



Zadaća br. 3

Izveštaj o *white box* testiranju

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: Studentski dom

Ime i prezime: Neira Novalić
Broj indexa: 18112

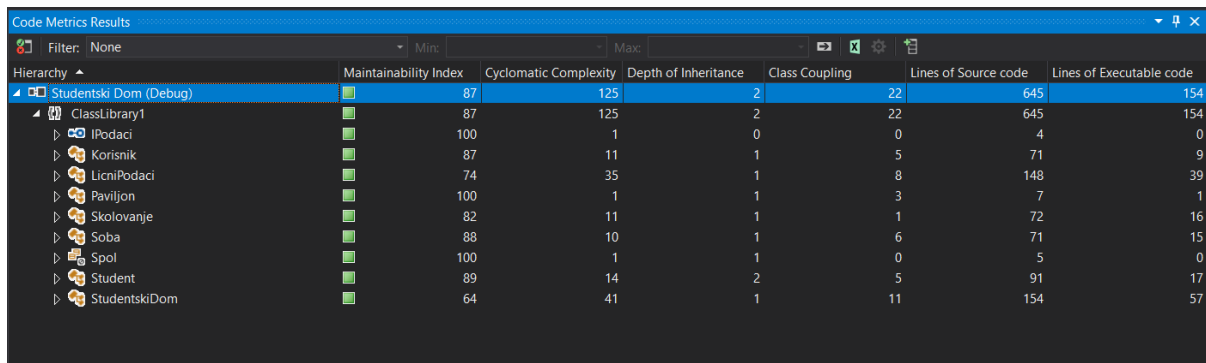
Ime i prezime: Mirnesa Salihović
Broj indexa: 18115.

Ime i prezime: Lejla Pirija
Broj indexa: 18238

Zadatak 1. (Metrike i refaktoring)

Potrebno je, koristeći alat *Visual Studio Code Metrics*, izračunati vrijednost najvažnijih metrika i pronaći metodu koja ima najveću vrijednost McCabe metrike. Ukoliko ima više takvih metoda, potrebno je odabrati bilo koju od njih.

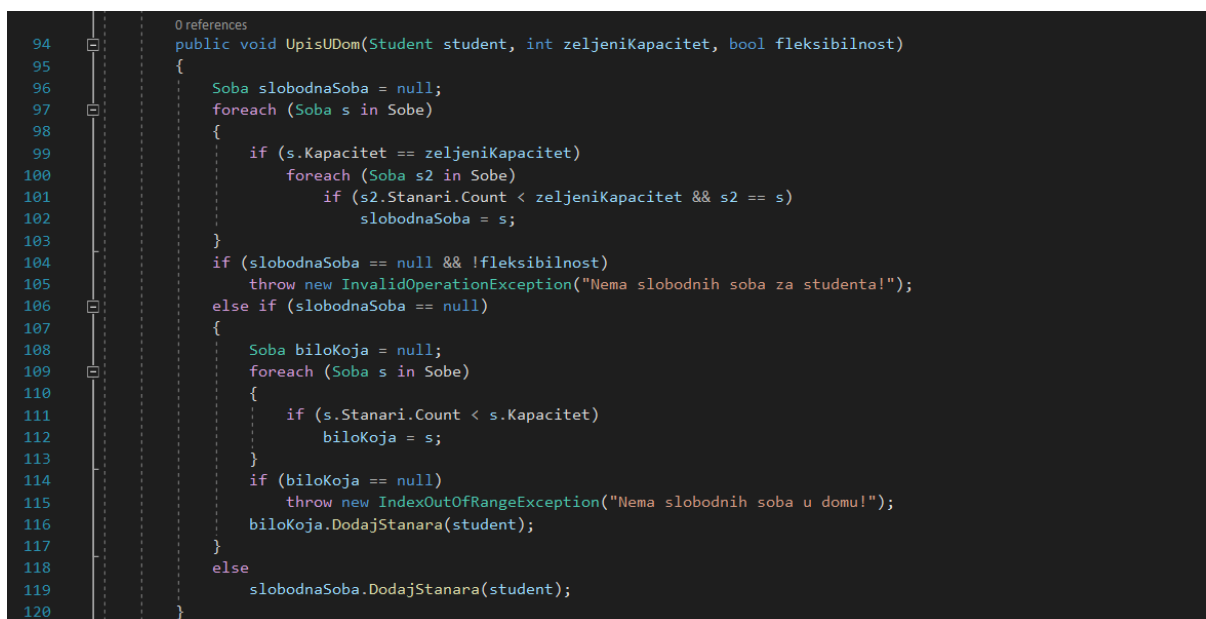
Prikaz rezultata analize metrika koda putem alata *Visual Studio Code Metrics*:



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
Studentski Dom (Debug)	87	125	2	22	645	154
ClassLibrary1	87	125	2	22	645	154
IPodaci	100	1	0	0	4	0
Korisnik	87	11	1	5	71	9
LicniPodaci	74	35	1	8	148	39
Paviljon	100	1	1	3	7	1
Skolovanje	82	11	1	1	72	16
Soba	88	10	1	6	71	15
Spol	100	1	1	0	5	0
Student	89	14	2	5	91	17
StudentskiDom	64	41	1	11	154	57

Metoda koja ima najveću ciklomatsku kompleksnost: `UpisUDom`

Prikaz programskog koda metode sa najvećom ciklomatskom kompleksnošću:



```
0 references
94 public void UpisUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
95 {
96     Soba slobodnaSoba = null;
97     foreach (Soba s in Sobe)
98     {
99         if (s.Kapacitet == zeljeniKapacitet)
100             foreach (Soba s2 in Sobe)
101                 if (s2.Stanari.Count < zeljeniKapacitet && s2 == s)
102                     slobodnaSoba = s;
103     }
104     if (slobodnaSoba == null && !fleksibilnost)
105         throw new InvalidOperationException("Nema slobodnih soba za studenta!");
106     else if (slobodnaSoba == null)
107     {
108         Soba biloKoja = null;
109         foreach (Soba s in Sobe)
110         {
111             if (s.Stanari.Count < s.Kapacitet)
112                 biloKoja = s;
113         }
114         if (biloKoja == null)
115             throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
116         biloKoja.DodajStanara(student);
117     }
118     else
119         slobodnaSoba.DodajStanara(student);
120 }
```

