Rapport de Projet

Méthode de conception : Jeu de Stratégie

Georges Dosseh 22012628
Idres Anis 22212192
Aidi Mohammed 22309377
Rezala Ayoub 22309776

Université de Caen Normandie
UFR des Sciences Département
Informatique
3ème année de licence
d'informatique

Table des matières

1 Présentation du projet

Le projet consiste à développer un jeu de stratégie au tour par tour. Chaque joueur évolue sur une grille et utilise diverses armes pour éliminer ses adversaires, avec pour objectif de rester le dernier survivant.
Les joueurs sont éliminés lorsqu'ils n'ont plus de points de vie. Durant leur tour, ils disposent d'un nombre limité d'actions qu'ils peuvent utiliser pour se déplacer, attaquer, activer un bouclier ou réaliser d'autres actions.
Un joueur peut effectuer autant d'actions qu'il le souhaite tant qu'il lui reste des points d'action. Son tour se termine lorsqu'il n'a plus de points d'action ou choisit volontairement de passer sans utiliser le reste de ses actions.

2 Structure MVC (Modèle/Vue/Contrôleur)

L'application est organisée selon l'architecture MVC, répartie en quatre packages principaux :

- **Modèle** : Contient les classes gérant la logique du jeu et les données essentielles, telles que les joueurs, les règles et les éléments de la grille.
- **Vue** : Regroupe les composants d'affichage et l'interface utilisateur, permettant aux joueurs d'interagir avec le jeu via des éléments graphiques.
- **Contrôleur** : Assure la communication entre le modèle et la vue, en capturant les actions des joueurs et en mettant à jour l'affichage en fonction des changements dans le jeu.
- **Patterns** : Regroupe les classes mettant en œuvre divers design patterns (comme le *Proxy*, *Strategy* ou *Factory*), facilitant la modularité, la réutilisabilité et la gestion des comportements spécifiques dans le jeu.

Cette organisation garantit une séparation claire des responsabilités, rendant le code plus maintenable et évolutif.

3 Compréhension de la conception logicielle

3.1 Modèle et patterns

Cette section présente les différents modèles et patterns utilisés dans l'application pour structurer le code, faciliter son évolution, et optimiser les fonctionnalités.

3.1.1 Modèle

Le modèle représente la logique centrale et les données essentielles de l'application. Voici les principaux éléments qui structurent le modèle :

- **Gestion de la grille (`PrincipalGrid` et `Grid`)** : Ces classes encapsulent la logique de la grille principale du jeu, comprenant la gestion des cases (libres ou occupées), des obstacles (`Mur`), et des éléments interactifs comme les bonus (`Pastille`), mines, ou bombes. La classe `RandomGridGeneretor` permet de générer une grille procédurale selon des paramètres définis.
- **Représentation des joueurs (`Player`)** : Gère les attributs d'un joueur tels que ses points de vie, ses munitions et sa position. Les joueurs peuvent interagir avec la grille et utiliser des armes (`Weapon`), comme des fusils (`Fuisil`), des bombes (`Bomb`) ou des mines.
- **Paramétrage global (`GameConfig`)** : Cette classe centralise les constantes du jeu, comme les dommages des armes ou les dimensions de la grille. Les paramètres sont configurables via un fichier externe, rendant l'application adaptable sans recompilation.
- **Interactions dynamiques** : La classe `PrintThread` gère les actions liées à l'affichage ou au suivi en temps réel. Les mouvements des joueurs sont régis par des directions (`Direction`), et les interactions avec les éléments de la grille suivent des règles prédéfinies.

Ce modèle est conçu pour être extensible et maintenable, avec une gestion claire des responsabilités entre les différentes entités du jeu.

3.1.2 Paramétrage du jeu

Le jeu utilise plusieurs constantes que nous souhaitons pouvoir modifier facilement à travers toute l'application. Pour cela, un fichier de configuration texte est utilisé. Chaque ligne du fichier suit un format tel que *"BOMB-DAMAGE=5"*. La classe `GameConfig` lit ce fichier et stocke les valeurs dans des variables statiques, accessibles depuis n'importe quelle classe. Ainsi, pour modifier une constante, il suffit de mettre à jour le fichier de configuration et de relancer l'application, sans nécessiter de recompilation. De plus, cette approche permet à l'utilisateur d'ajuster les paramètres selon ses préférences, sans toucher au code source.

