TP 2 : Bataille navale

Présentation du sujet

L'objectif de ce TP est de réaliser un jeu de bataille navale en ligne de commande (CLI).

Le programme devra respecter les spécifications suivantes :

- Pouvoir dessiner une grille de jeu pour les navires du joueur ;
- Pouvoir dessiner une grille de taille identique pour les frappes sur la grille adverse ;
- Gérer la pose de navires sur la grille ;
- Gérer les frappes sur la grille ;
- La taille des grilles de jeu pourra être choisie au démarrage par l'utilisateur.

Le jeu se déroulera en 2 phases :

- La **phase de placement**, durant laquelle l'utilisateur doit entrer les positions de ses 5 navires.
- La **phase de jeu**, durant laquelle l'utilisateur entre les positions de ses frappes.

Nous utiliserons les quatre types de navires suivants :

- Destroyer (D), de taille 2 ;
- Submarine (S), de taille 3 ;
- BattleShip (B), de taille 4 ;
- Aircraft-Carrier (C), de taille 5.

Plusieurs fichiers vous sont fournis. Vous les rencontrerez au fur et à mesure du TP. Néanmoins, voici une liste de ces classes ainsi que leur utilité :

- `AIPlayer` correspond à l'entité d'un joueur associé à une Intelligence Artificielle. Nous en aurons besoin en fin de TP.
- `BattleShipsAI` correspond à la logique de l'intelligence artificielle utilisée dans la classe précédente.
- `ColorUtil` est une classe qui vous aidera à écrire en couleur dans la console.
- `Game` est le point d'entrée de notre jeu. C'est cette classe qui gère l'enchaînement des phases de jeu.
- `Hit` est une enum représentant le résultat d'une frappe. Nous en parlerons dans l'exercice 6.
- `IBoard` est une interface qui sera implémentée par l'interface `Board`. Nous l'utiliserons dans l'exercice 3.
- `InputHelper` est une classe qui vous aidera à récupérer des entrées utilisateur dans un format spécifique.

- `Player` correspond à l'entité associée à un joueur de notre jeu de bataille navale.

Gestion de version

Afin de conserver une trace de votre avancement à chaque exercice, vous utiliserez le gestionnaire de versions **Git**. N'ayez crainte, si vous n'êtes pas familier de logiciel et de son fonctionnement, les commandes à exécuter vous seront fournies systématiquement.

Pour commencer, créez un répertoire dans lequel vous placerez vos fichiers de code. Placez-vous ensuite dans ce dossier, et exécutez la commande suivante : `git init`.

Protocole de rendu

Vous rendrez votre TP par mail, sous forme d'une archive au format zip ou tgz contenant votre dossier de travail (c'est-à-dire le dossier contenant vos fichiers source **et** le dossier `.git` contenant l'historique de vos exercices). Cette archive devra être nommée avec votre nom et votre prénom : `tp2_nom_prenom.zip` (ou `.tgz`, bien sûr). Sous Linux, vous pourrez utiliser la commande suivante :
`tar cvfz tp2_nom_prenom.tgz dossierDeTravail`

Vous enverrez votre mail **avant le 24/02/2019 à 20h** aux adresses suivantes :
vprotois@excilys.com ou mgirbal@excilys.com en fonction de vos intervenants.

Informations utiles

Avant de vous plonger dans le TP, voici quelques informations utiles :

- Pensez à **documenter votre code** :
  - Faites en sorte d'écrire un code lisible par lui-même (des noms d'attributs et de méthodes clairs, cohérents, qui ont un sens).
  - Pour le code complexe, documentez son fonctionnement.
  - Documentez vos choix d'architecture (pourquoi laisser un constructeur vide, pourquoi mettre un attribut en static, etc.).
- À la fin des exercices, vous trouverez peut-être une liste de points à prendre en compte dans votre réflexion et votre code. Ces points recevront une attention particulière lors de la correction, ne les négligez donc pas. :-)

Exercice 1 : Affichage du « board »

Notions abordées : Classe, constructeurs valués, tableaux, conditions, boucles, écriture sur la console...

Dans cet exercice, nous allons créer une classe `Board`, qui représente le plateau de jeu d'un joueur. Ce plateau est composé de deux grilles de même taille destinées à recevoir respectivement les bateaux placés par le joueur et les frappes réalisées par le joueur sur la grille adverse.

