Projet à rendre - Legolas contre l'orc ninja Uruk-Hai

(12 octobre 2023)

---

1 Projet à rendre

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

Vous devrez rendre **IMPERATIVEMENT** pour **le mardi 14/11/2023** sur arche :

- des sources **commentées** qui **compilent** : tout problème dans la compilation ou l'absence de commentaire entrainera 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

Le projet va avoir pour objectif de représenter (très simplement) un combat entre le puissant Legolas et l'orc ninja de la tribu Uruk-Hai. De manière plus précise, il s'agit de programmer plusieurs classes pour modéliser des guerriers et leurs arcs :

- la classe `Arc` a pour objectif de représenter un arc avec ses dégats et les flèches disponibles dans son carquois ;
- la classe `Guerrier` a pour objectif de représenter un guerrier avec son nom, ses points de vie et son arc.

3 Structuration du projet

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

- 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 plusieurs sous-répertoires
  - un répertoire `src` qui doit contenir les fichier .java ;
  - un répertoire `javadoc` qui contiendra le résultat de la javadoc.

4 La classe `Arc`

Question 1

Écrire la classe `Arc` dans le fichier `Arc.java` avec les indications suivantes.

4.1 Attributs

Le classe `Arc` a pour objectif de représenter une arme qu'un guerrier pourra utiliser pour combattre d'autres guerriers. Un arc est caractérisé par plusieurs attributs privés :

- un attribut entier `degats` précisant les dégâts que fait l'arc lorsqu'il est utilisé ;
- un attribut entier `fleches` désignant le nombre de flèches restantes dans le carquois de l'arc.

4.2 Constructeurs

La classe `Arc` possède deux constructeurs :

- un constructeur sans paramètre qui construit un arc par défaut (avec 5 dégats et 3 flèches) ;
- un constructeur avec deux paramètres entiers `dg` et `fl` qui construit un arc faisant `dg` dégats et possédant `fl` flèches.

Les dégâts et le nombre de flèches doivent être positifs ou nuls. Lorsque l'un des paramètres est négatif, la valeur de l'attribut correspondant doit être de 0.

4.3 Méthodes Accesseurs

Écrire les méthodes `getDegats` et `getFleches`.

Attention ! Sauf indication contraire, ces accesseurs ne doivent servir que dans vos méthodes de test pour vérifier les bonnes valeurs des attributs.

4.4 Autres Méthodes

- `recharger` : Écrire la méthode `recharger` qui prend en paramètre un entier `nFleches` et ajoute ces `nFleches` nouvelles flèches au nombre de flèches du carquois. Lorsque le nombre de flèches passé en paramètre est négatif, rien ne doit se passer.

- `utiliser` : Écrire la méthode `utiliser` qui ne prend aucun paramètre et consiste à utiliser l'arc. Cette méthode réduit de un le nombre de flèches disponibles et retourne les dégats de l'arc. Si le nombre de flèches est déjà égal à 0 quand la méthode est appelée, le nombre de flèches ne diminue pas et l'arc ne fait pas de dégats (les dégats retournés par la méthode valent 0).

- `toString` : Écrire la méthode `toString` qui retourne la chaine `"-arc"` suivie de ses `degats` et du nombre `fleches` de flèches restantes sous la forme suivante [1]
  `"-arc(d:`*`degats`*`,f:`*`fleches`*`)"`
  où *degats* et *fleches* représentent les valeurs de ces attributs.
  Ainsi, la chaine retournée pour un arc faisant 2 de dégâts et possédant 5 flèches doit être `"-arc(d:2,f:5)"`

4.5 Tests

Question 2

A l'aide des indications de la section 6, écrire la classe de test `TestArc` associée à la classe `Arc`. N'oubliez pas de tester toutes les situations possibles (un test différent par situation).

5 La classe `Guerrier`

Question 3

Écrire la classe `Guerrier` dans le fichier `Guerrier.java` avec les indications suivantes.

5.1 Attributs

Le classe `Guerrier` a pour objectif de représenter un guerrier spécialiste dans le maniement de l'arc. Un guerrier est caractérisé par plusieurs attributs privés :

- un attribut `String nom` correspondant au nom de l'archer (`"Legolas"`, ...) ;
- un attribut `pv` de type entier correspondant au nombre de points de vie du guerrier. Cet attribut ne peut jamais être négatif.
- un attribut `arc` de type `Arc` permettant de savoir quel arc le guerrier possède. Si l'archer n'a pas d'arc sur lui, cet attribut vaut `null`.

