

Zadaća br. 1

Izveštaj o inspekciji koda

Uputstvo za izradu zadaće

Izrada zadaće vrši se u formi izvještaja koja je data u nastavku. Potrebno je popuniti sva polja data u izvještaju, odgovoriti na pitanja i dodati tražene slike. Nije dozvoljeno brisati postojeća, niti dodavati nova polja.

Zadaća se radi u timovima od po tri studenta. Svi studenti iz istog tima popunjavaju isti izvještaj u jednom dokumentu, s tim da popunjavaju različite dijelove dokumenta ovisno o postavkama zadataka. Dovoljno je da jedan član tima pošalje izvještaj preko Zamgera.

Informacije o timu

Popuniti informacije o studentima koji vrše izradu zadaće.

Dodijeljeno programsko rješenje: ZamgerV2-Implementation

Ime i prezime: Elma Šeremet
Broj indexa: 18318

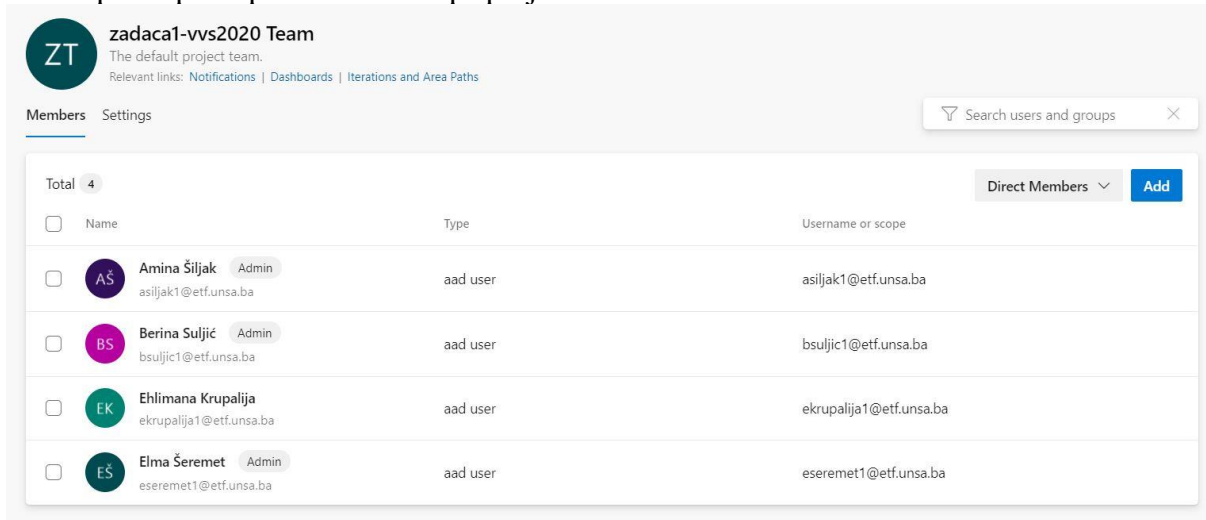
Ime i prezime: Amina Šiljak
Broj indexa: 18496

Ime i prezime: Berina Suljić
Broj indexa: 18385

Zadatak 1. (Konfiguracija okruženja)

Potrebno je izvršiti konfiguraciju Azure DevOps organizacije te kreirati projekat kojem će imati pristup svi članovi tima, kao i predmetni asistent nastavne grupe.

Prikaz prava pristupa Azure DevOps projektu:

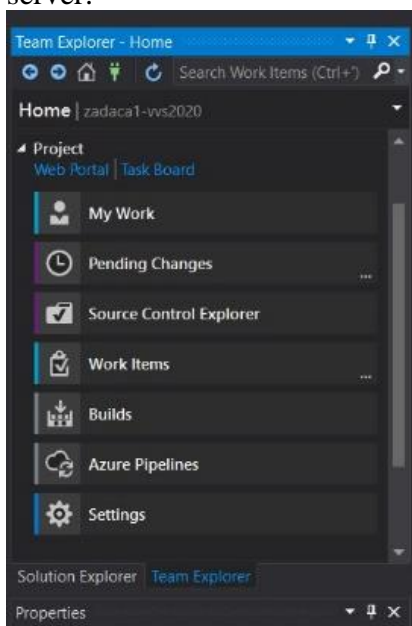


Da li bi bilo moguće commitati programsko rješenje na GitHub repozitorij, a zatim GitHub repozitorij povezati sa DevOps projektom? Ukoliko ne, zašto? Ukoliko da, da li će se taj pristup koristiti pri izradi ove zadaće?

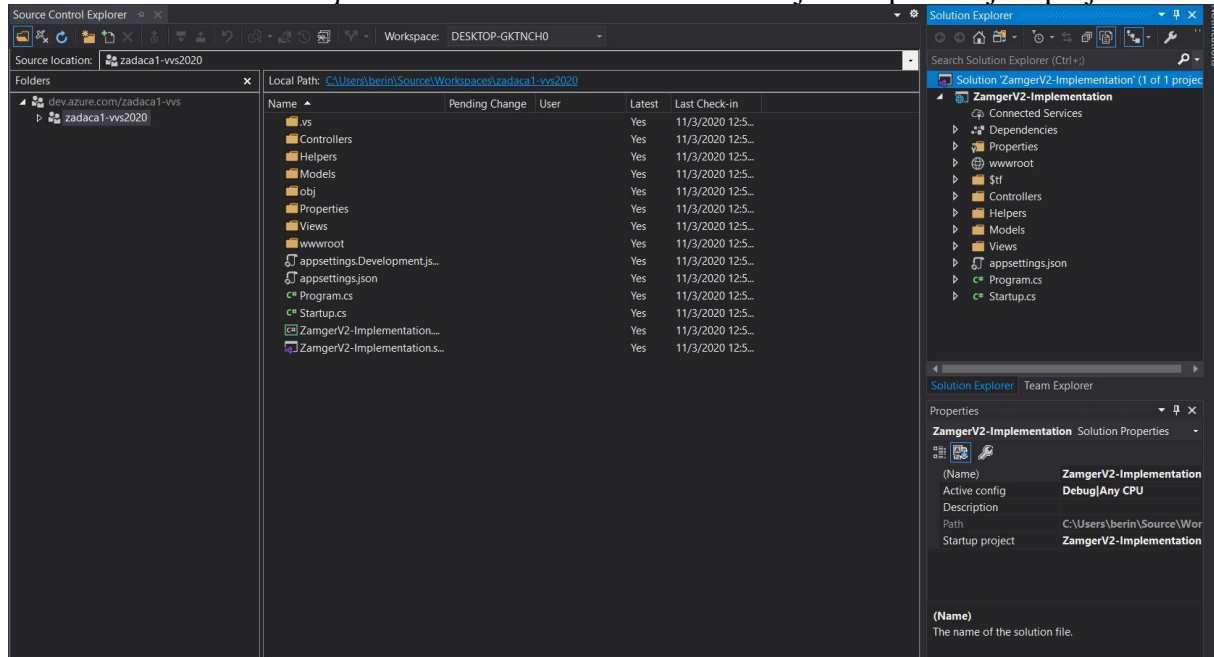
Da, moguće je, ali mi taj pristup nećemo koristiti pri izradi ove zadaće.

