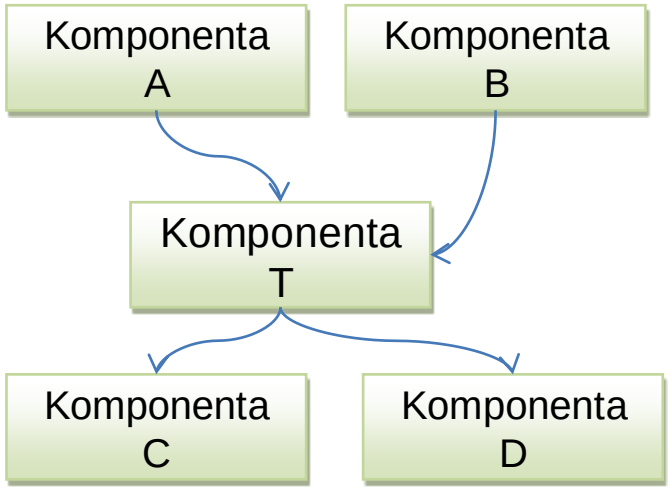
# Realizacija jediničnog testiranja

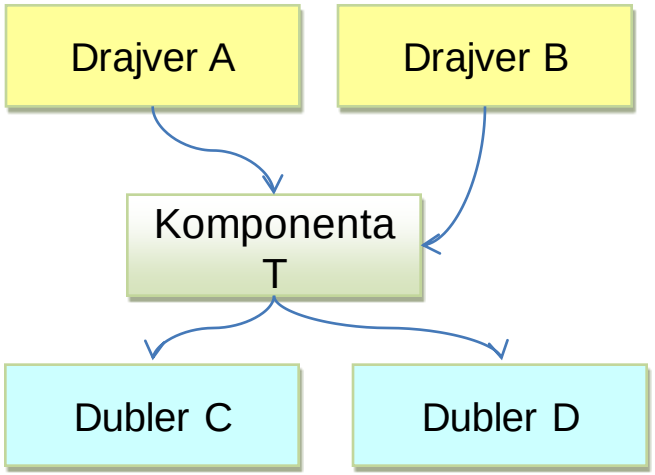
# Uvod

- Jedinično testiranje predstavlja testiranje izolovanih celina (komponenti) u sistemu.
- Uslov je da se komponenta koja se testira može posmatrati kao nezavisna celina koja se može izvući iz konteksta sistema i testirati izolovano od ostalih komponenti.
- Komponentama se mogu smatrati klase, moduli, paketi, čak i fragmenti koda
- Komponente mogu biti ili deo korisničkog interfejsa (strane ili forme) ili komponente koje vrše neke akcije i procesiraju rezultate (na primer komponente za implementaciju algoritama, dohvatanje podataka, komunikaciju sa ostalim sistemima).

# Okruženje za testiranje pojedinačne komponente

Komponenta koja se testira se izvlači iz konteksta sistema u kome komunicira sa ostalim komponentama i ubacuje se u test kontekst koji je simulacija pravog sistema gde se nalaze komponente koje simuliraju rad stvarnih komponenti. Primer je prikazan na slici – komponenta koja se testira T u realnom okruženju biva pozvana od strane komponenti A i B kojima pri pozivu vraća neke podatke i poziva komponente C i D koje joj daju informacije kada ih komponenta pozove.



# Okruženje za testiranje...

- Umesto sa realnim komponentama iz svog okruženja, komponenta u test kontekstu komunicira sa simuliranim komponentama koje je ili pozivaju ili joj odgovaraju na pozive na isti način kao i realne komponente. U zavisnosti od toga da li simulirane komponente pozivaju ili bivaju pozvane one se dele na dve vrste:
- **Drajveri** – komponente koje simuliraju rad realnih komponenti koje pozivaju druge komponente i očekuju neki odgovor. Ove komponente pokreću test pošto iniciraju pozive ka komponenti koja se testira.
- **Dubleri** – komponente koje simuliraju rad realnih komponenti koje primaju pozive i vraćaju iste rezultate kao i realne komponente.

# Dubleri

- Test dubleri mogu biti različitih vrsta i kompleksnosti:

  **Dummy** – prazan dubler je najjednostavniji oblik test dublera. Omogućava povezivanje programa pružajući podrazumevanu (konstantnu) povratnu vrednost gde je potrebno.

  **Stub** – potporni dubler dodaje pojednostavljenu logiku u odnosu na dummy, pružajući različite rezultate.

  **Spy** - špijun snima i čini dostupnim parametre i informacije o internom stanju komponente, što omogućava naprednije validacije.

  **Mock** – lažna komponenta definiše se od strane pojedinačnog testa da uradi validaciju specifičnog ponašanja za taj test, proverajući vrednosti parametara i redosled pozivanja.

  **Simulator** - simulator je sveobuhvatna komponenta koja obezbeđuje verniju aproksimaciju ciljne funkcionalnosti (za koju pišemo dublera).Simulator obično zahteva značajne dodatne napore za razvoj.

