Dossier de conception

SOMMAIRE

I - Introduction

Ce projet d'APO avait pour but de créer des sudoku et de les résoudre. Pour se faire nous avons travaillé à trois :

- Melvyn BAUVENT
- Lilas GRENIER
- Simon PRIBYLSKI

II - Contexte du projet

II.1 - Objectifs et enjeux, terminologie

L'objectif de ce projet est de produire une application en Java permettant de créer et de résoudre des sudoku ou multidoku de toutes tailles ou formes avec des règles de déduction diverses et variées. Le terme "doku" désignera simultanément sudoku et multidoku.

Un multidoku est l'assemblage de plusieurs sudoku avec des blocs en commun.

Un sudoku est un ensemble de cases/cellules avec des règles de déduction/contraintes.

Une cellule est une case qui peut être remplie, ou non, avec un symbole.

Pour qu'un doku soit résolu, il faut qu'il soit complet et qu'il soit valide. Pour être valide, un doku doit respecter toutes ses contraintes.

Une contrainte, ou règle de déduction, limite les symboles que nous pouvons poser dans une cellule, par exemple une contrainte de ligne bloque la pose de plus d'un symbole similaire sur une ligne pour garder la validité du doku.

Pour la résolution des doku, nous devons utiliser trois types d'algorithme :

- Un premier suivant les règles de déduction, en ne posant des symboles que dans les cellules ou les contraintes nous limite à un seul symbole possible pour que le doku reste valide, référé en tant que algorithme "humain".
- Un deuxième faisant du backtracking, en testant toutes les dispositions de symbole pour résoudre le sudoku jusqu'à en trouver une correcte, référé en tant que algorithme "machine".
- Un troisième avec le mixte des deux premiers, référé en tant que algorithme "mixed".

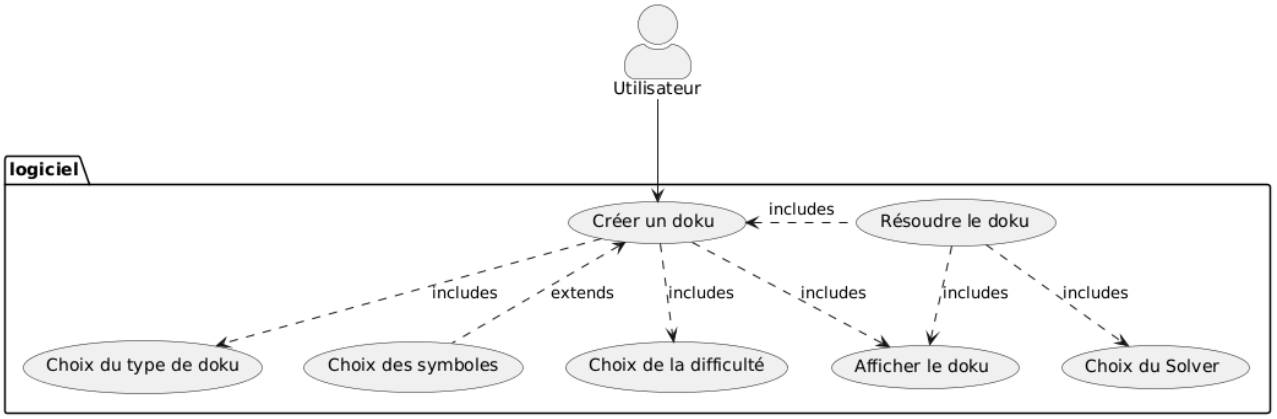
II.2 - Organisation

Le groupe de trois s'est réparti les tâches en fonction des affinités de chacun. Melvyn et Lilas ont été chargés du tronc commun, et Simon des extensions. Le code a été partagé dans un dépôt distant, les différentes parties ou tâches plus complexes réalisées dans des branches distinctes de la principale afin de ne pas empiéter sur le travail des autres.

