PATERNI PONAŠANJA

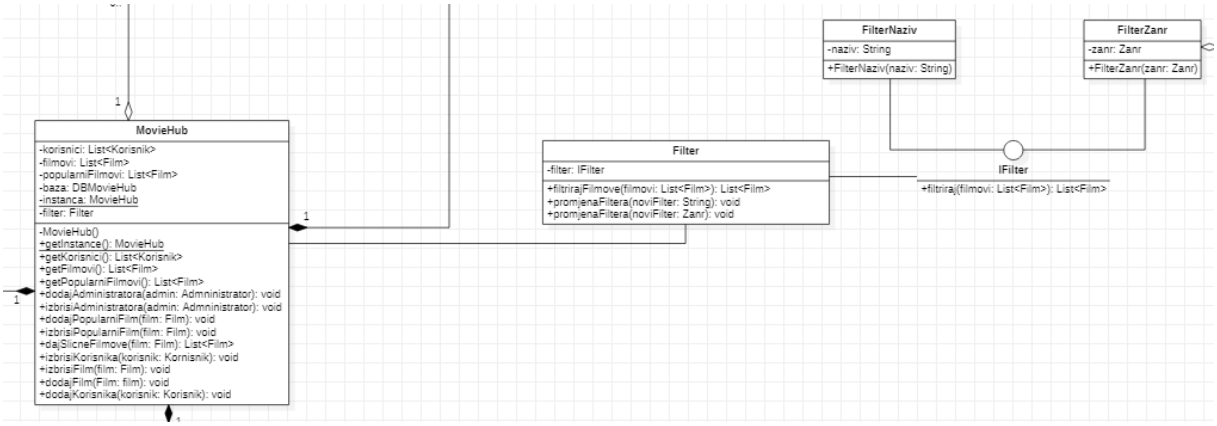
Strategy Pattern

Strategy 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.

U našem sistemu Strategy pattern možemo iskoristiti kada radimo filtriranje filmova prilikom pretrage. Moguće je filtrirati filmove ili po žanru ili po nazivu (pretraga na osnovu unesene riječi).

Primjenili smo ga tako što smo dodali novu klasu Filter koja ima atribut tipa IFilter, a IFilter je zapravo interfejs sa metodom filtriraj. Ovaj interfejs realizuju dvije klase, a to su FilterNaziv i FilterZanr, koje u zavisnosti koji je filter trenutno odabran, rade odgovarajuće filtriranje filmova. Promjena filtriranja se zapravo vrši u klasi Filter, koja je zadužena da to reguliše.

Korištenjem Strategy patterna olakšali smo korisniku da bira koje filtriranje želi, tako što smo odvojili različite algoritme u zasebne klase. Na slici ispod možemo vidjeti kako sada izgleda naš dijagram klasa:



State Pattern

State Pattern je dinamička verzija Strategy paterna. Objekat mijenja način ponašanja na osnovu trenutnog stanja. Postiže se promjenom podklase unutar hijerarhije klasa.

U našem sistemu nismo iskoristili ovaj patern, međutim kada bi napravili neke promjene mogao bi se iskoristiti. To promjene bi se ogledale u tome da uvedemo neko stanje filmova, tj. popularnost koja bi se određivala na osnovu broja pregleda. Dodali bi neku klasu Popularnost koja bi bila kao Context klasa, koja bi držala u sebi atribute potrebne za određivanje popularnosti, te također atribut nekog interfejsa IPopularity koji bi nasljeđivale klase npr. SlaboPopular, SrednjePopular, JakoPopular, itd. Interfejs IPopularity bi imao neku metodu dajPopularnost(), koju bi implementirale navedene klase.

