Univerzitet u Sarajevu
Elektrotehnički fakultet u Sarajevu
Odsjek za računarstvo i informatiku

ReadTheWorld

Strukturalni patterni

Objektno orijentisana analiza i dizajn

Iris Pjanić
Adna Husičić
Mirza Kadrić

Strukturalni patterni

Patterni koji su dodani: Composite pattern i Flyweight pattern (umjesto Proxy).

Adapter pattern

Adapter pattern koristi se u situacijama kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu. Što se tiče našeg sistema, smatramo da nije bilo potrebe za dodavanjem ovog patterna. Razlog je u tome što bi najočitija primjena adapter patterna bila za dodavanje novih mogućnosti za pregled rada, slično kao u zadatku sa tutorijala, međutim ovdje se vrši pregled ili objavu rada koji je u pisanom formatu i kao takav nema jake potrebe za dodavanjem drugih mogućnosti. Ukoliko bismo i pored toga htjeli da koristimo ovaj pattern, mogli bismo, recimo, uzeti primjer slučaja objave rada za takmičenje. Recimo da imamo poseban proces kojim se treba objaviti rad za takmičenje. U tom slučaju imali bismo Client klasu - Rad, interfejs ITarget koji definiše metodu objavaRada(), klasu Adapter koja implementira novi zahtijevani interfejs, te Adaptee klasu (koja zapravo predstavlja takmičarski rad) koja definira postojeći interfejs koji treba prilagoditi (u ovom slučaju npr. objaviRadZaTakmičenje()).

Facade pattern

Fasadni pattern ima za osnovni cilj da korisnicima pojednostavi korištenje kompleksnog sistema. Dakle, u slučaju da sistem ima više podsistema sa povezanim implementacijama i apstrakcijama poželjno je korištenje ovog patterna. Ni ovaj pattern nećemo primjenjivati jer je u našem sistemu komunikacija korisnika i sistema jasno definisana i jednostavna - korisnik ima forme za objavu, pretragu, ocjenjivanje itd. Ove operacije su dovoljno jednostavne i nema mnogo procesa koji se izvršavanju dok korisnik ne dobije odgovarajući odgovor sistema. Ukoliko bismo ipak htjeli da koristimo ovaj pattern,

prvo trebamo razmotriti koji proces je najsloženiji i ”najudaljeniji” od korisnika. U ovom slučaju to bi moglo biti takmičenje ili eventualno pretraga radova. Recimo da smo razmatramo takmičenje, jer je ono poprilično kompleksan proces i dosta stvari se dešava ”u pozadini”, bez znanja korisnika (što je zapravo i smisao takmičenja, da korisnik zna što manje o procesima između slanja svog rada i pregleda rezultata). Fasadna klasa bi se mogla zvati npr. FasadaTakmicenje i imala bi definisane prethodno spomenute metode. Neke od njih mogle bi biti: getPrijavljeniRadovi(), getRadoviKorisnika(Korisnik korisnik), izbaciRadIzKonkurencije(Rad rad), postaviTopN(int n) itd.

Proxy pattern

Najčešća upotreba ovog patterna jeste za zaštitu prisutpa podacima odnosno osiguravanje objekata od nedozvoljene upotrebe. U našem sistemu sama klasa Administrator naglašava potrebu za ovim patternom, jer administrator ima određene privilegije i prava pristupa nekim podacima kojima ostali korisnici ne mogu pristupiti, npr. podacima o prijavljenim radovima. Iz tog razloga potrebna je provjera podataka, te odobravanje pristupa jedino ukoliko podaci onoga ko traži pristup odgovaraju podacima administratora. Ipak, zbog drugačijeg pristupa ovakvim problemima odlučili smo umjesto ovog patterna iskoristiti Flyweight pattern koji će biti objašnjen na kraju dokumenta.

