ANALIZA STRUKTURNIH PATERNA

1) Adapter patern

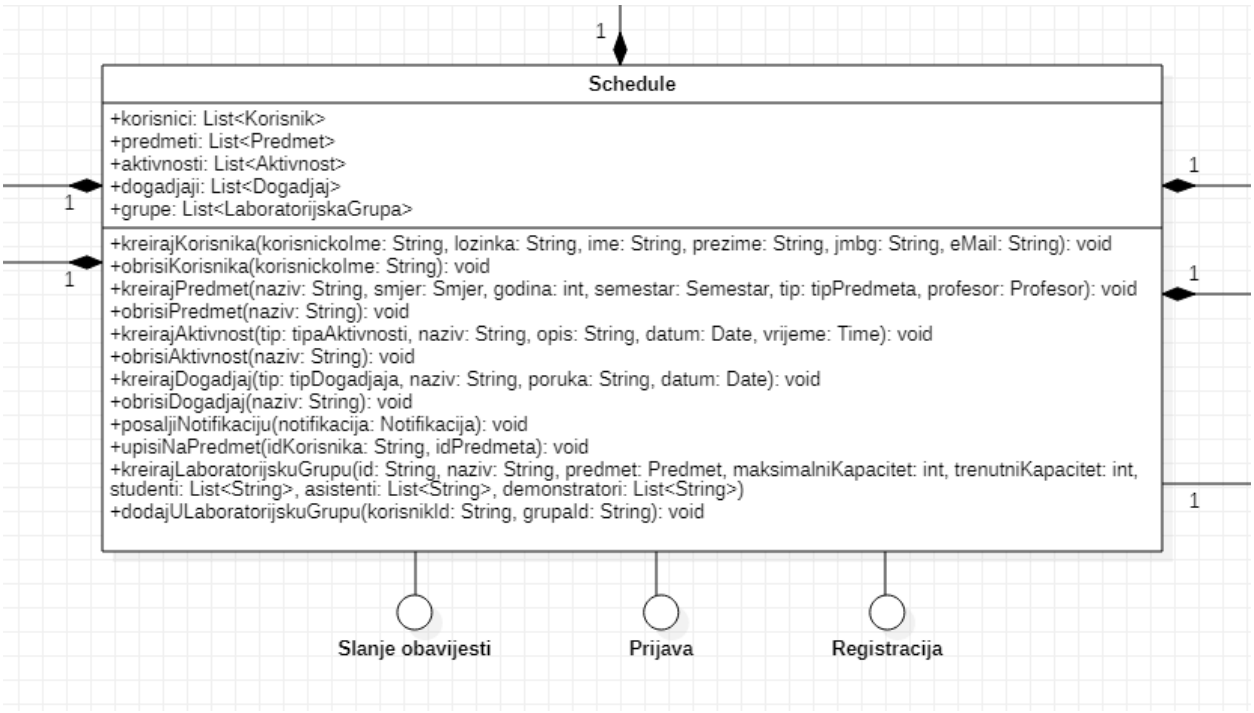
Adapter patern služi da se postojeći objekat prilagodi za korištenje na neki novi način u odnosu na postojeći rad, bez mijenjanja same definicije objekta. Na taj način obezbjeđuje se da će se objekti i dalje moći upotrebljavati na način kako su se dosad upotrebljavali, a u isto vrijeme će se omogućiti njihovo prilagođavanje novim uslovima.

Što se tiče **adapter paterna**, nismo mogli pronaći konkretnu primjenu koja bi odgovarala našem sistemu i učinila ga jasnijim i čitljivijim.

2) Fasadni patern

Fasadni patern služi kako bi se klijentima pojednostavilo korištenje kompleksnih sistema. Klijenti vide samo fasadu, odnosno krajnji izgled objekta, dok je njegova unutrašnja struktura skrivena. Na ovaj način smanjuje se mogućnost pojavljivanja grešaka jer klijenti ne moraju dobro poznavati sistem kako bi ga mogli koristiti.

Fasadni patern itekako pronalazi primjenu u našem sistemu. Da budemo precizniji,sistem je osmišljen tako da jedna klasa (u našem slučaju ***Schedule***) vrši kreiranje ***Korisnika, Profesora, Asistenata, Demonstratora i Studenata***. Sve metode koje koriste gore navedene klase se nalaze u klasi Schedule. Korisnik sistema ne mora biti upućen kako se pojedine metode odvijaju u pozadini jer je sve već spremno i implementirano za korištenje jedne složenije metode. Dio dijagrama gdje uočavamo ovaj patern možemo vidjeti na slici ispod.



