Programmation Orientée Objet JAVA

Projet

Lisez bien tout le document avant de commencer à travailler

Classe concernée : 3IRC
Date d'émission : mai 2015
Émetteur : Françoise Perrin
Sur une idée de Jacques Saraydaryan et Adrien Guénard

Contact : *fp@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 la majorité : 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, 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

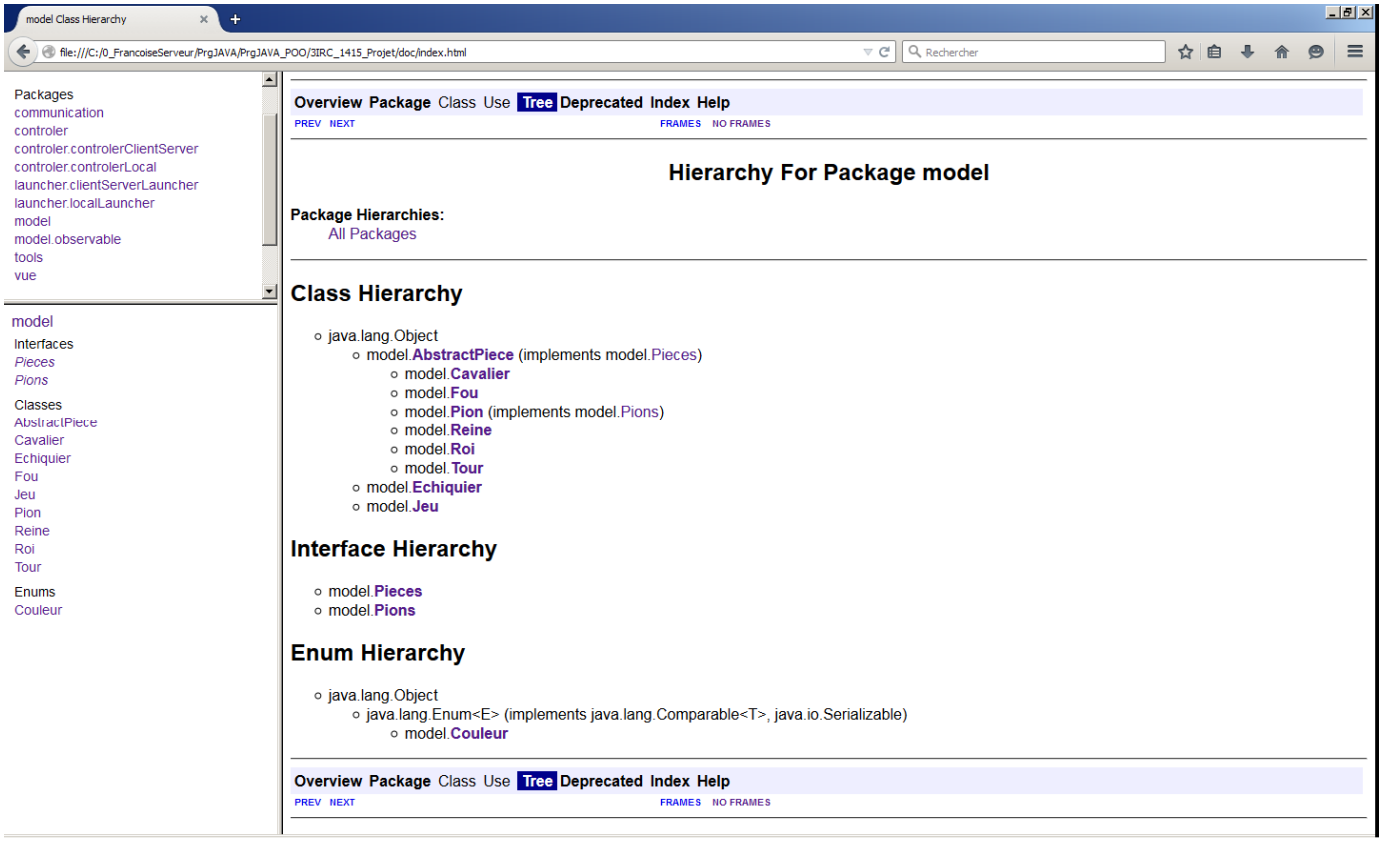
La conception du projet a permis d'identifier un certain nombre de classes « métier » (package « model ») et en particulier :

- 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).

