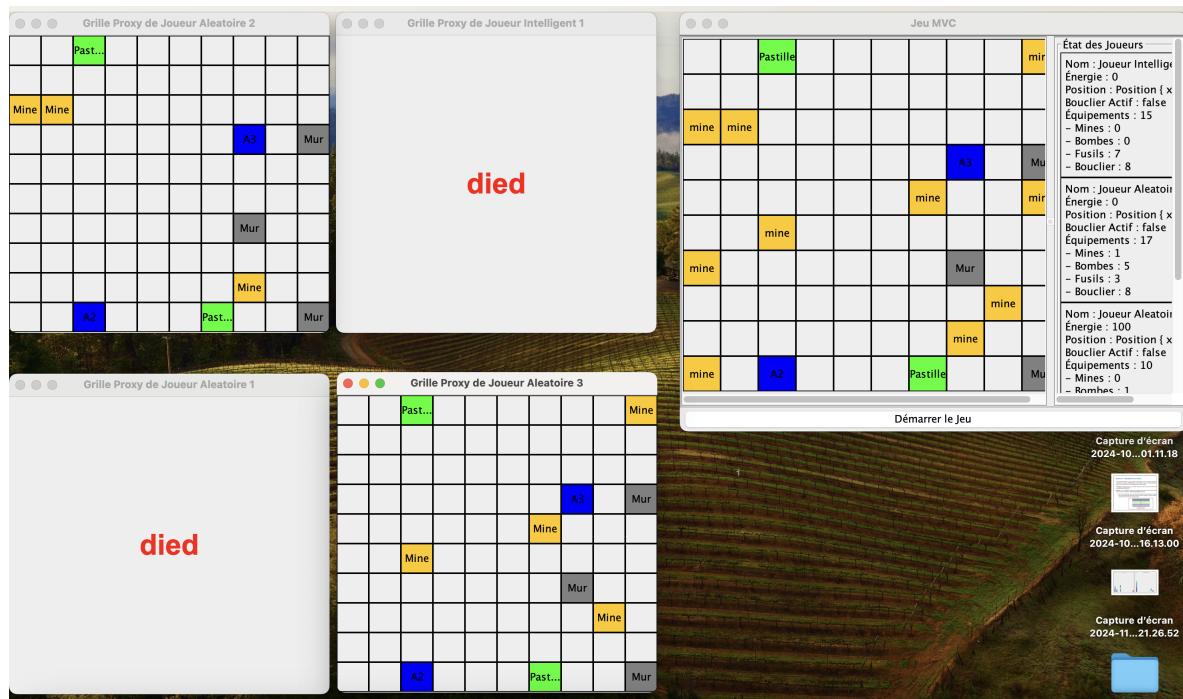


Jeu de stratégie - Méthodes de Conception 2024-2025

mis en page par
Youssouf DIARE 22008756,
Mohamed El Mamy EMAM 22019076,
Boubacar Sadio DIALLO 22211641,
Franck OLANGASSICKA 22112035
L3 info,
Groupe 2A M. Yann Mathet

2 décembre 2024



Sommaire

1	Introduction	3
2	Règles du jeu	3
3	partie réalisée	3
4	Éléments principaux du jeu	3
4.1	Classe : StrategieGrilleAleatoire (dans <i>model.strategie</i>)	3
4.1.1	Rôle :	3
4.1.2	Constructeur :	4
4.1.3	Utilisation dans le projet :	4
4.2	Classe : Main (dans <i>vue</i>)	4
4.2.1	Rôle :	4
4.2.2	Méthode principale :	4
4.2.3	Utilisation dans le projet :	4
4.3	Classe : VueJeu (dans <i>vue</i>)	4
4.3.1	Méthodes principales :	5
4.4	Classe : JeuController (dans <i>controller</i>)	5
4.4.1	Méthodes principales :	5
4.5	Classe : ConfigDuJeu (dans <i>model</i>)	5
4.5.1	constructeur	5
4.5.2	Attributs clés :	5
4.5.3	Méthodes principales :	5
4.6	Classe : Cellule (dans <i>model.grille</i>)	5
4.6.1	Attributs clés :	6
4.6.2	Méthodes principales :	6
4.6.3	Interface : TypeCellule	6
4.6.4	Utilisation dans le jeu :	6
4.7	Classe : Grille (dans <i>model.grille</i>)	6
4.7.1	Méthodes principales :	6
4.8	Classe : GrilleProxy (dans <i>model.grille</i>)	6
4.8.1	Méthodes principales :	7
4.9	Classe : Joueur (dans <i>model.joueur</i>)	7
4.9.1	Méthodes principales :	7
4.10	Résumé des interactions clés	7
5	Diagramme du Projet	8
6	Design pattern implémentés	8
6.1	Observer	8
6.1.1	Utilisation dans le projet :	8
6.1.2	Classes impliquées :	9
6.2	MVC (Modèle-Vue-Contrôleur)	9
6.2.1	Utilisation dans le projet :	9
6.2.2	Interactions clés :	9
6.2.3	Avantages :	9
6.3	Diagramme de classe de MVC - Observer	10
6.4	Stratégie	10
6.4.1	Utilisation pour le comportement des joueurs :	11

