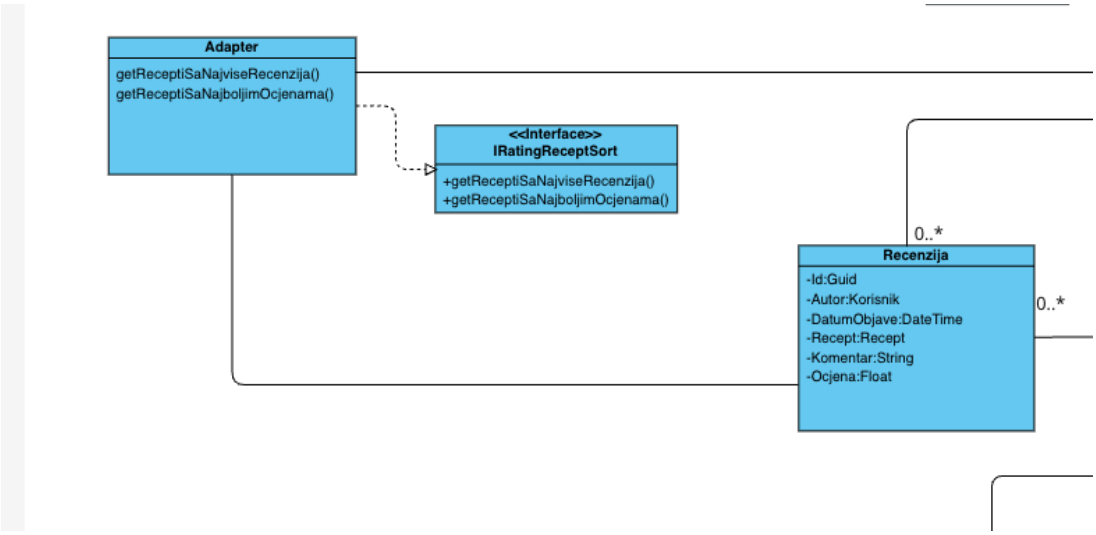
STRUKTURALNI PATTERNI ZA NUTRITIONIST APLIKACIJU

1) ADAPTER PATTERN

Svrha **Adapter paterna** je da omogući širu upotrebu već postojećih klasa. U situacijama kada je potreban drugačiji interfejs od onog koji pruža postojeća klasa, a ne želimo mijenjati njen kod, koristi se Adapter patern.Ovaj pattern kreira novu **adapter klasu** koja djeluje kao posrednik između originalne klase i traženog interfejsa.

Ukoliko nam je potrebna nova funkcionalnost sortiranja recepata prema recenzijama, ali ne želimo da dodatno opterećujemo klasu Recept, napravit ćemo klasu Adapter koja koristi podatke iz Recenzija. Klasa adapter će imati dvije metode:
GetReceptSaNajviseRecenzija() – metoda koja će proći kroz listu recepata i, koristeći podatke iz pripadajućih recenzija, odrediti koji recept ima najviše recenzija.
GetReceptSaNajboljomRecenzijom() – metoda koja će analizirati recenzije određenog recepta, izračunati prosječnu ocjenu te omogućiti sortiranje i dohvat recepta.

Prednosti korištenja ovog paterna su sljedeće:
Fleksibilnost: Ako se u budućnosti promijene kriteriji sortiranja ili se doda nova logika u određivanju "najboljeg recepta", izmjene će biti ograničene samo na adapter klasu.
Razdvajanje odgovornosti: Klasa Recept ostaje fokusirana na čuvanje podataka o receptu, dok adapter rješava poslovnu logiku sortiranja prema recenzijama.



Slika 1. Adapter pattern

2) FACADE PATTERN

Facade pattern se koristi kako bi se korisnicima olakšalo korištenje složenih sistema. Primjenjuje se kada ne želimo da krajnji korisnik mora poznavati detaljnu unutrašnju logiku sistema. Umjesto toga, korisniku se nudi pojednostavljen interfejs koji omogućava interakciju sa sistemom bez razumijevanja svih tehničkih detalja u pozadini.

U okviru naše aplikacije, Facade pattern bi se idealno uklopio u upravljanje zadacima administratora. Administrator ima pristup brojnim funkcijama, uključujući upravljanje korisnicima, receptima, recenzijama i newsletterom. Ove aktivnosti mogu biti kompleksne i međusobno zavisne.

