STRUKTURALNI PATERNI

U naš sistem smo dodali dva strukturalna paterna: Adaptern patern i Proxy patern. Detaljno smo objasnili i slučajeve u kojima bi se mogli iskoristiti i ostali strukturalni paterni.

1. ADAPTER PATERN

Ukoliko želimo nadograditi naš sistem na način da tutoru omogućimo prikaz sortiranih predmeta po cijeni (dakle da pored metode getPredmetiSaCijenom(), koja vraća sve predmete koje drži tutor sa njihovom cijenom, imamo i metodu getPredmetiSortiranoPoCijeni(), koja vraća sve predmete sortirane po cijeni u rastućem poretku), možemo iskoristiti Adapter patern.

Zašto ovdje koristimo **Adapter patern**?

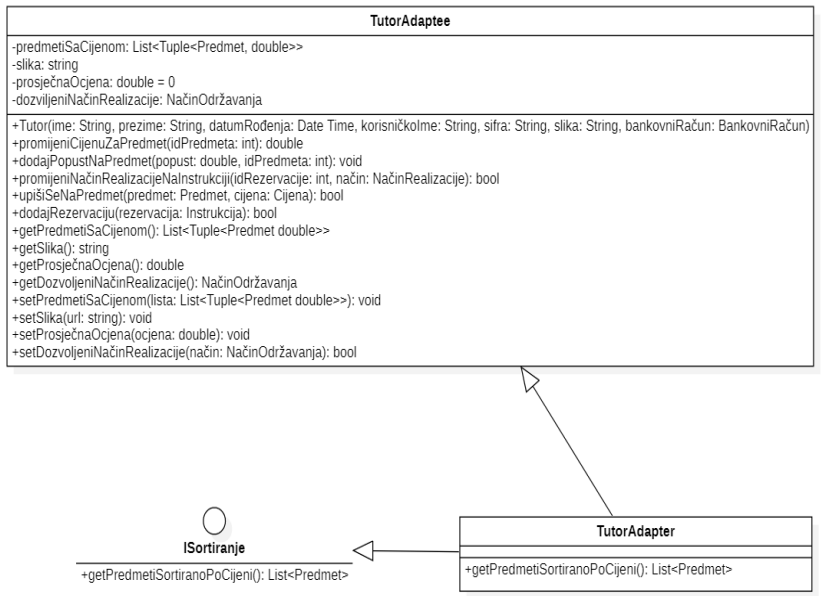
Adapter patern koristimo kada je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu. Tim postupkom se dobija željena funkcionalnost bez izmjena na originalnoj klasi.

U našem sistemu već postoji klasa Tutor koja ima metodu *getPredmetiSaCijenom()*. To je zapravo Adaptee klasa (stoga smo je preimenovali u *TutorAdaptee*), koju je potrebno adaptirati u cilju dostizanja željenog interfejsa (želimo promijeniti interfejs, te time nadograditi sistem). Dakle, kada korisnik želi pregled svih predmeta koje drži tutor, želimo da oni budu sortirani u rastućem redoslijedu po cijeni. Realizacija je postignuta na sljedeći način:

- Definišemo interfejs IPretraga sa metodom *getPredmetiSortiranoPoCijeni()*
- Definišemo klasu TutorAdapter, koja implementira interfejst IPretraga. U metodi *getPredmetiSortiranoPoCijeni()* pozivamo metodu TutorAdaptee klase *getPredmetiSaCijenom()* te prilagođavamo rezultat te metode u cilju unapređenja interfejsa.

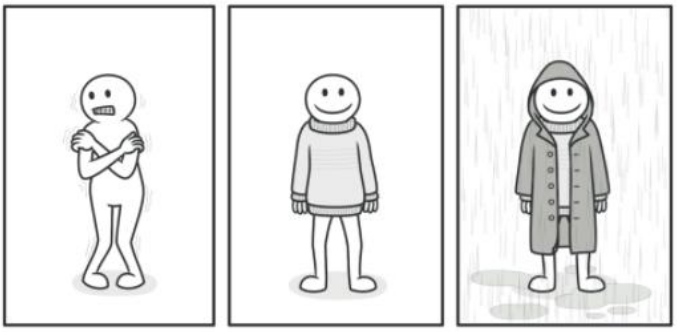
Jasno je da na ovaj način nismo vršili nikakve izmjene TutorAdaptee (odnosno Tutor) klase, ali smo unaprijedili njen interfejs. Naša aplikacija sada može koristiti interfejs, odnosno metodu *getPredmetiSortiranoPoCijeni()* implementira klasa TutorAdapter.

