Adam Mir-sadjadi et Florent Bélot
Master 1 Informatique

Rapport sur le projet de Software Engineering : Sudoku Solver

Table of Contents

Préambule

En amont du projet, nous nous sommes renseignés en détail sur le sudoku et les mathématiques que ce problème mobilise (https://fr.wikipedia.org/wiki/Math %C3%A9matiques_du_sudoku). On en tire les conclusions suivantes :

- Résoudre une grille de sudoku est un problème NP-Complet. La résolution d'une grille en un temps raisonnable passe par l'application de stratégies pré-établies sur une grille mais rien n'indique que les règles utilisées seront nécessairement suffisantes.

- Il peut exister plusieurs solutions pour une grille mais un puzzle bien conçu n'a en principe qu'une solution. On peut supposer qu'un programme qui résout une grille doit retourner systématiquement la même solution pour la même entrée tant qu'il n'intègre pas de dimension aléatoire.

De la page suivante (https://sudoku.com/fr/regles-du-sudoku), qui recense des stratégies de résolution de sudoku, nous avons conclu qu'une bonne échelle de difficulté pour débuter était de considérer les règles suivantes :

-DR1 -> Singletons nus (https://sudoku.com/fr/regles-du-sudoku/singletons-nus/) :

Notre règle « la moins forte », qui consiste, lorsqu'il n'y a plus qu'une valeur possible dans une cellule vide, à lui assigner cette valeur.

-DR2 -> Singletons cachés (https://sudoku.com/fr/regles-du-sudoku/singletons-caches/) :

Similaire à DR1. Consiste, lorsqu'il n'y a qu'une seule cellule qui peut recevoir une valeur donnée dans une ligne, colonne ou carré, à assigner cette valeur à cette cellule.

-DR3 -> Paires nues (https://sudoku.com/fr/regles-du-sudoku/singletons-caches/) :

