COO et Design Patterns
Exercices

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.

Objectif pédagogique

L'objectif pédagogique 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.

Moyens

- Un gros projet dont une base est donnée, à compléter pas à pas.
- Quelques exercices hors projet.
- Des sources permettant d'illustrer les patterns à mettre en œuvre dans le projet.

Conception

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

- Des pièces : roi, reine, fous, pions, cavaliers, tours. Il existe 16 pièces blanches et 16 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).

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.

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.

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.

Classe Echiquier

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

Elle est munie de méthodes qui vérifient 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 celui attendu par le type de pièce, 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…

1 DP Factory Method + introspection de classe

Le pattern Factory Method introduit une méthode abstraite de création d'un ou plusieurs objets qui sont ensuite utilisés par les instances de cette classe. Cette méthode est rendue concrète dans les sous-classes permettant ainsi de créer des objets de types différents.

Ce pattern est illustré dans l'exemple du répertoire « courrier ».

Il existe des classes dérivées pour chaque type de courrier. Le programme de test doit donc connaitre cette hiérarchie de classes.

Le but de l'exercice est de transformer la classe `Courrier` de manière à ce qu'elle soit générique et qu'il ne soit plus nécessaire de la dériver en `CourrierTexte` et `CourrierHtml`. Elle prendra en paramètre un type générique qui sera une implémentation de l'interface `Contenu` : `Courrier<T extends Contenu>`. Ce type générique sera utilisé pour typer son attribut `contenu`.

Le programme de test ne devra plus connaitre la hiérarchie de sous-classes de `Courrier` mais en revanche, il devra connaitre les classes utilisées dans sa composition.

- Transformez la classe Courrier en ce sens.
- Transformez la fonction main() en conséquence.

Trucs et Astuces

- Voir site http://www.jmdoudoux.fr/java/dej/chap-introspection.htm
- Une instance de classe générique se crée ainsi :
  `NomClasse<NomType> nomObjet = new NomClasse<NomType>(liste des arguments);`
  Il peut être pratique qu'un des arguments soit une instance de la classe `Class` correspondant au `NomType`…
- On obtient un objet de type Class<T> de 2 manières :
  `Class<T> c = NomType.class` ou bien
  `Class<T> c = Class.forName("NomType");`
- On obtient dynamiquement une instance d'une classe en invoquant son constructeur par défaut :
  `c.newInstance();`

2 Testez les classes « métier » du projet « Jeu d'échec »

- Créez votre projet Java avec l'IDE de votre choix (Eclipse ou autre) puis intégrez les interfaces/classes du fichier zippé « 4IRC DP Fichiers pour Projet 1516 » (e-campus) en les mettant dans les bons packages (copiez tous les fichiers dans le répertoire par défaut, créez les packages, déplacez les fichiers dans les bons packages : les liens seront MAJ automatiquement).
- A la lumière de la conception et des fichiers sources fournis, appropriez-vous l'organisation des classes métier en dessinant le diagramme de classe UML (sans détail à l'intérieur).
- Etudiez en détail les classes `Echiquier`, `Jeu`, `ChessPieceFactory`, `ChessPiecePos`, `AbstractPiece` et dérivées. Etudiez rapidement les classes utilitaires (`package tools`)
- Si besoin, enrichissez le pgm de test pour tester les méthodes de la classe `Echiquier` : ne vous servez pas tout de suite du lanceur de l'application (classe `LauncherCmdLine`) mais de la fonction `main()` de la classe `Echiquier`.

3 DP Template Method

Vous pouvez constater que dans la classe `Pion`, plusieurs méthodes ont des comportements différents en fonction de la couleur du pion. Ce n'est pas très objet : normalement un `Pion` « sait » qu'il est un pion blanc ou noir et agit en conséquence ; il ne doit pas avoir à « se demander » s'il est un pion blanc pour décider de son comportement.

- Remplacez la classe `Pion` par une classe abstraite et 2 classes dérivées. Mettez en œuvre le pattern « Template Method » lorsque cela vous parait opportun.
- Testez en commentant/dé-commentant les instructions nécessaires dans la classe `ChessPiecePos`.

