

Design Pattern State

State Pattern : le problème par l'exemple

```
public class Pizza {

    public final static int COOKED = 0;
    public final static int BAKED = 1;
    public final static int DELIVERED = 2;

    private String name;

    int state = COOKED;

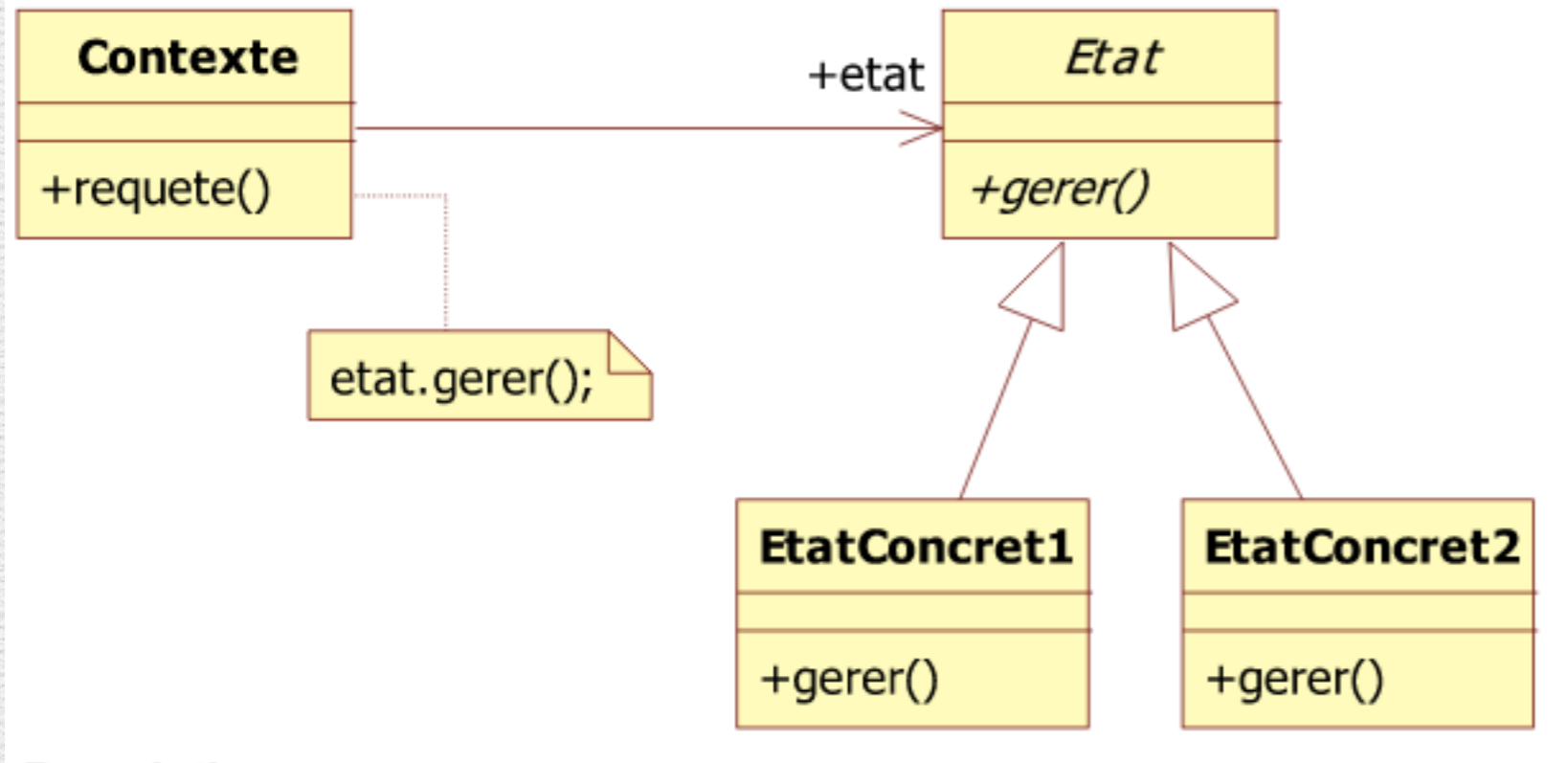
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getState() {
        return state;
    }
    public void setState(int state) {
        this.state = state;
    }
    public void bake() throws Exception {
        if(state == COOKED) {
            System.out.print("Baking the pizza...");
            state = BAKED;
        }
        else if(state == BAKED) {
            throw new Exception("Can't bake a pizza
already baked");
        }
        else if(state == DELIVERED) {
            throw new Exception("Can't bake a pizza
already delivered");
        }
    }
}
```

```
public void deliver() throws Exception {

    if(state == COOKED) {
        throw new Exception("Can't deliver a
pizza not baked yet");
    }
    else if(state == BAKED) {
        System.out.print("Delivering the
pizza...");
        state = DELIVERED;
    }
    else if(state == DELIVERED) {
        throw new Exception("Can't deliver a
pizza already delivered");
    }
}
```