Na ovaj način možemo unaprijediti interfejs klase Tutor i sa drugim metodama, a da time ne modificiramo postojeću klasu Tutor. Ovim načinom nadogradnje aplikacije poštujemo OTVORENO ZATVOREN PRINCIP, budući da nadograđujemo entitete softvera bez njihove modifikacije.



2. DECORATOR PATERN

Decorator patern služi za omogućavanja različitih nadogradnji objektima koji svi u osnovi predstavljaju jednu vrstu objekta (odnosno, koji imaju istu osnovu). Umjesto da se definiše veliki broj izvedenih klasa, dovoljno je omogućiti različito dekoriranje objekata (tj. dodavanje različitih detalja), te se na taj način pojednostavljuje i rukovanje objektima klijentima, i samo implementiranje modela objekata.

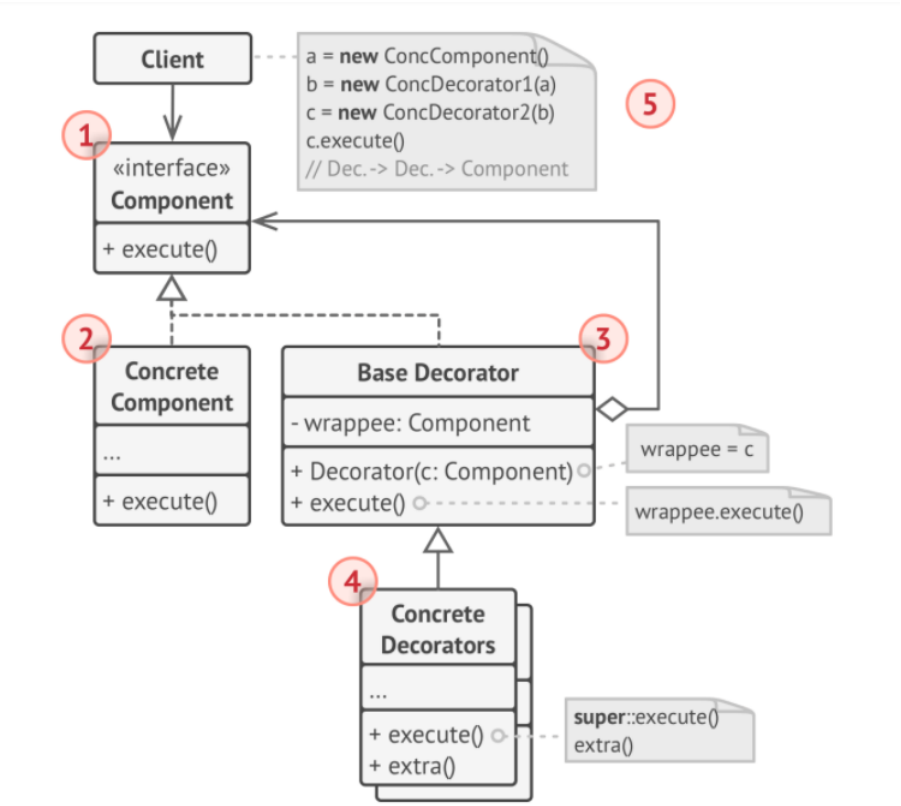


Bitno je naglasiti da je u pitanju *dinamičko* dodavanje novih elemenata i ponašanja postojećim objektima.

Decorator patern se koristi u slučajevima kada želimo dodijeliti dodatna ponašanja objektu tokom izvođenja programa bez da mijenjamo kod koji je na neki način u interakciji sa datim objektom.

Bitno je istaći razliku između Adapter i Decorator paterna. Adapter paternom doista postižemo mijenje interfejsa postojećeg objekta, dok Decorator paternom mijenjamo ponašanje objekta bez mijenjanja interfejsa.

Slikovito ćemo prikazati strukturu Decorator paterna, te prodiskutovati na koji način bi se on mogao uklopiti u naš sistem.



Pošto Tutor posjeduje atribut slika, moguće je u budućnosti da poželimo omogućiti korisniku manipulacije sa slikom (npr. rotacija, rezanje itd.). U tom slučaju je prvo potrebno u sistem dodati novu klasu (Slika), odnosno odvojiti aktivnosti vezane za sliku, od klase Tutor kako je bi bio narušen S SOLID princip (Princip pojedinačne odgovornosti). To u našem slučaju nije bilo potrebno, pošto sistem ne vrši nikakve manipulacije sa slikom. Slika je samo jedan od atributa koji opisuje Tutora (slično kao i email).

Potom je potrebno da napravimo interfejs (1), *ISlika*, koji idetificira klase objekata koje želimo dekorirati. Interfejs, između ostalog, treba sadržavati metodu edituj(), dok klasa Slika treba da implementira interfejs ISlika (2). Ta klasa je zapravo osnovna komponenta.