4 DP Decorator : décorateur de JComponent

On se propose d'enrichir l'exemple des décorateurs de `JComponent` disponible sur le e-campus.

- Programmez en Java une classe `EnabledDecorator` qui permette lorsqu'un utilisateur clique sur un composant de le rendre « disable » et lorsqu'il clique sur un autre composant du même type de rendre le précédent « enable ».
- Testez en dé-commentant les instructions correspondantes dans le pgm de test.

5 DP Facade + Bridge

Observez votre diagramme UML.

- Quels patterns sont mis en œuvre par les classes et interfaces `Echiquier`, `Jeu`, `BoardGames` et `Games`?

6 DP Adapter

Différentes IHM (console ou graphique) peuvent être amenées, éventuellement à travers un contrôleur, à invoquer des méthodes des classes métier et à exploiter/afficher leur état. Pour autant, il serait inopportun que les IHM manipulent directement des `Pieces` puisqu'elles pourraient alors invoquer leur méthodes de MAJ (`move()`, `capture()`), et ce, sans aucun contrôle par l'`Echiquier`. Il faut néanmoins trouver un moyen pour l'`Echiquier` qui sert de Facade aux autres classes, d'envoyer des informations exploitables par les IHM.

- Créez une classe `PieceIHM` qui soit un adaptateur de `Pieces` muni uniquement d'accesseurs (`getXXX()`) et d'une méthode `toString()` pour tester.
- Enrichissez les classes `Jeu` et `Echiquier` avec une méthode `getPiecesIHM()` qui retourne une `List<PieceIHM>` et testez.

7 DP Proxy

Pour faciliter l'utilisation de la classe `Echiquier`, on va simplifier son interface et y accéder à travers un proxy (classe `ChessGame`) qui devra a minima être muni des méthodes suivantes :

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

- Complétez votre diagramme UML.
- Pour programmer la classe `ChessGame`, 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 ;

o sa méthode `toString()` retourne la représentation de l'`Echiquier` plus le message relatif aux déplacements/capture du type « `OK : déplacement sans capture` ».

8 Mettez en place le pattern d'architecture MVC

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 que nous étudierons en détail plus tard :

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 (ou les) vue(s) 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`) 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 :

- Complétez votre diagramme de classe en ce sens (1 contrôleur `chessGameControler` qui implémente `chessGameControlers` + 2 vues = 1 vue console `ChessGameCmdLine` et une vue graphique `ChessGameGUI`).
- Programmez un contrôleur `chessGameControler` qui implémente l'interface `ChessGameControlers` (e-campus) afin qu'il transforme les messages venant des vues pour qu'ils soient compréhensibles par le modèle.
- 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 » pour l'instant).

9 DP Observer

- Complétez la classe `ChessGame` pour la rendre « observable ».
  - Réfléchissez aux avantages et inconvénients d'utiliser la classe `java.util.Observable` mais ne l'utilisez pas et créez **votre propre** mécanisme d'observation (vos propres classes/interfaces).
  - Prévoyez lors d'une notification aux observateurs de leur envoyer une `List<PieceIHM>` par invocation de la méthode `getPiecesIHM()` de la classe `Echiquier`.

- Complétez la classe `ChessGameCmdLine` de manière à la transformer en observateur et programmez sa méthode `update()` de manière à ce que le résultat après l'appel par le lanceur `chessGameControler.move(new Coord(3,6), new Coord(3, 4))` ressemble à la trace suivante :

```
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
```

10 DP Iterator

- Modifiez le parcours de liste codé précédemment (méthode `update()`) de manière à utiliser explicitement un itérateur (pas de boucle forEach). C'est très simple, on ne vous demande pas de ré-écrire l'itérateur de la classe `LinkedList` mais de l'utiliser ☺.

