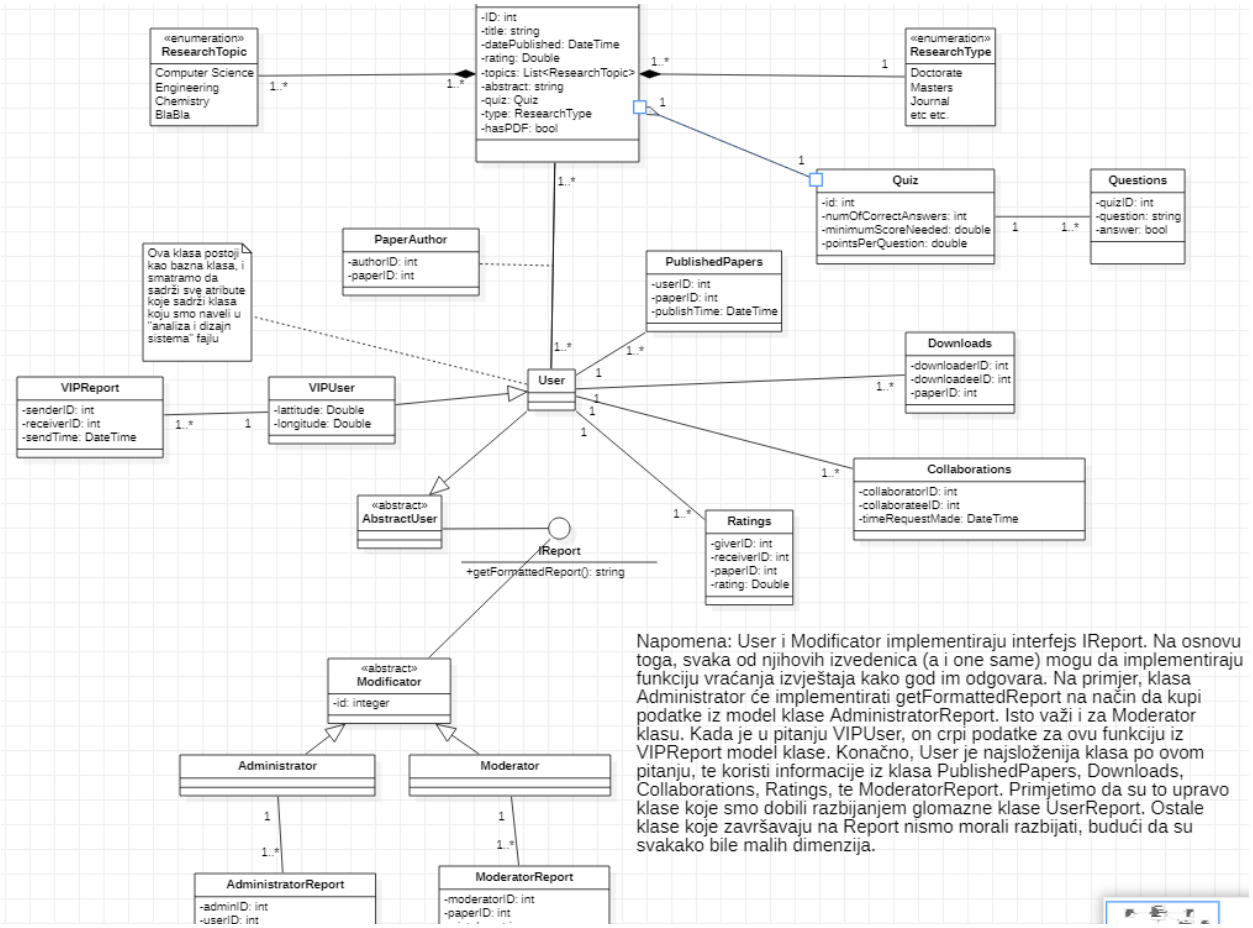
DIJAGRAM KLASA



Poštivanje SOLID principa

Proći ćemo kroz svaki od principa ukratko:

- Single responsibility principle: Kao što se da primjetiti, sve klase su relativno male po svom broju atributa, što je dobar indikator da svaka od njih ima po jednu odgovornost. Naime, najveća klasa je ResearchPaper, ali je većina atributa informativne prirode, a ne funkcionalne. U odnosu na Analizu i Dizajn sistema (task), gdje smo imali dosta velike klase (pogotovo klasa UserReport), sada smo ih razbili na manje klase (kao što su Downloads, PublishedPapers, Collaborators), kako zbog SRP principa, tako i zbog baze podataka.

