PATERNI

Strukturalni paterni

Flyweight patern (implementiran)

Flyweight patern je iskorišten na našem sistemu kod klasa ObicniKorisnik i VipKorisnik. Naime s obzirom da obje klase imaju atribute tipa lista Knjiga (u klasi ObicniKorisnik – procitaneKnjige, u klasi VipKorisnik – zatrazeneKnjigeZaDodavanje), i s obzirom da nema smisla da se u ovim listama drže iste knjige iskorišten je flyweight patern. Odnosno, uzmimo za primjer u klasi ObicniKorisnik, ako korisnik po drugi put iznajmi knjigu koju je već jednom iznajmljivao, nema potrebe da se ta ista knjiga drži dva puta u listi pročitanih knjiga. Analogno ako kojim slučajem vip korisnik zatraži dva puta istu knjigu također nema potrebe da se u listi zatraženih knjiga nalazi ta knjiga dva puta.

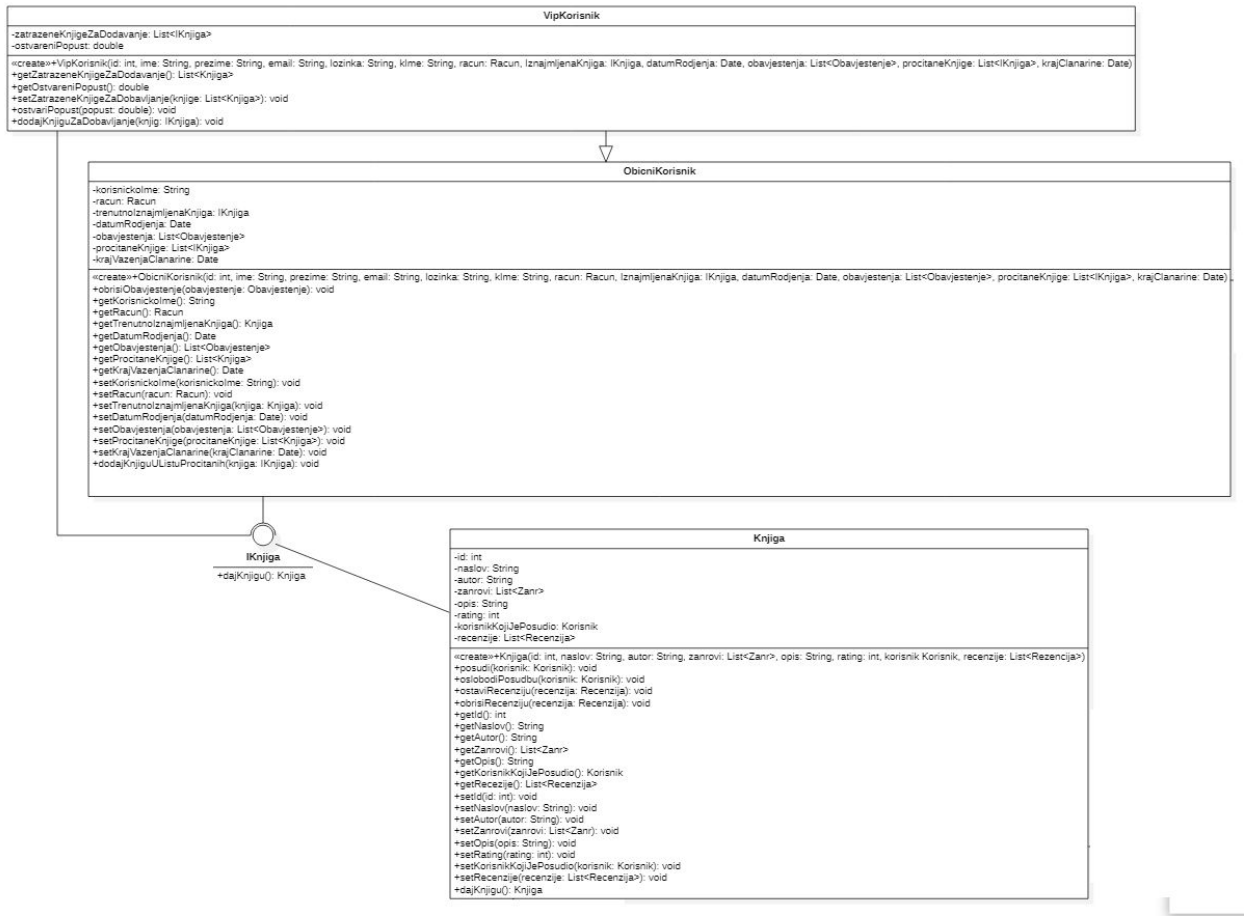
Izmjene na dijagramu klasa

Na dijagramu klasa dodan je novi interfejs IKnjiga, koji definiše metodu dajKnjigu(): Knjiga. Interfejs realizuje klasa Knjiga, odnosno implementira metodu dajKnjigu(). Dok s druge strane klase ObicniKorisnik i VipKorisnik sada umjesto atributa tipa Knjiga imaju atribute tipa IKnjiga. Također u klasi ObicniKorisnik dodana je i metoda dodajKnjiguUListuProcitanih(knjiga: IKnjiga): void. U okviru ove metode se pristupa atributu Biblioteka.knjige i vrši operacija if Biblioteka.Knjige.Find(x => x.dajKnjigu() == knjiga.dajKnjigu()) - na ovaj način se vrši ušteda memorije, a ako rezultat operacije bude null onda se nova knjiga dodaje u listu knjiga i povezuje sa korisnikom.

