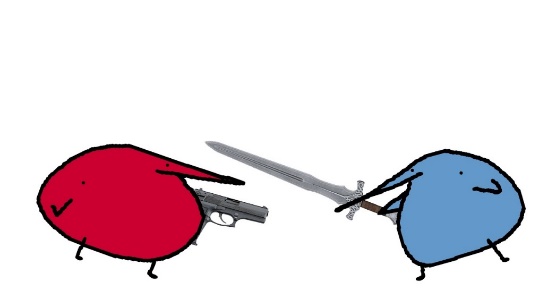
**Travail pratique 1 – Duel**

**Liste de contrôle**

****

**Nom : \_Hubert Laliberté\_\_\_\_\_\_\_\_\_**

**Nom : Antoine Chantal-Piché**

[*Berd*](https://www.youtube.com/channel/UCRei8TBpt4r0WPZ7MkiKmVg)

Vous devez compléter et remettre cette liste de contrôle avec votre travail.

*Lorsque vous ne pouvez pas cocher une case, vous devez l’expliquer avec un commentaire.*

**Exigences de fonctionnalités**

* Il est possible de :
  + Créer un guerrier;
  + Tout en respectant les contraintes énoncées au niveau de ses aptitudes.
* Il est possible de :
  + Créer un athlète;
  + Tout en respectant les contraintes énoncées au niveau de ses aptitudes.
* Il est possible de :
  + Créer un magicien;
  + Tout en respectant les contraintes énoncées au niveau de ses aptitudes.
* Il est possible de créer des capacités en spécifiant leurs caractéristiques.
* Un combattant peut posséder n’importe quelle capacité, même si elles ne sont pas les plus adaptées pour lui.
* Il est possible d’évaluer la puissance d’une capacité en fonction des aptitudes du combattant qui la met en œuvre et de la caractéristique de la capacité.
* Un combattant peut provoquer en duel un autre combattant, s’il utilise une capacité d’attaque.
* Un combattant provoqué en duel peut capituler et refuser de combattre.
* Un combattant provoqué en duel peut riposter avec une capacité d’attaque.
* Un combattant provoqué en duel peut riposter avec une capacité de défense.
* À la fin du duel, conformément aux règles énoncées :
  + Le gagnant reçoit ses récompenses : augmentation de ses aptitudes et gain d’une nouvelle capacité.
  + Le perdant est pénalisé : diminution de ses aptitudes, et s’il a perdu en ripostant, perte de points de vie.
* Un combattant peut aller se soigner à l’infirmerie, s’il a une capacité de soin.
* Un simulateur permet de démontrer clairement la création de capacités, de combattants, le déroulement de duels et l’application de soins à l’infirmerie.

**Exigences de qualité**

**Code**

* Les classes sont regroupées en « package » selon une organisation logique.
* Nommage : toutes les méthodes, les classes, les attributs, les variables et les constantes ont un nom significatif indiquant leur raison d’être ou leur contenu selon le cas.
* Il n’y a pas de conditions complexes (elles sont encapsulées).
* Il n’y aucun chiffre magique.
* Il n’y a aucune chaine de caractères magique.
* Les commentaires au travers du code sont réellement informatifs.
* Les conventions Java et les normes de programmation, telles que montrées, sont respectées.
* Toutes les classes sont petites (moins de 100 lignes).
* Toutes les méthodes sont les plus petites possibles (moins de 15 lignes).
* Les méthodes ont peu de paramètres.
* L’indentation du code est irréprochable.
* Dans l’ensemble, le code est clair, facile à lire et à comprendre.
* Dans l’ensemble, le code est simple (KISS : Keep it Simple, Stupid).
* Il n’y aucune erreur ni aucun avertissement lors de la compilation.  
  Note : si vous jugez qu’un avertissement n’a pas lieu d’être, consultez votre professeur et faites taire l’avertissement manuellement.
* Il n’y a pas de code mort (code en commentaire);
* Le simulateur s’exécute sans erreur.

**Exceptions**

* Des exceptions sont utilisées à la place de valeurs de retour codées ou de valeur de retour null.
* Des exceptions sont utilisées quand le flot normal du code est interrompu.
* Des exceptions « Runtime » sont utilisées pour gérer les erreurs de programmation.
* Les exceptions sont les plus spécifiques possibles.

**Tests**

* Toutes les classes sont testées.
* Toutes les méthodes de tests ont un nom significatif indiquant ce qui est testé, quitte à avoir un nom de méthode long.
* Chaque méthode de test ne teste qu’un seul comportement.
* Chaque méthode de test respecte l’organisation : Arrange/Act/Assert
* Des objets simulés (« mocks ») ont été utilisés dans les tests au besoin.
* Combien de tests sont exécutés ? \_\_\_\_\_\_85\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* Tous les tests passent avec succès.

**Conception objet**

* Encapsulation : toutes les propriétés d’une classe (exceptées les constantes) sont privées
* Le principe de conception suivant est respecté: Single Responsability Principle.
* Le principe de conception suivant est respecté: Loi de Demeter.
* Le principe de conception suivant est respecté: Query-Command Separation.
* Le principe de conception suivant est respecté: Tell, don’t ask.
* Polymorphisme : lorsque possible, le polymorphisme a été préféré à If/Else ou Switch/Case (éviter les if).
* Le mot clé « instanceof » n’est pas utilisé sauf pour tester les fabriques.
* Le principe de conception suivant est respecté: programmer avec des interfaces (ou des classes abstraites) pas avec leurs implémentations.
* Le principe de conception suivant est respecté: principe de séparation des interfaces.
* Le patron de conception suivant est utilisé : Strategy pattern.  
  Précisez: \_\_\_Fighter, warrior, wizard, athlete, skills\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* La pratique de conception suivante est utilisée : Simple Factory.

Précisez: \_\_\_\_\_\_\_\_\_fighterFactory, skillsFactory\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Si une nouvelle race de combattant est proposée dans le jeu, il est possible de développer cette nouvelle fonctionnalité dans le code sans modifier le code des classes existantes.  
  Si non : quelles classes seraient à modifier : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* Si une nouvelle capacité est proposée dans le jeu, il est possible de développer cette nouvelle fonctionnalité dans le code sans modifier le code des classes existantes.

Si non : quelles classes seraient à modifier : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_