Dossier : Bataille Navale

RAPPORT

REUT Corentin - EIDD1 - niveau 2

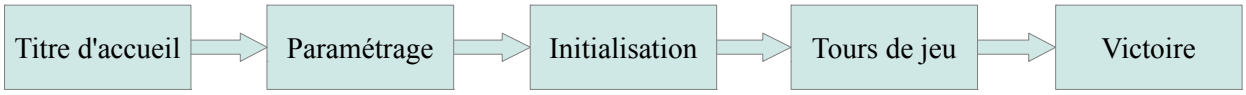
- Présentation générale :

Ce projet consistait à programmer un jeu de bataille navale en langage Java. J'ai utilisé pour ce faire Java JRE version 1.6 et Eclipse (Indigo x64). Le programme comporte 8 fichiers sources (*cf.* listing en annexe).
Voici les principales spécificités du programme communes à toutes les options (*cf.* paramétrage)

* L'affichage est en version console, défilant, monochrome. Les entrées se font au clavier.
* Le plateau est carré, avec ses lignes x lettrées et ses colonnes y chiffrées, de taille paramétrable.
* ' . ' = eau (propre plateau) ou état inconnu (plateau adverse) ; ' x ' = case bateau non touché ;
  ' T ' = case bateau touchée ; ' C ' = case bateau coulée.
* Quand un bateau est coulé, toutes ses cases passent en C.
* L'IA joue « une case sur deux » aléatoirement, cherche la direction du bateau quand elle en touche un, puis suit cette direction pour couler le bateau au plus vite. Elle ne joue jamais sur une case à coté d'une case Coulée car aucun bateau ne peut y figurer. (*cf.* règles Basiques)

* Le programme est constitué comme ceci :
  - Classe `Jeu` : contient la méthode main et des méthodes statiques. Permet tout le déroulement du jeu jusqu'à la victoire
  - Classe `Bateau` : implémente les différents bateaux des deux joueurs avec forme, nom et point de vie. Les méthodes permettent de placer les bateaux et d'encaisser les tirs adverses.
  - Interface `Joueur` : implémente un joueur, Classe `JoueurHumain` ou `JoueurRobot`.
    Les méthodes permettent de tirer sur une case et de faire tous les tests nécessaires à la validité et au résultat de ce tir.
    Pour le `JoueurHumain` : c'est ici qu'est géré tout l'affichage des plateaux propre et adverse, ainsi que le résultat du tir adverse précédent
    Pour le `JoueurRobot` : c'est ici qu'est gérée l'IA
  - Classe `Coord` : objet encapsulant deux entiers x et y. Ceci permet uniquement de gérer les deux coordonnées dans un seul objet, notamment pour le `return` des méthodes.
  - Classe `Case` : implémente les cases des plateaux de jeu, qui encapsulent plusieurs données :
    le Bateau se trouvant sur la case (`null` si rien) et son identifiant (0 si rien)
    un `boolean` qui renseigne si la case a déjà été jouée ou non
    l'Etat de la case
  - Enum `Etat` : type énuméré correspondant aux différents états possibles d'une case :
    `{INVALIDE, INC, BATEAU, EAU, MANQUE, TOUCHE, COULE}`

* Le programme s'exécute comme suit :



Le titre d'accueil s'affiche, puis l'utilisateur est invité à sélectionner les paramètres de sa partie. Les joueurs et les plateaux sont alors initialisés. Puis s'ensuivent alors les tours de jeu dans une boucle `while`, jusqu'à ce que l'un des joueurs gagne. On sort alors de la boucle et on affiche la victoire.

Détaillons maintenant les étapes de paramétrage, d'initialisation et de tours de jeu :

- Paramétrage de la partie :

Cette bataille navale offre différentes options de partie, que le joueur est successivement invité à rentrer.

`CpuOrPlayer` : demande si la partie doit être « Humain contre Ordinateur » ou « Humain contre Humain » l'Ordinateur désigne une intelligence artificielle implémentée dans le programme.

`autoOrNot` : demande si le placement des bateaux des joueurs humains doit se faire manuellement ou automatiquement par génération de coordonnées aléatoires

`X` : demande la taille de plateau souhaitée, pouvant aller de 10 à 26 (limite de l'alphabet)

`basicOrAdvanced` : demande quelles règles de jeu sont à appliquées :
- Basiques : les bateaux sont des segments plus ou moins longs et ne peuvent se toucher latéralement. Cette règle est imposée en mode de jeu contre l'ordinateur, par soucis d'une IA cohérente et équlibrée
- Avancées : les bateaux peuvent avoir tout type de forme et peuvent se toucher de tous les côtés

`typeFlotte` : demande quelle flotte utiliser parmis les 6 proposées (3 seulement en règles basiques)

Ces paramètres seront l'objet de conditions qui implémenteront tel ou tel code dans les méthodes du programme.

