Programmation Orientée Objet JAVA

Projet

Recommandation

- Lisez bien tout le document avant de commence à travailler..

Organisation

- Un travail original (entendre individuel…) est attendu de chaque binôme. Il convient de citer ses sources (Internet, autre binôme, etc.).
- L'objectif minimum à atteindre pour les moins rapides est la fin de l'itération 2.

Suivi

- Les séances sont encadrées par 2 enseignants.
- Chaque membre du binôme sera invité à s'exprimer et à justifier les choix de conception et de programmation.

Rendus

- Les sources Java. Un dépôt de fichier sera ouvert à cet effet sur le e-campus.
- Il n'y aura pas d'autres documents à rédiger ni à livrer.

Évaluation :

- Rentre en compte à hauteur de 20% dans l'évaluation du module.

Classe concernée : 3IRC – 4ETI majeure IWOC
Date d'émission : décembre 2015, release mai 2016
Émetteur : Françoise Perrin
Sur une idée de Jacques Saraydaryan et Adrien Guénard

Contact : fp@cpe.fr, martine.breda@cpe.fr, alain.becker@cpe.fr
Service : Info/Télécom – Bureau B123
Disponible sur : e-campus

1 Objectifs et enjeux du projet

1.1 Enjeux

Être capable de concevoir et développer en Java des programmes souples, extensibles et faciles à maintenir. Cela suppose de respecter les principes suivants afin de garantir une forte cohésion et un faible couplage :

- Responsabilité unique : une classe ne doit avoir qu'une seule raison de changer.
- Ouverture fermeture : une classe doit être ouverte aux extensions mais fermées aux modifications.
- Substitution de LISKOV : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaitre (polymorphisme).

1.2 Moyens

L'objectif pédagogique de ce projet est de mettre en œuvre les différents concepts de la Programmation Orientée Objet à travers un jeu d'échec.

3 itérations sont possibles :

1. Pour tous : programmer et tester les déplacements simples en mode console.
2. Pour tous : programmer et tester les déplacements simples en mode graphique évènementiel. 1 seul damier et 2 joueurs sur le même damier.
3. Pour les plus avancés, au choix :
   - Améliorer l'algorithme de déplacement des pièces pour gérer la présence de pièces intermédiaires, assurer la promotion du pion, effectuer le roque du roi, être capable de dire si le roi est en échec, en échec et mat, etc.
   - Permettre à 2 joueurs de jouer ensemble sur des postes distants (dans un 1er temps 1 damier pour chaque joueur sur le même poste).

La conception de la solution est imposée (itérations 1 et 2) dans le respect du pattern MVC. Les méthodes des classes proposées (Cf. Javadoc du projet sur le e-campus), doivent être codées en Java.

Un ensemble de interfaces/classes sont fournies (e-campus).

1.3 Points techniques abordés

- Programmation graphique et évènementielle : packages `javax.swing` et `java.awt`
- Framework de collections : package `java.util`
- Introspection de classes : packages `java.lang et java.lang.reflect`
- Sockets réseaux, sérialisation et threads : packages `java.lang, java.net, java.io`