**DANS CETTE VERSION
ON NE PEUT PAS
FACILEMENT RAJOUTER
UN ETAT ... UNE
METHODE DO SERAIT
MIEUX!!!**

Diagramme de classes :



State Pattern :
la solution
généralisée

Permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

✓ Extensibilité

1.1 Factorisation de protocole pour tous les états

1.2 L'ajout ou la suppression d'un état n'implique pas de modification de code

✓ Gestion simplifiée

2.1 Changement d'état possible à l'exécution sans destruction

2.2 Découplage du comportement de chaque état

```

public class Pizza {
    private String name;
    //State initialization
    private PizzaState state = new CookedPizzaState(this);

    public Pizza() { }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public PizzaState getState() {
        return state;
    }

    public void setState(PizzaState state) {
        this.state = state;
    }

    public void bake() throws Exception {
        this.state.bake();
    }

    public void deliver() throws Exception {
        this.state.deliver();
    }
}

```

```

public abstract class PizzaState {
    protected Pizza pizza;

    public PizzaState(Pizza pizza){
        this.pizza = pizza;
    }
    abstract public void bake() throws Exception;

    abstract public void deliver() throws Exception;

}

```

```

public class BakedPizzaState extends PizzaState {
    public BakedPizzaState(Pizza pizza) {
        super(pizza);
    }
    public void bake() throws Exception {
        throw new Exception("Can't bake a pizza already baked");
    }
    public void deliver() throws Exception {
        System.out.print("Delivering the pizza...");
        pizza.setState(new DeliveredPizzaState(this.pizza));
    }
}

```

State Pattern : la solution

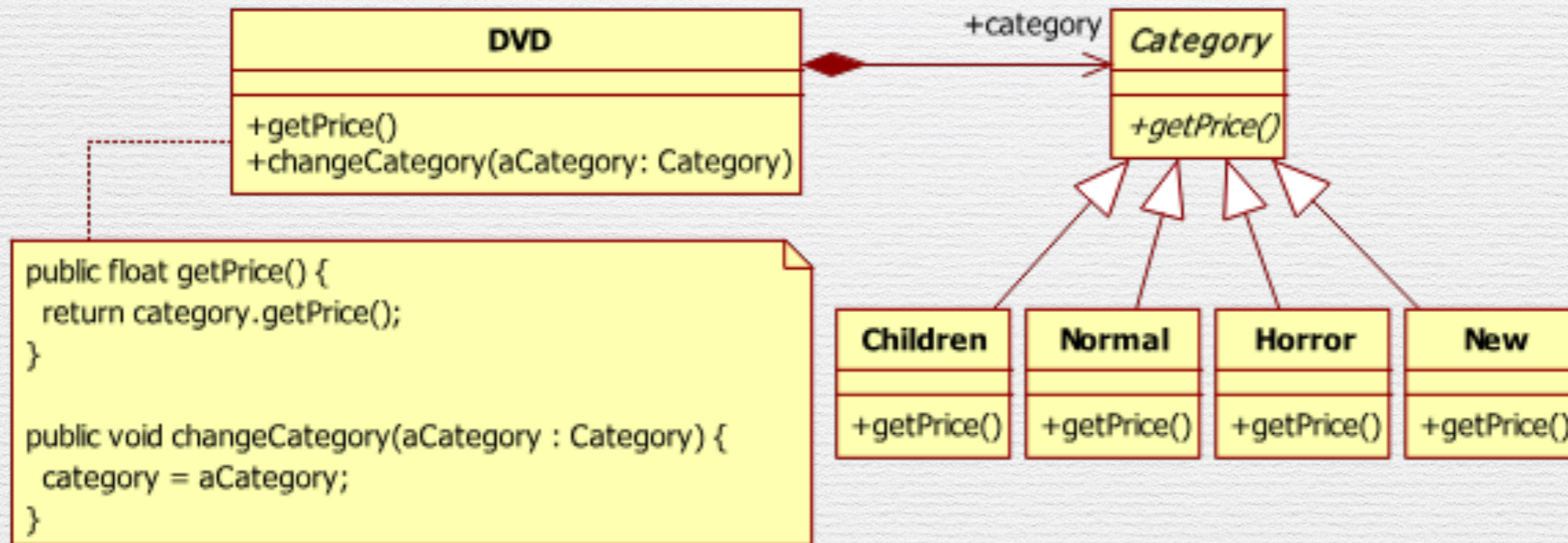
A vous...

