Strukturalni paterni

1. Adapter patern

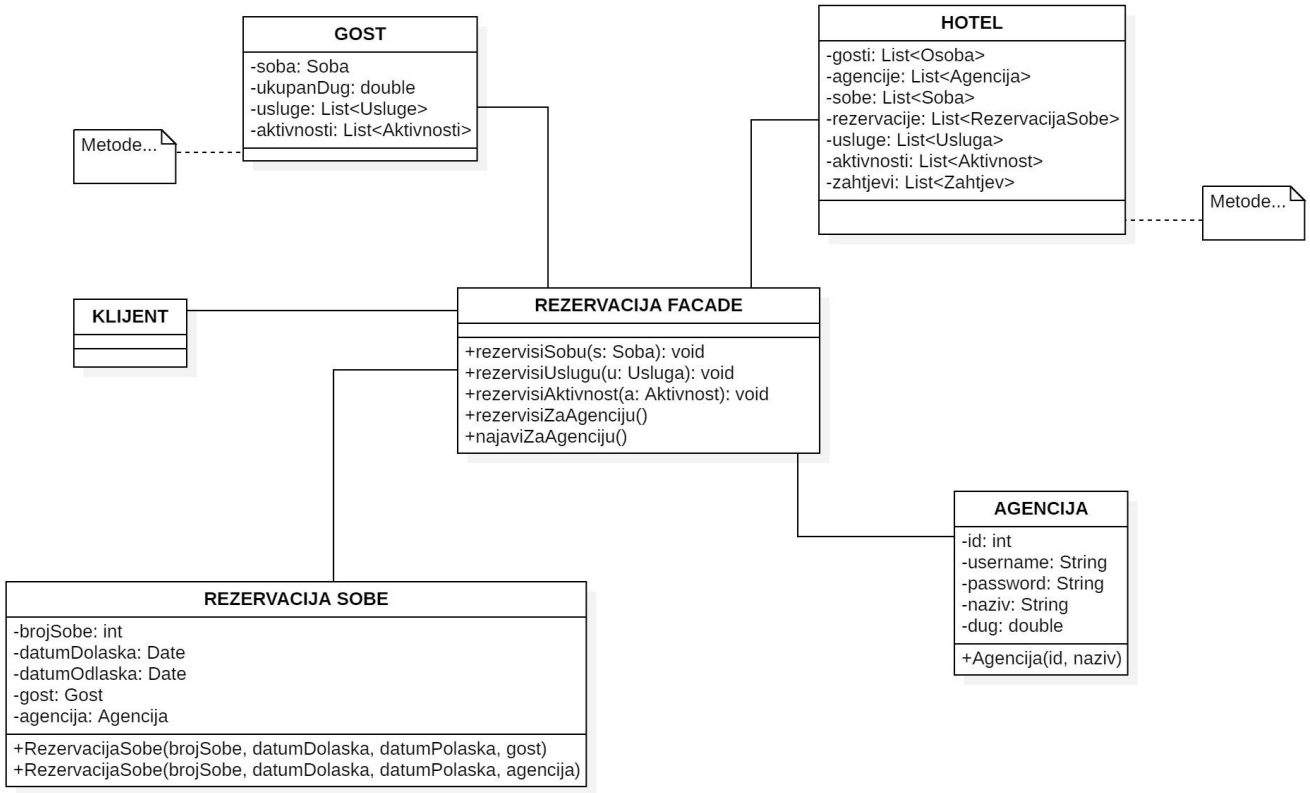
Adapter patern omogućava da se interfejs već postojeće klase prilagodi na drugačiji način korištenja, bez promjene definicije klase.

Ovaj patern nismo iskoristili u našem projektu. Da bismo opisali situaciju u kojoj bi se patern mogao iskoristiti, recimo da u sistemu imamo dva različita tipa hotela i da klijent želi da dobije izvještaj o gostima u svakom hotelu. Prvi tip ima metodu koja vraća listu stringova sa imenima i prezimenima svakog gosta, a drugi tip metodu koja vraća listu gostiju. Recimo da klijent želi izvještaj u obliku ispisa imena svih gostiju za oba hotela. Tada bi bilo potrebno definisati interfejs *IPrikaz* sa metodom *dajImenaGostiju (prima objekat tipa drugog hotela)* i klasu *Adapter* koja implementira taj interfejs. U toj metodi bi se lista gostiju konvertovala u listu stringova i onda ispisala na isti način kao i lista stringova koju vraća metoda klase prvog tipa hotela.

