



SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Kvalitet softvera - osnove

Lekcija 01

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Lekcija 01

KVALITET SOFTVERA - OSNOVE

- ✓ Kvalitet softvera - osnove
- ✓ Poglavlje 1: Osnove kvaliteta softvera
- ✓ Poglavlje 2: Osiguranje kvaliteta softvera (Software Quality Assurance)
- ✓ Poglavlje 3: Inspekcije, pregledi i revizije softvera
- ✓ Poglavlje 4: Tehnike verifikacije i validacije u SQA
- ✓ Poglavlje 5: Tehnike testiranja
- ✓ Poglavlje 6: Grupna vežba - Testiranje slučajeva korišćenja
- ✓ Poglavlje 7: Domaći zadatak 01
- ✓ Kvalitet, testiranje i održavanje softvera

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

Kvalitet softvera predstavlja sposobnost da se proizvede softver koji zadovoljava ili nadmašuje postavljene zahteve (prema definisanim merljivim kriterijumima) i koji je proizведен definisanim procesom. U procesu osiguranja kvaliteta softvera, mogu se primeniti statičke tehnike prevencije i otkrivanja grešaka, kao što su kontinuirana inspekcija (provera) i formalno pregledavanje, ali i različiti tipovi testiranja. Aktivnost testiranja softvera se normalno izvodi kao sredstvo pronalaženja grešaka sa ciljem njihovog odstranjivanja iz softverskog proizvoda. Testiranje softvera predstavlja deo šireg procesa – validacije i verifikacije softvera (V&V).

Cilj ove lekcije je:

1. *upoznati studente sa pojmom kvaliteta softvera*
2. *objasniti statičke tehnike prevencije i otkrivanja grešaka kao što su kontinuirana inspekcija (provera) i formalno pregledavanje ili review-a (FR) interne dokumentacije, koda, testova, itd.*
3. *objasniti aktivnosti procesa testiranja*
4. *objasniti različite vrste testiranja koje se mogu primeniti u cilju obezbeđenja što boljeg kvaliteta softvera*

▼ Poglavlje 1

Osnove kvaliteta softvera

IDEJA KVALITETA SOFTVERSKOG PROIZVODA

Kvalitet softvera jeste višedimenzionalni koncept koji se ne može jednostavno definisati.

Kvalitet softvera jeste višedimenzionalni koncept koji se ne može jednostavno definisati. Moderni softver koji se razvija, ima stalnu tendenciju rasta kompleksnosti. Sa porastom kompleksnosti realizovanih funkcija i primena, posebno je narastao zahtev za kvalitetom softvera u pogledu pouzdanosti, pogodnosti za testiranje i održavanje, ponovne upotrebljivosti, otpornosti na greške i drugih faktora kvaliteta softvera.

Veliki su gubici firmi usled grešaka u softveru koji su uslovili razvoj oblasti krivične odgovornosti proizvođača softvera i zaštite kupaca zbog neadekvatnog kvaliteta softverskog proizvoda.

Iz navedenih razloga, poslednjih godina intenzivirano je istraživanje u oblasti testiranja softvera u svetu, kao najvažnije aktivnosti u sistemu obezbeđenja kvaliteta softvera. Jedan od načina da se spreči isporuka softvera kupcu sa greškama je taj da sve kompanije ulažu sve više sredstava u obuku kadrova i opremu za testiranje softvera.

Pored konstantnog procesa poboljšanja velikog broja karakteristika procesa testiranja softvera npr. podizanje nivoa znanja i obučenosti testera (stručnjaci specijalizovani za testiranje softvera), automatizacije testova, razvoja novih alata, novih metoda testiranja softvera, veoma je važno razviti merljive tehnike za ocenu efikasnosti i efektivnosti postojećeg procesa testiranja softvera kako bi se otkrile njegove slabosti i prednosti, identifikovali rizici i njihove posledice.

Do danas nije razvijena metodologija testiranja softvera, kao najvažnija aktivnost u procesu obezbeđenja kvaliteta softvera, koja garantuje da će program biti bez grešaka, ali, sistematisovan i planirani postupak testiranja softvera, kroz izvršavanje programa sa pažljivo odabranim skupom ulaznih podataka, značajno povećava poverenje u korektnost programa.

Testiranje softvera odavno nije samo faza u procesu razvoja softvera, već paralelni podproces. Rešenja u softverskom inženjerstvu sve su složenija, uključuju više naučnih oblasti i tehnologija u različitim okruženjima. Pri određivanju kvaliteta softvera, potrebno je meriti više parametara s obzirom da postoji više uglova gledanja na kvalitet softvera. Jedan pogled na kvalitet je od strane korisnika (eksterne karakteristike kvaliteta), drugi na interne karakteristike softverskog proizvoda, a treći na sami proces razvoja softvera (PRS) tj. nivoa zrelosti po SEI CMM modelu ocenjivanja zrelosti softverske kompanije.

POJAM KVALITETA SOFTVERA I NAJČEŠĆE ZABLUDUDE

Kvalitet softvera predstavlja sposobnost da se proizvede softver koji zadovoljava ili nadmašuje postavljene zahteve i koji je proizведен definisanim procesom

Uspešne softverske kompanije uvele su sistem upravljanja kvalitetom (Quality Management System – QMS), koji povećava nivo poverenja u to da organizacija ostvaruje postavljene ciljeve kvaliteta, što podrazumeva definisanje i izbor različitih standarda koji se uključuju u proces razvoja softvera ili sam softverski proizvod. Izabrani standardi mogu biti ISO9000, JUS/ISO 12207, CMMI i dr. Svako preduzeće prema izabranom sistemu kvaliteta kreira svoj poslovnik o kvalitetu iz kojeg proističe dokumentacija sistema kvaliteta, a ova, opet, obezbeđuje kvalitet odvijanja odgovarajućih procesa u organizaciji i služi za uspešno vođenje različitih projekata, među kojima su i projekti razvoja softvera.

Kvalitet softvera predstavlja sposobnost da se proizvede softver koji zadovoljava ili nadmašuje postavljene zahteve (prema definisanim merljivim kriterijumima) i koji je proizведен definisanim procesom. Ovaj proces ne svodi se samo na zadovoljenje definisanih zahteva, već se u okviru njega moraju definisati mere i kriterijumi koji usmeravaju proces postizanja kvaliteta. Potrebno je usvojiti pravila za jedan ponovljiv i upravljiv proces čiji proizvodi će dostizati određeni nivo kvaliteta.

Jedna od definicija koja se pominje u softverskoj literaturi je i definicija sa stanovišta potrošača. Potrošač definiše kvalitet kao meru zadovoljavanja potreba kupca od strane proizvoda ili usluge. Drugim rečima, upotrebnim kvalitetom.

Najčešće zablude o kvalitetu:

1. **Kvalitet može biti naknadno dodat i "utestiran"** – kvalitet mora biti opisan i ugrađen u proces stvaranja proizvoda.
2. **Kvalitet dolazi sam od sebe** – kvalitet ne nastaje tek tako. Proces razvoja mora se definisati, sprovoditi i kontrolisati kako bi se dostigao određeni nivo kvaliteta.
3. **Kvalitet je jednodimenzionalna karakteristika i svakom znači isto-** **kvalitet ima više dimenzija od kojih su najvažnije:** funkcionalnost, pouzdanost, upotrebljivost, efikasnost, stepen podrške. Svaku od dimenzija prati odgovarajući tip testiranja softvera.

Da bi ostale profitabilne, organizacije koje razvijaju softver, moraju biti konkurentne po ceni i ponuditi kratke rokove završetka softverskog projekta. Tek nakon toga, tržište, na osnovu upotrebnog kvaliteta, daje realnu ocenu softverskog proizvoda.

MODELI I KARAKTERISTIKE KVALITETA SOFTVERA

Kvalitet procesa doprinosi kvalitetu proizvoda, a kvalitet proizvoda – upotrebnom kvalitetu.

Iako u softverskoj industrijskoj praksi ne postoji usvojen standard kvaliteta softvera koji određuje šta to čini dobar kvalitet softvera, generalno, kvalitetan softver jeste onaj koji zadovoljava potrebe kupca, doprinosi profitu, ne stvara ozbiljne probleme u poslovanju i po zdravje ljudi. Kvalitet softvera teško je definisati jer postoje različiti aspekti i termini koji su definisani i opisani u nizu standarda kao što su ISO 9000-3, ISO/IEC 9126 (Kvalitet softverskog proizvoda) i ISO/IEC 14598 (Vrednovanje softverskog proizvoda) i drugi.

Osnovne dimenzije kvaliteta softvera su:

- 1. Stepen zadovoljenja: kvantitativno izražen nivo zadovoljenja potreba kupca i njegovih očekivanja od softverskog proizvoda.
- 2. Vrednost proizvoda: kvantitativno izražen nivo vrednosti softverskog proizvoda za pojedine učesnike (vlasnike projekta) u odnosu na uslove konkurenčije.
- 3. Ključni atributi (Q - "lities"): kvantitativno izražen nivo kombinacije više karakteristika koje softver poseduje (npr. pouzdanost, upotrebljivost, pogodnost za održavanje).
- 4. Podložnost greškama: kvantitativno izražen nivo neispravnog funkcionisanja softvera u korisnikovom okruženju usled grešaka u isporučenom softveru.
- 5. Kvalitet procesa: kvalitet softvera izgrađuje se u procesu razvoja softvera koji treba da obezbedi postavljene ciljeve u pogledu kvaliteta (dobri projektanti koji na pravi i efikasan način izrađuju softver).

Povezanost kvaliteta procesa i kvaliteta proizvoda u upotrebi jasna je. Kvalitet procesa doprinosi kvalitetu proizvoda, a kvalitet proizvoda – upotrebnom kvalitetu. Moguća je i povratna sprega – od proizvoda u upotrebi, ka kvalitetu proizvoda tj. procesu kvaliteta. Interni atributi utiču na eksterne attribute, a eksterni – na kvalitet u upotrebi. Iz ovoga, može se zaključiti sledeće:

Zahlevi koji se odnose na kvalitet softverskog proizvoda uključuju kriterijume ocenjivanja za interni kvalitet, eksterni kvalitet i kvalitet u upotrebi radi zadovoljenja potreba projektanata, onih koji održavaju proizvod, naručilaca i korisnika. (ISO/IEC 14598-1:1999, tačka 8.)

STANDARDI KVALITETA SOFTVERSKOG PROIZVODA I SISTEMA: ISO/IEC 25000 SERIJA – SQUARE – KVALITET

Cilj standardizacije u kvalitetu i održavanju softvera je definisanje zajedničkog okvira koji omogućuje da svi učesnici u razvoju softvera komuniciraju i međusobno se razumeju

Standard je najveći stepen uređenosti neke oblasti, koji opisuje šta treba raditi, za razliku od "najbolje prakse" koja opisuje kako nešto treba raditi, ili "procedure" u sistemu kvaliteta koja opisuje kako se nešto radi u konkretnoj organizaciji.

Cilj standardizacije u kvalitetu i održavanju softvera je definisanje zajedničkog okvira koji omogućuje da svi učesnici u procesu razvoja, projektovanja i upravljanja softverom međusobno komuniciraju i razumeju se.

Pri tome se ne nameće određeni model, tehnika ili aktivnost, nego se sistem zasniva na prethodnom usaglašavanju seta pravila, kojih se svi učesnici u procesu nakon usvajanja moraju pridržavati.

Na taj način postupak održavanja softvera, koji u nekim slučajevima može činiti čak 80% ukupne cene softvera, postaje uređen, efikasan i ekonomičan.

ISO/IEC 25000 serija – SQuaRE – kvalitet softverskog proizvoda i sistema.

Naslednik je ranijeg SRPS ISO/IEC TR 9126-2:2010 standarda - Kvalitet softverskog proizvoda preko:

- *modela kvaliteta*
- *eksterne metrike*
- *interne metrike*
- *metrike kvaliteta u upotrebi,*

kao i ranijeg ISO/IEC 14598 – Vrednovanje softverskog proizvoda kroz:

- *planiranje i upravljanje*
- *procese*
- *dokumentaciju a zatim*

ISO/IEC TR 14143 serija – Merenje softvera – merenje funkcionalne veličine

Namenjeni projektantima, dobavljačima softvera, sticaocima (naručiocima) softvera i nezavisnim ocenjivačima.

▼ Poglavlje 2

Osiguranje kvaliteta softvera (Software Quality Assurance)

PLAN ZA OSIGURANJE KVALITETA SOFTVERA (SOFTWARE QUALITY ASSURANCE PLAN - SQA)

Uloga SQA je da se osigura da su planirani procesi prikladno izabrani i kasnije implementirani prema planu, i da je pružen relevantan proces merenja.

Plan za osiguranje kvaliteta softvera (eng. **Software Quality Assurance Plan - SQA**) pokušava da održava kvalitet kroz razvoj i održavanje proizvoda putem izvršavanja više različitih aktivnosti u svakoj fazi što može rezultirati ranom identifikacijom problema. Uloga SQA je da se osigura da su planirani procesi prikladno izabrani i kasnije implementirani prema planu, i da je pružen relevantan proces merenja.

SQA plan definiše način koji će biti korišćen kako bi se osiguralo da softver zadovoljava korisnikove zahteve i da je najvećeg mogućeg kvaliteta unutar ograničenja projekta. Radi zadovoljenja tog cilja, plan mora prvo obezbediti da je ciljni kvalitet jasno definisan i shvaćen. SQA Plan mora biti razmatran od strane menadžmenta i konzistentan sa SQM planom (eng.**software configuration management plan**).

Sa stanovišta menadžmenta, tri su osnovna međuzavisna parametra koje je potrebno kontrolisati tokom razvoja softvera: *troškove, redosled izvođenja radova i kvalitet*.

Kako bi se proizveo softver sa što nižim troškovima, razvoj softvera mora biti utemeljen na valjanoj inženjerskoj praksi.

TROŠKOVI ZA OSIGURANJE KVALITETA SOFTVERA

Teško je upravljati nečim što ne možemo izmeriti. Nedostatak pouzdanih merila proizvoda i procesa, otežava planiranje i kontrolu.

Softver ima neke specifičnosti koje uzrokuju probleme. Softver je apstraktni rezultat mentalnog procesa, on je nevidljiv. Opisi različitih aspekata softvera kao što su podaci, funkcije i kontrole, dostupne su samo u obliku dijagrama ili tekstova.

Softver je nematerijalan i samim tim, teško ga je meriti, a nečim što ne možemo izmeriti je veoma teško nadalje upravljati. Nedostatak pouzdanih merila proizvoda i procesa otežava planiranje i kontrolu.

Kvalitet isporučenog softvera se ne može garantovati: proizvodnja softvera nije kvantifikovana i značajno je niska, dok su troškovi razvoja i procene isporuke često nerealni.

Iskustvo pokazuje da se zahtevi softvera često menjaju, važne analize se sprovode površno, što uzrokuje promene zahteva i specifikacija softvera, prema nadolazećim potrebama, međutim, softver je lako menjati ali ga je teško menjati korektno.

MODEL TROŠKOVA KVALITETA SOFTVERA (ENGL. TOTAL QUALITY MANAGEMENT - TQM)

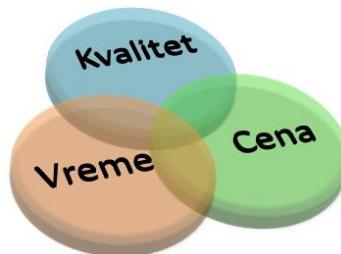
U inženjerstvu kvalitet softvera, cena i vreme su tri međusobno zavisna faktora.

TQM model (eng.[Total Quality Management](#)) podrazumeva potpuno uključivanje svih funkcija i svih zaposlenih u preduzeću, dobavljača i kupaca na ostvarivanju visokog kvaliteta softvera uz najniže troškove.

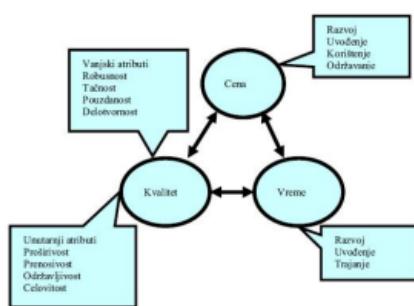
U inženjerstvu kvalitet softvera, cena i vreme su tri međusobno zavisna faktora, kako je to prikazano na slici 2.1.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ako su dva od ovih [faktora](#) konstante, treći [faktor](#) nije moguće kontrolisati. Ova tri elementa uzrokuju i najveće probleme, jer su troškovi i vreme isporuke softvera često iznad predviđenih veličina, a često softver ne zadovoljava zahteve korisnika (Slika 2.2).



Slika 2.1 Tri ključna faktora TQM modela [Izvor: NM SE321 - 2020/2021.]



Slika 2.2 Odnos kvaliteta, vremena i cene [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 3

Inspekcije, pregledi i revizije softvera

STATIČKE TEHNIKE PREVENCIJE GREŠAKA U PROCESU OSIGURANJA KVALITETA

Jedan od problema koji se rešava kontinuiranim testiranjem je eliminisanje troškova otklanjanja grešaka primenom statičkih tehnika, kao što je kontinuirana inspekcija.

Testiranje softvera je, u tradicionalnom pristupu razvoja softvera, poslednja faza u razvoju u toku koje su se otklanjale poslednje greške, koje su mogle biti ustanovljene ovim putem.

Međutim, moderni razvoj softvera podrazumeva korišćenje metodologije koja omogućava integraciju testiranja u kompletan razvojni ciklus, ne samo pred samu distribuciju softvera krajnjim korisnicima.

Jedan od problema koji se rešava kontinuiranim testiranjem je eliminisanje troškova otklanjanja grešaka primenom statičkih tehnika kao što su sukontinuirana inspekcija (provera), kontinuirano poboljšanje procesa razvoja i obezbeđenje kvaliteta softvera. Greške se prema McConnellu (1997), najjednostavnije i najekonomičnije ispravljaju u trenutku kada su nastale, odnosno u odgovarajućoj fazi razvoja softvera.

Kontinuirana inspekcija (provera) se odnosi na „statičku proveru, pre implementacije kompletног softvera, za razliku od testiranja koje podrazumeva „dinamičku“ proveru prilikom izvršavanja. Podrazumeva pregledavanje izvornog koda kako bi se pronašle anomalije, greške i delovi koda koji bi mogli izazvati neželjeno ponašanje prilikom izvršavanja. Može biti primenjeno u bilo kojoj fazi životnog ciklusa (zahtevi, projektovanje softvera), ali ne može zameniti testiranje softvera.

Može se primeniti i na nepotpunim - nedovršenim sistemima, pre konačne implementacije softvera. Pored pronalaženja grešaka, tokom ‘inspekcije’ mogu biti sagledane i druge važne karakteristike sistema, kao što su saglasnost, prenosivost i održivost, ali se ne mogu proveriti nefunkcionalne karakteristike sistema kao što su performanse, upotrebljivost i druge.

Kontinuirano poboljšanje se odnosi na primenu mehanizma poboljšanja softverskih sistema, ali i na poboljšanje čitavog procesa prenosa i usvajanja znanja.

Nakon izvršenja statičke analize kvaliteta, dobijaju se vrednosti softverskih metrika. Ukoliko posmatrano rešenje ne zadovoljava softverske metrike, dobija se povratna informacija na osnovu čega je moguće poboljšati proces prenosa i usvajanja znanja.

Sa druge strane, na osnovu vrednosti metrika, moguće je izvršiti ispravke (u smislu primene mehanizma poboljšanja softverskih sistema) i dobiti novi programski kod softverskog sistema. Novo rešenje je takođe moguće analizirati i poboljšavati, u cilju ostvarivanja željenog nivoa kvaliteta softvera.

FORMALNI PREGLED

FR je definisan, struktuirani, isproban, proveren i disciplinovani proces koji podrazumeva nalaženje defekata, eliminaciju defekata i korekciju i verifikaciju u bilo kojoj fazi razvoja SW.

Pored inspekcije važna statička tehnika prevencije i otkrivanja grešaka u toku razvoja softvera, koja se često koristi, je proces **formalnog pregleda** ili review-a (FR) interne dokumentacije, koda, testova, itd.

FR je definisan, struktuirani, isproban, proveren i disciplinovani proces koji podrazumeva nalaženje defekata, eliminaciju defekata kao i korekciju i verifikaciju u bilo kojoj fazi razvoja i održavanja softvera (SW) za proizvod u celini ili neki njegov deo.

Pod defektom se podrazumeva bilo koja **greška, otkaz ili nedostatak u radu**, kao i odstupanje od pravila kod zadovoljenja zahteva korisnika.

Osnovni cilj FR je eliminacija defekata u radu proizvoda sa što je moguće manjom cenom, samim tim i poboljšanje kvaliteta softvera u što ranijoj fazi životnog ciklusa SW. Formalnom review-u su podložni svi dokumenti počev od projektnih i razvojnih planova, funkcionalnih specifikacija, specifikacija dizajna i implementacije, sours koda, test planova i procedura. FR može početi bilo kad u životnom ciklusu SW proizvoda, ali je efikasniji ako se primenjuje još u ranim fazama razvoja, kada je cena pronalaženja i otklanjanja greške daleko manja.

Korist od FR nije samo u otkrivanju grešaka i smanjenju cena koštanja, već uključuje i poboljšanje **sinergije razvojnog tima, morala saradnika, povećanje dubine tehničkog znanja, ponosa zbog autorstva, a doprinosi i poboljšanju standarda za projekat**.

FR se može koristiti za određivanje kvaliteta SW proizvoda ili nekog njegovog dela, ali se nikako ne sme koristiti za procenu kvaliteta ljudi koji su učestvovali u razvoju proizvoda i čiji je rad predmet FR. **Zbog toga se preporučuje da menadžeri ne učestvuju u procesu, niti da prisustvuju sastancima review tima.**

Dakle, formira se **nezavisni tim za pregled i reviziju upravljanja razvojem softvera**. Osim toga, rezultate FR treba prezentovati menadžmentu samo u obliku statističkih podataka, popunjениh checklist-i, grupnih zaključaka i sl.

Takvi grupni zaključci će dati sliku menadžeru o rezultatima FR bez uplitanja bilo kog od autora ili učesnika FR procesa. Korišćenje FR procesa za određivanje sposobnosti rada nekog autora može prouzrokovati rezultate koji su manje pošteni i manje potpuni; nekim od saradnika će biti protiv volje identifikacija defekata i davanje loših ocena kolegama.

ČLANOVI NEZAVISNOG TIMA ZA PREGLED I REVIZIJU

U timu učestvuju: predsedavajući (moderator), autor, zapisničar, i najmanje jedan, a najviše 3 (4) inspektora.

FR proces je adaptabilna, **easy-to-learn** i **easy-to-use** tehnika koju realizuje nezavisni tim učesnika pregledom i unapređenjem (revizijom) proizvoda u nekoj fazi razvoja SW. Broj učesnika u FR procesu je najmanje 4, a najviše 8, i uključuje:

1. predsedavajućeg (moderatora),
2. autora,
3. zapisničara, i
4. najmanje jednog, a najviše 3 (4) inspektora.

Broj članova tima zavisi od veličine problema, i mogu se angažovati ako je potrebno njihovo mišljenje i stručnjaci iz odgovarajuće oblasti. Dobar tim čini dobra kombinacija Inspektora koji su iz različitih oblasti ekspertize (bitnih za problem).

Znanje i iskustvo svakoga od njih, gde svako posmatra proizvod iz svog ugla/perspektive, olakšava da mnogi propusti izadu na videlo. Sinergija ("Synergy"), gde ideja jednog člana tima često vodi ka drugoj ideji nekog drugog člana tima, je indikacija zdravog procesa inspekcije, odnosno FR.

Svaki od učesnika u FR procesu (član tima) ima jasno definisanu i dodeljenu ulogu, opisanu u dokumentaciji za sprovođenje procesa formalnog pregleda. Ukupan broj radnih sati po review-u je tipično u opsegu od 30 do 60 sati, u zavisnosti od veličine tima i broja pronađenih defekata.

ULOGE ČLANOVA NEZAVISNOG TIMA ZA PREGLED I REVIZIJU

Moderator je ključni čovek u FR procesu koji je odgovoran za upravljanje i vođenje review procesa.

Lider pregledavanja (Review Leader)/Moderator/Predsedavajući

Moderator je ključna osoba u FR procesu koja je odgovorna za upravljanje i vođenje review procesa. Uloga moderatora zahteva veštine u rukovođenju, upravljanju i stvaranju tima, kao i tehničku kompetentnost u oblastima softverskog inženjerstva kao i iz oblasti proizvoda koji je predmet pregleda (**review**). Moderator učestvuje u svim fazama FR procesa.

Zapisničar

Zapisničar je odgovoran za dokumentovanje zaključaka (na primer: pronađenih defekata, nekonistentnosti, propusta), odluka i preporuka koji su rezultat rada tima za pregledavanje. Ako postoji odgovarajuća baza podataka, zapisničar je odgovoran za unošenje odgovarajućih podataka. Njegovo zaduženje je i da vodi detaljan zapisnik za vreme sastanka, kao i da

preuzeće sav relevantni materijal od učesnika u FR procesu pre i posle sastanka. Materijal se nakon pregleda-a predaju Moderatoru (Predsedavajućem).

Član tima za pregledavanje (Review Team)/Inspektor

Svaki član tima je odgovoran za sopstvenu pripremu za pregled, kao i za uspešno obavljanje samog pregleda. Svi članovi tima zajedno su odgovorni za formulisanje preporuka na takav način da ih management može realizovati što pre. Inspektor treba da ispita proizvod na osnovu instrukcija Moderatora, što podrazumeva detekciju i dokumentovanje svih defekata u posebnoj log formi ili direktno u radnom dokumentu.

Identificuje sve greške u tekstu, uključujući slovne/tipografske, gramatičke i sintaksne, i vidno markira pronađene greške u radnom dokumentu.

Učestvuje u review procesu i sastancima u skladu sa instrukcijama moderatora, što uključuje detekciju i klasifikaciju grešaka uz eventualno pretraživanje oblasti interesovanja radi provere saglasnosti, jasnoće, testabilnosti i sl.

Autor

Autor ne učestvuje u procesu pregledavanja, a njegova zaduženja su da na sastanku obezbedi odgovore na pitanja članova tima za pregledavanje (Inspektori), a nakon sastanka i odluka menadžmenta da izvrši korekcije pronađenih defekata.

Posmatrač

Posmatrač ne učestvuje direktno u procesu pregledavanja. Podrazumeva se samo prisustvo na sastanku. Njegovo učešće na sastanku je opcionalno i određuje ga nalogodavac (management). Zaduženja na sastanku se odnose na procenu aktivnosti i načina rada članova tima za metode koje su korišćene, uočavanje prednosti i nedostataka primjenjenog postupka.

Menadžment

Menadžment je odgovoran da nakon procesa pregledavanja reaguje što pre na preporuke tima za pregledavanje i angažuje autora (autore) da izvrše tražene i usvojene korekcije.

TIPOVI PREGLEDA I REVIZIJA: REVIZIJA MENADŽMENTA

U IEEE1028-97 standardu je prikazano pet tipova pregleda ili revizija.

Revizija menadžmenta određuje adekvatnost planova, rasporeda, zahteva i nadzire njihov progres ili nekonzistentnost.

Zbog svrhe kratkog prikaza, pregledi i revizije su ovde posmatrane kao jedna tema, u odnosu na dve posebne teme kao u (IEEE12207.0-96). Proces pregleda i revizije široko je definisan u (IEEE12207.0-96) i detaljnije u (IEEE1028-97). Pet tipova pregleda ili revizija su prikazani u IEEE1028-97 standardu:

1. Revizije menadžmenta (eng. Management reviews)
2. Tehničke revizije (eng. Technical reviews)
3. Pregledi (eng. Walk-throughs)
4. Inspekcije (eng. Inspections)
5. Provere (eng. Audits)

Svrha revizije menadžmenta je nadgledanje progrusa, utvrđivanje statusa planova i rasporeda, potvrda zahteva i njihove alokacije, ili procena efektivnosti upravljačkog pristupa korišćenog za postizanje usklađivanja zbog svrhe" [IEEE1028-97].

Oni podržavaju odluke o promenama i korektivnim akcijama koje su potrebne u toku softverskog projekta. Revizije menadžmenta određuju adekvatnost planova, rasporeda, zahteva i nadziru njihov progres ili nekonistentnost. Ove revizije se mogu izvršavati nad proizvodima kao što su izveštaji revizije, izveštaji o progresu, V&V izveštaji, planovi više vrsta, koji uključuju menadžment rizika, projektni menadžment, menadžment konfiguracije softvera, bezbednost softvera i procenu rizika, između ostalih.

TIPOVI PREGLEDA I REVIZIJA: TEHNIČKA REVIZIJA I PREGLEDI

Svrha tehničke revizije je da proceni softverski proizvod kako bi utvrdila da li je prikladan za nameravanu upotrebu dok pregledi imaju za cilj ocenu softverskog proizvoda

Svrha tehničke revizije (eng.Technical reviews) je da proceni softverski proizvod kako bi utvrdila da li je prikladan za nameravanu upotrebu. Cilj je utvrđivanje odstupanja od odobrenе specifikacije i standarda. Rezultati bi trebalo da omoguće menadžmentu dokaze o potvrdi (ili ne) da proizvodi odgovaraju specifikaciji i pridržavaju se standarda, i da su sve promene kontrolisane" (IEEE1028-97)". Specifične uloge se moraju utvrditi prilikom tehničke revizije: donosilac odluka, vođa revizije, zapisivač, i tehničko osoblje za podršku aktivnosti revizije. Tehnička revizija zahteva da se obavezni ulazi ispune kako bi se nastavilo:

- *Izjava o ciljevima*
- *Specifičan softverski proizvod*
- *Poseban plan projektnog menadžmenta*
- *Lista problema u vezi da ovim proizvodom*
- *Procedura tehničke revizije*

Tim prati proceduru revizije. Tehnički kvalifikovana osoba prikazuje pregled proizvoda i ispitivanje se sprovodi tokom jednog ili više sastanaka. Tehnička revizija je završena kada se sve aktivnosti koje su navedene u ispitivanju završe.

Svrha pregleda (eng.Walk-throughs) je ocena softverskog proizvoda. On se može sprovesti za svrhu edukovanja publike o softverskom proizvodu." (IEEE1028-97) Glavni ciljevi su [IEEE1028-97]:

- *Nalaženje anomalija*
- *Unapređenje softverskog proizvoda*
- *Razmatranje alternativnih implementacija*
- *Ocena saglasnosti prema standardima i specifikacijama*

Pregled je sličan inspekciji, ali se obično smatra za manje formalan način.

Prezentacija se primarno organizuje od strane softverskog inženjera kako bi dao svojim članovima tima šansu da izvrše reviziju njegovog rada, kao tehnika uverenja.

TIPOVI PREGLEDA I REVIZIJA: INSPEKCIJE

Svrha inspekcije je da otkrije i identificuje anomalije softverskog proizvoda“ (IEEE1028-97).

Svrha **inspekcije** (eng. **Inspections**) je da otkrije i identificuje anomalije softverskog proizvoda“ (IEEE1028-97). Značajne razlike inspekcije u odnosu na reviziju su:

- 1. *Individua koja je pretpostavljena bilo kom od članova tima za inspekciju ne bi trebalo da učestvuje u inspekciji.*
- 2. *Inspekciju bi trebalo da vodi nepristrasan moderator koja je obučen da vrši inspekciju.* Inspekcije softvera uvek uključuju autora međuproizvoda ili finalnog proizvoda, dok kod ostalih revizija to ne mora da bude slučaj. Inspekcija, takođe, uključuje vođu, snimatelja, čitača, i nekoliko (2 do 5) inspektora. Članovi tima inspekcije mogu posedovati različita ekspertna znanja, kao što su poznavanje oblasti, ekspertiza metode dizajniranja ili ekspertiza jezika. Inspekcija se obično sprovodi na jednom relativno malom broju proizvoda, istovremeno.
- 3. *Svaki član tima mora da ispita softverski proizvod i druge ulaze za reviziju prioritetne za sastanak revizije,* možda primenjujući analitičku tehniku na malom delu proizvoda, ili na celom proizvodu sa fokusom na samo jednom aspektu, na primer, interfejsu. Bilo koja otkrivena anomalija se dokumentuje i šalje vođi inspekcije. Tokom inspekcije, vođa sprovodi sesiju i verifikuje da su svi spremni za inspekciju.
- 4. *Lista, sa anomalijama i pitanjima povezanim sa problemima od interesa, je uobičajen alat koji se koristi u inspekcijama.* Rezultujuća lista često klasificiše anomalije (više u IEEE1044-93) i razmatrana je njena potpunost i tačnost od strane tima.

Izlazne odluke inspekcije moraju da odgovaraju jednom od tri naredna kriterijuma:

1. *Prihvatanje bez ili sa najmanjim prepravkama*
2. *Prihvatanje sa verifikacijom prepravki*
3. *Ponovna inspekcija*

Sastanci inspekcije obično traju nekoliko sati, dok tehničke revizije i provere obično su opštije i traju duže.

TIPOVI PREGLEDA I REVIZIJA: PROVERE

Svrha softverske provere je obezbeđivanje nezavisne procene usklađenosti softverskih proizvoda u odnosu na primenljive regulacije, standarde, smernice, planove i procedure

Svrha **softverske provere** (eng. **Audits**) je obezbeđivanje nezavisne procene usklađenosti softverskih proizvoda i procesa u odnosu na primenljive regulacije, standarde, smernice, planove i procedure“ [IEEE1028-97]. Provera je formalno organizovana aktivnost, gde učesnici imaju specifične uloge, kao što su glavni revizor, pomoćni revizor, snimatelj, ili

inicijator, i uključuju predstavnika organizacije u kojoj se vrši provera.

Provera će identifikovati sve slučajeve odstupanja i kao rezultat toga dati izveštaj u kome se zahteva od tima da sprovedu korektivne akcije. Iako postoji dosta formalnih naziva za revizije i pregledе, kao oni koji su navedeni u standardu (IEEE1028-97), značajno je da ona mogu da se pojave za skoro bilo koji proizvod u bilo kojoj fazi razvoja ili procesa održavanja

▼ Poglavlje 4

Tehnike verifikacije i validacije u SQA

TEHNIKE VALIDACIJE U MODELU RAZVOJA SOFTVERA

Validacija se sprovodi u skladu s planiranim rešenjima kako bi se osiguralo da konačni proizvod može zadovoljiti zahteve za određenu primenu ili upotrebu za koju je predviđen.

Kao što smo istakli, testiranje predstavlja deo šireg procesa - validacije i verifikacije softvera (V&V).

Validacija (potvrda) se sprovodi u skladu s planiranim rešenjima kako bi se osiguralo da konačni proizvod može zadovoljiti zahteve za određenu primenu ili upotrebu za koju je predviđen. Važno je da se validacija izvrši pre isporuke ili same primene proizvoda. Njeni zapisi i ostale potrebne radnje moraju se održavati.

Radi lakšeg razumevanja značenja aktivnosti verifikacije i validacije, treba se samo prisetiti sledećih pitanja i odgovora u vezi ovih pojmoveva.

Verifikacija:

- > "Are we building the product right?"
- > Provera da li softver zadovoljava željenu funkcionalnost i postavljene zahteve.

Validacija:

- > "Are we building the right product?"
- > Provera da li softver zadovoljava stvarne potrebe korisnika (ili ulagača).
- > Validacija je važna jer postavljeni zahtevi ne definišu uvek precizno stvarne potrebe i želje korisnika softvera.

Cilj V & V je da se sa određenom sigurnošću pokaže da softver služi svrsi. Potreban stepen sigurnosti V & V zaljučaka zavisi od:

- Svrhe softvera - što je bitnije da softver bude pouzdan, to je potrebno da nivo sigurnosti V&V zaključaka bude viši.
- Očekivanja korisnika - korisnicima ne mora biti od presudne važnosti pouzdanost nekog softvera.
- Ekonomskog okruženja - nekada je bitno pojaviti se prvi na tržištu i po cenu manje pouzdanosti softvera.

ŠTA OBEZBEĐUJU VERIFIKACIJA I VALIDACIJA (V&V)?

Napori V&V teže obezbeđenju kvaliteta ugrađenog u softver kao i da softver ispunjava korisničke zahteve” (IEEE1059-93).

Radi uproštenosti, verifikacija i validacija (V&V) se ovde razmatraju kao jedna tema, pre nego dve posebne teme kao u standardu (IEEE12207.0-96). „Softverska V&V predstavlja disciplinovan pristup procene softverskih proizvoda tokom životnog ciklusa proizvoda.

Napori V&V teže obezbeđenju kvaliteta ugrađenog u softver kao i da softver ispunjava korisničke zahteve” (IEEE1059-93).

V&V se direktno odnosi na kvalitet proizvoda softvera i koristi tehnike testiranja koje mogu da lociraju defekte, tako da se oni mogu ispitati. Takođe ocenjuju i međuproizvode, i ipak, pri ovom kapacitetu, međukorake procesa životnog ciklusa proizvoda. Proces V&V određuje da li razvijeni proizvodi ili oni koji su dobijeni aktivnostima održavanja, ispunjavaju zahteve koje je ta aktivnost trebalo da ispuni, i da li finalna verzija softvera ispunjava predviđenu svrhu i zahteve korisnika. Verifikacija je pokušaj obezbeđenja da je proizvod pravilno napravljen, u smislu da izlazni proizvodi aktivnosti ispunjavaju specifikacije date u prethodnim aktivnostima. Validacija predstavlja pokušaj da se obezbedi da je pravi proizvod napravljen, odnosno, da proizvod ispunjava svoju određenu svrhu.

Oba procesa, i proces verifikacije i proces validacije, počinju u ranoj fazi razvoja ili fazi održavanja. Oni omogućavaju ispitivanje najvažnijih karakteristika proizvoda u odnosu na prethodni oblik proizvoda i specifikaciju koja mora da se zadovolji. Svrha planiranja V&V je da se obezbedi da je svaki resurs, uloga, kao i odgovornost, jasno određena. Rezultujući plan V&V dokumentuje i opisuje različite resurse, njihove uloge i aktivnosti, kao i tehnike i alate koji će biti korišćeni.

Razumevanje različitih svrha svake od V&V aktivnosti pomoći će u pažljivom planiranju tehniku i resursa koji su potrebni da bi ispunili svoje svrhe. Standardi (IEEE1012-98:s7 and IEEE1059-93: Appendix A) određuju šta obično ide u V&V plan.

Plan se, takođe, odnosi i na menadžment, komunikacije, politiku i procedure V&V aktivnosti i njihovu interakciju, kao i izveštavanje o defektima i zahtevima dokumentacije.

ORGANIZACIJA PROCESA TESTIRANJA PRI VERIFIKACIJI I VALIDACIJI

Za proces testiranja moraju biti realizovani standardizovani propisi i procedure koje definišu način rada i aktivnosti test odeljenja kao i svih učesnika u razvoju.

Proces testiranja se mora odvijati tokom svih faza životnog ciklusa. Za proces testiranja moraju biti realizovani standardizovani propisi i procedure koje definišu način rada i aktivnosti odeljenja za testiranje kao i svih učesnika u razvoju. Sve aktivnosti članova tima koji

učestvuju u realizaciji procesa testiranja moraju biti dokumentovane. Definisanje aktivnosti mora da pokrije sledeće:

1. Odgovornosti i zaduženja za planiranje testa, izvršenje i evaluaciju se eksplicitno moraju zadati.
2. Ciljevi testa, tipovi testa koji će se primeniti, kao i raspored i zaduženja kod testiranja moraju biti jasni i dokumentovani u Planu testiranja.

Regresiono testiranje je vrsta testiranja u kome se na osnovu jednom razvijenog test primera više puta sprovodi testiranje softvera (tipično posle neke izmene u softveru, da bi se utvrdilo da nije došlo do kvarenja funkcija softvera koje nisu obuhvaćene izmenom).

Testiranje softvera se odvija iz dvaугла:

1. Testiranje softvera prema tipu testova na osnovu zahteva za kvalitetom softvera tzv. testiranje višeg reda: testiranje sigurnosti, testiranje upotrebljivosti, test opterećenja itd..
2. Testiranje prema fazi razvoja softvera ili prema nivou složenosti (jedinično testiranje na nivou modula ili komponente, testiranje integracije, sistemsko testiranje, prijemno testiranje i sl.)

Tako na primer, prema nivou složenosti strukture softvera **jedinično testiranje** (engl. **unit testing**) primenjuje se na pojedine klase, module ili komponente programskog koda.

Tehnike jediničnog testiranja dele se na **tehnike crne kutije** i **tehnike bele kutije**.

Ovi metodi se obično kombinuju pri primeni. **Integraciono testiranje** primenjuje se na softverski podsistem ili sistem kao celinu.

TEHNIKE VERIFIKACIJE PREMA ZAHTEVIMA KVALITETA SOFTVERA - TIPOVI TESTIRANJA

U testiranja višeg reda spadaju: testiranje sigurnosti, upotrebljivosti, opterećenja, zagušenja, integriteta, instalacije...

U testiranja višeg reda spadaju :

- **testiranje sigurnosti** (engl. **security testing**) – da li su posmatrane funkcije dostupne onim i samo onim korisnicima kojima su namenjene
- **testiranje količine podataka** (engl. **volume testing**) – verifikovanje da li softver može obraditi veliku količinu podataka
- **testiranje upotrebljivosti** (engl. **usability testing**) – ocenjuju se estetski aspekti, konzistencija korisničkog interfejsa, korisnička dokumentacija i trenažni materijal
- **testiranje integriteta** (engl. **integrity testing**) – ocenjuje se robustnost (otpornost na otkaze), konzistentna upotreba resursa i slično
- **testiranje naprezanja** (engl. **stress testing**) – vrsta testa pouzdanosti sistema pod nenormalnim uslovima (ekstremna opterećenja, nedovoljno memorije ili drugih resursa, servisi koji nisu raspoloživi itd)

- **uporedno testiranje** (engl. **benchmark testing**) – vrsta testa performansi koja upoređuje novi nepoznati sistem sa poznatim referentnim sistemom
- **testiranje zagušenja** (engl. **contention testing**) – proverava da li softver može da zadovolji višestrukе zahteve različitih aktera za istim resursom
- **testiranje opterećenja** (engl. **load testing**) – vrsta testa performansi, kojim se procenjuju operativni limiti nepromenljivog sistema pod različitim opterećenjima, ili različitih konfiguracija sistema pri istom opterećenju. Merene veličine su najčešće protok i vreme odziva (srednja i vršna vrednost)
- **testiranje konfiguracije** (engl. **configuration testing**) – testira ponašanje softvera u različitim hardversko/softverskim okruženjima
- **testiranje instalacije** (engl. **installation testing**) – testira instaliranje softvera na različitim sistemima i u različitim situacijama (npr. prekid napajanja ili nedovoljno prostora na disku).

ZADUŽENJA I ODGOVORNOSTI

Sve aktivnosti članova tima koji učestvuju u realizaciji procesa testiranja moraju biti dokumentovane.

Prilikom testiranja:

1. *Odgovarajući materijal (dokumenta, test primeri i drugo) se čuvaju i nalaze se pod kontrolom repozitorijuma za upravljanje konfiguracijama (engl. Configuration Management, skraćeno CM).*
2. *Test materijal je podložan periodičnim aktivnostima kontrole kvaliteta.*

Nadalje u ovom odeljku, dat je opis uloga koje se tiču procesa testiranja; nisu uključene uloge koje se tiču procesa razvoja.

Menadžer kvaliteta odgovaran je za planiranje testa, kako dugoročno (u okviru menadžmenta projekta), tako i kratkoročno. Vrši kontrolu izvršenja, eventualne korekcije i evaluacije testiranja. Obezbeđuje informacije menadžmentu projekta o rezultatima rada test grupe. Vrši proveru korektnosti planova i rokove realizacije. Obezbeđuje neophodni materijal članovima zaduženim za razvoj i sprovođenje testiranja pre i nakon testiranja, izveštaje o testiranju, zbirne izveštaje o greškama, neophodnu dokumentaciju.

Član tima za razvoj testova učestvuje u razvoju **test primera** (engl. **test cases**) i **serije testova** (engl. **test suites**).

Član tima za sprovođenje testiranja vrši testiranje pojedinih modula ili celog softvera primenom testova iz Test specifikacije. Šalje izveštaje sa **testiranja**, prijavljuje otkaze.

Menadžer zahteva za promenama procenjuje pojedinačne i zbirne rezultate testova i na osnovu toga generiše zahteve za promenama, dodeljuje im prioritete i planira ih po iteracijama.

AKTIVNOSTI PROCESA TESTIRANJA

Aktivnosti procesa testiranja proizvoda su: planiranje, specifikacija, realizacija i evaluacija rezultata.

Uspešan završetak analize i prikupljanja zahteva, u kojoj su definisani svi funkcionalni zahtevi za proizvod u odgovarajućim dokumentima i modelima, jeste preduslov za početak ne samo faze projektovanja, već i za početak faze testiranja.

U narednim sekcijama dat je opis aktivnosti procesa testiranja proizvoda:

1. *planiranje testiranja*
2. *specifikacija testa*
3. *generisanje izveštaj o rezultatima testa*
4. *evaluacija rezultata.*

PLANIRANJE TESTIRANJA

Zahtevi za testiranjem identikuju šta će se testirati.

Tokom planiranja testiranja potrebno je da se obave sledeće aktivnosti:

1. *Identifikacija zahteva za testiranjem*
2. *Procena rizika i uspostavljanje prioriteta*
3. *Strategija testiranja*

Sledi opis pojedinih elemenata plana testiranja.

Identifikacija zahteva za testiranjem: Zahtevi za testiranjem identikuju šta će se testirati. Na primer: zahtevi za testiranjem funkcija, kao što im ime kaže, izvedeni su iz opisa funkcija posmatranog softvera. Minimalno svaki slučaj upotrebe treba da proizvede bar jedan zahtev za testiranjem. Zahtevi za testiranjem performansi izvode se iz specifikacije performansi posmatranog softvera. Tipično su performanse izražene preko vremena odziva i potrošnje resursa pri različitim režimima rada. Zahtevi za pouzdanošću izvode se iz više izvora, tipično iz nefunkcionalnih specifikacija softvera, vodiča za dizajniranje korisničkog interfejsa, projektnih i implementacionih pravilnika.

Procena rizika i uspostavljanje prioriteta: Procena rizika počinje ustanovljavanjem prioriteta testiranja. Slučajevi korišćenja ili komponente koji predstavljaju najrizičnije u pogledu otkaza ili imaju veliku verovatnoću otkaza treba da budu među prvim testiranim. Sledеći korak jeste da se odredi operacioni profil softvera koji se razmatra. Tipično, što je veći broj upotreba nekog slučaja upotrebe ili neke komponente od strane različitih aktera, biće veći indikator operacionog profila.

Finalni korak u proceni rizika i uspostavljanju prioriteta, jeste da se identifikuju i opišu pojedini slučajevi korišćenja ili komponente upotrebom indikatora prioriteta kao što su:

- H - mora se testirati
- M - trebalo bi testirati, ali tek pošto se obavi testiranje svih H celina

- L – možda će se testirati, ali tek posle svih H i M celina

Strategija testiranja: *Strategija testiranja* opisuje generalni pristup i ciljeve određenih aktivnosti testiranja. Dobra strategija trebalo bi da sadrži sledeće elemente:

- *tip testa koji će se sprovesti i cilj testiranja*
- *faza razvoja u kojoj će se testiranje primeniti*
- *tehnika testiranja*

SPECIFIKACIJA TESTOVA

Za realizaciju Specifikacije testova mogu se koristiti: Studija (ako postoji), specifikacija zahteva, model slučajeva upotrebe, razvojni plan, projektna dokumentacija i postojeći projektni modeli.

U ovoj fazi potrebno je identifikovati i opisati skup test primera dovoljnih za verifikaciju proizvoda i test procedura koje opisuju način izvršenja test primera.

Za realizaciju Specifikacije testova mogu se koristiti: Studija (ako postoji), specifikacija zahteva, model slučajeva upotrebe, razvojni plan, projektna dokumentacija i postojeći projektni modeli.

Specifikacija testa sadrži sledeća poglavlja (nadalje je dat i kratak opis sadržaja poglavlja):

- 1. **Beleška specifikatora testa:** Ovaj odeljak treba da sadrži opis za konkretni test primer, svrhu testa, tip testiranja i opis primenjene metodologije, eventualno dati potrebne preduslove za izvršenje, hardverske i softverske zahteve (ako postoje).
- 2. **Radnje/Ulazi:** Ovaj odeljak sadrži, za konkretni test primer, opis radnji, odnosno vrednosti koje treba obaviti/uneti kao ulaz.
- 3. **Očekivani Odzivi/Izlazi (Test Oracle problem):** U ovom odeljku, za konkretni test primer, dati opis ili slike odziva koji je rezultat primene radnji/ulaza opisanih u prethodnom odeljku. Podrazumeva se da se daju ispravni odzivi/izlazi, a ne ono što se dobije u testu.
- 4. **Završne radnje:** Ova sekcija sadrži informacije o svim neophodnim aktivnostima/radnjama koje su potrebne za "smirivanje" odnosno završetak celog testa, na primer: zatvaranja projekata, brisanje.
- 5. **Realizacija testiranja:** Faza implementacije test procedura, tj. izvršenja testa, se može ponavljati za svaki build. Moguće je manuelno sprovoditi testiranje, ili upotrebom automatizovanih alata (pri čemu se moraju razviti i održavati test skriptovi koji odgovaraju test slučajevima).

IZVEŠTAJ O REZULTATIMA TESTA

Sastoji se od sledećih sekcija: generalna procena testiranog softvera, uticaj test okruženja, preporučena poboljšanja, detalji o rezultatima testa, test LOG

Izveštaj o rezultatima testiranja treba da bude podeljen na sledeće sekcije kako bi se dao pregled rezultata testiranja:

Generalna procena testiranog softvera: Ova sekcija treba da:

1. Obezbedi generalnu procenu softvera preko rezultata testa datih u ovom izveštaju
2. Identifikovati sve preostale nedostatke, ograničenja ili prinudna rešenja koja su detektovana testiranjem. Za obezbeđivanje informacija o nedostacima koristiti izveštaje o problemima i promenama.

Uticaj test okruženja: Ova sekcija treba da obezbedi procenu načina kako se test okruženje može razlikovati od radnog/operativnog okruženja i uticaj tih razlika na rezultate testa.

Preporučena poboljšanja: Ova sekcija treba da obezbedi preporučena poboljšanja u specifikaciji, upotrebi ili testiranju softvera koji se posmatra.

Detalji o rezultatima testa: Ova sekcija treba da bude podeljena na podsekcije navedene kasnije tako da predstavlja detaljan opis rezultata primene svakog testa. Podsekcije su identifikovane jedinstvenim identifikatorom testa i dati su pregled rezultata testa, pronađeni problemi i odstupanja od predviđenih procedura testiranja.

Test LOG: Ova sekcija treba da obezbedi hronološki zapis test događaja. Preporuka je da se izradi Log obrazac u obliku tabele gde bi se upisivali događaji. Takav test Log treba da sadrži:

- Datum, vreme, lokacija gde je test izvršen
- Hardverska i softverska konfiguracija koja je korišćena za svaki test, uključujući (ako je dostupno) podatke o modulu, modelu, serijskom broju, proizvođaču itd. za hardver i obavezno broj verzije i ime softverske(ih) komponente(i) koja se koristi.
- Vreme i datum svake test aktivnosti, identitet članova tima koji realizuju testiranje.

EVALUACIJA REZULTATA I POBOLJŠANJA PROCESA TESTIRANJA

Poboljšanja procesa testiranja je konstantan proces zajedno sa svim drugim elementima razvoja softvera.

Na kraju neke faze testiranja radi se evaluacija rezultata testiranja. U ovoj fazi treba definisati mere (metrike) za evaluaciju kvaliteta testiranja i realizovati Izveštaj o proceni kvaliteta testiranja. U ovoj fazi se može uraditi i analiza pronađenih defekata sa ciljem da se ukaže na česte i zajedničke greške razvojnog dela tima, da se preporuče naknadne aktivnosti (kursevi, doobuka, promena alata i sl.).

Gene Krinz, direktor svemirskog programa u NASI je konstatovao da postoji problem u softverskom inženjerstvu po pitanju kvaliteta softvera tvrdnjom da: "Ne možete testiranjem ugraditi kvalitet softvera". Ne može se ugraditi kvalitet, niti otkloniti problemi u kvalitetu softvera samo aktivnostima testiranja softvera.

Ovo postaje očigledno ukoliko zahtevi za kvalitetom nisu postavljeni u zahtevima za projekat softvera, u fazi analize, dizajna ili implementacije, tako da se testiranjem softvera ne mogu ni ugraditi niti otkloniti problemi kvaliteta softvera kroz klasičan pristup testiranja softvera. Jedan od ciljeva uspešnih procesa razvoja je *evaluacija kvaliteta softverskog proizvoda*. Dok se nekoliko atributa kvaliteta softvera može oceniti aktivnostima u sklopu testiranja, dobar deo njih mora se ugraditi kroz dizajn softverskog proizvoda.

Poboljšanja procesa testiranja je konstantan proces zajedno sa svim drugim elementima razvoja softvera. Angažuju se sve veći resursi kako velikih tako i malih kompanija u proces testiranja i njegovo unapređenje.

Postoji više aspekata koji doprinose poboljšanju procesa testiranja:

- obuka kadrova za proces testiranja
- automatizacija procesa testiranja
- razvoj novih alata
- razvoj novih modela testiranja
- integracije procesa merenja parametara efikasnosti i efektivnosti procesa testiranja
- analize slabih i jakih strana postojećeg procesa testiranja
- identifikacije rizika i njihovih posledica na uspeh procesa razvoja

▼ Poglavlje 5

Tehnike testiranja

KARAKTERISTIKE I VRSTE TEHNIKA TESTIRANJA

Klasifikacija je zasnovana na tome kako su testovi generisani od strane softver inženjera (njegove intuicije i iskustva), specifikacije, strukture koda, polja primene i dr.

Jedan od ciljeva testiranja je otkrivanje što je moguće više potencijalnih grešaka i mnoge tehnike su razvijene s ciljem da pokušaju da obore program, putem pokretanja jednog ili više testova odabralih iz identifikovanih klasa izvršavanja.

Osnovni princip u tim tehnikama je biti sistematican koliko god je moguće u identifikovanju reprezentativnog skupa ponašanja programa, kao na primer: razmatranje podklasa ulaznog domena, scenarija, stanja i toka podataka.

Klasifikacija je zasnovana na tome kako su testovi generisani od strane softver inženjera (njegove intuicije i iskustva), specifikacije, strukture koda, polja primene, kao i same prirode aplikacije.

Nekada se ove tehnike klasifikuju kao white box (glass box), ako se test zasniva na informacijama o tome kako je softver dizajniran ili kodiran, ili black box, ako se test slučajevi zasnivaju na ponašanju ulaz/izlaz. Postoje tehnike koje kombinuju upotrebu dve ili više tehnika. Same tehnike ne koriste se jednakod strane svih inženjera.

TESTIRANJE CRNE KUTIJE, BELE KUTIJE I SIVE KUTIJE

Kod testiranja bele kutije je potrebno poznavanje interne strukture programa, kod crne kutije nije dok je kod sive kutije potrebno ali samo za dizajniranje test slučajeva

Karakteristike testiranja crne kutije (engl. Black box testing) su:

- Tehnike dizajnirane bez poznavanja interne strukture programa i dizajna
- Zasnovane na funkcionalnim zahtevima
- Često nazivano i funkcionalno testiranje

Vrste Black box testiranja su:

- Ekvivalentno particionisanje (eng. Equivalence partitioning)
- Testiranje graničnih vrednosti (eng. Boundary-value analysis)

- Cause-effect graphing
- Pogađanje grešaka (eng. Error guessing)

Karakteristike testiranja bele kutije (engl. White box testing) testiranja koje je poznato i kao jasno testiranje (engl. clear box testing) ili strukturalno testiranje (engl. structural testing) su:

- Ispitivanje internog dizajna programa
- Zahteva detaljno poznavanje strukture

Vrste White box (glassbox) testiranja su:

- Pokrivanje iskaza (eng. Statement coverage)
- Pokrivanje odluka (eng. Decision coverage)
- Pokrivanje uslova (eng. Condition coverage)
- Njihove kombinacije (eng. Decision-condition coverage, Multiple-condition coverage i dr.)

Testiranje sive kutije (engl. Gray box testing) je softverski metod testiranja koji je kombinacija black box testing metoda i white box testing metoda. Za razliku od testiranja crne kutije, gde je unutrašnja struktura programa koji se testira nepoznata testeru i testiranja bele kutije kod koje je poznata unutrašnja struktura, kod testiranja sive kutije, unutrašnja struktura je delimično poznata. To podrazumeva pristup unutrašnjim strukturama podataka i algoritmu za potrebe dizajniranja test slučajeva, ali se testiranje obavlja na korisničkom nivou ili nivou crne kutije. Grey box testing je nazvan tako što je softverski program u očima testera poput sive/poluprozirne kutije; unutar koje se delimično može videti.

TEHNIKE TESTIRANJA ZASNOVANE NA INTUICIJI SOFTVER INŽENJERA I NA SPECIFIKACIJAMA

Tehnike testiranja zasnovane na intuiciji softver inženjera: ad hoc i istraživačko testiranje; Tehnike zasnovane na specifikaciji: ekvivalentno particonisanje i testiranja graničnih vrednosti

Pored prethodno navedenih, postoje i druge različite tehnike testiranja kao što su:

Tehnike testiranja zasnovane na intuiciji softver inženjera gde spadaju:

1. Ad hoc testiranje: Jedna od najraširenijih tehnika testiranja. Testovi se izvode na osnovu veština, intuicije i iskustva softver inženjera sa sličnim programima. Ad hoc testiranje može biti korisno radi identifikacije specijalnih testova, koji nisu lako prepoznati formalnim tehnikama.
2. Istraživačko testiranje (engl. Exploratory testing): definise se kao simultano učenje, dizajniranje testa i njegovo izvršavanje. Drugim rečima, test se ne definiše unapred u uspostavljenom test planu, već se dinamički dizajnira, izvršava i modifikuje.

Tehnike zasnovane na specifikaciji (Specification Based Techniques) gde spadaju:

1. Ekvivalentno particonisanje (engl. Equivalence partitioning): ulazni domen je podeljen u kolekciju podskupova ili ekvivalentnih klasa, koji se smatraju

ekvivalentnim prema specifičnoj relaciji, i reprezentativan skup testova (nekada samo jedan) je uzet iz svake klase.

2. **Testiranja graničnih vrednosti** (engl. **Boundary-value analysis**): test slučajevi su izabrani na ili blizu granica ulaznog domena varijabli, sa idejom da se najveći broj greški koncentriše blizu ekstremnih vrednosti ulaza. Proširenje ove tehnike je testiranje robusnosti, gde se test slučajevi takođe biraju izvan ulaznog domena varijabli, radi testiranja robusnosti programa na neočekivane ili pogrešne ulaze.

TEHNIKE ZASNOVANE NA KODU I FAULT-BASED TEHNIKE

Tehnike zasnovane na kodu: Control-flow-based criteria i Data flow-based criteria; Fault-based tehnike: Error guessing i Mutation testing

Tehnike zasnovane na kodu (eng. **Code Based Techniques**)

1. **Control-flow-based criteria:** Imaju za cilj pokrivanje svih iskaza ili blokova iskaza u programu ili njihovih specifičnih kombinacija. Najsnažniji control-flow-based kriterijum je testiranje putanje, koji ima za cilj izvršavanje svih ulazno/izlaznih putanja. Pošto se teoretski ne može dostići puna primena testiranja putanje zbog petlji u programu, u praksi se koriste manje jaki kriterijumi, kao što su statement testing, branch testing, i condition/decision testing. Adekvatnost ovakvih testova se meri u procentima, npr. kada su testom sva grananja izvršena najmanje jednom, kaže se da je dostignuto 100% pokrivenost grananja.
2. **Data flow-based criteria:** U ovakovom tipu testiranja ističe se informacija kako su varijable programa definisane, korišćene i terminisane.

Fault-based tehnike: Fault-based tehnike testiranja određuju test slučajeve specifično ciljane ka otkrivanju kategorija predefinisanih grešaka.

1. **Error guessing:** Test slučajevi se posebno dizajniraju od strane softver inženjera pokušavajući da otkriju najverodostojnije greške u datom programu. Dobar izvor informacija u tom slučaju je istorija otkrivenih grešaka u ranijim projektima, kao i ekspertiza softver inženjera.
2. **Mutation testing:** Mutant je manje modifikovana verzija programa pod testom, na osnovu malih sintaksnih grešaka. Svaki test slučaj testira i original i sve mutante programa, i posmatra se uspešnost test slučaja u identifikovanju razlike između programa i mutanta.

USAGE-BASED TEHNIKE I TEHNIKE ZASNOVANE NA PRIRODI APLIKACIJE

Kod usage-based tehnike testovi se dizajniraju na osnovu očekivane relativne upotrebe dok se tehnike zasnovane na prirodi aplikacije primenjuju za specijalizovane softvere

Usage-based tehnike : U testiranju radi evaluacije pouzdanosti, test okruženje mora reprodukovati operaciono okruženje softvera što je više moguće. Ideja je na osnovu posmatranja test rezultata predvideti buduću pouzdanost softvera kada se on bude zaista koristio. Testovi se dizajniraju na osnovu očekivane relativne upotrebe

Tehnike zasnovane na prirodi aplikacije: Prethodno navedene tehnike primenjuju se na sve tipove softvera. Za specijalizovane tipove softvera postoje i specijalizovana test polja, zasnovana na prirodi aplikacije kao što su: **Objektno-orientisano testiranje, Komponent-bazirano testiranje, Web-bazirano testiranje, GUI testiranje, Testiranje konkurentnih programa, Testiranje sistema u realnom vremenu, Testiranje bezbednosno kritičnog softvera, Kombinovane tehnike.**

MERENJA U TESTIRANJU SOFTVERA

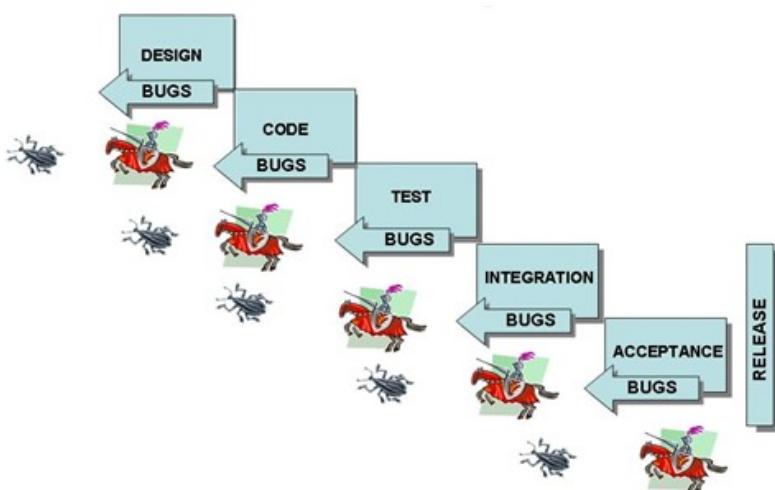
Merenja se često posmatraju kao instrument analize kvaliteta. Merenja se koriste i radi optimizacije planiranja i izvršavanja testa.

Dva pristupa test izborima se ne posmatraju kao alternativna, već kao komplementarna aktivnost. Koriste različite izvore informacija i namenjeni su otkrivanju različitih problema. Mogu da se koriste u kombinaciji, u zavisnosti od raspoloživog budžeta.

Takođe, test slučajevi mogu da se izaberu na deterministički način, prema navedenim različitim tehnikama ili random izvlačenjem iz skupa distribucije ulaza, što se često koristi kod testiranja pouzdanosti. U takvim slučajevima je uvek dobro napraviti analizu uslova koji čine jedan pristup efikasnijim od drugog.

Test-related merenja: Potrebno je napraviti jasnu razliku između test-related merenja, koje pruža evaluaciju programa pod nekim testom zasnovanom na posmatranju test izlaza, i onih merenja koji evaluiraju izabrani skup testova. Merenja se često posmatraju kao instrument analize kvaliteta. Merenja se koriste i radi optimizacije planiranja i izvršavanja testa.

Broj i tok grešaka u softveru: Broj grešaka koje se javljaju u softveru kumulativno se uvećava do faze isporuke proizvoda kupcu. Jedan od ciljeva menadžmenta kvaliteta softvera prikazan je na sledećoj slici, otkriti što više grešaka u samom početku razvoja i okrenuti tok pojavljivanja grešaka.



Slika 5.1 Način smanjenja broja grešaka [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 6

Grupna vežba - Testiranje slučajeva korišćenja

TESTIRANJE ZA BAZI SLUČAJEVA KORIŠĆENJA

Spada u statičko testiranje sistema koje se koristi za proveru nedostataka u softverskoj aplikaciji bez izvršavanja koda.

Statičko testiranje je tehnika softverskog testiranja koja se koristi za proveru nedostataka u softverskoj aplikaciji bez izvršavanja koda. Statičko testiranje se vrši kako bi se izbegle greške u ranoj fazi razvoja, jer je tada lakše identifikovati greške i rešiti ih. Takođe pomaže u pronalaženju grešaka koje možda neće pronaći dinamičko testiranje.

Za razliku od statičkog, vrši se i dinamičko testiranje koji proverava aplikaciju kada se kod pokreće.

Statičko ispitivanje se vrši iz sledećih razloga

- *Ranog otkrivanja i ispravljanja grešaka*
- *Smanjenja troškova i vremena testiranja*
- *Radi poboljšanja produktivnosti u razvoju*
- *Radi smanjenja grešaka u kasnijoj fazi ispitivanja*

U okviru statičkog testiranja testiraju se između ostalog :

- *Unit Test Case-ovi*
- *Dokument poslovnih zahteva (engl. Business Requirements Document (BRD))*
- *Slučajevi korišćenja (engl. Use Cases)*
- *Sistemski / funkcionalni zahtevi*
- *Prototip*
- *Dokument specifikacije prototipa*
- *Rečnik koji sadrži polja baze podataka*
- *Traceability Matrix dokument*
- *Korisnička dokumentacija: User Manual/Training Guides*
- *Strategija plana testiranja*

U ovoj vežbi će biti prikazano testiranja na bazi slučajeva korišćenja i to na primeru softvera za regulisanje saobraćaja u jednosmernom tunelu.

Slučajevi korišćenja opisuju put koji korisnici obično koriste da bi izvršili određeni zadatak u sistemima seoni često koriste kao način zapronalaženje grešaka u korišćenju tih putanja.

SPECIFIKACIJA SOFTVERA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU

Hardverska i softverska specifikacija sistema predstavljaju detaljan uvid u delove sistema, komponente i podsisteme.

U ovoj vežbi će biti testiran softver za regulisanje saobraćaja u jednosmernom tunelu za koji su data softverska i hardverska specifikacija. Testiranje će se izvršiti na bazi specifikacije i na bazi implementacije rešenja.

Softverska specifikacija sistema:

Program treba da reguliše saobraćaj u jednosmernom tunelu, koristeći senzore koji će očitavati broj vozila koja ulaze u tunel, kao i dopler semafor koji meri i prikazuje brzinu automobila koji ulaze u tunel. U cilju bezbednosti, u tunelu se ne sme nalaziti više od približno N vozila u datom trenutku čija vrednost će se odrediti na osnovu informacije saobraćajnih inženjera uključenih u ovaj projekat. Na početku projekta se zna da N zavisi od statistike gustine saobraćaja, modela automobila zbog različite dužine i brzine istih. Semafor na ulazu u tunel kontroliše prliv vozila, dok detektori vozila na ulazu i izlazu iz tunela prate tok vozila. Detektor na ulazu je složen senzor u kome je integriran dopler radar za merenje brzine i dužine vozila (dužina je jednaka proizvodu brzina i vremena praćenja doplerovim radarom tog vozila) i displej za prikaz brzine vozila i poruke o smanjenju brzine koju softver proračunava u slučaju da je u zagrebanje u tunelu. Detektor na izlazu ima zadatak samo da broji vozila koja su napustila tunel o čemu šalje komunikacionim linkom informaciju računaru. Procesi kontrolišu ulaz i izlaz iz tunela i na osnovu gustine i statistike saobraćaja menjaju svetla na semaforima. Vreme predviđeno za očitavanje senzora biće prvobitno podešeno na 10 sekundi, a ukoliko to bude potrebno (zahtevano od strane saobraćajnih inženjera i na osnovu procena u fazama merenja i testiranja) može se dodatno korigovati. Kada tunel postane pun (i na semaforu se upali crveno svetlo), proces koji nadgleda ulaz u tunel treba da signalizira da je tunel popunjeno kao i upozorenje vozilima da smanje brzinu ili da se zaustave promenom svetla na semaforu u crveno. Kada tunel postane prazan, proces koji nadgleda izlaz iz tunela treba da uz pomoć senzora pošalje signal da je tunel dovoljno ispravljen i promeni svetlo na ulazu u zeleno i tako omogući ulaz vozila koja čekaju na zeleno svetlo.

Vreme izrade grupne vežbe 45 minuta

HARDVERSKA SPECIFIKACIJA SOFTVERA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU

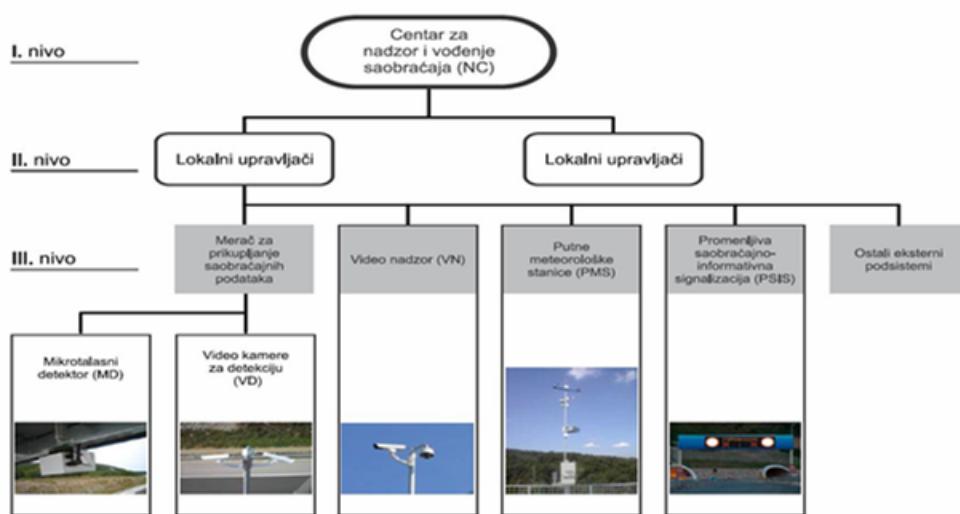
Hardverska šema sistema predstavlja hardverske komponente i veze između njih.

Celokupan sistem nadzora saobraćaja u tunelu sastoji se iz nekoliko podistema (prikazanih na slici 6.1):

- Automatska detekcija saobraćaja ADS
- Nadzorni upravljački sistem tunela
- Sistem za video nadzor saobraćaja
- Sistem nadzora i vođenje saobraćaja

Predstavljena je slika na kojoj se vidi nacrt hardverskog dela sistema. Tu se nalaze:

- Senzor prikupljanja saobraćajnih podatka (senzor ulaska u tunel sa video kamerama)
- Saobraćajna signalizacija
- Eksterni podsistemi



Slika 6.1.1 Hardverska šema sistema [Izvor: NM SE321 - 2020/2021.]

HARDVERSKA SPECIFIKACIJA SOFTVERA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU - NASTAVAK

Softver za regulisanje saobraćaja u jednosmernom tunelu predstavlja nadzorni sistem u kome je jako bitna pravovremena komunikacija između hardverskih i softverskih komponenata.

Sistem za automatsku detekciju saobraćaja (senzor sa dopler semaforom prikazan na slici 6.2) uz pomoć video slike iz kamere beleži vozila koja ulaze, njihovu brzinu, dužinu vozila, suprotan smer kretanja ili zaustavljeni vozilo na ulazu i preko ADS dela aplikacije na radnoj stanici upozorava operatera na ovu opasnost i upravlja ostalim sistemima (semaforima). Kod nadzornog sistema upravljanja dolazi do detekcije saobraćaja i eventualnih upozorenja (na požar, smanjena vidljivost na ulazu) paleći alarne i svetla na semaforima koja onemogućuju ulazak vozila u tunel koristeći prethodno definisani algoritam (trepćuća svetla, LED indikatore i table).

Sistem za video nadzor saobraćaja prikazuje sliku sa kamera i senzora koji pokrivaju područje tunela i upisuju u bazu podataka za eventualna istraživanja ili proveravanja statusa.

Sistem nadzora i vođenje saobraćaja se nalaze ispred tunela ili na trasi puta pre tunela gde

upozoravaju korisnike tunela na eventualne izmene u planu ili prikazuju brzinu kretanja uz obaveštavanje sistema o navedenim izmenama. Na ovaj način korisnici tunela (vozači imaju jasnu sliku pre ulaska u tunel šta ih očekuje.



Slika 6.1.2 Senzor merenja brzine i dužine vozila [Izvor: NM SE321 - 2020/2021.]

DIJAGRAM SLUČAJA KORIŠĆENJA SISTEMA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU

Funkcionalnosti sistema predstavljaju sve mogućnosti koje korisnik može da ima i kao takve moraju biti testirane kako bi se obezbedila što veća pouzdanost sistema.

Funkcionalnost sistema za regulisanje saobraćaja u jednosmernom tunelu koje treba da budu testirane se može prikazati sledećim dijagramom slučajeva korišćenja:

- Proveravanje svetla na semaforu
 - Slučaj očitavanja zelenog svetla na semaforu
 - Ulaz u tunel
 - Očitavanje na senzoru ulaza (podrazumeva merenje brzine i dužine vozila)
 - Izlazak iz tunela
 - Očitavanje na izlaznom senzoru
 - Slučaj očitavanja crvenog svetla
 - Zaustavljanje na definisanoj udaljenosti
 - Očitavanje zelenog svetla

Ovde se svaka aktivnost u izvršenju funkcionalnosti sistema tretira kao jedan slučaj korišćenja - use case. Ovaj dijagram slučaja korišćenja ima normalan (standardni) tok za slučaj očitavanja zelenog svetla na semaforu i alternativni tok za slučaj očitavanja crvenog svetla

TESTIRANJE SLUČAJA KORIŠĆENJA SISTEMA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU

Generisanje test slučajeva počinje u fazi modelovanja gde se na osnovu slučajeva korišćenja kreiraju testovi bilo na osnovu standardnog toka bilo da je bazirano na alternativnom toku.

Generisanje test slučajeva počinje u fazi modelovanja gde se na osnovu dijagrama slučajeva korišćenja kreiraju testovi bilo na osnovu standardnog toka bilo da je bazirano na alternativnom toku.

U slučaju testiranja dijagrama slučaja korišćenja sistema za regulisanje saobraćaja u jednosmernom tunelu, testira se put kroz ulaz u tunel (očitavanje na senzoru ulaska, prolazak i izlazak iz tunela) uz standardne sistemske korake koji prate prolazak korisnika. Ukoliko dođe do alternativnog scenarija obradiće se u okviru test slučaja koji će biti modifikacija standardnog procesnog puta. Često se dešava da su testiranja korišćenjem slučaja korišćenja direktno povezana sa mogućnostima testera i njegovim poznavanjem sistema i načina implementacije u sledećoj fazi. Tester na osnovu svog iskustva kreira i modifikuje slučajeve uglavnom počevši od glavnih procesa u sistemu. Ukoliko se primeti greška ili alternativni scenario poseže se za istraživanjem novog procesa u sistemu i testiranje njegovih funkcionalnosti. UseCase modeli mogu biti modelovani tako da zajedno čine grupu na kojoj se radi test ili se mogu pojedinačno testirati koristeći samo interakciju između korisnika i tog dela sistema. Koristeći sekvencijalne dijagrame na kome se vide procesi koji se odvijaju kako je u fazi modelovanja zamišljeno, može se definisati i rešiti pravac testiranja. Konkretno, vezano za UseCase testiranje, prilikom kreiranja test strategije mogu da baziraju na osnovim softverskim principima dok su greške koje se mogu javiti su sledeće:

Greška u scenariju korišćenja

Izvršenje slučajeva korišćenja zavisi od nekoliko operativnih scenarija. Svaki scenario odgovara za drugačiju sekvencu od dela za poruke u navedenom dijagramu. Za neki od datih scenario korišćenja, sekvence ili poruke nije potrebno pratiti navedeni put zavisno od uslova evaluacije

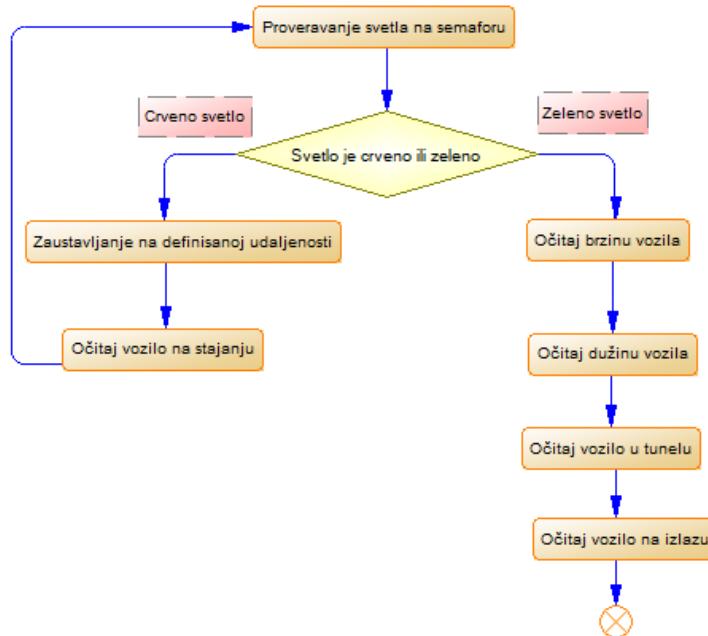
Greška prilikom sinhronizacije

Vezama povezani slučajevi korišćenja u okviru jednog dijagrama mogu direktno narušiti procese unutar sistema. Ukoliko jedan od procesa zakasni ili zakaže doći će do kašnjenja svih procesa u celom sistemu i navedena greška će se odraziti na sve procese.

DIJAGRAM AKTIVNOSTI

Dijagram aktivnosti na osnovu UseCase dijagrama se modeluje tako što se veze između svih slučajeva korišćenja kreiraju tako da svi slučajevi korišćenja međusobno povezani i da čine jednu

Na slici 6.3 je prikazan dijagramu aktivnosti koji odgovara dijagramu slučaju korišćenja sistema za regulisanje saobraćaja u jednosmernom tunelu. Prikazana su dva procesa, standardan scenario i alternativni scenario.



Slika 6.1.3 Dijagram aktivnosti izveden iz UseCase diagrama [Izvor: NM SE321 - 2020/2021.]

Standardni scenario podrazumeva sledeće aktivnosti sistema:

- PrikažiZelenoSvetlo
- OčitajBrzinuVozila
- OčitajDužinuVozila
- OčitajVozilouTunelu
- OčitajVozilonalzlasku

Alternativni scenario omogućava prikaz procesa ukoliko se na početnom semaforu prikaže crveno svetlo. U tom slučaju se prikazuje još jedna dodatna aktivnost "očitaj vozilo na stajanju" i nakon toga se proces vraća na deo gde senzor očitava ulazak vozila.

Sekvenca 1 (odgovara standardnom scenariju):

SEQ1.PrikaziZelenoSvetlo.OcitajBrzinuVozila.OcitajDuzinuVozila.OcitajVoziloUtunelu.OcitajVozilonalzlazu

Sekvenca 2 (odgovara alternativnom scenariju):

SEQ2.PrikaziCrvenoSvetlo.OcitajVoziloNaStajanju.PrikaziZelenoSvetlo.OcitajBrzinuVozila.OcitajDuzinuVozila

TESTIRANJE PUTANJA KOJI KORISNICI OBIČNO KORISTE U SISTEMU

Generisanje test slučajeva zavisi od kreiranih sekvenci. Kada se iz UseCase diagrama izvedu aktivnosti i svaka putanja i poruka između njih unese u diagram, nakon toga je jednostavno izve

1. Testiranje ulaska vozila u tunel i očitavanje na senzoru ulaska: Sekvenca ovog testa bi bila:

SQ01: VozacUlaziTunel. SenzorOčitavaUlazakVozačauTunel.

SenzorOčitavaBrzinuUlaskaVozača. SemaforPokazujeCrvenoSvetlo. SemaforPokazujeZelenoSvetlo

2. Testiranje ulaska vozača u tunel sa predviđenim nedostacma sistema.

Sekvenca ovog testa bi bila:

SEQ02: VozačNeUlaziTunel. SenzorNeOčitavaUlazakVozača. SemaforPrikazujeZelenoSvetlo.

SenzorbrzineNeOčitavaBrzinu. SemaforPokazujeCrvenoSvetlo

3. Testiranje prolaska vozila kroz tunel i očitavanje na senzoru izlaska:

Sekvenca ovog testa bi bila:

SEQ03: VozačProlaziKrozTunel. SenzorOčitavalzlazakVozilalzTunela. SistemBeležilzlazakVozača. SemaforNaUlazuPokazujeZelenoSvetlo

4. Testiranje prolaska vozila kroz tunel uz jasno definisane izmene: Sekvenca ovog testa bi bila:

SEQ04: VozačNijeZnalaOzTunela. SistemOčitavaDaSeVozačNijePojavioNaSenzorulzlaska. SistemŠaljeZnakDaSvetloNaSemaforuBudeCrveno. SistemPrikazujeGrešku

5. Testiranje crvenog svetla na ulasku u tunel: Sekvenca ovog testa bi bila:

SQ05: VozačOčitavaCrvenoSvetloNaUlazu. VozačSeZaustavljaNaOdređenojUdaljenosti. VozačČekaZelenoSvetlo. VozačUlaziUTunel

6. Testiranje crvenog svetla na ulasku u tunel uz predviđene izmene scenaria korišćenja: Sekvenca ovog testa bi bila:

SQ06: VozačNeOčitavaCrvenoSvetloNaUlazu. VozačProlaziSemaforBezStajanja. SenzorNeOčitavaUlaz

TESTIRANJE TOKA KOJI ODGOVARA STANDARDNOM SCENARIJU

Standardno UseCase testiranje se odnosi na popisivanje koraka koji su potrebni da bi se odradio jedan proces.

Test slučaja korišćenja u kome je prikazan jedan proces prolaska vozila kroz tunel i njegov izlazak iz tunela uz očitavanje na senzoru izlaska je prikazan na slici 6.4. Ovo je planom predviđen proces u okviru ovog sistema i svi rezultati koji su očekivani su se i desili prilikom testiranja ovog slučaja korišćenja.

Redni broj	Test koraci	Očekivani rezultati
1.	Vozac prolazi kroz tunel	Vozac se nalazi u tunelu
2.	Senzor očitava izlazak vozila iz tunela	Vozac je očitan na senzoru izlaska
3.	Sistem beleži izlazak vozača	Sistem ima informaciju da je vozač napustio tunel
4.	Semafor na ulazu pokazuje zeleno svetlo	Svetlo na semaforu je zeleno

Slika 6.1.4 UseCase testiranje prolaska vozila kroz tunel [Izvor: NM SE321 - 2020/2021.]

Nakon standardnog načina korišćenja sistema gde korisnik prolazi kroz tunel i prikazuje se na senzoru izlaska u ovom slučaju korisnik iz nekog razloga nije izašao iz tunela i tada nastaje problem definisan u ovom test slučaju. Sistem je za predviđeno vreme na osnovu brzine i dužine vozila trebao na senzoru izlaska da očita vozilo ali se ono nije pojavilo. U tom slučaju

sistem šalje znak da je došlo do zastoja u saobraćaju i svetlo na ulasku u tunel biva crveno za ostale vozače koji dolaze do ulaza u tunel.

IZRAČUNAVANJE TEST SLUČAJEVA PO JEDNOM SLUČAJU KORIŠĆENJA

Postoji formula za izračunavanje test slučajeva po jednom slučaju korišćenja

Formula za izračunavanje test slučajeva na osnovu kojih se izračunava njihov broj po jednom slučaju korišćenja je:

$$M^*N^*X$$

gde su vrednosti sledeće:

M = broj test sekvenci koje su generisane $n(n-1)/2$ gde je n broj use case sekvenci koje su definisane.

N = (p+q) gde su p i q dužine svakog kombinovanog slučaja korišćenja

X = broj različitih varijanti use case sekvenci.

Pokrivenost slučaja korišćenja na osnovu prethodne tabele gde se vidi koliko je od strane sistema pokriveno različitih navedenih faktora.

Slučaj korišćenja	Broj poruka	Putanje	Račvanja	Iskazi	Iteracije
Proveravanje svetla na semaforu	1	2	1	1	2
Očitavanje zelenog svetla	2	1	0	1	1
Očitavanje crvenog svetla	1	2	1	1	1
Zaustavljanje na definisanoj udaljenosti	1	1	1	1	2
Ulazak u tunel	1	1	0	2	2
Očitavanje na senzoru ulaska	2	1	0	1	1
Izlazak iz tunela	2	1	0	1	1
Prolazak kroz tunel	1	1	0	1	1
Očitavanje na senzoru izlaska	2	1	0	1	2
Ukupno	13	11	3	10	13

Slika 6.1.5 Opis slučajeva korišćenja [Izvor: NM SE321 - 2020/2021.]

POUZDANOST SOFTVERA NA OSNOVU DIAGRAMA SLUČAJEVA KORIŠĆENJA

Navedeni slučajevi korišćenja pokazuju da je prvi kreirani UseCase dijagram dobar, i da nema grešaka u okviru njegove putanje, već samo može postajati alternativni scenario.

Pouzdanost softvera na osnovu UseCase dijagrama prikazana je u procentima (slika 6.6).

Navedeni slučajevi korišćenja pokazuju da je prvi kreirani UseCase dijagram dobar, i da nema grešaka u okviru njegove putanje, već samo može postajati alternativni scenario.

Jedina mana koja se pojavila je da nedostaje određeni broj poruka koji prikazuju korisniku da je uspešno prošao kroz deo sistema i odradio neku funkcionalnost. U tom slučaju, potrebno je uraditi modifikacije za navedene slučajeve korišćenja i omogućiti vidljivost nakon završenog slučaja korišćenja.

Slučaj korišćenja	Pokrivenost poruka	Pokrivenost putanja	Pokrivenost račvanja	Pokrivenost iskaza	Pokrivenost iteracije
Proveravanje svetla na semaforu	1/1	2/2	1/1	1/1	2/2
Očitavanje zelenog svetla	1/2	1/1	0/0	1/1	1/1
Očitavanje crvenog svetla	1/1	2/2	1/1	1/1	1/1
Zaustavljanje na definisanoj udaljenosti	1/1	1/1	1/1	1/1	2/2
Ulazak u tunel	0/1	1/1	0/0	2/2	2/2
Očitavanje na senzoru ulaska	2/2	1/1	0/0	1/1	1/1
Izlazak iz tunela	2/2	1/1	0/0	1/1	1/1
Prolazak kroz tunel	1/1	1/1	0/0	1/1	1/1
Očitavanje na senzoru izlaska	2/2	1/1	0/0	1/1	2/2
Ukupno	95%	100%	100%	100%	100%

Slika 6.1.6 Pokrivenost slučajeva korišćenja [Izvor: NM SE321 - 2020/2021.]

DORADE I MODIFIKACIJE SISTEMA NA OSNOVU TESTIRANJA MODELA SLUČAJEVA KORIŠĆENJA

Vreme potrebno za izmene koje su vezane za uočene sistemske nedostatke mogu varirati u odnosu na to ko je zadužen za njih i koliko vremena ima do isporuke softvera.

Ispravke će biti vezane za :

- Očitavanje zelenog svetla
- Ulazak u tunel

u okviru kojih će biti dodate nove funkcije i omogućeno obaveštenje ka korisnicima u vidu poruka da su uspešno prošli traženi proces od sistema.

Predstavljeni resursi sistema (opisani u poglavlju "Predviđeni resursi" prikazuju vremenski period i softverske zahteve potrebne za uočene izmene i nedostatke u sistemu. Vreme je postavljeno u odnosu na iskustvo softverskog i saobraćajnog inženjera (oba inženjera učestvuju u ovom delu). Kada se definišu radni sati inženjera potrebno je tačno odrediti jer sve preko navedenih sati se dodatno plaća i sistem prelazi predviđene troškove razvoja kao i predviđene troškove dorade koji su planirani u modelovanju i specificiranju sistema.

Primećen nedostatak	Vreme potrebno za doradu aplikacije	Zadužena osoba	Hardverski resursi
Očitavanje zelenog svetla	20 radnih sati	Softverski inženjer 1 Saobraćajni inženjer	PC1, PC2
Ulazak u tunel	10 radnih sati	Softverski inženjer 2	PC3

Slika 6.1.7 Procena procesa dorade [Izvor: NM SE321 - 2020/2021.]

✓ 6.1 Vežba za samostalni rad

ZADACI ZA SAMOSTALNI RAD

Na osnovu obrađenih aktivnosti testiranja softvera uraditi testiranje mejl sistema.

Razmotrite scenario u kojem korisnik on line kupuje proizvod sa veb site-a za kupovinu.

Korisnik će se prvo logovati na sistem i započeti pretraživanje. Nakon toga on će odabrat jednu ili više stavki prikazanih u rezultatima pretraživanja i dodaće ih u korpu. Nakon svega ovoga, on će završiti kupovinu (izvršiti check out). Za ovoj primer logički povezane serije koraka koje će korisnik obaviti u sistemu da bi izvršio zadatku primenom testiranja na bazi slučajeva korišćenja:

1. Treba testirati tok transakcija u cijelom ovom sistemu od početka do kraja tako što ćete:
 - napraviti dijagram slučajeva korišćenja slično kako je to prikazano u pokaznoj vežbi
 - Definisati dijagram aktivnosti
 - Testirati moguće putanje iz dijagrama aktivnosti
2. Na osnovu rezultata testiranja izvesti zaključak o pouzdanosti sistema, eventualnim nedostacima i mogućim unapređenjima.

Vreme potrebno za izradu 1. zadatka 75 minuta (25 minuta po svakoj stavci u zadatku)

Vreme potrebno za izradu 2. zadatka 15 minuta.

✓ Poglavlje 7

Domaći zadatak 01

PRVI DOMAĆI ZADATAK - VREME IZRADE 150 MIN.

Nakon prve lekcije potrebno je uraditi prvi domaći zadatak

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

Treba testirati tok transakcija u ovom delu sistemu od početka do kraja tako što će te za izabrani deo ISUM-a:

1. Napraviti jedan dijagram slučajeva korišćenja (po izboru) slično kako je to prikazano u pokaznoj vežbi
2. Za njega definisati dijagram aktivnosti
3. Testirati moguće putanje iz dijagrama aktivnosti
4. Na osnovu rezultata testiranja izvesti zaključak o pouzdanosti sistema, eventualnim nedostacima i mogućim unapređenjima.

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ01-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br.Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

nikola.petrovic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ01.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Kvalitet, testiranje i održavanje softvera

KVALITET I TESTIRANJE SOFTVERA

Šta smo naučili u ovoj lekciji?

Kvalitet softvera je definisan kao sposobnost da se proizvede softver koji zadovoljava ili nadmašuje postavljene zahteve (prema definisanim merljivim kriterijumima) i koji je proizведен definisanim procesom. Objašnjeno je da kvalitet softvera jeste višedimenzionalni koncept koji se ne može jednostavno definisati. Moderni softver koji se razvija, ima stalnu tendenciju rasta kompleksnosti. Sa porastom kompleksnosti realizovanih funkcija i primena, posebno je narastao zahtev za kvalitetom softvera u pogledu pouzdanosti, pogodnosti za testiranje i održavanje, ponovne upotrebljivosti, otpornosti na greške i drugih faktora kvaliteta softvera.

U obezbeđenju kvaliteta softvera važno je sprovoditi: Inspekcije, recenzije i preglede, jer su specifične tehnike fokusirane na ocenjivanje artefakata (pogodno za dokumentaciju, programski kod) i efikasni su metodi poboljšanja kvaliteta i produktivnosti razvojnog procesa. Sprovode se na sastancima gde jedan od prisutnih igra ulogu voditelja, a drugi zapisničara (beleži zahteve za izmenama, probleme, pitanja itd).

Najvažniji način za poboljšanje kvaliteta softvera je testiranje. Istaknut je značaj poboljšanja procesa testiranja koji treba da se sprovodi kao konstantan proces zajedno sa svim drugim elementima razvoja softvera. Angažuju se sve veći resursi kako velikih tako i malih kompanija u proces testiranja i njegovo unapređenje. Mogu se testirati različiti aspekti softvera kao što su: testiranje sigurnosti (engl.**security testing**), testiranje količine podataka (engl.**volume testing**), testiranje upotrebljivosti (engl.**usability testing**), testiranje integriteta (engl.**integrity testing**) itd. i primeniti različiti tipovi testiranja o čemu je u lekciji bilo reči.

LITERATURA ZA LEKCIJU 01

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. I. Sommerville, Software Engineering, 9th ed., Addison-Wesley, 2011. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
3. P. Grubb and A.A. Takang, Software Maintenance: Concepts and Practice, 2nd ed., World Scientific Publishing, 2003. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)

4. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)

Dopunska literatura:

1. Laird L., Brennan C., Software measurement and estimation: a practical approach, John Wiley & Sons, Inc., New Jersey. 2006 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
3. Glenford J. Myers, The Art of Software Testing, second edition., John Wiley & Sons, Inc. 2004 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>



SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Faktori kvaliteta softvera –
procesi, modeli, zahtevi i metrika

Lekcija 02

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Lekcija 02

FAKTORI KVALITETA SOFTVERA – PROCESI, MODELI, ZAHTEVI I METRIKA

- ✓ Faktori kvaliteta softvera – procesi, modeli, zahtevi i metrika
- ✓ Poglavlje 1: Metrike kvaliteta softvera
- ✓ Poglavlje 2: Sistem upravljanja kvalitetom (QMS)
- ✓ Poglavlje 3: Zahtevi o kvalitetu softvera
- ✓ Poglavlje 4: Merenja i ocenjivanje softvera
- ✓ Poglavlje 5: Metode i tehnike merenja i ocenjivanje softvera
- ✓ Poglavlje 6: Grupna vežba - Testiranje softvera ATM uređaja
- ✓ Poglavlje 7: Vežba za samostalni rad
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Kvalitet softvera – zahtevi, modeli, metrika, obezbeđenje, merenje i ocenjivanje

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Obezbeđivanje kvaliteta, obezbeđuje da organizacija prati ciljeve kvaliteta, što uključuje definisanje i izbor različitih standarda koji se uključuju u proces razvoja softvera.

Usaglašavanje o zahtevima kvaliteta, kao i jasna komunikacija sa softverskim inženjerom o tome šta predstavlja kvalitet, zahteva da mnogi aspekti kvaliteta budu formalno definisani i razmatrani. Softverski inženjer bi trebalo da razume osnovna značenja koncepta kvaliteta i karakteristika kao i njihovih vrednosti za softver prilikom razvoja ili do održavanja.

Institut za softverski inženjering kreirao je model za procenu zrelosti softverskih procesa organizacije i identifikaciju ključnih praksi koje su neophodne za povećanje zrelosti procesa. Organizacije unapređuju karakteristike sopstvenih softverskih procesa i prolaze kroz različite faze zrelosti. U ovoj lekciji dat je pregled nekih od vodećih standarda kvaliteta softvera.

Za merenje kvaliteta softvera prema modelu kvaliteta koriste se interne i eksterne metrike. Interne metrike mogu se primeniti na softverske proizvode koji se ne izvršavaju (kao što su specifikacije ili izvorni kod) za vreme projektovanja i kodiranja. Eksterne metrike koriste merenja softverskog proizvoda koje su izvedene iz merenja ponašanja sistema čiji je on deo, testiranjem, izvršavanjem i posmatranjem softvera koji se izvršava ili sistema.

Model ocenjivanja kvaliteta softvera, je vrlo važna komponenta sistema upravljanje kvalitetom (QMS) u kompaniji. QMS je definisan u ISO 9001 kao "organizaciona struktura, odgovornost, procedure, procesi i resursi za uvođenje sistema upravljanje kvalitetom u kompaniji radi postizanja postavljenih ciljeva u vezi kvaliteta".

Obezbeđivanje kvaliteta, obezbeđuje da organizacija prati ciljeve kvaliteta, što uključuje definisanje i izbor različitih standarda koji se uključuju u proces razvoja softvera.

▼ Poglavlje 1

Metrike kvaliteta softvera

INTERNE I EKSTERNE METRIKE SOFTVERA

Interne metrike se mogu primeniti na softverske proizvode koji se ne izvršavaju (specifikacije ili izvorni kod) dok su eksterne metrike izvedene iz merenja ponašanja softvera

Za merenje kvaliteta softvera prema modelu kvaliteta koriste se interne i eksterne metrike.

Interne metrike softvera se mogu primeniti na softverske proizvode koji se ne izvršavaju (kao što su specifikacije ili izvorni kod) za vreme projektovanja i kodiranja. U toku projektovanja softverskog proizvoda među proizvodi mogu se vrednuju korišćenjem internih metrika koje mere unutrašnja svojstva. Osnovna namena internih metrika je postizanje zahtevanog eksternog kvaliteta i kvaliteta u upotrebi. Interne metrike obezbeđuju pomoć korisnicima, osobama koje vrše vrednovanje, osobama koje vrše testiranje i projektantima, omogućavaju vrednovanje kvaliteta softverskog proizvoda i ukazuju na elemente kvaliteta mnogo ranije nego što softverski proizvod postane izvršiv.

Interne metrike i kriterijumi prihvatljivosti se definišu u cilju određivanja atributa internog kvaliteta koji se mogu koristiti za verifikaciju da međuproizvod zadovoljava specifikacije internog kvaliteta tokom razvoja. Preporučuje se da interne metrike koje se koriste, imaju jaku vezu sa ciljnim eksternim metrikama, tako da se mogu koristiti za predviđanje vrednosti eksternih metrika. Međutim, prilično je teško razviti teorijski model koji obezbeduje jaku vezu između internih i eksternih metrika, o čemu će kasnije biti više reči.

Eksterne metrike softvera koriste merenja softverskog proizvoda koje su izvedene iz merenja ponašanja sistema čiji je on deo, testiranjem, izvršavanjem i posmatranjem softvera ili sistema koji se izvršava. Pre naručivanja ili korišćenja softverskog proizvoda, treba ga vrednovati korišćenjem metrika koje su zasnovane na poslovnim objektima povezanim sa upotrebom, eksploatacijom i upravljanjem proizvodom u specificiranom organizacionom i tehničkom okruženju. Eksterne metrike obezbeđuju pomoć korisnicima, osobama koje vrše vrednovanje, osobama koje vrše testiranje i projektantima, i omogućavaju im vrednovanje kvaliteta softverskog proizvoda u toku testiranja ili izvršavanja.

Kada se primenjuje interna i eksterna metrika ?

Nakon definisanja zahteva, definišu se i karakteristike i podkarakteristike koje se odnose na zahteve kvaliteta. U ovom trenutku se definišu i odgovarajuće eksterne metrike i kriterijumi prihvatljivosti kojima se potvrđuje da softver zadovoljava potrebe korisnika. Nakon toga se definišu interni atributi kvaliteta softvera i specificira plan radi konačnog postizanja zahtevanog eksternog kvaliteta i kvaliteta u upotrebi i ugrađuju u proizvod tokom razvoja.

METRIKE KVALITETA U UPOTREBI

Vrednovanje kvaliteta u upotrebi vrši validaciju kvaliteta softverskog proizvoda u scenarijima korisnik – zadatak.

Metrike kvaliteta u upotrebi se odnose na stepen do kog proizvod ispunjava potrebe korisnika radi postizanja specificiranih ciljeva sa efektivnošću, produktivnošću, bezbednošću i zadovoljstvom u kontekstu upotrebe. Vrednovanje kvaliteta u upotrebi vrši validaciju kvaliteta softverskog proizvoda u scenarijima korisnik – zadatak. Kvalitet u upotrebi je kvalitet sa stanovišta korisnika.

Veza između kvaliteta u upotrebi i drugih karakteristika kvaliteta softverskog proizvoda zavisi od vrste korisnika:

1. *krajnji korisnik, za kog je kvalitet u upotrebi uglavnom rezultat funkcionalnosti, pouzdanosti, upotrebljivosti i efikasnosti;*
2. *osoba koja je odgovorna za implementaciju softvera, za koju je kvalitet u upotrebi uglavnom rezultat pogodnosti za održavanje;*
3. *osoba koja prenosi softver, za koju je kvalitet u upotrebi uglavnom rezultat prenosivosti.*

Merenje se obično zahteva na sva tri nivoa, jer ispunjavanje kriterijuma za interni kvalitet je obično nedovoljno za obezbeđenje ispunjenja kriterijuma za eksterni kvalitet, a ispunjavanje kriterijuma za eksterno merenje podkarakteristika je obično nedovoljno za obezbeđenje ispunjenja kriterijuma za kvalitet u upotrebi.

Jedan od standarda koji se bavi ovom problematikom je [ISO/IEC 14598](#) (Vrednovanje softverskog proizvoda). ISO/IEC 14598-1 sastoji se od nekoliko delova koji se pojavljuju pod opštim nazivom Informaciona tehnologija – Vrednovanje softverskih proizvoda.

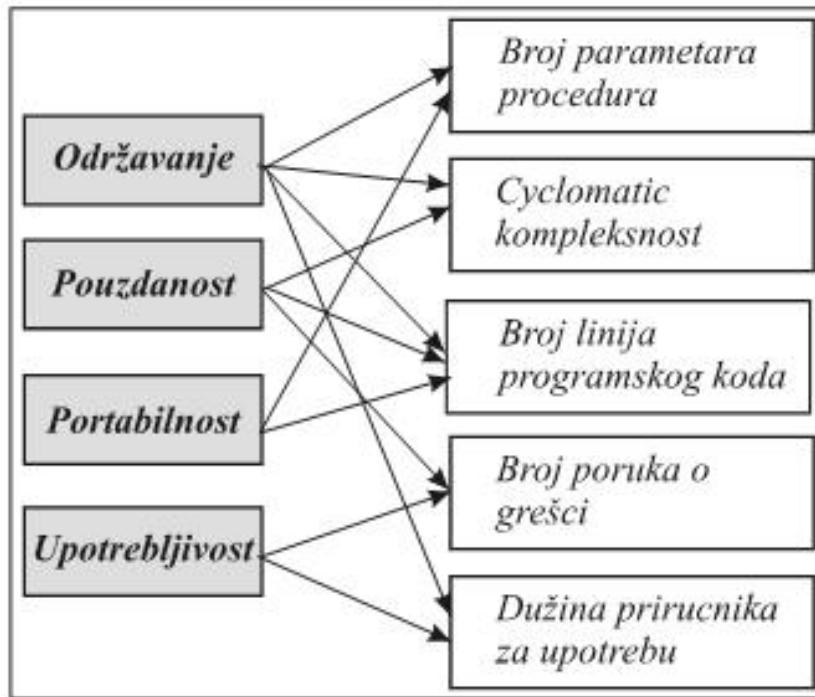
Vrednovanje softverskog kvaliteta odnosi se na model kvaliteta, metod vrednovanja, merenje softvera, i alate za podršku. Ovaj standard namenjen je projektantima, onima koji vrše nabavku i nezavisnim ocenjivačima. Primenjene vrednosti standarda koriste rukovodiocima i projektantima, analitičarima koji se bave vezom između internih i eksternih metrika.

METRIKA KONTROLE I METRIKA PREDVIĐANJA

Metrika kontrole se koristi u cilju kontrole procesa vezanih za softver, dok metrika predviđanja meri atribute proizvoda koji mogu biti korišćeni da predvide pripadajući kvalitet softvera.

Metrika se može podeliti na metriku kontrole i metriku predviđanja.

Metrika kontrole se koristi odstrane menadžmenta u cilju kontrole procesa vezanih za softver. Primeri ove metrike su potreban rad i korišćenje prostora na disku. Metrika kontrole nam može dati informacije o kvalitetu procesa i kvalitetu samog proizvoda.



Slika 1.1 Veza između internih i eksternih karakteristika [Izvor: NM SE321-2020/2021.]

Metrika predviđanja meri atribute proizvoda koji mogu biti korišćeni da predvide pripadajući kvalitet softvera.

Na primer, čitljivost prateće softverske dokumentacije se može proceniti na osnovu tog indeksa, ili jednostavnost održavanja softvera se može predvideti preko cyclomatic kompleksnosti (McNabe, 1976). Znači, merenjem jednog internog atributa, kao što je cyclomatic kompleksnost dobijamo vrednost jednog eksternog parametra, kao što je održavanje (Slika 1.1).

Pri merenju internih karakteristike, mora se voditi računa o tome da:

1. Interni atributi moraju biti tačno izmereni.
2. Mora postojati zavisnost između onoga što merimo i eksternih ponašanja atributa.
3. Relacija se mora razumeti, validirati i iskazati preko formule ili modela.

Mnogi autori, (Sommervile, Kitchenham) ističu da je poslednji uslov najbitniji, kao i to da se on često zanemaruje. Još jedan problem koji se javlja pri ovom merenju je suštinsko razumevanje šta izmereni podaci u suštini znače.

OSTALE VRSTE METRIKE SOFTVERA

Metrika kvaliteta dizajna meri: koheziju, povezanost, razumljivost itd.;

Metrika kvaliteta programa: dužinu koda, cyclomatic kompleksnost;

Metrika kvaliteta dokumentacije: Fog indeks

Pri definisanju metrike softvera obično se srećemo i sa:

1. **Metrikom kvaliteta dizajna** koja obuhvata: **koheziju, povezanost, razumljivost, adaptibilnost i kompleksnost.** Kompleksnost se može meriti kao Kompleksnost = Dužina (Fan - in x Fan - out)², gde je Fan-in broj linija koje ulaze u komponentu, a Fan-out , broj linija koje izlaze iz komponente, a dužina je predstavljena kao broj linja koda.
2. **Metrika kvaliteta programa** koja obuhvata: **dužinu koda, cyclomatic kompleksnost, dužinu identifikatora, dubinu uslovnog ugnježdavanja.**
3. **Metrika kvaliteta dokumentacije** koja obuhvata najpoznatiji Fog indeks, koji predstavlja odnos dužine teksta i kompleksnih izraza i ukazuje na „čitljivost“ teksta.

Svako ko se bavi izradom softvera koristi neku vrstu metrike za taj softver. Za izradu web sajtova, a posebno web aplikacija, potrebna je takođe odgovarajuća metrika. S obzirom da se Web sve više koristi kao izvor informacija, pri izradi sajtova neophodno je unapred, na početku realizacije projekta, imati definisan plan i troškove izrade (specifikaciju) kako bi se troškovi i potrebno vreme za izradu sajta sveli na minimum.

▼ Poglavlje 2

Sistem upravljanja kvalitetom (QMS)

OSNOVNE DIMENZIJE KVALITETA SOFTVERA

Povezanost procesa kvaliteta i kvaliteta u upotrebi je jasna. Proces kvaliteta doprinosi kvalitetu proizvoda, a kvalitet proizvoda upotrebnom kvalitetu.

Iako u softverskoj industrijskoj praksi ne postoji usvojen standard kvaliteta softvera koji određuje šta to čini dobar kvalitet softvera, generalno, kvalitetan softver je onaj koji zadovoljava potrebe kupca, doprinosi profitu, ne proizvodi ozbiljne probleme u poslovanju i po zdravlje ljudi. Kvalitet softvera je teško definisati jer postoje različiti aspekti i termini koji su definisani i opisani u nizu standarda kao što su pomenuti: ISO 9000-3, ISO/IEC 9126 (Kvalitet softverskog proizvoda) i ISO/IEC 14598 (Vrednovanje softverskog proizvoda) i drugi. Osnovne dimenzije kvaliteta softvera su sigurno:

- 1. **Stepen zadovoljenja:** to je kvantitativno izraženi nivo zadovoljenja potreba kupca i njegovih očekivanja od softverskog proizvoda.
- 2. **Vrednost proizvoda:** to je kvantitativno izraženi nivo vrednosti softverskog proizvoda za pojedine učesnike (vlasnike projekta) u odnosu na uslove konkurencije.
- 3. **Ključni atributi (Q - "ilities"):** to je kvantitativno izraženi nivo kombinacije više karakteristika koje softver poseduje (npr. pouzdanost, upotrebljivost, pogodnost za održavanje).
- 4. **Defektnost:** to je kvantitativno izraženi nivo neispravnog funkcionisanja softvera u korisnikovom okruženju usled grešaka u isporučenom softveru.
- 5. **Kvalitet procesa:** kvalitet softvera se ugrađuje u procesu razvoja softvera koji treba da obezbedi postavljene ciljeve u pogledu kvaliteta (dobri projektanti koji na pravi i efikasan način izrađuju softver).

Zahtevi koji se odnose na kvalitet softverskog proizvoda uključuju kriterijume ocenjivanja za interni kvalitet, eksterni kvalitet i kvalitet u upotrebi radi zadovoljenja potreba projektanata, onih koji održavaju proizvod, naručilaca i korisnika. (ISO/IEC 14598-1:1999, tačka 8.).

Povezanost procesa kvaliteta i kvaliteta u upotrebi je jasna. Proces kvaliteta doprinosi kvalitetu proizvoda, a kvalitet proizvoda upotrebnom kvalitetu. Moguća je i povratna sprega od proizvoda u upotrebi, ka kvalitetu proizvoda tj. procesu kvaliteta. Interni atributi utiču na eksterne attribute, a eksterni vrše uticaj na kvalitet u upotrebi. Može se zaključiti sledeće:

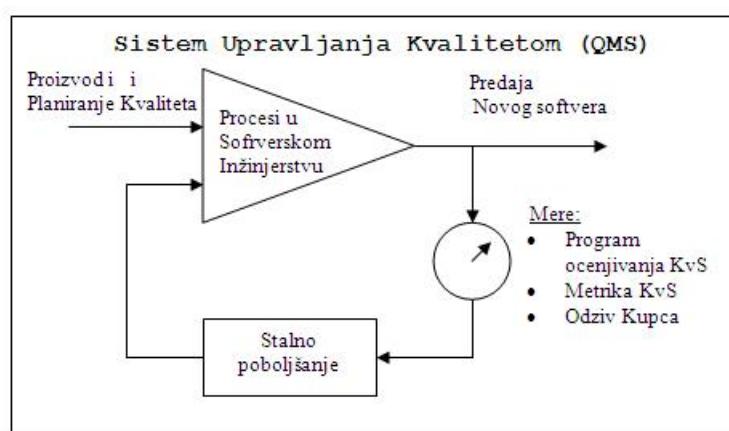
Izbor adekvatnog sistema obezbeđenja, kontrole i ocenjivanja kvaliteta softverskog proizvoda, je ključ (ne)uspeha. Standardizovani model ocenjivanja kvaliteta softvera (SSQA)

koji je razvila Motorola proširen je uključivanjem metodologije mini-periodične ocene kvaliteta softvera (KvS) koje su vrlo važne u kontinuiranom poboljšanju kvaliteta od strane različitih proizvodnih timova.

SISTEM UPRAVLJANJA KVALITETOM

QMS je definisan u ISO 9001 kao "organizacijska struktura, odgovornost, procedure, procesi i resursi za uvođenje sistema upravljanje kvalitetom radi postizanje postavljenih ciljeva."

Sistem upravljanja kvalitetom (engl. Quality Management System - QMS) je definisan standardom ISO 9001 kao "organizacijska struktura, odgovornost, procedure, procesi i resursi za uvođenje sistema upravljanje kvalitetom u kompaniji radi postizanje postavljenih ciljeva u vezi kvaliteta". U uspešnoj kompaniji, upravljanje kvalitetom dalo je naglasak ne samo na smanjenje stvari koje su pogrešno urađene, već je naglasak na povećanju stvari koje su učinjene na pravi način u zadovoljenju kupca.



Slika 2.1 Sistem upravljanja kvalitetom (QMS) [Izvor: NM SE321-2020/2021.]

Ovaj novi naglasak na upravljanje kvalitetom, stvorio je okruženje koje poboljšava produktivnost u procesima projektovanja, kao i realizovanim proizvodima i uslugama. Jednako je važan naglasak na poboljšanu komunikaciju, timski rad i zadovoljstvo zaposlenih. Dobijene povratne informacije od klijenata jesu da su oni zahvalni na našem naporu i da je usvajanjem QMS poboljšana komunikacija, kvalitet i ukupno zadovoljstvo kupaca. Sistem upravljanja kvalitetom (QMS) u uspešnoj kompaniji je prikazan na Slici 2.1.

Tri ključna elementa sistema upravljanja kvalitetom su:

- *Zajednički procesi*,
- *Mere i*
- *Neprekidno poboljšanje*

Za svaki razvijani softverski proizvod, uključujući nadogradnju prethodno izrađenog softverskog proizvoda, sva tri elementa sistema upravljanja kvalitetom su neophodni za postizanje i premašivanje zadovoljstva kupaca. Ključni element sistema upravljanja

kvalitetom je SSQA metodologija ocenjivanja kvaliteta koja omogućuje disciplinovan proces poboljšanja kvaliteta na proizvodnoj osnovi od softverskog proizvoda-do-proizvoda.

MENADŽMENT KVALITETA SOFTVERA

Primjenjuje se na sve perspektive softverskih procesa; definiše procese, vlasnike procesa i zahteve za procesima, merenja procesa i njihovih izlaza, kao njihove povratne informacije

Menadžment kvaliteta softvera (SQM) primjenjuje se na sve perspektive softverskih procesa, proizvode i resurse. On definiše procese, vlasnike procesa i zahteve za ovim procesima, merenja procesa i njihovih izlaza, kao i kanale povratnih informacija (eng.**feedback**). Proces menadžmenta kvaliteta softvera sastoji se od dosta aktivnosti. Neke mogu direktno da otkriju defekte, dok druge ukazuju gde može biti korisno raditi dalja ispitivanja. Druge se takođe, nazivaju aktivnosti direktnog otkrivanja defekata (eng.**direct-defect-finding**). Mnoge aktivnosti se koriste u oba slučaja.

Procesi menadžmenta kvaliteta softvera se moraju odnositi na to koliko dobro će softverski proizvodi zadovoljiti zahteve kupaca i stejkholdera, obezbediti vrednost za kupce i druge stejkholdere, i obezbediti kvalitet softvera koji je potreban za ispunjenje softverskih zahteva. SQM se može koristiti za procenu međuproizvoda kao i finalnog proizvoda. Neki od specifičnih SQM procesa su definisani u standardu (IEEE12207.0-96):

- Proces osiguranja kvaliteta (eng.**Quality assurance process**)
- Proces verifikacije (eng.**Verification process**)
- Proces validacije (eng.**Validation process**)
- Proces pregleda (eng.**Review process**)
- Proces revizije (eng.**Audit process**)

Ovi procesi podstiču kvalitet i, takođe, nalaze moguće probleme. Ali se malo razlikuju u svojom značaju. SQM procesi pomažu da se obezbedi kvalitet softvera u datom projektu. Oni, takođe obezbeđuju, kao sporedni proizvod, opšte informacije za menadžment, uključujući indikaciju kvaliteta čitavog procesa softverskog inženjerstva.

Rezultati ovih zadataka su sastavljeni u okviru izveštaja za menadžment pre sprovođenja korektivnih akcija. Menadžment SQM procesa ima zadatak da obezbedi da su rezultati ovih izveštaja tačni. Kao što je opisano u ovoj oblasti znanja, SQM procesi su blisko povezani; oni mogu da se preklapaju i ponekad su čak kombinovani.

Upravljanje rizikom (eng.**Risk Management**), takođe, može imati značajnu ulogu u isporuci kvalitetnog softvera. Uključivanje disciplinovane analize rizika i menadžment tehnika u procese životnog ciklusa softvera, može povećati potencijal za proizvodnju kvalitetnog proizvoda.

OSIGURANJE KVALITETA SOFTVERA

Pruža obezbeđenje da softverski proizvodi odgovaraju njihovim datim zahtevima planiranja, propisima itd. kako bi se obezbedilo poverenje da je kvalitet ugrađen u softver.

Procesi **osiguranja kvaliteta softvera** - **SQA** pružaju obezbeđenje da softverski proizvodi i procesi u životnom ciklusu projekta odgovaraju njihovim datim zahtevima planiranja, propisima i skupu aktivnosti kako bi se obezbedilo adekvatno poverenje da je kvalitet ugrađen u softver.

Ovo znači za osiguranje da je problem jasno i adekvatno naveden i da su zahtevi rešenja ispravno definisani i izraženi. SQA teži da održi kvalitet kroz razvoj i održavanje proizvoda putem izvršavanja više aktivnosti u svakoj fazi, što može rezultirati ranom identifikacijom problema, gotovo neizbežne pojave u bilo kojoj kompleksnoj aktivnosti. Uloga SQA u odnosu na proces je osiguranje da su planirani procesi prikladni i kasnije implementirani prema planu, i da je relevantan proces merenja obezbeđen za prikladnu organizaciju.

SQA plan definiše sredstva koja će biti korišćena kako bi se obezbedilo da softver, razvijen za određeni proizvod, zadovoljava korisnikove zahteve i da je najvećeg mogućeg kvaliteta unutar ograničenja projekta. Kako bi se to uradilo mora se prvo obezbediti da je cilj kvaliteta jasno definisan i shvaćen. Mora se uzeti u obzir menadžment, razvoj i planovi održavanja za softver. Za više detalja pogledajte standard (IEEE730-98).

Određene aktivnosti i zadaci kvaliteta su izloženi, zajedno sa njihovim troškovima i zahtevima za resurse, njihovim ukupnim ciljevima menadžmenta, i njihovim rasporedom u vezi sa onim ciljevima u menadžmentu softverskog inženjerstva, razvoja, ili planova održavanja.

SQA plan mora biti konzistentan sa planom menadžmenta softverske konfiguracije (detaljnije u Software Configuration Management KA). SQA plan identificuje dokumenta, standarde, prakse i konvencije kojima se reguliše projekat, i kako će se oni proveravati i pratiti radi obezbeđenja adekvatnosti i usaglašenosti. Plan SQA takođe, identificuje mere, statističke tehnike, procedure za izveštavanje o problemima kao i korektivne akcije, resurse kao što su, alati, tehnike, metodologije, obezbeđenje fizičkih medija, trening, i SQA izveštavanje i dokumentacija. Štaviše, SQA plan se odnosi i na aktivnosti obezbeđenja softvera bilo kog drugog tipa aktivnosti opisanog u planu softvera, kao što je nabavka od dobavljača softvera, do projekta ili instalacije komercijalnog off-the-shelf softvera (COTS), i servisiranje nakon isporuke softvera. Takođe, može sadržati kriterijume prihvatanja kao i aktivnosti izveštavanja i upravljanja koje su kritične za kvalitet softvera.

▼ Poglavlje 3

Zahtevi o kvalitetu softvera

ZAHTEVI KVALITETA SOFTVERA

Različiti faktori utiču na planiranje, menadžment i selekciju SQM aktivnosti i tehnika.

Na planiranje, menadžment i selekciju SQM aktivnosti i tehnika, utiču različiti faktori uključujući:

- Domene sistema u kojem će se nalaziti softver (sigurnosno-kritični, kritični u odnosu na misiju, poslovno-kritični)
- Sistemske i softverske zahteve
- Komercijalne (eksterne) ili standardne (interne) komponente koje će se koristiti u sistemu
- Primenu specifičnih standarda softverskog inženjerstva
- Metode i softverske alate koji će se koristiti za razvoj i održavanje, kao i za ocenjivanje kvaliteta i poboljšanja
- Budžet, osoblje, organizaciju projekta, planove i raspored svih procesa
- Namenjene zkorisnike i upotrebu sistema
- Nivo integrisanosti sistema

Informacije o uticajima ovih faktora utiču na to kako su SQM procesi organizovani i dokumentovani, kako se biraju specifične SQM aktivnosti, koji su resursi potrebni i koji će pomerati granice napora.

U slučajevima kada sistemski otkazi mogu imati izuzetno ozbiljne posledice, ukupna pouzdanost (hardver, softver i ljudi) je osnovni zahtev kvaliteta iznad osnovne funkcionalnosti. Softverska zavisnost uključuje i takve karakteristike kao što su tolerancija na greške, sigurnost, bezbednost, i upotrebljivost. Pouzdanost je takođe i kriterijum koji se može definisati terminima zavisnosti (ISO9126).

Nivo integriteta softvera se određuje na osnovu mogućih posledica otkaza softvera i verovatnoće otkaza. Za softver u kojem je važna sigurnost ili bezbednost, tehnike, kao što je analiza hazarda za bezbednost ili analiza pretnji za sigurnost, mogu se koristiti za razvoj plana aktivnosti, koji će identifikovati gde se nalazi tačka potencijalnih problema. Istorija neuspeha sličnog softvera.

Obezbeđivanje kvaliteta, obezbeđuje da organizacija prati ciljeve kvaliteta, što uključuje definisanje i izbor različitih standarda koji se uključuju u proces razvoja softvera ili sam softverski proizvod. Izabrani standardi mogu biti ISO 9000 ili JUS/ISO 12207 - Informaciona tehnologija - procesi životnog ciklusa softvera; i/ili neki nacionalni standardi, kao na primer BS 5750.

FAKTORI KOJI UTIČU NA PLANIRANJE AKTIVNOSTI SQA

Pošto softver može značajno uticati na rad i fukcionisanje poslovnog ili nekog drugog sistema, potrebno je da softver zadovoljava mnogobrojne atribute kvaliteta i da bude kvalitetan u celini.

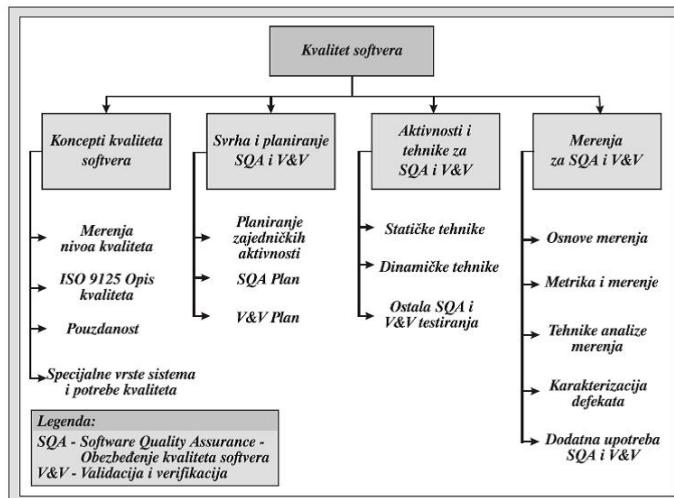
Ukoliko se posmatra u užem smislu, softver je "alat" koji se koristi unutar nekog sistema, ali sa druge strane, softver značajno utiče na funkcionisanje tog sistema. Pošto softver može značajno uticati na rad i fukcionisanje poslovnog ili nekog drugog sistema, potrebno je da softver zadovoljava mnogobrojne atribute kvaliteta i da bude kvalitetan u celini.

Softver mora da zadovolji, na prvom mestu, zahteve korisnika, i da poseduje određen nivo kvaliteta, a ne samo funkcionalnost.

Na primer kvalitet podataka se može specificirati kroz: tačnost (mera odstupanja neke vrednosti v od neke druge vrednosti v' koja se smatra tačnom); kompletност (stepen do koga je specifična vrednost uključena u kolekciju podataka), konzistentnost (koja se definiše na nivou prezentacije, vrednosti i izgleda), interpretaciju (format u kome je podatak specificiran i jasno definisan), pouzdanost (koja se definiše kroz pouzdanost podatka i pouzdanosti izvora), pravovremenost (mera koliko je podatak pravovremen i odgovarajući za neku operaciju).

Kao što je potrebno meriti i vrednovati ove karakteristike da bi se definisao kvalitet podataka, potrebno je meriti i mnoge druge karakteristike koje nam omogućavaju utvrđivanje kvaliteta softvera, pri čemu metrika softvera ima značajnu ulogu.

Zato se moraju definisati karakteristike, koje često nisu tako eksplisitne, ali značajno utiču na performanse i kvalitet softvera. Postoje brojni, različiti pristupi u definisanju seta karakteristika koje je potrebno meriti da bi se utvrdio kvalitet softvera, odnosno definisao modela kvaliteta, počev od klasika Boehm, McCall, pa do novijih pristupa ISO 9126 i ISO/IEC 25010:2011- kvalitet u procesu životnog ciklusa softvera. Slika 3.1 pokazuje koji aspekti kvaliteta softvera koji obuhvataju korišćenje različitih koncepcata, planiranje, aktivnosti, tehnike i merenja, utiču na obezbeđenje kvaliteta softvera. Sa druge strane postoje modeli koji opisuju kvalitet softvera.



Slika 3.1 Koncept: Faktori planiranja, aktivnosti, tehnike i merenje kvaliteta softvera [Izvor: NM SE321-2020/2021.]

KARAKTERIZACIJA DEFEKATA (DEFECT CHARACTERIZATION)

Postoje mnoge taksonomije za defekte (nedostatke), iako su izvršeni pokušaji da se dobije konsenzus o taksonomiji otkaza, literatura ukazuje da postoji dosta razlika u terminima.

SQM procesi pronađe defekte. Karakterizacija tih nedostataka vodi do razumevanja proizvoda, olakšava korekcije procesa ili proizvoda i informiše projektni menadžment ili kupce o statusu procesa ili proizvoda. Postoje mnoge taksonomije za defekte - nedostatke (eng. **Defect Characterization**) i iako su izvršeni pokušaji da se dobije konsenzus o taksonomiji otkaza i defekata, literatura ukazuje da postoji dosta termina u upotrebi.

Karakterizacija defekta (anomalija) se takođe koristi u revizijama i pregledima gde vođa revizije često prezentuje listu anomalija dobijenih od članova tima radi razmatranja, na sastancima revizije.

Kako nove metode dizajna i jezika evoluiraju, uz napredak opšte softverske tehnologije, nove vrste defekata se pojavljuju i mnogo truda je potrebno za interpretiranje prethodno definisanih klasa.

Kada se prate defekti, softverski inženjer se interesuje ne samo za broj defekata, već i za tip. Informacija sama, bez neke klasifikacije, i nije od neke koristi u utvrđivanju osnovnih uzroka defekata, jer bi određene vrste problema trebalo da budu grupisane zajedno kako bi se donele odluke o njima.

Suština je da se uspostavi taksonomija defekata, koja je od značaja za samu organizaciju i za softverske inženjere.

SQM otkriva informacije u svim fazama softverskog razvoja i održavanja. Tipično, gde je upotrebljena reč „defekat“, odnosi se na „nedostatak“ kao što je definisano ispod. Ipak,

različite kulture i standardi mogu upotrebljavati nešto drugačija značenja ovih termina, što vodi pokušaju da se definišu. Delimične definicije uzete iz standarda (IEEE610.12-90) su:

- **Greška** (engl.**Error**): "Razlika...između izračunatog rezultata i tačnog rezultata"
- **Defekat** (engl.**Fault**): "Pogrešan korak, proces ili definicija podataka u kompjuterskom programu"
- **Otkaz** (engl.**Failure**): "[Netačan] rezultat nastao na osnovu defekta"
- **Pogreška** (engl.**Mistake**): "Ljudska akcija koja proizvodi netačan rezultat"

▼ Poglavlje 4

Merenja i ocenjivanje softvera

OSNOVE TEORIJE MERENJA

Teorija merenja rasvetjava sve prednosti i nedostake softverskih metrika. Merenje se definiše kao proces u kome se brojevi ili simboli pridružuju atributima entiteta u realnom svetu.

Softverska merenja, kao u bilo kojoj drugoj disciplini moraju se bazirati na naučnoj teoriji merenja da bi bila prihvatljiva i ispravna. Teorija merenja rasvetjava sve prednosti i nedostatke softverskih metrika. **Merenje se definiše kao proces u kome se brojevi ili simboli pridružuju atributima entiteta u realnom svetu na taj način da ih opišu prema jasno definisanim pravilima.** Pod entitetom se smatra objekt, kao što je osoba ili softverska specifikacija odnosno događaj npr. faza testiranja softverskog projekta. Atribut je osobina entiteta npr. visina osobe, dužina specifikacije, trajanje testne faze itd.

Pridruživanje brojeva i simbola mora osigurati intuitivnu i empirijsku opservaciju entiteta i atributa. **U većini situacija, atribut, iako je dobro poznat, ima različito intuitivno značenje za različite ljude i zbog toga se mora definisati model za entitete koji se mere.** Kada je definisan model onda postoje jasne empirijske relacije između entiteta i atributa. Npr. kod jednostavnog merenja dužine programa brojem linija izvornog koda (LOC) zahteva se dobro definisan model programa koji omogućava da se identificuje nedvosmislen broj linija koda. Dva osnova tipa merenja su :

- *direktna merenja i*
- *indirektna merenja.*

Direktno merenje atributa ne zavisi od merenja bilo kog drugog atributa za razliku od indirektnog merenja koje obuhvata merenje jednog ili više atributa.

Dva glavna razloga za merenje su:

- **procena i**
- **predviđanje.**

Merenje za predviđanje atributa A zavisi od matematičkog modela koji stavlja u relaciju egzistirajuće metrike atributa $A_1 \dots A_n$. Tačnost merenja za predviđanje zavisi od tačnosti metrika za procenu atributa $A_1 \dots A_n$ i zato tačna merenja ključnih atributa za postojeće projekte omogućavaju tačno predviđanje za buduće projekte. Nadalje, za tačna merenja za predviđanje model sam za sebe nije dovoljan, naime potrebno je definisati još procedura.

PROBLEMI I EFIKASNOST MERENJA

Premda nema univerzalno prihvaćene teorije merenja mogu se definisati problemi koji se rešavaju merenjem, kao i karakteristike da bi merenje bilo efikasno.

Premda nema univerzalno prihvaćene teorije merenja većina pitanja se svode na rešavanje sledećih problema:

1. **šta je merljivo, a šta nije ?**
2. **koji tipovi atributa mogu ili ne mogu biti mereni i na kojoj skali merenja?**
3. **kako se zna da su atributi realno mereni ?**
4. **kako odrediti skalu merenja ?**
5. **koji je prag greške prihvatljiv ?**
6. **koji su izveštaji o merenju značajni ?**

Merenje da bi bilo efektivno, mora biti :

1. **Fokusirano na specifične ciljeve.**
2. **Primenjeno u svim fazama životnog ciklusa proizvoda, procesa i resursa.**
3. **Interpretirano na bazi razumevanja organizacionog konteksta, okruženja i ciljeva.**

To znači da merenje mora biti definisano u top-down (od vrha prema dnu) stilu i mora biti fokusirano i bazirano na ciljevima i modelima.

Drugi pristup bottom-up (od dna ka vrhu) nije dobar, jer postoje mnoge observabilne karakteristike softvera (npr. vreme, broj grešaka, kompleksnost, linije koda, napor, produktivnost itd.) ali nije sasvim jasno koje metrike upotrebiti i na koji način ih interpretirati bez odgovarajućeg modela i ciljeva. U ovom delu lekcije ukratko će se dati pregled glavnih metoda i tehnika merenja i ocenjivanje softvera, koji će u narednim lekcijama biti detaljno opisane.

CILJEVI MERENJA

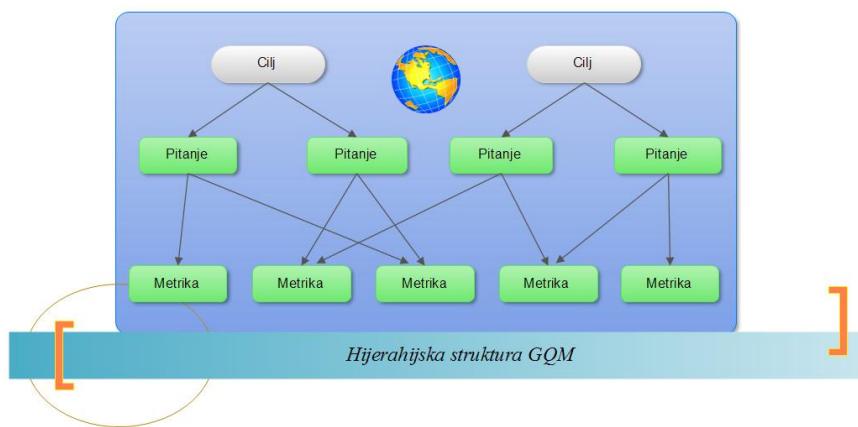
Bilo koja aktivnost merenja mora imati jasan cilj. U početku treba da bude jasno da li se merenje vrši zbog procene ili zbog predviđanja, moraju se znati entiteti interesantni za merenje

Bilo koja aktivnost merenja mora imati jasan cilj merenja (engl. Goal Question Metric – GQM). U početku, treba da bude jasno da li se merenje vrši zbog procene ili zbog predviđanja. Sledeće, moraju se znati entiteti interesantni za merenje. Zatim, treba odlučiti koji su atributi izabranih entiteta značajni. Mnogo godina su se softverski praktičari pitali: "za šta softverske metrike mogu biti upotrebljene?", zbog ignorisanja teorije merenja u softverskom inženjerstvu. Revolucionaran korak napred su napravili Basili i Rombach sa Goal/Question/

Metric (cilj/pitanje/metrika) paradigmom (GQM- Slika 4.1) koja postavlja ciljeve pre bilo koje aktivnosti merenja.

GQM model ima tri nivoa:

1. Konceptualni nivo
2. Operativni nivo
3. Kvantitativni nivo



Slika 4.1 Hijerahiska struktura GQM pristupa [Izvor: NM SE321-2020/2021.]

NIVOI GQM MODELA

Merenje po GQM modelu mora biti definisano u top-down (od vrha prema dnu) stilu i mora biti fokusirano i bazirano na ciljevima i modelima.

1. Konceptualni nivo (GOAL): Cilj se definiše za objekt tako što se vodi računa o različitim modelima kvaliteta i različitim tačkama gledišta. Objekti merenja su:

- Proizvodi koji nastaju u životnom ciklusu sistema, npr.: specifikacije, projekti, programi, testni podaci.
- Procesi koji obuhvataju aktivnosti vezane za vreme, npr.: specificiranje, projektovanje, testiranje, intervjuisanje.
- Rezursi upotrebljeni u procesu da bi se dobio produkt, npr.: personal, hardver, softver, radni prostor.

2. Operativni nivo (QUESTION): Ovaj nivo obuhvata niz pitanja koja se upotrebljavaju da specificiraju put za postizanje željenog cilja. Pitanja pokušavaju da karakterišu objekte merenja (proizvodi, procesi, rezursi) sa respektom na selektovani nivo izlazećeg kvaliteta sa različitim tačaka gledišta.

3. Kvantitativni nivo (METRIC): Ovaj nivo je predstavljen nizom podataka, pridruženih svakom pitanju, koji daju odgovor u kvantitativnoj formi. Podaci mogu biti objektivni i

subjektivni. Objektivni podaci zavise samo od objekta merenja ali ne i od tačke gledišta, za razliku od subjektivnih, koji zavise i od jednog i od drugog.

GQM model ima hijerarhijsku strukturu prikazanu na

slici 4.1. Kasnije su neki autori proširivali ovaj osnovni model na više nivoa. Ciljevi su podeljeni na podciljeve, podciljevi na domene, a domeni na poddomene. Pitanja su proširena pod pitanjima, a metrike su podeljene na karakteristične i pojedinačne. Proširenje se takođe odnosi i na tri nova nivoa: *prikupljanje podataka, modeliranje i implementaciju* iz kojih se može dobiti informacija o troškovima i dobiti, tj. o ekonomskoj opravdanosti programa za merenje. Ovaj model je nazvan GQM++. GQM prilaz je mehanizam za definisanje i interpretiranje merljivog softvera i može se upotrebiti zasebno ili u kombinaciji sa nekim drugim pristupima u smislu poboljšanja kvaliteta softvera.

Međutim, postoje situacije kada se koristi merenje softvera, a da ciljevi nisu jasno definisani. Ove situacije se obično dešavaju kada se mali broj metrika odnosi na različite ciljeve. Sledeći problem je - ko postavlja ciljeve. Najviše rukovodstvo može identifikovati loše ciljeve ili ciljeve za koje se ne mogu praktično izračunati metrike na inženjerskom nivou koji zahteva korisne i praktične metrike.

GQM je koristan za identifikaciju ciljeva merenja, ali da ne ukazuje na stvarne probleme u merenju (tehnički aspekt, kao što je skala merenja), ovaj kriticizam se proširuje na opravdanost, ekonomičnost i korektnost provere (jednostavan i intuitivan način).

▼ Poglavlje 5

Metode i tehnike merenja i ocenjivanje softvera

MCCABE-OVA METRIKA

Osnovni cilj merenja kompleksnosti sa McCabe-ovom metrikom je modularizacija softvera tako da rezultatni moduli budu pogodni za testiranje i održavanje.

Kao primer metrike, ovde je opisana [McCABE-ova metrika](#) kojom se meri kompleksnost softvera . McCabe je krenuo od toga da kompleksnost programa zavisi samo od strukture odluka u programu, tj. dodavanje ili oduzimanje funkcionalnih blokova ne utiče na kompleksnost programa. Nezavisnost kompleksnosti od veličine programa je ilustrovana time da program od 50 uzastopnih IF..THEN konstrukcija ima 33.5 miliona različitih upravljačkih puteva od kojih će mali procenat ikada biti testiran.

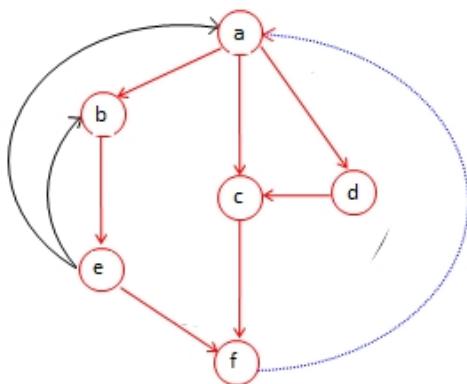
Osnovni cilj merenja kompleksnosti sa McCabe-ovom metrikom je modularizacija softvera tako da rezultatni moduli budu pogodni za testiranje i održavanje, što se odražava u vremenu uloženom za razvoj, i novcu potrošenom za održavanje. Merenje kompleksnosti se sastoji u određivanju ciklomatičnog broja koji indicira broj linearne nezavisnih upravljačkih puteva u programu. Kombinacijom ovih bazičnih puteva može se dobiti bilo koji put kroz program.

Pošto je McCabe-ova metrika vezana za poteškoće u testiranju programa, ona je prezentovana kao mera za računsku kompleksnost, jer se broj upravljačkih puteva indeksiran sa McCabe-ovom metrikom može odnositi i na broj mentalnih diskriminacija (komparacija) koje zahteva programski zadatak, jer dodatni upravljački putevi mogu učiniti program mnogo težim za razumevanje.

Ciklomatični broj $v(G)$ grafa sa N čvorova i E grana i P modula, se definiše sa : $v(G) = E - N + 2P$

gde su: E - broj grana na grafu programa, N - broj čvorova na grafu, P - broj nepovezanih komponenti ili se može posmatrati kao broj izlaza iz programa.

U strogo orijentisanom grafu, ciklomatični broj je broj linearne nezavisnih petlji. Za programe sa jednim ulazom i jednim izlazom, $v(G)$ je jednak broju odluka plus jedan. Za ovakve programe može se konstruisati orijentisani graf u kome svaki čvor odgovara bloku koda, a linije sa strelicama odgovaraju granama u programu. Primer na sl. 5.1 koji je McCabe upotrebio u svom originalnom radu ilustruje prirodu $v(G)$.



Slika 5.1 Graf programa [Izvor: NM SE321-2020/2021.]

GRAF PROGRAMA U MCCABE-OVA METRICI

Kompleksnost programa se može odrediti brojanjem predikata u programu, ali ako postoji graf, tada se kompleksnost određuje brojanjem regiona u grafu

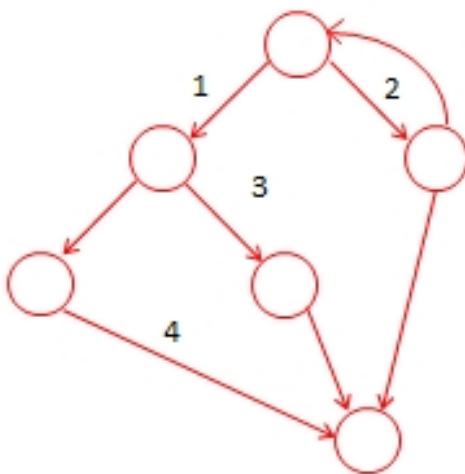
Svaki strukturalni program je napravljen od primitivnih struktura odnosno primova, koji su gradeći element programa i oni imaju različit doprinos ciklomatskom broju.

Postoji jednostavniji način posmatranja ovog načina za merenje kompleksnosti, bez potrebe za kreiranjem grafa, a koji uzima u obzir broj tačaka odluka u programu, plus jedan za izlazak iz programa:

$$M = D + 1$$

gde su: M -McCabe-ova metrika a D - broj tačaka odluke u programu.

Iz ovoga sledi da se kompleksnost programa može odrediti brojanjem tačaka odluke u programu a da se ne vodi računa o grafu. Ako imamo graf, tada se kompleksnost određuje vizuelnim putem, brojanjem regiona u grafu, što je prikazano na slici 5.2. U vezi sa testiranjem i održavanjem programa McCabe, na bazi empirijske evidencije, smatra se da kompleksnost programa, odnosno njegov ciklomatski broj ne treba da prelazi 10.



Slika 5.2 Regioni u grafu [Izvor: NM SE321-2020/2021.]

U ovom kontekstu, v može biti upotrebljen za osiguranje kvaliteta. Na osnovu cikomatskog broja mogu se predvideti očekivani troškovi testiranja i održavanja programa, i na osnovu toga izabratiti jedna od alternativa:

- ponovno pisanje programa u manjim modulima radi lakšeg održavanja ili
- nastaviti započeti posao.

METRIKE KÔDA

U vezi sa testiranjem i održavanjem programa McCabe, na bazi empirijske evidencije, smatra se da kompleksnost programa, odnosno njegov ciklomatski broj ne treba da prelazi 10.

Metrike koda su mere koje se određuju direktnim uvidom u izvorne fajlove programskog koda sistema. Ove metrike su na raspolaganju tek na kraju projekta, pošto je tek tada završen programski koda sistema.

Prva mera koja se koristila radi merenja složenosti softvera je bila broj linija koda. Ona je bila veoma pogodna i merodavna mera u proceduralnim jezicima. U njima je broj linija koda mogao manje više verno da odstoji veličinu softvera koji se implementira.

Najznačajnija metoda za procenu vremena na osnovu veličine softvera COCOMO II i danas koristi broj linija koda kao jedinicu mere veličine softvera. Linije koda se danas pretežno koriste u sistemskom softveru – veličine raznih operativnih sistema se mogu naći izražene u linijama koda.

Linije koda pored svoje jednostavnosti, razumljivosti i lakoće određivanja, imaju niz mana. Jedna od mana je utvrđivanje pravila definicije linije koda. Kao primer brojanja broja linija na različit način u zavisnosti od formata koda, može se videti kod na Slici 5.3 .

if(i>0)/*Prikaz*/ printf(i);	if(i>0) { // Prikaz printf(i); }
-------------------------------	---

Slika 5.3 Primer identičnog koda sa različitim brojem linija [Izvor: NM SE321-2020/2021.]

MERA - BROJA LINIJA KÔDA

U objektno orijentisanom softveru se može definisati niz dodatnih metrika koje se mogu određivati na osnovu analize kôda.

Iako je programski kod identičan, na desnoj strani bi se alatom za određivanje veličine koda dobila veća veličina merena u linijama koda. Da bi se prevazišli ovakvi problemi uvedene su dodatne mere broja linija koda:

1. LOC – broj fizičkih linija koda – koji predstavlja broj linija koda bez obzira na formatiranje,
2. ILOC – broj logičkih linija koda- koji predstavlja broj logičkih linija koda gde se uzima u obzir i način na koji su linije koda formatirane,
3. CLOC – broj linija koda koje predstavljaju komentare i opise u kodu.

Za određivanje fizičkih linija koda mogu se koristiti jednostavnii alati koji do broja linija koda dolaze pronalaženjem broja karaktera za prelazak u novi red u izvornom kodu, kao što Unix wc ili Windows PowerShell. Za određivanje broja logičkih linija i komentara je potrebno napraviti parser koji će pomoći određene translacione gramatike ili regularnih izraza, u skladu sa definicijom gramatike programskog jezika analizirati izvorni fajl i odrediti broj komentara i logičkih linija koda.

Broj linija koda je potpuno neprimenljiv u aplikacijama gde se koristi više različitih jezika istovremeno.

Kao očigledan primer se mogu videti web aplikacije koje se realizuju kombinovanjem java skripta i HTML-a na klijentskoj strani, standardnih jezika kao što su Java ili C# na serverskoj strani, i jezika specifičnih za baze podataka kao Oracle PL/SQL ili Microsoft T-SQL za implementaciju procedura i funkcija na serveru baze podataka. Kombinovanje podataka o broju linija koda iz različitih jezika je nemoguće, naročito ako se ima u vidu da za određene funkcionalnosti ne postoje ekvivalenti među njima.

Imajući u vidu nedostatke mere broja linija koda, vidi se da to nije pogodna mera za određivanje veličine softvera, sem u nekim specifičnim slučajevima. Iako broj linija koda nije pogodan za određivanje veličine, ova mera se može iskoristiti za određivanje nekih indirektnih mera u sistemu, kao što je kvalitet koda koji se izražava sa odnosom broja komentara i ukupnog broja linija koda, ili procenat generisanog koda.

U objektno orijentisanom softveru se može definisati niz dodatnih metrika koje se mogu određivati na osnovu analize koda. Ove metrike mogu biti Halstedova ili Mek Kejbova ciklomatska kompleksnost, dubina nasleđivanja, međuzavisnost modula i slično.

MERENJE FUNKCIONALNIH TAČAKA ZA PROCENU VELIČINE SOFTVERA

Mnogi softverski stručnjaci tvrde da funkcionalnost proizvoda daje bolju sliku o veličini proizvoda, nego njegova dužina.

Alati koji se koriste za analizu koda i merenje mogu biti integrисани u razvojna okruženja kao što je Visual Studio 2008 ili Eclipse, a mogu biti i nezavisni analizatori koda. Ovakvi alati mogu da analiziraju ili ceo sistem ili pojedinačne komponente (klase, funkcije) kako bi svakoj komponenti dale odgovarajuću kompleksnost. Ovakvi alati se uglavnom koriste po implementaciji projekta za pronalaženje problematičnih delova koda, pošto komponente koje imaju veću kompleksnost uglavnom predstavljaju probleme prilikom kasnijeg održavanja.

