IUT Charlemagne - Dept info 2019-2020
Programmation 1 DUT 1A

Projet à rendre - Le Gardien des Etoiles

---

1 Projet à rendre

Ce projet est un projet à faire de manière **individuelle** et à rendre à votre encadrant via le module *Programmation 1* sur arche. Il donnera lieu à une évaluation qui comptera à hauteur de 15% dans la note finale du module.

Vous devrez rendre **IMPERATIVEMENT** votre projet sur arche **à une date précisée par l'enseignant de votre groupe**. Cela inclut :

- des sources **commentées** qui **compilent** : tout problème dans la compilation ou l'absence de commentaire entraînera des points en moins ;
- une version déjà compilée de votre programme ;
- des classes de test **complétées** et intégralement **commentées** (cf fin de l'énoncé) ;
- un répertoire contenant la **javadoc** générée à partir de votre projet.

ATTENTION :

- Veuillez respecter **IMPERATIVEMENT** les noms des classes et méthodes que l'on vous donne (majuscule comprise, sans accent et sans faute d'orthographe - cf conventions en annexe du polycopié). Respectez aussi l'**ordre** des paramètres dans les fonctions ou méthodes.
- Tous les attributs doivent être déclarés en **private**, les méthodes sont déclarées en **public** sauf précision dans l'énoncé.
- Les méthodes ne doivent contenir aucun `System.out.println` sous peine de ne pas pouvoir être corrigées facilement. Des points pourront être retirés si cette consigne n'est pas respectée.
- Votre projet est à faire **en individuel uniquement**, tous les projets seront testés par une application capable de détecter automatiquement les tentatives de fraude et nous sévirons si besoin (comme cela a déjà été fait par le passé).

2 Présentation du projet

Dans la galaxie Farfaraway, le combat fait rage depuis plusieurs années. Les aliens, envoyés du général des armées Dark Kuroi, s'en prennent aux planètes colonisées par les humains. Le général Kuroi envoie régulièrement des éclaireurs pour trouver les poches de rébellion humaines avant d'envoyer une escouade punitive.

Seul, Herbert Solo est l'ultime rempart contre une attaque éminente. A bord de l'aigle centenaire, son vaisseau de combat, il détruit régulièrement les éclaireurs envoyés et évite que ceux-ci ne transmettent leur rapport sur la présence humaine.

L'objectif du projet est de développer un jeu rendant compte de ce combat. Le jeu sera organisé en 4 classes à écrire :

— la classe `Vaisseau` représentant le vaisseau que le joueur peut déplacer ;
— la classe `Tir` représentant le tir lancé par le vaisseau ;
— la classe `Monstre` représentant les monstres à détruire ;
— la classe `Jeu` rassemblant ces éléments pour modéliser le jeu en cours.

3 Structuration du projet

3.1 Répertoires

Le fichier `.zip` fourni sur arche vous donne une structure à respecter pour votre projet. Il contient la structure de sous-répertoires suivante :

- un répertoire principal `gr_Z_nom` dont le nom est à modifier
  - `Z` doit désigner votre numéro de groupe
  - `nom` doit désigner votre nom
- ce répertoire contient deux sous-répertoires :
  - un répertoire `src` qui doit contenir les fichiers `.java` ;
  - un répertoire `javadoc` qui contiendra le résultat de la javadoc.

3.2 Notion de packages

Le projet est organisé en package. Un package a pour objectif de rassembler des classes. Les classes d'un même package se situent dans le même répertoire.

Chacune des classes du package commence par la ligne de déclaration ``` ``package nom_package;'' ``` où `nom_package` désigne le nom du package. Ces classes doivent rester dans le répertoire du nom du package.

Le nom complet de la classe inclut le nom du package suivi d'un point. Par exemple le véritable nom de la classe `GUI` du package `graphisme` est donc `graphisme.GUI`.

Jusqu'à présent, vous n'aviez qu'un seul package `libtest`. Désormais, le projet est structuré en plusieurs packages réunis dans le répertoire `src` :

- le package `application` contient les classes `Vaisseau`, `Tir`, `Monstre` et `JeuShoot` que vous devrez programmer ;
- le package `test` contient les classes de test à compléter (on sépare les classes de test de l'application) ;
- le package `libtest` contient les classes utiles pour lancer les tests (comme en TP) ;
- le package `graphisme` contient des classes permettant d'interagir avec vos classes à l'aide d'une interface graphique ;
- le package `moteurJeu` contient des classes permettant de lancer vos classes sous la forme d'un jeu temps réel.

Seules les classes des packages `application` et `test` seront à modifier.

