1. Adapter pattern

   Adapter pattern se koristi kada treba omogućiti nekompatibilne interfejse da rade jedan sa drugim posredstvom Adapter klase koja impementira klijent interfejs i služi kao omotač za servis koji je potrebno adaptirati.

   Ovaj patern bi se u našem projektu mogao iskoristiti kada bi htjeli omogućiti običnom korisniku da šalje poruke Izvođaču. Tada bi ObicniKorisnikController trebao implementirati interfejs ISlanjePoruka koja posjeduje metodu posaljiPoruku(k: ObicniKorisnik). Adapter klasa, IzvodjacControllerAdapter implementira ISlanjePoruka tako da u metodi posaljiPoruku implementira logiku kojom se šalje poruka Izvođaču putem nove metode IzvodjacController::posaljiPoruku(k : Izvodjac)

2. Facade pattern
   Facade patern se koristi kada sistem ima više podsistema pri čemu su apstrakcije i implementacije podsistema usko povezane. U tom slučaju, facade patern omogućava interfejs višeg nivoa na podsisteme čija je implementacija sakrivena od korisnika.

   Ovaj patern bi se u našem projektu mogao iskoristiti u slučaju kada bi naša aplikacija postala složenija, na primjer kada bi imali odvojen kontroler za slanje poruka PorukeController . Tada bi u fasadi pri rezervaciji karte mogli uključiti kupiKartu iz ObicniKorisnikController i npr. posaljiPoruku iz PorukeController kojom korisnik svim prijateljima salje obavjest o rezervaciji.

3. Decorator pattern
   Osnovna namjena Decorator paterna je dinamičko dodavanje novih funkcionalnosti postojećim objektima tako što objekat stavimo u poseban wrapper koji sadrži ove funkcionalnosti

   Ovaj patern bi se u našem projektu mogao iskoristiti u slučaju kada bi korisnik imao mogućnost slanja poruka putem više servisa. Na primjer uz slanje poruke na aplikaciji, mogao bi poslati i mejl uz to. Da bi ovo implementirali, morali bi uvesti novu klasu Notifier koja bi implementirala interfejs INotifier sa metodom posaljiPoruku. Potrebna je i Abstraktna BaseNotifierDecorator klasa koja implementira INotifier interfejs i u konstruktoru prima INotifier, sada je moguće dodati razne konkretne dekoratore kao sto su MailDecorator, ViberDecorator koje nasljeđuju ovu klasu.

4. **Proxy patern** se koristi za omogućavanje pristupa i kontrolu pristupa stvarnim objektima. Manipulacija nad nekim objektom je dozvoljena samo ako je neki uslov ispunjen ili ako korisnik npr. nema pravo pristupa traženom objektu. U našem slučaju, proxy patern je iskorišten tako što smo na dijagramu dodali PopustProxy klasu koja implementira interfejs IRezervacijaSaPopustom. Svaki korisnik može rezervisati kartu, ali rezervacija se može uraditi uz popust pod određenim uslovima. Putem metode obracunajPopust u Proxy klasi možemo zaključiti da li ta karta ima popust ili ne, te zavisno od toga odrediti da li je tom korisniku dozvoljena rezervacija uz popust ili ne

5. **Bridge pattern**

   Osnovna namjena Bridge paterna je da omogući odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih impementacija za pojedine apstrakcije.

   Bridge pattern je implementiran u projektu da bi razvojili funkcionalnost pretrage na njenu apstrakciju i implementaciju, gdje je apstrakcija nova klasa Pretrazivac koja se oslanja na funkcionalnost implementacije, a implementacija je novi interfejs IPretraga čije klase sadrže logiku nisku nivoa za pretragu.

6. Flyweight pattern

   Flyweight pattern se može primjeniti da bi se napravio objekat koji bi minimizirao utrošak memorije tako što se pojedini atributi koji ne ovise o kontekstu, djeljivi i ne mijenjaju se u vrijeme izvršenja programa (intrinsic) odvoje u flyweight klasu. Tada se pravi flyweight factory sa metodom getFlyweight(key) kojoj se proslijede dijelovi intrinisic stanja. Factory pregledava prethodno izrađene Flyweight objekte i vraća odgovarajući ili stvara novi. Sada se pri kreaciji novih objekata prvo provjerava da li je intrinsic stanje sačuvano u factory-u, zatim mu se dodaje exstrinsic stanje.

   Da bi primjenili ovaj patern, prvo se moramo uvjeriti da je zaista potrebno negdje čuvati veliki broj objekata sa sličnim vrijednostima atributa. Kada pogledamo Koncert klasu, s obzirom da jedan izvođač vjerovatno izvodi jedan žanr muzike, Flyweight objekat bi se mogao sastojati od atributa izvodjac i žanr.

7. Composite pattern

   Composite patern omogućava da se objekti komponuju u strukturu drveta u kojima se individualni objekti i kompozicije objekata jednako tretiraju

   Ovaj patern bi smo mogli implementirati ukoliko omogućimo grupne rezervacije, na ovaj način bi imali Composite klasu koja sadrzi listu korisnika i leaf klasu kao pojedine korisnike različitih vrsta. Composite i leaf klase bi implementirale interfejs sa operacijama koje vršimo nad pojedinim objektima. Ove operacije se obavljaju na različite načine ovisno od objekta. Na primjer, kada pozovemo Composite.calculatePrice, pri računanju ukupne cijene grupne rezervacije moguće je da se ta cijena kod premium korisnika računa na različit način nego kod običnih korisnika iako se poziva ista operacija i nad leaf objektima (Leaf.calculatePrice)