2. Facade patern

Facade patern sakriva kompleksnost sistema i omogućava klijentu da koristi sistem preko pojednostavljenog interfejsa bez potrebe da poznaje unutrašnju strukturu sistema.

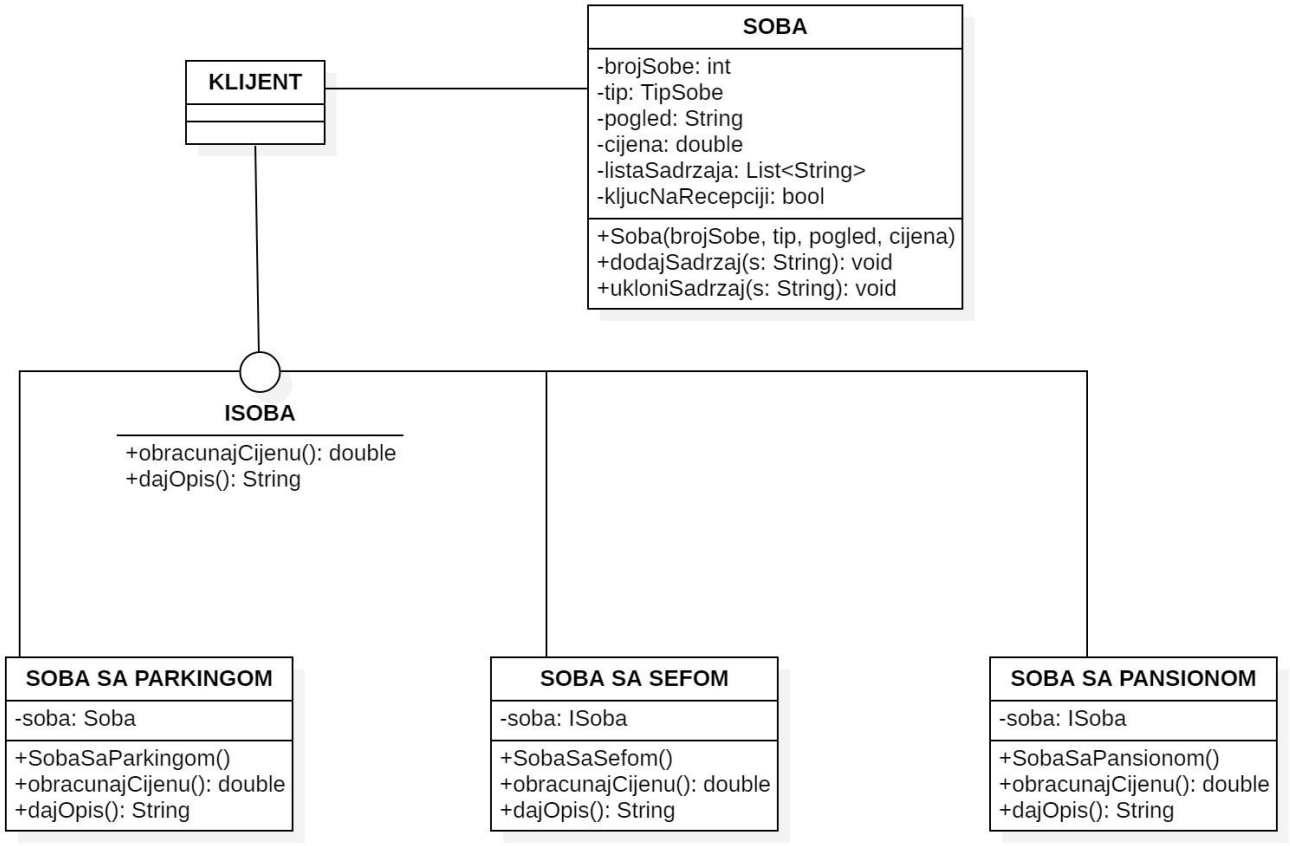
Ovaj patern smo iskoristili u našem projektu. Sve vrste rezervacija (rezervacija sobe, rezervacija usluge, rezervacija aktivnosti, rezervacija za agenciju) vrše se pozivanjem gotovih metoda klase *RezervacijaFacade*. Dakle, klijent ne vidi da se pozivanjem tih metoda dodaje nova rezervacija u klasu *Hotel*, preračunava dug gosta, dodaje veći broj rezervacija ako je u pitanju rezervacija za agenciju itd.



3. Decorator patern

Decorator patern omogućava nadogradnju tj dekoriranje objekta neke klase bez definisanja velikog broja izvedenih klasa.

