
PROGRAMMING
SOFTWARE
BUGS
TESTING
METHODS
DATABASE
USABILITY

TESTIRANJE SOFTVERA

Bitan korak u razvoju softvera je - Testiranje softvera.

Proizvođači softvera bi želeli da predvide broj grešaka u softverskim sistemima pre nego što ih primene, kako bi mogli da procene kvalitet kupljenog proizvoda i teškoće koje se javljaju u samom procesu održavanja.

Cilj testiranja softvera je da se ustanovi da li se softver ponaša na način koji je predviđen zadatom specifikacijom.

Prema tome, PRIMARNI CILJ Testiranja softvera je otkrivanje grešaka u softveru.

TESTIRANJE SOFTVERA

Jedno od velikih pitanja koje se javlja kod testiranja softvera je reprodukcija greške (testeri otkrivaju greške a programeri otklanjaju bagove).

Očigledno je da mora postojati koordinacija između testera i programera.

Reprodukcija greške je slučaj kada bi bilo najbolje da se problematični test izvede ponovo i da znamo kad, i na kom mestu u programu i pod kojim uslovima se tačno greška desila.

Dakle, idealnog testa nema, kao što nema ni idealnog proizvoda.

TESTIRANJE SOFTVERA

- ✓ Dizajniranje i pregled test strategija, test planova i test slučaja
- ✓ Upravljanje konfiguracijom i promenama
- ✓ Prilagođavanje procesa testiranja potrebama projekta
- ✓ Metrike
- ✓ Kriterijumi za završetak testiranja

TESTIRANJE SOFTVERA

Vrste nefunkcionalnih testova:

- ✓ **Installation testing** – testiranje instalacije
- ✓ **Performance testing** – testiranje karakteristika i performansi
  - **Stress testing** testiranje opterećenja
  - **Endurance testing** – testiranje izdržljivosti
  - **Load testing** – testiranje opterećenja
  - **Spike testing (kako system funkcioniše u takozvanim ekstremnim situacijama)**
- ✓ **Documentation testing** – testiranje dokumentacije
- ✓ **Reliability testing** – testiranje pouzdanosti
- ✓ **Security testing** - bezbedonosni testovi

Mutaciono testiranje

Spada u tehnike testiranja zasnovane na defektima.

Osnovni koncept testiranja na bazi defekata je da se izaberu slučajevi testiranja koji prave razliku između programa koji testiramo i alternativnih programa koji sadrže neke greške.

Ovo se obično postiže izmenom programa koji se testira da se proizvedu "pogrešni programi".

Ovakvo testiranje se može koristiti za:

- ✓ procenu temeljitosti (adekvatnosti) serije testova,
- ✓ za izbor novih test slučajeva da se dodaju seriji testova,
- ✓ da se proceni broj defekata u programu

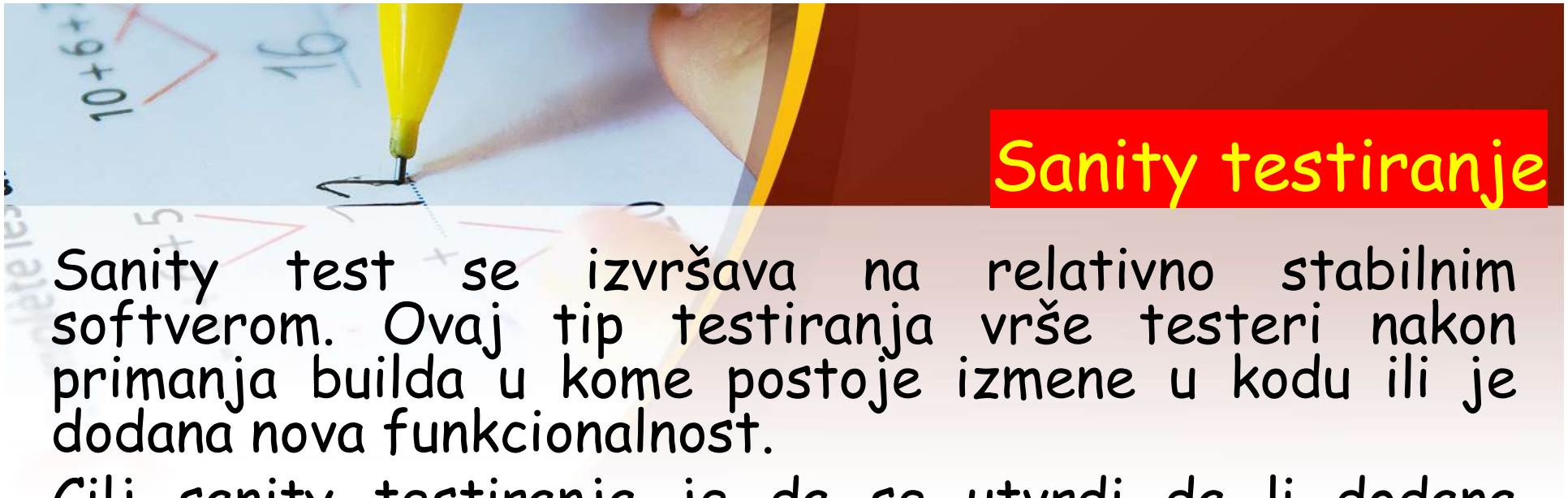
Sanity testiranje

U nekim literaturama se nalazi pod nazivom – test zdravog razuma (Sanity test). On je veoma sličan smoke testu.

Ovi termini se često mešaju u testiranju softvera.

Iako razlike između ova dva tipa nisu velike, one ipak postoje.

- Smoke test se koristi za testiranje builda softvera da bi se verifikovao ispravan rad osnovnih funkcionalnosti i izvršava se pre bilo kojeg detaljnog testiranja, kako testeri ne bi gubili vreme u slučaju nekih kritičnih problema.

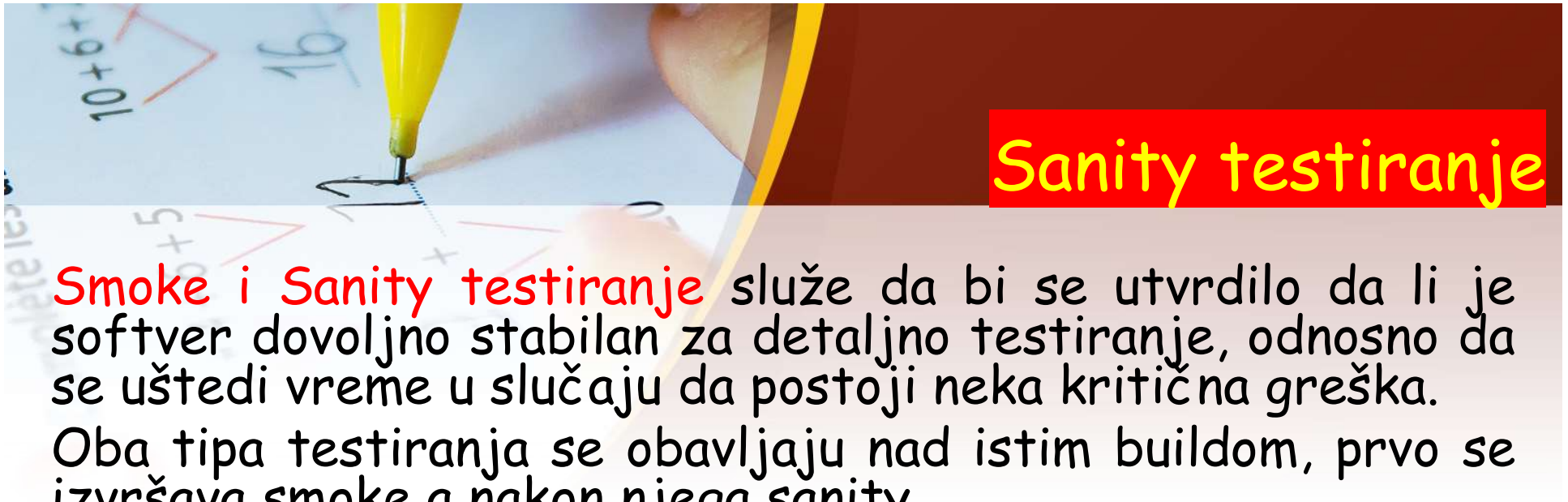


Sanity testiranje

Sanity test se izvršava na relativno stabilnim softverom. Ovaj tip testiranja vrše testeri nakon primanja builda u kome postoje izmene u kodu ili je dodana nova funkcionalnost.

Cilj sanity testiranja je da se utvrdi da li dodana funkcionalnost radi kako treba i da osnovne funkcionalnosti softvera nisu ugrožene.

Ako sanity test ne prođe, build se odbija i ne počinje se dalje testiranje, čime se štedi vreme i smanjuju troškovi.



Sanity testiranje

Smoke i Sanity testiranje služe da bi se utvrdilo da li je softver dovoljno stabilan za detaljno testiranje, odnosno da se uštedi vreme u slučaju da postoji neka kritična greška.

Oba tipa testiranja se obavljaju nad istim buildom, prvo se izvršava smoke a nakon njega sanity.

Ovi tipovi testiranja se mogu izvršiti i manuelno ili nekim alatom za automatizaciju.

