Rapport Projet POO2

Bontemps Clément TP8
Nemeth Killian TP7

Modélisation du projet Karuko :

Pour notre implémentation du jeu Kakuro, nous avons adopté une approche orientée objet avec une modélisation claire des différents éléments du jeu.

Structure des cellules

- **Cell** : classe abstraite représentant une cellule de la grille.
    - **EmptyCell** : cellule vide à remplir par un chiffre entre 1 et 9.
    - **FilledCell** : cellule déjà remplie par un chiffre de départ, non modifiable.
    - **ClueCell** : cellule contenant les indices de sommes horizontales et verticales.
    - **BlackCell** : cellule noire, bloquée, dans laquelle aucune valeur ne peut être entrée.

Structure de la grille et création

• **Grid** : classe abstraite qui représente la grille complète

- **Grid_Default** : implémentation pour charger une grille depuis un fichier texte standard

- **Grid_Json** : implémentation pour charger une grille depuis un fichier JSON

• **GridFactory** : implémentation du pattern Factory pour créer différents types de grilles

- **GridCreator** : interface qui définit la création des grilles
    - **DefaultGridCreator** : créateur de grilles au format standard
    - **JsonGridCreator** : créateur de grilles au format JSON
- **GridType** : énumération des différents types de grilles (DEFAULT, JSON)

Gestion du jeu et résolution

• **KakuroSolver** : implémentation de l'algorithme de backtracking pour résoudre la grille
• **KakuroGame** : gestion du chargement, de la sauvegarde et du lancement du jeu

Clarification des ambiguïtés du sujet :

Les principales ambiguïtés que nous avons rencontrées étaient :

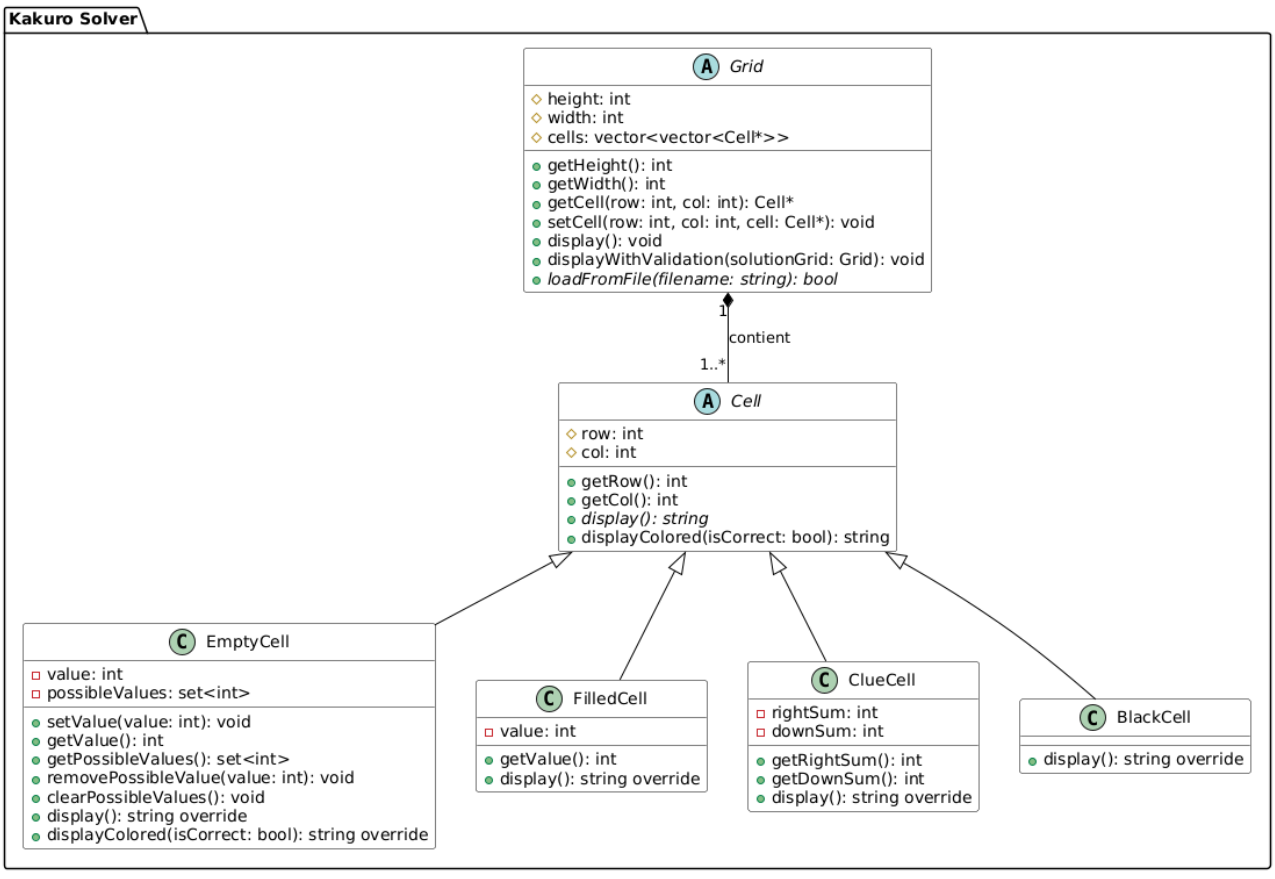
1.**Types de cellules** : Nous avons dû clarifier les différents types de cellules (vides, remplies, indices, noires) et leur comportement.

