Testiranje I Kvalitet Softvera

Softverske greške

Idealni ciljevi softverskog inženjerstva

Proizvodnja softvera:

- Koji je apsolutno korektan
- Uz minimalni napor
- Po najniţoj mogućoj ceni
- U najkraćem mogućem roku
- Maksimiziranje profita koji se dobija od softverskog proizvoda
- Proizvodnja softvera koji će zahtevati minimalno održavanje

Zadatak Softverskog inženjerstva je da vidi kako da se najviše približimo idealnim ciljevima. Umeće Softverskog inženjerstva je zapravo najbolji balans po pitanju svih idealnih ciljeva za svaki pojedinačni projekat.

Šta je testiranje softvera?

Manifestacija greške (vidljiva kroz ponašanje softvera) .
Nešto je loše u ponašanju softvera (odstupanje od specifikacije tj zahteva).

Testiranje: Uz pomoć eksperimenata, test-to-fail

- Pronalaženje grešaka u softveru
- Utvrđivanje kvaliteta softvera

Uspešan test:

- Pronalazi bar jednu grešku
- Daje sud o kvalitetu sa maksimalnom pouzdanošću i uz minimalan napor

test-to-pass

Zašto testiranje softvera - proces testiranja dokazuje da je softver **LOŠ!!!**

Šta je testiranje softvera?

„Proces izvršavanja računarskog softvera kako bi se utvrdilo da li su rezultati koje on proizvodi tačni“, Glass „79

„Proces izvršavanja programa sa namerom da se pronađu greške“, Myers „79

„Testiranje programa se može koristiti da se pokaže prisustvo grešaka, ali nikada njihovo odsustvo”, Dijkstra „72

„Cilj nije da se otkriju greške, već da se pruže ubedljivi dokazi da ih nema, ili da se pokaže da određene klase grešaka nisu prisutne“, Hennell „84

„Testiranje je mera kvaliteta softvera“, Hetzel „85

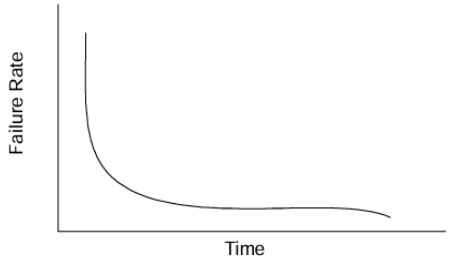
„Proces rada sistema ili komponente pod određenim uslovima, posmatranje ili beleženje rezultata i procena nekog aspekta sistema ili komponente.“, IEEE/ANSI, 1990

Testiranje je i stvar pristupa

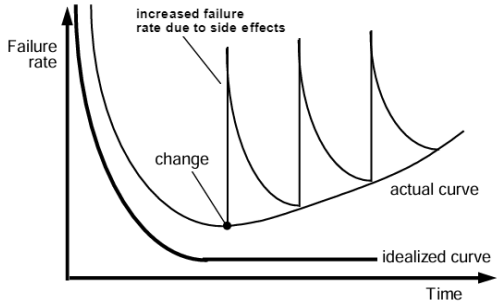
„Ako nam je cilj da pokažemo odsustvo grešaka, naći ćemo ih vrlo malo“
„Ako nam je cilj da pokažemo prisustvo grešaka, otkrićemo ih veliki broj“ (Myers, 1979)

Raspodela Softverskih grešaka

Idealna



Realna



Vrste testiranja

Greška pronađena nakon isporuke košta četiri puta!!!(W. Perry)

1. trošak: razvoj programa pogrešan
2. trošak: sistem mora da se testira da bi se otkrila greška
3. trošak: pogrešnaspecifikacija ili kod treba da se uklone a korektna verzija da se doda tj.generiše
4. trošak: sistem mora da se ponovo testira!

Praksa korekcije grešaka

- Cena korekcije grešaka raste sa etapom u životnom ciklusu u kojoj je greška detektovana 60%grešaka se napravi u fazi projektovanja, 40%u toku implementacije
- 2/3grešaka u projektovanju se otkrivaju tek kada je softver operativan!

Kvalitet softvera

- Neposredni tj.direktni
  - Korektnost
  - Robusnost
  - Sigurnost i pouzdanost
  - Jednostavan za korišćenje
  - Lak za učenje
  - Efikasan
- Dugoročni
  - Proširiv
  - Moguće ga koristiti ponovo (Reusability)
  - Portabilan

Kvalitet

- Kvalitet je odsustvo “nedostataka” (ili “bugova”)
- (IEEE def.) Nedostaci rezultuju iz neke mane izazvane nekim propustom.
- Greška,mana,(Error)–u slučajunedostatka, procena odstupanja od očekivanog rezultata.
- Nedostatak(Failure)–je bilo koji dogadjaj u izvršenju sistema koji narušava neki od kriterijuma kvaliteta

Primer: Failure-Fault-Mistake

Y2K problem

- Failure: godine osoba se javljaju kao negativne!
- Fault: kod za računanje godina daje negativnu vrednost ako je datum rodjenja u 20om veku a tekući datum je u 21om veku.
- Mistake: greška u računanju za datume nakon 20og veka

Zašto softver ima greške?

- Mi smo napravili neki propust
  - Nejasni zahtevi
  - Loše pretpostavke
  - Greške u projektovanju
  - Implementacione greške
- Neki aspekti sistema su teški za predvidjanje
  - Kod velikih sistema, niko ne razume ceo sistem
  - Neka ponašanja je teško predvideti
  - Izuzetno velika složenost
- Evidencija
  - Česte greške usled postojanja“programiranja sa n-verzija”

Obezbedjenje kvaliteta softvera

U toku celog procesa razvoja

- Vremenski plan aktivnosti
- Cena
- Postizanje cilja
- Stalno usavršavanje
- Odgovornost
- Validacija i verifikacija

Kada –Ko?

Uvek -u toku celog razvojnog ciklusa

- Pre –ugradjen u proces:
  - Proces (npr. CMMI, Agile, …)
  - Metodologija (zahtevi, formalne metode, paterni, …)
  - Alati, jezici
- Posle –verifikacija:
  - Testovi
  - Ostale statičke i dinamičke tehnike

Nivoi

- Izbegavanje grešaka
- Detekcija grešaka (verification)

- Otpornost na greške (Fault tolerance)

Posteriori verifikacija

- Statička (bez izvršenja)
  - Rivjui (čovek)
  - Provera koda i korišćenje alata
  - Statička analiza
  - Dokazivanje ispravnosti
- Hibridna (uglavnom statička)
  - Model checking
  - Apstraktna interpretacija
  - Simboličko izvršavanje
- Dinamička (mora da se izvršava)
  - Testiranje

Namena testiranja

- Namena testiranja je pronalazenje "bugova" (Preciznije: izazivanje grešaka koje se detektuju neočekivanim ponašanjem usled postojećih nedostataka)
- Test se može nazvati “uspešnim” ako zapravo “ne prodje”
- Test koji prolazi ništa ne govori (osim ako prethodno isti test nije prolazio) (regresionotestiranje))
- U principu testiranje uključuje ljude koji nisu developeri (ali i njih)
- Testiranje se zaustavlja kada se detektuje bag (ne obuhvata korekciju tj.debagiranje)

Kako testirati ?

- Potpuno testiranje nije moguće
  - test svih validnih ulaza
  - test svih nevalidnih ulaza
  - test svih promenjenih ulaza
  - test svih varijacija u vremenu ulaza
- Dokazivanje korektnosti programa je suvise teško
  - ⇨ izabrati dovoljno mali, jošuvek adekvatan,skup testova tj.slučajeva

Testiranje –okvirna procedura

- Identifikovati deo softvera koji će biti testiran
- Identikovati interesantne ulazne vrednosti
- Identifikovati očekivane rezultate (funkcionalne) i karakteristike izvršenja (nefunkcionalni zahtevi)
- Startovati softver za ulazne vrednosti
- Porediti rezultate i očekivane vrednosti i karakteristike izvršenja programa

Elementi testiranja

- Element koji se testira (Implementation Under Test IUT)
  Softverski (& eventualno hardverski) elementi koji se testiraju
- Test case
  Precizna specifikacija izvršenja kojom se pokriva moguća greška:
  - Stanje i okruženje pre izvršenja
  - Ulazi
- Test run
  Jedno izvršenjetest case-a
- Niz testova
  Kolekcija test case-ova

Elementi: ocena testa

- Očekivani rezultati (za test)
  Precizna specifikacija šta se od testa očekuje ako nema grešaka:
  - Povratne vrednosti
  - Poruke
  - Izuzeci
  - Rezultujuće stanje program & okruženja
  - Ne-funkcionalnekarakteristike(vreme, memorija...)
- Potvrda Testa
  - Mehanizam kojim se odredjuje da li je izvršeni test zadovoljio očekivane rezultate
  - Izlaz je po pravilu: prošao("pass“)ilinije prošao("fail“)

