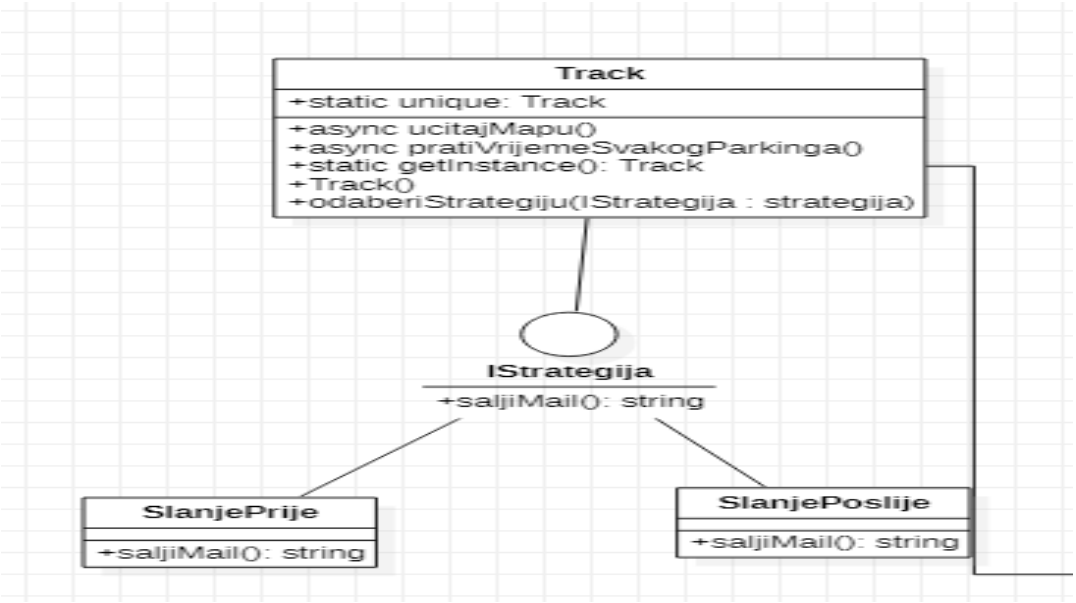
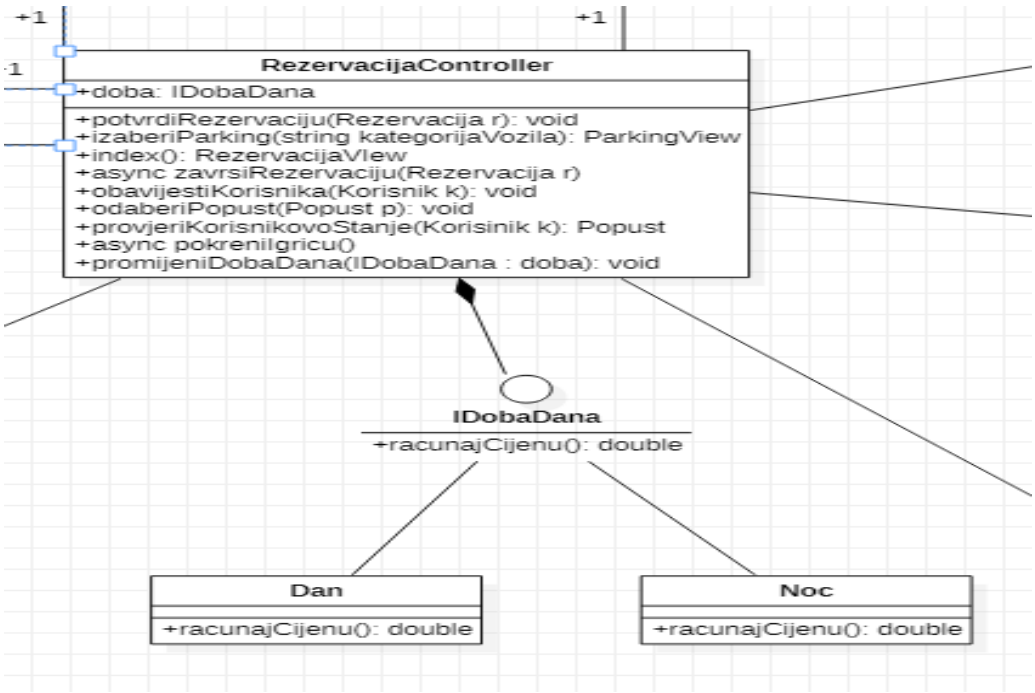
Patterni ponašanja

```
Implementirani patterni su:
 Strategy pattern
 State pattern
```