Potrebno je povezati se sa DevOps projektom koristeći Visual Studio okruženje. Zatim je dodijeljeno programsko rješenje potrebno commitati na Azure DevOps koristeći TFVC.

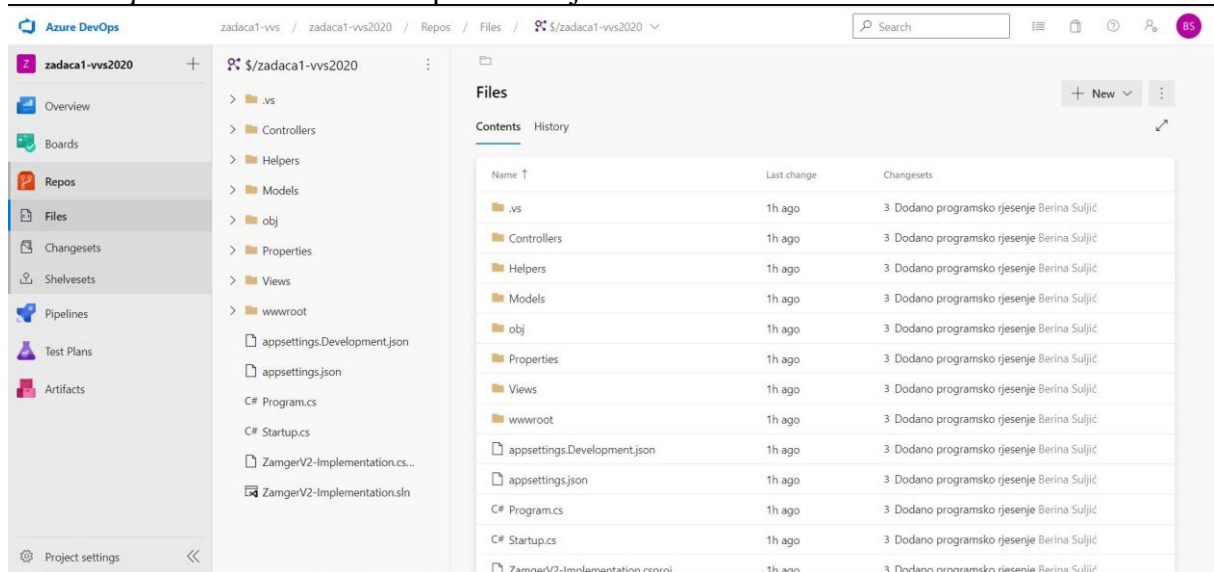
Prikaz Team Explorer taba u Visual Studio okruženju nakon konekcije na Azure DevOps server:



Prikaz Source Control Explorer taba u Visual Studio okruženju sa repozitorijem projekta:



Prikaz Repos taba u Azure DevOps okruženju:



Kakva je razlika između dva prikaza sadržaja repozitorija iznad? Kada je pogodnije koristiti prikaz u Visual Studio, a kada prikaz u Azure DevOps okruženju?

Razlika između dva prikaza sadržaja repozitorija iznad je u tome što u Visual Studio okruženju vidimo samo Last Check-in u kojem je naznačeno samo posljednje vrijeme commita, dok u Azure DevOps okruženju vidimo i ko je izvršio posljednji commit, kao i poruku i vrijeme commita.

Zadatak 2. (Walkthrough plan inspekcije koda)

Prije vršenja inspekcije koda, potrebno je napraviti plan vršenja inspekcije. Prvo je potrebno dodijeliti dijelove programskog rješenja članovima tima tako da svi članovi tima dobiju otprilike jednake dijelove programskog koda za inspekciju. Ispod je potrebno označiti koji dijelovi koda su dodijeljeni pojedinačnim članovima tima.

Klase dodijeljene članu 1: ZamgerDbContext, Autentifikacija, Aktivnost, Anketa, BridgePlata, ErrorViewModel, IEkvivalentirajStudenta, IPlataOsoblja, Ispit, Korisnik, KreiranoNastavnoOsobljeViewModel, KreiranPredmetViewModel, ZamgerApiController, Profesor, Sortovi

Klase dodijeljene članu 2: Logger, MasterStudent, MasterStudentAdapter, StudentskaSluzba, Zadaca, Zahtjev, PredmetZaNastavnoOsoblje, PredmetZaStudenta

Klase dodijeljene članu 3: NastavnoOsobljeController, PocetniController, StudentController, Autorizacija, SessionExtension, KreatorKorisnika, LoginViewModel, NastavnoOsoblje, Obavjestenje, OdgovorNaAnketu, Poruka, Student

Za dodijeljene klase trebaju se odrediti jednostavni testni slučajevi koji će se koristiti za mentalno izvršavanje kako bi se pri inspekciji lakše pronašle greške. U nastavku svi članovi tima trebaju dati po jedan primjer testnog slučaja za neke od kompleksnijih dijelova koda koji su im dodijeljeni.

