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 (partie 2)

Programmation Orientée-objet

(équipe de 4, max)

Deuxième partie (50%)

1. Prérequis

Dans la première partie du travail, vous deviez écrire les classes pour préparer la comparaison de stratégies de jeu de bataille navale. Nous présumons que les classes de la partie 1 fonctionnent correctement et que votre application montre la flotte du joueur qui est générée aléatoirement, sans chevauchement et à l'intérieur de la grille.

2. Stratégies et description générale du travail

Le reste de l'application sert à comparer diverses stratégies différentes pour la bataille navale. Nous allons commencer par comprendre la stratégie de quelqu'un qui joue pour la première fois et celle d'un débutant. Ensuite, d'autres stratégies vous sont proposées et vous pourrez les comparer entre elles via le nombre de tirs grâce au programme développé dans la partie 1.

À la fin de la partie 2, vous devez pouvoir choisir la stratégie de jeu en cliquant sur les boutons prévus à cet effet (que vous avez vu dans la partie 1) et voir l'ordinateur trouver les navires. Avec ce que nous vous fournissons, vous pouvez déjà voir les deux premières stratégies fonctionner après l'inclusion des modules et des classes fournies.

Au total, nous voulons implémenter cinq stratégies dont deux vous sont données et deux vous sont décrites. La dernière doit être de votre cru. Vous ne perdez pas de points si elle n'est pas réalisée mais vous obtenez 10 points si vous la réalisez et qu'elle fonctionne (bonus). Bien entendu, les deux précédentes doivent fonctionner d'abord.

3. Conception imposée

Il y a une classe par stratégie et dans chacune des classes, deux méthodes obligatoires :

Dans chacune des classes de stratégies, il y a au moins ces deux méthodes :

- La méthode **public Coord getTir()** génère une coordonnée et la retourne. La coordonnée est générée selon la stratégie implémentée.
- La méthode **public void aviserTir()** est appelée lorsque le tir généré par getTir a touché un navire de la flotte du joueur. L'ordi décide ce qu'il fait de cette information selon la stratégie implémentée.

3.1 StrategieOrdiPremiereFois (fourni)

Joue n'importe où au hasard et ne retient pas s'il a touché un navire ou non. Il peut tirer plusieurs fois au même endroit.

Dans getTir, il retourne un tir généré aléatoirement et ne fait rien dans la méthode aviserTouche().

3.2 StrategieOrdi Debutant (fourni)

Joue n'importe où au hasard dans les cases qui n'ont jamais reçu de tir. Il ne fait rien de spécial s'il touche un navire mais retient tous les coups joués pour ne pas les rejouer.

Dans getTir, un tir est généré aléatoirement et il est ajouté dans une collection (attribut locale privée) s'il n'en fait pas déjà partie et cela avant d'être retourné.

Rien n'est fait dans la méthode aviserTouche().

***Assurez-vous de bien comprendre le fonctionnement de ces deux classes, les modules utilitaires et le programme principal avant de poursuivre.

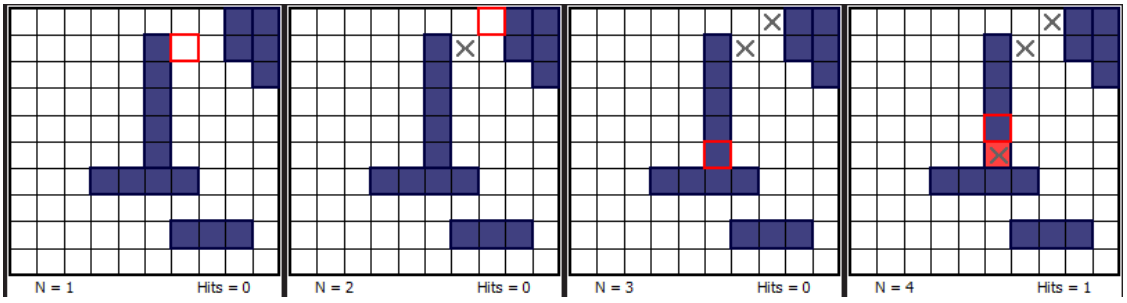
3.3 StrategieOrdiIntermediaire (à implémenter)

Une grosse différence ici est que dans la méthode aviserTouche(), on retient les coordonnées adjacentes et non en diagonal (NORD, SUD, EST et OUEST seulement) du dernier tir qui a été envoyé par getTir (celui qui vient de toucher). Il faut ajouter une collection en attribut pour cela en plus de celle qui retient les tirs effectués.

