04 - Decorator, Abstract Factory

Credits: Askar Gafurov

Dekorator

Pouzivame, ked chceme moct dynamicky (t.j. pocas behu programu) menit spravanie objektu viacerymi sposobmi, ktore su navzajom nevylucujuce, pricom chceme zachovat povodne rozhranie objektu.

Zakladna struktura:

```java
public interface Component {
    void operation1();
    void operation2();
}

public class ConcreteComponent implements Component {
    @Override
    public void operation1() {...}

    @Override
    public void operation2() {...}
}

// may also be `abstract`
// also, notice no references to concrete classes
// `implements Component` in order to preserve the original
// interface of an object
public class Decorator implements Component {
    // private, therefore not available to the concrete decorators
    private final Component component;

    public Decorator(Component component) {
        // store the decorated component
        this.component = component;
    }

    // reconnect with all public functions from the interface

    @Override
    public void operation1() {
        component.operation1();
    }

    @Override
    public void operation2() {
        component.operation2();
    }
}

public class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component component) {
        super(component);
    }

    // suppose we need to alter `operation1()` method
    @Override
    public void operation1() {
        // do something else
        super.operation1(); // the original method
                            // is called through `super`
    }

    // if we don't need to alter other operations, we can omit the rest
}
```

Pouzitie:

```
class Demo {
    void demo() {
        Component component = new ConcreteComponent();
        component.operation1(); // usage before decoration
        Component dComponent = new ConcreteDecorator(component);
        dComponent.operation1(); // usage after decoration,
                                 // nothing changed

        // we can also add more decorators on the same object
        Component d2Component = new ConcreteDecorator(dComponent);

        // or add another decorator on top
        Component d3Component = new ConcreteDecorator2(d2Component);
        d3Component.operation2(); // usage doesn’t change
    }
}
```

Ulohy:

U: Implementujte rozhranie `Coffee` s dvomi metodami:

- `double getCost()`
- `List<String> getIngredients()`

U: Implementujte triedu `Espresso` s rozhranim `Coffee`, ktore stoji 1 euro a jedina ingrediencia je “Espresso”.

U: Implementujte dekoratory `Water`, `Milk`, `MilkFoam`, `WhippedCream` a `Whiskey`, ktore budu zvysovat cenu respektivne o 0, 0.3, 0.5, 1 a 2 eura a budu rozsirovat zoznam ingrediencii.

U: Vyskusajte vytvorit nasledovne napoje:

1. Espresso
2. Americano = Espresso + Water
3. Latte = Espresso + Milk + Milk foam
4. Vienna coffee = Espresso + Whipped cream
5. Irish coffee = Espresso + Whiskey + Whipped cream

U*: Vytvorte a vypiste ceny a zlozenia tychto napojov cez for-cyklus vyuzitim struktury `LinkedHashMap<String, Coffee>` (`Map.Entry<String, Coffee>`).

Factory

Pouzivame, ked chceme schovat proces vytvarania objektov pred klientom a zaroven mozeme vytvarat objekty roznych tried so spolocnym rozhranim.

Zakladna struktura:

```
public interface Component {
    void operation();
}

class ConcreteComponentA implements Component {...}

class ConcreteComponentB implements Component {...}

class ConcreteComponentC implements Component {...}

public class ComponentFactory {
    public Component createComponent(String componentType) {
        if (componentType.equals("A")) {
            // maybe also do something interesting with the object
            return new ConcreteComponentA();
        }
        else if (componentType.equals("B")) {
            return new ConcreteComponentB();
        }
        else if (componentType.equals("C")) {
            return new ConcreteComponentC();
        }
        return null;
    }
}
```

Pouzitie:

```
class Demo {
    void demo() {
        ComponentFactory factory = new ComponentFactory();
        Component cA = factory.createComponent("A");
        Component cB = factory.createComponent("B");
        cA.operation();
        cB.operation();
    }
}
```

Ulohy:

U: Vytvorte rozhranie `Shelf` s metodami:

- `Coffee createEspresso()`
- `Coffee addMilk(Coffee basis)`
- `Coffee addMilkFoam(Coffee basis)`
- `Coffee addWater(Coffee basis)`
- `Coffee addWhiskey(Coffee basis)`

U: Vytvorte triedu `OrdinaryShelf` s rozhranim `Shelf`, ktora bude pouzivat doteraz vytvorene objekty.

