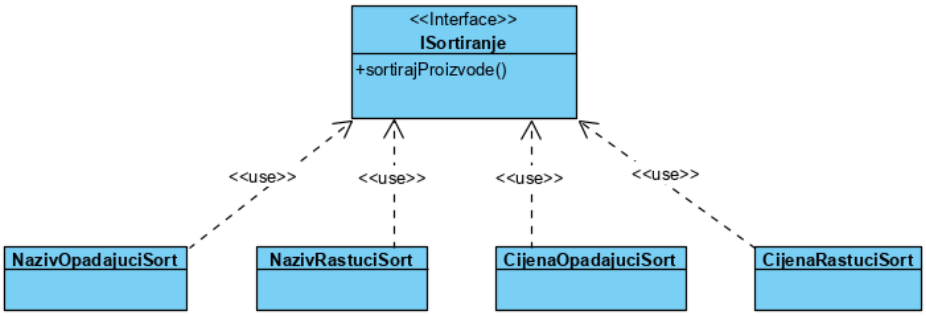
PATERNI PONAŠANJA

1. STRATEGY

Ovaj pattern se koristi ukoliko se jedan problem može riješiti sa više različitih algoritama. Najbolji primjer upotrebe ovog paterna možemo vidjeti kod algoritama sortiranja kojih postoji zaista mnogo, kao što su quick sort, merge sort, radix sort, selection sort itd. Ovaj patern smo iskoristili za soritanje proizvoda, pri čemu se sortiranje vrši na različite načine. Korisnik bira kategoriju po kojoj će se vršiti sortiranje. Dostupne kategorije su: abecedno, po cijeni itd. Pomoću ovog paterna neće biti teško dodati nove metode za sortiranje ukoliko se za njima ukaže potreba.
Potrebno je dodati interfejs (ISortiranje) koji će imati navedenu metodu sortirajProizvode te implementirati klase koje će izvesti taj interfejs. Promjene su prikazane na slici ispod:



Još je potrebno dodati metodu za sortiranje u klasi koja sadrži listu proizvoda. Osim te metode potrebno je dodati i atribut strategija tipa ISortiranje koji će biti indikator koji će se sort koristiti te će se za njega napraviti set metoda.

2. STATE

Sličan je Strategy paternu, tačnije predstavlja njegovu dinamičku verziju. On se koristi pri promjeni stanja objekta neke klase. Postiže se promjenom podklase unutar hijerarhije klasa. Za ovaj patern nismo pronašli primjenu. Međutim, ukoliko bi se odlučili na dodavanje neke funkcionalnosti kod koje bi bilo bitno npr. da li je korpa prazna ili ne, mogli bi popunjenost korpe posmatrati kao dva različita stanja te bi se ovaj patern mogao primijeniti.

