R4.01 - Architecture Logicielle

TP - Mini projet
Sudoku MVC

Introduction

L'objectif de ce TP et de ce mini-projet est de **refactorer** une solution existante de Sudoku en Java qui utilise une approche **anti-pattern**. Le but et de retravailler l'architecture de l'application en utilisant le pattern **MVC** en ajoutant en plus les patterns **Observateur**, **Stratégie**, et **Commande**.

Dans un premier temps, l'application devra permettre, comme la solution actuelle, de **résoudre** automatiquement un **sudoku** donné. Dans un second temps, l'application permettra à l'utilisateur de jouer.

Rappels

Le modèle **MVC** (Modèle-Vue-Controleur) est un modele de conception couramment utilisé lors du développement de logiciels. Il divise une application en trois parties distinctes : le modèle, la vue et le controleur.

- 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.

Consignes

Les consignes suivantes ont pour but de vous faire analyse la solution **anti-pattern** puis de construire une solution avec **MVC** en exploitant divers **deisgn patterns**. Vous serez partiellement guidé dans les grandes lignes de la conception d'une telle solution mais pour le détail, vous devrez réfléchir et proposer des solutions par vous même !

Solution Anti-pattern

La solution **anti-pattern** récupérée dans le mail de lancement du projet contient :

- Un dossier **dataset** contenant de fichiers de sudokus. Les valeurs des cellules sont séparées par des points virgules.
- Une classe **Sudoku** qui contient les données d'un sudoku (tableau à deux dimensions) et permet d'en construire à partir d'un fichier (du **dataset**). Cette classe permet aussi de vérifier la validité d'un nombre placé, d'afficher la grille, etc...
- Une classe **SudokuSolver** qui permet de résoudre le sudoku en appliquant un algorithme de **backtracking**.
- Une classe **SudokuApp** qui permet de lancer le **solver** sur un sudoku du dataset.

Nous souhaitons analyser cette solution.

Pour les questions qui nécessitent une réponse écrite, répondez dans un fichier **README** que vous rendrez avec le projet.

1 - Faites tourner la solution sur les 3 sudokus proposés.

2 - Réalisez le **diagramme de classes de conception** de cette solution.

3 - Quel(s) principe(s) **SOLID** est/sont violié(s) dans cette solution ?

4 - Dans la classe **Sudoku**, quels éléments pourraient appartenir à la partie **modèle** et quels éléments pourraient appartenir à la partie **vue** ?

Solution MVC et autres patterns

Votre objectif est donc de reprendre ce compte pour y appliquer le pattern **MVC** et y greffer d'autres patterns pour réaliser diverses fonctionnalités.

Observateur

Il est tout d'abord souhaitable de séparer la vue et le modèle et faire en sorte qu'une mise à jour d'une modèle entraîne une mise à jour de la vue.

Comme dans la section précédente, continuez de répondre aux questions qui nécessitent une réponse écrie dans le fichier **README**.

1 - Comment séparer la classe **Sudoku** en une entité **vue** et une entité **modèle** ?

2 - Comment utiliser la pattern **Observateur** pour faire en sorte qu'une mise à jour d'une cellule du sudoku déclenche une action sur la vue ? En l'occurrence, cette action sera simplement d'afficher les coordonnées mise à jour (avec la nouvelle valeur) puis d'afficher la grille.

3 - Initialisez un nouveau **diagramme de classes de conception** de votre solution en incluant les classes correspondants à la vue et au modèle liés au sudoku ainsi que les classes liées à la mise en place du pattern **Observateur**. N'oubliez pas de bien préciser les méthodes !

4 - Créez un projet **java** (ou alors simplement un package **pattern** dans la solution existante...) et implémentez votre début conception !

Contrôleur

Il est maintenant temps de placer un **contrôleur** pour faire l'intermédiaire entre la **vue** et le**modèle**. Pour l'instant, ce contrôleur va un peu jouer le rôle de ce que faisait **SudokuApp** dans la solution **anti-pattern** (c'est-à-dire, lancer le solveur).

Votre **contrôleur** doit être capable de gérer une un **modèle** et d'y faire appel. Il va effectuer des actions sur le modèle. Normalement, avec votre conception précédente, le modèle notifie directement la vue lorsqu'il est mis à jour, sans avoir besoin de passer par le contrôleur. Donc rien à changer de ce côté-là.

