TP5\_Travail Pratique 5

Jeu de rôle

Table des matières

[1 Mise en situation 2](#_Toc150432702)

[1.1 Informations générales 2](#_Toc150432703)

[1.2 Personnages 2](#_Toc150432704)

[1.3 Items 2](#_Toc150432705)

[1.4 Joueur et jeu 3](#_Toc150432706)

[2 Plan d’action général 4](#_Toc150432707)

[3 Modélisation des items 5](#_Toc150432708)

[3.1 Diagramme UML 5](#_Toc150432709)

[3.2 Détails 5](#_Toc150432710)

[3.3 À faire 6](#_Toc150432711)

[4 Modélisation des Personnages 7](#_Toc150432712)

[4.1 Diagramme UML 7](#_Toc150432713)

[4.2 Détails 8](#_Toc150432714)

[4.2.1 Hiérarchie d’héritage 8](#_Toc150432715)

[4.2.2 Classe Personnage 8](#_Toc150432716)

[4.2.3 Interface Attaquant 9](#_Toc150432717)

[4.3 À faire 9](#_Toc150432718)

[4.3.1 Héritage 9](#_Toc150432719)

[4.3.2 Interface Attaquant 10](#_Toc150432720)

[5 Joueur et Application Jeu 12](#_Toc150432721)

[5.1 À Faire 12](#_Toc150432722)

[6 Remise 13](#_Toc150432723)

**Consignes**

|  |
| --- |
| Toutes documentation permise.  Le travail est individuel.  Le code DOIT utiliser les notions vues en classe. |

# Mise en situation

## Informations générales

Vous devez modéliser un jeu de rôle.

Dans le **jeu**, le **joueur** a un **avatar**, soit un **personnage** avec des caractéristiques et qui peut engager des combats avec des **ennemis** et interagir avec discuter avec d’autres **personnages** pour faire avancer les quêtes.

Certains personnages peuvent utiliser des **items** comme des **armes**, des **armures**, des **trousses de santé** etc.

Certains personnages peuvent aussi utiliser des **capacités** magiques pour infliger des dommages à leurs adversaires.

Une partie du projet est déjà commencée, vous devez compléter principalement les parties qui traitent **d’héritage**, de **classes abstraites** et **d’interfaces**, puis mettre en place un peu de la **mécanique de jeu** pour tester le tout.

## Personnages

Les personnages sont divisés en 3 catégories :

* Les types d’avatars que peuvent avoir le joueur
* Les types d’ennemis
* Les types de personnages qui ne sont pas des ennemis (appelés personnages observateurs dans le modèle).

Chaque catégorie de personnage est sous-divisé en famille de personnage qui ont chacun leurs caractéristiques. Par exemple, un avatar **Chevalier** peut utiliser une arme et porter une armure, un ennemi **Sorcière** n’a pas d’arme ni d’armure mais utilise une capacité magique etc.

De plus, on désire identifier les catégories de personnages qui peuvent s’engager dans un combat, parce que ceux-ci auront une mécanique de combat spécifique. On identifie ces catégories en tant qu’**attaquants**. Les avatars et ennemis appartiennent aux catégories d’attaquants alors que les personnages observateurs ne sont pas attaquants.

Tous ces éléments du modèle sont regroupés dans un **package personnages**.

## Items

Les items de jeu sont divisés en familles d’items (Armes, Armures, etc.).

Les items du jeu peuvent être assignés à certains personnages. Certains de ces items sont équipé par les personnages (comme les armes et armures) alors que d’autres peuvent être simplement utilisés, comme les trousses de santé.

Notez que les capacités (magiques) seront considérées comme un item, parce qu’elles accomplissent le même rôle qu’une arme.

De plus, on désire identifier certains items comme **améliorable**. Si un item peut être amélioré, alors des caractéristiques augmentent à chaque niveau supplémentaire.

Tous ces éléments du modèle sont regroupés dans un **package items**.

## Joueur et jeu

Le **joueur** est une entité en soi et a des caractéristiques comme un alias, et il a un avatar, le personnage qu’il incarne dans le jeu et qui interagit avec les autres personnages.

Pour cette phase du développement le jeu est simplement mis en place dans une classe **AppJeu** qui inclus la méthode **main**.

