UNIVERZITET U BEOGRADU
Elektrotehnički fakultet
Katedra za računarsku tehniku i informatiku

Testiranje softvera

Vežbe - Integraciono testiranje

Profesor:
dr Dragan Bojić

Asistent:
dipl ing Dražen Drašković

Beograd, 16.12.2012.

Teorijske osnove

Jedinično testiranje (eng. *Unit testing*) osigurava da su svi pojedinačni moduli softverskog sistema testirani i da svaki od njih pojedinačno radi ispravno. Jedinično testiranje ipak ne garantuje da li će ovi moduli raditi u redu ukoliko se integrišu u sistem. Primećuje se da se neke greške javljaju tek kada se moduli spoje. Integraciono testiranje (eng. *Integration testing ili Integration and Testing*) je faza u testiranju softvera u kojoj se pojedinačni moduli softverskog sistema kombinuju i testiraju kao grupa i tako se otkrivaju greške. Ovo testiranje se dešava pre sistemskog, a nakon jediničnog testiranja.

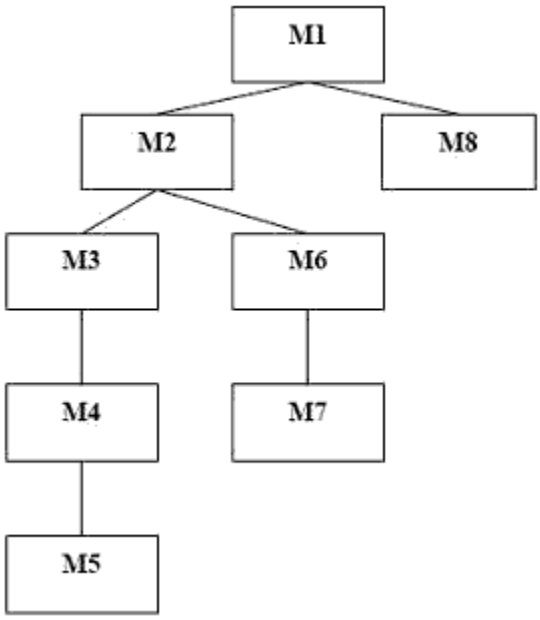
Mogu se javiti sledeće greške:

- Podaci mogu biti izgubljeni kroz interfejs. To su podaci koji izlaze iz modula, ali ne idu u željeni modul.
- Podfunkcije kada se kombinuju, ne mogu dati rezultat kao željena funkcija.
- Pojedinačno prihvatljive nepreciznosti, mogu biti uvećane do nekog neprihvatljivog nivoa. Na primer, neka u jednom modulu postoji tolerancija za grešku ±10 jedinica. Neka drugi modul ima istu preciznost. Ukoliko kombinujemo ta dva modula, pretpostavimo da u nekom trenutku može da se pojavi greška i u jednom i u drugom modulu, u tom slučaju preciznost će biti ±100, što je neprihvatljivo za sistem.
- Globalne strukture podataka mogu predstavljati problem: na primer u sistemu postoji zajednička memorija. Neka se moduli kombinuju i pristupaju toj zajedničkoj globalnoj memoriji. Zato što u jednom trenutku funkcije iz različitih modula mogu da pristupaju memoriji, može nastati problem nedostatka memorije (low memory problem).

Cilj integracionog testiranja je da se moduli, koji su jedinično testirani, integrišu, zatim da se pronađu greške, da se te greške uklone i da se izgradi celokupna struktura sistema, kao što je predviđeno dizajnom.

Vrste integracionog testiranja, zasnovane na dekompoziciji (eng. *Decomposition-Based Integration*):

- Integracija od vrha ka dnu (*Top-down testing*)
- Integracija od dna ka vrhu (*Bottom-up testing*)
- Mešovita integracija (*Sandwich testing*)
- Integracija po principu „velikog praska“ (*Big Bang*)



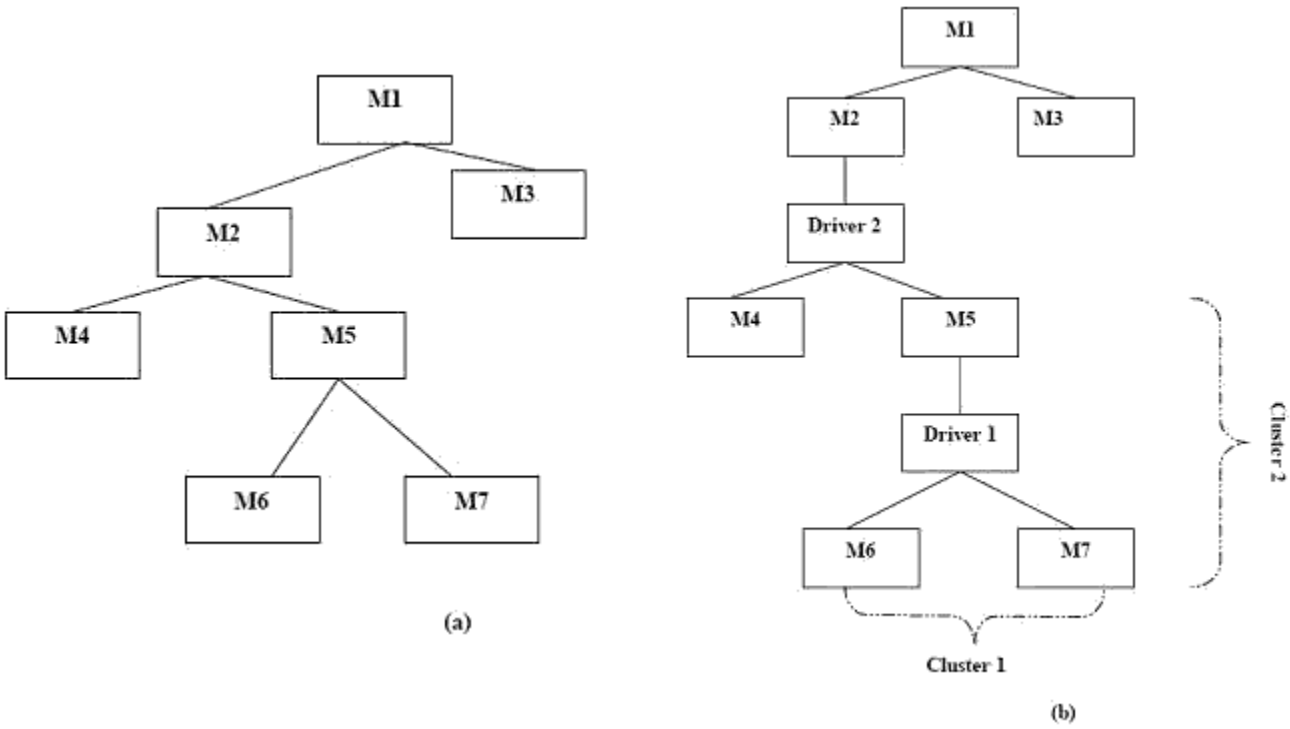
Integracija od vrha ka dnu (*Top-down testing*) je inkrementalni pristup izgradnji programske strukture. U integraciji odozgo nadole, prvo identifikujemo kontrolnu hijerarhiju, počevši od glavnog kontrolnog modula (glavnog programa).

