Durée totale : **3h**. Tous documents autorisés. Nombre de pages : **5**
barème indicatif : spécification JML et test = 2/3, model-checking = 1/3

Exercice 1 : Spécification JML et test : Sudoku

On s'intéresse au développement très partiel d'un paquetage `sudoku` (fig 1) pour un jeu de Sudoku interactif en ligne. On ne traitera pas la partie «correction» du sudoku qui décide si une valeur entrée dans une case est cohérente avec le reste de la grille. Une grille sera donc vue uniquement comme un tableau carré de dimensions 9*9 dont les cases peuvent être vides ou contenir une valeur entre 1 et 9. Les 9 lignes et colonnes sont numérotées de 1 à 9. La grille est subdivisée en blocs : des carrés de taille 3*3 numérotés de 1 à 9 de la gauche vers la droite et du haut vers le bas (le bloc 1 dans le coin supérieur gauche, le bloc 9 dans le coin inférieur droit). On ne traitera pas du tout la partie interface graphique. Le but de l'examen est de spécifier et tester une partie de l'implémentation.

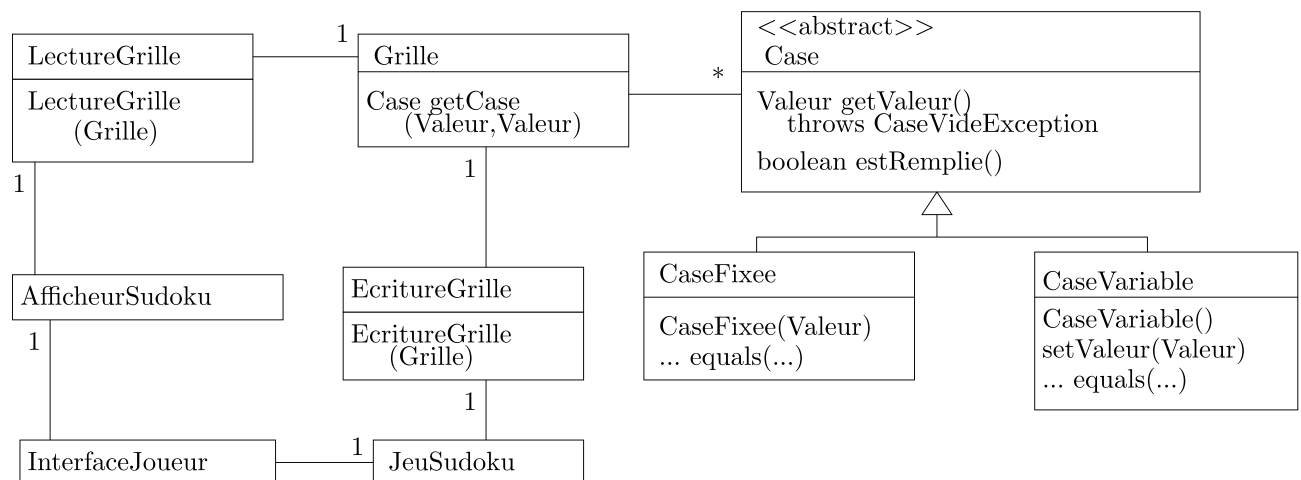

Fig. 1 – Diagramme UML du paquetage `sudoku`

Les spécifications des classes et méthodes seront données en JML, incluant la signature de la méthode mais **sans le code** de leur corps, sauf quand le sujet le précise explicitement. Les cas de tests seront rédigés en Java, dans une syntaxe se rapprochant de celle de JUnit[1] 3 ou 4 et accompagnés d'un commentaire. Tout l'exercice est à traiter dans l'esprit de la **programmation défensive**. Il n'est pas demandé d'écrire des tests purement unitaires, sauf mention explicite.

On suppose donné un type énuméré `Valeur` = {`un`, `deux`, `trois`, `quatre`, `cinq`, `six`, `sept`, `huit`, `neuf`} qui sert à la fois de valeurs pour les cases et d'indices pour les lignes, colonnes et blocs de la grille.

1.1 : À propos des cases

Les cases de la grille (classe abstraite `Case`) sont divisées en deux catégories : celles qui sont pré-remplies et dont la valeur n'est pas modifiable (classe `CaseFixee`) et celles qui sont à remplir et dont la valeur est modifiable (classe `CaseVariable`), voir figure 1. Une case qui n'a pas de valeur a par convention une valeur `null`. Deux cases sont égales si elles sont toutes les deux vides, ou toutes les deux remplies avec la même valeur. `Case` contient les méthodes abstraites :

- `Valeur getValeur() throws CaseVideException` qui retourne la valeur de la case ou lève une exception si elle est vide ;
- `boolean estRemplie()` indique si la case contient une valeur.

[1]Le but n'est pas de contrôler votre connaissance de la syntaxe de JUnit mais votre capacité à imaginer des tests pertinents pour l'application.

