

# KAKURO SOLVER

## 1 ) Introduction :

Ce projet a pour objectif de développer, un solveur automatique de grilles de Kakuro. L'application permet de charger une grille depuis un fichier, de la résoudre à l'aide d'un algorithme de backtracking, puis d'afficher la grille initiale ainsi que sa solution.

Plusieurs formats de fichiers sont pris en charge, notamment le format imposé par le sujet, un format INI, et un format personnalisé similaire. Le programme utilise le design pattern **Factory** pour sélectionner dynamiquement le lecteur de fichier approprié.

L'affichage des grilles peut se faire soit dans le terminal en mode texte, soit via une interface graphique Swing, pour une présentation visuelle plus claire.

Une interface en ligne de commande permet à l'utilisateur de naviguer à travers un menu interactif offrant plusieurs options : choix d'une difficulté, sélection d'un fichier, affichage de la grille, ou encore visualisation graphique.

Le projet respecte le **principe de responsabilité unique**, chaque classe ayant une responsabilité bien définie (lecture, affichage, résolution, etc.). Plus largement, il s'appuie sur les principes **SOLID** pour garantir un code modulaire, extensible et maintenable.

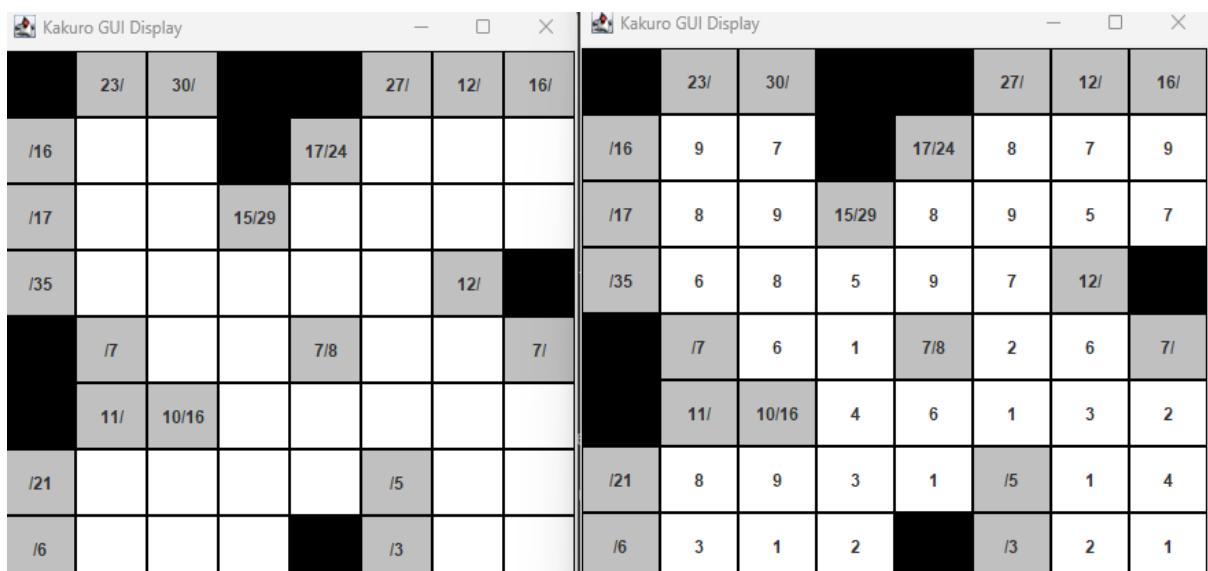


Figure 1 – Exemple GUI d'une résolution d'une grille Kakuro

## 2)-Architecture du code :

L'architecture suit une approche modulaire où chaque classe a une responsabilité claire.

### **1)- Classe Field :**

La classe Field modélise une case de la grille, qui peut être soit ajustable (blanche), soit non-ajustable (noire). Elle contient les informations nécessaires au type de case : les indices across et down pour les cases noires, et la valeur saisie pour les cases blanches. Deux constructeurs permettent d'instancier ces types de cases. Des accesseurs comme getAcross(), getDown(), et setPlayerValue() permettent de manipuler leur contenu.

Cette classe est au cœur de la représentation de la grille.