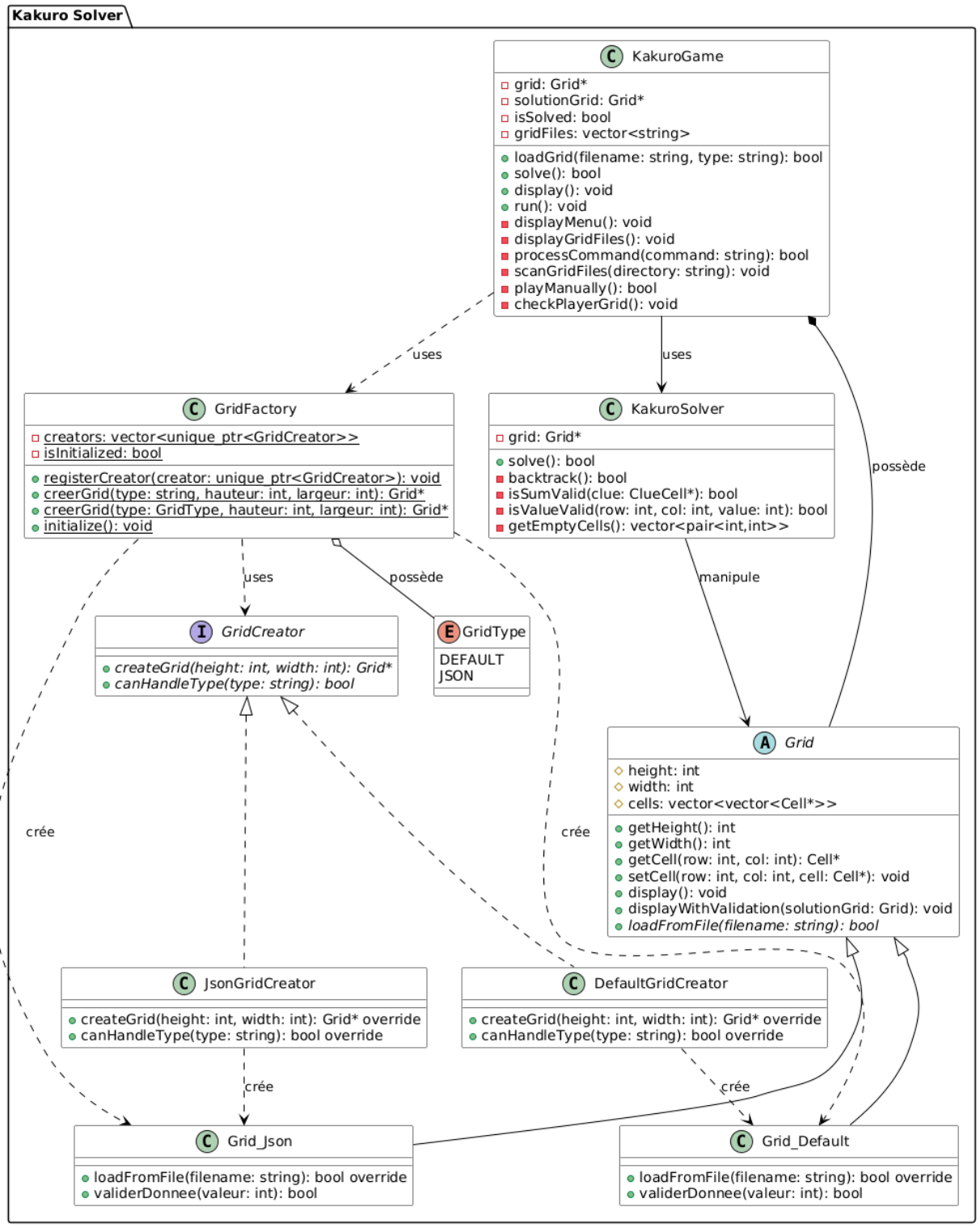
2.**Implémentation du Pattern Factory** : Nous nous sommes référés au cours et à la documentation recommandée pour appliquer correctement ce pattern dans notre projet.