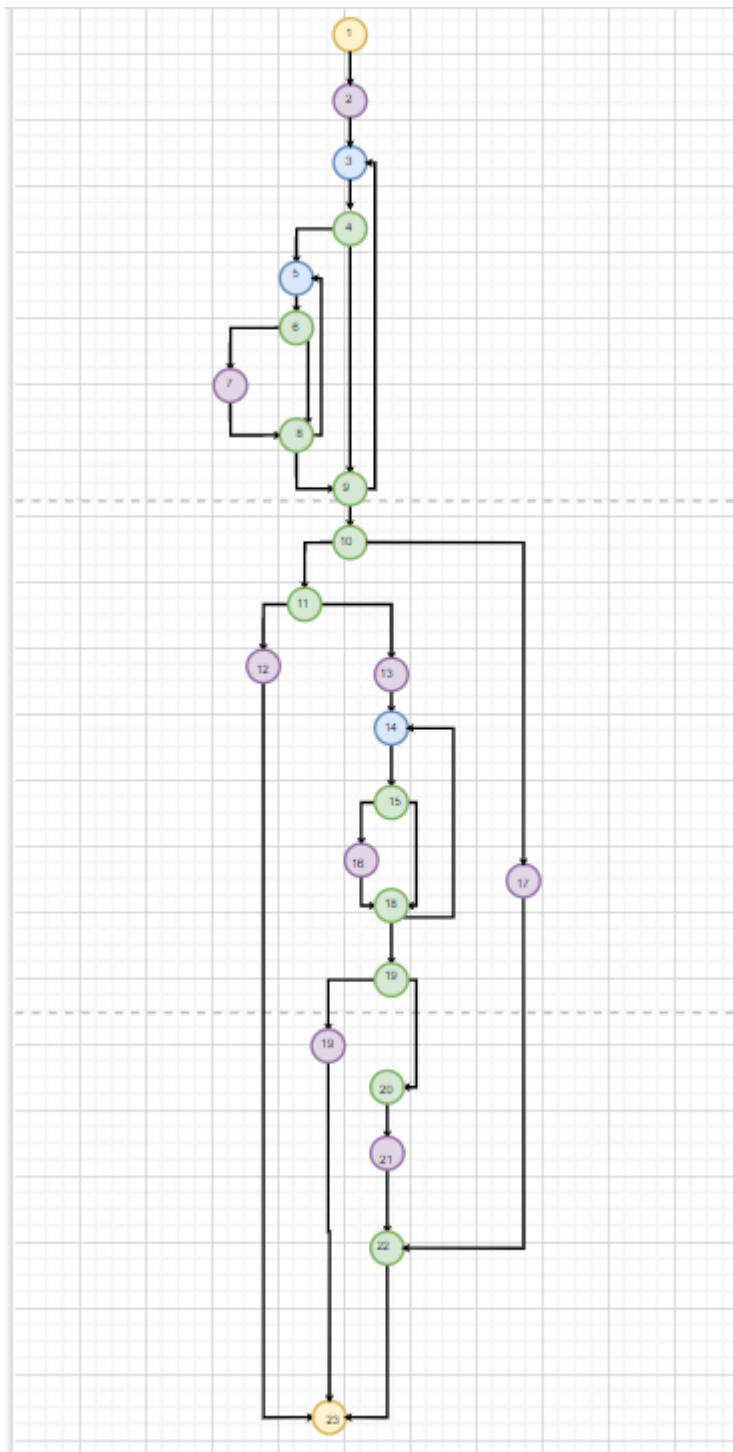
Šta je razlog za veliku ciklomatsku kompleksnost?

Razlog za veliku ciklomatsku kompleksnost je veliki broj if-else uslova, foreach petlji te operatora AND.

Kako bi se izvršila provjera dobivene vrijednosti McCabe metrike, potrebno je konstruisati kontrolni graf za metodu i manuelno izračunati vrijednost ciklomatske kompleksnosti.

Verifikacija i Validacija Softvera

Izgled kontrolnog grafa metode:



Broj grana grafa: 33

Broj čvorova grafa: 23

Broj neovisnih dijelova grafa: 1

Vrijednost McCabe metrike: 12

Prikaz programskog koda metode koja je odabrana za izračun Halstead metrika:

Verifikacija i Validacija Softvera

```

142 0 references
143 public void AutomatskoPostavljanjeNedostajucihPodataka()
144 {
145     if (email != null)
146         throw new ArgumentException("Email je već postavljen!");
147     if (ime != null)
148     {
149         Email = ime.ToLower().Substring(0, 1) + prezime.ToLower() + "1@etf.unsa.ba";
150     }
151     string dan = datumRodjenja.Day.ToString();
152     string mjesec = datumRodjenja.Month.ToString();
153     string godina = datumRodjenja.Year.ToString();
154     if (dan.Length == 1)
155         dan = "0" + dan;
156     if (mjesec.Length == 1)
157         mjesec = "0" + mjesec;
158     Slika = "server.etf.unsa.ba/slike/" + prezime + "_" + dan + mjesec + godina + ".jpg";
159 }
160
161 #endregion
162 }
163
164

```

Na osnovu čega je izvršen odabir metode? Šta se može postići izračunavanjem Halstead metrika za odabranu metodu?

Za računanje Halstead metrike odabrana je metoda koja sadrži veliki broj operatora i operanada, s obzirom da je cijela diskusija oko istog zasnovana na ovim parametrima. Metode u kojima ima veliki broj operatora i operanada su veoma "sumnjive" jer postoji velika mogućnost da one ne rade samo jednu stvar, da su previše kompleksne te da je velika podložnost greškama u istima. Iz tog razloga za njih računamo Halsteadovu metriku. Pored navedenih stvari, Halsteadova metrika će nam dati informaciju i o količini napora i vremena koja je potrebna za realizaciju date metode, a također važna nam je i informacija da broj isporučenih bugova aproksimira broj grešaka u modulu.

U tabele ispod potrebno je unijeti vrijednosti operatora i operanada koje metoda sadrži. Potrebno je izračunati broj pojedinačnih i ukupan broj operatora i operanada n i N . Ove vrijednosti neophodne su za izračunavanje Halstead metrika. Zatim je potrebno izračunati Halstead metrike koje se nalaze u trećoj tabeli.

Operandi:

Ime operatora	Broj ponavljanja
if	4
!=	2
=	7
{ }	2
+	10
==	2
.	10
Broj različitih operatora (n_1)	Ukupan broj operatora (N_1)
$n_1 = 7$	$N_1 = 37$

Operatori:

Verifikacija i Validacija Softvera

Ime operanda	Broj ponavljanja
email	1
ime	2
prezime	2
dan	5
mjesec	5
Email	1
Slika	1
godina	2
datumRodjenja.Day	1
datumRodjenja.Month	1
datumRodjenja.Year	1
null	2
"0"	2
"server.etf.unsa.ba/slike/"	1
" "	1
"jpg"	1
"1.etf.unsa.ba"	1
1	2
Broj različitih operanada (n_2)	Ukupan broj operanada (N_2)
$n_2 = 18$	$N_2 = 32$

Halstead metrike	
<i>Dužina programa</i>	$N = 69$
<i>Veličina vokabulara</i>	$n = 25$
<i>Volumen programa</i>	$V = 320,43$
<i>Nivo poteškoće</i>	$D = 5,28$
<i>Nivo programa</i>	$L = 0,19$
<i>Napor implementacije</i>	$E = 1691,87$
<i>Vrijeme implementacije</i>	$T = 93,99$
<i>Broj isporučenih bugova</i>	$B = 0,0473$

Izvršiti analizu dobivenih vrijednosti. Da li su vrijednosti nekih od metrika izvan referentnog opsega? Šta se može zaključiti na osnovu dobivenih vrijednosti?

Volumen date funkcije je u opsegu od 20 do 1000 te zaključujemo da ova funkcija ne radi previse stvari. S obzirom da se isti operandi ne pojavljuju veliki broj puta, i sam nivo poteškoće nije velik. Za samo testiranje najvažnija Halstead mjera je broj isporučenih bagova a ona je jako mala u ovoj konkretnoj analizi.

Prikaz programskog koda metode koja je odabrana za izračun metrike informacijskog toka:

Verifikacija i Validacija Softvera

```

98      public void UpisUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
99      {
100          Soba slobodnaSoba = null;
101          foreach (Soba s in Sobe)
102          {
103              if (s.Kapacitet == zeljeniKapacitet)
104              {
105                  foreach (Soba s2 in Sobe)
106                  {
107                      if (s2.Stanari.Count < zeljeniKapacitet && s2 == s)
108                          slobodnaSoba = s;
109                  }
110              }
111              if (slobodnaSoba == null && !fleksibilnost)
112                  throw new InvalidOperationException("Nema slobodnih soba za studenta!");
113              else if (slobodnaSoba == null)
114              {
115                  Soba biloKoja = null;
116                  foreach (Soba s in Sobe)
117                  {
118                      if (s.Stanari.Count < s.Kapacitet)
119                          biloKoja = s;
120                  }
121                  if (biloKoja == null)
122                      throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
123                  biloKoja.DodajStanara(student);
124              }
125              else
126                  slobodnaSoba.DodajStanara(student);
127          }
128      }

```

Na osnovu čega je izvršen odabir metode? Šta se može postići izračunavanjem metrike informacijskog toka za odabranu metodu?

Odabir metode je izvršen na osnovu broja varijabli, broja izuzetaka, broja linija koda, operacija, itd. Izračunavanjem metrike informacijskog toka možemo utvrditi kolika je povezanost posmatrane metode sa ostalim dijelovima koda. Visoka vrijednost ove metrike znači da je povezanost sa ostalim metodama velika, te da postoji velika mogućnost greške i upitna je pouzdanost samog sistema.

