Ingénierie Logicielle

Examen 2023 - 2024
Session 1

1

A

Le framework est extensible sans modification grâce au polymorphisme : l'ajout de sous-types de Hero ou Obstacle permet d'intégrer de nouvelles fonctionnalités

B

```
public class Pacman extends Heros {
    public Pacman() {}

    @Override
    public String getDescription() {
        return "Pacman";
    }
}
```

C.1

Ligne 2 du listing 2 (indirectement la méthode recontrer du listing 2). C'est une inversion de contrôle car c'est du code exterieur non nécessairement connu à la compilation (redéfinie dans les sous classes), on peut même dire que c'est un point d'extension

C.2

La redéfinition de getDescription dans les sous-classes de Hero permet à l'utilisateur d'ajouter des fonctionnalités spécifiques au framework

D.1

Ligne 4 & 5 du listing 1. Ce sont des affectation polymorphique, Hero et Obstacle sont des variables appartenent à plusieurs types, peuvent référencer n'importe quel sous-type

D.2

Il n'est pas possible de réaliser un framework sans polymorphisme car le but d'un framework est de lui ajouter de nouvelles fonctionnalités en l'étendant ou le paramétrant. Il définie un type générique à sa base que l'utilisateur spécialise via des sous-types pour ajouter des fonctionnalités

E

	this	**o**
statique	Heros	Obstacle
dynamique	Vrai type (Exemple : Pacman)	Vrai type (Exemple : Cerise)

2

A

Plusieurs raisons :

- Complexité : Code complexe, beaucoup de ligne pour une seule méthode
- Principe d'ingénieurie logicielle : Force le développeur à réecrire dans la méthode quand il veut rajouter un nouveau sous type, potentiel nouveaux bugs inatendue (non régréssion)
- Performance : Laisse le langage a objet choisir quelle méthode utiliser grâce à la liaison dynamique

B.1

`rencontrer(Fantome f)`. f est statiquement et dynamiquement typé en Fantome, la méthode est redéfinie dans la sous classe de Hero, Pacman, l'envoie de message ce fera sur la méthode de la classe Pacman

B.2

`rencontrer(Obstacle r)`. h est statiquement typé en Heros, comme aucunes méthode avec le type recontrer(Fantome) n'existe, on remonte dans la hiérarchie des classes et choisie le type Obstacle (Fantome est sous classe de Obstacle). On regarde ensuite si dans la classe Pacman est redéfinie la méthode recontrer(Obstacle), elle ne l'est pas. L'envoie de message ce fera donc sur la méthode de la classe Heros

B.3

`rencontrer(Obstacle r)`. f est statiquement typé en Obstacle, comme précédement, est choisie par défaut la méthode rencontrer(Obstacle) de la classe Heros. On regarde si la méthode rencontrer(Obstacle) est redéfinie dans la classe Pacman, elle ne l'est pas. L'envoie de message ce fera donc sur la méthode de la classe Heros

3

A

- Avanatages : Passer rapidement et facilement d'un état à un autre. Facilité d'extension, rajouter une classe pour rajouter un état
- Inconvénients : Difficile de cumuler les états. Beaucoup de classes différentes

B

- Avanatages : Facilité à cumuler les états (pacman peut avoir plusieurs super pouvoir en même temps)
- Inconvénients : Difficile de retirer des états. L'objet de l'instance n'est plus le même (rend les test instance of impossible)

C

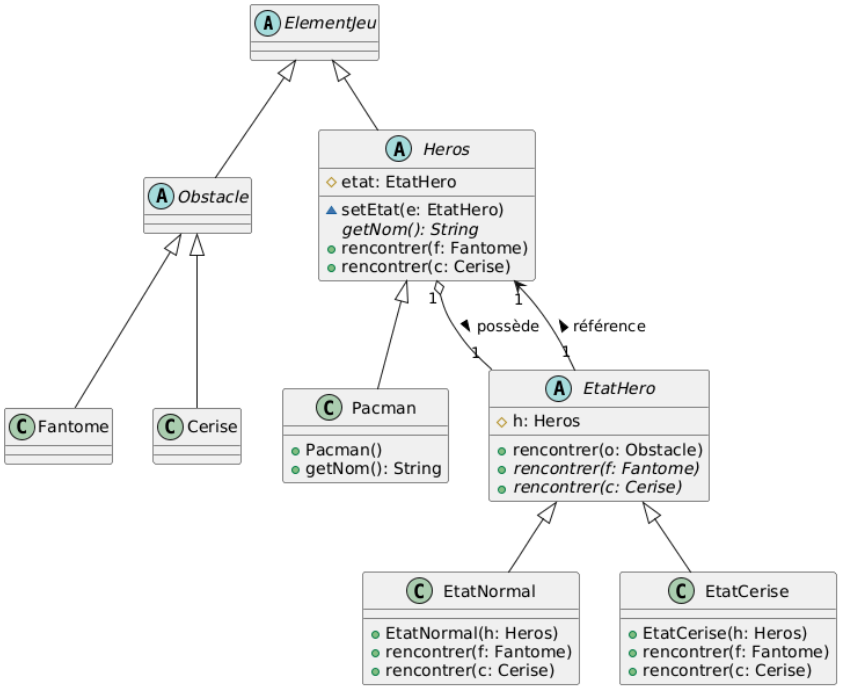
Dans notre cas le pattern State est bien plus adapté à notre problème. On veut pouvoir passé facilement et rapidement d'un état à un autre. Le fait que le pattern Decorator empeche de facilement retirer les états est trop contraingnant

Code :

```
abstract class EtatHero {
    protected Heros h;

    public void rencontrer(Obstacle f) {
        System.out.println("Suis je utile");
    }
    public abstract void rencontrer(Fantome f);
    public abstract void rencontrer(Cerise c);
}

public class EtatNormal extends EtatHero {
    public EtatNormal(Heros h) { this.h = h; }

    public abstract void rencontrer(Fantome f) {
        System.out.println(this.h.getNom() + "rencontre un Fantome, il meurt");
```

```
    }

    public abstract void rencontrer(Cerise c) {
        System.out.println(this.h.getNom() + "rencontre une Cerise, il envoie les
fantomes en prison");
    }
}

public class EtatCerise extends EtatHero {
    public EtatCerise(Heros h) { this.h = h; }

    public abstract void rencontrer(Fantome f) {
        System.out.println(this.h.getNom() + "rencontre un Fantome, il l'envoie en
prison");
    }

    public abstract void rencontrer(Cerise c) {
        System.out.println(this.h.getNom() + "rencontre une Cerise, il envoie les
fantomes en prison");
    }
}

abstract class Heros extends ElementJeu {
    protected EtatHero etat;

    void setEtat(EtatHero e) {
        this.etat = h;
    }
    abstract String getNom();

    public void rencontrer(Fantome f) {
        etat.rencontrer(f);
    }

    public void rencontrer(Cerise c) {
        etat.rencontrer(c);
    }
}

public class Pacman extends Heros {
    public Pacman() { this.setEtat(new EtatNormal(this)); }

    public String getNom() { return "Pacman"; }
}
```

UML :



4

A

o est statiquement typé en Obstacle et h en Heros, donc l'envoie de message ce fera à partir des méthodes de la classe Heros, de paramètre Obstacle. On cherche ensuite dans la sous classe de Heros une redéfinition de la méthode rencontrer(Obstacle), il n'y en a aucune, donc l'envoie de message se fera à partir de la méthode de la classe Heros et non de la classe Pacman

B

flm + pas compris sans gemini + si on arrive la on a 12 14 c'est large suffisant