ANALIZA PATERNA PONAŠANJA

1) Strategy pattern

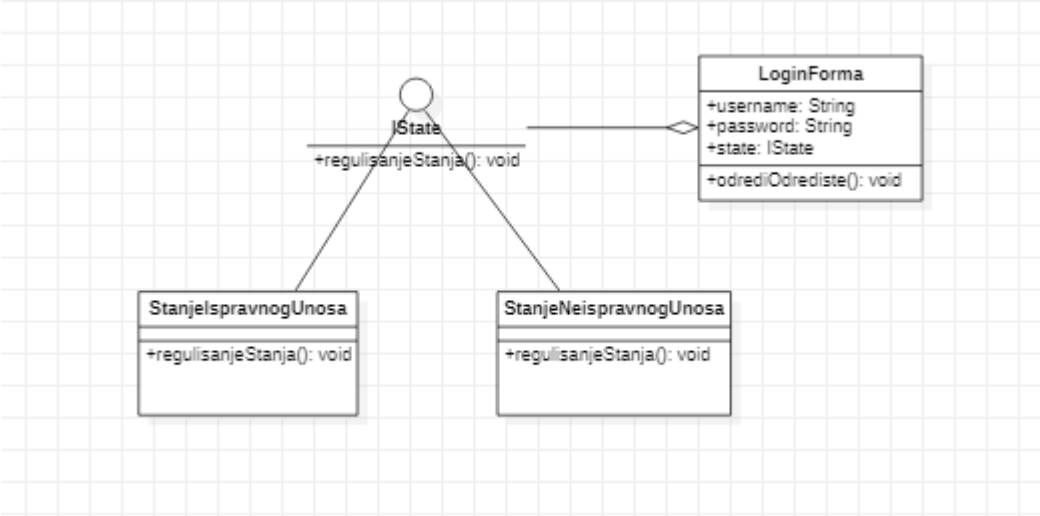
Startegy pattern *izdvaja algoritam iz matične klase i uključuje ga u posebne klase. Povoljan kada postoje različiti primjenivi algoritmi za neki problem.*

Naš sistem je koncipiran sa vrlo jednostavnim klasama i mjesto primjene ovog paterna nije očigledno, odnosno teško da možemo pronaći njegovu primjenu uopšte. Osim dobavljanja korisnikovih podataka iz baze podataka, nemamo nijednu primjenu algoritama u našem sistemu , dakle nemamo osnova za uvođenje ***strategy patterna***.

2) State patern

State pattern *je dinamička verzija strategy paterna.Objekat mijenja način ponašanja na osnovu trenutnog stanja*