Q 1.1 : On suppose donné un attribut JML `val` de type `Valeur` qui représente la valeur d'une case. Donner les spécifications JML des méthodes de `Case`. ☐

Q 1.2 : Si l'attribut `val` était de type entier (en remplaçant la référence `null` par la valeur 0), que faudrait-il ajouter à la spécification JML de `Case` ? ☐

Q 1.3 : On donne pour la méthode `estRemplie` de `CaseFixee` la postcondition JML suivante : `ensures \result ;` Donner le code de cette méthode. Dites si, à votre avis, cette postcondition respecte le principe du sous-typage comportemental, en justifiant votre réponse. ☐

Q 1.4 : Donner une classe de test JUnit `TestLesCases` contenant deux cas de test qui testent judicieusement les méthodes `getValeur` et `estRemplie` pour `CaseVariable` (on se passera des directives d'importation Java pour JUnit, mais le paquetage devra être spécifié.). ☐

Q 1.5 : Il est possible de comparer des cases en redéfinissant la méthode `equals` de `Object`. Donner la spécification JML de la méthode `equals` de `CaseVariable`. ☐

Q 1.6 : Donner les cas de test JUnit qui vous semblent judicieux pour tester la méthode `equals` de `CaseVariable`. ☐

1.2 : À propos de la grille

La `Grille` est implantée par une matrice 9*9 Java de `Case` indexée classiquement de 0 à 8. On suppose donné un attribut JML `grille` de type `Case[][]`. La méthode `Case getCase(Valeur l, Valeur c)` retourne une référence sur la `Case` placée ligne `l`, colonne `c` dans la grille.

	1	2	3	4	5	6	7	8	9
1		7	4		9		5		
2	1		5	8			6		4
3	6	9		5	4				1
4		6	7		8			4	3
5	8		3	6			9	5	
6						5			
7	3	5			7				9
8				2	6			7	
9		1	2	3			4		

FIG. 2 – une grille de sudoku

Q 1.7 : Soit `gEx` une `Grille` représentant la grille de vierge sudoku donnée fig 2 (les cases à valeur variable de la grille sont toutes vides). On souhaite tester la méthode `getCase` sur `gEx`. Expliquer brièvement :
- quelles sont les données de test ;
- comment vous choisissez ces données ;
- la liste des données choisies.

Enfin donner un cas de test JUnit correspondant à la donnée de votre choix. ☐

L'accès à la grille se fait via deux classess Java : `EcritureGrille` pour l'accès en écriture et `LectureGrille` pour l'accès en lecture.

La méthode `void ecrire(Valeur l, Valeur c, Valeur v) throws CaseFixeeException` de `EcritureGrille` écrit la valeur `v` dans la case à contenu variable (remplie ou non) de ligne `l` et de colonne `c`, avec levée d'une `CaseFixeeException` si la case n'est pas à contenu variable.

Q 1.8 : En utilisant la grille `gEx`, donner les cas de test JUnit qui vous semblent judicieux pour tester la méthode `ecrire` de `EcritureGrille`. □

Dans la classe `LectureGrille` la méthode `List<Case> getCasesBloc(Valeur b)` retourne une liste de cases contenant les cases du bloc `b` de haut en bas et de la gauche vers la droite.

Q 1.9 : On souhaite tester la méthode `getCasesBloc` sur la grille `gEx`. Donner l'ensemble des données de test qui vous semblent judicieuses (justifier brièvement ce choix) ainsi qu'un cas de test JUnit correspondant à la donnée de votre choix. □

1.3 : À propos de l'`InterfaceJoueur`

La classe `JeuSudoku` gère les différentes options de jeu (auto-correction, choix d'affichage, etc). Elle contient une méthode `void ecrire(Valeur l, Valeur c, Valeur v) throws CaseFixeeException` qui écrit dans la case (`l`, `c`) la valeur `v`, avec levée d'une `CaseFixeeException` si la case n'est pas à contenu variable.

La classe `InterfaceJoueur` fournit une méthode `selectionnerCase(Valeur l, Valeur c)` qui sélectionne la case de coordonnées (`l`,`c`). La méthode `ecrire(Valeur v)` permet d'écrire la valeur `v` dans la dernière case sélectionnée (si c'est possible). Cette méthode n'a aucun effet si aucune case n'a été sélectionnée. Dans le cas contraire, elle demande d'abord au `JeuSudoku` d'effectuer l'écriture. Ensuite, **uniquement** si la case concernée est bien à contenu variable, elle demande à la classe `AfficheurSudoku` de rafraîchir l'affichage de la grille (méthode `void rafraichir()`).

On souhaite tester la méthode `ecrire` de `InterfaceJoueur` par un **test d'interaction** basé sur l'utilisation de **mocks**.

Q 1.10 : Expliquer brièvement :
- quels mocks doit-on utiliser ;
- comment modifier le diagramme UML donné fig 1 pour le permettre ;
- quelle est la signature du contructeur de `InterfaceJoueur` ;
- quels sont les objets impliqués dans les tests (les déclarer et indiquer comment les construire si ce n'est pas un mock).

□

Q 1.11 : Décrivez les cas de test qui vous semblent judicieux en :
- décrivant brièvement en français le scénario testé ;
- indiquant en Java les appels à l'objet testé ;
- indiquant en français le comportement attendu des mocks.

□