Definisanje baznog dekoratera možemo preskočiti, te odmah preći na realizaciju klasa koje predstavljaju konkretne dekoratore.

Svaki konkretan dekorator treba da sadrži kao atribut ISlika, potom treba da implementira interfejs ISlika, te u slučaju potrebe dodati nove metode (što je vidljivo na stavkama 3 i 4 prethodne slike).

Klase *SlikaRezanje*, *SlikaRotacija* bi mogle predstavljati konkretne dekoratore u sistemu.

3. BRIDGE PATERN

Ukoliko želimo uvesti različite implementacije neke funkcionalnosti, npr. dodavanjem popusta studentima ako im je rođendan taj dan te bi plaćanje u slučaju rođendana bilo implementirano drugačije. Tutor bi idalje dobio punu cijenu koju bi mu sistem nadoknadio.

Zašto ovdje koristimo **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.

U našem sistemu postoji klasa Instrukcija getCijena() koju bi mogli da proširimo interfejsom Bridge koja bi u sebi imala istoimenu metodu, te bi na osnovu datuma rođenja studenta pozivala dvije različite implementacije od kojih bi jedna vraćala normalno obračunatu cijenu a druga bi vraćala sniženu cijenu te od sistema zatražila nadoknadu na bankovni račun Tutora.

4. PROXY PATERN

Ukoliko želimo dodatnu sigurnost u sistemu, npr. pri ocjenjivanju tutora nakon završenog termina, idealan način da ostvarimo to je pomoću Proxy paterna.

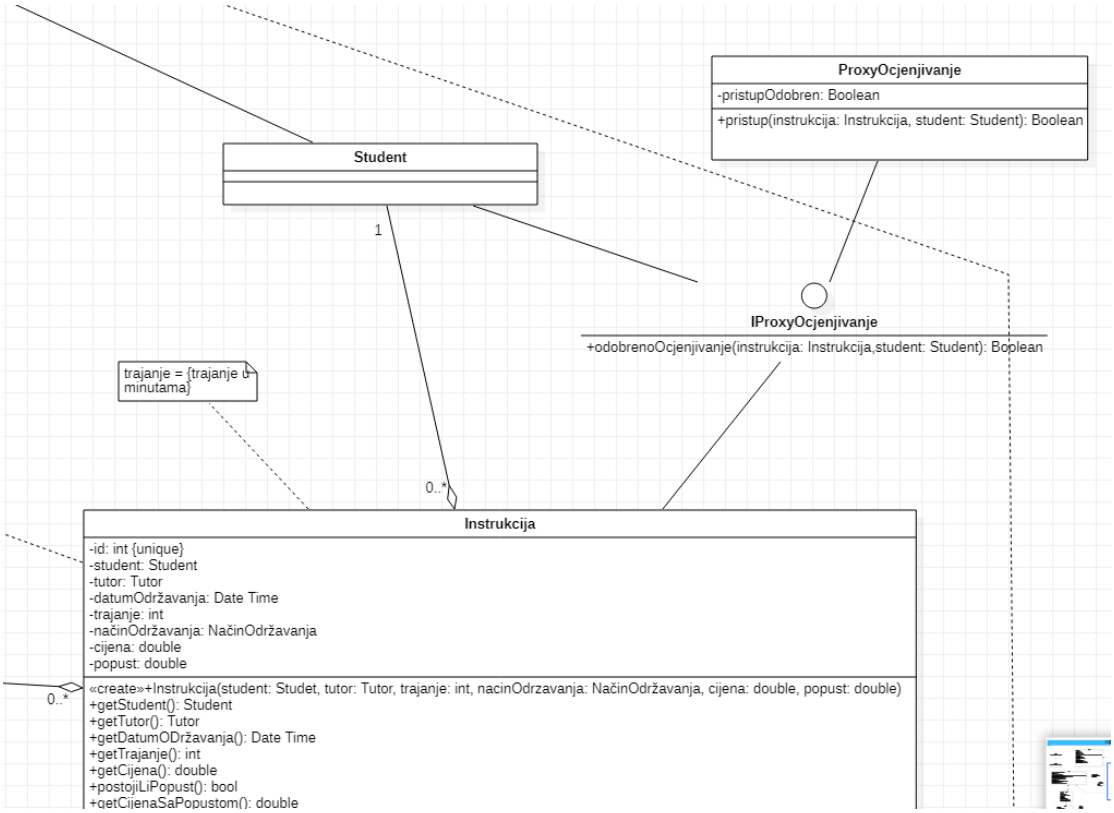
Zašto ovdje koristimo **Proxy patern**

Namjena Proxy paterna je da omogući pristup i kontrolu pristupa objektima ili metodama. Proxy je obično mali javni surogat objekat koji predstavlja kompleksni objekat čija aktivizacija se postiže na osnovu postavljenih pravila.