### **2)- Interface KakuroReader :**

• **KakuroReader** : est une interface qui définit la méthode read(String filename), utilisée pour charger une grille de Kakuro à partir d'un fichier. Cette méthode retourne une matrice de cases (Field[][])) représentant la grille.

L'interface est implémentée par plusieurs classes concrètes, chacune correspondant à un format de fichier spécifique : GridReader pour le format texte classique, GridReaderImposedFormat pour le format imposé avec symboles, et GridReaderIniFormat pour les fichiers au format INI. Ce découpage permet d'adapter facilement la lecture à différents types de fichiers tout en conservant une même interface.

Le choix du bon lecteur est délégué à la classe ReaderFactory, qui applique le **design pattern Factory**.

### **3)- Classe Reader Factory :**

La classe ReaderFactory applique le design pattern **Factory Method** pour sélectionner dynamiquement le bon lecteur (KakuroReader) en fonction du nom du fichier. Par exemple, elle retourne GridReaderImposedFormat si le nom contient "imposed", ou GridReaderIniFormat pour un fichier .ini. Par défaut, elle utilise GridReader, qui lit le format texte classique.

Cette architecture respecte les principes **SOLID** : elle suit le principe **Open/Closed** (ajout de nouveaux formats sans modifier le code existant) et le principe **Dependency Inversion** (le programme dépend de l'interface KakuroReader, pas des implémentations concrètes).

ReaderFactory centralise ainsi la logique de sélection dans un code clair, modulaire et extensible.

### **4)- GridReaderImposedFormat :**

GridReaderImposedFormat lit un format de grille basé sur des symboles visuels. Le fichier commence par deux entiers pour les dimensions, suivis de lignes où chaque cellule est séparée par un espace.

Le symbole « # » représente une case noire sans indices, « \_ » une case blanche, et une chaîne comme 12/5 indique une case noire avec des indices across = 12 et down = 5. Le lecteur identifie chaque type de cellule, extrait les valeurs si nécessaire, et construit la grille en conséquence.

Ce format rend les fichiers Kakuro plus lisibles et permet une saisie intuitive des grilles.

## **5)- Classe GridReader :**

GridReader lit une grille au format texte brut. Le fichier commence par deux entiers représentant les dimensions de la grille (lignes et colonnes), puis chaque cellule est codée par des entiers. La valeur « **-1** » désigne une case blanche. Toute autre valeur représente un indice across, immédiatement suivi d'un second entier pour down, indiquant une case noire avec indices. Le lecteur vérifie également la validité des données (dimensions minimales, valeurs incorrectes) avant de construire la grille à l'aide d'objets Field.

## **6)- Classe GridReaderIniFormat :**

GridReaderIniFormat permet de lire une grille décrite dans un fichier au format INI, structuré en sections. Il identifie d'abord la section « **[dimensions]** » pour extraire les lignes et colonnes, puis parcourt la section [cells], où chaque ligne associe des coordonnées « **(row\_col)** » à un type de case (white, black, ou une valeur avec indices). Ce format offre une représentation claire, structurée et facilement modifiable manuellement, idéale pour tester ou créer des grilles personnalisées.

## **7) Classe KakuroSolver :**

KakuroSolver contient l'algorithme de résolution du puzzle, fondé sur une approche récursive de **backtracking**. La méthode **solve(Field[][] grid, int row, int col)** parcourt la grille ligne par ligne, et saute les cases noires (**!isAdjustable()**). Pour chaque case blanche, elle teste les valeurs de 1 à 9.

Chaque valeur est validée par la méthode **valid(...)**, qui vérifie si elle respecte les contraintes du Kakuro : somme correcte (across, down) et absence de doublons. Cette vérification est déléguée à **isValidRow(...)** et **isValidCol(...)**.

Si une valeur est valide, elle est temporairement placée avec **setPlayerValue(...)**, et **solve(...)** est appelé sur la case suivante. En cas d'échec, la case est réinitialisée à 0 (backtracking).

Ce module est totalement indépendant de l'affichage, et se concentre uniquement sur la logique de résolution.

