Univerzitet u Sarajevu

Elektrotehnički fakultet u Sarajevu

Strukturalni dizajn paterni

FitnessFusion

Predmet: Objektno orijentisana analiza i dizajn

Naziv tima: Time – out

Članovi tima: Ahmetović Zerina

Bajrović Taner

Bosno Hamza

Brčaninović Hasan

Peljto Emina

Sarajevo, maj 2023

Strukturalni dizajn paterni:

Strukturalni dizajn paterni se koriste kako bi riješili uobičajene probleme prilikom dizajna klasa i objekata. Njihovom primjenom se postiže bolji dizajn sistema koji za posljedicu daje kvalitetniji softverski proizvod. U ovom dokumentu će se objasniti potreba za primjenom *Composite*, *Decorator* i *Proxy* dizajn paterna, te će se u sklopu ovog dokumenta također razmotriti primjena i drugih strukturalnih dizajn paterna za ovaj sistem.

Motivi zbog kojih se javlja potreba za primjenom dizajn paterna u našem sistemu su sljedeći problemi sa kojim smo se susreli prilikom dizajniranja klasnog dijagrama:

- Problem predstavljanja različitih *parametara* korisnika iz kojih se generišu različiti *rezultati* za odabrani *program aktivnosti*.
- Problem modeliranja *rasporeda* kao dinamične funkcionalnosti sistema.
- Način modeliranja i upotrebe različitih *termina* unutar korisnikovog rasporeda.
- Omogućavanja *paralelnog pristupa* rasporedu.
- Omogućavanje ograničenog *pristupa korisničkom računu* u sistemu.
- Predstavljanje načina plaćanja usluga u sistemu.

U nastavku slijedi detaljniji opisi problema zbog kojih su upotrebljeni *Composite* i *Decorator* dizajn paterni, te je ukratko predstavljen i način primjene *Proxy* dizajn paterna. Također su razmotreni i dodatni primjeri na kojima se mogu primjeniti i neki od ostalih strukturalnih paterna.

Composite dizajn patern:

Composite dizajn patern daje rješenje na problem predstavljanja strukture stabla (drveta) sačinjenog od različitih klasa. Pri tome se omogućava uniformni rad sa različitim kompozicijama izvedenim iz strukture stabla.

Ovaj patern smo primjenili kako bi riješili sljedeći problem koji se odnosi na dio sistema za parametre iz kojih se generišu korisnikovi rezultati.

Opis problema:

Dati sistem treba da omogući jedinstven način za pohranu vrijednosti parametara koji se mjere prilikom korisnikovih treninga. Iz datih parametara (kao što su: masa, squat, benchPress, tračnje i dr.) se generišu i prate korisnikovi rezultati tokom praćenja odabranog programa aktivnosti. Nakon što korisnik sistema odabire onaj predefinisani program aktivnosti koji mu najbolje omogućava da postigne svoje ciljeve, sistem omogućava treneru da prilikom unosa vrijednosti parametara unese vrijednosti i dodatnih parametara koji se mjere kada korisnika odabere program aktivnosti. U slučaju kada korisnik ne odabere niti jedan program aktivnosti, tada se treneru omogućava da unosi vrijednosti osnovnih parametara koji se uobičajeno mjere i koji su sastavni dijelovi svih programa aktivnosti.

