Strategy pattern

Strategy pattern je pattern ponasanja koji pruza mogucnost definisanja porodice algoritama, svaki u zasebnu klasu, te da se po potrebi mijenjaju tokom izvrsavanja programa pomocu kompozicije (za razliku od template method patterna koji to radi generalizacijom prilikom kompajliranja). Neke od prednosti ovog patterna su mijenjanje algoritama tokom izvrsavanja programa, izolacija implementacije od upotrebe, kompozicija naspram nasljedjivanja i pridrzavanje OCP-a. Neke od mana su prekomplikacija kada postoji samo mali broj veoma cesto koristenih algoritama, klijent/korisnik koda mora biti upoznat sa razlikama raznih “strategija” i veliki broj programskih jezika danas ima podrsku za funkcionalan tip pisanja koda te je ovaj pattern cesto predstavlja nepotreban trosak.

Ovaj algoritam se vrlo lahko moze implementovati u nas projekat. Primjer navodim za treninge, a vazi i za jela. Mozemo uvesti mogucnost korisniku da treninge sortira po nekom kriteriju: broj vjezbi, tezina, dan roka… Takodjer se isto moze uraditi i za vjezbe, sortirati po tezini, po trajanju… (analogija za recepte, jela i sastojke).

State pattern

State pattern je pattern ponasanja koji omogucava da objekt mijenja ponasanje kada se promijeni njegovo interno stanje. Time dobijamo dojam da objekat “mijenja” svoju klasu. Prednosti ovog patterna su slijedjenje SRP i OCP principa, kao i simplifikacija koda u slucaju velikog broja kondicionala. Mana ovog patterna je ista kao i za sve patterne, a to je da ako imamo mali broj stanja moze dovesti do overengineering-a.

Ovaj pattern u nasem projektu ima smisla upotrijebiti u klasama za jela (slican princip i za treninge). Neka korisnik ima razne recepte na raspolaganju. Uzmimo primjer da je za dorucak konzumirao nezdravu hranu. Zbog te odluke, za rucak nece moci pretraziti sve recepte, vec samo zdrave varijante. Vazi i obrnuto, ako mu je dorucak zdrav, postedicemo ga za rucak ako zeli. Jos jedan primjer je da ne dopustimo da korosnik unese bilo kakvu veceru poslije npr. 10 sati radi nesanice.

Template Method

Template Method pattern nam omogućava da izdvojimo određene korake algoritma u podklase. Dobra strana ovog patterna je ta što se struktura algoritma ne mijenja nego se mali dijelovi izdvajaju i imaju mogućnost da se implementiraju različito.

Na koji način bi se ovaj pattern mogao iskoristiti u našoj LifePlanner aplikaciji?

Predpostavimo da Registrovani korisnik može imati više tipova, na primjer obični korisnik i Premium korisnik. U klasi Registrovani korisnik neke funkcionalnosti su iste bez obzira na tip korisnika, a implementacija nekih bi zavisila od tipa korisnika. Predstavimo ovo jednim slikovitim primjerom. Mogućnosti uređivanja i brisanja profila su iste za obje vrste korisnika i one se mogu implementirati odmah u apstraktnoj klasi. Međutim funkcionalnosti koje omogućavaju korisniku da doda broj obaveza po danu će se implementirati različito od vrste korisnika, pa bi tako obični

korisnik mogao u svoju listu taskova dodati do 5 zadataka dnevno, dok bi kod Premium korisnika taj broj bio neograničen. Također obični korisnik bi mogao u prikazu za DailyTip dobiti samo prikaz savjeta, dok bi Premium korisniku bio omogućen i prikaz motivacione poruke i preporučene knjige.

Prednost ovog načina rada je što se smanjuje potreba za dupliranjem koda i promjene na jednoj klasi se neće odražavati na cijeli algoritam. Međutim jedan od nedostataka je što postoji mogućnost da Liskov princip bude narušen.

Observer Pattern

Observer pattern nam omogućava da implementiramo mehanizam subskripcije koji obavještave sve objekte koji prate neki drugi objekat o promjenama na tom objektu.

