R2-01-03 : TP 3

Développement orienté objets & Qualité de développement

Pour commencer

Continuer dans le projet crée pour le TP1 et TP2, pour se faire **créer** un package tp3.

Continuer à utiliser les outils mis à votre disposition vu en R1.01 et au début de ce module : les mécanismes automatiques dans l'IDE, le deboggueur, etc.

Exercice 1 : Billetterie

Objectifs R2-01 : héritage, association et polymorphisme.

La SNCF souhaite ajouter un billet réduit à sa billetterie automatique. Un billet réduit a son prix réduit de 20% par rapport à un billet classique.

Dans le paquetage **`tp3`**, créer le package **`billetterie`** et les classes **`Trajet`**, **`Billet`**, **`BilletReduit`**, **`BilletUtilitaire`** et **`TestBillet`**.

Compléter les classes **`Trajet`**, **`Billet`** et **`BilletReduit`** en respectant le diagramme UML ci-après. La classe **`Billet`** contient un objet de type **`Trajet`** (représenter par la flèche entre les deux classes). La classe **`BilletReduit`** hérite de la classe **`Billet`**.

Quelques contraintes supplémentaires seront à prendre en compte pour cette spécification :

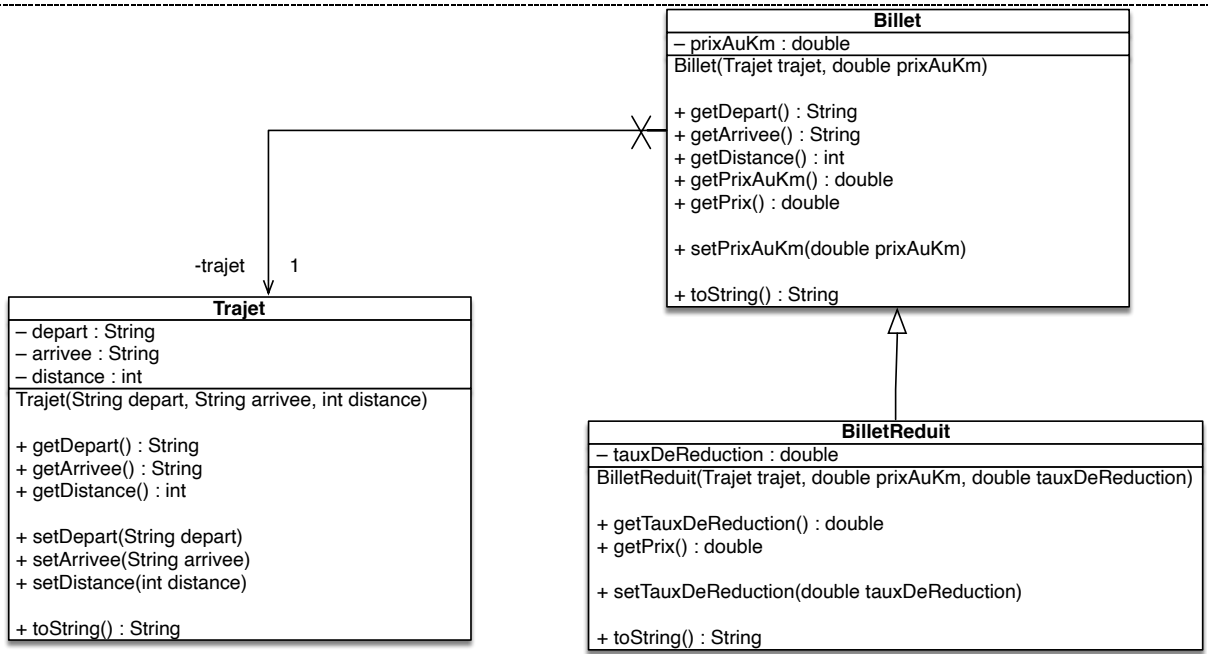
- Pour la classe **`Trajet`**, les villes de départ et d'arrivée seront en majuscule et la distance entre 2 villes est comprise entre 5 et 2000 km (bornez la distance par son min ou son max : si la distance donnée est < 5, la distance prend comme valeur 5 et si la distance donnée est > 2000, la distance prend comme valeur 2000) ;
- Pour la classe **`Billet`**, le prixAuKm (prix au kilomètre) est compris entre 0.1 et 2 (bornez le prix au kilomètre par son min ou son max). **Pour rappel**, le prix d'un billet est calculé en faisant le produit de la distance (du trajet) et du prix au kilomètre ;
- Pour la classe **`BilletReduit`**, le taux de réduction est compris entre 5% et 50% (bornez le taux de réduction par son min ou son max). **Pour rappel**, le prix d'un billet réduit est le prix du billet normal auquel s'applique le taux de réduction.
- Dans la classe BilletterieUtilitaire, **créer** une méthode static **`arrondir(double prix)`** qui retourne le prix arrondi à deux chiffres après la virgule (plusieurs solutions

sont possibles, trouvez s'en une). Cette méthode sera utilisée pour calculer le prix des billets.