Vous devrez réaliser les tâches suivantes :

- Créer la classe `Board`, composée d'un nom, d'un tableau 2D de `Character` pour les navires et d'un tableau 2D de `boolean` pour les frappes.
- Créer un premier constructeur valué, prenant en arguments le nom et la taille de la grille.
- Créer un second constructeur valué, prenant uniquement en argument un nom, et qui met par défaut la taille de la grille à la valeur 10.
- Créer une méthode `print()` qui dessine les deux grilles de jeu dans leur état respectif.
- Créer une classe `TestBoard` et y créer une méthode `main()` afin de tester le fonctionnement de l'affichage des grilles de jeu de la classe Board.

La grille des navires devra afficher le label du navire sur les positions où ce navire se trouve, et « . » sinon. Cependant, dans un premier temps, nous ne nous occupons pas des navires. La grille des navires sera donc entièrement peuplée de points « . » lorsque vous testerez votre fonction d'affichage.
La grille des frappes devra afficher « x » pour une position où une frappe a touché un bateau, et « . » sinon. Comme nous ne prenons pas encore en compte les bateaux, nous ne pouvons pas prendre en compte les frappes non plus. La grille des frappes sera donc également remplie de points lorsque vous testerez l'affichage.

Visuellement, le board affiché pourra ressembler à quelque chose de ce style :

```
Navires :                  Frappes :
   A B C D E F G H I J        A B C D E F G H I J
1  . . . . . . . . . .     1  . . . . . . . . . .
2  . . . . . . . . . .     2  . . . . . . . . . .
3  . . . . . . . . . .     3  . . . . . . . . . .
4  . . . . . . . . . .     4  . . . . . . . . . .
5  . . . . . . . . . .     5  . . . . . . . . . .
6  . . . . . . . . . .     6  . . . . . . . . . .
7  . . . . . . . . . .     7  . . . . . . . . . .
8  . . . . . . . . . .     8  . . . . . . . . . .
9  . . . . . . . . . .     9  . . . . . . . . . .
10 . . . . . . . . . .     10 . . . . . . . . . .
```

Pistes de réflexion :

- Avez-vous pensé à l'encapsulation dans votre classe `Board` ? :-)

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 1"
```

Exercice 2 : Création des classes de navires

Notions abordées : Classe, héritage, constructeur par défaut, constructeurs valués, enum...

Nous allons maintenant créer les différents types de navires qui pourront être utilisés dans notre jeu de bataille navale. Nous vous proposons ici de procéder par héritage. Nous aurons donc une classe abstraite `AbstractShip`, et quatre classes `Destroyer`, `Submarine`, `Battleship` et `Carrier` héritant de la classe `AbstractShip`.

La classe mère `AbstractShip` possèdera les éléments suivants :

- Un label qui permettra de représenter le type de navire sur la grille.
- Un nom qui correspond au nom entier du type navire.
- Une taille.
- Une orientation.
- Les accesseurs nécessaires pour faire de l'encapsulation.
- Un mutateur pour l'orientation.
- Un constructeur valué, prenant en argument un nom, un label, une taille et une orientation.

Les classes filles seront dotées d'un constructeur valué prenant en argument une orientation, ainsi que d'un constructeur par défaut qui définit l'orientation à l'est.

Vous devrez réaliser les tâches suivantes :

- Créer la classe abstraite `AbstractShip` avec ses constructeurs, attributs et méthodes.
- Créer les classes filles `Destroyer`, `Submarine`, `Battleship` et `Carrier` avec leurs constructeurs.

Pistes de réflexion :

- L'orientation des navires est un ensemble fini des quatre valeurs NORTH, SOUTH, EAST et WEST. Comment faire pour représenter au mieux cette information ?
- Nous commençons à avoir plusieurs fichiers source stockés pêle-mêle à la racine de notre répertoire de travail. Et ce nombre de fichier va continuer à croître au fil des exercices. Comment remédier à ce problème ?

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 2"
```

Exercice 3 : Placement des navires

Notions abordées : Interfaces, exceptions...

Il est maintenant temps de placer les navires sur la grille de l'exercice 1 ! Il semble nécessaire de doter notre classe `Board` de nouvelles méthodes afin de placer les navires et les frappes. Nous allons modifier la classe `Board` et faire en sorte qu'elle implémente l'interface `IBoard` qui vous est fournie. Cela sera utile plus tard dans le TP.