Svrha korištenja flyweight paterna

Iako upotrebom flyweight paterna se povećava broj potrebnih klasa u sistemu i naizgled dosta jednostavne metode se izdvajaju u poseban interfejs, svrha korištenja flyweight paterna se ugleda svakako u uštedi memorije, ali i u čitljivosti koda u fazi implementacije.

Slika sa izmjenama na dijagramu klasa slijedi ispod.



Composite patern (implementiran)

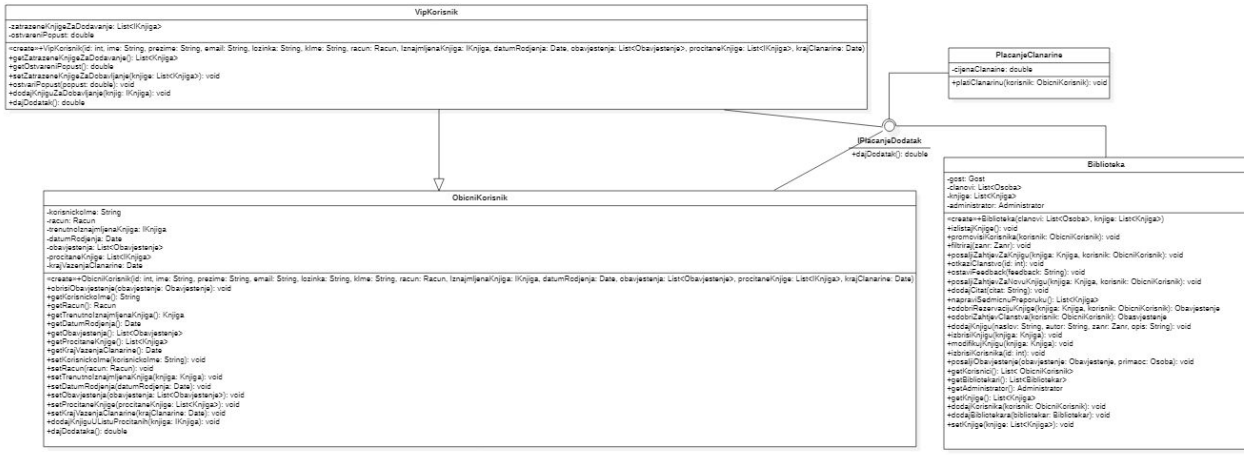
U našem sistemu composite patern smo iskoristili pri implementaciji funkcionalnosti placanja clanarine. Ovo smo uradili da bismo odrzali Open-Closed princip u sistemu. Definisat cemo interfejs kojeg ce nasljedjivati razlicite podklase, a koji se poziva pri placanju clanarine. Razlog zasto je izabran composite pattern, a ne bridge je jer se dodatak koji se koristi pri proracunu clanarine dobiva na potpuno razlicit nacin u zavisnosti od podklase. Dodatak se dobiva kao rezultat polimorfno realizovane funkcije dajDodatak().

Izmjene na dijagramu klasa

Umjesto pojedinačnih metoda u klasama ObicniKorisnik i VipKorisnik za plaćanje članarine, kreirana je klasa PlacanjeClanarine koja ima atribut cijenaClanarine: double kao i metodu platiClanarinu(korisnik: ObicniKorisnik): void. Definisemo interfejs IPlacanjeDodatka koji definise metodu dajDodatak() i kojeg nasljedjuju klase ObicniKorisnik i VipKorisnik.

Svrha korištenja composite paterna

Ovim paternom smo najprije željeli održati O princip, pa smo kreirali hijerarhiju objekata za lakši i jednostavniji rad sa njima.



Adapter patern (implementiran)

Adapter patern se naslanja na composite pattern i koristi se pri radu sa popustima u sistemu.

