STRUKTURALNI PATERNI

U sljedećoj tabeli navedeni su svi strukturalni paterni, njihove namjene i odgovor da li smo ih iskoristili na našem dijagramu. U nastavku slijede i detaljna objašnjenja zašto jesmo/nismo primjenili određeni strukturalni patern.

PATERN	NAMJENA PATERNA	DA LI SMO GA KORISTILI?
ADAPTER PATERN	U situacijama kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu koristi se Adapter patern.	DA
FACADE PATERN	Osnovna namjena Facade paterna je da osigura više pogleda visokog nivoa na podsisteme (implementacija podsistema skrivena od korisnika).	NE
DECORATOR PATERN	Osnovna namjena Decorator paterna je da omogući dinamičko dodavanje novih elemenata i ponašanja (funkcionalnosti) postojećim objektima	NE
BRIDGE PATERN	Osnovna namjena Bridge paterna je da omogući odvajanje apstrakcije i implementacije neke klase tako da ta klasa može posjedovati više različitih apstrakcija i više različitih implementacija za pojedine apstrakcije. Bridge patern pogodan je kada se implementira nova verzija softvera a postojeća mora ostati u funkciji.	NE
PROXY PATERN	Namjena Proxy paterna je da omogući pristup i kontrolu pristupa stvarnim objektima	DA
COMPOSITE PATERN	Osnovna namjena Composite paterna (kompozitni patern) je da omogući formiranje strukture stabla pomoću klasa, u kojoj se individualni objekti (listovi stabla) i kompozicije individualnih objekata (korijeni stabla) jednako tretiraju	NE
FLYWEIGHT PATERN	Osnovna namjena Flyweight paterna je upravo da se omogući da više različitih objekata dijele isto glavno stanje, a imaju različito sporedno stanje	NE

<u>Adapter Patern:</u>

Ukoliko bismo nadogradili naš sistem, na način da omogućimo klijentu da uz pretragu proizvoda po ključnoj riječi, može da pretražuje proizvode čija se cijena kreće u nekom zadanom intervalu, iskoristili bismo Adapter patern.

Dakle, kada kupac odabere ključnu riječ za pretragu, nudi mu se mogućnost da unese interval cijene koja mu odgovara, pa će se uz pretraživanje po ključnoj riječi, ponuda proizvoda filtrirati i po cijeni, to jeste prilagoditi se zahtjevu kupca (ovo će se desiti samo onda kada se za tim javi potreba, tj. zahtjev kupca, ali inače ukoliko se ovaj zahtjev ne pojavi kupac će vidjeti sve proizvode određene tom ključnom riječi).

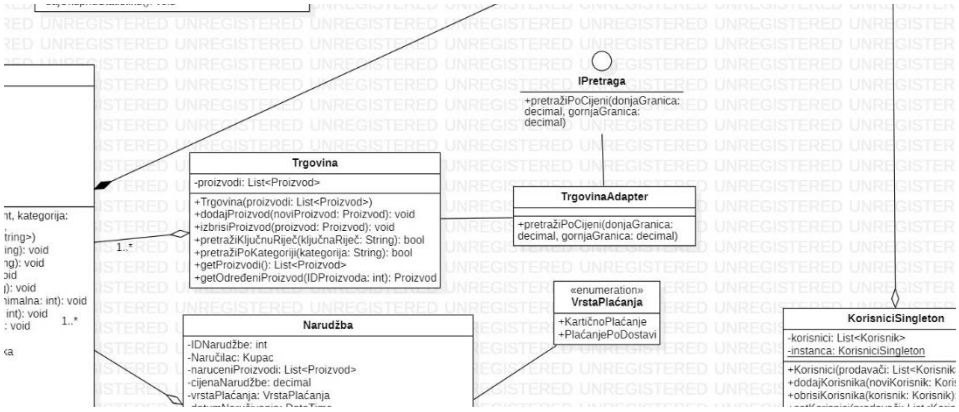
Koraci:
1. definisati interfejs IPretraga s metodom pretražiPoCijeni()

2. definisati klasu TrgovinaAdapter, koja implementira interfejs IPretraga, koja poziva metodu PretražiPoKljučnojriječi klase Trgovina, pri tome prilagođavajući ovu metodu da vraća proizvode filtrirane po cijeni.

Klasa klijent je zapravo ASP.NET aplikacija, to jeste, kontroleri koji će vršiti pozive ovih metoda iz Adapter paterna, za sada je nismo stavljali na dijagram, ali ona bi bila u vezi sa IPretraga.

Ova nadogradnja (korištenje Adapter paterna) bi omogućila da se kasnije sistem vrlo lahko nadogradi i za druge vrste pretraga, npr. ukoliko se javi potreba ida se proizvodi pretražuju i prema nekim recenzijama, materijalu(ukoliko je u pitanju odjeća i sl..), itd..
Na ovaj smo način naš sistem znatno unaprijedili korištenjem Adapter paterna, a ovaj patern ostavlja i mogućnost za nove nadogradnje koje bi ga unaprijedile.

Na slici je prikazano mjesto na našem dijagramu gdje smo upotrijebili Adapter patern:



Fasadni patern

Fasadni patern služi kako bi se klijentima pojednostavilo korištenje kompleksnih sistema. Klijenti vide samo fasadu, odnosno krajnji izgled objekta, dok je njegova unutrašnja struktura skrivena. Na ovaj način smanjuje se mogućnost pojavljivanja grešaka jer klijenti ne moraju dobro poznavati sistem kako bi ga mogli koristiti.

Na svom dijagramu nismo iskoristili ovaj patern, zato što nemamo naročitih dijelova koji su kompleksni, te ne vidimo potrebu za pravljenjem „fasade“ objekata.

Ali kada bismo se odlučili na korištenje ovog paterna, mogli bismo ga primjeniti kod funkcionalnosti naručivanja, jer klasa Narudžba ipak ima malo kompleksniju strukturu i u vezi je skoro sa svim ostalim klasama. Koraci:

1. Definisati novu klasu NapraviNarudžbuFasada koja implementira metode popuniNarudžbu()
2. Promijeniti definiciju u klasi Klijent (to je zapravo ASP.NET aplikacija, to jeste, kontroleri koji će vršiti pozive ovih metoda iz paterna) tako da samo poziva metode klase NapraviNarudžbuFasada prilikom dodavanja novog korisnika

Decorator patern

Na našem dijagramu klasa, trenutno ne postoji potreba za primjenom Decorator paterna, jer nemamo dijelova gdje će se neki objekti „uređivati“. Možda bi ovaj patern bilo moguće primjeniti na izmjenu detalja o nekim proizvodima, ponudama proizvoda i sl, na zahtjev prodavača i u skladu sa njegovim novonastalim potrebama.