U tabelu ispod potrebno je unijeti vrijednosti koje su neophodne za izračunavanje metrike informacijskog toka i izračunati vrijednost te metrike.

Dužina programa	$length = 19$
Lokalni tok u modul	$t_1 = 8$
Strukture podataka koje se koriste	$s_1 = 2$
Lokalni tok iz modula	$t_2 = 4$
Strukture podataka koje se mijenjaju	$s_2 = 1$
Fanin	$F_1 = 10$
Fanout	$F_2 = 5$
Metrika informacijskog toka (IFC)	$IFC = 2500$
Težinska metrika informacijskog toka (WIFC)	$WIFC = 47500$

Izvršiti analizu dobivenih vrijednosti. Da li su vrijednosti nekih od metrika izvan referentnog opsega? Šta se može zaključiti na osnovu dobivenih vrijednosti?

Verifikacija i Validacija Softvera

Na osnovu analize dobivenih vrijednosti zaključujemo da je ova metoda podložna greškama i evidentan je manjak kohezije u istoj. Rezultati koje smo dobili predstavljaju samo relativne indikatore toga da data metoda nije dovoljno pouzdana i sigurna te da je “dobar kandidat” za redizajn.

*Potrebno je odabrati metode koje su najpovoljnije za vršenje refaktoringa i izvršiti refaktoring, a zatim objasniti kakve su promjene nastale nakon refaktoringa. **Potrebno je da svaki član tima izvrši barem po jedan refaktoring.***

Refaktoring – član 1 (Ime i prezime: Neira Novalić.)

Prikaz programskog koda odabrane metode prije vršenja refaktoringa:

```
0 references
public void RadSaStudentom(Student student, int opcija)
{
    if (opcija == 0)
    {
        if (Studenti.Find(s => s.IdentifikacioniBroj == student.IdentifikacioniBroj) != null)
            throw new DuplicateWaitObjectException("Nemoguće dodati postojećeg studenta!");
        Studenti.Add(student);
    }
    else if (opcija == 1)
    {
        Soba soba = Sobe.Find(s => s.DaLiJeStanar(student));
        if (soba == null)
            throw new InvalidOperationException("Student nije stanar nijedne sobe!");
        soba.IzbaciStudenta(student);
    }
    else if (opcija == 2)
    {
        Student studentIzListe = Studenti.Find(s => s.IdentifikacioniBroj == student.IdentifikacioniBroj);
        if (studentIzListe == null)
            throw new MissingMemberException("Student nije prijavljen u dom!");
        Studenti.Remove(studentIzListe);
    }
}
```

Koje refaktoringe je moguće primijeniti i na koji način (opisati šta je neophodno uraditi, kojoj grupi refaktoringa navedeni postupci pripadaju i šta se time postiže)? Odabrati najpovoljniji refaktoring i obrazložiti zbog čega je najpovoljniji.

Moguće je primijeniti refactoring iskaza i refactoring na nivou rutine. U ovom slučaju ono što možemo uraditi je kompleksan blok if-else uslova zamijeniti switch-caseom. Iako nije najpogodnije rješenje, kod se znatno pojednostavljuje, čitljiviji je i nema bespotrebnog ponavljanja. Još jedan refaktoring koji je ovdje bilo moguće uraditi jeste koristiti klase za date opcije te primjenom paterna razložiti sve slučajeve. Na taj način bi uklonili sve if-else iskaze te bi se ciklomatska kompleksnost znatno smanjila.

Prikaz programskog koda metode nakon vršenja najpogodnijeg refaktoringa:

Verifikacija i Validacija Softvera

```
71  
72  
0 references  
73 public void RadSaStudentom(Student student, int opcija)  
74 {  
75  
76     Student s = Studenti.Find(s => s.IdentifikacioniBroj == student.IdentifikacioniBroj);  
77     Student studentPrijavljenUDom = Studenti.Find(s => s.IdentifikacioniBroj == student.IdentifikacioniBroj);  
78     switch (opcija)  
79     {  
80  
81         case 0:  
82             if(s!=null) throw new DuplicateWaitObjectException("Nemoguće dodati postojećeg studenta!");  
83             Studenti.Add(student);  
84             break;  
85         case 1:  
86             if (studentPrijavljenUDom == null) throw new InvalidOperationException("Student nije stanar nijedne sobe!");  
87             (Sobe.Find(s => s.DaLiJeStanar(student))).IzbaciStudenta(student);  
88             break;  
89         case 2:  
90             if(s==null) throw new MissingMemberException("Student nije prijavljen u dom!");  
91             Studenti.Remove(studentPrijavljenUDom);  
92             break;  
93     }  
94 }
```

Da li je došlo do poboljšanja ciklomatske kompleksnosti? Kakav to značaj ima za korištenje aplikacije, a kakav za *white box* testiranje?

Ciklomatska kompleksnost date metode prije vršenja refaktoringa je 7.

Vrijednost indeksa održavanja metode prije vršenja refaktoringa: 56

Vrijednost indeksa održavanja metode nakon vršenja refaktoringa: 64

Da li je indeks održavanja promijenio kategoriju (crvena, žuta, zelena) nakon vršenja refaktoringa? Da li su ciklomatska kompleksnost i indeks održavanja u korelaciji i zašto?

Nakon izvršenog refactoring indeks nije promijenio boju kategorije, ostala je zelena boja. Ciklomatska kompleksnost i indeks održavanja su u korelaciji. Smanjenjem ciklomatske kompleksnosti povećava se indeks održavanja metode u skoro svim slučajevima. Prilikom ovog refaktoringa nije došlo do promjene ciklomatske kompleksnosti, ali se indeks održavanja povećao, što znači da je metoda postala lakša za održavanje.

Refactoring – član 2 (Ime i prezime: Mirnesa Salihović)

Prikaz programskog koda odabrane metode prije vršenja refaktoringa:

```
0 references  
public List<Student> DajStudenteIzPaviljona(IPodaci paviljon)  
{  
    List<Student> studenti = new List<Student>();  
    foreach (Student s in Studenti)  
    {  
        if (s.Skolovanje.MaticniFakultet == paviljon.DajImePaviljona())  
            studenti.Add(s);  
    }  
  
    return studenti;  
}
```


Koje refaktoringe je moguće primijeniti i na koji način (opisati šta je neophodno uraditi, kojoj grupi refaktoringa navedeni postupci pripadaju i šta se time postiže)? Odabrati najpovoljniji refaktoring i obrazložiti zbog čega je najpovoljniji.

Ono što možemo uraditi jeste napraviti potpunu promjenu algoritma koji se koristi za pronalazak studenata. Umjesto manuelnog prolaska kroz listu studenata i dodavanje studenata u listu koja se zatim vraća kao rezultat metode, najbolje je takav algoritam zamijeniti postojećim algoritmom za pronalazak instanci koji implementira metoda klase List FindAll. Na ovaj način uklanja se foreach petlja i grananja. Ovo predstavlja refaktoring na nivou rutine.

Prikaz programskog koda metode nakon vršenja najpogodnijeg refaktoringa:

```
0 references
public List<Student> RefactoringDajStudenatIzPaviljona(IPodaci paviljon)
{
    List<Student> studenti = new List<Student>();
    studenti= Studenti.FindAll(s => s.Skolovanje.MaticniFakultet == paviljon.DajImePaviljona());
    return studenti;
}
```

Da li je došlo do poboljšanja ciklomatske kompleksnosti? Kakav to značaj ima za korištenje aplikacije, a kakav za *white box* testiranje?

Da doslo je do poboljšanja ciklomatske kompleksnosti. Ukoliko aplikacija pokrene ovu metodu ona ce se brze izvršiti u odnosu na staru metodu ber refaktoringa. White box testiranje ce imati manji broj testnih slucajeva , jer postoji manji broj minimalno obuhvatnih puteva.

