Solveur de Sudoku avec *Design Patterns*

Rapport de Projet

Introduction

Ce rapport présente le développement d'un solveur de Sudoku en Python, intégrant quatre règles de déduction et mettant en œuvre quatre *design patterns* pour une architecture logicielle robuste et extensible. Le projet vise non seulement à résoudre des grilles de Sudoku de différentes difficultés, mais également à évaluer la difficulté des grilles en fonction des règles de déduction utilisées pour les résoudre.

Objectifs du Projet

- Résoudre des grilles de Sudoku en utilisant jusqu'à quatre règles de déduction quand nécessaire.
- Évaluer la difficulté d'une grille en fonction des règles de déduction appliquées.
- Permettre une interaction avec l'utilisateur pour fournir des valeurs lorsque les règles de déduction ne suffisent pas.
- Appliquer au moins quatre *design patterns* pour améliorer la structure et la maintenabilité du code.

Structure du projet

- **`main.py`** : Point d'entrée du programme.
- **`solver.py`** : Contient la classe **`SudokuSolver`** responsable de la résolution des grilles.
- **`deduction_rules.py`** : Définit les différentes règles de déduction utilisées.
- **`parser.py`** : Gère la lecture et l'affichage des grilles.
- **`singleton_meta.py`** : Fournit une implémentation du patron *Singleton*.
- **`HowTo.md`** : Fournit des instructions sur l'utilisation du programme.

Règles de Déduction Implémentées

1. Résolution Directe (**`DirectSolve`**) : Remplit les cellules qui n'ont qu'un seul candidat possible.

2. Candidats Bloqués (**`LockedCandidate`**) : Élimine les candidats dans les unités (une unité : une ligne, une colonne ou un bloc) lorsque tous les candidats possibles d'un nombre se trouvent dans une seule ligne ou colonne au sein d'un bloc.

3. Paires Nues (**`NakedPair`**) : Identifie les paires de cellules dans une unité qui n'ont que les mêmes deux candidats, permettant d'éliminer ces candidats des autres cellules de l'unité.

4. Paires Cachées (**`HiddenPair`**) : Trouve deux nombres qui n'apparaissent que dans les mêmes deux cellules d'une unité, permettant de restreindre les candidats de ces cellules à ces deux nombres seulement.

Design Patterns Utilisés

1. *Chain of Responsibility*

Chaque règle de déduction est implémentée comme une classe qui hérite de **`DeductionRule`**. Les règles sont chaînées de manière à ce que si une règle ne peut pas faire de progrès, elle passe la responsabilité à la règle suivante dans la chaîne.

Avantages :

- Extensibilité : Il est facile d'ajouter de nouvelles règles sans modifier le code existant.
- Séparation des préoccupations : Chaque règle est encapsulée dans sa propre classe.

2. *Factory*

La classe **`DeductionRuleFactory`** est responsable de la création de la chaîne de règles de déduction. Elle encapsule la logique d'instanciation et de liaison des règles.

Avantages :

- Isolation de la création d'objets : Le code de création est séparé de la logique de l'application.
- Facilité de maintenance : Les modifications apportées aux règles de déduction n'affectent pas le reste du code.

3. *Strategy*

Les règles de déduction agissent comme des stratégies interchangeables pour résoudre les cellules du Sudoku. Chaque stratégie peut être appliquée sans que le reste du système soit affecté.

Avantages :

- Interchangeabilité : Il est possible de changer les règles de déduction à la volée (entre légèrement en conflit avec la *Chain of Responsibility*).
- Testabilité : Chaque stratégie peut être testée indépendamment.

4. *Singleton*

Le patron *Singleton* est implémenté via un module **`singleton_meta`** utilisant une métaclasse. Cette métaclasse, **`SingletonMeta`**, est utilisée pour la classe **`SudokuSolver`**, garantissant ainsi qu'une seule instance du solveur existe à tout moment.

Avantages :

- Contrôle de l'instanciation : Garantit qu'une seule instance du solveur existe, ce qui peut être utile pour gérer l'état global de la grille.
- Ré-initialisation contrôlée : La métaclasse permet de ré-initialiser l'instance avec de nouveaux arguments, ce qui est important lors de l'interaction avec l’utilisateur.

Difficultés rencontrées :

L'utilisation du Singleton avec **`SudokuSolver`** a causé des problèmes lors de l'interaction utilisateur. En effet, après que l'utilisateur ait entré une nouvelle valeur, le solveur devait être ré-initialisé pour prendre en compte cette

modification. La métaclasse **SingletonMeta** a été conçue pour permettre cette ré-initialisation.

Implémentation Détaillée

1. **DeductionRule** et les Règles de Déduction

Chaque règle de déduction hérite de la classe abstraite **DeductionRule** et implémente la méthode **apply**. Cette méthode tente d'appliquer la règle et retourne le numéro de la règle si des progrès ont été faits.