Prvi i osnovni zadatak za estimiranje troškova, kvaliteta i ostalih parametara softvera je da se pronađe broj funkcionalnih tačaka. **Mnogi softverski stručnjaci tvrde da funkcionalnost proizvoda daje bolju sliku o veličini proizvoda, nego njegova dužina.** U ranoj fazi projektovanja interesantnija je funkcionalnost kada su u pitanju napor i vreme od same fizičke veličine. Postoje različiti pristupi merenju funkcionalnosti, od kojih su najpoznatiji:

1. Albrecht-ove funkcionalne tačke (FP) (engl. functional points- FP)
2. DeMarco-ova funkcionalna "bang" metrika
3. Boehm-ov COCOMO model

Funkcionalne tačke, kao što i samo ime govori, mere količinu funkcionalnosti sistema opisanog specifikacijom. Da bi se izračunale FP, potrebno je identifikovati računljive diskretne komponente sledećeg tipa:

- **Eksterni ulazi** - procesni podaci ili upravljačke informacije koje dolaze izvana,
- **Eksterni izlazi** - procesni podaci i upravljačke informacije koje idu van,
- **Eksterni upiti** - interaktivni upiti koji zahtevaju izlaz,
- **Eksterne interfejs datoteke** - mašinski čitljive za druge sisteme,
- **Interne matične datoteke** - logičke interne datoteke.

Težinski faktori su prikazani u Tabeli na Slici 5.4 .

Bazni element	Težinski faktor		
	Jednostavan	Srednji	Kompleksan
Ulazi	3	4	6
Izlazi	4	5	7
Upiti	3	4	6
Ekst. datoteke	7	10	15
Inter. datoteke	5	7	10

Slika 5.4 Težinski faktori po Albrecht-u [Izvor: NM SE321-2020/2021.]

GLAVNA NAMENA FP METRIKE

Merenje veličine produkta FP metrikom se koristi u predviđanju napora i troškova u ranoj fazi životnog ciklusa softvera (specifikacija, projektovanje).

Albrecht je funkcionalne tačke FP računao po formuli :

- broj ulaza * 4 +
- broj izlaza * 5 +
- broj upita * 4 +
- broj interfejsa * 7 +
- broj matičnih datoteka * 10,

s tim da težinski faktori mogu varirati +/- 25%, zavisno od kompleksnosti programa. Pomoću gornje formule dobiju se nepodešene funkcionalne tačke UFC. Podešene funkcionalne tačke se dobiju množenjem UFC faktorom tehničke kompleksnosti TCF koji ima 14 doprinosećih faktora (on-line ulazi, on-line ažuriranje, kompleksnost interfejsa, ponovna upotreba, komuniciranje sa podacima itd.) i varira od 0.65 do 1.35.

Glavna namena FP je:

- *merenje veličine proizvoda koja se koristi u predviđanju napora i troškova u ranoj fazi životnog ciklusa softvera (specifikacija, projektovanje); predviđanje broja linija izvornog koda;*
- *konverzija veličine projekta pisanog u jednom jeziku u veličinu projekta pisanog u drugom jeziku, jer su FP nezavisne od jezika; merenje produktivnosti projekata koji su pisani u više jezika; normalizacija u odnosu na FP (defekti/FP, čovek/mesec/FP itd.) i upotreba FP kao baze za ugovaranje.*

Nedostaci FP su:

- *teškoća računanja, jer različiti ljudi mogu da različito računaju FP i potrebna je vrlo detaljna specifikacija; za razliku od LOC-a, ne mogu se automatski računati. Neke*

empirijske studije sugeriju da FP nisu vrlo dobre za predviđanje napora i da su nepotrebno kompleksne - problem sa subjektivitetom tehnološkog faktora;

- problem sa aplikacionim domenom (komercijalne aplikacije, aplikacije u realnom vremenu, naučne aplikacije)

Očekivani broj linija izvornog koda dobijen iz FP (ili na neki drugi način) omogućava predviđanje drugih karakteristika softvera.

BASILI-JEVI MODELI ZA MERENJE RESURSA

Ovakav model se derivira na osnovu lokalnog okruženja i istorijskih podataka.

Na osnovu resurs modela, koji se sastoji od empirijskih jednačina, obično se predviđa:

- E - napor u čovek/mesecima
- D - trajanje projekta
- DOC - broj linija dokumentacije

Međutim, treba imati u vidu da se empirijski podaci dobijaju na osnovu limitiranog i specifičnog broja projekata i resurs model nije odgovarajući za sve klase softvera.

Basili je resurs modele svrstao u četiri klase:

1. *statički jednovarijabilni model*
2. *statički multivarijabilni model*
3. *dinamički multivarijabilni model*
4. *teoretski model*

Statički jednovarijabilni model ima sledeću formu:

$$\text{resurs} = c_1 * (\text{očekivane karakteristike})^{c_2}$$

Konstante c_1 i c_2 se deriviraju iz prethodnih projekata. Kao primer, Pressman navodi studiju Walston-a i Felix-a, baziranu na 60 razvojnih projekata na osnovu kojih su došli do sledećih rezultata, odnosno modela:

$$E = 5.2 * L^{0.91}$$

$$D = 4.1 * L^{0.36}$$

$$S = 2.47 * E^{0.35}$$

$$DOC = 0.54 * E^{0.06}$$

$$DOC = 49 * L^{1.01}$$

E, D (trajanje projekta i kalendarsko trajanje projekta) i DOC su izvedeni na osnovu L, tj. broja linija izvornog koda u hiljadama.

Alternativno, S (potrebni ljudi) i trajanje projekta mogu biti izračunati iz deriviranog ili predviđenog napora E.

Gornje jednačine se ne mogu uzeti generalno, jer zavise od okruženja i specifičnosti aplikacija. Ovakav model se derivira na osnovu lokalnog okruženja i istorijskih podataka.

OSTALI TIPIČNI MODELI ZA MERENJE RESURSA

Dinamički multivarijabilni model projektuje zahteve za resursima kao funkciju vremena.

Ostali tipični modeli su:[3]

1. $E = 5.5 * 0.73 L^{1.16}$ (Bailey-Basili model)
2. $E = 3.2 * L^{1.05}$ (Bohem-ov jednostavni model)
3. $E = 3.0 * L^{1.12}$ (Bohem-ov srednji model)
4. $E = 2.8 * L^{1.20}$ (Bohem-ov kompleksni model)
5. $E = 5.288 * L^{1.047}$ (Doty model)

Ove modele je veoma teško porebiti zbog različitih definicija L (npr. da li su uračunati komentari ili ne) i E (da li se odnosi samo na kodiranje ili i na ostale faze životnog ciklusa softvera).

Upotrebom statičkog jednovarijabilnog resurs modela Schaefer predviđa broj čovek/meseci potrebnih za održavanje softvera godišnje E.maint na sledeći način:

$$E.maint = ACT * 2.4 * L^{1.05},$$

gde se ACT definiše sa:

$$ACT = (L - \text{za sistem koji se održava}) / CI,$$

pri čemu je CI broj linija koda koje su modifikovane ili dodane za vreme jednogodišnjeg održavanja.

Statički multivarijabilni model ima sledeću formu:

$\text{resurs} = c_{11} * e_1 + c_{21} * e_2 + \dots$, gde je e_i i-ta softverska karakteristika a c_{ij} empirijski derivirana konstanta za i-tu karakteristiku softvera.

Dinamički multivarijabilni model projektuje zahteve za resursima kao funkciju vremena.

Teoretski model pretpostavlja specifičnu distribuciju napora u toku razvoja projekta i na osnovu toga prati ponašanje resursa.

▼ Poglavlje 6

Grupna vežba - Testiranje softvera ATM uređaja

TESTIRANJE ATM UREĐAJA

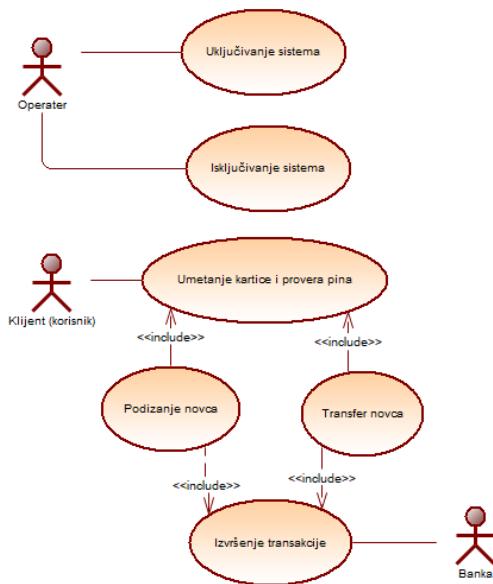
Plan obuhvata samo softver koji opslužuje ATM, kao i zahteve koji su definisani u početnoj fazi projekta

U suštini, ATM predstavlja interfejs (vezu) između klijenta i informacionog sistema banke – koji obavlja sve značajne operacije.

Predmet ovog plana testiranja neće biti veza ATM-a i banke, kao ni bančin informacioni sistem (IS). Veza sa bankom će, u prvim fazama razvoja, biti simulirana raznim stabovima, a IS banke će biti analiziran samo do nivoa potrebnog da bi se utvrdili tipovi podataka i struktura baze koja pohranjuje podatke o klijentima i njihovim računima. Takođe, neće biti testirana ni hardverska ispravnost ATM-ova, jer se pretpostavlja da je to obavio proizvođač pre isporuke. Ovaj plan obuhvata samo softver ATM-a koji opslužuje ATM, kao i zahteve koji su definisani u početnoj fazi projekta.

U ovom slučaju, mi imamo tri glavna učesnika: klijent, operater i banka. Svi oni imaju uticaj na funkcionisanje ATM-a i svi treba da budu opsluženi, po potrebi. Dijagram slučajeva korišćenja prikazan na slici 6.1. može biti vrlo koristan prilikom definisanja Test slučajeva ATM proizvoda. Slučajevi korišćenja su:

- 1. Uključivanje sistema
- 2. Isključivanje sistema
- 3. Umetanje kartice i provjera pina
- 4. Podizanje novca



Slika 6.1 Dijagram slučajeva korišćenja koji prikazuje funkcionalnost ATM-a [Izvor: NM SE321-2020/2021.]

- 5. Transfer novca
- 6. Izvršenje transakcije

od kojih će poslednja četiri biti testirana.

Vreme izrade grupne vežbe 45 minuta.

MCCABE-OVA SIKLOMATSKA SLOŽENOST - UMETANJE KARTICE I PROVERA PINA

Metrika koja će biti primenjena na ATM uređaju omogućava dobijanje rezultata koji predstavlja najmanji broj test slučaja za taj deo sistema.

Metrika koja će biti primenjena na ATM uređaju omogućava dobijanje rezultata koji predstavlja najmanji broj test slučaja za taj deo sistema.

Ciklomatični broj $v(G)$ grafa sa N čvorova i E grana i P modula se definiše sa : $v(G) = E - N + 2P$

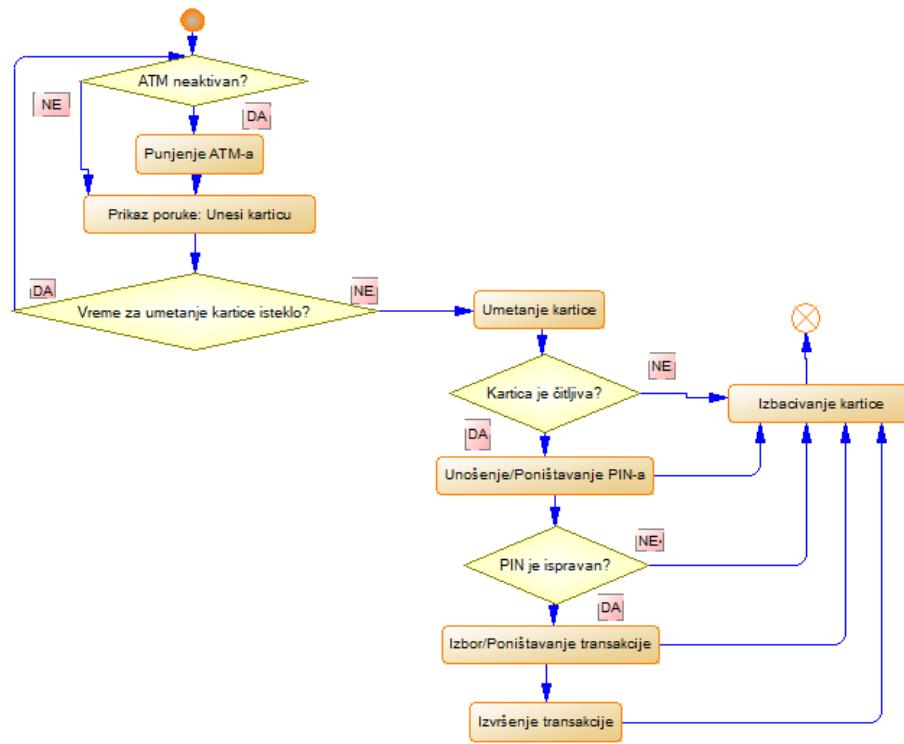
gde su:

- E - broj grana na grafu programa,
- N - broj čvorova na grafu,
- P - broj nepovezanih komponenti ili se može posmatrati kao broj izlaza iz programa

Na osnovu dijagrama prikazanog na slici 6.2 kojim se opisuje način izvršenja slučaja korišćenja Umetanje kartice i provera pina

$E = 18$; $N=11$; $P=1$; $M = E - N + 2P$; $M=18-11+2= 9$ što znači da za ovaj slučaj korišćenja potrebno je realizovati minimum 9 test slučaja (UC) na osnovu metrike.

ZADATAK: Opišite svih 9 test slučajeva za testiranje ovog UC-a.



Slika 6.2 Dijagram aktivnosti za slučaj korišćenja Umetanje kartice i provera pina [Izvor: NM SE321-2020/2021.]

MCCABE-OVA SIKLOMATSKA SLOŽENOST - PODIZANJA NOVCA

Primena McCabe-ove siklomatske složenosti na slučaju korišćenja - podizanje novca.

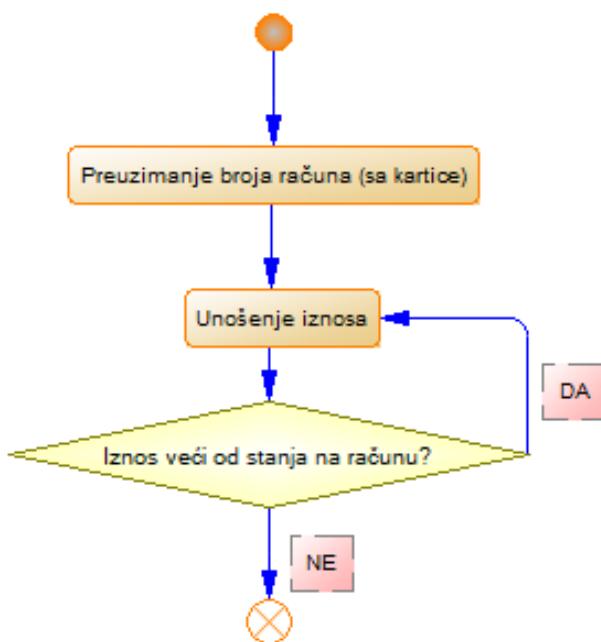
Na osnovu slike 6.3 koja pokazuje način izvršenja slučaja korišćenja Podizanje novca za ovaj slučaj

- $E = 5$
- $N=5$
- $P=1$

$$M=5-5+2=2$$

što pokazuje da je za ovaj slučaj korišćenja potrebno realizovati minimum dva test slučaja.

ZADATAK: Opišite 2 test slučaje za testiranje ovog UC-a.



Slika 6.3 Dijagram aktivnosti slučaja korišćenja Podizanje novca [Izvor: NM SE321-2020/2021.]

MCCABE-OVA SIKLOMATSKA SLOŽENOST - TRANSFER NOVCA

Primena McCabe-ove siklomatske složenosti na slučaju korišćenja - transfer novca.

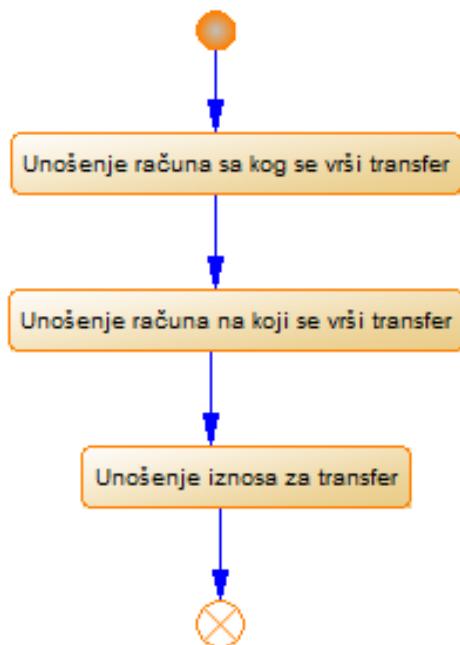
Na osnovu slike 6.4 koja pokazuje način izvršenja slučaja korišćenja Transfer novca za ovaj slučaj

- E= 4
- N= 5
- P= 1

$$M=4-5+2=1$$

što znači da je za ovaj slučaj korišćenja potrebno realizovati minimum jedan test slučaj.

ZADATAK: Opišite test slučaj za testiranje ovog UC-a.



Slika 6.4 Dijagram aktivnosti slučaja korišćenja Transfera novca [Izvor: NM SE321-2020/2021.]

MCCABE-OVA SIKLOMATSKA SLOŽENOST - OBAVLJANJE TRANSAKCIJE

Primena McCabe-ove siklomatske složenosti na slučaju korišćenja - obavljanje transakcije.

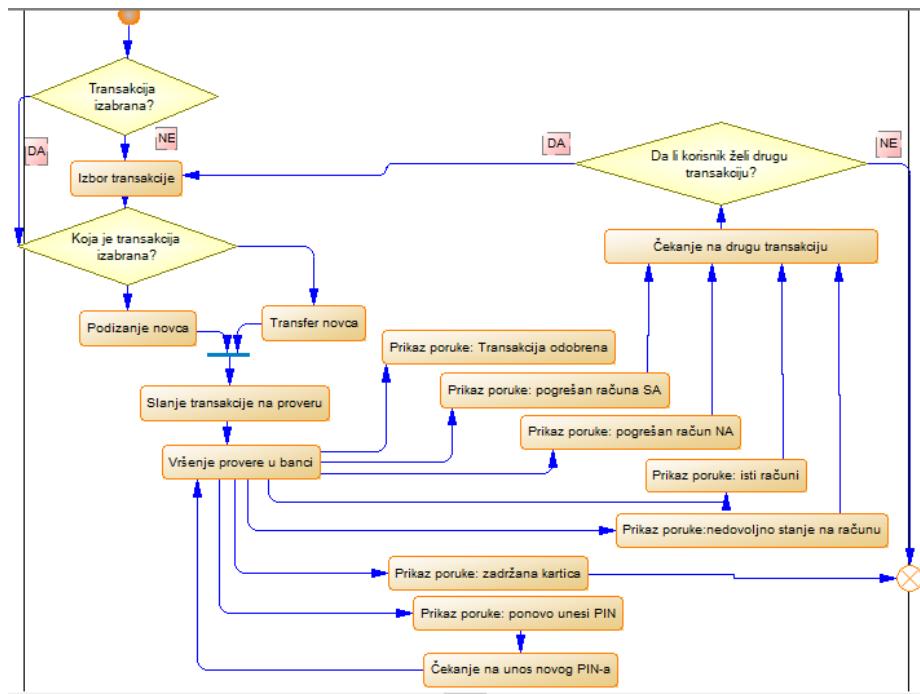
Na osnovu slike 6.5 koja pokazuje način izvršenja slučaja korišćenja Obavljanje transakcije za ovaj slučaj

- E= 21
- N= 25
- P=1

$$M=25-19+2=8$$

na osnovu čega proizilazi da je za ovaj deo sistema potrebno je realizovati osam test slučajeva.

ZADATAK: Opišite jedan od test slučajeve za testiranje ovog UC-a.



Slika 6.5 Dijagram aktivnosti za slučaj korišćenja Obavljanje transakcije [Izvor: NM SE321-2020/2021.]

✓ Poglavlje 7

Vežba za samostalni rad

TESTIRANJE NEKE FUNKCIJE ISUM SISTEMA

Na osnovu obrađenih aktivnosti testiranja softvera uraditi testiranje upotrebljivosti ISUM sistema.

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

Na osnovi korišćenja izabranog dela aplikacije:

1. Nacrtajte dijagram slučajeva korišćenja sa izdvojenim slučajevima korišćenja onako kako ih vi vidite. (**vreme izrade 15 minuta**)
2. Za svaki pojedinačni slučaj korišćenja nacrtajte dijagrame aktivnosti koji pokazuju logiku izvršenja slučaja korišćenja (**vreme izrade 30 minuta**)
3. Primenom McCabe-ova siklomatska složenosti za svaki slučaj korišćenja izračunajte minimum jedan test slučajeva (**vreme izrade 15 minuta**)
4. Opišite sve moguće test slučajeve (**vreme izrade 30 minuta**)

✓ Poglavlje 8

Domaći zadatak

DRUGI DOMAĆI ZADATAK - VREME IZRADE 150 MIN.

Nakon druge lekcije potrebno je uraditi drugi domaći zadatak

Odabratи aplikaciju kreiranu za potrebe projekta na nekom drugom predmetu (koji ste do sada uradili) i za nju:

Nacrtajte dijagram slučajeva korišćenja sa izdvojenim slučajevima korišćenja onako kako ih vi vidite.

Za jedan izabrani slučaj korišćenja nacrtajte dijagrame aktivnosti koji pokazuju logiku izvršenja slučaja korišćenja

Primenom McCabe-ova sikiomatska složenosti za izabrani slučaj korišćenja izračunajte broj test slučajeva koje je potrebno realizovati

Opišite sve moguće test slučajeve.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ02-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br.Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

nikola.petrovic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ02.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Kvalitet softvera – zahtevi, modeli, metrika, obezbeđenje, merenje i ocenjivanje

MODELI KVALITETA, METRIKA, OBEZBEĐENJE, MERENJE I OCENJIVANJE

QMS je definisan u ISO 9001 kao "organizacijska struktura, odgovornost, procedure, procesi i resursi za uvođenje sistema upravljanje kvalitetom radi postizanje postavljenih ciljeva".

Pojam „kvaliteta“ nije tako jednostavan kao što može da izgleda. Za bilo koji projektovani proizvod postoji više željenih kvaliteta koji su relevantni za posebnu perspektivu proizvoda koji se moraju razmatrati i utvrditi za vreme za koje se definišu zahtevi proizvoda. Karakteristike kvaliteta mogu biti obavezne ili ne, ili mogu biti tražene do višeg ili nižeg nivoa, i među njima se mogu izvršiti kompromisi.

Troškovi kvaliteta mogu biti podeljeni na troškove prevencije, troškove procene, unutrašnje troškove otkaza i spoljašnje troškove otkaza. Iako u softverskoj industrijskoj praksi ne postoji usvojen standard kvaliteta softvera koji određuje šta to čini dobar kvalitet softvera, generalno, kvalitetan softver je onaj koji zadovoljava potrebe kupca, doprinosi profitu, ne proizvodi ozbiljne probleme u poslovanju i po zdravlje ljudi. Kvalitet softvera je teško definisati jer postoje različiti aspekti i termini koji su definisani i opisani u nizu standarda kao što su pomenuti: ISO 9000-3, ISO/IEC 9126 (Kvalitet softverskog proizvoda) i ISO/IEC 14598 (Vrednovanje softverskog proizvoda) i drugi.

Standardizovani model ocenjivanja kvaliteta softvera (SSQA) koji je razvila Motorola, proširen je uključivanjem metodologije mini-periodične ocene kvaliteta softvera (KvS) koje su vrlo važne u kontinuiranom poboljšanju kvaliteta od strane različitih proizvodnih timova. QMS je definisan u ISO 9001 kao "organizacijska struktura, odgovornost, procedure, procesi i resursi za uvođenje sistema upravljanje kvalitetom u kompaniji radi postizanje postavljenih ciljeva u vezi kvaliteta".

Softverska merenja, kao u bilo kojoj drugoj disciplini, moraju se bazirati na naučnoj teoriji merenja da bi bila prihvatljiva i ispravna. Teorija merenja rasvetljava sve prednosti i nedostatke softverskih metrika. Merenje se definiše kao proces u kome se brojevi ili simboli pridružuju atributima entiteta u realnom svetu na taj način da ih opišu prema jasno definisanim pravilima. Pod entitetom se smatra objekt, kao što je osoba ili softverska specifikacija odnosno događaj npr. faza testiranja softverskog projekta. Atribut je osobina entiteta npr. visina osobe, dužina specifikacije, trajanje testne faze itd. Pridruživanje brojeva i simbola mora osigurati intuitivnu i empirijsku opservaciju entiteta i atributa. U većini situacija, atribut, iako je dobro poznat, ima različito intuitivno značenje za različite ljudi i zbog toga se mora definisati model za entitete koji se mere. Kada je definisan model, onda postoje jasne empirijske relacije između entiteta i atributa.

LITERATURA ZA LEKCIJU 02

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
3. dr Miladin Stefanović, mr Slobodan Mitrović, mr Milan Erić, MODEL KVALITETA SOFTVERA, Festival kvaliteta 2006., 33. Nacionalna konferencija o kvalitetu, Kragujevac, 10. - 12. maj 2006.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Mr Boris Todorović, EVALUACIJA KVALITETA SOFTVERA PREMA ISO 9126 STANDARDU, DOI 10.7251/POS1208099T.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>



SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Testiranje softvera – osnove,
ciljevi, pristupi, procesi,
planiranje i sprovodenje

Lekcija 03

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Lekcija 03

TESTIRANJE SOFTVERA – OSNOVE, CILJEVI, PRISTUPI, PROCESI, PLANIRANJE I SPROVOĐENJE

- ✓ Testiranje softvera – osnove, ciljevi, pristupi, procesi, planiranje i sprovođenje
- ✓ Poglavlje 1: Nivoi testiranja po fazama razvoja softvera
- ✓ Poglavlje 2: Klasifikacija tehnika testiranja po ulaznom domenu
- ✓ Poglavlje 3: Tehnike bazirane na kodu (Code-based)
- ✓ Poglavlje 4: Specifikacije i ocene testiranja
- ✓ Poglavlje 5: Izveštaj o testiranju
- ✓ Poglavlje 6: Pokazna vežba -Testiranje ATM uređaja
- ✓ Poglavlje 7: Domaći zadatak
- ✓ Testiranje softvera - ciljevi i pristupi

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Životni ciklus testiranja softvera ili STLC je sveobuhvatna grupa test povezanih koraka koji pokušavaju da isporuče kvalitetniji proizvod.

Testiranje je aktivnost koja se izvodi identifikacijom defekata i problema radi procene kvaliteta proizvoda, kao i za njegovo unapređenje. Testiranje softvera se sastoji iz dinamičke verifikacije ponašanja programa na konačnom skupu testiranja, pogodno selektovanog iz uglavnog beskonačnih područja izvršavanja, u odnosu na očekivano ponašanje.

Tokom godina, shvanje testiranja softvera je evoluiralo i danas je konstruktivnije. Testiranje se više ne posmatra kao aktivnost koja počinje samo posle završetka faze kodiranja i ima ograničenu svrhu otkrivanja nedostataka. Danas se testiranje softvera posmatra kao aktivnost koja bi trebalo da obuhvati celi razvojni proces i važan je deo aktuelnog konstruisanja proizvoda. Zaista, planiranje testiranja trebalo bi da počne u ranim fazama analize zahteva, i planovi i procedure testiranja moraju se sistematično i kontinualno poboljšavati paralelno sa razvojem. Ove aktivnosti planiranja i dizajniranja testova su konstituisane kao koristan ulaz dizajnerima zbog potencijalnih grešaka na visokom nivou (kao npr., dizajnerski propusti ili kontradikcije, i propusti ili dvomislenosti u dokumentaciji).

U softverskom kvalitetu (eng.[Software Quality Management Techniques](#)), tehnike upravljanja softverskim kvalitetom kategorisane su u statičke tehnike (bez izvršavanja koda) i dinamičke tehnike (izvršavanje koda). Obe kategorije su korisne. Ova oblast znanja se fokusira na dinamičke tehnike.

Životni ciklus testiranja softvera ili STLC je sveobuhvatna grupa test povezanih koraka koji pokušavaju da isporuče kvalitetniji proizvod. Testiranje je postalo važan fenomen za vreme i nakon razvoja bilo kog softverskog projekta. Najveći deo literature koja obrađuje oblast testiranja softvera je posvećena pojedinačnim, nekorelisanim tehnikama ili strategijama testiranja softvera sa unapred zadatim kriterijumima generisanja test slučajeva za koje se očekuje da pouzdano i nepogrešivo otkrivaju moguće greške u SWUT (eng.[Software Under Test](#)) koji se vidi i razume tj. identificuje preko: tekstualnih opisa, sintakse, dijagrama toka, toka izračunavanja, ili realizovanih funkcija SWUT.

▼ Poglavlje 1

Nivoi testiranja po fazama razvoja softvera

PROCES TESTIRANJA U FAZI RAZVOJA

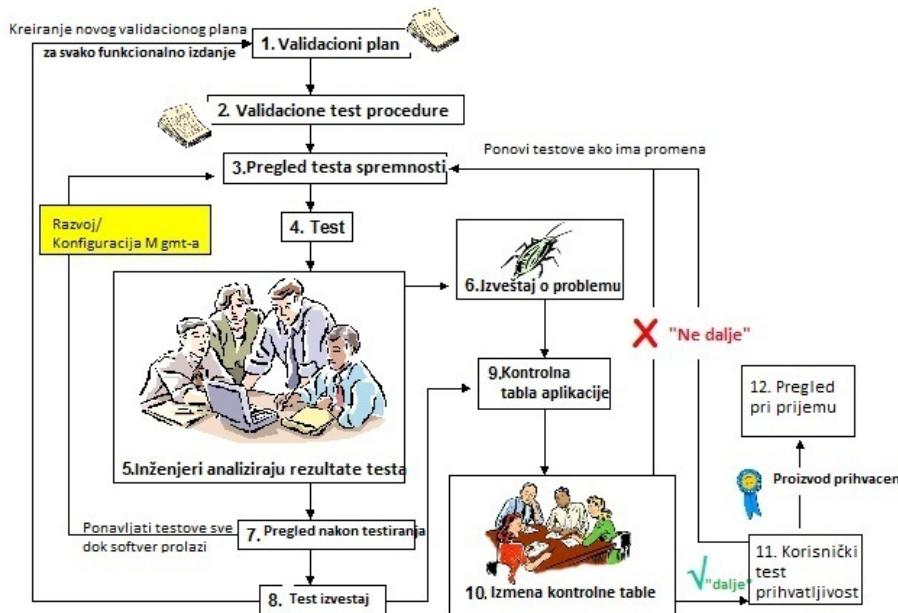
Testiranje je postalo važan fenomen za vreme i nakon razvoja bilo kog softverskog projekta.

Životni ciklus testiranja softvera (STLC) je sveobuhvatna grupa test povezanih koraka koja ima za cilj da isporuči kvalitetniji proizvod. Testiranje je postalo važan fenomen za vreme i nakon razvoja bilo kog softverskog projekta.

Tok testiranja je prikazan na Slici 1.1 . Testiranje se sastoji od sledećih aktivnosti:

1. Planiranje testiranja (*određuje se predmet, cilj i razlog testiranja na osnovu zahteva, modela i drugih ulaza testiranja, gde se testiranje vrši, kada se vrši i ko izvršava testove*)
2. Dizajn testova (*određuje kako se sprovodi testiranje na osnovu artefakta tj. test primera*)
3. Implementacija testova (*pravljenje višestruko upotrebljivih test skriptova koji realizuju test primere*)
4. Izvršavanje testova (*izvršavanje implementacije testa radi provere funkcionalnosti sistema*)
5. Evaluacija testova (*procena testova tj. validnosti izvršavanja testa, analiza izlaza, pregled zbirnih rezultata, uticaj promene zahteva i ulaza na plan testiranja*)

Svaka od ovih aktivnosti ima ulaze i izlaze. U procesu testiranja koriste se i različiti alati koji pospešuju proces i daju bolji uvid u rezultate (npr.Rational, Test Complete i sl.).



Slika 1.1 Tok testiranja - STLC [Izvor: NM SE321 - 2020/2021.]

MODELI PROCESA RAZVOJA I TESTIRANJA SOFTVERA

Do danas su razvijeni mnogi modeli procesa razvoja i testiranja softvera.

Do danas su razvijeni mnogi modeli procesa razvoja softvera. Modeli se koriste pojedinačno ili u kombinaciji, i teže ostvarenju određenog nivoa kvaliteta i ponovljivom i stabilnom procesu proizvodnje softvera. Mnogi od njih su već pomenuti i objašnjeni:

1. **Model vodopada,**
2. **Prototipski pristup i njegove varijacije,**
3. **Spiralni model,**
4. **Model procesa iterativnog razvoja,**
5. **Objektno orijentisani proces razvoja,**
6. **Cleanroom metodologija,**
7. **Proces prevencije defekata i dr.**

TESTIRANJA SOFTVERA PRE I U FAZI ISPORUKE

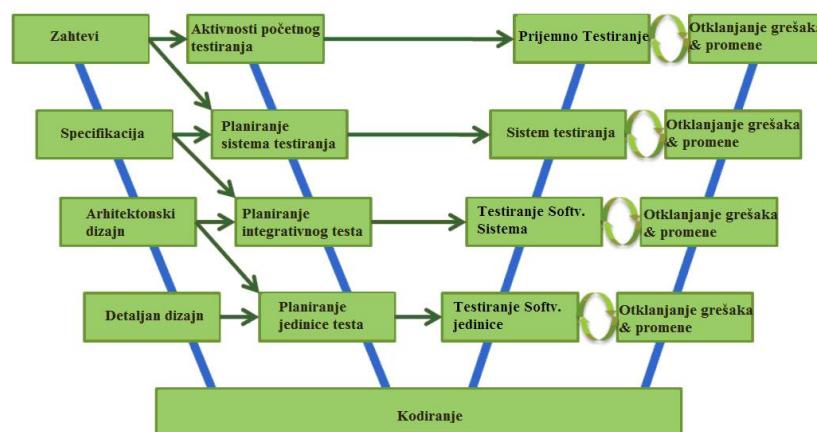
Sve aktivnosti testiranja do faze integracije se nazivaju testiranje softvera pre isporuke, dok se aktivnosti posle kodiranja i jediničnog testiranja nazivaju testiranje u fazi isporuke.

Procesi razvoja softvera variraju prema stepenu implementacije u različitim organizacijama, pa čak i po projektima. Prema okviru određenog procesa razvojni tim obično definiše

specifične elemente kao što su implementacione procedure, metode i alate, metrike i merenja itd. Neki modeli procesa odgovaraju tačno određenim tipovima projekata u određenom okruženju, ali uspeh projekta zavisi pre svega od zrelosti implementacije. Dimenzije povezane sa celokupnim sistemom upravljanja kvalitetom jako utiču na rezultate softverskih projekata. Za utvrđivanje zrelosti procesa organizacije ili projekata, primenjuju se različite metode: SEI (Software Engineering Institute) metode za utvrđivanje zrelosti, Istraživanja softverske produktivnosti (SPR), Melcom Baldrige disciplina i procesi, i ISO 9000 proces registracije. SEI i SPR metode se odnose na softverski proces, a ostala dva okvira su procesi kvaliteta i standardi upravljanja kvalitetom.

Shodno ovim modelima projektovanja softvera treba napraviti i model procesa testiranja softvera, kao što je za **W-model testiranja** dat na slici 1. 2. **Proces testiranja mora se odvijati tokom svih faza životnog ciklusa.**

Sve aktivnosti testiranja do faze integracije se nazivaju još i testiranje pre isporuke, dok se aktivnosti posle kodiranja i jediničnog testiranja nazivaju testiranje tokom isporuke. Za ovaj proces moraju se realizovati standardizovani propisi i procedure



Slika 1.2 W-model razvoja i proces testiranja softvera [Izvor: NM SE321 - 2020/2021.]

koje definišu način rada i aktivnosti test odeljenja, kao i svih učesnika u razvoju. Sve aktivnosti članova tima koji učestvuju u realizaciji procesa testiranja moraju biti dokumentovane i moraju pokriti sledeće:

- Odgovornosti i zaduženja za planiranje testa, izvršenje i evaluaciju.
- Ciljeve testa, tipove koji će se primeniti, raspored i zaduženja kod testiranja kroz plan testiranja.
- Odgovarajući materijal (dokumenta, test primeri i dr.). Test materijal je podložan periodičnim aktivnostima kontrole kvaliteta.

TESTIRANJE PREMA ŽIVOTNOM CIKLUSU SOFTVERA

Testiranje softvera treba da započne u ranoj fazi životnog ciklusa aplikacije, ne samo u tradicionalnoj validacionoj fazi testiranja, nakon što je kodiranje završeno.

Testiranje u životnom ciklusu softvera znači da se testiranje obavlja paralelno sa razvojnim ciklusom i da je kontinualan proces. **Testiranje softvera treba da započne u ranoj fazi životnog ciklusa aplikacije, ne samo u tradicionalnoj validacionoj fazi testiranja, nakon što je kodiranje završeno.**

S obzirom na testiranje posle razvoja tj. u fazi isporuke, svi predstavljeni modeli ranije su definisani na različite načine:

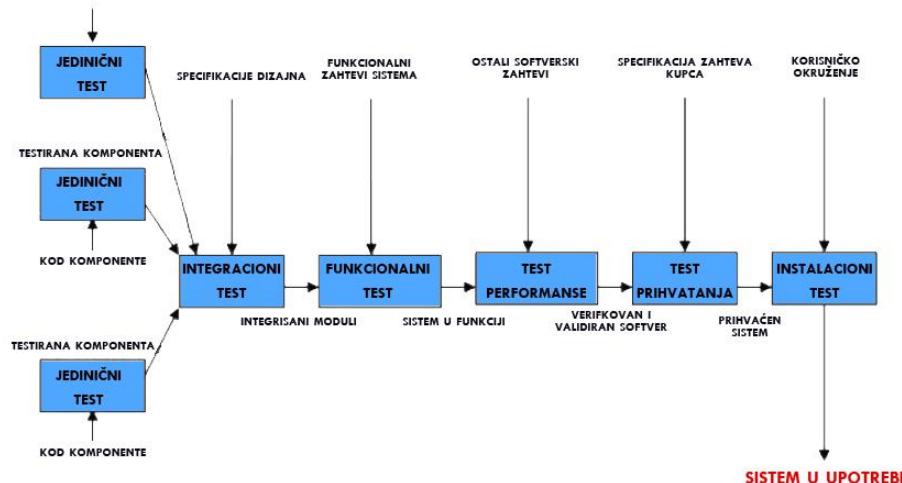
- **test aktivnosti najpre počinju nakon implementacije**
- **veza između različitih faza testiranja i osnova za test nije jasna**
- **uska veza između testa, ispravka greške i promena zadatka tokom faze testiranja nije jasna.**

Testiranje softvera se obično izvršava na različitim nivoima tokom razvoja i održavanja procesa (Slika 1.3). Odnosno, cilj testiranja može da varira: jedan modul, grupa takvih modula (povezanih svrhom, upotrebom, ponašanjem ili strukturu), ili čitav sistem.

Mogu se razlikovati tri velike **faze testa**, nazvane

1. **Jedinica** (engl. **Unit**),
2. **Integracija** (engl. **Integration**) i
3. **Sistem** (engl. **System**).

Ne podrazumeva se nijedan model procesa, niti se prepostavlja da bilo koja od ove tri faze ima veći značaj od preostale dve.



Slika 1.3 Faze testiranja softvera [Izvor: NM SE321 - 2020/2021.]

TESTIRANJE JEDINICE, TESTIRANJE INTEGRACIJE I TESTIRANJE SISTEMA

Testiranje jedinice: verifikuje funkcionisanje delova softvera;

Testiranje integracije: verifikacije interakciju između komponenti softvera;

Testiranje sistema razmatra ponašanje sistema

- 2. **Testiranje jedinice:** verifikuje funkcionisanje delova softvera u izolaciji koji se posebno mogu testirati. U zavisnosti od konteksta, to mogu biti individualni podprogrami ili veće komponente napravljene od usko vezanih jedinica. Test jedinica je preciznije definisana u **IEEE Standard for Software Unit Testing (IEEE1008-87)**, koji takođe opisuje integrисани pristup sistematskog i dokumentovanog testiranja jedinice. Obično, testiranje jedinice nastaje sa pristupom kodu koji se testira i uz pomoć *debugging* alata, i može da uključuje programere koji su napisali kod.
- 3. **Testiranje integracije** je proces verifikacije interakcije između komponenti softvera. Klasične strategije testiranja integracije, kao što su **top-down** ili **bottom-up** (odozgo na dole, i odozdo na gore), koriste se sa tradicionalnim softverom sa hijerarhijskom strukturom. Moderne sistematske strategije integracije pre su vođene arhitekturom, što navodi na integraciju komponenti softvera ili podsistema baziranih na identifikovanim funkcionalnim temama. Testiranje integracije je kontinualna aktivnost, na svakoj fazi softverski inženjeri moraju izdvojiti perspektive nižih nivoa i koncentrisati se na perspektive nivoa koji integrišu. Osim za male, jednostavne softvere, sistematske, inkrementalne integracione strategije testiranja se obično preferiraju stavljanjem svih komponenti zajedno, što se slikovito zove „big bang“ testiranje.
- 3. **Testiranje sistema** se razmatra uz ponašanje čitavog sistema. Većina funkcionalnih grešaka već bi trebalo da su identifikovane tokom testiranja jedinice i integracije. Testiranje sistema se obično smatra prikladnim za poređenje sistema i ne-funkcionalnih sistemskih zahteva, kao što su bezbednost, brzina, tačnost, i pouzdanost. Eksterni interfejsi prema drugim aplikacijama, uslužnim programima, hardverskim uređajima ili operativnom okruženju, takođe su procenjeni na ovom nivou.

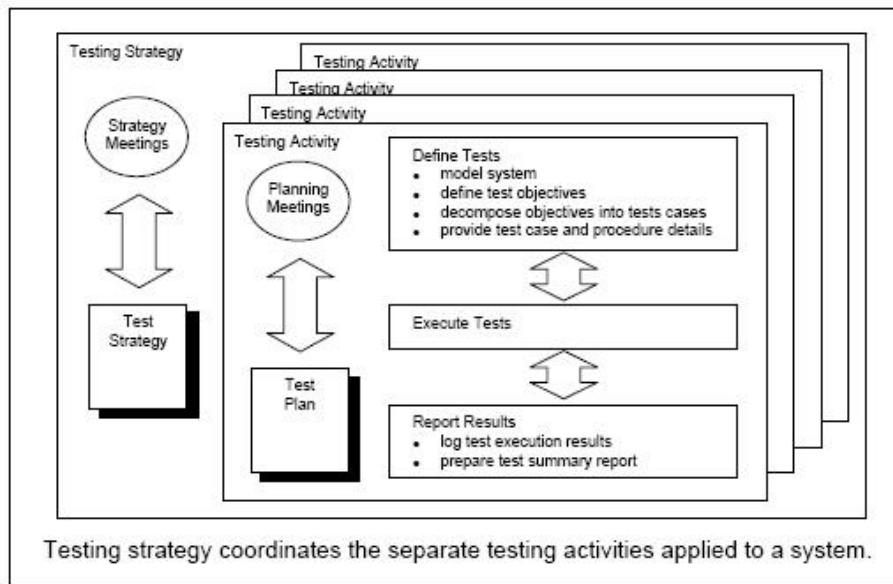
CILJEVI TESTIRANJA

Postavljanje ciljeva precizno, kvantitativnim terminima, omogućava da tokom procesa testiranja bude ustanovljena kontrola

>Testiranje se izvršava pri određenim ciljevima, koji su manje ili više eksplicitno izraženi, i sa varirajućim stepenima preciznosti.

Postavljanje ciljeva precizno, kvantitativnim terminima, omogućava da kontrola bude ustanovljena tokom procesa testiranja. Testiranje može biti namenjeno za verifikaciju različitih osobina. Slučajevi testa se mogu osmisliti kako bi proverili da li su funkcionalne specifikacije tačno implementirane, što se u raznim slučajevima u literaturi naziva **testiranje usaglašenosti** (engl. conformance testing), **testiranje korektnosti** (engl. correctness testing), ili **funkcionalno testiranje** (engl. functional testing). Ipak, nekoliko drugih nefunkcionalnih osobina se mogu testirati, uključujući performanse, pouzdanost, i korisnost, pored ostalih.

Drugi značajni ciljevi za testiranje uključuju (ali nisu ograničeni na) merenje pouzdanosti, procenu korisnosti i prihvatanja, za koje bi trebalo primeniti različite pristupe. Može se zapaziti da objekat testa varira sa ciljem testa; uopšteno, *različite svrhe se ispunjavaju različitim nivoom testiranja*. Neke vrste testiranja su prikladnije za softverske pakete pravljene po meri, **testiranje instalacija** (engl. installation testing), na primer; i druge generičke proizvode, kao **beta testiranje** (Slika 1.4).



Slika 1.4 Planiranje različitih testnih aktivnosti [Izvor: NM SE321 - 2020/2021.]

VRSTE TESTOVA

Testiranje prihvatanja/kvalifikacije, testiranje instalacije, Alfa i beta testiranje, testiranje usaglašenosti/Funkcionalno Testiranje/Testiranje tačnosti, postizanje pouzdanosti i procena

- **Testiranje prihvatanja /testiranje kvalifikacije** (engl. Acceptance/qualification testing): proverava ponašanje sistema u odnosu na zahteve kupaca, bez obzira na to kako su izraženi; kupci preduzimaju ili određuju tipične zadatke kako bi proverili da su njihovi zahtevi zadovoljeni, ili da je organizacija identifikovala njih za ciljno tržište softvera. Ova aktivnost testiranja može ali ne mora uključivati one koji razvijaju sistem.
- **Testiranje instalacije** (engl. Installation testing): Nakon kompletiranja testiranja softvera i prihvatljivosti, softver se može verifikovati u odnosu na instalaciju u ciljnoj sredini. *Testiranje instalacije može se posmatrati kao sistemsко testiranje izvođeno još jednom prema zahtevima hardverske konfiguracije.* Procedure instalacije mogu, takođe, biti verifikovane.
- **Alfa testiranje i beta testiranje**(engl. Alpha and beta testing): *Pre izbacivanja softvera, ponekad se daje malom, reprezentativnom skupu potencijalnih korisnika na probu, ili unutar (alfa testiranje) ili eksterno (beta testiranje).* Ovi korisnici izveštavaju o problemima sa proizvodom. Alfa i beta upotreba su često nekorelisane i često se na njih ne upućuje u planu testa.
- **Testiranje usaglašenosti / Funkcionalno Testiranje /Testiranje tačnosti** (engl. Conformance testing/Functional testing/Correctness testing): *usmereno je na validaciju bez obzira da li posmatrano ponašanje testiranog softvera odgovara svojim specifikacijama.*
- **Postizanje pouzdanosti i procena** (engl. Reliability achievement and evaluation): *Kako bi se pomogla identifikacija grešaka, testiranje bi trebalo da poboljša pouzdanost.* Nasuprot tome, statističke mere pouzdanosti se mogu izvesti slučajnim generisanjem slučajeva

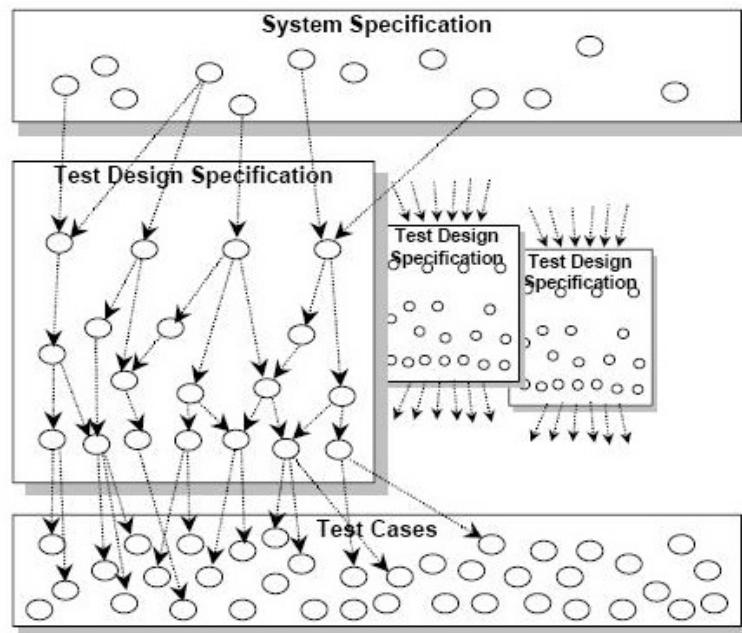
testa prema operativnom profilu. Korišćenjem modela rasta, oba cilja se mogu zajedno sprovesti.

- **Regresiono testiranje** (engl. **Regression testing**): Prema (IEEE610.12-90), regresiono testiranje je „selektivno retestiranje sistema ili komponenti kako bi se verifikovalo da modifikacije nisu proizvele neželjene efekte ...“ U praksi, *ideja je da se pokaže da softver koji je prethodno prošao testove i dalje to može*. Beizer to definiše kao bilo koje ponavljanje testova sa namerom da se pokaže da je ponašanje softvera nepromjenjeno, osim ukoliko se to ne zahteva. Očigledno je da se mora izvršiti kompromis između osiguranja datog regresionim testiranjem svakog puta kada se izvrši promena, i resursa potrebnih za to.

OSTALE VRSTA TESTOVA

Testiranje performansi, testiranje stresa, back-to-back testiranje, testiranje oporavka, testiranje konfiguracije itd.

- **Testiranje performansi** (engl. **Performance testing**): *Posebno je namenjeno verifikaciji da softver zadovoljava posebne zahteve performansi, na primer, kapaciteta i vremena odgovora.* Posebna vrsta testiranja performansi je testiranje diska (volume testing), u kome se ispituju interni programi ili limiti sistema.
- **Testiranje stresa** (engl. **Stress testing**): *Stres testiranje vežba softver na maksimumu opterećenja, kao i preko toga.*
- **Back-to-back testiranje**: *Jedan test se izvršava na dve implementirane verzije softverskog proizvoda i rezultati se porede.*
- **Testiranje oporavka** (engl. **Recovery testing**): *Testiranje oporavka namenjeno je verifikaciji sposobnosti restartovanja softvera nakon „sloma“.*
- **Testiranje konfiguracija** (engl. **Configuration testing**): *U slučajevima kada je softver izgrađen kako bi služio različitim korisnicima, testiranje konfiguracije analizira softver pod različito određenim konfiguracijama.*
- **Testiranje upotrebljivosti** (engl. **Usability testing**): *Ovaj proces procenjuje koliko je lako krajnjim korisnicima da nauče da rade sa softverom, uključujući dokumentaciju; koliko efektivno funkcije softvera podržavaju zadatke korisnika; i, na kraju, sposobnost oporavka nakon grešaka korisnika.*
- **Razvoj vođen testom** (engl. **Test-driven development**): *Ovaj razvoj nije na prvi pogled tehnika, zalažeći se za upotrebu testova kao surogata dokumentovane specifikacije potreba pre nezavisne provere da softver ima pravilno implementirane zahteve (Slika 1.5).*



Slika 1.5 Razvoj vođen testom: Dekompozicija zadatka testiranja obuhvata inkrementalno deljenje sistema na u ciljeve testiranja sve dok se ne identificuje skup slučajeva testiranja [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 2

Klasifikacija tehnika testiranja po ulaznom domenu

KLASIFIKACIJA TEHNIKA TESTIRANJA SOFTVERA

Polazeći od postavljenih ciljeva testiranja, razvijen je veliki broj tehnika tako da nijedna nije dovoljna ni sveobuhvatna, već poseduju i jake i slabe strane u odnosu na objektivni potencijal

Najveći deo literature koja obrađuje oblast testiranja softvera je posvećena pojedinačnim, nekorelisanim tehnikama ili strategijama testiranja softvera sa unapred zadatim kriterijumima generisanja test slučajeva za koje se očekuje da pouzdano i nepogrešivo otkrivaju moguće greške u softveru koji se testira (engl. Software Under Test - SWUT). Test koji se vidi i razume, se identificuje preko: tekstualnih opisa, sintakse, dijagrama toka, toka izračunavanja, ili realizovanih funkcija SWUT.

Testiranje softvera je staro koliko i istorijarazvoja softvera i danas je integralni deo ciklusa razvoja softvera, pažljivo struktuiran u zavisnosti od vrste softverskog proizvoda, iskustva projektanata i osoblja koje sprovodi testiranje, a na bazi jezika koji se koristi/koriste za implementaciju rešenja softverskog proizvoda. Pošto ne postoji izvodljiv i jeftin tj. efikasan teorijski metod verifikacije korektnosti dizajna i implementacije softverskog rešenja, testiranje softvera igra glavnu ulogu u validaciji (vrednovanju) oba procesa u razvoju softvera.

Ciljevi testiranja softvera su da se:

1. otkriju skriveni i unapred nepoznat broj defekata u softveru koji su generisani u fazi izrade projektnih zahteva, dizajna i implementacije;
2. obezbedi nivo poverenja u ocenu kvaliteta projektovanog softvera tj. nivo zaostalih grešaka u softveru
3. smanje troškovi u čitavom veku eksploatacije softverskog proizvoda.

Zbog ovih i ranije navedenih razloga, razvijen je veliki broj tehnika i strategija testiranja softvera polazeći od postavljenih ciljeva tako da nijedna nije dovoljna ni sveobuhvatna, već poseduju i jake i slabe strane u odnosu na objektivni potencijal (mogućnost) i koje su u literaturi analizirane na bazi njihove primene u praksi u ispunjenju postavljenog cilja, odnosno kriterijuma izbora test slučajeva tj. test ansambla.

Tehnike za detekciju grešaka (TDG) u softveru, generalno spadaju u tri kategorije analitičkih aktivnosti:

1. statička analiza,
2. dinamička analiza i

3. formalna analiza .

STATIČKO TESTIRANJE

Statičko testiranje obuhvata „analizu projektnih zahteva (specifikacije) softvera, projekta (dizajna), programskog koda i/ili ostalih proizvoda SDLC sprovedenih manuelno ili automatski.

Statičko testiranje se svodi na testiranje bez izvršenja softvera i obuhvata „analizu projektnih zahteva (specifikacije) softvera, projekta (dizajna), programskog koda i/ili ostalih proizvoda SDLC sprovedenih manuelno ili automatskim sredstvima, bez izvršavanja tj. stavljanja u operativnu funkciju objekata analize, a sa ciljem da se analiziraju leksičke, sintaksne, logičke karakteristike objekta analize, za razliku od drugih tehnika koje su usmerene na analizu ponašanja objekta testiranja“.

Ova vrsta tehnika testiranja softvera se koristi za ispitivanje proizvoda SDLC u svim njegovim fazama razvoja. Primeri statičke analize uključuju ali se time ne završavaju, na primer: inspekcija, pregledi, čitanje programskog koda, analiza algoritama praćenja tokova (engl. **Tracing**) i dr.

Drugi primeri su zasnovani na grafičkim predstavama softvera koji se testira, kao što su: analiza dijagrama toka, prikaz maštine sa konačnim brojem stanja, dijagram uzroka-posledica, a koji su često podržani alatima i sredstvima za automatizaciju ovih aktivnosti analize.

Tradicionalno, tehnika statičke analize se primjenjuje u fazi izrade projektnih zahteva (specifikacije) softvera, izradi projekta (dizajna) i programskog koda ali se može koristiti, takođe za proveru dokumentacije softverskog proizvoda, posebno u analizi izbora test slučajeva, za verifikaciju praćenja provere pojedinačnih projektnih zahteva (specifikacije) softvera i adekvatnosti testiranja softvera u odnosu na iste.

Da li se zahteva ispitivanje sintakse izvornog programskog koda pri statičkom testiranju softvera?

Ako se zahteva, onda je to klasa tehnika testiranja softvera nazvana **sintaksno testiranje**, ako se ne zahteva - onda je to klasa tehnika testiranja softvera nazvana **semantičko testiranje**.

Najdokumentovanija, citirana i formalno ustanovljena tehnika statičkog testiranja jeste **tehnika inspekcije**. U toku inspekcije, test inženjer ispituje programski kod i svu dokumentaciju sistema koji se ispituje sa ciljem da se otkriju greške svake vrste u SWUT.

TESTIRANJE NA OSNOVU RAZLIČITIH ULAZNIH DOMENA

Testiranje zavisi od ulaznog domena i može se vršiti kao slučajno testiranje, ekvivalentno deljenje, i analiza graničnih vrednosti

Kod slučajnog testiranja (engl. **Random testing**) testovi se generišu slučajno. Ovaj vid testiranja spada pod testiranje domena ulaza zasnovano na specifikaciji, jer da bismo mogli da uzimamo slučajne tačke unutar njega, oblast ulaza mora biti poznata. Efektivnost ovih testova oslanja se na znanje inženjera, koje se može izvesti iz mnogo izvora: posmatranog ponašanja proizvoda tokom testiranja, upoznatosti sa aplikacijom, platformom, procesom neuspeha, tipom mogućih grešaka i mana, rizika povezanog sa posebnim proizvodom i slično.

Ulezni domen kod testiranja može biti podeljen na kolekciju podskupova, ili ekvivalentnih klasa, koje se smatraju ekvivalentima prema određenim relacijama, i reprezentativan skup testova (ponekad samo jedan).

Ekvivalentno deljenje (engl. **Equivalence partitioning**): Domen ulaza je podeljen na kolekciju podskupova, ili ekvivalentnih klasa, koje se smatraju ekvivalentima prema određenim relacijama, i reprezentativan skup testova (ponekad samo jedan) je uzet iz svake klase.

Analize graničnih vrednosti (engl. **Boundary-value analysis**): Slučajevi testa izabrani su na, ili blizu granica oblasti ulaza promenljivih, sa obrazloženjem kao osnovom da mnoge greške imaju tendenciju da se koncentrišu blizu ekstremnih vrednosti ulaza.

Proširenje ove tehnike su testiranja robusnosti, gde su slučajevi testa takođe izabrani izvan ulazne oblasti promenljivih, kako bi se testirala robusnost programa prema neočekivanim ili pogrešnim ulazima.

Kako se vrši izbor test slučajeva ?

1. Ako se izbor test primera bazira na funkcijama ili strukturi softvera, tada se primenjena tehnika naziva funkcionalno - ispitivanje ili strukturno - ispitivanje, respektivno.
2. Ako se pak, izbor test primera vrši nasumično u odnosu na upotrebu tj. primenu softvera, tada se ta tehnika naziva testiranje slučajnim primerima (engl. **Random testing**).

Koja vrsta podataka o sistemu se dobija nakon primene tehnika testiranja?

Ako se ponašanje SWUT očekuje prema unapred i uvek poznatom scenariju tada je u pitanju determinističko testiranje, dok je testiranje statističko kada se test primeri biraju prema nekoj od funkcija raspodele slučajnih ulaznih promenljivih.

DINAMIČKO TESTIRANJE

Ako primenjena tehnika za testiranje softvera zahteva izvršavanje softvera u pitanju je dinamičko testiranje softvera. Deli se na metod crne, bele ili sive kutije.

Ako primenjena tehnika za testiranje softvera zahteva izvršavanje softvera u pitanju je dinamičko testiranje softvera.

Ako primenjena tehnika zahteva ispitivanje izvornog programskog koda u dinamičkom testiranju softvera, onda je to klasa tehnika testiranja softvera nazvana metod bele-kutije, ako ne zahteva - onda je to klasa tehnika testiranja softvera nazvana metod crne-kutije.

Sve tehnike testiranja softvera koje zahtevaju izvršavanje programa zasnovane su na:

1. *informacijama iz programa - P,*
2. *specifikacijama programa - S ili*
3. *koriste oba izvora informacija pa se tada nazivaju kombinovanim tehnikama TS.*

Zbog toga se testovi iz T ansambla klasificuju kao metod

1. bele kutije,
2. crne kutije ili
3. sive kutije.

Test slučajevi odabrani na osnovu informacija iz specifikacije softvera spadaju u metode testiranja softvera po principu crne kutije jer ne koriste informacije iz programskog koda P.

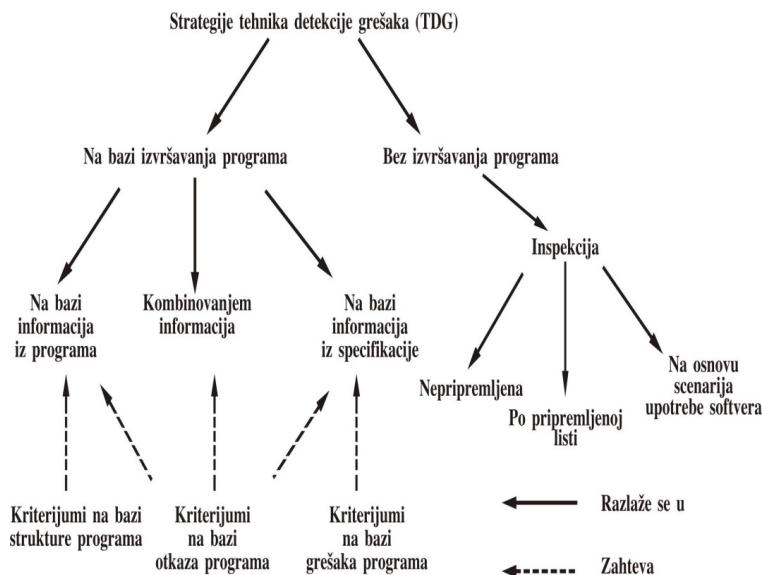
Test slučajevi odabrani na osnovu programskog koda spadaju u metode bele ili staklene kutije (engl. **White** ili **glass-box**), dok **metod koji kombinuje informacije iz S i P** naziva se kombinovani metod ili metod sive kutije (engl. **Gray-box**).

TESTIRANJE NA OSNOVU ADEKVATNOSTI TEST SLUČAJEVA

Test slučajevi se mogu uzeti na osnovu strukture programa, poznatim otkazima ili greškama iz iskustva

Na osnovu slike 2.1, dalja dekompozicija metoda testiranja softvera se zasniva na izboru kriterijuma adekvatnosti test slučajeva iz beskonačnog skupa mogućih, bez obzira na korišćen izvor informacija S ili P ili oba.

Test slučajevi iz T su adekvatno "dobri" u pogledu detekcije mogućih grešaka u softveru ako informacije o potencijalnim greškama crpe iz strukture programa, poznatim otkazima ili greškama iz iskustva, pa se metode testiranja softvera klasificuju kao:



Slika 2.1 Klasifikacija strategije testiranja softvera [Izvor: NM SE321 - 2020/2021.]

1. *Testovi bazirani na strukturi* (engl. *structurally-based*)
2. *Testovi bazirani na otkazima* (engl. *fault-based*)
3. *Testovi bazirani na greškama* (engl. *error-based*).

Tako na primer, **testovi zasnovani na strukturi** programa P, vrše izbor test slučajeva T, koji dovode do izvršavanja određene kontrolne strukture ili promenljive iz programskega koda P.

Testovi bazirani na otkazima vrše izbor test slučajeva po kriterijumu da pokažu da SWUT nema opšte poznate klase otkaza koje programeri prave u svom radu. Iako se najčešće fault-based tehnike testiranja softvera svrstavaju u metode na bazi informacija iz programa, neki autori smatraju da tehnike na bazi poznatih otkaza u softveru nastaju usled nerazumevanja funkcionalnih zahteva iz specifikacije S, tako da postavljaju kriterijum C za izbor test slučajeva u T i na bazi informacija iz P ili S ili kombinovano iz oba izvora informacija.

I na kraju, **testovi zasnovani na greškama** u tumačenju i implementaciji funkcionalnih zahteva iz S, postavljaju kriterijum za izbor test slučajeva upravo polazeći od pogrešnog tumačenja specifikacije i misije softvera u SWUT.

TESTOVI ZASNOVANI NA GREŠKAMA

Jedan od ciljeva testiranja je otkrivanje koliki je potencijal grešaka moguć, i mnoge tehnike su razvijene na osnovu ovoga, tehnike koje pokušavaju da „slome“ program itd.

Za to se prave i koriste liste za proveru (engl. *software inspection checklist*) potencijalnih grešaka, odnosno problema koji mogu nastati pri funkcionisanju sistema.

Dok je lista za proveru, unapred pripremljena, vrlo korisna u povećanju efikasnosti otkrivanja grešaka i problema u objektu (proizvodu) ispitivanja u odnosu na nepripremljen i nasumice

(engl. **Ad-hoc**) obavljen pregled (inspekcija), dotle je još efikasnija tehnika inspekcije na bazi scenarija, (engl. **Scenario-based reading**) tehnika inspekcije.

Očigledno, ove tehnike se ne koriste podjednako često. Uključene su u listu one koje bi inženjer trebalo da zna. Jedan od ciljeva testiranja je otkrivanje koliki je potencijal grešaka moguć, i mnoge tehnike su razvijene na osnovu ovoga, tehnike koje pokušavaju da „slome“ program, izvršavanjem jednog ili više testova izvedenog iz identifikovanih klasa ekvivalenta egzekucija.

Vodeći princip ovih tehnika je da budu što sistematičnije u identifikaciji reprezentativnog skupa programskega ponašanja; na primer, posmatranje podklaša oblasti ulaza, scenario, stanja i toka podataka. Teško je naći homogenu bazu za klasifikaciju svih tehnika, i ona koja je ovde data mora se smatrati za kompromisnu. Klasifikacija je bazirana na tome kako su testovi generisani iz intuicije i iskustva softverskih inženjera, kao i specifikacije, strukture koda, (realne ili veštačke) greške koje bi trebalo naći, prirode aplikacije.

Ponekad su ove tehnike klasifikovane kao bele-kutije, ako se test oslanja na informacije kako je softver dizajniran ili kakvi su kodovi, ili crne-kutije ukoliko se slučajevi testiranja oslanjaju samo na ulaz/izlaz ponašanje.

TEHNIKE ZASNOVANE NA KOMBINACIJI DVE ILI VIŠE TEHNIKA

Tu spadaju: Ad hoc testiranje, Istraživačko testiranje, Tabela odluka itd.

Poslednja kategorija odnosi se na kombinovanu upotrebu dve ili više tehnika. Bazirane na intuiciji i iskustvu softverskih inženjera

1. **Ad hoc testiranje:** Verovatno najrasprostranjenija tehnika ostaje ad hoc testiranje: *testovi se izvode oslanjajući se na inženjersko umeće, intuiciju, i iskustvo sa sličnim programima.* Ad hoc testiranje može biti korisno za identifikaciju specijalnih testova, onih koje nije lako zabeležiti formalizovanim tehnikama.
2. **Istraživačko testiranje** (engl. **Exploratory testing**): Ovo testiranje je definisano kao *simultano učenje, dizajn testa i izvršavanje testa; odnosno, testovi nisu određeni unapred u okviru utvrđenog plana testa, već su dinamički dizajnirani, izvršeni i modifikovani.*
3. **Tabela odluka** (engl. **Decision table**): Tabele odluka predstavljaju logičke veze između uslova (grubo rečeno, ulaza) i akcija (grubo, izlaza). *Slučajevi testa su sistematično dobijeni razmatranjem svake moguće kombinacije uslova i akcija.* Tehnika povezana sa ovom je grafičko predstavljanje uzrok-efekat.

Još neke tehnike testiranja su:

1. **Testiranje konačnog stanja bazirano na mašini** (engl. **Finite-state machine-based**): Modeliranjem programa kao mašine konačnih stanja, testovi se mogu odabratи sa ciljem da se pokriju stanja i tranzicije na njima.
2. **Testiranje iz formalnih specifikacija** (engl. **Testing from formal specifications**): Davanje specifikacije u formalnom jeziku omogućava *automatsku derivaciju funkcionalnih slučajeva testa i istovremeno, obezbeđuje referentni izlaz, za proveru*

rezultata testa. Metode postoje i za dobijanje slučajeva testa iz onih baziranih na modelu ili algebarskih specifikacija. Formalne tehnike analize uključuju rigorozne matematičke tehnike i metode provere projektnih zahteva (specifikacije) softvera, projekta (dizajna) i programskog koda, kao tehnika otkrivanja grešaka u navedenim aktivnostima SDLC.

3. Testiranje prihvatanja (engl. *Acceptance testing*), vrši se *pre prihvatanja softverskog proizvoda ili sistema od strane kupca*.

Još neki primeri testiranja koji se vrše prilikom dinamičke analize su: modelovanje i simuliranje, analiza vremena i količine izvršenih zadataka softverski realizovanih funkcija, ispitivanje prototipova SWUT, a koji se mogu primenjivati u svim fazama SDLC.

TEHNIKE ZA OTKRIVANJE GREŠAKA U SOFTVERU PO FAZAMA RAZVOJA SOFTVERA

Ove tehnike i metrika za ocenu njihovog doprinosa kvalitetu softvera, primenljive su za ocenu kvaliteta procesa testiranja softvera kao najvažnije aktivnosti obezbeđenja i kontrole kvaliteta

Tabela na Slici 2.2 daje spisak mogućih tehnika za detekciju grešaka u SWUT i njihovu primenu po fazama razvoja softvera koji se testira.

U ovoj tabeli, nazivi kolona predstavljaju skraćenice za pojedine faze izrade softvera i to :

1. **R za fazu izrade projektnih zahteva (specifikacije) softvera,**
2. **D za fazu izrade projekta (dizajna),**
3. **I za fazu izrade programskog koda (implementacije dizajna),**
4. **T za fazu testiranja**
5. **IC za fazu instalacije i provere SWUT kod kupca**
6. **OM u toku operativnog rada i održavanja SWUT, respektivno.**

Ove tehnike i metrika primenljive su za ocenu kvaliteta procesa testiranja softvera kao najvažnije aktivnosti obezbeđenja i kontrole kvaliteta softvera u SDLC u svim fazama SDLC, bez obzira koji model SDLC je primenjen (engl. *waterfall, spiral*).

TEHNIKA DETEKCIJE GRESAKA (TDG)	R	D	I	T	IC	OM
Analiza algoritama	✓	✓	✓	✓		✓
Testiranje s kraja – na kraj				✓		
Analiza graničnih vrednosti					✓	
Analiza toka kontrole programa	✓	✓	✓			✓
Analiza baze podataka	✓	✓	✓	✓		✓
Analiza toka podataka	✓	✓	✓			✓
Dijagram toka podataka	✓					
Tabela odlučivanja (tabela istinitosti)	✓					
Provera za stolom (čitanje programskog koda)				✓		
Analiza dokumenata	✓	✓	✓	✓	✓	✓
Sejanje grešaka					✓	
Mašine sa konačnim stanjima	✓					
Formalne metode (formalna verifikacija)	✓	✓				
Analiza toka informacija		✓				
Inspekcija	✓	✓	✓	✓	✓	✓
Analiza interfejsa	✓	✓	✓			
Testiranje interfejsa					✓	
Mutaciono testiranje i analiza					✓	
Testiranje performansi					✓	
Izrada prototipa / animacije	✓	✓	✓			
Regresiona analiza i testiranje	✓	✓	✓	✓	✓	✓
Pregled projektnih zahteva	✓					
Revizije	✓	✓	✓	✓	✓	✓
Analiza osjetljivosti					✓	
Simulacije	✓	✓	✓	✓	✓	✓
Analiza vrednosti podataka i vremenskih intervala	✓	✓	✓			✓
Isecanje programskog koda			✓			
Analiza krivudanja programskega koda			✓			
Testiranje na opterećenja					✓	
Simbolička provera			✓			
Test sertifikat					✓	✓
Analiza putanja (trajnica)	✓	✓	✓	✓		
Prolaz kroz program	✓	✓	✓	✓	✓	✓

Slika 2.2 Tehnike za otkrivanje grešaka u softveru [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 3

Tehnike bazirane na kodu (Code-based)

TEHNIKE BAZIRANE NA KONTROLNOM PROTOKU

Kriterijumi bazirani na kontrolnom protoku su usmereni za pokrivanje svih naredbi ili blokova naredbi u programu, ili njihovih posebnih kombinacija.

Ove tehnike su usmerene za pokrivanje svih naredbi ili blokova naredbi u programu, ili njihovih posebnih kombinacija. Nekoliko kriterijuma je predloženo kao pokrivanje uslov/odluka.

Najjači od kriterijuma baziranih na kontrolnom toku je **testiranje puta** (engl. **path testing**), koje ima za cilj izvršavanje svih **od-ulaska-do-izlaska putanja kontrolnog toka u grafikonu toka**.

Pošto testiranje putanja generalno nije izvodljivo zbog petlji, u praksi se koriste drugi manje strogi kriterijumi, kao što je testiranje izjava, testiranje grana, i testiranje uslov/odluka.

Adekvatnost takvih testova meri se procentima; na primer, kada su sve grane izvršene makar jednom u testu, kaže se da je postignuta 100% pokrivenost grana.

1. **Kriterijum baziran na toku podataka** (engl. **Data flow-based criteria**): U ovakovom testiranju, kontrolni grafik toka beleži se sa informacijama o tome kako su promenljive u programu definisane, korišćene, i ubijene (nedefinisane). Zahteva da se za svaku promenljivu i svaki segment kontrolnog toka iz definicije te promenljive dok se izvršava koristi putanja. Kako bi se smanjio broj potrebnih putanja, koriste se slabije strategije .
2. **Referentni modeli za testiranje bazirano na kodu** (engl. **flowgraph, call graph**): Iako sama po sebi nije tehnika, kontrolna struktura programa grafički se prikazuje korišćenjem grafa tokova u tehnikama testiranja baziranim na kodu. Graf tokova je usmereni graf od kojeg čvorovi i lukovi odgovaraju programskim elementima. Na primer, čvorovi mogu predstavljati naredbe ili neometane sekvene naredbi, i lukovi transfer kontrole između čvorova.

TEHNIKE BAZIRANE NA DEFEKTIMA

Posebno su namenjene za otkrivanje kategorija verovatno ili unapred definisanih defekata.

Sa različitim nivoima formalizacije, ove tehnike testiranja smišljaju slučajeve za testiranja, posebno namenjene za otkrivanje kategorija verovatno ili unapred definisanih defekata.

Pogađanje grešaka (engl. **Error guessing**): U pogađanju grešaka, slučajevi za testiranje su posebno dizajnirani od strane softverskih inženjera koji su pokušavali da otkriju najprihvativije greške datog programa. Dobar izvor informacija je istorija grešaka otkrivenih u ranijim projektima, kao i inženjerska ekspertiza.

Testiranje mutacija (engl. **Mutation testing**): Ovo je blago izmenjena verzija programa pod testom, razlikujući se malom, sintaksičkom promenom. Ukoliko je slučaj testiranja uspešan u utvrđivanju razlike između programa i mutanta, drugi se „ubija“. Originalno je osnovana kao tehnika evaluacije skupa testiranja.

Mutaciono testiranje je samo po sebi kriterijum testiranja: ili su testovi slučajno generisani dok se dovoljan broj mutacija ne ubije, ili su testovi posebno dizajnirani da ubiju preživele mutacije. U drugom slučaju, testiranje mutacija se može kategorisati kao tehnika bazirana na kodu. Osnovna pretpostavka testiranja mutacija, efekat združivanja, je ta da se traženjem jednostavnih sintaksičkih grešaka nalaze kompleksniji ali realni defekti. Kako bi tehnika bila efektivna, na sistematičan način se mora automatski proizvesti veliki broj mutanta.

TEHNIKE BAZIRANE NA UPOTREBI I PRIRODI APLIKACIJA

Tu spadaju Operativni profil i Testiranje projektovano na pouzdanosti softvera

U tehnike bazirane na upotrebi spadaju:

Operativni profil (engl. **Operational profile**): U testiranju procene pouzdanosti okruženja testa, mora se reprodukovati operativno okruženje softvera što je bliže moguće. Ideja je da se iz posmatranih rezultata testa zaključuje buduća pouzdanost softvera kada je u pravoj upotrebi. Kako bi se to uradilo, ulazima se dodeljuje raspodela verovatnoće, ili profil, prema njihovom pojavljivanju u stvarnim operacijama.

Testiranje projektovano na pouzdanosti softvera (engl. **Software Reliability Engineered Testing**): Testiranje projektovano na pouzdanosti softvera (SRET) je metoda testiranja koja obuhvata čitav proces razvoja, gde je testiranje „dizajnirano i vođeno ciljevima pouzdanosti i očekivanim relativnim korišćenjem i kritičnostima različitih funkcija u oblasti“.

Tehnike bazirane na prirodi aplikacije se odnose na sve tipove softvera. Ipak, za neke vrste aplikacija, potreban je dodatni know-how za izvođenje testa. Lista nekoliko specijalizovanih oblasti testiranja data ovde, bazirana na prirodi aplikacije je :

- *Testiranje bazirano na objektu* (eng.*Object-oriented testing*) *Testiranje bazirano na komponentama* (eng.*Component-based testing*)
- *Web-bazirano testiranje* (eng.*Web-based testing*)
- *GUI testiranje*
- *Testiranje konkurentnih programa*
- *Testiranje usaglašenosti protokola*
- *Testiranje sistema realnog vremena*
- *Testiranje bezbednosno-kritičnih sistema (IEEE1228-94)*

SELEKCIJA I KOMBINOVANJE TEHNIKA

Slučajevi testa se mogu izabrati na deterministički način, prema raznim tehnikama koje su nabrojene, ili slučajno iz nekih raspodela ulaza, kao što se obično radi u testiranju pouzdanosti.

Funkcionalne i strukturne: Tehnike testiranja bazirane na specifikaciji i kodu često su kontrastne kao funkcionalno nasuprot strukturnom testiranju. Ova dva pristupa odabiru testa ne treba posmatrati kao alternativne već komplementarne; zapravo, one koriste različite izvore informacija i dokazano je da daju akcenat na različite vrste problema.

Mogu se koristiti u kombinaciji, u zavisnosti od budžeta.

Determinističke nasuprot slučajnim: Slučajevi testa se mogu izabrati na deterministički način, prema raznim tehnikama koje su nabrojene, ili slučajno iz nekih raspodela ulaza, kao što se obično radi u testiranju pouzdanosti. Nekoliko analitičkih i empirijskih poređenja je sprovedeno kako bi se analizirali uslovi koji čine da je jedan pristup efektivniji od drugog.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 4

Specifikacije i ocene testiranja

ŠEMA POVEZANOSTI TESTOVA I ZAHTEVA

Plan testiranja opisuje opštu podelu testiranja na pojedinačne testove koji se odnose na konkretnе stavke.

Plan testiranja opisuje opštu podelu testiranja na pojedinačne testove koji se odnose na konkretnе stavke. Na primer, ako je u sistemu koji testiramo obrada raspoređena na nekoliko računara, funkcionalni testovi i testovi performanse mogu se dalje podeliti na testove za svaki podsistem.

Za svaki pojedinačni test pravi se posebna specifikacija i procena. U specifikaciji prvo navodimo one zahteve čiju realizaciju test treba da prikaže. Ovaj odeljak se poziva na dokumente zahteva i objašnjava svrhu testa.

Jedan od načina da se uspostavi veza između zahteva i testova jeste tabela ili šema kao na Slici 4. 1 . Obratite pažnju na to da zahtevi koji su navedeni s leve strane tabele, sadrže svoje brojčane oznake; svaki zahtev zahtevana broj testa, koji su označeni kao kolone tabele, u čijoj koloni se nalazi slovo X.

U tabeli su nabrojani zahtevi sistema koje će se testirati testovima od 01 do 06. Testovi performanse mogu se opisati na sličan način. Umesto da se navedu funkcionalni zahtevi, u tabeli se navode zahtevi u pogledu brzine pristupa, bezbednosti baze podataka itd.

		Test Case Number					
		01	02	03	04	05	06
Requirement	1	X					
	2	X				X	
	3	X	X	X	X	X	X
	4	X	X	X	X	X	X
	5	X				X	
	6	X					
	7	X	X	X	X	X	X
	8						
	9	X	X	X	X	X	X
	10	X				X	
	11	X	X	X	X	X	X
	12						
	13	X	X	X	X	X	X

Slika 4.1 Šema povezanosti testova i zahteva [Izvor: NM SE321 - 2020/2021.]

ŠTA KADA SE TEST SASTOJI OD KOLEKCIJE MANJIH TESTOVA?

U tom slučaju specifikacija testa pokazuje vezu manjih testova sa zahtevom.

Često se test sastoji od kolekcije manjih testova koji svi zajedno dokazuju ispunjavanje zahteva. U tom slučaju specifikacija testa pokazuje vezu manjih testova sa zahtevom.

Za svaki test postoji određena filozofija i skup prihvaćenih metoda. Međutim, filozofija i metodi su možda ograničeni drugim zahtevima i realnošću situacije u kojoj se vrši testiranje. U specifikaciji se navode i ti uslovi testiranja. Uslovi mogu biti neki od sledećih:

- *Da li sistem koristi stvarne ulazne podatke od korisnika ili uređaja, ili posebne slučajeve proizvodi neki program ili surogat uređaja?*
- *Koji su kriterijumi za pokrivenost testiranjem?*
- *Kako će se podaci beležiti?*
- *Da li za testiranje postoje ograničenja u smislu vremena, interfejsa, opreme, osoblja, baze podataka ili nečeg drugog?*

Ako se test sastoji od niza manjih testova, kojim redom ih treba izvršiti?

Ako podatke testiranja treba pre ocenjivanja obraditi, specifikacija testiranja opisuje tu obradu. Na primer, kada sistem proizvodi velike količine podataka, ponekad se koriste tehnike za svođenje podataka taka da rezultat bude praktičniji za procenu.

Uz svaki test se navodi na koji način će se proceniti da li je on kompletan. Zato se na kraju svake specifikacije opisuje kako ćemo znati da je test završen i da su ispunjeni relevantni zahtevi. Na primer, plan navodi raspon izlaznih rezultata koji zadovoljavaju zahteve.

Nakon kriterijuma završetka daje se metod za procenu. Na primer, podaci dobijeni tokom testiranja možda će se sakupiti i ručno urediti da bi ih zatim pregledao tim koji obavlja testiranje. Alternative mogu biti da tim koristi automatizovani alat za procenu nekih podataka i pregleda zbirne izveštaje ili da stavku po stavku poređi sa očekivanim rezultatima. Za procenjivanje potpunosti testiranja mogu se upotrebiti mere efikasnosti i delotvornosti opisane u preporuci 1 (data u daljem tekstu).

OPIS TESTA

Opis testa se piše za svaki test definisan u specifikaciji testa.

Dokument opisa testa koristimo za usmeravanje testiranja.

Opis testa se piše za svaki test definisan u specifikaciji testa. Dokument opisa testa koristimo za usmeravanje testiranja. Ovi dokumenti moraju biti detaljni i jasni i moraju da sadrže:

- **sredstva kontrole;**
- **podatke;**

- procedure.

Dokument započinje opštim opisom testa. Zatim navodimo da li će se testiranje pokretati i kontrolisati automatski ili ručno. Na primer, podaci mogu ručno da se unose sa tastature, ali će zatim automatizovani upravljački program izvršavati funkcije koje testiramo. Druga alternativa je da ceo postupak bude automatizovan.

Podaci testiranja mogu da se svrstaju u nekoliko grupa:

1. ulazni podaci,
2. ulazne komande,
3. ulazna stanja,
4. izlazni podaci,
5. izlazna stanja i
6. poruke dobijene od sistema.

Svaka od njih se detaljno opisuje. Na primer, daju se ulazne komande da bi tim znao kako da pokrene test, kako da ga zaustavi ili suspenduje, kako da ponovi ili nastavi test koji nije bio uspeo ili nije bio potpun i kako da prekine test.

Slično tome, tim mora da protumači poruke da bi utvrdio status sistema i kontrolisao testiranje. Ovde opisujemo kako će tim da utvrdi da li je otkaz nastao zbog ulaznih podataka, zbog neispravnog postupka testiranja ili zbog neispravnog rada hardvera (kad god je to moguće).

PREPORUKA 1: MERENJE EFIKASNOSTI I DELOTVORNOSTI TESTA

Delotvornost testa meri tako što se podeli broj grešaka pronađenih u datom testu sa ukupnim brojem pronađenih grešaka

Jedan aspekt planiranja i izveštavanja o testiranju, jeste merenje delotvornosti testa. Graharn (1996b) predlaže da se delotvornost testa meri tako što se podeli broj grešaka pronađenih u datom testu sa ukupnim brojem pronađenih grešaka (uključujući i greške pronađene tokom tog testa).

Na primer, uzmimo da je integracionim testom pronađeno 56 grešaka, a da je u sveukupnom procesu testiranja pronađeno 70 grešaka. U tom slučaju prema Grahamovoj meri delotvornosti značilo bi da je integracioni test bio 80% delotvoran. Međutim, uzmimo da je sistem isporučen nakon što je pronađeno 70 grešaka, a da je tokom prvih 6 meseci rada pronađeno još 70 grešaka. Tada bi integracioni test bio zaslužan za pronalaženje 56 od ukupno 140 grešaka, pa bi test bio delotvoran samo sa 40%. Ovaj način ocenjivanja učinka konkretnе faze ili tehnike testiranja može da se doteruje na više načina.

Na primer, može se svakom otkazu dodeliti stepen ozbiljnosti, pa se delotvornost testova zatim računa prema stepenu. Na ovaj način bi integraciono testiranje bilo delotvorno sa 50% u pronalaženju grešaka koje dovode do kritičnih otkaza, a 80% delotvorno u pronalaženju grešaka koje dovode do manjih otkaza.

Druga alternativa bi bila da se delotvornost testova kombinuje sa analizom uzroka, pa bismo mogli da opisujemo delotvornost u pronalaženju grešaka što je ranije moguće u toku razvoja. Na primer, integraciono testiranje bi moglo da pronađe 80% grešaka, ali pola tih grešaka je moglo biti pronađeno ranije, na primer tokom pregleda dizajna, zbog toga što je bilo reči o problemima u dizajnu.

Efikasnost testiranja se računa tako što se broj grešaka pronađenih tokom testiranja dele sa radom potrebnim da se izvrši testiranje, pa se dobija vrednost izražena u broju grešaka po čovek/ satu. Mere efikasnosti nam pomažu da izračunamo troškove pronalaženja grešaka, kao i relativne troškove pronalaženja grešaka u različitim fazama procesa testiranja. Mere delotvornosti i efikasnosti mogu da budu korisne za planiranje testiranja; želja nam je da na osnovu protekle istorije testiranja postignemo što veću delotvornost i efikasnost.

Prema tome, **dokumentacija u vezi sa trenutnim testiranjem trebalo bi da sadrži merenja koja će nam omogućiti da izračunamo delotvornost i efikasnost.**

PRIMER: SKRIPT TEST ZA TESTIRANJE RUTINE SORT

Procedura testiranja se ponekad naziva skript testa pošto nam daje uputstvo kako da korak po korak izvršimo testiranje.

Na primer, podaci o testiranju rutine SORT mogli bi da budu sledeći:

INPUT DATA:

Ulazni podaci će se dobijati od programa LIST. Program generiše slučajan spisak od N reči od alfanumeričkih znakova; svaka reč je dužine M. Program se poziva sa RUN LIST(N,M)

u pokretaču testiranja. Izlaz se smešta u globalno područje podataka -v-LISTBUF. Skupovi podataka za testiranje za ovaj test su sledeći:

Slučaj 1: Upotrebiti LIST sa parametrima N=5, M=5

Slučaj 2: Upotrebiti LIST sa parametrima N=10, M=5

Slučaj 3: Upotrebiti LIST sa parametrima N=15, M=5

Slučaj 4: Upotrebiti LIST sa parametrima N=50, M=10

Slučaj 5: Upotrebiti LIST sa parametrima N=100, M=1C

Slučaj 6: Upotrebiti LIST sa parametrima N=150, M=10

INPUT COMMANDS:

Rutina SORT se poziva komandom

RUN SORT (INBUF,OUTBUF) ili

RUN SORT (INBUF)

OUTPUT DATA:

Ako se upotrebe dva parametra, sortirani spisak se smešta u OUTBUF. Inače će se vratiti u INBUF.

SYSTEM MESSAGES:

Dok traje sortiranje, prikazuje se sledeća poruka:

"Sorting...please wait..."

Po završetku, SORT prikazuje na ekranu sledeću poruku:

"Sorting completed"

Da bi se testiranje prekinulo pre završne poruke, pritisnuti CONTROL-C na tastaturi.

Procedura testiranja se ponekad naziva **skript testa** pošto nam daje uputstvo kako da korak po korak izvršimo testiranje. Strogo definisan skup koraka obezbeđuje kontrolu nad testiranjem, tako da možemo da ponovimo uslove i ponovo dovedemo do otkaza, ako se to pokaže potrebnim dok budemo pokušavali da otkrijemo uzrok problema. Ako se test prekine iz nekog razloga, moramo da budemo u stanju da ga nastavimo, a da se ne vratimo na početak.

PRIMER: SKRIPT ZA TESTIRANJE FUNKCIJE „PROMENA POLJA“

Opis testa objašnjava redosled aktivnosti potrebnih za završetak testa. Te aktivnosti mogu da se sastoje od čitanja ili štampanja kritičnih podataka, prekidanja auto-matizovanih procedura

Na primer, deo skripta za testiranje funkcije „promena polja“ mogao bi da izgleda ovako:

- Korak N: Funkcijski taster 4: Pristup datoteci podataka.
- Korak N+1: Na ekranu se javlja zahtev da se upiše ime datoteke podataka.
 - Type "sys:test.txt"
- Korak N+2: Meni se pojavljuje i glasi: (1) brisati datoteku; (2)menjati datoteku; (3) preimenovati datoteku
 - Postaviti kurSOR pored "menjati datoteku" i pritisnuti taster RETURN.
- Korak N+3: Na ekranu se javlja zahtev da se upiše broj zapisa. Upisati '4017' .
- Korak N+4: Ekran će se ispuniti poljima podataka iz zapisa 4017:
 - Broj zapisa: 4017 X: 0042 Y: 0036
 - Tip zemljišta: glina
 - Filtriranje: 4
 - Visina zastora:
 - Vegetacija: kudzu
 - Konstrukcija: spoljna
 - Šifra održavanja: 3T/4F/9R

....I tako dalje se opisuju naredni

Koraci u skriptu za test su numerisani, a podaci u vezi sa svakim korakom su referencirani. Ako opisi nisu na drugom mestu, ovde opisujemo kako treba pripremiti podatke i lokaciju za test. Na primer, mogu se dati detalji za potrebna podešavanje opreme, definicije baze podataka i za komunikacione veze.

Zatim, skript tačno opisuje šta treba da se dogodi za vreme testiranja. Koji taster treba da se pritisne, šta se pojavljuje na ekranu, kakav izlaz se dobija, reagovanje opreme i sve ostale manifestacije. Objasnjavamo očekivani efekat ili izlaz i dajemo operatoru ili korisniku uputstva šta treba da radi ako stvarni rezultat nije jednak očekivanom.

Konačno, opis testa objašnjava redosled aktivnosti potrebnih za završetak testa. Te aktivnosti mogu da se sastoje od čitanja ili štampanja kritičnih podataka, prekidanja automatizovanih procedura ili isključivanja delova opreme.

▼ Poglavlje 5

Izveštaj o testiranju

IZVEŠTAJ SA ANALIZOM REZULTATA TESTIRANJA

Izveštaj o analizi testa mogu da čitaju ljudi koji nisu uključeni u postupak testiranja, ali su na druge načine upoznati sa sistemom i njegovim razvojem.

Kada se test izvrši, analiziramo rezultat da bismo utvrdili da li funkcije ili performanse koje smo testirali ispunjavaju zahteve. Ponekad je dovoljno samo prikazati funkciju. Međutim, najčešće za funkciju postoje ograničenja performanse.

Na primer, nije dovoljno utvrditi da neka kolona može da se sortira ili da se sabere. Moramo da izmerimo brzinu izračunavanja i da obratimo pažnju na ispravnost. Zbog toga je izveštaj o analizi testa potreban iz nekoliko razloga:

- On dokumentuje rezultate testa.
- Ako dođe do otkaza, izveštaj sadrži informacije potrebne da se ponovi otkaz (ako je to potrebno) da bi se pronašao i ispravio uzrok problema.
- On sadrži neophodne informacije za utvrđivanje da li je projekat razvoja završen.
- On obezbeđuje poverenje u performansu sistema.

Izveštaj o analizi testa mogu da čitaju ljudi koji nisu uključeni u postupak testiranja, ali su na druge načine upoznati sa sistemom i njegovim razvojem. Zbog toga *izveštaj sadrži kratak opšti pregled projekta, njegove ciljeve i relevantne reference za ovaj test*. Na primer, izveštaj o testiranju sadrži one delove iz zahteva, dizajna i implementacione dokumentacije u kojima se opisuju funkcije koje će se izvršiti u ovom testiranju. Izveštaj takođe ukazuje na delove plana testiranja i specifikacione dokumentacije u kojima se pominje ovaj test.

Kada se teren ovako pripremi, u izveštaju o analizi testa navode se funkcije i karakteristike performanse koje je trebalo prikazati i opisuju stvarni rezultati. Prikazuju se rezultati merenja funkcija, performanse i podataka, sa napomenama da li su ostvareni ciljni zahtevi. Ako se otkrije greška ili nedostatak, u izveštaju se opisuje njegov uticaj. Ponekad rezultate testa procenjujemo merom ozbiljnosti. Ova mera pomaže testerskom timu da odluči da li će nastaviti testiranje ili će sačekati dok se greška ne otkloni. Na primer, ako se poremećaj sastoji od nepravilnog znaka u gornjem delu ekrana, testiranje može da se nastavi dok se uzrok traži i ispravlja. Međutim, ako greška dovodi do toga da sistem padne ili da se datoteka izbriše, tim bi mogao da odluči da se testiranje prekine dok se greška ne otkloni.

OBRAZAC ZA IZVEŠTAVANJE O PROBLEMIMA I OBRAZAC ZA IZVEŠTAVANJE O GREŠCI

U obrazac za izveštavanje o problemima beležimo podatke o greškama i otkazima; Obrazac za izveštaj o grešci objašnjava kako je greška pronađena i ispravljena,

Obrazac za izveštavanje o problemima: Setiće se da smo ranije rekli da je greška takav problem u nekom sistemskom proizvodu koji može da dovede do otkaza sistema; grešku vidi učesnik u razvoju, a korisnik će doživeti otkaz. Tokom testiranja beležimo podatke o greškama i otkazima u obrazac za izveštavanje o problemima. **Obrazac za izveštavanje o odstupanju** je izveštaj o problemu u kojem se opisuju problemi u vezi sa ponašanjem ili atributima sistema koji se razlikuju od očekivanih. Opisuje se šta se očekivalo, šta se umesto toga dogodilo i okolnosti koje su dovele do otkaza.

Obrazac za izveštaj o grešci objašnjava kako je greška pronađena i ispravljena, posle podnošenja obrasca za izveštavanje o odstupanju. Svaki obrazac za izveštavanje o problemu bi trebalo da sadrži odgovore na nekoliko pitanja u vezi sa problemom o kojem je reč:

- **Lokacija:** Gde je došlo do problema?
- **Vreme:** Kada je nastupio problem?
- **Simptom:** Šta smo primetili?
- **Konačni rezultat:** Kakve su bile posledice?
- **Mehanizam:** Kako je došlo do problema?
- **Uzrok:** Zašto je došlo do problema?
- **Ozbiljnost:** Koliko je problem uticao na korisnika ili na poslovanje?
- **Cena:** Koliko je to koštalo?

Na slici 5.1 imamo primer pravog obrasca za izveštavanje o grešci iz jednog engleskog preduzeća. Obratite pažnju na to da svaka greška ima broj i da učesnici u razvoju evidentiraju datum kada su obavešteni da je došlo do problema, kao i datum kada je pronađen i otklonjen uzrok problema.

Pošto učesnici u razvoju ne rešavaju odmah svaki problem (na primer, u slučaju da drugi problemi imaju veći prioritet) oni takođe evidentiraju stvarni broj časova koji je bio potreban da se popravi ova konkretna greška.

Broj greške	Nedelja Prijave	Područje sistema	Vrsta greške	Nedelja rešenja	Trajanje popravke
...
F254	92/14	C2	P	92/17	5.5

Slika 5.1 Obrazac za izveštaj o greškama [Izvor: NM SE321 - 2020/2021.]

OBRAZAC ZA IZVEŠTAJ O GREŠKAMA

Uglavnom, obrazac za izveštavanje o grešci trebalo bi da sadrži sva pitanja i određenu vrstu detaljnih informacija.

Međutim, obrazac za izveštavanje o greškama bi trebalo da sadrži još mnoge druge podatke. Uglavnom, obrazac za izveštavanje o grešci trebalo bi da sadrži sva pitanja i sledeću vrstu detaljnih informacija (Fenton i Pfleeger 1997):

1. *Lokacija:* Identifikator unutar sistema, kao što je ime modula ili dokumenta.
2. *Vreme:* Faza razvoja u kojoj je greška napravljena, otkrivena i ispravljena.
3. *Simptom:* Vrsta poruke o grešci ili aktivnost koja je dovela do otkrivanja greške (na primer, testiranje, pregled, inspekcija).
4. *Konačni rezultat:* Otkazi koje je greška izazvala.
5. *Mehanizant:* Kako je izvor greške stvoren, otkriven i ispravljen.
6. *Uzrok:* Vrsta ljudskog propusta koji je doveo do greške.
7. *Ozbiljnost:* Odnosi se na ozbiljnost otkaza do kojeg je došlo ili do kojeg je moglo da dođe.
8. *Cena:* Vreme ili rad potrebni za pronalaženje i ispravljanje; može da obuhvati i analizu troškova koji bi nastali da je greška otkrivena u ranijim fazama razvoja.

Primetićete da svaki od ovih aspekata greške reflektuje shvatanje učesnika u razvoju o uticaju ove greške na ceo sistem.

S druge strane, izveštaj o odstupanju bi trebalo da reflektuje korisnikov doživljaj otkaza do kojeg je greška dovela. Pitanja su ista, ali odgovori se veoma razlikuju:

1. *Lokacija:* Instalacija na kojoj je primećen otkaz.
2. *Vreme:* CPU vreme, vreme po satu ili neka druga relevantna mera vremena.
3. *Simptom:* Vrsta poruke o grešci ili manifestacija otkaza.
4. *Konačni rezultat:* Opis otkaza, kao što je „pad operativnog sistema“, „pad kvaliteta usluge“, „gubitak podataka“, „pogrešan izlaz“ ili „nema izlaza“.
5. *Mehanizam:* Lanac događaja, uključujući komande sa tastature i podatke o stanju neposredno pre otkaza.
6. *Uzrok:* Referenca na moguće greške koje su dovele do otkaza.
7. *Ozbiljnost:* Uticaj na korisnika ili na poslovanje.
8. *Cena:* Cena ispravljanja plus cena propuštene potencijalne poslovne dobiti.

PRIMER OBRASCA ZA IZVEŠTAJ O GREŠKAMA

U obrascima za izveštaje o problemima potrebne su nam potpunije informacije da bismo mogli da procenimo delotvornost i efikasnost testiranja i razvojne prakse.

Na slici 5.2 nalazi se pravi obrazac za izveštaj o odstupanju, pogrešno nazvan izveštaj o „grešci“. Pomenuta su mnoga pitanja sa našeg spiska i otkazi su dosta dobro opisani.

Međutim, navode se samo stavke koje su promenjene, ali ne i opis stvarnog uzroka otkaza. Idealno bi bilo da se u ovom izveštaju pozove i na neki izveštaj o greškama da bi se moglo znati koje greške su dovele do kojih otkaza.

U obrascima za izveštaje o problemima potrebne su nam potpunije informacije da bismo mogli da procenimo delotvornost i efikasnost testiranja i razvojne prakse. Pogotovo kada imamo ograničene resurse, istorijske informacije zabeležene u izveštajima o problemima pomažu nam da shvatimo u kojim aktivnostima lako dolazi do grešaka i koji su postupci korisni za njihovo pronalaženje i ispravljanje.

IZVEŠTAJ O GREŠCI		S.P0204.6.10.3016		
IZVESTILAC:	Joe Bloggs			
KRATAK NAZIV:	Izuzetak 1 u 620. redu dps_cc.c koji pokreće NAS			
PUNI OPIS	Pokrenuto ispitivanje izdržljivosti NAS, radio je nekoliko minuta. Onemogućili aktivni NAS link (emulator prebačen na pauzu linka), zatim ponovo omogućili onemogućeni link nakon čega je došlo do navedenog izuzetka. (Ja mislim da je ponovo omogućavanje samo prividan razlog.) (tokom punjenja baze podataka)			
DODELJENO NA REŠAVANJE:	DATUM:			
KATEGORIZACIJA:	0 ① 2 3 Design Spec Docn			
POSLATI KOPIJE UZ OBAVEŠTENJE ZA:				
PROCENITELJ:	DATUM: 8/7/92			
ID KONFIGURACIJE	DODELJENO	DEO		
dpo_s.c				
KOMENTARI: dpo_s.c izgleda pokušava da koristi nedozvoljen CID, umesto da odbaci poruku. AWJ				
PROMENJENE STAVKE				
ID KONFIGURACIJE	IMPLEMENTATOR/DATUM	PREGLEDAČ/DATUM	BROJ IZDANJA	INTEGRATOR/DATUM
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92
KOMENTARI:				
ZATVORENO				
KONTROLOR GREŠAKA:	DATUM: 9/7/92			

Slika 5.2 Izveštaj o odstupanjima iz razvoja jednog sistema za kontrolu saobraćaja (Pfleeger i Hatton 1997) [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 6

Pokazna vežba -Testiranje ATM uređaja

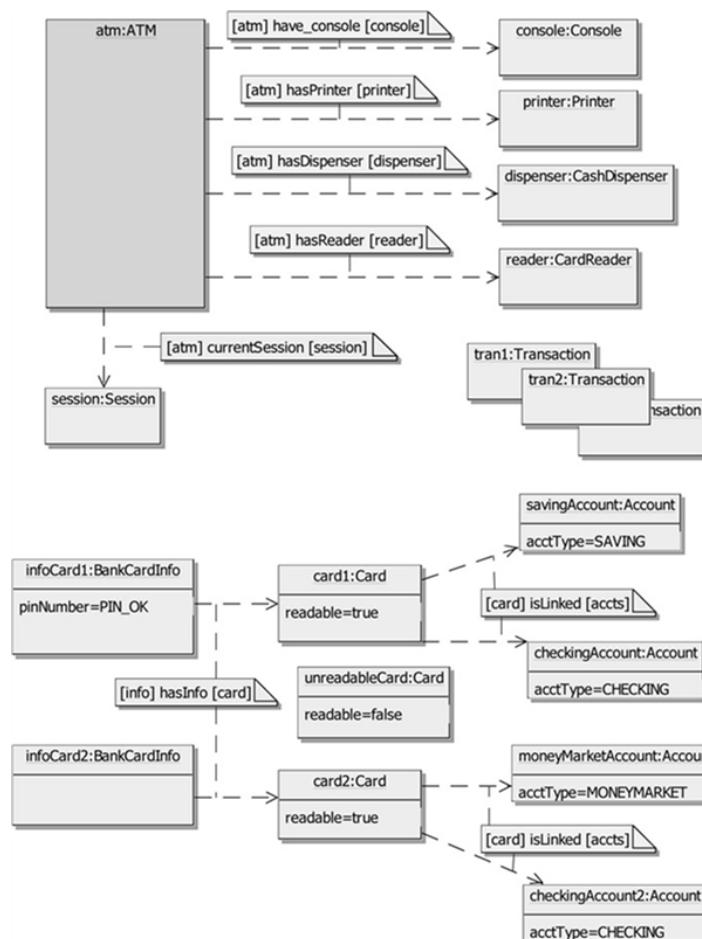
POKAZNA VEŽBA:TESTIRANJE SOFTVERA - PRIMER ATM UREĐAJA

Dijagram objekata ATM uređaja

Na slici 6.1 je dat dijagram objekata ATM uređaja koji želimo da testiramo. Dijagram objekata je dizajniran tako da pruži sve potrebne podatke za testiranje kao kombinacije mogućih ponašanja u dinamičnom modelu. Pri tom ćemo se pridržavati sledećih ograničenja:

1. Postoji jedan objekat sa svim komponentama: konzola kupca, štampač, dispenser i čitač kartica
2. Postoji samo jedna sesija zato što se svi objekti javljaju u toku jedne sesije
3. Posmatramo tri instance kartice: dve koje su čitljive li jedna nije.
4. Tri instance transakcije u dijagramu, što omogućava izvršenje tri transakcije u okviru jedne sesije

Vreme izrade grupne vežbe 45 minuta.



Slika 6.1.1 Dijagram objekata ATM uređaja [Izvor: NM SE321 - 2020/2021.]

GENERISANJE TEST SLUČAJEVA - PRIMER ATM UREĐAJ

Svaka putanja kroz mašinu stanja će biti testirana odvojeno ili nezavisno sa prethodnim startovanjem i gašenjem mašine.

Svaka putanja kroz mašinu stanja će biti testirana odvojeno ili nezavisno sa prethodnim startovanjem i gašenjem mašine.

Odlučili smo da testiramo i setCash parameter na max, zatim custEnteredAmount na min(20\$). Sa ovakvom konfiguracijom testova, generisan je 31 test slučaj od strane LTG koji pokriva 50 tranzicija stanja mašine.

U tabeli na slici 6.2 su prikazani neki test slučajevi.

No.	Test Case
	<i>Testing the ATM operator facilities</i>
T1	switchOn(); setCash(100); switchOff()
	<i>Testing card insertion and reading</i>
T2	switchOn(); setCash(100); custInsertCard(card1); custCancel(); switchOff()
T3	switchOn(), setCash(100); custInsertCard(unreadableCard); switchOff()
	<i>Testing initial PIN entry</i>
T4	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_KO); custCancel(); switchOff()
	<i>Testing PIN reentry</i>
T5	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_KO); custSelectTrans(WITHDRAWAL); custSelectAcct(CHECKING); custEnterAmount(20); custEnterPin(PIN_OK); custAnotherTrans(false); switchOff()
T6	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_KO); custSelectTrans(TRANSFER); custSelectAcct(CHECKING); custSelectAcct(SAVING); custEnterAmount(20); custEnterPin(PIN_OK); custAnotherTrans(false); switchOff()
T7	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_KO); custSelectTrans(WITHDRAWAL); custSelectAcct(CHECKING); custEnterAmount(20); custEnterPin(PIN_KO); custEnterPin(PIN_OK); custCancel(); custAnotherTrans(false); switchOff()
	<i>Testing card retention</i>
T8	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_KO); custSelectTrans(WITHDRAWAL); custSelectAcct(CHECKING); custEnterAmount(20); custEnterPin(PIN_KO); custEnterPin(PIN_KO); switchOff()
	<i>Testing transaction chaining</i>
T9	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_OK); custSelectTrans(TRANSFER); custSelectAcct(CHECKING); custSelectAcct(SAVING); custEnterAmount(20); custAnotherTrans(true); custCancel(); switchOff()
T10	switchOn(); setCash(100); custInsertCard(card1); custEnterPin(PIN_OK); custSelectTrans(WITHDRAWAL); custSelectAcct(CHECKING); custEnterAmount(20); custCancel(); custAnotherTrans(true); custCancel(); switchOff()

Slika 6.1.2 Tabela test slučajeva [Izvor: NM SE321 - 2020/2021.]

IZVRŠAVANJE TEST SLUČAJEVA - PRIMER ATM UREĐAJ

Svaki tim vršiće testiranje svojih modula i svog rada uporedo i po završetku rada.

Pre početka testiranja, moramo dati niz napomena vezanih za samu simulaciju ATM test rešenja.

1. Da bi pokrenuli ATM moramo pritisnuti ON dugme
2. Unosimo broj novčanica koji želimo da imamo u dispenser. Mada je ovo posao operatera, zarad simulacije ovo je neophodan korak.
3. Sesija:
 - 1. Pritisni "Click to insert card" dugme za simulaciju insertovanja kartice
 - 2. Unesi broj kartice(detalji će biti spomenuti u nastavku) I pritisni ENTER
 - 3. Unesi PIN asociran na karticu (vidi uputstvo dole). Iako može biti korišćena tastatura, relevantnije je koristeći simuliranu tastaturu.
 - 4. Izvrši bilo koje transakcije nad već kreiranim računima. Nakon završetka transakcije videće se simulacija izbacivanja kartice.
4. Ugasi ATM uređaj klikom na OFF dugme(Ne možemo ugasiti ATM ako je usred izvršenja)

Brojevi kartica koji su relevantni za simulaciju su:

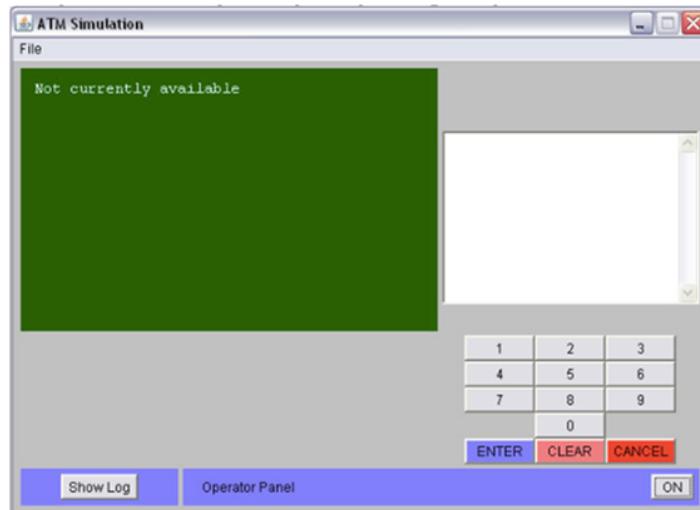
1 2

PIN Dostupni računi:

42 Checking account #1, Savings account #2 1234 Checking account #1, Money market account #3

Sledeći niz screenshot-ova daje detaljan uvid u uspešno izvršenje akcije unosa PIN-a kao i zaključak test inžinjera da je ovaj deo operacije ATM validino odraden.

1. korak - pritisni ON

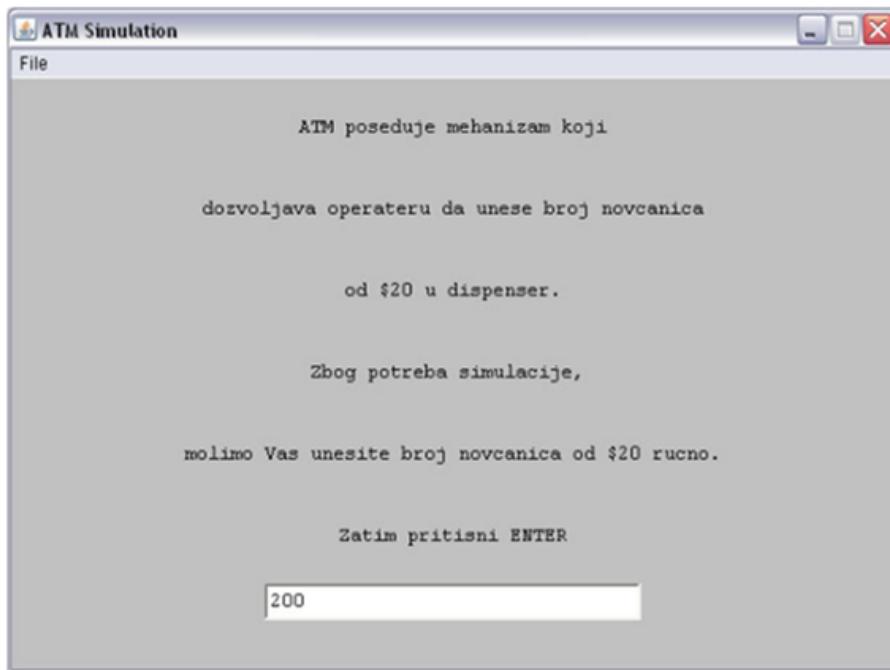


Slika 6.1.3 Izvršenje prvog test slučaja - Pritisakanje ON dugmeta [Izvor: NM SE321 - 2020/2021.]

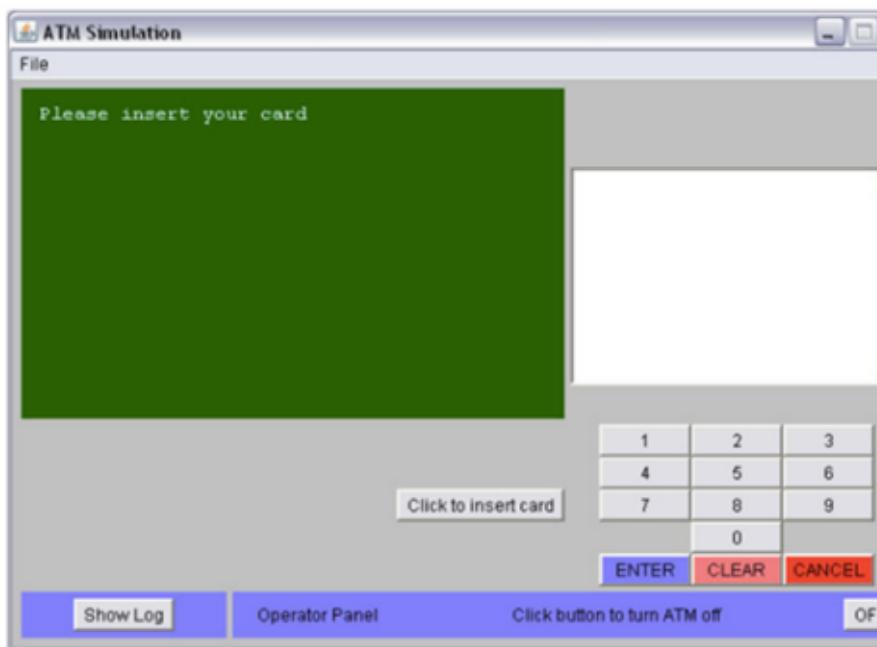
FAZA TESTIRANJA - PRIMER ATM UREĐAJA

Tokom četvrte faze razvoja projekta vrši se stres testiranje celokupnog sistema i finalno testiranje infrastrukture.

Prikaz testiranih ekrana softvera:



Slika 6.1.4 Izvršenje prvog test slučaja - Setovanje 200 novčanica po 20 dolara [Izvor: NM SE321 - 2020/2021.]



Slika 6.1.5 Izvršenje prvog test slučaja - Insert za unos kartice [Izvor: NM SE321 - 2020/2021.]

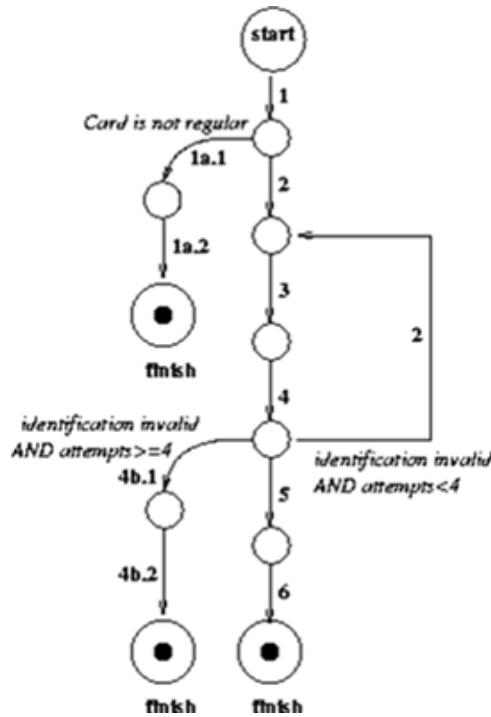
REZULTATI TESTIRANJA - PRIMER ATM UREĐAJA

Na kraju, kao rekapitulaciju testiranja ATM, dajemo uvid u graf scenarija i tabelu inicijalnig slučajeva.

- Naslov: Logovanje korisnika
 - Actors: User
 - Uslov: System je u stanju ON
1. User insertuje Card
 2. Sistem pita za Personal Identification Number (PIN)
 3. User kuca PIN
 4. System validira USER identifikaciju
 5. System prikazuje pozdravnu poruku
 6. System izbacuje karticu

Alternative:

- 1a: kartica nije regularna
 - 1a1: Sistem emituje alarm
 - 1a2: System izbacuje karticu
- 4a: Identifikacija je nekorektna i broj pokušaja < 4
 - 4a.1 Vrati se na korak 2
- 4b: Identifikacija korisnika nekorektna i broj pokušaja ≥ 4
 - 4b.1: Sistem emituje alarm
 - 4b.2: System izbacuje kartice



Slika 6.1.6 Unos PIN koda u ATM uređaj [Izvor: NM SE321 - 2020/2021.]

Proces kroz ATM uređaj kada korisnik unese svoj PIN kod na kraju transakcije. Postoje putevi kada kartica nije regularna, kada verifikacija nije prošla i kada je broj pokušaja prekoračen. Tada nastupaju alternativni putevi do finiša i podizanja novca sa ATM uređaja.

PREDSTAVLJANJE DOBIJENIH REZULTATA

Da bi se generisali test slučajevi, potrebno je testirati sve implicitne i eksplicitne tranzicije prikazane u mašini stanja i prikaz dobijenih rezultata

U nastavku su prikazane tabele sa rezultatima testiranja dobijenim kroz test fazu.

Slučaj korišćenja	Funkcija koja se testira	Inicijalno stanje sistema	Ulaz	Očekivani izlaz
Startovanje sistema	Sistem je startovan kada je prekidač u stanju ON	Sistem je u stanju OFF	Aktiviranje prekidača ON	Sistem traži inicijalnu količinu novca
Startovanje sistema	Sistem prihvata inicijalnu količinu novca	Sistem u stanju potražnje	Unos validne količine novca	Sistem je u stanju ON
Startovanje sistema	Sistem uspostavlja konekciju sa bankom	Sistem je u stanju ON	Ivršavanje legitimne probne transakcije	Izlaz mora da pokaže da je konekcija sa bankom uspešno uspostavljena
Gašenje sistema	Sistem je ugašen kada je u stanju OFF	Sistem nije na raspolaganje korisniku	Aktiviranje prekidač OFF	Sistem je u stanju OFF
Gašenje sistema	Sistem raskida konekciju sa bankom	Sistem se ugasio		Verifikacija od strane banke da konekcija ne postoji
Korišćenje sistema	Sistem čita karticu	Sistem nije na raspolaganju korisniku	Unos validne kartice	Kartica se prihvata
Korišćenje sistema	Sistem odbija nevalidnu karticu	Sistem nije na raspolaganju korisniku	Unos nevalidne kartice	Kartica je odbijena
Korišćenje sistema	Sistem prihvata PIN	Sistem traži PIN	Unos PIN-a	Sistem prikazuje MENI

Slika 6.1.7 Testiranje na osnovu dijagrama slučajeva korišćenja - prvi deo [Izvor: NM SE321 - 2020/2021.]

Slučaj korišćenja	Funkcija koja se testira	Inicijalno stanje sistema	Ulaz	Očekivani izlaz
Korišćenje sistema	Sistem dozoljava korisniku da izvrši transakciju	Sistem prikazuje MENI	Izvršavanje transakcije	Sistem pita za izvršenje druge transakcije
Korišćenje sistema	Sistem dozvoljava izvršenje više seriskih transakcija	Sistem pita kupca da li želi da izvrši drugu transakciju?	Odgovor DA	Sistem prikazuje MENI
Korišćenje sistema	Korišćenje sistema se završava	Sistem pita kupca da li želi da izvrši drugu transakciju?	Odgovor NE	Sistem izbacuje karticu
Izvršenje transakcija	Biće testirani Individualni tipovi transakcija			
Isplata novca	Sistem pita za račun sa kojeg se vrši isplata	Prikazan je MENI	Odabir transakcije ISPLATA	Sistem prikazuje meni sa računima
Isplata novca	Sistem pita da li je isplata u dolarima?	Prikazan je MENI sa računima	Bira se provera računa	Sistem prikazuje MENI sa mogućim isplatama sa računa
Isplata novca	Sistem korektno izvršava isplatu	Sistem prikazuje MENI za isplatu	Bira se iznos za isplatu	Sistem isplaćuje novac, daje izveštaj i ažurira stanje i log file-ove
Isplata novca	Sistem potvrđuje da ima dovoljno novca za isplatu	Sistem je strtovan sa manjom količinom novca	Odabir veće količine od maksimalne	Sistem prikazuje poruku sa greškom

Slika 6.1.8 Testiranje na osnovu dijagrama slučajeva korišćenja - drugi deo [Izvor: NM SE321 - 2020/2021.]

PLAN KVALITETA - PRIMER ATM UREĐAJA

Tim za osiguravanje kvaliteta (QA) će biti zadužen za testiranje softverskog rešenja ATM. Biće određeni članovi tima, zavisno od broja kombinacija test-case-ova koji će biti obavljeni.

Kada se odredi broj tih kombinacija i oformi tim za testiranje, pristupa se samom činu testiranja i, nakon toga sumiraju rezultati testiranja na osnovu kojeg se odredi neki od planova za dalje delovanje na projektu tj. da li je potrebno izvršiti dodatna doprogramiranja ili je ustanovljeno da je testiranje uspešno završeno.

Svaki funkcionalni tim koji učestvuje u implementaciji izvršavaće testiranje unutar granica njegovih funkcionalnosti. Svaki tester biće odgovoran za mali segment poslovnog procesa. Pri prelasku iz jednog u drugo funkcionalno područje odgovornost prelazi na testera zaduženog za taj određeni segment celokupnog poslovnog procesa. Ovaj odeljak će prikazati ciljeve i metode koje će biti korišćene da bi se obezbedio kvalitet razvijanog proizvoda. Poseban tim za osiguravanje kvaliteta (Quality Assurance) će biti formiran i njegov cilj je da obezbedi određen prag kvaliteta ispod kojeg se sistem smatra neprihvatljivim.

Odgovornosti menadžera kvaliteta

Menadžer kvaliteta će biti zadužen za nadgledanje i redovno izveštavanje o rezultatima svih testiranja, što modularnih, što integralnih. Njegova dužnost će takođe biti da redovno

kommunicira sa razvojnim timovima koji imaju problema sa kvalitetom da bi potpomogao njihovom razrešavanju. Dužnost razvojnih timova je da prihvate preporuke menadžera kvaliteta i ulože napor za popravak nedostatka ili greške.

Odgovornosti QA tima

Dužnost QA tima biće upravljanje integralnim testiranjem sistema. Nakon razvoja i podešavanja sistema, tim za osiguravanje kvaliteta će uz pomoć ključnih korisnika i konsultanata formirati detaljne, ispočetka manje kompleksne, a zatim i sve kompleksnije, realne scenarije za testiranje sistema. Nakon toga, ti scenariji će biti izvršeni (na instanci za testiranje sistema) i rezultati će biti detaljno analizirani od strane konsulanata, ključnih korisnika (a i nadzornog odbora projekta u specifičnim slučajevima). Sve nepravilnosti, neispravnosti ili nepredviđena ponašanja sistema će biti dokumentovani i pristupiće se njihovom otklanjanju.

CILJEVI UPRAVLJANJA KVALITETOM - PRIMER ATM UREĐAJ

Cilj je da se postigne najveći mogući kvalitet modula i da se izbegnu problemi tokom integrisanja sistema i integralnog testiranja.

Mali broj neispravnosti u modulima

Pronalaskom i otklanjanjem problema u modulima sugerisaće se da su nedostaci pronađeni pri integralnom testiranju posledica integracije sistema

Mali broj neispravnosti u sistemima

Cilj je da se postigne najveći mogući kvalitet sistema jednom kada su svi moduli integrисани u celinu. Sve neispravnosti nije moguće otkriti u realnim slučajevima pa se zbog toga sprovodi nadgledanje i održavanje sistema tokom njegovog rada.

Adekvatno čuvanje podataka

Analiza podataka je onoliko dobra koliko i podaci i ovaj cilj potpomaže da se svi oštećeni podaci adekvatno sačuvaju.

Adekvatno raspoređivanje vremena

Cilj osigurava da se adekvatan vremenski period uloži u testiranje i analizu rada modula i sistema.

Otklanjanje problema

Menadžer kvaliteta će isprva pokušati da reši sve modularne probleme sa timom koji je radio na tom modulu. Ukoliko tim ne uspe da razreši problem angažuju se dodatni ljudski resursi, konsultanti i projektanti iz drugih timova. O svemu mora biti obavešten menadžer projekta.

✓ 6.1 Vežba za samostalni rad

TESTIRANJE NEKE FUNKCIJE ISUM SISTEMA

Na osnovu obrađenih tehnika testiranja softvera uraditi testiranje upotrebljivosti ISUM sistema.

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

1. Pod pretpostavkom da ste član Tima za osiguravanje kvaliteta (QA), napravite plan obezbeđenja kvaliteta za ovaj deo ISUM sistema. (**vreme izrade 20 minuta**)
2. Napraviti detaljan plan testiranja i opisati funkcije koje će biti testirane (**vreme izrade 20 minuta**)
3. Prikazati proces otklanjanje uočenih problema na osnovu plana testiranja i ciljeva kvaliteta aplikacije (**vreme izrade 20 minuta**)
4. Odaberite neku od tehnika testiranja koje su opisane u predavanjima, dizajnjirajte test slučajeve i opišite ih (**vreme izrade 30 minuta**)

✓ Poglavlje 7

Domaći zadatak

TREĆI DOMAĆI ZADATAK - VРЕME IZRДЕ 180 MIN.

Nakon treće lekcije potrebno je uraditi treći domaći zadatak

Za odabranu aplikaciju u okviru prethodnog, drugog domaćeg zadatka (SE321-DZ02), primeniti sledeće:

1. Napraviti tabelu povezanosti zahtevi-testovi
2. Napravite primere testova
3. Sastavite izveštaj o testiranju korišćenjem obrasca za izveštaj o grešci

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ03-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

nikola.petrovic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ03.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Testiranje softvera - ciljevi i pristupi

TESTIRANJE SOFTVERA

Testiranje se izvodi u svim fazama životnog ciklusa softvera i mogu se testirati gotova programska rešenja ili samo pojedine komponente.

Testiranje softvera predstavlja važnu komponentu razvoja aplikacija. Nedostatak strukturiranog i definisanog procesa testiranja može da dovede do visokih troškova ispravljanja koda, probijanja budžeta i nezadovoljnih klijenata. Takođe, sistematično testiranje pomaže u smanjenju rizika i povećanju stepena kontrole.

Testiranje softvera je umetnost. Većina metoda testiranja nisu mnogo drugačije od pre dvadeset godina. Te tehnike nisu još zrele iako postoji mnogo dostupnih alata i tehnika za korišćenje. Kvalitetan proces testiranja takođe zahteva kreativnost, iskustvo i intuiciju, uz odgovarajuće tehnike.

Testiranje softvera je više nego samo otklanjanje grešaka. Testiranje se ne koristi samo da pronađe defekte i ispravi ih. Takođe se koristi u procesu validacije i verifikacije, i u merenju pouzdanosti. Testiranje se izvodi u svim fazama životnog ciklusa softvera i mogu se testirati gotova programska rešenja ili samo pojedine komponente.

Testiranje je skupo. Automatizacija je dobar način da se smanje troškovi i vreme. Efikasnost i efektivnost testiranja predstavlja kriterijum za tehnike testiranja bazirane na pokrivenosti koda.

Kompletno testiranje je neizvodljivo. Kompleksnost je koren problema. U nekom trenutku proces testiranja se mora zaustaviti i proizvod mora biti isporučen. Vreme i budžet odlučuju kad će se proces testiranja završiti, ili ukoliko procena pouzdanosti softvera zadovoljava zahteve.

Testiranje softvera možda nije najefikasniji metod poboljšanja kvaliteta softvera i neke alternativne metode su možda čak i bolje. Nije moguće dokazati da je softver bez grešaka ali se broj tih grešaka može svesti na minimum.

LITERATURA ZA LEKCIJU 03

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.net/> ili <http://itebooks.info/book/>)

2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. dr Miladin Stefanović, mr Slobodan Mitrović, mr Milan Erić, MODEL KVALITETA SOFTVERA, Festival kvaliteta 2006., 33. Nacionalna konferencija o kvalitetu, Kragujevac, 10. ' 12. maj 2006.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Mr Boris Todorović, EVALUACIJA KVALITETA SOFTVERA PREMA ISO 9126 STANDARDU, DOI 10.7251/POS1208099T.

Veb lokacije:

1. 1. <http://www.qsm.com/>
2. 2. <https://www.computersciencezone.org/software-quality-assurance/>
3. 3. www.mojauto.rs



SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Testiranje softvera – strategije,
tipovi (klase) tehnika testiranja

Lekcija 04

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEĐENJE KVALITETA, TESTIRANJE I EVOLUCIJA SOFTVERA

Lekcija 04

TESTIRANJE SOFTVERA – STRATEGIJE, TIPOVI (KLASE) TEHNIKA TESTIRANJA

- ✓ Testiranje softvera – strategije, tipovi (klase) tehnika testiranja
- ✓ Poglavlje 1: Tehnike detekcije grešaka
- ✓ Poglavlje 2: Tehnike analize
- ✓ Poglavlje 3: Detekcija grešaka na bazi specifikacije
- ✓ Poglavlje 4: Detekcija grešaka na bazi implementacije rešenja
- ✓ Poglavlje 5: Izbor test primera (Oracle problem)
- ✓ Poglavlje 6: Pokazna vežba - Testiranje softvera
- ✓ Poglavlje 7: Domaći zadatak
- ✓ Strategije i tehnike TS

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Testiranje softvera je ipak najviše primenjeno u praksi kao način verifikacije da softver ima svojstva koja su specifikacijom zahtevana.

Cilj ove lekcije je da se naglasi da svaka od tehnika testiranja softvera, koje su publikovane u literaturi ili se koriste u praksi, zahteva poznavanje karakteristika softverskog proizvoda, a to poznavanje softvera zahteva primenu neke od metoda analize.

Pošto jedna tehnika testiranja softvera može biti zasnovana na više sprovedenih metoda analize, isto tako jedna metoda analize se upotrebljava u više tehnika testiranja softvera, zato su tehnike analize diskutovane posebno, pa tek tehnike testiranja softvera koje te metode analize koriste. Jedna od metoda analize koja se može koristiti u fazi izrade projektnih zahteva i specifikacije, dizajna i implementacije rešenja softverskog proizvoda je metoda dokazivanja korektnosti kroz ocenu funkcionalnosti i korektnosti softvera u sistemu koji podržava.

Testiranje softvera je ipak najviše primenjeno u praksi kao način verifikacije da softver ima svojstva koja su specifikacijom zahtevana. Testiranje je dinamički način verifikacije softvera kojim se on izvršava sa unapred pripremljenim test slučajevima (primerima) čiji se očekivani odziv (ponašanje) unapred zna. Stoga, tehnike testiranja softvera uključuju metode analize da bi se odredile karakteristike softvera koje se testiraju nekom od poznatih tehnika, odnosno ocenjuje se da li softver poseduje te karakteristike ili ne.

Testiranje softvera (ručno, automatski ili kombinovano) u većini slučajeva je komplikovan proces. Najčešće se automatski izvode priprema i izvršavanje programa za test slučajeve, memorisanje odziva i verifikacija rezultata testova. U fazi projektovanja testova, mi identifikujemo svojstva softvera koja su predmet verifikacije. Za svaki sprovedeni test potrebno je da znamo očekivani rezultat kako bismo doneli odluku o uspešnosti ili neuspešnosti testa.

✓ Poglavlje 1

Tehnike detekcije grešaka

TEHNIKA DETEKCIJE GREŠAKA

Opisane su tri klase tehnika detekcije grešaka: na bazi specifikacije, na bazi implementacije rešenja i na bazi poznatih klasa grešaka u softveru

Ovde su opisane tri klase tehnika detekcije grešaka :

1. **tehnike detekcije grešaka na bazi specifikacije**
2. **tehnike detekcije grešaka na bazi implementacije rešenja i**
3. **tehnike detekcije grešaka na bazi poznatih klasa grešaka u softveru, a koje su deo skupa tehnika detekcije grešaka integrisane u generičko rešenje Integralnog i Optimiziranog Procesa testiranja Softvera (IOPTS).**

Tehnike detekcije grešaka na bazi specifikacije obezbeđuju da se aktivnostima testiranja softvera pokriju glavne karakteristike iz specifikacije pokrivene proverom.

Tehnike detekcije grešaka na bazi implementacije rešenja softvera koji se testira obezbeđuju pokrivanje glavnih karakteristika programskog koda.

Tehnike detekcije grešaka na bazi poznatih klasa grešaka u softveru obezbeđuju pokrivanje tipičnih grešaka u softveru.

Dobit od korišćenja raznih klasa tehnika detekcije grešaka je u komplementarnosti, jer nijedna tehnika detekcije nije dovoljna sama po sebi.

Ocena kvaliteta tehnika analize softvera i tehnika detekcije grešaka može biti na osnovu teorije ili prakse. Ovde ćemo koristiti obe forme ocene kvaliteta tih tehnika na osnovu kriterijuma za njihovo adaptibilno korišćenje zbog iskorišćenja jakih a ne slabih strana i kontrole procesa testiranja softvera. Upravljanje procesom testiranja softvera primenom više tehnika analize i detekcije grešaka mora biti sistematsko i rigorozno.

To se postiže u dva koraka:

1. **Prvo se mora izvršiti izbor tehnika koje više odgovaraju u konkretnom projektu.**
2. **Zatim se odabrane tehnike moraju sistematski primeniti.**

KAKO OCENITI KVALITET SOFTVERA?

Kvalitet softvera se ne može direktno ocenjivati prema specifikaciji zahteva, već je potrebno odrediti karakteristike softvera kroz proces analize kao važnog koraka u procesu testiranja SW.

Proces testiranja softvera ima smisla ako je napravljena specifikacija zahteva u vidu iskaza, dijagrama, objašnjenja i drugih izvora informacija koje bliže definišu performanse i ograničenja softvera. Ovde se ne pravi razlika u specifikaciji koja se odnosi na softversku jedinicu, već i na projektno rešenje visokog ili niskog nivoa apstrakcije, bilo da je specifikacija u pisanoj ili nekoj drugoj formi važno je da služi kao izvor informacija o karakteristikama softvera. Pored specifikacije, predmet procesa testiranja softvera je i implementacija rešenja tj. programski kod.

Zahtevi za kvalitetom softvera treba da su korektno specificirani, zbog čega je i sama specifikacija projektnih zahteva predmet testiranja, pa tek onda se pristupa testiranju implementacije softverskog rešenja sa aspekta ocene kompletnosti i korektnosti realizovanih karakteristika softvera. U praksi je dokazano da su greške moguće i u specifikacijama i implementaciji rešenja softvera.

Nažalost, kvalitet softvera se ne može direktno ocenjivati prema specifikaciji zahteva. Umesto toga, potrebno je odrediti karakteristike softvera kroz proces analize kao važnog koraka u procesu testiranja softvera.

Karakteristike softvera se odnose na performanse, faktore (atribute) kvaliteta, ili osobine koje su ili nisu zahtevane. U softverske karakteristike se može smatrati i broj operatora u izvornom programskom kodu kao i odziv softvera pri njegovom izvršavanju za odabранe test slučajeve. Prva karakteristika se određuje statickom analizom (bez izvršavanja programa), a druga karakteristika se obično određuje dinamičkom analizom (nekom od tehnika koje uključuju izvršavanje programa).

ADEKVATNO TESTIRANJE SOFTVERA

Uspostavljanjem kriterijuma adekvatnosti procesa testiranja softvera, cilj je da se poboljša kvalitet procesa testiranja softvera a time i kvalitet softvera.

Kao što smo u prethodnom odeljku naveli, nivo poverenja, koji zavisi od veličine uzorka (broja testova) u oceni kvaliteta softvera na bazi informacija dobijenih testiranjem softvera, se obezbeđuje postavljanjem kriterijuma adekvatnosti na osnovu kojih se vrši izbor test slučajeva.

Na primer, testiranje softvera se može smatrati neadekvatnim ako skup test slučajeva ne uključuje i granične vrednosti ulaznog domena koje su specificirane u zahtevima za kvalitetom softvera, ako se ne izvrši svaka instrukcija programskog koda ili nije predviđeno ponašanje softvera u situacijama otkaza pojedinih delova softvera u pojedinim situacijama.

Uspostavljanjem kriterijuma adekvatnosti procesa testiranja softvera, cilj je da se poboljša kvalitet procesa testiranja softvera a time i kvalitet softvera.

Pri tome ne treba mešati kriterijume za poboljšanje kvaliteta procesa testiranja softvera sa kvalitetom softverskog proizvoda koji se testira.

Konfuzija oko nerazumevanja ova dva pojma se vidi kada se kvalitet procesa testiranja softvera ocenjuje preko karakteristika softvera dobijenih procesom testiranja softvera tj. preko broja otkrivenih grešaka u softveru, srednjeg vremena između otkaza misije softvera i slično. To je očigledno i u slučaju korektnog tj. visoko kvalitetnog softvera koji je slabo testiran. *Kriterijum adekvatnosti igra dvostruku ulogu i kao specificirani zahtev i kao sudija:* kao specifikacija koja treba da ukaže na ograničenja koja moraju biti zadovoljena procesom testiranja softvera, i kao sudija koji ukazuje na slabosti (nedostatke) pojedinih tehnika detekcije grešaka.

Pošto izbor kriterijuma adekvatnosti i kompletnosti procesa testiranja softvera nije jednoznačan, nijedna tehnika detekcije grešaka nije superiorna da eliminiše upotrebu ostalih tehnika. Zbog tog **tehnike detekcije grešaka treba shvatati kao komplementarna sredstva verifikacije i validacije softvera koji se testira, a ne kao konkurente, jer razne tehnike detekcije grešaka kad se iskombinuju imaju najveće šanse da otkriju veliki broj potencijalnih grešaka.**

▼ Poglavlje 2

Tehnike analize

ŠTA SU TEHNIKE ANALIZE?

Svaka tehnika koja ima za cilj da utvrди karakteristike softvera je jedan vid analize.

Svaka tehnika koja ima za cilj da utvrđi karakteristike softvera je jedan vid analize.

Određivanje karakteristika softvera u toku njegovog razvoja, verifikacije i validacije tj. testiranja, otklanjanja uočenih problema i grešaka, izrade dokumentacije i održavanja, je od suštinskog značaja.

Ovde će biti prikazane neke tehnike analize koje su značajne u procesu verifikacije softverskog proizvoda, a posebno u procesu testiranja softvera.

Analitičke metode i postupci se mogu primeniti u svim fazama procesa testiranja softvera, uključujući izbor test slučajeva, izvršavanja testova, beleženje rezultata i ocenu rezultata testova. Izbor testova se zasniva na analizi različitih izvora informacija o softveru koji se testira, uključujući ali ne završavajući se na specifikaciji, implementaciji rešenja, potencijalnim greškama i problemima u softveru.

Prikupljanje informacija o softveru se može obaviti analizom teksta izvornog programskog koda.

Izvršavanje testova i dobijenih rezultata takođe zahteva analizu. Dodatni postupci i tehnike analize su potrebni pri izradi i verifikaciji test Oracle mehanizma.

Ovde su tehnike i postupci analize *svrstani prema tački posmatranja softverskog proizvoda (izvora informacija)*. Svaka tačka gledišta na softver ima svoj aspekt koji obezbeđuje specifične informacije o karakteristikama softvera koji se analizira. Nekoliko glavnih analitičkih gledišta, postupaka i tehnika koje one podržavaju su opisane u tekstu koji sledi. Neke tehnike analize koriste više aspekata softvera. Redosled prikaza tehnika analize je dat prema količini informacija koje pružaju.

Napomena: Oracle test je mehanizam za utvrđivanje da li je test prošao ili nije. Upotreba oracle testa uključuje poređenje izlaza sistema koji se testira, za date ulazni test-slučaj, sa izlazima koje oracle utvrđuje da proizvod treba da ima. Izraz "oracle test" prvi je put uveden u radu Vilijama E. Hovdena.

TEHNIKA ANALIZE: POSMATRANJE TEKSTA IZVORNOG KODA I POSMATRANJE SINTAKSE PROGRAMSKOG JEZIKA

Sa aspekta teksta izvornog koda softvera, program se tretira kao niz znakova ili imenica, dok se s aspekta sintakse, program posmatra kao hijerarhijska struktura sintaksnih elemenata

A. Posmatranje teksta izvornog koda softvera: Sa aspekta teksta izvornog koda softvera, program se tretira kao niz znakova ili imenica. Na osnovu tog aspekta analiza daje osnovnu meru ili karakteristiku programa kao što je dužina programskega koda, frekvencija pojavljivanja neke instrukcije, identifikatora, operatora i slično. Tekst editori manipulišu sa programom kao sa običnim tekstrom, isto kao što rade skeneri, brojači linija i drugi. Uputstva ili preporuke za pisanje programskega koda zasnivaju se na ovom pogledu na softver. Razni grafički uređaji, kao što su laserski štampači, imaju isti pogled na programski kod.

B. Posmatranje sintakse programskega jezika: Program može da se posmatra sa gledišta hijerarhijske strukture sintaksnih elemenata koji čine gramatiku jednog programskega jezika. Programi se mogu razložiti na manje jedinice tj. potprograme koji se pak sastoje od programskega instrukcija itd. dok se ne dođe do nivoa karaktera. Do sintaksnog pogleda se može doći i analizom teksta izvornog koda programa, kao što to čini raščlanjivač (engl. Parser) ili konstruisati direktno, kako to radi sintaksnii editor. Izvedene karakteristike nakon sintakse analize programa mogu biti broj instrukcija u programu,

unakrsna tabela identifikatora, dijagram poziva programa (koja procedura poziva drugu proceduru), deklarisane i nedeklarisane programske promenljive, frekvencija upotrebe vrednosti neke promenljive i druge.

Na osnovu navedenih prostih metrika mogu se iskonstruisati mnogo složenije mere karakteristika programa. Pogled na sintaksu programa koristan je da bi se izvorni ili objektni programski kod modifikovao kako bi se pratila unutrašnja dešavanja pri izvršavanju programa (ovaj postupak se često naziva instrumentalizacija programa). Pri izvršavanju ovako modifikovanog programa, mogu se odrediti razne karakteristike programa koje su korisne sa gledišta testiranja programa, npr. koje i u kom procentu su instrukcije ili grananja pri izvršavanju pokrivena, ili pak čitav otisk (trag) izvršene misije programa, pojedine vrednosti internih promenljivih itd.

TEHNIKA ANALIZE: POSMATRANJE DIJAGRAMA TOKA IZVRŠAVANJA PROGRAMA

Dijagram toka izvršavanja programa odnosi se na prikaz redosleda i uslova izvršavanja programskega elemenata.

C. Posmatranje dijagrama toka izvršavanja programa: Dijagram toka izvršavanja programa odnosi se na prikaz redosleda i uslova izvršavanja programskega elemenata.

Pomenuti programski elementi obično su jedna programska naredba, uslov odluke o nastavku toka izvršavanja programa, blok naredbi koje se u nizu jedna za drugom izvršavaju bez grananja i slično. Ako se elemenat B programa izvršava neposredno posle elementa A, tada (A,B) označava relaciju toka izvršavanja programa. U ovom slučaju B se izvršava bezuslovno tj. naredni elemenat se izvršava bez obzira na vrednost izračunavanja elementa programa A. Ali, ako elemenat A predstavlja uslov $x < x$, a elemenat B naredbu za štampanje promenljive x u liniji koda,

if $x < x$ then write(x),

tada i dalje važi relacija toka izvršavanja programa (A,B) bez obzira što je nejednakost $x < x$ tj. A netačna.

Graf koji se dobije na osnovu prikaza toka kontrole izvršavanja programskih elemenata naziva se dijagram kontrole toka.

Svaki čvor grafa predstavlja programski elemenat, koji se povezuju orijentisanim granama koje ukazuju na redosled izvršavanja programskih elemenata. Putanja na dijagramu kontrole toka prikazuje sekvensijalni redosled izvršavanja programskih elemenata pod određenim uslovima. Ako se može odrediti ulazna vrednost programa za tu putanju dijagrama kontrole toka tj. datu sekvencu redosleda izvršavanja programskih elemenata, tada se kaže da je data putanja prohodna ili realizibilna, u suprotnom je neprohodna. Program sa petljama ima enorman pa čak i beskonačan broj putanja.

Dijagram kontrole toka, za razliku od običnog dijagrama toka nema oznake ili komentare (objašnjenja) koji daju dodatne informacije o programu. Dijagram kontrole toka izvršavanja programa se generalno može efikasno napraviti iz sintaksne analize programskog koda. Iz njega se može izvesti više mera kao što je McCabe mera kompleksnosti programa koja korespondira sa razumljivošću i brojem grešaka programa.

TEHNIKA ANALIZE: POSMATRANJE TOKA PODATAKA

Relacije toka podataka u programu određuju programski elementi na osnovu njihovog pristupa i menjanja istih.

D. Posmatranje toka podataka: Relacije toka podataka u programu određuju programski elementi na osnovu njihovog pristupa i menjanja istih. Ako elemenat B koristi (ili se poziva na) neki objekat podataka koji je potencijalno mogao biti definisan u elementu A, tada (A,B) predstavlja relaciju toka podataka programa.

Dijagram toka podataka je orijentisani i označeni graf koji opisuje relacije toka podataka u kojem čvorovi predstavljaju elemente programa a orijentisane grane koje povezuju čvorove A i B su obeležene imenom promenljive npr. x jer (A,B) predstavlja relaciju toka podataka kada elemenat programa A definiše, a elemenat B koristi promenljivu x. Dijagram toka podataka se može se generisati na osnovu sintakse programa radi efikasnije obrade toka podataka. Program se može predstaviti preko dijagrama toka podataka sa prikazom informacija o definicijama promenljivih, njihovog pozivanja i promenljivih koje nisu definisane. Na osnovu ovakvog predstavljanja programa, informacije o toku podataka mogu se iskoristiti za optimizaciju koda, detektovanje anomalija i generisanja podataka za potrebe testiranja

softvera. Tokovi podataka koji odstupaju od korektnih (anomalije) su tokovi koji zahtevaju dodatna ispitivanja jer mogu da ukažu na probleme i greške u softveru.

Primeri su kada se neka promenljiva dva puta uzastopno definiše bez njenog korišćenja, korišćenje promenljive koja nije prethodno definisana ili

ukidanje promenljive koja posle definisanja nije korišćena. Moguće je generisati programsko parče u kome su eliminisane sve instrukcije u programskom kodu koje ne utiču na izračunavanja u nekom izrazu na određenoj lokaciji programa. Korel je adaptirao tehniku parčanja programa (engl. **Slicing**), za potrebe testiranja i otklanjanja grešaka u programu. Ovaj metod se zasniva na dinamičkim programskim isećcima koji imaju bar dve prednosti:

1. obrada vektorskih i dinamičkih struktura podataka može se preciznije pratiti i
2. programski isečak može se značajno smanjiti tako da se olakšava lokalizacija greške u programu

Ovo se može iskoristiti za realizaciju kriterijuma i tehnike detekcije grešaka pokrivanjem toka podataka u programu. Ove tehnike koriste aktuelne ulazne podatke koji se tokom izvršavanja programa efikasno prate, kao i tok izvršavanja programa. Ako se u procesu izvršavanja programa primeti neočekivan tok tj. da aktualna putanja ne odgovara očekivanoj, kontrolnoj putanji tada se npr. kod praćenja izvršavanja programa za funkciju minimizacije preko algoritma pretraživanja automatski pronalaze promenljive koje su uzrokovale prolaz aktuelne putanje što značajno skraćuje vreme lokalizacije problema u softveru.

TEHNIKA ANALIZE: POSMATRANJE TOKA IZRAČUNAVANJA

Skup izračunavanja je trag promena stanja podataka koje program proizvodi tokom njegovog izvršavanja za određeni ulazni skup podataka.

