# SOFTVERSKO INŽENJERSTVO

## školska 2024/2025 godina

## Vežba 13: Decorator Pattern

Decorator pattern pripada strukturnim (structural) dizajn šablonima.

Njegova osnovna svrha je dinamičko dodavanje novih funkcionalnosti objektu bez menjanja njegove osnovne klase ili drugih objekata.

Decorator omogućava da se objekti "obuče" u dodatne funkcionalnosti u toku izvršavanja programa, pri čemu se zadržava originalni interfejs. Ovim se postiže fleksibilnost i modularnost bez nasleđivanja i bez komplikovanih hijerarhija klasa.

Pored toga, decorator pattern omogućava kombinovanje različitih funkcionalnosti na jednostavan način, jer se dekoratori mogu slagati jedan na drugi. Ovo čini kod lakšim za održavanje i proširivanje, jer se nove funkcionalnosti mogu dodavati bez menjanja postojećeg koda.

---

### 📌 Problem koji Decorator rešava

Zamislimo da pravimo sistem za obradu tekstualnih poruka (npr. chat aplikaciju). Poruka može imati osnovni tekst, ali korisnik može da želi da joj doda neke opcije:

- Poruku treba formatirati kao **bold**
- Poruku treba **obojiti u crveno**
- Poruku treba **dodati podvučen tekst**
- Ili kombinaciju svih ovih efekata

Bez Decorator paterna, morali bismo da kreiramo posebne klase za svaku kombinaciju (npr. BoldRedMessage, BoldUnderlineMessage itd), što vodi u eksploziju klasa i duplikaciju koda.

### ✅ Rešenje: Decorator patern

Decorator omogućava:

- Da dodatne funkcionalnosti dodajemo objekatima dinamički (u toku rada programa)
- Da se ponašanje objekta širi bez menjanja postojeće klase
- Da se dekoratori mogu kombinovati jedan na drugi i da svi implementiraju isti interfejs kao i originalni objekat.

---

### 🧱 Struktura Decorator paterna

Struktura Decorator paterna jasno definiše uloge pojedinačnih elemenata i način na koji oni međusobno sarađuju kako bi se omogućilo dinamičko proširenje funkcionalnosti objekata bez izmene njihovih osnovnih klasa.

| Element | Opis |
|---|---|
| Component | Interfejs koji definiše osnovne operacije (npr. metoda operation()) |
| ConcreteComponent | Originalna klasa koja implementira Component interfejs (osnovni objekat) |
| Decorator | Apstraktna klasa koja implementira Component i poseduje referencu na Component objekat |
| ConcreteDecorator | Konkretna klasa koja nasledjuje Decorator i dodaje novu funkcionalnost |

### ✅ Prednosti Decorator paterna

- Omogućava fleksibilno proširivanje funkcionalnosti objekata u toku izvršavanja
- Smanjuje potrebu za velikim brojem podklasa (nasleđivanja)
- Poštuje princip otvorenosti-zatvorenosti (Open-Closed Principle)
- Dekoratori mogu biti višestruko složeni i kombinovani
- Razdvaja osnovnu logiku objekta od dodatnih funkcionalnosti

### 📍 Gde se koristi Decorator?

- **GUI frameworki** — često se koristi za dodavanje vizuelnih efekata i stilova komponentama, kao što su skrol barovi, ivice, senke ili animacije, bez menjanja same komponente.
- **Stream biblioteke** — prilikom čitanja i pisanja fajlova, dekoratori omogućavaju dodavanje funkcionalnosti poput filtriranja podataka, enkripcije ili kompresije, slojevito i dinamički.
- **Logovanje i merenje performansi** — u runtime-u se može dodavati beleženje aktivnosti ili merenje vremena izvršavanja metoda, što pomaže u praćenju i optimizaciji aplikacije.
- **Formatiranje i dekoracija podataka** — kod obrade teksta, slika ili drugih medija, dekoratori omogućavaju primenu različitih stilova ili formata, kao što su podebljavanje teksta, dodavanje okvira ili promene boja.
- **Složeni sistemi** — omogućava proširenje ponašanja objekata u velikim aplikacijama bez potrebe za menjanje osnovnih klasa, čime se povećava fleksibilnost i održivost sistema.

---

### 🔑 Ključne osobine

- **Isti interfejs** — i dekorator i originalni objekat implementiraju isti interfejs, što omogućava da dekoratori budu zamenjivi sa osnovnim objektom.
- **Delegiranje poziva** — dekorator poseduje referencu na objekat koji dekorira i prosleđuje mu pozive metoda, uz mogućnost da pre ili posle toga izvrši još radnji.
- **Dodavanje ponašanja** — dekorator može dodavati funkcionalnost pre ili posle poziva originalnog objekta, što omogućava fleksibilno proširenje ponašanja.
- **Složena kombinacija** — moguće je kombinovati više dekoratora jedan na drugi, stvarajući slojevitu strukturu koja se može lako menjati i proširivati.