Dans notre fonctionnement on aura alors la **vue** qui fera appel au **contrôleur** qui effectuera des actions sur le **modèle** qui déclenchera alors la mise à jour de la vue (grâce à l'observateur). La **vue** doit donc connaître et pouvoir agir directement sur le **contrôleur** !

Le contrôleur a aussi besoin de gérer le **solver** pour effectuer la résolution du sudoku. Il y a donc aussi besoin d'une méthode pour déclencher la résolution.

1 - Modifiez votre diagramme de classes pour inclure un **contrôleur**. Réfléchissez bien à ses attributs, son constructeur, ses méthodes, etc...

2 - Modifiez votre implémentation **java** en conséquence.

3 - Créez une novuelle classe **SudokuApp** contenant un **main** afin de tester la résolution d'un sudoku avec votre tout nouveau **contrôleur**.

Stratégie

Actuellement, nous utilisons un algorithme de **backtracking** pour résoudre le sudokuIl existe d'autres algorithmes pour résoudre un tel puzzle.

1 - Compte-tenu de cette information, en quoi votre conception actuelle n'est pas évolutive ?

2 - Comment utiliser la pattern **Strategie** pour y remédier ?

3 - Modifiez votre **diagramme de classes de conception** pour inclure le fait d'avoir plusieurs algorithmes de résolution en utilisant le pattern **Strategie**.

4 - Modifiez votre implémentation **java** en conséquence et vérifiez que tout fonctionne bien.

Commande

Nous allons maintenant inclure le fait de pouvoir jouer, c'est-à-dire pouvoir compléter la grille à partir de la **vue** en faisant appel au **contrôleur**.

Un tour se déroule ainsi :

1. Le joueur sélectionne une ligne puis une colonne
2. Le joueur sélectionne une valeur
3. La demande est envoyée au **contrôleur** qui exécute l'action sur le **modèle**.
4. La vue est mise à jour (grâce à l'**observateur**).

De plus, la vue doit afficher au joueur s'il a gagné ou non après chaque placement de valeur (et arrêter le programme s'il a gagné).

Côté **contrôleur** chaque **action** de modification d'une valeur dans la grille doit être enregistrée puis doit pouvoir éventuellement être annulée.

Du côté de la **vue**, il faut donc proposer au joueur au début de chaque tour le choix entre placer une valeur ou bien annuler sa dernière action. On proposera aussi d'arrêter le jeu ce qui mettra fin à l'exécution du programme.

1 - Comment utiliser la pattern **Commande** afin de mettre en place ce système ?

2 - Modifiez votre **diagramme de classes de conception** pour inclure le pattern **commande** au niveau de l'action de modification d'une valeur dans la grille. Il faudra aussi ajouter les méthodes nécessaires à la sélection des coordonnées et de la valeur au niveau de la **vue**.

4 - Modifiez votre implémentation **java** en conséquence.

5 - Ajoutez une méthode (sur votre diagramme puis dans votre implémentation) dans la **vue** qui sera la méthode principale lancée lors de l'exécution de votre programme. Cette méthode doit proposer le choix entre jouer une partie ou bien la résolution automatique du sudoku.

6 - Côté **java**, faites en sorte que votre programme (**SudokuApp**) lance cette méthode principale sur la vue.

Bonus

Quelques pistes pour améliorer votre projet :

— Faire en sorte de laisser l'utilisateur choisir le sudoku.
— Proposer la génération aléatoire d'un sudoku (valide).
— Utiliser une interface graphique, notamment pour la partie **vue**. Cela peut se faire avec JavaFX. Il pourrait être alors intéressant de réfléchir à l'utilisation du design pattern **composite** pour afficher la grille.
— Vos idées sont les bienvenues !

Si vous avez pour projet de faire des ajouts, suave gardez d'abord l'état final fonctionnel du projet (diagramme + java) et travaillez sur une copie pour vos ajouts ! Il faudra donc aussi penser à faire un diagramme de classes de conception évolué, si vous faites des ajouts.

Rendu

Votre rendu devra contenir :

— Le fichier **README** avec vos réponses pour les questions nécessitant une réponse écrite.
— Le **diagramme de classes de conception** de la solution **anti-pattern**.
— Le **diagramme de classes de conception** votre solution **MVC**.
— Le **code source** de votre solution **MVC**.
— Les ressources (code, diagrammes, etc...) liés aux éventuels ajouts **bonus** réalisés.

Le projet devra être déposé au plus tard le **4 avril 2023** (jusqu'à **23h55**) sur **Moodle** dans la zone de dépôt prévue à cet effet.