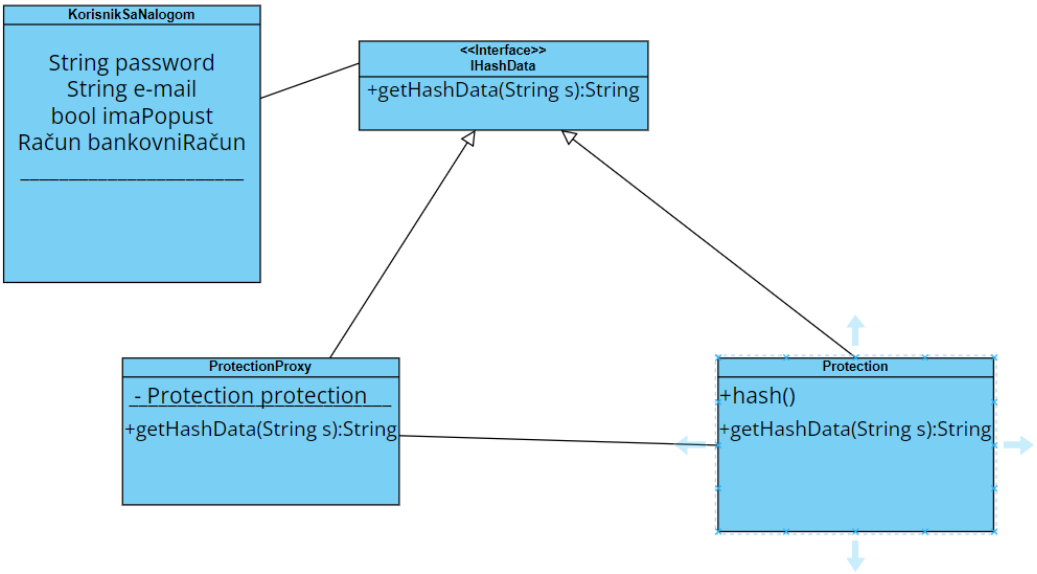
Facade pattern je design pattern koji se koristi za pružanje jednostavnog interfejsa (facade) prema kompleksnom podsistemu. Ovaj obrazac omogućava klijentima da komuniciraju sa složenim podsistemom putem jednostavnog interfejsa, prikrivajući detalje implementacije i olakšavajući korištenje složenih funkcionalnosti.
Facade pattern se obično implementira kroz klasu koja predstavlja fasadu. Fasada pruža jednostavne metode i operacije koje klijenti mogu koristiti za interakciju sa složenim podsistemom. Unutar fasade, kompleksne operacije i interakcije sa podsistemom se obavljaju, ali klijenti nisu svjesni tih detalja. Glavna ideja iza facade patterna je da se pojednostavi upotreba složenog podsistema tako što se pruža jasan i jednostavan interfejs. To olakšava klijentima da koriste složene funkcionalnosti bez potrebe da se detaljno upuštaju u kompleksnost unutrašnje implementacije.Klijenti vide samo fasadu, odnosno krajnji 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. U našem projektu, facade pattern se može koristiti kako bi se pojednostavila interakcija sa složenim podsistemom koji obuhvata različite komponente kao što su rezervacija karata, kupovina karata itd. Na ovaj način, klijenti mogu koristiti jednostavne metode koje pruža fasada ("RezervisiKartu","KupiKartu") kako bi interagirali sa složenim podsistemom. Fasada će se brinuti o detaljima implementacije i olakšati korišćenje različitih funkcionalnosti koje pruža kino.