Vrijednost indeksa održavanja metode prije vršenja refaktoringa: 74

Vrijednost indeksa održavanja metode nakon vršenja refaktoringa: 74

Da li je indeks održavanja promijenio kategoriju (crvena, žuta, zelena) nakon vršenja refaktoringa? Da li su ciklomatska kompleksnost i indeks održavanja u korelaciji i zašto?

Nije se promijenio.

Smanjenje ciklomatske kompleksnosti znači da je jednostavnije testirati kod. Indeks je ostao isti što znači održavanje osalo isto kao i prije refaktoringa.

Refaktoring – član 3 (Ime i prezime: Lejla Pirija.)

Prikaz programskog koda odabrane metode prije vršenja refaktoringa:

Verifikacija i Validacija Softvera

```

97 0 references
98 public void UpisiUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
99 {
100     Soba slobodnaSoba = null;
101     foreach (Soba s in Sobe)
102     {
103         if (s.Kapacitet == zeljeniKapacitet)
104             foreach (Soba s2 in Sobe)
105                 if (s2.Stanari.Count < zeljeniKapacitet && s2 == s)
106                     slobodnaSoba = s;
107     }
108     if (slobodnaSoba == null && !fleksibilnost)
109         throw new InvalidOperationException("Nema slobodnih soba za studenta!");
110     else if (slobodnaSoba == null)
111     {
112         Soba biloKoja = null;
113         foreach (Soba s in Sobe)
114         {
115             if (s.Stanari.Count < s.Kapacitet)
116                 biloKoja = s;
117         }
118         if (biloKoja == null)
119             throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
120         biloKoja.DodajStanara(student);
121     }
122     else
123         slobodnaSoba.DodajStanara(student);

```

Koje refaktoringe je moguće primijeniti i na koji način (opisati šta je neophodno uraditi, kojoj grupi refaktoringa navedeni postupci pripadaju i šta se time postiže)? Odabrati najpovoljniji refaktoring i obrazložiti zbog čega je najpovoljniji.

Moguće je primijeniti refactoring „Zamjenski algoritam“, Dakle, možemo zamijeniti foreach petlje i grananja pozivom metode Find nad listom. Time ćemo dobiti čitljiviji i uredniji kod.

Prikaz programskog koda metode nakon vršenja najpogodnijeg refaktoringa:

```

0 references
public void UpisiUDomRefactoring(Student student, int zeljeniKapacitet, bool fleksibilnost)
{
    Soba slobodnaSoba = null;
    slobodnaSoba = Sobe.Find(s => s.Kapacitet == zeljeniKapacitet && s.Stanari.Count < zeljeniKapacitet);
    if (slobodnaSoba == null && !fleksibilnost)
        throw new InvalidOperationException("Nema slobodnih soba za studenta!");
    else if (slobodnaSoba == null)
    {
        slobodnaSoba = Sobe.Find(s => s.Stanari.Count < s.Kapacitet);
        if (slobodnaSoba == null)
            throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
    }
    slobodnaSoba.DodajStanara(student);
}

```

Da li je došlo do poboljšanja ciklomatske kompleksnosti? Kakav to značaj ima za korištenje aplikacije, a kakav za *white box* testiranje?

Da, došlo je do poboljšanja ciklomatske kompleksnosti. Ona sada iznosi $M = 6$.



Verifikacija i Validacija Softvera

Za aplikaciju je to značajno jer će trebati manje vremena da se izvrši ova funkcija, a za white box testiranje će trebati manje testova.

Vrijednost indeksa održavanja metode prije vršenja refaktoringa: 59

Vrijednost indeksa održavanja metode nakon vršenja refaktoringa: 61

Da li je indeks održavanja promijenio kategoriju (crvena, žuta, zelena) nakon vršenja refaktoringa? Da li su ciklomatska kompleksnost i indeks održavanja u korelaciji i zašto?

Indeks održavanja nije promijenio kategoriju (zelena) nakon vršenja refaktoringa. Ciklomatska kompleksnost se smanjila, a indeks održavanja povećao. Smanjenje ciklomatske kompleksnosti znači da je jednostavnije testirati kod, a i povećana vrijednost indeksa održavanja znači da je metoda postala lakša za održavanje.

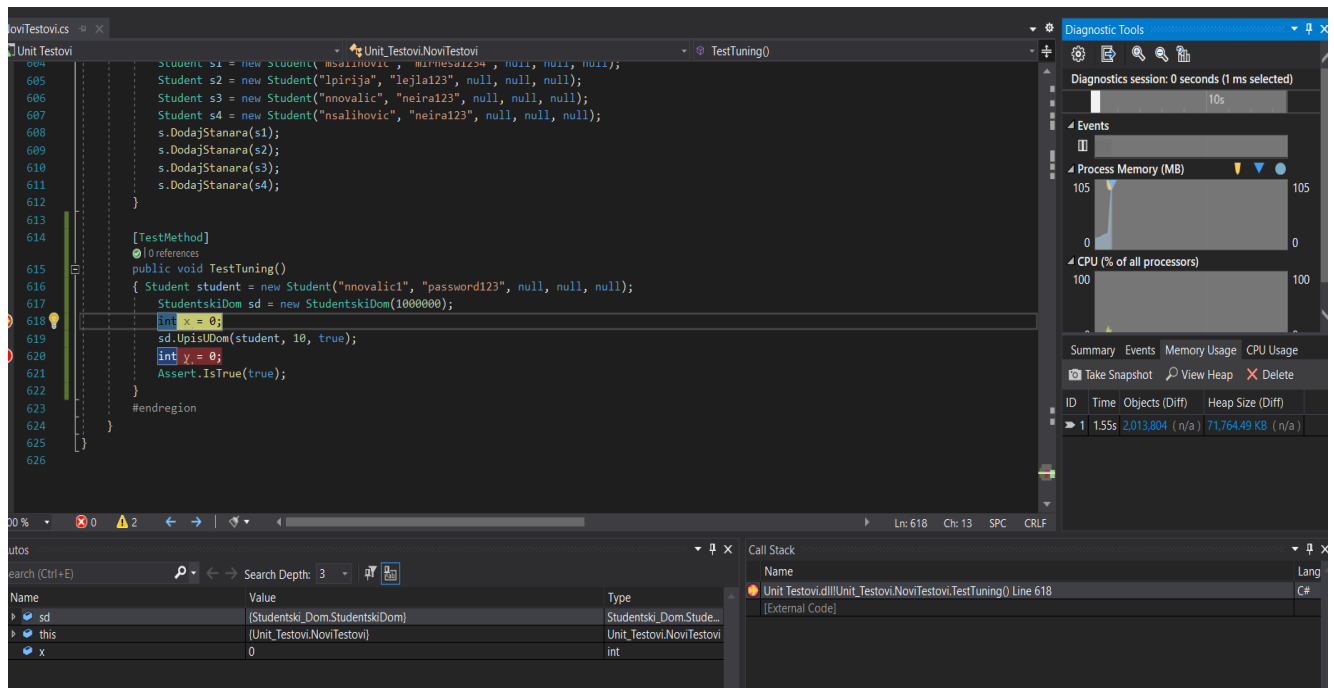
Zadatak 2. (Code tuning)

Za svaku nastavnu grupu definisana je različita metoda koja je pogodna za vršenje code tuninga. U nastavku je potrebno primijeniti tehnike code tuninga kako bi se smanjilo zauzeće memorijskih i procesorskih resursa pri pokretanju navedene metode.

