RAPPORT DU PROJET "BATAILLE NAVALE"

Quentin Dumont

L2 INFORMATIQUE
TD2B

UNIVERSITÉ
CAEN
NORMANDIE

http://www.unicaen.fr

Sommaire

1 INTRODUCTION

Le projet final de l'unité d'enseignement "Interfaces Graphiques et Design Pattern" était un sujet imposé : nous devions développer une bataille navale jouable dans le terminal ainsi qu'en interface graphique. L'architecture de l'application devait respecter le Design Pattern MVC, dont nous allons décrire le fonctionnement par la suite. Dans la première semaine du projet, j'ai rapidement développé un modèle jouable en terminal, puis une vue et un contrôleur pour pouvoir y jouer via une interface graphique. Nous décrirons tout d'abord la partie algorithmique de l'application, c'est-à-dire la conception du modèle. Dans la continuité de cette explication, nous passerons en revue ce qui a été nécessaire au développement de l'interface graphique. Étant plutôt à l'aise sur la mise en place de cette architecture, j'ai décidé d'aller plus loin et de développer des fonctionnalités supplémentaires qui seront détaillées au cours des différentes parties de ce rapport. Je vous souhaite une bonne lecture.

2 CONCEPTION DU MODÈLE

Notre jeu de bataille navale doit reposer sur un modèle solide et correctement conçu. Ci-dessous, le diagramme UML de classes le présente dans sa manière globale. Nous allons expliquer comment fonctionne ce modèle en deux temps. Nous verrons d'abord le fonctionnement de notre grille et de ses bateaux, puis nous passerons à la mise en place des entités (ou joueurs) qui peuvent être soit des humains soit des ordinateurs. Il est important de noter que les diagrammes présentés dans ce rapport ne sont pas exhaustifs de tous les attributs et méthodes des différentes classes. Ne sont présents dans le diagramme que les attributs et méthodes publiques ou accessibles via des `getter`. Il y a des exceptions sur certaines classes, quand j'estime que cela mérite éclaircissement. N'hésitez pas à zoomer sur le diagramme si vous souhaitez regarder dans le détail.

