

# STRUKTURALNI PATERNI

## Adapter pattern

Osnovna namjena Adapter paterna je da omogući širu upotrebu već postojećih klasa. U situacijama kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu koristi se Adapter patern. Adapter patern kreira novu adapter klasu koja služi kao posrednik između originalne klase i željenog interfejsa.

Adapter patern bi mogli iskoristiti kada bi trenutni tip atributa Slika koji je sada string zamijenili tipom JPG ili PNG. Tada bi uz pomoć metoda konverzija Adapter klase omogućili prihvatanje raznih formata slike.

## Facade pattern

Kako mu i sam naziv govori, ovaj patern se koristi s ciljem da osigura više pogleda visokog nivoa na podsisteme (implementacija podсистема skrivena od korisnika).

Facade patern bi mogli iskoristiti kada bi kreirali novu klasu PutovanjeFacade koja bi sadržavala samo attribute id Mačke, ime, vjerojatnoću odabira i id putovanja kako bi smanjili kompleksnost rada sistema pri odabiru mačaka za putovanje.

## Decorator pattern

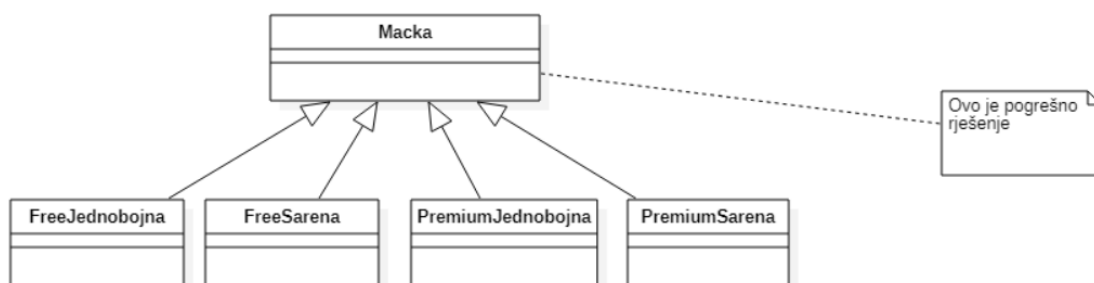
Osnovna namjena Decorator paterna je da omogući dinamičko dodavanje novih elemenata i ponašanja (funkcionalnosti) postojećim objektima. Objekat pri tome ne zna da je urađena dekoracija što je veoma korisno za ponovnu upotrebu komponenti softverskog sistema.

U našoj aplikaciji bi mogli dodati mogućnost editovanja slika kao što su rotacija, rezanje i dodavanje filtera koji postavlja astronautsku kacigu na profilnu sliku mačke.

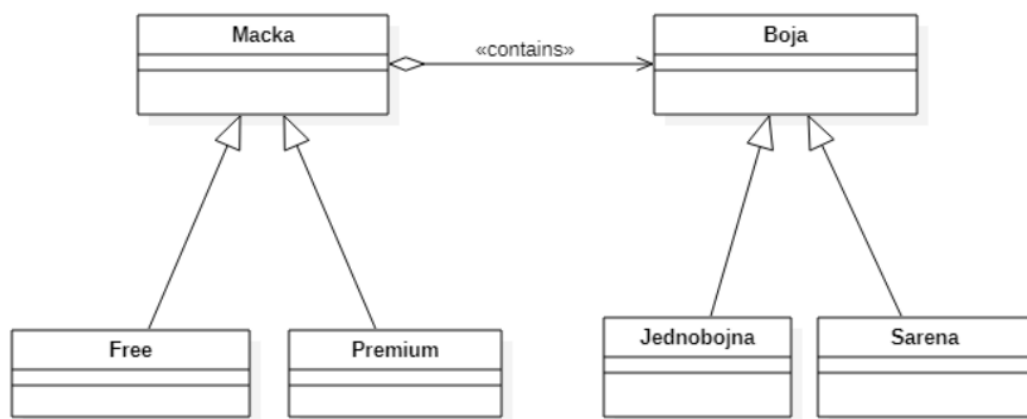
## Bridge pattern

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.

Ako u hijerarhiji klasa želimo dodatno razlikovati macke po bojama na jednobojne i šarene, pogrešno rješenje je imati 4 klase: FreeJednobojna, FreeSarena, PremiumJednobojna i PremiumSarena.



Bridge pattern predlaže da unutar klase **Macka** imamo atribut nove klase **Boja**.



