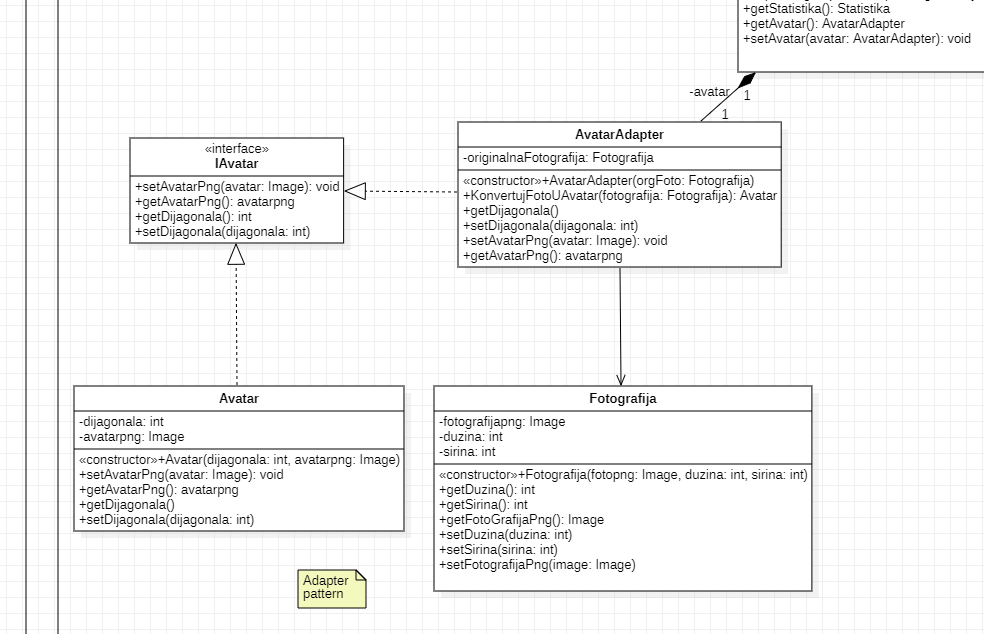
Strukturalni paterni

**Adapter patern**

Adapter pattern predstavlja strukturalni dizajn pattern koji omogućava objektima nekompatibilnih interfejsa da međusobno surađuju. Najprije kreiramo objekat klase koja predstavlja naš adapter. Njegov zadatak je preobražavanje interfejsa jednog objekta u formu razumljivu drugom objektu sistema, ali tako da je kompleksnost preobrazbe prikrivena. Generalno, strukturu adapter paterna čine četiri komponente:

* Client – klasa koja veže klijentski kod za podsistem unutar okvira paterna
* Client Interface – ovaj interfejs sadrži metode koje vanjske klase moraju implementirati kako bi komunicirale sa klijent kodom
* Service – klasa čije se usluge žele koristiti, bilo da je klasa unutar ili van sistema
* Adapter – osnovna klasa na kojoj patern počiva. U mogućnosti je da implemetnira Client Interface kao posrednik između Client koda i samog servisa. Najprije Adapter klasa omota objekat servisne klase, i to tako da ga proglasi svojim atributom. Nakon toga, realizacijom klijent interfejsa, adapter kontroliše zahtjeve na način da ih šalje u formatu razumljivom servisu.