E. Posmatranje toka izračunavanja: Na program se može gledati kao na konačan skup (potencijalno neograničen) izračunavanja.

Skup izračunavanja je trag promena stanja podataka koje program proizvodi tokom njegovog izvršavanja za određeni ulazni skup podataka. Detaljna analiza toka izračunavanja pri izvršavanju programa može poslužiti za procenu broja zaostalih grešaka u programu, kao mera kvaliteta u otkrivanju grešaka u softveru ili mogućnost postojanja skrivenih grešaka u programu.

Takva tehnika detekcije grešaka je, na primer, ona koja koristi statistički metod ocene broja i prirode zaostalih grešaka u softveru namernim ubacivanjem (sejanjem) grešaka u program koji se testira. Prvo se ubace poznate greške u program, zatim se program testira i broj otkrivenih poznatih grešaka se koristi za procenu broja još neotkrivenih grešaka u programu.

Problem u primeni ove tehnike detekcije grešaka je taj da greške koje se ubacuju moraju biti reprezentativne za one greške u programu koje još nisu otkrivene. *Jedna od najčešće*

korišćenih i opisanih tehnika detekcije grešaka koja se zasniva na metodi sejanja grešaka je **mutaciona analiza**.

Programi koji se dobiju ubacivanjem grešaka nazivaju se **mutantima**. **Programi mutantni izvršavaju se sa ciljem da se proveri da li je njihov rezultat različit od originalnog programa koji se testira. Mutanti koji se ponašaju različito od originala, uništavaju se nakon testa.** Korist od ovakvog načina analize programa je da se oceni (izmeri) sa kojim uspehom odabrani skup test slučajeva uništava mutante.

MUTACIONA ANALIZA

Mutaciona analiza predstavlja jednu od najčešće korišćenih i opisanih tehnika detekcije grešaka koja se zasniva na metodu sejanja grešaka

Mutanti se generišu primenom **mutacionih operatora**. Takav jedan operator, iz konačnog skupa mogućih, menja značenje izraza u originalnom programskom kodu. Na primer, konstanta u programu se može povećati za recimo jedan, umanjiti za jedan, ili biti zamenjena vrednošću nula, čime se bira jedan od tri mutanta. Zatim se primenjuje jedan od datog mutacionog operatora u svakoj tački programa koji se analizira ili testira, gde je to primenljivo iz konačnog skupa mutanata.

Postoje tri potrebna uslova da greška u programu dovede do njegovog otkaza misije, a to su:

1. *da se program izvršava,*
2. *da je inficiran sa greškom i*
3. *da se greška prosledi (propagira) do očekivanog rezultata (izlaza, ponašanja) programa koji se poredi sa tačnim rešenjem koje obezbeđuje **test Oracle**. To znači da posle sejanja (ubacivanja) greške iz poznate klase grešaka u originalni program, ta lokacija programa mora biti izvršena (posećena, trigerovana) tako da se vrednost promenljivih u programu zbog toga inficiraju (promene stanje, vrednost) i izazovu otkaz programa prosleđujući infektivni podatak do završetka programa.*

Morell i drugi autori analizirali su potrebne uslove da se inficiranje i propagacija greške u programu dogodi, jer u tom slučaju mutaciona analiza daje korisne informacije o programu.

Voas i drugi autori razvili su metodologiju analize osetljivosti programa na greške preko tri potrebna uslova da program otkaže, posvećujući posebnu pažnju mehanizmima infekcije i propagacije grešaka. **Analiza infekcije programa** zasniva se na mutacionoj analizi, sa ciljem da se utvrdi verovatnoća inficiranja nekog podatka, stanja programa nakon izvršavanja moguće pogrešne naredbe u programu. **Analiza propagacije** treba da proprati promenu stanja programa, podataka kako bi se ocenila verovatnoća da inficirani podatak, stanje izazove otkaz programa. Takođe, Voas je predložio način ocenjivanja verovatnoće da: izvršavanje, infekcija i propagacija zajedno dovedu do otkaza programa za određene klase otkaza.

Morell koristi model simboličnog izvršavanja (o kome će kasnije biti reči) programa kako bi se analiziralo kretanje greške u programu, tako što je ubacivao simboličke greške u program koji se zatim simbolički i izvršavao. Na taj način je povezao simboličko "hvatanje" grešaka u odzivu programa, očekujući isti efekat i kod stvarnog izvršavanja programa.

TEHNIKA ANALIZE: FUNKCIONALNO POSMATRANJE

Program se posma sa aspekta funkcije kojom se vrši transformaciju ulaznog u izlazni domen preko uređenog para (x,y) gde je y odziv, rezultat koji program proizvodi za ulaznu vrednost x

F. Funkcionalno posmatranje: Program se može posmatrati sa aspekta funkcije programa smatrajući da program vrši transformaciju ulaznog u izlazni domen preko uređenog para (x,y) gde je y odziv, rezultat koji program proizvodi za ulaznu vrednost x nakon zaustavljanja.

Snimajući odziv programa koji se izvršava i zaustavlja za odabrane test slučajeve, ustvari se snima i analizira se odzivna funkcija (misija) programa. Simbolička analiza se koristi za opis funkcije koju program računa, ali u najopštijem obliku.

Sistem koji izvršava simbolički program ima tri ulaza: originalni programske kod koji se analizira, simbolički ulaz za program koji se analizira putanjem koja treba da se u programu izvrši. Kao rezultat daje: simbolički izlaz koji opisuje izračunavanja programa na zadatoj putanji i uslove prolaza kroz tu putanju programa. Specifikacija putanje može da se interaktivno postavlja ili da se unapred zada.

Simbolički rezultat može se iskoristiti kao dokaz da se program ponašao prema specifikaciji, a uslov za prolaz datom programskom putanjom kao generator test slučajeva željene programske putanje.

Jedino strukture podataka mogu predstavljati problem, pošto je ponekad nemoguće izvesti zaključak koja mu je komponenta modifikovana.

▼ Poglavlje 3

Detekcija grešaka na bazi specifikacije

KARAKTERISTIKE TEHNIKA ZA DETEKCIJU GREŠAKA

Tehnike detekcije grešaka spadaju u dinamičke tehnike verifikacije programa i uključuju izbor test slučajeva, izvršavanje programa i analizu rezultata

Tehnike detekcije grešaka spadaju u dinamičke tehnike verifikacije programa. One uključuju sve tri, već istaknute aktivnosti:

1. izbor test slučajeva,
2. izvršavanje programa i na kraju
3. analizu rezultata koji treba da potvrde da li je program korektan ili ne.

Mora se povesti računa da se svi faktori okoline koji imaju uticaja na karakteristiku softvera koji se ocenjuje, uzmu u obzir.

Na primer, svi implicitni ulazni podaci moraju se posmatrati i uzeti u obzir (sistemski takt, stanje fajlova i dr.) isto tako kao i reprezentativni tekući parametri okoline (isti kompjajler, programski punjač, verzija operativnog sistema, servisni paket, hardverska konfiguracija računara, profil ulaznih podataka i dr.) za vreme izvršavanja testova.

Saglasje test okoline sa "realnim" slučajevima u toku izvršavanja testova je posebno važno pri simboličkom izvršavanju testova. Upotreba interpretera programa zahteva posebnu pažnju sa aspekta vernosti ciljne hardverske i programske specifikacije u eksploraciji softvera koji se testira.

Kao što smo prethodno istakli, treba koristiti sve raspoložive izvore informacija pri izboru test slučajeva, od specifikacije, implementacije rešenja softvera, potencijalnih grešaka iz baze podataka od ranijih sličnih projekata, pa do dokumentacije generisane u svakoj aktivnosti procesa razvoja i testiranja softvera ili čak kombinovanjem više navedenih izvora informacija. Tehnike detekcije grešaka koje su opisane u ovom predavanju su reprezentativni primeri korišćenja širokog spektra raspoloživih izvora informacija koje su klasifikovane kao:

1. **Tehnike za detekciju grešaka na bazi specifikacije**
2. **Tehnike za detekciju grešaka na bazi implementacije**

TEHNIKE DETEKCIJE NA BAZI SPECIFIKACIJE

Cilj ovih tehnika je da se potvrdi da su sve zahtevane i specificirane (opisane) karakteristike realizovane na korektan način, uključujući ulazni, izlazni domen i obradu podataka

Tehnike detekcije grešaka koje se zasnivaju na informacijama iz svih dostupnih dokumenata u procesu razvoja softvera, a posebno iz specifikacije i projektne dokumentacije u kojima se definišu ili opisuju karakteristike i ponašanje softvera, se klasificuju kao tehnike testiranje softvera na bazi specifikacije.

U osnovi ovih tehnika detekcije grešaka postavljen je cilj da se potvrdi da su sve zahtevane i specificirane (opisane) karakteristike realizovane na korektan način, uključujući ulazni, izlazni domen, kategorije ili klase ulaznih podataka koje softver obrađuje na isti (ekvivalentan) način u izvršavanju svoje misije.

Kriterijum adekvatnosti testiranja pomoću tehnika detekcija grešaka na bazi specifikacije je najčešće pokrivenost zahtevanih i specificiranih karakteristika softvera. To se postiže proverom softvera preko izvršavanja pojedinih programskih putanja ili delova programa koje su odgovorne za tu karakteristiku softvera.

Testiranje softvera na bazi specifikacije zasniva se na posmatranju i proveri funkcija programa pa se često naziva i funkcionalno testiranje ili testiranje crne kutije.

Deli se na:

1. Tehnike testiranja koje ne zavise od načina izrade specifikacije softvera
2. Tehnike testiranja koje zavise od načina izrade specifikacije softvera

TEHNIKE KOJE NE ZAVISE OD NAČINA IZRADE SPECIFIKACIJE SOFTVERA

U reprezentativne tehnike testiranja softvera na bazi specifikacije spadaju: Tehnike testiranja na osnovu definicije interfejsa i Tehnike testiranja na bazi funkcija programa

Bez obzira koji je metod korišćen u izradi specifikacije softvera koji se testira (prirodni jezik, algebarski jezik, SDL, Z i dr.) u specifikaciji se nalaze važne informacije i predstave o funkciji i misiji softvera.

Specifikacija obuhvata opis interakcije (interfejsa) delova softvera među sobom, sa drugim softverima, sa hardverskom okolinom i naravno korisnikom da bi se neka funkcija ili misija softvera izvršila. Definisanje interfejsa uključuje osobine ulaznih promenljivih programa, izlaznih promenljivih, vrste i opsega njihovih vrednosti što se često jednim imenom nazivaju ulazno izlaznim domenima.

Ponašanje programa obično se opisuje funkcijama izračunavanja (semantikom) kao i karakteristikama toka izvršavanja programa, kao što su potrebe za memorijskim prostorom

ili vremena izvršavanja pojedinih funkcija. U daljem delu teksta opisane su reprezentativne tehnike testiranja softvera na bazi specifikacije gde spadaju:

1. **Tehnike testiranja na osnovu definicije interfejsa**
2. **Tehnike testiranja na bazi funkcija programa**

TEHNIKE TESTIRANJA NA OSNOVU DEFINICIJE INTERFEJSA

Baziraju se na informacijama interakcije softvera, a izbor test slučajeva vrše na osnovu osobina ulaznog ili izlaznog domena programa i njihove međuzavisnosti.

Tehnike testiranja na osnovu definicije interfejsa se baziraju na informacijama interakcije softvera, a izbor test slučajeva vrše na osnovu osobina ulaznog ili izlaznog domena programa i njihove međuzavisnosti.

Testiranje ulaznog domena se može vršiti:

1. **Testiranjem ekstremnih vrednosti**
2. **Metoda ekvivalentnog parcelisanja**

Testiranjem ekstremnih vrednosti vrši se provera ponašanja programa izvršavanjem programa za ekstremne vrednosti ulaznog domena. Takođe se biraju test slučajevi iz sredine (unutrašnjosti) ulaznog domena. Ovakav način biranja test slučajeva motivisan je induktivnom metodom rezonovanja da ako se pokaže da program korektno izračunava funkcije za karakteristične vrednosti iz ulaznog domena, tada će verovatno korektno izračunavati vrednost funkcije i za ostale elemente ulaznog domena. Ako je ulazni domen struktuiran, tada se vrši izbor test slučajeva kombinovanjem ekstremnih vrednosti.

Kod metode ekvivalentnog parcelisanja specifikacije najčešće definišu ponašanje programa za određenu klasu ulaznih vrednosti, tako da program svaku vrednost iz te particije ulaznog domena tretira na isti način.

Ekvivalentno parcelisanje se vrši identifikacijom konačnog broja funkcija programa i njihovih ulazno izlaznih domena. Na primer, ako u specifikaciji стоји relacija $\{(x, y) \mid x \geq 0 \text{ } y = x \& x < 0 \text{ } y = -x\}$

Treba ulazni domen podeliti na dva podskupa koja određuju funkcije identiteta i negacije respektivno. Ovo parcelisanje takođe mora uzimati u obzir i ograničenja koja postoje u pogledu definisanosti funkcija na ulaznom domenu kao i greške izračunavanja funkcija. Kada se identificuju ekvivalentne particije domena, vrši se testiranje programa za ekstremne i unutrašnje vrednosti ekvivalentnih particija ulaznog domena.

Svaki program mora biti otporan (robustan) na nekorekstan format ulaznih podataka tako da prepozna takve slučajeve i da na odgovarajući način reaguje. Testiranje ponašanja programa na ovakve slučajeve se naziva sintaksna provera. Jedan od mogućih načina ovakve provere je da se program izvršava na velikom uzorku ulaznih vrednosti promenljivih. Drugi način je da se

koriste sintaksna pravila npr. BNF gramatika konkretnog programskog jezika da se kontroliše broj izabralih test slučajeva.

TESTIRANJE NA BAZI FUNKCIJA PROGRAMA

U ovu grupu tehnika spadaju: Testiranje programa sa specijalnim vrednostima ulazno/izlaznog domena i Testiranje programa prekrivanjem izlaznog domena

Tehnika testiranja softvera ekvivalentnim parcelisanjem ulaznog domena se fokusira kao što smo videli na konačan skup funkcija programa i njima pridruženih ulazno izlaznih domena. Može se, međutim, izbor test slučajeva izvršiti i na osnovu znanja o karakteristikama datih funkcija.

Posmatrajmo, na primer, funkciju čije vrednosti program izračunava na osnovu konačnog broja tačaka tj. određene vrednosti ulaznog domena se mapiraju pomoću date funkcije. Moguće je testirati korektnost izračunavanja te funkcije u datom programu čak i ako ne postoji kompletna specifikacija o tome. Ipak, poznavanje karakteristika funkcija datog programa je od suštinskog značaja sa aspekta sigurnosti adekvatnog pokrivanja izlaznog domena pri funkcionalnom testiranju programa.

U grupu tehnika testiranja na bazi funkcija programa spadaju:

- 1. **Testiranje programa sa specijalnim vrednostima ulazno/izlaznog domena:** Izbor test slučajeva korišćenjem poznatih osobina funkcije koja se izračunava naziva se testiranje softvera sa specijalnim vrednostima ulazno/izlaznog domena. Nedostatak ovog pristupa testiranja softvera je taj da je njegova primena ograničena uglavnom na matematička izračunavanja i u primenama gde su ona intenzivna.

Korišćenje poznatih osobina funkcija u izboru test slučajeva korisna su radi dokazivanja tačnosti sa kojom se vrše ta matematička izračunavanja. Na primer, ako se vrši izračunavanje sinusne funkcije, tada znanje osobine periodičnosti sinusne funkcije sugerije da test slučajeve treba birati tako da se oni razlikuju za 2π , jer ako su već proverena izračunavanja sinusa za te osnovne vrednosti, istu tačnost i rezultat treba očekivati i za vrednosti kojima je dodat umnožak 2π zbog periodičnosti sinusne funkcije.

- 2. **Testiranje programa prekrivanjem izlaznog domena:** Ako se na izračunavanje neke funkcije u programu primeni metod parcelisanja ulaznog domena na ekvivalentne particije, tada se svakoj particiji može pridružiti odgovarajući izlazni domen. Tada se može vršiti izbor test slučajeva kojim se vrši prekrivanje izlaznog domena karakterističnim vrednostima. Ovaj pristup je koristan, naročito u slučajevima provere korektnosti i tačnosti izračunavanja ekstremnih vrednosti, kao i svih kategorija poruka programa o greškama koje se zahtevaju u tim ekstremnim slučajevima obrade podataka.

TEHNIKE TESTIRANJA KOJE ZAVISE OD NAČINA IZRADE SPECIFIKACIJE SOFTVERA: ALGEBARSKI I AKSIOMATSKI PRIKAZ SPECIFIKACIJE

Dve od ovih tehnika su algebarski prikaz specifikacije i aksiomatski prikaz specifikacije

Tehnika izrade specifikacije softvera takođe može biti korisna u procesu testiranja softvera. Specifikacija pisana na način da može da se izvršava može se koristiti kao test Oracle ili kao generator testova. Strukturne osobine specifikacije se mogu koristiti kao vodič u procesu testiranja softvera. Ukoliko se formalni način prikaza specifikacije sastoji od određenih klasa prezentacije softvera koji se testira, osobine tih klasa modela ili zadataka softvera mogu se iskoristiti za izbor test slučajeva.

U ovu grupu tehnika spadaju sledeće tehnike:

- 1. **Algebarski prikaz specifikacije:** Kod algebarski predstavljenih specifikacija, osobine apstraktnih podataka su izražene u vidu aksioma ili izrađenih pravila. U jednom od sistema u kojem je implementiran algebarski prikaz specifikacije poznat kao DAITS, konzistentnost algebarske specifikacije neke implementacije softvera koji se testira, se proverava pomoću ovog sistema. Svaka aksioma se kompajlira u proceduru kojoj se onda pridružuje skup test slučajeva.

Pokretački (engl. **driver**) program snabdeva podatke za testove za proceduru kojom se proverava konzistentnost primene algebarske aksiome ili izrađenih pravila. Strukturalna pokrivenost, kako implementacije tako i same specifikacije, se izračunava u procentima. Algebarski metod prikaza specifikacije softvera veoma se koristi u objektno-orientisanom programiranju softvera.

- 2. **Aksiomatski prikaz specifikacije:** Uprkos velikim potencijala koji široko rasprostranjen račun predikata pruža, kao jezik za izradu specifikacije softvera, malo je publikovanih podataka izrade test slučajeva na osnovu ovog načina prikaza specifikacije softvera. Neke reference se mogu naći u radu Gourlay-a koji se odnosi na povezanost tehnika detekcije grešaka prekrivanjem putanja programa i računa predikata u specifikaciji softvera.

TEHNIKE TESTIRANJA KOJE ZAVISE OD NAČINA IZRADE SPECIFIKACIJE SOFTVERA: MAŠINSKA STANJA I TABELA ODLUKA TJ. USLOVA/AKCIJE

Tabele odluka je koncizan način predstavljanja metoda ekvivalentnog parcelisanja.

- 3. **Mašinska stanja** (engl. **state machines**): Mnogi programi se mogu prikazati tj. modelovati preko tehnike za prikaz stanja maštine, koji omogućava drugi način izrade

test slučajeva. Pošto je problem ekvivalentnosti dva konačna automata jednoznačno određen, za testiranje programa koji ustvari simulira konačni automat i koji je specificiran preko tehnike prikaza stanja mašine sa konačnim brojem čvorova, stoga se može koristiti tehniku ekvivalentnosti programa i specifikacije. Ovaj podatak se koristi za **testiranje osobina programa koje se mogu izraziti preko specifikacije konačnog broja stanja mašine (automata)** tj. za kontrolu toka klase softvera koji se testira za obradu transakcija.

- 4. **Tabela odluka** tj. uslova/akcije: Tabela odluka je koncizan način predstavljanja metoda ekvivalentnog parcelisanja. Redovi u tabeli predstavljaju uslove koje ulazne vrednosti programa moraju zadovoljiti, a kolone predstavljaju različit skup akcija koje program izvodi pod tim uslovima. Unešene vrednosti u tabelu, ukazuju da li će se određena akcija desiti ako je taj uslov ispunjen. Obično se unose vrednosti "DA", "NE" ili "Nije bitno". Svaki red u tabeli pruža niz informacija o mogućem izboru test slučajeva na osnovu unete vrednosti. Jedna od najčešće sistematski primenjivanih tehnika detekcije softvera koja izbor testova vrši na osnovu prevođenja specifikacije pisane u prirodnom, govornom jeziku u uzročno-posledični graf, je opisana u referencama.

▼ Poglavlje 4

Detekcija grešaka na bazi implementacije rešenja

TESTIRANJE SOFTVERA NA BAZI IMPLEMENTACIJE PROJEKTNOG REŠENJA

Tehnike testiranja na bazi implementacije projektnog rešenja koriste dokumentaciju vezanu za implementaciju rešenja na visokom, niskom nivou, programskom kodu i sličnim dokumentima

Tehnike testiranja softvera na bazi implementacije projektnog rešenja koriste dokumentaciju vezanu za implementaciju rešenja na visokom, na niskom nivou, programskom kodu i sličnim dokumentima za izbor testova. Cilj je da se pokaže da je implementirano rešenje softvera kompletno i u saglasnosti sa opisanim karakteristikama softvera u tim dokumentima. Zastupnici ovog pristupa u testiranju softvera tvrde da je veća verovatnoća otkrivanja grešaka ukoliko se, kao kriterijum izbora test slučajeva, koristi pomenuta projektna dokumentacija.

Svako izvršavanje programa odnosi se na prekrivanje određenog dela programa (putanje, uslov-odluka, definicija-upotreba podataka i sl.). Kriterijumi izbora testova svode se na odgovore na sledeća pitanja:

1. **Koje opisane karakteristike programa se proveravaju?**
2. **Koje programske putanje se izvršavaju iz kojih se vidi da je ta karakteristika programa korektno implementirana?**
3. **Koji su to test slučajevi koji izazivaju izvršavanje tih programske putanja?**
4. **Koje se sve karakteristike programa mogu posmatrati za taj skup test primera?**

Tehnike testiranja softvera nabazi implementacije projektnog rešenja zasnivaju se na činjenici da jedino tekst programa i opis projektnih rešenja može da pruži informaciju o tome kako je programer implementirao softverska rešenja. Iz razloga efikasnosti programeri često donose neke odluke o implementaciji detalja iz projektnog rešenja softvera koje nije precizirano ili nije eksplisitno istaknuto u specifikaciji prema sopstvenom shvatanju i iskustvu. Ovaj nedostatak u programskom kodu može se nadoknaditi jedino testiranjem softvera na bazi specifikacije, dok se merenjem pokrivenosti programskog koda (recimo, procentom izvršenih instrukcija-videti dole) pri struktturnom testiranju ovaj problem može uočiti.

Tehnike detekcije grešaka na bazi implementacije projektnog rešenja mogu se podeliti u odnosu na dva nezavisna pristupa:

1. na bazi grešaka u implementaciji i
2. na osnovu programskog koda koje su već diskutovani u prethodnim sekcijama kad su opisivane tehnike analize programa.

Prvo će biti opisane:

1. Tehnike detekcije na bazi programske strukture a zatim
2. Tehnike detekcije na bazi propagacije grešaka

TESTIRANJE SOFTVERA NA BAZI PROGRAMSKE STRUKTURE

Tehnike detekcije grešaka koje se zasnivaju na informacijama iz strukture programa izbor test slučajeva vrše na osnovu izvršavanja pojedinih struktturnih elemenata programa.

Tehnike detekcije grešaka koje se zasnivaju na informacijama iz strukture programa izbor test slučajeva vršena osnovu izvršavanja pojedinih struktturnih elemenata programa. Programski kod se instrumentalizuje tako da se prati procenat pokrivanja izvršenih struktturnih elemenata programa za izabrani skup testova koji su kriterijum adekvatnosti i kompletnosti testiranja softvera.

Najčešće se kao kriterijum pokrivenosti programske strukture koriste tačke grananja, izračunavanje i promena podataka, što je upravo i predmet razmatranja ovog odeljka.

A. Testiranje programskih naredbi: se zasniva na kriterijumu da svaka naredba programa bude izvršena tokom realizacije testova. Pošto je očigledno da ni 100% prekrivanje programskih naredbi ne garantuje da je program korektan, isto tako je jasno da svaki manji procenat izvršenih programskih naredbi znači da postoji deo programa koji nije proveren, tako da je verovatnoća da greške nisu otkrivene veća.

B. Testiranje grananja programa: Na primeru programske strukture sa grananjem je jasno da ni 100% pokrivenost izvršenih programskih naredbi ne garantuje da su sve grane u dijagramu toka programa proverene. Na primer, izvršavanjem naredbe IF ... THEN (bez ELSE) kada je testiran slučaj da je uslov ispunjen (logička istina, tačno) jasno je da je samo jedna grana programa dijagrama toka proverena. Stoga je kod kriterijuma na bazi pokrivanja grananja u dijagramu toka programa cilj da se osigura provera izvršavanja svake grane dijagrama toka programa. Kontrola prekrivanja grananja u dijagramu toka programa se lako postiže instrumentalizacijom programskog koda umetanjem proba (obično logička promenljiva ili brojač) u programskom elementu svake grane dijagrama toka koja beleži da je ta grana posećena. Ovakva instrumentalizacija za kontrolu prekrivanja grananja u dijagramu toka je u isto vreme dovoljna i za kontrolu izvršenih programskih naredbi.

TESTIRANJE NA BAZI PROPAGACIJE GREŠAKA: TESTIRANJE PROGRAMSKIH PUTANJA

Tehnika detekcije grešaka u softveru na bazi izvršavanja svih programskih putanja treba da obezbedi otkrivanje potencijalnih grešaka u programu.

Tehnika detekcije grešaka na bazi propagacije grešaka ima za cilj izbor test slučajeva koji obezbeđuju da se infekcija prosledi do potencijalne greške u programu izazivajući otkaz (defekt) programa. U suštini se zasniva na izboru programske putanje na bazi propagacionih karakteristika koje mogu da dovedu do otkaza programa.

A. Testiranje programskih putanja: Tehnika detekcije grešaka u softveru na bazi izvršavanja svih programskih putanja treba da obezbedi otkrivanje potencijalnih grešaka u programu. U praksi, naravno, nije uvek moguće izvršiti 100% prekrivanje svih programskih putanja iz više razloga.

- *Prvo*, svaki program sa beskonačnom petljom, ima beskonačno putanja izvršavanja jer za svaki prolaz petljom prolazi se jedna programska putanja sa različitim rezultatom obavljenе misije softvera. To znači da ne postoji konačan skup test slučajeva sa kojim bi se sve programske putanje izvršile i korektnost rezultata testa proverila.
- *Drugi mogući problem* je nerešivost uslova za prolaz kroz neku programsku putanju tj. postojanje u programu neizvršivih putanja. Naime, bez uspeha su pokušaji da se odredi test slučaj kojim se ta putanja izvršava, ali je obavezan napor da se pokaže i otkriju

takve programske putanje jer one ukazuju na potencijalne probleme u softveru.

- *Treće*, nerešiv je u opštem slučaju problem zaustavljanja programa za bilo koji test slučaj, pa je moguće da je ta putanja beskonačno duga tj. misija softvera se ne izvršava u konačnom vremenskom intervalu. Kao odgovor na ove slučajevе, nekoliko pojednostavljenih pristupa je predloženo. Beskonačno veliki broj programskih putanja se parcelisanjem na ekvivalentne klase može svesti na konačan broj na osnovu analize karakteristika petlji.

Problem je takođe i **testiranje petlji na unutrašnju (srednju) i granične vrednosti tj. za vrednost brojača jednaku nuli (ne izvršavanje petlje)**, izvršavanje petlje jedan put, ako je moguće maksimalan broj puta i na kraju, maksimalan broj puta plus jedan. Postoji više tehnika detekcije grešaka koja primenjuju opisani postupak, a među njima je poznata tehnika izvršavanja linearne programske sekvene i kriterijum skoka koja određuje putanju za izvršavanja hijerarhijski, počev od najprostije ka najsloženijoj. Tehnika detekcije grešaka u softveru na bazi izvršavanja svih programskih putanja ne uključuje prekrivanje uslovnih naredbi ili izraza za izračunavanje u programu.

TESTIRANJE NA BAZI PROPAGACIJE GREŠAKA: TESTIRANJE SLIČNO KOMPAJLIRANJU PROGRAMA

Kriterijumi za izbora test slučajeva su slični metodama koje se koriste pri prevođenju odnosno, kompjajliranju programa čime se povećava sigurnost programa u njegovu korektnost.

B. Testiranje programa slično kompjajliranju programa: Hamlet je u radovima predložio kriterijume izbora test slučajeva korišćenjem metoda koje se koriste pri prevođenju, odnosno, kompjajliranju programa čime se povećava sigurnost programa u njegovu korektnost.

Ulagano-izlazni parovi su navedeni kao komentari u programskom kodu, kao parcijalna specifikacija obrade (funkcije) datog programa (procedure). Procedura se onda izvršava za svaku ulaznu vrednost i proverava izlazna uparena vrednost iz komentara u programskom kodu.

Skup testova se smatra adekvatnim ako je svaki logički ili izraz izračunavanja proveren tj. nijedan izraz se ne može zameniti jednostavnijim a da pritom da korektni rezultat testa.

Jednostavniji izraz je onaj koji se može konačnim brojem zamena uprostiti. Tada, za vreme izvršavanja procedure, svaka izvršena zamena izraza se proverava u pogledu dobijenih vrednosti (stanja) podataka iz izraza. One zamene koje ne daju iste vrednosti, kao originalni izraz se odbacuju. Zamene izraza koje dovode do istog rezultata, a da dovode do otkaza programa, takođe se odbacuju kao nekorektna zamena.

TESTIRANJE NA BAZI PROPAGACIJE GREŠAKA: TESTIRANJE TOKA PODATAKA PROGRAMA

Kao kriterijum za izbor test slučajeva koristi se zahtev za pokrivanje svih tačaka u programu u kojima se neka promenljiva definiše i mesta gde se ona koristi.

C. Testiranje toka podataka programa: Analiza toka podataka u softveru može se, takođe, iskoristiti kao tehnika detekcije grešaka koja proverava korektnost relacija definisanja i upotrebe nekog podatka u programu. **Kao kriterijum za izbor test slučajeva koristi se zahtev za pokrivanje svih tačaka u programu u kojima se neka promenljiva definiše i mesta gde se ona koristi.** Parovi programskih tačaka koji se na taj način uspostavljaju se koriste za potrebne uslove infekcije i propagacije potencijalnih grešaka u programskom kodu.

U osnovi je motiv da se neadekvatnim proglose test slučajevi koji ne proveravaju parove definisanje-korišćenje (DEF-KOR) toka podataka u programu.

Primer parova DEF-KOR, je na primer, tok podataka za promenljivu x u tački A programa, upotrebljena u tački B programa bez redefinisanja promenljive x. Tada se par DEF-KOR naziva relacija (A,B). Jasno je da podaci koji nisu korektno definisani u programu ili se nikada ne koriste u programu, neće moći biti provereni ni pri testiranju korektnosti programa za izabrane test slučajeve nekom drugom tehnikom definisanja grešaka.

Tokovi podataka u programu mogu biti statistički ili dinamički određeni . Neki DEF-KOR parovi toka podataka se ne mogu otkriti zbog postojanja neizvrsivih programske putanja. Zbog ovakvih slučajeva, primenjuje se često istraživački (engl. **Heuristics**) metod generisanja test slučajeva.

TESTIRANJE NA BAZI PROPAGACIJE GREŠAKA: TESTIRANJE MUTIRANJEM PROGRAMSKOG KODA

Mutanti se izvršavaju da bi se proverilo da li je njihov rezultat različit od originalnog programa koji se testira. Mutanti koji se ponašaju različito od originala uništavaju se nakon testa.

D. Testiranje mutiranjem programskog koda : Jedna od najčešće korišćenih i opisanih tehnika detekcije grešaka zasniva se na metodu sejanja grešaka na bazi mutacione analize programa. Programi koji se dobiju ubacivanjem grešaka nazivaju se mutantima. **Programi mutanti se izvršavaju sa ciljem da se proveri da li je njihov rezultat različit od originalnog programa koji se testira. Mutanti koji se ponašaju različito od originala uništavaju se nakon testa.** Na ovaj način, može se pokazati da li je izabrani skup test slučajeva neadekvatan, jer se tako demonstrira da određene klase grešaka mogu biti napravljene u softveru a da sa tim skupom testova budu neotkrivene. Testiranje mutiranjem programskog koda zasniva se na dve hipoteze.

▼ Poglavlje 5

Izbor test primera (Oracle problem)

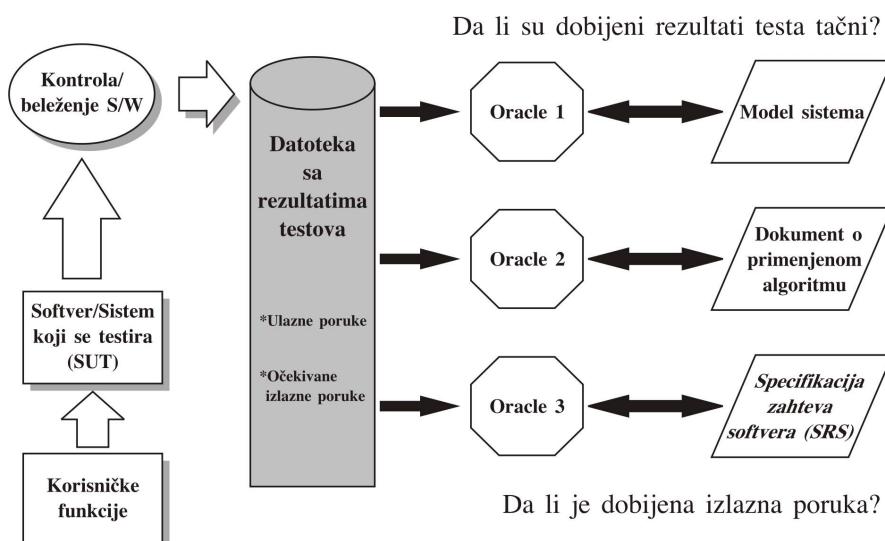
PROBLEM IZBORA TEST SLUČAJEVA I NJIHOVIH REŠENJA

Oracle se mora napraviti, verifikovati i njegov kvalitet (korektnost, pouzdanost, efikasnost, primenjivost, pogodnost za održavanje) validirati u svim fazama procesa testiranja softvera

Kao što smo već istakli, testiranje softvera/sistema je po prirodi eksperimenat sa merenjem odziva softvera/sistema koji se sastoji od:

1. **faze izbora test slučajeva,**
2. **faze izvršavanja testa (eksperimenta) i**
3. **krajnje faze odlučivanja o korektnosti odziva na osnovu poređenja dobijenih rezultata sa korektnim, očekivanim rezultatima testova.**

Poslednja faza odlučivanja obavlja se pomoću opšte prihvaćenog naziva test Oracle mehanizma koji ustvari predstavlja referentni sistem koji se testira ili etalon za proveru merenja kvaliteta softvera. Od referentnog sistema koji se testira tj. test Oracle se očekuje da na osnovu poređenja dobijenih i očekivanih rezultata testova odgovori sa „DA“ ako je sistem koji se testira korektan, tj. test uspešan, sa „NE“ ako sistem koji se testira nije korektant. test neuspešan, ili sa „NEODLUČAN“ kada ne može da se dobije decidan odgovor na osnovu analize izmerenog odziva softvera/sistema ili kada sistem koji se testira ne daje uopšte rezultat (problem nezaustavljivog programa) kao što je prikazano na slici 5. 1.



Slika 5.1 Model Testiranja Softvera na bazi test Oracle mehanizma [Izvor: NM SE321-2020/2021.]

ZAŠTO JE VAŽAN TEST ORACLE MEHANIZAM?

Test Oracle je ključan za obezbeđivanje tj. ugradnju kvaliteta softvera i mora se paralelno sa procesa razvoja sistema rešavati kao važna aktivnost u procesu testiranja softvera.

U literaturi je do sada obrađivan proces testiranja softvera sistematski u fazi izbora i izvršavanja testova dok se, najčešće, projektovanje referentnog sistema za testiranje zanemarivalo, uprkos činjenici da je faza odlučivanja o rezultatima testova kritična u procesu testiranja softvera i da je potreban ozbiljan napor (ukupni resursi) za pronalaženje rešenja testova.

Test Oracle je ključan za obezbeđivanje tj. ugradnju kvaliteta softvera imora se paralelno sa procesom razvoja sistema rešavati kao važna aktivnost u procesu testiranja softvera. Projektovanje referentnog sistema koji se testira može predstavljati značajan napor koji povećava troškove razvoja sistema, ali će krajnji efekat biti pozitivan jer će smanjiti ukupne troškove testiranja i održavanja softvera.

Oracle se mora napraviti, verifikovati i validirati njegov kvalitet(korektnost, pouzdanost, efikasnost, primenljivost, pogodnost za održavanje) u svim fazama procesa testiranja, od testiranja jedinice programskog koda, preko testiranja podsistema (testiranje u fazi integracije sistema) do sistemskog testiranja na sistematičan, disciplinovan i kontrolisan način kao i svaki drugi, pa i originalni sistem.

Pošto Oracle odlučuje o oceni kvaliteta softvera, proces testiranja neće ispuniti svoj glavni zadatak da potvrdi postojanje ili odsustvo grešaka na efikasan način jer ručna provera, niti je pouzdana niti jeftina. *Postavka eksperimenta tj. uslova koji su neophodni da bi se potencijalne greške u sistemu koji se testira manifestovale pri izvršavanju test slučajeva je podložan greškama.*

Pripremi, odnosno postavci eksperimenta ili jednostavno testa se obično, ne posvećuje dovoljna pažnja, jer se misli da test može da se pokrene iz bilo kog stanja sistem koji se testira. Da li je izvršavanje nekog zadatka sistema već pokrenuto? Da li je bankovni račun, kome želimo da pristupimo već otvoren u bazi podataka ili ga moramo otvoriti tj. kreirati u bazi podataka? Da li je trenutni dijalog na ekranu nekorektan? Koja ovlašćenja, privilegije ima trenutni korisnik sistema? Da li svi fajlovi u sistemu postoje, da li su oštećeni i slično? Koji je trenutni direktorijum kome pristupamo?

Ovo su neki od mogućih uslova za manifestaciju grešaka u softveru koje tester mora da poznaje i da kontroliše. Ali, oni su ozbiljan problem pri uvođenju automatskih sredstava za testiranje sistema.

OD ĆEGA ZAVISI POSTAVKA EKSPERIMENTA (USLOVA) DA BI SE MANIFESTOVALE POTENCIJALNE GREŠKE U SISTEMU KOJI SE TESTIRA

Odziv i ponašanje sistema koji se testira ne zavise samo od podataka i signala kojima se sistem stimuliše već zavise i od uslova izvođenja testova.

Projektanti test procedura najviše pažnje posvećuju aktivnostima stimulisanja sistema sa podacima ili signalima za test slučaj koji se izvršava, jer smatraju da je on bitan, a zanemaruju činjenicu da odziv i ponašanje sistema, zavise i od uslova izvođenja testova.

Ponašanje sistema koji se testira, za isti skup ulaznih podataka može biti različito u zavisnosti od okoline sistema koji se testira (okruženja), od verzije operativnog sistema, veličine memorije, procesorske brzine, intenziteta saobraćaja u računarskoj mreži, koji su drugi softveri instalirani u okruženju sistema i slično. Još su teži primeri analize uticaja fizičkih (elektro-mehaničkih, vremenskih, prostornih, klimatskih) parametara okoline sistema koji se testira.

Iz iskustva se zna da se greške u sistemu manifestuju zbog spoljašnje temperature, magnetskog polja, lošeg uzemljenja uređaja ili nekog drugog parametra okoline, a da se u laboratorijskim ispitivanjima one

nisu manifestovale. Iako su greške u sistemu, koje su se manifestovale iz gore navedenih razloga, neuobičajene, ipak se mora povesti računa o uzrocima njihovog manifestovanja pri postavci eksperimenta, posebno pri automatizaciji procesa testiranja sistema

Beleženje i ocenjivanje dobijenih rezultata testiranja sistema je jedan od koraka uspešnog procesa testiranja.

Kod ručnog testiranja, čovek-ekspert je taj koji je odgovoran za donošenje odluke o korektnom ponašanju sistema koji se testira na osnovu posmatranja ponašanja ili zabeleženih odziva koji je prezentovan na čoveku razumljiv način pri izvršavanju testa.

Obično je korak odlučivanja o rezultatima testa, pri automatskom testiranju mnogo složeniji, jer zahteva verifikovan i validiran rezultat testa tj. nepogrešiv test Oracle za svaki test slučaj.

ŠTA SE PODRAZUMEVA POD TERMINOM TEST ORACLE?

To je neki drugi program, softver/sistem ili drugi mehanizam (može se nazvati referentnim sistemom) za generisanje očekivanih rezultata sistema za svaki test slučaj.

U literaturi se pod terminom test Oracle podrazumeva više stvari u procesu testiranja sistema: proces generisanja očekivanih rezultata sistema, sama rešenja testova (očekivani

podaci) ili rezultat odlučivanja o uspešnosti ili neuspešnosti testa. Korektno je da se pod pojmom test Oracle tj. referentnim sistemom podrazumeva neki drugi program, softver/sistem ili drugi mehanizam za generisanje očekivanih rezultata sistema za svaki test slučaj.

Svaka tehnika detekcije grešaka zavisi od kvaliteta referentnog sistema koji se testira. U literaturi iz oblasti testiranja sistema, najviše pažnje se poklanja fazi generisanja test slučajeva ili adekvatnosti skupa testova, ali raspoloživost referentnog sistema tj. test Oracle za svaki test slučaj eksplicitno se ne daje ili se podrazumeva da rešenja testova (Oracle) jednostavno već postoji, mada konkretni opis primjenjenog Oracle se ne daje.

Pošto u procesu razvoja softvera postoje četiri evidentne oblasti softverskog inženjerstva čiji proizvod nije izvestan (određen): nepouzdan je korak analize zahteva i izrada specifikacije softverskog proizvoda, nepouzdan je proces projektovanja od sistemskih zahteva pa do implementacije softverskog rešenja (koda), nepouzdanost u ponovnoj upotrebi projektovanog softvera i nepouzdanost u korišćenju inženjerskih tehniki i metoda.

Proces testiranja softvera, kao paralelan proces koji prati sve navedene oblasti softverskog inženjerstva u kojima postoji nepouzdanost, takođe je nepouzdan, jer je čovek dominantno uključen u sprovođenje procesa testiranja i podleže maksimim neizvesnostima MUSE (engl. **Maxim of Uncertainty in Software Engineering-MUSE**). Posebno ako pogledamo jednu rutinsku i često ponavljaju aktivnost u procesa testiranja, kao što je provera dobijenog rezultata testa koja, ako se ručno obavlja, podložna je greškama, što unosi dodatnu neizvesnost (rizik) u oceni kvaliteta softvera.

VRSTE ORACLE MEHANIZAMA

Oracle koji daje tačan rezultat za svaki test slučaj, koji daje opseg očekivanih vrednosti odziva sistema i koji ne daje rešenje testa u nekim test slučajevima

Upravo iz navedenih razloga mora se u fazi planiranja procesa testiranja značajna pažnja posvetiti mehanizmu odlučivanja o rezultatima testova. Referentni sistem koji se testira je neophodan za ocenu kvaliteta softvera i njegova prirodazavis od više faktora koji se moraju uzeti u obzir pri projektovanju bilo ručnih ili automatskih testova sistema koji se testira.

Nije redak slučaj da se mora koristiti više Oracle mehanizama u samo jednom automatizovanom testu (eksperimentu) ili da se jedan Oracle mehanizam koristi u više testova. Ako se rezultat testa sistema koji se testira posmatra samo sa tačke gledišta odlučivanja o uspešnosti testa, tada postoje tri vrste Oracle mehanizama:

- *Oracle koji daje tačan rezultat tj. odziv sistema za svaki test slučaj,*
- *Oracle koji daje opseg očekivanih vrednosti odziva sistema i*
- *Oracle mehanizam koji ne daje rešenje testa u nekim test slučajevima.*

Relativno malo je opisan u literaturi problem referentnog sistema ili test Oracle mehanizma. Howden je prvi uveo termin „Test Oracle“. Weyuker je analizirala ograničenja u pronalaženju Oracle mehanizama.

Razvoj Oracle mehanizma na osnovu specifikacije softvera je diskutovan u literaturi. Barbey je dao strategiju u testiranju objektno-orientisanih programa koja koristi formalni način izrade specifikacije softvera koji ujedno služi i kao test Oracle. Ovaj pristup se zasniva na sličnim tehnikama koje su razvijene za verifikaciju apstraktnih tipova podataka (engl. abstract data types (ADT)), koje takođe ADT specifikaciju koriste kao Oracle.

Pregled Oracle mehanizama je opširnije dat u ovoj lekciji. Poston je dao listu Oracle mehanizama svrstanih u četri grupe: mehanizam provere opsega vrednosti (slično tehnici umetanja (engl. Assertions)), ručno izračunate očekivane vrednosti (pre-specifikacija), tehnika simulacije SUT i korišćenje referentnog SUT. Referentni SUT se dobija tako što se pri aktuelnom TS ocene rezultati testova, pa se prihvativi rezultati zabeleže pa se ti isti koriste kao referentni pri ponovljenim (regresionim) testovima.

Hoffman je dao vrlo dobar prikaz upotrebe više vrsta test Oracle mehanizama u procesu automatizacije TS, od mehanizama određivanja rezultata testova na bazi aproksimacije, statističkih metoda, ugrađenih provera i druge moguće heurističke mehanizme koji su od praktičnog značaja. Zbog toga se u ovoj lekciji daje opis karakteristika, tipova, odgovarajući primeri i metod rangiranja Oracle mehanizama sa aspekta efikasne realizacije u PTS.

▼ Poglavlje 6

Pokazna vežba - Testiranje softvera

FUNKCIONALNOSTI SISTEMA ZA REGULISANJE SAOBRĀČAJA U JEDNOSMERNOM TUNELU

Funkcionalnosti sistema predstavljaju sve mogućnosti koje korisnik može da ima i kao takve moraju biti testirane kako bi se obezbedila što veća pouzdanost sistema.

U lekciji 1. je data softverska i hardverska specifikacija sistema za regulisanje saobraćaja u jednosmernom tunelu na osnovu kojih proizilaze funkcije sistema koje trebaju biti testirane. To su:

- Proveravanje svetla na semaforu
 - Slučaj očitavanja zelenog svetla na semaforu
 - Ulaz u tunel
 - Očitavanje na senzoru ulaza (podrazumeva merenje brzine i dužine vozila)
 - Izlazak iz tunela
 - Očitavanje na izlaznom senzoru
 - Slučaj očitavanja crvenog svetla
 - Zaustavljanje na definisanoj udaljenosti
 - Očitavanje zelenog svetla

TESTIRANJE SISTEMA ZA REGULISANJE SAOBRĀČAJA UNUTAR TUNELA NA BAZI SPECIFIKACIJE - DEFINISANJE GLAVNIH KORAKA

U okviru dizajn faze test slučajevi su bazirani na dijagrame slučajeva korišćenja.

Prilikom testiranja sistema na bazi specifikacije potrebno je proći kroz sve funkcije sistema od dolaska korisnika ispred tunela do njegovog izlaska iz istog i na taj način uvideti da li se radi o pravilnom procesu ili u njemu ipak postoje neki nedostaci. Prvo je potrebno definisati složenost procesa i opisati korake:

1. Korisnik ima interakciju sa semaforom (vidi zeleno ili crveno svetlo)
2. Korisnik dobija svetlo od semafora
3. Ukoliko je zeleno svetlo, korisnik prolazi kroz očitavanje svoje brzine i dužine na ulaznom senzoru

4. Korisnik prolazi kroz tunel
5. Korisnik se očitava na izlaznom senzoru

Vreme izrade grupne vežbe 45 minuta.

Glavni koraci u sistemskom procesu	1. Korisnik očitava svetlo na semaforu i dobija crveno ili zeleno svetlo 2. Korisnik prolazi kroz senzor očitavanja na ulazu (dopler semafor) 3. Korisnik prolazi kroz tunel 4. Korisnik prolazi kroz senzor očitavanja na izlazu 5. Korisnik izlazi iz tunela
Mogući problemi na osnovu kojih su izvedeni test slučajevi	1. Korisnik nije u mogućnosti da očita svetlo na semaforu 2. Korisniku se ne očitava brzina 3. Svetlo na semaforu se ne menja

Slika 6.1.1 Glavni koraci sistemskog procesa [Izvor: NM SE321-2020/2021.]

U okviru ovih pet koraka opisan je proces prolaska jednog vozila kroz tunel i to su glavni slučajevi korišćenja u idealnim uslovima. Na osnovu ovih podataka kreirani su test slučajevi koji će pokazati funkcionalnost sistema. Detaljan opis koraka prikazan je na slici 6.1.

TEST SLUČAJ 1: NEPRAVILNO FUNKCIONISANJE SEMAFORA NA ULASKU U TUNEL

Prvi test slučaj bavi se nepravilnim funkcionisanjem semafora na ulasku u tunel koji je predstavljen i opisan tabelom.

Naslov	Svetlo na semaforu se ne menja	Rev 1	Autor	Nebojša Gavrilović	Datum	18.01.2015
Cilj	Cilj je provjeriti situaciju kada zakaže semafor na ulasku u tunel	Reference				
Test uslovi	Vreme neophodno za izradu test slučaja		10 min	Neophodno vreme za Izvršenje test slučaja	5 min	

Opis postavke za testiranje	
Semafor je podešen da prikazuje naizmenično crveno i zeleno svetlo na osnovu timer-a	
Pokrenuta je aplikacija	
Korisnik treba da očita svetlo na semaforu	

Definicija testa			Izvršenje testa		
	Uslovi	Ulagani podaci	Očekivani	Aktuelni rezultati	Broj problema
*	Aplikacija je podešena, semafor je kreiran i postavljen je timer i čeka se prvi korisnik.	<ul style="list-style-type: none"> Svetlo na semaforu Korisnički podaci 	Stanje korisnika na definisanoj udaljenosti	Ukoliko semafor ne radi, proces ne može da se nastavi i to je signal hardverski postavljanje table za upozorenje kojoj signalizira semafor pravo rešenje	Primećen je nedostatak koji će biti otklonjen

Opis postuslova	
Semafor je poslao grešku u sistem	
Sistem je potrebno hardverski doraditi	

Slika 6.1.2 Rezultati test slučaja nepravilnog funkcionisanja semafora [Izvor: NM SE321-2020/2021.]

TEST SLUČAJ 2: NEOČITAVANJE BRZINE NA SENZORU ULASKA

Dat je primer test slučaja neočitavanja brzine na senzoru ulaska

Na slici 6.3 prikazana je tabela sa postavkom i rezultatima testiranja neočitavanja brzine na senzoru ulaska.

Naslov	Korisniku se ne očitava brzina na ulasku u tunel	Rev 1	Autor	Nebojša Gavrilović	Datum	27.08.2020		
Cilj	Cilj je provera situacije kada zakaže senzor kojim se meri brzina kretanja	Reference						
Test uslovi	Vreme neophodno za izradu test slučaja	10 min	Neophodno vreme za Izvršenje test slučaja	5 min				
Opis postavke za testiranje								
Semafor je podešen da prikazuje naizmenično crveno i zeleno svetlo na osnovu timer-a								
Pokrenuta je aplikacija								
Nakon prolaska semafora korisnik dolazi na senzor ulaza								
Definicija testa				Izvršenje testa				
	Uslovi	Ulagani podaci	Očekivani	Aktuelni rezultati	Broj problema			
•	Aplikacija je podešena, semafor je kreiran i postavljen je timer i a nakon toga korisnik dolazi na merenje brzine i dužine vozila	<ul style="list-style-type: none"> Svetlo na semaforu Korisnički podaci 	Očitavanje brzine korisnika	Korisniku je potrebno očitati brzinu i dužinu vozila. Ukoliko senzor ne radi prijaviće grešku sistemu i na semaforu će se pojaviti crveno svetlo.	Nije bilo problema			
Opis postuslova								
Senzor ulaska je prijavio grešku sistemu								
Semafor je postavio crveno svetlo i onemogućio ulazak vozila								

Slika 6.1.3 Rezultati test slučaja neočitivanja brzine na senzoru ulaska [Izvor: NM SE321-2020/2021.]

TEST SLUČAJ 3: NEOČITAVANJEM SVETLA NA SENZORU ULASKA U TUNEL

Treći test slučaj se bavi neočitavanjem svetla na senzoru ulaska u tunel koji je predstavljen i opisan tabelom.

Naslov	Korisnik nije u mogućnosti da očita svetlo na semaforu	Rev 1	Autor	Nebojša Gavrilović	Datum	18.01.2015
Cilj	Cilj je provera situacije ukoliko korisnik iz nekog razloga ne očita svetlo na semaforu	Reference				
Test uslovi	Vreme neophodno za izradu test slučaja	10 min	Neophodno vreme za Izvršenje test slučaja		5 min	

Opis postavke za testiranje	
Semafor je podešen da prikazuje naizmenično crveno i zeleno svetlo na osnovu timer-a	
Pokrenuta je aplikacija	
Semafor pokazuje zeleno svetlo	

Definicija testa			Izvršenje testa		
	Uslovi	Ulazni podaci	Očekivani	Aktuelni rezultati	Broj problema
*	Aplikacija je podešena, semafor je kreiran i postavljen je timer i čeka se prvi korisnik	• Svetlo na semaforu	Ulazak korisnika u tunel	Korisnik je prišao semaforu po definisanom UseCase dijagramu potrebitno da očita svetlo. Ukoliko to ne uspe postoji rezervno rešenje unavedenim vidu obaveštenja koji se nalazi na vrhu tunela a prati rad prvog semafora.	Eventualni problemi biće potencijalno rezolvirani u sledećem rešenju.

Opis postuslova	
Korisnik je video svetlo na semaforu	
Aplikacija je prikazala da uz pravu hardversku podršku vrši precizan rad	

Slika 6.1.4 Rezultati test slučaja neočitivanja svetla na semaforu [Izvor: NM SE321-2020/2021.]

TESTIRANJE NA BAZI IMPLEMENTACIJE - PRIMER PROGRAMSKOG KODA KLASA PROGRAMA

Dat je programski kod klase "SistemZaKontroluSaobraćaja" i "Tunel" na kojima će biti izvršeno testiranje implementacije sistema primenom tehnika testiranja.

Programski kod klase "SistemZaKontroluSaobraćaja":

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package sistemzakontrolusaobracaja;

/**
 * Test klasa za klasu Tunel
 * @author Nebojsa
 */
public class SistemZaKontroluSaobracaja {

```

```
public SistemZaKontroluSaobracaja() {
    System.out.println("----- Testiranje klase tunel -----");
    Tunel tunel = new Tunel(20); // Pravimo instancu klase tunel i stavljamo da je maksimalan broj kola koja mogu ući u tunel 20
    System.out.println("Postavili smo da je maksimalan broj kola koja mogu ući u tunel 20");
    System.out.println("Ocitavamo broj kola u tunelu: " + tunel.getCarsInTunel());
    System.out.println("Ocitavamo svetlo na semaforu: " + tunel.getLight());
    //Pomocu for petlje ubacujemo 10 kola u tunel
    for(int i = 0; i < 10; i++){
        tunel.carEnter();
        System.out.println("Vozilo broj " + (i+1) + " ulazi u tunel");
    }
    System.out.println("Ocitavamo broj kola u tunelu: " + tunel.getCarsInTunel());
    System.out.println("Ocitavamo svetlo na semaforu: " + tunel.getLight());
    //Ulaze jos deset vozila u tunel
    for(int i = 10; i < 20; i++){
        tunel.carEnter();
        System.out.println("Vozilo broj " + (i+1) + " ulazi u tunel");
    }
    System.out.println("Ocitavamo broj kola u tunelu: " + tunel.getCarsInTunel());
    System.out.println("Ocitavamo svetlo na semaforu: " + tunel.getLight());
    //Izlaze iz tunela 5 vozila
    for(int i = 0; i < 5; i++){
        tunel.carExit();
        System.out.println("Vozilo broj " + (i+1) + " izlazi iz tunela");
    }
    System.out.println("Ocitavamo broj kola u tunelu: " + tunel.getCarsInTunel());
    System.out.println("Ocitavamo svetlo na semaforu: " + tunel.getLight());
    //Ocitavamo broj uslih i izaslih vozila iz tunela
    System.out.println("Broj vozila izaslih iz tunela od prethodnog ocitavanja: " + tunel.carsExited());
    System.out.println("Broj vozila uslih u tunel od prethodnog ocitavanja: " + tunel.carsEntered());
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    new SistemZaKontroluSaobracaja();
}
```

Programski kod klase Tunel:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package sistemzakontrolusaobracaja;

/**
 * Klasa kontrolise broj vozila u tunelu
 * @author Nebojsa
 */
public class Tunel {
    private int carsInTunel; // Atribut koji predstavlja trenutan broj vozila u tunelu
    private int n; // Maksimalan broj vozila u tunelu
    private Color light; // Svetlo na semaforu
    private int tempCarExited; // Atribut koji prebrojava koliko je kola u nekom periodu izaslo iz tunela
    private int tempCarEntered; // Atribut koji prebrojava koliko je kola u nekom periodu uslo u tunel

    /**
     * Modifikovan konstruktor, prima maksimalan broj vozila u tunelu, ostale podatke setuje na pocetnu vrednost
     * @param n
     */
    public Tunel(int n) {
        this.n = n;
        carsInTunel = 0;
        this.light = Color.Green;
        tempCarEntered = 0;
        tempCarExited = 0;
    }

    /**
     * Metoda koja se ocitava kad udju jedna kola u tunel
     */
    public void carEnter(){
        carsInTunel++;
        tempCarEntered++;
        checkLight();
    }

    /**
     * Metoda koja se ocitava kada izadje jedno vozilo iz tunela
     */
    public void carExit(){
        carsInTunel--;
        tempCarExited++;
        checkLight();
    }
}
```

```
/*
 * Metoda vraca broj vozila u tunelu
 * @return the carsInTunel
 */
public int getCarsInTunel() {
    return carsInTunel;
}

/**
 * Metoda setuje broj vozila u tunelu
 * @param carsInTunel the carsInTunel to set
 */
public void setCarsInTunel(int carsInTunel) {
    this.carsInTunel = carsInTunel;
}

/**
 * Metoda vraca maksimalan broj vozila u tunelu
 * @return the n
 */
public int getN() {
    return n;
}

/**
 * Metoda setuje maksimalan broj vozila u tunelu
 * @param n the n to set
 */
public void setN(int n) {
    this.n = n;
}

/**
 * Metoda vraca svetlo na semaforu
 * @return the light
 */
public Color getLight() {
    return light;
}

/**
 * Metoda postavlja svetlo na semaforu
 * @param light the light to set
 */
public void setLight(Color light) {
    this.light = light;
}

/**
 * Metoda proverava da li je pravo svetlo na semaforu
 */
public void checkLight(){
    if((carsInTunel>=n)&&(light.equals(Color.Green))){
```

```
        light = Color.Red;
    }else if ((carsInTunel<n)&&(light.equals(Color.Red))){ 
        light = Color.Green;
    }
}

/**
 * Metoda vraca broj kola koja su izasla iz tunela u periodu od proslog
ocitavanja
 * @return
 */
public int carsExited(){
    int temp = tempCarExited;
    tempCarExited=0;
    return temp;
}

/**
 * Metoda vraca broj kola koja su usla u tunel u periodu od proslog ocitavanja
 * @return
 */
public int carsEntered(){
    int temp = tempCarEntered;
    tempCarEntered=0;
    return temp;
}

/**
 * Metoda pravi delay 10 sekundi
 */
public void delay10Seconds() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex) {
    }
}
}
```

TESTIRANJE NA BAZI IMPLEMENTACIJE SISTEMA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU - TEST SLUČAJ 1

Prvi test slučaj u okviru faze implementacije biće izvršen korišćenjem JUnit testova i prikazan je proces izvršavanja testa i dobijeni rezultati nakon izvršenog testa.

U okviru ove faze kreirani su JUnit testovi na kojima je prikazan rad same aplikacije i programskog koda. Biće ispitane i funkcionalnosti sistema kao i uočeni eventualni propusti. Test slučajevi su rađeni u NetBeans razvojnem okruženju.

run:

```
----- Testiranje klase tunel -----
Postavili smo da je maksimalan broj kola koja mogu ući u tunel 20
Ocitavamo broj kola u tunelu: 0
Ocitavamo svetlo na semaforu: Green
Vozilo broj 1 ulazi u tunel
Vozilo broj 2 ulazi u tunel
Vozilo broj 3 ulazi u tunel
Vozilo broj 4 ulazi u tunel
Vozilo broj 5 ulazi u tunel
Vozilo broj 6 ulazi u tunel
Vozilo broj 7 ulazi u tunel
Vozilo broj 8 ulazi u tunel
Vozilo broj 9 ulazi u tunel
Vozilo broj 10 ulazi u tunel
Ocitavamo broj kola u tunelu: 10
Ocitavamo svetlo na semaforu: Green
Vozilo broj 11 ulazi u tunel
Vozilo broj 12 ulazi u tunel
Vozilo broj 13 ulazi u tunel
Vozilo broj 14 ulazi u tunel
Vozilo broj 15 ulazi u tunel
Vozilo broj 16 ulazi u tunel
Vozilo broj 17 ulazi u tunel
Vozilo broj 18 ulazi u tunel
Vozilo broj 19 ulazi u tunel
Vozilo broj 20 ulazi u tunel
Ocitavamo broj kola u tunelu: 20
Ocitavamo svetlo na semaforu: Red
Vozilo broj 1 izlazi iz tunela
Vozilo broj 2 izlazi iz tunela
Vozilo broj 3 izlazi iz tunela
Vozilo broj 4 izlazi iz tunela
Vozilo broj 5 izlazi iz tunela
Ocitavamo broj kola u tunelu: 15
Ocitavamo svetlo na semaforu: Green
Broj vozila izaslih iz tunela od prethodnog ocitavanja: 5
Broj vozila uslih u tunel od prethodnog ocitavanja: 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

Test je kreiran da zadovolji broj vozila koji ulaze u tunel. U ovom testu to je broj 20 da bi se na realnom broju vozila videlo da li sistem funkcioniše na pravi način. Senzori očitavaju ulazak vozila u tunel i na taj način menjaju svetla na semaforu. Kada vozilo izđe iz tunela izlazni senzor šalje signal da se proces može nastaviti i tada se pokazuje zeleno svetlo i nastavlja proces očitavanja ulaznih vozila.

TESTIRANJE NA BAZI IMPLEMENTACIJE SISTEMA ZA REGULISANJE SAOBRAĆAJA U JEDNOSMERNOM TUNELU - TEST SLUČAJ 2

Predstavljen je drugi test slučaj u fazi implementacije, izvršavanje i dobijeni rezultati.

run:

```
----- Testiranje klase tunel -----
Postavili smo da je maksimalan broj kola koja mogu ući u tunel 20
Ocitavamo broj kola u tunelu: 0
Ocitavamo svetlo na semaforu: Green
Vozilo broj 1 ulazi u tunel
Vozilo broj 2 ulazi u tunel
Vozilo broj 3 ulazi u tunel
Vozilo broj 4 ulazi u tunel
Vozilo broj 5 ulazi u tunel
Vozilo broj 6 ulazi u tunel
Vozilo broj 7 ulazi u tunel
Vozilo broj 8 ulazi u tunel
Vozilo broj 9 ulazi u tunel
Vozilo broj 10 ulazi u tunel
Ocitavamo broj kola u tunelu: 10
Ocitavamo svetlo na semaforu: Green
Vozilo broj 11 ulazi u tunel
Vozilo broj 12 ulazi u tunel
Vozilo broj 13 ulazi u tunel
Vozilo broj 14 ulazi u tunel
Vozilo broj 15 ulazi u tunel
Vozilo broj 16 ulazi u tunel
Vozilo broj 17 ulazi u tunel
Vozilo broj 18 ulazi u tunel
Vozilo broj 19 ulazi u tunel
Vozilo broj 20 ulazi u tunel
Ocitavamo broj kola u tunelu: 20
Ocitavamo svetlo na semaforu: Green
Ocitavamo svetlo na semaforu: Red
Vozilo broj 1 izlazi iz tunela
Vozilo broj 2 izlazi iz tunela
Vozilo broj 3 izlazi iz tunela
Vozilo broj 4 izlazi iz tunela
Vozilo broj 5 izlazi iz tunela
Ocitavamo broj kola u tunelu: 15
Ocitavamo svetlo na semaforu: Green
Broj vozila izaslih iz tunela od prethodnog ocitavanja: 5
Broj vozila uslih u tunel od prethodnog ocitavanja: 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

Na ovom test slučaju ponovljen je broj vozila koja ulaze (20 vozila) ali je takođe napravljena izmena da kada u tunel uđe 20 vozila svetlo i dalje bude zeleno. Ukoliko se to desi, nastaje

problem, i zato je prikazan ovaj test slučaj. Rešenje će biti uvođenje upozorenja koji se nalazi na vrhu ulaza u tunel i koji će rešiti ove propuste dok se to u aplikaciji ne može rešiti bez hardverskog dela. U test slučaju se vidi da je nakon zelenog svetla prikazano odmah crveno uz upozorenje koje će pokazati korisnicima da je potrebno da se zaustave zbog nemogućnosti ulaska u tunel. Ovaj problem je uviđen u testiranju u dizajn fazi pa je ovaj test kreiran da bi se videla moguća softverska rešenja i ispravili ovi nedostaci. Sama aplikacija radi na principu merenja dužine i brzine vozila i na osnovu toga propuštanja vozila u tunel, ali ukoliko se definiše broj vozila koja moraju da uđu (eventualno zbog nastalih gužvi u saobraćaju na ulazu) dolazi do ovog problema i ovaj test služi da se taj problem reši.

KRITERIJUM USPEŠNOSTI SISTEMA I SUSPENZIJE PROCESA TESTIRANJA

Uspešnost sistema ce u biti omogućena ukoliko su se svi test slučajevi kritičnog , visokog, srednjeg i niskog prioriteta uspešno završeni.

Testiranje može biti suspendovano samo u slučaju nepopravljive greške vezane za sam sistem (otkaz hardvera, neispravnost third-party softvera), s tim da će se testiranje nastaviti kada odgovarajući problemi budu otklonjeni. Pojedini test slučajevi mogu biti suspendovani, ali moraju biti izvršeni, u slučaju neuspeha komponenti potrebnih za test slučaj.

Kada treba prekinuti testiranje

1. Broj bagova je približno jednak krajnjem očekivanom – svi rizici najvišeg nivoa su otklonjeni
2. Svi planirani testovi su sprovedeni – po master test planu pokrivenost koda 97%
3. Sav kod je pređen
4. Svaka funkcija sistema radi

RIZICI

Sledeće stavke se mogu identifikovati kao najverovatniji projektni rizici koji mogu dovesti do nepredviđenih događaja.

- Server baze podataka postaje nedostupan - Testiranje će se, u tom slučaju, odložiti dok situacija ne bude razrešena - Moguća je potreba da se uposli dodatno osoblje za rad na testiranju
- Test alati nisu dostupni/ne rade kako treba - Ovo bi dovelo do odlaganja automatizovanog testiranja i rezultovalo u povećanju manualnog testiranja, opet uz mogućnost potrebe upošljavanja dodatnog osoblja
- Test osoblje je nepotpuno/nedostupno. Kao posledica ovoga, mora se voditi računa o mogućnosti upošljavanja dodatnog osoblja.
- Veliki broj grešaka/defekata praktično onemogućavaju pokretanje ostalih testova - Ma koliko testova uspešno bilo izvršeno, IS direktor u dogovoru sa rukovodstvom "Kompanije" će doneti odluku da li broj defekata/incidenata iziskuje odlaganje finalnog puštanja sistema u rad

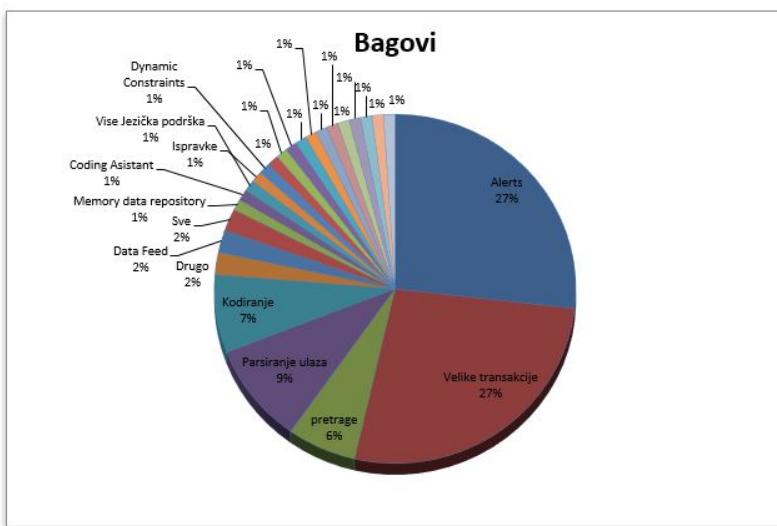
- Nedovoljno vremena da se završe svi testovi - Ukoliko vreme ne može biti produženo, pojedini individualni testovi će biti preskočeni, počevši od onih sa najmanjim prioritetom.

Analiza rizika identificuje one rizike čije ostvarenje može da ugrozi uspešnost projekta. Koristimo skalu od 0 do 5 da ocenimo uticaj rizika. Uticaj od 5 znači da projekat neće biti realizovan.

Što se test okruženja tiče počemo od hardvera

- Intel® Core™ i5-4460, 22nm
- GeForce GTX750-Ti 2GB DDR5, 128bit
- DDR3 8GB
- Western Digital 1TB HDD

Ostali hardver nije bitan za testiranje sistema



Slika 6.1.5 Pie chart dijagram [Izvor: NM SE321-2020/2021.]

ODRŽAVANJE SISTEMA

Podrazumeva kreiranje plana održavanja koji analizira sve nedostatke u fazi razvoja softvera.

Nakon izvršenja testova „tim“ za testiranje je zaključio da softver nema nikakvih grešaka prilikom izgradnje i da je jedina greska koja može da se desi prepun hard disk sa podacima. U tom slučaju tim za održavanje softvera treba da dođe i da sve podatke prebaci na mobilne medijume, kao sto su DVD, eksterni hard disk.

Tim za održavanje treba da ima kompletну tehničku dokumentaciju, da ako bi došlo do nasilnog tehničkog otkaza izazvanim lošim rukovanjem operativnim sistemom, ili oštećenja servera baze podataka, mogli da reaguju u cilju pravljenja plana za otklanjanje kvara. Tokom održavanja mogu da nastanu sledeće situacije:

1. *Analiza problema modifikacije:* U slučaju da softver aktuelan ali da se tokom vremena razvila potreba za nekim novim mogućnostima softvera treba analizirati

dizajn i naći način kako da se implementiraju date funkcije. Modifikacija je moguća u slučaju pronalaženja optimizovanijeg koda od postojećeg. U svakom slučaju, tim za održavanje treba da poseduje kompletну tehničku dokumentaciju kako bi mogao da analizira softver.

2. *Implementacija modifikacije:* Kada je modifikacija analizirana, treba napraviti plan kako je ubaciti u već postojeći softver a da ne ugrozi podatke koji su na njemu. Napraviti izveštaj o tome kako se modifikovana aplikacija ponaša
3. *Testiranje modifikacije:* Testiranje modifikacije je izveštaj o tome kako korisnik reaguje na promene unutar aplikacije. Proveravaju se promene novim testom korisnosti. Ako su ocene manje od prethodnih treba se odmah vratiti na Analizu problema modifikacije
4. *Proces migracije:* U ovom slučaju korisnik je obavezan da se pridržava pravila za imlementaciju softvera. Re-implementacija na drugi operativni sistem ili na druge računare nije sastavni deo ugovora
5. *Penzionisanje:* Vremenom svaki softver stari i stalno izlaze novije verzije Sistema. U slučaju izlaska druge verzije sistema tim za održavanje je dužan da prebaci sve podatke iz prethodne verzije i da ih ubaci u noviju aplikaciju kako to ne bi remetilo ili ugrozilo dosadašnji rad klijenta.

✓ 6.1 Vežba za samostalni rad

TESTIRANJE NA BAZI SPECIFIKACIJE SISTEMA

Na osnovu obrađenih tehnika testiranja softvera uraditi testiranje različitih delova ISUM sistema.

Odabrati deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka). Za izabrani deo uradite sledeće zadatke:

1. Definisati pet scenarija testiranja za odabrani deo sistema na bazi specifikacije koju ste napravili i nju opisali način na koji vi mislite da odabrani deo ISUM-a treba da radi. **(vreme izrade zadatka 30 minuta)**
2. Izvršiti testiranje aplikacije **(vreme izrade zadatka 30 minuta)**
3. Dokumentovati izvršene test slučajeve i napraviti tabelu po uzoru na tabelu datu u vežbama za testiranje na bazi specifikacije sistema. **(vreme izrade zadatka 30 minuta)**

✓ Poglavlje 7

Domaći zadatak

ČETVRTI DOMAĆI ZADATAK - VREME IZRADE 150 MIN.

Nakon četvrte lekcije potrebno je uraditi četvrti domaći zadatak

Za proizvoljno odabranu aplikaciju koja je rađena na nekom od predmeta koji ste prethodno slušali i položili primeniti sledeće:

1. Definisati test slučajeve primenom testiranja na bazi specifikacije sistema i izvršiti ih.
2. Napraviti izveštaj o uočenim greškama ukoliko ih ima
3. Izvršiti JUnit testiranje dve odabrane klase aplikacije.

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ04-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

nikola.petrovic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ04.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templejt) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Strategije i tehnike TS

UPOTREBLJIVOST TEHNIKA I STRATEGIJA TESTIRANJA SOFTVERA

Upotrebljivost TDG zavisi od raspoloživosti rešenja testova koji se nameravaju izvesti tj. koji je to postupak za što lakše i pouzdanije pronalaženje korektnog rešenja datog testa.

Teorijska ograničenja, koja su već istaknuta u prethodnim lekcijama, jasno ukazuju da je nemoguće predložiti i implementirati metodologiju testiranja softvera koja je potpuno tačna i primenljiva na bilo koju vrstu softvera (po aplikaciji i složenosti). Zato se u ovoj lekciji nismo bavili iscrpno i dominantno sa svim tehnikama detekcije grešaka u softveru, već samo onoliko koliko je potrebno sa aspekta njihovog značaja u generičkom rešenju integralnog i optimiziranog procesa testiranja softvera koje su ocenjene tako da u ovaj proces predstavljaju jednu zasebnu oblast (bazen, skup) mogućih tehnika detekcija grešaka koje treba oceniti i njihove slabosti izbeći, dopunjavajući jedne sa drugim pri izradi optimalnog scenarija testiranja softvera u konkretnom slučaju.

Pošto su tehnike detekcije grešaka jedan vid verifikacije softvera, one se ne mogu primeniti bez postojanja projektnih zahteva i specifikacije softverskog proizvoda. Pored pisanih zahteva (u bilo kom obliku) u specifikaciji, navedenih standarda koje softver treba da zadovolji i slično, moraju se uzeti u obzir sve dostupne implicitne informacije (u projektnoj dokumentaciji, korisničkim, servisnim uputstvima i sl.) iz kojih se vidi očekivano ponašanje, karakteristike projektovanog softvera. Pored očiglednih inherentnih ograničenja raspoloživih tehnika detekcije grešaka, postoji više praktičnih pitanja koja mogu sprečiti primenu neke od tehnika u konkretnom slučaju.

Generisanje očekivanih rezultata testova izvodi se pomoću nekog pouzdanog mehanizma koji se engleski zove Test Oracle ili prosto rešenje testova. Međutim, mnoge tehnike detekcije grešaka zanemaruju napor za pronalaženje efikasnog načina generisanja rešenja testova, dok neke tehnike o tome uopšte ne vode računa iako je to kritična aktivnost u testiranju. Kvalitet tj. upotrebljivost tehnika zavisi od raspoloživosti rešenja testova koji se nameravaju izvesti tj. koji je to postupak za što lakše i pouzdanije pronalaženje korektnog rešenja datog testa, odnosno očekivanog ponašanja softvera.

LITERATURA ZA LEKCIJU 04

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>).
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>).
3. dr Miladin Stefanović, mr Slobodan Mitrović, mr Milan Erić, MODEL KVALITETA SOFTVERA, Festival kvaliteta 2006., 33. Nacionalna konferencija o kvalitetu, Kragujevac, 10. - 12. maj 2006.
4. Programiranje korisničkog interfejsa Testiranje, evaluacija, greške, <http://ms1pki.etf.rs/predavanja/2015/PKI%2009.pdf>.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Mr Boris Todorović, EVALUACIJA KVALITETA SOFTVERA PREMA ISO 9126 STANDARDU, DOI 10.7251/POS1208099T.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Testiranje softvera – tehnika crne
kutije (funkcionalno testiranje)

Lekcija 05

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 05

TESTIRANJE SOFTVERA – TEHNIKA CRNE KUTIJE (FUNKCIONALNO TESTIRANJE)

- ✓ Testiranje softvera – tehnika crne kutije (funkcionalno testiranje)
- ✓ Poglavlje 1: Ručno i automatsko testiranje
- ✓ Poglavlje 2: Metoda crne kutije – “Black-box”
- ✓ Poglavlje 3: Klase ekvivalencije
- ✓ Poglavlje 4: Analiza graničnih vrednosti
- ✓ Poglavlje 5: Tablice odlučivanja
- ✓ Poglavlje 6: Tehnika prelaska stanja
- ✓ Poglavlje 7: Grupna vežba: Tehnike crne kutije - 1 deo
- ✓ Poglavlje 8: Grupna vežba: Testiranje ATM-a metodom crne kutije
- ✓ Poglavlje 9: Domaći zadatak
- ✓ Testiranje softvera –tehnika crne kutije (funkcionalno testiranje) i Test Oracle mehanizmi

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Testiranje metodom crne kutije uvodi testove u slučajevima u kojima nije poznata detaljna, interna struktura sistema ili komponenta koja se testira.

Testiranje zasnovano na sistemskim zahtevima ili testiranje komponenti. Veoma često se izvode direktno, na osnovu dokumenata funkcionalnih zahteva, pri čemu se svaki funkcionalni zahtev prevodi u kriterijum funkcionalnog testa. Testna pokrivenost, pri funkcionalnom testiranju, obično se postiže izradom matrice tragova, koja obuhvata zahteve i testne kriterijume za testiranje tih zahteva. Nepokriveni zahtevi su zahtevi koji nemaju zadovoljavajuće, pridružene testne kriterijume.

Kada dokumenti sa funkcionalnim zahtevima nisu dostupni, tester mora da protumači zahteve sistema posmatranjem operativnog ponašanja sistema, ili komponenti pod testom. Kod ovakve vrste testiranja koje je nazvano testiranje crne kutije, dizajn i implementacija softvera nisu poznati testeru već se softver posmatra samo kao neka vrsta funkcije kojom se ulazi transformišu u izlaz. Testiranje metodom crne kutije uvodi testove u slučajevima u kojima nije poznata detaljna, interna struktura sistema ili komponente koja se testira. Zasnovano je na specifikaciji sistemskih zahteva koja je jedino poznata testeru.

Koristi se i više drugih termina za testiranje metodom crne kutije, uključujući:

- **testiranje na bazi specifikacija**
- **input / output testiranje**
- **funkcionalno testiranje**

✓ Poglavlje 1

Ručno i automatsko testiranje

RUČNO TESTIRANJE

Ručno testiranje je postupak testiranja softvera u kojem se test slučajevi ručno izvršavaju bez upotrebe automatizovanog alata.

Testiranje metodom crne kutije o čemu će u ovoj lekciji biti reči, spada u postupak ručnog testiranja gde spadaju još i metode testiranja bele kutije i testiranja sive kutije o čemu će biti reči u narednim lekcijama. Pored ručnog postoji još i automatski način testiranja.

Ručno testiranje je postupak testiranja softvera u kojem se test slučajevi ručno izvršavaju bez upotrebe automatizovanog alata. Sve slučajeve testiranja tester izvršava ručno, u skladu sa perspektivom krajnjeg korisnika. Ručnim testiranjem se proverava da li aplikacija radi, kao što je to naznačeno u dokumentu zahteva ili ne. Slučajevi testiranja se planiraju i implementiraju kako bi se pokrilo skoro 100% softverske aplikacije. Izveštaji o testovima se takođe generišu ručno.

Ručno testiranje je jedan od najvažnijih procesa testiranja jer se mogu naći i vidljivi i skriveni defekti softvera. Defekt se definiše kao razlika između očekivanih izlaza i izlaza, koje daje softver.

Ručno testiranje je obavezno za svaki novo razvijeni softver pre automatskog testiranja. Ovo testiranje zahteva velike napore i vreme, ali obezbeđuje da se dobije softver bez grešaka. Ručno testiranje zahteva poznavanje ručnih tehnika ispitivanja, ali ne i bilo koji automatizovani alat za testiranje. Ručno testiranje je neophodno jer je jedan od osnova testiranja softvera da „stopostotna automatizacija nije moguća“.

Testiranje crne kutije obavlja tester (test inženjer), koji proverava funkcionalnost aplikacije ili softvera u skladu sa potrebama klijenta / klijenta. Pri tome se kod ne vidi tokom izvođenja ispitivanja; zato je poznato pod nazivom testiranje crne kutije.

Testiranje bele kutije obavlja developer, gde proverava svaku liniju koda pre nego što ga predaje testnom inženjeru. Pošto je kod vidljiv programeru tokom testiranja, testiranje se naziva testiranje bele kutije.

Testiranje sive kutije je kombinacija testiranja bele kutije i crne kutije. Može da ga obavlja osoba koja poznaje i kodiranje i testiranje.

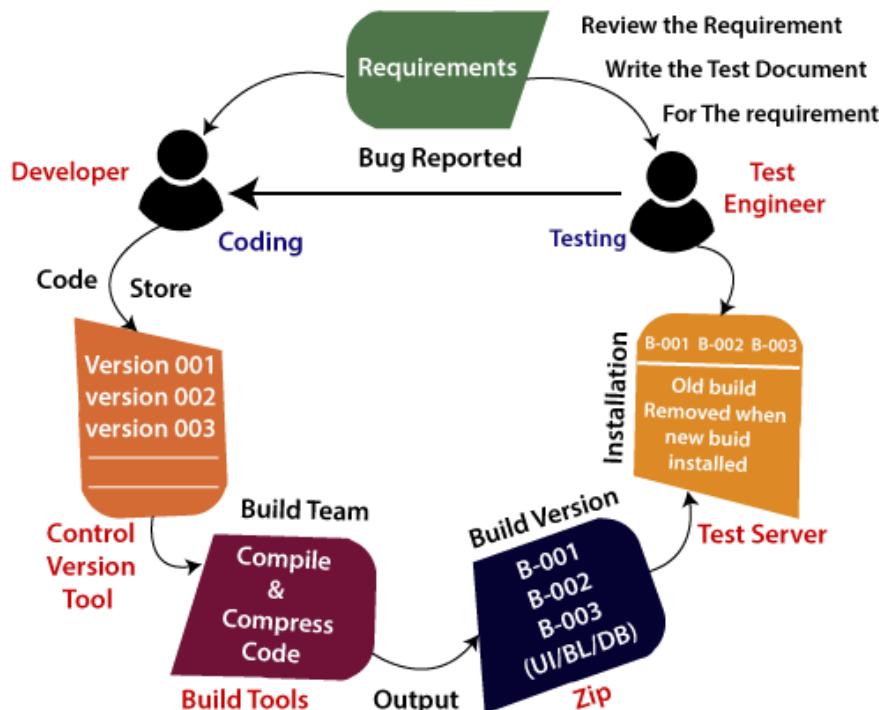
KAKO SE VRŠI RUČNO TESTIRANJE

Tester razvija test slučajeve prema dokumentu zahteva. Svi test slučajevi se izvode ručno korišćenjem testiranja crne kutije i bele kutije

Ručno testiranje se vrši na sledeći način:

1. Prvo, tester posmatra sva dokumenta koja su relevantna za softver kako bi odabrao oblasti za testiranje.
2. Tester analizira dokument zahteva kako bi pokrio sve zahteve koje je postavio kupac.
3. Tester razvija test slučajeve prema dokumentu zahteva.
4. Svi test slučajevi se izvode ručno korišćenjem testiranja crne kutije i bele kutije.
5. Ako se pojave greške, tim za testiranje obaveštava razvojni tim.
6. Razvojni tim ispravlja greške i softver daje timu za testiranje na ponovni test.

U toku ručnog testiranja vrši se proces bildovanja softvera (engl. Software Build Process) koji se vrši na način prikazan na slici 1.



Slika 1.1 Bildovanje softvera, Izvor: <https://www.javatpoint.com/manual-testing> [Izvor: NM SE321-2020/2021.]

BILDOVANJE SOFTVERA

Tokom tetsiranja se vrši proces bildovanja za šta se poštuje utvrđena procedura

Proces bildovanja softvera (engl. **software building**) vrši se na sledeći način:

- 1. Kada se zahtevi prikupe, formiraju se dva različita tima za razvoj i testiranje.
- 2. Nakon dobijanja zahteva, programer će početi sa pisanjem koda.
- 3. U međuvremenu, test inženjeri čitaju zahteve kako bi ih razumeli i pripremaju potrebne dokumente, do kada programer može dopuniti kod i smestiti ga u alat opšte poznat kao **Control Version tool**. Primarni cilj ovog alata je praćenje promena koje se izvršavaju nad postojećim kodom (file.om)
- 4. Kada se kod menja, tim promenama rukovodi jedan odvojeni tim, koji je poznat kao build team. Ovaj tim kompajlira i pakuje kod uz pomoć nekog alata za bildovanje (engl. **build tool**) čija je uloga da kod kompajlira tj. prevede ga iz višeg programskog jezika u mašinski jezik. Posle kompilacije, veličina file se povećava tako da se file mora upakovati (komprimovati). Izlaz iz alata za bildovanje je zip file, koji je poznat kao Build (aplikacije ili softvera). Svaki build ima jedinstveni broj kao (B001, B002).
- 5. Tada se taj konkretni Build instalira na testnom serveru. Nakon toga će testni inženjer pristupiti ovom testnom serveru uz pomoć URL-a za testiranje i započeti testiranje aplikacije.
- 6. Ako je test inženjer pronašao neku grešku (engl. **bug**), ona će biti prijavljena dotičnom programeru.
- 7. Tada programer reprodukuje bug na testnom serveru, popravlja bug i ponovo postavlja kod u Control version tool (instalira novu ažuriranu datoteku i uklanja staru datoteku); ovaj proces se nastavlja dok ne dobijemo stabilni **Build**.
- 8. Jednom kada dobijemo stabilni build, on će biti predat kupcu.

U ručnom testiranju, za različite vrste testiranja poput jedinice, integracije, sigurnosti, performansi i praćenja grešaka, su nam na raspolaganju različiti alati kao što su Jira, Bugzilla, Mantis, Zap, NUnit, Tessy, LoadRunner, Citrus, SonarQube itd. Neki od alata su otvorenog koda, a neki su komercijalni.

AUTOMATSKO TESTIRANJE

Proces testiranja se izvodi pomoću posebnih automatizovanih alata kojima se vrši kontrola izvršenja test slučajeva i upoređivanje stvarnog rezultata sa očekivanim rezultatom.

Kada se slučajevi testiranja izvršavaju pomoću automatizovanih alata za testiranje, takvo testiranje je poznato kao **automatsko testiranje**. Proces testiranja se izvodi pomoću posebnih automatizovanih alata kojima se vrši kontrola izvršenja test slučajeva i upoređivanje stvarnog rezultata sa očekivanim rezultatom. Automatsko testiranje zahteva prilično veliko ulaganje resursa i novca.

Generalno, u automatizovanom testiranju se testiraju akcije koje se ponavljaju kao što su npr. regresijski testovi. Alat za testiranje koji se koristi u automatskom testiranju se koristi ne samo za regresijsko testiranje, već i za automatizovanu interakciju s GUI-em, generisanje podataka, zapisivanje kvarova i instalaciju proizvoda.

Cilj automatskog testiranja je smanjiti ručne slučajeve testiranja, ali ne i eliminisati nijedan od njih. Testovi se mogu beležiti pomoću automatizovanih alata a tester može da ponovo reprodukuje testove prema zahtevima.

✓ Poglavlje 2

Metoda crne kutije –“Black-box”

FUNKCIONALNO I NEFUNKCIONALNO TESTIRANJE METODOM CRNE KUTIJE

U funkcionalno testiranje spada jedinično, integraciono, sistemsko i testiranje prihvatljivosti. U ne funkcionalno testiranje spada testiranje kompatibilnosti, performansi i upotrebljivosti

Testiranje crne kutije se deli na funkcionalno i nefunkcionalno testiranje.

U funkcionalno testiranje spada:

1. Testiranje jedinica ili jedinično testiranje (engl. **unit testing**)
2. Integraciono testiranje (engl. **integration testing**)
3. Testiranje sistema (engl. **system testing**)
4. Testiranje prihvatljivosti (engl. **user acceptance testing**)

U nefunkcionalno testiranje spada:

1. Testiranje kompatibilnosti (engl. **compatibility testing**)
2. Testiranje performansi (engl. **performance testing**)
3. Testiranje upotrebljivosti (engl. **usability testing**)

Ovde će uglavnom biti reči o funkcionalnom testiranju.

KARAKTERISTIKE FUNKCIONALNOG TESTIRANJA METODOM CRNE KUTIJE

Unutrašnja struktura, dizajn i implementacija softvera nisu poznati testeru već se softver posmatra samo kao neka vrsta funkcije kojom se ulazi transfo

Testiranje zasnovano na sistemskim zahtevima ili testiranje komponenti se veoma često izvode direktno, na osnovu dokumenata funkcionalnih zahteva, pri čemu se svaki funkcionalni zahtev prevodi u kriterijum funkcionalnog testa. **Testna pokrivenost, pri funkcionalnom testiranju, obično se postiže izradom matrice tragova, koja obuhvata zahteve i testne kriterijume za testiranje tih zahteva.** Nepokriveni zahtevi su zahtevi koji nemaju zadovoljavajuće, pridružene testne kriterijume.

Kada dokumenti sa funkcionalnim zahtevima nisu dostupni, tester mora da protumači zahteve sistema posmatranjem operativnog ponašanja sistema, ili komponenti pod testom.

Testiranje je nazvano funkcionalno testiranje crne kutije, zato što unutrašnja struktura, dizajn i implementacija softvera nisu poznati testeru, već se softver posmatra samo kao neka vrsta funkcije kojom se ulazi transformišu u izlaz. Ova vrsta testiranja uvodi testove u slučajevima u kojima nije poznata detaljna, interna struktura sistema ili komponente koja se testira. Zasnovano je na specifikaciji sistemskih zahteva koja je jedino poznata testeru.

Koristi se i više drugih termina za testiranje metodom crne kutije, uključujući testiranje na bazi specifikacija, input / output testiranje, funkcionalno testiranje.

Testiranje metodom crne kutije tipično se izvršava kao životni ciklus razvoja skoro kompletno integrisanog sistema, tokom integracionog testiranja, testiranja interfejsa, sistemskog testiranja i testiranja prihvatljivosti. Na tom nivou, komponente sistema su dovoljno integrisane da pokažu da li su kompletni zahtevi ispunjeni skoro u potpunosti. Tipovi grešaka, koje se obično pronađu tokom testiranja metodom crne kutije, uključuju:

1. neispravne ili nedostajuće funkcije
2. greške interfejsa, u načinu na koji različite funkcije međusobno povezane, u načinu na koji su povezani sistemski interfejsi sa fajlovima i strukturama podataka, ili u načinu povezivanja sistemskih interfejsa sa drugim sistemima npr. preko mreže
3. greške učitavanja i performansi i greške inicijalizacije i završetka.

KAKO SE VRŠI FUNKCIONALNO TESTIRANJE METODOM CRNE KUTIJE

Svi test slučajevi su dizajnirani uzimanjem u obzir ulaze i izlaze određene funkcije. Tester zna određene izlaze za određene ulaze, ali ne i o tome kako nastaje rezultat.

Ova vrsta testiranja se odvija na sledeći način:

1. Test je zasnovan na specifikaciji zahteva, pa se izvršava na početku.
2. Tester kreira pozitivan scenario testiranja i nepovoljan scenario testiranja odabirom važećih i nevažećih ulaznih vrednosti kako bi proverio da li ih softver pravilno ili nekorektno obrađuje.
3. U trećem koraku, tester razvija razne test slučajeve kao što su tabela odlučivanja, test svih parova, ekvivalentno parcelisanje, procena grešaka, grafikon uzroka-efekta itd.
4. Četvrta faza uključuje izvršenje svih test slučajeva.
5. U petom koraku, tester upoređuje očekivani izlaz sa stvarnim izlazom.
6. U šestom i poslednjem koraku, ako postoji neki nedostatak u softveru, on se ispravlja i ponovo testira.

Testiranje ne zahteva znanje programiranja. Svi test slučajevi su dizajnirani uzimanjem u obzir ulaze i izlaze određene funkcije. Tester zna određene izlaze za određene ulaze, ali ne i o tome kako nastaje rezultat.

Postoje razne tehnike koje se koriste u testiranju crne kutije za testiranje poput tehnike tablice odlučivanja, tehnike analize graničnih vrednosti, tranzicije stanja, ispitivanja svih parova, tehnike grafikona uzroka-efekta, tehnika ekvivalentnog parcelisanja, tehnika nagađanja grešaka, upotrebe slučaja korišćenja.

Test slučajevi se kreiraju razmatranjem specifikacije zahteva. Prilikom testiranja, testira se i pozitivni scenario testiranja tako što se uzimaju važeće vrednosti ulaza i nepovoljni scenario testiranja tako što se uzimaju nevažeće ulazne vrednosti da bi se odredio tačan izlaz. Test slučajevi su uglavnom dizajnirani za funkcionalno testiranje, ali se mogu koristiti i za nefunkcionalno testiranje. Test slučajeva dizajnira tim za testiranje, bez uključenja razvojnog tima softvera.

Najbitnije tehnike funkcionalnog testiranja metodom crne kutije o kojima će u ovom predavanju biti reči su:

1. Klase ekvivalencije
2. Testiranje graničnih vrednosti
3. Tabele odlučivanja
4. Testiranje zasnovano na modelu prelaska stanja

PREDNOSTI I NEDOSTACI FUNKCIONALNOG TESTIRANJA

Funkcionalno testiranje ima sledeće prednosti i nedostatke

Prednosti funkcionalnog testiranja su:

1. Testiranje može biti ne tehničko (od testera se ne zahtevaju tehnička znanja, kao i poznavanje tehničkih termina).
2. Ovo testiranje će najverovatnije pronaći one greške koje bi korisnik pronašao.
3. Testiranje pomaže u identifikovanju nejasnoća i protivrečnosti funkcionalnih specifikacija.
4. Test slučajevi mogu biti izrađeni po završetku funkcionalnih specifikacija.
5. Razvijeni testovi su nezavisni od implementacije pa se mogu koristiti i u slučaju kada se implementacija sistema promeni
6. Testovi se mogu razvijati paralelno sa razvojem programa čime se štedi vreme

Nedostaci funkcionalnog testiranja su:

1. Šanse za ponavljanje testova koje je već odradio programer.
2. Test ulaza (inputa) zauzima veliki deo prostora.
3. Teško je utvrditi sva moguća testiranja ulaza u ograničenom vremenu. Pisanje test slučaja je prilično sporo i teško.
4. Postoje šanse za posedovanje neidentifikovanih staza tokom testiranja tj. neke putanje kroz program mogu ostati netestirane

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Klase ekvivalencije

POJAM “KLASA EKVIVALENCIJE”

Kod funkcionalnih testiranja, tester može samo zaključiti o strukturi implementacije komunicirajući sa aplikacijom pomoću interfejsa definisanog sistemskom specifikacijom.

Analizira se samo izvršavanje specificiranih funkcija i vrši se provera ulaznih i izlaznih podataka. Tačnost izlaznih podataka proverava se na osnovu specifikacije zahteva za softver. U ovim testovima se ne vrši analiza izvornog koda. Problem funkcionalnog testiranja može da se pojavi u slučaju dvomislenih zahteva i nemogućnosti opisivanja svih načina korišćenja softvera. Skoro 30% svih grešaka u kodu posledica su problema nepotpunih ili neodređenih specifikacija.

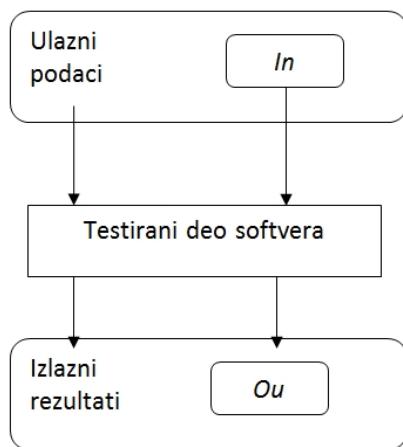
Sinonimi za black box uključuju: ponašanje, funkcionalnost, neprozirnost i zatvorenost. Test-primeri izvode se isključivo iz specifikacije dotičnog dela softvera. Ili, drugačije rečeno, softver se promatra kao “crna kutija” o kojoj jedino znamo da bi ona (prema specifikaciji) za zadane ulaze trebala proizvesti zadane izlaze.

Zbog postojanja grešaka, ulazi iz određenog (nepoznatog) skupa (In) će izazvati neispravno ponašanje, koje ćemo prepoznati po izlazima iz skupa (Ou). U postupku testiranja nastojimo izabrati ulaze za koje postoji velika verovatnoća da pripadaju skupu In . Izbor takvih test primera najčešće se zasniva na iskustvu softverskog inženjera. Ipak, moguće je i sistematičniji pristup zasnovan na podeli skupa svih mogućih ulaznih podataka (domene) na klase. Ovde reč “klasa” nije upotrebljena u smislu objektno-orientisanog pristupa, nego u smislu matematičkog pojma “klasa ekvivalencije”.

Očekujemo da će se program ponašati slično za sve podatke iz iste klase. Biramo barem po jedan test primer iz svake klase. Takođe je dobro isprobati “ivice” klase (Slika 1) .

Kod funkcionalnih testiranja, tester može samo zaključiti o strukturi implementacije komunicirajući sa aplikacijom pomoću interfejsa definisanog sistemskom specifikacijom.

Test slučajevi su dizajnirani tako da izaberete različite ulazne vrednosti, pokušavajući da aktivirate svako pravilo u funkcionalnim specifikacijama ponašanja softvera u obavljanju zahtevane misije u uređaju ili sistemu.



Slika 3.1 Funkcionalno testiranje metodom Black box - "crna kutija" [Izvor: NM SE321-2020/2021.]

TEORIJA (OSNOVE) STRATEGIJE KLASA EKVIVALENCIJE

Ideja je da se skup svih mogućih ulaznih podataka izdeli na podskupove (klase), pri čemu u istu klasu ulaze oni ulazni podaci koji daju iste (slične) rezultate

Ideja je da se skup svih mogućih ulaznih podataka izdeli na podskupove (klase), pri čemu u istu klasu ulaze oni ulazni podaci koji daju iste (slične) rezultate, na primer, otkrivaju istu grešku.

U idealnom slučaju podskupovi su međusobno disjunktni i pokrivaju ceo skup ulaza (relacija "sličnosti" ulaza je relacija ekvivalencije).

U cilju identifikacije klasa, posmatraju se svi uslovi koji proizilaze iz specifikacije. Za svaki uslov se posmatraju dva grupe klasa prema zadovoljenosti uslova:

1. *legalne klase obuhvataju ispravne situacije (ulazne podatke).*
2. *nelegalne klase obuhvataju sve ostale situacije (ulazne podatke).*

Uputstva za identifikaciju klasa:

1. Ako neki uslov podrazumeva opseg vrednosti nekog ulaznog podatka, postoji jedna legalna klasa (unutar opsega) i dve nelegalne klasе (ispod i iznad opsega).
2. Ako neki uslov podrazumeva broj vrednosti nekog ulaznog podatka, postoji jedna legalna klasа (dozvoljen broj pojavljivanja) i dve nelegalne klasе (ni jedno pojavljivanje i suviše pojavljivanja).
3. Ako neki uslov podrazumeva skup ulaznih vrednosti od kojih se svaka na poseban način obrađuje u programu, postoji za svaku vrednost po jedna legalna klasа i jedna zajednička nelegalna klasа.
4. Ako neki uslov podrazumeva situaciju obaveznosti ("mora biti slovo"), postoji jedna legalna klasа ("jeste slovo") i jedna nelegalna klasа ("nije slovo").

5. U slučaju da program ne tretira jednako sve elemente neke klase ekvivalencije, nju treba podeliti na više manjih klasa.

EKVIVALENTNO PARCELISANJE

Podrazumeva se da pojedinačna vrednost u ekvivalentnoj particiji reprezentuje sve druge vrednosti u particiji.

Ekvivalentno parcelisanje je zasnovano na pretpostavci da ulazi i izlazi softvera ili njene komponente mogu biti podeljeni u klase, u skladu sa specifikacijom softvera ili komponente, koje mogu biti tretirane slično (ekvivalentno) od strane komponente. Pretpostavka je da slični ulazi pobuđuju slične odgovore. Podrazumeva se da pojedinačna vrednost u ekvivalentnoj particiji, reprezentuje sve druge vrednosti u particiji.

Ovo se koristi da bi se umanjio problem nemogućnosti testiranja svake ulazne vrednosti. Cilj ekvivalentnog testiranja je izbor vrednosti, koje imaju ekvivalentnu obradu, tako da možemo pretpostaviti, da ako test prođe sa reprezentativnom vrednošću, proći će i sa svim ostalim vrednostima iste particije.

Neke ekvivalentne particije mogu uključiti kombinacije:

- ispravnih i neispravnih ulaznih i izlaznih vrednosti
- numeričkih vrednosti sa negativnim, pozitivnim i 0 vrednostima
- stringova koji su prazni i stringova koji nisu prazni
- liste koje su prazne, i one koje nisu prazne
- datoteke sa podacima koje postoje, ili ne postoje, koje se mogu čitati ili ne, i u koje se može pisati ili ne
- godine pre 2000-te ili posle 2000-te, prestupne godine ili proste godine (specijalan slučaj je 29. februar 2000. godine, koji ima sopstvenu, specijalnu obradu)
- datumi, koji su u 28, 29, 30. ili 31. danu u mesecu
- radni dani ili vikendi
- tip datoteke, na primer, tekst, formatizovani podaci, grafički, video ili zvučni izvor datoteke ili odredište datoteke, na primer, čvrsti disk, disketna jedinica, CD-ROM, mreža.

Primer : Razmislite o funkciji programa koju profesor koristi za određivanje konačne ocene studenta na ispitu, generate_grading, koja ima sledeću specifikaciju: Funkciji se prosleđuje ocena sa završnog ispita (do 75) i ocena za rad tokom kursa (do 25), na osnovu kojih se generiše ocena za predmet u opsegu od "A" do "D". Ocena za predmet se izračunava kao suma ocena na ispitu i rad na kursu, na sledeći način:

- veće od ili jednako 70 - "A"
- veće od ili jednako 50, ali manje od 70 - "B"
- veće od ili jednako 30, ali manje od 50 - "C"
- manje od 30 - "D"
- van očekivanih granica, generiše se greška ("FM").

Svi ulazi se prosleđuju u obliku celih brojeva.

ANALIZA PARTICIJA

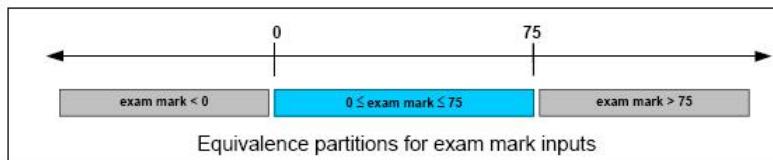
Particija je skup vrednosti, odabranih tako, da se za sve vrednosti u particiji očekuje da budu obrađene na isti način (na primer, imaju ekvivalentnu obradu).

Tester, za komponentu koja se testira, obezbeđuje ulazne i izlazne vrednosti. Ulazi i izlazi su izvedeni iz specifikacije ponašanja komponente.

Particija je skup vrednosti, odabranih tako, da se za sve vrednosti u particiji očekuje da budu obrađene na isti način (na primer, imaju ekvivalentnu obradu). Biraju se particije i za ispravne i za neispravne vrednosti (Slike 2 i 3).

Za funkciju generate_grading, identifikovana su dva ulaza:

- ocena na ispitu i
- ocena za rad tokom kursa.



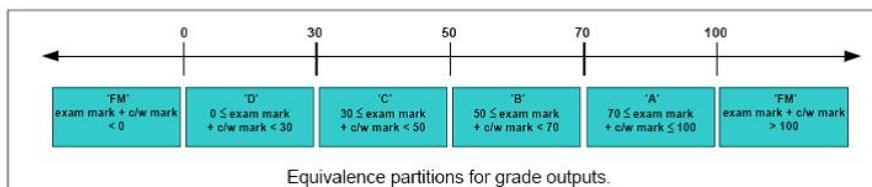
Slika 3.2 Ekvivalencija rasčlanjivanja za ispitivanje ulaznih ocena

Slika 3.3 Izbor particija [Izvor: NM SE321-2020/2021.]

Manje jasne ekvivalentne particije uključile bi i brojeve za ulaz koji nisu celi. Na primer:

- ocena na ispitu = realni broj
- ocena na ispitu = alfabetska
- ocena za rad na kursu = realni broj
- ocena za rad na kursu = alfabetska
- stepen

Sledeće, razmatraju se izlazi funkcije generate_grading kao na slici 4:



Slika 3.4 Ekvivalencija rasčlanjivanja za izlazni kvalitet [Izvor: NM SE321-2020/2021.]

ANALIZA PARTICIJA ZA NEISPRAVNE SLUČAJEVE

Za neispravne slučajeve možemo na primer predložiti neodgovarajuće izlaze

Takođe, ekvivalentne particije mogu biti razmatrane i za neispravne izlaze. Teško je identifikovati neoznačene izlaze, ali se moraju razmatrati. Ako je moguće usloviti jedno dešavanje, imaćemo identifikovani nedostatak u komponenti, ili u njenoj specifikaciji, ili i u komponenti i u specifikaciji.

Na primer, za izlaz stepena, možemo predložiti neodgovarajuće izlaze:

- ocena za predmet = 'E'
- ocena za predmet = 'A+'
- ocena za predmet = 'null'

Predložili smo 19 ekvivalentnih particija za ovaj primer. Kod razvoja ekvivalentnih particija, tester mora da napravi lični izbor. Na primer, dodatni, neispravni ulazi i neispravni izlazi. Zbog subjektivnosti, različiti testeri će napraviti različite ekvivalentne particije.

DIZAJNIRANJE SLUČAJEVA TESTIRANJA ZA PARTICIJE

Kada se razvijaju testni slučajevi, odgovarajući ulazi i izlazi se variraju da provere particiju.

Slučajevi testiranja se dizajniraju da bi proverili particije. Slučaj testiranja uključuje:

- ulaze u komponentu,
- particije,
- očekivani ishod testnog slučaja.

Dostupna su dva pristupa za razvoj testnih slučajeva za proveru particija:

1. ***Odvjeni test slučajevi su generisani za svaku particiju, na bazi jedan-na-jedan.***
2. ***Minimalni skup testnih slučajeva generisan je da pokrije sve particije. Ista particija može biti ponovljena u različitim testnim slučajevima.***

Kada se razvijaju testni slučajevi, odgovarajući ulazi i izlazi se variraju da provere particiju. Druge ulazne i izlazne vrednosti, koje nisu povezane sa particijom koja se proverava, su podešene na proizvoljnu vrednost. .

JEDAN-NA-JEDAN TESTNI SLUČAJEVI ZA PARTICIJE

Testnih slučajeva jedan-na-jedan za particijeza na primeru ocena na ispitu

Testni slučajevi, za ocene na ispitu u ulaznim particijama su dati na Slici 5 .

Testni slučaj	1	2	3
Ulaz (ocena na ispitu)	44	-10	93
Ulaz (ocena za rad na kursu)	15	15	15
Ukupna ocena (izračunato)	59	5	108
Testiranje particija (za ocenu na ispitu)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Očekivani izlaz	'B'	'FM'	'FM'

Slika 3.5 Testni slučajevi, za ocene na ispitu u ulaznim particijama [Izvor: NM SE321-2020/2021.]

Vrednost od 15 se koristi kao ulaz za ocenu rada na kursu. Testni slučajevi za ocenu rada na kursu u ulaznim particijama su dati an Slici 6 .

Slika 3.6 Testni slučajevi za ocenu rada na kursu u ulaznim particijama [Izvor: NM SE321-2020/2021.]

Vrednost od 40 se koristi kao ulaz za ocenu na ispitu. Testni slučajevi, za druge neispravne vrednosti u ulaznim particijama su dati9 na slici 7 :

Slika 3.7 Testni slučajevi, za druge neispravne vrednosti u ulaznim particijama [Izvor: NM SE321-2020/2021.]

Testni slučajevi, za ispravne izlazne vrednosti u particijama su na slici 8:

Slika 3.8 Testni slučajevi, za ispravne izlazne vrednosti u particijama [Izvor: NM SE321-2020/2021.]

Ulagne vrednosti, za ocenu na ispitu i ocene rad na kursu izvedene su iz ukupne ocene su na slici 9.

Slika 3.9 Testni slučajevi, ulagne vrednosti, za ocenu na ispitu i ocene rada na kursu [Izvor: NM SE321-2020/2021.]

Konačno, razmatraju se neispravni izlazi na slici 10:

Slika 3.10 Testni slučajevi, razmatranje neispravnih izlaza [Izvor: NM SE321-2020/2021.]

MINIMALNI SKUP TESTNIH SLUČAJEVA ZA VIŠE PARTICIJA

Za proveru više particija u isto vreme, moguće je razviti jedan skup test slučajeva.

U mnogim slučajevima, gornji test slučajevi su slični, ali su namenjeni različitim ekvivalentnim particijama. Za proveru više particija u isto vreme, moguće je razviti jedan skup test slučajeva. Ovaj pristup omogućava da tester smanji broj testnih slučajeva, potrebnih za pokrivanje svih ekvivalentnih particija.

Kao primer, razmotrimo sledeći testni slučaj:

Testni slučaj	1
Ulaz (ocena na ispitu)	60
Ulaz (ocena za rad na kursu)	20
Ukupna ocena (izračunato)	80
Očekivani izlaz	'A'

Slika 3.11 Primeri za testni slučaj [Izvor: NM SE321-2020/2021.]

Gornji testni slučaj proverava tri particije:

1. $0 \leq \text{ocena na ispitu} \leq 75$
2. $0 \leq \text{ocena za rad na kursu} \leq 25$
3. ukupna ocena = 'A' : $70 \leq \text{ocena na ispitu} + \text{ocena za rad na kursu} \leq 100$

Slično, testni slučajevi mogu biti kreirani za proveru neispravnih vrednosti više particija:

Testni slučaj	2
Ulaz (ocena na ispitu)	-10
Ulaz (ocena za rad na kursu)	-15
Ukupna ocena (izračunato)	-25
Očekivani izlaz	'FM'

Slika 3.12 Testni slučajevi, za proveru neispravnih vrednosti [Izvor: NM SE321-2020/2021.]

Gornji testni slučaj proverava druge tri particije:

1. ocena na ispitu < 0
2. ocena za rad na kursu < 0
3. ukupna ocena = 'FM' : ocena na ispitu + ocena za rad na kursu < 0

NEDOSTATAK PRISTUPA DIZAJNIRANJA TESTNIH SLUČAJEVA

Nedostatak pristupa jedan-na-jedan je što zahteva više testnih slučajeva dok se nedostatak minimalističkog pristupa gleda u teškoći određivanja uzroka pri dešavanju otkaza.

Poređenje jedan-na-jedan i minimalističkog pristupa

Nedostatak pristupa jedan-na-jedan je što zahteva više testnih slučajeva. Međutim, za identifikovanje particija potrebno je više vremena nego za generisanje i izvršavanje testnih slučajeva. Ušteda u smanjenju broja testnih slučajeva relativno je mala u poređenju sa troškom primene tehnike za predlaganje particija.

Nedostatak minimalističkog pristupa se ogleda u teškoći određivanja uzroka pri dešavanju otkaza. Razlog je što se više različitih particija izvršava istovremeno. Problem nastaje više zbog težeg dibagovanja, nego li zbog uticaja na proces testiranja.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

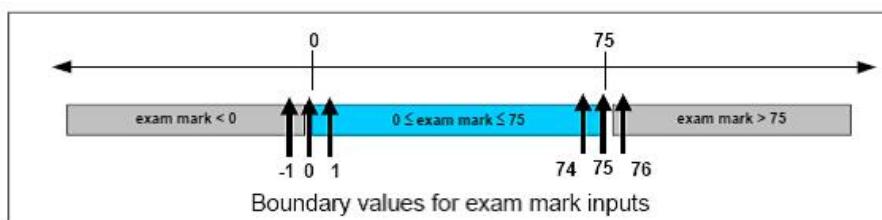
▼ Poglavlje 4

Analiza graničnih vrednosti

TESTIRANJE VREDNOSTI U ODREĐENIM GRANICAMA

Za ocene na ispitu i kursu, u primeru sa ekvivalentnim particijama, razmatrali smo realne brojeve i alfabetske vrednosti za particije. Sada se particije uvećavaju graničnim vrednostima

Za ocene na ispitu i kursu, u primeru sa ekvivalentnim particijama, razmatrali smo realne brojeve i alfabetske vrednosti za particije. Sada se u primeru koji je prethodno korišćen, particije za ocenu na ispit uvećavaju graničnim vrednostima ocena na ispit: -1, 0, 1, 74, 75 i 76 (Slika 1) .



Slika 4.1 Granica vrednosti za ispitivanje ulaznih ocena [Izvor: NM SE321-2020/2021.]

Testni slučajevi, zasnovani na graničnim vrednostima za ocenu na ispitu, su:

Slika 4.2 Testni slučajevi, zasnovani na graničnim vrednostima za ocenu na ispitu [Izvor: NM SE321-2020/2021.]

Vrednost 15 se koristi kao ulaz za ocenu za rad na kursu. Slično, ocene za rad na kursu sa graničnim vrednostima od 0 i 25, povećavaju testne slučajeve (Slika 3) .

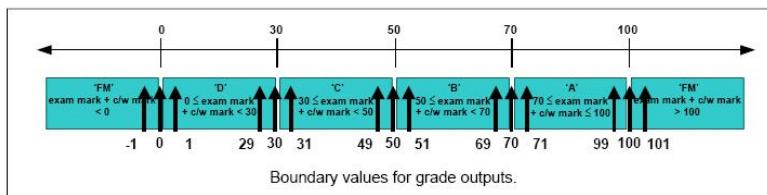
Testni slučaj	7	8	9	10	11	12
Ulaz (ocena na ispitu)	40	40	40	40	40	40
Ulaz (ocena za rad na kursu)	-1	0	1	24	25	26
Ukupna ocena (izračunato)	39	40	41	64	65	66
Granična vrednost (ocena za rad na kursu)		0			25	
Očekivani izlaz	'FM'	'C'	'C'	'B'	'B'	'FM'

Slika 4.3 Testni slučajevi, ocene za rad na kursu sa graničnim vrednostima od 0 i 25 [Izvor: NM SE321-2020/2021.]

Vrednost od 15 koristi se kao ulaz za ocenu na ispitu. Za ocene na ispitu i kursu, u primeru sa ekvivalentnim particijama, razmatrali smo realne brojeve i alfabetske vrednosti za particije.

Čak i ako su ovo validne ekvivalentne particije, one nemaju identifikovane granice, i zbog toga, granične vrednosti ne mogu biti razmatrane za pravljenje testnih slučajeva.

Razmatrane su i ekvivalentne particije za ukupnu ocenu. Granice ukupne ocene su 0, 30, 50, 70 i 100:



Slika 4.4 Granica vrednosti za izlazni kvalitet [Izvor: NM SE321-2020/2021.]

TESTNI SLUČAJEVI, ZASNOVANI NA GRANIČNIM VREDNOSTIMA ZA UKUPNU OCENU

Za sve particije može biti pretpostavljeno da su povezane sa tipom podataka, koji je korišćen za ulaz ili izlaz.

Testni slučajevi, zasnovani na graničnim vrednostima za ukupnu ocenu, su:

Testni slučaj	13	14	15	16	17	18
Ulaz (ocena na ispitu)	-1	0	0	29	15	6
Ulaz (ocena za rad na kursu)	0	0	1	0	15	25
Ukupna ocena (izračunato)	-1	0	1	29	30	31
Granična vrednost (ukupna ocena)	0				30	
Očekivani izlaz	'FM'	'D'	'D'	'D'	'C'	'C'

Slika 4.5 Testni slučajevi, zasnovani na graničnim vrednostima za ukupnu ocenu [Izvor: NM SE321-2020/2021.]

Slika 4.6 Testni slučajevi, zasnovani na graničnim vrednostima za ukupnu ocenu - nastavak [Izvor: NM SE321-2020/2021.]

Slika 4.7 Testni slučajevi, zasnovani na graničnim vrednostima za ukupnu ocenu - konacna [Izvor: NM SE321-2020/2021.]

Vidi se da su mnoge, od identifikovanih particija, ograničene samo s jedne strane, tj.

- **ocena na ispitu > 75**
- **ocena na ispitu < 0**
- **ocena za rad na kursu > 25**
- **ocena za rad na kursu < 0**
- **ocena na ispitu + ocena za rad na kursu > 100**
- **ocena na ispitu + ocena za rad na kursu < 0**

Za ove **particije** može biti pretpostavljeno da su povezane sa tipom podataka, koji je korišćen za ulaz ili izlaz. Na primer, 16-bitni celi brojevi imaju granice sa vrednostima 32767 i -32768. Stoga, možemo predložiti drugih 18 testnih slučajeva, na primer, za ocenu na ispitu, imamo:

Testni slučaj	28	29	30	31	32	33
Ulaz (ocena na ispitu)	32766	32767	32768	-32769	-32768	-32767
Ulaz (ocena za rad na kursu)	15	15	15	15	15	15
Granična vrednost (ukupna ocena)		32767			-32768	
Očekivani izlaz	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Slika 4.8 Testni slučajevi, za ocenu rada na kursu, kao i za ukupnu ocenu [Izvor: NM SE321-2020/2021.]

Slični testni slučajevi mogu se napraviti za ocenu rada na kursu, kao i za ukupnu ocenu.

✓ Poglavlje 5

Tablice odlučivanja

TESTIRANJE NA OSNOVU TABLICA ODLUČIVANJA

Tablica odlučivanja je jednostavan i pouzdan pristup za identifikaciju scenarija testiranja za složenu poslovnu logiku.

Tablica odlučivanja je jednostavan i pouzdan pristup za identifikaciju scenarija testiranja za složenu poslovnu logiku.

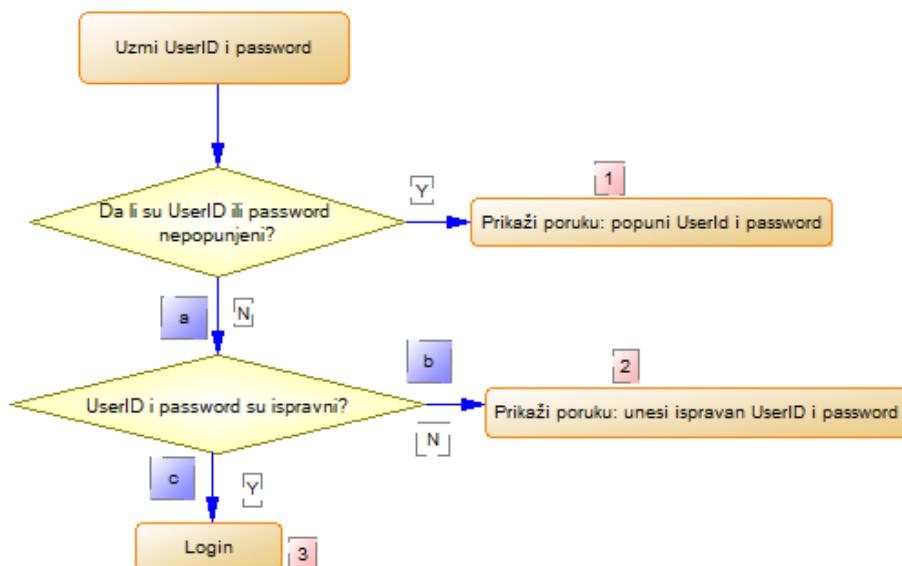
Evo ilustracije: Svi znamo da se pravilima i njihovoj validaciji koja se vezuju za razne poslovne slučajeve posvećuje veliki deo zahteva koje postavljaju kupci. Posmatrajući kako analitičari ili kupci ove zahteve predstavljaju i prenose timu projektanata, saznajemo da je većina poslovnih pravila i logike predstavljena u logičkom dijagramu toka procesa.

Logički dijagram toka složenog zahteva sastoji se od velikog broja grana, čvorova i tačaka odlučivanja. Od testera se očekuje da pokrije sve te grane i stigne do svake tačke tako složenog logičkog stabla. Tablica odlučivanja može olakšati pripremu testnih scenarija za složenu poslovnu logiku.

Primer: Pisanje slučajeva testiranja za testiranje ekrana za logovanje pomoću tehnike tabele odlučivanja:

Prvi korak je da imenujemo sve grane i listove stabla brojevima ili slovima kao što je to prikazano na slici 1:

1,2,3 su oznake listova dok su a, b, c oznake grana.



Slika 5.1 Logički dijagram toka za logovanje korisnika [Izvor: NM SE321-2020/2021.]

PREDNOSTI I OGRANIČENJA TABLICA ODLUČIVANJA

Prednost je da su stabla odlučivanja laka za razumevanje a ograničenje da se ne u njih ne mogu direktno mapirati druge tehnike kao što su

Tada možemo kreirati tablicu odlučivanja kao na slici 2.

Pri kreiranju tabela odlučivanja se treba pridržavati sledećih pravila:

1. Sve validacije navedene u tačkama odlučivanja trebaju biti unete kao kolone tabele
2. Svi rezultati (listovi) navedeni u dijagramu toka trebaju biti pokriveni tablicom odlučivanja
3. Sve kombinacije ulaznih podataka potrebnih za postizanje određenih rezultata treba navesti u koloni kombinacija i uključiti prilikom pisanja testnih slučajeva.
4. Nakon popunjavanja tablice odlučivanja, potrebno je samo proveriti da li su sve grane i listovi u logičkom stablu pokriveni.

Rbr	Grana	Čvor	UserID/Password nepotpunjeni	UserID/Password ispravni	Očekivani rezultat	Kombinacije
1	a	1	Y	N	Poruka: popuni UserID i password	Oba nepotpunjena
2	b	2	N	N	Poruka: unesи ispravan UserID i password	Oba pogrešna
3	c	3	N	Y	Login	NA

Slika 5.2 Tablica odlučivanja za slučaj logovanja korisnika [Izvor: NM SE321-2020/2021.]

Prednosti upotrebe tehnike tabele odlučivanja:

1. Ovom tehnikom se može pokriti bilo koji složeni poslovni tok predstavljen kao dijagram.
2. Nije potrebno test slučajeve preispitivati više puta kako bi bili sigurni da su dobri.
3. Tehnika je laka za razumevanje. Svako može napraviti test slučajeve na osnovu logičkog dijagrama toka.
4. Preispitivanje testova i scenarija može se potpuno izbeći, jer daje potpunu pokrivenost pri prvom kadru.

Ograničenja upotrebe tehnike tabele odluke:

1. Određene tehnike pripreme testnih slučajeva kao što su analiza granične vrednosti, podjela za ekvivalentne klase se ne mogu se direktno smestiti u tablicu odlučivanja, ali se mogu zabeležiti kao kolone sa kombinacijama i koristiti ih za pisanje testova.

✓ Poglavlje 6

Tehnika prelaska stanja

KADA SE KORISTI TEHNIKA PRELASKA STANJA

Koristi se za snimanje ponašanja softverske aplikacije kada se istim funkcijama daju različite ulazne vrednosti.

Opšte značenje prelaska stanja se može poistovetiti sa različitim oblicima iste situacije, a prema značenju, metoda prelaska stanja čini isto. Koristi se za snimanje ponašanja softverske aplikacije kada se istim funkcijama daju različite ulazne vrednosti.

Svi koristimo bankomate: kada podignemo novac s njega, prikazuju se detalji računa. Kada ponovo uradimo drugu transakciju, ponovo će se prikazati detalji računa. Detalji prikazani nakon druge transakcije se razlikuju od detalja prikazanih nakon prve transakcije, ali se oba detalja prikazuju korišćenjem iste funkcije bankomata. Dakle, ovde se koristi ista funkcija, ali je svaki put izlaz bio drugačiji, što se naziva prelazom stanja. U slučaju testiranja softverske aplikacije, ova metoda testira da li funkcija sledi specifikacije prelaska stanja prilikom unošenja različitih ulaza.

Ovo se može primeniti npr. na one vrste aplikacija koje omogućavaju određeni broj pokušaja pristupa aplikaciji, poput funkcije za prijavu u aplikaciju koja se zaključava nakon navedenog broja pogrešnih pokušaja. Pogledajmo detaljno, funkciji za logovanje koju koristimo u e-mailu. Nakon prelaska maksimalnog broja pokušaja pristupa aplikaciji unošenjem e-maila i lozinke, dobijamo poruku o grešci da je e-mail zaključan.

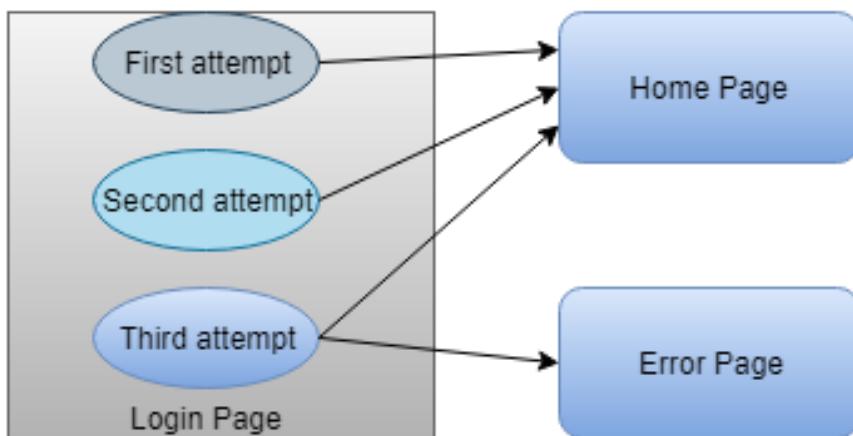
The screenshot shows a user interface for logging in. At the top, it says "Login Here". Below that, there are two input fields: one for "Email" and one for "Password". The "Email" field contains the placeholder "Enter Your Email". The "Password" field contains the placeholder "10 characters". At the bottom, there are two buttons: a blue "Back" button on the left and a blue "Login" button on the right.

Slika 6.1 Funkciji za logovanje koju koristimo u e-mailu [Izvor: NM SE321-2020/2021.]

TABELA PRELASKA STANJA - PRIMER

Tabela prelaska stanja za funkciju prijave na aplikaciju koja pruža najviše tri broja pokušaja

Pogledajmo to na dijagramu: Postoji funkcija prijave za aplikaciju koja pruža najviše tri broja pokušaja, a nakon prekoračenja tri pokušaja biće preusmerena na stranicu sa greškama.



Slika 6.2 Prijave za aplikaciju koja pruža najviše tri broja pokušaja [Izvor: NM SE321-2020/2021.]

U tabeli prelaska stanja na slici 3. vidimo da stanje S1 označava prvi pokušaj prijave. Kada je prvi pokušaj nevažeći, korisnik će biti usmeren na drugi pokušaj (stanje S2). Ako je i drugi pokušaj nevažeći, korisnik će biti usmeren na treći pokušaj (stanje S3). Ako je treći i poslednji pokušaj nevažeći, korisnik će biti usmeren na stranicu sa greškama (stanje S5). Ali ako je treći pokušaj validan, biće usmeren na početnu stranicu (stanje S4).

Slika 6.3 Tabela prelaska stanja ako je treći pokušaj ne validan [Izvor: NM SE321-2020/2021.]

Pogledajmo tabelu prelaska stanja ako je treći pokušaj validan (slika 4):

Slika 6.4 Tabelu prelaska stanja ako je treći pokušaj validan [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 7

Grupna vežba: Tehnike crne kutije - 1 deo

ŠTA JE TEHNIKA ANALIZE GRANIČNIH VREDNOSTI

Analiza graničnih vrednosti proširuje ekvivalentno parcelisanje uključenjem vrednosti oko granica particija.

Analiza graničnih vrednosti je standardizovana tehnika koja je poznata i opisana u u dатој literaturi. Analiza graničnih vrednosti proširuje ekvivalentno parcelisanje uključenjem vrednosti oko granica particija. Kao i sa ekvivalentnim particionisanjem, prepostavljamo da softver ili njegove komponente, skup vrednosti tretiraju slično. Međutim, projektanti su skloni da prave greške u tretiranju vrednosti na granicama particija. Na primer, elementi liste mogu biti obrađivane slično, i mogu se grupisati u jednu ekvivalentnu particiju. Međutim, u obradi elemenata, developer možda ne obrađuje ispravno ili prvi ili poslednji element liste.

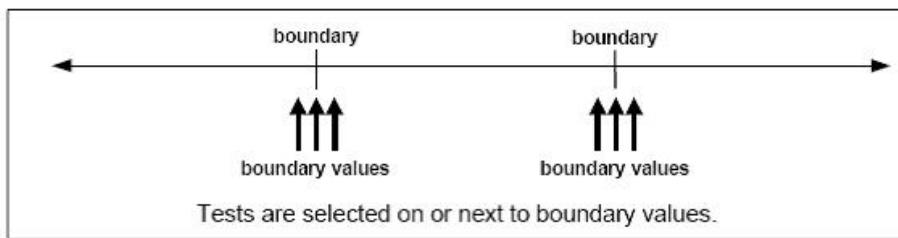
Granične vrednosti su uobičajeno granice ekvivalentnih klasa. Primeri uključuju (Slika 1) :

- ponedeljak i subotu za vikende
- januar i decembar za mesece u godini
- 32767 i -32768 za 16-bitne cele brojeve
- gornju levu i donju desnu poziciju kursora na ekranu
- prvu i poslednju liniju štampanog izveštaja
- 1. januar 2000. godine za obradu godine sa dva broja
- string od jednog karaktera i maksimalnu veličinu stringa

Testni slučajevi su odabrani da provere vrednosti sa obe strane granice.

Vrednosti na granicama biraju se kao uvećano rastojanje od granice, koje je najmanje značajna vrednost za posmatrani tip podataka (na primer, uvećanje od 1 za cele brojeve, \$0.01 za dolare).

Primer : Razmotrimo opet generate grading funkciju, prethodno korišćenu u primeru za tehniku ekvivalentnih particija. Ovaj primer je izведен iz pomenutog gore navedenog primera. Inicijalno, ekvivalentne particije su identifikovane (ovo je isto kao u prethodnoj tehniци), i koriste se za predlog graničnih vrednosti.



Slika 7.1 Testiranje vrednosti u određenim granicama [Izvor: NM SE321-2020/2021.]

Vreme trajanja ove grupne vežbe je 20 minuta.

EKVIVALENTNO PARCELISANJE - POKAZNI PRIMER

Ovom tehnikom se ulazni podaci dele u različite klase podataka ekvivalencije. Tehnika se možete primeniti tamo gde postoji opseg u polju za unos

Ekvivalentno parcelisanje je tehnika crne kutije (kod nije vidljiv testeru) koja se može primeniti na sve nivoje ispitivanja poput jedinice, integracije, sistema, itd. U ovoj tehnici skup uslova testiranja treba podeliti na particije koje se može smatrati istim. Ovom tehnikom se ulazni podaci dele u različite klase podataka ekvivalencije. Tehnika se možete primeniti tamo gde postoji opseg u polju za unos

Primer 1: Razmotrimo ponašanje tekstualnog polja za naručivanje pica prikazanog na slici 2.

Order Pizza:

Submit

Slika 7.2 Tekstualno polje za naručivanje pica [Izvor: NM SE321-2020/2021.]

1. Vrednosti za količinu pica od 1 do 10 smatraju se važećim. Prikazuje se poruka o uspešnom naručivanju
2. Vrednosti od 11 do 99 se smatraju nevažećim za narudžbinu i pojaviće se poruka o grešci, "Samo 10 pica može se naručiti"

Uslovi za testiranje su:

- 1. Bilo koji broj veći od 10 unet u polje Naruči picu (recimo 11) smatra se nevažećim.
- 2. Bilo koji broj manji od 1 koji je 0 ili manji, smatra se nevažećim.
- 3. Brojevi od 1 do 10 smatraju se važećim
- 4. Bilo koji broj od 3 cifre npr. 100 se smatra nevažećim.

Ne možemo testirati sve moguće vrednosti jer ako to učinimo, broj test slučajeva će biti veći od 100. Da bismo rešili ovaj problem, koristimo hipotezu o podeli na klase ekvivalencije gde delimo moguće vrednosti na grupe ili skupove, kao što je prikazano na slici 3. za koje se ponašanje sistema može smatrati istim



Slika 7.3 Podeli na klase ekvivalencije za koje se ponašanje sistema može smatrati istim [Izvor: NM SE321-2020/2021.]

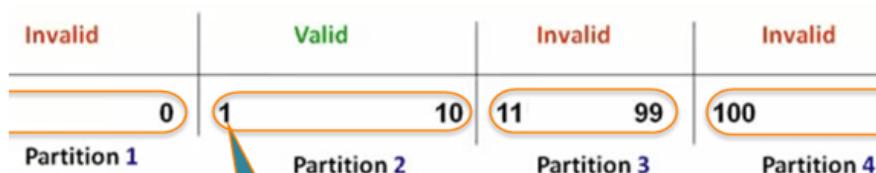
Slika 7.4 Izbor jedne vrednosti iz svake particije [Izvor: NM SE321-2020/2021.]

Zatim biramo samo jednu vrednost iz svake particije za testiranje. Hipoteza ove tehnike je da ako jedan uslov / vrednost u particiji prođe svi ostali će takođe proći. Isto tako, ako jedan uslov u particiji ne uspe, svi ostali uslovi u toj particiji neće uspeti.(slika 4)

ANALIZI GRANIČNIH VREDNOSTI

U analizi graničnih vrednosti testiraju se granice između ekvivalentnih particija

U analizi graničnih vrednosti testiraju se granice između ekvivalentnih particija. (slika 5)



Slika 7.5 Granične vrednosti između particija [Izvor: NM SE321-2020/2021.]

U našem prethodnom primeru, umesto provere po jedne vrednost za svaku particiju proverićemo vrednosti na particijama poput 0, 1, 10, 11 i tako dalje. Kao što možete primetiti, testiraju se vrednosti i na važećim i na nevažećim granicama. Analiza granične vrednosti se takođe naziva i provera opsega.

Ekvivalentno parcelisanje i analiza granične vrednosti su usko povezani i mogu se koristiti zajedno na svim nivoima testiranja.

Primer 2: Polje za unos lozinke na slici 6. prihvata najmanje 6 znakova i maksimalno 10 znakova To znači da bi rezultati za particije 0-5, 6-10, 11-14 trebali biti jednaki

Enter Password:

Пуштаји

Slika 7.6 Polje za unos lozinke [Izvor: NM SE321-2020/2021.]

- *Test scenario 1:* Uneti 0 do 5 karaktera u polje pasvorda: Očekivani izlaz: sistem ne prihvata

- *Test scenario 2:* Uneti 6 do 10 karaktera u polje pasvorda: Očekivani izlaz: sistem prihvata
- *Test scenario 3:* Uneti 11 do 14 karaktera u polje pasvorda: Očekivani izlaz: sistem ne prihvata

Primer 3: Ulazno polje treba da prihvati vrednosti 1 do 10:

Test scenario 1: Granična vrednost: 0 Očekivani izlaz: sistem ne prihvata

Test scenario 2: Granična vrednost: 1 Očekivani izlaz: sistem prihvata

Test scenario 3: Granična vrednost: 2 Očekivani izlaz: sistem prihvata

Test scenario 4: Granična vrednost: 9 Očekivani izlaz: sistem prihvata

Test scenario 5: Granična vrednost: 10 Očekivani izlaz: sistem prihvata

Test scenario 6: Granična vrednost: 11 Očekivani izlaz: sistem ne prihvata

TABLICE ODLUČIVANJA

Da bismo dizajnirali test slučajeve tehnikom tablice odlučivanja, moramo razmotriti uslove kao ulaz i akcije kao izlaz.

Tehnika tablica odlučivanja je jedna od najčešće korišćenih tehnik dizajniranja slučajeva testiranja crne kutije. Ovo je sistematski pristup gde se različite kombinacije ulaza i odgovarajuće ponašanje sistema beleže u tabelarnom obliku. Zato je ova tehnika poznata i kao tablica uzroka-posledica.

Tablice odlučivanja se koriste za sistematično odabiranje test slučajeva; omogućavaju uštedu vremena testiranja i daju dobru pokrivenost područja ispitivanja softverske aplikacije. Tehnika tablica odlučivanja je pogodna za funkcije kod kojih postoji logička povezanost između dva i više od dva ulaza.

Ova tehnika omogućava ispravnu kombinaciju ulaza i dobijanje određenih rezultata za različite kombinacije unosa. Da bismo dizajnirali test slučajeve tehnikom tablice odlučivanja, moramo razmotriti uslove kao ulaz i akcije kao izlaz.

Pokažimo to na primeru: Većina nas koristi nalog e-maila, a kada želite da ga iskoristite, za to je potrebno da unesete e-mail adresu i pripadajuću lozinku.

Ako se i e-mail i lozinka podudaraju, korisnik će biti usmeren na početnu

stranicu naloga e-maila; u suprotnom će se vratiti na stranicu za prijavu sa porukom o grešci koja je naznačena sa „Netačna e-mail adresa“ ili „Netačna lozinka“. Kako izgleda tablica odlučivanja za funkciju prijave pomoću e-maila i lozinke. I e-mail i lozinka su uslovi, a očekivani rezultat je akcija.

Email (condition1)	T	T	F	F
Password (condition2)	T	F	T	F
Expected Result (Action)	Account Page	Incorrect password	Incorrect email	Incorrect email

Slika 7.7 Tablica odlučivanja za funkciju prijave pomoću e-maila i lozinke. [Izvor: NM SE321-2020/2021.]

U tabeli su četiri stanja ili test slučajevi za testiranje funkcije prijave:

1. U prvom stanju, ako su i e-mail i lozinka tačni, korisnika treba uputiti na početnu stranicu naloga.
2. U drugom stanju ako je adresa e-maila tačna, ali lozinka nije tačna, tada bi funkcija trebalo da prikaže: Netačna lozinku.
3. U trećem stanju ako je adresa e-pošte pogrešna, ali je lozinka tačna, tada bi trebalo da se prikaže: Netačan e-mail.
4. U četvrtom i poslednjem stanju, i adresa e-pošte i lozinka nisu tačni, onda bi funkcija trebalo da prikaže: Netačan e-mail.

TABLICE ODLUČIVANJA-FORMULA ZA PRONALAŽENJE SVIH MOGUĆIH USLOVA

Koristeći tehniku tablica odlučivanja, tester određuje očekivani izlaz. Ako funkcija daje očekivani izlaz, testiranje je uspešno a ako ne, onda nije uspešno.

U ovom primeru su uključeni svi mogući uslovi ili test slučajevi, a na isti način tim za testiranje uključuje i sve moguće slučajeve testiranja kako bi se greške mogle ispraviti na nivou testiranja.

Da bi pronašao broj svih mogućih uslova, tester koristi 2^n formulu gde n označava broj ulaza; u primeru je broj ulaza 2 (jedan je tačan a drugi je lažan).

Broj mogućih uslova = $2^{\text{Broj vrednosti drugog uslova}}$

Broj mogućih uslova = $2^2 = 4$

Koristeći tehniku tablica odlučivanja, tester određuje očekivani izlaz. Ako funkcija daje očekivani izlaz, testiranje je uspešno a ako ne, onda nije uspešno. Softver kog kojeg je testiranje neuspešno se vraća razvojnom timu da ispravi kvar.

✓ Poglavlje 8

Grupna vežba: Testiranje ATM-a metodom crne kutije

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 1

Prvi test slučaj odnosi se na unos korisničkih kredencijala prilikom pristupa ATM uređaju.

Test slučaj simulira pristupanje ATM uređaju koristeći podatke o kreditnoj kartici (lozinka) i unosom kartice u bankomat. U tabeli 1 su prikazani test rezultati i postavka testiranja.

Test dopuni tako što ćeš prilikom testiranja unosa korisničkih kredencijala primeni tehniku tablica odlučivanja.

Vreme traje ove grupne vežbe je 25 minuta.

Naslov	Unos korisničkih kredencijala u ATM	Rev	1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provjeri rada glavne funkcionalnosti ATM uređaja	Reference					
Test uslovi		Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za Izvršenje test slučaja		5min	

	Opis postavke za testiranje
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabratи svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik treba da unese korisničke podatke odabir kartice i unos korisničke lozinke za unetu karticu

	Definicija testa		Izvršenje testa	
	Ulazni podaci	Uslo	Ulazni podaci	Aktuelni rezultati
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup		Broj problema

	Opis postuslova
•	Korisnik dobija informaciju da je uspešno uneo podatke
•	Korisnik može da pristupi sledećim funkcijama ATM uređaja

Slika 8.1.1 Test-unos korisničkih kredencijala u ATM [Izvor: NM SE321-2020/2021.]

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 2

Testiranje stanja na računu ATM uređaja izvedeno je korišćenjem opisanog test slučaja.

Drugi test slučaj odnosi se na proveravanje stanja na računu ATM uređaja. Potrebno je da korisnik unese svoje korisničke podatke a nakon toga opcijom na ATM uređaju proveri stanje koje se nalazi na njegovom računu. Takođe se koristi tehnika crne kutije gde testera ne zanimaju procesi u ATM uređaju već samo izlaz iz ATM-a. U tabeli 2 su prikazani test rezultati i postavka testiranja.

Test dopuni tako što ćeš za testiranje unosa odgovarajućeg vrednosti računa korisnika primeniti tehniku ekvivalentnog parcelisanja pod pretpostavkom da računi moraju biti u opsegu vrednosti od 10.000 do 70.000

Naslov	Proveravanje stanja računa u ATM		Rev	1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provjerava rada druge glavne funkcionalnosti ATM uređaja					Reference		
Test uslovi		Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za izvršenje test slučaja		5min		

Opis postavke za testiranje	
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabrati svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik treba da odabere opciju za pregled stanja na računu i proveri svoj račun

Definicija testa			Izvršenje testa	
	Ulazni podaci	Uslovi	Ulazni podaci	Aktuelni rezultati
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup,	Odabir pregleda stanja na računu	

Opis postuslova	
•	Korisnik dobija informaciju o stanju na svom računu
•	Korisnik može da proveri podatke
	Korisnik može da odabere drugu vrstu provere stanja

Slika 8.1.2 Test- proveravanje stanja računa u ATM [Izvor: NM SE321-2020/2021.]

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 3

Testiranje podizanja novca sa ATM uređaja izvedeno je korišćenjem opisanog test slučaja.

Test podizanja novca sa ATM uređaja podrazumeva unos svih podataka sa korisničke strane i provera izlazne informacije a to je u ovom test slučaju novac. Proverava se izlaz na osnovu poznatih ulaznih elemenata. U tabeli 3 su prikazani test rezultati i postavka testiranja.

Test dopuni tako što ćeš za testiranje unosa odgovarajućeg iznosa koji korisnik želi da podigne sa račina primeniti tehniku ekvivalentnog parcelisanja a zatim tehniku graničnih vrednosti pod pretpostavkom da iznosi ne mogu biti manji od 1.000 a veći od 60.000.

Naslov	Podizanje novca sa ATM uređaja	Rev	1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provjeravanje funkcionalnosti za podizanje novca ATM uređaja				Reference		
Test uslovi			Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za izvršenje test slučaja	5min	

Opis postavke za testiranje	
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabrati svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik treba da unese korisničke podatke odabir kartice i unos korisničke lozinke za unetu karticu
	Korisnik treba da unese određenu vrednost za podizanje novca

Definicija testa			Izvršenje testa		
	Ulazni podaci	Uslovi	Ulazni podaci	Aktuelni rezultati	Broj problema
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup	Cifra za podizanje novca		

Opis postuslova	
•	Korisnik dobija informaciju da je uspešno uneo podatke
•	Korisnik je dobio podatke o podignutom novcu ili grešku da nema dovoljno novca na računu

Slika 8.1.3 Test- podizanje novca sa ATM uređaja [Izvor: NM SE321-2020/2021.]

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 4

Testiranje štampanja priznanice na ATM uređaju izvedeno je korišćenjem opisanog test slučaja.

Štampanje priznanice je funkcija koja se pojavljuje nakon podizanja novca i omogućava da korisnik ima potvrdu o podignutom novcu. Test se odvija u realnom okruženju korišćenjem tehnike crne kutije. U tabeli 4 su prikazani test rezultati i postavka testiranja.

Test izvrši primenom tehnike prelaska stanja.

Naslov	Štampanje priznanica o podignutom novcu	Rev	1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provera štampanja priznanice o podignutom novcu		Reference				
Test uslovi		Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za Izvršenje test slučaja	5min		

Opis postavke za testiranje	
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabratи svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik nakon podizanja novca ima mogućnost da odabere štampanje priznanice

Definicija testa			Izvršenje testa	
	Ulazni podaci	Uslo	Aktuelni rezultati	Broj problema
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup,	Podignut novac sa ATM uređaja	

Opis postuslova	
•	Korisnik dobija informaciju da je uspešno izvršena transakcija
•	Korisnik može da odabere štampanje priznanice
•	Korisnik dobija štampanu priznаницу из ATM uređaja

Slika 8.1.4 Test- štampanje priznanice o podignutom novcu [Izvor: NM SE321-2020/2021.]

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 5

Testiranje provere kursa na ATM uređaju izvedeno je korišćenjem opisanog test slučaja.

Provera kursa je opcija koja se nalazi na ATM uređaju i omogućava brzu proveru. Test se odvija u realnim uslovima, rezultati se beleže korišćenjem računara a test obavljaju test inženjeri. U tabeli 5 su prikazani test rezultati i postavka testiranja.

Test provere kursa izvrši primenom tehnike za koju smatraš da je najadekvatnija.

Naslov	Provera kursa na ATM uređaju		Rev	1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provera rada kursa na bankomatu		Reference					
Test uslovi		Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za izvršenje test slučaja		5min		

Opis postavke za testiranje	
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabratи svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik treba da unese korisničke podatke odabir kartice i odabir opcije za proveru kursa

Definicija testa			Izvršenje testa	
	Ulazni podaci	Uslo	Ulazni podaci	Aktuelni rezultati
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup	Odabir opcije za proveru kursa	

Opis postuslova	
•	Korisnik dobija informaciju o kursu na taj dan u banci koja je vlasnik ATM uređaja
•	Korisnik, ukoliko nije postavljen kurs za taj dan, dobija informaciju da nije moguće proveriti kurs

Slika 8.1.5 Test- provera kursa na ATM uređaju [Izvor: NM SE321-2020/2021.]