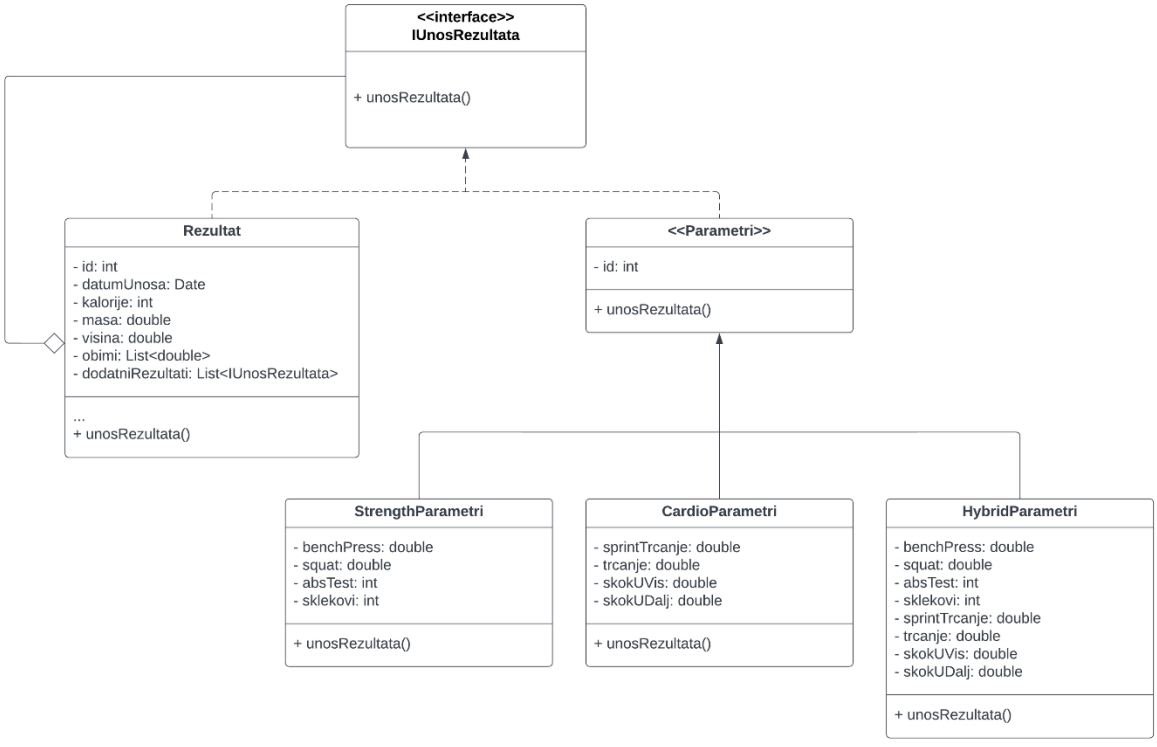
Opis rješenja:

Za ovaj problem smo primjenili Composite dizajn patern jer nam on omogućava da predstavimo traženu strukturu stabla parametara koji se mjere, te da na unificiran način vršimo unos. Struktura stabla se ugleda u tome što kao osnovu (korijenski čvor) koristimo klasu za *osnovne parametre*. Iz te klase se dalje pojavljuju i druge klase koje čine listove, a to su klase koje modeliraju parametre koji se mjere kada korisnik izabere određeni program aktivnosti. U skladu sa tim,

potrebno je omogućiti unificiran pristup prilikom unosa za sve kompozicije parametara. Na osnovu svega navedenog, pokazuje se da je ovaj patern odgovarajuće rješenje za ovaj problem.

Način na koji smo primjenili ovaj patern za naše modele je prikazan na narednoj slici, gdje su uloge pojedinih elemenata sljedeće:

- *IUnosRezultata* je interface koji omogućava definisanje uniformne metode za unos rezultata tj. vrijednosti izmjerenih parametara koji se prate.
- *Rezultat* je kompozitna klasa koja implementira *IUnosRezultata* interface i pohranjuje list (eng. *leaf*) objekte.
- *StrengthParametri, CardioParametri, HybridParametri* su leaf klase koje nasljeđuju *Parametri* klasu i svi oni implementiraju svoju metodu za unos.



Primjenom ovog paterna se postiže OCP koji nam omogućava jednostavno proširenje sistema u slučaju da se pojave nove kategorije parametara.

Decorator dizajn patern:

Decorator dizajn patern se koristi da se omogući dinamičko dodavanje novih elemenata i funkcionalnosti postojećim objektima. Cilj ovog paterna je da omogući slojevito poboljšanje nad komponentam, što se pokazalo kao dobro rješenje za problem koji se odnosi na dio sistema za planiranje korisnikovih aktivnosti.

Opis problema:

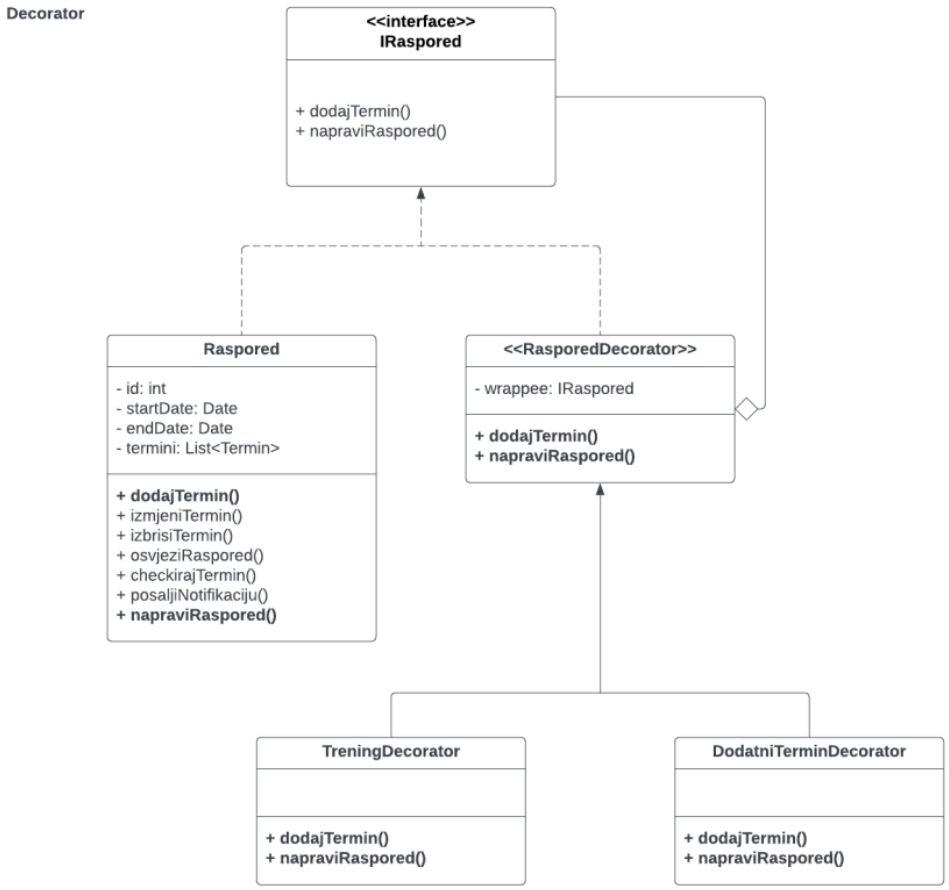
Sistem omogućava korisniku da kreira svoj raspored tako što dodaje termine u kojima navodi informacije za svaki od termina. Termini se dijele u dvije kategorije:

- Treninzi – predefinisani termini iz programa aktivnosti
- Dodatni termini – korisnički definisani termini

Korisnik ima mogućnost da dinamički mijenja i dodaje sadržaj na svom rasporedu.

Opis rješenja:

Odgovor na ovaj problem daje Decorator dizajn patern koji smo upotijebili kako bismo simulirali dinamički rad korisnikovog rasporeda. Ovim paternom smo postigli dinamičko dodavanje termina unutar rasporeda kao pojednostavljeni pristup modeliranju klasa koje se odnose na različite kategorije termina. U nastavku je prikazana slika koja prikazuje način na koji smo primjenili ovaj patern.



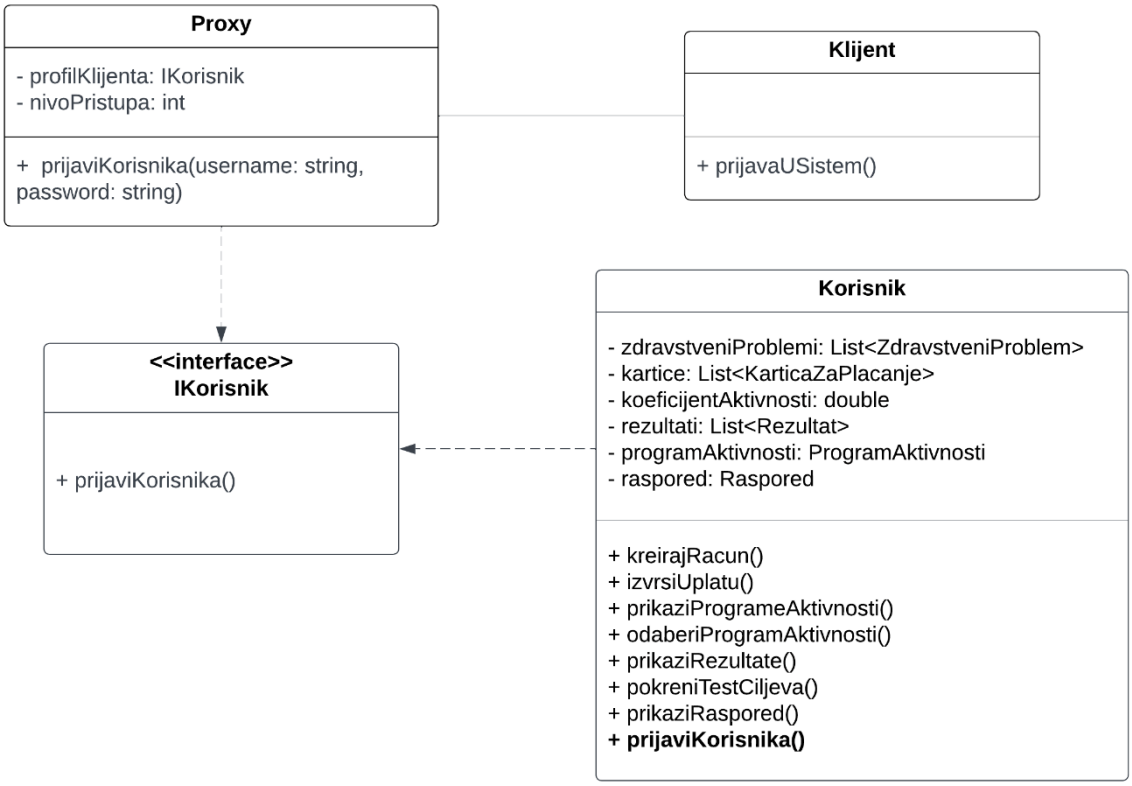
Elementi koji su predstavljeni na slici su:

- *IRaspored* je interface koji definiše metode *dodajTermin()* i *napraviRaspored()*
- *Raspored* je komponenta koja predstavlja konkretnu komponentu ovog rasporeda nad kojom se vrši dodavanje dekoracije
- *RasporedDecorator* je klasa koja odgovara *IRaspored* interface-u i ona koristi objekat nad kojim se dodaje dekoracija
- *TreningDecorator* i *DodatniTerminDecorator* su dekoracije koje se dodaju

Proxy dizajn patern:

Proxy dizajn patern koji se može primjeniti na ovaj sistem jeste *remote proxy* koji se tipično koristi za funkcionalnosti ograničenog pristupa objektu, kao što je

prijava u sistem. U ovom sistemu smo upotrijebili ovaj patern kao bismo podržali da naš sistem vrši kontrolu pristupa. Način na koji je primjenjen ovaj patern je prikazan na narednoj schemi.



U nastavku slijedi analiza potencijalne primjene paterna koji bi se mogli upotrijebiti za ovaj sistem.

Adapter dizajn patern:

Adapter dizajn patern omogućava prilagođavanje interface-a postojeće klase tako da se ona može prilagoditi za rad sa drugačijim interface-ima. U našem sistemu se ovaj patern može upotrijebiti u slučaju da koristimo eksterni servis. Na primjer, u slučaju da želimo koristiti eksterni sistem za analizu korisnikovih rezultata, bilo

bi potrebno adaptirati podatke na način koji dati sistem zahtjeva npr. u JSON formatu. Ovaj patern bi primjenili na taj način što bi kreirali interface koji bi se implmentirao u adapter klasi, te service klasu kojom bi se omogućile usluge eksternog servisa.

Facade dizajn patern:

Facade patern se koristi u situacijama kada želimo da na jednostavan način pristupimo kompleksnim podsistemima unutar sistema. U našem slučaju, ovaj patern bi se mogao primjeniti za klasu *Korisnik* koja pristupa različitim dijelovima našeg sistema. Da bismo to postigli, potrebno je kreirati klasu *Facade* koju bi koristila naša *Korisnik* klasa kako bi pristupila dijelovima podsistema. Također, kako kompleksnost ovog sistema bude rasla, tako će se potreba za upotrebom ovog paterna povećavati.

Bridge dizajn patern:

Bridge patern možemo iskoristiti u slučaju kada želimo da proširimo naš platni podsistem. Na primjer, ako želimo da podržimo novi način plaćanja, tada možemo da iskoristimo osobinu ovog paterna, a to je da se dio metoda koje se izvršavaju isto za svaki način plaćanja obavljaju uvijek na isti način, što bi omogućilo ponovnu upotrebu već postojećeg koda. Način na koji bi se primjenio ovaj patern je da se kreira klasa Bridge koja implementira interface za plaćanje.

Flyweight dizajn patern:

Ovaj patern se u našem sistemu može primjeniti za način specificiranja programa aktivnosti. Flyweight nam omogućava da optimiziramo utrošak memorije na

objekte klase *Korisnik* koji su u većini slučajeva isti te se jedino razlikuju po statusu programa aktivnosti. Način na koji bi se on mogao da ostvari je da se kreira *flyweight* klasa koja bi predstavljala različite programe aktivnosti (u tom slučaju bi se programi aktivnosti drugačije definisali), *context* klasu bi predstavljala strukturu korisnika, te *flyweight factory* klasu koja bi upravljala objektima.