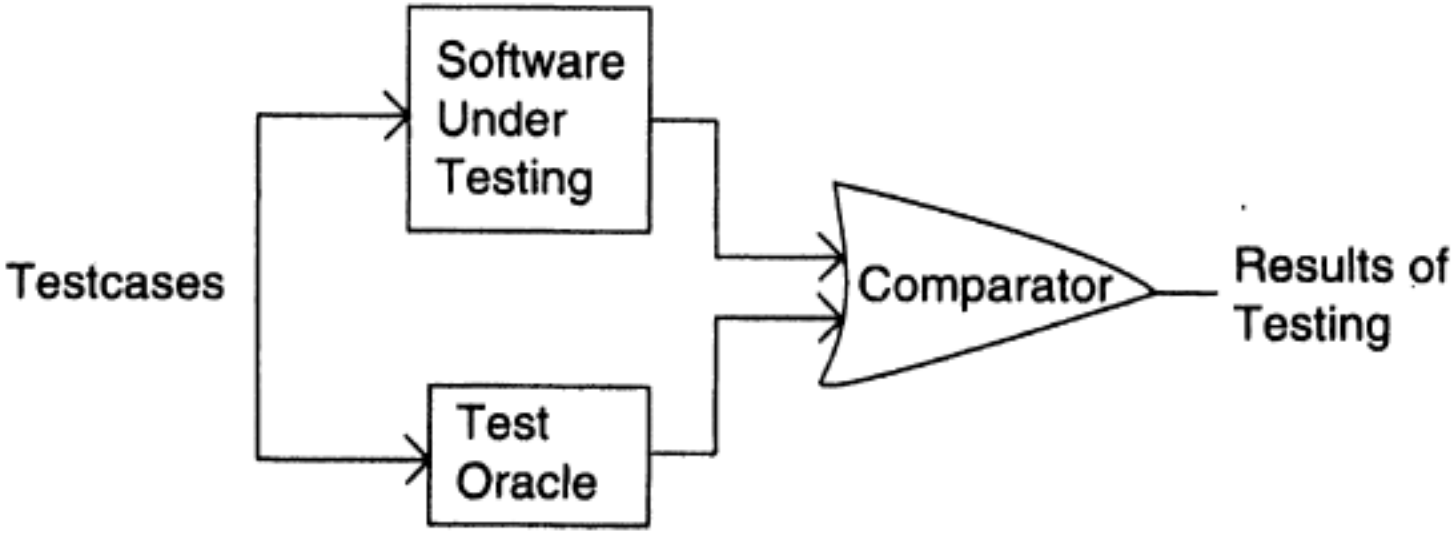
# Realizacija drajvera i dublera

- Drajveri i dubleri se često programiraju, sem u slučaju da drajver nije tester koji ručno poziva komponente grafičkog interfejsa.
- Za implementaciju drajvera se, u zavisnosti od programskog jezika na kojem je napisana komponenta, može koristiti JUnit(programski jezik Java), NUnit(programski jezik C#), DBUnit (SQL skriptovi), i tako dalje.
- Dubleri se implementiraju tako što se definišu komponente koje će simulirati stvarne komponente, i u njima se definiše šta se vraća za određene sekvence ulaznih podataka. Postoje razne biblioteke koje olakšavaju proces definisanja ovakvih komponenti kao što su JMock, NMock, Rhino Mock i slično.

# Realizacija drajvera i dublera

- I ***Drajveri*** i ***dubleri*** predstavljaju dodatni “trošak”. To je zbog toga što oni takođe predstavljaju softver koji treba napisati (prave se uobičajenim metodama), ali se ne isporučuju sa finalnom verzijom programa (koji se testira pomoću njih). Ukoliko su ***drajveri*** i ***dubleri*** jednostavni i sam dodatni trošak je mali. Nažalost, mnoge komponente ne mogu biti adekvatno pojedinačno testirane sa jednostavnim (jeftinim) softverom. U tom slučaju, kompletan proces testiranja se produžava (odlaže), dok se ne završi korak integracionog testiranja (u kome se koriste ***drajveri*** i ***dubleri***).
- Testiranje modula je pojednostavljeno kada je programska komponenta kompaktno dizajnirana. Kada samo jedna funkcija “komunicira” sa komponentom, broj test primera je smanjen i greške se mogu lakše predvideti i otkriti.

# Očekivani rezultati testa




- Predviđanje, predikcija rezultata testa (test oracle)
  - Prediktor testa je mehanizam, nezavisan od samog programa, koji se može upotrebiti za proveru ispravnosti rada programa za test primere.
  - Konceptualno, test primeri se zadaju programu i prediktoru i njihovi izlazi potom međusobno upoređuju

# Pristupi problemu predikcije rezultata

- Neposredna verifikacija izlaza programa
- Redundantna izračunavanja
- Provere konzistencije
- Redundantni podaci

# Neposredna verifikacija izlaza programa

- Ako program dolazi sa jasnom specifikacijom koji izlaz se očekuje za koji ulaz, ispravnost izlaza testa može da proveri čovek (kvalitetno, ali skupo rešenje).
- Alternativno, može se isprogramirati automatski neposredni verifikator izlaza

# Neposredna verifikacija izlaza programa

- Primer: verifikator programa za sortiranje ulaznog niza (preudokod):

```
Input: Structure S
    Make copy T of S
    Sort S
     // Verify S is a permutation of T
     Check S and T are of the same size
     For each object in S
          Check if object appears in S and T same number of times
     // Verify S is ordered
     For each index i but last in S
          Check if S[i] <= S[i+1]
```

- Verifikator nije prosto druga implementacija iste funkcije (sortiranja). U opštem slučaju, pisanje verifikatora može biti zahtevan posao.

# Redundantna izračunavanja

- Neposredna verifikacija nije uvek moguća: npr. šta ako program računa npr. sin(x)? Čoveku nije lako da ručno verifikuje rezultat u opštem slučaju.
- U pristupu R.I. za verifikaciju se upotrebljava druga implementacija sin(x), po mogućnosti neka kojoj verujemo da je ispravna tj. “zlatni standard”.

# Redundanta izračunavanja

- Problem: ako za verifikaciju programa P koristimo program S, šta se dešava ako i S sadrži defekte (zašto jednoj implementaciji verovati više nego drugoj)?
- Neka za test t P daje rezultat P(t), a S daje S(t)
- Ako je $P(t) \neq S(t)$, jedna ili obe implementacije sadrže grešku: problem će biti uočen i najverovatnije P (ili S ili oba) biti ispravljeni.
- Ako je P(t) = S(t), a i P i S imaju identične defekte, problem tada neće biti ni uočen
  - Rešenje sa nezavisnim razvojem P i S nije uvek efektivno, pošto je nekad teško zaista obezbediti nezavistan razvoj, a drugo, eksperimenti potvrđuju da se identični defekti znatno češće javljaju nego što bi dozvoljavala pretpostavka nezavisnog razvoja. Problem je što je za neke vrednosti podataka teže napraviti korektnu implementaciju i upravo za te vrednosti javljaju se isti defekti u različitim implementacijama
  - Često je S prethodno izdanje softvera P, u tehnici regresivnog testiranja
  - Bolje rešenje je upotreba drugog algoritma za rešavanje istog problema, npr. binarno traženje treba verifikovati npr. insertion sort algoritmom.

# Redundantni podaci

- U određivanju korektnosti izlaza programa za dati ulaz, može se iskoristiti ponašanje programa za druge ulaze
- Na primer, za OO kolekcije, dodavanje elementa kolekciji i potom njegovo uklanjanje iz kolekcije ima dobro definisan efekat na kolekciju:
  - za neke kolekcije (npr. bag - vreća) zbirni efekat je da nema promene
  - Za druge kolekcije (npr. skup) može ili ne mora biti promene u jednom elementu, zavisno da li je taj element već bio u kolekciji pre testa.
- Drugi primer, koji koristi isti kod ali za drugačije vrednosti pri testu je za sin(x) da se iskoristi identitet: sin(a+b)=sin(a)cos(b)+cos(a)sin(b).
  - Ponavljati test za proizvoljne vrednosti a,b

# Redundanti podaci

- Kod softvera kod koga postoji neki pojam kontinuiteta u ulaznim vrednostima, za “bliske” ulazne vrednosti, izlaz će se često menjati kontinualno.
- Npr. avionski sistem za davanje saveta pilotu komercijalnog aviona za izbegavanje kolizije sa drugim avionima, daje savete o visini (ne menjaj, penji se, spusti se) na osnovu raznih faktora: položaja drugih aviona, visine od zemlje itd.
- Ako se samo malo promene položaji drugih aviona, očekuje se da se savet neće promeniti u odnosu na prethodni: ako bi savetovanje bilo nestabilno za bliske vrednosti, pilot ne bi polagao mnogo poverenja u njega i to bi bio defekat sistema.

