Exercice 4 – héritage

Voici un exercice récapitulatif des 4 premiers modules.

Vous allez devoir, dans un premier temps, faire le diagramme UML correspondant à l'énoncé. Ensuite, seulement, vous programmerez la solution en Python.

Battle DD - analyse

BattleDD est un jeu qui voit deux équipes constituées de divers personnages se combattre.

Diagramme UML

Dans un premier temps, établissez le diagramme UML correspondant à l'énoncé suivant. Pour chaque classe, indiquez les attributs et les méthodes qui ressortent de l'énoncé.

Chaque combat (pour lequel on retient la date) voit 2 équipes s'affronter. Chaque équipe possède un nom, un nombre de parties jouées, un nombre de parties perdues et un nombre de parties gagnées. Une équipe est constituée de 0 à 5 personnages actifs et d'un nombre illimité de personnage de réserve (aucun lors de sa création).

Chaque personnage a un nom, de la force, de la vie, de l'armure, appartient à une (et une seule) équipe et est actif ou non. Tous les personnages peuvent attaquer quelqu'un, prendre des dégâts ou se faire soigner. Il doit également être possible de changer le statut du personnage (passer d'actif à non actif et vice versa).

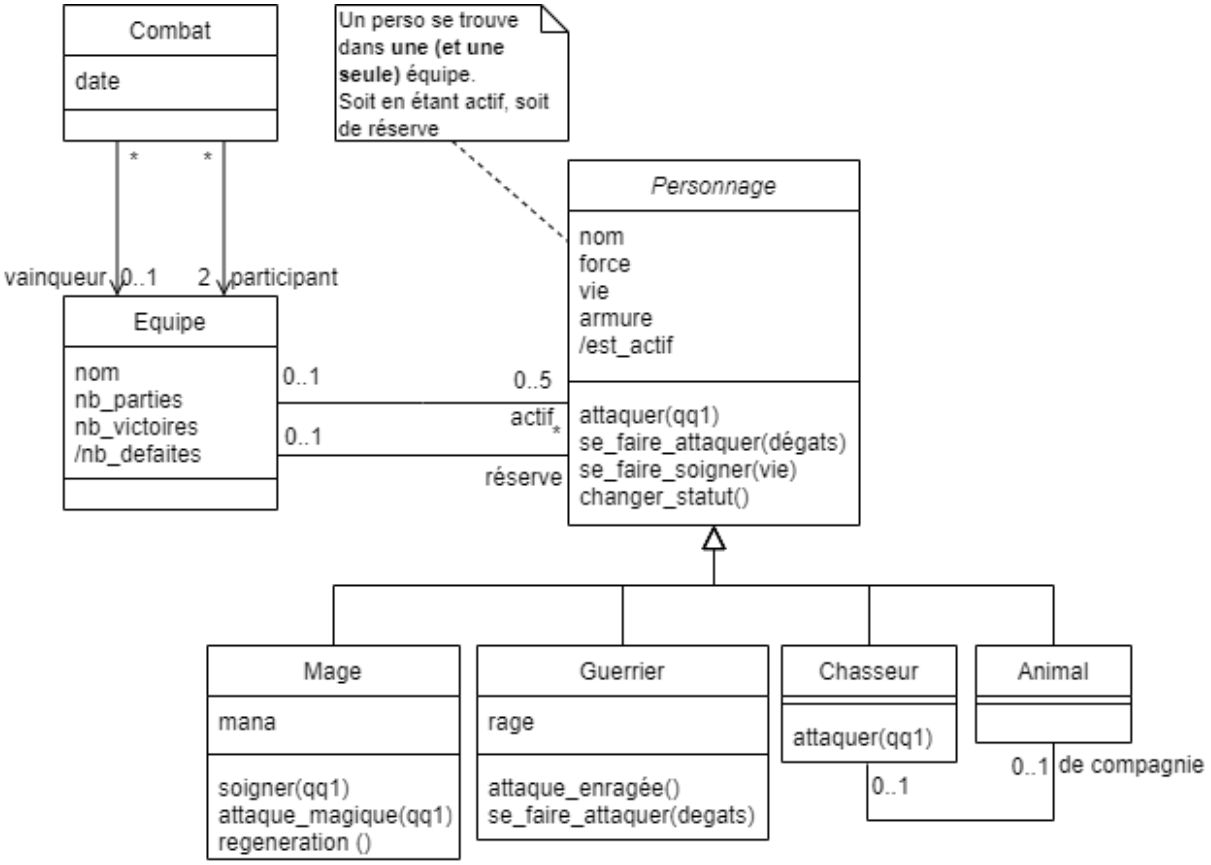
Il y plusieurs types de personnages : les mages, les guerriers, les chasseurs et les animaux. Un mage est un personnage qui possède, en plus, du mana. Il peut soigner, faire une attaque magique ou régénérer son mana. Un Guerrier possède quant à lui de la rage et peut faire une attaque enragée. Il redéfinira la méthode qui prend des dégâts car il ne les prend pas de la même manière que les autres. Les chasseurs peuvent avoir un animal (de compagnie) et leur attaque est différente de celle des autres personnages. Les animaux ont maximum un maître (chasseur).

