Topic 8 – Objektno orijentisana analiza(OOA)

Objektno orijentisana analiza se zasniva na ideji da modeluje problematičnu oblast kroz njene objekte i njihove interakcije.

Da bi se opisao problem potreban je opisni jezik, u ovom slučaju se koristi UML(Unified Modeling Language).

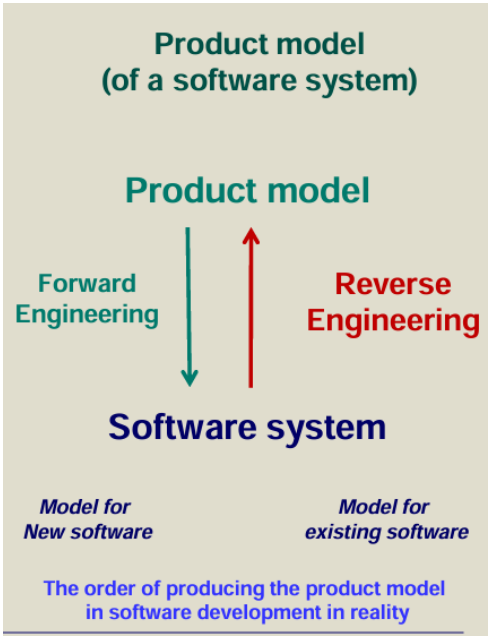
Objektno orijentisana analiza je metoda analize koja ispituje zahteve iz perspektive klasa i objekata koje nalazimo u rečniku domena problema.

OOA je jedna od metoda analize koja se koristi u prvoj fazi razvoja softvera(analize i definicije). Zapravo je metod modelovanja koji se koristi u praksi.
Osnova je objektno orjentisana softverska paradigma koja uzima objekte iz domena problema kako bi razvila zahteve za softver.

UML je grafički opisni jezik koji obuhvata mnogo dijagrama za opisivanje strukturalnih i ponašajnih osobina softvera. Neki od najvažnijih dijagrama su dijagrami klasa, dijagrami paketa, dijagrami stanja i dijagrami saradnje.

OOA modeli obuhvataju statične i dinamične modele za opisivanje strukture i ponašanja softvera. Dinamički modeli imaju za cilj da specificiraju semantiku statičkog modela, naročito semantiku klasa (životni ciklus objekta), korisničke slučajeve i metode klasa.
Statički model obuhvata use case pogled i class diagram
Dinamički model obuhvata pogled interakcije i state diagram pogled
Kompleksni sistemi moraju biti modularizovani pomoću dijagrama paketa.
Model proizvoda(produkta)

Mesto OOA u vodopadnom modelu je u fazi analize i definicije, tj u fazi definici u delu modela proizvoda(slajd 10).



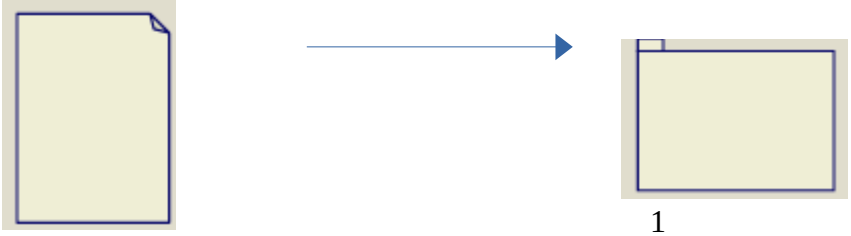
Model proizvoda softverskog sistema

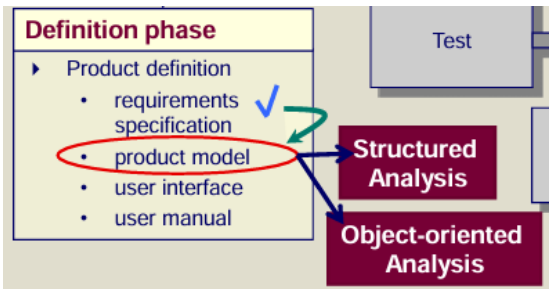
Softverski sistem ——apstrakcija——▶ model proizvoda

Modeli proizvoda preciznije definišu zahteve za softverski sistem nego tekstualni zahtevi, npr. korišćenjem osnovnih pojmova.
Jedan od pristupa produktnim modelima je strukturirana analiza i OOA.

Modeli proizvoda formalizuju tekstualne specifikacije zahteva koristeći osnovne koncepte.
Specifikacija zahteva je verbalna, neformalna i koristi prirodan jezik.
Sklonija je da bude neprecizna i da dodje do nesporazuma.
Product model je formalizovaniji što ga čini više preciznim.

Specifikacija zahteva primena osnovnih koncepata product model





Isti softverski sistem može biti opisan tekstualnim zahtevima kao i formalnijim modelom proizvoda.
Model proizvoda:
- Opisuje osnovne aspekte softverskog sistema
- Sastoji se od nekoliko pregleda softvera

Aspekti OOA su podeljeni prema različitim nivoima znanja i veština
Tu spadaju:

- Paradigme modela – Dekompozicija softversih sistema u objekte(statički/ dinamički model – različiti pogledi).
- Jezik za modeliranje – UML : kolekcija dijagrama
- Metodologija – Kako razviti model?
- Alati - Together, Rational Rose, Paradigm Plus

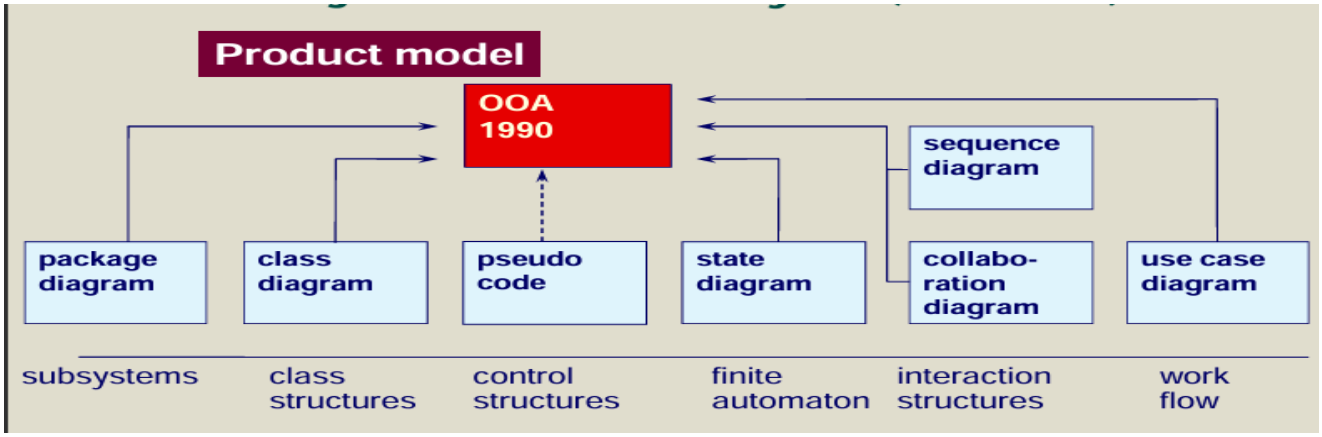
Tipovi UML dijagrama

Strukturalni dijagrami:

1. Dijagram klasa
2. Dijagram komponenti
3. Composite structure diagram
4. Dijagram rasporeda(deployment)
5. Dijagram objekata
6. Dijagram paketa
7. Dijagram profila

Dijagrami ponašanja

1. Dijagram aktivnosti
2. Dijagram komunikacija
3. Dijagram pregleda interakcije
4. Dijagram sekvenci
5. Dijagram stanja
6. Timing dijagram
7. Use Case dijagram



OO paradigma: osnovni pojmovi

Osnovna ideja – dekompozicija Softverskog sistema u objekte i klase(objekata).
Osnovni pojmovi:
Objekat- koristi se u svim fazama SW razvoja, ne samo u programiranju, nego i u fazi analize i definicije kao i u fazi dizajna.

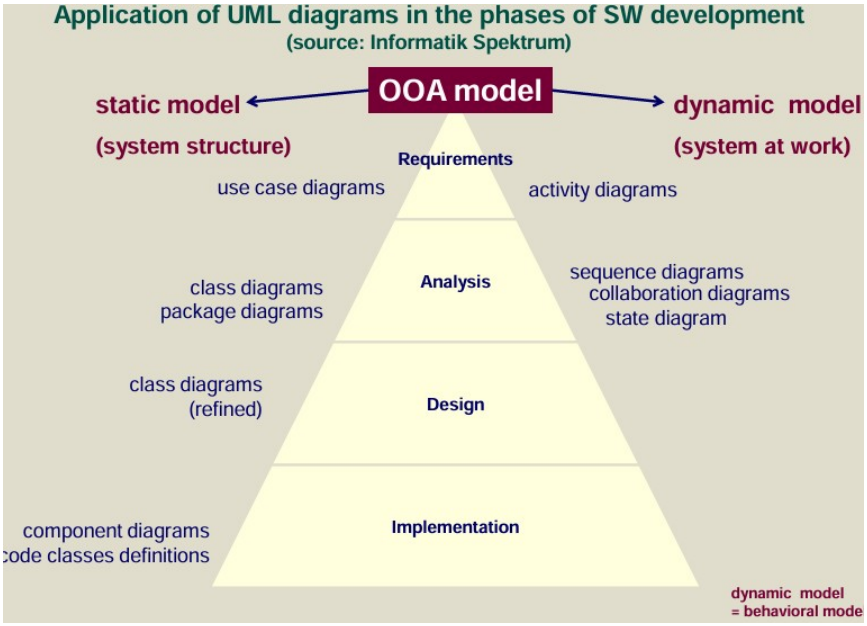
Šta je objekat?
Ovde imamo kombinaciju više definicija:

a) Objekat je kombinacija dva dela:
1. Data – podaci su stanja objekta koji se održava unutar njega
2. Operacije – svi mehanizmi za pristup i manipulaciju stanjem

b) Objekti su apstrakcije podataka sa interfejsom imenovanih operacija i skrivenim lokalnim stanjem. Objekat ima stanje, ponašanje i indetitet; struktura i ponašanje sličnih objekata definissani su u njihovoj zajedničkoj klasi.

c)Objekat se sastoji od neke privatne memorije i skupa operacija

Podmodeli u UML-u: statički i dinamički model



Statički model – Kako tumačiti dati statički OOA model?
Prvi osnovni dijagram statičkog modela je dijagram klasa koji predstavlja objektno-orijentisani pogled.
Notacije u class dijagramu:
Svaka klasa je sačinjena od naziva, atributa i operacija.
Veze između klasa(nisu sve spominjali na slajdovima samo ove dve i opisam sam uzela sa MIS-a):
1. Asocijacije - Predstavlja semantičku vezu između dve ili više klasa. Asocijacija može biti imenovana čime se opisuje njeno značenje.Za asocijaciju se definišu kardinaliteti.
2.Generalizacija koja služi za nasledjivanje - Hijerarhijska veza između nadređene i podređene klase. Kardinalnost preslikavanja je uvek jedan. Npr u programiranju kada jedna klasa nasleđuje sve atribute i operacije iz nadređene klase.
Kada postoji više veza asocijacija između klasa onda tim vezama dodeljujemo uloge.
Kardinalitet predstavlja broj objekate klase sa kojima je posmatrani objekat povezan.Kardinalitet može biti 1..1, 0..1, 0..*, 1..*

npr client *_____0..1 company gde * znači da kompanija može da ima više klijenata a 0..1 znači da klijnet može biti povezan sa 1 ili 0 kompanijama.

Statički model – Paketi

Zbog činjenice da dijagrami mogu postati prilično složeni pojavila se potreba da se pojedinačne jedinice okupe u opštiju zbog čega i postoji dijagram paketa za izgradnju podsistema.

Cilj: Skupiti komponente u veću jedinicu

UML: Paket skuplja elemente modela(npr klase) i druge dijagrame. Može da sadrži i pakete.
Izgled samog paketa
Veza između paketa je usmerena isprekidana strelica koja predstavlja zavisnost

Dinamički model– Stanjem orjentisan pogled

Jedan od načina za opis ponašanja sistema je sačinjen od stanjem orjentisanog pogleda na sistem.
Koncentrišemo se na osnovne pojmove konačnog automata (dijagram stanja) i dijagrama aktivnosti.
State oriented view obuhvata softver opisan pogledom na unutrašnja stanja tokom izvršavanja(runtime).

Dijagram stanja opisuje stanja automata kao što se učilo na predmetu.

Stanja automata cilj i primena:

Cilj: Modelovati ponašanje sistema, koje zavisi od unutrašnjih stanja u sistemu gde ta unutrašnja stanja zavise od prethodnog input-a.