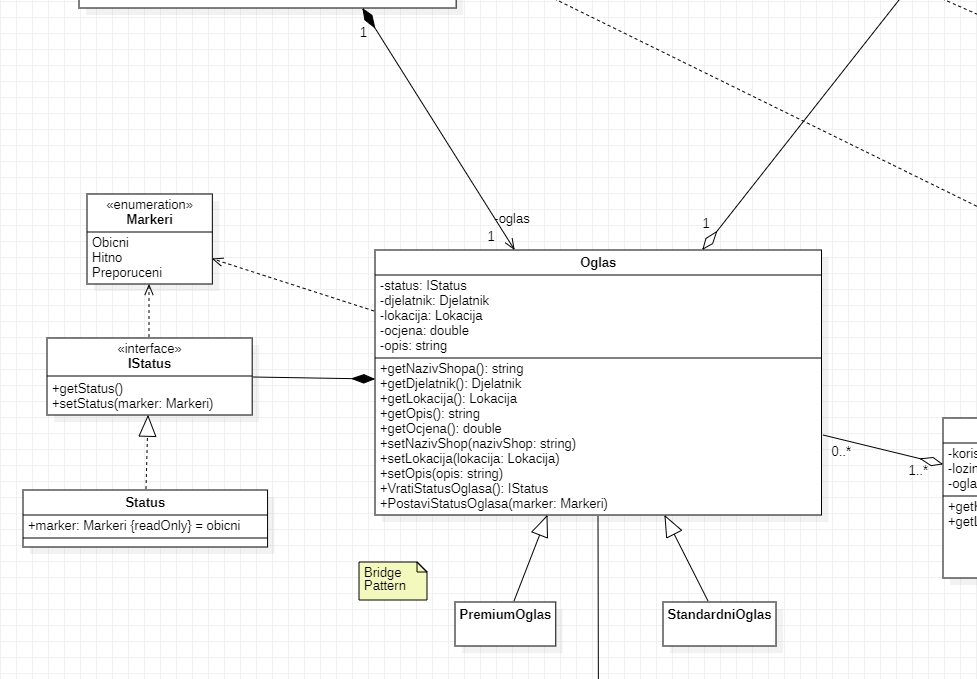
Prednost ovog paterna je što promjene ne utiču na klijentski kod. U koliko dođe do promjena unutar servisne klase, ili se pojavi potreba za dodavanjem nove servisne klase, dovoljno je dodati novi adapter koji implementira klijent interfejs.



**Bridge patern**

Bridge pattern predstavlja strukturalni patern koji nam omogućava da skup logički uskopovezanih klasa, ili kompleksnu klasu karakterističnu po većem broju logički različitih osobina, rastavimo u dva hijerarhijska stabla – Stablo apstrakcije, te Stablo implementacije koje ga koristi. Najčešće se koristi u sistemima gdje se implementira princip nasljeđivanja nad klasama kod kojih funkcionalnost objekata ovisi o samoj kombinaciji vrijednosti atributa pomenute bazne klase. Koristi se i nad klasama čija su buduća logička proširenja međusobno isključiva. Generalno, strukturu Bridge patterna čini pet različitih komponenti, s tim što je jedna od njih opcionalna. To su:

* Client – klasa koja veže klijentski kod za podsistem unutar okvira paterna
* Abstraction – apstraktna klasa koja sadrži viši kontrolnu logiku, obično predstavlja primarnu osobinu prema kojoj se naslijeđeni objekti razlikuju. Atribute čine svaki od interfejsa, koje će kasnije realizovati konkretne implementacije. Neformalno rečeno, svaki interfejs realizuje jednu od osobina objekata uglađenih apstrakcija, odnosno nasljeđujućih klasa
* Implementation – zajednički interfejs za sve konkretne realizacije. Unutar njega su izložene sve metode koje koristimo u vezi sa apstraktnom klasom.
* Concrete Implementaion – klasa koje sadrže sve atribute i metode koje jednu konkretnu implementaciju razlikuju od druge, npr. zvono na mobitelu i zvono za ulazna vrata
* Refined abstractions – takozvane uglažene apstrakcije, koje pružaju nove varijante kontrolne logike ili proširanja na postojuću. Ne predstavljaju neophodan dio Bridge paterna

