NOM :

PRÉNOM :

Nº ÉTUDIANT :

Test logiciel (XMS1IE301)
Contrôle Continu Écrit (CCE)

Feuille recto-verso de notes manuscrites autorisée

Durée : 2h

Sujet écrit sur 11 pages

XX/XX/20XX

- **N'oubliez pas de mettre votre nom et numéro en majuscules dans les cases en haut de chaque feuille.**
- **Écrivez directement sur le sujet dans les zones de réponse. Ne pas déborder de ces zones.**
- **Si vous n'avez plus assez de place dans une zone de réponse, écrivez *"voir intercalaire"*, et écrivez dans un intercalaire demandé à un surveillant, avec votre nom et le numéro de la question sur l'intercalaire.**
- **Merci de rendre vos feuilles avec les pages dans le bon ordre et dans le bon sens.**
- **Le barème est donné à titre indicatif, il sera susceptible de changer.**
- **La taille d'une boîte de réponse n'est pas forcément représentative de la longueur attendue de la réponse.**

NE PAS RETOURNER CETTE PAGE AVANT LE TOP DÉPART DE L'ÉPREUVE.

Exercice 1 – Questions de cours ($\approx$ 3 points)

Q.1 Expliquez la différence entre un *défaut* et une *défaillance*

Q.2 Quel est la particularité du *test unitaire,* et quel est l'intérêt spécifique de cette pratique ?

NOM :

PRÉNOM :

Nº ÉTUDIANT :

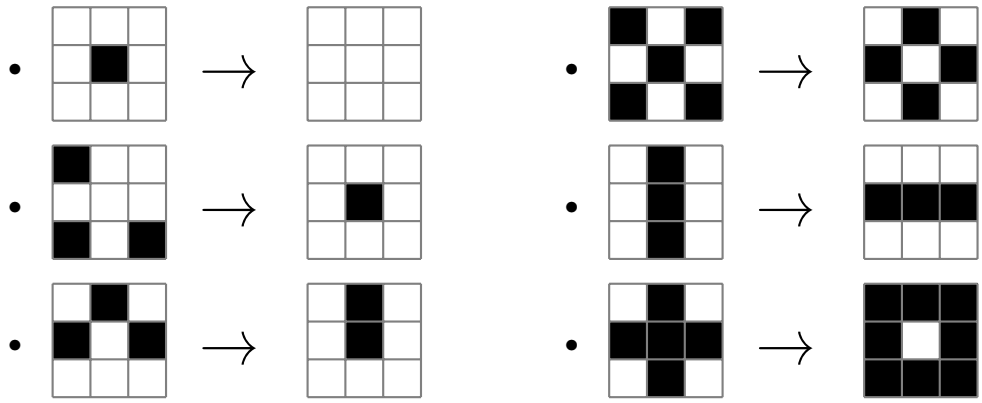
Exercice 2 – Approche combinatoire (≈ 9 points)

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 étapte donnant lieu à une nouvelle grille. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines 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 **??**. 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.

```java
/**
 * A Conway game of life. It is composed of a table of cells, each cell
 * being either dead ('false' value) or alive ('true') value.
 *
 * Each cell can be accessed and changed using its (x,y) coordinates in the
 * table.
 */
public interface GameOfLife {

    /**
     * The x size of the table. Is always equal or larger than 3.
     * @return The x size of the table.
     */
    int getSizeX();

    /**
     * The y size of the table. Is always equal or larger than 3.
     * @return The y size of the table.
     */
    int getSizeY();

    /**
     * Retrieve the current state of a specific cell of the grid.
     *
     * @param x the x coordinate of the cell
     * @param y the y coordinate of the cell
     * @return true is the cell is currently alive, false otherwise
     * @throws IllegalArgumentException if the coordinates are invalid (negative
     *                                  values, or outside the grid)
     */
    boolean getCellState(int x, int y) throws IllegalArgumentException;

    /**
     * Manually change the state of a given cell.
     *
     * @param x the x coordinate of the cell
     * @param y the y coordinate of the cell
     * @throws IllegalArgumentException if the coordinates are invalid (negative
     *                                  values, or outside the grid)
     */
    void setCellState(int x, int y, boolean newValue) throws IllegalArgumentException;

    /**
     * Compute what should be the next state of a given cell of the grid, following
     * the game rules:
     * - if cell is *dead* and has exactly 3 *alive* neighbors, it becomes *alive*,
     * - if cell is *alive* and has exactly 2 or 3 *alive* neighbors, it stays
     * *alive*, else becomes *dead*.
     *
     * The neighbors of a cell are the cells that are adjacent horizontally,
     * vertically or diagonally.
     * The number of neighbors of a cell depend on whether the cell is positioned on
     * a border (5 neighbors), in a corner (3 neighbors), or neither (8 neighbors).
     *
     * Note that calling this service does not change the grid.
     *
     * @param x the x coordinate of the cell
     * @param x the y coordinate of the cell
     * @return true is the cell should stay alive, false otherwise
     * @throws IllegalArgumentException if the coordinates are invalid (negative
     *                                  values, or outside the grid)
     */
    boolean computeNextStateOfCell(int x, int y) throws IllegalArgumentException;

    /**
     * Evolve the grid into a new state, using the rules of the game of life.
     * Will first call computeNextStateOfCell() on each cell of the grid, then
     * recreate the grid accordingly.
     */
    void evolve();
}
```

NOM :

PRÉNOM :

Nº ÉTUDIANT :

Q.3 Donnez les domaines d'entrée et de sortie de la méthode.

