POBJ – Examen Janvier 2011

Durée : 2 heures Tous documents autorisés

PROBLÈME : LES ARÈNES DU DESTIN

Dans un jeu de rôle massivement multi-joueurs en ligne, on vous a confié le développement d'un module du jeu où chaque joueur peut envoyer son personnage affronter d'autres personnages en tournoi dans des arènes.

Vu du joueur, le mécanisme est le suivant : il inscrit son personnage auprès d'un grand prêtre organisateur du tournoi qui l'ajoute à sa liste de combattants. Une fois le personnage inscrit, il peut continuer à vivre sa vie jusqu'à sa convocation pour le tournoi. Lorsque 32 combattants sont inscrits, le grand prêtre déclenche le tournoi, les 16 premiers combattants qui s'y rendent s'affronteront jusqu'à la mort. Un seul combattant est destiné à survivre. Tous les types de personnages (guerriers, magiciens, elfes, nains... la liste est très longue) peuvent participer à ces tournois. D'autres particularités seront décrites plus loin dans cet énoncé.

1 : Combats et combattants

Un tournoi est composé de combats à mort. Pour permettre la réalisation des combats, les programmeurs du jeu ont fait en sorte que :

- toutes les classes de personnages (sauf les magiciens) implémentent une interface *IConvocable* qui définit la méthode *public void convoquer(Tournoi t)*. Cette méthode permet au grand prêtre de notifier tous les combattants qui se sont inscrits auprès de lui de la tenue prochaine d'un tournoi lorsque 32 combattants sont inscrits.
- toutes les classes définissant des types de personnages (sauf les magiciens) implémentent l'interface *ICombattant,* qui étend *IConvocable*, définie ci-dessous :

```
public interface ICombattant extends IConvocable {
        /**
         * indique si le personnage a encore des points de vie du
         */
        public boolean isDead();
        /**
         * diminue le nombre de points de vie du personnage
         * @param pv : nombre de points perdus
         */
        public void perdPV(int pv);
        /**
         * renvoie le nom du personnage
         */
        public String getNom();
        /**
```

```
         * realise une attaque sur un adversaire
         * @return le nombre de points d'attaque de cette attaque
         */
        public int attaquer();
        /**
         * realise une parade d'un coup de l'adversaire
         * @return le nombre de points de défense de cette parade
         */
        public int defendre();
}
```

Le mécanisme d'un combat est le suivant.

- A chaque tour de jeu, un combattant attaque et l'autre défend.
- Si le nombre de points de défense renvoyé par la parade du défenseur est supérieur ou égal au nombre de points d'attaque du coup de l'attaquant, le défenseur perd un seul point de vie. Sinon, le défenseur perd un nombre de points de vie égal à la différence entre les deux scores.
- On vérifie si le défenseur est toujours vivant. S'il est mort, le combat s'arrête. Sinon, les rôles de défenseur et d'attaquant sont inversés.
- On recommence ainsi jusqu'à la mort d'un des deux combattants.

Question 1

Dans une classe *Combat*, implémentez une méthode statique *ICombattant confronter(ICombattant c1, ICombattant c2)* qui réalise le mécanisme ci-dessus et renvoie le combattant survivant. On considère que c'est le combattant *c1* qui attaque en premier.

Une solution parmi d'autres (Yann en propose une plus courte en permutant attaquant et défenseur)

```
package arenes;

public class Combat {

        public static ICombattant confronter(ICombattant adv1, ICombattant adv2){
                System.out.println("Combat entre "+adv1.getNom()+ " et "+ adv2.getNom());
                while (true){
                        int attaque1 = adv1.attaquer();
                        int defense1 = adv2.defendre();
                        int degat1 = Math.max(1, attaque1-defense1);
                        adv2.perdPV(degat1);
                        if (adv2.isDead()) {
                                System.out.println(adv2.getNom() + " est mort");
                                return adv1;
                        }
                        int attaque2 = adv2.attaquer();
                        int defense2 = adv1.defendre();
                        int degat2 = Math.max(1, attaque2-defense2);
                        adv1.perdPV(degat2);
                        if (adv1.isDead()) {
                                System.out.println(adv1.getNom() + " est mort");
                                return adv2;
                        }
                }
        }
}
```

Vérifiez que l'algo est correct, -25% par type de faute syntaxique (oubli de parenthèses, ...), -50% par type de fautes plus graves, sur l'algo ou sur les bases de la prog objet. Ne pénalisez pas ce qui marche mais est inefficace ou inélégant, l'étudiant se pénalise lui-même en perdant du temps.

2 : Cas particulier des magiciens.

