Projet "Jeu de Bataille Navale"

EL FEJER Marwa, DIALO Mamadou Alpha,
KOTIN Mahudo Narcisse, SOW Mariama Saoudatou

09/04/2023

Table de matiére

1 Introduction

La bataille navale est un jeu classique qui a été joué pendant des décennies. Avec l'avènement de la technologie et de l'informatique, ce jeu a été transformé en une application interactive, permettant aux joueurs de jouer en ligne ou contre un adversaire informatique.

Dans ce projet, nous avons concevoir , programmer **un jeu de bataille navale** interactif et créé une interface graphique pour le jeu, permettant à l'utilisateur de jouer contre l'ordinateur en utilisant une interface graphique intuitive et convivial. Ce rapport présentera les différentes étapes de la conception et de la programmation du jeu et de l'interface graphique, en mettant l'accent sur les choix de conception, les problèmes rencontrés et les solutions mises en place pour y remédier. En outre, il fournit également une analyse des fonctionnalités et des performances du jeu.

Interface au debut du jeu:

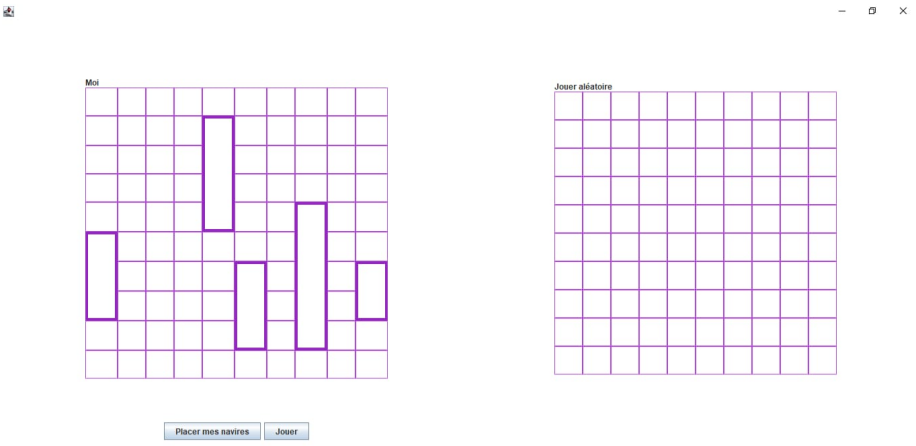

Figure 1: interface vide

Interface pendant le jeu:

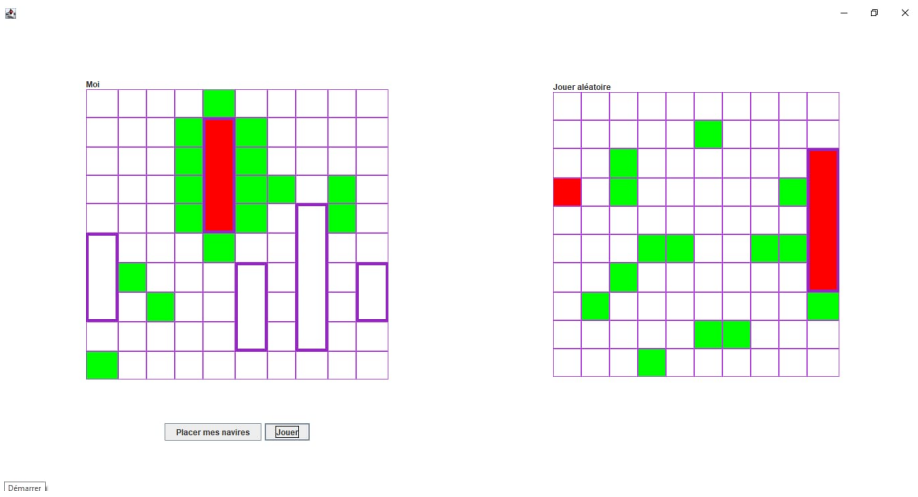

Figure 2: interface pendant le jeu

2 Cahier de charge

2.1 Contextualisation du projet

Le jeu de bataille navale est un jeu de stratégie classique qui a été joué par des millions de personnes dans le monde entier. Il consiste en un plateau de jeu de 10x10 cases sur lequel chaque joueur place ses bateaux, et tente de deviner la position des bateaux de son adversaire afin de les couler. Avec l'avènement de la technologie et de l'informatique, ce jeu a été transformé en une application interactive, permettant aux joueurs de jouer en ligne ou contre un adversaire informatique.

