

Programmation Orientée Objets

(Pseudo-)Examen L3 2006/2007

Avant-propos

Règles du jeu

Vous avez 1h30 pour récolter autant de points que possible. Pour obtenir la note maximale, il vous suffit de 100 points. L'énoncé vous permet théoriquement d'obtenir environ 180 points, ce qui vous permet de choisir les questions auxquelles vous préférez répondre. Profitez-en.

Tous les documents sont autorisés. Par contre, le règlement de l'université ainsi que la loi française vous interdisent de communiquer entre vous sous quelque forme que ce soit ou de copier les uns sur les autres. Les ordinateurs personnels sont interdits. L'ordinateur que vous employez en ce moment est déconnecté du réseau.

Rédaction

Il vous est demandé de commenter vos algorithmes, aussi bien sous la forme d'une documentation de maintenance que d'une documentation publique (JavaDoc) et d'explications à *rendre sur feuille*. Les explications doivent contenir notamment

- votre démarche
- une description de la liste des modifications que vous avez effectuées.

En particulier, si essayez de résoudre un exercice mais que vous oubliez de rédiger votre réponse sur papier, *la question ne sera ni corrigée, ni notée*. À l'inverse, un exercice dont la réponse est rédigée mais dont le programme ne fonctionne pas recevra une partie des points. Enfin, si vous décidez de résoudre un exercice ou une partie d'un problème d'une manière différente de ce qui est indiqué dans l'énoncé, précisez-le et justifiez votre choix.

Conseils

N'essayez pas de résoudre plusieurs exercices en une seule fois. C'est une bonne manière de se planter. Si vous n'arrivez pas à terminer un exercice, *avant de passer à un autre exercice*, mettez en commentaires ce que vous auriez voulu faire (aussi bien le code source qui ne fonctionne pas que vos remarques en français) et revenez à un état dans lequel votre programme compile et s'exécute. Sans cela, vous risquez de contaminer votre exercice suivant avec vos difficultés.

Si vous ne trouvez pas un identificateur

Dans tout cet examen, les noms des classes, méthodes et champs existants sont tirés de la version de référence de l'enseignant. Vous êtes tout à fait autorisés à avoir des noms différents pour vos classes, méthodes et champs, mais vous pourrez avoir besoin de regarder la version de l'enseignant de temps à autres, pour comprendre à quoi font référence certaines questions.

Si vous cherchez la documentation de Java, elle est disponible sur votre disque dur, dans le répertoire `C:\...`

Problème : Tremblements de Terre (environ 180 points)

L'objectif de ce problème est d'ajouter au démineur un phénomène inédit : le tremblement de Terre. Les tremblements de Terre sont déclenchés aléatoirement et, de temps à autres, déplacent aléatoirement toutes les mines qui ne sont pas solidement attachées sous un drapeau.

Exercice 1. (environ 20 points) Ajoutez à votre classe `Terrain` une méthode

```
/**
 * Remplace une case minée par une case non minée.
 *
 * Cette méthode remplace un objet a de la classe CaseMinee par un objet b
 * de la classe CaseNonMinee autrement identique, c'est-à-dire tel que
 *   a.isExploree() == b.isExploree()
 *   a.isMarqueeDUndrapeau() == b.isMarqueeDUndrapeau()
 *
 * @param x
 * @param y Coordonnées d'une case. Si la case n'est pas minée, cette
 * méthode n'a aucun effet.
 *
 * @return true Si la case était minée
 * @throws ArrayIndexOutOfBoundsException si (x,y) ne sont pas les
 * coordonnées d'une case.
 */
protected boolean enleverUneBombe(int x, int y)
```

Correction possible L'énoncé demande de remplacer une case de la classe `CaseMinee` par un objet de la classe `CaseNonMinee`. Cela ne pose pas de réelle difficultés, puisqu'il suffit de

- construire une nouvelle case non minée `b`
- si `cases[x][y].isExploree()` est vraie, faire en sorte que `b` soit explorée
- si `cases[x][y].isMarqueeDUnDrapeau()` est vraie, faire en sorte que `b` soit marquée d'un drapeau
- placer `b` dans `cases[x][y]`

Pour ce corrigé, nous allons prendre une liberté et employer la classe `CaseFactory` pour faire quelque chose d'un peu plus générique.

