Université du Québec
École de technologie supérieure

Service des enseignements généraux

Auteur : Mathieu Nayrolles, Pierre Bélisle

Révision:

Pierre Bélisle (A2021)

Fred Simard (A2021)

INF111, Travail pratique #2

Programmation Orientée-objet

(équipe de 4, max)

1. Prérequis

Ce travail est prévu pour vous permettre d'utiliser les notions théoriques vues au cours et pratiquées en laboratoire. Nous présumons que les notions suivantes sont acquises :

- Syntaxe Java (non négligeable).
- Classe:
    - Constructeurs, accesseurs, mutateurs, this (actuel), equals, toString, clone.
- Collection de base de Java (utilisation):
    - Stack, LinkedList, Vector, …
- Tableau 1D et 2D, référence, objet, méthode vs sous-programme, attribut vs champ, constante et enregistrement vs classe.
- Programmation modulaire basée-objet (utilisant les classes et l'encapsulation).
- Gestion et levée d'exception.

2. Nécessaire

- S'assurer d'avoir les prérequis.
- Avoir le réflexe de résoudre un problème à la fois, en sous-programmes.
- Utiliser les bonnes pratiques d'écriture de code pour sauver du temps lors de la relecture.
- Bien comprendre les exigences et les respecter

3. Introduction à la bataille navale

La bataille navale, appelée aussi parfois touché-coulé mais plus communément Battleship, est un jeu de société dans lequel deux joueurs doivent placer des navires sur une grille tenue secrète et tenter de toucher les navires adverses. Le gagnant est celui qui parvient à torpiller complètement les navires de l'adversaire avant que tous les siens ne le soient.

Histoire

Le principe du jeu de bataille navale semble trouver son origine dans le jeu français « L'Attaque lors de la Première Guerre mondiale », on a aussi trouvé des liens de parenté avec le jeu de E. I. Horseman en 1890 (Baslinda) et on dit que des officiers russes y auraient joué antérieurement à la première guerre. La première version commerciale du jeu fut publiée en 1931 par la Starex Novelty Co. sous le nom de Salvo. Ce jeu est devenu populaire lors de son apparition en 1943 dans les publications américaines de divertissement de la Milton Bradley Company qui l'exploita sous la forme papier jusqu'en 1967, où elle sortit un jeu de plateau, puis en réalisa une version électronique en 1977.

3.1 Liste des navires

Chaque joueur possède les mêmes navires, dont le nombre et le type dépendent des règles du jeu choisies.

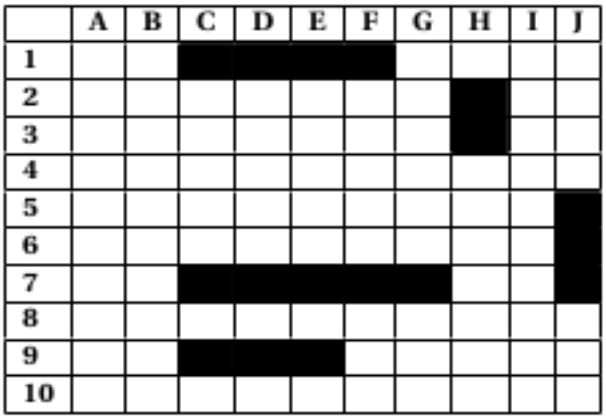
Une disposition peut ainsi comporter :

1. 1 porte-avions (5 cases)
2. 1 croiseur (4 cases)

3. 1 contre-torpilleurs (3 cases)
4. 1 sous-marin (3 cases)
5. 1 torpilleur (2 cases)

3.2 Grille de jeu

Qu'elle soit en version électronique ou non, la grille de jeu est habituellement toujours la même, numérotée de 1 à 10 horizontalement et de A à J verticalement.



Nous en ferons une variante puisque nos coordonnées en ligne et en colonne seront des entiers et nous n'imposerons pas cette limite de grille. Nous y reviendrons.

3.3 Règles

La bataille navale oppose deux joueurs qui s'affrontent. Chacun a une flotte composée de 5 bateaux, qui sont les suivants : 1 porte-avion (5 cases), 1 croiseur (4 cases), 1 contre torpilleur (3 cases), 1 sous-marin (3 case), 1 torpilleur (2 cases). Au début du jeu, chaque joueur place ses bateaux sur sa grille. Celle-ci est toujours numérotée de A à J verticalement et de 1 à 10 horizontalement. Un à un, les joueurs vont tirer sur une case de l'adversaire. Par exemple B.3 ou encore H.8. Le but est donc de couler les bateaux adverses. En général, les jeux de société prévoient des pions blancs pour les tirs dans l'eau (donc qui ne touchent aucun bateau adverse) et des pions rouges pour les toucher. Au fur et à mesure, il faut mettre les pions sur sa propre grille afin de se souvenir de nos tirs passés. Numériquement, nous remplaçons les informations que les pions fournissent par une collection de coordonnées pour les coups déjà joués. Nous y reviendrons.

