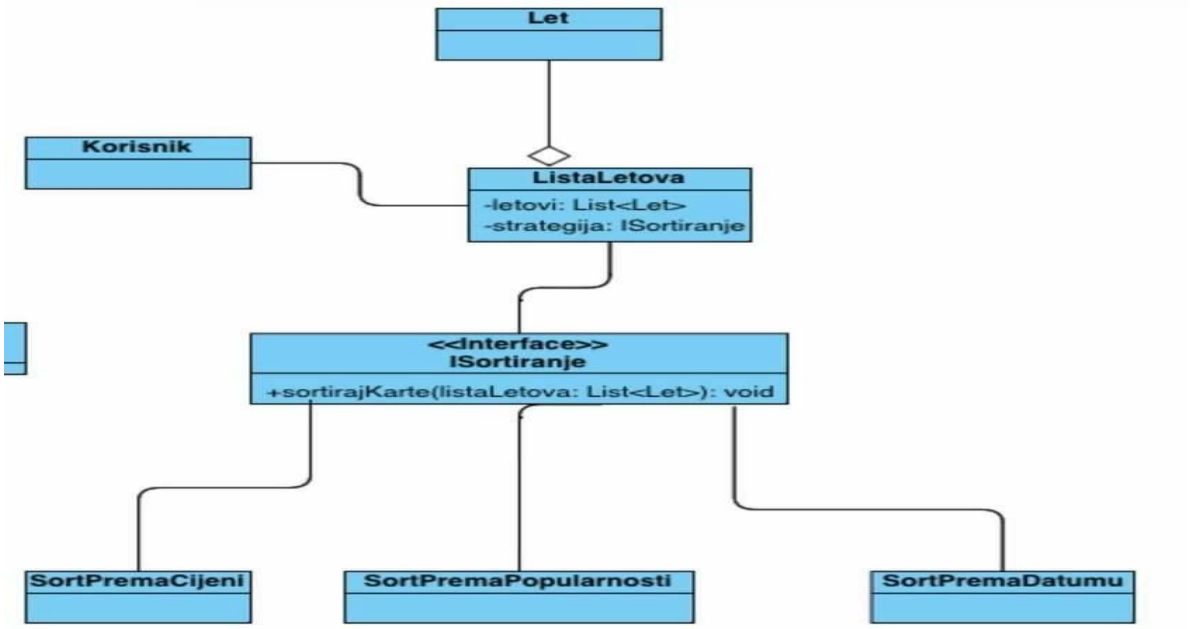
Paterni Ponašanja

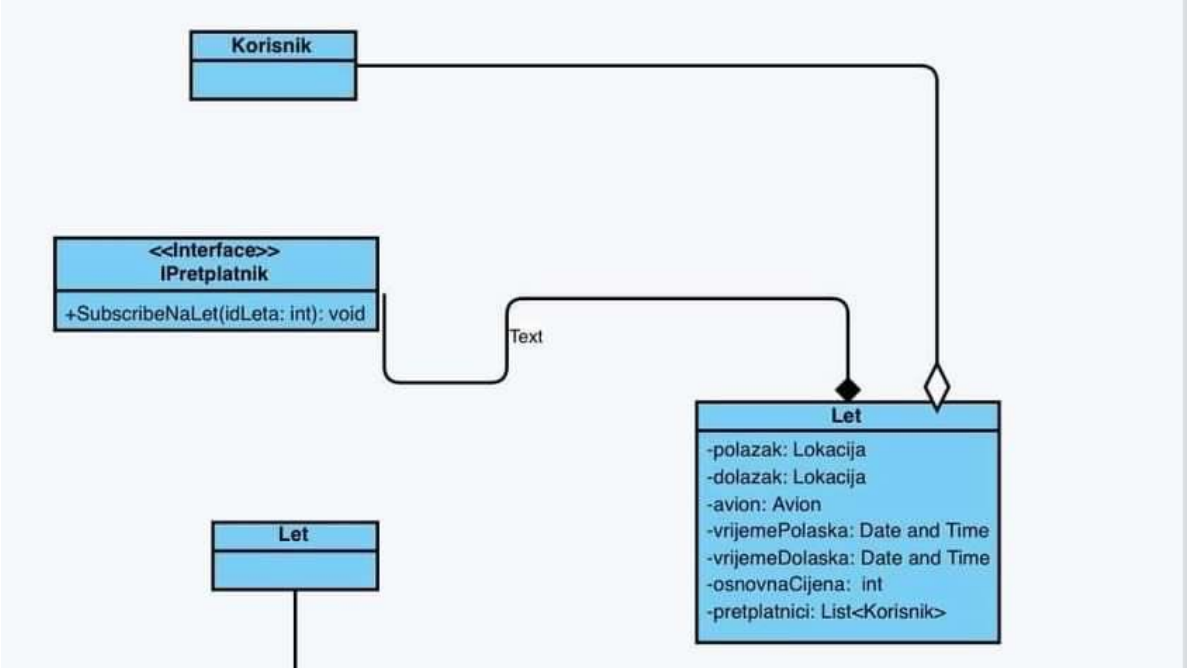
1. **Iterator pattern** - ovaj pattern se koristi kada je potrebno imati uniforman način pristupa bilo kojoj kolekciji. Ako recimo želimo iz nekog razloga da primimo ArrayList, Array i HashMapu, možemo iskoristiti iterator interface pomoću kojeg ćemo najbolje omogućiti uniforman pristup, skratiti kod, napraviti bolji polimorfizam. Ukoliko imamo rad sa više vrsta kolekcija implementacija ovog paterna je veoma korisna. Pošto u našoj implementaciji imamo samo listu, upotreba ovog paterna nije od nekog značaja. Također ovaj pattern se može biti od koristi ukoliko korisnik ima potrebu pretraživati letove na temelju određenih kriterija. Iterator pattern može biti koristan za iteriranje kroz rezultate pretrage. Mogao bi se iskoristiti u nekoj posebnoj klasi koja bi imala realizovan Iterator interface i imala npr. metodu koja će raditi nešto sa elementima svih kolekcija. Svejedno, patern se mogao iskoristiti za prolazak kroz listu putnika unutar klase Let.