À la fin du combat, on veut retenir l'équipe vainqueur. De plus, il n'est pas nécessaire de savoir retrouver, à partir d'une équipe, tous les combats auxquels elle a participé.

Pour être sûr que vous partiez sur de bonnes bases pour la programmation, une solution vous est donnée à la page suivante. Cependant, il serait judicieux de ne pas bêtement regarder cette solution mais d'essayer d'en faire une avant puis de corriger…
PS : à l'examen, vous devrez bien le faire tout seul…

Avant de programmer

Voici le diagramme UML qui ressort de l'énoncé précédent :

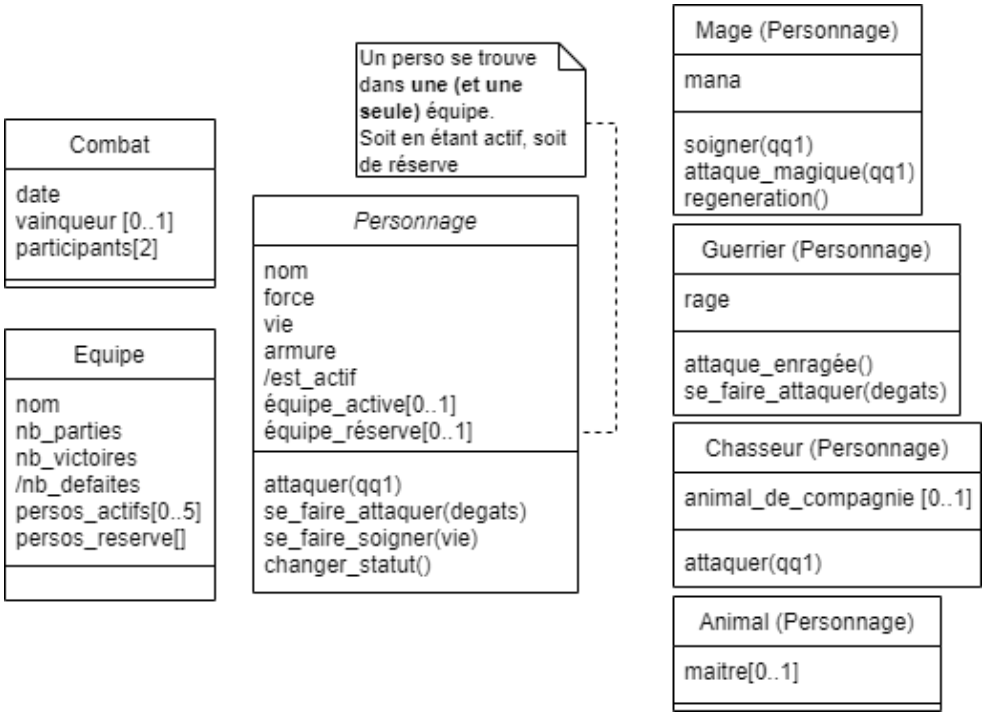


Dans un premier temps, réfléchissez à comment vont se traduire tous les liens entre les classes.