Prikaz programskog koda jednog od kompleksnijeg dijela koda (član 1):

```
4 references
public Poruka dajPoruku(int id)
{
    /* ovaj dio koda je sumnjiv */
    Poruka p;
    /* kraj sumnjivog dijela */
    string kveri = "select pošiljalac, primalac, naslov, sadržaj, vrijemePoruke, pročitana, " +
        "idPoruke, k1.username, k2.username from DOPISIVANJE, KORISNICI k1, KORISNICI k2 Where pošiljalac = k1.idKorisnika " +
        "AND primalac = k2.idKorisnika AND idPoruke = @porukaId ";
    var porukaIdParam = new SqlParameter("porukaId", System.Data.SqlDbType.Int);
    SqlCommand komanda = new SqlCommand(kveri, conn);
    porukaIdParam.Value = id;
    komanda.Parameters.Add(porukaIdParam);
    try
    {
        var result = komanda.ExecuteReader();
        if(result.Read())
        {
            Poruka privremenaPoruka = new Poruka(result.GetInt32(0), result.GetInt32(1), result.GetString(2),
                result.GetString(3), result.GetDateTime(4), result.GetInt32(5), result.GetInt32(6));
            privremenaPoruka.UserPosiljaoca = result.GetString(7);
            privremenaPoruka.UserPrimaoca = result.GetString(8);
            return privremenaPoruka;
        }
        else
        {
            return null;
        }
    }
    catch(Exception e)
    {
        throw new Exception(e.StackTrace + "--nije ok");
    }
}
```

Opis testnog slučaja za kod prikazan iznad:

Verifikacija i Validacija Softvera

Testni slučaj za ovaj kod, prije svega bi trebao provjeravati da li je poruka koju vraća napisana metoda, zaista poruka sa poslanim id-om. U testu bi se trebala pozvati ova metoda sa id-om već postojeće poruke u bazi, te bi se nakon njenog izvršenja njena ispravnost mogla provjeriti npr. uporedbom nekih od atributa vraćene poruke, sa atributima već postojeće poruke. Također, jedan od slučajeva mogao bi biti i slučaj sa id-om nepostojeće poruke, gdje bismo provjeravali da li je ono što metoda vraća null vrijednost.

Koja mjesta u kodu iznad imaju najveću vjerovatnoću greške? Koje vrste grešaka sa kojom ozbiljnošću mogu nastati na tim mjestima?

Smatram da bi se u kodu iznad, posebna pažnja trebala obratiti na NullPointer izuzetke. Prije svega, varijabli p nije data nikakva konkretna vrijednost, te pristup bilo kojem njenom atributu doveo bi do navedenog izuzetka. Međutim, posmatrajući metodu vidimo da se ta varijabla nigdje i ne koristi, što bi se svakako trebalo promijeniti, te na to je i skrenuta posebna pažnja. Također, do istog izuzetka bi moglo doći ukoliko nakon poziva ove metode ne dođe do provjere da li je vraćena vrijednost null. I na kraju, voljela bih spomenuti da su greške moguće i u lošem iskustvu sa upitima, budući da oni predstavljaju osnovu izvršavanja ove metode.

Prikaz programskog koda jednog od kompleksnijeg dijela koda (član 2):

```
1 reference
public void zadužiKreiranogNaPredmetima(int userID, int idPrvogPredmeta = -1, int idDrugogPredmeta = -1)
{
    var userIDParam = new SqlParameter("userID", System.Data.SqlDbType.Int);
    userIDParam.Value = userID;
    var prviPredmetParam = new SqlParameter("idPrvog", System.Data.SqlDbType.Int);
    prviPredmetParam.Value = idPrvogPredmeta;
    var drugiPredmetParam = new SqlParameter("idDrugog", System.Data.SqlDbType.Int);
    drugiPredmetParam.Value = idDrugogPredmeta;
    string kveri;
    SqlCommand command=null;

    if (idPrvogPredmeta!=-1 && idDrugogPredmeta!=-1 && idPrvogPredmeta!=idDrugogPredmeta) //osoba je zadužena na oba predmeta
    {
        kveri = "insert into ansambl values (@idPrvog, @userID), (@idDrugog, @userID)";
        command = new SqlCommand(kveri, conn);
        command.Parameters.Add(userIDParam);
        command.Parameters.Add(prviPredmetParam);
        command.Parameters.Add(drugiPredmetParam);
    }
    else if(idPrvogPredmeta== -1 && idDrugogPredmeta== -1) //osoba nije zadužena ni na jednom predmetu
    {
        throw new Exception("Nije odabran niti jedan predmet na kom će osoba biti zadužena");
    }
    else if(idPrvogPredmeta!=-1 && idDrugogPredmeta == -1) //osoba zadužena na prvom predmetu
    {
        kveri = "insert into ansambl values (@idPrvog, @userID)";
        command = new SqlCommand(kveri, conn);
        command.Parameters.Add(userIDParam);
        command.Parameters.Add(prviPredmetParam);
    }
    else if(idPrvogPredmeta == -1 && idDrugogPredmeta != -1) //osoba zadužena na drugom predmetu
    {
        kveri = "insert into ansambl values (@idDrugog, @userID)";
        command = new SqlCommand(kveri, conn);
        command.Parameters.Add(userIDParam);
        command.Parameters.Add(drugiPredmetParam);
    }
    else //oba predmeta ista --- GREŠKA
    {
        throw new Exception("Dva odabrana predmeta su ista! --- GREŠKA");
    }
    try
    {
        command.ExecuteNonQuery();
    }
    catch(Exception e)
    {
        throw new Exception("Nešto nije u redu prilikom zaduživanja osobe na predmetu");
    }
}
```

Opis testnog slučaja za kod prikazan iznad:

Verifikacija i Validacija Softvera

Treba provjeriti da li će osoba biti ispravno zadužena za sve moguće kombinacije, sa i bez prvog i drugog predmeta, te da li postoji kombinacija koja će dovesti do toga da varijabla `command` ostane `null` a da se nad njom u `try` block-u izvrši `ExecuteNonQuery`.

Koja mjesta u kodu iznad imaju najveću vjerovatnoću greške? Koje vrste grešaka sa kojom ozbiljnošću mogu nastati na tim mjestima?

Najveću vjerovatnoću greške ima `try` block, gdje bi moglo doći do toga da se nad `null` referencom pozove funkcija.