Koraci:

-Dodati interfejs IProizvoda koji će sadržati definicije metoda izmijeni() i getProizvod(), koje su prethodno definisane u klasi Proizvod;

- Dodati nekoliko novih klasa koje će se odnositi na pojedine vrste izmjene proizvoda(promijeniKategoriju, promijeniCijenu, promijeniOpis, i sl...), koje će naslijediti interfejs IProizvod i sadržavati objekat tipa klase koja je na sljedećem nivou apstraktnosti.

Bridge patern

Pri analizi Bridge paterna, došli smo u dilemu da li smo na našem dijagramu već primjenili ovaj patern, ili se ipak radi o Proxy paternu. Kod interfejsa IregledProizvoda, pomislili smo da smo primjenili Bridge patern, zbog toga što klasa Gost i klasa Kupac, na sličan način realizuju ovaj interfejs i njegove metode, ali smo na kraju ipak shvatili da se radi o Proxy paternu, jer se ovdje vodi računa o osiguravanju objekta od pogrešne upotrebe i pravima pristupa.
Dakle, na našem dijagramu nije primjenjen Bridge patern, ali slijedi objašnjenje zašto je primjenjen Proxy patern.

-Dodati novi interfejs IPregledProizvoda, koji će sadržavati definiciju metode za pregled detalja o proizvodu

- Dodati novu klasu Bridge, koja će sadržavati apstrakciju i kojoj će klijent jedino imati pristup.

Proxy patern

Proxy patern služi za dodatno osiguravanje objekata od pogrešne ili zlonamjerne upotrebe. Primjenom ovog paterna omogućava se kontrola pristupa objektima, te se onemogućava manipulacija objektima ukoliko neki uslov nije ispunjen, odnosno ukoliko korisnik nema prava pristupa traženom objektu.

Na trenutnom dijagramu klasa, primjetili smo da već postoji mjesto gdje smo nesvjesno primjenili Proxy patern, uz potrebne za nekim izmjenama, ali smo uočili potrebu za ovim paternom još dok nismo znali ni šta on prestavlja.
Uvođenjem aktera Gost, dali smo mu mogućnost da on može da pregleda detalje o proizvodu koji se tiču opisa, cijene i sl, ali ne i da vidi statistiku prodaje tog proizvoda u recimo zadnjih mjesec dana, niti da naruči taj proizvod. Tu mogućnost ima akter Kupac (registrovani korisnik). Dakle, pregled proizvoda je funkcionalnost koja treba biti dostupna i Gostu i Kupcu, ali na različite načine. Također, ukoliko bi se

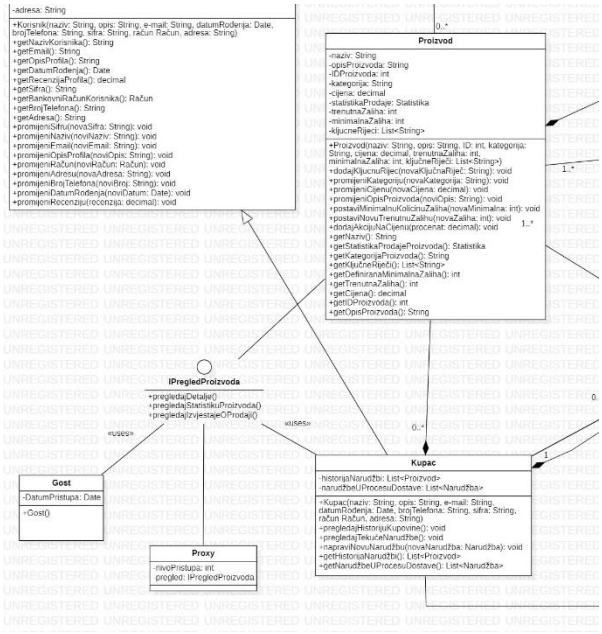
nekad u budućnosti javila potreba za još nekim akterima, kao što je recimo neka Inspekcija, koja bi mogla da pregleda i proizvode i njihovu statistiku prodaje i neke izvještaje o prodaji pojedinačnih proizvoda, to bi stvorilo probleme prilikom nadogradnje sistema, koja bi zahtijevala modifikaciju postojećih klasa. Stoga smo se odlučili iskoristiti Proxy patern upravo ovdje, zbog potrebe za kontrolom prava pristupa detaljima o proizvodu.
Koraci:

1. Prepraviti već postojeći interfejs IPregledProizvoda tako da u okviru njega budu definisane metode za pregled detalja o proizvodu
2. Definisati klasu Proxy koja će implementirati metodu pristup() u okviru koje će biti izvršena autentifikacija, te koja će naslijediti interfejs IpregledProizvoda i njegove metode.

Klasa klijent je zapravo ASP.NET aplikacija, to jeste, kontroleri koji će vršiti pozive ovih metoda iz Proxy paterna. Za sada je nismo stavljali na dijagram, ali ona bi bila u vezi sa klasom Proxy.

Upotrebom Proxy paterna smo unaprijedili svoj sistem i dali mogućnost da čak i ukoliko se javi potreba za uvođenjem novih klasa koje bi mogle zahtijevati pregled proizvoda i njihovih detalja, preko ovog paterna možemo kontrolisati njihovo pravo pristupa određenim objektima i detaljima o proizvodima.

Na slici je prikazano mjesto na našem dijagramu gdje smo upotrijebili Proxy patern:



<u>Composite patern</u>

Composite patern služi za kreiranje hijerarhije objekata. Koristi se kada svi objekti imaju različite implementacije nekih metoda, no potrebno im je svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija.

Ovaj patern nismo primjenili na svoj diagram, ali bi se sistem mogao unaprijediti upotrebom ovog paterna na sljedeći način: Ukoliko bi se pojavio zahtjev da neka klasa recimo želi saznati aktivnost svih korisnika sistema na određeni način. Recimo, da se aktivnost korisnika kupca račun kao ukupni broj pregledanih proizvoda kroz ukupni broj kupljenih proizvoda, a da se aktivnost korisnika prodavača računa kao ukupan broj ponuđenih proizvoda kroz ukupan broj prodanih proizvoda.
Tada bi se Composite patern realizovao na sljedeći način:

Koraci:

1. Dodati interfejs IAktivnostKorisnika koja će sadržavati definiciju metode za dobivanje vrijednosti aktivnosti nekog korisnika
2. Naslijediti ovaj interfejs od strane sve tri klase (Korisnici, Prodavač i Kupac), kako bi se kreirala hijerarhija objekata

Flyweight patern

Flyweight patern koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji svi u suštini predstavljaju jedan objekat. Samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (tzv. specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (tzv. bezlično stanje).