III - Fonctionnalités du projet

III.1 - Diagramme de cas d'utilisation

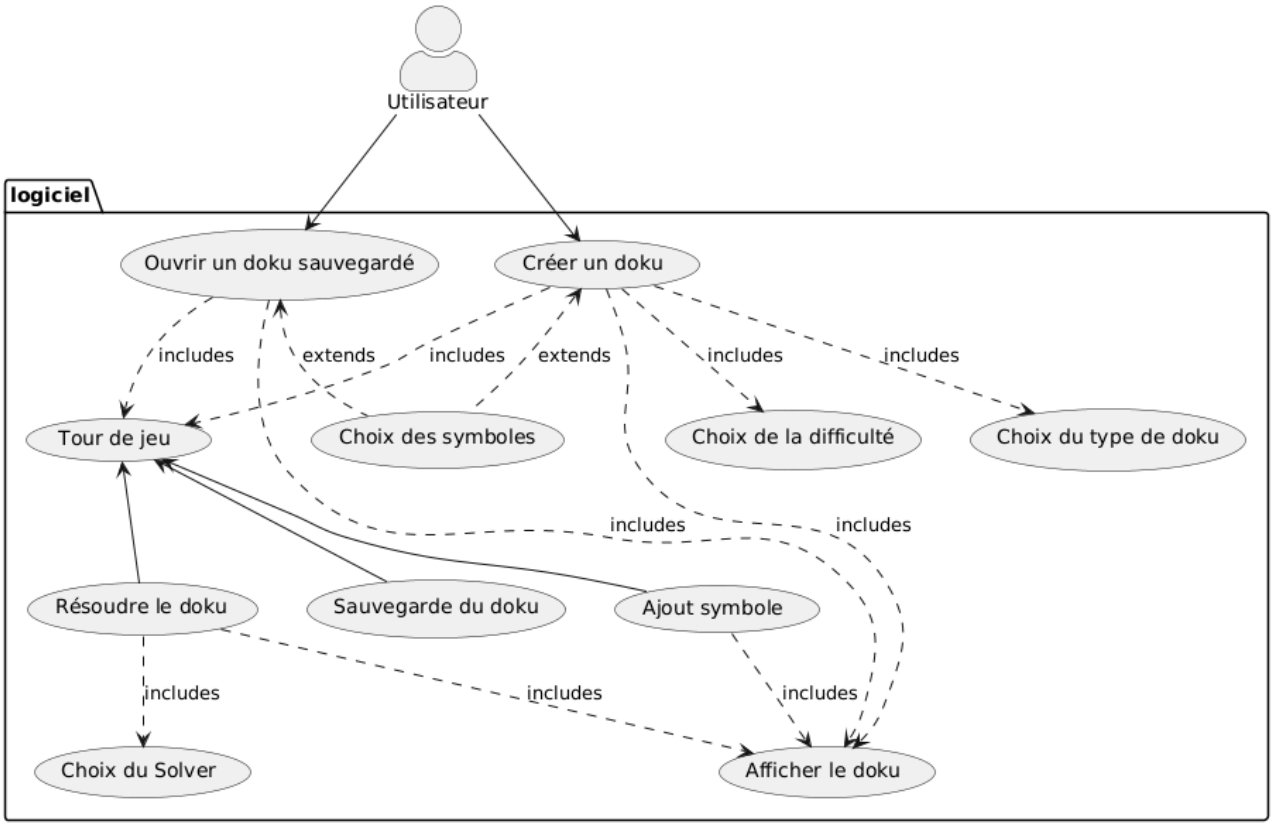
Voici le diagramme de cas d'utilisation (DCU) du projet, qui regroupe les fonctionnalités hors extensions.



Liste des cas d'usage :

- **Choix de la difficulté :** l'utilisateur choisit la difficulté du soku à générer, entre "very easy" (jusqu'à 10% de cases vides), "easy" (resp. 25%), "medium" (resp. 50%), "hard" (resp. 75%), "impossible" (resp. 100%, difficulté maximale, sujet à d'éventuels crashs de l'application).
- **Choix du type de doku :** l'utilisateur choisit la taille des blocs ainsi que le type, simple ou multidoku, du doku à générer.
- **Choix des symboles :** l'utilisateur peut choisir les symboles à utiliser pour l'affichage.
- **Créer un doku :** un doku est généré à partir des données renseignées par l'utilisateur. Les symboles par défaut sont les numéros.
- **Choix du Solver :** l'utilisateur choisit l'algorithme souhaité pour résoudre le doku : "machine", "humain" ou "mixed".
- **Résoudre le doku :** à partir de l'algorithme de résolution choisi par l'utilisateur, résout le doku.
- **Afficher le doku :** affiche le doku.

