Examen de *Design Patterns*
2019–2020

— Durée : 2h
— Type : papier
— Une unique feuille A4 recto-verso manuscrite est autorisée comme document.
— Toutes vos affaires (sacs, vestes, *etc.*) doivent être placées à l'avant de la salle.
— Aucun téléphone ne doit se trouver sur vous ou à proximité, même éteint.
— Les déplacements et les échanges ne sont pas autorisés.
— Aucune question ne peut être posée aux enseignants, posez des hypothèses en cas de doute.

Question de cours (3pts)

1. Quel *design pattern* définit une famille d'algorithmes encapsulés dans des objets, permettant ainsi de les interchanger durant l'exécution ? (0.5pt)
2. Quel *design pattern* fournit une interface unifiée, facilitant ainsi l'utilisation d'un ensemble de classes et d'interfaces d'un sous-système ? (0.5pt)
3. Citez deux exemples d'utilisation du *design pattern* Procuration. (1pt)
4. Expliquez le *design pattern* Observateur à travers un diagramme de séquence. (1pt)

Star Wars : L'Ascension des *Design Patterns* (17pts)

Pour la sortie au cinéma du nouveau *Star Wars*, nous proposons de développer un jeu vidéo dans cet univers. La figure 1 illustre une hiérarchie de classes représentant différents types de personnage (*e.g.*, humains, wookiees, droïdes). L'interface `Personnage` comporte 7 méthodes :

— `nom()` retourne le nom du personnage ;
— `vie()` retourne les points de vie restants du personnage ;
— `attaque()` retourne la force d'attaque du personnage ;
— `combattre(Personnage p)` permet de combattre un autre personnage (ce dernier subit des dégâts équivalents à la force d'attaque du personnage) ;
— `subir(int x)` inflige x points de dégâts au personnage ;
— `guérir(int x)` redonne x points de vie au personnage ;
— `estMort()` retourne vrai si le personnage n'a plus de points de vie, faux sinon.

Dans cet examen, vous pouvez directement utiliser ces types, mais pas les modifier (sources non fournies).

Les questions suivantes peuvent être répondues indépendamment.

1. Nous avons récupéré une classe `Yoda` (écrite par le vieux Ben) dont nous ne disposons pas non plus des sources, mais que nous souhaitons utiliser comme un personnage dans le jeu. Cette classe est illustrée à la figure 2 et comporte 4 méthodes :
   — `getYoda()` retourne une instance de Yoda ;
   — `force()` retourne la force d'attaque de Yoda ;
   — `esquive()` retourne vrai si Yoda esquive une attaque, `faux` sinon ;