ATTENTION ICI ON NE GERE PAS LE CHANGEMENT D'ETAT AUTOMATIQUEMENT

- ✓ Modéliser le fonctionnement d'une vidéothèque. La vidéothèque met à disposition de ses clients des DVD selon trois catégories : Enfant, Normal et Nouveauté. Un DVD est dans la catégorie Nouveauté pendant quelques semaines, puis passe dans l'une des autres catégories. Le prix des DVD dépend de la catégorie. Il est probable que le système évolue pour que la catégorie Horreur soit ajoutée.

http://www.goprod.bouhours.net/?page=problem&pb_id=5

Pattern Etat



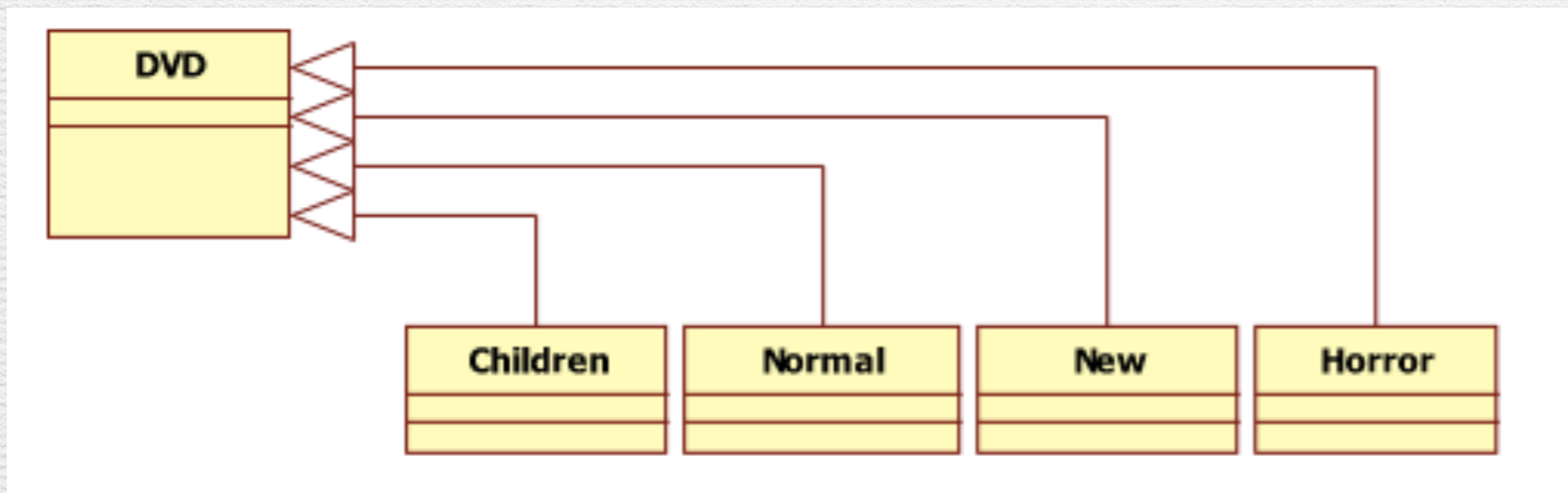
Extensibilité

- 1.1 Factorisation de protocole pour tous les états
- 1.2 L'ajout ou la suppression d'un état n'implique pas de modification de code

