Patterni ponašanja

1. Strategy – *izdvaja algoritam iz matične klase i uključuje ga u posebne klase*

   U okviru aplikacije DressCode, Strategy pattern bi se mogao koristit kod sortiranja artikala po kategorijama (vrsta artikla, veličina, boja, spol, itd.). Ovime se izbjegava pretjerano grananje u kodu i postigli da možemo zamijeniti algoritme u toku izvršavanja programa, odvojiti implementaciju od koga koji ga koristi i ne narušavamo Open/Closed princip jer možemo uvesti nove strategije bez promjene postojećih klasa.

2. State – *mijenja način ponašanja objekata na osnovu trenutnog stanja*

   Ovim paternom možemo provjeravati stanje artikla (da li je u korpi, u narudžbi, kupljen). Ovim paternom se ne narušava Single responsibility princip jer dio koda koji se odnosi na pojedinačna stanja je u posebnim klasama niti Open/Closed princip jer uvodimo nova stanja bez promjene postojećih stanja klasa.

3. TemplateMethod

   Ovaj patern se može implementirati u procesu postavljanja novog artikla koji sadži više koraka: validacija podataka te stavljanje artikla u bazu. Ovim bismo dobili jednostavniju strukturu, a pojedine dijelove implementacije bi bilo jednostavnije za mijenjati.

4. Chain of responsibility – *predstavlja listu objekata, ukoliko objekat ne može da odgovori prosljeđuje zahtjev narednom u nizu*

   Može se implementirati kod validacije narudžbe. Ako jedan dio validacije ne prođe, korisniku se vraća greška. Omogućava da se dodaju nove validacije bez izmjene postojećeg koda.

5. Command – *razdvaja klijenta koji zahtijeva operaciju i omogućava slanje zahtjeva različitim primaocima*

   Kada korisnik krene da naruči određeni artikal, umjesto da to direktno radi kontroler, imamo objekte koji brinu o tim radnjama. Na taj način, radnje se mogu vrlo lahko logirati, zapisati ili poništiti.

6. Iterator – *omogućava pristup elementima kolekcije sekvencijalno bez poznavanja interne strukture*

Ovaj patern omogućava lakši prolazak kroz različite kolekcije elemenata poput dostupnih artikala, korisnika, radnika, na zahtjev koristnika što poboljšava čitljivost koda i održivost, bez brige kako su ti podaci spremljeni.

7. Mediator – *enkapsulira protokol za komunikaciju među objektima dozvoljavajući da objekti komuniciraju bez međusobnog poznavanja interne strukture objekta*
   Ovaj patern bi mogao poslužiti za komunikaciju između narudžbe i korisniškog profila. Ako se narudžba uspješno izvrši, mediator brine o postavljanju obavijesti korisniku i dodavanju narudžbe u listu narudžbi korisnika, bez da se metode za obavijesti i dodavanje pozivaju što skmanjuje zavisnost klasa jednim o drugima i čini kod preglednijim i jednostavnijim.

8. Observer – *uspostavlja relaciju između objekata takvu da kad se stanje jednog objekta promijeni, svi vezani objekti dobiju informaciju*

   Ovaj patern omogućava da jedan objekat (korpa) obavještava druge objekte (npr. Notifikacije) automatski o promjenama svog stanja, bez da ih direktno povezuje. Na primjer, kada se status korpe promijeni, svi objekti odmah dobiju obavijest o tome i mogu reagovati, recimo pokazati koristniku ili poslati notifikaciju radnicima.

9. Visitor – *definira i izvršava nove operacije nad elementima postojeće strukture ne mijenjajući samu strukturu*

   Može se koristiti za dodavanje različitih operacija nad artiklima, poput računanja popusta ili generisanja izvještaja o artiklima – sve bez promjene same strukture klase Artikal. Na primjer, možemo dodati Visitor koji obilazi sve artikle i pravi izvještaj za administraciju bez da se mijenja kod same klase Artikal.

10. Interpreter – *podržava interpretaciju instrukcija napisanih za određenu upotrebu*

    Korisnik bi mogao napisati neki složeniji upit za filtriranje artikala, a interpreter bi to protumačio i vratio odgovarajuće rezultate.

11. Memento – *omogućava spašavanje internog stanja nekog objekta van sistema i njegovo ponovno vraćanje*

    Ovaj patern se može koristiti za spašavanje artikala koji više nisu dostupni.