Mi ga nismo iskoristili na našem sistemu, ali bi to bilo moguće.
Mogućnost upotrebe ovog paterna može se uvidjeti u dijelu instanciranja klase Gost. Upotreba ovog paterna bi bila korisna u slučajevima kada je potrebno vršiti uštedu memorije.
Kada gost pristupi sistemu, ne bi se uvijek vršilo instanciranje novog objekta tipa Gost ukoliko on ponovo pristupa sistemu istog dana, već bi se na dnevnom nivou instancirao jedan objekat ovog tipa, a svako ponovno korištenje tog objekta, razlikovalo bi se samo po broju pristupa za taj dan. Kada bi osoba pristupila više puta sistemu u ulozi gosta istog dana, on ne bi znao da svaki put prustupa preko iste instance, ali bi to bilo veoma korisno za sistem da ne bi instancirao svakog puta novi objekat tipa Gost, a zapravo se radi o istoj osobi.

Samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (tzv. specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (tzv. bezlično stanje).

Dakle, u našem slučaju, specifično stanje bi bilo pojavljivanje novog gosta, koji taj dan nije pristupao sistemu, a bezlično stanje je opća istanca koja već postoji (instancirana je).

Koraci:

-Dodati interfejs IPristupGosti koji će sadržati definiciju metode dajBrojPristupa() za razlikovanje rednog broja pristupa gosta;

- Definisati jedinstvenu metodu dajPristup() u okviru klase Pristup koju će klijent pozvati kada neko želi pristupiti kao gost.

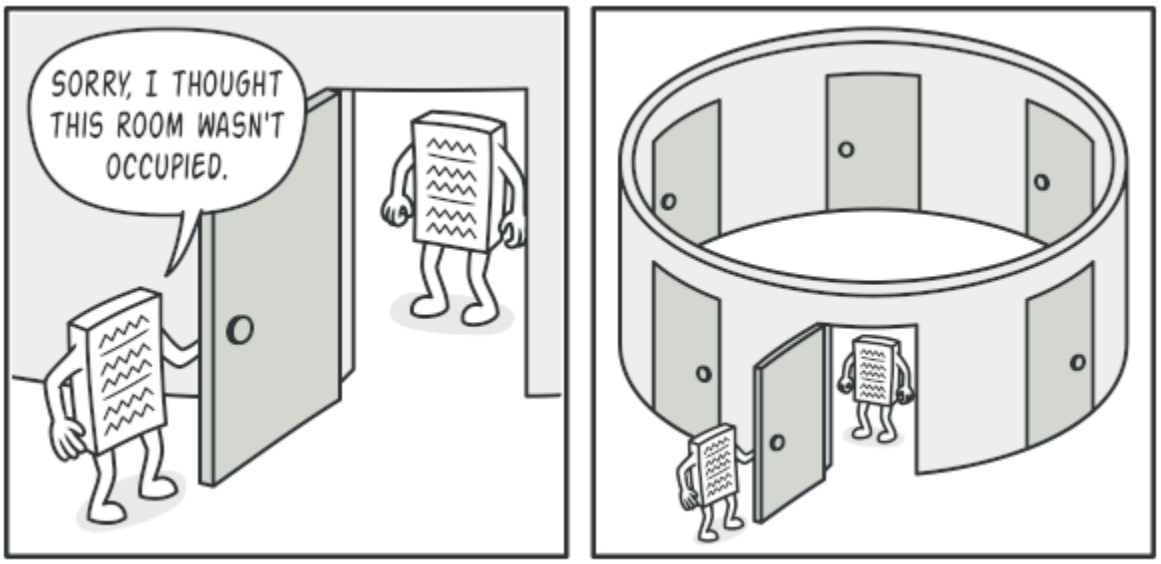
KREACIJSKI PATERNI

PATERN	NAMJENA PATERNA	DA LI SMO GA KORISTILI?
Singleton patern	Uloga Singleton paterna je da osigura da se klasa može instancirati samo jednom i da osigura globalni pristup kreiranoj instanci klase.	DA
Prototype patern	Uloga Prototype paterna je da kreira nove objekte klonirajući jednu od postojećih prototip instanci (postojeći objekat). Prototype patern je često koristan i prilikom višestrukog korištenja podataka iz baze: Npr. Potrebno je uraditi analizu prodaje na skupu podataka iz baze. Potrebno je kopirati podatke iz baze podataka, enkapsulirati ih u objekat i nad njima raditi analizu	DA
Factory Method patern	Uloga Factory Method paterna je da omogući kreiranje objekata na način da podklase odluče koju klasu instancirati.	ZA SADA NISMO
Abstract Factory patern	Abstract Factory patern omogućava da se kreiraju familije povezanih objekata/produkata. Na osnovu apstraktne familije produkata kreiraju se konkretne fabrike (factories) produkata različitih tipova i različitih kombinacija.	NE, ALI POSTOJI MOGUĆNOSTI
Builder patern	Uloga Builder paterna je odvajanje specifikacije kompleksnih objekata od njihove stvarne konstrukcije. Isti konstrukcijski proces može kreirati različite reprezentacije.	NE

<u>Singleton patern</u>

Uloga Singleton paterna je da osigura da se klasa može instancirati samo jednom i da osigura globalni pristup kreiranoj instanci klase.
Potrebu za singleton paternom na našem djagramu smo uočili kod klase Korisnici, jer smatramo da je najbolje rješenje da ova klasa bude instancirana samo jednom, kako bi postojao isključivo jedan objekat koji objedinjuje sve korisnike. Ukoliko to ne bi bio slučaj, skupine korisnika bi se mogle nalaziti u više različitih objekata tipa Korisnici, ali smatramo da to nije dobra ideja, jer svi korisnici trebaju da budu „ravnopravni“ – svi prodavači i svi kupci zapravo su „isti“, kada se radi o njihovim nalozima. Nema nikakve potrebe da se neki od prodavača ili kupaca, naprimjer nalaze u jednom objektu Korisnici, a neki

drugi prodavači ili kupci u drugim objektima. Cilj nam je objediniti ih na jedno mjesto, kako bi se lakše obavljale sve funkcionalnosti koje se tiču korisnika uopšteno. S tim ciljem je i kreirana klasa Korisnici, kako bih sve zajedno držala na okupu kao korisnike. To nas je dovelo do zaključka da ćemo ovdje upotrijebiti Singleton patern, kako bismo zaista postigli ono što je prikazano na sljedećoj fotografiji, a za čim postoji potreba na našem dijagramu:



Dakle, kada se naš korisnik registruje, on neće moći „birati“ na koja vrata želi uči, to jeste kojim korisnicima se želi pridružiti, već će postojati samo mogućnost da se svi oni svrstaju u jednu jedinu skupinu korisnika. Svaki pokušaj kreiranja nove skupine korisnika, rezultirao bi zapravo nadogradnjom na postojeći objekat Korisnici, te sve funkcionalnosti bi se odnosile zapravo na već postojeći objekat.

Koraci kreiranja Singletone paterna:

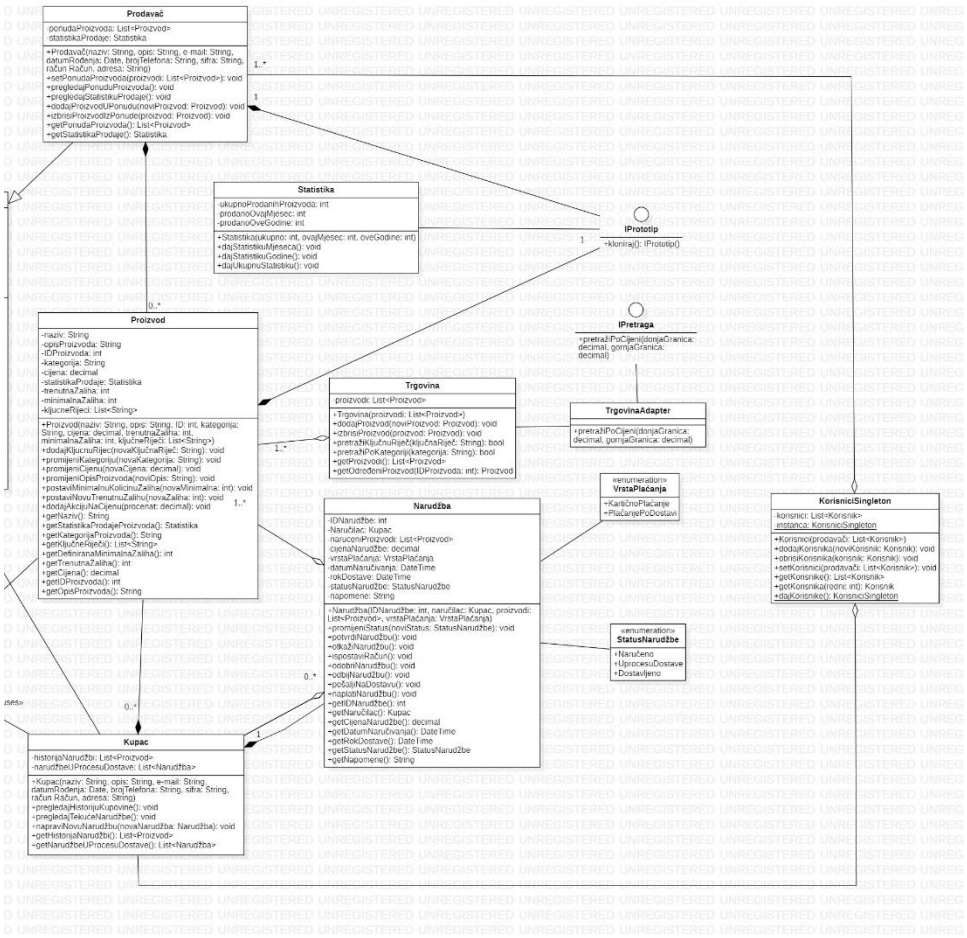
- Dodati statički atribut u klasu KorisniciSingleton tipa KoricniciSingleton, koji se označava kao statički;

- Dodati statičku metodu dajKorisnike() u klasi KorisniciSingleton koja će vršiti vraćanje jedinstvenog statičkog atributa iz ove klase;

-Ostale metode u klasi KorisniciSingleton će kontrolisati eventualnu promjenu korisnika (ali istog, jedinstvenog objekta)

Na isti način je moguće upotrijebiti Singleton patern kod klase Trgovina, kako bi se ova klasa instancirala samo jednom, ali još uvijek imamo dilemu, da li da primjenimo Singleton patern, ili da na ovom mjestu primjenimo Abstract Factory patern...

Na slici je prikazano mjesto na našem dijagramu gdje smo upotrijebili Singleton patern:



<u>Prototype patern</u>

Moguću potrebu za ovim paternom smo uočili na dijelu našeg dijagrama koji se tiče Statistike prodaje nekog kupca. Sa dijagrama se uočava da će statistika prodaje biti zabilježena za svaki pojedinačni proizvod, te će se takve statistike spremati u bazu podataka. Ali onda kada treba vršiti statističke analize za sve pojedinačne proizvode iz ponude nekog prodavača, ne bi bilo praktično pristupati ponovo bazi podataka i čitati pojedinačne statistike. Zbog toga smo ovdje uvidjeli potrebu za Prototype paternom, tako da bi se, kada treba vršiti statističku analizu prodaje prodavača, ove postojeće statistike pojedinačnih proizvoda klonirale, i nad njima bi se vršile potrebne analize.