1. **Strategy pattern** služi za implementaciju određene funkcionalnosti kroz različite algoritame pri čemu se svaki od tih algoritama izdvaja u nasljeđene klase one klase u kojoj se funkcionalnost implementira. U našem slučaju on bi se mogao implementirati na način da se track klasa razdvoji na dvije podklase od kojih bi jedna slala mailove nekoliko vremena prije završetka rezervacije, a druga nakon završetka rezervacije. Metodom odaberiStrategiju se odabere kojom će se slati, metoda provjeriVrijemeSvakogParkinga se poziva metoda saljiMail u Istrategija.



2. **State pattern** je sličan strategy patternu samo što on zavisi od stanja a ne od “ručnog” izbora strategije. U našem slučaju on bi se mogao implementirati na način da se Rezervacija razdvoji na dvije podklase od kojih bi jedna predstavljala rezervacije u toku noći, a druga u toku dana pri čemu bi svaka od navedene dvije računale krajnje cijene (nakon cijene mjesta i nakon uračunavanja popusta) na svoj način, a klasa Track bi pozivala metodu za promjenu doba dana.



3. **Template method** odvaja dio algortima iz bazne klase u nasljeđene klase. U ovom sistemu to se može primjeniti kod klasa za upravljanje mjestom. Npr. moglo bi se napraviti da klasa Mjesto računa cijenu na osnovu sprata (što je visočije, to je parking jeftiniji, jer je dalje), a da klase AutobusMjesto, AutomobilMjesto i sl. računaju na osnovu kategorije vozila nakon izračunate cijene na osnovu sprata.

4. **Observer pattern** služi da primjeti promjene i onda da obavijesti druge klase o tim promjena. Kod nas to može biti implementirano kod klase Track koja bi obavijestila korisnike o tome kada im istekne rezervacija da dođu po svoje vozila ili da nakon što se čitav parking popuni, pošalje obavijest svim korisnicima da nema više mjesta na tom parkingu.

5. **Implementiranjem iterator patterna** omogućava se različit način kretanja kroz liste objekata. Ako bismo htjeli uvesti ovaj pattern u našu aplikaciju, to bi mogli uraditi tako što bi napravili pomoćni interfejs koji bi nasljedila klasa Track koja čuva sve rezervacije. Iz ovog interfejsa bi bio nasljeđen interfejs IIterator koji ima metodu zavrsavanjeRezervacija, a iz ovog interfejsa bi bili nasljeđeni RandomIterator i OrderedIterator gdje bi prvi izabrao random rezervaciju za provjeravanje da li je istekla, a drugi bi se kretao redom kroz listu i provjeravao da li je istekla rezervacija.

6. **Chain of responsibilty pattern** služi da se različite dužnosti dodjele pogodnim klasama, a u ovoj aplikaciji to je moguće primjeniti kod RezervacijaController klase koja bi pozivala prvo odabir kategorije, zatim odabir mjesta, a na kraju odabir popusta.

7. **Medijator pattern** se može iskoristiti za tretiranje različitih vrsta korisnika na različite načine. Npr. gost korisnik nema pravo na popuste na koje ima RegistrovaniKorisnik pa tako RezervacijaController koja ima metodu koja dodjeljuje popuste prvo poziva IPopust koji provjeri za trenutnog korisnika (na osnovu njegove uloge) ima li ta vrsta korisnika pravo na popust, a tek onda provjerava da li zadovoljava kriterije u zaduženim klasama.