6.4.2	Algorithme pour le joueur intelligent	11
6.4.3	Utilisation pour le remplissage initial de la grille :	12
6.4.4	Avantages du pattern Stratégie :	12
6.4.5	Diagramme de classe : Strategy	13
6.5	Proxy	14
6.5.1	Utilisation dans le projet :	14
6.5.2	Avantages :	14
6.5.3	Classes impliquées :	14
6.5.4	Diagramme de classe :Proxy	15
7	Difficulté rencontrée	15
8	Conclusion	16

1 Introduction

Ce rapport présente la conception et la réalisation d'un jeu de stratégie en utilisant une approche modulaire et des design patterns. Le projet vise à offrir une solution modulaire, extensible, et maintenable.

2 Règles du jeu

Le jeu est un combat stratégique au tour par tour, opposant entre 2 et n joueurs sur une grille de taille paramétrable. Nous avons effectué quelques ajustements aux règles originales :

- Les explosions des bombes d'un joueur n'affectent pas les 8 cases voisines.
- Les tirs d'un joueur ne se font pas sur une ligne entière, mais uniquement dans une direction précise (haut, bas, gauche, droite).

3 partie réalisée

on n'avais pas trop d'idée sur quel nom donner à cette partie ,mais retenez juste que dans cette partie nous allons vous expliquer tous ce qui est possible de faire sur notre jeu et certaines fonctionnalité cool que nous avons .

Il est possible de :

- faire jouer autant de joueur de différent type que nous voulons dans la même partie
- au lancement de la partie chaque joueur à sa grilleProxy dans une fenêtre ,mais mon a aussi la grille générale dans une autre fenêtre.
- faire jouer un joueur Humain contre un joueur Aléatoire ou Intelligent qui on des stratégie de jeu différente.

En dehors de tous ce qui est du jeu nous avons fais un fichier XML qui permet de le lancer ;on a aussi écrits des testes permettant de tester certaines de nos methodes et pour finir ,nous avons fait une java doc.

4 Éléments principaux du jeu

Cette section détaille les classes principales organisées par packages.

4.1 Classe : StrategieGrilleAleatoire (dans *model.strategie*)

La classe **StrategieGrilleAleatoire** implémente l'interface **StrategyGrille** et est responsable de la disposition aléatoire des éléments sur la grille au début du jeu.

4.1.1 Rôle :

- Positionner les joueurs, murs, mines de manière aléatoire sur la grille principale.
- Garantir une répartition uniforme des éléments tout en respectant les règles de placement (par exemple, éviter de placer plusieurs éléments sur une même cellule).

4.1.2 Constructeur :

Le constructeur de la classe **StrategieGrilleAleatoire** prend les arguments suivants dans l'ordre pour configurer le placement aléatoire des éléments :

- la grille.
- Le nombre de joueurs Aleatoire
- Le nombre de murs à placer sur la grille.
- Le nombre de pastilles d'énergie à placer sur la grille.
- Le nombre de joueurs Intelligent
- Le nombre de joueurs Humains

4.1.3 Utilisation dans le projet :

- Cette classe est utilisée par le contrôleur ou le modèle pour initialiser la grille avant le début du jeu.
- Elle offre une alternative aux stratégies personnalisées, comme **StrategyGrillePersonnalisee**.

4.2 Classe : Main (*dans vue*)

La classe **Main** est le point d'entrée du programme. Elle initialise l'interface graphique du jeu et lance le processus principal.

4.2.1 Rôle :

- Configurer et démarrer l'application.
- Créer une instance de la classe **VueJeu**, qui gère l'affichage et les interactions avec l'utilisateur.