Les magiciens (personnages qui sont des instances de la classe *Magicien*) ne disposent pas d'une méthode *attaquer()* ni d'une méthode *defendre()*, mais d'une méthode *jeterSortOffensif(),* une méthode *jeterSortDefensif()* et une méthode *jeterSortDeSoin()* qui renvoient toutes les trois un entier correspondant à un score d'attaque, un score de défense ou bien un nombre de points de vie regagnés. Quand c'est leur tour d'attaquer (ou de défendre), ils peuvent choisir de jeter un sort offensif (ou défensif), ou bien un sort de soin. S'ils choisissent de jeter un sort de soin, cela fait remonter leur nombre de points de vie et leur score d'attaque (ou de défense) est nul.

On supposera que les magiciens ont une méthode *isDead()* et une méthode *perdPV()* comme tous les autres types de personnages, bien qu'ils n'implémentent pas l'interface *Icombattant*. Ils ont aussi une méthode *gagnePV()* similaire à *perdPV()* qui permet de remonter leur nombre de points de vie.

Question 2

Quel *design pattern* faut-il utiliser pour que les magiciens puissent participer aux combats ? Proposez une implémentation pour ce *design pattern*. Vous donnerez les attributs nécessaires, les constructeurs et l'implémentation des méthodes *getNom(), attaquer(), defendre()* et *isDead().* Vous supposerez que le choix de se soigner est aléatoire, de probabilité 50%. On vous donne ci-dessous le code de la méthode *convoquer(Tournoi t)*, qu'on trouve aussi dans toutes les classes de personnages convocables.

```
public void convoquer(Tournoi t) {
        System.out.println(getNom() + " est convoqué");
        t.entrerEnLice(this);
    }
```

```
package arenes;

public class MagicienToCombattantAdapter implements ICombattant {
private Magicien mag ;

    public MagicienToCombattantAdapter(Magicien g) {
    mag = g;
    }
    @Override
    public int attaquer() {
        if (Math.random()>0.5)              return mag.jeterUnSortOffensif();
        else {
            mag.gagnePV(mag.jeterUnSortDeSoin());
                return 0;
        }
    }

    @Override
    public int defendre() {
        if (Math.random()>0.5)              return mag.jeterUnSortDefensif();
        else {
            mag.gagnePV(mag.jeterUnSortDeSoin());
                return 0;
        }
    }
```

```
	@Override
	public String getNom() {
		// TODO Auto-generated method stub
		return mag.getNom();
	}

	@Override
	public boolean isDead() {
		// TODO Auto-generated method stub
		return mag.isDead();
	}
	@Override
	public void perdPV(int pv) {
		// TODO Auto-generated method stub

	}

	@Override
	public void convoquer(Tournoi t) {
		System.out.println(getNom() + " est convoquÃ©");
		t.inscrire(this);
	}

}
```

+25% pour avoir identifié l'adaptateur
+35% pour l'avoir réalisé correctement (attribut, implements...)
+10% par méthode correcte sur les 4 demandées

3 : Bénédiction ou malédiction

Un combattant qui participe à un tournoi peut être « béni». Si cela se produit, le score de défense du personnage est doublé systématiquement durant tout le tournoi. Il peut aussi être « maudit », auquel cas c'est son score d'attaque qui est doublé systématiquement durant tout le tournoi.

Question 3

Quel *design pattern* proposez-vous d'utiliser pour implémenter ce mécanisme de bénédiction ? Proposez une implémentation de ce mécanisme. Vous donnerez les attributs nécessaires, les constructeurs et l'implémentation des méthodes *attaquer(), defendre() et isDead().*

```
public class CombattantDecore {
	private ICombattant combattant;

	public CombattantDecore(ICombattant c){
		combattant = c;
	}
	public ICombattant getCombattant(){
		return combattant;
	}

	public int attaquer(){
		return combattant.attaquer();
	}

	public int defendre(){
		return combattant.defendre();
	}

	public String getNom() {
		return combattant.getNom();
	}

	public boolean isDead() {
		return combattant.isDead();
	}
```

```
        public void perdPV(int pv) {
                combattant.perdPV(pv);
        }

        public void convoquer(Tournoi t) {
                combattant.convoquer(t);
        }

}

public class CombattantMaudit extends CombattantDecore implements ICombattant{
        public CombattantMaudit(ICombattant c){
                super(c);
        }
        public int attaquer(){
                return getCombattant().attaquer()*2;
        }
}

public class CombattantBeni extends CombattantDecore implements ICombattant{
        public CombattantBeni(ICombattant c){
                super(c);
        }
        public int defendre(){
                return getCombattant().defendre()*2;
        }
}
```

30% pour avoir identifié décorateur
50% pour l'avoir implémenté correctement (attribut et interfaces)
10% chacune pour les méthodes attaquer() et defendre()
si < 100%, bonus de 20% pour avoir pensé à utiliser l'héritage pour factoriser le code (mais max à 100%)

4 : Convocation des combattants

Le grand prêtre est responsable de la convocation des personnages. Les tournois n'ont lieu que lorsque 32 personnages se sont inscrits auprès de lui. Lorsque c'est le cas, le grand prêtre organise un tournoi (il crée une instance de la classe *Tournoi*) et il notifie les combattants en appelant leur méthode *convoquer(Tournoi t)* qui prend en paramètre le tournoi nouvellement créé.

