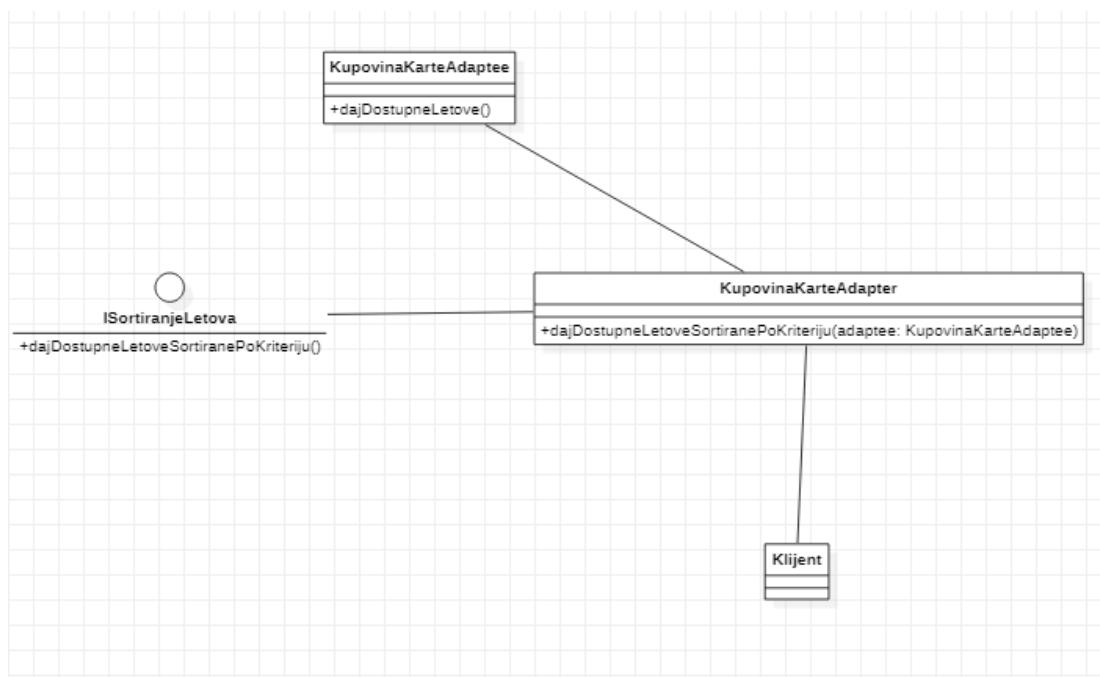


# STRUKTURALNI PATTERNI – SOFTWARE

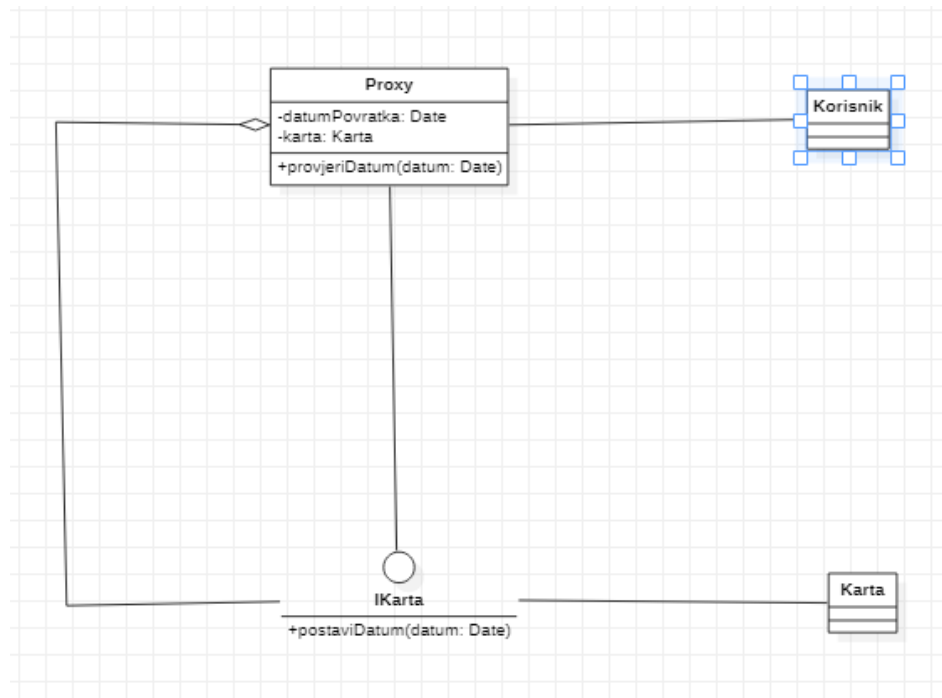
1. **Adapter Pattern** koristimo kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu.