TESTIRANJE ATM UREĐAJA TEHNIKOM CRNE KUTIJE - TEST SLUČAJ 6

Testiranje odabira jezika ATM uređaja izvedeno je korišćenjem opisanog test slučaja.

Korisnički interfejs omogućava odabir jezika interfejsa ATM uređaja prilikom početne iteracije. Testiranje se odvija tehnikom crne kutije u realnim uslovima koji podrazumevaju uslove u kojima će korisnik koristiti ATM uređaj. U tabeli 6 su prikazani test rezultati i postavka testiranja.

Test odabira jezika izvrši primenom tehnike za koju smatraš da je najadekvatnija.

Naslov	Odabir jezika na ATM uređaju	Rev 1	Autor	Nebojša Gavrilović	Datum	01.11.2015
Cilj	Cilj je provera rada promene jezika interfejsa na ATM uređaju		Reference			
Test uslovi		Vreme neophodno za izradu test slučaja	2min	Neophodno vreme za Izvršenje test slučaja	5min	

Opis postavke za testiranje	
•	Potrebno je pristupiti ATM uređaju putem korisničkog interfejsa, odabrati svrhu korišćenja uređaja i pristupiti unosu podataka
•	ATM uređaj je startovan
•	Korisnik treba da odabere jezik na kome će biti interfejs u toku transakcije

Definicija testa			Izvršenje testa	
	Ulagani podaci	Uslovi	Ulagani podaci	Aktuelni rezultati
•	Korisnička kartica	Posedovanje kartice, posedovanje validne lozinke za pristup	Odabir jezika	

Opis postuslova	
•	Korisnik je dobio interfejs na odabranom jeziku
•	Korisnik može da se vrati i odabere drugi jezik na ATM uređaju

Slika 8.1.6 Test- odabir jezika interfejsa u ATM [Izvor: NM SE321-2020/2021.]

✓ 8.1 Vežba za samostalni rad

FUNKCIONALNO TESTIRANJE (TEHNIKE CRNE KUTIJE)

Na osnovu obrađenih tehnika testiranja softvera uraditi testiranje ISUM sistema

Svaki student treba da na primeru poznatog sistema ISUM identificuje i razvije scenarije testiranja i na osnovu scenarija izvrši testiranje aplikacije korišćenjem navedenih tehnika crne kutije.

Zadatak:

Odabrati deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

Definisati četiri scenarija testiranja za odabrani deo sistema:

1. Jedan scenario testiranja izvršiti primenom tehnike Klase ekvivalencije (**vreme izrade zadatka 20 minuta**)
2. Jedan scenario testiranja izvršiti primenom tehnike graničnih vrednosti (**vreme izrade zadatka 20 minuta**)
3. Jedan scenario testiranja izvršiti primenom tehnike prelaska stanja (**vreme izrade zadatka 25 minuta**)

4. Jedan scenario testiranja izvršiti primenom tehnike tablica odlučivanja (**vreme izrade zadatka 25 minuta**)

Dokumentovati izvršene test slučajeve i napraviti tabelu po uzoru na tabelu datu u vežbama.

✓ Poglavlje 9

Domaći zadatak

PETI DOMAĆI ZADATAK

Nakon pete lekcije potrebno je uraditi peti domaći zadatak

Za odabranu aplikaciju koju ste radili na nekom od predmeta koje ste prethodno slušali i položili primeniti sledeće:

1. Testirati aplikaciju korišćenjem dve od navedene četiri tehnike black box tehnike za koje smatrate da su za vas najkorisnije.
2. Dokumentovati izvršene slučajeve testiranja

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ05-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ05.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Testiranje softvera -tehnika crne kutije (funkcionalno testiranje) i Test Oracle mehanizmi

TEHNIKA CRNE KUTIJE (FUNKCIONALNO TESTIRANJE)

U praksi testiranja softvera najčešće čovek-ekspert ima ulogu referentnog (ispravnog, etalonskog) SUT i donosi odluku o uspehu testa

Obrađeno je testiranje zasnovano na sistemskim zahtevima ili testiranje komponenti metodom crne kutije. Veoma često se izvodi na osnovu dokumenata funkcionalnih zahteva, pri čemu se svaki funkcionalni zahtev prevodi u kriterijum funkcionalnog testa.

Testna pokrivenost, pri funkcionalnom testiranju, obično se postiže izradom matrice tragova, koja obuhvata zahteve i testne kriterijume za testiranje tih zahteva. Nepokriveni zahtevi su zahtevi koji nemaju zadovoljavajuće, pridružene testne kriterijume.

Testiranje metodom crne kutije tipično se izvršava kao životni ciklus razvoja skoro kompletno integrisanog sistema, tokom integracionog testiranja, testiranja interfejsa, sistemskog testiranja i testiranja prihvatljivosti. Na tom nivou, komponente sistema su dovoljno integrisane da li su kompletni zahtevi ispunjeni skoro u potpunosti.

LITERATURA ZA LEKCIJU 05

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>) .
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>) .
3. dr Miladin Stefanović, mr Slobodan Mitrović, mr Milan Erić, MODEL KVALITETA SOFTVERA, Festival kvaliteta 2006., 33. Nacionalna konferencija o kvalitetu, Kragujevac, 10. ' 12. maj 2006.
4. Black Box Software Testing, <http://www.testingeducation.org/BBST/bugadvocacy/BugAdvocacy2008.pdf>

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link:
<https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Mr Boris Todorović, EVALUACIJA KVALITETA SOFTVERA PREMA ISO 9126 STANDARDU, DOI 10.7251/POS1208099T.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.testingeducation.org/BBST/>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Testiranje softvera – tehnike bele
kutije

Lekcija 06

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 06

TESTIRANJE SOFTVERA – TEHNIKE BELE KUTIJE

- ✓ Testiranje softvera – tehnike bele kutije
- ✓ Poglavlje 1: White box ili strukturno testiranje
- ✓ Poglavlje 2: Pokrivenost koda
- ✓ Poglavlje 3: Tehnika pokrivenosti naredbi
- ✓ Poglavlje 4: Tehnika pokrivenosti grana
- ✓ Poglavlje 5: Tehnika pokrivenosti odluka
- ✓ Poglavlje 6: Testiranje kontrole toka
- ✓ Poglavlje 7: Testiranje petlji i uslovnih grana
- ✓ Poglavlje 8: White box - Dizajniranje test slučajeva
- ✓ Poglavlje 9: Studija slučaja
- ✓ Poglavlje 10: Grupna vežba: Korišćenje CodeCover
- ✓ Poglavlje 11: Vežba za samostalni rad
- ✓ Poglavlje 12: Domaći zadatak
- ✓ White-box ili strukturno testiranje

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Opšti cilj testera je da proveri svaku putanju u programu. Ali, kao što je poznato, to je nemoguće. Čak i u slučaju jednostavnog programa postoji trilion različitih putanja.

White box (testiranje bele kutije) ili struktorno testiranje obuhvata ispitivanje interne strukture programa ili sistema. Testni podaci se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji.

Opšti cilj testera je da proveri svaku putanju u programu. Ali, kao što je poznato, to je nemoguće. Čak i u slučaju jednostavnog programa, postoji veliki broj različitih putanja. Zato je kod testiranja bele kutije veoma važno ispitati pokrivenost koda koja opisuje stepen testiranja izvornog koda programa i pomoću skupa test slučajeva pronađi područja programa koja se ne koriste.

U lekciji se govori o tehnikama koje se koriste prilikom testiranja pokrivenosti koda kao što su:

1. **Pokrivenost naredbi**
2. **Pokrivenost grana**
3. **Pokrivenost putanja**
4. **Pokrivenost odluka**
5. **Pokrivenost kontrole toka**

Takođe, opisane tehnike su pokazane na primeru studije slučaja u kojoj se izvršava binarno pretraživanje. Program počinje izborom vrednosti u sredini niza. Zatim, poredi odabranu vrednost sa ključem. Ako su ove dve vrednosti iste, program daje izveštaj o pronađenju ključa na središnjoj poziciji i završava se.

▼ Poglavlje 1

White box ili struktурно testiranje

METODA BELE KUTIJE – WHITE BOX

White box ili struktурно testiranje obuhvata ispitivanje interne strukture programa ili sistema bez brige o zahtevima koje treba da zadovolji.

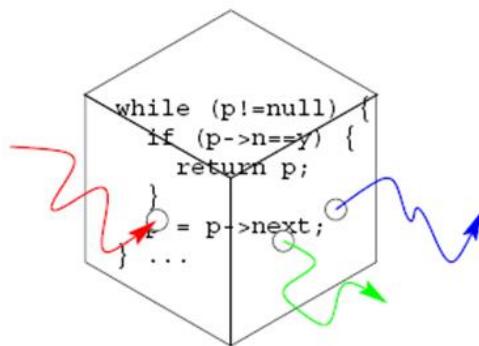
White box testiranje ili struktурно testiranje obuhvata ispitivanje interne strukture programa ili sistema. Testni podaci se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji. Tester poznaje internu strukturu i logiku programa, isto kao što mehaničar zna unutrašnji mehanizam automobila. Specifični primjeri koji spadaju u ovu kategoriju uključuju testiranje izraza, testiranje pokrivenosti grana, testiranje uslova i sl.

Prednost ovog pristupa je da je potpun i da se fokusira na proizvedeni kod. Greške i nemamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni.

Jedna od mana ovog testiranja je da ne proverava da li su specifikacije tačne, već se fokusira samo na internu logiku. Druga loša karakteristika je da ne postoji način da se otkriju elementi koji nedostaju i greške vezane za osetljive podatke. Npr. ako izraz u programu treba da se kodira kao „ako $a - b < 10$ “, ali se kodira kao „ako $(a - b) < 1$ “, nemoguće je otkriti grešku bez detalja iz specifikacije. Poslednji nedostatak je da ovaj tip testiranja ne može da obuhvati kompletну logiku programa, jer bi to podrazumevalo kreiranje jako velikog broja testova.

Ovo testiranje proverava i analizira izvorni kod i zahteva dobro poznavanje programiranja, odgovarajućeg programskega jezika, karakteristika konkretnog softverskog proizvoda. Plan testiranja se određuje na osnovu elemenata implementacije softvera, kao što su programski jezik, logika i stilovi. Testovi se izvode na osnovu strukture programa.

Kod ove metode postoji mogućnost provere skoro celokupnog koda (Slika 1), na primer proverom da li se svaka linija koda izvršava barem jednom, proverom svih funkcija ili proverom svih mogućih kombinacija različitih programskih naredbi. Specifičnim testovima može se proveravati postojanje beskonačnih petlji ili koda koji se nikada ne izvršava.



Slika 1.1 Metoda bele kutije – "White box" [Izvor: NM SE321-2020/2021.]

KO VRŠI TESTIRANJE METODOM BELE KUTIJE?

Testiranje bele kutije rade programeri, a zatim aplikaciju ili softver šalju testeru, koji će izvršiti testiranje crne kutije kako bi izvršio verifikaciju aplikacije u odnosu na zahteve sistema,

Testiranje bele kutije rade programeri, a zatim aplikaciju ili softver šalju testeru, koji će izvršiti testiranje crne kutije kako bi izvršio verifikaciju aplikacije u odnosu na zahteve sistema, identifikovao greške i poslao ih programerima na ispravku.

Programeri ispravljaju greške (engl. **bug**) ponovo testiraju kod metodom bele kutije i šalju ga testerima. Ovde, ispravka grešaka podrazumeva da je greška otklonjena i da određena funkcija aplikacije dobro funkcioniše.

Testeri se ne uključuju u ispravku grešaka jer to može usporiti testiranje ostalih funkcija. **Tim za testiranje pronalazi greške, a programeri ispravljaju greške.**

Testiranje bele kutije sadrži različite testove kao što su :

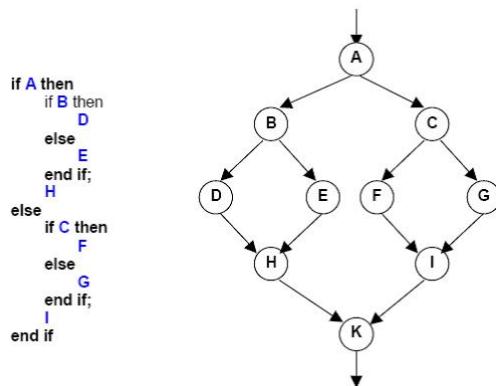
- Testiranje putanja (engl. **Path testing**)
- Testiranje petlji (engl. **Loop testing**)
- Testiranje uslova (engl. **Condition testing**)
- Testiranje iz perspektive memorije (engl. **Testing based on the memory perspective**)
- Testiranje performansi programa (engl. **Test performance of the program**)

TESTIRANJE PUTANJE - GRAF TOKA

Prilikom testiranja putanja, treba uraditi grafove toka i testirati sve nezavisne putanje.

Prilikom **testiranja putanja**, treba uraditi **grafove toka** (engl. **flow graphs**) i **testirati sve nezavisne putanje**.

Graf toka može da predstavi tok programa. Na primer, na slici 1. prikazani su programski kod i graf toka, izведен na osnovu prikazanog programskega koda.



Slika 1.2 Primer programskog kôda i grafa toka [Izvor: NM SE321-2020/2021.]

Graf toka može takođe da prikaže i kako se svaki program nadovezuje jedan na drugi kao što se vidimo na slici 2. Testiranje svih nezavisnih putanja podrazumeva da pod pretpostavkom da prvo testiramo putanju od Main () do Function G, postavimo parametre i testiramo da li je program ispravan na toj određenoj putanji, a zatim na isti način testiramo sve ostale putanje i ispravljamo eventualne greške.

Slika 1.3 Primer grafa toka u slučaju dodavanja jednog programa drugom Izvor:
<https://www.javatpoint.com/control-flow-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

Graf toka se sastoji od:

- čvorova (**nodes**), koji predstavljaju naredbe (ili pod programe), koji mogu biti uključeni tokom izvršavanja programa;
- grana (**edges**), koje predstavljaju način na koji logika programa dozvoljava da se program izvršava, prolaskom od jedne naredbe (ili pod programa) do druge. Grane su orijentisane.
- uslovnih čvorova (**branch nodes**), čvorova koji imaju više od jedne izlazne grane;
- uslovnih grana (**branch edges**), grana za izlazak iz uslova;
- putanja (**paths**), odnosno mogućih načina za kretanje granama od čvora do čvora.

TESTIRANJE PETLJI - UNIT TEST

Testiranje petlji podrazumeva testiranje petlji tipa while, for, and do-while, proveru tačnosti uslova završetka i veličinu uslova. Radi su uz korišćenje jediničnih programa za testiranje

Testiranje petlji podrazumeva testiranje petlje tipa while, for, and do-while itd., proveru tačnosti uslova završetka (za izlazak iz petlje) i veličinu uslova.

Na primer: imamo program gde je programer zadao izvršenje petlje 50.000 puta

```
{
while(50,000)
....
```

.....
}

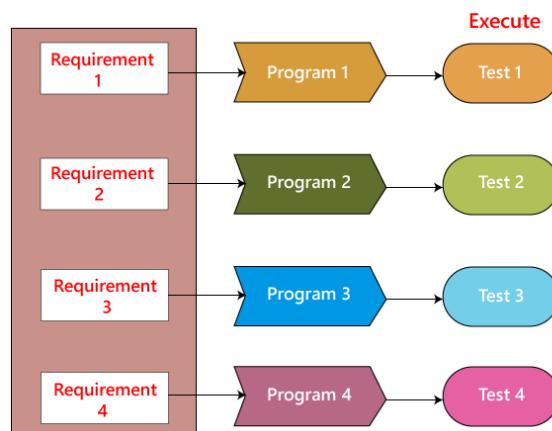
Testiranje petlje se neće vršiti tako što će se program testirati za svih 50.000 ciklusa već umesto toga programeri piše mali program P, poznat kao jedinični program za testiranje (engl. Unit test), na sličnom jeziku kao program izvornog koda koji će nam pomoći u testiranju.

```
Test P
{
.....
.... }
```

Unit testovi se mogu koristiti i u drugim slučajevima. Na primer imamo različite zahteve označene kao 1, 2, 3, 4 na osnovu kojih je programer napisao programe 1, 2, 3, 4 koji se sekvensijalno izvršavaju. Programer će obaviti testiranje bele kutije, tako što će testirati linije koda za svih pet programa da bi pronašao grešku. Ako se u nekom od programa pronađe neka grešku, ispraviće se a nakon toga se mora

opet testirati ceo sistem, pri čemu taj proces zahteva mnogo vremena i truda i usporava vreme završetka softverskog proizvoda.

Prepostavimo da imamo drugi slučaj, kada klijenti žele da izmene zahteve, pa će programer izvršiti potrebne izmene i ponovo testirati sva četiri programa, za šta je opet potrebno mnogo vremena i napora. Ovi problemi se mogu rešiti na način šematski prikazan na slici 4. Programer će napisati kodove za testiranje na srodnom jeziku kao izvorni kod a zatim pojedinačno izvršiti ove jedinični programe za testiranje bez potrebe za izvršenjem celog programa. Jedinični programi za testiranje su povezani sa glavnim programom i implementirani kao pod programi.



Slika 1.4 Primer upotrebe jediničnih programi za testiranje Izvor: <https://www.javatpoint.com/branch-coverage-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

TESTIRANJE USLOVA

U ovom slučaju se testiraju svi logički uslovi - za vrednosti za koje je uslov ispunjen (true) i za vrednosti za koje on nije ispunjen (false);

U ovom slučaju se testiraju svi logički uslovi - za vrednosti za koje je uslov ispunjen (**true**) i za vrednosti za koje on nije ispunjen (**false**); to jest, proveravaju se i uslov ako (**if**) i uslov inače (**else**).

Na primer:

```
if(condition) - true
{
    ...
    ...
    ...
}
else - false
{
    ...
    ...
    ...
}
```

Gornji program će raditi dobro za oba uslova, što znači ako je uslov tačan, a onda bi inače (**else**) trebalo biti netačno i obrnuto.

TESTIRANJE IZ PERSPEKTIVE MEMORIJE

Memorija se može preopteretiti kada programeri ne primenjuju princip ponovne upotrebe koda, uzorke dizajna i deklarišu veliki broj funkcija i promenljivih koji se nikad ne koriste

Veličina koda se može povećati u situacijama kada programeri:

- Ne primenjuju princip ponovne upotrebe koda: uzimimo jedan primer, gde imamo četiri programa iste aplikacije, a prvih deset redova programa su slični. Ovih deset redova možemo napisati kao diskretnu funkciju, koja bi trebalo da bude dostupna gore navedenim programima. Takođe, ako postoji neka greška, možemo da izmenimo liniju koda u funkciji, a ne ceo kod.
- Ne primenjuju uzorke dizajna već za rešenje problema koriste sopstvenu logiku koja se može modifikovati. Tako na primer, jedan programer može da napiše kod veličine 250kb, onda bi drugi programer mogao napisati sličan kod koristeći različitu logiku veličine do 100kb.
- Deklarše veliki broj funkcija i promenljivih koji se nikad ne koriste u bilo kom delu koda zbog čega se veličina programa povećava.

Na primer, u gore navedenom kodu možemo videti da integer a nikada nije pozvan nigde u programu, a takođe i funkcija Create user. To uzrokuje preopterećenje memorije.

```
Int a=15;
Int b=20;
String S= "Welcome";
...
```

```
....  
....  
....  
....  
Int p=b;  
Create user()  
{  
....  
....  
.... 200's line of code  
}
```

Ovakve greške ne možemo ispraviti ručnom proverom koda najčešće zbog veličine koda, već se za to mogu koristiti alati koji nam pomažu da testiramo nepotrebne promenljive i funkcije. Jedan od takvih alata je npr. alat **Rational purify** koji se koristi za programske jezike C i C++ dok za druge jezike postoje drugi srodnici.

TESTIRANJE PERFORMANSI (BRZINE, VREMENA ODZIVA) PROGRAMA

Prilikom testiranja bele kutije, programeri mogu da shvate da kod radi sporo ili ne mogu ručno da prođu kroz program i provere koja linija koda usporava program

Aplikacija može biti spora iz sledećih razloga:

- Kada se koristi složena logika.
- Za uslovne slučajevе kada neadekvatno koristimo or & and.
- Kada umesto Switch case, koristimo ugnježdeno ako (engl. **nested if**)

Prilikom testiranja bele kutije, programeri mogu da shvate da kod radi sporo ili ne mogu ručno da prođu kroz program i provere koja linija koda usporava program. Za to se može koristiti alat, na primer alat koji se zove **Rational Quantify**, koji takve probleme automatski rešava.

U trenutku kada je ceo kod spreman, **Rational Quantify** će proći kroz kod i izvršiti ga.

Nepomena: Prilikom korišćenja Rational Quantify, rezultat se može videti u obliku debelih i tankih linija pri čemu su debelim linijama određeni delovi programa čije izvršenje traje mnogo vremena.

Kako bi testiranjem moglo da se lakše upravlja i kako bi se u svakom trenutku lako razumelo šta je sledeći zadatak, testiranje bele kutije se izvršava kroz tačno definisan redosled koraka. Osnovni koraci u testiranju bele kutije su:

- U **prvom koraku** treba dizajnirati sve scenarije testiranja, slučajeve testiranja i odredite im prioritete

- **Drugi korak** obuhvata proučavanje koda tokom izvršenja kako bi se ispitala upotreba resursa, područja koja nisu dostupna, vreme koje je potrebno za izvršenje različitih metoda i operacija i tako dalje.
- U **trećem koraku** se vrši testiranje internih potprograma. Interni potprogrami, poput ne javnih metoda (engl. **nonpublic methods**), interfejsa su u mogućnosti da odgovarajuće vrste podataka obrađuju na odgovarajući ili ne odgovarajući način.
- **Četvrti korak** se fokusira na testiranje kontrolnih naredbi kao što su petlje i uslovne naredbe radi provere efikasnosti i tačnosti za različite unose podataka.
- **Posljednji korak** testiranja bele kutije uključuje testiranje sigurnosti kako bi se proverili svi mogući sigurnosni nedostaci.

PREDNOSTI I NEDOSTACI TESTIRANJA METODOM BELE KUTIJE

Pošto su testovi zasnovani na postojećim putanjama, nepostojeći putevi ne mogu biti otkriveni metodom bele kutije.

Metod bele kutije ima nekoliko nedostataka:

- Broj putanja, koje treba izvršiti može biti veoma veliki, i u tom slučaju, pokušaj da se testiraju svi putevi izvršavanja, skoro je nemoguć.
- Odabrani test slučajevi ne mogu da otkriju greške osjetljive na vrednost podataka. Na primer: $p=q/r$ se može izvršiti korektno, izuzev u slučaju kada je $r = 0$, $y=2*x$ napisano umesto $y=x^2$ proći će testne slučajeve $x=0, y=0$ i $x=2, y=4$
- Metod podrazumeva da je kontrola toka ispravna (ili bar blizu ispravnoj). Pošto su testovi zasnovani na postojećim putanjama, nepostojeći putevi ne mogu biti otkriveni metodom bele kutije.
- Tester mora da poseduje programerske veštine da bi razumeo i procenio softver koji se testira. Nažalost, većina testera danas nema zahtevane veštine.
- Nemogućnost testiranja "špageti" koda, zbog prevelikog broja puteva.
- Postojanje problema sa višestrukom procesima, nitima, prekidima, obradom događaja.

Prednosti:

- S obzirom da je preduslov za korišćenje metode bele kutije poznavanje interne strukture koda, veoma je lako saznati koji tipovi ulaznih i izlaznih podataka mogu doprineti efektivnom testiranju programa.
- Tester može biti siguran da je svaka putanja, u softveru koji se testira, identifikovana i testirana.
- Korišćenje metode bele kutije pomaže pri optimizaciji koda.
- Korišćenje ovog metoda pomaže da se uklone nepotrebne linije koda, koje mogu voditi ka skrivenim greškama.
- Postojanje automatskih alata, koji predlažu puteve za testiranje i nadgledaju pokrivenost tokom testiranja koda.
- Poznavanjem unutrašnje strukture (tj. samog koda) vrlo lako je uvideti koji tip ulaznih podataka može pomoći u efikasnom testiranju aplikacije,

- Pomoć u optimizaciji koda,
- Uklanjanje suvišnih linija koda koje mogu izazvati defekte.

▼ Poglavlje 2

Pokrivenost koda

ŠTA JE POKRIVENOST KODA?

Pokrivenost koda je mera koja opisuje stepen testiranja izvornog koda programa; pomoću skupa test slučajeva pronađi područja programa koja se ne koriste.

Testiranje bele kutije se vrši da bi se

1. Identifikovali unutrašnji sigurnosni propusti.
2. Proverio način unosa podataka u kod.
3. Proverila funkcionalnost uslovnih petlji.
4. Testirale funkcije, objekti i naredbe na pojedinačnom nivou.

Opšti cilj testera je da proveri svaku putanju u programu. Ali, kao što je poznato, to je nemoguće. Čak i u slučaju jednostavnog programa, postoji trilion različitih putanja. Zato, umesto provere svake od mogućih putanja, metoda bele kutije obezbeđuje mehanizme za izbor samo pojedinih putanja za testiranje.

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

Pokrivenost koda je mera koja opisuje stepen testiranja izvornog koda programa. To je jedan oblik testiranja bele kutije koji pomoći skupa test slučajeva pronađi područja programa koja se ne koriste. Takođe kreira test slučajeve kako bi se povećala pokrivenost i utvrdila kvantitativna mera pokrivenosti koda.

Uobičajeno je da se u projektima postavi ciljni nivo pokrivenosti koji, zatim, upućuje programere i testere na način za testiranje programa.

Testiranje metodom bele kutije primenjuje se na nivou izvornog koda (programski nivo).

TEHNIKE KOJE SE KORISTE PRILIKOM TESTIRANJA POKRIVENOSTI KODA

Koriste se sledeće tehnike

Prilikom testiranja pokrivenosti koda koriste se sledeće tehnike:

1. Pokrivenost naredbi (engl. Statement Coverage)

2. Pokrivenost grana (engl. Branch Coverage)
3. Pokrivenost putanja (engl. Path Coverage)
4. Pokrivenost odluka (engl. Decision Coverage)
5. Pokrivenost kontrole toka (engl. Control Flow Testing)
6. Pokrivenost toka podataka (engl. Data Flow Testing) itd.

✓ Poglavlje 3

Tehnika pokrivenosti naredbi

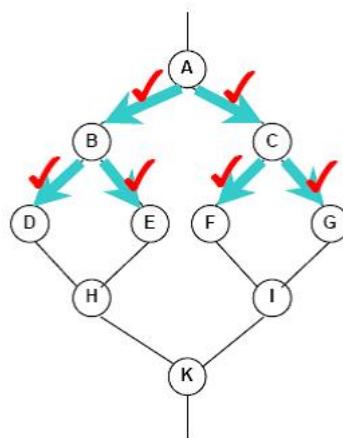
POKRIVENOST NAREDBI

Izvršavanje test slučajeva omogućava da program izvršava određene naredbe, i na taj način prolazi kroz određeni put kroz graf toka.

Pokrivenost naredbi određuje se na osnovu procene vrednosti broja ispitanih naredbi, tokom izvršavanja predloženih test slučajeva. Ako je tokom jednog testa svaka programska naredba izvršena bar jednom, smatra se da je pokrivenost naredbi 100%.

Pokrivanje naredbi odgovara izvršavanju čvorova na grafu toka. 100% pokrivenost je ostvarena ako su posećeni svi čvorovi na putevima kroz koje su prošli test slučajevi. Putanja A, B, D, H, K predstavlja jedan put izvršavanja programa. Na putu je posećeno 5, od ukupno 10 čvorova, što predstavlja 50% pokrivenost. (Slika 1)

Stvaran nivo pokrivanja naredbi može se razlikovati od nivoa pokrivanja čvorova, što zavisi od načina definisanja grafa, odnosno zavisi od broja naredbi grupisanih u jedan čvor.



Slika 3.1 Primer prikazuje graf toka sa 10 čvorova [Izvor: NM SE321-2020/2021.]

KADA SE KORISTI TEHNIKA POKRIVENOSTI NAREDBI?

Koristi se za izračunavanje ukupnog broja izvršenih naredbi u izvornom kodu u odnosu na ukupan broj naredbi prisutnih u izvornom kodu.

Pokrivanje naredbi je jedan od široko korišćenih načina testiranja softvera koji spada u testiranje bele kutije.

Tehnika pokrivenosti naredbi se koristi za dizajniranje test slučajeva prilikom primene tehnika testiranja bele kutije. Ova tehnika uključuje izvršenje svih naredbi izvornog koda najmanje jednom. Koristi se za izračunavanje ukupnog broja izvršenih naredbi u izvornom kodu u odnosu na ukupan broj naredbi prisutnih u izvornom kodu.

Pokrivenost naredbi = (broj izvršenih naredbi / ukupan broj naredbi) * 100

Kod testiranja bele kutije, testeri se koncentrišu na interni izvorni koda i dijagrama toka ili graf toka koda.

Primer: Uzimamo izvorni kod za kreiranje dva različita scenarija za različite ulazne vrednosti, kako bismo proverili procenat pokrivenosti naredbi za svaki scenario.

Struktura izvornog koda:

- Unosimo dve vrednosti: a=0 i b=1.
- Pronalazimo sumu tih vrednosti
- Ako je suma veća od 0, tada štampamo "This is the positive result."
- Ako je suma manja od 0, tada štampamo "This is the negative result."

```
input (int a, int b)
{
    Function to print sum of these integer values (sum = a+b)
    If (sum>0)
    {
        Print (This is positive result)
    } else
    {
        Print (This is negative result)
    }
}
```

SCENARIJA TESTIRANJA

Posmatraćemo dva različita scenarija i izračunati procenat pokrivenosti naredbi za dati izvorni kod.

Posmatraćemo dva različita scenarija i izračunati procenat pokrivenosti naredbi za dati izvorni kod.

Scenario 1: Ako je a = 5, b = 4

```
print (int a, int b) {
    int sum = a+b;
    if (sum>0)
        print ("This is a positive result")
    else
```

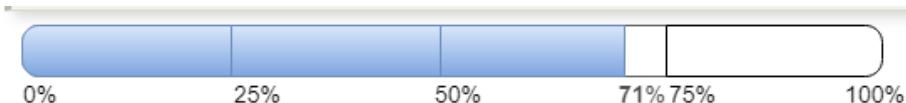
```
print ("This is negative result")
}
```

U scenariju 1, možemo videti da će vrednost sume biti 9 koja je veća od 0, a kao rezultat dobićemo „This is a positive result“. Da bismo izračunali pokrivenost naredbi u prvom scenariju, uzimamo ukupan broj naredbi koji je 7 i broj izvršenih naredbi koji je 5.

Ukupan broj naredbi = 7

Broj izvršenih naredbi = 5

Pokrivenost naredbi = $5/7 * 100 = 500/7 = 71\%$



Slika 3.2 Pokrivenost naredbi u scenariju 1. [Izvor: NM SE321-2020/2021.]

Scenario 2: Ako je A = -2, B = -7

U scenariju 2, možemo videti da će vrednost sume biti -9 koja je manja od 0, a prema uslovima, dobiće se rezultat "This is negative result". Da bismo izračunali pokrivenost naredbi drugog scenarija, uzimamo ukupan broj naredbi koji je 7 i broj izvršenih naredbi koji je 6.

Ukupan broj naredbi = 7

Broj izvršenih naredbi = 6

Pokrivenost naredbi = $6/7 * 100 = 600/7 = 85\%$



Slika 3.3 Pokrivenost naredbi u scenariju 2. [Izvor: NM SE321-2020/2021.]

Možemo videti da su sa oba scenarija obuhvaćene sve naredbe i možemo uzeti u obzir da je ukupna pokrivenost naredbi 100%.

Dakle, tehnika pokrivanja naredbi obuhvata mrtav kod, nekorišćeni kod i grananja.

▼ Poglavlje 4

Tehnika pokrivenosti grana

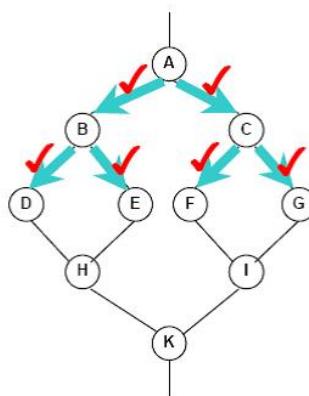
POKRIVANJE GRANA I POKRIVANJE PUTANJA

Ako je svaka grana u programu izvršena bar jednom tokom jednog testa, smatra se da je pokrivenost grana 100%.

Pokrivenost grana se određuje na osnovu izvršenja grana od strane skupa predloženih test slučajeva. Ako je svaka grana u programu izvršena bar jednom tokom jednog testa, smatra se da je pokrivenost grana 100%.

Pokrivenost grana odgovara izvršavanju uslovnih grana na grafu toka. 100% pokrivenost je ostvarena ako su posećene sve uslovne grane, na putevima kroz koje su prošli test slučajevi.

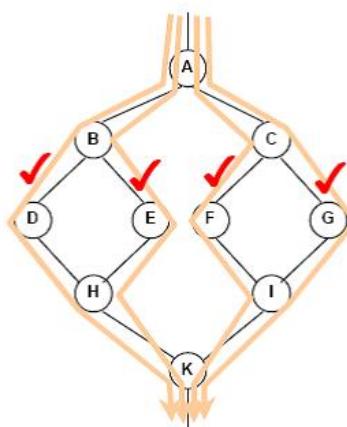
Primer prikazuje graf toka sa 6 uslovnih grana (Slika 1) . A, B, D, H, K predstavlja jedan put izvršavanja programa. Na putu su posećene 2 od ukupno 6 uslovnih grana, pa je pokrivenost 33%.



Slika 4.1 Primer programskog fragmenta sa 6 uslovnih grana [Izvor: NM SE321-2020/2021.]

Pokrivenost putanja se s druge strane određujena osnovu izvršavanja putanja pomoću skupa predloženih test slučajeva. Ako je svaka putanja u programu izvršena najmanje jednom u toku jednog testa, smatra se da je pokrivenost putanja 100%. Pokrivanje putanja odgovara izvršavanju putanja na grafu toka. 100% pokrivenost je ostvarena ako su posećene sve putanje na putevima kroz koje su prošli test slučajevi.

Primer prikazuje graf toka sa 4 moguće putanje (Slika 2) . A, B, D, H, K predstavlja jedan put izvršavanja programa. Na putu se prolazi kroz 1 od ukupno 4 putanje, pa je pokrivenost 25%.



Slika 4.2 Primer programskog fragmenta sa 4 moguće putanje [Izvor: NM SE321-2020/2021.]

ŠTA OBEZBEĐUJE TEHNIKA POKRIVENOSTI GRANA

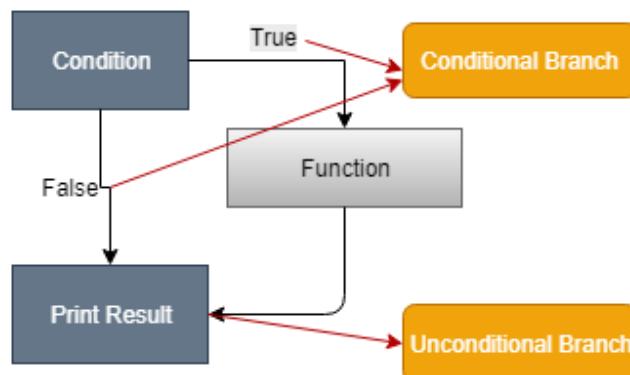
Tehnika pokrivenosti grana je tehnika testiranja bele kutije koja osigurava da se svaka grana svake tačke odluke mora izvršiti.

Testiranje pokrivenosti grana se koristi kako bi se pokrile sve grane grafikona kontrolnog toka. Ona pokriva sve moguće ishode (tačne i netačne) svakog uslova u tačkama odlučivanja barem jedanput. Tehnika pokrivenosti grana je tehnika testiranja bele kutije koja osigurava da se svaka grana svake tačke odluke mora izvršiti.

Međutim, iako su tehnike pokrivenosti grana i tehnika pokrivenosti odluka vrlo slične, postoji ključna razlika između njih. Tehnika pokrivenosti odluka obuhvata grane određene tačke odluke, dok testiranje pokrivenosti grana pokriva sve grane svih tačaka odluke koda.

Postoji mnogo različitih metrika koje se mogu koristiti za pronalaženje pokrivenosti grana i pokrivenost odluka, ali neke od najosnovnijih metrika su: pronađak procenta programa i putanja izvršenja tokom izvršenja programa.

Kao i pokrivenost odluka, ova tehnika takođe koristi grafikon kontrolnog toka za izračunavanje broja grana.



Slika 4.3 Primer grafikona kontrolnog toka Izvor: <https://www.javatpoint.com/branch-coverage-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

Postoji nekoliko metoda za izračunavanje pokrivenosti grana, ali najčešća metoda je pronalaženje puta (pathfinding). U ovoj metodi, se za izračunavanje pokrivenosti grana koristi broj putanja izvršenih grana.

Tehnika pokrivenosti grana se može koristiti kao alternativa pokrivanju odluke. Negde, ona nije definisana kao individualna tehnika, ali je različita od pokrivenosti odluke i neophodna je za testiranje svih grana grafa kontrolnog toka.

PRIMER

Provera pokrivenosti grana se vrši metodom pronalaženja puta

```
Read X
Read Y
IF X+Y > 100 THEN
Print "Large"
ENDIF
If X + Y<100 THEN
Print "Small"
ENDIF
```

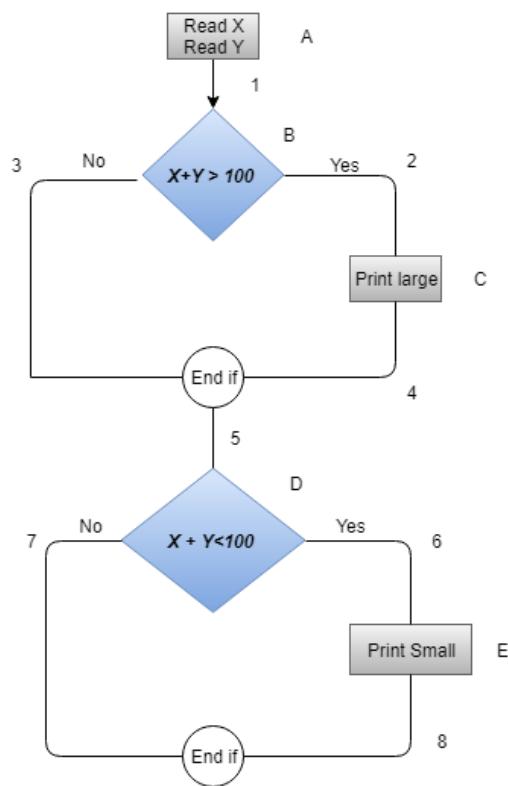
Ovo je osnovna struktura koda gde smo uzeli dve promenljive X i Y i dva uslova.

Ako je prvi uslov tačan, tada se štampa „Large“, a ako je netačan, prelazi se na sledeći uslov.
Ako je drugi uslov tačan, štampa se „Small“.

Graf kontrolnog toka za datu strukturu koda je predstavljen na slici 4. U prvom slučaju prolazeći kroz odluku "Yes", putanja je A1-B2-C4-D6-E8, a pokrivenе grane su 1, 2, 4, 5, 6 i 8, dok grane 3 i 7 nisu pokrivenе na ovoj putanji. Da bismo pokrili ove grane, moramo da pređemo kroz odluku „No“. U slučaju odluke „No“, putanja je A1-B3-5-D7, a brojevi pokrivenih grana su 3 i 7. Dakle, prolaskom kroz ove dve putanje pokrivenе su sve grane.

Putanja 1: A1-B2-C4-D6-E8

Putanja 2: A1-B3-5-D7



Slika 4.4 Graf kontrolnog toka za datu strukturu koda Izvor: <https://www.javatpoint.com/branch-coverage-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 5

Tehnika pokrivenosti odluka

KAKO SE PRONALAZI PROCENAT POKRIVENOSTI ODLUKA?

Procenat pokrivenosti odluka se može naći deljenjem broja proverenih rezultata s ukupnim brojem ishoda, pomnoženo sa 100.

Tehnika pokrivenosti odluka spada pod testiranje bele kutije koja pokrivenost odluka daje kao boolean vrednosti.

Ova tehnika izveštava o tačnim i netačnim ishodima boolean izraza. Kad god postoji mogućnost dva ili više ishoda naredbi, kao što je do while naredba, if naredbai case naredba (naredbe za kontrolu toka), one se mogu tretirati kao tačkeodluke jer postoje dva ishoda: istinit ili lažni.

Pokrivenost odluka obuhvata sve moguće ishode svakog boolean uslova u kodu upotreboom grafikona ili dijagrama kontrolnog toka.

Generalno, tačka odluke ima dve vrednosti odluke, jedna je tačna, a druga je netačna, zbog čega je u većini slučajeva ukupan broj ishoda dva. Procenat pokrivenosti odluka se može naći deljenjem broja proverenih rezultata s ukupnim brojem ishoda, pomnoženo sa 100.

Pokrivenost odluka = (broj odluka čiji su izlazi provereni / ukupan broj izlaza odluka) * 100

U ovoj je tehnici teško dobiti 100% pokrivenost jer se ponekad izrazi iskomplikuju. Zbog toga postoji nekoliko različitih metoda za izveštavanje o pokrivenosti odluka

Prednost ovih metoda je poboljšanje osetljivosti kontrolnog toka.

Broj pokrivenosti odluka možemo pronaći na sledeći način.

PRIMER: POKRIVENOST ODLUKE ZA TAČAN ISHOD

U tom slučaju procenat pokrivenosti odluke je 50%

Primer. Razmotrimo kod koji će se primeniti tehnika pokrivenosti odluka:

```
Test (int a)
{
If(a>4)
a=a*3
Print (a)
```

}

Scenario 1: vrednost za a je 7 (a=7)

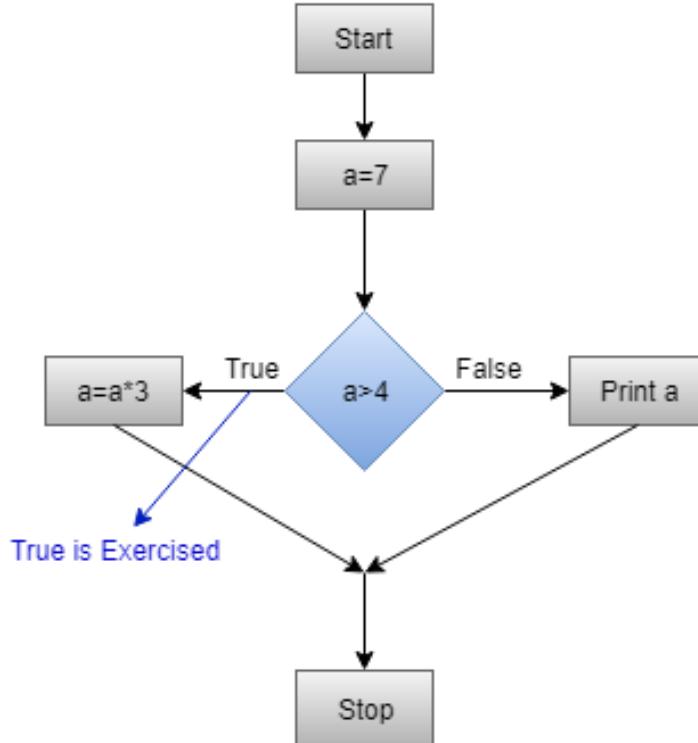
```
Test (int a=7)
{ if (a>4)
a=a*3
print (a)
}
```

Ishod ovog koda je „Tačno“ ako se proveri uslov ($a > 4$). Grafikon kontrolnog toka kada je vrednost a 7 je dat na slici 1.

Sračunavanje procента pokrivenosti odluka:

Pokrivenost odluke = $\frac{1}{2} * 100$ (izvršava se samo "True") = $100/2 = 50$

Pokrivenost odluke 50%



Slika 5.1 Grafikon kontrolnog toka kada je vrednost a 7 Izvor: <https://www.javatpoint.com/decision-coverage-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

PRIMER: POKRIVENOST ODLUKE ZA NETAČAN ISHOD

I u tom slučaju procenat pokrivenosti odluke je 50%

Scenario 2: vrednost za a je 3 (a=3)

```
Test (int a=3)
{ if (a>4)
a=a*3
print (a)
}
```

Ishod ovog koda je „Netačno“ ako se proverava uslov ($a > 4$). Grafikon kontrolnog toka kada je vrednost $a=3$ je data na slici 2.

Sračunavanje procента pokrivenosti odluka:

$$= \frac{1}{2} * 100 \text{ (izvršava se samo "Netačno" is exercised)} = 100/2 = 50$$

Pokrivenost odluke = 50%

Tabela rezultata pokrivenosti odluke data je na slici 3.

Test Case	Value of A	Output	Decision Coverage
1	3	3	50%
2	7	21	50%

Slika 5.2 Tabela rezultata pokrivenosti odluke [Izvor: NM SE321-2020/2021.]

Slika 5.3 Grafikon kontrolnog toka kada je vrednost a 3 Izvor: <https://www.javatpoint.com/decision-coverage-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

✓ Poglavlje 6

Testiranje kontrole toka

ŠTA JE CILJ OVE TEHNIKE TESTIRANJA

Cilj ove tehnike je utvrđivanje redosleda izvršenja naredbi ili instrukcija programa kroz kontrolnu strukturu

Cilj ove tehnike je utvrđivanje redosleda izvršenja naredbi ili instrukcija programa kroz kontrolnu strukturu.

Kontrolna struktura programa se koristi za izradu test slučajeva za program. U ovoj tehnici, tester bira određeni deo velikog programa da bi ustanovio putanju testiranja. Najčešće se koristi u jediničnom testiranju. Test slučajevi se generišu na osnovu kontrolnog grafa programa koji se formira od:

- 1. **čvorova** (engl. **nodes**): čvorovi se na grafikonu kontrolnog toka koriste za kreiranje putanja procedura. U osnovi, čvor predstavlja sekvencu, redosled pojavljivanja procedura. U primeru koji sledi, možemo videti da prvi čvor predstavlja proceduru pokretanja, sledeći proceduru dodeljivanja vrednosti, nakon dodeljivanja vrednosti je čvor odluke koji odlučuje o sledećoj proceduri prema vrednosti n, ako je 18 ili više od 18. Sledeći čvor je čvor spajanja, a poslednji čvor je zaustavni čvor da bi se procedura zaustavila.
- 2. **grana** (engl. **edges**): služe za povezivanje čvorova u smeru njihovog izvršavanja te su grane orientisane. Kao što se iz primera koji sledi može videti, za povezivanje čvorova u odgovarajućem smeru se koriste strelice.
- 3. **čvorovi grananja** (engl. **branch nodes**): to su čvorova koji imaju više od jedne izlazne grane; Oni u kontrolnom grafu služe da se odredi sledeći čvor (procedura) na osnovu ispitivane vrednosti. U zadatom primeru čvor grananja služi da se odluči o sledećem čvoru procedure na osnovu vrednosti n; ako je n jednak ili veće 18 izvršiće se Eligible procedure u suprotnom ako je manje od 18 ne izvršava se Eligible procedure
- 4. **čvorova spajanja** (engl. **junction nodes**): Predstavlja gde se spajaju najmanje tri veze.

PRIMER

Primerom se ispituju kriterijumi podobnosti za glasanje prema starosti: ako je starost 18 ili više od 18 godina ili ako je manja od 18

```
public class VoteEligibilityAge{
```

```

public static void main(String []args){
int n=45;
if(n>=18)
{
    System.out.println("You are eligible for voting");
} else
{
    System.out.println("You are not eligible for voting");
}
}

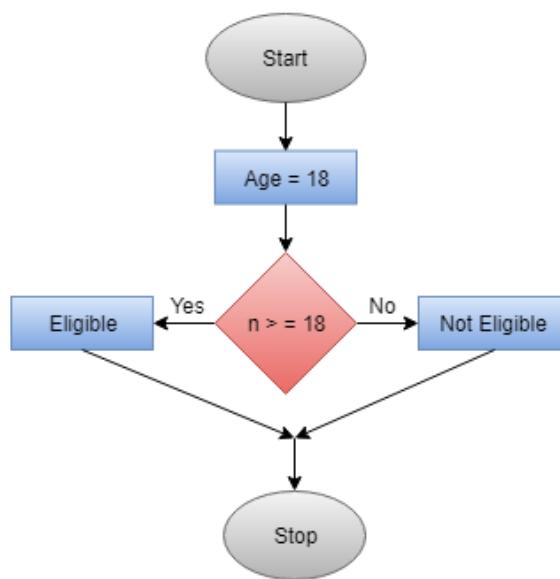
```

Ovim primerom se ispituju kriterijumi podobnosti za glasanje prema starosti: ako je starost 18 ili više od 18 godina, ispisuje se poruka „Imate pravo da glasate“ ako je manji od 18 onda se štampa „Ne ispunjavate uslove za glasanje“.

U svrhu ispitivanja, kreiran je graf kontrole toka za program iz primera. U grafu kontrole toka:

- 1. čvorovi su: početak, starost, podoban, nepodoban i zaustavljanje,
- 2. čvor odluke je $n >= 18$ kojim će odlučiti koji će se deo (if ili else) izvršiti zavisno od date vrednosti.
- 3. Veza između "eligible" čvora i "not eligible" čvora je ostvarena preko stop čvora.

Test slučajevi su dizajnirani kroz dijagram toka programa kako bi se utvrdilo da je put izvršenja tačan ili ne. Svi čvorovi, spoj, ivice i odluka su bitni delovi za dizajn test slučajeva.



Slika 6.1 Kontrolni graf programa Izvor: <https://www.javatpoint.com/control-flow-testing-in-white-box-testing> [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 7

Testiranje petlji i uslovnih grana

TESTIRANJE PETLJI - TESTIRANJE OSNOVNOG PUTEA

Postoje brojne preporuke za izbor puteva kroz petlju za testiranje.

Postojanje petlji u programu uglavnom rezultira postojanjem velikog broja puteva kroz program. Uglavnom, nije izvodljivo testiranje svih mogućih puteva. Postoje brojne preporuke za izbor puteva kroz petlju za testiranje, neke od njih su:

- Skoro je nemoguće ili neizvodljivo testirati sve puteve u programu koji sadrži petlje
- Postoje brojne preporuke za izbor puteva kroz petlju za testiranje.
- Postoji šest elemenata pokrivenosti za svaku petlju L (sa uslovom C)
 - Najmanje jedan test slučaj uzrokuje da C bude na startu netačno – "Petlja se izvršava 0 puta"
 - Najmanje jedan test slučaj uzrokuje da C bude tačno prvi put netačno drugi put – "Petlja se izvršava 1 put"
 - Najmanje jedan test slučaj "izvršava petlju 2 puta"
 - Ako postoji maksimalni broj puta n: Najmanje jedan test slučaj izvršava petlju n puta
 - Najmanje jedan test slučaj izvršava petlju n-1 puta
 - Najmanje jedan test slučaj izvršava petlju n+1 puta

Poslednja 3 elementa smatraju se neizvodljivim ako ne postoji jasno definisano n.

Testiranje osnovnog puta: Osnovne putanje obezbeđuju mehanizam za testiranje puteva koji uključuju petlje.

1. Nulti-put (**Zero-path**): testira nula iteracija u telu petlje.
2. Jedan-put (**One-path**): testira jednu iteraciju.

Osnovni putevi predstavljaju sastavne puteve svih drugih puteva. Svi mogući putevi su kombinacija i sekvene osnovnih puteva. Kod testiranja petlji pomoću osnovnih puteva, cilj testera je da se izvrši svaki osnovni put sa najmanje jednim testom. I nulti-put i jedan-put treba da budu pokriveni.

Kombinacije i sekvene osnovnih puteva ne treba da budu izvršavane posebnim testom. Podrazumeva se da će biti pokriveni kombinacijama i sekvcencama već testiranih

PRIMER OSNOVNOG PUTEA

Potrebno je kreirati testove za izvršavanje ovih puteva.

Pogledaćemo primer programa za binarno pretraživanje: (Slika 1) .

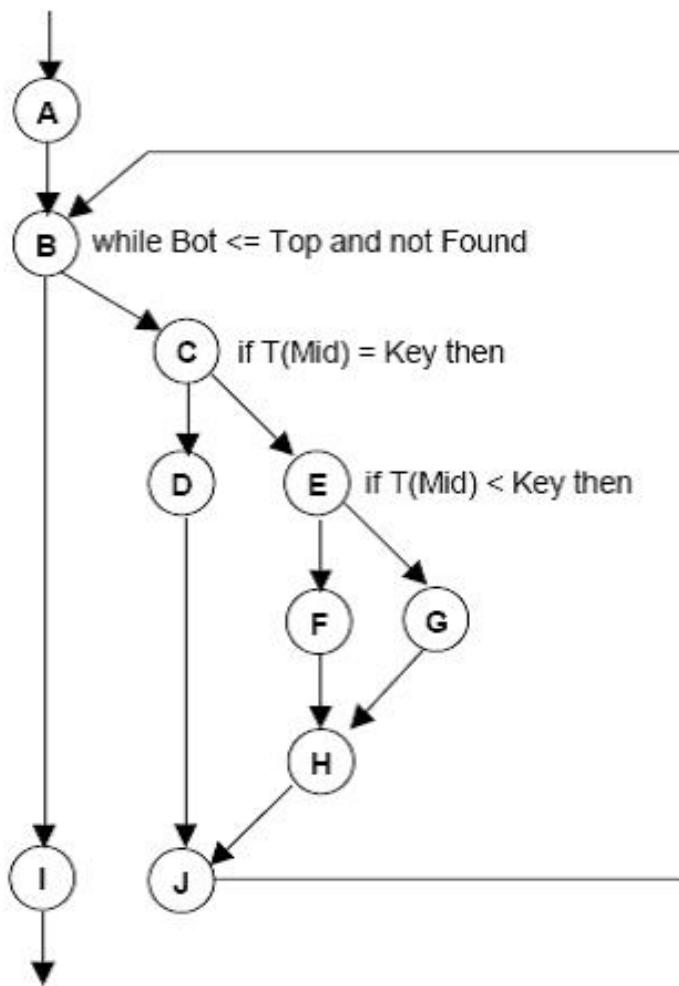
Osnovni putevi kroz ovaj program su:

1. Zero-path
 - B, C
2. One-paths
 - B, C, D, J
 - B, C, E, F, H, J
 - B, C, E, G, H, J

Potrebno je kreirati testove za izvršavanje ovih puteva.

Testiranje osnovnog puta ne razmatra:

1. očekivani završetak petlje (dostizanje maksimalne vrednosti).
2. između dve iteracije, može se preskočiti provera promene uslova.



Slika 7.1 Primer programa za binarno pretraživanje [Izvor: NM SE321-2020/2021.]

TESTIRANJE PETLJI PO BEIZER-U

Beizer predlaže brojne, druge testne kategorije za iteraciju petlji.

Beizer predlaže brojne, druge testne kategorije za iteraciju petlji:

1. Bypass: bilo koja vrednost, koja će usloviti trenutni izlaz iz petlje
2. Once: vrednosti koje dovode da se petlja izvrši tačno jednom
3. Twice: vrednosti koje dovode do izvršavanja petlji tačno dva puta
4. Typical: uobičajeni broj iteracija
5. Max: maksimalni broj dozvoljenih iteracija
6. Max + 1: jedna iteracija više od maksimalno dozvoljenog broja
7. Max - 1: jedna iteracija manje od maksimalno dozvoljenog broja i
8. Min: minimalni broj dozvoljenih iteracija
9. Min + 1: jedna iteracija više od minimalno dozvoljenog broja
10. Min - 1: jedna iteracija manje od minimalno dozvoljenog broja
11. Null: jedna sa null ili praznom vrednošću za broj iteracija
12. Negative: jedna sa negativnom vrednošću za broj iteracija

Neki slučajevi se mogu preklapati, ili ne mogu biti primenljivi u svakoj petlji, na primer, Bypass sa Min/Null/Negative.

POKRIVANJE USLOVNIH GRANA

Testiranje uslovnih grana identificuje individualne bulove operative u okviru uslova.

Testiranje uslovnih grana identificuje individualne bulove operative u okviru uslova. Ovi uslovi obično se koriste u okviru if-then-else, ili while (loop) uslova.

Testovi izvršavaju pojedinačne vrednosti bulovih operanada, kao i njihove kombinacije.

```

if A or (B and C) then
    do_something
else
    do_something_else
  
```

U ovom slučaju, uslov je A or (B and C). Bulovi operandi su A, B i C.

Pokrivanje uslovnih grana sastoji se od skupa testova, pomoću kojih se testira svaki operand za rezultate TRUE i FALSE.

Testiranje prethodnog primera (A or (B and C)) opisan je tabelom na sledećoj slici:

Testni slučaj	A	B	C	Izlaz
1	true	false	false	true
2	false	true	true	true

Slika 7.2 Primer programa za binarno pretraživanje: test slučajevi [Izvor: NM SE321-2020/2021.]

Slabosti prethodnog dizajna test slučajeva:

- Može biti dostignuto bez izvršavanja oba rezultata za uslov. U prethodnom primeru, u oba slučaja rezultat uslova je TRUE.

- Može biti dostignuto sa samo dva slučaja. Na primer (Slika 3) :

Testni slučaj	A	B	C	Izlaz
1	false	false	false	false
2	true	true	true	true

Slika 7.3 Primer programa za binarno pretraživanje: test slučajevi koji otklanjaju slabost prethodnih
[Izvor: NM SE321-2020/2021.]

Pokrivanje kombinacija kod uslovnih grana obavlja se izvršavanjem svih kombinacija TRUE i FALSE vrednosti za sve operande. Moguće kombinacije za prethodni primer (A or (B and C)):

Slika 7.4 Primer programa za binarno pretraživanje: Moguće kombinacije za prethodni primer (A or (B and C)) [Izvor: NM SE321-2020/2021.]

MODIFIKOVANO POKRIVANJE KOMBINACIJA KOD USLOVNIH GRANA

S obzirom na veoma često postojanje velikog broja kombinacija kod uslovnih grana, neophodno je smanjiti broj testnih slučajeva.

Kod uslovnih grana, 100% pokrivanje kombinacija zahteva $2n$ slučajeva za n operanada.

- 2 slučaja za 1 operand
- 4 slučaja za 2 operanda
- 8 slučajeva za 3 operanda itd.

Za kompleksnije uslove vrlo brzo postaje neizvodljivo.

S obzirom na veoma često postojanje velikog broja kombinacija kod uslovnih grana, neophodno je smanjiti broj testnih slučajeva.

Namera je da se pokaže da svaki bulov operand može nezavisno uticati na rezultat uslova. Svakom operandu se menjaju vrednosti između TRUE i FALSE, dok se vrednost ostalih operanada zadržavaju na određenim, fiksnim vrednostima. Rezultat treba da bude različit kôd promena vrednosti operanda.

Kako je ova situacija moguća, zahteva se pronalaženje fiksnih vrednosti, koje će biti dodeljeni drugim operandima.

PRIMER: MODIFIKOVANO POKRIVANJE KOMBINACIJA KOD USLOVNIH GRANA

Kako je ova situacija moguća, zahteva se pronalaženje fiksnih vrednosti, koje će biti dodeljeni drugim operandima.

Na primer, rezultat uslova = A or (B and C)

Vrednosti za operand A se menjaju, dok vrednosti za B i C ostaju iste kao u tabeli na sledećoj slici:

Case	A	B	C	Outcome
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE

Slika 7.5 Vrednosti za operand A se menjaju, dok vrednosti za B i C ostaju iste [Izvor: NM SE321-2020/2021.]

Menjaju se vrednosti za operand B, dok vrednosti za A i C ostaju iste kao u sledećoj tabeli na slici:

Case	A	B	C	Outcome
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE

Slika 7.6 Menjaju se vrednosti za operand B, dok vrednosti za A i C ostaju iste [Izvor: NM SE321-2020/2021.]

Menjaju se vrednosti za operand C, dok vrednosti za A i B ostaju iste kao u sledećoj tabeli na slici:

Case	A	B	C	Outcome
C1	FALSE	TRUE	FALSE	FALSE
C2	FALSE	TRUE	TRUE	TRUE

Slika 7.7 Menjaju se vrednosti za operand C, dok vrednosti za A i B ostaju iste [Izvor: NM SE321-2020/2021.]

100% pokrivenost zahteva se postiže sa:

- minimum $n+1$ test slučajeva
- maksimum $2n$ testnih slučajeva

Koristeći prethodni primer, minimalni testovi: A1 & B1 su isti, B2 & C1 su isti kao u sledećoj tabeli na slici:

Case	A	B	C	Outcome
1 (A1, B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2, C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

Slika 7.8 Koristeći prethodni primer, minimalni testovi: A1 & B1 su isti, B2 & C1 su isti [Izvor: NM SE321-2020/2021.]

Ili objedinjeno kao što je prikazano na sledećoj slici:

Slika 7.9 Koristeći prethodni primer, objedinjeni test slučajevi [Izvor: NM SE321-2020/2021.]

RAZMATRANJE POKRIVANJA USLOVNIH GRANA

Metod bele kutije koristan je za analizu i testiranje određenih puteva u programu pomoću test slučajeva.

U programskom kodu, bulov uslov se može nalaziti van tačke odlučivanja.

Na primer:

FLAG := A or (B and C)

if FLAG then ...

U ovom slučaju, potrebno je testirati sve bulove izraze. Kod prethodnog primera, to znači:

A or (B and C)

FLAG

Neki kompjajleri uvode optimizacije, koje mogu praviti teškoće kod prikaza pokrivenosti za sve operande. Na primer, C i C++ imaju skraćene operatore `&&` i `||` koji ne rade absolutno isto kao konvencionalni "and" i "or" (mogu preskočiti analizu drugog operanda).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 8

White box - Dizajniranje test slučajeva

ODABIR PUTANJA NA OSNOVU DEFINISANOG KRITERIJUMA

Greške i nemamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni.

U prethodnoj sekciji smo istakli da **White box** ili **struktурно тестирање** obuhvata ispitivanje interne strukture programa ili sistema. Testni podaci se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji. Tester poznaje internu strukturu i logiku programa, isto kao što mehaničar zna unutrašnji mehanizam automobila. Specifični primeri koji spadaju u ovu kategoriju uključuju testiranje izraza, gransko testiranje, testiranje uslova i sl. Prednost ovog pristupa je da je potpun i da se fokusira na proizvedeni kod. Greške i nemamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni.

Osnovna ideja struktornog testiranja je testiranje:

- Dve vrste osnovnih programske naredbi:
 - Naredbe dodeljivanja ($x = 2*y;$)
 - Uslovne naredbe (if(), for(), loop, while(), ...)
- Kontrola toka
 - Sukcesivno izvršavanje programske naredbe se posmatra kao kontrola toka.
 - Uslovne naredbe menjaju podrazumevani tok.
- Programska putanja
 - Programska putanja je niz naredbi koje se izvršavaju od starta do završetka programa.
 - U programu može postojati veliki broj putanja.
 - Za svaki put postoji par (ulaz, očekivani izlaz).
 - Izvršavanje određenog puta zahteva pozivanje programa sa odgovarajućim ulazom.
 - Putanje se biraju korišćenjem kriterijuma za selekciju putanja.

Kriterijum za selekciju putanja

- Programske putanje se selektivno izvršavaju. Pitanje: Koje putanje izabrati za testiranje?
- Koncept kriterijuma za selekciju putanja se koristi da bi se odgovorilo na postavljeno pitanje.
- Prednosti odabira putanja na osnovu definisanog kriterijuma: Garancija da će se sve programske konstrukcije izvršiti najmanje jednom.

- Ponavljanje izvršenja istog puta se izbegava, i na taj način izbegava rasipanje resursa.
- Može se jednostavno odrediti šta da se testira, a šta ne.

KRITERIJUM ZA SELEKCIJU PUTANJA

Kriterijumi za selekciju putanja je odabrati puteve koji omogućavaju potpunu pokrivenost naredbi, pokrivenost grana i pokrivenost uslova

Kriterijumi su:

- Odabrati sve puteve.
 - Na ovaj način se mogu otkriti svi nedostaci, izuzev onih koji se javljaju zbog nepostojećih puteva.
 - Poželjno je korišćenje ovog kriterijuma, ali je teško dostižno u praksi (zbog velikog broja puteva)
- Odabrati puteve koji omogućavaju potpunu pokrivenost naredbi.
 - Postiže se ukoliko se svaka naredba izvrši makar jednom
 - Predstavlja najslabiji kriterijum
 - Bilo koji testni paket koji postiže manju pokrivenost od potpune pokrivenosti naredbi, smatra se neprihvatljivim.
- Odabrati puteve koji omogućavaju potpunu pokrivenost grana.
- Odabrati puteve koji omogućavaju potpunu pokrivenost uslova.

Pokrivenost skupom testnih slučajeva

- Pokrivenost skupom testnih slučajeva je mera srazmere pokrivanja naredbi, grana i putanja skupom test slučajeva.
- Pokrivenost naredbi, pokrivenost grana i pokrivenost putanja su tehnike strukturnog testiranja, koje se koriste da predlože test slučajeve na osnovu logičke strukture programa.

POKRIVENOST SKUPOM TESTNIH SLUČAJEVA - DIZAJNIRANJE TEST SLUČAJEVA

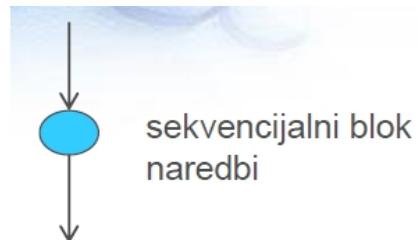
U cilju dizajniranja testnih slučajeva polazi se od grafa toka izvršavanja programa na osnovu kontrolnih struktura.

Dizajniranje test slučajeva za pokrivenost:

1. Kreiranje grafa toka na osnovu analize
2. Predlog puteva testiranja na osnovu grafa tokada bi se postigla zahtevana pokrivenost
3. Procena testnih uslova za ostvarenje svakog puta i
4. Predlog ulaznih i izlaznih vrednosti na osnovu uslova.

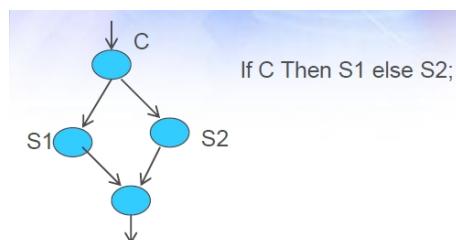
U cilju dizajniranja testnih slučajeva polazi se od grafa toka izvršavanja programa na osnovu sledećih kontrolnih struktura:

1. Sekvencijalni blok naredbi (Slika 1)



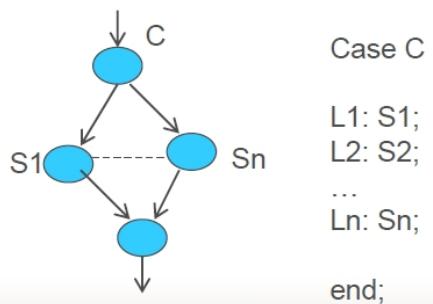
Slika 8.1 Sekvencijalni blok naredbi [Izvor: NM SE321-2020/2021.]

2. Grananje toka: If C Then S1 else S2; (Slika 2)



Slika 8.2 Grananje toka: If C Then S1 else S2 [Izvor: NM SE321-2020/2021.]

3. Izbor toka: Case (Slika 3)

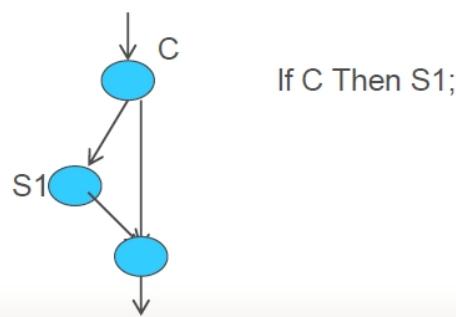


Slika 8.3 Grananje toka: Case [Izvor: NM SE321-2020/2021.]

TIPOVI PUTANJA I KRITERIJUMI ZA POKRIVANJE

U cilju dizajniranja testnih slučajeva polazi se od grafa toka izvršavanja programa na osnovu kontrolnih struktura: If ... then... i case.

4. Grananje toka: If C Then S1; (Slika 4)

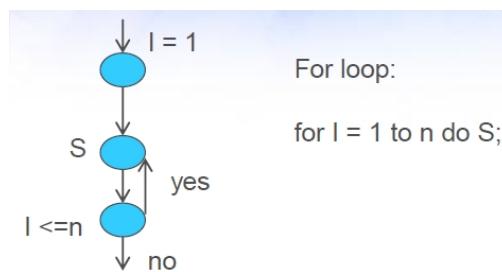


Slika 8.4 Grananje toka: If C Then S1; [Izvor: NM SE321-2020/2021.]

5. Petlja:While C do S (Slika 5)

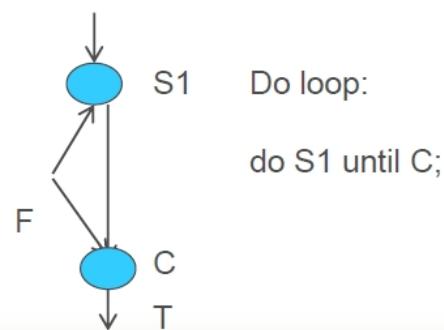
Slika 8.5 Petlja:While C do S [Izvor: NM SE321-2020/2021.]

6. For loop: for I = 1 to n do S; (Slika 6)



Slika 8.6 Petlja:For loop [Izvor: NM SE321-2020/2021.]

7. Petlja: Do loop - do S1 until C; (Slika 7)



Slika 8.7 Petlja: Do loop [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 9

Studija slučaja

PROGRAM KOJI IZVRŠAVA BINARNO PRETRAŽIVANJE

Program počinje izborom vrednost u sredini niza. Zatim, poredi odabranu vrednost sa ključem.

Razmotrićemo program, koji izvršava binarno pretraživanje prikazano na Slici 1 . Primenićemo pristup "bele kutije, a koncept bele kutije je već poznat studentima, a ovde ćemo demonstrirati jedan primer primene koncepta "bele kutije". .

Program počinje izborom vrednosti u sredini niza. Zatim, poredi odabranu vrednost sa ključem. Ako su ove dve vrednosti iste, program daje izveštaj o pronalaženju ključa na središnjoj poziciji i završava se. Ako vrednosti, koje se porede, nisu iste, upoređuje se vrednost na sredini niza sa ključem:

- Ako je vrednost manja od ključa, tada ključ može biti jedino pronađen posle središnje pozicije i kraja niza, srednje i gornje vrednosti niza se menjaju i nastavlja se pretraživanje sa novim podešavanjima.
- Ako je vrednost veća od ključa, tada ključ mora biti u donjem delu niza, srednje i gornje vrednosti se menjaju i nastavlja se pretraživanje sa novim podešavanjima.

```

procedure Binary_Search (Key : ELEM; T : ELEM_ARRAY;
    Found : in out BOOLEAN; L : in out ELEM_INDEX ) is
    -- Assume that T'FIRST and T'LAST are both
    -- than or equal to zero and T'LAST >= T'FIRST
    Bot : ELEM_INDEX := T'FIRST;
    Top : ELEM_INDEX := T'LAST;
    Mid : ELEM_INDEX;
begin
    L := (T'FIRST + T'LAST) mod 2;
    Found := T(L) = Key;
    while Bot <= Top and not Found loop
        Mid := (Top + Bot) mod 2;
        if T( Mid ) = Key then
            Found := true;
            L := Mid;
        elsif T( Mid ) < Key then
            Bot := Mid + 1;
        else
            Top := Mid - 1;
        end if;
    end loop;
end Binary_Search;

```

Slika 9.1 Program koji izvršava binarno pretraživanje [Izvor: NM SE321-2020/2021.]

KORACI KOJI SE IZVRŠAVAJU U BINARNOM PRETRAŽIVANJU

Prilikom binarnog pretraživanja izvršavaju se sledeći koraci:

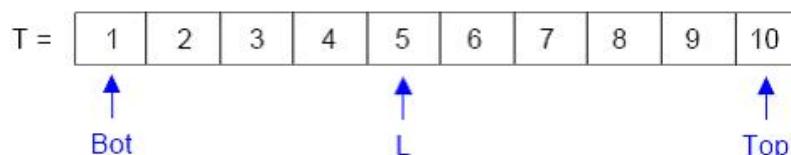
Sledi ilustracija koraka u pretraživanju.

Start: Key = 6 (Slika 2)

T =	1	2	3	4	5	6	7	8	9	10
-----	---	---	---	---	---	---	---	---	---	----

Slika 9.2 Korak: Start [Izvor: NM SE321-2020/2021.]

Iteracija 0 (Slika 3) :



Slika 9.3 Korak: Iteracija 0 [Izvor: NM SE321-2020/2021.]

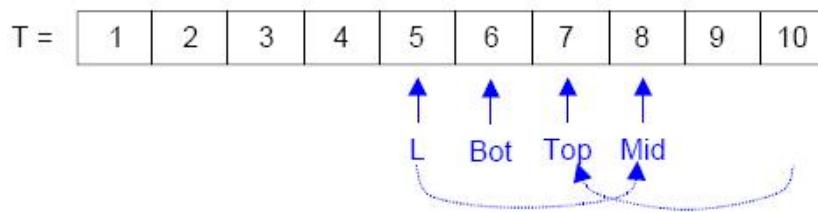
Iteracija 1 (Slika-4)

Slika 9.4 Korak: Iteracija 1 [Izvor: NM SE321-2020/2021.]

Sledeća ilustracija koraka u pretraživanju je:

- Key = 6
- T(Mid) = 5
- Found = FALSE

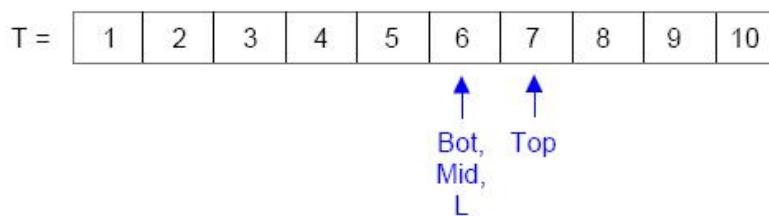
Iteracija 2 (Slika 5) :



Slika 9.5 Korak: Iteracija 2 [Izvor: NM SE321-2020/2021.]

- Key = 6
- T(Mid) = 8
- Found = FALSE

Iteracija 3 (Slika 6) :



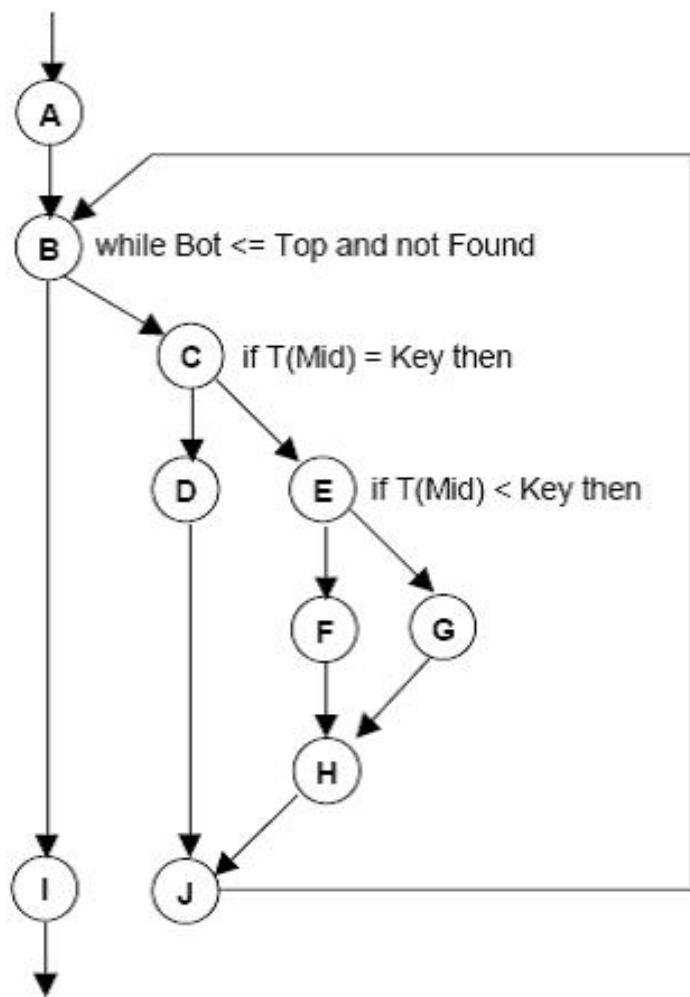
Slika 9.6 Korak: Iteracija 3 [Izvor: NM SE321-2020/2021.]

PROGRAM KOJI IZVRŠAVA BINARNO PRETRAŽIVANJE - NASTAVAK

Ako vrednosti, koje se porede, nisu iste, upoređuje se vrednost na sredini niza sa ključem.

- Key = 6
- T(Mid) = 6
- Found = TRUE
- L = 6
- Exit Program (Found = TRUE; L = 6)

Graf toka je prikazan na sledećoj slici 7:



Slika 9.7 Graf toka razmatranog programa [Izvor: NM SE321-2020/2021.]

PRIMER: POKRIVANJE PUTEVA - PREDLOZI

Da bi se dobio test slučaj, moguće je ići unazad kroz put, ali taj način zahteva određenu analizu.

Postoji jedan put koji omogućava 100% pokrivanje naredbi i grana (100% pokrivanje putanja nije moguće):

- A, B, C, E, F, H, J, B, C, E, G, H, J, B, C, D, J, B, I

Ovaj put odgovara primeru ilustrovanom napred.

U većini slučajeva, mala je verovatnoća da će biti izabran slučaj koji radi na ovaj način. Postoje kombinacije puteva, koje neće biti dostupne u okviru jednog testnog slučaja. Na primer, sledeći put daje 100% pokrivenost naredbi i grana na grafu toka:

- A, B, C, D, J, B, C, E, F, H, J, B, C, E, G, H, J, B, I

Međutim, kada pokušate da ovaj put realizujete pomoću testnog slučaja, uvidećete da to nije moguće. B, C, D, J iteracija na početku znači da se neće desiti više iteracija u okviru petlje. U stvari, prva iteracija u petlji neće nikad zadovoljiti uslov $T(\text{Mid}) = \text{Key}$, zato što je ova činjenica odbačena pre ulaska u petlju ($T(\text{L}) = \text{Key}$).

Uopšteno, bolje je krenuti sa jednostavnim test slučajevima, posebno za različite oblasti pokrivenosti. Na primer, početi sa jednom jednostavnom iteracijom kroz petlju, kao što je testiranje prolaska kroz srednju tačku jednom, pre pronalaženja ključa u nizu. Ulazi za ovaj test mogu biti specificirani veoma jednostavno, na primer koristeći sledeći put:

- A, B, C, E, G, H, J, B, C, D, J, B, I

Da bi se ostvarila 100% pokrivenost putanje, potrebno je posetiti još samo čvor F.

Postoji i sličan test slučaj, koji omogućava prolazak kroz čvor F:

- A, B, C, E, F, H, J, B, C, D, J, B, I

Mnogo je jednostavnije da se koristi više jednostavnijih test slučajeva, nego da se jednim test slučajem, koji ima veoma dugu putanju, pokuša dostizanje 100% pokrivenost.

ANALIZA PUTEVA ZA VREDNOSTI TEST SLUČAJA

Nije jednostavno odrediti izvodljive puteve i vrednosti za test slučajeve, kojima će se pokriti svi putevi prolaska kroz program.

Da bi se dobio test slučaj, moguće je ići unazad kroz put, ali taj način zahteva određenu analizu.

Međutim, ovaj način može se primeniti u slučaju kada nije moguće obezbediti pokrivanje određene putanje. Da bi se analizirala putanja, potrebno je obezbediti programske uslove, koji će usloviti prolazak kroz putanju koja se testira i zapamtitи uslove, potrebne za ostanak na njoj.

Pretpostavimo da predloženi put ne prolazi kroz petlju:

- A, B, I

Put kroz petlju se ne izvršava i čvor B, u uslovu za petlju, pada pri prvom pokušaju. Prema tome, može se zaključiti da uslov za petlju

- Bot \leq Top and not Found

ima vrednost FALSE.

To znači, da bi se izvršio ovaj put, moramo imati ulaze u test, čiji je rezultat:

- Bot $>$ Top – ulazni niz je prazan

Found – ključ je pronađen pri prvom pristupu, odnosno, ključ je u sredini niza.

U drugom slučaju, tester treba da napravi ulaze u test, koji zadovoljavaju uslov za praćenje puta. Međutim, što je put duži, to je teže napraviti ulaze u test.

Kao što se može videti, izvođenje vrednosti za test slučaj nije jednostavno, i ponekad je slično dibagovanju. Ipak, prolazeći kroz ovaj proces moguće je naići na problem, čak i pre izvršenja samo jednog test slučaja, za program koji se testira.

ANALIZA PUTEVA ZA VREDNOSTI TEST SLUČAJA - PREGLED PRIMERA ZA POKRIVANJE

*Da bi se kreirali ulazi, potrebni za kretanje određenim putem,
neophodno je određeno razumevanje programske logike.*

Nije jednostavno odrediti izvodljive puteve i vrednosti za test slučajeve, kojima će se pokriti svi putevi prolaska kroz program. *Da bi se kreirali ulazi, potrebni za kretanje određenim putem, neophodno je određeno razumevanje programske logike.* Jednostavnije je imati više manjih puteva, koji će zajedno omogućiti zahtevani nivo pokrivenosti.

Najčešće, pokrivanje naredbi, grana i putanja ne dobija se posle prve analize programa. Tipičnije je da se primene neki test slučajevi, predloženi kroz tehnike testiranja metodom crne kutije, a da se zatim analiziraju programi, kako bi se izveli dodatni test slučajevi, neophodni za korišćenje metode bele kutije.

Kod tehnika pokrivanja, postavlja se pitanje kreiranja dovoljnog broja test slučajeva. Na primer, nije pokriven slučaj u kome niz nema vrednost, koja je ista kao vrednost ključa. Nije ni razmotrena situacija kada je niz prazan, (Bot > Top). Takođe, petlje nisu pokrivene u svim detaljima.

▼ Poglavlje 10

Grupna vežba: Korišćenje CodeCover

PREDUSLOVI ZA KORIŠĆENJE ALATA CODECOVER

*Ova aplikacija je sistem koji se testira (System under test - SUT).
CodeCover ima instrumente za tok izvršavanja sistema koji testiramo, i
kako rezultati testa mogu biti vizuelno prikazani.*

Prilikom testiranja programa metodom bele kutije veoma je važno dobiti informaciju koliko detaljno smo izvršili testiranje konkretnog programa određenim tehnikama testiranja koje smo primenili. Ovde se daje primer alata **CodeCover** koji je namenjen testiranju Java programa metodama bele kutije i kojim će se testirati:

- **Pokrivenost naredbi** - da li je izvršena određena naredba
- **Pokrivenost odluka (grana)** - da li je izvršavanje programa prošlo određenom granom
- **Pokrivenost prostih uslova** (u smislu da li je tokom izvršavanja programa u datoj tački ispunjen uslov koji je napisan u programu - ne i njegova negacija)
- **Pokrivenost petlji** - posmatrana petlja se smatra pokrivenom ako je izvršeno nula, jedna i više od jedne iteracije.

U okviru ovog dokumenta ćemo prikazati korišćenje dodatka CodeCover za razvojno okruženje Eclipse. Koristićemo Java aplikaciju SimpleJavaApp preuzetu sa zvaničnog sajta ovog alata u formi Eclipse projekta. Link za preuzimanje je: <https://goo.gl/> Opisaćemo CodeCover instrumente i izvršavanje sistema koji testiramo, i kako rezultati testa mogu biti vizuelno

prikazani i prikazani u vidu izveštaja. Za korišćenje tutorijala je neophodno preuzeti i pokrenuti razvojno okruženje Eclipse. Eclipse se može preuzeti sa sledećeg linka:
<http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars1>

Nakon pokretanje razvojnog okruženja Eclipse treba odabratи stavku Install New Software iz menija Help. Zatim treba kliknuti na dugme Add koje pokreće dijalog za dodavanje novog repozitorijuma.

Podaci za ažuriranje Eclipse-a:

Name: CodeCover Update Site

URL: <http://codecover.org/>

PRIMER: programa koji želimo da testiramo - SimpleJavaApp:

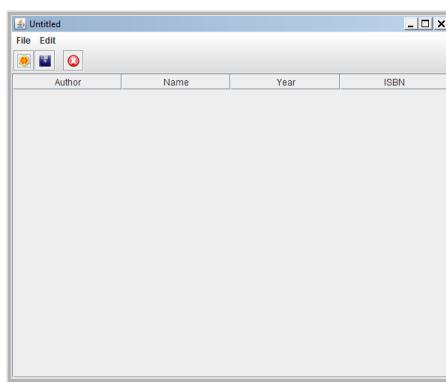
SimpleJavaApp je mala **Swing** aplikacija, koja se koristi za prikazivanje i editovanje liste knjiga. Otvorićemo Eclipse i importovati SimpleJavaApp u naš radni prostor.

Odaberimo opciju *Import* u meniju *File* i odaberimo *Existing Projects into Workspace* u kategoriji *General* i zatim pronađimo folder gde se nalazi aplikacija.

MERENJE POKRIVENOSTI - IZBOR KRITERIJUMA ZA POKRIVENOSTI KOJE ĆE SE KORISTITI

Potrebno je da dozvolimo korišćenje dodatka CodeCover za projekat koji smo importovali.

Pokretanjem aplikacije dobijamo sledeću formu:

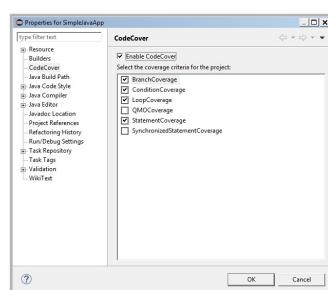


Slika 10.1 Forma ekrana pri pokretanju SimpleJavaApp programa [Izvor: NM SE321-2020/2021.]

Potrebno je da dozvolimo korišćenje dodatka CodeCover za projekat koji smo importovali. Otvorićemo opciju *Properties* na projektu i odabratim CodeCover kategoriju. Izaberimo checkbox kao što je prikazano na slici 2. Sada smo aktivirali CodeCover za projekat SimpleJavaApp. Takođe, potrebno je da odaberemo kriterijume za pokrivenosti koje će se koristiti. U našem slučaju odabrali smo četiri osnovna kriterijuma (Slika 2).

Merenje pokrivenosti: Odaberimo klase koje želimo da posmatramo. Otvorićemo *Package explorer* i u izvornom direktorijumu projekta SimpleJavaApp izabrati klase koje želimo da posmatramo.

Kliknimo desnim dugmetom miša i odaberimo *Use For Coverage Measurement* za svaku klasu koju želimo da označimo (istovremeno će se promeniti ikonice). U ovom primeru, sve klase projekta SimpleJavaApp su označene za praćenje pokrivenosti (slika 3).



Slika 10.2 Izbor kriterijuma za pokrivenosti koje će se koristiti [Izvor: NM SE321-2020/2021.]

Slika 10.3 Klase projekta SimpleJavaApp koje su označene za praćenje pokrivenosti [Izvor: NM SE321-2020/2021.]

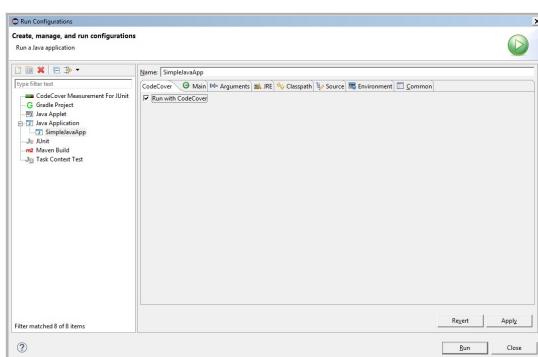
POKRETANJE APLIKACIJE SA CODECOVER

Da bismo koristili CodeCover, potrebno je da pokrenemo aplikaciju u tom režimu, tako što u dijalogu Run configuration naglasimo da koristimo taj dodatak.

Da bismo koristili CodeCover, potrebno je da pokrenemo aplikaciju u tom režimu, tako što u dijalogu *Run configuration* naglasimo da koristimo taj dodatak. Potrebno je u tabu CodeCover odabrati opciju *Run with CodeCover*, kao što je prikazano na slici 4.

Izvršavanje: Postoji nekoliko načina da se izvrši sistem koji se testira. Možemo izabrati pokretanje aplikacije kao i obično, i u tom slučaju svi izmereni podaci se prikupljaju u jedan test.

Ako nakon pokretanja izaberemo sledeće opcije u aplikaciji:
File -> New BookEdit -> Delete -> YesFile -> Quit -> No



Slika 10.4 Dijalog Run configuration [Izvor: NM SE321-2020/2021.]

CodeCover će nam u okviru programskog koda prikazati koje instrukcije, naredbe i grananja su pokrivene, a koje nisu, kao na slici 5:

Slika 10.5 Prikaz koje instrukcije, naredbe i grananja su pokrivene [Izvor: NM SE321-2020/2021.]

Druga varijanta je da koristimo *Live Notification* funkciju i snimamo pojedinačne testove. Treća varijanta je da koristimo postojeće JUnit grupe testova da definišemo naše testove.

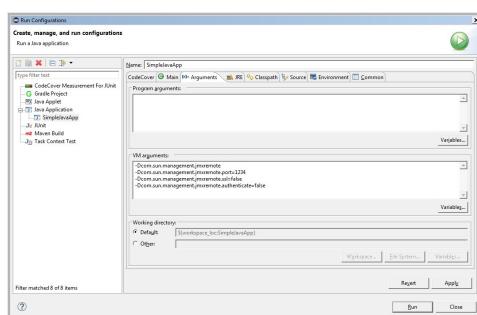
Obično izvršavanje: Potrebno je samo da pokrene postojeću konfiguraciju. CodeCover se izvršava u pozadini. Pokrenimo testove i nakon završetka rada sa aplikacijom, merenja će automatski biti sačuvana kao testovi u test sesiji nazvanoj "eclipsersun". Merenja možemo pratiti u prozoru u Eclipsu ili kreiranjem HTML izveštaja iz tih podataka.

LIVE NOTIFICATION IZVRŠAVANJE

Da bismo koristili ovu opciju, potreban je još jedan dodatni korak, a to je da za pokretanje programa dodamo parametre

Live Notification izvršavanje: Da bismo koristili ovu opciju, potreban je još jedan dodatni korak, a to je da u tab sa argumentima konfiguracije za pokretanje programa dodamo sledeće parametre kao VM argumente:

- *Dcom.sun.management.jmxremote - Dcom.sun.management.jmxremote.port=1234 -*
- Dcom.sun.management.jmxremote.ssl=false -*
- Dcom.sun.management.jmxremote.authenticate=false*



Slika 10.6 Pokretanje Live Notification izvršavanje [Izvor: NM SE321-2020/2021.]

Pokrenimo aplikaciju sa ovom konfiguracijom, otvoříme Live Notification prozor u Eclipse. Upisaćemo *localhost* za hostname i 1234 za port. Kliknimo na Connect dugme i Live notification će sada biti aktiviran kao na slici 7.

Slika 10.7 Aktiviranje Live notification opcije [Izvor: NM SE321-2020/2021.]

Upišimo ime u test i krenućemo da ga izvršavamo nakon što kliknemo na dugme *Start Test Case*. Na primer izvršićemo brisanje knjige. Na kraju testa, pritisnimo dugme *End Test Case*.

Možete snimiti i više testova ponavljanjem ovih koraka. Konačno, da završimo snimanje testova, izabratimmo opciju *Finish Test Session*. Ne moramo da koristimo opciju *Download Coverage Log File*, zato što SimpleJavaApp nije Web aplikacija. Nakon izvršavanja testova, merenja će automatski biti sačuvana.

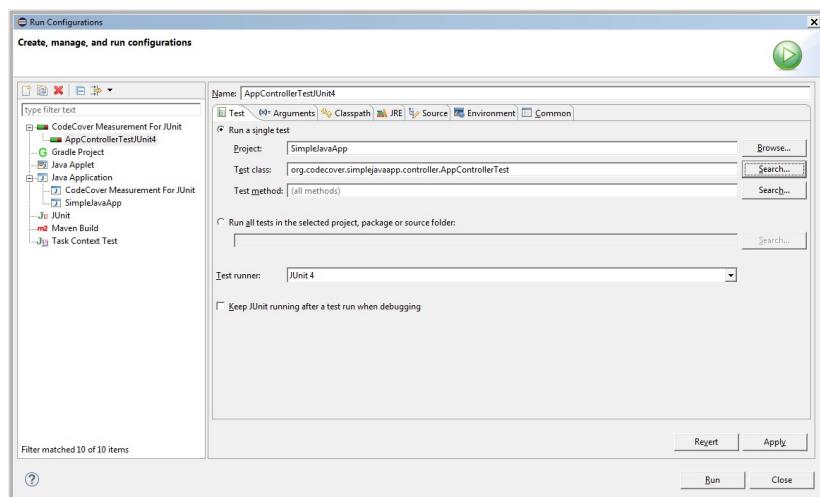


Slika 10.8 Izbor Start Test Case opcije [Izvor: NM SE321-2020/2021.]

JUNIT IZVRŠAVANJE

Da bismo koristili postojeće grupe testova, potrebno je da kreiramo novu konfiguraciju CodeCover Measurement For JUnit.

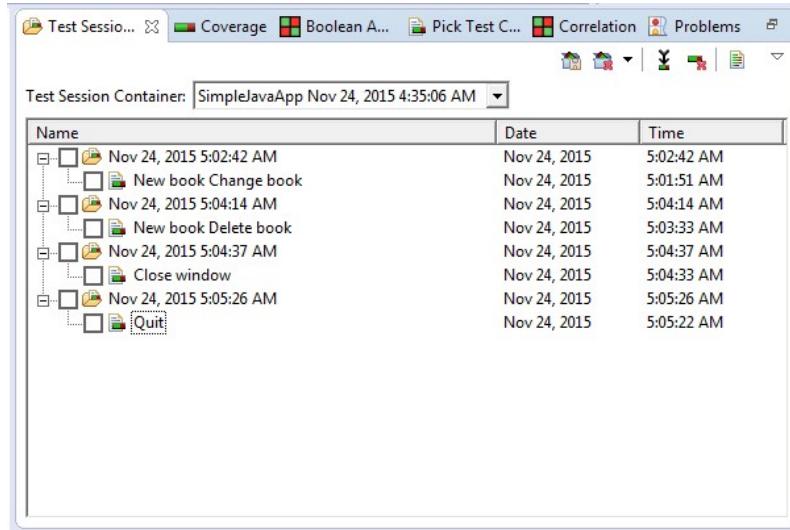
JUnit izvršavanje : Da bismo koristili postojeće grupe testova, potrebno je da kreiramo novu konfiguraciju *CodeCover Measurement For JUnit*. Kao što je prikazano na slici 9, neophodno je da izaberemo klasu koja sadrži naše JUnit test slučajeve ili *test suite*. Za izvršavanje testova možemo izabrati JUnit 3 ili JUnit 4 (opcija: *test runner*), zavisno od toga na čemu se zasniva naša postojeća grupa testova. Nakon toga ćemo pokrenuti aplikaciju koju testiramo, da bismo pratili merenja. Nakon izvršavanja, merenja će automatski biti zabeležena u test sesiji nazvanoj *eclipserun*, koja čuva sve testove koji su definisani u našoj grupi testova.



Slika 10.9 Nova konfiguracija CodeCover Measurement For JUnit [Izvor: NM SE321-2020/2021.]

Pregled izmerenih podataka u Eclipse-u: U ovoj sekciji su opisane osnovne odlike koje CodeCover pruža.

Test Sessions View: Ovaj pogled prikazuje test sesije i testove u grupama. Podržane su sledeće opcije: select, deselect, rename, delete i merge. Drugi pogledi koriste samo test slučajeve u prikazivanju.

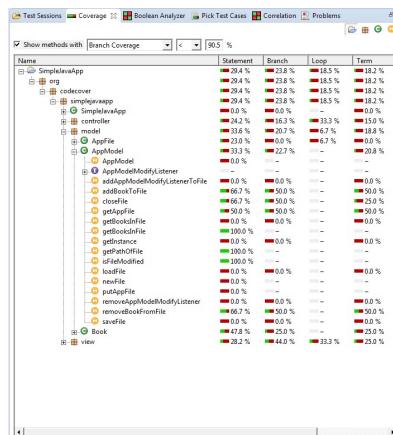


Slika 10.10 Test sesije i testove u grupama [Izvor: NM SE321-2020/2021.]

COVERAGE I CORRELATION VIEW

U ovom pogledu možemo videti pokrivenost određenih delova aplikacije koju smo testirali.

Coverage View: U ovom pogledu možemo videti pokrivenost određenih delova aplikacije koju smo testirali. Kao što je prikazano na slici 11, svaka metrika ima svoju posebnu kolonu.



Slika 10.11 Coverage View [Izvor: NM SE321-2020/2021.]

Correlation View: Ovaj pogled se koristi da uporedi test slučajeve sa ostalima. Možemo postaviti cursor miša iznad nekog bloka u matrici i videti koliko jedan test pokriva iste delove koda u odnosu na drugi test. Stablo prikazuje test koji u potpunosti pokrivaju delove koda koji drugi testovi pokrivaju, u hijerarhiji. Takođe (Slika 12), možemo eksportovati podatke iz matrice u fajl, koji je kompatibilan sa mnogim aplikacijama (Excel i druge).

Slika 10.12 Correlation View [Izvor: NM SE321-2020/2021.]

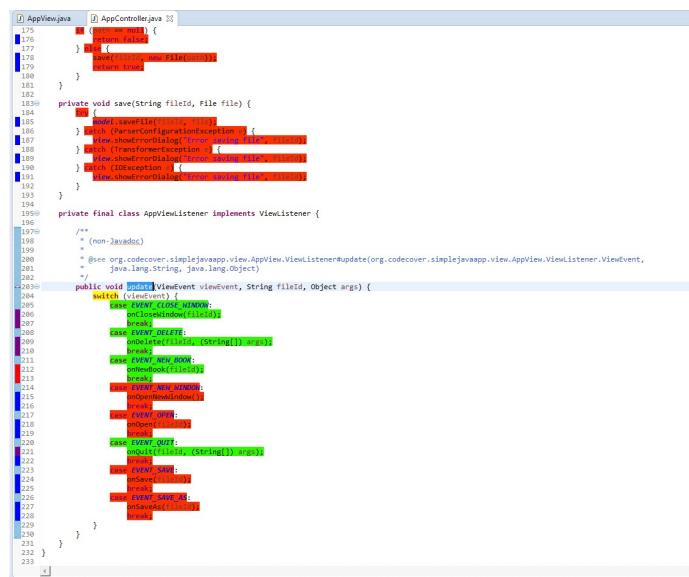
Boolean Analyzer: Ovim pogledom možemo zadati uslov i videti pokrivenost stanja. Potrebno je odabrat ključnu reč uslova u kodu, kliknuti desnim dugmetom miša i izabrati opciju "Analyze Term" u meniju, koja će automatski postaviti taj uslov u Boolean analizator.

Slika 10.13 Boolean Analyzer [Izvor: NM SE321-2020/2021.]

CODE HIGHLIGHTING AND HOT PATH - OPCIJA

Izvorni kod može da se oboji na osnovu pokrivenosti. Taj prikaz je dat samo ukoliko izvorni kod nije menjan od poslednjeg merenja pokrivenosti.

Code Highlighting & Hot Path: Izvorni kod može da se oboji na osnovu pokrivenosti. Taj prikaz je dat samo ukoliko izvorni kod nije menjan od poslednjeg merenja pokrivenosti. Kao što je prikazano na slici 14, metode `onQuit(...)` i `onSaveAs(...)` se ne izvršavaju u vreme merenja pokrivenosti. Putanja izvornog fajla koja se više izvršava je prikazan na levoj strani editora. Iskazi koji su češće izvršavani, postaju više crveni. Na primer metoda `onNewBook(...)` se izvršava mnogo češće nego druge metode, pa je crvene boje.



```

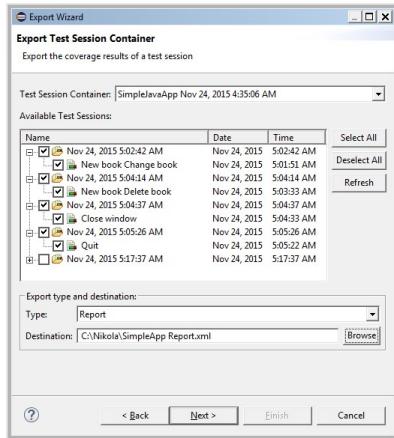
175     }
176     private void save(String fileId, File file) {
177         ...
178     }
179     private void save(String fileId, File file) {
180         ...
181     }
182     private void save(String fileId, File file) {
183         ...
184     }
185     private void save(String fileId, File file) {
186         ...
187     }
188     private void save(String fileId, File file) {
189     }
190     private void save(String fileId, File file) {
191     }
192     ...
193     private final class AppViewListener implements ViewListener {
194         ...
195         /**
196          * @non-Javadoc
197          * @see org.codecover.simplejavapp.view.AppView.ViewListener#update(org.codecover.simplejavapp.view.AppView.ViewEvent,
198          * @param event)
199          */
200         public void handle(ViewEvent event, String fileId, Object args) {
201             switch (event.type) {
202                 case ViewEvent.ON_CLOSE:
203                     ...
204                     ...
205                     ...
206                     ...
207                     ...
208                     ...
209                     ...
210                     ...
211                     ...
212                     ...
213                     ...
214                     ...
215                     ...
216                     ...
217                     ...
218                     ...
219                     ...
220                     ...
221                     ...
222                     ...
223                     ...
224                     ...
225                     ...
226                     ...
227                     ...
228                     ...
229             }
230         }
231     }
232 }

```

Slika 10.14 Code Highlighting & Hot Path - opcija [Izvor: NM SE321-2020/2021.]

Pick Test Case View: Ovaj pogled prikazuje koji testovi pokrivaju koji deo koda, na primer koji testovi izvršavaju metod `onNewWindow()`. On prati ono što je selektovano u trenutno otvorenom editoru i prikazuje listu testova. Takođe se prikazuje i test sesija.

Export Report: Možemo eksportovati podatke kao HTML izveštaj. Odaberite opciju `Export` u `File` meniju u Eclipse-u i odaberite `Coverage Result Export` u kategoriji `CodeCover`. Zatim izaberimo skup koji sadrži testove koje želimo da eksportujemo. Postavite tip za eksportovanje na `Report`. Potrebno je da odaberemo i destinaciju za izveštaje, slično kao što je prikazano na slici 15.

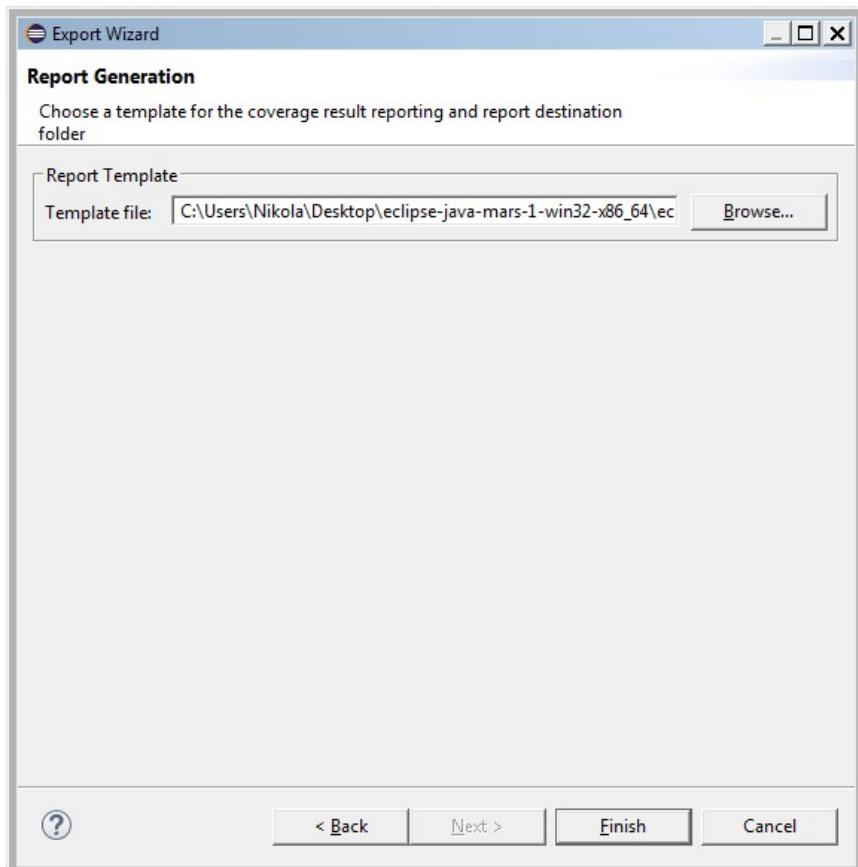


Slika 10.15 Export Report [Izvor: NM SE321-2020/2021.]

IZABOR ŠABLONA ZA IZVEŠTAJ

Pritiskom na Next dugme (slika 15) da izaberemo šablon za izveštaj.

Pritisnimo Next dugme (slika 16) da izaberemo šablon za izveštaj. Šablon za izveštaj se nalazi u folderu `plugins\org.codecover.report.html_1.0.1.2`.



Slika 10.16 Izabor šablona za izveštaj [Izvor: NM SE321-2020/2021.]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Vreme potrebno za grupnu vežbu je 45 minuta.

✓ Poglavlje 11

Vežba za samostalni rad

ZADATAK ZA SAMOSTALNI RAD

Tekst zadatka za samo stalni rad

Zadatak 1. Pronaći proizvoljni kod aplikacije rađene za potrebe projekta ili primer programskog koda sa interneta. Programski kod treba da sadrži: naredbe, odlučivanje (grananje), definisane proste uslove i petlje. Ukoliko primer ne pokriva navedene stavke potrebno je doraditi programski kod tako da sadrži sve navedeno.

Vreme potrebno za izradu vežbe (zadatka 1) 30 minuta.

Zadatak 2. Pokrenuti CodeCoder i prikazati:

1. Testiranje pokrivenosti naredbi - izvršavanje svih naredbi u programskom kodu
2. Pokrivenost odluka (grana) - koje određene grane se koriste prilikom izvršavanja programa
3. Pokrivenost prostih uslova - da li se tokom izvršavanja programa u datoj tački ispunjava uslov koji je napisan u programu - ne i njegova negacija)
4. Pokrivenost petlji - posmatrana petlja se smatra pokrivenom ako je izvršeno nula, jedna i više od jedne iteracije.
5. Generisati izveštaj i u dokumentu opisati dobijene rezultate.

Vreme potrebno za izradu vežbe (zadatka 2) 60 minuta.

✓ Poglavlje 12

Domaći zadatak

ŠESTI DOMAĆI ZADATAK

Tekst šestog domaćeg zadatka

Za odabranu aplikaciju u okviru prethodnih domaćih zadataka primeniti sledeće:

Testirati aplikaciju korišćenjem CodeCodera tako da se prikaže primer testiranja:

1. Pokrivenost naredbi
2. Pokrivenost odluka (grana)
3. Pokrivenost prostih uslova
4. Pokrivenost petlji

Za izvršeno testiranje generisati izveštaj.

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ06-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ06.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

❖ White-box ili struktурно testiranje

TESTIRANJE SOFTVERA -TEHNIKA BELE KUTIJE (STRUKTURNO TESTIRANJE)

Metod bele kutije koristan je za analizu i testiranje određenih puteva u programu pomoću test slučajeva.

Metod bele kutije koristan je za analizu i testiranje određenih puteva u programu pomoću test slučajeva.

U većini situacija, testiranje metodom bele kutije znači testiranje velikog broja test slučajeva. Razlog je što testovi treba da pokriju veliki broj puteva izvršavanja kroz program.

Testiranje metodom bele kutije uobičajeno se koristi u kombinaciji sa testiranjem metodom crne kutije. Posle primene metode crne kutije, radi se analiza nivoa pokrivanja pri testiranju metodom bele kutije. To ukazuje na oblasti u kojima je testiranje metodom crne kutije možda bilo neodgovarajuće, što se odnosi pre svega na puteve u programu. Mogu biti predloženi dodatni test slučajevi, kako bi se podigao potreban nivo pokrivenosti.

Svaka moguća izvršna putanja softverskog modula ima odgovarajuću putanju od ulaznog do izlaznog čvora kontrolnog grafa toka modula. To predstavlja osnovu metodologije struktornog testiranja. Da bi se za određeni program iscrtao graf toka kontrole potrebno je numerisati sve iskaze programa.

Numerisani iskazi predstavljaju se čvorovima u grafu toka kontrole. Između dva čvora postoji grana ukoliko, po izvršenju iskaza koji je pridružen izvorišnom čvoru, kontrola može da se prenese na iskaz koji je pridružen odredišnom čvoru.

LITERATURA ZA LEKCIJU 06

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. dr.sc. Tihana Galinac Grbac, Testiranje programskog proizvoda. (Link: <http://www.riteh.uniri.hr/>)
4. White Box Software Testing, <http://www.testingeducation.org/BBST/bugadvocacy/BugAdvocacy2008.pdf>

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
3. British Computer Society, Standard for Software Component Testing, Date: 27 April 2001.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <https://www.softwaretestinghelp.com/white-box-testing-techniques-with-example/>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Testiranje softvera – tehničke sive kutije, ne funkcionalno testiranje

Lekcija 07

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 07

TESTIRANJE SOFTVERA – TEHNIKE SIVE KUTIJE, NE FUNKCIONALNO TESTIRANJE

- ✓ Testiranje softvera – tehnike sive kutije, ne funkcionalno testiranje
- ✓ Poglavlje 1: White box testiranje toka podataka
- ✓ Poglavlje 2: Grey box - testiranje sive kutije
- ✓ Poglavlje 3: Regresiono testiranje
- ✓ Poglavlje 4: Nefunkcionalno testiranje
- ✓ Poglavlje 5: Upravljanje rizikom
- ✓ Poglavlje 6: Grupna vežba: Testiranje toka podataka
- ✓ Poglavlje 7: Domaći zadatak
- ✓ Strukturno testiranje i tehnike sive kutije (hibridno testiranje)

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Greške i nemamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni metodom struktturnog testiranja.

U prethodnoj lekciji smo istakli da white box ili struktorno testiranje obuhvata ispitivanje interne strukture programa ili sistema. Testni podaci se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji. Tester poznaje internu strukturu i logiku programa, isto kao što mehaničar zna unutrašnji mehanizam automobila. Specifični primeri koji spadaju u ovu kategoriju uključuju testiranje izraza, gransko testiranje, testiranje uslova i sl. Prednost ovog pristupa je da je potpun i da se fokusira na proizvedeni kod. Greške i nemamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni. Pored tehnika bele kutije koji su navedeni u prethodnom predavanju, postoji još jedna tehnika poznata kao testiranje toka podataka koja je opisana u ovom predavanju. Ona je fokusirana na testiranje protoka podataka tj. na tačke u kojima se promenljivim dodeljuju neke vrednosti i na tačke u kojima se ove vrednosti koriste.

U predavanju se opisuje i testiranje po metodi sive kutije, koja koristi kombinaciju testiranja po metodi crne kutije i bele kutije. Postoji nekoliko tehnika testiranja metodom sive kutije od kojih je u predavanju posebno istaknuta tehnika regresionog testiranja kojim se osigurava da će softverski proizvod dobro funkcionišati i nakon dodavanja nove funkcionalnosti, ispravci grešaka ili bilo kojih promena postojećih funkcija.

Pošto smo utvrdili da sistem tačno izvršava funkcije navedene u zahtevima, prelazimo na način na koji se te funkcije izvršavaju tj. na nefunkcionalne zahteve sistema. U nefunkcionalno testiranje spada testiranje performansi, testiranje kompatibilnosti i testiranje upotrebljivosti o čemu se takođe govorи u ovoj lekciji.

✓ Poglavlje 1

White box testiranje toka podataka

ŠTA PREDSTAVLJA OVAJ NAČIN TESTIRANJA?

Predstavlja proces prikupljanja informacija o tome kako promenljive protiču kroz program. Fokusira na tačke u kojima se promenljivim dodeljuju vrednosti i kojima se ove vrednosti koriste

Pored tehnika testiranja pokrivenosti koda o kojima je bilo reči u prethodnom predavanju kao što su:

1. Pokrivenost putanja
2. Pokrivenost kontrole toka
3. Pokrivenost grananja
4. Pokrivenost naredbi
5. Pokrivenost odluka

postoji još jedna vrlo značajna tehnika testiranja bele kutije poznata kao Testiranje toka podataka (engl.Data Flow Testing). Testiranje toka podataka se koristi za analizu tokova podataka u okviru programu. To je proces prikupljanja informacija o tome kako promenljive (podaci) protiču kroz program. Predstavljaju pokušaj da se dobije informacije o svakoj pojedinačnoj tački procesa.

Testiranje toka podataka je grupa strategija testiranja za ispitivanje kontrolnog toka programa u cilju ispitivanja redosleda promenljivih prema redosledu događaja. Uglavnom se fokusira na tačke u kojima se promenljivim dodeljuju neke vrednosti i na tačke u kojima se ove vrednosti koriste čime se može testirati protok podataka.

Testiranje toka podataka koristi graf kontrolnog toka za otkrivanje nelogičnih stvari koje mogu prekinuti tok podataka. Anomalije u tokovima podataka se otkrivaju u trenutku ostvarivanja veze između vrednosti i promenljivih:

- ako se promenljive koriste bez inicijalizacije
- ako se inicirana promenljive ni jedanput ne koristi

PRIMER: DEFINISNJE PUTANJA

Uz datog primera koda se vidi da se ne mogu pokriti sve naredbe u jednoj putanji već se za pokrivanje koda koriste dve putanje.

Objasnimo to na sledećem primeru:

```

1. read x;
2. If(x>0)          (1, (2, t), x), (1, (2, f), x)
3. a= x+1           (1, 3, x)
4. if (x<=0) {      (1, (4, t), x), (1, (4, f), x)
5.   if (x<1)       (1, (5, t), x), (1, (5, f), x)
6.     x=x+1; (go to 5) (1, 6, x)
else
7.   a=x+1           (1, 7, x)
8. print a;          (6,(5, f)x), (6,(5,t)x)
                      (6, 6, x)
                      (3, 8, a), (7, 8, a).

```

Slika 1.1 Kod za testiranje [Izvor: NM SE321-2020/2021.]

U ovom kodu imamo ukupno 8 naredbi i izabraćemo putanje koje pokrivaju svih 8 naredbi. Kao što se vidi u kodu, ne možemo pokriti sve naredbe u jednoj putanji jer ako je tačna naredba 2, neće biti obuhvaćene naredbe 4, 5, 6, 7, a ako je tačna naredba 4, tada neće biti obuhvaćene naredbe 2 i 3. Dakle, koristimo dve putanje za pokrivanje svih naredbi:

```

1. x= 1
Path - 1, 2, 3, 8
Izlaz = 2
2. Set x= -1
Path = 1, 2, 4, 5, 6, 5, 6, 5, 7, 8
Output = 2

```

Da bi se vrednost x prvo postavila na 1, dolazi se na korak 1 za čitanje i dodeljivanje vrednosti x (**uzeli smo 1 na putanji**), a zatim prelazimo na naredbu 2 (if $x > 0$) (**uzeli smo 2 na putanji**) što je tačno i dolazimo do naredbe 3 ($a = x + 1$) (**uzeli smo 3 na putanji**), dolazimo do naredbe 8 gde se štampa vrednost x (izlaz je 2).

Da bi smo postavili vrednost x na -1 prvo, u koraku 1 čitamo i dodeljujemo vrednosti za x (**uzeli smo 1 na putanji**), a zatim idemo na korak 2 koji je pogrešan (false), jer x nije veći od 0 ($x = -1$). Zbog neispunjena uslova, nećemo doći na naredbu 3 već ćemo direktno preći na naredbu 4 (**uzeli smo 4 na putanji**), uslov u naredbi 4 ($x \leq 0$) je tačan (**thrut**) jer je x manji od 0, a zatim dolazimo do naredbe 5 ($x < 1$) (**uzeli smo 5 u na putanji**) što je takođe tačno, pa ćemo doći do naredbe 6 ($x = x + 1$) (**uzeli smo 6 na putanji**), ovde je x uvećan za 1, tako da je $x = -1 + 1$ tj. $x = 0$ (vrednost za x je sada postala 0). Sada idemo do naredbe 5 ($x < 1$) (**uzimamo 5 na putanji**) sa vrednošću 0 i kako je 0 manje od 1, uslov je ispunjen. Idemo na naredbu 6 ($x = x + 1$) (**uzimamo 6 na putanji**) tako da je ($x = 0 + 1$ tj. $x = 1$). Ovde x postaje 1 i ponovo idemo na naredbu 5 ($x < 1$) (**uzimamo 5 na putanji**) i sada kako 1 nije manje od 1, uslov nije ispunjen pa dolazimo do grane else - naredbe 7 ($a = x + 1$) (**uzimamo 7 na putanji**) gde je vrednost za x jednaka 1 i tu vrednost dodeljujemo promenljivoj a ($a = 2$). Na kraju dolazimo da naredbe 8 i štampamo vrednost (izlaz je 2).

PRIMER: DEFINISANJE ASOCIJACIJA

U asocijacijama listamo sve definicije sa svim njihovim korišćenjima, ovde je dat prvi deo liste asocijacija

U asocijacijama listamo sve definicije sa svim njihovim korišćenjima:

(1, (2, f), x), (1, (2, t), x), (1, 3, x), (1, (4, t), x), (1, (4, f), x), (1, (5, t), x), (1, (5, f), x), (1, 6, x),
 (1, 7, x), (6,(5, f)x), (6,(5,t)x), (6, 6, x), (3, 8, a), (7, 8, a).

Kako napraviti asocijacije u testiranju tokova podataka?

1 read x;	
2 If(x>0)	(1, (2, t), x), (1, (2, f), x)
3 a= x+1	(1, 3, x)
4 if (x<=0) {	(1, (4, t), x), (1, (4, f), x)
5 if (x<1)	(1, (5, t), x), (1, (5, f), x)
6 x=x+1; (go to 5)	(1, 6, x)
else	
7 a=x+1	(1, 7, x)
8 print a;	(6,(5, f)x), (6,(5,t)x) (6, 6, x) (3, 8, a), (7, 8, a).

Slika 1.2 Kod za testiranje [Izvor: NM SE321-2020/2021.]

- **(1, (2, t), x), (1, (2, f), x)** - ova asocijacija se pravi sa naredbom 1 (read x;) i naredbom 2 (If (x>0)) tako da je x definisano na liniji broj 1 a koristi se na liniji broj 2, dakle x je promenljiva. Naredba 2 je logička i ona može biti ispunjena (**thru**) ili neispunjena (**false**) zbog čega je asocijacija definisana na dva načina: jedan je (1, (2, t), x) za ispunjen uslov a druga je (1, (2, f), x) za neispunjen uslov.
- **(1, 3, x)** - asocijacija je napravljena između naredbe 1 (read x;) i naredbe 3 (a= x+1) gde je x definisano u naredbi 1 a koristi se u naredbi 3. *Ovo je primer korišćenja izračunavanja (engl. computation use)*
- **(1, (4, t), x), (1, (4, f), x)** - ova asocijacija je napravljena između naredbe 1 (read x;) i naredbe 4 (If(x<=0)) gde je x definisano na liniji broj 1 a koristi se na liniji broj 4, pri čemu je x promenljiva. Naredba 4 je logička i ona može biti ispunjena (**thru**) ili neispunjena (**false**) zbog čega je asocijacija definisana na dva načina: (1, (4, t), x) za slučaj kada je ispunjena a drugi je (1, (4, f), x) za neispunjen uslov.
- **(1, (5, t), x), (1, (5, f), x)** - ova asocijacija je napravljena između naredbe 1 (read x;) i naredbe 5 (if (x<1)) gde je x definisano na liniji broj 1 a koristi se na liniji broj 5, pri čemu je x promenljiva. Naredba 5 je logička, i ona može biti ispunjena (**thru**) ili neispunjena (**false**) zbog čega je asocijacija definisana na dva načina: (1, (5, t), x) za slučaj kada je ispunjena a drugi je (1, (5, f), x) za neispunjen uslov.
- **(1, 6, x)** - ova asocijacija je napravljena između naredbe 1 (read x;) i naredbe 6 (x=x+1), x je definisano u naredbi 1 a koristi se u naredbi 6. Ovo je primer korišćenja izračunavanja (engl. *computation use*)
- **(1, 7, x)** - ova asocijacija je napravljena između naredbe 1 (read x;) i naredbe 7 (a=x+1), x je definisano u naredbi 1 a koristi se u naredbi 7. *Ovo je primer izračunavanja (engl. computation use).*

PRIMER: DEFINISANJE ASOCIJACIJA - NASTAVAK PRIMERA

U asocijacijama listamo sve definicije sa svim njihovim korišćenjima, ovde je dat drugi deo liste asocijacija

1 read x;	
2 If($x > 0$)	(1, (2, t), x), (1, (2, f), x)
3 $a = x + 1$	(1, 3, x)
4 if ($x \leq 0$) {	(1, (4, t), x), (1, (4, f), x)
5 if ($x < 1$)	(1, (5, t), x), (1, (5, f), x)
6 $x = x + 1$; (go to 5)	(1, 6, x)
else	
7 $a = x + 1$	(1, 7, x)
8 print a;	(6, (5, f)x), (6, (5, t)x) (6, 6, x)
	(3, 8, a), (7, 8, a).

Slika 1.3 Kod za testiranje [Izvor: NM SE321-2020/2021.]

- **(6, (5, f) x), (6, (5, t) x)** - ova asocijacija je napravljena između naredbe 6 ($x = x + 1$;) i naredbe 5 (if ($x < 1$)) gde je x definisano na liniji broj 6 a koristi se na liniji broj 5. Naredba 5 je logička, i ona može biti ispunjena (thru) ili neispunjena (false) zbog čega je asocijacija definisana na dva načina: (6, (5, f) x) za slučaj kada je ispunjena a drugi je (6, (5, t) x) za neispunjen uslov. *Ovo je primer predikatnog korišćenja (engl. predicted use)*
- **(6, 6, x)** - ova asocijacija je napravljena između naredbe 6 koja koristi vrednost promenljive x a tada definiše novu vrednost za x
 $x = x + 1$
 $x = (-1 + 1)$
Naredba 6 koristi promenljivu x koja je -1 i tada definiše novu vrednost za x [$x = (-1 + 1) = 0$] koja je 0.
- **(3, 8, a)** - ova asocijacija je napravljena između naredbe 3 ($a = x + 1$) i naredbe 8 gde je promenljiva definisana u naredbi 3 a koristi se naredbi 8.
- **(7, 8, a)** - ova asocijacija je napravljena između naredbe 7 ($a = x + 1$) i naredbe 8 gde je promenljiva definisana u naredbi 7 a koristi se u naredbi 8.

PRIMER: DEFINICIJE PROMENLJIVIH

Definicija promenljive je pojavljivanje promenljive kada se za promenljivu vezuje odgovarajuća vrednost.

Sledeći zadatak je da sve asocijacije grupišemo u definicije sledećih kategorija: c-use pokrivenost, p-use pokrivenost, c-use some p-use pokrivenost, p-use some c-use pokrivenost.

Definicija promenljive je pojavljivanje promenljive kada se za promenljivu vezuje odgovarajuća vrednost. Za dati kod:

```
1 read x;  
2 If(x>0) (1, (2, t), x), (1, (2, f), x)  
3 a= x+1 (1, 3, x)  
4 if (x<=0) { (1, (4, t), x), (1, (4, f), x)  
5 if (x<1) (1, (5, t), x), (1, (5, f), x)  
6 x=x+1; (go to 5) (1, 6, x)  
else  
7 a=x+1 (1, 7, x)  
8 print a; (6,(5, f)x), (6,(5,t)x)  
(6, 6, x)  
(3, 8, a), (7, 8, a).
```

Slika 1.4 Kod za testiranje [Izvor: NM SE321-2020/2021.]

Asocijacije koje sadrže definicije kategorije Predicate use (p-use), Computation use (c-use) su:

(1, (2, f), x), (1, (2, t), x), (1, 3, x), (1, (4, t), x), (1, (4, f), x), (1, (5, t), x), (1, (5, f), x), (1, 6, x),
(1, 7, x), (6, (5, f), x), (6,(5,t), x), (6, 6, x), (3, 8, a), (7, 8, a)

U gornjem kodu, vrednost se vezuje za promenljivu u prvoj naredbi i zatim počinje da teče.

- If($x > 0$) je naredba 2 sa kojom je povezana vrednost za x
 - Asocijacija za naredbu 2 je (1, (2, f), x), (1, (2, t.), x)
- a= x+1 je naredba 3 povezana sa vrednošću x
 - Asocijacija naredbe 3 je (1, 3, x)

Sve definicije pokrivaju

(1, (2, f), x), (6, (5, f) x), (3, 8, a), (7, 8, a).

C-USE, P-USE, C-USE SOME P-USE, P-USE SOME C-USE POKRIVENOST U TESTIRANJU TOKA PODATAKA

Nakon prikupljanja ovih grupa tester može videti sve upotrebљene naredbe i promenljive i one koje nisu upotrebљene treba da izbaci iz koda.

Ako se promenljiva koristi da bi se odlučilo o izvršenju putanje, takvo korišćenje se naziva predikatno korišćenje (Predicate use ili p-use). U kontrolnom toku naredbi postoje dve.

Primer: Naredba 4 if ($x \leq 0$) je primer predikatnog korišćenja jer ona može biti predikat za istinito (engl. thru) ili pogrešno (engl. false)

Ako se vrednost promenljive koristi za izračunavanje vrednosti izlaza ili definisanje druge promenljive kakvo korišćenje se naziva **korišćenje računanja** (Computation use ili c-use).

Primeri c-use su:

- naredba 3 $a = x + 1 (1, 3, x)$
- naredba 7 $a = x + 1 (1, 7, x)$
- naredba 8 print a (3, 8, a), (7, 8, a).

jer se vrednost x koristi za izračunavanje vrednosti a koja se koristi kao izlaz

Sve c-use pokrivenosti

(1, 3, x), (1, 6, x), (1, 7, x), (6, 6, x), (6, 7, x), (3, 8, a), (7, 8, a).

Sve c-use some p-use pokrivenosti

(1, 3, x), (1, 6, x), (1, 7, x), (6, 6, x), (6, 7, x), (3, 8, a), (7, 8, a).

Sve p-use some c-use pokrivenosti

(1, (2, f), x), (1, (2, t), x), (1, (4, t), x), (1, (4, f), x), (1, (5, t), x), (1, (5, f), x), (6, (5, f), x), (6, (5, t), x), (3, 8, a), (7, 8, a).

Nakon prikupljanja ovih grupa, (ispitivanjem svake tačke da li se promenljiva koristi najmanje jednom ili ne) tester može videti sve upotrebljene naredbe i promenljive. Naredbe i promenljive koje se ne koriste, ali postoje u kodu, uklanaju se iz koda.

✓ Poglavlje 2

Grey box - testiranje sive kutije

ŠTA PRESTAVLJA METODA TESTIRANJA SIVE KUTIJE?

Kombinaciju testiranja crne i bele kutije, jer dizajniranje test slučajeva uključuje pristup internom kodu (bela kutija) a testiranje se vrši na nivou funkcionalnosti (crna kutija)

Testiranja sive kutije (engl. **grey box**) je metoda testiranja softverskih sistema sa delimičnim poznavanjem njegove interne strukture. To je kombinacija testiranja crne kutije i bele kutije, jer dizajniranje test slučajeva uključuje pristup internom kodu kao u slučaju testiranja bele kutije, a testiranjese vrši na nivou funkcionalnosti kao kod testiranja crne kutije.

Ovo testiranje nije "Crna kutija" zato što tester zna neke interne radnje softvera pod testom. U ovom testiranju, tester sprovodi ograničen broj scenarija na osnovu interne strukture softvera. U preostalom delu ovog testiranja, koristi pristup "crne kutije" u primenjivanju inputa na softver i prati izlaze.

Testiranje sive kutije ne znači da tester mora dizajnirati test slučajeve iz izvornog koda. Za izvođenje ovog testiranja mogu se dizajnirati test slučajevi na osnovu znanja o arhitekturi, algoritmu, internim stanjima ili drugim opisima ponašanja programa na visokom nivou. Za testiranje funkcija mogu se koristiti sve tehnike testiranja crne kutije.

Na primer; ako tester tokom testiranja, naiđe na bilo koji nedostatak, on vrši promene u kodu kako bi otklonio nedostatak i zatim ga ponovo testira u realnom vremenu. Kako bi povećao pokrivenost testiranjem, on se koncentriše na sve slojeve složenog softverskog sistema. Omogućava testiranje kako prezentacionog sloja tako i interne strukture koda. Primarno se koristi u integracionom testiranju i testiranju penetracije.

Tehnika testiranja sive kutije je veoma moćna ideja. Koncept je jednostavan; ako neko zna nešto o tome kako radi proizvod unutra, taj ga može testirati bolje, čak i spolja.

Uključuje mogućnost pristupa internim strukturama podataka i algoritama zbog potrebe izrade testnih slučajeva, ali i testiranja kod korisnika ili na black box nivou.

RAZLOZI I KORACI U TESTIRANJU METODOM SIVE KUTIJE

Razlozi na primenu tehnike sive kutije koja se izvodi u osam koraka su mnogobrojni.

Razlozi za testiranje metodom sive kutije su sledeći:

- Pruža kombinovane prednosti testiranja metodom crne kutije i bele kutije.
- Istovremeno uključuje ulazne vrednosti definisane od strane programera i testera što poboljšava opšti kvalitet proizvoda.
- Smanjuje potrošnju vremena dugog procesa funkcionalnog i nefunkcionalnog testiranja.
- Programeru daje dovoljno vremena da otkloni nedostatke proizvoda.
- Uključuje gledišta korisnika, a ne gledište projektanta ili testera.
- Omogućava detaljnije ispitivanje zahteva i određivanje specifikacija prema stanovištu korisnika.

Generički koraci za izvođenje testiranja sive kutije su:

1. Biraju se i identifikuju ulazi na osnovu ulaza iz testiranja bele i crne kutije.
2. Identifikuju se očekivani izlazi za izabrane ulaze.
3. Identifikuju se sve glavne putanje kojima treba proći tokom perioda testiranja.
4. Identifikuju se podfunkcije koje su deo glavnih funkcija za izvođenje testiranja.
5. Identifikuju se ulazi za podfunkcije.
6. Identifikuju se očekivani rezultati podfunkcija.
7. Izvršavaju se test slučajevi za podfunkcije.
8. Proverava se tačnosti rezultata.

Test slučajevi dizajnirani za testiranje sive kutije se mogu odnositi na bezbednost, GUI, operativni sistem i bazu podataka.

TEHNIKE TESTIRANJA METODOM SIVE KUTIJE

Tu spadaju: matrično testiranje, regresijsko testiranje, testiranje ortogonalnih nizova i testiranje na bazi uzoraka

Matrično testiranje (engl. Matrix Testing): metoda kojom se definišu sve korišćene promenljive određenog programa (promenljive su elementi kroz koje vrednosti teku unutar programa) da bi se na osnovu njih uklonile neiskorišćene i neinicijalizovane promenljive.

Regresijsko testiranje (engl. Regression testing): se koristi da bi se verifikovalo da modifikacija bilo kog dela softvera nije prouzrokovala bilo kakve negativne ili neželjene efekte u bilo kom drugom delu softvera. Pre nego što se softver preda na korišćenje, da bi radio na način kako je to predviđeno, vrši se konfirmaciono testiranje (engl. confirmation testing), kada se otklanjaju bilo koji nedostaci u softveru. Međutim, postoji mogućnost da je otklonjeni nedostatak negde drugde u softveru uneo drugačiji nedostatak. Dakle, regresijsko testiranje brine o ovoj vrsti nedostataka testiranjem strategija poput ponovnog testiranja rizičnih slučajeva korišćenja, ponovnog testiranja firewall-a itd.

Testiranje ortogonalnih nizova (engl. Orthogonal Array Testing (OAT)) : svrha ovog testiranja je da maksimalno pokrije kod sa minimalno test slučajeva. Test slučajevi su dizajnirani na način da pokrivaju maksimalan kod kao i GUI funkcije sa što manjim brojem test slučajeva.

Testiranje uzoraka (engl. Pattern Testing): primenljivo na onu vrstu softvera koja je razvijena praćenjem uzorka prethodno razvijenog softvera. U ovoj vrsti softvera postoji mogućnost da

se pojave iste vrste nedostataka. Testiranje uzorka utvrđuje razloge grešaka kako bi se oni mogli ispraviti u sledećem softveru.

Za sprovođenje procesa testiranja metodom sive kutije se obično koriste automatizovani alati za testiranje softvera.

▼ Poglavlje 3

Regresiono testiranje

ŠTA OMOGUĆAVA REGRESIONO TESTIRANJE?

Njime se osigurava da će softverski proizvod dobro funkcionišati sa novom funkcionalnošću, ispravkama grešaka ili bilo kojom promenom postojećih funkcija.

Regresiono testiranje se koristi se za potvrdu autentičnosti da promena koda u softveru ne utiče na postojeću funkcionalnost proizvoda. **Njime se osigurava daće softverski proizvod dobro funkcionišati sa novom funkcionalnošću, ispravkama grešaka ili bilo kojom promenom postojećih funkcija.**

Kod ove vrste softverskog testiranja, ponovnim izvršavanjem test slučajeva se proverava da li prethodna funkcionalnost aplikacije dobro funkcioniše, a nove promene nisu proizvele greške. Regresiono testiranje se može izvršiti kada na novoj verziji kada dođe do značajnih promena u originalnoj funkcionalnosti. Osigurava da kod i dalje radi čak i kada se promene dešavaju. **Regresija znači ponovno testiranje onih delova aplikacije koji su nepromenjeni.**

Regresioni testovi su poznati i kao metod verifikacije. Test slučajevi su često automatizovani jer je test slučajeve potrebno izvršavati mnogo puta, a ručno izvršenje istog test slučaja iznova, može biti dugotrajno i zamorno.

Primer regresionog testiranja: Razmotrite softverski proizvod Y, čija je jedna od funkcija potvrđivanje, prihvatanja i slanja e-maila. Treba izvršiti testiranje kako bi bili sigurni da promena koda ne utiče na funkcionalnost proizvoda.

Regeresiono testiranje ne zavisi od bilo kog programskog jezika kao što su Java, C++, C # itd. Ovaj metod se koristi za testiranje proizvoda na modifikacije ili izvršena ažuriranja. Osigurava da bilo kakva promena proizvoda ne utiče na postojeće module proizvoda. Verifikuje da ispravljene greške i novo dodate funkcije nisu stvorile nikakav problem u prethodnoj radnoj verziji softvera.

KADA I KAKO MOŽEMO IZVRŠITI REGRESIONO TESTIRANJE?

Prilikom regresionog testiranja se mogu koristiti sledeće tehnike ponovnog testiranja: (1) svim test slučajevima (2) izabranim test slučajevima (3) test slučajevima po zadatom prioritetu

Regresiono testiranje možemo izvršiti:

1. Kada se u aplikaciju doda nova funkcionalnost.
2. Kada postoji zahtev za promenom.
3. Kada je fiksirana greška
4. Kada postoji problem sa performansama, npr. učitavanje početne stranice traje 5 sekundi, potrebno je smanjiti vreme učitavanja na 2 sekunde.
5. Kada dođe do promene okruženja, npr. kada ažuriramo bazu podataka sa MySql na Oracle

Regresiono testiranje se može izvršiti korišćenjem sledećih tehnika:

- 1. **Ponovno testiranje svim test slučajevima** (engl. **Re-test All**): U ovom pristupu, treba ponovo izvršiti sve test slučajeve. Na primer, utvrđujemo da ne možemo da izvršimo definisane test slučajeve zbog pojave softverske greške. Greška je prijavljena i možemo očekivati novu verziju softvera u kojoj je greška otklonjena. U ovom slučaju, da bismo potvrdili i bili sigurni da smo otklonili grešku, moraćemo ponovo da izvršimo test. Ovo je poznato kao ponovno testiranje (re-testing). Neki ovo nazivaju testiranjem potvrde (confirmation testing).
Ponovni test je veoma skup, jer zahteva ogromno vreme i resurse.
- 2. **Ponovno testiranje izabranim test slučajevima** (engl. **Regression test Selection**): U ovoj tehnici će se izvršiti izabrani test slučajevi, a ne svi test slučajevi. Odabrani test slučajevi su podeljeni u dve grupe:
 - Test slučajevi za višekratnu upotrebu: mogu se koristiti u sledećem regresionom ciklusu.
 - Zastareli test slučajevi: ne mogu se koristiti u sledećem regresionom ciklusu.
- 3. **Ponovno testiranje test slučajevima pozadatom prioritetu** (engl. **Prioritization of test cases**): U zavisnosti od poslovnog uticaja, slučajevima testiranja se zadaju prioriteti čime se može smanjiti skup regresionih testova.

ALATI ZA REGRESIONO TESTIRANJE

Ručno izvršavanje test slučajeva povećava vreme izvršavanja testa kao i troškove pa je automatizacija regresionih test slučajeva pametan izbor

Ako se softver podvrgne čestim promenama, troškovi regresionog testiranja mogu eskalirati. U takvim slučajevima ručno izvršavanje test slučajeva povećava vreme izvršavanja testa kao i troškove. Automatizacija regresionih test slučajeva je pametan izbor u takvim slučajevima. Obim automatizacije zavisi od broja test slučajeva koji se mogu ponovo upotrebiti za uzastopne regresione cikluse.

Slede najvažniji alati koji se koriste za funkcionalno i regresiono testiranje u softverskom inženjerstvu:

1. **Selenium**: To je open source alat koji se može koristiti za regresiono testiranje veb aplikacija koje se baziraju na korišćenju browser-a

2. Quick Test Professional (QTP): je automatizovani softver koji služi za automatizaciju funkcionalnih i regresionih test slučajeva a baziran je na korišćenju jezika VBScript
3. Rational Functional Tester (RFT): IBM-ov alat koji se bazira na Java programskom jeziku i prvenstveno se koristi za automatizaciju regresionih test slučajeva

REGRESIONO TESTIRANJE I UPRAVLJANJE KONFIGURACIJOM

Upravljanje konfiguracijom tokom regresionog testiranja postaje imperativ naročito tamo gde se kod kontinuirano menja.

Upravljanje konfiguracijom tokom regresionog testiranja postaje imperativ naročito u agilnim okruženjima gde se kod kontinuirano menja. Da biste osigurali efikasne regresione testove, treba se pridržavati sledećeg:

- *Na kod koji se podvrgava regresionom testiranju treba primeniti alat za upravljanje konfiguracijom*
- *Tokom faze regresionog testa, ne smeju se dozvoliti promene u kodu niti u bazi podataka koja se koristi za testiranje.*

Primer: kod regresionog testiranja imamo različite verzije sistema koje imaju različite scenarije testiranja.

Verzija 1:

- Klijent iskazuje poslovne potrebe.
- Razvojni tim počinje da razvija funkcije sistema.
- Tim za testiranje započinje pisanje test slučajeva; na primer, pišu 900 test slučajeva
- Projektni tim počinje sa primenom test slučajeva.
- Nakon što se proizvod završi, kupac vrši testiranje prihvatljivosti.

Verzija 2:

- Sada kupac traži dodavanje 3-4 dodatne (nove) funkcije i specificira zahteve za njih.
- Razvojni tim počinje da razvija nove funkcije.
- Tim za testiranje započinje pisanje test slučajeva za nove funkcije i napiše npr. oko 150 novih test slučajeva. Stoga je ukupan broj napisanih test slučajeva 1050 za obe verzije.
- Sada tim za testiranje započinje testiranje novih funkcija korišćenjem 150 novih test slučajeva.
- Kada završi, započinju testiranje starih funkcija uz pomoć 900 test slučajeva kako bi proverili da li je dodavanje novih funkcija uticalo na stare funkcije ili ne (postupak poznat kao regresiono testiranje).
- Jednom kada su sve karakteristike (nove i stare) testirane, proizvod se predaje kupcu, a zatim će kupac vršiti testiranje prihvatljivosti.

Verzija 3:

- Nakon druge verzije, kupac želi da ukloni jednu od funkcija

- Brišu se svi test slučajevi koji pripadaju obrisanoj funkciji (npr. oko 120 test slučajeva).
- Zatim se testiraju druge funkcije da bi se proverilo da su one ostale neoštećene nakon uklanjanja test slučajeva za obrisanu funkciju (postupak poznat kao regresiono testiranje).

RAZLIKA IZMEĐU PONOVNOG TESTIRANJA I REGRESIONOG TESTIRANJA:

Ponovnim testiranjem se proverava da li otkrivena greška otklonjena a regresionim testiranjem se utvrđuje da izmenjeni kod ne utiče na druge delove softvera koji nisu menjani

Ponovno testiranje (engl. Re-Testing) znači ponovno testiranje funkcionalnosti nakon otkrivanja greške kako bi se osiguralo dobijanje fiksnog koda (kada kod kojeg ne postoje greške).

Regresiono testiranje znači testiranje softverske aplikacije kada se ona podvrgne promeni koda kako bi se osiguralo da novi kod ne utiče na druge delove softvera.

Razlike su sledeće:

- Regresiono testiranje se vrši za test slučajeva koji su prethodno prošli, dok se ponovno testiranje vrši samo za neuspešne test slučajeve.
- Regresiono testiranje proverava neočekivane neželjene efekte, dok se ponovnim testiranjem proverava da li je prvobitna greška ispravljena. Regresiono testiranje ne uključuje verifikaciju grešaka, dok ponovno testiranje uključuje verifikaciju grešaka.
- Regresiono testiranje je poznato kao generičko testiranje, dok je ponovno testiranje planirano.
- Regresiono testiranje je moguće uz upotrebu automatizacije, dok ponovno testiranje nije moguće.

▼ Poglavlje 4

Nefunkcionalno testiranje

TESTIRANJE PERFORMANSI

Testiranje performansi se vrši u odnosu na ciljeve koje je postavio kupac, onako kako su navedeni u nefunkcionalnim zahtevima

Pošto smo utvrdili da sistem izvršava funkcije navedene u zahtevima, prelazimo na način na koji se te funkcije izvršavaju. U nefunkcionalno testiranje spada:

1. Testiranje performansi (engl. **Performance Testing**)
2. Testiranje kompatibilnosti (engl. **Compatibility testing**)
3. Testiranje upotrebljivosti (engl. **Usability Testing**)

Performansa sistema se meri u odnosu na ciljeve koje je postavio kupac, onako kako su navedeni u nefunkcionalnim zahtevima. Brzina odgovora na komande korisnika, preciznost rezultata i dostupnost podataka porede se sa performansom koju je propisao kupac. **Testiranje performanse** projektuje i izvršava tim za testiranje, a rezultati se predaju kupcu. Pošto testiranje performanse obično uključuje i hardver i softver, u tim se mogu uključiti i hardverski inženjeri.

Mogu se obavljati različite vrste testova performansi.

VRSTE TESTOVA PERMANSE

Testiranje performanse se zasniva na zahtevima, pa se vrste testova određuju prema vrstama navedenih nefunkcionalnih zahteva.

Testiranje performanse se zasniva na zahtevima, pa se vrste testova određuju prema vrstama navedenih nefunkcionalnih zahteva. Pod testiranjem performansi se podrazumeva:

- 1. **Testiranje opterećenja** ocenjuju sistem kada se on optereti do svojih granica u kratkom vremenskom periodu. Ako u zahtevima stoji da sistem treba da opsluži najviše određen broj uređaja ili korisnika, test opterećenja ocenjuje performanse sistema kada su istovremeno aktivni svi ti uređaji ili korisnici. Ovaj test je posebno značajan za sisteme koji obično funkcionišu ispod maksimalnog kapaciteta, ali su žestoko opterećeni u određenim trenucima vršne potražnje.
- 2. **Testiranje kapaciteta** posmatraju obradu velike količine podataka u sistemu. Na primer, gledamo da li su strukture podataka (recimo, redovi ili stekovi) definisani dovoljno široko za sve moguće situacije. Osim toga, proveravamo polja, zapise i datoteke, i proveravamo

da li će moći da prime sve očekivane podatke. Takođe proveravamo da li sistem reaguje pravilno kada skupovi podataka dosegnu svoj maksimum.

- 3. **Testiranje konfiguracije** analiziraju različite softverske i hardverske konfiguracije navedene u zahtevima. Ponekad se gradi sistem koji će koristiti različiti auditorijumi, a sistem u stvari ima ceo spektar konfiguracija. Na primer, možemo da definišemo minimalnu konfiguraciju sistema koja će opslužiti samostalnog korisnika, a druge konfiguracije se nadovezuju na tu konfiguraciju kako bi se opslužili dodatni korisnici. Test konfiguracije ocenjuje sve moguće konfiguracije i proverava da li svaka od njih zadovoljava zahteve.
- 4. **Testiranje vremenskog odziva** proveravaju zahteve koji se odnose na vreme odgovora korisniku i vreme izvršenja neke funkcije. Ako neka transakcija mora da se izvrši u zadatom vremenskom intervalu, test izvršava tu transakciju i proverava da li su zahtevi zadovoljeni. Vremenski testovi se obično rade u kombinaciji sa testovima opterećenja da bi se videlo da li su vremenski zahtevi zadovoljeni čak i u situaciji kada je sistem maksimalno opterećen.
- 5. **Testiranje okruženja** posmatraju sposobnost sistema da funkcioniše na lokaciji instalacije. Ako su u zahtevima postavljene granice tolerancije na temperaturu, vlažnost, kretanje, prisustvo hemikalija, vlagu, prenosivost, električna ili magnetna polja, prekide napajanja ili neke druge karakteristike okruženja vezanog za lokaciju, tada naši testovi treba da garantuju pravilnu performansu našeg sistema i u tim uslovima.

TESTOVI KOMPATIBILNOSTI

Vrše se u odnosu na softver, hardever, mobilnu platformu i odnosu na mreže

Vrše se sledeće vrste testiranja kompatibilnosti:

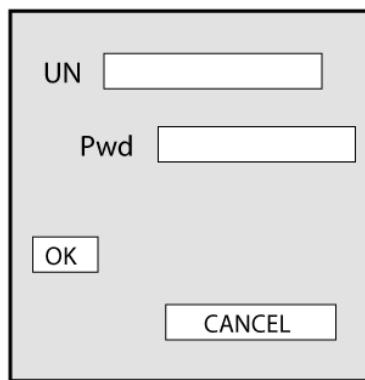
1. **Testiranje kompatibilnosti u odnosu na softver:** podrazumeva proveru na različite operativne sisteme (Linux, Windows i Mac), a takođe i proveravu kompatibilnost softvera na različitim verzijama operativnih sistema poput Win98, Window 7, Window 10, Vista, Window XP, Window 8, UNIX, Ubuntu i Mac, i različite čitače (engl. browser) kao što su Google Chrome, Firefox, and Internet Explorer itd.
2. **Testiranje kompatibilnosti u odnosu na hardver:** podrazumeva proveru da li je aplikacija kompatibilna sa različitim veličinama poput RAM-a, hard diska, procesora i grafičke kartice itd.
3. **Provera kompatibilnosti u odnosu na mobilnu platformu** kao što su iOS, Android itd.
4. **Provera kompatibilnosti u odnosu na mrežu** podrazumeva proveru kompatibilnosti softvera sa različitim mrežnim parametrima, kao što su brzina rada, propusni opseg i kapacitet.