3.3 Utilisation des classes d'un package

Il est possible d'utiliser les classes d'un autre package grâce à la commande `import` suivie du nom complet de la classe. Par exemple, ‘‘`import java.util.Scanner`’’ demande à java d'utiliser la classe `Scanner` située dans le package `java.util`.

ATTENTION : Pour pouvoir utiliser correctement les classes des packages, il faut compiler et exécuter vos fichiers à partir du répertoire `src`.

Pour compiler les classes du package, il suffit d'aller **dans le répertoire `src` du projet** et de compiler tous les fichiers `.java` contenus dans le sous-répertoire du package. Par exemple, pour compiler le package `graphisme`, cela donne :

```
1 javac graphisme\*.java
```

Pour lancer la classe principale `Principale` du package `graphisme`, il faut se mettre dans le répertoire `src` du projet [1] et lancer la commande JAVA avec le nom complet de la classe `graphisme.Principale`.

```
1 java graphisme.Principale
```

3.4 le package `graphisme`

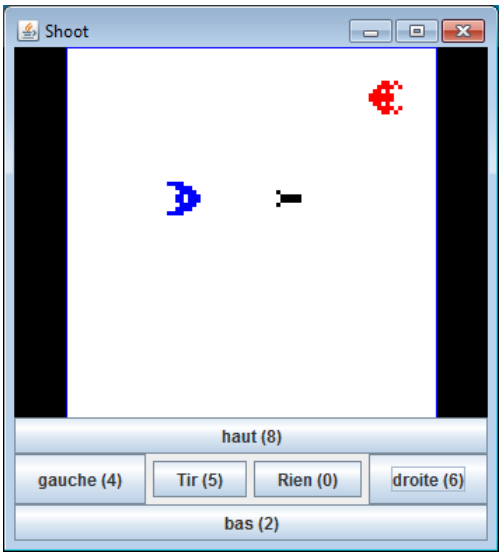

FIGURE 1 – Rendu visuel de l'application avec les classes du package `graphisme`. Chaque bouton permet de transmettre une action au vaisseau. Après chaque action de l'utilisateur, la classe `JeuShoot` fait évoluer le jeu en fonction de la commande passée (le monstre et le tir évoluent automatiquement)

Le package `graphisme` propose une application graphique permettant de vérifier visuellement que votre application fonctionne. Il ne permet pas de tester votre application

1. et PAS dans le sous-répertoire `graphisme`

(ce sont les classes de test qui sont chargées de le faire), mais de pouvoir interagir avec elle. Vous verrez en S2 comment il est possible de construire une interface graphique comme celle présentée ici.

La figure 1 montre le rendu obtenu lorsqu'on exécute

```
java graphisme.Principale
```

Chaque bouton permet de contrôler le vaisseau et utilise les méthodes que vous aurez programmées dans la classe `JeuShoot`. Conformément aux règles du jeu (cf classes décrites ultérieurement), après chaque action joueur, le monstre et le tir éventuel doivent se déplacer.

3.5 le package `moteurJeu`

Un second package nommé `moteurJeu` est fourni pour lancer l'application sous la forme d'un vrai jeu (dans le sens où le temps s'écoule de manière continue). L'affichage reste le même, mais désormais, le vaisseau du joueur se contrôle avec les touches `'Q'`, `'Z'`, `'S'`, `'D'` et la touche `'espace'` du clavier.

Pour cela, il suffit d'exécuter les commandes :

```
javac moteurJeu\*.java
java moteurJeu.Principale
```

4 Descriptif du jeu

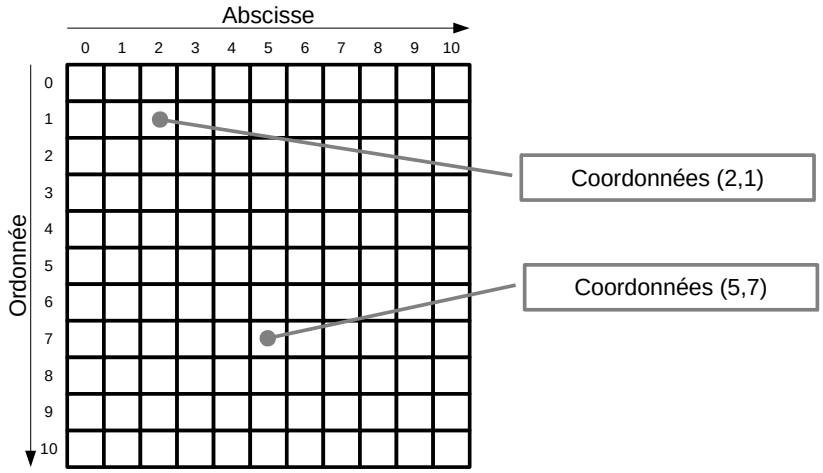

FIGURE 2 – Cette figure présente l'arène dans laquelle le vaisseau et le monstre évoluent. La taille de l'arène est fixe (11 cases sur 11 cases) et les cases sont numérotées de 0 à 10.