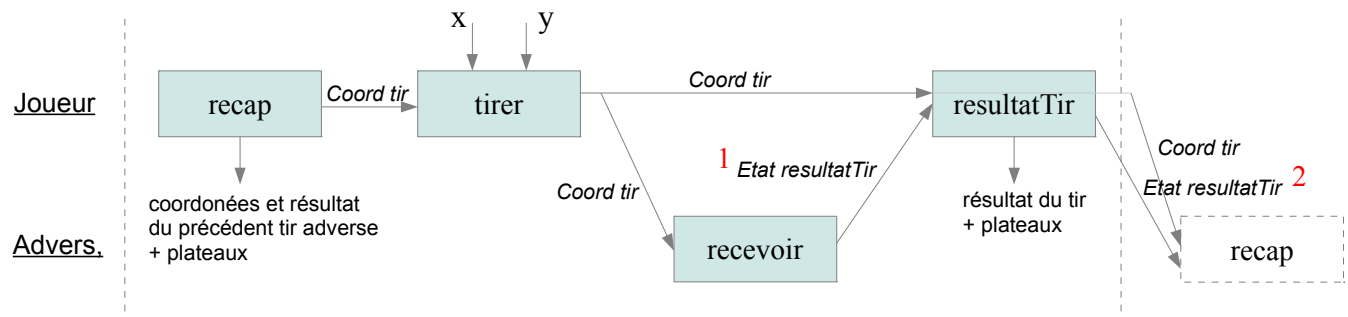
- Initialisation :

Ici sont initialisés les 2 joueurs ainsi que leurs plateaux.
Si `CpuOrPlayer=1`, on initialise `JoueurHumain j1` et `JoueurRobot j2`
Si `CpuOrPlayer=2`, on initialise `JoueurHumain j1` et `JoueurHumain j2`

Est alors appelée la méthode de placement des bateaux, avec la flotte choisie :
Flotte est un tableau de `Bateau` parcouru en entier, avec appel de `placerManuel` (utilise un `scanner`) ou `placerAuto` (utilise `Math.random()`) pour chaque. Pour le `JoueurRobot`, `placerAuto` est toujours appelé. Pour le `JoueurHumain`, cela dépend du paramètre `autoOrNot`.
Pour gérer la rotation des bateaux à 90°, 180° ou 270°, la méthode `pivot` est appelée 1, 2 ou 3 fois. Elle effectue une rotation de la forme du bateau `char[][] forme` d'un quart de tour vers la gauche (*cf.* TD).
Pour tester la validités de la position, la méthode `testCol` est appelée. `forme` ne contient que des `' '` (vide), `'X'` (morceau de bateau) et `'.'` (faux morceau de bateau, utilisé dans les règles basiques pour éviter la tangence).
La méthode parcourt alors `forme` et teste, quand un 'X' ou un `'.'` y figure, si la case du plateau qui devrait recevoir cet élément n'est pas déjà occupée. Les `'.'` sont donc considéré comme des éléments de bateaux par `testCol` et permettent d'avoir une seule méthode pour les deux règles basiques et avancées.

- Tours de jeu:

Les tours de jeu sont une succession de méthodes correspondant intuitivement à ce qui se passe lors d'une bataille navale sur papier. Voici le diagramme complet d'un tour :



On a ici les entrées (`scanner`) et sorties (`println`) ainsi que les arguments et `return` des méthodes.
Attardons-nous sur quelques points particuliers :

Ces méthodes sont celles apparaissant dans la méthode `Jeu.tour`, mais font en réalité appel à d'autres méthodes.

* La méthode `tirer` détecte qu'un tir a déjà été joué grâce au `boolean deja` de la `Case` visée.

* Suite au tir, c'est le joueur adverse qui informe du résultat du tir : *`MANQUE`* ou *`BATEAU`*, (1: méthode `recevoir`) puis le joueur jouant qui en déduit si cela correspond à un `Etat` *`EAU`*, *`TOUCHE`* ou *`COULE`* (grâce à `testCouler` qui accède au nombre de point de vie du bateau touché) (2: méthode `resultatTir`)

De plus, bien que le joueur ne connaisse pas réellement quel bateau il a touché, son gestionnaire de jeu le sait grâce à un identifiant (`int typeBat`) correspondant à un bateau précis. Ainsi, lorsqu'un tir a pour résultat *`TOUCHE`*, le joueur adverse informe le gestionnaire de quel `typeBat` a été touché et le `plateauAdv` du joueur jouant est mis à jour. Puis, lorsqu'un tir a pour résultat *`COULE`*, tout le `plateauAdv` est parcouru afin de voir quelles autres cases du bateau touché (`Case` avec un `typeBat` identique) doivent être passées en *`COULE`*.
J'ai recouru ici à un entier car les tests d'égalité sont bien plus simples qu'entre deux `Bateau`.

