Rapport de Projet : Sudoku Solver

Lina GASMI
Ramlat MAOULANA CHARIF

SOMMAIRE

I - Introduction

Le projet a pour but de développer un solveur de Sudoku capable de résoudre une grille en appliquant une série de règles de déduction successives. Trois règles de déduction (DR1, DR2 et DR3) ont été implémentées pour résoudre le puzzle de manière progressive. En cas de besoin, l'utilisateur est invité à remplir une case manuellement. L'objectif final est de fournir un programme qui, à partir d'un fichier texte contenant une grille de Sudoku initiale, applique les règles de déduction jusqu'à ce que la grille soit remplie ou qu'il devienne nécessaire d'interagir avec l'utilisateur.

Le programme est développé en Java et utilise plusieurs design patterns pour structurer le code de manière modulaire et maintenable. Ce rapport décrit les problèmes rencontrés, les choix techniques effectués et les solutions apportées.

II - Conception

1 - Initialisation

Besoin :
Pour résoudre le sudoku on doit d'abord récupérer le sudoku donné par l'utilisateur.

Choix technique :
La gestion de ce besoin à été pris en charge dans la classe `SudokuBoard`.

- Le `SudokuBoard` récupère un fichier texte.
- Il traite ligne par ligne le fichier.
- Il remplit le plateau initial à partir des lignes.

2 - Itérabilité du plateau

Besoin :
Durant ce projet nous avons eu besoin à plusieurs reprises de parcourir diverses collections: tableaux, lignes, colonnes et blocs.

Choix technique :
Nous avons utilisé le **design pattern Itérator** pour répondre à ce besoin.

- **Pourquoi ce choix ?** Ce pattern fournit une manière uniforme et abstraite de parcourir les éléments, indépendamment de la manière dont ils sont organisés. Chaque collection peut être créée dans une classe distincte qui implémente l'interface `CollectionsIterator`.

Avantages :

- **Encapsulation de la structure interne** : le parcours de la collection se fait sans connaissance de la structure de celle-ci.
- **Uniformité d'accès** : peu importe la structure de l'élément parcouru l'itération reste la même. Il suffit juste d'appeler les méthodes **hasNext()** et **next()**.
- **Personnalisation du parcours** : on peut choisir de parcourir l'élément comme on le veut, c'est par exemple ce que l'on a fait pour le parcours d'un bloc.

3 - Choix des DR

Besoin :
Implementation d'au moins 3 règles de déduction.

Choix technique :
Nous avons implémenté trois règles de déduction : **DR1**, **DR2**, et **DR3** :

- **DR1** : élimination des possibilités, on identifie les chiffres possibles qui peuvent être placés dans une case donnée, en éliminant ceux qui sont déjà présents dans la même ligne, colonne ou le même bloc.
- **DR2** : technique du Locked Candidates, on élimine des candidats dans une grille en se basant sur l'interaction entre une **ligne (ou colonne)** et un **bloc.**
- **DR3** : technique **XY-Wing**, nécessitant une analyse des relations entre plusieurs cellules pour en déduire une valeur.

4. Gestion de l'enchaînement des règles

Besoin :
Il est nécessaire d'appliquer les règles de déduction (DR1, DR2, DR3) de manière successive. Chaque règle doit s'exécuter dans un ordre précis, et si le sudoku n'est pas résolu par une règle, la suivante doit être appliquée.

Choix technique :
Nous avons utilisé le **design pattern Chain of Responsibility** pour répondre à ce besoin.

- **Pourquoi ce choix ?** Ce pattern permet de structurer l'enchaînement des règles tout en gardant un système flexible et extensible. Chaque règle est implémentée dans une classe distincte, dérivée de la classe `DeductionRule`, tandis que le `RuleManager` centralise et orchestre l'application des règles.

  Rôle du RuleManager :

  - Le `RuleManager` initialise la chaîne des règles.
  - Il passe la grille de sudoku à chaque règle successivement.
  - Il vérifie après chaque exécution si la grille est résolue et arrête ou continue le processus en conséquence.

