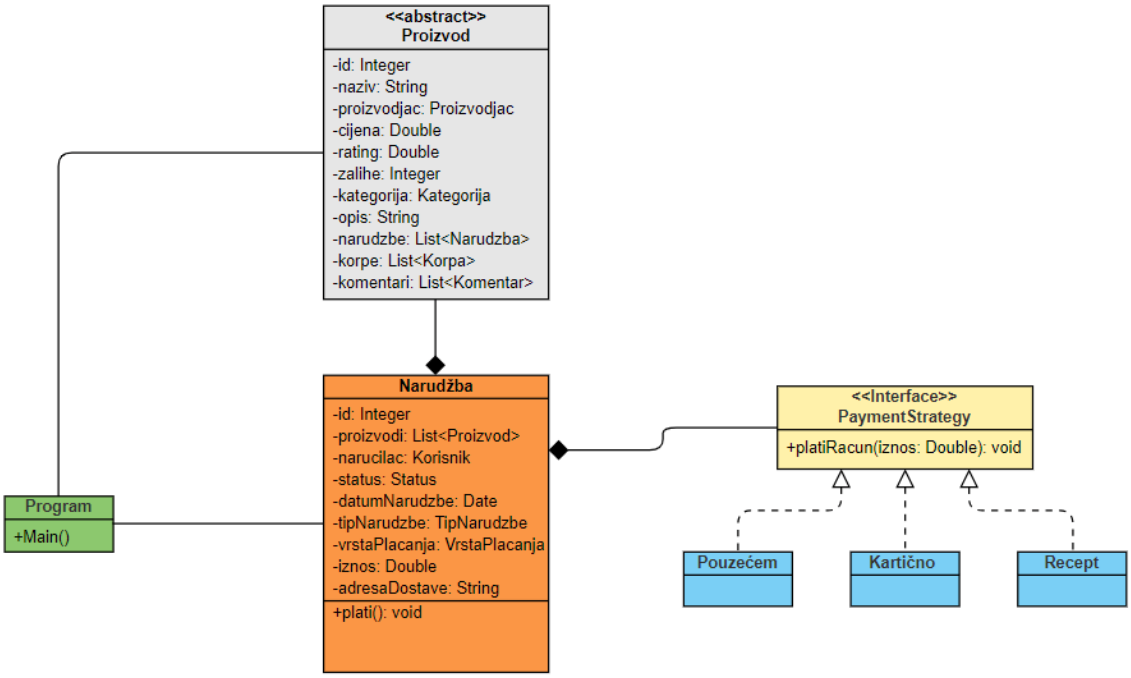
Paterni ponašanja

1. Strategy pattern

Potreba za strategy patternom javlja se kada želimo imati različite načine postizanja istog cilja. U našem eApoteka sistemu, imamo različite načine plaćanja, kao što su kartica, gotovina ili online plaćanje. Svaki od tih načina plaćanja ima svoje prednosti i može biti prikladan u određenim situacijama. Korištenjem strategijskog obrasca možemo fleksibilno odabrati strategiju plaćanja koja nam najbolje odgovara u određenom trenutku. Na taj način, ne moramo mijenjati sam sistem svaki put kada želimo promijeniti način plaćanja. Ovaj obrazac nam omogućava da lakše upravljamo različitim strategijama i olakšava nam održavanje i proširivanje našeg eApoteka sistema u budućnosti.



2. State pattern

U našem sistemu potreba za state patternom javlja se kada objekt može imati različita stanja i ponašanja u različitim situacijama. U našem eApoteka sistemu, možemo razmotriti primjer korisnika koji može biti u različitim stanjima, poput "aktivan", "neaktivan" ili "blokiran". Svako od tih stanja zahtijeva specifično ponašanje i ograničenja za korisnika. Korištenjem state patterna, možemo organizirati i upravljati tim stanjima na efikasan način.

Na primjer, kada je korisnik u stanju "neaktivan", ne može pristupiti određenim funkcionalnostima sistema, poput pravljenja narudžbe ili dodavanja komentara. Kada se stanje korisnika promijeni u "blokiran", onemogućuje se pristup svim funkcionalnostima sistema.