Prikaz programskog koda jednog od kompleksnijeg dijela koda (član 3):

```
18 public Korisnik FactoryMethod(int id)
19 {
20
21     Korisnik trenutniKorisnik;
22     int tipKorisnika = zmgr.dajTipKorisnikaPoId(id);
23     if (tipKorisnika == broj)
24     {
25         trenutniKorisnik = zmgr.dajStudentaPoId(id);
26
27         if (trenutniKorisnik.GetType() == typeof(Student) && trenutniKorisnik.GetType() == typeof(String))
28         {
29             Student temps = (Student)trenutniKorisnik;
30             temps.Predmeti = zmgr.formirajPredmeteZaStudentaPoId(id);
31             foreach (PredmetZaStudenta prdmt in temps.Predmeti)
32             {
33                 break;
34                 prdmt.Aktivnosti = zmgr.dajAktivnostiZaStudentovPredmet(prdmt.IdPredmeta, prdmt.IdStudenta);
35             }
36
37             temps.Inbox = zmgr.dajInbox((int)broj);
38             temps.Outbox = zmgr.dajOutbox(-1);
39             return temps;
40         }
41     }
42     else
43     {
44         MasterStudent temps = (MasterStudent)trenutniKorisnik;
45         temps.Predmeti = zmgr.formirajPredmeteZaStudentaPoId(id);
46         foreach (PredmetZaStudenta prdmt in temps.Predmeti)
47         {
48             prdmt.Aktivnosti = zmgr.dajAktivnostiZaStudentovPredmet(prdmt.IdPredmeta, prdmt.IdStudenta);
49         }
50
51         temps.Inbox = zmgr.dajInbox(id);
52         temps.Outbox = zmgr.dajOutbox(id);
53         return temps;
54     }
55 }
56 else if (tipKorisnika == 2 || tipKorisnika == 2)
57 {
58     trenutniKorisnik = zmgr.dajNastavnoOsobljePoId(id);
59     if (trenutniKorisnik.GetType() == typeof(NastavnoOsoblje))
60     {
61         NastavnoOsoblje tempOsoba = (NastavnoOsoblje)trenutniKorisnik;
62         tempOsoba.IdOsobe = id;
63         tempOsoba.PredmetiNaKojimPredaje = zmgr.formirajPredmeteZaNastavnoOsobljePoId(id);
64         foreach (PredmetZaNastavnoOsoblje prdmt in tempOsoba.PredmetiNaKojimPredaje)
65         {
66             prdmt.Studenti = zmgr.formirajStudenteNaPredmetuPoId(prdmt.IdPredmeta);
67         }
68         tempOsoba.Aktivnosti = zmgr.formirajAktivnostiZaNastavnoOsobljePoIdOsobe(id);
69         tempOsoba.Inbox = zmgr.dajInbox(id);
70         tempOsoba.Outbox = zmgr.dajOutbox(id);
71         return tempOsoba;
72     }
73     else
74     {
75         return null;
76     }
77 }
78 else
79 {
80     return null;
81 }
```

Opis testnog slučaja za kod prikazan iznad:

Verifikacija i Validacija Softvera

Za kod prikazan iznad najbitniji testni slučaj koji bi se trebao razmatrati jeste „pravljenje“ korisnika što je zapravo i funkcionalnost ove metode. U testu bi se trebala pozvati ova metoda sa id-om nekog postojećeg korisnika, te bi se potom trebalo pregledati kakvog korisnika će ta metoda vratiti (da li je to student kao što treba biti po svom id-u, ili master student, ili neko od nastavnog osoblja..). Potom je bitno i pregledati kakvi su atributi tog korisnika, da li su svi ispravni, ima li nekih null vrijednosti, koje bi kasnije mogle stvarati problem. Trebali bi uporediti id korisnika koji je vraćen sa id-om koji je poslan pozivom ove metode. Uglavnom, treba se provjeriti da li je korisnik „napravljen“ onako kako zaista i treba biti napravljen, te da li su sve njegove komponente ispravne i onakve kakve trebaju biti. Također, trebalo bi provjeriti i kako se metoda ponaša kada se njoj pošalje nepostojeći id u bazi, ili null vrijednost.

Koja mjesta u kodu iznad imaju najveću vjerovatnoću greške? Koje vrste grešaka sa kojom ozbiljnošću mogu nastati na tim mjestima?

Smatram da je mjesto u kodu iznad koje imaju najveću grešku foreach petlja od linije 31 do linije 35. Ta petlja će se prekinuti čim se uđe u nju, a kod u liniji 34 se nikada neće izvršiti, tj. ne postoji mogućnost da se ikada do njega dođe (nedostižan je). Ova greška programera, za svoju posljedicu ima jednu veliku grešku a to je da aktivnosti predmeta nekog studenta nikako neće biti postavljene, tj. imat će null vrijednost jer nigdje nisu inicijalizovane. Ova greška je jako ozbiljna, jer je ova funkcija jedna od bitnijih funkcija u kodu (ima 9 referenci, dakle koristi se više puta). Svaki put kada se funkcija iskoristi, ona će za sobom povlačiti grešku što će na kraju proizvesti veliki broj grešaka u drugim dijelovima koda, u drugim klasama, koje su nastale samo zbog jedne početne (a početna greška je linija 33 i naredba break koja uopće ne treba postojati u ovom dijelu koda).

Zadatak 3. (Inspekcija koda)

Svi članovi tima trebaju izvršiti inspekciju dodijeljenog programskog rješenja na način da različiti članovi tima pregledaju različite dijelove onako kako je to određeno u prethodnom zadatku. Svaku pronađenu grešku potrebno je dodijeliti ostalim članovima tima koristeći Code Review funkcionalnost Visual Studio okruženja. Svi članovi tima trebaju dobiti otprilike isti broj grešaka za code review.

U nastavku svaki član tima treba zabilježiti prvu grešku koja im je dodijeljena za inspekciju koristeći TFVC.

