**Correction DS POO avancée 4ETI mars 2016**

# 1. Questions / réponses rapides (6 points)

## Q1 – isAllowed[A|B|C](…) ? (1,5 point)

Réponse attendue : isAllowedC(…) doit seule être utilisée

Réponses acceptées et explications : idem infra

isAllowedA(…) : NON ! On continue à itérer même si la valeur est trouvée, donc la méthode ne renverra true que si le dernier élément de la liste correspond à la valeur passée en argument.

Nota : Attention, return est une instruction de contrôle du flux d'exécution. Si le sortir d'un simple if… else… ne nécessite pas toujours davantage de précautions, il faut néanmoins être très prudent, en particulier avec les boucles, switch, etc...

isAllowedB(…) : oui (noter le break qui nous évite le problème précédent), MAIS uniquement avec ce main(…) dans lequel la liste de reference (pluralWithX) ne contient aucune valeur null ; en effet, s.equals(str) nous expose à une NullPointerException si s est null.

(Nota : style très C, mais si on aime...)

isAllowedC(…) : OUI ! ; n'expose à aucune exception quelles que soient les données.

(Nota : et qu'est-ce que c'est sobre ! :-) C'est tellement simple qu'il ne serait même pas nécessaire de déporter ce test dans une méthode spécifique…)

Nota : contains() se base sur equals(), et non sur ==. Les refus de cette méthode explicitement basés sur la crainte d'une comparaison des références et non des valeurs n'ont pas été pénalisés.

## Q2 – (s == str) ? (1 point)

Réponse attendue : NON, l'opérateur == compare les références et non les valeurs.

## Q3 – (str.equals(s)) ? (1 point)

Réponse attendue : NON (avec ce main(…)), provoquera une NullPointerException avec la valeur null de la liste à épurer, dont est issue str.

Réponse parfaitement acceptable : OUI dans l'absolu (si on fait abstraction des valeurs du main(…)) : dans les deux cas, rien ne nous protégeant de valeurs null dans les données, les deux écritures se valent : souffrent du même problème et sont fonctionnellement identiques.

## Q4 – Instruction A|B|C ? (1,5 points)

Réponse attendue :

A : OUI (seule méthode « prudente » qui permette de supprimer l'objet courant, issu de l'itérateur, et dont la valeur vient d'être testée comme « non autorisée / absente de la liste de référence », de la liste sous-jacente à l'itérateur ; ce qui est souhaité)

B : NON (expose à une ConcurrentModificationException : il faut passer par un itérateur pour effectuer une suppression durant une itération)

C : NON (ne permet pas d'obtenir le résultat attendu : cette instruction ne modifie pas la liste candidates, mais uniquement la valeur de la variable locale s)

## Q5 – vals.size() == 5 ; pourquoi ? (1 point)

Réponse attendue :

La classe Grammaire nettoie une copie nommée candidates de la liste vals (instruction new ArrayList<…>(…) dans le constructeur). Aussi la liste d'origine n'est-elle pas modifiée.

Complément appréciable : le a contrario du précédent.

## QX – Bonus possibles (1 point)

- avoir relevé l'erreur suivante : l'attribut reference aurait dû être nommé refernces pour être cohérent avec le reste du code.

- avoir relevé que la JavaDoc du constructeur est incohérente avec le code (les nom des deux @param devraient être candidates et references).

- avoir considéré la possibilité d'une valeur nulle dans la liste de référence

- avoir osé dire que cette classe était quand même un peu bidon (+0,5 pour la pertinence, -0,5 pour l'impertinence, ça s'annule, caramba, encore raté!)

- avoir assorti le propos précédent de la référence à une méthode statique de la classe Collections qui faisait tout ça beaucoup mieux que nous (chapeau ! Là on ne garde plus que l'aspect pertinent)

# 2 – Questions de réflexion sur le projet (6 points)

## QA – Pourquoi coordonnees\_valides(…) est-elle déclarée static ? (1,5 points)

Elle ne dépend pas d'attributs ou de méthodes d'instance de la classe.

Quoiqu'il ne serait pas strictement erroné de la laisser au rang de méthode d'objet, c'est à considérer a minima comme inutile ; en faire une méthode de classe est même préférable en termes de performances, puisqu'elle est partagée entre les objets, et même ne nécessite aucun objet pour être appelée.

## QB – Pourquoi isMoveOk(…) est-elle déclarée abstract ? (1,5 points)

Les détails de son implémentation ne peuvent être définis (implémentés, codés) que dans les classes concrètes qui hériteront de la classe abstraite AbstractPiece : en effet, l'algorithme de déplacement des différentes pièces dépendra de leur type concret (effectif), et variera d'autant.

(Contrairement aux autres méthodes qui définissent des comportements communs à toutes les pièces)

## QC – ArrayList vs. LinkedList ? (1,5 points)

Le code de notre projet requiert des listes de pièces pour des parcours séquentiels (le code, fourni, de findPiece(…) en est une bonne illustration ; de même, on balaye via ces listes l'ensemble des pièces d'un jeu, comme par exemple lors des tests de mise en échec du Roi).

Par ailleurs, ces listes sont construites « une fois pour toutes » avec les 16 pièces d'un Jeu, sans qu'on y ajoute ni n'en soustraie jamais aucune pièce (la prise étant mémorisée par un pasage des coordonnées à -1 et non par une suppression de la liste).