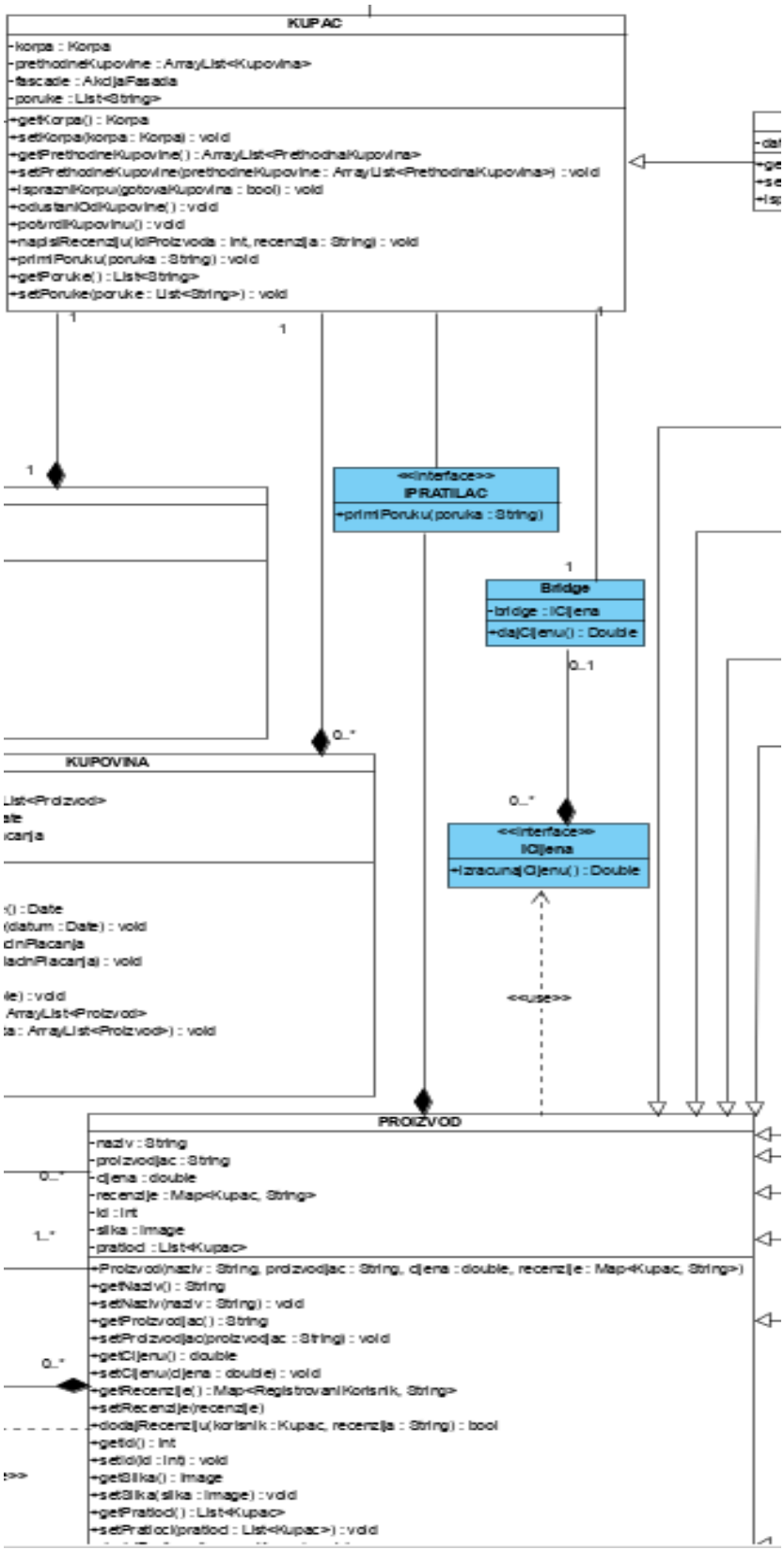
3. TEMPLATEMETHOD

Koristi se za izdvajanje koraka algoritma u podklase. Većinom se upotrebljava u sličnim situacijama kao i Strategy patern. Ovaj patern nismo iskoristili ali bi se npr. mogao iskoristiti prilikom plaćanja. Imamo klasu Kupac i klasu Student. Algoritam plaćanja im se razlikuje ali u suštini bi obavljao istu stvar. Mogli bismo taj algoritam izdvojiti u posebnu

klasu (mogao bi se koristiti interfejs), koja bi imala metodu za plaćanje. Klasa Kupac i klasa Student bi imale različite implementacije ove metode jer bi se kod klase Student vršio izračun popusta kojeg nemamo kod kupca.