3.1.3 Pattern Factory

L'instanciation des différents types de joueurs est gérée par la classe `PlayerFactory`, qui dispose d'une méthode `build()` retournant une instance de `Player` configurée

avec divers paramètres. Cela permet de créer facilement des archétypes de joueurs en définissant des méthodes spécifiques qui appellent `build()` avec des paramètres prédéfinis, par exemple pour générer un joueur doté d'un bonus de points de vie mais disposant de moins de munitions. Actuellement, les archétypes sont identifiés par une chaîne de caractères dans un attribut de `Player`. Une amélioration envisageable serait de rendre la classe `Player` abstraite et d'utiliser la factory pour instancier des sous-classes représentant ces archétypes.

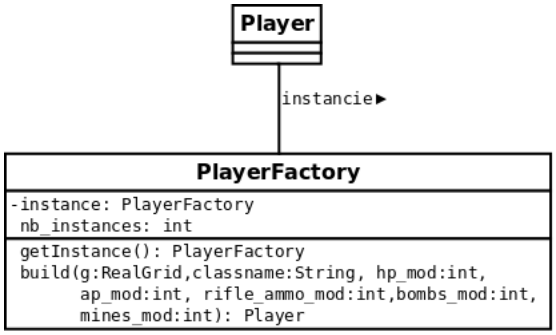

FIGURE 1 – Pattern Factory pour les joueurs

3.1.4 Pattern Proxy

Pour optimiser la gestion des données, une seule grille centralisée (`RealGrid`) stocke l'ensemble des informations du jeu. Toutefois, chaque joueur doit avoir une vue partielle de la grille, limitée aux éléments qu'il a lui-même découverts ou placés (comme les explosifs). Le pattern Proxy permet de répondre à ces besoins en créant une classe `PlayerGrid` associée à chaque joueur. Cette classe est liée à la `RealGrid` unique, mais elle adapte les informations visibles en fonction des permissions du joueur. Par exemple, la méthode `getTileAt()` de `PlayerGrid` intercepte l'appel pour vérifier si le joueur a le droit de voir la case demandée. Si ce n'est pas le cas, une case vide est retournée. De la même manière, `toString()` génère une représentation adaptée à la vision du joueur.

3.1.5 Pattern Strategy

Le pattern *Strategy* est utilisé à plusieurs endroits dans l'application pour offrir une flexibilité dans les comportements et les algorithmes. Voici les principales utilisations :

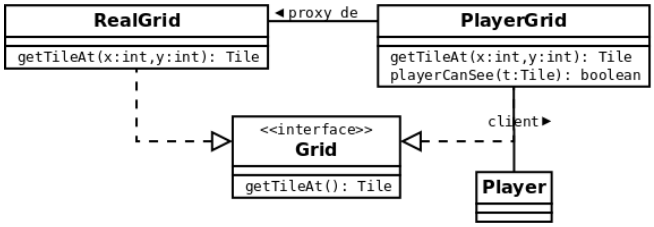

FIGURE 2 – Pattern Proxy pour les vues joueur

Génération de la grille : L'interface `GridStrategy` définit une méthode abstraite `generate()`, permettant de créer différents algorithmes de génération de grilles. Par exemple, la classe `RandomStrategy` implémente cette interface pour générer une grille avec des obstacles et éléments placés aléatoirement. Cette approche facilite l'ajout de nouvelles stratégies (comme une grille symétrique ou basée sur des motifs prédéfinis) sans modifier le code existant.

Comportement des joueurs : Une autre utilisation du pattern *Strategy* est représentée par l'interface `PlayerStrategy`. Elle permet de définir différents comportements pour les joueurs. Par exemple, un joueur humain peut être contrôlé directement, tandis qu'un joueur robot suit une stratégie automatique pour ses déplacements et ses actions.

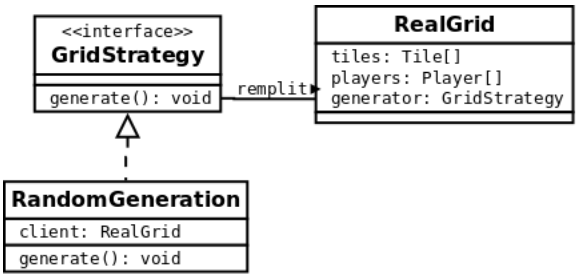

FIGURE 3 – Pattern Strategy pour la génération de grille

3.2 Contrôleur

Le contrôleur de l'application, représenté par la classe `GameController`, gère la logique de jeu, l'ajout de joueurs et le chargement de la grille de jeu. Il agit comme une interface entre le modèle de données et les vues, assurant la mise à jour de l'état du jeu et la gestion des interactions.