Q.4 En vue de tester la méthode `computeNextStateOfCell()` selon l'approche combinatoire étudiée en cours, on souhaite modéliser le domaine d'entrée sous la forme d'un *arbre de classification*. **Proposez un arbre de classification pour ce problème. Si besoin, ajoutez dans l'encart dédié les *contraintes* nécessaires pour votre modèle.**

Contrainte(s) :

NOM :

PRÉNOM :

Nº ÉTUDIANT :

Q.5 Proposez un *tableau des combinaisons* qui satisfait un critère "*Base Choice Coverage*" où la combinaison de base correspond au cas suivant :

- Les paramètres sont valides,
- La cellule est vivante,
- La cellule possède 8 voisines, dont 2 vivantes.

Q.6 À partir de votre tableau des combinaisons, proposez un *tableau des scénarios* avec toutes les colonnes attendues.

NOM :

PRÉNOM :

Nº ÉTUDIANT :

Exercice 3 – Testabilité et doublures (≈ 8 points)

On considère le code Java montré en Figure 1 et 2 qui fait partie d'un petit système logiciel dédié à la *domotique*.

Notre objectif est de tester l'implémentation de l'opération `decide()` spécifiée dans l'interface `HomeAutomation`, et implémentée dans la classe `HomeAutomationImpl`. Cette méthode permet de décider si les volets (en anglais *shutters*) d'une maison doivent être fermés ou non, en fonction de la quantité de lumière, de la quantité de vent, et de la direction du vent. Ces informations sont récupérées auprès de capteurs à l'aide de la classe RealSensors.

Plus précisément, on souhaite écrire un script JUnit pour le scénario de test suivant :

« Lorsque la luminosité est 200, la vitesse du vent 12, et la direction du vent venant du nord, alors on décide de fermer les volets. »

```
1  interface HomeAutomation{
2      /* Get information from sensors, then decides whether shutters must be opened or closed.
3      *
4      * - If luminosity is strictly less than 400, close shutters for the night.
5      * - If luminosity is between 400 and 100 000, open shutters.
6      * - If luminosity is strictly more than 100 000, close to avoid the heat.
7      * - If wind direction is WEST, and wind speed strictly more than 40, open the shutters so that they don't break (because the windows
         ↪  mostly face west). THIS RULE OVERRIDES ALL OTHER RULES.
8      * - In all other cases, we do nothing.
9      *
10     * @return "OPEN" if shutters should be opened, "CLOSE" if they should be closed, "NOTHING" if nothing must be done.
11     */
12     PlannedShuttersAction decide();
13 }
14
15 class HomeAutomationImpl implements HomeAutomation {
16
17     private Sensors sensors;
18
19     public HomeAutomationImpl(Sensors sensors) {
20         this.sensors = sensors
21     }
22
23     @Override
24     public PlannedShuttersAction decide() {
25         if (this.sensors.readWindDirection() == Direction.WEST
26             && this.sensors.readWindSpeed() > 40 ) {
27             return PlannedShuttersAction.OPEN;
28         }
29         if (this.sensors.readLuminosity() < 400 || this.sensors.readLuminosity() > 100 000) {
30             return PlannedShuttersAction.CLOSE;
31         } else {
32             return PlannedShuttersAction.OPEN;
33         }
34         return PlannedShuttersAction.NOTHING;
35     }
36 }
37
38 enum PlannedShuttersAction {
39     OPEN, CLOSE, NOTHING
40 }
```

FIGURE 1 – Interface HomeAutomation et son implémentation la classe HomeAutomationImpl.

```java
interface Sensors {
    Direction readWindDirection();
    int readWindSpeed();
    int readLuminosity();
}

class RealSensors implements Sensors{
    /**
     * Constructor. Requires real sensors hardware port address.
     */
    public RealSensors(int hardwarePortAddress) { ... }
    ...
}

enum Direction {
    NORTH, SOUTH, EAST, WEST
}
```

FIGURE 2 – Interface Sensors et un bref extrait de son implémentation RealSensors.

Q.7 Quel(s) problème(s) de testabilité présente la méthode `decide()` ?

Q.8 A-t-on besoin de doublures pour tester `decide()` ? Si oui, pour chaque doublure :

- Donnez l'interface/classe qui doit être remplacée par une doublure,
- Décrivez quels changements sont nécessaires dans le code pour pouvoir utiliser cette doublure.

NOM :

PRÉNOM :

Nº ÉTUDIANT :

Q.9 À l'aide de JUnit et Mockito, implémentez le scénario de test souhaité. Vous considérerez que les changements que vous avez estimé nécessaires dans la question précédente ont bien été réalisés.
Pour rappel, avec Mockito :

- `mock(MyInterface.class)` permet de créer un objet mock,
- `when(myMock.myMethod(5)).thenReturn(8);` permet de configurer un objet mock.

Il n'est pas demandé d'écrire les `imports` Java nécessaires.