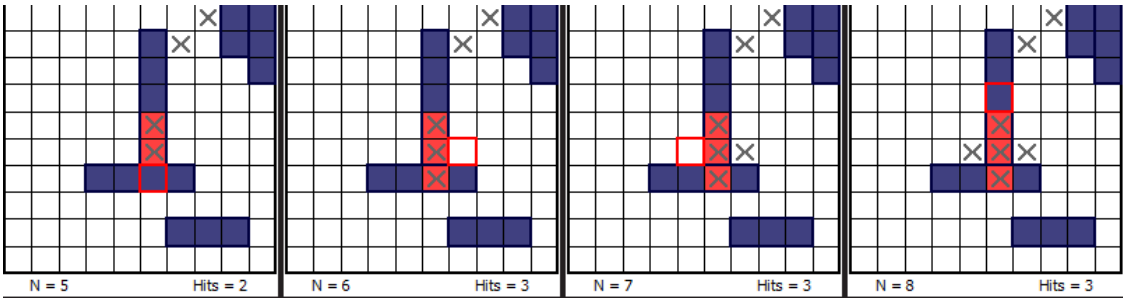
Donc, dans getTir(), tant que la collection des coordonnées adjacentes n'est pas vide et qu'on n'a pas obtenu un tir déjà joué, on obtient un autre tir de la collection. En d'autres mots, cette boucle se termine si elle vous a donné un tir qui n'est pas déjà joué ou qu'elle est vide.

En sortant de la boucle, s'il n'a pas trouvé un tir, le tir est généré au hasard dans des coups non déjà joués (comme dans la stratégie précédente).

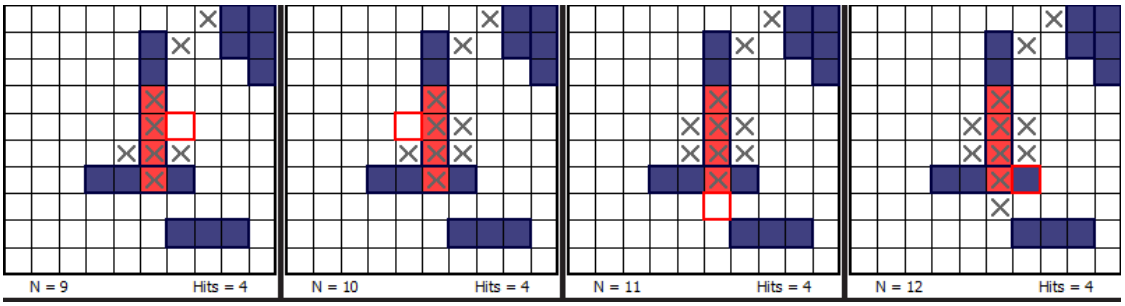
Les figures qui suivent illustrent une partie. Vous pouvez utiliser cette stratégie ou une autre. À vous d'utiliser une collection adaptée (Stack, LinkedList, Vector, ArrayList ?) selon votre stratégie.



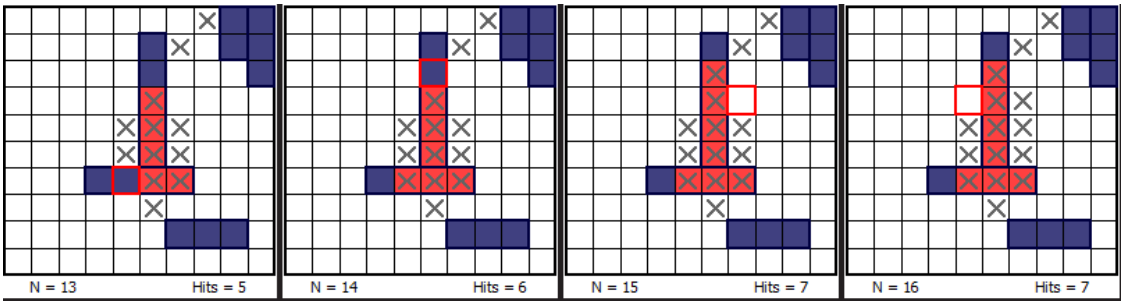
Initialement, la collection est vide alors le tir est généré aléatoirement. S'il touche un navire Les quatre cases adjacentes (NORD, SUD, EST, OUEST) sont ajoutées à la collection des cibles potentielles par la méthode aviserTouche(). Au tour 4 l'algorithme prend un tir non déjà joué, ici au nord, et touche à nouveau un navire. Les cases N, S, E et O de ce nouveau touché sont ajoutées à la collection.



