages d'échec et pat et le choix des pièces
    pour la promotion
52.          */
53.         private String legende = "";
54.
55.         /**
56.          * Fin de la page HTML (1ere partie)
57.          */
58.         private static String bottom1 = "<div id=\"basdepage\">Cette page vous a été
    presentée avec honneur par Andrea et Quentin.<BR>Nous vous souhaitons bonne chance dans
    votre partie.</div>\n</body>\n</html>";
59.
60.         /**
61.          * Constructeur du HTMLGen permettant de remplir les différentes chaines de
    caractère en fonction de l'état du plateau et des sauvegardes
62.          * @param b le plateau de jeu
63.          * @param saves la liste des sauvegardes
64.          * @throws OutOfBoardException
65.          * @throws NonPossibleMoveException
66.          */
67.         public HTMLGen(Board b, ArrayList<String> saves, String partieASupprimer)
68.                         throws OutOfBoardException, NonPossibleMoveException {
69.                 Boolean promotion = false;
70.                 if (b.getNumeroCoupMax() > b.getNumeroCoup()) {
71.                         Coup c = b.getListeCoups().get(b.getNumeroCoup());
72.                         if (c.getIsPromotion() && c.getMovedPiece() instanceof Pawn)
73.                                 promotion = true;
74.                 }
75.                 Boolean finPartie = this.remplirOptions(b, promotion,
    partieASupprimer);
76.                 this.getLeft(b, saves);
77.                 Boolean nonCliquable = (finPartie || promotion);
78.                 this.remplirListe(b);
79.                 this.remplirEatenPieces(b);
80.                 if (!b.getSelectedCase().equals("00"))
81.                         this.possibleMoves = b.getPiece(b.getSelectedCase())
82.                         .possibleMovesSE(b);
83.                 this.body = "<BR><BR>" + "<form>\n" + "<table align=center>\n"
```

```
84.                                 + "        <tr>\n" + "                <td
    class=\"corner\"></td>\n"
85.                                 + "                <td class=\"border\">A</td>\n"
86.                                 + "                <td class=\"border\">B</td>\n"
87.                                 + "                <td class=\"border\">C</td>\n"
88.                                 + "                <td class=\"border\">D</td>\n"
89.                                 + "                <td class=\"border\">E</td>\n"
90.                                 + "                <td class=\"border\">F</td>\n"
91.                                 + "                <td class=\"border\">G</td>\n"
92.                                 + "                <td class=\"border\">H</td>\n"
93.                                 + "                <td class=\"corner\"></td>\n" +
    "        </tr>\n";
94.                 for (int i = 8; i > 0; --i) {
95.                         this.body += "</tr>\n";
96.                         this.body += "<td class=\"border\">" + i + "</td>\n";
97.                         for (int j = 1; j < 9; ++j)
98.                                 this.body += printPiece(i, j, b, nonCliquable) + "\n";
99.                         this.body += "<td class=\"border\">" + i + "</td>\n";
100.                          this.body += "<tr>\n";
101.                 }
102.                 this.body += "        <tr>\n" + "                <td
    class=\"corner\"></td>\n"
103.                                  + "                <td class=\"border\">A</td>\n"
104.                                  + "                <td class=\"border\">B</td>\n"
105.                                  + "                <td class=\"border\">C</td>\n"
106.                                  + "                <td class=\"border\">D</td>\n"
107.                                  + "                <td class=\"border\">E</td>\n"
108.                                  + "                <td class=\"border\">F</td>\n"
109.                                  + "                <td class=\"border\">G</td>\n"
110.                                  + "                <td class=\"border\">H</td>\n"
111.                                  + "                <td class=\"corner\"></td>\n" +
    "        </tr>\n" + "</table>\n"
112.                                  + "</form>\n";
113.
114.         }
115.
116.         /**
117.          * getter de la chaine de caractère head
118.          *
119.          * @return head
120.          */
121.         public static String getHead() {
122.                 return head;
123.         }
124.
125.         /**
126.          * setter de la chaine de caractère head
127.          *
```

