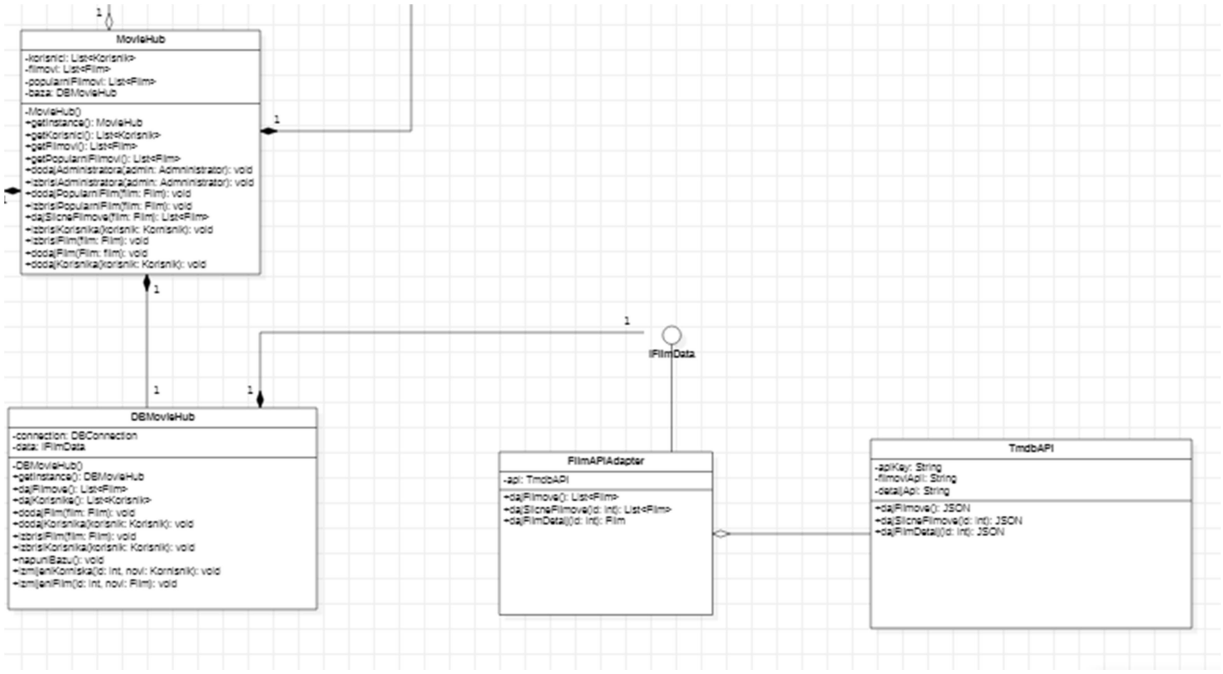
STRUKTURALNI PATERNI

Adapter patern

Adapter patern služi da se postojeći objekat prilagodi za korištenje na neki novi način u odnosu na postojeći rad, bez mijenjanja same definicije objekta. S obzirom da je naša tema vezana za filmove, to nam daje mogućnost da koristimo razne API servise bilo da „pupunimo“ bazu s njima ili da privremeno koristimo podatke s tih API-a. Kako većina API servisa vraćaju podatke u obliku JSON-a potrebno je prilagoditi podatke iz tog oblika da bi bili upotrebljivi u klasi Film. Ovdje se *Adapter* patern sam nameće, tako da smo na dijagramu klasa dodali sljedeće:



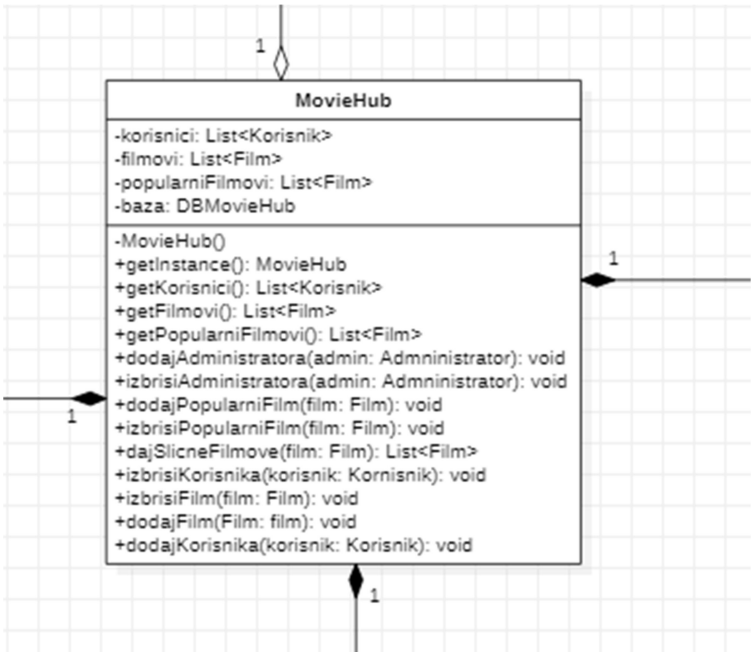
Dakle, glavna klasa MovieHub koristi klasu DBMovieHub koja predstavlja logiku komuniciranja sa bazom. U toj klasi postoji metoda napuni() koja će napuniti bazu ukoliko je ona prazna sa recimo prvih 250 filmova. Ona zahtjeva interfejs IFilmData koji obezbjeđuje podatke kojim će se baza napuniti. Taj intefejs realizira klasa FilmAPIAdapter koji će adaptirati podatke dobijene u JSON formatu iz klase TmdbAPI u instance klase Film tako da se mogu koristiti u ostatku sistema. Bitno je napomenuti da ova logika upisivanja s API-a u bazu ima samo smisla u našem sistemu, jer mi nismo konkretno neki realni servis puput Netflix-a koji posjeduje velike data centre za pohranjivanje svih tih filmova i serija. Na ovaj način možemo obezbjediti da ukoliko neko želi testirati našu aplikaciju, to može učiniti i bez skidanja naše orginalne baze

(koja može biti velika nekoliko GB), već jednostavno se može pozvati metoda napuni() koja će s API-a ispuniti bazu osnovnim podacima.

Facade patern

Facade patern se koristi kada sistem ima više identificiranih podsistema (*subsystems*) pri čemu su apstrakcije i implementacije podsistema usko povezane. *Facade* patern na neki način daje jednostavan interfejs za korištenje kompleksnijih povezanih subsistema.

Nakon implemeniranja adapter paterna mogli smo uočiti da klasa MovieHub zapravo ima odlike *Facade* paterna.



Kada posmatramo ovu klasu s ugla klijenta (klijenta kao nekog *code-a* koji će je koristiti, a ne *end usera*) zapravo dobijamo jednostavan intefejs za korištenje cijelog sistema, odnosno tu fasadu sistema. Uočavamo da veoma lagano možemo iskorisiti metode dodajFilm(), dajFilm(), izbrisiKorisnika() bez znanja šta se zapravo dešava u pozadini, da li podaci dolaze iz baze ili API-a čime se oslabila zavisnost klijenta od same izvedbe pohrane podataka.

Decorator patern

Decorator patern služi za omogućavanja različitih nadogradnji objektima koji svi u osnovi predstavljaju jednu vrstu objekta (odnosno, koji imaju istu osnovu). U našem sistemu nije prisutan nijedan očit primjer za upotrebu ovog paterna. Ukoliko bi imali potrebu da u klasi Film ne čuvamo žarn kao atribut već da na neki način žarn utiče na određene metode klase neko bi mogao pomisliti da jednostavno napravimo nasljeđivanje za svaki konkretan žanr, npr PsiholoskiFilm, TrilerFilm i sl. No nameće se problem ukoliko bi bilo potrebno da jedan film

spada u više žanrova (što je gotovo uvijek slučaj u praksi). U tom slučaju bi bilo potrebno napraviti klasu za svaku moguću kombinaciju žanrova što bi prouzrokovalo "eksplozijom" klasa, te bi za recimo ukupno 5 žanrova bilo potrebno implementirati 31 klasu. Taj problem bi se mogao elegantno riješiti upotrebom *decorator* paterna, te bi a svaki žarn napravili *decorator* klasu koja bi dekorisala neki film i riješila bi problem eksplozije klasa. Naravno ovo je samo hipotetički slučaj ukoliko bi bilo potrebno polimorfizam implementirati na osnovu žanra.