State patterni omogućavaju nam da lako upravljamo različitim stanjima korisnika, olakšavaju nam dodavanje novih stanja u budućnosti i čine kod modularnim i lakim za održavanje. Također nam omogućavaju da se fokusiramo na specifična ponašanja i ograničenja u određenim stanjima, što rezultira čistim i čitljivim kodom.

3. Template method

Template method pattern javlja se kada imamo slične operacije koje imaju zajednički okvir ili korake, ali se specifični koraci mogu razlikovati. U našem sistemu, možemo razmotriti primjer procesa narudžbe. Bez obzira na vrstu narudžbe (dostava ili lično preuzimanje), postoji zajednički okvir ili koraci koji se moraju slijediti, kao što su provjera dostupnosti proizvoda, izračun ukupne cijene, ažuriranje stanja inventara i slanje potvrde korisniku.

Korištenjem obrasca šablonske metode, možemo definirati apstraktnu klasu "Narudžba" koja sadrži osnovni okvir za proces narudžbe. Svaka konkretna klasa narudžbe, poput "DostavaNarudžba" ili "LičnoPreuzimanjeNarudžba", nasljeđuje apstraktnu klasu i implementira specifične korake koji se razlikuju za tu vrstu narudžbe. Na primjer, koraci za dostavu narudžbe mogu uključivati izračun troškova dostave i odabir adrese dostave, dok koraci za lično preuzimanje mogu uključivati odabir lokacije preuzimanja i provjeru raspoloživosti.

Template method pattern omogućava nam da izbjegnemo dupliranje koda, promoviše ponovnu iskoristivost i olakšava održavanje sistema. Omogućavaju nam da se fokusiramo na specifične korake u svakoj konkretnoj klasi, dok zajednički okvir ostaje nepromijenjen. Ovaj pattern također olakšava dodavanje novih vrsta narudžbi u budućnosti, jer je jednostavno naslijediti apstraktnu klasu i implementirati specifične korake.

4. Observer

Potreba za observer patternom javlja se kada želimo uspostaviti odnos između objekata "izdavača" (subject) i "pretplatnika" (observer) na način da se automatski obavještavaju o promjenama stanja izdavača. U našem eApoteka sistemu, možemo razmotriti primjer promjene stanja proizvoda, poput raspoloživosti ili cijene. Kada se stanje proizvoda promijeni, želimo da se automatski obavijeste pretplatnici, kao što su korisnici ili administator sistema.

Korištenjem obrasca promatrača, možemo definirati interface "Izdavač" (Subject) koje sadrži metode za registraciju, odjavu i obavještavanje pretplatnika. Svaki pretplatnik implementira sučelje "Pretplatnik" (Observer) koje definira metodu za primanje ažuriranja od izdavača. Kada se promijeni stanje proizvoda, izdavač obavještava sve pretplatnike putem metode za ažuriranje.

Ovaj pattern omogućava fleksibilnost i rast sistema, jer pretplatnici mogu dinamički registrirati ili odjaviti se od izdavača i primati samo relevantna ažuriranja. Na primjer, korisnik koji je pretplaćen samo na određene proizvode će biti obaviješten samo o promjenama tih proizvoda, dok administrator sistema može biti obaviješten o svim promjenama u stanju proizvoda.

5. Iterator

Potreba za iterator patternom javlja se kada želimo efikasno i jednostavno prolaziti kroz kolekciju objekata, bez da otkrivamo unutarnju strukturu kolekcije. Zamislimo da imamo poseban dio u našoj eApoteka aplikaciji koji prikazuje sve proizvode iz kolekcije, ali želimo imati fleksibilnost da promijenimo način prolaska kroz te proizvode.

Koristeći iterator pattern, možemo definirati interfejs "Iterator" koje sadrži metode poput "sljedeći" (next) i "imaSljedeći" (hasNext), koje omogućavaju da iteriramo kroz proizvode jedan po jedan. Implementacija iteratora će ovisiti o unutarnjoj strukturi kolekcije, ali izvana, možemo koristiti isti sučelje bez obzira na to koja je struktura kolekcije iza.