GREŠKE / PROBLEMI KOD TESTIRANJA KOMPATIBILNOSTI

Greške usled kompatibilnosti se generalno odnose na korisnički interfejs, a neki od problema koji se mogu javiti se odnose na: poravnanje, preklapanje i rasutost elemenata na GUI-u

Greške / problemi kod testiranja kompatibilnosti su oni koji se dešavaju na jednoj platformi, ali se ne javljaju na drugoj platformi. Generalno, greške zbog kompatibilnosti se uglavnom odnose na korisnički interfejs, a neki od UI problema su sledeći:

Problemi koji se odnose na poravnanje: javljaju se kada određeni elementi na stranici nisu poravnati u odgovarajućem formatu kao što možemo videti na slici 1:



Slika 4.1 Primer elemenata koji nisu dobro poravnati na stranici [Izvor: NM SE321-2020/2021.]

Problem preklapanja: Kada se jedan atribut preklapa sa drugim atributom. To se može dogoditi kada pokušamo da otvorimo aplikaciju na različitim platformama, čitačima, kao što vidimo na slici 2:

Slika 4.2 Primer elemenata koji se preklapaju [Izvor: NM SE321-2020/2021.]

Problem rasutosti: se javlja kada test inženjeri vrše testiranje kompatibilnosti aplikacije, a aplikacija nije kompatibilna sa svim čitačima i platformom (primer na slici 3.):

Slika 4.3 Primer elemenata koji su rasuti [Izvor: NM SE321-2020/2021.]

ALATI ZA TESTIRANJE KOMPATIBILNOSTI

Neki od najčešće korišćenih alata za testiranje kompatibilnosti su: LambdaTest, BrowserStack i BrowseEMail

Neki od najčešće korišćenih alata za testiranje kompatibilnosti su:

1. **LambdaTest**: To je alat za testiranje kompatibilnosti otvorenog koda u klaudu. Pomoću ovog alata možemo da testiramo našu veb aplikaciju na skoro svim mobilnim i desktop čitačima. LambdaTest omogućava da napravimo screenshots naših veb stranica i aplikaciju testiramo na velikom broju realnih čitača za mobilne uređaje i desktop računare radi provere kompatibilnosti aplikacije.
2. **BrowserStack**: Ovaj alat nam pomaže da testiramo kompatibilnost veb site-ova i mobilnih aplikacija na više čitača i platformi. Na ovaj način možemo da testiramo veb aplikacije korišćenjem različitih čitača i android i iOS mobilne aplikacije na svim mobilnim uređajima. Glavni proizvod BrowserStack alata su Live, Automate, App Live, and App Automate koji nam pomažu da smanjimo vreme, cenu i opšte troškove održavanja povezane sa testiranjem.
3. **BrowseEMAIL**: Ovaj alat može da izvrši aplikaciju na različitim operativnim sistemima kao što su Linux, Windows i macOS, a može se koristiti i za testiranje na različitim čitačima. Koristi se za testiranje aplikacije na svim mobilnim i desktop čitačima, a možemo ga direktno koristiti na našoj lokalnoj mašini i u našoj lokalnoj mreži.

TESTIRANJE UPOTREBLJIVOSTI

Vrsta testiranja, kojim se proveravaju greške u interakciji softvera ili proizvoda sa krajnjim korisnikom. Poznato i kao testiranje korisničkog iskustva (user experience - UX).

Ovo je takođe važan deo nefunkcionalnog testiranja koje zahteva poznavanje aplikacije.

Testiranje upotrebljivosti proverava da li je razvijeni softver lak za korišćenje i da li se može koristiti bez ikakvih problema. To je vrsta testiranja, kojim se proveravaju greške u interakciji softvera ili proizvoda sa krajnjim korisnikom.

Testiranje upotrebljivosti je takođe poznato i kao testiranje korisničkog iskustva (**user experience - UX**). Može se obaviti u fazi projektovanja životnog ciklusa razvoja softvera (SDLC) i pomaže da se dobije jasnija slika o potrebama korisnika.

Lako korišćenje (engl. **user-friendliness**) se može opisati sa nekoliko aspekata, kao što su:

- 1. Lako razumevanje
 - Krajnjim korisnicima moraju biti vidljive sve funkcije softvera ili aplikacije.
- 2. Dobra obrada grešaka
 - Prikazivanjem ispravnih poruka o greškama pomoći će poboljšanju korisničkog iskustva i upotrebljivosti aplikacije.
- 3. Jednostavan pristup
 - Aplikacija koja je laka za razumevanje treba da bude dostupna svima.
 - Da bi se korisnik zainteresovao, izgled aplikacije treba da bude dobar i atraktivan.
 - GUI aplikacije treba da bude dobar, jer ako to nije slučaj, korisnik može izgubiti interesovanje tokom korišćenja aplikacije ili softvera.
 - Kvalitet proizvoda ocenjuje klijent.
- 4. Brz pristup

- Pristup aplikaciji treba da bude brz, što znači da vreme odgovora iz aplikacije treba da bude kratko
- Ako odgovor iz sistema dugo traje, može se desiti da to iritira korisnika. Moramo biti sigurni da je vreme odgovora iz aplikacija 3 do 6 sekundi.
- 5. Efikasna navigacija je najvažniji aspekt softvera. Neki od aspekata efikasne navigacije su:
 - Dobro interno povezivanje
 - Informativna zaglavља
 - Dobre funkcije pretraživanja

KOMPONENTE TESTOVA UPOTREBLJIVOSTI

Kada testiramo aplikaciju ili softver pomoću testa upotrebljivosti, najčešće nailazimo na greške koje predstavljaju rupe u putanjama.

Različite komponente testova prihvatljivosti su:

1. **Efikasnost:** Krajnjem korisniku je potrebno minimum vremena da izvrši svoj osnovni zadatak.
2. **Memoribilnost:** Memoribilnost aplikacije je dobra ukoliko smo u mogućnosti da izvršimo osnovni zadatak bez ikakve pomoći nakon određenog vremenskog perioda nekorишћenja. Ako nismo u mogućnosti da to uradimo, onda možemo reći da memoribilnost aplikacije nije dobra.
3. **Savladivost:** Krajnjem korisniku treba najmanje vremena da nauči na koji način da izvrši osnovni zadatak.
4. **Zadovoljstvo:** Kupac mora biti zadovoljan aplikacijom i slobodno je koristiti.
5. **Greške:** Krajnjim korisnicima treba pomoći da reše greške koje su napravili i ponovo izvrše svoje zadatke.

Kada testiramo aplikaciju ili softver pomoću testa upotrebljivosti, najčešće nailazimo na greške koje predstavljaju rupe u putanjama.

Prednosti testiranja upotrebljivosti su:

1. **Bolji kvalitet proizvoda**
2. **Veće zadovoljstvo krajnjeg korisnika**
3. **Veća efikasnost sistema**

▼ Poglavlje 5

Upravljanje rizikom

ŠTA ZNAČI UPRAVLJANJE RUZIKOM?

Ne samo praćenja i raspoređivanja projekta već i otkrivanje nekih nepravilnosti koje se mogu pojaviti u toku razvoja i održavanja projekta i minimiziranje njihovih negativnih posledica

Mnogi menadžeri softverskih projekata preuzimaju korake kako bi za svoj projekat osigurali da se izvrši u planiranim vremenskim okvirima, sa predviđenim (**effort**) i u okviru predviđenih troškova. Međutim upravljanje projektom zahteva mnogo više napora od praćenja i raspoređivanja projekta. Menadžer treba da utvrdi da li će se neke nepravilnosti pojaviti u toku razvoja i održavanja projekta i da planira njihovo izbegavanje, ili ako su one neizbežne, da minimizira njihove negativne posledice.

Menadžeri projekta često koriste risk management u cilju razumevanja i kontrolisanja rizika u svojim projektima.

Risk management uključuje nekoliko važnih koraka:

1. **Procena rizika projekta**- razumevanje šta se može desiti u toku razvoja ili održavanja. Procena se sastoji od tri aktivnosti: identifikacija rizika, analiziranje i određivanje prioriteta za svakog od njih.
2. **Ako se izrađuje sličan sistem**- može se iskoristiti spisak problema koji se mogu pojaviti, korišćenjem dokumentacije izrade prethodnog sistema
3. **Napraviti proveru**- da li će projekat biti izložen navedenim rizicima
4. **Proširenje spiska** - analizom svake aktivnosti u razvoju projekta dekomponovanjem procesa na manje delove, kada su u pitanju noviji sistemi
5. **Analiziranje pretpostavki**- donošenje odluke o tome kako će projekat biti urađen, ko će voditi projekat i koje resurse će koristiti i nazad, procenjivanje pretpostavki radi utvrđivanja rizika.
6. **Analiza rizika** - identifikovanje rizika i razumevanje kada, zašto i gde se mogu pojaviti
7. **Dodeljivanje prioriteta rizicima** - nacrt prioritetnih rizika omogućuje odvajanje određenih resursa kojima preti neki rizik

RISK BOX - TEHNIKE

Upravljanje rizicima u ovom modulu definiše šest koraka tokom kojih tim upravlja tekućim rizicima, planira i izvršava strategije upravljanja rizicima.

Naš risk box ima sledeće osobine:

- *podržava proaktivno upravljanje rizicima, kontinualnu ocenu rizika i odlučivanje tokom životnog ciklusa projekta.*
- *kontinualno procenjuje, nadgleda i aktivno upravlja rizicima.*

Upravljanje rizicima u ovom modulu definiše **šest koraka** tokom kojih tim upravlja tekućim rizicima, planira i izvršava strategije upravljanja rizicima:

- **Identifikovanje rizika** - primenom brainstorming-a mogu se identifikovati svi potencijalni rizici.
- **Analiza rizika** - prema proceni verovatnoće događaja rizika i njegovog uticaja na sistem, rizici se sortiraju prema prioritetu.
- **Planiranje rizika** - koriste se informacije dobijene analizom rizika kako bi se formulisali planovi, strategije i akcije. Za svaki rizik se procenjuje njegov uticaj na ishod projekta, navode se načini njegovom umanjenja i koraci koje treba sprovoditi ukoliko do rizika dođe.
- **Praćenje rizika** -nadgleda se status određenih rizika.
- **Kontrolisanje rizika** - proces izvršavanja planova akcija i njihovog izveštavanja.
- **Učenje iz rizika** -formulišu se naučene lekcije kako bi se to znanje ponovo upotrebilo u sličnim slučajevima kod budućih projekata.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 6

Grupna vežba: Testiranje toka podataka

TESTIRANJE TOKA PODATAKA - DATA FLOW TESTING

Testiranje toka podataka je efikasna metodologija za otkrivanje defekata upravo ovog tipa, tj. nekorektnog korišćenja podataka usled grešaka u kodiranju.

Retko koji programer nije napravio sledeću grešku:

```
main() {int x;if (x==42){ ...}}
```

Greška je u tome što je referisana promenljiva, a da joj nije prethodno dodeljena nikakva vrednost.

Naivniji programeri misle ili očekuju da će kompjajler ili sitem za izvršavanje programa setovati sve varijable na nulu, blankove, TRUE, 42 ili nešto drugo što će im baš trebati u programu.

Jednostavni program u C-u ilustruje ovaj defekt:

```
#include <stdio.h>main() {int x;printf ("%d",x);}
```

Vrednost koja će biti odštampana može biti bilo šta što se zateklo u memoriji na lokaciji koja je dodeljena promenljivoj x, a to najčešće nije ono što programer očekuje.

Kao što je rečeno, testiranje toka podataka je efikasna metodologija za otkrivanje defekata upravo ovog tipa, tj. nekorektnog korišćenja podataka usled grešaka u kodiranju.

Promenljive koje sadrže podatke poseduju svoj životni ciklus:

- Varijable se kreiraju
- Varijable se koriste
- Varijable se brišu

U nekim programskim jezicima (kao što su na pr. FORTRAN i BASIC), kreiranje i brisanje su automatski. Varijabla se kreira kad joj se prvi put dodeli vrednost, a briše se završetkom programa (program exit).

U drugim jezicima (kao što su C, C++, java), kreiranje je formalno.

KAKO TREBA KORISTITI PROMENLJIVE?

Promenljive se mogu koristiti pri računanju: $a = b + 1$ ili kao uslovi: if ($a > 42$) no u oba slučaja je podjednako važno da im se dodeli vrednost pre poziva

Varijable su deklarisane instrukcijama tipa:

`int x; // x is created as an integer` `string y; // y is created as a string`

Deklaracije se najčešće pojavljuju u blokovima programskog koda koji počinju otvorenom zagradom "{" i završavaju se zatvorenom zagradom"}".

U takvim blokovima se varijable kreiraju onda kad se njihove definicije (deklaracije) izvrše automatski se brišu na kraju bloka.

Ovo je poznato kao scope varijabli (okvir ili opseg) varijabli.

Primer:

```
{ // begin outer block
int x; // x is defined as an integer within this outer block
...; // x can be accessed here
{ // begin inner block
int y; // y is defined within this inner block
...; // both x and y can be accessed here
} // y is automatically destroyed at the end of
// this block
...; // x can still be accessed, but y is gone
} // x is automatically destroyed
```

Promenljive se mogu koristiti

1. pri računanju: $a = b + 1$
2. kao uslovi: if ($a > 42$)

no u oba slučaja je podjednako važno da im se dodeli vrednost pre poziva na njih, tj. pre njihove upotrebe.

Pri prvoj pojavi varijable u kodu moguća su tri slučaja:

1. ~d variabla ne postoji (označe no sa ~), a zatim je definisana (d) -(defined)
2. ~u variabla ne postoji, a zatim je korišćena (u) -(used)
3. ~k variabla ne postoji, a zatim je obrisana (k) -(killed)

- Prva mogućnost je korektna - Varijabla ne postoji, zatim je definisana.
- Druga mogućnost nije korektna - Varijabla ne postoji, i ne sme se koristiti
- Treća mogućnost nije korektna - Varijabla ne postoji, zatim se briše - brisanje varijable pre njene kreacije je programska greška

ŠTA JE IDEJA TEHNIKE TESTIRANJA TOKA PODATAKA?

Analiza parova i redosled dogadjaja definisan (d), korišćen (u) i obrisan (k) (define-use-kill) je osnovna ideja date tehnike.

Sada posmatrajmo sledeće parove sačinjene od defined (d), used (u), i killed (k), koji su uređeni tako da se drugi činilac događa nakon prvog u programu:

- dd – definisana i opet definisana – nije nekorektno ili je sumnjivo. Verovatno programska greška.
- du – definisana i korišćena – potpuno korektno (engl. **perfectly correct**). Normalni slučaj.
- dk – definisana zatim obrisana – nije nekorektno ili je sumnjivo. Verovatno je programska greška.
- ud – korišćena i definisana – prihvatljivo (engl. **acceptable**).
- uu – korišćena i opet korišćena – prihvatljivo (engl. **acceptable**).
- uk – korišćena i obrisana – prihvatljivo (engl. **acceptable**).
- kd – obrisana i definisana – prihvatljivo (engl. **acceptable**). Varijabla je obrisana a zatim je redefinisana
- ku – obrisana i korišćena – ozbiljan defekt. Korišćenje varijable koja ne postoji ili koja nije definisana je uvek greška i
- kk – obrisana i opet obrisana – Verovatno je programska greška

Važno: **Analiza parova i redosled dogadjaja definisan (d), korišćen (u) i obrisan (k) (define-use-kill) je osnovna ideja date tehnike.**

GRAFOVI TOKA PODATAKA

Grafovi toka podataka su slični grafovima toka kontrole izvršavanja programa, utoliko što takođe pokazuju tok izvršavanja kroz modul.

Grafovi toka podataka su slični grafovima toka kontrole izvršavanja programa, utoliko što takođe pokazuju tok izvršavanja kroz modul. Oni prate definiciju, korišćenje i destrukciju (brisanje) promenljivih.

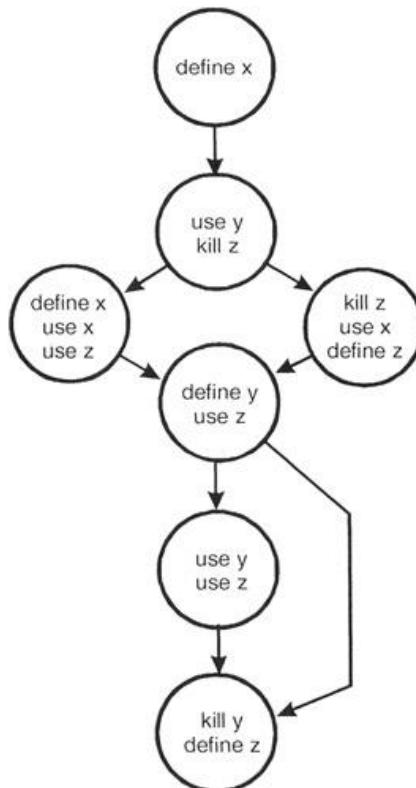
Etape primene tehnike su sledeće:

1. Konstrukcija grafova ili dijagrama toka podataka
2. Analiza redosleda definicija, korišćenje i destrukcija za svaku promenljivu

Najpre se izvršava staticka provera grafa toka. Pod "statickim" se podrazumeva provera korektnosti grafa (bilo kroz formalne postupke kao što su inspekcije i pregledi, bilo kroz neformalne provere).

Nakon toga se izvršava dinamičko testiranje modula. Pod "dinamičkim" se podrazumeva konstrukcija i izvršavanje test slučajeva.

Posmatrajmo sledeći graf toka podataka sa anotacijom define-use-kill za svaku od promenljivih u modulu (slika 1.).

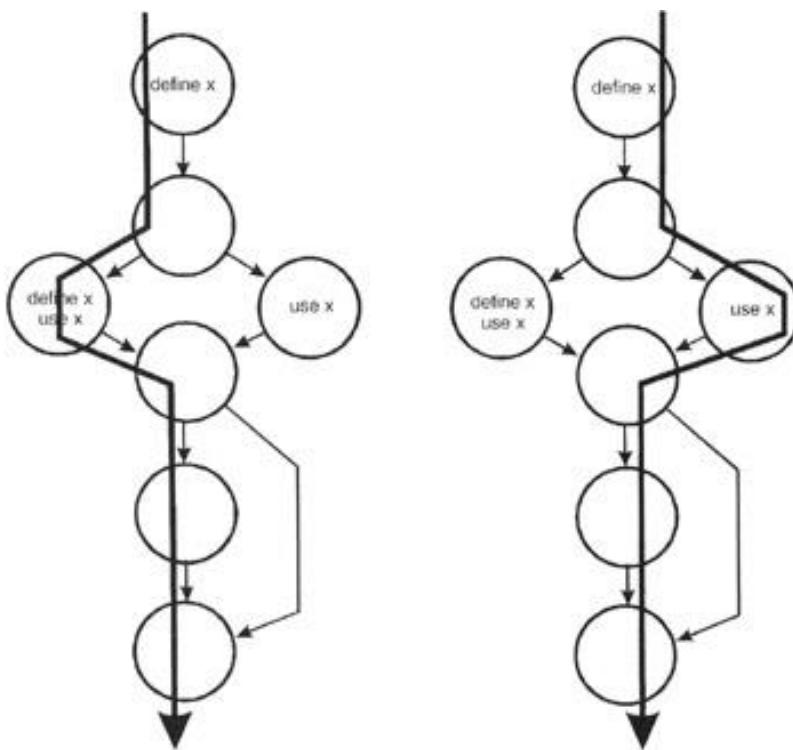


Slika 6.1.1 Graf toka podataka sa anotacijom define-use-kill [Izvor: NM SE321-2020/2021.]

GRAF TOKA PODATAKA SA ANOTACIJOM DEFINE-USE-KILL ZA PROMENLJIVU X

Posmatrajmo redosled u nizu definicija, korišćenje i destrukcija (define-use-kill) kroz graf za promenljivu x.

Sada posmatrajmo redosled u nizu definicija, korišćenje i destrukcija (define-use-kill) kroz graf. Posmatrajmo najpre promenljivu x. Može se nacrtati najpre leva a zatim desna putanja (Sl. 2)



Slika 6.1.2 Graf toka podataka sa anotacijom define-use-kill za promenljivu x [Izvor: NM SE321-2020/2021.]

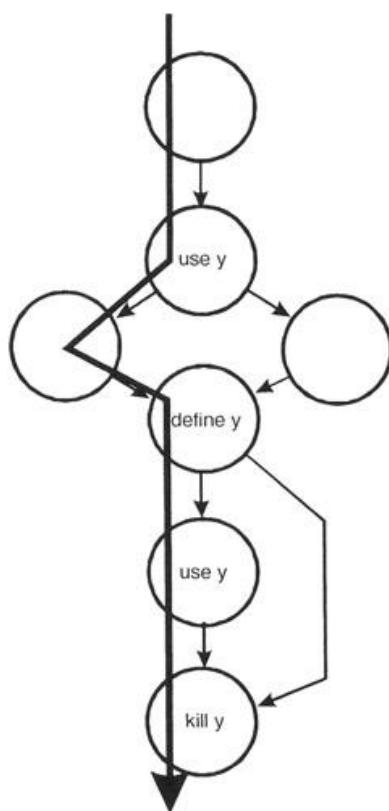
Prođimo putanje i posmatrajmo redosled define-use-kill (definicija, korišćenje i destrukcija). Imamo (Slika 2):

- \sim define, korektno, normalni slučaj
- define-define, sumnjivo, moguća programska greška
- define-use , korektno (engl. correct), normalni slučaj

GRAF TOKA PODATAKA SA ANOTACIJOM DEFINE-USE-KILL ZA PROMENLJIVU Y

Posmatrajmo redosled u nizu definicija, korišćenje i destrukcija (define-use-kill) kroz graf za promenljivu y.

Razmotrimo sada promenljivu y. Graf toka podataka za promenljivu y je dat na Slici 3 .



Slika 6.1.3 Graf toka podataka sa anotacijom define-use-kill za promenljivu y [Izvor: NM SE321-2020/2021.]

Na osnovu grafa toka podataka za promenljivu y koji je dat na slici 3 imamo:

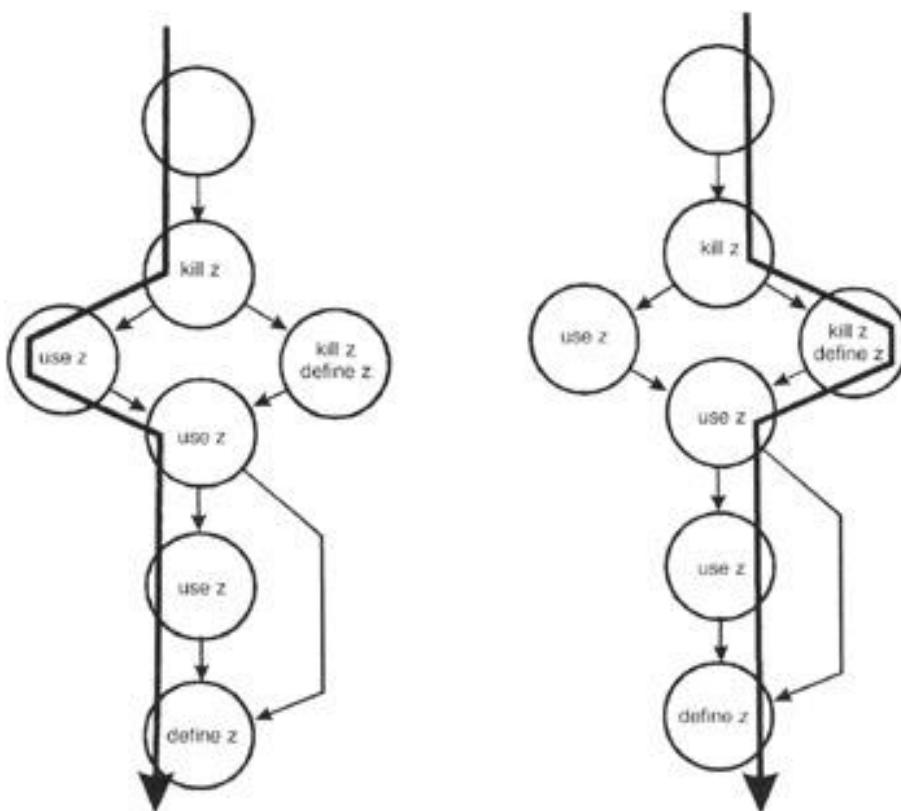
- ~use katastrofalna greška
- use-define prihvatljivo
- define-use korektno, normalan slučaj
- use-kill prihvatljivo
- define-kill verovatna programska greška

GRAF TOKA PODATAKA SA ANOTACIJOM DEFINE-USE-KILL ZA PROMENLJIVU Z

Posmatrajmo redosled u nizu definicija, korišćenje i destrukcija (define-use-kill) kroz graf za promenljivu z.

Na kraju, razmotrimo sada promenljivu z.

Graf toka podataka sa anotacijom define-use-kill za promenljivu z je dat na Slici 4 .



Slika 6.1.4 Graf toka podataka sa anotacijom define-use-kill za promenljivu z [Izvor: NM SE321-2020/2021.]

Imamo:

- \sim kill programska greška
- kill-use katastrofalna greška
- use-use korektno, normalan slučaj
- use-define acceptable
- kill-kill verovatna programska greška
- kill-define acceptable
- define-use korektno, normalan slučaj

Rezimirajmo: Tokom statičke analize posmatranog dijagrama toka podataka, uočeni su sledeći nedostaci:

- x: define-define
- y: \sim use y: define-kill
- z: \sim kill z: kill-use z: kill-kill

NEDOSTACI STATIČKE ANALIZE TOKA PODATAKA

Iako statička analiza može otkriti većinu grešaka u toku podataka, nažalost ne može ih otkriti sve koje se mogu javiti.

Iako statička analiza može otkriti većinu grešaka u toku podataka, nažalost ne može ih otkriti sve koje se mogu javiti. Razmotrimo sledeće situacije:

- 1. Array (niz podataka u programu) je kolekcija podataka koji imaju isto ime i tip. Na primer: Int stuff[100]; definiše array koji se zove stuff i koji se sastoji od 100 integer (celobrojnih) elemenata. U C, C++ i Javi se pojedini elementi adresiraju sa stuff[0], stuff[1], stuff[2], itd. Array se definišu i brišu kao celina, no elementi se koriste pojedinačno. Programeri često referišu stuff[j], pri če mu se j dinamički menja tokom izvršenja programa. U opštem slučaju, statička analiza ne može otkriti da li se procedura define-use-kill poštuje za promenljivu tipa array, sem ako se svaki od elemenata ne posmatra posebno (a što može biti neizvodljivo).
- 2. U slučaju kompleksnih grafova toka kontrole programa moguće je da postoje putanje koje se nikad ne izvršavaju. Neadekvatna shema define-use-kill može postojati, no kako se ne izvršava ne može predstavljati defekt, pa otuda i nije neadekvatna.
- 3. Kod sistema sa obradom interapta, može doći do aktiviranja sheme define-use-kill na nivou interapta, pri čemu se može odvijati druga define-use-kill shema na nivou glavnog programa. Sem toga, kod višenivojskog interapta, statička analiza svih mogućih interakcija može biti prosti neizvodljiva.

Iz navedenih razloga se uvodi dinamicka analiza toka podataka.

DINAMIČKA ANALIZA TOKA PODATAKA

Testiranje na bazi analize toka podataka je zasnovano na testiranju na bazi analize toka kontrole izvršavanja programa, i predstavlja njegovo proširenje i dopunu.

Kako se testiranje toka podataka zasniva na grafu toka kontrole izvršavanja modula, prepostavlja se da je tok kontrole izvršavanja korektan.

Proces testiranja toka podataka se sastoji i tome da se izabere dovoljno test slučaja tako da budu zadovoljeni uslovi:

- Svaki "define" sledi "used"
- Svaki "used" je posledica odgovarajućeg "define.."

Da bi se ovo obezbedilo treba uraditi sledeće:

- Numerišite putanje u modulu, koristeći istu proceduru kao i kod testiranja toka kontrole izvršavanja.
- Podite od ulaza u modul, izaberite prvu putanju s leva i pratite je do izlaza.
- Vratite se na početak i vratite (promenite) ishod prvog grananja na prethono stanje, a zatim varirajte (promenite) ishod drugog grananja. Pratite tu putanju do izlaza.
- Ponovite dati postupak sa narednim grananjima, sve dok ne iscrpete sve putanje.
- Na kraju, za svaku putanju kreirajte bar jedan test slučaj.

Primena i ograničenja testiranja na bazi analize toka podataka

Testiranje na bazi analize toka podataka je zasnovano na testiranju na bazi analize toka kontrole izvršavanja programa, i predstavlja njegovo proširenje i dopunu.

Otuda, kao i testiranje na bazi analize toka kontrole treba da bude primenjivano kod modula koji se ne mogu dovoljno dobro proveriti kroz inspekcije i pregledе.

Nedostatak je da tester mora posedovati dovoljan nivo poznavanja programiranja da bi mogao da razume kod, logiku programa i varijable.

Testiranje na bazi analize toka podataka može zahtevati puno vremena da bi se proverile sve putanje, sve varijable i svi moduli.

Vreme potrebno za izradu vežbe 45 minuta.

▼ 6.1 Vežba za samostalni rad

ZADATAK ZA SAMOSTALNI RAD

Tekst zadatka za samostalni rad

Zadatak 1:

Uzeti primer programskog koda dva modula određenog sistema. Testirati programski kod korišćenjem open source alata Selenium. Sačuvati dobijene rezultate.

Izmeniti funkcionalnost u jednom modulu i zatim izvršiti regresiono testiranje. Uporediti dobijene rezultate nakon prvog i drugog testiranja.

(vreme izrade zadatka 45 minuta)

Zadatak 2:

Odabrati deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka) i za izabrani deo izvršiti testiranje upotrebljivosti prema kriterijumima koje sam zadaje. Pri tom treba da se fokusira se na sledeće aspekte upotrebljivosti:

1. Lako razumevanje
2. Dobra obrada grešaka
3. Jednostavan pristup
4. Brzina pristup
5. Efikasnost navigacije
6. Dokumentovati izvršeno testiranje u tabelarnom obliku.

(vreme izrade zadatka 45 minuta)

✓ Poglavlje 7

Domaći zadatak

SEDMI DOMAĆI ZADATAK

Nakon sedme lekcije potrebno je uraditi sedmi domaći zadatak.

Za odabranu aplikaciju u okviru prethodnih domaćih zadataka primeniti sledeće (izabrati 2 od tri moguća zadatka):

1. Izvršiti testiranje primenom tehnike testiranja toka podataka
2. Izvršiti regresorno testiranje primenom nekog alata
3. Izvršiti testiranje upotrebljivosti

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane diajgrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ07-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ07.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Strukturno testiranje i tehnike sive kutije (hibridno testiranje)

WHITE BOX, GRAY BOX TEHNIKE TESTIRANJA I NEFUNKCIONALNO TETSIKANJE SOFTVERA

Testni podaci primenom strukturnih tehnika testiranja se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji.

Testni podaci primenom strukturnih tehnika testiranja se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji. Tester poznaje internu strukturu i logiku programa, isto kao što mehaničar zna unutrašnji mehanizam automobila. Specifični primeri koji spadaju u ovu kategoriju uključuju testiranje izraza, gransko testiranje, testiranje uslova i sl. Veoma česta programska greška je referisanje na vrednost varijable pre nego što je ikakva vrednost dodeljena datoj varijabli.

Da bi se u kodu otkrile ovakve situacije, koristi se tehnika bele kutije poznata kao testiranje toka podataka koja koristi graf kontrolnog toka za otkrivanje nelogičnih stvari koje mogu prekinuti tok podataka. Graf kontrolnog toka omogućava proveru da li se poštuje pravilo o redosledu akcija definisanje-korišćenje-destrukcija (define-use-kill) promenljivih u modulu.

U predavanju je takođe bilo reči i tehnikama nefunkcionalnog testiranje softvera kojima se testiraju ne funkcionalni zahtevi sistema.

LITERATURA ZA LEKCIJU 07

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. dr.sc. Tihana Galinac Grbac, Testiranje programskog proizvoda. (Link: <http://www.riteh.uniri.hr/>)
4. Dr Kelvin Ross, Practical Guide to Software System Testing, K. J. Ross & Associates Pty. Ltd., 1998.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. British Computer Society, Standard for Software Component Testing, Date: 27 April 2001.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <https://www.softwaretestingclass.com/what-is-cause-and-effect-graph-testing-technique/>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Organizacija procesa testiranja SW

Lekcija 08

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 08

ORGANIZACIJA PROCESA TESTIRANJA SW

- ✓ Organizacija procesa testiranja SW
- ✓ Poglavlje 1: Nivoi i tipovi testiranja softvera
- ✓ Poglavlje 2: Jedinično testiranje
- ✓ Poglavlje 3: Mocking testiranje
- ✓ Poglavlje 4: Testiranje Integracije
- ✓ Poglavlje 5: Sistemsko testiranje
- ✓ Poglavlje 6: Pokazna vežba
- ✓ Poglavlje 7: Vežba za samostalni rad
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

U ovoj lekciji se govori o različitim nivoima testiranja softvera. Prema V modelu, za razliku od razvoja softvera koji se vrši od vrha ka detaljima (top-down), testiranje se vrši od detaljnijeg ka globalnijem nivou (bottom up). Najčešće se sreće podela testiranja softvera na četiri nivoa:

1. Testiranje modula ili jedinica (engl. **Unit Testing**)
2. Testiranje integracije (engl. **Modul Integration Testing**)
3. Testiranje sistema (engl. **System Testing**)
4. Primopredajno testiranje (engl. **Acceptance Testing**)

Testiranje jedinica (modula) može biti vršeno u razvojnom okruženju uz upotrebu pomoćnih modula: drajvera i stabova. Integraciono testiranje se može odvijati prema trima strategijama: Big-bang integraciji, Top-down integraciji i Down-top integraciji. U okviru testiranja sistema, pored softvera testira se i hardver, dokumentacija i sve drugo što se isporučuje korisniku.

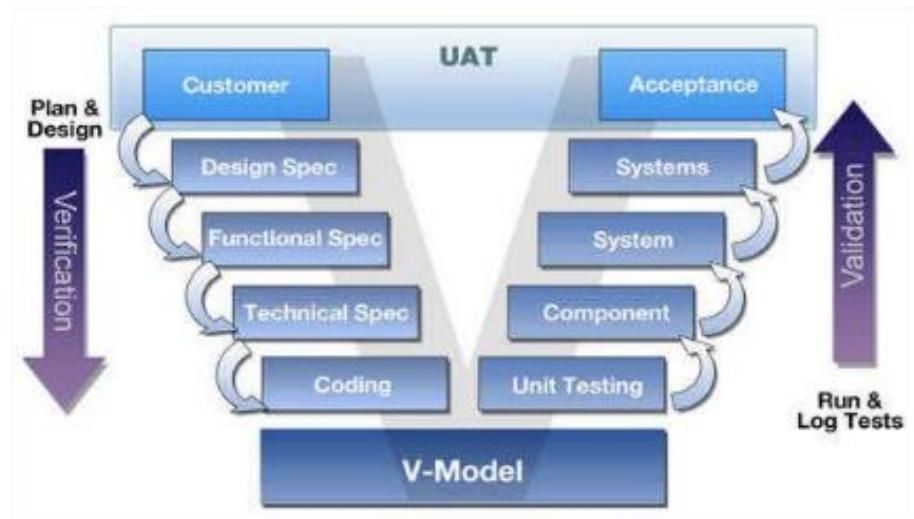
✓ Poglavlje 1

Nivoi i tipovi testiranja softvera

NIVOI TESTIRANJA

Prema V modelu, razvoj softvera se vrši od celine ka detaljima (top-down) a testiranje obrnutim redosledom, od detalja ka celini (bottom up).

Nivoi testiranja su u uskoj vezi sa planiranjem razvoja softvera. Klasično planiranje testiranja je zasnovano na vodopadnom (**waterfall**) modelu (zahtevi, dizajn, implementacija) čijim se proširivanjem na aspekte testiranja sistema dobija poznati **V model** kao na slici 1:



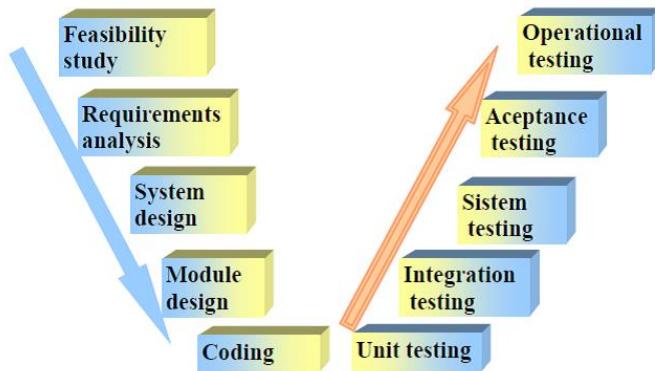
Slika 1.1 V model razvoja i testiranja sistema [Izvor: NM SE321-2020/2021.]

Kao što se vidi, razvoj softvera je u levoj grani odozgo-nadole (top-down) tj. od celine ka detaljima, dok se testiranje, odvija u desnoj grani obrnutim redosledom, odozdo nagore (bottom up) tj. od detalja ka celini.

Primetimo da svaka od faza analize, dizajna i implementacije daje rezultate (**deliverables**) koji odgovaraju svakom od nivoa testiranja.

Specifikacija projektnih zahteva, kao najbitniji opis ponašanja sistema, je polazna tačka za testiranja na nivou sistema, koja čine funkcionalna (po pravilu black box) testiranja. Strelice u levo pokazuju da se testiranjem otkrivaju defekti koji nastaju na datom nivou. Na sličan način se odvija proces specifikacije i testiranja i na drugim nivoima. Napomenimo da je ovo veoma pojednostavljen izlaganje, u cilju veće jasnoće, a da u praksi, (ili u okviru predmeta, zavisno od potreba) može biti mnogo i varijacija i dodatnih detalja. Za razmatranja testiranja

je pogodnija sledeća predstava V modela koja je bliža inženjerskom aspektu razvoja softvera (Slika 2).



Slika 1.2 Razrada V modela razvoja i testiranja sistema [Izvor: NM SE321-2020/2021.]

TESTIRANJE MODULA ILI JEDINICA

Testiranje jedinica (modula) može biti vršeno u razvojnom okruženju uz upotrebu pomoćnih modula: drajvera i stabova

Kao što smo već napomenuli, najčešće se sreće podela testiranja softvera na četiri nivoa:

1. *Testiranje modula ili jedinica (engl. Unit Testing)*
2. *Testiranje integracije (engl. Modul Integration Testing)*
3. *Testiranje sistema (engl. System Testing)*
4. *Primopredajno testiranje (engl. Acceptance Testing)*

Navedimo najpre objašnjenje ovih pojmoveva.

Testiranje modula ili jedinica (engl. **Unit Testing**): Modul, jedinica (**Unit**) je "najmanji" deo softvera koji se kreira pri razvoju, a što je definisano projektnom dokumentacijom. Tipično, to je ono što jedan programer uradi, a zavisno od jezika, i može se reći da je to funkcija u C-u, klasa u C++ ili Javi. Možda je najkorektnije definisati ga kao najmanji deo softvera koji treba testirati.

Testiranje modula može biti vršeno u razvojnom okruženju uz upotrebu pomoćnih modula:

1. Drajvera
2. Stabova

Klasičan primer je C program:

```
/* main program */
void oops(int);
int main(){
    oops(42); /* call the oops function passing an integer */
    return 0;
}
/* function oops (in a separate file) */
```

```
#include <stdio.h>
void oops(double x) /* expects a double, not an int! */
printf ("%f\n",x); /* Will print garbage (0 is most likely)
*/
}
```

Drajver je modul koji zamenjuje deo sistema ka korisniku i obezbeđuje ulaze za testiranje, prikazuje ili registruje izlaze iz modula, i/ili ih upoređuje sa očekivanim vrednostima.

Stab (engl. *stub*) je pomoćni modul koji zamenjuje deo sistema ka periferiji (hardver ili baze podataka).

INTEGRACIONO TESTIRANJE

Integraciono testiranje se odvija prema trima strategijama: Big-bang integracija, Top-down integracija i Down-top integracija

Testiranje integracije modula (engl. *Integration Testing*). Ujedinjavanjem modula (units) u celine se dobijaju podsistemi ili sistem. Iako su svi moduli zadovoljili testiranje svaki za sebe, posle integracije može doći do neadekvatnog ponašanja – defekata. Svaka za sebe od funkcija može biti korektna, defekt se pojavljuje pri integraciji.

Testiranje modula u sadejstvu sa drugim modulima (ostatkom sistema) se odvija prema trima strategijama:

- 1. Big-bang integracija - svi moduli koji sačinjavaju sistem se testiraju ponaosob i integrišu odjednom. Moduli se testiraju u okviru sistema.
- 2. Top - down integracija inkrementalno integriše i testira module počevši od osnovnog (najvišeg) modula, a zatim se dodaju jedan po jedan (ili više) modul i testiraju. Testiranje se vrši na konfiguracijama za testiranje koje uključuju stabove.
- 3. Bottom - up integracija inkrementalno integriše i testira module počevši od najnižeg modula, a zatim se dodaju jedan po jedan (ili više) modul i testiraju. Testiranje se vrši na konfiguracijama za testiranje koje uključuju drajvere radi zamene ostatka sistema. Tehnika baznih putanja koju je McCabe primenio na testiranje modula se uspešno generalizuje na testiranje sistema. Integraciona kompleksnost je broj, koji je mera kompleksnosti sistema kao skupa modula, a koji je analogan ciklomatskoj kompleksnosti za programsku strukturu.

Za testiranje integracije se mogu dati sledeće preporuke.

- **Što je manje integracionih celina, što je manje drajvera i stabova, što je manji broj integracionih testova, to će biti teže otkrivanje i izolacija defekata.**
- **Što je sistem kompleksniji, to je teži za otkrivanje defekata, to treba sporije i pažljivije vršiti testiranje.**
- **Kao i kod testiranja baznih putanja modula, integracioni bazni testovi mogu propustiti važne defekte, jer pokrivaju samo relacije pozivaoc-pozvani.**

INTEGRACIJA MODULA U CELINE

Integracijom modula u celine se kreira celina (engl. builds) - to je skup integrisanih modula koji su međuzavisni i/ili kooperiraju i/ili komuniciraju i koji čine deo sistema

Pod celinom ćemo podrazumevati skup integrisanih modula koji su međuzavisni i/ili kooperiraju i/ili komuniciraju i koji čine deo sistema. Ove celine su obično definisane arhitekturom sistema, no veoma često, analizom rizika koji unose mogući su otkazi koji se javljaju usled interakcije modula. Interakcije se ostvaruju preko podataka, toka kontrole izvršavanja, ili komunikacijom između modula.

Ako su moduli uspešno položili sve testove na nivou modula, to ne predstavlja garanciju da neće biti defekata kada počnu da kooperiraju međusobno.

Tehnike bele kutije mogu biti uspešno primenjene za otkrivanje defekata u interfejsu modula i time poboljšati nivo kvaliteta softvera. Kao i kod testiranja modula, i ovde je od značaja kompromis između kvaliteta testiranja i njegove cene. Nego, pođimo redom. Razmotrimo najpre nivoe testiranja, testiranje modula kao i načine integracije modula.

SISTEMSKO TESTIRANJE I PRIMOPREDAJNO TESTIRANJE

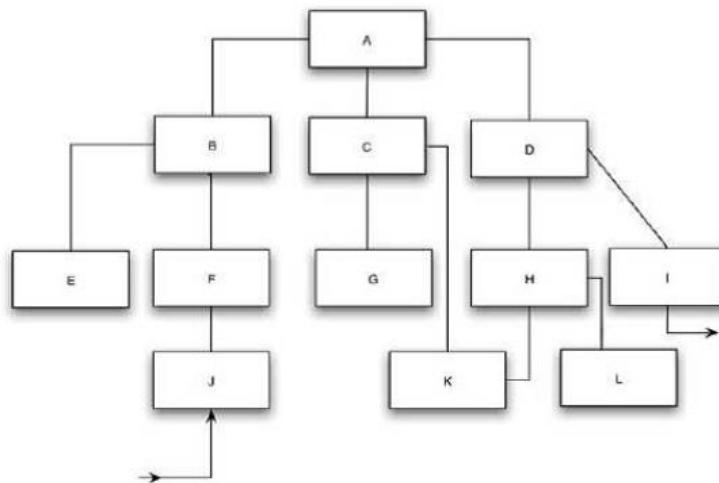
Pre nego što kupcu predamo sistem za koji verujemo da ispravno funkcioniše, neophodno je obaviti više vrsta testiranja.

Testiranje Sistema (System Testing). Sistem pored softvera čini i hardver, dokumentacija i drugo što se isporučuje korisniku.

Sem funkcionalnih testova (da li sistem radi ono što bi trebalo) postoji niz osobina – atributa koje treba proveriti kao što su pouzdanost, prenosivost, upotrebljivost (**usability**) i drugo.

U daljem razmatranju ćemo posmatrati primer sistema čija je arhitektura prikazana na slici 1

Na ovom primeru ćemo ilustrovati pojedine aspekte testiranja modula, kako izolovanih, tako i integrisanih u kooperativne celine.



Slika 1.3 Primer arhitekture sistema [Izvor: NM SE321-2020/2021.]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Primopredajno testiranje (engl. **Acceptance Testing**). Najčešće se definiše kao testiranje koje će, ukoliko je uspešno, rezultovati prihvatanju proizvoda od strane korisnika.

VIDEO

XXXXXX

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

Jedinično testiranje

DA LI OBAVEZNO VRŠITI JEDINIČNO TESTIRANJE?

Preskakanje jediničnog testiranja može dovesti do većih nedostataka tokom testiranja integracije, testiranja sistema i testiranje prihvatljivosti ili čak tokom Beta testiranja.

U hijerarhiji nivoa testiranja, jedinično testiranje je prvi nivo testiranja koje se radi pre integracionog i preostalih nivoa testiranja softvera.

Kao što smo videli, softver uopšteno prolazi kroz četiri nivoa testiranja: testiranje jedinica, testiranje integracije, testiranje sistema i testiranje prihvatljivosti, ali ponekad, zbog uštete vremena, testeri posvećuju minimalno vremena jediničnom testiranju. Međutim, preskakanje jediničnog testiranja može dovesti do većih nedostataka tokom testiranja integracije, testiranja sistema i testiranje prihvatljivosti ili čak tokom Beta testiranja koje se odvija nakon završetka softverske aplikacije.

U nastavku su navedeni neki ključni razlozi:

1. **Jedinično testiranje pomaže testeru i programerima da razumeju osnovu koda što im omogućava da brzo promene prouzrokovane greške**
2. **Testiranje jedinica pomaže u dokumentaciji.**
3. **Testiranje jedinica ispravlja nedostatke vrlo rano u fazi razvoja, zato postoji mogućnost da se pojavi manji broj nedostataka u narednim nivoima testiranja.**
4. **Pomaže kod ponovne upotrebe koda migracijom koda i test slučajevima.**

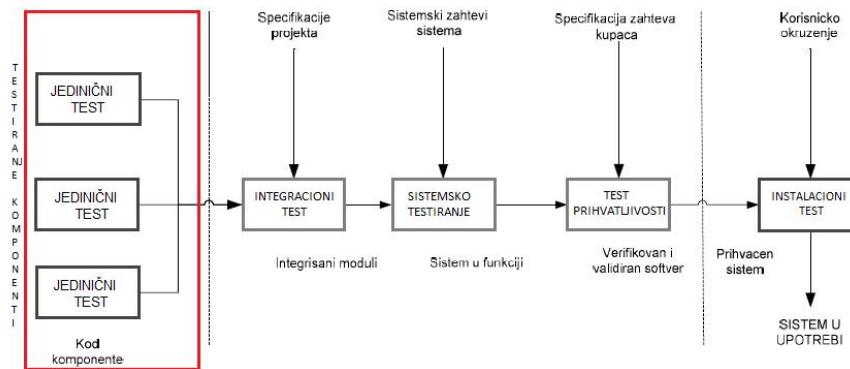
JEDINIČNI TEST

Jedinični test omogućava proveru da li se jedinica ponaša onako kako je namenjeno i da li odgovara projektu specifikacije. Opšta preporuka je da se vrši paralelno sa implementacijom.

Osnovna ideja jediničnog testiranja je testiranje malog dela izvornog koda (jedinica komponenta i/ili funkcija). Jedinični test znači da se softver sastoji iz jedinica koje su odvojeni (moguće ih je zasebno testirati) delovi proizvoda. Individualni program, klasa, metoda, funkcija itd. mogu biti takve jedinice.

Jedinični test (slika 1) omogućava proveru da li se jedinica ponaša onako kako je namenjeno i da li odgovara projektu specifikacije. Ovi testovi pružaju mogućnost samostalnog testiranja svake jedinice.

Iako se podrazumeva da se testiranje vrši posle implementacije, neka opšta preporuka je da se ove dve faze vrše paralelno.



Slika 2.1 Faza jediničnog testiranja softvera [Izvor: NM SE321-2020/2021.]

Pojednostavljeno gledano, to znači da se testovi za klase pišu i izvršavaju uporedo sa implementacijom samih klasa. Prednost je očigledna, kada se završi implementacija, dobijaju se već testirane i proverene klase.

Sa druge strane, ako se testiranje vrši posle implementacije, vrlo je verovatno da se neće izvršiti detaljno kao u toku implementacije. Razlozi mogu biti sledeći:

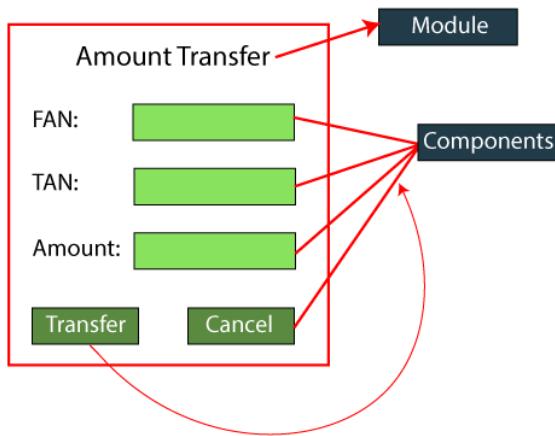
- **Nema vremena:** Pošto je implementacija jedna od poslednjih faza, kraj vremenskog roka za izradu softvera je obično veoma blizu.
- **Programeri često zaborave tačnu specifikaciju funkcionalnosti klase** i implementacione detalje, pa se ne sete svih situacija i ograničenja koja bi trebalo testirati.
- **Psihološka barijera zbog činjenice da se odjednom pišu testovi za sve klase** (već gotovog programa), i da je to ogroman posao.

PRIMER JEDINIČNOG TESTIRANJA

Jedinično testiranje na primeru modula za transfer novčanih sredstava

Pogledajmo primer kako bismo bolje razumeli koncept jediničnog testiranja.

Želimo da testiramo modul kojim se vrši transfer novčanih sredstava a njegov GUI je dat na slici 2. Modul se sastoji od više komponenti.



Slika 2.2 GUI modula za transfer novčanih sredstava [Izvor: NM SE321-2020/2021.]

FAN - označava broj računa sa kojeg vršimo prenos

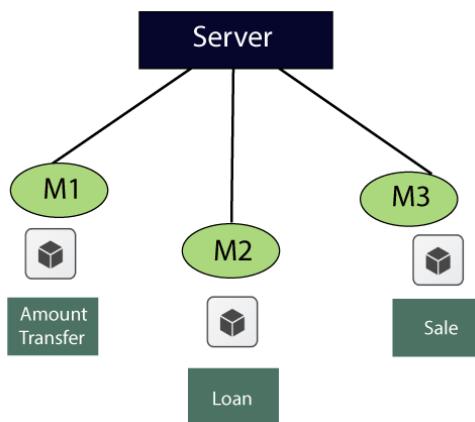
TAN - označava broj računa na koji vršimo prenos

Iznos - označava iznos koji prenosimo sa računa na račun

Cancel - služi sa prekid prenosa

Da bi korisnik u aplikaciji došao do ovog GUI potrebno je da

1. Unese URL za login stranicu
2. Unese Username/password a zatim klikne OK što će ga odvesti na home page
3. Da bi došao do modula za Transfer sredstava treba da prođe sledeći put:
Loans → Sales → Amount transfer (slika 3.)



Slika 2.3 Primer softvera za transfer novčanih sredstava koji se sastoji od tri modula: Loan (pozajmica), Sales (prodaja) i Amount Transfer (transfer iznosa) [Izvor: NM SE321-2020/2021.]

PRAVILA JEDINIČNOG TESTIRANJA

Prilikom testiranja se treba pridržavati određenih pravila i zahteva

Prilikom izvršavanja jedniničnog testiranja, potrebno je da se pridržavamo sledećih pravila:

- **Potrebno je da test izvršimo za pozitivnu vrednost**
- **Potrebno je da test izvršimo za negativnu vrednost**
- **Ne treba testirati ništa što je nepotrebno**
- **Prilikom testiranja ne vršimo nikakve prepostavke**

Kada smatramo da smo dostigli maksimalnu pokrivenost testom, zaustavljamo testiranje.

Testiranje počinjemo jediničnim testiranjem različitih komponenti kao što su:

- Sa računa broj (FAN)
- Na račun broj (TAN)
- Iznos
- Cancel

Zahtevi za testiranje komponenti su sledeći:

1. Broj računa (FAN) sa kojeg se vrši prenos - polje je tekstualno dužine 4 cifara.
2. Broj računa na koji se vrši transfer sredstava (TAN) - polje je tekstualno dužine 4 cifara.
3. Iznos (Amount) - polje je tekstualno dužine 4
4. Transfer - dugme koje omogućava transfer sredstava
5. Cancel - dugme kojim se ukida transfer sredstava

ZAHTEVI ZA TESTIRANJE KOMPONENTI

U sekciji su navedeni zahtevi za testiranje jediničnih komponenti GUI-a za prenos novčanih sredstava: FAN, TAN, Iznos, Transfer i Cancel

- 1. Testiranje FAN komponente:

Vrednost	Opis
1234	Prihvaćeno
4311	Poruka: Iznos je prihvatljiv ili ne
prazno	Poruka: unesi vrednost
5 cifara/ 3 cifre	Poruka: prihvata se samo 4 cifra
Alfanumerik	Poruka: prihvataju se samo cifre
Vrednost uneta sa Copy-Paste	Poruka: ukucaj vrednost
FAN isti kao i TAN	Poruka o grešci

Slika 2.4 Primeri testiranja FAN komponenete [Izvor: NM SE321-2020/2021.]

- 2. Za TAN komponentu:
 - Obezbedi vrednosti kao i za FAN komponentu
- 3. Za Iznos:
 - Obezbedi vrednost kao i za FAN i TAN komponentu
- 4. Za Transfer komponentu:
 - Unesi validnu vrednost za FAN
 - Unesi validnu vrednost za TAN
 - Unesi tačnu vrednost za Iznos
 - Klikni na dugme Transfer - transfer uspešno izvršen (poruka o potvrdi)
- 5. Za Cancel komponentu:
 - Unesi vrednost za FAN
 - Unesi vrednost za TAN
 - Unesi vrednost za Iznos
 - Klikni na dugme Cancel - svi podaci će biti izbrisani

ALATI ZA JEDINIČNO TESTIRANJE

Postoji veliki broj alata za jedinično testiranje kao što su: NUnit, TestNG, PHPUnit, Mockito

Alati za jedinično testiranje su nam potrebni kada želimo da pronađemo i potvrdimo identitet određenog modula ili jedinice koda. Uz pomoć ovih alata možemo da napravimo siguran dizajn i dokumentaciju i smanjimo broj grešaka. Generalno, jedinično testiranje je ručni proces, mada su sada neke firme uz pomoć ovih alata automatizovale jedinično testiranje. Korišćenjem alata za jedinično testiranje, možemo postići maksimalno testiranje pokrivenosti, performansi, kompatibilnosti i integracije. Svi alati za jedinično testiranje implementirani su kao dodatak za eclipse. Alati za jedinično testiranje programeri koriste za testiranje izvornog koda aplikacije. Najčešće korišćeni alati za jedinično testiranje su:

- **NUnit**: alat otvorenog koda i inicijalno prenet iz JUnit-a, koji radi za sve .Net jezike. NUnit je u potpunosti napisan na jeziku C# i redizajniran je tako da se mogu iskoristiti prednosti mnogih karakteristika .Net jezika.
- **JUnit**: još jedan okvir za jedinično testiranje otvorenog koda, koji je napisan na Java programskom jeziku. Uglavnom se koristi u razvojnog test okruženju. JUnit nudi anotaciju koja nam pomaže da pronađemo metodu ispitivanja. Ovaj alat pomaže da se poboljša efikasnost programera, pošto obezbeđuje konzistentnost razvojnog koda i smanjuje vreme otklanjanja grešaka.
- **TestNG (Test Next Generation)**: još jedan alat otvorenog koda koji podržava Javu i .Net programske jezike. TestNG je napredni alat za jedinično testiranje, koji je stimulisan iz okvira za testiranje JUnit i NUnit. Ipak, nekoliko novih funkcionalnosti (**Additional Annotation, Parallel Execution, Group Execution, Html Report, and Listener**) čine TestNG moćnijim alatom. U automatizovanim procesima testiranja, TestNG se koristi za rukovanje komponentama okvira i bachi izvršenje bez ikakvog ljudskog uplitanja.
- **PHPUnit**: još jedan alat za jedinično testiranje je PHPUnit, koji je napisan na PHP programskom jeziku. To je instanca xUnit arhitekture i zasnovana je na JUnit okviru. Može

- da generiše rezultate testa u mnogo različitih formata pomoću JSON, JUnit XML, TestDox protokola. Test slučajevi mogu da se pokrenu na operativnom sistemu sa više platformi.
- **Mockito:** To je okvir koji se koristi u jediničnom testiranju, a napisan je na programskom jeziku Java. Mockito je takođe alat otvorenog koda predstavljen MIT licencom ([Massachusetts Institute of Technology](#)). Uz pomoć Mockita možemo lako izvršiti testiranje aplikacije, tzv. mockito testiranje. Primarni cilj upotrebe ovog alata i načina testiranja je pojednostavljinjanje razvoja testova zanemarivanjem spoljnih zavisnosti jedinice koja se testira. Može se koristiti sa drugim okvirima za testiranje kao što su TestNG i JUnit.

PREDNOSTI I NEDISTACI JEDINIČNOG TESTIRANJA

Jedinično testiranje ima svoje prednosti ali i nedostatke

Jedinično testiranje može dati najbolje rezultate praćenjem sledećih koraka:

1. Slučajevi testiranja moraju biti nezavisni, jer ako dođe do bilo kakve promene u zahtevima, to neće uticati na slučajeve testiranja.
2. Konvencije o imenovanju jediničnih test slučajeva moraju biti jasne i dosledne.
3. Tokom jediničnog testiranja, identifikovane greške se moraju popraviti pre prelaska na narednu fazu SDLC-a.
4. U jednom trenutku treba testirati samo jedan kod.
5. Test slučajeve treba uskladiti sa pisanjem koda, jer ukoliko to ne učinite, broj putanja izvršenja će se povećati.
6. Ako postoje promene u kodu bilo kog modula, proverite da li je odgovarajući test jedinice dostupan ili ne za taj modul.

Ovde će takođe biti navedeni prednosti i nedostaci jediničnog testiranja.

Prednosti:

- Jedinično testiranje koristi modularni pristup zbog kojeg se bilo koji deo može testirati bez čekanja na završetak testiranja drugih delova.
- Da bi se razumeo API jedinice, razvojni tim se fokusira na funkcionalnost jedinice koju ona pruža i na to kako funkcionalnost treba da izgleda.
- Jedinično testiranje omogućava programeru da refaktoriše kod nakon nekoliko dana i osigura da modul radi bez grešaka.

Nedostaci:

- Ne može prepoznati grešku integracije ili na širem nivou jer se radi sa jediničnim kodom.
- U jediničnom testiranju, procena svih putanja izvršenja nije moguća, tako da jediničnim testiranjem nije moguće da se otkriju sve greške u programu.
- Najbolje je da se poveže sa ostalim aktivnostima testiranja.

✓ Poglavlje 3

Mocking testiranje

ŠTA JEDINIČNO TESTIRATI A ŠTA NE, U SLUČAJU VELIKE NASLEĐENE POSLOVNE APLIKACIJE?

Ne može se testirati cela aplikacija već jedinični testovi treba da pokriju kod sa „poslovnom logikom“ kome pristupaju drugi moduli, koji se često menja i kod koga se javlja mnogo grešaka

Kada znamo šta su jedinični testovi, zašto su oni potrebni i kako mogu pomoći programerima, verovatno ćete prepostaviti da je testiranje velike poslovne aplikaciju vaše kompanije koja sadrži hiljade redova koda takođe lako. Postavlja se pitanje odakle treba početi?

Pre pisanja pojedinačnih jediničnih testova, neophodno je tačno odrediti šta se testira. Velika poslovna aplikacija može da sadrži milijardu redova koda, pa da li je realno napisati testove za čitav kod? Svakako ne. Dakle, važno je da testovi budu usredsređeni na ono što je zapravo važno programerima, aplikaciji i korisnicima aplikacije.

Postavlja se pitanje šta je ono što NE TREBA jedinično testirati? To su:

- Druge biblioteke (treba prepostaviti da rade ispravno)
- Bazu podataka (trebalo bi prepostaviti da ispravno radi kada je dostupna)
- Ostale spoljne resurse (opet prepostaviti da rade ispravno kada su dostupni)
- Trivijalan kod (poput getera i setera, na primer)
- Kod koji ima nedeterminističke rezultate (tj. thread order or random numbers)

Šta TREBA jedinično testirati?

Jedno od zlatnih pravila jediničnog testiranja je da jedinični testovi treba da pokriju kod sa „poslovnom logikom“. To je deo koda u kojem se obično manifestuje većina grešaka. To je takođe deo koji se često menja jer se menjaju zahtevi korisnika i jer je to ono što je specifično za vašu aplikaciju.

Šta se zapravo događa ako najdete na zastarelju aplikaciju bez jediničnih testova? Šta ako deo „poslovne logike“ zauzima hiljade redova koda? Odakle da počnete? U ovom slučaju bi trebalo odrediti prioritete i napisati jedinične testove kako bi se testirao:

- Osnovni kod kome pristupaju mnogi drugi moduli
- Kod kod kojeg se javlja puno grešaka
- Kod koji menja više različitih programera (često radi prilagođavanja novim zahtevima)

Kad znamo na koja područja da se fokusiramo, možemo da počnemo da analiziramo koliko je testiranja potrebno da bismo bili sigurni u svoj kod.

POSTAVLJANJE LAŽNE MOCK ALTERNATIVNE REALNOSTI

Lažni (mocking) objekti su dizajnirani da „zavaraju“ Java objekt (koji se jedinično testira) tako da se ima utisak da on komunicira sa drugim stvarnim objektima.

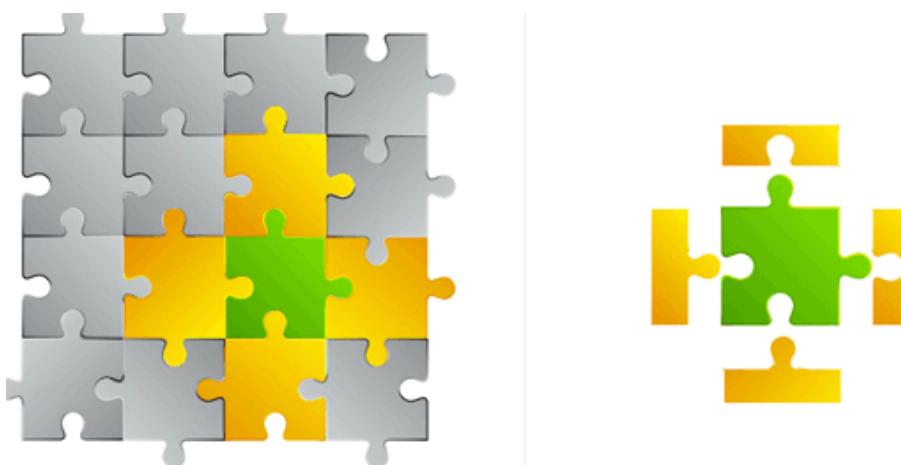
Jedinično testiranje je dobro koristiti kada započnete pisanje nove aplikacije kod koje je sve čisto i netaknuto. Ali šta ako nasledite neuređen kod neke velike poslovne aplikacije? Taman kad ste pomislili da sve radi idealno, javlja se problem... gde u složenim i zastarem aplikacijama započeti implementaciju jediničnih testova?

Na primer, ako se u takvoj aplikaciji nešto promeni, kako možete da znate na šta će to uticati? Neželjeni efekti vaših promena treba da budu jasni od početka, što se postiže putem Mocking-a, koji omogućava kreiranje alternativne realnosti za vašu aplikaciju, tako da ona može da se testira bez ikakvih neželjenih efekata ili posledica u realnosti (nekako poput Trumanovog šoua (engl. **Truman show**).

Zašto je upotreba **Mocking okvira** (framework) bolja od ručnog rada i nekih dostupnih alata?

Mocking pokušava da na lak način reši kreiranje lažnih objekata (klasa) koji pomažu procesu jediničnog testiranja. Kao i u poređenju sa Trumanovim šouom, lažni objekti su dizajnirani da „zavaraju“ Java objekt (koji se jedinično testira) tako da se ima utisak da on komunicira sa drugim stvarnim objektima. Sa stanovišta testiranog objekta, sve teče normalno - on komunicira sa drugim objektima i dobija sve odgovarajuće odgovore. Međutim, to je samo predstava koju ste pažljivo isplanirali.

Lažni objekti omogućavaju da postavite kontrolisano okruženje sa strogo definisanim determinističkim ponašanjem svih objekata koji su uključeni u objekat koja se testira. Mocking se vizuelno može predstaviti kao na slici 1:



Slika 3.1 Mock alternativna realnost [Izvor: NM SE321-2020/2021.]

S leve strane slike je predstavljen realni sistem sa klasom koja se testira (označena zelenom bojom), dok su klase od kojih ona zavisi predstavljene žutom bojom, a klase od kojih ne zavisi sivom. S desne slike je predstavljena Mock alternativna realnost sa klasom koja se testira

(takođe označene zelenom bojom), dok žute klase označavaju mock-ove koji se koriste za jedinično testiranje.

KADA TREBA KORISTITI MOCKING?

Mockito okvir se koristi kako bi se stvorilo okruženje u kojem se pažljivo mogu nadgledati sve interakcije testirane klase kako bi se proverila njena efikasnost.

Mockito okvir se koristi kako bi se stvorilo okruženje u kojem se pažljivo mogu nadgledati sve interakcije testirane klase kako bi se proverila njena efikasnost. Tako se može utvrditi da li učesnici pravilno izvršavaju svoje zadatke.

Upotreba mocking-a se predlaže kada klasa koju želite da jedinično testirate komunicira sa drugim klasama koje imaju neželjene efekte koji mogu biti pozvani samo u stvarnom produkcionom okruženju npr. punjenje kreditne kartice ili bankovnog računa, štampanje medicinskih kartona, računa, ličnih dokumenata, slanje zahteva spoljnem sistemu. Mockito možete koristiti i u drugim slučajevima. Uobičajena praksa je i mocking nekih klasa koje su vrlo spore i baziraju se na mnogo zavisnosti koje nisu neophodne prilikom jediničnog testiranja. Mocking je takođe dobar i kada želite da simulirate ne tako uobičajene greške kao što su pun hard disk, greška u mreži itd.

Kada ne treba koristiti mocking?

Uvek imajte na umu da treba mock-ovati objekte s kojima klase dolaze u kontakt, jer vas zanima sama klasa i ništa drugo. Ako zapravo želite da vidite interakciju celog modula, tada su vam potrebni testovi integracije, o čemu će kasnije biti govora.

Stoga, ne bi trebalo da koristite mocking kada vas zanimaju:

- Podaci koji dolaze iz spoljnih resursa ili dB
- Transakcije
- Interakcija vaše aplikacije sa serverskom aplikacijom ili okruženjem
- Testiranje više modula zajedno kao jedne komponente

Za sve ove slučajeve su potrebni testovi integracije, jer se tada gleda šire od samih klasa. Ako postoji nekoliko objekata koji ispravno rade izolovano (jer prolaze jedinične testove), to ne znači da će oni raditi ispravno kada se koriste zajedno što treba verifikovati testovima integracije.

▼ Poglavlje 4

Testiranje Integracije

INTEGRACIONO TESTIRANJE

Jedan od najvažnijih aspekata softverskih razvojnih projekata je integraciona strategija.

Jedan od najvažnijih aspekata softverskih razvojnih projekata je integraciona strategija.

Integracija može biti izvedena odjednom, od vrha ka dnu ili od dna ka vrhu ili drugom integracionom strategijom. Generalno, što su veći projekti, to je integraciona strategija važnija. Veoma mali sistemi često su sakupljeni i testirani u jednoj fazi.

Za većinu realnih sistema, ovo je nepraktično iz dva bitna razloga. Prvo, sistem bi pao na mnogo mesta odjednom, tako da pokušaji debagovanja i retestiranja bi bili potpuno nepraktični.

Dруго, задовољавање критеријума тестирања „беље кутије“ би било веома тешко, због огромне количине детаља разједињавањем улазних датотека.

Велики системи могу захтевати велики број интеграционих фаза, почеvши од сакупљања модула у ниско рангираним подсистемима, па онда окупљање подсистема у веће подсистеме и коначно склapanje подсистема виših нивоа у целокупан систем. Да би биле ефикасније, технике интеграционог тестирања треба да се добро укlope у целокупну интеграциону стратегију.

У вишефазној интеграцији, тестирање у свакој фази помаже да се грешка уочи раније, а систем држи под контролом. Изводећи појединачно тестирање у раној интеграционој фази, а затим користећи rigoroznije критеријуме у finalnoj etapi, реална је само варијанта visokog rizika, poznata kao „big bang“ (veliki prasak) интеграционо тестирање. Извођење rigoroznih тестова на нивоу целог softvera у свакој интеграционој фази, ангажује доста nepotrebног dupliranja truda. Rešenje je premostiti целокупну интеграцију система, ostvariti strogo тестирање у свакој фази и smanjiti dupliranje truda.

RELACIJE IZMEĐU JEDINIČNOG I INTEGRACIONOG TESTIRANJA

Jedinično i integraciono tестирање се могу вршити одвојено или могу бити kombinovani, тако што се детаљи извршавања сваког модула потврђују у интеграционом контексту.

Važno je razumeti relacije između jediničnog testiranja (modula) i integracionog testiranja. Sa jedne strane, jedinice (moduli) su testirani pre nego što je pokušana bilo koja integracija. Tokom integracionog testiranja se u potpunosti koncentrišemo na interakciju modula, smatrujući da su detalji u svakom modulu tačni.

Sa druge strane, moduli i integraciono testiranje mogu biti kombinovani, tako što se detalji izvršavanja svakog modula potvrđuju u integracionom kontekstu.

Oba gledišta integracionog testiranja mogu biti prikladna za bilo koji zadati projekat, tako da bi integracioni metodi testiranja trebalo da budu dovoljno fleksibilni i da se prilagode svima. Najjednostavnija primena strukturnog testiranja je kombinovanje jediničnog testiranja sa integracionim testiranjem tako da osnovni setovi putanja kroz sve jedinice (module) budu izvršeni u integracionom kontekstu.

Kriterijumi jediničnog testiranja često mogu biti uopšteni na nekoliko mogućih načina. Kao što je diskutovano, najčešće uopštavanje je zadovoljiti kriterijume jediničnog testiranja u integracionom kontekstu, koristeći ceo program kao sredinu za testiranje drafverima za svaki modul. Međutim, ovo trivijalno uopštavanje ne pravi razliku između jediničnog testiranja i integracionog testiranja.

Korisnije uopštavanje je prilagođavanje kriterijuma jediničnog testiranja, tako da oni budu fokusirani na interakciju između modula, a ne na pokušaj testiranja svih detalja svakog od izvršavanja modula u kontekstu integracije.

Iako je strukturno testiranje detaljnije, pristup je isti. Dok primena strukturnih tehnika kod jediničnog testiranja zahteva da celokupna logika toka kontrolnog grafa u testiranoj jedinici (modulu) bude testirana nezavisno, kod integracionog testiranja se vrši odgovarajuće uopštavanje tako da se uključuje samo određena logika koja se odnosi na poziv ostalih modula. Ovo navodi na zaključak da ove dve vrste testiranja treba vršiti nezavisno.

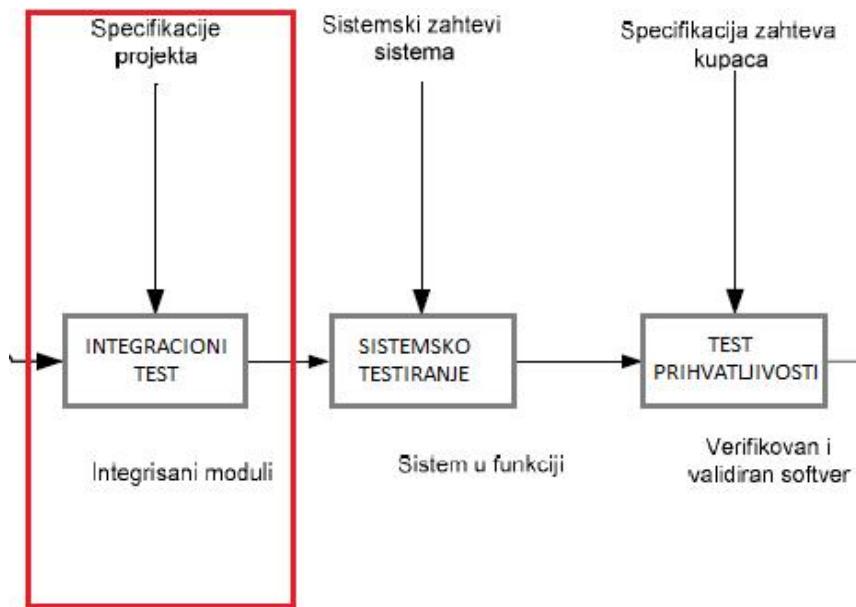
STRATEGIJE INTEGRACIONOG TESTIRANJA SOFTVERA

Klasične strategije (top-down ili bottom-up) se koriste kod hijerarhijski strukturiranog softvera, dok su moderne strategije više arhitektonski struktuirane

Integraciono testiranje je proces verifikovanja interakcije između softverskih komponenti.

Klasične strategije integralnog testiranja, kao što su top-down ili bottom-up, se koriste kod tradicionalnog, hijerarhijski strukturiranog softvera. Moderne sistematične integrativne strategije su više arhitektonski struktuirane, što implicira integrisanje softverskih komponenti ili podistema na bazi identifikovanih funkcionalnih delova.

Integraciono testiranje (Slika 1) je kontinualna aktivnost u kojoj softver-inženjeri moraju umesto posmatranja na nižem nivou, da se koncentrišu na perspektivu na integralnom nivou.



Slika 4.1 Faza integracionog testiranja softvera [Izvor: NM SE321-2020/2021.]

Osim za male, jednostavnije softvere, integralne test strategije se obično izvode putem testiranja svih komponenata zajedno u istom trenutku.

Glavna prednost bottom-up pristupa je lako nalaženje greški. Prednost kod top-down je lakše uočavanje nedostataka u celini (kako se razvoj projekta približava kraju tako na površinu isplivaju nedostaci). Proces se ponavlja sve dok se komponenta na vrhu hijerarhije ne istestira. U ovom slučaju softver-inženjer mora umesto posmatranja na nižem nivou, da se koncentriše na perspektivu na integralnom nivou.

Posle jediničnog testiranja i ako svi moduli rade individualno ispravno, to ne znači da će nakon integracije raditi dobro zbog pojavljivanja problema interfejsa između modula. Podaci mogu biti izgubljeni kroz interfejs, jedan modul može imati neželjene efekte na drugi, kombinovani moduli ne produkuju željenu funkciju, javljaju se problemi sa globalnim strukturama podataka itd. Integracioni test uzima jedinično testirane module i od njih gradi programsku strukturu diktiranu od dizajna za testiranje koje treba da otkrije greške koje se pripisuju interfejsu između modula.

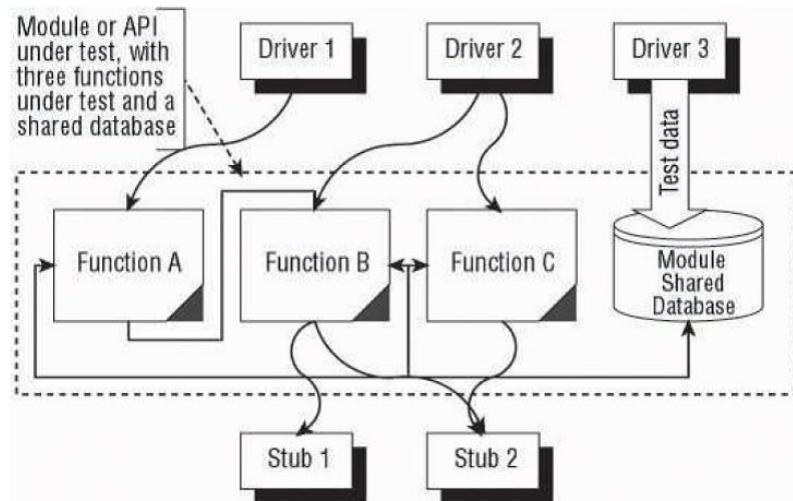
KONFIGURACIJE ZA INTEGRACIONA TESTIRANJA

Pri testiranju interakcije, ne moraju svi moduli koji čine sistem biti završeni. U tom slučaju, nedostajući moduli moraju biti simulirani korišćenjem drajvera i stub-ova

Pri testiranju integracije ne moraju svi moduli koji čine sistem biti završeni. U tom slučaju, nedostajući moduli moraju biti simulirani.

Sem toga, pri testiranju modula, komponenti ili integracionom testiranju ili uvek kad želite da testirate kod posredstvom API-a (**Application Program Interface**-a), potrebno je da simulirate nedostajuće delove ili tokove podataka koji nedostaju.

Drajveri i stub-ovi koji zamenjuju nedostajuće delove sistema. Relacija drajvera, modula koji se testiraju i stub-ova je data na sledećoj slici 2.



Slika 4.2 Drajveri i Stub-ovi [Izvor: NM SE321-2020/2021.]

Drajver je funkcija ili simulacioni modul koji šalje i/ili prima podatke od modula koji se testiraju. **Stub** je funkcija ili simulacioni modul kome moduli koji se testiraju šalju podatke i/ili od koga primaju podatke.

Kolekcija drajvera i stubova (sa drugim hardversko-softverskim modulima) za testiranje integracije se naziva **konfiguracija integracionog testiranja**, a često i probni sto ili maketa za testiranje (engl. Harness, mock-up).

Konfiguracije za integraciona testiranja, ili drajveri, stubovi, makete i dr. služe da možete pristupiti jednom modulu (ili grupi modula) u okviru sistema, a takođe da ih izolujete od ostatka, tako da defekti koje otkrivate budu defekti modula koji se testiraju.

Konfiguracije za integraciona testiranja ne moraju biti sofisticirane, štaviše, zbog cene je od interesa da budu što jednostavnije.

No, one ne smeju sa svoje strane unositi defekte u module koji se testiraju (defekti koji se ne manifestuju u drugim okruženjima modula), dakle moraju biti testirani.

Dakle, prepostavlja se da su konfiguracije za integraciona testiranja jednostavne i da su testirane.

VRSTE ITEGRACIONOG TESTIRANJA

Postoje četiri načina integracije sistema: Integracija odozgo nadole (top-down), Integracija odozdo nagore (bottom-up), Big-Bang Integracija i Bekbon (Backbone) Integracija

Testiranje integracije se vrši radi provere i potvrde da komponente sistema korektno kooperiraju, u skladu sa zahtevima za sistem.

Prvi nivo integracionog testiranja obično vrši razvojni tim da bi obezbedio korektno funkcionisanje sistema. Nakon toga testiranje može vršiti tim za kontrolu kvaliteta.

Testiranje integracije zavisi od načina integracije, ili ono što nazivamo "integracijom" zavisi od toga šta zovemo "sistemom".

Kao što je već ranije spomenuto, postoje četiri načina integracije sistema, koje se biraju zavisno od projekta i koje definišu i konfiguracije za integraciona testiranja. To su:

1. Integracija odozgo nadole (**top-down**)
2. Integracija odozdo nagore (**bottom-up**)
3. Big-Bang Integracija
4. Bekbon (**Backbone**) Integracija

i o njima će se ovde detaljnije govoriti.

INTEGRACIJA ODOZGO NA DOLE

Integracija počinje s osnovnim modulom u sistemu, a nakon toga se bira sledeći niži modul koji će biti integrisan i testiran. Umesto drajvera, top down integracija koristi stub-ove

Integracija počinje s najvišim ili osnovnim modulom u sistemu. Nakon toga se bira sledeći modul koji će biti integrisan, pri čemu je jedino pravilo da to mora biti niži modul po subordinaciji i da treba da bude testiran.

Najviši nivo je obično interfejs prema korisniku, ali može biti bilo koji viši nivo arhitekture. Umesto drajvera, top down integracija koristi stub-ove. I u ovom slučaju možete imati dobru izolaciju defekata ukoliko idete korak po korak i dovoljno pažljivo.

U primeru sistema koji se ovde razmatra (slika 3), modul A je osnovni modul koji treba da se testira. U tu svrhu se kreira stab koji zamjenjuje module B,C,D. To ne mora biti lak zadatak, jer ako stab nije dovoljno reprezentativan, testiranje neće biti verodostojno. U protivnom, može se desiti da verodostojni stab mora da bude složen, a ako se po složenosti približi modulu koji treba da zameni, postavlja se pitanje njegovog testiranja i da li je opravdanosti dupliranja posla (razvoj modula + staba):

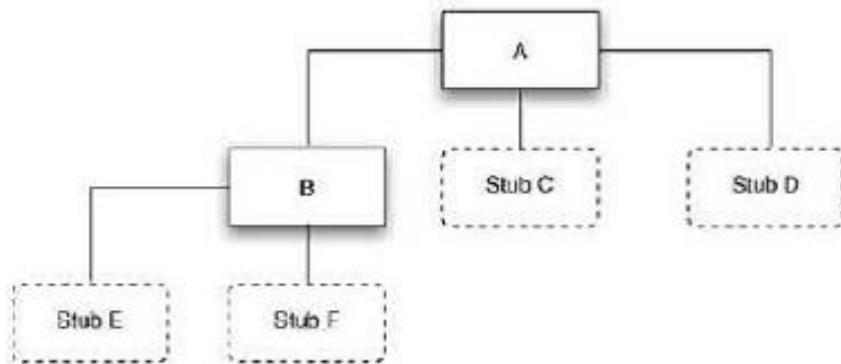
Nakon testiranja modula A, stab je zamenjen realnim modulom, a njemu je dodat odgovarajući stab. Ima više mogućnosti kako nastaviti. Npr. kako su B,C,D paralelni, jedan tester može uzeti kombinaciju A,B, a ostale kombinacije drugi testeri.

U opštem slučaju nema najboljeg redosleda integracije, no mogu se dati sledeće preporuke (Slika 3):

1. Ako postoje kritični moduli na pr. B (sa stanovišta uspešnosti celog sistema ili projekta), oni treba da budu što ranije integrisani.
2. Moduli sa I/O funkcijama treba da budu što ranije integrisani.

Nedostaci prilaza su sledeći: Ako su defekti na samom dnu, nećete ih otkriti sve do poslednje etape testiranja. Na primer, ako baza podataka ima spori odziv za realistične podatke, možda

ćete morati da ograničite funkcionalnosti garfičkog interfejsa sa korisnikom. Možda testovi upotrebljivosti (**usability tests**) koje ste na početku uradili neće više biti adekvatni.



Slika 4.3 Integraciona celina u primeru (međukorak) [Izvor: NM SE321-2020/2021.]

INTEGRACIJA ODOZDO NA GORE

Počinje se od razvijenih modula najnižeg nivoa koji se integrišu u celine koje se mogu instalirati, izvršavati i testirati. U tu svrhu se razvijaju i odgovarajući drajveri.

Najniži nivo "dno" (**bottom**) je definisano arhitekturom sistema koji se testira, i to je često nivo modula koji komuniciraju sa hardverom ili bazom podataka.

Počinje se od razvijenih modula najnižeg nivoa koji se integrišu u celine koje se mogu instalirati, izvršavati i testirati. U tu svrhu se razvijaju i odgovarajući drajveri. Kako ste na najnižem nivou sistema, interakcija sa hardverom ili bazom podataka postoji, pa je potreba za stub-ovima minimalna. Tako dobijena celina se testira.

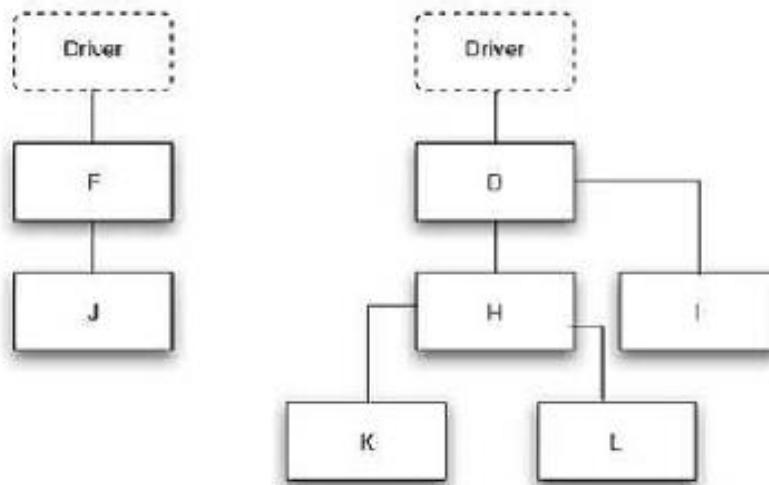
Proces se ponavlja inegriranjem novih modula i/ili celina. Može se napredovati nivo po nivo, ili deo po deo na datom nivou. Izbor koliko modula i koje module dodati u sledeću celinu zavisi od rizika pojave defekata integracije, potrebe za što lakšom izolacijom bagova i potrebnog vremena.

Ako su moduli F i D kritični, tada se integraciona celina može napraviti prema slici 4. Za oba modula se kreira drajver koji šalje test ulaze, poziva testirani modul i prikazuje rezultate (i/ili ih upoređuje sa očekivanim vrednostima):

Svaka nova celina za testiranje u principu zahteva nove drajvere. Proses se ponavlja sve dok se ne završi integracija sistema. **Ovaj prilaz nudi mogućnost lake izolacije defekata.**

No, ima i potencijalnih nedostataka. Ako je defekt na višim nivoima sistema, nećete ga otkriti **sve do poslednje etape integracije i testiranja.** Primer, ako garfički interfejs sa korisnikom (**user interface**) nije dovoljno jasan i adekvatan, može se desiti da ćete morati da menjate arhitekturu sistema. Time ćete ujedno morati da ponovite sve dotada urađene integracione testove.

Oseća se i nedostatak skeletnog programa na kome se mogu demonstrirati funkcije sistema. U stvari sve do kraja integracije sistem ne postoji, a i tada se testiranja vrše na realnom sistemu.



Slika 4.4 Integraciona celina u primeru (međukorak) [Izvor: NM SE321-2020/2021.]

BIG-BANG INTEGRACIJA

U ovom prilazu svi moduli koji sačinjavaju sistem se sakupe i integrišu odjednom.

U Big-Bang Integraciji svi moduli koji sačinjavaju sistem se sakupe i integrišu od jednom. Srećom, svi moduli su već zadovoljili prethodni nivo testiranja modula (**unit testing**). Nakon rešavanja problema integracije koje vam kompjajler i linker ne dozvoljavaju da ih zanemarite, imate sistem koji radi.

Prednosti ovog metoda su da je brz i jeftin. Nema ni drajvera ni stub-ova ni maketa ... Nije potrebna posebna pažnja i vreme pri pojedinačnoj integraciji, testiranju, dodavanju novih modula, ponovnom testiranju. I ove prednosti često prevagnu nad nedostacima metoda.

Ali, ako se pojavi problem, kako locirati defekt? Koji modul (ili koji moduli) ga poseduju? Može biti potrebno veoma obimno debagiranje radi korekcije

Drugo, ušteda na vremenu može biti čista iluzija, posebno za velike sisteme. Vreme do postizanja adekvatnog nivoa kvaliteta može biti duže nego što bi bilo da su napravljeni stubovi, drajveri i drugo na konfiguracijama za testiranje i sprovedena testiranja. Ova činjenica je dobro uočena u tzv. "**test driven development**" softveru.

Treće, čeka se da svi moduli budu završeni da bi počela integraciona testiranja, što odlaže i otkrivanje bagova. Ako se zna da je "što kasnije defekt otkriven, to su troškovi veći", zašto čekati ako se ne mora?

Izuzev za male sisteme ili manje modifikacije tokom održavanja, big-bang je obično pogrešna ekonomija vremena i novca.

Međutim, za manje sisteme i manje modifikacije, mogućnost uštede može prevagnuti nad povećanim rizikom u pogledu kvaliteta.

BEKBON INTEGRACIJA

Za razliku od postupaka gde se integracija vrši na osnovu arhitekture sistema, bekbon integracija polazi od tehničke i biznis procene rizika.

Bekbon integracija: Za razliku od postupaka gde se integracija vrši na osnovu arhitekture sistema, bekbon integracija polazi od tehničke i biznis procene rizika. Za prvu integracionu celinu (prvi bekbon) se biraju najkritičniji moduli, tj. oni čiji otkaz može dovesti u pitanje funkcije sistema i uspeh projekta. Za prvi bekbon mogu biti potrebni i drajveri i stabovi.

U sledećim koracima se dodaju moduli svakom od narednih bekbona. Naravno, puno ima smisla da se uključe moduli koji komuniciraju sa modulima koji su već integrисани, a koji su na sledećem nivou po kritičnosti. Ako je analiza rizika dobra, imate šanse da testiranje bude zaista efikasno.

Sendvič Integracija: Ova metoda integracije se sastoji u kombinaciji top-down i down-top prilaza, što uglavnom najčešće odgovara praksi. Sama po sebi ne unosi ništa novo u odnosu na dve pomenute tehnike.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER INTEGRACIONOG TESTIRANJA

Primer sistema na online prodaju

Pretpostavimo da radite za IT organizaciju koja treba da razvijete veb site za online kupovinu kompanije Camp World, koja se bavi prodajom opreme za kampovanje.

Nakon završetka faze prikupljanja zahteva, analize i projektovanja, pojedinačnim programerima je dodeljen zadatak da razviju po jedan od sledećih modula:

1. Registracija i autentifikacija / prijava korisnika
2. Katalog proizvoda
3. Kupovina (Shopping Cart)
4. Plaćanje
5. Integracija sa bankom za potrebe plaćanja
6. Isporuka i praćenje isporuke

Nakon što je svaki modul dodeljen programeru, programeri su počeli sa kodiranjem funkcionalnosti modula na svojim sopstvenim računarima. Pošto su završili razvoj, testirali su funkcionalnosti svojih modula u sklopu jediničnog testiranja i pronašli neke defekte, koje su otklonili. Nakon toga su smatrali da su njihovi moduli kompletirani. QA Manager je tada predložio da se izvrši integraciono testiranje kako bi se potvrdilo da svi moduli rade zajedno.

Kada su programeri svoj kod postavili na zajedničku mašinu, otkrili su da aplikacija ne radi kako se očekivalo, jer pojedinačni moduli nisu dobro radili kada ih je trebalo integrisati. Pojavile su se sledeće greške:

- nakon prijavljivanja, korisnička korpa (shopping card) nije prikazivala stavke koje je korisnik prethodno dodao,
- iznos računa nije uključivao troškove isporuke itd.

Testiranje integracije nam pomaže da identifikujemo, rešimo probleme i osiguramo da aplikacija u celini radi kako se očekuje.

Integraciono testiranje je posebno važno kada treba integrisati neki softver sa postojećim sistemom preduzeća nezavisnog proizvođača ([third - party enterprise systems](#)). Kao jedna od ključnih tačaka koje osiguravaju uspeh projekta, integraciono testiranje uključuje slučajevе testiranja za integraciju komponenti (pokrivaju interakciju između integrisanih modula u jednom sistemu) i testove sistemskih integracija (pokrivaju integraciju nekoliko sistema koji međusobno deluju).

TEHNIKE INTEGRACIONOG TESTIRANJA

Šta se može primeniti tokom integracionog testiranja

Za integraciono testiranje se može primeniti bilo koja tehnika Black box, White box, i Grey box testiranja. Neke od njih su:

Black Box testiranje:

1. [State Transition technique](#)
2. [Decision Table Technique](#)
3. [Boundary Value Analysis](#)
4. [All-pairs Testing](#)
5. [Cause and Effect Graph](#)
6. [Equivalence Partitioning](#)
7. [Error Guessing](#)

White Box testiranje:

1. [Data flow testing](#)
2. [Control Flow Testing](#)
3. [Branch Coverage Testing](#)
4. [Decision Coverage Testing](#)

✓ Poglavlje 5

Sistemsko testiranje

ŠTA JE SISTEMSKO TESTIRANJE?

Sistemsko testiranje je formalni proces koji izvodi tim za testiranje sa ciljem da utvrди logičku ispravnost i svrshodnost testiranog programa.

Sistemsko testiranje ima veliki značaj, jer se nakon ove faze softver isporučuje naručiocima pa je:

- obaveza proizvođača je da isporuči kvalitetan softver (u interesu i krajnjeg korisnika i proizvođača)
- cilj testiranja je otkrivanje grešaka u softveru pre isporuke softvera (u svim fazama se ulažu napori za prevazilaženje nedostataka)

Sistemsko testiranje je formalni proces koji izvodi tim za testiranje sa ciljem da utvrdi logičku ispravnost i svrshodnost testiranog programa.

- sprovodi se na računaru uz poštovanje određenih test procedura koje se primenjuju na određene test slučajeve
- testiranjem se značajno smanjuju gubici koje proizvođači softvera imaju usled grešaka i otkaza softvera nastalih nakon njegove isporuke kupcu
- proces testiranja je skup i dugotrajan i zato svi proizvođači softvera ulažu velike napore da ga učine što efikasnijim

Cilj sistemskog testiranja je provera da li sistem u potpunosti ispunjava sve zahteve korisnika, tako da može da bude isporučen (da li se razvija potreban sistem?).

Sistemsko testiranje je:

- najviši, završni nivo testiranja
- provera da li se sistem, kao celina, ponaša u skladu sa specifikacijom zahteva koje je postavio kupac
- većina funkcionalnih zahteva je već proverena na nižim nivoima testiranja, pa je naglasak na nefunkcionalnim zahtevima (brzina, pouzdanost, efikasnost, veze prema drugim aplikacijama itd.)
- u proces sistemskog testiranja se mora uključiti ceo razvojni tim pod kontrolom rukovodioca projekta.

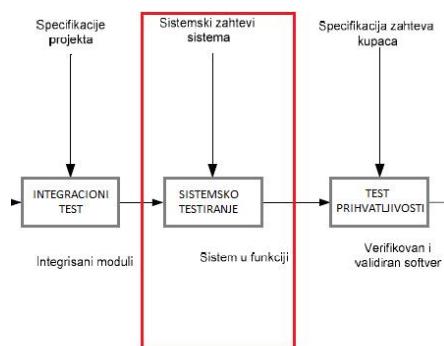
Posebna pažnja se posvećuje testiranju nefunkcionalnih zahteva proverom načina na koji se funkcije izvršavaju:

- performanse, bezbednost i pouzdanost sistema se mere u odnosu na ciljeve koje je postavio kupac.

SISTEMSKO TESTIRANJE - FUNKCIONALNO TESTIRANJE

U ovoj fazi se zanemaruje struktura sistema i usredsređuje se na funkcionalnost.

Sistemsko testiranje (Slika 1) se bavi ponašanjem celog sistema. Većina funkcionalnih otkaza trebalo bi da su već identifikovani tokom jediničnog testiranja i integracionog testiranja. Sistemsko testiranje se, kao što je rečeno, obično smatra prikladnim za poređenje sistema sa nefunkcionalnim sistemskim zahtevima, kao što su sigurnost, brzina, preciznost ili pouzdanost. Spoljašnji interfejs ka drugim aplikacijama, pomoćnim programima, hardverskim uređajima, ili operativnim okruženjima se takođe razmatra na ovom nivou.



Slika 5.1 Faza funkcionalnog testiranja softvera [Izvor: NM SE321-2020/2021.]

Preduslovi za testiranje sistema:

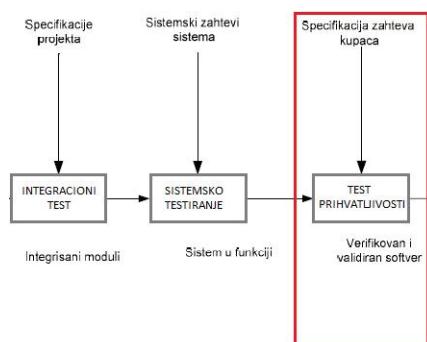
- sve komponente treba da su istestirane jediničnim testovima,
- da je izvršena uspešna integracija komponenti,
- kreirano produkciono okruženje.

U ovoj fazi se zanemaruje struktura sistema i usredsređuje se na funkcionalnost. Sada je pristup bliži zatvorenoj nego otvorenoj kutiji. Nije potrebno znati koja se komponenta izvršava, već šta bi sistem trebalo da radi. Prema tome, funkcionalni test se zasniva na funkcionalnim zahtevima sistema. Svakoj funkciji se mogu pridružiti one komponente sistema koje je izvršavaju. Za neke funkcije to može da obuhvati i ceo sistem.

TESTIRANJE PRIHVATLJIVOSTI I INSTALACIONO TESTIRANJE

Testiranje prihvatljivosti proverava ponašanje sistema prema zahtevima kupaca, bez obzira kako oni bili iskazani; Instalaciono testiranje se sprovodi prema zahtevima hardverske konfiguracije

Testiranje prihvatljivosti (acceptance testing) proverava ponašanje sistema prema zahtevima kupaca, bez obzira kako oni bili iskazani (Slika 2). Kupac podnosi ili specificira tipične zadatke radi provere da li su njihovi zahtevi ispunjeni. Ova aktivnost testiranja može, ali i ne mora, da uključuje razvojni tim sistema. Pre testa prihvata, aplikacija mora biti u potpunosti razvijena. Druge faze testiranja (jedinično, integraciono i funkcionalno) su završene.



Slika 5.2 Faza test prihvatljivosti [Izvor: NM SE321-2020/2021.]

Na slici 2 je obeležena faza testa prihvata. U ovoj fazi se stavlja fokus na simulaciju stvarnog okruženja i testira se ponašanje aplikacije. Test slučajevi su napisani po scenarijima koji će se najčešće koristiti u stvarnom okruženju.

Uobičajeno se, nakon kompletiranja softverskog dela i testa prihvatljivosti, softver verifikuje po instalaciji u ciljnem okruženju. **Instalaciono testiranje** može se posmatrati kao sistemsko testiranje sprovedeno prema zahtevima hardverske konfiguracije. Instalacione procedure mogu takođe biti predmet provere. Instalaciono testiranje najviše je testirano u području testiranja. Ovaj tip testiranja je predviđen kako bi se osiguralo da su sve mogućnosti i opcije pravilno instalirane. Takođe je predviđeno da proveri da li su sve komponente aplikacija pravilno instalirane.

Slika 5.3 Faza instalacionog testiranja [Izvor: NM SE321-2020/2021.]

Kada je u pitanju vrlo mali sistem instalacija se može postaviti direktno, dok kod većih sistema postoji nekoliko načina za instalaciju (implementaciju) sistema (slika 3).

TESTIRANJA VIŠEG REDA

Pored raznih tehnika i metoda testiranja softvera razvijeni su i mnogi alati koji olakšavaju ili potpuno automatizuju proces testiranja.

U testiranja višeg reda spadaju:

- Testiranje sigurnosti (**security testing**): da li su funkcije dostupne onim i samo onim korisnicima kojima su i namenjene.
- Testiranje količine podataka-verifikovanje da li softver može obraditi veliku količinu podataka.
- Testiranje upotrebljivosti (**usability testing**) – estetski aspekti, konzistentnost korisničkog interfejsa, korisnička dokumentacija.
- Testiranje integriteta (**integrity testing**) – robustnost (otpornost na otkaze), konzistentna upotreba resursa i sl.
- Test u stresnim uslovima (**stress testing**) – vrsta testa pouzdanosti sistema pod nenormalnim uslovima (velika opterećenja sistema, nedovoljno memorije ili drugih resursa, neraspoloživi servisi i sl.).
- Etalonski test - upoređivanje performansi novog sistema sa nekim, referentnim, poznatim sistemom.
- Test zagušenja – provera da li sistem može da zadovolji višestruke zahteve različitih aktera za istim resursom.
- Test opterećenja – vrsta testa performansi kojim se procenjuju operativni limiti nepromenljivog sistema pod različitim opterećenjima ili različitim konfiguracijama sistema pri istom opterećenju. Najčešće se mere protok i vreme odziva (srednja i granična vrednost).
- Test konfiguracije (**configuration testing**) – testira ponašanje softvera u različitim hardversko/softverskim okruženjima.
- Test instalacije (**installation testing**) – testira instaliranje softvera na različitim sistemima i u različitim situacijama (npr. prekid napajanja ili nedovoljno prostora na disku).

✓ Poglavlje 6

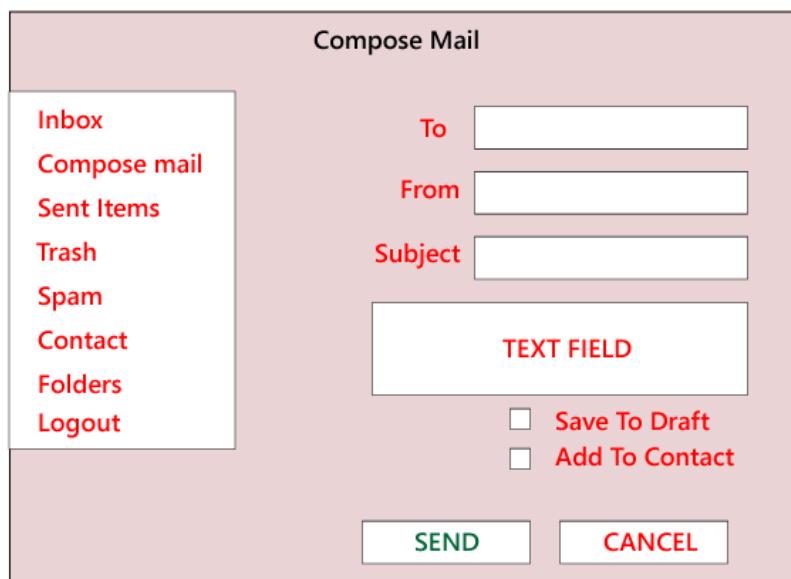
Pokazna vežba

1. PRIMER INTEGRACIONOG TESTIRANJA

Testiranje Gmail aplikacije

Prepostavimo da imamo Gmail aplikaciju na kojoj vršimo testiranje integracije.

Prvo ćemo uraditi funkcionalno testiranje na stranici za login koja uključuje komponente kao što su korisničko ime, lozinka, dugme za slanje (SEND) i otkazivanje (CANCEL). Tada možemo da izvršimo testiranje integracije pod pretpostavkom da je svaka opcija (Inbox, Compose mail, Sent items, Trash, Spam, Contact, Folder i Logout) iz ponuđenog menija



Slika 6.1 Interfejs Gmail aplikacije za slanje elektronske pošte [Izvor: NM SE321-2020/2021.]

sa leve strane GUI-a na slici 1. realizovana kao poseban modul.

Različiti scenariji integracije su sledeći:

Scenario 1:

- Prvo se prijavljujemo kao korisnik P, kliknemo na Compose mail i izvšimo funkcionalno testiranje za određene komponente (prikazane sa desne strane slike).
- Sada kliknemo na Send a takođe proverimo i Save Drafts.
- Pošto pošaljemo poštu korisniku Q, proveravamo u Send Items folderu korisnika P da li je poslata pošta tamo.

- Sada ćemo se odjaviti kao P i prijaviti se kao korisnik Q i prebaciti se u njegov Inbok kako bi proverili da li je pošta stigla.

Scenario 2:

Takođe vršimo testiranje integracije na Spam folderu. Ako je određeni kontakt označen kao neželjeni (spam) sadržaj, svaka pošta koju je taj korisnik poslao treba da ide u neželjenu poštu (spam folder), a ne u inbok.

Vreme trajanja 15 minuta.

2. PRIMER INTEGRACIONOG TESTIRANJA

Testiranje modula za dodavanje, brisanje, izmenu i listanje korisnika

Ovaj primer integracionog testiranja se odnosi na testiranje modula za dodavanje, brisanje, izmenu i listanje korisnika

Kao što vidimo na donjoj slici (slika 2), moguće je izvršiti funkcionalno testiranje za sva tekstualna polja i svaku funkciju aplikacije kao i testiranje integracije za funkcije koje su međusobno povezane. Funkcije koje je moguće testirati su: dodavanje korisnika (add user), listanje korisnika (list of users), brisanje korisnika (delete user), ažuriranje korisnika

The screenshot shows a web-based application for managing users. The main title is "Add Users". On the left, there is a sidebar with the following menu items: Add User, Delete User, List User, Edit User, Product Sales, Product Purchases, Search Users, and Help. The main content area contains several input fields: "User Name", "Password", and "Designation" (which has a dropdown menu showing "Team Lead", "Manager", and three other options). Below these are "Email", "Telephone", and "Address" fields. At the bottom right of the main area are two buttons: "Submit" and "Cancel".

Slika 6.2 Integraciono testiranje za međusobno povezane funkcije [Izvor: NM SE321-2020/2021.]

(edit user) i pretraživanje korisnika (search user).

ZADATAK: Definišite funkcije gde može da se vrši samo funkcionalno testiranje i funkcije gde je moguće obaviti funkcionalno i integraciono testiranje. Izvršite testiranje prateći sledeće smernice:

- Odredite prioritete kojima ćete vršiti testiranje
 - otvorite aplikaciju i odaberite koju funkciju treba prvo testirati.
 - idite na tu funkciju i odaberite komponentu koju morate prvo testirati.

- idite na tu komponentu i odredite vrednosti koje treba prvo uneti. Nemojte svuda primenjivati isto pravilo jer se logika testiranja razlikuje od funkcije do funkcije.
- Tokom izvođenja testiranja, trebalo bi da testirate jednu funkciju u potpunosti, a zatim da pređete na drugu funkciju.
- Između dve funkcije, sprovedite pozitivno ili pozitivno i negativno testiranje integracije.

Vreme trajanja 15 minuta.

3. PRIMER INTEGRACIONOG TESTIRANJA

Testiranje inbox modula

Sledeći primer na slici 3. prikazuje home stranicu Gmail-ovog Inbox-a gde kada kliknemo na Inbox, odlazimo na Inbox stranicu. Ovde nemamo potrebu za integracionim testiranjem, jer u ovom slučaju ne postoji ni roditelj (modul višeg nivoa) ni dete (modul nižeg nivoa za) za ovu stranicu.



Slika 6.3 GUI za Inbox modul [Izvor: NM SE321-2020/2021.]

Vreme trajanja 15 minuta.

✓ Poglavlje 7

Vežba za samostalni rad

ZADATAK ZA SAMOSTALNI RAD

Tekst zadatka za samostalni rad

Zadatak 1 :

Pronaći proizvoljni kod aplikacije rađene za potrebe projekta ili primer programskog koda sa interneta. Programski kod treba da sadrži minimum tri klase Ukoliko primer ne sadrži tri klase potrebno je doraditi programski kod tako da ispunjava navedeni uslov. Na navedenom programskom kodu izvršiti:

- jedinično testiranje
- mocking testiranje

Rezultat testiranja predstaviti kroz izveštaj u Word dokumentu sa objašnjenjem dobijenih rezultata. U okviru dokumenta navesti i osnovne razlike između navedena dva tipa testiranja. **(45 minuta)**

Zadatak 2:

Odabratи deo ISUM-a namenjen studentima i svaku od opcija ISUM-a - pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka - posmatrati kao poseban modul.

1. Definišite funkcije gde može da se vrši samo funkcionalno testiranje i funkcije gde je moguće obaviti funkcionalno i integraciono testiranje. **(15 minuta)**
2. Odredite prioritete kojima će se vršiti testiranje **(15 minuta)**
3. Izvršite najpre funkcionalna testiranja odabirom odgovarajućih komponenti (tokom izvođenja testiranja, trebalo bi da testirate jednu funkciju u potpunosti, a zatim da pređete na drugu funkciju). **(30 minuta)**
4. Između dve funkcije, sprovedite pozitivno ili pozitivno i negativno testiranje integracije. **(30 minuta)**

✓ Poglavlje 8

Domaći zadatak

OSMI DOMAĆI ZADATAK

Nakon sedme lekcije potrebno je uraditi sedmi domaći zadatak.

Za odabranu aplikaciju u okviru prethodnih domaćih zadataka primeniti sledeće:

1. Izvršiti mocking testiranje
2. Izvršiti integraciono testiranje

Za izradu domaćih zadataka preuzeti i koristiti šablon koji se nalazi nakon ovog uputstva u lekciji.

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnem okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

Domaći zadatak se imenuje: SE321-DZ08-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ08.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Zaključak

ŠTA SMO NAUČILI U OVOJ LEKCIJI?

Testiranje softvera može biti izvršeno delujući po određenim modelima.

U ovoj lekciji se govori o različitim nivoima testiranja softvera koji se vrši na nivou jedinica (engl. **Unit Testing**), integracije (engl. **Modul Integration Testing**) i sistema (engl. **System Testing**).

Testiranje modula može biti vršeno u razvojnog okruženju uz upotrebu pomoćnih modula drajvera i stabova. Drajver je modul koji zamenjuje deo sistema ka korisniku i obezbeđuje ulaze za testiranje, prikazuje ili registruje izlaze iz modula, i/ili ih upoređuje sa očekivanim vrednostima. Stab (**stub**) je pomoćni modul koji zamenjuje deo sistema ka periferiji (hardver ili baze podataka). U slučaju velikih nasleđenih poslovnih aplikacija, ne može se testirati celu aplikaciju već se primenjuje mocking testiranje u okviru kojeg jedinični testovi pokrivaju isključivo kod sa „poslovnom logikom“ kojem pristupaju drugi moduli, koji se često menja i kod kojeg se javlja puno grešaka.

Testiranje modula u sadejstvu sa drugim modulima (ostatkom sistema) se odvija prema trima strategijama:

- **Big-bang** integracija - kada se svi moduli koji sačinjavaju sistem testiraju ponaosob i integrišu odjednom.
- **Top-down** integracija - kod koje se uključivanje modula tokom integracije vrši odozgo na dole
- **Down-top** integracija - kod koje se uključivanje modula tokom integracije vrši odozdo na gore

LITERATURA ZA LEKCIJU 08

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. dr.sc. Tihana Galinac Grbac, Testiranje programskog proizvoda. (Link: <http://www.riteh.uniri.hr/>)
4. Dr Kelvin Ross, Practical Guide to Software System Testing, K. J. Ross & Associates Pty. Ltd., 1998.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. British Computer Society, Standard for Software Component Testing, Date: 27 April 2001.

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <https://www.softwaretestingclass.com/what-is-cause-and-effect-graph-testing-technique/>
4. <https://www.jrebel.com/blog/mock-and-unit-testing-with-testng-junit-and-mockito>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Organizacija i dokumentacija
testiranja softvera

Lekcija 09

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 09

ORGANIZACIJA I DOKUMENTACIJA TESTIRANJA SOFTVERA

- ✓ Organizacija i dokumentacija testiranja softvera
- ✓ Poglavlje 1: Organizacija testiranja
- ✓ Poglavlje 2: Životni ciklus testiranja softvera
- ✓ Poglavlje 3: Dokumentacija: scenarija i slučajevi testiranja
- ✓ Poglavlje 4: Dokumentacija: plan testiranja
- ✓ Poglavlje 5: Dokumentacija: Matrica sledljivosti
- ✓ Poglavlje 6: Nivoi zrelosti procesa testiranja (TMM)
- ✓ Poglavlje 7: Grupna vežba: dokumentacija testiranja
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Upravljanje procesom testiranja softvera

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

Postupak softverskog testiranja, poznat i kao Životni ciklus testiranja softvera (**Software Testing Life Cycle - STLC**) omogućava da se testiranje izvodi na dobro isplaniran i sistematičan način. Životni ciklus testiranja softvera sadrži šest koraka: analiza zahteva, izrada plana testiranja, podešavanje okruženja, izvršenje test slučajeva, evidentiranje nedostataka (defekata) i zatvaranje ciklusa testiranja koji su detaljnije opisani u predavanju sa aspekta ulaznih i izlaznih kriterijuma i aktivnosti koji se dešavaju u okviru svakog od njih.

Tokom ili pre testiranja softverske aplikacije kreiraju se različite vrste dokumentacije za testiranje. Na primer:

- **Scenario testiranja** je dokument koji definiše višestruke načine ili kombinacije testiranja aplikacije. Generalno, priprema se kako bi se razumeo tok aplikacije i se sastoji od slučajeva testiranja.
- **Slučaj testiranja** se može definisati kao grupa uslova pod kojima tester određuje da li softverska aplikacija radi prema zahtevima kupca ili ne. Dizajn slučajeva testiranja uključuje preduslove, naziv slučaja, uslove unosa i očekivani rezultat.
- **Plan testiranja** je detaljan dokument koji opisuje područja i aktivnosti testiranja softvera. U njemu se navode strategija testiranja, ciljevi, raspored testiranja, potrebni resursi (ljudski resursi, softver i hardver), estimacija testa i rezultati testiranja.
- **Matrica sledljivosti** (**Traceability Matrix**) je dokument tabelarnog tipa koji se koristi u razvoju softverskih aplikacija za praćenje zahteva. Može se koristiti za praćenje unapred (od zahteva do projektovanja ili kodiranja) i unazad (od kodiranja do zahteva).

Pored ovoga u lekciji se govori o modelima zrelosti procesa testiranja softvera (TMM), koji je objavio **Illinois Institute of Technology** iz Čikaga 1996 godine, analogno CMM modelu zrelosti procesa razvoja softvera (CMM) koji je razvio **Software Engineering Institute (SEI) of Carnegie Mellon University**. Ovde se daje kratak opis glavnih karakteristika svakog nivoa modela zrelosti procesa testiranja softvera (TMM).

▼ Poglavlje 1

Organizacija testiranja

ODNOS PREMA PROCESU TESTIRANJA

Kada se razvija sistem za potrebe kupaca, njih ne zanima da li sistem pravilno funkcioniše u nekim situacijama već da li sistem pravilno funkcionišati u svim situacijama

Programeri početnici nisu navikli da posmatraju testiranje kao proces otkrivanja grešaka u svim aktivnostima projektovanja softvera. Dok ste student, pišete programe prema specifikacijama koje dobijate od nastavnika. Po završenom dizajniranju programa, pišete kod i prevodite ga da biste utvrdili postoje li neke sintaktičke greške. Kada predajete program na ocenjivanje, obično nastavniku predajete listing programa, podatke koje ste koristili kao probni ulaz i eventualni izlaz u kojem se vidi kako je vaš program obradio ulazne podatke. Kod, ulaz i izlaz, zajedno, predstavljaju dokaz da vaš kod pravilno radi, a obično birate ulazne podatke tako da ubedite predavača da kod funkcioniše kao što je traženo u zadatku.

Svoj program ste verovatno posmatrali samo kao rešenje jednog problema: možda niste uzeli u obzir i sam problem. Ako je to slučaj, podatke za testiranje ste verovatno birali tako da potverdite rezultate u specijalnim slučajevima, a ne da dokažete odsustvo grešaka. Tako napisan program služi kao dokaz vaše programerske veštine. Zbog toga kritiku programa psihološki doživljavate kao kritiku vaših sposobnosti. Testiranje kao dokaz ispravnosti programa služi da biste nastavniku pokazali svoje znanje.

Međutim, kada razvijate sistem za potrebe kupaca, njih ne zanima da li sistem pravilno funkcioniše u nekim situacijama. Njih daleko više zanima mogu li da budu sigurni da će sistem pravilno funkcionišati u svim situacijama.

Zato bi vaš cilj kao projektanta ili programera trebalo da bude da otkrijete što više grešaka bez obzira na to gde se greške nalaze i bez obzira na to ko ih je napravio.

Kada se otkrije greška ili kada dođe do otkaza, depersonalizovani razvojni tim se bavi ispravljanjem greške, a ne prebacivanjem krivice na konkretnog izvršioca.

KO VRŠI TESTIRANJE?

Kao što ćemo videti, učesnici u razvoju su pre svega zaduženi za funkcionalni test i za test performanse, ali kupac ima veliku ulogu u završnom testu prihvatljivosti i instalacionim testom.

Čak i kada se sistem razvija bezličnim pristupom, ponekad je teško potisnuti lična osećanja iz procesa testiranja. Zato se za testiranje sistema često koristi nezavisan **tim za testiranje**. Na taj način izbegavamo konflikt između vlastitog osećanja odgovornosti za greške i potrebe da se pronađe što više grešaka.

Osim toga, postoje dodatni argumenti u prilog nezavisnog tima. Prvo, moguće je da nemamo greške prilikom tumačenja dizajna, određivanja programske logike, pisanja dokumentacije ili implementiranja algoritama.

Jasno je da ne bismo dali program na testiranje kad bismo mislili da on ne funkcioniše u skladu sa specifikacijama.

Ali možda nam je kod previše blizak da bismo bili objektivni i prepoznali suptilnije greške.

Osim toga, nezavisni tim za testiranje može da učestvuje u pregledu komponente tokom celokupnog razvoja.

Tim može da učestvuje u pregledima zahteva i dizajna, može pojedinačno da testira komponente koda a takođe i ceo sistem dok se on integriše i daje kupcima na prihvatanje.

Na taj način, testiranje se može obavljati istovremeno sa kodiranjem; **tim za testiranje može da testira i sklapa komponente čim se one završe, dok programeri za to vreme i dalje kodiraju preostale komponente.**

TIM TESTERA

Profesionalni testeri organizuju i izvršavaju testove. Oni su uključeni od početka, projektuju planove za testiranje i slučajeve za testiranje paralelno sa napretkom projekta.

Kao što ćemo videti, učesnici u razvoju su pre svega zaduženi za funkcionalni test i za test performanse, ali kupac ima veliku ulogu u završnom testu prihvatljivosti i instalacionim testovima. Međutim, za sva testiranja se timovi sastavljaju iz obe ekipe.

Često se programeri koji učestvuju u projektu ne uključuju u testiranje sistema; previše im je bliska struktura implementacije i njene namere da bi bili u stanju da uoče razliku između implementacije i zahtevanih funkcija ili performanse.

Zbog toga je tim za testiranje često odvojen od osoblja koje radi na implementaciji. Bilo bi idealno da neki članovi testerskog tima imaju iskustva u testiranju. Obično su ovi „profesionalci“ bivši analitičari, programeri i dizajneri koji sada posvećuju sve svoje vreme testiranju sistema. Oni poznaju specifikaciju sistema, ali takođe i metode i alate za testiranje.

Profesionalni testeri organizuju i izvršavaju testove.

Oni su uključeni od početka, projektuju planove za testiranje i slučajeve za testiranje paralelno sa napretkom projekta.

Oni sa timom za upravljanje konfiguracijom sarađuju na izradi dokumentacije i drugih mehanizama za povezivanje testova sa zahtevima, komponentama dizajna i kodom.

Profesionalci u obavljanju testiranja usredsređuju se na razvijanje testova, na metode i procedure. Pošto oni možda ne poznaju najbolje detalje iz zahteva, za razliku od onih koji su ih pisali, u testerski tim se uključuju ljudi kojima su zahtevi poznati. Analitičari koji su bili uključeni u prvočitno definisanje i specifikaciju zahteva korisni su prilikom testiranja zato što razumeju problemna način kako ga kupac vidi.

Veliki deo testiranja sistema sastoji se od poređenja novog sistema sa prvočitnim zahtevima, a analitičari imaju dobar osećaj za potrebe i ciljeve kupca.

Pošto su sarađivali sa projektantima na oblikovanju rešenja, analitičari imaju predstavu o tome kako sistem treba da radi u cilju rešavanja problema.

ULOGA PROJEKTANATA SISTEMA, SPECIJALISTA ZA UPRAVLJANJE KONFIGURACIJOM UI KORISNIKA PROCESU TESTIRANJA

Projektanti sistema pomažu u pripremi slučajeva testiranja, specijalisti za upravljanje konfiguracijom promene nastale zbog ispravki grešaka unose u dokumentaciju.....

Projektanti sistema dodaju timu za testiranje dimenziju namere. Projektanti shvataju šta smo predložili kao rešenje, kao i ograničenja tog rešenja. Oni takođe znaju kako je sistem podeljen na funkcionalne podsisteme ili podsisteme koji zavise od podataka i kako bi sistem trebalo da radi.

Kada tim za testiranje priprema slučajeve za testiranje i obezbeđuje pokrivenost testiranjem, on poziva projektante u cilju prikaza svih mogućnosti.

Pošto su testovi i slučajevi direktno povezani sa zahtevima i dizajnom, u tim se uključuje i predstavnik grupe za upravljanje konfiguracijom. Kada dođe do greške i treba uneti promene, specijalista za upravljanje konfiguracijom se brine da se promene unesu u dokumentaciju, zahteve, dizajn, kod i ostale proekte procesa razvoja. U stvari, promene potrebne da bi se ispravila greška možda zahtevaju promene i u drugim slučajevima za testiranje ili u većem delu plana testiranja.

Specijalista za upravljanje konfiguracijom implementira te promene i koordinira reviziju testiranja.

Konačno, u tim se uključuju korisnici. Oni su najkvalifikovaniji da procene pitanja koja se tiču prikladnosti, lakoće korišćenja i ostalih ljudskih faktora. Ponekad se korisnici malo pitaju u ranim fazama projekta.

Predstavnici kupca koji učestvuju u analizi zahteva možda ne nameravaju sami da koriste sistem, ali ih posao povezuje sa onima koji će ga koristiti. Na primer, predstavnici su možda rukovodioci budućih korisnika sistema ili tehnički predstavnici koji su otkrili problem posredno vezan za njihov posao.

Međutim, ovi predstavnici su možda toliko udaljeni od stvarnog problema da je opis zahteva neprecizan i nepotpun. Kupac možda nije svestan potrebe da se zahtevi dorade ili prošire.

Prema tome, **korisnici predloženog sistema su bitni, pogotovo ako nisu bili prisutni kada su na početku definisani zahtevi sistema.**

Korisnik je verovatno upoznat sa problemom jer svakodnevno nailazi na njega i može da pruži neprocenjivu pomoć u ocenjivanju sistema i proveri da li on rešava problem.

POGLEDI NA PREDMET TESTIRANJA - METODA CRNE KUTIJE

Za neke predmete testiranja, tim za testiranje ne može da napravi skup reprezentativnih slučajeva koji dokazuju pravilno funkcionisanje u svim situacijama

Kada testirate komponente, grupu komponenti, podsisteme ili sistem, vaš pogled na predmet testiranja (tj. komponentu, grupu, podsistemu ili sistem) može da utiče na način izvođenja testa. **Ako predmet testiranja posmatrate spolja kao zatvorenu kutiju, ili crnu kutiju sa nepoznatim sadržajem, vi prilikom testiranja ubacujete podatke u zatvorenu kutiju i beležite dobijene rezultate.** U tom slučaju cilj testiranja jeste da se potvrdi da su prosleđeni svi tipovi ulaznih podataka i pri tome dobijeni očekivani rezultati. Ova vrsta testiranja koja se u literaturi naziva **testiranje metodom crne kutije** ima svoje prednosti i mane. Očigledna prednost je to što je zatvorena kutija oslobođena brige o ograničenjima koja nameće unutrašnja struktura i logika predmeta testiranja.

Međutim, nije uvek moguće ceo test izvršiti na taj način. Na primer, prepostavite da jedna jednostavna komponenta prihvata kao ulaz tri broja a, b i c i kao rezultat daje dva rešenja jednačine

$$ax^2 + bx + c = 0$$

ili poruku „nema realnih rešenja“. Komponentu nije moguće testirati tako da joj se podnesu sve moguće uređene trojke (a, b, c).

U tom slučaju će tim za testiranje možda izabrati reprezentativne podatke za testiranje da bi pokazao da se sve moguće kombinacije pravilno obrađuju.

Na primer, mogu se izabrati podaci tako da imamo sve kombinacije pozitivnih, negativnih vrednosti i nule za a, b i c. Ukupno dvadeset sedam mogućnosti. Ako znamo nešto o rešavanju kvadratnih jednačina, možda ćemo birati vrednosti takve da diskriminanta $b^2 - 4ac$ bude u svakoj od tri klase: pozitivna, negativna ili jednak nuli. (U tom slučaju, mi u stvari nagađamo kako je komponenta implementirana.) Međutim, čak i kada test za sve razmatrane klase ne otkrije nijednu grešku, nemamo nikakvu garanciju da komponenta stvarno ne sadrži greške. Komponenta još uvek može u nekom konkretnom slučaju da da pogrešan rezultat, zbog finesa kao što su greške u zaokrugljivanju ili neusklađeni tipovi podataka.

Za neke predmete testiranja, tim za testiranje ne može da napravi skup reprezentativnih slučajeva koji dokazuju pravilno funkcionisanje u svim situacijama

POGLEDI NA PREDMET TESTIRANJA - METODA BELE KUTIJE

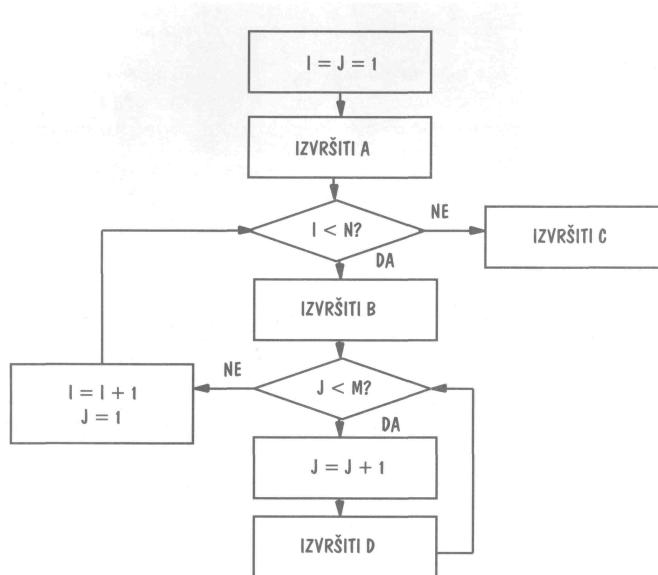
Možemo posmatrati predmet testiranja kao otvorenu kutiju (koja se ponekad naziva i belom kutijom); tada koristimo unutrašnju strukturu predmeta testiranja u cilju sprovođenja testova.

Za prevazilaženje ovog problema možemo posmatrati predmet testiranja kao otvorenu kutiju (koja se ponekad naziva i belom kutijom); tada možemo koristiti unutrašnju strukturu predmeta testiranja u cilju sprovođenja različitih testova.

Na primer, možemo da smislimo testove tako da se izvrše sve naredbe ili svi tokovi kontrole unutar komponente, kako bismo se uverili da predmet testiranja ispravno radi. Međutim, ta vrsta pristupa, koja se u literaturi naziva testiranje po metodi bele kutije, može da bude nepraktična.

Na primer, komponenta sa mnogo grananja i petlji sadrži veliki broj putanji koje treba proveriti. Čak i sa relativno jednostavnom logičkom strukturom, teško je temeljito testirati komponentu sa značajnim brojem iteracija ili rekurzija. Prepostavite da je logika komponente tako postavljena da se petlja izvršava nm puta, kao što se vidi na slici 1.

Ako su i n i m svaki jednaki 100.000, primer koji se testira bi trebalo da se izvrši deset milijardi puta da bi prošao sve logičke putanje. Mogli bismo usvojiti strategiju testiranja u kojoj se petlja izvršava samo nekoliko puta i pri tome proveriti manji broj relevantnih slučajeva, koji će biti reprezentativni za ceo skup mogućnosti.



Slika 1.1 Primer logičke strukture [Izvor: NM SE321-2020/2021.]

ODLUKE O NAČINU TESTIRANJA

Testiranje po modelu zatvorene kutije možemo posmatrati kao jednu, a testiranje po modelu otvorene kutije kao drugu krajnost sveukupnog prostora testiranja.

U ovom primeru možemo izabrati jednu vrednost za I koja je manja od n, drugu vrednost koja je jednak n i još jednu koja je veća od n; slično tome, možemo da ispitamo J koje je manje od m, jednak m i veće od m i njihove kombinacije sa kombinacijama vrednosti I. **U opštem slučaju strategija može da se zasniva na podacima, strukturi, funkciji ili nekim drugim kriterijumima**, kao što ćemo kasnije videti.

Prilikom donošenja odluke o načinu testiranja, ne moramo da biramo pristup otvorene ili zatvorene kutije. Testiranje po modelu zatvorene kutije možemo posmatrati kao jednu, a testiranje po modelu otvorene kutije kao drugu krajnost sveukupnog prostora testiranja. Sve filozofije testiranja padaju negde između te dve krajnosti.

Uglavnom, izbor filozofije testiranja zavisi od više faktora, uključujući i:

- broj mogućih logičkih putanja;
- prirodu ulaznih podataka;
- količinu potrebnog izračunavanja;
- složenost algoritama.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

Životni ciklus testiranja softvera

OD ČEGA SE SASTOJI ŽIVOTNI CIKLUS TESTIRANJA SOFTVERA?

Sadrži sledeće korake: analiza zahteva, izrada plana testiranja, podešavanje okruženja, izvršenje test slučajeva, evidentiranje nedostataka (defekata), zatvaranje ciklusa testiranja

Postupak softverskog testiranja, poznat i kao Životni ciklus testiranja softvera (Software Testing Life Cycle - STLC) uključuje šest faza procesa testiranja. Proces testiranja se izvodi na dobro isplaniran i sistematičan način. Sve aktivnosti se vrše radi poboljšanja kvaliteta softverskog proizvoda.

Životni ciklus testiranja softvera sadrži sledeće korake:

1. Analiza zahteva
2. Izrada plana testiranja
3. Podešavanje okruženja
4. Izvršenje test slučajeva
5. Evidentiranje nedostataka (defekata)
6. Zatvaranje ciklusa testiranja

ANALIZA ZAHTEVA I IZRADA PLANA TESTIRANJA

Tokom analize zahteva tester analizira dokument zahteva SDLC-a kako bi ispitao zahteve koje je naveo klijent. Tokom izrade plana testiranja definisane sve strategije testiranja.

Analiza zahteva: Prvi korak postupka ručnog testiranja je analiza zahteva. U ovoj fazi tester analizira dokument zahteva SDLC (Životni ciklus razvoja softvera) kako bi ispitao zahteve koje je naveo klijent. Nakon ispitivanja zahteva, tester pravi plan testiranja kako bi proverio da li softver ispunjava zahteve ili ne.

Ulagani kriterijumi: Za planiranje specifikacije zahteva za plan testiranja treba da budu na raspolaganju dokument o arhitekturi aplikacije i dobro definisani kriterijumi prihvatanja.

Aktivnosti:

1. Priprema liste svih zahteva i pronalaženje rešenja pripremljenih od strane tehničkog menadžera / potencijalnih klijenata, arhitekata sistema, poslovnih analitičara i klijenta.
2. Pravljenje liste svih vrsta testova (performansi, funkcionalnosti i sigurnost) koje treba izvršiti.
3. Pravljenje liste o detaljima test okruženja, koja treba da sadrži sve neophodne alate za izvršavanje test slučajeva.

Izlazi: Spisak svih potrebnih testova za testiranje zahteva i detalji okruženja za testiranje

Izrada plana testiranja: Izrada plana testiranja je presudna faza STLC-a gde su definisane sve strategije testiranja. Tester određuje procenjeni napor i troškove celog projekta. Ova faza se odvija nakon uspešnog završetka faze analize zahteva. U ovoj fazi se definiše dokument kojim se utvrđuje strategija testiranja i procena potrebnih napora. Nakon uspešnog završetka izrade plana testiranja može se započeti sa izvršenjem test slučaja.

Ulazni kriterijumi: Dokument zahteva

Aktivnosti:

1. Definisanje cilja i opseg softvera.
2. Navođenje metoda uključenih u testiranje.
3. Utvrđivanje pogleda na proces testiranja.
4. Postavljanje testnog okruženje
5. Priprema rasporeda testiranja i kontrolnih procedura.
6. Određivanje uloga i odgovornosti.
7. Navođenje rezultata testiranja, definisanje rizik ako postoji.

Izlaz:

Dokument o strategiji testiranja.

Dokumenti sa procenjenim naporima za potrebe testiranja.

PODEŠAVANJE OKRUŽENJA I IZVRŠENJE TEST SLUČAJEVA

Postavljanje testnog okruženja je važan deo postupka ručnog testiranja dok izvršenje test slučajeva podrazumeva pisanje detaljnih slučajeva testiranja i pripremu i podatka za testiranje

Podešavanje okruženja: Postavljanje testnog okruženja je nezavisna aktivnost i može se započeti zajedno sa razvojem test slučajeva. Ovo je važan deo postupka ručnog testiranja, jer ono nije moguće bez testiranja okruženja. Podešavanje okruženja zahteva podešavanje osnovnog softvera i hardvera za kreiranje testnog okruženja. Tim za testiranje nije uključen u postavljanje okruženja za testiranje, već programeri koji ga kreiraju.

Ulazni kriterijumi:

1. Strategija testiranja i dokument plana testiranja.
2. Dokument test slučaja.

3. Podaci za testiranje.

Aktivnosti:

1. Priprema liste potrebnog softvera i hardvera na osnovu analize specifikacije zahteva.
2. Nakon podešavanja testnog okruženja, izvršenje probnih slučajeva testiranja kako bi se proverila spremnost testnog okruženja.

Izlazi: Izveštaj o izvršenju. Izveštaj o nedostacima.

Izvršenje test slučaja: Odvija se nakon uspešnog završetka planiranja testiranja. U ovoj fazi, tim za testiranje započinje aktivnosti razvoja i izvršenja testova. Tim za testiranje piše detaljne slučajeve testiranja, po potrebi priprema i podatke za testiranje. Pripremljene test slučajeve pregledavaju članovi tima za testiranje ili menadžer osiguranja kvaliteta.

U ovoj fazi se takođe priprema Matrica za praćenje zahteva (**Requirement Traceability Matrix - RTM**) . Matrica za praćenje zahteva je format industrijskog nivoa koji se koristi za praćenje zahteva. U njoj se svaki test slučaj mapira sa određenom specifikacijom zahteva. Putem RTM-a mogu se pratiti zahtevi unazad i unapred.

Ulazni kriterijumi: Dokument zahteva

Aktivnosti:

1. Izrada test slučajeva.
2. Izvršenje test slučajeva.
3. Mapiranje test slučajeva prema zahtevima.

Izlaz: Rezultat izvršenja testa.

Spisak funkcija sa detaljnim objašnjenjem nedostataka.

EVIDENTIRANJE DEFEKATA I ZATVARANJE CIKLUSA TESTIRANJA

U fazi evidentiranja defekata se određuju karakteristike i nedostaci softvera. Tokom zatvaranja ciklusa testiranja se procenjuje strategiju razvoja, postupak testiranja, mogući nedostaci itd.

Evidentiranje defekata: Testeri i programeri ocenjuju kriterijume koje ispunjava softver na osnovu pokrivenosti testovima, kvaliteta, utroška vremena, troškova i kritičnih poslovnih ciljeva. U ovaj fazi se određuju karakteristike i nedostaci softvera. Kako bi se otkrile vrste defekta i njihova težina detaljno se analiziraju slučajevi testiranja i izveštaji o greškama.

Analiza evidentiranih defekata se uglavnom radi zbog otkrivanja distribucije defekata u zavisnosti od težine i vrste. Ako se otkrije bilo koja neispravnost, onda se softver vraća razvojnom timu radi otklanjanja defekta, a zatim se softver ponovo testira sa svih aspekata testiranja.

Kada se ciklus testiranja u završi, pripremaju se završni izveštaji i metrike testa.

Ulazni kriterijumi:

Izveštaj o izvršenju test slučaja. Izveštaj o nedostacima (defektima)

Aktivnosti:

1. Procena kriterijuma o dovršenosti softvera na osnovu pokrivenosti testom, kvaliteta, utroška vremena, troškova i kritičnih poslovnih ciljeva.
2. Analiza evidentiranih defekata kojom se otkriva distribuciju defekata prema vrstama i težini.

Izlaz: Završni izveštaj. Metrike testa

Zatvaranje ciklusa testiranja: Izveštaj o zatvaranju ciklusa testiranja uključuje svu dokumentaciju koja se odnosi na dizajn softvera, razvoj, rezultate testiranja i izveštaje o defektima.

U ovoj fazi se procenjuje strategiju razvoja, postupak testiranja, mogući nedostaci kako bi se sva ta iskustva koristila u budućnosti ako postoji softver sa istim specifikacijama.

Ulazni kriterijumi:

Svi dokumenti i izveštaji koji se odnose na softver.

Aktivnosti:

1. Procena strategije razvoja, postupka ispitivanja, mogućih nedostatke radi korišćenje ovih iskustava u budućnosti ako postoji softver sa istim specifikacijama

Izlaz:

Izveštaj o zatvaranju testa

▼ Poglavlje 3

Dokumentacija: scenarija i slučajevi testiranja

ŠTA SAČINJAVA DOKUMENTACIJU ZA TESTIRANJE?

Artefakti koji su kreirani tokom ili pre testiranja softverske aplikacije.

Dokumentacija za testiranje je dokumentacija o artefaktima koji su kreirani tokom ili pre testiranja softverske aplikacije. Dokumentacija odražava važnost procesa za kupca, pojedinca i organizaciju.

Projekti koji sadrže svu potrebnu dokumentaciju imaju visok nivo zrelosti. Pažljivo kreirana dokumentacija može uštedeti vreme, napore i povećati vrednost organizacije.

Pre nego što se započne postupak izvršenja testa, inženjeri za testiranje pišu neophodan referentni dokument za testiranje. Generalno, dokument za testiranje pišemo kad god su programeri zauzeti pisanjem koda. Jednom kada je dokument za testiranje spreman, ceo postupak izvršenja testa zavisi od dokumenta za testiranje. Primarni cilj pisanja dokumenta za testiranje je smanjenje ili otklanjanje sumnji u vezi sa aktivnostima testiranja.

Tipovi test dokumenata: U testiranju softvera postoje različiti tipovi test dokumenata kao što su:

1. Scenarija testiranja (Test scenarios)
2. Slučajevi testiranja (Test case)
3. Plan testiranja (Test plan)
4. Matrica za praćenje zahteva (Requirement traceability matrix - RTM)
5. Strategija testiranja (Test strategy)
6. Podaci za testiranje (Test data)
7. Izveštaj o greškama (Bug report)
8. Izveštaj o izvršenju testa (Test execution report)

SCENARIJA TESTIRANJA

Scenario testiranja je dokument koji definiše višestruke načine ili kombinacije testiranja aplikacije. Generalno, priprema se kako bi se razumeo tok aplikacije.

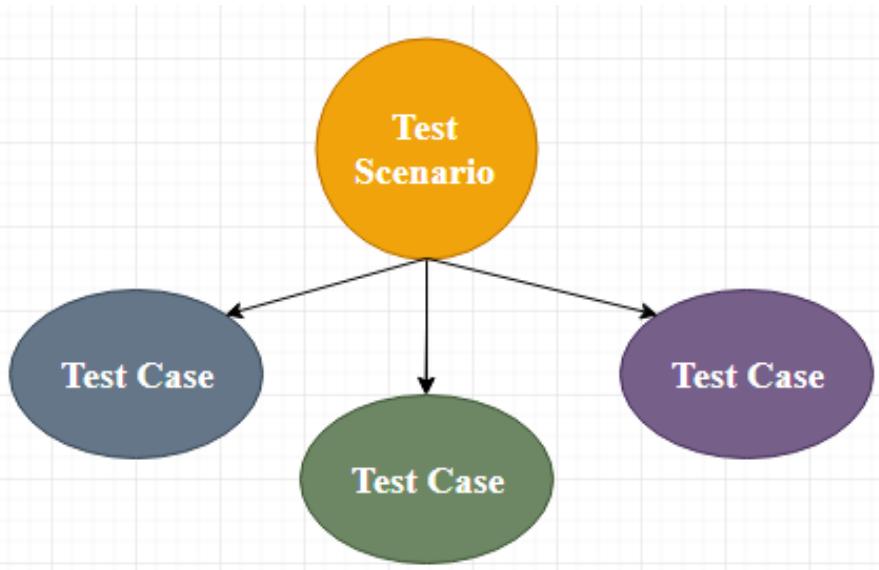
Scenario testiranja je detaljan dokument slučajeva testiranja koji pokrivaju celokupnu funkcionalnost softvera u obliku sekvence linearnih iskaza (operacija). Svaki iskaz sesmatra scenarijem. Scenarijo testiranja je klasifikacija zahteva testiranja na visokom nivou. Ovi

zahtevi su grupisani na osnovu funkcionalnosti modula i dobijeni su iz slučajeva korišćenja (**use cases**).

U scenariju testiranja je opisan detaljan postupak testiranja zbog postojanja velikog broja povezanih slučajeva testiranja. Pre izvođenja scenarija testiranja, tester mora da razmotri slučajeve testiranja za svaki scenario.

U scenariju testiranja, testeri moraju da se postave na mesto korisnika jer testiraju softversku aplikaciju s tačke gledišta korisnika. Priprema scenarija je najkritičniji deo postupka testiranja.

Za pripremu scenarija testiranja potrebno je potražiti savet ili pomoć od kupaca, zainteresovanih strana (**stakeholder**) ili programera.



Slika 3.1 Scenario testiranja se sastoji od slučajeva testiranja Izvor: <https://www.javatpoint.com/test-scenario> [Izvor: NM SE321-2020/2021.]

KAKO NAPISATI SCENARIJE TESTIRANJA?

Za kreiranje scenarija testiranja treba slediti tačno definisane korake

Za kreiranje scenarija testiranja treba slediti sledeće korake:

1. **Pročitajte dokument zahteva**, kao što su BRS (Business Requirement Specification) - specifikacija poslovnih zahteva), SRS (System Requirement Specification) specifikacija sistema zahteva i FRS Functional Requirement Specification) specifikacija funkcionalnih zahteva softvera koji se testira.
2. **Odredite sve tehničke aspekte i ciljeve** za svaki zahtev.
3. **Pronadite sve moguće načine** na koje korisnik može upravljati softverom.
4. **Utvrđite sva moguća scenarija** zbog kojih se sistem može zloupotrebiti i takođe otkrijte korisnike koji mogu biti hakeri.
5. Nakon čitanja dokumenta zahteva i završetka potrebne analize, **napravite listu različitih scenarija testiranja** da biste verifikovali svaku funkciju softvera.
6. Jednom kada ste naveli sve moguće scenarije testiranja, **kreirajte matricu sledivosti** kako biste saznali da li svaki zahtev ima odgovarajući test scenario ili ne.
7. **Supervizor projekta pregledava sve scenarije. Kasnije ih procenjuju druge zainteresovane strane u projektu.**

Karakteristike scenarija testiranja:

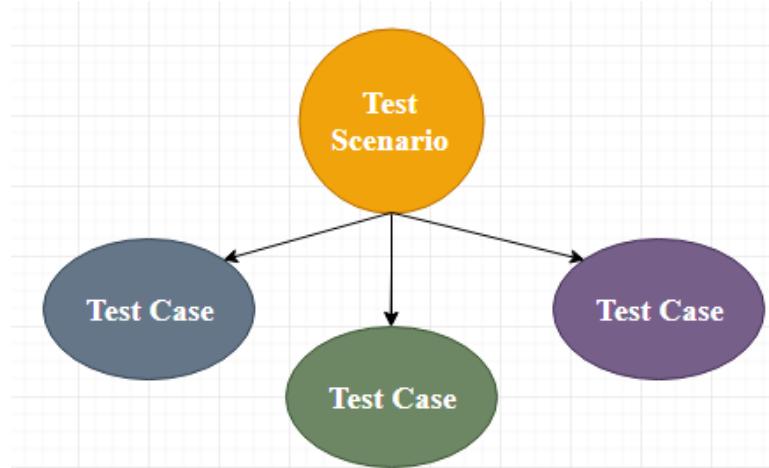
1. Scenario testiranja se piše kao linijska sekvenca operacija koje treba izvršiti a koje ukazuju testerima kakvu sekvencu testiranja treba da sprovedu.
2. Scenario testiranja smanjuje složenost i potrebu za ponavljanjem testiranja.
3. Scenario testiranja omogućava detaljnu diskusiju i razmišljanje o testovima napisanih kao linearne sekvene operacija - izjava.
4. Scenariji testiranja postaju važni kada tester nema dovoljno vremena da napiše slučajeve testiranja, a članovi tima se slože sa detaljnim linearnim scenarijem.
5. Scenariji testiranja su aktivnost koja štedi vreme.
6. Omogućava lako održavanje, jer su dodavanje i modifikacija scenarija testiranja laki i nezavisni.

SLUČAJEVI TESTIRANJA

Slučaj testiranja se može definisati kao grupa uslova pod kojima tester određuje da li softverska aplikacija radi prema zahtevima kupca ili ne.

Slučaj testiranja se može definisati kao grupa uslova pod kojima tester određuje da li softverska aplikacija radi prema zahtevima kupca ili ne. Dizajn slučajeva testiranja uključuje preduslove, naziv slučaja, uslove unosa i očekivani rezultat. Slučaj testiranja je akcija prvog nivoa i izведен je iz scenarija testiranja (slika 2.)

To je detaljni dokument koji sadrži sve moguće ulaze (pozitivne kao i negativne) i korake za navigaciju koji se koriste u procesu izvršavanja testa. Pisanje slučajeva testiranja se radi jedanput a može se koristiti i u budućnosti za vreme regresionog testiranja.



Slika 3.2 Slučaj testiranja je izведен iz scenarija testiranja [Izvor: NM SE321-2020/2021.]

Slučajevi testiranja daju detaljne informacije o strategiji testiranja, procesu testiranja, preduslovima i očekivanim rezultatima. Oni se izvršavaju tokom procesa testiranja da bi se proverilo da li softverska aplikacija izvršava zadatak za koji je razvijena ili ne.

Slučaj testiranja pomaže testeru da izvesti o defektu povezivanjem defekta sa ID-em slučaja testiranja. Detaljna dokumentacija o slučajevima testiranja deluje kao potpuni zaštitnik tima za testiranje, jer ako je programer nešto propustio, to može biti otkriveno tokom izvršavanja slučajeva testiranja.

