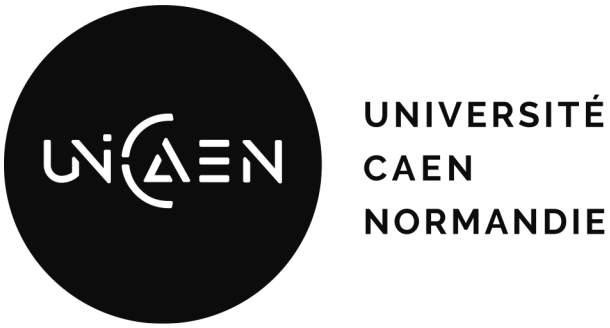
Présentation du Jeu Tour par Tour

Annou Rayane
Madjid Selmane
Mayas Ould Kaci
Ait Ali Belkacem Sonia

December 2, 2024



Contents

1 Introduction

Dans le cadre de ce projet, nous avons développé une application de jeu de combat opposant plusieurs joueurs sur une grille en deux dimensions. Le jeu intègre une mécanique stratégique où les joueurs doivent gérer leurs déplacements, leurs ressources et leurs attaques tout en respectant les règles définies. Ce rapport décrit la conception, les rôles des classes, et les principes de développement suivis.

Objectifs du Projet

Fonctionnalités principales

- Opposer entre 2 et n combattants sur une grille.
- Intégrer des actions variées :Déplacement, Attaque,Défense,Récupération de ressources.

2 Architecture MVC du projet

Le projet suit une architecture MVC (Modèle-Vue-Contrôleur), qui sépare les responsabilités entre les différentes couches pour garantir modularité, extensibilité et maintenabilité.

2.1 Modèle (Model)

Le modèle est au cœur de l'application et gère :

Les données du jeu :

- Combattants (`Combattant`, `Bot`).
- Armes (`Arme`, `Bombe`, `Mine`, `Bouclier`).
- Plateau (`Plateau`) et ses entités (`Mur`, `Vide`, `Energie`, `Munitions`).

La logique métier :

- Gestion des déplacements (`Plateau`, `Combattant`).
- Interactions entre entités (`Combattant`, `Arme`).
- Gestion des ressources comme l'énergie et les munitions (`Energie`, `Munitions`).

La notification des changements :

- La classe `AbstractListenableModel` permet de signaler les modifications aux observateurs (vue ou contrôleur).
- Les classes `Plateau`, `Combattant` et leurs sous-classes utilisent cette fonctionnalité pour informer des changements.

2.2 Vue (View)

La vue est responsable de l'affichage et de l'interaction avec l'utilisateur :

Représentation graphique des éléments :

- `Vue` gère l'affichage principal de l'application.
- `PartieGraphic` affiche graphiquement la partie, le plateau et les combattants.
- `AdapterJTableCombattant` adapte les données des combattants pour être affichées dans une table Swing.

Actualisation dynamique :

- La vue s'actualise automatiquement en réponse aux modifications signalées par le modèle grâce à l'observabilité.

2.3 Contrôleur (Controller)

Le contrôleur agit comme un médiateur entre le modèle et la vue :

Capture des interactions utilisateur :

- La classe `Controller` gère les événements déclenchés par l'utilisateur (clics, touches de clavier).

Traduction en actions sur le modèle :

- Les interactions utilisateur (comme le déplacement d'un combattant ou l'utilisation d'une arme) sont traduites en actions sur le modèle via le contrôleur.

Coordination des mises à jour :

- Le contrôleur veille à ce que les modifications dans le modèle soient correctement reflétées dans la vue.

3 Package Modèle

Utilisation des design patterns

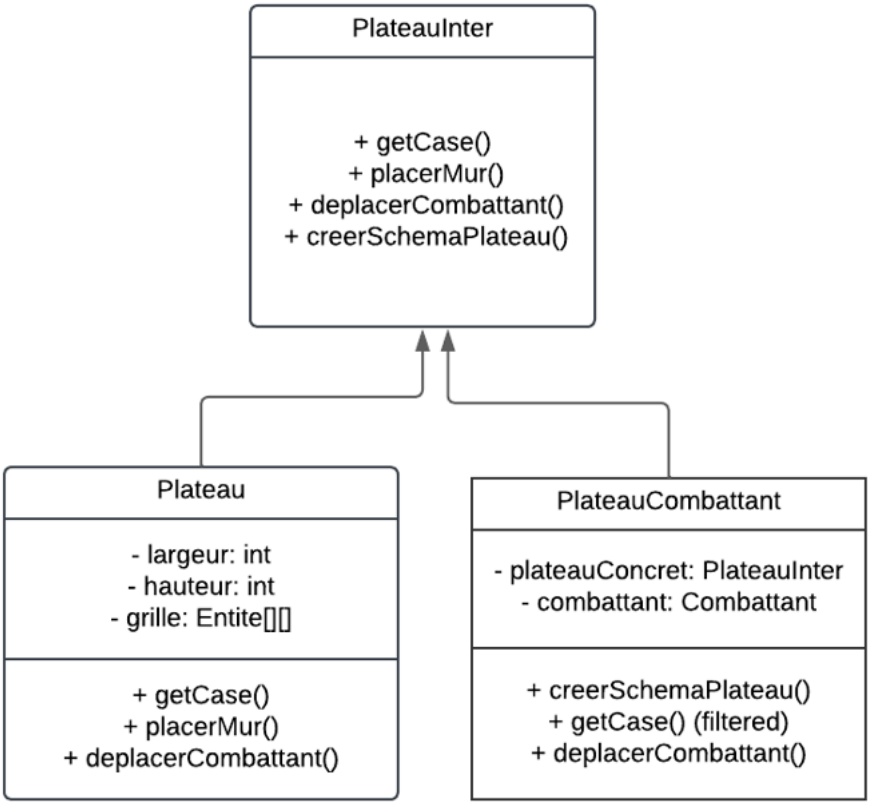
3.1 Proxy

Le `Proxy` est utilisé dans la classe `PlateauCombattant` pour filtrer l'accès aux informations du plateau. Cela permet de masquer certaines entités (comme les armes appartenant à d'autres combattants) et de limiter ce qu'un combattant peut voir sur le plateau.