Gestion simplifiée

- 2.1 *Changement d'état possible à l'exécution sans destruction*
- 2.2 *Découplage du comportement de chaque état*

Pattern Etat Abimé



1. Extensibilité

✓ 1.1 *Factorisation de protocole pour tous les états*

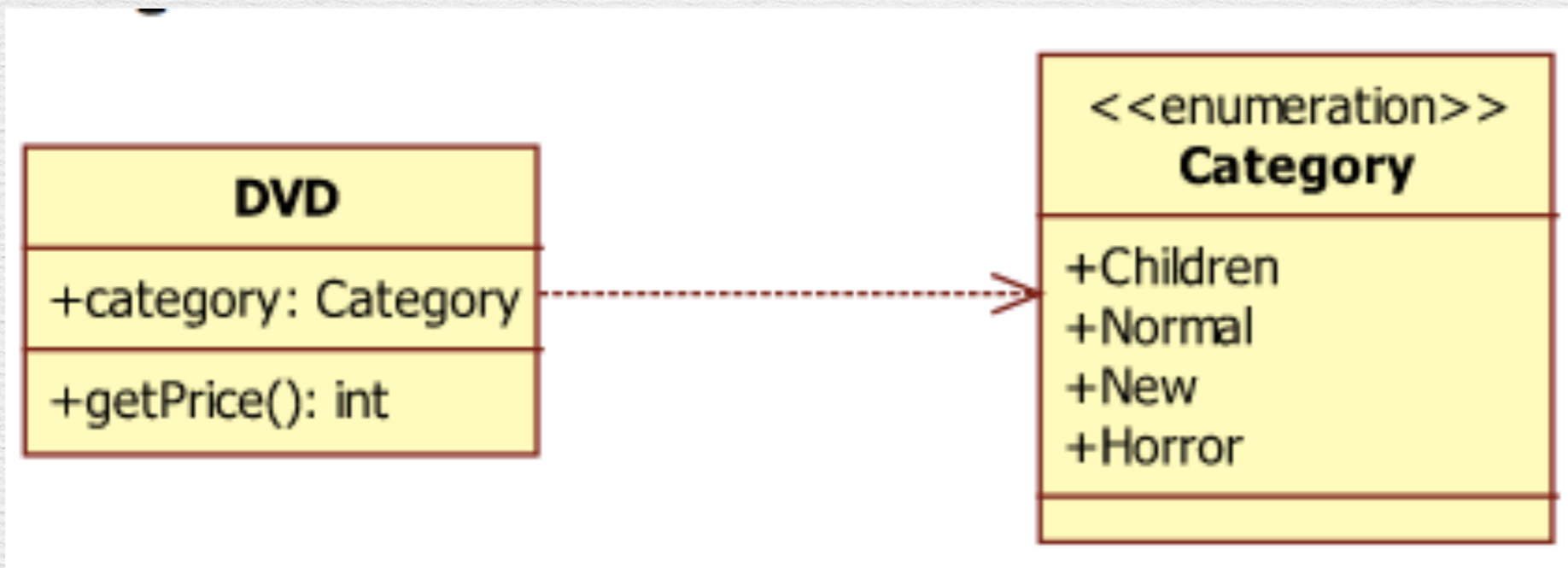
✓ 1.2 *L'ajout ou la suppression d'un état n'implique pas de modification de code*

2. Gestion simplifiée

✗ 2.1 *Changement d'état possible à l'exécution sans destruction*

✗ 2.2 *Découplage du comportement de chaque état*

Pattern Etat Abimé



1. Extensibilité

✘ 1.1 *Factorisation de protocole pour tous les états*

✘ 1.2 *L'ajout ou la suppression d'un état n'implique pas de modification de code*

2. Gestion simplifiée

✔ 2.1 *Changement d'état possible à l'exécution sans destruction*

✘ 2.2 *Découplage du comportement de chaque état*