Programmation et Conception Orientées Objet
TD n° 3 : Patrons de conception

Présentation du travail

Le but de ce TD est de proposer une architecture de classes et d'interfaces qui exploite le polymorphisme à l'aide d'un exemple concret. Plusieurs patrons de conception classiques sont introduits et leur utilité mise en évidence.

Un patron de conception (*design pattern*) est une technique de programmation orientée objet qui apporte une solution adaptée à un problème classique. Les patrons présentés dans ce TD sont :

— le patron de fabrique (*factory method pattern*),
— le patron singleton (*singleton pattern*),
— le patron de composition (*composite pattern*),
— le patron d'adaptation (*adapter pattern)*,
— le patron de stratégie (*strategy pattern)*,
— le patron de décoration (*decorator pattern*) et
— le patron de méthode (*method template pattern*).

1 Contexte

Vous faites partie d'une équipe de développement d'un nouveau RPG (*role playing game*, jeu de rôle). Le joueur va être confronté à des monstres qu'il devra vaincre pour avancer dans le scénario. On cherche ici à gérer les affrontements qui vont survenir.

Le joueur est caractérisé par :
— sa position,
— ses points de vie,
— les dégâts qu'il inflige et
— sa vitesse de déplacement.

Les monstres possèdent les mêmes caractéristiques que le joueur et réagissent en fonction d'un comportement prédéfini. Pour l'instant, trois types de monstres sont proposés.

Dénomination	**PV**	**Dégâts**	**MVT**	**Comportement**
Archer gobelin	4	1	15	Fuit si blessé, se met à distance (10m), attaque
Guerrier orc	10	3	10	Engage le joueur, attaque
Brigand	8	2	10	Fuit à mi-PV, engage le joueur, attaque

Fuir signifie s'éloigner à pleine vitesse du joueur à pleine vitesse. Un monstre est engagé avec le joueur s'il est à moins d'un mètre de celui-ci. Lors d'une attaque, le monstre inflige ses dégâts au joueur (on ne gère pas encore le cas de la mort du joueur).

À chaque fois qu'un monstre agit, il réalise la première action de son comportement qui n'est pas sans effet. Par exemple, un gobelin blessé fuit, un gobelin non blessé à moins de 10 mètres du joueur s'éloigne sinon il attaque.

L'équipe a déjà écrit une classe pour faciliter la gestion des déplacements :

```java
public class Pos {
    private final double x;
    private final double y;

    public Pos(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /* S'appoche (ou s'éloigne si d < 0) de la position pos jusqu'à une
       distance minimale de 1. */
    public Pos moveToward(Pos pos, double distance) {
        double dx = x - pos.x;
        double dy = y - pos.y;
        double d = getDistance(pos);
        double moveDistance = (d < 1) ? 0 : Math.min(distance, d - 1);
        return new Pos(dx / d * moveDistance, dy / d * moveDistance);
    }

    /* Retourne la distance entre le point courant et celui passé en
       paramètre. */
    public double getDistance(Pos pos) {
        double dx = x - pos.x;
        double dy = y - pos.y;
        return Math.hypot(dx, dy);
    }

    public double getX() {  return x;  }
    public double getY() { return y; }
}
```

Proposez une implémentation.

2 Génération des monstres (*factory and singleton pattern*)

Pour générer un monstre, il faut appeler son constructeur pour l'instancier. On cherche à faciliter la création en créant une classe qui sait générer tous les monstres via un identifiant donné sous forme de chaîne de caractères.

Quels sont les avantages d'une telle classe ?

Y-a-t-il un intérêt à avoir plusieurs instances de cette classe ? Et une instance unique ?

Proposez une implémentation.

3 Monstres fusionnés (*composite pattern*)

Par une magie impie, il est possible de fusionner deux monstres pour créer un horrible mutant. Celui-ci possède les points de vie et les dégâts combinés des deux monstres qui le composent. Cependant, il se déplace à la vitesse la plus petite de ceux-ci. Son comportement se limite à engager le joueur et à l'attaquer. Notez que les pires sorciers du pays n'hésitent pas à fusionner des monstres déjà issus d'une fusion.

Proposez une implémentation.

4 Le code du stagiaire (*adapter pattern*)

Un stagiaire enthousiaste a voulu créer ses propres monstres pour les ajouter au jeu. Certaines de ses propositions paraissent intéressantes mais malheureusement il ne s'est pas synchronisé avec le reste de l'équipe. Les monstres qu'il propose sont codés dans des classes qui implémentent l'interface :

```
public interface OtherMonster {
    double getX();
    double getY();
    double setX(double x);
    double setY(double y);
    int getMaxLifePoints();
    int getLifePoints();
    void takeDamages(int damages);
    int getDamages();
    double getMovement();
    void act(Player player);
}
```

Vous ne souhaitez pas regarder le reste de son code abscons en détail, . . . et encore moins le modifier. Quelle solution envisager pour tout de même pouvoir l'utiliser dans votre projet ?

5 Factoriser le code (*strategy, decorator and method template pattern*)

Vous vous rendez compte que posséder une classe par type de monstre va rapidement faire exploser la quantité de code à écrire et à maintenir. Vous envisagez une classe de monstres générique dont le comportement est paramétrable. Seuls les monstres vraiment particuliers disposeront alors d'une classe spécifique.

Comment paramétrer le comportement ?

Vous remarquez également que le comportement global d'un monstre est constitué d'une liste de comportements plus simples. L'action effectivement réalisée est celle liée au premier comportement qui fait réellement réagir le monstre. L'équipe envisage également d'ajouter des sortilèges qui peuvent modifier dynamiquement le comportement des monstres affectés.

Trouvez une solution à ce problème pour enfin pouvoir partir en pause café.