Ukoliko želimo nadograditi naš sistem tako da korisniku omogućimo mogućnost da vrši prikaz sortiranih letova po nekom kriteriju (npr. vrijeme polaska, cijena, itd...). Dakle pored postojećih metoda koje vraćaju listu letova, možemo izvršiti i ovu nadogradnju koristeći Adapter pattern. To ćemo izvršiti na sljedeći način (prikazano u kratkim crtama):

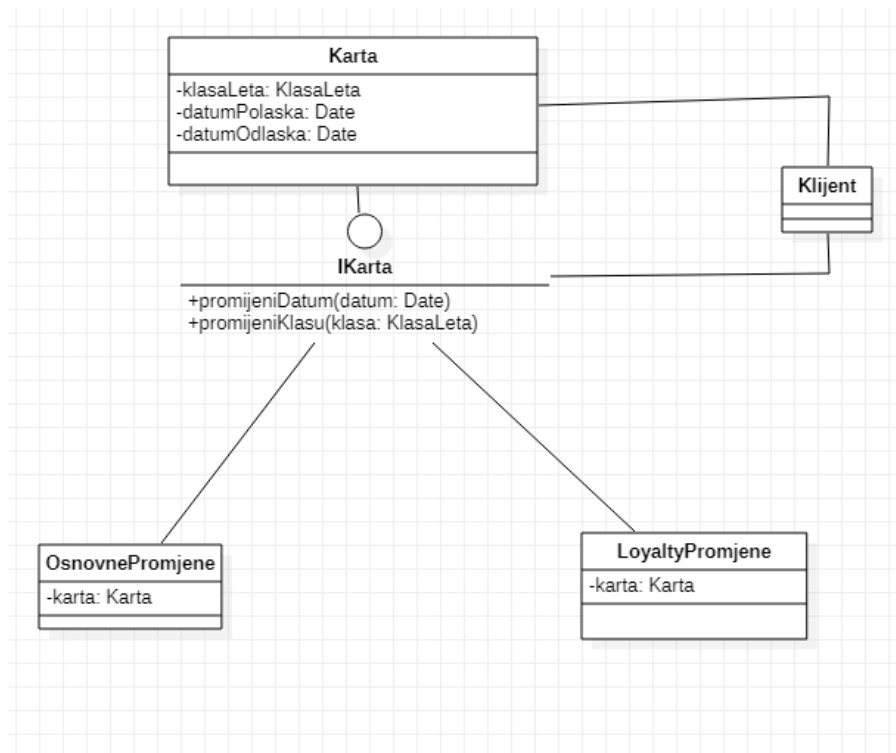
- U našem sistemu postoji klasa KupovinaKarte koja ima metodu `dajDostupneLetove()`, te će zbog toga ona biti naša Adaptee klasa tj. nju je potrebno adaptirati u cilju dostizanja željenog interfejsa.
- Definirat ćemo novi interfejs `ISortiranjeLetova` sa nekim metodama koje uz vraćanje letova, vrše i sortiranje po određenom kriteriju.
- Također ćemo definisati adapter klasu `KupovinaKarteAdapter`, koja će implementirati gore navedeni interfejs. U nekoj od metoda novog interfejsa npr `dajDostupneLetoveSortiranePoCijeni()`, pozivamo metodu `dajDostupneLetove()` adaptee klase (`KupovinaKarte`), te prilagođavamo rezultat te metode kako bi unaprijedili naš interfejs.



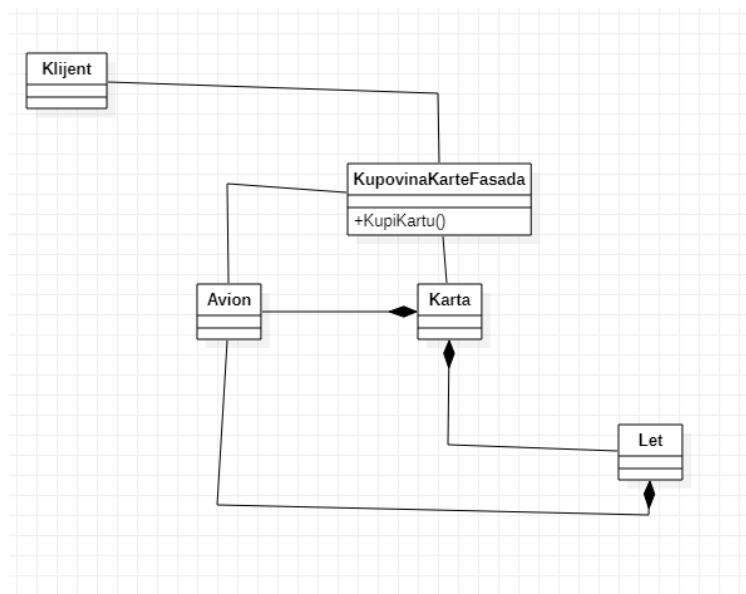
2. **Proxy pattern** koristimo kada želimo omogućiti dodatnu sigurnost u sistemu. Namjena Proxy paterna je da omogućiti pristup i kontrolu pristupa objektima ili metodama. Proxy je obično mali javni surogat objekat koji predstavlja kompleksni objekat čija aktivizacija se postiže na osnovu postavljenih pravila. Sto se tice proxy paterna, mozemo ga iskoristiti u nasem sistemu tako sto prilikom odabira datuma za povratnu kartu, onemogućimo korisniku da izabere datum povratka prije datuma polaska, te mu tako onemogućimo kupovinu te karte.