Il n’est jamais utile d’accéder au ième élément, opération facilitée par une ArrayList.

Les deux implémentations de List considérées présentent des performances relativement équivalentes pour ce qui concerne les accès séquentiels, avec un léger bonus pour la LinkedList.

Les meilleures performances de la LinkedList pour les ajouts ou suppressions ne sont d'aucun usage ici.

Il aurait donc été pertinent d'utiliser également une ArrayList.

MAIS pour 16 pièces, il n'est pas nécessairement pertinent… de se poser la question :-)

## QD – Méthodes « vides » (1,5 points)

Toute classe implémentant une interface doit fournir l'implémentation de toutes les méthodes définie par celle-ci, cette implémentation consistât-elle à ne rien faire.

La classe ChessGameGUI, en tant qu'implémentation des interfaces MouseListener et MouseMotionListener, se doit de fournir des implémentations pour toutes les méthodes de ces deux interfaces, quitte à le faire sous forme de blocs vides de type { } pour les événements auxquels elle ne s'intéresse pas.

# 3. Maintenance évolutive du projet (8 points)

## QA – Intérêt de la classe PieceIHM (1 point)

Réponses indispensables :

- encapsulation : la vue n'a pas accès aux objets métiers (les objets implémentant Pieces), ne peut pas les manipuler directement.

- découplage entre la vue et les données : le code client est « protégé » de l'évolution de l'implémentation des objets métier, dont PieceIHM fait abstraction.

Pertinence :

- 2 vues (terminal et GUI/Swing)

Compléments souhaitables :

- participe au modèle MVC en fournissant un modèle de transmission des données entre Modèle et Vue

- découplage entre les données et la/les vues : le code métier n'a pour responsabilité que de fournir les données « métier », nécessaires et suffisantes indépendamment des spécificités de l'interface utilisateur (principe de partage des responsabilités)

## QB – Évolution de la classe PieceIHM (1 point)

public class PieceIHM {

// ...

PieceIHM (Pieces piece) {

this.type = piece.getType() ;

this.couleur = piece.getCouleur() ;

this.coord = new Coord( piece.getX(), piece.getY() ) ; // \*

}

// …

}

Il fallait ici (\*) faire attention à bien encapsuler les données, et à ne pas transmettre l'objet Coord lui-même (données métier modifiables), mais une copie de ses données.

Toutes variations correctes sont bien sûr acceptées si elles respectent l'encapsulation.

## QC – Écrire Jeu.getPiecesIHM(…) (1 point)

// …

class Jeu {

protected List<Pieces> pieces ; // Cf page 10 du sujet

// …

public List<PieceIHM> getPiecesIHM() {

List<PieceIHM> liste = new ArrayList<PieceIHM>() ; // \*

for (Pieces piece : pieces) {

liste.add(new PieceIHM(piece)) ;

}

return liste ;

}

// …

}

Le type concret de liste choisi ici (\*) ne nous importe pas dans la correction, tant qu'il s'agit bien d'une classe concrète.

Toutes variations correctes sont bien sûr acceptées.

## QD – Écrire Echiquier.toString() (5 points)

Exemple (non normatif) :

// …

class Echiquier {

// …

public String toString() {

String st = "";

// On récupère les PieceIHM des deux jeux

List<PieceIHM> pieces = jeuCourant.getPiecesIHM();

pieces.appendAll( jeuOppose.getPiecesIHM() );

// Par commodité, on construit une Map[\*] ayant pour clés les coord. des pièces

Map<Coord,PieceIHM> echiquierIHM = new HashMap<Coord,PieceIHM>();

// Remplissage de la map avec le contenu des deux jeux

for (PieceIHM piece : pieces) {

echiquierIHM.put(piece.getCoord(), piece);

}

// On utilise la convention suivante : indice -1 <=> ligne de numérotation ("titres")

for (int numLigne = -1 ; numLigne <= 7 ; numLigne++) { // Boucle "lignes"

st += (( numLigne == -1) ? " " : numLigne ); // Numéro de ligne

for (int numCol = 0 ; numCol <= 7 ; numCol++) { // Boucle "colonnes"

st += " "; // Marge gauche des cases

if ( numLigne == -1) { // Cases de N° de colonnes

st += " " + numCol + " "; // Numéro de colonne

} else { // Cases standard

// On cherche dans la Map s'il y a une pièce aux coord. de la case

PieceIHM p = echiquierIHM.get( new Coord(numLigne, numCol) );

if (p == null) { // Case sans pièce

st += "\_\_\_\_";

} else { // Case avec pièce

st += p.getCouleur().name().substring(0,1) // Couleur

+ "\_"

+ p.getTypePiece().substring(0,2); // Type (2char)

}

} // endif numLigne < 0

} // end for numCol

st += "\n"; // Retour à la ligne

// FIXME : ce code n'est pas indépendant de l'OS…

} // end for numLigne

return st;

}

// ...

}

// [\*] : On aurait aussi pu utiliser une tableau bi-dimensionnel 8x8, ou autre toute autre solution…

// Ceci n'est pas indispensable, mais nous permet de ne parcourir notre liste qu'une seule fois,

// et pas 64 fois.