- Pour la classe **`Trajet`**, **`BilletReduit`** et **`BilletReduit`**, les méthodes **`toString()`** retourne une chaîne de caractères représentant leurs caractéristiques, se reporter à la trace attendue ci-après.

Aide : le méthode **`this.getClass().getSimpleName()`** retourne le nom de la classe de l'objet courant, exemple :

```
@Override
public String toString() {
    return  this.getClass().getSimpleName();
}
```



CONSEIL : réalisez les classes sans lien d'agrégation ou d'héritage en premier et **testez les** (ici, la classe **`Trajet`**). Puis réalisez et **testez** les classes avec des agrégations (ici, la classe **`Billet`**). Puis réalisez et **testez** les classes filles – avec liens d'héritage (ici, la classe **`BilletReduit`**). **POUR RESUMER**, ne développez pas tout d'un coup ! Découpez votre travail et testez rapidement !

Vérifier avec le test donné ci-après et la trace attendue (attention à l'arrondi). Le mécanisme de **polymorphisme** s'applique ici : les billets réduits ne sont rien d'autres que des billets. Ils peuvent donc être placés dans une liste de billets. Cependant, les méthodes appelées seront bien les méthodes de leur classe « d'origine » (Billet ou Billet Reduit).

`TestBilletterie`

```
// Creation d'une liste de trajets
ArrayList<Trajet> trajets = new ArrayList<>();

// Creation de trajets
Trajet T1, T2, T3;
T1 = new Trajet("Paris", "Grenoble", 576);
T2 = new Trajet("Grenoble", "Marseille", 307);
T3 = new Trajet("Grenoble", "Voreppe", 16);

// Ajout des trajets dans la liste
```

```
trajets.add(T1);
trajets.add(T2);
trajets.add(T3);

// Affichage des trajets
BilletterieUtilitaire.afficheTrajets(trajets);

// Creation d'une liste de billets
ArrayList<Billet> billets = new ArrayList<>();

// Creation de billets "normaux"
Billet B1, B2, B3;
B1 = new Billet(T1, 0.53);
B2 = new Billet(T2, 0.43);
B3 = new Billet(T3, 0.82);

// Ajout des billets "normaux" dans la liste de billets
billets.add(B1);
billets.add(B2);
billets.add(B3);

// Affichage de la liste de billets
BilletterieUtilitaire.afficheBillets(billets);

// Creation de billets réduits
BilletReduit BR1, BR2, BR3;
BR1 = new BilletReduit(T1, 0.53, .2);
BR2 = new BilletReduit(T2, 0.43, .3);
BR3 = new BilletReduit(T3, 0.82, .4);

// Ajout des billets réduits dans la liste de billets
billets.add(BR1);
billets.add(BR2);
billets.add(BR3);

// Affichage de la liste de billets
// Vérifier le prix pour chaque billet normal ou réduit
// Le mécanisme de polymorphisme s'applique ici !
BilletterieUtilitaire.afficheBillets(billets);
```

Trace attendue

```
----- Les trajets -----
Trajet : PARIS -> Grenoble (576 km)
Trajet : GRENOBLE -> Marseille (307 km)
Trajet : GRENOBLE -> Voreppe (16 km)

----- Les billets -----
Billet : [Trajet : PARIS -> Grenoble (576 km)], prix : 305.28 euros
Billet : [Trajet : GRENOBLE -> Marseille (307 km)], prix : 132.01 euros
Billet : [Trajet : GRENOBLE -> Voreppe (16 km)], prix : 13.12 euros

----- Les billets -----
Billet : [Trajet : PARIS -> Grenoble (576 km)], prix : 305.28 euros
Billet : [Trajet : GRENOBLE -> Marseille (307 km)], prix : 132.01 euros
Billet : [Trajet : GRENOBLE -> Voreppe (16 km)], prix : 13.12 euros
BilletReduit : [Trajet : PARIS -> Grenoble (576 km)], prix : 244.22 euros,
avec une réduction de 20.0%
BilletReduit : [Trajet : GRENOBLE -> Marseille (307 km)], prix : 92.41
euros, avec une réduction de 30.0%
```

```
BilletReduit : [Trajet : GRENOBLE -> Voreppe (16 km)], prix : 7.87 euros,
avec une réduction de 40.0%
```