Postoje strategije po dubini i po širini.

U strategiji po dubini, prvo se integrišu svi moduli na glavnoj kontrolnoj putanji. Za primer sa slike, redosled integracije bi bio: (M1, M2, M3), M4, M5, M6, M7 i M8.

Strategija po širini, uključuje sve neposredno podređene komponente na jednom nivou krećući se kroz srukturu programa horizontalno. Koristeći strategiju po širini, redosled integracije bi bio: (M1, M2, M8), (M3, M6), M4, M7 i M5.

Integracija od dna ka vrhu (*Bottom-up testing*) podrazumeva konstrukciju i testiranje počevši od najnižih modula u hijerarhiji. Pošto se komponente integrišu od dna ka vrhu, obrada podataka koje se vršila u podređenim nivoima hijerarhije je sada dostupna (jer su i podređeni moduli već integrisani), pa se zbog toga i stabovi suvišni.



Koraci po kojima se radi integracija odozdo nagore:

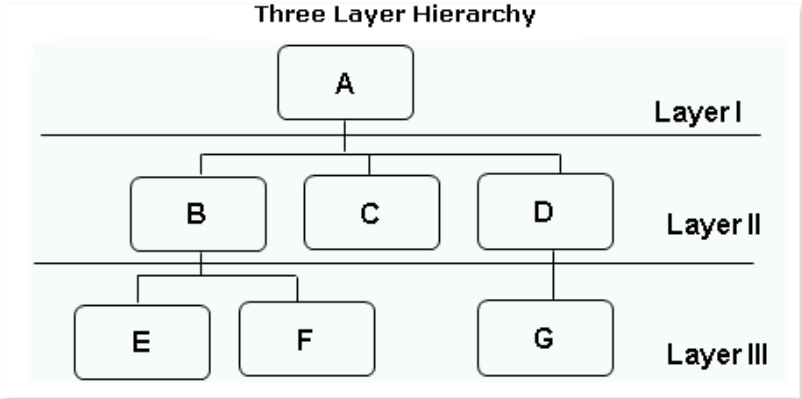
1. Moduli na nižem nivou se kombinuju u klaster (eng. *cluster*) koji vrše neku podfunkciju softverskog sistema. Nazivaju se još i grozdovi.
2. Pravi se drajver (kontrolni program za testiranje) koji koordiniše testove na ulazu i izlazu.
3. Testira se klaster.
4. Drajveri se uklanjaju, a klasteri se kombinuju i pomeraju naviše u programskoj strukturi.

Na slici je prikazano kako radi integracija odozdo nagore. Uvek kada dodajemo novi modul kao deo integracionog testiranja, menjamo strukturu programa. To mogu biti novi tokovi podataka, neki novi ulazi ili izlazi ili neke nove kontrolne logike. Te izmene mogu da prouzrokuju mnoge probleme jer bi se promenila funkcionalnost testiranih modula, koji su ranije radili ispravno.

Da bi se otkrile greške, treba primeniti regresivno testiranje. Regresivno testiranje je ponovno izvršavanje nekog podskupa testova koji su već sprovedeni, kako bi se osiguralo da promene ne izazovu neke neželjene sporedne efekte ili nove greške.

Mešovita integracija (*Sandwich testing*) je pristup u kome se kombinuje integraciono testiranje od vrha ka dnu, sa testiranjem od dna ka vrhu. Kod ovog testiranja postoje sledeća tri sloja:

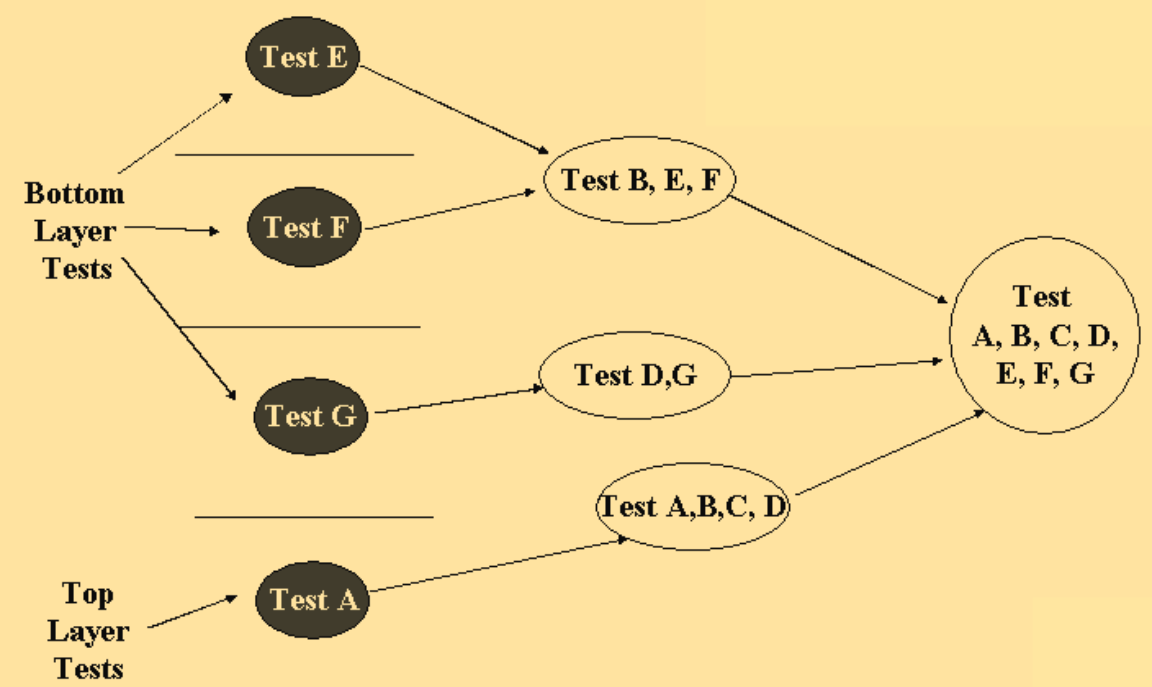
- Ciljni sloj, koji je izabran u sredini
- Sloj iznad ciljnog sloja
- Sloj ispod ciljnog sloja



Testiranje konvergira ka ciljnom sloju.