Na primjer, u našoj aplikaciji možemo imati kolekciju proizvoda koja je implementirana kao niz ili kao povezana lista. Koristeći iterator pattern, možemo dobiti iterator za tu kolekciju i koristiti iste metode "sljedeći" i "imaSljedeći" bez obzira na to koja je struktura kolekcije korištena.

Ovaj obrazac olakšava prolazak kroz kolekcije objekata na jednostavan i konzistentan način. Omogućava nam da se fokusiramo samo na prolazak kroz objekte, a ne na detalje implementacije kolekcije. Također nam omogućava da promijenimo unutarnju strukturu kolekcije bez mijenjanja koda koji koristi iterator.

Korištenje iterator patterna olakšava rad s kolekcijama i povećava modularnost sustava. Omogućava nam da jednostavno dodajemo nove vrste kolekcija i mijenjamo način iteracije bez utjecaja na ostatak koda. Također nam omogućava da kod koji prolazi kroz kolekcije bude neovisan o konkretnoj implementaciji kolekcije, što olakšava održavanje i fleksibilnost sistema.

6. Chain of responsibility

Potreba za Chain of Responsibility patternom javlja se kada imamo niz objekata koji mogu obraditi određeni zahtjev, ali ne znamo unaprijed koji će objekt to učiniti. Zamislimo da imamo aplikaciju za obradu narudžbi u eApoteci, gdje svaka narudžba prolazi kroz niz koraka obrade, poput provjere dostupnosti, provjere cijene, provjere recepta i slanja narudžbe.

Koristeći obrazac Chain of Responsibility, svaki korak obrade predstavljen je posebnim objektom koji ima metodu "obradiZahtjev" (handleRequest). Ako korak može obraditi zahtjev, to čini i završava obradu. Ako ne može obraditi zahtjev, prosljeđuje zahtjev sljedećem koraku u lancu. Na taj način, svaki korak ima mogućnost da obradi ili preusmjeri zahtjev.

Na primjer, prvi korak u lancu može provjeriti dostupnost proizvoda. Ako je proizvod dostupan, taj korak obrađuje zahtjev i završava obradu. Ako proizvod nije dostupan, taj korak preusmjerava zahtjev sljedećem koraku, na primjer, koraku za provjeru cijene.

Ovaj pattern olakšava dodavanje i konfiguraciju novih koraka obrade u lancu. Možemo jednostavno dodati ili ukloniti korake bez da mijenjamo ostatak koda. Također nam omogućava da imamo fleksibilnost u konfiguriranju redoslijeda koraka ili u definiranju novih lanaca obrade za različite vrste narudžbi.

