# Paterni ponašanja

## Strategy patern

Strategy patern vrši izdvajanje algoritma iz matične klase u posebne klase čime omogućava klijentu izbor algoritma iz familije algoritama, te održava neovisnost algoritama od klijenata koji ih koriste.

Ukoliko bismo htjeli omogućiti više vrsta sortiranja recepata, tako da se odabere najefikasniji način u zavisnosti od broja recepata koje je potrebno sortirati, mogli bismo iskoristiti ovaj patern. Tada bismo dodali novi interface ISortiranje sa metodom sortiraj(). Dodali bismo i klase QuickSort, MergeSort i InsertionSort, te bi ove klase implementirale pomenuti interface. Kreirali bi i klasu KolekcijaRecepata u koju bi smjestili listu recepata koju treba sortirati, te bi dodatni atribut te klase bio i tipa ISortiranje i određivao bi upravo koja strategija sortiranja će biti upotrijebljena prilikom sortiranja ove liste recepata, odnosnokoja verzija metode sortiraj() će biti pozvana.

## State patern

State patern je dinamička verzija Strategy paterna i omogućava promjenu načina ponašanja objekta u zavisnosti od stanja u kom se nalazi. Za razliku od Strategy paterna, objekat sam mijenja svoje stanje, ono nije izabrano od strane klijenta.

Recimo da želimo omogućiti korisnicima da žele svoj profil učiniti privatnim. Tada bi klasa Korisnik imala dva mode-a: privatni i javni, odnosno ovo bi bile klase koje implementiraju interface IMode. U zavisnosti od klase, odnosno mode-a bi omogućili drugim korisnicima prikaz profila tog korisnika, odnosno poziv metode prikaziProfil klase KorisnikController.

## TemplateMethod patern

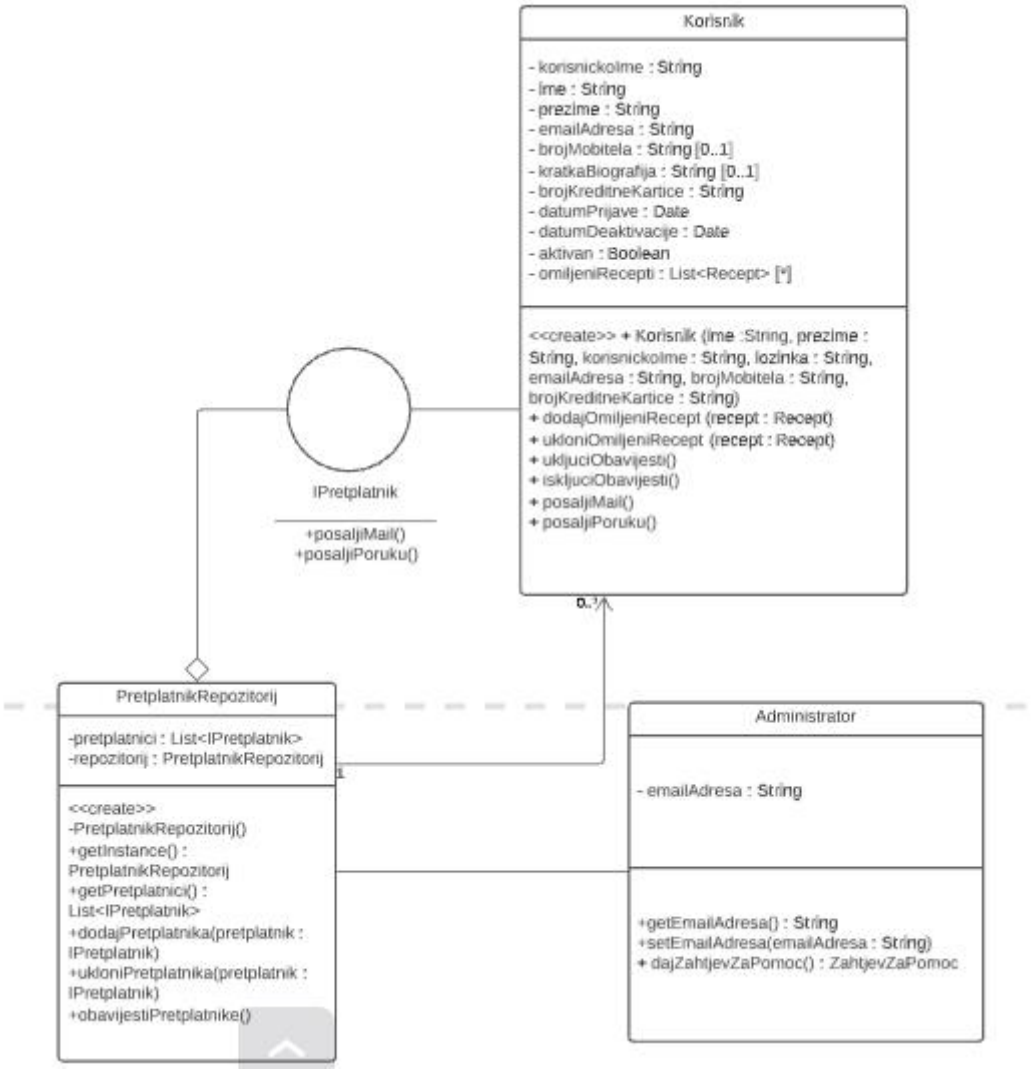
Ovaj patern omogućava izdvajanje pojedinih dijelova algoritama u podklase. Ovi izdvojeni dijelovi se mogu implementirati različito.