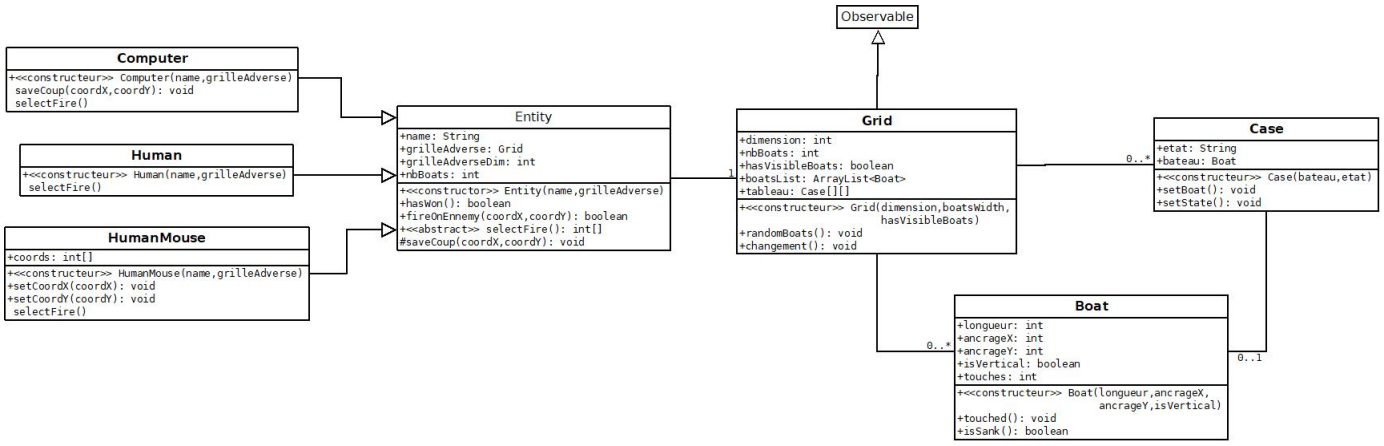

FIGURE 1 – Diagramme de classes du modèle de l'application

2.1 Grille et Bateaux

Dans mon jeu de bataille navale, je conçois les mers des joueurs comme des grilles composées de cases, qui peuvent contenir des bateaux. En considérant ce principe d'un point de vue objet, j'en arrive à la configuration suivante : la mer d'un joueur est un objet de type `Grid`, qui contient un tableau à deux dimensions d'objets de type `Case`. Enfin, ces objets de type `Case` peuvent contenir un bateau, qui est donc un objet de type `Boat`. Voilà pour la conception dans les grandes lignes. Détaillons maintenant ce que fait chaque classe.