3.**Fonctionnalités supplémentaires** : Nous avons implémenté deux fonctionnalités additionnelles :

- Système d'indices : aide le joueur en remplissant une case correctement à la demande
- Système de vérification par couleurs : utilisation de couleurs pour indiquer si les cases remplies sont correctes (vert) ou incorrectes (rouge)

2) Diagrammes de classes :




Kakuro Solver
KakuroGame
- grid: Grid*
- solutionGrid: Grid*
- isSolved: bool
- gridFiles: vector<string>
+ loadGrid(filename: string, type: string): bool
+ solve(): bool
+ display(): void
+ run(): void
- displayMenu(): void
- displayGridFiles(): void
- processCommand(command: string): bool
- scanGridFiles(directory: string): void
- playManually(): bool
- checkPlayerGrid(): void
uses
uses
possède
GridFactory
- creators: vector<unique_ptr<GridCreator>>
- isInitialized: bool
+ registerCreator(creator: unique_ptr<GridCreator>): void
+ creerGrid(type: string, hauteur: int, largeur: int): Grid*
+ creerGrid(type: GridType, hauteur: int, largeur: int): Grid*
+ initialize(): void
KakuroSolver
- grid: Grid*
+ solve(): bool
- backtrack(): bool
- isSumValid(clue: ClueCell*): bool
- isValueValid(row: int, col: int, value: int): bool
- getEmptyCells(): vector<pair<int,int>>
uses
possède
manipule
GridCreator
+ createGrid(height: int, width: int): Grid*
+ canHandleType(type: string): bool
GridType
DEFAULT
JSON
Grid
height: int
width: int
cells: vector<vector<Cell*>>
+ getHeight(): int
+ getWidth(): int
+ getCell(row: int, col: int): Cell*
+ setCell(row: int, col: int, cell: Cell*): void
+ display(): void
+ displayWithValidation(solutionGrid: Grid): void
+ loadFromFile(filename: string): bool
crée
crée
JsonGridCreator
+ createGrid(height: int, width: int): Grid* override
+ canHandleType(type: string): bool override
DefaultGridCreator
+ createGrid(height: int, width: int): Grid* override
+ canHandleType(type: string): bool override
crée
crée
Grid_Json
+ loadFromFile(filename: string): bool override
+ validerDonnee(valeur: int): bool
Grid_Default
+ loadFromFile(filename: string): bool override
+ validerDonnee(valeur: int): bool

3) Explication des algorithmes principaux sous forme synthétique :

Classe Grid

La classe Grid représente une grille de cellules et gère leur manipulation et affichage :

-**Constructeur** : initialise une matrice 2D de pointeurs Cell* à nullptr

-Destructeur : libère chaque cellule individuellement pour éviter les fuites mémoire

- getCell() : accède à une cellule avec vérification de validité des coordonnées

- setCell() : modifie une cellule avec vérification de validité des coordonnées

- display() : affiche chaque cellule ou un ? si elle est nulle

- displayWithValidation() : compare chaque cellule à une grille solution et affiche en vert si c'est correct, en rouge si c'est incorrect
(utilise dynamic_cast sur EmptyCell/FilledCell)

Classe Cell et dérivées

La classe Cell définit une cellule avec sa position :

- Constructeur et destructeur virtuels
- Méthodes getRow() et getCol() pour récupérer les coordonnées
- Méthode virtuelle pure display() pour l'affichage
- Méthode displayColored() pour afficher en couleur selon la validité

Les classes dérivées (EmptyCell, FilledCell, ClueCell, BlackCell) implémentent ces méthodes selon leur comportement spécifique.

Classe KakuroGame

La classe KakuroGame gère le déroulement du jeu :

•**loadGrid(filename, type)** :

- Lit les dimensions depuis un fichier texte ou JSON
- Utilise GridFactory pour créer l'instance appropriée (Grid_Default ou Grid_Json)
- Charge la grille et la grille solution
- Résout la grille solution automatiquement avec KakuroSolver

•**checkPlayerGrid()** :

- Compare la grille du joueur à la solution
- Compte les erreurs
- Affiche les résultats avec un retour visuel

•**solve()** :

- Utilise KakuroSolver pour résoudre automatiquement la grille
- Retourne le succès ou l'échec de la résolution

•**playManually()** :

- Gère l'interface utilisateur pour le mode de jeu manuel
- Permet au joueur de remplir les cases et de demander des indices

•**run()**, **displayMenu()** et **processCommand()** :

- Boucle interactive du jeu avec menu utilisateur
-Traitement des commandes entrées par le joueur

Algorithme de résolution (KakuroSolver)

Notre solveur utilise l'algorithme de backtracking (retour sur trace) :

1.**Backtracking** :

- Récupère la liste des cellules vides
- Pour chaque cellule vide, essaye successivement les valeurs de 1 à 9
- Pour chaque valeur, vérifie si elle respecte les règles du jeu
- Si valide, place la valeur et continue récursivement
- Si aucune solution n'est trouvée, revient en arrière et essaye une autre valeur

2.**Validation des valeurs** :

- Vérifie que la valeur est unique dans sa séquence (ligne/colonne)
- Calcule les sommes partielles et vérifie qu'elles ne dépassent pas les indices
- Vérifie que les sommes sont correctes quand une séquence est complète

4) Répartition du travail

Pour ce projet en binôme, nous avons réparti le travail de la façon suivante :

•**Conception UML** : travail collaboratif
•**Implémentation du code** : principalement réalisée par Clément par Killian
•**Rédaction du rapport** : principalement réalisée par Killian aidé par Clément