U: Vytvorte triedu `Barista` s verejnymi metodami:

- `Barista(Shelf shelf)` - konstruktor, ktory na vstupe dostane tovaren na ingrediencie
- `Coffee getCoffee(String name)`, ktora dostane na vstupe nazov napoja (podla popisu z predoslej sekcie) a vrati pripraveny objekt, pricom objekty bude vytvarat pomocou objektu rozhrania `Shelf`, ktory dostane pri konstruovani.

Trieda `Barista` je teda “netrivialna” **tovaren**, a trieda `OrdinaryShelf` je “nudna” **tovaren**. V danom pripade sa na triedu `OrdinaryShelf` taktiez mozeme divat ako na konkretnu **strategiu**.

U*: Pridajte dekorator `Logger` pre rozhranie `Shelf`, ktory pri kazdom volani metod bude logovat pouzite ingrediencie do suboru, nazov ktoreho dostane pri konstruovani. Vyskusajte ho.

Abstract Factory

Pouzivame vtedy, ked mame viacere sady objektov, ktore treba pouzivat v spolocnom konktexte (napriklad objekty pre rozne operacne systemy alebo rozne typy vypoctov (CPU/parallel CPU/GPU) a chceme mat jednotny aplikacny kod pre vsetky kontexty.

Zakladna struktura:

```
public interface ComponentA {
    void foo();
}

public interface ComponentB {
    void bar();
}

class BlueComponentA implements ComponentA {...}
class BlueComponentB implements ComponentB {...}

class RedComponentA implements ComponentA {...}
class RedComponentB implements ComponentB {...}

// this is the abstract factory (may also be an abstract class)
public interface ComponentFactory {
    ComponentA createComponentA();
    ComponentB createComponentB();
}

public class BlueComponentFactory implements ComponentFactory {
    @Override
    public ComponentA createComponentA() {
        return new BlueComponentA();
    }

    @Override
    public ComponentB createComponentB() {
        return new BlueComponentB();
    }
}

public class RedComponentFactory implements ComponentFactory {
    @Override
    public ComponentA createComponentA() {
        return new RedComponentA();
    }

    @Override
    public ComponentB createComponentB() {
        return new RedComponentB();
    }
}
```

Pouzitie:

```
class Demo {
    void demo() {
        String context = "red"; // maybe OS name or live input

        // the creation of a factory could also be encapsulated into
        // a separate FactoryProducer class
        ComponentFactory factory;
        if (context.equals("red")) {
            factory = new RedComponentFactory();
        }
        else if (context.equals("blue")) {
            factory = new BlueComponentFactory();
        }
        else {
            factory = null; // or raise an error
        }

        // from this point on, only interfaces are used, therefore
        // this same code could be used in any context
        ComponentA cA = factory.createComponentA();
        ComponentB cB = factory.createComponentB();
        cA.foo();
        cB.bar();
    }
}
```

Ulohy:

U: Vytvorte triedy `PremiumEspresso`, `PremiumWater`..., ktore sa budu lisit tym, ze budu mat ovela vyssiu cenu (a teda trochu ine nazvy ingrediencii).

U: Vytvorte triedu `PremiumShelf` nad rozhranim `Shelf`, ktora bude pouzivat tieto “premiove” triedy.

U: Vyskusajte triedu `Barista` s triedou `PremiumShelf`. V danom pripade teda je rozhranie `Shelf` **abstraktnou tovarnou**, a triedy `OrdinaryShelf` a `PremiumShelf` su konkretnymi tovarnami. Pre abstraktnu tovaren `Shelf` je trieda `Barista` **klientom**.