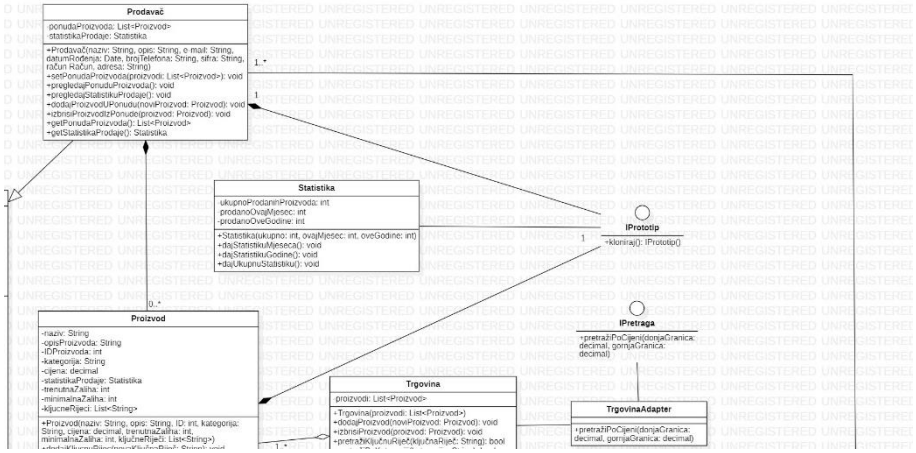
Ovdje smo, dakle, uočili primjenu Prototype paterna, jer je on često koristan upravo prilikom višestrukog korištenja podataka iz baze: Potrebno je kopirati podatke iz baze podataka, enkapsulirati ih u objekat i nad njima raditi analizu. Ako je potrebno uraditi i druge analize nad istim skupom podataka nije dobra ideja ponovo pristupati bazi podataka, čitati podatke i kreirati objekat za njihovo smještanje. Ako se koristi Prototype patern tada se objekat koji se koristi za analizu prodaje klonira i koristi za drugu analizu. Naprimjer, moguće je zatražiti statistiku prodaje nekog proizvoda do polovine tekućeg mjeseca, kopirati je u neki objekat, sačuvati je kao zasebnu statistiku, vršiti izmjene nad njom, a da to ne utječe na orginalni objekat.
Koraci:

-Definisati interfejs IPrototip koji se sastoji od metode kloniraj();

- Naslijediti interfejs IPrototip od klase Statistika, te implementirati metodu kloniraj() koja će kreirati duboku kopiju objekta;

- Promijeniti implementaciju klijenta tako da se vrši kloniranje postojećeg oblika pri njegovom kopiranju.

Na slici je prikazano mjesto na našem dijagramu gdje smo upotrijebili Prototype patern:



<u>Factory Method patern</u>

Uloga Factory Method paterna je da omogući kreiranje objekata na način da podklase odluče koju klasu instancirati. Factory Method instancira odgovarajuću podklasu(izvedenu klasu) preko posebne metode na osnovu informacije od strane klijenta ili na osnovu tekućeg stanja.
Ovaj patern bi se na našem dijagramu mogao upotrijebiti kod procesa registracije korisnika, tako da se instancira objekat tipa Kupac ili objekat tipa Prodavač, na osnovu informacije od strane klijenta. Ovaj patern bi u tom smislu znatno unaprijedio naš sistem, ukoliko se ukaže potreba da je potrebno registrovati račune nekih drugih Korisnika, koji nisu ni Kupac ni Prodavač. Naprimjer, ukoliko se ukaže potreba za nekom novom klasom Dostavljač, nadogradnja sistema bi bila u tom slučaju puno jednostavnija, jer bi pomoću ovog paterna registacija korisnika ostala ista, bilo bi potrebno samo dodati novu klasu Dostavljač. Dakle, registacija bi se vršila na osnovu informacije od strane klijenta ili tekućeg stanja, te bi se instancirala klasa koja je potrebna.
Za sada, nismo upotrijebili ovaj patern na našem dijagramu, ali na osnovu ove analize smo uvidjeli gdje bi njegova upotreba mogla biti koristna.
Koraci:

- IProduct – interfejs za produkte;
- ProductA, ProductB klase koje implementiraju interfejs;
- Creator klasa posjeduje FactoryMethod() metodu; FactoryMethod() metoda odlučuje koju klasu instancirati.

Klijent može imati više od jednog kreatora za različite tipove produkata.

Abstract Factory patern

Abstract Factory patern omogućava da se kreiraju familije povezanih objekata/produkata. Na osnovu apstraktne familije produkata kreiraju se konkretne fabrike (factories) produkata različitih tipova i različitih kombinacija.

Na našem dijagramu, nismo iskoristili ovaj patern, ali smo dobili ideju kako bi se naš sistem mogao unaprijediti upotrebom istog. Ukoliko bi se javio zahtjev da naši proizvodi budu razvrstani po nekim zamišljenim „prodavnicama“, kao naprimjer u Tržnom centru, tada bi se kreirale zasebne vrste prodavnica, koje nude potpuno različite vrste proizvoda: MiniMarket (nudi neke prehrambene proizvode i sl...), Pijaca (nudi poljoprivredne proizvode, domaće prerađevine i sl...), Butik (nudi isključivo odjevne proizvode, nakit...), Drogerija(nudi kozmetiče proizvode, higijenske proizvode, i sl...), Namještaj(nude namještaj, proizvode za uređenje doma, i sl..)... U ovisnosti od vrsta prodavnica razlikovali bi se prizvodi koje nude.
Tada bi sve ove prodavnice zajedno funkcionisale kao u „Tržnom centru“, kao familija povezanih objekata. Na osnovu apstraktne familije produkata kreirale bi se konkretne fabrike (factories) produkata različitih tipova i različitih kombinacija.

Koraci:

- IFactory interfejs- apstraktni interfejs sa Create operacijama za svaku fabriku (factory) proizvoda;
- Factory1, Factory2,...klase – implementiraju operacije kreiranja za pojedinačne fabrike;
- IProductA, IProductB,... apstraktni interfejsi za pojedinačne grupe produkata (A,B,...)
- ProductA1, ProductA2, ProductB1, ProductB2 klase koje implementiraju IProductA, IProductB interfejse i definiraju objekte produkata koji se kreiraju za odgovarajuću fabriku.
- U sastavu paterna je i Client klasa - preko interfejsa IFactory i IProductA, IProductB koristi objekte (produkte) fabrike.

Builder patern

Uloga Builder paterna je odvajanje specifikacije kompleksnih objekata od njihove stvarne konstrukcije. Isti konstrukcijski proces može kreirati različite reprezentacije. Upotreba Builder paterna se često može naći u aplikacijama u kojima se kreiraju kompleksne strukture. Koristi se kada je neovisan algoritam za kreiranje pojedinačnih dijelova, kada je potrebna kontrola procesa konstrukcije, kada se više objekata na različit način sastavlja od istih dijelova.