Bridge pattern je design pattern koji se koristi za razdvajanje apstrakcije od implementacije. Ovaj obrazac omogućava da se apstrakcija i implementacija mogu nezavisno razvijati i promjenjivati, što omogućava veću fleksibilnost i olakšava dodavanje novih funkcionalnosti.
Bridge pattern se obično implementira kroz upotrebu dvije hijerarhije klasa - apstraktne klasa (Abstraction) koja predstavlja apstrakciju i implementatorske klase (Implementor) koje predstavljaju implementaciju. Apstrakcija sadrži referencu na objekat implementacije i definiše interfejs za klijente, dok implementatorske klase definišu konkretne detalje implementacije. Na taj način, promjena u implementaciji ne utiče na klijente koji koriste apstrakciju, a isto tako, promjena u apstrakciji ne zahtijeva izmjene implementacije Ovaj patern veoma je važan jer omogućava ispunjavanje Open-Closed SOLID principa, odnosno uz poštivanje ovog paterna omogućava se nadogradnja modela klasa u budućnosti te osigurava da se neće morati vršiti određene promjene u postojećim klasama. U našem projektu, bridge pattern se može upotrijebiti za razdvajanje apstrakcije različitih vrsta filmova (žanrova) od njihove konkretne implementacije. Na taj način, možemo omogućiti fleksibilnost u dodavanju novih vrsta filmova i promjeni detalja implementacije, ne utječući direktno na klijente koji gledaju filmove.

Proxy pattern je strukturalni dizajn patern koji omogućuje da se jedan object djelomično ili u potpunosti zamjeni s drugim objektom koji djeluje kao njegov zastupnik. Proxy object uspostavlja vezu između klijenta I stvarnog objekta.
Proxy objekt može pružiti korisne funkcije kao što su kontrola pristupa (npr. autentifikacija), skraćivanje vremena odziva i daljinski pristup (npr. pristup objektima

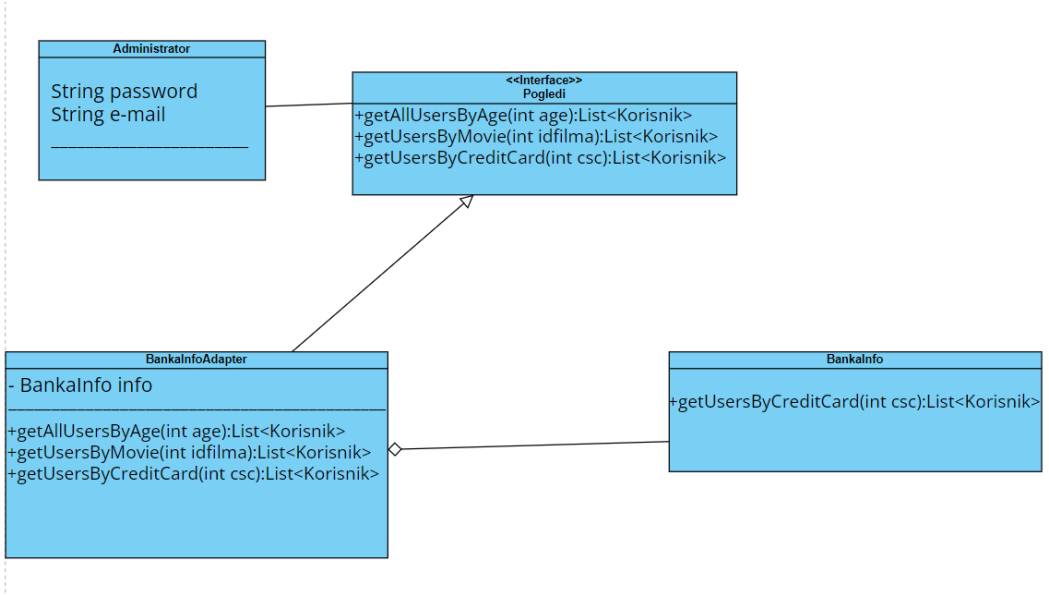
na drugim računarima. Proxy može biti usmjeren prema nekoliko vrsta objekata, uključujući CacheProxy, FirewallProxy, SmartReferenceProxy, RemoteProxy, VirtualProxy i ProtectionProxy. Primjena Proxy dizajn uzorka ima nekoliko prednosti kao što su povećanje sigurnosti, povećane brzine i smanjenje resursa. Međutim, zamjena objekata može uticati na performanse sistema. Proxy dizajn pattern je poželjan u aplikacijama koje rade s objektima koji su skupi za stvaranje ili koji se ne kreiraju često. Također se može koristiti u aplikacijama koje trebaju uspostaviti sigurnost, kao što su aplikacije za bankovne transakcije ili za pristup bazi podataka