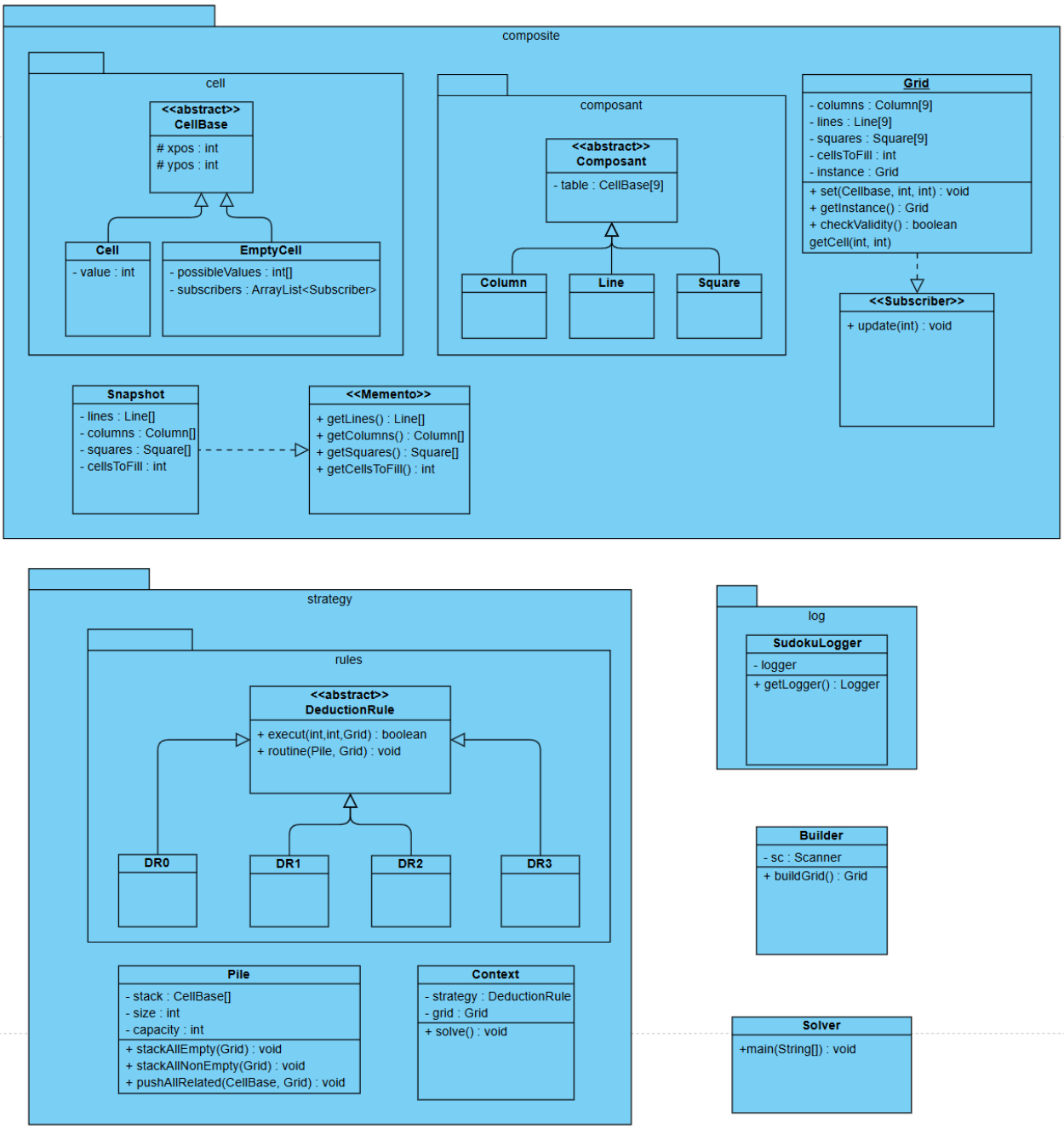
Notre règle « la plus forte », qui consiste, lorsque deux cases au sein d'un carré ont les deux seules mêmes valeurs possibles restantes, à retirer ces valeurs des valeurs possibles des autres cases du carré.
Nous avons effectué ce choix car sans ces règles « de base », la plupart des règles avancées nous ont semblé trop spécifiques à certaines situations, et auraient risqué de ne pas suffire, là où avec cette sélection nous pouvons espérer résoudre la plupart des grilles jusqu'à une certaine difficulté, avec la possibilité de rajouter des règles plus complexes par dessus celles-ci.

A ces trois règles, nous ajoutons DR0, la « règle d'actualisation », qui consiste, pour une cellule dont la valeur est fixée, à retirer sa valeur des valeurs possibles des cellules vides de la même ligne / colonne / carré (donc à mettre à jour la grille lorsqu'une valeur est saisie ou à l'initialisation).

Après avoir défini une architecture ensemble au travers d'un UML, nous avons débuté l'implémentation du programme. Nous détaillons sa conception et son implémentation dans les sections suivantes, en nous attardant sur certains problèmes spécifiques rencontrés et nos réponses à ces-derniers.

Architecture logicielle

UML



L’UML ci-dessus est simplifié pour éviter la surcharge d’informations, il ne fait par exemple pas apparaître tous les getters et setters de chaque classe.

Modélisation de la grille

Composite

On représente la grille par 9 lignes, 9 colonnes et 9 carrés en implémentant le design pattern composite. Ainsi, une grille est constituée de ces 27 composants, stockés dans 3 tableaux de 9 cases chacun. Chaque composant possède sa propre classe concrète dont l'utilité principale est la lisibilité du code et sa facilité de compréhension.
Cette méthode duplique les références aux cellules qui sont contenues dans chacun de ces composants (une cellule est une Cell ou une EmptyCell qui dérivent de la classe abstraite CellBase), mais elle permet de clarifier le processus qui a lieu au sein des DeductionRules.

Nous avons également envisagé d'implémenter le pattern itérateur en rendant notre Grid itérable (on pourrait itérer sur les carrés de la grille, ses colonnes ou ses lignes, ou sur des éléments plus précis) et de seulement stocker un tableau de 81 valeurs, mais il semblait plus logique dans un aspect de programmation orienté objet de créer un objet « complexe » plutôt que de chercher l'efficacité technique.