Primena u OOA:
-Dinamički pogled na objekte klase(životni ciklus objekta – promena stanja objekata).
- specifikacija operacija klase(promena stanja tokom izvršavanja operacija)
-specifikacija use case-ova(menjanje stanja tokom interakcije korisnika i sistema).

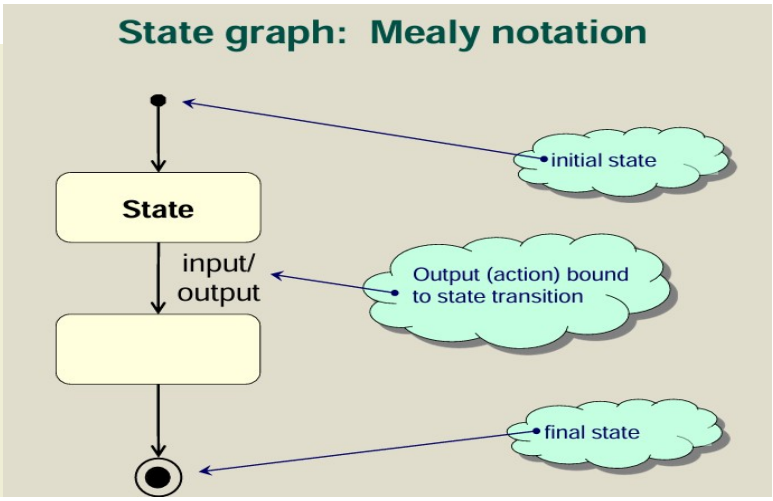
Autotmat stanja (konačni automat) definicija:
1. Skup stanja (Z) – konačan skup unutrašnjih stanja
2. Skup ulaza (X) – ulazi, signali, događaji
3. Skup izlaza (Y) – izlazi, akcije
4. Prelazi stanja – novo stanje zavisi od ulaza i trrenutnog stanja
5. Izlazno ponašanje: izlaz zavisi od ulaza i trenutnog stanja
slika 2 je čisto da se podsetimo kako izgleda dijagram

1st mathematical model: automaton = 5 tuple

- A = [X, Y, Z, f, g] with
  - X, Y, Z are finite sets (input/output/state alphabet)
  - f: X x Z →Z (state transition function)
  - g: X x Z →Y (output function)

2nd mathematical model: State transition graph

→ Mealy automaton, Moore automaton, Harel automaton (state charts)



Automati stanja se često koriste za modelovanje zivotnog ciklusa objekta.
Generalno nije neophodno kreirati automate za svaku klasu.
Dinamički model u OOA koristi automate stanja

Životni ciklus objekta i dijagram klasa
Samo operacije odgovarajuće klase su dozvoljene kao akcije i aktivnosti: operacije su jedini način da se pristupi klasi.

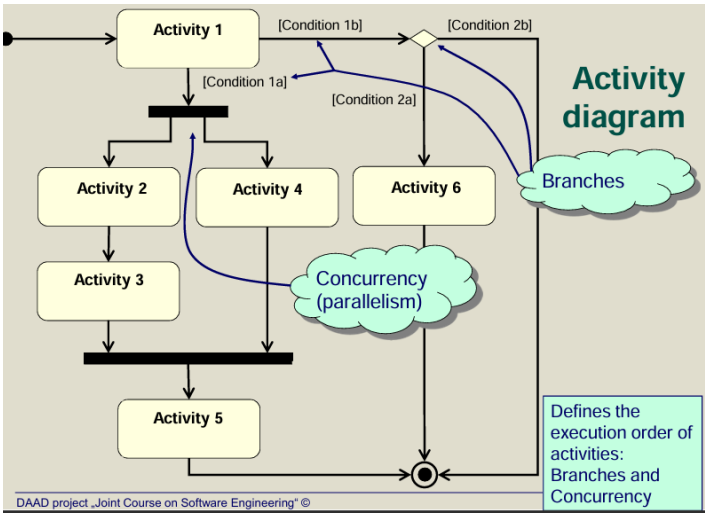
Dijagrami stanja
Događaji mogu da promene stanje objekta
Događaj može biti:

1. uslov koji se zadovoljio(postao true)
2. signa iz GUI-a
3. isteklo vreme(proteklo)
4. ili pojava određenog trenutka(vremena)

Poslednja dva slučaja se bave vremenskim događajima.

Dijagram aktivnosti
Klasifikovani kao alternativna notacija za mašine stanja (slika onog dijagrama)
Dijagrami aktivnosti su bolji i moćniji jer dozvoljavaju **paralelizam**.

1. Primena – opis slučajeva korišćenja i operacija klasa(predstavlja varijantu stanja automata)
2. Dijagram aktivnosti = Opis algoritma prema aktivnim stanjima i njihovim vezama.
3. Aktivno stanje = korak (aktivnost) tokom izvršenja algoritma ili poslovnog processa
4. Aktivno stanje(stanje aktivnosti) se napušta kada se aktivnost povezana sa njim završi
5. Može da se poredi sa starim dijagramima toka.



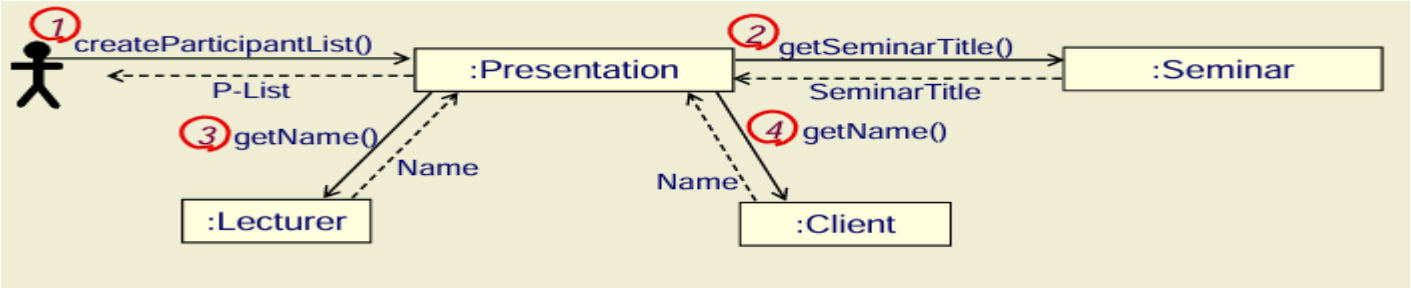
Dinamički model: scenariom baziran pogled(interakcije)

Predstavlja interakciju izmedju uloga(actors) i objekata
Dijagram sekvenci i saradnje su dva ekvivalneta izbora specifikacije.

Osnovni pojmovi su poruke i scenariji.

Poruka
Pošiljalac(sender-klijent) zahteva od primaoca(receiver – server) da odradi određenu uslugu slanjem prouke odnosno pozivom operacije primaoca.

Primer

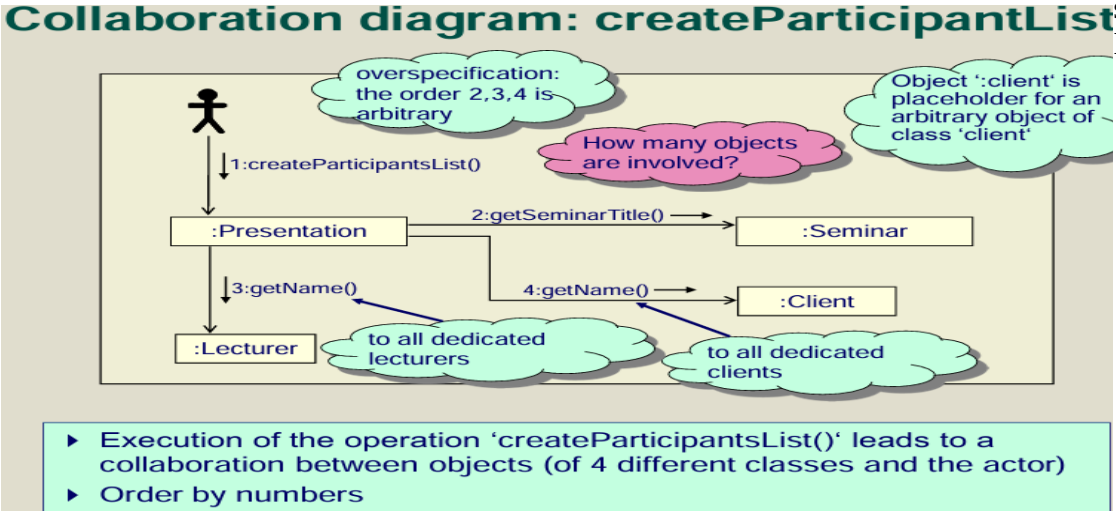


Scenario – sekvenca poruka gde je definisan redosled.
Use case može biti dokumentovan kolekcijom scenarija
Dijagrami interakcija su saradnjei sekvence (ako bude pitanje)

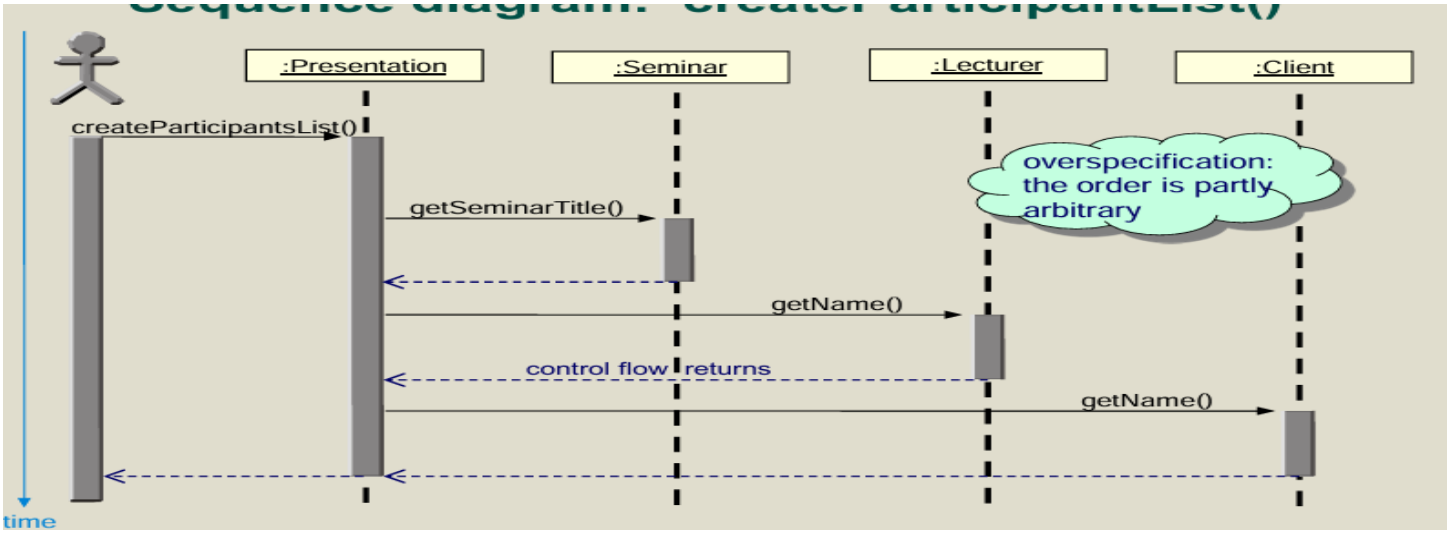
Dijagram saradnje

Osnovni koncepti prikaza zasnovanog na scenariju: opisivanje sekvenci poruka između aktera i objekata sistema