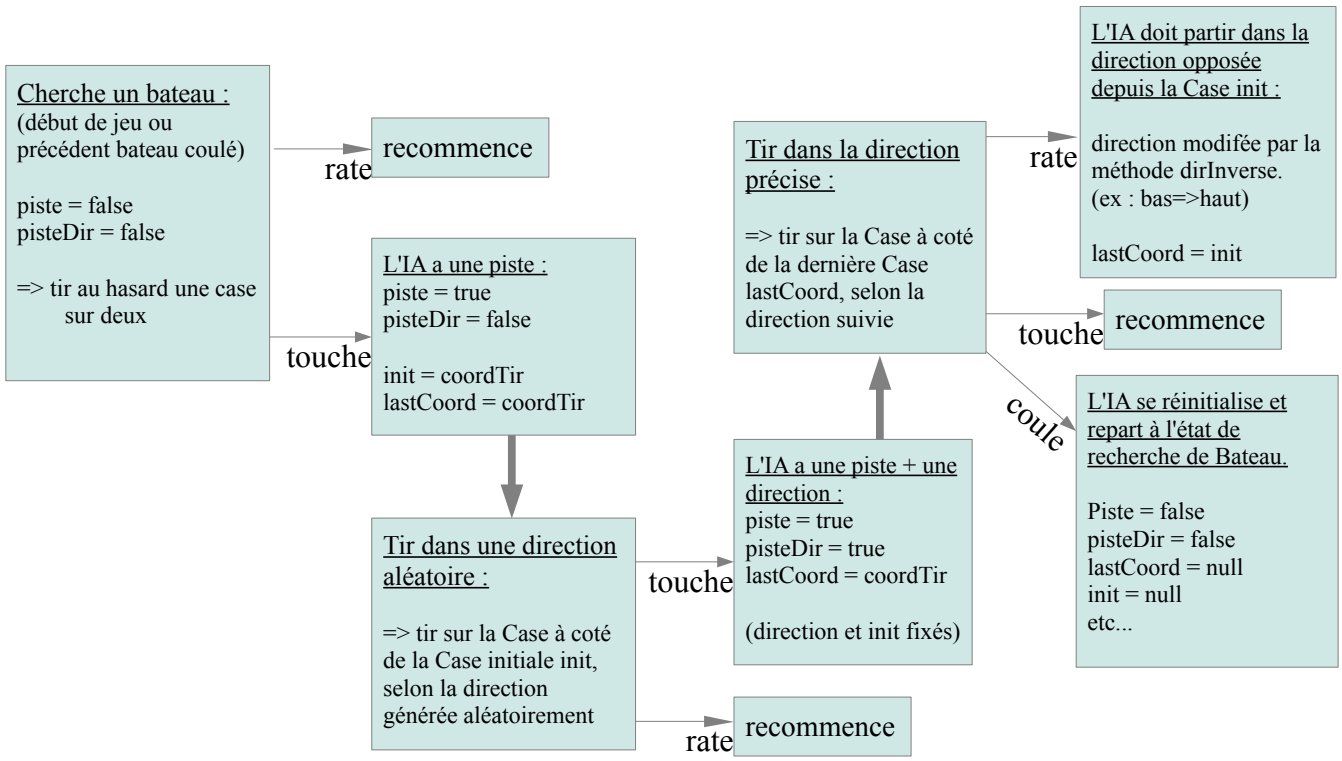
* La méthode `resultatTir` appelle aussi les méthodes `refresh` des deux joueurs pour mettre à jour leurs plateaux avec toutes les informations citées ci-dessus.

- JoueurRobot - intelligence artificelle :

Dans l'ensemble, le `JoueurRobot` reprend les même méthodes que le `JoueurHumain`, avec les entrées et sorties adaptées à une IA :
- l'affichage est inexistant car inutile.
- les entrées ne se font pas par `Scanner` mais par génération aléatoire (`Math.random()`) ou conditionnée par l'algorithme d'IA qui fonctionne comme suit :

Quand le `JoueurRobot` cherche un bateau, il tire au hasard, une case sur deux (exemple : toutes les cases noires d'un échiquier) pour minimiser les essais (car la taille minimale des bateaux est de 2 cases).