3) 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.

Što se tiče **decorator paterna**, on teško da može pronaći svoju primjenu u našem sistemu. Jedino potencijalno mjesto gdje bi se **decorator patern** možda mogao primijeniti je kod klase ***Administrator***, konkretno metoda ***izmijeniKorisnika***. Ali primjena ovog paterna će samo zahtjevati dosta posvećenosti pri implementaciji te veću količinu izgubljenog vremena, dok će rezultati biti minorni. Tako da ćemo njegovu potencijalnu primjenu preskočiti.

4) Bridge patern

Bridge patern služi kako bi se apstrakcija nekog objekta odvojila od njegove implementacije. Ovaj patern veoma je važan jer omogućava ispunjavanje Open-Closed SOLID principa, odnosno uz poštivanje ovog paterna omogućava se

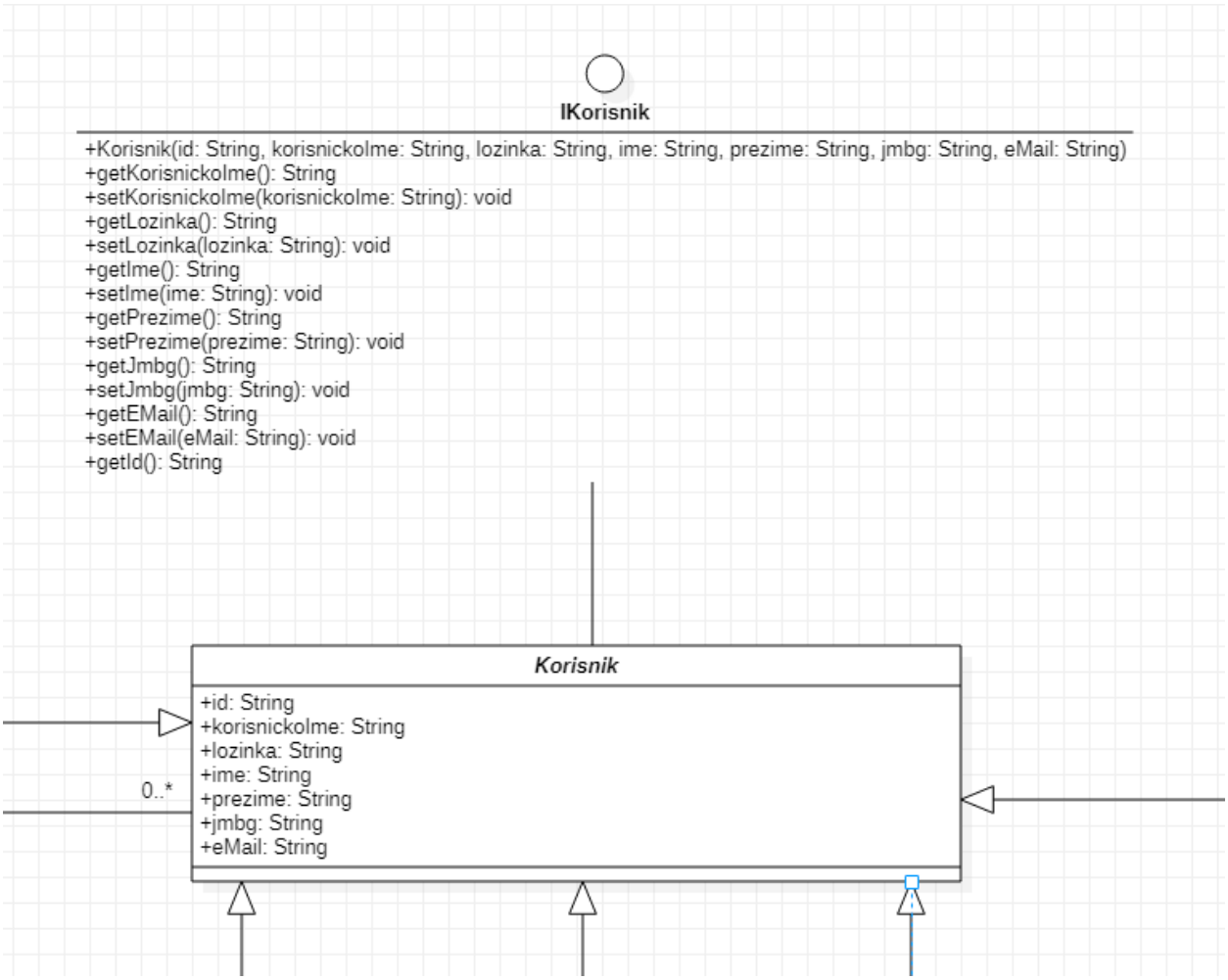
nadogradnja modela klasa u budućnosti te osigurava da se neće morati vršiti određene promjene u postojećim klasama.