Dans le cadre de notre formation en Licence 2 Informatique, nous avons été chargé de concevoir et de programmer un jeu de bataille navale en utilisant le langage Java et la bibliothèque Java Swing pour l'interface graphique. Le but de ce projet est de mettre en pratique mos connaissances en programmation orientée objet, en algorithmie et en conception d'interfaces graphiques, tout en apprenant à travailler en équipe.

Le développement du jeu nécessite la création d'une structure de données pour représenter le plateau de jeu, ainsi que la mise en place d'un algorithme pour la gestion de la logique du jeu. L'interface graphique devra également être conçue de manière à offrir une expérience utilisateur intuitive et conviviale.

En outre, le projet mettra en avant les compétences de travail d'équipe, de planification et de gestion de projet, car nous travaillerons en groupe pour concevoir et développer le jeu. Nous aurons besoin de communiquer efficacement et de collaborer pour résoudre les problèmes rencontrés et assurer que le projet soit livré dans les délais impartis.

En somme, ce projet de développement d'un jeu de bataille navale en Java avec l'interface graphique Java Swing est une opportunité pour nous de consolider nos connaissances en programmation et en conception d'interfaces graphiques, tout en acquérant des compétences essentielles pour le travail en équipe et la gestion de projets informatiques.

2.2 Travail demandé

Le jeu de la bataille navale consiste en un affrontement naval entre deux joueurs, chacun disposant d'une flotte de navires positionnée sur une grille. Les joueurs tirent chacun leur tour sur une position de la grille de l'adversaire, cherchant à toucher les navires ennemis. Si un navire est touché, le joueur en est informé par un marquage spécifique sur la grille, tandis que les tirs ratés sont également signalés. Le but du jeu est de couler tous les navires adverses avant que l'adversaire ne coule les siens. Le jeu peut être joué en ligne de commande ou avec une interface graphique, cette dernière venant se greffer sur le modèle MVC. Le projet de programmation de la bataille navale doit être conçu de manière à respecter les critères d'évaluation tels que la qualité de l'architecture logicielle, la clarté et la facilité de compréhension du code, la

facilité de maintenance et d'évolution, ainsi que la robustesse.

2.3 Equipe

nous sommes un groupe de 4 etudiants du licence 2 informatique :

- EL FEJER Marwa/22209199 .
- DIALLO Mamadou Alpha/22107614 .
- KOTIN Mahudo Narcisse/22211719 .
- SOW Mariama Saoudatou/22209828 .