---

### ⚠️ Izazovi i potencijalne zamke

- **Složenost koda** — preterana upotreba dekoratora može dovesti do zagušenja koda i otežanog razumevanja toka izvršavanja, jer se pozivi prenose kroz više slojeva.
- **Teže debagovanje** — zbog višestrukih slojeva i delegiranja, pronalaženje uzroka grešaka može biti komplikovanije.
- **Pažljiv dizajn** — neophodno je dobro planiranje strukture kako bi se izbegla redundantnost i nepotrebno preklapanje funkcionalnosti među dekoratorima.

### 🎯 Scenario primera

Zamislimo jednostavan sistem za prikazivanje tekstualnih poruka u korisničkom interfejsu, kao što su notifikacije, poruke u četu ili statusne informacije. Želimo da korisnik može videti osnovne poruke, ali i da po potrebi dodamo stilove — **bez menjanja osnovne logike** poruke.

Decorator pattern je savršen za ovakve situacije jer omogućava da fleksibilno dodajemo dodatne efekte kao što su **boldovanje** ili **promena boje** teksta bez potrebe da stalno pravimo nove podklase za svaku kombinaciju (npr. BoldGreenMessage, ItalicRedMessage itd.).

U ovom primeru, poruka može imati sledeće varijante:

- Obični (plain) tekst
- Tekst prikazan u **bold** stilu
- Tekst prikazan u **određenoj boji** (npr. crveno, zeleno...)
- Kombinacije stilova (npr. boldovani tekst u plavoj boji)

---

### 📁 Struktura primera

message/

├ Message.java

├ PlainMessage.java

├ MessageDecorator.java

├ BoldDecorator.java

├ ColorDecorator.java

└ Main.java

---

### ✅ 1. Message.java – interfejs komponente

Ovo je osnovni interfejs koji definiše zajednički metod za sve poruke i dekoratore.

```
public interface Message {
    String getContent();
}
```

### ✅ 2. PlainMessage.java – konkretna komponenta

Osnovni tekst poruke bez dodatnih stilova.

```
public class PlainMessage implements Message {
    private String text;

    public PlainMessage(String text) {
        this.text = text;
    }

    @Override
    public String getContent() {
        return text;
    }
}
```

---

### ✅ 3. MessageDecorator.java – apstraktni dekorator

Sadrži referencu na Message objekat i delegira pozive, služi kao osnov za sve dekoratore.

```
public abstract class MessageDecorator implements Message {
    protected Message message;

    public MessageDecorator(Message message) {
        this.message = message;
    }

    @Override
    public String getContent() {
        return message.getContent();
    }
}
```

### ✅ 4. BoldDecorator.java – konkretni dekorator

Dodaje bold stil tekstu.

```
public class BoldDecorator extends MessageDecorator {

    public BoldDecorator(Message message) {
        super(message);
    }

    @Override
    public String getContent() {
        return "<b>" + super.getContent() + "</b>";
    }
}
```

---

### ✅ 5. ColorDecorator.java – konkretni dekorator

Dodaje boju tekstu (npr. crvenu).

```
public class ColorDecorator extends MessageDecorator {
    private String color;

    public ColorDecorator(Message message, String color) {
        super(message);
        this.color = color;
    }

    @Override
    public String getContent() {
        return "<span style='color:" + color + "'>" +
               super.getContent() + "</span>";
    }
}
```

## ✅ 6. Main.java (glavni program)

```
public class Main {
    public static void main(String[] args) {
        Message msg = new PlainMessage("Pozdrav, studenti!");

        // Običan tekst
        System.out.println(msg.getContent());

        // Dodajemo bold stil
        msg = new BoldDecorator(msg);
        System.out.println(msg.getContent());

        // Dodajemo crvenu boju na bold tekst
        msg = new ColorDecorator(msg, "red");
        System.out.println(msg.getContent());

        // Kombinacija: bold + plava boja
        Message msg2 = new ColorDecorator(
                            new BoldDecorator(
                                new PlainMessage("Drugi primer")),
                                "blue");
        System.out.println(msg2.getContent());

        // Dodatni primer: samo boja (bez bolda)
        Message msg3 = new ColorDecorator(
                            new PlainMessage("Samo obojen tekst"),
                        "green");
        System.out.println(msg3.getContent());
```

```
        // Još jedan primer: višestruka dekoracija
          (bold + crvena + dodatni bold)
        Message msg4 = new BoldDecorator(
                            new ColorDecorator(
                                new BoldDecorator(
                                    new PlainMessage("Složen primer")),
                                "red"));
        System.out.println(msg4.getContent());
    }
}
```

**Rezultat izvršavanja:**

Pozdrav, studenti!

Pozdrav, studenti!

