# Paterni Ponašanja

## Strategy Pattern

Strategy pattern je obrazac koji se koristi kada želimo omogućiti izbor između više različitih algoritama ili logika, bez potrebe da mijenjamo osnovni kod koji ih koristi. Umjesto da u programu imamo veliki broj if-else uslova koji određuju koju logiku da primijenimo, Strategy pattern omogućava da tu logiku „upakujemo“ u zasebne klase i da ih mijenjamo po potrebi. Ovo čini kod preglednijim, fleksibilnijim i lakšim za održavanje.

U aplikaciji OffroadAdventure, Strategy pattern je primijenjen prilikom obračuna popusta za korisnike koji žele da iznajme vozila. Konkretno, metoda IzracunajPopust, koja se nalazi u ZahtjevZaRentanjeController kontroleru, koristi broj dana trajanja najma i broj izabranih vozila da izračuna koliki popust korisnik ostvaruje. Sama logika je sljedeća: za svaki dodatni dan (računajući i prvi dan kao minimum) dodaje se 2% popusta, dok se za svako dodatno vozilo dodaje 5% popusta. Zbir ovih procenata se ograničava na najviše 50% ukupnog popusta. Nakon što se izračuna popust, osnovna cijena se množi sa (1 - popust / 100) da bi se dobila krajnja cijena koju korisnik treba platiti. Ova metoda se koristi kako prilikom kreiranja zahtjeva (Create), tako i prilikom uređivanja (Edit), čime se osigurava da se ista logika primjenjuje dosljedno.

Trenutno ova logika predstavlja tzv. standardni model obračuna popusta, ali je struktura koda takva da bi se lako mogla proširiti u pravi Strategy pattern. To bi značilo da se kreira interfejs (npr. IPopustStrategija) i više klasa koje implementiraju različite načine obračuna (npr. vikend popust, popust za stalne korisnike, akcijski popust itd.). Kontroler bi onda dinamički birao koju strategiju da koristi, bez potrebe da mijenja svoju osnovnu logiku. Na taj način bi aplikacija bila fleksibilnija, spremna za rast i jednostavnija za održavanje.

## State Pattern

State pattern je dizajn obrazac koji omogućava da objekat mijenja svoje ponašanje na osnovu svog trenutnog stanja. Umjesto da se sve kontroliše kroz if-else grane, ponašanje se mijenja dinamički u zavisnosti od aktivnog stanja objekta. U aplikaciji OffroadAdventure, ovaj obrazac je primijenjen kroz model ZahtjevZaRentanje, koji koristi enumeraciju StatusZahtjeva za praćenje stanja svakog zahtjeva. Ta stanja su definisana kao NA_CEKANJU, ODOBREN i ODBIJEN. Prilikom kreiranja zahtjeva (u metodi Create), status se automatski postavlja na StatusZahtjeva.NA_CEKANJU, čime se označava da zahtjev još nije obrađen. U zavisnosti od

odluke administratora ili zaposlenika, putem metoda kao što su Odobri i Odbij, status se mijenja na ODOBREN ili ODBIJEN direktnim ažuriranjem polja statusZahtjeva. Ove promjene stanja imaju direktan uticaj na dalji tok aplikacije — kada je zahtjev odobren, korisniku se automatski šalje notifikacija, a u prikazima su dostupne druge akcije u odnosu na zahtjeve koji su još na čekanju ili odbijeni. Iako nije korištena puna implementacija State patterna sa odvojenim klasama za svako stanje, korištenje enum vrijednosti u kombinaciji sa uvjetnom logikom i kontrolom toka u potpunosti odražava osnovni princip ovog obrasca. Na osnovu trenutnog stanja objekta (statusZahtjeva), aplikacija zna kako da se ponaša, šta da prikaže korisniku, koje podatke da ponudi, i koje naredne akcije su dozvoljene. Ovakav pristup je jednostavan, jasan i lako proširiv, a u potpunosti ispunjava svrhu State patterna u kontekstu web aplikacije ovog tipa.