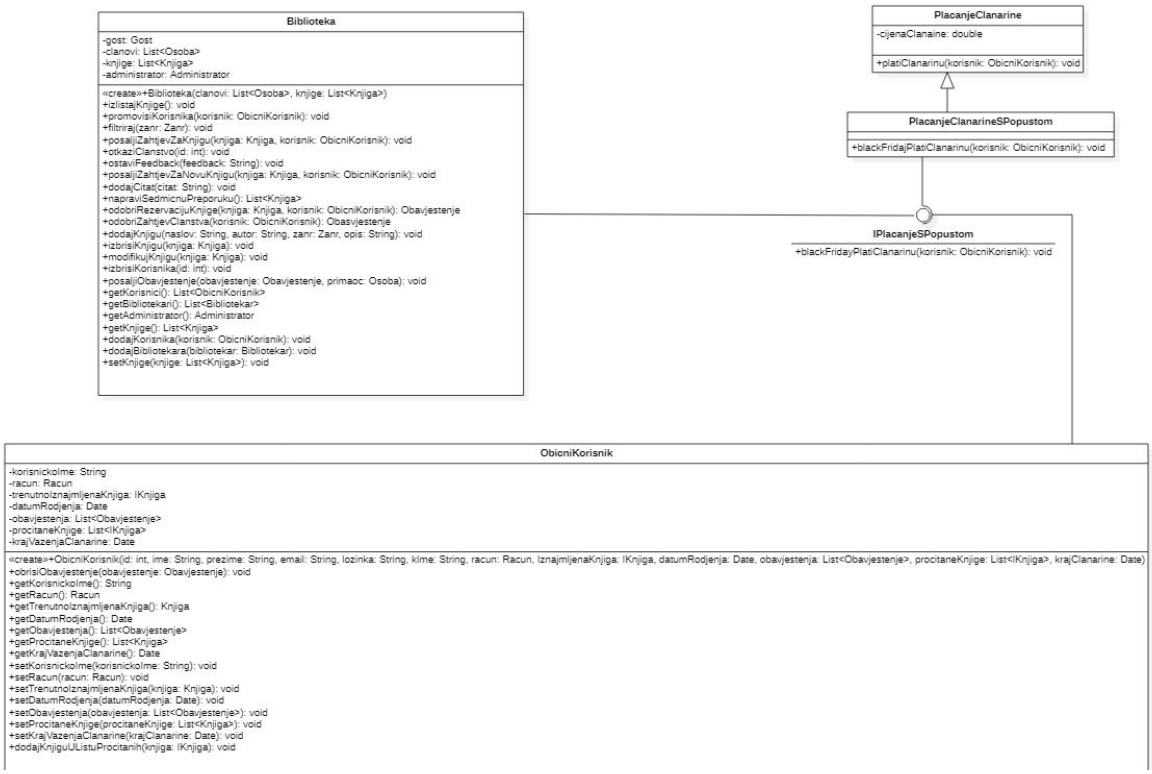
Izmjene na dijagramu klasa

Za realizaciju adapter paterna dodan je interfejs IPlacanjeSPopustom koji definiše metodu blackFridajPlatiClanarinu(korisnik: ObicniKorisnik): void, zatim je dodana adapter klasa PlacanjeClanarineSPopustom koja realizira interfejs, odnosno implementira njegovu metodu.

Svrha korištenja adapter paterna

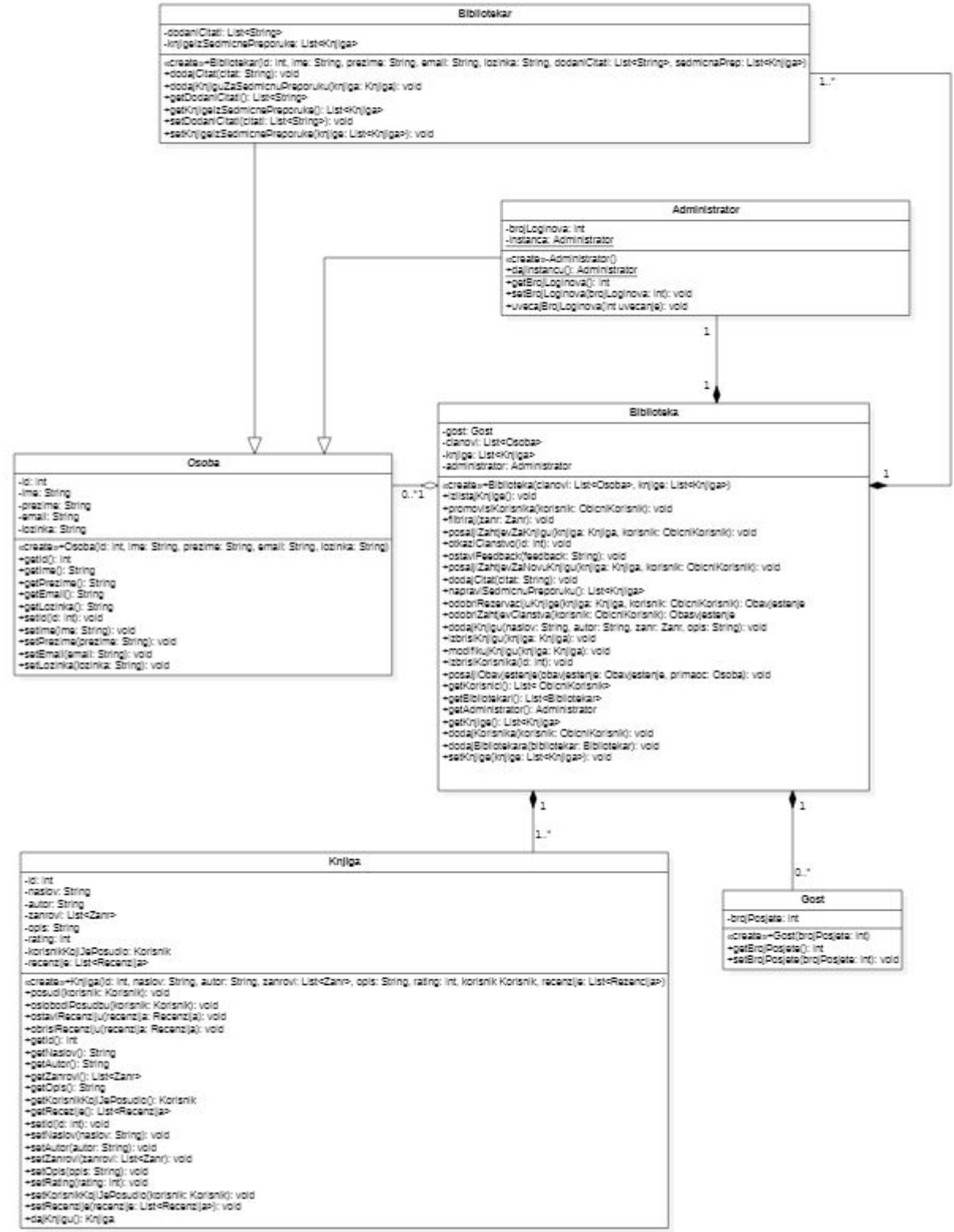
Svrha adapter paterna se ogleda u tome da sa ovakvim pristupom imamo mogućnost dodavanja novih funkcionalnosti koje su potrebne u nekim specijalnim okolnostima, a koje se sve oslanjaju na istu funkcionalnost plaćanja članarine. Mi smo sada u mogućnosti ne samo da implementiramo neki specijalni obračun popusta za tzv. crni petak, a zatim pozovemo metodu platiClanarinu iz klase PlacanjeClanarine, nego smo u mogućnosti da dodajemo različite implementacije obračunavanja popusta za različite prillike npr. neki praznik, bez potrebe za kreiranjem dodatnih klasa. Sve što je potrebno jeste dodati novu metodu u interfejs i u klasu PlacanjeClanarineSPopustom.