Da bismo napisali slučaj testiranja, moramo imati zahteve na osnovu kojih se mogu definisati ulazi, a scenariji testiranja moraju biti napisani tako da se pokriju sve funkcije za testiranje. Kako bismo održali jednoobraznost (uniformnost) i obezbedili da svaki inženjer testiranja sledi isti pristup u pripremi dokumentacije za testiranje, prilikom izrade slučajeva testiranja bi trebalo koristiti obrazac (template).

Generalno, slučaj testiranja pišemo kad god je programer zauzet pisanjem koda.

KADA I ZAŠTO PIŠEMO SLUČAJEVE TESTIRANJA?

Razlozi su mnogobrojni

Slučajeve testiranja pišemo:

1. Kada kupac iskaže poslovne potrebe, programer započinje razvoj i kaže da mu npr. treba 3,5 meseca da napravi softverski proizvod koji će ih zadovoljiti.
2. U međuvremenu, tim za testiranje započinje sa pisanjem slučajeva testiranja.
3. Kada se slučajevi testiranja završe, šalju se Test lideru na pregledavanje.
4. Kada programeri završe sa razvojem softverskog proizvoda, predaju ga timu za testiranje
5. Inženjeri za testiranje tokom testiranja softverskog proizvoda ne proveravaju zahteve već slučajeve testiranja tako da testiranje ne zavisi od raspoloženja već od kvaliteta inženjera za testiranje.

Slučajeve testiranja pišemo iz sledećih razloga:

- 1. Da bi se postigla doslednost (konzistentnost) u izvršenju test slučajeva: testiranje aplikacije treba vršiti na osnovu onoga što se vidi u slučajevima testiranja.
- 2. Da bi se osigurala bolja pokrivenost testom: da bi se ovo postiglo, potrebno je da uradimo sve moguće scenarije testiranja i dokumentujemo ih kako se scenarija testiranja ne bi morali da prisećamo iznova i iznova
- 3. Da testiranje ne bi zavisilo od pojedinca već predstavljalo organizovan proces: prepostavimo da je inženjer testa testirao prvi riliz aplikacije, zatim drugi riliz aplikacije ali da je u vreme trećeg riliza napustio kompaniju. Tokom testiranja inženjer testa je uspeo da razume aplikaciju i izvede mnoge zaključke. Ako je tokom trećeg riliza test inženjer napustio kompaniju, novoj osobi koja bi se uključila u proces testiranja bi bilo vrlo teško. Stoga je potrebno dokumentovati sve izvedene zaključke, kako bi se oni mogli koristiti u budućnosti.
- 4. Da bi se izbegla obuka za svakog novog inženjera za testiranje: Kada inženjer testa ode, on / ona odlazi sa puno znanja o scenarijima testiranja. Te scenarije treba dokumentovati tako da novi inženjer testa može da obavi testiranje sa postojećim scenarijima ali i da takođe napiše nove scenarije.

OBRAZAC ZA PISANJE SLUČAJEVA TESTIRANJA

Navodimo standardna polja koja treba uzeti u obzir prilikom pripreme obrasca za slučaj testiranja .

Test Scenario ID		Test Case ID					
Test Case Description	I	Test Priority					
Pre-Requisite		Post-Requisite					
Test Execution Steps:							
S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments

Slika 3.3 Primer obrasca (template) za pisanje slučajeva testiranja Preuzeto sa:
<https://www.softwaretestinghelp.com/test-case-template-examples/> [Izvor: NM SE321-2020/2021.]

Sa site-a <https://www.softwaretestinghelp.com/test-case-template-examples/> se može preuzeti word verzija ovog template-a koji je takođe dat i kao prilog ovog predavanja

Izgled i sadržaj obrazaca (**template**) za slučajeve testiranja se razlikuju u zavisnosti od funkcionalnosti softvera kome su namenjeni. Međutim, u nastavku je dat obrazac koji se uvek može koristiti za dokumentovanje slučajeva testiranja, bez obzira šta aplikacija radi. U nastavku je navedeno nekoliko standardnih polja treba koje treba uzeti u obzir prilikom pripreme obrasca za slučaj testiranja:

- Test Scenario ID: ID scenarija kojem pripada slučaj testiranja
- Test case ID: ID slučaja testiranja, postoje određene konvencije u dodeljivanju ID-a npr. 'TC_UI_1' označava slučaj testiranja 1 za korisnički interfejs.
- Prioritet testa (nizak (Low)/srednji (Medium)/visok (High)): Prioritet testa za poslovna pravila i funkcionalne slučajeve testiranja može biti srednji ili visok dok za korisnički interfejs može biti nizak.
- Test Case Description: Ukratko opisuje ciljeve testa
- Pre-Requisite: Bilo koji preduslov koji mora biti ispunjen pre izvršenja testa.
- Post-Requisite: Kakvo treba da bude stanje sistema posle izvršenja slučaja testiranja?
- Test Execution Steps: Detaljna lista svih koraka koje treba izvršiti. Treba je napisati u redosledu u kojem treba izvršavati korake. Za svaki korak treba opisati sledeće detalje:
 - S. No: redni broj koraka
 - Action: akciju koju treba izvršiti
 - Input: podaci za testiranje koji se koriste kao ulaz u slučaj testiranja (različiti skupovi podataka sa tačnim vrednostima).
 - Expected Output: Šta treba da bude očekivani izlaz iz sistema posle izvršenja testa?
 - Actual Output: Šta je stvarni izlaz iz sistema posle izvršenja testa?
 - Test Result (prošao/nije prošao): ukoliko je stvarani rezultat isti kao očekivani, test treba označiti kao "prošao" u suprotnom "nije prošao"
 - Test comment: komentar testa

PROCES PISANJA SLUČAJEVA TESTIRANJA

Način pisanja slučajeva testiranja može se opisati kroz korake, koji su navedeni u nastavku

Koraci u pisanju slučajeva testiranja:

1. **Studija sistema:** Cilj ovog koraka je razumeti aplikaciju gledajući zahteve ili SRS, koji daje kupac.
2. **Identifikacija svih scenarija:**
 - Kada je softverski proizvod lansiran, treba pronaći sve moguće načine na koje ga krajnji korisnik može koristiti
 - Sve moguće scenarije treba dokumentovati u dokumentu koji se naziva dizajn testa.
 - Dizajn testa je zapis koji sadrži sve moguće scenarije.
3. **Pisanje testova**
 - Konvertujte sve identifikovane scenarije u testove tako što ćete scenarije grupisati prema njihovim karakteristikama a pri i pisanju testova koristiti standardne obrasce (template) za pisanje test slučajeva, koje ste odlučili da koristite.
4. **Pregledavanje testova**
 - Slučajeve testiranja pregledava šef tima a nakon povratnih informacija dobijenih od njega treba izvršiti popravke.
5. **Odobrenje slučajeva testiranja**
 - Nakon popravke slučajeva testiranja, treba ih poslati ponovo na odobrenje.
6. **Čuvanje u repozitorijumu slučajeva testiranja**
 - Nakon odobrenja određenih slučajeva testiranja, treba ih sačuvati u repozitorijumu test slučajeva.

▼ Poglavlje 4

Dokumentacija: plan testiranja

PLAN TESTIRANJA

Plan testiranja je detaljan dokument koji opisuje područja i aktivnosti testiranja softvera. U njemu se navode strategija testiranja, ciljevi, raspored testiranja, potrebni resursi itd.

Plan testiranja je detaljan dokument koji opisuje područja i aktivnosti testiranja softvera. U njemu se navode strategija testiranja, ciljevi, raspored testiranja, potrebni resursi (ljudski resursi, softver i hardver), estimacija testa i rezultati testiranja.

Plan testiranja je osnova za testiranje svakog softvera. To je najvažnija aktivnost koja osigurava dostupnost svim listama planiranih aktivnosti u odgovarajućem redosledu.

Plan testiranja je obrazac za sprovođenje aktivnosti testiranja softvera kao definisanog procesa koji u potpunosti nadgleda i kontroliše menadžer testiranja. Plan testiranja pripremaju test lider (60%), rukovodilac testiranja (20%) i inženjer testa (20%).

Postoje tri vrste plana testiranja:

- **Master (glavni) plan testiranja:** vrsta plana testiranja koji ima više nivoa testiranja. Sadrži kompletну strategiju testiranja.
- **Fazni plan testiranja:** vrsta plana testiranja koji se bavi bilo kojom fazom strategije testiranja.
- **Specifični planovi testiranja:** dizajniran za nefunkcionalno testiranje kao što su testiranje bezbednosti, testiranje opterećenja, testiranje performansi itd.

Kako napisati plan testiranja?

Izrada plana testiranja je najvažniji zadatak procesa upravljanja testovima. Prema IEEE 829, treba slediti sledećih sedam koraka za pripremu plana.

- Prvo analizirajte strukturu i arhitekturu softverskog proizvoda.
- Dizajnirajte strategiju testiranja.
- Definišite sve ciljeve testiranja.
- Definišite područje testiranja.
- Definišite sve korisne resurse.
- Rasporedite sve aktivnosti na odgovarajući način.
- Odredite sve izlaze (**Deliverables**) testiranja.

KOMPONENTE PLANA TESTIRANJA: CILJEVI I OPSEG

Plan testiranja se sastoji od različitih delova koji nam pomažu da definišemo sve aktivnosti testiranja.

Plan testiranja se sastoji od različitih delova koji nam pomažu da definišemo sve aktivnosti testiranja.

Ciljevi: Sastoje se od informacija o modulima, karakteristikama, podacima o testiranju itd., koji ukazuju na cilj aplikacije.

Opseg: Sadrži informacije o tome šta treba testirati u okviru aplikacije. Opseg se može podeliti na dva dela:

- u opsegu: moduli koje treba detaljno testirati.
- van opsega: moduli koji ne moraju biti detaljno testirani.

*Na primer: pretpostavimo da je potrebno testirati Gmail aplikaciju, i u okviru nje funkcije: Sastavljanje pošte (**Compose mail**), Poslata pošta (**Sent Items**), Primljena pošta (**Inbox**), **Draft** a funkcije koje se ne testiraju su Pomoć (**Help**), i ostale. To znači da u fazi planiranja, odlučujemo koja funkcionalnost softvera mora biti testirana a koja ne na osnovu vremenskog ograničenja datog proizvoda.*

Kako odlučujemo koje karakteristike nećemo testirati?

Na osnovu sledećih aspekata:

Primer 1: Pretpostavimo da imamo jednu aplikaciju koja ima P, K, R i S funkcije, koje treba razviti na osnovu zahteva. Ali funkciju S je dizajnirala i koristi neka druga kompanija. Dakle, razvojni tim će kupiti S od te kompanije i integrisati sa dodatnim funkcijama kao što su P, K i R.

Nećemo vršiti funkcionalno testiranje funkcije S jer je ona već korišćena u realnom vremenu. Ali ćemo izvršiti integraciono i sistemsko testiranje između P, K, R i S, jer nove funkcije možda neće raditi ispravno sa funkcijom S.

Primer 2: Pretpostavimo da su u prvom rilizu softverskog proizvoda razvijeni elementi, kao što su P, K, R, S, T, U, V,Ks, I, Z. U drugom rilizu, klijent će dati zahteve za nove funkcije koje poboljšavaju proizvod i tako će biti specificirane nove funkcije kao što su A1, B2, C3, D4 i E5.

U planu testiranja obim ćemo definisati kao:

Karakteristike koje treba testirati: A1, B2, C3, D4, E5 (nove funkcije)

P, K, R, S, T.

Karakteristike koje se ne testiraju: V... .Ks, I, Z

Stoga ćemo prvo proveriti nove karakteristike, a zatim nastaviti sa starim karakteristikama (obavićemo jedan krug regresivnog testiranja za P, K, R..., T karakteristike).

KOMPONENTE PLANA TESTIRANJA: METODOLOGIJA I PRISTUP TESTIRANJA

Metodologija testiranja sadrži informacije o izvođenju različitih vrsta testiranja dok se pristup testiranja koristi za opisivanje toka kroz aplikaciju tokom izvođenja testiranja.

Metodologija testiranja: Sadrži informacije o izvođenju različitih vrsta testiranja kao što su funkcionalno testiranje, integraciono testiranje, sistemsko testiranje itd. na aplikaciji. Ovde treba odlučiti koje vrste testiranja treba sprovesti nad različitim funkcijama sistema definisanim u okviru zahteva. Takođe treba definisati kakvu vrstu testiranja ćemo koristiti u metodologijama testiranja, tako da svi, poput menadžmenta, razvojnog tima i tima za testiranje, mogu lako da ih razumeju.

Na primer, za samostalnu aplikaciju kao što je Adobe Photoshop, možemo izvršiti sledeće vrste testiranja:

Funkcionalno testiranje → Integraciono testiranje → Testiranje sistema → Adhoc testiranje → Testiranje kompatibilnosti → Regresiono testiranje → Testiranje pristupačnosti → Testiranje upotrebljivosti → Testiranje pouzdanosti → Testiranje oporavka → Testiranje instalacije ili deinstalacije

Pristup testiranja: Ovaj atribut se koristi za opisivanje toka kroz aplikaciju tokom izvođenja testiranja. Tok kroz aplikaciju možemo razumeti uz pomoć sledećih aspekata:

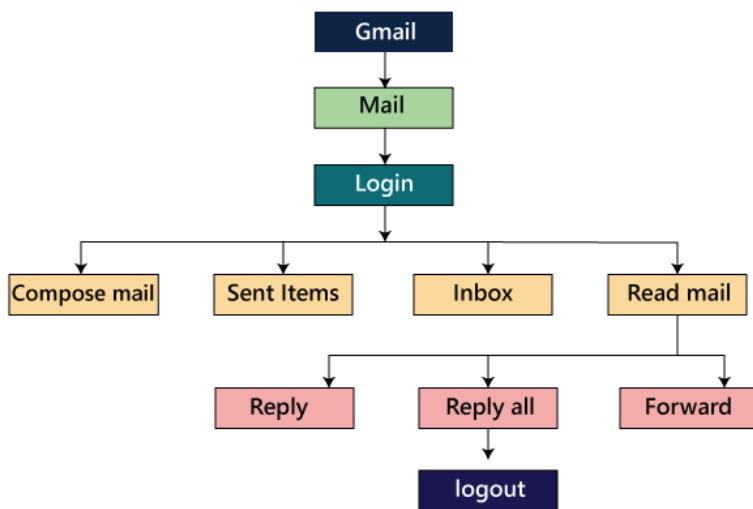
- Pisanjem scenarija na visokom nivou
- Pisanjem grafa toka

Primer: Pisanje scenarija na visokom nivou za testiranje Gmail app:

- Prijavite se na Gmail - pošaljite e-mail i proverite da li se nalazi na stranici Poslatih mail-ova Sent Items
- Prijavite se na

Scenarije visokog nivoa pišemo da bismo opisali pristupe koji se moraju preuzeti za testiranje sv proizvoda i to samo njegovih kritičnih funkcija. Ovde se nećemo fokusirati na pokrivanje svih scenarija, jer o tome koje funkcije treba testirati ili ne odlučuju inženjeri testa.

Pisanjem grafa toka: Grafove toka (slika 1) izrađujemo kako bismo ostvarili sledeće prednosti kao što su: laka pokrivenost i lako merđovanje.



Slika 4.1 Graf toka Gmail aplikacije Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

KOMPONENTE PLANA TESTIRANJA: PREPOSTAVKE, RIZICI, PLAN UBLAŽAVANJA ILI PLAN ZA VANREDNE SITUACIJE

Prepostavka sadrži informacije o problemima tokom procesa testiranja; Rizici su izazovi sa kojima se moramo suočiti dok se planom za vanredne situacije oni mogu prevazići

Prepostavka: Sadrži informacije o problemima koji se mogu pojaviti tokom procesa testiranja i pisanja planova testiranja a odnose se na resurse, tehnologiju itd.

Rizik: To su izazovi sa kojima se moramo suočiti prilikom testiranja aplikacije u trenutnom rilizu. Na primer, odlaganje datum objavljivanja datog riliza.

Plan ublažavanja ili plan za vanredne situacije: To je rezervni plan koji je pripremljen za prevazilaženje rizika ili problema.

Primer: Pogledajmo jedan primer prepostavke, rizika i plan za nepredviđene slučajevе, obzirom da su oni međusobno povezani.

Prepostavićemo da su u proces testiranja nekog softverskog proizvoda uključena tri inženjera za testiranje i da su svakom od njih dodeljeni različiti moduli kao što su P, K i R. U ovom konkretnom scenariju rizik bi mogao biti da neki od inženjera testiranja napusti projekat u toku procesa testiranja.

Zbog toga će se u planu nepredviđenih događaja definisati primarni i sekundarni vlasnik svake od aktivnosti testiranja. Dakle, ako jedan inženjer testa ode, sekundarni vlasnik preuzima tu specifičnu funkciju i pomaže novom inženjeru testa, kako bi mogao da razume dodeljene module.

KOMPONENTE PLANA TESTIRANJA: ULOGE I ODGOVORNOSTI

Definišu kompletan zadatak koji treba da obavi ceo tim za testiranje.

Definišu kompletan zadatak koji treba da obavi ceo tim za testiranje. Kod velikih projekata, test menadžer je osoba koja piše plan testa. Ako postoje 3-4 mala projekta, tada će test menadžer dodeliti svaki od projekata pojedinačnim vođama testa (**Test Lead**) koji pišu plan testa za projekat koji im je dodeljen.

Primer: Pogledajmo jedan primer gde ćemo razumeti uloge i odgovornost menadžera testa, vođe testa i inženjera testa.

Uloga: Test menadžer

Ime: Rian

Odgovornosti:

- Priprema (piše i pregledava) plan testa
- Održavanje sastanka sa razvojnim timom
- Održavanje sastanaka sa timom za testiranje
- Održavanje sastanaka sa kupcem
- Objava beleške o rilizu
- Rukovanje eskalacijama i problemima

Uloga: Vođa testa (Test Lead)

Ime: Harvey

Odgovornosti:

- Priprema (pisanje i pregledavanje) plan testa
- Održavanje svakodnevnih stand up sastanaka
- Pregledavanje i odabir test slučajeva
- Priprema RTM i izveštaja
- Dodela modula
- Pravljenje rasporeda

Uloga: Test inženjer 1, Test inženjer 2, i Test inženjer 3

Ime: Louis, Jessica, Donna

Dodeljeni moduli: M1, M2, i M3

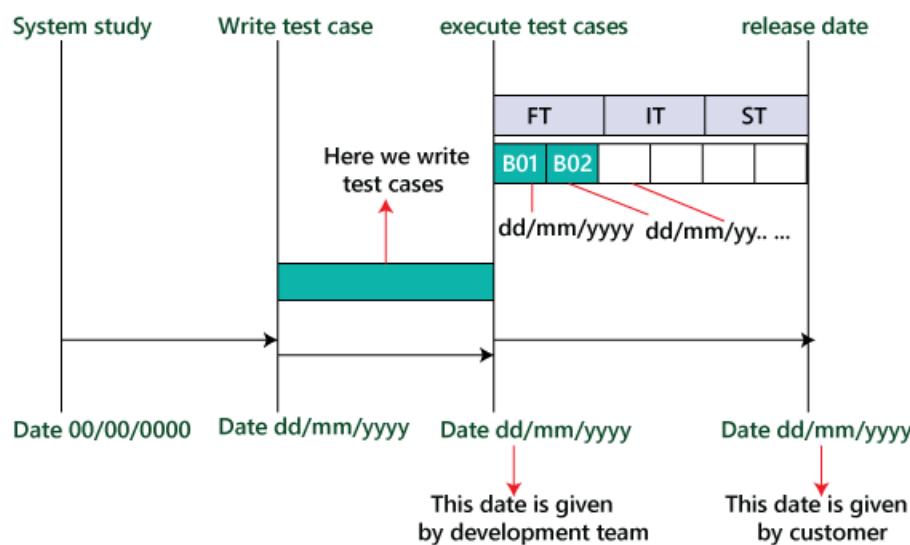
Odgovornosti:

- Pisanje, pregledavanje i izvršavanje test dokumenata koji se sastoje od test slučajeva i scenarija testiranja
- Čitanje, pregledavanje, razumevanje i analiziranje zahteva
- Utvrđivanje toka prolaska kroz aplikaciju
- Izvršenje test slučajeva
- RTM za odgovarajuće module
- Praćenje defekata
- Pripremite izveštaj o izvršenju testa i slanje vođi testa.

KOMPONENTE PLANA TESTIRANJA: RASPORED TESTIRANJA

Raspored testiranja se koristi za definisanje kada tačno koju aktivnost testiranja treba započeti i završiti.

Raspored testiranja se koristi za definisanje kada tačno koju aktivnost testiranja treba započeti i završiti.



Slika 4.2 Raspored testiranja Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

Kao što vidimo na slici 2. za svaku od aktivnosti (npr. pisanje test slučajeva, izvršenje test slučajeva itd.) se definiše datum početka i datum završetka. Takođe se navodi i datum završetka testiranja kojim se izbacuje određeni riliza softvera.

KOMPONENTE PLANA TESTIRANJA: PRAĆENJE DEFEKATA I TEST OKRUŽENJE

Praćenjem defekata definišemo kako pratimo greške, tehnike i alate za praćenje, ozbiljnost i prioritet defekata. Test okruženje je okruženje u kojima ćemo testirati aplikaciju.

Praćenje defekata: obično se radi korišćenjem alata, jer je nemoguće ručno pratiti status svake greške. Takođe je potrebno opisati na koji način greške koje su identifikovane tokom procesa testiranja prenosimo i šaljemo nazad razvojnom timu i kako će razvojni tim na to odgovoriti. Ovde takođe spominjemo prioritet grešaka kao što su visok, srednji i nizak.

Ovde su navedeni neki od aspekata praćenja defekata:

- Tehnike zapraćenja defekata
 -
 -
- Alati za praćenje defekata: Možemo da navedemo naziv alata koji ćemo koristiti za praćenje grešaka. Neki od najčešće korišćenih alata za praćenje grešaka su Jira, Bugzilla, Mantis i Trac, itd.
- Ozbiljnost defekta: može biti:
 - defekt koji blokira testiranje (blocked): ne možemo dalje da testiramo dati modul ali možemo da nastavimo sa testiranjem drugih modula.
 - kritičan: defekt će uticati na poslovanje.
 - glavni
 - sporedni: nedostaci koji utiču na izgled aplikacije.
- Prioritet: na osnovu prioriteta defekti se mogu kategorizirati na visokog, srednjeg i niskog prioriteta (P1, P2, P3 i P4).

Test okruženje: okruženje u kojima ćemo testirati aplikaciju. Imamo dve vrste okruženja: softverske i hardverske konfiguracije.

Konfiguracijom softvera se definišu detalji o operativnim sistemima kao što su Windows, Linux, UNIX i Mac i raznim čitačima kao što su Google Chrome, Firefox, Opera, Internet Explorer itd.

Hardverskom konfiguracijom se definišu informacije o različitim veličinama RAM-a, ROM-a i procesora.

Razvojni tim će obezbediti informacije kako da se instalira softver. Ako razvojni tim ne obezbedi taj postupak, u plan testa ćemo ga zapisati kao razvoj zasnovan na zadacima (**Task-Based Development - TBD**).

KOMPONENTE PLANA TESTIRANJA: KRITERIJUMI ZA ULAZAK I IZLAZAK

Ulagni i izlagni kriterijumi su neophodni uslovi koji treba ispuniti pre započinjanja procesa testiranja i u trenutku odluke o prestanku testiranja.

Ulagni i izlagni kriterijumi su neophodni uslovi koji treba ispuniti pre započinjanja procesa testiranja i u trenutku odluke o prestanku testiranja.

Treba ispuniti sledeće kriterijume pre ulaska u testiranje:

- Treba završiti testiranje bele kutije.
- Treba razumeti i proanalizirati zahteve i pripremiti odgovarajuća dokumenata za testiranje
- Treba spremiti podatke za testiranje
- Treba pripremiti aplikaciju za testiranje
- Moduli ili funkcije moraju biti dodeljeni različitim test inženjerima.
- Treba spremiti potrebne resurse

Izlazni kriterijumi podrazumevaju ispunjenje sledećih uslova:

- Treba izvršiti sve slučajeve testiranja.
- Većina testova treba da bude uspešno završena.
- Završetak zavisi od ozbiljnosti grešaka, što znači da ne sme biti blokera (grešaka koji blokiraju testiranje) ili velikih greške, dok neke manje greške mogu da postoje.

Gore navedene ulazne kriterijume treba ispuniti pre nego što počnemo sa obavljanjem funkcionalnog testiranja. Nakon što smo izvršili funkcionalno testiranje a pre početka integracionog testiranja, trebalo bi ispuniti izlazne kriterijume za funkcionalno testiranje, jer se o % izlaznih kriterijuma odlučuje na sastanku na kojem su prisutni i menadžer za razvoj i menadžer za testiranje. Ako se ne ispoštuju izlazni kriterijumi za funkcionalnog testiranja, onda ne možemo da nastavimo sa integracionim testiranjem.

KOMPONENTE PLANA TESTIRANJA: AUTOMATIZACIJA TESTIRANJA, PROCENA NAPORA I IZLAZI IZ TESTIRANJA

U okviru automatizacije testiranja treba odlučiti sledeće: Koje funkcije moraju biti automatski testirane, a koja ne?

U okviru **automatizacije testiranja** treba odlučiti sledeće:

- Koje funkcije moraju biti automatski testirane, a koja ne?
- Koji alat za automatizaciju testa ćemo koristiti?

Slučajevi testiranja se automatizuje tek nakon prvog riliza.

Ovde se postavlja pitanje da li ćemo i na osnovu čega odlučiti koje karakteristike moramo testirati?

Uobičajeno je da najčešće korišćene funkcije treba ponovo i ponovo testirati. Prepostavimo da testiramo Gmail aplikaciju čije su osnovne funkcije Sastavljanje pošte (compose mail), Poslata pošta (Sent Items) i Primljena pošta (inbox). Zato ćemo ove karakteristike automatski testirati, jer za ručno testiranje treba više vremena, a takođe postaje monoton posao.

Kako odlučujemo koje karakteristike neće biti testirane?

Prepostavimo da se funkcija Help aplikacije Gmail ne testira više puta jer se ta funkcija ne koristi redovno, pa ne moramo često da je proveravamo.

Ako su s druge strane neke funkcije nestabilne i imaju puno grešaka, te funkcije nećemo automatski testirati već ćemo kod njih primeniti ponovno ručno testiranje.

Procena napora (**Effort estimation**) sadrži planiran napor koji je potrebno da uloži svaki član tima.

Izlazi iz testiranja (**Test Deliverable**): to su dokumenta koje generiše tim za testiranje po završetku testiranja a koja se isporučuju kupcu zajedno sa softverskim proizvodom. Uključuju sledeće:

- Plan testiranja
- Slučajeve testiranja
- Skripta za testiranje
- RTM (Requirement Traceability Matrix)
- Izveštaj o defektima
- Izveštaj o izvršenju testa
- Grafikone i metrike
- Napomene o rilazu

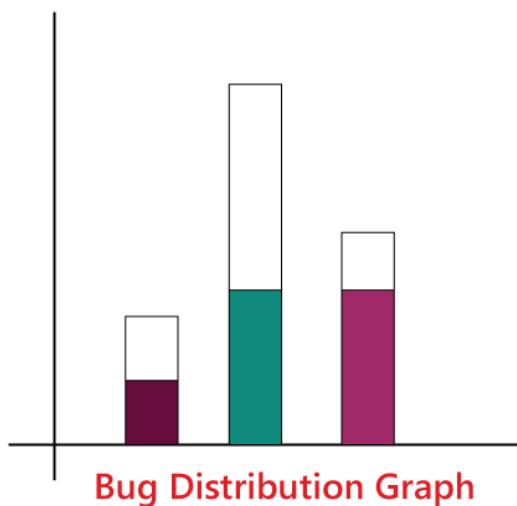
KOMPONENTE PLANA TESTIRANJA: GRAFIKONI (1-2)

Dajemo četiri različita grafikona koji prikazuju različite aspekte procesa testiranja. Ovde su dati primeri grafikona 1-2

Ovde će biti predstavljene neke vrste grafikona koje možemo uraditi tokom testiranja i biti dat uzorak svakog grafikona.

Kao što vidimo, imamo četiri grafikona koji prikazuju različite aspekte procesa testiranja.

Grafikon 1: Pokazuje koliko je defekata identifikovano i koliko je defekata otklonjeno u svakom modulu (slika 3)



Slika 4.3 Graf distribucije grešaka Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

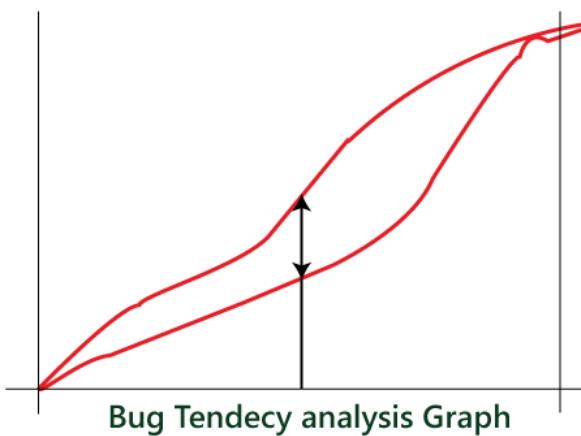
Grafikon 2: Slika 4 prikazuje koliko je kritičnih, glavnih i manjih defekata identifikovano za svaki modul, a koliko je ispravljeno za odgovarajuće module.

Slika 4.4 Raspored kritičnih, glavnih i manjih defekata po modulima Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

KOMPONENTE PLANA TESTIRANJA: GRAFIKONI (3-4) I METRIKE

U nastavku su dati grafikoni (3-4) kao primeri komponenata plana testiranja.

Grafikon3: Vođa testiranja (**test lead**) dizajnira grafikon analize trenda grešaka koji se kreira svakog meseca i šalje ga menadžmentu. Isto predviđanje se vrši i na završetku softverskog proizvoda. Na ovom grafikonu možemo da procenimo opseg ispravke i uočimo da kriva na donjoj slici ima tendenciju rasta.



Slika 4.5 Graf analite tendencije grešaka Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

Grafikon4: Ovu vrstu grafikona (slika 6.) kreira Test Manager. Grafikon je namenjen razumevanju raskoraka koji postoji između procenjenih grešaka i stvarnih grešaka koje su se dogodile, a takođe pomaže i u poboljšanju procene grešaka u budućnosti.

Slika 4.6 Razlika između stvarnih i procenjenih grešaka Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

KOMPONENTE PLANA TESTIRANJA: METRIKE

Na osnovu podataka iz grafikona, možemo kreirati metrike

Kao i u prethodnim slučajevima, kreiramo grafikon distribucije grešaka, koji je dat na slici 3, a pomoću gore pomenutih podataka takođe ćemo dizajnirati metriku. Na primer

Test Engineer Names	Critical		Major		Minor	
	Found	Fixed	Found	Fixed	Found	Fixed
John	50	46	60	20	80	10
James
Sophia

Slika 4.7 Metrika Izvor: <https://www.javatpoint.com/test-plan> [Izvor: NM SE321-2020/2021.]

Na gornjoj slici prikazujemo sve test inženjere na određenom projektu i broj nedostataka koji su utvrdili i otklonili. Ove podatke takođe možemo koristiti za buduću analizu. Kada dođe novi zahtev, na osnovu broja defekata koje su ranije otkrili prema gornjim pokazateljima, možemo da odlučimo kome ćemo dodeliti novu funkciju za testiranje. Bićemo u situaciji da znamo ko se može dobro nositi sa problematičnim funkcijama i pronaći maksimalan broj grešaka.

ANGAŽOVANJE NA IZRADI DOKUMENTACIJE TESTIRANJA

Angažovanje članova tima za testiranje na izradi plana testiranja i slučajeva testiranja

- 1. **Ko piše plan testiranja?**
 - Vodja testiranja (**Test Lead**) - 60%
 - Menadžer testa - 20%
 - Inženjer testa - 20% S
- 2. **Ko pregledava plan testiranja?**
 - Vodja testiranja
 - Menadžer testa
 - Inženjer testa
 - Kupac
 - Razvojni tim

Inženjer testa pregledava plan testiranja iz perspektive svog modula, a menadžer testa pregledava plan testiranja na osnovu mišljenja kupaca.

- 3. **Ko odobrava plan testiranja?**
 - Kupac
 - Menadžer testa
- 4. **Ko piše test slučajeve?**
 - Vodja testiranja
 - Inženjer testa
- 5. **Ko razmatra test slučajeve?**

- Inženjer testa
 - Vodja testiranja
 - Kupac
 - Razvojni tim
- 6. **Ko odobrava test slučajeve?**
- Menadžer testa
 - Vodja testiranja
 - Kupac

SMERNICE ZA IZRADU PLANA TESTIRANJA; ZNAČAJ PLANA TESTIRANJA

Prilikom pisanja plana testiranja treba se pridržavati određenih smernica. Plan testiranja je značajan dokument i može se tretirati kao knjiga pravila koja se mora poštovati.

Smernice za izradu plana testiranja:

1. Smanjite plan testiranja što je više moguće.
2. Izbegavajte preklapanja i bilo kakve suvišne informacije.
3. Ako mislite da vam nije potrebna neka od gore pomenutih komponenti u okviru plana testiranja, izbrišite je i nastavite dalje.
4. Budite precizni. Na primer, kada navedete softverski sistem kao deo okruženja za testiranje, navedite verziju softvera umesto samo imena softvera.
5. Izbegavajte dugačke pasuse.
6. Koristite liste i tabele gde god je to moguće.
7. Ažurirajte plan po potrebi.
8. Ne koristite zastarele dokumente.

Značaj plana testiranja:

1. Plan testiranja daje pravac našem razmišljanju. On je poput knjige pravila koja se mora poštovati.
2. Plan testiranja pomaže u utvrđivanju potrebnih napora za potvrđivanje kvaliteta softverske aplikacije koja se testira.
3. Plan testiranja pomaže onim ljudima koji treba da razumeju detalje testa a povezani su sa programerima, poslovnih menadžerima, kupcima itd.
4. U planu testiranja su dokumentovani važni aspekti poput rasporeda testiranja, strategije testiranja, obima testiranja itd., tako da ih tim menadžmenta može pregledati i ponovo koristiti za druge slične projekte.

▼ Poglavlje 5

Dokumentacija: Matrica sledljivosti

MATRICA SLEDLJIVOSTI

Matrica sledljivosti (Traceability Matrix) je dokument tabelarnog tipa koji se koristi u razvoju softverskih aplikacija za praćenje zahteva.

Matrica sledljivosti (Traceability Matrix) je dokument tabelarnog tipa koji se koristi u razvoju softverskih aplikacija za praćenje zahteva. Može se koristiti za praćenje unapred (od zahteva do projektovanja ili kodiranja) i unazad (od kodiranja do zahteva). Takođe je poznata i kao Matrica sledljivosti zahteva (Requirement Traceability Matrix RTM) ili Cross Reference Matrik (CRM).

Priprema se pre procesa izvršenja testa kako bi se osiguralo da su svi zahtevi pokriveni slučajevima testiranja i kako ne bismo propustili nijedno testiranje. U RTM dokumentu mapiramo sve zahteve i odgovarajuće slučajeve testiranja kako bismo bili sigurni da smo napisali sve test slučajeve za svaki uslov.

Inženjer testa priprema RTM za module koji su mu dodeljeni, a zatim ih šalje vođi testiranja. Vođa testiranja proverava rezervorijum kako bi proverio da li postoje slučajevi testiranja ili ne i na kraju vođa testiranja objedinjava i priprema jedan RTM dokument.

Ovaj dokument obezbeđuje da svaki zahtev bude pokriven slučajem testiranja, a da slučaj testiranja bude napisan na osnovu poslovnih potreba koje daje klijent. Piše se kako bi se osiguralo da budu pokriveni svi zahtevi. Na slici 1. možemo primetiti da se zahtevi pod brojevima 2 i 4 ne spominju (zbog toga su istaknuti) tako da možemo lako razumeti da se slučajevi testiranja moraju napisati za njih.

TRACEABILITY MATRIX

Requirement Number	Test Case Name
1	...
2	
3	...
4	
5	...
6	...
7	...
8	...

Slika 5.1 Primer matrice sledljivosti Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

OBRAZAC MATRICE SLEDLJIVOSTI

Generalno, ovaj dokument je poput radnog lista tabelarnog oblika, ali postoji i mnogo korisnički definisanih obrazaca (template) koji se mogu koristiti za matricu sledljivosti.

Generalno, ovaj dokument je poput radnog lista tabelarnog oblika, ali postoji i mnogo korisnički definisanih obrazaca (**template**) koji se mogu koristiti za matricu sledljivosti. Svaki zahtev u matrici sledljivosti je povezan je sa odgovarajućim test slučajem, tako da se testovi mogu uzastopno izvršavati u skladu sa određenim zahtevima.

RTM ne pišemo u trenutku kada pišemo testove, jer može biti nepotpuna, a takođe ni nakon pisanja test slučajeva jer neki slučaj testiranja može biti odbijen. RTM dokument osigurava da postoji barem jedan slučaj testiranja za svaki zahtev, ali ne osigurava da svi mogući test slučajevi budu napisani za određeni zahtev.

Na slici 2. je dat jedan od mogućih obrazaca koji se može koristiti pri izradi matrice sledljivosti a na slici 3. jedan primer matrice.

Ciljevi matrice sledljivosti:

- Pomaže u praćenju dokumenata koji su razvijeni tokom različitih faza SDLC-a.
- Osigurava da softver u potpunosti ispunjava zahteve kupca.
- Pomaže u otkrivanju osnovnog uzroka bilo koje greške.

Requirement no	Module name	High level requirement	Low level requirement	Test case name

Slika 5.2 Jeden od mogućih obrazaca koji se može koristiti pri izradi matrice sledljivosti Izvor: Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

Slika 5.3 Primer matrice sledljivosti Izvor: Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

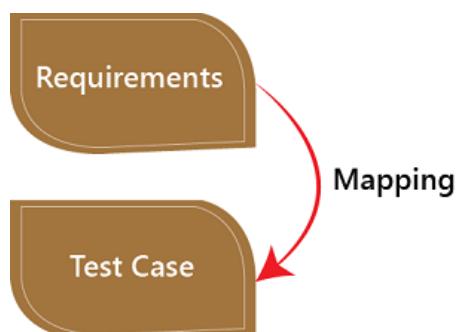
VRSTE MATRICE ZA ISPITIVANJE SLEDLJIVOSTI

Matrice sledljivosti se mogu klasifikovati u tri različite vrste i to:

Sledljivost unapred; Sledljivost unazad ili riverzna sledljivost;

Dvosmerna sledljivost

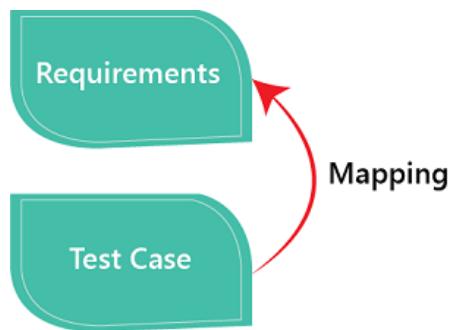
Matrica sledljivosti testova unapred se koristi kako bi se osiguralo da se poslovne potrebe ili zahtevi izvršavaju pravilno u aplikaciji i takođe strogo testiraju. Glavni cilj ovoga je proveriti da li razvoj proizvoda ide u dobrom smeru. U njoj su zahtevi mapirani u smeru ka test slučajevima. (slika 4.)



Slika 5.4 Matrica sljedivosti testova unapred Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

Riverzna ili matrica sledljivost unazad se koristi za proveru da proizvod ne povećavamo poboljšavanjem elemenata dizajna, koda, testiranjem drugih stvari koje nisu pomenute u poslovnim potrebama. Glavni cilj je da postojeći projekat ide u ispravnom smeru. U ovoj vrsti matrice se zahtevi mapiraju u smeru unazad prema slučajevima testiranja. (slika 5.)

Matrica dvosmerne sledljivosti: To je kombinacija matrice sledljivosti unapred i unazad, koja se koristi kako bi se obezbedilo da se u test



Slika 5.5 Riverzna ili matrica sledljivost unazad Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

slučajevima izvrše sve poslovne potrebe. Ona takođe procenjuje modifikaciju zahteva koja se javlja zbog grešaka u aplikaciji.

Slika 5.6 Matrica dvosmerne sljedivost Izvor: <https://www.javatpoint.com/traceability-matrix> [Izvor: NM SE321-2020/2021.]

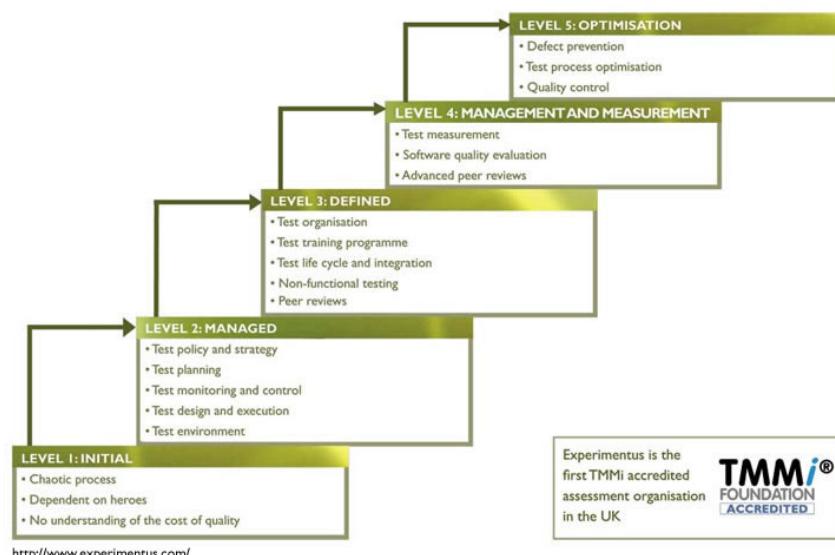
✓ Poglavlje 6

Nivoi zrelosti procesa testiranja (TMM)

MODEL ZRELOSTI PROCESA TESTIRANJA SOFTVERA - TMM NIVO 1 I 2

Na TMM 1 nivou testiranje je haotičan proces i nije razgraničen sa fazom otklanjanja grešaka (debugging). Na TMM 2 nivou testiranje je odvojeno od debugging-a i vrši se posle kodiranja

U novije vreme, uvođeni su modeli zrelosti testiranja softvera (Test Maturity Model - TMM), a koji je objavio Illinois Institute of Technology iz Čikaga 1996 godine, analogno CMM modelu zrelosti procesa razvoja softvera (CMM) koji je razvio Software Engineering Institute (SEI) of Carnegie Mellon University. Ovde se daje kratak opis glavnih karakteristika svakog nivoa modela zrelosti procesa testiranja softvera (TMM) kao što je na slici 1 prikazano.



Slika 6.1 Pet nivoa zrelosti procesa testiranja po TMM modelu [Izvor: NM SE321-2020/2021.]

TMM Nivo 1 (Inicijalna faza): Testiranje softvera (TS) je haotičan proces, loše je definisan i nije jasno razgraničen sa fazom otklanjanja grešaka (debugging). Testiranju se pristupa neplanirano i na kraju faze kodiranja programa. Cilj testiranja softvera je da se pokaže da program radi. Softver se iznosi na tržište bez primene sistema obezbeđenja kvaliteta. Nedostaju resursi, alati i adekvatno obučen kadar. Ovaj tip organizacije odgovara SEI/CMM Level 1, zrelosti softverske kompanije.

TMM Nivo 2 (Faza Definisanja): Testiranje softvera je odvojeno od faze otklanjanja grešaka (debugging) i definisano je kao odvojena faza nakon kodiranja. Mada je planirana kao aktivnost, Testiranje softvera na Nivou 2, je definisano nakon faze kodiranja zbog nezrelosti samog procesa Testiranja softvera. Glavni cilj testiranja softvera, na ovom nivou zrelosti, je da se pokaže da je softver zadovoljio specifikaciju. Primenuju se osnovne tehnike i postupci. Mnogi problemi vezani za kvalitet softvera na ovom TMM nivou, posledica su planiranja testiranja softvera kasno u ciklusu razvoja softvera (SDLC). Dalje, greške (otkazi) softvera u ranim fazama prostiru se do poslednjih faza SDLC tj. ne otkrivaju se blagovremeno, odnosno onda kada se i generišu.

MODEL ZRELOSTI PROCESA TESTIRANJA SOFTVERA - TMM NIVO 3 I 4

Na TMM nivou 3 testiranja softvera se odvija od početka SDLC pa do kraja najčešće V modela SDLC. Na TMM nivou 4 proces testiranja softvera se meri i ocenjuje se kvalitet softvera

TMM Nivo 3 (Faza Integrisanja): Testiranje softveranije nije više faza koja sledi fazu kodiranja, naprotiv, testiranje softvera je integrисани deo SDLC. Za razliku od TMM Nivoa 2, na Nivou 3 aktivnost testiranja softvera se odvija i planira od početka SDLC tj. projektnih zahteva za softver pa do kraja najčešće V modela SDLC. Ciljevi i zadaci testiranja softvera su utvrđenina bazi zahteva klijenata i mogućih kupaca softvera i koriste se u fazi dizajna test primera i kriterijuma uspešnog odziva testa.

Organizaciono je uspostavljena grupa za testiranje softvera, i testiranje je profesionalni posao članova tima. Obuka kadra je fokusirana na oblast testiranja. Osnovna sredstva, alati za testiranje softvera su u upotrebi. Iako organizacije na ovom TMM nivou znaju za značaj kontrole i obezbeđenja kvaliteta, ova funkcija nije formalno primenjena u SDLC. Program merenja kvaliteta testiranja softvera kao i samog kvaliteta softvera kao proizvoda nije još uspostavljen.

TMM Nivo 4 (Faza Merenja i Upravljanja): Proces testiranja softvera se meri i ocenjuje se kvalitet (cena, efikasnost, efektivnost) softvera. Inspekcije revizije se primenjuju planski u svim fazama SDLC kao obavezna aktivnost u testiranju softvera i kontroli kvaliteta. Softverski proizvod se testira radi ocene faktora kvaliteta kao što su pouzdanost, upotrebljivost i pogodnost za održavanje. Ažurira se baza podataka o test-primerima sa svih projekata radi ponovne upotrebe pri regresionom (ponovljenom) testiranju. Otkazi, greške, se evidentiraju u bazi podataka o otkazima, greškama i dodeljuje im se značaj (kritičnost). Nedostatak testiranja na ovom TMM nivou je i dalje nedovoljno primenjena preventivna aktivnost generisanja softverskih grešaka, slabo razvijena metrika kvalitetatestiranja kao i sredstva automatizacije testiranja.

TREND U KVALITETU SOFTVERA PREMA CMM NIVOIMA ZRELOSTI - TMM NIVO 4 I 5

TMM Nivo 5 pristupa finom podešavanju i stalnom unapređenju kvaliteta testiranja. Testiranje je kontrolisano statističkim postupcima uzorkovanja i metrikom kvaliteta testiranja .

TMM Nivo 5 (Faza Optimizacije, Prevencije grešaka i Kontrola kvaliteta): Nakon uspešne izgradnje infrastrukture kroz sazrevanje TMM od Nivoa 1 do 4, za koji se može reći da je testiranje softvera definisan i kontrolisan, preko metrika kao što su troškovi, efikasnost, efektivnost sada se na TMM Nivo 5 pristupa finom podešavanju i stalnom unapređenju kvaliteta testiranja. Proces testiranja je kontrolisan statističkim postupcima uzorkovanja i metrikom kvaliteta testiranja kao što su troškovi, efikasnost, efektivnost. Uspostavljena je procedura za izbor i ocenu sredstava i alata za testiranje. Automatska sredstva testiranja se koriste u svim fazama testiranja softvera dizajnu test-primera, izvršavanju testova, ponovnom izvršavanju, ažuriranju baze podataka o otkazima, greškama, alati za metriku, praćenje generisanja i analizu uzroka istih kao i sredstva održavanja tzv. "Testware". Poboljšanje kvaliteta softverskog proizvoda uvođenjem CMM i TMM modela u kompaniji je prikazano na slici 2.

CMM(I) Maturity level	Design Faults / KLOC (Keene)	Delivered Defects / FP (Jones)	Shipped Defects / KLOC (Krasner)	Relative Defect Density (Williams)	Shipped Defects (Rifkin)
5	0,5	0,05	0,5	0,05	1
4	1	0,14	2,5	0,1	5
3	2	0,27	3,5	0,2	7
2	3	0,44	6	0,4	12
1	5-6	0,75	30	1,0	61

Slika 6.2 Trend u kvalitetu softvera prema CMM nivoima zrelosti [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 7

Grupna vežba: dokumentacija testiranja

PRIMERI RAZLIČITIH TIPOVA TEST DOKUMENATA

U ovoj grupnoj vežbi će biti dati primeri nekih od test dokumenata.

Kao što je rečeno, u testiranju softvera postoje različiti tipovi test dokumenata kao što su:

1. Scenarija testiranja ([Test scenarios](#))
2. Slučajevi testiranja ([Test case](#))
3. Plan testiranja ([Test plan](#))
4. Matrica za praćenje zahteva ([Requirement traceability matrix - RTM](#))
5. Strategija testiranja ([Test strategy](#))
6. Podaci za testiranje ([Test data](#))
7. Izveštaj o greškama ([Bug report](#))
8. Izveštaj o izvršenju testa ([Test execution report](#))

U ovoj grupnoj vežbi će biti dati primeri nekih od njih.

PRIMERI SCENARIJA TESTIRANJA: ZA MODUL PRIJAVA (LOGIN) I SASTAVLJANJE MAIL-A (COMPOSE)

Primer scenarija testiranja se odnosi na Gmail aplikaciju u okviru koje se testiraju moduli: Prijava (Login), Sastavljanje mail-a (Compose), Pregledavanje mail-ova (Inbox) i Brisanje mail-

Primer scenarija testiranja se odnosi na Gmail aplikaciju. Scenariji testiranja su napisani za različite module koji se najčešće koriste u okviru Gmail aplikacije kao što su: prijava ([Login](#)), sastavljanje mail-a ([Compose](#)), pregledavanje primljenih mail-ova ([Inbox](#)) i brisanje mail-ova ([Trash](#)).

Scenario testiranja za modul Prijava ([Login](#)):

- Unesite validne detalje za login (korisničko ime, lozinka) i proverite da li se prikazuje početna stranica.
- Unesite nevažeće korisničko ime i lozinku (username i password) i proverite početnu stranicu.
- Korisničko ime i lozinku ostavite prazne i proverite da li se prikazuje poruka o grešci.
- Unesite važeće podatke za prijavu, kliknite na otkaži i proverite da li se polja resetuju.

- Unesite nevažeće podatke za prijavu više od tri puta i proverite da li je račun blokiran.
- Unesite važeću prijavu i proverite da li je korisničko ime prikazano na home ekranu.

Scenario testiranja za modul Sastavljanje mail-a ([Compose](#)):

- Proverite da li mogu da unose polja o korisnicima kojima se šalje pošta (To, Cc i Bcc).
- Proverite da li se mogu uneti različiti ID-ovi e-pošte u polja To, Cc i Bcc.
- Sastavite mail, pošaljite ga i proverite da li se pojavljuje poruka za potvrdu o slanju mail-a.
- Sastavite mail, pošaljite ga i proverite da li on postoji u poslatim porukama pošiljaoca u poštanskom sandučetu (inbox-u).
- Sastavite mail, pošaljite ga i proverite validnost formata mail-a, da li je ID mail-a pogrešan ili tačan, proverite mail u poštanskom sandučetu pošiljaoca.
- Sastavite mail, izbrišite ga, a zatim proverite poruku o potvrdi slanja mail-a i proverite draft.
- Sastavite mail, sačuvajte mail kao draft i proverite poruku za potvrdu slanja mail-a
- Sastavite mail, kliknite na zatvori i proverite poruku o potvrdi slanja mail-a kao i da li je mail sačuvan kao draft.

PRIMERI SCENARIJA TESTIRANJA: ZA MODUL PREGLEDAVANJE PRIMLJENIH MAIL-OVA (INBOX) I BRISANJE MAIL-OVA (TRASH)

Primer scenarija testiranja se odnosi na Gmail aplikaciju u okviru koje se testiraju moduli: pregledavanje primljenih mail-ova (Inbox) i brisanje mail-ova (Trash)

Scenario testiranja za modul pregledavanje primljenih mail-ova ([Inbox](#)):

- Kliknite na prijemno sanduče (Inbox) i proverite da li je sva primljena pošta prikazana i istaknuta([highlighted](#)) u prijemnom sandučetu.
- Proverite da li je u primljenoj pošti ispravno prikazan id mail adrese pošiljaoca.
- Izaberite mail, odgovorite i prosledite ga; proverite poslate mail-ove pošiljaoca i primljene mail-ove primaoca.
- Proverite da li postoje prilozi (attachments) uz poštu koji su preuzeti ili ne.
- Proverite da li je prilog (attachments) ispravno skeniran na virusne pre preuzimanja.
- Izaberite jedan mail, odgovorite na njega i prosledite ga kao [draft](#), a zatim proverite poruku o potvrdi u [draft](#) sekciju.
- Proverite da li mejlovi koji su označeni kao pročitani nisu istaknuti ([highlighted](#)).
- Proverite da su svi primaoci mail-a u kopiji Cc vidljivi svim korisnicima.
- Proverite da su svi primaoci mail-a u skrivenoj kopiji Bcc nevidljivi korisnicima.
- Izaberite jedan mail, izbrišite ga, a zatim proverite odeljak [Trash](#).

Scenario testiranja za modul brisanje mail-ova ([Trash](#)).

- Otvorite [Trash](#), proverite svu izbrisani poštu.
- Uradite restore [Trash](#)-a.

- Izaberite jedan mail iz **Trash-a**, izbrišite ga i proverite da li je mail trajno izbrisano.

Napomena: planirano vreme za prikaz primera scenarija testiranja 15 minuta

PRIMER SLUČAJA TESTIRANJA: LOGIN NA VEB APLIKACIJU - POZITIVAN SCENARIO

Na osnovu obrasca koji je dat u predavanjima, ovde je dat primer koji prikazuje koncepte na razumljiviji način.

U okviru ove grupne vežbe su takođe dati primjeri slučajeva testiranja za čiju izradu je korišćen obrazac priložen u predavanju. Primer se odnosi na testiranje funkcionalnosti prijave (login) na bilo koju veb aplikaciju, recimo Facebook. Na slici 1. je dat pozitivan scenario testiranja

Test Scenario ID	Login-1		Test Case ID	Login-1A			
Test Case Description	Login – Positivan slučaj testiranja		Test Priority	visok			
Pre-Requisite	Validan korisnički račun (user account)		Post-Requisite	NA			
Test Execution Steps:							
S.No	Akcija	ulazi	Očekivani izlazi	Stvarni izlaz	Test Bro wser	Rezultat testa	Komentar testa
1	Pokreni aplikaciju	https://www.facebook.com/	Facebook home	Facebook home	IE - 11	prošao	[Priya 12/17/2016 11:44 AM]: Uspešno pokretanje
2	Unesi ispravan Email & Password i klikni na dugme login	Email id : test@xyz.com Password: *****	Uspešan login	Uspešan login	IE - 11	prošao	[Priya 12/17/2016 11:45 AM]: Uspešan Login

Slika 7.1.1 Primer slučaja testiranja: login na veb aplikaciju - pozitivan scenario [Izvor: NM SE321-2020/2021.]

Napomena: primer ovog slučaja testiranja zajedno sa korišćenim obrascem (template) je priložen kao resource file

PRIMER SLUČAJEVA TESTIRANJA: LOGIN NA VEB APLIKACIJU - NEGATIVAN SCENARIO

Negativan scenario za slučaj testiranja logina na veb aplikaciju dat je na slici.

Negativan scenario za slučaj testiranja logina na veb aplikaciju dat je na slici 2.

Test Scenario ID	Login-1		Test Case ID	Login-1B			
Test Case Description	Login – Negativan slučaj testiranja		Test Priority	visok			
Pre-Requisite	NA		Post-Requisite	NA			
Test Execution Steps:							
S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments
1	Pokreni aplikaciju	https://www.facebook.com/	Facebook home	Facebook home	IE - 11	prošao	[Priya 12/17/2016 11:44 AM]: Uspešno pokretanje
2	Unesi pogrešan Email & Password i klikni na dugme login	Email id : invalid@xyz.com Password: *****	Email adresa ili broj telefona koji ste uneli se ne podudara ni sa jednim računom. Otvori računom.	Email adresa ili broj telefona koji ste uneli se ne podudara ni sa jednim računom. Otvori računom.	IE - 11	prošao	[Priya 12/17/2016 11:45 AM]: Pogrešan login stopiran pokušaj
3	Unesi Email & pogrešan Password i klikni na dugme login	Email id : valid@xyz.com Password: *****	Password koji ste uneli je pogrešan. Zaboravili ste password?	Password koji ste uneli je pogrešan. Zaboravili ste password?	IE - 11	prošao	[Priya 12/17/2016 11:46 AM]: Pogrešan login stopiran pokušaj

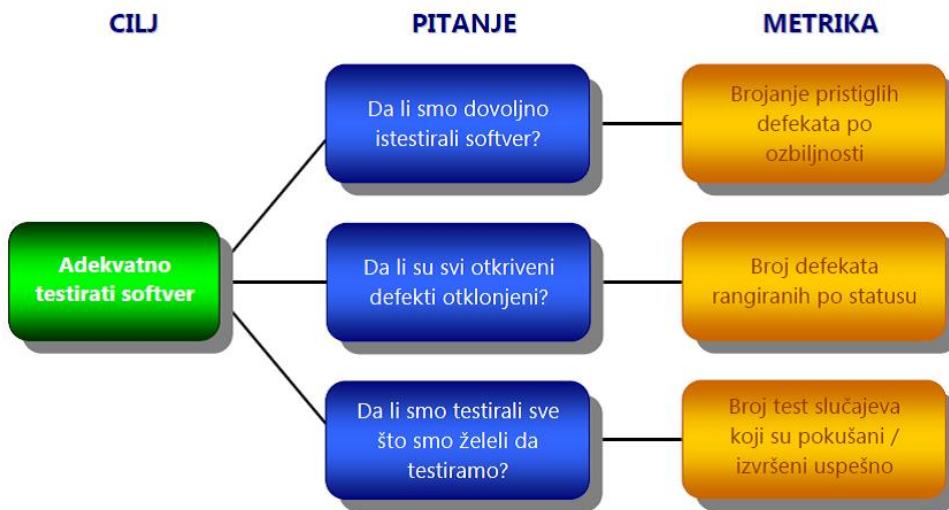
Slika 7.1.2 Negativan scenario za slučaj testiranja logina na veb aplikaciju [Izvor: NM SE321-2020/2021.]

Napomena: planirano vreme za prikaz primera slučajeva korišćenja 10 minuta

IZVEŠTAVANJE TOKOM PROCESA TESTIRANJA

Usled prilično komplikovane strukture zadatka koji tim za testiranje treba da obavi, javlja se potreba za kvalitetnim vidom praćenja čitavog procesa i izveštavanjem svih učesnika u testiranju

Usled prilično komplikovane strukture zadatka koji tim za testiranje treba da obavi, javlja se potreba za kvalitetnim vidom praćenja čitavog procesa i izveštavanjem svih učesnika u testiranju. Planom testiranja može biti definisano nekoliko osnovnih načina praćenja čitavog procesa, ali se njihov broj po potrebi može i povećati. Sami načini merenja napretka testiranja (metrike) će biti definisani na osnovu konkretnih ciljeva i pitanja na koja treba odgovoriti. To se najbolje može videti iz dijagrama prikazanog na slici 3.



Slika 7.1.3 Metrike procesa testiranja [Izvor: NM SE321-2020/2021.]

IZVEŠTAJ O PRONAĐENIM DEFEKTIMA

Svako izvršeno testiranje će pratiti odgovarajući Izveštaj o testiranju, koji sastavlja grupa koja je testiranje izvršila (ili tester) a pregleda i overava test menadžer.

Svako izvršeno testiranje će pratiti odgovarajući Izveštaj o testiranju, koji sastavlja grupa koja je testiranje izvršila (ili tester) a pregleda i overava test menadžer. Obrazac za ovakav izveštaj je dat na slici 4 i 5.

IZVEŠTAJ O TESTIRANJU			
Datum:	/ /20 .		
Mesto:			
ID broj testiranja:			
Test osoblje:			
Voda tima:			
Prvi član:			
Drugi član:			
Treći član:			
Vreme testiranja:			
Od	/ /20 . (datum)	:	(sat)
Do	/ /20 . (datum)	:	(sat)
Fokusne komponente testiranja:			
Hardverske:			
Softverske:			
Upotrebљeni resursi testiranja:			
Hardverski:			
Softverski:			
Test slučajevi:			
ID skupa test slučajeva:			
Opis:			
Rezultati testiranja:			
ID test slučaja:	Dobijene vrednosti:		
ID test slučaja:	Dobijene vrednosti:		
ID test slučaja:	Dobijene vrednosti:		
ID test slučaja:	Dobijene vrednosti:		
ID test slučaja:	Dobijene vrednosti:		
Podudaranja i nesaglasnosti sa očekivanim rezultatima:			
Pronadene anomalije:			
ID izveštaja:	Opis:	Otklonjena:	
ID izveštaja:	Opis:	Otklonjena:	

Slika 7.1.4 Izvještaj o testiranju [Izvor: NM SE321-2020/2021.]

Ukoliko se u toku testiranja otkriju neki defekti (bilo koje kategorije), sastaviće se odgovarajući Izveštaj o pronađenim defektima, čiji je obrazac dat na slici 5. Ovaj izveštaj je obavezan sastaviti onaj tim koji je pronašao defekt(e), a test menadžer će potvrditi njegovu validnost.

Slika 7.1.5 Izvještaj o testiranju - nastavak [Izvor: NM SE321-2020/2021.]

IZVEŠTAJ O ZAVRŠENOJ FAZI TESTIRANJA

Nakon svake faze testiranja sastaviti Izveštaj o završenoj fazi testiranja koji treba da pokrije sve obavljene testove u toj fazi i proceni opšti napredak procesa testiranja.

Za potrebe izveštavanja menadžera projekta, kao i za potrebe unutrašnjeg planiranja, test menadžer će nakon svake faze testiranja sastaviti Izveštaj o završenoj fazi testiranja koji treba da pokrije sve obavljene testove u toj fazi i proceni opšti napredak procesa testiranja. Prilikom pravljenja ovog izveštaja koristiće adekvatne metode grafičke predstave realizacije testiranja, kao i obrazac koji je dat na slici 6.

IZVEŠTAJ O PRONAĐENIM DEFEKTIMA	
Datum:	/ /20 .
Mesto:	
ID broj Izveštaja:	
Test osoblje:	
Vodi tima:	
Prvi član:	
Dруги član:	
Treći član:	
Vreme testiranja:	
Od / /20 .(datum) : (sat)	
Do / /20 .(datum) : (sat)	
Fokusne komponente testiranja:	
Hardverske:	
Softverske:	
Upotrebljeni resursi testiranja:	
Hardverski:	
Softverski:	
Naziv detektovane greške:	
Kategorija detektovane greške:	
Testiranje pri kojem se detektovala greška:	
Test slučaj pri kojem se detektovala greška:	
Izvor defekta:	
Hardverska komponenta:	
Softverska komponenta:	
Opis greške i njenih simptoma:	
Nivo propagacije greške:	
Mogući uzroci i eventualni načini otklona greške:	
Način na koji je greška otklonjena (popunjavanje test menadžer):	
Potpis vode tima: _____	
Pregledao i overio test menadžer: _____	

Slika 7.1.6 Izveštaj o pronađenim defektima [Izvor: NM SE321-2020/2021.]

Ovaj izveštaj je obavezan, njega će sastaviti onaj tim koji je pronašao defekt(e), a test menadžer će potvrditi njegovu validnost.

Za potrebe izveštavanja menadžera projekta, kao i za potrebe unutrašnjeg planiranja, test menadžer će nakon svake faze testiranja sastaviti Izveštaj o završenoj fazi testiranja koji treba da pokrije sve obavljene testove u toj fazi i proceni opšti napredak procesa testiranja.

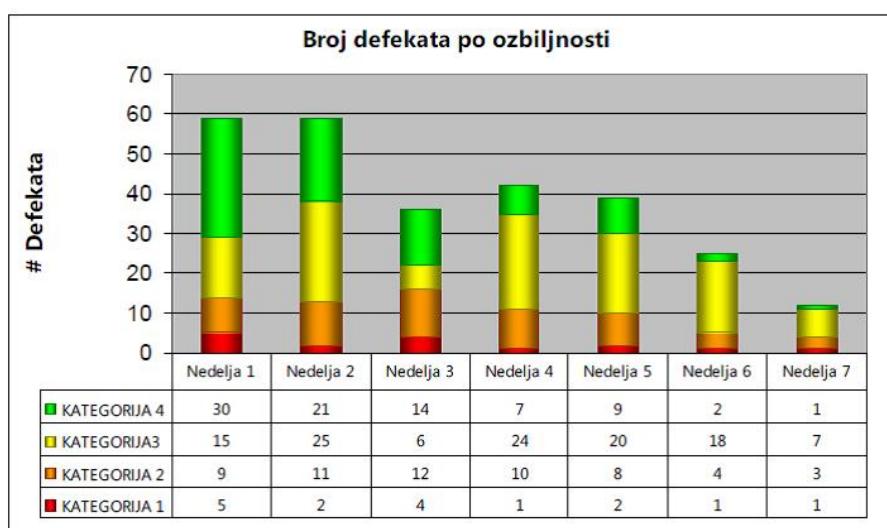
Prilikom pravljenja ovog izveštaja koristiće adekvatne metode grafičke predstave realizacije testiranja, kao i obrazac koji je dat na slici 7.

Slika 7.1.7 Izveštaj o završenoj fazi testiranja [Izvor: NM SE321-2020/2021.]

PRAĆENJE OTKRIVENIH DEFEKATA

Broj pristiglih defekata rangiranih po katastrofalnosti može se prikazati dijagramu.

BROJ PRISTIGLIH DEFEKATA RANGIRANIH PO KATASTROFALNOSTI može se prikazati dijagramu ssličnim onim koji je prikazan na slici 8:



Slika 7.1.8 Primer grafikona broja defekata po ozbiljnosti [Izvor: NM SE321-2020/2021.]

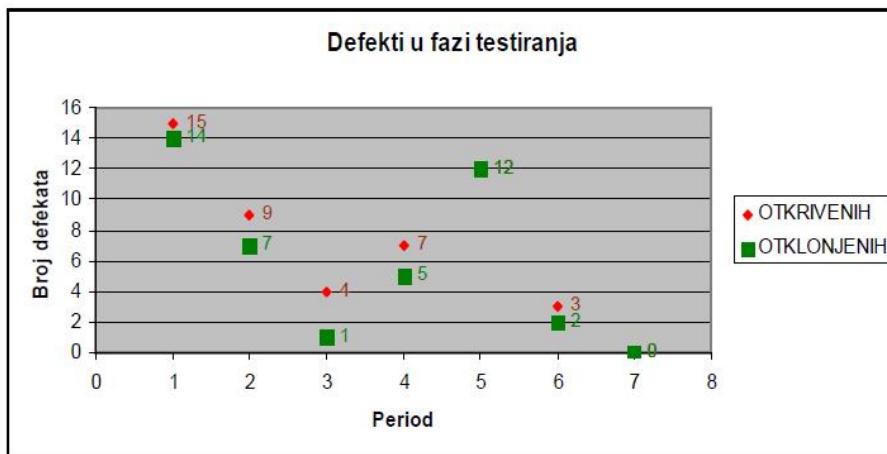
Na Slici 9 sledi primer grafikona koji prikazuje napredak u izvršavanju jedne faze testiranja.

Slika 7.1.9 Primer grafikona o napretku testiranja [Izvor: NM SE321-2020/2021.]

GRAFIKON BROJA OTKRIVENIH I ISPRAVLJENIH DEFEKATA

Pored navedenog, poželjno je priložiti i grafikon koji će prikazivati broj otkrivenih i ispravljenih defekata u konkretnoj fazi testiranja.

Pored navedenog, poželjno je priložiti i grafikon koji će prikazivati broj otkrivenih i ispravljenih defekata u konkretnoj fazi testiranja. Primer ovog grafikona dat je na slici 10:



Slika 7.1.10 Primer grafikona o defektima [Izvor: NM SE321-2020/2021.]

Ovi izveštaji će se tretirati kao poverljivi i mogu se dostavljati samo sledećim licima:

- Menadžer projekta;
- QA menadžer;
- Sponzor projekta;
- Stručni konsultant za testiranje.

Takođe, po jedan primerak svakog izveštaja ide u projektnu dokumentaciju.

Napomena: planirano vreme za prikaz primera grafikona 20 minuta

✓ 7.1 Individualne vežbe 09

PLAN TESTIRANJA SOFTVERA

Na osnovu obrađenih tehnika testiranja softvera izraditi proces testiranje mejl sistema ili ISUM sistema.

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka). Napraviti plan testiranja koji treba da obuhvati što veći broj navedenih elemenata:

- Ciljevi testiranja
- Opseg testiranja
- Metodologija testiranja
- Pristup testiranja
- Prepostavke, rizici, plan ublažavanja ili plan za vanredne situacije
- Uloge i odgovornosti
- Raspored testiranja
- Praćenje defekata
- Test okruženje

- Kriterijumi za ulazak i izlazak
- Automatizacija testiranja,
- Procena napora
- Grafikoni i metrike

Prilikom izrade plana testiranja koristiti deo predavanja u kojima su date smernice za opis svake od komponenti plana testiranja.

Vreme izrade zadatka 90 minuta

✓ Poglavlje 8

Domaći zadatak

DEVETI DOMAĆI ZADATAK

Nakon devete lekcije potrebno je uraditi deveti domaći zadatak.

Za odabranu aplikaciju koju ste radili na nekom od predmeta koje ste prethodno slušali i položili primeniti sledeće:

1. Napraviti plan testiranja tako da sadrži najmanje 5 elemenata
2. Napraviti matricu sledljivosti i napišite nekoliko slučajeva testiranja koje ste u njoj predviđeli

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnog okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ09-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ09.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Upravljanje procesom testiranja softvera

ZAKLJUČAK

Šta smo naučili u ovoj lekciji?

U ovoj lekciji je bilo reči o postupku softverskog testiranja, koji poznat i kao Životni ciklus testiranja softvera (**Software Testing Life Cycle - STLC**) kao i različitim vrstama dokumenta za testiranje koji se generišu tokom ili pre testiranja softverske aplikacije. Tu spadaju:

1. Scenarija testiranja (**Test scenarios**)
2. Slučajevi testiranja (**Test case**)
3. Plan testiranja (**Test plan**)
4. Matrica za praćenje zahteva (**Requirement traceability matrix - RTM**)
5. Strategija testiranja (**Test strategy**)
6. Podaci za testiranje (**Test data**)
7. Izveštaj o greškama (**Bug report**)
8. Izveštaj o izvršenju testa (**Test execution report**)

od kojih su neke detaljno opisane u predavanju i dati odgovarajući primeri kako bi studenti mogli da ih koriste i kreiraju tokom svog praktičnog rada.

U lekciji se govori i o modelima zrelosti testiranja softvera (**Test Maturity Model - TMM**) kojih ima pet.

LITERATURA ZA LEKCIJU 09

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. Anne Mette Jonassen Hass, Guide to Advanced Software Testing, © 2008 ARTECH HOUSE, INC.
4. Dr Kelvin Ross, Practical Guide to Software System Testing, K. J. Ross & Associates Pty. Ltd., 1998.

Dopunska literatura:

1. Burnstein I., Practical Software Testing, Springer-Verlag New York, 2003 (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. British Computer Society, Standard for Software Component Testing, Date: 27 April 2001.

Veb lokacije:

1. <https://www.javatpoint.com/testing-documentation>
2. <https://www.javatpoint.com/test-plan>
3. <https://www.javatpoint.com/test-scenario>
4. <https://www.javatpoint.com/test-case>
5. <https://www.javatpoint.com/testing-documentation>
6. <https://www.javatpoint.com/traceability-matrix>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Merenje i predikcija kvaliteta
softvera

Lekcija 10

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 10

MERENJE I PREDIKCIJA KVALITETA SOFTVERA

- ▼ Merenje i predikcija kvaliteta softvera
- ▼ Poglavlje 1: Greške u testiranju softvera
- ▼ Poglavlje 2: Životni ciklus grešaka
- ▼ Poglavlje 3: SQA model
- ▼ Poglavlje 4: Statički SQA alati
- ▼ Poglavlje 5: Dinamički SQL alati
- ▼ Poglavlje 6: Grupna vežba
- ▼ Poglavlje 7: Domaći zadatak
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Kvantifikacija analize defekata podrazumeva uvođenje mera, koje omogućavaju upoređivanje različitih defakata, njihovo klasifikovanje i druge operacije, kao i kod merenja dužine ili težine.

U ovoj lekciji će biti izloženi najjednostavniji metodi merenja i upravljanja kvalitetom, koji su baš zbog jednostavnosti i efikasnosti stekli popularnost u drugim oblastima primene, a uspešno se mogu primeniti i u softverskom inženjerstvu.

Razvoj računarske tehnike i povećanje kompleksnosti softvera su sve više praćeni tendencijom uvođenja kvantitativnih metoda u obezbeđenje kvaliteta softvera. Ove metode se sastoje pretežno u sledećem:

1. **Informacije o defektima se prikupljaju i klasikuju.**
2. **Za svaki defekt se identificuje mogući uzrok (npr. neusklađenost sa specifikacijama, greška u dizajnu, loša komunikacija s korisnikom ...)**
3. **Koristeći Paretov princip (80% defekata je izazvano sa 20% uzroka) izolovati dominantne uzroke**
4. **Nakon izolovanja malog broja najznačajnijih uzroka preuzeti mere za njihovo otklanjanje- Ovaj relativno prosti koncept predstavlja važan korak ka izgradnji adaptivnog softverskog inženjerskog procesa koji se menja da bi popravio elemente koji izazivaju defekte u proizvodu.**

Kvantifikacija analize defekata podrazumeva uvođenje mera, koje omogućavaju upoređivanje različitih defakata, njihovo klasifikovanje i druge operacije, kao i kod merenja dužine ili težine, koji su nam mnogo bliži.

Svrha ove lekcije je da ukaže da otkrivanje grešaka u ranim fazama životnog ciklusa softverskog proizvoda dosta manje koštaju od onih koje se otkriju u kasnijim fazama. Da bi softverski proizvod zadovoljio potrebe korisnika on treba da zadovolji opšte standarde kvaliteta softverskog proizvoda. Organizacije koje razvijaju softver žele da zadovolje potrebe svojih korisnika i na taj način ostanu profitabilne. Da bi softverski proizvod zadovoljio standarde kvaliteta on mora biti detaljno testiran.

✓ Poglavlje 1

Greške u testiranju softvera

KADA SE POJAVLJUJU GREŠKE U TESTIRANJU SOFTVERA?

Greška je neformalni naziv nedostataka, što znači da softver ili aplikacija ne rade prema zahtevu.

Ovde ćemo govoriti o greškama (**bug**) u testiranju softvera, zašto se javljaju, osnovnoj terminologiji i alatima za praćenje grešaka.

Šta je greška u testiranju softvera?

Greškaje **neformalni naziv za nedostatak**, koji ukazuje da softver ili aplikacija ne rade prema zahtevu.

U testiranju softvera, softverska greška / bug se može okarakterisati korišćenjem različitih termina kao: problem (**issue**), greška (**error**), nedostatak (**fault**) ili kvar (**failure**) itd.

Do greške dolazi kada programeri naprave bilo kakav propust tokom razvoja proizvoda. Tokom testiranja aplikacije ili izvršavanja test slučajeva, inženjer testa nekada neće dobiti očekivani rezultat prema zahtevu, što ukazuje na postojanje greške.

Za opisivanje grešaka može se koristiti različita terminologija:

- **Defekt (Defect)** – označava situaciju kada aplikacija ne radi prema zahtevima, pronalazi ga test inženjer
- **Bug** – neformalno ime za defekt, pronalazi test inženjer
- **Greška (Error)** – problem u kodu koji dovodi do greške, pronalazi programer, test inženjer ili kupac
- **Problem (Issue)** – kada aplikacija ne zadovoljava poslovne zahteve, pronalazi kupac
- **Greška (Mistake)** – problem koji postoji u nekom dokumentu
- **Kvar (Failure)** – do njega dovodi mnogo defekata

Zašto se javljaju defekti / bug?

U testiranju softvera, greška se može pojaviti iz sledećih razloga:

- **Pogrešnog kodiranja:** označava nepravilnu implementaciju. Na primer: Prepostavimo da koristimo aplikaciju Gmail i kada kliknemo na "Inbox" predemo na stranicu „Draft“, to se može dogoditi zbog pogrešnog koda koji je uradio programer, jer se pojavio bug.

- **Nedostatka dela koda:** znači da programer možda nije razvio kod za određenu funkciju. *Na primer: ako uzmemo gornji primer i kliknemo na "Inbox", videćemo da se ona neće izvršiti, što znači da kod za tu funkciju nije razvijen.*
- **Prekomernog kodiranja:** znači da programeri razvijaju dodatne funkcije, koje prema zahtevima klijenta nisu potrebne. *Na primer: pretpostavimo da imamo formu za prijavu koja u skladu sa zahtevima klijenta treba da ima polja za ime i prezime. Ali, programeri mogu dodati i tekstualno polje „Srednje ime“, koje nije potrebno u skladu sa zahtevima klijenta.*

ALATI ZA PRAĆENJE GREŠAKA: JIRA, BUGZILLA, REDMINE

Neki od alata za praćenje grešaka su: Jira, Bugzilla, Redmine

U testiranju softvera dostupni su različiti tipovi alata za praćenje grešaka koji nam pomažu da što lakše pratimo otkrivene greške. Neki od najčešće korišćenih alata za praćenje grešaka su sledeći:

- **Jira** je jedan od najvažnijih alata za praćenje grešaka. Jira je alat otvorenog koda koji se koristi za praćenje grešaka, upravljanje projektima i praćenje problema u ručnom testiranju. Jira uključuje različite funkcije poput izveštavanja, snimanja i praćenje toka posla. U Jiri možemo pratiti sve vrste grešaka i problema (**bugs, issues**) koji su povezani sa softverom i koje je generisao inženjer testa. Da biste dobili sve detalje o Jira alatu, pogledajte link: <https://www.javatpoint.com/jira-tutorial>
- **Bugzilla** je još jedan važan alat za praćenje grešaka, koji koriste mnoge organizacije. Bugzilla je alat otvorenog koda koji se koristi za pomoć kupcu i klijentu za održavanje evidencije o greškama. Takođe se koristi kao alat za upravljanje testiranjem, jer on omogućava lako povezivanje sa drugim alatima za upravljanje testiranjem kao što su ALM, , quality Centre itd. Bugzilla podržava razne operativne sisteme, kao što su Windows, Linux i Mac. Bugzilla ima neke funkcije koje nam pomažu da lako izveštavamo o greškama:
 - Greška može biti navedena u više formata
 - Moguće je obaveštavanje putem e-pošte pod kontrolom korisničkih preferenci.
 - Sadrži napredne mogućnosti pretraživanja
 - Odlična sigurnost
- **Redmine** je alat otvorenog koda koji se koristi za praćenje problema i alat za upravljanje projektima baziranim na vebu. Alat Redmine je napisan na programskom jeziku Ruby i takođe je kompatibilan sa više baza podataka kao što su MySQL, Microsoft SQL i SQLite. Neke uobičajene karakteristike alata Redmine su sledeće:
 - Fleksibilna kontrola pristupa zasnovana na ulogama
 - Funkcija praćenja vremena
 - Podrška za više jezika (albanski, arapski, holandski, engleski, danski i tako dalje)

ALATI ZA PRAĆENJE GREŠAKA: MANTISBT, BACKLOG

Neki od alata za praćenje grešaka su: MantisBT, Backlog

- **MantisBT** je skraćenica od **Mantis Bug Tracker**. To je veb baziran sistem za praćenje grešaka, a takođe je i alat otvorenog koda. MantisBT se koristi za praćenje softverskih defekata. Izvršava se na PHP programskom jeziku. Neke od uobičajenih karakteristika MantisBT-a su sledeće:
 - Pretraživanje celog teksta
 - Integracija sa sistemom za kontrolu revizije
 - Kontrola revizije tekstualnih polja i napomena
 - Notifikacije
 - Grafičko predstavljanje odnosa između problema
- **Backlog** se široko koristi za upravljanje IT projektima i praćenje grešaka. Uglavnom je napravljen za prijavljivanje grešaka za razvojni tim sa kompletним detaljima problema i komentarima. To je softver za upravljanje projektima. Karakteristike Backlog alata su sledeće:
 - Izrada Gantt dijagrama
 - Podržava Git i SVN repozitorijume
 - IP kontrola pristupa
 - Podržava iOS i Android aplikacije

NIVOI GREŠAKA

Kvantifikacija analize grešaka podrazumeva uvođenja mera, koje omogućavaju upoređivanje različitih defekata, njihovo klasifikovanje i druge operacije.

Razvoj računarske tehnike i povećanje kompleksnosti softvera su sve više praćeni tendencijom uvođenja kvantitativnih metoda u obezbeđenje kvaliteta softvera. Ove metode se sastoje pretežno u sledećem:

1. Informacije o greškama se prikupljaju i klasificuju.
2. Za svaku grešku se identificuje mogući uzrok (npr. neusklađenost sa specifikacijama, greška u dizajnu, loša komunikacija s korisnikom ...)
3. Koristeći Pareto princip (80% defekata je izazvano sa 20% uzroka) izolovati dominantne uzroke
4. Nakon izolovanja malog broja najznačajnijih uzroka preduzeti mere za njihovo otklanjanje. Ovaj relativno prosti koncept predstavlja važan korak ka izgradnji adaptivnog softverskog inženjerskog procesa koji se menja da bi popravio elemente koji izazivaju defekte u proizvodu.

Kvantifikacija analize grešaka podrazumeva uvođenja mera, koje omogućavaju upoređivanje različitih grešaka, njihovo klasifikovanje i druge operacije, kao i kod merenja dužine ili težine, koji su nam mnogo bliži.

U tu svrhu se uvodi rangiranje grešaka, tj dodeljivanje broja zavisno od posledica koje mogu da nastanu. Uobičajena su četiri nivoa grešaka:

- **Rang 1 -greška ima najveći prioritet:** Otkaz može biti kritičan (gubitak života ljudi ili ogromni poslovni gubici), rizik je visok, nema mnogo podataka za praćenje
- **Rang 2 -greška ima veliki prioritet:** Otkaz ima neprihvatljive posledice, rizik je neizvestan
- **Rang 3 -greška ima srednji prioritet:** Otkaz ima posledice koje se preživljavaju, rizik je umeren
- **Rang 4 -greška ima mali prioritet:** Otkaz ima zanemarljive posledice, rizik je nije bitan

Postoji više načina da se analiziraju merenja softvera poznata i kao softverska metrika.

U ovoj lekciji će biti izloženi najjednostavniji metodi merenja i upravljanja kvalitetom, koji su baš zbog jednostavnosti i efikasnosti stekli popularnost u drugim oblastima primene, a uspešno se mogu primeniti i u softverskom inženjerstvu.

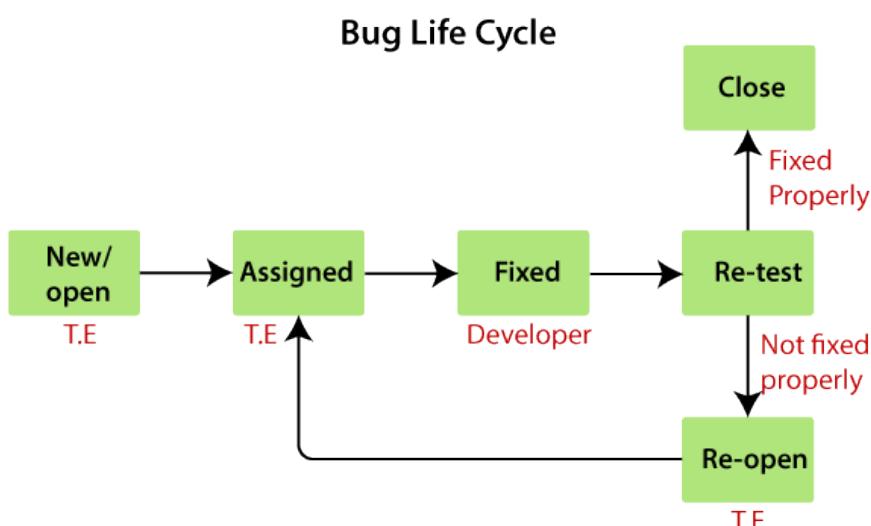
✓ Poglavlje 2

Životni ciklus grešaka

OD ČEGA SE SASTOJI ŽIVOTNI CIKLUS GREŠAKA?

Od faza kada je pronađena, otklonjena, ponovo testirana i zatvorena.

Ovde ćemo govoriti o životnom ciklusu grešaka i različitim statusima grešaka kao i obrascu (**template**) za izveštavanje o greškama. Govorimo o kompletном životnom ciklusu greške od faze kada je pronađena, otklonjena, ponovo testirana i zatvorena.



Slika 2.1 Životni ciklus grešaka Izvor: <https://www.javatpoint.com/software-testing-bug-life-cycle> [Izvor: NM SE321-2020/2021.]

Čim inženjer testa pronađe grešku, dodeljuje joj se status "nova" (**new/open**), što znači da je greška upravo pronađena. Novu grešku treba prijaviti dotičnom programeru koji treba da se kao odgovorna osoba pobrine za grešku kada njen status treba promeniti u "dodeljena" (**assigned**),

Tada programer prvo analizira grešku, što znači da čita sve korake za navigaciju kako bi odlučio da li je stvarno u pitanju greška ili nije. Na osnovu ovoga, ako je greška validna, programer započinje reprodukciju greške u aplikaciji, a nakon što se greška uspešno reproducuje, programer će analizirati kod, izvršiti potrebne promene i promeniti status na "otklonjena" (**fixed**).

Kada se izvrši promena koda i greška se ispravi, inženjer testa ponovo testira grešku, što znači da inženjer testa ponovo izvršava istu radnju koja je navedena u izveštaju o grešci i u skladu sa tim menja status:

- "zatvorena" (**closed**) - ako je greška ispravljena i funkcija radi u skladu sa zahtevom
- "ponovo otvorena" (**re-open**) - ako greška i dalje postoji ili ne radi ispravno u skladu sa zahtevom, onda će greška biti ponovo poslata programeru.

Ovaj proces se kontinuirano nastavlja sve dok se sve greške ne isprave i zatvore.

STATUSI GREŠAKA: NEVAŽEĆE / ODBIJENE GREŠKE

Kada je inženjer testa napisao pogrešan izveštaj o grešci zbog nerazumevanja zahteva, programer neće prihvati grešku, daće joj status "nevažeća" i izveštaj poslati nazad.

Kada pripremimo izveštaj o grešci i pošaljemo ga programeru, programer će prihvati grešku i početi da vrši potrebne promene koda što predstavlja pozitivan tok životnog ciklusa greške.

Može postojati nekoliko situacija kada programeri možda neće izvršiti potrebne promene koda, što predstavlja negativan tok ili status životnog ciklusa greške.

Različiti statusi životnog ciklusa grešaka su:

- Nevažeće / odbijene (**Invalid/rejected**)
- Duplikat (**Duplicate**)
- Odložena (**Postpone/deferred**)
- Ne može se popraviti (**Can't fix**)
- Nije je moguće ponoviti (**Not reproducible**)
- Zahtev za poboljšanje (**Request for Enhancement - RFE**)

Nevažeće / odbijene greške: Kada je inženjer testa napisao pogrešan izveštaj o grešci zbog nerazumevanja zahteva, programer neće prihvati grešku, daće joj status "nevažeća" i izveštaj poslati nazad. Ovaj status greške se može pojaviti i ukoliko program nije dobro razumeo zahteve. Svaka greška koju programer ne prihvati, poznata je kao nevažeća greška.

STATUSI GREŠAKA: DUPLIKAT

Kada različiti inženjeri testa više puta prijave istu grešku, javiće se dupla greška. Razlozi za pojavu statusa greške duplikat su sledeći: zajedničke funkcije i

Duplikat: Kada različiti inženjeri testa više puta prijave istu grešku, javiće se dupla greška. Razlozi za pojavu statusa greške duplikat su sledeći:

- **Zajedničke funkcije:**

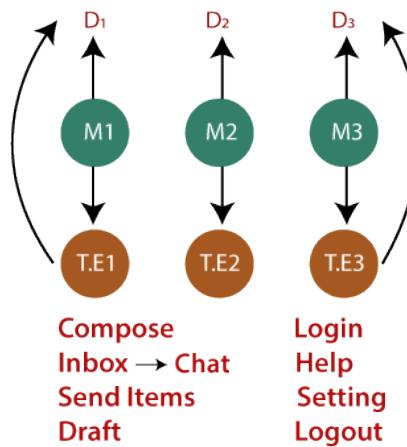
Na primer: Pretpostavimo da imamo inženjere za testiranje P i K koji testiraju jednu od funkcija softvera poput prijave na aplikaciju. Inženjer testa P unosi važeće korisničko ime i lozinku i kliknite na dugme za prijavu, nakon čega će se otvoriti prazna stranica, što znači da je reč o grešci. Nakon toga, P priprema izveštaj o grešci za nađenu grešku

i šalje ga programeru. Tada se inženjer testa K takođe prijavljuje na aplikaciju i dobija istu grešku. K takođe priprema izveštaj o grešci i šalje ga programeru. Kada programer dobije oba izveštaja o greškama inženjera, on / ona šalje izveštaj o grešci K-u i kaže da je duplikat.

- **Zavisni moduli**

Kao što vidimo na slici 2, inženjer testa (T.E.1) želi da sastavi poštu (**compose**), ali pre toga treba da se prijavi (**login**). Ako je grešku za prijavljivanje već pronašao inženjer testa koji je testirao taj modul (T.E.3) može se desiti da istu grešku prijavi (T.E.1) jer modul za sastavljanje mail-a zavisi od modula za prijavljivanje.

Kako izbeći dupliranu grešku?



Slika 2.2 Mogućnost pojave duple greške Izvor: <https://www.javatpoint.com/software-testing-bug-life-cycle> [Izvor: NM SE321-2020/2021.]

Ako je programer naišao na dupliranu grešku, otići će u repozitorijum grešaka i potražiti grešku, tj. proveriti da li greška postoji ili ne.

- Ako postoji ista greška, tada je nije potrebno ponovo prijaviti u izveštaju ili
- Ako greška ne postoji, prijavite je i sačuvajte u repozitorijumu grešaka. Zatim pošaljite izveštaj programerima kao i inženjerima za testiranje dodajući ih u [CC].

STATUSI GREŠAKA:GREŠKU NIJE MOGUĆE PONOVITI

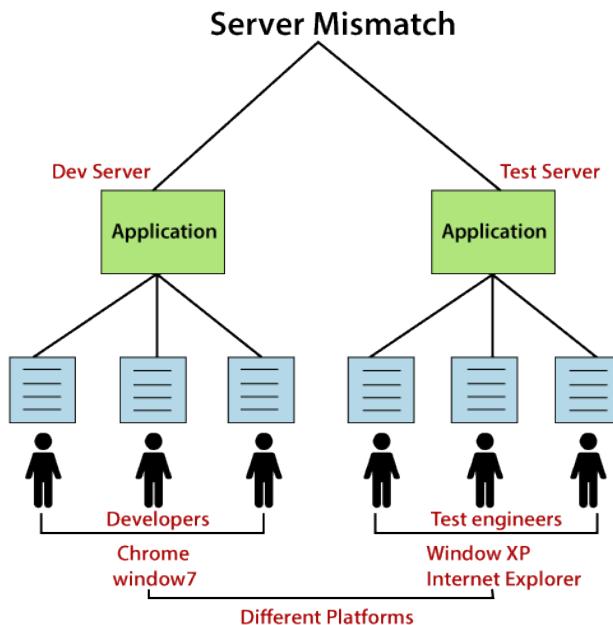
To je greška koju programer ne može da pronađe prolaskom kroz korake koji je inženjer testa dao

Programer prihvata grešku, ali iz nekih razloga nije u mogućnosti da je reprodukuje. To je greška koju programer ne može da pronađe prolaskom kroz korake koji je inženjer testa dao u izveštaju o grešci.

Razlozi za status greške koja se ne može ponoviti su sledeći:

- **Nepotpuni izveštaj o grešci:** Inženjer testa nije naveo sve korake navigacije u izveštaju.
- **Nepodudarnost okruženja:** Nepodudarnost okruženja se može opisati na dva načina:

- **Nepodudaranje servera:** Test inženjer koristi jedan server (Test Server), a programer drugi (Development Server) za reprodukciju greške kao što možemo videti na slici 3:
- **Neusaglašenost platforme:** Test inženjer koristi jednu platformu (npr. Window 7 i Google Chrome), a programer drugu platformu (Window XP i Internet Explorer).
- **Nepodudaranje podataka:** Tokom testiranja inženjer testa i programer koristi različite vrednosti. Na administrator i korisnik mogu koristiti različite vrednost za isti modul za prijavljivanje. Npr. test engineer koristi: User name → abc
Password → 123, a admin koristi: User name → aaa
Password → 111
- **Nekonzistentnost greške:** Bug se ponekad pojavi, a nekada se to neće dogoditi.



Slika 2.3 Nepodudarnost servera koja može dovesti do greške koju nije moguće ponoviti Izvor: <https://www.javatpoint.com/software-testing-bug-life-cycle> [Izvor: NM SE321-2020/2021.]

- **Neusklađenost bildova:** Inženjer testa će pronaći grešku u jednoj verziji, a programer reprodukuje istu grešku u drugoj verziji. Greška se može automatski ispraviti ispravljanjem druge greške.

Rešenje za greške koje nije moguće ponoviti: Čim pronađete grešku, napravite snimak ekrana, tako da programer može ponovo potvrditi grešku i ispraviti je ako postoji.

STATUSI GREŠAKA:GREŠKU NIJE MOGUĆE ISPRAVITI ILI SE ISPRAVKA GREŠKE ODLAŽE

Grešku nekada nije moguće ispraviti zbog postojanja nekih ograničenja ili se greška može odložiti za buduće rilize zbog vremenskih ograničenja.

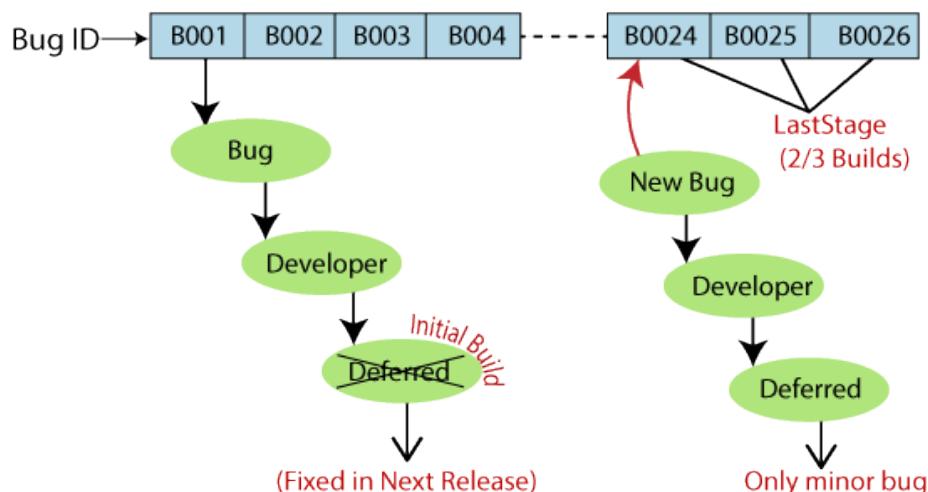
Slučaj kada programer prihvati grešku i može da je reprodukuje, ali ne može da izvrši potrebne promene zbog nekih ograničenja.

Slede ograničenja ili razlozi zbog kojih nije moguće ispraviti grešku:

- **Nepostojanje tehničke podrške:** U programskom jeziku koji smo koristili nije moguće da se reši problem.
- **Greška je u osnovnikoda (okvira):** Ako je greška manja (nije važna i ne utiče na aplikaciju), vođa programera saopštava da će se ona ispraviti u sledećem rilizu. Ali ako je greška kritična (regularno se koristi i važna je za poslovanje), vođa programera ne može da je odbaci.
- **Troškovi popravke greške su veće od zadržavanja.**

Greške mogu imati i status koji označava da se greške odlažu na buduće rilize zbog vremenskih ograničenja.

Kao što vidimo na slici 4: Greška ID-B001 je pronađena u početnoj verziji, ali neće biti ispravljena u istom bildu već će se odložiti i popraviti u sledećem rilizu.



Slika 2.4 Primeri grešaka koje se odlažu za sledeći bild i ispravljaju u istom bildu Izvor: <https://www.javatpoint.com/software-testing-bug-life-cycle> [Izvor: NM SE321-2020/2021.]

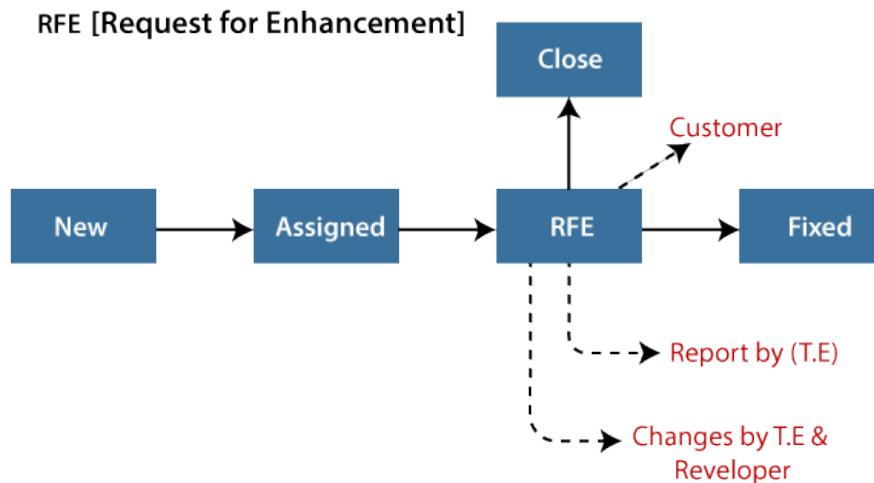
Bug ID - B0024, B0025 i B0026 su greške koje su pronađene u poslednjem bildu, i one će biti otklonjene jer se radi o manjim greškama.

STATUSI GREŠAKA: ZAHTEV ZA POBOLJŠANJE

Ovo su predlozi inženjera za poboljšanje aplikacije u obliku izveštaja o greškama.

Ovo su predlozi inženjera za poboljšanje aplikacije u obliku izveštaja o greškama. RFE je skraćenica od Zahtev za poboljšanje ([Request for Enhancement](#)).

Kao što možemo videti na primeru slike 5, inženjer testa smatra da izgled aplikacije ili softvera nije dobar jer testira aplikaciju kao krajnji korisnik i menja status na RFE. A ako kupac kaže "da", onda bi status trebao biti ispravljena ([fixed](#)) ili ako kupac kaže 'ne', onda bi status trebao biti zatvorena ([closed](#)).



Slika 2.5 Zahteva za poboljšanje Izvor: <https://www.javatpoint.com/software-testing-bug-life-cycle>
[Izvor: NM SE321-2020/2021.]

KOME DODELITI GREŠKU?

Greška se može dodeliti: programerima, vođi programera ili vođi test inženjera

Greška se može dodeliti:

- **programerima:** u slučaju da znamo ko je razvio modul u kome se pojavila greška
- **vođi programera:** ako ne pozajemo programera koji je razvio određeni modul.
- **vođi testiranja:** kada nemamo nikakvu interakciju sa razvojnim timom.

Kada je greška ispravljena i zatvorena ali ima bilo kakav uticaj na drugi modul, iniciramo novi izveštaj o grešci. Takođe, kada je status greške "ponovo otvorena" (nije popravljena) i utiče na drugi modul, moramo da pripremimo novi izveštaj o grešci.

OBRAZAC ZA IZVEŠTAVANJE O GREŠKAMA

Primer jednog obrasca za izveštavanje o greškama sa opisima elemenata koji sadrži

Na slici 6. je dat primer jednog obrasca za izveštavanje o greškama. Neke važni atributi izveštaja o greškama su:

- **ID greške (Bug ID):** jedinstveni broj koji se dodeljuje greški.
- **Naziv slučaja testiranja (Test Case Name):** Kada pronađemo grešku, šaljemo izveštaj o grešci, a ne slučaj testiranja dotičnom programeru. Koristi se kao referenca za inženjera testa.
- **Ozbiljnost (Severity):** To je uticaj greške na aplikaciju. Može biti blokator, kritična, glavna i sporedna.
- **Prioritet:** Na osnovu prioriteta odlučujemo koju grešku moramo prvo otkloniti. Može biti P1 / P2 / P3 / P4, hitan, visok, srednji i nizak.
- **Status:** Statusi greške mogu biti dodeljena, pogrešna, duplikat, odložena itd.
- **Izvestilac:** ime osobe koja je pronašla grešku. To bi mogao biti inženjer testa, a nekada programer, poslovni analitičar, kupac itd.
- **Datum:** datum pronalaska greške.
- **Verzija bild-a / riliza:** broj riliza u kome se javlja greška, kao i verzija bild-a aplikacije.
- **Platforma:** Detalji platforme, gde tačno pronalazimo grešku.
- **Opis:** Ovde ćemo objasniti korake navigacije, očekivane i stvarne rezultate određene greške.
- **Prilozi:** Priložite snimke ekrana greške koju ste pronašli jer pomaže programerima da vide grešku.

Bug Report Template	
Bug ID	
Module	
Requirements	
Test Case Name	
Release	
Version	
Status	
Reporter	
Date	
Assign To	
CC	
Severity	
priority	
Server	
Platform	
Build No.	
Test Data	
Attachment	
Brief Description	
Navigation Steps	
Observation	
Expected Result	
Actual Result	
Additional Comments	

Slika 2.6 Obrazac za izveštavanje o greškama <https://www.javatpoint.com/software-testing-bug-life-cycle> [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 3

SQA model

ŠTA OBJAŠNJAVA SQA MODEL?

Greške otklonjene u ranoj fazi manje koštaju od onih koje se otkriju kasnije.

Da bi softverski proizvod zadovoljio potrebe korisnika, on treba da zadovolji opšte kvalitete softverskog proizvoda. Organizacije koje razvijaju softver žele da zadovolje potrebe svojih korisnika i na taj način ostanu profitabilne. Da bi softverski proizvod zadovoljio standarde kvaliteta on mora biti detaljno testiran. Postoje preporuke kako da organizacije uspešno i na vreme isporuče softver, uz minimalne troškove. Dat je SQA model (Software Quality Assurance) koji objašnjava kako greške otklonjene u ranoj fazi manje koštaju od onih koje se otkriju kasnije.

Otkrivanje grešaka u ranim fazama životnog ciklusa softverskog proizvoda dosta manje košta od onih koje se otkriju u kasnijim fazama.

U tekstu se daje preporuka kako da organizacije koje se bave razvojem softvera uspešno i na vreme isporuče softver uz minimalne troškove.

Pojam testiranja (pronalaženje i uklanjanje grešaka) danas se razlikuje od vremena sedamdesetih godina prošlog veka, kada je to značilo isključivo traženje grešaka. Sada ono obuhvata niz aktivnosti od planiranja, dizajniranja, izgradnje, održavanja, pa sve do sprovođenja testova. Potrebno je definisati načine merenja pojedinačnih faktora kvaliteta i definisati kriterijume njihovog zadovoljenja.

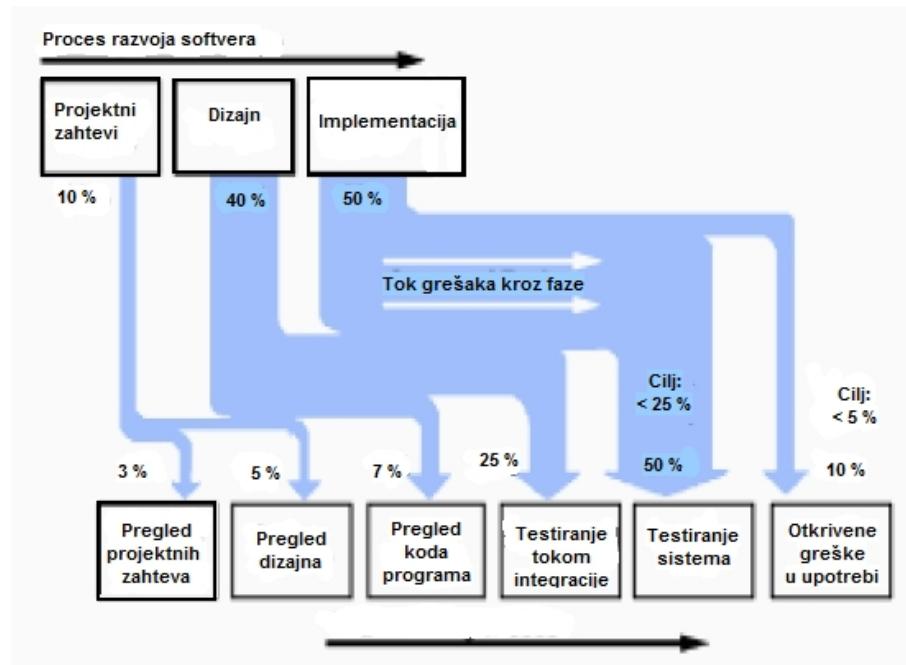
Pronalaženje i uklanjanje grešaka u softveru je složen posao koji obuhvata testiranje jedinca, modula, podsistema, sistema i prijemno testiranje. Ciljevi testiranja su različiti. Ogledaju se kroz demonstraciju-proveru korektnosti dizajna i implementacije softvera, destrukciju-detekciju grešaka na nivou programiranja, evaluaciju-otkrivanju grešaka u specifikaciji zahteva, dizajnu i implementaciji i prevenciju-sprečavanju grešaka kroz fazu razvoja softvera.

POREKLO GREŠAKA I NJIHOVA DISTRIBUCIJA

Greške (u fazi u kojoj su nastale) se distribuiraju tokom razvojnog procesa, od početka projekta do njegovog kraja.

Greške (u fazi u kojoj nastaju) se često distribuiraju tokom razvojnog procesa, od početka projekta do njegovog kraja. Istraživanja koja su sprovedena od strane glavnih razvijatelja softvera, kao što su IBM i TRW, sažeta u radovima Boehm i Jones, otkrivaju slične obrasce

porekla grešaka, njihovu distribuciju tokom životnog ciklusa kao i proces filtriranja (detekcije i otklanjanja) kao što je na slici. 1 prikazano.



Slika 3.1 Proces filtriranja (detekcije i otklanjanja) grešaka [Izvor: NM SE321-2020/2021.]

Razvoj softvera pokazuje da se taj uzorak nije bitno promenio u zadnje dve decenije. Karakteristična distribucija porekla grešaka softvera na temelju pomenutih radova, prikazana je u tabeli 1 na slici 2.

Slika 3.2 Karakteristična distribucija porijekla defekata softvera [Izvor: NM SE321-2020/2021.]

EFIKASNOST UKLANJANJA GREŠAKA

Prepostavka je da bilo koja aktivnost osiguranja kvaliteta filtrira (engl. screens) određeni postotak postojećih nedostataka.

Prepostavka je da bilo koja aktivnost osiguranja kvaliteta filtrira (engl. screens) određeni postotak postojećih nedostataka. Treba napomenuti da je u većini slučajeva, postotak uklonjenih grešaka nešto manji od postotka otkrivenih (detektovanih) nedostataka primenom neke test metode (oko 10% u skladu sa Jones-ovim radovima) tj. neefikasan je ili je neodgovarajući postupak "dibagiranja". Preostale greške, neotkrivene ili nekorektno ispravljene su prošle u naredne faze razvoja.

Aktivnosti osiguranja kvaliteta, koje slede, imaju zadatku da se izbore sa ovim kombinovanim greškama: onim koje su preostale nakon sprovedenih prethodnih aktivnosti osiguranja kvaliteta, zajedno sa "novim" napravljenim greškama u tekućoj fazi razvoja. Prepostavlja se da efikasnost filtriranja grešaka svake aktivnosti osiguranja kvaliteta nije manja od 40% (tj. aktivnost uklanjanja grešaka treba da je najmanje 40% od prispevki defekata).

Tipična prosečna efikasnost filtriranja grešaka različitim aktivnostima osiguranja kvaliteta, po fazama razvoja, utemeljene na radovima Boehm i Jones, navedeni su u Tabeli na Slici 3.

Br.	Aktivnosti osiguranja kvaliteta	Prosečna stopa efikasnosti filtriranja grešaka
1.	Pregled specifikacije zahteva	50%
2.	Ispekcija dizajna	60%
3.	Pregled dizajna	50%
4.	Ispekcija koda	65%
5.	Jedinični test	50%
6.	Jedinični test poslije ispekcije koda	30%
7.	Integracioni test	50%
8.	Sistem test/ test prihvatljivosti	50%
9.	Pregled dokumentacije	50%

Slika 3.3 Prosek efikasnosti filtriranja (uklanjanja grešaka) u aktivnosti osiguranja kvaliteta [Izvor: NM SE321-2020/2021.]

TROŠKOVI UKLANJANJA NEDOSTATAKA U SOFTVERU

Podaci o troškovima projekata, prikupljeni na više razvojnih projekata, pokazuju da trošak otkrivanja i uklanjanja nedostataka varira kroz faze razvoja.

Podaci o troškovima projekata, prikupljeni na više razvojnih projekata, pokazuju da trošak otkrivanja i uklanjanja nedostataka varira kroz faze razvoja, dok je porast tih troškova znatan (eksponencijalan) kako se razvojni proces odvija.

Na primer, uklanjanje defekta u design fazi može zahtevati ulaganje od 2,5 radna dana; uklanjanje istog kvara može zahtevati 40 radnih dana u toku testiranja prilikom isporuke softvera kupcu. Nekoliko anketa sprovedene su od strane IBM-a, TRW, GTE, Boehm i drugih, za procenu relativnih troškova za ispravljanje grešaka u svakoj fazi razvoja.

Iako su podaci o statistici i troškovima uklanjanja grešaka prilično retki, stručnjaci se slažu da je proporcionalno povećanje troškova uklanjanja grešaka ostala konstantna na bazi sprovedenih anketa u 1970-tim i 1980-tim.

Model je zasnovan na sledećim pretpostavkama:

- Razvoj procesa je linearan i sekvenčnajan, prati poznati vodopad model razvoja softvera.
- Broj "novih" grešaka se uvodi (pravi) u svakoj fazi razvoja.
- Pregled i testiranje programa, kao aktivnosti osiguranja kvaliteta, služe kao filteri uklanjajući određeni postotak ulaznih defekata i ostavljajući deo neotkrivenih da budu otkriveni u sledećoj razvojnoj fazi. Na primer, ako je broj dolaznih grešaka 30, a efikasnost filtriranja je 60%, tada će biti 18 grešaka uklonjeno, a 12 grešaka će ostati i proći da bude otkrivene u sledećoj aktivnosti osiguranja kvaliteta.
- U svakoj fazi, sabiraju se dolazni broj grešaka koji nije uklonjen u prethodnoj aktivnosti osiguranja kvaliteta zajedno s "novim" greškama uvedenim (stvorenim) u tekućoj fazi razvoja.

- Troškovi uklanjanja defekata se izračunavaju za svaku aktivnost osiguranja kvaliteta množenjem broja uklonjenih nedostatka sa prosečnom cenom (troškom) uklanjanja jednog defekta u toj fazi (vidi tabelu na Slici 4).
- Preostale defekte, nažalost, koji su dospeli do kupca, otkriće korisnik po znatno većoj jediničnoj ceni

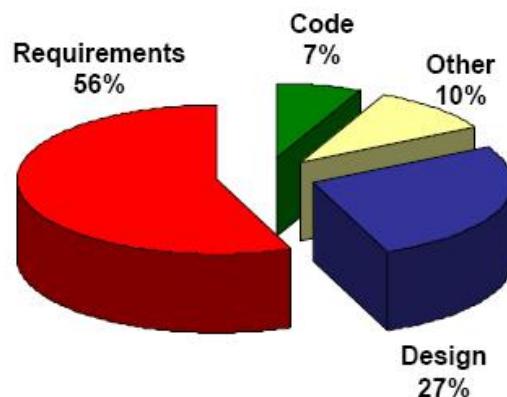
Br.	Faze razvoja softvera	Prosečna relativna cena otklanjanja greške (jedinica cene - cu)
1.	Specifikacija zahteva	1
2.	Dizajn	2.5
3.	Jedinični test	6.5
4.	Integracioni test	16
5.	Sistem test/ test prihvatljivosti/ pregled dokumetacije	40
6.	Operativna upotreba (nakon isporuke kupcu)	110

Slika 3.4 Prosečni relativni troškovi uklanjanja grešaka [Izvor: NM SE321-2020/2021.]

TIPIČNA RASPODELA MESTA NASTANKA I TROŠKOVI UKLANJANJA GREŠAKA

Tipična raspodela mesta nastanka i troškovi uklanjanja grešaka je prikazana na slikama

Tipična raspodela mesta nastanka i troškovi uklanjanja grešaka je prikazana na slikama 5 i 6.



Slika 3.5 Tipična raspodela mesta nastanka grešaka u SDLC [Izvor: NM SE321-2020/2021.]

Slika 3.6 Tipična raspodela troškova uklanjanja grešaka u SDLC [Izvor: NM SE321-2020/2021.]

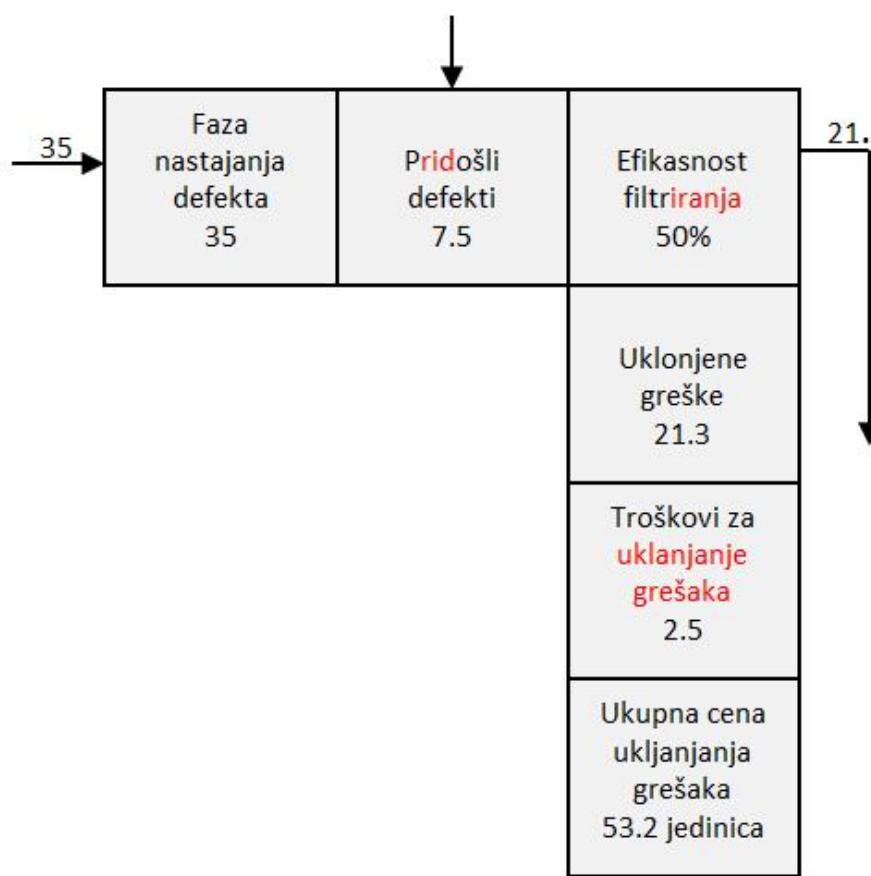
MODEL AKTIVNOSTI OSIGURANJA KVALITETA U SDLC

U modelu, svaka od aktivnosti osiguranja kvaliteta predstavlja jedan stepen filtera.

U modelu, svaka od aktivnosti osiguranja kvaliteta predstavlja jedan stepen filtera, kao što je prikazano na slici 7.

Model predstavljaju sledeće veličine:

- POD = Faza nastajanja Defekta (iz Tabele na Slici 2)
- PD = Prošli (prebegli) Defekti (iz bivše faze ili aktivnosti osiguranja kvaliteta)
- % FE = % Efikasnosti Filtriranja (također nazvan % „screening“ efikasnost iz Tabele na Slici 3)
- RD = Uklonjani Defekti
- CDR = Troškovi Uklanjanja jedne greške (iz Tabele na Slici 4)
- TRC = Totali troškovi uklanjanja: TRC = RD × CDR.

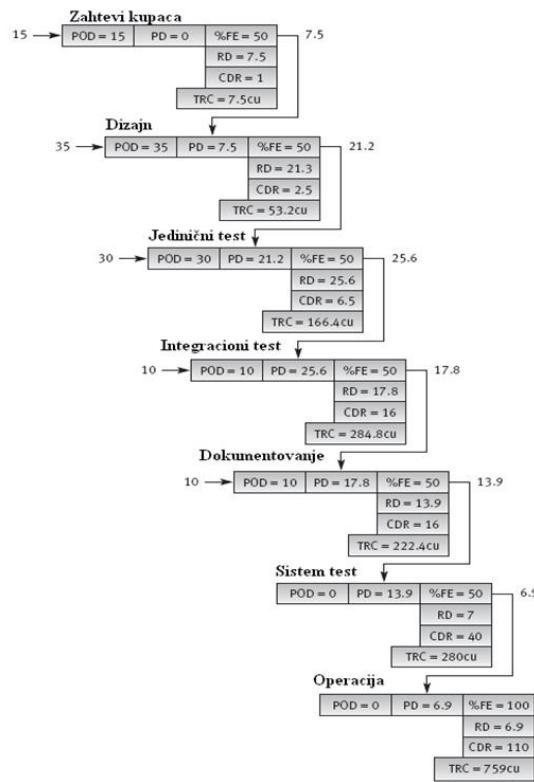


Slika 3.7 Model aktivnosti osiguranja kvaliteta u SDLC [Izvor: NM SE321-2020/2021.]

STANDARNI I SVEOBUVATAN PLAN OSIGURANJA KVALITETA

Sveobuhvatni plan, u celini, je mnogo ekonomičniji od standardnog plana jer štedi 41% od ukupnih sredstava uloženih u uklanjanje grešaka u poređenju sa standardnim planom.

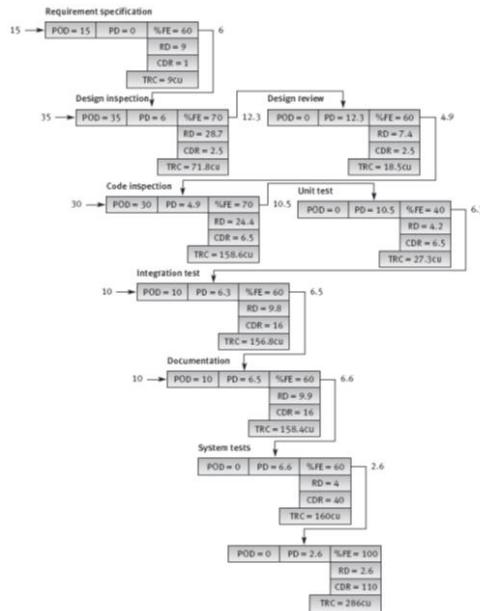
Standardni model - plan procesa uklanjanja grešaka prikazan je na slici 8.



Slika 3.8 Standardni model - plan procesa uklanjanje 100 grešaka [Izvor: NM SE321-2020/2021.]

Sveobuhvatan plan osiguranja kvaliteta (slika 9) za razliku od standardnog ("sveobuhvatno filtriranje grešaka sistema") postiže sledeće:

1. dodaju se dve aktivnosti osiguranja kvaliteta, tako da se oba izvode u dizajn fazi kao i u fazi kodiranja.
2. poboljšava efikasnost "filtriranja" drugih aktivnosti osiguranja kvaliteta.



Slika 3.9 Sveobuhvatni model - plan procesa uklanjanje 100 grešaka [Izvor: NM SE321-2020/2021.]

POREĐENJE STANDARNOG I SVEOBUHVATNOG PLANА OSIGURANJA KVALITETA

Standardni plan uspešno uklanja samo 57,6% (28,8 grešaka iz 50) od grešaka koji je nastao u fazi zahteva i dizajna, dok je za sveobuhvatni plan efikasnost 90,2%, pre početka kodiranja.

Nakon toga je izvršeno upoređivanje rezultata standardnog procesa obezbeđenja kvaliteta softvera naspram sveobuhvatnog plana otkrivanja grešaka u softveru.

Najvažniji zaključci izvučeni iz upoređivanja su:

1. **standardni plan uspešno uklanja samo 57,6% (28,8 grešaka iz 50) od grešaka koji je nastao u fazi zahteva i dizajna, dok je za sveobuhvatni plan efikasnost 90,2%, pre početka kodiranja.** Ovo se očekuje kao direktna posledica intenzivnijeg postupka uklanjanje greška - povećani su naporovi testiranja koji sveobuhvatni plan uključuje.
2. **sveobuhvatni plan, u celini, je mnogo ekonomičniji od standardnog plana jer štedi 41% od ukupnih sredstava uloženih u uklanjanje greške u poređenju sa standardnim planom.**
3. **U odnosu na standardni plan, sveobuhvatni plan doprinosi većem zadovoljstvu kupaca uz drastično se smanjuje stope otkrivenih grešaka tokom redovne operativne upotrebe softvera (od 6,9% na 2,6%).**

▼ Poglavlje 4

Statički SQA alati

ALATI ZA POBOLJŠANJE KVALITETA

Izbor i korišćenje alata kvaliteta zavise od dve vrste primenjenih metoda: Statične i Dinamične metode automatizacije kontrole kvaliteta.

Alati za poboljšanje kvaliteta su koncepti, tehnike, metode, studije, sredstva, odnosno, uopšteno govoreći, svi naporci usmereni ka poboljšanju kvaliteta (Pareto dijagram, regresiona analiza, tehnika kontrolnih karti, metode uzorkovanja i prihvatanja, studija preciznosti, tačnosti i stabilnosti procesa i dr.), i primenjuju se u sistemu kvaliteta, u okviru aktivnosti poboljšanja kvaliteta kao integralnog dela upravljanja kvalitetom.

Primena metodologije i principa upravljanja poboljšanjem kvaliteta podrazumeva korišćenje alata kvaliteta. Izbor i korišćenje alata kvaliteta zavise od dve vrste primenjenih metoda:

- Statične metode automatizacije kontrole kvaliteta
- Dinamične metode automatizacije kontrole kvaliteta.

Uloga i značaj alata kvaliteta u aktivnostima poboljšanja kvaliteta istaknuti su u standardu ISO 9004-4, uz napomenu da će primena bilo kog alata kvaliteta dati izvesno poboljšanje kvaliteta.

Razvoj alata kvaliteta (tehnika inženjerstva kvaliteta) započeo sa prvim elementima primene statističke teorije na polju inspekcije, da bi se danas došlo do desetina različitih tehniki, što je detaljno razmatrano i u nastavku ove lekcije.

Prema nekim autorima, alati kvaliteta svrstani su u sledećih 14 kategorija: alati grupne dinamike, statistički alati, menadžment alati, implementacioni alati, alati procesa, alati znanja, napredni statistički alati, sistemski alati, alati za upravljanje događajima, alati kupaca, alati rukovodstva, inovacioni alati, softver alati, i alati organizacionog ponašanja.

Metodologija za poboljšanje kvaliteta podrazumeva primenu korektivnih ili preventivnih mera, pa se, prema tome, alati kvaliteta mogu podeliti prema karakteru dejstva - na korektivne i preventivne. Ovi alati se pretežno primenjuju za Statične metode automatizacije kontrole kvaliteta.

Alati kvaliteta korektivnog dejstva su obrazac za prikupljanje podataka, kontrolne karte, metode uzorkovanja i prihvatanja, analiza i obrada podataka, studija preciznosti, tačnosti i stabilnosti, metode kumulativnih vrednosti, i ulazna, procesna i izlazna inspekcija, dok ostali relevantni alati kvaliteta imaju preventivno dejstvo.

ŠTA SPADA U STATIČKE SQA ALATE?

Tu spadaju Čekliste (eng. checklist or check sheet), Pareto dijagram (Pareto diagram), Histogram (histogram), Dijagram rasipanja (scatter diagram) i drugi.

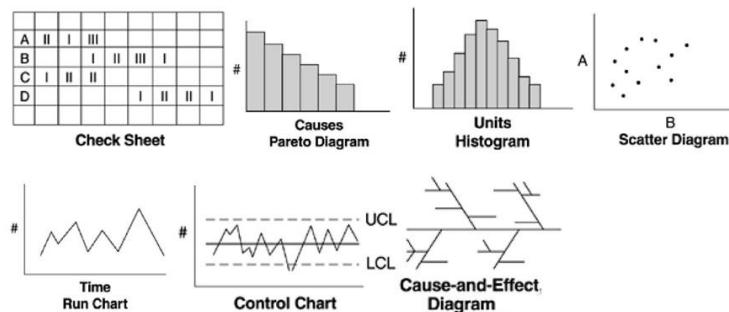
Osnovne metode kontrole kvaliteta je promovisao Ishikawa (1989) i one se mogu primeniti u procesu razvoja softvera. Ove metode su namenjene pre svega rukovođenju projektom razvoja, ali programerima neće biti od velike koristi u poboljšanju kvaliteta dizajna i implementacije.

Metode o kojima je reč su veoma efikasne u upravljanju kvalitetom procesa proizvodnje. No, razvoj softvera je mnogo složeniji proces, tako da je moguće da ove metode ne budu dovoljno efikasne i da je potrebna primena složenijih i sofisticiranih alata kao što je npr. CASE (Computer Aided Software Engineering).

Osnovnih sedam metoda kontrole kvaliteta su prikazane na Slici 1.

U statične metode automatizacije kontrole kvaliteta spadaju:

1. Čekliste (eng. checklist or check sheet),
2. Pareto dijagram (Pareto diagram),
3. Histogram (histogram),
4. Dijagram rasipanja (scatter diagram) i drugi.



Slika 4.1 Išikavinih sedam osnovnih metoda [Izvor: NM SE321-2020/2021.]

Dinamične metode automatizacije kontrole kvaliteta se odnose na proces i kvalitet proizvoda.

ČEKLISTA

To je ustvari struktuirani formular pripremljen za prikupljanje podataka i njihovu analizu. U procesu razvoja softvera se čekliste mogu koristiti u svim fazama.

Čeklista (eng. checklist or check sheet) se takođe naziva i **dijagram koncentracije defekata**. To je u stvari struktuirani formular pripremljen za prikupljanje podataka i njihovu

analizu. Primer čekliste za registrovanje prekida telefona je dat na slici 2. Čekliste se obično koriste:

- Kada podatke prikuplja jedna osoba ili se prikupljaju s jedne lokacije,
- Kada se prikupljaju podaci o defektima, problemima, učestanostima, uzrocima.

U procesu razvoja softvera se čekliste mogu koristiti u svim fazama. Jedan specifični primer su čekliste u procesu prevencije defekata (**defect prevention process DPP**) koji se sastoji iz tri koraka:

1. *Analize defekata u vidu otkrivanja uzroka*
2. *Sugestija preventivnih akcija u cilju otklanjanja uzroka defekata*
3. *Implementacija preventivnih akcija*

Telefonski prekidi

Razlog	Dan					
	Ponedeljak	Utorak	Sreda	Četvrtak	Petak	Ukupno
Pogrešan broj	5	2	1	5	7	20
Info-zahtev	2	2	2	2	2	10
Šef	5	2	7	1	4	19
Ukupno	12	6	10	8	13	49

Slika 4.2 Primer čekliste [Izvor: NM SE321-2020/2021.]

Drugi primer su PTF liste koje IBM primenjuje u PTF (**program temporary fix**) – privremenom rešenju defekta otkrivenog u programu koji je već u eksploataciji.

PARETO DIJAGRAM

Ekonomista Vilfred Pareto je uočio "da većina efekata (80%) dolazi od relativno malog broja uzroka (20%), što je poznato kao Paretov princip.

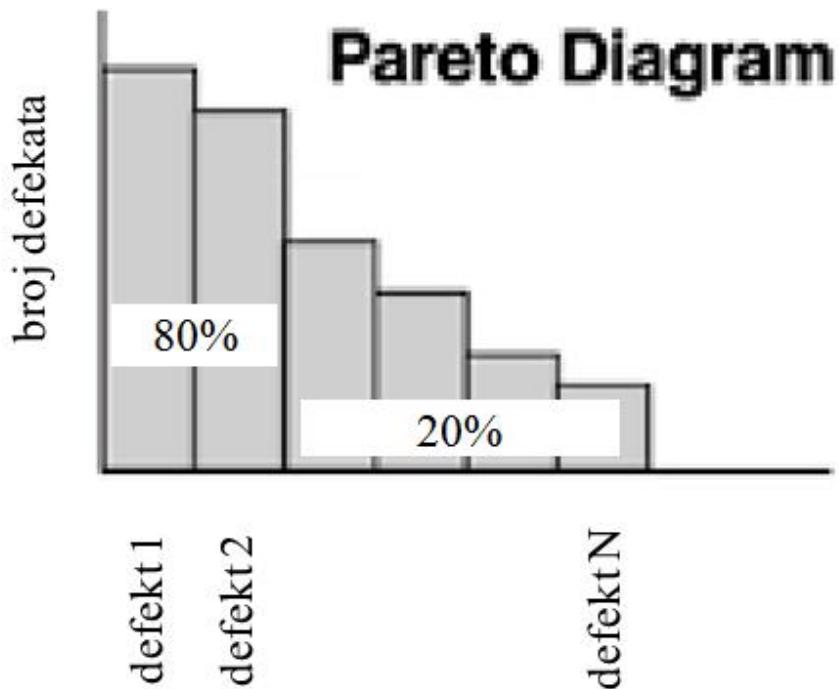
Pareto dijagram je grafička prezentacija defekata koji su rangirani po značaju, počevši od najznačajnijeg. Dobio je ime po ekonomisti Vilfredu Paretu koji je uočio "da većina efekata (80%) dolazi od relativno malog broja uzroka (20%), što je poznato kao Paretov princip.

Na apscisi su nanešeni pokazatelji - defekti ili uzroci (ili događaji), a na ordinati njihov broj ili frekvencu (ili neka od mera defekata, kao npr: troškovi, gubici). Obično se predstavljaju dva grafika, prvi koji predstavlja broj pojave datog defekta, a drugi kumulativni broj defekata (tj. sumu pojave datog i prethodnih defekata na apscisi).

Pareto grafici se koriste:

- **Pri analizi učestanosti pojave defekata u procesu**
- **Pri analizi koji je od defekata značajniji**

Pareto grafika je prikazana na sledećoj slici.



Slika 4.3 Primer pareto grafika [Izvor: NM SE321-2020/2021.]

ČEKLISTE - PROCEDURA ZA PRIMENU

U primeni čeklisti se može primeniti priložena procedura

Procedura za primenu je sledeća:

- Izabrati događaj (ili problem) koji se istražuje. Uvesti definicije pojmove za primenu.
- Izabrati gde se podaci prikupljaju i koliko dugo.
- Napraviti formular tako da podaci mogu da se prikupljaju što jednostavnije, stavljanjem crtica, znaka x ili slično.
- Označiti sva polja na formularu
- Svaki put kada dolazi do datog događaja, upisati podatke na formular
- Testirati listu tokom probnog perioda da se utvrdi da li se prikupljaju adekvatni podaci.

Primer čekliste defekta dat je na slici 4.

Defect Check Sheet:

Quality Improvement Project: _____

Defect / Defect Cause of Interest: _____

Instructions to Recorders on When to Record a Defect: _____

Type of modules or services being observed for defects: _____

Setting: _____

Time Frame for Data Collection: _____

Data Recorders: _____

Defect or Defect Cause	Count

Slika 4.4 Primer čekliste defekta [Izvor: NM SE321-2020/2021.]

PARETO DIJAGRAM - PROCEDURA ZA PRIMENU

U primeni Pareto dijagrama se može primeniti priložena procedura

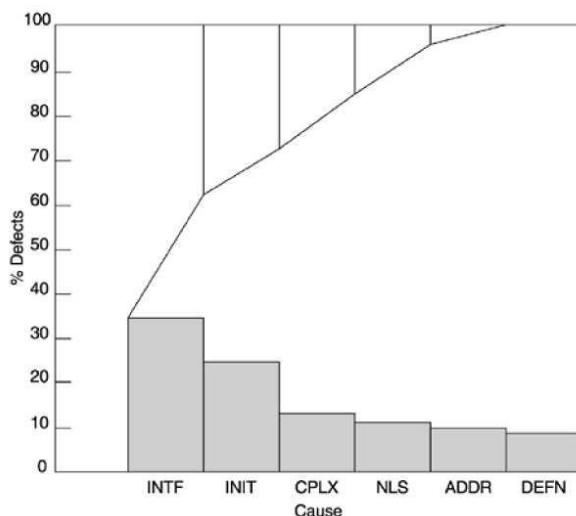
Procedura konstrukcije:

1. Izaberi kategorije (defekte) koje će biti posmatrane
2. Izaberi pokazatelje - mere za pojedine kategorije (najčešće su to učestanost-broj pojava, količina, cena, vreme)
3. Izaberi vremenski period koji se posmatra (nedelja, mesec, ceo projekat ...)
4. Prikupi podatke za grafik (ili uzmi već postojeće)
5. Odredi pokazatelje za svaku od kategorija. Veoma je pogodno izraziti pokazatelje u relativnim vrednostima, npr. procentima. U tu svrhu treba svaku vrednost pokazatelja podeliti ukupnom sumom vrednosti pokazatelja.
6. Odredi kumulativnu vrednost pokazatelja.
7. Nacrtaj Pareto grafik

Na slici 5 je prikazan primer Pareto grafika na bazi analize defekata i uzroka za jedan IBM proizvod.

Pokazalo se da su:

- problemi interfejsa (INTF) i
- problemi inicijalizacije podataka (INIT) dominantni uzroci defekata za taj proizvod.



Slika 4.5 Paretoov grafik za IBM-ov proizvod [Izvor: NM SE321-2020/2021.]

Nakon toga je obrađena veća pažnja na ove oblasti tokom dizajna implementacije i testiranja softvera. Uz to su izvršene konsultacije sa ekspertima, i kao rezultat su ostvarena poboljšanja.

Drugi uzroci defekata (ili oblasti gde se defekti javljaju) su:

- Kompleksni logički problemi (CPLX)
- Problemi usled drugog jezika (ne engleskog) (NLS)
- Problemi adresiranja (ADDR)
- Problemi definisanja podataka (DEFN)

HISTOGRAM

Histogram je grafička predstava koliko puta neki događaj pri višestrukom ponavljanju pripada određenim kategorijama.

Histogram je grafička predstava koliko puta neki događaj pri višestrukom ponavljanju pripada određenim kategorijama. Tipična je predstava učestanosti po kategorijama događaja (koliko puta se koja kategorija javlja).

U specijalnom, no veoma čestom slučaju, histogram predstavlja raspodelu - koliko puta određena promenljiva uzima konkretne vrednosti, odnosno pada u interval vrednosti.

Histogram se koristi u slučajevima:

- **Kada su podaci numerički**
- **Kada je od interesa distribucija veličine koja se analizira**
- **Kada se želi porebiti dat proces sa drugim procesom**

Konstrukcija histograma:

- Prikupiti podatke iz najmanje 50 uzastopnih događaja posmatranog procesa
- Odredite elemente na apscisi dijagrama.

- Izabrati broj intervala B i širinu W. Vrši se procena prema preporuci u tabeli na sledećoj slici:

Broj podataka N	50			100		150		200
Broj intervala (B)	7	8	9	10	11	12	13	14

Slika 4.6 Izbor broja intervala B i širine W histograma [Izvor: NM SE321-2020/2021.]

U tabeli je: B = Broj intervala;

Opseg podataka:

- R = najveća vrednost - najmanja vrednost
- Širina intervala: $W = R / B$ (zaokrugliti)
- Granice intervala: (L₁, L₂, ..., L_B) ;

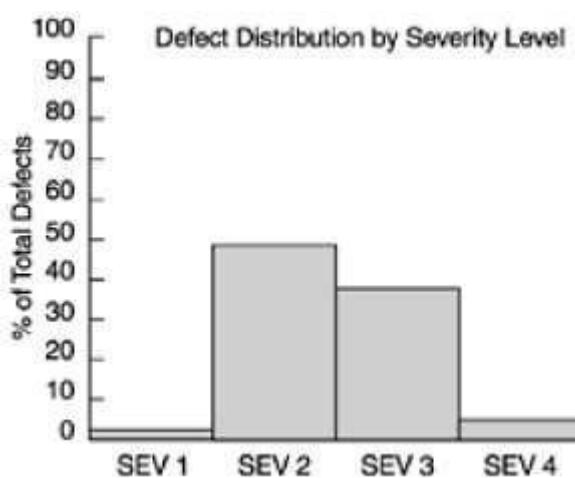
Izabrati L₁: Zatim je L₂ = L₁ + W, L₃ = L₂ + W,

Nacrtati x-y grafik. Označite na x-osi vrednosti (L₁, L₂, ..., L_B). Za svaki interval nanesite na y osu odgovarajući broj događaja.

PRIMERI HISTOGRAMA

Priložena su dva primera korišćenja histograma

Na slici 7. su prikazani primeri histograma koji su korišćeni u menadžmentu i upravljanju kvalitetom.

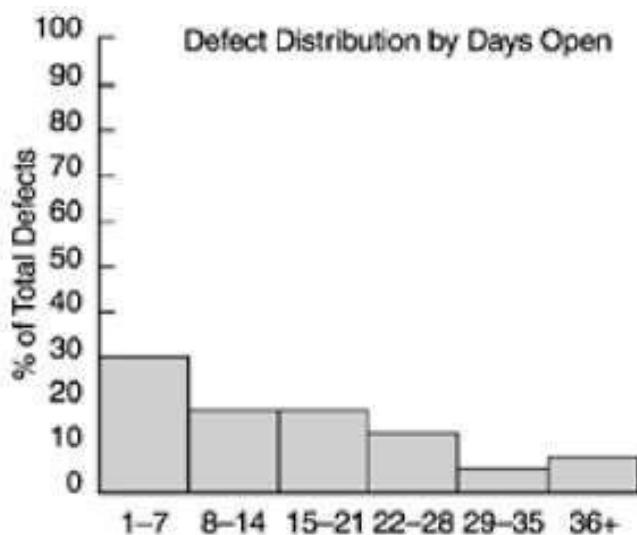


Slika 4.7 Raspodela defekata po nivou kritičnosti [Izvor: NM SE321-2020/2021.]

Grafik prikazuje učestanost defekata po nivoima kritičnosti (od 1 do 4, pri čemu je nivo 1 najkritičniji):

- Nivo 1: Kritični defekti – Posledice mogu biti gubitak ljudskih života ili ogromni poslovni gubici firmi
 - Nivo 2: Ozbiljni defekti – posledice mogu biti ozbiljne povrede lica ili veliki poslovni gubici
 - Nivo 3: Umereni defekti – posledice mogu biti povrede lica ili poslovni gubici, ali bez trajnih posledica
 - Nivo 4: Trivijalni defekti – posledice nisu ozbiljne ili ih nema.
- Očigledno je da raspodela broja defekta po nivoima predstavlja značajan pokazatelj kvaliteta softvera.

Sledeći histogram na slici 8. prikazuje učestanost defekata (broj defekata po intervalu vremena) tokom testiranja po danima koliko je bilo potrebno za njihovo otklanjanje (1 do 7 dana, 8 do 14 itd). Histogram pokazuje vreme reagovanja na otkaze, odnosno efikasnost tima.



Slika 4.8 Raspodela defekata po broju dana do otklanjanja [Izvor: NM SE321-2020/2021.]

✓ Poglavlje 5

Dinamički SQL alati

ŠTA SPADA U DINAMIČKE SQL ALATE?

U dinamičke SQL alate spadaju: Vremenski dijagram, Dijagram rasipanja, Tabela trenda testa, Dijagram kontrole

Dinamične metode automatizacije kontrole kvaliteta se odnose na proces i kvalitet proizvoda pri čemu:

- Na kvalitet razvijenog proizvoda utiče kvalitet produpcionog procesa
- Posebno je bitan u razvoju softvera, jer su neki atributi kvaliteta proizvoda teški za procenu
- Ipak, postoji veoma kompleksna i loše razumljiva veza između softverskog procesa i kvaliteta proizvoda.

U dinamične metode automatizacije kontrole kvaliteta spadaju: metode vezane za proces merenja, skupljanje i analizu podataka.

Podatke treba odmah skupljati i ako je moguće automatski. Postoje tri tipa automatskog prikupljanja podataka:

1. **Statička analiza proizvoda**
2. **Dinamička analiza proizvoda**
3. **Proces svrstavanja podataka**

Jedna od dinamičkih metoda u procesu svrstavanja podataka je:

1. **Raspodela defekata po broju dana do otklanjanja,**
2. **Vremenski dijagram (run chart), kao i drugi tip dijagrama, koje je veoma sličan prethodnom je tzv S-kriva i drugi.**
3. **Metode vezane za metrike proizvoda.**

U dinamičke SQL alate spadaju:

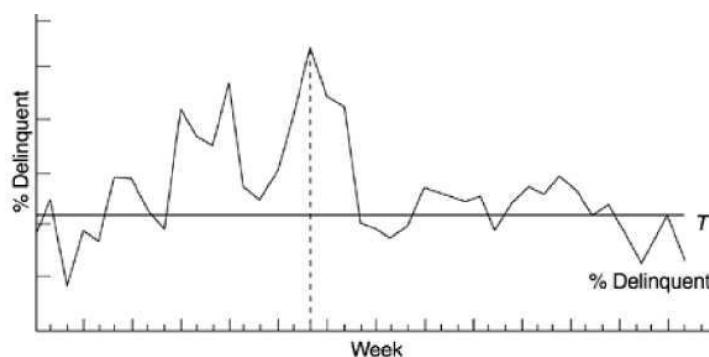
1. **Vremenski dijagram**
2. **Dijagram rasipanja**
3. **Tabela trenda testa**
4. **Dijagram kontrole**

VREMENSKI DIJAGRAM

Vremenski dijagram (run chart) prikazuje promenu veličine od interesa u vremenu.

Vremenski dijagram (run chart) prikazuje promenu veličine od interesa u vremenu. Kao primer, otkrivanje defekata tokom nedelje se može prikazati grafikom. Njime se postiže praćenje i nivoa kvaliteta (y osa) i dinamika u vremenu (x osa).

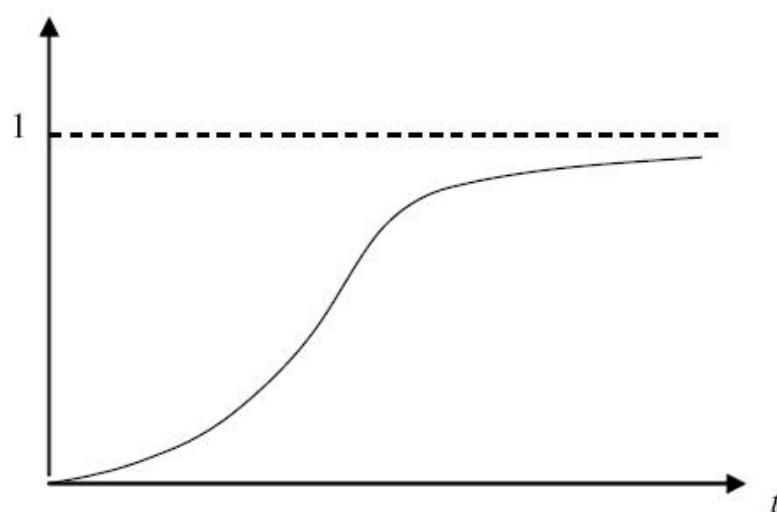
Na slici 1. je prikazan broj (u %) neotklonjenih defekata IBM-ovog proizvoda. Horizontalna linija (označena sa T) označava maksimalni dozvoljeni broj defekata po jedinici vremena. Isprekidana vertikalna linija označava trenutak kad je otpočela korektivna akcija za smanjenje broja defekata. Svaki izveštaj o defektu je analiziran pažljivo i preduzete su odgovarajuće akcije. Kao rezultat, kroz mesec dana je nivo defekta pao ispod kritične vrednosti.



Slika 5.1 Raspodela defekata po broju dana do otklanjanja [Izvor: NM SE321-2020/2021.]

Drugi tip dijagrama, koje je veoma sličan prethodnom je tzv S-kriva. Ovaj dijagram, prikazan na slici 2, u datom trenutku prikazuje sumu svih defekata koji su se desili od početka posmatranja.

S-krive se koriste u IBM-u za praćenje kvaliteta procesa razvoja softvera.



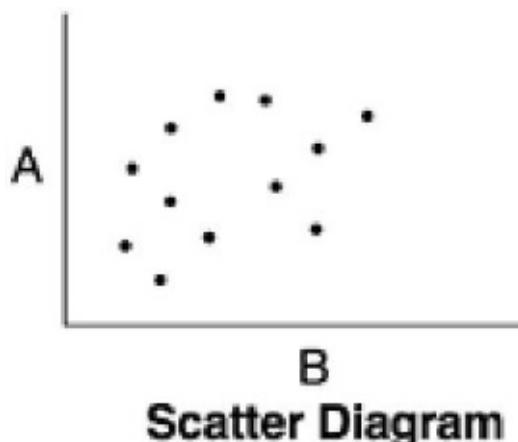
Slika 5.2 S- krive [Izvor: NM SE321-2020/2021.]

DIJAGRAM RASIPANJA

Ovaj dijagram prikazuje parove dveju veličina u x-y ravni, radi analize uzajamne relacije.

Dijagram rasipanja prikazuje parove dveju veličina u x-y ravni, radi analize uzajamne relacije. Ako su veličine korelisane, dobiće se tačke koje se grupišu oko linije ili krive. Što je korelacija bolja, to je rasturanje u odnosu na krivu manje (Slika 3). Dijagram rasipanja se koristi kada se želi utvrditi da li su dve veličine zavisne kao što je:

- Pri pokušaju utvrđivanja uzroka defekata
- Kada postoje numerički podaci
- Nakon konstrukcije dijagrama uzrok posledica, da bi se objektivno utvrdilo da li je dijagram korektan
- Pri utvrđivanju da li veličine (događaji) koje izgledaju zavisne, obe imaju isti uzrok.
- Pri proveri auto korelacije



Slika 5.3 Dijagram rasipanja [Izvor: NM SE321-2020/2021.]