2 Conception du projet

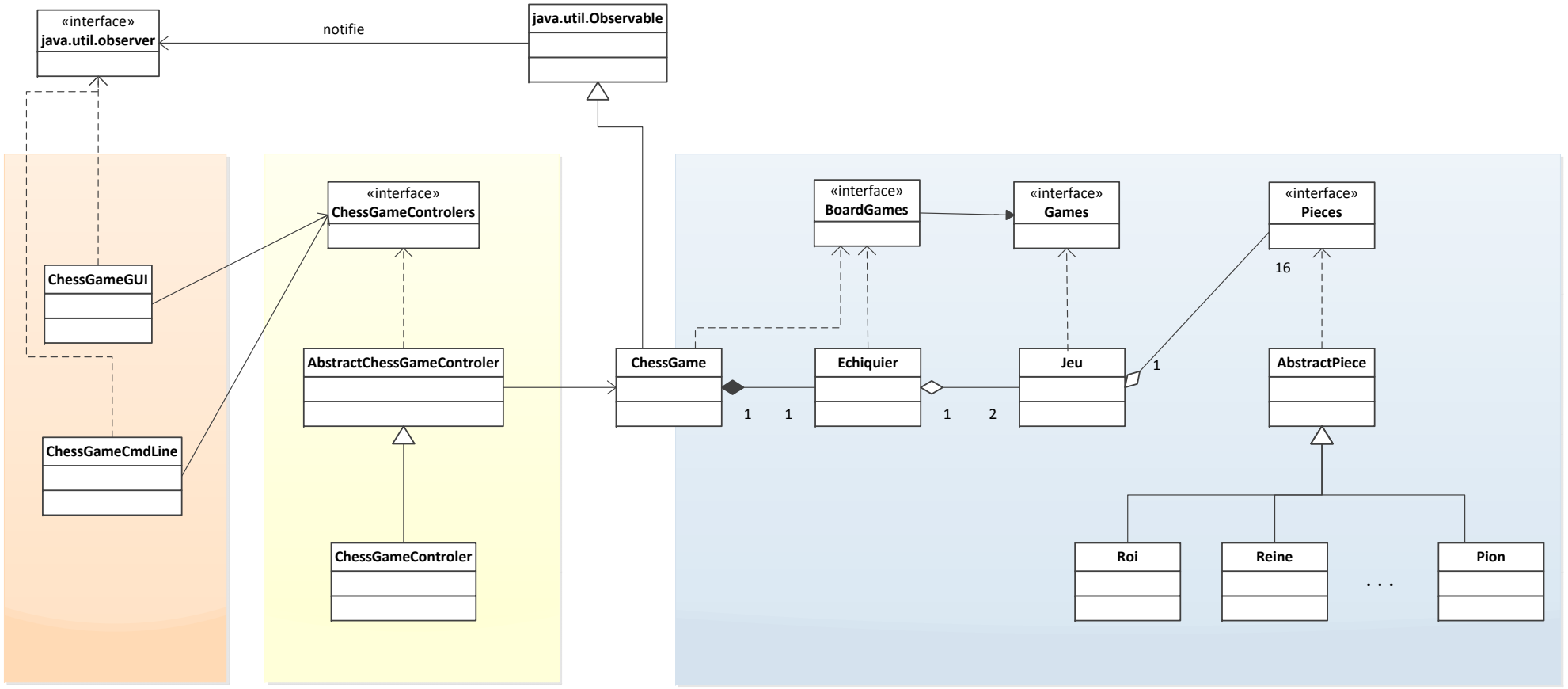
La conception du projet a permis d'identifier un certain nombre de classes « métier » (package « model ») et en particulier :

- Des pièces : roi, reine, fous, pions, cavaliers, tours. Il existe 16 pièces blanches et 16 pièces noires.
- Des jeux : ensemble des pièces d'un joueur. Il existe 1 jeu blanc et 1 jeu noir.
- 1 échiquier qui contient les 2 jeux.

Chaque classe a ses propres responsabilités. Ainsi la classe `Jeu` est responsable de créer ses `Pieces` et de les manipuler. `L'Echiquier` quant à lui crée les 2 jeux mais ne peut pas manipuler directement les pièces. Pour autant, c'est lui qui est capable de dire si un déplacement est légal, d'ordonner ce déplacement, de gérer l'alternance des joueurs, de savoir si le roi est en échec et mat, etc. Pour ce faire, il passe donc par les objets `Jeu` pour communiquer avec les `Pieces`.

Les classes sont donc parfaitement bien encapsulées et les seules interactions possibles avec une IHM se font à travers `L'Echiquier` et en aucun cas une IHM ne pourra directement déplacer une `Pieces` sans passer par les méthodes de `L'Echiquier` (en fait à travers une classe `ChessGame` – Cf. plus loin).

L'architecture globale de l'application est schématisée ci-dessous.



2.1 Interfaces « métier »

L'interface `BoardGames` définit le comportement attendu de tous les jeux de plateau (Pourrait être implémentée par un jeu d'échec, de dames, etc.).

L'interface `Games` définit le comportement attendu de toutes les implémentations de jeux de plateau (Pourrait être implémentée par 2 jeux qui contiennent chacun une liste de pièce comme c'est le cas dans notre application, mais aussi par 1 tableau 2D, etc.).

L'interface `Pieces` définit le comportement attendu de toutes les pièces.

2.2 Classe `AbstractPiece`

La classe `AbstractPiece` définit le comportement attendu de toutes les pièces (spécifié dans l'interface `Pieces`). En revanche, c'est à chaque classe dérivée de `AbstractPiece` (par exemple `Pion`), connaissant ses coordonnées initiales (`x`, `y`), de dire si un déplacement vers une destination finale est possible.