Question 4

Quel *design pattern* peut-on utiliser pour implémenter ce mécanisme d'inscription et convocation ? Donnez le code complet de la classe *GrandPretre* qui réalise l'inscription et la convocation. Vous supposerez que les personnages appellent une méthode *inscrire(IConvocable perso)* de la classe *GrandPretre* en lui passant une référence sur eux-mêmes.

```
public class GrandPretre {
        private ArrayList<IConvocable> heros;

        public GrandPretre(){
                heros = new ArrayList<IConvocable>();
        }
        public void inscrire(IConvocable hero){
                heros.add(hero);
                if (heros.size()==32){
                        Tournoi t = new Tournoi();
                        for (IConvocable comb : heros)
                        comb.convoquer(t);
                }
        }
}
```

25% pour avoir identifié Observer
25% pour l'attribut
25% pour le constructeur
25% pour la méthode inscrire

5 : Préparation et réalisation du tournoi

Lorsque les personnages sont convoqués, seuls les 16 premiers personnages qui entrent en lice y participent. Aussitôt que le 16ième combattant entre en lice, le tournoi commence. Dans un premier temps, un combattant choisi au hasard est béni et un combattant est maudit. Cela peut être le même combattant qui est choisi deux fois.

Puis les 16 combattants s'affrontent au sein du tableau du tournoi. Ils sont répartis dans 8 combats singuliers dans l'ordre dans lequel ils sont entrés en lice. On commence par tous les « huitièmes de finale », puis les quarts de finale, etc. Le survivant d'un combat rencontre le survivant du combat voisin dans l'arbre, jusqu'au combat final qui confronte les deux derniers survivants. On ne considère pas le fait que des combattants puissent regagner des points de vie entre deux combats.

Les questions qui suivent sont destinées à réaliser l'implémentation de la classe *Tournoi* qui permet de réaliser ce qui est décrit ci-dessus.

Question 5.1

Proposez dans la classe *Tournoi* l'implémentation d'une méthode nommée « *entrerEnLice* » dont vous choisirez les paramètres et le type de retour, qui permet à un personnage de rejoindre le tournoi.

```
public class Tournoi {
	private ArrayList<ICombattant> heros;
	private static Random generateur;

	public Tournoi(){
		heros=new ArrayList<ICombattant>();
		generateur = new Random();
	}
	public void entrerEnLice(ICombattant hero){
		heros.add(hero);
		if (heros.size()==16){
			benir();
			realiser();
		}
	}
```

Il faut penser à déclencher le tournoi, donc appeler benir() et realiser()...
Si l'idée est bonne, -25% pour les fautes syntaxiques, -50% pour les fautes plus graves
Sinon, 50% max
Ne pas pénaliser s'ils ne donnent pas la classe ni le constructeur (pas demandé explicitement)

Question 5.2

Proposez l'implémentation d'une méthode nommée *void benir()* qui choisit aléatoirement le combattant béni et le combattant maudit et procède à leur transformation.

```
	public void benir() {
		int index1 = generateur.nextInt(heros.size());
		ICombattant hero = heros.get(index1);
		CombattantMaudit heroBeni = new CombattantMaudit(hero);
		heros.remove(index1);
		heros.add(heroBeni);
```

```
            System.out.println(heroBeni.getNom()+ " est beni");
            int index2 = generateur.nextInt(heros.size());
            ICombattant hero2 = heros.get(index2);
            CombattantBeni heroMaudit = new CombattantBeni(hero2);
            heros.remove(index2);
            heros.add(heroMaudit);
            System.out.println(heroMaudit.getNom()+ " est maudit");
    }
```

Si l'idée est bonne, -25% pour les fautes syntaxiques, -50% pour les fautes plus graves
Sinon, 0

Question 5.3

Proposez l'implémentation d'une méthode nommée *void realiser()* qui gère la succession des combats jusqu'à ce qu'il n'y ait plus qu'un seul survivant.

```
    public void realiser(){
            while (heros.size()>1){
                    int taille = heros.size();
                    for (int index=0;index< taille;index+=2){
                            Combat.confronter(heros.get(index), (ICombattant)heros.get(index+1));
                    }
                    for (int i=taille-1;i>=0; i--){
                            if (heros.get(i).isDead()) heros.remove(i);
                    }
            }
    }
```

Attention, remove() dans une liste qu'on est en train de parcourir = danger !
Ici, je la parcours en marche arrière.
D'autres solutions sont possibles, mais soyez vigilants, c'une une question difficile qui devrait faire la différence entre les bons et les très bons.