**Note** : dans une version plus complète, le Jeu aurait sa propre classe, avec des caractéristiques comme un nom, des informations de connexion etc.

Le joueur et le programme principal **AppJeu** sont regroupés dans un **package jeu**.

# Plan d’action général

Une partie du modèle est déjà construit pour vous.

Toutes les classes sont déjà codées et présentes, aucune classe supplémentaire n’est requise.

Vous devrez cependant modifier certaines de ces classes pour mettre en place la hiérarchie d’héritage, d’abstraction ou d’interfaces.

Plusieurs champs et certaines méthodes sont aussi déjà codées. Vous devrez potentiellement ajouter certains champs et méthodes aux différentes classes.

De plus, vous devrez mettre en place suffisamment de mécanique de jeu pour tester votre modèle.

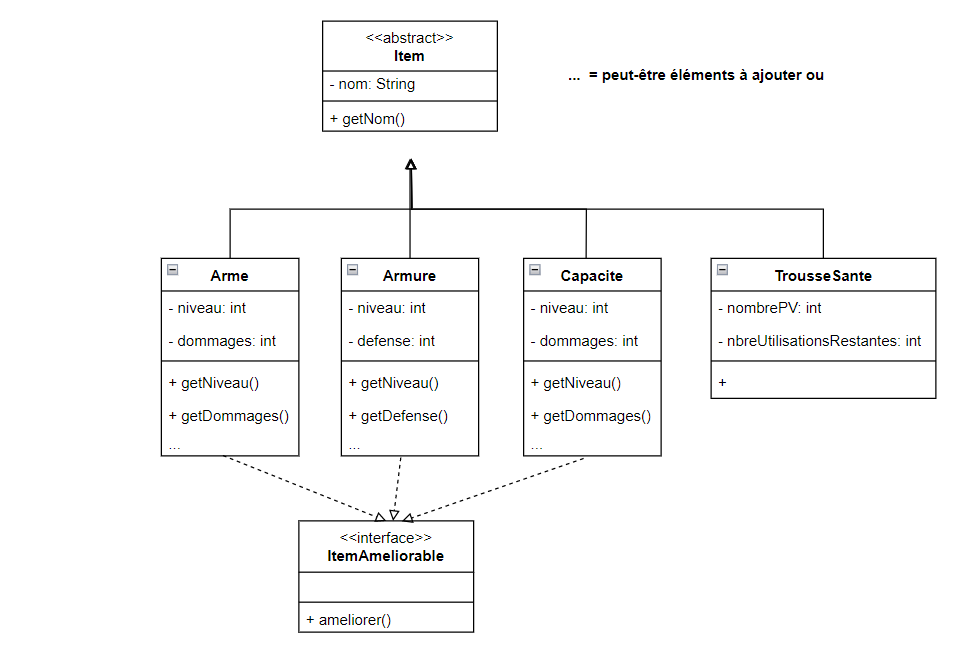
On approche le problème en sections qui correspondent aux packages déjà en place, dans l’ordre suivant :

1. Items -> structure un peu plus simple
2. Personnages -> structure plus complexe
3. Joueur et jeu -> mise en place des mécaniques de jeu

Pour chaque section, vous disposez d’un diagramme UML des éléments déjà présents et vous devez suivre les consignes fournies.

# Modélisation des items

## Diagramme UML



## Détails

Les membres qui sont communs à toutes les familles d’items sont regroupés dans la classe abstraite **Item**, qui comprend le champ nom et sa méthode d’accès.

Chaque famille d’items (Arme, Armure etc.) a déjà certains champs et leurs méthodes d’accès. Les caractéristiques utilisées sont :

* **dommages** : les points de dommages donnés lors d’une attaque avec cet item
* **defense** : les points de défense de cet item, ceux-ci réduisent les dommages reçus
* **niveau** : le niveau d’un item, pour ceux qui peuvent être améliorés
* **nombrePV** : le nombre de points de vie rendus à un personnage pou une utilisation de l’item
* **nbreUtilisationsRestantes** : le nombre de fois qu’on peut encore utiliser cet item avant qu’il soit vide

Pour les items qui peuvent être améliorés, on veut s’assurer qu’une méthode **améliorer()** est toujours mise en place, on aura donc une interface **ItemAmeliorable** qui modélise ceci et qui est appliquée aux items qui peuvent être améliorés.