Metoda koja je dodijeljena za code tuning: `StudentskiDom.UpisUDom`

Potrebno je definisati unit test projekat i dodati jedan unit test u kojem se poziva tražena metoda (sam rezultat testa nije važan – dovoljno je da se test može pokrenuti u svrhu pokretanja tražene metode). **Kolekcije elemenata koje se koriste u dodijeljenoj metodi trebaju imati 1,000,000 elemenata.** Pri pokretanju je potrebno izvršiti debugging kako bi se detektovale vrijednosti zauzeća memorije i procesorskih jedinica nakon završetka rada metode. Kako bi se dobili što relevantniji rezultati, **potrebno je izvršiti usrednjavanje dobivenih vrijednosti iz 10 izvršavanja.**

Prikaz Visual Studio Diagnostic Tool alata pri dostizanju breakpointa nakon završetka izvršavanja metode (dovoljan je prikaz jednog od 10 izvršavanja):



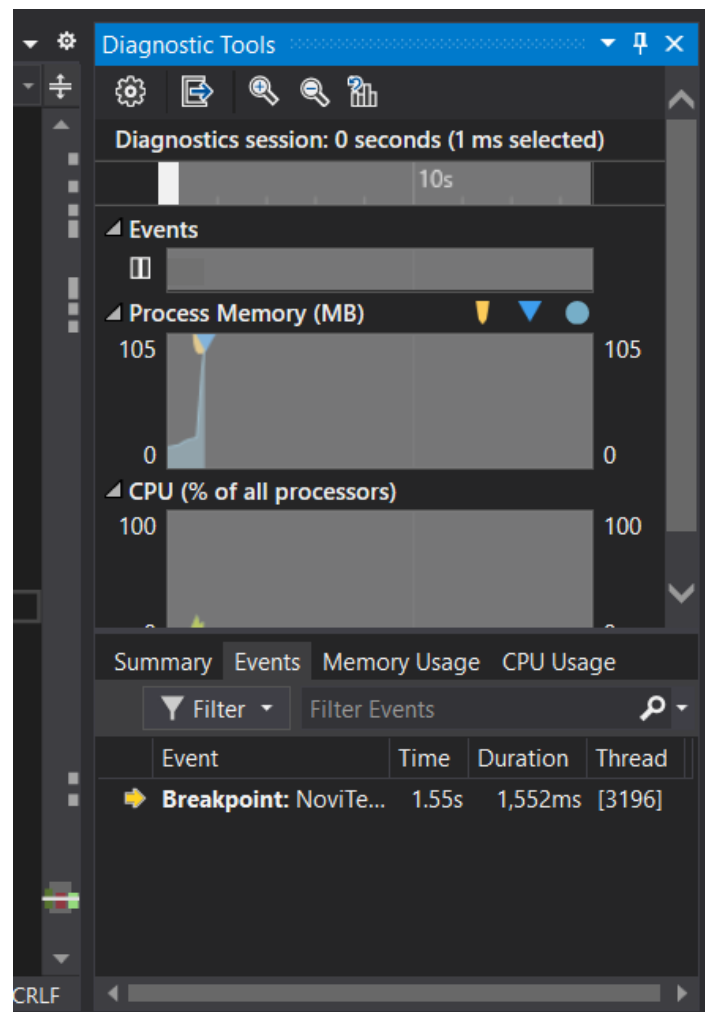
The screenshot shows the Visual Studio IDE with a unit test project named 'Unit Testovi'. The test method 'TestTuning()' is highlighted, and a breakpoint is set at line 618. The diagnostic tools window is open, showing a summary of the test run. The 'Summary' tab is selected, displaying the following data:

ID	Time	Objects (Diff)	Heap Size (Diff)
1	1.55s	2,013,804 (n/a)	71,764.49 KB (n/a)

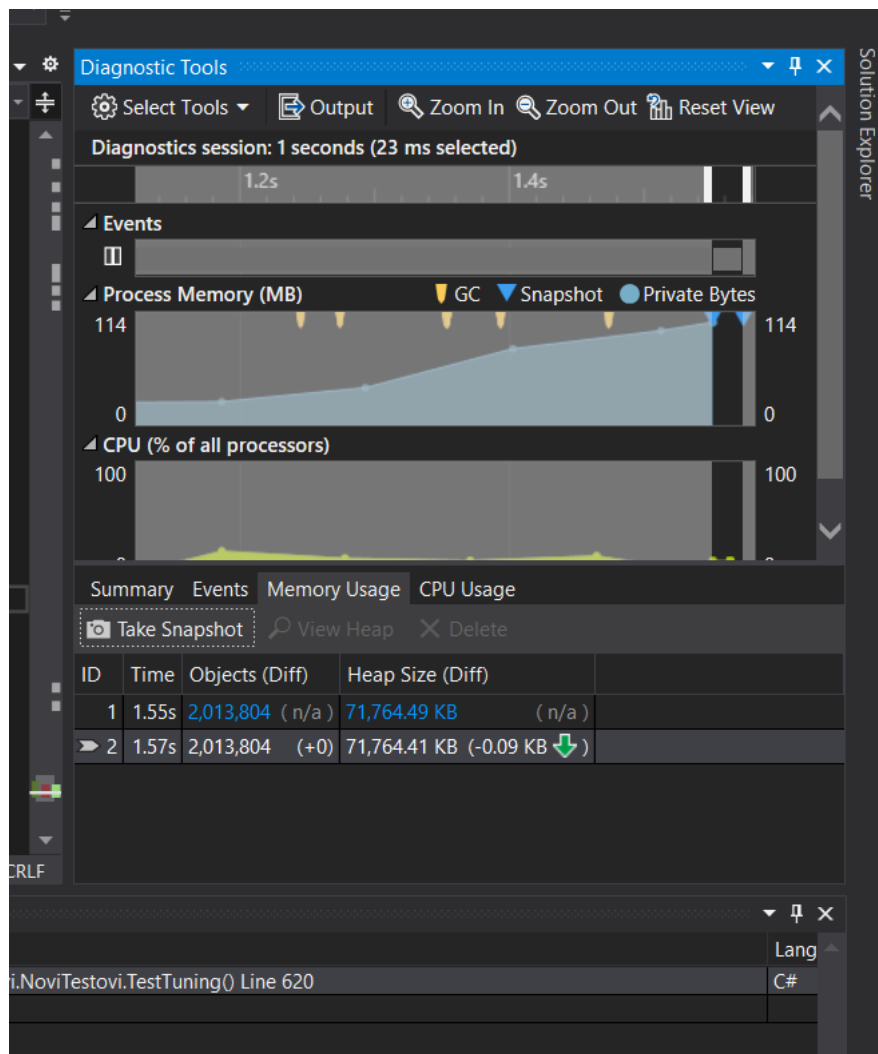
The 'Call Stack' window is also open, showing the call stack for the test method. The stack includes the following frames:

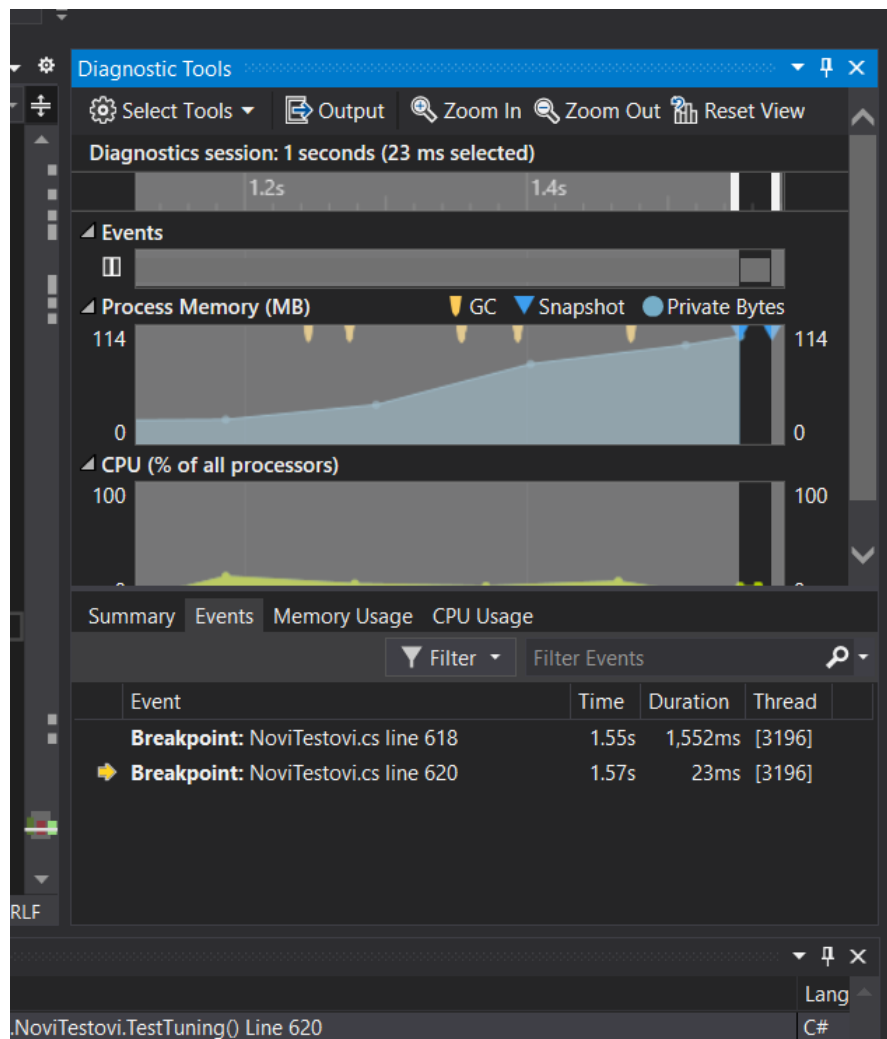
- Unit Testovi.dll\Unit_Testovi.NoviTestovi.TestTuning() Line 618
- [External Code]

