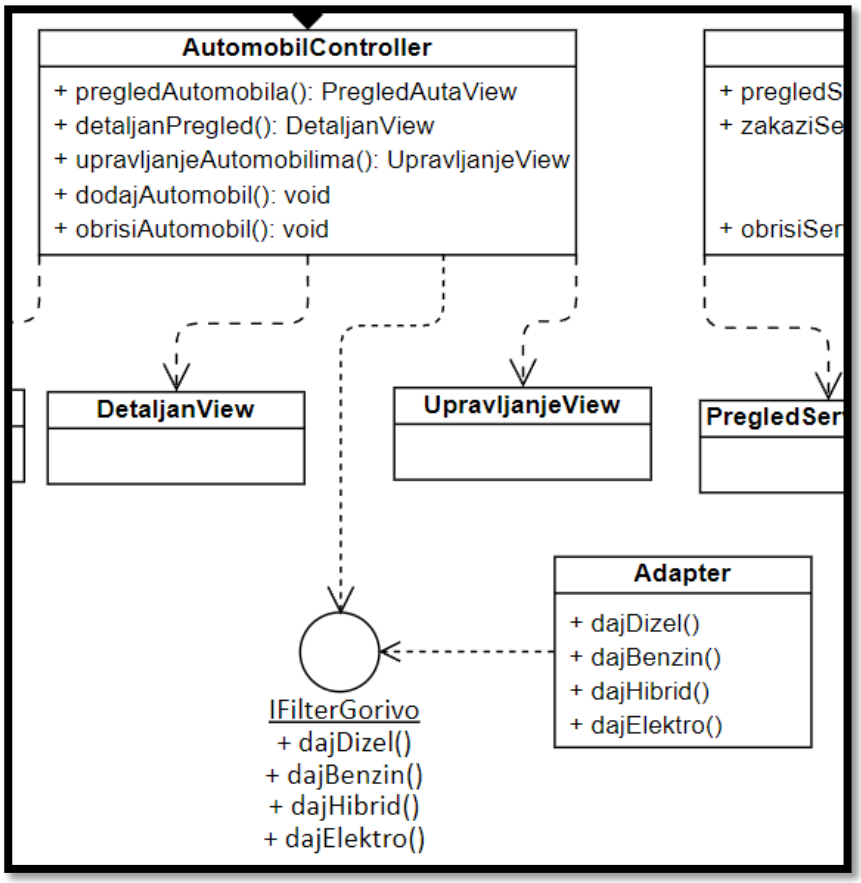
Strukturni patterni

1. **Adapter pattern**

   Adapter pattern je pattern čija je osnovna namjena da interfejs jedne klase pretvori u neki željeni interfejs kako bi se ona mogla koristiti u situaciji u kojoj bi inače problem predstavljali nekompatibilni interfejsi. Primjenom Adapter paterna dobija se željena funkcionalnost bez izmjena na originalnoj klasi i bez ugrožavanja integriteta cijele aplikacije.

   Za našu primjenu, kreirati ćemo novu mogućnost putem koje možemo filtrirati vozila po gorivu koje koristi. Kako i to uradili, kreiramo interfejs **IFilterGorivo.** Taj interfejs će posjedovati metode **dajDizel**(), **dajBenzin**(), **dajHibrid**() i **dajElektro**(). Svaka od metoda vraća odgovarajuća vozila. Te metode će biti implementirane u klasi **Adapter.** Izgled dijagrama nakon implementacije je sljedeći:



2. **Facade pattern**

   Facade pattern se koristi kada sistem ima više identificiranih podsistema pri čemu su apstrakcije i implementacije podsistema usko povezane. Osnovna

namjena Facade patterna je da osigura više pogleda visokog nivoa na podsisteme. Ovaj patern sakriva implementaciju podsistema od korisnika i pruža pojednostavljeni interfejs putem kojeg korisnik pristupa sistemu.

Ovaj pattern nismo iskoristili jer naš sistem ne sadrži veliki broj podsistema čije su implementacije usko vezane.

3. **Bridge pattern**

   Svrha Bridge pattern-a je omogućavanje da se iste operacije primjenjuju nad različitim podklasama. Ovim izbjegavamo nepotrebno dupliciranje koda, odnosno izbjegavamo kreiranje novih metoda za već postojeće funkcionalnosti. Dakle, bridge patern razdvaja pojedinačnu klasu na više zasebnih klasa koje se mogu razvijati odvojeno jedna od druge.