11 DP Composite + Observer

- Créez votre environnement de test en mode graphique avec la classe `LauncherGUI` (e-campus) qui lance l'application et la classe `ChessGameGUI` (à créer) qui teste et affiche les résultats en mode graphique. Cette dernière doit étendre `JFrame` et implémenter les interfaces `MouseListener`, `MouseMotionListener` plus l'interface que vous avez définie pour assurer le job d'observateur. 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 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.
- Repartez du code réalisé en projet l'an dernier pour la classe `ChessGameGUI` ou repartez de 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.

- L'objectif est de vous servir de la puissance du pattern Observer pour rafraichir l'affichage après un déplacement, une promotion du pion, etc.
    - Votre vue (classe `ChessGameGUI`) observe votre modèle (classe `ChessGame`) et doit être munie d'une méthode `update()`.
    - Lorsque l'utilisateur sélectionne une image de pièce pour la déplacer, le programme doit vérifier si le déplacement est légal en interrogeant le `chessGameControler`.
    - 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
```

- Dans la classe `chessGameControler` définissez une méthode `isPlayerOK()` et invoquez là correctement dans la vue pour empêcher de déplacer une pièce si ce n'est pas le tour du joueur.
- Ajoutez un affichage des cases accessibles lors de la sélection d'une pièce sur le damier. Réfléchissez bien à la distribution des rôles entre les différentes classes (impact sur toutes les couches M V C).

12 DP Strategy + Singleton

L'objectif est maintenant de programmer une version très simplifiée de « Tempête sur l'Echiquier » (https://fr.wikipedia.org/wiki/Temp%C3%AAte_sur_l'%C3%A9chiquier).

Une 1ère variante consiste à considérer qu'une pièce n'aura plus toujours son algorithme de déplacement propre mais aura un algorithme de déplacement différent selon sa position sur le damier. Elle pourra garder son image, mais n'aura plus toujours le même comportement.

Son comportement sera défini par colonne :

- Toutes les pièces positionnées sur les colonnes 0 et 7 auront un comportement de Tour.
- Toutes les pièces positionnées sur les colonnes 1 et 6 auront un comportement de Cavalier.
- Toutes les pièces positionnées sur les colonnes 2 et 5 auront un comportement de Fou.
- Toutes les pièces positionnées sur les colonnes 3 et 4 auront leur propre comportement.

Dans une 2ème variante, on pourrait imaginer que certaines pièces ne sont plus supprimées du jeu lorsqu'elles sont prises mais retournent à leur position initiale si elle n'est pas occupée. Nous ne programmerons pas les détails de cette variante, mais ferons comme si elle existait.

Etc.

Pour réaliser cette nouvelle version, il s'agit donc de définir les différents comportements (`isMoveOk()`) dans autant de Strategy différentes. Vous testerez dans un 1er temps une version qui utilise les stratégies mais pour laquelle le comportement de vos pièces sera le même que dans la version sauvegardée (ce qui vous permettra de comparer les 2). Puis dans un 2ème temps, vous commenterez la 1ère version et testerez la tempête.

- Commencez par créer un nouveau projet avec toutes vos classes existantes, étant bien entendu que notre conception étant bien faite, les seuls impacts sont sur le modèle et plus précisément sur les `Pieces` et dérivées et sur la façon de les fabriquer.
- Créez les différentes stratégies qui permettront de répondre à la question `isMoveOk()` (faites en des Singleton) : 1 strategy par type de pièce (tour, etc.). N'hésitez pas à adapter le mécanisme de Template Method de l'exercice 3 pour la stratégie de déplacement des pions.
- Réorganisez (créez, modifiez, supprimez) la hiérarchie de classes qui implémentent l'interface `Pieces`. Les autres classes du modèle (`Jeu, Echiquier, ChessGame`) ne doivent pas être modifiées.
    - Pensez Introspection de classe ou sinon, préférez utiliser une `Map` à une succession de 'if' pour créer les différentes stratégies de déplacement.
    - Eventuellement créez une fabrique simple de stratégies de déplacement.
