

Paterni ponašanja

Strategy pattern

Strategy patern izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Pogodan je ukoliko postoje različiti primjenjivi algoritmi za neki problem. Potencijalna mjesta za primjenu ovog paterna vidimo pri sortiranju te pretraživanju podataka o pacijentima.

Uposlenici klinike bi mogli odabrati neki algoritam za sortiranje medicinskih kartona preko klase Nalaz. Ukoliko razmatramo primjenu ovog paterna pri sortiranju podataka o pacijentima u sistemu, možemo to realizirati na sljedeći način. Možemo omogućiti dva načina sortiranja, po imenu i po prezimenu. Da bismo implementirali ovaj patern, potrebno je da imamo interfejs ISortiraj u kojem imamo metodu sortiraj() te konkretne implementacije ovog interfejsa, što su ovom slučaju klase SortiranjePoImenu i SortiranjePoPrezimenu. U ovim klasama se nalazi implementacije metode sortiraj().

Pozivanje ovih metoda može biti aktivirano pritiskom na dugme ili neku opciju na grafičkom okruženju aplikacije.

Observer patern

Glavna uloga ovog paterna je da uspostavi relaciju između objekata tako da kada jedan objekat promijeni stanje, drugi zainteresirani objekti se obavještavaju.

U našem projektu, ovaj patern smo iskoristiti kada korisnik odabere slobodan termin za pregled te taj zahtjev biva odobren korisnik dobije obavještenje. Također ukoliko se desi neka promjena korisnik biva obaviješten.

Unutar klase Termin nalazila bi se broj telefona pacijenta na koji se treba poslati obavještenje. Trebali bi kreirati interfejs IPosaljiObavjest koja bi imala funkciju +posaljiSMS(pacijent: Pacijent, termin: Termin). Funkcija ovog interfejsa bi se pozivala prilikom neke promjene nad klasom Termin.

State pattern

State patern koristimo kada imamo objekt koji se ponaša različito ovisno o trenutnom stanju u kojem se taj objekat nalazi. Ovaj pater možemo realizirati tako što ćemo napraviti klase koje će tačno opisivati trenutno stanje objekta.

Ovaj patern bi mogli iskoristiti kada bi na osnovu trenutnog broja zakazanih (i odobrenih) termina od strane korisnika formirali određena stanja. Svaki korisnik sa jednim aktivnim terminom ne ostvaruje nikakav popust/privilegije i nalazi se na stanju Pacijent1, dok za svaka dva termina, pacijent prelazi u stanje Pacijent2, Pacijent3... Ove izmjene bi se izvršavale automatski u zavisnosti od broja trenutno zakazanih termina.

Iterator pattern

Iterator pattern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolekcija struktuirana. Struktura iterator patterna se sastoji iz nekoliko dijelova. Prvi je kolekcija objekata kroz koju se želi iterirati.

Ovaj pattern možemo iskoristiti za prolazak kroz listu doktora klinike unutar klase UposlenikUsluga. Da bi se implementirao ovaj pattern, potrebno je kreirati klasu Iterator koja će implementirati interfejs IIterator. Pomenuta klasa bi imala attribute listu tipa UposlenikUsluga i indeks trenutnog elementa i metode `dajSljedeceg()` i `dajTrenutnog()`.

TemplateMethod pattern

Ovaj pattern omogućava izdvajanje određenih koraka algoritma u odvojene klase. Sama struktura algoritma se ne mijenja, nego se mali dijelovi operacija izdvajaju i oni se mogu različito implementirati.

Ovaj pattern bismo mogli iskoristiti u našem sistemu npr. za pretraživanje pacijenata i usluga. To bismo uradili na način da imamo klasu Pretrazivanje i u toj klasi metodu `pretrazi(IPretrazivanje obj, int nacinPretrazivanja)`. Zatim bismo dodali interfejs `IPretrazivanje` koji bi sadržavao definicije metoda `pretraziPoImenu()`, `pretraziPoPrezimeni`, `pretraziPoNazivuUsluge()`, itd. Pošto nam se lista korisnika i termina nalazi u klasama `Osoba` i `Termin`, mogli bismo staviti da klase `Osoba` i `Termin` implementira interfejs `IPretrazivanje`, i u njoj dodati implementacije metoda koje su navedene u interfejsu.

Chain of responsibility pattern

Ovaj pattern predstavlja listu objekata. Ukoliko objekat ne može da odgovori na zahtjev, proslijeđuje ga narednom objektu u nizu. U našem sistemu ovaj pattern bi mogli primijeniti kada bismo dodali funkcionalnost za editovanje termina, odnosno ukoliko korisnik želi da odabere termin u ordinaciji koji mu više odgovara da može da uredi već postojeći termin sa novim ažuriranim podacima. Implementacija bi izgledala na sljedeći način. Kreirali bismo novu klasu `PromjenaZakazanogTermina` unutar koje bismo imali metodu `KreirajTermin()` koja bi pozivala klasu `ZahjevZaPromjenu` koja služi za kreiranje zahtjeva za promjenu mjesta. Nakon toga uloga administratora sistema bi bila da odobri ili odbije zahtjev. Ukoliko se desi da admin odobri zahtjev, kreirala bi se nova instanca klase `Termin`.

Mediator pattern

Mediator definiše objekat koji enkapsuliše način na koji upravo ovi povezani objekti komuniciraju. Ovaj vrlo koristan pattern se koristi kada u sistemu postoji mnogo povezanih klasa koje zavise jedna od druge. Unutar našeg sistema ovaj pattern bismo mogli iskoristiti na način da napravimo `IMediator` interfejs unutar kojeg bi imali metodu `DaLiJeDoktor` koja vraća vrijednost `true` ukoliko je uposlenik doktor a `false` ukoliko je neko od drugih uposlenika klinike. Također bi napravili klasu `PristupKartonu` pomoću koje bismo kontrolisali pristup kartonima pacijenata. Na ovaj način bismo neke od povjerljivih podataka koje se nalaze unutar kartona pacijenta enkapsulirali i ograničili im pristup.