```
128.          * @param head
129.          */
130.         public static void setHead(String head) {
131.                 HTMLGen.head = head;
132.         }
133.
134.         /**
135.          * getter de la chaine de caractère bottom1
136.          *
137.          * @return bottom1
138.          */
139.         public static String getBottom1() {
140.                 return bottom1;
141.         }
142.
143.         /**
144.          * setter de la chaine de caractère bottom1
145.          *
146.          * @param bottom1
147.          */
148.         public static void setBottom1(String bottom1) {
149.                 HTMLGen.bottom1 = bottom1;
150.         }
151.
152.         /**
153.          * Fonction permettant l'accés au code html de la page complète
154.          *
155.          * @return chaine de caractère contenant la page entière en html
156.          */
157.         public String getPage() {
158.                 return HTMLGen.getHead() + this.boutonsUndoRedo + this.left +
    this.body
159.                                 + this.legende + HTMLGen.getBottom1() +"\n\n";
160.         }
161.
162.         /**
163.          * Fonction permettant de remplir la partie de la page contenant la pièce
    dans le plateau
164.          *
165.          * @param row
166.          * @param column
167.          * @param b
168.          * @return chaine de caractère contenant le nom de la piéce et son raccourci
169.          */
170.         public String getNomPiece(int row, int column, Board b) {
171.                 if (b.isEmpty(row, column))
172.                         return "blank.svg\" alt=\" \"";
173.                 Piece p = b.getPiece(row, column);
```

```
174.                 return getNomPiece(p);
175.         }
176.
177.         /**
178.          * Fonction permettant de remplir la partie de la page contenant la pièce
    dans le plateau
179.          *
180.          * @param p
181.          * @return chaine de caractère contenant le nom de la pièce et son raccourci
182.          */
183.         public String getNomPiece(Piece p) {
184.                 String nom = "";
185.                 nom += p.getColor() + p.getNom() + ".svg\" alt=\"" + p.getShortcut()
186.                                 + "\"";
187.                 return nom;
188.         }
189.
190.         /**
191.          * Fonction permettant l'accés du code HTML complet d'affichage d'une pièce
192.          *
193.          * @param row
194.          * @param column
195.          * @param b
196.          * @return code HTML complet d'affichage d'une pièce
197.          */
198.         public String printPiece(int row, int column, Board b, Boolean promotion) {
199.                 String pieceLine = "<td class=\"";
200.                 int somme = row + column;
201.                 int wkRow = b.getWhiteKing().getRow();
202.                 int wkColumn = b.getWhiteKing().getColumn();
203.                 int bkRow = b.getBlackKing().getRow();
204.                 int bkColumn = b.getBlackKing().getColumn();
205.                 if (((b.isEchec("white", wkRow, wkColumn) && row == wkRow && column
    == wkColumn) || (b
206.                                 .isEchec("black", bkRow, bkColumn) && row == bkRow &&
    column == bkColumn))
207.                                 && (!nameCase(row,
    column).equals(b.getSelectedCase())))
208.                         pieceLine += "echec";
209.                 else {
210.                         if (somme % 2 == 1)
211.                                 pieceLine += "even";
212.                         else
213.                                 pieceLine += "odd";
214.                 }
215.                 if (!b.getSelectedCase().equals("00")) {
216.                         if (isPlayable(row, column, b) && !promotion) {
```

```
217.                                    pieceLine += "Select\"><input type=\"image\"
    name=\"to"
218.                                            + nameCase(row, column) + "\"
    src=\"pieces/"
219.                                            + getNomPiece(row, column, b) + "
    width=32 /></td>";
220.                                    return pieceLine;
221.                            }
222.                    }
223.                    if (b.isEmpty(row, column)) {
224.                            pieceLine += "\" ><img src=\"pieces/blank.svg\" alt=\" \"
    width=32 /></td>";
225.                            return pieceLine;
226.                    }
227.                    Piece p = b.getPiece(row, column);
228.                    if (p.getColor().equals(b.getCurrentPlayer()) && !promotion) {
229.                            if (nameCase(row, column).equals(b.getSelectedCase())) {
230.                                    pieceLine += "CurrentSelect\"><input type=\"image\"
    name=\""
231.                                            + nameCase(row, column) + "\"
    src=\"pieces/"
232.                                            + getNomPiece(row, column, b) + "
    width=32 /></td>";
233.                                    return pieceLine;
234.                            } else {
235.                                    pieceLine += "\"><input type=\"image\" name=\""
236.                                            + nameCase(row, column) + "\"
    src=\"pieces/"
237.                                            + getNomPiece(row, column, b) + "
    width=32 /></td>";
238.                                    return pieceLine;
239.                            }
240.                    }
241.                    pieceLine += "\"><img src=\"pieces/" + getNomPiece(row, column, b)
242.                                    + " width=32 /></td>";
243.                    return pieceLine;
244.            }
245.
246.            /**
247.             * Fonction permettant l'accés au nom d'une case à partir de ses coordonnées
248.             *
249.             * @param row
250.             * @param column
251.             * @return nom d'une case
252.             */
253.            public String nameCase(int row, int column) {
254.                    String name = "";
255.                    switch (column) {
```

