Conception Orientée Objet et Design Patterns 2

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).
- Inversion de dépendance : une classe doit dépendre d'abstraction et non pas de classes concrètes.
- Ségrégation des interfaces : toute classe implémentant une interface doit implémenter chacune de ces fonctions.

1.2. Objectif pédagogique

L'objectif pédagogique de ce projet est de mettre en œuvre, conjointement dans une même application, plusieurs Design Patterns en appliquant les bonnes pratiques de conception objet :

- Encapsuler ce qui varie.
- Préférer la composition à l'héritage (au sens délégation).
- Programmer des interfaces (au sens abstraction) et non des implémentations :
    - En Java une abstraction est soit une interface soit une classe abstraite.
    - En Java, une implémentation est une classe instanciable.
- Coupler faiblement les objets qui interagissent.

1.3. Moyens

A partir des classes « métier » d'un jeu d'échec (fournies), et dans le respect du pattern MVC, il s'agit de :

1. Programmer et tester les déplacements simples en mode graphique évènementiel : 1 seul damier et 2 joueurs sur le même damier.
2. Permettre à 2 joueurs de jouer ensemble sur des postes distants (dans un 1er temps 1 damier pour chaque joueur sur le même poste puis 1 damier sur chaque poste).
3. Pour les plus avancés, améliorer l'algorithme de déplacement des pièces pour gérer le roque du roi, être capable de dire si le roi est en échec, en échec et mat, etc.

Un ensemble de interfaces/classes est fourni (e-campus).

1.4. 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. Modalités pratiques

2.1. Organisation

- Un travail original (entendre individuel…) est attendu de chaque binôme. Il n'est cependant pas interdit de travailler en collaboration avec d'autres binômes, le préciser dans ce cas mais il est interdit de s'échanger du code. Il convient également de citer ses sources (Internet ou autre).
- L'objectif minimum à atteindre pour les moins rapides est la fin de l'itération 2.

2.2. Suivi

- Les séances sont encadrées par 2 enseignants.
- Les modèles de conception (de chaque étape) doivent être validés par les enseignants avant de commencer le codage (sauf en cas d'attente trop longue ☺).
- Chaque membre du binôme sera invité à s'exprimer et à justifier les choix de conception et de programmation.

2.3. Rendus

- Chaque fin d'étape (théorique) fera l'objet d'un livrable : diagramme UML (éventuellement sous forme papier) et 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.

2.4. Évaluation : Projet 100%.

- Dans la note seront appréciés :
    - La maitrise (conceptuelle) des patterns mis en œuvre volontairement ou utilisés car natifs dans le JDK.
    - L'aisance en Java.
    - La cohérence des conceptions avec les réalisations (à chaque étape).
    - La pertinence des réponses aux questions posées (à l'oral en cours de séance).
    - L'exhaustivité des procédures de test.
    - L'originalité de la solution finale.
    - Le tout étant pondéré par la progression des compétences de chacun.
- Une note individuelle pourra être donnée aux différents membres d'un même binôme en cas de différence de niveau flagrante.

3. Conception

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).

3.1. 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 éventuellement les interfaces existantes et/ou de créer de nouvelles interfaces.

3.2. 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.

3.3. Classe `Jeu`

La classe `Jeu` stocke ses pièces dans une liste de `Pieces`. Elle fait appel à une fabrique pour créer les pièces à leurs coordonnées initiales.

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, capturer une piece, etc.

3.4. Classe `Echiquier`

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

Elle est munie de méthodes qui permettent de vérifier s'il est possible de déplacer une pièce depuis ses coordonnées initiales vers ses coordonnées finales, puis de rendre effectif le déplacement avec prise éventuelle.

Le déplacement est possible 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 conforme à celui attendu par le type de pièce concerné, indépendamment des autres pièces.
- Le déplacement est possible par rapport aux autres pièces qui sont sur la trajectoire avec prise éventuelle.

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

4. 1ère itération : IHM en mode graphique

4.1. Testez les classes « métier »

- Créez votre projet Java avec l'IDE de votre choix (Eclipse ou autre) puis intégrez les interfaces/classes du fichier zippé « 5ETI DP Fichiers pour Projet 1516 » (e-campus) en les mettant dans les bons packages.
- A la lumière de la conception (§3) et des fichiers sources fournis, appropriez-vous l'organisation des classes en dessinant le diagramme de classe UML (sans détail à l'intérieur).
- Etudiez en détail les classes `Echiquier`, `Jeu`, `ChessPieceFactory`, `AbstractPiece et dérivées`.
- Si besoin, enrichissez le pgm de test (`main()`) pour tester les méthodes de la classe `Echiquier`.
- Quel pattern met en œuvre la classe `Echiquier` ?

4.2. Mettez en place le pattern d'architecture MVC

On postule que les IHM (vues console fournie et vue graphique à écrire) vont manipuler pour gérer les déplacements 2 objets `Coord` (x, y) et non pas 4 `int`, contrairement à la classe `Echiquier`.