La classe `GameController` est responsable de plusieurs tâches essentielles :

— **Gestion de la grille** : Le contrôleur maintient une référence à la grille de jeu via l'objet `PrincipalGrid`, permettant de manipuler et de récupérer l'état de la grille.

— **Ajout de joueurs** : Le contrôleur ajoute des joueurs à la partie et associe une image représentative à chaque joueur grâce à la méthode `addPlayer()`. Les joueurs sont stockés dans un `HashMap` associant chaque joueur à son image.

— **Chargement de la grille** : Le contrôleur est également responsable du chargement de la grille à partir d'un fichier XML via la méthode `loadGrid()`. Cette méthode utilise un parser SAX pour extraire les données du fichier, les analyser et créer la grille avec des couches d'éléments de jeu.

— **Gestion des tuiles** : La méthode `computeTileGrid()` permet de calculer la configuration des tuiles sur la grille en fonction des couches définies dans le fichier XML. Les tuiles peuvent être des cases libres, des murs ou des éléments spéciaux (comme des bonus), et sont associées à des images grâce à la classe `ImagesAdministrator`.

— **Réutilisation de données** : La méthode `copyList()` permet de dupliquer des listes de données afin d'éviter des modifications inattendues et de garantir la stabilité du jeu.

En résumé, `GameController` assure la gestion de la logique du jeu en manipulant la grille et les joueurs, et en garantissant que les données du jeu sont correctement mises à jour et visualisées.

3.3 Vue

La composante **Vue** de l'application est responsable de l'affichage et de l'interaction utilisateur. Elle regroupe plusieurs classes essentielles, permettant une représentation graphique et textuelle des éléments du jeu. Voici un aperçu des principales classes de cette composante :

Classes principales :

— **GameView** : La classe centrale de la vue, `GameView`, est chargée de représenter la grille et les joueurs graphiquement. Elle intègre les éléments visuels nécessaires pour afficher l'état du jeu et réagir aux actions des joueurs.

— **ImagesAdministrator** : Cette classe gère le chargement et l'association des images aux différents éléments du jeu, comme les tuiles ou les joueurs. Elle permet une gestion centralisée des ressources graphiques.

— **ComponentsView** : Elle regroupe les composants visuels, comme les boutons et panneaux, utilisés pour l'interaction avec le joueur. Cette classe est utile pour structurer les éléments de l'interface.
— **ConsoleView** : Fournit une interface textuelle en complément de l'interface graphique, utile pour les messages de débogage ou les interactions minimales.
— **LevelHandlerParser** : Cette classe facilite le chargement des ressources à partir de fichiers, notamment les niveaux de jeu et les images associées aux joueurs.

Écoute des événements :

— **listeners** : Ce package contient des classes comme `AbstractListenableModel`, `ListenableModel`, et `ModelListener`. Ces dernières implémentent un mécanisme d'écoute pour réagir aux changements dans le modèle et mettre à jour la vue en conséquence. Cela assure une synchronisation entre la logique et l'affichage.

En résumé, la composante **Vue** fournit une interface utilisateur riche et réactive, combinant des éléments graphiques, textuels et interactifs. Elle joue un rôle crucial dans l'engagement des joueurs avec le jeu tout en restant découplée de la logique métier.

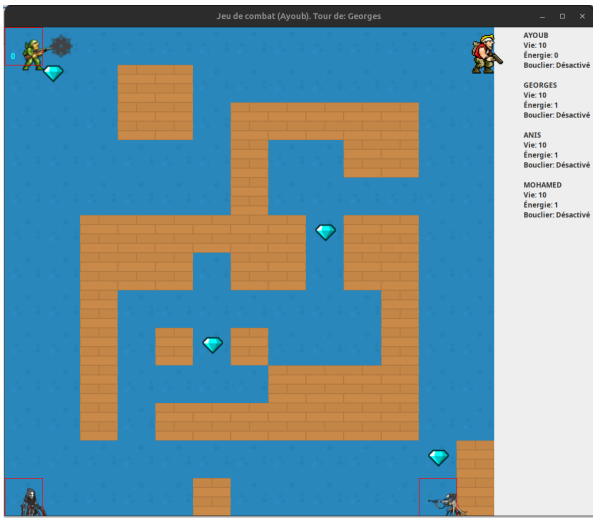

FIGURE 4 – Résultat de la vue (Interface)