## Template Method Pattern

Template Method pattern je obrazac koji se koristi kada želimo definisati osnovnu strukturu nekog algoritma, ali ostaviti mogućnost da se pojedini koraci mijenjaju ili proširuju bez narušavanja glavne logike. U aplikaciji OffroadAdventure, ovaj obrazac se koristi u dijelu koji se odnosi na prikaz komentara (recenzija), konkretno u metodi Recenzije unutar KomentarController-a. Ova metoda sadrži „šablon“ algoritma koji uključuje: dohvat svih komentara iz baze (_context.Komentar.Include(k => k.user)), zatim filtriranje na samo one komentare koji su osnovne recenzije (one bez komentarId, odnosno koji nisu odgovori), izračun prosječne ocjene, broj komentara, i na kraju – sortiranje. Upravo taj korak sortiranja je mijenjajući dio šablona, i on se dinamički prilagođava na osnovu parametra poredak koji se prenosi iz URL-a (npr. ?poredak=asc ili ?poredak=desc). U zavisnosti od vrijednosti poredak, komentari se sortiraju rastuće (OrderBy(k => k.ocjena)) ili opadajuće (OrderByDescending(k => k.ocjena)), bez potrebe da se ostatak metode mijenja.

Ono što je ključno jeste da struktura metode ostaje ista — podaci se učitavaju, filtriraju i pripremaju za prikaz — ali je logika sortiranja fleksibilna i mijenja se u zavisnosti od korisničkog izbora. Taj izbor se realizuje u prikazu putem select elementa koji automatski mijenja URL, a backend obrađuje zahtjev i vraća komentare u traženom redoslijedu. Ovakav pristup omogućava elegantno proširivanje – npr. lako se može dodati i sortiranje po datumu ili korisniku, bez izmjene osnovne strukture metode. Upravo ta kombinacija stabilne šablonske logike i fleksibilnog detalja je tipična primjena Template Method patterna u praksi.

# Chain of Responsibility Pattern

Chain of Responsibility (lanac odgovornosti) je obrazac koji omogućava da se neki zahtjev proslijedi kroz niz objekata sve dok jedan od njih ne preuzme odgovornost za njegovo obrađivanje. Svaki objekat u tom lancu ima priliku da obradi zahtjev ili ga proslijedi sljedećem. Ovaj princip omogućava fleksibilno upravljanje logikom bez da objekti međusobno direktno zavise jedni od drugih.

U aplikaciji OffroadAdventure, Chain of Responsibility pattern je primijenjen kroz hijerarhijsko upravljanje zahtjevima za rentanje vozila, gdje u lancu učestvuju dvije uloge: zaposlenici i administratori (vlasnik). Kada korisnik kreira zahtjev za rentanje (u metodi Create), on dobija status NA_CEKANJU i time ulazi u sistem obrade. Prvi u lancu koji može obraditi zahtjev su zaposleni ([Authorize(Roles = "Zaposlenik, Administrator")]), koji imaju pristup akcijama kao što su Odobri, Odbij, Edit i Details. Ukoliko zaposleni ne preuzmu obradu ili postoji potreba za višim nadzorom, administrator (vlasnik) također ima pristup istim metodama i može donijeti konačnu odluku.

Bitna stvar je da nijedna akcija ne zavisi direktno od druge — zahtjev se jednostavno prosljeđuje kroz korisničke uloge i odgovarajući handler (ZahtjevZaRentanjeController) reaguje u skladu sa ulogom prijavljenog korisnika. Svaka akcija (Odobri, Odbij, Delete, Edit) implicitno funkcioniše kao potencijalni “član u lancu” koji može preuzeti i obraditi zahtjev. Ova fleksibilna logika omogućava da se zahtjev u različitim fazama obrađuje od strane različitih korisnika, bez da se logika mora duplicirati ili povezivati direktno s određenim korisničkim ID-em ili specifičnim handlerom.

Na primjer, kada zaposlenik klikne na dugme "Odobri", poziva se metoda Odobri(int id) koja ažurira status zahtjeva i šalje notifikaciju korisniku. Ako to nije urađeno, administrator također ima pristup istom lancu putem istih metoda i može izvršiti tu akciju. Tako se zahtjev "putuje" kroz sistem, a lanac se prekida čim neka instanca preuzme odgovornost.

# Command Pattern

Command pattern je ponašajni dizajn obrazac koji omogućava da se korisnički zahtjevi (akcije) enkapsuliraju kao posebni objekti. Na taj način, umjesto da se neka akcija izvrši odmah, možemo je pohraniti, kasnije izvršiti, poništiti ili kombinovati s drugim. Svaki „komandni“ objekat sadrži sve potrebne informacije za izvršenje: ko je inicijator, koji je tip zahtjeva i koje parametre sadrži.

