PATERNI PONAŠANJA

STRATEGY PATTERN

Ovaj patern izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Pogodan je kada postoje različiti primjenjivi algoritmi (strategije) za neki problem. Strategy patern omogućava klijentu izbor jednog od algoritma iz familije algoritama za korištenje. Algoritmi su neovisni od klijenata koji ih koriste.

U našem sistemu postoji idealno mjesto za iskorištavanje ovog paterna. Naime kada želimo da sortirati Product-e po zahtjevu korisnika (recimo da korisnik želi sortirati Pruduct-e po cijeni od najniže ka najvišoj) mi možemo ponuditi razne “strategije” sortiranja, odnosno možemo osigurati različite algoritme sortiranja (recimo da su to Quick sort, Bubble sort i Selection sort). Svaki od ovih algoritama pruža različito vrijeme izvršavanja no u konačnici svi će imati isti rezultat, odnosno sortiranu listu Product-a. Zamišljeni dijagram bi izgledao ovako:

STATE PATTERN

State patern predstavlja samo dinamička verzija Strategy paterna. Objekat mijenja način ponašanja na osnovu trenutnog stanja. To se postiže promjenom podklase unutar hijerarhije klasa.

Ovaj pattern bi mogli hipotetski iskoristiti pri ocjenjivanju korisnika odnosno u klasi Review. Recimo da želimo imati informaciju da li smo već ocijenili tog korisnika i da se u zavisnosti od toga može ili ne može željeni korisnik ocijeniti (recimo da ukoliko se ne može ponovo ocijeniti možemo vidjeti ocjenu koju smo prethodno dali). Potrebno je kreirati interfejs IOcjena koji će implementirati dvije klase Ocjenjen i Neocjenjen. Da bi se dinamički odredilo koja ce klasa biti pozvana potrebno je provjeriti u bazi da li postoji review za odabranog korisnika koji je trenutni korisnik ostavio.

TEMPLATE METHOD PATTERN

Omogućava izdvajanje određenih koraka algoritma u odvojene podklase. Struktura algoritma se ne mijenja odnosno samo mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

Ovaj pattern je moguće i poželjno iskoristiti u našem sistemu. Naime potrebno je omogućiti korisniku da vrši filtriranje i sortiranje proizvoda na osnovu nekog kriterija. Da bi izbjegli dupliciranje koda i komplikacije koristit ćemo ovaj patern jer pristupamo istim Products iz baze te je njih potrebno sortirati i/ili filtrirati. Da bi se sve zamišljeno omogućilo kreirat ćemo klasu Sort koja sadrži listu Product-a te implementira metode sortj() te condition() koja ce biti apstraktna i njena implementacija ce se vršiti u izvedenim klasama. Recimo da imamo izvedene klase PriceSort, ConditionSort, GenderSort. Svaka od ovih klasa sortira listu Product-a po logičkom kriteriju (PriceSort po cijeni, ConditionSort po condition, GenderSort po gender).

ITERATOR PATTERN

Ovaj patern se koristi za prolaženje kroz elemente kolekcije bez izlaganja strukture te kolekcije. Odnosno pruža nam pristup elementima, bez obzira na to kako je kolekcija strukturirana, da bi se oni mogli koristiti u ostalim dijelovima koda.

Iterator patern možemo iskoristiti na bilo kojem mjestu gdje se javlja lista Product-a (klasa UserProducts i Cart). Kako je Product apstraktna klasa na njenom mjestu se može nalaziti bilo koja klasa izvedena iz nje što nam nudi idelno mjesto za iterator pattern, jer mi želimo biti u mogućnosti prolaziti kroz listu bez obzira koja je konkretna instanca klase u listi. Da bi ovaj pattern bio moguć potrebno je kreirati interfejse ICollection i Iterator. Klasa ProductCollection ce implementirati interfejs ICollection dok ce klasa ProductIterator

implementirati interfejs Iterator. ICollection ce posjedovati metodu createIterator() dok će Iterator posjedovati metode koje omogućuju prolazak kroz Collection, te neke dodatne kao što su hasNext() i/ili hasMore().

OBSERVER PATTERN

Uloga ovog patterna je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju.

Ovaj patern mi nećemo implementirati u našem sistemu ali zgodna upotreba ovog paterna jeste da kada se ocijeni korisnik dobije obavijest o toj ocjeni (moguće je obavijestiti o ocjeni i o korisniku koji je ocijenio). Pri izvedbi ovog paterna potrebno je kreirati interfejs IObserve. Potrebno je da organizujemo mehanizam koji će obavještavati User-a. Kako je ovo u potpunosti nepotrebno u sistemu, nema potrebe da se dalje razrađuje ideja ovog paterna.

CHAIN OF RESPONSIBILITY PATTERN

Ovaj pattern omogućava objektu da pošalje naredbu bez znanja koji će objekt primiti i rukovati njime. Zahtjev se šalje s jednog objekta na drugi čineći ih dijelovima lanca i svaki objekt u ovom lancu može obraditi naredbu, proslijediti je ili učiniti oboje.

Ovaj patern je najlakše objasniti na primjeru GUI-a. Naime pri kliku na interfejs propagiraju se informacije handlerima da bi se izvršila željena akcija koja je inicirana klikom. Različiti Handleri izvršavaju različite akcije (tipa Edit, Remove i slično).

MEDIATOR PATTERN

Mediator pattern koristimo da bismo izbjegli tijesno povezane frameworks, potreban nam je mehanizam koji će olakšati interakciju između objekata na način da objekti nisu svjesni postojanja drugih objekata.

Jedan od primjera predstavljaju Dialog klase u okvirima GUI aplikacija. Prozor dijaloga je kolekcija grafičkih kontrola. Klasa Dialog pruža mehanizam koji olakšava interakciju između kontrola. Na primjer, kada je nova vrijednost odabrana iz objekta ComboBox, oznaka mora prikazati novu vrijednost. I ComboBox i Label nisu svjesni međusobne strukture i svom interakcijom upravlja objekt Dialog. Svaka kontrola nije svjesna postojanja drugih kontrola.