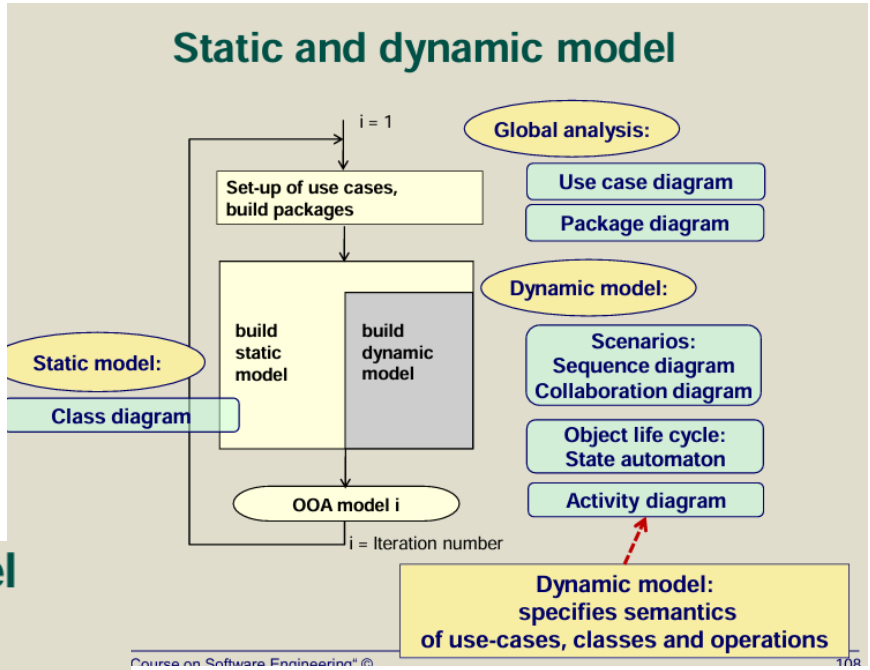
Izgled

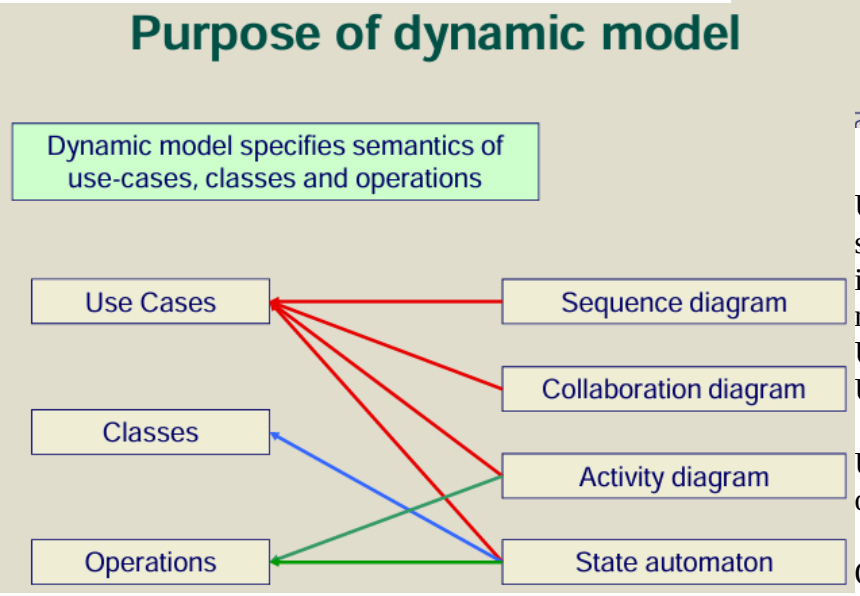


Dijagram sekvenci redosled je po vremenskoj osi. Više je grafički od saradnje i dalje je u delu tekstualan al i semiformal-formalizovan



Glavna svrha dinamičkog modela je da precizira semantiku statičkog modela. To znači da moraju biti opisani slučajevi korišćenja, klase i njihove operacije - kao deo statičkog modela - kroz njihovo ponašanje na dijagramima dinamičkog modela.





Use case nisu samo alat za definisanje zahteva sistema. Takođe utiču na njegov dizajn, implementaciju i testiranje; to jest, utiču na proces razvoja.
USDP = object-oriented software development with UML .

Use case ne mora biti povezan sa OO. Oni samo opisuju esencijalnu funkcionalnost.

Opisi slučajeva korišćenja idikuju na moguće objete u oblasti problema

Sažetak lekcije.
Objektno-orijentisana analiza (OOA) je model proizvoda
• koji je primer opšteg modelovanja
• za formalno specificiranje problema na osnovu objekata problem oblasti
• koji opisuje zahteve za softverom preciznije od tekstualnih specifikacija
• treba da se razvije na početku razvojnog procesa.

OOA model se sastoji od dva podmodela: statičkog i dinamičkog modela. Cilj dinamičkog modela je da definiše semantiku statičkog modela.
UML je grafički jezik specifikacije za opisivanje OOA modela. UML obuhvata nekoliko tipova dijagrama za različite poglede na softver.
Dijagrami za statički model: dijagram klasa, use case, dijagram paketa
Dijagrami za dinamički model: dijagram stanja, dijagram aktivnosti, dijagram saradnje i dijagram sekvenc
Dijagram stanja: opisuje životni ciklus objekta.
Dijagram aktivnosti: opisuje algoritme.
Dijagram saradnje/sekvenci: opisuje razmenu poruka između aktera i objekata.

Topic 12 – Uvod u testiranje softvera

Testiranje je dinamička analiza kojom se proverava ponašanje softvera.
Metrike su statička analiza kojom se meri kvalitet softvera.
Razvoj visokokvalitetnih softvera se radi testiranjem, metrikama, ponovnim pregledom (provera dokumentacije kritičkim čitanjem)i upravljanjem projetima(razvojni proces, planiranje npr koliko će koštati obradjeno topic 3).

Problem vodopadnog modela je taj što testiranje dolazi samo nakon implementacije, što je prekasno.
V model je bolji jer u svakom koraku mi imamo odgovarajuće testiranje, kroz ceo razvoj(slajd 17 za sliku).

Testirati aktivnosti prema zadacima kao što su
1. Određivanje test slučajeva
2. Izvršavanje testa
3. Evaluacija testa
4. Praćenje testa
5. Dokumentacija testa
6. Odabir alata za testiranje
***Glavni problemi u struci**

Pitanja povezana za testiranjem softvera

U oblasti *test menagmenta*:
1. Ko je odgovoran za testiranje softvera?
2. Da li je korisno da imamo stručnjaka za testiranje?
3. Da li je korisno imati spoljne savetnike?
4. Šta je sa testiranjem u drugim inženjerskim oblastima?

U oblasti *podrške alata*
1. Kako smanjiti napor testiranja?
2. Kako podržati testiranje softvera: alati za testiranje?
3. Da li postoji „najbolji“ softverski alat ili postoji više alata za različite aktivnosti testiranja?

U oblasti *modela testiranja*
1. Koje su podoblasti testiranja softvera?
2. Modeli softverskog testiranja u uopštenom softverskom razvoju
3. Da li možemo testirati softver samo ako je softver završen?
4. Koje osobine softvera možemo testirati: tačnost, efikasnost, ...?

U oblasti *kvalitet testiranja*
1. Koji su kriterijumi kvaliteta testiranja softvera?
2. Da li možemo meriti kvalitet testiranja?
3. Kako možemo dokazati da smo dobro testirali?

U oblasti *pristupa testu*
1. Da li postoje različiti pristupi testiranju u zavisnosti od aplikacije?

Kada biramo alate često se pitamo koji je najbolji, najoptimalniji po ceni/performansu, koliko je lako za korišćenje, koju test aktivnost podržava, koju vrstu softvra treba testirati?

Testiranje u razvoju softvera pokrivaju 50% cene i više(kako nzm na slici piše 8% unit test i 7% integration test pokrivaju testovi al aj)

XCTL kao primer

Problemi kod XCTL projekta

1. Testiranje je oduzelo previše vremena jer je bilo oko 70k LOC, puno fajlova sa puno klasa, tipova,funkicija
   (Bolje naći što pre greške tako što će se češće testirati)
2. Određivanje test slučajeva nije bilo sistematski.
   (Mnogi developeri doprinose odreeđivanju test slučajeva svojom intuicijom)
3. Da li smo našli sve bitne test slučajeve.
   (Proveriti test slučajeve za potpunost)

Alati koji su se koristili za različit problem

1.ATOS – Automatsko regresno testiranje (ponoviti test nakon modifikacije)
2 TESTONA – Podrška za određivanje test slučajeva
3. SOTA - Proverite kvalitet test slučajeva: izmerite stepen pokrivenosti

Na ovaj način ručno testiranje koje je trajalo 2 dana sada traje sat vremena.

Zašto je testiranje bitno?

Naglašava se da uzmia dosta novca i dosta ljudi.

1. Ekonomski pritisak – kao primer dat je pad boeing 737 i to dva puta jer je bio problem sa softverom.
-Situacija kojom se pravilno rukovalo.
-Senzori mogu da pruže pogrešne podatke o statusu i softver mora to da bude u stanju da ih obradi, tj. da uzme u obzir da ti podaci mogu biti pogrešni.
-Međutim, softver je pogrešno verovao podacima senzora i pomerio avion u pogrešnom pravcu (nadole) a pilotima nije bilo dozvoljeno da ručno prekidaju.

2. **Zahtevi za sisteme od kritične bezbednosti**
- U mnogim oblastima koje su kritične za bezbednost postoje prilično strogi zahtevi i standardi za testiranje (npr. medicinski uređaji, ugrađeni sistemi u avionima):
-Programeri moraju da dokažu da su pravilno testirali (inače: proizvod nije odobren / prihvaćen).
-primer DO-248B: MC/DC test kriterijum je potreban za testiranje u najvišoj bezbednosnoj klasi (Nivo A) u softverskim sistemima za avione (FAA = Federalna uprava za avijaciju), 2001.
-Kako dokazati da je dobro testiran?

Test podaci adekvatni/kompletni: ispuniti propisani kriterijum npr. DO-248B: MC/DC testni kriterijum / validacija kompajlera

Testna dokumentacija: test podaci i uspeh testiranja

U sustini bitno je testiranje zbog:
-Ekonomski razlog: to zahteva vreme i novac
- Pogrešan softver može uništiti kompaniju
- Postoje strogi zahtevi za softver koji je kritičan za bezbednost

Testiranje nije jedinstveno samo za softver jer procesni modeli nisu izmišljeni od stranje softverskog inženjerstva

Bilo je testiranje airbus, testiranje kofera.

Ko je zadužen-odgovoran za testiranje softvera?

Pre svega sam programer jer bi trebao da ima osnovno znanje o testiranju softvera.

Prema SWEBOK -u
Testiranje softvera obuhvata
1. Osnove tesitranja softvera(terminologiju vezanu za testiranje, ključne probleme, vezu testiranja sa drugim aktivnostima)
2. Nivoe testiranja (cilj testa, ciljevi samog testiranja)
3. Tehnike testiranja(kodom bazirane, greskom bazirane, odabir i kombinovanje tehnika, tehnike baziranje na intuiciji i iskustvu soft inženjera.)
4. Mere vezane za test (evaluacija programa pod testom, evaluacija preformansa testa)
5. Test proces (praktična razmatranja, test aktivnosti)
6. alati za testiranje softvera(podrška za alat, kategorije alata)

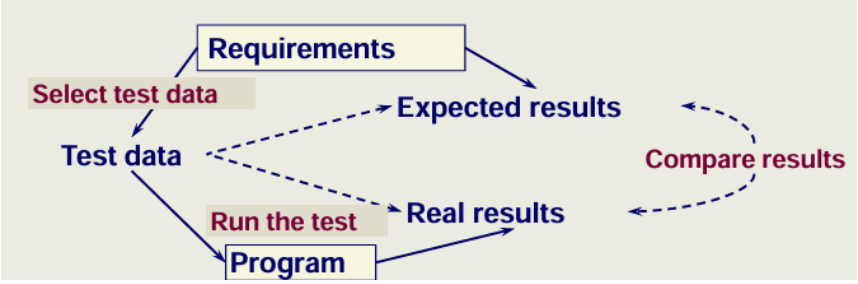
Pored samog programera, specijalne grupe za testiranje softvera u kompanijama su odgovorne za testiranje, sami stručnjaci u oblasti testiranja softvera

Postoji i sertifikat za softver testere koji je priznat svuda ISTBQ(International Software Testing Qualifications Board)

Pored specijalnih grupa i samog programera, odgovorni su i spoljni savetnici.
Npr. SQS nudi usluge za testiranje softvera za druge kompanije(„pozajme“ radniika drugim kompanijama)

b) Testiranje modela

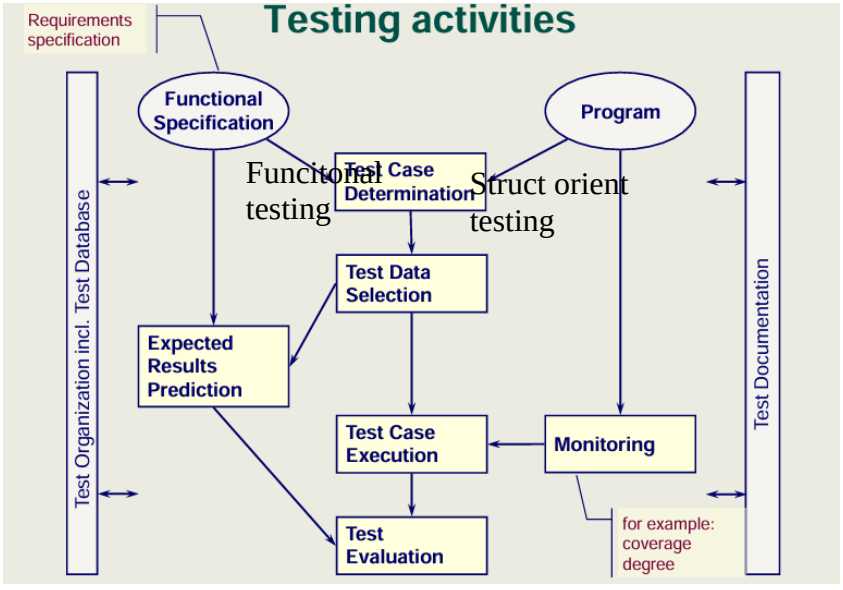
Šta znači testirati?
Izabrati podatke za test
Primeniti program nad test podacima i evaluirati rezultate.



