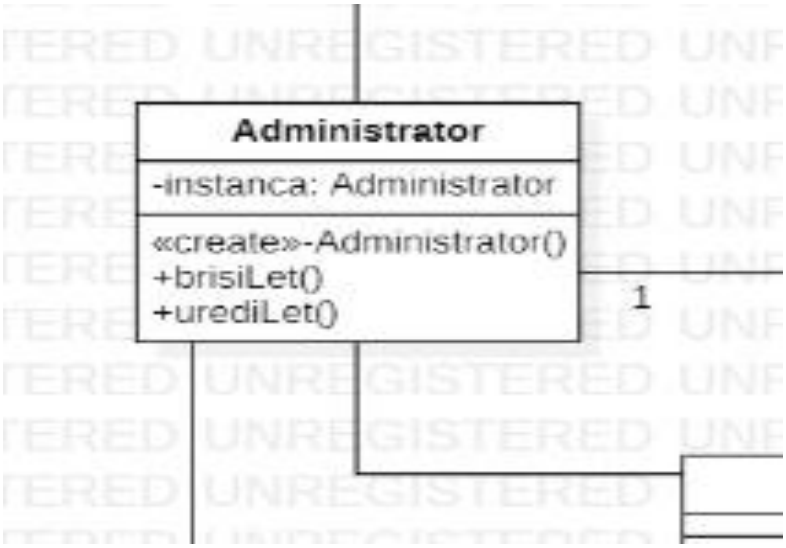
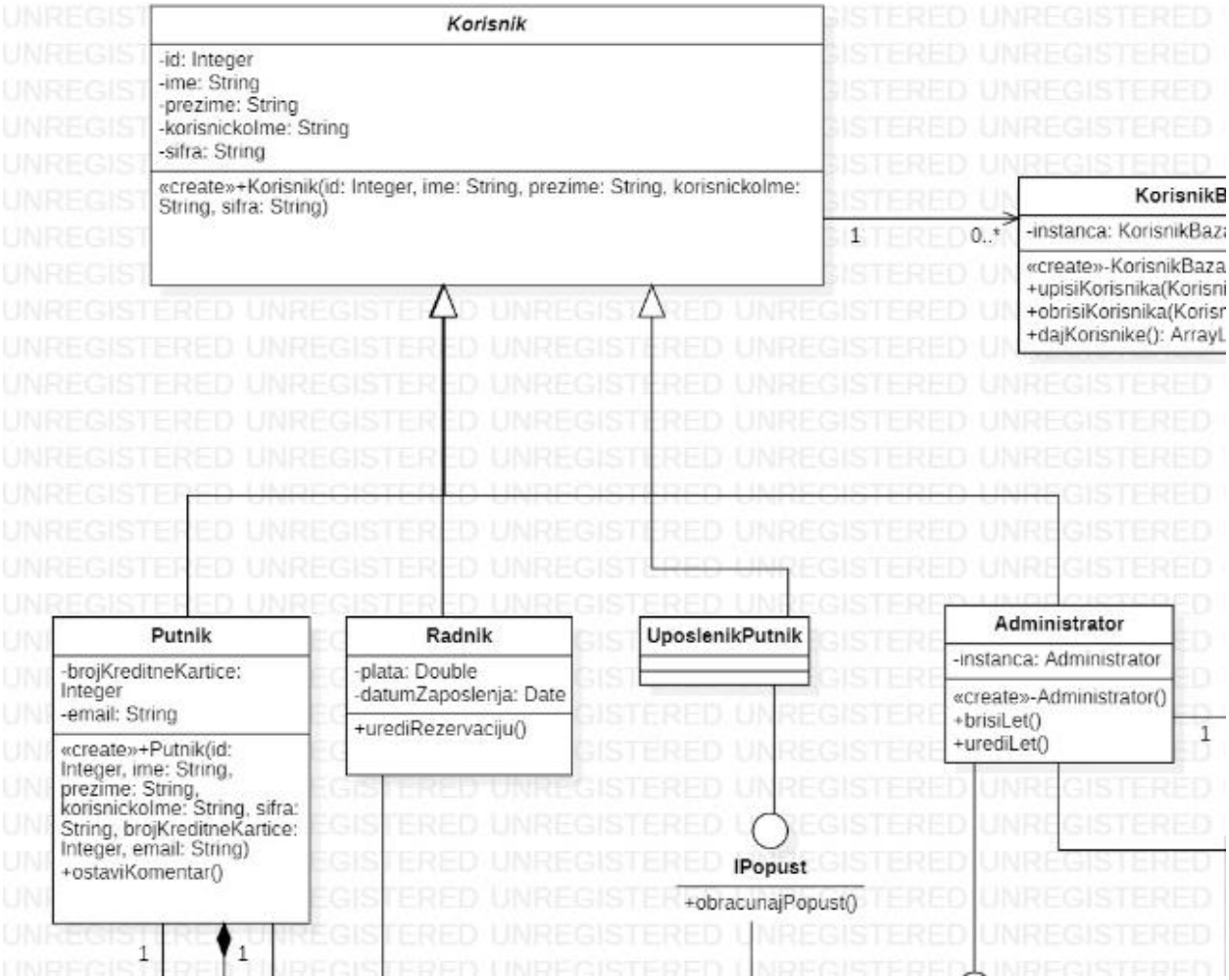
# Kreacijski paterni