3 Conception du projet

3.1 Diagramme uml des packages

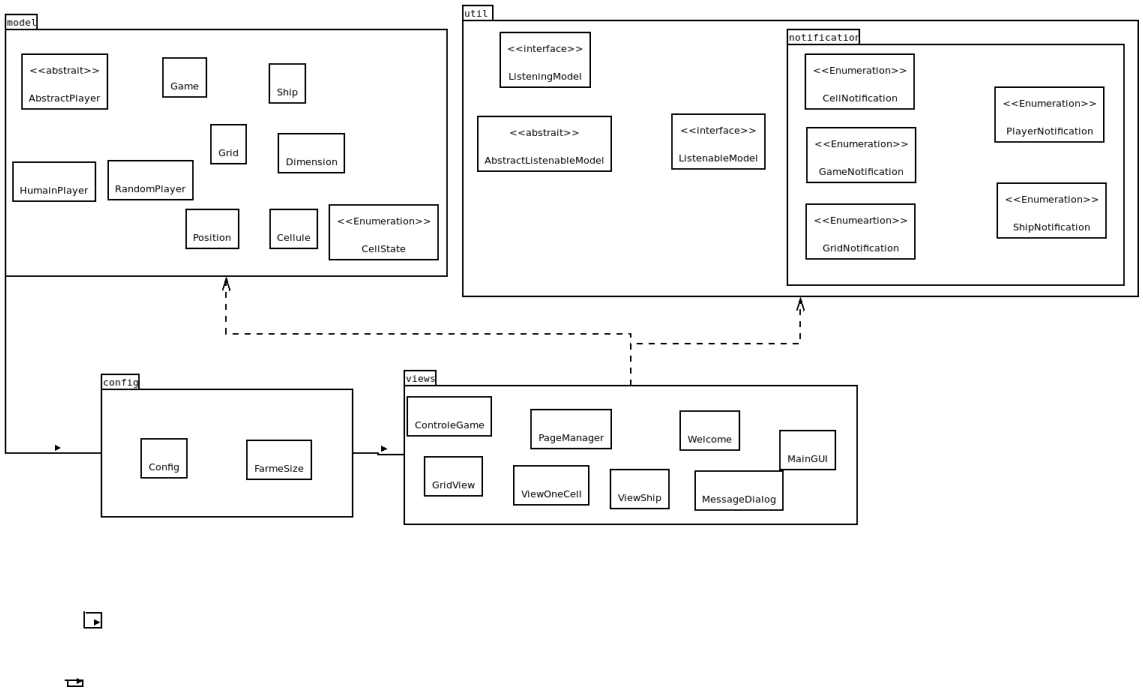

Figure 3: Diagramme des package

Figure 4: Diagramme des classes generale

3.2 Explication de cette conception

Notre projet a été conçu en utilisant une architecture de packages pour faciliter la gestion et la maintenance du code. Nous avons utilisé quatre packages différents :

- Package Model
- package Util
- Package Views
- Package Config

Le package Model contient toutes les classes liées à la logique du jeu et à la gestion des données. C'est là que nous avons stocké toutes les classes qui gèrent la logique du jeu, les classes Grid, Game , Position, Cellule,Dimension, CellState, AbstractPlayer, HumainPlayer, RandomPlayer et Ship. Les classes de ce package sont utilisées par les autres packages pour récupérer ou modifier des données liées au jeu.

Le package Util contient des classes qui permettent de mettre en œuvre le pattern MVC (**Modèle-Vue-Contrôleur**).

Le package Views contient toutes les classes liées à l'interface utilisateur. C'est là que nous avons stocké toutes les classes qui gèrent l'affichage des éléments graphiques, comme les fenêtres de dialogue et les grilles de jeu. Les classes de ce package utilisent les classes du package Model pour récupérer les données liées au jeu et les afficher à l'utilisateur.

Enfin, **le package Config** contient toutes les classes liées à la configuration du jeu. C'est là que nous avons stocké toutes les classes qui gèrent les paramètres du jeu, comme la taille de la grille et le nombre de navires autorisés. Ces classes sont utilisées pour initialiser le jeu avec les paramètres spécifiés par l'utilisateur.

En résumé, notre projet est organisé en packages pour une gestion plus facile et une maintenance efficace. Les packages Model, Util, Views et Config sont tous liés les uns aux autres pour assurer le bon fonctionnement du jeu.