Exercice 2 : Collections et comparaison

Objectifs R2-01 : comprendre les interfaces et leur usage.
Objectifs R2-03 : utiliser les interfaces **`Comparable`** et **`Comparator`**, utiliser une collection spécifique

Créer un paquetage **`etudiant`** dans le package **`tp3`**

Exercice 2.1 : Etudiants triés dans l'ordre naturel (interface Comparable)

On définit l'ordre naturel de tri sur les **`Etudiant`**s comme suit : tri en ordre croissant sur le **`nom`** et pour le même nom tri en ordre croissant sur le **`prenom`**. Il faut donc que la classe **`Etudiant`** implante (**`implements`**) l'interface **`Comparable`** comme illustré dans diagramme ci-dessous. Pour implanter l'interface **`Comparable`**, la classe **`Etudiant`** doit redéfinir (**`@override`**) la méthode **`compareTo()`**.

Mettre à jour la classe **`Etudiant`** dans le package **`tp2.universite`** pour qu'elle intègre cette spécification.

```
public class Etudiant extends Personne implements Comparable<Etudiant> {
```

CONSEIL : Pour réaliser la méthode **`compareTo(…)`** de la classe **`Etudiant`**, vous pouvez utiliser la méthode **`compareTo(…)`** de la classe **`String`**. Pensez à regarder la documentation pour savoir ce qu'elle retourne.

Dans le paquetage **`tp3.etudiant`**, créer une classe **`TestOrdreNaturelEtudiant`**.

Compléter la procédure principale **`main`** de **`TestOrdreNaturelEtudiant`** pour correspondre aux commentaires ci-dessous :

```
// Créer les étudiants

// Créer la liste d'étudiants quelconque et ajouter les étudiants
ArrayList<Etudiant> mesEtudiantsOrdreQcq = new ArrayList<>();

// Afficher la liste d'étudiants quelconque

// Créer l'ensemble d'étudiants trié et ajouter les étudiants
TreeSet<Etudiant> mesEtudiantsTries = new TreeSet<>();

// Afficher l'ensemble d'étudiants trié
```

La variable **`mesEtudiantsTries`** est de type **`TreeSet`** pour pouvoir stocker les **`Etudiant`**s dans une collection qui utilise l'ordre établit par la comparaison.

Pour afficher les deux collections (**`ArrayList`** et **`TreeSet`**), créer une méthode static **`affichageEtudiants(Collection<Etudiant> etudiants)`** qui affiche sur le terminal une collection d'étudiant (**`List`** ou **`Set`**).

```
private static void affichageEtudiants(Collection<Etudiant> etudiants) {
    for(Etudiant etudiant : etudiants) {
        System.out.println(etudiant.getNom() + ", " + etudiant.getPrenom());
```

```
    }
}
```

Trace attendue :

```
--------------
Les étudiants : mesEtudiantsOrdreQcq
Sanz, Floriant
Porte, Pierre-antoine
Burlat, Nils
Brunet-manquat, Raphaël
Brunet-manquat, Maxime
--------------
Les étudiants triés : mesEtudiantsTries
Brunet-manquat, Maxime
Brunet-manquat, Raphaël
Burlat, Nils
Porte, Pierre-antoine
Sanz, Floriant
```

Exercice 2.2 : Classement des étudiants selon leur moyenne (interface Comparator)

Objectifs R2-03 : utiliser l'interface **`Comparator`**

On voudrait obtenir la liste des étudiants triés sur la moyenne. La classe Etudiant implantant déjà l'ordre naturel, il faut passer par un comparateur.

Mettre en place l'une des deux solutions possibles pour implanter un comparateur sur la moyenne pour les **`Etudiant`**s : soit une classe autonome, soit comparateur static final.

Dans le paquetage **`tp3.etudiant`**, créer une classe **`TestEtudiantTriMoyenne`**.