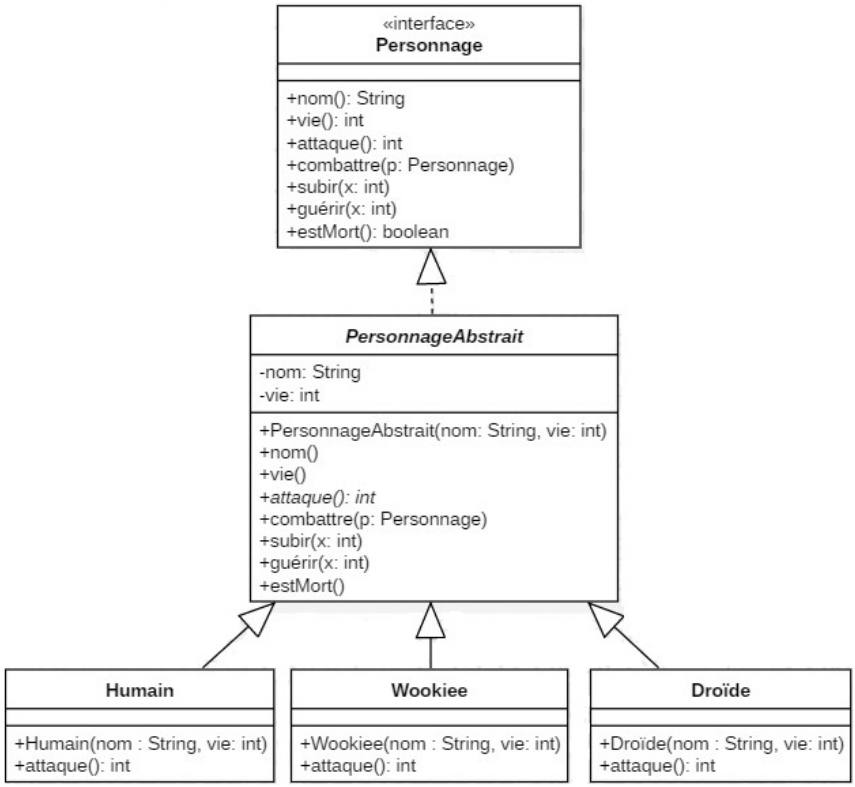

FIGURE 1 – Hiérarchie de classes *Star Wars*

— `réplique()` retourne une réplique culte de Yoda (*e.g.*, « Réussis ton examen ou ne le réussis pas, mais il n'y a pas de rattrapage. »).

(a) Quel *design pattern* pouvez-vous utiliser pour résoudre le problème ? (0.5pt)

(b) Modélisez le problème sous la forme d'un diagramme de classes UML. (1pt)

(c) Écrivez en Java une implémentation des classes non fournies du diagramme. (1.5pts)

(d) Écrivez un code client qui crée un personnage `Yoda` (de 900 points de vie) et le fait combattre contre un droïde. (1pt)

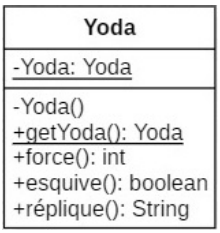

FIGURE 2 – Classe `Yoda`

2. Nous souhaitons pouvoir créer des *Jedi*, quel que soit le type de personnage. Selon son rang (*i.e.*, padawan, chevalier, maître), la force d'attaque d'un *Jedi* est multipliée par 2, 5 et 10, respectivement.

(a) Quel *design pattern* pouvez-vous utiliser pour résoudre le problème ? (0.5pt)

(b) Modélisez le problème sous la forme d'un diagramme de classes UML. (1pt)

(c) Écrivez en Java une implémentation des classes non fournies du diagramme. (2pts)

(d) Écrivez un code client qui crée un maître Jedi Wookiee et le fait combattre contre un droïde. (0.5pt)

3. Nous souhaitons pouvoir créer des troupes de combat. Une troupe est composée de différents personnages ou de sous-troupes. Une troupe doit non seulement pouvoir affronter une autre troupe, mais aussi un personnage seul (et *vice versa*). La vie et la force d'attaque d'une troupe correspondent respectivement à la somme des points de vie et des forces d'attaque de tous les personnages vivants qui la composent. Lorsqu'une troupe subit une attaque, les dégâts sont répartis de manière équitable entre ses membres encore en vie. Une troupe est morte lorsque tous ses membres sont morts.
   (a) Quel *design pattern* pouvez-vous utiliser pour résoudre le problème ? (0.5pt)
   (b) Modélisez le problème sous la forme d'un diagramme de classes UML. (1pt)
   (c) Écrivez en Java une implémentation des classes non fournies du diagramme. (3pts)
   (d) Écrivez un code client qui crée une troupe composée d'un humain, d'un droïde et d'une sous-troupe de 3 wookiees, et la fait combattre contre un droïde. (0.5pt)
4. Nous souhaitons pouvoir planifier des batailles entre personnages. Dans une bataille, un personnage peut en combattre un autre, subir des dégâts d'une attaque externe, et regagner des points de vie. Une fois toutes les actions de la bataille planifiées, celles-ci peuvent être exécutées au moment souhaité.
   (a) Quel *design pattern* pouvez-vous utiliser pour résoudre le problème ? (0.5pt)
   (b) Modélisez le problème sous la forme d'un diagramme de classes UML. (1.5pts)
   (c) Écrivez en Java une implémentation des classes non fournies du diagramme. (1.5pt)
   (d) Écrivez un code client qui planifie un combat entre un humain et un droïde, redonne 22 points de vie au personnage humain, puis lui fait subir 17 points de dégâts. Enfin, la bataille est simulée. (0.5pt)