# Provere konzistencije

- Sasvim je uobičajeno da programeri zahtevaju izvesna ograničenja u korišćenju struktura podataka. Na primer, da dati kontejner nikada ne drži duplirane objekte. Provera ovih "invarijanti" je izuzetno efikasan način za pronalaženje kvarova.
- Programeri obučeni u razvoju softvera “po ugovoru” (design by contract) mogu da proizvedu kod za takve provere u toku normalnog razvoja. Za objektno-orijentisani softver, takve provera su obično organizovane oko invarijanti objekta kao i preduslova metoda objekata i post-uslova.
- Takve provere (assert checks) mogu se uključiti u toku testiranja i potom isključiti, ako je neophodno zbog performansi, tokom normalnog funkcionisanja.

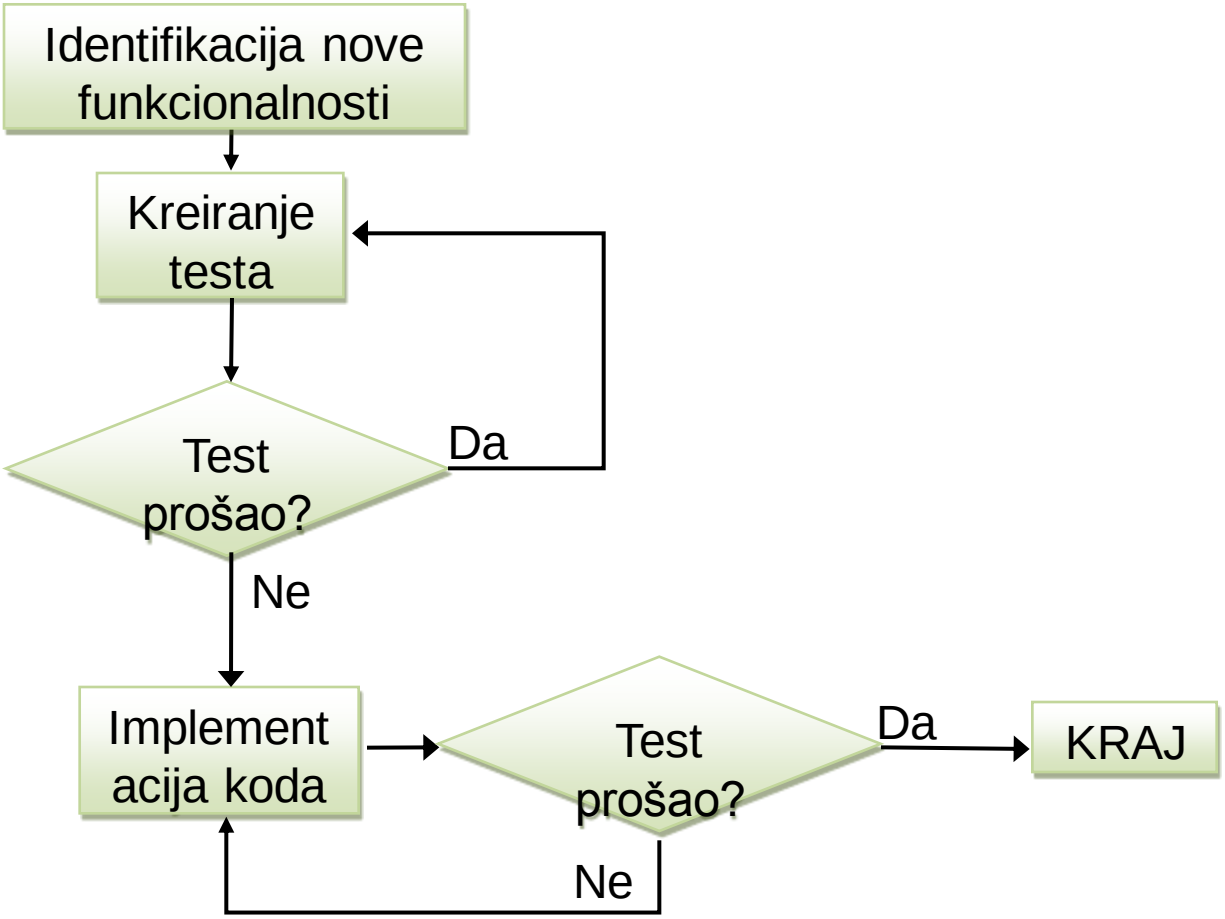
# Primer programiranja po ugovoru (java modeling language)

- Invarijanta klase mora važiti po završetku konstruktora na ulazu i izlazu iz svake javne funkcije članice.
- Za te funkcije definišu se preduslovi (requires) i postuslovi (ensures) koji omoćavaju održavanje invarijante
- Alat JMLUnitNG može automatski da izgeneriše testove za TestNG na osnovu ovoga

```
public class Date {
    /*@spec_public@*/ int day;
    /*@spec_public@*/ int hour;

    /*@invariant 1 <= day && day <= 31; @*/ //class invariant
    /*@invariant 0 <= hour && hour < 24; @*/ //class invariant
    /*@
    @requires 1 <= d && d <= 31;
    @requires 0 <= h && h < 24;
    @*/
    public Date(int d, int h) { // constructor
        day = d;
        hour = h;
    }
    /*@
    @requires 1 <= d && d <= 31;
    @ensures day == d;
    @*/
    public void setDay(int d) {
        day = d;
    }
    /*@
    @requires 0 <= h && h < 24;
    @ensures hour == h;
    @*/
    public void setHour(int h) {
        hour = h;
    }
}
```

# Razvoj zasnovan na testiranju (Test Driven Development)



# Razvoj zasnovan na testiranju

Koraci u procesu su sledeći:

- Izbor inkrementa zahtevane funkcionalnosti. On treba normalno da bude mali i da može da se isprogramira u nekoliko linija koda.
- Piše se test za ovu funkcionalnost i implementira u vidu automatizovanog testa. To znači da test može biti izvršen i daće izveštaj da li je prošao ili nije.
- Zatim se pokrene test, zajedno sa svim ostalim testovima koji su već napisani. U početku, nije realizovana funkcionalnost tako da novi test neće uspeti. To je tako namerno, jer pokazuje da test dodaje nešto skupu testova.
- Zatim se isprogramira funkcionalnost i ponovo pokrene test. To može da podrazumeva refaktorisanje postojećeg koda u cilju poboljšanja ili dodavanje novog koda na već postojeći.
- Kada svi testovi uspešno prođu, prelazi se na implementaciju sledećeg inkrementa funkcionalnosti.