## **8)- Classe KakuroDisplay :**

KakuroDisplay permet d'afficher la grille dans la console en mode texte. La méthode **display(Field[][] grid)** parcourt la grille et appelle **getCellString(...)** pour formater chaque case.

- Les **cases blanches** sont affichées sous forme de [ x ], ou [ ] si elles sont vides.
- Les **cases noires avec indices** affichent les valeurs across et down formatées à l'intérieur.
- Les **cases noires pleines** sont rendues vides visuellement.

Cette classe offre une représentation lisible de la grille dans le terminal, utile pour tester et valider la logique sans interface graphique.

```

Grille initiale :
[ 0\0 ][ 3\0 ][ 6\0 ][ 0\0 ][ 0\0 ][ 0\0 ]
[ 0\4 ][      ][      ][ 7\0 ][ 8\0 ][ 0\0 ]
[ 0\10][      ][      ][      ][      ][ 3\0 ]
[ 0\0 ][ 0\10][      ][      ][      ][      ]
[ 0\0 ][ 0\0 ][ 0\0 ][ 0\3 ][      ][      ]

Solution trouvée avec succès !
Solution found:
[ 0\0 ][ 3\0 ][ 6\0 ][ 0\0 ][ 0\0 ][ 0\0 ]
[ 0\4 ][ 1 ][ 3 ][ 7\0 ][ 8\0 ][ 0\0 ]
[ 0\10][ 2 ][ 1 ][ 3 ][ 4 ][ 3\0 ]
[ 0\0 ][ 0\10][ 2 ][ 4 ][ 3 ][ 1 ]
[ 0\0 ][ 0\0 ][ 0\0 ][ 0\3 ][ 1 ][ 2 ]

```

Figure 2 – Affichage Terminale

### **9)- Classe *KakuroGUI* :**

**KakuroGUI** permet d'afficher une grille de Kakuro dans une interface graphique à l'aide de la bibliothèque Swing. La méthode **show(Field[][] grid)** crée une fenêtre avec une grille composée de JLabel, représentant chaque case.

- Les **cases blanches** sont affichées en blanc, avec leur valeur si elle a été remplie.
- Les **cases noires** sans indices sont entièrement noires.
- Les **cases grilles avec indices** affichent les sommes across et/ou down.

La classe utilise un GridLayout pour adapter dynamiquement l'interface à la taille de la grille. Elle ne gère pas d'interaction utilisateur, mais sert uniquement à **visualiser la grille de manière lisible et esthétique**.

### **10)- Classe *Kakuro* :**

La classe Kakuro constitue le point central de l'application. Elle orchestre le chargement de la grille, son affichage, sa résolution, et la visualisation du résultat.

Elle propose deux constructeurs :

- l'un qui demande à l'utilisateur de sélectionner un fichier via une fenêtre (**JFileChooser**),
- l'autre qui prend directement le nom du fichier en paramètre.

Le fichier est lu à l'aide d'un KakuroReader obtenu via la ReaderFactory, ce qui permet de gérer différents formats de manière unifiée.

La méthode **solveAndDisplay()** utilise le solveur (**KakuroSolver**) pour résoudre la grille, puis affiche la solution avec KakuroDisplay.

Cette classe représente le point d'entrée logique de l'application et illustre bien la séparation des responsabilités entre lecture, traitement et affichage.

```

Veuillez choisir une option :
1. Afficher et voir la solution de tous les puzzles du dossier "puzzles" (affichage terminal)
2. Choisir un fichier à afficher et voir sa solution (sélection via une fenêtre, affichage terminal)
3. Afficher un puzzle choisi en interface graphique (sélection via une fenêtre)
4. Choisir une difficulté et voir la solution du puzzle correspondant (affichage terminal)
5. Quitter
Votre choix :

```

Figure 3 – Menu Principal

### Conclusion du fonctionnement Du Programme :

Le programme Kakuro est conçu pour charger, résoudre et afficher des grilles de Kakuro de manière automatisée. L'utilisateur sélectionne un fichier de grille, qui est traité par la classe **ReaderFactory** : celle-ci applique le design pattern Factory pour choisir dynamiquement le lecteur (**GridReader**, **GridReaderImposedFormat**, ou **GridReaderIniFormat**) selon le format du fichier.