Ideja se može iskazati sljedećim pseudokodom

```
public class Film(){
    ... // atributi
}
public interface IZanr(){
    void obogatiZanrove();
}

public class AkcijaZanr() implements IZanr{
    Film film;
    void obogatiZanrove(){
    ... // implementacija metode
    }
}

public class KomedijaZanr() implements IZanr {
    Film film;
    void obogatiZanrove(){... // logika kojom bi obogaćivali
    }
}
// za svaku novu zanr klasu potrebno je samo dodati odgovarajući dekorator
```

***Bridge* patern**

Bridge patern služi kako bi se apstrakcija nekog objekta odvojila od njegove implementacije. Ovaj patern nije očigledno primjenjiv na naš sistem. Međutim, ukoliko bi u našem sistemu postojala potreba da u klasi Film imamo metodu reproduciraj(), te da bi u zavisnosti od toga da li je film spremljen u bazi ili na nekom udaljenom servisu koji naša kompanija koristi, npr. neki API koji vraća link filma koji je spremljen na nekom drugom servisu (poput YouTube, Vimeo i sl.) Ali također svi filmovi u metodi reproduciraj imaju zajednički dio koda koje se izvršava, hipotetski recimo da je to logika kojom se ispisuju osnovni podaci filma na ekran (naziv, glumci i sl.). U tom slučaju bilo bi korisno napraviti interfejs IReproduciraj kojeg bi implementirale konkretne klase ReproducirajAPI, ReporducirajDB i sl. Tada bi klasa Film umjesto *string-a* filmPath zahtjevala interfejs Ireproduciraj, te bi unutar metode reproduciraj() pozivala metodu interfejsa, dakle imali bi nešto kao

```
public interface IReproduciraj(){
    void reproduciraj();
}

public class ReproducirajAPI implements IReproduciraj{
    ReproducirajAPI(String api){ ... }
    void reproduciraj(){
        ...
    }
}
public class ReproducirajDB implements IReproduciraj{
    ReproducirajAPI(String filePath){ ... }
    void reproduciraj(){
        ...
    }
}
public class Film{
    IReproduciraj video = new ReproducirajAPI("api/link");
    /* naravno ovo bi se spicificiralo u konstruktoru */
    ...
    ...
    void reproduciraj(){
        ...
        // najprije izvršavanje akcija koje su zajedničke za sve filmove
        // ispisivanje osnovih podataka o filmu na ekran i sl.
        video.reproduciraj();
    }
}
```

Isječak koda se naravno ne kompjalira, već je cilj bio predstaviti ideju.

Sada još ako zamislimo da umjesto konkretne klase film imamo apstaktnu klasu Media iz koje ćemo nasljeđivati konkretne tipove možemo uvidjeti punu snagu ovog paterna, s ozbirom da on omogućava ispunjenje *Open-Closed* principa. Naravno, bitno je napomenuti da je i ovo hipotetiski slučaj što ne mora značiti da se ovaj patern neće konkretno primijeniti u daljnjem razvoju projekta.

Composite **patern**

Što se tiče Composite paterna, on se koristi kada objekti imaju različitu implementaciju neke metode, no međutim potrebno im je svima pristupati na isti tačin.