Što se tiče "uslova" za primjenu **bridge paterna**, imamo pet klasa koje nasljeđuju baznu klasu (klasu ***Korisnik*** nasljeđuju ***Administrator, Student, Demonstrator, Asistent, Profesor***) ali ne postoji niti jedna metoda koju sve ove klase primjenjuju na različit način,a da se međusobno preklapaju nazivi metoda, odnosno da neka od izvedenih klasa koristi metodu koja se bitno razlikuje od metoda preostalih klasa. Sve su one samo *getteri i setteri*, tako da **bridge patern** nije pronašao svoju primjenu u našem sistemu.

5) Composite patern

Composite patern 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.

Što se tiče **composite paterna**, naš sistem je doista i organizovan u slijedu hijerarhije, tako da primjena **composite paterna** odgovara našem sistemu.
Hijararhija našeg sistema : ***Administrator – Profesor – Asistent – Demonstrator – Student*** i svi oni naslijeđeni iz bazne klase ***Korisnik*** .
Kako bi pojednostavili hijerarhiju objekata u formi, moguće je primijeniti **composite patern** na način da metode bazne klase zamijenimo sa interfejsom **IKorisnik**, te da klase koje su prethodno naslijedile klasu ***Korisnik***, naslijede ovaj interfejs. Dio class dijagrama ,sa primjenom composite paterna izgleda ovako :

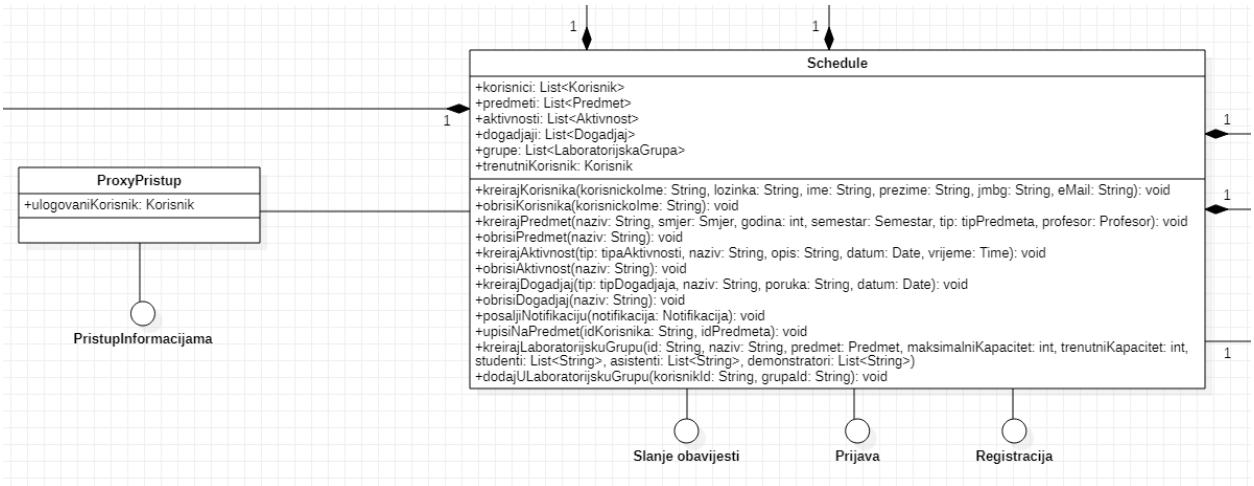


6) Proxy patern

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

Primjena **proxy paterna** u našem sistemu je moguća i jako korisna. Jedna od mogućih primjena je kod logiranja u naš sistem. Naime, potrebno je da na osnovu login podataka korisnika odrediti koje će sve funkcionalnosti on posjedovati kada se logira. Kako bi iskoristili **proxy patern**, dodamo klasu **ProxyPristup**, koja će imati svoj atribut **ulogovaniKorisnik** i implementirani interfejs **PristupInformacijama**. Na osnovu login informacija, ažurirat će se atribut **ulogovaniKorisnik**. Na osnovu atributa **ulogovaniKorisnik**, korisnik dobija sve

mogućnosti pristupa koje posjeduje klasa ulogovaniKorisnik. Dio class dijagrama sa primjenjenim **proxy paternom** će sada izgledati ovako :