Izmjene na dijagramu klase su prikazane na sljedećoj slici:



Facade patern (implementiran)

Facade patern jeste patern koji se često intuitivno iskoristi i bez razmišljanja. Tako i u našem sistemu, ako pogledamo npr. klasu Biblioteka, ona objedinjuje mnoge metode i funkcionalnosti u svom interfejsu i time pruža više pogleda visokog nivoa na podsisteme. Klasa Biblioteka se koristi različitim klasama, između ostalog klasama Knjiga, Administrator, Bibliotekar, Gost, Osoba, Zanr, Obavjestenje. Metode unutar klase Biblioteka su kompleksnije, ali time se omogućava da korisnik pozivom samo jedne metode dobije ono što traži, bez da je upoznat sa kompleksnošću koja stoji iza te metode. Za primjer uzmimo metodu posaljiZahtjevZaKnjigu(knjiga: Knjiga, korisnik: ObicniKorisnik): void. Klijentu je dovoljno da proslijedi dva parametra, a metoda će izvršiti sve što je neophodno da se obavijesti bibliotekar o zatraženoj knjizi, o tome ko je zatražio knjigu i kada. Klasa Biblioteka je centralni dio sistema preko kojeg se većina funkcionalnosti odvija, a na slici ispod vidimo klasu sa njenim vezama sa drugim klasama. Klasa Biblioteka koristi čak 5 drugih klasa u svom radu, tako da klijent ne treba imati nikakvo poznavanje unutrašnjeg rada pojedinačnih klasa kako bi mogao koristiti kontejnersku klasu.