# Integraciono testiranje

# Svrha integracionog testiranja

- Ukoliko su sve komponente uspešno prošle pojedinačno testiranje, zašto bismo imali bilo kakve nedoumice da li će ispravno raditi kada se povežu zajedno u celinu?

# Svrha integracionog testiranja

- Problem je, naravno, u njihovom međusobnom povezivanju.
  - Dok prolaze kroz interfejs, podaci se mogu izgubiti; jedna komponenta može imati nepovoljan uticaj na neku drugu;
  - određena kombinacija podfunkcija može dati neželjen rezultat glavne funkcije;
  - nepreciznosti koje su prihvatljive u pojedinačnim komponentama, mogu u međusobnom povezivanju da narastu do neprihvatljivo velikih vrednosti;
  - globalne strukture podataka mogu da predstavljaju problem, itd.

# Primer

- U septembru 1999, Mars Climate Orbiter misija je propala posle uspešnog putovanja od 416 miliona milja za 41 nedelju. Letelica je nestala kada je trebalo da počne da kruži oko Marsa.
- Defekat je trebalo da bude otkriven integracionim testiranjem: Lokid Martin Astronautika koristila je podatke o ubrzanju u engleskim jedinicama (funte), dok je Jet Propulsion laboratorija uradila svoje proračune sa metričkim jedinicama (newtons). NASA je utrošila 50.000 dolara da otkrije kako je ovo moglo da se desi (Fordahl, 1999).

# Integraciono testiranje

- Integraciono testiranje (ponegde se zove integracija i testiranje, I&T) je faza u testiranju softvera u kojoj se pojedinačne kompnente kombinuju i testiraju kao grupa.

- IT sledi jedinično testiranje, a prethodi sistemskom testiranju.

- Integraciono testiranje polazi od komponenata koji su prošle jedinično testiranje, grupiše ih u veće celine, primenjuje testove definisane u planu integracionog testiranja i daje kao izlaz integrisan sistem spreman za sistemsko testiranje.

# Vrste integracionog testiranja

- Integracija po principu “velikog praska”
- Postupna integracija zasnovana na leksičkoj strukturi programa (include, odnosno import package zavisnostima):
  - Integracija od vrha ka dnu
  - Integracija od dna ka vrhu
  - Mešovita (sandwich) integracija
- Postupna integracija zasnovana na grafu poziva
  - Integracija po parovima (Jorgensen)
  - Integracija po susedstvu (Jorgensen)

# Big bang integracija

- Pristup nepostupne integracije, a to je da se program pravi po principu “velikog praska”.
- Sve komponente (moduli) se unapred iskombinuju i onda program testira kao celina.
- Što često za rezultat ima opšti haos. Nailazi se na mnogobrojne greške. Usled veličine i složenosti programa uzroci grešaka se teško nalaze, a zbog toga se i ispravke teško prave. Onog trenutka kada se te greške isprave, nove se pojavljuju i ceo proces traženja i otklanjanja grešaka se tako nastavlja do beskonačnosti.

# Postupna integracija zasnovana na leksičkoj strukturi programa

- Program se pravi i testira u malim postupnim (inkrementalnim) koracima,
  - greške se mogu jednostavnije uočiti i ispraviti;
  - interfejsi se potpunije testiraju, i može se izvršiti sistematičnije testiranje.
- Pristupi:
  - Od vrha ka dnu
  - Od dna ka vrhu
  - Sendvič

# Integracija od vrha ka dnu

- Pri integraciji se kreće na dole niz kontrolnu hijerarhiju, počevši od glavnog kontrolnog modula (glavnog programa).
- Moduli podređeni glavnom se uključuju u strukturu po jednoj od strategija:
  - “po dubini” ili
  - “po širini”.

# Integracija od vrha ka dnu

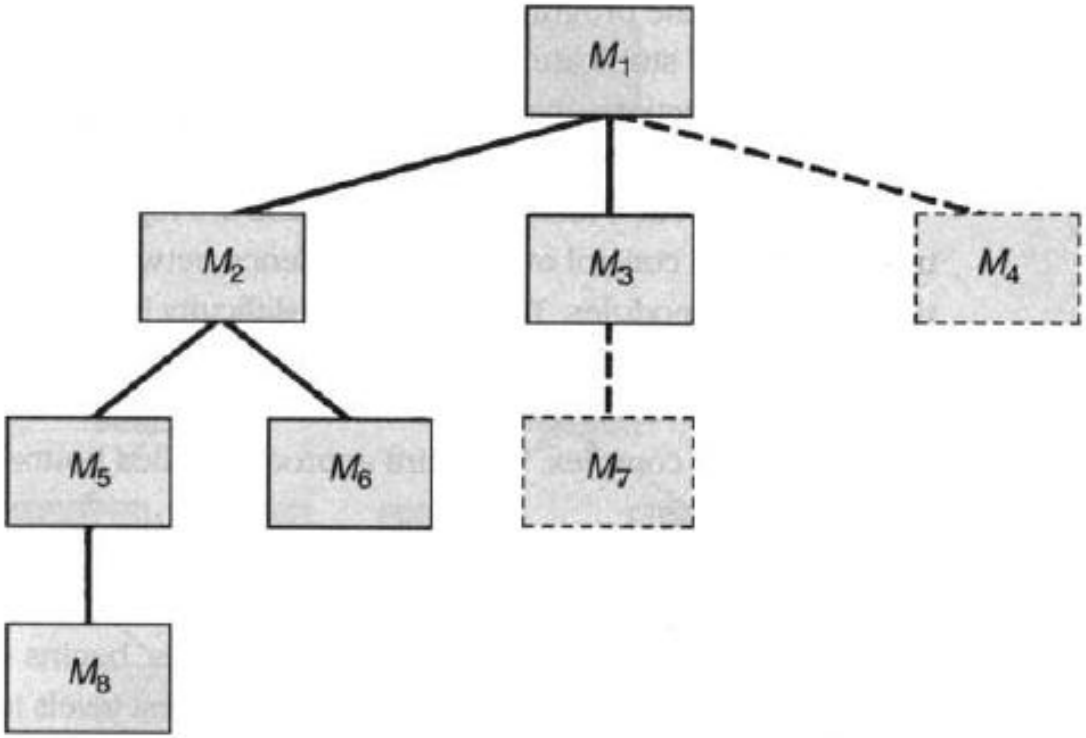
- Strategija “**po dubini**” će izvršiti integraciju svih komponenti na glavnoj kontrolnoj putanji strukture programa. Izbor glavne putanje je donekle proizvoljan i zavisi od karakteristika specifikacije programa (aplikacije).