Dans une implémentation idéale, Cell et EmptyCell seraient également des composants (ainsi que la Grid, ce qui est facilement faisable actuellement mais n'a pas de réel intérêt même du point de vue des bonnes pratiques de développement en POO), mais le comportement d'une cellule a peu de similarités avec celui de son conteneur (ligne/colonne/carré) et il était plus approprié de dédier une classe à part entière à CellBase pour le reste de notre architecture.

Singleton

Il n'existe qu'une seule instance de la grille, pour éviter une utilisation du programme innatendue qui pourrait causer des problèmes non pris en charge. Notre Solver peut résoudre plusieurs grilles en une seule exécution du programme, mais il les résout séquentiellement. L'instance de la grille est remplacée lorsqu'on a fini de traiter une grille donnée.

Memento

Afin de pouvoir recharger la grille sans avoir à appeler le Builder à nouveau, et pour pouvoir progresser par itération, nous avons implémenté le design pattern Memento, qui permet de créer un Snapshot (une sauvegarde de l'état) de la grille à un instant t, et de recharger cet état lorsqu'on le désire. La création du Snapshot se fait toutefois sans certitude quant à la validité de la grille à un moment donné sauf si une cellule vide n'a plus de valeurs possibles, ce qui n'arrive pas forcément immédiatement. Nous aurions pu limiter la création du Snapshot à la partie du code qui résout automatiquement la grille avant que l'utilisateur ne saisisse de chiffre manuellement, étant donné que si les règles sont bien implémentées, le Solver ne peut pas rentrer une fausse valeur pour une cellule de la grille, mais nous avons choisi d'également sauvegarder l'état de la grille après une entrée utilisateur tant que notre condition de vérification de la sanité de la grille est remplie. Si une erreur non détectée a été commise dans les étapes précédentes, l'utilisateur peut recharger le relancer le programme depuis le départ, ce qui serait le comportement normal sans Memento. Durant nos tests, pour le moment, cette séquence d'évènements ne s'est pas produite de manière

fortuite, et nous parvenons tout de même à résoudre les grille très difficile sans relancer le programme.

Observer

Le pattern observateur a également été implémenté « à l'envers » si l'on peut dire. Normalement, un publisher informe de nombreux souscripteurs d'un évènement survenu. Dans notre cas, de multiples publishers (les cellules vides) informent un souscripteur unique (la Grid) si leur nombre de valeurs possibles chute à 0. Cela nous permet de détecter qu'une erreur a été commise, en passant le champ isWrong de la Grid à true, condition qui est vérifiée à chaque itération de la boucle de saisie de valeur par l'utilisateur. Ce pattern a été implémenté car il permettrait aussi à l'avenir d'améliorer le comportement du solver en permettant d'informer le Solver que telle cellule n'a plus qu'une valeur possible (pour appliquer la DR1 dessus par exemple), ou que telle cellule n'en a plus que 2 pour la DR3, et permettre ainsi un comportement qui cible en priorité les cellules dont la valeur a le plus de chances d'être trouvée. A l'heure actuelle, ce mécanisme ne sert qu'à vérifier la sanité de la grille, mais toute la structure est là pour optimiser le solver, ce qui n'a pas été notre priorité durant les finalisations du projet.

Deduction rules

Strategy

L'application des différentes règles est régie par un Context afin de respecter le Design Pattern Strategy. Le context possède une stratégie (l'une des DR qui dérive de la classe abstraite DeductionRule), que l'on peut remplacer par une autre, et une méthode pour appliquer cette stratégie sur la grille en insérant toutes ses cellules vides (sauf pour DR0 où l'on prend les cellules fixées) dans la Pile définie dans le même package. Ce Design Pattern permet de compartimenter les algorithmes de chaque DR dans leur classe respective et de déleguer la gestion de l'application des DeductionRule à une classe dédiée pour alléger le code contenu dans le Solver, qui se contente de remplacer la stratégie du Context lorsque l'on passe un pallier de difficulté (i.e. quand la règle précédente ne suffit pas).