Bibliotekar
-dodaniCitati: List<String>
-knjigeIzSedmicnePreporuke: List<Knjiga>
«create»+Bibliotekar(id: int, ime: String, prezime: String, email: String, lozinka: String, dodaniCitati: List<String>, sedmicnaPrep: List<Knjiga>)
+dodajCitat(citat: String): void
+dodajKnjiguZaSedmicnuPreporuku(knjiga: Knjiga): void
+getDodaniCitati(): List<String>
+getKnjigeIzSedmicnePreporuke(): List<Knjiga>
+setDodaniCitati(citati: List<String>): void
+setKnjigeIzSedmicnePreporuke(knjige: List<Knjiga>): void
1..*
Administrator
-brojLoginova: int
-instanca: Administrator
«create»-Administrator()
+dajInstancu(): Administrator
+getBrojLoginova(): int
+setBrojLoginova(brojLoginova: int): void
+uvecajBrojLoginova(int uvecanje): void
1
1
Biblioteka
-gost: Gost
-clanovi: List<Osoba>
-knjige: List<Knjiga>
-administrator: Administrator
«create»+Biblioteka(clanovi: List<Osoba>, knjige: List<Knjiga>)
+izlistajKnjige(): void
+promovisiKorisnika(korisnik: ObicniKorisnik): void
+filtrirajZanr: Zanr): void
+posaljiZahtjevZaKnjigu(knjiga: Knjiga, korisnik: ObicniKorisnik): void
+otkaziClanstvo(id: int): void
+ostaviFeedback(feedback: String): void
+posaljiZahtjevZaNovuKnjigu(knjiga: Knjiga, korisnik: ObicniKorisnik): void
+dodajCitat(citat: String): void
+napraviSedmicnuPreporuku(): List<Knjiga>
+odobriRezervacijuKnjige(knjiga: Knjiga, korisnik: ObicniKorisnik): Obavjestenje
+odobriZahtjevClanstva(korisnik: ObicniKorisnik): Obavjestenje
+dodajKnjigu(naslov: String, autor: String, zanr: Zanr, opis: String): void
+izbrisiKnjigu(knjiga: Knjiga): void
+modifikujKnjigu(knjiga: Knjiga): void
+izbrisiKorisnika(id: int): void
+posaljiObavjestenje(obavjestenje: Obavjestenje, primaoc: Osoba): void
+getKorisnici(): List< ObicniKorisnik>
+getBibliotekari(): List<Bibliotekar>
+getAdministrator(): Administrator
+getKnjige(): List<Knjiga>
+dodajKorisnika(korisnik: ObicniKorisnik): void
+dodajBibliotekara(bibliotekar: Bibliotekar): void
+setKnjige(knjige: List<Knjiga>): void
1
Osoba
-id: int
-ime: String
-prezime: String
-email: String
-lozinka: String
«create»+Osoba(id: int, ime: String, prezime: String, email: String, lozinka: String)
+getId(): int
+getIme(): String
+getPrezime(): String
+getEmail(): String
+getLozinka(): String
+setId(id: int): void
+setIme(ime: String): void
+setPrezime(prezime: String): void
+setEmail(email: String): void
+setLozinka(lozinka: String): void
0..*1
1
1..*
1
0..*
Knjiga
-id: int
-naslov: String
-autor: String
-zanrovi: List<Zanr>
-opis: String
-rating: int
-korisnikKojJePosudio: Korisnik
-recenzije: List<Recenzija>
«create»+Knjiga(id: int, naslov: String, autor: String, zanrovi: List<Zanr>, opis: String, rating: int, korisnik Korisnik, recenzije: List<Recenzija>)
+posudi(korisnik: Korisnik): void
+oslobodiPosudbu(korisnik: Korisnik): void
+ostaviRecenziju(recenzija: Recenzija): void
+obrisiRecenziju(recenzija: Recenzija): void
+getId(): int
+getNaslov(): String
+getAutor(): String
+getZanrovi(): List<Zanr>
+getOpis(): String
+getKorisnikKojJePosudio(): Korisnik
+getRecenzije(): List<Recenzija>
+setId(id: int): void
+setNaslov(naslov: String): void
+setAutor(autor: String): void
+setZanrovi(zanrovi: List<Zanr>): void
+setOpis(opis: String): void
+setRating(rating: int): void
+setKorisnikKojJePosudio(korisnik: Korisnik): void
+setRecenzije(recenzije: List<Recenzija>): void
+dajKnjigu(): Knjiga
Gost
-brojPosjeta: int
«create»+Gost(brojPosjeta: int)
+getBrojPosjeta(): int
+setBrojPosjeta(brojPosjeta: int): void

Bridge patern (nije implementiran)

U našem sistemu bridge patern bismo mogli iskoristiti ako uvedemo da metoda platiClanarinu iz klasa ObicnikKorisnik i VipKorisnik se implementira na sljedeci nacin: kod obicnog korisnika clanarina iznosi 20 KM, a kod vip korisnika se na tu istu cijenu sabere dodatak. Dodali bismo interfejs IClanarina koji bi imao metodu platiClanarinu i njega bi nasljeđivali ObicniKorisnik i VipKorisnik, te bismo dodali klasu Bridge koja bi imala atribut IClanarina i metodu platiClanarinu.

Ovim paternom se omogućuje nadogradnja modela u budućnosti npr sa novom klasom GodisnjiKorisnik čija bi se članarina računala kao osnovica*12 - popust, tako da ovim paternom održavamo O princip.

Proxy patern (nije implementiran)

U našem sistemu proxy patern bismo mogli iskoristiti sa klasama ObičniKorisnik i VipKorisnik ukoliko uvedemo funkcionalnost da Vip korisnik može pogledati koji je u listi čekanja za rezervaciju određene knjige, dok ta funkcionalnost nije omogućena običnom korisniku. Ovo bismo uradili tako što definišemo klasu Proxy koja je povezana sa klasom ObičniKorisnik i nasljeđuje interfejs IRedniBrojUListi i njegove metode, te ima atribut tipa boolean pravoPregleda i atribut List<Knjiga> kojeg bismo prebacili iz klase Biblioteka kojoj se moze pristupiti samo ako je pravoPristupa true, inace se vraca null. Pored toga definišemo i interfejs IRedniBrojUListi koji bi bio povezan sa klasom Proxy i klasom Knjiga u koju bismo ubacili atribut tipa lista<Osoba> u kojoj bi držali listu osoba koje čekaju da iznajme tu knjigu sortiranu po datumu predaje zahtjeva i taj interfejs bi imao metodu dajRedniBrojUListi. Ukoliko je tip Korisnika koji traži pristup svom rednom broju u listu Vip on će dobiti svoj redni broj, a ukoliko je u pitanju obični korisnik bit će mu odbijen zahtjev.