Recimo da želimo uvesti još jedan tip korisnika koji imaju dodatne opcije. Neka su to certifikovani kuhari koji imaju opciju dodavanja linkova na stranicu svoga restorana ili svoje kuharice na svoj profil. U tom slučaju mogli bismo iskoristiti TemplateMethod patern da razdvojimo metodu urediProfil() na dijelove, pri čemu se glavni dijelovi implementiraju isto za sve klase korisnika, a pojedini dijelovi drugačije za svaku klasu. Uveli bi apstraktnu klasu Korisnik koja bi imala metode dodajSliku(), dodajOpis() i promijeniKorisnickoIme() implementirane, dok bi metoda dodajPoveznice() bila apstraktna. Klase KorisnikAmater i KorisnikChef bi naslijeđivale ovu klasu, a morale bi zasebno implementirati dodajPoveznice(). U slučaju klase KorisnikChef bi implementirali ovu metodu tako da omogućimo korisniku dodavanje potrebnih linkova/poveznica, dok bi u slučaju klase KorisnikAmater onemogućili korisniku upotrebu ove funkcionalnosti i obavijestili ga o razlogu prikladnom porukom.

## Observer patern

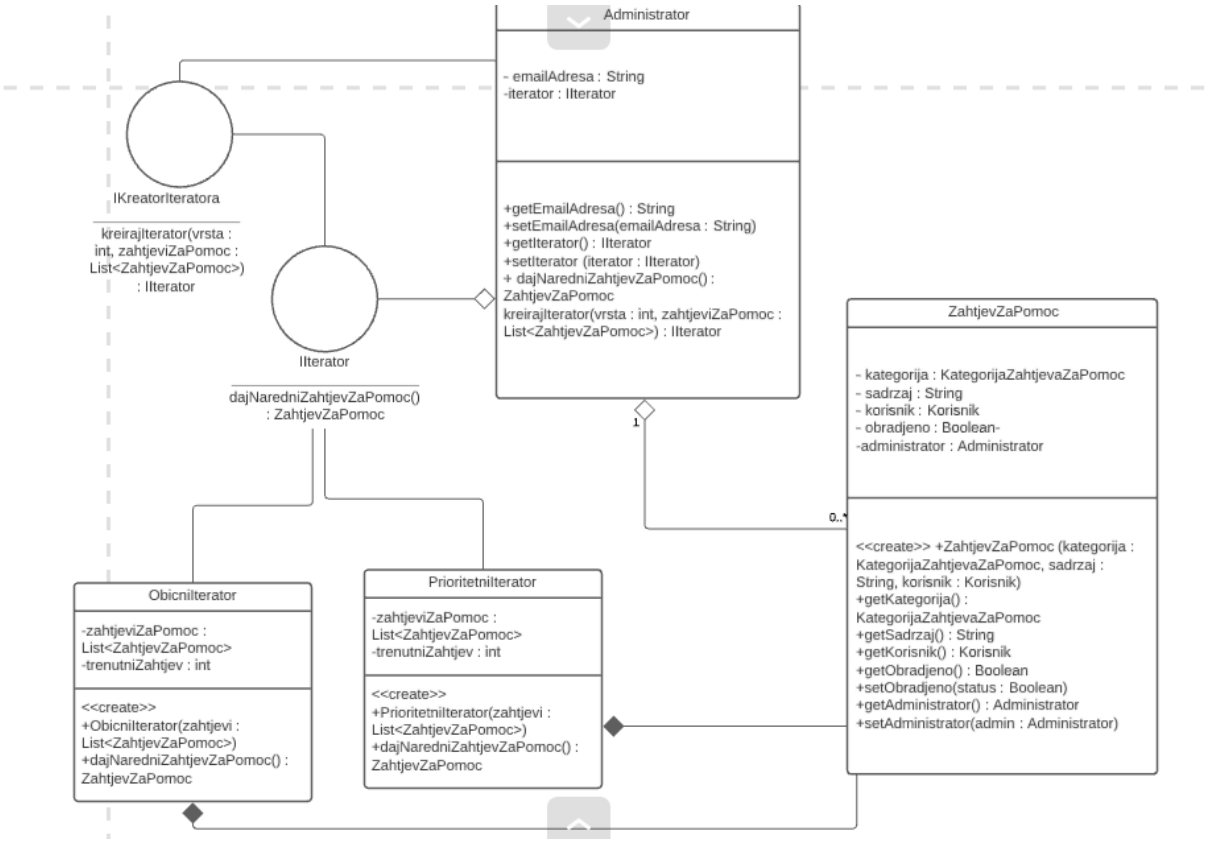
Observer patern služi za uspostavljanje takve relacije između objekata, da promjene koje se dešavaju nad jednim objektom potiču slanje obavijesti drugim zainteresiranim objektima o tim promjenama.