Le détail des algorithmes de déplacement des pièces dans un jeu d'échec est décrit globalement sur http://fr.wikipedia.org/wiki/échecs et en détail sur http://fr.wikipedia.org/wiki/Cavalier_(échecs), etc.

2.3 Classe `Jeu`

La classe `Jeu` stocke ses pièces dans une liste de `Pieces`. Il fait appel à une fabrique pour créer les pièces à leurs coordonnées initiales (Cette fabrique vous est donnée).

Le jeu est capable de « relayer » les demandes de l'échiquier auprès de ses pièces pour savoir si le déplacement est possible, le rendre effectif, etc.

2.4 Classe `Echiquier`

La classe `Echiquier` crée ses 2 `Jeu` et connait le jeu courant.

Elle est munie d'une méthode qui vérifie si le déplacement est possible et une autre qui permet de déplacer une pièce depuis ses coordonnées initiales vers ses coordonnées finales avec prise éventuelle.

La méthode `isMoveOk()` retourne `true` si le déplacement est effectif, c'est-à-dire si :

- La pièce concernée appartient au jeu courant.
- La position finale est différente de la position initiale et dans les limites du damier.
- Le déplacement est possible par rapport au type de pièce, indépendamment des autres pièces.
- Le déplacement est possible par rapport aux autres pièces qui seront sur la trajectoire avec prise éventuelle.

Pour autant, la classe `Echiquier` ne communique pas directement avec les `Pieces`…

3 1ère itération : classes « métier » et IHM en mode console

Téléchargez le fichier zippé « 3IRC 4ETI POO Projet 1516.zip » fourni sur le e-campus.

3.1 Mettez en place l'architecture du projet

- En 1er lieu, étudiez la Javadoc du projet (point d'entrée `index.html`) et appropriez-vous l'organisation des classes dans les différents packages en les rapprochant du diagramme de classe UML.
- Etudiez ensuite rapidement le détail des méthodes de chaque classe. Vous y reviendrez chaque fois que vous programmerez une méthode.

- Créez votre projet Java avec l'IDE de votre choix (Eclipse ou autre) en créant les packages, les classes dans les packages les méthodes et les attributs (identifiables par les getters et setters) dans les classes selon les indications de la Javadoc. Intégrez les interfaces/classes du fichier zippé « 3IRC 4ETI POO Fichiers pour projet 1516 » (e-campus) en les remettant dans les bons packages.

3.2 Programmez la hiérarchie des pièces

- Codez l'interface `Pieces`, la classe `AbstractPiece`, puis la classe `Tour`. La méthode `toString()` de la classe `AbstractPiece` retourne le nom et les coordonnées x et y de la pièce.
- Testez les différentes méthodes de la classe `Tour` (celles héritées puis celles spécifiques). Pour ce faire, créez une fonction main(), dans la classe `AbstractPiece` par exemple, avec des instructions du type :
  `Pieces maTour = new Tour(Couleur.NOIR, new Coord(0, 0)), etc.`
- Quand toutes les méthodes de la classe `Tour` ont le comportement attendu, codez et testez les déplacements des autres pièces pour vérifier si sans tenir compte de la position des autres pièces vos algorithmes sont exacts. Programmez éventuellement la classe `Pion` sans tenir compte des déplacements en diagonale dans un 1er temps.

3.3 Programmez la classe Jeu

- Etudiez l'interface `Games`.
- Etudiez la classe `ChessPieceFactory` du package `tools` (lancez sa fonction `main()` pour observer la trace d'exécution, puis analysez son code et celui des autres Class/Enum qu'elle utilise) – ce n'est pas très grave si vous n'en comprenez pas toutes les petites lignes. Programmez le constructeur de la classe `Jeu` en utilisant cette fabrique.
- Munissez la classe `Jeu` d'une méthode `toString()` (qui invoque la méthode `toString()` de chaque `Pieces`) et d'une fonction `main()` puis testez que les pièces sont bien construites.
- Codez et testez une méthode de recherche qui sera utilisée par beaucoup d'autres méthodes, dont la signature est la suivante : `private Pieces findPiece(int x, int y)`.
- Codez et testez au fur et à mesure les autres méthodes de la classe `Jeu`.
  - La méthode `getPiecesIHM()` un peu complexe est donnée en annexe (les `PieceIHMs` enveloppent des `Pieces` – mise en œuvre du DP Adapter).
  - Ne codez pas dans cette 1ère itération les méthodes `capture()` et `setPossibleCapture()` qui nécessitent la prise en compte des pièces intermédiaires.