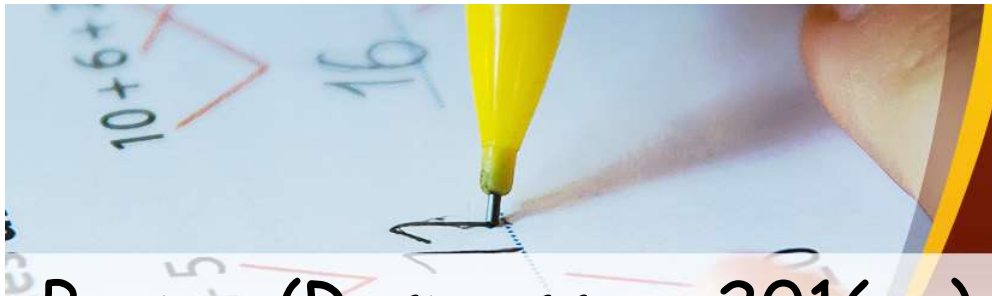
U slučaju da se koristi neki alat za automatizaciju, najčešće se pokretanje tih testova inicijalizira sa istim procesom koji pravi sam build.

Smoke testiranje

Smoke testiranje obuhvata niz jednostavnih testova koji za cilj imaju potvrdu rada najvažnijih funkcija sistema.

Ovaj tip testiranja se radi kako bi se ispitalo da li je sistem dovoljno stabilan da podrži dalje testiranje tako da se ne ulazi u "dublji" test ako ključne funkcije ne rade.

Ovaj oblik testiranja softvera nije detaljan, niti ulazi u dubinu funkcionalnosti.



Smoke testiranje

Prema (Rasmusson, 2016.,) se navode dobre osobine testiranje :

- ✓ Proverava je li aplikacija ispravno isporučena
- ✓ Proverava je li okruženje ispravno postavljeno,
- ✓ Da li su svi delovi arhitekture ispravno povezani i postavljeni pravilno.

Smoke testiranje

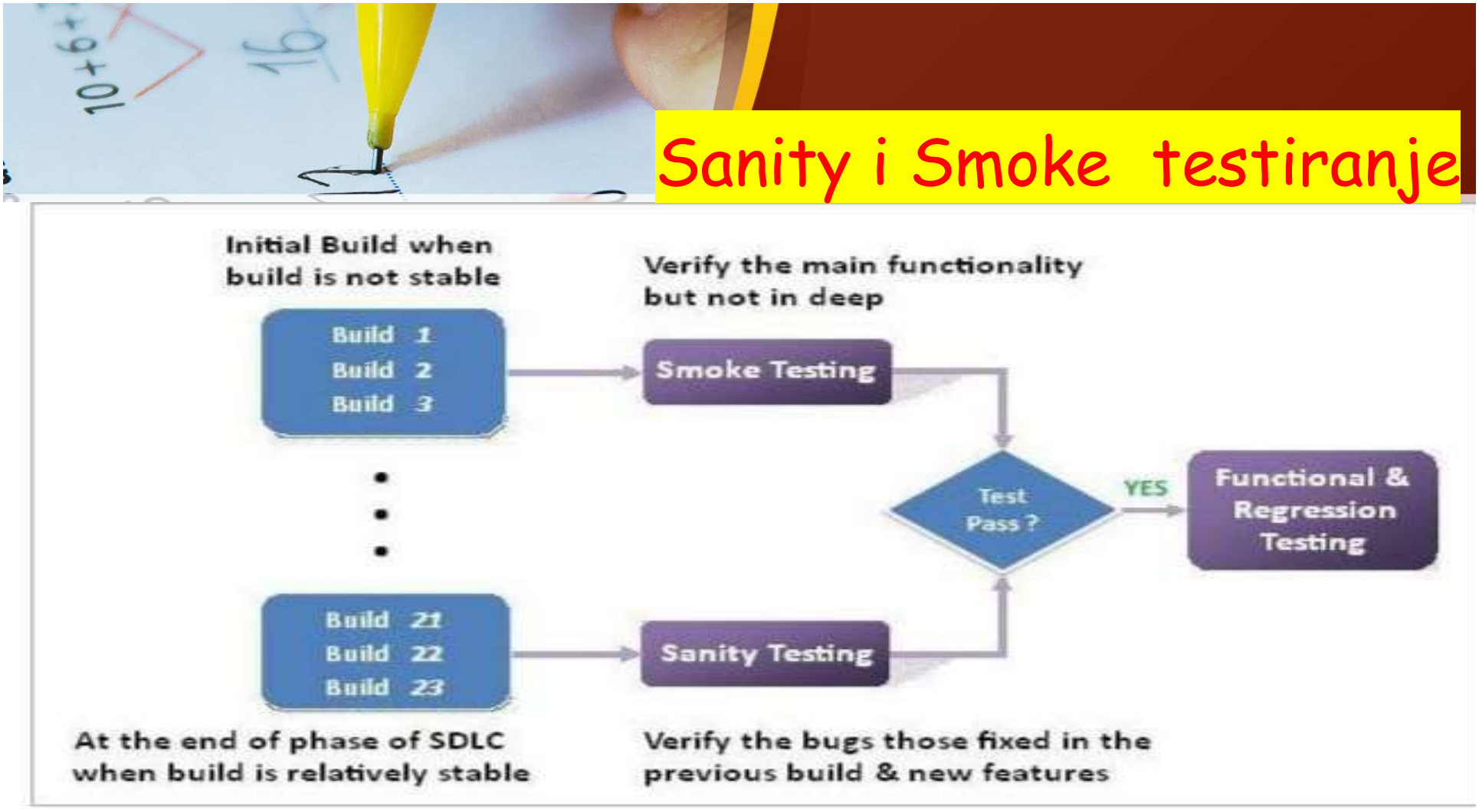
U nekim radovima se ovo testiranje navodi kao Build Verification Testing.

Smoke testiranje predstavlja test ključnih funkcija. Ukoliko su ključne funkcije ispravne, moguće je dalje testirati softver.

Prilikom svake izmene koda, stanje sistema se menja.

Zbog toga je, pre bilo kakve detaljnije provere softvera, potrebno testirati osnovne funkcionalnosti (Smoke Testing).

Ukoliko nema kritičnih softverskih grešaka, može se nastaviti sa testiranjem, a ukoliko neka od osnovnih funkcionalnosti nije ispunjena, ona obično blokira testiranje i mora se popraviti u najkraćem roku


Sanity i Smoke testiranje
Initial Build when build is not stable
Build 1
Build 2
Build 3
Verify the main functionality but not in deep
Smoke Testing
Test Pass ?
YES
Functional & Regression Testing
Build 21
Build 22
Build 23
Sanity Testing
At the end of phase of SDLC when build is relatively stable
Verify the bugs those fixed in the previous build & new features

TESTIRANJE SOFTVERA

Prihvaćeni način za dokazivanje stvari u programima je korišćenje pravila zaključivanja kao što ih je izneo Hoare.
Ova pravila se obično formulišu tako da se mogu primeniti na poslednju programsku liniju, čime se proverava jedan ili više kratkih programa i možda neke logičke formule.

Iterativnom primenom pravila zaključivanja, zadatak dokazivanja ispravnosti programa svodi se na dokazivanje programskih linija u računu verovatnoće

TESTIRANJE SOFTVERA

PODSEĆANJE:

Tipovi defekata pronađeni statičkim testiranjem:

- Nejasna i nepotpuna specifikacija zahteva
- Greške u arhitekturi
- Greške u dizajnu
- Problemi u specifikaciji interfejsa između komponenti
- Neusklađenost sa zahtevanim i postavljenim standardima
- Prekomplikovan kod

Testiranja softvera

Pozitivni i negativni test TESTIRANJA:

Pozitivni test koji predstavlja osnovni scenario korišćenja softvera. Kada korisnik ispoštuje sve procedure unosa (unosi podatke na predviđeni i korektni način) . Tada je rezultat u skladu sa očekivanim.

Dobro namerni korisnik

Dobra praksa je testiranje što većeg broja slučajeva, ako ne i svih poslovnih pravila, kako bi se na vreme identifikovali propusti. Uneti više istih set podataka nekoliko puta.

Negativni test predstavlja scenarijo u kojima je osnovna pretpostavka da neće sve da prođe kao što je očekivano.

Ne dobro namerni korisnik (strah od novog, strah od nepoznatog)

TESTIRANJE SOFTVERA

Na osnovu toga da li je i u kojoj meri program adekvatan za obavljanje datog posla, testiranje se može definisati kao aktivnost:

- ✓ Provere usklađenosti realizacije programa sa njegovom specifikacijom (neraskidivo povezano sa osobinom tzv. korektnosti programa, koja predstavlja meru u kojoj program zadovoljava specifikaciju),
- ✓ Provere njegovog ponašanja u okruženju (usko povezano sa pouzdanošću programa, koji predstavlja otpornost na ulazne podatke koji su neregularni).
-

TESTIRANJE SOFTVERA

Testiranje slabih struktura je, kao što se pojavljuje u literature je slaba varijanta strukturnog testiranja.

Slabo strukturno testiranje je lakši oblik izvođenja manualnog strukturnog testiranja.

Prema tome, cilj slabo strukturnog testiranja je da se izvrši testiranje programa po značajno manjoj ceni pre automatske podrške kada je strukturno testiranje dostupno.

TESTIRANJE SOFTVERA

Pod pojmom uspešnog testa mogu se podrazumevati dva, i to suprotna koncepta, što je i od najveće važnosti za opšti pristup testiranju.

Posmatrano sa stanovišta izrade korektnog programa, test se smatra uspešnim ukoliko nema grešaka.

No, međutim, ukoliko se uzme u obzir utrošeno vreme kao i uloženi budžet na testiranje sasvim je opravdano diskutovati o uspešnom testu u smislu dokaza da grešaka ima.

Obzirom na to da i jedan i drugi prilaz imaju smisla, u prvom slučaju govorimo o konstruktivnom pristupu testiranju, a u drugom o destruktivnom pristupu testiranju i isto tako o konstruktivno uspešnom i destruktivno uspešnom testu.

TESTIRANJE SOFTVERA

Destruktivno testiranje može se definisati kao aktivnost analize izvršavanja programa sa ciljem otkrivanja grešaka.