- **Singleton patern** - uloga Singleton paterna je da osigura da se klasa može instancirati samo jednom i da osigura globalni pristup kreiranoj instanci klase. Instanciranjem više od jednog objekta mogu nastati problemi. Singleton klasa sadrži mehanizam za jedinstveno instanciranje same sebe. Unutar klase je private static varijabla (uniqueInstance) koja čuva jednu/jedinstvenu instancu klase, static metoda (getInstance) preko koje se pristupa Singleton klasi. Važan dio Singleton paterna je inicijalizacija resursa u Singleton konstruktoru. U našem sistemu Singleton patern je implementiran u klasi Administrator tako što smo dodali private static konstruktor i private static varijablu (jedinstvenaInstanca) koja čuva jednu/jedinstvenu instancu klase. Također sve klase za rad sa bazama podataka su singleton, u suprotnom bi imali problema sa drajverima na bazu.




- **Abstract factory patern** – ovaj patern služi kako bi se izbjeglo korištenje velikog broja if-else uslova pri kreiranju različitih hijerarhija objekata. Ukoliko postoji više tipova istih objekata te različite klase koriste različite podtipove, te klase postaju fabrike za kreiranje objekata zadanog podtipa bez potrebe za specificiranjem pojedinačnih objekata. Na ovaj način se, korištenjem nasljeđivanja, ukida potreba za postojanjem if-else uslova jer određeni tip fabrike sadrži određene tipove objekata i zna se tačno koju podklasu će instancirati. Ovaj patern smo implementirali kod klasa Korisnik, Putnik, Radnik i Administrator.




- **Prototype patern** - uloga Prototype paterna je da kreira nove objekte klonirajući jednu od postojećih prototip instanci. Time smanjujemo kompleksnost kreiranja novog objekta, posebno kada objekti sadrže veliki broj atributa koji su za većinu instanci isti. Ovaj patern nismo primijenili. Ali bismo to mogli uraditi za kloniranje rezervacije leta uvođenjem interfejsa IPrototip, koji sadrži metodu clone(). Ova metoda će kreirati novu rezervaciju sa istim atributima kao i rezervacija nad kojom je pozvana ova metoda.