Implementacijom fasade, moguće je objediniti sve ove funkcionalnosti u jedan interfejs koji će upravljanje učiniti jednostavnijim i efikasnijim.

Prednosti primjene:

Jednostavnost korištenja:
Umjesto da se administrator bavi nizom pojedinačnih metoda i koordinacijom među njima, fasada može pružiti jedinstvene metode koje obavljaju više koraka odjednom.
Sakrivanje kompleksnosti:
Klijent koristi samo ono što mu je potrebno, dok se interna logika sistema nalazi iza fasade i ostaje nevidljiva.

3) DECORATOR PATTERN

Decorator pattern omogućava dinamičko proširivanje funkcionalnosti objekata bez izmjene njihovog izvornog koda. Umjesto kreiranja velikog broja podklasa koje nasljeđuju osnovnu klasu, ovaj obrazac omogućava fleksibilno „umotavanje“ objekata dodatnim funkcionalnostima putem posebnih dekoratorskih klasa. Na taj način se zadržava princip otvorenosti za proširenje, ali zatvorenosti za izmjene.

Glavna prednost Decorator patterna je u tome što omogućava dodavanje novih ponašanja u toku izvršavanja programa, bez narušavanja postojećeg dizajna i bez potrebe da se svi korisnici objekta upoznaju sa dodatnim složenostima.

U kontekstu naše aplikacije, Decorator pattern se može efikasno primijeniti na klase kao što su Recept i Newsletter, u cilju proširivanja funkcionalnosti.

Za klasu *Recept*, možemo zamisliti interfejs *IReceptDecorator* sa metodom *prikazi()*, koja omogućava prikaz informacija o receptu. Na osnovu ovog interfejsa, kreiraju se dekoratori koji proširuju postojeći prikaz dodatnim detaljima.

Na primjer, jedan dekorator može dodati nutritivne informacije kao što su kalorijska vrijednost i udio makronutrijenata, dok drugi može omogućiti prikaz slike jela uz osnovne podatke. Ovi dekoratori pozivaju originalnu *prikazi()* metodu i nadograđuju je vlastitim sadržajem, što omogućava fleksibilno kombinovanje više dodataka bez potrebe za izmjenom osnovne klase.

Slično se može pristupiti i dekoraciji klase *Newsletter*. Definisanjem interfejsa *INewsletter* sa metodama za prikaz sadržaja i slanje korisniku, moguće je razviti dodatke koji proširuju osnovnu funkcionalnost newslettera. Jedan dekorator može omogućiti dodavanje privitaka, poput PDF dokumenata i slika, dok drugi može uvesti personalizovane poruke i preporuke prilagođene korisnikovim interesovanjima. Još jedan može dodati plan ishrane kreiran od strane nutricioniste, čime se korisnicima pruža dodatna stručna vrijednost.

4) BRIDGE PATTERN

Bridge pattern služi za odvajanje apstrakcije od njene implementacije, što omogućava da jedna klasa istovremeno podržava više različitih apstrakcija i različitih implementacija. Na taj način se postiže veća fleksibilnost i proširivost sistema, jer se apstrakcija i implementacija mogu mijenjati nezavisno jedna od druge.

U našoj aplikaciji ovaj pattern bi podržali na sljedeći način:

Registrovani korisnik pristupa naprednom prikazu recepata koji uključuje dodatne informacije kao što su status omiljenih recepata, personalizovani savjeti i mogućnost dodavanja komentara. S druge strane, gost korisnik vidi osnovni prikaz recepata sa ograničenim informacijama, bez mogućnosti interakcije poput dodavanja u omiljene ili komentarisanja. Oba korisnika mogu pretraživati recepte i dobiti listu rezultata, ali prikaz i funkcionalnosti se razlikuju.

