# PATERNI PONAŠANJA

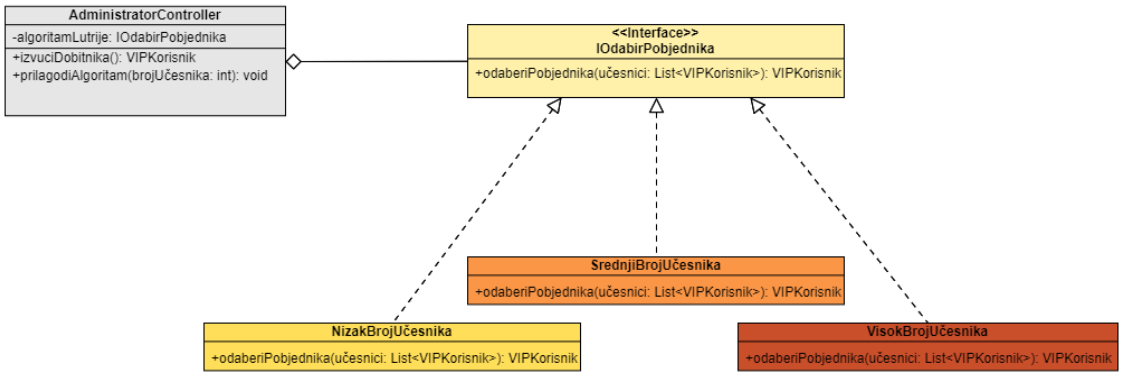
Dizajn paterni ponašanja imaju tri osnovna područja primjene, a to su: organizacija algoritama, dodjeljivanje odgovornosti, te organizacija komunikacije između objekata.

U nastavku će biti prezentovano nekoliko konkretnih dizajn paterna koji se svrstavaju pod ovu kategoriju, pri čemu će uz njihov kratak opis biti naveden i primjer primjene istih u našem sistemu.

## 1. Strategy patern

Uloga *Strategy* paterna je izdvajanje algoritma iz matične klase i uključivanje istog u posebnu klasu. Ovaj patern je pogodan u situacijama kada za neki problem postoje različiti primjenjivi algoritmi (tj. strategije).

*Strategy* patern bi se u našem sistemu mogao implementirati ako bi iskoristili ideju da se za odabir mjesečnog dobitnika nagradnih poena koriste različiti algoritmi (tj. strategije) u zavisnosti od toga koliko je korisnika uključeno u tu nagradnu igru. Naravno, nije realistično da za svaki mogući različiti broj korisnika postoji poseban algoritam, međutim, ideja bi se mogla realizovati na način da za sve vrijednosti iz nekog opsega koristimo jedan algoritam. Recimo da broj korisnika iz opsega od 1 do 20 klasifikujemo kao nizak, od 21 do 50 kao srednji, a preko 51 kao visok. Tada bi zajednički interfejs mogli nazvati *IOdabirPobjednika*, odakle bi bile izvedene klase *NizakBrojUčesnika*, *SrednjiBrojUčesnika* i *VisokBrojUčesnika*. Svaka od tih klasa bi implementirala algoritam *odaberiPobjednika()* na sebi karakterističan način. *Context* klasa bi predstavljala administratora u sklopu kojeg bi se nalazila instanca interfejsa *IOdabirPobjednika* pomoću koje bi se algoritam za odabir pokretao, a pored toga u njoj bi se nalazile još metoda *izvuciDobitnika()*, iz koje bi se samo pozivala metoda spomenutog interfejsa nad njegovom instancom, i metoda *prilagodiAlgoritam(int brojUčesnika)*, koja bi omogućavala promjenu korištenog algoritma po potrebi. Na slici ispod prikazan je isječak iz dijagrama klasa na kojem je jasnije vidljiv način primjene ovog paterna u našem sistemu.



## 2. State patern

*State* patern je dinamička verzija *Strategy* paterna. Kod ovog paterna je princip takav da objekat mijenja način ponašanja na osnovu trenutnog stanja.

