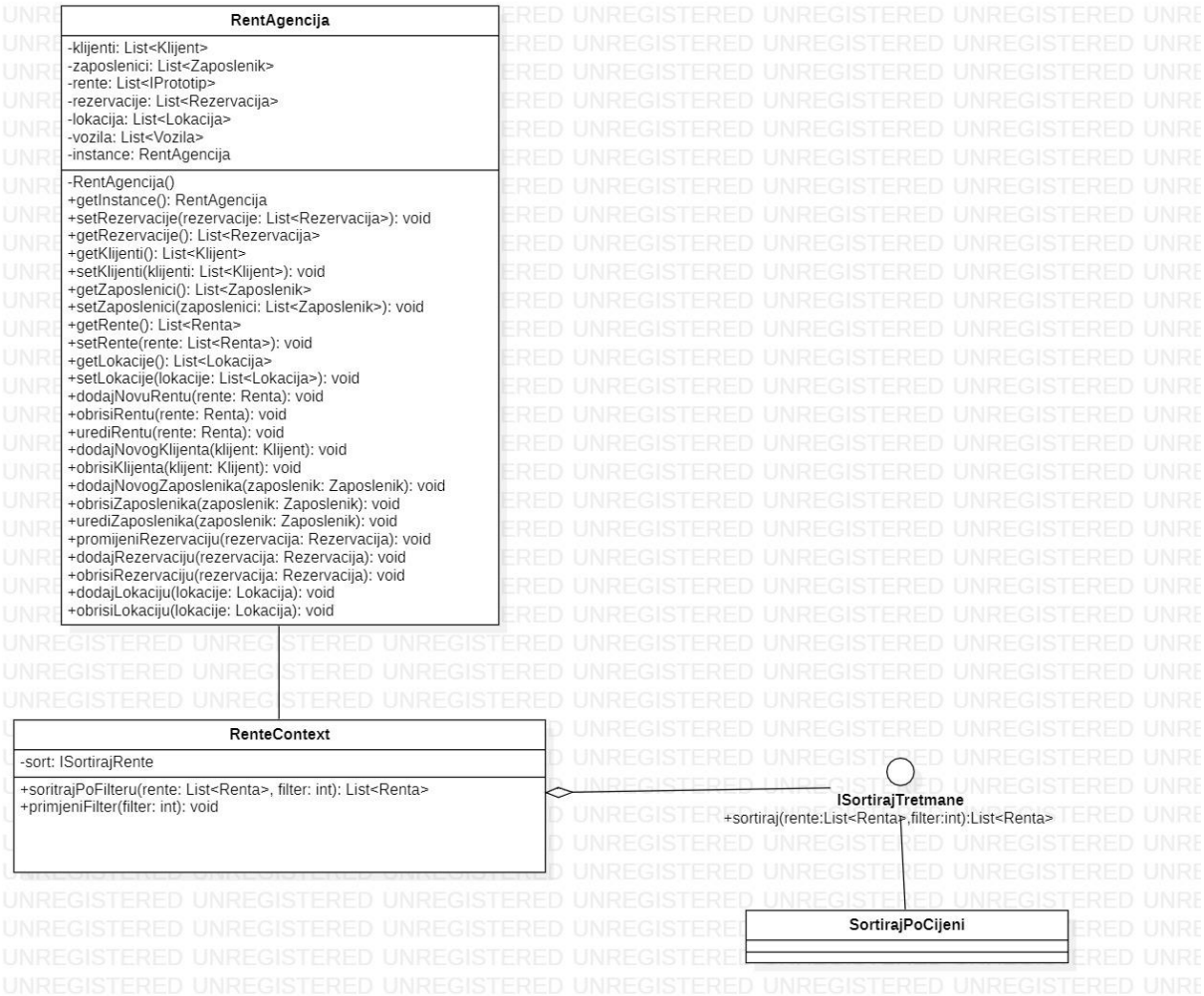
Paterni ponašanja

1.Strategy pattern

Strategy patern izdvaja algoritam iz matične klase i uključuje ga u posebne klase.Pogodan je kada postoje različiti primjenjivi algoritmi za neki problem.Strategy patern omogućava klijentu izbor jednog od algoritama iz familije algoritama za korištenje.

Strategy patern u našem sistemu smo primjenili tako što bi klijentu dodijelili mogućnost prikaza rente automobila sortirane po cijenama auta.

