Trucs et astuces pour retrouver certains patterns dans du code

Christophe Damas

Decorator

- Le pattern decorator est présent si une classe implémente une interface et a un attribut de cette même interface

```
public class StarMario implements IMario {
  private IMario decoratedMario;
  private int timer = 1000;
```

```
public class Captain implements BattleShip {
  private BattleShip battleship;
  public Captain() {}
```

```
public abstract class Additif implements Boisson {
  private Boisson boisson;
  public Additif(Boisson boisson) {
     if (boisson == null) throw new IllegalArgumentException();
     this.boisson = boisson;
  }
```

new(new(new)))

- Le pattern decorator est facilement identifiable car l'instanciation se fait via un chainage de new()

```
new Lait(new Sucre(new Decafeine()));
```

- Attention, le pattern Chain Of Responsibility est aussi instancié similairement mais se termine par un null:

```
new ValidationVisa(new ValidationMastercard(new ValidationDiscover(
   new ValidationDinersClub(new ValidationAmEx(null)))))
```

Composite

- Le pattern composite est présent si une classe implémente une interface et a un attribut qui est une collection sur cette interface

```
public class CompoundShape implements Shape {
  private List<Shape> shapes= new ArrayList<Shape>();
```

```
public class Et implements Strategy{

  private Strategy[] strats;
```

Template method

- Le pattern template method est présent s'il y a une classe abstraite contenant une méthode concrète et une méthode abstraite, et la méthode concrète appelle la méthode abstraite

```
public abstract class ListePersonnes {
  private Personne[] personnes;
  public ListePersonnes(Personne... personnes) {
    this.personnes = personnes;}
  public void tri() {
    int cpt; Personne element;
    for (int i = 1; i < personnes.length; i++) {
      element = personnes[i]; cpt = i - 1;
      while (cpt >= 0 &&
          this.compare(personnes[cpt],element)) {
        personnes[cpt + 1] = personnes[cpt]; cpt--;}
      personnes[cpt + 1] = element;}
    for (Personne pers : personnes)
      System.out.println(pers);}

  public abstract boolean compare(Personne personne, Personne element);
}
```

```
public abstract class DwarvenMineWorker {
  public abstract void work();
  private void action(Action action) {
    switch (action) {
      case GO_TO_SLEEP: goToSleep(); break;
      case WAKE_UP: wakeUp(); break;
      case GO_HOME: goHome(); break;
      case GO_TO_MINE: goToMine(); break;
      case WORK: work(); break;
      default: System.out.println("Action non definie"); break;}
    }
```

Factory method

- Cas particulier de Template method où la méthode abstraite fait un new

```
public abstract class Magasin {
  private Map<String, Product> bac = new HashMap<String, Product>();
  public void ajouterProduit(String name, int anneeDeParution) {
    Product produit = createProduit(name, anneeDeParution);
    bac.put(name, produit);}
  public abstract Product createProduit(String name, int anneeDeParution);
… }

public class MagasinDeDVD extends Magasin {
  @Override
  public Product createProduit(String name, int anneeDeParution) {
    return new DVD(name, anneeDeParution);}
}
```

Facade

- Le pattern facade est présent si le projet contient plusieurs classes et que le main ne fait appel qu'à une seule classe

```
public class Main {
  public static void main(String[] args) {
    DwarvenGoldmine jjGoldmine = new DwarvenGoldmine();
    jjGoldmine.startNewDay();
    jjGoldmine.digOutGold();
    jjGoldmine.endDay();}
}
```

Difference Strategy/Visitor

- Le pattern visitor ressemble à une Strategy avec autant de méthodes que de type de nœuds différents.
- En général, les strategy ont une seule méthode.

```
public interface ComparatorStrategy{
  public boolean compare(Personne p1,Personne p2);
}
```

```
public interface Distance {
  String getDistance();
}
```

```
public interface Visitor {
  String visitCircle(Circle circle);
  String visitSquare(Square s);
  String visitCompoundShape(CompoundShape cs);}
```

```
public interface Traitement {
  void traiteValeur(Valeur unique);
  void traiteGroupe(Groupe plusieurs);
}
```

State

- Le pattern state est identifiable facilement car les méthodes délèguent leur comportement aux états

```
public class Cellule {
  private int ligne, colonne;
  private Situation situation;
  public Cellule(int ligne, int colonne) {
    this.ligne = ligne; this.colonne = colonne;
    this.situation = EstMorte.getInstance();}
  public void vit() {situation = situation.vit();}
  public void meurt() {situation = situation.meurt();}
  public void toggle() {situation = situation.toggle();}
```

```
public class MachineACafe {
  private int montantEnCours = 0;
  private State etatCourant;
  private ToucheBoisson boisson = null;
  public MachineACafe(){ etatCourant = State.IdleState;}
  public void rendreMonnaie() {
    etatCourant.rendreMonnaie(this);}
  public void entrerMonnaie(Piece piece) {
    etatCourant.entrerMonnaie(piece, this);}
  public void selectionnerBoisson(ToucheBoisson toucheBoisson) {
    etatCourant.selectionnerBoisson(toucheBoisson, this);}
```

Attention à ne pas confondre avec un decorateur!

```
public class StarMario implements IMario {
  private IMario decoratedMario;
  private int timer = 1000;

  public StarMario(IMario decoratedMario) {
    this.decoratedMario = decoratedMario;}

  @Override
  public void takeDamage() {
    // StarMario does not take damage}

  @Override
  public void jump() {
    decoratedMario.jump();}

  @Override
  public void moveForward() {
    decoratedMario.moveForward();}

  @Override
  public void update() {
    timer--;
    if (timer == 0) {
      removeStar();}
    decoratedMario.update();}

  public void removeStar() {
    GameSingleton.instance.mario =
                                    decoratedMario}
}
```