Kako izabrati ciljni sloj, ako postoji više od 3 nivoa?

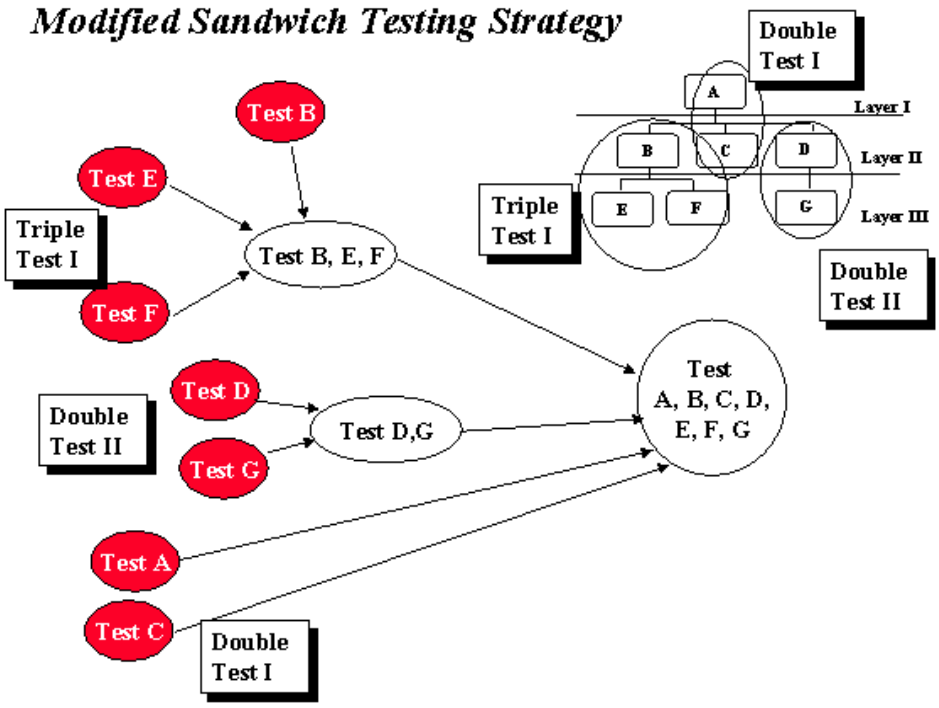
Heuristika: pokušati da se minimuzuje broj stabova i drajvera



Prednosti i nedostaci mešovite integracije:

- Testovi na gornjim i donjim nivoima mogu se raditi u paraleli
- Ne testiraju se temeljno i pojedinačno podsistemi, pre nego što se integrišu
- Rešenje: modifikovana mešovita integracija (eng. modified sandwich testing strategy)

Modifikovana mešovita integracija



- Testovi u paraleli:
    - Srednji sloj sa drajverima i stabovima
    - Sloj na vrhu sa stabovima
    - Sloj na dnu sa drajverima

    - Sloj na vrhu pristupa srednjem sloju (sloj na vrhu zamenjuje drajvere)
    - Srednji sloj pristupa sloju na dnu (sloj na dnu zamenjuje stabove)

Koraci u testiranju zasnovano na komponentama:

1. Na osnovu strategije integracije, izabrati komponentu koja se testira. Sve klase u komponenti testirati jediničnim testiranjem.
2. Spojiti komponente koje zajedno testiramo, uraditi ukoliko je neophodno neke preduslove za integraciono testiranje (drajveri, stabovi).
3. Uraditi funkcionalno testiranje: definisati testove koji pokrivaju sve slučajeve za izabranu komponentu.
4. Uraditi testiranje strukture: definisati testove koji vrše testiranje izabrane komponente.
5. Izvršiti testove performansi.
6. Voditi evidenciju o testovima i aktivnostima.
7. Ponavljati korake 1-7 dok se ceo sistem potpuno ne testira.

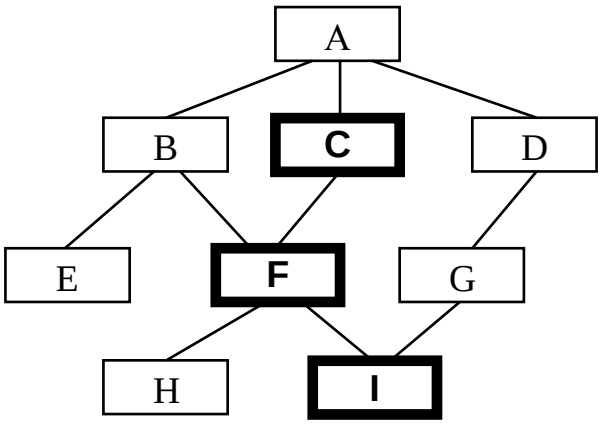
Zadatak 1

Sistem sa slike integraciono se testira pristupom od vrha ka dnu, u nizu koraka pri čemu se u svakom dodaje novi modul već postojećim. Podebljano su označeni kritični moduli. Skicirati šta tačno sadrži deo sistema koji se testira (koje module i stabove) u četvrtom koraku testiranja ako se primenjuje strategija:

a) po širini

b) po dubini

c) navesti karakteristike kritičnih modula kod integracionog testiranja



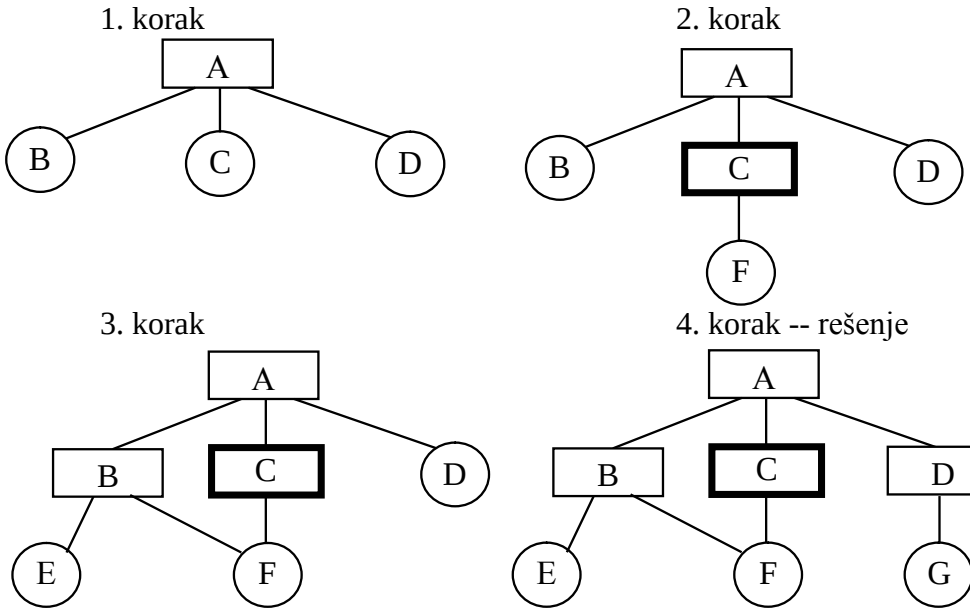
Teorijske osnove

Postupak integracionog testiranja od vrha ka dnu, kako je objašnjeno u materijalima sa predavanja, sastoji se iz sledećih aktivnosti:

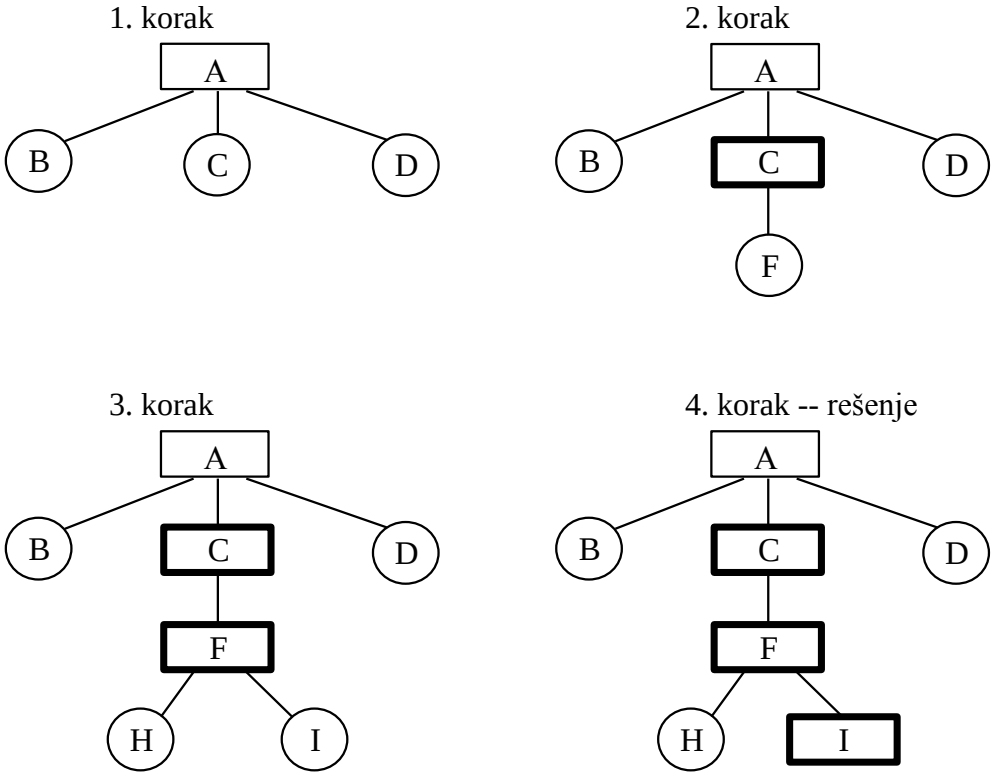
1. Glavni modul se koristi kao pokretač testova, a umesto svih njegovih sledbenika u grafu programa koriste se njihovi stub-ovi.
2. Izvršava se odgovarajući skup testova.
3. Prema vrsti testiranja (po širini ili po dubini), jedan od stub-ova se zamenjuje stvarnim modulom.
4. Po potrebi se sprovodi regresivno testiranje.
5. Ponavljaju se koraci 2. do 5. sve dok se kompletna programska struktura ne integriše i istestira.

Rešenje

a) "po širini": integracija ide po nivoima programske strukture, prvo se integrišu sve komponente na nivou N, pa se potom prelazi na nivo N + 1. Kvadrati su implementirani moduli, krugovi predstavljaju stub-ove.



b) "po dubini": integracija ide po glavnoj kontrolnoj strukturi koja obuhvata kritične module.



c) Kritični modul (koji treba testirati što je pre moguće) poseduje jednu ili više od sledećih karakteristika:

(1) zadovoljava veći broj funkcionalnih zahteva
(2) poseduje visok stepen kontrole (nalazi se relativno visoko u programskoj strukturi)

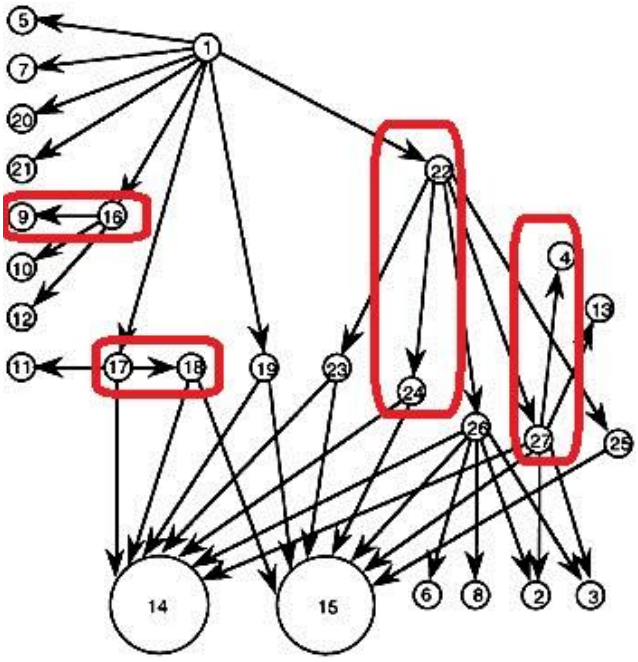
(3) kompleksan je ili podložan greškama (kao indikator može se koristiti ciklomatska kompleksnost)
(4) zahtevane su konačne performanse