- Strategija integracije “**po širini**” uključuje sve neposredno podređene komponente na jednom nivou krećući se kroz srukturu programa horizontalno.

# Integracija od vrha ka dnu

- Proces integracije se sprovodi po redu u 5 koraka :
  1. Glavni kontrolni modul se koristi kao test drajver, a sve komponente neposredno podređene glavnom kontrolnom modulu se zamenjuju sa stabovima.
  2. U zavisnosti od strategije integracije (npr. da li je “po dubini” ili “po širini”), podređeni stabovi će se odgovarajućim redosledom, jedan po jedan zamenjivati stvarnim komponentama.
  3. Testovi se sprovode posle svake integrisane komponente
  4. Posle (uspešnog) završetka svake grupe testova, po jedan stab se zamenjuje sa stvarnom komponentom
  5. Može se sprovesti regresivno testiranje da bi bili sigurni da nismo napravili neke nove greške
- Proces se nastavlja od koraka 2. sve dok ne završimo izgradnju kompletne strukture programa.

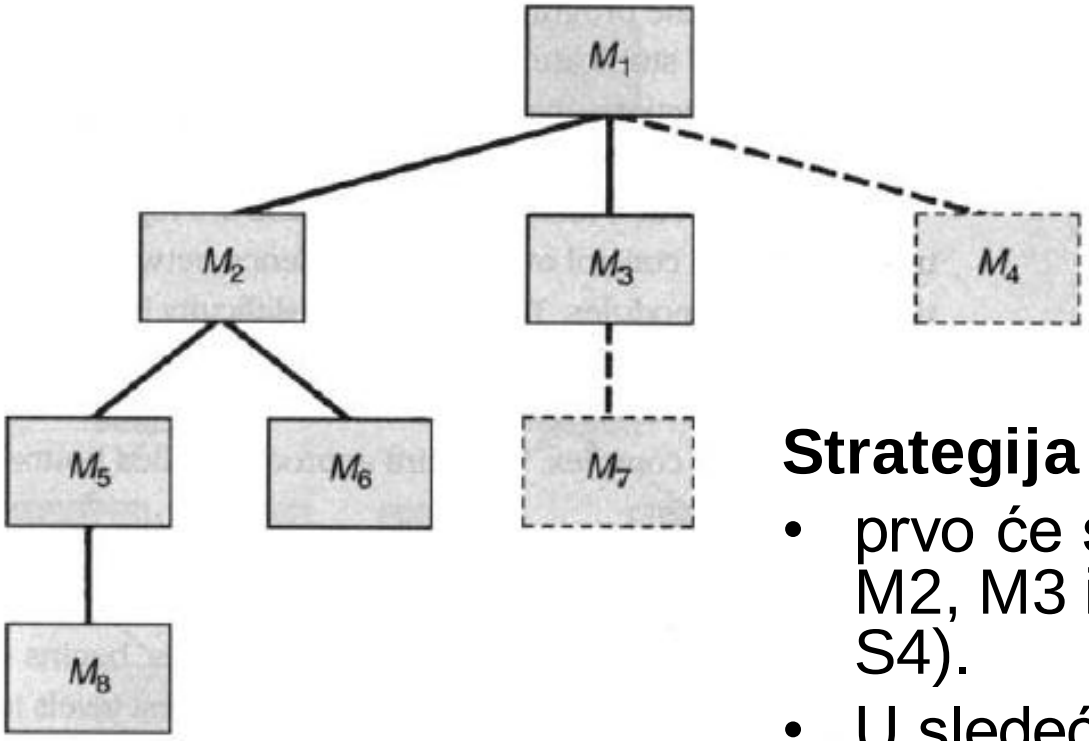
# Primer




**Strategija "po dubini":**

- Izborom leve putanje, prvo će biti integrisane komponente M1, M2, M5.
- Zatim će biti integrisani M8, ili (radi ispravnog funkcionisanje M2) M6.
- Onda će biti napravljene i integrisane, srednja, a zatim i desna kontrolna putanja.

# Primer




**Strategija "po širini":**

- prvo će se integrisati komponente M2, M3 i M4 (i zamena za stab S4).
- U sledećem kontrolnom nivou M5, M6, itd.

# Integracija od vrha ka dnu

- Integraciona strategija “od vrha ka dnu” verifikuje ranije, u procesu testiranja, glavnu kontrolu ili tačke odlučivanja. U dobro izdeljenoj strukturi programa, odluke se donose na višim hijerarhiskim nivoima i zbog toga se prvo tu sa njima i susrećemo. Ako imamo problema u glavnoj kontrolnoj strukturi, njihovo rano otkrivanje je vrlo bitno. Ukoliko odaberemo integraciju “po dubini”, možemo implementirati i demonstrirati upotrebu softvera u potpunosti. U ranim fazama razvoja demonstracije funkcionalnih mogućnosti programa, grade poverenje, kako kod onoga koji razvija taj program, tako i kod naručioca.
- Na prvi pogled strategija “od vrha ka dnu” ne izgleda komplikovano, ali u praksi se mogu javiti neki logistički problemi. Jedan od najčešćih problema te vrste javlja se kada se zahteva obrada u nižim hijerarhiskim nivoima, kako bi se adekvatno testirali viši nivoi. Na početku testiranja “od vrha ka dnu”, stabovi zamenjuju module na nižim hijerarhiskim nivoima.Zbog toga nikakvi značajni podaci ne mogu da cirkulišu od nižih ka višim strukturama programa.

# Integracija od vrha ka dnu

- Onaj ko testira (tester) je zbog ovoga ostavljen sa tri moguća izbora:
  1. Da odloži mnoge testove, dok se stabovi ne zamene stvarnim modulima;
  2. Da konstruiše složenije stabove koji će vršiti ograničene obrade i tako simulirati stvarne module;
  3. Ili da integraciju strukture programa vrši od dna ka vrhu (na više).
- Prvi pristup (odlaganje nekih testova) uzrokuje da izgubimo kontrolu nad vezom između određenih testova i ubacivanja (ugradnje) određenih modula. To može voditi ka teškoćama u određivanju uzroka greškama, što za posledicu ima narušavanje vrlo stroge prirode strategije “od vrha ka dnu”.
- Drugi pristup (složeniji stabovi) može da vrši posao, ali neizostavno vodi ka povećanju cene i dodatnom vremenu i poslu projektovanja sve složenijih i složenijih stabova.
- Treći pristup, koji je nazvan integracijom “od dna ka vrhu” biće razmotren u sledećem odeljku.