Problemi tokom testiranja:
1. problem izbora(kako odabrati test podatke, logički problem jer test podaci moraju biti adekvatni, ne premali ali ni da ih je previše)
2. problem evaluacije(kako primeniti program na test podatke i uporediti rezultate, organizacioni problem, zahteva dosta vremena i sklono je greškama).

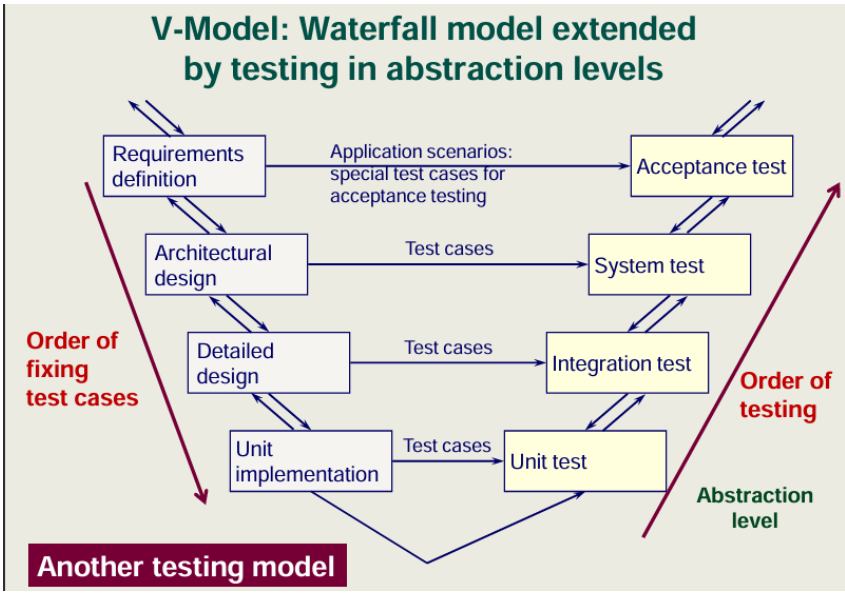
U vodopadnom modelu imamo testiranje koje obuhvata detaljno testiranje aktivnosti

Slika prikazuje activity model.



Testiranje kod V modela

Ono što je bitno je da se u svakom nivou apstrakcije primenjuje prethodna slika testiranja aktivnosti što je veza izmedju ova dva modela.



Testiranje nefunkcionalnih zahteva

(sve pre ovoga su bili funkcionalni zahtevi poput tačnosti i slicno iz prethodnih lekcija)

1.Efikasnost J. Wegener sa evolucionarnim testiranjem ponaašanja vremena u real‘time sistemima.
Npr Efikasnost u runtime je koliko vremena treba da se otvori airbag u autu.

2. Skalabilnost (ponašanje softvera pod opterećenjem, dobra sklabilnost je kada je efikasnost proporcionalna veličini opterećenja nije eksponencijalna)

Npr kad ima puno korisnika websit.

Testiranje opterećenja za skalabilnost bi obuhvatalo davanje velikih ulaza kojima se rukuje bez (linearnih)gubitaka efikasnosti, stim da se samo linerano povecava vreme/podaci.

c) Klasifikacija tehnika

Klasifikacija – način odabira test slučajeva(case test determination sa slike)
Pronalaženje odgovarajućih test slučajeva je odlučujući logički problem u testiranju.

Klase tehnika testiranja:
1.Strukturno orjentisano(control flow orjentisani, data flow orjentisano)(predstavlja white box test gde programska struktura odlučuje o nalaženju testova)
2.Funkcionalna(black box test- funkcionalni zahtevi odlučuju o nalaženju testova)
3. Nasumični testovi(velika količina podataka)
4.Analiza graničnih vrednosti(Metod orijentisan na greške: fokus na izuzetne situacije, granice i limita)

Regresiono testiranje

Problem: celoukupno testiranje posle svake modifikacije

Testiranje regresije:

-Testiranje da nakon modifikacija nema novih grešaka (nema koraka unazad = nema regresije)
-Isti test slučajevi pre i nakon modifikacija
-Podrška alata: Automatsko pokretanje i dokumentacija uspeha i neuspeha

Sa slike aktivnosti testiranja, ATOS obuhvata aktivnosti kao što su:
1.Test execution
2.Test evaluation
3.Test organization ukljucujući test baze
4. Test documentation

TOPIC 13 – Funkcionalno testiranje

Glavna tačka:
• Izvesti testne slučajeve iz specifikacije softvera (specifikacija zahteva)
• Program je nepoznat (blackbox testing)

a) **Priručnik za klase zadataka**

Bazirano na:
-Intuiciji i iskustvu softverskog inženjera
-Prirodi primene (klase zadataka)

Prema SWEBOK- u u okviru Test tehnika:
-3.1 Tehnika bazirana na intuiciji i iskustvu(ovo je mesto za priručnik)
-3.5 Korišćenjem bazirana tehnika preko interakcije putem GUI (snimanje i ponavljanje preko alata ATOS)
-3.7 Tehnike bazirane prema prirodi primene (Kompajler: skup za verifikaciju (izbor test slučajeva orijentisanih na sintaksu))

Izvlačenje test slučajeva iz posebnih klasa zadataka(pretraga):
Primer
Klasa zadatka: Pretraga elementa u nekoj sekvenci
Pretraga: pronaći ključni element u sekvenci elementata(nepoznat je algoritam pretrage, da je binarna pretraga, sekvencijalna…)
Pozitinvi ishod: vraćanje dve vrednosti- index i found = true
Negativan ishod: vraćanje vrednosti indexa=-1 i found false
Treba imati u vidu cilj funkcionalnog testiranja – izvlačenje test slučajeva iz specifikacije; program nije poznat.

Semantičke smernice u klasi zadataka (pretraga sekvence kao primer, liste i nizovi)
1. Testirati softver sa sekvencom ako ima samo jedan element(ako je moguće i bez elemenata)
Jer često programeri zaborave na specijalne slučajeve kao što je 1 element ili prazno.
2. Sekvence različitih dužina bi trebalo da se koriste u različitim testovima
3.Kreirati test slučajeve u kojima je prvi, poslednji i srednji element onaj koji se traži
4.Element koji se traži bi trebao i ne bi trebao da bude u sekvenci.

Ove smernice mogu da podrže program kao blackbox.
Problemi sa smernicama
Zaključak:
1.Tester treba da ima iskustva sa ovakvom klasom zadataka
2.Šta ako ima nova klasa zadataka bez definisanih pravila.
3. Takođe, kreirani su testovi bez formulisanja povezanih pravila (sortiranost sekvence igra ulogu)
4.Opšti postupak za svaki problem je poželjan
?

Metoda klasifikacionog stabla

Primer je indetifikacija slike
Computer vision system
Zadatak: Prepoznati i klasifikovati veličinu građevinskih blokova. Svaki građevinski blok može imati specijalne boje, oblik i veličinu.

Stvarni skup podataka: - Potencijalno beskonačan - Nejasno

Testiranje može otkriti greške, ali ne može dokazati odsustvo grešaka (beskonačan skup ulaza).

Osnovna ideja: napraviti konačan broj klasa

Osnovni princip: Klasifikacija prostora ulaznih podataka
Klasifikacija: Dekompozicija u ekvivalente klase (particiono testiranje)
Hipoteza particionog testiranja
1. Prostor ulaznih podataka može biti organizovan na končan skup klasa
2. Svi podaci u klasi se ponašaju na isti način(testiranje jednog prestavnika iz svakog podskupa je dovoljno)
3. Npr ako imamo prostor podeljen na šest klasa, dovoljno je 6 ulaznih podataka

Kvalitet izbora ulaznih podataka nije uvek primetan.
Rešenje kako da sve klase budu pokrivene jednako je metod klasifikacionog stabla

Metod klasifikacionog stabla

Osnovni princip = Klasifikacija + izbor test slučajeva
Koraci:
1. Klasifikacija prostora input podataka – Klasifikaciono stablo
    1.1 Ustanoviti bitne **aspekte** za ulazne podatke test objekata
    1.2 Za svaki aspekt odrediti **klase**
    1.3 Hijerarhijska struktura aspekata i klasa (struktura stabla klasifikacije)
2. odrediti test slučajeve

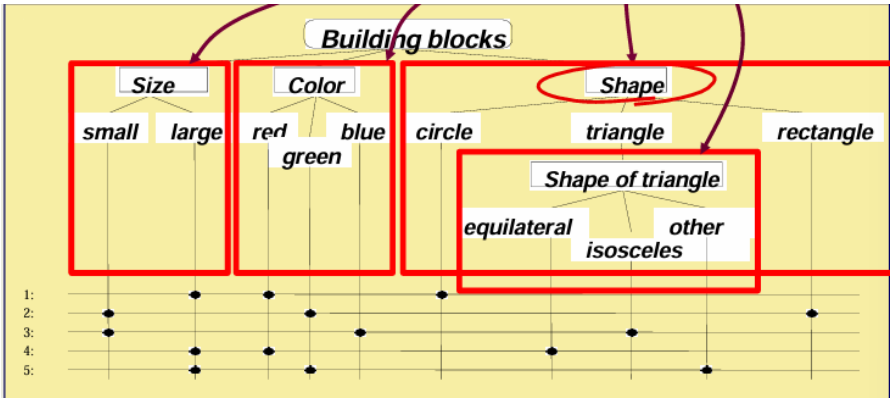
Za naš primer indentifikacije slike, koraci su primenjeni ovako:

1. Korak – konstruisanje klasifikacionog stabla
Zadatak: Prepoznati i klasifikovati veličinu blokova, svaki blok može imati specijalnu boju, oblik, veličinu(npr velika crvena kocka)
1.1 Aspekti - karakteristične osobine ulaznih podataka
Asspekt → postaje klasifikacija(boja, velicina, oblik)
Klasifikacija : disjunktna dekompozicija skupa
1.2 Klasa – za svaki aspect
Klase:
Veličina: mala, velika
Boja: Crvena, plava, zelena
Oblik: krug, trougao, pravougaonik (osnovni oblik, ne 3D)
1.3 Hijerarhija – Klase mogu biti dodatno klasifikovane (npr klase aspekta trougao: jednakostranicni, jednakokraki)

Klasifikaciono stablo pokriva sve aspekte(klasifikacije) i klase
Aspekt = Klasifikaciona klasa

Klasifikaciono stablo obuhvata više klasifikacija istog prostora ulaznih podataka.

Sve crveno je neka vrsta klasifikacije



2.Korak primera – izvlačenje test slučajeva iz klasifikacionog stabla

Selekcija test slučajeva: kombinacija klasa iz različitih klasifikacija
Sa slike svaka linija označena bojom je jedan test slučaj
1.veliki crveni krug…
Iz ovog vidimo da su isti ulazni podaci dekomponavani na 3 klasifikacije (veličina boja oblik)

Kada je izbor test slučajeva u pitanju gledamo **minimalni kriterijum (u svakom test slučaju, svaka klasa bi trebala da bude predstavljena bar jednom)**

Minimalan broj test slučajeva zavisi od broja klasa u najvećoj klasifikaciji(što je oblik imamo krug, pravougaonik i tri vrste trouglova, što je 5 testova jer je 5 disjunktnih klasa).

Maximalni broj slučajeva 2*3*5(broj klasa ispod klasifikacije), zbog činjenice da se svaka klasa iz klasifikacije može kombinovatu sa svakom klasom iz drugih klasifikacija.

Problem sa aspektima je da li su svi dobro odabrani, treba obraditi pažnju i na specifikaciju zahteva, šta sam kupac želi.

Dati aspekti na slici nisu dovoljni, mogu se izvući još korisnih, npr ugao bloka, razmak izmedju blokova, brzina trake…

Koji aspekti su važni zavise od iskustva testera, primene, specifikacije zahteva. Nikada nije što više aspekata i klasa moguće da se naprave.