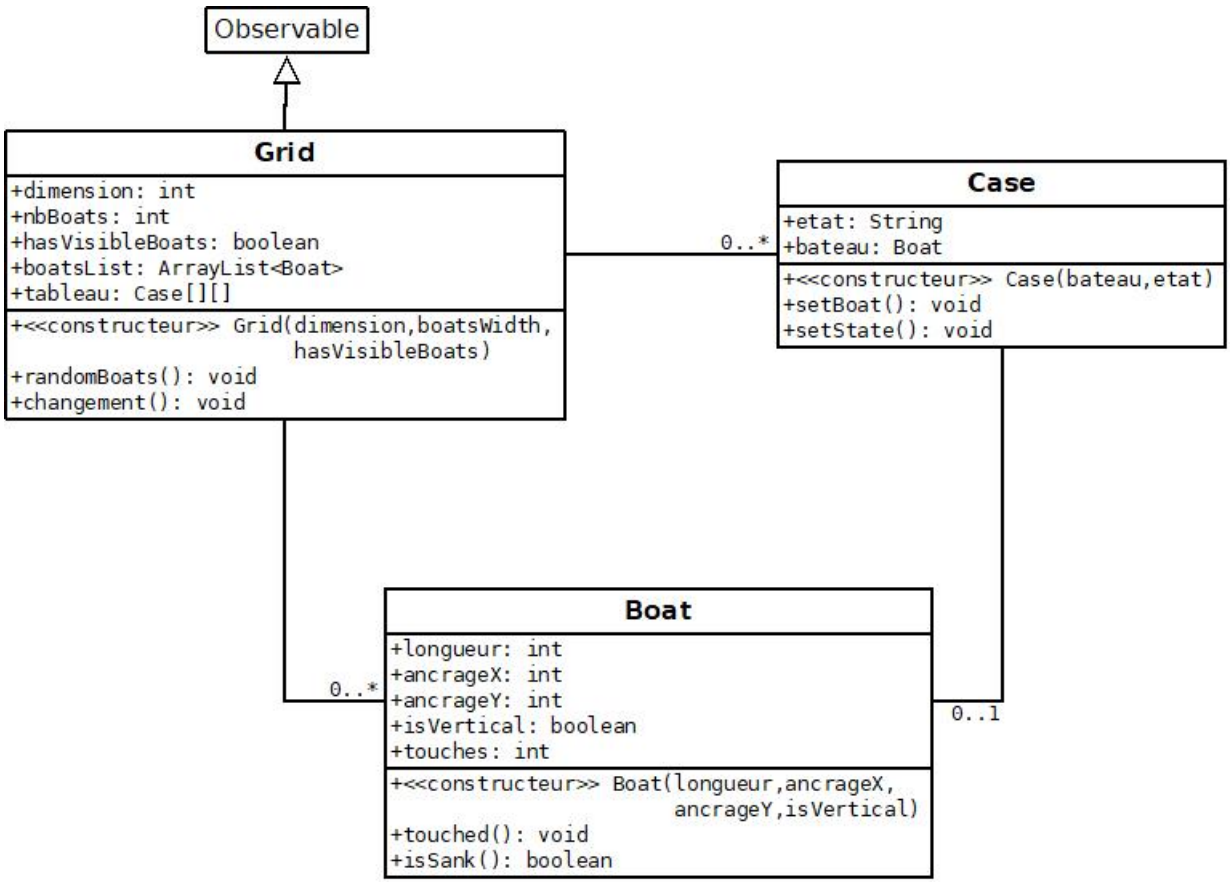

FIGURE 2 – Diagramme de la partie "Grille" du modèle

Commençons par le `Boat`. Il connaît sa longueur et son ancrage, qui est un couple de coordonnées à placer dans le tableau de `Case` de la `Grid`. Il sait si il est placé verticalement ou horizontalement, et connaît aussi le nombre de tirs qu'il a essuyé. L'ancrage est donc le point le plus haut du bateau s'il est placé verticalement, ou le point le plus à gauche s'il est placé horizontalement. Ce système nous permettra de placer le bateau dans la `Grid` et de l'identifier sans avoir à connaître toutes les coordonnées de son placement.

Poursuivons avec les `Case`. Une `Case` contient deux choses. Tout d'abord un état, pour savoir si la case est encore neutre ou a déjà été touchée. Dans ce dernier cas : soit l'obus est dans l'eau, soit il a touché un bateau. On a donc trois états distincts que j'ai définis en constantes de ma classe `Grid` : `NEUTRAL`, `MISSED`, et `HIT`. Ces `Case` contiennent aussi un `Boat`, mis à `null` par défaut.

Maintenant que nous possédons les objets que la `Grid` utilise, nous pouvons décrire le fonctionnement de la `Grid`. La `Grid` connaît les dimensions de son tableau de `Case`. Elle connaît également tous les bateaux qu'elle accueille, et sait si elle doit les afficher ou non, grâce à un booléen donné en paramètre de son constructeur. A la construction de la `Grid`, on initialise toutes les `Case` du tableau, et on ordonne la liste des longueurs de bateaux donnés en paramètre. La méthode principale de `Grid` est la méthode `randomBoats()` qui va placer aléatoirement des bateaux sur le tableau de `Case` en fonction de la liste des longueurs de bateaux passée en paramètre du constructeur. Elle les place dans l'ordre de la liste qu'on a triée, du plus grand au plus petit pour plus de performance algorithmique et pour éviter des bogues dûs à des impossibilités de placement des bateaux. `Grid` possède une autre méthode nommée `changement()` qui sera utile pour la partie vue de l'application. Elle hérite d'ailleurs de la classe `Observable`. Nous y reviendrons. Enfin, `Grid` redéfinit la méthode `toString()` pour pouvoir afficher la grille dans

le terminal. J'ai d'ailleurs choisi d'utiliser des caractères unicode pour plus de compréhension. On parcourt toutes les `Case` et on affiche leur état, et si le booléen `hasVisibleBoats` est à `true` et que la `Case` contient un `Boat` (différent de `null`), alors on affiche un caractère différent de l'état `NEUTRAL`.

Grâce à ces trois objets, nous sommes à même de construire une mer de la dimension que l'on souhaite, de la remplir aléatoirement de bateaux dont on donne juste la longueur, et de l'afficher de deux manières (bateaux visibles ou non).