## À faire

1. Modifiez la classe **Item** pour qu’elle soit une classe abstraite : on ne créera jamais un objet de cette classe, ce sera toujours un objet d’une des familles d’item.
2. Modifiez les classes **Arme**, **Armure**, **Capacite** et **TrousseSante** pour qu’elles héritent de la classe **Item**.
3. Pour les 4 familles d’items, ajoutez un constructeur qui prend un paramètre pour chaque champ de la classe (incluant les champs hérités) et qui les initialise à partir des paramètres.
4. Modifiez la classe **ItemAmeliorable** pour qu’elle soit une interface plutôt qu’une classe et décommentez la déclaration de méthode **ameliorer()**.
5. Modifiez les classes **Arme**, **Armure** et **Capacite** pour qu’elles implémentent l’interface **ItemAmeliorable**. La famille d’items **TrousseSante** n’a pas de niveau et n’est pas améliorable, elle n’implémente pas l’interface.
6. L’implémentation de l’interface **ItemAmeliorable** vous force maintenant à ajouter une méthode **ameliorer()** aux classes améliorables. Ajoutez cette méthode aux bons endroits en suivant les consignes suivantes :

* Pour ces méthodes, aucun paramètre requis et pas de valeur de retour.
* Améliorer une **Arme** augmente son niveau de 1 et ses dommages sont augmentés de 5 à chaque amélioration
* Améliorer une **Armure** augmente son niveau de 1 et sa défense est augmentée de 15 à chaque amélioration
* Améliorer une **Capacite** augmente son niveau de 1 et les dommages sont toujours ajustés selon la formule **dommages = 0.5 \* (niveau)2**.

1. Pour la **TrousseSante**, ajoutez une méthode **utiliser()** qui respecte les consignes suivantes :

* Pas de paramètre
* Valeur de retour qui représente le nombre de points de vie fourni par une utilisation
* Utilisable seulement s’il reste des utilisations, sinon, retourne 0
* Réduit le nombre d’utilisation de 1

1. Vérifiez tous les aspects habituels de votre code, méthodes d’accès et de non-accès, standards de nomenclature, meilleures pratiques etc.

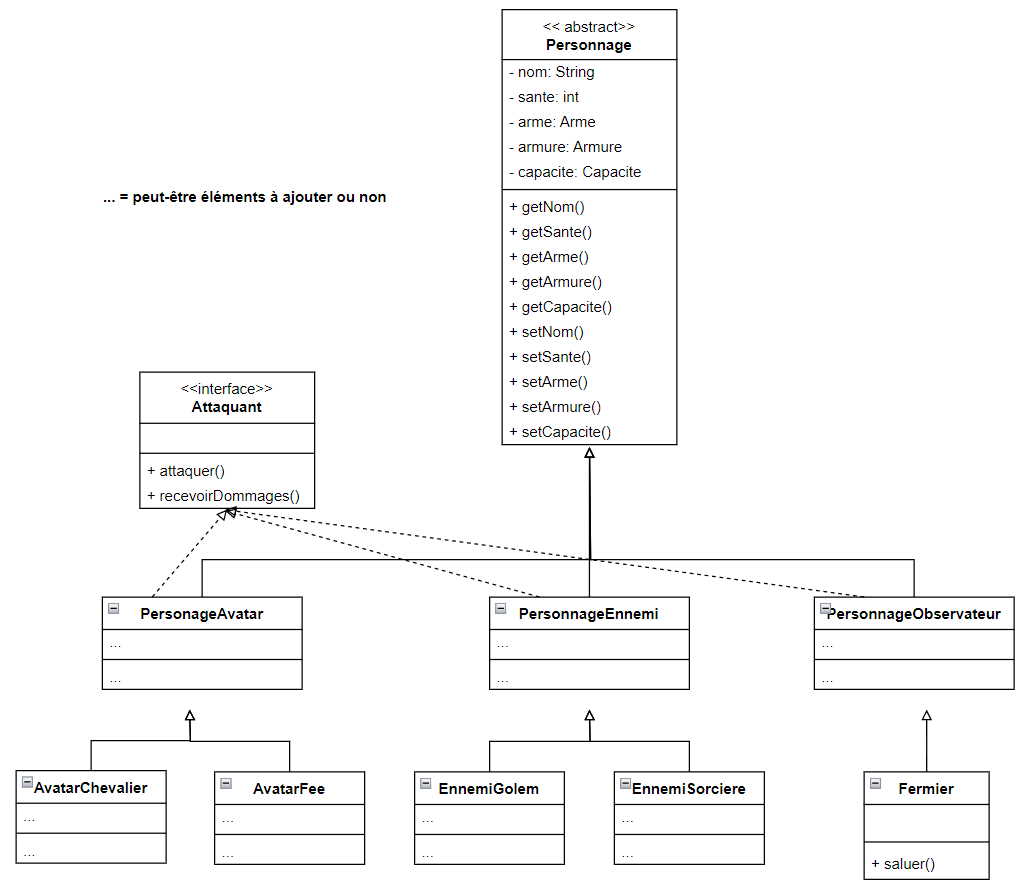
Vous désirez tester votre travail? Bonne idée, avant qu’il n’y ait trop de code à déboguer!

