PATERNI PONAŠANJA

1. **Strategy pattern**

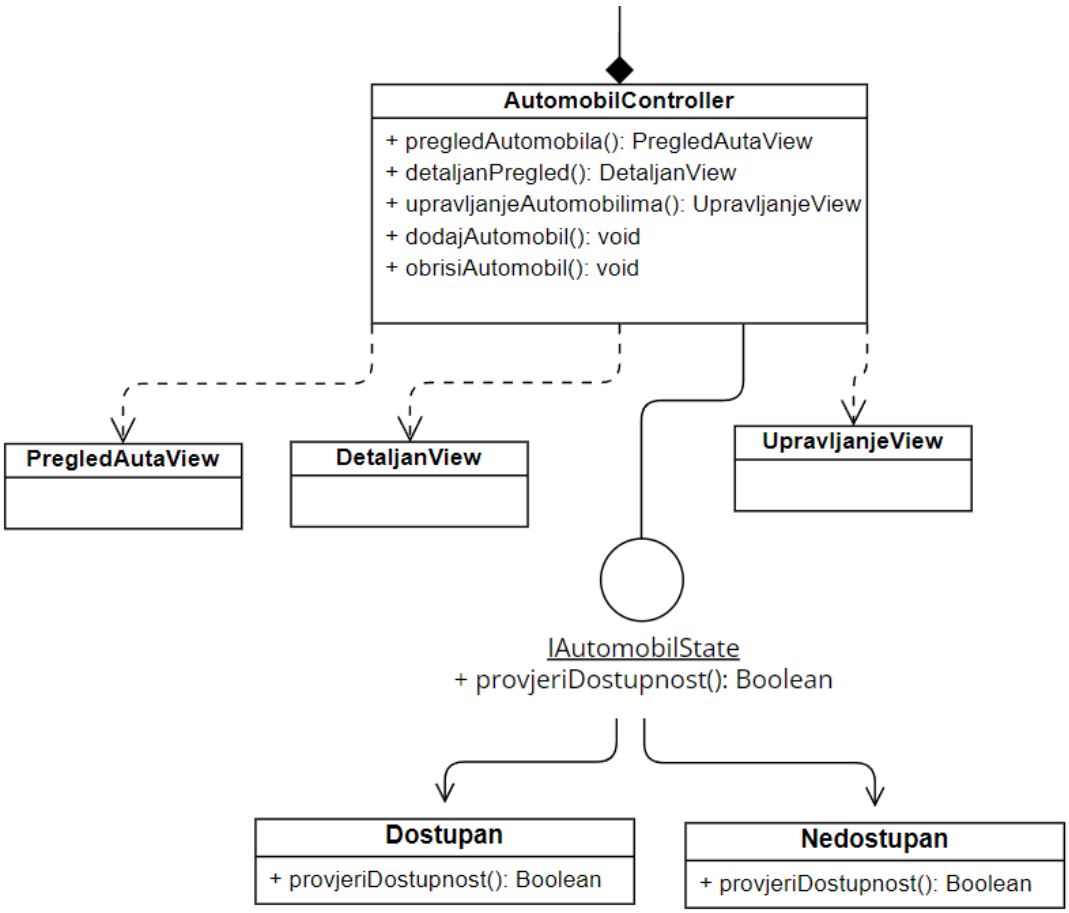
   Strategy pattern definiše porodicu algoritama, enkapsulira svaki od njih i omogućava njihovu zamjenu. Klijenti koji koriste algoritme mogu to raditi nezavisno od konkretne implementacije algoritma. Ovo omogućava fleksibilnost i proširivost, jer se novi algoritmi mogu lako dodati bez menjanja postojećeg koda.

   Strategy pattern bi mogli iskoristiti kod klase Konfiguracija gdje možemo imati različite strategije za izračunavanje konačne cijene konfiguracije, poput standardne cijene, cijene bez PDV-a, cijene sa popustom i sl. Kako bi to odradili napravili bi novu klasu Popust koja bi bila naša Context klasa, i također bi imali interfejs Strategy iz kojeg bi naslijedili te različite cijene.