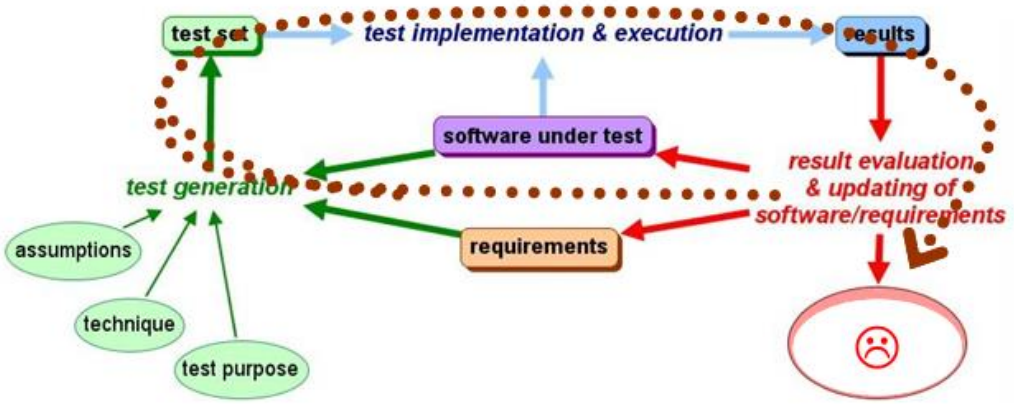
Elementi: izvršenje testa

- Test driver
  Program ili programskielement (npr.klasa), korišćeni za primenu scenarija testiranja na jedan IUT
- Stub
  Privremena implementacija softverskog elementa, koja menja trenutnu implementaciju, u toku testiranja nekog drugog elementa. U opštem slučaju ne zadovoljava potpunu specifikaciju elementa. Može da posluži u slučajevima:
  - Kada softverski element još nije napisan
  - Externi softver koji ne može da se startuje u toku testa(npr. Zato što se zahteva pristup hardveru ili nekoj živoj bazi)
  - Softverski element koji zahteva suviše mnogo vremena i memorije da bi se startovao i čiji rezultati mogu biti simulirani za potrebe testiranja.
- Oprema za testiranje
  Setup, koji uključuje testdrivere i ostale neophodne elemente,koji omogućavaju testiranje

Verifikacija i validacija*

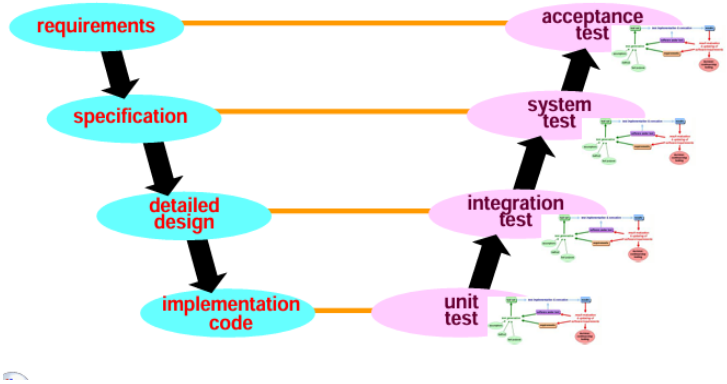
- Verifikacija:Proces evaluacijesistema ili komponenteu cilju odredjivanja da li proizvodi date faze zadovoljavaju uslove postavljene na početku te faze. (Uglavnom paper-based aktivnost koja zahteva potvrdu da svaki korak razvoja ispunjava zahteve definisane u prethodnom koraku.)
- Validacija:Procesevaluacije sistema ili komponente u toku ili na kraju procesa razvoja da bi se odredilo da li on zadovoljava specificirane zahteve. (Uglavnom potvrda da implementirani sistem/komponentaradi po specifikaciji.)

Konceptualna šema testiranja

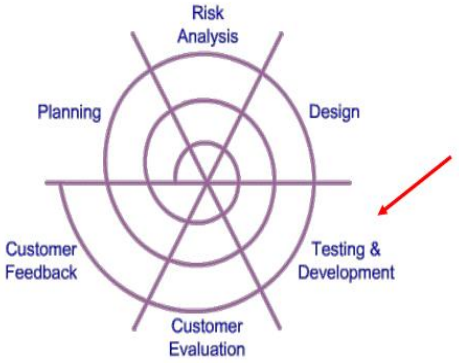


Mesto testiranja u razvoju softveraMesto testiranja u razvoju softvera

V-model:



Agile/spiralni model:



Softver i testiranje

- (faze) Unit vs. Integracija sistema
- (jezik) imperative/object-oriented/hardware design/binary/…
- (interfejs) data-oriented/interactive/ embedded/distributed/…

Zahtevi

- funkcionalni:
  - Ponašanje sistema treba da bude korektno
  - Zahtevi mogu biti precizni ali često to nisu
- ne-funkcionalni:
  - performanse, pouzdanost, kompatibilnost, robusnost(stress/volume/recovery), ...
  - Zahtevi mogu biti kvantitativni, i uvekneodredjeni

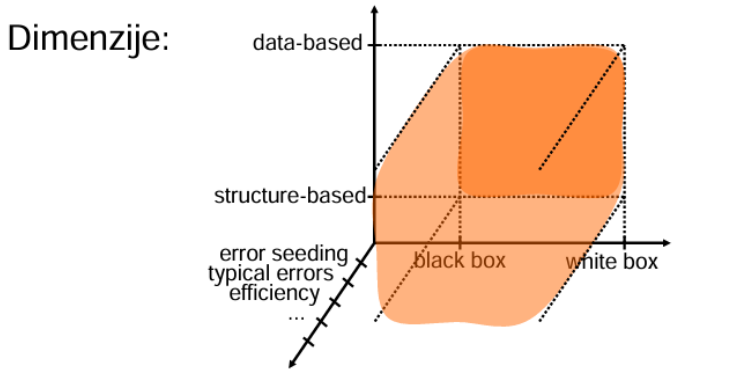
Generisanje testova: namena

Koje greške želimo da nadjemo?

Zavisi od faze u razvoju softvera:

- unit faza
  - Tipične typogreške, funkcionalni propusti
- integracija
  - Greške u interfejsima
- sistem/prijem: greške-zahtevi
  - Neimplementirane funkcionalnosti „softverne radi sve što treba da radi“
  - Implementirane funkcionalnosti koje se ne zahtevaju

Generisanje testova: tehnika



black box: nemamo pristup softveru (strukturi) koji testiramo
white box: imamo pristup softveru

Generisanje testova

Pretpostavke, ograničenja

- obične/višestrukegreške:
  - Objedinjavanje, medjusobna zavisnost grešaka
- Prethodne izmene
- heuristike:
  - Znanje o uobičajenim greškama i propustima
  - Istorija softvera (njegovog razvoja)
  - Paradoks pesticida(Beizer 90: “Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.”

Implementacija testova i izvršenje

Implementacija platforma
- batch?
- ulazi, izlazi, upravljanje, ...

Izvršenje
- Aktuelno trajanje
- manuelno/interaktivnoili automatsko
- U paraleli na više sistema
- ponovljivo?

Ko izvršava koji task?

Realizatori softvera
- programer
- odeljenje za testiranje

Korisnici softvera
- krajnji korisnici
- menadžment

Ostali(Third party)
- Testeri koji su eksterno angažovani
- Sertifikacione organizacije

Evaluacija rezultata

Za test:
- pass/fail resultat
- Dijagnostika izlaza
- Koji zahtevi su bili a koji nisu bili ispunjeni

Statističkeinformacije:
- pokrivenost (programskikod, zahtevi, ulazni domen, izlaznidomen)
- progres testiranja (broj grešaka pronadjenih u jedinici vremena testiranja: smanjenje?)

Odluke:
- kraj (satisfied)
- kreiraj/izvršidodatnetestove(još nije dovoljno sigurno)
- podesisoftveri/ilizahteve, kreiraj/izvršidodatne testove(da se greška ispravi)

Klasifikacija testiranja
- Na osnovu cilja
- Po nameni
- Na osnovu oblasti
- Na osnovu faze u procesu razvoja
- Na osnovu dostupnih informacija

Klasifikacija na osnovu cilja

- Funkcionalno testiranje
- Testiranje performansi
- Stress (“load”) testiranje

Klasifikacijapo oblasti

- Testiranje delova (Unit testing)
- Integraciono testiranje (Integration testing)
- Testiranje sistema (System testing)

Klasifikacija na osnovu dostupnih informacija

White-box testiranje ili

- implementation-based,
- structural,
- glass box,
- clear box.