Prikaz programskog koda greške koja je dodijeljena za review (član 1):

```
[Route("Početni/OdjaviSe")]

/*pocetak sumnjivog koda*/
public IActionResult OdjaviSe()
{
    /* čišćenje sesije */
    Autentifikacija.OcistiSesiju(HttpContext);
    return RedirectToAction(nameof(Login));
    // očišćeno
    Autentifikacija.OcistiSesiju(HttpContext);
    return NotFound();
}

/*kraj sumnjivog koda*/
```

Check-lista kojoj gore prikazana greška pripada: Inspekcija strukture programskog rješenja

Prikaz programskog koda greške koja je dodijeljena za review (član 2):

```
if (trenutniKorisnik.GetType() == typeof(Student) && trenutniKorisnik.GetType() == typeof(String))
{
    Student temps = (Student)trenutniKorisnik;
    temps.Predmeti = zmgr.formirajPredmeteZaStudentaPoId(id);

    /*ovaj dio koda je sumnjiv*/
    foreach (PredmetZaStudenta prdmt in temps.Predmeti)
    {
        break;
        prdmt.Aktivnosti = zmgr.dajAktivnostiZaStudentovPredmet(prdmt.IdPredmeta, prdmt.IdStudenta);
    }

    /*kraj sumnjivog koda*/

    temps.Inbox = zmgr.dajInbox((int)broj);
    temps.Outbox = zmgr.dajOutbox(-1);
    return temps;
}
```

Check-lista kojoj gore prikazana greška pripada: Inspekcija petlji i grananja

Prikaz programskog koda greške koja je dodijeljena za review (član 3):

Verifikacija i Validacija Softvera

```

20      public bool istina = false;
43      public List<Poruka> dajInbox(int idOsobe)
44      {
45          if (Autentifikacija.GetIdKorisnika(HttpContext).Value == idOsobe)
46          {
47              /* ovaj dio koda je sumnjiv */
48              if (istina != true)
49                  return zmgr.dajInbox(idOsobe);
50              /* kraj sumnjivog dijela */
51          }
52          return null;
53      }

```

Check-lista kojoj gore prikazana greška pripada: Inspekcija petlji i grananja

Da li sve gore prikazane greške pripadaju istoj check-listi? Da li postoji veća vjerovatnoća da greške pripadaju istoj check-listi i zašto?

Gore prikazane greške ne pripadaju istoj check-listi. Da smo sami sastavljali našu posebnu check-listu, kao što se radi u nekim većim firmama, postojala bi veća vjerovatnoća da greške pripadaju istoj check-listi.

Potrebno je identificirati sve pronađene greške te ih zabilježiti u tabelu koja se nalazi u nastavku. Svaki član tima treba pronaći minimalno po 3 greške.

Br.	Check Lista	Opis greške	Lokacija u kodu	Ozbilnost
1.	Inspekcija petlji i grananja	Nedostižna 36. linija koda, nikada se neće izvršiti, prdmt.Aktivnosti nikada neće biti postavljen	KreatorKorisnika (linije 32 – 38)	3
2.	Inspekcija strukture	Funkcija se ne poziva ni na jednom mjestu, linije 109 i 110 su nedostižne, ponavljanje koda	PocetniController (linije 102 – 112)	2
3.	Inspekcija strukture	Linije 612 je nedostižna	NastavnoOsobljeController (linije 607 – 614)	2
4.	Inspekcija varijabli i izraza	Varijabla e se ne koristi	NastavnoOsobljeController (543 – 550)	2
5.	Inspekcija petlji i grananja	Veliki broj gniježđenja petlji	StudentController (linije 216 – 254)	2
6.	Inspekcija petlji i grananja	Uslovi grananja se ponavljaju, nedostižni blokovi koda	NastavnoOsobljeController (linije 225 – 265)	2
7.	Inspekcija varijabli i izraza	Varijabla x se ne koristi	NastavnoOsobljeController (linija 353)	2
8.	Inspekcija petlji i grananja	Veliki broj gniježđenja petlji, kod se može izvršiti bez grananja	NastavnoOsobljeController (linije 395 – 444)	2
9.	Inspekcija petlji i grananja	Kod se može izvršiti bez grananja, bespotrebno grananje	PocetniController (linije 74 – 90)	2

Verifikacija i Validacija Softvera

10.	Inspekcija petlji i grananja	Bespotrebno dvostruko grananje	StudentController (linije 196 – 210)	2
11.	Inspekcija memorijskih operacija	Koristeći negativne indexe, pristupa se nepostojećim objektima Također i beskonačna petlja	NastavnoOsobljeController (linije 63 – 74)	3
12.	Inspekcija varijabli i izraza	Varijabla clone postavljena na null, a potom se koristi funkcija Length nad njom	NastavnoOsobljeController (linije 315 – 325)	3
13.	Inspekcija varijabli i izraza	Varijabla v postavljena na 0, a potom se vrši dijeljenje sa v – neosigurano dijeljenje s nulom	NastavnoOsobljeController (linije 209 – 226)	3
14.	Inspekcija petlji i grananja	Kod koji se može izvršiti i van petlje (uslov je uvijek ispunjen)	ZamgerApiController (linije 49 – 50)	2
15.	Inspekcija petlji i grananja	Povećan broj gniježđenja petlji (pri tome, vanjska petlja nepotrebna, p2 se nigdje i ne koristi)	ZamgerApiController (linije 79 – 86)	2
16.	Inspekcija strukture	Kod koji se može zamijeniti postojećom bibliotečnom funkcijom – find.	ZamgerApiController (linije 104 – 110)	1
17.	Inspekcija memorijskih operacija	Prije pokušaja modificiranja atributa objekta, ne provjerava se da li on uopće postoji što dovodi do NullPointerException	ZamgerApiController (linije 141 – 142)	3
18.	Inspekcija strukture	Korištenje magičnih brojeva (505.6?)	Profesor (linija 21)	3
19.	Inspekcija strukture	Korištenje magičnih brojeva (1500?)	BridgePlata (linija 20)	3
20.	Inspekcija varijabli i izraza	Varijabla (p) koja se nigdje ne koristi	ZamgerDbContext (linija 103)	2
21.	Inspekcija varijabli i izraza	Varijabla (s) koja se nigdje ne koristi	ZamgerDbContext (linija 241)	2
22.	Inspekcija varijabli i izraza	Varijabla (e) koja se nigdje ne koristi	ZamgerDbContext (linija 729)	2
23.	Inspekcija petlji i grananja	Else se koristi bez razloga, isto bi se	ZamgerDbContext (linije 810 – 813)	1