Ovim paternom bi se omogucila kontrola pristupa informacijama koje su delikatnijeg tipa.

Decorator patern (nije implementiran)

U našem sistemu decorator patern bismo mogli iskoristiti ako uvedemo da se u klasi Knjiga samo čuva informacija o njenom imenu, a da Administrator može naknadno dodavati informacije o njoj npr. Datum izdavanja, ime autora, izdavačka kuća, slika, broj stanica itd. Ovaj patern ćemo implementirati na sljedeći način: deifinišemo interfejs IKnjiga koji će sadržavati metode dodajInformacije() i dajKnjigu() i dodaćemo klase ObicnaKnjiga, KnjigaSaSlikom, KnjigaSaAutorom ...(koliko već hoćemo detalja) koje će kao atribut imati objekat tipa IKnjiga. ObicnaKnjiga ima samo informacije o nazivu knjige, dok ostale klase imaju metode dodajInformaciju() kojim se nadograđuju informacije o knjizi. Samo klasa ObicnaKnjiga kao atribut ima objekat tipa Knjiga, a sve ostale knjige imaju kao atribut IKnjiga. Ovime se osigurava da se ne može napraviti knjiga sa slikom, a da se prethodno nije napravila obična knjiga.

Ovaj patern nam pomaže da jednostavno nadograđujemo i rukujemo odabranom klasom.

Kreacijski paterni

Singleton patern (implementiran)

U sistemu, klasa koja je realizovana kao singleton je klasa Administrator.

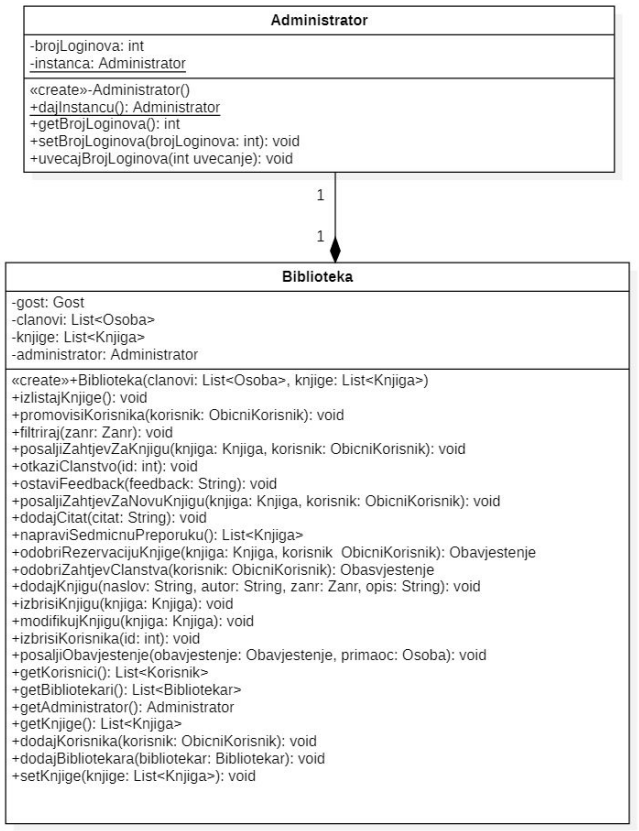
Izmjena na dijagramu klasa

U klasi Administrator konstruktoru je postavljen nivo pristupa na private, dodan je static atribut instanca tipa Administrator kao i static metoda dajInstancu(): Administrator, što je sve shodno primjeni singleton paterna. Takođe u klasi biblioteka iz konstruktora je izbačen parametar za administratora, s obzirom da će se atribut administrator moći inicijalizirati pozivom metode dajInstancu() klase Administrator.

Svrha korištenja paterna

Kako primjenom ovog paterna izbjegavamo instanciranje više istih objekata, dobivena je ušteda memorije, a dobijamo i prirodniji pogled na sistem.

Slika sa izmjenama slijedi ispod.