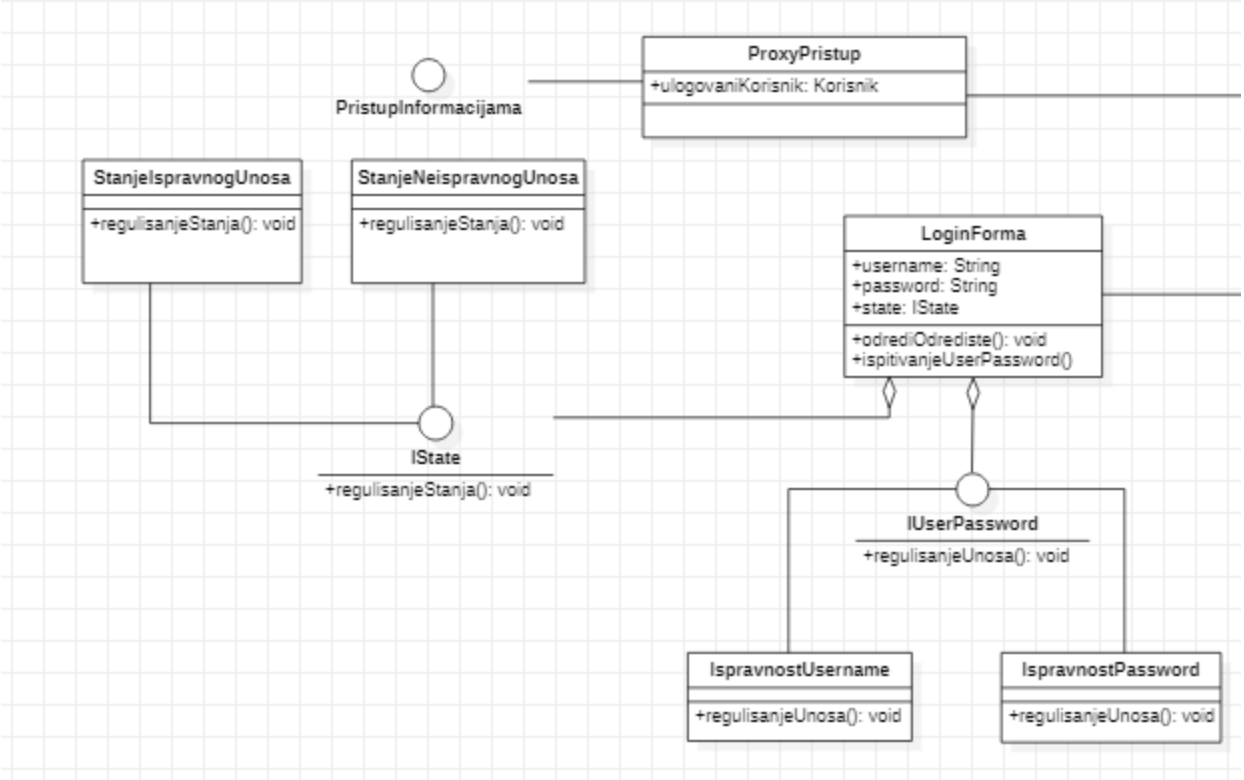
State pattern u našem sistemu pronalazi sjajnu primjenu. Jedno dobro mjesto primjene je kod logiranja korisnika. Naime, kod logiranja korisnika moguća su dva ishoda : *uneseni podaci su ispravni* i *uneseni podaci nisu ispravni*. **State patern** će biti primjenjen upravo za to, da na osnovu ispitivanja korektnosti ulaznih podataka odredi stanje sistema i da sistem nastavi svoje izvršavanje na osnovu tog stanja. Kako bi primjenili **state patern**, dodajemo interfejs **IState**. Dodajemo klasu **LoginForma,** sa svojim atributima *username* i *password*, te još jednim atributom *state*, koji će predstavljati stanje login forme, te metodom ***odrediOdrediste()*** koja će pozivati metodu *state* atributa. Ona će također biti povezana sa interfejsom kako bi interfejs mogao pristupiti podacima **LoginForme**. Dodajemo i dvije klase stanja : **StanjeIspravnogUnosa** i **StanjeNeispravnogUnosa** koje implementiraju **IState** interfejs. Dijagram klasa će nakon primjene **state paterna** izgledati ovako :



3) TemplateMethod patern

TemplateMethod patern *omogućava izdvajanje određenih koraka algoritma u odvojene podklase*