2. **Strategy pattern** – ovaj pattern izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Strategy patern je koristan kada postoje različiti primjenjivi algoritmi (strategije) za neki problem. Ukoliko imamo ograničeno, ali obavezano ponašanje različitih podklasa u hijerarhiji, tada je ovaj patern zgodno iskoristiti. Strategy patern omogućava klijentu izbor jednog od algoritma iz familije algoritama za korištenje. Prednost ovog paterna je što se mogu definirati dodatne metode za nove strategije neovisno od ostalih klasa. U našem sisitemu ovaj pattern možemo implementirati tako da omogućimo korisniku prikaz letova sortiranih po određenim kategorijama (cijeni ili datumu polijetanja). Što bi značilo da ćemo definisati interfejs SortiranjeStrategija s metodom poput sortirajLetove() koja će sortirati listu letova na temelju odabranog kriterija. Na osnovu predhodnog možemo implementirati konkretne strategije kao SortiranjePoCijeni, ili SortiranjePoPopularnosti, koje će implementirati metodu sortirajLetove() na odgovarajući način. Naša web aplikacija može dinamički odabrati i primijeniti odgovarajuću strategiju sortiranja proizvoda na temelju korisničkog odabira.



3. **State pattern** – ovaj pattern predstavlja dinamičku verziju Strategy patterna gdje objekat mijenja način ponašanja na osnovu trenutnog stanja. Postiže se promjenom podklase unutar hijerarhije klasa. Ovaj pattern je jako koristan i iskoristili smo ga u našem sistemu za određivanje stanja leta, tj da li je let popunjen, odgođen itd. Ovo bismo implementirali dodavanjem interfejsa IStanje sa metodom dajStanje(), koju bi implementirale klase Popunjen, Odgođen i TrebaPoletjeti.

4. **Observer pattern** - Observer patern koristimo kada je potrebno uspostaviti relaciju između objekata tako da kada jedan objekat promijeni stanje, drugi zainteresirani objekti dobijaju obavijest o promjeni. Ovaj patern smo primjenili tako da za svaku promjenu stanja leta putnici dobivaju obavijest o tome. Implementirali smo ga dodavanjem interfejsa IObavijest sa metodom Azuriraj(). U klasi Putnik smo dodali metodu Azuriraj() koja azurira atribut za stanje leta. Također ovaj pattern možemo koristiti ukoliko korisnik želi da se pretplati na primanje novosti u vezi najnovijih letova. Svaki korisnik može biti promatrač koji je pretplaćen na subjekt (novosti) i kada se pojave nove vijesti, subjekt će automatski obavijestiti sve promatrače.



5. **TemplateMethod Pattern** – ovaj pattern mogućava izdvajanje određenih koraka algoritma u odvojene podklase. Struktura algoritma se ne mijenja - mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito. Ovaj patern u našoj web aplikaciji koristimo za slanje različite vrste e-mail poruka kao što su: potvrda rezervacije leta ili reklamne poruke. Na osnovu ovoga možemo koristiti Template method. Možemo definisati apstraktnu klasu EmailTemplate koja će sadržavati osnovnu strukturu i metode za generiranje email poruka, poput generirajNaslov() ili generirajSadržaj(). Zatim možemo implementirati konkretne klase za svaku vrstu e-mail poruke koje će nasljeđivati EmailTemplate i pružiti specifične implementacije metoda prema vrsti poruke. Na taj način ćemo imati zajednički template za generiranje e-mail poruka, ali će se specifični detalji poruke razlikovati za svaku vrstu. Također ovaj pattern se može koristiti u sistemu ako postoje podjele registrovanog korisnika, na običnog korisnika i VIPKorisnika (što nije slučaj u našem sisitemu), kako bi regulisali različite načine plaćanja, s obzirom da bi VIPKorisnik imao neke pogodnosti.

6. **Chain of responsibility** – ovaj pattern je moguće iskoristiti na sljedeći način:
   - Obrada rezervacije letova: Kada korisnik napravi rezervaciju leta, možemo koristiti Chain of Responsibility pattern za obradu te narudžbe kroz različite korake. Svaki korak u lancu može predstavljati određenu vrstu obrade, poput provjere dostupnosti leta, provjere podataka o plaćanju i sl. Svaki objekat u lancu ima svoju logiku za obradu određenog koraka, a ako ne može obraditi rezervacija, prenosi je sljedećem objektu u lancu. Na taj način možemo uspostaviti fleksibilan sistem za obradu rezervacije letova.

7. **Mediator** – ovaj pattern je moguće iskoristiti na sljedeće načine:
   -Obrada letova: Kada korisnik napravi rezervaciju leta, Mediator pattern može poslužiti kao objekat posrednika koji će koordinirati obradu te rezervacije između različitih podsistema u sistemu. Na primjer, posrednik može obavijestiti aerodrom o novoj rezervaciji, obračunati plaćanje, poslati obavijest kupcu itd.
   -Promjena rezervacije: Naša web aplikacija omogućava korisnicima promjenu rezervacije leta. Mediator pattern može poslužiti kao objekat posrednika koji će upravljati tim promjenama. Na primjer, kada korisnik odabere određeni let, posrednik će obavijestiti objekte u sistemu. Ovo omogućuje koordinaciju promjenama rezervacije letova.