Le jeu à programmer est un jeu de shoot'em up consistant à déplacer un vaisseau capable de tirer des projectiles sur un ennemi qui se déplace. Les positions des ennemis, des tirs et du vaisseau du joueur sont restreintes à une arène de 11 sur 11 (cf figure 2) :

les coordonnées des différents objets sont donc comprises dans l'intervalle $[0, 10]$ bornes incluses.

Le joueur dispose de plusieurs commandes. Chaque commande est représentée par un entier. Ces entiers sont choisis en référence à la disposition du pavé numérique d'un clavier :

- '8' permet de monter (diminuer ordonnée de 1) ;
- '2' permet de descendre (augmenter ordonnée de 1) ;
- '4' permet de se déplacer à gauche (diminuer abscisse de 1) ;
- '6' permet de se déplacer à droite (augmenter abscisse de 1) ;
- '0' permet de ne pas bouger ;
- '5' permet au vaisseau de tirer. Un seul tir est possible, quand un tir est déjà sur la zone de jeu, cette action n'a pas d'effet.

Après chaque action effectuée par le joueur, le tir puis le monstre se déplacent :

- le tir avance d'une case vers la droite tant qu'il est sur l'écran ;
- le monstre se déplace verticalement initialement vers le haut. S'il arrive à un bord de l'arène, il avance d'une case vers la gauche et change son sens de déplacement (cf figure 3 dans la classe `Monstre`).

Si, à un instant donné, le monstre et le tir se trouvent sur la même case, le monstre est détruit, le score du joueur augmente de 1 et un nouveau monstre est créé.

Le jeu continue indéfiniment jusqu'à ce qu'un monstre parvienne à traverser tout l'écran. Le joueur a alors perdu et le jeu s'arrête.

5 Classe `Tir`

Écrire la classe `Tir` du package `application` avec les indications suivantes.

5.1 Attributs

Le classe `Tir` a pour objectif de représenter un tir émis par le joueur. Un tir est caractérisé par plusieurs attributs :

- deux entiers `x` et `y` représentant les coordonnées du tir.

5.2 Constructeurs

La classe `Tir` possède un constructeur :

- un constructeur qui prend deux paramètres entiers en entrée `px` et `py` et qui construit un objet de `Tir` situé à la position spécifiée `(px, py)`. On supposera que les coordonnées passées sont des coordonnées valides (c'est à dire correspondant à une position dans l'arène).

5.3 Méthodes accesseurs

Afin de pouvoir accéder aux attributs privés, on vous demande d'écrire des méthodes retournant les valeurs des attributs. Ces méthodes sont appelées accesseurs et s'écrivent

habituellement sous la dénomination anglaise `getNom()` où `Nom` désigne le nom de l'attribut.

Écrire les méthodes `getX` et `getY`.

5.4 Autres Méthodes

- `evoluer` : Écrire la méthode `evoluer`. Cette méthode ne prend pas de paramètre et retourne un booléen. Cette méthode a pour objectif de faire avancer le tir en incrémentant son abscisse de 1. Si le tir doit sortir de l'arène (son abscisse est plus grande ou égale à la taille), il sort effectivement de l'arène mais cette méthode retourne `true` pour préciser que le tir n'est plus visible. La méthode retourne `false` dans les autres cas.
- `toString` : Écrire la méthode `toString`. Cette méthode ne prend pas de paramètre et retourne un `String` représentant l'affichage du tir. Le tir sera représenté par une chaine commençant par `"T"` et précisant ensuite les coordonnées du tir entre parenthèses.
  Par exemple, si le Tir est aux coordonnées (1,2), la méthode `toString` devra retourner la chaine ``` ``T(1,2)'' ```

5.5 Tests

A l'aide des indications section 9, écrire la classe de test `TestTir` associée à la classe `Tir`. Prenez le temps de valider la classe `Tir` avec vos tests avant de passer aux classes suivantes.

6 Classe `Vaisseau`

Écrire la classe `Vaisseau` du package `application` avec les indications suivantes.