2.2 Entités

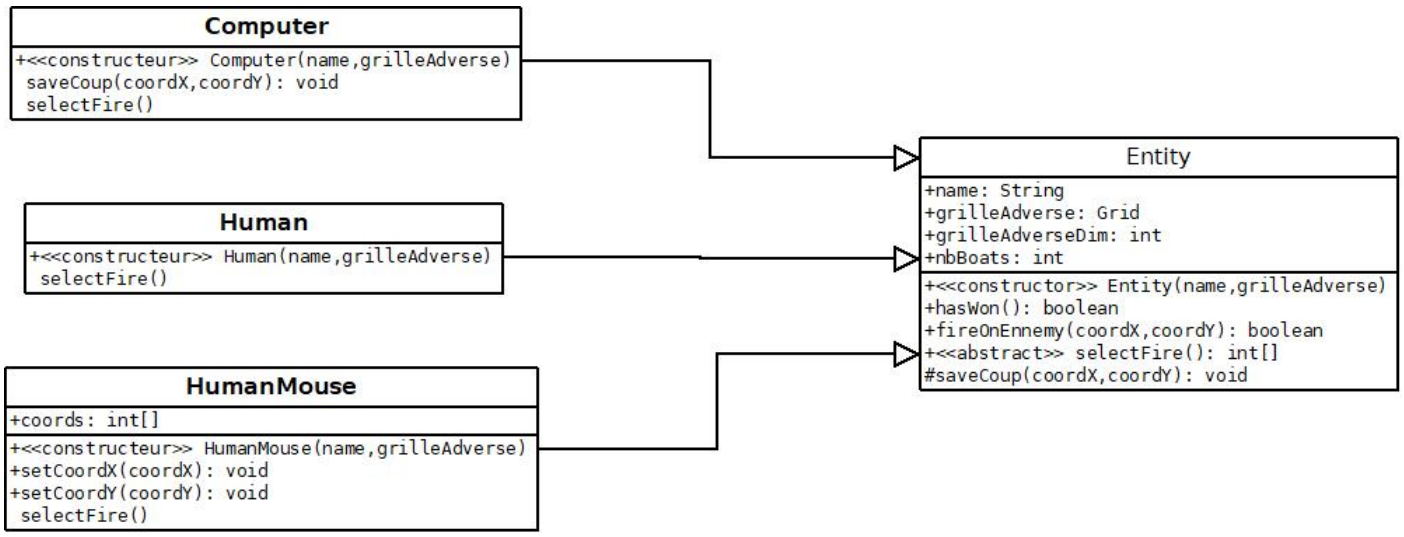

FIGURE 3 – Diagramme de la partie "Entités" du modèle

Si nous pouvons désormais construire une grille et la remplir de bateaux, il faut maintenant pouvoir tirer dessus. C'est là que les entités entrent en jeu. J'ai donc créé une classe abstraite `Entity` qui possède différentes méthodes, dont certaines sont redéfinies par les entités instanciables. Ces trois entités sont `Computer`, `Human` et `HumanMouse`, et héritent donc de la classe abstraite `Entity`. Nous parlerons de la dernière dans la partie vue.

A sa construction, une `Entity` prend en paramètre un nom et la grille sur laquelle elle tire (la grille adverse donc). `Entity` possède une méthode non abstraite `fireOnEnnemy`, qui prend en paramètre des coordonnées et applique un tir à ces coordonnées sur la grille adverse. Si le tir touche un bateau et que la case n'a pas déjà été tirée, le bateau concerné incrémente son compteur de touches, et s'il dit qu'il est coulé, alors il sera affiché sur la grille (s'il ne l'était pas avant) et le compteur de bateaux de l'entité est décrémenté.

La méthode `fireOnEnnemy()` prend des coordonnées en paramètres, et ces coordonnées sont choisies par la méthode `selectFire()`, qui est une méthode polymorphe puisque différente selon l'entité instanciée. Détaillons d'abord son fonctionnement sur un objet de type `Human`. `Human` est un joueur humain qui doit pouvoir renseigner un coup dans le terminal. `selectFire()` lui fait donc renseigner ses coups avec un objet de type `Scanner`. Cette méthode est beaucoup plus complexe sur un `Computer`. En effet, j'ai amélioré l'intelligence du "joueur ordinateur" pour qu'il s'acharne sur un bateau qu'il aurait trouvé par hasard en tirant aléatoirement, tant qu'il ne l'a pas coulé. Je laisse les commentaires présents dans la fonction détailler l'algorithme mis au point. Pour finir, une `Entity` doit savoir si elle a gagné, ce pourquoi elle est dotée d'une méthode `hasWon()` qui renvoie le booléen `nombreDeBateaux == 0`. En effet on rappelle que l'entité ne

connaît pas sa grille mais la grille sur laquelle elle tire, sans jamais accéder aux valeurs des cases pendant la phase de choix des coordonnées. Ce choix de conception peut paraître étrange mais s'est révélé plus efficace lors du développement de l'intelligence du "joueur ordinateur".