4.2.2 Méthode principale :

— **main(String[] args)** :

- Initialise les paramètres du jeu, tels que la taille de la grille
- Instancie la Stratégie d'initialisation de la grille
- Instancie les composants principaux, notamment la **VueJeu**.
- Démarrer l'application en appelant les méthodes de la vue pour afficher l'interface utilisateur.

4.2.3 Utilisation dans le projet :

- La classe **Main** sert de point d'entrée pour tester et lancer le jeu.
- Elle est utilisée uniquement pour l'exécution et n'intervient pas directement dans la logique du jeu.

4.3 Classe : VueJeu (*dans vue*)

Cette classe gère l'affichage graphique des éléments principaux du jeu.

4.3.1 Méthodes principales :

- **afficherGrille()** : Affiche graphiquement la grille principale en parcourant chaque cellule et en affichant son contenu.
- **afficherEtatJoueurs()** : Affiche les statistiques des joueurs encore en jeu.
- **afficherToutesGrillesProxy()** : Affiche une vue restreinte de la grille pour chaque joueur.
- **somethingHasChanged(Object source)** : Réagit automatiquement aux changements du modèle pour actualiser l'affichage.

4.4 Classe : JeuController (dans controller)

Cette classe orchestre les actions des joueurs et la progression des tours.

4.4.1 Méthodes principales :

- **startTour()** : Déclenche un tour dans le modèle en appelant *jouerPartie()*.

4.5 Classe : ConfigDuJeu (dans model)

Le modèle principal qui gère la logique du jeu.

4.5.1 constructeur

son constructeur prend en paramètre dans cet ordre :

- une grille ,
- un nombre de joueurs
- un nombre de mure
- un nombre de pastille

par ailleurs la classe StrategieGrilleAleatoire hérite d'elle et l'étend

4.5.2 Attributs clés :

- **Grille** : La grille principale.
- **proxies** : Une *Map* associant chaque joueur à sa vue restreinte.
- **joueurs** : La liste des joueurs actifs.

4.5.3 Méthodes principales :

- **getGrille()** : Retourne la grille principale.
- **getProxies()** : Retourne la liste des grilles restreintes des joueurs.
- **jouerPartie()** : Met à jour l'état du jeu pour un tour.

4.6 Classe : Cellule (dans model.grille)

La classe **Cellule** représente une unité de la grille du jeu. Chaque cellule peut contenir un élément spécifique (comme un mur, une mine, ou un joueur) grâce à son attribut principal **typeCellule**.

4.6.1 Attributs clés :

- **typeCellule** : Cet attribut désigne le type de la cellule. Il est défini par une interface appelée **TypeCellule**, qui est implémentée par plusieurs classes représentant les différents éléments possibles sur la grille, tels que :
 - **Mur** : Représente un obstacle infranchissable.
 - **Pastille** : Une source d'énergie que les joueurs peuvent collecter.
 - **Mine** : Une arme qui explose lorsqu'un joueur se déplace sur sa case.
 - **Bombe** : Une arme qui explose après un délai ou lorsqu'elle est déclenchée.
 - **Joueur** : Représente un joueur positionné sur la grille.

4.6.2 Méthodes principales :

- **getTypeCellule()** : Retourne l'objet **TypeCellule** associé à cette cellule, permettant d'identifier son contenu (ex. Joueur, Mur, Bombe).
- **setTypeCellule(TypeCellule type)** : Définit ou modifie le type de la cellule.
- **estVide()** : Vérifie si la cellule ne contient aucun élément. Retourne un booléen.
- **toString()** : Retourne une représentation textuelle de la cellule, généralement utilisée pour afficher le symbole associé à son type.

4.6.3 Interface : TypeCellule

L'interface **TypeCellule** définit les comportements communs pour tous les éléments pouvant être placés dans une cellule. Chaque type d'élément (par exemple, Mur, Pastille, Mine) implémente cette interface et ajoute ses propres propriétés et comportements spécifiques.

4.6.4 Utilisation dans le jeu :

- La classe **Cellule** sert de conteneur pour les éléments de la grille. Elle est utilisée par la classe **Grille** pour gérer et manipuler les éléments positionnés sur la grille.
- Grâce à **typeCellule**, chaque cellule peut contenir des éléments dynamiques, rendant la gestion de la grille flexible et extensible.
- En vérifiant le type d'une cellule, le jeu peut appliquer des actions spécifiques, comme gérer une explosion, un déplacement ou une collecte d'énergie.

4.7 Classe : Grille (dans *model.grille*)

Représente la grille principale.

