Université de Caen Normandie

Licence 2 Informatique

GROUPE 07

Devoir de contrôle continue en

Design patterns & interfaces graphiques

« Jeu de Bataille Navale »

Auteurs :

HADJ BENABDELMOULA Lamia
KHELALFA Selssabil
DIALLO Hammady Ii
DIALLO Marlyatou

4 avril 2025

Table des matières

1 Introduction

Dans le cadre de notre module *"Interfaces Graphiques et Design Patterns"*, nous avons réalisé une version du célèbre jeu de stratégie bataille navale. Ce projet nous a permis de travailler sur deux points importants : une architecture modulaire suivant le pattern MVC(Model View Controller) et une interface utilisateur intuitive avec Swing.

Dans cette première partie de notre rapport, nous allons présenter le contexte et les objectifs du projet, ainsi que les principes de base du jeu bataille navale. Nous décrirons ensuite l'organisation de notre projet, en expliquant l'architecture du programme et les éléments techniques utilisés. Enfin, nous présenterons les expérimentations et les usages que nous avons réalisés avec notre implémentation du jeu bataille navale, et nous conclurons en dressant un bilan de nos travaux et en proposant des améliorations possibles pour notre projet

1.1 Contexte du projet et Objectif du projet

Le projet consiste à développer une version informatique du jeu de bataille navale (touché-coulé). L'objectif principal est de proposer une version personnalisable du jeu, permettant à l'utilisateur de définir les paramètres de la partie : type de joueurs (humain ou IA), taille de la grille, et mode d'interaction (console ou interface graphique).

1.2 Le jeu en quelques mots

La Bataille Navale est un jeu stratégique opposant deux joueurs, chacun tentant de localiser et de couler les navires ennemis. Notre version conserve les principes fondamentaux tout en y ajoutant :

— Une interface visuelle moderne et un système de tour à tour équilibré

— Des options de personnalisation et un placement aléatoire des navires

2 Architecture technique

2.1 Une structure MVC rigoureuse

Nous avons utilisé une organisation claire qui sépare les responsabilités comme suit :
Modèle : Le cœur du jeu bataille navale avec les règles, et la logique du jeu qui totalement indépendant du controller et la view.
Vue : L'interface utilisateur et les composants graphiques là où est affiché la mer

du joueur courant et de son adversaire.
Contrôleur Le médiateur entre le modèle et la vue