Na našem dijagramu nismo upotrijebili ovaj patern, jer nemamo dijelova gdje je potrebno nešto kompleksno graditi od manjih dijelova, tj. da je potrebna kontrola njihove konstrukcije. U našem sistemu, uglavnom smo razdvojili sve pojedine dijelove, tako da klase sadrže sve što je potrebno za neki objekat. Ovaj patern bi se, međutim, mogao upotrijebiti kod kreiranja kompleksnijih korisničkih profila, nego što smo mi to zamislili u našem sistemu. Ukoliko bi, recimo, kreiranje korisničkih računa zahtijevalo da svaki profil izgleda onako kako to korisnik želi, mogli bismo iskoristiti ovaj patern. Tada bi se proces kreiranja korisničkog profila znatno zakomplikovao, jer bi, naprimjer nudili mogućnost da se profil sastoji

od više različitih dijelova, koji bi se razlikovali od profila do profila. U našem sistemu svaki korisnički račun imat će iste dijelove. Ukoliko to ne bi bio slučaj, moglo bi se uvesti da, naprimjer, svaki korisnički profil može, ali ne mora, da ima fotografiju profila, može da ima opis profila, ali ne mora, i sl... Pošto smo mi zamislili da svi naši korisnici moraju da registruju račun koji sadrži tačno određene dijelove, nismo uočili potrebu za ovim paternom.

Da smo se ipak odlučili na upotrebu ovog paterna, koraci pri upotrebi bi bili sljedeći:

- IBuilder- interfejs koji definira pojedinačne dijelove koji se koriste za izgradnju produkta;
- Director klasa koja sadrži neophodnu sekvencu operacija za izgradnju produkta;
- Builder klasa koja se poziva od strane direktora (Director klasa) da se izgradi produkt;
- Product klasa na osnovu koje se kreira objekat koji se gradi preko dijelova.

PATERNI PONAŠANJA

PATERN	NAMJENA PATERNA	DA LI SMO GA KORISTILI?
Strategy patern	Izdvaja algoritam iz matične klase i uključuje ga u posebne klase.	DA
State patern	Mijenja način ponašanja objekata na osnovu trenutnog stanja.	DA
TemplateMethod patern	Omogućava algoritmima da izdvoje pojedine korake u podklase.	NE
Observer patern	Uspostavlja relaciju između objekata takvu da kad se stanje jednog objekta promijeni svi vezani objekti dobiju informaciju.	DA
Iterator patern	Omogućava pristup elementima kolekcije sekvencijalno bez poznavanja interne strukture kolekcije.	NE

<u>Strategy patern</u>

Uloga: Strategy patern izdvaja algoritam iz matične klase i uključuje ga u posebne klase.
Pogodan je kada postoje različiti primjenjivi algoritmi (strategije) za neki problem.

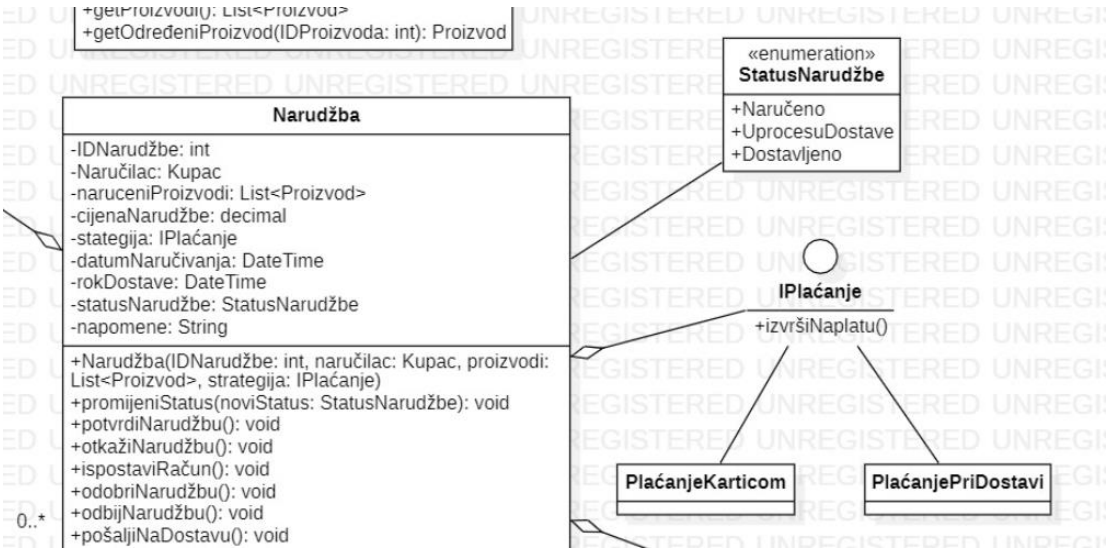
Moguća upotreba ovog paterna na našem dijagramu uočava se na dijelu koji se tiče plaćanja narudžbe. Kupac može da izabere plaćanje narudžbe karticom i plaćanje prilikom dostave narudžbe. Pri tome bi proces plaćanja imao dva različita algoritma(strategije).

Koraci:

Context (Narudžba) - Klasa preko koje **Client** klasa daje kontekstualne informacije za IStrategy algoritme. Sadržavat će **switch iskaz ili kaskadni if iskaz** za odluku koja strategija se bira (na osnovu vrstaPlaćanja iz Narudžba)

IStrategy(u našem slučaju IPlaćanje) - definira zajednički interfejs za sve algoritme (strategije).

StrategyA, StrategyB (PlaćanjeKarticom, PlaćanjePriDostavi) – Klase koje implementiraju algoritme (konkretne strategije) tj. IStrategy interfejs.



State patern

State Pattern je dinamička verzija Strategy paterna.
Objekat mijenja način ponašanja na osnovu trenutnog stanja.

Upotrebu ovog paterna na našem dijagramu smo uočili kod dijela promjene statusa (stanja) narudžbe. Objekat tipa Narudžba može da ima sljedeća stanja: Naručeno, UProcesuDostave, Dostavljeno.