Et en voici une version concernant l'affichage console, avec cette fois les extensions :



Liste des cas d'usages étendus :

- **Ouvrir un doku sauvegardé :** l'utilisateur pourra charger un doku à partir d'un fichier de sauvegarde préexistant, dont l'utilisateur renseignera le numéro.
- **Tour de jeu :** l'utilisateur pourra choisir entre plusieurs actions, dont la sauvegarde du doku, l'ajout d'un symbole dans une case vide du doku, ainsi que l'appel à un Solver pour résoudre le doku dans son état actuel.
- **Ajout symbole :** l'utilisateur entrera les coordonnées d'une case vide ainsi que le symbole à placer, et le doku sera affiché de nouveau une fois le symbole ajouté.
- **Sauvegarde du doku :** l'utilisateur pourra sauvegarder le doku actuel, en sélectionnant potentiellement une sauvegarde préexistante à écraser.

III.2 - Fonctionnalités générales

Notre application permettra d'implémenter n'importe quel doku, de procéder à sa résolution, et de générer de nouvelles grilles. En particulier notre programme permettra, via un menu textuel :

- Entrer une grille à résoudre.
- Choisir la méthode de résolution : que des règles avec un ensemble à préciser (quitte à ne pas réussir à résoudre), que du retour sur trace, ou un mix des 2.
- Afficher le doku résolu (ou dire si il n'est pas résolvable).
- Afficher/loguer la suite d'opérations effectuées pour résoudre la grille (ou pourquoi elle n'est pas résolvable).
- Générer une ou plusieurs grilles à résoudre à partir d'une grille complète et d'un niveau de difficulté.

III.3 - Fonctionnalités étendues

Notre application aura une interface graphique, toutes les extensions seront accessibles via celle-ci. Cette interface graphique comportera aussi une musique originelle, un fond d'écran personnalisable, et le choix des symboles.

Pour le production de notre application nous utiliserons :

- Une plateforme git auto hébergée avec un système d'intégration continue qui effectuera nos tests à chaque commit, pour gérer le suivi des versions tout au long de notre projet (cf. Annexe 1).
- Junit pour faire des tests unitaires de nos méthodes principales afin de s'assurer de leur bon fonctionnement et de l'intégration continue, comme vue précédemment.

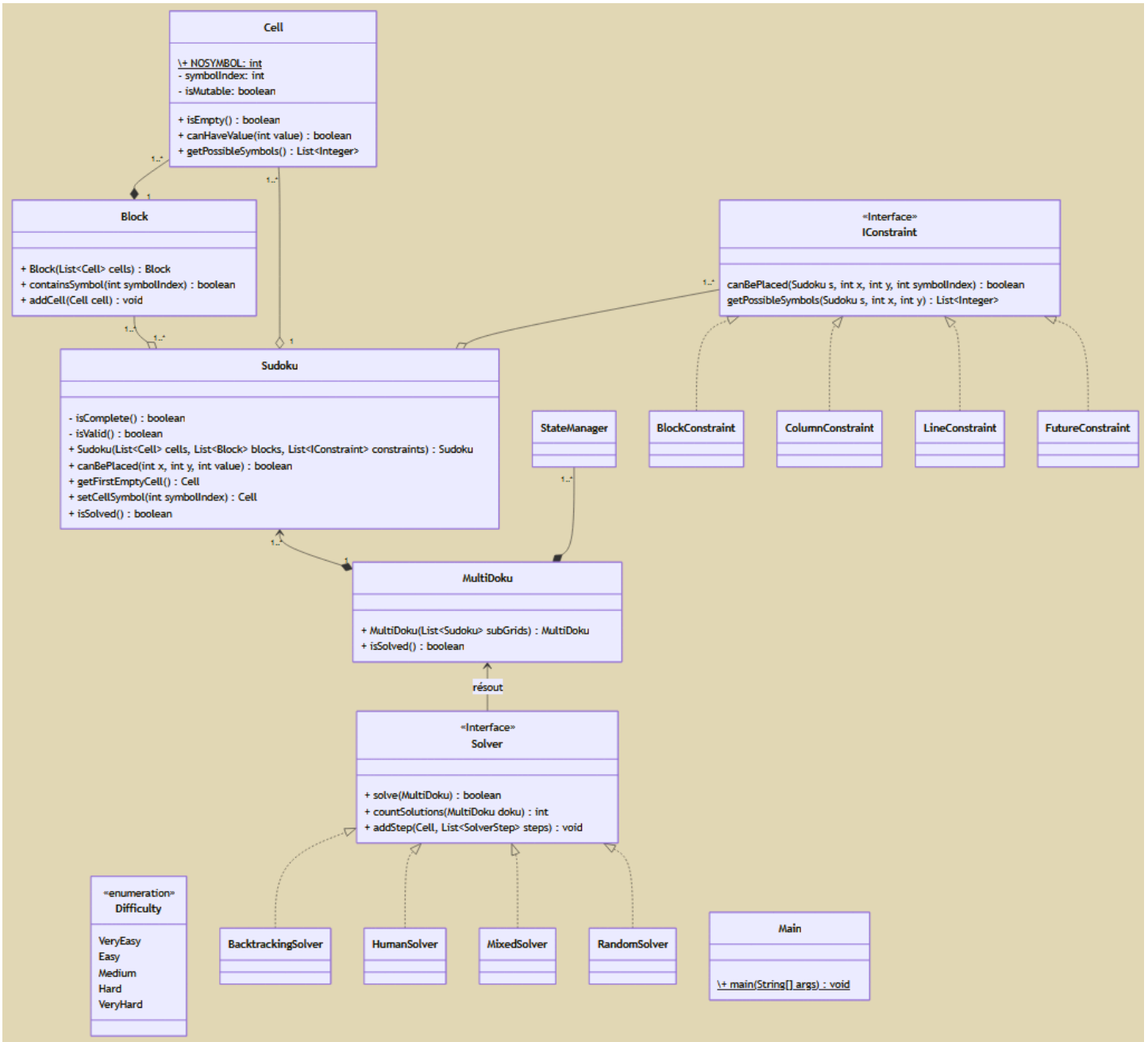
Notre application permettra, en plus des fonctionnalités générales :