Vous pouvez créer des objets dans la méthode main de **AppJeu** et tester si les constructeurs fonctionnent, si les méthodes d’accès fonctionnent, si les méthodes **améliorer()** ou **utiliser()** fonctionnent et affectent les champs correctement.

Vous désirez automatiser vos tests? JUnit vous permet de faire ceci de façon automatique et réutilisable. PAS OBLIGATOIRE pour ce TP, c’est votre choix.

# Modélisation des Personnages

## Diagramme UML



## Détails

Pour la section des personnages on a 2 niveaux d’héritage et une interface.

### Hiérarchie d’héritage

La classe **Personnage**, une classe abstraite, comprend tout ce qui est commun à toutes les catégories de personnages.

Les sous-classes de **Personnage** (**PersonnageAvatar**, **PersonnageEnnemi**, **PersonnageObservateur**), sont des catégories de personnages qui ont des points communs ensemble. Une de leur utilité sera d’implémenter l’interface **Attaquant** aux personnages appropriés.

Les familles de personnages (**AvatarChevalier**, **EnnemiSorciere**, etc.) sont des sous-classes de leur classe de catégorie respective. Ex : **Fermier** est une sous-classe de **PersonnageObservateur**.

Note : Les familles de personnages héritent donc de la classe Personnage ET de la classe de catégorie de personnage qui lui est parente. Ex : **AvatarChevalier** hérite de **Personnage** ET **PersonnageAvatar**.

### Classe Personnage

Observez les champs de la classe **Personnage** et notez qu’on a choisi d’y mettre certains champs qui ne sont pas utilisés par tous les personnages. Ex : la Fee ne porte pas d’armure, le Fermier n’a pas d’arme etc.

Ce choix a été fait parce que les champs sont assez communs qu’il est plus pratique de ne pas les utiliser à certains endroits que de les réécrire dans plusieurs sous-classes. C’est un choix de conception, un autre choix aurait aussi pu être valide.

Dans le code observez le constructeur de la classe personnage, qui a déjà été codé pour vous.

Décommentez les lignes en commentaires dans le constructeur pour les ré-activer.

Notez que le constructeur accepte nom et sante en paramètre, mais que **Arme**, **Armure** et **Capacite** sont initialisés avec des valeurs de départ qui indiquent l’absence de cet item.

Notez finalement la méthode **estVivant()**, déjà codée pour vous, qui sera utilisée dans la mécanique d’un combat.

### Interface Attaquant

Certains personnages peuvent s’engager dans un combat avec d’autres personnages; ce sont des personnages attaquants. Dans notre système, ce sera un avatar qui attaquera un ennemi et vice-versa, un combat jusqu’à la victoire!

On veut que tous les personnages attaquants mettent obligatoirement en place les 2 méthodes standard suivantes :

**attaquer()**

Reçoit un paramètre **personnage** de type **Personnage** et retourne un **int**, soit le nombre de dommages causés par l’attaque. C’est dans cette méthode qu’on programmera la mécanique d’une attaque.