4 Programmation du projet

4.1 Package Model

4.1.1 Diagramme UML des classes

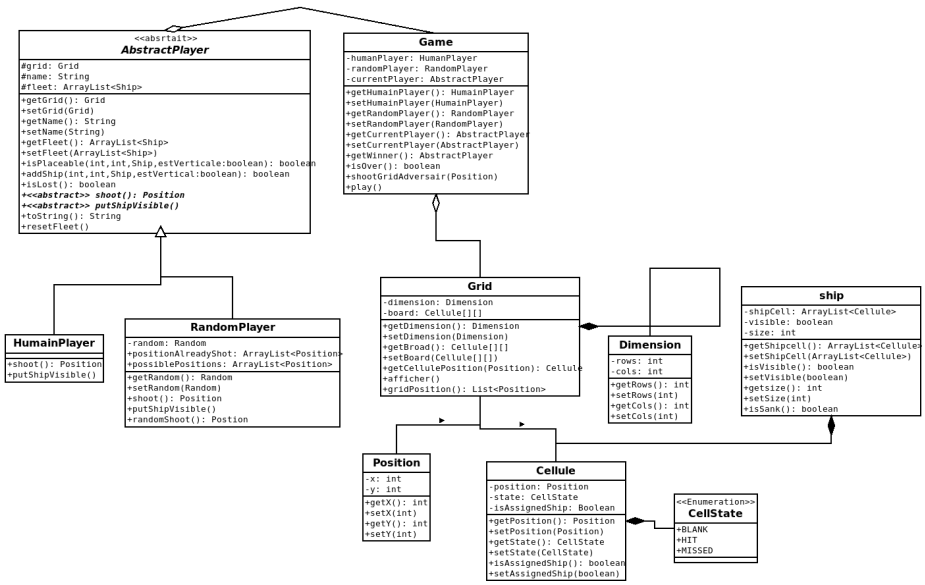

Figure 5: Package Model

4.1.2 Explication

Le modèle d'un jeu de bataille navale repose sur **une grille de taille N x N** , où chaque cellule représente une case sur laquelle un joueur peut placer un navire ou effectuer une attaque. Dans notre version du jeu, chaque joueur dispose d'une grille de taille **10 x 10** et doit placer cinq navires : un porte-avion de 5 cellules, un croiseur de 4 cellules, un contre-torpilleur de 3 cellules, un sous-marin de 3 cellules et un torpilleur de 2 cellules.

Chaque navire peut occuper plusieurs cellules adjacentes sur la grille et peut être orienté horizontalement ou verticalement. Le but du jeu est de découvrir et de détruire tous les navires de l'adversaire avant que celui-ci ne découvre et ne détruise les vôtres.

Notre **modèle** de jeu comprend plusieurs **méthodes** permettant aux joueurs de placer leurs navires, de tirer sur la grille de l'adversaire et de déterminer le gagnant de la partie.

La méthode **"addShip"** est commune aux deux joueurs et permet de placer un navire sur la grille. Elle prend en entrée les coordonnées x et y du navire, le

navire lui-même et un booléen indiquant si le navire doit être placé de façon verticale ou horizontale. La méthode utilise une méthode auxiliaire **"isPlaceable"** pour vérifier si le placement du navire est possible, en parcourant la grille du joueur et en vérifiant si le navire dépasse la taille de la grille ou si une cellule est déjà occupée par un autre navire.

La méthode **"addShipRandomly"** est également commune aux deux joueurs et permet à chacun de placer ses cinq navires de façon aléatoire en utilisant la méthode auxiliaire **"isPlaceable"**.

Chaque joueur dispose d'une méthode **"shoot()"** lui permettant de tirer sur la grille de l'adversaire. Pour le joueur humain, cette méthode permet de choisir une position pour effectuer son tir et retourne la position choisie.Pour le joueur aléatoire, nous avons rendu le tir intelligent. Nous avons d'abord évité de tirer plusieurs fois au même endroit. Pour ce faire, nous sauvegardons toutes les positions de tir possibles dans un tableau. À chaque tir, nous choisissons une position aléatoire dans le tableau puis nous retirons cette position du tableau après le tir. Ensuite, nous avons fait en sorte que pour chaque tir, la position soit choisie en premier parmi les position adjacentes des cellules touchees pour faire en sorte de faire couler les navires plus vite. S'il n'y a pas de telles positions, nous tirons aléatoirement suivant la première méthode.

Nous avons aussi veillé à réduire la complexité algorithmique dans nos méthodes. Cela a été rendue d'autant plus facile grâce à une bonne modélisation. Par exemple, nous avons associé à chaque cellule le navire auquel il est lié. Ainsi, lorsqu'une cellule est touchée, nous retrouvons le navire en une seule opération.

Chaque joueur dispose également d'une méthode **"putShipVisible()"** lui permettant d'afficher ses navires sur la grille. Pour le joueur aléatoire, la méthode permet également d'afficher tous les navires coulés de sa flotte.

Enfin, chaque joueur dispose d'une méthode **"isLost()"** qui retourne un booléen **"true"** si l'intégralité de sa flotte est détruite et **"false"** s'il reste au moins un navire non coulé. Cette méthode utilise la méthode **"isSank()"** de la classe Ship pour vérifier si toutes les cellules d'un navire ont été touchées ou non.

Le déroulement du jeu est géré par la méthode **"play()"**, qui permet de jouer une partie complète du jeu en utilisant les méthodes **"shoot()"**, **"getWinner()"** , **"shootGridAdversaire()"** et **"isOver()**.