Verifikacija i Validacija Softvera



Verifikacija i Validacija Softvera





Srednja vrijednost zauzeća memorije: 65.4

Srednja vrijednost korištenja procesorskih jedinica: 21.5 %

Srednje vrijeme izvršavanja metode: 1.35 s

Da li su navedene vrijednosti prihvatljive i zašto?

S obzirom na veliki broj podataka koji se obrađuju u datoj metodi, rezultati su poprilično prihvatljivi iako je moguće veliko poboljšanje. Iz tog razloga u nastavku radimo code tuning i vršimo analizu.

Zauzeće procesorskih jedinica je veliko s obzirom na to da se obrađuje samo ova metoda, a i samo izvršavanje metode je dugo.

*Potrebno je izvršiti code tuning nad datom metodom kako bi se poboljšale dobivene vrijednosti zauzeća resursa. **Potrebno je da svaki član tima uradi po jedan code tuning nad metodom.** Code tuning se vrši iterativno, tako da se na kraju dobivaju četiri metode – originalna metoda, metoda nakon vršenja prvog code tuninga, metoda nakon vršenja prvog i drugog tuninga, kao i metoda nakon vršenja svih code tuninga.*

Prikaz programskog koda metode prije vršenja code tuninga:

Verifikacija i Validacija Softvera

```
98 public void UpisUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
99 {
100     Soba slobodnaSoba = null;
101     foreach (Soba s in Sobe)
102     {
103         if (s.Kapacitet == zeljeniKapacitet)
104             foreach (Soba s2 in Sobe)
105                 if (s2.Stanari.Count < zeljeniKapacitet && s2 == s)
106                     slobodnaSoba = s;
107     }
108     if (slobodnaSoba == null && !fleksibilnost)
109         throw new InvalidOperationException("Nema slobodnih soba za studenta!");
110     else if (slobodnaSoba == null)
111     {
112         Soba biloKoja = null;
113         foreach (Soba s in Sobe)
114         {
115             if (s.Stanari.Count < s.Kapacitet)
116                 biloKoja = s;
117         }
118         if (biloKoja == null)
119             throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
120         biloKoja.DodajStanara(student);
121     }
122     else
123         slobodnaSoba.DodajStanara(student);
```

Prikaz programskog koda metode nakon vršenja prvog *code tuninga*:

```
0 references
public void UpisUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
{
    foreach (Soba s in Sobe)
    {
        if (s.Kapacitet == zeljeniKapacitet && s.Stanari.Count < zeljeniKapacitet)
        {
            s.DodajStanara(student);
            return;
        }
    }
    if (!fleksibilnost)
        throw new InvalidOperationException("Nema slobodnih soba za studenta!");
    else
    {
        Soba biloKoja = null;
        foreach (Soba s in Sobe)
        {
            if
                (s.Stanari.Count < s.Kapacitet)
                biloKoja = s;
        }
        if (biloKoja == null)
            throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
        biloKoja.DodajStanara(student);
    }
}
```


Kategorija izvršenog *code tuninga* i kratak opis onog što je urađeno:

Ovaj tuning spada u tuning logičkih iskaza. U prvom koraku uočeno je bespotrebno korištenje dodatne promjenljive slobodnaSoba te je ona uklonjena iz datog koda i izvršene su izmjene na mjestima gdje se ona pojavljivala. Dodavanje studenta koje se vršilo na kraju programa je prebačeno u prvu foreach petlju i izvršava se ukoliko su zadovoljeni uslovi za pronalazak odgovarajuće sobe. Također uklonjena je i druga petlja koja je prolazila kroz iste podatke.

Prikaz programskog koda metode nakon vršenja prvog i drugog *code tuninga*:

```
0 references
public void UpisUDom(Student student, int zeljeniKapacitet, bool fleksibilnost)
{
    foreach (Soba s in Sobe)
    {
        if (s.Kapacitet == zeljeniKapacitet && s.Stanari.Count < zeljeniKapacitet)
        {
            s.DodajStanara(student);
            return;
        }
    }
    if (!fleksibilnost)
        throw new InvalidOperationException("Nema slobodnih soba za studenta!");
    else
    {
        foreach (Soba s in Sobe)
        {
            if (s.Stanari.Count < s.Kapacitet)
            {
                s.DodajStanara(student);
                return;
            }
        }

        throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
    }
}
```

Kategorija izvršenog *code tuninga* i kratak opis onog što je urađeno:

Kao i prethodni i ovaj code tuning spada u tuning logičkih iskaza. Analogno kao u prvoj iteraciji, u ovoj je uklonjena varijabla biloKoja te je cijeli kod prilagođen toj izmjeni.

Prikaz programskog koda metode nakon vršenja svih *code tuninga*:

Verifikacija i Validacija Softvera

```
0 references
public void UpisUDomPROBA(Student student, int zeljeniKapacitet, bool fleksibilnost)
{
    Soba nova = null;
    foreach (Soba s in Sobe)
    {
        if (s.Kapacitet == zeljeniKapacitet && s.Stanari.Count < zeljeniKapacitet)
        {
            s.DodajStanara(student);
            return;
        }
        else if (fleksibilnost && s.Stanari.Count < s.Kapacitet)
        {
            nova = s;
        }
    }
    if (!fleksibilnost)
        throw new InvalidOperationException("Nema slobodnih soba za studenta!");
    else if (nova == null)
    {
        throw new IndexOutOfRangeException("Nema slobodnih soba u domu!");
    }

    nova.DodajStanara(student);
    return;
}
```

Kategorija izvršenog *code tuninga* i kratak opis onog što je urađeno:

Ovaj code tuning spada u kategoriju tuninga petlji, preciznije Jamming. On predstavlja kombinovanje 2 petlje koje imaju iste kontrolne uslove i koje rade na istom setu elemenata. U ovom slučaju smo uslove druge petlje “prebacili” u prvu te na taj način smo uklonile bespotrebni prolazak kroz isti set elemenata dva puta (set soba). Logika izvršavanja cijelog programskog koda je ostala ista.

U tabelu ispod potrebno je unijeti vrijednosti vremena izvršavanja i zauzeća resursa od strane svih pojedinačnih metoda sa vršenjem *code tuninga*. Sve metode kao parametar trebaju primati listu sa 1,000,000 elemenata. Potrebno je vršiti usrednjavanje dobivenih vrijednosti iz 10 izvršavanja.