**recevoirDommages()**

Reçoit un paramètre **dommages** de type **int** et ne retourne rien. Cette méthode réduira les points de vie d’un personnage en fonction des dommages reçus.

Tel que démontré sur le diagramme UML, les catégories de personnages Avatar et Ennemi seront de type attaquant.

## À faire

### Héritage

1. Modifiez la classe **Personnage** pour qu’elle soit une classe abstraite : on ne créera jamais un objet de cette classe, ce sera toujours un objet d’une des familles d’item.
2. Les classes **PersonnageAvatar**, **PersonnageEnnemi**, et **PersonnageObservateur** sont es classes normales, elles ne sont ni abstraites ni des interfaces, pas de modifications requises de ce point de vue.
3. Modifiez les classes **PersonnageAvatar**, **PersonnageEnnemi**, et **PersonnageObservateur** pour qu’elles héritent de la classe **Personnage**.
4. Ajoutez un constructeur aux classes **PersonnageAvatar**, **PersonnageEnnemi**, et **PersonnageObservateur** qui accepte des paramètres **nom** et **sante** et qui initialisent ces paramètres en appelant le constructeur parent (avec super).
5. Modifiez les classes **AvatarChevalier** et **AvatarFee** pour qu’elles héritent de **PersonnageAvatar**.
6. Modifiez les classes **EnnemiGolem** et **EnnemiSorciere** pour qu’elles héritent de **PersonnageEnnemi**.
7. Modifiez la classe **Fermier** pour qu’elle hérite de **PersonnageObservateur**.

### Interface Attaquant

1. Modifiez la classe **Attaquant** pour qu’elle soit une interface plutôt qu’une classe.
2. Ajoutez les méthodes **attaquer()** et **recevoirDommages()** à la classe **Attaquant** en respectant les signatures décrites plus haut.
3. Modifiez les classes **PersonnageAvatar** et **PersonnageEnnemi** pour qu’elles implémentent l’interface **Attaquant**.
4. Les classes **PersonnageAvatar** et **PersonnageEnnemi** sont maintenant obligées d’avoir des méthodes **attaquer()** et **recevoirDommages()**. Ajoutez ces méthodes en respectant les consignes suivantes :

* **attaquer()**
  1. Retourne les dommages provoqués selon un formule.
  2. Les dommages provoqués sont :
     1. (Les dommages de l’arme du personnage actuel) – (les points de défense de l’armure du personnage reçu en paramètre).

**Ex** : **chevalier1.attaquer(golem1)** retourne les dommages de l’arme du chevalier moins la défense du golem.

* **recevoirDommages()**
  1. Reçoit une valeur des dommages reçus et réduis la santé du personnage de ce montant. La santé ne peut pas être négative, au minimum, 0.

1. Pour les 5 familles de personnages créez un constructeur qui respecte les consignes ci-dessous :

* **AvatarChevalier**
  + Reçoit un **nom** en paramètre et initialise ce champ à partir du paramètre
  + A une santé de **500**
  + A une arme nommée **Épée**, de niveau **1** qui fait **100** de dommages
  + A une armure nommée **Plastron**, de niveau **1**, qui a une défense de **15**
  + Le Chevalier n’a pas de capacité magique, ce champ n’est pas modifié (valeur par défaut)
* **AvatarFee**
  + Reçoit un **nom** en paramètre et initialise ce champ à partir du paramètre
  + A une santé de **200**
  + A une capacite nommée **Enchantement**, de niveau **1** qui fait **40** de dommages
  + La fée n’a ni arme ni armure, ces champs ne sont pas modifiés (valeur par défaut)
* **EnnemiGolem**
  + Reçoit un **nom** en paramètre et initialise ce champ à partir du paramètre
  + A une santé de **800**
  + A une arme nommée **Poings**, de niveau **1** qui fait **50** de dommages
  + A une armure nommée **Carapace**, de niveau **1**, qui a une défende de **40**
  + Le golem n’a pas de capacité magique, ce champ n’est pas modifié (valeur par défaut)
* **EnnemiSocrciere**
  + Reçoit un **nom** en paramètre et initialise ce champ à partir du paramètre
  + A une santé de **350**
  + A une capacite nommée **Mauvais sort**, de niveau **1** qui fait **100** de dommages
  + La sorcière n’a ni arme ni armure, ces champs ne sont pas modifiés (valeur par défaut)
* **Fermier**
  + Reçoit un **nom** en paramètre et initialise ce champ à partir du paramètre
  + A une santé de **0**
  + Le fermier n’a ni arme, ni armure, ni capacité, ces valeurs ne sont pas modifiées (valeur par défaut)