Black-box testiranje ili

- responsibility-based,
- Functional.

Klasifikacija po nameni

- Testiranje usmereno na otkrivanje greške
- Prihvatno testiranje
- Regresiono testiranje
- Testiranje promenam (Mutation testing)

Klasifikacija po fazi u procesu razvoja

- Unit testing: implementacija
- Integraciono testiranje: integracija podsistema
- Testiranje sistema: integracija sistema
- Prihvatnotestiranje: deployment
- Regresionotestiranje: održavanje

Testiranje je teško: zašto?

Vrste softverskih proizvoda

- Desktop
- Client-server
- Distribuirani
- Web Objektno-orijentisani
- Komponente
- Service orijentisani

Proces testiranja jako zavisi od vrste softvera!!!

Vrste testiranja - strukturno testiranje

UNIT TESTING

Testiranje Unit-ag –se fokusira na najmanje elemente (celine, module) u softveru.

- U kontekstu OO softvera odgovara mu testiranje klasa.

Korišćenje white-box testiranja.

Pri generisanju Unit testova moraju se uzeti u razmatranje mnogi faktori koji mogu da dovedu do greške:

- Interfejs
- Lokalne strukture podataka
- Uslovi
- Nezavisni putevi
- Putevi za obradu grešaka

Generisanje Unit Testa

Razmatranje interfejsa

- # ulaznih parametara= # argumenata?
- Atributi parametaraiargumenatase poklapaju?
- Da li je redosled korektan(akojeto važno)?
- Broj i redosled argumenata korektan?
- Reference na parametre nisu pridružene ulazima?
- Pokušaj modifikacije čisto ulaznih parametara?
- Definicije globalnih parametara konzistentne?
- Ograničenja su uneta kao argumenti?
- Podrazumevani parametri

Razmatranje eksternihI/O

- Atributi fajlova korektni?
- OPEN/CLOSE korektni? Format specifikacije odgovaraju I/O naredbama?
- Buffer size odgovara record size?
- Fajlovi su otvoreni pre korišćenja?
- EOF se koristi korektno? I/O greške se hendluju?
- Tekstualne greške u ulazima?

Razmatranje struktura podataka

- Neodgovarajuće ili nekonzistentno postavljanje tipova?
- Pogrešnainicijalizacija ili default vrednosti?
- Nekorektna imena promenljivih? (jednine, množine)
- Nekonzistentni tipovi podataka?
- Podkoračenje, prekoračenje igreške adresiranja?
- Pointeri

Testovimoraju da pokriju sve puteve

Moraju da se ispitaju greške računanja:

- Nekorektna aritmetika
- Nekorektna inicijalizacija
- Netačna preciznost
- Nekorektna simbolička reprezentacija izraza

Ostala potrebna testiranja

- Nekompatibilni tipovi podataka u poredjenjima
- Nekorektni logički operatori ili prioritet
- Problemi poredjenja(npr. == na podacima tipa float) Problemi sa petljama(granice petlji, uslov za kraj, modifikacija indeksa petlje)

OO kod, Web programi,…

Testovi upravljanja greškama

- Exception-handling je nekorektan?
- Opis grešaka jenerazumljiv, nedovoljan ili nekorektan?
- Greške dovode do pada sistema pre nego što je kompletiran error handling?

INTEGRACIONO TESTIRANJE

- Sistematski pristup u konstruisanju programske strukturepri čemu se testovima otkrivaju greške vezane za interfejse.
- Cilj: Otkriti greške koje se odnose na kompatibilnost modula
- Po V-modelu glavni nivoi granularnosti kod testiranja su testiranje: modula, integracija, sistem i prihvatno testiranje.
- Integracionim testiranjem se ne otklanjaju problemi u modulima.
- Kvalitet sistema zavisi od kvaliteta delova.
- Često nekritični moduli mogu imati značajan uticaj i efekat.
- Primer:2004 god. Buffer overflow u široko korišćenoj biblioteci za čitanje PNG (portable network graphics) izazvala niz problema u web browserima i mail klijentima na Windowsu, Linuxu i Mac OS X

Nekompatibilnost modula

- Ariane 5 incident (4.7.1996. pad)
- Uzrok: nekompatibilnost modula sa zahtevima za Ariane 5
- Modul je za Ariane 4 koja je nešto manja od Ariane 5 i imala je manju horizontalnu brzinu
- Nije testiran za trajektoriju Ariane 5 i uslove njenog leta

Vrste integracionog testiranja

Postoji tendencija ka Ne-Inkrementalnoj integraciji. Uglavnom rezultuje u velikim greškama.

- Inkrementalna integracija-program se konstruiše i testira u malim segmentima.
  - Top-Down Integraciono testiranje
  - Bottom-Up Integraciono testiranje
- Testiranje Integracije kritičnih modula

Top –down integracija

- Početak konstruisanja i testiranja od glavnog modula.
- Stub-ovi menjaju sve podmodule.
- Podmoduli se menjaju aktuelnim modulima jedan po jedan.
- Testovi se dodaju kad se integriše svaki modul.
- Po završetku svakog skupa testova, sledeći stub se menja realnim modulom. Regresionotestiranjemože da se izvrši da se proveri da nisu možda napravljene nove greške.

Prednosti:

- Rano verifikuje glavne kontrolne tačke ili tačke odluke u testiranju.
- Korišćenjem integracionog testiranja po dubini, kompletna funkcijasoftvera može da se demonstrira.

Nedostaci:

- Pošto stub-ovimenjaju module na nižim nivoima, ne mogu značajni podaci da budu prosledjeni u glavni modul.

Bottom up pristup

U ovom načinu konstruisanje i testiranje počinje od modula na najnižim nivoima.

- Moduli na nižim nivoima se kombinuju u klastere.
- Pišu se drajveri da bi mogli da se koordiniraju ulazi i izlazi testova.
- Klaster se testira.
- Drajveri se uklanjaju i klasteri se kombinuju uz pomeranje naviše po hijerarhiji.

Prednosti:

- Lakše generisanje testova i nema stubova.

Nedostaci:

- Program kao celina ne postoji sve dok se ne integriše i poslednji modul.

Sendvič testiranje: kombinovani pristup

- Top-down pristup za gornje nivoe i Bottomup pristup za podnivoe.

REGRESIONO TESTIRANJE

Ponovno izvršenje nekih testova radi provere da li je neka izmena unela bočne efekte. Regresiono testiranje treba da sadrži tri različite klase testova:

- Reprezentativni skup testova koji izvršavaju sve funkcije softvera
- Dodatni testovi koji se fokusiraju na funkcije na koje učinjene promene mogu da utiču.
- Testovi koji se fokusiraju na komponente koje se menjaju.

VALIDACIONO TESTIRANJE

- Konačna provera da li softver ispunjava sve funkcionalne zahteve kao i performanse.--Isključivo korišćenjeBlack-box testiranja.
- Nakon svakog validacionog testa ili se potvrdjuje da je softver po specifikaciji ili se pak detektuje neko odstupanje.
- Alpha iBeta testiranje
  - Alpha test–na strani developera od strane korisnika.

- Beta test–Na strani korisnika u živom okruženju.

TESTIRANJE SISTEMA

Niz testova kojima se verifikuje da su svi elementi sistema ispravno integrisani.

Recovery Testiranje:

- Softver se dovodi u stanje greške na različite načine i zatim se testira da li se oporavljanje (recovery) obavlja korektno.

Security Testiranje:

- Verifikacija zaštitnih mehanizama softvera.

Stress testiranje:

- Izvršava se sistem na način gde se koristi abnormalna količina resursa, frekvenca ili veličina.

Testiranje performansi:

- Testiranje run time performansi sistema u kontekstu integrisanog sistema.

STRUKTURNO TESTIRANJE

- Sinonimi–Testiranje bele kutije, testiranje zasnovano na implementaciji, clear box, itd.
- Poznata struktura koda
- Vrste
  - Testiranje naredbi
  - Testiranje grana
  - Testiranje odluka
  - Testiranje puteva
  - Testiranje poziva procedura

SEKVENCE I SKOKOVI (LINEAR CODE SEQUENCE AND JUMP)

- Polazna tačka u LCSAJ je:odredišna linija skokaili prva linija programa.
- Krajnja tačka u LCSAJ je bilo koja linija do koje se može doći od polazne tačke sekvencom i iz koje je moguće napraviti skok.
- LCSAJ je odredjena polaznom linijom, krajnjom linijom i odredišnom linijom(na koju se skače).