4.2 Package Util

4.2.1 Diagramme UML des classes

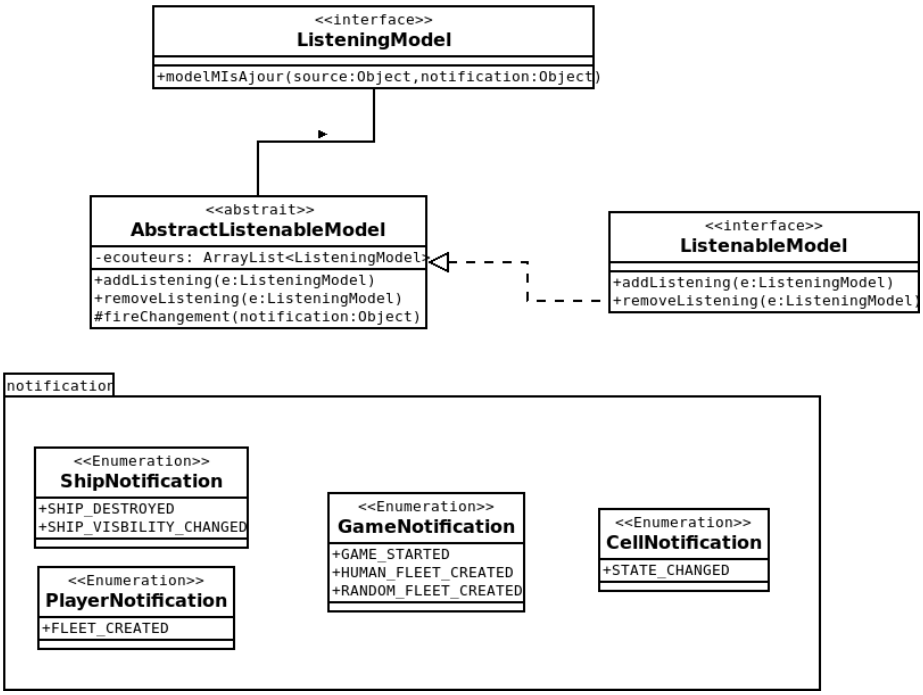

Figure 6: Package Util

4.2.2 Explication

Le package util de notre projet contient des classes qui permettent de mettre en œuvre le pattern MVC (**Modèle-Vue-Contrôleur**). La classe abstraite **AbstractListenableModel** sert de modèle à toutes les classes qui souhaitent être écoutées. Cette classe contient un attribut "écouteurs" de type ArrayList qui permet de stocker tous les objets qui écoutent les changements du modèle, ainsi que des méthodes **addListeningaddListening** et **removeListening** pour ajouter ou supprimer un écouteur. La méthode **fireChangement** permet de notifier tous les écouteurs en passant un objet de notification en paramètre.

L'interface **ListeningModel** est utilisée par les écouteurs pour être notifiés des changements de modèle. Cette interface contient une méthode **modelMIsAJour** qui est appelée lorsqu'un changement est détecté dans le modèle, et prend en paramètre un objet source et un objet de notification.

L'interface **ListenableModel** est utilisée par les modèles qui souhaitent être écoutés. Elle contient des méthodes **addListening** et **removeListening** pour ajouter ou supprimer un écouteur.

Le package notification contient **des classes d'énumération** qui sont utilisées pour notifier les changements de jeu. Par exemple, la classe **ShipNotification** contient les énumérations **SHIP_DESTROYED et SHIPVISIBILITY_CHANGED**, la classe **PlayerNotification** contient l'énumération **FLEET_CREATED** , la classe **GameNotification** contient les énumérations **GAME_STARTED, HUMAIN_FLEET_CREATED et RANDOM_FLEET_CREATED**, et la classe **CellNotification** contient l'énumération **STATE_CHANGED**.

En résumé, **le package util** de notre projet est utilisé pour mettre en place **le pattern MVC** en permettant la communication entre le modèle, les écouteurs et les vues. Les classes d'énumération **du package notification** sont utilisées pour notifier les changements de jeu aux écouteurs.

