

M1 ALMA – Test Logiciel – CCE1

Durée : 1h20. Documents non autorisés.

???

Nom : _____

Prénom : _____

Numéro étudiant : _____

| | Points obtenus |
|------------|----------------|
| Exercice 1 | 4 |
| Exercice 2 | 1 |
| Exercice 3 | 1 |
| Exercice 4 | 1 |
| Exercice 5 | 2.5 |
| Exercice 6 | 4 |
| Exercice 7 | 4.5 |
| Exercice 8 | 2 |
| total | 20 |

Quelques informations :

- Le sujet comprend 8 exercices, pensez à parcourir le sujet rapidement avant de commencer.
- Le barème est donné à titre indicatif dans la marge droite, mais est susceptible de changer.
- Si la place manque dans une case, utilisez le verso de la feuille et indiquez le.
- La taille d'une boîte de réponse n'est pas représentative de la taille de la réponse attendue.

1 Questions de cours

Exercice 1 *QCM*

—/4

Pour chaque question, cochez toutes les bonnes réponses. Une question peut ne pas avoir de bonne réponse. Une bonne réponse vaut 0.5. Une mauvaise réponse vaut -0.25 . Ne pas mettre de réponse vaut 0.

(a) JUnit est capable de faire :

- ☐ du test unitaire
- ☐ du test système

(b) Un scénario de test pour un logiciel codé en Java est nécessairement codé en Java.

- ☐ vrai
- ☐ faux

(c) Le test logiciel est une approche :

- ☐ exhaustive
- ☐ partielle
- ☐ statique
- ☐ dynamique

(d) Débogage signifie :

- ☐ définir de nouveaux tests
- ☐ localiser et corriger le défaut qui cause une défaillance
- ☐ mesurer la qualité d'une suite de tests

(e) Une assertion permet de préparer l'exécution du programme à tester.

- ☐ vrai
- ☐ faux

Exercice 2

—/1

En quoi le test unitaire facilite-t-il la localisation des défauts ?

Exercice 3

—/1

Nommez les trois sorties possibles lors de l'exécution d'un scénario de test, et expliquez dans quelles situations elles se produisent.

Exercice 4

—/1

Expliquez la différence entre la vérification et la validation

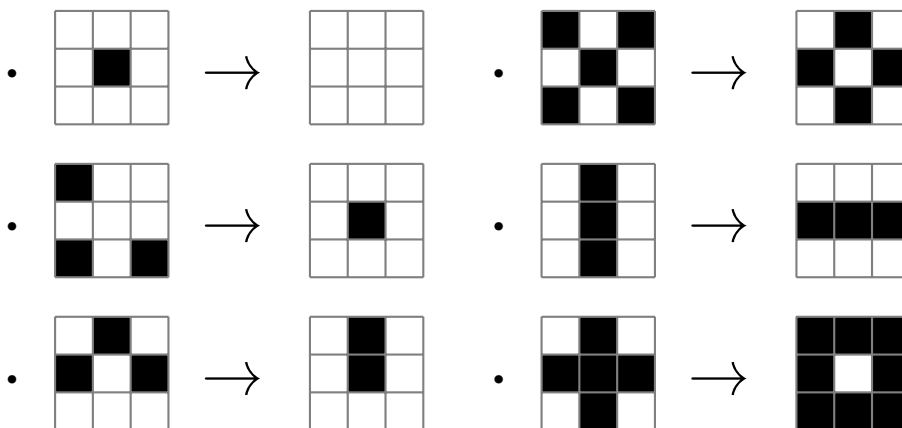
2 Méthode fonctionnelle

On souhaite tester une implémentation du *jeu de la vie*. Le jeu de la vie est un automate cellulaire imaginé par John Horton Conway en 1970. C'est un jeu de simulation au sens mathématique plutôt que ludique, car il n'y a pas vraiment de joueur.