Exemple pour la classe **DirectSolve** :

```
class DirectSolve(DeductionRule):
    def __init__(self, successor=None):
        super().__init__(rule_number=1,
                         successor=successor)
    def apply(self, grid, candidates):
        # Logique de la résolution directe
        # ...
```

2. Chaînage des Règles avec **DeductionRuleFactory**

La classe **DeductionRuleFactory** crée et lie les règles de déduction :

```
class DeductionRuleFactory:
    @staticmethod
    def create_rule_chain():
        hidden_pair = HiddenPair()
        naked_pair = NakedPair(successor=hidden_pair)
        locked_candidate =
LockedCandidate(successor=naked_pair)
        direct_solve =
DirectSolve(successor=locked_candidate)
        return direct_solve
```

Bien que le patron *Singleton* ne soit pas appliqué ici, la *factory* est utilisée pour centraliser la création de la chaîne de règles.

3. Classe **`SudokuSolver`** avec *Singleton* via Métaclasse

Le solveur gère l'état de la grille, les candidats possibles pour chaque cellule, et applique les règles de déduction jusqu'à ce que la grille soit résolue ou qu'aucun progrès ne puisse être fait.

Points clés :

- Utilisation de la métaclasse **`SingletonMeta`** : Garantit une seule instance du solveur.
- Ré-initialisation contrôlée : Après l'entrée de l'utilisateur, le solveur est ré-initialisé en appelant **`__init__`** sur l'instance existante.
- Gestion des candidats : Maintient une matrice de sets représentant les candidats possibles pour chaque cellule.
- Application des règles : En boucle jusqu'à ce qu'aucun progrès ne soit possible.
- Évaluation de la difficulté : Le niveau de difficulté est déterminé par la règle la plus élevée utilisée.

4. Gestion de l’Interaction Utilisateur

Lorsque les règles de déduction ne suffisent pas à résoudre la grille, le programme demande à l'utilisateur d'entrer une valeur pour la cellule avec le moins de candidats. Grâce à la métaclasse **`SingletonMeta`**, le solveur est ré-initialisé avec la nouvelle grille sans créer une nouvelle instance, ce qui assure la cohérence de l'état global.

Refactoring pour Intégrer les *Design Patterns*

Initialement, le code était monolithique et manquait de modularité. Le refactoring a permis de :

- Modulariser le code en séparant les responsabilités dans différents modules.
- Améliorer la maintenabilité en utilisant des patrons de conception appropriés.
- Faciliter l'ajout de nouvelles fonctionnalités, comme l'évaluation de la difficulté.

Évaluation de la Difficulté

La difficulté d'une grille est évaluée en fonction de la règle de déduction la plus complexe utilisée pour la résoudre :

- Facile : Seulement la résolution directe (DR1) a été nécessaire.
- Moyen : Jusqu'à la règle des candidats bloqués (DR2).
- Difficile : Jusqu'à la règle des paires nues (DR3).
- Très difficile : Jusqu'à la règle des paires cachées (DR4).
- Impossible : La grille n'a pas pu être résolue même après l'interaction utilisateur.

Défis Rencontrés et Solutions

1. Problème avec le *design pattern Singleton* et la Ré-initialisation

- Problème : L'utilisation du *Singleton* pour **`SudokuSolver`** empêchait le solveur de se ré-initialiser correctement après l'entrée utilisateur, car l'état interne pouvait conserver des informations obsolètes.
- Solution : La métaclasse **`SingletonMeta`** a été implémentée pour permettre la ré-initialisation de l'instance unique en appelant **`__init__`** avec les nouveaux arguments. Cela a nécessité une attention particulière pour s'assurer que tous les attributs internes sont correctement ré-initialisés.

2. Gestion des Progrès dans la Méthode **`solve`**

- Problème : Le solveur s'arrêtait prématurément si les règles de déduction ne faisaient pas de progrès, même si **`fill_single_candidates`** pouvait encore remplir des cellules.
- Solution : Modification de la logique de la boucle pour continuer tant que des progrès sont faits, que ce soit par les règles de déduction ou par le remplissage des candidats uniques.

3. Incohérences dans les États Internes

- Problème : Après l'entrée utilisateur, certaines structures de données internes comme les candidats ou les règles pouvaient ne pas être synchronisées avec la grille mise à jour.
- Solution : S'assurer que lors de la ré-initialisation, toutes les structures de données internes sont recalculées en fonction de la nouvelle grille.

Conclusion

Le projet a atteint ses objectifs en implémentant un solveur de Sudoku capable de résoudre des grilles de différentes difficultés en utilisant des règles de déduction plus ou moins avancées. L'utilisation de *design patterns* a amélioré la structure du code, le rendant plus modulaire et maintenable. L'implémentation du patron *Singleton* via une métaclasse a permis de contrôler l'instanciation du solveur tout en permettant sa ré-initialisation après l'interaction utilisateur.

Perspectives d'Amélioration

- Ajout de nouvelles règles de déduction pour résoudre des grilles encore plus complexes.
- Implémentation d'une interface graphique pour une meilleure interaction utilisateur.
- Optimisation des performances pour gérer des grilles plus grandes ou des résolutions en série.

Références

- Règles de déductions : https://www.pennydellpuzzles.com/wp-content/uploads/2019/03/How-to-Solve-Sudoku.pdf?srsltid=AfmBOooUhB_xgGPFb8pIxyllUa2Y_49XnmkuU8-L3msjPy8OXemzNH7Q
- Design Patterns : https://refactoring.guru/design-patterns/python