Rapport – SudokuSolver

Auteur : Ibrahima **Diallo**

Date: le 09/11/2024
Projet: Software Engineering – Solveur de Sudoku

Introduction

Durant ce projet de Software Engineering, j'ai développé un solveur de Sudoku en Java. Mon objectif était de créer une solution robuste utilisant différentes stratégies de déduction tout en maintenant une architecture claire.

Design Patterns Utilisés

Pour structurer mon code, j'ai choisi d'utiliser quatre Design Patterns principaux :

1. Observer/Observable

Pour maintenir la cohérence de ma grille, j'ai implémenté ce pattern :

```
public class observable {
    private final List<Observer> observers = new ArrayList<>();
    protected void notifyObservers(SudokuGrid grid) {
        for (Observer observer : observers) {
            observer.update(grid);
        }
    }
}
```

Ce choix m'a permis d'informer automatiquement les règles de déduction des changements dans la grille.

2. Singleton (SudokuGrid)

J'ai utilisé ce pattern pour ma grille :

```
public class SudokuGrid {
    private static SudokuGrid instance;
    public static SudokuGrid getInstance() {
        if (instance == null) {
            instance = new SudokuGrid();
        }
        return instance;
    }
}
```

Cette décision m'assurait qu'une seule instance de la grille existait à tout moment.

3. Factory & 4. Strategy

J'ai combiné ces patterns pour gérer mes règles de déduction, ce qui m'a donné une structure flexible et extensible.

Règles de Déduction et Difficultés Rencontrées

DR1 (Facile)

Implémentation : Pour cette règle basique, j'ai créé un algorithme qui :

1. Parcourt chaque cellule vide
2. Vérifie ligne, colonne et bloc 3x3

Difficulté rencontrée : Initialement, je perdais certaines possibilités de résolution. **Solution :** J'ai implémenté une boucle qui continue tant que des changements sont possibles avec une limite de tentatives pour éviter les boucles infinies.

DR2 (Moyenne)

Implémentation : J'ai développé une approche qui vérifie chaque unité (ligne, colonne, bloc) pour chaque chiffre (1-9).

Difficultés rencontrées :

1. **Messages de debug trop nombreux :**

DR2 a placé 7 dans la ligne 8

DR2 a placé 5 dans la ligne 1

Solution : J'ai décidé de les garder temporairement car ils m'aident à suivre le processus de résolution, mais ils sont configurés pour être facilement désactivables.

2. **Progression de la résolution :** J'avais du mal à savoir si la règle progressait réellement. **Solution :** J'ai ajouté un système de retour boolean pour chaque tentative de placement.

DR3 (Difficile)

Implémentation : La règle la plus complexe avec trois techniques :

- Hidden Singles
- Pointing Pairs
- Hidden Pairs

Difficultés rencontrées :

1. **Complexité algorithmique :** La détection des patterns était très complexe. **Solution :** J'ai séparé chaque technique dans sa propre méthode avec des responsabilités claires.
2. **Tests difficiles :** Il était compliqué de trouver des grilles appropriées pour tester. **Solution :** J'ai créé des grilles de test spécifiques pour chaque technique.

3. **Interaction avec autres règles :** Les règles pouvaient interférer entre elles. **Solution :** J'ai implémenté un système de priorité : DR1 → DR2 → DR3.

Problèmes Majeurs et Solutions

1. Gestion des Mises à Jour

Problème : Au début, je perdais des mises à jour de la grille. **Solution :** Pattern Observer pour notifier automatiquement toutes les règles :

```
public class observable {
    protected void notifyObservers(SudokuGrid grid) {
        for (Observer observer : observers) {
            observer.update(grid);
        }
    }
}
```

2. Évaluation de la Difficulté

Problème : Déterminer correctement le niveau. **Solution :** Test séquentiel des règles :

java

```
if (solveWithRule(grid, "DR1")) return EASY;
if (solveWithRules(grid, "DR1", "DR2")) return MEDIUM;
if (solveWithRules(grid, "DR1", "DR2", "DR3")) return HARD;
return VERY_HARD;
```

3. Messages de Debug Verbeux

Problème : Sortie console surchargée. **Solution :** J'ai gardé les messages pour le débogage mais je les ai structurés par niveau de règle pour une meilleure lisibilité.

4. Intervention Manuelle

Problème : Quand permettre l'intervention utilisateur ? **Solution :** J'ai implémenté une vérification systématique :

1. Essai de toutes les règles
2. Vérification de la complétude
3. Si échec → demande d'intervention
4. Validation de chaque entrée manuelle

5. Tentative avec Backtracking

Expérimentation : J'ai initialement implémenté un algorithme de backtracking qui s'est révélé extrêmement efficace.

Problème rencontré : Bien que très puissant (résolvant presque toutes les grilles), le backtracking entrait en conflit avec mon approche par règles de déduction. Il résolvait les grilles si rapidement qu'il rendait difficile l'évaluation réelle de la difficulté et l'application progressive de mes règles.

Décision : J'ai décidé de l'abandonner pour rester fidèle à l'objectif du projet : une résolution progressive basée sur des règles de déduction humainement compréhensibles. Cette décision m'a permis de :

- Mieux évaluer la difficulté des grilles
- Maintenir une résolution étape par étape claire
- Permettre une meilleure compréhension du processus de résolution

Conclusion

Ce projet m'a permis de comprendre l'importance :

- D'une architecture bien pensée
- Du débogage progressif
- De la gestion des cas d'erreur

Les choix techniques que j'ai faits, notamment l'utilisation des Design Patterns et l'abandon du backtracking au profit d'une approche plus pédagogique, ont abouti à une solution qui, bien qu'ayant encore des points d'amélioration (comme l'optimisation des messages de debug), répond efficacement aux exigences initiales.

Améliorations Futures Possibles

1. Optimisation des messages de debug
2. Interface graphique
3. Plus de règles de déduction
4. Système de logging plus sophistiqué

Ce fut un projet enrichissant qui m'a fait comprendre l'importance d'une bonne architecture et des choix de design appropriés.