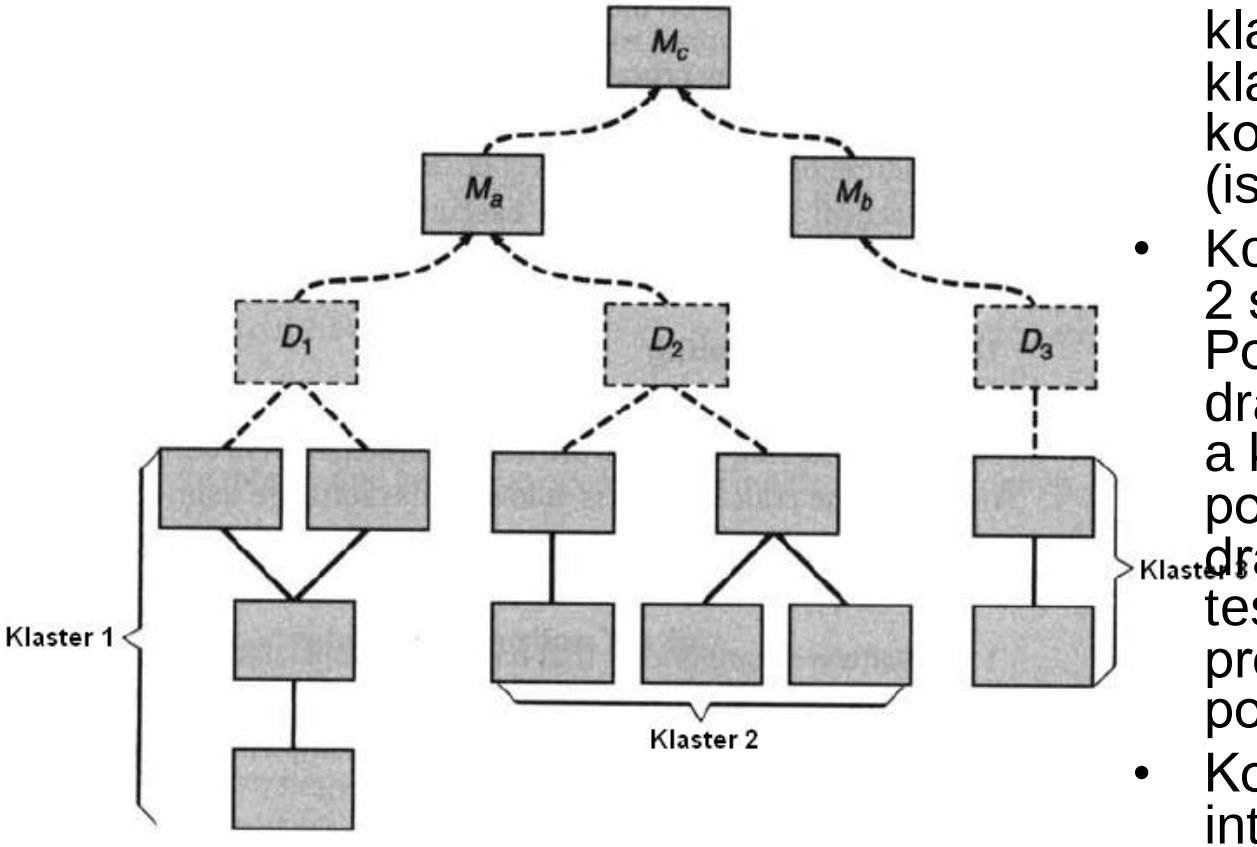
# Integracija od dna ka vrhu

- 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.

# Integracija od dna ka vrhu

- Može implementirati u sledećim koracima:
  1. Da bi se obavila određena softverska podfunkcija, komponente najnižeg nivoa se kombinuju u povezuju u takozvane “klastere” (grozdove).
  2. Da bi se koordinisali izlazi i ulazi test primera napisan je drajver (kontrolni program za testiranje)
  3. Testira se klaster.
  4. Drajveri se uklanjaju, a klasteri se pomeraju nagore u programskoj strukturi, kombinujući ih međusobno

# Primer




- Komponente se grupišu u klastere 1, 2 i 3. Svaki od klastera se, ponaosob, testira korišćenjem drajvera (isprekidani blok).
- Komponente u klasterima 1 i 2 su podređene modulu Ma. Po završetku testiranja drajveri D1 i D2 se uklanjaju, a klasteri se direktno povezuju na Ma. Slično se i drajver D3, koji je služio za testiranje klastera 3, uklanja pre nego što se ceo klaster 3 poveže na modul Mb.
- Konačno će i Ma i Mb biti integrisani sa Mc itd.