2. **State pattern**

   State pattern omogućava objektu da promjeni svoje ponašanje kada mu se interno stanje promeni. Svako stanje je enkapsulirano kao posebna klasa koja definiše ponašanje objekta u tom stanju. Na ovaj način, objekti mogu dinamički mjenjati klase kako bi reflektovali promjenu stanja.

   Ovaj pattern ćemo implementirati za klasu AutomobilController tako što ćemo dodati interface IAutomobilState koji će govoriti o dostupnosti vozila. Iz interfacea ćemo izvesti klase Dostupan i Nedostupan. Izgled u dijagramu klasa je na slici ispod.



3. **Template method pattern**

   Template method pattern definiše skelet algoritma u okviru metode, ostavljajući određene korake podklasama da ih implementiraju. Ovo omogućava ponovnu upotrebu osnovne strukture algoritma dok podklase mogu prilagoditi specifične korake. Na ovaj način se postiže kontrola toka algoritma uz mogućnost prilagođavanja.

   Klasa Servis može definisati šablon metode za proces servisiranja. Konkretne klase kao što su MaliServis i VelikiServis mogu implementirati specifične korake unutar tog šablona. Na primer, koraci kao što su proveravanje ulja, zamena filtera, itd. mogu biti zajednički, dok konkretne akcije mogu varirati zavisno od tipa servisa.

4. **Observer pattern**

   Observer pattern definiše zavisnost jedan-na-više između objekata, tako da kada se stanje jednog objekta promjeni, svi zavisni objekti budu obavješteni i automatski ažurirani. Ovo omogućava reaktivno

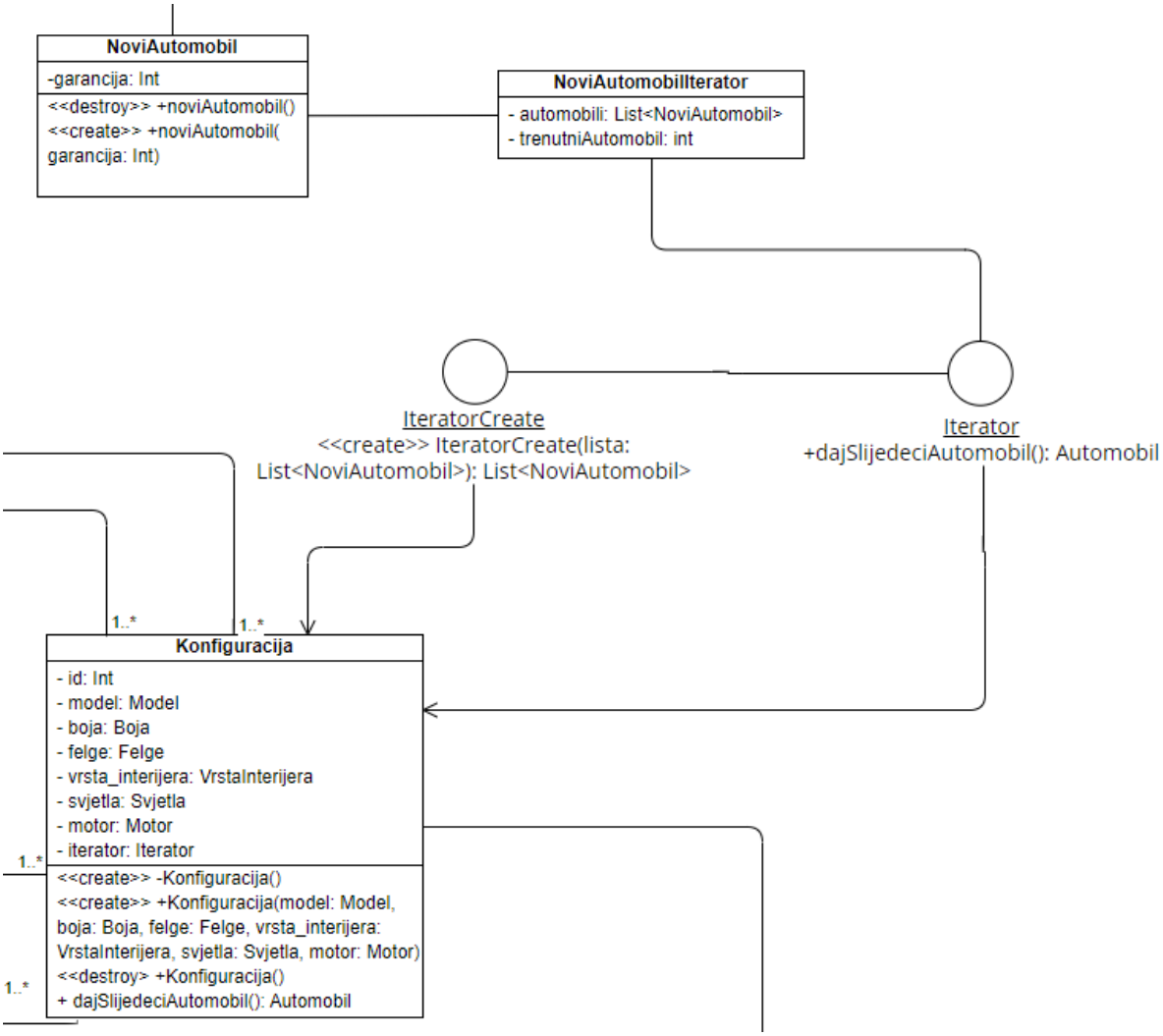
programiranje gdje promjene u jednom dijelu sistema propagiraju kroz sistem.

