Paterni ponašanja

STRATEGY PATTERN

On omogućava enkapsulaciju, tj. izdvajanje različitih algoritama u odvojene klase, čime se aplikacija čini fleksibilnijom i skalabilnijom, lakše ćemo ih mijenjati u budućnosti ako to bude potrebno. U našoj aplikaciji „DnevnaDoza“ mi ga koristimo za sortiranje i filtriranje proizvoda u katalogu, gdje svaki algoritam sortiranja (npr. po cijeni, nazivu ili popularnosti) nasljeđuje zajednički interfejs. Kada korisnik odabere kriterij sortiranja, aplikacija kreira odgovarajući objekt strategije i prosljeđuje ga komponenti koja prikazuje listu proizvoda. Ovakav pristup smanjuje ovisnost između dijelova koda i poboljšava testabilnost - svaka strategija se može zasebno testirati. Također, ako se pojavi potreba za specifičnim algoritmom sortiranja po farmakološkoj kategoriji ili zalihama, jednostavno možemo dodati novu klasu bez utjecaja na postojeće. Na primjer, pri primjeni Strategy paterna prvo definišemo interfejs ISortiranjeProizvoda koji sadrži metodu sortiraj(List<Proizvod>).
Zatim kreiramo konkretne klase, poput SortirajPoCijeni, SortirajPoNazivu ili SortirajPoDatumu, koje svaka zasebno implementira ovu metodu prema svojoj logici. Na taj način, kad god poželimo dodati novi način sortiranja ili izmijeniti postojeći, dovoljna je izrada ili prilagodba odgovarajuće klase, bez potrebe za mijenjanjem koda. Ovakva struktura omogućava visoku proširivost i održivost aplikacije, jer se algoritmi sortiranja nalaze izolovani od osnovne logike aplikacije.

STATE PATTERN

State pattern omogućava objektu da mijenja svoje ponašanje u zavisnosti od trenutnog stanja, bez potrebe za gomilom if-else ili switch naredbi. U našoj aplikaciji „DnevnaDoza“ ga koristimo za upravljanje životnim ciklusom narudžbe koja može biti, naprimjer, u stanjima ***obrađena*** i ***neobrađena***. Svako stanje predstavlja zasebnu klasu koja implementira isti interfejs, a kontekst (u našem slučaju je to narudžba) određuje pozive metodama konkretnog stanja. Kada zaposlenik označi narudžbu kao obrađenu, internim pozivom mijenjamo stanje konteksta i time prilagođavamo sljedeće dozvoljene operacije (npr. slanje računa ili otkazivanje). Time se naša poslovna logika čisti od uslovnih grananja i postaje kasnije dosta lakše proširiva ukoliko nekada uvedemo nova stanja kao što su **„na čekanju“** i slično.

TEMPLATE METHOD PATTERN

Template Method pattern definira kostur ili osnovni tok algoritma, dok podklase mogu same specificirati svoju implementaciju pojedinih koraka. U našem sistemu ovaj pattern primjenjujemo kod metoda plaćanja (**kartično plaćanje, plaćanje pouzećem**). Zbog same upotrebe ovog patterna, mi izbjegavamo dupliciranje koda jer je struktura toka uvijek ista, a fleksibilnost omogućava precizno prilagođavanje ponašanja za svaku vrstu plaćanja. Slično, ovaj pattern možemo koristiti i za različite uloge korisnika (**admin, zaposlenik, kupac**). On kao rezultat daje dosljednost u načinu izvršavanja procesa, ali i lakše je održavanje jer se promjene u globalnom toku obrade rade na jednom mjestu. Template Method također potiče dijeljenje koda i pridržavanje principa DRY (Don’t Repeat Yourself).

ITERATOR PATTERN

Iterator pattern omogućava pristup elementima kolekcije bez izlaganja njene unutrašnje strukture. U našoj aplikaciji „DnevnaDoza“ ga koristimo za pregled proizvoda u katalogu, raspoređenih po kategorijama ili stranicama, tako da GUI komponenta može jednostavno dobiti sljedeći ili prethodni proizvod. Kad se koristi Iterator pattern, mi njegovom upotrebom razdvajamo kolekciju od tzv. mehanizma iteracije, pa ukoliko promijenimo način pohrane proizvoda (npr. prelazak na vanjski cache ili stream), iteratori ostaju nepromijenjeni. Ovo poboljšava i olakšava testiranje koda. On je također koristan i pri izvozu podataka, tj. može poslužiti i za generisanje PDF izvještaja o proizvodima.

OBSERVER PATTERN

Observer pattern uspostavlja vezu između objekata, gdje promjena u jednom objektu automatski obavještava sve zavisne objekte, tj. faktički registrovane posmatrače. U našem sistemu koristimo ga za obavještavanje korisnika o promjenama dostupnosti proizvoda: kada se zalihe promijene, tj. kada proizvod koji je korisnik dodao u korpu više nije dostupan. Ovaj pristup osigurava da je UI uvijek ažuriran bez potrebe za ručnim osvježavanjem podataka. Observer pattern pridonosi labavoj povezanosti komponenti- tj. on ne poznaje tačno na koje načine će se obavijesti obrađivati, nego samo zna da ih mora poslati.

COMMAND PATTERN

Command pattern enkapsulira zahtjeve u zaseban objekt, omogućavajući im skladištenje, redo/undo, logiranje ili izvršavanje u kasnijem trenutku. U našoj aplikaciji „DnevnaDoza“ implementirali smo ga za funkcionalnost „Otkaži narudžbu“, gdje svaka akcija korisnika (npr. dodavanje, brisanje ili izmjena stavke u korpi) kreira Command objekt. Ti objekti se stavljaju u CommandHistory, pa korisnik može poništiti posljednju promjenu ili vratiti više koraka unatrag kroz undo/redo operacije.

MEMENTO PATTERN

Memento patern omogućava spremanje i kasniji povrat prethodnog stanja objekta bez narušavanja enkapsulacije. U aplikaciji se koristi za funkcionalnost Dodaj u korpu, gdje korisnik ima mogućnost undo opcije, vraćajući prethodno stanje korpe u slučaju greške ili promjene odluke. Na primjer, korisnik može urediti adresu isporuke, a sistem sačuva prethodnu verziju u memento objektu. Može se implementirati klasa Profil, koja može stvoriti objekt tipa ProfilMemento, a klasa Caretaker čuva sve verzije. Kasnije, korisnik može odlučiti da vrati staru adresu pozivom vratiStanje().