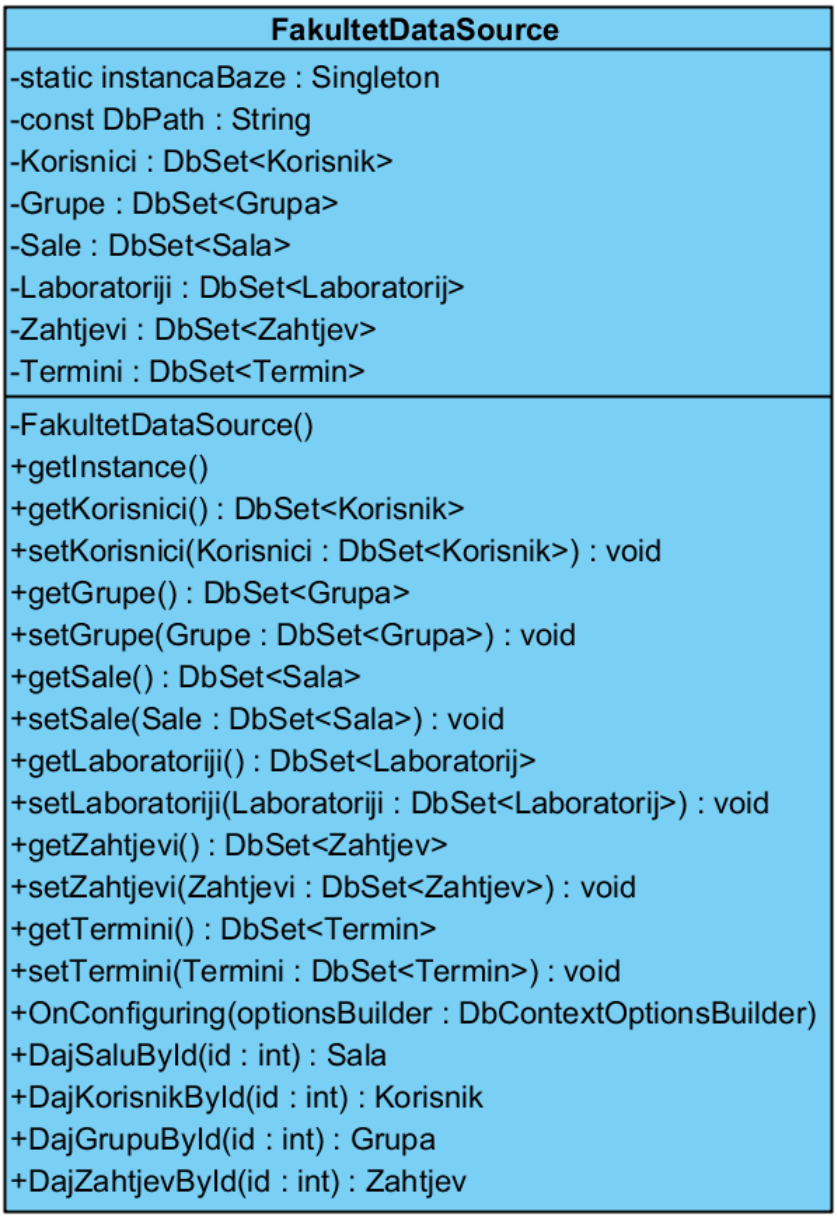
Dizajn paterni

1 KREACIJSKI DIZAJN PATERNI

1.1 SINGLETON

Ovaj patern bi se mogao koristiti za klasu koja pripada Data Access Layeru, FakultetDataSource. Prvenstveno iz razloga što služi za pristupanje bazi koja pripada određenom frameworku, te bi nadogradnja korištenjem nekog drugog paterna bila nezgodna. Drugi razlog je što je dovoljna jedna instanca klase koja komunicira s bazom (jer komunicira samo sa jednom bazom), te bi u slučaju višenitnog pristupa moglo doći do konflikta nad resursom. Implementacija je trivijalna, što se vidi na slici:



Istu funkcionalnost bismo mogli postići kad bismo klasu proglasili za statičku (što nam C# omogućava), čime bi sve operacije nad klasom morale biti proglašene statičkim.

Napomena – jako je teško dodati C# prototip (korištenjem stereotipa) u Visual Paradigm.

1.2 PROTOTYPE

U suštini, u sklopu našeg projekta nema velike potrebe za korištenjem prototipa, jer nema naročito velikih objekata.