```
256.                 case 1:
257.                         name += 'A';
258.                         break;
259.                 case 2:
260.                         name += 'B';
261.                         break;
262.                 case 3:
263.                         name += 'C';
264.                         break;
265.                 case 4:
266.                         name += 'D';
267.                         break;
268.                 case 5:
269.                         name += 'E';
270.                         break;
271.                 case 6:
272.                         name += 'F';
273.                         break;
274.                 case 7:
275.                         name += 'G';
276.                         break;
277.                 case 8:
278.                         name += 'H';
279.                         break;
280.                 }
281.                 name += row;
282.                 return name;
283.         }
284.
285.         /**
286.          * Fonction permettant de savoir si une case est jouable par la pièce
    actuellement selectionnée
287.          *
288.          * @param row
289.          * @param column
290.          * @param b
291.          * @return booléen
292.          */
293.         public boolean isPlayable(int row, int column, Board b) {
294.                 Iterator<Square> it = this.possibleMoves.iterator();
295.                 while (it.hasNext()) {
296.                         Square s = it.next();
297.                         if (s.isThisSquare(row, column))
298.                                 return true;
299.                 }
300.                 return false;
301.         }
302.
```

```
303.          /**
304.           * Fonction remplissant la liste des coups
305.           *
306.           * @param b
307.           */
308.          public void remplirListe(Board b) {
309.                  String debut = "<b><font face =\"courier\" size =\"2\">\n";
310.                  String fin = "</font></b>\n";
311.                  String debutFont = "<font color =\"white\">";
312.                  String finFont = "</font>\n";
313.
314.                  this.listeCoups += debut;
315.                  for (int i = 1; i < b.getNumeroCoupMax(); ++i) {
316.                          if (i >= b.getNumeroCoup())
317.                                  debutFont = "<font color =\"grey\">\n";
318.                          if (i % 2 == 0)
319.                                  this.listeCoups += debutFont + " - "
320.                                                  +
    b.getListeCoups().get(i).afficherCoup() + finFont
321.                                                  + "<BR>";
322.                          else {
323.                                  this.listeCoups += debutFont;
324.                                  if ((i + 1) / 2 < 10)
325.                                          this.listeCoups += "0";
326.                                  this.listeCoups += (i + 1) / 2 + ". "
327.                                                  +
    b.getListeCoups().get(i).afficherCoup() + finFont;
328.                          }
329.                  }
330.                  this.listeCoups += fin;
331.          }
332.
333.          /**
334.           * Fonction permettant de remplir le code html de la légende et des boutons
    undo et redo
335.           *
336.           * @param b
337.           * @param promotion
338.           * @return
339.           */
340.          public Boolean remplirOptions(Board b, Boolean promotion, String
    partieASupprimer) {
341.                  Boolean finPartie = false;
342.                  String player = b.getCurrentPlayer();
343.                  Piece roi = b.getWhiteKing();
344.                  // String colorEnFrancais = "blanc";
345.                  String adversaire = "black";
346.                  if (player.equals("black")) {
```

