Strukturalni paterni

1.Adapter pattern

Adapter pattern koristimo kada nam je potrebna šira upotreba već postojećih klasa, tj. kada nam je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu. Tim postupkom se dobija željena funkcionalnost bez izmjena na orginalnoj klasi i bez ugrožavanja integriteta cijele aplikacije.

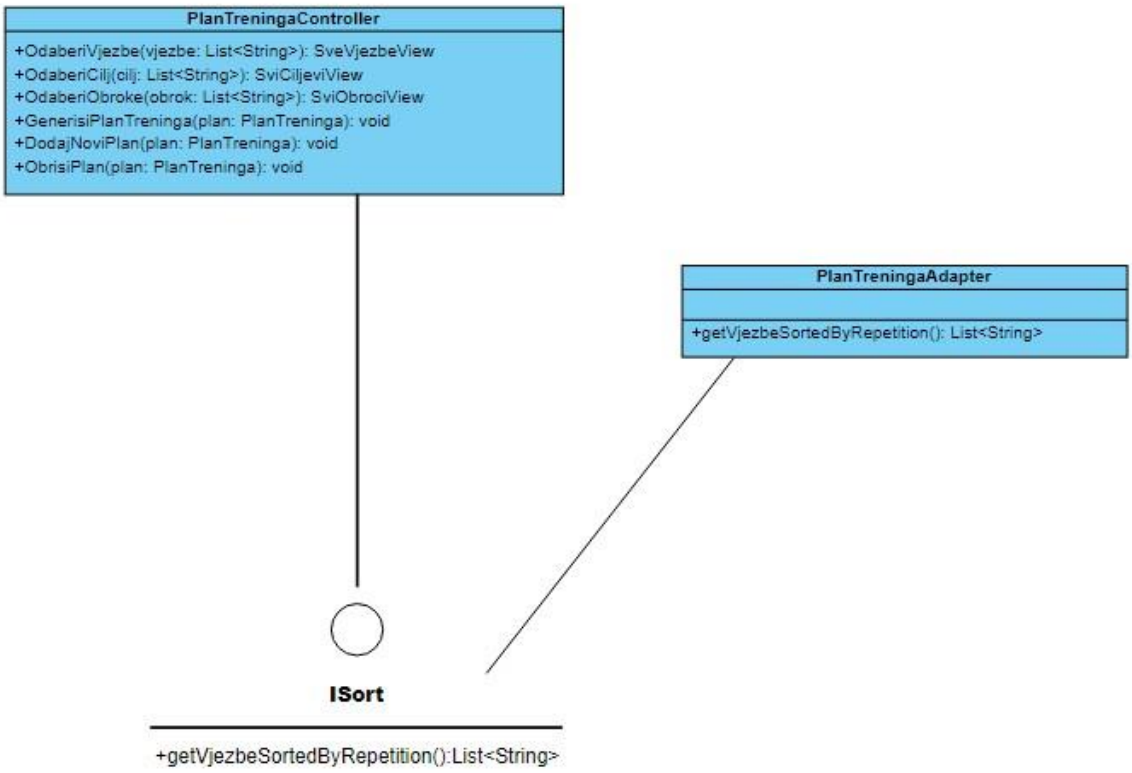
- Naš sistem možemo nadograditi tako da korisniku omogućimo sortiranje vježbi po broju ponavljanja serija u planu treninga, kako bi uvidio koji plan traje duže/kraće. To radimo na slijedeći način :

1. Definišemo interfejs ISort sa metodom getVjezbeSortedByRepetition();

2. Definišemo klasu PlanTreningaAdapter, koja implementira prethodno definisan interfejs ISort;

3. Pozivamo metodu getListaVjezbi() Adaptera klase PlanTreninga

Na ovaj način smo unaprijedili interfejs klase PlanTreninga, a da time nismo modificirali postojeću klasu.



2. Facade pattern

Facade pattern koristimo kada sistem ima više identificiranih podsistema pri čemu su apstrakcije i implementacije podsistema usko povezane. Njegova osnova namjena je da osigura više pogleda visokog nivoa na podsisteme.

- Naš sistem je jednostavan i nema usko povezanih podsistema. Ukoliko bi imali neke pomoćne interfejse (npr. za generisanje i prikazivanje plana vježbanja u posebnim formatima) tada bismo mogli iskoristiti ovu vrstu patterna kako bi se objednili ti interfejsi u neku FormatFacade klasu.

3. Decorator pattern

Decorator pattern koristimo da omogućimo dinamičko dodavanje novih elemenata i ponašanje postojećim objektima (dolazi do njihove nadogradnje). Dodatna ponašanja se dodjeljuju objektu tokom izvođenja programa bez mijenjanja koda koji je u interakciji sa datim objektom.

- Ova vrsta patterna bi se mogla iskoristiti u našem sistemu za klase PlanTreninga ili Grupe na način da se Administratoru omogući da uređuje izgled prikaza plana treninga i/ili grupa, nebitno da li je se prikazuju informacije ili slike. Potrebno bi bilo implementirati ISlika i/ili ITekst interfejs koji bi sadržavali metode za uređivanje odnosno dekorisanje, kao i klasu koja implementira taj interfejs.

4. Bridge pattern

Bridge pattern koristimo da omogućimo odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije. Bridge pattern pogodan je kada se implementira nova verzija softvera a postojeća mora ostati u funkciji.