- Résoudre manuellement les dokus.
- Sauvegarder et charger des dokus et leurs avancés via des fichiers de sauvegardes.
- Ajouter des contraintes facilement dans le code, grâce à notre implémentation avec une interface IConstraint.
- Jouer en multijoueur dans un mode "battle royal" (cf. Annexe 2) :
    - Un sudoku unique pour tous les joueurs, à résoudre simultanément.
    - Le premier qui le résout gagne la partie et interrompt les autres joueurs.
    - Un système de leaderboard, pour voir le premier et sa place par rapport aux autres.

IV - Conception de l'application

IV.1 - Diagramme de classes (DCL)

L'analyse des acteurs a mené au diagramme de classes (DCL) suivant :



Remarques :

Un MultiDoku est composé d'au moins 1 Sudoku, et s'il est détruit, le Sudoku le sera avec.

Un Sudoku contient des Cells et peut contenir des Blocks. (Si un Sudoku n'a pas contrainte de Block il n'a pas besoin de Block).
Si un Block est nécessaire, il sera composé de Cells.

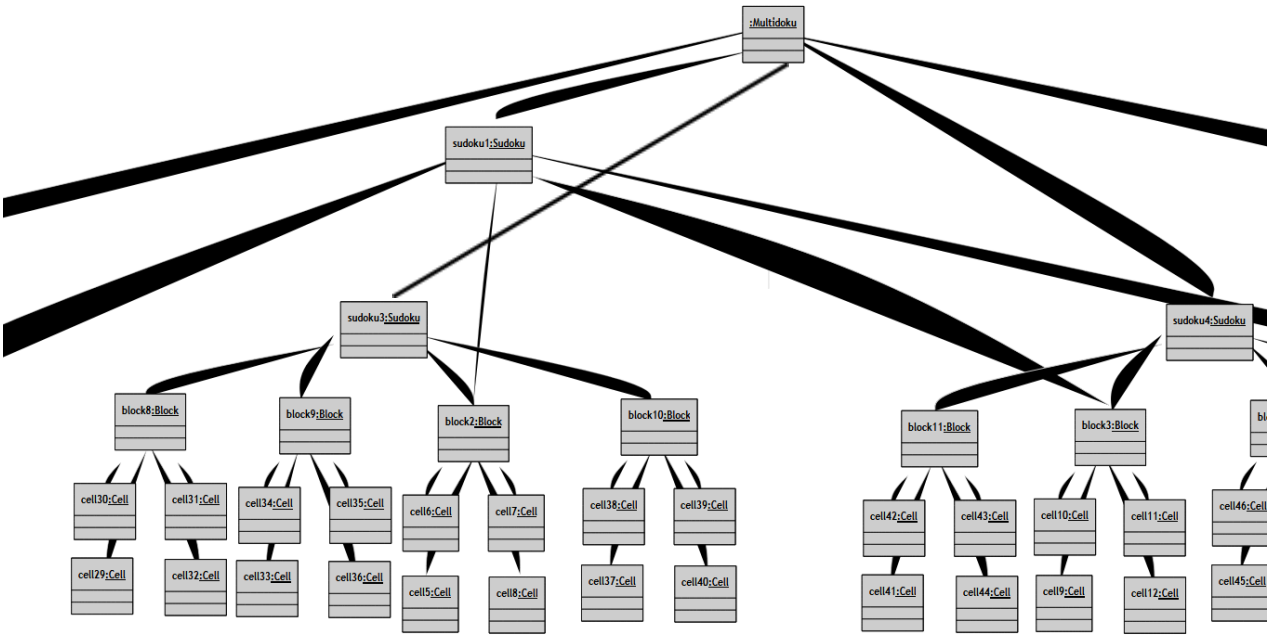
Une Cell contient un int représentant l'index de son symbole, avec NOSYMBOL (-1) si elle n'en a pas et elle est modifiable ou immuable.
Ce système d'index de symbole nous permettra de séparer l'algorithmie de l'affichage. Avec les Solvers qui utiliseront les index dans leur algorithme et à l'affichage on pourra remplacer l'index par n'importe quel type de symbole, comme des nombres, des lettres, des smileys etc.