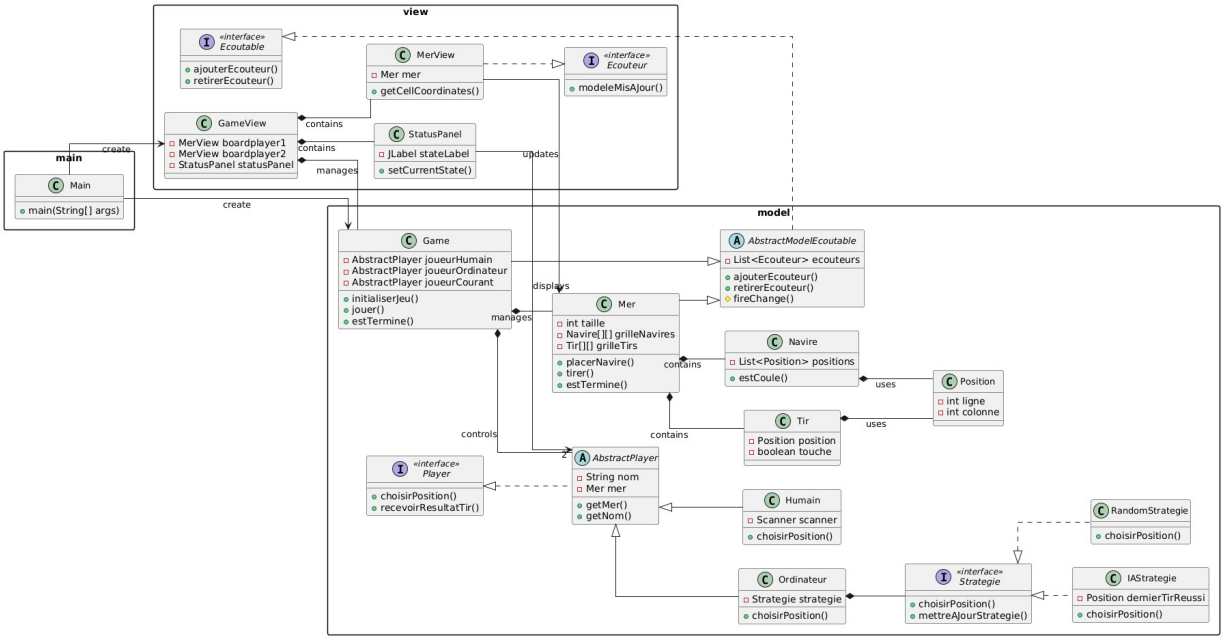

FIGURE 1 – Diagramme de classe UML

2.2 Description des packages et les classes

2.2.1 Couche Modèle

La couche modèle est totalement indépendante de la vue. Elle contient l'ensemble des classes chargées de gérer la logique du jeu. C'est cette couche qui représente les données, les règles et les mécanismes internes du système.

Elle est composée des classes suivantes :

— **Game** : Classe principale du jeu. Elle s'occupe de l'initialisation des joueurs, du placement aléatoire des navires, et contient la boucle principale qui fait tourner le jeu.

— **Mer** : Représente le plateau de jeu. Elle gère le placement des navires, la validation des coups, ainsi que l'état des cases (occupée, libre, touchée).

— **Navire** : Modélise un navire par sa taille et les positions qu'il occupe. Elle contient également la logique de vérification si un navire est touché ou coulé.

— **Tir** : Représente un tir effectué sur une position donnée, en conservant son état (manqué ou réussi).

— **Player** : Interface définissant le contrat de base d'un joueur (humain ou IA), avec les méthodes principales comme `attaquer`, `recevoirTir`, etc.

— **AbstractPlayer** : Classe abstraite permettant de factoriser les comportements communs aux différents types de joueurs (nom, mer, adversaire, etc.).

— **Humain** et **Ordinateur** : Sous-classes concrètes de joueur. Le joueur humain choisit manuellement une position tandis que l'IA utilise une stratégie (aléatoire ou intelligente) pour tirer.

— Autres classes utilitaires : telles que **Position**, **Strategie** etc., qui facilitent la modélisation du jeu.

2.2.2 Couche Vue

Cette couche est responsable de l'affichage graphique du jeu :

— **GameView** : Fenêtre principale contenant les deux grilles.

— **MerView** : Affiche visuellement la mer du joueur.

— **StatusPanel** : Affiche les messages de jeu.

— **AbstractModelEcoutable** / **Ecouteur** / **Ecoutable** : Mécanisme d'observation du modèle (pattern *Observer*).

2.2.3 Couche Contrôleur

Le contrôleur fait le lien entre la vue et le modèle en suivant le pattern MVC :

— **Rôle principal** : Gère les interactions utilisateur et met à jour le modèle

— **Communication** : Reçoit les événements de la vue et les traduit en actions sur le modèle

— **Implémentation** : Dans notre architecture, le contrôleur est intégré dans :
  — La classe **Main** qui coordonne les actions des joueurs
  — Les écouteurs d'événements (MouseListener) dans **MerView**

— **Flux de données** : Lorsqu'un joueur clique sur une case, le contrôleur :
  1. Valide l'action via le modèle
  2. Met à jour l'état du jeu
  3. Notifie la vue des changements via le pattern Observer

2.3 Fonctionnalités implémentées

Nous allons voir ci-dessous les différentes implémentations que nous avons intégrées dans le projet pour atteindre l'objectif :

2.3.1 Placement aléatoire des navires

Les navires sont placés d'une manière aléatoire, ce qui offre une expérience de jeu équilibrée et variée. Notre algorithme prend en compte plusieurs contraintes pour assurer un placement valide :

— **La vérification des collisions** : Avant de placer un navire, le système vérifie qu'il y a assez d'espace libre pour éviter le chevauchement tout en plaçant les navires de grande taille en premier.

— **Le respect des bords** : Tous les navires sont placés dans la zone de la grille sans débordement.

— **L'Orientation aléatoire** : D'une manière aléatoire chaque navire peut être placé horizontalement ou verticalement.