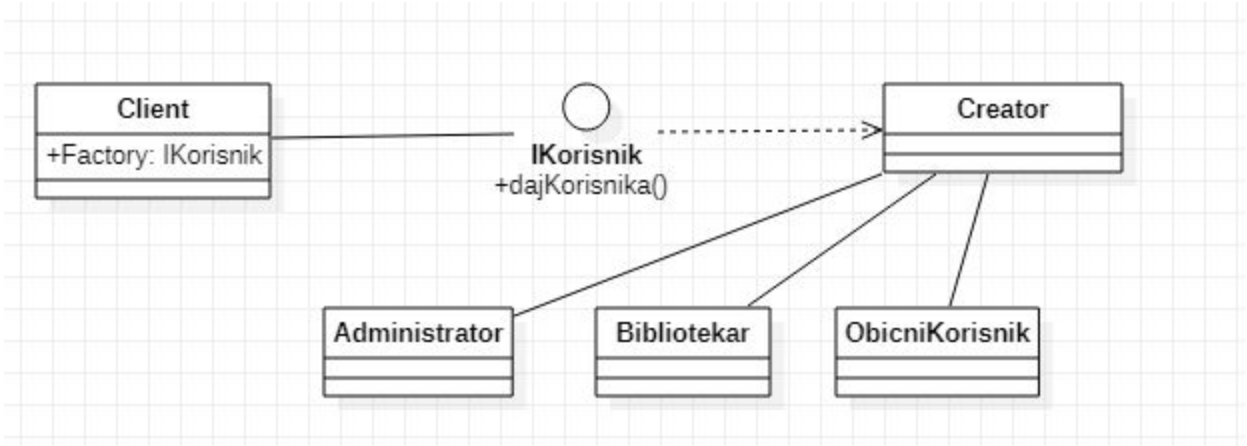


Factory method pattern (implementiran)

Factory method pattern je u nasem sistemu iskoristen kod klasa koje su izvedene iz klase Osoba. Obzirom da su klase Administrator, Bibliotekar i ObicniKorisnik izvedene iz osnovne klase Osoba, kreiranje instanci ove klase mozemo prebaciti na jedno mjesto u programu, tako da je moguce lakse podrzavanje koda, kao i lakse dodavanje novih tipova korisnika ukoliko to bude bilo potrebno u nekoj buducoj fazi razvoja aplikacije.

Izmjena na dijagramu klasa

Na nas dijagram klasa se dodaje interfejs koji smo nazvali IKorisnik. Zatim tu imamo vec postojece klase Administrator, Bibliotekar i ObicniKorisnik koje implementiraju taj interfejs. Dodali smo takodjer i Creator klasu koja posjeduje FactoryMethod() metodu. FactoryMethod() metoda odlucuje koju klasu instancirati.Takodjer je dodana i klasa Client koja prikazuje zahtjev klijenta, tj onoga ko ce koristiti taj interfejs IKorisnik. Taj dio dijagrama klase je dat na slici ispod:



Svrha korištenja paterna

Ovaj patern enkapsulira kreaciju objekata sto ga cini laksim za promijeniti poslije ukoliko se promijeni nacin na koji se objekti kreiraju ili se mogu dodati novi objekti pri cemu se promjena desava u samo jednoj klasi.

Abstract factory pattern (nije implementiran)

Ovaj patern se koristi pri radu sa familijama slicnih produkata, te ukoliko postoji više tipova istih objekata te različite klase koriste različite podtipove. Obzirom da mi u nasem sistemu ne posjedujemo familije slicnih produkata, jer ne postoji nikakva ponuda koja vodi u tom smislu, ovaj patern nije moguce primijeniti. Jedan od mogucih nacina, koji smo zamislili, na koji bi se mogao primijeniti ovaj patern jeste kada bi htjeli da u zavisnosti od nekog datuma, ili odredjenog dogadjaja u isto vrijeme “uparimo” citat dana i preporuku za taj dan. Naprimjer, mogli bismo napraviti da preporuka i citat dana budu neki bosanskohercegovacki, ili strani, ili pisci odredjenog pravca, u ovisnosti od tadasnjeg datuma ili nekog dogajdaja. Tada bismo mogli dodati klase kao sto su: BhPreporuka ili KlasiciPreporuka ili HororiPreporuka ili MesaSelimovicPreporuka ili tome slicno, gdje bi naprimjer za 1. mart (dan Nezavisnosti BiH) za citat dana i preporuku literature bili ponudjeni iskljucivo bosanskohercogovacki pisci i slicno,

gdje bi za rodjendan nekog poznatog pisca bili ponudjena njegova djela i slicno. Za to bi nam bila potrebna klasa koja bi se zvala PreporukaICitat koja bi imala metode : +dodajCitat() i metodu +dodajKnjigu, a iz te klase bi mogle biti naslijedjene vec spomenute klase (BhPreporuka, KlasiciPreporuka, HororiPreporuka,…) koje bi imale iste metode i povezivale bi te knjige sa tim dogadjajem. Na taj nacin bi ta jedna preporuka predstavljala familiju koja zavisi od nacionalnosti, pravca ili neceg drugog, te bi time bio ispunjen ovaj design pattern. Naravno potrebno bi bilo dodati neke klase (kao naprimjer klasu autor koja bi bila jedan od atributa u klasi Knjiga, umjesto sadasnjeg Stringa koji predstavlja autora) te odredjene atribute i metode u druge klase.