Pozdrav, studenti!

Drugi primer

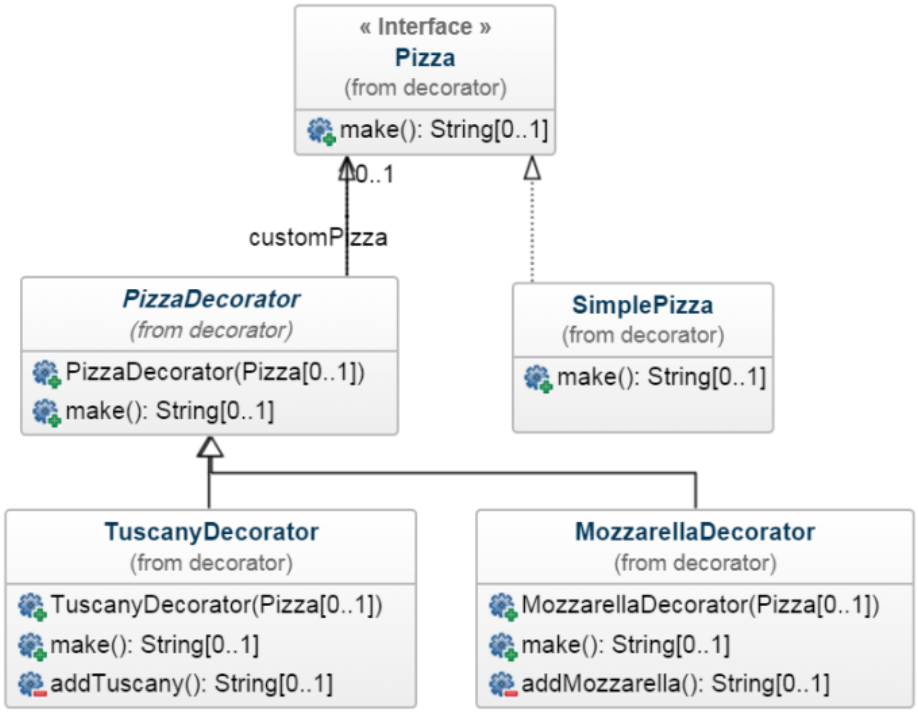
Samo obojen tekst

Složen primer

## 🧠 Zaključak

- Decorator omogućava dodavanje novih funkcionalnosti bez menjanja originalnog objekta
- Funkcionalnosti se mogu dinamički slagati i kombinovati
- Pomaže da sistem ostane otvoren za proširenja, a zatvoren za modifikacije
- Primer je korisno primeniti tamo gde se želi fleksibilnost i modularnost u dodavanju osobina

Decorator obrazac podstiče princip **otvorenosti za proširenje, ali zatvorenosti za izmenu (Open/Closed Principle)** – što je jedan od temeljnih principa objektno-orijentisanog dizajna. Time se omogućava razvoj održivog i skalabilnog softverskog rešenja.




Primer UML klasnog dijagrama za Decorator pattern

---

🛒 **Zadatak za samostalni rad: Online prodavnica sa dodacima (Decorator pattern)**

Zamislite da pravite sistem za **online prodavnicu** gde korisnik može da kupi **osnovni proizvod** (npr. laptop). Pored osnovnog proizvoda, korisnik ima mogućnost da **dodaje dodatke po izboru** (npr. torbu, antivirus, dodatni punjač).

Ovi dodaci **ne menjaju klasu osnovnog proizvoda**, već se **dinamički dodaju** koristeći **Decorator dizajn šablon**.

---

✅ **Cilj implementacije:**

Kreirati strukturu koja omogućava da korisnik:

- Odabere osnovni proizvod
- Po želji doda više dodataka
- Dobije **ukupnu cenu** i **kombinovani opis** proizvoda sa svim dodacima

📦 **Funkcionalnosti koje treba implementirati:**

**1. Component interfejs (Product.java)**

Definiše metode:

double getPrice();

String getDescription();

**2. ConcreteComponent (BasicLaptop.java)**

Predstavlja osnovni proizvod, npr. laptop:

getPrice() -> 800.0

getDescription() -> "Osnovni laptop"

**3. Decorator klasa (ProductDecorator.java)**

- Apstraktna klasa koja implementira Product
- Ima referencu na drugi Product objekat
- Prosleđuje pozive getPrice() i getDescription()

**4. ConcreteDecorator klase**

Svaka klasa dodaje dodatnu funkcionalnost:

| **Klasa** | **Cena** | **Dodatak opisu** |
|---|---|---|
| WarrantyDecorator | +100.0 | „+ Produžena garancija“ |
| BagDecorator | +40.0 | „+ Torba za laptop“ |
| AntivirusDecorator | +50.0 | „+ Antivirus softver“ |
| ChargerDecorator | +30.0 | „+ Dodatni punjač“ |