Ovaj pattern omogućava da podijelimo monolitnu klasu u nekoliko hijerarhija klasa. Možemo mijenjati klase u svakoj hijerarhiji nezavisno od klase u ostalim. Ovaj pristup olakšava buduće održavanje koda.

# Composite pattern

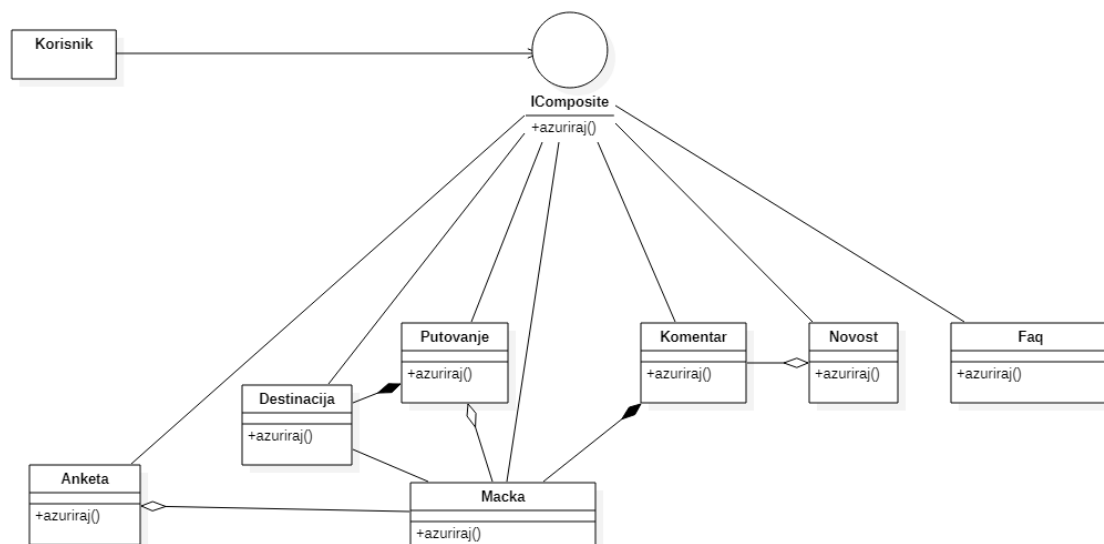
Composite pattern služi za kreiranje hijerarhije objekata. Koristi se kada svi objekti imaju različite implementacije nekih metoda, no potrebno im je svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija.

Composite pattern predlaže da kroz zajednički interfejs radimo sa svim objektima nad kojima vršimo slične akcije.

Umjesto kreiranja više metoda koje bi u bazi podataka zasebo ažurirale podatke za Faq, Macku, Novost i ostale klase, možemo napraviti interfejs IComposite sa metodom azuriraj().

Ta metoda se može koristiti nad objektima različitih kompleksnosti.

Da bi implementirali Composite pattern cijeli sistem trebamo predstaviti kao stablo. Ovaj uvjet je zadovoljen, jer sve klase u C# imaju zajedničku baznu klasu Object. Listovi stabla su klase koje ne mogu proslijediti naredbu na neku drugu klasu.



Prilikom pozivanja azuriraj() nad instancom klase Faq, Faq sam ažurira podatke u bazi, jer nema klase kojima bi proslijedio naredbu.

Novosti mogu sadržavati komentare.

Prilikom brisanja instance klase Novosti sa metodom azuriraj(), Novost neće ažurirati komentar, već će komentaru biti proslijeđena naredba da sam pozove metodu azuriraj().

## Proxy pattern

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

U našoj aplikaciji ovaj patern koristimo kod unosa lozinke prilikom login-a. Autentifikacijski proxy prije prosljeđivanja pristupa profilu prvo pita za password, te u slučaju dobro unešenog passworda odobrava pristup mogućnostima registrovanog korisnika, dok ga u suprotnom onemogućuje.

## Flyweight pattern

Flyweight patern se koristi 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 instance objekta (tzv. bezlično stanje).

Sve klase pretežno imaju različite vrijednosti atributa, pa samim tim ne možemo implementirati flyweight pattern.

Kada bi smo npr. u klasi Putovanje imali attribute za raketu, dronship za slijetanje, i količinu goriva koju raketa može da drži, ovi atributi bi uvijek imali istu vrijednost, jer SpaceCat koristi istu raketu za sve letove.

Uštedjeli bi na memoriji kada bi kreirali instancu nove klase Raketa i u njoj čuvali vrijednosti ovih atributa, za razliku od njihovog pohranjivanja u svakoj instanci klase Putovanje. Na ovaj način bi uštedjeli na memorijskim resursima.