```
347.                         roi = b.getBlackKing();
348.                         adversaire = "white";
349.                         // colorEnFrancais = "noir";
350.                 }
351.                 this.boutonsUndoRedo += "<center>\n";
352.                 if (b.getNumeroCoup() > 1 && !promotion)
353.                         this.boutonsUndoRedo += "<a href=\"?Undo\"
    class=\"boutonpage\">UNDO</a>\n";
354.                 else
355.                         this.boutonsUndoRedo += "<a
    class=\"boutonpagegris\">UNDO</a>\n";
356.                 if (b.getNumeroCoup() < b.getNumeroCoupMax() && !promotion)
357.                         this.boutonsUndoRedo += "<a href=\"?Redo\"
    class=\"boutonpage\">REDO</a>\n";
358.                 else
359.                         this.boutonsUndoRedo += "<a
    class=\"boutonpagegris\">REDO</a>\n";
360.                 this.boutonsUndoRedo += "</center>\n";
361.
362.                 this.legende += "<BR>\n\n";
363.                 this.legende += "<center>\n";
364.                 if (!partieASupprimer.equals("")){
365.                         this.legende += "<a class=\"bottom\">Delete " +
    partieASupprimer + " ?</a><BR>" + "<a class=\"bouton404\" href=\"?Suppr=" +
    partieASupprimer + "\"> YES </a>" + "<a class=\"bouton404\" href=\"?\"> NO </a>";
366.                 }
367.                 else if (promotion)
368.                         this.legende += "<td class=\"bottom\">\n"
369.                                         + "<a href=\"?Rook\"><img src=\"pieces/" +
    adversaire
370.                                         + "Rook.svg\"   alt=\"T\" width=32 /></a>\n"
371.                                         + "<a href=\"?Knight\"><img src=\"pieces/" +
    adversaire
372.                                         + "Knight.svg\" alt=\"C\" width=32 /></a>\n"
373.                                         + "<a href=\"?Bishop\"><img src=\"pieces/" +
    adversaire
374.                                         + "Bishop.svg\" alt=\"F\" width=32 /></a>\n"
375.                                         + "<a href=\"?Queen\"><img src=\"pieces/" +
    adversaire
376.                                         + "Queen.svg\"  alt=\"D\" width=32 /></a>\n"
    + "</td>\n";
377.                 else if (b.isEchec(player, roi.getRow(), roi.getColumn())) {
378.                         try {
379.                                 if (b.isEchecEtMat(player)) {
380.                                         this.legende += "<a
    class=\"bottom\">CHECKMATE</a>\n";
381.                                         finPartie = true;
382.                                 } else {
```

```
383.                                         if (b.getCurrentPlayer().equals("white")){
384.                                                 this.legende += "<a
    class=\"bottomwhite\">White</a>\n";
385.                                         }
386.                                         else{
387.                                                 this.legende += "<a
    class=\"bottomblack\">Black</a>\n";
388.                                         }
389.                                 }
390.                         } catch (Exception e) {
391.                                 e.printStackTrace();
392.                         }
393.                 } else if (b.isPat()) {
394.                         this.legende += "<a class=\"bottom\">STALEMATE</a>\n";
395.                         finPartie = true;
396.                 } else {
397.                         if (b.getCurrentPlayer().equals("white")){
398.                                 this.legende += "<a
    class=\"bottomwhite\">White</a>\n";
399.                         }
400.                         else{
401.                                 this.legende += "<a
    class=\"bottomblack\">Black</a>\n";
402.                         }
403.                 }
404.                 this.legende += "</center><BR>\n\n";
405.                 return finPartie;
406.         }
407.
408.         /**
409.          * Fonction permettant de remplir le code HTML du cimetière de pièces
410.          *
411.          * @param b board
412.          */
413.         public void remplirEatenPieces(Board b) {
414.                 this.eatenPieces += "<div id=\"menuright\">\n";
415.                 this.eatenPieces += "<div id=\"menuhaut\">\n";
416.                 this.eatenPieces += "<font color = \"black\">Black eaten pieces
    :<BR>\n";
417.                 this.eatenPieces += "<table align=\"center\">";
418.                 ArrayList<Piece> blackEatenPieces = b.getBlackEatenPieces();
419.                 Iterator<Piece> blackIt = blackEatenPieces.iterator();
420.                 int tr = 0;
421.                 while (blackIt.hasNext()) {
422.                         if (tr == 3) {
423.                                 this.eatenPieces += "</tr>\n";
424.                                 tr = 0;
425.                         } else {
```