1.3 FACTORY METHOD

Mogao bi se koristiti za klase koje služe za slanje zahtjeva, iako bi bilo donekle nepotrebno. Klasa ZahtjevRezervacija je izvedena iz klase Zahtjev. Možemo izbjeći eksplicitno instanciranje različitih tipova klasa ako ih, u slučaju da korisnik šalje zahtjev sa dodatnom informacijom, enkapsuliramo u Factory method.

1.4 ABSTRACT FACTORY

Nema ovolikog nivoa kompleksnosti u projektu.

1.5 BUILDER

Kao i za abstract factory, nema klasa koje enkapsuliraju dovoljan broj drugih da bi korištenje ovog paterna imalo smisao

2 STRUKTURALNI DIZAJN PATERNI

2.1 ADAPTER

U suštini, svaka komponenta Viewmodela je predstavnik adapter paterna. Same interakcije između pogleda i modela još nisu do kraja implementirane u projektu, ali su tipičan primjer povezivanja kakvo vrši adapter. Adapterom se i nadogradnja novih pogleda čini lakšom, u smislu da ne moramo dirati biznis logiku klasa da bismo dodali novu stranicu u layout.

2.2 FAÇADE

Fasada obezbjeđuje pojednostavljeni pogled na podsistem (poput abstract factory), kakvih u suštini nemamo u projektu, tako da nema mogućih primjena.

2.3 DECORATOR

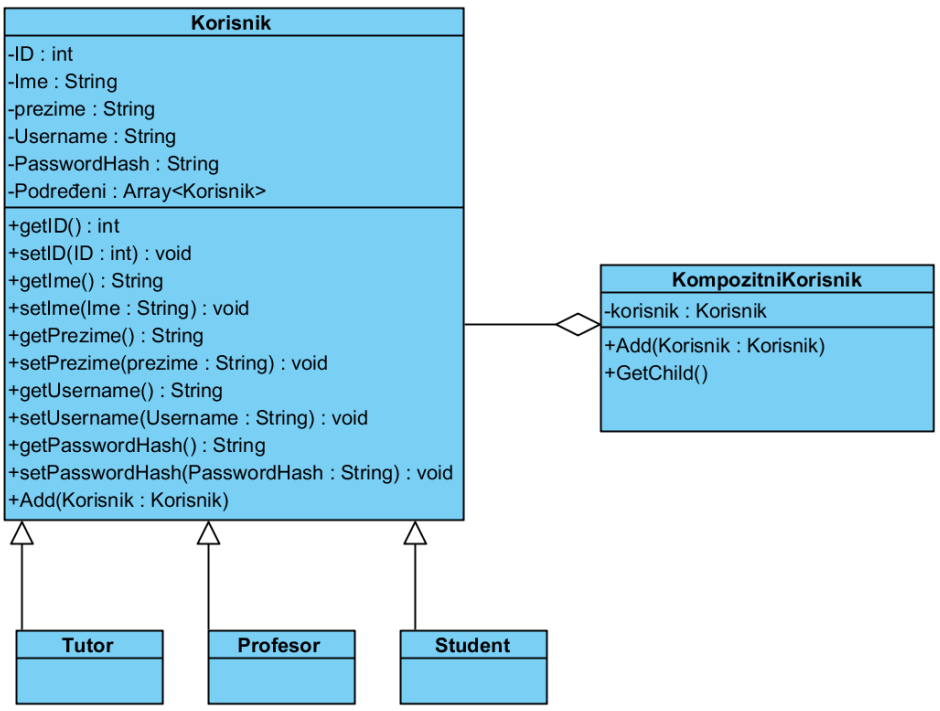
Nema razloga za korištenje dekoratera, pošto se svaka instanca klase tretira isto, te predstavlja red u tabeli, samo bi narušilo strukturu interne logike, iako direktno ne mijenja originalni objekat.

2.4 BRIDGE

Kako promjene u implementaciji ne bi trebale imati utjecaja na klijenta, korisno je bridge patternom razdvojiti implementacionu logiku od interfejsa. Ovo bi mogao biti neki sloj apstrakcije između modela i viewmodela.