Dakle ovdje na slici u implementaciji imamo korisnika sa nalogom koji je u ovom slučaju klijent kojem se omogućava heširanje podataka. Dalje nasljeđujemo iz interfejsa Proxy klasu I RealService klasu koja je u ovom slučaju klasa Protection. Ovo funkcioniše tako što u Proxy klasi imamo atribut tipa RealService klase i preko njega koristimo sve metode koje RealService klasa nudi. Naprimjer u implementaciji metoda u Proxy klasi možemo koristiti atribut tipa Protection i koristit metode koje su u Protection klasi, tako da kad korisnik pozove Proxy klasu i neku njenu metodu uopšte nema pojma da su za implementaciju te metode korištene metode iz neke druge klase. Također poželjno je i do sada obrađene podatke staviti u HashMap() da se ne bi gubilo vrijeme na heširanje već do sada heširanih podataka, odnosno ideja je da se do sada već heširani podaci dobave u konstantnom vremenu. Za tu svrhu možemo koristiti HashMap u ProtectionProxy klasi.

Adapter pattern je jedan od najčešće korištenih dizajn patterna u softverskom inžinjeringu. Ovaj pattern služi za povezivanje različitih klasa ili interfejsa tako da mogu međusobno komunicirati.
Adapter se koristi kada želimo koristiti postojeći kod s drugim kodo koji ima drugačiji interfejs. U tom slučaju, adapter preuzima ulogu posrednika između koda koji koristimo i koda s kojim želimo surađivati. Koristi se u situacijama kada postojeći kod

ne odgovara novim zahtjevima ili kada trebamo integrirati postojeći kod u drugi dio softvera.
Adapter se može implementirati na dva načina: klasom adaptera (class adapter) ili objektom adaptera (object adapter). Klasa adaptera koristi nasljeđivanje (koje ćemo mi koristiti u našoj implementaciji adapter patterna), dok objekt adaptera koristi kompoziciju.
Naprimjer, ako imamo klasu koja ima svoj interfejs koji se razlikuje od interfejsa druge klase s kojom želimo surađivati, možemo koristiti adapter kako bi smo omogućili komunikaciju između te dvije klase.
Adapter pattern pomaže u poboljšanju fleksibilnosti softvera. Također omogućava reutilizaciju postojećeg koda, što smanjuje potrebu za pisanjem novog koda i smanjuje trošak razvoja softvera.



Pretpostavimo da administrator iz sigurnosnih razloga ne može pristupiti svim CSC brojevima kartica svakog od korisnika, da bi administratoru dali mogućnost recimo da vidi sve članove sortirane po CSC brojevima kartica koristit ćemo *adapter* klasu, te ćemo u nju kao atribut staviti *adaptee* klasu koja je u ovom slučaju BankaInfo. Dakle, administrator će pozvati adapter klasu ukoliko bude želio da sazna informacije koje su u vezi sa bankom, odnosno osjetljive informacije za koje on ne bi trebao da ima direktan pristup. S obzirom da adapter klasa ima kao atribut objekat adaptee klase, adapter klasa će moći u svojim metodama koristiti sve metode adaptee klase, tako naprimjer ukoliko želimo da imamo sve korisnike po CSC brojevima, u metodi getUsersByCreditCard koja se nalazi u adapter klasi ćemo moći koristiti istoimenu metodu adaptee klase koja ima znanje da nam da željenu listu. Implementacija metode se nalazi u adaptee klasi dok u adapter klasi možemo samo pozvati tu metodu. Tako da administrator može dobiti te informacije tj.recimo sortirane korisnike po CSC brojevima bez da zna neke osjetljive podatke. Dakle osnovna ideja je da u nekoj metodi adapter klase možemo koristiti metode adaptee klase koje će diretkno odlučiti o tome šta će koja metoda u adapter klasi raditi. Korisnik naravno poziva metode adapter klase koje mu mogu dati informacije koje želi, iako korisnik nema svijest o postojanju adaptee klase, te nema svijest da je zapravo u adaptee klasama implementacija metoda koje određuju ponašanje metoda u adapter klasi.