Compléter la procédure principale **`main`** de **`TestEtudiantTriMoyenne`** pour correspondre aux commentaires ci-dessous :

```
// Créer les étudiants

// Ajouter des notes aux étudiants

//  Créer la liste d'étudiants et ajouter les étudiants
ArrayList<Etudiant> mesEtudiantsTries = new ArrayList<>();

// Afficher la liste des étudiants avant tri

// trier la liste
// Collections.sort(mesEtudiantsTries, …);

// Afficher la liste des étudiants après tri
```

Trace attendue :

```
--------------
Les étudiants avant Collections.sort() :
Moyenne = 16.0 : Sanz, Floriant
Moyenne = 9.0 : Porte, Pierre-antoine
Moyenne = 13.0 : Burlat, Nils
Moyenne = 19.0 : Brunet-manquat, Raphaël
Moyenne = 18.0 : Brunet-manquat, Maxime
```

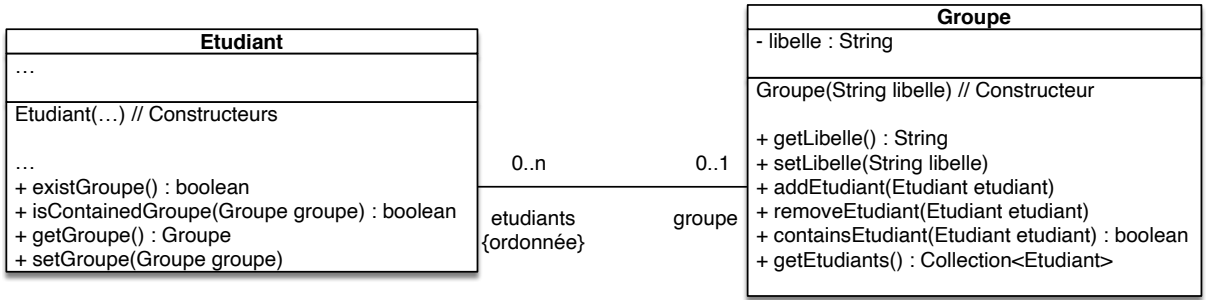
```
--------------
Les étudiants après Collections.sort() :
Moyenne = 9.0 : Porte, Pierre-antoine
Moyenne = 13.0 : Burlat, Nils
Moyenne = 16.0 : Sanz, Floriant
Moyenne = 18.0 : Brunet-manquat, Maxime
Moyenne = 19.0 : Brunet-manquat, Raphaël
```

Exercice 2.3 : un groupe d'étudiants

Objectifs R2-03 : associations bidirectionnelles

On définit maintenant une classe **`Groupe`** pour représenter différents groupes d'**`Etudiant`**s. Le modèle de la classe **`Groupe`** est très simple : un groupe a un **`libellé`** (nom du groupe), et un **`Groupe`** peut contenir plusieurs étudiants dont on veut qu'ils soient triés dans l'ordre naturel définit sur **`Etudiant`**.

Le modèle des classes **`Groupe`** et **`Etudiant`**, ainsi que l'association bidirectionnelle entre **`Etudiant`** et **`Groupe`** est donné par le diagramme UML suivant :



Dans le paquetage **`tp2.universite`**, créer la classe **`Groupe`** et compléter la classe **`Etudiant`** pour qu'elle implante le diagramme UML.

Attention : vous aurez deux classes **`Groupe`** une dans le package **`tp1.universite`** et une dans le package **`tp2.universite`**

Dans le paquetage **`tp3.etudiant`**, créer une classe **`TestEtudiantGroupe`** et ajouter le jeu de test suivant dans la procédure principale **`main`** :

`TestEtudiantGroupe`

```
// Créer des groupes
Groupe groupeA = new Groupe("A");
Groupe groupeB = new Groupe("B");

// Créer des étudiants
Etudiant etudiant1 = new Etudiant("SANZF", "FLORIANT", "Sanz");
Etudiant etudiant2 = new Etudiant("portepa", "Pierre-Antoine", "Porte");
Etudiant etudiant3 = new Etudiant("burlatn", "nils", "burlat");
Etudiant etudiant4 = new Etudiant("brunetr", "Raphaël", "Brunet-Manquat");
Etudiant etudiant5 = new Etudiant("brunetm", "Maxime", "Brunet-Manquat");

// Ajouter TOUS les étudiants au groupe A
// ATTENTION à ne pas boucler à cause de l'association bidirectionnelle !
groupeA.addEtudiant(etudiant1);
groupeA.addEtudiant(etudiant1);     // On ajoute plusieurs fois le même
étudiant
groupeA.addEtudiant(etudiant1);     //
groupeA.addEtudiant(etudiant2);
groupeA.addEtudiant(etudiant3);
```

```java
groupeA.addEtudiant(etudiant4);
groupeA.addEtudiant(etudiant5);

// Ajouter les deux derniers étudiants au groupe B
etudiant4.setGroupe(groupeB);
etudiant5.setGroupe(groupeB);

// Afficher les étudiants du groupe A
System.out.println("---------");
System.out.println("Etudiants du Groupe A");
for (Etudiant etudiant : groupeA.getEtudiants()) {
    System.out.println(etudiant.getGroupe().getLibelle() + " " +
etudiant.getNom() + " " + etudiant.getPrenom());
}

// Afficher les étudiants du groupe B
System.out.println("---------");
System.out.println("Etudiants du Groupe B");
for (Etudiant etudiant : groupeB.getEtudiants()) {
    System.out.println(etudiant.getGroupe().getLibelle() + " " +
etudiant.getNom() + " " + etudiant.getPrenom());
}
```