2.3.2 Algorithmes de jeu

On propose aux joueurs deux niveaux de difficultés distincts pour l'ordinateur :

1. **RandomStratégie** (Stratégie Aléatoire) :
   — L'adversaire computer choisit aléatoirement une case parmi les cases non encore attaquées, c'est une implémentation simple mais peu efficace. Elle est parfaite pour les joueurs débutants.
2. **IAStratégie** (Stratégie Intelligente) :
   — Nous avons implémenté un algorithme de chasse : lorsqu'un navire est touché, l'IA concentre ses prochaines attaques autour de cette position.

2.3.3 Personnalisation du jeu

Notre application offre aux utilisateurs la possibilité de paramétrer le jeu comme ils le souhaitent : **La taille de la grille** ex : 6 x 6, **Niveau de difficulté** : le choix des deux joueurs parmi les trois(IA, Random et Humain tout en demandant l'humain son nom)

3 Expérimentations et Usages

3.1 Lancement de l'application et l'affichage en mode console

Pour lancer l'application, vous devez ouvrir un terminal etant dans le dossier « **livraison** » et puis exécuter les commandes suivantes :

1. ***ant init*** : pour initialiser le projet, en créant les dossiers de base bin, ...
2. ***ant compile*** : pour compiler le projet.

3. ***ant run*** : pour lancer l'application sur le terminal(mode graphique / mode console)
4. ***ant javadoc*** : pour générer la Javadoc
5. ***ant packaging*** : pour générer le fichier jar
6. ***ant clean*** : pour nettoyer le projet

Après avoir paramétrer et lancer le jeu en mode console vous devez avoir quelques chose comme ceci :

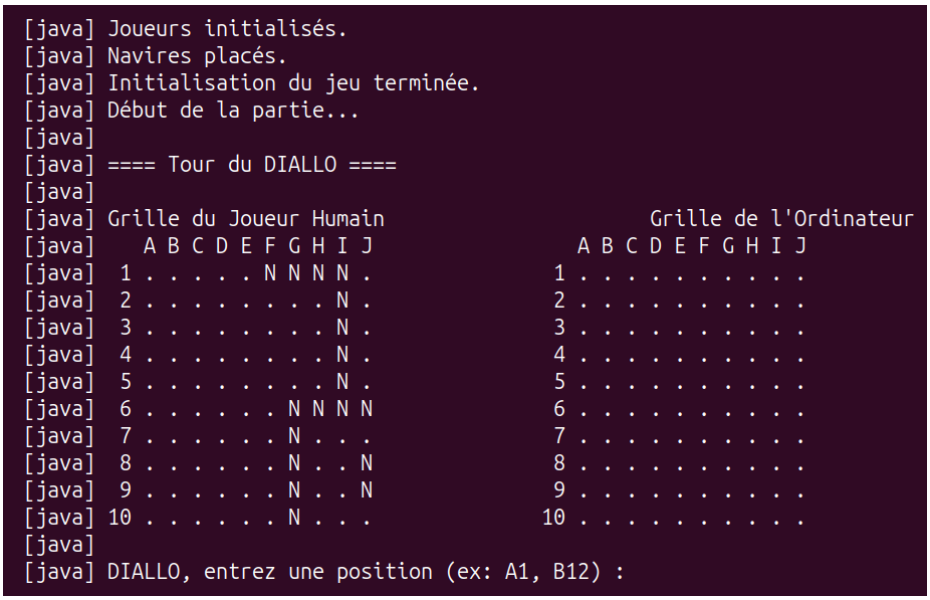

FIGURE 2 – Lancement du jeu en mode console

Et à la fin du jeu voici l'affichage :

```
[java] Position choisie par  DIALLO : (7, 6)
[java] DIALLO a reçu le résultat du tir : à la position (7, 6) : Touché
[java] Grille du Joueur Humain                    Grille de l'Ordinateur
[java]    A B C D E F G H I J                       A B C D E F G H I J
[java]  1 ! . . . ! X N X X !                    1 ! . ! . ! X . . ! .
[java]  2 . . ! ! ! ! ! ! X .                    2 . ! . . . X ! ! . !
[java]  3 . . ! ! . . ! . X !                    3 . . ! . . X . . ! .
[java]  4 . ! ! . . . . . N .                    4 . . . ! . X . ! . !
[java]  5 . . . ! . ! ! ! N !                    5 . . . . ! ! ! . . !
[java]  6 ! ! . . . . X N N N                    6 . . . . . ! X X X X
[java]  7 . ! . ! ! . X . ! !                    7 ! X X X X X . ! . !
[java]  8 . . . . ! . X ! ! X                    8 . X ! . ! ! ! . ! .
[java]  9 ! ! . ! . . X . . X                    9 . X . . . . . ! . !
[java] 10 ! ! . ! . . X . . .                   10 . ! . . ! X X X ! .
[java]
[java] Le jeu est terminé le vainqueur est DIALLO.
[java] ======= Fin du jeu en mode console. ======
```