- Da bi ovaj pattern mogli implementirati u našem sistemu, potrebno je dodati neku novu funkcionalnost. Kao primjer te funkcionalnosti može biti omogućavanje korisniku da u svoj plan treninga doda još neke vježbe za drugačiju težinu od one koju je prethodno generirao, dok bi ostale vježbe ostale iste. Tada bi klasu PlanTreninga proširili Bridge interfejsom sa metodom getListaVjezbi() koja bi vraćala vježbe u zavisnosti od podataka koje korisnik šalje.

5. Composite pattern

Composite pattern koristimo da omogućimo formiranje strukture stabla pomoću klasa, u kojoj se individualni objekti (listovi stabla) i kompozicije individualnih objekata (korijeni stabla) jednako tretiraju. Koristi se kada svi objekti imaju različite implementacije nekih metoda, ali im je potrebno svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija.

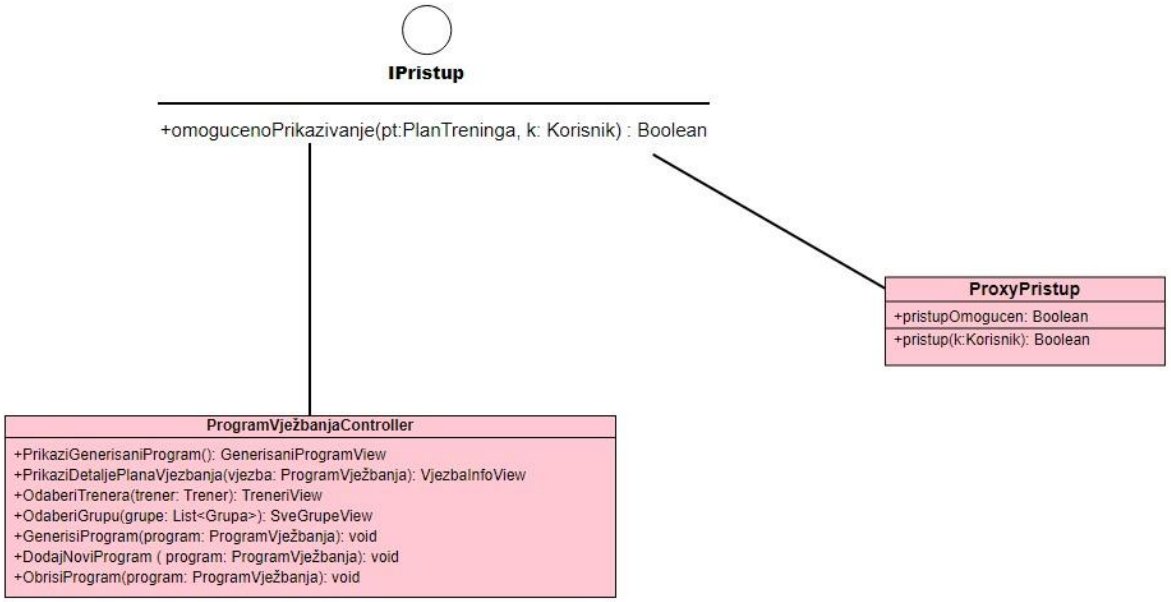
- Ovaj pattern bi u našem sistemu mogli iskoristiti na način da dva različita objekta prikažemo na isti način, a to su PlanTreninga i ProgramVježbanja. Oba se prikazuju u listi i moguće je otvoriti detalje svakog pojedinačnog objekta, ali se detalji i informacije razlikuju. Tada je potrebno napraviti interfejs IPrikazivanje koje definira interfejs-operacije za objekte u kompoziciji i implementira defaultno ponašanje koje je

zajedničko za objekte oba tipa, te Composite klasu nazvanu ListaPrikazivanja koja će sadržavati listu objekata tipa IPrikazivanje. IPrikazivanje implementiraju komponente PlanTreninga i PlanVježbanja kao i ListaPrikazivanja.

6. Proxy pattern

Proxy pattern koristimo da omogućimo pristup i kontrolu pristupa stvarnim objektima. Proxy je obično mali javni surogat objekat koji predstavlja kompleksni objekat čija aktivizacija se postiže na osnovu postavljenih pravila.

- U našem sistemu, ovaj pattern ćemo iskoristiti na način da ćemo postaviti kontrolu pristupa za ProgramVježbanja, koji se ne bi trebao moći otvoriti i prikazati ukoliko korisnik nema svoj profil. Da bi to postigli potrebno je dodati interfejs IPristup i klasu ProxyPristup koja sadrži metodu koja vrši provjeru pristupa.



7. Flyweight pattern

Flyweight pattern omogućava da smjestimo više objekata u raspoloživu količinu memorije dijeljenjem zajedničkih podataka između više objekata, umjesto da zadržimo sve podatke u svakom objektu. Pomoću njega postižemo racionalniju upotrebu resursa i brže izvršavanje programa. Ovaj pattern je potrebno koristiti kada sistem treba podržati kreiranje velikog broja sličnih objekata. Podaci unutar objekta se dijele na glavno i sporedno stanje, gdje glavno stanje predstavlja konstantne podatke koji su zajednički za sve objekte, a sporedno podatke koji nisu zajednički i češće se mijenjaju. Osnovna namjena Flyweight patterna je upravo da se omogući da više različitih objekata dijele isto glavno stanje, a imaju različito sporedno stanje.

- Ovaj pattern bi se u našem sistemu mogao iskoristiti za vježbe jer su to objekti kojih ima najviše, ali budući da svaka vježba ima svoje informacije karakteristične za nju, potrebno je proširiti sistem tako da pored tog sporednog stanja vježba imaju i glavno

stanje (npr. zajednička defaultna slika koja se javlja ukoliko administrator ne postavi odgovarajuću sliku vježbe).