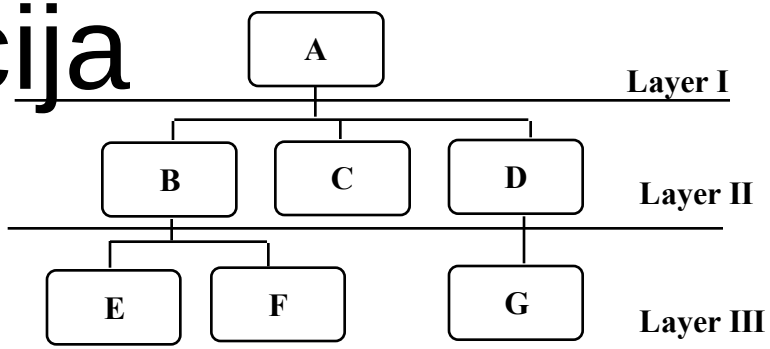
# Od dna ka vrhu - zaključak

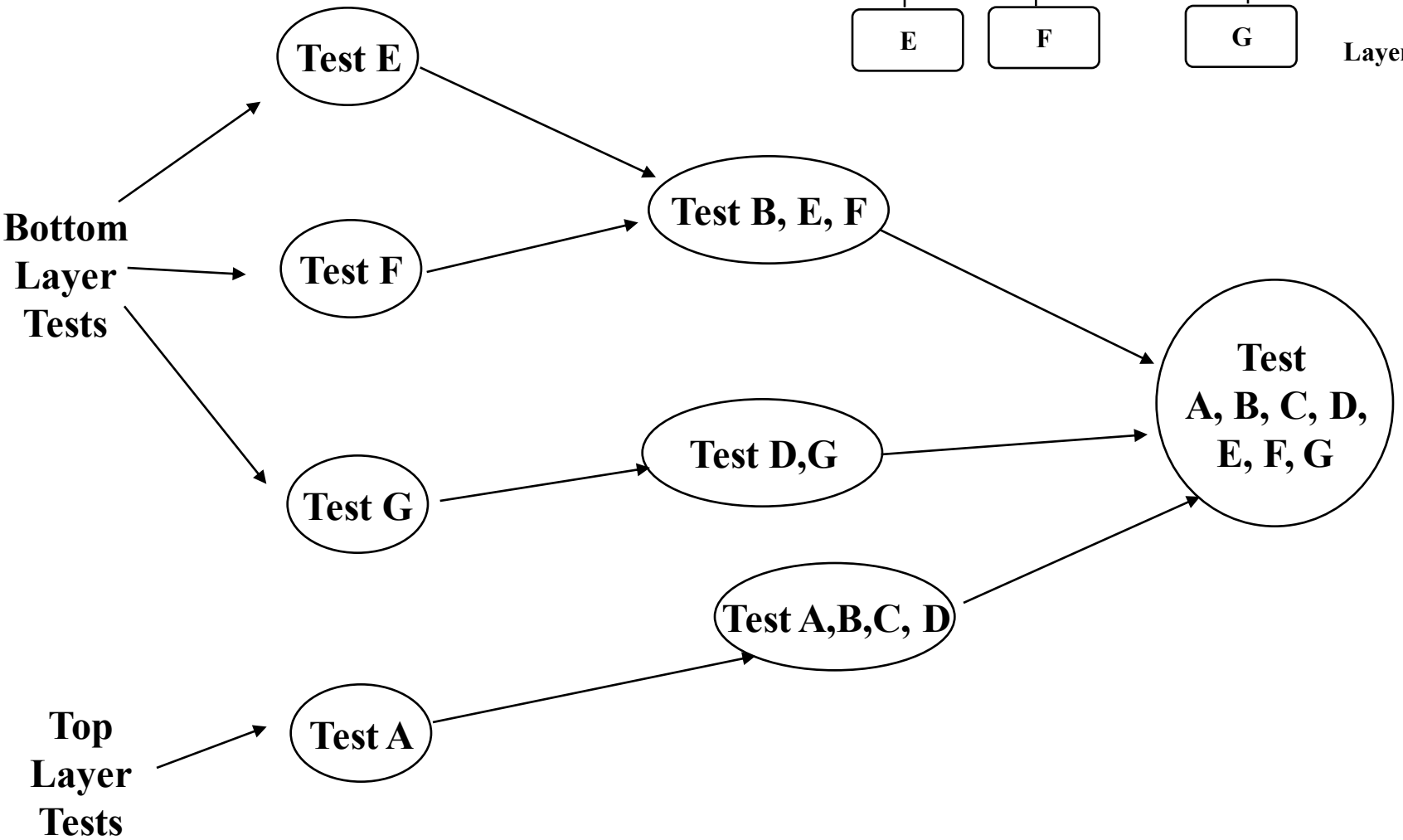
- Prednosti:
  - Manje “throw-away” programiranja
  - Lako kreiranje okruženja za testiranja
  - Laka obrada izuzetaka
- Mane:
  - Nema prototipa
  - Glavni progam se testira poslednji
  - Projektne greške se kasnije identifikuju
  - Relativno visok trošak korekcije grešaka

# Sandwich integracija

- Kombinuje pristupa od vrha ka dnu i od dna ka vrhu
- *Sistem posmatramo kao da ima tri sloja*
  - Ciljni sloj u sredini
  - Sloj iznad ciljnog => pristup od vrha ka dnu
  - Sloj ispod ciljnog => pristup od dna ka vrhu
  - Testiranje konvergira ka ciljnom sloju
- Kako izabrati ciljni sloj kada ima više od tri sloja?
  - Heuristika: Probati minimizovati broj drajvera i stabova

# Sandwich integracija



# Komentari o sandwich integraciji

- Prednosti:
  - Gornji i donji slojevi mogu se testirati u paraleli
  - Manje stubova i drajvera potrebno
  - Lako se konstruišu test primeri
  - Komponente se mogu integrisati čim se implementiraju
- Mane:
- I dalje je potreban “throw-away” kod
- Defekti se teže izoluju pošto se ovaj pristup podseća na veliki prasak ali u okviru podstabla

# Integracija po grafu poziva

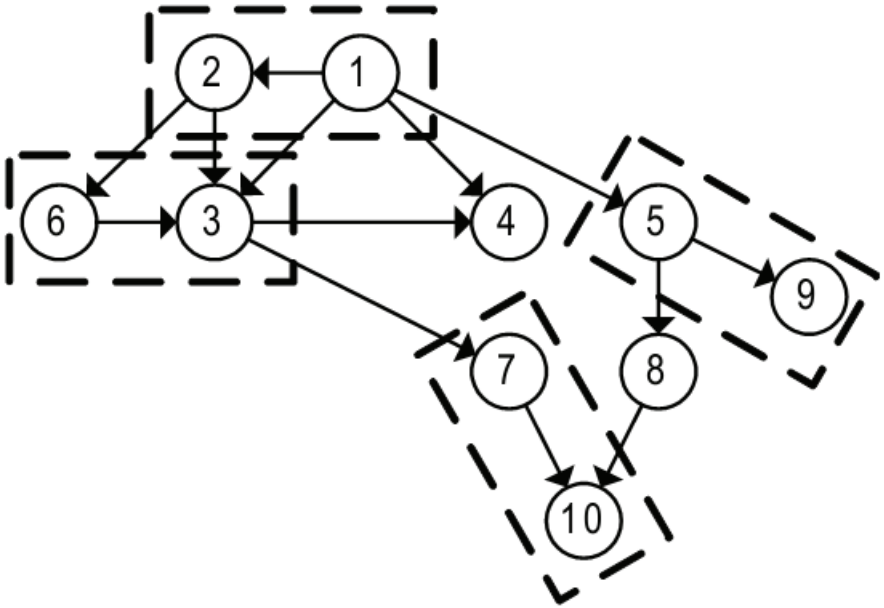
- Argumenti protiv integracionog testiranja (kakvo smo do sada proučili) su veliki broj iteracija (buildova koda), i količina koda koji se odbacuje pre isporuke.
- Argumenti za: testiranje se vrši ranije i defekte je lakše locirati i izolovati
- Ideja integracije po grafu poziva je da se broj iteracija i količina test-only koda smanji, bez da se bitno naruši ranije testiranje i lako lociranje defekata

# Integracija po grafu poziva

- Sistem se predstavlja kao usmereni graf gde su čvorovi komponente a grane predstavljaju interakciju među komponentama putem pozivanja funkcija
- Dva pristupa:
  - Po parovima
  - Po susedstvu

# Integracija po parovima

- Svaka iteracija testiranja upotrebljava kompletan sistem sa realnim kodom, ali se cilja par komponenata koje sarađuju (tj. imaju zajedničku granu u grafu). Time se redukuje problem izolacije otkrivenog defekta. Test sesija je potrebna za svaku posebnu granu u grafu.