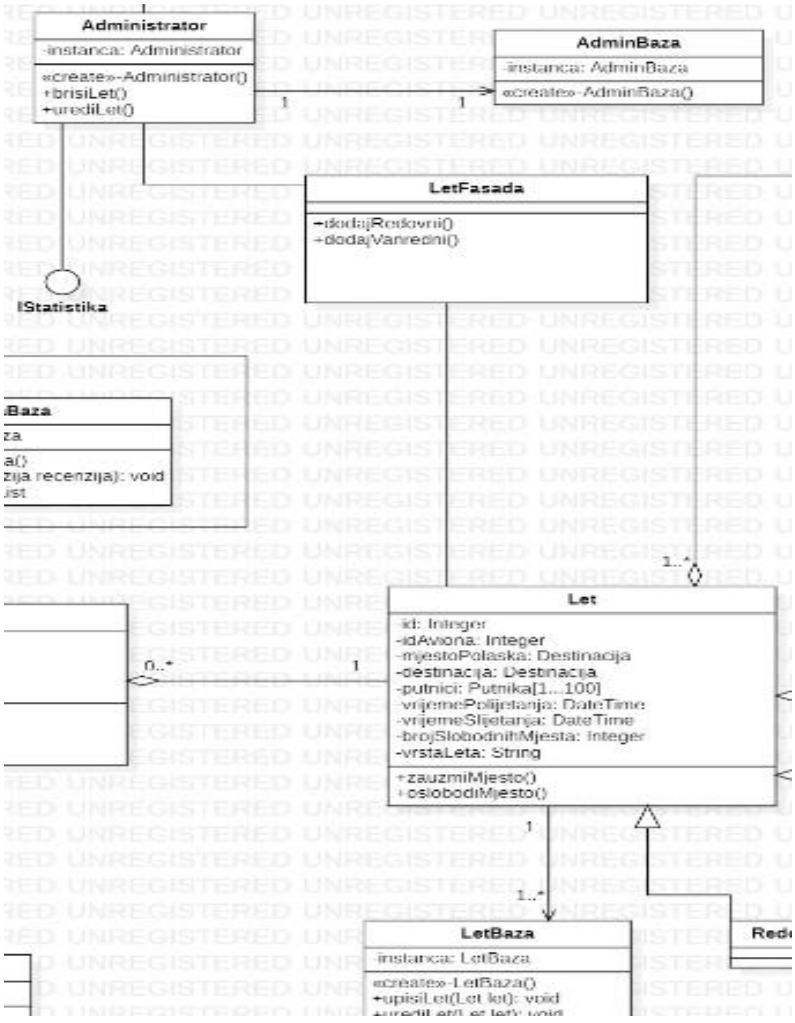
- **Factory Method pattern** - Uloga Factory Method paterna je da omogući kreiranje objekata na način da podklase odluče koju klasu instancirati. Različite podklase mogu na različite načine implementirati interfejs. Factory Method instancira odgovarajuću podklasu(izvedenu klasu) preko posebne metode na osnovu informacije od strane klijenta ili na osnovu tekućeg stanja. Factory patterne (method/abstract) koristimo kad želimo napraviti da nam je kreiranje instance objekata modularno i da jednostavno možemo dodavati nove tipove klasa koje želimo kreirati. Ovaj patern nismo iskorisitili u našem projektu. Bilo bi ga moguće primijeniti kada bismo imali više nasljeđivanja u

sistemu, npr. kada bismo različite tipove aviona predstavljali preko izvedenih klasa. U tom slučaju, svaka od podklasa klase Creator koja može instancirati neku od klasa za tipove aviona, implementirala bi factory metodu koja bi zavisno od toga koju opciju je izabrao korisnik, u toku izvršavanja aplikacije, instancirala odgovarajuću klasu.

- **Builder pattern** - služi za apstrakciju procesa konstrukcije objekta, kako bi se kao rezultat mogle dobiti različite specifikacije objekta koristeći isti proces konstrukcije. Ovaj patern nam omogućava da proizvedemo različite tipove i reprezentacije objekata koristeći isti konstrukcijski kod. U našem projektu nismo koristili ovaj patern, ali bismo ga mogli iskoristiti da smo imali funkcionalnost da klijent odabere da li želi puni pansion tokom putovanja.

