SOFTVERSKO INŽENJERSTVO

školska 2024/2025 godina

Vežba 12: Strategy Pattern

Strategy patern pripada **ponašajnim (behavioral)** dizajn šablonima.
Njegova osnovna svrha je da **omogući promenu ponašanja objekta u toku izvršavanja programa**, bez menjanja samog objekta.

Koristi se kada imamo **više algoritama ili ponašanja** za izvršavanje neke operacije, a želimo da omogućimo njihovu **zamenu u hodu** (dinamički), **bez uslovljavanja (if/else ili switch) i bez duplikacije koda**.

📌 Problem koji Strategy rešava

Zamislimo da pravimo aplikaciju za ispis dokumenata.
U zavisnosti od konteksta, korisnik može izabrati:

- da štampa kao PDF,
- da štampa na običnom štampaču,
- da sačuva dokument kao sliku.

Bez Strategy paterna, morali bismo da koristimo grane logike unutar jedne metode:

```
if (type.equals("pdf")) {
    // Štampaj kao PDF
} else if (type.equals("printer")) {
    // Štampaj na štampaču
} else if (type.equals("image")) {
    // Sačuvaj kao sliku
}
```

Ovakav pristup krši **Open-Closed princip** (klase treba da budu otvorene za proširenje, ali zatvorene za izmenu) i otežava održavanje i testiranje.

✅ Rešenje: Strategy patern

Strategy patern omogućava:

- da **različita ponašanja (algoritme)** enkapsuliramo u posebne klase,
- da ih dinamički menjamo bez promene glavne logike,
- da se ponašanje (strategija) ubrizga objektu i zameni u toku rada.

🧱 Struktura Strategy paterna

Ova struktura omogućava jasan razdvajanje ponašanja (strategija) od objekta koji ih koristi (konteksta), čime se izbegava čvrsto povezivanje i olakšava buduće proširenje sistema bez modifikacije postojećeg koda.

Element	Opis
Strategy	Interfejs koji definiše zajedničko ponašanje (npr. metoda execute())
ConcreteStrategy	Konkretne klase koje implementiraju različite verzije ponašanja
Context	Klasa koja koristi strategiju – delegira posao objektu strategije
Client	Postavlja strategiju i poziva kontekst da izvrši ponašanje

✅ Prednosti Strategy paterna

- Izbegava kompleksne if/else i switch grane
- Omogućava **laku zamenu ponašanja** u toku izvršavanja
- Poštuje **Open-Closed princip**
- Povećava **fleksibilnost i testabilnost** Sistema
- Pojednostavljuje dodavanje novih ponašanja

📍 Gde se koristi Strategy?

- **Sort algoritmi** – kada korisnik može birati između različitih metoda sortiranja (npr. bubble sort, quick sort, merge sort), bez potrebe da zna detalje njihove implementacije)
- **Raspoređivanje zadataka** – u sistemima gde različite strategije upravljaju izvršavanjem (npr. Round Robin, FIFO, Priority Scheduling)
- **Plaćanje u aplikacijama** – omogućava jednostavno dodavanje novih metoda plaćanja (kreditna kartica, PayPal, kriptovalute) bez menjanja osnovnog koda
- **Generisanje izveštaja** – kada korisnik može birati da li želi PDF, CSV ili HTML izveštaj, svaka forma može biti strategija
- **Kompresija fajlova** – aplikacije koje nude različite metode kompresije (ZIP, RAR, GZIP) mogu koristiti strategije za lako proširivanje bez duplikacije koda.

🔑 Ključne osobine

1. **Ponašanje se može menjati u toku izvršavanja** – bez gašenja aplikacije, moguće je prebaciti se sa jedne strategije na drugu
2. **Kontekst ne zna detalje implementacije strategije** – koristi interfejs, čime se postiže slabija povezanost
3. **Strategije su zamenljive i nezavisne jedna od druge** – svaka implementacija može se razvijati, testirati i optimizovati zasebno
4. **Višekratna upotreba** – ista strategija može se primeniti u više delova sistema ili više različitih aplikacija

---

⚠️ Izazovi i potencijalne zamke

- **Previše strategija može napraviti prekomeran broj klasa** – što može otežati održavanje u jednostavnim sistemima
- **Klijent mora da zna koju strategiju da izabere** – loš izbor može dovesti do neefikasnog rada
- **Prekomerna apstrakcija** – ako se koristi tamo gde nema realne potrebe za zamenljivošću, može zakomplikovati dizajn bez benefita
- **Teže praćenje toka izvršavanja** – posebno kada se strategije dinamički menjaju tokom runtime-a

🎯 Scenario primera

Zamislimo sistem za **plaćanje u online prodavnici**, gde korisnik može izabrati **različite načine plaćanja**:

- Kreditnom karticom
- PayPal-om
- Kriptovalutama

Koristićemo Strategy patern kako bismo mogli lako dodavati nove metode plaćanja bez menjanja osnovnog koda.

Ovakav pristup omogućava da svaka metoda plaćanja bude izdvojena u posebnu klasu, što doprinosi boljoj organizaciji i lakšem održavanju sistema.