```
426.                                 if (tr == 0)
427.                                         this.eatenPieces += "<tr>\n";
428.                                 Piece p = blackIt.next();
429.                                 this.eatenPieces += "<td><img src=\"pieces/" +
    getNomPiece(p)
430.                                                 + " width=32 /></td>\n";
431.                                 tr++;
432.                         }
433.                 }
434.                 if (tr != 0) {
435.                         this.eatenPieces += "</tr>\n";
436.                         tr = 0;
437.                 }
438.                 this.eatenPieces += "</table>";
439.                 this.eatenPieces += "</font>\n";
440.                 this.eatenPieces += "</div>\n";
441.                 this.eatenPieces += "<div id=\"menubas\">\n";
442.                 this.eatenPieces += "<font color = \"white\"><BR>White eaten pieces
    :<BR>\n";
443.                 this.eatenPieces += "<table align=\"center\">";
444.                 ArrayList<Piece> whiteEatenPieces = b.getWhiteEatenPieces();
445.                 Iterator<Piece> whiteIt = whiteEatenPieces.iterator();
446.                 this.eatenPieces += "</tr>\n";
447.                 while (whiteIt.hasNext()) {
448.                         if (tr == 3) {
449.                                 this.eatenPieces += "</tr>\n";
450.                                 tr = 0;
451.                         } else {
452.                                 if (tr == 0)
453.                                         this.eatenPieces += "<tr>\n";
454.                                 Piece p = whiteIt.next();
455.                                 this.eatenPieces += "<td><img src=\"pieces/" +
    getNomPiece(p)
456.                                                 + " width=32 /></td>\n";
457.                                 tr++;
458.                         }
459.                 }
460.                 if (tr != 0) {
461.                         this.eatenPieces += "</tr>\n";
462.                         tr = 0;
463.                 }
464.                 this.eatenPieces += "</table>";
465.                 this.eatenPieces += "</font>\n";
466.                 this.eatenPieces += "</div>\n";
467.                 this.eatenPieces += "</div>\n";
468.         }
469.
470.         /**
```

```
471.          * Fonction permettant de remplir le code HTML du bandeau de gauche
472.          *
473.          * @param b
474.          */
475.         public void getLeft(Board b, ArrayList<String> saves) {
476.                 this.left += "<div id=\"menu\">\n";
477.                 this.left += "<a class= \"bouton\" href= \"?NewGame\">NEW
    GAME</a>\n";
478.                 this.left += "<form class= \"deroulant\" action=\"\"
    method=\"GET\">\n SAVE GAME<BR><BR><input type=\"text\" name=\"Save\"/>\n <input
    type=\"submit\" value=\"Save Game\"/>\n</form>";
479.                 this.left += "<p class= \"deroulant\">LOAD GAME<BR><BR>";
480.                 Iterator<String> it = saves.iterator();
481.                 while (it.hasNext()) {
482.                         String s = it.next();
483.                         this.left += "<a href= \"?Load=" + s + "\">" + s +
    "</a><BR><BR>\n";
484.                 }
485.                 this.left += "</p>\n";
486.                 this.left += "<p class= \"deroulantred\">DELETE SAVE<BR><BR>";
487.                 Iterator<String> it2 = saves.iterator();
488.                 while (it2.hasNext()) {
489.                         String s2 = it2.next();
490.                         this.left += "<a href= \"?delete=" + s2 + "\">" + s2
491.                                         + "</a><BR><BR>\n";
492.                 }
493.                 this.left += "</p>\n";
494.                 this.remplirListe(b);
495.                 this.remplirEatenPieces(b);
496.                 this.left += "<a class= \"deroulant\">\nMOVE LIST<BR><BR>\n";
497.                 this.left += this.listeCoups;
498.                 this.left += "</a>\n";
499.                 this.left += "<a class= \"deroulant\">\nCIMETERY<BR><BR>\n";
500.                 this.left += eatenPieces;
501.                 this.left += "</a>\n";
502.                 this.left += "</div>\n";
503.         }
504.
505. }
```

Coup.java

```java
import java.io.Serializable;

/**
 * Classe permettant la sauvegarde d'un coup.
 */
public class Coup implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Pièce en cours de déplacement.
	 */
	private Piece movedPiece;

	/**
	 * Pièce précédente sauvegardée dans le cas d'une promotion.
	 */
	private Piece oldPiece;

	/**
	 * Booleen vérifiant si une pièce a été mangé pendant un coup.
	 */
	private Boolean hasEaten;

	/**
	 * Booleen verifiant si le coup est un petit roque.
	 */
	private Boolean isPetitRoque;

	/**
	 * Booleen verifiant si le coup est un grand roque.
	 */
	private Boolean isGrandRoque;

	/**
	 * Pièce mangée sauvegardée.
	 */
	private Piece eatenPiece;

