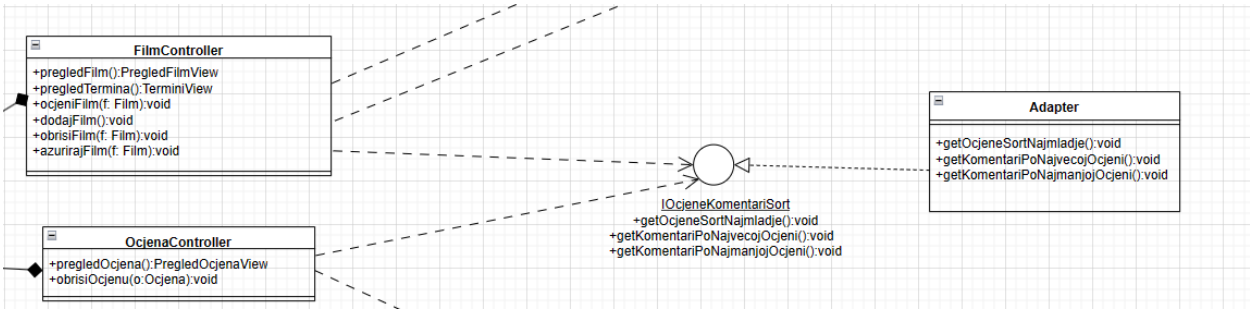
1. Adapter pattern

Svrha Adapter patern-a je da omogući širu upotrebu već postojećih klasa. U situacijama kada je potreban drugačiji interfejs već postojeće klase, a ne želimo je mijenjati, koristi se Adapter pattern. Ovaj patern kreira novu adapter klasu koja služi kao posrednik izmedu originalne klase i željenog interfejsa. Adapter pattern možemo koristiti u zastarjelim sistemima koji funkcionišu dobro, čije metode možemo i dalje koristiti, ali želimo nadograditi sistem nekim novim funckionalnostima.

U našem sistemu možemo nadograditi da korisniku omogućimo sortiranje ocjena po visini ocjena i u nekim slučajevima ovisnosti i od komentara.

Kreirat ćemo interfejs "IOcjeneKomentariSort" koji će posjedovati metode: getOcjeneSort Najmladje(), getKomentariPoNajvecojOcjeni(), getKomentariPoNajmanjojOcjeni().

Klasa Adapter implementira sve nabrojanje metode iznad.



2. Facade pattern

Fasadni pattern služi kako bi se korisnicima pojednostavilo korištenje kompleksnih sistema, odnosno koristimo kada imamo neki sistem koji ne želimo da razumijemo kako funkcioniše "ispod haube". Korisnici vide samo kranji izgled objekta, dok je njegova unutrašnja struktura skrivena. Na ovaj način smanjuje se mogućnost pojavljivanja grešaka, jer klijenti ne moraju dobro poznavati sistem kako bi ga mogli koristiti.

Ovaj pattern nećemo implementirati, ali ako bi smo htjeli to bi smo mogli na način da pretpostavimo da u našem sistemu imamo jasno definisane podsisteme, npr podsisteme za ocjenjivanje i komentarisanje filmova, za

pretraživanje filmova, za kupovinu karata... U slučaju da imamo veliki broj ovakvih podsistema, kao i dovoljnu raznolikost među korisnicima da im omogućimo različite poglede na sistem, mogli bismo napraviti klasu Fasada koja bi objedinila sve ove podsisteme i koja bi pružala jedinstven interfejs u ovisnosti o kojem se klijentu radi. Međutim mi nemamo dovoljno funkcionalnosti da bismo bili u potrebi da koristimo ovaj pattern.

3. Decorator pattern

Decorator pattern služi za omogućavanja različitih nadogradnji objektima koji svi u osnovi predstavljaju jednu vrstu objekta. Umjesto da se definiše veliki broj izvedenih klasa, dovoljno je omogućiti različito dekoriranje objekata (tj. dodavanje različitih detalja), te se na taj način pojednostavljuje i rukovanje objektima klijentima, i samo implementiranje modela objekata. U suštini, decorator pattern se koristi u slučajevima kada želimo dodijeliti dodatna ponašanja objektu tokom izvođenja programa bez da mijenjamo kod koji je na neki način u interakciji sa datim objektom.

