STRUKTURALNI PATTERNI – SOFTWAIR

1. **Adapter Patern** 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, mozemo izvršiti i ovu nadogradnju koristeći Adapter pattern. To ćemo izvrstiti 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.
   - Definisat ćemo novi interfejs sortiranjeLetova sa nekim metodama koje uz vraćanje letova, vrše i sortiranje po određenom kriteriju.
   - Također ćemo definisati adapter klasu KupovinaKarteAdapter, koja će implementirani 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 omogući 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. Kao ideju gdje bi mogli iskoristiti ovaj pattern jeste prilikom prijave korisnika. Potencijalno bi mogli implementirati klasu ProvjeraProxy koja bi u sebi sadržavala metodu koja bi odobravala ili odbijala pristupanje korisničkom računu ukoliko neki podatak nije tačan.

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 koristenjem ovog patterna. Bilo bi potrebno omogućiti

neke dodatne funkcionalnosti rukovodiocima/administratorima kao npr. neke funkcije koje vrše manipulacije sa slikom kao što imamo u wordu (rotate, crop…)

4. **Facade patern** 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, mogli bismo na sličan način iskoristiti facade patern gdje bi redom na mjestu klasa Tačka, Linija, Krug, Četverougao GeometrijskiOblikFacade bili naše klase Avion, Let, Karta, KupovinaKarteFacade sa metodom kupiKartu().
   Da napomenemo, ovo je samo način kako bi mogli iskoristiti ovaj patern. Zasad ćemo se zadržati na trenutnom dizajnu.

5. **Flyweight patern** 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 koristenja 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 patern** 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.

7. **Bridge patern** 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 patern 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.