Primarna osobina ovog načina testiranja je mogućnost dokaza prisustva grešaka ne podrazumevajući pri tome dokaze da je program korektan.

Glavni cilj destruktivnog testiranja je otkriti što više grešaka uz najmanje troškove, tj. u najkraćem mogućem vremenskom okviru.

TESTIRANJE SOFTVERA

Destruktivni pristup testiranju podrazumeva veliki broj metoda koje se mogu podvesti pod tri strategije:

- ✓ Analiza programa (bez primene računara),
- ✓ Strategija bele kutije (White Box Strategy) ili strukturno testiranje
- ✓ Strategija crne kutije (Black Box Strategy) ili funkcionalno testiranje

TESTIRANJE SOFTVERA

Neki od problema u razvoju softvera na koje treba da se obrati pažnja:

- ✓ Precizno planiranje(resursa, budžet samog projekta, trajanja, obuke potencijalnog kadra koji bi radio na datom softveru i drugo)
- ✓ Identifikaciju, procenu i kontrolu rizika na softverskom projektu
- ✓ Utvrđivanje merenja kvaliteta softverskog proizvoda
- ✓ Kvantitativno upravljanje procesom testiranja tj. aktivnostima osiguranja kvaliteta softvera u cilju povećanja efikasnosti i efikasnosti otkrivanja i otkrivanja grešaka u toku razvoja softvera


TESTIRANJE SOFTVERA
TESTIRANJE SOFTVERA
STATIČKE
DINAMIČKE
REVIEW
WALKTROUGH
INSPECTION
CRNA KUTIJA
BELA KUTIJA
SIVA KUTIJA


White-box
ili
Metodologija strukturnog
testiranja
ili
Providna ili staklena kutija

TESTIRANJE SOFTVERA- *White-box*

Program se shvata kao otvorena (bela) kutija čija je unutrašnjost poznata. Opšti cilj testera koji koristi ovu metodu je da proveri svaku putanju u programu.

Analogno pojmu model crne kutije uveden je pojam model bele kutije, kod koga su poznati zakoni ponašanja i procesi u njemu kao dinamičkom sistemu.

Tester bira načine testiranja koda te on određuje scenarije testnog procesa.

DA Li jE OVO MOGUĆE ?

Ako imamo jednostavni program postoji jako puno različitih putanja.

Dakle, generišu se testovi koji izvršavaju sve naredbe u programu, pozivaju sve funkcije i prolaze kroz sve tokove kontrole unutar komponente

Mnogo rešenja...

TESTIRANJE SOFTVERA- *White-box*

- ✓ Podrazumeva dostupnost izvornog koda, testira se i analizira programski kod, to jest, zasniva se na manuelnoj, statičkoj analizi izvornog koda programa.
- ✓ plan testiranja se formira na osnovu strukture programa
- ✓ Dizajn konkretnog softverskog proizvoda

Opšti cilj testera je da proveri svaku putanju u programu.

Kada se vrši Testiranje nekih kompleksnih softverskih rešenja postoji mogućnost da se koriste obe (white i black box) metode.

TESTIRANJE SOFTVERA- White-box

Neke od dobrih osobina su:

- ✓ Testiranje može početi rano u projektu
- ✓ Optimizacija koda i mogućnost pronalaska skrivenih bagova (grešaka).
- ✓ Testiraju se svi delovi koda
- ✓ Metodom analize se može otkriti više (suptilnih) grešaka nego upotrebom alata za automatizovano testiranje
- ✓ Pokrivenost testovima - obezbediti da svaka linija koda bude proverena nekim testom, odnsono što testovi koriste ulaze kako bi pokrili sva grananja u kodu
- ✓ Prednost - pokrivenost testovima obezbeđuje određenu sigurnost

TESTIRANJE SOFTVERA- White-boxA

Neki od nedostaka su:

- Vremenski zahtevno
- Kod korisničkog unosa prilikom izvršavanja programa, je teško utvrditi da li postoji neki bagovi (grešaka)ili ne
- Mana - veliki broj testova je neupotrebljiv čim se kod promeni
- Testovi jako kompleksni i potrebno je da ih piše i testira veoma stručan tester koji dobro poznaje razvoj softvera
- Testiranje bele kutije oduzima puno vremena, potrebno je vreme za potpuno testiranje većih aplikacija.
- Nemogućnost testiranja softverskih performansi (vreme odgovora, pouzdanost, load)

TESTIRANJE SOFTVERA- *White-box*

Analogno pojmu model crne kutije uveden je pojam model bele kutije, kod koga su poznati zakoni ponašanja i procesi u njemu *Metoda podrazumeva detaljnu analizu, što podrazumeva puno utrošenog vremena.*

Testeri moraju da analiziraju kod koji je neophodan kako bi se pronašao uzrok. (kada nešto ne radi)

Ispitivanje na osnovu strukture i staklene kutije su druga imena za ovaj metod.

Osnova za testiranje je izvorni kod test objekta, Pa se zato u literaturu se koristi termin testiranje zasnovano na strukturi ili kodu.

TESTIRANJE SOFTVERA- White-box

Strategija bele kutije je testiranje koje je veoma vremenski zahtevno i obično se primenjuje na manje delove sistema.
Korisno je za nalaženje grešaka u dizajnu, za nalaženje logičkih grešaka Primarni izvor za projektovanje testova je izvorni kod sa "akcentom" na tok kontrole i tok podataka.

TESTIRANJE SOFTVERA- *White-box*

U literature se može videti i sledeći termin "*glass box*" testiranje .

U slučaju tehnike **bele kutije**, unutrašnja struktura sistema za testiranje je poznata.

Tester ima pristup kodu softvera koji se testira i razume način njegove implementacije.

- Akcenat je na strukturnim elementima kao što su naredbe i petlje.

Testiranje zasnovano na principu bele kutije se uobičajeno koristi za verifikaciju softvera.

Kriterijum uobičajene „bele kutije" je da izvrši svaki izvršni iskaz tokom testiranja i da svaki ishod tokom testiranja bude zapisan u dnevnik testiranja.

TESTIRANJE SOFTVERA- *White-box*

Kod ovog testiranja se celokupan izvorni kod uzima u obzir tokom testiranja, što olakšava uočavanje greške, čak i kada su softverske pojedinosti nejasne ili nekompletne.

U slučaju testiranja metodom crne kutije, nije moguće upoznati se sa korišćenom logikom, pa se za izradu testnih slučajeva moraju koristiti specifikacije ili intuicija.

Posmatranjem koda, tokom testiranja metodom bele kutije, koristi se struktura koda, na osnovu koje se definišu vrste potrebnih testova. U mnogim slučajevima, preporučeni testovi ispituju putanje u kodu i prolazak kroz određene strukture koda

TESTIRANJE SOFTVERA- *White-box*

Često se tehnike testiranja metodom bele kutije povezuju sa metrikama pokrivanja, kojima se meri procenat izabranih putanja za testiranje, koje se pr overavaju u pojedinačnim testnim slučajevima.

Strukturno testiranje, poznato kao metoda bele ili staklene kutije, je tehnika testiranja gde je testerima poznata implementacija softvera.
Za razliku od modela crne kutije koji je fokusiran na to šta softver radi, model bele kutije je fokusiran na to kako softver radi

TESTIRANJE SOFTVERA- *White-box*

Kod white box testiranja postoje više alternativnih pristupa planiranja testnih slučajeva za testiranje podataka i korektnosti programa.

- Testiranje naredbi
- Testiranje grana
- Testiranje uslova
  - jednostavno testiranje uslova
  - višestruko testiranje uslova
  - minimalno višestruko testiranje uslova
- Testiranje putanja

Testiranje naredbi

Fokus je dat samo na pokrivenost svih naredbi. Ovde postoji problem jer ne pokriva sve ili dovoljno moguće tokove programa.

- Ono ne pokriva izvršavanje grana koje nemaju naredbe

- Čak i za ovakve testove je veoma teško obezbediti veliku (100) pokrivenost
  - Pojedine izuzetke u kodu je teško izazvati

Testiranje grana

Važna praktična tvrdnja je da je lakše testirati stanje i razgranatost nego putanje.

Testiranjem novih stanja ili grana uvek se može poboljšati sam proces strukturnog testiranja, Prema tome, prvo se treba koncentrisati na netestirane grane izlaza, a zatim se premešta na netestirane putanje.

Konačno, možda i najznačajnije, automatski alat obezbeđuje tačnu verifikaciju koji bi trebao da zadovolji zadate uslove testiranja programa.

Manualno poreklo testiranja podataka je proces sklon greškama, u kojem nameravane putanje nisu često izvođene za set datih podataka

Testiranje grana

Pronalazi i dizajnira minimalni broj testnih slučajeva tako da se izvrše svi mogući pravci kretanja (grane).

Tehnika je naprednija od testiranja naredbi

- Grane koje postoje u grafu toka programa se analiziraju
- Prati se i analiziraju se koje grane toka programa se izvršavaju (od uslovnih izraza)
- Pokrivenost se određuje u odnosu na procenat grana izvršenih u okviru testa

$$(broj\ izvršenih\ grana\ /\ ukupan\ broj\ grana)\ *100\%$$

Potrebno je pokriti i grane bez naredbi

- npr. if naredbu bez else dela

Ovaj koncept nije praktičan u mnogim slučajevima jer je potrebno puno resursa i testnih slučajeva za njegovo izvršavanje.