Trenutno mi ovaj patern nismo iskoristili u našem projektu, međutim postoje situacije u kojima bi se mogao iskoristiti. Ukoliko bi imali u našem projektu neki sistem reputacija za sve korisnike koji su upisani u sistem, onda bi ovaj patern mogao dosta pomoći. Za svakog korisnika bi se reputacija računala na drugačiji način, te ukoliko bi neko želio da pristupi pregledanju liste reputacija svih korisnika, ovaj patern bi riješio taj problem. Napravili bismo jedan interfejs koji

bi nazvali IReputacijeIzvjestaj koji bi definisao metodu dajIzvjestajReputacija(), a taj interfejs bi implementirala neka klasa Reputacije, te svi različiti tipovi korisnika koji postoje u sistemu. Pozivom metode dajIzvjestajReputacija(), pozivaoc će dobiti spisak svih reputacija, bez obzira što se za različite tipove korisnika na različit način računa reputacija, a patern će riješiti taj problem da možemo imati izvještaj na jednom mjestu.

Proxy patern

Proxy patern služi za dodatno osiguravanje objekata od pogrešne ili zlonamjerne upotrebe, te se primjernom ovog paterna omogućava kontrola pristupa objektima, te se onemogućava manipulacija objektima ukoliko neki uslov nije ispunjen, odnosno ukoliko korisnik nema prava pristupa traženom objektu. U našem sistemu bi mogli Proxy patern iskoristiti tako da napravimo da samo Administratori imaju pristup svim korisnicima i svim filmovima u sistemu, tako da imaju razne mogućnosti izmjene, dodavanja i brisanja sadržaja samo oni korisnici koji su Adminstratori. Ovaj patern bi mogli implementirati na način da dodamo neki interfejs IpristupSadrzaju koji bi imao metode za dobijanje i mijenjanje sadržaja koji se nalazi u bazi. Taj interfejs bi implementirale klase DBMovieHub i nova klasa Proxy koja bi imala u sebi nivo pristupa u zavisnot od toga ko pokušavama pristupati. Nivo pristupa bi bio odobren ukoliko bi administrator bio taj koji pokušava pristupiti sadržaju, te bi za to bila zadužena metoda pristup() koja bi se nalazila u klasi Proxy. Također klasa Proxy bi u sebi kao atribut imala listu korisnika i listu filmova, jer su to upravo liste kojima bi samo administatori imali pristup. MovieHub klasa sistema bi imala u sebi atribut tipa Proxy, koja bi služila za regulisanje tog pristupa, na koje bi imali samo pravo Administratori.

Flyweight patern

Flyweight patern koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji svi u suštini predstavljaju jedan objekat. Njegova namjena je prvenstveno da smanji utrošak RAM memorije. Kako u našem sistemu uglavnom sve instance objekta su različite i nemaju zajedničkih atributa npr. nazivi filmova su različiti, *username-i* korisnika su različiti. Ovdje se pod zajedničkim atributom misli na konkretno atribute koji imaju istu vrijednost kod tih objekata. Hipotetički, možemo zamisliti slučaj kada bi u klasi Korisnik bilo prisutno nekoliko atributa tipa int, string, list<double> i sl. Jednostavno pretpostavimo da su oni prisutni i potrebni u toj klasi, te da su potpuno identični za sve konkretne tipove korisnika tj. Administatora i Registrovanog korisnika. Recimo da je memorijsko zauzeće skupine tih atributa 20 KB. Sada ukoliko bi kreirali u 1 000 000 instanci Registrovani korisnik (ako smo Netflix ☺) tada bi memorijsko zauzeće iznosilo 1 000 000 * 20KB što je približno 20GB. Naravno ove cifre su poprilično nerealne, ali cilj je bio dokazati šta se može dogoditi u nekom ekstremnom slučaju, kada razmišljamo u terminima *big data.* Kako bi se riješio ovaj problem pogodno je iskorisiti *flyweight* patern. Naime, zajedničke pomenute atribute možemo grupisati u jedan poseban

objekat nazovimo ga *Object*, te u klasi Korisnik ćemo umjesto pojedinačkih atributa čuvati referencu na taj objekat. Sada ćemo prilikom kreiranja instanci Registrovnih korisnika uvijek proslijeđivati referencu na jedan već kreirani *Object,* čime će se uštediti značajna količina memorije.