U našem sistemu postoji klasa Student i studenti će moći ocjenjivati Tutora nakon održanog termina(Instrukcije), taj proces ćemo osigurati sa Proxy paternom da bi osigurali da ocjenjivati mogu samo Studenti koji su zaista imali Instrukcije kod datog Tutora.

Da bi postigli to dodan je interfejs IProxyOcjenjivanje i klasa ProxyOcjenjivanje koja kroz metodu pristup(instrukcija: Instrukcija, student:Student) da vrši autentifikaciju Studenta koji želi da ocjenjuje i odobrava ocjenjivanje tutora koji je držao termin Instrukcija , u slučaju da je taj student pristupao datom terminu Instrukcija.

Na ovaj način možemo unaprijediti sigurnost sistema, a da time ne modificiramo postojeće klase. Ovim načinom nadogradnje aplikacije poštujemo OTVORENO ZATVOREN PRINCIP, budući da nadograđujemo entitete softvera bez njihove modifikacije.



5. COMPOSITE PATTERN

Osnovna namjena kompozitnog paterna je omogućavanje formiranja strukture stabla pomoću klasa, u kojoj se kompozicije individualnih objekata (korijeni stabla) i individualni objekti (listovi stabla) ravnopravno tretiraju, odnosno moguće je pozvati zajedničku metodu nad svim klasama. Omogućava nam uređenu hijerarhijsku strukturu koja zadržava uniformnost tako što je moguće istu metodu primjeniti nad različitim implementacijama.

U našem sistemu možemo proširiti klasu Materijali korisničkim zahtjevom da postoje dvije vrste materijala koje se nude, to su video materijali i tekstualni materijali. Tako bi se stvorila binarna struktura u kojoj je klasa Materijal korijen stabla, a nove klase VideoMaterijal i TekstualniMaterijal su listovi stabla. Da bi ostvarili kompozitni patern potrebno je implementirati interfejs ICijene koji ima metodu dajCijenu() koja će da vrati cijenu pojedinačnog materijala, a koja će se moći pozivati unutar klase Materijali ukoliko se doda klasa dajSveCijene() koja vraća listu svih cijena za svaki materijal. Pritom video i tekstualni materijali nemaju ništa zajedničko, odnosno cijene se vraćaju na različit način s različitom programskom logikom.

6. FLYWEIGHT PATTERN

Primarna funkcija Flyweight paterna je da dozvoli da više različitih objekata dijele isto glavno stanje, a da im sporedna stanja budu različita.

Koristimo ga jer pomoću njega postižemo manju i racionalniju upotrebu resursa kao što je radna memorija i znatno brže izvršavanje programa.

Naš sistem je moguće prosiriti korisničkim zahtjevom da se omogući svim korisnicima aplikacije da imaju svoju profilnu sliku (i tutoru i studentima), u tom slučaju bilo bi potrebno koristiti unaprijed zadanu sliku, odnosno avatara, ukoliko korisnik ne bude želio da prikazuje vlastitu sliku. S obzirom da je moguće da više korisnika zadrži tu default-nu sliku potrebno je implementirati flyweight pattern kako bi ti korisnici koristili jedan zajednički resurs.

To možemo postići kreiranjem interfejsa ISlika koji će nam vratiti sliku svih korisnika. Taj interfejs će implementirati klase Slika koja čuva individualnu sliku korisnika i AvatarSlika koja čuva defaultnu sliku, potrebna je samo jedna njena instanca u programu. Interfejs ima metodu dajSliku() koja daje sliku korisnika.

7. FACADE PATTERN

Glavna namjena facade paterna je osiguravanje više pogleda visokog nivoa na podsisteme čija je implementacija skrivena od korisnika.

Naš sistem je moguće proširiti, tako što bi kreirali klasu InstrukcijeFasada koja bi objedinjavala naše podsisteme vezane za klase Instrukcija i Materijal. Klasa instrukcijeFasada bi implementirala metode:

- zakaziInstrukciju koja kreira novu instancu klase Instrukcija kojoj je potreban datum, predmet i odabrani tutor
- promijeniTerminInstrukcije kojoj je potrebna instrukcija kojoj se mijenja datum i novi datum instrukcija
- promijeniNacinOdrzavanjaInstrukcije potrebna instrukcija kojoj se mijenja datum i novi datum instrukcija
- otkaziInstrukciju koja otkazuje odabranu instrukciju kojoj je potreban id instrukcije
- dajMaterijaleZaInstrukciju koja daje materijale potrebne za te instrukcije i taj predmet