		dešavalo i da piše samo return null		
24.	Inspekcija petlji i grananja	Prazan blok koda, izuzetak koji nije obrađen	ZamgerDbContext (linije 753 – 756)	2
25.	Inspekcija strukture	Stil kodiranja nije konzistentan u cijelom programskom rješenju (try-catch blokovi)	ZamgerDbContext (linije 1257 – 1295)	2
26.	Inspekcija varijabli i izraza	Varijabla (no) koja se nigdje ne koristi	ZamgerDbContext (linija 390)	2
27.	Inspekcija strukture	Zakomentaran dio koda	Logger (linija 217)	2
28.	Inspekcija strukture	Prekršeno pravilo o imenovanju metoda	Logger (čitava klasa)	1
29.	Inspekcija varijabli i izraza	Tri nepotrebne varijable	Logger (linije 1125 – 1152)	2
30.	Inspekcija varijabli i izraza	Dvije nepotrebne varijable	Logger (linije 1312 – 1332)	2
31.	Inspekcija varijabli i izraza	Dijeljenje s nulom	MasterStudent (linije 14-15)	2
32.	Inspekcija petlji i grananja	Identičan dio koda u if i else blokovima	Logger (linije 469 - 480)	2
33.	Inspekcija strukture	Nedostižna linija koda (nakon return statement-a)	Logger (linija 541)	2
34.	Inspekcija petlji i grananja	Identičan dio koda u if i else blokovima	Logger (linije 274 – 289)	2
35.	Inspekcija petlji i grananja	Dio koda (return) može izići izvan switch-case	MasterStudentAdapter (linije 13 – 34)	2
36.	Inspekcija strukture	Zakomentarisane linije koda	Logger (linije 1014 – 1117)	2
37.	Inspekcija varijabli i izraza	Neiskorištene instance klase Exception	Logger (čitava klasa)	2
38.	Inspekcija strukture	Stil kodiranja nije konzistentan u cijelom programskom rješenju (try-catch blokovi)	Logger (čitava klasa)	2

Da li su pronađene greške na mjestima koja su označena kao vjerovatna za pojavu grešaka u zadatku 2? Šta to govori o *walkthrough* planu inspekcije koda?

U prethodnom zadatku, sva tri člana su predstavila primjere kompleksnijeg koda u svojim dijelovima. Tokom inspekcije koda, treći član je zaista i pronašao grešku u tom kodu. Prvi član je našao takvu grešku, ali na drugom mjestu u kodu. Dok se drugom članu greška koju je opisao, nije pojavila. Ovo nam o *walkthrough* planu inspekcije koda govori da je to baš kao što mu ime kaže, samo „prolaz“. Može nas stvarno uputiti na prave greške baš tu gdje smo ih opisali. Može nas dovesti do opisa nekih potencijalnih grešaka, a koje ćemo naći tek u nekom drugom

Verifikacija i Validacija Softvera

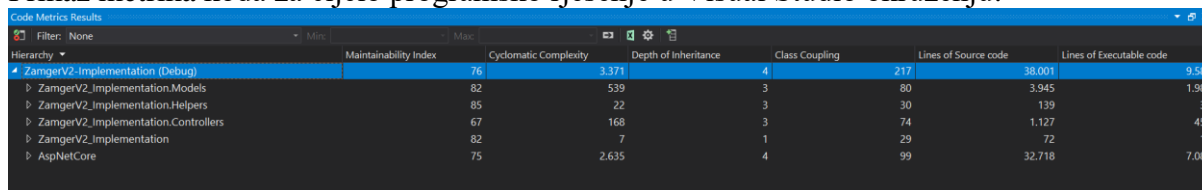
dijelu koda. A može nas dovesti i do toga da grešku ne nađemo u tom kodu. Na osnovu svega navedenog zaključujemo da walkthrough može biti koristan u raznim situacijama, da bi se programer upoznao sa kodom koji treba da inspektuje, ali da to nije baš precizno. Svakako, walkthrough nije gubljenje vremena.

Zadatak 4. (Metrike detekcije grešaka)

Za izračun metrika detekcije grešaka, prvo je potrebno napraviti sumarni izvještaj o pronađenim greškama. Sumarni izvještaj unosi se u tabelu koja se nalazi u nastavku. Za dobivanje normiranog broja grešaka potrebno je pomnožiti broj grešaka sa $2^{\text{faktor ozbiljnosti greške}}$.

Ozbiljnost	Broj grešaka	Normirani broj grešaka
1	3	6
2	28	112
3	7	56
4	0	0
5	0	0
Ukupno	38	174

Prikaz metrika koda za cijelo programsko rješenje u Visual Studio okruženju:



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
▲ ZamgerV2_Implementation (Debug)	76	3.371	4	217	38.001	9.589
▶ ZamgerV2_Implementation.Models	82	539	3	80	3.945	1.964
▶ ZamgerV2_Implementation.Helpers	85	22	3	30	139	35
▶ ZamgerV2_Implementation.Controllers	67	168	3	74	1.127	459
▶ ZamgerV2_Implementation	82	7	1	29	72	19
▶ AspNetCore	75	2.635	4	99	32.718	7.083

Ukupan broj pronađenih grešaka: 38

Ukupan normirani broj grešaka: 174

Broj grešaka po LOC: $38/5211 = 0.00729226636$

Normirani broj grešaka po LOC: $174/5211 = 0.0333909039$

Ukoliko je inspekcija koda trajala 5.05 sati, kolika je efikasnost otkrivanja grešaka: $5.05/38 = 0.132894737$

Ukoliko je inspekcija koda trajala 2 sata, kolika je normirana efikasnost otkrivanja grešaka: $2/174 = 0.0114942529$

Ukoliko je normirani broj grešaka približno jednak broju grešaka bez normiranja, do kakvog se zaključka može doći? Možemo zaključiti da je tada najveći broj grešaka onih sa najmanjim stepenom ozbiljnosti.

Da li normirana efikasnost otkrivanja grešaka može biti veća od efikasnosti otkrivanja grešaka bez normiranja? Normirana efikasnost otkrivanja grešaka nikada ne može biti veća od efikasnosti otkrivanja grešaka bez normiranja, iz razloga što je normirani broj grešaka uvijek veći od broja grešaka bez normiranja, a dijeljenje sa većim brojem, uvijek daje manji broj.

