Université Paris Cité **Programmation Orientée Objet**
L2 **Année 2024-2025**

TD - Séance n° 5 - Correction

Héritage et Modélisation

On souhaite modéliser en Java les différents éléments d'un jeu de rôle très simplifié (personnages, actions de ces personnages).

1 Personnages et Actions

Modélisation des personnages. Chaque joueur choisit une classe de personnage parmi un éventail de possibilités, ici au nombre de quatre. Il existe deux grandes catégories de personnages : *guerrier* et *initié*. Un guerrier peut être soit un *paladin*, soit un *voleur*, tandis qu'un initié peut être soit un *sorcier*, soit un *moine*. Au début du jeu, chaque personnage reçoit un nom ainsi qu'un certain nombre de points de vie : 200 points pour un paladin, 150 pour un voleur, et 100 pour chaque type d'initié.

Exercice 1 *Hiérarchie des personnages*

1. Quelle est la hiérarchie naturelle permettant de représenter les personnages du jeu ? Quels seront les champs nécessaires, leur nature (statique ou d'instance) et leur visibilité (privée ou publique) ?
2. Quelles sont parmi ces classes celles dont on peut être sûr qu'on ne créera jamais d'instances (et donc, si vous connaissez cette notion, qui devraient être abstraites) ?
3. Donner une implémentation minimale des classes de la branche de cette hiérarchie permettant de créer un voleur.

Correction : La hiérarchie et les noms des classes se déduisent naturellement de l'énoncé, de même que les classes qui devraient être abstraites. Voici une implémentation partielle de la branche menant à la classe du voleur :

```
public abstract class Personnage {
    private String nom;
    private int pdv;
    public Personnage(String nom) {
        this.pdv = pdvInitiaux();
        this.nom = nom;
    }
    public abstract int pdvInitiaux();  /* ou { return 0;} */
}
```

```
public abstract class Guerrier extends Personnage {
    public Guerrier(String nom) {
      super(nom);
    }
}
```

```
public class Voleur extends Guerrier implements Cloneable {
    public static final int pointsVieInitiaux = 150;
    public Voleur(String nom) {
      super(nom);
    }
    public int pdvInitiaux() {
      return pointsVieInitiaux;
    }
}
```

Modélisation des actions. Lors de la création de son personnage, chaque joueur choisit deux actions possibles : une action primaire et une action secondaire, parmi deux grandes catégories d'actions : utiliser une arme ou lancer un sort. Une arme peut être soit une épée, soit un gourdin. Un sort peut être un sort de lumière, un sort des ténèbres ou un sort élémentaire.

Les effets des différentes actions sont similaires : ils entraînent une altération des points de vie de l'auteur de l'action (négative, positive ou nulle) et une altération des points de vie de la cible de l'action (idem). Seules les valeurs numériques de ces altérations varient d'une action à l'autre, selon les règles suivantes :

élément	effet sur l'auteur	effet sur la cible
épée	-10 pv	-30 pv
gourdin	+5 pv	-10 pv
sort des ténèbres	-5 pv	-20 pv
sort élémentaire	+5 pv	-10 pv
sort de lumière	aucun	+20 pv

Exercice 2 *Hiérarchie des actions*

1. Quelle est la hiérarchie naturelle permettant de représenter les actions ? Quels seront les champs et méthodes nécessaires dans les classes de cette hiérarchie ?
2. Donner une implémentation minimale des classes de la branche de cette hiérarchie permettant de créer une épée.
3. Quelles seront les modifications des classes de personnages nécessaires pour munir chaque personnage de deux actions dès sa création, et pour lui permettre d'effectuer ses actions primaire et secondaire sur une cible ?

Correction : Là encore, la hiérarchie et les noms des classes se déduisent naturellement de l'énoncé, de même que les classes qui devraient être abstraites :

```
public abstract class Action /* ... */ {
  public abstract int alterationSource();
  public abstract int alterationCible();
}
```

```
public abstract class Arme extends Action {}
```

```
public class Epee extends Arme {
  public static final int alterationSource = -2;
  public static final int alterationCible  = -20;
  public int alterationSource() {
    return alterationSource;
  }
  public int alterationCible() {
    return alterationCible;
  }
}
```

Il faudra ajouter aux constructeurs des classes de personnages (concrètes et abstraites) deux arguments de plus (de type `Action` ou un sous-type d'`Action`, voir plus bas). La classe `Personnage` devient :

```
public abstract class Personnage {
  private String nom;
  private int pdv;
  private Action primaire;
  private Action secondaire;
  public Personnage(String nom, Action primaire, Action
      secondaire) {
    this.pdv = pdvInitiaux();
    this.nom = nom;
    this.primaire = primaire;
    this.secondaire = secondaire;
  }
  public abstract int pdvInitiaux();
  private void action(Personnage personnage, Action action) {
    personnage.pdv += action.alterationCible();
    pdv += action.alterationSource();
  }
  public void actionPrimaire(Personnage personnage) {
    action(personnage, primaire);
  }
  public void actionSecondaire(Personnage personnage) {
    action(personnage, secondaire);
  }
}
```