Implementacija

Treba nam klasa BHFabrika koja ima atribut BHCitat i BHPreporuka, i tako za sve strane kombinacije (svaka strana kombinacija ima svoju zasebnu Factory klasu). Sve Factory klase nasljeđuju apstraktnu klasu Factory. Biblioteka bi imala List<Factory> fabrike koje bi omogućavale kreiranje novih preporuka i citata u listama. Svaka fabrika bi imala atribute tipa izvedenih klasa onog tipa koji je potreban za korištenje, kojima bi se pristupalo po potrebi.

Prototype pattern (nije implementiran)

Ovim paternom se omogucava pojednostavljenje procesa kreiranja novih instanci, narocito kada objekti sadrze veliki broj atributa koji su za vecinu instanci isti. U nasem sistemu, bilo da govorimo o klasi Osoba ili klasama koje su iz nje izvedene, bilo da govorimo o klasi Knjiga, svaka instanca ima razlicite atribute (u slucaju Osoba to su: ime, prezime, lozinka i slicno, a u slucaju Knjiga to su: naslov, autor, zanr i slicno), gdje nema smisla klonirati ili kopirati te instance, tako da se ovaj patern ne moze dodati u nas sistem. Vecina atributa su licne stvari koje se vezu za svaku posebnu instancu te ih nema potrebe klonirati, a nema ni smisla.

Kako bi se mogao dodati patern

Jedan od slucajeva u kojem bi bilo smisleno upotrijebiti ovaj patern jeste slucaj kada bi zeljeli da podrzimo izdavanje istih knjiga na vise razlicitih jezika. Naprimjer, ukoliko zelimo dodati knjigu na engleskom jeziku, koja vec postoji u Biblioteci na bosanskom jeziku, bilo bi potrebno napraviti novu instancu klase, koja bi imala sve iste podatke kao i knjiga na bosanskom jeziku, ali bi trebala imati i atribut koji bi govorio da je ta knjiga na drugom jeziku. Tada bi imalo smisla klonirati tu knjigu i naknadno joj promijeniti taj atribut koji govori na kojem je jeziku novokreirana knjiga.

Implementacija

Ovaj patern bi dodali na nacin sto bi klasa Knjiga imala metodu “clone” (naslijedjenu od IPrototype interfejsa) koja bi se zatim pozvala u klasi Biblioteka u metodi “dodajIzdanje” a onda bi se samo promijenio atribut jezika.

Builder pattern (nije implementiran)

Builder patern služi za apstrakciju procesa konstrukcije objekta, kako bi se kao rezultat mogle dobiti različite specifikacije objekta koristeći isti proces konstrukcije. Obzirom da u nasem sistemu za sve klase vrijedi da imaju iste specifikacije (npr. prilikom kreiranja objekta klase Knjiga, svi parametri se moraju koristiti, ne postoje slucajevi kada se autor ne mora unijeti, ili

naslov, ili bilo koji drugi parametar konstruktora klase Knjiga. Ovo sto je receno za klasu Knjiga vrijedi i za sve druge klase naseg sistema), ovaj patern se ne moze koristiti.

Kako bi se mogao dodati patern

Ukoliko bi bila uvedena klasa Autor, koja bi sadrzavala osnovne podatke za jednog autora (ime, prezime, knjige i slicno), onda bi prilikom kreacije objekta klase Knjiga, umjesto stringa za naziv autora knjige, bilo potrebno u konstruktor poslati instancu klase Autor, koja bi predstavljala pisca te knjige. Obzirom da neke knjige mogu imati dva ili cak i vise autora (narocito naucne knjige) onda bi ponekad bilo potrebno poslati vise od jednog autora u konstruktor, a u vecini slucajeva (za romane ili slicne knjige koje imaju samo jednog pisca) bi ta dodatna polja ostala prazna. Ukoliko bi pored toga neka knjiga mogla pripadati vise zanrova ili ukoliko neka knjiga moze imati i vise naslova po kojim se moze raspoznati (sto se rijetko desava ali nije nemoguce) onda bi svi ti slucajevi bili izuzetno pogodni za koristenje Builder patterna koji bi unutar interfejsa mogao imati metode kao sto su:

+dodajPisce(Autor prvi, Autor drugi): void
+dodajZanrove(Zanr prvi, Zanr drugi): void
+dodajNaslove(String prvi, String drugi): void
+dajKnjigu():

Ili nesto slicno tome, ovaj patern bi tada pomogao pri kreiranju drugacijih specifikacija istog produkta (u ovom slucaju knjige).

Implementacija

Navedene metode bile bi dio IBuilder interfejsa, a zatim bismo napravili builder klase: BuilderReprint, BuilderPenguinBooks, BuilderNaucniRad i druge vrste buildera. Oni bi u sebi imali atribut Knjiga i pri kreiranju nove knjige kombinovali bi razne pozive ovih metoda - neki bi pozvali samo određene, neki bi automatski dali vrijednosti za određene dijelove, itd itd. Sve ovo bi naravno bilo pozvano pri dodavanju nove knjige u klasi Biblioteka.