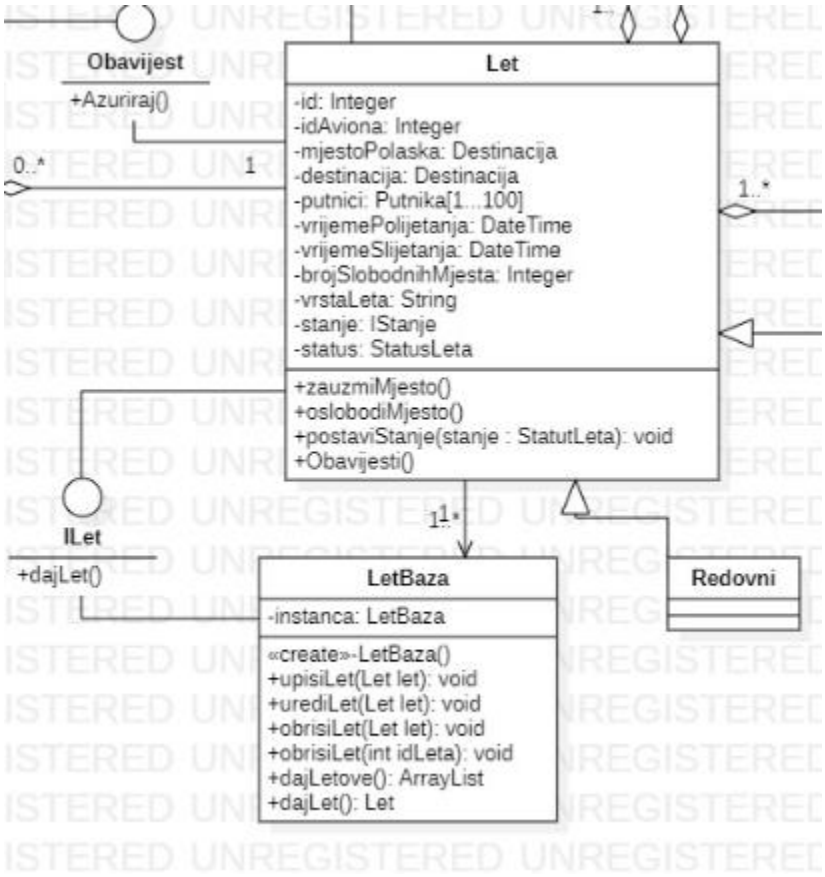
# Strukturalni paterni

- **Fasade patern** - fasadni patern služi kako bi se klijentima pojednostavilo korištenje kompleksnih sistema. 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 dijagramu smo ovaj patern implementirali kod kreiranja leta u administratorskoj klasi. Klasa administrator je klijent I ona putem metoda kreirajRedovniLet() i kreirajVandredniLet() iz klase LetFasada dodaje letove u bazu.




- **Flyweight pattern** - koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji svi u suštini predstavljaju jedan objekat. Samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (tzv. specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (tzv. bezlično stanje). Korištenje ovog paterna veoma je

korisno u slučajevima kada je potrebno vršiti uštedu memorije. Ovaj patern smo implementirali kod kreiranja novog leta. Svaki put kada admin želi dodati novi let koji je sličan nekom već ranije kreiranom letu, ne mora se ponovo kreirati taj novi let nego se može vratiti iz baze let koji je sličan i samo urediti određene atribute. Ovo smo implementirali tako što smo dodali interfejs ILet sa metodom dajLet(), također smo I u bazu podataka dodali metodu dajLet().




- **Adapter patern** – osnovna uloga Adapter paterna je da omogući širu upotrebu već postojećih klasa. U situacijama kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu koristi se Adapter patern. Adapter patern kreira novu adapter klasu koja služi kao posrednik između originalne klase i željenog interfejsa. Tim postupkom se dobija željena funkcionalnost bez izmjena na originalnoj klasi i bez ugrožavanja integriteta cijele aplikacije. Adapter patern smo u našoj aplikaciji mogli primjeniti za slučaj kada se radnik aviokompanije odluči za putovanje. Tako bi on imao pravo na popust. Adapter patern bismo implemetirali kreiranjem interfejsa IPopust sa metodom obracunajPopust() i dodavanjem klasa UposlenikPutnik i RezervacijaSaPopustom. Tako omogućavamo radniku aviokompanije pravo na popust bez izmjene postojećeg objekta.
- **Decorator patern** - koristi se da omogući dinamičko dodavanje novih elemenata i funkcionalnosti postojećim objektima i pri tome objekat ne zna da je urađena dekoracija

što je veoma korisno za iskoristljivost i ponovnu upotrebu komponenti softverskog sistema. Pattern bismo mogli u našoj aplikaciji implementirati kod kupovine karata gdje bi uvijek moglo nešto novo da se doda, npr. neka nova funkcionalnost, dinamički, bez da poremetimo rad trenutne verzije aplikacije.

- **Bridge patern** - njegova osnovna namjena je da omogući odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije. 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 programu ovaj patern nije primijenjen, te mislimo da nemamo mogućnost primijene istog.

- **Proxy pattern** - Proxy patern služi za dodatno osiguravanje objekata od pogrešne ili zlonamjerne upotrebe. Primjenom ovog paterna omogućava se kontrola pristupa objektima, te se onemogućava manipulacija objektima ukoliko neki uslov nije ispunjen, odnosno ukoliko korisnik nema prava pristupa traženom objektu. Ovaj patern nismo implementirali ali bismo ga vrlo lahko mogli implementirati za kontrolu pristupa određenim dijelovima aplikacije, kao što je pravo pristupa bazama podataka ili parvo pristupa izvještaju.

- **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. Ovaj patern nismo implementirali ali kada bismo se odlučili da proširimo aplikaciju, mogli bismo ga implementirati uvođenjem posebnih vrsta korisnika, VIPKorisnik, koji ima posebne pogodnosti. Iako bi ova dva korisnika bila na različitim nivoima, pristupalo bi im se na isti način i implementacija bi bila pojednostavljena.