Dodali bi klasu RentaContext koja će interfejsu ISoritirajRente proslijediti neophodne podatke.Ovaj interfejs ima metodu sortiraj (List<Rente> rente, int cijena):List<Rente>, koju će implementirati klasa SortirajPoCijeni u kojima se vrši sortiranje po samom tipu filtera.Klasa RentaContext kao privatni atribut ima objekt tip ISortirajRente te metode sortirajPoFilteru(List<Renta>rente,int cijena):List<Renta> i promijeniFIlter(int filter).



2.State pattern

State Pattern je dinamička verzija Strategy paterna.Objekat mijenja način ponašanja na osnovu trenutnog stanja.Postiže se promjenom podklase unutar hijerarhije klasa.

Ovaj patern nismo primjenili u našem sistemu.Kada bismo imali mogučnost pri rezervaciji automobila izabrati želimo li jeftini , srednje jeftin ili skup automobil mogli bismo primijeniti navedeni patern.Imali bismo interfejs IState , koji ima metodu promjeniStatus i prima parametar tipa Context.

Dodali bi i klasu Context koja bi kao atribute imala status rente tipa IState ,statusRente tipa String i metodu operacijaPromjene(String buduceStanje) u kojoj bismo trenutno stanje promijenili u buduće. Stanje rente prije rezervacije se smatra trenutnim stanjem i ono je na početku uvijek „slobodno“ (statusRente) .

Također , naš sistem bi imao još 3 klase : CheapCarRent, MediumCarRent, ExpensiveCarRent koje bi implemenitrali interfejs IState i koje bi unutar sebe mijenjali vrijednost atributa StatusRente.

3.Template Method pattern

Template method pattern omogućava izdvajanje određenih koraka algoritma u odvojene podklase. Struktura algoritma se ne mijenja tj. mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

Ovaj patern također nismo primijenili, ali kada bismo uveli mogućnost plačanja kroz sistem mogli bismo ga primijeniti na sljedeći način. Dodali bi klasu Placanje koja bi u sebi sadržavala metodu void plati(IObracun x),gdje je IObracun interfejs. Ova metoda obračunava cijenu koliko klijent treba platiti za svoje rentanje automobila. Imali bismo klase Klijent i VipKlijent(koje implementiraju intefejs IObracun),pa ta cijena ne bi bila ista u ovisnosti koji je klijent u pitanju(VipKlijent uvijek ima popust npr 10%).

Interfejs IObracun bi imao metode String Operacija1() i String Operacija2() koje bi računale cijenu tretmana i popusta repsektivno. Ove dvije metode bi se pozivale iz metode plati nad njenim parametrom.Operacija1 bi vraćala cijenu rentanog automobila i ona bi bila ista kod obje vrste klijenata ,dok bi operacija2 vraćala nulu u klasi Klijent, a u klasi VipKlijent vrijednost recimo 10% cijene.

U metodi plati bismo mogli recimo napisati npr.

Double ukupno=x.Operacija1()-x.Operacija2();

4.Observer pattern

Observer patern uspostavlja relaciju između objekata tako kada jedan objekat promijeni stanje drugi „zainteresirani“ objekti se obavještavaju.Ovaj patern nismo primjenili u našem sistemu pa ćemo ga pokušati primjerom razmotriti.Želim omogućiti da se pri promjeni cijene usluge rentanja automobila obavijeste korisnici.U klasu Renta dodat ćemo metodu obavještenje(List<Korisnik>) koja će primati listu korisnika kojima će se u slučaju promjene cijene usluge slati obavještenje.Dodatno ćemo kreirati i iterfejs IKorisnik koji će imati jednu metodu update() koja će biti implementirana u klasi Korisnik i izvršavati promjenu cijena usluga koje korisnik ima nakon što se izvrši promjena cijene.

5.Iterator pattern

Iterator patern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolecija struktuirana. Ovaj patern nismo primjenili u našem sistemu pa ćemo ga razmotriti čisto hipotetički.Zamislimo da želimo proći kroz sve zaposlenike rent-a-car agencije, ali da nemamo interfejs IZaposlenici i kolekciju tih instanci sačuvanih u listi u klasi VlasnikSingelton.

Napravili bismo klasu Zaposlenici što bi simuliralo klasu Collection i u njoj bi čuvali sve zaposlenike u rent-a-car agenciji.Interfejs IZaposleni simulirao bi interfejs IEnumerable i u sebi bi imao metodu getZaposleni koju implementira klasa Zaposleni, a ona bi simuliralarad metode getEnumerator() tj. pružala bi vrijednost kolekcije u sekvenci.U klasi VlasnikSingeltone čuvali bismo kolekciju npr. Listu zaposlenih i u njoj bismo mogli kroz neku metodu ili funkciju unutar klase prolaziti foreach petljom i obavljati neku akciju koju želimo.