Vous devrez réaliser les tâches suivantes :

- Modifier la classe `Board` afin qu'elle implémente l'interface `IBoard`.
- Modifier la méthode `print()` de `Board` (si nécessaire) afin d'afficher les navires sur la grille correspondante.
- Modifier la méthode `main()` de la classe `TestBoard` pour placer quelques navires sur la grille de jeu et vérifier le fonctionnement de vos nouvelles méthodes.

Pistes de réflexion :

- Les indices de position des navires commencent-ils à 1 ou à 0 ?
- Que se passe-t-il si la valeur (position + longueur) d'un navire mène en dehors de la grille de jeu ? Que faire dans ce cas ?
- Que se passe-t-il si deux navires se chevauchent ? Que faire dans ce cas ?

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 3"
```

Exercice 4 : Entrées utilisateur

Notions abordées : Scanners, gestion des exceptions...

Nous souhaitons maintenant que notre application puisse interagir avec l'utilisateur. L'application devra proposer au joueur de placer 5 navires sur sa grille, par ordre de taille croissante, selon la répartition suivante : 1 Destroyer, 2 Submarine,

1 BattleShip, 1 Aircraft-Carrier.
L'utilisateur entrera les positions des navires au format « A1 n », « B4 w », « D3 s » …

Remarques :
À chaque fois que l'on récupère une entrée utilisateur, il est **très important** de vérifier la cohérence et l'exactitude de ces données. Il faudra donc utiliser un bloc `try … catch()` pour s'assurer que les valeurs entrées sont correctes.

Afin de gagner du temps, vous pouvez utiliser la classe `InputHelper` fournie, dont la méthode `readShipInput()` récupère les entrées utilisateur et les convertit en données exploitables.

Vous devrez réaliser les tâches suivantes :

- Modifier la classe `Player` qui vous est fournie en complétant la méthode `putShips()` :
  - Appeler la méthode `readShipInput()` tant que tous les navires ne sont pas correctement placés ;
  - Paramétrer le `Board` avec les valeurs retournées ;
  - Afficher l'état du `Board` entre chaque saisie.

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 4"
```

Exercice 5 : État des navires et des frappes

Notions abordées : Refactoring, exceptions...

Dans l'état actuel de notre programme, nous rencontrons deux problèmes principaux :

- Dans le `Board`, la grille des frappes contient des éléments de type `boolean`, ce qui implique que les frappes ne peuvent valoir que `true` ou `false`. On peut donc considérer que la valeur `true` correspondrait à une frappe réussie, et que `false` correspondrait à une frappe ratée. Mais dans ce cas, comment indiquer un emplacement où une frappe n'a pas encore eu lieu ? Une valeur de type `boolean` ne suffit pas !
- Dans le `Board`, les navires sont placés avec un tableau de `Character`. Comment peut-on savoir où le navire commence, où le navire se termine, et donc s'il est totalement détruit ou non ?

Nous allons donc faire ce qu'on appelle un « refactoring » du code.

Nous aurons besoin d'une classe `ShipState` intermédiaire entre le navire et la grille, capable de mémoriser l'état du navire en un point précis. La classe `ShipState` possède :

- un attribut de type `AbstractShip`, qui est une référence vers le navire concerné par cet état ;
- un attribut boolean `struck`, qui vaut "vrai" si le navire est touché en cet endroit ;
- une méthode `void addStrike()`, pour marquer le navire comme "touché" ;
- une méthode `boolean isStruck()`, qui retourne la valeur de l'attribut "struck" ;
- une méthode `String toString()`, qui retourne le label du navire associé (en rouge si le navire est touché en cet endroit) ;
- un méthode `boolean isSunk()`, qui retourne "vrai" si le navire est totalement détruit ;
- une méthode `AbstractShip getShip()` qui retourne le navire concerné par cet état.

Vous devrez réaliser les tâches suivantes :

- Dans `AbstractShip`, créer un attribut entier `strikeCount` ainsi qu'une méthode `addStrike()` permettant de manipuler le nombre de frappes que le navire a reçu au total. Créer la méthode `isSunk()` ;
- Créer la classe `ShipState` ;
- Dans `Board`, changer le tableau de `boolean hits` en un tableau de `Boolean` ;
- Dans `Board`, changer le tableau de `Character ships` en un tableau de `ShipState` ;
- Dans `Board`, changer la méthode `print()` pour afficher « . » si un Hit est null, « X » en blanc si un hit est faux, et « X » en rouge si un hit est vrai.

