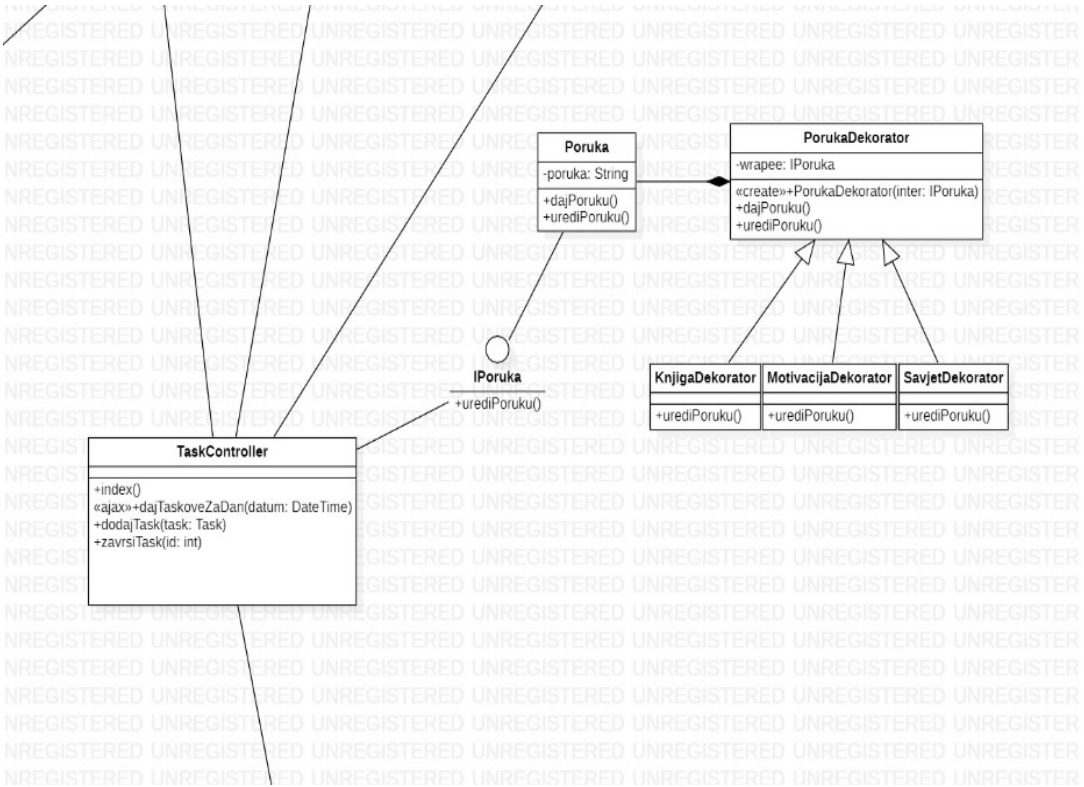
ADAPTER PATTERN

Adapter strukturalni pattern nam dozvoljava da u nasem sistemu iskoristimo neku klasu (servis) koja ima nekompatibilan interfejs sa nasim, vec postojecim.

Ovaj adapter je iskoristen u nasoj aplikaciji za generisanje PDF izvjestaja korisnika. Inicijalno, korisnik ima mogucnost da generise PDF za odredjenu kategoriju (taskovi, vjezbe…). Medjutim, nema mogucnost da generise PDF sa svim kategorijama odjednom. Za te potrebe kreiran je servis *GenerisiPDFFull* koji implentira nekompatibilan interfejs sa nasim *IgenerisanjePDFPoKategorijama.* Adapter klasa, kada zeli da pozove metodu iz *GenerisiPDFFull*, samo ce izostaviti parametar *kategorija*, cime smo uskladili nas interfejs sa interfejsom servisa.

