SOFTVERSKO INŽENJERSTVO

školska 2024/2025 godina

Vežba 8: Uvod u dizajn paterne, Singleton patern

U **softverskom inženjerstvu**, veoma je važno da softver bude razvijen na način koji omogućava da bude:

- **jasan** – kod treba da bude čitljiv i razumljiv ne samo autoru već i drugim članovima tima,
- **ponovno upotrebljiv** – komponente softvera treba da se mogu koristiti i u drugim projektima bez prepravljanja,
- **održiv i proširiv** – lako se dodaju nove funkcionalnosti, ispravljaju greške i poboljšava performansa bez narušavanja postojećeg koda.

Jedan od ključnih alata koji nam pomažu da postignemo ove ciljeve su **obrasci dizajna** (engl. *design patterns*).

❓ Šta su obrasci dizajna?

Obrasci dizajna su **opšta, ponovo upotrebljiva rešenja za česte probleme** koji se javljaju prilikom projektovanja softverskih sistema. Oni ne predstavljaju gotov programski kod, već:

- **opis arhitektonske ideje** kako rešiti neki konkretan problem,
- **šablon** koji se može prilagoditi konkretnim potrebama projekta.

Učenje i primena obrazaca dizajna pomaže programerima da:

- izbegnu ponavljanje grešaka,
- koriste već proveren pristup rešavanju problema,
- razvijaju softver koji je strukturiran, efikasan i razumljiv.

📂 Glavne kategorije obrazaca dizajna

Obrasci dizajna su najčešće podeljeni u **tri glavne kategorije**, u zavisnosti od problema koji rešavaju:

1. Kreacioni obrasci (Creational Patterns)

Ovi obrasci se bave **načinom kreiranja objekata**. Cilj im je da izoluju proces instanciranja objekata kako bi aplikacija bila nezavisna od konkretnih klasa koje koristi.

- **Singleton** – obezbeđuje da postoji samo jedna instanca klase u aplikaciji.
- **Factory Method** – definiše interfejs za kreiranje objekta, ali dopušta podklasama da odluče koji objekat će biti kreiran.
- **Abstract Factory** – kreira porodice povezanih objekata bez preciziranja konkretnih klasa.
- **Builder** – razdvaja konstrukciju kompleksnog objekta od njegove reprezentacije.
- **Prototype** – kreira nove objekte kopiranjem postojećih (kloniranjem).

Kada koristiti: Kada ne želimo da klasa direktno zavisi od konkretne implementacije objekta koji koristi, ili kada je proces kreiranja kompleksan i zavisi od više parametara.

2. Strukturni obrasci (Structural Patterns)

Ovi obrasci se fokusiraju na **sastavljanje klasa i objekata u veće strukture** dok zadržavaju fleksibilnost i jednostavnost.

- **Adapter** – omogućava interakciju između klasa koje inače ne bi mogle da rade zajedno zbog različitih interfejsa.
- **Decorator** – omogućava dinamičko dodavanje novih ponašanja objektu bez menjanja njegove osnovne strukture.
- **Facade** – pruža jednostavan interfejs za kompleksan podsistem.
- **Proxy** – kontroliše pristup drugom objektu, može se koristiti za keširanje, logovanje...
- **Composite** – omogućava tretiranje pojedinačnih objekata i kolekcija objekata na isti način (npr. hijerarhije).
- **Bridge** – odvaja apstrakciju od implementacije kako bi se obe mogle menjati nezavisno.

Kada koristiti: Kada želimo da pojednostavimo interakciju sa kompleksnim sistemima, da sakrijemo implementacione detalje ili omogućimo zamenu komponenti bez uticaja na druge delove sistema.

3. Ponašajni obrasci (Behavioral Patterns)

Ovi obrasci se fokusiraju na **interakciju između objekata** i **odgovornost za obavljanje zadataka**.

- **Observer** – omogućava da objekti budu obavešteni o promenama u drugim objektima (npr. UI osvežavanje kada se podaci promene).
- **Strategy** – omogućava izbor algoritma u toku izvršavanja bez promene koda koji koristi algoritam.
- **State** – omogućava objektu da menja svoje ponašanje kada se promeni njegovo stanje.
- **Command** – enkapsulira zahtev kao objekat, omogućava logovanje, poništavanje, itd.
- **Iterator** – omogućava pristup elementima kolekcije bez otkrivanja njene implementacije.
- **Mediator** – omogućava komunikaciju između objekata bez direktne reference među njima (smanjuje povezanost).
- **Chain of Responsibility** – omogućava više objekata da obrade zahtev bez eksplicitnog pozivanja svakog ponaosob.

Kada koristiti: Kada želimo fleksibilnu komunikaciju između objekata, zamenu algoritama u hodu, ili upravljanje događajima bez uske zavisnosti među komponentama.