L'ensemble des modifications tient dans la méthode `enleverLaBombe` de la classe `Terrain`.

Exercice 2. (environ 20 points) À l'aide de la méthode `enleverUneBombe`, ajoutez à votre classe `Terrain` une méthode

```
/**
 * Remplace toutes les cases minées par des cases non minées.
 *
 * Cette méthode parcourt toutes les cases du terrain et enlève chaque
 * case minée, la remplaçant par une case non minée autrement identique.
 */
void enleverLesBombes()
```

Correction possible Cette méthode ne présente aucune subtilité : juste deux boucles imbriquées. L'ensemble des modifications tient dans la méthode `enleverLesBombes` de la classe `Terrain`.

Exercice 3. (environ 20 points) Ajoutez à votre classe `Terrain` une méthode

```
/**
```

```

* Remplace une case non minée par une case minée.
*
* Cette méthode remplace un objet a de la classe CaseNonMinee par un
* objet b de la classe CaseNonMinee autrement identique, c'est-à-dire
* tel que
*   a.isExploree()      == b.isExploree()
*   a.isMarqueeDUndrapeau() == b.isMarqueeDUndrapeau()
*
* @param x
* @param y Coordonnées d'une case. Si la case est minée, cette
* méthode n'a aucun effet.
*
* @return true Si la case n'était pas minée
* @throws ArrayIndexOutOfBoundsException si (x,y) ne sont pas les
* coordonnées d'une case.
*/
protected boolean mettreUneBombe(int x, int y)

```

Correction possible Cette méthode est exactement symétrique de `enleverUneBombe`. On pourrait d'ailleurs les combiner en une seule méthode.

Exercice 4. (environ 20 points) À l'aide de la méthode `mettreUneBombe`, ajoutez à votre classe `Terrain` une méthode

```

/**
* Place des bombes parmi les cases non visitées.
*
* Cette méthode choisit aléatoirement un certain nombre d'emplacements
* distincts de cases non minées et non visitées dans le terrain et
* remplace chacune de ces cases non minées par une case minée autrement
* identique.
*
* @param nombreDeBombes Le nombre de bombes à placer.
*
* @throws TerrainException S'il n'y a pas assez de cases non visitées pour
* placer toutes les bombes.
*/
protected void placerLesBombes(int nombreDeBombes) throws TerrainException

```

Correction possible Il y a plusieurs manières de placer toutes ces bombes. Dans tous les cas, il faut commencer par vérifier si `nombreDeBombes` est inférieur ou égal au nombre de cases non minées et non visitées du terrain.

À partir de là, on pourrait, par exemple, faire un tableau ou une liste de toutes les cases non minées et non visitées et tirer aléatoirement des cases de cette liste, à remplacer.

Il est plus simple, plus économe en mémoire vive – mais potentiellement plus lent – de s'inspirer du constructeur de la classe `Terrain` et de continuer à tirer des coordonnées aléatoirement parmi la liste de toutes les coordonnées, jusqu'à en avoir trouvé `nombreDeBombes` convenables.

L'ensemble des modifications au code tient dans la méthode `placerLesBombes` de la classe `Terrain`. Comme nous avons vérifié préalablement que `nombreDeBombes` est inférieur ou égal au nombre de cases disponibles, la boucle doit forcément finir par s'achever.

Exercice 5. (environ 30 points) À l'aide de toutes ces méthodes, ajoutez à votre classe `Terrain` une méthode

```

/**
* Déclenche un tremblement de Terre.
*
* Cette méthode redistribue aléatoirement les bombes entre les cases non
* visitées. Cette méthode ne modifie pas le nombre affiché sur les boutons
* qui représentent les cases visitées !

```

```
*/
public void faireTremblerLaTerre()
```

Vous pourrez vous inspirer du constructeur de la classe `Terrain` pour la (re)distribution aléatoire des bombes. Cette méthode est un peu plus compliquée.