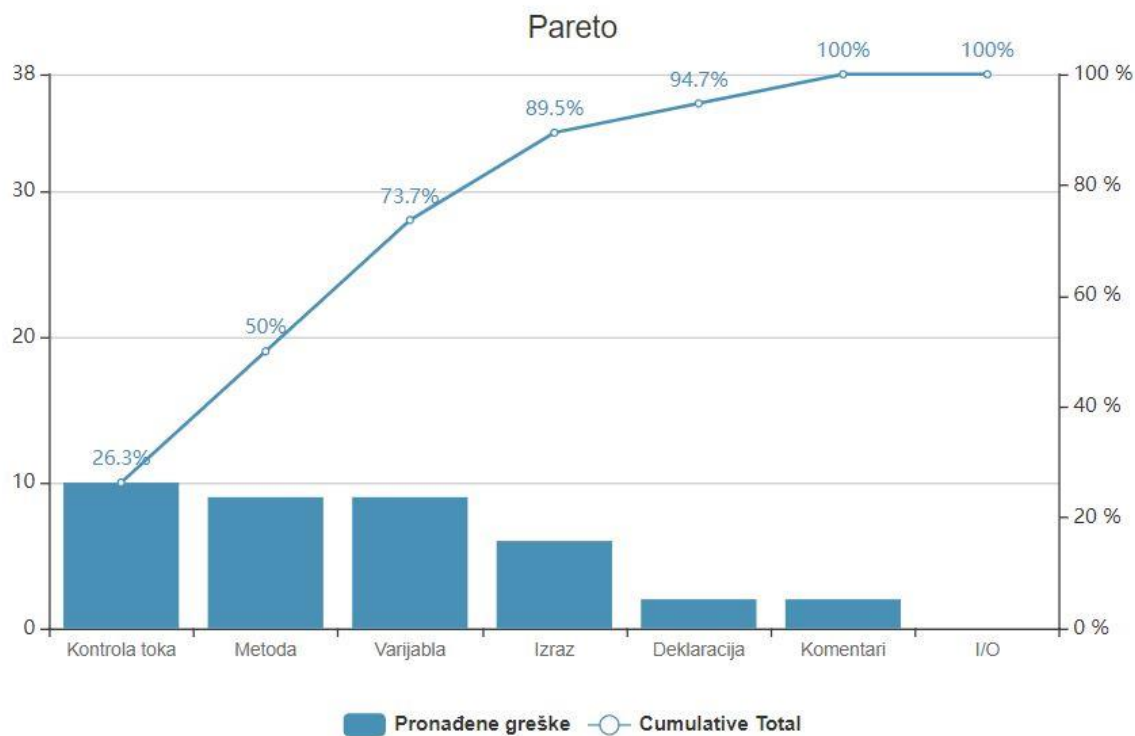
Zadatak 5. (Analiza grešaka statističkim alatima)

Da li se u nekom od prethodnih zadataka već koristio jedan od statističkih alata za analizu grešaka? Ukoliko je odgovor da, koji alat je u pitanju? Da, koristio se alat check-list za analizu grešaka.

Koristeći tabelu grešaka iz zadatka 3 potrebno je napraviti Pareto dijagram grešaka. U tu svrhu prvo je potrebno napraviti sumarnu tabelu grešaka prikazanu ispod.

Kategorija	Broj grešaka	Kumulativni broj grešaka	Kumulativni postotak grešaka
Kontrola toka	10	10	26.3%
Metoda	9	19	50%
Varijabla	9	28	73.7%
Izraz	6	34	89.5%
Deklaracija	2	36	94.7%
Komentari	2	38	100%
I/O	0	38	100%

Prikaz Pareto dijagrama za pronađene greške:



Šta se može zaključiti iz izgleda Pareto dijagrama? Da li je rast kumulativnog postotka linearan i šta to znači?

Verifikacija i Validacija Softvera

Iz izgleda ovog Pareto dijagrama možemo zaključiti da su kontrola toka, metoda i varijabla najdominantniji uzroci defekata u ovom programskom rješenju. Ako bi se fokusirali na ove tri oblasti prilikom otklanjanja grešaka, sigurno bi puno više stvari poboljšali, nego fokusiranjem na kategorije izraz, deklaracija, komentari i I/O, jer prve tri kategorije sadrže znatno više grešaka. U svakom slučaju, lakše je grupisati greške u istu kategoriju, i onda ih rješavati. Rast kumulativnog postotka je linearan, a to znači da zapažavamo sve više grešaka krećući se kroz kategorije check-listi, i tako broj grešaka raste.

Koristeći sumarni izvještaj o greškama iz zadatka 4 potrebno je napraviti histogram ozbiljnosti grešaka.

Prikaz histograma ozbiljnosti grešaka:



Šta se može zaključiti iz izgleda histograma? Za koju vrstu analize je pogodniji ovakav prikaz, a za koju vrstu analize je pogodnije korištenje Pareto dijagrama?

Iz izgleda ovog histograma možemo zaključiti da je najveći broj defekata onih sa ozbiljnošću tipa 2, i da te vrste defekata jako dominiraju. To znači da su rijetke greške koje utiču na osnovne funkcionalnosti programa, već da su najčešće one greške koje samo utiču na neugodnost programera, i lahko su otklonjive. Vidimo da ozbiljnijih grešaka (tipa 4 i 5) uopće nema. Histogram je pogodniji za analizu ozbiljnosti grešaka u programskom rješenju, dok je Pareto dijagram pogodniji za analizu kategorija kojima pripadaju greške.

Svaki član tima treba otkloniti najmanje 3 pronađene greške u rješenju koje su im drugi članovi tima dodijelili. Nakon otklanjanja svake pojedinačne greške potrebno je zabilježiti vrijednost metrike ciklomske kompleksnosti koja se dobiva u okviru prikaza metrika koda u Visual Studio okruženju.

Potrebno je prikazati izvršene promjene na način da se u okruženju prikazuje kod prije i nakon promjene. Svaki član tima u nastavku treba prikazati jedan primjer otklonjene greške.