Encore une fois, pour éviter un faux départ, une solution vous est donnée à la page suivante. Et, encore une fois, il serait judicieux de ne pas bêtement regarder cette solution. À l'examen, il vous sera donné un diagramme UML à traduire en classe. Ce sera à vous de penser à transformer les liens entre classes CORRECTEMENT…

Petite réflexion

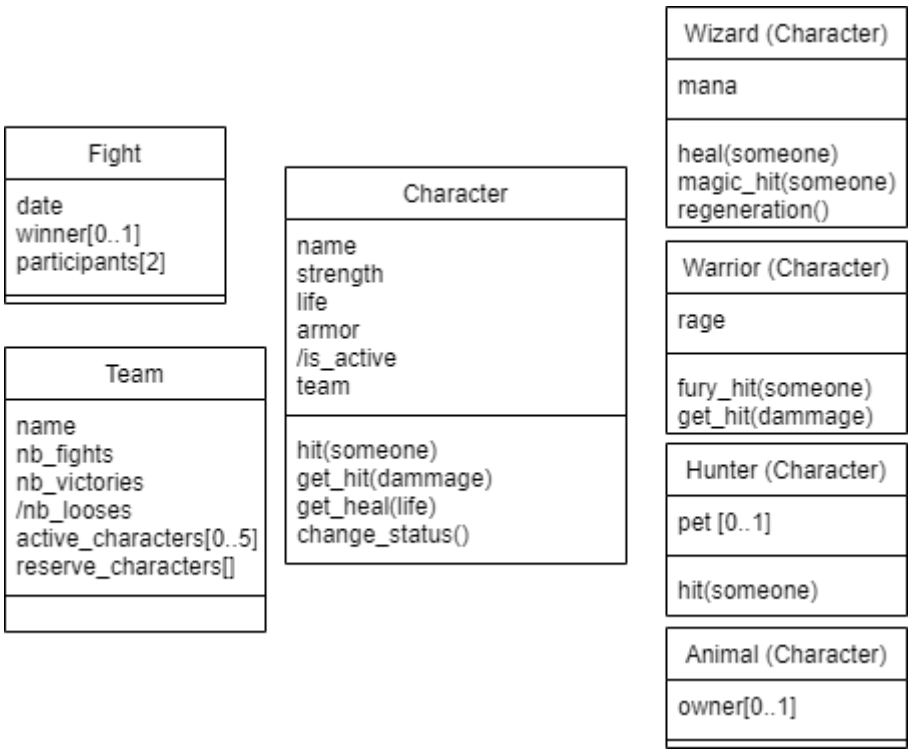
Voici à priori la « traduction » du diagramme UML en classes simples :



Prenons 2 minutes pour analyser le commentaire qui concerne la classe Personnage et plus particulièrement les attributs équipe_active et équipe_reserve.
En effet, les deux attributs sont facultatifs mais l'un ou l'autre sera toujours garni (mais pas en même temps). Au final, un personnage appartiendra à une et une seule équipe à la fois.

➔ **Nous allons donc remplacer ces 2 attributs facultatifs par un seul obligatoire : équipe.**

Voici donc le schéma modifié (et en anglais pour la suite) :



Battle DD - programmation

Chaque énoncé de cette partie est donné comme il serait donné en examen.

Mais étant donné que ce n'est pas encore l'examen, que nous ne sommes pas là pour vous aiguiller en cas de problème, que nous ne voudrions pas que vous restiez bloqués sur un exercice et encore moins que vous déprimiez à cause du cours de programmation (et pour éviter les spams), des petites aides sont régulièrement données en *gris italique*.

Dans un premier temps, essayer de faire sans et si vous êtes bloqué ou en manque d'inspiration, lisez-les. Si, malgré l'aide, vous êtes encore bloqué, envoyez-nous un petit mail. Essayez d'éviter les essais-erreurs. En effet, à l'examen, vous ne pourrez pas tester votre code sur un compilateur… Essayez donc de bien réfléchir avant de programmer.

L'exercice est conséquent mais suivez les étapes une à une. Le problème dans sa globalité est compliqué à implémenter en une fois. Mais, séparément, chaque petite question n'est pas si compliquée que ça. Et si on vous le donne comme exercice, c'est qu'on pense que vous en êtes capables.

Bon travail ☺

(Si vous en avez marre, faites une petite pause et revenez-y plus tard… vous avez les semaines de Pâques pour le terminer)

Classe Team (Equipe)

1. **Implémentez la classe `Team`** dont le constructeur ne prend qu'un seul paramètre : le nom.
   Les autres attributs sont d'office à 0 (pour les entiers) ou à liste vide (pour les listes). Tous les attributs sont publics.