Testiranje uslova

Analiziraju se konkretni uslovni izrazi koji uzrokuju prelazak toka programa na određenu granu

Za razliku od testiranja grana koje se fokusira na proveru grana, testiranje uslova se bavi time šta dovodi do toga da program pređe u određenu granu

Uslov može biti složen

- ✓ više prostih uslova povezanih logičkim operatorima
- ✓ prost uslov je onaj koji
  - ✓ daje logičku vrednost kao rezultat i
  - ✓ ne sadrži logičke operatore
  - ✓ sadrži relacione operatore koji daju logički rezultat

TESTIRANJE SOFTVERA- *White-box*

U nekim situacijama, testerski tim ne može da napravi skup reprezentativnih slučajeva koji dokazuju pravilno funkcionisanje u svim situacijama. Za prevazilaženje ovog problema može se posmatrati predmet testiranja kao otvorenu kutiju (koja se ponekad naziva i ***belom kutijom (whitebox)***.

- Tada možemo koristiti unutrašnju strukturu predmeta testiranja u cilju sprovođenja različitih testova. Na primer, mogu se osmisliti testovi tako da se izvrše sve naredbe ili svi tokovi kontrole unutar komponente, kako bi bilo potvrđeno da predmet testiranja ispravno radi. Međutim, ta vrsta pristupa, koja se u literaturi naziva testiranje po metodi bele kutije, može da bude nekad i nepraktična.

TESTIRANJE SOFTVERA- *White-box*

Plan testiranja se određuje na osnovu elemenata implementacije softvera, kao što su programski jezik, logika i stilovi.

Testovi se izvode na osnovu strukture programa.

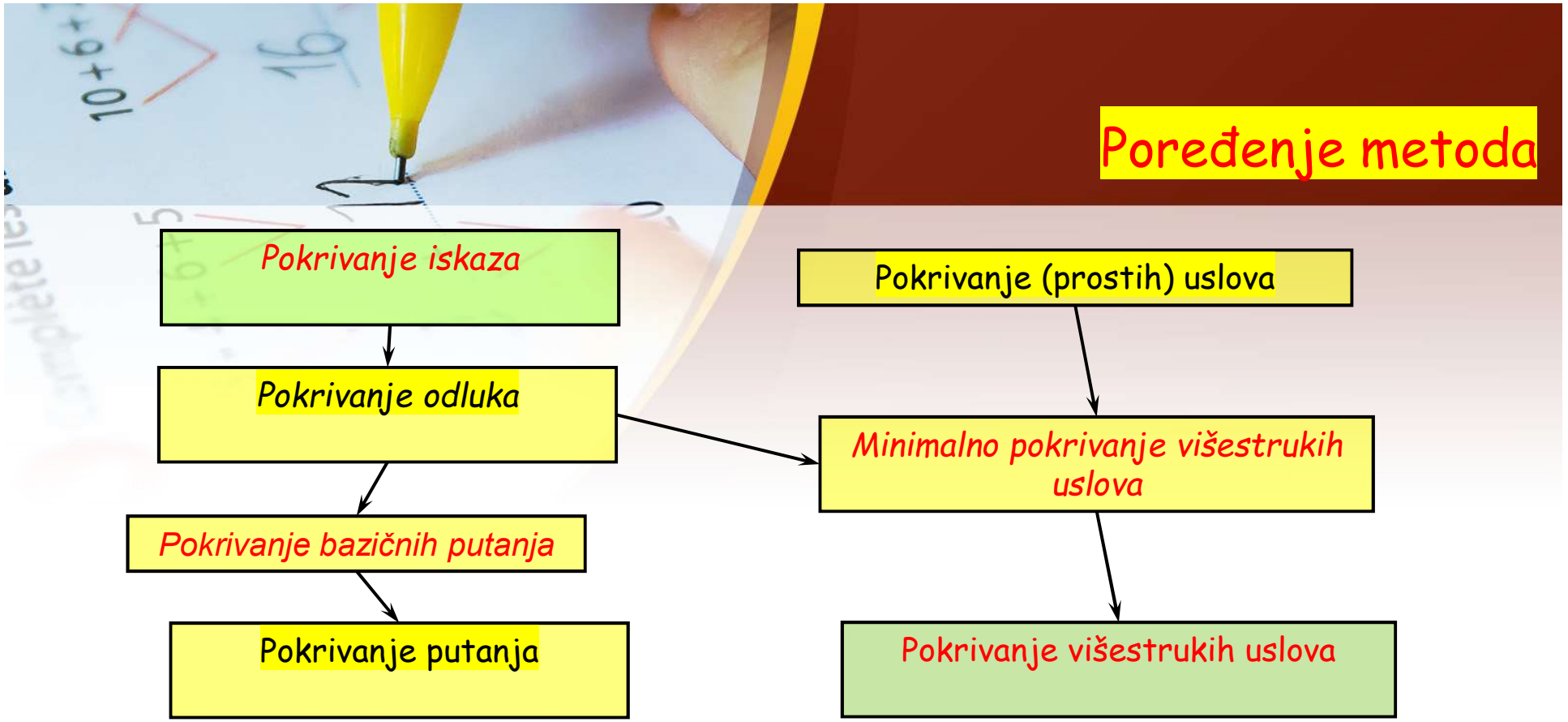
Specifičnim testovima može se proveravati postojanje beskonačnih petlji ili koda koji se nikada ne izvršava.

Pri dizajnu test slučajeva uzima se u obzir koji delovi koda će se izvršiti pri pokretanju test slučajeva

TESTIRANJE SOFTVERA- *White-box*

- Tehnike zasnovane na toku kontrole
- Pokrivanje iskaza
- Pokrivanje odluka
- Pokrivanje putanja
- *Pokrivanje bazičnih putanja*
- *Minimalno pokrivanje višestrukih uslova*
- Tehnike zasnovane na toku podataka

Poređenje metoda

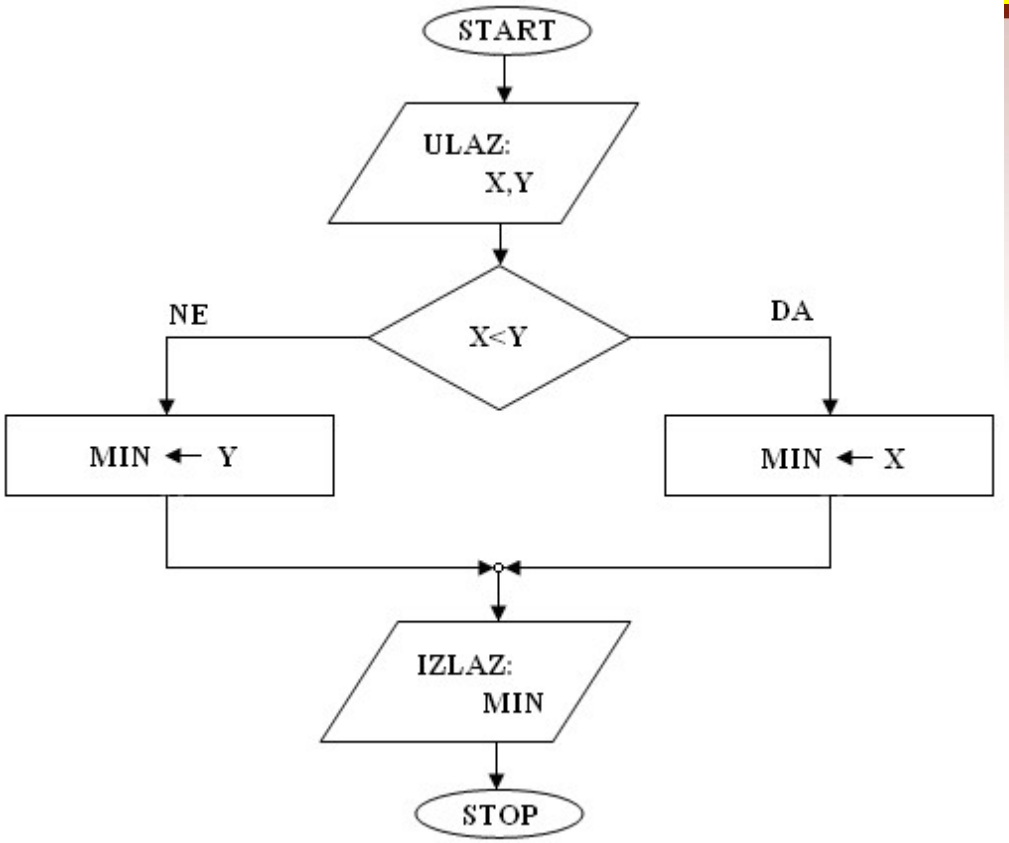


TESTIRANJE SOFTVERA

Koliko testnih slučajeva je potrebno da se postigne 100 % pokrivenost odluka (*decision coverage*).

Cilj pokrivanja uslova je proveriti pojedinačne rezultate za svaki logički uslov.

TESTIRANJE SOFTVERA



- Da li su svi uslovi ispunjeni ?

