Software Engineering

Rapport de développement du Solver de Sudoku

MASTER INFORMATIQUE

UNIVERSITÉ CÔTE D'AZUR

Dylan Malaussena

Introduction

Le but de ce projet est de développer en Java un solver de Sudoku capable de résoudre des grilles en appliquant successivement trois règles de déduction (DR1, DR2, DR3). L'objectif est de résoudre automatiquement les grilles de Sudoku en appliquant les règles dans un ordre hiérarchique. En cas d'échec des déductions, l'utilisateur sera invité à compléter manuellement les cases restantes. Le projet inclut également une hiérarchisation de la difficulté des grilles en fonction des règles appliquées pour la résolution.

Problématique

1. **Résolution automatique des grilles de Sudoku** : Le solver doit pouvoir résoudre une grille en appliquant trois règles de déduction, avec une complexité croissante de DR1 à DR3.
2. **Hiérarchisation de la difficulté des grilles** : Une grille est classée comme facile si elle est résolue par DR1 seule, moyenne si DR2 est nécessaire, et difficile si DR3 est utilisée.
3. **Gestion des cas non résolus** : Si la grille ne peut pas être résolue avec les trois règles, l'utilisateur est invité à compléter les cases manquantes.

Structure du projet et choix de conception

1. Organisation des classes

Le projet est organisé en plusieurs classes pour assurer une structure ordonnée et lisible :

- **SudokuSolver** : Classe principale orchestrant l'application des règles de déduction.
- **DR1, DR2, DR3** : Classes dédiées aux trois règles de déduction.
- **Grille** : Gère la structure et les opérations de la grille de Sudoku.
- **DeductionRule** : Interface pour les règles de déduction, permettant une implémentation du **pattern Strategy**.
- **Parser** : Classe pour charger les grilles depuis un fichier texte.
- **main** : Point d'entrée du programme.
- **Observer**
- **Observable**
- **Factory**

2. Implémentation des règles de déduction

DR1 : Remplissage des cases uniques

La première règle de déduction (DR1) consiste à remplir les cases lorsqu'une seule valeur possible est détectée dans un sous-carré 3x3.

Pour DR1 nous avons fait le choix de choisir une règle de déduction très simple pour dans un premier temps pour simplifier nos choix concernant nos autres règles c'est pour cela qu'on a choisi comme règles de remplir les dernières cases vides

Pourquoi on a fait notre DR1 comme ceci :

Premièrement on utilise une fonction isSingleEmptyCellInSubgrid , tout simplement pour s'assurer qu'on a bien une seule valeur possible dans un des sous carré tout simplement et si cette valeur est égale a -1 (vide) on applique notre deuxième fonction findUniqueValue qui nous permet de trouver la valeur restante qu'on doit mettre dans la case vide. Pour ça on check bien entendu les lignes et colonnes ainsi que les sous carrés (pour être sûr qu'on a pas de doublon) puis on applique notre valeur à notre case et on print notre déduction. On a rajouté un test supplémentaire car on avait des bugs comme quoi des fois on avait -1 au lieu d'une valeur comprise entre 1 et 9.Il existe sûrement plus facile pour réussir cette deduction rule mais ..

```
First Rule of
Programming:
If it works,
don't touch it.
```

DR2 : Remplissage des lignes et colonnes en évitant les doublons

Pour notre DR2 on a choisi le fait de résoudre notre sudoku a l'aide des lignes et des colonnes qui est complémentaire avec nos premières règles et qui est assez simple a implémenter.
On reprend certains principe que notre DR1 (checking des colonnes et lignes) mais on les adaptes pour notre règles
On ajoute deux fonction qui nous permet d'avoir dans un premier temps toute les valeurs qu'on a (vue) dans nos lignes et de les garder dans une liste et on fait la même chose avec nos colonnes ensuite on regarde si notre case est vide si oui on appel notre fonction qui nous permet de chercher la valeur à placer (celle de notre DR1 qu'on a adapter). Une fois fait on supprime le doublon sur notre ligne / colonne. Par exemple admettons que la valeur qu'on a trouver a mettre dans notre case est un 9 on supprime de notre liste des valeurs possible le 9. (j'ai eu cette idée avec un camarade pour ma DR3 naked pairs(le fait d'avoir une liste des valeurs possible en l'occurrence pour chaque sous carré dans la DR3 mais ici pour nos lignes et colonnes) , jusqu'à présent elle n'y était pas dans notre DR2 mais c'est quand j'ai essayé une autre grille puis que je l'ai testé à la main que je me suis aperçu d'un problème ma DR2 des fois ne remplissez pas une casse alors qu'elle aurait du donc j'ai adapté mon code de DR2 a cette idée est cela a fonctionné parfaitement .)