Sličan primer ovome je driving assistant koji određuje distancu izmedju objekata, npr kolika je udaljenost(velika, mala) , boja auta, oblika auta pa pod njim i tip…, pored ovih aspekata bitni su i dodatni kao što je osvetljenje, da li se vozi danju nocu, kvalitet puta i njegova boja, brzina auta...

Tipični problemi početnika

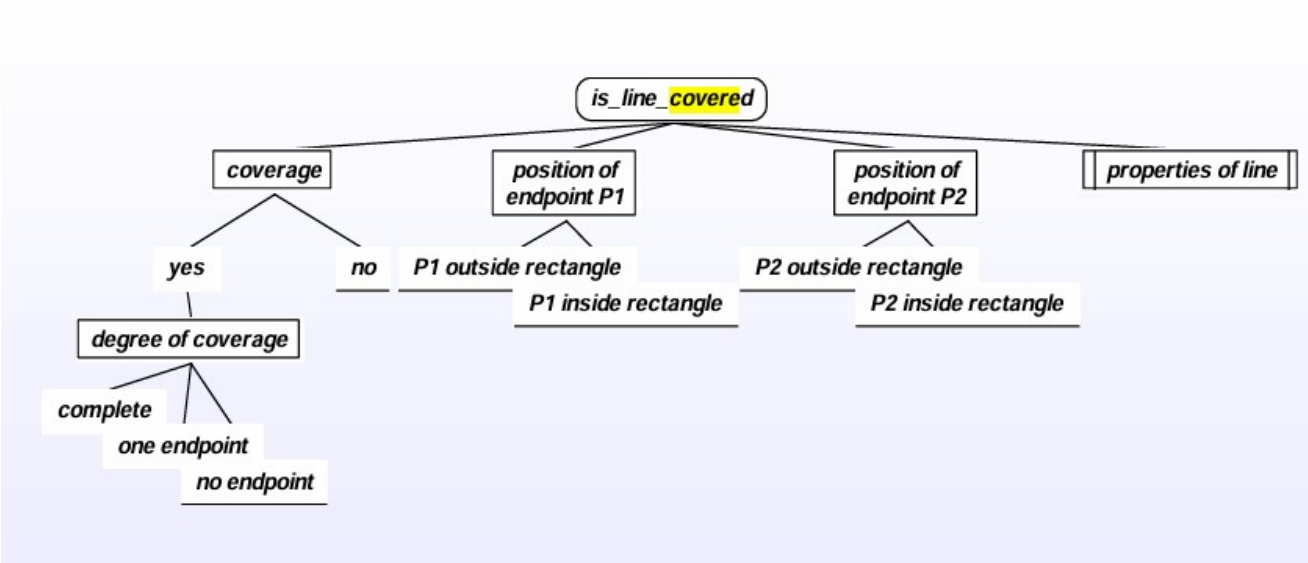
Izlaz(output) je uključen unutar klasifikacionog stabla
Mesto klasifikacionog stabla prema SWEBOK-u je unutar 3.2 Ulazom bazirane tehnike(input domain – based) i 3.4 greškom bazrane tehnike(ovo je i analiza graničnih vrednosti)

TESTONA

Alat koji se koristi kao podrška za metod klasifikacionog stabla
Ovde je dota slajdova kako koristiti alat(kontam da to ne treba)

Kompleksan primer za CT method
(CT je valjda classification tree)

Javlja se problem za testova pokrivanja
Primer:
Zadatak:
-Testirati da li je linija prekrivena(barem jednom tačkom) pravugaonikom
-Aspekti su: prekrivanje, tip prekrivanja(kompletno, jedna strana, u sredini)…
Prekrivanje:da, ne
Ostatak na slici



Sad ovde može da se dodaju još neki aspekti al ovo je baza

Topic 14
Testiranje orijentisano na strukturu

Ova vrsta testiranja obuhvata to da programski kod određuje test slučajeve.
Npr imamo kod iz kog ide control flow graf (grafička reprezentaci sintaksne strukture koda) iz kog dobijam test slučajeve.

Poređenje funkcionalnog testiranja i testiranja orjentisano na strukturu

Funkcionalno testiranje:

1. bolji logički izbor
2. test slučajevi izvučeni iz zahteva(iz logičkog opisa problema)
3. test slučajevi ne zavise od programskog koda koji može da ima greške..
4. .. ili je možda nedovršen(fali mu funkcionalnost)

Testiranje orjentiano na strukturu

1. loše jer uzima test slučajeve iz programa koji može biti pogrešan ili nedovršen
2. više se preferira u industrijskim aplikacijama
3. moguće je meriti kvalitet test slučajeva
4. dokaz da su test slučaajevi potpuni je lakši **(pokrivanje izvornog koda)**

Ova vrsta testiranja (SO) je **white box test:** struktura programa odlučuje
Obuhvata control flow oriented i data oriented

Control flow:(struktura iskaz(statement))
1. Statement(node) coverege(pokrivanje čvorova)
2.pokrivanje grana
3. pokrivanje putanje
4. Test granica-unutar staza (Boundary-interior path test)
5. Pokrivanje uslova (jednostavno, minimalno više, više)

Control flow se koristi u praksi iz razloga da se kvalitet test podataka može izmeriti(postoje alati)

Data flow – Pored toga: Ponašanje transformacije podataka izraza, bolje je od control flow, moćnije al komplikovanije
1. Def/Uses kriterijumi
    1.1 All defs – kriterijumi
    1.2 All uses
    1.3 All du-paths

Control-flow oriented tehnike

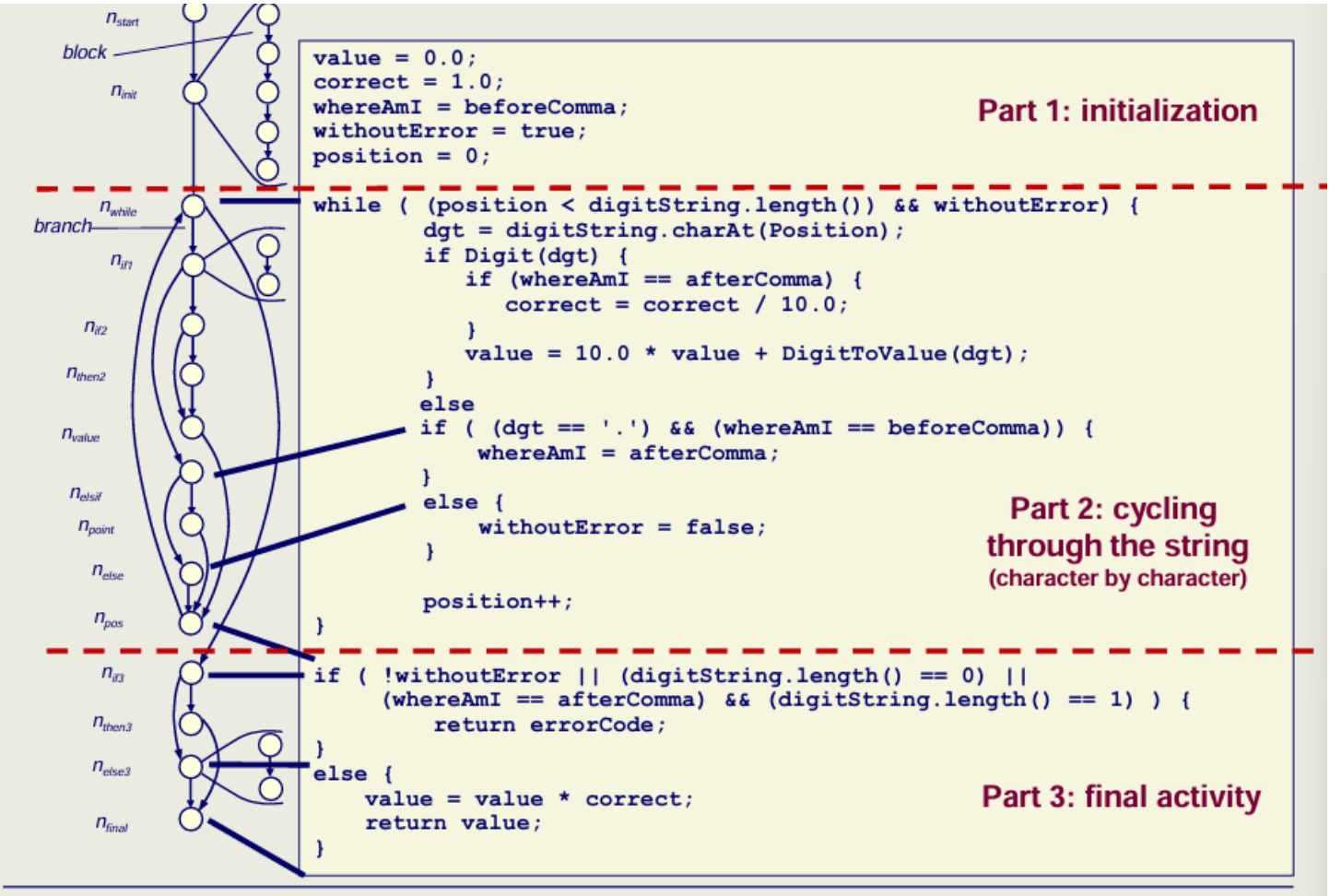
Ideja: pomoću odabranih test slučajeva možemo sistematski prolaziti kroz program
Nekoliko varijanti (za sistematičnost):
Pokriće iskaza, pokriće grana ... (vidi ispod)

Struktura iskaza- Graf kontrolnog toka (CFG)
Čvorovi: niz iskaza(statements) ili izraza(expressions)
Granice: moguće veze između iskaza

Primer koji se provlači kroz celu lekciju:
Leksička analiza ulaznog stringa(deo kompajlera)

Prebacivanje stringa u floating point broj(realan broj).
Proveriti da li je sam string realan broj(sa decimalnom tačkom, bez eksponenta) i pretvoriti ga u broj.

String „1.34“ i da li je „1.A5“ legalan floating point
Pozitivan ishod je pretvaranje stringa u broj npr „1.34“ u 1.34
Negativan ishod je error poruka npr errorCode



Primer grafa i koda dat na slici
imamo blok koji redukovan u čvor n init jer predstavlja inicijalizaciju promenljivih i nije neophodno za svaku promenljivu da imamo po čvor jer se sve na isti način inicijalizuju. Sekvencijalno se izvode iskazi (Graph reduction)

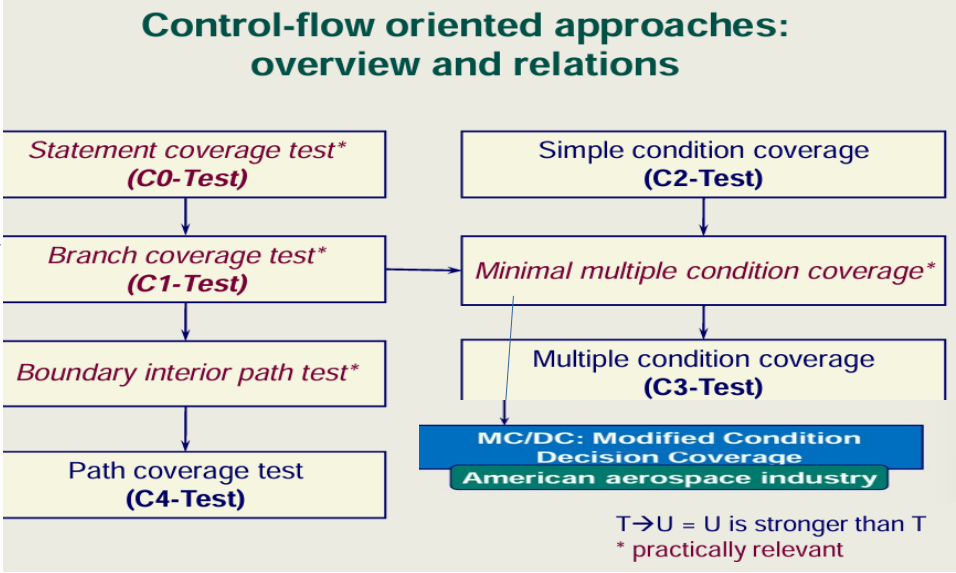
Pristup orijentisan na kontrolni tok: praktično relevantne metode

Pokriće iskaza(statement coverage): svaki iskaz će biti izvršen bar jednom od strane skupa svih test slučajeva (tj. da proće kroz sve čvorove CFG-a)

Pokrivenost grane: proći kroz sve ivice(edges) CFG-a ->jači uslov

Minimalna višestruka pokrivenost uslova(Minimal multiple condition coverage): Struktura uslova se uzima u obzir (logičke veze). Svaki elementarni uslov mora biti evaluiran od strane test slučajeva bar jednom kao TAČNO i jednom kao NETAČNO.