7) Flyweight patern

Flyweight patern koristi se 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 instanca objekta (tzv. bezlično stanje). Korištenje ovog paterna veoma je korisno u slučajevima kada je potrebno vršiti uštedu memorije

Nismo mogli pronaći potencijalnu primjenu ovog paterna u našem sistemu.

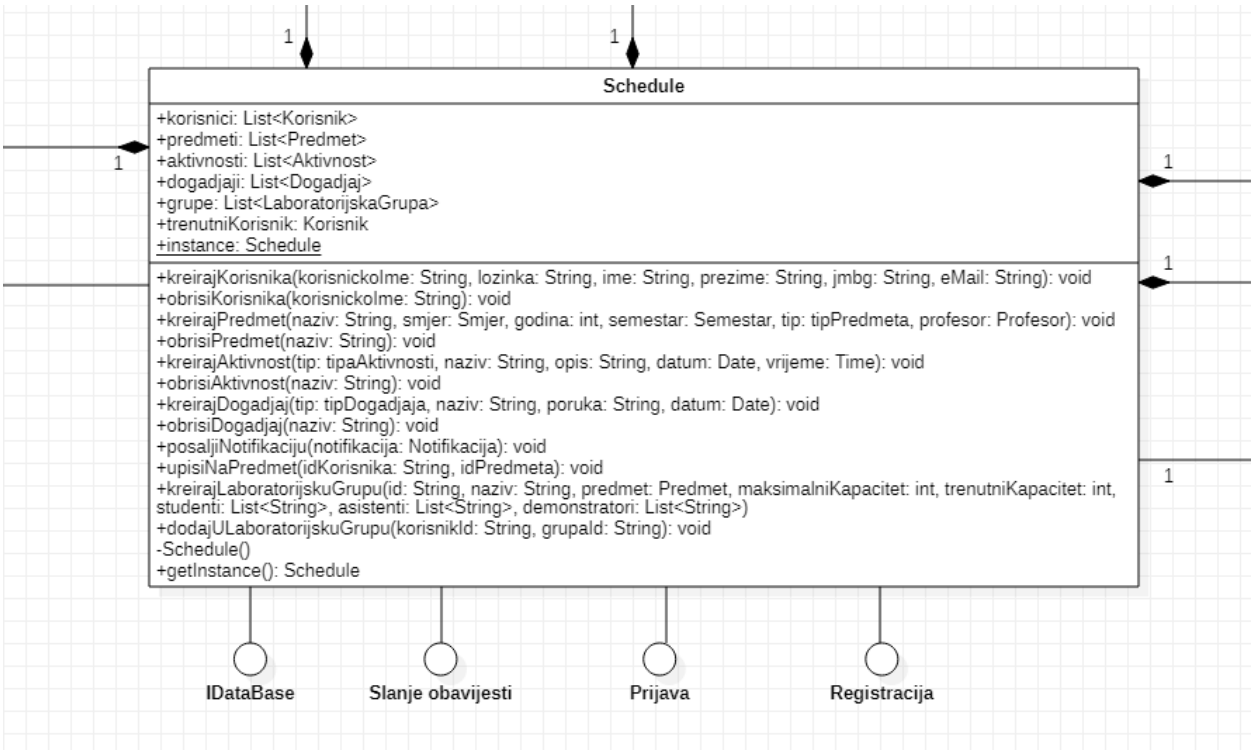
ANALIZA KREACIJSKIH PATERNA

1) Singleton patern

Singleton patern služi kako bi se neka klasa mogla instancirati samo jednom. Na ovaj način može se omogućiti i tzv. lazy initialization, odnosno instantacija klase tek

onda kada se to prvi put traži. Osim toga, osigurava se i globalni pristup jedinstvenoj instanci - svaki put kada joj se pokuša pristupiti, dobiti će se ista instanca klase. Ovo olakšava i kontrolu pristupa u slučaju kada je neophodno da postoji samo jedan objekat određenog tipa.