Si l'idée est bonne, -25% pour les fautes syntaxiques, -50% pour les fautes plus graves
Sinon, 0

6 : Combats entre équipes

Pour ajouter du piment au jeu, on décide de permettre l'organisation de combats entre équipes de personnages.

Pour réaliser cette extension, vous décidez de faire en sorte qu'une équipe de personnages puisse être vue comme un combattant unique, afin de ne pas avoir à toucher à la classe *Combat*.

- Une équipe s'inscrit comme un combattant unique.
- Quand c'est au tour d'une équipe d'attaquer, on ajoute les points d'attaque de tous ses membres encore vivants. De même pour défendre, avec les points de défense.
- Les dégats infligés à une équipe sont répartis équitablement entre tous ses membres encore vivants. Une équipe est considérée comme morte quand tous ses membres sont morts.
- Une équipe peut être bénie ou maudite de la même façon qu'un combattant individuel (sans donner lieu à une bénédiction/malédiction des combattants individuels).
- Enfin, quand on convoque une équipe, tous ses membres sont convoqués individuellement.

Question 6.1

Quel *design pattern* proposez-vous d'utiliser pour implémenter cette notion d'équipe ? Proposez une implémentation complète de la notion d'équipe. Vous pourrez notamment coder une méthode privée

void purger() qui supprime d'une équipe tous ses membres morts. Si vous faites ce choix, vous ferez attention à appeler cette méthode là où c'est nécessaire.

```
public class Equipe implements ICombattant{
	private ArrayList<ICombattant> membres;

	@Override
	public int attaquer() {
		purger();
		int retour = 0;
		for (ICombattant combattant : membres){
			retour += combattant.attaquer();
		}
		return retour;
	}

	@Override
	public int defendre() {
		purger();
		int retour = 0;
		for (ICombattant combattant : membres){
			retour += combattant.defendre();
		}
		return retour;
	}

	@Override
	public String getNom() {
		String retour = "equipe composee de :";
		for (ICombattant combattant : membres){
			retour += combattant.getNom() +",";
		}
		return retour;
	}
	private void purger(){
		int taille = membres.size();
		for (int i=taille-1;i>=0; i--){
			if (membres.get(i).isDead()) membres.remove(i);
		}
	}
	@Override
	public boolean isDead() {
		for (ICombattant combattant : membres){
			if (!combattant.isDead()) return false;
		}
		return true;
	}
	@Override
	public void perdPV(int pv) {
		int partage = pv/membres.size();
		for (ICombattant combattant : membres){
			combattant.perdPV(partage);
		}
	}

	@Override
	public void convoquer(Tournoi t) {
		System.out.println(« L' » + getNom() + « est convoquée »);
		t.entrerEnLice(this);
		}
	}
}
```

25% pour avoir identifié le pattern Composite
25% pour l'implémentation correcte (implements et attribut)
Le reste pour les méthodes (nombreuses façons de s'y prendre)

Question 6.2

Dans une classe *Main* et sa méthode principale *main()* dotée des paramètres habituels, écrivez le

code qui permet de créer une équipe de personnages constituée d'un magicien et d'un ou plusieurs guerriers dont les noms sont passés en paramètres à l'appel du programme. Vous supposerez l'existence de constructeurs publics *Guerrier(String nom)* et *Magicien(String nom)* qui, entre autres, initialisent les points de vie des personnages sans que vous ayez à vous en occuper. *Par ailleurs,* vous ferez l'hypothèse qu'au moins deux noms sont passés en paramètre au programme.

```
import java.util.ArrayList;

public class Main {

	public static void main(String[] args) {
		GrandPretre pretre = new GrandPretre();
		ArrayList<Guerrier> guerriers = new ArrayList<Guerrier>();
		Magicien m1 = new Magicien(args[0]);

		for (int i=1;i<args.length;i++){
			guerriers.add(new Guerrier(args[i]));
		}
		pretre.inscrire(new MagicienToCombattantAdapter(m1));
		for (Guerrier g : guerriers){
		pretre.inscrire(g);
		}
	}
}
```

Si l'idée est bonne, -25% pour les fautes syntaxiques, -50% pour les fautes plus graves
`Sinon, 0`