Flyweight pattern je dizajn pattern koji se koristi za efikasno upravljanje objektima koji imaju malo zajedničkih podataka, odnosno koristi se kako bi se onemogućilo nepotrebno stvaranje velikog broja instanci koje ustvari predstavljaju jedan objekat. Najveća prednost ovog patterna je smanjenje memorije, na način da se podaci dijele među objektima umjesto da se uvijek ponavljaju za svaki novi objekat. Novi objekat će se kreirati samo ako postoji potreba za njegovim kreiranjem odnosno ako taj objekat ima jedinstvene karakteristike tada će se izvršiti njegova instatacija, u suprotnom će se iskoristiti postojeća instanca objekta. Najčešće za čuvanje podataka koji su zajednički svim objektima, koriste se tabele i mape. Tako da kada se kreira novi objekat, Flyweigth pattern prvo vrši provjeru da li već postoji objekat sa zadanim podacima. U slučaju da postoji, onda se taj objekat vraća. Ukoliko ne postoji, kreira se novi objekat sa zadanim podacima te se dodaje u strukturu kako bi mogao biti iskorišten kasnije. Neke od najbitnijih prednosti koje donosi korištenje Flyweighta patterna su ušteda memorije, također sa smanjenjem memorije se mogu povećati I performanse programa. Također flyweight olakšava ponovno korištenje objekata koji posjeduju zajedničke podatke, ali I poboljšava samu čitljivost koda. Međutim, Flyweight pattern može izgubiti svoju korisnost u situacijama kada nije moguće dijeliti zajedničke podatke između objekata ili ako objekti imaju veliki broj različitih podataka.

Decorator pattern je strukturni softverski patern koji omogućava dodavanje funkcionalnosti objektu dinamički, omotavanjem objekta u drugi objekt, poznat kao dekorator. Ovaj patern omogućava proširenje ponašanja objekta bez potrebe za izmenom izvornog koda objekta. Kada se koristi, kreira se novi objekat koji prihvata ulazne zahteve ili pozive metoda i dodaje dodatnu funkcionalnost prije ili poslje izvršavanja tih zahtjeva, često bez menjanja samog objekta. Dekoratori mogu biti slojeviti, gde se jedan dekorator dodaje na drugi kako bi se postigla dodatna funkcionalnost. Prednosti upotrebe decorator paterna su fleksibilnost, reupotrebljivost, razdvajanje odgovornosti i mogućnost kompozicije. Međutim, postoje i izazovi i mane kao što su složenost, redoslijed izvršavanja, performanse i potreba za zajedničkim interfejsom objekata.

Composite Pattern služi za kreiranje hijerarhije objekata. Koristi se kada svi objekti imaju različite implementacije nekih metoda, no potrebno im je svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija. Klijenti objekata mogu komunicirati sa hijerarhijom bez potrebe da prave razliku između pojedinačnih objekata i grupa objekata. Composite Pattern je koristan u različitim domenima, kao što su grafički korisnički interfejsi, sistemi datoteka i bilo koji scenario u kojem morate da predstavite odnose u hijerarhijskoj strukturi. Ovaj patern nećemo implementirati, ali kada bi se odlučili da proširimo aplikaciju, omogućili bi posebnu vrstu korisnika VIPKorisnik, koji je uplatio godišnju kartu, te ima posebne pogodnosti. Iako bi ova dva korisnika bila na različitim nivoima, pristupalo bi im se na isti način i implementacija bi bila pojednostavljena.