1. parenthèses incluses, pas d'accent, pas d'espace entre les virgules ou les :

5.2 Constructeurs

La classe `Guerrier` possède deux constructeurs :

- un constructeur qui prend en entrée un seul paramètre `pNom` de type `String` correspondant au nom du guerrier. Ce constructeur construit un guerrier dont le nom correspond à `pNom`, qui possède 10 points de vie et qui ne possède pas d'arc ;

- un constructeur qui prend deux paramètres en entrée : un `String pNom` correspondant au nom du guerrier et un entier `p` correspondant aux points de vie initiaux du guerrier. Ce constructeur construit un guerrier dont le nom correspond à `pNom`, qui possède `p` points de vie et qui ne possède pas d'arc. Si le paramètre `p` est inférieur à 1, le guerrier doit tout de même avoir 1 point de vie.

5.3 Méthodes accesseurs

Écrire les méthodes `getPv`, `getArc` et `getNom`.

Attention ! Ces accesseurs ne doivent servir que dans vos méthodes de test pour vérifier les bonnes valeurs des attributs. **Ils ne doivent pas être utilisés dans les classes `Guerrier` et `Arc` - sauf indication contraire dans le descriptif de la méthode**.

5.4 Autres méthodes

- `etreBlesse` : Écrire la méthode `etreBlesse` qui ne prend pas de paramètre et retourne un booléen qui vaut `true` si et seulement si le guerrier est à 0 points de vie.

- `subirDegat` : Écrire la méthode `subirDegat` sans résultat qui prend un entier `degat` en paramètre et réduit les points de vie du guerrier de ces dégâts. Les dégâts ne peuvent pas être négatifs (sinon rien ne se passe) et un guerrier ne peut jamais avoir moins de 0 points de vie (si un guerrier arrive à moins de 0 points de vie, ses points de vie sont remis à 0).

- `prendreArc` : Écrire la méthode `prendreArc`. Cette méthode prend un paramètre `arc` de type `Arc` correspondant à l'arc que le guerrier souhaite prendre et retourne un booléen. Le guerrier prend l'arc si et seulement si il ne possède pas déjà d'arc. Lorsque le guerrier ne possède pas d'arc, l'arc passé en paramètre devient l'arc du guerrier et la méthode retourne `true`. Si le guerrier possède déjà un arc, rien ne se passe et la méthode doit alors retourner `false`. Cette méthode ne fait quelque chose que si le guerrier n'est pas blessé.

- `poserArc` : Écrire la méthode `poserArc`. Cette méthode ne prend aucun paramètre en entrée et permet à un guerrier de poser un arc qu'il aurait en sa possession. Cette méthode met à jour les attributs concernés et retourne un objet de type `Arc`. Si le guerrier a un arc, il pose cet arc et la méthode retourne l'arc posé. Si le guerrier n'a pas d'arc en sa possession lorsque la méthode est

appelée, la méthode doit retourner `null`. Cette méthode ne peut fonctionner que si le guerrier n'est pas blessé.

- `attaquer` : Écrire la méthode `attaquer` qui permet à un guerrier d'attaquer un autre guerrier. Cette méthode prend en paramètre la cible de l'attaque `victime` de type `Guerrier`. Cette méthode consiste à retirer les points de dégâts de l'attaque aux points de vie de la cible. Bien entendu, un guerrier ne peut attaquer que s'il n'est pas blessé. La méthode retourne un booléen qui vaut `true` si et seulement si l'attaque a été lancée (et a fait des dégats > 0). Vous ferez attention à ce que les dégats correspondent bien aux dégats de l'arc du guerrier (et que les flèches diminuent bien). Si un guerrier ne possède pas d'arc, il inflige 0 dégat à sa victime.

- `toString` : Écrire la méthode `toString` qui permet d'afficher le statut du guerrier. Cette méthode retourne une chaine de la forme :
  nom`(`*pv*`)-arc(d:`*degats*`,f:`*fleches*`)`
  où
  — *nom* désigne le nom du guerrier ;
  — *pv* désigne les points de vie du guerrier ;
  — *degats* désigne les dégâts de l'arc possédé par le guerrier ;
  — *fleches* désigne le nombre de flèches dans le carquois de l'arc du guerrier.
  Si le guerrier ne possède pas d'arc, l'affichage se limite à *nom*`(`*pv*`)`, par exemple, `"Legolas(10)"`.

5.5 Tests

Question 4

Écrire la classe de test `TestGuerrier` qui teste l'ensemble des méthodes de `Guerrier` pour l'ensemble des situations proposées.

5.6 Main à écrire

En plus des tests, on souhaite écrire un programme principal qui vérifie que les méthodes fonctionnent bien. Le programme principal doit effectuer les opérations suivantes dans l'ordre :

1. dans une variable `legolas`, créer un `Guerrier` nommé `"Legolas"` avec 4 points de vie (Légolas sort d'un combat difficile) ;
2. dans une variable `orc`, créer un `Guerrier` nommé `"Ugluk"` avec 3 points de vie ;
3. faire attaquer `Ugluk` par `Legolas` : Legolas n'a pas d'arme et ne fait aucun dégât ;
4. créer un objet `arcElfique` faisant 2 dégats avec 1 flèche ;
5. faire prendre l'`arcElfique` par `Legolas` ;
6. faire attaquer `Ugluk` par `Legolas` : Legolas tire une flèche sur Ugluk et fait 2 dégats, Ugluk a 1 point de vie.