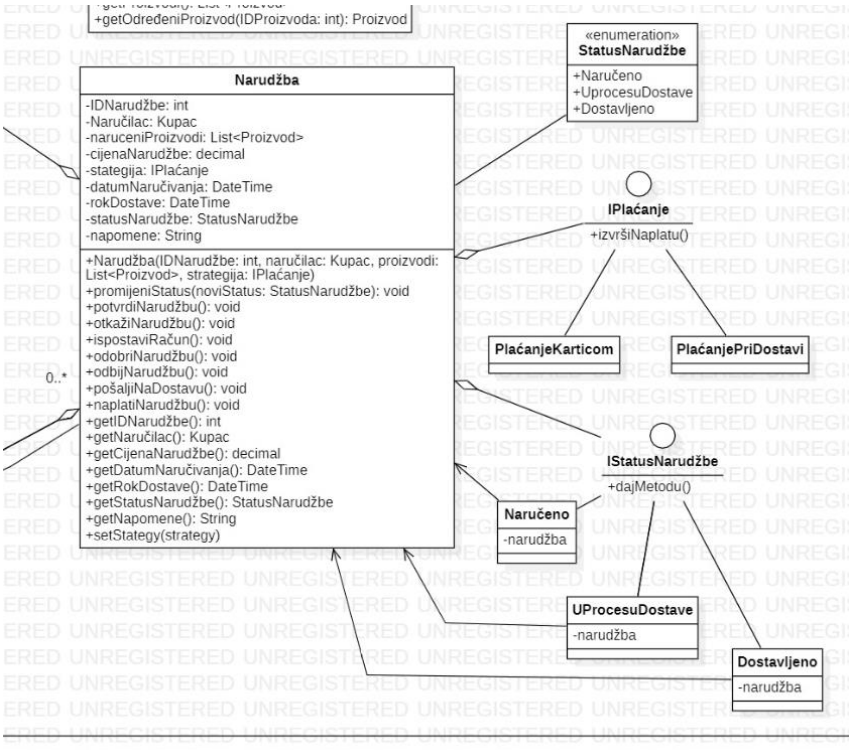
Na osnovu stanja(statusa) koji narudžba trenutno ima, objekat Narudžba će mijenjati način ponašanja. Naprimjer, ukoliko narudžba ima status Naručeno (kada kupac tek naruči i narudžba čeka da ode u proces dostave), svi proizvodi koji su uključeni u tu narudžbu, automatski treba da prestanu biti u daljnjoj ponudi tih određenih prodavača, jer su zapravo već prodani i više ih ne može kupiti niko drugi (ili ukoliko prodavač ima više takvih identičnih proizvoda koji su prodani, njihova trenutna zaliha se treba smanjiti).
Nakon što narudžba poprimi status UProcesuDostave, ovakva narudžba biva dodana u listu NarudžbeUProcesuDostave kupca koji je naručio tu narudžbu.
Kada narudžba dobije status Dostavljeno, ona biva dodana u HistorijaNarudžbi kupca, a izbrisana iz NarudžbeUProcesuDostave kupca.

Koraci:

Context (Narudžba): održava instancu stanja koja definira **tekući kontekst i interfejs** koji je od interesa za klijenta. **Client klasa** komunicira sa ovom klasom.

Istate (IStatusNarudžbe): Interfejs ili apstraktna klasa koja definira ponašanje povezano sa svim mogućim stanjima klijenta.

StateA, StateB:,..(Naručeno, UProcesuDostave, Dostavljeno) Klase koje implementiraju konkretno stanje objekta (klijenta) Context klase. Svako stanje se predstavlja sa jednom konkretnom klasom.



TemplateMethod patern

Omogućava **izdvajanje određenih koraka algoritma u odvojene podklase**.

Struktura algoritma se ne mijenja - mali dijelovi operacija se izdvajaju i ti se dijelovi mogu implementirati različito.

Što se tiče TemplateMethod paterna, imali smo dilemu da li primjeniti ovaj patern kod procesa Plaćanja narudžbe ili upotrijebiti Strategy patern, te nam je objašnjenje Strategy paterna bilo puno logičnije kod ove situacije, jer TemplateMethod patern, zapravo služi kako bi se jedan algoritam izdvojio na odvojene podklase (kao neki koraci rješavanja algoritma), a Strategy podrazumijeva upotrebu potpuno različitih algoritama rješavanja istog problema.

Da smo se odlučili za upotrebu TemplateMethod paterna, kada bi postojao jedan način plaćanja koji bi se sprovodio u više koraka, koraci za upotrebu TemplateMethod paterna bi bili sljedeći:

Algorithm – klasa koja uključuje **TemplateMethod.**

TemplateMethod (Plaćanje)– metoda koja izdvaja dijelove svojih operacija u druge klase.

IPrimitive(INaplati) -interfejs koji definira operacije koje TemplateMethod izdvaja u druge klase (validirajPlaćanje, izvršiNaplatu…)

AnyClass (Narudžba) – klasa koja implementira IPrimitives interfejs.

Operation (izvršiNaplatu) jedna od metoda koje TemplateMethod treba da bi završio svoju operaciju.

Observer patern

Uloga Observer paterna je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju.

Pošto je naš system namijenjen za trgovinu, naravno da je vrlo korisna upotreba Observe paterna.

Nudit će se mogućnost da kupci mogu da se pretplate na obavještenja koja se tiču novih ponuda proizvoda, akcijskih cijena, i sl…

Kupcu se šalje upit da li želi da mu stižu obavještenja na njegov e-mail o pojavi novih proizvoda ili izmjeni cijena (dodavanje akcijskih cijena) određenih proizvoda koji su u domenu interesovanja kupaca.

Koraci:

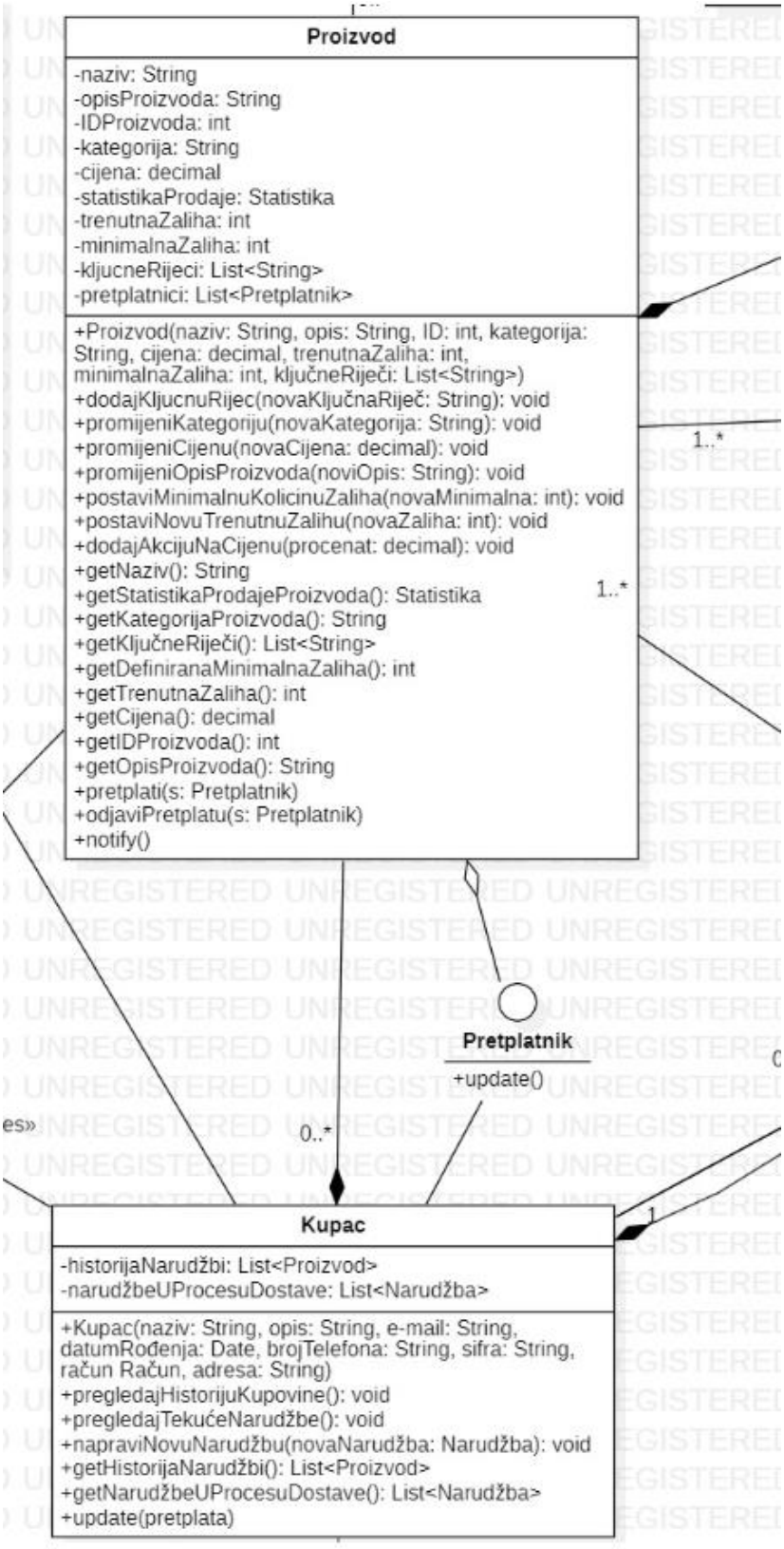
Subject klasa (Proizvod) – instanca ove klase mijenja svoje stanje i obavještava Observers klase.

IObserver (Pretplatnik)– Interfejs za Observer klase, sadrži samo jednu metodu (Update) koja se poziva kada se promijeni stanje neke Subject instance.

Observer(Kupac(konkretan Pretplatnik)) – konkretna klasa koja obezbjeđuje konkretnu implementaciju za IObserver interfejs.

Update (prijem obavijesti putem e-maila) – metoda koja formira interfejs između klasa Subject i Observer.

Notify (e-mail koji se salje svim kupcima koji su odabrali pretplatu na novosti) -*Event* mehanizam za pozivanje Update operacije za sve posmatrače (Observer objekte).


Proizvod
-naziv: String
-opisProizvoda: String
-IDProizvoda: int
-kategorija: String
-cijena: decimal
-statistikaProdaje: Statistika
-trenutnaZaliha: int
-minimalnaZaliha: int
-kljucneRijeci: List<String>
-pretplatnici: List<Pretplatnik>
+Proizvod(naziv: String, opis: String, ID: int, kategorija: String, cijena: decimal, trenutnaZaliha: int, minimalnaZaliha: int, ključneRiječi: List<String>)
+dodajKljucnuRijec(novaKljučnaRiječ: String): void
+promijeniKategoriju(novaKategorija: String): void
+promijeniCijenu(novaCijena: decimal): void
+promijeniOpisProizvoda(noviOpis: String): void
+postaviMinimalnuKolicinuZaliha(novaMinimalna: int): void
+postaviNovuTrenutnuZalihu(novaZaliha: int): void
+dodajAkcijuNaCijenu(procenat: decimal): void
+getNaziv(): String
+getStatistikaProdajeProizvoda(): Statistika
+getKategorijaProizvoda(): String
+getKljučneRiječi(): List<String>
+getDefiniranaMinimalnaZaliha(): int
+getTrenutnaZaliha(): int
+getCijena(): decimal
+getIDProizvoda(): int
+getOpisProizvoda(): String
+pretplati(s: Pretplatnik)
+odjaviPretplatu(s: Pretplatnik)
+notify()
1..*
1..*
Pretplatnik
+update()
0..*
es»
Kupac
-historijaNarudžbi: List<Proizvod>
-narudžbeUProcesuDostave: List<Narudžba>
+Kupac(naziv: String, opis: String, e-mail: String, datumRođenja: Date, brojTelefona: String, sifra: String, račun Račun, adresa: String)
+pregledajHistorijuKupovine(): void
+pregledajTekućeNarudžbe(): void
+napraviNovuNarudžbu(novaNarudžba: Narudžba): void
+getHistorijaNarudžbi(): List<Proizvod>
+getNarudžbeUProcesuDostave(): List<Narudžba>
+update(pretplata)
1

Iterator patern

Iterator patern omogućava pristup elementima kolekcije sekvencijalno bez poznavanja interne strukture kolekcije.
Upotreba iterator paterna na našem dijagramu je moguća kod klase Korisnici, kako bi se na jednostavan način moglo pristupati (npr. prilikom pretrage) korisnicima, ali korištenje iteratora može biti manje učinkovito od prolaska kroz elemente nekih specijaliziranih kolekcija direktno. Zbog toga, mi smo se ipak odlučili da ne upotrijebimo ovaj patern, jer je u pitanju obična lista korisnika, kojim već možemo pristupati na jednostavan način.
Ukoliko bismo se odlučili na upotebu ovog paterna, koraci bi bili sljedeći:

Client - sadrži Collection objekt i koristi foreach iskaz za iteraciju kolekcije.
IEnumerable(I – definira interfejs za operaciju GetEnumerator.
Collection (Korisnici) – tip podatka koji sadrži sposobnost generiranja kolekcije vrijednosti.
GetEnumerator – metoda koja pruža vrijednosti kolekcije u sekvenci.
OtherOperations – druge metode koje pružaju vrijednosti kolekcije u nekom drugom redoslijedu ili obliku.