26.11.2021.

Objektno-orijentisano programiranje 2

Dekorator

Motivacija i osnovna ideja:

Pretpostavimo da imamo zadatak da napišemo klasu/klase za predstavljanje ratnika u nekoj video igri. Ratnik recimo može imati metode kojima zadaje udarac i prima udarac. Pretpostavimo da ratnik može (ali ne mora) imati mač, oklop i štit. Metode za zadavanje udarca i primanje udarca zavise od toga koju opremu ratnik ima.

Jedno moguće "rešenje" bilo bi da imamo zasebnu klasu za svaku moguću kombinaiciju opreme (klase RatnikSaŠtitom, RatnikSaMačem, RatnikSaOklopom, RatnikSaŠtitomIMačem, itd). Šta ako hoćemo da dodamo i kacigu? Štaviše, možda ćemo nezavisno od opreme imati izvedene klase RatnikTipaA, RatnikTipaB. Dodavanje izvedenih klasa za sve moguće slučajeve praktično je nemoguće.

Drugi pristup bio bi da prosto klasi ratnik dodamo kao opcione atribute pokazivače na štit, mač i oklop, a da zatim kompleksnom logikom implementiramo ponašanje metoda za trčanje, zadavanje i primanje udarca u zavisnosti od toga koju opremu ima. Ovde bi dodavanje novog tipa opreme povlačilo za sobom usložnjavanje osnovne klase. Čak i kada se osnovna klasa ne bi značajno usložnjavala, možda iz nekog razloga **ne želimo ili ne možemo da je promenimo**, a hoćemo da ***unapredimo* ponašanje** objekta te klase, a prosto nasleđivanje nije zadovoljavajuće rešenje.

Projektni uzorak dekorator (engl. Decorator) rešava opisani problem. Uzorak treba koristiti kada želimo da unapredimo ponašanje postojećih funkcionalnosti postojećih klasa, ali na način da je moguće lako konfigurisanje i kombinovanje različitih unapređenja/dodataka/*dekoracija*.

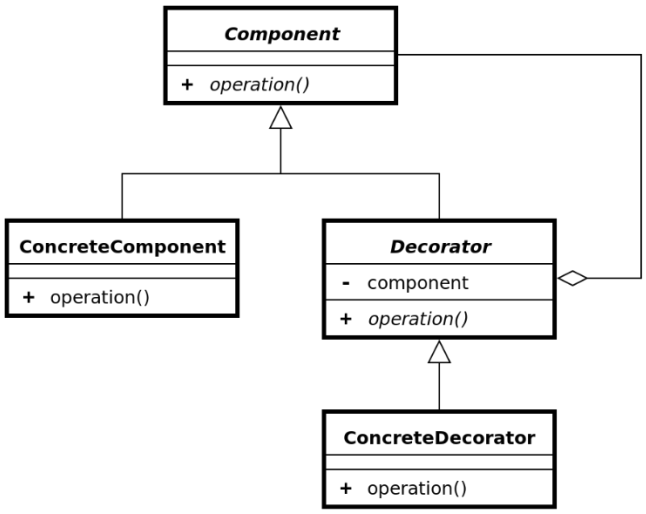
Kako sve ovo postići? Pretpostavimo da hoćemo *dekorisati* klase izvedene iz neke klase Component. Pre svega, potrebna nam je apstratkna klasa Decorator koju izvodimo iz klase Component. Tako postižemo da s dekoratorom možemo da komuniciramo na isti način (i preko pokazivača tipa Component) kao sa objektima iz hijerarhije Component. Druga jako bitna stvar je da Decorator ima pokazivač tipa Component. Drugim rečima, dekorator "zna" koga treba da dekoriše. Konkretni dekoratori treba da delegiraju pozive metoda objektu koji dekorišu, ali i da pre ili nakon što dekorisani objekat izvrši metodu izvrše neku dodatnu logiku.

Opisani način organizacije klasa omogućuje nam da konstrukcijom objekta mi iskombinujemo željene dekoracije na način (uzmimo za primer ratnika sa početka fajla koji treba da ima oklop i mač):

```
Ratnik* ratnik = new RatnikSaOklopom(new RatnikSaMačem(new RatnikTipaA()));
```

Štaviše, kao što vidimo, dekorator nam omogućuje i da definišemo redosled dekoracija koje će se primenjivati (RatnikSaOklopom i RatnikSaMačem su konkretne dekoracije), ukoliko je to od značaja za funkcionalnost.

UML dijagram projektnog uzorka Dekorator dat je na slici 1.