- Open/Closed princip: Recimo ukoliko želimo uvesti novi tip korisnika u sistem, dovoljno je da se naslijedi User (ili ako to ne želimo ne moramo, možemo samo naslijediti IReport), i tada smo u mogućnosti da sa nekim novim klasama dodanim za funkcionalnosti tog novog user-a, recimo realiziramo funkciju

DIJAGRAM KLASA

getFormattedReport(), čime ćemo bez mijenjanja starih stvari (samo nadogradnja, dodavanje novih klasa i nasljeđivanja) ostvariti upgrade sistema, u ovom primjeru sa novim tipom korisnika.

- Liskov princip zamjene: ovo možemo testirati na nasljeđivanjima, budući da se ovaj princip odnosi na ispitivanje pravilnog nasljeđivanja. Kod Modificator-a, i Administrator i Moderator zaista jesu jedna vrsta (specijalni tip) Modifictor-a (onoga koji mijenja sistem) sistema. Bilo koja funkcija koju obavlja Modificator, može je obavljati i Administrator/Moderator. Kada je u pitanju User i VIPUser, očito da je VIPUser samo specijalna vrsta običnog User-a. On ima sve kao i obični user, ali ima i još par dodatno omogućenih funkcionalnosti zahvaljujući svome statusu.

- Interface segregation princip: Ovdje se radi o tome da neki klijent u vidu klase ne treba da implementira interfejs kog neće koristiti. Kod nas ima samo jedan interfejs (IReport), i on ima samo jednu funkciju, i ta funkcija se koristi od strane svih korisnika sistema (VIPUser, User), kao i od strane svih Modifikatora sistema, tako da ovdje ne treba razdvajati ništa.

- Dependency inversion principle: ovo pravilo se ogleda u tome da bazne klase trebaju da budu apstraktne. Ovo sam ostvario kod Modificator-a sistema, on je apstraktni, i bazni za Administratora i Moderatora. Međutim, u slučaju User-a je stvar malo komplikovanija. Naime, prvobitno sam i naumio da User bude apstraktna klasa, koju će naslijeđivati GuestUser, a kojeg će opet naslijediti VIP korisnik. Međutim, ispostavlja se da GuestUser nema nikakvih konkretnih funkcionalnosti (sve se svodi na browsing, i pregled stranice), tako da je on ispao iz igre. Prijavljeni korisnik (User) je postavljen da naslijeđuje AbstractUser-a, koji je naravno apstraktan. Time je ovaj DIP princip koliko-toliko zadovoljen.

Strukturalni patterni našeg sistema

DIJAGRAM KLASA

U našem sistemu implementirali smo tri patterna - Proxy, Decorator, i Composite. Što se tiče ostalih patterna, njih ćemo samo objasniti, gdje bi mogli da se nalaze u našem sistemu. Također, objasnit' ćemo i potrebu za onim patternima koje smo uveli u sistem.

Adapter pattern

Ovaj pattern nismo koristili. Ipak, moguće ga je iskoristiti sa klasama VIPUser i User. Pretpostavimo da želimo napraviti kolaboracijski zahtjev, preko neke funkcije collaborationRequest(). To radi User, i to bi bio naš specifični zahtjev. U tom slučaju, User bi bio Adaptee - onaj čiji je zahtjev potrebno prilagoditi.

Sada, budući da imamo sličnu funkcionalnost u VIPUser-u (to je zahtjev za odlazak na piće, kod kojeg je eventualno potrebno promijeniti tekst poruke, ili nešto slično), moguće je napraviti novi interfejs "IRequest" koji bi nam služio za bilo kakve zahtjeve koje želimo korsititi. VIPUser bi se sada koristio kao prilagođivač (adapter), gdje bi on kao request funkciju, implementirao novi vid zahtjeva (zahtjev za odlazak na piće), ali tako što bi prilagodio već postojeću funkciju collaborationRequest(), koju svakako dobiva jer nasljeđuje User-a. Npr. samo se toj funkciji proslijedi drugačiji tekst, neki parametri se izostave, i slično.

