Strukturalni paterni

U trenutnom dizajnu imamo snažno povezane (tight coupling) klase kao što su Osoba, Rezervacija, Plaćanje, Podrška, i Vozilo. Takav dizajn:

- Otežava testiranje pojedinačnih komponenti
- Ograničava proširivost sistema (npr. zamjena klase Plaćanje bez mijenjanja Rezervacija)
- Ne podržava fleksibilno korištenje različitih korisničkih interfejsa, servisa i tipova objekata

Dodavanjem strukturalnih paterna dizajna, možemo poboljšati modularnost, fleksibilnost i održivost.

Naši odabrani paterni koje ćemo primijeniti na dijagram klasa su Adapter i Facade te ćemo njih detaljnije objasniti u poređenju sa ostalim strukturalnim paternima.

1. Adapter patern

Adapter patern omogućava povezivanje klasa koje inače nisu kompatibilne po interfejsu. Na primjer, ako imamo novu klasu Rezervacija koja treba da koristi staru klasu Plaćanje, ali one imaju različite metode, Adapter služi kao posrednik. Tako ne moramo mijenjati postojeći kod, nego uvodimo sloj koji prevodi pozive iz jednog formata u drugi. Ovo omogućava integraciju starijih komponenti u nov sistem i olakšava prelazak na druge načine plaćanja (npr. kripto, vanjski API).

Problem: Klasa Rezervacija direktno koristi klasu Plaćanje. Ako se logika plaćanja promijeni (npr. prelazak sa kartica na kripto ili vanjski API), morali bismo mijenjati Rezervaciju.

Rješenje: Uvesti interfejs IPaymentProcessor koji definiše metode kao procesirajPlacanje() i generisiPotvrdu(). Zatim kreirati adaptere za različite implementacije (KarticniAdapter, KriptoAdapter, VanjskiApiAdapter), koje se mogu mijenjati bez uticaja na Rezervaciju

2. Facade patern

Facade patern se koristi kada je sistem previše kompleksan za krajnjeg korisnika. U slučaju rezervacija, korisnik bi morao ručno pozvati više metoda da izvrši rezervaciju, tj. odabir vozila, unos plaćanja, potvrda. Fasada objedinjuje sve ove korake u jednu jednostavnu metodu, npr. kreirajRezervaciju(). Ovim se pojednostavljuje upotreba sistema i sakrivaju tehnički detalji, a krajnji korisnici rade sa jasnim, jednostavnim interfejsom.

Problem: Korisnik (Osoba) trenutno mora direktno pozivati više metoda (rezervisiVozilo(), platiOnline(), potvrdiRezervaciju()), što je kompleksno.

Rješenje: Uvesti RezervacijaServis kao fasadu, koja nudi jednostavan metod kreirajRezervaciju(vozilo, korisnik, kartica) i interno poziva sve potrebne metode.

3. Decorater patern

Decorator patern omogućava dinamičko dodavanje funkcionalnosti objektima bez mijenjanja njihove osnovne strukture. Na primjer, osnovno vozilo može imati dodatke kao što su GPS, dječije sjedište, dodatno osiguranje. Umjesto pravljenja više podklasa za svaku kombinaciju, koriste se dekoratori koji omotavaju objekat i dodaju funkcionalnosti, čime se izbjegava eksplozija klasa i čuva fleksibilnost.

4. Bridge patern

Bridge patern odvaja apstrakciju od njene implementacije tako da mogu nezavisno evoluirati. U sistemu koji podržava više interfejsa (npr. web, mobilni), logika rezervacije može ostati ista, dok se prezentacija (UI) razlikuje. Bridge omogućava da UI sloj koristi istu poslovnu logiku kroz različite platforme, bez dupliciranja koda.

5. Proxy patern

Proxy patern se koristi kao zamjena za stvarni objekat u situacijama kada želimo kontrolisati pristup, odgoditi instanciranje ili pratiti korištenje. U sistemu rezervacija, Proxy se može koristiti za kontrolu pristupa bazi podataka ili za keširanje rezultata učestalih upita (npr. dostupnost vozila), čime se štede resursi i povećava sigurnost.

6. Composite patern

Composite patern omogućava tretiranje pojedinačnih objekata i njihovih grupa na isti način. U kontekstu vozila, možemo imati kategorije kao što su Automobil, SUV, Kombi, ali i super-kategorije koje sadrže više podkategorija. Umjesto da ručno upravljamo svakim vozilom posebno, Composite omogućava da se svi elementi tretiraju kao jedan entitet kada je to potrebno, olakšavajući grupne operacije i prikaz hijerarhija.