6.1 Attributs

Le classe `Vaisseau` a pour objectif de représenter le vaisseau que le joueur contrôle. Un objet `Vaisseau` est caractérisé par plusieurs attributs

- deux attributs entiers `x` et `y` représentant les coordonnées du vaisseau ;
- un attribut `tirCourant` de type `Tir` représentant le tir effectué par le vaisseau (éventuellement `null` si le vaisseau n'a pas de tir en cours).

6.2 Constructeurs

La classe `Vaisseau` possède deux constructeurs :

- un constructeur sans paramètre qui construit un vaisseau par défaut. Ce Vaisseau est situé aux coordonnées `(5,5)`. L'attribut `tirCourant` est initialement `null` ;

- un constructeur qui prend deux paramètres `px` et `py` et qui construit un vaisseau situé en `(px, py)`. Si ces paramètres sont hors de l'arène, le vaisseau construit a pour coordonnées, les coordonnées par défaut `(5,5)`.

6.3 Méthodes Accesseurs

Écrire les méthodes `getTirCourant`, `getX` et `getY`.

6.4 Autres Méthodes

- `deplacer` : Écrire la méthode `deplacer` qui prend deux paramètres `dx` et `dy` et ajoute ces paramètres aux coordonnées du vaisseau. Cette méthode est `private` pour éviter qu'on l'utilise en dehors de la classe (elle n'est donc pas testable directement). Le vaisseau doit rester confiné au sein de l'arène et ne peut pas effectuer un déplacement qui le ferait sortir.

- `tirer` : Écrire la méthode `tirer` qui ne prend aucun paramètre. Cette méthode permet au vaisseau de tirer. Lorsque la méthode est appelée, si le vaisseau n'a pas de tir courant, il en crée un à partir de sa position (voir la classe `Tir`) et met à jour son attribut `TirCourant`. La position initiale du nouveau tir doit alors correspondre à la position du vaisseau.

- `faireAction` : Écrire la méthode `faireAction` qui permet de prendre en compte l'action effectuée par le joueur. Cette méthode prend en paramètre un entier représentant la commande du joueur et modifie l'état du vaisseau en conséquence. Voici les actions à effectuer en fonction de l'entier commande :
  — **0** : le vaisseau ne fait rien ;
  — **4** : le vaisseau se déplace d'une case vers la gauche (-1,0) ;
  — **6** : le vaisseau se déplace d'une case vers la droite (1,0) ;
  — **8** : le vaisseau se déplace d'une case vers le haut (0,-1) ;
  — **2** : le vaisseau se déplace d'une case vers le bas (0,1) ;
  — **5** : le vaisseau tente de lancer un tir.
  C'est par l'intermédiaire de cette méthode publique que la méthode `déplacer` pourra être testée.

- `detruireTir` : Écrire la méthode `detruireTir` qui ne prend aucun paramètre. Cette méthode permet de détruire le tir en cours. L'attribut `TirCourant` est alors remis à `null`.

- `evoluerTir` : Écrire la méthode `evoluerTir` qui ne prend aucun paramètre. Cette méthode permet de faire avancer le tir d'une case si le vaisseau a effectivement un tir en cours. Si le tir sort de l'arène (cf méthode `evoluer` de `Tir`), le vaisseau n'a plus de tir courant (ce qui permet au vaisseau de tirer à nouveau).

- `toString` : Écrire la méthode `toString` qui retourne la chaine `"Vaisseau"` suivie de ses coordonnées entre parenthèses et séparées par une virgule. Si le vaisseau possède en plus un tir, la chaine doit en outre contenir `"-"` suivi de la chaine décrivant le tir.

Ainsi, la chaine associée à un vaisseau ayant un x de 5, un y de 2 et un tir situé en (6,2) doit être ``Vaisseau(5,2)-T(6,2)''.

La chaine associée à un vaisseau ayant un x de 5, un y de 2 sans tir doit être ``Vaisseau(5,2)''.

6.5 Tests

A l'aide des indications section 9, écrire la classe de test `TestVaisseau` associée à la classe `Vaisseau`. Prenez le temps de valider la classe `Vaisseau` avec vos tests avant de passer aux classes suivantes.

7 Classe `Monstre`

Écrire la classe `Monstre` du package `application` avec les indications suivantes.

7.1 Attributs

Le classe `Monstre` a pour objectif de représenter le monstre qui se déplace dans l'arène. Un objet `Monstre` est caractérisé par plusieurs attributs

- deux entiers `x` et `y` représentant les coordonnées du monstre au sein de l'arène ;
- un booléen `deplacerHaut` qui représente la direction de déplacement du monstre. Lorsque ce booléen vaut `true`, le monstre est en train de se déplacer vers le haut, lorsqu'il vaut `false`, le mouvement du monstre est vers le bas (cf image figure 3) ;
- un booléen `etreMort` valant `true` si et seulement si le monstre est mort.