4.3 Package Views

4.3.1 Diagramme UML des classes

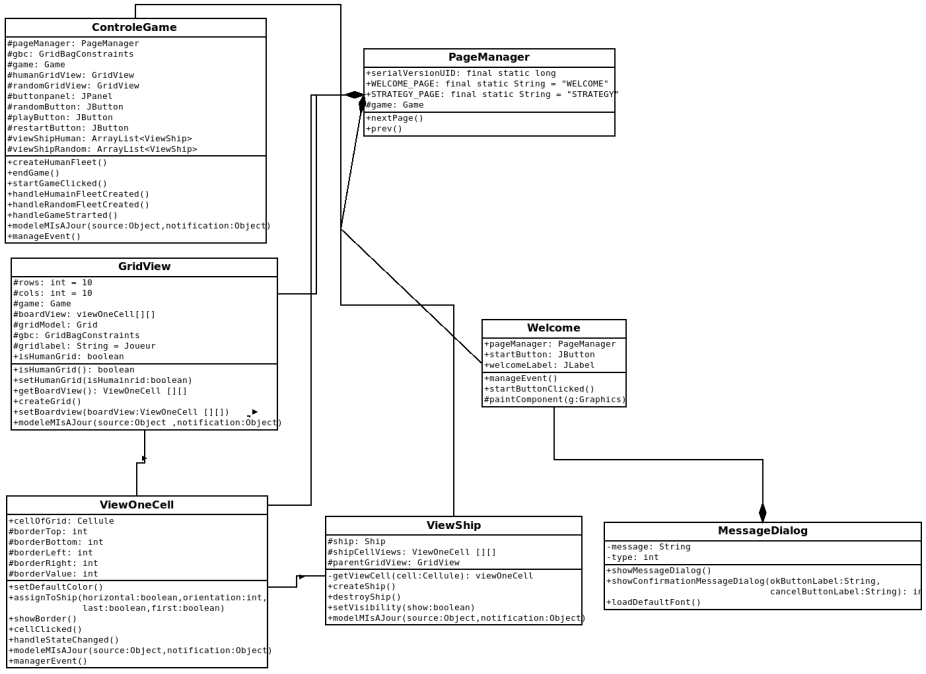

Figure 7: Package Views

4.3.2 Explication

Le package views contient les classes représentant l'interface graphique de l'application. Dans l'architecture MVC, il représente essentiellement la partie "Vue-Controller".

L'interface graphique est composée de deux pages, une page d'accueil et une page où se déroule le jeu proprement dit. Nous avons utilisé un gestionnaire de pages, qui permet de naviguer entre les pages du jeu grâce à **un CardLayout de Swing**. La page principale du jeu est composée des deux grilles des joueurs et des boutons de contrôles. Par ailleurs, l'une des difficultés que nous avons rencontrée consistait à bien disposer ces éléments sur l'interface. Nous avions commencé par utiliser **un GridLayout**, mais nous nous sommes rendu compte que nous étions limité. C'est ce qui nous a poussé à utiliser**un GridBagLayout** qui nous offrait plus de maniabilité grâce aux **GridBagConstraints** .

Les grilles sont représentées par une classe **GridView** qui étends **un JPanel** . Chaque instance de grille va être ensuite ajoutée à l'interface principale. Pour créer la grille, deux possibilités s'offraient à nous. La première consistait à déssiner des **Rectangle Swing** grâce à la méthode **drawRect** sur plusieurs positions da manière à former une grille. La deuxième était de disposer des vues de cellules, les une à côté des autres pour former une grille, toujours grâce au GridBagLayout. Chaque cellule est une insance de la classe **ViewOneCell** . La classe **ViewOneCell** est une carré déssiner dans **un JPanel** . Chacune de ses options avaient ses avantages. Au final nous avons choisi la deuxième option, car elle nous permettait entre autres de représenter les bateaux plus facilement sur la grille plus tard et de mieux gérer les propriétés des cellules.

Représenter graphiquement les bateaux sur la grille a été justement une des parties très intéressante de notre développement . Notre première idée était de créer d'autres cellules qu'on superposerait aux cellules de la grille pour chaque position de bateau. Mais en plus d'être compliqué à réaliser graphiquement, nous avions aussi entrevu à quel point il serait complexe et lourd algorithmiquement parlant de gérer les événements de tirs des joueurs avec cette méthode. Nous avons donc opté pour une approche plus simple: une classe **ViewShip** qui se contente de stocker dans un tableau les vues des cellules de chaque bateau et ensuite de donner une bordure à ces cellues pour qu'elles donnent l'aspect d'un bateau et le tour était joué. C'était non seulement beaucoup plus simple à implémenter, mais aussi beaucoup moins consommateur en mémoire puisque le tableau fait juste référence à des cellules déjà existantes au lieu d'en créer de nouvelles. Mais plus encore, ça nous permettrait de gérer les événements de tirs de manière vraiment optimale plus tard. Ainsi lorsqu'un joueur tir sur une cellule, on gère l'événement directement sur la cellule impactée et on peut appeler la fonction du modèle correspondante.