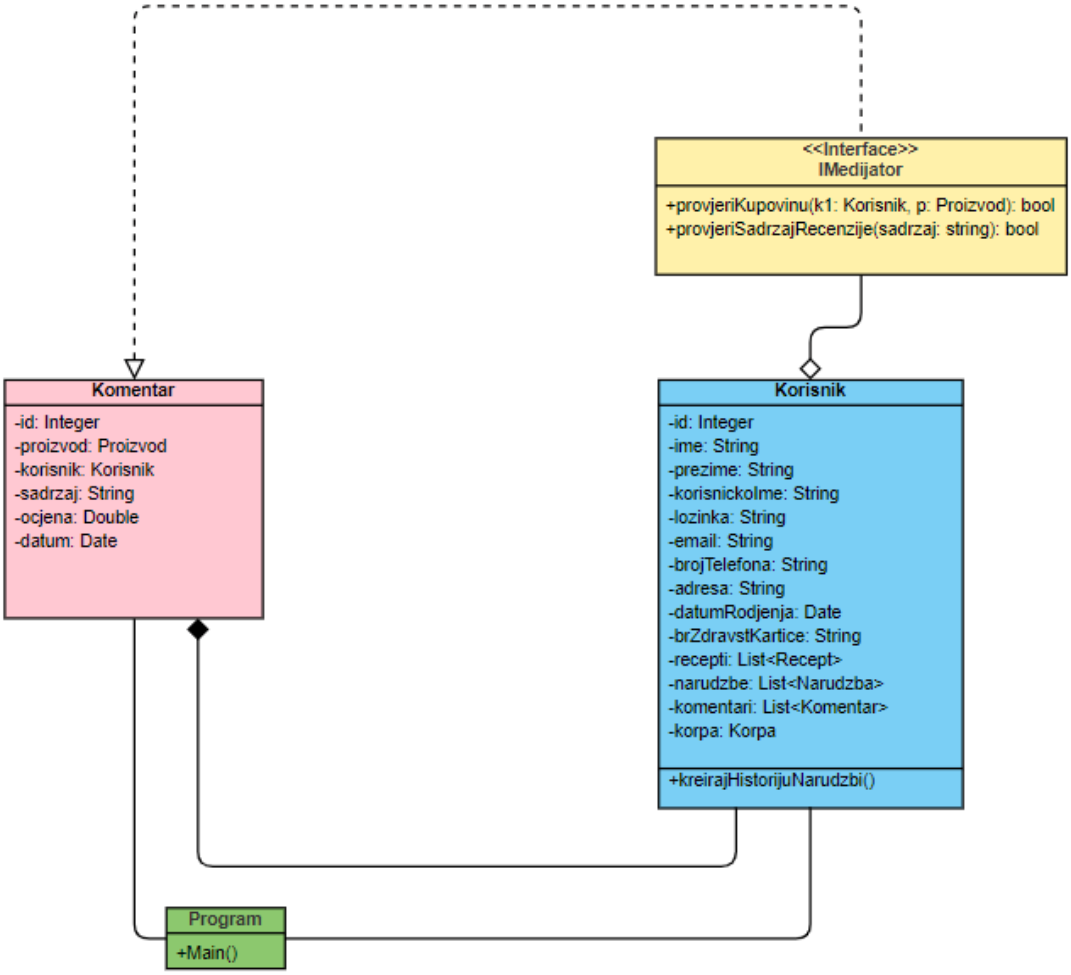
7. Mediator

Mediator pattern je koristan za naš sistem jer omogućava koordinaciju komunikacije između različitih dijelova sistema prilikom objavljivanja komentara. Kada korisnik pošalje komentar, Mediator klasa ima ključnu ulogu u provjeri da li je korisnik već kupio proizvod prije nego što se komentar objavi.

Mediator klasa komunicira s klasama Korisnik i Proizvod kako bi provjerila da li korisnik ima prethodnu kupovinu tog proizvoda. Ako korisnik ima prethodnu kupovinu, komentar se objavljuje, a ako nema, komentar se odbija ili preusmjerava na odgovarajuću stranicu.

Korištenje Mediator pattern pruža nekoliko prednosti. Prvo, odvaja složenu logiku komunikacije između objekata i centralizira je u Mediator klasu, čime se olakšava održavanje i proširivost sistema. Drugo, smanjuje ovisnosti između objekata jer objekti ne moraju direktno komunicirati, već koriste Mediator klasu kao posrednika. Ovo povećava fleksibilnost sistema i olakšava dodavanje novih provjera ili funkcionalnosti pri objavljivanju komentara.

U našem slučaju, korištenje Mediator pattern omogućava nam provjeru prethodne kupovine prije objavljivanja komentara, čime se osigurava da samo korisnici koji su već kupili proizvod mogu ostaviti komentar. To pomaže u pružanju relevantnih i pouzdanih informacija drugim korisnicima, poboljšava iskustvo korisnika i održava integritet komentara u sistemu eApoteka.


<<Interface>>
IMedijator
+provjeriKupovinu(k1: Korisnik, p: Proizvod): bool
+provjeriSadrzajRecenzije(sadrzaj: string): bool
Komentar
-id: Integer
-proizvod: Proizvod
-korisnik: Korisnik
-sadrzaj: String
-ocjena: Double
-datum: Date
Korisnik
-id: Integer
-ime: String
-prezime: String
-korisnickoIme: String
-lozinka: String
-email: String
-brojTelefona: String
-adresa: String
-datumRodjenja: Date
-brZdravstKartice: String
-recepti: List<Recept>
-narudzbe: List<Narudzba>
-komentari: List<Komentar>
-korpa: Korpa
+kreirajHistorijuNarudzbi()
Program
+Main()