Procedura konstrukcije i analize dijagrama rasipanja:

1. Prikupiti podatke o dve veličine koje treba analizirati, tako da svakom događaju odgovara po jedan par datih veličina.
2. Nacrtajte grafik nanoseći za svaki par jednu veličinu na x osu, a drugu na y osu.
3. Ako grafik podseća na pravilnu krivu ili pravu liniju, bez mnogo "testerisanja", možete stati. Našli ste zavisnost dveju veličina. U protivnom, nastavite postupak.
4. Podelite grafik na četiri kvadranta. Neka je X tačaka na grafiku. Izbrojte X/2 tačaka odozgo nadole i povucite horizontalnu liniju. Izbrojte X/2 tačaka s leva udesno i povucite vertikalnu liniju. Ako je broj tačaka neparan, povucite liniju kroz srednju tačku.
5. Izbrojte tačke u svakom kvadrantu. Ne računajte tačku na liniji.
6. Saberite broj tačaka diagonalnih kvadrantata. Nađite manju od sume:
 - A = tačke gore-levo + tačke dole-desno;
 - B = tačke gore-desno + tačke dole-levo;
 - Q = manja od A i B;

- $N = A + B$
7. Odredite limit za tabelu trenda.
- Ako je Q manje od limit-a, dve varijable su zavisne
 - Ako je Q veće ili jednako Limit-u, varijable nisu zavisne i forma na grafiku je potpuno slučajna.

TABELA TRENTA TESTA

Tabela trenda testa se koristi ta izradu Dijagrama trenda testa.

Tabela trenda testa je prikazana na slici 4.

N	1-8	9-11	12-14	15-16	17-19	20-22	23-24	25-27	28-29	30-32	33-34	35-36	37-39
Limit	0	1	2	3	4	5	6	7	8	9	10	11	12

N	40-41	42-43	44-46	47-48	49-50	51-53	54-55	56-57	58-60	61-62	63-64	65-66	67-69
Limit	13	14	15	16	17	18	19	20	21	22	23	24	25

N	70-71	72-73	74-76	77-78	79-80	81-82	83-85	86-87	88-89	90			
Limit	26	27	28	29	30	31	32	33	34	35			

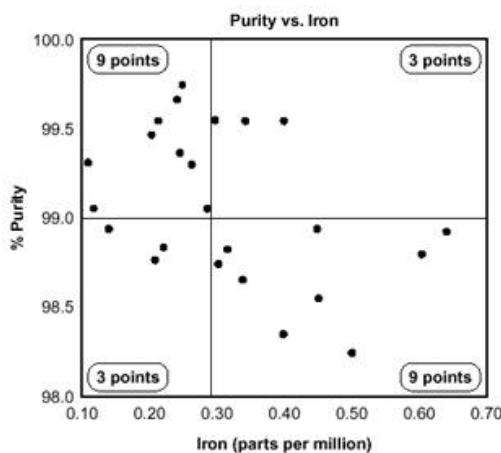
Slika 5.4 Tabela trenda testa [Izvor: NM SE321-2020/2021.]

Primer: Na dijagramu (Slika 5) ima 24 tačaka. Primenjujući proceduru, računamo:

- $A = \text{tačke gore-levo} + \text{tačke dole-desno} = 9 + 9 = 18$
- $B = \text{tačke gore-desno} + \text{tačke dole-levo} = 3 + 3 = 6$
- $Q = \text{manja od } A \text{ i } B = \text{manja od } 18 \text{ i } 6 = 6$
- $N = A + B = 18 + 6 = 24$

Sada tražimo Limit za dato N iz tabele trenda. Za $N = 24$ Limit = 6. Q = Limit, prema tome, ne postoji zavisnost veličina na dijagramu rasipanja.

Razmotrimo primenu dijagrama rasipanja na razvoj softvera. Na slici 6. je prikazana relacija nivoa defekata i McCabe-ovog indeksa kompleksnosti. Svaka tačka predstavlja jedan modul čija je x koordinata indeks kompleksnosti, dok je na y osi nivo defekata datog modula.



Slika 5.5 Dijagram trenda testa [Izvor: NM SE321-2020/2021.]

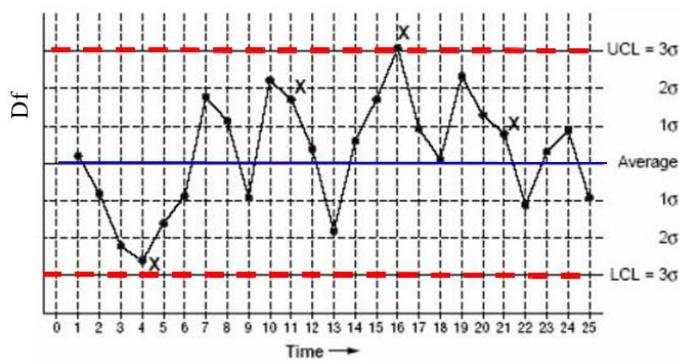
Ovakva zavisnost se može koristiti za predviđanje defekta pri dizajnu novih modula, ili pri redizajnu da bi se ostvarili manji nivoi defekta.

Slika 5.6 Dijagram rasipanja [Izvor: NM SE321-2020/2021.]

DIJAGRAM KONTROLE

Dijagram ili grafik kontrole služi za statističko praćenje procesa u vremenu.

Dijagram ili grafik kontrole služi za statističko praćenje procesa u vremenu (Slika 7). Podaci se crtaju u funkciji vremena. Grafik kontrole uvek ima srednju vrednost (average) i gornju i donju granicu (UCL upper control limit, LCL Lower control limit).



Slika 5.7 Grafik kontrole [Izvor: NM SE321-2020/2021.]

Ove linije su određene na bazi prethodnih merenja procesa. Poređenjem sa ovim linijama se može zaključiti da li je proces kontrolabilan (vrednosti posmatrane veličine su u granicama) ili je van kontrole.

Procedura konstrukcije grafika:

- 1. Izaberite odgovarajuću veličinu procesa za praćenje
- 2. Izaberite odgovarajući period za prikupljanje podataka
- 3. Prikupite podatka, nacrtajte grafik i analizirajte podatke.
- 4. Potražite "signale van kontrole" (van 1σ , 2σ , 3σ). Kada ih nadjete, označite ih i istražite uzrok. Dokumentujte šta ste našli i kako je to korigovano. Signali van kontrolnih granica na slici.
 - Jedna tačka (br 11) je iznad gornje granice LCL.
 - Dve od tri susedne tačke su sa iste strane srednje linije i van 2σ . Na slici su to tačke 3 i 4.
 - Osam susednih tačaka je sa iste strane (iznad) srednje linije. To su tačke 14 do 21
- 5. Nastavite da unosite tačke sve dok pristižu podaci. Za svaku novu tačku proverite da li je van kontrolnih granica
- 6. Kada počnete novi dijagram, proces može biti van kontrolnih granica. U tom slučaju su kontrolne granice sračunate na bazi prvih 20 tačaka uslovne. Kada imate najmanje 20 tačaka od trenutka kad je proces u kontroli, sračunajte nove kontrolne granice

Statistički se definiše sposobnost procesa ([process capability](#)) kao: $Cp = (USL - LSL) / 6\sigma$

Gde je;

- LSL donja propisana granica (Lower Specification Limit)..
- USL gornja propisana granica (upper Specification Limit)
- σ je standardna devijacija procesa
- 6σ je totalna varijacija procesa

▼ Poglavlje 6

Grupna vežba

MERENJE BROJA FUNKCIONALNIH TAČAKA

Prvi i osnovni zadatak za estimiranje troškova, kvaliteta i ostalih parametara softvera je da se pronađe broj funkcionalnih tačaka.

U okviru ove vežbi biće predstavljen način merenja, metrike i ocenjivanja kvaliteta softvera kao i primena Casper Jones-ovih pravila.

Mnogi softverski stručnjaci tvrde da funkcionalnost proizvoda daje bolju sliku o veličini proizvoda nego njegova dužina. U ranoj fazi projektovanja interesantnija je funkcionalnost kada su u pitanju napor i vreme od same fizičke veličine.

Funkcionalne tačke (FP) mere količinu funkcionalnosti sistema opisanog specifikacijom. Da bi se izračunale funkcionalne tačke potrebno je identifikovati računljive diskretne komponente sledećeg tipa:

- Eksterni ulazi -procesni podaci ili upravljačke informacije koje dolaze izvan
- Eksterni izlazi - procesni podaci i upravljačke informacije koje idu van
- Eksterni upiti - interaktivni upiti koji zahtevaju izlaz
- Eksterni interfejs datoteke - mašinski čitljive za druge sisteme
Interne matične datoteke - logičke interne datoteke

Albrecht je funkcionalne tačke FP računao po formuli :

Broj FP= broj ulaza x 4 + broj izlaza x 5 + broj upita x 4 + broj matičnih datoteka x 10 + broj interfejsa x 7

BAZNI ELEMENT	TEŽINSKI FAKTOR		
	JEDNOSTAVAN	SREDNJI	KOMPLEKSAN
ULAZI	3	4	6
IZLAZI	4	5	7
UPITI	3	4	6
EKSTERNI FAJLOVI	7	10	15
INTERNE DATOTEKE	5	7	10

Slika 6.1.1 Tabela Faktori za podešavanje FP-a [Izvor: NM SE321-2020/2021.]

sa tim da težinski faktori mogu varirati +/- 25% zavisno od kompleksnosti programa.
Pomoću gornje formule se dobiju nepodešene funkcionalne tačke (UFC).

Podešene funkcionalne tačke se dobiju množenjem UFC sa faktorom tehničke kompleksnosti TCF koji ima 14 doprinosećih faktora (on-line ulazi, on-line ažuriranje, kompleksnost interfejsa, ponovna upotreba, komuniciranje sa podacima itd.) i varira od 0.65 do 1.35.

NAMENA FUNKCIONALNIH TAČAKA

Prednosti i nedostaci funkcionalnih tačaka predstavljeni su u vidu stavki.

Glavna namena i prednosti funkcionalnih tačaka su:

- merenje veličine proizvoda koja se koristi u predviđanju napora i troškova u ranoj fazi životnog ciklusa softvera (specifikacija, projektovanje)
- predviđanje broja linija izvornog koda
- konverzija veličine projekta pisanog u jednom jeziku u veličinu projekta pisanog u drugom jeziku jer su FP nezavisne od jezika
- merenje produktivnosti projekata koji su pisani u više jezika
- normalizacija u odnosu na FP (defekti/FP, čovek mesece/FP itd.)

Nedostaci funkcionalnih tačaka su:

- teškoća računanja jer različiti ljudi mogu da različito računaju FP i potrebna je vrlo detaljna specifikacija
- za razliku od LOC-a ne mogu se automatski računati
- neke empirijske studije sugerisu da FP nisu vrlo dobre za predviđanje napora i da su nepotrebno kompleksne
- problem sa subjektivitetom tehnološkog faktora
- problem sa aplikacionim domenom (komercijalne aplikacije, aplikacije u realnom vremenu, naučne aplikacije)

PRIMENA 12 CASPER JONES-OVIH PRAVILA - OD 1 DO 5

U nastavku data su prva četiri Casper Jones-ova pravila.

Sa merenjem broja funkcionalnih tačaka vezuju se i Casper Jones-ovi pravila kojih ima 9.
Pravilo 1 - (Estimacija veličine izvornog koda): tiče se izračunavanja funkcionalnih tačaka i pretvaranja u linije koda. To je pravilo demonstrirano u prethodnom poglavlju.

$L = 160 \times 53 = 8480$ LOC
ako koristimo C++ ili Javu.

Pravilo 2 – Estimacija dokumentacije: Ovo pravilo nam omogućava da estimiramo svu dokumentaciju vezanu za softverski paket, uključujući specifikaciju i uputstva.

$FP(\text{broj podešenih funkcionalnih tačaka}) = 160$

Dokumentacija = FP 1.15 stranica

Dokumentacija = $160 \cdot 1.15 = 342$ stranice

Pravilo 3- Estimacija odudaranja korisničkih zahteva: Ovo pravilo kaže da prilikom realizacije projekta odudaranje od korisničkih zahteva je u proseku oko 2 % mesečno. Stoga je važno estimirati troškove ovakvog problema koji se pojavljuje.

Pravilo 4- Estimacija broja slučajeva testiranja: Funkcionalne tačke nam takođe omogućavaju da estimiramo broj test slučajeva koji su potrebni da

se dobro testira softverski proizvod. Ta estimacija se radi na sledeći način:

$FP(\text{broj podešenih funkcionalnih tačaka}) = 160$

Broj test slučajeva = FP1.2

Broj test slučajeva = $160 \cdot 1.2 = 441$ test slučajeva

Pravilo 5- Estimacija potencijalnog broja grešaka: Postoji pet velikih vrsta grešaka u razvoju softvera:

1. Greške u zahtevima
2. Greške u dizajnu
3. Greške u kodiranju
4. Greške u dokumentaciji
5. Loše ispravljene greške, tzv. sekundarne greške koje su prouzrokovane ispravljanjem neke druge greške.

Više od polovine svih softverskih grešaka se isprave u prve dve faze.

Način na koji možemo da estimiramo mogućnost defekta u našem softveru je sledeći:

Potencijalni broj grešaka = FP1.25

Potencijalni broj grešaka za OQT BOX komponentu je:

Potencijalni broj grešaka = $160 \cdot 1.25 = 569$ mogućih grešaka

PRIMENA 12 CASPER JONES-OVIH PRAVILA - 6 - 9

Nakon prvih pet pravila, prikazana su pravila šest, sedam, osam i devet.

Pravilo 6 – Estimacija efikasnosti otklanjanja greške: Ovo pravilo nam govori kako je teško pronaći greške prilikom testiranja. Ovo pravilo kaže da svaki korak testiranja pronalazi i otklanja određen broj grešaka u svakoj fazi. Ti brojevi dati su u vidu tabele.

DEFECTS		FOIND IN PHASE %:						
		P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇
PHASE INSERTED :	Requirement	42%	17%	16%	11%	11%	2%	0%
	HL Design	0	52%	26%	4%	14%	4%	0%
	LL Design	0	0	45%	27%	25%	3%	0%
	Code (Unit test)	0	0	0	55%	43%	3%	0%
	Integration/ System test	0	0	0	0	90%	10%	0%
	Acceptance (User test)	0	0	0	0	0	10%	0%
	Operation Post - release	0	0	0	0	0	0	0%
	Defect Removal Efficiency [%] In phase	42.3	27.4	25.4	29.7	26.4	2.9	0.0
		Cumulative Removal Efficiency [%]	13.4	25.9	41.2	70.7	97.1	100.0
		Total DRE [%]						

Slika 6.1.2 Matrica efikasnosti otklanjanja greške [Izvor: NM SE321-2020/2021.]

Pravilo 7- Estimacija efikasnosti organizovanog otklanjanja grešaka: Ovakav način otklanjanje grešaka je skuplji i zahtevniji vremenski ali i efikasniji. Ovo pravilo kaže da svaka inspekcija dizajna će pronaći i otkloniti određen procenat grešaka u tom trenutku. Ti procenti dati su na Slici 2. Matrica efikasnosti otklanjanja greške

Pravilo 8- Estimacija efikasnosti otklanjanja grešaka nakon puštanja softvera u rad: Programeri zaduženi za održavanje softvera mogu ispraviti određen broj grešaka koji zavisi od TMM i CMM nivoa.

Pravilo 9- Estimacija trajanja realizacije projekta: Estimacija trajanja realizacije projekta DM (Development in months) predstavlja jednu od najvažnijih informacija za klijente, project menadžere i softverske programere. Izračunava se na sledeći način:

FP(broj podešenih funkcionalnih tačaka)=160

DM=FP0.4 [KM] kalendarskih meseci DM= FP0.4=1600.4= 7.5 [KM] kalendarskih meseci

PRIMENA 12 CASPER JONES-OVIH PRAVILA - 10 -12

Prikazana su pravila deset, jedanaest i dvanaest

Pravilo 10- Estimacija potrebnih ljudi za realizaciju projekta: Za planiranje projekta je veoma važno koliko vam ljudi treba za realizaciju samog projekta. Ovo pravilo nam pomaže u tome.

FP(broj podešenih funkcionalnih tačaka)=160

Prosečna produktivnost projektanata=150

Broj projektanata= FP/Prosečna produktivnost projektanata Broj projektanata=160 / = 2 projektanta

Pravilo 11- Estimacija ljudi potrebnih za održavanje softvera:

Slično prethodnom pravilu. Izračunava se na sledeći način:

FP(broj podešenih funkcionalnih tačaka)=160

Broj ljudi za održavanje= FP/ prosečna efikasnost održavanja

Broj ljudi za održavanje=FP/ 750= 160 /750 = 1 čovek

Pravilo 12- Estimacija ukupnih napora u realizaciji softverskog projekta: Ovo pravilo predstavlja kombinaciju pravila 9 i 10. Primjenjuje se na sledeći način: Ukupni Napor= Ukupno vreme * broj ljudi= $7.5 \times 2 = 15$ čovek-meseci

COCOMO I COCOMO II MODEL ZA ESTIMACIJU TROŠKOVA

Estimaciju troškova moguće je izvršiti korišćenjem Cocomo i Cocomo II modela.

COCOMO (COnstructive COst MOdel) predstavlja model za procenu cene koštanja softvera koji je razvijen kao otvoreni model od strane Direktora Centra za Softverski inženjeriranje na USC Dr Barry Boehm-a. Polazna osnova u izračunavanju u COCOMO modelu je korišćenje Effort Equation (jednačine ulaganja) koja ima za cilj procenjivanje veličine podatka osoba/mesec koja je neophodna za razvoje projekta. Svi ostali COCOMO rezultati, uključujući i procene sa zahtevima i održavanjem se izvode iz ove veličine. COCOMO proračuni se baziraju na procenjenoj veličini projekta koja se meri brojem linija izvornog koda Source Lines of Code (SLOC). SLOC se definiše kao: jedino linije koda koje se isporučuju kao deo softverskog proizvoda se broje - drajveri za testiranje i drugi softver za podršku se ne računaju; linije koda koji su generisani zaposleni programeri - kod kreiran od strane generatora koda se isključuju; jedan SLOC je jedna logička linija koda; deklaracije se računaju kao SLOC; komentari se ne računaju kao SLOC.

COCOMO II model sadrži i **Scale Drivers** - uticajne faktore koji najznačajnije utiču na vreme i trajanje projekta. Potrebno je proceniti pet uticajnih faktora koji su elementi jednačine ulaganja. Uticajni faktori su: specifičnost, fleksibilnost razvoja, arhitektura / razrešavanje rizika, kohezija tima i zrelost procesa.

COCOMO II, takođe sadrži i 17 uticajnih faktora na cenu. Cenovni uticajni faktori određuju trud koji je neophodan za kompletiranje softverskog projekta. Na primer, ukoliko se razvija softver za kontrolu leta avio kompanije, uticajni faktor Zahtevana Pouzadnost Softvera - Required Software Reliability (RELY) i njegova cena biće postavljeni na vrlo visoku vrednost.

Jednačina ulaganja u COCOMO II modelu procenjuje potreban broj osoba/meseci person/months - PM na osnovu procene veličine projekta merenog u hiljadama SLOC ili KASLOC.

$$\text{Effort} = 2.94 \times \text{EAF} \times (\text{KASLOC})^{\text{E}}$$

Gde je:

- KASLOC - broj adaptiranih linija koda
- EAF - faktor podešavanja ulaganja (Effort Adjustment Factor) koji pripada grupi cenovnih faktora (Cost Drivers)
- E - eksponent dobijen iz pet veličinskih uticajnih faktora (Scale Drivers)

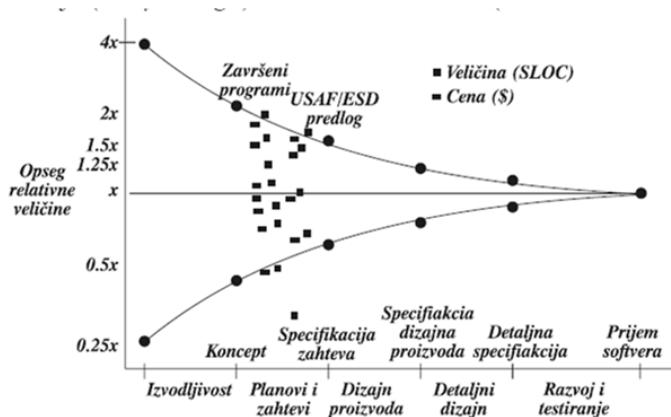
Faktori podešavanja ulaganja su proizvod svih uticajnih faktora.

FAZA RAZVOJA MODELA

U Fazi Early Design modela uključuju se pitanja korišćenja alternativnih softvera/arhitekture sistema i koncepata.

U ovom stadijumu, obično, nema dovoljno poznatih informacija za fino kalibriranje modela. U ovoj fazi COCOMO II softvera koriste se 7 cenovnih uticajnih faktora (i dva Sposobnost zaposlenih – **Personnel Capability** i Iskustvo zaposlenih - **Personnel Expirinece**, dok se u drugoj fazi **Post Arhitecture** modela koriste 6 faktora i različiti aspekti osposobljenosti, iskustva i kvaliteta zaposlenih.)

Post-Architecture model uključuje stvarni razvoj i održavanje softvera. Ova faza daje najbolje rezultate ukoliko razvija arhitekturu softvera, koja odgovara misiji sistema, konceptu funkcionisanja i ukoliko uspostavlja okvir za kompletan prozvod.



Slika 6.1.3 Procena veličine i cene softvera u zavisnosti od faze [Izvor: NM SE321-2020/2021.]

Vreme izvođenja vežbe 45 minuta

✓ 6.1 Individualne vežbe 10

IZRADA AKTIVNOSTI SISTEMSKOG TESTIRANJA SOFTVERA

Na osnovu obrađenih teknika testiranja izraditi aktivnosti sistemskog testiranja softvera mejl sistema ili ISUM sistema.

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

1. Za odabrani softver opisati aktivnosti estimacije prema 12 Capers Jones-ovih pravila. **Vreme izrade vežbe 45 minuta**
2. Izabratи jedan od alata za praćenje grešaka, instalirati ga i proučiti da li je uz njegovu primenu moguće proći kroz čitav životni ciklus praćenja grešaka. **Vreme izrade vežbe 45 minuta**

✓ Poglavlje 7

Domaći zadatak

DESETI DOMAĆI ZADATAK

Nakon desete lekcije potrebno je uraditi deseti domaći zadatak.

Za odabranu aplikaciju koju ste radili na nekom od predmeta koje ste prethodno slušali i položili primeniti sledeće:

Primeniti 12 Casper Jones-ovih pravila i taksativno navesti objašnjenje za svako od pravila (i dobijenih rezultata primjenjenog pravila) na koji način utiče na proces projektovanja softvera.

Poslati tekstualni dokument kao rešenje domaćeg zadatka.

Domaći zadaci treba da budu realizovani u razvojnom okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

Domaći zadatak se imenuje: SE321-DZ10-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ10.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Zaključak

MERENJE I PREDIKCIJA KVALITETA SOFTVERA, GRAFICI KONTROLE

Grafici kontrole se mogu koristiti radi praćenja stabilnosti procesa a i radi provere da li preduzete mere imaju efekta.

Testiranje je jedna od najvažnijih aktivnosti u razvoju softvera i troši između 30 i 50% od ukupnih troškova razvoja softvera u skladu s mnogim studijama. Da bi omogućili dizajneru softvera da postigne veći kvalitet dizajna, bolji uvid u proces obezbeđenja kvaliteta kroz poboljšanje planova testiranja, ponuđen je odgovarajući SQA (Software Quality Assurance) model. U ovom radu opisan je predloženi SQA model koji omogućuje smanjivanje troškova u planu testiranja, kada trenutni plan (odabir tehnika verifikacije, validacije i kvalifikacije) ne može ispuniti zadati kvalitet softvera, u zadatom vremenu za testiranje pri čemu se troškovi uklanjanja grešaka svode na minimum.

Radi kvantifikovanja defekata se uvode četiri nivoa.

- **Nivo 1 Kritični defekti** – posledice mogu biti gubitak ljudskih života ili ogromni poslovni gubici firmi
- **Nivo 2 Ozbiljni defekti** – posledice mogu biti ozbiljne povrede lica ili veliki poslovni gubici
- **Nivo 3 Umereni defekti** – posledice mogu biti povrede lica ili poslovni gubici, ali bez trajnih posledica
- **Nivo 4 Trivijalni defekti** – posledice nisu ozbiljne ili ih nema.

Opisano je više metoda koje se mogu koristiti svakodnevno: **Čeklista, Pareto grafik, grafik uzrok-posledica i dijagram rasturanja (scatter)**, koje se mogu zajedno koristiti da se identifikuju dominantni faktori i njihovi uzroci.

Grafici kontrole se mogu koristiti radi praćenja stabilnosti procesa, a i radi provere da li preduzete mere imaju efekta. Sedam osnovnih metoda su zaista osnovne. No, ako se adekvatno integrišu u dizajn proces, njihov učinak može biti veliki, kao što je pokazano na primerima iz IBM-a. Na kraju, statistička kontrola kvaliteta (statistical process control SPC) nije ograničena samo na crtanje dijagrama. Međutim, metode koje su upravo izložene su prihvaćene u literaturi i u praksi, i postaju nezaobilazne pri razvoju i primeni novih metoda, kao što su CASE (Computer Aided Software Engineering).

LITERATURA ZA LEKCIJU 10

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. Anne Mette Jonassen Hass, Guide to Advanced Software Testing, © 2008 ARTECH HOUSE, INC.

Dopunska literatura:

1. Lj. Lazić , Software Quality & Testing Metrics, WSEAS 7th WSEAS EUROPEAN COMPUTING CONFERENCE (ECC '13), Dubrovnik, Croatia, June 25-27, 2013 . (link: <http://www.wseas.org/wseas/cms.action?id=4102>

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.pisa.rs>
4. <https://www.javatpoint.com/bug-in-software-testing>
5. <https://www.javatpoint.com/software-testing-bug-life-cycle>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Osnovno o održavanju softvera,
vrste i kategorije održavanja SW

Lekcija 11

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 11

OSNOVNO O ODRŽAVANJU SOFTVERA, VRSTE I KATEGORIJE ODRŽAVANJA SW

- ✓ Osnovno o održavanju softvera, vrste i kategorije održavanja SW
- ✓ Poglavlje 1: Definicije i terminologija održavanja
- ✓ Poglavlje 2: Kandidati za redizajn softvera
- ✓ Poglavlje 3: Vrste i kategorije održavanja SW
- ✓ Poglavlje 4: Adaptivno, perfektivno, preventivno održavanje
- ✓ Poglavlje 5: Ključni aspekti softverskog održavanja
- ✓ Poglavlje 6: Grupna vežba
- ✓ Poglavlje 7: Domaći zadatak
- ✓ Vrste i kategorije održavanja SW

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

VRSTE I KATEGORIJE ODRŽAVANJA SW

Postoji više vrsta aktivnosti koje nazivamo održavanjem: Održavanje u smislu ispravke softverskih grešaka; Održavanje u smislu prilagođavanja softvera različitim operativnim okruženjima itd.

U ovoj lekciji razmatraćemo deo procesa softverskog inženjeringu koji se odvija pošto je softver isporučen. Možda će vas iznenaditi činjenica da za mnoge dugotrajne sisteme cena održavanja softvera iznosi više od 75% ukupne cene softvera tokom njegovog veka trajanja.

Mada je održavanje jedan neprekidan proces, ozbiljno razmatranje mora, eventualno, biti dato redizajniranju softverskog sistema. Glavna briga je kako odrediti da li je sistem nepovratno otkazao ili se može uspešno opraviti i održavati. Grupe za testiranje koje rade sa softverom sve vreme, treba da posmatraju opšte stanje softvera i kada bude neophodno, preporuče da je softver bolje redizajnirati nego ga nastaviti korektivno održavati. Troškovi i povoljnosti nastavka održavanja softvera kada postane sklon greškama, neefikasan i skup, moraju biti vagani sa onima za redizajniranje sistema čak i kada redizajniranje nije najbolje rešenje.

Postoji više vrsta aktivnosti koje nazivamo održavanjem: **Održavanje u smislu ispravke softverskih grešaka; Održavanje u smislu prilagođavanja softvera različitim operativnim okruženjima i Održavanje u smislu dodavanja i modifikacija sistemskih funkcionalnosti.**

Adaptivno održavanje je proces menjanja softvera u skladu s novim trendovima na tržištu. Očekivane promene hardvera, sistema, interoperabilnosti aplikacionih paketa i novih karakteristika nametnutih od strane konkurenциje su važni u određivanju potrebe za adaptivnim održavanjem softvera.

Veći broj ključnih tema mora se razmotriti radi osiguranja efikasnog održavanja softvera. Važno je razumeti da softversko održavanje pruža jedinstvene tehničke i menadžerske izazove softverskim inženjerima.

✓ Poglavlje 1

Definicije i terminologija održavanja

STANDARD ZA ODRŽAVANJE SOFTVERA ISO/IEC/IEEE 14764 (TERMINI, ZNAČAJ, AKTIVNOSTI ODRŽAVANJA SOFTVERA...)

Održavanje softvera je integralni deo životnog ciklusa softvera. Ono obuhvata sve aktivnosti koje se preduzimaju da bi se obezbedila efikasna i ekonomski prihvatljiva podrška korisnicima SW.

Održavanje softvera je integralni deo životnog ciklusa softvera. Ono obuhvata sve aktivnosti koje se preduzimaju da bi se obezbedila efikasna i ekonomski prihvatljiva podrška korisnicima softvera.

Područje održavanja softvera je pokrivenom nizom standarda, kao što su IEEE 1219, IEEE 1061, ISO/IEC 14764, MIL-HDBK-347. U ovoj lekciji dat je pregled međunarodnih standarda koji se bave održavanjem softvera, te njihova veza s priručnikom softverskog inženjerstva SWEBOK v3.0.

Osnovni cilj svakog softverskog proizvoda je da zadovolji potrebe korisnika. Da bi se taj cilj ispunio, neophodno je da kroz svoj životni ciklus pretrpi modifikacije. Moglo bi se zato reći da je održavanje softvera ustvari, upravljanje procesom modifikacija.

Jedna od najcitanijih studija o održavanju softvera, koju su objavili 1980. godine istraživači sa UCLA, Lientz i Swanson, ponudila je klasifikaciju održavanja softvera na 4 kategorije, koje su kasnije prihvaćene i standardizirane u ISO/IEC 14764:2006:

- **Korektivno održavanje:** Reaktivne modifikacije softvera koje se vrše nakon isporuke, s ciljem popravljanja uočenih problema,
- **Adaptivno održavanje:** Modifikacije softvera koje se vrše nakon isporuke da bi softverski proizvod ostao upotrebljiv u okruženju koje se menja,
- **Perfektivno održavanje:** Modifikacije softvera koje se vrše nakon isporuke, da bi se popravile performanse ili pogodnost za održavanje,
- **Preventivno održavanje:** Modifikacije softvera koje se vrše nakon isporuke s ciljem otkrivanja latentnih grešaka pre nego što te greške dovedu do otkaza ili zastoja.

Niz istraživanja sprovedenih na tu temu pokazala su da preventivno održavanje čini manje od 5% svih aktivnosti održavanja, korektivno i adaptivno čine po oko 20%, dok perfektivno održavanje obuhvata više od 50% svih aktivnosti održavanja.

OSAM LEHMANOVIH ZAKONA ODRŽAVANJA SOFTVERA

Održavanje softvera je tema o kojoj su pisali mnogi autori, a često se referiraju na tzv. Lehmanovim zakonima evolucije softvera.

Održavanje softvera je tema o kojoj su pisali mnogi autori, a često se referiraju na tzv. Lehmanovim zakonima evolucije softvera, koji su nakon objave više puta modificirani. Taj set pravila uspostavljen je kako bi se proizvodom tako podložnom promenama kao što je softver, lakše upravljalo tokom životnog ciklusa. Tih osam Lehmanovih zakona glase:

- 1. "**Kontinuirane promene**" - sistem se mora kontinuirano adaptirati ili progresivno postaje manje zadovoljavajući (1974);
- 2. "**Narastajuća kompleksnost**" - kako sistem evoluira, njegova kompleksnost raste osim ako se ne radi ništa na održavanju ili smanjenju kompleksnosti (1974);
- 3. "**Samoregulacija**" - procesi evolucije sistema su samoregulišući sa distribucijom proizvoda i mera procesa bliskih normalnom (1974);
- 4. "**Očuvanje stabilnosti organizacije (invarijantni nivo rada)**" - prosečni efektivni globalni nivo aktivnosti u evoluirajućem sistemu je invarijantan po životnom ciklusu proizvoda (1978);
- 5. "**Očuvanje familijarnosti**" - kako sistem evoluira, svi povezani s njim, developeri, prodavači i korisnici, moraju ostati odlično upoznati sa sadržajem i ponašanjem softvera kako bi se postigla zadovoljavajuća evolucija. Preteran rast umanjuje tu informisanost. Stoga, prosečni inkrementalni rast ostaje invarijantan kako sistem evaluira. (1978)
- 6. "**Kontinuirani rast**" - funkcionalni sadržaj sistema mora se kontinuirano povećavati da bi se održalo zadovoljstvo korisnika tokom životnog ciklusa (1991);
- 7. "**Opadajući kvalitet**" - izgledaće da kvalitet sistema opada ako se ne održava rigorozno i adaptira promenama operativnog okruženja (1996);
- 8. **Sistem povratne veze**" - Evolutivni procesi sastoje se od sistema povratne veze sa više slojeva, više petlji, više agenata, i moraju se tretirati na način da se postiže značajno unapređenje nad svakom razumnom bazom (1996);

Ovi zakoni propisuju da je potreba za funkcionalnim promenama u softverskom sistemu neizbežna, a ne da je posledica nepotpune ili netačne analize zahteva ili lošeg programiranja. Oni navode da postoje granice koje razvojni tim softvera može postići u smislu bezbedne implementacije promena i novih funkcionalnosti.

Pored ISO/IEC modela procesa održavanja koji je opisan u SWEBOK i standardima ISO/IEC 12207 i ISO/IEC 14764, analizirani su i alternativni opisi procesa održavanja iz standarda ISO/IEC 15288 i britanskog de facto standarda ITIL (IT Infrastructure Library), koji održavanju prilaze sa aspekta životnog ciklusa sistema, umesto životnog ciklusa softvera.

PREGLED TEMA KOJE OBUVATA ODRŽAVANJE SOFTVERA PREMA STANDARDU ISO/IEC/IEEE 14764

Održavanje se definiše kao skup svih aktivnosti neophodnih da se obezbedi efikasna podrška korisnicima softvera, koja se deli na podršku pre i nakon isporuke softvera.

Održavanje se definiše kao skup svih aktivnosti neophodnih da se obezbedi efikasna podrška korisnicima softvera, koja se deli na podršku pre i nakon isporuke softvera, a podrška nakon isporuke obuhvata izmene softvera, obuku i službu za podršku.

Svrha održavanja softvera opisana je u standardu ISO/IEC 14764, a sledeća slika pokazuje teme koje obuhvata održavanje softvera prema tom standardu.

Održavanje softvera				
Osnove održavanja softvera	Ključni problemi održavanja softvera	Procesi održavanja	Tehnike održavanja	Alati za održavanje softvera
Definicije i pojmovi	Tehnička pitanja	Procesi održavanja	Razumijevanje programa	
Priroda održavanja	Pitanja menadžmenta	Aktivnosti održavanja	Reinženjering	
Potrebe za održavanjem	Ocjena troškova održavanja		Reverzni inženjering	
Većina troškova održavanja	Mjerenje održavanja softvera		Migracija	
Evolucija softvera				
Kategorije održavanja			Povlačenje	

Slika 1.1 Pregled tema koje obuhvata održavanje softvera prema standardu ISO/IEC/IEEE 14764 [Izvor: NM SE321-2020/2021.]

Cilj standardizacije u održavanju softvera je definisanje zajedničkog okvira koji omogućuje da svi učesnici u procesu razvoja, projektovanja i upravljanja softverom međusobno komuniciraju i razumeju se. Pri tome se ne nameće određeni model, tehnika ili aktivnost, nego se sistem zasniva na prethodnom usaglašavanju seta pravila, kojih se svi učesnici u procesu nakon usvajanja moraju pridržavati. Na taj način postupak održavanja softvera, koji u nekim slučajevima može činiti čak 80% ukupne cene softvera, postaje uređen, efikasan i ekonomičan.

U drugom skupu tema, opisan je primer važnosti pisanja preglednog koda i dokumentovanja za slučajeve gde programeri koji razvijaju kod i oni koji kasnije vrše održavanje, nisu iste osobe. Na pogodnost za održavanje jako utiče **preglednost koda**, posebno ako se ima u vidu velika fluktuacija IT kadrova, koji su jedna od najmobilnijih profesija koja često menja radno mesto, kako unutar iste, tako između različitih firmi. Pored esencijalnih veština kodiranja, poznavanja sintakse programske jezike, matematičke logike, veoma važan je i **stil kodiranja**. Treća grupa tema opisuje aktivnosti održavanja: *implementaciju procesa, analizu problema i modifikacija, implementaciju modifikacija, pregled/prihvatanje promena, migraciju i povlačenje softvera*.

▼ Poglavlje 2

Kandidati za redizajn softvera

OSNOVE ODRŽAVANJA SOFTVERA I HARDVERA

Na prvi pogled izgleda čudno da se termin "održavanje" odnosi na softver.

Na prvi pogled izgleda čudno da se termin "održavanje" odnosi na softver. Najzad, softver se ne troši na način kako to čine kompjuteri pri uključivanju i isključivanju, odnosno, kako se troše elektro-mehanički delovi sistema. Medij na kome je softver pohranjen može se menjati s vremenom zbog promena na magnetnim česticama koje se istroše sa površine diska ili trake ili savijanjem diska. Pažljivi sistemski administratori čuvaće kopije svih osnovnih softvera (i korisničkih fajlova), najbolje na izdvojenim lokacijama da bi izbegli probleme sa vatrom i poplavama.

Takođe, uzmite u obzir da **softver ne može da rđa ili da se ošteti zato što je prljavština ušla između električnih spojeva**. Glavne promene na napajanju kompjutera (110V, 60Hz u USA; 220V, 50Hz u EVROPI) ne utiču na softver. Naravno, **smanjenje napona može da ima uticaja, ali ovi problemi su dobro poznati**.

Oprezni administratori kompjuterskog sistema koriste kombinaciju električnih zaštitnih kola, podesivih izvora energije i neprekidnog napajanja da omoguće odgovarajući, zahtevani nivo usluge.

Ovi problemi su jasno povezani sa hardverom i ne predstavljaju nikakav problem za softverske inženjere - komponente koje otkazuju, greške u instalaciji ovih komponenti i neodgovarajuća upotreba od strane neiskusnog operatora.

OSNOVE ODRŽAVANJA HARDVERA

Pouzdanost hardverskih sistema uglavnom prati poznatu krivu „kade“ otkaza

Pouzdanost hardverskih sistema uglavnom prati poznatu krivu „kade“ otkaza (slika 1). Relativno visok broj nedostataka na početku rada hardvera je uglavnom zbog tri faktora:

- komponente koje otkazuju,
- grešaka u instalaciji ovih komponenti i
- neodgovarajućoj upotrebi od strane neiskusnog operatora.

Povećan broj otkaza na desnoj strani grafika predstavlja greške koje se javljaju pri kraju veka trajanja hardvera uzrokovane pojačanim otkazivanjem komponenti.

Održavanje hardvera je drugačije od njegovog testiranja. Na primer, očekujete da je nov auto koji ste skoro kupili sastavljan od strane eksperata koji koriste visoko kvalitetne delove i podsisteme (kao što su oni za elektronske i kočione operacije) testirane na sigurnost. Takođe, auto je trebalo da ima i druge, osnovne, preglede kada je odvezen sa proizvodne linije i kada je stigao u izložbeni prostor. Ove aktivnosti se smatraju aktivnostima testiranja.



Slika 2.1 Intenzitet otkaza u životnom veku hardverskih sistema [Izvor: NM SE321-2020/2021.]

Odgovarajuće održavanje hardvera uključuje **održavanje čistoće**, sa posebnim naglaskom na pokretne delove, **zamenu otkazalih komponenti** i opštim planiranjem zamene komponenti koje su stare, nedostajuće, imaju grešku ili postoji rizik od skrog otkaza.

Analiza cena/dobit treba da se izvede da bi se odredilo da li je trud održavanja neophodan i predstavlja cilj organizacije. Iznenadujuće, ovaj prilaz je takođe dobar način da se analizira softversko održavanje.

FAKTOVI KOJI ZAHTEVAJU DA SE SOFTVER ODRŽAVA

Procena trajanja preostalog životnog veka softvera takođe se mora uzeti u obzir pri određivanju da li održavanje vredi dodatnih troškova.

Softversko održavanje moglo bi se opisati kao sistematski proces menjanja softvera koji još uvek radi, tako da se postigne prevencija sistemskih otkaza i da se poboljšaju performanse.

Softversko održavanje uključuje:

- držanje softverskog interfejsa prostim i standardnim, poklanjajući posebnu pažnju problematičnim modulima,
- zameni neispravnih komponenti i
- opštim planiranjem za promenu komponenti koje su stare, funkcionalno zastarele i rizične po mogući otkaz.

Procena trajanja preostalog životnog veka softvera takođe se mora uzeti u obzir pri određivanju da li održavanje vredi dodatnih troškova.

Postoji nekoliko faktora koji zahtevaju da se softver održava (označeni su rednim brojevima 1-12):

1. Hardverska platforma se menja ili postaje zastarella.

2. Operativni sistem se menja ili postaje zastareo.
3. Kompajler se menja ili postaje zastareo.
4. Softverski jezički standardi se menjaju ili postaju zastareli.
5. Komunikacioni standardi se menjaju ili postaju zastareli.
6. Grafički interfejsi (GUI) se menjaju ili postaju zastareli.
7. Spektar namene softverskih paketa se menja ili postaje zastareo.
8. Veze sa proizvođačima drugih aplikacija ili sistemskih softvera se menjaju.
9. Softver može biti defektan, a to postaje očigledno tek kada ga korisnik upotrebi.
Ovi defekti moraju biti popravljeni
10. Korisnici traže nove funkcionalne i nefunkcionalne (brzina, resursi, pouzdanost, bezbednost i sl.) karakteristike.
11. Softver mora biti poboljšan da može da se poredi sa novim konkurentnjim proizvodom.
12. Postojeće softverske greške moraju se sprečiti primenom preventivnih mera za slučaj novog puštanja u rad.

KATEGORIZACIJA FAKTORA KOJI ZAHTEVAJU ODRŽAVANJE SOFTVERA

Tipični problemi koji ukazuju na potrebu redizajna softvera moraju biti podeljeni u nekoliko različitih grupa

Nakon faktora koji govore u kom slučaju se softver treba održavati na ovakav ili onakav način, navećemo i neke tipične probleme koji ukazuju na potrebu redizajna softvera. Ovi faktori moraju biti podeljeni u nekoliko različitih grupa:

- Faktori od 1 do 9 mogu biti klasifikovani kao faktori za adaptivno održavanje, i oni se koriste da se softver prilagodi novoj tehnologiji.
- Faktori od 10 do 11 mogu biti klasifikovani kao faktori za perfektivno održavanje, i oni se koriste da softver bude korektniji pod novim uslovima, u smislu da ima što manje grešaka. Termin "perfekcionističko održavanje" se takođe koristi u ovom slučaju.
- Faktor 12 može biti klasifikovan kao faktor za preventivno održavanje, i on se koristi da smanji mogućnost nastajanja greške na softveru.

Međutim, postoji i zajednička nit. **Bitan korak u svim tipovima softverskog održavanja je razumevanje softverskog sistema koji održavamo.** Pre nego što serviser počne da menja softver da bi ispravio problem ili ga okarakterisao, mora razumeti softver u celini, ali i određene module koji moraju biti modifikovani.

Mnogi eksperimenti su pokazali da razumevanje programa, što uključuje razumevanje softverskih zahteva i dizajna, oduzima otprilike pola napora na održavanju softvera.

Različiti prilazi softverskom održavanju će biti obrađeni u ostatku ovog poglavlja. Međutim treba imati na umu da ništa u životu nije besplatno i svako održavanje softvera mora biti finansirano kao deo softverskog budžeta organizacije.

Opisaćemo tipičan prilaz procesu održavanja softvera koji podrazumeva dve radnje: **određivanje gde je problem nastao i njegovu popravku.**

Mnogi problemi u softveru mogu se pronaći na neki od ovih načina:

- **Nepričučivost softvera razvojnim standardima. Ovi problemi uključuju loš interfejs između modula, kao što je prosleđivanje pogrešnog broja ili tipa argumenta funkcijama.**
- Logičke greške u programskim modulima, kao što su petlje, račvanje ili nekonzistentna stanja programa.
- **Neusklađenost između zahteva, dizajna, koda i dokumentacije informacionog sistema.**

DA LI SOFTVER BOLJE REDIZAJNIRATI ILI ODRŽAVATI?

Mada ne postoje apsolutna pravila kada je bolje redizajnirati sistem nego održavati postojeći, neki faktori koji se uzimaju u obzir su razatrani u ovom odeljku.

Nakon faktora koji govore u kom slučaju se softver treba održavati na ovakav ili onakav način, navećemo i neke tipične probleme koji ukazuju na potrebu redizajna softvera, odn. izrade novog.(IEEE Std 1219-1998, IEEE Standard for Software Maintenance, IEEE, 1998).

Mada je održavanje jedan neprekidan proces, ozbiljno razmatranje mora, eventualno, biti dato redizajniranju softverskog sistema. Glavna briga je kako odrediti da li je sistem nepovratno otkazao ili se može uspešno opraviti i održavati.

Grupe za testiranje koje rade sa softverom sve vreme treba da posmatraju opšte stanje softvera, i kada bude neophodno, preporuče da je softver bolje redizajnirati nego ga nastaviti korektivno održavati. Troškovi i povoljnosti nastavka održavanja softvera kada postane sklon greškama, neefikasan i skup, moraju biti vagani sa onima za redizajniranje sistema čak i kada redizajniranje nije najbolje rešenje.

Odluka o redizajniranju ili o zaustavljanju održavanja sistema može biti izvršena na nekoliko načina:

- **Podrška može jednostavno biti uklonjena i dozvoliće se da sistem postane zastareo i nekoristan;**
- **Mora se obezbediti minimum podrške neophodne za funkcionisanje sistema dok novi sistem ne bude izgrađen;**
- **ili sistem može biti podmlaćivan sektor po sektor, čime mu se produžava životni vek.**

Kako će se redizajn pokazati, zavisi od individualnih okolnosti sistema, njegovog operativnog okruženja i potreba organizacije koja ga podržava.

Mada ne postoje apsolutna pravila kada je bolje redizajnirati sistem nego održavati postojeći, neki faktori koji se uzimaju u obzir su razatrani u ovom odeljku.

GLAVNI POKAZATELJI ZA REDIZAJN: ČESTI SISTEMSKI OTKAZI; KODOVI STARI VIŠE OD PET GODINA

Aplikacija kojoj stalno treba korektivno održavanje je kandidat za redizajn. Promenom tehnologija aplikacije mogu postati neefikasne, teške za proveru i u nekim slučajevima zastarele

Glavni pokazatelji za redizajn, suprotstavljeni nastavku održavanja, direktno su proporcionalni broju karakteristika izlistanih u narednim odeljcima. Što je veći broj karakteristika, veći su pokazatelji za redizajn.

Česti sistemski otkazi: Aplikacija kojoj praktično stalno treba korektivno održavanje je kandidat za redizajn. Sa starenjem sistema potrebno je dodatno održavanje, sistem postaje sve nepostojaniji i osetljiv na promene. Što je stariji kod, sve su potrebnije modifikacije, novi zahtevi i proširenja mogu izazvati pad sistema.

Greške treba analizirati da bi se odredilo da li je ceo sistem odgovoran za otkaze ili su odgovorni neki moduli, ili su delovi koda pogrešni. Ako je ovo drugo krivac za otkaze, onda će redizajniranje samo tih delova biti dovoljno.

Kodovi stari više od pet godina: Predviđeni životni ciklus velikog dela aplikacija je 3-5 godina. Pogoršanje karakteristika softvera sa godinama je rezultat brojnih opravki i krpljenja. Ako je sistem star više od pet godina, verovatno je zastareo i preskup za rad, ovo je stanje većine kodova koji su sada u upotrebi. Posle pet godina održavanja, mnogi sistemi su dovedeni do tačke kada dodatno proširenje i popravke postaju vremenski vrlo duge. Deo kodova je verovatno loše strukturiran ili loše napisan. Međutim, kodovi koji su tačni i adekvatni za originalno okruženje, promene u tehnologiji i aplikacijama mogu napraviti neefikasnim, teškim za proveru i u nekim slučajevima zastarelim. Sa druge strane, aplikacije koje su dizajnirane i razvijane sistemski, na način lak za održavanje, i ako je održavanje izvođeno pažljivo i dokumentovano u skladu sa uspostavljenim standardima i po direktivama, moguće je da rade efikasno i efektivno još mnogo godina.

GLAVNI POKAZATELJI ZA REDIZAJN: KOMPLEKSNE PROGRAMSKE STRUKTURE I LOGIČKI TOKOVI; KODOVI PISANI ZA PRETHODNU GENERACIJU HARDVERA

Ako sistem sadrži veliki deo koda siromašne programske strukture i dokumentacija je oskudna- on je kandidat za redizajn. Treba redizajnirati i softvere pisane za prethodnu generaciju hardver

Previše kompleksne programske strukture i logički tokovi: "Održati jednostavnost" trebalo bi da bude zlatno pravilo svih standarda i putokaza u programiranju. Prečesto, programeri pokušavaju da pišu delove kodova upotrebljavajući najmanji broj iskaza ili zauzimajući

najmanji mogući memorijski prostor. Ovaj prilaz kodiranju je rezultirao kompleksnim, praktično nerazumljivim kodom. Siromašna programska struktura doprinosi kompleksnosti. Ako sistem sadrži veliki deo kodova ovog tipa i dokumentacija je često oskudna- on je kandidat za redizajn. Kompleksnost se takođe odnosi na nivo donošenja odluka u kodu. Što veći broj grananja (mogućnost) odluke, kompleksniji će biti softver.

Takođe, što je veći broj linearne nezavisnih kontrolnih grana u programu, veća je kompleksnost programa. Programi sa nekim ili svim nabrojanim karakteristikama su obično veoma teški za održavanje i kandidati su za redizajn:

- **Preterana upotreba DO koraka**
- **Preterana upotreba IF iskaza**
- **Nepotrebni GOTO iskazi**
- **Ugradivanje konstanti i literala**
- **Nepotrebna upotreba globalnih promenjivih**
- **Samo-promenjivi kod.**
- **Mnogostruki ulazni ili izlazni moduli**
- **Preterana interakcija između modula.**

Kodovi pisani za prethodnu generaciju hardvera: Samo par industrija ima iskustvo sa tako brzim rastom kao kompjuterska industrija, naročito u oblasti hardvera. Ne samo da postoji značajno tehnološko napredovanje, već je i cena hardvera desetostruko opala tokom zadnje decenije.

Ovaj fenomen je generisao raznovrsne snažne hardverske sisteme. Softver pisan za stariju generaciju hardvera je često neefikasan u novijim sistemima. Pokušaji da se kodovi površno modifikuju da dostignu noviji hardver su generalno uzaludni, skupi i dugotrajni.

GLAVNI POKAZATELJI ZA REDIZAJN: VEOMA VELIKI MODULI ILI PODRUTINSKE JEDINICE; PREKOMERNA ZAHTEVANJA RESURSA

Kandidati za redizajn su i sistemi koji rade po principu emulacije, mega-sistemi koji su pisani kao jedan ili više vrlo velikih programa i koji zahtevaju prekomerne resurse

Sistemi koji rade po principu emulacije: Svaki softverski proizvod ne može bez odgovarajućeg hardvera da izvrši misiju i obrnuto, skoro da nema hardvera bez softvera. **Jedna od tehnika koje se koriste za održavanje sistema u radu na novijem hardveru je emulacija-imitiranje originalnog hardvera i operativnog sistema.** Imitiranje se koristi kada zbog resursa nije moguće promeniti sistem ili su troškovi preveliki. Za ove sisteme, linija između korisnosti i zastarelosti je veoma tanka.

Jedna od najvećih teškoća u održavanju ovakvih sistema je pronalaženje personala koji je upoznat sa originalnim hardverom i voljni su da ga održavaju. Zato što je hardver koji se imitira zastareo, specifične veštine potrebne za održavanje ovakvih sistema imaju malu primenu drugde, i perspektiva održavanja takvog sistema nije obećavajuća.

Veoma veliki moduli ili podrutinske jedinice: Mega-sistemi koji su pisani kao jedan ili više vrlo velikih programa ili potprograma (hiljade ili desetine hiljada redova kodova po programu) mogu biti ekstremno teški za održavanje. Veličina modula obično je direktno proporcionalna nivou napora neophodnom za njegovo održavanje. Ako veliki moduli mogu biti restrukturirani i podeljeni na manje, funkcionalno povezane sektore, pogodnost za održavanje tih sistema će biti poboljšana (unapređena).

Prekomerno zahtevanje resursa: Aplikativni sistem koji zahteva znatno vreme centralne procesorske jedinice, veliku memoriju ili resurse drugih sistema može da bude veoma ozbiljno opterećenje za sve korisnike. Ovakvi sistemi sprečavaju obavljanje drugih poslova i snižavaju nivo i kvalitet usluge za sve druge korisnike.

U pitanja koja treba postaviti u pogledu ovih sistema treba uključiti i pitanja da li je jeftinije dodati još snage kompjuterima ili redizajnirati i reimplementirati sistem, i da li će redizajn smanjiti zahteve za resursima. Ako zahtevi za resursima neće biti smanjeni, onda nema koristi od redizajna.

TVRDO-KÔDIRANI PARAMETRI

Mnogi stariji sistemi su dizajnirani sa vrednostima parametara korišćenih za izvođenje specifičnih izračunavanja tvrdo-kodiranih u izvornom kôdu umesto da se čitaju iz tablica ili fajlo

Tvrdo-kodirani parametri su subjekti za izmenu: Mnogi stariji sistemi su dizajnirani sa vrednostima parametara korišćenih za izvođenje specifičnih izračunavanja tvrdo-kodiranih u izvornom kodu umesto da se čitaju iz tablica ili fajlova podataka. Kad su neophodne promene ovih parametara svaki program u sistemu mora biti ispitivan, izmenjen i ponovo sastavljen ako je neophodno. Ovo je dugotrajan i proces sklon greškama koji troši vreme i resurse neophodne za ove izmene.

Ako je moguće, program bi trebalo modifikovati unosom parametara u jedan modul ili čitanjem parametara iz centralne tabele vrednosti. Ako ovo ne može biti urađeno, treba ozbiljno razmotriti redizajniranje sistema.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Vrste i kategorije održavanja SW

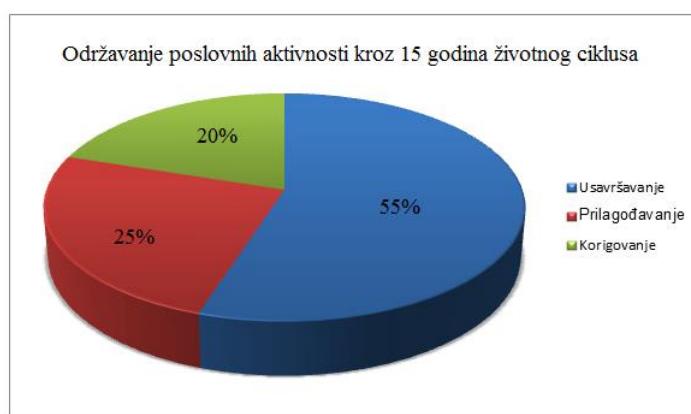
KATEGORIJE ODRŽAVANJA SOFTVERA

Generalno posmatrano, postoje četiri kategorije održavanja: 1. Korektivno održavanje; 2. Adaptivno održavanje 3. Perfektivno održavanje 4. Preventivno održavanje

Postoji više vrsta aktivnosti koje nazivamo održavanjem:

- Održavanje u smislu ispravke softverskih grešaka
- Održavanje u smislu prilagođavanja softvera različitim operativnim okruženjima
- Održavanje u smislu dodavanja i modifikacija sistemskih funkcionalnosti.

Sa slike se vidi da najveći deo održavanja, gotovo dve trećine otpada na dodavanje funkcionalnosti i modifikacije tj. usavršavanje softvera, dok manji deo uzimaju korigovanja i prilagođavanja. (Slika 1 i 2)



Slika 3.1 Vrste održavanja [Izvor: NM SE321-2020/2021.]

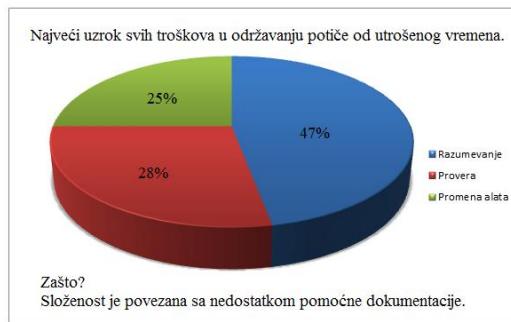
Generalno posmatrano, postoje tri kategorije održavanja:

1. **Korektivno održavanje**
2. **Adaptivno održavanje**
3. **Perfektivno održavanje**

Ove definicije su kasnije ažurirane u standardu za softversko održavanje, ISO/IEC 14764 i uključuju 4 kategorije:

1. **Korektivno održavanje;**
2. **Adaptivno održavanje;**

3. Perfektivno održavanje i 4. Preventivno održavanje



Slika 3.2 Troškovi održavanja [Izvor: NM SE321-2020/2021.]

KOREKTIVNO ODRŽAVANJE SOFTVERA

Prvi korak u korektivnom održavanju softvera je određivanje šta treba da se popravi. Nakon toga treba doneti odluku koji, kako ili kada će problem biti rešen.

Korektivno održavanje softvera: Prvi korak u korektivnom održavanju softvera je određivanje šta treba da se popravi. U korektivnom održavanju softvera sve aktivnosti počinju sa identifikacijom problema u postojećem softveru. Tačno određivanje modula koji prouzrokuje problem može biti teško.

Ovaj osnovni prilaz baziran je na tehnički pokušaja i grešaka, vođen serviserovim poznavanjem sistema i njegovim osnovnim poznavanjem kompjuterske nauke i softverskog inženjeringu.

Kad problem jednom bude određen, odluka koja će biti donesena je koji, kako ili kada će problem biti rešen. Ovo obično zahteva nekoliko povezanih radnji:

- **Zaključak da problem postoji.**
- **Dokumentovanje problema.**
- **Određivanje važnosti problema.**
- **Određivanje prioriteta problema po redu važnosti za popravku.**
- **Određivanje koji problemi neće biti popravljeni zbog nedostatka resursa.**

- **Rešavanje problema.**
- **Testiranje sistema da vidimo da li rešenje ovog problema može da prouzrokuje grešku u drugom delu softvera.**
- **Dokumentovanje rešenja kao dela izvornog koda.**
- **Dokumentovanje rešenja u drugim formama ako su izvršene promene originalnog dizajna sistema.**
- **Ažuriranje baze podataka sa informacijama o softverskim greškama.**

Možda ovo izgleda kao mnogo posla. Održavanje softvera zahteva, po svojoj prirodi, veliku količinu rada sa dokumentima da bi se ažurirala dokumentacija. Najzad, proizvođači softverskih sistema ili vlasnici projekta retko zauvek ostaju na istim projektima. Sa velikom

promenom (gubitkom) personala, pisana dokumentacija o svim izmenama mora biti ostavljena budućim serviserima sistema.

UOČAVANJE SOFTVERSkiH PROBLEMA

Problem može uočiti osoba koja je upoznata sa kompletnim sistemskim zahtevima

Uočavanje softverskih problema može se dogoditi na više nivoa.

U najidealnijoj situaciji problem je uočila osoba koja je upoznata sa kompletним sistemskim zahtevima. U ovom slučaju dokumentovanost problema je trenutna i određivanje izvora problema je tada obično lako.

Češća situacija je da je osoblje koje uoči problem potpuno neupoznato sa unutrašnjom strukturom softvera. Zaista, osoblje koje uoči problem može čak biti i neupoznato sa operacijama koje se očekuju od softvera.

Ovo je tipičan primer problema koji su prvo primetili novi korisnici koji su zatim telefonom pozvali tehničku podršku za upoznavanje sa procesima ili sličnim primenama softvera. Tehnički, personal koji radi na tehničkom izveštavanju je odgovoran za vođenje korisnika koji traži pomoć kroz ovaj prepostavljeni problem, da traži problem u podešavanju i konfiguraciji softvera, ukratko da odredi da li je problem nastao greškom korisnika ili je greška u softveru. Tamo, u osnovi postoje dva problema: korisnik nije razumeo objašnjenja za instalaciju i rad, ili je softver u otkazu.

Kad personal za tehničku podršku odredi da korisnik nije razumeo dokumentaciju instalacije sistema ili sistemskih operacija, ostaje im da urade dva zadatka. Korisnik mora biti vođen pravilnim instrukcijama dok ne reši problem.

Personal za tehničku podršku mora takođe dokumentovati problem tako da dokumentacija i korisnički interfejs budu poboljšani. Ovo se često radi u novoj verziji koja će se tek pojaviti. Personal za tehničku podršku treba da potpuno opiše i dokumentuje probleme i predstavi to osobama odgovornim za nove verzije sistema tako da će korisnici, nove verzije smatrati unapređenim. U prošlosti je većina kompjuterske dokumentacije mogla biti unapređena upošljavanjem tehnički obrazovanih pisaca.

U slučaju softverske greške, od korisnika se ne očekuje da obezbedi detaljnju dokumentaciju o detaljima problema. Tipičan korisnik nema toliko tehničko znanje i, u svakom slučaju to nije njegov posao.

DOKUMENTOVANJE SOFTVERSkiH PROBLEMA

Najjednostavnije rešenje je unos podataka o održavanju u tekstu formatu direktno u tabelu ili u bazu podataka.

Personal za tehničku podršku je odgovoran da uzme informacije od korisnika i organizuje ih na način na koji mogu biti iskorišćene u procesu opravke.

Cilj ličnog dokumentovanja softverskih problema je u sposobnosti ispunjavanja formulara za softversko održavanje ili formulara za izveštaj o održavanju. **Formulari pomažu u potpunom opisivanju problema i za procenu relevantne važnosti problema.** Na osnovu tih formulara, rukovodilac softverskog održavanja će proceniti i odrediti prioritete na osnovu kojih će se vršiti održavanje softvera.

Nakon što je problem rešen, još jedan formular treba da se ispuni. Ovaj formular pokazuje odgovor na formular sa pitanjima o softverskom održavanju i kao takav je često nazivan formular sa odgovorima za softversko održavanje. **Formular sa odgovorima uvek pokazuje listu zahvaćenih (zaraženih tim problemom) modula i vreme koje je bilo potrebno za rešavanje problema.**

Treba primetiti i važnost da se ovim formularima doda ekstra dokumentacija. Veoma je korisno ako se kao prilog doda i odštampan sadržaj ekrana, na kom se vidi manifestacija greške. Ako nema ovih mogućnosti, slika ekrana može biti dovoljna. Očigledno, digitalna slika je najbolja, direktno u tabelu ili u bazu podataka.

Svakako, nema potrebe za formularima pisanim na papiru. Formulari mogu biti čuvani u elektronskoj formi sa automatskim unosom u bazu podataka. Idealno, serviserima treba da budu dostupni softverski paketi koji podržavaju aktivnosti u održavanju, automatski stvarajući formulare i snimaju informacije u bazu podataka . U mnogim softverskim okruženjima nedostatak sposobnosti generisanja formulara je najkritičniji problem.

Najjednostavnije rešenje je unos podataka o održavanju u tekst formatu direktno u tabelu ili u bazu podataka.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 4

Adaptivno, perfektivno, preventivno održavanje

ADAPTIVNO ODRŽAVANJE

Adaptivno održavanje je proces menjanja softvera u skladu s novim trendovima na tržištu.

Adaptivno održavanje je proces menjanja softvera u skladu s novim trendovima na tržištu. Očekivane promene hardvera, sistema, interoperabilnosti aplikacionih paketa i novih karakteristika nametnutih od strane konkurenčije su važni u određivanju potrebe za adaptivnim održavanjem softvera.

Poznat je svima primer MICROSOFT WORD-a koji je prošao kroz mnoge verzije od njegovog početka. Neke od ovih promena su uključivale korisnički interfejs i lakoću korišćenja. Druge izmene su obezbedile dodatne karakteristike i zajednički format fajlova. Često, dokumenta otvorena u jednoj od kasnijih verzija WORD-a ne mogu biti čitana ili menjana u ranijim verzijama istog softvera (ovaj problem nije jedinstven za MICROSOFT, isti iskaz se može primeniti na COREL WORD PERFECT, između ostalih).

Za neke korisnike, prednost zajedničkog formata fajlova u kojima dokumenti mogu biti čitani i na PC-iju i MACINTOSH-u je važnija naspram sposobnosti stvaranja velikih fajlova za dokumente. Za druge korisnike su važnije mogućnosti koje obezbeđuju makro-i nego dodatni sigurnosni zahtevi za tretiranje makro-virusa koji se mogu prenositi sa PC-a na MACINTOSH i obratno.

Kao i kod korektivnog održavanja, ovde su neophodna znatna testiranja softvera, naročito regresione testiranje.

To je podesno do te tačke da se pokažu neki problemi koji nisu bili očekivani da će se otkriti čak i tokom najiscrpnijih testiranja softvera.

Postoji još jedan problem koji naročito uznemirava softverske serviserne. Mnogi softverski inženjeri, naročito oni koji su imali iskustvo sa kompjuterima koji su imali veliku količinom fizičke memorije, ne misle da njihovi sadašnji sistemi imaju ikakva ograničenja. Naime, **memorija može biti potrošena naročito ako je sastavljena od jeftinjih proizvoda čiji su memorijski zahtevi nepoznati.**

Treba biti svestan da je situacija naročito složena za održavanje ako je softver koji treba održavati u vezi sa jeftinim komponentama hardvera čija je unutrašnja struktura nepoznata.

Jasno je da softverski serviser mora imati mnogo više veština od običnog iskustva u programiranju.

PERFEKTIVNO ODRŽAVANJE SOFTVERA

Perfektivno održavanje odnosi se na promene zahtevane od strane korisnika sistema ili programera koji usavršavaju isti na način koji neće izmeniti njegovu funkcionalnost.

Perfektivno održavanje predstavlja modifikacije softverskog proizvoda nakon isporuke radi unapređenja performansi.

Softver je „rođen“ da bi bio menjan, poboljšavan. Perfektivno održavanje odnosi se na promene zahtevane od strane korisnika sistema ili programera koji usavršavaju isti na način koji neće izmeniti njegovu funkcionalnost. Stara poslovica kaže: „Ne popravljaj ono što nije pokvareno“. Perfektivno održavanje zapravo je baš to- popravljanje, odnosno usavršavanje softvera koji radi.

Ciljevi kojima se teži su:

- **Smanjenje troškova za korišćenje sistema**
- **Povećanje pogodnosti za održavanje (meintabilnosti) sistema**
- **Bliže zadovoljavanje potreba korisnika**

Perfektivno održavanje koristi sav napor na doterivanje softvera i preciziranje dokumentacije.

Važno je da potencijalna dobrobit perfektivnog održavanja nadjačava:

- **troškove održavanja kao i**
- **opportunitetne troškove poboljšanja na drugim mestima ili korišćenje resursa za nove razvoje.**

Zbog svega navedenog pre nego što se upustimo u perfektivno održavanje potrebno je da prođemo kroz proces analize.

U svakom slučaju, sasvim malo perfektivnog održavanja može da doneše veoma teatralne efekte.

PREVENTIVNO ODRŽAVANJE SOFTVERA

Predstavlja modifikacije softverskog proizvoda nakon isporuke radi detektovanja ili korekcije skrivenih grešaka u softverskom proizvodu pre nego što se aktiviraju.

Preventivno održavanje predstavlja modifikacije softverskog proizvoda nakon isporuke radi detektovanja ili korekcije skrivenih grešaka u softverskom proizvodu pre nego što se aktiviraju.

ISO/IEC 14764 klasificuje adaptivno i perfektivno održavanje kao unapređenja. Takođe, grupiše zajedno i korektivno i preventivno održavanje u kategoriju korekcije, kao što je

prikazano u sledećoj tabeli. Preventivno održavanje je najnovija kategorija i najčešće se izvodi na softverskim proizvodima gde je ključno pitanje sigurnosti (Slika 1).

	Korekcije	Unapređenja
Proaktivno	Preventivno	Perfektivno
Reaktivno	Korektivno	Adaptivno

Slika 4.1 Proaktivno i reaktivno održavanje SW [Izvor: NM SE321-2020/2021.]

Preventivno održavanje softvera treba da proaktivno spreči softverske probleme u postojećem softveru pre nego što oni nastanu.

Najverovatnije najbolji način da se ilustruje preventivno održavanje na tematskom primeru-poznat kao „Y2K problem 2000 godine“.

Ova situacija je izgledala bezopasno: mnogi kompjuterski problemi

prikazuju datume koristeći dva mesta za cifre i 2000. godina je tražila novu interpretaciju ovih već postojećih fajlova. Zašto je ovaj problem tako važan? Postoje zapravo dva dela za odgovor: prvi u veličini problema koji su razumele organizacije koje koriste softver ili su odgovorne za njega, i drugi mogući skriveni problemi čiji opseg se može samo nagadati. Svaki od ovih scenarija ima problem sa prekoračenjem (overflow) koji će morati da bude rešen. Velika količina softvera na koje se odrazio Y2K problem je prouzrokovao noćnu moru za održavanje, naročito zbog manjka kvalifikovanog personala.

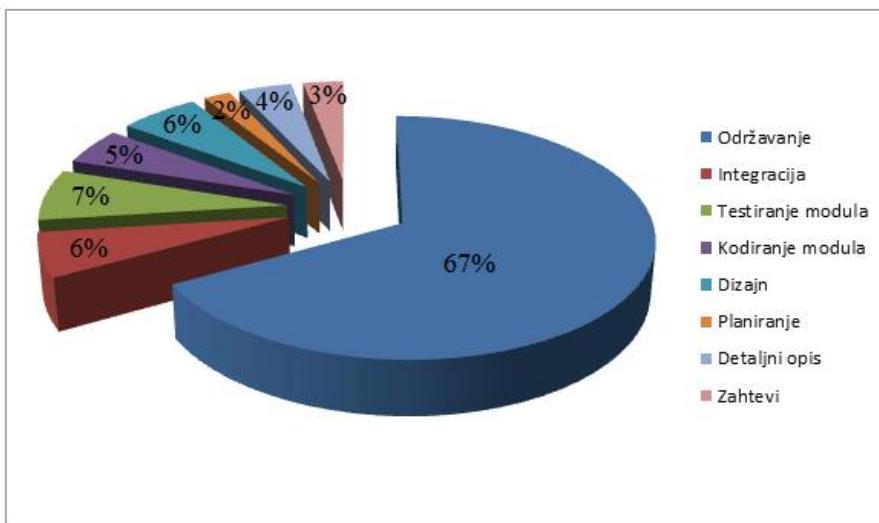
Postoji mnogo scenarija za buduće probleme slične problemu Y2K:

- Iscrpljivanje telefonskih pozivnih kodova u Americi do 2020 godine
- Datumski i vremenski mehanizmi u UNIX operativnim sistemima baziranim na broju sekundi je startovao 01.01.1970 godine. Broj sekundi se čuva u 32-bitnom integratoru koji će se prepuniti 19.01.2038 godine.
- Devetocifreni brojevi socijalnog osiguranja će u Americi biti iscrpljeni otprilike u periodu od 2050 do 2075 godine.

AKTIVNOSTI ODRŽAVANJA

To su: prerada softverskog koda, optimizacija performansi softvera, migracija na druge platforme, konverzija u nove arhitekture, uklanjanje neaktivnog koda itd.

Na slici 2 su prikazani troškovi životnog ciklusa razvoja softvera.



Slika 4.2 Troškovi pri razvoju softvera [Izvor: NM SE321-2020/2021.]

Aktivnosti u održavanju softvera, a koji služe i kao pokazatelji kvaliteta softverskog proizvoda su:

- prerada softverskog koda,
- optimizacija performansi softvera,
- migracija na druge platforme,
- konverzija u nove arhitekture,
- uklanjanje neaktivnog koda,
- uklanjanje skrivenih aplikacija,
- povlačenje softvera iz upotrebe.

Ostale aktivnosti tokom procesa održavanja:

- kontakti sa klijentima
- pisana korespondencija
- izlasci na teren i dr.

▼ Poglavlje 5

Ključni aspekti softverskog održavanja

TEHNIČKI ASPEKTI ODRŽAVANJA SOFTVERA: OGRANIČENO RAZUMEVANJE

Ograničeno razumevanje odnosi se na to kako brzo softver inženjer može razumeti gde da napravi promene ili korekcije u softveru koji nije razvijao.

Veći broj ključnih tema mora se razmotriti radi osiguranja efikasnog održavanja softvera. Važno je razumeti da softversko održavanje pruža jedinstvene tehničke i menadžerske izazove softverskim inženjerima. Jedan od primera je pokušaj pronalaženja greške u softveru sa 500 hiljada linija koje softver inženjer nije razvijao. U nastavku su date neke tehničke i menadžment teme vezane za softversko održavanje, koje su grupisane u sledeće kategorije:

- Tehnički aspekti
- Menadžment aspekti
- Procena troškova
- Merenja

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Tehnički aspekti održavanja softvera su:

- Ograničeno razumevanje
- Testiranje
- Analiza uticaja (eng. Impact analysis)
- Lakoća održavanja

Ograničeno razumevanje se odnosi na to kako brzisoftver inženjer može razumeti gde da napravi promene ili korekcije u softveru koji nije razvijao. Istraživanja pokazuju da je 40% do 60% napora održavanja posvećeno razumevanju softvera koji se modifikuje. Prema tome, tema sveobuhvatnog razumevanja softvera je od velikog interesa softver inženjerima. Razumevanje je teže u text-oriented reprezentacijama, u izvornom kodu, na primer, gde je često teško praćenje i evolucija softvera tokom brojnih verzija, ako promene nisu dokumentovane i kada programeri nisu na raspolaganju za objašnjenje, što je najčešće i slučaj u praksi.

TEHNIČKI ASPEKTI ODRŽAVANJA SOFTVERA: ANALIZA UTICAJA; LAKOĆA ODRŽAVANJA

Analiza uticaja opisuje kako sprovesti kompletну analizu uticaja promena u postojećem softveru; Lakoću održavanja definiše jednostavnost sa kojom se softver može održavati

Analiza uticaja opisuje kako sprovesti kompletnu analizu uticaja promena u postojećem softveru. Održavaoci moraju posedovati suštinsko znanje strukture i sadržaja softvera. Oni koriste to znanje radi izvođenja analize uticaja, koja identificuje sve sistemske i softverske proizvode pogodene zahtevima za softverskim promenama i razvija procenu resursa potrebnih za ispunjenje tih promena.

Kao dodatak, određuje se i rizik pravljenja tih promena. Zahtev za promenama, nekada se naziva i **modification request (MR)**, mora prvo biti analiziran i preveden u softverske termine.

Neki autori navode da su ciljevi analize uticaja:

- Određivanje opsega promena u cilju planiranja i implementacije rada
- Određivanje tačnih procena resursa potrebnih radi izvođenja rada
- Cost/benefit analiza zahtevanih promena
- Komunikacija sa ostalima učesnicima o kompleksnosti datih promena.

IEEE [IEEE610.12-90] definiše lakoću održavanja (**maintainability**) kao jednostavnost sa kojom se softver može održavati, unaprediti, adaptirati i ispravljati radi zadovoljenja specificiranih zahteva. ISO/IEC definiše lakoću održavanja kao jednu od karakteristika kvaliteta.

Podkarakteristike lakoće održavanja mogu biti specificirane, pregledane, kontrolisane tokom aktivnosti softverskog razvoja u cilju smanjena troškova održavanja. Ako se to uradi uspešno, lakoća održavanja softvera će se unaprediti. Ovo je često teško postići zato što ove podkarakteristike nisu važne tokom procesa razvoja softvera.

Developeri su zaokupljeni mnogim drugim stvarima i često ne obraćaju pažnju na zahteve održavaoca softvera. Ovo može, a često se i dešava, rezultirati u nedostatku sistemske dokumentacije, koji je vodeći razlog za probleme u razumevanju programa i analizi uticaja. Može se reći da prisustvo sistemskih i razvijenih procesa, tehnika i alata pomaže unapređenju lakoće održavanja sistema.

MENADŽMENT ASPEKTI ODRŽAVANJA SOFTVERA

Tim koji razvija softver nije obavezno i dodeljen na održavanje softvera kada postane operativan.

Menadžment aspekti održavanja softvera su:

- Usaglašavanje sa organizacionim ciljevima

- Osoblje
- Procesi
- Organizacioni aspekti održavanja
- Outsourcing (usluge trećih lica)

Usaglašavanje sa organizacionim ciljevima: Organizacioni ciljevi opisuju kako demonstrirati povraćaj investicija od aktivnosti softverskog održavanja. Inicijalni softverski razvoj je obično zavisni od tipa projekta (**project-based**), sa definisanim vremenskim opsegom i budžetom. Osnovni značaj je isporuka na vreme i unutar budžeta za ispunjenje korisničkih potreba. Nasuprot tome, softversko održavanje često ima za cilj produžavanje životnog ciklusa softvera koliko god je moguće. Kao dodatak, može biti vođen potrebama korisnika radi ispunjenja korisničkih zahteva za ažuriranja i unapređenja softvera.

U oba slučaja, povraćaj investicija je mnogo manje jasan, pa je pogled na menadžment nivou često trošenje značajnih sredstava sa nejasnim merljivim (kvantitativnim) benefitom za organizaciju.

Osoblje: Ovaj aspekt se odnosi na to kako zainteresovati i zadržati osoblje za softversko održavanje. Održavanje se ne posmatra kao privlačan posao. Suočavanje i rešavanje problema koji su prouzrokovani radom drugih nije posebno primamljivo. Mnogi autori daju čitavu listu „**staffing-related**“ problema zasnovanih na istraživanju. Osoblje za održavanje se najčešće vidi kao "second-class citizens", čime se sa moralnog stanovišta otvaraju mnoga pitanja.

Organizacioni aspekti održavanja: Organizacioni aspekt opisuje kako identifikovati koja organizacija i/ili funkcija će biti odgovorna za održavanje softvera. Tim koji razvija softver nije obavezno i dodeljen na održavanje softvera kada postane operativan. U odlučivanju gde se locira funkcija softverskog održavanja, organizacija može, na primer, ostati sa originalnim developerima, ili ići na odvojeni tim (održavaoci).

Pošto postoji mnogo za i protiv za svaku od ovih opcija, odluka treba da se doneše od slučaja do slučaja. Važna je delegacija ili dodeljivanje odgovornosti održavanja jednoj grupi ili osobi, shodno organizacionoj strukturi.

MENADŽMENT ASPEKTI ODRŽAVANJA SOFTVERA: USLUGE TREĆIH LICA - OUTSOURCING

Mnoge firme će outsource-ovati samo ako pronađu način zadržavanja strateške kontrole. Problem je međutim, što je teško pronaći kontrolne mere u tom slučaju.

Usluge trećih lica – Outsourcing : Outsourcing (izvan kompanije) održavanje je danas veoma masovna pojava. Velike korporacije outsource-uju kompletan portfolio softverskog sistema, uključujući softversko održavanje. Najčešće, outsourcing opcija se bira za manje " **mission-critical** " softvere, pošto kompanije nisu spremne da izgube kontrolu nad softverom koji se koristi u osnovnom biznisu.

Mnoge firme će outsource-ovati samo ako pronađu način zadržavanja strateške kontrole. Problem je međutim, što je teško pronaći kontrolne mere u tom slučaju.

Jedan od najvećih izazova za outsourcing je određivanje zahtevanog opsega održavanih servisa i detalji ugovora. McCracken navodi da 50% outsource-era obezbeđuju servise bez jasnog nivoa servisiranja prema ugovoru. Outsourcing kompanije tipično potroše nekoliko meseci procenjujući softver pre bilo kakvog ulaska u ugovorni sporazum. Još jedan izazov je i tranzicija softvera outsource-eru.

✓ Poglavlje 6

Grupna vežba

GRUPNA VEŽBA: AKTIVNOSTI ODRŽAVANJA SOFTVERA LAMS

Softver inženjeri moraju razumeti različite kategorije softverskog održavanja u cilju adresiranja pitanja procene troškova softverskog održavanja.

Procena troškova održavanja (Cost Estimation): Za svrhe planiranja, procena troškova je važan aspekt softverskog održavanja. ISO/IEC14764 navodi da dva najpopularnija pristupa u procenjivanju resursa za software maintenance je korišćenje parametarskih modela i iskustava. Najčešće se koristi kombinacija ovih pristupa.

Merenja softverskog održavanja (Software Maintenance Measurement): Grady and Caswell u svojim radovima, diskutuju uspostavljanje corporate-wide software measurement programa. The Practical Software and Systems Measurement (PSM) projekat opisuje tematski vođen proces merenja koji se koristi od strane mnogih organizacija.

Specifična merenja: Održavalac mora da odredi koja merenja su odgovarajuća za datu organizaciju. Standardi [IEEE1219-98; ISO9126-01] predlažu merenja koja su više specifična za softversko održavanje. Ovo uključuje brojna merenja za svake od 4 podkarakteristike maintainability:

- **Analyzability** - Merenja maintainer napora ili resursa potrošenih u pokušaju dijagnosticiranja uzroka padova ili identifikovanju delova koji treba da se modifikuju
- **Changeability**- Merenja maintainer napora pridruženih implementaciji specificiranih modifikacija
- **Stability**- Merenja neočekivanih ponašanja softvera, uključujući i ona koja se pojavi prilikom testiranja
- **Testability**- Merenja maintainer i users napora u pokušaju testiranja modifikovanog softvera.

Maintenance Process : Podoblast **Maintenance Process** pruža reference i standarde korišćene za implementaciju procesa softverskog održavanja.

Tema Maintenance Aktivnosti razlikuje maintenance od development-a i prikazuje njegovu povezanost sa ostalim aktivnostima softverskog inženjerstva.

Grupna vežba traje 45 minuta.

KLJUČNI ASPEKTI SOFTVERSKOG ODRŽAVANJA

Važno je razumeti da softversko održavanje pruža jedinstvene tehničke i menadžerske izazove softverskim inženjerima.

Jedan od primera je pokušaj pronalaženja greške u softveru sa 500 hiljada linija koje softver inženjer nije razvio. Takmičenje za neophodne resurse u poređenju sa sredstvima za razvoj je stalna borba. Planiranje za buduće verzije, dok se kodiraju sledeće verzije i šalju dopune i zakrpe za tekuće verzije, takođe predstavlja izazov.

U nastavku su date neke tehničke i menadžment teme vezane za softversko održavanje, koje su grupisane u sledeće kategorije:

- Tehnički aspekti
- Menadžment aspekti
- Procena troškova
- Merenja

Tehnički aspekti su:

- Ograničeno razumevanje
- Testiranje
- Analiza uticaja (eng. Impact analysis)
- Lakoća održavanja

Menadžment aspekti su:

- Usaglašavanje sa organizacionim ciljevima
- Osoblje
- Procesi
- Organizacioni aspekti održavanja
- Outsourcing

Procena troškova vrši se na osnovu:

- Parametarskih modela
- Iskustva

Merenja su:

- Specifična merenja

OGRANIČENO RAZUMEVANJE

Ograničeno razumevanje odnosi se na to kako brzo softver inženjer može razumeti gde da napravi promene ili korekcije u softveru koji nije razvijao.

Istraživanja pokazuju da je 40% do 60% napora održavanja posvećeno razumevanju softvera koji se modifikuje. Prema tome, tema sveobuhvatnog razumevanja softvera je od velikog interesa softver inženjerima. Razumevanje je teže u text-oriented reprezentacijama, u izvornom kodu, na primer, gde je često teško praćenje i evolucija softvera tokom brojnih verzija, ako promene nisu dokumentovane i kada programeri nisu na raspolaganju za objašnjenja, što je najčešće i slučaj u praksi.

Analiza uticaja opisuje kako sprovesti kompletну analizu uticaja promena u postojećem softveru. Održavaoci moraju posedovati suštinsko znanje strukture i sadržaja softvera. Oni koriste to znanje radi izvođenja analize uticaja, koja identificuje sve sistemske i softverske proizvode pogođene zahtevima za softverskim promenama i procenu resursa potrebnih za ispunjenje tih promena. Kao dodatak, određuje se i rizik pravljenja tih promena. Zahtev za promenama, nekada se naziva i Modification Request (MR), mora prvo biti analiziran i preveden u softverske termine.

Neki autori navode da su ciljevi analize uticaja:

- Određivanje opsega promena u cilju planiranja i implementacije rada
- Određivanje tačnih procena resursa potrebnih radi izvođenja rada
- Cost/benefit analiza zahtevanih promena
- Komunikacija sa ostalim učesnicima o kompleksnosti datih promena.

IEEE [IEEE610.12-90] definiše maintainability kao lakoću sa kojom se softver može održavati, unapredijevati, adaptirati i ispravljati radi zadovoljavanja specificiranih zahteva. ISO/IEC definiše lakoću održavanja kao jednu od karakteristiku kvaliteta.

Podkarakteristike lakoće održavanju mogu biti specificirane, pregledane, kontrolisane tokom aktivnosti softverskog razvoja u cilju smanjenja troškova održavanja. Ako se to uradi uspešno, lakoća održavanje softvera će se unaprediti. Ovo je često teško postići zato što ove podkarakteristike nisu važne tokom procesa razvoja softvera.

Developeri su zaokupljeni sa mnogo drugih stvari i često ne obraćaju pažnju na zahteve održavaoca. Ovo može, a često se i dešava, rezultirati u nedostatku sistemske dokumentacije, koji je vodeći razlog za problem u razumevanju programa i analizi uticaja. Može se reći da prisustvo sistemskih i razvijenih procesa, tehnika i alata pomaže unapređenju lakoće održavanja sistema.

MENADŽERSKA PERSPEKTIVA ODRŽAVANJA SOFTVERA

Iako bi to moglo iznenaditi studente, održavanje softvera je jedna od najskupljih tačaka u životnom ciklusu softvera.

Ono što vole menadžeri je smanjivanje troškova. Vi često možete pomoći menadžeru preporučivanjem načina da se smanje troškovi održavanja.

Baza podataka sa zahtevima i poslovima za održavanje može da pomogne.

Na primer, sistem opisan u prethodnom poglavlju ima bazu podataka za održavanje u kojoj obično ispitivanje pokazuje da je jedan jedini modul prouzrokovao 44% problematičnih softverskih grešaka. Problemu se ušlo u trag novim dijalektom komandnog jezika koji se

kreira sa svakom novom verzijom softvera. Zamrzavanje komandnog jezika prouzrokuje male funkcionalne gubitke u pogledu korisnika, ali redukuje troškove održavanja za znatnu količinu novca. Menadžeri vole ove informacije. To čini njihov posao lakšim.

Ova vrsta analize napora na održavanju je ekstremno važan za menadžere na svim nivoima. Za najkomercijalnije organizacije, male dobiti, tokom održavanja naspram puštanja novih softvera, su planirane. Zato se troškovi održavanja minimizuju gde god je moguće.

Naravno, uvek je najvažnije pratiti uputstva za standarde i postupke svoje organizacije za procedure dokumentovanja.

ODRŽAVANJE SOFTVERA - PRIMER LAMS

Za održavanje softvera potrebno je definisati resurse koji će biti korišćeni po kasnije definisanim fazama.

- Vođa projekta Č1- Radiće na upravljanju svih aktivnosti na projektu, dobijaće izveštaje od strane softverskih inženjera i administratora sistema. Predviđeno radno vreme na projektu je 160 radnih sati. Cena rada po satu iznosi 250 dinara.
- Administrator sistema Č2 – Podešavanje softvera i hardvera u serverskom delu sistema, unos novih korisnika, nastavnih materijala kao i povezivanje korisnika sa predmetima su aktivnosti koje će raditi administrator sistema. Predviđeno radno vreme na projektu je 752 radna sata.Cena rada po satu iznosi 180 dinara.
- Softverski inženjer Č3 – Biće zadužen za vezu DITA sistema i LAMS-a, na kreiranju modela baze podataka i na planiranju modula ažuriranje materijala. Predviđeno radno vreme na projektu je 400 radnih sati.Cena rada po satu iznosi 200 dinara.
- Softverski inženjer Č4- Biće zadužen za vezu DITA sistema i LAMS-a, na kreiranju modela baze podataka i na planiranju modula ažuriranje materijala. Predviđeno radno vreme na projektu je 400 radnih sati.Cena rada po satu iznosi 200 dinara.
- Tester Č5 – Angažovanje prilikom testiranja sistema, njegovih osnovih funkcija i novih modula. Testiraće i bazu podataka sa velikim brojem unetog materijala kao i grafički interfejs sistema. Predviđeno radno vreme na projektu je 48 radnih sati. Cena rada po satu iznosi 100 dinara.

Ostali resursi:

- Server - 180.000 din
- PC 1 - 60.000 din
- PC 2 - 60.000 din
- PC 3 - 60.000 din

Tehnike estimacije održavanja: Da bismo utvrdili estimaciju napora, na raspolaganju imamo sledeće opcije [Pressman 1992]:

- estimacija kašnjenja, do kasno u projektu,
- korišćenje tehnike dekompozicije za generisanje estimacije projekta,
- razvoj empirijskog modela, ili
- sticanje automatizovanih alatki

ODRŽAVANJE SOFTVERA - PRIMER LAMS - DEKOMPOZICIJA FAZE RAZVOJA

Nakon definisanja potrebnih resursa, potrebno je podeliti održavanje softvera na određene faze.

Faza razvoja je podeljena na sledeće pod-zadatke:

- **Prva faza (Pr)** obuhvata montiranje i podešavanje razvojnog okruženja, modifikovanje i postavku servera, instalaciju operativnog sistema, softverskih alata i testiranje opreme.
- **Druga faza (De)** projekta podrazumeva analizu zahteva za softver koji su dostavljeni od strane rektora univerziteta i nastavnog kadra. Potrebno je proučiti i proveriti mogućnost ispunjenja traženih zahteva i dokumentovanje istih.
- **Treća faza (Te)** projekta se bazira na prveravanju mogućnosti novog sistema. Akcenat je na mogućnosti unosa korisnika koji je predstavljen kao jedan od najbitnijih zahteva kao i mogućnost podrške raznih media fajlova i modifikovanje i kreiranje grupa predmeta u sklopu nastavnog plana.
- **Četvrta faza (Ce)** projekta u okviru koje se prelazi na implementiranje novog e-learning sistema. Nakon prikupljanja zahteva i istraživanja potrebnih stavki dizajn aktivnost predstavlja kreiranje modifikovane baze podataka putem konceptualnog diagrama i predstavljanje članovima tima. Radi se na izradi završnog izgleda korisničkog interfejsa kao i na stranama za prikaz predavanja, koja je predstavljena kao stavka u zahtevima na kojoj je potrebno zadovoljiti kriterijume tima i studenata koji su učestvovali na izradi zahteva.
- **Peta faza (Pe)** projekta podrazumeva razvijanje sistema, preko instalacije potrebne verzije na server, implementacije nove baze podataka, kreiranja modula koji su predviđeni zahtevima i svega ostalog što je predviđeno za uraditi u ovoj fazi. Takođe, unos nastavnog plana i programa kao i materijala koji ide u sklopu svakog predmeta se podrazumevaju na kraju ove faze, kada je potrebno kompletirati i spremiti sistem za testiranje.

Ukupni napor potreban prilikom modifikovanja LAMS sistema

Ukupni napor je kompozitivna metrika koja se sastoji od zbir napora utrošenog u razvoj svih pod-zadataka:

Ukupni napor=Pr+De+Te+Ce+Pe

Pod-zadaci	Prosek utrošenih sati
Dizajn prikaza lekcije	4.7
Dodavanje audio resursa	0.1
Dodavanje grafičkog resursa	5.0
Dodavanje video resursa	5.0
Aktivnosti pitanja i odgovori	2.35
Programski kod	0.5
Testiranje	4.25

Slika 6.1.1 Prosečno vreme razvoja LAMS sistema [Izvor: NM SE321-2020/2021.]

PRIKAZ PREDVIĐENIH REZULTATA

Rezultati su dobijeni na osnovu analize procesa održavanja i faktora troškova.

Izračunavanje proseka prijavljenog napora utrošenog na razvoj LAMS-a:

Dizajn prikaza lekcije	$[D_e] = (4.7 * N) + (4.0 * \# recenzije)$
Dodavanje audio resursa	$[A_e] = (0.1 * \text{broj audio fajlova})$
Dodavanje grafičkog resursa	$[G_e] = (5.0 * \text{broj slika})$
Dodavanje video resursa	$[V_e] = (5.0 * \text{broj video segmenata})$
Aktivnosti pitanja i odgovori	$[E_e] = (2.35 * N)$
Programski kod	$[P_e] = (0.5 * N)$
Testiranje	$[T_e] = (4.25 * N)$

Slika 6.1.2 Pregled napora utrošenog na razvoj LAMS-a [Izvor: NM SE321-2020/2021.]

Slika 6.1.3 Pregled vrednosti faktora troškova [Izvor: NM SE321-2020/2021.]

Slika 6.1.4 Vrednost faktora troškova [Izvor: NM SE321-2020/2021.]

Slika 6.1.5 Procena ukupno potrošenih sati za LAMS-u [Izvor: NM SE321-2020/2021.]

REZULTATI NA OSNOVU FAZA ODRŽAVANJA SOFTVERA

Na kraju analize predstavljeni su rezultati potrebnih sati po fazama održavanja softvera.

Pod-zadatak		N sati*	Faktor troškova		I sati*
Prva faza	$(2.25 * 6) =$	13.5	* 1.575≈	21.26	25.82
Druga faza	$(0.5 * 6) =$	3.0	* 5.27≈	15.81	6.56
Treća faza	$(0.5 * 6) =$	1.5	* 0.645≈	0.9675	1.42
Četvrta faza	$(1.0 * 6) =$	6.0	* 2.66≈	15.96	8.27
Peta faza	$(1.25 * 6) =$	7.5	* 1.68≈	12.6	9.0
Testiranje	$(0.25 * 6) =$	1.5	* 0.9≈	1.35	1.35
	sabrano	33.0		67.94	52.42

N sati*=Neprilagođeni sati

I sati*=ukupni izračunati sati

Slika 6.1.6 Potrebni sati na osnovu faza održavanja [Izvor: NM SE321-2020/2021.]

Rezultat su potrošeni sati : 52.42

Metrika estimacije – LAMS

Slika 6.1.7 Formule za izračunavanje broja sati po fazama [Izvor: NM SE321-2020/2021.]

Gde je N= broj strana LAMS-a projektovana za predviđenu fazu.

✓ 6.1 Individualna vežba

ODRŽAVANJE SOFTVERA

Na primeru Zimbra sistema potrebno je razviti plan održavanja softvera.

Svaki student treba da na primeru nekog od modula ISUM sistema, izradi plan održavanja i izvrši analizu efikasnosti i troškova održavanja na bazi dostupne dokumentacije.

Održavanje predstavlja fazu koja se ponavlja u okviru životnog veka jednog softvera.

1. Izraditi Plan održavanja odabranog softvera koji podrazumeva prikaz članova tima, definisanje faza održavanja softvera i broj časova po definisanim fazama. **(vreme izrade 45 minuta)**
2. Navesti broj članova tima, faze i broj potrebnih časova rada na osnovu kreiranih faza. **(vreme izrade 45 minuta)**

✓ Poglavlje 7

Domaći zadatak

JEDANAESTI DOMAĆI ZADATAK

Nakon jedanaeste lekcije potrebno je uraditi jedanaesti domaći zadatak.

Za odabranu proizvoljnu aplikaciju koju ste razvili na nekom od predmet uraditi sledeće:

1. Prikazati model estimacije troškova prilikom održavanja odabrane aplikacije
2. Prikazati model karakteristika kvaliteta softvera za odabranu aplikaciju

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnog okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ11-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ11.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Vrste i kategorije održavanja SW

AKTIVNOSTI, VRSTE I KATEGORIJE ODRŽAVANJA SW

Treba biti svestan da je situacija naročito složena za održavanje ako je softver koji treba održavati u vezi sa jeftinim komponentama hardvera čija je unutrašnja struktura nepoznata.

Ljudske sposobnosti primanja informacija za naša čula su ograničene, a povećani pritisak izaziva nervozu i bes. Mnoga od novijih tehnoloških unapređenja namenjena radu u kancelariji, ignorisu logiku ljudskog ponašanja.

Održavanje kvaliteta softverskog proizvoda i sistema, kao i rešenje informacijske krize ima ekonomske i društveno-humanističku važnost koja zahteva radikalnu promenu svesti. Ako se ne prate dostignuća nauke i tehnologije, povećava se verovatnoća donošenja pogrešnih odluka što može imati pogubne posledice za pojedine ljudske delatnosti i celokupnu društvenu zajednicu.

Dobar sistem sam sebe održava. Da bi se došlo na takav organizacijski nivo, potrebno je znanje o njegovim životnim ciklusima i znanja koja nam omogućuju da takva znanja merimo, simuliramo i predviđamo.

Može se očekivati da će značaj održavanja i unapređivanja softverskih sistema u budućnosti još više rasti, i da će cena ovih procesa višestruko prevazići cenu razvoja. Najveći deo prihoda softverskih kompanija dolaziće od održavanja i podrške za postojeće sisteme, dok će sam inicijalni proizvod biti sve jeftiniji.

Postoji još jedan problem koji naročito uznemirava softverske servisere. Mnogi softverski inženjeri, naročito oni sa prvobitnim iskustvom sa kompjutera koji su imali veliku količinom fizičke memorije, ne misle da njihovi sadašnji sistemi imaju ikakva ograničenja. Naime, memorija može biti potrošena naročito ako je sastavljena od jeftinih proizvoda čiji su memorijski zahtevi nepoznati.

Treba biti svestan da je situacija naročito složena za održavanje ako je softver koji treba održavati u vezi sa jeftinim komponentama hardvera čija je unutrašnja struktura nepoznata.

Postoji više vrsta aktivnosti koje nazivamo održavanjem:

- **Održavanje u smislu ispravke softverskih grešaka**
- **Održavanje u smislu prilagođavanja softvera različitim operativnim okruženjima**
- **Održavanje u smislu dodavanja i modifikacije sistemskih funkcionalnosti.**

LITERATURA ZA LEKCIJU 11

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. Anne Mette Jonassen Hass, Guide to Advanced Software Testing, © 2008 ARTECH HOUSE, INC.
4. V. B. Spasojević, i ostali, Organizacija sistema kvaliteta i alati kvalitet, Industrija 2004, vol. 32, br. 4, str. 91-107

Dopunska literatura:

1. Samir Lemeš, Nevzudin Buzađija, STANDARDNI MODELI ODRŽAVANJA SOFTVERA, 5. Konferencija „ODRŽAVANJE - MAINTENANCE 2018“ Zenica, B&H, 10. – 12. Maj 2018.
2. Lj. Lazić , Software Quality & Testing Metrics, WSEAS 7th WSEAS EUROPEAN COMPUTING CONFERENCE (ECC '13), Dubrovnik, Croatia, June 25-27, 2013 . (link: <http://www.wseas.org/wseas/cms.action?id=4102>

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.pisa.rs>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Analiza problema, izbor tehnika
održavanja

Lekcija 12

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 12

ANALIZA PROBLEMA, IZBOR TEHNIKA ODRŽAVANJA

- ✓ Analiza problema, izbor tehnika održavanja
- ✓ Poglavlje 1: Aktivnosti održavanja softvera
- ✓ Poglavlje 2: Tehnike održavanja softvera
- ✓ Poglavlje 3: Reverzni (obrnuti) inženjering softverskog sistema
- ✓ Poglavlje 4: Reinženjering softverskog sistema
- ✓ Poglavlje 5: Planiranje održavanja softvera
- ✓ Poglavlje 6: Grupna vežba
- ✓ Poglavlje 7: Individualna vežba
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Zaključak 12

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

Održavanje informacionog sistema, a posebno softvera, je teško. Imajući u vidu da je sistem već u eksploataciji, tim za održavanje uravnotežuje potrebu za izmenom sa potrebom da sistem ostane dostupan korisnicima. Postoji mnogo personalnih i organizacionih razloga koji održavanje čine teškim. Osoblje mora da deluje kao posrednik između problema i njegovog rešenja, popravljujući i krojeći softver da bi obezbedilo da rešenje prati problem kako se on menja.

Pored uravnotežavanja korisničkih potreba sa potrebama softvera i hardvera, tim za održavanje ima posla i sa ograničenjima ljudskog razumevanja. Postoji granica brzine kojom neko može da proučava dokumentaciju i iz nje izvlači materijal relevantan za problem koji se rešava. Štaviše, mi često tražimo više ključeva nego što je stvarno potrebno za rešenje problema.

U mnogim organizacijama koje koriste glomazan softver, održavanje tih sistema predstavlja pravi izazov. Organizacije koje su u takvoj situaciji moraju da donose teške odluke o tome kako da učine svoje sisteme pogodnijim za održavanje. Njihov izbor se kreće u opsegu od poboljšanja do potpune zamene novom tehnologijom. Svaki izbor treba da sačuva ili poveća kvalitet softvera, uz istovremeno zadržavanje troškova na što je moguće nižem nivou.

U predavanju je fokus stavljen na tehnike održavanja softvera, gde spadaju redokumentovanje i restrukturiranje sistema, kao i reverzni (obrnuti) inženjering i reinženjering sistema.

▼ Poglavlje 1

Aktivnosti održavanja softvera

RAZLOZI ZA ODRŽAVANJE SOFTVERA

Neki od najčešćih razloga za održavanje softera su: promena tržišnih uslova, zahtevi klijenata, modifikacija hosta, promene u organizaciji

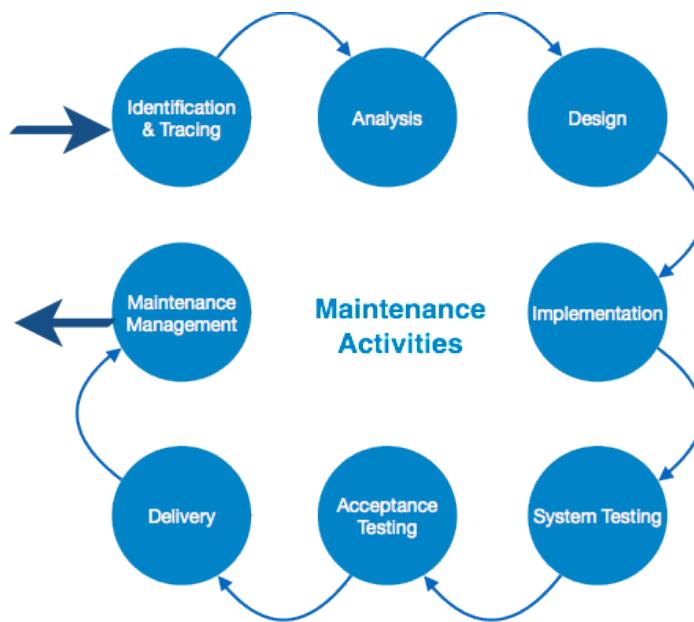
Održavanje softvera je danas kao što je poznato široko prihvaćen deo životnog ciklusa razvoja softvera. Označava sve modifikacije i nadogradnje izvršene nakon isporuke softverskog proizvoda. Kao što je već rečeno, postoje brojni razlozi zašto su modifikacije potrebne, a neki od njih su:

- **Tržišni uslovi** - propisi koji se vremenom menjaju, poput novog načina oporezivanja i novouvedenih ograničenja, načina vođenja knjigovodstva, mogu pokrenuti potrebu za izmenama.
- **Zahtevi klijenta** - Vremenom kupac može zatražiti nove funkcije ili karakteristike softvera.
- **Modifikacije hosta** - Ako se bilo koji od hardvera i / ili platforme (kao što je operativni sistem) hosta promeni, potrebne su promene softvera da bi se zadržala njegov prilagodljivost.
- **Promene u organizaciji** - Ako na strani klijenta dođe do bilo kakvih promena na poslovnom nivou, kao što je slabljenje organizacije, otvaranje nove kompanije, otvaranje novog posla, može se pojaviti potreba za izmenama u originalnom softveru.

AKTIVNOSTI ODRŽAVANJA SOFTVERA PREMA IEEE

IEEE je definisao okvir za sekvenčalne aktivnosti procesa održavanja koje se mogu proširiti kastomizovanim predmetima i procesima.

IEEE je definisao okvir za sekvenčalne aktivnosti procesa održavanja. Okvir se može koristiti na iterativni način, a takođe se može proširiti tako da se uključe novi kastomizovani predmeti i procesi.



Slika 1.1 Aktivnosti održavanja Izvor: https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm [Izvor: NM SE321-2020/2021.]

Ove aktivnosti idu ruku pod ruku sa svakom od sledećih faza:

- **Identifikacija i praćenje** - uključuje aktivnosti koje se odnose na identifikaciju zahteva za modifikacijom ili održavanjem. Generišu ih korisnici ili sam sistem putem dnevnika iz log file-ova ili poruka o greškama. Ovde treba navesti i tip održavanja.
- **Analiza** - Zahtevane modifikacije se analiziraju zbog eventualnog uticaja na ostatak sistema, uključujući bezbednosne i sigurnosne implikacije. Ako je mogući uticaj ozbiljan, traži se alternativno rešenje. Skup potrebnih modifikacija se zatim materijalizuje u specifikacije zahteva. Troškovi modifikacije / održavanja se analiziraju i donosi se procena.
- **Dizajn** - Novi moduli, koje treba zameniti ili modifikovati, dizajnirani su prema specifikacijama zahteva postavljenim u prethodnoj fazi. Test slučajevi se koriste za validaciju i verifikaciju.
- **Implementacija** - Novi moduli se kodiraju uz pomoć projekta kreiranog u koraku dizajniranja. Očekuje se da će svaki programer paralelno vršiti jedinično testiranje.
- **Sistemsko testiranje** - Među novokreiranim modulima se vrši integraciono testiranje. Testiranje integracije se takođe vrši između novih modula i ostatka sistema. Konačno, sistem se testira u celini, prateći postupke regresivnog testiranja.

ZAVRŠNE AKTIVNOSTI ODRŽAVANJA SOFTVERA PREMA IEEE

To su: Testiranje prihvatljivosti, isporuka i upravljenje održavanjem

- **Testiranje prihvatljivosti** - Nakon internog testiranja, sistem se testira na prihvatljivost uz pomoć korisnika. Ako se u tom trenutku, korisnik žali na neke probleme oni se rešavaju, ili se beleži da ih treba rešiti u sledećoj iteraciji.

- **Isporuka** - Nakon testa prihvatljivosti, sistem se postavlja u organizaciji ili korišćenjem paketa za ažuriranje postojećeg sistema ili novom instalacijom sistema. Završno testiranje se odvija na lokaciji klijenta nakon isporuke softvera.
Ako je potrebno, uz štampani primerak korisničkog priručnika obezbeđuje se i obuka.
- **Upravljanje održavanjem** - Upravljanje konfiguracijom je suštinski deo održavanja sistema. Za to se mogu koristiti različite vrste alata za kontrolu verzija (**version control tools**).

POGODNOST ZA ODRŽAVANJE KAO KARAKTERISTIKA KVALITETA SOFTVERA

Pogodnost za održavanje karakterišu sledeći atributi: mogućnost analize, mogućnost promena, stabilnost, mogućnost testiranja, mogućnost usaglašavanja sa standardima

Pogodnost za održavanje (**Maintability**) softvera karakterišu sledeći atributi:

- Mogućnost analize (**analysability**)
- Mogućnost promena (**changeability**)
- Stabilnost (**stability**)
- Mogućnost testiranja (**testability**)
- Mogućnost usaglašavanja sa standardima (**maintainability compliance**)

Ključna pitanja na koja treba pripremiti odgovor da bi se zadovoljili atributi za održavanje su:

- Da li se sistemske greške mogu lako utvrditi?
- Da li je sistem jednostavan za izmene?
- **Da li sistem može nastaviti sa funkcionisanjem tokom izmena?**
- Da li je omogućeno lako testiranje softvera?

Prilikom davanja odgovora voditi računa da je održavanje softvera proces modifikovanja softverskog sistema ili komponente nakon isporuke radi otklanjanja grešaka, poboljšanja performansi ili drugih atributa, ili prilagođavanja promenljivom okruženju.

Softverski proizvod se podvrgava modifikaciji koda ili dokumentacije usled problema ili potrebe za unapređenjem. Cilj je modifikovanje postojećeg softverskog proizvoda dok se istovremeno čuva njegov integritet.

Održavanje obično ne obuhvata značajne izmene arhitekture sistema, jer se izmene implementiraju modifikovanjem postojećih komponenti i dodavanjem novih komponenti sistemu.

Treba uzeti u obzir da su osnovni faktori koji utiču na cenu održavanja:

- **Celovitost polazne specifikacije** - ukoliko odmah uključimo sve zahteve, kasnije će biti manje perfekcionog održavanja.

- **Kvalitet dizajna** – dobar dizajn je jeftiniji za održavanje. Smatra se da su sa stanovišta održavanja najbolji objektno-orientisani sistemi, koji se sastoje od malih modula sa jakom unutrašnjom kohezijom i labavim vezama prema spolja.
- **Način implementacije** – Kod u “strožem” programskom jeziku poput Jave lakše se održava nego kod u jeziku poput C-a. Strukturirani kod (if, while) sa smisleno imenovanim varijablama razumljiviji je od kompaktnog koda s mnogo goto naredbi.
- **Stepen verifikovanosti** – dobro verifikovani softver ima manje grešaka pa će zahtevati manje korekcionog održavanja.

ADAPTACIJA SOFTVERA

Adaptacija softvera podrazumeva skup aktivnosti koje obezbeđuju da softver nastavi da ispunjava organizacione i poslovne ciljeve na isplativ način

Na atribute održavanja može uticati i evolucija (adaptacija) softvera jer evolucija podrazumeva:

- **Skup aktivnosti koje obezbeđuju da softver nastavi da ispunjava organizacione i poslovne ciljeve na isplativ način**
- **Skup aktivnosti čiji je cilj generisanje nove verzije softvera na osnovu starije operacione verzije**
- **Primenu aktivnosti i procesa održavanja softvera kojima se generiše nova operativna verzija softvera sa novim funkcionalnostima ili karakteristikama u odnosu na prethodnu operativnu verziju pri čemu su obuhvaćene i aktivnosti obezbeđivanja kvaliteta.**

Najveći deo budžeta za razvoj softvera u velikim kompanijama je u mnogo većoj meri namenjen adaptaciji (evoluciji) postojećeg softvera, a u manjoj razvoju novog softvera.

Osnovni razlozi za evoluciju softvera su:

- **Promene korisničkih zahteva**
- **Proširenja ili modifikacije koje je zahtevao korisnik**
- **Otklanjanja bagova**

Planirane aktivnosti otklanjanja problema:

- **Nužno otklanjanje** (uglavnom skupo zbog velikog pritiska)
- **Promene formata podataka**
 - > Y2K, Euro, poreske rate, poštanski kodovi, telefonski brojevi, ...
 - > Novi standardi: UML, XML, COM, DCOM, CORBA, ActiveX, WAP
- **Hardverske promene**
- **Povećanje efikasnosti**

Programi, kao i ljudi stare. Starenje ne možemo sprečiti, ali možemo razumeti njegove uzroke, preduzeti korake da ograničimo njegove efekte, privremeno otklonimo štetu koju je izazvalo i pripremimo se za dan kada softver više neće biti upotrebljiv. Razlozi za starenje softvera su:

- **Održavanje**
- **Nefleksibilnost od početka projekta**
- **Nedovoljna ili nekonzistentna dokumentacija**
- **Pritisak krajnjih rokova**
- **Dupliranje funkcionalnosti (dupliranje koda)**
- **Nedostatak modularnosti itd.**

Moguće rešenje: **reinženjerstvo**.

▼ Poglavlje 2

Tehnike održavanja softvera

PODMLAĐIVANJE SOFTVERA

Podmlađivanje softvera izazov da sisteme učini pogodnjim za održavanje, rešava tako što pokušava da poveća ukupan kvalitet postojećeg sistema.

U mnogim organizacijama koje koriste glomazan softver održavanje tih sistema predstavlja pravi izazov.

Npr. uzmimo osiguravajuće društvo koje nudi novu vrstu životnog osiguranja. Da bi podržalo svoj proizvod, društvo razvija softver za rad sa polisama osiguranja, informacijama o vlasnicima polisa i finansijskim i računovodstvenim informacijama. Takve polise treba da se održavaju desetinama godina. Rezultat svega toga je da osiguravajuće društvo verovatno treba da podržava mnogo različitih aplikacija na različitim platformama, sa velikim brojem implementacionih jezika.

Organizacije koje su u takvoj situaciji, moraju da donose teške odluke o tome kako da učine svoje sisteme pogodnjim za održavanje. Njihov izbor se kreće u opsegu od poboljšanja do potpune zamene novom tehnologijom. Svaki izbor treba da sačuva ili poveća kvalitet softvera, uz istovremeno zadržavanje troškova na što je moguće nižem nivou.

Podmlađivanje softvera odgovara na izazov održavanja pokušavajući da poveća ukupan kvalitet postojećeg sistema. Ono se vraća proizvodima rada sistema pokušavajući da dobije dodatne informacije ili da ih reformatira na razumljiviji način. Postoji više aspekata softverskog podmlađivanja koje uključuje sledeće tehnike održavanja:

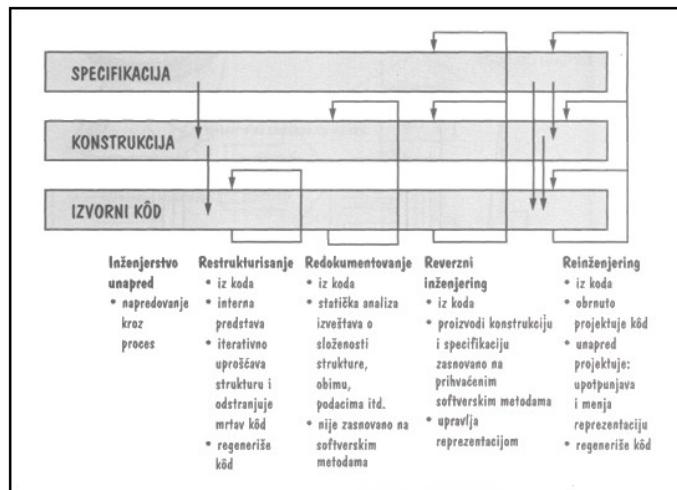
- Redokumentovanje
- Restrukturisanje
- Reverzni inženjering
- Reinženjering.

TEHNIKE ODRŽAVANJA SOFTVERA

To su: redokumentovanja sistema, restruktuisanja sistema, reverzni inženjering sistema i reinženjering

Kod **redokumentovanja sistema**, izvodimo statičku analizu izvornog koda proizvodeći dodatne informacije da bismo pomogli serviserima u razumevanju i referenciranju koda. Analiza ne čini ništa da bi transformisala aktuelni kod. Ona samo proizvodi informacije.

Međutim, kod **restruktuisanja sistema**, stvarno menjamo kod, transformišući loše strukturisani kod u dobro strukturisani. Obe ove tehnike bave se samo izvornim kodom.



Slika 2.1 Taksonomija podmlađivanja softvera [Izvor: NM SE321-2020/2021.]

Da bismo izvršili **reverzni inženjerstvo sistema**, vraćamo se od izvornog koda ka proizvodima koji su mu prethodili, ponovo stvarajući informacije o konstrukciji i specifikaciji iz koda.

Reinženjerstvo je još obuhvatniji jer se obavlja reverzni inženjerstvo postojećeg sistema koji se zatim „projektuje unapred“ radi izmena u specifikaciji i konstrukciji koje upotpunjaju logički model. Zatim se stvara novi sistem na osnovu revidirane specifikacije i konstrukcije. Na slici 1 ilustrovani su odnosi između četiri vrste podmlađivanja.

Stepen u kome informacija može da se izdvoji iz finalnog proizvoda zavisi od više činilaca:

1. jezika koji se koriste;
2. interfejsa baze podataka;
3. korisničkih interfejsa;
4. interfejsa ka sistemskim uslugama;
5. interfejsa ka drugim jezicima;
6. zrelosti i stabilnosti domena;
7. raspoloživih alata.

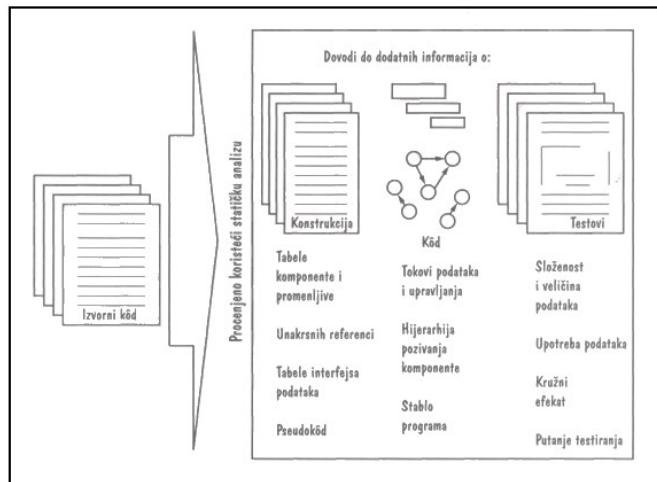
Sposobnost servisera, njihovo znanje i iskustvo takođe igraju važnu ulogu kada je reč o stepenu u kome informacije mogu uspešno da se interpretiraju i koriste.

REDOKUMENTOVANJE SOFTVERSKOG SISTEMA

Redokumentovanje obuhvata statičku analizu izvornog kôda da bi se proizvela dokumentacija sistema.

Redokumentovanje obuhvata statičku analizu izvornog koda da bi se proizvela dokumentacija sistema.

Možemo da ispitamo upotrebu promenljivih, pozive komponenti, putanje upravljanja, veličinu komponenti, pozivajuće parametre, putanje testiranja i druge odnosne mere koje nam pomažu da razumemo šta i kako kod radi. Informacije koje proizvede statička analiza koda mogu da budu grafičke ili tekstualne. Na slici 2 ilustrovan je proces redokumentovanja.



Slika 2.2 Redokumentovanje [Izvor: NM SE321-2020/2021.]

Uobičajeno je da serviser počinje redokumentovanje analizirajući kod analitičkim alatom. Izlaz može da uključi:

- pozivne odnose između komponenti;
- hijerarhije klasa;
- informacije o rečniku u vezi s podacima;
- tabele ili dijagrame toka podataka;
- tabele ili dijagrame toka upravljanja;
- pseudokod;
- putanje testiranja;
- unakrsno referenciranje komponenti i promenljivih.

Grafičke, tekstualne i tabelarne informacije mogu da se upotrebue za procenu da li sistem zahteva ili ne zahteva restrukturisanje. Međutim, kako nema preslikavanja između specifikacije i restrukturisanog koda, rezultujuća dokumentacija odražava ono što jeste, a ne ono što bi trebalo da bude.

RESTRUKTUISANJE SISTEMA

To je proces restrukturiranja i rekonstrukcije postojećeg softvera. Sastoji se u reorganizaciji izvornog koda, bilo u istom programskom jeziku ili iz jednog programskog jezika u drugi.

To je proces restrukturisanja i rekonstrukcije postojećeg softvera. Sastoji se u reorganizaciji izvornog koda, bilo u istom programskom jeziku ili iz jednog programskog jezika u drugi.

Restrukturiranje se može odnositi ili na restrukturiranje izvornog koda, ili na restrukturiranje podataka, ili i na jedno i na drugo. **Restrukturiranje ne utiče na funkcionalnost softvera, ali poboljšava pouzdanost i održivost.**

Restrukturiranjem se :

- mogu promeniti ili ažurirati programske komponente koje vrlo često uzrokuju greške
- može se prevazići zavisnost softvera od zastarele hardverske platforme

▼ Poglavlje 3

Reverzni (obrnuti) inženjering softverskog sistema

ŠTA JE REVERZNI (OBRNUTI) INŽENJERING SOFTVERSKOG SISTEMA?

Reverzni (obrnuti) inženjering se definiše kao "proces analiziranja predmeta sistema u cilju identifikacije sistemskih komponenti, njihovih međusobnih odnosa i kreiranja prezentacije sistema"

Reverzni (obrnuti) inženjering se definiše kao "proces analiziranja predmeta sistema u cilju identifikacije sistemskih komponenti, njihovih međusobnih odnosa i kreiranja prezentacije sistema u drugom obliku ili na višem nivou apstrakcije". U skladu sa tim, reverzni inženjering je proces ispitivanja, nije proces promena i zato ne uključuje menjanje softvera prilikom ispitivanja. Iako je softverski reverzni inženjering nastao u okviru softverskog održavanja, on se odnosi na probleme i u mnogim drugim oblastima. Chikofsky i Cross su identificovali šest ključnih ciljeva reverznog inženjeringa:

1. **suočavanje sa kompleksnošću,**
2. **generisanje alternativnih pogleda,**
3. **vraćanje izgubljenih podataka,**
4. **otkrivanje sporednih pojava,**
5. **sinteza viših apstrakcija i**
6. **omogućavanje ponovnog korišćenja.**

REVERZNI (OBRNUTI) INŽENJERING - STANDARD IEEE-1219

Standard IEEE-1219 preporučuje reverzni inženjering kao ključnu tehnologiju za podršku, sa zadatkom da se bavi sistemima čiji je izvorni kod jedina pouzdana predstava.

Standard IEEE-1219 preporučuje reverzni inženjering kao ključnu tehnologiju za podršku, sa zadatkom da se bavi sistemima čiji je izvorni kod jedina pouzdana predstava.

Primeri problema u oblastima gde je reverzni inženjering uspešno primenjen uključuju **identifikovanje elemenata za ponovno korišćenje, pronalaženje objekata u proceduralnim programima, pronalaženje arhitekture, izvođenje konceptualnih podataka modela, otkrivanje duplikata, transformisanje binarnih programa u**

izvorni kod, obnavljanje korisničkih interfejsa, paralelizovanje sekvenčnih programa, prevodenje i migracije.

Reverzni inženjering je teško definisati kao proces u strogim uslovima, jer je to nova oblast koja se veoma brzo razvija. Tradicionalno gledano, reverzni inženjering je posmatran kao proces iz dva koraka: ekstrakcija informacija i apstrakcija. Ekstrakcija informacija analizira sistemski artifakte, prvenstveno izvorni kod, radi prikupljanja nesređenih podataka, dok apstrakcija informacije kreira korisnički-orientisana dokumenta i poglede.

Autori Tilley i Paul predlažu preliminaran korak koji se sastoji od izgradnje modela za specifične domene sistema, koristeći konceptualne tehnike modelovanja.

EVOLUIRANJE PROCESA REVERZNOG INŽENJERINGA

IEEE standard za softversko održavanje ukazuje da se proces reverznog inženjeringa evoluira kroz šest koraka.

IEEE standard za softversko održavanje ukazuje da se proces reverznog inženjeringa evoluira kroz šest koraka:

1. raščlanjavanje izvornog koda u formalne jedinice;
2. semantički opis formalne jedinice i kreiranje funkcionalnih jedinica;
3. opis linkova za svaku jedinicu (dijagram ulazno / izlaznih jedinica);
4. kreiranje mape svih jedinica i onoga što sledi od uzastopno povezanih jedinica (linearni ciklusi);
5. deklaracije i semantički opis sistemskih aplikacija
6. stvaranje anatomije sistema.

Prva tri koraka se bave lokalnom analizom na nivou jedinica (u malom), dok se ostala tri koraka odnose na globalnu analizu na sistemskom nivou (u velikom).

Benedusi zagovara potrebu za uvođenjem visokog nivoa organizacionog obrasca prilikom postavljanja složenih procesa u oblasti, kao što je reverzni inženjering, u kojem metodologija i alati nisu stabilni, ali se stalno razvijaju. Uloga takvog obrasca nije samo da definiše okvir u kojem se mogu koristiti dostupne metode i alati, nego i da dopusti ponavljanje procesa u cilju boljeg razumevanja istih i učenja iz njih.

Predložen je obrazac, nazvan Ciljevi/Modeli/Alati, koji deli postavljanje procesa reverznog inženjeringa u tri sekvenčne faze: Ciljevi, Modeli i Alati.

- **Ciljevi** - Ovo je faza u kojoj se analizira motivacija za postavljanje procesa kako bi se identifikovale potrebe za informacijama i apstrakcije koje treba uraditi.
- **Modeli** - Ovo je faza u kojoj se apstrakcije identifikovane u prethodnoj fazi, analiziraju kako bi se definisali modeli za prihvatanje informacija potrebnih za njihovu izradu.
- **Alati** - Ovo je faza za utvrđivanje, uzimanje, jačanje, integraciju ili izgradnju:
 - **Ekstrakcionih alata i procedura**, koji služe za ekstrakciju (vađenje) sistemskih grešaka iz nesređenih podataka potrebnih za instalaciju modela definisanog u fazi modela,

- Apstrakcionih alata i procedura, koji služe za transformaciju programskog modela u apstraktioni model definisanog u fazi "Ciljevi".

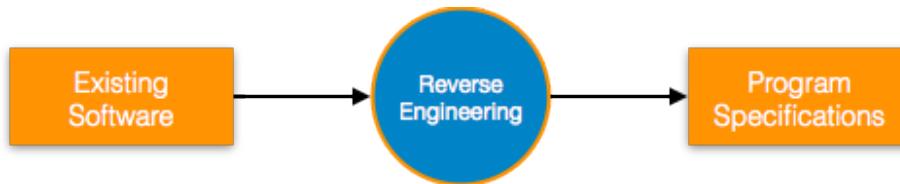
Obrazac Ciljevi/Modeli/Alati se opsežno koristi za definisanje i izvršavanje nekoliko realnih procesa reverznog inženjeringu.

PRIMER REVERZNOG INŽENJERINGA

Reverzni inženjering se može posmatrati kao obrnuti SDLC model, tj. pokušavamo da dobijemo viši nivo apstrakcije analizom nižih nivoa apstrakcije.

Kao što je rečeno, **reverzni inženjering je proces koji omogućava da se temeljnom analizom dobije specifikacija sistema, razumevanjem postojećeg sistema.** Ovaj proces se može posmatrati kao obrnuti SDLC model, tj. pokušavamo da dobijemo viši nivo apstrakcije analizom nižih nivoa apstrakcije.

Postojeći sistem je prethodno implementiran, ali o njemu ne znamo ništa. Projektanti vrše obrnuti inženjering gledajući kod i pokušavajući da iz koda dobiju projekat (dizajn). S projektom u ruci, pokušavaju da donesu zaključak o specifikacijama sistema. Dakle, radi se o obrnutom prelasku od koda prema specifikaciji sistema.



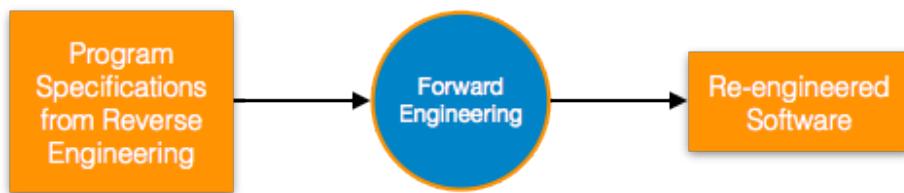
Slika 3.1 Postupak reverznog inženjeringu Izvor: https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm [Izvor: NM SE321-2020/2021.]

INŽENJERSTVO UNAPRED

Inženjerstvo unapred (Forward engineering) je proces dobijanja željenog softvera iz specifikacija koje su dobijene pomoću reverznog inženjeringu.

Inženjerstvo unapred (Forward engineering) je proces dobijanja željenog softvera iz specifikacija koje su dobijene pomoću reverznog inženjeringu. Prepostavlja se da je neki softverski inženjering već urađen u prošlosti.

Inženjerstvo unapred je isto što i proces softverskog inženjerstva sa samo jednom razlikom - izvodi se uvek nakon reverznog inženjeringu.



Slika 3.2 Inženjerstvo unapred Izvor: https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm [Izvor: NM SE321-2020/2021.]

▼ Poglavlje 4

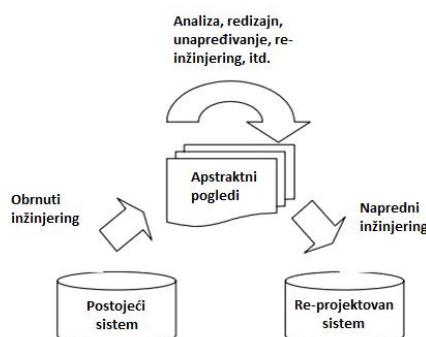
Reinženjering softverskog sistema

ZAŠTO REINŽENJERING SOFTVERSKOG SISTEMA?

Praksa reinženjeringu softverskog sistema doprinosi boljem razumevanju tog softvera i njegovom održavanju, dugo je već prihvaćena od strane zajednice softverskog održavanja.

Praksa reinženjeringu softverskog sistema doprinosi boljem razumevanju tog softvera i njegovom održavanju, dugo je već prihvaćena od strane zajednice softverskog održavanja. Autori Chikofsky i Cross definišu **reinženjering** kao "ispitivanje i izmenu predmeta sistema u cilju rekonstrukcije u novu formu, a zatim i implementaciju tih novih formi". Arnold, daje sveobuhvatnu definiciju: "**Softverski Reinženjering je aktivnost koja:**

1. **poboljšava razumevanje softvera, ili**
2. **priprema i poboljšava sam softver, obično za povećanu sposobnost održavanja, ponovno korišćenje, ili evoluciju.**



Slika 4.1 Reverzni inženjering i reinženjering [Izvor: NM SE321-2020/2021.]

"Evidentno je da reinženjering podrazumeva neki oblik reverznog inženjeringa za kreiranje više apstraktног pogledа sistema, a regeneracija ovog apstraktног pogledа sledi iz naprednih inženjerskih aktivnosti za realizaciju sistema u novom obliku". Ovaj proces je prikazan na slici 1.

Softverski reinženjering se pokazao kao važan iz nekoliko razloga. Britcher, identificuje sedam glavnih razloga koji pokazuju relevantnost reinženjeringu:

1. Reinženjering može pomoći pri smanjenju rizika evolucije
2. Reinženjering može pomoći organizaciji pri nadoknadi svojih ulaganja u softver
3. Reinženjering može doprineti lakšem menjanju softvera
4. Reinženjering je veliki biznis

5. Reinženjering sposobnosti proširuju CASE alate
6. Reinženjering je katalizator za automatsko održavanje softvera
7. Reinženjering je katalizator za primenu tehnika veštačke inteligencije

Primeri scenarija u kojima se reinženjering pokazao korisnim uključuje migraciju sistema s jedne na drugu platformu, smanjenje veličine, prevodenje, smanjenje troškova održavanja, poboljšanje kvaliteta, migraciju i reinženjering podataka.

OD ČEGA ZAVISI REINŽENJERING SOFTVERSKOG SISTEMA?

Neki primeri aktivnosti koje doprinose uspehu reinženjeringu su: definisani ciljevi reinženjeringu, formiranje tima i njihova obuka, stepen do kojeg se alati mogu integrisati

Standard IEEE-1219 pokazuje da reinženjering ne samo što može revitalizovati sistem, već i pruža koristan materijal za budući razvoj, uključujući okvire za objektno-orientisana okruženja.

Softverski reinženjering je složen proces koji reinženjering alati mogu samo da podrže, ali i ne da potpuno automatizuju. Postoji i dobar deo ljudske intervencije u bilo kom softverskom reinženjering projektu.

Uspeh softverskog reinženjeringu zahteva mnogo više od kupovine jednog ili više reinženjerskih alata. Definisanje reinženjerskih ciljeva, formiranje tima i njihova obuka, procena stepena do kojeg se alati mogu integrisati, samo su neki primeri aktivnosti koje doprinose određivanju uspeha reinženjeringu projekta. Sneed predlaže pet koraka koje treba uzeti u obzir prilikom planiranja reinženjeringu projekta: opravdanost projekta, portfolio analizu, procenu troškova, analizu koristi i troškova i ugovaranje.

Da bi se ispravno shvatilo, gde i kako nastaju greške u projektovanju i izradi softvera, a što značajno utiče na proces održavanja softvera koji treba na optimalan način uspostaviti i sprovoditi, u narednom poglavlju biće opisani elementi jednog modela procesa održavanja softvera.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER REINŽENJERINGA

Reinženjering softvera se vrši kada je potrebno da se softver ažurira kako bi ga zadržali na trenutnom tržištu, bez uticaja na njegovu funkcionalnost

Kao što je rečeno, kada je potrebno da se softver ažurira kako bi ga zadržali na trenutnom tržištu, bez uticaja na njegovu funkcionalnost, to se naziva reinženjering softverskog sistema. To je temeljni proces gde se dizajn softvera menja i programi ponovo pišu.

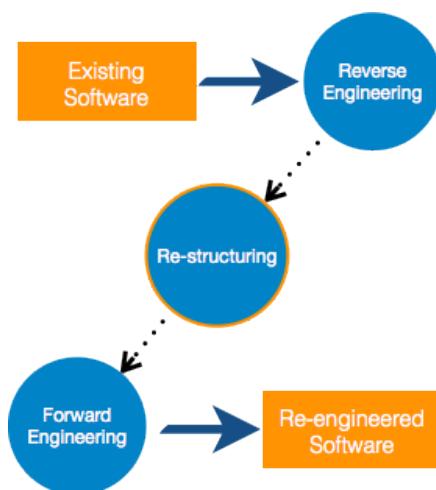
Na primer, nasleđeni softveri (**Legacy software**) ne mogu uvek da se usaglase sa najnovijom tehnologijom dostupnom na tržištu. Kako hardver zastareva, ažuriranje takvih softvera postaje glavobolja. Čak i ako softver vremenom stari, njegova funkcionalnost to ne čini.

Kao primer može da posluži Unix, koji je u početku bio razvijen na asemblerском jeziku. Kada je nastao jezik C, Unix je prerađen u C, jer je rad na asemblerском jeziku bio težak.

Osim ovoga, ponekad programeri primete da je nekim delovima softvera potrebno više održavanja nego drugima, a takođe da im je potreban i reinženjering.

Proces reinženjeringu se sastoji od nekoliko koraka:

- **Odlučiti šta će se podvrgnuti reinženjeringu. Da li je to ceo softver ili njegov deo?**



Slika 4.2 Primer reinženjeringu Izvor: https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm [Izvor: NM SE321-2020/2021.]

- **Izvršiti reverzni inženjering kako bi se dobile specifikacije postojećeg softvera.**
- **Ako je potrebno, izvršiti restrukturiranje softvera. Na primer, promenu programa orijentisanih na funkcije u objektno orijentisane programe.**
- **Prestrukturirati podatke prema potrebi.**
- **Primeniti koncepte inženjerstva unapred (**Forward engineering**) kako bi se dobio redizajnirani softver.**

✓ Poglavlje 5

Planiranje održavanja softvera

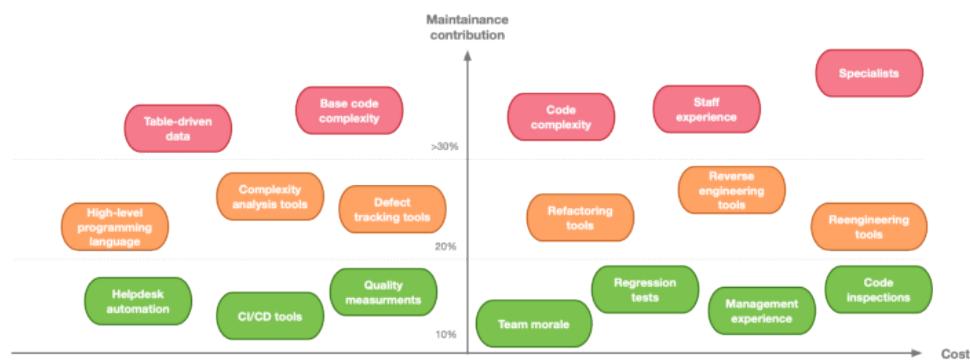
KADA PLANIRATI ODRŽAVANJE SOFTVERA?

Ne treba dozvoliti da se planiranje održavanja softvera dogodi suviše kasno tokom izrade softverskog projekta.

Ne treba dozvoliti da se planiranje održavanja softvera dogodi suviše kasno tokom izrade softverskog projekta. **Planiranje održavanja softvera zahteva ne samo razmišljanje o aktivnostima koje bi se trebale odvijati tokom perioda održavanja, već obezbeđenje svih ljudi i potrebnih resursa kako bi oni bili dostupni kada je to potrebno.** Projektovanje (dizajn) softverskog sistema ima integralnu ulogu u njegovom budućem državanju. Definisanje arhitekture, postavljanje razvojnih ciljeva i standarda koje treba primenjivati prilikom projektovanja način razmeštanja sistema (**deployment**) mogu pozitivno ili negativno doprineti budućem održavanju. Dužnost arhitekte sistema je da održavanje smatra delom originalnih zahteva sistema.

Postoji mnogo toga što se može uraditi, projektovati i pripremiti kako bi se osiguralo neometano održavanje nakon što sistem počne sa radom. Kao što se vidi na slici 1, različite stvari mogu doprineti održavanju vašeg sistema.

Kada prilikom projektovanja softvera vodite računa o njegovom održavanju, treba izvršiti procenu trenutne situacije što se tiče ljudi, njihovih veština i sposobnosti, infrastrukture i kulture. Utvrdite šta treba popraviti, promeniti ili poboljšati da biste isporučili softver na koji ste ne samo ponosni u trenutku isporuke već i u godinama koje dolaze dok ga održavate.



Slika 5.1 Elementi koji utiču na održavanje softvera tokom njegovog projektovanja Izvor:
<https://medium.com/swlh/types-of-software-maintenance-2b0503848b43>

PLAN ODRŽAVANJA SOFTVERA

Plan održavanja softvera treba dokumentovati u dokumentu Plana održavanja softvera kojeg treba da pročita i razume korisnik softvera.

Plan održavanja softvera treba dokumentovati u dokumentu Plana održavanja softvera kojeg treba da pročita i razume korisnik softvera.

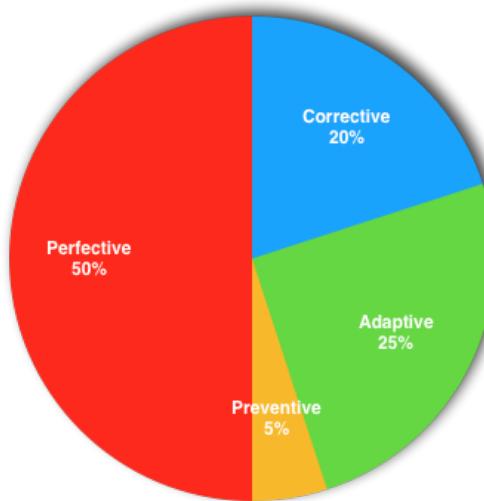
Na primer, ako kapacitet za održavanje nakon isporuke projekta iznosi 2 čovek meseca za narednih 12 meseci, s tim korisnika treba unapred upoznati. Ako vaš call-centar radi samo tokom radnog vremena, korisnika treba obavestiti o alternativnim mehanizmima koje treba uspostaviti kako bi se zahtevi mogli podneti u bilo kom trenutku.

Važna tema kada se govori o održavanju softvera odnosi se na uloženi rad. Koliko, u smislu uloženog rada (koji se izražava u čovek / meseci), treba planirati za održavanje softvera koji je upravo isporučen? Da li najveći deo rada treba rezervisati za ispravku grešaka (korektivno održavanje) ili adaptaciju softvera u okruženju koje se brzo menja (adaptivno održavanje)?

Uobičajena zabluda je da veći deo uloženog rada tokom održavanja treba biti rezervisan za ispravljanje grešaka. Naravno, svaki softver će imati greške i na kraju će mu trebati ispravke. Međutim, ako ste pratili temeljni proces razvoja, velika većina grešaka trebalo je da bude ispravljena pre konačne isporuke.

U praksi je rađeno nekoliko studija koje su pokazale da korektivno

održavanje iznosi približno 20%, tako da se preostalih 80% može koristiti za preostale tri aktivnosti održavanja (slika 2):



Slika 5.2 Distribucija uloženog rada u održavanje softvera Izvor: <https://medium.com/swlh/types-of-software-maintenance-2b0503848b43>

Gornja distribucija se, naravno može razlikovati u skladu sa tačnim okolnostima, kulturom i poslovnim domenom projekta, ali može poslužiti kao dobra polazna osnova pri planiranju. Ako se radi o većoj kompaniji u kojoj se dizajnira, implementira i isporučuje više softverskih

projekata, prikupljanje statističkih podataka o održavanju projekata biće neprocenjiv alat za uspostavljanje polaznih vrednosti specifičnih za kompaniju.

▼ Poglavlje 6

Grupna vežba

GRUPNA VEŽBA: REDOKUMENTOVANJE - SLUČAJEVI KORIŠĆENJA FAZA 1

Nakon prve verzije aplikacije pojavila se potreba da umesto sadašnje arhitekture, aplikacija bude veb orijentisana zbog velikog broja korisnika.

Nakon prve verzije aplikacije pojavila se potreba da umesto sadašnje arhitekture, aplikacija bude veb orijentisana zbog velikog broja korisnika.

Nakon ove odluke dokumentacija je ponovo urađena od korisničkih zahteva, preko implementacije do korisničkog uputstva koje nije postojalo u prvoj verziji zvaničnog dokumenta.

Korisnici koji će imati pristup aplikaciji su studenti i profesori. Zavisno od toga imaju određene mogućnosti za korišćenje aplikacije. Potrebno je samo da su upisani u bazu aplikacije. Korisnici nove veb aplikacije su profesori i studenti za koje treba obezbediti sledeće mogućnosti:

Profesor treba da ima mogućnosti:

- Unos teksta predavanja
- Unos teksta pitanja
- Unos tekstualne pomoći
- Unos konačnog rešenja
- Odabir težine pitanja

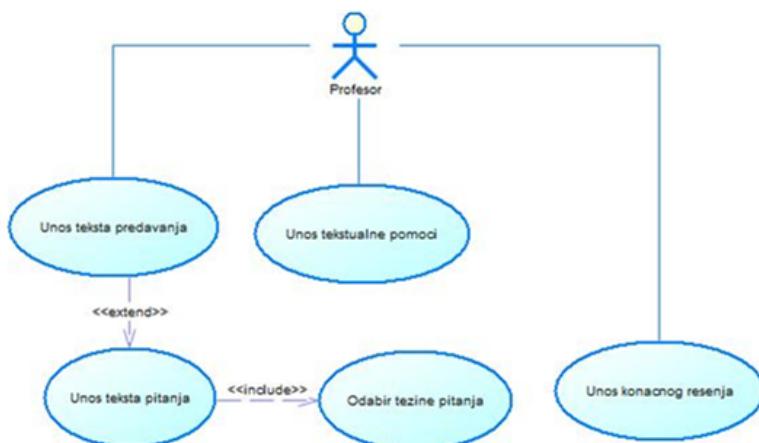
Studentu su ponuđene sledeće opcije:

- Odabir težine vežbe
- Odabir oblasti vežbanja
- Čitanje predavanja
- Rešavanje problema
- Odabir tekstualne pomoći
- Prikazivanje konačnog rešenja i upoređivanje sa rešenjem u aplikaciji
- Ocenjivanje sopstvenog rada

REDOKUMETOVANJE SLUČAJEVA KORIŠĆENJA

Prikazani slučajevi korišćenja su dobijeni redokumentovanjem i opisuju funkcije sistema za potrebe profesora i studenta u novoj veb aplikaciji.

Na slikama 1 i 2 su predstavljeni dijagrami slučajeva korišćenja dobijeni redokumentovanjem koji opisuju funkcije sistema za potrebe profesora i studenta respektivno u novoj veb aplikaciji.



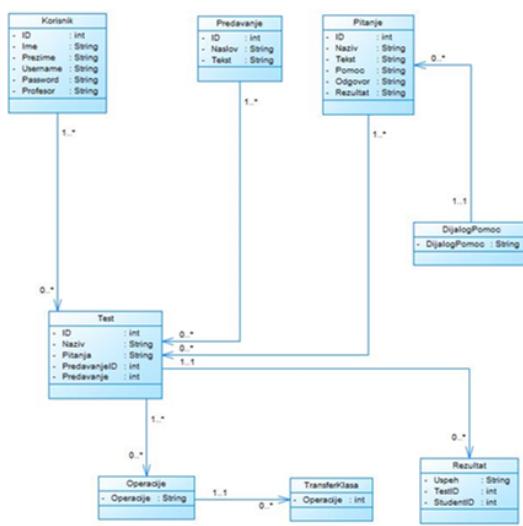
Slika 6.1 Use Case dijagram slučajeva korišćenja za profesora [Izvor: NM SE321-2020/2021.]

Slika 6.2 Use Case dijagram slučajeva korišćenja za studenta [Izvor: NM SE321-2020/2021.]

REDOKUMENTOVANJE KLASNIH DIJAGRAM I DIJAGRAMA ARHITEKTURE

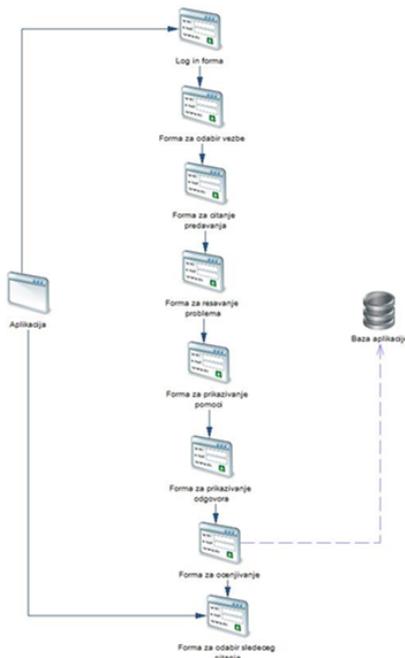
Pored dijagrama slučajeva korišćenja potrebno je izvršiti redokumentovanje i klasnog i dijagrama arhitekture (slike 3 i 4)

- Korisnik (Atributi: ID, Ime, Prezime, Username, Password, Profesor)
- Test (Atributi: ID, Naziv, Pitanja, PredavanjeID, Predavanje)
- Predavanje (Atributi: ID, Naslov, Tekst)



Slika 6.3 Klasni diagram aplikacije [Izvor: NM SE321-2020/2021.]

- **Pitanje** (Atributi: ID, Naziv, Tekst, Pomoć, Odgovor, Rezultat)
- **DijalogPomoc** (Atributi: ID, DijalogPomoc)
- **Rezultat** (Atributi: Uspeh, TestID, StudentID)
- **TransferKlasa** (Atributi: ID, Operacije)
- **Operacije** (Atributi: ID, Operacije)



Slika 6.4 Diagram arhitekture aplikacije sa gledišta studenta [Izvor: NM SE321-2020/2021.]

REDOKUMENTOVANJE - IMPLEMENTACIJA FAZA 1

Pristup aplikaciji je omogućen korišćenjem log in strane gde aplikacija prilikom pokretanja komunicira sa bazom podataka i tabelom za korisnike.