Pour les contraintes, nous implémenterons une interface IConstraint pour pouvoir ajouter facilement des contraintes.

Idem pour les Solvers avec l'interface Solver.

Enfin pour l'utilisation de la généricité de Java, nous avons utilisé des classes implémentant l'interface List<E>.

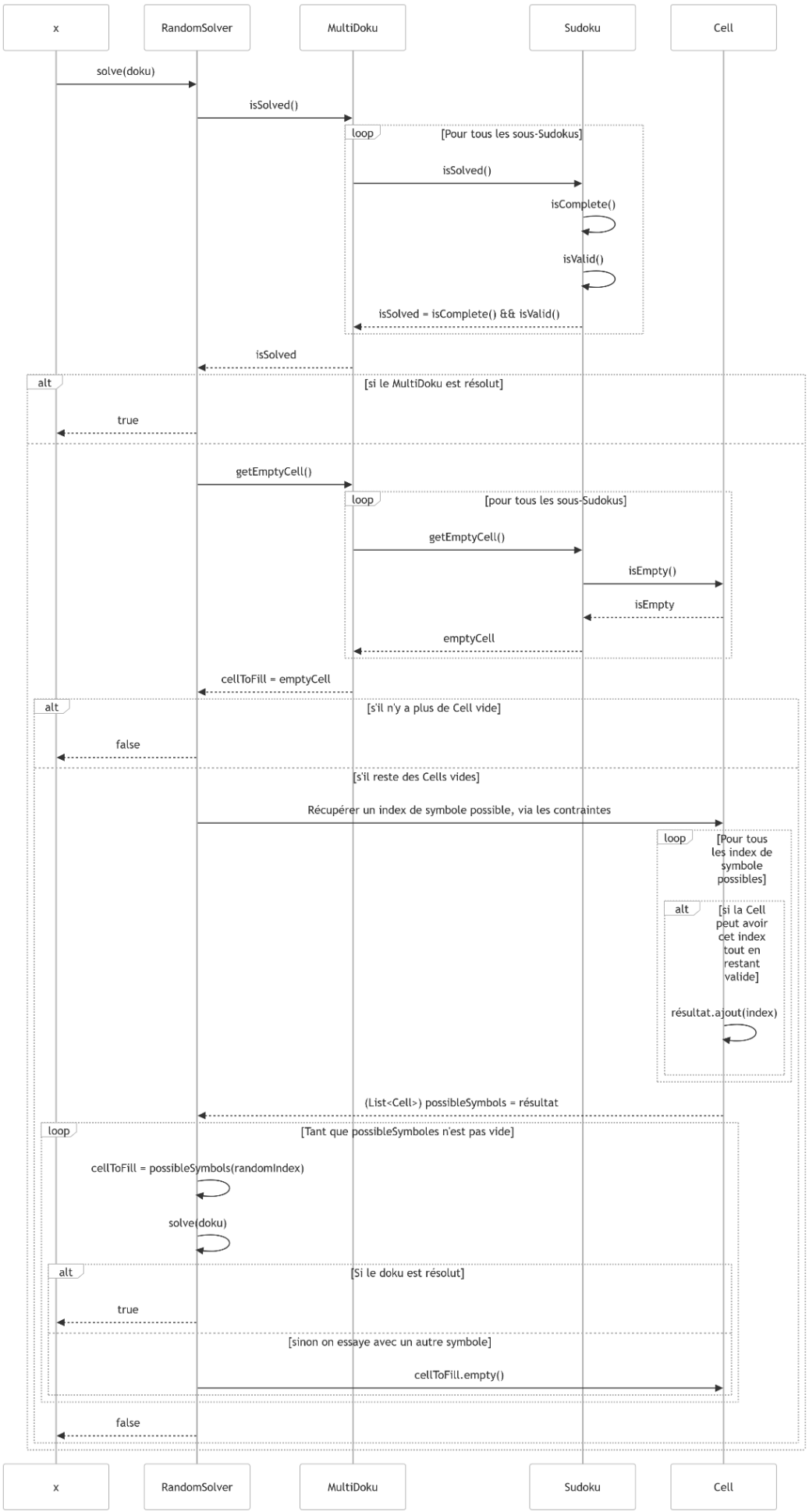
Pour aider à la visualisation d'un MultiDoku complexe avec cette conception, voici une partie du diagramme d'objet d'un MultiDoku en forme de X de taille 2 (en entier à l'Annexe 3) :