- Modifiez les fabriques de pièces en conséquence (`ChessPiecesFactory` et `ChessSinglePieceFactory`).
- Testez le comportement usuel puis le comportement « Tempête ».

13 DP Abstract Factory

Dans l'exercice précédent, vous avez modifié la méthode `isMoveOk()` des pièces avec une version en mode normal puis une version en utilisant la variante « Tempête ».

Il s'agit maintenant de découpler les pièces des styles de jeu (normal ou tempête), afin que la méthode `isMoveOk()` des pièces ait toujours le même comportement.

- Commencez par créer un nouveau projet avec toutes vos classes existantes.
- Etudiez les classes du répertoire `fabriqueAnimaux` et dessinez le diagramme UML correspondant en indiquant aussi les noms de méthodes dans les classes/interfaces.

- Dans notre projet, en style « normal » on souhaite être capable de fabriquer pour 1 pièce sa Strategy de déplacement (en fonction de son type – Tour, etc.) et sa Strategy de capture (ne pas le coder) et en mode « Tempête » on veut faire la même chose, sauf que la strategy de déplacement agit différemment (en fonction de son type et/ou de sa position) :
    - Trouvez l'analogie entre ce diagramme et votre projet, i.e. identifiez bien quelle classe/interface représentera AbstractctFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Client et lanceur (main()) et dessinez/complétez le diagramme UML de votre application.
    - Faites le valider avant de commencer à coder.
- Programmez et testez.
    - Faites apparaitre une popup pour choisir le style de jeu (normal ou tempête).
    - Normalement, le constructeur de toutes vos classes métier doit changer puisqu'il faut lui passer en paramètre la bonne fabrique.
    - Pensez Introspection de classe ou sinon, préférez utiliser une `Map` à une succession de 'if' pour créer les différentes stratégies de déplacement.

14 DP Command

Toujours dans le cadre de « Tempête sur l'échiquier », on imagine que l'utilisateur peut s'amuser à annuler les coups (cela rembobine l'action), une ou plusieurs fois de suite, et bien sûr peut vouloir les rejouer.

- Commencez par créer un nouveau projet avec toutes vos classes existantes.
- Propagez une méthode `undoMove()` de votre modèle vers votre contrôleur de manière à ce qu'elle puisse être invoquée par votre vue. Normalement, de nombreux attributs de classe ont été initialisés dans la classe Jeu pour que ce soit très vite fait… n'oubliez pas cependant que quand il y a déplacement, il peut y avoir aussi capture. Attention, ils ne permettent pas de gérer l'historique et vous ne pourrez pas enchainer plusieurs « undo/redo » de suite.
- Testez-là rapidement en programmant l'envoi d'une popup lors du clic droit de la souris qui proposerait le choix entre « Undo et Redo » ou bien avec des boutons dans une ToolBar, ou avec une combinaison de touches (CTRL+z, CTRL+y) - ne perdez pas de temps avec l'IHM, faites au plus rapide.
- Etudiez le DP Command sur le site de http://zenika.developpez.com/tutoriels/java/patterns-command/ (copie sources sur e-campus).
- Faites l'analogie avec votre besoin, enrichissez votre diagramme UML en ce sens.
- Dans une version simple, i.e. sans n'autoriser l'action que dans le style de jeu « Tempête », programmez et testez les classes/interfaces nécessaires, avec le « exec, undo, redo ».

15 Améliorez vos classes « métier »

- Commencez par créer un nouveau projet avec toutes vos classes existantes.
- 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 éventuels impacts sur l'IHM (promotion, etc.).
- Vous pouvez :
    - Annuler le déplacement si le roi est en échec (ça tombe bien, le undoMove/Capture est déjà codé…), arrêter la partie lorsqu'elle est gagnée (échec et mat) ou lorsqu'elle est nulle, etc. Appréciez dans ce contexte l'intérêt du pattern Bridge entre les classes `Echiquier` et `Jeu`.
    - Envisager la promotion du pion.
    - Programmer le roque du roi : les booléens dispersés à travers le code laissent supposer qu'à travers l'IHM, l'utilisateur a sélectionné le roi et l'a positionné sur la tour pour indiquer qu'il souhaite « roquer ». Pour déplacer le roi et la tour à leurs bonnes coordonnées, rappelez-vous de l'existence du pattern Bridge entre `Echiquier` et `Jeu`…