TemplateMethod Pattern

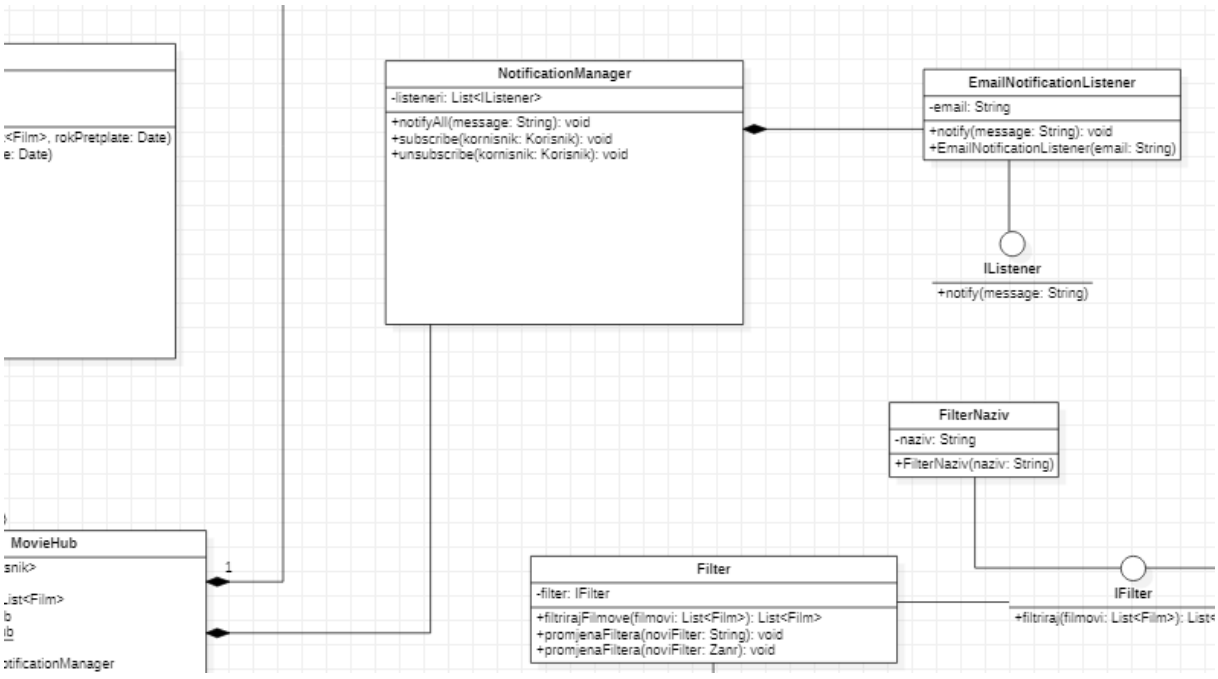
Omogućava izdvajanje određenih koraka algoritma u odvojene podklase. Struktura algoritma se ne mijenja - mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

Trenutno nigdje nije iskorišten ovaj patern, međutim kada bi u našem sistemu imali dvije podjele registrovanih korisnika na ObicniKorisnik i VIPKorisnik mogli bismo ga iskoristiti kako bi regulisali različite načine plaćanja pretplate, s obzirom da bi VIPKorisnik imao neke pogodnosti, kao npr. da plaća pretplatu 50% jeftinije od ObicniKorisnik.

Kada bi ga implementirali ovaj patern, uradili bi to na način da bi dodali neku klasu Clanarina, te bi ta klasa imala neku metodu TemplateMethod(). Definišemo također interfejs IClanarina koji ima metodu placanjeClanarine(). Ovaj interfejs bi implementirale gore navedene klase ObicniKorisnik i VIPKorisnik.Metoda TemplateMethod() bi pozivala odgovarajuću metodu placanjeClanarine() u ovisnosti koji je korisnik u pitanju.

Observer Pattern

Uloga *Observer* paterna je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju. U našem sistemu su ovaj patern implementirali na način da smo dodali klasu NotificationManager koja predstavlja *Subject* klasu te se unutar nje registruju (odnosno *subscribe-ju*) korisnici. Dalje, imamo interfejs IListener kojeg realiziju konkretni listeneri. Za sada u našem sistemu potreban je samo EmailNotificationListener. Unutar klase MovieHub se nalazi instanca tipa NotificationManager, te u zavisnosti od promjene liste popularni filmovi korisnicima koji su pretplaćeni, šalje se notifikacioni mail. Pretplaćivanje korisnika je zamišljeno putem nekih od *controllera*.



Iterator Pattern

Iterator patern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolekcija strukturirana.

U našem sistemu mi nismo iskoristili ovaj patern, iz razloga što za sve što je nama trenutno potrebno, imamo foreach petlju koju možemo koristiti za kretanje kroz neku kolekciju, tako da bi primjenjivanje ovog paterna za sada bilo suvišno. Međutim, ako bi kojim slučajem željeli da pravimo neku listu korisnika po tome kada su registrovali na naš sistem, tako da bude prvi korisnik onaj koji je najduže registrovan, pa zatim sljedeći po datumu registracije, ovaj patern bi bio iskorišten. Ako bi ga implementirali, to bi uradili na način da u našu kontejnersku klasu MovieHub dodamo jedan atribut iterator, koji će biti tipa Iterator. Imali bi jedan interfejs IterableCollection koji bi imao metodu createIterator(), a taj interfejs bi implementirala klasa MovieHub. Gore smo već spomenuli novi klasu koju bi napravili, a to je Iterator, a ona bi u sebi sadržavala jednu listu koja bi se sastojala od datuma registracije korisnika i taj atribut bi bio registracija : list<Date>, te jedan atribut indeks, koji bi čuvao trenutni indeks. Od metoda ta klasa bi imala getNext() koja bi ustvari vršila logiku iteriranja i hasNext() koja bi u ovom slučaju vraćala false ako smo došli do posljednjeg registrovanog korisnika, jer nema svrhe da implementiramo cirkularnu listu za ovaj slučaj. Dakle bitna nam je metoda hasNext(), jer ona vraća sljedeći element po redu, u ovom slučaju onog koji je registrovan poslije. Ukoliko dođe do registracije novih korisnika, metoda createIterator() bi se pozivala ponovo sa ažuriranom listom.