3.4 Programmez la classe Echiquier

- Etudiez l'interface `BoardGames`.
- La classe `Echiquier` manipule 1 `Jeu` blanc, 1 `Jeu` noir, sait quel est le `Jeu` courant et le `Jeu` non courant. Ne prévoyez pas de getter ni setter sur ces attributs ; soyez capables de justifier pourquoi.
- Munissez-la d'un attribut `message` avec 1 getter public et 1 setter privé, de manière à ce qu'une IHM puisse tracer le bon déroulement de l'exécution (« déplacement OK, déplacement interdit, etc. »).
- Programmez son constructeur, munissez-la d'une méthode `toString()` (qui invoque la méthode `toString()` de chaque `Jeu`) et testez la création et l'affichage d'un `Echiquier` dans une fonction `main()`.
- Programmez et testez dans la classe `Echiquier` la méthode `switchJoueur()`.
- Programmez et testez dans la classe `Echiquier` la méthode `isMoveOk()` qui vérifie si un déplacement simple qui ne tient pas compte des pièces intermédiaires ni des éventuelles captures de pièce est autorisé.
- Programmez et testez dans la classe `Echiquier` la méthode `move()` qui effectue les déplacements s'ils sont possibles.
- Programmez et testez dans la classe `Echiquier` la méthode `getPiecesIHM ()` qui retourne 1 liste de `PiecesIHMs` fabriquée à partir de celle des 2 `Jeu`.

3.5 Etudiez la mise en œuvre du pattern MVC et testez le mode console

Le pattern d'architecture MVC distingue :

1. Le modèle qui correspond à la partie métier (Echiquier, Jeu, etc.),
2. La vue qui est une représentation (rendu) des données du métier et qui permet l'interaction avec l'utilisateur,
3. Le contrôleur qui fait le lien entre la vue (action de l'utilisateur) et le modèle. Il contrôle la cohérence des informations envoyées par la vue, les transforme éventuellement avant de les passer au modèle et effectue certains traitements pour la vue.

A minima, dans notre architecture MVC sont mis en œuvre 3 Design Patterns :

1. Composite : la vue est composée d'un ensemble de composants graphiques et est elle-même un composant graphique.
2. Strategy : la vue délègue au contrôleur des traitements qui utilisent ou qui ont un impact sur les données du modèle (elle ne communique donc pas directement avec le modèle).
   - En particulier, on postule que la vue manipule pour gérer les déplacements 2 objets `Coord` (x, y) et non pas 4 `int`, comme attendu par la classe `Echiquier`, donc le contrôleur effectuera la transformation.
   - Par ailleurs, la vue (graphique) empêchera tout déplacement de l'image de la pièce par le joueur si ce n'est pas son tour de jouer. Pour autant, c'est au contrôleur de le vérifier.
3. Observer : le modèle est observé par la vue ce qui implique qu'à chaque changement d'état du modèle (par exemple déplacement d'une pièce ou promotion du pion), la vue soit mise à jour (déplacement/changement de l'image de la pièce sur le damier). Pour ce faire, on doit rendre le modèle observable (classe `ChessGame` fournie) pour qu'il soit observé par la vue. Cette dernière (la vue) devra implémenter l'interface `Observer` et sera munie d'une méthode `update()` qui aura la responsabilité de rafraichir l'affichage après un déplacement.

Pour mettre en place cette architecture, un certain nombre de classes vous sont fournies (e-campus), plus ou moins à compléter :

- La classe `LauncherCmdLine` lance l'application (crée un objet de la classe `ChessGameCmdLine` qui teste et affiche les résultats en mode console).
- La classe `ChessGame` réduit légèrement l'interface de la classe `Echiquier` et surtout le rend "observable".
- La classe `chessGameControler (+ AbstractChessGameControler)` sert d'intermédiaire entre la vue (`ChessGameCmdLine`) et le modèle (`ChessGame`) et transforme les messages venant de la vue pour qu'ils soient compréhensibles par le modèle. La vue ne communique donc qu'avec le contrôleur.