✅ **Zašto je važno poznavati obrasce dizajna?**

- **Standardizacija** – Obrasci omogućavaju timovima da koriste zajednički rečnik. Kada svi razumeju pojmove poput „Singleton“ ili „Observer“, komunikacija postaje brža i jasnija.
- **Efikasnost** – Umesto da svaki put rešavamo problem iznova, možemo primeniti proverene obrasce koji su već dokazani u praksi. Time štedimo vreme i smanjujemo broj grešaka.
- **Kvalitet** – Kod zasnovan na obrascima je organizovaniji, lakši za testiranje i održavanje, što je posebno važno kod većih i dugoročnih projekata.
- **Fleksibilnost** – Obrasci podstiču modularni dizajn. To znači da možemo menjati ili unapređivati delove sistema bez velikih promena u ostatku koda.
- **Bolje razumevanje arhitekture** – Kroz obrasce programeri uče kako da grade jasne, održive i proširive softverske sisteme.

🔶 Design Pattern Singleton

Singleton je obrazac dizajna koji osigurava da klasa ima **samo jednu instancu (objekat)** i pruža **globalnu tačku pristupa** toj instanci.

To je kao da postoji **samo jedan printer u kancelariji**, i svi računari (objekti) u mreži moraju da koriste **taj isti printer** – tj. jednu jedinu instancu klase.

🎯 Osnovna ideja Singleton-a:

- Ne dozvoliti da se klasa instancira više puta
- Sačuvati jedinu instancu unutar same klase
- Dozvoliti pristup toj instanci preko metode getInstance()

📌 Kada koristiti Singleton?

- Kada je potrebno **centralizovano upravljanje** (npr. pristup konfiguraciji, logovanju, kešu).
- Kada je **potrebna samo jedna instanca objekta** (npr. konekcija ka bazi podataka).
- Kada više delova aplikacije treba da **dele iste podatke ili stanje**.

⚙️ Struktura Singleton klase:

1. **Privatni konstruktor** – da drugi delovi programa ne mogu napraviti novu instancu pomoću new.
2. **Privatna statička promenljiva** – da čuva jedinu instancu klase.
3. **Javna statička metoda** – getInstance() vraća jedinstvenu instancu.

🧪 Prednosti:

- Kontrolisana upotreba resursa (npr. konekcija ka bazi)
- Globalna pristupna tačka
- Laka primena

⚠️ Nedostaci:

- Može biti teško testirati (jer uvodi globalno stanje)
- Prekomerna upotreba može napraviti "bog-objekat" (objekat koji zna previše)

✅ Implementacija u Javi:

```
public class Logger {

    // 1. Privatna statička instanca
    private static Logger instance;

    // 2. Privatni konstruktor – sprečava instanciranje spolja
    private Logger() {
        System.out.println("Logger kreiran!");
    }

    // 3. Javni metod za pristup instanci
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger(); // instancira se samo prvi put
        }
        return instance;
    }

    // Primer metode
    public void log(String poruka) {
        System.out.println("[LOG]: " + poruka);
    }
}
```

🧪 Korišćenje Singleton klase:

```
public class Main {
    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("Ovo je prva poruka.");
        logger2.log("Ovo je druga poruka.");

        // Provera da su obe promenljive ista instanca
        if (logger1 == logger2) {
            System.out.println("Oba logger-a su ista instanca (singleton).");
        }
    }
}
```

🟡 Rezultat:

```
Logger kreiran!
[LOG]: Ovo je prva poruka.
[LOG]: Ovo je druga poruka.
Oba logger-a su ista instanca (singleton).
```

Napomena: Konstruktor se poziva samo **jednom**, bez obzira koliko puta zovemo `getInstance()`.

📘 Objašnjenje korak po korak:

Korak	Šta se dešava	Zašto je bitno
private static Logger instance	čuva jedinu instancu	omogućava globalni pristup
private Logger()	konstruktor je sakriven	sprečava direktno kreiranje objekta
public static Logger getInstance()	pristup kroz statičku metodu	pravi instancu ako još ne postoji
logger1 == logger2	poređenje instanci	potvrđuje da je u pitanju Singleton

❓ Zadatak

Napraviti main program za klasu AppConfig koja implementira Singleton obrazac i sadrži dve konfiguracione vrednosti: appName i version. Metoda printConfig() ispisuje te vrednosti.

```
public class AppConfig {

    // Privatna statička instanca
    private static AppConfig instance;

    // Konfiguracije
    private String appName;
    private String version;

    // Privatni konstruktor
    private AppConfig() {
        appName = "Moja Aplikacija";
        version = "1.0.0";
    }

    // Pristup Singleton instanci
    public static AppConfig getInstance() {
        if (instance == null) {
            instance = new AppConfig();
        }
        return instance;
    }

    // Metod za prikaz konfiguracije
    public void printConfig() {
        System.out.println("Naziv aplikacije: " + appName);
        System.out.println("Verzija: " + version);
    }
}
```