2. **Prévoyez un accesseur pour l'attribut dérivable.**

`nb_looses` est un attribut dérivable. Il ne sera donc pas présent comme attribut dans le constructeur. Néanmoins, il faut quand même prévoir un accesseur pour celui-ci pour qu'il soit possible d'accéder à celui-ci depuis l'extérieur comme s'il existait.

Pour ce faire : écrivez l'accesseur comme si l'attribut `nb_looses` existait. Puis, plutôt que de mettre `return self._nb_looses`, vous allez calculer le nombre de défaites et retourner cette valeur.
De cette manière, de l'extérieur de la classe, on pourra faire, par exemple, `print(my_team.nb_loses)` comme si `nb_looses` était un attribut… Vive l'encapsulation.

3. **Prévoyez une méthode `add_reserve_character()` et `add_active_character()`** qui prennent un personnage en paramètre et l'ajoutent respectivement à la liste des joueurs en réserve ou la liste des joueurs actifs.
   Attention : il ne peut jamais y avoir plus de 5 personnages dans `active_character`. Si on tente d'y ajouter un 6ème joueur, il sera ajouté à `reserve_character` à la place.

Classe Fight (Combat)

1. **Implémentez la classe `Fight`** avec un constructeur qui prend en paramètre uniquement `team1` et `team2`, les deux équipes qui prendront part au combat. Le constructeur définit `date` comme étant la date d'aujourd'hui[1], `winner` vaut `None` pour le moment et `participants` est un tuple qui contient les deux équipes passée en paramètre.
   De plus, à la création du combat, ajouter dans les stats des deux équipes 1 combat

« Lors de la création du combat », c'est-à-dire dans le constructeur, ajoutez aux deux équipes 1 à nb_fight.

2. Mettez vos trois attributs en **privé** et écrivez pour chacun un **accesseur**.
   Il n'est pas nécessaire de prévoir de modificateur. En effet, il n'est pas prévu de pouvoir modifier un des attributs en dehors de la classe.

[1] Utilisez la classe datetime (voir atelier 3)

Classe Character (Personnage)

1. **Implémentez la classe `Character`** dont le constructeur prend en paramètre le nom du personnage, son équipe, sa force, sa vie et son armure. Lors de la création du personnage, celui-ci est automatiquement ajouté à la liste des joueurs de réserve de son équipe.

« Lors de la création du personnage », c'est-à-dire dans le constructeur, ajoutez le personnage à reserve_character de l'équipe grâce à la bonne méthode.

Plus particulièrement la méthode `add_reserve_character` appelée sur `self._team` et qui prend le personnage courant en paramètre (donc `self`). Ce qui donne : `self._team.add_reserve_character(self)`

2. Prenez en compte le fait que, une fois les valeurs des attributs définies, les attributs ne **sont plus modifiables** depuis l'extérieur.

Vous devez donc rendre les attributs privés et n'écrire que des sélecteurs pour chacun d'entre eux.

3. **Prévoyez un accesseur pour l'attribut dérivable.**

De la même manière que pour nb_looses dans Team.

4. **Implémentez la méthode `get_heal()`** qui prend un entier en paramètre et ajoute cet entier à la vie du personnage.

5. **Implémentez la méthode `get_hit()`** qui prend la quantité de dommages en paramètre. Les dégâts réellement pris par le personnage est la valeur de ces dommages moins la valeur de son armure. Le joueur ne peut en aucun cas regagner de la vie via cette méthode.
   De plus, si la vie de personnage arrive à 0 (ou moins), changez son statut (grâce à `change_status ()` implémentée au point 7)

Donc veillez à ne pas supprimer la différence entre les dommages et l'armure si cette différence est négative !

6. **Implémentez la méthode `hit()`** qui prend en paramètre un personnage et va lui infliger des dommages équivalent à la force de celui qui attaque.

Pensez que vous ne pouvez pas accéder à la vie de personnage attaqué. Il faudra donc passé par une des méthodes que vous avez implémentée.