LCSAJ

- Moraju da se identifikuju svi LCSAJu programu
- Treba da se generišu testovi koji izvršavaju sve LCSAJ

Primer:Funkcija za izračunavanje faktorijela

```
(1) function factorial(n : integer) return integer is
(2)   result : integer := 1;
(3) begin
(4)   for I in 2 .. n loop
(5)         result := result *I;
(6)   end loop;
(7)   return result;
(8) end factorial
```

LCSAJs za program za faktorijel:

LCSAJ	start	end	odredište
1	3	4	7
2	3	6	4
3	4	6	4
4	4	4	7
5	7	7	exit

Osobine LCSAJ
Prednost: Obezbedjuje izvršenje petlji potpunije od drugih tehnika
Nedostatak:Teško za primenu

Testiranje toka podatka

- U prvi plan se stavljaju podaci tj. način njihovog korišćenja.
- Generišu se testovi koji slede šablon definicije podataka i njihovog korišćenja u programu.
- Originalno korišćen za statističko otkrivanje anomalija u kodu.
- Klasifikovanje pojavljivanja promenljivih u **3** kategorije:
  - ***def:*** definicija
  - ***c-use:*** (computational-use)
  - ***p-use****:* (predicate-use)

Pokrivanje toka podataka

- Definišu se za promenljivu: **def**, **c-use** i **p-use** mesta u kodu.
- Promenljiva V ima ***def*** na svim mestima u kodu gde uzima neku vrednost (npr. neka dodela).
- V ima ***c-use*** na mestima u kodu gde se koristi u evaluaciji izraza ili u nekoj naredbi izlaza.
- V ima ***p-use*** na mestima gde se koristi u iskazima i tako utiče na tok kontrole samog programa. *def* i *c-use* se predstavljaju čvorovima a *p-use* kao grane u grafu.

- **d-u parovi**

Glavna ideja kod testiranja koje se zasniva na pokrivanju toka podataka:

- Identifikovati i klasifikovati sva pojavljivanja promenljivih u programu i za svaku promenljivu generisati test tako da se izvrše sve definicije i sve linije gde se promenljive koriste.
- Ne pravi se razlika izmedju *c-use* i *p-use* mesta.
- Koristi se naziv d-u parovi (**definicija-korišćenje**).

- Primer DU puteva:

```
1.x := 0;
2.      while x < 2
3.      begin
4.              writeln (“looping”);
5.              x := x + 1;
```

6. end

DU putevi za x su: 1 - 2, 1 – 2 – 3 – 4 - 5, 5 – 6 - 2, 5 – 6 – 2 – 3 – 4 - 5.

- Broj DU parova je uvek konačan.
- Skup testova pokriva sva mesta korišćenja ako su ulazni podaci takvi da dovode do izvršenja svakog DU para za sve promenljive

- **Kriterijumi za pokrivanje:**
  - all-nodes (statement coverage)
  - all-edges (branch coverage)
  - all-defs
  - all-p-uses
  - all-c-uses/some-p-uses
  - all-p-uses/some-c-uses
  - all-uses
  - all-du-paths
  - all-paths

- **svi-du-putevi**
  - Najstroži oblik testiranja toka podataka.
  - Uključeni su i p-use i c-use pa se mogu označiti kao ***uses.***
  - Identifikuju se sva pojavljivanja promenljivih u programu i onda se za svaku promenljivu generiše test tako da se izvrše svi du-parovi.

Zadatak: *funkcija factorial.* - Konstruisati tabelu sa DU parovima. Generisati testove za izrvšenje svih du parova.

```
(1)int factorial(int n) {
(2)     int result = 1;
(3) {
(4)     for (int i=2; i < n; i++)
(5)             result = result * i;
(6) }
(7) return result;
(8) }
```

Primer: *Obrada stringa*

```
main() {
(1)     char a[20], ch, response = „y“;
(2)     int x, i;
(3)     bool found;
(4)     cout << “Input an integer between 1 and 20: “;
```

```
(5)     cin >> x;
(6)     while ( x < 1 || x > 20) {
(7)             cout << "Input an integer between 1 and 20: ";
(8)             cin >> x;
        }
(9)     cout << "input " << x << " characters: ";
(10)    for (int i = 0; i < x; ++i) cin >> a[i];
(11)    cout << endl;
(12)    while ( response == „y“ || response == „Y“) {
(13)            cout << "input character to search: ";
(14)            cin >> ch;
(15)            found = false;
(16)            i = 0;
(17)            while (!found && i < x)
(18)                    if (a[i++] == ch) found = true;
(19)            if (found) cout <<  ch << " at: " << i << endl;
(20)            else cout << ch << " not in string" << endl;
(21)            cout << "search for another character? [y/n]";
(22)            cin >> response;
        }
```

DU-parovi za primer za obradu stringa:
Promenljive su *x*, *i*, *ch*, *found*, *response*, i polje *a*:

- Za **x** - 8 du-para
- Za **i** - 9 du-para
- Za **ch** - 3 du-para
- Za **found** - 4 du-para
- Za **response** – 1 du-par
- Polje **a** je problem jer je teško da se odredi koji element polja se koristi.
  Na primer, *i* se menja dinamički u toku izvršenja programa.
  Rešenje: tretirati celo polje kao jednu promenljivu ==> **samo jedan DU-par** (10-18)

Karakteristike testiranja toka podataka:

Prednosti:

- Strog oblik testiranja
- Generišu se testovi koji prate način manipulacije podacima umesto praćenja "veštačkih" grana kao u prethodnim vrstama strukturnog testiranja
  Problem: pointeri; teško je odrediti koja promenljiva je referencirana

Nedostatak:

- Teoretski granica broja testova je eksponencijalna. Zašto? ***2^d***, gde je *d* broj odluka u programu sa 2 grane.
- U praksi mali broj testova je potreban.

Automatsko testiranje toka podataka

Globalna analiza toka podataka:

- Određivanje informacija o toku podataka može biti automatizovano.
- Potrebne su informacije o mestima definisanja (l-values) i mestima korišćenja (r- values).
- "**globalno**" znači lokalno za funkciju ali *globalno* (duž blokova) za graf toka kontrole.
- Podsećanje: blok (jedna ulazna i izlazna tačka).
- Koriste se jednačine toka podataka.

Jednačine toka podataka:

out[S] = gen[S] U (in[S] - kill[S])

out[S] - skup svih def koji napuštaju blok S (tj. One su "žive").
gen[S] – sve nove def koje su generisane u bloku S
in[S] - svi def koji ulaze u blok S
kill[S] - svi def koji su ubijeni sa def u bloku S

Kada pišemo *out[S]* podrazumevamo da postoji jedinstveni **end point** od koje kontrola ide van bloka S tj. napušta blok.
Postoje delovi koda (funkcije) koje su "zakačene" za *procedure calls, pointere* i *polja* radi prikupljanja podataka tj. detektovanja interesantnih situacija.

Dostizanje definicije:

- *definicija* od *x* je naredba koja dodeljuje u *x*.
- *definicija d* dostiže tačku *p* ako postoji put od tačke definicije do *p*, tako da *d* nije ubijena duž tog puta.

Predstavljanje skupova:

Skup definicija (npr. *gen[S]*) može se kompaktno predstaviti korišćenjem vektora bitova

- Svakoj definiciji treba dodeliti jedinstveni broj.
- bit vektor koji predstavlja skup definicija ima 1 na poziciji *i* ako je definicija *i* u skupu.

Algoritam za odredjivanje dostizanja definicija:

Ulaz: cfg za koje *kill[B]* i *gen[B]* su izračunate za svaki blok *B*
Izlaz: in[B], out[B] za svaki blok *B*

metod: koristi se iterativni pristup, polazeći od toga da je *in[B]* prazan skup za sve *B*.
Boolean promenljiva, *change*, se koristi za registrovanje da li je in promenjeno u bloku u svakoj iteracij. Ako nije, onda kraj.

Za svaki blok *B* do *out[B] = gen[B]*
change = true
while *change* do begin

change = false
for each block *B* do
in[B] = U out[P], P a predecessor of *B*
oldout = out[B]
out[B] = gen*[B] U (in[B] - kill[B])*
if *out[B]* != *oldout* then *change* = true
end for
end while

Odredjivanje skupova *in* i *out*

Block B	initial	initial	pass 1	pass 1	pass 2	pass 2
	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B1	000 0000	111 000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Funkcionalno (BlackBox) Testiranje

Šta je black box testiranje?

- Poznato i kao testiranje zasnovano na **specifikaciji**.
- Black box testiranje se odnosi na test aktivnosti korišćenjem metoda zasnovanih na specifikaciji i kriterijuma u cilju otkrivanja grešaka zasnovanih na zahtevima i specifikaciji.