U našem sistemu mogli bismo primijeniti decorator pattern na sljedeći način:

- Pretpostavimo da smo uveli atribut slika u klasi Korisnik. To znači da svi prijavljeni imaju sliku na svom profilu.

- Uvedimo interfejs ISlika sa metodom edituj()

- Uvest ćemo također dvije klase: SlikaFilteri i SlikeOsnovnePromjene. Ove klase će posjedovati atribut tipa Korisnik i svaka će na drugačiji način uređivati njegovu sliku pozivajući metodu interfejsa. Obavili smo dinamičko dodavanje novog elementa i ponašanja postojećem objektu dijagrama klasa. Objekat pri tome ne zna da je urađena dekoracija što je veoma korisno za ponovnu upotrebu komponenti softverskog sistema.

4. Bridge Pattern

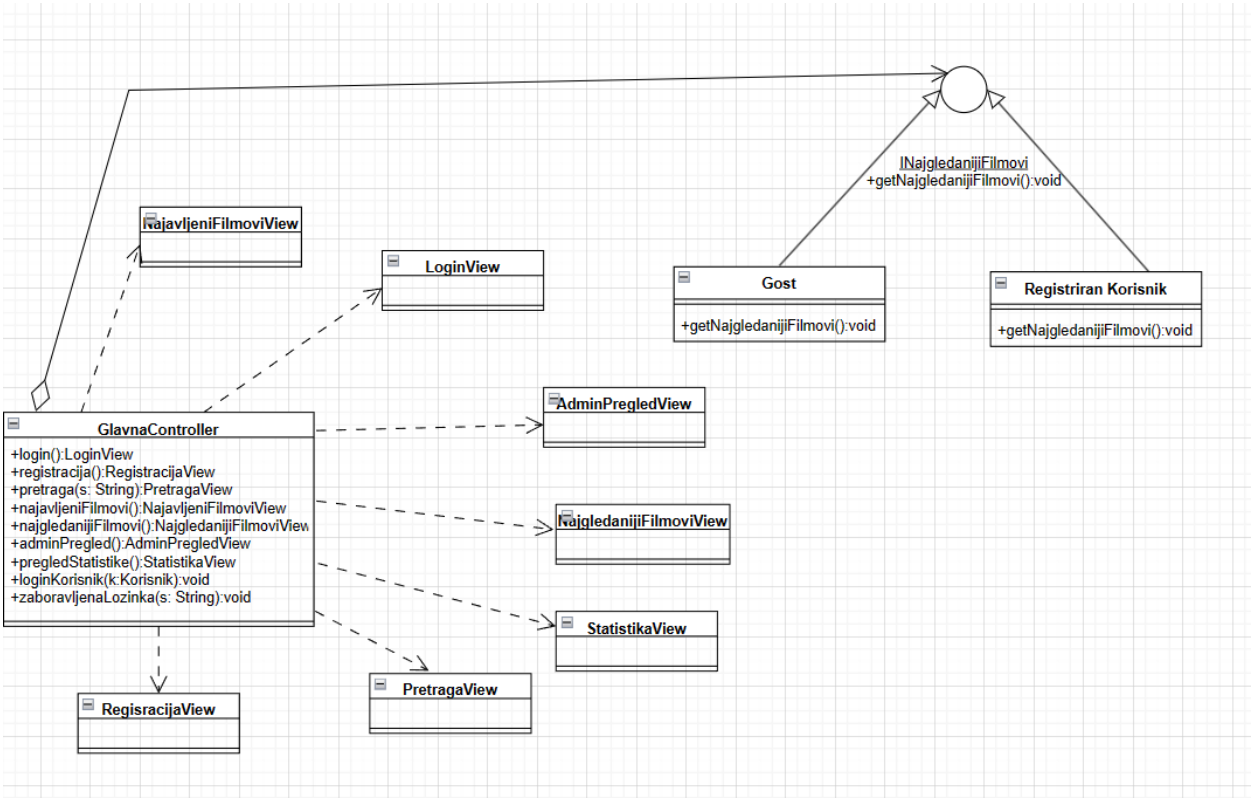
Osnovna namjena Bridge paterna je da omogući odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije.

