M. Gasquet, N. Lazaar, C. Nadal, N. Palleja, S. Robillard

Patron MVC

1 Contexte

L'objectif de cet exercice est de refactoriser une solution existante de Sudoku en Java qui utilise une approche antipattern en utilisant le pattern MVC avec les patrons qui vont avec (Stratégie, observateur, composite, commande).

2 Rappel

Le modèle MVC (Modèle-Vue-Contrôleur) est un modèle de conception couramment utilisé en développement de logiciels. Il divise une application en trois parties distinctes : le modèle, la vue et le contrôleur.

- Le modèle représente les données et la logique métier de l'application. Il est responsable de la gestion des données et de leur traitement. Dans le cas d'un jeu de Sudoku, le modèle gère la grille de jeu et les règles du jeu.
- La vue représente l'interface utilisateur de l'application. Elle est responsable de l'affichage des données et de la réception des entrées utilisateur. Dans le cas d'un jeu de Sudoku, la vue affiche la grille de jeu et permet à l'utilisateur d'entrer les valeurs dans les cases.
- Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Il est responsable de la gestion des interactions entre l'utilisateur et le modèle. Dans le cas d'un jeu de Sudoku, le contrôleur reçoit les entrées de l'utilisateur, les valide avec le modèle et met à jour la vue en conséquence.

La séparation des responsabilités entre ces trois parties rend le code plus facile à comprendre, à maintenir et à étendre. Le modèle peut être utilisé indépendamment de la vue ou du contrôleur, ce qui permet de le tester facilement. La vue peut être facilement remplacée ou mise à jour sans affecter le modèle ou le contrôleur. Le contrôleur peut également être facilement remplacé ou étendu pour prendre en charge de nouvelles interactions utilisateur.

En somme, le modèle MVC permet de mieux organiser et structurer une application en séparant les différentes responsabilités en trois parties distinctes.

3 Solution Antipattern

Voici la solution antipattern :

- La classe `SudokuSolver` contient la logique de résolution du Sudoku.
- La classe `Sudoku` contient les données du jeu.
- La classe `SudokuApp` est responsable de l'exécution du programme.

Question 1 • Donner le diagramme de classes de la solution antipattern.

Question 2 • Quel(s) principe(s) SOLID est/sont violé(s) dans la solution antipattern ?

Question 3 • Comment le pattern MVC peut-il aider à résoudre ce problème ?

Question 4 • Quels sont les avantages du pattern MVC par rapport à l'approche antipattern utilisée ici ?

Question 5 • Quels sont les avantages des patrons Stratégie, observateur, composite, commande en combinaison avec MVC dans ce contexte de Sudoku ?

4 Solution MVC

Question 6 • Quels sont les problèmes de la solution antipattern actuelle ?

Question 7 • Comment peut-on diviser la solution en trois parties distinctes : modèle, vue et contrôleur ?

Question 8 • Comment peut-on appliquer le patron Observateur pour que la vue soit notifiée des changements dans le modèle ?

Question 9 • Comment peut-on appliquer le patron Stratégie pour que différents algorithmes de résolution du Sudoku puissent être utilisés dans le modèle ?

Question 10 • Comment peut-on appliquer le patron Commande pour permettre au contrôleur de modifier le modèle de manière encapsulée ?

Question 11 • Comment peut-on appliquer le patron Composition pour que la vue puisse afficher la grille de Sudoku et les cellules individuelles de manière hiérarchique ?

Question 12 • Comment peut-on relier toutes les parties entre elles pour que le jeu de Sudoku fonctionne correctement ?

Question 13 • Donner le diagramme UML de la solution MVC.

5 Partie Java

Question 14 • Créer la classe `SudokuModel` qui contiendra les données du jeu, avec les méthodes `getValueAt`, `isValueValid`, `setValueAt`, `getBoardSize`, `registerObserver` et la méthode `isGameFinished`.

Question 15 • Créer la classe `SudokuView` qui sera responsable de l'affichage du jeu, avec la méthode `update` qui affichera la grille du jeu et la méthode `displayVictoryMessage` qui affichera un message de victoire.

Question 16 • Créer la classe `SudokuController` qui sera responsable de la gestion des événements utilisateur, avec la méthode `startGame` qui initialisera le jeu et la méthode `handleUserInput` qui gérera les entrées de l'utilisateur.

Question 17 • Créer la classe `BacktrackingSolver` qui implémentera l'algorithme de résolution du Sudoku, avec la méthode `solve`.

Question 18 • Créer la classe `SudokuObserver` qui sera l'interface de base pour tous les observateurs du Sudoku.

Question 19 • Créer la classe `SudokuCellView` qui sera un observateur de la cellule du Sudoku.

Question 20 • Créer la classe `SudokuCommand` qui représentera une commande qui pourra être annulée et répétée, avec la méthode `execute` et la méthode `undo`.

Question 21 • Créer la classe `SetValueCommand` qui représentera une commande pour définir la valeur d'une cellule, avec la méthode `execute` et la méthode `undo`.

Question 22 • Utiliser le patron Observateur pour que `SudokuModel` hérite de `SudokuObserver` et puisse informer tous les observateurs (ici `SudokuCellView`) d'un changement de valeur.

Question 23 • Utiliser le patron Stratégie pour que `BacktrackingSolver` puisse être remplacé par d'autres algorithmes de résolution sans impacter le reste du code.

Question 24 • Utiliser le patron Commande pour permettre d'annuler et de répéter les actions de l'utilisateur dans le `SudokuController`.

Question 25 • Utiliser le patron Composition pour construire la vue du Sudoku à partir de multiples `SudokuCellView`.