Fokusi testiranja su:

- Funkcionalne greške (na osnovu specifikacije)
- Greške ponašanja komponenata/sistema (na osnovu specifikacije)
- Greške po pitanju performansi (na osnovu specifikacije)
- Greške u korišćenju iz korisničkog ugla
- Greške u interfejsima

Kod black-box testiranja testiraju se: Softverske komponente, podsistemi, ili sistemi.

Šta je potrebno imati?

- Za softverske komponente: specifikacija komponente, dokumentaciju za interface.
- Za softverski podsistem ili sistem: specifikacija zahteva, i dokument o specifikaciji proizvoda.
- Takođe je potrebno:
  - Metode za Specification-based testiranje softvera.
  - Kriterijumi za Specification-based testiranje.
  - Dobro razumevanje softverskih komponenata (ili sistema).

Primer: *Testiranje trougla*

Specifikacija programa :

Ulaz: 3 broja odvojena zarezima ili blanko znakom

Obrada:

Odrediti da li 3 broja mogu da čine trougao; ako ne mogu, prikazati poruku NIJE TROUGAO.
Ako čine trougao, klasifikovati ga kao nejednakostranični, jednakokraki i jednakostranični.
Ako čine trougao, klasifikovati ga u odnosu na najveći ugao kao: oštrougli, tupougli ili pravougli.

Izlaz: U jednoj liniji se prikazuju: 3 broja koja čine ulaz, klasifikacije ili poruka da nije trougao.

Primer:3,4,5 Nejednakostarnični Pravougli
6,1,6 Jednakokraki Oštrougli
5,1,2 NIJE TROUGAO

Funkcionalni Testovi:

	Oštrougli	**Tupougli**	**Pravougli**
Nejedn:	6, 5, 3	5, 6, 10	3, 4, 5
Jednakokr:	6, 1, 6	7, 4, 4	1, 1, 2^(0.5)
Jednakostr:	4, 4, 4	Nije moguć	Nije moguć

Funkcionalni Testovi:

Ulaz	**Očekivani rezultati**
4,4,4	jednakostr. oštrougli
1,2,8	Nije trougao
6,5,3	Nejednak. Oštro.
5,6,10	Nejednak. Tupo.
3,4,5	Nejednak. Pravo.
6,1,6	Jednakokr. Oštro.
7,4,4	Jednakokr. Tupo.

Testovi za specijalne vrednosti ulaza i nekorektan format:

3,4,5,6	četiri strane
646	trocifreni broj
3,,4,5	dva zareza
3 4,5	nedostaje zarez
3.14.6,4,5	dve decimalne tačke
4,6	dve strane
5,5,A	karakter kao strana
6,-4,6	negativan broj kao strana
-3,-3,-3	svi negativni brojevi prazan ulaz

Granični testovi:

(1) Granični uslovi za korektne trouglove

1,1,2	Obrazuju liniju a ne trougao	
0,0,0	Obrazuju tačku a ne trougao	
1,2,3.00001	Vrlo blizu trougla ali jos nije trougao	
9170,9168,3	Vrlo mali ugao	Nejedanko. Oštro.
.0001,.0001,.0001	Vrlo mai trougao	jednakostr. Oštro.
83127168,74326166,96652988	Vrlo velik trougao, nejedanko, tupoug.	

(2) Granični uslovi za klasifikaciju po stranama:

3.0000001,3,3	Vrlo blizu jednakostr. , jednakokrak, oštro.

2.999999,4,5 Vrlo blizu jednakokr Nejedankostr, oštro.

(3) Granični uslovi za klasifikaciju po uglovima:

3,4,5.000000001 blizu pravouglom nejedankostr. Tupo.
1,1,1.41141414141414 blizu pravouglom jedankokrak, oštro

Black-box metode

- Particionisanje u ekvivalentne skupove
- Analiza graničnih vrednosti
- Pretpostavljanje grešaka
- Graf posledičnih efekata

1. Particije ekvivalencije

– Na osnovu specifikacije.
– Nije potrebno posmatrati kod.
– Podeli ulaze u **particije ekvivalencije** ili klase podataka.
– Svaka klasa se tretira na isti način.
– Bilo koji podatak izabran iz klase je validan kao i bilo koji drugi.
– **Prednost:** smanjuje se ulazni domen na razumnu veličinu.
– Da bi se identifikovale particije treba da se u specifikaciji traže pojmovi tipa "**opseg**", "**skup**" I sl.
– Klase ekvivalencije se ne preklapaju!
– Pored izbora jednog podatka iz klase, ***nevalidni podaci*** takodje mogu da se izaberu.

Primer 1: ***Kvadratna jednačina*** – iz specifikacije se vidi da postoje slučaj sa 0, 1 ili 2 realna korena
Uzmimo proces ocenjivanja gde su ocene kategorisane kao A, B, C, D, F.
Pretpostavimo A>=90, B>=80, C>=70, D>=60.
Mogu se particionisati ocene, g, u sledeće particije: 0<=g<60, 60<=g<70, 70<=g<80, 80<=g<90, g>=90 i g>100, g<0.

ulaz	izlaz	Očekivani izlaz
50		F
65		D
75		C
85		B
95		A
-5		Nekorektan ulaz
105		Nekorektan ulaz

Primer 2: ***Program za traženje karaktera u stringu:***
Program omogućuje unos pozitivnog celog broja iz intervala 1 do 20 a zatim i niz karaktera te dužine. Nakon toga program očekuje unos karaktera i nakon unosa vraća poziciju prvog pojavljivanja tog karaktera u prethodno unetom nizu karaktera.
Ako se uneti karakter ne nalazi u nizu prikazuje se poruka.
Korisnik ima mogućnost ponovnog traženja pozicije.

Postoje 3 klase ekvivalencije:

- Jedna klasa celih brojeva u dozvoljenom opsegu
- Dve klase celih brojeva van opsega

Izlazni domen se sastoji od dve klase:

- Pozicija na kojoj je pronađen karakter u stringu
- Poruka da nije pronađen karakter

x	Ulazni string	search karakter	response	Očekivani izlaz
34				Loš ulaz
0				Loš ulaz
3	abc	c		pos = 3
			y	
		k		Nije u stringu
			n	

Prednosti:

- Smanjuje se veličina ulaznog domena
- Pogodan za aplikacije gde se lako identifikuju ulazne promenljive i mogu uzeti različite vrednosti
- Nije jednostavno za aplikacije gde je ulazni domen jednostavan a obrada složena

Problem:

- Specifikacija može da sugeriše da se klasa podataka obrađuje identično što ne mora da bude slučaj
- primer: y2k problem
- Takodje tehnika ne nudi algoritam za pronalaženje particija

2. Analiza graničnih vrednosti

- Koristi se u vezi sa particionisanjem klasa ekvivalencije.
- Ova tehnika se fokusira na slične izvore grešaka: granice klasa ekvivalencije.
- Tehnika se oslanja na već kreirane klase ekvivalencije.

ulaz	izlaz	Očekivani izlaz
50		F
65		D
75		C
85		B
95		A
-5		Nekorektan ulaz
105		Nekorektan ulaz
60		D
70		C
80		B
90		A

Primer: ***Program za traženje u stringu:***

Celobrojne vrednosti 0, 1, 20 i 21 su očigledan izbor.

Kao i pronalaženje karaktera na prvoj i poslednjoj poziciji.

x	Ulazni string	search karakter	response	Očekivani izlaz
21				Van opsega
0				Van opsega
1	a	a		pos = 1
			y	
		X		Ne nalazi se u stringu
			n	
20	abcdefghijkl mnopqrst	a		pos = 1
			y	
		t		pos = 20
			n	

3. Pretpostavka greški

- ***ad hoc*** pristup, zasnovano na intuiciji i iskustvu.
- Identifikuj testove koji su slični otkrivenim greškama.
- Napraviti ***listu*** mogućih grešaka ili situacija koje izazivaju greške i generisati testove na osnovu liste.
- neki slučajevi:
  - Prazna ili null lista/string
  - Nula instanci/pojavljivanja
  - Blanko ili null karakteri u stringovima
  - Negativni brojevi.

Prednosti / Nedostaci:

- Intuicija se često koristi.
- Efikasna tehnika.
- Tehnika bazirana na iskustvu; nije uvek dostupno.
- ad hoc priroda ostavlja sumnju o kvalitetu testiranja: "da li sam zaboravio neku situaciju gde se uobičajeno javljaju greške?"