4.7.1 Méthodes principales :

- **getCellule(int x, int y)** : Retourne le contenu d'une cellule donnée.
- **ajouterJoueur(Joueur joueur)** : Ajoute un joueur à une position donnée.

4.8 Classe : GrilleProxy (dans *model.grille*)

Permet une vue restreinte de la grille principale.

4.8.1 Méthodes principales :

- **getCellule(int x, int y)** : Retourne uniquement les cellules visibles pour le joueur par exemple si la cellule est une mine il vérifie si la mine est au joueur à qui appartient le proxy grâce à la méthode **getJoueur()** de la mine .
- **getTaille()** : Retourne la taille de la grille restreinte.

4.9 Classe : Joueur (dans *model.joueur*)

Représente un joueur dans le jeu.

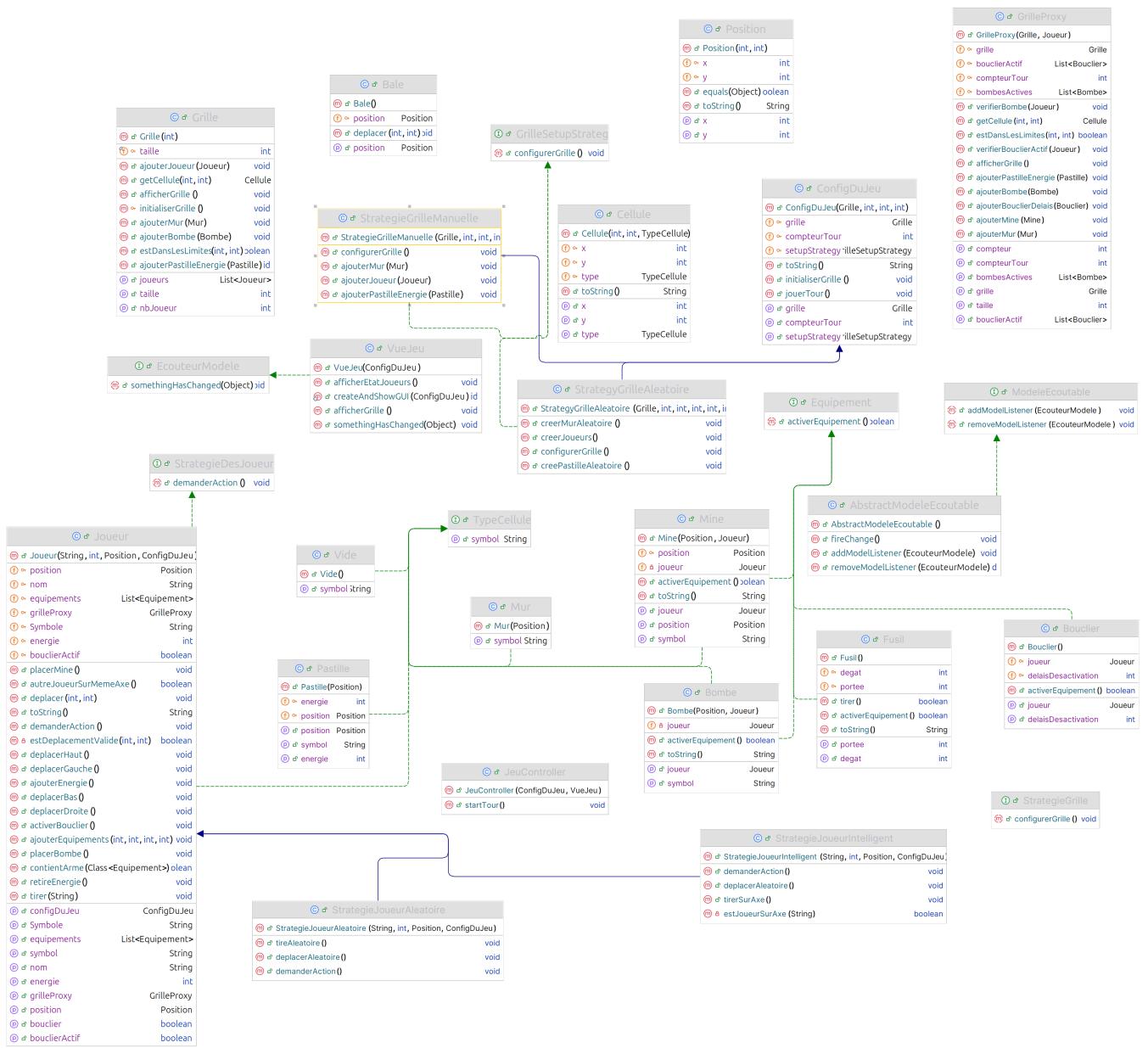
4.9.1 Méthodes principales :

- **deplacer(int direction)** : Déplace le joueur dans une direction donnée.
- **tirer(int direction)** : Tire dans une direction spécifique, affectant les adversaires dans la portée.
- **activerBouclier()** : Protège le joueur contre les attaques pour un tour.