Testez :

- Etudiez la classe `ChessGameCmdLine` et en particulier sa méthode `update()` puis lancez l'application (`LauncherCmdLine`) pour vérifier la bonne mise en œuvre du pattern MVC. Le résultat après l'appel de `chessGameControler.move(new Coord(3,6), new Coord(3, 4));` serait le suivant :

```
Déplacement de 3,6 vers 3,4 = OK : déplacement sans capture

    0     1     2     3     4     5     6     7
0 N_To  N_Ca  N_Fo  N_Re  N_Ro  N_Fo  N_Ca  N_To
1 N_Pi  N_Pi  N_Pi  N_Pi  N_Pi  N_Pi  N_Pi  N_Pi
2 ____  ____  ____  ____  ____  ____  ____  ____
3 ____  ____  ____  ____  ____  ____  ____  ____
4 ____  ____  ____  B_Pi  ____  ____  ____  ____
5 ____  ____  ____  ____  ____  ____  ____  ____
6 B_Pi  B_Pi  B_Pi  ____  B_Pi  B_Pi  B_Pi  B_Pi
7 B_To  B_Ca  B_Fo  B_Re  B_Ro  B_Fo  B_Ca  B_To
```

- Complétez la classe `ChessGameCmdLine` pour tester quelques déplacements supplémentaires. En cas d'erreur, si vos tests unitaires ont été bien faits sur les `Pieces`, vous ne devriez avoir à ne modifier que votre classe `Echiquier` et éventuellement les méthodes de la classe `Jeu`.

4 2ème itération : IHM en mode graphique

4.1 Travail préalable

- Créez votre environnement de test en mode graphique avec la classe `LauncherGUI` (e-campus) qui lance l'application et la classe `ChessGameGUI` qui teste et affiche les résultats en mode graphique (à coder). Cette dernière doit étendre `JFrame` et implémenter les interfaces `MouseListener`, `MouseMotionListener` et `Observer`.
  Vous allez donc coupler une nouvelle vue (`ChessGameGUI`) avec le même contrôleur (`chessGameControler`) qui agit sur le même modèle (`ChessGame` qui sert de façade à votre modèle).
- Créez un dossier « images » dans votre projet et copiez-y les images fournies sur le e-campus.
- Etudiez l'exemple d'affichage et de déplacement de pièces sur un jeu d'échec sur le site http://www.roseindia.net/java/example/java/swing/chess-application-swing.shtml.
- Enregistrez le code de l'exemple dans un fichier de test, mettez à jour les noms des images avec ceux de quelques images récupérées du e-campus et testez. Vous constatez que vous pouvez déplacer des images de pièces, pour autant, vous n'avez pas de logique « métier » derrière pour vérifier la « légalité » de vos déplacements.
- Etudiez la classe `ChessImageProvider` du package `tools` (lancez sa fonction `main()` pour observer la trace d'exécution, puis analysez son code et celui de la classe `ChessPieceImage`). Sa méthode vous servira pour créer les images des pièces sur votre damier.

4.2 Programmez la classe `ChessGameGUI`

- Inspirez-vous de l'exemple pour identifier vos premiers attributs et coder votre constructeur. Ce dernier construit le plateau de l'échiquier sous forme de damier 8*8, et le rend écoutable par les évènements `MouseListener` et `MouseMotionListener`.
- Créez dans un 1er temps un damier vide (sans les images des pièces) et testez.

- Ajoutez les pièces à leur position initiale en vous servant des méthodes de la classe `ChessImageProvider` et testez.

- Programmez les déplacements et testez :
  - Votre vue (classe `ChessGameGUI`) observe votre modèle (classe `ChessGame`) et doit être munie d'une méthode `update()` qui a la responsabilité de rafraichir l'affichage après un déplacement, une promotion du pion, etc.
  - Lorsque l'utilisateur sélectionne une image de pièce pour la déplacer, le programme doit vérifier si le déplacement est légal en interrogeant le `chessGameControler`, sinon, l'image de la pièce est repositionnée à sa position initiale.
    Au fur et à mesure, les messages (« `OK : déplacement simple` ») apparaissent dans la console ou dans une popup.

- Dans la classe `chessGameControler` complétez la méthode `isPlayerOK()` et invoquez là correctement dans la vue pour empêcher de déplacer une pièce si ce n'est pas le tour du joueur.