REMARQUE : si l'on ajoute plusieurs fois le même **`Etudiant`** dans un **`Groupe`** l'étudiant n'apparaitra qu'une seul fois. La collection utilisée dans le groupe étant un **`TreeSet`** un ensemble trié donc sans doublons !

CONSEIL : dans les associations bidirectionnelles, vous devez faire attention à ce que les données soit synchronisées et aux problèmes de cycles qui peuvent en découler ! Dans notre exemple, dans le cas d'un ajout d'étudiant au groupe ou d'un groupe à un étudiant.

Trace attendue :

```
---------
Etudiants du Groupe A
A Burlat Nils
A Porte Pierre-antoine
A Sanz Floriant
---------
Etudiants du Groupe B
B Brunet-manquat Maxime
B Brunet-manquat Raphaël
```

IMPORTANT : avant de passer à la suite du TP (exercice bonus), demandez à votre enseignant de valider votre travail !

Exercice 3 : Exercice « fil rouge » : La bataille de Faërun (étape 3 et 4)

Cet exercice sera présent dans chaque TP et permettra de revenir sur les notions abordées dans les TDs et les TPs sous la forme d'un petit jeu sur le terminal. Cette séquence n'est pas forcément à commencer dès les premières semaines, elle pourra être commencée à tout moment quand vous serez plus à l'aise avec la notion d'objet. **Si vous n'avez pas encore réalisé l'étape 1, rendez-vous à la fin du TP1 !**

IMPORTANT : cet exercice sera évalué dans le cadre de la ressource R02.03 : bonne pratique, réalisation, documentation, gestion de version, test, etc.

Dans la suite, vous compléterez et modifierez les classes du package ***jeu****.*

Etape 3 : création du château - entrainement des guerriers novices

Un château contient des guerriers novices (normalement donné par le joueur). A chaque tour, le château entraine un nombre limité de guerriers novices en fonction de ses ressources. La Figure 1 propose une solution possible (d'autres existes) pour réaliser la classe **`Chateau`**. La méthode **`entrainer()`** retourne une liste de guerriers entrainés (0 ou plusieurs guerriers) en fonction des ressources disponibles dans le château à un instant i. Les guerriers ont un certain coût d'apprentissage (d'entrainement) pour pouvoir aller combattre (se reporter aux documents sur Chamilo).