	/**
	 * Case de départ pour le déplacement.
	 */
```

```
46.         private Square caseDepart;
47.
48.         /**
49.          * Case d'arrivée pour le déplacement.
50.          */
51.         private Square caseArrivee;
52.
53.         /**
54.          * Numéro du coup.
55.          */
56.         private int numeroCoup;
57.
58.         /**
59.          * Booleen vérifiant si le coup est une prise en passant.
60.          */
61.         private Boolean isPriseEnPassant;
62.
63.         /**
64.          * Booleen vérifiant si le coup est une promotion.
65.          */
66.         private Boolean isPromotion;
67.
68.         // dans le cas d'une promotion le pion devient eatenPiece et la nouvelle
69.         // piece devient movedPiece
70.
71.         /**
72.          * Getter de isPromotion.
73.          *
74.          * @return vrai si le coup est une promotion.
75.          */
76.         public Boolean getIsPromotion() {
77.                 return isPromotion;
78.         }
79.
80.         /**
81.          * Setter de isPromotion.
82.          *
83.          * @param isPromotion booleen si le coup est une promotion.
84.          */
85.         public void setIsPromotion(Boolean isPromotion) {
86.                 this.isPromotion = isPromotion;
87.         }
88.
89.         /**
90.          * Getter de movedPiece.
91.          *
92.          * @return la pièce en encours de déplacement.
93.          */
```

```
94.         public Piece getMovedPiece() {
95.                 return movedPiece;
96.         }
97.
98.         /**
99.          * Setter de movedPiece.
100.          *
101.          * @param movedPiece la pièce en encours de déplacement.
102.          */
103.         public void setMovedPiece(Piece movedPiece) {
104.                 this.movedPiece = movedPiece;
105.         }
106.
107.         /**
108.          * Getter de oldPiece.
109.          *
110.          * @return la pièce précédente (avant le roque).
111.          */
112.         public Piece getOldPiece() {
113.                 return oldPiece;
114.         }
115.
116.         /**
117.          * Setter de oldPiece
118.          *
119.          * @param oldPiece la pièce précédente (avant le roque).
120.          */
121.         public void setOldPiece(Piece oldPiece) {
122.                 this.oldPiece = oldPiece;
123.         }
124.
125.         /**
126.          * Getter de hasEaten.
127.          *
128.          * @return vrai si une pièce a été mangée pendant le coup.
129.          */
130.         public Boolean getHasEaten() {
131.                 return hasEaten;
132.         }
133.
134.         /**
135.          * Setter de hasEaten.
136.          *
137.          * @param hasEaten
138.          */
139.         public void setHasEaten(Boolean hasEaten) {
140.                 this.hasEaten = hasEaten;
141.         }
```

```
142.
143.         /**
144.          * Getter de isPetitRoque.
145.          *
146.          * @return vrai si le coup est un petitRoque.
147.          */
148.         public Boolean getIsPetitRoque() {
149.                 return isPetitRoque;
150.         }
151.
152.         /**
153.          * Setter de isPetitRoque.
154.          *
155.          * @param isPetitRoque
156.          */
157.         public void setIsPetitRoque(Boolean isPetitRoque) {
158.                 this.isPetitRoque = isPetitRoque;
159.         }
160.
161.         /**
162.          * Getter de isGrandRoque.
163.          *
164.          * @return
165.          */
166.         public Boolean getIsGrandRoque() {
167.                 return isGrandRoque;
168.         }
169.
170.         /**
171.          * Setter de isGrandRoque.
172.          *
173.          * @param isGrandRoque
174.          */
175.         public void setIsGrandRoque(Boolean isGrandRoque) {
176.                 this.isGrandRoque = isGrandRoque;
177.         }
178.
179.         /**
180.          * Getter de eatenPiece.
181.          *
182.          * @return la pièce mangée.
183.          */
184.         public Piece getEatenPiece() {
185.                 return eatenPiece;
186.         }
187.
188.         /**
189.          * Setter de eatenPiece.
```

```
190.          *
191.          * @param eatenPiece
192.          */
193.         public void setEatenPiece(Piece eatenPiece) {
194.                 this.eatenPiece = eatenPiece;
195.         }
196.
197.         /**
198.          * Getter de caseDepart.
199.          *
200.          * @return la case de départ.
201.          */
202.         public Square getCaseDepart() {
203.                 return caseDepart;
204.         }
205.
206.         /**
207.          * Setter de caseDepart.
208.          *
209.          * @param caseDepart
210.          */
211.         public void setCaseDepart(Square caseDepart) {
212.                 this.caseDepart = caseDepart;
213.         }
214.
215.         /**
216.          * Getter de caseArrivee.
217.          *
218.          * @return la case d'arrivée.
219.          */
220.         public Square getCaseArrivee() {
221.                 return caseArrivee;
222.         }
223.
224.         /**
225.          * Setter de caseArrivee.
226.          *
227.          * @param caseArrivee
228.          */
229.         public void setCaseArrivee(Square caseArrivee) {
230.                 this.caseArrivee = caseArrivee;
231.         }
232.
233.         /**
234.          * Getter du numero du coup.
235.          */
236.         public int getNumeroCoup() {
237.                 return numeroCoup;
```