Facade pattern

Ovaj pattern također nije iskorišten. Pretpostavimo da u našem sistemu imamo jasno defnisane podsisteme: Podsistem za rejtinge, za zahtjeve, te za izvještaje. U slučaju da imamo dovoljno veliki broj ovih podsistema, kao i dovoljnu raznolikost među korisnicima, da bismo im trebali omogućiti različite poglede na sistem, mogli bismo napraviti Facade klasu, koja bi enkapsulirala sve ove podsisteme, te pružala jedinstven interfejs, u ovisnosti od toga o kojem klijentu se radi. Međutim, ovo je srednje do lakše složen sistem, tako da nismo imali dovoljno funkcionalnosti, a ni vrsti klijenata, da bismo bili u potrebi da koristimo ovakav pattern.

Decorator pattern

Ovaj pattern je iskorišten sa izvještajima. Naime, kada se korisnik tek prijavi na sistem, možda on ne želi da dobiva dnevne izvještaje sa svoga računa. Zbog toga, ovo ne bi trebalo da se automatski šalje, i smeta korisniku (spamuje korisnika mailovima). Zato, uveden je Decorator pattern koji će "dinamički proširiti" funkcionalnost izvještaja, tako što će krenuti slati dnevne izvještaje tek onda kada se korisnik složi sa time. Dakle,

DIJAGRAM KLASA

DecoratorReport klasa enkapsulira izvještaj kao Interfejs, te ukoliko bilo koji od raznih vrsta korisnika (bio User ili Modificator vrste) odobri (enable-a) slanje mail-ova, tada se ova funkcionalnost pokreće, i slanje zahtjeva od strane ove funkcije samo poziva zahtjev od enkapsuliranog usera kroz odgovarajući interfejs. Također, imamo mogućnost i onemogućavanja slanja ovih interfejsa, u bilo kojem trenutku.

Bridge pattern

Ovaj pattern nije iskorišten. Ipak, da smo bili u situaciji da omogućimo da jedan korisnik bude istovremeno i moderator i administrator, tada bismo u njega mogli enkapsulirati IReports interfejs, na osnovu kojeg bi on, u ovisnosti od toga u kojem modu rada se nalazi, pozivao ili izvještaj za moderatora, ili izvještaj za administratora. Budući da nismo omogućili takvu vrstu korisnika, to ovaj pattern nismo ni iskoristili :(

Proxy pattern

Ovaj pattern je iskorišten kod download-a. Opće je poznato da smo rekli da VIP korisnike neće morati provjeravati da li je došlo do prekoračenja maksimalnog dnevnog iznosa radova koje isti smije skinuti. Međutim, sasvim druga priča se javlja kada je u pitanju obični registrovani korisnik (User). Kod njega ne smijemo download-ovati ukoliko nismo prvo sigurni da nismo izvršili prekoračenje. Stoga, uvodi se ProxyDownload koji će omogućiti automatsku provjeru dostizanja dnevne granice (limita skinutih radova). Ukoliko ona nije pređena, standardno se poziva funkcija za skidanje rada na računar. Ukoliko jeste pređena, nemamo pravo skinuti na uređaj, i izbacujemo alert našem korisniku.

Composite pattern

Ovaj pattern je iskorišten kod rejtinga radova. Potrebno je napomenuti da je ResearchPaper ustvari spojen odgovarajućom vezom sa Ratings modelskom klasom, ali jednostavno nije bilo prostora da se to učini na iole uredan način. Anyways, imamo interfejs “IRating” koji nam omogućuje da vratimo rejtinge za odgovarajući rad, iz kojeg se ova funkcija i poziva. Sada, da bismo mogli da vršimo iole kompleksnije pozive (nešto slično stream-ovima u Javi), kreiramo CompositeRating klasu, koja čuva niz IRating-a. Na ovaj način, kada implementiramo funkciju getRatings(), možemo vratiti ne samo rejtinge jednog rada, već svih radova koji se nalaze u promatranom nizu. Međutim, pored toga, moguće je držati u IRating i same instance CompositeRating klase (koja svakako implementira IRating interfejs). Zbog toga, moguće je vršiti još kompleksnija vraćanja ocjena (odnosno rejtinga) za odgovarajuću skupinu radova koja nam je od značaja.

DIJAGRAM KLASA