OOAD Projekat – Playoff
Dokumentacija za Refaktoring

Nastavna grupa: RI2 – Grupa 9
Naziv projekta: Playoff
Naziv tima: C--

Članovi tima:

- Mujić Sulejman (17873)
- Muminović Amir (17744)
- Salman Haris (17822)

Tokom dizajniranja i implementiranja aplikacije, trudili smo se da što kvalitetnije pišemo kod. Primjenjeno je znanje iz prethodno savladanih predmeta kao što su ASP i RPR kako bi se pisao (relativno) efikasan i lagano nadogradiv kod koristeći objektno orijentisane principe. Prilikom čitanja mogućih refaktoringa, neke izmjene su pravljene neposredno nakon prvih verzija nekih klasa, tako da ćemo u sljedećih par stavki navesti na koji način je izvršen taj refaktoring.

1) Zamjena „magičnih brojeva“ imenovanim konstantama

Kratkim pregledom koda se vidi da su sve bitne konstante implementirane striktno kao imenovane konstante, te im je dodijeljeno ime po kojem je lagano naslutiti njihovu ulogu.

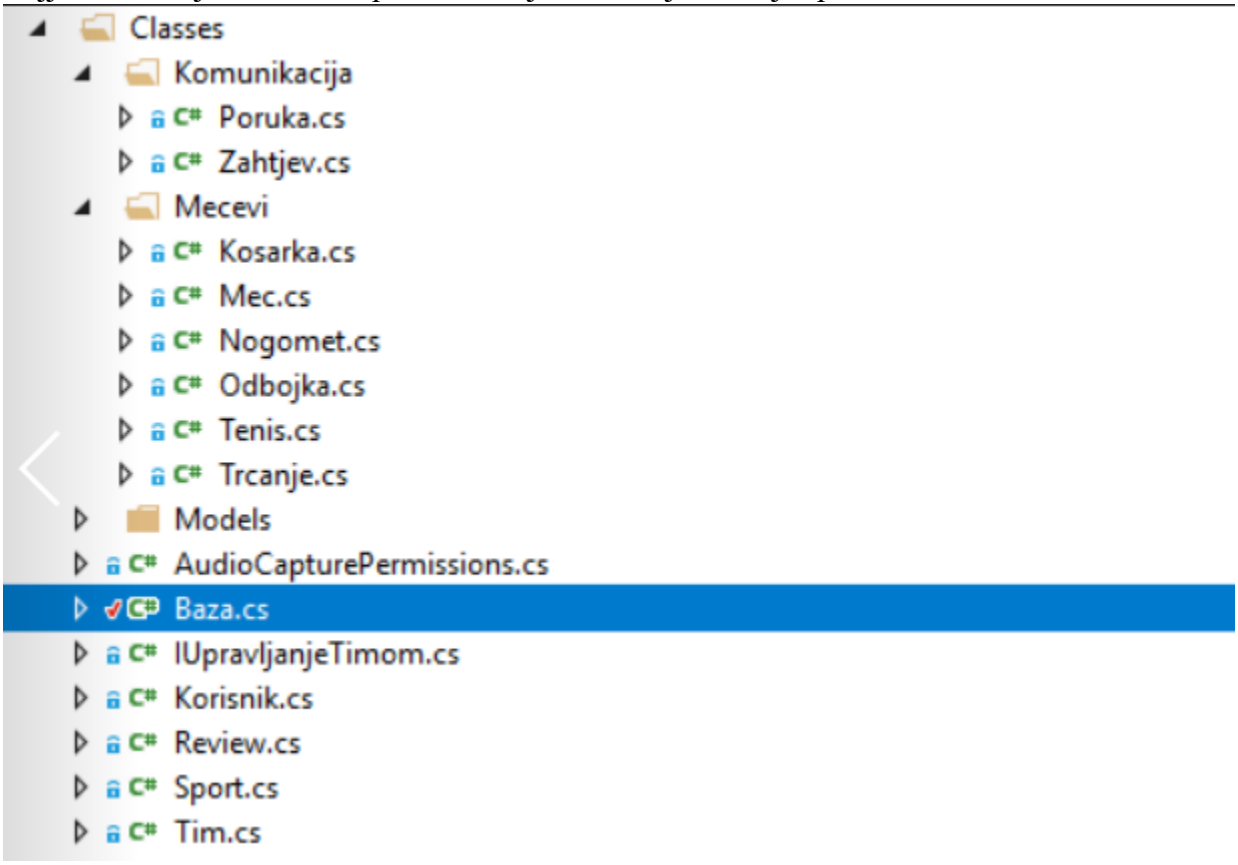
```
// Web API
static string[] podaci = new string[12]{"OOADKorisnicis","OOADTimovis","OOADProsliTimovis","OOADPorukas",
    "OOADSports","OOADMecs","OOADRezultats","OOADReviews","OOADNaziviPozicijas","OOADClanoviTimas","OOADSampionats","OOADZahtjevs" };

        static public async Task<List<OOADSampionat>> DajSampionate() {
            return JsonConvert.DeserializeObject<List<OOADSampionat>>(await Povrat(podaci[10]));
        }


        static public async Task<int> DajID() {
            var kor = await DajKorisnike();
            foreach (var x in kor) if (x.Username.ToLower() == Logged1) return x.ID;
            return -1;
        }

        static public async Task<List<OOADZahtjev>> DajPrimljeneZahtjeve() {
            var x = JsonConvert.DeserializeObject<List<OOADZahtjev>>(await Povrat(podaci[11]));
            List<OOADZahtjev> vr = new List<OOADZahtjev>();
            foreach (var y in x) if (y.Primaoc == ID1) vr.Add(y);
            return x;
        }

        static public async Task<List<OOADZahtjev>> DajPoslaneZahtjeve() {
            var x = JsonConvert.DeserializeObject<List<OOADZahtjev>>(await Povrat(podaci[11]));
            List<OOADZahtjev> vr = new List<OOADZahtjev>();
            foreach (var y in x) if (y.Posiljaoc == ID1) vr.Add(y);
            return x;
```

