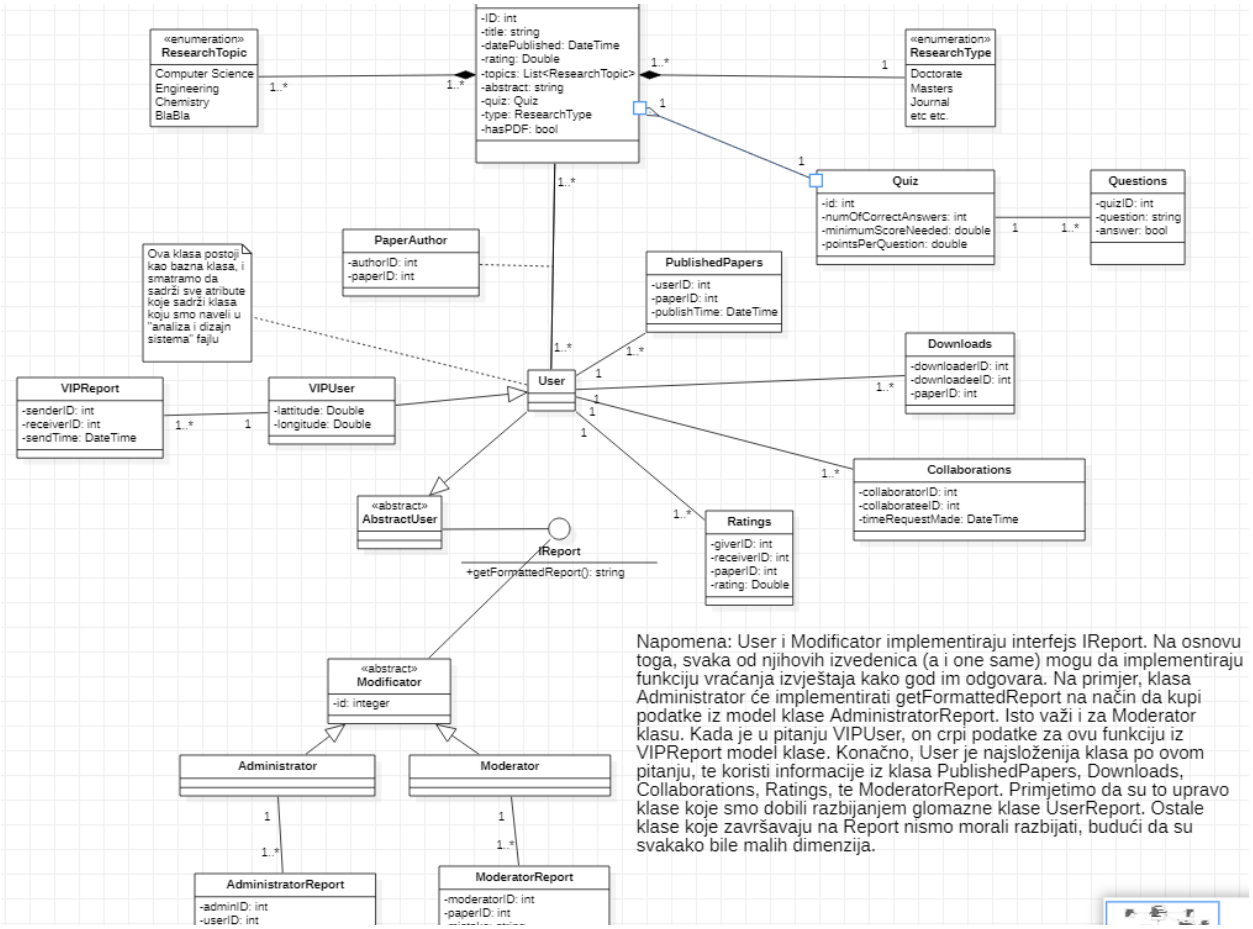
DIJAGRAM KLASA



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.