Slika 1 Projektni uzorak Dekorator

Zadaci:

1. Implementirati opšti oblik projektnog uzorka Dekorator u skladu sa UML dijagramom na slici 1.
2. Primeniti projektni uzorak Dekorator na situaciju opisanu u uvodu ovog fajla. Dodati atribute od značaja (recimo, int odbrana, napad).
3. Napisati klasu MessageSender sa metodom void send(const std::string&); Dodati dekoratore koji omogućavaju:
    a. Dodavanje trenutnog datuma i vremena na kraj poruke. Datum i vreme možemo dobiti sa:
    time_t my_time = time(NULL);
    std::string s = std::ctime(&my_time);
    (potrebno je postaviti argument _CRT_SECURE_NO_WARNINGS u Configuration Properties>>C/C++>>Preporocessor>>Preprocessor Definitions>> da bismo mogli da koristimo ctime).
    b. Čuvanje istorije poruka u lokalni fajl.
    c. Jednostavnu enkripciju poruke Cezarovom šifrom (za zadat broj k šiftovaćemo sve karaktere u karakter sa ascii kodom većim za k).

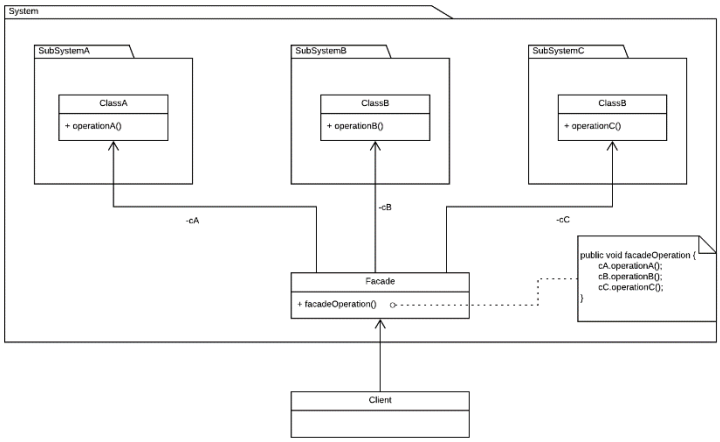
Fasada

Motivacija i osnovna ideja:

Pretpostavimo da imamo mnoštvo klasa koje imaju međusobne zavisnosti i za kompletiranje nekog posla potrebno je izvršiti mnoštvo poziva različitih metoda/funkcija različitih klasa. Pritom, naglasimo da su postojeće klase već dobro dizajnirane i kompleksnost koja se javlja nije rezultat lošeg dizajna, već same prirode problema i fleksibilnosti koje naš sistem nudi.

Pretpostavimo takođe da većina korisnika koristi kompleksni sistem na isti način (ili možda par načina). Tada možemo kreirati posebnu klasu koja omogućuje korisnicima da kompleksni sistem koriste na prostiji način (koji je za prosečne korisnike dovoljan). Takvu klasu nazivamo *fasadom*. Fasada nije tu da sakrije kompleksni sistem, već samo da pruži dodatni lakši način za korišćenje sistema. Korisnicima mora biti dostupno i direktno korišćenje kompleksnog sistema u slučaju da uprošćeni interfejs koji nudi fasada ne može da zadovolji njihove potrebe.

Učesnici projektnog uzorka Fasada i njihova komunikacija dati su UML dijagramom prikazanim na slici 2.



Slika 2 Projektni uzorak Fasada

Zadaci:

4. Napisati sistem koji simulira gledanje filmova na nekom sajtu. Da bi se pokrenuo film potrebno je prvo na osnovu imena filma pronaći sve izvore strimovanja datog filma. Zatim odabrati željeni izvor. Opciono, pre pokretanja filma mogu se učitati prevodi filma. Prvo se poziva statička metoda getHash(string movie) klase MovieToHash koja vraća heš kod na osnovu koga se pronalaze dostupni titlovi datog filma za željeni jezik. Klasa SubtitleDownloader sa statičkom metodom getSubtitle(int hash, string language) vraća listu prevoda datog filma za željeni jezik. Više prevoda može se učitati filmu, a bira se jedan koji se prikazuje. Pre pokretanja filma bira se i željena rezolucija. Ukoliko se odabere

opcija auto, poziva se funkcija assessOptimalResolution() koja poziva metodu getDownloadSpeed() klase NetworkSpeedMeter i na osnovu toga vraća optimalnu rezoluciju.
Ono što je većini korisnika potrebno i dovoljno je da film strimuju sa bilo kog izvora (prvog na listi recimo) uz učitane bilo koje prevode (transkripte) na engleskom i srpskom, a sa srpskim prevodom koji prikazujemo i uz optimalnu rezoluciju. Implementirati opisane klase i napraviti fasadu koja omogućuje pojednostavljeni način pokretanja videa.