Ovaj patern smo iskoristili u našem projektu. Klasa *Soba* predstavlja osnovnu ponudu hotela. Ukoliko je potrebno, klijent može nadograđivati ponudu različitim opcijama (pansion, parking, korištenje sefa). Interfejs *ISoba* sadrži definicije metoda *obracunajCijenu()* i *dajOpis()*. Klase *SobaSaParkingom*, *SobaSaPansionom*, *SobaSaSefom* implementiraju interfejs *ISoba*. Pozivom metode *obracunajCijenu()* dobija se osnovna cijena sobe sabrana sa dodatkom za odabranu opciju.



4. Bridge patern

Bridge patern omogućava da se odvoji apstrakcija nekog objekta od njegove implementacije.

Bridge patern nismo iskoristili u projektu. Bilo bi ga moguće iskoristiti kad bi naš sistem bio orijentisan prema upravljanju poslovanjem hotela, tj. prema radu sa podacima o zaposlenicima i njihovim aktivnostima. U tom slučaju imali bismo više klasa koje bi predstavljale različite tipove zaposlenika. Svaka od tih klasa implementirala bi interfejs koji sadrži definicije metode za obračun plate (različit za svakog tipa zaposlenika). Klasa Bridge bi tada na neki osnovni dio plate koji dobijaju svi zaposlenici dodavala različito obračunate iznose za svakog zaposlenika posebno.

5. Composite patern

Composite patern koristi se kada objekti imaju različite implementacije neke metode, a potrebno im je svima pristupati na isti način.

Ovaj patern nismo iskorisitili u našem projektu. Mogli bismo ga iskoristiti u sljedećoj situaciji. Recimo da je obračunavanje plate koje je razmatrano u opisu Bridge paterna drugačije, odnosno da se za svakog zaposlenika plata računa na potpuno različit način, bez ikakvog zajedničkog dijela. Tada, ako bi klijent želio da dobije izvještaj o platama svih zaposlenika, trebao bi postojati interfejs IZaposlenik i klasa HotelComposite. Interfejs bi definisao metodu dajPlatu(), a implementirale bi ga klase HotelComposite i sve klase koje predstavljaju tipove zaposlenika. Metoda dajPlatu() u HotelComposite bi objedinjavala plate svakog zaposlenika posebno u jedan izvještaj i klijent bi mogao pozivom te metode dobiti na jednom mjestu plate svih zaposlenika bez obzira što se one računaju na različite načine.

6. Proxy patern

Proxy patern omogućava kontrolu pristupa objektima.

Ovaj patern nismo iskorisitili u našem projektu. Mogli smo ga iskoristiti na sljedeći način. Pretpostavimo da *Gost* može samo dobiti podatak o ukupnom broju slobodnih soba u traženom terminu, *Recepcioner* ima pravo da vidi podatke o dostupnosti svake sobe kao i podatke o svakom gostu, a *Menadžer* može vidjeti i sve zahtjeve pristigle od strane agencija. Tada bi klasa *Proxy*, pomoću metode *pristup()*, provjeravala da li sistemu pristupa gost, recepcioner ili menadžer, te na osnovu toga određivala vrijednost atributa *nivoPristupa*. Interfejs *IPregled* definisao bi metode vezane za pregled podataka o sobama, gostima i zahtjevima, a implementirale bi ga klase *Proxy* i *Hotel*. U zavisnosti od nivoa pristupa, naslijeđene metode u klasi *Proxy* korisniku vraćale bi odgovarajuće podatke.

7. Flyweight patern

Flyweight patern se koristi kako bi se izbjeglo kreiranje velikog broja objekata koji su slični.

Ovaj patern nismo iskoristili u našem projektu. Kada bismo u sistemu upravljali i narudžbama jela u hotelskom restoranu, patern bismo mogli iskoristiti na sljedeći način. Pretpostavimo da u restoranu ima nekoliko tipova jela. Svaki put kada gost naruči neko jelo, ne mora se ponovo kreirati to isto jelo ako je nekad prije bilo naručeno, nego se može vratiti već kreirani objekat. U dijagramu klasa imali bismo pored klasa koje predstavljaju različite tipove jela, još i klasu *Restoran* i interfejs *IJelo*. Klasa *Restoran* bi čuvala sva već kreirana jela (objekte tipa *IJelo*) u listi. Metoda za dodavanje jela bi novo jelo dodavala samo u slučaju da jelo tog tipa već ne postoji, a postojala bi i metoda koja vraća jelo traženog tipa.