FIGURE 3 – Les grille à la fin du jeu avec le vainqeur

3.2 Implémentation et fonctionnement de l'interface graphique

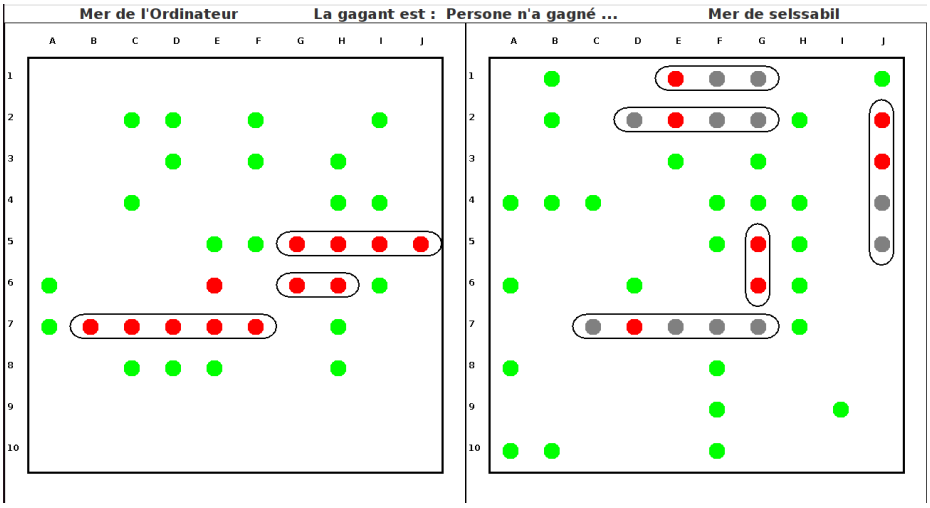

FIGURE 4 – une vue graphique

L'interface graphique du jeu affiche simultanément deux grilles : celle du joueur et celle de l'adversaire . Chaque tir est représenté visuellement pour une meilleure lisibilité : un point vert indique un tir raté, tandis qu'un point rouge signale un tir réussi (un navire a été touché). Lorsqu'un navire est entièrement coulé,cercle noir épais vient entourer l'ensemble de ses cases, permettant au joueur d'identifier rapidement les navires détruits. La grille du joueur montre ses propres navires

ainsi que les tirs adverses, tandis que celle de l'adversaire révèle uniquement les tirs effectués par le joueur, masquant les navires ennemis pour rendre le jeu plus équitable possible. Cette représentation visuelle, est combinée à des mises à jour dynamiques grâce au pattern Observer.

4 Conclusion

Ce projet nous a permis d'acquérir une expérience concrète en architecture logicielle, tout en relevant des défis techniques stimulants pour « - L'équipe de développement » :

— **Synchronisation modèle-vue** : Nous avons opté pour le pattern Observer pour résoudre ce problème de synchronisation .

— **Comportement de l'IA** : Implémentation du L'algorithme qui permet de choisir une position(*choisirPosition*) tout en vérifiant dans la liste des postions mémorisées si de la position du dernier tir réussi y ait et éviter de tirer sur le même endroit deux fois.

4.1 Réalisations majeures

— Code de base bien architecturée et documentée

— Performances stables

— Expérience utilisateur soignée et intuitive

4.2 Amélioration futures

Amélioration	**Description**
Mode réseau	Ajout d'un système multijoueur en ligne
Historique des parties	Visualisation des parties précédentes
Éditeur de navires	Personnalisation des flottes
IA avancée	Implémentation d'une IA plus intelligente

Table 1 – Tableau des évolutions possibles

Références

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[2] Krasner, G. E., & Pope, S. T. (1988). *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1(3), 26-49.

[3] Nystrom, R. (2014). *Game Programming Patterns.* Genever Benning.

[4] Russell, S. J., & Norvig, P. (2016). *Artificial Intelligence : A Modern Approach.* Pearson.