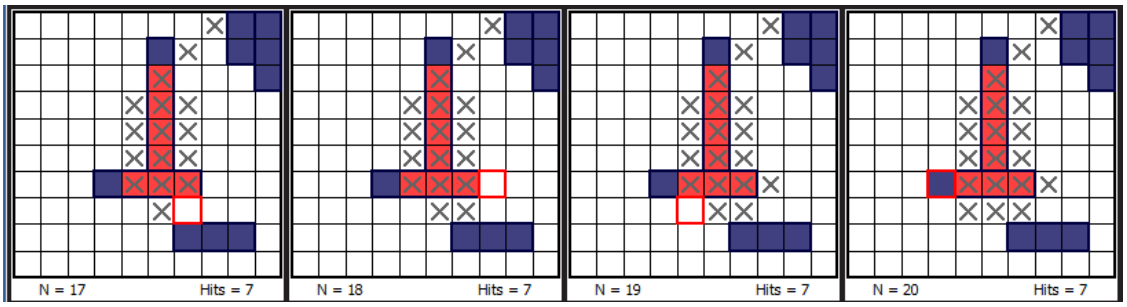
Au tour 5, l'algorithme touche encore, on ajoute donc les cases adjacentes. Les tours 6 et 7 ne touchent pas.



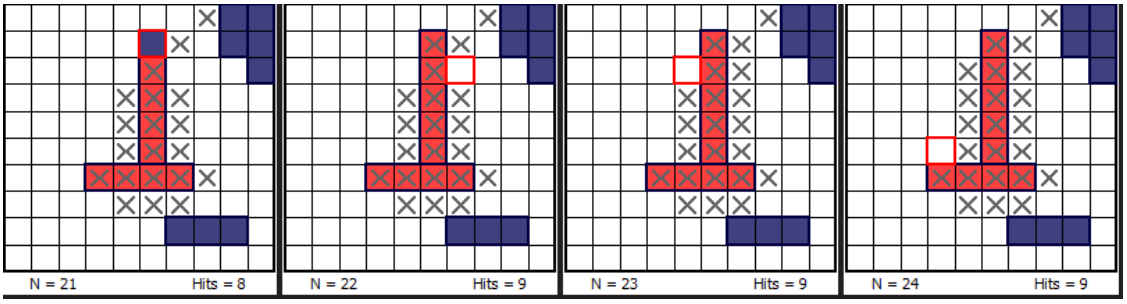
Les tours 9 et 10 montre comment l'algorithme teste et élimine les cases des deux côtés du navire. Au tour 12, l'algorithme découvre un nouveau navire.



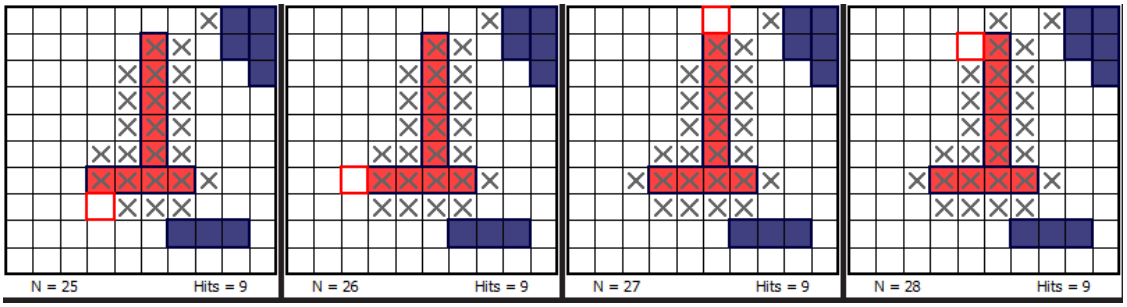
L'exploration de cet amas de navires continue.



