DESIGN PATTERNS

STRUKTURALNI

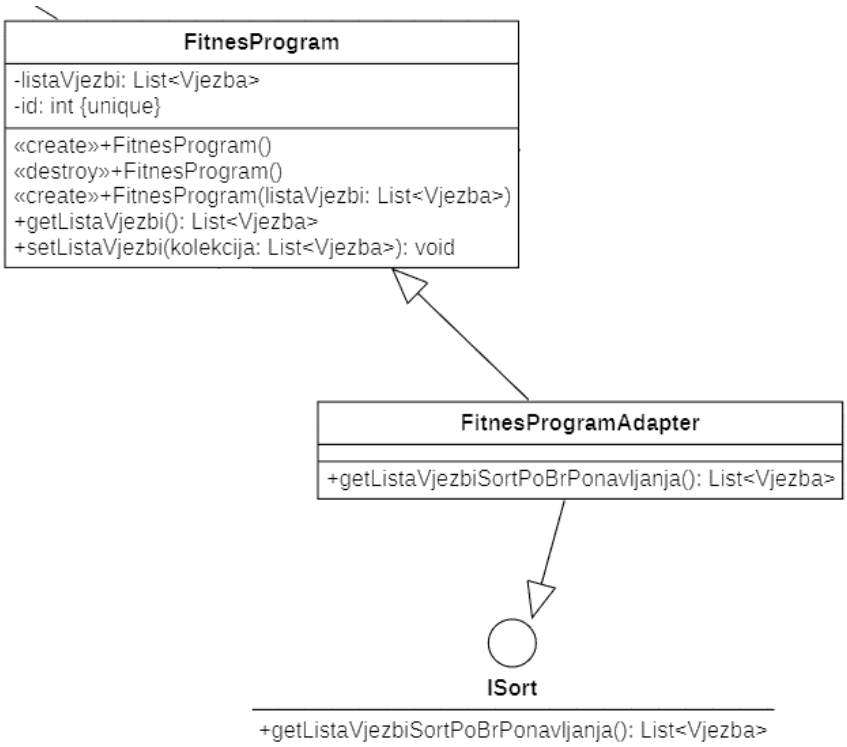
1. ADAPTER PATTERN

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 originalnoj klasi i bez ugrožavanja integriteta cijele aplikacije.

Naš sistem možemo nadograditi tako da korisniku omogućimo sortiranje vježbi po broju ponavljanja u fitnes programu kako bi lakše uvidio koje vježbe duže traju. To se može uraditi tako što primijenimo Adapter pattern na način da definišemo sljedeće:

- interfejs ISort sa metodom getListaVjezbiSortPoBrPonavljanja()
- klasu FitnesProgramAdapter, koja implementira interfejs ISort
- u metodi getPredmetiSortiranoPoCijeni() pozivamo metodu getListaVjezbi() Adaptee klase FitnesProgram

Na ovaj način smo unaprijedili interfejs klase FitnesProgram, a da time nismo modificirali postojeću klasu. Njen interfejs naknadno možemo unaprijediti i drugim metodama bez modificiranja postojeće klase ukoliko se ukaže potreba.



2. FACADE PATTERN

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

Naš sistem je poprilično jednostavan i nema usko povezanih podsistema. Kada bi imali neke pomoćne interfejse recimo za pravljenje i prikazivanje fitnes programa u različitim posebnim formatima, tada bismo mogli iskoristiti Facade pattern kako bi se objedinili ti interfejsi u neku FormatFacade klasu i samim tim lakše koristili, skrivajući implementacijske detalje.

3. DECORATOR PATTERN

Decorator pattern koristimo da omogućimo dinamičko dodavanje novih elemenata i ponašanja postojećim objektima, tj. vršimo njihovu nadogradnju. Dodatna ponašanja se dodjeljuju objektu tokom izvođenja programa bez mijenjanja koda koji je u interakciji sa datim objektom.

U našem sistemu bi ovaj pattern mogli primijeniti za klasu Vježba ili Recept, na način da se Administratoru omogući da uređuje izgled prikaza vježbi i recepata, bilo informacija bilo slika koje se prikazuju. Za to bi bilo potrebno implementirati interfejs ISlika ili ITekst koji bi sadržavali metode za uređivanje odnosno dekoriranje, 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.