Un combattant ne doit voir que les entités qui lui sont pertinentes.

Implémentation :

- `PlateauCombattant` implémente l'interface `PlateauInter`.
- Elle agit comme un proxy vers une instance concrète de `Plateau`.
- Les méthodes comme `getCase` ou `getGrille` sont redéfinies pour masquer les entités non visibles par le combattant.


PlateauInter
+ getCase()
+ placerMur()
+ deplacerCombattant()
+ creerSchemaPlateau()
Plateau
- largeur: int
- hauteur: int
- grille: Entite[][]
+ getCase()
+ placerMur()
+ deplacerCombattant()
PlateauCombattant
- plateauConcret: PlateauInter
- combattant: Combattant
+ creerSchemaPlateau()
+ getCase() (filtered)
+ deplacerCombattant()

3.2 Adapter

Le **Pattern Adapter** est appliqué dans les sous-classes de `Arme` pour standardiser l'utilisation d'armes ayant des comportements différents.

Les armes comme les bombes, les mines, et les boucliers ont des comportements spécifiques, mais doivent être utilisables de manière uniforme.

Implémentation :

- La classe `Arme` définit une interface commune avec des méthodes comme `utiliser`, `getDegats`, et `getCoutEnergie`.
- Les sous-classes (`Bombe`, `Mine`, `Bouclier`) adaptent leur propre logique d'utilisation en implémentant ces méthodes.

3.3 Strategy

Le **Pattern Strategy** est utilisé pour rendre les comportements des bots et les stratégies de remplissage du plateau dynamiques.

3.3.1 Comportement des Bots

Les bots peuvent adopter des comportements variés (attaque, défense, etc.).

Implémentation :

- L'interface `Comportement` définit une méthode `agir`.
- Les classes `ComportementAttaquant` et `ComportementDefensif` implémentent cette méthode avec des logiques spécifiques.
- La classe `Bot` utilise une instance de `Comportement` et peut changer de comportement dynamiquement.

3.3.2 Stratégies de Remplissage du Plateau

L a disposition initiale des entités sur le plateau peut varier selon la stratégie choisie.

Implémentation :

- L'interface `RemplirGrille` définit une méthode `remplirGrille`.
- Les classes `RemplissageMurs`, `RemplissageEnergies`, et `RemplissageEquillibre` implémentent cette méthode avec des logiques spécifiques.
- Le constructeur de `Plateau` accepte une instance de `RemplirGrille` pour déterminer la stratégie à utiliser.

3.4 Factory

Le **Pattern Factory** est utilisé pour centraliser la création d'objets complexes ou spécifiques dans le jeu.

Implémentation :

- **Création de Combattants et de Bots :**
  - La classe `CombattantFactory` utilise des méthodes statiques comme `creerCombattant` et `creerBot` pour générer des instances préconfigurées de `Combattant` ou de `Bot` en fonction de leur type (`guerrier`, `sniper`, `tank`, etc.).
  - Ces méthodes intègrent les spécificités de chaque type, telles que les armes, les munitions, et les points de vie, sans que le reste du code ait besoin de gérer ces détails.

4 Package Vue

4.1 Composants Principaux

- `Vue` : Affiche le plateau de jeu sous forme de grille interactive et une table pour les informations des combattants. Elle s'actualise automatiquement grâce au pattern **Observer**.
- `PartieGraphic` : Gère la logique d'une partie en mode graphique, incluant le déroulement des tours, les bombes déposées, et les états des combattants.
- `PartieTerminal` : Une alternative en mode texte, permettant de jouer en console via des entrées clavier.

4.2 Design Patterns Utilisés

- **Observer** : Permet à la vue (`Vue`, `PartieGraphic`) de réagir automatiquement aux changements dans le modèle.
- **Adapter** : Utilisé dans `AdapterJTableCombattant` pour adapter les données des combattants à un affichage dans une table (`JTable`).
- **Controller** : Centralise la gestion des événements utilisateur et leur traduction en actions sur le modèle.

5 Package Observer

5.1 `ModelListener`

Une interface définissant une méthode `somethingHasChanged(Object source)`, utilisée pour informer les observateurs des changements dans le modèle.

5.2 `ListenableModel`

Une interface décrivant les fonctionnalités qu'un modèle observable doit avoir, notamment la possibilité d'ajouter ou de retirer des observateurs.

5.3 `AbstractListenableModel`

Une classe abstraite qui implémente `ListenableModel` et gère une liste d'observateurs. Elle contient la méthode `fireChange()` pour notifier tous les observateurs lorsqu'un changement survient.

6 Conclusion

Ce projet de jeu en tour par tour repose sur une architecture claire et bien structurée, mettant en œuvre des design patterns clés pour assurer modularité, flexibilité et extensibilité. La séparation entre le modèle, la vue et le contrôleur garantit une gestion efficace des données, une interface utilisateur réactive et une synchronisation fluide des interactions. Cette organisation offre une base robuste pour maintenir et enrichir facilement le jeu à l'avenir.