4.4 Package Config

4.4.1 Diagramme UML des classes

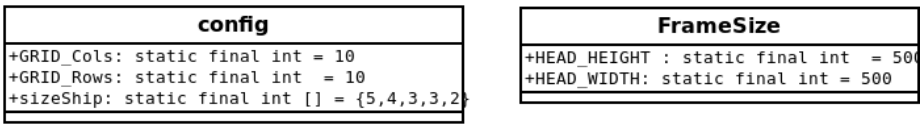

Figure 8: Package Config

4.4.2 Explication

Le package Config contient une classe **final Config** qui est chargée de stocker les configurations de base du jeu. La classe contient trois attributs statiques finaux : **GRID_Cols** qui représente le nombre de colonnes de la grille, **GRID_Rows** qui représente le nombre de lignes de la grille, et **sizeShip** qui est un tableau de tailles de navires possibles.

contient aussi la classe FrameSize qui a deux attributs statiques finaux qui representes la hauteur et la largeur de la fenetre.

La classe Config a pour but de centraliser les configurations du jeu et d'en faciliter l'accès à d'autres classes du projet. Par exemple, pour accéder à la taille des navires, il suffit d'appeler Config.sizeShip.

L'utilisation de constantes statiques finales permet également de rendre le code plus lisible et éviter les erreurs de saisie, car les valeurs ne peuvent pas être modifiées.

En somme, le package Config a pour objectif de faciliter la maintenance et la modification des configurations du jeu en centralisant toutes les constantes dans une seule classe. Cela permet également de garantir la cohérence des configurations tout au long du projet.

5 Contraint et Solution

En ce qui concerne les difficultés rencontrées, la principale a été liée à l'utilisation de Git. Au départ, nous n'avions pas une compréhension complète de son fonctionnement et nous avons continué à coder sans faire de commit. Malheureusement, cela a eu pour conséquence la perte de toutes nos données lorsque la machine sur laquelle nous travaillions a planté. Nous avons donc été contraints de recommencer le projet à zéro. Cependant, nous avons rapidement surmonté cette difficulté et sommes parvenus à mener à bien notre projet dans les délais impartis.

Une autre difficulté rencontrée a été de mettre à jour la vue, lorsque le modèle change d'état. Il faut d'abord noter que chaque entité (game, grid, ship, cell) est représentée par une vue. Chacune de ses vues écoute sur son modèle correspondant (**architecture mvc**). Par exemple, un bateau a deux types d'états : visible ou pas et détruit ou pas. Lorsque le modèle **Ship** change donc l'un de ses états et le notifie à son écouteur **ViewShip** , il fallait donc savoir lequel des deux types d'états avait changé, pour appliquer la mis à jour correspondante. Nous avons donc pensé à créer **des enums de notifications** pour chaque modèle. Dans chacun d'eux, nous avons listé les types de changements d'état qu'il peut y avoir dans le modèle correspondant. Ainsi, lorsqu'un modèle change d'état, en plus de le notifier à son écouteur (la vue), il doit aussi spécifier le type d'état qui a changé grâce à **un paramètre supplémentaire notification** , pour que la vue fasse la mis à jour adéquate.

6 Conclusion

En conclusion, ce projet de développement d'un jeu de bataille navale en Java a été une expérience enrichissante pour notre équipe. Nous avons eu l'opportunité d'appliquer nos connaissances en programmation orientée objet, en architecture MVC, en gestion de projet et en collaboration en équipe. Nous avons également pu mettre en pratique notre créativité et notre capacité à trouver des solutions efficaces à des problèmes complexes.

Le jeu que nous avons développé est fonctionnel, ludique et satisfait toutes les fonctionnalités requises. Nous sommes convaincus qu'il peut être amélioré et enrichi avec de nouvelles fonctionnalités et une interface utilisateur plus avancée.

En somme, nous sommes fiers du travail accompli et nous espérons que notre projet inspirera d'autres développeurs à explorer de nouvelles voies dans la création de jeux en Java.