DECORATOR PATTERN

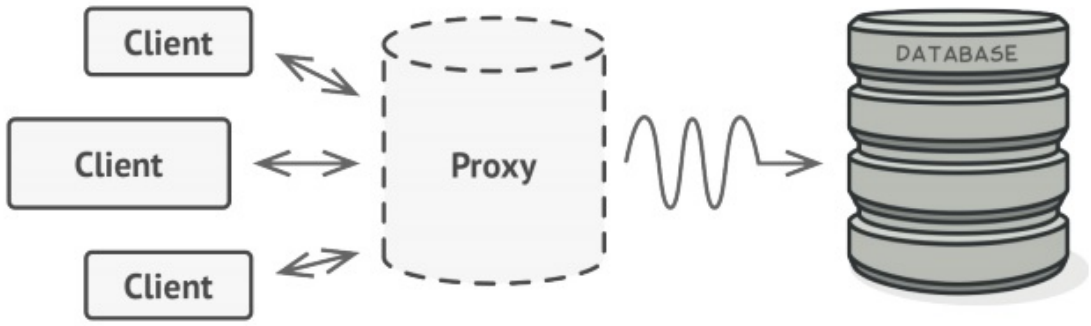
Decorator pattern nam omogućava da dodamo nova ponašanja na objekte, tako što postavimo objekte u određene omotače. Ovdje objekat ne zna da je izvršena dekoracija što je korisno za ponovnu upotrebu komponenti softverskog sistema.
Na koji način bi se ovaj pattern mogao iskoristiti u našoj LifePlanner aplikaciji?
Jedna od funkcionalnosti naše aplikacije jeste ta kada se korisnik prijavi u svoj račun da na početnoj stranici između ostalog bude prikazana i jedna motivacijska poruka, koja se mijenja svaki dan. Za potrebe izrade ovoga zadatka odlučili smo proširiti tu funkcionalnost na sljedeći način. Korisnik ima mogućnost odabira da li želi da mu se u tom polju prikaže samo motivacijska poruka ili želi da uz motivacijsku poruku se prikaže i neki Daily Tip ili neka preporučena knjiga za taj dan ili sve troje. Plan je da postoje baze podataka koje će sadržavati ove stringove i onda će se po potrebi random kombinirati po jedan string iz svake baze to jeste željenih baza. Dakle ta poruka koja je ustvari string objekat se može dodatno po želji korisnika ukrašavati to jeste nadograđivati. Pored samog ispisivanja poruke preko ovog patterna želimo omogućiti da se tako formirani string to jeste poruka i zapisuje u neki fajl koji će korisnik moći preuzeti i pregledati sve motivacijske poruke, savjete i knjige koji su mu bili preporučeni od početka korištenja aplikacije. Bitno je napomenuti da ovaj pattern za razliku od Adapter patterna neće mijenjati sam interfejs nego samo ponašanje objekta, to jeste stringa koji se isisuje i zapisuje.
Skica dijela dijagrama na koji se odnosi dati pattern se nalazi u nastavku kao i u posebnom fajlu.
IPoruka je zajednički interfejs između omotača i omotanog objekta. Poruka je objekat koji se omotava. kasnije može biti uređena od strane dekoratera. PorukaDekorator delegira sve operacije omotanom objektu. SavjetDekorator, KnjigaDekorator i MotivacijaDekorator definišu dodatna ponašanja. Oni overridaju metode osnovnog dekoratora.



PROXY PATTERN

Proxy je strukturalni dizajn pattern koji nam omogućava da postavimo zamjenu za neki drugi objekat. On kontrolira pristup orginalnom objektu i omogućava nam da obavimo određene akcije prije ili nakon što zahtjev prođe ka orginalnom objektu. To nam je na primjer potrebno kada imamo neki veliki objekat koji zahtjeva puno resursa sistema, a pristup tom objektu nam nije uvijek potreban. Proxy klasa se kreira tako da sadrži isti interfejs kao i usluđni objekat i zatim se ažurira aplikacija tako da preko proxy objekat se prosljeđuje klijentima originalnog objekta.
Na koji način bi se ovaj pattern mogao iskoristiti u našoj LifePlanner aplikaciji?
Pretpostavimo da Administrator želi saznati podatke o nekom korisniku (ime, prezime, e-mail) na osnovu njegovog Korisničkog imena. ovo su podaci koje korisnik naše aplikacije ne može mijenjati nakon što se sa njima registruje. Administrator u tom slučaju treba poslati upit ka bazi podataka da bi dobio te podatke. To međutim postaje neefikasno, ako Administrator više puta zatraži podatke za istog korisnika, zato što samo pristupanje bazi i traženje određenih podataka zna biti veoma sporo, a i baza u nekim situacijama može biti blokirana. Ovdje je rješenje sljedeće. Potrebno je napraviti proxy klasu koja imitira bazu, to jeste koja uz metode dohvaćanja iz baze također prati za koje je korisnike Administrator tražio podatke i vraća keširane rezultate u slučaju da za istog korisnika budu zatraženi podaci više puta. Time se izbjegava komunikacija klijenta sa bazom, a klijent uvijek dobiva podatke iz baze bez da je baza svjesna njegovog postojanja.

Prednost ovog načina rada jeste što možemo kontrolirati uslužni objekat bez da klijent zna za njega. Proxy radi kada objekat nije dostupan. Također ostaje očuvan Open/Closed princip s obzirom da možemo predstaviti nove Proxy-je bez da mijenjamo klijenta ili uslužni objekat.