4.10 Résumé des interactions clés

- **Vue <-> Modèle** : La vue utilise les données du modèle pour afficher la grille et les joueurs, et réagit automatiquement aux changements.
- **Contrôleur <-> Modèle** : Le contrôleur déclenche les actions du modèle, comme avancer un tour.
- **Modèle <-> Grilles Proxy** : Le modèle crée et met à jour les grilles proxy pour chaque joueur, garantissant une visibilité limitée.

5 Diagramme du Projet



6 Design pattern implémentés

Dans ce projet, plusieurs design patterns ont été implémentés pour assurer la modularité, la maintenabilité et la flexibilité du code. Voici les principaux patterns utilisés :

6.1 Observer

Le pattern **Observer** est utilisé pour gérer les relations entre les différentes parties du jeu (modèle, vue, contrôleur) afin de maintenir la synchronisation des états.

6.1.1 Utilisation dans le projet :

- Le modèle principal (*ConfigDuJeu*) est l'observé.

- Les vues (*VueJeu*) sont les **observateurs**.
- Lorsque le modèle change (par exemple, après qu'un joueur joue), il notifie automatiquement les vues via l'interface **EcouteurModele**.

6.1.2 Classes impliquées :

- **ConfigDuJeu** : Contient les données principales et notifie les changements à ses observateurs.
- **VueJeu** : Implémente l'interface **EcouteurModele** pour réagir aux modifications du modèle.
- **AbstractModeleEcoutable** : Implémente la logique générique pour ajouter, supprimer, et notifier les observateurs.
-

6.2 MVC (Modèle-Vue-Contrôleur)

Le pattern **MVC** est utilisé pour séparer la logique métier, l'interface utilisateur, et le contrôle.

6.2.1 Utilisation dans le projet :

- **Modèle (Model)** : La classe *ConfigDuJeu* gère les règles et l'état du jeu.
- **Vue (View)** : La classe *VueJeu* affiche les éléments du jeu (grille, joueurs, etc.).
- **Contrôleur (Controller)** : La classe *JeuController* agit comme un intermédiaire entre la vue et le modèle.

6.2.2 Interactions clés :

- La vue observe le modèle et se met à jour automatiquement en cas de changement.
- Le contrôleur gère les actions utilisateur et les traduit en mises à jour du modèle.

6.2.3 Avantages :

- Séparation claire des responsabilités.
- Facilite les modifications et extensions futures.

6.3 Diagramme de classe de MVC - Observer



6.4 Stratégie

Le pattern **Stratégie** permet de définir plusieurs comportements ou algorithmes interchangeables pour un objet. Il est utilisé dans ce projet pour deux aspects majeurs : la disposition initiale de la grille et le comportement des joueurs.

6.4.1 Utilisation pour le comportement des joueurs :

Nous avons défini une interface appelée **StrategieDesJoueur**, qui représente une stratégie de jeu. Cette interface est implémentée par différentes classes représentant des types de joueurs. La méthode clé de cette interface est **demandeAction()**, qui détermine l'action qu'un joueur doit effectuer à chaque tour. Cette méthode est redéfinie par toutes les classes implémentant **StrategieDesJoueur**.

Types de joueurs : Trois types de joueurs sont implémentés, chacun avec un comportement spécifique :

- **Joueur Humain** : Ce type de joueur demande une entrée de l'utilisateur pour choisir une action, permettant normale une interaction directe avec l'interface, mais dans notre cas par manque de temps lorsque l'on lance une partie dans laquelle on à des joueur aléatoire et humain ,quand c'est au tour du joueur humain il est obliger d'effectuer son action depuis le terminal .
- **Joueur Aléatoire** : Effectue des actions de manière aléatoire parmi celles autorisées (déplacement, tir, pose de mine, etc.).
- **Joueur Intelligent** : Suit une stratégie plus avancée, par exemple en choisissant des actions qui maximisent ses chances de victoire (se déplacer vers une pastille d'énergie, attaquer un adversaire proche, etc.).