Teorijske osnove

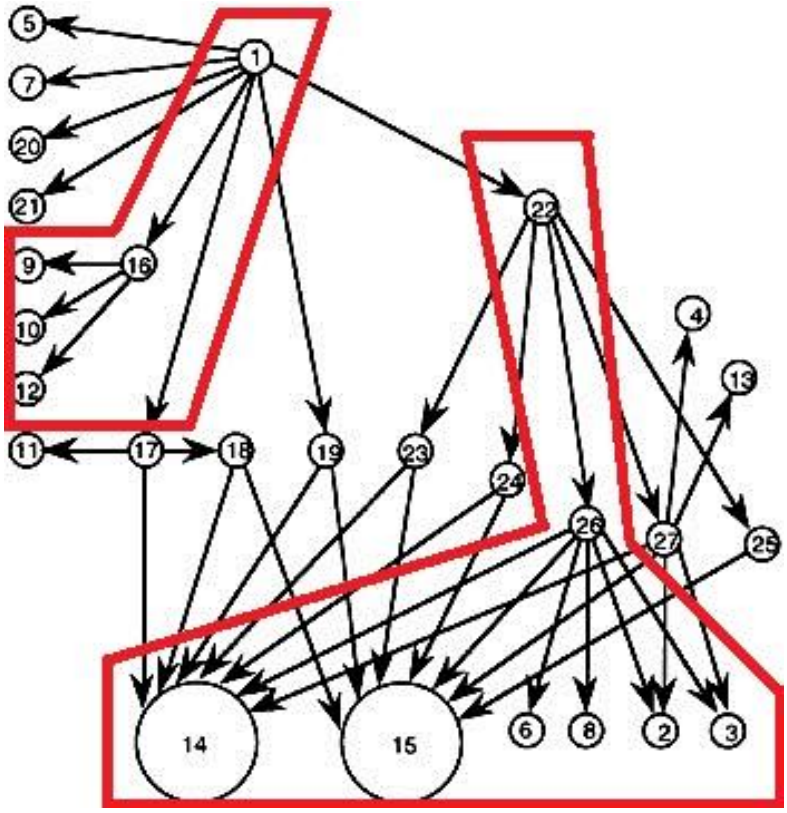
Jedan od nedostataka integracija zasnovane na dekompoziciji je zato što je osnova te integracije funkcionalna dekompozicija stabla. Ako umesto toga, koristimo usmerene grafove u integracionom testiranju (engl. *Call Graph-Based Integration*), mi ćemo ublažiti ovaj nedostatak, a takođe ćemo ostati u strukturalnom testiranju. Ideja je da program predstavimo preko usmerenog grafa, što nas dovodi do novih pristupa u integracionom testiranju:

- Integracija parova (engl. *Pairwise Integration*)
- Integracija suseda (engl. *Neighborhood Integration*)

Ideja koja je zastupljena kod integracije parova je da se eliminiše razvijanje stub-ova i driver-a. Umesto da se razvijaju stub-ovi i driver-i, zašto ne koristiti originalni programski kod? Na prvi pogled, ovo deluje kao integracija metodom velikog praska, ali ovde ograničavamo sesiju na samo par jedinica po grafu. Krajnji rezultat je da imamo po jednu sesiju integracionog testiranja za svaku vezu (svaki poziv) u grafu. Na slici su prikazane 4 sesije integracije parova, od ukupno 40, u nekom softverskom sistemu.



Pojam susedstva u topologiji možemo još više da proširimo, tako da ne obuhvatamo samo jedan par čvorova u grafu. Susedstvo radijusa 1 nekog čvora u grafu predstavlja skup čvorova koji su na udaljenosti 1 od tog čvora. U usmerenom grafu, taj skup obuhvata sve neposredne prethodnike tog čvora i sve neposredne sledbenike tog čvora (primititi da oni obuhvataju skup stub-ova i skup driver-a tog čvora). Na slici je prikazana integracija suseda za čvorove 16 i 26.



Ukupno u ovom sistemu postoji 11 sesija integracionog testiranja suseda:

Čvor	Prethodnici	Sledbenici
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15
23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	-	5, 7, 2, 21, 16, 17, 19, 22

Broj susedstva za određeni usmereni graf može da se izračuna na više načina. Broj unutrašnjih čvorova se može izračunati na sledeći način:

Broj_unut_čvorova = Broj_čvorova - (Broj_čvorova_na_vrhu + Broj_čvorova_na_dnu)

Broj_susedstva = Broj_unut_čvorova + Broj_čvorova_na_vrhu

Broj_susedstva = Broj_čvorova - Broj_čvorova_na_dnu

U našem primeru:

Broj_čvorova = 27
Broj_unut_čvorova = 10
Broj_čvorova_na_vrhu = 1
Broj_čvorova_na_dnu = 16

Broj_susedstva = 11

Integracija sustedstva daje drastično smanjenje broja test sesija integracije i izbegava stvaranje stub-ova i driver-a. Krajnji rezultat je u suštini susedstvo sendviča, o kojima smo već pričali (ali naravno drugačiji, jer je baza informacija za susedstva pozivanje usmerenog grafa, a ne razgradnja stabla kao kod sendvič integracije). Integraciono testiranje pomoću suseda mnogo teže izoluje greške od integracije metodom „srednjeg praska“ (sendvič).

Zadatak 2

Prikazati pojedinačne grafove svih modula u programu *integrationNextDate*, koji izračunava i ispisuje sledeći datum. Izvršiti funkcionalnu dekompoziju svih modula korišćenjem stabla i nacrtati usmereni graf. Za module smatrati procedure i funkcije.

```
1.   Main integrationNextDate
     Type Date
          Month As Integer
          Day as Integer
          Year as Integer
     EndType
     Dim today As Date
     Dim tomorrow As Date
2.   GetDate(today)
3.   PrintDate(today)
4.   tomorrow = IncrementDate(today)
5.   PrintDate(tomorrow)
6.   End Main
```

```
7.   Function isLeap(year)              Boolean
8.   If (year divisible by 4)
9.   Then
10.       If (year is NOT divisible by 100)
11.            Then isLeap = True
12.       Else
13.            If (year is divisible by 400)
14.                 Then isLeap = True
15.            Else isLeap = False
16.            EndIf
17.       EndIf
18.  Else isLeap = False
19.  EndIf
20.  End (Function isLeap)

21.  Function lastDayOfMonth(month, year)    Integer
22.       Case month Of
23.            Case 1:   1, 3, 5, 7, 8, 10, 12
24.                      lastDayOfMonth = 31
25.            Case 2:   4, 6, 9, 11
26.                      lastDayOfMonth = 30
27.            Case 3:   2
28.                      If (isLeap(year))
29.                           Then lastDayOfMonth = 29
30.                           Else lastDayOfMonth = 28
31.                      EndIf
32.       EndCase
33.  End (Function lastDayOfMonth)

34.  Function GetDate(aDate)       Date
          dim aDate As Date
35.       Function ValidDate(aDate)     Boolean
          dim aDate As Date
               dim dayOK, monthOK, yearOK as Boolean
36.            If ((aDate.Month > 0) AND (aDate.Month <=
12)
37.                 Then monthOK = True
38.            Else monthOK = False
39.            EndIf
40.            If (monthOK)
41.                 Then
```