Remarques :
Vous pourrez avantageusement utiliser la classe `ColorUtil` qui vous est fournie.
Exemple d'utilisation :

```
System.out.print(ColorUtil.colorize("Hello World with COLOR!!!", ColorUtil.Color.RED));
```

Pour les utilisateurs d'**Eclipse** (il est donc fort probable que ce soit votre cas), vous pouvez télécharger le plugin « ANSI Escape in Console » pour que la console supporte l'affichage des couleurs. Le plugin est disponible à l'adresse suivante :
https://marketplace.eclipse.org/content/ansi-escape-console

Pistes de réflexion :

- Si on appelle `addStrike()` plus d'une fois par `ShipState`, le navire pourra donc être touché plus que le permet sa longueur.
  Comment gérer cet "état illégal" ?

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 5"
```

Exercice 6 : Envoyer des frappes

Notions abordées : Enums...

Maintenant que la phase de *refactoring* est terminée, nous allons nous attaquer à la gestion des frappes. Nous allons devoir ajouter une méthode à la classe `Board` afin de recevoir les frappes de l'adversaire et réciproquement.

Nous appellerons la nouvelle méthode `sendHit()` sur le `Board` de notre adversaire, et inversement.

```
/**
 * Sends a hit at the given position
 * @param x
 * @param y
 * @return status for the hit (eg : strike or miss)
 */
Hit sendHit(int x, int y);
```

L'enum `Hit` (qui vous est fourni) permet de renvoyer le statut d'une frappe. Les valeurs peuvent être `MISS` dans le cas d'une frappe ratée, `STRUCK` dans le cas d'une frappe réussie, ou bien le nom d'un des quatre navires si le navire vient d'être intégralement coulé.

Remarques :
L'enum `Hit` est particulier : Il possède un constructeur, ce qui nous permet de lui faire porter des valeur. Cela sera pratique lorsque nous voudrons créer l'enum directement à partir de la longueur du navire, grâce à la méthode `fromInt()`, ou lorsque nous voudrons avoir le nom ("label") du navire détruit.

Vous devrez réaliser les tâches suivantes :

- Copier le code ci-dessus correspondant à la déclaration de la méthode `sendHit()`, et le coller dans l'interface `IBoard` ;
- Modifier la classe `Board` pour qu'elle implémente cette méthode supplémentaire. Vous prendrez garde à retourner la bonne valeur dans le cas où un navire est détruit.
- Modifier la classe `TestBoard` pour lui faire utiliser la méthode `sendHit()`. Vérifiez que les navires et les frappes prennent la couleur attendue en cas de touche.
- Vérifier que si vous coulez un navire, `navire.isSunk()` renvoie bien « vrai » et que le dernier appel à `sendHit()` retourne la valeur `Hit.TYPE_DU_NAVIRE`. Lorsque cela se produit, afficher un message du type « LabelDuNavire coulé ».
- Compléter la méthode `sendHit()` de la classe `Player`.

Pistes de réflexion :

- Que se passe-t-il si on appelle `sendHit()` deux fois sur la même position d'un navire ?
- Que renvoie la méthode `hasShip(x,y)` lorsque le navire en (x,y) a été coulé ?

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 6"
```

Exercice 7 : Intelligence artificielle

Notions abordées : Random, listes, ...

En principe, nous disposons actuellement d'une application permettant de saisir les coordonnées de nos navires, de les placer correctement et sans erreur, d'envoyer des frappes sur les navires adverses, et détecter si un navire est touché ou coulé suite à une frappe. Nous avons donc toute la logique pour que notre jeu de bataille navale fonctionne. Cependant, il nous manque un élément essentiel pour pouvoir jouer : un adversaire !

Vous trouverez parmi les classes fournies la classe `BattleShipAI` qui propose une intelligence artificielle rudimentaire.

Remarques :
`BattleShipsAI` a besoin de deux objets `Board` (un par joueur) pour fonctionner. Votre `Board` implémente l'interface `IBoard`, ce qui permet à `BattleShipsAI` de savoir comment interagir avec votre `Board`, sans en connaître les détails d'implémentation. `IBoard` est en quelque sorte le "Manuel d'utilisation" d'un objet `Board`.