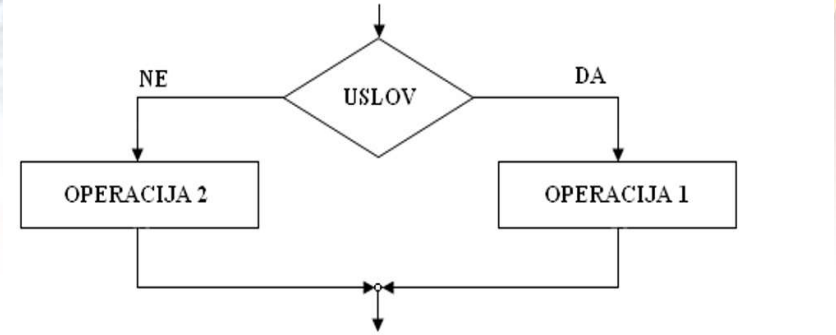
Na primer / šta se dešava kada je

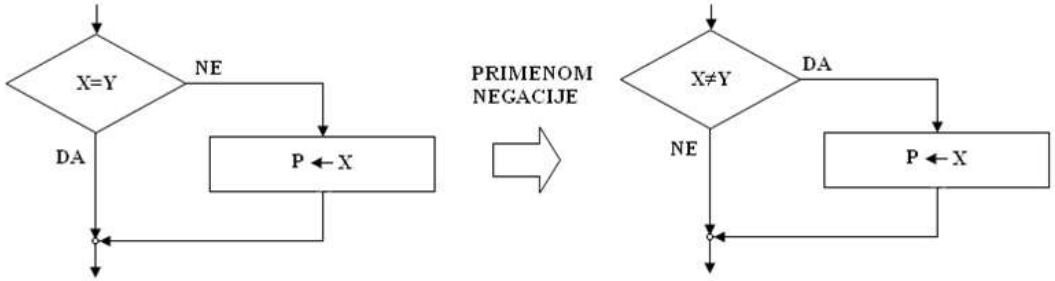
X=Y

Smisliti dobre - odgovarajuće poruke da budu [to jasniji korisnicima

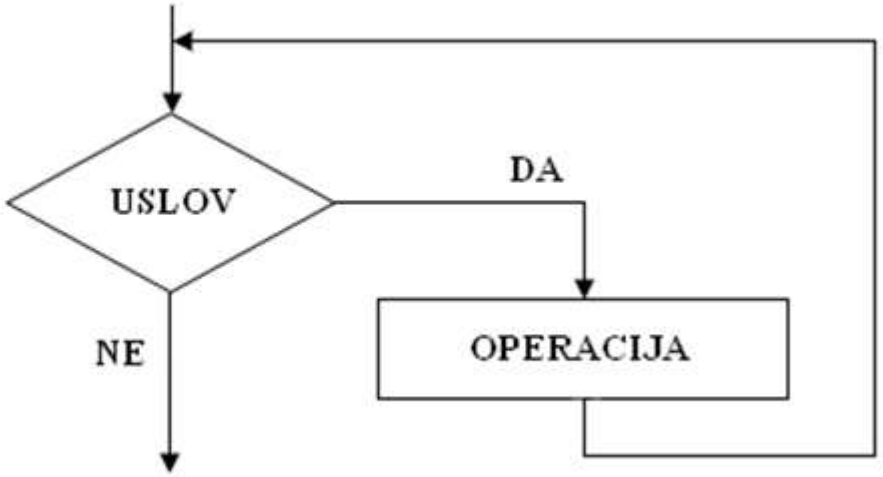
TESTIRANJE SOFTVERA- White-box

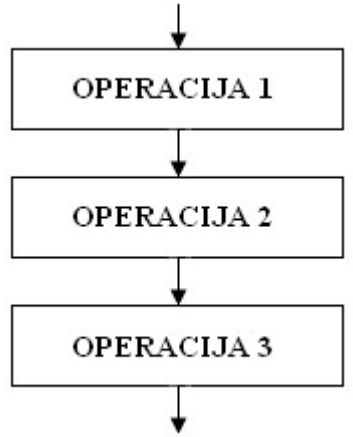
D

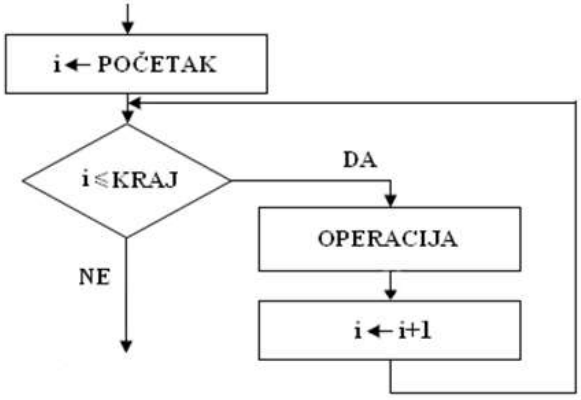




TESTIRANJE SOFTVERA-







TESTIRANJE SOFTVERA- White-box

Tehnike zasnovane na kontroli toka

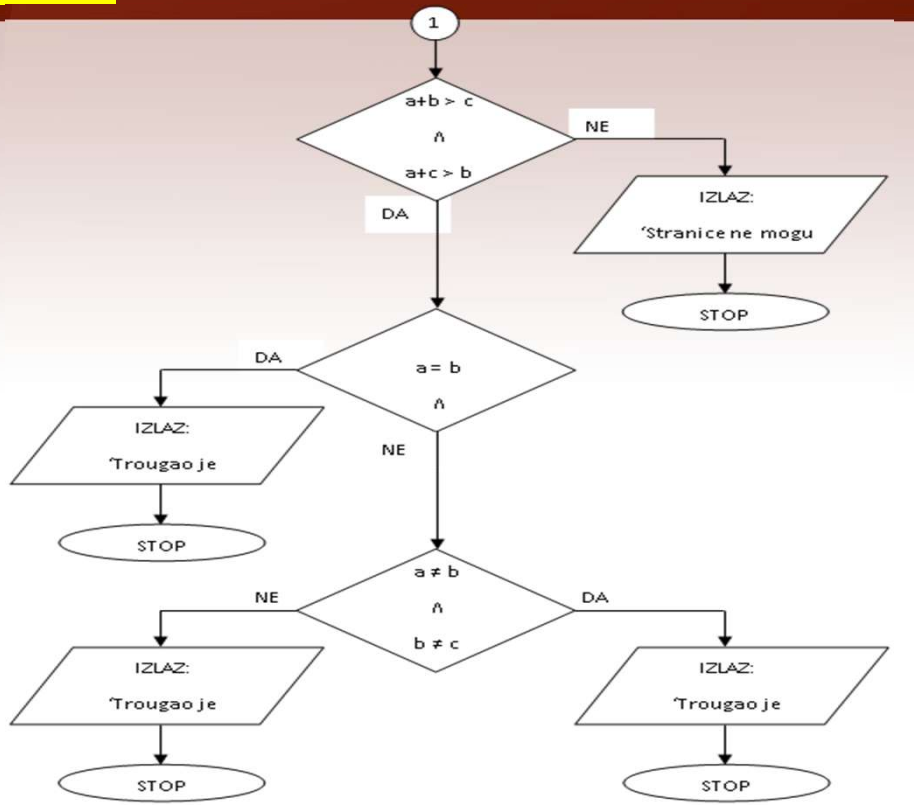
- Pokrivenost višestrukih uslova (Multiple Condition Coverage)
- Modifikovana pokrivenost odluka i uslova (MC/DC)
- Pokrivenost uslova (Condition Coverage)
- Pokrivenost odluka i uslova (Decision/Condition Coverage)
- Pokrivenost odluka ili pokrivenost grana (Decision or Branch Coverage)
- Minimalna pokrivenost višestrukih uslova (Minimal Multiple Condition Coverage)
- Pokrivenost iskaza (Statement Coverage) negde se zove i pokrivenost koda

TESTIRANJE SOFTVERA- *White-box*

Komponenta sa mnogo grananja i petlji sadrži veliki broj putanji koje treba proveriti.

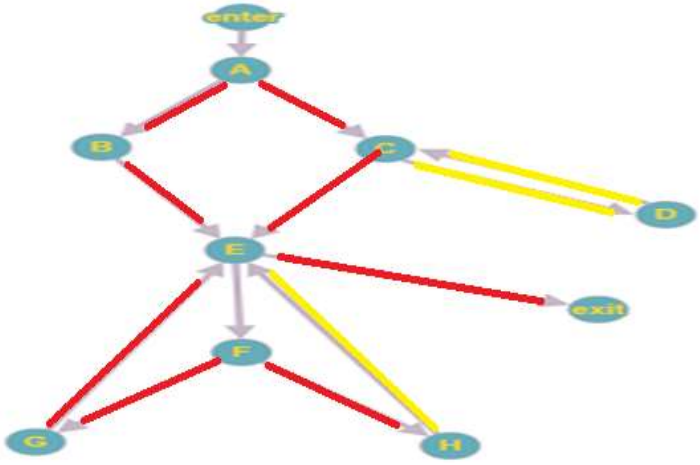
Teško je temeljno testirati komponentu sa velim brojem putanja.

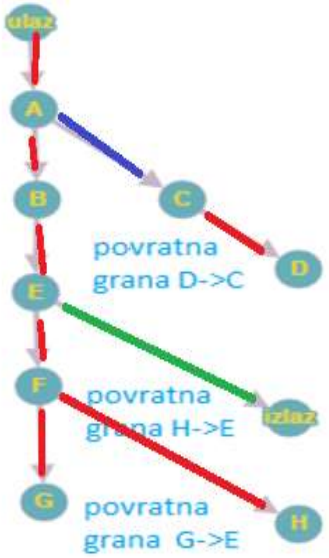
Preporučuje se za testiranje algoritama.