7.2 Constructeurs

La classe `Monstre` possède un constructeur :

- un constructeur qui prend deux paramètres entiers en entrée `px` et `py` et qui construit un objet de `Monstre` situé à la position spécifiée. L'objet `Monstre` construit se déplace initialement vers le haut et n'est pas mort. On vérifiera que le monstre est bien construit dans l'arène. Dans le cas contraire, on le placera au coordonnées `(10,5)`.

7.3 Méthodes accesseurs

Écrire les méthodes `getEtreMort`, `getX` et `getY`.

7.4 Autres Méthodes

- `evoluer` : Écrire la méthode `evoluer`. Cette méthode ne prend pas de paramètre et ne retourne rien. Si le monstre n'est pas mort, cette méthode a pour objectif de déplacer le monstre d'une case selon ses règles de déplacement (cf figure 3). Cette méthode ne doit s'exécuter que si le monstre n'est pas mort. Si le monstre

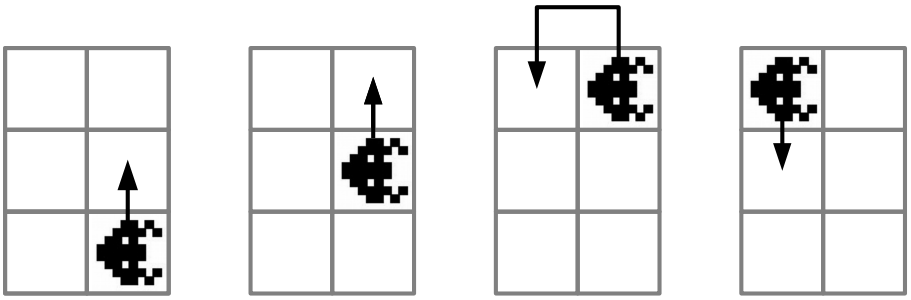

FIGURE 3 – A chaque évolution, le monstre se déplace verticalement quand il le peut. Quand il doit sortir de l'arène, au lieu de sortir, il avance d'une case vers la gauche et change son sens de déplacement. Sur cet exemple, (1) le monstre avance vers le haut ; (2) le monstre avance encore vers le haut ; (3) le monstre devrait sortir de l'arène, il avance d'une case vers la gauche et change son sens de déplacement ; (4) maintenant, le monstre avance vers le bas.

est encore en vie, il se déplace verticalement d'une case dans la direction de son déplacement. Dès que le déplacement du monstre devrait le faire sortir de l'arène, à la place, le monstre avance d'une case vers la gauche et son sens de déplacement est modifié.

- `avoirTraverse` : Écrire la méthode `avoirTraverse`. Cette méthode ne prend pas de paramètre et retourne un booléen. Ce booléen vaut `true` si et seulement si le monstre est arrivé tout à gauche de l'arène (auquel cas le joueur a perdu).
- `avoirCollision` : Écrire la méthode `avoirCollision`. Cette méthode prend un objet de type `Tir` (éventuellement `null`) en paramètre et retourne un booléen. Ce booléen vaut `true` si et seulement si le tir n'est pas `null` et que le monstre se trouve sur la même case que le tir. Dans ce cas, les attributs du monstre doivent en plus se mettre à jour pour refléter que le monstre est mort.
- `toString` : Écrire la méthode `toString`. Cette méthode ne prend pas de paramètre et retourne un `String` représentant l'affichage du monstre. Le monstre sera représenté par une chaine commençant par `"Monstre"` et précisant ensuite les coordonnées du monstre entre parenthèses.
  Par exemple, si le monstre se trouve aux coordonnées (1,2), la méthode `toString` devra retourner la chaine ``` ``Monstre(1,2)'' ```

7.5 Tests

Écrire la classe `TestMonstre` et valider la classe `Monstre`.

8 Classe `JeuShoot`

Écrire la classe `JeuShoot` du package `application` avec les indications suivantes.

8.1 Attributs

Le classe `JeuShoot` a pour objectif de représenter un jeu en cours. Un objet `JeuShoot` est caractérisé par plusieurs attributs

- un objet de type `Vaisseau` nommé `joueur` représentant le vaisseau du joueur ;
- un objet de type `Monstre` nommé `ennemi` représentant l'ennemi affronté par le joueur. Dés qu'un ennemi est touché, il disparait et sera remplacé par un nouvel ennemi.
- un entier `score` correspondant au score du joueur. Ce score s'incrémentera chaque fois que le joueur tuera un ennemi.
- un booléen `perdu` permettant de détecter la fin de partie.