Implementacija metode	Vrijeme izvršavanja (ms)	Zauzeće memorije (MB)	Zauzeće procesora (%)
Bez code tuninga	85 ms	65.4 KB	21.5 %
Tuning 1	51.66 ms	149.6KB	27.6%
Tuning 2	26.16 ms	84,06KB	31.3%
Tuning 3	30.6 ms	125.768KB	26.6%

Koja implementacija pokazuje najbolje rezultate? Da li je to posljednja implementacija, nakon vršenja svih *code tuninga*? Ukoliko ne, zbog čega je to tako? Šta je razlog zbog čega ostale implementacije ne pokazuju jednako dobre rezultate?



Verifikacija i Validacija Softvera

Najbolje rezultate pokazuje 2.metoda,a ne posljednja implementacija. Naime code tuning ne mora značiti da će doći do poboljšanja svih parametara. Optimizacije koje koriste tuning petlji nisu rezultovale smanjenjem zauzeća memorije i ubrzanjem cijelog procesa.

Zadatak 3. (White box testiranje)

Za svaku nastavnu grupu definisana je različita metoda za koju će biti izvršeno white box testiranje.

Metoda koja je dodijeljena za white box testiranje: StudentskiDom.StudentskiDom

Prikaz programskog koda metode:

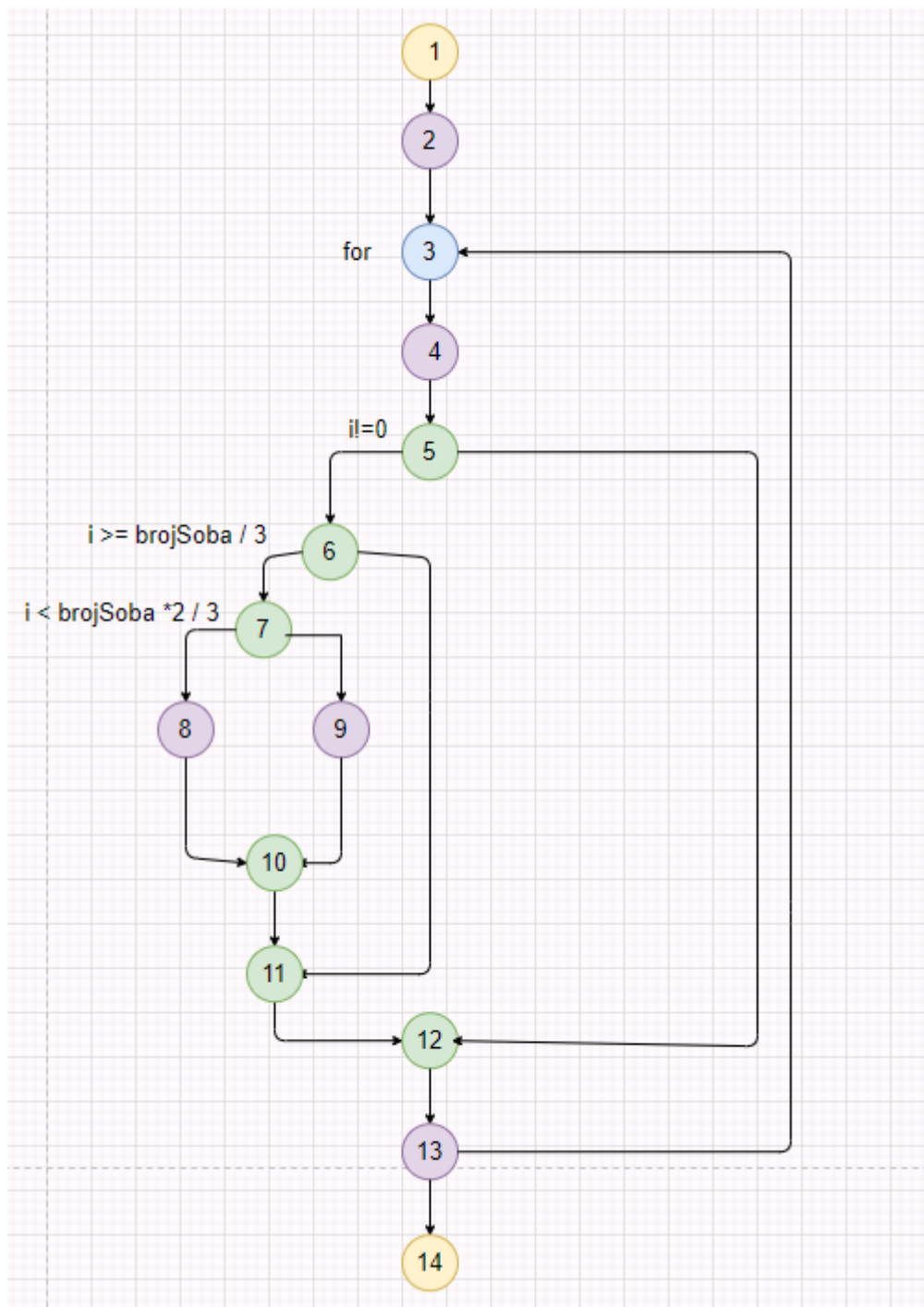
```
#region konstruktor

0 references
public StudentskiDom(int brojSoba)
{
    studenti = new List<Student>();
    sobe = new List<Soba>();
    for (int i = 0; i < brojSoba; i++)
    {
        int brojSobe = 100 + i;
        int kapacitet = 2;
        if (i != 0 && i >= brojSoba / 3 && i < brojSoba * 2 / 3)
        {
            brojSobe += 100;
            kapacitet += 1;
        }
        else if (i != 0 && i >= brojSoba * 2 / 3)
        {
            brojSobe += 200;
            kapacitet += 2;
        }
        Sobe.Add(new Soba(brojSobe, kapacitet));
    }
}

#endregion
```

Potrebno je napraviti graf programskog koda za dodijeljenu metodu. Svi čvorovi na grafu trebaju biti numerisani.

Prikaz grafa programskog toka metode:



U tabelu ispod potrebno je unijeti sve puteve kroz graf. Ukoliko se neki od puteva nikada ne može izvršiti zbog međusobne kontradiktornosti uslova ili drugih razloga, **takve puteve potrebno je označiti crvenom bojom**.

Redni broj puta	Čvorovi koje put obuhvata
01.	1-2-3-4-5-6-7-8-10-11-12-13-14
02.	1-2-3-4-5-6-7-9-10-11-12-13-14
03.	1-2-3-4-5-6-11-12-13-14
04.	1-2-3-4-5-12-13-14

Verifikacija i Validacija Softvera

05.	1-2-3-4-5-12-13-3-4-5-12-13-14
06.	

Šta je razlog zbog čega se putevi označeni crvenom bojom nikada ne mogu izvršiti (ukoliko takvi postoje):

Nema takvih puteva

*Potrebno je formirati sve testne slučajeve za potpuni obuhvat puteva, linija koda i neovisnih puteva. **Svaki član tima treba formirati testne slučajeve za jedan potpuni obuhvat. Testni slučajeve podrazumjevaju i specifikaciju vrijednosti parametara i ostalih varijabli koje će dovesti do izvršavanja određenih linija koda, kako je prethodno formiranim putevima definisano. Nakon toga potrebno je izvršiti diskusiju o razlikama između formiranih testnih slučajeva.***

Navesti sve testne slučajeve za postizanje potpunog obuhvata puteva:

1-2-3-4-5-6-7-8-10-11-12-13-14
1-2-3-4-5-6-7-9-10-11-12-13-14
1-2-3-4-5-6-11-12-13-14
1-2-3-4-5-12-13-14
1-2-3-4-5-12-13-3-4-5-12-13-14

TESTNI SLUČAJ: Parametar: brojSoba=3

Navesti sve testne slučajeve za postizanje potpunog obuhvata linija koda:

1-2-3-4-5-6-7-8-10-11-12-13-14
1-2-3-4-5-6-7-9-10-11-12-13-14
1-2-3-4-5-6-11-12-13-14
1-2-3-4-5-12-13-14

TESTNI SLUČAJ: Parametar: brojSoba=3

Navesti sve testne slučajeve za postizanje potpunog obuhvata neovisnih puteva:

1-2-3-4-5-6-7-8-10-11-12-13-14
1-2-3-4-5-6-7-9-10-11-12-13-14
1-2-3-4-5-12-13-14

Koja je razlika između tri prethodno formirana skupa testnih slučajeva? Koji od potpunih obuhvata zahtijeva najveći, a koji najmanji broj testova? Koje su prednosti, a koje mane odabira svakog od skupova testnih slučajeva?