5 3ème itération : au choix

5.1 Permettre à 2 joueurs de jouer ensemble sur des postes distants.

Il s'agit cette fois d'avoir 2 instances, de ce qui semble être le même programme, qui s'exécutent en même temps et qui communiquent ensemble. En effet, après chaque déplacement autorisé sur l'un des damiers, le déplacement doit être « visible » sur le damier de l'autre joueur.

La conception respectée jusqu'à maintenant fait qu'il n'y aura aucun changement a effectuer sur aucune classe métier, ni sur la vue graphique. Il suffit d'un nouveau controleur `chessGameControler` qui implémente l'interface `Runnable` et qui invoke des méthodes du modèle d'une part (`chessGame.move(…)`) et des méthodes d'un canal de communication à travers des sockets d'autre part.

En réalité, il existe 1 application coté Server (à lancer en 1er) et 1 coté Client (à lancer après), chacune ayant 1 modèle, 1 vue, 1 contrôleur ou bien 1 serveur et 2 clients (1 seule instance du modèle) ce qui est plus usuel mais plus difficile à mettre en œuvre (à votre choix). Des « threads » permettant de gérer respectivement l'envoi et la réception de données permettront une communication non bloquante entre les 2 programmes.

5.1.1 Travail préalable

- Etudiez le cours « d'Introduction aux sockets » sur : http://openclassrooms.com/courses/introduction-aux-sockets-1 Une version simplifiée de son exemple de synthèse est disponible dans le fichier « ExempleSocketOpenClassroms » sur le e-campus. Testez-là pour vous approprier le mode de fonctionnement des sockets et des threads en Java.
- Eventuellement, complétez ou commencez votre étude avec le chapitre sur les threads du tutoriel « Apprenez à programmer en Java » sur : https://zestedesavoir.com/tutoriels/404/apprenez-a-programmer-en-java/558/java-et-la-programmation-evenementielle/2710/executer-des-taches-simultanement/

  Le chapitre sur la synchronisation de plusieurs threads et les paragraphes suivants ne sont pas indispensables à ce projet.

5.1.2 Mise en œuvre dans votre projet

- Complétez votre diagramme de classe.
- Transformer les classes de l'exemple d'OpenClassrooms (e-campus) de manière à sérialiser des `Object` et non plus des `String` ou sinon, prévoyez de programmer votre parseur de `String`.
- Intégrez la gestion de la communication à votre projet. La nouvelle classe `chessGameControler` s'appuie sur les classes modifiées (`Object` au lieu de `String`) qu'il faut légèrement aménager.
  - Testez dans un 1er temps en local : "127.0.0.1".
  - Puis testez sur 2 machines distinctes (et donc distantes). Pour connaitre l'adresse IP de la machine que vous considèrerez comme étant votre serveur : `ifconfig` (sous Linux).

5.2 Améliorez vos classes « métier »

- Prenez soin de les sauvegarder (package model) dans un autre dossier avant de les modifier.
- Vous pouvez :
  - Proposer à l'utilisateur l'affichage des destinations possibles lorsqu'il a sélectionné une pièce.
  - Enrichir l'algorithme de déplacement en prenant en compte les pièces intermédiaires et la capture, en gérant le déplacement en diagonale du pion, en prévoyant la promotion du pion, le roque du roi, etc.
  - Annuler le déplacement si le roi est en échec, arrêter la partie lorsqu'elle est gagnée (échec et mat) ou lorsqu'elle est nulle. etc.
- Attention :
  - Identifiez bien les interfaces/classes et méthodes responsables des nouvelles actions en veillant à respecter l'encapsulation initiale et à limiter l'impact des évolutions sur les classes existantes.
  - Réfléchissez aux impacts sur l'IHM (promotion, destinations, etc.).

6 Annexe

```java
/**
 * @return une version réduite de la liste des pièces en cours
 * ne donnant que des accès en lecture sur des PieceIHMs
 * (type piece + couleur + coordonnées)
 */
public List<PieceIHMs> getPiecesIHM(){
	PieceIHMs newPieceIHM = null;
	List<PieceIHMs> list = new LinkedList<PieceIHMs>();

	for (Pieces piece : pieces){
		// si la pièce est toujours en jeu
		if (piece.getX() != -1){
			newPieceIHM = new PieceIHM(piece);
			list.add(newPieceIHM);
		}
	}
	return list;
}
```