Test granične unutrašnje putanje(Boundary interior path test): Ciklus se sistematski testira (najmanje jednom i dvaput prolazi kroz telo ciklusa)



Pokriće iskaza(Statement coverage)

Zahtev: Svaki čvor u CFG mora biti posećen primenom svih test slučajeva, tj. svaki iskaz + svaki izraz(expression) koji se izvršava

Zaključak:

- Neizvršen kod je ispitan → posledica: proširenje test slučajeva
- U slučaju neizvršivog(„mrtvog“) koda

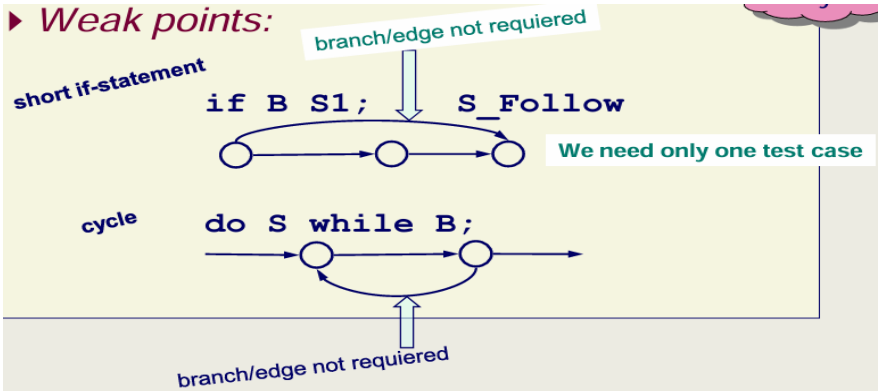
Cstatement = 100% nije moguće (C je mera pokrivenosti)

Stopa(eng.Rate): 18% svih grešaka pronađeno(iskustvo) ovim načinom pokrivanja

Problemi sa pokrićem iskaza

Imamo da je pokrivenost iskaza ispunjena, ali ne možemo biti zadovoljni sa test slučajevima

Problemi
1.Slabe tačke
Ako imamo jedan slučaj koji će proći kroz svaki čvor super, al ne znači da će proći i proveriti svaku granu što je loše i zato imamo i branch coverage.



Test pokrivanjem grana
(Branch coverage test)

Zahtev: Sve ivice(grane) unutar CFG moraju biti posećene, pređene primenom svih test slučajeva

Zaključak: Na tačkama odluke(decision points): sve grane su pređene -> važan praktični kriterijum

Stopa(Procenat): 35% svih pronađenih grešaka ovim testom pokrivanja

Metrike za kvalitet test podataka

Za metrike imam podršku alata i to se na onoj slici na 10 strani na kraju strane nalazi u delu gde je monotoring u kom se skupljaju posećeni čvorovi i grane.
Metrika za stepen pokrivenosti iskaza:

$$C_{Statement} = \frac{\text{Number of visited nodes}}{\text{Number of all nodes}}$$

Metrika za stepen pokrivenosti grana

$$C_{Branch} = \frac{\text{Number of visited branches}}{\text{Number of all branches}}$$

Prednost structure oriented test kao što smo naglasili je to da se kvalitet test podataka može meriti.
Kod funkcionalnog testiranja nije moguće jer je kod nepoznat sve je iz funkcionalnih zahteva što je tekstulano.

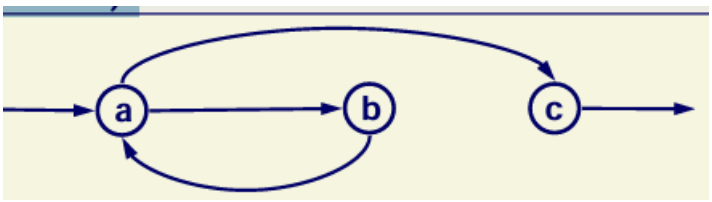
Problemi branch coverage testa:

Slabe tačke:
-Kombinacija grana nije uzeta u obzir(rešenje test pokrivanja putanja - **path coverage test**)
-Petlje su lošije testirane: ponavljanje testa(rešenje **boundary interior path test)**.
- Kompleksni uslovi nisu uzeti u obzir(uslovi koji imaju više && .. rešenje je **conditions coverage**)

Path coverage test

Zahtev: sve putanje u CFG su uzeti, gde je putanja ustvari put od ulaznog čvora do izlaznog.
Bolji pristup ali nije moguća primena zbog ciklusa(koliko putanja možemo imati u jednoj petlji)

Boundary interior path test



Zahtev: Test zatvorenog puta(putanje)- ponavljanje tela ciklusa nije potrebno više od jednom(slabija vrzija path testa).
Koje situacije se uzimaju u obzir tokom testiranja?

1. Telo petlje se ne izvršava(boundary) - ac
2. Telo petlje se jednom izvršilo(boundary) -abac
3. Telo petlje se jednom ponovilo(interior = jedno ponavljanje)-ababac

Za svaku situaciju(1,2,3) imamo kompletan path coverage i neophodan je!
Broj test slučajeva zavisi od strukture tela ciklusa b.

Stopa(rate): 70% svih grešaka je nađenom ovim načinom testiranja.

Condition coverage(pokrivanje uslova)

Kada pogledamo na 18 strani prvu sliku to su sa desne strane pristupi.

Simple condition coverage

Cilj: Uzeti u obzir logičku strukturu uslova
Zahtev: Svi atomski uslovi moraju se evaluirati na tačno i netačno
Primer: if (ch == 'x' || ch == 'y') ch = 'a';

1. test case: ch == 'x'
2. test case: ch == 'y'

Problem: svi zahtevi ispunjeni, ali nemamo pokrivanje grana(samo true grana prema test slučajevima)

Multiple conditions coverage

Zahtev: Sve kombinacije vrednosti atomskih uslova moraju biti evaluirane, isti primer al nova tabela

Test case / atom. cond.	ch == 'x'	ch == 'y'	ch == 'a'	ch == ?
ch == 'x'	true	false	false	true
ch == 'y'	false	true	false	true
	up to now	up to now	new	impossible

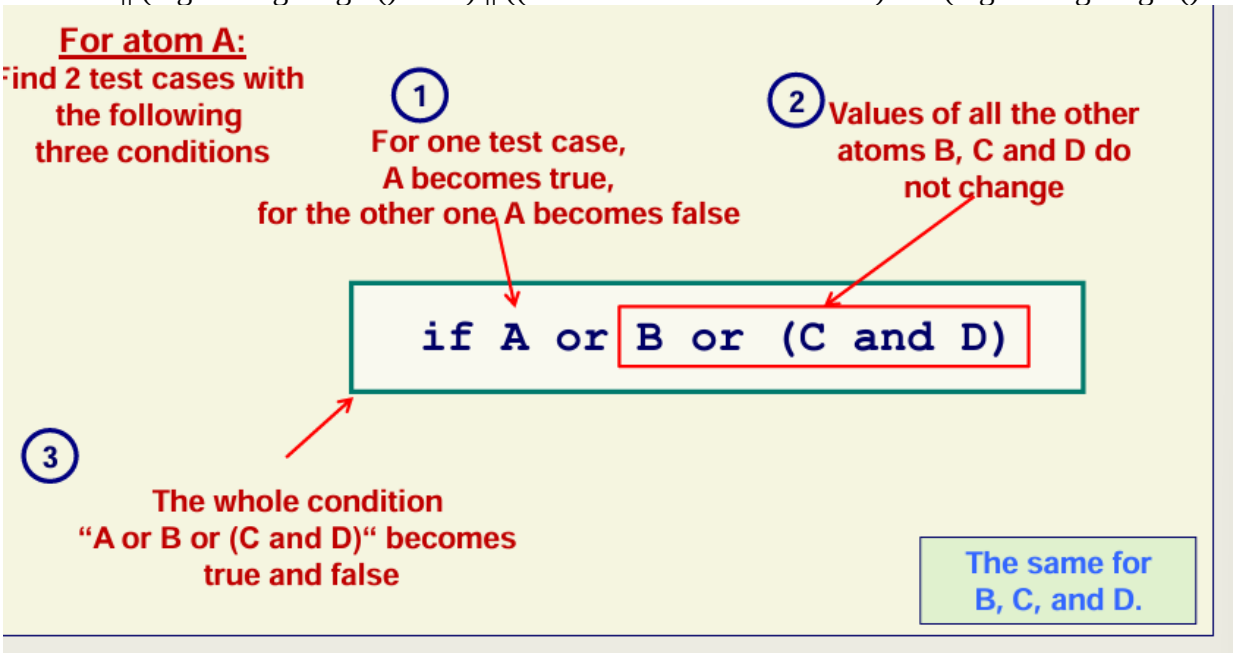
Minimal multiple conditions coverage

Zahtev: svi uslovi(atomski ili kombinacija) moraju se evaluirati na tačno ili netačno
Bolje pokrivanje grana jer grana samo u obzir uzima kobinovane uslove

Test cases / all conditions	ch == 'x'	ch == 'y'	ch == 'a'
ch == 'x'	true	false	false
ch == 'y'	false	true	false
(ch == 'x' || ch == 'y')	true	true	false

MC/DC
Modified Condition Decision Coverage

Varijantaa minimal multiple condition coverage
Ideja:
Selektovati sve test slučajeve sa osobinom:
-Za svaki atomski uslov može biti prikazano da ima uticaj na vrednost kompletnog uslova
Svaki atomski uslov je bitan.
Primer:
if (!withoutError || (digitString.length() == 0) || ((whereAmI == afterComma) &&(digitString.length() == 1))



Imamo 4 para test slučajeva ukupno 8 test slučajeva

Test case 1: "ab" → (true || false || (false && false) → complete: true
Test case 2: "12" → (false || false || (false && false) → complete: false

Test case 3: "" → (false || true || (false && false) → complete: true
Test case 4: "12" → (false || false || (false && false) → complete: false

Poređenje strukturom orientisnog i funkcionalnog testiranja urađeno gore na početku topic 14

Poređenje strukturom orientisne i funkcionalne selekcije test podataka i test strategije

Izvor za *Strukturom orijentisan* je sam **program**
Benefit: možemo meriti kvalitet test podataka(pokrivanje koji deo koda je izvršen)
Problemi: ako nedostaju funkcionalnosti ne mogu biti pronađeni(npr greške u zaboravljenim specijalnim slučajevima)
Alati: 1. za merenje stepena pokrivenosti 2. nema podršku za izbor test podataka

Izvor za *Funkcionalno* je **specifikacija**
Benefiti: program testiran prema specifikaciji
Problemi: (verbalna**)** specifikacija nije precizna i nije kompletna(spec. Zahteva)
nema merenja kvaliteta test podataka
Alati: sistematična selečcija test podataka **(Testona)**

Imamo zajedničku upotrebu ove dva načina testiranja:
-Funkcionalna selekcija test podataka
-Strukturom orijentisana evalucija

TESSY- alat za testiranje koji integriše strukturom orijentisano testiranje(monitoring) i funkcionalno testiranje(testona)

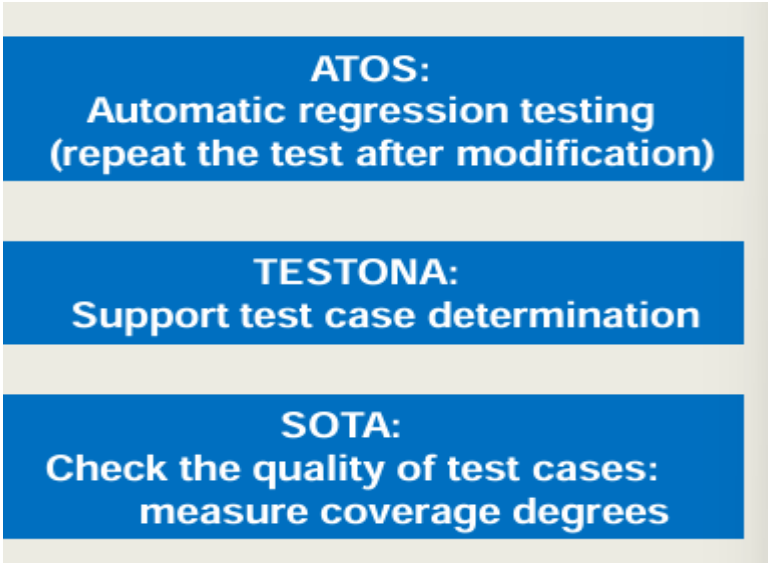
SOTA
Structure Oriented Testing Assistance