Prikaz primjera otklonjene greške (član 1):

```
Autentifikacija.PokreniNovuSesiju(idKorisnika, HttpContext, tipKorisnika);

/*pocetak sumnjivog dijela koda*/
if (tipKorisnika == TipKorisnika.StudentskaSluzba)
{
    return RedirectToAction("Dashboard", new RouteValueDictionary(
        new { controller = "Studentska", action = "Dashboard" }));
}
else if (tipKorisnika == TipKorisnika.Student)
{
    return RedirectToAction("Dashboard", new RouteValueDictionary(
        new { controller = "Studentska", action = "Dashboard" }));
}
else
{
    return RedirectToAction("Dashboard", new RouteValueDictionary(
        new { controller = "Studentska", action = "Dashboard" }));
}

/*kraj sumnjivog dijela koda*/

conn.Close();
result.Read();
int idKorisnika = result.GetInt32(0);
TipKorisnika tipKorisnika = (TipKorisnika)result.GetInt32(1);

Autentifikacija.PokreniNovuSesiju(idKorisnika, HttpContext, tipKorisnika);

return RedirectToAction("Dashboard", new RouteValueDictionary(
    new { controller = "Studentska", action = "Dashboard" }));
}
}
catch (Exception)
```

Prikaz primjera otklonjene greške (član 2):

```
/*pocetak sumnjivog dijela koda*/
[Route("/student/{lokacija}/greska/{idPoruke}")]
public IActionResult prikaziGresku(string lokacija, int idPoruke)
{
    if (idPoruke == -11)
    {
        ViewBag.poruka = "A";
    }
    else if (idPoruke == 3)
    {
        ViewBag.poruka = "A";
    }
    return View();
}

/*kraj sumnjivog dijela*/

[Route("/student/{lokacija}/greska/{idPoruke}")]
0 references
public IActionResult prikaziGresku(string lokacija, int idPoruke)
{
    if (idPoruke == -11 || idPoruke == 3)
    {
        ViewBag.poruka = "A";
    }
    return View();
}
```

Prikaz primjera otklonjene greške (član 3):

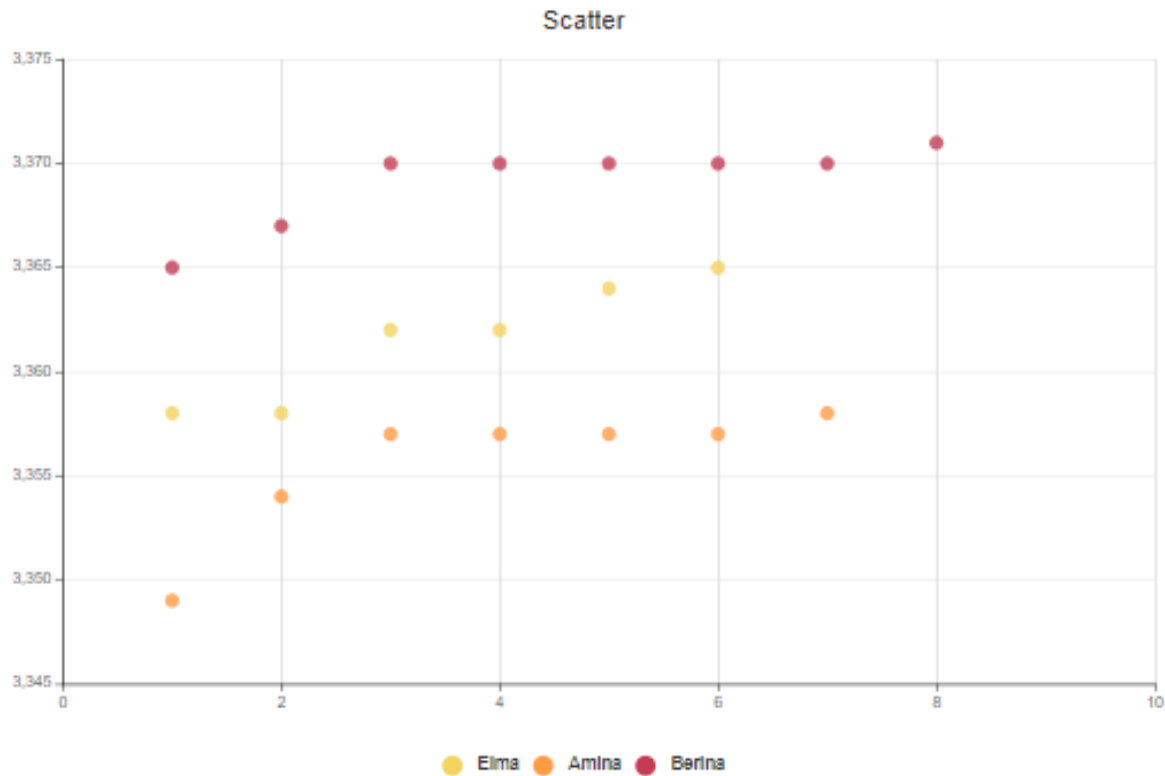
```
28      public bool istina = false;
43      public List<Poruka> dajInbox(int idOsobe)
44      {
45          if (Autentifikacija.GetIdKorisnika(HttpContext).Value == idOsobe)
46          {
47              /* ovaj dio koda je sumnjiv */
48              if (istina != true)
49                  return zmgr.dajInbox(idOsobe);
50              /* kraj sumnjivog dijela */
51          }
52          return null;
53      }
```

```
public List<Poruka> dajInbox(int idOsobe)
{
    if (Autentifikacija.GetIdKorisnika(HttpContext).Value == idOsobe)
    {
        return zmgr.dajInbox(idOsobe);
    }
    return null;
}
```

Verifikacija i Validacija Softvera

Nakon otklanjanja grešaka, potrebno je koristeći zabilježene vrijednosti metrike ciklomske kompleksnosti iz Visual Studio okruženja formirati scatter dijagram defekata.

Prikaz *scatter* dijagrama defekata:



Šta se može zaključiti iz izgleda *scatter* dijagrama? Da li se sa smanjenjem broja defekata smanjuje i kompleksnost koda? Da li to znači da je kompleksniji kod manje podložan greškama?

Iz izgleda *scatter* dijagrama zaključujemo da se otklanjanjem grešaka, smanjila i kompleksnost koda. Nije se značajno puno smanjila kompleksnost. Otklanjanjem nekih grešaka se nije smanjivala kompleksnost, ali generalno kompleksnost se smanjila. To znači da otklanjanjem više grešaka, to se više smanjuje kompleksnost. Samim tim zaključujemo da što manje grešaka imamo, to je i manja kompleksnost. S obzirom na to, to znači da je kompleksniji kod zapravo više podložan greškama.