4. OBSERVER

On se koristi da bi se na jednostavan način napravio mehanizam za pretplatu. Pretplatnici dobivaju obavještenja o sadržajima na koje su pretplaćeni, a za slanje obavještenja zadužena je nadležna klasa. Na ovaj način uspostavlja se relacija između klasa kako bi se mogle prilagoditi međusobnim promjenama.
Ovaj pattern smo iskoristili tako što smo dodali potrebnu funkcionalnost. Zamisao je da kupac može uključiti obavijesti za neki proizvod iz naše baze, te kad god se promijeni stanje tog proizvoda (misli se na promjenu dostupne količine ili trenutne cijene) pošalje se obavijest kupcu.
Kao prvo, potrebno je u klasi Proizvod dodati listu kupaca (kao atribut) koji prate posmatrani proizvod. Nazovimo taj atribut pratioci. U klasi Proizvod je potrebno dodati i metodu koja obavještava o promjeni stanja proizvoda, nazovimo je obavijestiKupca. U klasi Kupac je potrebno dodati metodu azuriraj koja će poslati poruku kupcu preko njegovog UI-a. Zbog ove funkcionalnosti smo odlučili dodati i atribut „poruke“ u klasu Kupac koji će čuvati sve pristigle poruke za kupca. Također je potrebno napraviti interfejs koji sadrži metodu azuriraj koja će se upravo i preklopiti u klasi Kupac, tj. klasa Kupac će implementirati ovaj interfejs. Novododane dijelove je potrebno dodati i u dijagram klasa:


KUPAC
-korpa : Korpa
-prethodneKupovine : ArrayList<Kupovina>
-fasciade : AkcijaFasada
-poruke : List<String>
+getKorpa() : Korpa
+setKorpa(korpa : Korpa) : void
+getPrethodneKupovine() : ArrayList<PrethodnaKupovina>
+setPrethodneKupovine(prethodneKupovine : ArrayList<PrethodnaKupovina>) : void
+IsprazniKorpu(gotovaKupovina : bool) : void
+odustaniOdKupovine() : void
+potvrdiKupovinu() : void
+napisiRecenziju(idProizvoda : Int, recenzija : String) : void
+primiPoruku(poruka : String) : void
+getPoruke() : List<String>
+setPoruke(poruke : List<String>) : void
<<interface>>
IPRATILAC
+primiPoruku(poruka : String)
Bridge
-bridge : ICijena
+dajCijenu() : Double
KUPOVINA
List<Proizvod>
ste
icanja
() : Date
(datum : Date) : void
cinPlacanja
lacniPlacanja() : void
le) : void
ArrayList<Proizvod>
a : ArrayList<Proizvod>) : void
<<interface>>
ICijena
+IzracunajCijenu() : Double
<<use>>
PROIZVOD
-naziv : String
-proizvodjac : String
-cijena : double
-recenzije : Map<Kupac, String>
-id : Int
-slika : Image
-pratioci : List<Kupac>
+Proizvod(naziv : String, proizvodjac : String, cijena : double, recenzije : Map<Kupac, String>)
+getNaziv() : String
+setNaziv(naziv : String) : void
+getProizvodjac() : String
+setProizvodjac(proizvodjac : String) : void
+getCijenu() : double
+setCijenu(cijena : double) : void
+getRecenzije() : Map<RegistrovaniKorisnik, String>
+setRecenzije(recenzije)
+dodajRecenziju(korisnik : Kupac, recenzija : String) : bool
+getId() : Int
+setId(id : Int) : void
+getSlika() : Image
+setSlika(slika : Image) : void
+getPratioci() : List<Kupac>
+setPratioci(pratioci : List<Kupac>) : void
1
0..*
1..*

5. ITERATOR

Koristi se za sekvencijalni pristup elementima kolekcije bez poznavanja njene strukture. Ovaj patern preporučljivo je iskoristiti kada se za iteriranje koristi kompleksna logika koja ovisi o više kriterija. Ovaj patern bismo mogli implementirati ukoliko bismo npr. imali funkcionalnost da se kupcu nude proizvodi po nekom specifičnom redoslijedu, npr. neki „shuffle“ poredak ili slično. Pošto tu funkcionalnost nemamo, za ove dosadašnje funkcionalnosti smatramo da nema prevelike potrebe za ovim paternom iz razloga što su nam liste većinom po nekom uobičajenom redoslijedu (abecedno, po cijeni, po vrsti proizvoda i sl.).