La hiérarchie de classes « métier » est représentée ci-dessous.



2.1 Interfaces « métier »

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

L'interface `Pions` définit le comportement supplémentaire des `Pion` pour tester la prise en diagonale.

Pour enrichir l'algorithme de déplacement/prise (3ème itération), il conviendra d'enrichir l'interface `Pions` (pour promotion du pion) ou de créer de nouvelles interfaces (pour roque du roi, etc.).

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 pour chaque type de pièce 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 permet de déplacer une pièce depuis ses coordonnées initiales vers ses coordonnées finales avec prise éventuelle.

Cette méthode 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

3.1 Etudiez la structure 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 dessinant le 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 POO Fichiers pour projet 1415 » (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()` retourne le nom et les coordonnées x et y de la pièce.
- Testez les différentes méthodes de la classe Tour (celles héritées puis celle spécifique). Pour ce faire, créez une fonction main() dans la classe `AbstractPiece` avec des instructions du type :
  `Pieces maTour = new Tour(« N_To1 », 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 localement (sans tenir compte des autres pièces) vos algorithmes sont exacts. Programmez la classe `Pion` sans tenir compte des déplacements en diagonale.

3.3 Programmez la classe `Jeu`

- 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`.
- Munissez la classe `Jeu` d'une méthode `toString()` et d'une fonction `main()` et testez que les pièces sont bien construites.
- Codez et testez la méthode `findPiece()` qui est utilisée par beaucoup d'autres méthodes.
- Codez et testez au fur et à mesure les autres méthodes de la classe `Jeu` qui font appel aux méthodes des `Pieces`. Ne codez pas dans cette 1ère itération les méthodes `capture()` et `isPieceToCatchHere()` qui nécessitent la prise en compte des pièces intermédiaires.

3.4 Programmez la classe `Echiquier`

- 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 retourne une représentation du jeu d'échec sous la forme ci-dessous et testez la création et l'affichage d'un `Echiquier` dans une fonction `main()`.

```
      0    1     2     3     4     5     6     7
0  N_To1 N_Ca1 N_Fo1 N_Re1 N_Ro1 N_Fo2 N_Ca2 N_To2
1  N_Pi1 N_Pi2 N_Pi3 N_Pi4 N_Pi5 N_Pi6 N_Pi7 N_Pi8
2   ___   ___   ___   ___   ___   ___   ___   ___
3   ___   ___   ___   ___   ___   ___   ___   ___
4   ___   ___   ___   ___   ___   ___   ___   ___
5   ___   ___   ___   ___   ___   ___   ___   ___
6  B_Pi1 B_Pi2 B_Pi3 B_Pi4 B_Pi5 B_Pi6 B_Pi7 B_Pi8
7  B_To1 B_Ca1 B_Fo1 B_Re1 B_Ro1 B_Fo2 B_Ca2 B_To2
```

avec "N_To1" qui est le nom de la 1ère tour noire, etc.

- Programmez dans la classe `Echiquier` la méthode `switchJoueur()`.
- Programmez dans la classe `Echiquier` la méthode `move()` qui gère les déplacements simples sans tenir compte des pièces intermédiaires ni des éventuelles captures de pièce.

3.5 Etudiez votre environnement de test en mode console et testez

- La classe `LauncherCmdLine` (e-campus) lance l'application (crée un objet de la classe `ChessGameCmdLine` (e-campus) qui teste et affiche les résultats en mode console).
- La classe `ChessGame` (e-campus) réduit légèrement l'interface de la classe `Echiquier` et surtout le rend "observable". Cela servira pour mettre à jour la vue graphique (pas dans l'immédiat donc).
- La classe `chessGameControler` (e-campus) 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.