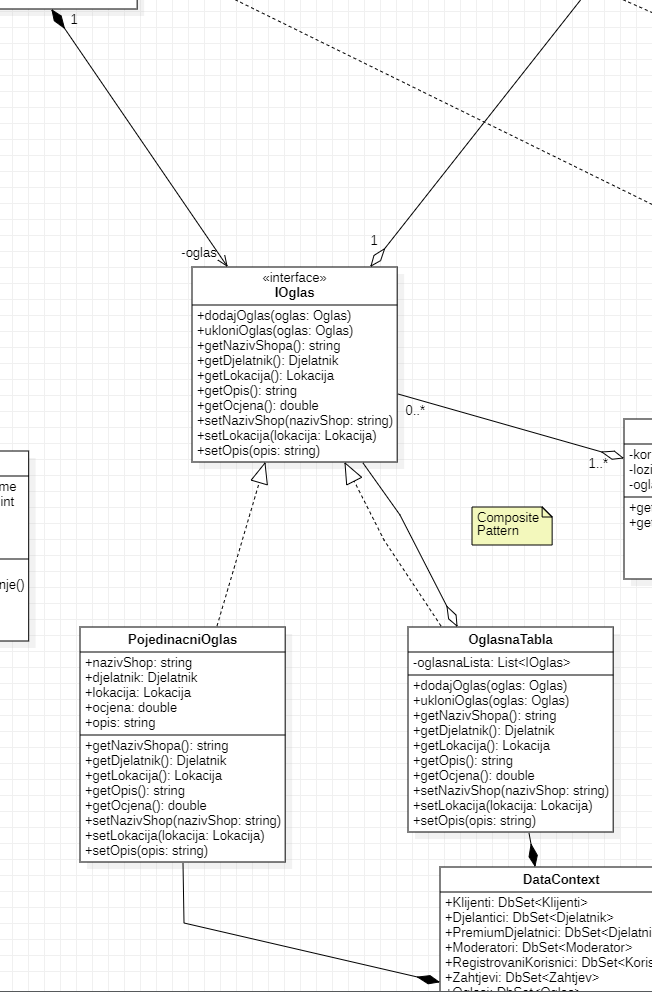


Pošto i moderator i djelatnik imaju mogućnost brisanja oglasa (oba implementiraju interfejs IObrisiOglas) napravili smo ObrisiOglas klasu i implementirali bridge patern. Oba brisanja imaju isti dio tj. brisanje iz baze koji je obuhvaćen bridge paternom dok se moderatoru pruža mogućnost prijave korisnika administratoru prilikom brisanja oglasa.

**Composite patern**

Composite pattern je patern koji omogućava da sistem klasa spojimo u jedno hijerarhijsko stablo i pristupamo klasama tako da stablo posmatramo kao individualne objekte. Koristimo ga u sistemima kod kojih su i sami atributi klasa kompleksne strukture – nove grane sa svojim kompleksnim atributima i metodama. Ukoliko bismo zahtjevali pristupom svakom od elemenata sistema, on bi praktično bio neizvodiv i nemoguć. Uz pomoć Composite paterna, svi elementi se tretiraju jednako. S obzirom da će svi koristiti isti interfejs, ne moramo brinuti o konkretnim objektima stabla. Princip se svodi na prosljeđivanje zahtjeva, a formiranje rezultat, odnosno odgovora, se svodi na recipročno sumiranje kroz sve elemente stabla, bilo da se radi o jednostavnim ili kompleksnim objektima. Generalno, strukturu Composite paterna čine četiri komponente:

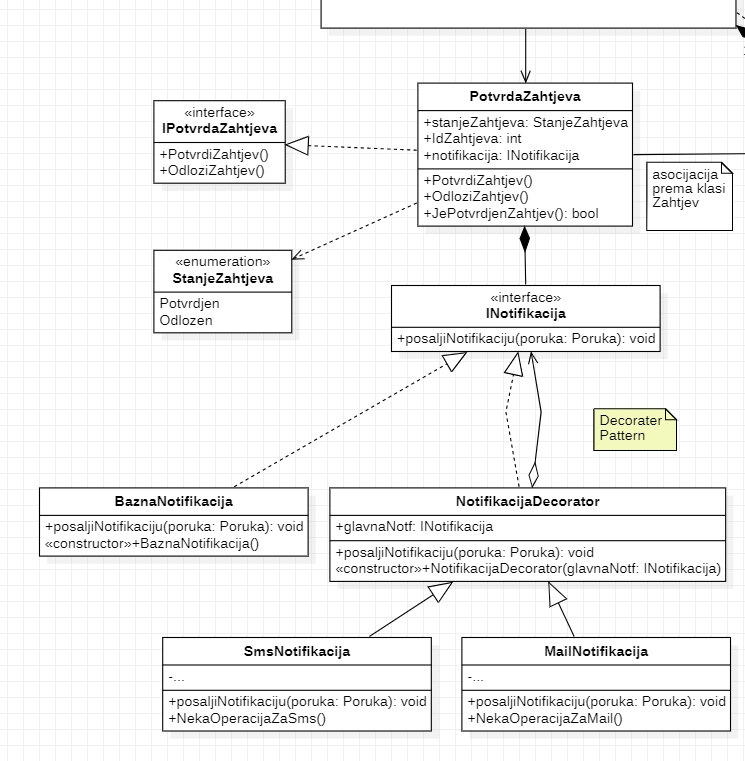
* Component – interfejs koji opisuje metode koje su zajedničke i za jednostavne i za kompleksne objekte, te veže klijentski kod za njih
* Leaf – jednostavni objekti; krajevi/listovi stabla. Radi se o konkretnim klasama koje ispunjavaju zahtjev. Sadrže primitivne tipove atributa koji opisuju sami objekat
* Composite – element koji se sastoji od leaf klasa i drugih kontejnerskih atributa. Sa stanovišta kontejnerskih atributa, nije potrebno poznavanje tipa elemenata kontejnera. Upotrebom Component interfejsa, elementi računaju međurezultat koji se na samom kraju šalje kao zbirni rezultat Client kodu
* Client – klasa koja veže klijentski kod za podsistem unutar okvira paterna. Korištenjem interfejsa uniformno radi kako sa jednostavnim, tako i sa kompleksnim objektima