Decorater pattern

Osnovna namjena Decorator paterna je da omogući dinamičko dodavanje novih elemenata i ponašanja (funkcionalnosti) postojećim objektima. Ovaj pattern je koristan ukoliko želimo vršiti različite nadogradnje istih vrsta objekata bez potrebe za uvođenjem velikog broja izvedenih klasa. Trenutni zahtjevi u našem sistemu ne ukazuju na potrebu za ovim patternom ali se

veoma jednostavno mogu vidjeti situacije u kojima bismo mogli modificirati zahtjeve tako da se iskoristi decorater pattern. Recimo, najjednostavniji primjer je uređivanje radova. Ukoliko bismo dozvolili da se rad nakon objavljivanja može uređivati mogli bismo da iskoristimo ovaj pattern. Omogućavanjem uređivanja rada nakon objave korisnici bi mogli u potpunosti izmijeniti sadržaj rada i time mu promijeniti i sam žanr, kategoriju i slično. Ovo se ne smije dopustiti pa time odbacujemo potrebu za ovim patternom. Eventualna mogućnost je recimo provjera da izmjena ne uključuje više od određenog postotka cjelokupnog rada, međutim to su nepotrebne komplikacije i nećemo ih uvoditi. Što se tiče uređivanja profila, i to je jedna od mogućnosti ali ono je ipak zamišljeno drugačije i njegovo uređivanje nije baš tipično uređivanje koje bismo uveli korištenjem decorater patterna.

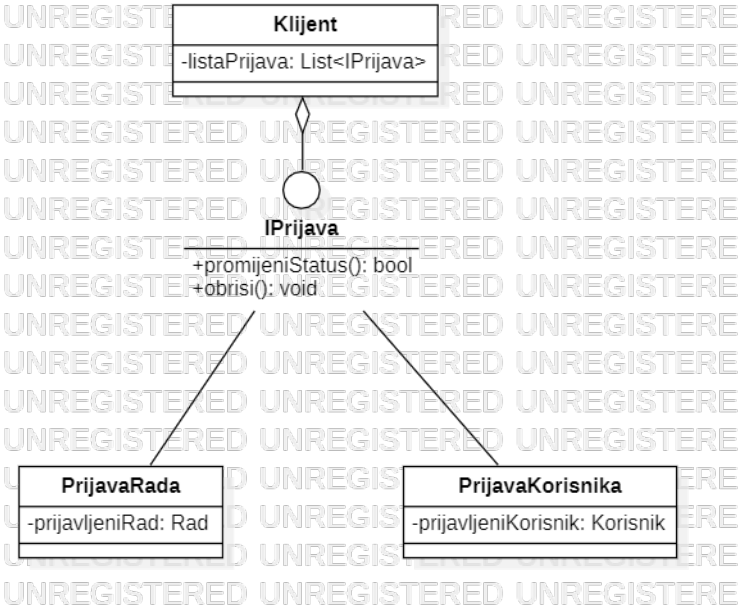
Bridge pattern

Osnovna namjena Bridge paterna je da omogući odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije. Bridge patern pogodan je kada se implementira nova verzija softvera a postojeća mora ostati u funkciji. Ovaj pattern se koristi i za sisteme za razmjenu poruka. U jednoj od prvih vježbi smo imali ideju da korisnici mogu slati jedni drugima poruke, međutim taj zahtjev smo odbacili, ali kada bismo i dalje imali tu funkcionalnost koristio bi se bridge pattern. U trenutnoj verziji sistema ovaj pattern bi se mogao iskoristiti i za ažururanje pobjednika na takmičenjima. Recimo da za takmičenje imamo top listu koju formiraju administratori. Međutim, kako bi podsticali korisnike na što veću interakciju sa sistemom, želimo da dodamo i top listu po izboru korisnika. Dakle, sada imamo 2 kategorije ljestvica: top radovi po izboru administratora i top radovi po ocjenama korisnika. Pretpostavimo da želimo objaviti rezultate takmičenja. Tada bismo mogli imati recimo klase RezultatiA i RezultatiB koji bi implementirali interfejs koji će formirati

ljestvicu za iste radove ali svaki prema svom kriteriju. Time bismo odvojili apstrakciju od implementacije i formirali bismo novu ljestvicu(top radovi po izboru korisnika) a stara ljestvica bi i dalje ostala u funkciji.