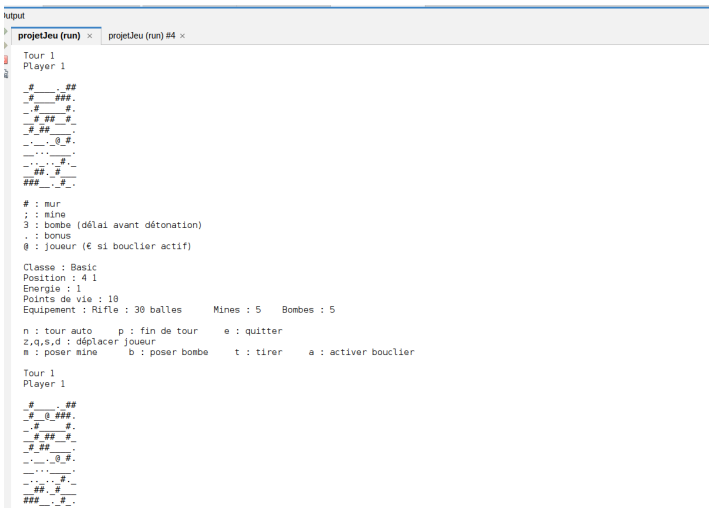

FIGURE 5 – Résultat de la vue (Console)

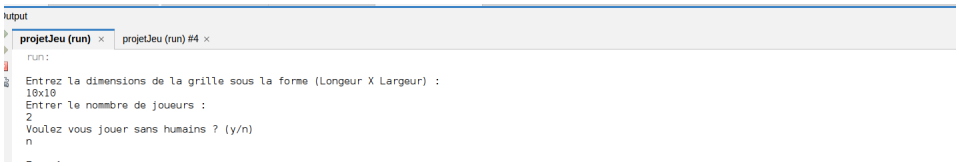

FIGURE 6 – Le choix dans la console

4 Algorithmes non triviaux

Plusieurs algorithmes complexes sont utilisés dans le projet pour gérer des aspects critiques du jeu. Voici une présentation des principaux :

- **Génération procédurale de la grille** : Implémentée dans la classe `RandomGridGeneretor`, cet algorithme génère des grilles de jeu en plaçant des éléments (murs, bonus, obstacles) de manière aléatoire tout en respectant des contraintes comme l'accessibilité des chemins entre les points clés.
- **Gestion des grilles partielles avec le Pattern Proxy** : La classe `GridProxy` limite dynamiquement la visibilité des informations de la grille en fonction du joueur. Chaque joueur voit uniquement les parties découvertes ou accessibles de la grille, nécessitant une gestion fine des permissions.
- **Chargement dynamique des niveaux** : La classe `LevelHandlerParser` analyse les fichiers XML pour charger les niveaux du jeu. Elle transforme les données structurées en objets manipulables dans le jeu, en s'assurant que les configurations sont cohérentes et sans erreurs.
- **Comportement des joueurs contrôlés** : À travers l'interface `RandomStrategy`, le comportement des bots est défini, incluant des décisions contextuelles comme le mouvement, l'attaque, ou la défense. Cet algorithme garantit un robot compétitif et adaptable.
- **Gestion des événements avec des listeners** : Les classes `ListenableModel` et `ModelListener` implémentent un système d'écoute pour synchroniser la vue avec le modèle. Cela permet une mise à jour dynamique de l'interface utilisateur en réponse aux actions des joueurs.

Ces algorithmes illustrent des solutions avancées pour répondre aux défis de la conception et du développement d'un jeu interactif.

5 Compilation et Exécution

5.1 Compilation

Pour compiler le projet en ligne de commande :
Placez-vous dans la racine du projet et tapez les commandes suivantes :

```
ant compile # compilation
ant run     # execution
```

Exécution

Pour exécuter le projet depuis la ligne de commande, accédez au dossier `dist` et tapez les commandes suivantes :

5.1.1 Partie Console

```
java -jar "projetJeuConsole.jar"
```

5.1.2 Partie Graphique

```
java -jar "projetJeuGraphic.jar"
```