📁 Struktura primera

payment/

├ PaymentStrategy.java

├ CreditCardPayment.java

├ PayPalPayment.java

├ CryptoPayment.java

├ PaymentContext.java

└ Main.java

✅ 1. PaymentStrategy.java – interfejs strategije

Ova klasa definiše zajednički interfejs za sve strategije plaćanja. Svaka konkretna strategija mora implementirati metodu pay, čime se obezbeđuje doslednost u korišćenju različitih načina plaćanja.

```
public interface PaymentStrategy {
    void pay(double amount);
   // Svi načini plaćanja moraju implementirati ovu metodu
}
```

✅ 2. CreditCardPayment.java – konkretna strategija

Ova klasa implementira plaćanje kreditnom karticom. Detalji kao što su broj kartice kapsulisani su unutar klase, a metoda pay prikazuje kako se transakcija vrši tim putem.

```
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Plaćanje " + amount + " RSD kreditnom
                           karticom: " + cardNumber);
    }
}
```

✅ 3. PayPalPayment.java – konkretna strategija

Strategija koja omogućava plaćanje putem PayPal naloga. E-mail korisnika se koristi za identifikaciju, a implementacija metode pay definiše konkretan način izvršenja plaćanja.

```
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Plaćanje " + amount + " RSD preko PayPal
                           naloga: " + email);
    }
}
```

✅ 4. CryptoPayment.java – konkretna strategija

Ova klasa omogućava plaćanje kriptovalutama. Svaka transakcija se izvršava pomoću zadate adrese novčanika, a implementacija metode pay simulira izvršenje takvog plaćanja. Dodavanjem ove klase lako proširujemo sistem za savremene metode plaćanja bez promene osnovne logike.

```
public class CryptoPayment implements PaymentStrategy {
    private String walletAddress;

    public CryptoPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Plaćanje " + amount + " RSD kriptovalutom sa
                           adrese: " + walletAddress);
    }
}
```

✅ 5. PaymentContext.java – kontekst koji koristi strategiju

PaymentContext je odgovoran za izvršavanje plaćanja koristeći odabranu strategiju. Klasa omogućava dinamičku promenu načina plaćanja u toku izvršavanja programa, čime se korisniku pruža fleksibilnost i sistemu modularnost.

```
// Klasa koja koristi PaymentStrategy i omogućava promenu strategije u
toku izvršavanja
public class PaymentContext {
    private PaymentStrategy strategy;

    // Dozvoljavamo menjanje strategije dinamički
    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }
```

```
    // Izvršava plaćanje koristeći trenutnu strategiju
    public void executePayment(double amount) {
        if (strategy == null) {
            System.out.println("Greška: Nije izabrana strategija
                                plaćanja!");
        } else {
            strategy.pay(amount);
        }
    }
}
```

✅ 6. Main.java (glavni program)

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

        // Plaćanje kreditnom karticom
        context.setPaymentStrategy(new CreditCardPayment
                                    ("1234-5678-9012-3456"));
        context.executePayment(5000);

        // Plaćanje preko PayPal-a
        context.setPaymentStrategy(new PayPalPayment("user@example.com"));
        context.executePayment(7500);

        // Plaćanje kriptovalutom
        context.setPaymentStrategy(new CryptoPayment("0xABCDEF123456"));
        context.executePayment(10000);
    }
}
```

📌 **Rezultat izvršavanja:**

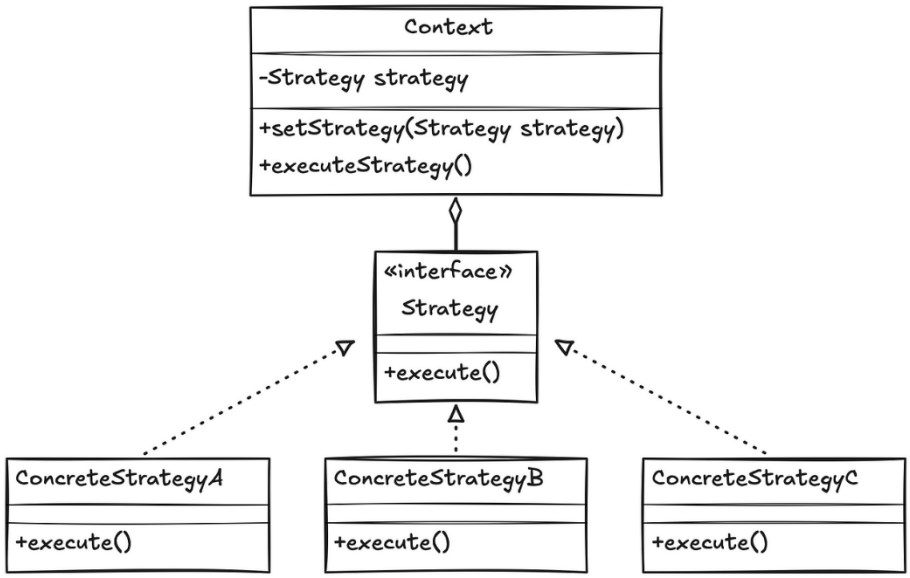
Plaćanje 5000.0 RSD kreditnom karticom: 1234-5678-9012-3456

Plaćanje 7500.0 RSD preko PayPal naloga: user@example.com

Plaćanje 10000.0 RSD kriptovalutom sa adrese: 0xABCDEF123456

🧠 **Zaključak:**

- Dodavanje nove metode plaćanja (npr. ApplePay) se postiže dodavanjem **nove klase** bez ikakvih izmena postojećeg koda.
- **Context klasa (PaymentContext) ne zna ništa o konkretnim strategijama**, samo zna da poziva pay().
- Možemo **promeniti ponašanje objekta u toku izvršavanja**, što je suština Strategy paterna.



Najvažnije što treba upamtiti u vezi ovog patterna:
Strategy obrazac omogućava da se strategija (kao što je metoda plaćanja) menja u svakom trenutku dok program radi, bez potrebe za ponovnim pokretanjem ili velikim izmenama koda. Ovo znači da korisnik može fleksibilno prebacivati između različitih metoda plaćanja u realnom vremenu, a sistem će ispravno izvršavati izabranu strategiju bez zastoja ili komplikacija.