Vous devrez réaliser les tâches suivantes :

- Compléter la méthode `putShips()` de la classe `BattleShipAI`. Vous pourrez utiliser la classe java.util.Random. N'hésitez pas à vous référer à la documentation de cette classe pour savoir comment l'utiliser.
- Écrire une classe `TestGame` et sa fonction main() afin de tester le fonctionnement du jeu. Pour cela, nous ferons jouer l'IA contre elle-même. Votre test devra :
  - Initialiser un objet `board` de type `Board` et l'afficher à l'état initial ;
  - Initialiser une liste de navires ;
  - Initialiser un objet ai de type `BattleShipAI`, qui utilise le même board pour le joueur et son adversaire ;
  - Créer un compteur pour le nombre de navires détruits ;
  - Tant qu'il reste des navires non coulés sur la grille :
    - Appeler la méthode `ai.sendHit()` ;
    - Afficher les coordonnées du hit et son résultat (« touché », « manqué » ou « *TypeDuBateau* coulé »).
      La méthode `Hit.getLabel()` ou `Hit.toString()` vous sera utile.
    - Afficher le nouvel état du board.

Remarques :
Vous pouvez utiliser la méthode `sleep()` suivante pour temporiser la boucle de jeu :

```
private static void sleep(int ms) {
        try {
                Thread.sleep(ms);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}
```

Pistes de réflexion :
- Quel type de liste est le plus approprié ici ?

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 7"
```

Exercice 8 : Place au jeu !

Notions abordées : Scanners, héritage, ...

Notre jeu est quasiment terminé. Pour le moment, nous n'avons qu'une IA qui joue contre elle-même. Il est temps de lui rajouter un véritable adversaire !

Nous allons utiliser la classe `Player`. Comme vous aurez pu le deviner, cette classe représente un joueur. Elle possède entre autre deux Board (le sien et celui du joueur adverse), ainsi qu'une liste de navires. Ses méthodes `putShip()` et `sendHit()` doivent lire l'entrée clavier pour respectivement placer les navires sur sa grille et les frappes sur la grille ennemie. (cf. exercices 4 à 6)

La classe `AIPlayer` hérite de la classe `Player`. Elle redéfinit les méthodes `putShip()` et `sendHit()` pour utiliser son IA plutôt que les entrées clavier. (cf. exercice 7)

Vous devrez réaliser les tâches suivantes :

- Compléter les classes `Player` et `BattleShipAI` si cela n'a pas été entièrement fait lors des exercices précédents (vérifiez notamment que les méthodes `putShip()` et `sendHit()` ont été correctement implémentées) ;
- Compléter la classe `AIPlayer` ;
- Compléter la classe `Game` qui vous est fournie.
    - Compléter l'initialisation avec un joueur et une IA.
    - Modifier la boucle principale : tant qu'aucun joueur n'est hors jeu, …
        - Afficher le nom du joueur 1 ainsi que son Board ;
        - Saisir les coordonnées de la frappe ;
        - Ré-afficher le Board ;
        - Afficher les coordonnées du hit et son résultat ;
        - Recevoir la frappe de l'adversaire et réafficher le Board ;
        - Afficher les coordonnées du hit et son résultat.

Versionning : Lorsque vous aurez terminé cet exercice, avant de passer au suivant, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Exercice 8"
```

Bonus 1 : Multi-joueur

Modifier le code pour proposer un mode 2 joueurs.

Versionning : Lorsque vous aurez terminé ce bonus, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Bonus 1"
```

Bonus 2 : Sauvegarde d'une partie

L'idée est qu'à chaque tour de jeu, notre application écrive sur le disque dur un fichier qui mémorise l'état actuel du jeu. Au lancement du jeu, on vérifie la présence de ce fichier et on le charge le cas échéant.

Bien qu'elle puisse sembler complexe à première vue, cette fonctionnalité ne devrait pourtant pas vous demander trop de temps pour l'implémenter ! En effet, tout objet Java peut être sauvegardé dans un fichier, à condition qu'il soit **« Serializable »**. Un objet dit « serializable » est un objet qui implémente l'interface `Serializable`, et dont tous les attributs implémentent aussi cette interface.

Versionning : Lorsque vous aurez terminé ce bonus, placez-vous dans le répertoire de travail de votre TP et utilisez les commandes suivantes pour sauvegarder votre progression dans Git :

```
git add . -A
git commit -m "Bonus 2"
```