Chain of Responsibility

Le pattern Chain Of Responsibility n'était pas le pattern comportemental que nous visions initialement, bien qu'il ait été envisagé et que nous ayons finalement choisi la Strategy, mais force est de constater que la routine contenue dans la classe de chaque DR (qui consiste à appliquer successivement la DR actuelle ainsi que les DR précédentes aux cellules qui lui sont données en entrée) est une pseudo-implémentation du pattern Chain Of Responsiblity, sans son formalisme, où chaque règle se passe successivement la cellule jusqu'à ce que l'une d'elle soit capable de la traiter (à l'aide d'un retour booléen de la fonction execut(CellBase, Grid) de chaque DR, qui précise si la DR a pu se charger de la cellule ou non).

Visitor (abandonné)

Parmi les difficultés que nous avons rencontré, la difficulté majeure est sûrement l'implémentation du pattern Visitor qui était prévue afin d'éviter l'utilisation de instanceof pour distinguer les Cell des EmptyCell. En effet, les DR prennent en argument une CellBase. Il est donc impossible de distinguer quel héritier de CellBase a été passé en entrée sans demander à la classe en question directement ou utiliser une méthode détournée comme instanceof (ce que nous souhaitions éviter). Le Visitor devait répondre à ce problème, en donnant aux DR la possibilité d'aller « visiter » les héritiers de CellBase en se passant lui même en argument d'une fonction surchargée dans les héritiers accept(Visitor). A la réception, une EmptyCell aurait déclenché en retour Visitor.visitEmptyCell() alors qu'une Cell aurait déclenché Visitor.visitCell(), permettant ainsi d'appliquer le code que l'on souhaite en fonction de la classe concrète passée à la DR. Cependant, les DeductionRule n'ont pas un comportement constant au cours de leurs fonctions execut(CellBase, Grid). Parfois, on teste si l'instance de CellBase est EmptyCell pour ensuite faire quelque chose, puis on le refait de nouveau quelques lignes plus loin pour faire autre chose. La DR3 consiste en une imbrication de ces instructions à tel point qu'il aurait fallu créer de très nombreux visiteurs ou de très nombreuses méthodes surchargées et appeler la méthode correcte en fonction d'un paramètre donné, ce qui faisait d'un design pattern « élégant » un casse-tête pour la maintenance et un labyrinthe de méthodes à lire pour comprendre un algorithme plutôt qu'une base solide à intégrer au programme. Nous étions peut-être proche d'une solution convenable mais faute de l'avoir trouvée à temps nous avons préféré abandonner ce Design Pattern et nous concentrer sur le reste de l'application.

Pile

La pile définie dans le package strategy est une pile de gestion des cellules qui se remplit et se vide à l'appel de chaque routine de DR. Elle nous permet de traiter chaque cellule souhaitée de la grille à chaque itération à l'aide de méthodes qui empilent soit les cellules liées à une autre (même ligne/colonne/carré), soit les cellules vides de la grille par exemple, et d'éviter de traiter chaque cellule systématiquement. Nous avons implémenté notre propre Pile, qui nous permet de rédiger du code spécifique à notre problème plutôt que d'ajouter une couche à une structure déjà implémentée. Par exemple, nous n'empilons une cellule dans la pile que si elle n'y figure pas déjà. Nous aurions pu passer par une HashMap mais nous avons fait le choix de développer notre propre structure en partant du principe qu'à l'avenir nous pourrions ainsi y ajouter des méthodes supplémentaires si des situations que nous n'avons pas prévu se produisent et que cette solution y répond.

Fonctionnement du solver

Builder

L'implémentation d'un Builder permet de déplacer le code de construction de la grille hors de son constructeur afin de s'assurer qu'elle soit construite en suivant une série d'étapes bien définies, l'idée générale étant que la cellule 1 de la ligne 1 soit bien la même que la cellule 1 de la colonne 1 (par exemple), afin qu'une modification de l'une entraîne la modification de l'autre (techniquement c'est juste le même objet qui est pointé) et éviter d'alourdir le code de la classe Grid qui est déjà bien fourni. Le builder prend un Scanner en argument lors de sa création, qui est

supposé contenir la ou les grille(s) au format d’entrée CSV, et construit la grille à partir d’un tableau d’entiers sous forme de caractères qu’il convient de convertir en entier à la création des cellules.