Méthode demandeAction() : Chaque implémentation de **StrategieDesJoueur** redéfinit la méthode **demandeAction()**, qui est invoquée par le modèle pour décider de l'action du joueur à chaque tour. Voici quelques exemples d'implémentations :

- Pour un **Joueur Humain**, **demandeAction()** attend une commande entrée par l'utilisateur via la console.
- Pour un **Joueur Aléatoire**, **demandeAction()** retourne une action choisie aléatoirement.
- Pour un **Joueur Intelligent**, **demandeAction()** contient un algorithme naïf que nous fait ,dans lequel il tire si il a un joueur en face de lui ,si il n'a pas de tire il active son bouclier ,ou effectue d'autre action que l'on vous montrera dans un algo .

6.4.2 Algorithme pour le joueur intelligent

Le joueur intelligent suit une stratégie basée sur des décisions conditionnelles pour maximiser ses chances de victoire. Voici un exemple d'algorithme implémenté dans la méthode **demandeAction()** :

Algorithm 1: Stratégie du joueur intelligent pour demanderAction

Input: Aucun paramètre
Output: Action choisie par le joueur

```

1 if un ennemi est détecté sur le même axe then
2   if le joueur possède un fusil then
3     Tirer sur l'ennemi avec le fusil;
4     Afficher "Déetecte un ennemi sur le même axe et tire";
5     return ;
6   else
7     Se déplacer aléatoirement;
8     Afficher "Déetecte un ennemi mais ne peut pas tirer. Se déplace
      aléatoirement";
9     return ;
10 if le joueur ne détecte aucun ennemi sur le même axe then
11   if le joueur possède une bombe then
12     Placer une bombe;
13     Afficher "Ne détecte aucun ennemi mais place une bombe";
14     return ;
15   if le joueur possède une mine then
16     Placer une mine;
17     Afficher "Ne détecte aucun ennemi mais place une mine";
18     return ;
19 if le joueur possède un bouclier et que le bouclier n'est pas actif then
20   Activer le bouclier;
21   Afficher "Active son bouclier";
22   return ;
23 Se déplacer aléatoirement;
24 Afficher "Aucune arme ou option disponible. Se déplace aléatoirement";

```

6.4.3 Utilisation pour le remplissage initial de la grille :

Le remplissage initial de la grille utilise différentes stratégies définies par **StrategyGrille**.

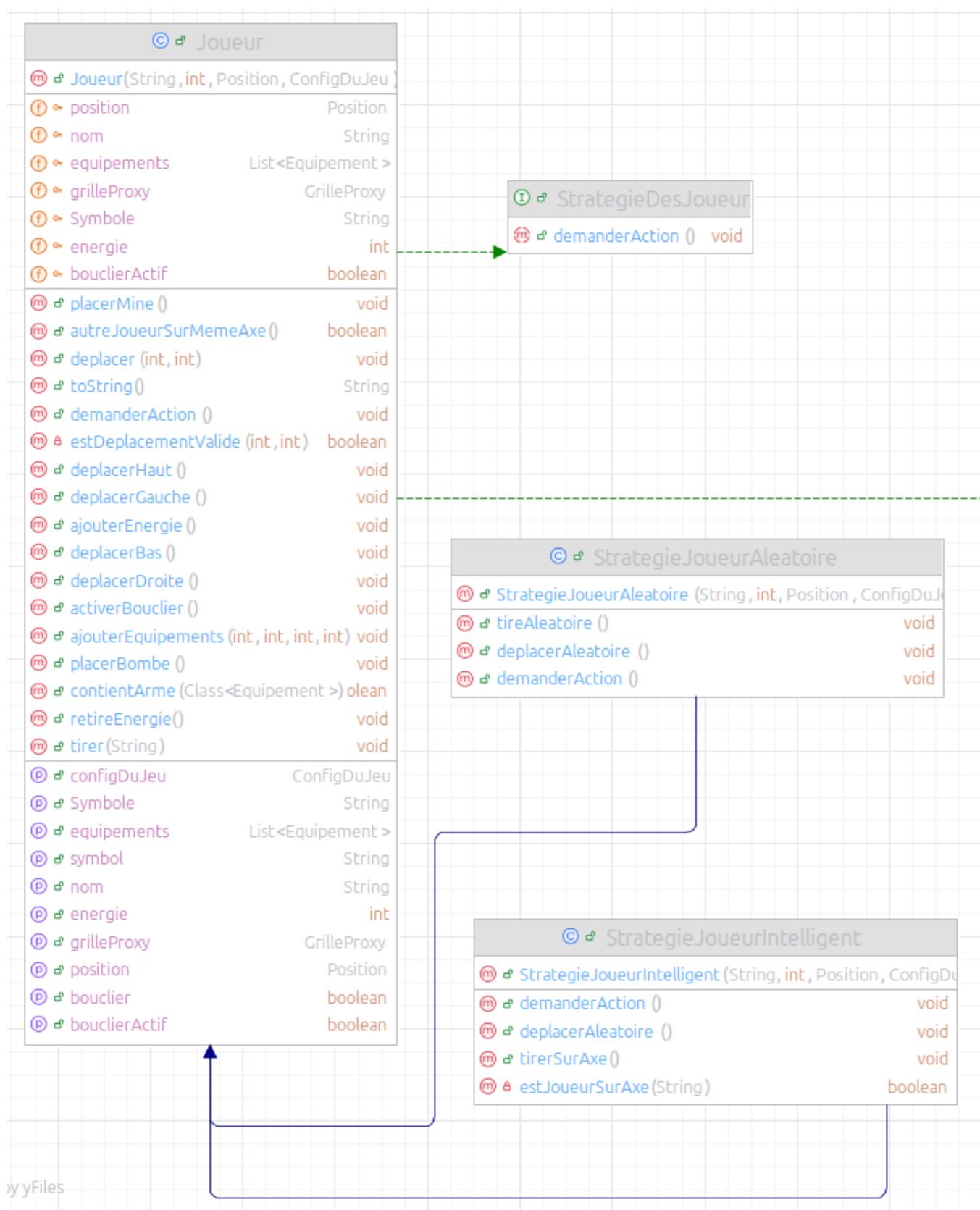
Exemple de stratégies :