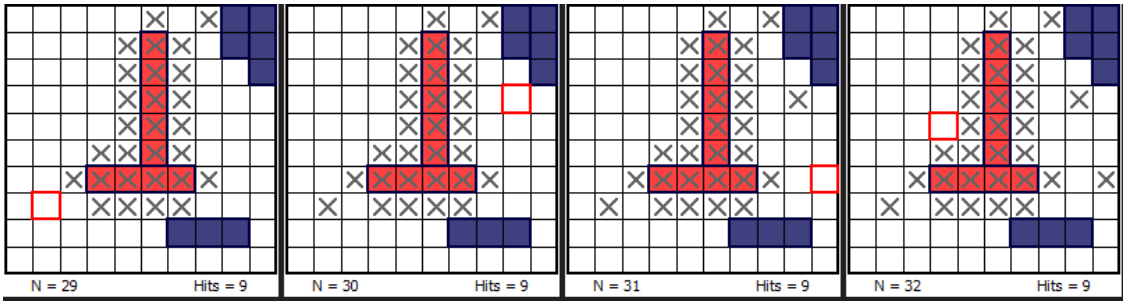
Pour être certain, nous devons tirer sur toutes les cases adjacentes à une case qui a donné un touché.



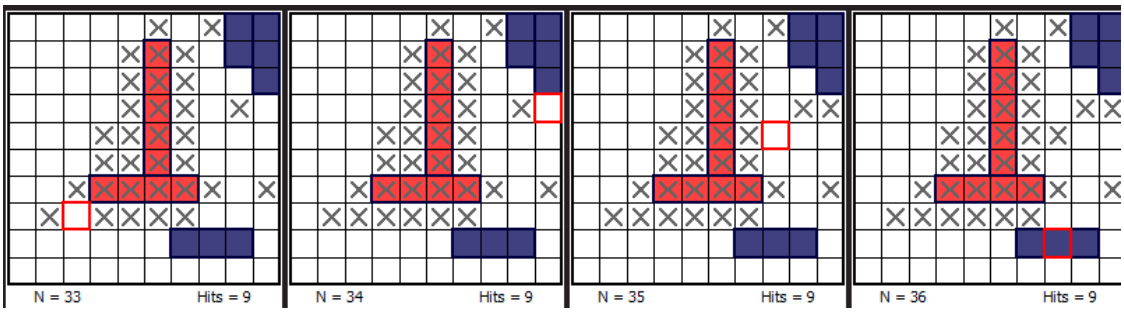
Au tour 21, nous avons un nouveau touché.



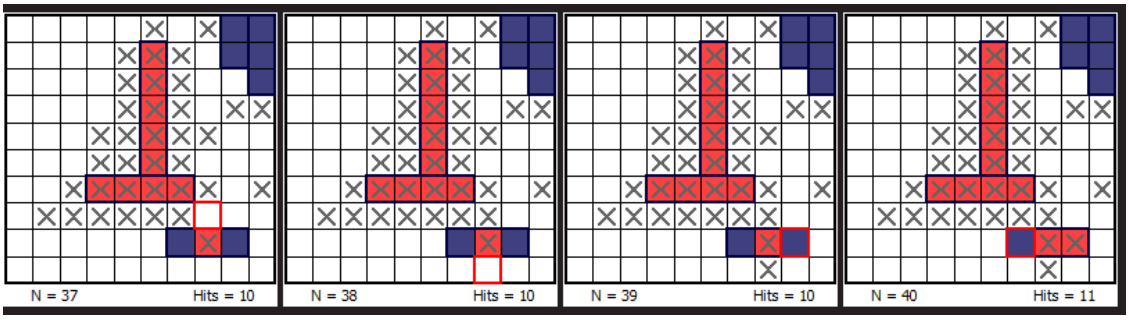
Au tour 28, la dernière cible potentielle a été retirée de la collection, elle devient vide et le prochain tir sera généré aléatoirement.