TESTIRANJE SOFTVERA- *White-box*

Dijagram toka (flow chart) i **graf programskog toka (flow graph) se koriste za** određivanja broja puteva i broja linija



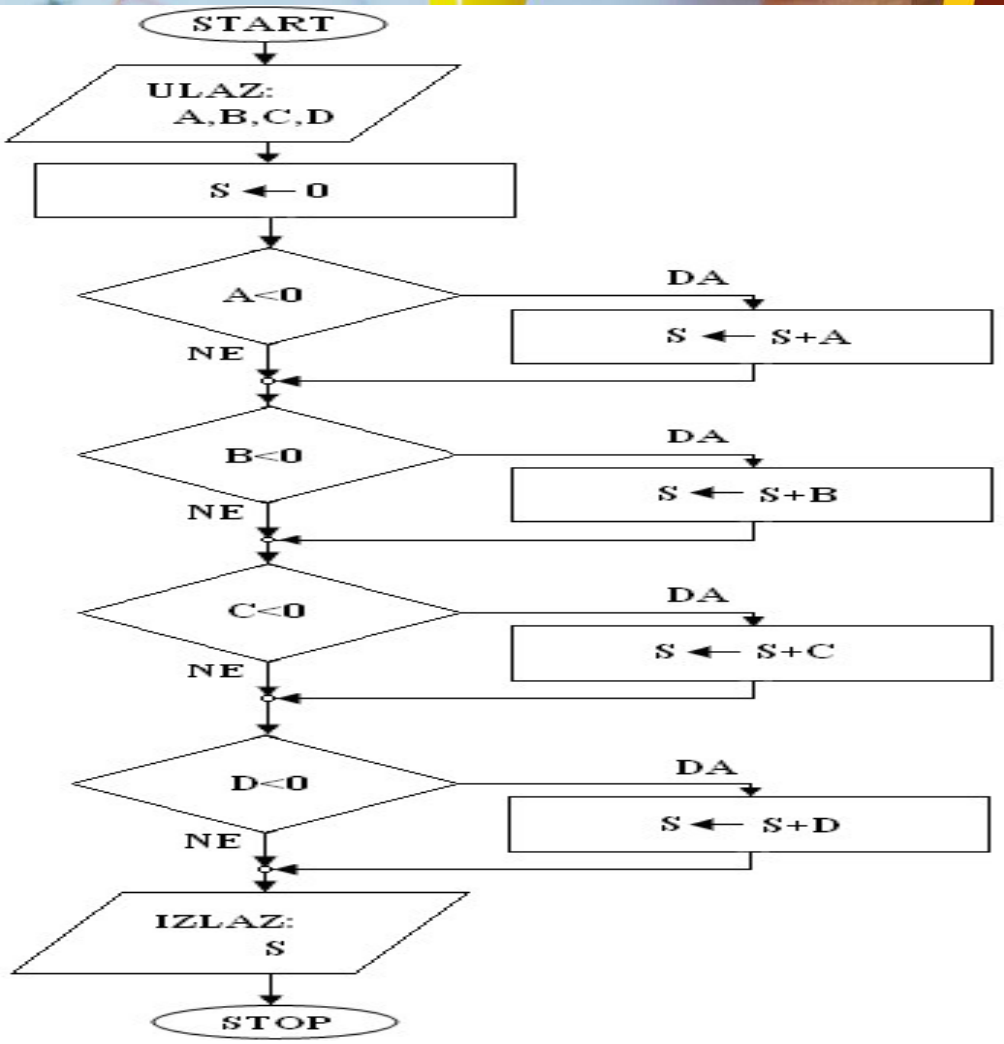


TESTIRANJE SOFTVERA- *White-box*

- ✓ Izvršavanje testova procesiranja podataka i korektnosti računanja,
- ✓ Testiranje kvaliteta softvera u odnosu na standarde,
- ✓ Testove održavanja (koliko je zahtevno održavanje) i
- ✓ Testove ponovne upotrebe koda (testira da li modul predložen za ponovno korištenje (reuse) odgovara standardima i procedurama za uključivanje u biblioteku komponenti).

Testovi procesiranje podataka i korektnosti računanja

- ✓ Zadatak testova korektnosti ("white box correctness test") je da se ispita svaka operaciju.
- ✓ Ispitivanje se vrši sa pripremljenim testnim slučajevima.
- ✓ Testni slučaj je skup akcija koji se izvršava da bi se izvršila verifikacija pojedinačne karakteristike ili funkcionalnosti aplikacije.
- ✓ Testni slučaj sadrži ulazne podatke i na osnovu tih podataka se računa izlaz za koji se proverava da li je on očekivan ili nije.
- **Odobrene test procedure** – Procesi testiranja se izvršavaju u skladu sa testnim planom i procedurama testiranja koje su odobrene



TESTIRANJE SOFTVERA

- Da li su svi uslovi ispunjeni ?

Unos slova ?

Na primer / šta se dešava kada je

A, B,C,D promeni znak (>)

Odgovarajuće poruke

Pokrivanje iskaza – Statement coverage

- ✓ Test primeri se tako projektuju da se svaki iskaz programa izvrši bar jednom.
- ✓ Ne možemo znati da li u nekom iskazu postoji greška ukoliko ga ne izvršimo

Ograničenja metoda pokrivanja iskaza je :

- ✓ Ukoliko se utvrdi da se iskaz ispravno izvršava za jednu ulaznu vrednost to znači da :
- ✓ Nema garancije da će se ispravno izvršavati za svaku drugu ulaznu vrednost

TESTIRANJE SOFTVERA- *White-box*

Poređenje različitih tehnika

- Pokrivanje uslova
- Ne garantuje pokrivanje svih odluka
- Samim tim nije garantovano ni pokrivanje svih iskaza

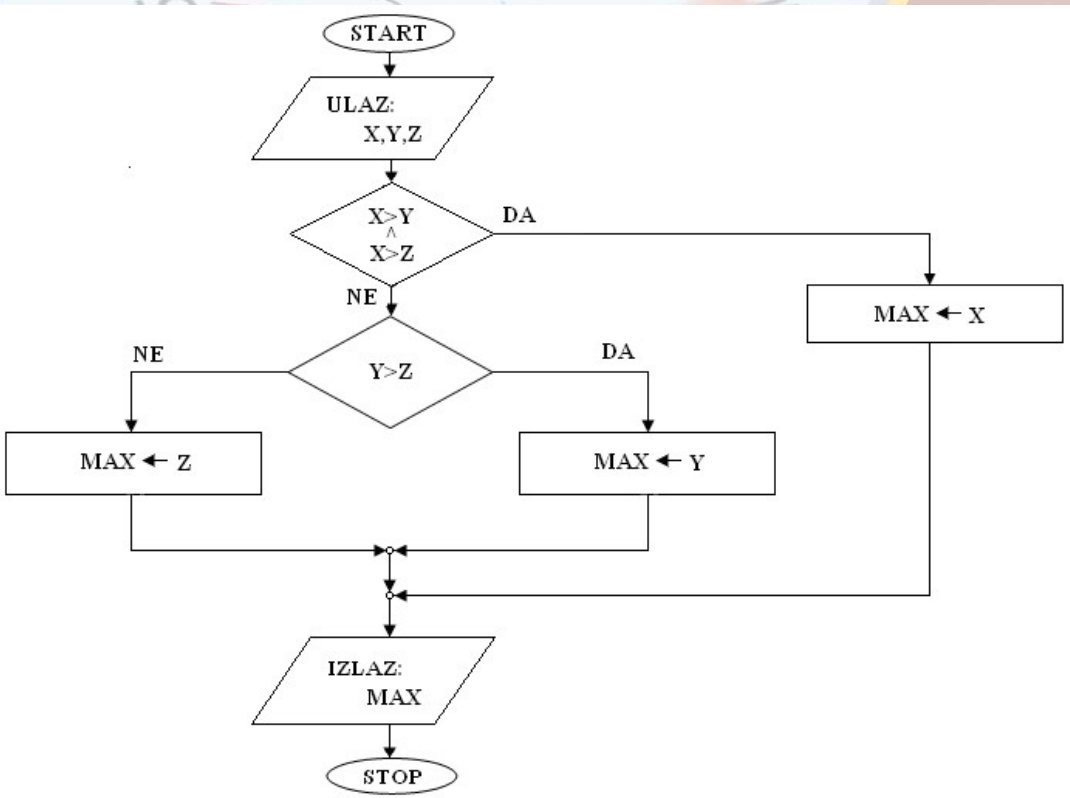
TESTIRANJE SOFTVERA

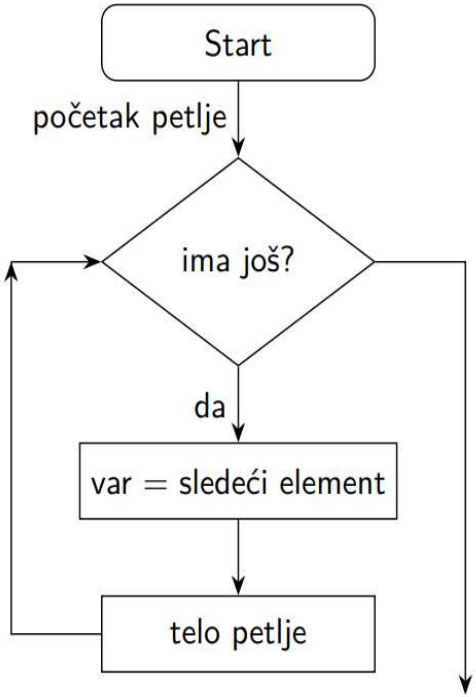
Pokrivenost testiranja (coverage) je mera do koje je neki softver ili određena softverska komponenta istestirana nekim skupom testova, u smislu procenta obuhvaćenih stavki.

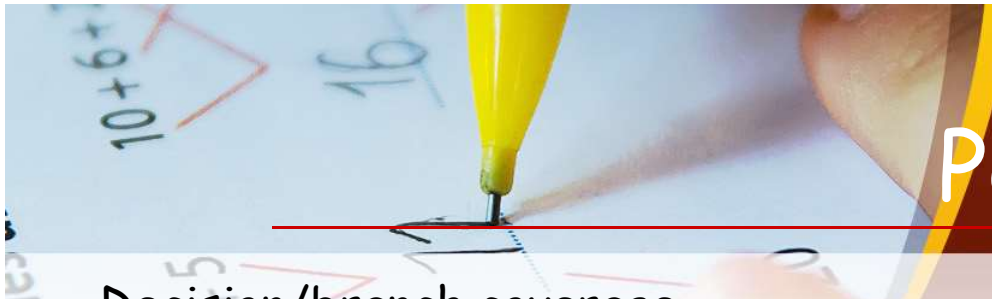
Da li su sve test stavke obuhvaćene testom ?