4. Graf posledičnih efekata

- Funkcionalna ili na specifikaciji zasnovana tehnika.
- Sistemski pristup u selekciji skupa testova koji predstavlja kombinaciju ulaznih uslova.
- Rigorozna metoda za transformisanje specifikacije u prirodnom jeziku u specifikaciju izraženu formalnim jazikom.
- Otkriva nekompletnost i dvosmislenost u specifikaciji.

- Specifikacija se analizira i
- Svi mogući slučajevi se identifikuju: ulazi, sve što otkriva neki odgovor sistema.
- Svi mogući efekti se identifikuju: izlazi, izmene u stanju sistema.
- Razlozi (uslovi) i efekti moraju da se daju tako da mogu da se evaluiraju kao tačni ili netačni (zadovoljeni ili ne).

- Uslovi i efekti se kombinuju u boolean graf koji opisuje njihove relacije.
- Svaki uslov i efekat ima jedinstven broj.
- Kreira se graf koji pokazuje veze izmedju uslova i efekata.
- Konstruišu se testovi koji pokrivaju sve moguće kombinacije uslov/efekat.

Grafovi se kombinuju korišćenjem operatora:

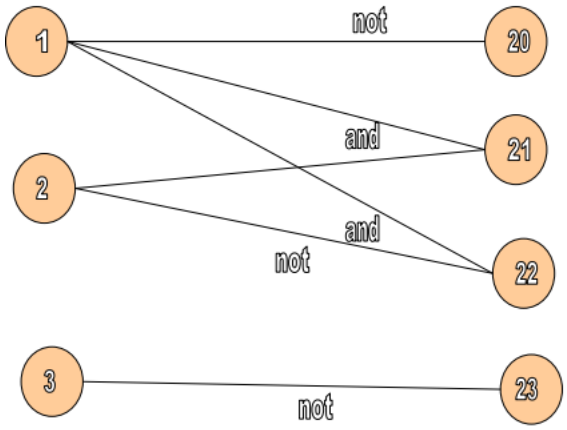
- **not**
- **and**
- **or**
- **nor**

Primer: ***ispitivanje stringa***
Uslovi:

(1) ceo broj u opsegu 1-20
(2) traženje karaktera u stringu
(3) traženje sledećeg karaktera

Efekti:

(20) ceo broj van opsega
(21) pozicija karaktera u stringu
(22) karakter nije pronadjen
(23) program se završava



Prednosti / Nedostaci:

- Vrši se kombinacija testnih podataka.
- Očekivani rezultati su deo procesa kreiranja testova.
- Glavna posledica je boolean graf: veliki broj uslova i efekata dovodi do velikog grafa.
- Rešenje ovog nedostatka: identifikovati podprobleme.

Karakteristike:

- Korisno za podešavanje specifikacije.
- Može se pristupiti testiranju pre kodiranja.
- Može da pronađe klase grešaka koje drugi pristupi ne mogu.
- Mogu se koristiti i formalna i neformalna specifikacija zahteva tj. na bilo kom nivou.
- Normalno, kompletna specifikacija daje bolje testove.
- Lako se otkrivaju izostavljeni zahtevi, kontradiktornosti itd.
- Ako su zahtevi izostavljeni, nema koda pa strukturno testiranje ne može otkriti ovakve slučajeve.
- Korisno za bolje pojašnjenje specifikacije.
- Nije skupa metoda, jeftinija od strukturnog testiranja.
- Primenjuje se na sve nivoe granularnosti:
  - Unit (specifikacija modula)
  - Inegracija (specifikacija podsistema)

- Sistem (specifikacija sistema)
- Regresiono testiranje (specifiklacija sistema + istorija bug-ova)

Postupak:

- ❖ Dekompozicija specifikacije (ako je specifikacija velika izdvojiiti manje nezavisne celine).
- ❖ Selekcija reprezenata (ulazi, izlazi tj ponašanje modela).
- ❖ Kreiranje specifikacije testova.
- ❖ Kreiranje testova i njihovo izvršavanje i analiza rezultata.

Testiranje objektnoorijentisanog softvera

Nivoi testiranja:

Konvencionalni softver:

- Počinjemo sa **jediničnim testiranjem** (modula, procedura i/ili komponenata).
- Sledi **integracija komponenata** (strategije integracionog testiranja i regresivno testiranje) da se otkriju greške u interfejsima modula i bočni efekti dodavanja novih modula
- Na kraju se sistem testira **kao celina** da se otkriju eventualne nesaglasnosti u odnosu na početne zahteve (testiranja višeg reda).

1) Jedinično testiranje OO softvera

- Najmanja jedinica testiranja je **klasa** tj. objekti koji obuhvataju podatke i funkcije za manipulaciju tim podacima.
- Testiranje klase za OO softver je ekvivalent jediničnog testiranja kod konvencionalnog softvera.
- Metod klase ne može se testirati izolovano nego samo u kontekstu cele klase (i njenih osnovnih klasa).

Zašto ne nezavisno testiranje svake metode klase?

- o Ponašanje metode klase ne zavisi samo od vrednosti ulaznih parametara već i od stanja objekta.
- o Privatne metode se ne mogu posebno testirati.
- o Pri testiranju metoda izvedenih klasa, testove za neke metode osnovne klase treba ponoviti.
- o Posebnu pažnju treba posvetiti testiranju svih metoda koje koriste polimorfne funkcije.
- o Apstraktne klase se ne mogu testirati nezavisno.
- o Obrada izuzetaka, takođe, u mnogome otežava testiranje jer se bacanjem izuzetka menja osnovni tok izvršenja programa-
- o U objektno-orijentisanim jezicima se vrlo često koristi konkurentno izvršavanja više metoda koje često medju sobom komuniciraju.

Ilustrativni primer

```
class Base {
public:
    int func() { return x + vfunc();}
    int func2(){ return x+5; }
    virtual int vfunc() { return 0; }
    Base(): x(10) {}
private: int x;
};

class Derived: public Base {
public:
    virtual int vfunc() { return 1; }
};

Base b;
Derived d;
```

- Funkcija **func()** daje različite rezultate ako se pozove u kontekstu klase Base, odnosno Derived, iako postoji jedinstvena implementacija
- **b.func()** daje rezultat 10
- **d.func()** daje rezultat 11

U svakoj od podklasa metod X se poziva u kontekstu operacija i atributa koji su definisani za tu podklasu.
Znači taj kontekst je različit u opštem slučaju, pa je potrebno X testirati u svim tim kontekstima.
Za metode za koje smo sigurni da im je kontekst definisan u nekoj klasi i da je nepromenljiv, nije potrebno testiranje u kontekstu podklasa (prethodni primer **func2**).

Primenljivost klasičnih tehnika jediničnog testiranja

- **Metode bele kutije:**
  Uzimajući u obzir prethodno rečeno, moguće je primeniti sve ranije opisane tehnike toka kontrole i podataka na metode klasa.
  U OO programiranju metodi se generalno implementiraju sa malo linija koda => napor za primenu tehnika bele kutije bolje je usmeriti na specifične OO tehnike testiranja klasa.
- **Metode crne kutije:**
  Ove tehnike su jednako upotrebljive kao i kod konvencionalnih sistema.
  Slučajevi upotrebe (use cases) kao i model stanja klase daju korisni ulaz za tehnike crne kutije.

Specifične tehnike OO testiranja na nivou klase:

- Slučajno testiranje - Pozivanje metoda klase u slučajno određenom redosledu.
- Particiono testiranje - Slično klasičnom metodu podele na klase ekvivalencije.

I. Slučajno testiranje klase

Procedura testiranja:

- **Identifikacija operacija** (metoda) primenljivih na klasu
- **Definisanje ograničenja** za njihovu primenu
- **Identifikacija minimalne sekvence testiranja** - Sekvenca operacija koja definiše minimalan "životni ciklus" objekta te klase
- **Generisanje većeg broja slučajnih** (ali validnih) **sekvenci testiranja** - Izvršavanje drugih (kompleksnijih) životnih ciklusa instanci te klase.

Primer: Bankarska aplikacija ima klasu account sa sledećim operacijama: **open**, **setup**, **deposit**, **withdraw**, **balance**, **summarize**, **creditLimit** i **close**.
Svaka od ovih operacija može se primeniti na **account**, uz određena ograničenja (npr. **account** mora da se otvori pre drugih operacija i zatvori posle svih drugih operacija) koja proizilaze iz prirode problema.

Čak i sa ograničenjima, moguće su mnoge različite sekvence operacija. Minimalna istorija ponašanja instance računa je:

open•setup•deposit•withdraw•close

Ovo je i minimalna test sekvenca za account. Opis dozvoljenih sekvenci:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close

Sekvence se mogu generisati na slučajan način. Na primer:

Test case r1: *open•setup•deposit•deposit•balance•summarize•withdraw•close*

Test case r2: *open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close*

Izvršavanjem ovih testova se isprobavaju različiti "životni ciklusi" instanci klase račun.