Click or tap here to enter text.

Potrebno je definisati unit testove koristeći tri strategije testiranja: potpuni obuhvat odluka, uslova i petlji. **Svaki član tima treba izvršiti unit testiranje koristeći jednu strategiju.** Kako bi prikaz pokrivenosti koda metode bila relevantna, potrebno je pokrenuti samo testove iz određene strategije testiranja a zatim prikazati pokrivenost koda metode. Preporučuje se razdvajanje testnih strategija u više testnih klasa i pokretanje samo testova iz jedne testne klase za lakši prikaz pokrivenosti koda.

Prikaz unit testova za potpuni obuhvat odluka:

```
[TestMethod]
0 references
public void TestTuning()
{
    StudentskiDom a = new StudentskiDom(3);
    Assert.IsTrue(true);
}
```

Prikaz pokrivenosti koda metode nakon testiranja:

```
1 reference | 1/1 passing
32 public StudentskiDom(int brojSoba)
33 {
34     studenti = new List<Student>();
35     sobe = new List<Soba>();
36     for (int i = 0; i < brojSoba; i++)
37     {
38         int brojSobe = 100 + i;
39         int kapacitet = 2;
40         if (i != 0 && i >= brojSoba / 3 && i < brojSoba * 2 / 3)
41         {
42             brojSobe += 100;
43             kapacitet += 1;
44         }
45         else if (i != 0 && i >= brojSoba * 2 / 3)
46         {
47             brojSobe += 200;
48             kapacitet += 2;
49         }
50         Sobe.Add(new Soba(brojSobe, kapacitet));
51     }
52 }
53
```

Testirana je samo metoda(konstruktor) StudentskiDom klase StudentskiDom

StudentskiDom	22	83	105	179	20.9%	<div style="width: 20.9%;"></div>
---------------	----	----	-----	-----	-------	-----------------------------------

Prikaz unit testova za potpuni obuhvat uslova:

```
[TestMethod]
0 references
public void TestTuning()
{
    StudentskiDom a = new StudentskiDom(3);
    Assert.IsTrue(true);
}
```

Prikaz pokrivenosti koda metode nakon testiranja:

```
1 reference | 1/1 passing
32 public StudentskiDom(int brojSoba)
33 {
34     studenti = new List<Student>();
35     sobe = new List<Soba>();
36     for (int i = 0; i < brojSoba; i++)
37     {
38         int brojSobe = 100 + i;
39         int kapacitet = 2;
40         if (i != 0 && i >= brojSoba / 3 && i < brojSoba * 2 / 3)
41         {
42             brojSobe += 100;
43             kapacitet += 1;
44         }
45         else if (i != 0 && i >= brojSoba * 2 / 3)
46         {
47             brojSobe += 200;
48             kapacitet += 2;
49         }
50         Sobe.Add(new Soba(brojSobe, kapacitet));
51     }
52 }
53
```

Testirana je samo metoda(konstruktor) StudentskiDom klase StudentskiDom

StudentskiDom	22	83	105	179	20.9%	<div style="width: 20.9%; height: 10px; background: linear-gradient(to right, green, red);"></div>
---------------	----	----	-----	-----	-------	--

S obzirom da data metoda sadrži petlju u kojoj se nalaze if uslovi, koje mi zapravo i trebamo testirati, prosljeđivanjem odgovarajućeg parametra u funkciju, pri izvršavanju petlje provjeravat će se i svi uslovi. Prosljeđivanjem parametra 3 u metodu, ostvaruje se potpuni obuhvat uslova.

Prikaz unit testova za potpuni obuhvat petlji:

Verifikacija i Validacija Softvera

```
[TestMethod]
✓ | 0 references
public void TestTuning()
{
    StudentskiDom studentskiDom = new StudentskiDom(1);
    Assert.AreEqual(studentskiDom.Sobe.Count, 1);
}

[TestMethod]
✓ | 0 references
public void TestTuning1()
{
    StudentskiDom studentskiDom = new StudentskiDom(2);
    Assert.AreEqual(studentskiDom.Sobe.Count, 2);
}

[TestMethod]
✓ | 0 references
public void TestTuning2()
{
    StudentskiDom studentskiDom = new StudentskiDom(5);
    Assert.AreEqual(studentskiDom.Sobe.Count, 5);
}

[TestMethod]
✓ | 0 references
public void TestTuning3()
{
    StudentskiDom studentskiDom = new StudentskiDom(6);
    Assert.AreEqual(studentskiDom.Sobe.Count, 6);
}

[TestMethod]
✓ | 0 references
public void TestTuning4()
{
    StudentskiDom studentskiDom = new StudentskiDom(7);
    Assert.AreEqual(studentskiDom.Sobe.Count, 7);
}

[TestMethod]
✓ | 0 references
public void TestTuning5()
{
    StudentskiDom studentskiDom = new StudentskiDom(8);
    Assert.AreEqual(studentskiDom.Sobe.Count, 8);
}
```

Prikaz pokrivenosti koda metode nakon testiranja:

Verifikacija i Validacija Softvera

```

6 references | 6/6 passing
public StudentskiDom(int brojSoba)
{
    studenti = new List<Student>();
    sobe = new List<Soba>();
    for (int i = 0; i < brojSoba; i++)
    {
        int brojSobe = 100 + i;
        int kapacitet = 2;
        if (i != 0 && i >= brojSoba / 3 && i < brojSoba * 2 / 3)
        {
            brojSobe += 100;
            kapacitet += 1;
        }
        else if (i != 0 && i >= brojSoba * 2 / 3)
        {
            brojSobe += 200;
            kapacitet += 2;
        }
        Sobe.Add(new Soba(brojSobe, kapacitet));
    }
}

```

Testirana je samo metoda(konstruktor) StudentskiDom klase StudentskiDom

StudentskiDom	22	83	105	179	20.9%	<div style="width: 20.9%;"></div>
---------------	----	----	-----	-----	-------	-----------------------------------

Da li su prikazi pokrivenosti koda za sva tri scenarija isti i zašto?

Da, pokrivenost koda za sva tri scenarija je ista. Razlog za to je što su u sva tri scenarija pokrivene sve linije koda metode, tj. nema nedostižnih linija koda niti za jedan scenarij.

Na koji način se iz pokrivenosti koda može doći do zaključka koja strategija testiranja je iskorištena?

U našem slučaju iz pokrivenosti koda ne možemo zaključiti koja strategija testiranja je iskorištena jer za sve strategije testiranja imamo istu pokrivenost koda.

Koje su prednosti, a koje mane korištenja svakog od pojedinačnih pristupa testiranja?

-Obuhvat uslova- glavni nedostatak ovog pristupa testiranja je eksponencijalni rast broja slučajeva koji se trebaju testirati sa brojem uslova, a prednost ovog pristupa jeste što se svaki logički iskaz zasebno ispituje te ukoliko se ustanovi da u svakom slučaju ispravno radi testiranje tog dijela koda daje izvrsne rezultate.

-Obuhvat odluka - glavna prednost ovog pristupa testiranja je činjenica da je potreban mali broj testnih slučajeva da se izvrše svi mogući pravci kretanja

-Obuhvat petlji- glavni nedostatak ovog pristupa je činjenica da je potreban veliki broj testnih slučajeva za testiranje svih uslova ali s obzirom da su petlje osnovni dio skoro svakog algoritma, pristup testiranju iz ovog aspekta