```
238.         }
239.
240.         /**
241.          * Setter du numero du coup.
242.          *
243.          * @param numeroCoup le numero du coup.
244.          */
245.         public void setNumeroCoup(int numeroCoup) {
246.                 this.numeroCoup = numeroCoup;
247.         }
248.
249.         /**
250.          * Getter de isPriseEnPassant.
251.          *
252.          * @return true si le coup est une prise en passant.
253.          */
254.         public Boolean getIsPriseEnPassant() {
255.                 return isPriseEnPassant;
256.         }
257.
258.         /**
259.          * Setter de isPriseEnPassant.
260.          *
261.          * @param isPriseEnPassant
262.          */
263.         public void setIsPriseEnPassant(Boolean isPriseEnPassant) {
264.                 this.isPriseEnPassant = isPriseEnPassant;
265.         }
266.
267.         /**
268.          * Constructeur du coup.
269.          */
270.         public Coup() {
271.                 this.isPromotion = false;
272.                 this.isPriseEnPassant = false;
273.                 this.hasEaten = false;
274.                 this.isGrandRoque = false;
275.                 this.isPetitRoque = false;
276.         }
277.
278.         /**
279.          * Affichage du coup.
280.          *
281.          * @return         un string contenant les informations
282.          *                         sur son type sa case de départ et sa case
    d'arrivée.
283.          */
284.         public String afficherCoup() {
```

```
285.                 if (this.isPetitRoque)
286.                         return "0-0";
287.                 if (this.isGrandRoque)
288.                         return "0-0-0";
289.                 String s = "";
290.                 s += this.caseDepart.getNomCase();
291.                 if (this.hasEaten)
292.                         s += "x";
293.                 else
294.                         s += "-";
295.                 s += this.caseArrivee.getNomCase();
296.                 if (this.isPromotion)
297.                         s += "=" + this.movedPiece.getShortcut();
298.                 return s;
299.         }
300.
301. }
```

SavedGameList.java

```java
import java.io.Serializable;
import java.util.ArrayList;

/**
 * Liste des sauvegardes
 *
 */
public class SavedGamesList implements Serializable {

	/**
	 * Default serial version id
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Liste de sauvegardes
	 */
	private ArrayList<String> savedGames;

	/**
	 * getter de la liste des sauvegardes
	 * @return liste des sauvegardes
	 */
	public ArrayList<String> getSavedGames() {
		return savedGames;
	}

	/**
	 * setter de la liste des sauvegardes
	 * @param savedGames liste des sauvegardes
	 */
	public void setSavedGames(ArrayList<String> savedGames) {
		this.savedGames = savedGames;
	}

	/**
	 * Ajout d'une sauvegarde à la liste
	 * @param s une sauvegarde
	 */
	public void newSave(String s) {
		savedGames.add(s);
	}

	/**
	 * constructeur sans paramètre
```

```
46.          */
47.         public SavedGamesList() {
48.                 this.savedGames = new ArrayList<String>();
49.         }
50.
51.         /**
52.          * Verification qu'une sauvegarde existe
53.          * @param s nom de sauvegarde
54.          * @return true si la sauvegarde est dans la liste
55.          */
56.         public Boolean isSave(String s) {
57.                 return savedGames.contains(s);
58.         }
59.
60. }
```

OutOfBoardException.java

```java
/**
 * Exception prévue pour un coup qui serait en dehors du plateau de jeu.
 */
public class OutOfBoardException extends NonPossibleMoveException {

	private static final long serialVersionUID = 1L;

	public OutOfBoardException(String s) {
		super(s);
	}

}
```

NonPossibleMoveException.java

```java
/**
 * Exception prévue pour un coup impossible.
 */
public class NonPossibleMoveException extends Exception {

	private static final long serialVersionUID = 1L;

	public NonPossibleMoveException(String s) {
		super(s);
	}

}
```

EchecException.java

```java
/**
 * Exception prévue pour un coup qui mettrai son roi en échec.
 */
public class EchecException extends NonPossibleMoveException {

	private static final long serialVersionUID = 1L;

	public EchecException(String s) {
		super(s);
	}

}
```