*State* patern bi se u naš sistem mogao ugraditi ukoliko bi se ograničio broj ulaznica koje se mogu kupiti za neki sportski događaj. Dakle, registrovanom korisniku je omogućena kupovina ulaznica za neki sportski događaj, međutim, nakon što pređe određeni limit broja kupljenih ulaznica, sistem ulazi u stanje u kojem tom korisniku više nije omogućena kupovinu ulaznica za taj konkretan sportski događaj. U ovom kontekstu, klasa *RegistrovaniKorisnikController* bi i dalje korištenjem metode *kupiUlaznicu()* omogućavala kupovinu ulaznica, međutim, sada bi u tu klasu dodali i novi atribut koji bi bio instanca interfejsa *IStanjeKupovine*. Interfejs *IStanjeKupovine* bi imao metodu *realizujKupovinu()* koja bi se u metodi *kupiUlaznicu()* klase *RegistrovaniKorisnikController* koristila za kupovinu. Ovaj interfejs bi implementirale dvije klase, od kojih bi jedna predstavlja korektnu implementaciju kupovine (npr. klasa *UnutarLimita*), a druga bi predstavljala stanje kada je limit broja ulaznica probijen i umjesto realizovanja kupovine davala bi neki izuzetak (npr. klasa *LimitProbijen*). Stanje, tj. konkretna klasa koja implementira ovaj interfejs, bila bi mijenjana na osnovu broja ulaznica koje je taj korisnik kupio za taj događaj.

## 3. Iterator patern

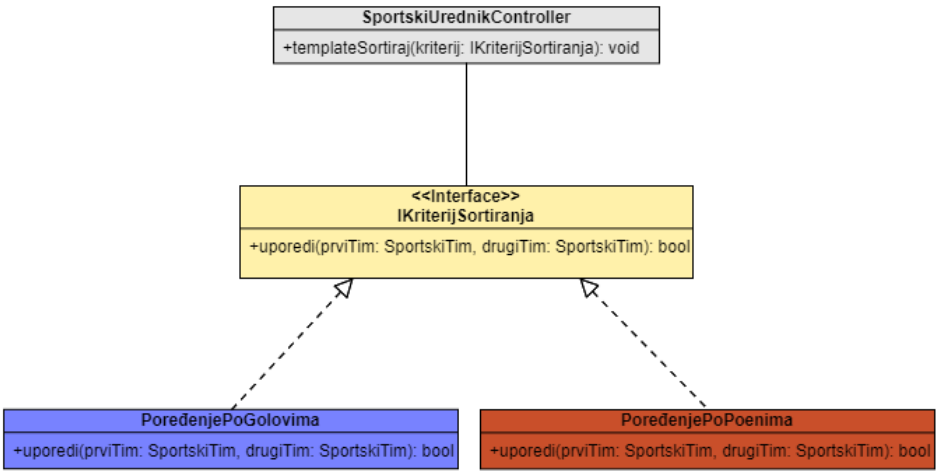
*Iterator* patern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolekcija strukturirana.

*Iterator* patern bi se mogao implementirati u naš sistem. Pretpostavimo da tabelu neke lige modeliramo kao kolekciju podataka. U tom slučaju ulogu *Collection* klase imala bi klasa *TabelaLige*. Ona bi, pored nekih informacionih atributa kao što je naziv, sadržavala i neki niz ili listu svih timova te lige. U ovom slučaju bi nam korištenje *Iterator* paterna omogućilo da se kroz neku tabelu možemo kretati *foreach* petljom. Klasa *TabelaLige* implementirala bi interfejs *IEnumerable* i njegovu metodu *getEnumerable()* bi definisala tako da se iterirajući kroz *foreach* petlju dobiva prolazak kroz kompletnu ligu, tj. njezine timove. Dodatno bi mogli osigurati da je taj prolazak uvijek u opadajućem redoslijedu prema broju osvojenih poena ili prema nekom drugom kriteriju. Ulogu klase *Client* mogla bi imati klasa *SportskiUrednikController* koja bi koristila tu prednost koju bi nam ovaj patern pružio.

## 4. TemplateMethod patern

*TemplateMethod* patern omogućava izdvajanje određenih koraka algoritma u odvojene podklase. Treba istaći da se struktura algoritma ne mijenja, samo se mali dijelovi operacija izdvajaju i ti se dijelovi mogu implementirati različito.