Choix des actions. Les choix d'actions primaire et secondaire possibles pour un personnage sont déterminés par sa classe, selon les règles suivantes :

personnage	**élément pour l'action primaire**	**élément pour l'action secondaire**
Paladin	épée	gourdin ou sort élémentaire
Voleur	épée ou gourdin	sort des ténèbres
Sorcier	sort des ténèbres	sort de lumière ou élémentaire
Moine	gourdin	sort de lumière

Exercice 3 *Contraintes de construction*

1. Comment faire en sorte d'interdire la compilation d'un programme dans lequel on créerait explicitement un personnage non conforme aux règles du jeu ?
2. Donner une nouvelle version de la classe du voleur ne permettant de créer un voleur qu'en respectant cette contrainte.

Correction : Au lieu de définir dans chaque classe concrète de personnage un unique constructeur attendant deux arguments de type `Action`, on écrit dans chaque classe un constructeur pour chaque choix de couples d'actions possibles, par exemple :

```
public Voleur(String nom, Epee epee, Tenebres tenebres) {
  super(nom, epee, tenebres);
}
public Voleur(String nom, Gourdin gourdin, Tenebres tenebres)
    {
  super(nom, gourdin, tenebres);
}
```

La construction d'un voleur équipé d'un autre couple d'actions que ceux mentionnés par ces constructeurs est alors interdite par simple application des règles de typage. Les autres classes de la hiérarchie des personnages n'ont pas à être modifiées.

Exercice 4 *Notions d'égalité de personnages et d'actions*

On souhaite redéfinir la méthode `equals()` de certaines des classes ci-dessus de manière à définir pour les actions et personnages les notions d'égalité suivantes :

— Deux actions seront considérées comme égales si et seulement si elles sont des instances de la même classe [1].

— Deux personnages seront considérés comme égaux s'ils sont des instances de la même classe, s'ils portent le même nom, ont le même nombre de points de vie et des choix d'action primaire et secondaires égaux au sens de la notion précédente.

Dans quelles classes faut-il redéfinir `equals()`, et avec quelle implémentation ?

Correction : Dans `Action` et dans `Personnage`. Les classes d'actions ne contiennent pas de champs, et les classes descendantes strictes de `Personnage` ne contiennent pas de champs d'instance supplémentaires, donc ces deux réimplémentations suffisent :

1. Dans cette modélisation simpliste, toutes les instances d'une même classe d'actions se valent : il est par exemple inutile de créer plus d'une seule instance d'une épée. Sans chercher à raffiner ce modèle, on pourrait éventuellement se contenter ici d'une égalité par référence.

```java
public abstract class Action implements Cloneable {
  // ...
  public boolean equals(Object o) {
    return this.getClass() == o.getClass();
  }
}
```

```java
public class Personnage {
  // ...
  public boolean equals(Object o) {
    if (o.getClass() != this.getClass()) {
      return false;
    }
    Personnage p = (Personnage) o;
    return nom.equals(p.nom) &&
        pdv == p.pdv &&
        primaire.equals(p.primaire) &&
        secondaire.equals(p.secondaire);
  }
}
```

2 États du jeu

L'état courant du jeu sera représenté par une classe encapsulant la liste des personnages encore présents dans le jeu à un instant donné.

Exercice 5 *Représentation d'un état*

1. Définir une classe permettant de représenter l'état courant du jeu.
   Correction :

```java
import java.util.ArrayList;

public class EtatJeu {
    private ArrayList<Personnage> etat;

    public EtatJeu() {
        etat = new ArrayList<>();
    }
    public EtatJeu(ArrayList<Personnage> etat) {
        this.etat = etat;
    }
    public void ajouterPersonnage(Personnage personnage)
        {
        etat.add(personnage);
    }
    public ArrayList<Personnage> getEtat() {
```

```
        return etat;
    }
}
```

2. Comment proposeriez-vous d'implémenter la méthode `equals()` de cette classe ?

   Correction : Pour `equals()`, on pourrait considérer que deux états du jeu sont égaux si leurs listes ont même longueur, et si à chaque position commune, on trouve deux personnages égaux au sens de l'exercice précédent. Ce choix est évidemment le plus simple à implémenter, mais pourrait poser problème si l'ordre de la liste était modifié au cours du temps. Il faudrait dans ce cas vérifier que chaque élément de l'une des deux listes est égal à un élément de l'autre.

   Implémentation de la première idée (plus simple) :

```
public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        EtatJeu autreEtat = (EtatJeu) obj;

        if (etat.size() != autreEtat.etat.size()) return
            false;

        for (int i = 0; i < etat.size(); i++) {
            if (!etat.get(i).equals(autreEtat.etat.get(i
               ))) {
                return false;
            }
        }
        return true;
    }
```

Exercice 6 *Clonage d'un état*

Afin de pouvoir conserver l'historique des états du jeu, on souhaite pouvoir faire une copie profonde d'un état (un clone).