7. **Implémentez la méthode `change_status()`** qui s'occupe de déplacer le personnage dans la liste de réserve de l'équipe s'il était actif et dans la liste des personnages actifs sinon.

Pour cela, utilisez is_active. Et n'oubliez pas qu'il ne fait pas simplement ajouter le joueur à la nouvelle liste mais aussi penser à le supprimer de l'autre liste (avec remove).

8. **Redéfinissez la méthode `__str__()`** pour que la description d'un personnage corresponde à : <nom> **(**<vie>**-**<force>**-**<armure>**)**

__str__() doit retourner une chaine de caractères.
Ce qui est entre <> doit être remplacé par les valeurs des attributs correspondant. Ce qui est en gras reste inchangé.

Classe Wizard (Mage)

1. **Implémentez la classe `Wizard`** dont la valeur initiale de chaque attribut est passée en paramètre. Par défaut, le mana est à 5.

N'oubliez pas que Wizard hérite de Character. Dans le constructeur de Wizard, faites appel au constructeur de Character.

Pour le premier, si vous avez vraiment du mal, voici l'exemple :

```
class Wizard (Character):
    def __init__(self, name, team, strength, life, armor, mana=5):
        super().__init__(name, team, strength, life, armor)
        self._mana = mana
```

2. Prenez en compte le fait que, une fois le mana défini par le constructeur, il n'est **ni modifiable ni consultable** depuis l'extérieur.

Vous devez donc rendre l'attribut mana privés et ne pas écrire de sélecteur ni de modificateur.

3. **Implémentez la méthode `regeneration()`** qui ajoute 2 au mana.

4. **Implémentez la méthode `heal()`** qui prend la personne à soigner en paramètre. Si le mage a encore du mana, il soigne cette personne d'un nombre de PV équivalents à la moitié (arrondie à l'entier inférieur) du mana restant. Le mage perd ensuite un de mana.

Si le mage a encore du mana signifie que la valeur de l'attribut mana doit être supérieure à 0.
La moitié arrondie à l'entier inférieur correspond à une division entière.
N'oubliez pas que vous n'avez pas accès à la vie de la personne soignée. Il faut donc passer par une méthode…

5. **Implémentez la méthode `magic_hit()`** qui prend la personne à attaquer en paramètre. Si le mage a encore du mana, il inflige à cette personne des dégâts équivalents à la quantité de mana dont il dispose. Le nombre de mana du mage est ensuite divisé par 2 (arrondi à l'entier inférieur).

N'oubliez pas que vous n'avez pas accès à la vie de la personne attaquée. Il faut donc passer par une méthode…

6. **Redéfinissez la méthode `__str__()`** pour que la description d'un mage corresponde à : **Wizard** <description du personnage> **[**<mana>**].**

Ce qui est entre <> doit être remplacé par les valeurs des attributs correspondant. Ce qui est en gras reste inchangé.
Pour la description du personnage, pensez à la méthode __str__ définie dans la superclasse.

→ `super().__str__()` ou `str(super())`

Classe Warrior (Guerrier)

1. **Implémentez la classe `Warrior`.** Le constructeur prend en paramètre le nom, l'équipe, la force, la vie et l'armure. La rage est toujours à 0 à la création du personnage et est un **attribut privé inaccessible** (ni en lecture ni en écriture) de l'extérieur.

N'oubliez pas que Warrior hérite de Character.
De plus, rage est privé et n'a ni sélecteur ni de modificateur.

2. **Implémentez la méthode `fury_hit()`** qui prend la personne à attaquer en paramètre et inflige à cette personne des dégâts équivalents à la force du guerrier fois la moitié (arrondie à l'entier inférieur) de la rage du guerrier. La rage du guerrier retombe alors à 0.

3. **Redéfinissez la méthode `get_hit()`** pour le guerrier. En effet, un guerrier ne prend pas les coups comme les autres personnages. La méthode prend toujours la quantité de dommages en paramètre. Mais si le coup touche le guerrier, sa rage augmente de 1.
   Si la vie de personnage arrive à 0 (ou moins), n'oubliez pas de changer son statut.

« Si le coup touche le guerrier » signifie en fait si les dommages réellement subits (donc après avoir retiré les points d'armure) sont positifs.

