SOFTVERSKO INŽENJERSTVO

školska 2024/2025 godina

Vežba 11: Observer Pattern

Observer patern pripada **behavioralnim (ponašajnim)** dizajn paterima. Njegova osnovna svrha je da obezbedi **mehanizam za automatsko obaveštavanje više objekata** kada se stanje nekog objekta promeni.

Koristan je kada postoji **jedan objekat koji menja stanje (subject)**, a **više objekata treba da reaguje** na tu promenu bez međusobne direktne povezanosti.

📌 **Najviše dolazi do izražaja u aplikacijama gde je važno reagovati na događaje u realnom vremenu** — npr. korisnički interfejsi, sistemi notifikacija, igre, IoT uređaji i finansijske aplikacije. U tim slučajevima, Observer omogućava da se sistem lako proširuje dodavanjem novih reakcija na događaje, bez narušavanja postojećeg koda.

🔍 Problem koji Observer rešava

Zamislimo vremensku stanicu (WeatherStation) koja beleži temperaturu. Želimo da više komponenti reaguje kada se temperatura promeni:

- Prikaz na sajtu
- Mobilna aplikacija
- Digitalni displej u gradu

Bez Observer paterna, morali bismo ručno pozivati metode u svakoj komponenti kada se promeni temperatura:

```
weatherStation.updateTemperature(25);
mobileApp.display(25);
cityDisplay.show(25);
website.update(25);
```

Ovakav pristup dovodi do snažne međuzavisnosti između klasa (tight coupling), što značajno otežava održavanje sistema, jer svaka promena u jednoj komponenti može zahtevati izmene u više drugih. Takođe, dodavanje novih funkcionalnosti postaje komplikovano i sklono greškama.

🧠 Rešenje: Observer

Observer patern omogućava:

- Da objekti registruju interesovanje za promene stanja (subscribe)
- Da **Subject automatski obavesti sve posmatrače (observers)** kada se stanje promeni
- Da se **dodavanje/uklanjanje novih posmatrača** vrši bez promene Subject klase

---

📐 Struktura Observer paterna

Element	Opis
Subject	Objekat koji sadrži stanje i obaveštava posmatrače
Observer	Interfejs koji sve posmatračke klase implementiraju
ConcreteSubject	Konkretna implementacija Subject-a (npr. WeatherStation)
ConcreteObserver	Konkretni posmatrači koji reaguju na promenu (npr. AppDisplay)
Client	Registruje posmatrače i menja stanje

✅ Prednosti Observer paterna

- ✔ Slaba povezanost između subjekta i posmatrača
- ✔ Jednostavno dodavanje novih posmatrača bez menjanja postojeće logike
- ✔ Automatsko obaveštavanje svih zainteresovanih strana
- ✔ Povećava fleksibilnost i proširivost sistema

🧰 Gde se koristi Observer?

- GUI biblioteke (npr. Swing ActionListener)
- Model-View-Controller (MVC) arhitektura
- Event-driven sistemi
- Notifikacioni sistemi (npr. kada se korisniku javi nova poruka)
- Sistemi za sinhronizaciju stanja
- Posmatranje fajl sistema, baza podataka.

💡 Ključne osobine

🔔 1. Dinamičko obaveštavanje

Subject ne zna konkretno ko ga posmatra – poziva samo update() metodu svih registrovanih posmatrača.

🧩 2. Fleksibilnost

Dodavanje novih prikaza (Observers) ne zahteva izmenu vremenske stanice.

🔁 3. Višestruki posmatrači

Više posmatrača može biti registrovano istovremeno i svi će biti obavešteni kada se stanje promeni.

💥 Izazovi i potencijalne zamke

Iako koristan, Observer patern može doneti i izazove:

- **Neočekivani efekti lančane reakcije** – Ako više posmatrača modifikuje stanje pri obaveštenju.
- **Teže debagovanje** – Povezanost između subjekta i posmatrača nije eksplicitna.
- **Curenje memorije** – Ako se posmatrači ne odjave pravilno (posebno u GUI aplikacijama).

Zato se preporučuje pažljivo upravljanje životnim ciklusom objekata i dokumentovanje zavisnosti.

🧪 Vežba: Vremenska stanica sa Observer paternom

Napravimo sistem gde WeatherStation obaveštava više prikaza kada se promeni temperatura.

1. Interfejs Observer

```
public interface Observer {
    void update(int temperature);
}
```

2. Interfejs Subject

```
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

3. Klasa WeatherStation (Subject)

```
import java.util.ArrayList;
import java.util.List;

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int temperature;

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }
```

```
    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(temperature);
        }
    }

    public void setTemperature(int temp) {
        this.temperature = temp;
        notifyObservers(); // automatski obavesti sve
    }
}
```

4. Prikazi – ConcreteObservers

AppDisplay

```
public class AppDisplay implements Observer {
    private String name;

    public AppDisplay(String name) {
        this.name = name;
    }

    @Override
    public void update(int temperature) {
        System.out.println(name + ": Prikaz temperature na aplikaciji - "
                           + temperature + "°C");
    }
}
```

CityDisplay

```
public class CityDisplay implements Observer {
    @Override
```

```
    public void update(int temperature) {
        System.out.println("📺 Displej u centru: Nova temperatura je " +
                           temperature + "°C");
    }
}
```

5. Glavni program

```
public class Main {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();

        Observer app1 = new AppDisplay("Mobilna aplikacija");
        Observer cityBoard = new CityDisplay();

        station.registerObserver(app1);
        station.registerObserver(cityBoard);

        System.out.println("🌤️ Menjanje temperature na 22°C...");
        station.setTemperature(22);

        System.out.println("\n🌡️ Menjanje temperature na 28°C...");
        station.setTemperature(28);
    }
}
```

Izlaz:

🌤️ Menjanje temperature na 22°C...

Mobilna aplikacija: Prikaz temperature na aplikaciji - 22°C

📺 Displej u centru: Nova temperatura je 22°C

🌡️ Menjanje temperature na 28°C...

Mobilna aplikacija: Prikaz temperature na aplikaciji - 28°C

📺 Displej u centru: Nova temperatura je 28°C