Examen de *Design Patterns*
2021–2022

— Durée : 1h30
— Type : papier
— Aucun document autorisé.
— 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 (5pts)

1. Quel *design pattern* permet de définir une interdépendance de type un-à-plusieurs entre des objets de sorte que, lorsqu'un objet change d'état, tous les objets dépendants en soient notifiés afin de pouvoir réagir conformément ? (0.5pt)
2. Quel *design pattern* permet de définir une nouvelle opération pour une hiérarchie sans avoir à modifier ses classes ? (0.5pt)
3. Quel *design pattern* est illustré à la figure 1 ? (1pt)

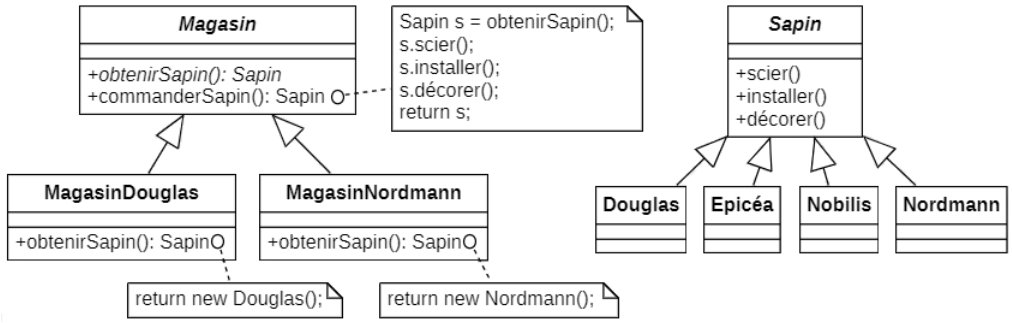

FIGURE 1 – Diagramme de classes

4. Quel *design pattern* est illustré à la figure 2 ? (1pt)

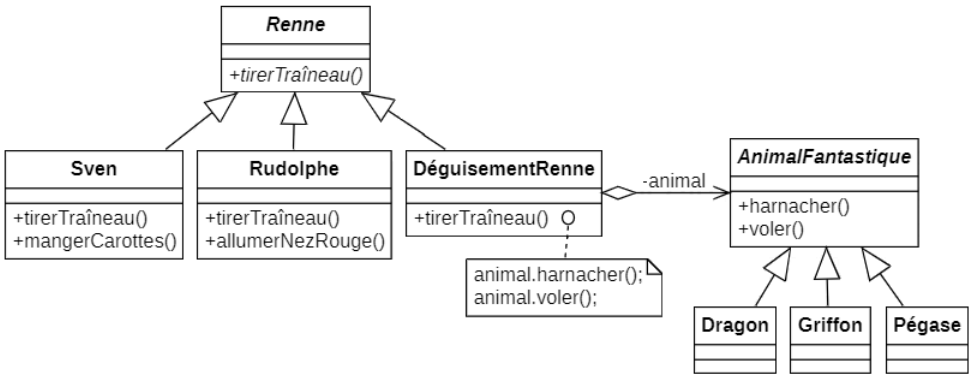

FIGURE 2 – Diagramme de classes

5. Nous souhaitons modéliser des macro-commandes, c'est-à-dire une suite de commandes. Modifiez le diagramme de classes du *design pattern* Commande afin de représenter des macro-commandes en utilisant le *design pattern* Composite. (2pts)

Design Patterns Resurrections (15pts)

***N.B.* : les phrases en italique correspondent à des dialogues du film *Matrix*, elles ne sont pas utiles pour la compréhension et la résolution des problèmes.**

Choisis la pilule bleue et tout s'arrête. Après, tu pourras faire de beaux rêves et penser ce que tu veux. Choisis la pilule rouge, tu restes en salle d'examen et tu réponds aux questions suivantes.

Pour la sortie au cinéma du nouveau *Matrix*, nous proposons de développer un jeu vidéo dans cet univers. La figure 3 illustre une hiérarchie de classes représentant différents types d'entité (*i.e.*, humains, agents, machines) présents dans le jeu.