1. La fée et la sorcière n’attaquent pas avec une arme et leur capacité magique passe à travers les défenses. Le calcul de dommages de la méthode héritée **attaquer()** convient au chevalier et au golem, mais pas aux personnages magiques. Redéfinissez (override) la méthode **attaquer()** pour les classes **Fee** et **Sorciere** pour qu’elles utilisent les dommages de leur capacité au lieu d’une arme, et qu’elles ne tiennent pas compte de la défense de l’opposant.

Ici aussi et si ce n’est pas déjà fait, testez le travail fait jusqu’à maintenant, en utilisant la méthode de votre choix.

# Joueur et Application Jeu

La classe **joueur** est déjà faite pour vous, vous pouvez en consulter le code qui est assez simple.

Le constructeur accepte un alias (nom du joueur) et un **PersonnageAvatar** en paramètres et initialise ces champs à partir des paramètres.

Des méthodes d’accès et de mutation sont aussi présentes.

## À Faire

Pour tester les modèles, ont créé des objets et on simule des actions des personnages, incluant une bataille.

1. Créez un objet **avatar1** de type AvatarChevalier.
2. Créez un objet **avatar2** de type AvatarFee.
3. Créez un objet **joueur1**, de type Joueur, avec l’avatar du Chevalier (avatar1)
4. Créez un objet **ennemi1**, de type EnnemiGolem
5. Créez un objet **ennemi2**, de type EnnemiSorciere
6. Créez un objet **figurant1** de type fermier
7. Créez un objet **ts1** de type TrousseSante, avec 400 PV et 3 utilisations
8. Simulez un combat entre l’avatar du joueur et l’ennemi1. Un algorithme possible pour un combat est le suivant :

|  |
| --- |
| Afficher un message de début de combat, genre **Nom\_avatar engage le combat avec Nom\_ennemi**  Tant que avatar du joueur est vivant et que ennemi est vivant  Avatar du joueur attaque ennemi (retourne des dommages)  Ennemi reçoit les dommages de l’attaque  Affiche points de vie de avatar et ennemi  Si ennemi est vaincu (n’est plus vivant)  Affiche un message du genre **Vous êtes vaincu! fin de la partie**.  Sinon (ennemi est vivant)  Ennemi attaque avatar (retourne des dommages)  Avatar reçoit des dommages  Affiche points de vie de avatar et de ennemi  Si avatar est vaincu (n’est plus vivant)  Affiche un message du genre **Vous êtes vaincu! fin de la partie.** |

1. Qui gagne le combat?
2. Une fois ces tests concluants procédez aux étapes qui suivent.
   1. Entre la création des objets et le combat ajoutez le code pour :
      1. Utilisez la trousse de sante **ts1** et augmentez les PV de l’avatar du joueur du montant reçu de la trousse
      2. Améliorer l’arme de l’avatar du joueur
      3. Améliorer l’armure de ennemi1
      4. Faites dire Bonjour au fermier
   2. Exécutez à nouveau le programme. Gagnez-vous le combat?
3. Le TP doit être remis dans l’état où il est à cette étape. Après avoir procédé aux tests ci-dessous, assurez-vous de revenir à ces configurations d’avatar et d’ennemis.
4. Finalement, testez différentes combinaisons d’avatars et d’ennemis. Est-ce que la mécanique d’attaque fonctionne correctement? La fée et la sorcière devraient ignorer la défense de leur opposant…

# Remise

Remettez le répertoire de projet **C34\_TP5** et tout son contenu sous format compressé (**zip**), sur **Léa** dans la section **travaux** sous **Travail Pratique 5**.