*TemplateMethod* patern bi se mogao u naš sistem ugraditi ukoliko bi se uvela mogućnost prikaza timova u tabeli, ali sortiranih u odnosu na različite kriterije. Preciznije rečeno, ako bi omogućili korisniku da ima više različitih opcija za sortirani prikaz timova neke tabele ovaj patern bi bio idealan alat za realizaciju toga. Naime, recimo da je omogućen izbor sortiranja u odnosu na broj postignutih golova i u odnosu na broj ostvarenih poena. Tada bi klasa *SportskiUrednikController* u sebi sadržavala metodu *TemplateSortiraj(IKriterijSortiranja kriterij)*, a interfejs *IKriterijSortiranja* bi sadržavao opšti oblik operacije kojom se u metodi *TemplateSortiraj* vrši upoređivanje elemenata, pri čemu bi tu operaciju mogli nazvati *uporedi(SportskiTim prviTim, SportskiTim drugiTim)*. Klasa *PoređenjePoGolovima* bi implementirala interfejs *IKriterijSortiranja* i realizovala metodu uporedi na svoj način, dok to isto vrijedi i za klasu *PoređenjePoPoenima*. Na slici ispod prikazan je isječak iz dijagram klasa na kojem je jasnije vidljiv način primjene *TemplateMethod* paterna u našem sistemu.




# 5. Observer patern

Uloga *Observer* paterna je uspostavljanje relacije između objekata takve da kada jedan objekat promijeni stanje svi drugi zainteresirani objekti bivaju obavješteni.

*Observer* patern bi se mogao ugraditi u naš sistem ukoliko bi dodali opciju da registrovani korisnik može primati obavijesti svaki put kada neki novi sportski događaj postane aktuelan. U tu svrhu, klasa *SportskiUrednikController* imala bi ulogu *Subject* klase. *IPorukaObavijesti* bi bio interfejs koji bi imao ulogu *IObserver* interfejsa, tj. u njemu bi se nalazila metoda koja omogućuje da registrovani korisnik prihvati obavijest o novosti. *RegistrovaniKorisnikController* bi imao ulogu *Observer* klase i on bi implementirao interfejs *IPorukaObavijesti* i njegovu metodu *primiPoruku()* koja bi imala ulogu *Update* metode. U klasu *SportskiUrednikController* ubacili bi metodu koja bi služila za realizaciju slanja poruke (npr. putem e-maila) i koja bi imala ulogu *Notify* metode. Ta metoda bi automatski pozivala metodu *primiPoruku()* nad svim korisnicima koji su

pretplatnici ove usluge. Također, u klasu *SportskiUrednikController* ubacili bi i metodu kojom se korisnik može registrovati, tj. prijaviti, da želi primati takve obavijesti.

# 6. Chain of Responsibility patern

*Chain of Responsibilty* patern omogućava da se neki kompleksniji korisnički zahtjev obrađuje korištenjem nekoliko različitih objekata. Naime, svaki od objekata imao bi određeni zadatak koji bi predstavljao dio kompletne funkcionalnosti. Treba istaći da je najčešće bitan i redoslijed u kojem se tim objektima prenosi zahtjev. Na taj način zahtjev se obrađuje prijenosom sa objekta na objekt pri čemu svaki od tih objekata realizuje po jedan dio kompletne funkcionalnosti. Na taj način spomenuti objekti grade jedan lanac.

U naš sistem ovaj patern bi se mogao implementirati ukoliko bi se realizacija livechat-a malo kompleksnije implementirala. Naime, kada bi korisnik uputio svoju poruku drugim korisnicima koji učestvuju u razgovoru, taj zahtjev bi se mogao proslijediti lancu objekata od kojih bi svaki vršio po jednu radnju u sklopu obrade korisnikove poruke. Naprimjer, prvi objekt u lancu bi preuzimao poruku direktno od korisnika, drugi bi ispitivao primjerenost sadržaja poruke, a treći bi upućivao poruku ostalim korisnicima koji učestvuju u razgovoru. Primijetimo da je u navedenom primjeru bitan redoslijed u kojem spomenuti objekti prosljeđuju zatraženi korisnički zahtjev jedan drugom.

# 7. Mediator patern

*Mediator* patern ima ulogu smanjivanja broja direktnih veza između klasa. U suštini, ovim paternom se vrši centralizacija sistema.

Ovaj patern bi se mogao implementirati u našem sistemu ukoliko bi se htio ubaciti neki posrednik koji reguliše slanje poruka u livechat. Dakle, kada korisnik uputi poruku ostalim sudionicima chata, mediator bi imao ulogu da provjeri da li je sadržaj te poruke primjeren i samo ako jeste pusti tu poruku u chat.