Broj mogućih permutacija operacionih sekvenci kod slučajnog testiranja može biti veoma veliki. Za poboljšanje efikasnosti testiranja koristi se particiono testiranje (podela životnih ciklusa na klase ekvivalencije).

II. Particiono testiranje klase

- Particiono testiranje redukuje broj test primera neophodnih da se ispita klasa na način sličan klasičnoj tehnici podele na klase ekvivalencije.
- Ulaz i izlaz se kategorizuju i potom se projektuju test primeri da pokriju te kategorije.

Tri načina podele u kategorije:

1. **Podela zasnovana na stanjima** kategorizuje operacije klase prema njihovom uticaju na promenu stanja klase.
   - Za klasu *account*, operacije koje menjaju stanje su **deposit** i **withdraw**, a operacije koje ne menjaju stanje uključuju **balance**, **summarize** i **creditLimit**.
   - Testovi se prave posebno za izvršavanje sekvenci operacija koje menjaju stanje, a posebno za operacije koje ga ne menjaju.
     Test case p1: open•setup•deposit•deposit•withdraw•withdraw•close
     Test case p2: open•setup•deposit•summarize•creditLimit•withdraw•close
   - P1 menja stanje, a P2 ne (izuzev što mora da uključi minimalnu sekvencu operacija).

2. **Podela zasnovana na atributima** kategorizuje operacije klasa na osnovu atributa (članova) klase koje koriste.
   - Za klasu *account*, atributi **_balance** i **_creditLimit** mogu se koristiti za definisanje particija. Za **_creditLimit** npr. operacije se dele u tri particije:
     (1) Operacije koje čitaju **_creditLimit**,
     (2) Operacije koje menjaju **_creditLimit** i
     (3) Operacije koje ne čitaju niti menjaju **_creditLimit**.
   - Test sekvence se projektuju za svaku particiju posebno.

3. **Podela zasnovana na kategorijama funkcija** particioniše operacije u odnosu na generičke funkcije koju obavljaju.

- Na primer, operacije u klasi account mogu se podeliti na inicijalizacione operacije (**open**, **setup**), računanja (**deposit**, **withdraw**), upite (**balance**, **summarize**, **creditLimit**) i završne operacije (**close**).

Integraciono testiranje OO softvera

- Strategija integracije OO softvera fokusira se na grupe klasa koje sarađuju ili komuniciraju međusobno.
- Klasični pristupi od vrha ka dnu ili od dna ka vrhu imaju malo smisla s obzirom da OO softver nema hijerarhijsku kontrolnu strukturu.
- Dodatno, u opštem slučaju nemoguće je dodavanje jedne po jedne operacije u klasu pri integraciji jer operacije interaguju sa drugim operacijama i atributima klase.

- **Strategije integracionog testiranja objektno-orijentisanog softvera:**
  - Integraciono slučajno testiranje
  - Integraciono particiono testiranje
  - Testiranje zasnovano na scenarijima
  - Testiranje zasnovano na modelu ponašanja

Koraci u generisanju slučajnih test primera za više klasa:

1. Za objekte svake klijentske klase, generisati seriju slučajnih test sekvenci sastavljenih od (lokalnih) operacija nad tom klasom. Neke od tih operacija će slati poruke drugim (serverskim) objektima.
2. Za svaku generisanu poruku, odrediti kolaborirajući serverski objekat i operaciju njegove klase koja se aktivira porukom.
3. Za svaku operaciju serverskog objekta, odrediti poruke koja ona emituje drugim objektima.
4. Za svaku od poruka, utvrditi sledeći nivo operacija koje su pozvane I uključiti ih u test sekvencu.

Primer: razmotrimo sekvence poruka klase **ATM** upućene klasi **banka**:

$$verifyAcct \bullet verifyPIN \bullet [[verifyPolicy \bullet withdrawReq] | d\ epositReq | acctInfoREQ]^n$$

Jedan slučajan test koji je u skladu sa gornjim opisom mogao bi biti:

$$r3 = verifyAcct \bullet verifyPIN \bullet depositReq$$

Razmotrimo kolaboracije objekata u okviru testa r3. **Banka** mora kolaborirati sa **ValidationInfo** da izvrši **verifyAcct** i **verifyPIN**. Banka mora kolaborirati sa account da izvrši **depositReq**. Dopunimo test t3 unosom ovih kolaboracija:

$r4 = verifyAcct_{Bank}[validAcct_{ValidationInfo}] \bullet verifyPIN_{Bank} \bullet [validPin_{ValidationInfo}] \bullet deposit\ Req \bullet [deposit_{account}]$

Integraciono particiono testiranje

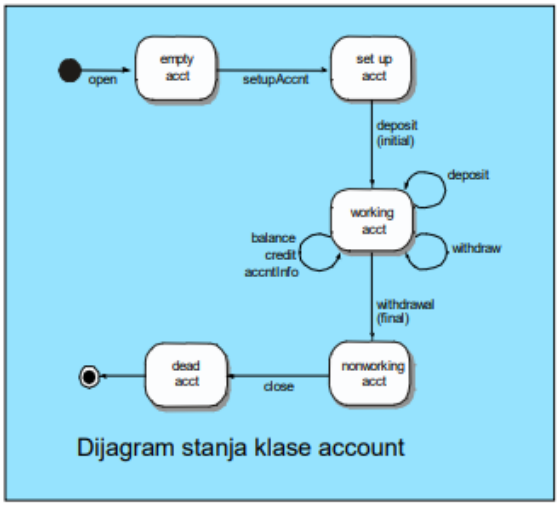
- Pristup particionom testiranju skupa klasa sličan je particionom testiranju pojedinačne klase.

- U prvoj varijanti, pojedinačna klase se particioniše kao pri jediničnom testiranju. Potom se test sekvenca proširi operacijama nad kolaborirajućim klasama koje se aktiviraju porukama iz posmatrane klase.
- Alternativa je da se podela izvrši u odnosu na kolaborirajuće klase. U primeru sa ATMom, klasa **banka** prima poruke od klasa **ATM** i **cashier**. Operacije klase **banka** mogu se podeliti na one koji opslužuju **ATM** i one koje opslužuju klasu **cashier**. Particioniranje zasnovano na stanjima može poslužiti da se dodatno profine particije.

Testiranje zasnovano na modelu ponašanja

- Dijagram stanja (DS) služi kao model dinamičkog ponašanja klase.
- Pomoću DS klase mogu se izvesti sekvence testova ponašanja klase i kolaborirajućih klasa.

- Inicijalni prelazi odvijaju se kroz stanja empty acct i setup acct.
- Najveći deo aktivnosti instanci klase dešava se u stanju working acct.
- Završno podizanje i zatvaranja izazivaju prelaz u stanja nonworking acct i dead acct states, respektivno.



Dijagram stanja klase account

- Testovi se projektuju da postignu pokrivanje svih stanja tj. sekvence operacija treba da izazovu prelaze kroz sva dozvoljena stanja:
  Test case s1: open•setupAccnt•deposit (initial)•withdraw(final)•close
- Ovo je minimalna sekvenca operacija koja pokriva sva stanja u primeru.
- U situacijama kada klasa kolaborira sa drugim klasama, prave se višestruki dijagrami stanja da bi se pratilo ponašanje celog sistema.
- Model stanja može da se obilazi na način "po širini". Ovo konkretno znači da se svaki test primer fokusira na izvršavanje jedne određene grane, te kada treba testirati još netestiranu granu, pravi se test primer koji obuhvata samo testirane grane i tu novu granu.

Primer: Objekat korišćenja kreditne kartice ima stanja: **undefined**, **defined**, **submitted**, **approved**.
Inicijalno stanje je **undefined** (nije još podneta kreditna kartica).
Posle čitanja kartice u čitaču, objekat ide u stanje a **defined**. To znači da su sada definisane vrednosti atributa **card number**, **expiration date**, i **bank_id**.
Objekat korišćenja kreditne kartice je u stanju **submitted** kada se šalje na autorizaciju i prelazi u stanje **approved** kada se primi potvrda o autorizaciji.

Ako se držimo pristupa "po širini", testovi neće izazivati prelaz u stanje **submitted** pre nego što se istestiraju prelazi u **undefined** i **defined**.