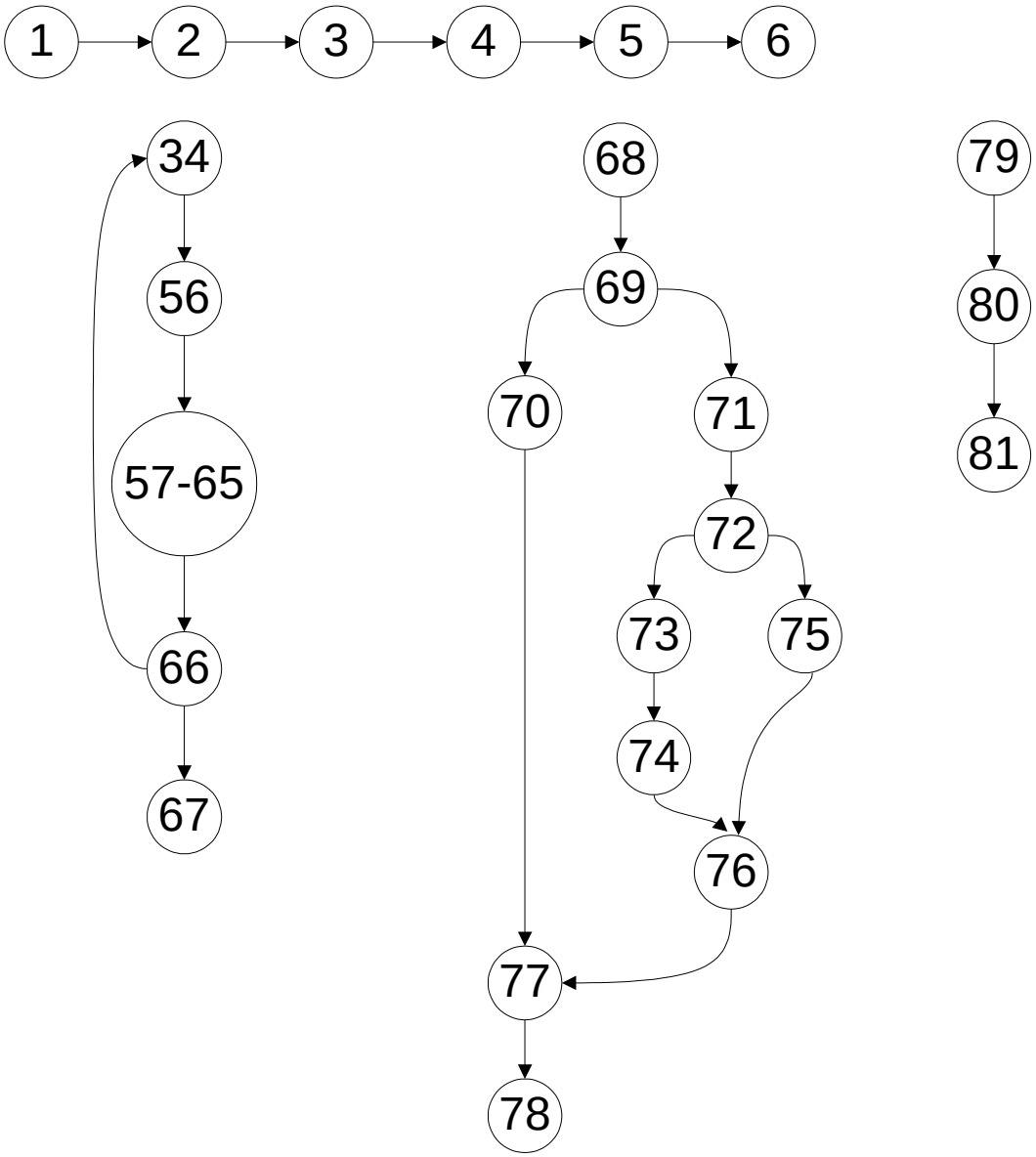
```
42.            If ((aDate.Day > 0) AND (aDate.Day
               <=   lastDayOfMonth(aDate.Month,aDate.Year))
43.                      Then dayOK = True
44.                      Else dayOK = False
45.                      EndIf
46.            EndIf
47.       If ((aDate.Year > 1811) AND (aDate.Year <= 2012))
48.            Then yearOK = True
49.            Else yearOK = False
50.       EndIf
51.       If (monthOK AND dayOK AND yearOK)
52.            Then ValidDate = True
53.            Else ValidDate = False
54.       EndIf
55.       End (Function ValidDate)

GetDate telo funkcije
56.  Do
57.       Output("Unesite mesec?")
58.       Input(aDate.Month)
59.       Output("Unesite dan?")
60.       Input(aDate.Day)
61.       Output("Unesite godinu?")
62.       Input(aDate.Year)
63.       GetDate.Month = aDate.Month
64.       GetDate.Day = aDate.Day
65.       GetDate.Year = aDate.Year
66.  Until (ValidDate(aDate))
67.  End (Function GetDate)
68.  Function IncrementDate(aDate)      Date
69.       If(aDate.Day < lastDayOfMonth(aDate.Month))
70.            Then aDate.Day = aDate.Day + 1
71.            Else aDate.Day = 1
72.                 If (aDate.Month = 12)
73.                      Then aDate.Month = 1
74.                      aDate.Year = aDate.Year + 1
75.                 Else aDate.Month = aDate.Month + 1
76.                 EndIf
77.       EndIf
78.  End (IncrementDate)
```

```
79.  Procedure PrintDate(aDate)
80.       Output ("Danas je ", aDate.Month, "/", aDate.day,
"/", aDate.Year)
81.  End (PrintDate)
```