8.2 Constructeurs

La classe `JeuShoot` possède un constructeur par défaut

- ce constructeur ne prend aucun paramètre en entrée. L'objet `JeuShoot` construit initialise le vaisseau du joueur aux coordonnées (0,5), et le monstre aux coordonnées (10,5). Initialement, le score est de 0 et perdu vaut `false`.

8.3 Méthodes accesseurs

Écrire les méthodes `getJoueur()`, `getEnnemi()`, `getPerdu()` et `getScore()`.

Afin de pouvoir tester correctement la classe `Jeu`, on doit pouvoir facilement changer le `Monstre` et le `Vaisseau` du jeu. Pour cela, écrire les méthodes `void setEnnemi(Monstre m)` et `void setJoueur(Vaisseau v)` destinées à modifier les attributs de la classe `Jeu`.

ATTENTION : Ces méthodes de modification d'attribut ne doivent être utilisées que pour faciliter les tests. Vos attributs ne doivent pas être modifiés directement pour une autre raison.

8.4 Autres Méthodes

- `gererCollision` : Écrire la méthode `gererCollision` qui ne prend aucun paramètre et ne retourne rien. Cette méthode vérifie si le tir éventuel du joueur entre en collision avec le monstre (vous penserez à réutiliser des méthodes déjà écrites). Si c'est le cas, le score augmente de 1, un nouveau monstre arrive dans l'arène aux coordonnées `(10,5)` et le tir du vaisseau est détruit ;

- `evoluer` : Écrire la méthode `evoluer`. Cette méthode prend un paramètre entier correspondant à la commande du joueur (cet entier sera fourni par la méthode `demanderJoueur` présentée ci-dessous) et fait évoluer le jeu en conséquence. Cela consiste à
  - (1) faire évoluer le vaisseau du joueur en fonction de la commande ;
  - (2) faire évoluer le tir éventuel ;
  - (3) vérifier les collisions entre le tir et le monstre ;
  - (4) faire évoluer le monstre ;

— (5) vérifier à nouveau les collisions entre le tir et le monstre (après son déplacement) ;
— (6) enfin, mettre à jour le booléen perdu si le monstre a traversé toute l'arène.

8.5 Autres Méthodes - A NE PAS TESTER

Ces méthodes ont pour objectif de fournir une application complète exécutable. Elles effectuent des demandes utilisateurs et des affichages écran. Ces méthodes ne sont donc pas testables automatiquement.

- `demanderJoueur` : Écrire la méthode `demanderJoueur`. Cette méthode ne prend pas de paramètre et retourne un entier correspondant à la commande du joueur. Pour cela, on utilisera la classe `Scanner` pour demander au joueur d'entrer au clavier l'entier correspondant à la commande qu'il souhaite faire.
- `poursuivre` : Écrire la méthode `poursuivre`. Cette méthode consiste simplement à demander au joueur sa commande, à faire évoluer le jeu en conséquence et à afficher le contenu du jeu après évolution. Pour cela, on utilisera les méthodes définies précédemment.
- `toString` : Écrire la méthode `toString`. Cette méthode retourne une chaine décrivant l'état du jeu. Cette chaine soit contenir le descriptif du joueur, un saut de ligne (représenté par le caractère `"\n"`), le descriptif du monstre, un saut de ligne et le score du joueur. Par exemple cela, donnera une chaine de la forme :

```
1 Vaisseau(0,5)
2 Monstre(10,4)
3 0
```

- `main` : Écrire une méthode `main`. Cette méthode doit créer un jeu puis le lancer plusieurs fois (dans l'idéal, cela se ferait avec une boucle tant que le jeu n'est pas perdu). Pour le moment, vous pouvez vous limiter à appeler plusieurs fois la méthode `poursuivre`.

Une fois la méthode `main` écrite, vous pouvez exécuter le programme correspondant en lançant la classe `application.JeuShoot`.

8.6 Tests

Ecrire la classe `TestJeuShoot` et valider la classe `JeuShoot` (sans les méthodes "A ne PAS tester").

Vérifier que toutes vos classes de test sont encore valides avant de rendre le projet.

9 Tests

Avant de rendre votre projet, vous vérifierez que celui-ci fonctionne correctement en développant une classe de test par classe écrite (comme celles que vous avez utilisées pendant les TPs).