Le lecteur retourne une grille modélisée sous forme d'une matrice de **Field**, chaque objet représentant une case blanche ou noire. La classe **KakuroSolver** applique un algorithme de backtracking pour remplir les cases blanches en respectant les contraintes du jeu (sommes et unicité).

Une fois la grille résolue, elle peut être affichée soit dans la console via **KakuroDisplay**, soit dans une fenêtre graphique avec **KakuroGUI**. La classe **Kakuro** centralise l'ensemble du processus : lecture, résolution et affichage. Chaque composant est indépendant et respecte les principes SOLID, permettant une architecture modulaire, claire et évolutive.

## 3)- Les Choix de modélisation et Architecture :

L'architecture du programme repose sur une approche **clairement orientée objet**, avec une répartition des responsabilités logique et cohérente. Le choix de modéliser chaque case de la grille par une classe **Field** permet de centraliser toutes les informations associées à une cellule (type, indices, valeur), ce qui facilite la manipulation de la grille tout au long du programme.

Pour la lecture des fichiers, j'ai défini une interface **KakuroReader** afin d'unifier le comportement des différents lecteurs, tout en permettant une extension simple à plusieurs formats (txt, imposed, ini). Le choix d'utiliser un **design pattern Factory** dans la classe **ReaderFactory** découle directement de ce besoin de flexibilité : il permet d'instancier dynamiquement le bon lecteur en fonction du fichier, sans modifier la logique principale.

L'algorithme de résolution, implémenté dans **KakuroSolver**, est entièrement découplé des autres composants : il se concentre uniquement sur la logique de backtracking, ce qui garantit sa réutilisabilité. L'affichage est également séparé en deux modules (**KakuroDisplay** pour le texte, **KakuroGUI** pour l'interface graphique), ce qui respecte le principe de responsabilité unique.

Globalement, l'ensemble du projet a été pensé pour respecter les **principes SOLID**, en particulier :

- **Open/Closed** : ajout de nouveaux formats ou modes d'affichage sans modification des composants existants ;
- **Dependency Inversion** : les classes principales dépendent d'abstractions (**KakuroReader**) et non d'implémentations concrètes.
- **Single Responsibility** : chaque classe remplit un rôle précis (lecture, affichage, résolution...) sans empiéter sur les autres responsabilités ;
- **Liskov Substitution** : les implémentations de **KakuroReader** peuvent être utilisées de manière interchangeable sans affecter le fonctionnement global du programme

### Conclusion :

Une des principales ambiguïtés rencontrées concernait le choix de l'architecture logicielle, notamment la manière d'organiser les interactions entre les composants du programme. Plutôt que de coupler directement les classes entre elles (par exemple, laisser le solveur ou l'afficheur gérer eux-mêmes la lecture de fichier), j'ai opté pour une architecture orientée contrôleur, où la classe Kakuro centralise la coordination tout en déléguant chaque tâche à un module spécialisé (lecture, résolution, affichage). Cette approche m'a permis de réduire le couplage entre les classes, de respecter le principe d'inversion des dépendances, et de maintenir une flexibilité maximale. Elle s'est révélée particulièrement pertinente pour gérer l'extensibilité du projet, notamment l'ajout de nouveaux formats de fichier ou de modes d'affichage, sans introduire de dépendances croisées entre les composants métier.