DIJAGRAM KLASA

Strukturalni patterni našeg sistema

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

DIJAGRAM KLASA

Ovaj pattern je iskorišten kod modifikacije ResearchPaper klase. Budući da ne želimo mijenjati ovu klasu, obezbjedili smo njenu dinamičku promjenu preko dekoratera. Naime, napravljen je interfejs IEdit, koji ima niz funkcija koje se odnose na rad sa osobinama ResearchPaper klase. Također, imamo i dva dekoratora koji nasljeđuju taj interfejs. Jedan služi za dodavanje novih stvari u ove atribute (uz ostajanje starih, recimo “topics” niz se može proširiti temama na koje se rad odnosi). Drugi služi za modifikaciju postojećih (recimo modifikacija abstract-a, modifikacija pdf dokumenta, i slično). Naravno, nakon obrade ovim dekoratorima, objekat se vraća preko odgovarajućih gettera. Moguće je vršiti i ugniježđeno editovanje, gdje se prvo objekat edituje preko Change dekoratora, a zatim i sa Add dekoratorom (dekorator za dodavanje novih osobina objektu).

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 :(. Naravno, moguće je u Bridge-u sastaviti zajednički dio koda (spoj, most) između ovih izvještaja, i zatim u odgovarajućoj interfejsnoj funkciji, samo pozivati dijelove koji se razlikuju, preko polimorfizma.

Proxy pattern

Pattern iskorišten kod download-a. Napravljen je interfejs za skidanje dokumenata, sa funkcijom “download”. Napravljena je proxy klasa, preko koje je User-u dopušteno da download-a dokumente. Naravno, prvo je potrebno da se autentificira, te mu je nakon toga dozvoljeno da download-a dokumente. Dokumenti se mogu download-ati i direktno preko ResearchPaper klase (budući da ona nasljeđuje download), ali je potrebno proslijediti šifru, koju je moguće dobiti samo ukoliko radimo preko proxya. Naravno, ukoliko postoji neka urgentna potreba za ovom funkcionalnošću, moguće je da administrator dodijeli odgovarajućoj klasi dopuštenje na korištenje šifre, u svrhu zaobilaženja proksija (ukoliko se radi o nekom poznatom, povjerljivom izvoru).

Composite pattern

Ovaj pattern nije iskorišten. Ipak, mogli smo ga iskoristiti kod prikaza osnovnih informacija korisniku. Napravimo IComposite, koji će imati recimo funkciju “show()”, za prikazivanje. Nakon toga, moguće je konkretnim klasama (koje bi naslijedile ovaj IComposite interfejs, recimo klase za rejtinge, slične radove, i ostale informacije)

DIJAGRAM KLASA

prikazivati specifične informacije, implementacijom ovog interfejsa. Konačno, da bi sve ovo bilo zajedno prikazano, moguće je napraviti Composite klasu, koja nasljeđuje i agregira IComposite interfejs. Ona bi držala niz IComposite objekata. Naravno, tako bismo u ovoj klasi, implementacijom metode “show” mogli primjeniti elementwise - svaku od pojedinačnih metoda “show” od objekata iz niza IComposite vrijednosti. Na taj način, bismo imali mogućnost da jednostavnim pozivom show funkcije od Composite klase, prikažemo jedan cijeli kompleksni dio prikaza (sačinjen od ocjena - rejtinga, sličnih radova, grupe autora koji su pisali projekat, i drugog).

Flyweight pattern

Što se tiče flyweight-a, ni on također nije iskorišten. Ipak, ukoliko bismo htjeli da optimizujemo našu aplikaciju, mogli bismo ga koristiti recimo kod svih user računa. Potrebno je prvo da uočimo na koji način najčešće profil biva napravljen, i koje se recimo oblasti i opcije najčešće biraju. To zatim zapamtimo, i pri svakom novom kreiranju korisničkog računa, provjerimo da li se račun sastoji od istih komponenti koje smo do sada već keširali (naravno, ovdje promatramo one dijelove koji su potencijalni kandidati da budu isti - odnosno oni kod kojih je to po prirodi moguće). Ukoliko imamo takve dijelove računa, dodijelimo ih promatranom korisniku, bez potrebe za novim pouzećem memorije. Ipak, ovo je nešto naprednije, a i ostavljeno je na predavanju studentima za analizu, tako da nećemo detaljnije opisivati ovaj pattern.

Kreacijski patterni i njihova primjena

U nastavku će biti objašnjene obavljene, i moguće implementacije kreacijskih patterna u našem projektu:

1) Singleton pattern - iskorišten je sa bazom podataka, odnosno sa DataContext klasom. Kada god neki dio programa bude htio koristiti ovaj pattern, onda ćemo prvo provjeriti da li je kreiran ovaj objekat. Ako jeste, njega koristimo. Ako nije, kreiramo ga, pa ga zatim koristimo.