   Nismo odlučili implementovati ovaj pattern, ali kada bih ga implementovali koristili bi ga za različit način prikazivanja servisa. Radnik i korisnik na isti način dobivaju pregled servisa, ali razlika je u tome što korisnik samo vidi svoje servise, dok radnik ima uvid svih servisa u sistemu. Dodavanjem klase IServisi sa metodom getServisi() i Bridge klase koja vraća listu servisa u istom formatu i za radnika i za korisnika, mi veoma jednostavno to možemo uvesti.

4. **Proxy pattern**

   Proxy pattern je pattern koji pruža objekat koji se ponaša kao substitute objekat za pravi objekat koji pruža neku uslugu. Proxy objekat prihvata korisničke zahtjeve i obrađuje ih u zavisnosti od prava pristupa korisnika tj. onemogućen je slučajni pristup podacima od strane korisnika koji nema odgovarajuće permisije.

   Iako nismo odlučili implementovati ovaj pattern, način na koji smo ga mogli iskoristiti je tako što bismo uveli kontrolu pristupa bazi podataka. Kao jedan primjer zašto bi to radili, možemo reći da hoćemo zabraniti unošenje novih automobila u bazu svim drugim akterima osim administratoru.

5. **Flyweight pattern**

   Flyweight pattern koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji u suštini predstavljaju isti objekat.

   Ovaj pattern nećemo koristiti u našem sistemu, pošto nemamo potrebu za kreiranjem nekih “defaultnih”, tj. dijeljenih resursa. Ali primjer korištenja takvog patterna može biti sljedeći. Recimo da konfiguracija novih vozila se spašava u bazu podataka, te da za novo kreirane korisnike se odmah spasi neka defaultna konfiguracija, i svaka naknadna promjena konfiguracija se

spašava. Kako bi uštedili na kreiranju te nove konfiguracije, napravili bi interfejs **IKonfiguracija**, koja bi imala metodu **dajKonfiguraciju**(). Taj novi interfejs će koristiti dvije nove klase, **Konfiguracija** koja čuva promijenjenu konfiguraciju korisnika i **DefaultKonfiguracija** koja sadrži tu početnu konfiguraciju koja će se vezati za novo kreirane korisnike.

6. **Composite patern**

   Osnovna namjena Composite paterna je da omogući formiranje strukture drveta pomoću klasa, gdje se objekti (listovi) i kompozicije objekata (korijeni) jednako tretiraju. Ovo zapravo znači da je moguće pozivati metodu koja je zajednička na nivou svih tih klasa.