**Decorator patern**

Decorator pattern je patern koji nam omogućava da dodamo nova ili ograničimo postojuće funkcionalnosti objekta klase, tako što ga smijestimo unutar specijalnog omotačkog objekta wrapper klase. Umjesto nasljeđivanja, patern koristi agregaciju ili kompoziciju. Umjesto da glavni objekat nasljeđuje ponašanje, kreira se objekat koji će sadržati referencu na glavni objekat i izvršavati traženu funkciju za njega. Wrapper sadrži iste metode kao i glavni omotani objekat i šaljemo sve zahtjeve koje primi, ali je u stanju da utiče na rezultat izvršavanjem dodatnih metoda prije ili nakon prosljeđivanja pomenutih zahtjeva. Neophodno je da omotač nasljeđuje isti interfejs kao i glavni objekat. U slučaju višestrukog omotavanja, posljednji sloj bi predstavljao objekat sa kojim klijent kod radi. Generalno, strukturu Decorator paterna čini pet komponenti:

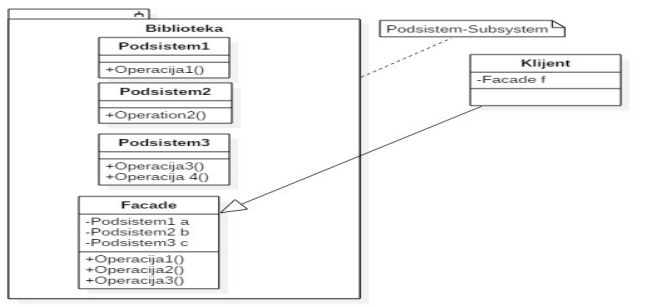
* Component – interfejs koji opisuje metode koje su zajedničke i za omotače i za omotane objekte
* Concrete Component – klase koje predstavljaju konkretne objekte. To su objekti nad kojima ćemo vršiti omotavanje tj- proširivanje ili ograničavanje funkcionalnosti
* Base Decorater – bazna klasa za kreiranje dekoratora. Tip klase je component interfejs kako bi se iz nje mogli naslijediti kako konkretni objekti, tako i dekoratori. Zadužen je za prosljeđivanje zahtjeva od konkretnih dekoratora do konkretnih komponenti na koje upućuju.
* Concrete Decorators – konkretni dekoratori, dinamički definišu ponašanje omotanog objekta.
* Client – klasa koja veže klijentski kod za podsistem unutar okvira paterna. Korištenjem interfejsa uniformno radi kako sa jednostavnim, tako i sa kompleksnim objektima. Izvršavaju override nad metodama baznih dekoratora, kao i sve dodatne funkcionalnosti prije ili nakon pozivanja metoda baznog dekoratora