Finalement, nous avons réalisé un modèle qui ne manque plus que d'un contrôleur pour ordonner la partie et les tours des deux entités pour pouvoir jouer à cette bataille navale dans le terminal. La conception des entités laisse une certaine liberté de paramétrage, en permettant de faire s'affronter deux ordinateurs, deux humains ou bien un humain et un ordinateur.

3 CONCEPTION DE LA VUE ET DU CONTRÔLEUR

Une fois le modèle terminé, il est temps de développer ce qu'on appelle la vue dans le design pattern MVC Modèle-Vue-Contrôleur. Le fait d'avoir préparé le terrain en faisant hériter `Grid` de `Observable` va nous être fort utile.

3.1 Liaison avec le modèle

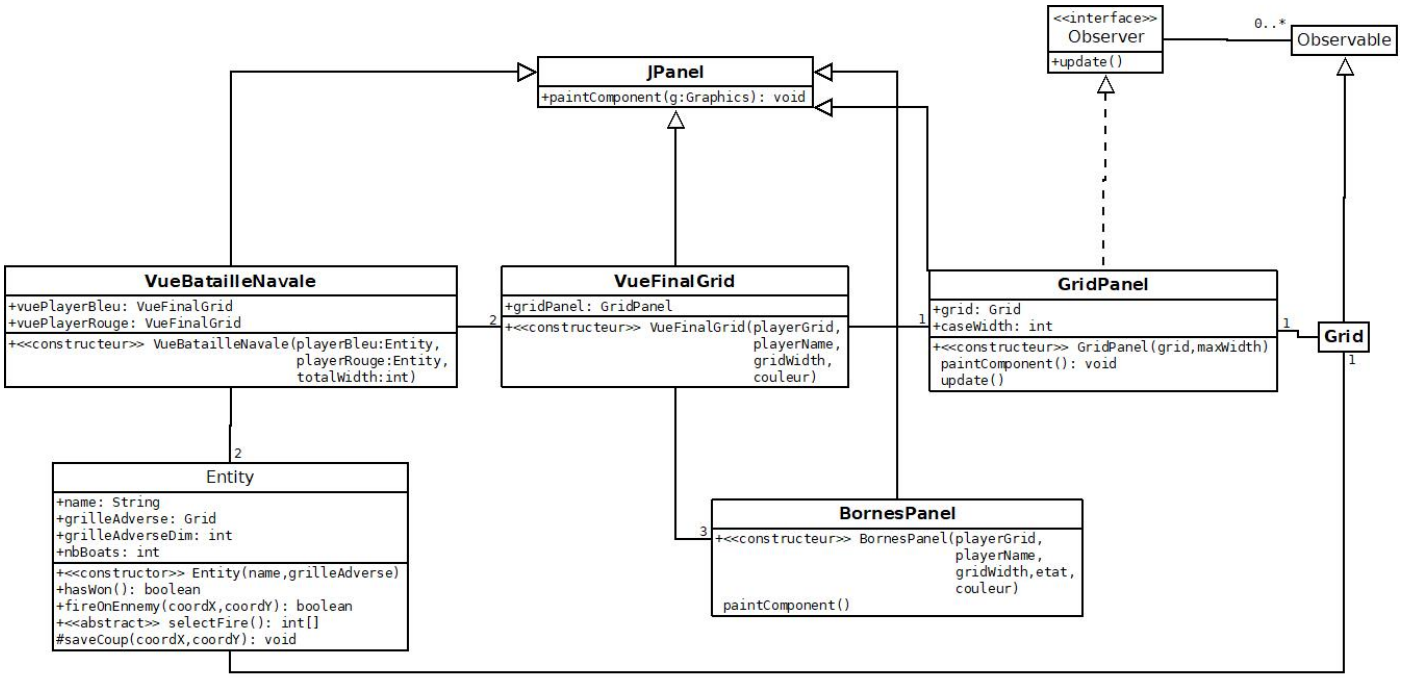

FIGURE 4 – Diagramme de la partie "Vue" de la vue/contrôleur

Le diagramme ci-dessus, quoi qu'un peu barbare, résume bien ce que je m'apprête à expliquer. Pour que notre interface graphique fonctionne, il faut qu'elle soit automatiquement mise à jour dès qu'un joueur tire sur une grille. Et pour respecter le pattern de conception, le modèle ne doit rien connaître de la vue. C'est là qu'`Observer` vient nous sauver la mise. La `Grid` est `Observable`. Ce qui veut dire qu'elle peut invoquer les méthodes suivantes : `setChanged()` et `notifyObservers()`. Les deux noms sont assez explicites. Ce sont ces deux méthodes qui sont utilisées dans la méthode `changement()` de la grille, dont nous avons parlé tout à l'heure. Créons maintenant la classe `GridPanel` qui implémente `Observer` et ajoute une `Grid` dans sa liste d'objets `Observable`. Grâce à `Observer`, `GridPanel` est en capacité de détecter tous les appels de la méthode `changement()` de `Grid`, et donc de se mettre à jour. A partir de là, c'est gagné. Nous n'avons plus qu'à redéfinir la méthode `update()` de `GridPanel` et lui dire d'invoquer la méthode `repaint()` pour qu'il se réaffiche et se mette à jour.

3.2 Disposition des composants