Correction possible Il suffit de supprimer toutes bombes, à l'aide d'une variante de `enleverLesBombes` puis d'invoquer `placerLesBombes`. Pour simplifier les choses, l'idéal serait de modifier la méthode `enleverLesBombes` pour faire en sorte qu'elle renvoie le nombre de bombes enlevées. Malheureusement, nous ne pouvons pas nous permettre de faire cela car cela impliquerait de modifier le type de retour de `enleverLesBombes`. En particulier, si un projet plus complexe définissait une sous-classe de `Terrain` et remplaçait la méthode `enleverLesBombes`, modifier le type de retour de `Terrain.enleverLesBombes` casserait le fonctionnement de la sous-classe.

Nous allons donc recopier presque tel quel le contenu de `enleverLesBombes` dans notre méthode `faireTremblerLaTerre`. Il serait possible de recopier de même une version légèrement modifiée de `placerLesBombes` dans `faireTremblerLaTerre`, pour gagner un peu de temps, mais c'est plus risqué qu'utile. Nous allons donc réutiliser `placerLesBombes`.

Reste à régler le problème de l'exception `TerrainException`, que la méthode `placerLesBombes` peut lancer. Effectivement, dans des cas complexes, la méthode `placerLesBombes` peut lancer cette exception malgré nos vérifications préalables. Nous pourrions modifier la signature de la méthode `faireTremblerLaTerre` pour ajouter un `throws TerrainException`, mais cela serait en contradiction avec les spécifications. Nous avons donc choisi d'encapsuler l'exception dans une `RuntimeException` – qui n'a pas besoin d'être déclarée – et de la relancer sous cette forme.

L'ensemble des modifications du code tient alors dans la méthode `faireTremblerLaTerre` de la classe `Terrain`.

Exercice 6. (environ 10 points) Pourquoi faut-il probablement modifier le nombre affiché sur les boutons qui représentent les cases visitées ?

Correction possible Considérons la configuration suivante (en gris, les cases non explorées, B marque une bombe).

Avant le tremblement de Terre :

0		
0		
1	B	

Le tremblement de Terre peut placer la bombe dans n'importe quelle case grise. Notamment, on

peut obtenir, après le tremblement de Terre :

1	B	
0		
0		

En particulier, le nombre de bombes voisines a changé pour deux des cases explorées.

Exercice 7. (environ 15 points) À l'aide de la méthode précédente, ajoutez à votre classe `TerrainPanneau` une méthode

```
/**
 * Déclenche un tremblement de Terre.
 *
 * Cette méthode redistribue aléatoirement les bombes entre les cases non
 * visitées et modifie, si nécessaire, le nombre affiché sur chaque bouton
 * représentant une case visitée.
 */
public void faireTremblerLaTerreEtSAdapter()
```

Correction possible Il suffit d'invoquer la méthode précédente puis de parcourir toutes les cases dont le nombre de voisines minées est visible et de mettre à jour ce nombre. Pour parcourir toutes ces cases, nous avons ajouté une méthode `isExploree` à la classe `Terrain`, qui détermine si une case de coordonnées `x`, `y` a été explorée. Une fois cela fait, il nous suffit de deux boucles imbriquées balayant l'ensemble des coordonnées du terrain. Pour chaque coordonnée possible sur le terrain, nous vérifions à l'aide de `isExploree` si la case a été explorée. Le cas échéant, à l'aide de `getNombreDeBombesVoisines`, nous (re)calculons le nombre de cases minées autour de notre bombe. Il ne nous reste alors plus qu'à changer le texte affiché sur le bouton correspondant.

Les modifications tiennent dans la méthode `isExploree` (classe `Terrain`) et la méthode `faireTremblerLaTerre` (classe `TerrainPanneau`).

Exercice 8. (environ 20 points) Créez une nouvelle classe, que nous appellerons `Tectonique`, qui implante l'interface `java.awt.event.ActionListener`, dont le constructeur prend en argument un objet de la classe `TerrainPanneau` et dont la méthode `actionPerformed` invoque un tremblement de Terre sur cet objet.

Correction possible Pour faire trembler la Terre, il suffit d'utiliser la méthode précédente.

Exercice 9. (environ 25 points) Ajoutez à votre classe `TerrainPanneau` un objet de la classe `Tectonique`, que nous appellerons `tectonique`, et un objet de la classe `javax.swing.Timer`, que nous appellerons `chrono`. À l'aide de `chrono` et de `tectonique`, faites en sorte que le tremblement de Terre soit invoqué automatiquement une fois par minute à partir du début de la partie.