# Projet dont vous êtes le Héros

## Introduction

Ce projet consiste à produire un jeu du type "vous êtes le héros". Ce style de jeu très populaire s'adapte à différents types de récit : aventure, romance, horreur ; et permet de plonger le.la joueur.euse dans une histoire dont l'issue sera déterminée par ses choix et possiblement des événements aléatoires (e.g., un choix à pile ou face). Les choix réalisés seront déterminants car ils impacteront l'issue du récit à un moment plus ou moins proche des décisions prises, assurant au joueur (à la joueuse) une fin heureuse ou terrible.

Les différents choix possibles méneront donc à différents parcours de jeu. Tous ces parcours mènent à une représentation sous forme d'un graphe orienté acyclic (acyclic pour éviter une partie infinie). Le graphe est composé de nœuds (chaque nœud représentant une situation) et d'arcs (chaque arc représentant une transisiton entre deux nœuds). Il existera différents types de nœud, des nœuds de *décision* pour lesquels le.la joueur.euse est amené.e à faire un choix, des nœuds *chance* où un évenement aléatoire ce produit, et des nœuds *terminaux* pour la fin du jeu (e.g., le.la joueur.euse termine vainqueur.euse d'une lutte épique couvert.e de lauriers).

Pour imerger le.la joueur.euse dans un univers riche et immersif, on travaillera également, une hierarchie sur les personnages possibles du jeu. Vous serez libre de choisir cet univers mais vous devrez respecter certaines contraintes. En effet, l'ensemble de vos classes devront présenter des relations d'héritage et de composition, et vous devrez utiliser des interfaces, des classes abstraites, et des enumérations.

**Par exemple** (vous devez donc proposer quelquechose d'autre), on pourrait imaginer un univers d'aventure fantastique dans lequel trois peuples (les orcs, les humains, et les elfes) se battent dans une guerre sans merci. Dans cet univers, le.la joueur.euse pourra incarner différent personnages (classe `Personnage`). Au fur et à mesure de l'histoire le.la joueur.euse fera des choix qui le.la meneront à devenir soit un.e soldat.e soit un.e mage (classe `Mage`). La classe `Soldat` est une classe abstraite dont dérive les classes `Lancier` ou `Archer`. Il y a ici des relations d'héritage entre les classes `Personnage`, `Mage`, `Soldat`, `Lancier`, et `Archer`. Les classes `Archer` et `Mage` implémentent une interface commune `AttaqueADistance`. Pour les énumérations, nous pouvons en utiliser une pour représenter les différents peuples (orcs, humains, elfes), ou encore une pour représenter les différents grade d'un soldat (caporal, sergent, capitaine, commandant). Enfin, pour la composition, nous pouvons utiliser le fait que chaque personnage dispose d'une ame et créer une autre hierarchie de classes à partir d'une classe mère `Arme`.

Ce travail est à réaliser **en binôme**. En plus de votre code, vous devrez rendre un *rapport* contenant :
- Une présentation de la structure de votre code, et une justification des choix concernant vos hierarchies de classes.
- Une présentation de vos extensions éventuelles.
- Une discussion sur ce qui a été le plus dur à implémenter pour vous et sur le niveau de difficulté du projet.

Votre projet devra également contenir un fichier *ReadMe* présentant succinctement votre projet et détaillant les instructions à réaliser pour le compiler, l'exécuter, puis l'utiliser.

→ Les détails de la soumission de votre projet seront donnés sur l'espace moodle du cours. Il y aura notamment un jalon à rendre à mi-parcours.

**Ce projet compte pour 40% de la note de l'UE. Il est donc souhaitable que la note corresponde au travail de votre groupe, et non aux conseils d'autres groupes, d'autres étudiants ou d'internet. Si vous utilisez des sources (articles de recherche, posts sur internet, etc...), vous devez mentionner vos sources dans le rapport.**

## Planning de travail

**À la fin du cours sur l'héritage et le polymorphisme :**

- Constituer des binômes.

Travail sur le package `representation`.

- Créer un *package* `representation` qui contiendra les classes représentant le graphe des états possibles de votre jeux.
- Dans le package `representation`, créer les *classes* `Node`, `InnerNode`, `ChanceNode`, `DecisionNode`, et `TerminalNode`.
  - Un `Node` représente un nœud du graphe de l'histoire. Il présente une situation au joueur (à la joueuse) à partir d'un champ `String description` et d'une méthode qui affiche cette description `public void display()`. Chaque instance de la classe `Node` aura un attribut `int id`, correspondant à un identifiant unique attribué à l'instance. Chaque `Node` devra également être combiné par composition avec les classes de votre package *univers* (voir prochain paragraphe). Un `Node` possède une méthode `public Node chooseNext()` permettant de désigner le prochain nœud à considérer.
  - Un `TerminalNode` est un nœud terminal du graphe (et donc la fin de l'histoire). La méthode `chooseNext()` devra donc renvoyer le noeud lui même.
  - Un `InnerNode` est un nœud non terminal du graphe. Pour simplifier nous suposerons dans un premier temps qu'un `InnerNode` peut mener à quatre autres nœuds renseignés en attributs.
  - Un `DecisionNode` est un nœud où le joueur doit prendre une décision. Ainsi la méthode sélectionnant le prochain noeud devra intéragir avec l'utilisateur, e.g., via la console. Vous pouvez regarder la documentation de la classe `Scanner` ici.
  - Un `ChanceNode` est un nœud où un évenement mêlant une part d'aléatoire intervient. Ainsi la méthode sélectionnant le prochain noeud devra simuler une loie aléatoire. Vous pouvez regarder la documentation de la classe `Random` ici.