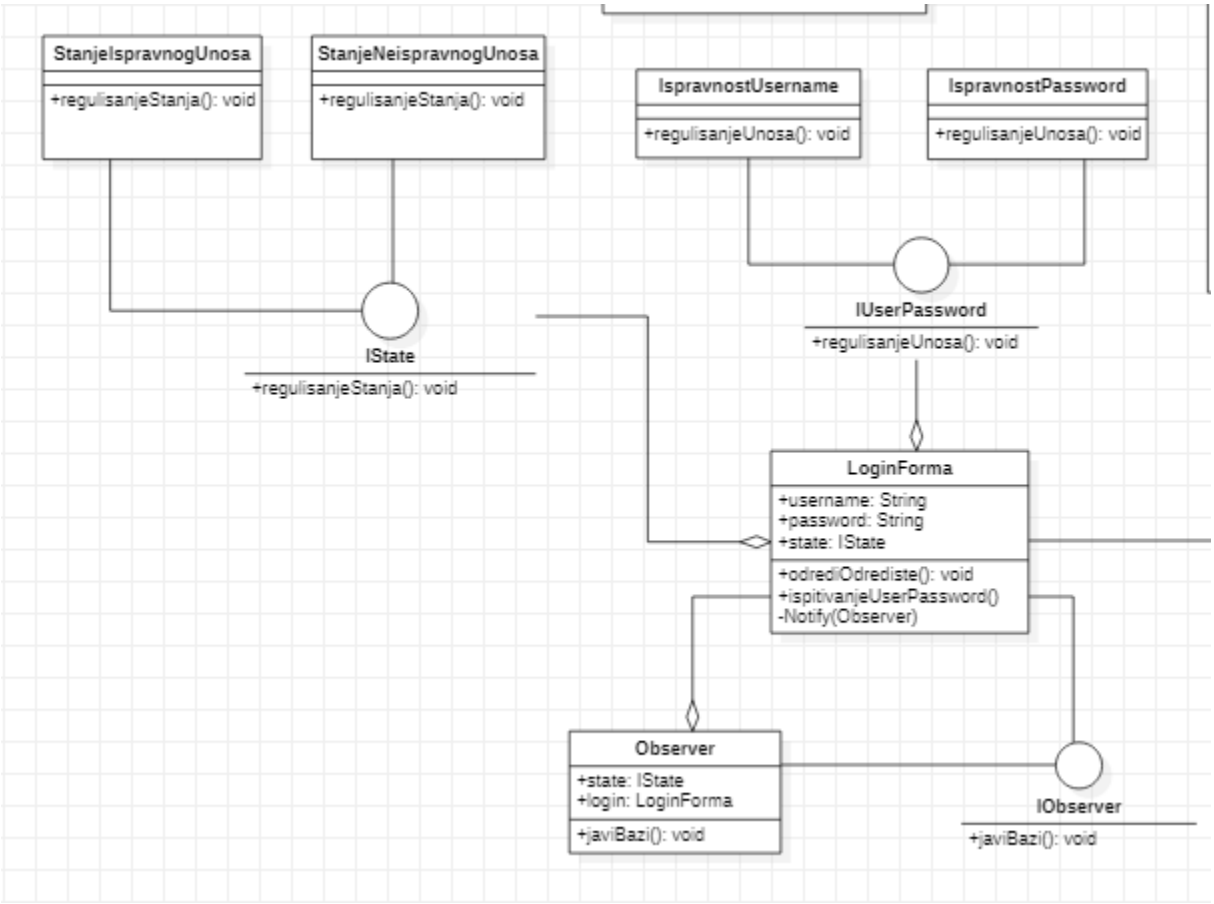
Što se tiče **TemplateMethod paterna**, potencijalno mjesto gdje bi se mogao primijeniti je kod login forme. Konkretno, da prilikom logiranja na sistem, da se prvo ispita *username* pa onda *password*, a ne da testiramo istovremeno. Ono što ćemo uraditi je to da će klase **IspravnostUsername** i **IspravnostPassword** koristiti interfejs **IUserPassword,** implementirati njegovu metodu ***ispravnostUnosa()*** i pomoću ovog interfejsa ćemo pristupiti atributima **LoginForme** te podijeliti zadatke ispitivanja formi unosa. Izgled dijela dijagrama klase nakon primjene ovog paterna će izgledati ovako :



4) Observer patern

*Uloga **Observer paterna** je da uspostavi relaciju između objekata tako kada jedan objekat promijeni stanje drugi zainteresirani objekti se obavještavaju.*

Primjena **Observer paterna** se isto može iskoristiti kod logiranja na sistem . S obzirom da imamo ograničen broj znakova koji je dozovljen za *username*, možemo primijeniti ovaj patern tako što ćemo na osnovu broja znakova na *username* formi unosa, obavijestiti bazu podataka da nema potrebe da vrši pretragu jer ovaj korisnik 100% ne postoji. Na taj način bi uštedili vrijeme. Dobra je ideja izvršiti primjenu ovog paterna nakon primjene **TemplateMethod paterna**. Za implementaciju ovog paterna, dodajemo interfejs **iObserver**, koji koristi klasa **Observer**, implementirajući njegovu metodu ***javiBazi()*** , te dodajemo potrebnu metodu ***Notify()*** u klasu **LoginForma** koja će slati obavijesti **Observeru**. Izgled dijagrama klasa nakon primjenjenog **observer paterna** :



5) Chain of Responsibility patern

Chain of Responsibility patern *predstavlja listu objekata, ukoliko objekat ne može da odgovori prosljeđuje zahtjev narednom u nizu.*

Što se tiče ovog paterna, nismo mogli pronaći odgovarajuću primjenu u našem sistemu. Naš sistem je jednostavan u smislu da ne postoji situacija gdje je jedan objekat potrebno proslijediti više od jednoj klasi. Implementacija ovog interfejsa bi bila suvišna.

6) Command patern

Command patern *razdvaja klijenta koji zahtjeva operaciju i omogućava slanje zahtjeva različitim primateljima*.