Primjena **singleton paterna** u našem sistemu se ogleda kroz klasu Schedule. Ona sadrži static atribut tipa Schedule, privatni konstruktor te metodu getInstance. Razlog što ovaj patern koristimo na ovom mjestu leži u tome što ova klasa predstavlja konekciju na bazu podataka, privatnim konstruktorom postižemo da imamo samo jednu instancu, a pozivanjem navedene metode dobijamo tu instancu. Na slici ispod možemo vidjeti kako taj dio dijagrama sada izgleda.



2) Prototype patern

Prototype patern omogućava smanjenje kompleksnosti kreiranja novog objekta tako što se uvodi operacija kloniranja. Na taj način prave se prototipi objekata koje je moguće replicirati više puta a zatim naknadno promijeniti jednu ili više karakteristika, bez potrebe za kreiranjem novog objekta nanovo od početka. Ovime se osigurava pojednostavljenje procesa kreiranja novih instanci, posebno kada objekti sadrže veliki broj atributa koji su za većinu instanci isti.

Primjena prototype paterna u našem sistemu zaista nije potrebna. Kloniranje objekata ne pronalazi svoju primjenu u našem sistemu.

3) Factory method patern

Factory method patern služi za omogućavanje instanciranje različitih vrsta podklasa koristeći factory metodu koja odlučuje koja će se podklasa instancirati i koja programska logika izvršiti. Na ovaj način osigurava se ispunjavanje O SOLID principa, jer se kod za kreiranje objekata različitih naslijeđenih klasa ne smješta samo u jednu zajedničku metodu, već svaka podklasa ima svoju logiku za instanciranje željenih klasa, a samo instanciranje kontroliše factory metoda koju različite klase implementiraju na različit način.

Factory method patern pronalazi svoju primjenu u našem sistemu. Jedna dobra primjena je prilikom logiranja korisnika na naš sistem . Naime, u zavisnosti od tipa korisnika koji se ulogovao, korisnik će da posjeduje različite priviligije, tako da je u tom momentu potrebno instancirati ulogovanog korisnika. S obzirom da sistem posjeduje veliki broj naslijeđenih klasa, **factory method patern** će ovdje poslužiti jako dobro.
Primjena **factory method paterna** u sistemu skoro pa se može poistovjetiti sa primjenom **proxy paterna,** jer služe istoj svrsi.

4) Abstract factory patern

Abstract factory patern služi kako bi se izbjeglo korištenje velikog broja if-else uslova pri kreiranju različitih hijerarhija objekata. Ukoliko postoji više tipova istih objekata te različite klase koriste različite podtipove, te klase postaju fabrike za kreiranje objekata zadanog podtipa bez potrebe za specificiranjem pojedinačnih objekata. Na ovaj način se, korištenjem nasljeđivanja, ukida potreba za postojanjem if-else uslova jer određeni tip fabrike sadrži određene tipove objekata i zna se tačno koju podklasu će instancira.

Primjena **abstract factory paterna** u našem sistemu je moguća, ali nije potrebna. Moguće je da kreiramo različite tipove *fabrika*, za različite tipove klasa koje koristimo, pomoću kojih ćemo tačno znati koja će se klasa instancirati. Na taj način bi, klasa ***Administrator*** mogla da funkcioniše bez svojih metoda za kreiranje. Međutim, trenutni sistem sa ovako koncipiranom klasom ***Administrator*** bi bio dosta jednostavniji za preglednost i čitljivost, nego sistem sa implementiranim **abstract factory paternom**.

5) Builder patern

Builder patern služi za apstrakciju procesa konstrukcije objekta, kako bi se kao rezultat mogle dobiti različite specifikacije objekta koristeći isti proces konstrukcije. Ovaj patern koristi se kako bi se izbjeglo kreiranje kompleksne hijerarhije klasa te kako bi se izbjegao kompleksni programski kod konstruktora jedne klase koja može imati različite konfiguracije atributa. Različiti dijelovi konstrukcije objekta izdvajaju se u posebne metode koje se zatim pozivaju različitim redoslijedom ili se poziv nekih dijelova izostavlja, kako bi se dobili željeni različiti podtipovi objekta bez potrebe za kreiranjem velikog broja podklasa.

Sistem nije u dovoljnoj mjeri kompleksan da bi bilo potrebe da primjeni **builder patern**. Pored kolekcijskih klasa, ne postoje druge klase koje sadrže neku klasu kao svoj atribut, što implicira da se sistem sastoji od jednostavnih klasa.