- **StrategyGrilleAleatoire** : Positionne aléatoirement les éléments (joueurs, murs, mines) sur la grille.
- **StrategyGrillePersonnalisee** : Permet de configurer manuellement la disposition initiale de la grille.

6.4.4 Avantages du pattern Stratégie :

- Simplifie l'ajout de nouvelles stratégies pour les joueurs ou la grille sans modifier les classes existantes.
- Encourage la réutilisation du code.
- Permet une grande flexibilité et extensibilité pour ajouter des comportements plus complexes (comme un joueur basé sur une intelligence artificielle).

6.4.5 Diagramme de classe : Strategy



6.5 Proxy

Le pattern **Proxy** est utilisé pour fournir une vue limitée de la grille à chaque joueur.

6.5.1 Utilisation dans le projet :

- Chaque joueur dispose d'une **GrilleProxy**, qui est une version restreinte de la grille principale.
- La grille proxy permet de masquer certains éléments (comme les bombes cachées des autres joueurs).

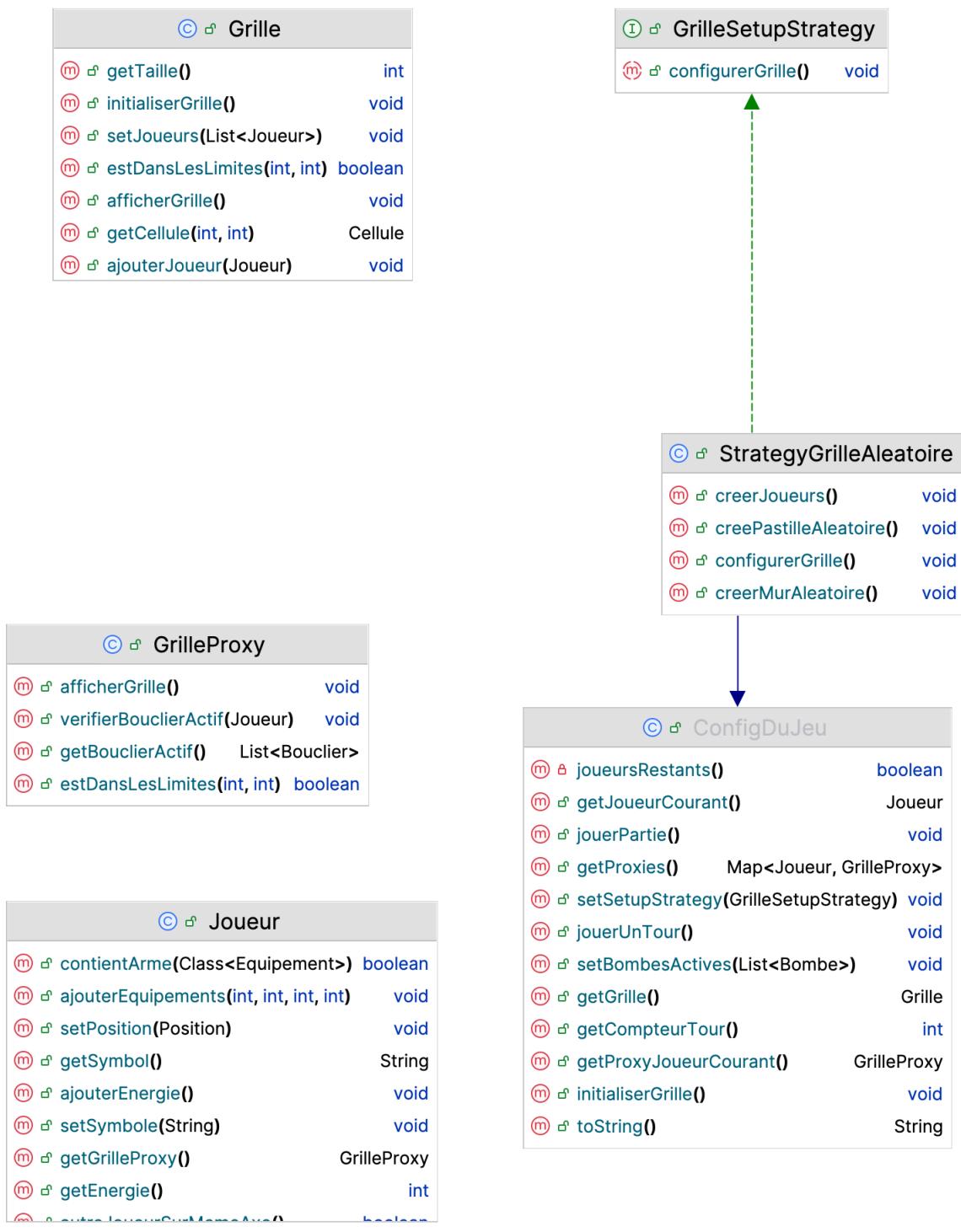
6.5.2 Avantages :

- Réduit les risques d'accès non autorisé aux données sensibles.
- Améliore l'efficacité en évitant de dupliquer les données de la grille principale.

6.5.3 Classes impliquées :

- **Grille** : Représente la grille principale.
- **GrilleProxy** : Fournit une vue filtrée de la grille pour chaque joueur.

6.5.4 Diagramme de classe :Proxy



7 Difficulté rencontrée

ce projet n'a pas été facile .nous avons eu un peu de mal à : implémenté le design pattern proxy . quand nous lançons des partie de jeu impliquant que des joueurs aléatoire ,à certains moment on à une Out Of Bounds Exception (on est entrain de la réglé ... si lors de vos teste vous ne la rencontrez pas c'est que l'on a réglé avant).