Un 1er contrôleur va servir d'intermédiaire entre la vue et le modèle en transformant les messages venant de la vue pour qu'ils soient compréhensibles par le modèle. La vue ne communiquera donc qu'avec le contrôleur. Un 2ème contrôleur sera nécessaire pour la 2ème itération (jeu sur des postes distants).

Par ailleurs, on va rendre le modèle observable (classe `ChessGame`) pour qu'il soit observé par votre vue graphique. Cette dernière 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 :

- Complétez votre diagramme de classe en ce sens (1 contrôleur + 1 transformation du modèle pour le rendre observable + 2 vues).
- Programmez la classe `ChessGame` avec, a minima, l'interface précisée en annexe. Elle réduit légèrement l'interface de la classe `Echiquier` et surtout le rend "observable". Quelques points de détail :
  - Son constructeur crée l'objet `Echiquier`.
  - sa méthode `move()` :
    - vérifie si le déplacement est possible
    - effectue le déplacement s'il est possible
    - effectue aussi le changement de joueur si le déplacement est OK ;
  - sa méthode `toString()` retourne la représentation de l'`Echiquier` plus le message relatif aux déplacements/capture du type `« Déplacement de 3,6 vers 3,4 : OK : déplacement simple »`.
  - n'oubliez pas de la rendre « observable » par le moyen de votre choix. Réfléchissez aux avantages et inconvénients d'utiliser la classe `java.util.Observable`.
  - Pensez « réutilisation » et utilisez les bons patterns.
- Programmez un contrôleur `chessGameControler` qui implémente l'interface `ChessGameControlers` (Cf. annexe).
- Testez en utilisant la classe `LauncherCmdLine` qui lance l'application en créant un objet de la classe `ChessGameCmdLine` qui affiche les résultats en mode console (ne teste pas l'aspect « Observable »).
- Quels patterns mettent en œuvre chacune de ces nouvelles classes ?

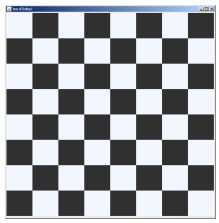
4.3. Programmez votre IHM graphique

- 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.

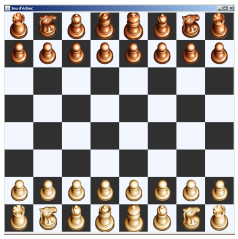
  Enregistrez le code dans un fichier de test, mettez à jour les noms des images avec ceux de quelques images récupérées du e-campus et testez. Vous constatez que vous pouvez déplacer des images de pièces, pour autant, vous n'avez pas de logique « métier » derrière pour vérifier la « légalité » de vos 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`). Sa méthode vous servira pour créer les images des pièces sur votre damier.
- Créez votre environnement de test en mode graphique avec une classe `LauncherGUI` qui lance l'application et une classe `ChessGameGUI` qui teste et affiche les résultats en mode graphique.
  Cette dernière doit étendre `JFrame` et implémenter les interfaces `MouseListener`, `MouseMotionListener` et `java.util.Observer` (ou autre interface que vous créerez lui permettant d'être un « observateur »).

  Comme elle observe votre modèle (classe `ChessGame`) elle doit donc être munie d'une méthode `update()` qui a la responsabilité de rafraichir l'affichage après un déplacement.
  Elle est couplée avec le même contrôleur (`chessGameControler`) que la vue console qui agit sur le même modèle (`ChessGame`).

  N'oubliez pas de faire écouter votre modèle par votre vue dans le launcher.
- 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.

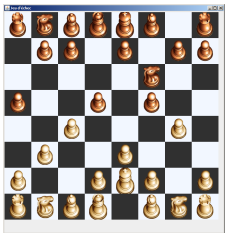


- Affichez les pièces à leur position initiale en vous servant de la méthode de la classe `ChessImageProvider` et testez. Pour ce faire, pensez éventuellement à exploiter le côté « observable » du modèle.



- 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 apparaissent dans la console ou dans une popup.

```
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. 2ème itération : jeu à 2 sur 2 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 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 (rappelez vous le pattern illustré par ce controleur…).

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

5.5.1. 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.

  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.

5.5.2. 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 parser 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).

6. 3ème itération : Améliorez vos classes « métier » ou d'IHM

- Prenez soin de les sauvegarder 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 test du déplacement en prévoyant la promotion du pion, le roque du roi.
  - 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.).

7. Annexes

- Méthodes minimales attendues dans la classe `ChessGame` :

```
public String toString();
public boolean move (int xInit, int yInit, int xFinal, int yFinal);
public boolean isEnd();
public String getMessage();
public Couleur getColorCurrentPlayer();
```

- Méthodes minimales attendues dans l'interface `ChessGameControlers` :

```
public boolean move(Coord initCoord, Coord finalCoord);
public String getMessage();
public boolean isEnd();
public Couleur getColorCurrentPlayer();
```