Quand il trouve un bateau, deux `boolean` interviennent : `piste` et `pisteDir`, correspondant respectivement à « l'IA a trouvé un bateau » et « l'IA sait en plus dans quelle direction sont les autres cases du Bateau ». Ainsi qu'un entier `int direction` correspondant à la direction à suivre pour le prochain tir, par rapport à la dernière case visée (avec la même correspondance que la méthode `Bateau.pivot`).
De plus, il enregistre les coordonnées de la première `Case` touchée de ce `Bateau` dans `Coord init`, et son dernier coup joué dans `Coord lastCoord`.



De plus, quand il tire, `JoueurRobot` ne fait pas que tester si le coup a déjà été joué. Il teste aussi si le coup a une probabilité non-nulle de toucher, car selon les règles Basiques les `Bateau` n'ont pas le droit de se toucher latéralement → méthode `tirInutile`.
Si le tir est inutile, il essaye un autre tir, ou change de direction (`dirInverse`) directement quand il piste déjà un `Bateau`.

- Autres fonctionnalités :

* La classe `Jeu` possède deux méthode `convertNL` et `convertLN` qui permettent respectivement de convertir un entier en sa lettre correspondante dans l'ordre alphanumérique, ou inversement. Ceci permet de se rapprocher de la bataille navale traditionnelle ou les coordonnées sont du type « A-6 » et non « 1-6 ».
Ces méthodes sont aussi bien utilisées en entrée (scanner) qu'en sortie (affichage).

* La taille des plateaux étant paramétrable, leur affichage doit s'adapter dynamiquement (légendes des lignes et colonnes, espaces suffisant, etc). Tout l'algorithme se situe dans la méthode `toString` de `JoueurHumain` : se reporter au code et aux commentaires pour l'explication.

* En mode Humain contre Humain, le masquage du tour précédent de l'adversaire se fait uniquement en sautant suffisamment de lignes pour obtenir un écran blanc (testé sur une résolution de 768 pixels de haut sur la console *Eclipse,*réglages par défaut) et on demande confirmation de fin de jeu et de début de jeu en entrant un caractère au hasard.

- Bugs connus :

* Lorsque le placement des bateaux est généré aléatoirement, il arrive que le programme bloque. J'en ai déduit qu'il n'arrivait pas à placer tous les bateaux. Cela arrive le plus souvent avec la flotte 1 (la plus grosse) et un plateau de10*10 (le plus petit possible).

* Si une lettre est entrée alors qu'on attendait un chiffre (rotation et position Y), le programme s'arrête et soulève une exception `InputMismatchException`. Je n'ai pas réussi à la traiter afin d'éviter un arrêt du programme par inadvertance.

- Améliorations possibles :

* Corriger les bugs ci dessus. (utilisation possible du Backtracking pour placer les bateaux sur le plateau à coup sûr + traitement de l'exception)

* Tout transposer à une interface graphique.
Les coordonnées saisies deviennent les cordonnées de la souris. Les caractères des plateaux correspondant à des états différents deviennent des .png transparents, etc …

* Lire une liste de bateaux à partir d'un fichier et crée une méthode qui affichent les différentes flottes proposées. Ceci permet à l'utilisateur d'éditer les flottes possibles et de créer ses propres flottes sans avoir à modifier le code source.

* Dans l'IA de JoueurRobot : prendre en compte la taille du plus grand bateau restant, pour l'éviter de tirer sur des cases où l'on devine qu'il n'y est pas. (exemple : o **...** o → un bateau de taille 4 ou plus ne peut pas se trouver ici).

* Implémenter des tirs spéciaux pour créer d'autres variantes aux règles (non fait par manque de temps, mais j'avais des idées de code) : ex: tir à tête chercheuse qui touche obligatoirement , et un tir espion qui agit comme un tir normal, mais qui dévoile en plus s'il y a des bateaux sur les cases entourant la case visée.

nb. **:** ce rapport est à lire avec le code source sous les yeux et vient compléter les commentaires et la Javadoc compris dans le code.
Il a été réalisé sous LibreOffice 3.4.4.

ANNEXES :

*** Fichiers sources :**

dir ../Projet/src/ :

- *Jeu.java*
- *Joueur.java*
- *JoueurHumain.java*
- *JoueurRobot.java*
- *Bateau.java*
- *Etat.java*
- *Coord.java*
- *Case.java*