Ovaj pattern bi mogao biti iskorišten ako bismo htjeli da realizujemo različit način prikazivanja filmova. Na primjer, korisnik i gost na isti način pretražuju filmove. Međutim, razlika u načinu prikazivanja najgledanijih filmova je u tome što korisniku filmovi budu sortirani i uz to prikazani sitni note koji naglašava ukoliko je on pogledao taj film (kupio kartu ili rezervisao kartu za taj film), dok kod gosta to ne može. Ovo bismo implementovali na sljedeći način:

- Uveli interfejs INajgledanijiFilmovi sa metodom getFilmovi()

- Dodali klasu Bridge koja vraća listu Filmova u formatu koji je isti i za gosta i korisnika

- Interfejs pozivaju dvije klase koje na različite načine sortiraju filmove. Jedna klasa se odnosi na način sortiranja filmova kod korisnika, a druga kod gosta.



5. Composite pattern

Composite pattern koristimo da omogućimo formiranje strukture stabla pomoću klasa, u kojoj se individualni objekti (listovi stabla) i kompozicije individualnih objekata (korijeni stabla) jednako tretiraju. Koristi se kada svi objekti imaju različite implementacije nekih metoda, ali im je potrebno svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija.

U našem slučaju to bi mogli biti obični korisnik i VIP korisnik kada bi smo ga implementirali, koji bi imali posebne pogodnosti kao što je popust prilikom kupovine karata, neki pokloni... Iako su ova dva korisnika na "različitim nivoima", pristupalo bi im se na isti način i implementacija bi bila pojednostavljena. Tada je potrebno napraviti interfejs IPristup koji definira zajedničke operacije za objekte, te Composite klasu koja će sadržavati listu objekata tipa IPristup. Klase korisnik i VIP korisnik će implementirati na drugačiji način metodu interfejsa.

6. Proxy pattern

Svrha Proxy patern-a je da omogući pristup i kontrolu pristupa stvarnim objektima. Proxy je obično mali javni surogat objekat koji predstavlja kompleksni objekat čija aktivizacija se postiže na osnovu postavljenih pravila.

Ovaj pattern možemo primijeniti u našem sistemu tako što ćemo postaviti kontrolu pisanja komentara. Jedino korisnik aplikacije može pisati komentare, dok osoba koja je prijavljena ko gost nema tu mogućnost.

7. Flyweight pattern

Ovaj pattern se koristi kada kreiramo objekte samo po potrebi kada imaju različito specifično stanje, a osnovno stanje je isto za sve objekte. Korištenjem ovog patterna se onemogućava stvaranje velikog broja instanci objekata koji u suštini predstavljaju jedan objekat. Samo ukoliko postoji

potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (bezlično stanje). Korištenje ovog patterna je veoma korisno u slučajevima kada je potrebno vršiti uštedu memorije.

Naš sistem je moguće proširiti na način da se omogući svim korisnicima aplikacije da imaju svoju profilnu sliku. U tom slučaju bilo bi potrebno koristiti unaprijed zadanu sliku, ako korisnik ne želi da prikazuje vlastitu. S obzirom da je moguće da više korisnika zadrži tu default-nu sliku potrebno je implementirati flyweight pattern kako bi ti korisnici koristili jedan zajednički resurs. To možemo postići kreiranjem interfejsa ISlika sa metodom dajSliku(), koji će nam vratiti sliku korisnika. Taj interfejs će implementirati klase Slika(čuva individualnu sliku korisnika) i DefaultSlika (čuva defaultnu sliku i potrebna je samo jedna njena instanca)