## 4)- Diagramme Des Classes :

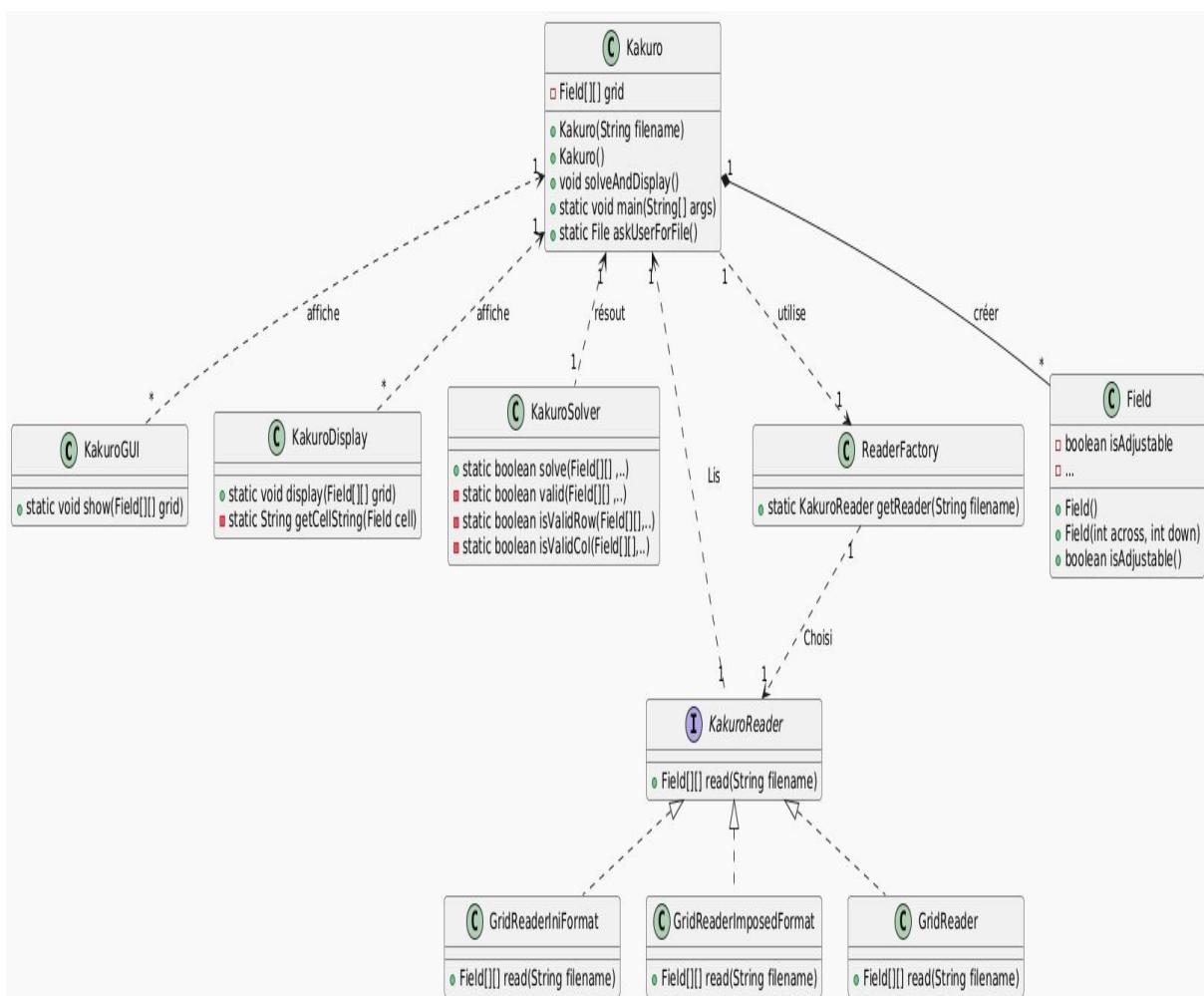


Figure 4 – Diagramme des Classes (UML)

- Ce diagramme représente une version allégée du diagramme de classes UML, en ne montrant que les relations directes et essentielles entre les principales classes du projet. L'UML complet Pour une meilleure lisibilité se trouve dans le fichier : `Projet_Uml_Version_complete`.

## 5)- Affichage graphique (GUI) :

Le projet intègre une **fonctionnalité** sous forme d'interface graphique, accessible uniquement via l'**option 3** du menu principal.

Cette interface, réalisée avec la bibliothèque **Swing**, permet d'afficher visuellement la grille Kakuro dans une fenêtre, avec les éléments suivants :

- Les **cases blanches** (vides ou remplies)
- Les **cases noires avec indices** affichées en **gris** avec le texte down/across
- Les **cases noires sans indices** complètement noires

Le rendu graphique permet de mieux visualiser la structure du puzzle et de comparer facilement l'état **avant et après résolution**, sans passer par le terminal.

Cette fonctionnalité n'était pas obligatoire, elle a été ajoutée en **bonus** pour améliorer l'ergonomie et démontrer la maîtrise de l'affichage Swing.