Što se tiče **command paterna**, on nije primjenjiv za naš sistem zato što ne postoji neka veća putanja koju podaci trebaju proći u klasi. Najveća putanja je na relaciji login forma – baza podataka, kako bi dobili podatke iz baze. Posrednik između ovo dvoje nije neophodan. Čak štaviše, možemo reći da smo primjenom **observer paterna**, tj. klasom **Observer** na neki način i dobili nekog posrednika na ovoj relaciji. Možemo reći da je **command patern** spontano ispoštovan.

7) Iterator patern

Iterator patern *omogućava pristup elementima kolekcije sekvencijalno bez poznavanja interne strukture kolekcije.*

S obzirom da naš sistem koristi listu kao kolekciju podataka, a lista ima svoj iterator liste, možemo reći da smo ispoštovali ovaj patern ponašanja.

8) Mediator patern

Mediator patern *enkapsulira protokol za komunikaciju među objektima dozvoljavajući da objekti komuniciraju bez međusobnog poznavanje interne strukture objekta.*

Mediator patern je u našem sistemu regulisan kroz login formu, jer login forma sadrži *username* i *password*, te klasa **LoginForma** služi kao medijator između njih dvoje, odnosno jedno ne zna za drugo ali jedno bez drugog ne funkcionišu, međusobno se nadopunjuju formirajući potrebnog *korisnika* za pristup sistemu.

9) Visitor patern

Visitor patern *definira i izvršava nove operacije nad elementima postojeće strukture ne mijenjajući samu strukturu.*

Primjena **visitor paterna** u našem sistemu nema dobru osnovu.Sistem ne sadrži veliki broj klasa sa različitim interfejsima na koje želimo primijeniti određene operacije u zavisnosti od tipova klase. Primjena Visitor paterna kod login forme je moguća, ali njegovu svrhu već obavlja**TemplateMethod patern.**

10) Interpreter patern

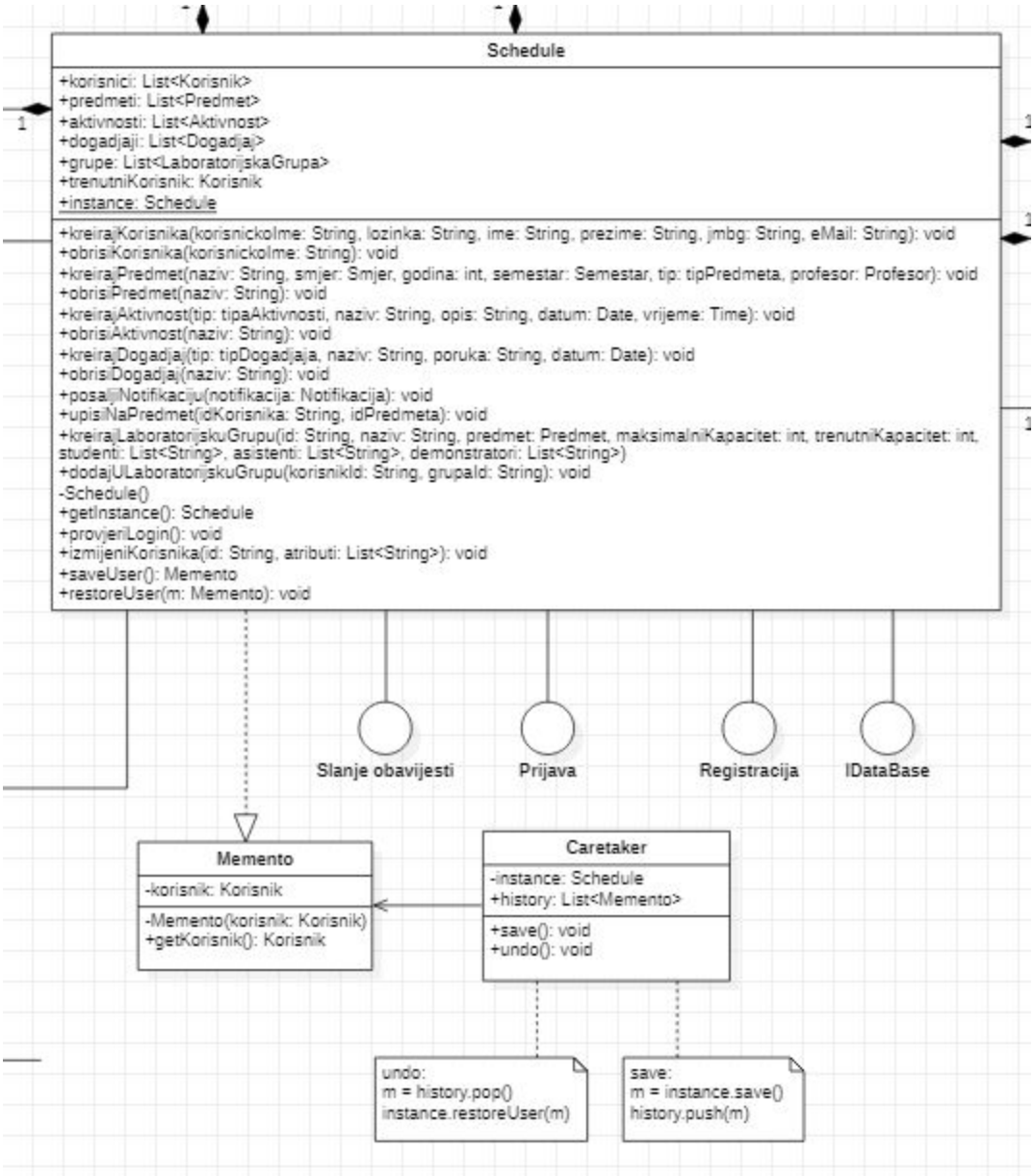
Intepreter patern *podržava interpretaciju instrukcija napisanih za određenu upotrebu.*