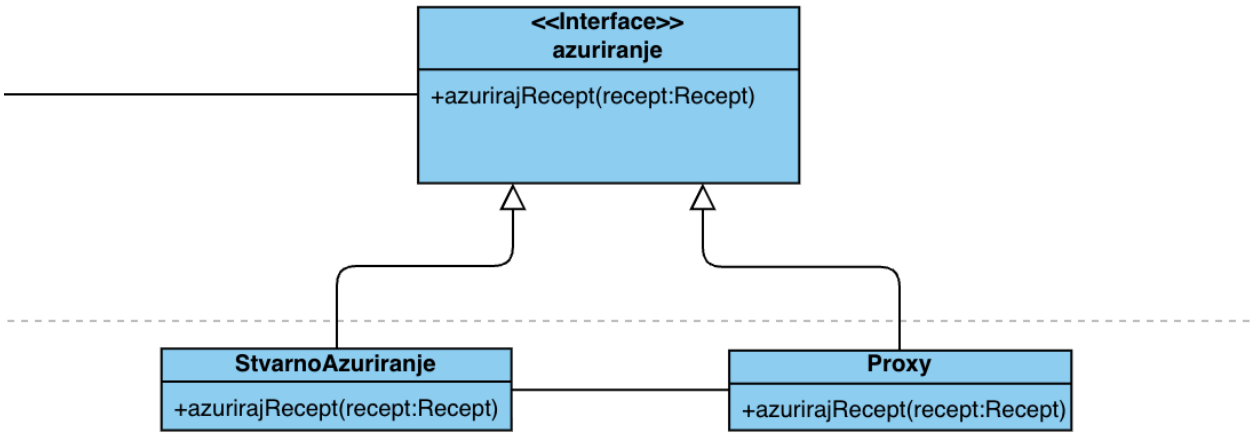
Koristeći Bridge pattern, apstrakcija prikaza recepata se odvaja od logike dohvata i obrade podataka. Time omogućavamo da se različiti prikazi (za registrovanog i gosta) lako implementiraju i proširuju, a da pri tome ne bude potrebe za mijenjanjem osnovne logike rada sa podacima. Ova separacija pojednostavljuje održavanje i unapređenje aplikacije.

5) PROXY PATTERN

Proxy pattern ima za cilj da upravlja pristupom stvarnim objektima, djelujući kao njihov zamjenski predstavnik ili surogat. Proxy objekat je obično jednostavan i lagan, dok pravi, složeniji objekat može imati skupu ili vremenski zahtjevnu inicijalizaciju. Proxy omogućava kontrolu pristupa, odnosno odlučuje kada i kako će se pravi objekat aktivirati, u skladu sa definisanim pravilima ili uslovima.

Na taj način, Proxy pattern optimizuje rad sistema tako što odlaže ili ograničava stvarno kreiranje ili pozivanje kompleksnih objekata dok to zaista nije neophodno, a ujedno može služiti i za dodatne funkcionalnosti poput keširanja, autentifikacije ili praćenja pristupa.

U našoj aplikaciji proxy pattern je implementiran kroz interface *ažuriranje*, koji definiše metode za ažuriranje podataka.



Slika 2. Proxy pattern

Klijent (u ovom slučaju korisnik) koristi isti interfejs za ažuriranje, bez potrebe da zna da li komunicira direktno sa stvarnim objektom ili proxyjem. Proxy klasa provjerava da li korisnik ima prava da izvrši ažuriranje i tek tada, ukoliko su uslovi zadovoljeni, prosleđuje poziv stvarnoj klasi koja izvršava ažuriranje recepta.

Na ovaj način Proxy pattern omogućava:

Kontrolu pristupa — samo ovlašteni korisnici mogu izvršiti određene izmjene,
Transparentnost klijenta — klijent ne mora znati da li koristi proxy ili stvarnu implementaciju,
Fleksibilnost i sigurnost — proxy centralizovano provjerava pristup i može dodati dodatne provjere ili logiku.

6) COMPOSITE PATTERN

Composite pattern se koristi za kreiranje hijerarhijskih struktura u obliku stabla, gdje se pojedinačni objekti (listovi) i kompozicije tih objekata (čvorovi ili korijeni) tretiraju na isti način. Ovaj pattern omogućava da različite klase koje imaju različite implementacije istih metoda mogu biti objedinjene kroz zajednički interfejs, pojednostavljujući time upravljanje složenim strukturama.

Ovaj pattern bi se mogao implementirati na sljedeći način:

Organizacija recepata u hijerarhiju kao što su kategorije i podkategorije, gdje su i pojedinačni recepti i kategorije predstavljeni kao objekti koji implementiraju isti interfejs.

Na primjer:

- Kategorija recepata može sadržavati podkategorije i/ili same recepte,
- Klijent može pristupiti i upravljati ovom hijerarhijom bez obzira da li je u pitanju pojedinačni recept ili cijela kategorija.