Podrška sa testiranje orijentisano strukturom tako što dostavlja stepene pokrivanja za dat skup test podataka i program(**ne izvlači** test podatke iz programa), vizualizje pokrivanje

Proverava kvalitet test slučajeva- meri stepen pokrivenosti
Nalazi se u okviru monitoringa sa slike.
-

Sažetak lekcija od 12-14

Šta je bilo bitno za testiranje?
1. Dva modela za testiranje koji su komplementi jedan drugom – Model testiranja aktivnosti i testiranje faznog modela (V modela)
2. Određivanje dobrih test slučajeva je bitan
3. Funkcionalno testiranje i strukturno testiranje su glavni metodi za nalaženje test slučajeva
4. Glavni metod za funkcionalno testiranje je CT metod
5. Specijalni metod funkcionalngo tesitranje: koncentirisanje na granične vrednosti.
6. Glavni metodi za strukturno testiranje: pokrivanje iskaza, grana, BI, pokrivanje uslova, MC/DC
7. Idealna strategija utvrđivanja test slučajeva je kombinacija strukturnog i funkcionalnog pristupa
8. Testiranje mora biti podržano alatima
9. Ne postoji jedinstveni niti najbolji alat za testiranje, sve zavisi od aktivnosti koja treba biti podržana



Treći test topic 8 je u svesci
Topic 18 - Metrike

Testiranje je bilo dinamička analiza i proveravalo je ponašanje softvera

Metrike su statička analiza – mere kvalitet softvera

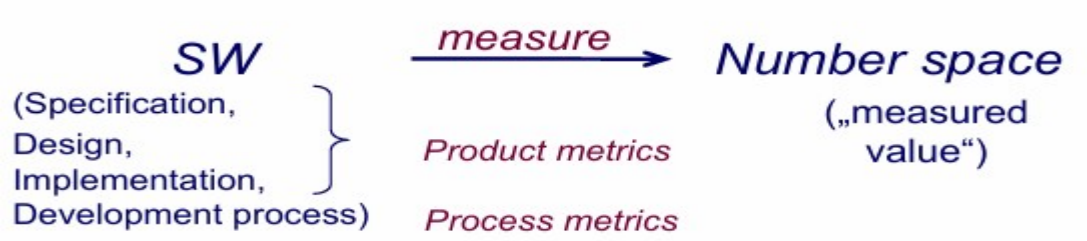
Motivacija: Zašto nam trebaju softverske metrike?
Kvantifikovane merenjem
1. Greške u softverskom razvoju su otkrivene kasno. (Research from ‘Software Metrics Symposium)
2. Ponovno korišćenje komponenti u razvoju objektno orjentisanih sistema smanjuje stopu grešaka.(Experiment in CACM)
3. Trud u ranim fazama razvoja softvera su vredni(kvalitativno)(Eksperiment univerziteta u montrealu)
4. Arhitektura softvera programa X je dobra (OO metrike)
5. Implementacija nekog programa X je veoma kompleksna i dugačka(LOC)

Ako se prisetimo definicije softverskog inženjerstva
SE zahteva merenja
SE predstavlja primenu sistematskog, disciplinovanog, **kvantifikovanog** pristupa razvoju, operaciji, i održavanju softvera; to je pprimena inženjerstva na softver.

Definicija softverske metrike

Pre nje same, Softverske metrike ciljaju na kvantitativnu evaluaciju softvera ili razvoja softvera

Def. „Metrika kvaliteta softvera: je funkcija čiji su ulazi podaci softvera i čiji je izlaz jedna numerička vrednost koja može biti iterpretirana kao stepen u kom softver poseduje određen atribut koji mu utiče na kvalitet“



Statistika grešaka: poreklo i detekcija

Prvi eksperiment:
Teza 1: Greške u razvoju softvera su otkrivene jako kasno

Drugi eksperiment
Teza 2: Ponovno korišćenje komponenti u razvoju objektno orjentisanih sistema smanjuje stopu grešaka.

Treci eksperiment: Univerziteta u Motrealu
Teza 3: Trud u ranim fazama razvoja softvera se isplate(pogotovo u dizajnu).
Pod trudom se podrazumeva utrošeno vreme.
Ono što se isplati je SW kvalitet: Korektnost(test), čitljivost (dobro dokumentovano kompaktno rešenje- nema duplikata koda kao ni mrtvog koda)

Zašto merimo?

Merimo da bi odgovorili na pitanja poput:
1. Koliko je veliko?
2. Koliko je održivo?
3. Da li je tehnika A više produktivna od tehnike B?
4. Kakve su karakteristike programa koji su skloni greškama?
5. Koliko dugo će trajati?

Ako nema pitanja onda nema potrebe ni da se meri.
Merenje pomaže da se prevaziđe subjektivnost i daje nam nivo preciznosti, koji ne možemo drugačije dobiti.

Istorijski razvoj metrika

1960-e

Problem „malih“ računara – Resursi za izvršavanje programa su bili jako bitni(korišćenje memorije, vremena izvršavanja)

1970-e

Problem: Puno grešaka u kompleksnim softverima – Mere kompleksnosti softvera Halstead i McCabe

1980-e

Problem: Greške u ranim fazama razvoja → Metrike su uključuju i specifikaciju i dizajn softvera(pre: samo program)

1990-e

Problem: Visoki troškovi razvoja → Razvojni proces je uključen za razliku od pre gde je bio samo proizvod.

2000-e

Problem: Neuspeh projekta → Metrike menadžementa projekta(npr. menadžer projekta trajno informisan o stanju projekta putem metrika, cena po liniji koda)

Klasifikacija metrika

1.Upotreba(Usage) – Vreme izvršavanja, upotreba memorije, dostupnost

2. Test
    Stopa grešaka:
        Prema fazama testiranja: jedinično testiranje, integraciono testiranje, sitemsko testiranje
        Prema vrsti greške
    Stepen pokrivenosti testa
3.Kod: LOC, ciklomatska složenost, Halstead-Metrics, Style-Metrics
4. Dizajn: Oometrike (stepen objektne orjentisanosti, veze(bindings), povezivanja(couplings))
5. Analize : Funkcionalni poeni, COCOMO(sta god ovo bilo)
6. Proces/poject menagment: Trud u razvoju – MM(vreme, osobe – u svakoj fazi)

Gde su metrike definisane(funckija: domen, raspon vrednosti):
LOC: program → dužina programa (broj)
Funkcionalni poeni: specifikacija zahteva -> funkcionalni porni(broj)
Pokrivenost iskaza(statement): program X test data → [0 .. 1]

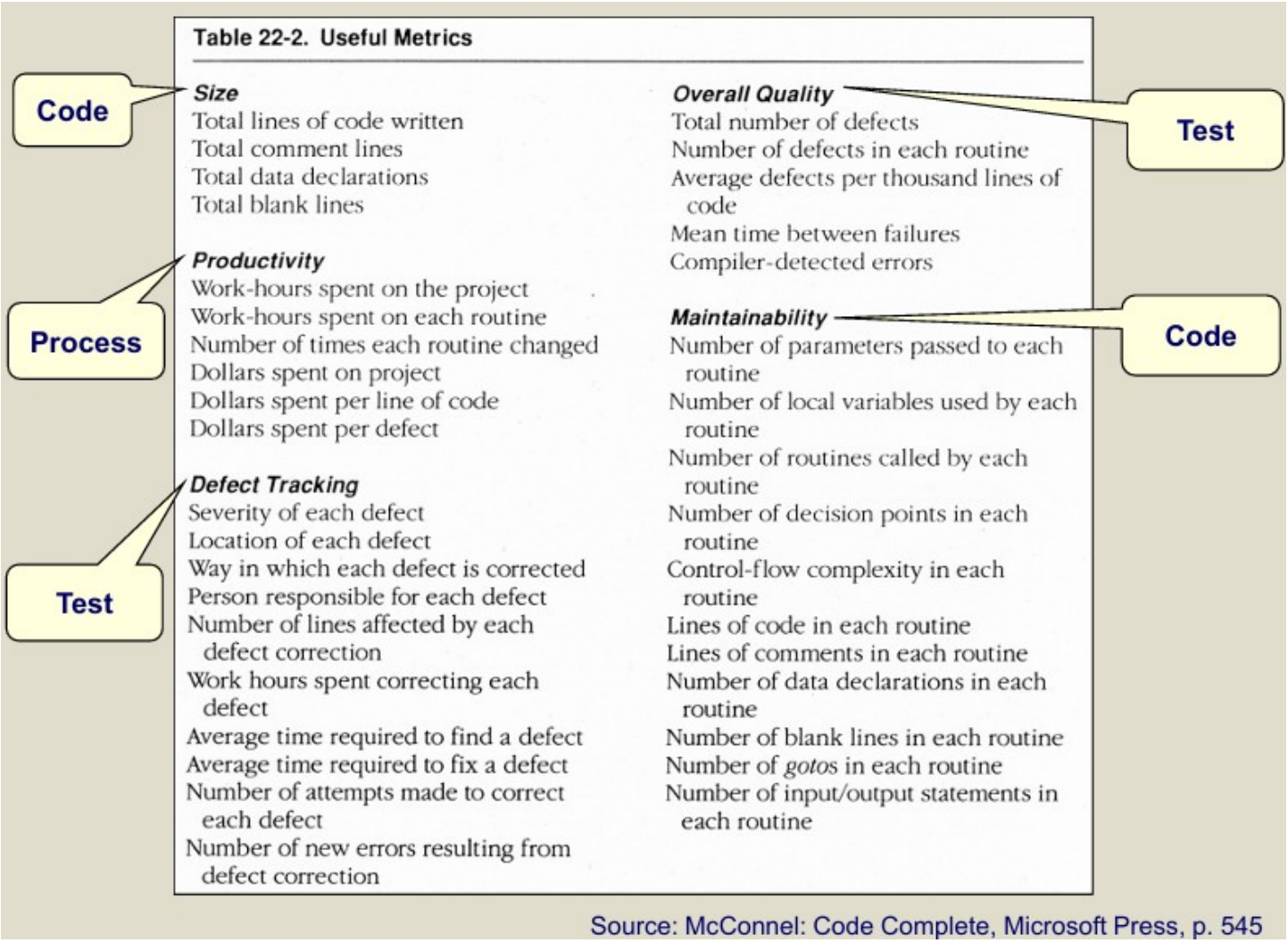


Table 22-2. Useful Metrics

Size
Total lines of code written
Total comment lines
Total data declarations
Total blank lines

Productivity
Work-hours spent on the project
Work-hours spent on each routine
Number of times each routine changed
Dollars spent on project
Dollars spent per line of code
Dollars spent per defect

Defect Tracking
Severity of each defect
Location of each defect
Way in which each defect is corrected
Person responsible for each defect
Number of lines affected by each defect correction
Work hours spent correcting each defect
Average time required to find a defect
Average time required to fix a defect
Number of attempts made to correct each defect
Number of new errors resulting from defect correction

Overall Quality
Total number of defects
Number of defects in each routine
Average defects per thousand lines of code
Mean time between failures
Compiler-detected errors

Maintainability
Number of parameters passed to each routine
Number of local variables used by each routine
Number of routines called by each routine
Number of decision points in each routine
Control-flow complexity in each routine
Lines of code in each routine
Lines of comments in each routine
Number of data declarations in each routine
Number of blank lines in each routine
Number of *gotos* in each routine
Number of input/output statements in each routine

Source: McConnel: Code Complete, Microsoft Press, p. 545

Cikolmatska kompleksnost
(Cyclomatic complexity v(G))

Vezana je za sam kod kad je u pitanju klasifikacija metrika
Osmislio McCabe 1976.
v(G)
Implementirana u mnogim alatima

Cilj: *Indetifikacija kompleksnih funkcionalnih komponenti (procedura, metoda, podprograma).*

→ kritično za podložnodt greškama, čitljivost, modifikabilnost
→ testing effort derivable

Control flow graph funkcija(unit)

Podsetnik za struct orient test isto
CFG opisuje strukturu iskaza
CFG sačinjen od:

1. N – skup čvorova(predstavljaju izraze(expresson) i iskaze(statements))
2. E – Skup grana (predstavlja tok kontrole između čvorova)

Ograničiti CFG na jednu funkciju(jedinicu)
Jedinica ima tačno jedan ulaz i jedan izlaz

Cikolmatska kompleksnost grafa
(Cyclomatic complexity v(G))

Def. Neka je G(N,E) CFG gde su N čvorovi, E grane, ciklomatska kompleksnost je definisana kao:

v(G) = e– n + 2 gde je e= |E| i n = |N| .