Klase u našem sistemu su dovoljno pojednostavljene da nema potrebe za **interpretorom**, odnosno detaljnijom interpretacijom podataka i njihovim daljim prosiljeđivanjem.

11) Memento patern

Memento patern *omogućava spašavanje internog stanja nekog objekta van sistema i njegovo ponovno vraćanje.*

Potencijalna primjena **memento paterna** u našem sistemu bi bila korisna radi funkcije **izmijeniKorisinka**. Dobro bi bilo da,u slučaju loše izmjene, bude moguće vratiti podatke korisnika onakve kakve su bile prije izmjene.Za tu svrhu će nam poslužiti upravo **memento patern**.Dodat ćemo klasu **Memento**, koja će ujedno i predstavljati ono što želimo sačuvati i klasu **Caretaker** sa svojim metodama koja će regulisati izmjene ili čuvanje podataka klase. Izgled dijagrama klasa sa primjenjenim **memento paternom** možemo vidjeti na sljedećoj slici :


Schedule
+korisnici: List<Korisnik>
+predmeti: List<Predmet>
+aktivnosti: List<Aktivnost>
+dogadjaji: List<Dogadjaj>
+grupe: List<LaboratorijskaGrupa>
+trenutniKorisnik: Korisnik
+instance: Schedule
+kreirajKorisnika(korisnickoIme: String, lozinka: String, ime: String, prezime: String, jmbg: String, eMail: String): void
+obrisiKorisnika(korisnickoIme: String): void
+kreirajPredmet(naziv: String, smjer: Smjer, godina: int, semestar: Semestar, tip: tipPredmeta, profesor: Profesor): void
+obrisiPredmet(naziv: String): void
+kreirajAktivnost(tip: tipaAktivnosti, naziv: String, opis: String, datum: Date, vrijeme: Time): void
+obrisiAktivnost(naziv: String): void
+kreirajDogadjaj(tip: tipDogadjaja, naziv: String, poruka: String, datum: Date): void
+obrisiDogadjaj(naziv: String): void
+posaljiNotifikaciju(notifikacija: Notifikacija): void
+upisiNaPredmet(idKorisnika: String, idPredmeta): void
+kreirajLaboratorijskuGrupu(id: String, naziv: String, predmet: Predmet, maksimalniKapacitet: int, trenutniKapacitet: int, studenti: List<String>, asistenti: List<String>, demonstratori: List<String>)
+dodajULaboratorijskuGrupu(korisnikId: String, grupaId: String): void
-Schedule()
+getInstance(): Schedule
+provjeriLogin(): void
+izmijeniKorisnika(id: String, atributi: List<String>): void
+saveUser(): Memento
+restoreUser(m: Memento): void
1
Slanje obavijesti
Prijava
Registracija
IDataBase
Memento
-korisnik: Korisnik
-Memento(korisnik: Korisnik)
+getKorisnik(): Korisnik
Caretaker
-instance: Schedule
+history: List<Memento>
+save(): void
+undo(): void
undo:
m = history.pop()
instance.restoreUser(m)
save:
m = instance.save()
history.push(m)