1. En dehors de la classe représentant un état, dans quelles classes faudra-t-il réimplémenter (avec la visibilité `public`) la méthode `clone`, et comment ?

   Correction : Dans `Action` et `Personnage`. Là encore, les classes d'actions ne contiennent pas de champs, et les classes descendantes strictes de `Personnage` ne contiennent pas de champs d'instance supplémentaires, donc ces deux réimplémentations suffisent.

   Pour le "comment ?", la réponse la plus simple est "comme dans le cours." :

```
public abstract class Action implements Cloneable {
  // ...
  public Action clone() throws
     CloneNotSupportedException {
```

```
    return (Action) super.clone();
  }
}
```

```
public abstract class Personnage implements Cloneable {
  // ...
  public Personnage clone() throws
     CloneNotSupportedException {
    Personnage cloned = (Personnage) super.clone();
    cloned.primaire = primaire.clone();
    cloned.secondaire = secondaire.clone();
    return cloned;
  }
```

Pour le "pourquoi de cette manière ?", la réponse est évidemment plus complexe, mais elle pourrait être éventuellement détaillée, en commençant par expliquer la notion de package :

(a) La méthode `clone` d'`Object` est en `protected Object clone()` dans le package `java.lang`.

(b) Une méthode en `protected` dans une classe `A` est librement invocable sur une instance de `A` dans le même package, et peut être redéfinie dans toutes les classes descendantes de `A`, même dans un autre package. La redéfinition peut accéder à l'ancienne implémentation via `super`.

(c) Lorsque l'on redéfinit une méthode, on peut à la fois : étendre sa visiblité (passer de `protected` à `public`) ; restreindre son type de retour (passer d'un type `C` à un type `D` sous-type de `C`).

(d) Si `A` contient une méthode `protected` et si `B` est dans un autre package, on ne peut invoquer cette méthode que : dans une classe `B` sous-classe de `A` ; sur une référence dont le type est `B` ou un sous-type de `B` (en particulier `this`).

Les règles (a), (b) et (c) expliquent les possibilités de redéfinir `clone` dans `Action` et `Personnage`, d'accéder dans les deux à celle d'`Object` via `super`, et de restreindre les types de retour de ces deux redéfinitions.

La règle (d) explique pourquoi il est nécessaire de commencer par étendre la visibilité de `clone` dans `Action` pour pouvoir invoquer `clone` sur `primaire` et `secondaire` dans la redéfinition de `clone` de `Personnage`. Sans cette redéfinition, la méthode `clone` d'une action serait celle héritée d'`Object` et de niveau `protected`, et il faudrait pour que cette invocation soit valide que la classe `Action` soit descendante de `Personnage`.

2. Donner l'implémentation de `clone` dans la classe représentant un état du jeu.
   Correction : Il faut créer une nouvelle liste, y ajouter un clone de chaque élément de la liste initiale, renvoyer la liste créée.

3 Bonus

Exercice 7 Dans une modélisation un peu plus élaborée du jeu, certains événements pourraient avoir un effet plus durable que la simple modification ponctuelle

des points de vie des joueurs par leurs actions. Par exemple, un personnage pourrait être à chaque instant dans l'un des trois états suivants, et passer d'un état à l'autre en fonction du déroulement du jeu :

1. Normal : l'effet des actions du personnage est celui décrit en début d'énoncé.
2. Berserk : les altérations de chaque action sont multipliées par 2.
3. Fatigué : les altérations de chaque action sont divisées par 2.

Sans se préoccuper de notion d'égalité ou de clonage, comment modéliseriez-vous cet aspect du jeu en exploitant seulement l'héritage et la liaison dynamique, en particulier sans aucun `if` ?

Correction : En créant une hiérarchie d'états de personnages, en encapsulant dans chaque instance de personnage son état courant, et en déléguant le calcul de l'effet courant d'une action à cet état à partir de son effet initial :

```
public abstract class EtatPersonnage {
  abstract int alteration(int pdv);
}
```

```
public class EtatNormal extends EtatPersonnage {
  public int alteration(int pdv) {
    return pdv;
  }
}
```

```
public class EtatBerserk extends EtatPersonnage {
  int alteration(int pdv) {
    return pdv * 2;
  }
}
```

```
public class EtatFatigue extends EtatPersonnage {
    int alteration(int pdv) {
  return pdv / 2;
    }
}
```

```
public abstract class Personnage /* ... */ {
  // ...
  private EtatPersonnage etat = new EtatNormal();
  public void setEtat(EtatPersonnage etat) {
    this.etat = etat;
  }

  private void action(Personnage personnage, Action action) {
    personnage.pdv += etat.alteration(action.alterationCible
        ());
    pdv += etat.alteration(action.alterationSource());
```

```
    }
    // ...
}
```