U slučaju da pokrivenost nije 100%, potrebno je proširiti skup testova s novim testovima. Pokrivenost se povećava tako da se sistemski pišu novi testovi da bi se pokrili svi delovi softvera (projekta), odnosno da se svi delovi softvera izvrše bar jednom

Pokrivanje iskaza – Statement coverage







Pokrivanje odluka

Decision/branch coverage

Prednosti pokrivenosti testiranja:

- Pomaže u kreiranju dodatnih testova kako bi se povećala pokrivenost;
- Pomaže u pronalaženju delova programa koji uopšte nisu pokriveni testovima;
- Određuje kvantitavni meru pokrivenosti koda, koja indirektno predstavlja meru kvaliteta softvera.

Mane pokrivenosti testiranja:

- Može da izmeri pokrivenost koda koji je napisan, ali se ne zna ništa o delu koji još uvek nije napisan;
- Nije efikasna u slučaju da specificirana funkcionalnost nije implementirana ili ukoliko je izostavljena iz specifikacije, pošto se posmatra samo kod.

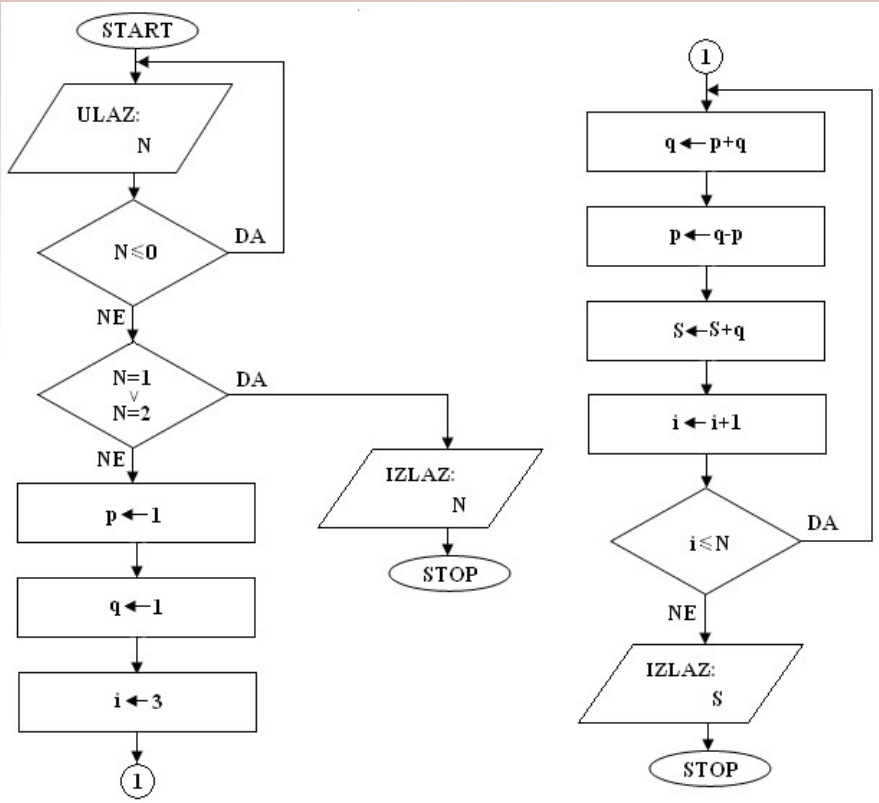
TESTIRANJE SOFTVERA

```
switch(k) {
                case 6:   cprintf("PETAK");
                          break;
                case 0:   cprintf("SUBOTA");
                          break;
                case 1:   cprintf("NEDELJA");
                          break;
                case 2:   cprintf("PONEDELJAK");
                          break;
                case 3:   cprintf("UTORAK");
                          break;
                case 4:   cprintf("SREDA");
                          break;
                case 5:   cprintf("CETVRTAK");
        }
```

Zbir prvih **N** članova Fibonačijevog niza.

Fibonačijev niz je:

1, 1, 2, 3, 5, 8, 13, 21, ...



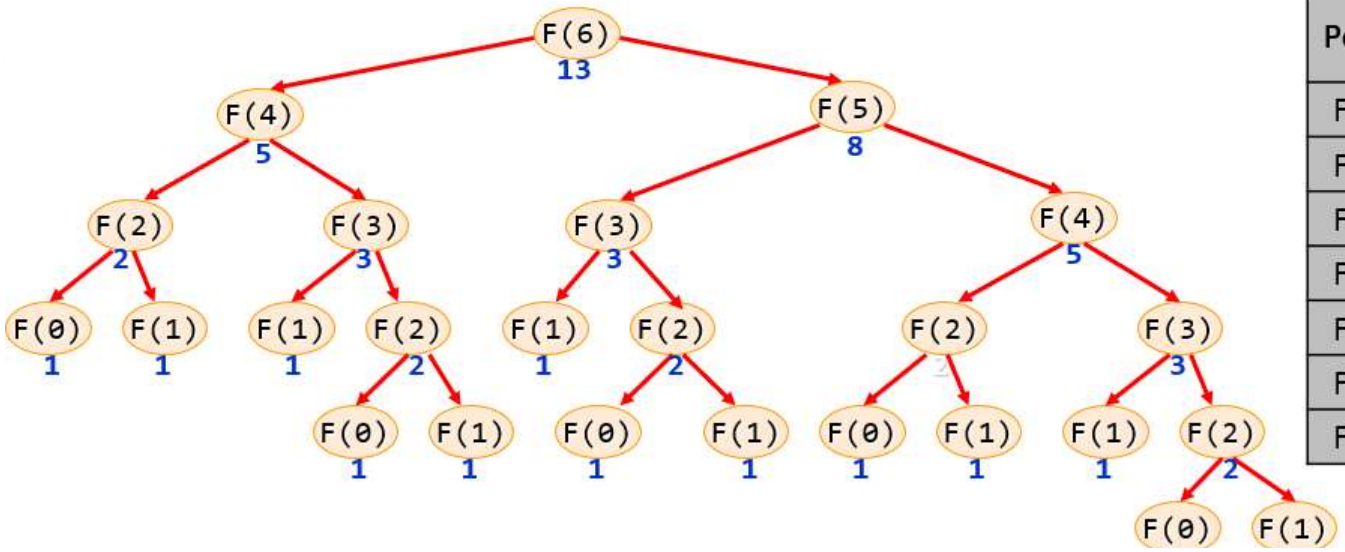
FIBONAČIJEV BROJ

1, 1, 2, 3, 5, 8, 13, 21, 34,...

Fibonačijevi brojevi predstavljaju niz 1,1,2,3,5,8,...
sekvenca počinje sa 1,1
sledeći broj se računa kao zbir prethodna dva

$F_0=F_1=1$

$F_i=F_{i-1}+F_{i-2};\ i>1$



Poziv	Broj izvršavanja
F(6)	1
F(5)	1
F(4)	2
F(3)	3
F(2)	5
F(1)	8
F(0)	5

FIBONAČIJEV BROJ

File Edit Format Run Options Window Help

```
# KOMENTARI
# 1 1 2 3 5 8 13 21

print(" FIBONACIJEV NIZ / FIBONACI")
# DEFINISEMO FUNKCIJU
def fibonacijevbroj(n):
    if n == 1:
          return 1

    elif n == 2: #
          return 1

    else: # Rekurzivni pozivi
         return fibonacijevbroj(n-1) + fibonacijevbroj(n-2)
n = int (input ("Unesite poziciju broja za Fibonačijev broj: "))
# Izlaz prikaz Fibonačijevog broja
print ("Fibonačijev broj za indeks", n, "je", fibonacijevbroj(n))
```

File Edit Shell Debug Options Window Help

```
Python 3.10.6 (tags/v3.10.6:9c7b4bd, Aug  1 2022, 21:
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" fo
>>>
========== RESTART: C:/Users/markoni/Desktop/PYTHO
 FIBONACIJEV NIZ / FIBONACI
Unesite poziciju broja za Fibonačijev broj: 7
Fibonačijev broj za indeks 7 je 13
>>>
```

TESTER:

Da li postoje DOBRONAMERNI KORISNIK DA ILI NE ?

Mane:

U kodu nema dovoljan broj komentara

Unos negativninog broja

Unos slova

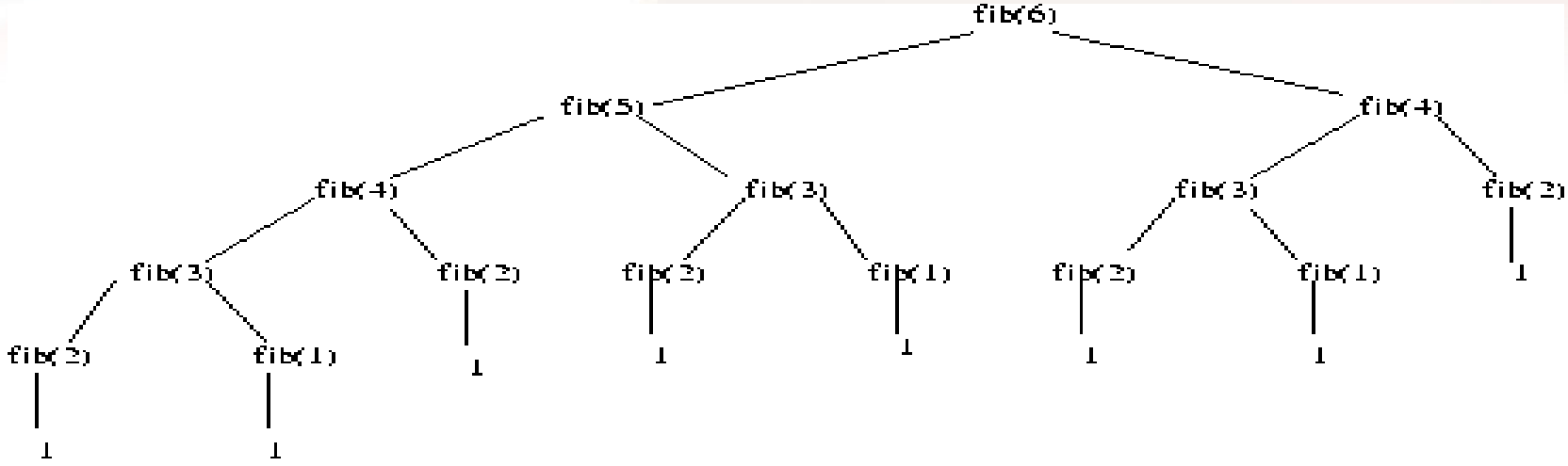
Fibonacci rekurzivno

:

- ✓rekurzija se zasniva na manjim vrednostima
- ✓postoji osnovni slučaj koji se ne oslanja na rekurziju

Da li će rekurzivna varijanta raditi efikasno?...