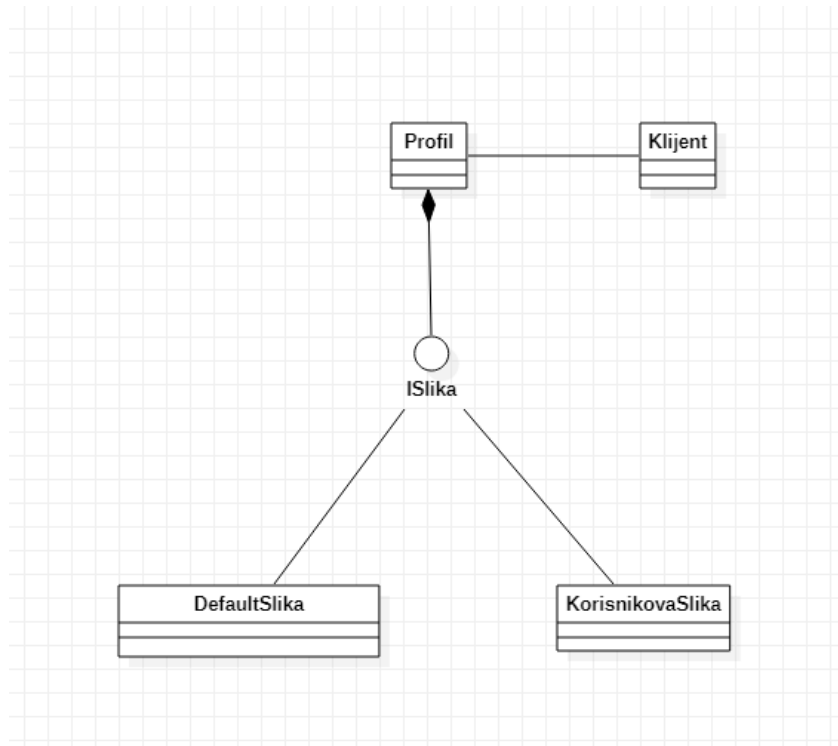
3. **Decorator pattern** koristimo prilikom dinamičkog dodavanja novih elemenata i ponašanja postojećim objektima. Omogućavaju se različite nadogradnje objektima koji svi u osnovi predstavljaju jednu vrstu objekta, bolje reći imaju osnovu. Umjesto da se definiše veliki broj izvedenih klasa, dovoljno 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 trenutnoj fazi projekta ne primjećujemo neku potrebu za korištenjem ovog patterna. Potencijalno u budućnosti mozemo ga iskoristiti u nasem sistemu tako sto bi omogućili korisniku da izvrši promjene nad kupljenom kartom.