2) Prototype pattern - ovaj pattern će vrlo vjerovatno biti iskorišten prilikom implementacije, jer će nam vjerovatno trebati. Naime, VIP korisnik u našoj aplikaciji će se vjerovatno postati od običnog korisnika (User-a), tako da ćemo morati nekako napraviti novi objekat. To je moguće uraditi tako što ćemo napraviti zajednički interfejs

DIJAGRAM KLASA

za User-a i VIPUser-a, i nakon toga, jednostavno napraviti funkciju “clone_v1”, i recimo “clone_v2” - sa ove dvije funkcije, mogli bismo mijenjati mod korisnika iz običnog usera u super korisnika, ali i obrnuto. Recimo, u clone_v1 možemo napraviti od običnog korisnika, VIP korisnika - na način da dodatne parametre popunimo default-nim vrijednostima (gdje ćemo se mi dogovoriti šta su to defaultne vrijednosti). Kod clone_v2, budući da VIP korisnik prestaje biti VIP, to ćemo mu uzeti moguće funkcionalnosti, tako što ćemo ga prebaciti u običnog User-a preko ove funkcije.

3) Factory method - ovaj pattern je iskorišten. Naime, asistentica Ehlymana je jasno naglasila da kod ovog metoda nije naglasak na nasljeđivanju, već na različitim implementacijama (pa čak i iste klase). Mi smo odlučili da nam createFormattedReport() bude metoda koja će kreirati (biti fabrika, factory) za string izvještaja koji šaljemo na mail. Treba imati na umu da String nije ništa drugo do klasa, tako da se ovdje radi o različitim instanciranjima klase String u ovisnosti od toga koja vrsta korisnika / moderator implementira ovu faktory metodu.

4) Abstract factory - ovo je dosta složen pattern, pa ćemo navesti i malo složenije mjesto na kojem se on može koristiti. Naime, ukoliko bismo odlučili da monetizujemo našu aplikaciju, na način da je potrebno plaćati učešće za različite račune koje na njoj pravimo, mogli bismo napraviti sljedeće. Odrediti recimo četiri plana kupovine računa na našoj aplikaciji. Mogli bismo kupovati račune u parovima, pa bi to bile mogućnosti: {Administrator, User}, {Administrator, VIPUser}, {Moderator, User}, {Moderator, VIPUser}. Naravno, tada bismo napravili 4 klase, recimo naziva PlanA, PlanB, PlanC, u kojem bismo za svaki od metoda imala po dva navedena korisnika, sa konkretnim tipom (a ne apstraktnim), i još nekim informacijama o planu plaćanja (recimo cijena, pogodnosti koje dolaze uz račune i slično). Sada, napravimo interfejs IFactory, koji će imati getModificator(), i getAbstrUser(), gdje će svaki od planova naslijediti ovaj interfejs i implementirati ga. Na taj način izvršili smo grupaciju sličnih i zajedno korištenih familija objekata i klasa, što je suština abstract patterna.

5) Builder pattern - nije iskorišten, ali vrlo vjerovatno da bi i on mogao biti iskorišten. Naime, prilikom pretrage, dostupno je brdo opcija koje je moguće pretražiti. Zbog toga, moguće je recimo formirati klasu Search, koja će držati neke bitne informacije o trenutnoj pretrazi (šta smo ukucali u searchbox, topics koje smo odabrali za pretragu, ime autora koje smo upisali i slično). Naravno, ovo ne bi bila jednostavna klasa. Sada, budući da je nekada moguće i da korisnici ne unesu u neka polja ništa, moguće je kreirati ManualSearchBuilder, i AutomaticSearchBuilder. Ova dva Buildera zatim mogu nasljediti interfejs IBuilder, u kojem ćemo imati razne metode za pravljenje pojedinih

dijelova Search query-ja. Naravno, u Manual builderu bi se stvari kreirale manualno, tj. Na osnovu pojedinih stvari koje su korisnici ukucali (stvarna pretraga). Kod automatskog buidlera, ovaj dio bi se koristio za one stvari koje korisnici ne ukucaju nikako, pa moramo za njih automatski odlučiti šta želimo da postavimo u naša polja za pretragu.