U našoj aplikaciji zainteresirani korisnici će moći primati obavijesti o ažuriranju kategorije najboljih recepata putem e-maila i poruke na mobilni uređaj. Kako bi implementirali ovu funkcionalnost, poslužićemo se Observer paternom. Kreiraćemo interface IPretplatnik koji pruža metode posaljiMail() i posaljiSMS(), a ovaj interface implementiraće klasa Korisnik. Kada se korisnik pretplati na primanje obavijesti, on se dodaje u listu pretplatnika koja se čuva u Singleton klasi PretplatnikRepozitorij. Pristup ovoj klasi omogućen je klasi Administrator. U klasi PretplatnikRepozitorij imamo metodu obavijestiPretplatnike(), pa unutar te metode upravo možemo za svakog od pretplatnika koji se nalazi u listi u klasi PretplatnikRepozitorij pozvati metode posaljiMail() i posaljiSMS() da obavijestimo te korisnike o izvršenoj promjeni. U budućnosti možemo proširiti ovaj patern dodavanjem novih vrsta obavijesti, a korištenjem iste klase za pretplatnike.



## Iterator patern

Iterator patern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolekcija strukturirana.
Ovaj patern iskoristićemo kako bi omogućili da administrator prolazi kroz listu zahtjeva za pomoć tako da mu zahtjevi s većim prioritetom, odnosno oni na koje što prije treba da odgovori, dolaze prvi. Ta klasifikacija po prioritetu se može vršiti upravo na osnovu kategorije zahtjeva za pomoć (Neprimjereni sadržaj ima najveći prioritet (1), a Pitanja i sugestije najmanji (4)). Dodajemo u klasu Administrator atribut tipa IIterator, odnosno interface koji implementira metodu dajNaredniZahtjevZaPomoc(). Ovaj interface implementiraju klase PrioritetniIterator i ObicniIterator, koje interno sadrže i listu zahtjeva za pomoć. U PrioritetniIterator klasi bi metoda dajNaredniZahtjevZaPomoc() bila implementirana tako da vraća zahtjev za pomoć iz liste s najvišim prioritetom, dok bi u klasi ObicniIterator ona prosto vraćala sljedeći objekat iz te liste. Također dodajemo i IKreatorIteratora interface koji pruža metodu kreirajIterator(int vrsta, List<ZahtjevZaPomoc> zahtjevi) za kreiranje potrebnog iteratora, a koji implementira klasa Administrator.




## Chain Of Responsibility patern

Chain Of Responisbility patern omogućava proslijeđivanje zahtjeva kroz lanac handler klasa. Kada primi zahtjev, svaka handler klasa odlučuje da li će ga obraditi ili proslijediti sljedećoj u lancu.

Ovaj patern bi se mogao primijeniti kod akcije dodavanja recepta. Napravili bismo klasu ReceptMaker koja interno sadrži objekat tipa Recept, te jednu instancu same klase

ReceptMaker. To bi bio bazni handler, a ostale handler klase bi naslijeđivale tu klasu. To bi bile klase ObradaSlike, ObradaVidea i ObradaSastojaka. Osim toga, klasa ReceptMaker implementirala bi interfejs IHandler koji zahtijeva implementaciju metode proslijedi(). Unutar klase ReceptMaker implementirala bi se metoda obradi() koja bi obradila osnovne podatke o receptu, a dalje proslijedila recept na obradu ostalim handler klasama, pri čemu bi svaka od tih klasa 'odradila svoj posao'. U slučaju da jedna od klasa ne uspije obradu zbog nekog problema, zaustavio bi se proces.

## Mediator patern

Mediator patern otklanja međuovisnosti objekata tako što zabranjuje direktnu komunikaciju između tih objekata i dopušta im samo komuniciranje preko medijator objekta.

Ovaj patern bi se mogao iskoristiti ukoliko bismo uveli opciju dopisivanja sa drugim korisnicima, odnosno chat. Tada bi klasa Chat bila medijator između korisnika koji se dopisuju. Ova klasa bi implementirala interfejs IMediator sa metodom provjeriPoruku(). Ova metoda bi se pozivala pri svakom slanju poruke od strane jednog korisnika ka drugom i služila bi za provjeru sadržaja poruke, tako što bi odbacila sve poruke u kojima detektuje neprimjeran sadržaj i sumnjive linkove. Interno bi klasa Chat sadržala listu svih korisnika u sistemu, odnosno potencijalnih klijenata. Klasa Korisnik bi sadržala interno objekat tipa IMediator, te liste primljenih i poslanih poruka.