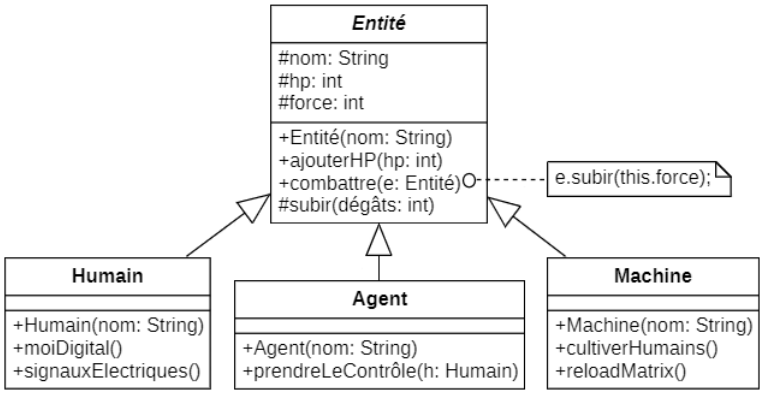

FIGURE 3 – Hiérarchie de classes *Matrix*

La Matrice est universelle. Elle est omniprésente. Elle est avec nous ici, en ce moment même. Tu la vois chaque fois que tu regardes par la fenêtre, ou lorsque tu écris sur ta copie d'examen.

1. Nous souhaitons créer une classe `Matrice` représentant la Matrice (*i.e.*, *le monde qu'on superpose au regard des humains pour les empêcher de voir la vérité*). En particulier, cette classe contient des méthodes pour ajouter et retirer dynamiquement des humains de la Matrice, et une méthode `simulationNeuroInteractive()` qui envoie des signaux électriques au cerveau des humains. Nous nécessitons qu'il n'existe qu'une seule instance de cette classe à tout moment dans le jeu, afin d'éviter des problèmes d'incohérence et d'écrasement de données. De plus, cette instance doit être facilement accessible aux autres classes du jeu.
   (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. (2pts)
   (c) Écrivez en Java une implémentation des classes du diagramme. (1.5pts)
   (d) Écrivez un code client qui crée 42 humains, les ajoute à la Matrice et exécute la simulation neuro-interactive. (1pt)

Ceci est notre structure, une sorte de programme de chargement, on peut y afficher ce qu'on veut : vêtements ou équipements, armes à feu, simulateur d'entraînement, tout ce qui nous est nécessaire.

2. La méthode `moiDigital()` de la classe `Humain` correspond à *l'image intérieure résiduelle* d'un humain, c'est-à-dire une projection mentale de son *moi digital*. De base, cette méthode affiche dans la console les caractéristiques par défaut de l'humain. Nous souhaitons pouvoir enrichir dynamiquement cette opération, mais sans modifier l'interface de la classe `Humain`.

Ainsi, il doit être possible d'afficher l'image résiduelle d'un humain avec divers vêtements (*e.g.*, un manteau long) et accessoires (*e.g.*, des lunettes noires), ou encore diverses armes (*e.g.*, un pistolet). Tous les éléments ajoutés seront listés dans la console en sus.

(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 du diagramme. (2.5pts)

(d) Écrivez un code client qui crée une humaine, nommée *Trinity*, dont l'image intérieure résiduelle est vêtue d'un manteau long, porte des lunettes noires et a deux pistolets. (0.5pt)

– *Je connais le Kung-Fu.*
– *Prouve-le moi.*

3. La méthode `subir(int)` de la classe `Entité` retire aux points de vie (*i.e.*, l'attribut `hp`) de l'entité, les points de dégât passés en paramètre. Pour se défendre, un humain peut apprendre les arts martiaux, comme le Kung-Fu, ce qui lui permettra de réduire les dégâts reçus lors d'un combat comme suit : d'abord de moitié, puis d'un quart, et enfin de rien (il subit alors tous les points de dégât infligés), puis de nouveau de moitié, d'un quart, et de rien, et ainsi de suite. D'autre part, s'il est élu de l'espèce humaine, il ne peut pas mourir, et donc tous les dégâts qu'il reçoit sont réduits à zéro. De plus, ses points de vie augmentent du nombre de points de dégât qu'il aurait dû recevoir. Ainsi, nous souhaitons pouvoir modifier dynamiquement le comportement de la méthode `subir(int)` en fonction des différentes compétences acquises par un humain au cours du jeu. Il doit également être possible d'ajouter dans le jeu de nouvelles compétences pour se défendre, sans avoir à modifier cette méthode.

   (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 du diagramme et modifiez, si nécessaire, les classes déjà fournies. (2.5pts)

   (d) Écrivez un code client qui crée un humain, nommé *Néo*, qui d'abord ne sait pas se battre, puis apprend le kung-fu, et enfin se révèle être l'élu de l'espèce humaine. À chacune de ces étapes, un agent, nommé *Smith*, le combat. (0.5pt)