4. **Redéfinissez la méthode `__str__()`** pour que la description d'un guerrier corresponde à : **Warrior** <description du personnage> **[**<rage>**].**

Le principe est le même que pour Wizard

Classe Animal

1. **Implémentez la classe `Animal`.** Le constructeur prend en paramètre le nom, l'équipe, la force et la vie. L'armure d'un animal vaudra toujours 0 et à sa création, il ne possède aucun maitre.

Encore une fois, n'oubliez pas que Animal hérite de Character. Réutilisez le constructeur de Character en passant 0 pour le paramètre armor.

2. **Rendez l'attribut `owner` privé** et écrivez lui un sélecteur et un modificateur.

3. **Redéfinissez la méthode `__str__()`** pour que la description d'un animal corresponde à : **Animal** <description du personnage> **owned by** <nom du chasseur>.
   Si l'animal n'a pas de propriétaire, n'ajoutez pas le « owned by … ».

Le principe est le même que pour Wizard et Warrior cependant, vous devez ajouter une condition pour voir s'il a un propriétaire. Dans l'affirmative, vous devez rajouter « owned by… » à la chaine de caractère à renvoyer.

Classe Hunter (Chasseur)

1. **Implémentez la classe `Hunter`.** Le constructeur prend en paramètre le nom, l'équipe, la force, la vie et l'armure. À la création du personnage, il ne possède aucun animal de compagnie.

Encore une fois, n'oubliez pas que Warrior hérite de Character.
De plus, pet vaudra None pour le moment

2. **Rendez l'attribut `pet` privé** et écrivez lui un sélecteur et un modificateur.
   Le modificateur de pet, qui prend un animal en paramètre, va d'abord vérifier que l'animal fait partie de la même équipe que le chasseur avant de modifier l'attribut pet.
   Si l'animal et le chasseur ne sont pas dans la même équipe, il ne se passe rien.
   Si l'animal et le chasseur font partie de la même équipe, l'animal devient l'animal de compagnie du chasseur et ce dernier devient son maitre.

Le sélecteur sera comme d'habitude.
Le modificateur devra tester si les deux ont la même équipe. Dans l'affirmative, pet sera bien modifier et il faudra également modifier l'attribut owner de l'animal.

3. **Redéfinissez la méthode `hit()`** pour le chasseur. Lorsqu'un chasseur attaque, il ajoute à sa force celle de son animal de compagnie.

Attention, vous ne pouvez pas juste faire someone.get_hit(self.strength + self.pet.strength) car dans le cas où le chasseur n'a pas d'animal (donc pet = None), self.pet.strength n'existe pas (None n'a pas d'attribut strength).
Vous devez donc penser à vérifier que pet existe avant d'accéder à sa force !

4. **Redéfinissez la méthode `__str__()`** pour que la description d'un chasseur corresponde à : **Hunter** <description du personnage> **with** <nom de l'animal>.
   Si le chasseur n'a pas d'animal de compagnie, n'ajoutez pas le « with … ».

Le principe est le même que pour Animal.

« Petit » test

1. Afin de pouvoir tester ce qui a été fait jusqu'ici, **ajoutez une méthode `show_details()`** dans la classe `Team` qui affiche tous les détails de l'équipe en respectant ce pattern :

```
***Equipe <nom> ( <nb victoires> / <nb match joué> )***
Joueurs actifs :
- <description des joueurs actifs>
Joueurs inactifs :
- < description des joueurs inactifs>
```

Par exemple, prenons une équipe « Super Team » qui a joué 10 parties et gagnée 8 d'entre elles. L'équipe a un animal nommé « Rantanplan » en réserve (avec 10 de vie et une force de 1). Elle a aussi un mage nommé « Nostradamus » avec 12 de vie, 2 de force, 2 d'armure et 6 de mana. Voici comment devrait s'afficher

```
***Equipe Super Team ( 8/10 )***
Joueurs actifs :
- Wizard Nostradamus (10-2-2)[6]
Joueurs inactifs :
- Animal Rantanplan (10-1-0)
```

De plus, les joueurs actifs/inactifs sont stockés dans des listes. Il vous suffit donc de boucler sur ces listes et à chaque tour de boucle, afficher le descriptif du personnage.
(Le descriptif correspond à celui prévu par __str__())