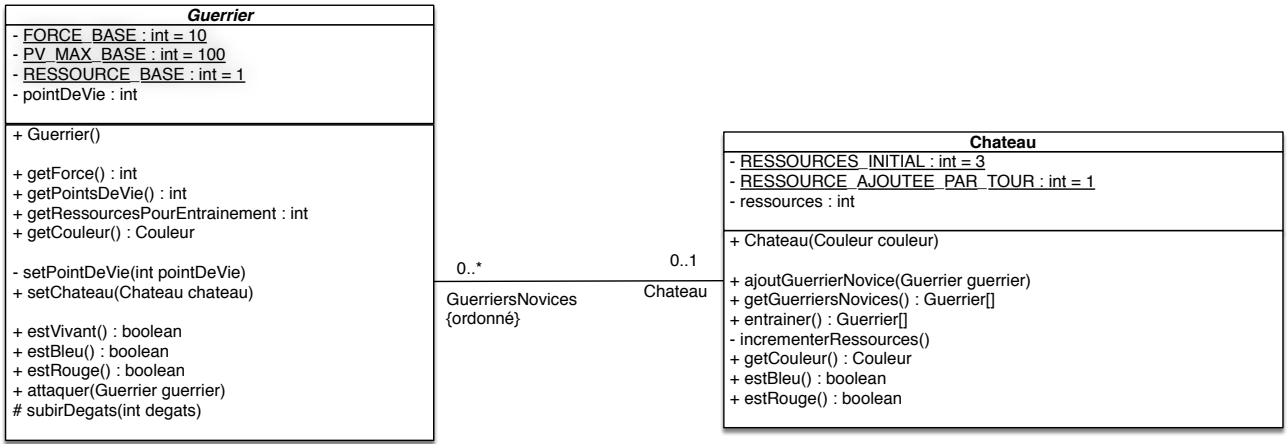


Figure 1 : diagramme UML classe Guerrier et Chateau

Etape 4 : création du plateau - déplacement des guerriers

Deux équipes vont s'affronter : les bleus et les rouges. Le plateau est leur champ de bataille. Vous devez dans un premier temps, réaliser les classes **`Plateau`** et **`Carreau`** en vous aidant de la Figure 2 (une solution possible parmi d'autres). Pour réaliser la méthode **`deplaceGuerriers(),`** se reporter à l'explication sur Chamilo faerunExplicationDeplacement.pdf.

CONSEIL : décomposez vos tests ! Testez dans un premier temps le déplacement des bleus (de gauche à droite – un seul guerrier puis plusieurs), puis testez le déplacement des rouges (de droite à gauche) et enfin testez le déplacement des bleus et des rouges en même temps (avec arrêt quand ils se rencontrent sur le même carreau) !

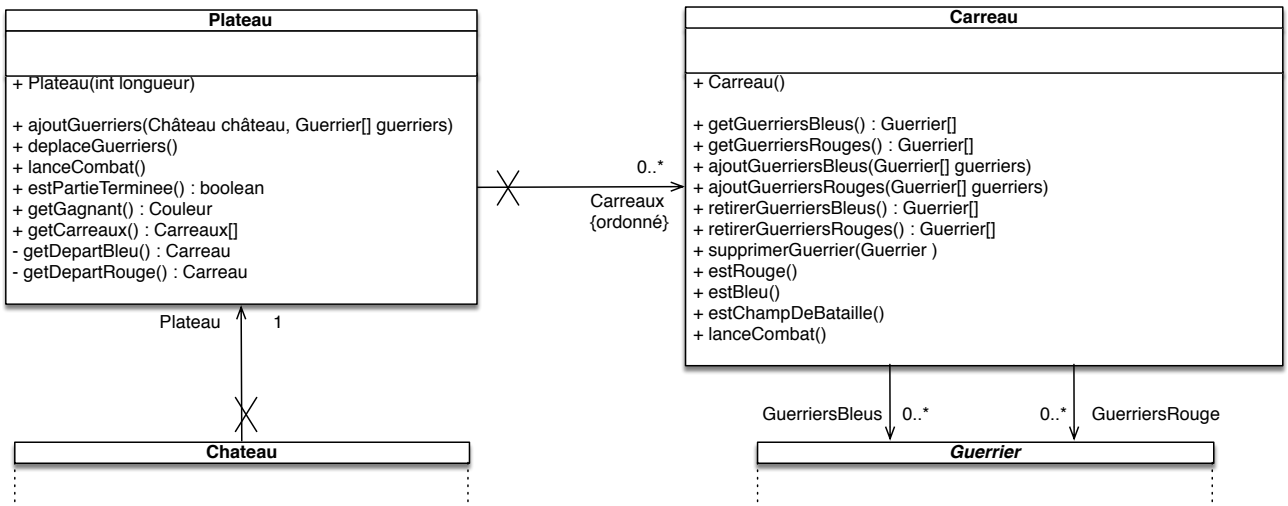


Figure 2 : diagramme UML classe Plateau et Carreau

à suivre …

Etape 5 : plateau - mêlée sanglante (combat des guerriers sur le même carreau)

Etape 6 : réaliser le moteur de jeu (itération comportant les tours de jeu jusqu'à la victoire d'un des joueurs)

ATTENTION, plusieurs solutions sont possibles pour réaliser la demande. MAIS nous souhaitons une solution avec un héritage et orientée objet ! Donc n'ajoutez pas d'attributs qui ne sont pas dans le diagramme. L'important est de comprendre que les spécificités des guerriers nain, elfe, etc doivent se retrouvent dans leurs classes !

Si vous hésitez sur une solution, demandez à votre enseignant.