lors de vos testes nous vous invitons dans le main à lancer une partie n'impliquant que des

joueurs humain comme ça vous pourrez tester la logique du jeu aussi plus facilement .

8 Conclusion

Grâce à l'utilisation de différents design patterns, tels que **Observer**, **MVC**, **Stratégie** et **Proxy**, nous avons pu garantir une architecture claire, maintenable et extensible.

Les différents modules, allant de la gestion de la grille à l'affichage graphique en passant par les stratégies des joueurs, ont permis de répondre aux exigences du jeu tout en offrant une base solide pour des améliorations futures. Les stratégies définies pour les joueurs et le placement des éléments démontrent la flexibilité et la puissance des patterns de conception implémentés.

Ce projet constitue une base de travail que l'on pourra faire évoluer par la suite par :

- L'intégration de nouvelles stratégies de jeu pour les joueurs intelligents.
- L'amélioration des interactions utilisateur avec une interface graphique plus avancée.
- L'ajout de nouvelles règles et mécaniques de jeu pour enrichir l'expérience.

En conclusion, ce travail nous a permis de mettre en pratique des concepts avancés appris en cours de méthode de conception. Nous espérons que ce projet servira de point de départ pour de futures explorations .Nous avons pour objectif,plus tard d'implémenter un planificateur qui nous permettra d'avoir des stratégies plus intelligente .