DR3 implémentation des "naked pairs"

Concernant DR3 on a décidé d'implémenter les “Paires Dénudées” ou le principe repose sur le fait que cette règle identifie des paires de valeurs possibles qui apparaissent dans deux cases spécifiques d'une même ligne, colonne, ou sous-carré 3x3. Si deux cellules contiennent uniquement les mêmes deux valeurs possibles, ces valeurs ne peuvent apparaître dans aucune autre cellule de cette ligne, colonne, ou sous-carré.
L’idée de faire cette règle a partir de notre DR2 nous est venue naturellement mais cela s'est avéré un problème car on a réussi à implémenter la logique mais le changement de case est impossible on trouve les pairs aucun soucis mais on n’arrive pas a résoudre avec donc on boucle car notre DR3 applique la logique de notre DR2 on s’est rendu compte trop tard qu’il fallait changer la logique pour qu’elle s’applique pour notre DR3.

3. Classe `SudokuSolver` (singleton)

Cette classe nous sert de hub qui réunit l’utilisation de nos règles dans un premier temps ainsi que de savoir si nous avons résolu notre solver. Dans un deuxième temps elle nous permet si la grille ne c’est pas résolu avec nos algorithmes de la résoudre manuellement le sudoku On utilise un simple fonction qui nous permet de “scanner les valeurs de notre grille et si on trouve -1 (une case vide) alors nous n’avons pas résolu notre grille.
Concernant le choix de notre classe, pourquoi avoir fait la fonction ResolutionMainici et pas ailleurs ? simplicité, on réunit toutes nos fonctions de résolution au même endroit.
on utilise un simple scanner pour que l’utilisateur puissent entre la réponse sur l’entrée standard et une simple boucle for qui get les cellules vide (-1) et nous demande par quoi la remplacer.
Ajout d’un timer pour que quand 10 secondes se sont écoulées, on passe à notre remplissage manuel.

4. Classe `Grille`

La classe Grille est dédiée à la gestion de la structure du Sudoku et de ses valeurs :

On a choisi d’implémenter une classe Grille qui nous permet de gérer tout ce qui concerne notre grille de sudoku ainsi que les valeurs qui lui sont associées avec des get, des set etc.., grâce à cela on a une vision plus claire car c’est ici qu’est défini notre affichage.
Simple choix à faire, pratique d’avoir tout au même endroit les fonctions choisis pour l’affichage est encore une fois implémenter un peu “barbare” mais efficace pour un affichage simple

5. Classe `Parser`

La classe Parser simplifie le chargement des grilles de Sudoku à partir de l'entrée standard

Cette classe nous permet de parser le fichier.txt (qui se trouve dans le dossier "projet_malausena" c'est ce dernier qui s'occupe de mettre en forme le fichier comme demander.
Pourquoi le choix d'un bufferreader ? Simple d'utilisation et facile à implémenter

6. Interface `DeductionRule` (pattern Strategy)

L'interface `DeductionRule` permet d'appliquer le Strategy Pattern pour les règles de déduction (DR1, DR2, DR3). Chaque règle implémente `DeductionRule`, permettant à `SudokuSolver` d'appliquer chaque règle indépendamment. On utilise une interface pour les règles de déduction permettant de les appliquer de manière interchangeable et de centraliser leur utilisation dans `SudokuSolver`.

7. Classe `main`

Main de notre projet on utilise args[0] pour avoir notre fichier sur l'entrée standard

8. Classe `Observer`

Elle contient la méthode `update`, que chaque observateur doit implémenter pour réagir aux notifications de changement.

9. Classe `Observable`

Elle contient les méthodes pour ajouter, supprimer et notifier les observateurs.
on utilise que les ajouts et les notifications

10. Classe Factory

Elle contient la méthode pour crée nos DR

Conclusion

Ce projet de solver de Sudoku en Java démontre l'importance de structurer le code autour de patterns de conception pour répondre efficacement aux exigences. Les choix techniques effectués, comme l'application du Strategy Pattern pour les règles de déduction et la vérification utilisateur pour les valeurs manquantes, ont permis de créer un programme robuste et extensible. Les solutions proposées permettent non seulement de résoudre des grilles variées mais aussi de faciliter la gestion et l'extension du code.

Sources :

- https://stackoverflow.com/questions/4234985/how-to-for-each-the-hashmap
- https://stackoverflow.com/questions/521171/a-java-collection-of-value-pairs-tuples
- https://masteringsudoku.com/naked-pairs/
- https://stackoverflow.com/questions/35157024/how-best-to-implement-naked-single-and-hidden-single-in-scheme
- https://www.w3schools.com/js/js_assignment.asp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_OR_assignment
- https://www.geeksforgeeks.org/list-set-java/
- https://www.baeldung.com/java-hashmap-inside-list
- https://www.javatpoint.com/sudoku-in-java
- https://www.baeldung.com/java-observer-pattern
- https://www.geeksforgeeks.org/set-in-java/
- https://www.codejava.net/java-core/collections/java-set-collection-tutorial-and-examples
- https://stackoverflow.com/questions/40851640/observer-pattern-notify-with-state
- https://support.smartbear.com/testcomplete/docs/app-objects/specific-tasks/grids/jtable/obtaining-setting-cell-values.html

vidéos :

- https://www.youtube.com/watch?v=D2gXHE_GF1U
- https://www.youtube.com/watch?v=mcXc8Mva2bA