Rešenje

Prvo ćemo nacrtati grafove glavnog programa (Main) i modula prvog nivoa (GetDate, IncrementDate i PrintDate):



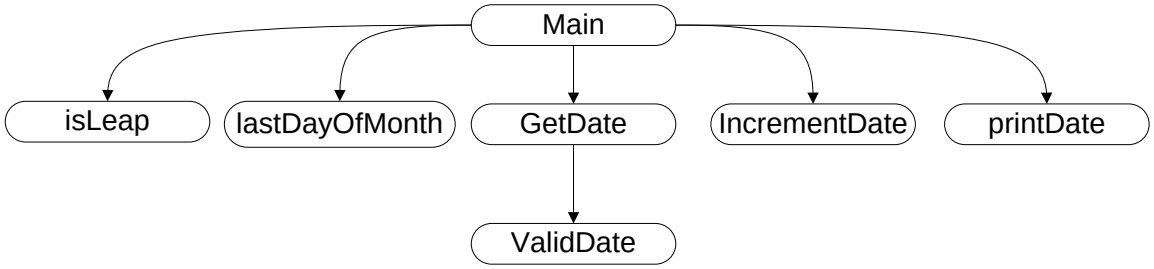
Izračunaćemo vrednost ciklomatske kompleksnosti za ove module:

Main: V(G) = 1
GetDate: V(G) = 2
IncrementDate: V(G) = 3
PrintDate: V(G) = 1

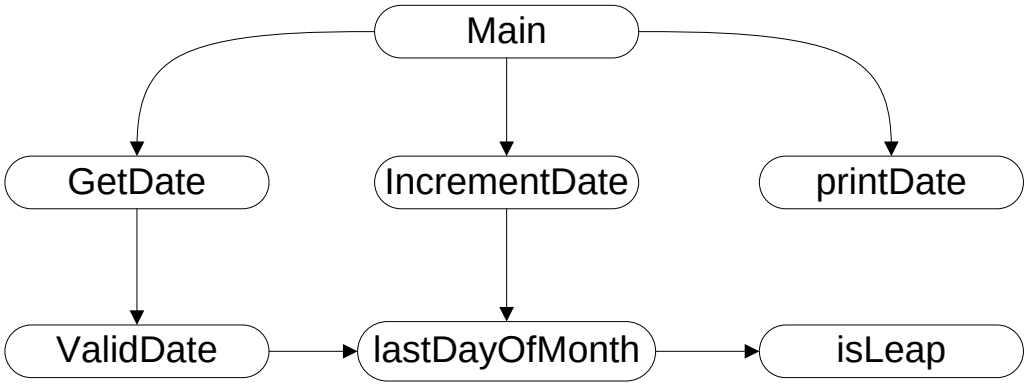
Slično treba nacrtati grafove modula nižih nivoa. Njihove ciklomatske kompleksnosti su:

isLeap: V(G) = 4
LastDayOfMonth: V(G) = 4
ValidDate: V(G) = 6

Integracionim testiranjem dobijamo funkcionalnu dekompoziciju u vidu stabla:



Integracionim testiranjem dobijamo sledeći usmereni graf:



Zadatak 3

Navesti i ukratko objasniti različite kriterijume za završetak testiranja.

Rešenje

Osnovno pitanje testiranja je: Kada završiti sa testiranjem?

Trivijalna rešenja nisu dobra:

1. okončati ga kada istekne vreme koje smo isplanirali za testiranje
2. okončati ga kada prodju svi izabrani test-problemi

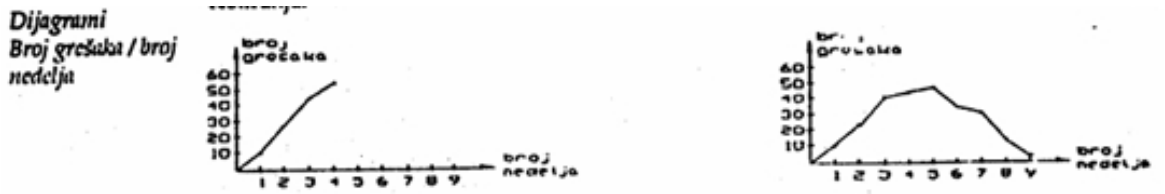
Bolji pristup

3. odabrati metode testiranja koje sistem mora da prodje, ali ni to nije dovoljno dobro, jer za različite sisteme neke su metode bolje, a neke lošije

Efikasniji kriterijum je

4. Testiranje dok se ne otkrije zadati broj grešaka. Taj broj se ne daje nasumice, nego na osnovu iskustva i/ili matematičkih modela. U slučaju da je broj grešaka manji od predvidjenog, biramo krajnji kriterijum
5. vodjenje evidencije o broju pronadjenih grešaka u odredjenom vremenskom intervalu

Kombinacijom 3), 4) i 5) najbolje se predvidja ternutak okončanja testiranja.



Ako se broj grešaka koje se otkriju za 7 dana povećava (prvi grafik), tada nema ni govora o prestanku testiranja. Druga kriva pokazuje da se posle 9 nedelja broj grešaka smanjio dovoljno da se može okončati testiranje.

Metod testiranja ne treba menajti nasumice, bez uvida u broj otkrivenih grešaka primenom svake pojedine metode.
Metod testiranja se menja tek kada broj otkrivenih grešaka spadne na neki mali, zadovoljavajući broj.

Zadatak 4

Dat je programski sistem od 10000 linija koda. Posle inspekcije koda, procena je da na svakih 100 iskaza ima 5 grešaka, pri čemu je 40% nastalo u kodiranju, a ostalo u projektovanju. Sistem se testira primenom jediničnog (back and white box), integracionog i sistemskog testiranja pri čemu se koriste podaci sa ranijih projekata prikazani u tabeli. Definisati kriterijume završetka pojedinih vrsta testiranja, ako je za ceo proces testiranja na raspolaganju najviše 11 meseci.

Врста тестирања	Откривене грешке кодирања	Откривене грешке пројектовања	Утрошено време
Јединично (white box)	65%	0%	36%
Јединично (black box)	30%	60%	36%
Интеграционо & Системско	3%	35%	28%
Тотал	**98%**	**95%**	**100%**

Rešenje

Ukupna procena je 500 grešaka (10000/100 * 5), od toga 200 u kodiranju (40%), a 300 u projektovanju (60%).