  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 simple

    0     1     2     3     4     5     6     7
0 N_To1 N_Ca1 N_Fo1 N_Re1 N_Ro1 N_Fo2 N_Ca2 N_To2
1 N_Pi1 N_Pi2 N_Pi3 N_Pi4 N_Pi5 N_Pi6 N_Pi7 N_Pi8
2  ___   ___   ___   ___   ___   ___   ___   ___
3  ___   ___   ___   ___   ___   ___   ___   ___
4  ___   ___   ___  B_Pi4  ___   ___   ___   ___
5  ___   ___   ___   ___   ___   ___   ___   ___
6 B_Pi1 B_Pi2 B_Pi3  ___  B_Pi5 B_Pi6 B_Pi7 B_Pi8
7 B_To1 B_Ca1 B_Fo1 B_Re1 B_Ro1 B_Fo2 B_Ca2 B_To2
```

- Complétez la classe `ChessGameCmdLine` (e-campus) et testez les déplacements des différentes pièces en vérifiant si vos algorithmes sont exacts, cette fois ci en considérant l'ensemble des pièces. 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`.

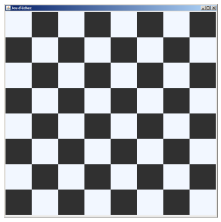
4 2ème itération : IHM en mode graphique

4.1 Travail préalable

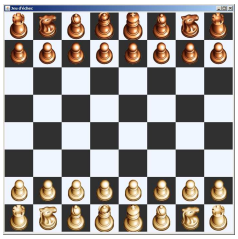
- Créez votre environnement de test en mode graphique avec la classe `LauncherGUI` (e-campus) qui lance l'application et la classe `ChessGameGUI` qui teste et affiche les résultats en mode graphique. Cette dernière doit étendre `JFrame` et implémenter les interfaces `MouseListener` et `MouseMotionListener`. **Vous allez donc coupler une nouvelle vue (`ChessGameGUI`) avec le même contrôleur (`chessGameControler`) qui agit sur le même modèle (`ChessGame` qui sert de facade 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 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`). Ses méthodes vous serviront 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.

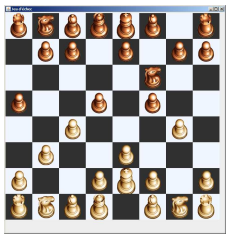


- Programmez les déplacements et testez :
  - Lorsque l'utilisateur sélectionne une image de pièce pour la déplacer, le programme doit vérifier si le déplacement est légal en interrogeant le `chessGameControler`, sinon, l'image de la pièce est repositionnée à sa position initiale.
  - Selon votre niveau et/ou si vous souhaitez mettre en place les sockets pour la 3ème itération, vous pouvez/devez mettre en œuvre le pattern MVC. En effet, votre modèle est observable (classe `ChessGame`) et peut être observé par votre vue (classe `ChessGameGUI`). Dans ce cas votre vue doit implémenter l'interface `Observer` et sera munie d'une méthode `update()` qui aura la responsabilité de rafraichir l'affichage après un déplacement. N'oubliez pas de dire au modèle qu'il est observé par la vue dans le `LauncherGUI` :

```
chessGame.addObserver((Observer) frame ;
```

- Au fur et à mesure des déplacements, les messages apparaissent dans la console.

```
OK : déplacement simple
KO : c'est au tour de l'autre joueur
OK : déplacement simple
OK : déplacement simple
KO : la position finale ne correspond pas à algo
de déplacement légal de la pièce
OK : déplacement  + capture
```



5 3ème itération : au choix

5.1 Améliorez vos classes « métier »

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

5.2 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 et son propre `Echiquier` (la classe « métier ») doit être mis à jour (`1 Pieces` a changé de coordonnées et 1 autre a peut être été prise).

La conception respectée jusqu'à maintenant fait qu'il n'y aura aucun changement a effectuer sur aucune classe métier, ni sur la vue (sous réserve de bien avoir une méthode `update`). 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 existera 1 instance de `chessGameControler` coté Server (à lancer en 1er) et 1 coté Client (à lancer après). Des « threads » permettant de gérer respectivement l'envoi et la réception de données permettront une communication non bloquante entre les 2 programmes.

5.2.1 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.

  Le poly suivant est également assez clair (avant ou après ou à la place du tuto d'OpenClassrooms) http://users.polytech.unice.fr/~karima/teaching/courses/I6/cours/module_I6_Sockets.pdf

- 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.

5.2.2 Mise en œuvre dans votre projet

- Transformer les classes de l'exemple d'OpenClassrooms (e-campus) de manière à sérialiser des `Object` et non plus des `String`.
- Intégrez la gestion de la communication à votre projet. La nouvelle classe `chessGameControler` (à écrire…) s'appuie sur les classes précédemment 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).