Composite pattern

Za primjenu Composite patterna potrebno je dodati nove funkcionalnosti u našu aplikaciju. Prije svega, klase kao što su Vježba i Task bi trebale imati novi atribut koji u obliku broja označava određeni dio opterećenja. Obje klase bi trebale implementirati interface IOpterećenje koji bi imao metodu racunajOpterecenje() u zavisnosti od toga da li je u pitanju Vježba ili Task. U svrhu bolje ilustracije primjene ovog patterna, potrebne su dodatne kategorije taskova i treninga. Composite pattern može se primijeniti u slučaju da želimo da aplikacija na kraju dana računa koliko smo bili optrećeni taj dan time što bi, prolazeći kroz sve vrste taskova i treninga, aplikacija akumulirala vrijednosti opterećenja svakog taska i svake vježbe i u zavisnosti od vrijednosti totale prikazivala određenu animaciju, sliku, boju i slično. Kreće se od današnjeg dana, koji se grana na taskove i treninge. Taskovi i treninzi se dijele u kategorije, i svaka kategorija sadrži određen broj taskova/treninga. Task ne sadrži podklasu jer predstavlja list tj. osnovni element, dok trening sadrži vježbe koje su također osnovni elementi. Navedeni postupak bi se obavljao putem jednog zajedničkog interfejsa koji bi imao metodu npr. dajUkupnoOpterećenje() koja bi prolazila kroz listu List<Iopterećenje> koja predstavlja djecu trenutne klase, te prolazeći kroz listu bi računala ukupno opterećenje. Ovim postupkom rekurzivno prolazimo kroz čitavu hijerarhiju dok ne dođemo do osnovnih elemenata i tek tada zapravo vršimo račun.

Facade pattern

Facade pattern bi se mogao primijeniti ukoliko dodamo mogućnost davanja donacija razvojnom timu. Ovaj pattern se koristi kada želimo iskoristiti samo mali dio jednog velikog, kompleksnog sistema. Patter također sakriva kompleksnosti sistema i pruža jednostavan interfejs klijentu. Da bi omogućili donacije, koje podrazumijevaju uplaćivanje određene sume na bankovni račun primaoca, koristit ćemo PayPal API koji, između ostalog, sadrži PayPal Checkout koji olakšava internet plaćanje, što je također jedan primjer facade patterna jer nam je potreban samo jedan dio PayPal API-ja, koji sadrži vlastiti interfejs koji sakriva detalje plaćanja od korisnika. Pored ovoga, potrebno je implementirati DonationView zajedno sa kontrolerom DonationController. Potrebno je također dodati u navigacijski pogled opciju 'Donate'. Klikom na opciju, otvara se DonationView koji sadrži polje u kojem unosi željenu količinu novca u dolarima, i PayPal dugme preko kojeg se inicira proces plaćanja. PayPal dugme treba da reaguje na korisnikov klik, nakon čega poziva određene metode koje pruža PayPal API, i time započinje tzv. checkout tok.

Bridge pattern

Ovaj pattern omogućava razdvajanje apstrakcije i implementacije na način da klasa može posjedovati vise raličitih implementacija za određenu apstrakciju

Primjena ovog patterna u nasem slucaju je sledeca – ukoliko imamo klasu Jelo sa metodom izracunajKalorije() te ta metoda poziva istoimenu metodu za svaki sastojak (pri cemu postoji vise klasa Sastojaka a ne samo jedan npr. SastojakSlano,SastojakSlatko,.. etc) te na kraju se izvrsi neka operacija kao npr. Mnozenje/dijeljenje ukupnog broja kalorija sa 1000.

Flyweight pattern

Flyweight pattern (Cache) je pattern koji sluzi da se po mogucnosti smanji potrosnja RAM-a kada istog nedostaje. A nedostaje ga zato sto se u kratkom vremenskom periodu stvara veliki broj instance jedne klase, od kojih svaka instanca ima neki atribut koji dijeli svaka druga instanca. Stoga se taj atribut izdvoji te se spasi samo na jednom mjestu, a zatim sve instance umjesto da ga posjeduju, samo pokazuju na njega (pokazivac/referenca zauzima dosta manje memorije nego instanca klase).

Ovaj pattern, iako veoma zanimljiv, nismo uspjeli implementovati u nasoj aplikaciji. Ne postoji proces u nasem sistemu takav da ce se stvarati ogroman broj klasa iste ili slicne klase u kratkom vremenskom periodu. Jedan od mogucih scenarija bi bio kada bi imali neku animaciju na prikazu sistema. Neka ta animacija bude: kada korisnik ocijeni dan kao odlican, da stotine malih smajlija iskoci na ekran i neka se krecu po citavom ekranu par sekundi. Sliku smajlija je dovoljno cuvati na jednom mjestu, dok se **samo** koordinate cuvaju u svakoj od pojedinacnih instanci smajlija.