7. faire attaquer `Ugluk` par `Legolas` : Legolas n'a plus de flèche et fait 0 dégât ;
8. faire prendre par `Ugluk` un arc `orcArc` qui fait 5 dégât avec 3 flèches ;
9. faire attaquer `Legolas` par `Ugluk` : Legolas prend 5 dégât et est blessé ;
10. dans une variable `arwen`, créer un `Guerrier` nommé `"Arwen"` avec 10 points de vie ;
11. faire prendre par `arwen` l'`arcElfique` ;
12. recharger l'arc elfique avec 1 flèche ;
13. faire attaquer `Ugluk` par `Arwen` : Ugluk prend 2 dégât et est blessé ;
14. faire attaquer `Arwen` par `Ugluk` : Ugluk est blessé et son attaque échoue, il s'effondre, terrassé !! (grande fête chez les elfes !!)

Question 5

Écrire le `main` correspondant dans la classe `ProgCombat` et vérifier (en affichant les guerriers au fur et à mesure que le combat se déroule comme prévu).

6 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 (`Arc` et `Guerrier`), vous allez

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

6.1 Sélection des tests

Normalement, chaque méthode et chaque 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 le constructeur à deux paramètres de la classe `Arc`. Il est dit dans l'énoncé que si un paramètre est négatif, la valeur associée à l'attribut doit être nulle. On peut donc envisager quatre cas d'utilisation du constructeur :

- soit les deux paramètres passés sont positifs ;

- soit le paramètre correspondant aux dégâts est négatif ;
- soit le paramètre correspondant au nombre de flèches est négatif ;
- soit les deux paramètres sont négatifs.

6.2 Écriture de la classe de Test

Une fois l'ensemble des cas de test déterminé, il reste à vérifier que tous les tests sont validés lorsque votre programme s'exécute. Vous complèterez les classes de test correspondantes (`TestArc` et `TestGuerrier`) en ajoutant une méthode de test pour chacun des cas identifiés.

6.2.1 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 `TestGuerrier` pour les méthodes de `Guerrier`) ;
- 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 `'_Recharger'` ;
  - puis d'un descriptif **sans accent** du cas testé, par exemple `'_RechargeOK'` ;

Ainsi, la méthode vérifiant que la méthode `recharger` de la classe `Arc` fonctionne bien quand on recharge avec un nombre positif de flèches s'appellera `'test_Recharger_RechargerOK()`. Elle se trouvera dans la classe `TestArc`.

Pour finir, chaque méthode de test est précédée d'un commentaire javadoc qui explique le cas que teste la méthode, par exemple « `Test recharger arc avec un nombre de fleches positif` ».

6.2.2 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.

6.2.3 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"`.

```
String s1="bon";
String s2="jour";
String s3=s1+s2;
assertEquals("erreur: mauvaise concatenation","bonjour",s3);
```

6.2.4 Exemple

Pour le test de la méthode `recharger` de la classe `Arc`, la classe `TestArc` s'écrirait de la manière suivante :

```
import static libtest.Lanceur.lancer;
import static libtest.OutilTest.assertEquals;
import libtest.*;

/******************
 * test classe Arc
 ******************/
public class TestArc {

  /**
   * methode de lancement des tests
   */
  public static void main(String[] args) {
    lancer(new TestArc(), args);
  }


  /**
   * quand l'arc est recharge correctement
   */
  public void test_recharger_OK() {
    // preparation des donnees
    Arc arc = new Arc(3,5);

    // methode testee
    arc.recharger(2);

    // verifications
    assertEquals("arc doit toujours faire 3 degats", 3, arc.getDegats());
    assertEquals("arc doit avoir 7 fleches", 7, arc.getFleches());
```

```java
  }

  /**
   * quand l'arc est recharge avec un nombre de fleches negatif
   */
  public void test_recharger_negatif() {
    // preparation des donnees
    Arc arc = new Arc(3,5);

    // methode testee
    arc.recharger(-2);

    // verifications
    assertEquals("arc doit toujours faire 3 degats", 3, arc.getDegats());
    assertEquals("arc doit toujours avoir 5 fleches", 5, arc.getFleches());
  }


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

6.3 Classes de test fournies

On vous fournit sur l'ENT le package `libtest` ainsi qu'un début des classes de test `TestArc.java` et `TestGuerrier.java`.

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 doit 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.

6.4 Démarche

Pour faire correctement votre projet, nous vous conseillons de penser à tous les tests possibles dans un premier temps (cf partie sélection des tests), 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.

7 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 situés dans `src`, à savoir `'src/*.java'`.

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

8 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).