2.5 COMPOSITE

Kompozitni patern će biti veoma koristan kad se (i ako) bude radilo proširivanje aplikacije za upotrebu studentima, jer može implementirati hijerarhiju korisnika. To bi izgledalo otprilike ovako:



2.6 PROXY

Proxy je dobar za kontrolu pristupa i slične zaštite, naročito ako je klasa koju proxy implementira sakrivena u libraryju (recimo dll). Obzirom da je login pomoću RFID kartice trenutno hardkodiran (odnosno vrijednosti kartica), ovaj patern bi se mogao iskoristiti za poboljšanje tog dizajna. Vanjski čitač se ponaša identično kao i tastatura, te je potrebno programski odrediti o kakvom se ulazu zapravo radi, te izbjeći da osoba koja nije vlasnik kartice jednostavno unese šifru ručno.

2.7 FLYWEIGHT

Nema potrebe za smanjenjem potrošnje memorije, aplikacija je sama po sebi dovoljno lagana.

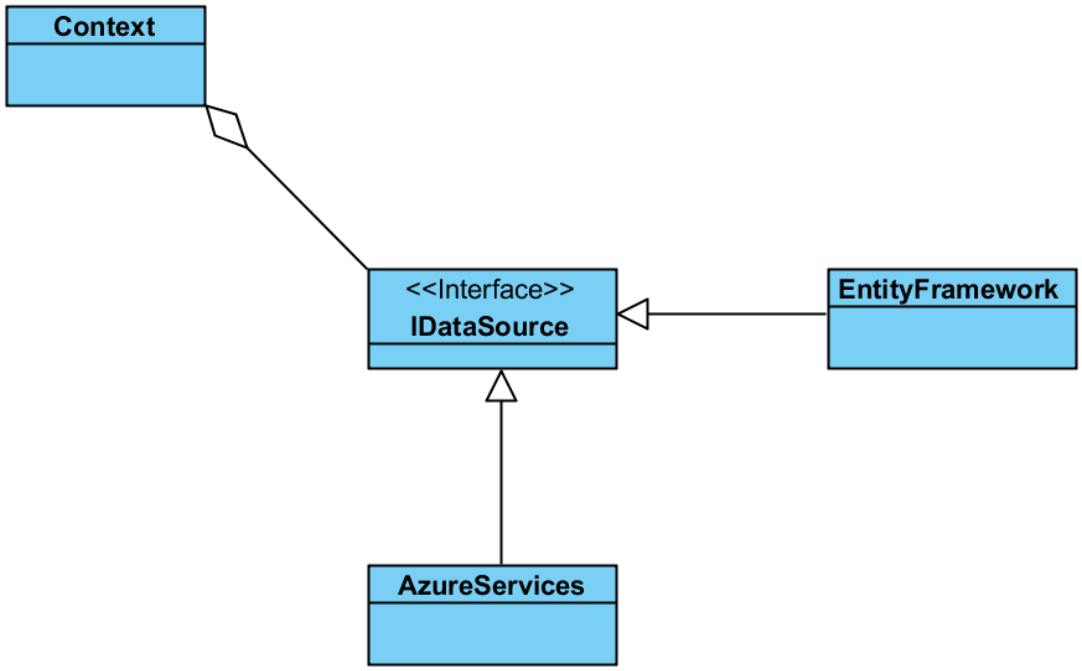
3 PATERNI PONAŠANJA

3.1 STRATEGY

Strategy patern bi se mogao iskoristiti pri implementaciji data access layera. Konkretno, u našem projektu postoje dva izvora podataka – lokalna sqlite baza i web servis azure. U ovom slučaju bi korisnik bi birao odakle se dobavljaju podaci, što nije sasvim intuitivno ponašanje aplikacije.

3.2 STATE

Ovaj patern je već primjenjiv u istom kontekstu kao i prethodni, sa logičnijim slijedom događaja, što se vidi na slici ispod.



Recimo, da se u zavisnosti od stanja podaci dobavljaju sa interneta ili iz lokalne baze. Intuitivno – ukoliko je uređaj spojen na internet koristi se AzureServices, a ako nije, koristi se EntityFramework.

3.3 TEMPLATE METHOD

Naš projekat nema složenih algoritama koje je potrebno razdvajati u posebne klase.

3.4 OBSERVER

Observer patern bi se mogao primijeniti na isti način kao i state patern, uz dinamičku detekciju promjene stanja mreže.

3.5 ITERATOR

Projekat ne sadrži kolekcije podataka, svi podaci se pribavljaju dinamički, pa nema smisla implementirati iterator.