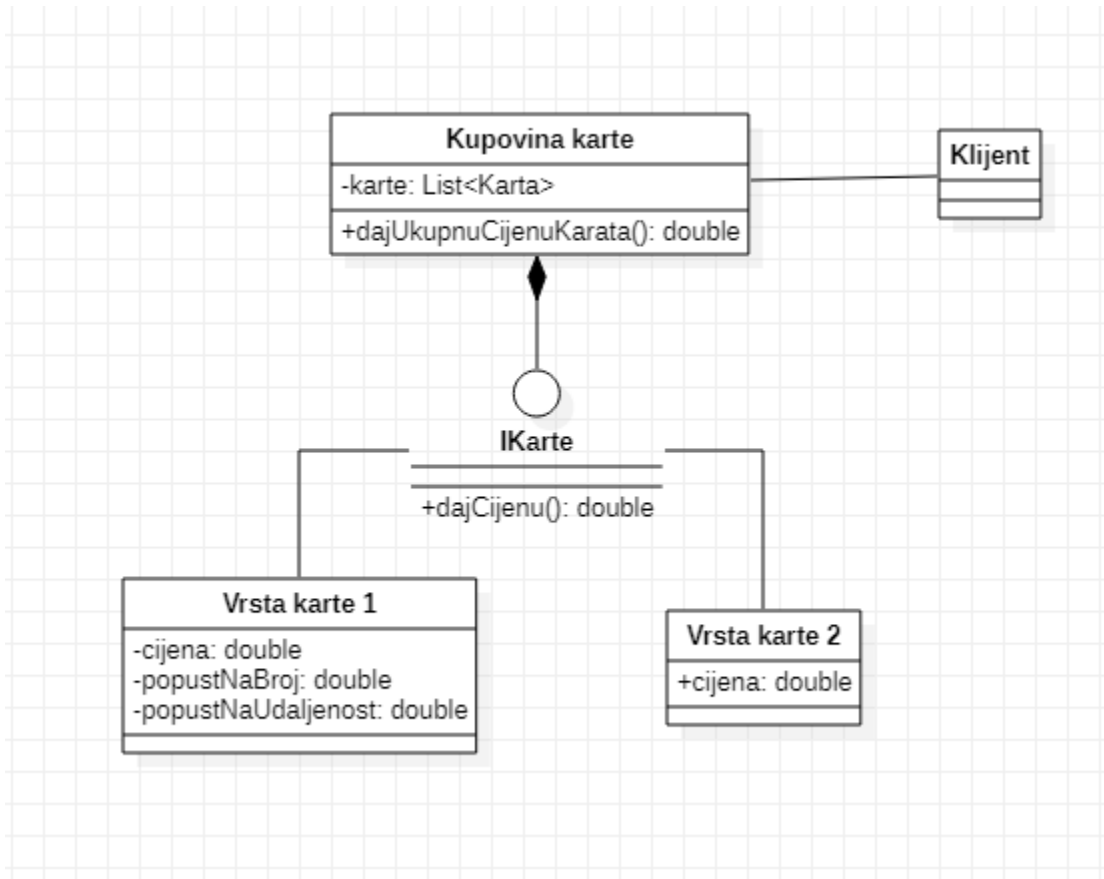
4. **Facade pattern** se koristi da bi se korisnicima pojednostavilo korištenje složenog sistema koji se sastoji od više podsistema sa povezanim implementacijama. Gledajući naš sistem, kao i primjer sa tutorijala, već smo iskoristili ovaj patern. Klasa Avion je sastavni dio klase Let, kao sto je i klasa Let sastavni dio klase Karte, itd...Složeni sistem KupovinaKarte je razdvojen na više malih podsistema, stoga omogućava lakše proširivanje sistema.



5. **Flyweight pattern** je strukturalni patern koji bi mogli iskoristiti kada bi naši modeli imale neku osobinu koja se naknadno postavlja, te se toj osobini dodjeljuje neka default vrijednost koja se nalazi na samom sistemu radi smanjenog korištenja memorije. Spomenuta osobina nije značajan factor za samo odvijanje aplikacije. Trenutno dizajnirani sistem koristi samo podatke koji su zaista potrebni aplikaciji, te iz tog razloga ne vidimo neku potrebu za ovim paternom. Međutim, mogli bi ovaj patern iskoristiti za default sliku prilikom kreiranja novog korisničkog računa ili default sliku prilikom uvođenja nove lokacije u naš sistem od strane administratora.



6. **Composite pattern** služi da bismo postigli hijerarhiju objekata, tako što ćemo kreirati strukturu stabla pomoću klasa, u kojoj se kompozicije individualnih objekata (korijeni stabla) i individualni objekti (listovi stabla) ravnopravno tretiraju, odnosno moguće je pozvati zajedničku metodu nad svim klasama. Kao ideju za ovaj patern, možemo omogućiti da kada se korisniku pokaze konacni racun za kupovinu karata, cijena se racuna drugacije u zavisnosti od smjera karte, te broja karata. Potrebno je još naglasiti da bi se u tom slucaju morale napraviti posebne klase za razlicite vrste karti koje bi onda na razlicite nacine implementirale metodu za kalkulaciju cijene.



7. **Bridge pattern** se koristi da bi se omogućilo odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije. Bridge pattern pogodan je kada se implementira nova verzija softvera a postojeća mora ostati u funkciji. U našem sistemu bi to mogli uraditi na sljedeći način: Klasa KupovinaKarte sadrži instrukciju `dajCijenu()` koju bi mogli proširiti sa interfejsom Bridge koji bi u sebi imao istoimenu metodu, te bi na osnovu toga jel korisnik član loyalty club-a ili ne pozivao dvije različite implementacije navedene metode gdje bi se u jednom slučaju vraćala normalna cijena, a u drugom slučaju snižena cijena pod određenim popustom.