Avantages :

- **Flexibilité** : Ajouter ou modifier une règle est simple. Il suffit d'implémenter une nouvelle classe de règle et de l'intégrer dans la chaîne via le `RuleManager`.
- **Séparation des responsabilités** : Chaque règle reste autonome, ce qui facilite la maintenance et le débogage.
- **Centralisation** : Le `RuleManager` offre un point d'accès unique pour gérer les règles, ce qui simplifie la coordination et garantit un fonctionnement cohérent.

5 - Gestion de l' Interaction avec l'utilisateur

Besoin :

Lorsqu'un enchaînement de règles ne suffit pas à résoudre le sudoku, l'utilisateur doit être invité à entrer une valeur manuellement dans une cellule.

Choix technique :

La gestion de cette interaction est assurée par la classe **UserInputHandler.** Nous avons choisi d'utiliser le **design pattern Observer** pour répondre à ce besoin.

- **Pourquoi ce choix ?** Le pattern Observer est adapté pour notifier automatiquement la classe `UserInputHandler` lorsque le `RuleManager` détecte que le sudoku n'est pas entièrement résolu.

Rôle de UserInputHandler :

- La classe **UserInputHandler** est responsable de la gestion complète de l'entrée utilisateur.
- Lorsqu'elle est notifiée par le `RuleManager`, elle sollicite une action de l'utilisateur (saisir une valeur dans une cellule) et met à jour le plateau de sudoku en conséquence.

Avantages :

- **Automatisation** : La notification entre le `RuleManager` et le `UserInputHandler` est entièrement automatique, sans intervention explicite dans le code principal.
- **Découplage** : Le `RuleManager` n'a pas besoin de connaître les détails de la gestion des entrées utilisateur. Cela réduit le couplage entre les classes et améliore la modularité.
- **Responsabilités clairement définies** : Chaque classe se concentre sur sa propre tâche : le `RuleManager` gère les règles, tandis que l'`UserInputHandler` gère les interactions avec l'utilisateur.

6 - Gestion de la difficulté du sudoku (Bonus)

Besoin :
Déterminer la difficulté d'une grille de Sudoku en fonction des règles appliquées pour la résoudre.

- **Choix technique** :
  Création d'une énumération (`Difficulty`), définissant trois niveaux : **FACILE**, **MOYEN**, et **DIFFICILE.**
- Le `RuleManager` est responsable de la gestion de la difficulté, en mettant à jour le niveau après chaque cycle d'application de règles.

7 - Unicité des instances (Singleton)

Besoin :
Nous avons remarqué que certaines classes, comme `RuleManager` et `UserInputHandler`, n'ont besoin que d'une seule instance dans tout le projet.

Choix technique :
Nous avons décidé d'implémenter le **design pattern Singleton** pour ces classes.

- **Pourquoi ce choix ?**

  Le pattern Singleton garantit qu'il n'y a jamais plus d'une instance des classes `RuleManager` et `UserInputHandler`. Cela centralise leur gestion et permet un accès global à ces objets, tout en évitant des problèmes liés à des instances multiples (comme des données incohérentes ou des comportements imprévisibles).

Avantages :

- **Contrôle global** : L'accès centralisé à une instance unique simplifie la gestion .
- **Prévention des conflits** : L'unicité des instances élimine les problèmes potentiels liés à des modifications concurrentes .

8 - Modularité et collaboration

Besoin :

Le projet devait être organisé de manière à permettre à plusieurs personnes de travailler indépendamment sur différentes parties sans qu'il y ait d'interférences entre elles. Cette approche favorise également la maintenance et l'évolutivité à long terme.