Solver (fonction main)

Le solver Prend en argument le chemin des fichiers à (un ou plusieurs). Il permet d’activer la possibilité d’écrire la solution (s’il la trouve, avec ou sans aide de l’utilisateur) dans un fichier en sortie au travers de la variable statique printToFile, à modifier avant compilation si souhaité.
Nous avons également utilisé un logger pour nous aider à voir les problèmes lors de l'exécution du code et pouvoir activer/désactiver l’affichage des modifications de la grille et l’application DR utilisées, notamment à des fins de débogage.
Pour activer l’affichage des informations détaillées d’éxecution du logger il faut remplacer Level.WARNING par Level.INFO au début du programme du Solver.

Le solver commence par créer un Builder en lui passant le Scanner d’entrée et lui demande de construire la grille.
Il passe ensuite cette grille au context de la strategy qu’il initialise avec la DR0.
Une fois les valeurs possibles de chaque cellule mise à jour, il change la stratégie du contexte et essaie de résoudre le sudoku seul en appliquant d’abord uniquement la DR1, puis la DR1 et la DR2 si DR1 ne suffit pas, puis DR3, DR2 et DR1 si nécessaire. Si la grille n’est pas résolue à ce stade, il sauvegarde son état dans un memento et la deuxième partie de la fonction prend la main. Elle consiste en une alternance de demande de saisie d’une valeur pour une cellule donnée à l’utilisateur avec une application des DR avec la stratégie la plus forte (tout en répétant DR0 à chaque saisie utilisateur), afin d’essayer de résoudre itérativement la grille en collaborant avec l’utilisateur.
A chaque itération, le Solver sauvegarde l’état de la grille s’il ne détecte pas d’erreur (ce qui ne garantit pas qu’une erreur n’a pas déjà été commise par le user, juste qu’elle n’est pas encore détectée). Si une erreur est détectée, il recharge le dernier état de la grille. Si besoin de relancer le programme pour effacer les memento précédents suite à une erreur non détectée, l’utilisateur doit quitter le programme via le signal Ctrl+C.

Tests

La plupart des tests réalisés ont été des tests manuels, étant donné qu'hormis vérifier que la grille est toujours valide et que les valeurs des cellules sont cohérentes, nous avons trouvé peu de tests automatisables. Nous avons envisagé de comparer les solutions données par le solver à des solutions attendues, étant donné que notre solver devrait toujours retourner la même solution pour un sudoku donné qui ne soit pas « très difficile », mais il était plus rapide de vérifier qu'il trouvait toujours la solution à chaque niveau de difficulté de grille manuellement que d'implémenter ce genre de tests, qui n'ont pas beaucoup de pertinence (on s'aperçoit vite si le programme ne fonctionne plus en l'éxecutant sur une grille). Nous vérifions donc que la solution donnée est valide en fin de programme, et à l'aide d'une assertion nous vérifions qu'il n'y a pas eu d'erreur durant l'application automatique initiale des DR, mais c'est tout. Les tests d'intégration des différents modules ont également été réalisés manuellement, le temps passé à écrire des scripts de test dépassant le temps devant être passé à tester l'intégration manuellement pour un programme de cette taille.

Documentation

La documentation du programme a été générée à l'aide de doxygen en paramétrant le fichier de configuration Doxyfile. Les fonctions et champs des différentes classes ont été commentés manuellement suivant le format de commentaire indiqué pour donner une documentation la plus complète et détaillée possible de notre programme.

Répartition du travail

La répartition du travail a été équitable. Nous avons effectué la conception ensemble et communiqué tout au long du projet. L'approche a été plutôt agile, chacun travaillant quand il le pouvait, mais globalement notre charge de travail sur ce projet est équilibrée. Nous avons chacun écrit ou modifié des parties de code dans chaque classe du programme et sommes satisfaits dans l'ensemble du résultat obtenu.