Tumačenje: Što je veća ciklomatska kompleksnost date funkcije to je više grananja.
Problemi sa razumevanjem, modifikacijom i sl.

Programi koji imaju sekvencijalne cvorove, tj nemaju grananja nego su samo čvor za cvorem, imaju minimalnu ciklomatsku kompleksnost 1.

Preporučena gornja granica ciklomatske vrednosti je 10.

Računanje ove formule troši dosta vremena i sklono je greškama u slučaju kada se ručno radi.
Postoje jednostavne alternative:

-Brojanje predikata
-Brojanje u kodu
-Brojanje regiona CFG

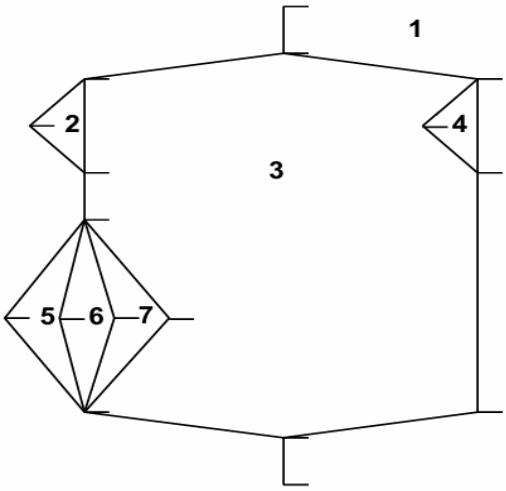
Brojanje binarnih predikata
Ako su svi uslovni iskazi(tačake odluke) binarni i ima ih p onda je **v(G) = p + 1**

Brojanje n-arnih predikata
Kada imamo više uslovnih iskaza, slično računamo. Počnemo sa 1 i dodajemo nešto za svaki predikat. Npr ako imamo dve grane dodajemo 1, ako imamo 3 grane dodajemo 2, za 4 dodajemo 3 itd.

Brojanje u kodu
brojanje predikata u kodu s tim da imamo u vidu da su u javi if i while binarni predikati.
Boolean operatori u if i while iskazima dodaju 1 u kompleksnost
primer slika desno

```
void complexity6(int i, int j)
{
  if (i > 0 && (j > 0)
  {
    while (i > j)
     {
      if (i % 2 && (j \% 2)
            printf("%d\n",i);
       else
            printf("%d\n",j);
       i--;
     }
  }
}
```

v(G) = 6



Brojanje regiona
Neka je CFG planaran, tj. da u njemu nema presecanja grana.
Neka bude podeljen u R regije (uključujući i beskonačnu regiju u kojoj se sam CFG nalazi).
Formula izračunavanja ciklomatske složenosti je
v(G) = R
Slika pored kao primer, 1 je taj region u kom leži sam CFG, kao celoukupan CFG je jedna regija

Esencijalna kompleksnost

Osnove:
Esencijalna kompleksnost meri stepen nestruktuiranosti programa.
Strukturno programiranje je bez goto(break..)

Primitivne strukture kontrole u strukturnom programiranju su sekvence, if, switch, whilr i do while

Definicija:
Neka je G(N,E) jedan CFG. Esencijalna kompleksnost je ev(G) neke jedinice i zahteva da se sve primitivne strukture kontrole u strukturnom programiranju rekurzivno brišu iz G (što duže može i to iznutra).
Neka G‘ redukovan graf onda je ev(G) = v(G‘)

Prednosti Ciklomatske kompleksnosti
1.Lako je izračunati iz teksta programa, kao i iz flowgraph-a
2. Podržava proces razvoja od vrha ka dnu radi kontrole složenosti modula u fazi dizajna, tj. pre nego što dođe do stvarnog kodiranja.
3. Pomaže u određivanju maksimalnog skupa nezavisnih test putanja.
4. Može se koristiti za kontrolu složenosti programskih modula. (McCabe je preporučio da se koristi gornja granica od 10 kao smernica za kontrolu složenosti programskih modula. Preporuku su podržali Schneidewind i Walsh /SCHN79a/ i Walsh /WALS79/.
5. **Može se koristiti za procenu alternativnog dizajna programa kako bi se pronašla najjednostavnija moguća struktura programa.**

6. Služi za deljenje strukture programa na visoku ili nisku učestalost grešaka prema njenoj vrednosti.
7. Služi za deljenje strukture programa na visoke ili niske vreme pronalaženja i uklanjanja grešaka prema njenoj vrednosti.
8. **Može se koristiti kao smernica za raspodelu resursa za testiranje.**

Mane:
1.Mera je samo psihološka složenost. Ne meri računarsku složenost.
2. Gleda sve predikate, bilo da su to alternativni ili iterativni, kao da doprinose istom stepenu složenosti.
3. Nije osetljiva na nivo **ugnježdenosti unutar različitih konstrukcija.**
4. ..... učestalost i vrste ulaznih i izlaznih aktivnosti.
5. ..... **veličinu čistih sekvencijalnih programa.**
6. ..... **broj promenljivih u programu.**
7. ..... **intenzivnost operacija sa podacima (tj. broj operatera i operanada) u programu.**
8. ..... zavisnost kontrolnih tokova od prethodnih operacija sa podacima.
9. ..... situacija u kojoj je jedan uslov "maskiran" ili "blokiran" od strane drugog unutar ugnježdene konstrukcije.
10. ..... stil programa i upotrebe GOTO izskaza.
11. Ne meri vrste i nivoe interakcije modula, niti nivoe invokacije modula.
12. Ne odražava tačno poboljšanje koje je proizašlo iz revizije programa.

Halsteadova metrika

Postoji od 1977.
Cilj: izmeriti tekstualnu kompleksnost(veličinu, samo leksičku)
Slično je LOC samo nezavisno od gustine linija
cyclomatic complexity: structured complexity (syntax)

Operatori: Simboli i ključne reči koje predstavljaju neku akciju npr. +, -, \, *, if, while, =, (,), {, }
Operandi: Simboli podataka(konstante i promenljive)

n_1: broj različitih operatora
n_2: broj različitih operanada
N_1: Ukupan broj iskorišćenih operatora
N_2: Ukupan broj iskorišćenih operanada
$n= n_1+n_2$: veličina vokabulara(simbola)
$N= N_1+N_2$: dužina implementacije
$D= (n_1*N_2)/(2n_2)$: težina programa(da se napiše, razume, održava)
N_2/n_2 :prosečan broj iskorišćenih operanada

Jednostavije je ovo izračunati nego ciklomatsku kompleksnost: tekstualna kompleksnost je evaluirana samo iz leksike.
Nezavisno od rasporeda: prazne linije, gustina linija (vidi LOC), prazni karakteri
Problem: samo leksikografska složenost, određivanje i evaluacija rezultata je teža nego kod LOC
Među najčešće korišćenim metrikama: ciklomatska složenost, Halstead, LOC

LOC - Metrike

LOC i KLOC: broj linija
Cilj: merenje tekstualne kompleksnosti slično halstedovoj.
Jednostavno je začunanje (nema sintakse, nema leksike)
Brži pregled izvornog kodaa
Zavisi od layout-a: prazne linije, linije sa komentarima gustina linija
Evaluacija layout-a: Stepen komentara, praznih linija
Siroko korišćena metrika (takođe i u drugim metrikama: produktivnosti, stope ponovnog korišćenja)

Metrika stila

Cilj: Kvalitet stila programiranja je izmeren
Zavisi od jezika u kom se programira i direktive zavisne od kompanije
Faktori za evaluaciju:(pretežno za C)
• Dužina jedinice (klase, funkcije: LOC (bez komentara ili praznih linija)
• Dužina identifikatora: broj karaktera u korisnički definisanim identifikatorima
• Komentari: Procenat komentarskih linija u ukupnom LOC
• Prazne linije: Procenat praznih linija u ukupnom LOC
• Uvlaka: Odnos vodećih razmaka prema ukupnom LOC
• Dužina linija: broj karaktera u linijama bez razmaka
• Razmaci po linijama (čitajnost izraza i deklaracija)
• Definicije konstanti: Procenat definicija konstanti u ukupnim deklaracijama
• Procenat standardnih identifikatora (stepen ponovne upotrebe: biblioteke klasa)
• Broj Include-Fajlova (strukturiranje programa)
• Procenat Goto
Alat
AssessStyle: Automated assessment of style in Java

AssessStyle: calculates metrics and „grade"

Parameter:
1. Weight of errors, for example:
   - missing comment 4
   - missing empty line 2
   - bad indentation 1
   - file too long 9
2. Relationship between metrics and grades

Grade	Metrics P
1	0 – 20
2	20 – 40
3	41 – 60
4	61 – 80
5	81 – 100
6	> 100

$$P = \sum_{f \in F} h(f) * g(f)$$

Set of all types of errors
Number of errors of type f in 100 LOC
Weight of the error f

Program	Operating system	GUI	File Types	Aim of application
AssessStyle	independent	yes	Java	display of style errors, assessment of style
Checkstyle	independent	no	Java	display of style errors
Artistic Style	independent	no	C, C++, C#, Java	new formatting
JStyle	windows	yes	Java	display of style errors, metrics
Praktomat	web-interface	web-interface	Java, C++ u. a.	function tests and display of style errors, manually assisted assessment
Jalote-Prog.	?	?	C	assessment of style

Checkstyle: Proverava puno stilova programiranja i open source je ali nema gui ni procenu
Jstyle: Moćan ali je komercijalan (1000USD / licenci), nema procenu stila ali ima metrike kao što je halsted

OO metrike

Cilj: izmeriti stepen i kvalitet objektne orijentisanosti
Aspekti: nasleđivanje, kompleksnost klasa, kohezija i coupling

Nasleđivanje:
DIT – dubina stabla nasleđivanja jedne klase – broj predaka klase + 1
Tumačenje: Što je veći DIT, veća mogućnost uvođenja grešaka (Java-API)

NOC(broj dece u klasi): Broj direktnih naslednika klase
Tumačenje: Što je veći NOC, više klasa je pogođeno promenama u klasi (podložne greškama). U ovoj situaciji, širina hijerarhije nasledstva je od značaja

Kompleksnost klasa

WMC(weighted method per class): broj metoda klase
Tumačenje: Što je veci WMC, veća mogućnoct uvođenja grešaka
Šta je dobar WMC?
Postoje različite granice koje su definisane.
Jedan način ograničavanja je po broju metoda u klasi npr 20 ili 50.
Drugi način je da se specificira da maksimum 10% klasa mogu da imaju više od 24 metoda.
Ovo dopušta većim klasama ali većina klasa bi trebalo da su male.

Ciklomatska kompleksnost(vezana za klase):
-sum v(G) : ukupan broj svih metoda u klasi
- avg v(G) : prosečan boj metoda u klasi
- max v(G) : maksimalna vrednost broja metoda u klasi
- max ev(G) : maksimalna vrednost broja esencijalne kompleksnosti metoda
Tumačenje: Indetifikacija klasa sa kompleksnim metodama(teže za čitanje i održavanje)

Coupling između klasa
CBO(Coupling between classes): broj klasa sa kojima klasa komunicira, tj servisi ovih klasa su korišćeni(osim u vezi nasledjivanja)

Tumačenje: Što je veći CBO, veća je mogućnost uvođenja grešaka.
"Koliko visoko je previše visoko? CBO > 14 je previše visoko"

Kohezija između elemenata klasa

LOCM(lack of cohesion in methods):
Manjak unutrašnje kohezije u klasi
Atributi kao vezivni elementi operacija
Koliko parova metoda ne pristupaju zajedničkim podacima

Inverzija
TCC(Tight Class Cohesion): Relativan broj parova metoda, koji imaju makar jedan zajednički atribut
Tumačenje: manja kohezija → deljenje klase(splitting)

Metrike: Scenario primene

1.Kao deo pregleda koda: Koristiti nepoznati softver, procena prema metrikama
2. Restrukturiranje softvera: Poboljšanje
3. Upravljanje projektom:
    Pregled stanja razvoja, kvalitet
    Evaluacija gotovih projekata(slabe tačke) – Telekom Baza metrika

Pažnja – Nema automatizacije u interpretaciji podataka metrike
Osnovno pravilo: Loše vrednosti metrike(npr. Visoka ciklomatska kompleksnost) znači **mogući** problemi
Potrebno je dobro ispitati problematične slučajeve
Dobra struktura je dobra i sa lošim vrednostima
Primer: Visoka ciklomatska kompleksnost, moguci razlozi: switch(mnogi slučajevi su prirodni ili nedostaje nasleđivanje)

Pogledati CASE study za XCTL na kraju lekcije ima dosta slika pa nisam izvlacila