U našem sistemu da bi ovaj pattern mogli implementirati potrebno je dodati neku novu funkcionalnost, npr. da se omogući korisniku da u svoj fitnes program doda još neke vježbe veće težine od onih koje je prethodno generirao ukoliko promijeni podatke o željenoj težini, dok bi ostale osobine vježbi ostale iste. Tada bi klasu FitnesProgram proširili Bridge interfejsom sa metodom getListaVjezbi() koja bi vraćala vježbe u zavisnosti od podataka koje korisnik šalje.

5. COMPOSITE PATTERN

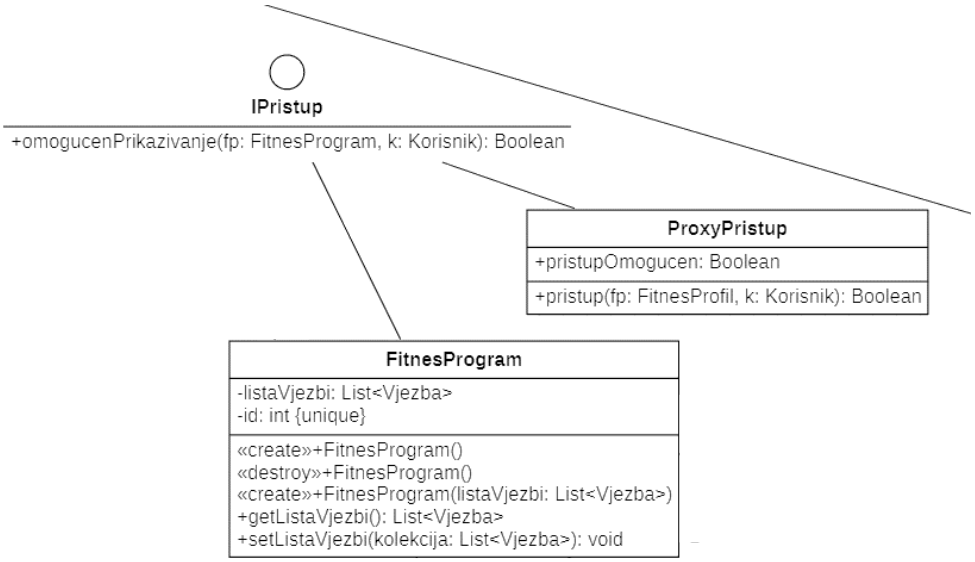
Composite pattern koristimo da omogućimo formiranje strukture stabla pomoću klasa, u kojoj se individualni objekti (listovi stabla) i kompozicije individualnih objekata (korijeni stabla) jednako tretiraju. Koristi se kada svi objekti imaju različite implementacije nekih metoda, ali im je potrebno svima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

Ovaj pattern možemo iskoristiti u našem sistemu tako što bi prikazivali dva različita objekta na isti način, a to su Recept i Vježba. 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 koji definira interfejs-operacije za objekte u kompoziciji i implementira defaultno ponašanje koje je zajedničko za objekte oba tipate, te Composite klasu nazvanu ListaPrikazivanja koja će sadržavati listu objekata tipa IPrikazivanje. IPrikazivanje implementiraju komponente Vježba i Recept 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.

Ovaj pattern ćemo primijeniti u našem sistemu tako što ćemo postaviti kontrolu pristupa za FitnesProgram, koji se ne bi trebao moći otvoriti i prikazati ukoliko korisnik nema popunjen svoj fitnes 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 podatake 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 paterna je upravo da se omogući da više različitih objekata dijele isto glavno stanje, a imaju različito sporedno stanje.

U našem sistemu bi se ovaj pattern mogao iskoristiti za vježbe jer su to objekti kojih ima najviše, ali budući da svaka vježba ima svoje podatke karakteristične za nju, potrebno je proširiti sistem tako da pored tog sporednog stanja vježbe imaju i glavno stanje, koje može biti npr. zajednička defaultna slika koja se javlja ukoliko administrator ne postavi odgovarajuću sliku vježbe.