2. **Dans le bloc main :**
    a. **Créez une équipe** appelée « Les Profs »
    b. **Créez un guerrier** appartenant à cette équipe et appelé « Andhrien ». Il a 12 de vie, 3 de force et 3 d'armure.
    c. **Créez un animal** appartenant à cette équipe et appelé « Pixel ». Il a 10 de vie et 6 de force.
    d. **Créez un chasseur** appartenant à cette équipe et appelé « Valendi ». Il a 10 de vie, 2 de force et 3 d'armure.
    e. **Créez un animal** appartenant à cette équipe et appelé « Pong ». Il a 8 de vie et 4 de force.
    f. **Ajoutez Pong** comme étant l'animal de compagnie de Valendi.
    g. **Créez un mage** appartenant à cette équipe et appelé « Hans ». Il a 10 de vie, 3 de force et 2 d'armure. Et la valeur de mana prévue par défaut.
    h. **Créez un animal** appartenant à cette équipe et appelé « Tesla ». Il a 6 de vie et 4 de force.
    i. **Affichez les détails** de l'équipe « Les Profs »

Après exécution, vous devriez avoir ceci :

```
***Equipe Les Profs ( 0 / 0 )***

Joueurs actifs :

Joueurs inactifs :
- Warrior Andhrien (12-3-3)[0]
- Animal Pixel (10-8-0)
- Hunter Valendi (10-2-3) with Pong
- Animal Pong (8-4-0) owned by Valendi
- Wizard Hans (10-3-2)[5]
- Animal Tesla (6-4-0)
```

3. Toujours dans le bloc main, à la suite, **changez le statut de tous les personnages** présents puis réaffichez les détails de l'équipe.

reserve_character est une liste, vous pouvez donc être tenté de boucler dessus à l'aide d'un for. SAUF QUE la méthode change_status() que vous allez utiliser modifie cette liste pendant que le for boucle dessus !
Vous pouvez donc le faire un à un.

Après exécution, vous devriez avoir ceci si tout a été programmé correctement :

```
***Equipe Les Profs ( 0 / 0 )***

Joueurs actifs :

Joueurs inactifs :
- Warrior Andhrien (12-3-3)[0]
- Animal Pixel (10-8-0)
- Hunter Valendi (10-2-3) with Pong
- Animal Pong (8-4-0) owned by Valendi
- Wizard Hans (10-3-2)[5]
- Animal Tesla (6-4-0)

***Equipe Les Profs ( 0 / 0 )***

Joueurs actifs :
- Warrior Andhrien (12-3-3)[0]
- Animal Pixel (10-8-0)
- Hunter Valendi (10-2-3) with Pong
- Animal Pong (8-4-0) owned by Valendi
- Wizard Hans (10-3-2)[5]

Joueurs inactifs :
- Animal Tesla (6-4-0)
```

Déroulement d'un combat

Pour pouvoir lancer un combat, il manque encore certaines méthodes qu'on vous donne[2].

1. **Dans la classe `Fight`, ajoutez la méthode `start()` suivante :**

```
def start(self):
        # vérification équipes
        if len(self.participants[0].active_characters)==0:
            print("L'équipe", self.participants[0].name, "n'a aucun personnage actif")
            return None
        if len(self.participants[1].active_characters)==0:
            print("L'équipe", self.participants[1].name, "n'a aucun personnage actif")
            return None

        # la partie peut démarrer
        self.participants[0].show_details()
        self.participants[1].show_details()

        print("--- C'est partit ! ---")
        turn = 0
        current_team = random.choice((0,1))

        while len(self.participants[current_team].active_characters)>0 :
            indice = (turn//2) % len(self.participants[current_team].active_characters)
            current_player = self.participants[current_team].active_characters[indice]
            current_player.play_turn(self.participants[(current_team+1)%2].active_characters)

            turn += 1
            current_team = (current_team+1)%2

        # il y a un gagnant
        if len(self.participants[0].active_characters)==0 :
            self._winner = self.participants[1]
        else :
            self._winner = self.participants[0]

        print("Victoire de l'équipe", self._winner.name)
        self._winner.nb_victories +=1
        return self._winner
```

2. Dans la classe `Character`, ajoutez une méthode `play_turn()`. Elle prend la liste des joueurs adverses actifs en paramètre et en tape un au hasard. Comme ceci :

```
def play_turn(self, oponents):
        victim = random.choice(oponents)
        print(self.name, "attaque", victim.name)
        self.hit(victim)
```

Vous avez maintenant le strict minimum pour jouer mais cette méthode `play_turn` générale ne prend pas en compte les aptitudes particulières de chaque personnage. **Pour aller plus loin, redéfinissez `play_turn` pour un mage et un guerrier**.