Na koji način bi se ovaj pattern mogao iskoristiti u našoj LifePlanner aplikaciji?

Pretpostavimo da će ova implementacija ovog projekta biti uspješna i da će developeri ovog projekta dobiti prilično primamljivu ponudu (moglo bi se i reći ponudu koju neće moći odbiti) od kompanije koja na svojoj Web Stranici svakog mjeseca objavljuje knjigu koja je bestseller za taj mjesec. Budući da su bitni ljudi iz te kompanije saznali za mogućnost u našoj aplikaciji koja korisnicima, ako žele preporučuje neku knjigu za čitanje, došli su na ideju da tu našu funkcionalnost još malo prošire. Oni žele da omoguće korisnicima naše aplikacije da se pretplate da mailom dobijaju obavijest o najpopularnijim knjigama. Dakle u našoj aplikaciji bi trebalo dodati klasu NajpopularnijeKnjige koja bi čuvala najpopularnije knjige. Također bi bilo potrebno da se preko API ključa povežemo sa stranicom naših novih saradnika i njihovom bazom podataka. Kada bi korisniku došla obavijest na mail o najpopularnijoj knjizi za taj mjesec, korisnik bi mogao na aplikaciji odabrati opciju prikaži knjigu (slično kao za DailyTip).

Subjekt je u ovom slučaju kompanija koja izdaje knjige, Iobserver je obavijest preko maila, Observer je RegistrovaniKorisnik, Update je prijem obavijeti, Notify je email, State je prikaz knjige na našoj aplikaciji.

Iterator pattern

U sistemu imamo liste treninga, raspoloženja, jela itd. i sve vrste elemenata navedenih listi su kompleksni objekti tj. nisu ugrađeni tipovi u C#. Napravit ćemo jednu vrstu iteratora koji će imati interfejs koji će definisati metode **getNext**(), **hasMore**(), **count**() (broj elemenata u listi), **current**() (vraća trenutni objekat na kojem smo pozicionirani), **rewind**() (vraća iterator na prvu poziciju) i svaki kontroler koji ima neku interakciju sa navedenim listama će po potrebi koristiti taj iterator. Također ćemo napraviti ConcreteIterator koji će u sebi sadržavati kolekciju koja će se inicijalizirati tokom kreiranja konkretnog iteratora. TreningController će korititi iterator i ubaciti listu treninga, JeloController će inicijalizirati iterator i ubaciti u njega listu jela itd. Ako želimo iterator za treninge, napišemo npr. Iterator iterator = new Iterator(listaTreninga). Treba naglasiti da opisani iterator ima najviše smisla kada sve klase koje mogu biti elementi liste implementiraju isti interfejs tj. da trening, jelo, raspoloženje itd. imaju isti interfejs. Pošto ovdje to nije slučaj, umjesto nekog konkretnog interfejsa stavit ćemo da listu čine elementi tipa Object.

Mediator pattern

Mediator pattern ćemo implementirati na način da ubacimo nove vrste korisnika (u svrhu ilustracije). Korisnici će se sada dijeliti na: NeregistrovaniKorisnik, RegistrovaniKorisnik, PremiumKorisnik, SupporterKorisnik (premium plaća sve usluge dok supporter plaća dio usluga). Sve navedene vrste korisnika imaju različitu implementaciju. Dodat ćemo i funkcionalnost **chat rooms** gdje jedan od premium korisnika ili administratora može kreirati chat room u koji se svaka od vrsta korisnika mogu pridružiti i komunicirati. Sve poruke se šalju ChatRoom klasi koja će imati metodu showMessage, koja će primati pošiljaoca i poruku koju šalje. Administratori su zaduženi za brisanje chat soba. Chat room je u ovom primjeru Mediator klasa koja će imati svrhu komunikacijskog posrednika između dva ili više korisnika. Na ovaj način svaki do korisnika može slati svoje poruke I vidjeti poruke drugih korisnika , a uz to ne mora biti upoznat sa implementacijom drugih korisnika.