U aplikaciji OffroadAdventure, Command pattern je primijenjen u procesu kreiranja zahtjeva za rentanje vozila. Kada korisnik ispuni formu i klikne na dugme za potvrdu (akcija Create u

ZahtjevZaRentanjeController), njegovi unosi — uključujući lične podatke, datume, odabrana vozila i dodatne zahtjeve — se enkapsuliraju u objekat tipa ZahtjevZaRentanje. Ovaj objekat predstavlja komandu koju korisnik šalje sistemu. Nakon što se zahtjev kreira i sačuva u bazi, ne izvršava se odmah u potpunosti — već prelazi u status NA_CEKANJU i čeka dalju obradu od strane drugog aktera (npr. zaposlenika ili administratora). Kasnije, zaposleni ili vlasnik sistema izvršavaju komandu tako što odobravaju (Odobri) ili odbijaju (Odbij) zahtjev, čime se komanda aktivira i pokreće dodatne akcije (npr. slanje notifikacije korisniku da je njegov zahtjev odobren ili odbijen).

Ova logika omogućava odvajanje korisničkog unosa od njegove obrade, što je suština Command patterna — korisnik samo „izda komandu“, dok neko drugi, u drugo vrijeme, odlučuje kada i kako će je sistem izvršiti. Takođe, sistem kroz model ZahtjevZaRentanje čuva sve potrebne informacije o zahtjevu, što omogućava naknadno upravljanje (izmjena, brisanje, statusna obrada) bez ponovnog unosanja podataka. Na ovaj način postiže se fleksibilnost, transparentnost i mogućnost reakcije sistema u više koraka, uz potpunu kontrolu nad svakom individualnom akcijom.

## Iterator Pattern

Iterator pattern koristi se da omogućimo prolazak kroz elemente neke kolekcije, bez otkrivanja unutrašnje strukture te kolekcije. Drugim riječima, klijent (view, kontroler...) može da pristupi svakom elementu iz kolekcije jedan po jedan, bez da zna da li je ta kolekcija niz, lista iz baze, filtrirana kolekcija ili nešto četvrto. To čini sistem fleksibilnim i pojednostavljuje pristup podacima.

Iterator pattern smo koristili u više dijelova naše aplikacije OffroadAdventure, jer nam je omogućio da jednostavno prolazimo kroz kolekcije objekata bez potrebe da znamo njihovu tačnu strukturu. Poenta ovog obrasca je da omogućava kretanje kroz listu elemenata, jedan po jedan, bez otkrivanja kako su ti elementi organizovani ili sačuvani.

U našoj aplikaciji, Iterator pattern se najviše vidi na ekranima za prikaz komentara i vozila.

Na ekranu Recenzije, gdje korisnici ostavljaju ocjene i komentare, koristimo foreach petlju da prikažemo sve komentare iz modela koji se prosljeđuje view-u. Za svaki komentar takođe iteriramo i kroz njegove odgovore, koji su logički povezani preko komentarId atributa. Bez obzira na to da li komentari dolaze direktno iz baze, da li su prethodno filtrirani ili sortirani – view koristi jednostavnu foreach petlju i prikazuje ih jedan po jedan. Upravo ta mogućnost da se "krene kroz listu" bez potrebe da znamo kako je ta lista izgrađena, predstavlja čistu primjenu Iterator patterna.

Slično tome, na stranici za izbor dostupnih vozila, prolazimo kroz listu objekata Vozilo i za svako vozilo prikazujemo karticu sa slikom, nazivom i cijenom, a korisnik može označiti vozila koja želi da rezerviše. I ovdje koristimo foreach (var vozilo in Model) – što je još jedan primjer gdje se Iterator koristi da bismo prikazali sve stavke jedne kolekcije bez potrebe za znanjem o pozadini podataka.

Na ovaj način, aplikacija ostaje fleksibilna, pregledna i jednostavna za održavanje, jer view-ovi ostaju odvojeni od konkretne implementacije podataka, a prikazivanje se svodi na jednostavno iteriranje kroz elemente.

# Mediator pattern

Mediator pattern koristimo kako bismo centralizovali komunikaciju između različitih komponenti sistema, bez da one direktno zavise jedna od druge. Umjesto da svaka komponenta direktno komunicira sa drugima, sve interakcije se obavljaju putem jednog posrednika – tzv. medijatora. Time se smanjuje međusobna zavisnost i olakšava održavanje, jer se logika koordinacije prebacuje u jednu centralnu tačku.

U našoj aplikaciji OffroadAdventure, ovaj obrazac smo implementirali kroz NotifikacijasController, koji služi kao posrednik između svih kontrolera i akcija koje generišu događaje – kao što su kreiranje, odobravanje ili odbijanje zahtjeva, plaćanje, odgovaranje na komentare, i drugih promjena stanja – i korisnika koji trebaju biti obaviješteni putem notifikacija.

