Examen de *Design Patterns*
2018–2019

— 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 (4pts)

1. Pourquoi ajouter de nouveaux types de produit dans le cadre d'une fabrique abstraite est problématique ? (1pt)
2. Quels compromis sont faits lorsque les opérations de gestion des fils d'un composite sont déclarées dans l'interface `Composant` ? Ou uniquement dans la classe `Composite` ? (1pt)
3. Expliquez les différences entre un adaptateur de classe et un adaptateur d'objet. (1pt)
4. 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. (1pt)

Astérix : le secret des *design patterns* (16pts)

Pour la sortie au cinéma du nouvel Astérix, nous proposons de développer un jeu vidéo dans son univers. Le joueur incarne Astérix dont le but est de casser du Romain, combattre des pirates ou encore chasser des sangliers. Chaque fois qu'Astérix détruit un ennemi, le joueur gagne des points qui sont ajoutés à son score. Un extrait d'une implémentation du jeu est fourni en annexe.

Les questions suivantes peuvent être répondues indépendamment.

1. Nous souhaitons créer une classe `Asterix` représentant le personnage Astérix contrôlé par le joueur. En particulier, cette classe contient le score du joueur et une méthode `ajouterPoints(int)` qui prend en argument un nombre de points à ajouter au score du joueur. Nous nécessitons qu'il n'existe qu'une seule instance de cette classe à tout moment dans le jeu (pour éviter des incohérences). 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. (1.5pts)
   (c) Écrivez en Java une implémentation des classes du diagramme. (1.5pts)
   (d) Complétez la méthode `detruire()` de la classe `Ennemi` pour ajouter les points obtenus lors de la destruction de l'ennemi au score du joueur. (0.5pt)
2. Nous souhaitons ajouter divers équipements (*e.g.*, glaive, javelot, bouclier, casque...) aux Romains. Ainsi, il doit être possible de créer des Romains avec un javelot et un bouclier, avec un casque, avec deux glaives, *etc.* Chaque équipement ajouté augmente le nombre de points obtenus par le joueur lorsque Astérix bat le Romain. Attention, pour faire cela, l'interface de la classe `Romain` ne doit pas être changée.

(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. (1pt)

(d) Écrivez un code client qui crée un Romain avec un javelot, un bouclier et un casque. (0.5pt)

3. Astérix peut ramasser divers items (*e.g.*, potion magique, couronne de laurier, amphore...) dans le jeu. Ces items peuvent avoir un effet sur l'ajout des points au score du joueur. Par exemple, lorsque Astérix ramasse une potion magique, le nombre de points obtenus à la destruction d'un ennemi est multiplié par deux, puis par trois pour l'ennemi suivant, et ainsi de suite pendant dix secondes (retour à la normale à la fin du timing). Un autre exemple est au contraire d'annuler l'ajout des points (*i.e.*, les réduire à zéro) pour les trois prochains ennemis détruits lorsque Astérix ramasse une amphore. Nous souhaitons donc pouvoir modifier dynamiquement le comportement de la méthode `ajouterPoints(int)` de la classe `Asterix` en fonction du dernier item ramassé par Astérix. Il doit être possible d'ajouter de nouveaux types d'item dans le jeu 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. (1.5pts)

   (d) Ajoutez une méthode `ramasserItem(Item)` à la classe `Asterix` permettant de modifier le comportement de la méthode `ajouterPoints(int)` selon l'item ramassé. (0.5pt)

4. Après avoir bu de la potion magique, Astérix devient invincible, ce qui a pour effet de modifier le comportement des ennemis qui prennent alors la fuite. D'autres aspects du jeu (*e.g.*, la musique) changent également à ce moment-là. Nous souhaitons pour cela limiter le couplage entre la classe `Astérix` et les éventuelles classes concernées.

   (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. (1.5pts)

   (d) Écrivez un code client où Astérix ramasse une potion magique, faisant ainsi fuir les ennemis. (0.5pt)

Annexe

```
public abstract class Ennemi {
  ...
  public void detruire() { ... }
  public void fuir() { ... }
  /** Retourne le nombre de points obtenus lorsque l'ennemi est detruit */
  public abstract int getNbPoints();
}
public class Pirate extends Ennemi {
  public int getNbPoints() { return 10; }
}
public class Romain extends Ennemi {
  public int getNbPoints() { return 20; }
}
public class Sanglier extends Ennemi {
  public int getNbPoints() { return 5; }
}
public interface Item { ... }
public class Amphore implements Item { ... }
public class CouronneDeLaurier implements Item { ... }
public class PotionMagique implements Item { ... }
```