Composite pattern
Composite pattern služi za kreiranje hijerarhije objekata. Osnovna namjena ovog patterna je da omogući formiranje strukture stabla pomoću klasa, u kojoj se individualni objekti i kompozicije individualnih objekata jednako tretiranju. U našem sistemu možemo iskoristiti ovaj pattern tako što ćemo Prijavu realizirati kao interfejs, a zatim tako kreiran interfejs će naslijediti PrijavaKorisnika i PrijavaRada. Ovaj pattern je izabran jer ima dosta situacija kada se tražena operacija treba koristiti u sklopu više prijava ali i kada je bitno da se tražena operacija izvrši samo nad jednom prijavom, ne mijenjajući sadržaj ostalih. Ako imamo listu prijava za neki rad, brisanjem tog rada trebamo odmah obrisati i sve prijave za taj rad. S druge strane, kada vidimo da neki rad ima određeni broj prijava, mi prolazimo kroz te prijave i gledamo da li je svaka od njih validna, te svakoj prijavi mijenjamo status. Tako jedna prijava može imati status NaČekanju, dok smo za drugu prijavu za isti rad utvrdili da ima status "Neosnovana". Pošto je druga prijava neosnovana, nećemo brisati rad (zajedno sa svim njegovim prijavama), već prelazimo na razmatranje prijave koja ima status "NaČekanju". Composite patern sastavljen je od sljedećih klasa:
Klijent – manipulira objektima u kompoziciji preko IComponent interfejsa, u našem slučaju on upravlja prijavama. Sadrži listu elemenata tipa IPrijava, te ćemo nad tim elementima pozivati metode promijeniStatus() i obriši() koje su definisane u IPrijava. Načinom kako se realizira poziv tih funkcija upravljaju PrijavaRada i PrijavaKorisnika, svaka na svoj način.
IComponent – definira interfejs-operacije za objekte u kompoziciji i implementira defaultno ponašanje koje je zajedničko za objekte oba tipa (jedan objekat i kompoziciju

objekata). Te operacije mogu biti npr. obriši, promijeniStatus itd. U našem slučaju to je IPrijava.
Component – implementira interfejs klase za osnovne objekte - prijavu.
Composite – implementira interfejs (operacije) koji je primjenjiv na kompozitne objekte - lista prijava korištenjem implementacije za pojedinačne komponente.



Flyweight pattern
Ovaj pattern se koristi da se onemogući bespotrebno stvaranje instanci koje predstavljaju isti objekat. Bazira se na tome da se instantacija objekta vrši samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama tj. specifičnim stanjem, dok se u protivnom koristi postojeća instanca tj. bezlično stanje. Ovaj pattern ćemo koristiti za instantaciju takmičenja. Što se tiče radova, instanciranje velikog broja objekata nije moguće izbjeći obzirom da svaki od njih predstavlja jedinstven rad i ne možemo izvesti neki opći "bezlični" oblik za njega. Prva asocijacija je bila da pattern primijenimo na profile korisnika, obzirom da kada se korisnik tek registruje, njegov profil, bez obzira koliko unikatan, i dalje posjeduje neke karakretistike koje bi se mogle uzeti kao opće. Na primjer, kada se korisnik tek registruje znamo da će imati

titulu novog korisnika (newbie), da neće imati radova niti prijava i slično. Ipak, i tu postoje karakteristike koje su unikatne, jer svaki korisnik ima različit username, ime, prezime i sl. Nešto bolja primjena je da pattern iskoristimo za kreiranje takmičenja, što smo u ovoj verziji i uradili. U početku smo rekli da su takmičenja zamišljena da budu raznovrsna, ali bilo bi od koristi imati jedan opšti oblik za instanciranje takmičenja koji se u slučaju potrebe može preuređivati. Ovo je posebno korisno ukoliko imamo neka takmičenja koja se redovno održavaju u istom obliku, recimo sedmično takmičenje stripova. Za realizaciju patterna je potrebno imati klasu FlyweightFactory koja u sebi sadrži Flyweight interfejs. Klasa ConcreteFlyweight će implementirati taj interfejs i imat će konkretne realizacije takmičenja, Da se dobije jedan flyweight objekat na korištenje, poziva se metoda iz FlyweightFactory koja vraća traženi objekat.