4. Énoncé du travail

Nous nous intéressons à la stratégie qu'il faut employer pour jouer à ce jeu. Nous décidons d'écrire un programme qui nous aide à trouver et à tester des algorithmes différents. Nous allons simuler numériquement plusieurs stratégies de jeu.

Le travail est divisé en deux parties. Cet énoncé ne présente que la partie 1, la partie 2 sera fourni à mi-mandat.

4.1 Partie 1 (50%)

Pour débuter, nous allons concentrer nos efforts à conserver les données sur la flotte de navires et à les afficher. L'écran pourrait avoir l'air de ceci.

Ensuite, vous entamerez la partie 2 sur les stratégies de jeu (moins de code à écrire).

5. Conception imposée

Voici le détail des classes fournies et à écrire :

5.1 Classes fournies

5.1.1 Constantes

(À consulter) : La TAILLE de la grille y est définie.

5.1.2 GrilleGUI

Nous vous fournissons ce code qui servira seulement à montrer la flotte de navires et les options de menu. Cliquer sur une case ne génère aucune réaction. Vous pourrez changer la couleur d'une case c'est tout.

5.1.3 UtilitaireGrilleGui

Tout ce qui est nécessaire est fourni sauf une procédure écrite décrite plus loin.

5.1.4 Coord

Un enregistrement (champs publics) auquel nous avons ajoutez un constructeur, un comparateur (equals) et toString() (aide au débogage et pour éviter la répétition de code). Si vous en sentez le besoin, vous pouvez ajouter des méthodes mais celles mentionnées sont minimales et doivent être utilisées dans ce travail.

La description des différentes classes qui suit, est réalisée en approche ascendante mais vous pouvez développer en approche descendante (commencer par le main) le moment venu.

5.2 La classe Navire

Vous devez écrire cette classe au complet. Il y a plusieurs informations à conserver et plusieurs façons d'y parvenir.

5.2.1 Attributs

- Le nom du navire : « porteAvions », « sous-marin », …
- Le nombre de cases du navire (sa longueur)
- La couleur du navire (pour fin d'affichage)
- Une collection de votre choix entre Vector, LinkedList ou ArrayList de

java.util. Dans cette collection, vous gardez les coups qui ont touché le navire.

5.2.2 Le constructeur [1]

Vous devez écrire un constructeur qui lève des messages d'Exception avant d'initialiser les attributs. Vous recevez les valeurs suivantes : Le nom, la coordonnée de début, la coordonnée de fin et pour finir la couleur. Les coordonnées doivent être en ordre en ligne (de haut en bas) et en colonne (de gauche à droite). Elles doivent aussi être entre 0 et Constantes.TAILLE – 1 inclusivement.

```
 // Exemple de déclaration
public Navire(String nom, Coord debut, Coord fin, Color couleur)
```

Compter d'abord le nombre de lignes et de colonnes à partir de coordonnées reçues. On trouve ces nombres en faisant le calcul : fin – début + 1 (selon le bon champ évidemment).

Les messages d'exception possibles sont :

- "Coordonnées NORD_SUD invalide"

  Le nombre de lignes est plus grand que 1 et les colonnes sont différentes entre le début et la fin.

- "Coordonnées EST_OUEST invalide"

  Le nombre de colonnes est plus grand que 1 et les lignes sont différentes entre le début et la fin.

- "Ligne invalide"

  La ligne n'est pas dans l'intervalle de la grille ou la ligne de début est plus grande que la ligne de fin.

- "Colonne invalide"

  La colonne n'est pas dans l'intervalle de la grille ou la colonne de début est plus grande que la colonne de fin.

Il reste à initialiser les attributs s'il n'y a pas eu de message d'exception levé.

[1] Le constructeur par défaut n'est pas utile pour ce travail.

5.2.3 Les autres méthodes publics

- Une méthode boolean estCoule() qui retourne vrai si le bateau est coulé. Autrement dit, il a été touché autant de fois qu'il a de position entre début et fin, à des positions différentes (deux fois sur la même position ne compte pas).
- Une méthode boolean dejaRecuTir(Coord tir) qui retourne vrai si la coordonnée reçue à déjà touché au navire.
- Une méthode boolean tirAtouche (Coord tir) qui retourne vrai si la coordonnée reçue touche au navire actuel (this) et faux autrement (implique une boucle de recherche). Retient aussi la coordonnée si elle l'a touché. Voici l'algorithme en pseudocode.

```
Début
       Mettre un booléen à faux
       Si le navire n'est pas coulé
              Si le navire actuel n'a pas déjà reçu ce tir
                     Si le tir touche au navire actuel (méthode privée)
                            On retient la coordonnée dans la collection
                                   On met le booléen à vrai.
                     Fin
              Fin
       Fin
Fin
```

- Une méthode boolean chevauche(Navire navire) qui retourne vrai si une des positions du navire reçu touche à une des position du navire actuel (en ligne ou en colonne).

5.2.4 Une méthode privée

Une méthode utilitaire boolean positionTouche(Coord tir) qui retourne vrai si la coordonnée du tir est entre le début et la fin du navire (pas de boucle, juste la comparaison). Elle est utilisée pour savoir si le tirAtouche.