IV.2 - Diagramme de séquence (DSEQ)

Pour chaque cas d'utilisation, un diagramme de séquence est réalisable.

Mais nous n'en avons fait qu'un, celui qu'on a trouvé le plus utile, le DSEQ d'un Solver qui utilise un algorithme de backtracking.


x
RandomSolver
MultiDoku
Sudoku
Cell
solve(doku)
isSolved()
loop
[Pour tous les sous-Sudokus]
isSolved()
isComplete()
isValid()
isSolved = isComplete() && isValid()
isSolved
alt
[si le MultiDoku est résolut]
true
getEmptyCell()
loop
[pour tous les sous-Sudokus]
getEmptyCell()
isEmpty()
isEmpty
emptyCell
cellToFill = emptyCell
alt
[s'il n'y a plus de Cell vide]
false
[s'il reste des Cells vides]
Récupérer un index de symbole possible, via les contraintes
loop
[Pour tous les index de symbole possibles]
alt
[si la Cell peut avoir cet index tout en restant valide]
résultat.ajout(index)
(List<Cell>) possibleSymbols = résultat
loop
[Tant que possibleSymboles n'est pas vide]
cellToFill = possibleSymbols(randomIndex)
solve(doku)
alt
[Si le doku est résolut]
true
[sinon on essaye avec un autre symbole]
cellToFill.empty()
false
x
RandomSolver
MultiDoku
Sudoku
Cell

V - Annexes

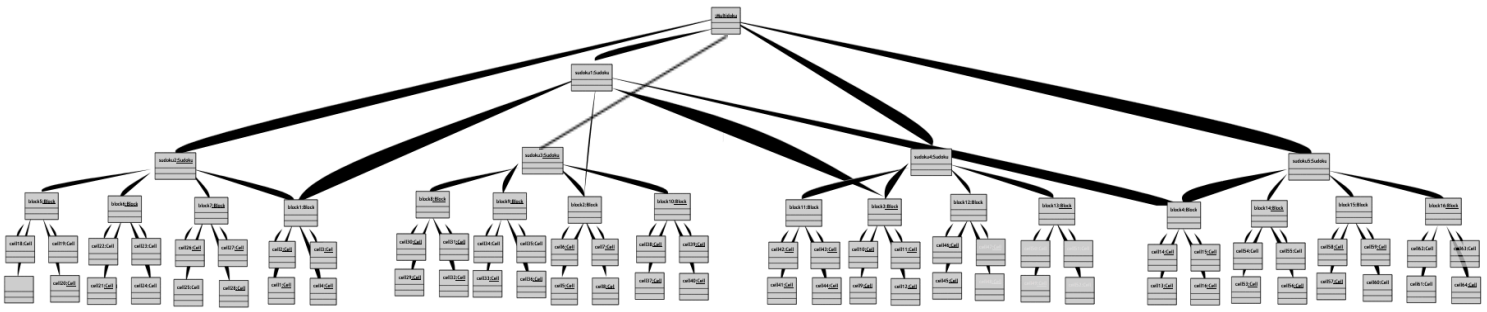
Annexe 1 - Dépôt git

Dépôt git du projet : https://git.ale-pri.com/Ryuk/Sudoku

Annexe 2 - Fonctionnement du Multi joueur

Voir le document Sudoku networking

Annexe 3 - Diagramme d'objet d'un MultiDoku en forme de X de taille 2 :



avec 1 MultiDoku, 5 Sudokus, 16 Blocks et 64 Cells