Choix technique :

Nous avons structuré le projet en adoptant une architecture modulaire basée sur les **principes SOLID**, avec une attention particulière au **Single Responsibility Principle (SRP).**

Structure modulaire :

1. **Module SudokuBoard** :
    - Gère la structure de la grille, son état (cellules remplies ou non), et les opérations liées à la manipulation de la grille.
2. **Module RuleManager** :
    - Orchestre l'application des règles de déduction (DR1, DR2, DR3) et gère leur enchaînement.
3. **Module UserInputHandler** :
    - Gère l'interaction avec l'utilisateur en cas de blocage dans la résolution automatique.
4. **Module Iterator**:
    - Permet la navigation et l'itération sur les cellules du Sudoku pour évaluer leur état ou appliquer les règles de déduction.

Avantages :

1. **Respect du principe SRP** :
   Chaque classe ou module se concentre sur une tâche précise, réduisant les responsabilités croisées et les dépendances inutiles. Cela améliore la lisibilité et facilite le test unitaire de chaque composant.
2. **Collaboration facilitée** :
   La modularité permet à chaque membre de l'équipe de travailler sur un module distinct sans conflit, grâce à des responsabilités bien définies.
3. **Maintenance simplifiée** :
   Les bugs ou améliorations sont plus faciles à identifier et à mettre en œuvre car chaque module est indépendant.
4. **Lisibilité et clarté** :
   Une architecture bien organisée rend le projet plus compréhensible pour de nouveaux développeurs ou pour une reprise ultérieure du code.

III - Développement et Problèmes rencontrés

Développement des itérateurs :

Lors de la conception de ce projet, nous avons initialement tenté d'utiliser uniquement le **design pattern Iterator** pour implémenter l'extraction et la création de lignes, colonnes et blocs itérables dans une grille. Cependant, cette approche a rapidement révélé ses limites. En essayant de tout centraliser dans un unique design pattern, j'ai rencontré des difficultés importantes : complexité accrue du code, manque de flexibilité, et difficulté à gérer les différentes structures d'itération.

Pour surmonter ces obstacles, nous avons choisi de décomposer le problème en plusieurs sous-problèmes, chacun étant mieux adapté à un design pattern spécifique. On a notamment utilisé :

- Le **pattern *Factory*** pour créer dynamiquement des objets représentant les lignes, colonnes et blocs.
- Le **pattern *Iterator*** pour parcourir facilement les éléments de ces structures.

Cette approche a permis une solution plus modulaire, extensible et maintenable. En isolant les responsabilités dans des composants bien définis, chaque aspect du problème a pu être traité de manière efficace tout en respectant les principes de conception orientée objet.

IV - Bilan

Le projet de solveur de Sudoku est bien conçu, avec une architecture modulaire et l'utilisation judicieuse de **design patterns** pour répondre à des besoins spécifiques.

Voici les points essentiels :

Points forts :

1. **Modularité** : Une séparation claire des responsabilités entre les modules (SudokuBoard, RuleManager, etc.), facilitant la collaboration et la maintenance.
2. **Design patterns** adaptés :
    - **Iterator** pour parcourir les éléments (lignes, colonnes, blocs).
    - **Factory** pour la création dynamique des structures.
    - **Chain of Responsibility** pour enchaîner les règles de déduction (DR1, DR2, DR3).
    - **Observer** pour automatiser les interactions utilisateur.
    - **Singleton** pour garantir l'unicité des composants critiques.
3. **Règles de déduction progressives** : DR1, DR2 et DR3 gèrent les cas simples à complexes.
4. **Interaction utilisateur gérée automatiquement** via un système flexible.

Problèmes et solutions :

- Difficulté initiale avec le pattern Iterator : résolue en combinant **Iterator** et **Factory**.

Améliorations possibles :

1. Ajouter des **tests**
2. **Optimiser** les règles complexes comme XY-Wing.
3. Proposer une **interface utilisateur** plus intuitive (graphique ou interactive).
4. **Mieux gérer** les erreurs et anomalies dans les fichiers d'entrée.
5. Utiliser de manière plus optimale les méthodes.