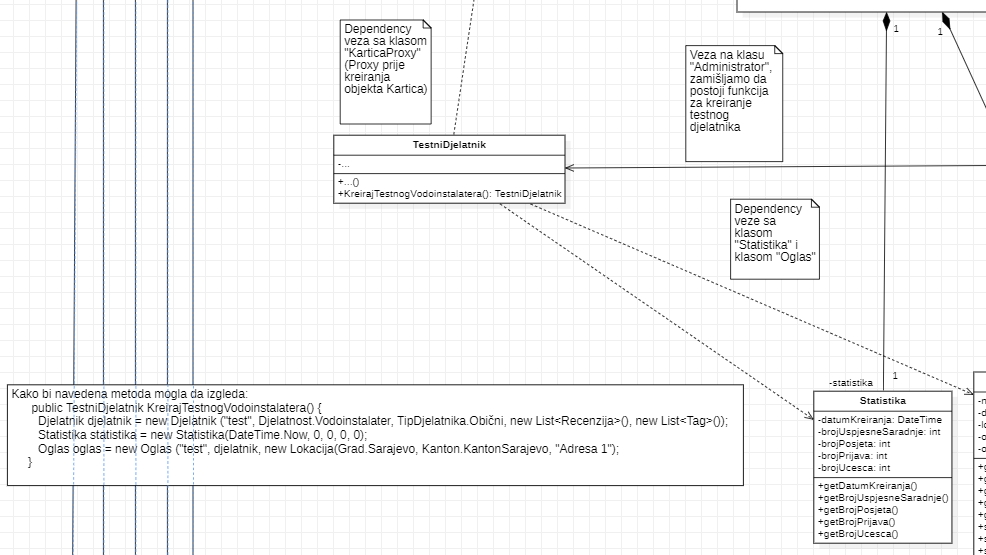


**Facade patern**

Facade patern se koristi kada sistem ima više identificiranih podsistema (subsystems) pri čemu su apstrakcije i implementacije podsistema usko povezane. Facade patern je klasa koja omogućava jednostavni interfejs za kompleksni podsistem koji sadrži puno „moving parts“. Facade patern ne omogućava sve funkcionalnosti za razliku od podsistema kada bi radili direktno sa tim podsistemima. Međutim, facade patern uključuje one funkcionalnosti koje zanimaju klijenta koda. Postojanje facade paterna je jako korisno kada želimo integrirati našu aplikaciju sa velikim brojem biblioteka koje imaju nekoliko desetina opcija, a mi želimo iskoristiti samo neke njene funkcionalnosti.

Strukturu Facade paterna čine sljedeće klase: Facade – definira i implementira jedinstven interfejs za skup operacija nekog podsistema, SubsystemClassN – definira i implementira N-ti interfejs u skupu interfejsa nekog podsistema (N = 1, 2, 3, ...). Struktura Facade paterna je data na slici:

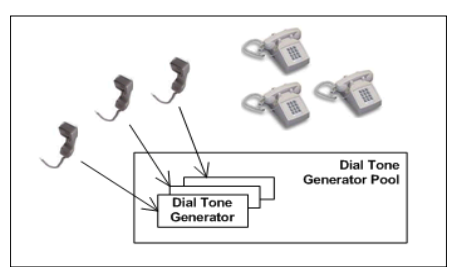


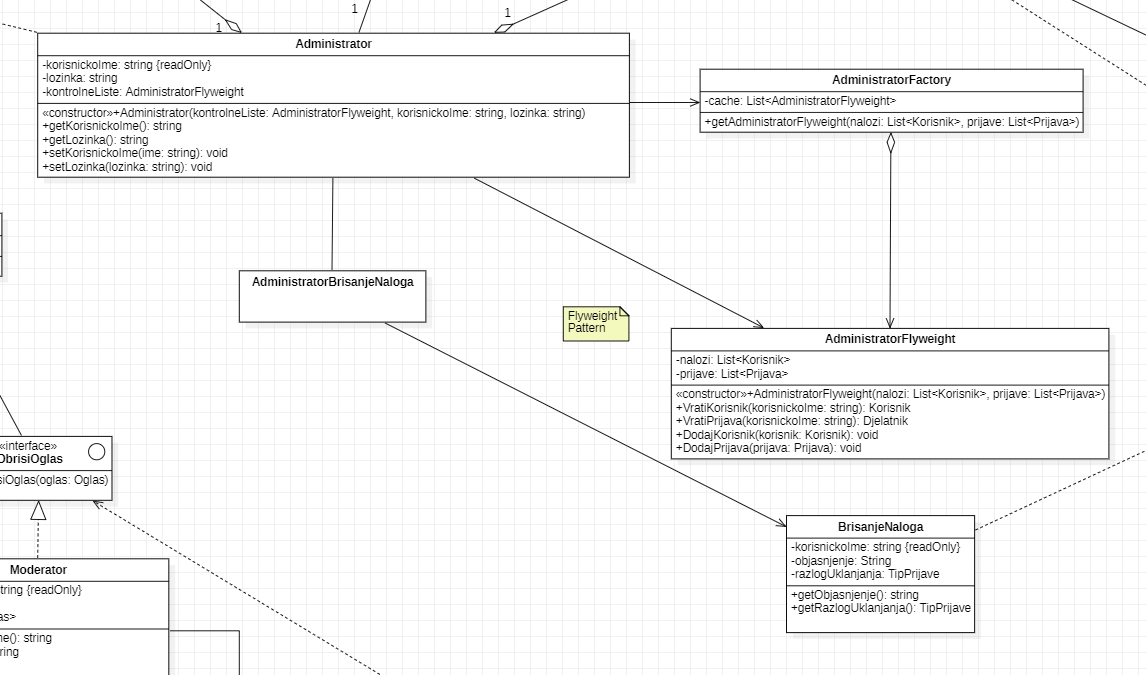


**Flyweight patern**

Postoje situacije u kojima je potrebno da se omogući razlikovanje dijela klase koji je uvijek isti za sve određene objekte te klase (tzv. glavno stanje (engl. intrinsic state)) od dijela klase koji nije uvijek isti za sve određene objekte te klase (tzv. sporedno stanje (engl. extrinsic state)). Osnovna namjena Flyweight paterna je upravo da se omogući da više različitih objekata dijele isto glavno stanje, a imaju različito sporedno stanje.