  Quelles relations d'héritage instaurez-vous entre ces classes ; avec quelles redéfinitions ?

Travail sur le package `univers`.

- Créer un *package* `univers` qui contiendra les classes représentant l'univers dans lequel va évoluer le.la joueur.euse.
- Créer les premières classes de votre univers, notamment avec une classe "personnage de base".

Sur les deux packages.
- Travailler sur la *visibilité* de vos attributs et méthodes.
- Réaliser des *getters*, des *setters*, et des *constructeurs*. Faites attention au lieu entre héritage et constructeurs.
- Redéfinir les méthodes `toString` et `equals` de la classe `Object` dans vos différentes classes (documentation ici).
- Écrire une méthode `main` pour tester votre programme. Vous devriez pouvoir simuler une toute petite histoire.

À la fin du cours sur les interfaces et les classes abstraites :
- Déterminer si certaines de vos classes devraient être *abstraites*.
- Continuer à travailler sur le package `univers` pour rajouter des énumérations, des classes, abstraites, des interfaces, et de la composition. Vous devez obtenir au moins un petite dizaine de classes en comptant tout.
- Voici un point un peu plus avancé ! On souhaite maintenant rajouter des fonctionnalités à nos nœuds pour qu'ils puissent, avant l'affichage de la description de la situation, afficher une image ou jouer un son. Pour ce faire, au lieu de complexifier notre hierarchie, nous allons utiliser le design pattern *Decorator*.
  - Créer une interface `Event` qui sera implémentée par la classe `Node`. Elle demandera l'implémentation des méthodes :
    - `public void display();`
    - `public Displayable chooseNext();`.
  - Créer un décorateur abstrait implémentant `Event` et deux décorateurs concrets (`ImageNode` et `SoundNode`) selon le design pattern *Decorator*.
  - Pour jouer un son avec Java, chercher des informations sur les classe du package *javax.sound.sampled*, **puis** n'hésitez pas à demander des conseils à votre chargé.e de TD/TP ou au responsable d'UE.
  - Pour montrer une image avec Java, chercher des informations sur les classes des packages *java.awt* et *java.swing*, **puis** n'hésitez pas à demander des conseils à votre chargé.e de TD/TP ou au responsable d'UE.
- Différencier désormais une classe `Game` qui modélise la partie jouer par le joueur de la classe Main qui configurera la partie si besoin et lancera le jeu (avec une instance de la classe `Game`).

À la fin du cours sur Git :
- Placer votre projet sur *github* afin de pouvoir *versionner* votre code pour la suite du projet.

À la fin du cours sur le Javadoc et les tests unitaires :
- Commencer la *documentation Javadoc* de votre projet. Celle-ci devra être relativement complète au moment du rendu.
- Choisir deux classes de votre choix et créer deux classes de *tests* JUnit pour ces classes afin de réaliser des *tests unitaires*. Toutes les méthodes ne devront pas être testées mais vous devait écrire une variété de tests illustrant les possibilités de l'outil JUnit.
- Rajouter les *annotations* nécessaires à votre projet telles que les `@Override`.

À la fin du cours sur les collections et les maps :

— Dans la classe `InnerNode`, faites en sorte qu'il puisse y avoir un nombre non défini (et pas forcément quatre comme précédemment défini) de prochains noeuds possibles en utilisant une collection de nœuds comme attribut.
— Rajouter des collections ou des maps dans votre package `univers`.

À la fin du cours sur les entrées-sorties :
— Grace à la sérialisation proposer un système de sauvegarde de partie. Lors d'une partie le joueur aura la possibilité de sauvegarder la partie afin de reprendre plus tard.
— Créer un menu intéractif afin d'intéragir dans la console avec l'utlisateur.rice de votre programme dans la classe `Main`. Le joueur peut alors quitter le programme, lancer une nouvelle partie, ou reprendre une sauvegarde. Les sauvegardes devront êtres présentées de la plus récente à la plus ancienne.

À la fin du cours sur les exceptions :
— Gérer les différentes *exceptions* possibles de votre programme. Il faudra notamment gérer les exceptions générées par les flux d'entrées-sorties et les exceptions générés par l'utilisation d'arguments inapropriées (par exemple, un nombre de munitions négatif pour un archer).

Pour la suite : Si vous voulez aller plus loin :
— Réaliser une interface graphique digne des plus grands jeux vidéos.
— Gérer le fait qu'il puisse y avoir plusieur joueur.euse, chacun gérant plusieurs parties disjointes.