5.3 La classe Flotte

La flotte contient une collection de navires et contient les méthodes pour la gérer. On y retrouve les méthodes pour ajouter un navire, pour vérifier si un des navires a été touché et autres fonctionnalités dont voici les détails :

5.3.1 Le constructeur :

Le constructeur par défaut ne fait qu'instancier la collection de navires.

5.3.2 Les méthodes publiques :

- Une méthode **public boolean dejaRecuCoup(Coord tir)** retourne si le tir a touché un des navires.
- Une méthode **public boolean jeuTermine()** qui retourne si tous les navires de la flotte sont coulés.
- Une méthode **public Navire[] getTabNavires()** qui retourne un tableau statique contenant les navires (voir toArray()).
- Une méthode **public boolean leTirTouche(Coord tir)** qui retourne si le tir a touché un des navires de la flotte.

5.3.3 Les méthodes privées

- Une méthode **int ajouterNavire(Navire navire)** qui ajoute un navire seulement si les coordonnées du navire sont valides. Elles doivent être à l'intérieur de la grille et ne pas chevaucher un navire actuellement dans la flotte. La procédure retourne un des codes d'erreur suivant défini localement dans la classe Flotte : AUCUNE_ERREUR, NAVIRE_DEJA_SUR_PLACE et POSITION_INVALIDE. S'il n'y a aucune erreur, le bateau est ajouté.
- Une méthode **Navire obtenirNavireAleatoire(String nom, int longueur, Color couleur)** qui retourne un navire dont les coordonnées ont été générées aléatoirement, validées et ordonnées (haut en bas et de gauche à droite). Un bon truc est d'utiliser les messages d'exception que le constructeur de la classe Navire lève. Tant qu'il lance des messages, on régénère des coordonnées.

- Ici on ne s'occupe que des coordonnées valides à l'intérieur de la grille et ordonnées. Le chevauchement est géré par la procédure suivante.
- Il faut créer des navires de la longueur demandée (fin – début + 1 == longueur).
- Pour trouver une direction au hasard, un bon truc est de générer un nombre entre 1 et 4 (NORD ==1 , SUD == 2, ...)

```
Début
Tant que le début ou la fin est invalide
      Générer une coordonnée de début et une direction
      Calculer la fin dans le bon ordre
      Essayer de créer un navire
      S'il n'y a pas de message d'exception, c'est valide
Fin
```

- Une méthode void genererPosNavireAleaInsererDsGrille() qui ajoute un à un les navires dans la flotte. Il y a autant de boucles qu'il y a de navires. Une boucle se termine lorsque ajouterNavire retourne AUCUNE_ERREUR.

5.3.4 Une méthode public static

Une méthode **static obtenirFlotteAleatoire()** crée une flotte, génère la position des navires aléatoire (appel du SP précédent) et la retourne.

5.4 Le module UtilitaireGrilleGui

5.4.1 Une seule méthode public

- La méthode **static montrerFlotte(GrilleGui gui, Flotte flotte)** utilise le gui pour afficher tous les navires de la flotte de la bonne couleur.

6. Concernant la remise

Ce travail est à réaliser en équipe de quatre maximum. La progression du devoir sera évaluée à mi-mandat. Vous devrez déposer votre partie 1 le 19 octobre (gr.01) ou le 21 octobre (gr.02). Le programme remis sera évalué par rapport à la fonctionnalité et la qualité globale et cette évaluation comptera pour 50% de votre note.

La remise finale des 2 parties est le 2 novembre (gr.01) ou le 3 novembre (gr.02).

Aucun retard n'est accepté.

7. Barème de correction

Exécution et tests 40%

Doit respecter le comportement décrit dans l'énoncé.

Qualité de programmation 60%

Liste non-exhaustive des erreurs pénalisées:

1. Nom de fonctions et méthodes non représentatives.
2. Nom de variables et constantes non représentatives.
3. Aération et/ou indentation laissent à désirer
4. Découpage en sous programmes insuffisants.
5. Répétition inutile de code dû au manque de sous programmes.
6. Répétition inutile de code dû à l'incompréhension de l'utilité du paramétrage.
7. Constantes non définies.
8. Constantes non utilisées lorsque possible (même dans les commentaires).
9. Constantes en minuscules.
10. Commentaire d'en-tête de programme manquant (explication, auteurs et version).
11. Commentaires des constantes manquantes.
12. Commentaires des variables manquantes.
13. Commentaires de déclaration des sous-programmes manquants (API).
14. Commentaires non judicieux ou inutiles.

15. Commentaires manquants sur la stratégie employée dans chaque sous-programme dont l'algorithme n'est pas évident et nécessite réflexion (explique comment).
16. Commentaires mal disposés (ex : en bout de ligne).
17. Non utilisation d'un sous-programme lorsque c'est possible ou exigé.
18. Code inutile.
19. Affichage dans une fonction de calcul.
20. Qualité du français dans les commentaires.
21. Non-respect des droits d'auteurs (mettre toutes les références incluant les vôtres).
22. Non utilisation de boucle lorsque possible.
23. Autres (s'il y a une pratique non énumérée qui n'a pas de sens).