Notez que si l'un d'entre nous a fait une faute de frappe dans ses attributs, cela pourrait poser problème… Si vous avez une erreur du type « un attribut n'existe pas », vérifiez que leur noms concordent.

[2] Pour faciliter le copier-coller, elles se trouvent sur moodle

Test complet

1. Dans votre `main` gardez votre équipe Les Profs et **créez une seconde équipe** au choix de la même manière.

Pour avoir par exemple :

```
***Equipe Les Profs ( 0 / 0 )***

Joueurs actifs :
- Warrior Andhrien (12-3-3)[0]
- Animal Pixel (10-8-0)
- Hunter Valendi (10-2-3) with Pong
- Animal Pong (8-4-0) owned by Valendi
- Wizard Hans (10-3-2)[5]

Joueurs inactifs :
- Animal Tesla (6-4-0)

***Equipe étudiants ( 0 / 0 )***

Joueurs actifs :
- Warrior Toto (12-3-3)[0]
- Animal Toutou (10-8-0)
- Warrior Bob (10-2-3)[0]
- Animal Chouchou (8-4-0)
- Wizard Nostradamus (10-3-2)[5]

Joueurs inactifs :
```

Le code correspondant à cette seconde équipe est le suivant :

```
team_stu = Team("étudiants")
c1 = Warrior("Toto",team_stu, life=12, strength=3, armor=3)
c2 = Animal("Toutou", team_stu, life=10, strength=8)
c3 = Warrior("Bob", team_stu, life=10, strength=2, armor=3)
c4 = Animal("Chouchou", team_stu, life=8, strength=4)
c5 = Wizard("Nostradamus", team_stu, life=10, strength=3, armor=2)
c1.change_status()
c2.change_status()
c3.change_status()
c4.change_status()
c5.change_status()
```

2. **Créez un combat** entre les deux équipes **et démarrez-le.**

3. **Affichez les détails des deux équipes** pour voir leur état à la fin du combat

Voilà un exemple de ce que ça donne chez moi si on prend les deux équipes proposées :

```
***Equipe Les Profs ( 0 / 1 )***

Joueurs actifs :
- Warrior Andhrien (12-3-3)[0]
- Animal Pixel (10-8-0)
- Hunter Valendi (10-2-3) with Pong
- Animal Pong (8-4-0) owned by Valendi
- Wizard Hans (10-3-2)[5]

Joueurs inactifs :
- Animal Tesla (6-4-0)

***Equipe étudiants ( 0 / 1 )***

Joueurs actifs :
- Warrior Toto (12-3-3)[0]
- Animal Toutou (10-8-0)
- Warrior Bob (10-2-3)[0]
- Animal Chouchou (8-4-0)
- Wizard Nostradamus (10-3-2)[5]

Joueurs inactifs :
--- C'est partit ! ---
Andhrien attaque Nostradamus
Toto attaque Pixel
Pixel attaque Chouchou
Toutou attaque Pong
```

…

```
Bob attaque Hans
Valendi attaque Bob
Victoire de l'équipe Les Profs

***Equipe Les Profs ( 1 / 1 )***

Joueurs actifs :
- Warrior Andhrien (7-3-3)[1]
- Hunter Valendi (5-2-3) with Pong
- Wizard Hans (3-3-2)[5]

Joueurs inactifs :
- Animal Tesla (6-4-0)
- Animal Pong (0-4-0) owned by Valendi
- Animal Pixel (-4-8-0)

***Equipe étudiants ( 0 / 1 )***

Joueurs actifs :

Joueurs inactifs :
- Animal Chouchou (0-4-0)
- Wizard Nostradamus (0-3-2)[5]
- Animal Toutou (-2-8-0)
- Warrior Toto (-1-3-3)[3]
- Warrior Bob (-2-2-3)[4]
```