Rekurzivno rešenje je neefikasno jer obavlja puno ponovljenih izračunavanja!



Pokrivanje uslova

- I na prstom primeru ili na najedntavni primer može da se desi da i on bude težak za testere

PRIMER> Napisati deo C programa koji utvrđuje najveću vrednost između tri uneta broja **x**, **y** i **z** i rezultat upisuje u promenljivu **max**.

1 Način

```
if(x>y)
{
                    if(x>z)          max=x;
                      else max = z;
}else{
                    if(y>z) max = y;
                      else max = z;
}
```

2 Način

```
if(x>y&&x>z) max = x;
else {
                                  if(y>z) max = y;
                                    else max = z;
}
```

3 Način

```
max = x;
if(y>max) max = y;
        if(z>max) max = z;
```

- **TESTERI:**
- **Koji je najefikasniji način**
- **Da li testiramo sve načine ?**
- **RAD PO MODULIMA**
- **Problem kad svako u timu piše delove koda koje nisu koordinisane sa različito sa PROJEKT MENADŽER ILI PRODUKT MENADŽER**
- **OPTIMIZACIJA KODA**

Data flow coverage

- Za dizajn testnih slučajeva biraju se putevi u skladu sa lokacijama definicije i korištenja promenljivih.
- Ova tehnika nije praktična za intezivno korištenje.
- Pogodna je za module sa ugnježdenim if iskazima i petljama.

TESTIRANJE SOFTVERA

Matematički bazirano strukturno testiranje ima mnogo prednosti.
Neke od prednosti su te što svi osnovni setovi putanja pokrivaju sve grane grafa toka programa i one zadovoljavaju kriterijume strukturnog testiranja, a samim time automatski zadovoljavaju slabije grane i osnovne kriterijume.
Sledeće kod strukturnog testiranja je to, da je testiranje proporcionalno kompleksnosti, odnosno minimalni broj testiranja potreban da zadovolji kriterijum strukturnog testiranja je zapravo ciklična kompleksnost.
Pored toga, pošto je minimum potrebnih brojeva testova poznat unapred, strukturno testiranje podžava planiranje testiranja i traži konstantan nadzor procesa testiranja

TESTIRANJE SOFTVERA

Druga prednost strukturnog testiranja je ta, da precizne matematičke interpretacije „nezavisnog" kao „linearno nezavisnog" strukturnog testiranja garantuju nezavisno testiranje izlaza. Na osnovu ovoga se može videti da, za razliku od drugih uobičajenih strategija testiranja, strukturno testiranje ne dozvoljava unošenje proizvoljnih ulaznih podataka za testiranje.

Prikazan je jedan C program:

```
Prazna funkcija()
{
   if (uslov1)
         a = a + 1;
   if (uslov2)
         a = a - 1;
}
```

DA LI JE OVO PRIMER DELA DOBROG PROGRAMA ?

TESTIRANJE SOFTVERA

U ovom C primeru pretpostavlja se da je vrednost promenljive „a" neizmenjena pod bilo kojim uslovima (samim tim ta prepostavka je očigledno netačna).
Obzirom da u ovom C programu postoje dva uslova, pri čemu se moraju posmatrati oba ishoda, proizilazi da postoje četiri moguća rešenja.
Kriterijum testiranja može biti zadovoljen sa dva testa koji nisu uspeli da otkriju grešku.
Prvo neka obe odluke izlaza budu NETAČAN, u kojem slučaju vrednost promenljive „a" nije promenjena.
Zatim se postavljaju obe odluke izlaza da budu TAČAN, u kojem slučaju je vrednost promenljive „a" prvo povećana, a zatim smanjena na početnu vrednost.
Sa druge strane, strukturno testiranje garantovano uočava grešku. Potrebne su tri nezavisne test putanje, tako da makar jedan (od dva ishoda) bude TAČAN, a drugi NETAČAN, ostavljajući vrednost promenljive rastućom ili opadajućom, prema čemu se detektuje greška.

TESTIRANJE SOFTVERA

Mnogi autori pokazuju korelaciju između kompleksnosti i grešaka, jednako kao i odnos između kompleksnosti i teškoće da se razume kako je nastala potencijalna greška.
Pouzdanost je kombinacija testiranja i razumevanja.

" Softver je toliko dobar da je test koji je izvršen adekvatan da predvidi željeni nivo poverenja. "McMillan

Pošto kompleksnost čini softver težim za testiranje i težim za razumevanje, kompleksnost je blisko vezana sa pouzdanošću.

Sa jedne tačke gledišta, kompleksnost meri napore potrebne za postizanje datog nivoa pouzdanosti.

- Poziv

```
File  Edit  Format  Run  Options  Window  Help
def calc():
    while True:
            print ("PROGRAM KALKULATOR!")
            print ("Izaberite stavku menija:")
            print (" ")
            print ("1) SABIRANJE")
            print ("2) ODUZIMANJE")
            print ("3) MNOŽENJE")
            print ("4) DELJENJE")
            print ("5) IZLAZ")
            print (" ")
            izbor = int (input("Unesite jednu od  opciju:"))
            if izbor == 1:
                n1= int (input ("Unesi prvi broj A:"))
                n2= int (input ("Unesi drugi broj B:"))
                rez = n1+n2
                print (" Rezultat /SABIRANJE JE A +B:\n ", rez )
            elif izbor == 2:
                n1= int (input ("Unesi prvi broj A:"))
                n2= int (input ("Unesi drugi broj B:"))
                rez = n1-n2
                print (" Rezultat /Oduzimanje A - B \n", rez )
            elif izbor == 3:
                n1= int (input ("Unesi prvi broj A:"))
                n2= int (input ("Unesi drugi broj B:"))
                rez = n1 * n2
                print (" Rezultat /MNOZENJE A * B \n", rez )
            elif izbor == 4:
                n1= int (input ("Unesi prvi broj A:"))
                n2= int (input ("Unesi drugi broj B:"))
                if n2 == 0:
                   print("Drugi broj ne moze biti 0")
                else :
                   rez = n1 / n2
                   print (" Rezultat / DELJENJE A / B \n", rez )
            elif izbor ==5:
                print (" KORISTIMO KALKULATOR " )
                break
            ch= input("IZABERI K da izadjes iz programa")
            if ch in "Kk":
                print (" KORISTIMO KALKULATOR " )
                break
```

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.6 (tags/v3.10.6:9c7b4bd, Aug
AMD64)] on win32
Type "help", "copyright", "credits" or "
>>>
======= RESTART: C:/Users/markoni/Desktc
PROGRAM KALKULATOR!
Izaberite stavku menija:

1) SABIRANJE
2) ODUZIMANJE
3) MNOŽENJE
4) DELJENJE
5) IZLAZ

Unesite jednu od  opciju:4
Unesi prvi broj A:5
Unesi drugi broj B:0
Drugi broj ne moze biti 0
IZABERI K da izadjes iz programa
```

Graf toka kontrole

Predstavlja reprezentaciju programa u obliku grafa i implicitno pokazuje putanje izvršavanja.

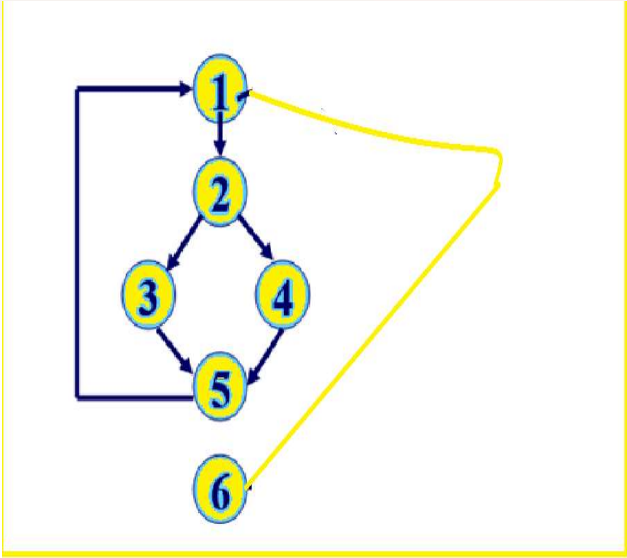
Čvorovi grafa su instrukcije.

Graf toka kontrole programa opisuje redosled u kome se izvršavaju instrukcije programa

Graf kontrole toka je grafovska reprezentacija svih puteva kojima se moze proći kroz program tokom njegovog izvršavanja.

Ovaj graf se pravi odvojeno za svaku funkciju.

- Svaki ˇcvor grafa kontrole toka predstavlja
osnovni blok naredbi koje ne sadrži nikakvo grananje, pa se stoga izvršavaju sekvencijano.



Graf toka kontrole

Treba zadati neki kriterijum po kome se vrši testiranje:

Sve putanje u Grafu toka kontrole treba dobro rešiti pre svega serijom testova