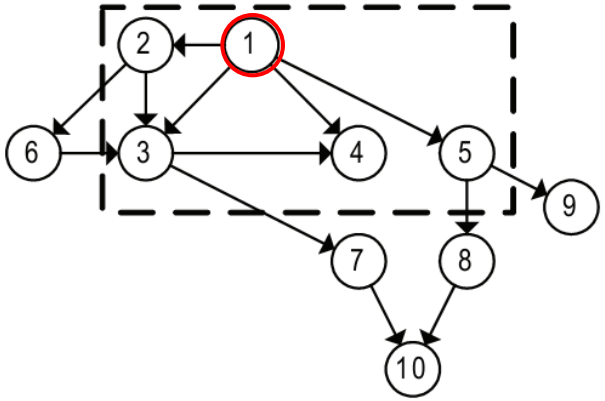


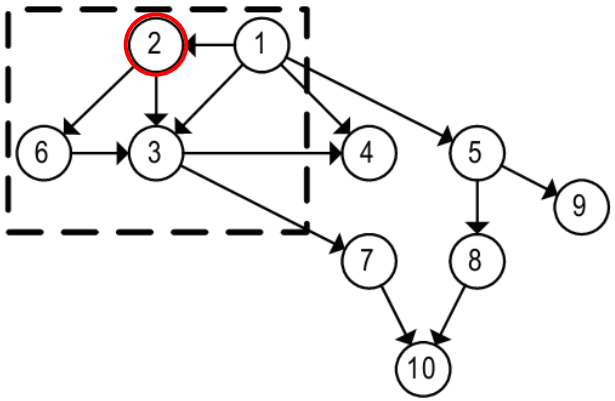
# Prednosti i mane integracije po parovima

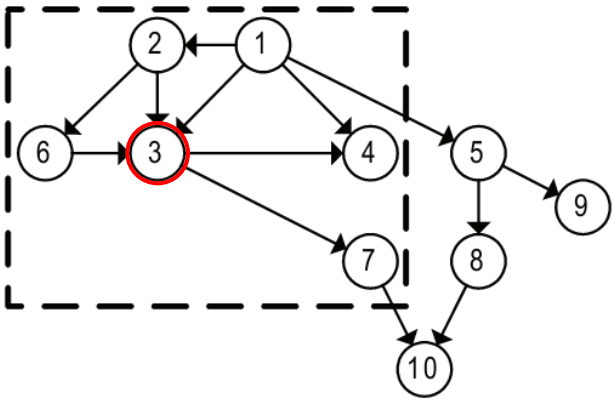
- Prednosti:
  - Eliminisana potreba za stubovima i drajverima i višestrukim prevođenjem koda
  - Upotreba realnog koda
- Mane:
  - Puno test sesija.
  - Testiranje se vrši kasnije jer je potreban kompletno implementiran sistem.

# Integracija po susedima

- Testiranje se obavlja na realnom sistemu kao i kod parova, ali ovde se za potrebe testiranja grupiše ciljni čvor i svi susedni čvorovi (oni koji su u vezi s ciljnim, bilo da su prethodnici ili sledbenici u grafu).



# Prednosti i mane integracije po susedima

- Prednosti:
  - Eliminisana potreba za stubovima i drajverima i višestrukim prevođenjem koda
  - Upotreba realnog koda
  - Redukcija test sesija u odnosu na int. o parovima
- Mane:
  - Teže se izoluju defekti (tzv. medium bang integration)
  - Testiranje se vrši kasnije jer je potreban kompletno implementiran sistem.

# Kritični moduli u integracionom testiranju

- Kako teče integraciono testiranje, onaj ko testira (tester) trebalo bi da uoči *kritične module*.
- **Kritični modul** ima jednu ili više od sledećih nabrojanih karakteristika:
  1. Obrađuje nekoliko softverskih zahteva;
  2. Ima visok nivo kontrole (obično se nalazi na višim nivoima strukture programa);
  3. Složen je ili je verovatnoća pojavljivanja grešaka kod njega veoma velika (ciklomatična (cyclomatic) složenost može biti indikator);
  4. Ima tačno određene zahteve u pogledu preformansi koje treba da zadovolji.
- Kritične module treba testirati što je moguće ranije. Dodatno, regresivno testiranje treba fokusirati na testiranje kritičnih modula.

# Regresivno testiranje

# Regresivno testiranje

- Regresivno testiranje predstavlja metodu testiranja koja se često koristi u iterativnim metodama razvoja softvera.
- Tokom iterativnog razvoja se tokom svake iteracije implementira deo funkcionalnosti sistema i fokus testiranja je na tim funkcionalnostima. Rizik kod ovakve metode testiranja je u tome što funkcionalnosti koje su ranije testirane i potvrđene mogu prestati da rade zato što je tokom trenutne iteracije promenjen neki deo aplikacije koji utiče i na ranije implementirane funkcionalnosti.
- Na ovaj način, u testovima prolaze sigurno samo funkcionalnosti implementirane u trenutnoj iteraciji, dok sve ranije implementirane funkcije mogu da rade pogrešno zato što ih niko ne proverava

# Regresivno testiranje

- **Regresivno testiranje** je metoda kojom se ponavljaju testovi koji su izvršeni u ranijim iteracijama kako bi se ispitalo da li i ranije implementirane funkcionalnosti i dalje rade ispravno. U prvoj iteraciji se testiraju samo funkcionalnosti implementirane u toj iteraciji, u drugoj se testiraju funkcionalnosti kreirane u trenutnoj ali i one koje su već testirane u prvoj iteraciji. U svakoj iteraciji se pored trenutnih funkcionalnosti testiraju i sve one koje su već testirane u prethodnim iteracijama.
- Razlog za ovaj pristup testiranja leži u činjenici da funkcionalnosti napravljene u različitim iteracijama nisu nezavisne celine nego deo isto softvera gde dele podatke i servise. Bilo koja promena u podacima ili servisima koje koristi neka funkcionalnost koja se implementira u trenutnoj iteraciji može da utiče na funkcionalnosti koje su ranije implementirane.

# Regresivno testiranje

- Postoje dva načina da se izvrši regresivno testiranje:
- Potpuno regresivno testiranje gde se u svakoj iteraciji testiraju sve funkcionalnosti implementirane u prethodnim iteracijama.
- Regresivno testiranje zavisnih funkcionalnosti gde se u svakoj iteraciji testiraju samo funkcionalnosti urađene u prethodnim iteracijama koje imaju nekih dodirnih tačaka se funkcionalnostima koje su urađene u trenutnoj iteraciji.
  - Za ovo potrebna informacija tzv. ***Traceability matrix*** u kojoj su funkcionalnosti povezane sa softverskim komponentama koje ih izvršavaju