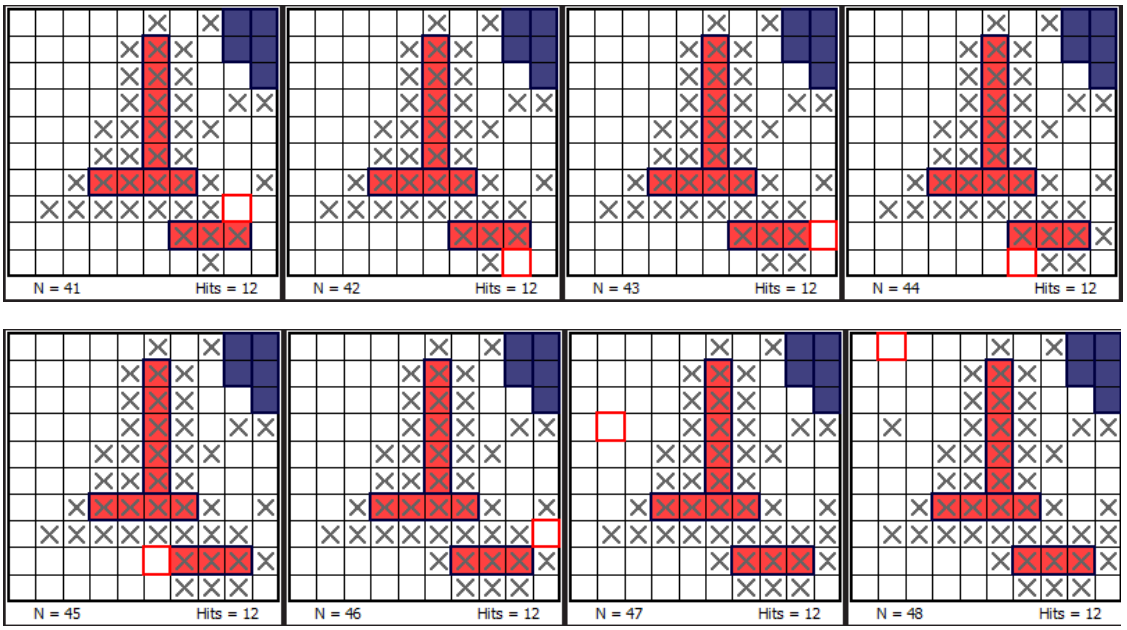
Aucun touché ici.


N = 33
Hits = 9
N = 34
Hits = 9
N = 35
Hits = 9
N = 36
Hits = 9

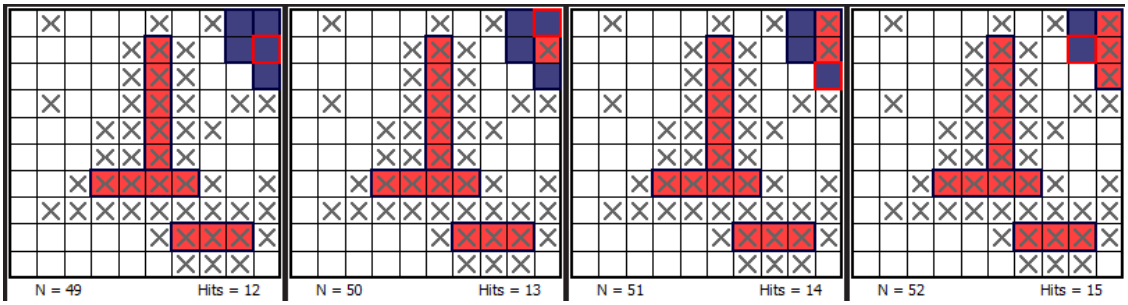
*Au tour 36, nous avons un touché alors aviserTouche()*conserve les cases adjacentes.



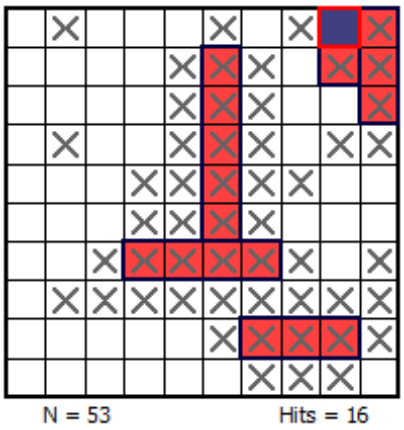
Au tour 40, le croiseur est coulé.



Le raté du tour 45 fait repasser la collection à vide.



Nous avons un nouveau touché au tour 49



Finalement, les navires sont coulés au tour 53.

Vous comprendrez que le jeu consiste à un appel à getTir() suivi d'un appel à aviserTouche() si le tir touche à la flotte. Cette responsabilité a été déléguée au programme principal décrit plus loin et fourni.

3.4 StrategieOrdiAvance

Vous implémentez exactement la même stratégie que la précédente concernant la collection des cibles potentielles dans aviserTouche (exactement pareil).

Dans getTir(), si la collection est vide, les tirs générés sont sur la première diagonale. Si toutes les cases de cette diagonale ont été visitées, on tire sur la deuxième diagonale. Lorsque les deux diagonales ont toutes été visées, on tire au hasard dans des cases non déjà touchées par un tir.

