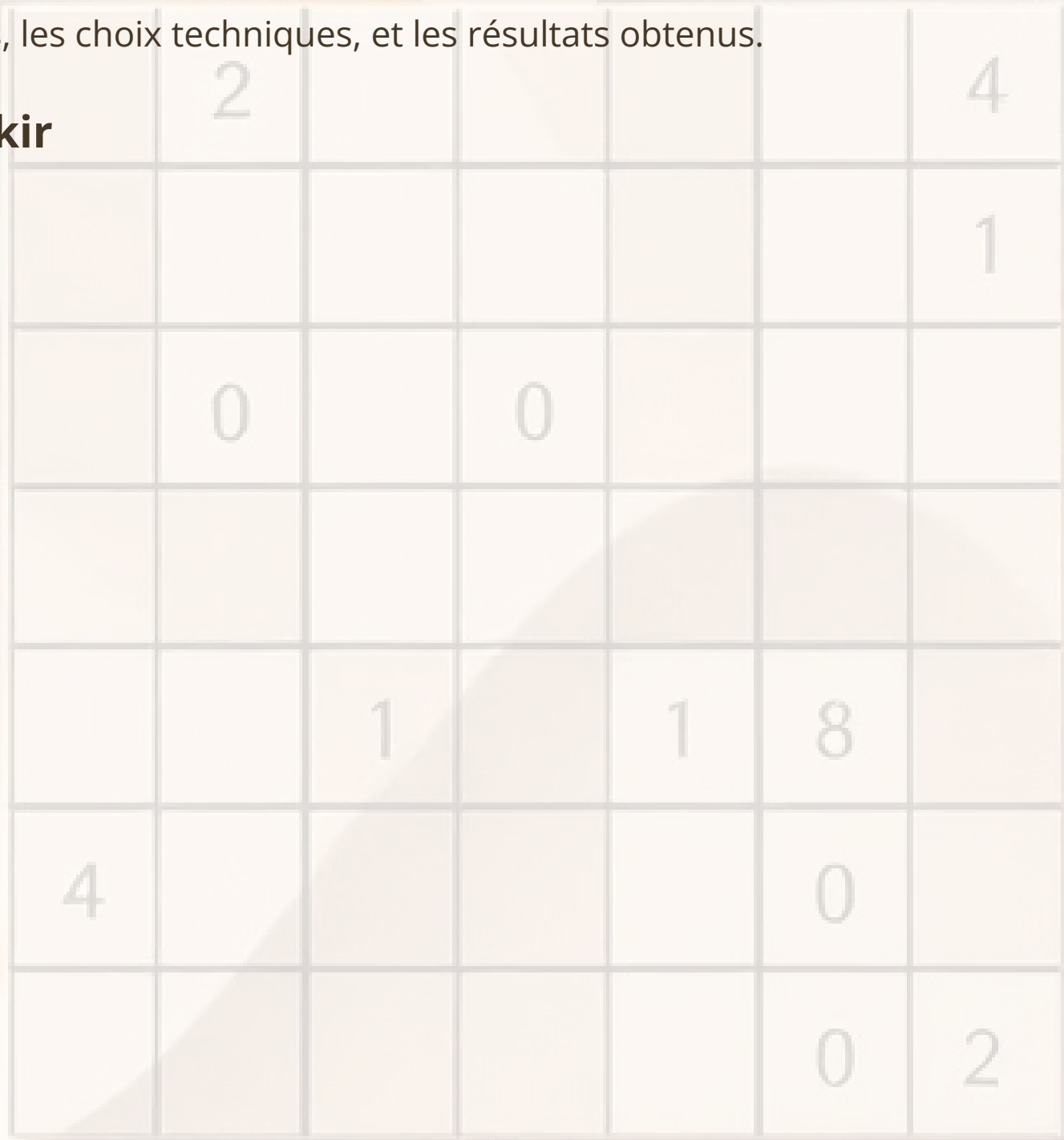
Développement d'un Solveur de Sudoku en Java

Ce document présente le développement d'un programme Java pour résoudre des grilles de Sudoku. Le projet implique la création d'un solveur utilisant des règles de déduction pour remplir les cases de la grille, avec une interaction utilisateur si nécessaire. Il couvre la structure du programme, les problèmes rencontrés, les choix techniques, et les résultats obtenus.

LF **by lamyae fakir**



Introduction au Projet

Pour ce projet, j'ai développé un programme en Java qui résout des grilles de Sudoku. L'objectif était de créer un solveur de Sudoku en appliquant plusieurs règles de déduction pour remplir les cases de la grille. Le Sudoku, c'est ce jeu où on doit placer les chiffres de 1 à 9 dans une grille de 9x9, en respectant certaines contraintes (chaque chiffre doit apparaître une seule fois par ligne, colonne et sous-grille de 3x3).

J'ai défini trois règles de déduction (DR1, DR2 et DR3), que le programme applique pour essayer de compléter la grille. Si ces règles ne suffisent pas pour résoudre complètement la grille, le programme demande à l'utilisateur de remplir certaines cases lui-même. Ce projet m'a permis de pratiquer Java et de mieux comprendre la logique de la résolution de Sudoku.

