DESIGN PATTERNS

PONAŠANJA

1. STRATEGY PATTERN

Strategy pattern izdvaja algoritam iz matične klase i uključuje ga u posebne klase, te kako bi se omogućila brza i jednostavna promjena implementacije koja se želi koristiti u bilo kojem trenutku. Na ovaj način omogućava se i jednostavno brisanje ili dodavanje novog algoritma koji se može koristiti po želji. Podržava open-closed princip.

Ovaj pattern možemo lako iskoristiti u našem sistemu pri filtriranju vježbi. Potrebno je uvesti klasu Filter koja bi bila Context klasa te napraviti interfejs IFiltriranje iz kojeg bi naslijedili različite vrste filtriranja, npr. za vježbe SerijaFilter, PonavljanjeFilter, LevelFilter

2. STATE PATTERN

State pattern mijenja način ponašanja objekata na osnovu trenutnog stanja. On omogućava objektu da mijenja svoja stanja, od kojih zavisi njegovo ponašanje. Sa promjenom stanja objekat se počinje ponašati kao da je promijenio klasu. Stanja se ne mijenjaju po želji klijenta, već automatski, kada se za to steknu uslovi. State Pattern je dinamička verzija Strategy patterna. Postiže se promjenom podklase unutar hijerarhije klasa. Podržava open-closed princip.

U našem sistemu bi se ovaj pattern mogao iskoristiti na način da proširimo sistem mogućnostima da korisnik dobije recepte odgovarajućeg obroka u zavisnosti od doba dana automatski. U tom slučaju bi nam trebao interfejs IState koji određuje koji recepti će biti prikazani i njega bi dodali u ReceptController. Trebala bi nam i nova klasa koja bi implementirala taj interfejs i sadržavala metode koje prikazuju odgovarajuće recepte.

3. TEMPLATE METHOD PATTERN

Template Method pattern omogućava algoritmima da izdvoje pojedine korake u podklase. Omogućavaju se izmjene ponašanja u jednom ili više dijelova. Najčešće se primjenjuje kada se za neki kompleksni algoritam uvijek trebaju izvršiti isti koraci, ali pojedinačne korake moguće je izvršiti na različite načine. Struktura algoritma se ne mijenja a mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

U našem sistemu bi se ovaj pattern mogao iskoristiti ukoliko bi u budućnosti odlučili dodati pored običnih korisnika i premium korisnike. U tom slučaju bi većina funkcionalnosti bile iste za obe vrste korisnika dok bi implementacija nekih određenih funkcionalnosti ovisila od vrste korisnika. Takve funkcionalnosti bi mogle biti recimo da obični korisnici imaju pristup osovnom skupu vježbi, a

premium korisnici bi imali pristup kako osnovnom skupu tako i skupu u kojem se nalazi nekoliko premium vježbi. Na ovaj način se smanjuje potreba za dupliciranjem koda.

4. Observer pattern

Observer pattern uspostavlja relaciju između objekata tako da kada jedan objekat promijeni stanje, drugi zainteresirani objekti se obavještavaju.

U našem sistemu bi se mogao iskoristiti u slučaju da želimo, na primjer, da se korisnik može prijaviti da dobija obavijesti putem mail-a kada se doda nova vježba koja odgovara njegovim informacijama i zahtjevima iz profila. Tada je potrebno dodati IObserver interfejs kojeg bi implementirala Observer klasa, u našem slučaju to je Korisnik, a klasa Vježba bi mijenjala stanje i obavještavala Observer klasu.

5. Iterator pattern

Iterator pattern omogućava sekvencijalni pristup elementima kolekcije bez poznavanja kako je kolekcija strukturirana.

Ovaj pattern će nam pomoći u određivanju načina na koji će se korisnik kretati kroz playlistu pjesama, budući da to do sada nije bilo određeno. Kada korisnik odabere playlistu, pjesme će biti poredane leksikografski u odnosu na naziv pjesme. On ima mogućnost odabrati shuffle opciju za randomiziran poredak ili loop opciju za poredak u kojem nakon zadnje pjesme dolazi ponovno prva. Za to je potrebno kreirati interfejs IIterator koji bi sadržavao metodu koja daje sljedeću pjesmu, interfejs IKolekcija koji deklariše metode za pravljenje iteratora kompaktibilnim za kolekciju, klase konkretnih iteratora koje implementiraju algoritme kretanja kroz kolekciju, te klasu konkretne kolekcije, u našem slučaju Player, koja vraća novu instancu nekog konkretnog iteratora svaki put kada ga korisnik zatraži.

6. Chain of Responsibility pattern

Chain of responsibility pattern namijenjen je kako bi se jedan kompleksni proces obrade razdvojio na način da više objekata na različite načine procesiraju primljene podatke.

Ovaj pattern bi se mogao iskoristiti u našem sistemu ukoliko bi dodatno omogućili korisniku da prilikom popunjavanja fitnes profila u svrhu generisanja fitnes program može odabrati one podatke i zahtjeve koji su mu prioritetni da vježbe u generisanom programu sadrže. Tada bi kreirali interfejs IHandler sa metodom koja prosljeđuje zahtjeve za obradu po redu prioriteta određenih od strane korisnika. Potrebno bi bilo dodati nove klase kao handlere koje bi obrađivale zahtjeve filtriranja i svaka bi vršila onaj dio obrade za koji je zadužena. Interfejs bi nasljeđivala klasa ProgramFilter a nju bi nasljeđivale klase handleri i ta bi klasa sadržavala listu programa i trenutni program koji se obrađuje.

7. MEDIATOR PATTERN

Mediator pattern namijenjen je za smanjenje broja veza između objekata. Umjesto direktnog međusobnog povezivanja velikog broja objekata, objekti se povezuju sa međuobjektom medijatorom, koji je zadužen za njihovu komunikaciju. Kada neki objekt želi poslati poruku drugom objektu, on šalje poruku medijatoru, a medijator prosljeđuje tu poruku namijenjenom objektu ukoliko je isto dozvoljeno.

Budući da u našem sistemu nema kompleksnijih veza ovaj pattern trenutno nije moguće iskoristiti, ali ukoliko bismo odlučili nadograditi sistem na način da je korisnicima omogućeno da ostavljaju ocjene ili komentare na vježbe ili recepte koje bi ostali korisnici kao i administratori mogli vidjeti, tu bi ovaj pattern pronašao svoju ulogu. Tada bi se komentari slali klasi Komentar koja bi igrala ulogu Mediator klase, te bi se logika razmjenjivanja komentara izdvojila u poseban interfejs kojeg bi kao atribut imale klase koje ostavljaju komentare, u našem slučaju Korisnik.