U sistemu za upravljanje narudžbama automobila, Observer Pattern se koristi za obaveštavanje korisnika o promenama statusa njihovih narudžbi. Kada korisnik kreira narudžbu, ta narudžba postaje subjekt sa listom posmatrača (korisnika). Kada se status narudžbe promeni, narudžba obaveštava sve posmatrače o novom statusu. Na primer, ako se status narudžbe promeni iz "u obradi" u "poslano", korisnici će automatski biti informisani o toj promeni. Ovaj patern omogućava korisnicima da uvek budu ažurirani o stanju svojih narudžbi bez potrebe za ručnim proveravanjem, čime se poboljšava korisničko iskustvo i komunikacija u sistemu.

5. **Iterator pattern**

   Iterator pattern pruža način za sekvencijalno pristupanje elementima agregatnog objekta bez izlaganja njegove osnovne reprezentacije. Klijent može da koristi iterator za traversiranje kolekcije objekata uniformno.

   Ovaj pattern ćemo implementirati na sljedeći način, tako što ćemo imati iterator za klasu NoviAutomobil, i tim iteratorom će moći iterirati klasa Konfiguracija. Kako bi ovo implementirali, potrebna nam je klasa NoviAutomobilIterator koja je izvedena, treba nam interface IteratorCreate i Iterator, i trebamo dodati atribut iterator u klasi Konfiguracija. U našem dijagramu to izgleda ovako.



6. Chain of responsibility pattern

Chain of responsibility pattern omogućava niz objekata koji mogu sukcesivno pregledati i obraditi zahtjev. Svaki objekat u lancu ima mogućnost da obradi zahtjev ili da ga proslijedi sljedećem objektu u nizu. Na taj način se izbjegava čvrsta povezanost pošiljaoca i primaoca zahtjeva.

U kontekstu obrade narudžbi, različite faze obrade (npr. validacija, provjera dostupnosti, finalizacija) mogu biti predstavljene kao lanci odgovornosti. Kada se narudžba kreira, ona prolazi kroz različite obrađivače. Prvi obrađivač može biti zadužen za validaciju podataka, drugi za provjeru dostupnosti automobila, a treći za finalizaciju narudžbe.

7. Mediator pattern

Mediator pattern obezbjeđuje centralizovani objekat koji upravlja komunikacijom između skupa objekata, smanjujući njihovu međusobnu zavisnost. Na ovaj način se pojednostavljuje komunikacija i smanjuje kompleksnost sistema.

U kontekstu narudžbi i servisa, posrednik može koordinirati komunikaciju između različitih komponenti sistema. Posrednik može koordinirati interakcije između narudžbi i servisa. Na primer, kada se narudžba kreira, posrednik može obavestiti servis da je novi automobil naručen i da treba zakazati inicijalni servis.