Pour cela, pour chaque classe (`Tir`, `Vaisseau`, `Monstre` et `JeuShoot`), vous allez

- réfléchir aux différents tests à écrire ;
- ajouter ces tests dans la classe de test fournie ;
- lancer vos tests et vérifier que votre classe fonctionne correctement par rapport à vos attentes.

9.1 Sélection des tests

Normalement, chaque méthode et constructeur doit être testé pour tous les cas qui peuvent se présenter. Il faut donc dans un premier temps :

- déterminer toutes les méthodes et constructeurs à tester pour la classe qui vous intéresse ;
- déterminer les différents cas à tester pour chaque méthode ;
- pour chaque cas à tester, décider des valeurs de départ utilisées pour faire le test, les valeurs de retour attendues, faire le test et vérifier que les retours des méthodes correspondent bien à vos attentes.

Voici, par exemple, les tests à faire pour tester la méthode `evoluer()` de la classe `Tir`. Il est dit dans l'énoncé qu'un tir peut sortir de l'arène, la méthode doit alors retourner `true`. On doit donc envisager 2 cas à tester

- soit le tir se trouve dans une case loin du bord (on pourra prendre comme valeur de départ `(7,5)`), le tir évolue et la méthode doit retourner `false` ;
- soit le tir se trouve au bord de l'arène (on pourra prendre comme valeur de départ `(10,5)`), le tir évolue et la méthode doit retourner `true`.

9.2 Ecriture de la classe de Test

Une fois l'ensemble des cas de test déterminés, il reste à vérifier que tous les tests sont validés lorsque votre programme s'exécute. Vous complèterez la classe de test correspondante (`TestTir`, `TestVaisseau`, `TestMonstre` ou `TestJeuShoot`) en ajoutant une méthode de test pour chacun des cas identifiés.

9.2.1 Organisation des méthodes de test

Les méthodes de test doivent **IMPERATIVEMENT** suivre les conventions de nommage suivantes :

- chaque méthode de test se trouve dans la classe correspondant à la classe à tester (par exemple dans la classe `TestTir` pour les méthodes de `Tir`) ;
- le nom de chaque méthode de test
  - débute par le mot `'test'` ;
  - est suivie par le nom de la méthode testée précédé d'un underscore, par exemple `'_Evoluer'` ;
  - puis d'un descriptif **sans accent** du cas testé, par exemple `'_ResteArene'` ;

Ainsi, la méthode vérifiant que la méthode `evoluer` de la classe `Tir` fonctionne bien quand le tir ne sort pas de l'arène s'appellera `'test_Evoluer_ResteDansArene()`. Elle se trouvera dans la classe `TestTir`.

Pour finir, chaque méthode de test est précédée d'un commentaire qui explique le cas que teste la méthode, par exemple `‘‘vérifier que le tir avance bien et que la méthode retourne false quand il reste dans l’arène’’`.

9.2.2 Ajout d'une méthode de test

Une méthode de test a pour objectif de vérifier que le test est effectivement valide.

- la méthode ne prend pas de paramètre et ne retourne rien ;
- les premières instructions de la méthode préparent les données (en appelant des constructeurs par exemple) ;
- les instructions suivantes exécutent le test à proprement parler (à savoir la méthode testée avec les bonnes données) ;
- les instructions de vérification appellent la méthode `assertEquals` à chaque fois qu'une condition attendue doit être vérifiée ;
- vous pouvez utiliser les accesseurs pour vérifier les valeurs des attributs mais les accesseurs n'ont pas besoin d'être testés (ils sont très simples).

Une fois qu'une méthode de test est écrite, elle doit rester dans votre classe de test. Ainsi, à l'issue du projet, vos classes de test possèderont toutes les méthodes de tests (une par cas à considérer) et permettront de tester l'ensemble de l'application.

9.2.3 Méthode `assertEquals`

Pour effectuer une vérification dans un test, il faut utiliser la méthode `assertEquals`. Cette méthode possède le profil suivant :

`void assertEquals(String erreur, Object attendu, Object obtenu)`

- `erreur` correspond au message d'erreur à afficher si la vérification n'est pas correcte ;
- `attendu` désigne la valeur attendue ;
- `obtenu` désigne la valeur obtenue lors du test.