Na slici vidimo primjenu imenovanih konstanti u klasi za bazu podataka kako bi se koristio WEB API dio aplikacije. Ovo nam ujedno dozvoljava da na jednostavan način vršimo eventualne promjene koda, tako da pri promjeni jedne metode ne brinemo o funkcionalnosti drugih.

Svi članovi tima su radili na pravilnoj upotrebi konstanti pri pisanju koda aplikacije.

2) Klase su dizajnirane za dobro definisan, povezan skup funkcionalnosti

Najjednostavniji način da se prikaže namjenski dizajn klasa je spiskom svih klasa:



Vidimo da je svaka klasa dizajnirana da podržava tačno jedan dio funkcionalnosti, i ne dolazi do preklapanja funkcionalnosti između više različitih klasa. Ovakvom implementacijom klasa smo omogućili ne samo urednost koda, već i drastično smanjenje pojedinih zavisnosti klasa. Također, možemo vrlo jednostavno mijenjati klase a da to ni na kakav način ne utiče na ostale klase, tako da nam je poprilično lagano ubaciti dodatne funkcionalnosti u slučaju da ih korisnici traže.

Haris i Amir su najviše radili na dizajnu klasa.

3) Spisak razloga za refaktoring, i zašto se oni ovdje ne javljaju

Nakon što smo detaljno naveli dva tipa refaktoringa, ukratko ćemo preći kompletan spisak problema koji predstavljaju razloge za refaktoring pokrivenih u predavanju 6, te ćemo za svaki problem dati postojeće rješenje:

1) Hijerarhija nasljeđivanja se paralelno modificira / Podijeljene promjene unutar klase – Već je objašnjeno kako su klase od početka dizajnirane što manje ovise jedna od druge, te svaka metoda klase radi tačno ono što se od nje očekuje. Nasljeđivanje se zapravo koristi samo za specijalizaciju različitih mečeva, te kod Zahtjeva koji je nadogradnja od Poruke.
2) Neispravna hijerarhija nasljeđivanja – Već objašnjeno kroz druge primjere.
3) Posrednik ne radi ništa – Ne postoje rutine koje samo zovu druge, a da nisu smisleno ubačene radi olakšane upotrebe.
4) Duplikacija koda – Ne javlja se u aplikaciji, svi često korišteni blokovi koda su smješteni u metode klasa.
5) Rutina ima loše ime – Svaka rutina je imenovana tako da je se lagano sjetiti i pozvati po potrebi.
6) Preduge rutine – Sve rutine su kratke i rade tačno ono što njihov naziv nalaže.
7) Podklasa koristi samo mali procenat svojih roditelj rutina – Već objašnjeno kroz druge primjere.
8) Nekonzistentan nivo apstrakcije – Već objašnjeno kroz druge primjere.
9) Preduboke i preduge petlje – Ne postoje petlje sa većom složenosti od $O(n^2)$, niti petlje koje su duže od 10 linija koda.
10) Parametar lista sa previše parametara – Parametar liste su uglavnom kratke, oko 1-2 parametra.
11) Promjene zahtijevaju paralelne modifikacije više klasa – Promjene utiču samo na datu klasu.
12) Case iskazi se modificiraju paralelno – Ne postoje switch-case iskazi, koristi se polimorfizam.
13) Povezani podaci koji se koriste zajedno nisu organizirani u klasu – Klase su ispravno dizajnirane.
14) Rutina koristi više karakteristika druge klase nego svoje klase – Rutine skoro isključivo pozivaju svoje metode.
15) Primitivni podaci su preklopljeni – Nije bilo potrebe za ovim.
16) Klasa ne radi puno – Već objašnjeno kroz druge primjere.
17) Lanac rutina prima tramp podatke – Skoro nigdje se ne javljaju lanci rutina.
18) Globalne varijable se koriste – Jedina „globalna“ varijabla je informacija o trenutno prijavljenom korisniku.
19) Podaci su public – Podacima se pristupa preko gettera i settera.
20) Komentari se koriste da objasne težak kod – Komentari se isključivo koriste da se objasni sama funkcionalnost neke metode, kod je čitljiv i razumljiv.

Kao što je već rečeno, svi članovi tima su radili na pisanju ispravnog, efikasnog koda.