Na primjer, u ZahtjevZaRentanjeController, kada se zahtjev odobri ili odbije, ne šaljemo direktno poruku korisniku, niti upravljamo prikazom notifikacija. Umjesto toga, samo pozovemo metodu KreirajNotifikaciju() u NotifikacijasController, koja kreira i upisuje notifikaciju u bazu. Zatim se te notifikacije prikazuju korisnicima pomoću metode PrikaziPopup(), a korisnik ih može označiti kao pročitane ili ih obrisati.

Zahvaljujući ovoj strukturi, svi djelovi sistema koji trebaju slati obavještenja komuniciraju isključivo sa NotifikacijasController, koji koordinira cijeli proces. Time smo postigli čistu arhitekturu: svaki kontroler zna samo da je nešto "desilo", i da to treba prenijeti korisniku – ali ne zna *kako*, *kada* ni *kojim putem*. Sve to rješava naš "medijator", što je upravo i suština Mediator patterna.

# Observer Pattern

Observer pattern je obrazac koji omogućava automatsko obavještavanje svih zainteresovanih objekata kada dođe do promjene stanja u nekom subjektu. Umjesto da svaki objekat pojedinačno prati promjene, ovaj obrazac omogućava da se objekti prijave kao "posmatrači" (eng. observers), a subjekat im sam šalje obavještenje kad do promjene dođe. Na taj način se postiže bolja modularnost, manja povezanost među komponentama i reakcija sistema u realnom vremenu.

U našoj aplikaciji OffroadAdventure, ovaj princip smo primijenili kroz sistem notifikacija. Kada se promijeni status određenog zahtjeva za rentanje (npr. kada zaposlenik odobri, odbije ili kada korisnik izvrši plaćanje), ta promjena pokreće kreiranje notifikacije za korisnika kojem se to tiče. Ta logika je implementirana u različitim kontrolerima (npr. ZahtjevZaRentanjeController), koji pozivaju NotifikacijasController.KreirajNotifikaciju().

Korisnik koji je prijavljen ima automatski prikaz notifikacija u zaglavlju aplikacije, jer se periodično šalje AJAX zahtjev ka metodi ProvjeriNove() u NotifikacijasController, koja provjerava da li ima novih nepročitanih obavještenja. Ako ih ima, prikazuje se popup sa sadržajem, preko metode PrikaziPopup(). Na taj način smo postigli da korisnik ne mora ručno osvježavati stranicu da bi saznao šta se desilo, već sistem sam obavještava sve zainteresovane strane — baš kao što to predviđa Observer pattern.

Drugim riječima, kontroleri koji mijenjaju stanje sistema su naši "subjekti", a korisnici koji primaju obavijesti su "posmatrači", koji preko UI-a reaguju na te promjene. Sve to smo realizovali bez eksplicitnih IObserver i ISubject klasa, ali smo jasno ispratili logiku ovog obrasca.

# Visitor Pattern

Visitor patern omogućava dodavanje novih operacija nad objektima bez izmjene njihovih klasa, što je korisno u sistemima koji često zahtijevaju proširivanje funkcionalnosti nad postojećim strukturama podataka. U aplikaciji *OffroadAdventure*, ovaj patern nije implementiran, jer sistem ne zahtijeva izvođenje dodatnih operacija nad objektima kao što su User, Vozilo ili Komentar, van onih koje su već direktno definisane u kontrolerima.

# Interpreter pattern

**Interpreter patern** koristi se za interpretaciju i evaluaciju izraza definisanih u nekom domenskom jeziku. U aplikaciji *OffroadAdventure*, ovaj patern **nije implementiran**, jer ne postoji potreba za analizom i izvršavanjem posebnih izraza, pravila ili sintakse. Sve funkcionalnosti aplikacije se realizuju direktno kroz programski kod i standardne kontrole, bez upotrebe sopstvenog jezika ili parsera.

# Memento pattern

Memento patern koristi se za čuvanje prethodnog stanja objekta, kako bi se po potrebi moglo vratiti u to stanje, bez narušavanja enkapsulacije. U aplikaciji *OffroadAdventure*, ovaj patern nije implementiran, jer sistem ne predviđa funkcionalnosti poput undo/redo mehanizama ili vraćanja prethodnih verzija zahtjeva, komentara ili drugih podataka. Sve akcije u sistemu su konačne i direktno utiču na trenutno stanje objekata.