Problèmes Rencontrés

1 Structuration des Règles

Au début, j'ai eu un peu de mal à structurer les règles de déduction. Il fallait que chaque règle puisse être appliquée de manière indépendante, sans créer de conflits dans la grille. J'ai aussi dû trouver un moyen d'enchaîner les règles sans qu'elles interfèrent les unes avec les autres.

2 Interaction Utilisateur

Un autre problème s'est posé quand j'ai réalisé que les règles seules ne suffisaient pas toujours pour finir la grille. J'ai donc ajouté une interaction utilisateur : quand le programme est bloqué, il demande à l'utilisateur d'entrer des valeurs manuellement. Ça semble simple, mais ça impliquait de vérifier que les valeurs entrées par l'utilisateur n'entraînent pas de conflits dans la grille.

3 Tests et Vérifications

J'ai dû faire pas mal de tests pour m'assurer que chaque règle (DR1, DR2, et DR3) fonctionnait bien, sans casser la logique de la grille. C'était un peu complexe, mais ça m'a permis de mieux comprendre comment structurer le code.

Structure du Programme

Mon programme est organisé autour de différentes classes et interfaces pour bien structurer les règles et les stratégies de résolution. Voici un aperçu des composants principaux :

1

DeductionRule (classe abstraite)

Cette classe sert de base pour toutes les règles, avec une méthode applyRule que chaque règle implémente, et une méthode isValueValid pour vérifier qu'on ne met pas de chiffre au mauvais endroit.

2

DR1, DR2, DR3 (classes pour les règles)

DR1 remplit une case si elle ne peut contenir qu'un seul chiffre possible. DR2 essaie de placer les chiffres en fonction des lignes, colonnes et sous-grilles, et DR3 utilise une méthode plus avancée pour éliminer certains candidats en identifiant des "paires nues".

3

DeductionRuleFactory

Cette fabrique permet de centraliser la gestion des règles, ce qui rend le programme un peu plus organisé et flexible.

4

SolvingStrategy (interface)

Déclare une méthode solve pour appliquer les règles de déduction.

5

SimpleSolveStrategy et ComplexSolveStrategy

SimpleSolveStrategy applique seulement DR1, tandis que ComplexSolveStrategy applique DR1, DR2 et DR3 en boucle tant qu'on trouve des cases à remplir.

6

GridObserver et GridSubject

Ces interfaces permettent de notifier les règles dès qu'il y a un changement dans la grille. Cela m'aide à garder le programme synchronisé sans complexité excessive.

7

SudokuSolver (classe principale)

C'est ici que tout se passe. Cette classe gère la résolution de la grille, charge le fichier de la grille, applique les règles et interagit avec l'utilisateur si nécessaire.

8

Main

C'est le point de départ du programme. Ici, je charge la grille, je lance le solveur, et j'affiche le résultat final.

Choix Techniques

Pour que tout fonctionne bien ensemble, j'ai pris quelques décisions de conception :

1. Organisation des règles : En créant une classe pour chaque règle, je peux facilement modifier les règles ou en ajouter d'autres si besoin. Cela rend le code plus flexible et plus facile à comprendre.
2. Stratégies de résolution : En séparant SimpleSolveStrategy et ComplexSolveStrategy, je peux tester différentes approches pour voir laquelle fonctionne le mieux selon la grille.
3. Gestion des observateurs : En utilisant GridObserver, je peux notifier chaque règle dès qu'il y a un changement dans la grille. Cela permet de garder le programme bien synchronisé sans trop de complexité.

Format d'Entrée et d'Affichage

Le programme utilise un fichier texte (sudoku.txt) pour la grille de Sudoku. Chaque ligne du fichier représente une ligne de la grille, avec les valeurs séparées par des virgules, et les cases vides marquées par des 0. Ce format est simple et pratique, et il correspond aux consignes du projet.

Tests et Validation

J'ai testé le programme avec plusieurs grilles pour m'assurer que les règles fonctionnent bien et que le programme reste stable. J'ai aussi vérifié que les erreurs d'entrée de l'utilisateur (par exemple, si l'utilisateur entre un chiffre qui provoque une incohérence) sont bien détectées. Ces tests m'ont permis de repérer et corriger des bugs et d'améliorer la stabilité du programme.

Tout le code source, le README et ce rapport sont stockés sur GitHub. Cela m'a permis de mieux organiser le projet et de garder une trace des modifications. Je vais envoyer le lien au professeur une fois le projet terminé, comme demandé.

Conclusion

En résumé, ce projet m'a permis de mettre en pratique des concepts de base de la programmation orientée objet en Java tout en travaillant sur un problème concret et intéressant. Le solveur fonctionne bien et applique les règles de déduction que j'ai définies, avec la possibilité pour l'utilisateur de compléter la grille si nécessaire. Si je devais améliorer ce programme, je pourrais ajouter des règles de déduction plus avancées ou intégrer une technique de backtracking pour compléter la grille sans intervention.

Auteur : Lamyae Fakir