Flyweight je strukturalni patern koji omogućava pohranu objekata u raspoloživu količinu RAM memorije tako što dijeli zajedničke dijelove stanja između više objekata umjesto da skladišti isti broj podataka za svaki objekat.

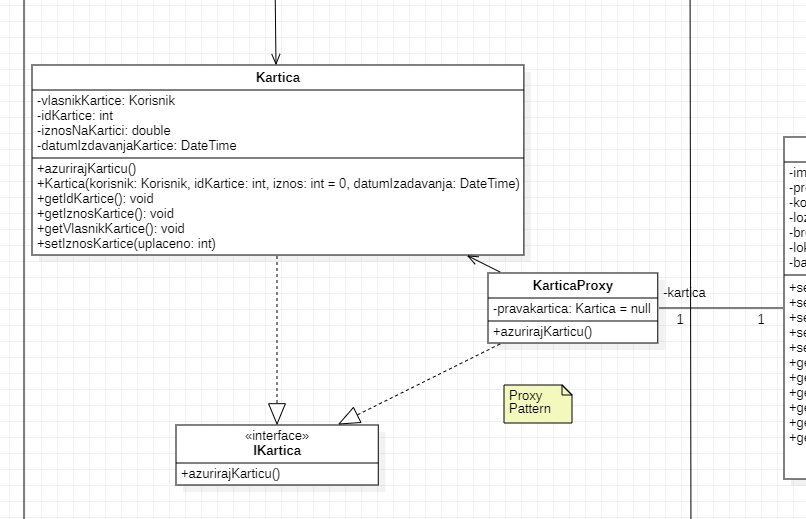




**Proxy patern**

Namjena Proxy paterna je da omogući pristup i kontrolu pristupa stvarnim objektima. Proxy je obično mali javni surogat objekat koji predstavlja kompleksni objekat čija aktivizacija se postiže na osnovu postavljenih pravila. Proxy je strukturalni patern koji omogućava čuvar mjesta(placeholder) za drugi objekt. Proxy kontrolira pristup originalnom objektu, dozvoljavajući vam da izvršite nešto prije ili nakon što zahtjev prođe do originalnog objekta. Proxy obrazac preporučuje da kreiramo novu proxy klasu sa istim interfejsom kao originalni objekt usluge. Zatim ažurirate svoju aplikaciju tako da prosljeđuje proxy objekt svim klijentima originalnog objekta. Nakon što primi zahtjev od klijenta, proxy kreira pravi objekt usluge i proslijeđuje sav posao tom objektu.

Ali koja je svrha proxy paterna? Ako trebate izvršiti nešto prije ili poslije osnovne logike klase, proxy vam dozvoljava da to učinite bez promjene te klase. Budući da proxy implementira isti interfejs kao originalna klasa, može se proslijediti svakom klijentu koji očekuje pravi objekat usluge. Struktura Proxy paterna je sastavljena od klasa: Subject (ISubject) zajednički interfejs za realne/stvarne subjekte i proksije-surogate (proxies) koji omogućava da se oni koriste naizmjenično. RealSubject -je glavni objekat kojeg ‘predstavlja’ proxy. Proxy- implementira isti interfejs kao RealSubject tako da se Proxy može koristiti umjesto RealSubject objekta. Proxy vrši kontrolu pristupa RealSubject objektu, može kreirati i brisati taj objekat.



U našem sistemu koristimo proxy patern da bi smo kontrolisali konstrukciju objekta klase Kartica. Klasu KarticaProxy služi kako bismo odložili kreiranje kartice sve dok nam ona zaista ne bude potrebna tj. Dok se Klijent ne odluči na bezgotovinsko plaćanje, pod uslovom da i djelatnik ima svoju Karticu. Interfejs IKartica nam omogućava da poistovjećujemo objekat klase KarticaProxy sa objektom klase Kartica.