La spécification du jeu est la suivante :

- Le jeu se déroule sur une grille à deux dimensions dont les cases — qu'on appelle des « cellules », par analogie avec les cellules vivantes — peuvent prendre deux états distincts : « vivante » ou « morte ».
- Une cellule possède de trois à huit voisins, qui sont les cellules adjacentes horizontalement, verticalement et diagonalement.
- Une exécution du jeu consiste en une séquence d'*étapes*, chaque étape donnant lieu à une nouvelle grille. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisins de la façon suivante :
 - une cellule morte possédant exactement trois voisines vivantes devient vivante ;
 - une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Quelques exemples d'étapes :



Le logiciel à tester repose notamment sur l'interface Java `Cellule` fournie en Figure 1. Cette interface est utilisée pour représenter une unique cellule de la grille. On souhaite tester la méthode `evolution`, et par conséquent on suppose que les autres opérations (`estVivante` et `getCellulesAdjacentes`) n'ont aucun défaut et se comportent exactement comme attendu.

Exercice 5

____/2.5

Identifiez toutes les données d'entrée et toutes les données de sortie de la méthode `computeNextStateOfCell`. Précisez le domaine de chaque entrée et sortie.

Exercice 6

____/4

Proposez un arbre de classification pour modéliser le domaine d'entrée de la méthode `computeNextStateOfCell`.

Exercice 7

____/4.5

Proposez un tableau des combinaisons qui permet de satisfaire le critère de couverture "Multiple Base Choice" appliqué à deux combinaisons de base : (1) avec une cellule vivante entourée de deux voisines vivantes, (2) avec une cellule morte entourée de deux voisines vivantes.

Exercice 8

____/2

À partir de vos combinaisons, proposez une suite de tests sous la forme d'un tableau des scénarios.

```

1  /**
2   * A Conway game of life. It is composed of a table of cells, each cell
3   * being either dead ('false' value) or alive ('true') value.
4   *
5   * Each cell can be accessed and changed using its (x,y) coordinates in the table.
6   */
7  public interface GameOfLife {
8
9      /**
10       * The x size of the table. Is always equal or larger than 3.
11       *
12       * @return The x size of the table.
13       */
14      int getSizeX();
15
16      /**
17       * The y size of the table. Is always equal or larger than 3.
18       *
19       * @return The y size of the table.
20       */
21      int getSizeY();
22
23      /**
24       * Retrieve the current state of a specific cell of the grid.
25       *
26       * @param x the x coordinate of the cell
27       * @param y the y coordinate of the cell
28       * @return true is the cell is currently alive, false otherwise
29       * @throws IllegalArgumentException if the coordinates are invalid (negative values, or outside the grid)
30       */
31      boolean getCellState(int x, int y) throws IllegalArgumentException;
32
33      /**
34       * Manually change the state of a given cell.
35       *
36       * @param x the x coordinate of the cell
37       * @param y the y coordinate of the cell
38       * @throws IllegalArgumentException if the coordinates are invalid (negative values, or outside the grid)
39       */
40      void setCellState(int x, int y, boolean newValue) throws IllegalArgumentException;
41
42      /**
43       * Compute what should be the next state of a given cell of the grid, following the game rules:
44       * - if cell is *dead* and has exactly 3 *alive* neighbors, it becomes *alive*,
45       * - if cell is *alive* and has exactly 2 or 3 *alive* neighbors, it stays *alive*, else becomes *dead*.
46       *
47       * The neighbors of a cell are the cells that are adjacent horizontally, vertically or diagonally.
48       * The number of neighbors of a cell depend on whether the cell is positioned on
49       * a border (5 neighbors), in a corner (3 neighbors), or neither (8 neighbors).
50       *
51       * Note that calling this service does not change the grid.
52       *
53       * @param x the x coordinate of the cell
54       * @param y the y coordinate of the cell
55       * @return true is the cell should stay alive, false otherwise
56       * @throws IllegalArgumentException if the coordinates are invalid (negative values, or outside the grid)
57       */
58      boolean computeNextStateOfCell(int x, int y) throws IllegalArgumentException;
59
60      /**
61       * Evolve the grid into a new state, using the rules of the game of life.
62       * Will first call computeNextStateOfCell() on each cell of the grid, then recreate the grid accordingly.
63       */
64      void evolve();
65  }

```

FIG. 1 : L'interface GameOfLife