Patterni ponašanja

U nastavku su opisani implementirani patterni ponašanja, kao i mjesta gdje bismo ih mogli upotrijebiti:

Strategy pattern

- Ovaj pattern nije niskorišten. Ipak, mogli bismo ga iskoristiti na način da imamo algoritam rangirajRadove(), pri čemu bismo sa njim određivali kojim redoslijedom bi se radovi trebali prikazivati našem korisniku, prilikom pretrage. Naravno, ukoliko je on unio kriterije, ovo ne bi imalo smisla, međutim ukoliko samo pokrene pretragu, bez ikakvih kriterijuma, možemo sortirati radove na koji god način mi želimo. Ovo bi bila apstraktna metoda, koju bismo u ovisnosti od različite strategije (konkretnog algoritma kojeg koristimo) implementirali na različite načine (sortiramo abecedno, po broju dobivenih ocjena, i slično - za svaku od ovih strategija naravno ide po jedna klasa, kako se to i radi sa ovim patternom)

State pattern

- Ovaj pattern nike iskorišten. Ipak, mogli bismo ga koristiti prilikom odvijanja kolaboracije između dva naučnika. Ovu kolaboraciju bismo podijelili na više dijelova, pri čemu bismo u ovisnosti od trenutne situacije našeg sistema (dakle ne iz želje User-a, već u ovisnosti od toga da li je došao odgovor ili ne - dakle neki vanjski uticaj, odnosno trigger) mijenjali stanje u kojem se nalazimo. Stanja koja bismo imali mogla bi biti Poslana poruka, Čekanje poruke, PrimljenaPoruka, i slično

DIJAGRAM KLASA

Observer pattern

- Ovaj pattern nije iskorišten. Međutim, mogli bismo ga koristiti kod dnevnih reporta, odnosno izvještaja. Naime, u našem sistemu ostavili smo da se ovo izvještavanje odvija automatski za svakog prijavljenog korisnika sistema. Međutim, sa ovim patternom mogli bismo omogućiti da svaki od korisnika odluči da li želi da se “subscribe-a” na našu aplikaciju, gdje bismo mu slali dnevne izvještaje. Dakle, user-i sistema bi bili Observer-i, odnosno oni koji su se pretplatili na slanje mailova svaki dan, dok bi naš sistem imao temu “mailova”, gdje bi svakome slao njegov mail, naravno ukoliko je on “prikačen”, subscribe-ovan na promatranu temu slanja mailova.

Iterator pattern

- Iterator pattern je iskorišten. Naime, ukoliko želimo i imamo potrebu da se krećemo kroz pitanja kviza, to nam je sada omogućeno. Napravili smo interfejs **Iterator,** koji ima metodu nextQuestion(), koja vraća iduće pitanje promatranog kviza. Implementirali smo interfejs od strane klasa LengthIt i LexicographicIt, koje daju pitanja po dužini teksta u pitanju, odnosno po leksikografskom rasporedu pitanja, respektivno. U Quiz klasi smo agregirali interfejs Iterator, i u ovisnosti od toga šta prilikom kreiranja kviza proslijedimo kao klasu u ovu varijablu (LengthIt ili LexicographicIt), možemo da pozivom metode Iterator.nextQuestion(this, redni_broj) (jer se poziva iz kviza) pronađemo iduće pitanje.

Template method pattern

- Ovaj pattern je implementiran. Naime, imali smo sreću da je ovaj pattern svakako bio implementiran, što se da vidjeti u odnosu klasa User, i VIPUser. Naime, u ovom metodu je glavni cilj da se pokaže da je neke metode (koje proglasimo kao virtualne) moguće ili override-ati, ili preuzeti njihovu implementaciju, ukoliko želimo istu implementaciju. To je upravo urađeno od strane VIPUser-a, koji je preuzeo istu implementaciju funkcije getCollaborations(), a override-ao implementaciju funkcije publishPaper(). Naravno, što se tiče metode

DIJAGRAM KLASA

getCollaborations(), ona nije potrebno da se mijenja, budući da i obični i vip user imaju isti način obavljanja kolaboracija. Međutim, prilikom objavljivanja rada, moguće je recimo da ograničimo broj naučnih radova koje obični korisnik može da objavi. Također, moguće je da VIPUser-ima omogućimo upload-anje veće količine informacija o samome radu (jer što bi inače bili VIP). Zbog toga, metoda publishPaper() se override-a od strane ove klase, u cilju omogućavanja većeg broja informacija koje se mogu objaviti o samome radu, i zbog toga što nam npr. nije potrebna provjera o tome da li smo dostigli mjesečni limit objave naučnih radova, jer se sada radi o VIP korisniku, a ne User korisniku koji ima ograničene mogućnosti.