Si la collection n'est pas vide, on commence à obtenir les tirs de la collection des cibles potentielles. On en cherche un pas déjà joué. Si elle devient vide, un tir est choisi au hasard dans les coups non joués,

3.4.1 Description d'attributs la classe nécessaire à getTir()

On définit deux attributs de type booléen. Un pour dire si on traite la première diagonale et un pour la deuxième. La première diagonale est mise à *true* et l'autre à *false* au départ.

On définit un attribut de la classe Coord pour conserver la coordonnée du dernier tir jouée dans une des diagonales (appelons-la cDiag). Elle est mise à (0,0) au départ.

Algorithme de getTir() en pseudocode

```
On se définit une variable locale à retourner de type Coord
```

Si la collection est vide

Si l'attribut booléen pour la première diagonale est à true (vrai au départ)

On commence une boucle et tant que cDiag a déjà été joué, on se déplace sur la diagonale (ligne++ et colonne++). La boucle arrête aussi si la ligne devient égale à Constantes.TAILLE.

Au sortir de la boucle précédente, si la ligne est égale à Constantes.TAILLE c'est que la première diagonale a été jouée en entier, on met alors l'attribut associé à **false** et celui de la deuxième à **true**. Il faut aussi placer cDiag sur la deuxième diagonale (à vous de jouer).

Si l'attribut booléen pour la deuxième diagonale est à true (remarquer qu'il n'y a pas de else).

On commence une boucle et tant que cDiag a déjà été joué, on se déplace sur la deuxième diagonale grâce à cDiag.

Au sortir de la boucle, si la ligne ou la colonne de cDiag est invalide c'est que la deuxième diagonale a été jouée en entier, on met alors le deuxième attribut à **false**.

Si les deux attributs booléens sont à **false** (toujours pas de else)

Il faut placer cDiag avec une coordonnée au hasard qui n'a pas déjà été joué.

On copie cDiag dans la variable qui sera retourné.

Fin (bloc si la collection est vide)

Sinon (la collection n'est pas vide)

On obtient dans la variable de retour et on supprime un à un les tirs de la collection jusqu'à en trouver un qui n'a pas déjà été joués ou que la collection devient vide (boucle).

Si la file est rendu vide en sortant de la boucle, on met un tir qui n'a pas déjà été joués dans la variable de retour.

Fin du sinon

Dans les deux cas (vide ou pas) on ajoute le tir choisi à la collection des tirs joués.

On retourne la variable de retour (pas cDiag)
Fin

Il est important d'avoir en tête que l'algorithme prévoit tous les scénarios possibles lors du choix du tir à jouer. Cet algorithme a lieu une fois par tour

3.5 StrategieOrdiExpert

Partez de la stratégie avancée et essayez d'ajouter une variante qui améliore cette stratégie (pas besoin d'être compliqué).

4. Programme principal et modules utilitaires fournis

Pour vous donner un coup de pouce, nous vous fournissons un programme principal (demarrerBatailleNavale) mais vous devez le modifier pour y ajouter les trois stratégies manquantes (exercice de lecture de code). Nous fournissons aussi deux modules utilitaires. Le premier, UtilitaireFonctions, retourne des coordonnées au hasard et l'autre, Utilitairecollection, permet d'obtenir un coup pas déjà joué dans une collection de type Vector, LinkedList ou ArrayList.

Nous vous fournissons aussi la classe Coord et une classe Constantes à nouveau pour nous assurer qu'elles fonctionnent bien avec notre code.

Finalement : Ajoutez cette méthode à la classe UtilitaireGrilleGui.

```
/**
* Utile pour contrôler le temps de jeu et laisser le temps
* à l'OS de prendre conscience des événements du GUI.
*
* @param tempsDeDelai
*/
public static void pause (int tempsDeDelai){
	try {
		Thread.sleep(tempsDeDelai);
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
}
```

5. Concernant la remise

La remise de la partie 2 est le 2 novembre pour le gr. 1 et le 3 novembre pour le gr. 2. Un lien de remise sera activé sur Moodle. Aucun retard n'est accepté. Vous créez un répertoire compressé (.zip) qui contient seulement le dossier src/.

Il y a moins de code à écrire dans cette partie-ci que dans la partie 1 mais cela vous demande une bonne compréhension. Encore une fois, ne négligez pas le temps de compréhension et le temps de débogage.

Bon travail !