Par exemple, si on veut tester que le '+' correspond bien à la concaténation, on ajouterait la chaine `"bon"` à la chaine `"jour"` et on vérifierait que le résultat serait bien la chaine `"bonjour"`.

```
1 String s1="bon";
2 String s2="jour";
3 String s3=s1+s2;
4 assertEquals("erreur: mauvaise concaténation","bonjour",s3);
```

9.2.4 Exemple

Pour le test de la méthode `evoluer` de la classe `Tir`, la classe `TestTir` s'écrirait de la manière suivante :

```
1 /******************
2 * test classe Tir
3 ******************/
```

```java

public class TestTir{

  /**
  * evoluer quand le tir reste dans l'arene
  */
  public void test_Evoluer_ResteDansArene() {
    // preparation des données
    Tir t = new Tir(6, 5);

    // methode testee
    boolean res = t.evoluer();

    // verifie retour false
    assertEquals("tir devrait rester dans arene", false, res);
    // verifie position aprés evolution
    assertEquals("tir devrait evoluer en X", 7, t.getX());
    assertEquals("tir devrait rester en Y", 5, t.getY());
  }

  /**
  * evoluer quand le tir sort de l'arene
  */
  public void test_Evoluer_SortirArene() {
    // preparation des données
    Tir t = new Tir(10, 5);

    // methode testee
    boolean res = t.evoluer();

    // verifie retour true car sortie de l arene
    assertEquals("tir devrait sortir de arene", true, res);
    // verifie position apres evolution
    assertEquals("tir devrait evoluer en X", 11, t.getX());
    assertEquals("tir devrait rester en Y", 5, t.getY());
  }

  //... autres tests de la classe Tir
}
```

9.3 Classes de test fournies

On vous fournit sur l'ENT le package `libtest` ainsi qu'un début des classes de test `TestTir.java`, `TestVaisseau.java`, `TestMonstre.java` et `TestJeuShoot.java` dans le package `test`.

Les classes de test fournies possèdent déjà :
- une méthode `main` qui lance tous les tests que vous aurez écrits dans la classe ;
- un premier test qui vérifie que vos méthodes sont correctement écrites ;
- quelques autres tests qui vérifient vos constructeurs (pour être sûr que vous créez les bons objets).

Ces classes doivent compiler correctement si vos méthodes sont bien déclarées. Il ne faut pas changer les tests initiaux qui vérifient que vos méthodes sont conformes au sujet.

Une fois que vos tests seront ajoutés à cette classe, l'exécution de tous les tests ne doivent conduire à aucune erreur (puisque tous vos tests doivent être validés). Faire de bons tests est un moyen de vous assurer de rendre un projet conforme aux attentes.

ATTENTION, **il ne vous est pas demandé de rendre un menu**, mais de rendre les classes de tests complétées par vos méthodes de test. Tout rendu autre que ce qui est attendu ne sera pas évalué.

9.4 Démarche

Pour faire correctement votre projet, nous vous conseillons de penser à tous les tests possibles dans un premier temps (cf partie sélection des test), de les programmer et ensuite seulement de commencer à écrire les différentes classes de votre programme. Cela vous permettra

— d'avoir pensé à tous les cas particuliers lorsque vous écrirez votre programme ;
— de disposer d'un programme plus sûr.

10 Génération de la javadoc

Pensez à écrire la `javadoc` dans vos fichiers sources et générez la javadoc dans le répertoire `javadoc` situé à la racine de votre projet.

Pour cela, le plus simple est de se placer dans le répertoire `gr_Z_nom` et d'exécuter la commande `javadoc`

- avec l'option `'-d dest'` pour spécifier le répertoire de destination (ici le répertoire de destination sera `javadoc`) ;
- avec comme argument les fichiers sources du package `application` situés dans `src`, à savoir `'src/application/*.java'`.

```
1 javadoc -d javadoc src/application/*.java
```

La `javadoc` générée est à rendre avec votre projet.

Une fois la javadoc générée, il est possible de la consulter en ouvrant le fichier `javadoc/index.html`.

11 Dernier conseil

Le projet que vous allez rendre forme un tout (classes + tests + javadoc).
— Si vous faites les bons tests, vous pourrez facilement détecter les erreurs de votre projet et le corriger.
— Inversement si les tests ne sont pas complets, vous risquez d'avoir des classes qui fonctionnent mal et un projet qui ne fournit pas les bons résultats.

Ainsi, le meilleur moyen d'avoir une bonne note est de mettre l'accent sur les tests qui vous permettront d'écrire un code correct et valide (et bien entendu de corriger les classes lorsque les tests ne passent pas).

Pour information, le fait d'écrire les tests dans le corrigé a permis d'éviter la présence de plusieurs erreurs d'inattention dans le corrigé du projet. Ils vous assurent aussi qu'une fois que votre projet est fini, tous les tests restent valides (vous n'avez donc pas introduit une nouvelle erreur dans votre programme en faisant une modification).