Naš cilj je da testiranjem uklonimo 98% grešaka u kodiranju i 95% u projektovanju, odnosno 196 i 285 respektivno.

Jedinično white box testiranje može da uoči samo greške implementacije. Ovo testiranje završavamo u trenutku kada je pronadjeno 130 grešaka (65%), odnosno nakon isteka 4 meseca.
Jedinično black box testiranje završavamo u trenutku kada je pronadjeno 60+180=240 grešaka (30% + 60%), odnosno nakon isteka 4 meseca.

Integraciono i sistemsko testiranje završavamo kada se otkrije 6+105=111 grešaka, odnosno nakon isteka 3 meseca.

Zadatak 5

Nabrojati (i kratko komentarisati) klase grešaka koje se mogu utvrditi metodom inspekcije koda.

Rešenje

Metoda inspekcije koda - obično je vrši tim od četiri člana. Jedan je koordinator, obično iskusni programer, ali nikako autor. Nekoliko dana pre sastanka on timu deli listing programa i radne zadatke. Zakazuje i vodi sastanak, beleži otkrivene greške i drugo. Drugi član tima je autor, treći programer, a četvrti stručnjak za testiranje. Najpre autor objašnjava logiku programa, instrukciju po instrukciju i tada on u interakciji sa ostalima i sam, otkriva najviše grešaka. Program se analizira prema listama najčešćih grešaka u programima, koje treba da ima svaka softverska kuća. Ukoliko se na kraju sastanka otkrije veliki broj grešaka, zakazuje se novi sastanak, a po potrebi dopunjava lista grešaka. Optimalno, sastanak traje 90 minuta (150 instrukcija na sat). Naravno, sve ovo ne isključuje testiranja koja autor vodi samostalno.

Lista grešaka prema Glenfordu Myersu

Klasa grešaka pri definisanju podataka

1. Da li u programu postoje nedefinisane i/ili neinicijalizovane promenljive?
2. Da li svi indeksi imaju vrednosti u okviru deklarisanih dimenzija?
3. Da li svi indeksi polja imaju celobrojnu vrednost?
4. Da li vodi računa da su različiti podaci kojima se dodeljuje isti memorijski prostor istog tipa?
5. Da li je vrednost promenljive, tipa i osobine drugačija od onih koje kompajler očekuje?
6. Ako se podaci koriste u više potprograma, da li su oni tamo definisani na isti način?

Klasa grešaka pri deklarisanju podataka

1. Da li su deklaracije promenljivih implicitno date? (primer u Fortran-u: izraz X=A(1) ako A nije definisano, biće shvaćen kao poziv funkcije)
2. Da li su promenljive ispravno inicijalizovane u deklarativnim naredbama?
3. Da li je svaka promenljiva korektne dužine i tipa?
4. Postoje li promenljive sličnih imena? (imena treba da imaju mnemonička svojstva i ne smeju da budu slična, da se ne bi pomešala)

Klasa grešaka pri izračunavanju izraza

1. Da li postoje izrazi koji sadrže nekonzistentne tipove podataka?
2. Ako jezik dozvoljava mešovite izraze, da li su u mešanju poštovana pravila konverzije?
3. Da li je prilikom izračunavanja nekog izraza došlo do toga da je vrednost sa desne strane veća od formata promenljive sa leve strane?
4. Da li u procesu izračunavanja može doći do prekoračenja?
5. Može li se negde desiti da delilac postane jednak nuli?
6. Da li sistemske netačnosti mogu da utiču na rezultat ako se računa u pokretnom zarezu? (primer: 10 * 0,1)
7. Da li se vrednost promenljivih kreće unutar realnih, tj. očekivanih opsega?
8. Da li su izrazi pisani vodeći računa o prioritetu operatora?
9. Da li postoji neka nedozvoljena upotreba celobrojne aritmetike? (primer: 2 * I/2, samo ako je I parno)

Klase grešaka proisteklih iz poređenja

1. Postoji li poređenje nekonzistentnih tipova?
2. Da li se porede promenljive različitog formata?
3. Da li su operatori poređenja korektni?
4. Da li su svi Bulovi, i uopšte relacioni izrazi korektni? (treba maksimalno iskoristiti zagrade, zbog kompajlera)

Klasa grešaka u kontroli toka

1. Ako program sadrži višestruka grananja, može li indeksirana promenljiva da primi sve vrednosti za grananje?
2. Da li svaka petlja ima završetak?
3. Da li svaki potprogram ima svoj završetak?
4. Da li je moguće da se neka petlja nikada ne izvrši?
5. Za petlje koje se kontrolišu iteracijama i Bulovim izrazima proveriti šta se dešava ako se uslovi iz petlje ne ostvare.
6. Proveriti da nema sečenja kontrolnih struktura, jer su dozvoljena samo ugnježđavanja.
7. Da li ima odluka donesenih primenom logike eliminacije? (primer: ako ulazna promenljiva može biti samo 1, 2 i 3, ne treba izbegavati proveru i treće vrednosti)

Klasa grešaka pri komunikaciji sa potprogramima

1. Da li je broj primljenih argumenata od strane potprograma jednak broju očekivanih?
2. Da li osobine poslatih argumenata po tipu, formatu i veličini odgovaraju onome što potprogram očekuje?
3. Da li su poslati argumenti u istim jedinicama koje očekuje potprogram? (na primer: mešanje stepena i radijana)
4. Da li potrogram menja argument koji ima samo funkciju ulazne veličine? (primer: CALL SUB(J,3). Stvarni drugi argument predstavlja adresu konstante i ako se desi da u potprogramu menjamo drugi argument, promenićemo konstantu)

Klasa ulazno/izlaznih grešaka

1. Da li je format u skladu sa ulazno-izlaznim izrazom, tj. da li je specifikacija u FORMAT naredbi u skladu sa onom u READ naredbi?
2. Da li su sve datoteke otvorene pri korišćenju?
3. Da li su uslovi za kraj datoteke detektovani ispravno?
4. Ukoliko je datoteka eksplicitno otvorena, proveriti da li su svi njeni atributi korektno navedeni.

Ostale greške

1. Kompajler po prevođenju ne naznačuje grešku, ali daje upozorenje da neki standard ili nešto drugo nije poštovano.
2. Ukoliko kompajler daje listinge, treba ih prekontrolisati s osvrtom na tipove promenljivih, ako se negde zaglavi.