Les composants d'une interface graphique développée en Java doivent être découpés en de nombreuses classes si l'on veut garder de la maintenabilité. J'ai en effet réalisé une classe pour la vue de la grille uniquement nommée `GridPanel`, une autre pour ses bornes et sa description nommée `BornesPanel`. Ensuite j'ai créé une autre classe pour contenir la grille et ses bornes nommée `VueFinalGrid`, et enfin une dernière qui dispose deux `VueFinalGrid`, nommée `VueBatailleNavale`. Toutes ces classes héritent de la classe `JPanel`, et le découpage de ces panneaux m'a permis d'agencer tous ces composants très simplement, le plus souvent avec un `BorderLayout()`.

3.3 Multithreading

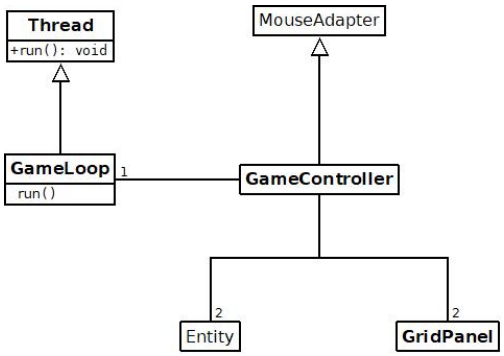

FIGURE 5 – Diagramme de la partie "Contrôleur" de la vue/contrôleur

Nous passons maintenant à la partie contrôleur de l'application. Je l'ai nommée "Multithreading" car j'ai été obligé de créer un Thread pour gérer la boucle de jeu en interface graphique. En effet il fallait réussir à mettre en pause la boucle pour qu'un joueur humain puisse cliquer sur la grille adverse et renseigner ainsi des coordonnées de tir à la méthode `fireOnEnnemy()`. Mais avant de mettre en place cette boucle de jeu, il faut que les grilles des deux joueurs captent le clic souris. Possible si l'on ajoute un `MouseListener` sur notre `GridPanel`. Les coordonnées de la grille en fonction de l'endroit du clic de la souris sont déduites avec une division des coordonnées du clic par rapport à la taille en pixels qu'on a donné au `GridPanel`. Une fois ce problème résolu, nous pouvons créer un contrôleur `GameController`. Il va fonctionner en parallèle de la boucle réelle de jeu `GameLoop`, en surveillant les clics de l'utilisateur. Le Thread `GameLoop` quant à lui va systématiquement attendre le clic de l'utilisateur si c'est au tour d'une entité de type `HumanMouse`. Pour cela, on fait s'endormir `GameLoop`, et dès que `GameController` assigne des coordonnées de tir, on réveille `GameLoop` qui applique le tir.

J'ai bien aimé concevoir cette application et aller plus loin en développant des fonctionnalités plus complexes. J'ai même développé un petit menu d'options à l'ouverture de l'application, vous permettant de paramétrer la partie (plus d'informations dans le README du dossier dist). Merci pour le temps et les explications accordées en séances de TP !