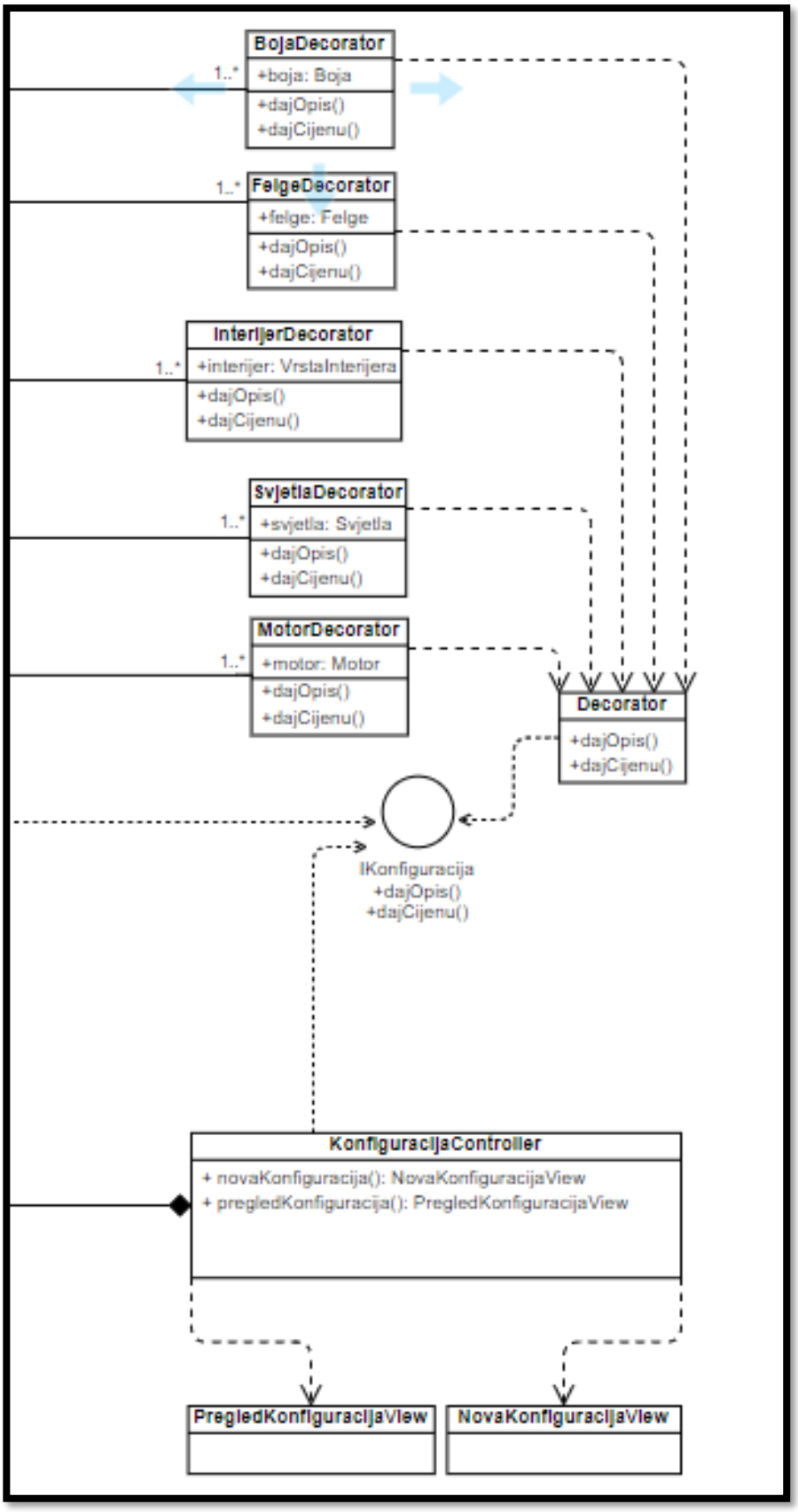
   Ovaj obrazac omogućava da se ponašanje izvršava rekurzivno preko svih komponenata stabla objekta, bez potrebe da se znaju konkretne klase objekata koji čine stablo.

   U našem sistemu composite patern možemo uvesti kada hoćemo rukovati sa korisnicima. Pomoću njega možemo dopustiti administratoru pregled informacija svih korisnika. Na ovaj način bi se stvorila zahtijevana struktura stabla, gdje bi korijen stabla bila klasa **Korisnik**, a listovi bi nam mogli biti **Zaposlenik** i **Kupac**.

7. **Decorator patern**

   Svrha Decorator patern-a je je da omogući dinamičko dodavanje novih elemenata i ponašanja (funkcionalnosti) postojećim objektima. Objekat pri tome ne zna da je uradena dekoracija što je veoma korisno za ponovnu upotrebu komponenti softverskog sistema.

   U našem slučaju dodali smo decorator patern za konfiguraciju automobila. Uz pomoć dvije opcije koje se nalaze u sklopu interfejsa **dajOpis***()* i **dajCijenu***()*, naš program može da dinamički mijenja opis i cijenu konfiguracije. Opis i cijena se mijenja pomoću **Decoratora** i njegovih **ConcreteDecorator** klasa, kao što su **BojaDecorator**, **FelgeDecorator**, **InterijerDecorator**, **SvjetlaDecorator** *i* **MotorDecorator**, gdje je svaka od njih vezana za enumeratorske klase, gdje odabirom neke vrijednosti iz istoimene enumeratorske klase dobije određenu cijenu za proizvod.


BojaDecorator
1..*
+boja: Boja
+dajOpis()
+dajCijenu()
FelgeDecorator
1..*
+felge: Felge
+dajOpis()
+dajCijenu()
InterijerDecorator
1..*
+interijer: VrstaInterijera
+dajOpis()
+dajCijenu()
SvjetlaDecorator
1..*
+svjetla: Svjetla
+dajOpis()
+dajCijenu()
MotorDecorator
1..*
+motor: Motor
+dajOpis()
+dajCijenu()
Decorator
+dajOpis()
+dajCijenu()
IKonfiguracija
+dajOpis()
+dajCijenu()
KonfiguracijaController
+ novaKonfiguracija(): NovaKonfiguracijaView
+ pregledKonfiguracija(): PregledKonfiguracijaView
PregledKonfiguracijaView
NovaKonfiguracijaView