Dokumentovanje OO test primera

1) Svaki test primer treba da je jednoznačno identifikovan i eksplicitno pridružen klasi koja se testira.
2) Treba da bude navedena svrha testa
3) Navodi se lista koraka pri testiranju koja treba da sadrži:
   a. Lista stanja objekta koji se testira.
   b. Lista poruka i operacija koje će biti izvršene testom.
   c. Lista izuzetaka koji se mogu desiti tokom testiranja.
   d. Lista spoljnih uslova (tj. šta mora da se podesi u spoljnom okruženju softvera da bi test mogao pravilno da se sprovede)
   e. Dodatne informacije koje pomažu u razumevanju ili implementaciju/sprovođenje testa.

Testiranje Web aplikacija

Tipična troslojna arhitektura web sajta

- Web server, tj. prezentacioni sloj ili pezentacioni lejer – kreira vizuelni sadržaj koji će krajnji korisnik da vidi.
- Biznis lejer je centralni deo serverske aplikcije. U njemu je implementirana kompletna logika aplikacije. Bitne funkcionalnosti ovog lejera su:
  - **Obrada transakcija**
  - **Prepoznavanje korisnika**
  - **Validacija podataka**
  - **Prijavljivanje na aplikaciju**
- Data layer je zadužen za čuvanje i pretraživanje podataka.
  - Za skladištenje podataka se obično koriste relacione baze podataka.
  - Na ovom nivou se definiše interfejs preko koga biznis lejer komunicira sa bazom podataka.

Kritične tačke o kojima treba voditi računa pri planiranju testiranja web aplikacija

- **Veliki broj korisnika aplikcije**
  Koriste različite browsere, različite verzije istog browsera, različite operativne sisteme, različite uređaje za pristup Internetu...
- **Biznis okruženje**
  Naša aplikacija sarađuje sa drugim partnerskim aplikacijama.
- **Lokalitet korisnika**
  Korisnici aplikacije se mogu nalaziti u različitim zemljama pa treba obezbediti:
  Prevod na različite jezike,
  Usklađivanje časovnih zona,
  Konverziju valuta...
- **Test okruženje**

Za testiranje aplikcije treba obezbediti identičan duplikat hardverske podrške aplikaciji.
Sledi: testranje web aplikcija je veoma skupo

- **Zaštita**
  Svaki web sajt je izložen napadima hakera
- **Vreme testiranja**
  Web aplikacije najčešće imaju i informacionu ulogu. Zakasnela informacije prestaje da bude informacija.
  Sledi: Pažljivo treba birati šta će se sve testirati i kada testiranje treba prekinuti.

Testiranje Web aplikacije

- **2 faze:**
  - Testiranje svakog lejera (sloja) aplikacije ponaosob
  - Testiranje integrisanog sistema

I. Testiranje prezentacionog lejera

Najbitniji lejer sa aspekta korišćenja aplikacije. Ukoliko korisnik kada otvori stranu vidi greške u fontu, loš prevod ili neučitane delove sajta (animacije nisu mogle da se pokrenu ili slično), odustaće od korišćenja vašeg web sajta.
Šta sve treba testirati u ovom lejeru:

- Sadržaj web strana
- Arhitekturu web sajta (provera linkova, slika...)
- Korisničko okruženje (web browser, operativni sistem, konfiguracija...)

1) Testiranje sadržaja

- **Testiranje estetskih parametara:**
  Da li se korišćeni fontovi vide u svim bowserima korektno,
  Da li se korišćene boje vide dobro,
  Da li je rezolucija slika odgovarajuća...
- **Testiranje semantike sadržaja:**
  Da li su prevodi na različite jezike gramatički ispravni,
  Da li su korektno postavljena default-na podešavanja (pri otvaranju web strane sve kontrole treba da imaju neka unapred definisana podešavanja)...

2) Testiranje arhitekture Web sajta

- **Cilj** ovog testiranja je otkrivanje grešaka u navigaciji ili stukturi sajta kao što su:
  - Prekinuti linkovi,
  - Linkovi vode do pogrešnih strana,
  - Uključivanje pogrešnih slika...
- **Uzrok** ovakvih grešaka su:
  - Programer promeni ime fajla, link postaje nevalidan
  - Promeni se ime nokog grafičkog elementa ili premesti na drugu lokaciju...
- **Sprečavanje** ovakvih grešaka:

- Korišćenje automatskih build-era sajtova koji kada se promeni im fajla ili lokacija, automatski menjaju sve linkove na njega

Načini testiranje arhitekture Web sajta

- Korišćenjem alata za automatsko testiranje strukture web sajta
  - proverava se da li je svaki link povezan na neki fajl
  - Ne može se utvrditi proveriti i da li je to stvarno željeni fajl
- Manualno testiranje
  - Koriste se metode bele kutije – najčešće metoda prolaska kroz sve puteve

3) Testiranje korisničkog okruženja

- Poznato kao **browser-compatibility** testing
- Broj kombinacija browser-operativni sistem je veoma veliki
- Ako se dodaju i različite verzije istog *browser*-a, pa i različita podešavanja *browser*-a, broj test scenarija koje treba izvršiti eksponencijalno raste
- Testiranje korisničkog okruženja je obavezno ukoliko web sajt sadrži skripte koje se izvršavaju na klijentskoj strani.
- Ovom testiranju treba posvetiti posebnu pažnju ukoliko web sajt sadrži:
  ActiveX kontrole, JavaScript, VBScript, Java aplete...
- Ovo testiranje će se pojednostaviti ako se u zahtevima navede kojim web browserima je aplikacija namenjena.

II. Testiranje biznis lejera

- Cilj je otkrivanje grešaka u biznis logici.
- Za testiranje ovog lejera se mogu primeniti sve metode korišćene za testiranje **stand-alone** aplikacija (i metode bele i metode crne kutije).

Primer: testiranje biznis lejera u jednoj e-commerce aplikaciji

Testiranje takve aplikacije treba da obuhvati:

- **Testiranje performansi:** vreme odziva, propusnost sistema.
- **Validaciju podataka:** otkrivanje grešaka u podacima koji se dobijaju od kupca
- **Testiranje transakcija**: Procesiranje kreditnih kartica, Izračunavanje poreskih taksi,...

1. Testiranje performanci

- U funkcionalnoj specifikaciji aplikacije bi trebalo definisati i performanse koje se od nje očekuju.
- Pre testiranja treba proveriti da li hardverski elementi zadovoljavaju specificirane performanse (brzina internet veze, propusnost rooter-a, firewalla...)
- Testiranje performansi treba ponoviti u uslovima "stresa"

2. Validacija podataka

- Neki podaci koje korisnik unosi imaju standardni format, pa je moguće, donekle, proveriti njihovu validnost. Tu spadaju: Broj telefona, E-mail adrese, Broj kreditne kartice...

3. Testiranje transakcija

- U e-commerce sistemima obrada transakcija mora da se izvršava korektno u **100%** slučajeva.
- Obrada transakcije obuhvata:
  - Pretraživanje zaliha,
  - Prikupljanje proizvoda koje je kupac poručio,
  - Preračunavanje cene poručenih proizvoda, poreza na promet i procesiranje finansijskih transakcija,
  - Potvrda kupcu da je transakcija završena (obično emailom).
- **Kritična tačka: procesiranje finansijskih transakcija**
  Osim testiranje unutrašnjih procesa treba testirati i eksterne servise, npr. Validaciju kreditne kartice, Bankarske transakcije, Verifikaciju adrese kupca...
  Najosetljivija tačka je komunikacija sa finansijskim institucijama.
  Ovu komunikaciju **obavezno** treba testirati.

III. Testiranje *data layer*-a

Na ovom nivou treba testirati:

- Vreme odziva,
- Integritet podataka,
- Robusnost i mogućnost opravka (recoverability).

1) Vreme odziva

Vreme za koje se izvršavju osnovne SQL naredbe (INSERT, UPDATE, DELETE i SELECT).
Za ovo testiranje se koriste metode crne kutije.

2) Integritet podataka

Testiranje integriteta podataka treba da utvrdi da li su podaci zapaćeni u validnoj formi.
Npr: Uz datum i vreme treba pamtiti i časovnu zonu na koju se ono odnosi.
Za predstavljanje teksta treba koristiti internacionalni kod (UNICODE) koji povećava memoriju potrebnu za paćenje podataka.

3) Robusnost i mogućnost oporavka

- Cilj svake web aplikacije je da padovi sistema budu što ređi, a da oporavak nakon pada bude što kraći.
- Testirati koliko korisnika istovremeno DBMS može da opslužuje. Ako je to nedovoljno, treba praviti paralelno alternativni DB server koji se aktivira kada glavni bude zakrčen**.**
- Testirati da li DB server uspeva da se oporavi nakon pada i za koliko vremena. U tu svrhu izazvati namerni pad DB servera**.** Ukoliko server ne uspeva da se vrati u stanje pre pada, znači da BackUp podataka ne radi dovoljno često.