Pristup aplikaciji je omogućen korišćenjem log in strane gde aplikacija prilikom pokretanja komunicira sa bazom podataka i tabelom za korisnike. Proverava se id korisnika, da li je student ili profesor i na osnovu toga aplikacija otvara sledeću formu sa funkcijama predviđenim za tog korisnika.

```
public Form1()
{
    InitializeComponent();
}

[reference]
public int IdKorisnika { get; set; }

[reference]
private void buttonOk_Click(object sender, EventArgs e)
{
    try
    {
        var korisnik = korisniciTableAdapter.GetDataByUserPass(textBoxKorime.Text, textBoxLozinka.Text);
        if (korisnik.Count == 1)
        {
            IdKorisnika = korisnik[0].IdKorisnika;
            if (korisnik[0].IdUloge == 1) // predavac
            {
                Visible = false;
                FormPredavac frm = new FormPredavac(korisnik[0].IdKorisnika);
                frm.ShowDialog();
            }
            else // polaznik
            {
                Visible = false;
                FormPolaznik frm = new FormPolaznik(korisnik[0].IdKorisnika);
                frm.ShowDialog();
            }
            Close();
        }
    }
}
```

Slika 6.5 Pokretanje aplikacije i provera korisničkih kredencijala [Izvor: NM SE321-2020/2021.]

Aplikacija komunicira sa bazom podataka i izvlači vežbe i predavanja. Radi na principu provere popunjениh polja u bazi i ukoliko se nalazi test ili predavanje on ga automatski vrati studentu i prikaže mu u padajućoj listi. Na slici je prikazana klasa koja uzima podatke i prikazuje je u formi za odabir vežbanja studentima. Klasa FormaZaProfesora se bavi ubacivanjem predavanja u bazu. Prvo se proverava da li taj korisnik postoji u bazi i da li ima privilegije profesora. Ukoliko ima dozvoljeno mu je da upisuje u tabelu predavanja. Proverava se da li ima već isti naziv ili tekst, ukoliko ne postoji profesoru je dozvoljeno da unosi naslov i tekst predavanja.

Slika 6.6 Unos novog predavanja [Izvor: NM SE321-2020/2021.]

REDOKUMENTOVANJE - IMPLEMENTACIJA FAZA 2

Ukoliko vežba ili zadatak postoje u bazi aplikacija će onemogućiti njegovo unošenje

Profesor može da unosi naziv pitanja, pomoć i konačan odgovor i klikom na dugme ga čuva u bazi.

Kada je student pristupio vežbama, aplikacija mu prvo izlistava, putem klase PitanjeKontrola,

naziv i tekst pitanja. U tom ocenjivanje mu nije aktivno. Tek nakon završenog zadatka ova polje postaju aktivno i on svoju ocenu može zabeležiti u bazu.

```
private void ucitajPitanjaToolStripMenuItem_Click(object sender, EventArgs e)
{
    for (int i = tcMain.TabPages.Count - 1; i > 0; i--)
        tcMain.TabPages.RemoveAt(i);

    DataRowView drvPredavanje = (DataRowView)cmbPredavanja.SelectedItem;
    OsTester.PredavanjaRow predavanje = (OsTester.PredavanjaRow)drvPredavanje.Row;
    int count = 1;
    foreach (var pitanje in predavanje.GetPitanjaRows())
    {
        TabPage tp = new TabPage();
        tp.Text = "Pitanje " + count;
        count++;
        UserControlPitOdg pitanjeOdgovor = new UserControlPitOdg(pitanje);
        pitanjeOdgovor.Dock = DockStyle.Fill;
        tp.Controls.Add(pitanjeOdgovor);
        tcMain.TabPages.Add(tp);
    }
    sacuvajTestToolStripMenuItem.Enabled = true;
}
```

Slika 6.7 Izlistavanje pitanja [Izvor: NM SE321-2020/2021.]

Kada student odradi test i bude siguran da želi da ga preda bira opciju sačuvaj test. U tom slučaju aplikacija upisuje u bazu podataka rezultat odgovora na pitanja i pruža studentu mogućnost da sam sebe oceni ocenama od 5 do 10. Na ovaj način student na osnovu svoje procene unese ocenu koju misli da je zaslužio i nakon toga profesor pregleda test i u početku ima uvid u realno znanje studenta.

Napomena: Vreme izrade vežbe 45 minuta

✓ Poglavlje 7

Individualna vežba

TEHNIKE ODRŽAVANJA SOFTVERA

Na primeru proizvoljnog koda treba primeniti tehnike održavanja softvera

U okviru vežbe treba uraditi sledeće zadatke:

1. Svaki student treba da na internetu nađe neki proizvoljni kod i da na osnovu njega
 - izvrši reverzni (obrnuti) inženjering (na osnovu koda definiše specifikacije sistema). **vreme izrade 30 minuta**
 - pokuša, ako je to moguće da izvrši reinženjering (ažurira ga bez uticaja na njegovu funkcionalnost) **vreme izrade 30 minuta**
 - izvrši restrukturiranje koda **vreme izrade 30 minuta**

✓ Poglavlje 8

Domaći zadatak

DVANAESTI DOMAĆI ZADATAK

Nakon dvanaeste lekcije potrebno je uraditi dvanaesti domaći zadatak.

Za odabranu proizvoljnu aplikaciju koju ste razvili na nekom od predmeta uraditi jedan od sledećih zadataka po izboru:

1. izvršiti reverzni (obrnuti) inženjerинг (na osnovu koda definisati specifikacije sistema).
2. izvršiti reinženjerинг (ažurirati kod ga bez uticaja na njegovu funkcionalnost)
3. izvršiti restrukturiranje koda

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnog okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ12-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)

jovana.jovic@metropolitan.ac.rs (za studente u Nišu)

sa naslovom (subject mail-a) SE321-DZ12.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Zaključak 12

ŠTA SMO NAUČILI U OVOJ LEKCIJI?

Nedostaci softvera su skupi. Štaviše, troškovi pronalaženja i ispravljanja grešaka predstavljaju jednu od najskupljih aktivnosti razvoja softvera.

Svaki izbor tehnike održavanja softvera treba da sačuva ili poveća kvalitet softvera, uz istovremeno zadržavanje troškova na što je moguće nižem nivou.

Nedostaci softvera su skupi. Štaviše, troškovi pronalaženja i ispravljanja grešaka predstavljaju jednu od najskupljih aktivnosti razvoja softvera. Iako defekti mogu biti neizbežni, možemo minimizirati njihov broj i uticati na naše projekte sprovođenjem procesa upravljanja greškama, koji se fokusira na sprečavanju kvarova, pronalaženju grešaka u što ranijem procesu, i smanjenju njihovog uticaja.

U lekciji se govorilo o mogućnostima primene različitih tehnika za održavanje softvera kao i aktivnostima održavanja koje nam mogu pomoći prilikom plana održavanja. Ne treba dozvoliti da se planiranje održavanja softvera dogodi sviše kasno tokom izrade softverskog projekta, već to treba obaviti već u fazi projektovanja (dizajna sistema).

LITERATURA ZA LEKCIJU 12

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
3. Anne Mette Jonassen Hass, Guide to Advanced Software Testing, © 2008 ARTECH HOUSE, INC.
4. V. B. Spasojević, i ostali, Organizacija sistema kvaliteta i alati kvalitet, Industrija 2004, vol. 32, br. 4, str. 91-107

Dopunska literatura:

1. Samir Lemeš, Nevzudin Buzađija, STANDARDNI MODELI ODRŽAVANJA SOFTVERA, 5. Konferencija „ODRŽAVANJE - MAINTENANCE 2018“ Zenica, B&H, 10. – 12. Maj 2018.

2. Lj. Lazić , Software Quality & Testing Metrics, WSEAS 7th WSEAS EUROPEAN COMPUTING CONFERENCE (ECC '13), Dubrovnik, Croatia, June 25-27, 2013 . (link: <http://www.wseas.org/wseas/cms.action?id=4102>

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.pisa.rs>
4. https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm
5. <https://medium.com/swlh/types-of-software-maintenance-2b0503848b43>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Procena vrste i obima i troškova
(modeli estimacije) poslova u
procesu održavanja SW

Lekcija 13

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 13

PROCENA VRSTE I OBIMA I TROŠKOVA (MODELI ESTIMACIJE) POSLOVA U PROCESU ODRŽAVANJA SW

- ▼ Procena vrste i obima i troškova (modeli estimacije) poslova u procesu održavanja SW
- ▼ Poglavlje 1: Udeo održavanja u ukupnim troškovima SDLC-a
- ▼ Poglavlje 2: Glavni troškovi održavanja softvera
- ▼ Poglavlje 3: Razlozi za održavanje softvera
- ▼ Poglavlje 4: Troškovi održavanja na bazi funkcionalnih tačaka
- ▼ Poglavlje 5: Troškovi održavanja na osnovu nivoa granularnosti
- ▼ Poglavlje 6: Grupna vežba
- ▼ Poglavlje 7: Domaći zadatak
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Merenjem softvera se dobijaju informacije o veličini sistema koji je potrebno izgraditi, što je izuzetno važna informacija u procenama vremena potrebnog za izradu, cene i planiranja razvoja

Merenje veličine softvera je izuzetno značajna aktivnost u procesu razvoja softvera. **Merenjem softvera se dobijaju informacije o veličini sistema koji je potrebno izgraditi, što je izuzetno važna informacija u procenama vremena potrebnog za izradu, cene i planiranja razvoja i održavanja softvera.** U ovoj lekciji pokušaćemo da izvršimo estimaciju veličine, vremena, troškova i napora procesa održavanja softvera.

Kao mera veličine softverskog proizvoda, najčešće se koristi metod procene grubih funkcionalnih tačaka a na osnovu kojih se može proceniti veličina softvera izražena brojem linija izvornog koda, i na osnovu toga, proceniti obim održavanja datog softvera. **Mnogi softverski stručnjaci tvrde da funkcionalnost proizvoda daje bolju sliku o veličini proizvoda, nego njegova dužina.** U ranoj fazi projektovanja, interesantnija je funkcionalnost kada su u pitanju napor i vreme od same fizičke veličine.

Popravke kvara su verovatno najvažnije aktivnosti održavanja. Popravke kvara imaju za cilj održavanje softvera u operativnom stanju posle pada ili izveštaja greške. Troškovi popravke kvara se obično apsorbuju od strane softverske grupe koja pravi aplikaciju, ili je pokrivena eksplicitnim ugovorom garancije.

▼ Poglavlje 1

Udeo održavanja u ukupnim troškovima SDLC-a

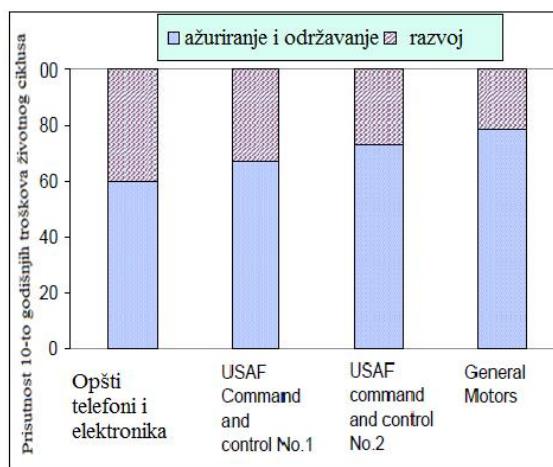
RAZVOJ SOFTVERA I TROŠKOVI ODRŽAVANJA U VELIKIM ORGANIZACIJAMA

Uprkos značajnom delu ukupnih troškova životnog ciklusa softvera povezanim sa aktivnostima održavanja, relativno malo se zna o faktorima koji utiču na njegovo održavanje

Merenje veličine softvera je izuzetno značajna aktivnost u procesu razvoja softvera. Merenjem softvera se dobijaju informacije o veličini sistema koji je potrebno izgraditi, što je izuzetno važna informacija u procenama vremena potrebnog za izradu, cene i planiranja razvoja softvera.

Jones navodi da je "2001 godine, više od 50% globalne softverske populacije bilo angažovano na izmeni postojećih aplikacija, pre nego na pisanju novih aplikacija."

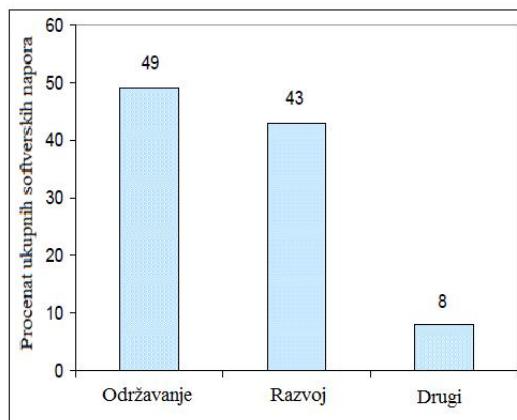
Postoje i neke druge interesantne činjenice koje je sastavio Boehm.



Slika 1.1 Razvoj softvera i troškovi održavanja u velikim organizacijama [Izvor: NM SE321 - 2020/2021.]

Slika 1 pokazuje procenat troškova održavanja softvera 10-ogodišnjeg životnog ciklusa u velikim organizacijama, u rasponu od 60% za telefoniju i elektroniku do 75% za General Motors.

Slika 2, zasnovana je na podacima iz 487 instalacija koje se bave obradom poslovnih podataka, pokazuje da je odnos prosečnog razvoja i održavanje između ispitanih instalacija bio 47 do 53.



Slika 1.2 Razvoj softvera i troškova održavanja u 487 poslovnim organizacijama [Izvor: NM SE321 - 2020/2021.]

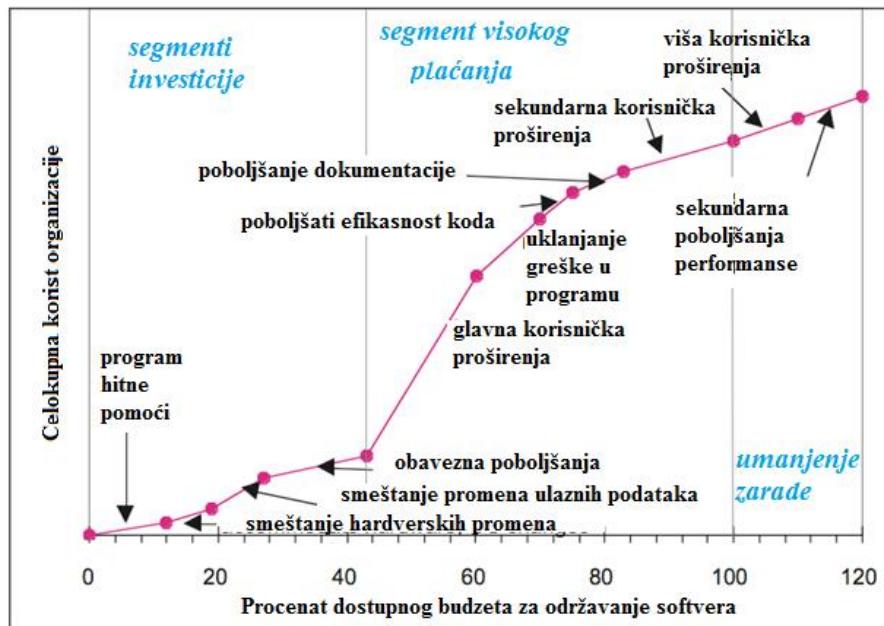
Uprkos značajnom delu ukupnih troškova životnog ciklusa softvera povezanim sa aktivnostima održavanja, relativno malo se zna o procesu održavanja softvera i faktorima koji utiču na njegovu cenu.

FUNKCIJA PRODUKCIJE ODRŽAVANJA SOFTVERA

Najznačajnija karakteristika je veoma visok napor održavanja (40% do 50%), koji koristi segment investicije.

Tipična funkcija isplative proizvodnje softverskog održavanja je prikazana na slici 3.

Segment investicije: čine one aktivnosti održavanja koje se moraju izvršiti ako program ne predstavlja pogoršanje vrednosti: popravke programa hitne pomoći, smeštaj promena u hardveru, operativnim sistemima, bazi osnovnih podataka, ulaznim podacima, itd), i mandatna proširenja (na primer, novi porez na dohodak izveštavanja itd.).



Slika 1.3 Funkcija produkcije održavanja softver [Izvor: NM SE321 - 2020/2021.]

Segment visoko-isplative krive se sastoji od: osnovnih prioriteta poboljšanja za korisnike, primarnih poboljšanja u programu efikasnosti, pouzdanosti i dokumentacije, kao i skupa sekundarnih poboljšanja korisnika koje pružaju manji, ali još uvek pozitivan višak prednosti nad troškovima.

Segment krive smanjenja zarade se sastoji od "lepo je imati" karakteristika (kao što su, izveštaji i prikazi ograničenih zahteva, prepisivanje slabo strukturiranih, ali stabilnih modula itd.) Sve ove karakteristike daju neke koristi, ali ne toliko u odnosu na njihove troškove aktivnosti u visoko-isplativom segmentu.

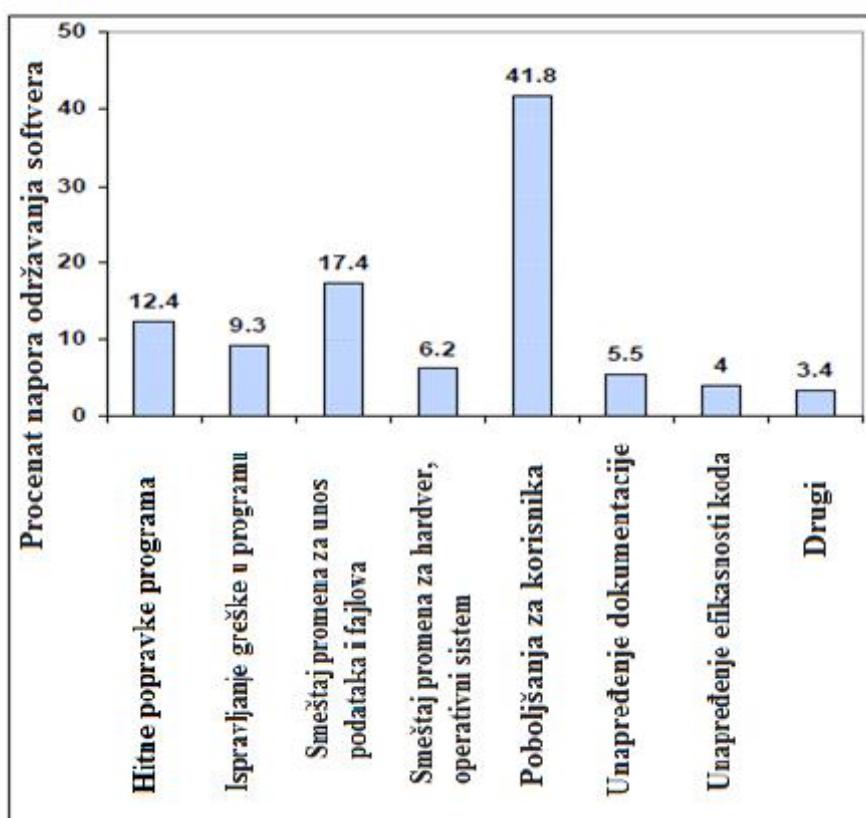
Najznačajnija karakteristika na slici 3 je veoma **visok napor održavanja** (40% do 50%), koji koristi segment investicije. On predstavlja rad koji se mora izvršiti da bi se samo održale vrednosti sistema na približno sadašnjem nivou. Jasno, to je oblast u kojoj se koriste savremene prakse programiranja da bi se smanjila potreba za fiksiranjem programa hitne pomoći a minimizirao efekat ekoloških promena, koji može imati snažan uticaj na ukupne napore za održavanje.

DISTRIBUCIJA NAPORA ZA ODRŽAVANJE SOFTVERA PO DELATNOSTIMA

Korektivno održavanje (popravke programa hitne pomoći i rutinsko otklanjanje grešaka), generalno najveći deo za održavanje hardvera budžeta, ima tendenciju da troši relativno malo.

Na osnovu ankete od 487 aplikacija za obradu poslovnih podataka, slika 4 pokazuje koliko se napor za održavanje softvera obično distribuirala među glavnim kategorijama update-ovanja i popravke.

Slika pokazuje prema ovim izvorima, da korektivno održavanje (popravke programa hitne pomoći i rutinsko otklanjanje grešaka), generalno najveći deo za održavanje hardvera budžeta, ima tendenciju da troši samo relativno mali (21,7%) deo napora za održavanje softvera kao što je definisano ovde. Dakle, postizanje razvoja softvera bez greške ne eliminiše potrebu za značajnim budžetom za održavanja softvera.

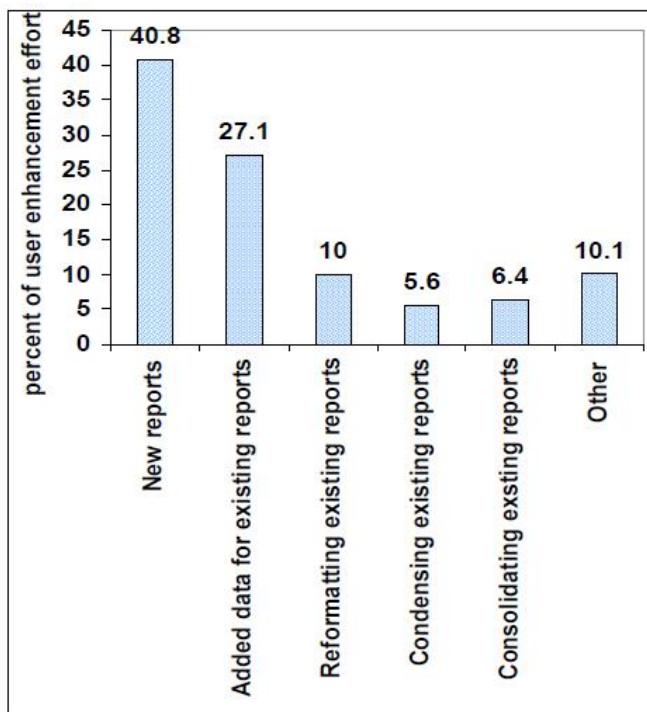


Slika 1.4 Distribucija napora za održavanje softvera [Izvor: NM SE321 - 2020/2021.]

DISTRIBUCIJA NAPORA ZA ODRŽAVANJE SOFTVERA - POBOLJŠANJE SOFTVERA

Najveći deo (41,8%) napora održavanja softvera je posvećen poboljšanju softvera za korisnike.

Najveći deo (41,8%) napora održavanja softvera je posvećen poboljšanju softvera za korisnike. Slika 5 prikazuje kako se napor unapređenja korisnika obično distribuira, u smislu izveštaja koje proizvodi softverski sistem. Postaje jasno iz ove distribucije da (bar za obradu poslovnih podataka), strukture fleksibilnih podataka i mogućnosti generisanja izveštaja igraju važnu ulogu u poboljšanju efikasnosti održavanja softvera.



Slika 1.5 Distribucija napora proširenja korisnika [Izvor: NM SE321 - 2020/2021.]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

Glavni troškovi održavanja softvera

PODRAZUMEVANE VREDNOSTI ZA OBIME ODRŽAVANJA SOFTVERA

Autor Jones je publikovao nominalne podrazumevane vrednosti za procenu raznih vrsta aktivnosti održavanja

Autor Jones je publikovao nominalne podrazumevane vrednosti za procenu raznih vrsta aktivnosti održavanja, kao što je prikazano u tabeli na Slici 1. Metrike korištene da bi se iskazale ove podrazumevane vrednosti su "obimi zadatka" (A obimi), i "stope proizvodnje" (P stope). Metrika stopa proizvodnje se odnosi na količinu softvera na kojem može da se primeni održavanje po intervalu jedinice vremena. Termin obim zadatka se odnosi na iznos softvera koji jedan programer za održavanje softvera može da održi operativnim u normalnom toku godine, pod pretpostavkom rutinskih popravki kvarova i manjih ispravki. Iznos softvera se meri u funkcionalnim tačkama (FP).

Kao što se može posmatrati iz tabele na Slici 1, obimi dodele mogu da variraju od 300 do 5000 funkcionalnih tačaka, sa prosečnom vrednosti od oko 2000 funkcionalnih tačaka. Druga studija otkriva da je prosečna stopa uklanjanja grešaka oko 8 ispravljenih grešaka po zaposlenom mesečno.

Kao što je sam Jones ispravno istakao, nijedna od ovih vrednosti nije dovoljno rigorozna za formalnu procenu troškova, ali su dovoljne za ilustraciju nekog od tipičnih trendova u različitim vrstama održavanja. Očigledno je da je potrebno i podešavanje iskustva tima, složenost aplikacije, programski jezici, i mnogi drugi lokalni faktori.

U nastavku, diskutujemo o detaljima nekih od važnih aktivnosti održavanja.

Aktivnosti	<i>A obimi, FP</i>	<i>P stope, FP mesečno</i>	<i>P stope, radno vreme po FP</i>	<i>P stope, LOC na mesečnom nivou po osobi</i>
Korisnička podrška	5.000	3.000	0,04	300,000
Kod restrukturiranja	5.000	1.000	0,13	100,000
Analiza Složenosti	5.000	500	0,26	50,000
Obrnuti inženjering	2,500	125	1,06	12,500
Penzionisanje	5,000	100	1,32	10,000
Oblasť usluge	10,000	100	1,32	10,000
Dead Code uklanjanje	750	35	3,77	3,500
Proširenja (manja)	75	25	5,28	2,500
Reinženjering	500	25	5,28	2,500
Održavanje (popravke kvara)	750	25	5,28	2,500
Garancija popravke	750	20	6,60	2,000
Migracija na novu platformu	300	18	7,33	1,800
Proširenja (glavna)	125	15	8,80	1,500
Nacionalizacija	250	15	8,80	1,500
Konverzija u novi interfejs	300	15	8,80	1,500
Obavezna promena	750	15	8,80	1,500
Optimizacija performansi	750	15	8,80	1,500
Godina-2000 popravke	2,000	15	8,80	1,500
Eurocurrenci konverzije	1,500	15	8,80	1,500
Greška-skloni modul za uklanjanje	300	12	11,00	1,200
Prosek	2,080	240	5,25	24,000

Slika 2.1 Nominalne podrazumevane vrednosti za procenu raznih vrsta aktivnosti održavanja autora Capers Jones [Izvor: NM SE321 - 2020/2021.]

POPRAVKE SOFTVERSKOG KVARA

Popravke kvara imaju za cilj održavanje softvera u operativnom stanju posle pada ili izveštaja greške.

Popravke kvara su verovatno najvažnije aktivnosti održavanja. **Popravke kvara imaju za cilj održavanje softvera u operativnom stanju posle pada ili izveštaja greške**. Troškovi popravke kvara se obično apsorbuju od strane softverske grupe koja pravi aplikaciju, ili je pokrivena eksplisitnim ugovorom garancije.

Popravke kvara su neizbežne. Prosek efikasnosti uklanjanja kvara u realizaciji softvera u Americi iznosi oko 85%. U prvih nekoliko godina nakon puštanja softverskog paketa, prijavljene popravke kvarova od strane korisnika iznose značajan trošak kako za interne tako i za komercijalne softverske pakete.

Popravke kvara se retko mere korišćenjem normalno izračunatih funkcionalnih tačaka, zato što većina popravki kvara odgovara otprilike jednoj četvrtini funkcionalne tačke, a za razlomljene funkcionalne tačke ne postoje pravila brojanja.

Zajednička metrika za merenje produktivnosti održavanja je mesečni broj ispravljenih kvarova. U uslovima ove metrike norme u US-a su oko 8 ispravljenih kvarova mesečno. Ali postoje značajne varijacije u broju mesečnih ispravki kvara. Neke organizacije za održavanje sastavljene od iskusnog osoblja za održavanje i sa velikom podrškom alata za promenu kontrole i za praćenje kvara, mogu da postignu prosečnu stopu od 16 do 20 ispravljenih grešaka mesečno.

Druga metrika može biti metrika po ceni kvara. Problem sa ovim metrikama se javlja u situacijama u kojima osoblje za održavanje troši vreme i budžet održavanja projekta, a ne prijavljuje greške. Na taj način se mogu oštetiti visoko kvalitetne aplikacije (sa manje

prijavljenih grešaka mesečno) i predstaviti da nisko kvalitetna aplikacija (sa mnogo grešaka prijavljenih mesečno), izgledaju bolje nego što zaista jesu.

VRSTE SOFTVERSKIH KVAROVA

To su: problematični (Abeyant) kvarovi, nevažeći kvarovi, loše ispravljen unos i duplirani kvarovi

U Tabeli na Slici 2 prikazane su nominalni odzivi za popravke kvara, koristeći skalu sa četiri tačke ozbiljnosti koji je razvio IBM 1960-te. Faktori koji utiču na estimaciju popravke kvara su:

- **Problematični (Abeyant) kvarovi:** Za oko 10 odsto dolazećih izveštaja klijenata o greškama, tim za održavanje softvera ne može da spreči da se javi isti problem. Ovi problematični kvarovi su zasnovani na nekim jedinstvenim kombinacijama događaja. Troškovi pronalaženja i popravki problematičnih grešaka sastoje se od grešaka bilo kojeg datog nivoa težine, i vreme koje je potrebno može se značajno produžiti.
- **Nevažeći kvarovi:** Za oko 15% dolazećih izveštaja o kvaru od strane klijenata, greška zapravo nije izazvana zbog softverskih aplikacija zbog kojih je prijavljena. Ponekad klijent jednostavno pravi grešku, ponekad je greška zaista hardverski problem, a ponekad i greška uzrokovana nekim drugim softverskim aplikacijama i pogrešnim dijagnozama. Za komercijalne softverske pakete sa mnogo korisnika, najmanje 15% vremena provedenog od strane pomoćnog osoblja podržanog od strane kupca i 10 % vremena provedenog od programera održavanja je završen sa nevažećim izveštajima greške.
- **Loše ispravljen unos:** Svaki put kada je softverski kvar popravljen, postoji verovatnoća da sama popravka može da sadrži novi kvar.

Nivo ozbiljnosti	Značenje	Vreme utrošeno od izveštaja do inicijalne popravke	Procenat Prijava
Ozbiljnost 1	Aplikacija se ne pokreće	12 sati	1
Ozbiljnost 2	Velika funkcija onemogućena	48 sati	12
Ozbiljnost 3	Manja funkcije onemogućene	30 dana	52
Ozbiljnost 4	Kozmetička greška	120 dana	35

Slika 2.2 Nominalno vreme odaziva za popravke greški od strane nivoa ozbiljnosti [Izvor: NM SE321 - 2020/2021.]

Opšti naziv ovih izvedenih grešaka je loša ispravka, i one su iznenađujuće uobičajene. Posmatrani niz ubacivanja loših ispravki iznosi od manje od 1% popravljenih kvarova do više od 20%, u US-u je prosek oko 7% popravljenih kvarova koji aktiviraju novi kvar. Loše ispravke se manje verovatno javljaju kada je aplikacija dobro strukturirana, i kada su aplikacije pažljivo napravljene, bez nepotrebne žurbe od strane iskusnog osoblja.

- **Duplirani kvarovi:** Za svaki softverski paket sa više korisnika, verovatno da će više od jednog korisnika pronaći i prijaviti isti problem. Sa stanovišta estimacije, izveštaji dupliranih kvarova trebaju samo jednom biti ispravljeni. Međutim, značajan iznos podrške vremena klijentima je posvećen bavljenjem ovim duplikatima. Za kompanije kao što su

Corel, Computer Associates, IBM ili Microsoft, oko 10% vremena podrške klijentima će ići u radu sa mnogim žalbama o istom kvaru.

MODULI SKLONI GREŠKAMA

Moduli skloni grešci ne mogu se stabilizovati, zato što stopa unošenja loših ispravki može da dostigne 100%, što znači da svaki pokušaj da se popravi greška može da uvede novu grešku.

Istraživanje IBM-a 1960 god. je otkrilo da se softverske greške u velikim sistemima retko distribuiraju. Umesto toga, one imaju tendenciju da se grupišu u manji broj veoma kvarljivih particija, pod nazivom moduli skloni greškama. Kao primer razmotrimo sledeće: u 1970 IBM veliki proizvod baze podataka, sistem upravljanja informacijama (IMS), koji sadrži 425 modula. Oko 300 tih modula nikada nije dobio grešku prijavljenu od strane klijenta. Ali, 31 modul od 425 je primio više od 2000 izveštaja kupaca o grešci svake godine, ili oko 60% od ukupnog broja izveštaja o greškama celog proizvoda.

Moduli skloni grešci ne mogu se stabilizovati, zato što stopa unošenja loših ispravki može da dostigne 100%, što znači da svaki pokušaj da se popravi greška može da uvede novu grešku.

Moduli skloni greškama su uobičajeni među velikim loše strukturiranim sistemima i aplikacijama, i veoma su skupi za održavanje tako da bi u potpunosti trebali biti eliminisani. Kumulativni troškovi modula sklonih greškama mogu biti veći od troškova za normalne module za skoro 500%.

Najčešćih razlozi za postojanje ovakvih modula koji su skloni greškama su:

- Prekomerni raspored pritiska
- Nedovoljno obučeni programeri koji nemaju znanje o strukturiranim tehnikama
- Nedovoljno praćenje kvarova u cilju shvatanja da su moduli skloni greškama prisutni

▼ Poglavlje 3

Razlozi za održavanje softera

PODRŠKA KORISNICIMA I RESTRUKTUIRANJE PROGRAMSKOG KÔDA

Uloga podrške korisniku je jedna od veza između klijenata i timova za popravku kvarova. Restrukturiranje koda ima za cilj smanjenje nivoa složenosti softvera

Razlozi za održavanje softvera mogu biti mnogobrojni. Ovde su navedeni neki za koje se može smatrati da su najčešći.

Podrška korisnicima: Uloga podrške korisniku je jedna od veza između klijenata i timova za popravku kvarova. Aktivnost se uglavnom koristi za komercijalne softvere sa više korisnika. Veliki deo rada podrške kupcima se prekida i pokreću ga sami kupci kada imaju probleme, ili kada im trebaju odgovori na neka pitanja. Osoblje podrške korisnicima obično ne ispravlja probleme. Javlja se novi problem, uloga podrške korisnicima je da zabeleži simptome i da uputi problem na odgovarajuću grupu za popravke.

Procene korisničke podrške su zasnovane na očekivanom broju grešaka u aplikacijama, ali značajnija promenljiva estimacija je broj korisnika ili klijenata. Kao grubo pravilo, pod pretpostavkom da je kontakt telefon primarni kanal, 1 osoba iz korisničke podrške može mesečno da primi pozive od oko 150 klijenata, dok za e-mail-ove i elektronske kontakte odnos može da bude 1 osoba korisničke podrške za 1000 klijenata, pod pretpostavkom da je softver prosečnog kvaliteta i nije bilo većih oštećenja u vreme puštanja.

Restrukturiranje koda : Obično se izvršava pomoću automatskih alata, i ima efekat smanjenja nivoa složenosti softvera. Smanjenje kompleksnosti, zauzvrat, olakšava održavanje jer je brže i lakše pronaći stvari u dobro struktuiranom softveru, nego u veoma složenom softveru. Dakle, to nije jedna nezavisna aktivnost održavanja, već korisna prethodnica drugih aktivnosti održavanja. Obično, restrukturiranje ne degradira performanse ili uvećanje grešaka. Međutim, neke restrukturirane aplikacije sadrže oko 10 % više koda nego što su ranije sadržale.

PROJEKTI MIGRACIJE I KONVERZIJA U NOVE ARHITEKTURE

Projekti migracije su oni koji su povezani sa kretanjem aplikacije sa jedne platforme na drugu. Projekti konverzije obično uključuju promene na interfejsu ili strukturi podataka

Projekti migracije: Projekti migracije su oni kojisu povezani sa kretanjem aplikacije sa jedne platforme na drugu. Primeri migracija uključuju potiranje jedne aplikacije iz jednog operativnog sistema u drugi, ili iz jednog procesora na drugi. Finansiranje projekata za migraciju varira od slučaja do slučaja, iako su većina projekata migracije za unutrašnje softvere plaćeni od strane korisnika. Za komercijalni softver, migracija se obično obavlaju u cilju proširenja tržišta, ili dostizanja novog.

Ako su alati dostupni za renoviranje i specifikacije softvera dobro dokumentovani, migracija se može kretati stopama u više od 50 funkcionalnih tačaka po osobi, mesečno. Ako je softver napisan nejasnim jezicima, a ako nedostaju specifikacije ili su zastarele, migracija može da nastavi u stopi manje od 5 funkcionalnih tačaka po zaposlenom mesečno.

Projekti konverzije: Projekti konverzije obično uključuju promene na interfejsu ili strukturi podataka, kao što je prelazak iz organizacije podatakakorišćenjem fajlova u relacione baze podataka, ili prebacivanje od monolitne aplikacije na klijent/server aplikaciju. Pošto konverzija projekata često dodaje funkcije ili objekte koji koriste korisnicima, oni su često motivisani da finansiraju te projekte.

Kvalitet specifikacije igra ključnu ulogu u određivanju produktivnosti, i za konverziju projekata. U slučaju nestalih ili nepotpunih specifikacija, inverzni inženjeri softvera treba da se izvrši da bi se izdvojile neke od informacija dizajna koje nedostaju. Reverzni inženjeri softvera može da se vrši pomoću automatskih alata.

OBAVEZNE PROMENE I OPTIMIZACIJA PERFORMANSI

Obavezne promene su modifikacije softvera napravljene kao odgovor na promene u zakonu ili politici. Optimizacija se obično bavi smanjivanjem kašnjenja u transakcijama

Obavezne promene : Obavezne promene su modifikacije softvera napravljene kao odgovor na promene u zakonu ili politici.

Na primer, promene poreskih stopa ili promene u zdravstvenim ustanovama će zahtevati promene u mnogim softverskim paketima. Izvor sredstava za obavezne promene je dvosmislen i može da varira od slučaja do slučaja. Obavezne promene su jedan od zabrinjavajućih oblika održavanja softvera, jer su troškovi često visoki, raspored pritisaka je intenzivan, a mogu postojati stroge kazne za nepridržavanje. Da bi stvari bile još gore, gotovo je nemoguće predvideti takve obavezne promene u unapred.

Optimizacija performansi projekta : Optimizacija performansi projekta se obično se vrši za aplikacije i sisteme sa veoma visokim stopama transakcije, kao što su obrada kreditnih kartica ili aplikacije u realnom vremenu. Optimizacija se bavi smanjivanjem kašnjenja u transakcijama pronalazeći odeljke koji usporavaju stvari preduzimajući odgovarajuće korake optimizacije.

PROŠIRENJA SOFTVERA

Proširenja su zabrinuta dodavanjem novih mogućnosti u postojeći softver da bi odgovorala eksplizitnim zahtevima korisnika.

Proširenja softvera : Proširenja su zabrinuta dodavanjem novih mogućnosti u postojeći softver da bi odgovorala eksplizitnim zahtevima korisnika. Pošto nove funkcije pronađe nove zahteve korisnika, finansiranje proširenja se obično izvodi od strane korisnika i deli poboljšanja u glavna i sporedna. Manja poboljšanja su ograničena na primenu ispravke koja se rade za nedelje ili manje, što podrazumeva veličinu od 5 funkcionalnih tačaka. Ova veličina je dovoljna za dodavanje novih izveštaja ili novih elemenata ekrana. Dok glavna poboljšanja za ispravke uglavnom prelaze 20 funkcionalnih tačaka. Na ovoj veličini, značajne ispravke kao što su dodavanje potpuno novih karakteristika u sistem često se pronađe veoma teško. Između ispravki koji su definitivne kao manje i one koje su definitivne kao velike je jedna dvostrinska zona koja se može definisati na bilo koji način na osnovu specifičnih okolnosti. Godišnja stopa poboljšanja postojećih aplikacija u proseku neto porasta u ukupnim funkcionalnim tačkama aplikacija je od oko 7 % godišnje, mada postoje velike razlike.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 4

Troškovi održavanja na bazi funkcionalnih tačaka

TROŠKOVI ODRŽAVANJA – MERENJE VELIČINE SOFTVERA

Funkcionalne tačke mere količinu funkcionalnosti sistema opisanog specifikacijom.

Kao mera veličine softverskog proizvoda najčešće se koristi metod procene grubih funkcionalnih tačaka, a na osnovu kojih se može proceniti veličina softvera izražena brojem linija izvornog koda, i na osnovu toga proceniti obim održavanja datog softvera. Procenu grubih funkcionalnih tačaka izvršićemo na osnovu dijagrama toka podataka date aplikacije. Mnogi softverski stručnjaci tvrde da funkcionalnost proizvoda daje bolju sliku o veličini proizvoda, nego njegova dužina. U ranoj fazi projektovanja, interesantnija je funkcionalnost kada su u pitanju napor i vreme, od same fizičke veličine.

Funkcionalne tačke (FP) mere količinu funkcionalnosti sistema opisanog specifikacijom. Da bi se izračunale funkcionalne tačke potrebno je identifikovati računaljive diskretnе komponente sledećeg tipa:

- **Eksterni ulazi** -procesni podaci ili upravljačke informacije koje dolaze izvan
- **Eksterni izlazi** - procesni podaci i upravljačke informacije koje idu van
- **Eksterni upiti** - interaktivni upiti koji zahtevaju izlaz
- **Eksterne interfejs datoteke** - datoteke koje se koriste od drugih sistema
- **Interne matične datoteke** - logičke interne datoteke

Albrecht je funkcionalne tačke (poeni) - FP računao po formuli :

broj ulaza * 4 + broj izlaza * 5 + broj upita * 4 + broj matičnih datoteka * 10 + broj interfejsa * 7

sa tim, da težinski faktori mogu varirati +/- 25% zavisno od kompleksnosti programa. Težinski faktori su prikazani u tabeli na slici 1.

BAZNI ELEMENT	TEŽINSKI FAKTOR		
	Jednostavan	Srednji	Kompleksni
ULAZI	3	4	6
IZLAZI	4	5	7
UPITI	3	4	6
EKSTERNI FAJLOVI	7	10	15
INTERNE DATOTEKE	5	7	10

Slika 4.1 Faktori za podešavanje FP-a [Izvor: NM SE321 - 2020/2021.]

GLAVNA NAMENA FP MODELA

Glavna namena FP je merenje veličine proizvoda koja se koristi u predviđanju napora i troškova u ranoj fazi životnog ciklusa softvera (specifikacija, projektovanje).

Pomoću gornje formule se dobiju nepodešene (grube) funkcionalne tačke UFC. Podešene funkcionalne tačke se dobiju množenjem UFC sa faktorom tehničke kompleksnosti TCF koji ima 14 doprinosećih faktora (on-line ulazi, on-line ažuriranje, kompleksnost interfejsa, ponovna upotreba, komuniciranje sa podacima itd.) i varira od 0.65 do 1.35.

Glavna namena FP je:

- *merenje veličine proizvoda koja se koristi u predviđanju napora i troškova u ranoj fazi životnog ciklusa softvera (specifikacija, projektovanje)*
- *predviđanje broja linija izvornog koda*
- *konverzija veličine projekta pisanog u jednom jeziku u veličinu projekta pisanog u drugom jeziku jer su FP nezavisne od jezika*
- *merenje produktivnosti projekata koji su pisani u više jezika*
- *normalizacija u odnosu na FP (defekti/FP, čovek meseci/FP itd.)*
- *upotreba FP kao baze za ugovaranje.*

Očekivani broj linija izvornog koda dobijen iz FP (ili na neki drugi način) omogućava predviđanje drugih karakteristika softvera.

Na osnovu resurs modela, koji se sastoji od empirijskih jednačina, obično se predviđa:

E - napor u čovek / mesecima

D - trajanje projekta

DOC - broj linija dokumentacije

Međutim, treba imati u vidu da se empirijski podaci dobijaju na osnovu limitiranog i specifičnog broja projekata i resurs model nije odgovarajući za sve klase softvera. Profesor Basili je resurs modele svrstao u četiri klase:

- **statički jednovarijabilni model**
- **statički multivarijabilni model**
- **dinamički multivarijabilni model**
- **teoretski model**

Statički jednovarijabilni model ima sledeću formu:

$$\text{resurs} = c_1 * (\text{očekivane karakteristike})^{c_2}$$

Konstante c_1 i c_2 se deriviraju iz prethodnih projekata. Kao primer, Pressman navodi studiju Walston-a i Felix-a baziranoj na 60 razvojnih projekata na osnovu klijih su došli do sledećih rezultata, odnosno modela.

EKSPERIMENTALNO UTVRĐENI MODELI

Eksperimentalno utvrđeni modeli se ne mogu uzeti generalno jer zavise od okruženja i specifičnosti aplikacija.

Eksperimentalno utvrđeni modeli su:

- $E = 5.2 * L^{0.91}$
- $D = 4.1 * L^{0.36}$
- $D = 2.47 * E^{0.35}$
- $S = 0.54 * E^{0.06}$
- $\text{DOC} = 49 * L^{1.01}$

gde su E, D (trajanje projekta i kalendarsko trajanje projekta) i DOC su izvedeni na osnovu L tj. broja linija izvornog koda u hiljadama.

Alternativno, S (potrebni ljudi) i trajanje projekta mogu biti izračunati iz deriviranog ili predviđenog napora E.

Gornje jednačine se ne mogu uzeti generalno, jer zavise od okruženja i specifičnosti aplikacija. Ovakav model se derivira na osnovu lokalnog okruženja i istorijskih podataka.

OSTALI TIPIČNI MODELI

Ove modele je veoma teško porebiti zbog različitih definicija L (npr. da li su uračunati komentari ili ne) i E (da li se odnosi samo na kodiranje ili i na ostale faze životnog ciklusa)

Ostali tipični modeli su:

- $E = 5.5 * 0.73 L^{1.16}$ (Bailey-Basili model)
- $E = 3.2 * L^{1.05}$ (Bohem-ov jednostavni model)
- $E = 3.0 * L^{1.12}$ (Bohem-ov srednji model)
- $E = 2.8 * L^{1.20}$ (Bohem-ov kompleksni model)
- $E = 5.288 * L^{1.047}$ (Doty model)

Ove modele je veoma teško porebiti zbog različitih definicija L (npr. da li su uračunati komentari ili ne) i E (da li se odnosi samo na kodiranje ili i na ostale faze životnog ciklusa softvera). Upotreboom *statičkog jednovarijabilnog resurs modela*, Schaefer predviđa broj čovek / meseci potrebnih za održavanje softvera godišnje E.maint na sledeći način :

$$E.\text{maint} = ACT * 2.4 * L^{1.05}$$

gde se ACT definiše sa :

$$ACT = (L - \text{za sistem koji se održava}) / CI$$

pri čemu je CI broj linija koda koje su modifikovane ili dodane za vreme jednogodišnjeg održavanja.

Statički multivarijabilni model ima sledeću formu :

$$\text{resurs} = c_{11} * e_1 + c_{21} * e_2 + \dots$$

gde je e_i i-ta softverska karakteristika a c_i empirijski derivirana konstanta za i-tu karakteristiku softvera.

Dinamički multivarijabilni model projektuje zahteve za resursima kao funkciju vremena. *Teoretski model* prepostavlja specifičnu distribuciju napora u toku razvoja projekta i na osnovu toga prati ponašanje resursa.

IZRAČUNAVANJE BROJA LINIJA KÔDA

Na osnovu izračunatih vrednosti funkcionalnih tačaka se može odrediti numerički nivo jezika, i što je taj nivo viši potrebno je manje linija kôda po funkcionalnoj tački.

Na osnovu izračunatih vrednosti funkcionalnih tačaka se može odrediti numerički nivo jezika i što je taj nivo viši potrebno je manje linija koda po funkcionalnoj tački. Numerički nivo jezika omogućava prečicu za konvertovanje veličine softverskog proizvoda iz jednog jezika u drugi. Odnos nivoa jezika i srednje produktivnosti dat je u sledećoj tabeli (slika 2):

NIVO JEZIKA	SREDNJA PRODUKTIVNOST
PO ČOVEK-MESECU	
1 - 3	5 - 10 FP
4 - 8	10 - 20 FP
9 - 15	16 - 23 FP
16 - 23	15 - 30 FP
24 - 55	30 - 50 FP
> 55	40 - 100 FP

Slika 4.2 Relacija nivoa jezika i srednje produktivnosti čoveka po mesecu [Izvor: NM SE321 - 2020/2021.]

Sledi lista nekih jezika kao i njihovih nivoa i srednjih brojeva izvornih kodova po funkcionalnoj tački (slika 3).

Slika 4.3 Relacije jezici-nivoi-izvorne linije kôda [Izvor: NM SE321 - 2020/2021.]

▼ Poglavlje 5

Troškovi održavanja na osnovu nivoa granularnosti

PHASE-LEVEL MODELI

Fokusirani su na napor rutinskog rada održavanja u toku određenog perioda ili u toku cele faze održavanja softvera.

Obzirom na značaj održavanja softvera, uveden je određen broj modela i primenjen na estimaciju troškova održavanja. Ovi modeli se koriste za održavanja softvera, npr. ispravljanje grešaka, funkcionalnih poboljšanja, tehničkih renoviranja, i reinženjeringu. Na osnovu nivoa granularnosti fokusa estimacije, oni se mogu grubo klasifikovati u tri tipa:

1. **phase-leve modeli za estimaciju održavanja**
2. **release-leve modeli za estimaciju održavanja**
3. **task-level modeli za estimaciju održavanja.**

Phase-level modeli: predstavljaju skup modela za održavanje koji su fokusirani na napore rutinskog rada održavanja u toku određenog perioda ili u toku cele faze održavanja softvera. Rutinski rad održavanja se odnosi na sve aktivnosti tokom rada softverskog sistema nakon što je isporučen. To uključuje ispravke grešaka, manje funkcionalne promene i poboljšanja, kao i tehnička unapređenja, čiji je glavni cilj da obezbedi redovno funkcionisanje sistema. Ovi modeli za održavanje su integrirani u COCOMO, SEER-SEM, PRICE-S, SLIM, i KnowledgePlan. U ovim modelima, troškovi održavanja su obično deo estimacije proizvoda prilikom estimiranja troškova novog sistema koji će biti razvijen. Dakle, ključni ulaz za određivanje napora potrebnog za održavanje je dimenzija sistema.

Većina od ovih modela koriste dodatne troškove za korisnike koji su specifični za održavanje softvera. Na primer, COCOMO koristi dva parametra, naime razumevanje softvera (**software understanding**- SU) i nivo nepoznavanja od strane programera (**level of unfamiliarity of the programmer**- UNFM) za svoje dimenzionisanje održavanja; SEEK-SEM koristi parametre kao što su održavanje veličine rasta tokom vremena i održavanje strogosti (temeljitost održavanja aktivnosti koje treba obaviti), u njegovoj kalkulaciji, troškova održavanja.

Kod estimiranja troškova održavanja sistema koji se razvija, estimacija cene je važna za analizu arhitekture kompromisa i izradu investicionih odluka na sistemu koji se vrednuje. Ipak, zbog više prepostavki o sistemu koji tek treba da se razvija, teško je estimirati troškove održavanja sistema precizno.

Drugi mogući razlog je da su ti modeli, sa izuzetkom COCOMO-a, vlasnički, i detalji nisu u potpunosti objavljeni, što ih čini teškim za istraživanje u istraživačkom kontekstu. Za estimaciju adaptacije i ponovnog korišćenja, COCOMO, SEEK -SEM, PRICE- S, SLIM, i

KnovledgePlan, pružaju metode za izračunavanje veličine rada i napora. Očigledno, ovi modeli prepostavljaju da adaptacija i ponovno korišćenje imaju iste osobine sa novim razvojem softvera. Nažalost, ova prepostavka nikada nije bila empirijski potvrđena.

RELEASE-LEVEL MODELI: MODEL BASILI I SARADNICI; MODEL RAMIL I LEHMAN

Model Basili i saradnici kao promenljivu koristi broj linija koda, dok su Ramil i Lehman uveli ocene linearnih regresija modela za estimaciju napora potrebnog da se razvije sistem.

Umesto faze estimacije troškova održavanja kao celine, druga grupa modela fokusira se na troškove održavanja na nivo finije granulitnosti, estimaciju napora planiranog skupa zadataka održavanja ili planirane isporuke. Ovaj pristup se obično uključuje korišćenjem podataka iz prethodnih riliza i analizira promene u estimaciji troškova sledećih riliza.

Basili i saradnici objavili su i opisali jednostavan model regresije za estimaciju napora održavanja riliza (izdanja) različitih vrsta, kao što su ispravljanje grešaka i poboljšanje.

Ovaj model, kao promenljivu koristi broj linija koda (**Source lines of code - SLOC**), koja je merena kao suma dodatih, izmenjenih i obrisanih linija koda (**SLOC-ova**), uključujući komentare i praznine.

Predviđanje tačnosti nije objavljeno, iako je koeficijent determinacije relativno visok ($R^2 = 0,75$), što ukazuje na to da je SLOC dobar prediktor napora za održavanje.

Nakon početnog riliza sistema **Ramil i Lehman** su uveli ocene linearnih regresija modela za estimaciju napora potrebnog da se razvije sistem.

Njihovi modeli uzimaju u obzir sve zadatke održavanja neophodne za rast sistema koji mogu da uključuju korekcije greške, funkcionalna poboljšanja, tehnička unapređenja, itd. Prediktori modela su veličine mernih jedinica grubo merene preciznosti nivoa, moduli (broj dodatih, izmenjenih modula) i podsistemi (broj dodatih, izmenjenih), i broj izmena modula, plus svi izmenjeni moduli.

RELEASE-LEVEL MODELI: CAIVANO MODEL; SNEED MODEL

Caivano model koristi precizne metrike dobijene iz izvornog koda, dok je Sneed predložio ManCost model koji grupiše poslove održavanja u četiri različita tipa

Caivano i saradnici su opisali metod i sredstvo podrške za dinamičku kalibraciju modela procene napora obnove (reverzni inženjerинг и restauracija) projekata.

Model prihvata promenu informacija prikupljenih tokom izvršavanja projekta i kalibrirao se za bolju dinamiku održavanja, trenutne i buduće trendove projekta. Na početku projekta, model počinje sa svojim najčešćim oblikom kalibriranim iz završenih projekata. U toku izvođenja projekta, model može da promeni svoj prediktor i konstantu pomoću tehničke postepene regresije.

Metod koristi precizne metrike dobijene iz izvornog koda, kao što su broj linija izvornog koda, McCabe-ova ciclomatic složenost, Halstead-ova kompleksnost, broj modula dobijenih posle obnove procesa. Oni su validirali metode koristeći i podatke iz nasleđenog sistema i simulacije. Ustanovili su da su preciznije estimacije i modeli dinamične rekalibraciju efikasni za poboljšanje preciznosti modela i potvrdili da estimacija modela zavisi od procesa. Kasnije, empirijska studija dodatno proverava ove zaključke. Druge studije su takođe izvestile neke uspehe u poboljšanju tačnosti predviđanja podešavajući modele estimacije iterativnim razvojnim okruženjem.

Sneed je predložio model koji se zove **ManCost model** za estimaciju troškova održavanja softvera primenom različitih podmodela za različite tipove zadataka održavanja. Sneed grupiše poslove održavanja u četiri različita tipa, dakle, četiri podmodela :

- Korekcija grešaka: troškovi ispravljanja grešaka pre isporuke.
- Rutinske promene: troškovi održavanja i implementacije rutinskih promena.
- Funkcionalna poboljšanja: troškovi za dodavanje, menjanje, brisanje, poboljšanje funkcionalnosti softvera.
- Tehnička renoviranja: troškovi tehničkih unapređenja, kao što su performanse i optimizacija.

Sneed sugerira da ovi tipovi zadataka imaju različite karakteristike, dakle, svaki zahteva odgovarajuću estimaciju podmodela. Ipak, indeksi prilagođenosti i produktivnosti su uobičajene mere koje se koriste u ovim podmodelima. Prilagođena veličina je određena uzimajući u obzir efekte kompleksnosti i kvalitet faktora. Iako su primeri dati da objasne upotrebu podmodela za estimaciju napora održavanja, nije bilo prijavljenih valjanosti za estimaciju performanse ovih podmodela. Sneed je takođe predložio nekoliko proširenja na račun reinženjeringu zadataka i održavanja Web aplikacija.

RELEASE-LEVEL MODELI: FSM METODE

FSM metoda je namenjena merenju veličine softvera kvalifikovanjem funkcionalnih korisničkih zahteva softvera. COSMIC-FFP je uobičajena ISO standardizovana FSM metoda

Široko prihvatanje FSM (Functional size measurement) metode je privuklo veliki broj studija u cilju razvoja i unapređenja modela estimacije održavanja korišćenjem FSM metrike.

Većina ovih studija usmerena je na troškove adaptivnog održavanja koji poboljšavaju sistem dodavanjem, menjanjem i brisanjem postojećih funkcija. Pošto je utvrđeno da FPA (Function Point Analysis) ne prikazuju dobro dimenziju malih izmene, Abran i Maya predstavljaju proširenje FPA metode deljenjem funkcija nivoa složenosti na finije intervale. Ovo proširenje koristi korake manje dimenzije i odgovarajućih težina za diskriminaciju malih promena za koje su utvrđili da su uobičajene u okruženju za održavanje. Oni su potvrdili model na podacima

dobijenim iz finansijskih institucija i pokazali da tehnike finijih dimenzija bolje karakterišu veličinu karakteristike malih aktivnosti održavanja.

Abran i saradnici su izvestili o primeni **COSMIC-FFP metoda** merenja funkcionalne veličine u izgradnji modela estimacije napora za projekte adaptivnog održavanja. Oni su opisali korišćenje metode merenja funkcionalne veličine u dve oblasti studija, jedan sa modelima podignutim na 15 projekata implementacije funkcionalnih poboljšanja softverskog programa za jezičku aplikaciju, drugi sa 19 održavanja projekata u realnom vremenu, ugrađeni softver. Dve oblasti studija ne koriste isti skup pokazatela, ali oni uključuju tri metrike, napor i nivo težine projekta. Autori su pokazali da, dok god napor na projektu i funkcionalne veličine imaju pozitivnu korelaciju, ova korelacija je dovoljno jaka da izgradi dobre modele estimacije napora koji koriste meru iste veličine. Međutim, kako su pokazali, pouzdaniji modeli estimacije mogu biti izvedeni uzimajući u obzir doprinos drugih kategoričnih faktora, kao što je teškoća projekta.

Napomena: FSM (Functional size measurement) metoda je namenjena merenju veličine softvera kvalifikovanjem funkcionalnih korisničkih zahteva softvera. COSMIC-FFP je uobičajena ISO standardizovana FSM metoda.

FPA (Function Point Analysis) -analiza funkcionalnih tačaka meri funkcionalnu veličinu softvera gledajući (funkcionalne) transakcije i (logičke) datoteke podataka koje su relevantne za korisnika

TASK-LEVEL MODELI: BRIAND I BASILI MODEL

Model na nivou zadatka estimira troškove realizacije svakog održavanja koje dolazi u obliku izveštaja o greškama ili promenama zahteva.

Model na nivou zadatka estimira troškove realizacije svakog održavanja koje dolazi u obliku izveštaja o greškama ili promenama zahteva. Ovaj tip modela bavi se malim naporom estimacije, obično u rasponu od nekoliko sati do mesec dana.

Sneed je uveo proces seven-step i alat koji se zove SoftCalc za estimaciju veličine i troškova potrebnih za sprovodenje održavanja. Model koristi različite veličine mere, uključujući SLOC (**physical lines of code and statements**), funkcionalne tačke, objekat-tačke, a i tačke podataka. Napor održavanja je izračunat korišćenjem prilagođene veličine i indeksa produktivnosti. Umesto generisanja tačne estimacije napora, bilo bi korisno, da se klasificuje održavanje promena zahteva u pogledu nivoa težine ili nivoa potrebnog napora, i da se ove informacije koriste za planiranje sredstva klasifikacije respektivno.

Briand i Basili predložili su pristup modelovanja izgradnje modela za klasifikaciju napora održavanja promena zahteva. Postupak modeliranja podrazumeva četiri koraka na visokom nivou, identifikovanje predvidljive metrike, identifikovanje značenja predvidljive metrike, generisanje klasifikacija funkcija, kao i potvrđivanje modela. Opseg svake varijable i truda je podeljen u intervalima, svaki se predstavlja kao broj koji se zove indeks teškoće.

Opseg napora ima pet intervala (ispod jednog sata, od jedanog sata do jedanog dana, između jednog dana i jedne nedelje, između jedne nedelje i jednog meseca, više od mesec dana), koji su indeksirani, od 1 do 5, respektivno. Da bi se estimirao pristup, Briand i Basili koriste skup podataka o 163 promene zahteva iz četiri različita projekata na NASA Goddard Space Flight Centeru. Pristup proizvodi klasifikaciju ispravnosti modela od 74 % do 93 %.

Briand-ovo i Basili-evo modeliranje pristupa se može implementirati na dinamičkom konstruisanju modela prema specifičnim okruženjima. Organizacije mogu izgraditi opšti model koji koristi skup unapred definisanih pokazatelja, kao što su vrsta modifikacije; broj dodatih, izmenjenih, i izbrisanih komponenti, broj dodatih, izmenjenih i izbrisanih linija koda. Ovaj opšti model je primenjiv na početku faze. Međutim, kako je Basili istakao, teško je odrediti ulaze modela pravilno, jer oni nisu dostupni sve dok se promena sprovodi.

Basili i saradnici predstavljaju klasifikaciju modela koji klasificuje troškove dorada u biblioteci softverskih komponenti za višekratnu upotrebu, tj Ada fajlove.

TASK-LEVEL MODELI: JORGENSEN MODEL

Jorgensen je ocenio jedanaest različitih modela za estimaciju napora pojedinih poslova održavanja koristeći regresije, neuronske mreže, i obrazac pristupa priznavanja.

Jorgensen je ocenio jedanaest različitih modela za estimaciju napora pojedinih poslova održavanja koristeći regresije, neuronske mreže, i obrazac pristupa priznavanja. U poslednjem pristupu Optimized Set Reduction (OSR) korišćen je da izabere najrelevantniji podskup varijabli za prediktore napora. Svi modeli koriste veličinu za održavanja, koja se meri kao zbir dodatih, ažuriranih i izbrisanih linija koda (Source lines of code - SLOC-ova), kao glavne promenljive. Četiri druga prediktora su odabrana pošto su značajno povezana sa održavanjem produktivnosti. To su sve Indikatori prediktora:

1. **uzrok** (bez obzira da li je zadatak korektivno održavan) ,
2. **promena** (bez obzira da li se više od 50 % napora očekuje da bude potrošeno na izmene postojećeg koda u odnosu na umetanje i brisanje koda) ,
3. **modul** (ili ne više od 50% napora trebalo bi da bude potrošeno na razvoj novih modula) ,
4. **poverenje** (bez obzira da li sa održavaocem ima veliko poverenje u rešavanju održavanja zadataka).

Ostale varijable, tip jezika, iskustvo montera, prioritet zadataka, starosti aplikacija i veličine aplikacija, su pokazatelji da nema značajne korelacije za održavanje produktivnosti. Kao što je naveo Jorgensen, ovaj rezultat ne pokazuje da svaka od ovih varijabli nema uticaja na napor održavanja. Za proveru modela, Jorgensen koristi skup podataka od 109 slučajno odabranih poslova održavanja prikupljenih iz različitih aplikacija iste organizacije.

Od jedanaest modela izgrađenih i upoređenih, najbolji model čini vrstu dnevnika linearne regresije i hibridnog tipa, na osnovu prepoznavanja obrazaca i regresija.

Imajući u vidu nisko predviđanje tačnosti, Jorgensen je preporučio da se formalni model može koristiti kao dopuna ekspertima za predviđanja i predložio da Bayesian analiza može biti odgovarajući pristup koji kombinuje procene ispitivanih modela i stručna predviđanja.

▼ Poglavlje 6

Grupna vežba

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - PRVA FAZA: MONTIRANJE I PODEŠAVANJE RAZVOJNOG OKRUŽENJA

Kroz proces održavanja LAMS-a potrebno je proći kroz sve faze unutar sistema i odrediti potrebne resurse i plan rada.

Prva faza obuhvata montiranje i podešavanje razvojnog okruženja, modifikovanje i postavku servera, instalaciju operativnog sistema, softverskih alata i testiranje opreme. Resursi koji su planirani i potrebni za ovaj deo su:

1. Ljudski resursi:
 - Softverski inženjer
 - Administrator sistema
2. Materijalni resursi:
 - Server
 - PC1
 - PC2

Unutar ove faze obuhvaćene su dve stavke: Montiranje i postavljanje potrebne serverske opreme i Instalacija operativnog sistema i aplikacija na serveru. Planirani broj radnih dana za ove aktivnosti je 20 dana.

Montiranje i postavljanje potrebne serverske opreme. Za ovaj sistem koristi se server koji se nalazi u prostorijama Univerziteta sa definisanim hardverskim parametrima koji odgovaraju uputstvu za produpcionu instalaciju LAMS-a. Potrebno je odabratи operativni sistem (Linux distribucija) kao i potrebne prateće programe za uspešan rad sistema. Za ovaj deo predviđeni su dva administratora sistema koji će biti zaduženi

za istraživanje hardverskih zahteva i podešavanje opreme shodno njima. Predviđeno vreme za ovu fazu iznosi 14 dana.

Planirani ljudski resursi:

- Administrator sistema Č2 – 112 radnih sati
- Administrator sistema Č3 – 112 radnih sati

Administratori sistema zaduženi za rad sa LAMS-om dostavili su spisak potrebnih nadogradnji i dodataka osnovnom operativnom sistemu na serveru koje je potrebno instalirati i podesiti za rad na server. Aktivnost koja je po planu predviđena za 6 radnih dana i koja se može obavljati

tek nakon završene prethodnog montiranja i postavljanja serverske opreme. Planirana su dva administratora sistema i jedan softverski inženjer za ovu fazu.

Planirani ljudski resursi:

- Administrator sistema Č2 – 48 radnih sati
- Administrator sistema Č3 – 48 radnih sati
- Softverski inženjer Č4 – 48 radnih sati

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - DRUGA FAZA: PRIKUPLJANJE ZAHTEVA

Druga faza projekta podrazumeva analizu zahteva za softver koji su dostavljeni od strane rektora univerziteta i nastavnog kadra

Potrebno je proučiti i proveriti mogućnost ispunjenja traženih zahteva i dokumentovanje istih. Resursi koji su planirani i potrebni za ovaj deo su:

1. Ljudski resursi:
 - Vođa projekta
 - Softverski inženjer
 - Softverski inženjer
2. Materijalni resursi
 - PC1
 - PC2
 - PC3
 - PC4

Tri faze su planirane za ovu aktivnost:

- Prikupljanje zahteva za sistem,
- Analiza dokumentovanih zahteva i
- Predstavljanje i donošenje odluke o poslednjoj verziji zahteva.

U narednom delu svaka od faza biće detaljno opisana i predstavljena.

Prikupljanje zahteva za sistem: Najbitnija faza u razvoju svakog projekta. Prikupljanje zahteva podrazumeva planiranje funkcionalnosti novog sistema, njegovog korišćenja kao i svih stavki vezanih za njegov rad. Ukoliko se ova faza ne odradi na pravi način sADBina celog projekta je neizvesna i zato je potrebno sakupiti sve zahteve od svih korisnika sistema (profesora, studenata, dekana) i na osnovu njih doneti finalnu verziju koja će biti dokumentovana i po kojoj će se krenuti u implementaciju sistema. Za ovu fazu predviđena su dva softverska inženjera koja svojim znanjem i iskustvom treba da na pravi način objasne korisnicima šta je potrebno da dostave kao zahteve i na koji način da ih predstave timu. Predviđen vremenski rok iznosi četiri dana a inženjerima su na raspolaganju dva računara u prostorijama Univerziteta.

Planirani ljudski resursi:

- Softverski inženjer Č4 – 32 radnih sati
- Softverski inženjer Č5 – 32 radnih sati

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - DRUGA FAZA: ANALIZA ZAHTEVA I DONOŠENJE ODLUKE O ZAHTEVIMA

Nakon prikupljanja zahteva, sledeća faza predstavlja analizu dostavljenih zahteva i pripremu finalne verzije.

Analiza dokumentovanih zahteva: Softverski inženjeri koji su se bavili prikupljanjem zahteva radiće i na analizi dokumentovanih zahteva jer dobro poznaju te zahteve kao i osobe koje su ih dostavile, pa mogu zatražiti dodatna objašnjenja od istih. Analiziranje je predviđeno za dva radna dana u kojima će dva softverska inženjera spremiti zahteve za sledeću fazu. Planirana su dva računara za ovu aktivnost.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Planirani ljudski resursi:

- Softverski inženjer Č4 – 16 radnih sati
- Softverski inženjer Č5 – 16 radnih sati

Predstavljanje i donošenje odluke o poslednjoj verziji zahteva: Prisustvo vođe projekta u ovoj fazi je od ključnog značaja za razvoj celog sistema, na njemu je da doneše konačnu odluku o zahtevima kojima će se voditi administratori i softverski inženjeri u narednim aktivnostima implementacije sistema. Za ovu fazu planirana su tri računara, dva softverska inženjera i vođa projekta, dok je predviđeni vremenski rok za obavljanje tri dana.

Planirani ljudski resursi:

- Vođa projekta – 24 radnih sati
- Softverski inženjer Č4 – 24 radnih sati
- Softverski inženjer Č5 – 24 radnih sati

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - TREĆA FAZA: RAZVOJ SISTEMA

Ova faza projekta podrazumeva razvijanje sistema, instalaciju potrebne verzije na server, implementaciju nove baze podataka, kreiranje modula koji su predviđeni zahtevima itd.

Pored razvijanja podrazumeva se instalacija potrebne verzije na server, implementacije nove baze podataka, kreiranja modula koji su predviđeni zahtevima i svega ostalog što je predviđeno za uraditi u ovoj fazi. Takođe, unos nastavnog plana i programa kao i materijala

koji ide u sklopu svakog predmeta se podrazumevaju na kraju ove faze, kada je potrebno kompletirati i spremiti sistem za testiranje.

Podizanje produkcionе verzije na serveru: Prva faza unutar Razvoja je podizanje produkcionе verzije na postavljen server, na kome su prethodno podešeni svi parametri za ovu aktivnost. Nakon istraživanja i proučene dokumentacije o sistemu, na ovoj aktivnosti je predviđen jedan administrator sistema koji ima zadatak da obavi ovu aktivnost za tri radna dana i uspostavi funkcionalnu vezu sistema sa ostalim članovima tima. Na raspolaganju su mu tri računara sa kojih može pristupiti serveru.

Implementacija modifikovane baze podataka: U skladu sa izmenjenom bazom podataka i novim zahtevima koji su stigli od strane rektora i nastavnog kadra potrebno je izvršiti implementaciju kreirane baze podataka. Baza će biti postavljena na server nakon instalacije produkcionе verzije i produkcionе baze. Potrebno je nizom upita ka bazi ispitati njen rad i podesiti sve nove upise u postojeće tabele. Baza će biti SQL, jer dosadašnja baza nije dovoljno pouzdana i može doći do njenog preopterećenja usled velikog broja materijala. Biće angažovana dva softverska inženjera i jedan administrator sistema na tri računara u vremenskom periodu od pet radnih dana.

Unos korisničkih naloga u okviru sistema: Kada je implementirana nova SQL baza podataka na administratorima sistema je da izvrše unos novih korisnika u sistem. Predviđeno je da se upitima u okviru stare baze Univerziteta Metropolitan dobiju potrebne informacije o svim aktivnim korisnicima i da se nakon toga prenesu u novi sistem. Potrebno je voditi evidenciju o broju unetih korisnika da se ne bi duplirali nalozi i time stvarali netačni podaci u okviru nove baze, što može izazvati negativne posledice u drugim fazama razvoja. Predviđen je jedan radni dan za administratore da izvrše unos.

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - ČETVRTA FAZA: TESTIRANJE

Nakon razvojnog dela projekta prelazi se na petu fazu, testiranje i provera, svega do sada kreiranog i implemetiranog u novi sistem.

Potrebno je uočiti eventualne probleme ili pronaći greške koje je moguće ispraviti jer ukoliko se sistem pusti u proces sa greškama to će biti jako teško ispraviti uzimajući u obzir veliki broj koji koristi sistem za internet učenje u svakom trenutku u toku dana.

Testiraće se korisnički nalozi, nova baza podataka, implementirani korisnički interfejs, moduli koji su pravljeni za vezu sa DITA sistemom a biće testirano preuzimanje nastavnog materijala i izgled strana za prikaz predavanja.

Testiranje korisničkih naloga: Prilikom unosa svih korisnika iz stare baze definisana su korisnička imena i lozinke koje obuhvataju ime, prezime kao i broj indeksa u kombinaciji sa brojevima 1,2 i 3. Treba proveriti nekoliko studentskih i profesorskih naloga da bi se utvrdilo da se nije provukla neka greška u unosu kredencijala koji će biti naknadno izmenjeni prilikom prvog pristupanja sistemu svakog od korisnika. Predviđeno vreme za ovu aktivnost je jedan dan za jednog testera sistema.

Testiranje baze podataka prilikom unosa velikog broja nastavnih materijala: Da bi se utvrdilo ponašanje baze podataka potrebno je isprobati unos svih podržanih fajlova kao i materijala koji će biti unet. Neka od pitanja koja se postavljaju odnose se na količinu tekstualnih fajlova kao i video materijala koji može biti veličine od nekoliko stotina megabajta,

kako će se ponašati baza u ovim uslovima koji predstavljaju realnu situaciju. Jedan tester sistema će izvršiti ovu aktivnost za jedan radni dan.

AKTIVNOSTI ODRŽAVANJA LAMS SISTEMA - ČETVRTA FAZA: DORADA

Poslednja faza projekovanja sistema predstavlja deo u kome je potrebno izvršiti sve modifikacije i izmene postojećeg kreiranog sistema ukoliko su se pojavili nedostatci u testiranju određenih de

Ovo predstavlja bitnu fazu jer ukoliko se na ovom nivou reše ključni problemi i izvrše izmene sistem će biti spreman za početak nastavnog semestra i neće biti potrebno “gasiti” ga u toku semestra i onemogućavati studentu da prate nastavu.

Modifikovanje i ispravljanje nepravilnosti u sistemu uočenih nakon testiranja: Ova faza je predviđena ukoliko se u prethodnoj fazi testiranja nađe na neki problem ili nedostatak koji je potrebno otkloniti pre puštanja sistema u rad i predstavljanja studentima. Ukoliko dođe do modifikovanja u ovoj fazi kao što je već navedeno izbegava se dodatni trošak nemogućnosti praćenja nastavnog materijala u toku semestra kao i dodatni rad na aktivnostima koje su već prošle testiranje. Vremenski rok je sedam dana a na njemu će biti uključena dva softverska inženjera i dva administratora sistema.

Za predstavljanje vežbe je potrebno 45 minuta.

▼ 6.1 Individualna vežba

PLAN ODRŽAVANJA SOFTVERA

Na osnovu prethodnih vežbi potrebno je kreirati plan održavanja softvera i proći kroz sve faze unutar odabranog softverskog sistema i odrediti potrebne resurse i plan rada.

Svaki student treba da za ISUM ili neki drugi poznati sistem koji može naći na internetu

1. Izradi Plan održavanja softvera korišćenjem nekog od obrazaca (template) za dokument Plan održavanja softvera koji se može naći na internetu (na primer sa neke od adresa https://images.techstreet.com/direct/SWM_samples.pdf, https://cds.cern.ch/record/1277556/files/EMI-DSA1.1-1277556-Software_Maintenance_Support_Plan-v1.0.pdf). Prilikom izrade plana održavanja treba obuhvatiti aktivnosti održavanja softvera datim u ovom i prethodnom predavanju kao i korišćenjem ostale raspoložive dokumentacije. (**vreme izrade zadatka 45 minuta**)

2. Za svaku od aktivnosti održavanja treba da odredi potrebne resurse i plan rada tako da se mogu predvideti troškovi održavanja po aktivnostima (**vreme izrade zadatka 45 minuta**)

✓ Poglavlje 7

Domaći zadatak

TRINAESTI DOMAĆI ZADATAK

Nakon trinaeste lekcije potrebno je uraditi dvanaesti domaći zadatak.

Za odabranu proizvoljnu aplikaciju koju ste razvili na nekom od predmet uraditi

1. Plan održavanja softvera kojim treba obuhvatiti aktivnosti održavanja softvera datim u ovom i prethodnom predavanju
2. Za svaku od aktivnosti održavanja treba da odrediti potrebne resurse tako da se mogu predvideti troškovi održavanja po aktivnostima

U zip arhivi poslati dokument kao i modelovane dijagrame u navedenim okruženjima.

Domaći zadaci treba da budu realizovani u razvojnog okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ13-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ13.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Zaključak

PROCENA VRSTE, OBIMA I TROŠKOVA (MODELI ESTIMACIJE) POSLOVA U PROCESU ODRŽAVANJA SW

Mnogo pažnje koja je posvećena objavljivanju estimacije može ukazivati široko na rasprostranjenost prakse u održavanju softvera, da je održavanje organizovano u nizu operativni

Jasno je iz održavanja modela za estimaciju, da estimacija troškova razvoja, isporuke i izdanja dobija mnogo pažnje. Mnogo pažnje koja je posvećena objavljivanju estimacije, može ukazivati široko na rasprostranjenost prakse u održavanju softvera, da je održavanje organizovano u nizu operativnih izdanja. Svako izdanje uključuje poboljšanja i izmene koje se mogu meriti i koristiti kao ulazne veličine za model estimacije napora. Izvorni kod je osnovna mera za dominantnu veličinu ulaza, to potvrđuje činjenicu da su ove merne jedinice osnovna metrika za model estimacije napora u novijem razvoju. Promena i dodavanje koda se koriste u većini modela, dok neki modeli koriste i brisanje koda.

S druge strane, čak i sa modelima koji su generički dovoljni, postoji nedostatak empirijskih studija dokumentovanja i potvrđivanja prijava predloženog modela u kontekstu održavanja u više organizacija sa raznolikim procesima, tehnologijama i ljudima. Izgradnja generičkih modela koji mogu biti ponovno iskorišćeni na određene organizacije ili čak i projekte će biti mnogo potrebniji u estimaciji održavanja.

Može se očekivati da će značaj održavanja i unapređivanja softverskih sistema u budućnosti još više rasti i da će cena ovih procesa višestruko prevazići cenu razvoja. Najveći deo prihoda softverskih kompanija dolaziće od održavanja i podrške za postojeće sisteme, dok će sam inicijalni proizvod biti sve jeftiniji.

Razvojni model otvorenog koda ([Open source](#)) pruža dobru sliku moguće budućnosti u ekonomskom smislu - sam proizvod je besplatan (ili vrlo jeftin), dok se zarade ostvaruju od održavanja, obuke, unapređivanja i svih drugih vidova podrške.

LITERATURA ZA LEKCIJU 13

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura: 1. P. Grubb and A.A. Takang, Software Maintenance: Concepts and Practice, 2nd ed., World Scientific Publishing, 2003. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)

2. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)

2. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)

4. V. B. Spasojević, i ostali, Organizacija sistema kvaliteta i alati kvalitet, Industrija 2004, vol. 32, br. 4, str. 91-107

Dopunska literatura: 1. Samir Lemeš, Nevzudin Bužadija, STANDARDNI MODELI ODRŽAVANJA SOFTVERA, 5. Konferencija „ODRŽAVANJE - MAINTENANCE 2018“ Zenica, B&H, 10. - 12. Maj 2018.

2. Lj. Lazić , Software Quality & Testing Metrics, WSEAS 7th WSEAS EUROPEAN COMPUTING CONFERENCE (ECC '13), Dubrovnik, Croatia, June 25-27, 2013 . (link: <http://www.wseas.org/wseas/cms.action?id=4102>+

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.pisa.rs>



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Tehnike održavanja,
podmlađivanja i redizajna
softvera

Lekcija 14

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 14

TEHNIKE ODRŽAVANJA, PODMLAĐIVANJA I REDIZAJNA SOFTVERA

- ✓ Tehnike održavanja, podmlađivanja i redizajna softvera
- ✓ Poglavlje 1: Reupotrebljivost (višekratna upotrebljivost)
- ✓ Poglavlje 2: Tehnologija višekratne upotrebljivosti
- ✓ Poglavlje 3: Iskustva sa višekratnom upotrebljivošću
- ✓ Poglavlje 4: Uticaj kvaliteta softvera na proces održavanja
- ✓ Poglavlje 5: Alati za održavanje softvera
- ✓ Poglavlje 6: Grupna vežba
- ✓ Poglavlje 7: Individuelne vežbe
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Tehnike održavanja softvera

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

U ovoj lekciji je fokus stavljen na osobine višestruke primenljivosti, moguće vrste i pristupe koji se mogu koristiti.

Pod višekratnom upotrebljivošću softvera podrazumevamo ponovnu upotrebu bilo kog dela softverskog sistema: dokumentacije, koda, konstrukcije, zahteva, slučajeva za testiranje i drugog. Postoje dve vrste višekratnog korišćenja, kako se to određuje iz perspektive korisnika. Proizvođač višekratno koristi komponente, ali ih i potrošač ponovo koristi u sledećim sistemima. Pristupi višekratnoj upotrebljivosti su ili kompozicioni ili generativni. Aktivnost koja čini zajedničku osnovu za obe vrste višekratne upotrebljivosti je analiziranje domena aplikacije sa ciljem da se identifikuju zajedničke oblasti i načini za njihovo opisivanje.

Postoje iskustva mnogih kompanija sa višekratnom upotrebljivošću, npr. kompanije Raytheon, GTE koja su predstavljena u ovom predavanju.

Koristi višekratne upotrebljivosti se sastoje u: povećanju produktivnosti i interoperabilnosti sistema, povećanju kvaliteta softvera i produktivnosti itd.

▼ Poglavlje 1

Reupotrebljivost (višekratna upotrebljivost)

VIŠEKRATNA UPOTREBLJIVOST SOFTVERA

Pod višekratnom upotrebljivošću softvera podrazumevamo ponovnu upotrebu bilo kog dela softverskog sistema: dokumentacije, koda, konstrukcije, zahteva, slučajeva za testiranje i drugog.

Višekratna upotrebljivost (reusability) zahteva da procenimo proizvode iz postojećih i budućih razvojnih projekata, da bismo odredili da li su oni takve vrste i kvaliteta kakav bismo želeli za sledeći sistem koji budemo gradili. Dakle, razmotrićemo detaljnije neka pitanja u vezi sa višekratnom upotrebljivošću koja bi mogla da utiču na naše procene.

Mnogi softverski sistemi koje gradimo slični su jedan drugom; na primer, svaka kompanija verovatno ima računovodstveni sistem, personalni sistem i možda sistem za naplatu. Postoje zajednički delovi među sistemima sa sličnom namenom. Na primer, svaki kadrovske sistem uključuje podatke i funkcije koje se odnose na radnike u kompaniji.

Umesto da stalno pravimo nove sisteme, možemo da pregledamo komponente iz starijih sistema i odredimo da li one mogu da se prilagode ili čak cele ponovo iskoriste za novi sistem. Čak i kada su aplikacije vrlo različite, moramo da ispitamo da li je odmah potrebno pisati drugu rutinu za sortiranje, komponentu za pretraživanje ili program za ulaz ekrana. Softverski istraživači veruju da višekratna upotrebljivost ima veliki potencijal za povećavanje produktivnosti i smanjivanje troškova, a neki radovi na primeni višekratne upotrebljivosti potvrđuju to njihovo uverenje.

Pod višekratnom upotrebljivošću softvera podrazumevamo ponovnu upotrebu bilo kog dela softverskog sistema: dokumentacije, koda, konstrukcije, zahteva, slučajeva za testiranje i drugog. Basili (1990) smatra da održavanje treba posmatrati u smislu višekratne upotrebljivosti; uzimamo postojeći sistem i ponovo koristimo njegove delove da bismo izgradili sledeću verziju.

VRSTE VIŠEKRATNE UPOTREBLJIVOSTI

Postoje dve vrste višekratnog korišćenja, kako se to određuje iz perspektive korisnika. Proizvođač višekratno koristi komponente, ali ih i potrošač ponovo koristi u sledećim sistemima

Postoje dve vrste višekratnog korišćenja, kako se to određuje iz perspektive korisnika. Proizvođač višekratno koristi komponente, ali ih i potrošač ponovo koristi u sledećim sistemima (Bollinger i Pfleeger 1990).

Kao potrošači, možemo da koristimo ili ceo proizvod, bez modifikacije (to se zove višekratna upotrebljivost crne kutije), ili da ga modifikujemo kako bi odgovarao našim posebnim potrebama (višekratna upotrebljivost bele ili providne kutije).

Višekratnu upotrebljivost crne kutije izvodimo često, kada koristimo rutine iz matematičkih biblioteka, ili pravimo grafičke interfejse iz unapred definisanih skupova komponenti.

Važno pitanje u višekratnoj upotrebljivosti tipa crne kutije je naša sposobnost da utvrdimo da višekratno upotrebljiv modul obavlja funkciju koja nam je potrebna na prihvatljiv način.

Ako je komponenta kod, želimo da budemo sigurni da je bez greške; ako je to skup testova, želimo da smo sigurni da potpuno izvršava funkciju za koju je namenjen.

Višekratna upotrebljivost providne kutije (to se ponekad zove višekratna upotrebljivost bele kutije) je češća, ali još uvek kontroverzna, jer rad uložen da bi se komponenta razumela i modifikovala, mora da bude manji od onog koji je potreban da bi se napisala nova ekvivalentna komponenta.

Da bi se taj problem rešio, mnoge višekratno upotrebljive komponente su izgrađene sa više parametara da bi se jednostavnije prekrajale spolja gledano, i da projektanti ne bi bili prinuđeni da se upuštaju u razumevanje komponente iznutra.

Nekoliko dostignuća u softverskom inženjerstvu čini višekratnu upotrebljivost izvodljivom. Jedno od njih je interes i usvajanje objektno orijentisanog razvoja (**object-oriented development, OOD**).

S obzirom na to da OOD ohrabruje softverske inženjere da grade konstrukciju oko nepromenljivih komponenti, mnogi delovi konstrukcije i koda mogu da se primene na više sistema. Druga prednost koja ohrabruje višekratnu upotrebljivost je povećana složenost alata koji protežu svoje efekte kroz celokupan životni ciklus razvoja.

PRISTUPI VIŠEKRATNOJ UPOTREBLJIVOSTI

Pristupi višekratnoj upotrebljivosti su ili kompozicioni ili generativni.

Alati koji uključuju spremište za komponente razvijene za potrebe jednog sistema, mogu biti podsticajni za višekratnu upotrebljivost tih komponenti u kasnijim sistemima. Najzad, konstrukcija jezika može da ohrabruje ponovno korišćenje komponenti. *Na primer, Ada obezbeđuje mehanizme za restrukturisanje softvera koji podržavaju pakovanje, a njene tehnike parametrizacije korisne su jer omogućuju generalizaciju softvera.*

Pristupi višekratnoj upotrebljivosti su ili kompozicioni ili generativni.

Kompoziciona višekratna upotrebljivost posmatra višekratno upotrebljive komponenti kao skup gradivnih blokova, a razvoj se obavlja odozdo nagore, gradeći ostatak sistema oko raspoloživih ponovo upotrebljivih komponenti.

Za takvu vrstu višekratne upotrebljivosti, komponente se skladište u biblioteci (koja se često zove spremište višekratno upotrebljivih komponenti). Komponente moraju da se klasifikuju ili uvedu u kataloge, a sistem za izvlačenje mora da se koristi da bi se pregledale i birale komponente za upotrebu u novom sistemu. *Na primer, naše spremište bi moglo da sadrži rutinu za pretvaranje vremena iz 12-časovnog u 24-časovni format (odnosno, da 9:54 P.M. postane 21.54).*

Slično tome, generator sistema za upravljanje bazom podataka Genesis obuhvata gradivne blokove koji nam dozvoljavaju da konstruišemo DBMS (sistem za upravljanje bazom podataka) skrojen prema našim potrebama (Prieto-Diaz 1993).

S druge strane, generativna višekratna upotrebljivost je specifična za određeni domen aplikacije. To znači da su komponente projektovane posebno da reše potrebe određene aplikacije i da se kasnije ponovo koriste u sličnim aplikacijama. Na primer, Nasa je razvila dosta softvera za praćenje satelita u orbiti. Komponente za analizu efemerida mogu da se projektuju tako da budu ponovo upotrebljive u više Nasinih sistema. Međutim, ti moduli nisu dobri kandidati za opšte spremište višekratno upotrebljivih komponenti, zato što oni verovatno neće biti potrebni u drugim domenima.

Generativna višekratna upotrebljivost obećava isplativost visokog potencijala, a praksa se usredstvila na generatore aplikacija u specifičnim domenima. Neki od najpoznatijih takvih generatora su Lex i Yacc, koji su projektovani da generišu leksičke analizatore i parsere.

ZAJEDNIČKA OSNOVA KOMPOZICIONE I GENERATIVNE VIŠESTRUE UPOTREBLJIVOSTI

Aktivnost koja čini zajedničku osnovu za obe vrste višekratne upotrebljivosti je analiziranje domena aplikacije sa ciljem da se identifikuju zajedničke oblasti i načini za njihovo opisivanje

Aktivnost koja čini zajedničku osnovu za obe vrste višekratne upotrebljivosti je analiza domena, proces je analiziranje domena aplikacije sa ciljem da se identifikuju zajedničke oblasti i načini za njihovo opisivanje (Prieto-Diaz 1987). Kompoziciona višekratna upotrebljivost se oslanja na analizu domena da bi se identifikovale zajedničke funkcije nižeg nivoa u širokoj klasi sistema. Generativni pristup zahteva aktivniju analizu; analitičar domena pokušava da definiše generički domen arhitekture i da specifikuje standarde interfejsa između delova arhitekture.

Te ideje su dovele do pojmove horizontalne i vertikalne višekratne upotrebljivosti. Vertikalna višekratna upotrebljivost se bavi istom oblašću aplikacije ili domenom, ali horizontalna višekratna upotrebljivost ukršta domene. Višekratna upotrebljivost Nasinih rutina za efemeride je vertikalna, ali ponovna upotreba programa za sortiranje iz sistema baze podataka u grafičkom sistemu je horizontalna.

▼ Poglavlje 2

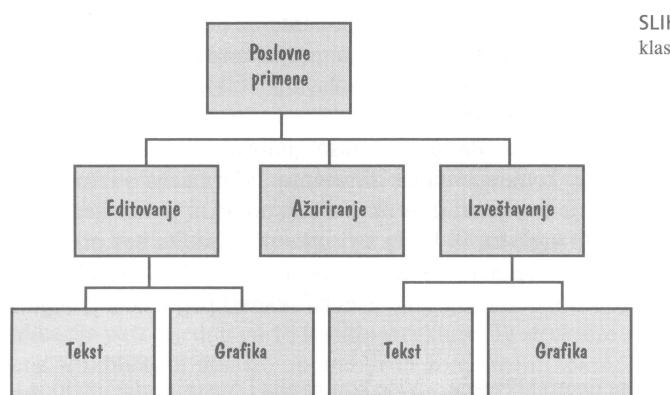
Tehnologija višekratne upotrebljivosti

SISTEMI ZA KLASIFIKACIJU KOMPONENTI ZA VIŠEKRATNU UPOTREBLJIVOST

Umesto da se za klasifikaciju komponenti koristi hijerarhija, svaka komponenta se opisuje pomoću uređene liste karakteristika koje se zovu facete

Jedna od najvećih prepreka za višekratnu upotrebljivost je potreba da se pretražuje u velikom skupu softverskih proizvoda kako bi se pronašao onaj koji je najbolji za neku posebnu potrebu. Taj posao je sličan prevrtanju gomile knjiga, umesto da se jede u biblioteku. Rešenje je u klasifikaciji komponenti, gde se zbirke višekratno upotrebljivih komponenti organizuju i uvode u kataloge prema određenoj klasifikacionoj šemi.

Komponente je moguće klasifikovati u hijerarhijskoj šemi, vrlo slično kao što su knjige organizovane u biblioteci. Kategorija na najvišem nivou se deli u podkategorije, koje se opet dele na podkategorije i tako dalje, kao što je prikazano na slici 1. Međutim, taj pristup nije fleksibilan i novi predmeti mogu lako da se dodaju samo na najnižem nivou.



Slika 2.1 Primer hijerarhijske klasifikacione šeme [Izvor: NM SE321 -2020/2021.]

Štaviše, potrebno je obimno unakrsno referenciranje, zato što, na primer, moramo da budemo u stanju da nađemo rutinu koja radi sa tekstrom pod izveštavanjem, kao i pod editovanjem.

Rešenje tog problema je [Prieto-Diazova facetna šema klasifikacije](#) (Prieto-Diaz i Freeman 1987). Umesto da se koristi hijerarhija, svaka komponenta se opisuje pomoću uređene liste karakteristika koje se zovu facete. Faceta je vrsta deskriptora koji pomaže da se identifikuje

komponenta, a zbirka faceta nam dozvoljava da identifikujemo više karakteristika odjednom. Na primer, facete ponovno upotrebljivog koda mogu da budu:

- aplikacijski domen;
- funkcija;
- jezik za programiranje;
- operativni sistem.

Svaka komponenta koda se identificuje stavkom za svaku od faceta.

Na primer, određena rutina bi mogla da bude označena kao
<switching system, sort, telephone number, Ada, UNIX>

SIŠTEM ZA IZVLAČENJE KOMPONENTI ZA VIŠEKRATNU UPOTREBLJIVOST

Sistem za klasifikaciju se podržava pomoću sistema za izvlačenje iz spremišta, koji može da pretražuje i izvlači komponente u skladu sa korisnikovim opisom.

Sistem za klasifikaciju se podržava pomoću sistema za izvlačenje iz spremišta, automatizovane biblioteke koji može da pretražuje i izvlači komponente u skladu sa korisnikovim opisom.

Spremište često ima rečnik sinonima koji nam pomaže da razumemo terminologiju upotrebljenu u klasifikaciji. Na primer, rečnik može da nam kaže da pokušamo upotrebu reči „search“ kao sinonima za „look up“.

Pored toga, spremište bi trebalo da rešava problem konceptualne bliskosti za izdvajanje sličnih faceta, koji nisu baš identični željenoj komponenti. Na primer, ako ne može da se nađe komponenta „modify“, možemo da se usmerimo na „add“ i „delete“, koje su slične.

Sistem za izvlačenje, takođe, može da zapisuje informacije o zahtevima korisnika. Veliki broj neispunjениh zahteva određene vrste može da pokrene bibliotekara da predloži pisanje određene vrste komponente.

Na primer, ako više korisnika traži komponentu koja menja datum iz američkog formata (11/05/98, ili November 5, 1998) u evropski format (05/11/98, ili 5. novembar 1998), a takva komponenta ne postoji, bibliotekar može da obezbedi da se ona napravi.

Slično tome, ako više projektanata traži komponente za crtanje geometrijskih oblika, ali nijedna nije na raspolaganju, bibliotekar može da identifikuje dobre kandidate i da ih doda u biblioteku.

Najzad, sistem za izvlačenje može da sačuva opisnu informaciju o komponenti koja nam pomaže da odlučimo koju komponentu da izaberemo. Informacija o izvoru komponente (odnosno, o tome gde je kupljena, ili u okviru kog projekta je razvijena), njenoj pouzdanosti (broju otkrivenih grešaka, ili broju sati u kome je radila bez otkaza), ili u kom kontekstu je prethodno korišćena, može da nas ubedi da koristimo neku komponentu, a ne neku drugu.

▼ Poglavlje 3

Iskustva sa višekratnom upotrebljivošću

ISKUSTVO KOMPANIJE RAYTEON

Jedan od prvih izveštaja o iskustvima sa ponovnom upotrebom je onaj iz kompanije Raytheon (Lanergan i Grasso 1984).

Više kompanija i organizacija imalo je uspeha u ponovnom korišćenju značajne količine svog softvera. U svakom slučaju, zajednička karakteristika bila je predana i bezuslovna podrška rukovodstva (Braun i Prieto-Diaz 1990). Takva podrška je neophodna, zato što rukovodstvo mora da podnese dodatne troškove koje proizvođač ima zbog višekratne upotrebljivosti, dok te troškove ne potrebuje ušteda koju će imati na strani potrošača.

Štaviše, višekratna upotrebljivost često izgleda radikalno i preteće projektantima koji su navikli da pišu softver od samog početka.

Jasno je da mnoga društvena, kulturološka i pravna pitanja, kao i ona tehnička stoje u vezi sa višekratnom upotrebom softvera .

Jedan od prvih izveštaja o iskustvima sa ponovnom upotrebom je onaj iz kompanije Raytheon (Lanergan i Grasso 1984). Organizacija za informacione procesne sisteme kompanijinog sektora za raketne sisteme, zapazila je da je 60% konstrukcija i kodova njenih poslovnih aplikacija bilo redundantno i da su oni zato dobri kandidati za standardizaciju i ponovno korišćenje. Osnovan je program ponovnog korišćenja i ispitano je preko 5000 COBOL izvornih programa, koji su klasifikovani u tri velike klase modula: za editovanje, ažuriranje i izveštavanje.

U Rayteonu su takođe otkrili da je većina njihovih poslovnih aplikacija pala u jednu od tri logičke strukture. Te strukture su standardizovane i napravljena je biblioteka ponovo upotrebljivih komponenti. Nove aplikacije su varijacije standardnih softverskih struktura i grade se sklapanjem komponenti iz biblioteke. Softverski inženjeri su obučeni za korišćenje biblioteke i testirano je njihovo prepoznavanje kada neka struktura može ponovo da se upotrebi da bi se izgradila nova aplikacija. Ponovno korišćenje je sada obavezno. Posle šest godina ponovnog korišćenja koda na taj način, Raytheon je izvestio da je novi sistem sadržao u proseku 60% višekratno upotrebljenog koda, što je povećalo produktivnost za 50%.

ISKUSTVO KOMPANIJE GTE

Jedan od najuspešnijih programa višekratne upotrebljivosti o kome postoji izveštaj u literaturi, bio je onaj u GTE-u.

Jedan od najuspešnijih programa višekratne upotrebljivosti o kome postoji izveštaj u literaturi, bio je onaj u GTE-u. GTE Data Services je osnovala Asset Management Program da bi razvila kulturu višekratne upotrebljivosti u korporaciji. Program je bio sličan onima koje smo do sada opisivali. Počelo se od analize postojećih sistema i identifikovanja ponovo upotrebljivih objekata.

Svaki softverski proizvod koji bi mogao u potpunosti ili delimično da se ponovo upotrebi, postao je kandidat za takav objekat. Zbirka objekata je zatim klasifikovana i uvedena u katalog u automatizovanom bibliotečkom sistemu. U cilju održavanja biblioteke, promovisanja višekratne upotrebljivosti i podrške onima koji rade sa višekratno upotrebljivim kodom, formirano je nekoliko grupa:

- **rukovodeća grupa za podršku**, da bi obezbedila inicijativu, fondove i politiku višekratne upotrebljivosti;
- **grupa za identifikaciju i kvalifikaciju**, da bi identifikovala moguća oblasti višekratne upotrebljivosti i identifikovala, kupovala i sertifikovala nove priloge zbirci;
- **grupa za održavanje**, da bi održavala i ažurirala nove višekratno upotrebljive komponente;

U cilju održavanja biblioteke, formirano je nekoliko grupa:

- **razvojna grupa**, da bi pravila nove višekratno upotrebljive komponente;
- **grupa za podršku korisnicima**, da bi im pomogla i obučila ih da ispituju i procenjuju komponente koje se mogu ponovo upotrebiti.

Ne želeći da višekratna upotrebljivost postane obavezna, GTE je uvela podsticaje i nagrade. Programerima je plaćeno do 100 USD po svakoj komponenti koja je prihvaćena u biblioteci, a autori programa su dobijali stimulaciju kad god bi neka njihova komponenta bila ponovo upotrebljena u novom projektu. Ustanovljena je nagrada „Ponovni korisnik meseca“, a menadžerima su davani bonusi ili povećanja budžeta za ostvarenje ciljeva višekratne upotrebljivosti. U prvoj godini programa, jedna od firmi je izvestila o 14% višekratnosti na svojim projektima, što je donelo uštede u vrednosti od 1,5 miliona USD (Swanson i Curry 1987).

ISKUSTVA SA VIŠEKRATNOM UPOTREBLJIVOŠĆU OSTALIH KOMPANIJA

Širom sveta pokrenuti su i drugi programi višekratne upotrebljivosti; većina njih usredsredila se na ponovno korišćenje koda.

U kompaniji Nipon Novel, koja je zapošljavala oko 100 softverskih inženjera, započeo je program višekratne upotrebljivosti u kome su takođe korišćeni novčani podsticaji. Plaćano je 5 centi (SAD) po redu koda projektantu koji bi ponovo upotreboio neku komponentu, a tvorac te komponente bi dobio 1 cent po redu koda koji bi bio ponovo upotrebljen. U 1993. godini, program je kompaniju koštao 10 000 USD, što je daleko manje od cene pisanja ekvivalentnog koda (Frakes i Isoda 1994).

Širom sveta pokrenuti su i drugi programi višekratne upotrebljivosti; većina njih usredstvila se na ponovno korišćenje koda. Vladine organizacije, kao što su European Space Agency i U.S. Department of Defense imaju velike institucionalizovane programe za ponovno korišćenje sa spremištima komponenti.

Komercijalne kompanije, kao što su Hewlett-Packard, Motorola i IBM, takođe su dosta uložile u višekratnu upotrebljivost. I Evropska zajednica je bila sponzor više poduhvata u vezi sa višekratnom upotrebljivošću, uključujući Reboot i Surprise.

Međutim, Griss i Wasser (1995) nas podsećaju da efikasno ponovno korišćenje ne zahteva ni široku implementaciju, ni usavršenu automatizaciju.

KORISTI VIŠEKRATNE UPOTREBLJIVOSTI: POVEĆANJE PRODUKTIVNOSTI I INTEROPERABILNOST SISTEMA

Neke od koristi su povećanje produktivnosti i interoperabilnost sistema

Višekratna upotrebljivost obećava povećanje produktivnosti i kvaliteta u celom procesu razvoja softvera. Produktivnost može da se poveća ne samo smanjivanjem vremena kodiranja, nego i smanjivanjem vremena testiranja i izrade dokumentacije.

Neka od najvećih smanjenja troškova nude se ponovnim korišćenjem zahteva i konstrukcije, zato što ona ima efekat talasa: ponovno korišćenje konstrukcije automatski obuhvata ponovno korišćenje koda, na primer. Štaviše, ponovno korišćenje komponenti može da poveća performansu i pouzdanost, zato što komponente mogu da se optimizuju i provere pre nego što se stave u spremište.

Dugoročna korist je poboljšana interoperabilnost sistema. Objavljeni su opisi pozitivnih efekata koje višekratna upotrebljivost može daima na razvoj; vrlo malo ih je govorilo o zamkama i nedostacima.

Primer dobre analize ponovnog korišćenja je Limova (1994), u kojoj se opisuju napori na ponovnom korišćenju, preduzeti kod Hewlett-Packarda.

Njegova pažljiva procena dva velika programa višekratne upotrebljivosti pokazuje značajna povećanja kvaliteta i produktivnosti i veliko smanjenje vremena prodaje. U tabeli na slici 1 dat je sumarni pregled njegovih nalaza.

Neka od najvećih smanjenja troškova nude se ponovnim korišćenjem zahteva i konstrukcije, zato što ona ima efekat talasa: ponovno korišćenje konstrukcije automatski obuhvata ponovno korišćenje koda, na primer.

TABELA Kvalitet, produktivnost i vreme prodaje kod Hewlett-Packarda (prilagođeno iz Lima 1994) ©1996 IEEE

Karakteristika projekta	Hewlett-Packard Projekat 1	Hewlett-Packard Projekat 2
Veličina	1100 nekomentarisanih izvornih iskaza	700 nekomentarisanih izvornih iskaza
Kvalitet	51% smanjenje greške	24% smanjenje greške
Proektivnost	57% povećanje	40% povećanje
Vreme za prodaju	Podaci neraspoloživi	42% smanjenje

Slika 3.1 Kvalitet, produktivnost i vreme prodaje kod Hewlett-Packarda [Izvor: NM SE321 -2020/2021.]

KORISTI VIŠEKRATNE UPOTREBLJIVOSTI: POVEĆANJA KVALITETA I PRODUKTIVNOSTI

Ponovno korišćenje komponenti može da poveća performansu i pouzdanost, zato što komponente mogu da se optimizuju i provere pre nego što se stave u spremište.

Štaviše, ponovno korišćenje komponenti može da poveća performansu i pouzdanost, zato što komponente mogu da se optimizuju i provere pre nego što se stave u spremište. Dugoročna korist je poboljšana interoperabilnost sistema. Objavljeni su opisi pozitivnih efekata koje višekratna upotrebljivost može da ima na razvoju; vrlo malo ih je govorilo o zamkama i nedostacima. Primer dobre analize ponovnog korišćenja je Limova (1994), u kojoj se opisuju napori na ponovnom korišćenju, preduzeti kod Hewlett-Packarda. Njegova pažljiva procena dva velika programa višekratne upotrebljivosti pokazuje značajna povećanja kvaliteta i produktivnosti i veliko smanjenje vremena prodaje. U tabeli na slici 2 dat je sumarni pregled njegovih nalaza.

TABELA Kvalitet, produktivnost i vreme prodaje kod Hewlett-Packarda (prilagođeno iz Lima 1994) ©1996 IEEE

Karakteristika projekta	Hewlett-Packard Projekat 1	Hewlett-Packard Projekat 2
Veličina	1100 nekomentarisanih izvornih iskaza	700 nekomentarisanih izvornih iskaza
Kvalitet	51% smanjenje greške	24% smanjenje greške
Produktivnost	57% povećanje	40% povećanje
Vreme za prodaju	Podaci neraspoloživi	42% smanjenje

Slika 3.2 Kvalitet, produktivnost i vreme prodaje kod Hewlett-Packarda [Izvor: NM SE321 -2020/2021.]

On je takođe uporedio troškove sa višekratnom upotrebljivošću i troškove izrade koda od početka u tri projekta. U tabeli na slici 3 prikazani su njegovi rezultati, koji nagoveštavaju da u stvari postoje značajni dodatni troškovi u stvaranju višekratno upotrebljivog koda i u njegovoj upotrebni.

Joos (1994) nudi pogled na upravljačku stranu pokretanja programa višekratne upotrebljivosti. U objašnjavanju pilot studija višekratne upotrebljivosti u kompaniji Moto rola, ona ističe potrebu za dobrom obukom, upravljanjem očekivanjima i dobijanjem unapred definisane obaveze da se komponente višekratno koriste.

TABELA Troškovi proizvodnje i višekratne upotrebljivosti kod Hewlett-Packarda (Lim 1994) © 1996 IEEE

	Sistem za kontrolu vazdušnog saobraćaja (%)	Sistem za upravljanje menijima i formularima (%)	Grafički firmver (%)
Relativni troškovi da se napravi višestruko upotrebljiv kod	200	120 do 180	111
Relativni troškovi ponovnog korišćenja	10 do 20	10 do 63	19

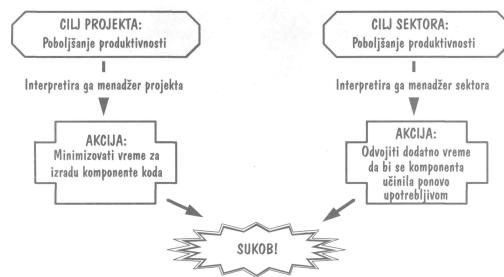
Slika 3.3 Troškovi proizvodnje i višekratne upotrebljivosti kod Hewlett-Packarda [Izvor: NM SE321 -2020/2021.]

ISKUSTAVA SA PROGRAMIMA VIŠEKRATNE UPOTREBLJIVOSTI

Ciljevi višekratne upotrebljivosti mogu da budu kontradiktorni, pa rukovodstvo mora na vreme i jasno da ih razreši.

Pfleeger (1996) takođe predstavlja priču o ponovnoj upotrebljivosti, formulišući mešavinu iskustava sa programima višekratne upotrebljivosti koji nisu bili tako uspešni kao oni koje smo do sada opisali. Ona nagoveštava više ključnih lekcija koje su dobijene iz tih pokušaja:

- Ciljevi višekratne upotrebljivosti treba da budu merljivi, tako da možemo da kažemo da li smo ih ispunili.
- Ciljevi višekratne upotrebljivosti mogu da budu kontradiktorni, pa rukovodstvo mora na vreme i jasno da ih razreši. Na primer, kao što je pokazano na slici 4, cilj koji menadžer sektora želi da postigne višekratnom upotrebljivošću može da bude u sukobu sa ciljem menadžera projekta, pa da se nikada ne završi nijedan takav projekat.



Slika 3.4 Sukobljavanje interpretacije ciljeva [Izvor: NM SE321 -2020/2021.]

- Različite perspektive mogu da stvore različita pitanja. Dok programeri postavljaju specifična tehnička pitanja, kao što je: „Da li je rutina za sortiranje napisana na jeziku C++?”, potpredsednik se usredsređuje na korporaciju kao celinu i njen položaj na tržištu pitajući: „Da li smo već uštedeli pare?”. Čak i kada su pitanja u suštini ista, možemo da dobijemo različite odgovore. Različite tačke gledišta odražavaju različite prioritete za učesnike u programu višekratne upotrebljivosti.
- Svaka organizacija mora da odluči na kom nivou se određena ključna pitanja postavljaju i na njih odgovara. Na primer: ko plaća troškove višekratne upotrebljivosti? Kada se troškovi unesu u projekat u nadi da će se oni nadoknaditi u budućim projektima, oni moraju da odražavaju udružene izbor i nameru. Ko je vlasnik višekratno upotrebljivih komponenti? Ko je njihov vlasnik i ko je odgovoran za njihovo održavanje mora da se odredi na višem nivou. Ko pravi komponente? Drugi nivo mora da bude odgovoran za konstrukciju, punjenje i podršku biblioteka za višekratne upotrebljivosti, kao i za merenje podataka koje one generišu. Ta pitanja su povezana sa načinom na koji je organizovano poslovanje, tako da su brige oko profita i gubitaka u vezi sa strategijama ulaganja i stavljanja u funkciju.

PITANJA PRINCIPA VIŠEKRATNE UPOTREBLJIVOSTI

Pfleeger (1996) takođe postavlja pitanja na koja mora da se odgovori da bi se princip višekratne upotrebljivosti uspešno sproveo

Pitanja i odgovori moraju da budu podržani merenjem, pa je važno da se zna ko pita i kome treba odgovor.

Integrišite procese višekratne upotrebljivosti u proces razvoja, ili učesnici neće znati kada se od njih očekuje da se drže višekratne upotrebljivosti.

Povežite merenja sa procesom višekratne upotrebljivosti da biste mogli da merite svoj proces i da ga poboljšate. Neka vaši poslovni ciljevi nagoveste šta da se meri.

Pfleeger (1996) takođe postavlja pitanja na koja mora da se odgovori da bi se princip višekratne upotrebljivosti uspešno sproveo:

- Da li imate pravi model višekratne upotrebljivosti?
- Šta su kriterijumi za uspeh?
- Kako postojeći modeli troškova mogu da se doteraju da bismo posmatrali zbirke projekata, a ne samo pojedinačne projekte? Odluke o mogućim uštedama u troškovima zahtevaju tehnike procene troškova koje obuhvataju više projekata. Ali, većina komercijalnih modela troškova se usredstavlja na pojedinačne projekte, a ne na zbirke projekata potrebne da bi se opravdalo ulaganje u višekratnu upotrebljivost.

Potrebno je modifikovati postojeće ili izraditi nove modele troškova, koji mogu da integrišu aspekte višestrukih projekata da bi se podržale odluke o višekratnoj upotrebljivosti.

- Kako se uobičajeni pojmovi iz računovodstva (kao što su povraćaj ili ulaganje) uklapaju u višekratnu upotrebljivost? Za one koji su visoko u poslovodstvu, ulaganje u višekratnu upotrebljivost isto je kao bilo koja druga investicija. Njih zanimaju opcije, indikatori i povraćaj od investicije, a ne broj računara, jezici ili tehnike pretraživanja biblioteke. Za softverske inženjere je važno da terminologija višekratne upotrebljivosti može da se prevede u računovodstvenu terminologiju da bi investicija u višekratnu upotrebljivost mogla da se uporedi sa drugim investitorskim inicijativama u korporaciji.
- Ko je odgovoran za kvalitet komponente? Šema mora da se analizira i organizuje tako da odgovara korporativnoj kulturi. Šta se dešava kada autor napusti kompaniju? Šta se događa kada se komponenta razvija ili se umnožava u više verzija? Koji kriterijumi se uspostavljaju za kvalitet komponente, kako za početno prihvatanje u biblioteci, tako i za održavanje komponente kako se ona menja u toku vremena?

PITANJA PRINCIPIA VIŠEKRATNE UPOTREBLJIVOSTI - ODGOVORNOST ZA KVALITET PROCESA

Neko treba da probere neiskorišćene komponente i analizira ih da bi odredio zašto se one ne koriste.

- Ko je odgovoran za kvalitet procesa i održavanje? Višekratna upotrebljivost je više od uspostavljanja biblioteka i njihovog punjenja komponentama. Neko treba da nadgleda nivo ponovnog korišćenja, obezbeđujući da se dobre komponente identifikuju, ispravno obeleže i često koriste. Neko treba da probere neiskorišćene komponente i analizira ih da bi odredio zašto se one ne koriste. Neko mora da proceni proces analize domena da bi odredio da li su ciljne komponente stvarno ponovo upotrebljive. A neko treba da proučava efekat ponovnog korišćenja na nivou cele korporacije, da bi se izračunao povraćaj od investicije i osmišljavala buduća politika ponovnog korišćenja. Te aktivnosti se mogu obavljati na različitim nivoima u korporaciji i moraju da se prepoznaju kao neophodni delovi programa višekratne upotrebljivosti.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 4

Uticaj kvaliteta softvera na proces održavanja

TEHNIKE KVALITETA SOFTVERA KOJE UTIČU NA ODRŽAVANJE

Što je sistem složeniji, izmena utiče na više komponenti. Iz tog razloga, upravljanje konfiguracijom, koje je važno u toku razvoja, postaje kritično u vreme održavanja.

Jedan od načina da se smanji rad u procesu održavanja jeste da se kvalitet ugradi od samog početka. Pokušaj da se dobra konstrukcija i struktura sprovede na već izgrađenom sistemu, manje je uspešan od pravilne gradnje sistema od početka. Međutim, pored dobre prakse, postoji više drugih tehnika koje podižu razumljivost i kvalitet. To su:

1. Upravljanje konfiguracijom
2. Upravljanje izmenama
3. Analiza uticaja
4. Horizontalna i vertikalna pratljivost

UPRAVLJANJE KONFIGURACIJOM

Upravljanje konfiguracijom, koje je važno u toku razvoja, postaje kritično u vreme održavanja.

Praćenje izmena i njihovih efekata na druge komponente sistema nije lak zadatak. Što je sistem složeniji, izmena utiče na više komponenti. Iz tog razloga, upravljanje konfiguracijom, koje je važno u toku razvoja, postaje kritično u vreme održavanja.

S obzirom na to da kupci i korisnici iniciraju mnogobrojne izmene u fazi održavanja (kako se pojavljuju otkazi ili zahtevaju poboljšanja), ustanavljava se odbor za upravljanje konfiguracijom sa ciljem da nadzire unošenje izmena. **Odbor čine predstavnici svih zainteresovanih strana, uključujući kupce, projektante i korisnike. Svakom problemu se pristupa na sledeći način:**

- 1. Problem otkriva korisnik, kupac, ili projektant, koji evidentira simptome na propisanom obrascu za upravljanje izmenom. Alternativno, kupac, korisnik, ili projektant zahteva poboljšanje: novu funkciju, varijaciju stare funkcije, ili uklanjanje postojeće funkcije. Obrazac, sličan izveštaju o otkazu koji smo proučili u prethodnoj lekciji, mora da sadrži

informacije o sadašnjem načinu rada sistema, prirodi problema ili poboljšanja i budućem načinu rada sistema.

- 2. O predloženoj promeni izveštava se odbor za upravljanje konfiguracijom.
- 3. **Odbor za upravljanje konfiguracijom se sastaje da bi raspravlja o problemu.** Prvo, on određuje da li predlog predstavlja otkaz u sklopu zahteva, ili se radi o zahtevu za poboljšanje. Ta odluka obično utiče na to ko će da finansira resurse potrebne za implementaciju izmena.
- 4. **Za otkaz o kome je izvešten, odbor za upravljanje konfiguracijom raspravlja o verovatnom uzroku problema.** Za zahtevano poboljšanje, odbor raspravlja o delovima sistema na koje će verovatno uticati izmena. U oba slučaja, programeri i analitičari mogu da opišu opseg svake neophodne izmene i očekivano vreme njihove implementacije. Upravni odbor dodeljuje prioritet ili nivo ozbiljnosti zahtevu, a programera ili analitičara imenuje odgovorno lice za realizaciju izmene u sistemu.
- 5. **Imenovani analitičar ili programer pronalazi odakle potiče problem ili pronalazi komponente na koje se odnosi zahtev i zatim identifikuje neophodne izmene.** Radeći na testnom primerku a ne na operativnoj verziji sistema, programer ili analitičar implementira i testira izmene da bi bio siguran da one rade.
- 6. **Imenovani analitičar ili programer radi sa programskim bibliotekarem u cilju kontrolisanja implementacije izmena u operativnom sistemu.** Sva relevantna dokumentacija se ažurira.
- 7. **Programer ili analitičar pravi izveštaj o promeni u kome detaljno opisuje sve izmene.**

NAJKRITIČNIJI KORAK U PROCESU UPRAVLJANJA KONFIGURACIJOM

Najkritičniji korak u procesu je onaj pod brojem 6 koji se odnosi na rad sa programskim bibliotekarem u cilju kontrolisanja implementacije izmena.

Najkritičniji korak u procesu je onaj pod brojem 6 koji se odnosi na rad sa programskim bibliotekarem u cilju kontrolisanja implementacije izmena. U svakom trenutku, tim za upravljanje konfiguracijom mora da zna stanje svake komponente ili dokumenta u sistemu. Shodno tome, upravljanje konfiguracijom bi trebalo da pojača komunikaciju između onih čije akcije utiču na sistem. Cashman i Holt (1980) sugerisu da je neophodno uvek znati odgovore na sledeća pitanja:

1. **Sinhronizacija:** Kada je napravljena izmena?
2. **Identifikacija:** Ko je napravio izmenu?
3. **Imenovanje:** Koje komponente sistema su izmenjene?
4. **Provera autentičnosti:** Da li je izmena napravljena ispravno?
5. **Autorizovanje:** Ko je odobrio da se napravi izmena?
6. **Rutiranje:** Ko je bio obavešten o izmeni?
7. **Obustavljanje:** Ko može da obustavi zahtev za izmenom?
8. **Delegiranje:** Ko je odgovoran za izmenu?
9. **Vrednovanje:** Koji je prioritet izmene?

10. Imajte u vidu da su to pitanja rukovođenja, a ne tehnička pitanja. Da bismo pažljivo upravljali izmenom, moramo da koristimo procedure.

UPRAVLJANJE IZMENAMA

U svakom trenutku, tim za upravljanje konfiguracijom mora da zna stanje svake komponente ili dokumenta u sistemu.

Upravljanje izmenom može da se potpomogne sprovođenjem više konvencija. Prvo, svakoj radnoj verziji se dodeljuje identifikacioni kod ili broj. Kada se verzija menja, revizioni kod ili broj se dodeljuje svakoj promjenjenoj komponenti. Pravimo zapis o svakoj verziji komponente i statusu, kao i istorijat svih izmena. Potom, u svakom trenutku životnog veka sistema, tim za upravljanje konfiguracijom može da identifikuje sadašnju verziju radnog sistema i broj revizije svake komponente u upotrebi.

Tim takođe može da otkrije kako se različite verzije razlikuju, ko je pravio izmene i zašto su one napravljene. Iz vaše studentske perspektive, ove konvencije upravljanja konfiguracijom verovatno izgledaju nepotrebne. Vašim školskim projektima verovatno upravljaju pojedinci ili male grupe programera, koristeći verbalnu komunikaciju da bi se pratile izmene i poboljšanja.

Međutim, zamislite haos koji bi nastao zbog korišćenja tih istih tehnika u razvoju i održavanju sistema od 200 komponenti. Veliki sistem često razvija više nezavisnih grupa koje istovremeno rade na različitim aspektima sistema; ponekad se te grupe nalaze u različitim delovima grada, ili čak i u različitim gradovima.

Kada loša komunikacija dovede do otkaza, tim za upravljanje konfiguracijom mora da bude u stanju da ponovo vrati sistem u njegovo prethodno stabilno stanje; taj korak može da se preduzme samo kada tim tačno zna ko je napravio kakve izmene, na kojim komponentama i kada.

ANALIZA UTICAJA

Analiza uticaja je procena stepena rizika u vezi sa promenom, uključujući procenu uticaja na resurse, rad i vremenske rokove.

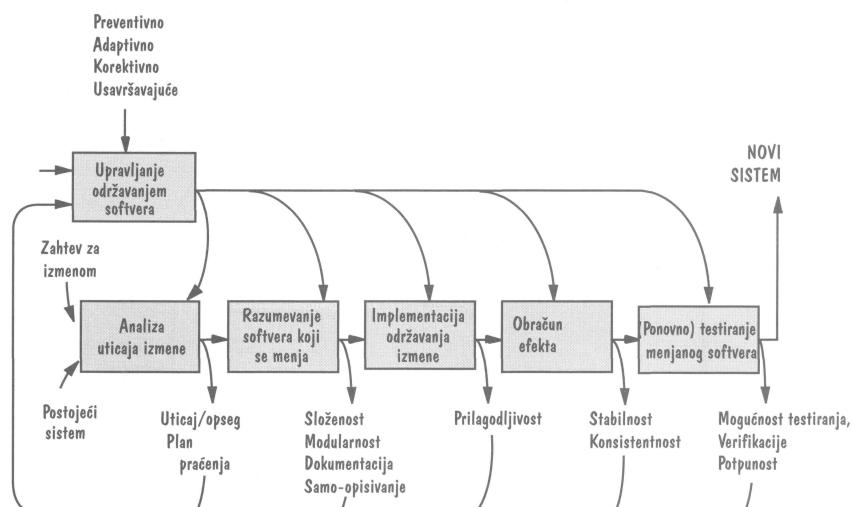
Prema tradicionalnom shvatanju životnog veka softvera, održavanje počinje po završetku razvoja softvera.

Međutim, održavanje softvera zavisi od korisničkih zahteva i počinje još sa njima. Prema tome, principi razvoja dobrog softvera primenjuju se i na proces razvoja i na proces održavanja. S obzirom na to da razvoj dobrog softvera podržava njegovu izmenu, o njoj treba razmišljati u toku celog životnog veka softverskog proizvoda. Štaviše, naizgled minorna izmena je često obimnija (i zato skuplja za implementaciju) nego što se to očekuje. **Analiza uticaja je procena stepena rizika u vezi sa promenom, uključujući procenu uticaja na resurse, rad i vremenske rokove.**

Efekti mnogostruktih izmena u sistemu mogu da se vide u neodgovarajućoj ili zastareloj dokumentaciji koja nastaje u nepravilno ili nepotpuno popravljanom softveru, loše strukturisanom dizajnu ili kodu, artefaktima koji ne podležu standardima i još mnogo čemu. Problem je mešavina usložnjavanja, dužeg vremena potrebnog da bi projektanti razumeli kod koji je pretrpeo izmene i više sporednih uticaja koje izmene mogu da imaju na druge delove sistema.

Ti problemi povećavaju troškove održavanja, a rukovodstvo bi želelo da te troškove drži pod kontrolom. Analizu uticaja možemo da koristimo da bismo podržali upravljanje troškovima održavanja.

Pfleeger i Bohner (1990) su istraživali načine za merenje uticaja koji predložena izmena ima na razne aspekte softvera da bi odredili rizike i odmerili različite opcije. Oni opisuju model održavanja softvera koji uključuje merenje povratne sprege. Na dijagramu na slici 1, ilustrovane su aktivnosti koje se izvode kada se zahteva izmena, a natpsi uz strelice u donjoj polovini dijagrama predstavljaju merenja koja daju informacije koje menadžeri mogu da koriste pri donošenju odluke kada i kako da se napravi izmena.



Slika 4.1 Aktivnosti održavanja softvera [Izvor: NM SE321 -2020/2021.]

HORIZONTALNA PRATLJIVOST

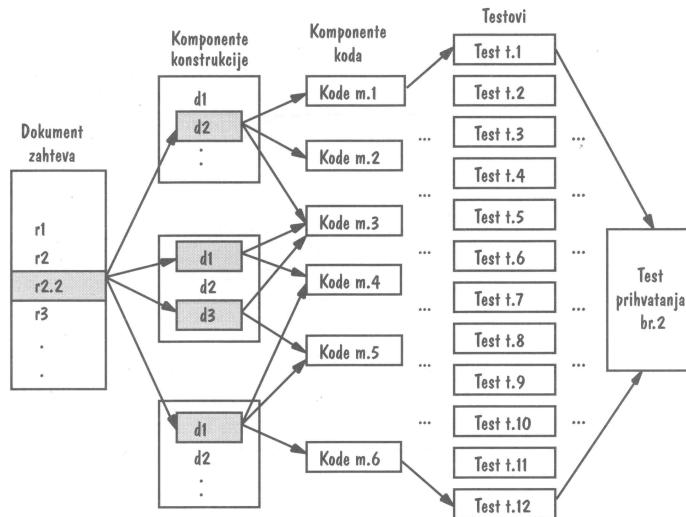
Horizontalna pratljivost bavi se odnosima komponenti u kolekciji proizvoda. I horizontalnu i vertikalnu pogodnost za praćenje možemo da prikažemo koristeći usmerene grafove

Horizontalna pratljivost bavi se odnosima komponenti u kolekciji proizvoda. Na primer, svaka komponenta konstrukcije prati se do nivoa komponenti koda koje implementiraju taj deo konstrukcije. Obe vrste pratljivosti su potrebne za razumevanje potpunog skupa odnosa u postupku analize uticaja.

I horizontalnu i vertikalnu pogodnost za praćenje možemo da prikažemo koristeći usmerene grafove. Usmereni graf je jednostavno zbirka objekata koji se zovu čvorovi i pridružena zbirka

uređenih parova čvorova, koji se zovu grane. Prvi čvor grane zove se čvor izvor, a drugi je čvor odredište. Čvorovi predstavljaju informacije koje se nalaze u dokumentima, člancima i drugim artefaktima. Svaki artefakt sadrži čvor za svaku komponentu. Na primer, možemo da predstavimo projekat kao zbirku čvorova, sa po jednim čvorom za svaku komponentu konstrukcije, a specifikaciju zahteva tako da ima po jedan čvor za svaki zahtev. **Usmerene grane predstavljaju odnose unutar jednog proizvoda i između proizvoda.**

Na slici 2, ilustrovano je kako se određuju grafički odnosi i pratljive veze između odgovarajućih proizvoda.



Slika 4.2 Horizontalna pratljivost u softverskim proizvodima rada [Izvor: NM SE321 -2020/2021.]

Ispitujemo svaki zahtev i crtamo vezu između zahteva i komponenti konstrukcije koje ga implementiraju. Zatim, povezujemo svaku komponentu konstrukcije sa komponentama koda koje je implementiraju. Natzad, povezujemo svaki modul koda sa skupom slučajeva za testiranje. Povezanost koja se tako dobije formira graf koji prikazuje odnose između proizvoda.

HORIZONTALNA PRATLJIVOST - GRAF UKUPNE PRATLJIVOSTI

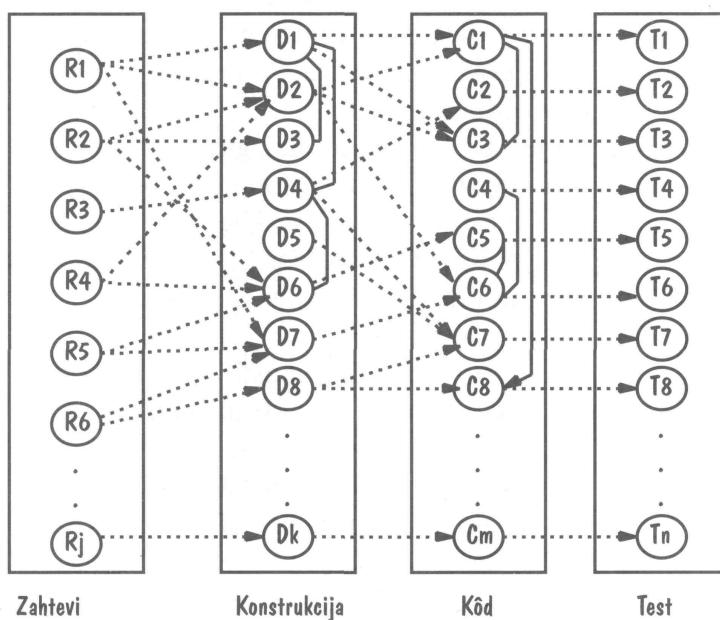
U graf ukupne pratljivosti svaki artefakt glavnog procesa (zahtevi, konstrukcija, kod i test) prikazan je kao pravougaonik oko konstitutivnih čvorova.

Na slici 3, prikazano je kako bi mogao da izgleda graf ukupne pratljivosti. **Svaki artefakt glavnog procesa (zahtevi, konstrukcija, kod i test) prikazan je kao pravougaonik oko konstitutivnih čvorova.** Pune grane unutar svakog pravougaonika su odnosi vertikalne pratljivosti komponenti u pravougaoniku. **Isprekidane grane između pravougaonika prikazuju veze horizontalne pratljivosti u sistemu.** Postoji više dokaza da su neke mere složenosti dobri indikatori potrebnog rada i učestalosti grešaka (Card i Glass 1990.) Ti pojmovi mogu da

se prošire na karakteristike grafa pratljivosti, da bi se procenio uticaj predložene izmene. Uzmimo, na primer, graf vertikalne pratljivosti unutar svakog pravougaonika na slici 3.

Ukupan broj čvorova, broj grana čije je odredište u čvoru (tako zvani ulazni stepen čvora) i grana čiji je izvor u čvoru (izlazni stepen), zatim mere kao što je ciklomatski broj, mogu da se izračunaju pre i posle izmene.

Ako izgleda da se veličina i složenost grafa sa izmenom povećavaju, verovatno je da će se veličina i složenost odgovarajućih proizvoda takođe povećavati. Koristeći tu informaciju, odbor za upravljanje konfiguracijom može da odluči da implementira izmenu na drugačiji način, ili da to uopšte ne radi. Čak i ako rukovodstvo odluči da uvede predloženu izmenu, rizik koji ide uz nju može bolje da se razume pomoću slike zasnovane na merenjima.



Slika 4.3 Graf ukupne pratljivosti [Izvor: NM SE321 -2020/2021.]

VERTIKALNA PRATLJIVOST

Mere vertikalne pratljivosti su one mere proizvoda koje pokazuju dejstvo izmene na svaki proizvod koji se održava.

Mere vertikalne pratljivosti su one mere proizvoda koje pokazuju dejstvo izmene na svaki proizvod koji se održava. Izmerene karakteristike na grafu horizontalne pratljivosti daju procesni pogled na izmenu.

Za svaki par proizvoda, možemo da formiramo podgrafove sledećih odnosa:

- **prvi podgraf sa odnosima između odgovarajućeg zahteva i konstrukcije,**
- **drugi - sa odnosima između odgovarajuće konstrukcije i koda, i**
- **treći - koji pokazuje odnose između koda i test slučajeva.**

Nakon toga, merimo veličinu i složenost odnosa da bismo odredili veličinu štetnosti uticaja. Štaviše, možemo da sagledamo ukupnu horizontalnu pratljivost da bismo videli da li je ukupna pratljivost posle izmene postaje otežana ili olakšana. Pfleeger i Bohner (1990) su tražili minimalni skup putanja koje obuhvata taj graf; ako se broj putanja povećava posle izmene, onda će sistem verovatno biti nezgrapniji i teži za održavanje. Slično tome, ako se ulazni i izlazni stepeni značajno povećaju, sistem će se u budućnosti teže održavati.

▼ Poglavlje 5

Alati za održavanje softvera

AUTOMATIZOVANI ALATI ZA ODRŽAVANJE

Praćenje statusa svih komponenti i testova je ogroman posao. Na sreću, postoji više automatizovanih alata koji nam pomažu u održavanju softvera.

Praćenje statusa svih komponenti i testova je ogroman posao. Na sreću, postoji više automatizovanih alata koji nam pomažu u održavanju softvera. Ovde opisujemo neke vrste takvih alata.

Editori teksta: Editori teksta su na više načina korisni za održavanje. Prvo, editor može da kopira kod ili dokumentaciju sa jednog mesta na drugo, sprečavajući greške koje se javljaju kada sami pravimo duplikat teksta. Drugo, kao što smo ranije videli, neki editori teksta u posebnim datotekama prate izmene koje se odnose na osnovnu datoteku. Mnogi od tih editora označavaju vreme i podatke za svaki unos i obezbeđuju način da se vrati sa trenutne verzije datoteke na prethodnu, ako je to potrebno.

Komparator datoteka: je koristan alat za vreme održavanja, koji poređi dve datoteke i izveštava o njihovim razlikama. Često ga koristimo kako bismo se uverili da su dva sistema ili programa stvarno istovetni, ako radimo s takvom pretpostavkom. Program čita obe datoteke i ukazuje na nesaglasnosti.

Kompajleri i linkerji: Kompajleri i linkerji često imaju svojstva koja pojednostavljaju održavanje i upravljanje konfiguracijom. Kompajler proverava sintaksne greške u kodu, ukazujući u mnogim slučajevima na mesto i vrstu greške. Kompajleri nekih jezika, kao što su Modula-2 i Ada, takođe proveravaju doslednost preko posebno prevedenih komponenti.

Kada se kod ispravno prevede, linker (povezivač ili editor veza) povezuje kod sa drugim komponentama koje su potrebne za izvršavanje programa. Na primer, linker povezuje datoteku `fbyname.h` sa odgovarajućom datotekom `filename.c` u jeziku C. Ili, linker može da naznači poziv potprograma, biblioteke ili makroa, automatski donoseći potrebne datoteke da bi napravio celinu koju je moguće prevesti.

Neki linkerji takođe prate brojeve verzija svake od zahtevanih komponenti, tako da se povezuju samo odgovarajuće verzije. Ta tehnika pomaže da se odstrane problemi čiji su uzrok pogrešne kopije sistema ili podistema kada se ispituje izmena.

NEKI KORISNI ALATI ZA ODRŽAVANJE SOFTVERA

Upravljanje konfiguracijom bilo bi nemoguće bez biblioteka informacija koje kontrolišu proces izmene.

Alati za pronalaženje i uklanjanje grešaka: Alati za pronalaženje i uklanjanje grešaka pomažu pri postepenom praćenju logike programa, ispitujući sadržaj registara i memorijskih područja i postavljajući indikatore i pokazivače.

Generatori unakrsnih referenci: O značaju pratljivosti govorili smo ranije u ovoj lekciji. Automatizovani sistemi generišu i skladište unakrsne reference da bi obezbedili i razvojnom timu i timu za održavanje bolje upravljanje izmenama u sistemu. Na primer, neki **alati** za unakrsne reference deluju kao skladišta sistemskih zahteva, a takođe sadrže i veze ka drugim sistemskim dokumentima i kodu koji se odnose na pojedinačne zahteve. Kada se predloži izmena zahteva, možemo da upotrebimo alat da bi nam on saopštio na koje druge zahteve i delove dizajna i koda će one uticati. Neki alati za unakrsno referenciranje sadrže skup logičkih formula koje se zovu uslovi za verifikaciju; ako svaka od formula daje vrednost „istinito“, onda kod zadovoljava specifikaciju na osnovu koje je nastao. To svojstvo je posebno korisno za vreme održavanja, da bi se obezbedilo da promenjeni kod bude i dalje u saglasnosti sa svojim specifikacijama.

Statički analizatori koda: Statički analizatori koda proračunavaju informacije o strukturnim atributima koda, kao što su dubina ugnježdavanja, broj obuhvatnih putanja, ciklomatski broj, broj redova koda i nedostižnih iskaza. Te informacije možemo da izračunamo kada pravimo nove verzije sistema koje održavamo, da bismo videli da li one postaju veće, složenije i teže za održavanje. Merenja nam takođe pomažu da odaberemo jednu varijantu dizajna među onima koje su nam ponuđene, posebno kada ponovo projektujemo delove postojećeg koda.

Spremišta upravljanja konfiguracijom: Upravljanje konfiguracijom bilo bi nemoguće bez biblioteka informacija koje kontrolišu proces izmene. Ta spremišta mogu da sadrže izveštaje o problemima, uključujući informacije o svakom od njih, o organizaciji koja je o njemu izvestila i organizaciji koja ga rešava. Neka spremišta dozvoljavaju korisnicima da čuvaju beleške o najavljenim problemima u sistemima koje koriste.

▼ Poglavlje 6

Grupna vežba

ZAŠTO PRIMENJIVATI VIŠEKRATNU UPOTREBLJIVOST?

Višekratna upotrebljivost se smatra jednim od osnovnih ciljeva kvalitetnog dizajna.

Višekratna upotrebljivost se smatra jednim od osnovnih ciljeva kvalitetnog dizajna.

Mnoge karakteristike objektno-orientisanog sistema su namenjene upravo unapređenju mogućnosti višekratne upotrebljivosti. Višekratna upotrebljivost na dizajn utiče na tri načina:

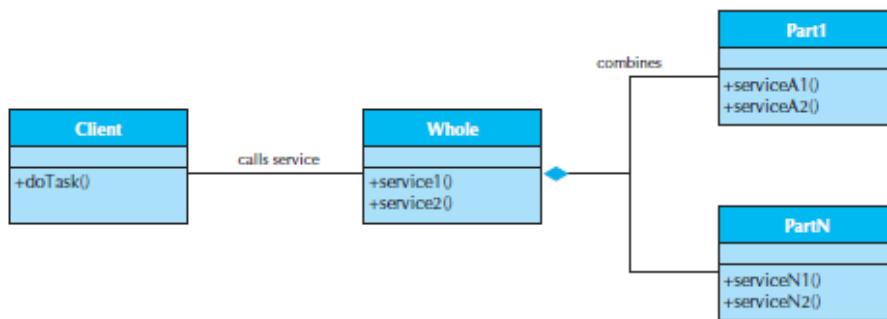
1. prvo, treba razmotriti kako se višekratna upotrebljivost može realizovati primenom nasleđivanja;
2. drugo, treba ispitati mogućnost korišćenja uzorka (pattern) koji predstavljaju templejte za dizajniranje rejuzibilnih elemenata;
3. treće, treba pokušati da se postojeće klase učine reupotrebljivim, bilo direktno bilo izdvajanjem podklasa.

U dizajnu radi primene višekratne upotrebljivosti, postoji mogućnost korišćenja uzorka dizajna, okvira (eng. **framework**) biblioteka i komponenti o čemu će ovde biti reči.

Na slikama 1 i 2 su dati primeri dva uzorka dizajna: Uzorak Celina – deo (eng. **Whole-Part pattern**) i uzorka Command Processor čija je

Vreme trajanja vežbe 45 minuta

primarna svrha da natera dizajnera da eksplicitno izdvoji interfejs kao objekat Command od izvršenja operacije (Supplier) koja se dešava iza interfejsa. Neki uzorci dizajna podržavaju različite fizičke arhitekture. Na primer uzorak Forwarder-Receiver podržava peer-to-peer arhitekturu.



Slika 6.1 Uzorak Celina – deo (eng. Whole-Part pattern), [Izvor: NM SE321 -2020/2021.]

Slika 6.2 Uzorak Command Processor [Izvor: NM SE321 -2020/2021.]

OKVIRI, BIBLIOTEKE KLASA I KOMPONENTE KAO NAČIN ZA POSTIZANJE VIŠEKRATNE UPOTREBLJIVOSTI

U dizajnu, radi primene višekratne upotrebljivosti, pored uzorka dizajna, postoji mogućnost korišćenja okvira (eng.framework) biblioteka i komponenti o čemu će ovde biti reči.

Okvir (eng.**framework**) je sastavljen od skupa klasa za implementaciju koji se može koristiti kako osnova implementacije aplikacije. Na primer, postoje okviri za CORBA i DCOM na kojima može da se bazira implementacija dela arhitekture sistema. Mnogi okviri omogućavaju kreiranje pod klasa koje nasleđuju klase iz okvira. Naravno, prilikom nasleđivanja klasa u okvirima, kreira se zavisnost (povećava se povezanost nasleđivanja od pod klasa do super klase). Zbog toga, ako se koristi okvir i proizvođač okvira napravi neke izmene u okviru, moramo u najmanju ruku da ponovo kompiliramo sistem da bi ga podigli na noviju verziju okvira.

Biblioteka klasa je slična okviru u tome što obično ima skup klasa za implementaciju koje su dizajnirano tako da se mogu više puta iskoristiti. Međutim, okviri imaju tendenciju da budu specifični za određeni domen i mogu biti sastavljeni od klasa iz biblioteke. Tipična biblioteka klasa se može kupiti kako bi podržala numeričke ili statističke obrade, upravljanje fajlovima (data management layer) ili razvoj korisničkog interfejsa (human-computer interaction layer). U nekim slučajevima, mogu se kreirati instance klase koje se nalaze u biblioteci klasa, dok u drugim slučajevima klase iz biblioteke klasa se mogu proširiti kreiranjem pod klasa.

Kao i kod okvira, ako se za rejuzibilnost klasa u biblioteci klasa koristi nasleđivanje, može se naići na niz problema koji se odnose na povezanost nasleđivanja. Ako direktno instanciramo klase u biblioteci klasa, kreiraćemo zavisnost između naših objekata i objekata iz biblioteke, što povećava povezanost interakcije između objekata iz biblioteke i naših objekata.

Komponenta sadrži deo softvera koji može biti ubačen (eng. **plugged in**) u sistem kako bi obezbedio specifičan skup funkcionalnih zahteva. Danas postoji mnogo raspoloživih komponenti koje se mogu kupiti a koje su implementirane korišćenjem ActiveX ili JavaBean tehnologija. Komponenta ima dobro definisan API (eng. application program interface), koji

predstavlja skup metoda za pristup objektima sadržanim u komponentama. Interni rad komponenti je sakriven iza API-a. Komponente se mogu implementirati korišćenjem biblioteka klasa ili okvira. Međutim, komponente se mogu koristiti za implementaciju okvira. Kada se promeni API, potrebno je upgrad-ovati se na novu verziju komponente, što normalno zahteva samo ponovno linkovanje komponente u aplikaciju. U takvim slučajevima se obično ne zahteva ponovno kompiliranje.

KOJI PRISTUP KORISTITI?

Zavisi od toga šta želimo da kreiramo

Generalno, okviri se najviše koriste u razvoju objekata u sistemskoj arhitekturi, interakciji čovek-računar i sloju za upravljanje podacima; komponente se prvenstveno koriste da bi se pojednostavio razvoj objekata u domenu problema i sloju za interakciju čovek-računar; biblioteke klasa se koriste za razvoj okvira i komponenti koji treba da podrže osnovni sloj. Bez obzira koji od ovih pristupa se koristi, treba imati na umu da reupotrebljivost ima mnogo potencijalnih koristi ali i da može da stvori mnoge probleme. Na primer, softver je prethodno bio validiran i verifikovan, što može smanjiti količinu testiranja sistema. Međutim, ako se softver na kojem smo bazirali naš sistem promeni, tada ćemo verovatno i mi morati da promenimo sistem. Takođe, ako je softver vlasništvo treće firme, stvara se zavisnost od te firme.

Do današnjih dana, OO razvoj nije postigao nivo reupotrebljivosti koji se očekivao. Ponovno korišćenje nije lak zadatak. Na primer, da bi se postigla reupotrebljivost softverskih klasa, dizajner mora da poznaje postojeće klase, i da bude u stanju da s jedne strane prepozna da se njen interfejs podudara sa interfejsom klase koja mu treba, a s druge strane da se metode tih klasa podudaraju sa onim što mu treba. Da bi se utvrdilo da li neka od raspoloživih klasa zadovoljava zahteve, zahtevana klasa mora prethodno biti dizajnirana. Tako se ekonomičnost na osnovu reupotrebljivosti može javiti za vreme konstrukcije softvera i zahteva promenu kulture upravljanja projektom koji podržava reupotrebljivost; to znači da menadžer projekta mora biti u stanju da prepozna uštede koje se javljaju zbog nepisanja i testiranja linija koda.

PRIMER PONOVNE UPOTREBE KOMPONENTA

Kao primer komponete može poslužiti procedura prijavljivanja (login) koja se koristi na vebu, kao i sistem kojim se vrši štampanje u softveru.

Komponenta je deo softverskog programskog koda koji izvršava nezavisni zadatak u sistemu. To može biti mali modul ili podsistem.

Primer: Procedura prijavljivanja (login) koja se koristi na vebu se može smatrati komponentama, kao i sistem kojim se vrši štampanje u softveru.

Komponente imaju visoku funkcionalnu koheziju (functionality) i niži stepen povezivanja (coupling) tj. funkcionišu samostalno i mogu izvršavati zadatke bez zavisnosti od drugih modula.

U modularnom programiranju, moduli su kodirani da izvršavaju određene zadatke koji se mogu koristiti u brojnim drugim softverskim programima.

Postoji potpuno nova vertikala koja se zasniva na ponovnoj upotrebi softverskih komponenti

i poznata je kao softversko inženjerstvo zasnovano na komponentama ([Component Based Software Engineering CBSE](#)).

Ponovna upotreba se može izvršiti na različitim nivoima:

- **Nivo aplikacije** - tamo gde se cela aplikacija koristi kao podsistem novog softvera.
- **Nivo komponente** - tamo gde se koristi podsistem aplikacije.
- **Nivo modula** - tamo gde se reupotrebljavaju funkcionalni moduli.

Prilikom procesa ponovne upotrebe mogu se usvojiti dve vrste metoda:

- zadržavanjem istih zahteva i podešavanjem komponenata ili
- zadržavanjem istih komponenata i modifikovanjem zahteva.

U tom procesu se treba pridržavati sledećih koraka:

- Specifikacija zahteva - Specificiraju se funkcionalni i nefunkcionalni zahtevi, kojima softverski proizvod mora odgovarati, uz pomoć postojećeg sistema, korisničkog unosa ili i jednog i drugog.
- Dizajn - Ovo je takođe standardni korak SDLC procesa, gde su zahtevi definisani u smislu softverskog jezika. Kreira se osnovna arhitektura sistema u celini i njegovih podsistema.
- Specifikacija komponenti - Proučavajući dizajn softvera, dizajneri odvajaju iz čitavog sistema manje komponente ili podsisteme. Kompletni dizajn softvera pretvara se u kolekciju ogromnog skupa komponenata koje rade zajedno.
- Pretraživanje odgovarajuće komponente - Dizajneri pozivaju spremište softverskih komponenata kako bi pretražili i našli odgovarajuću komponentu na osnovu funkcionalnosti i predviđenih softverskih zahteva.
- Uključivanje komponenata - Sve odgovarajuće komponente se pakuju zajedno da bi se oblikovale kao celovit softver.

▼ Poglavlje 7

Individualne vežbe

PRIMENA VIŠEKRATNE UPOTREBLJIVOSTI NA PRIMERU ISUM-A

Na osnovu obrađenih pristupa na primenu višekratne upotrebljivosti, proanalizirajte njihu primenu na ISUM sistemu.

Odabratи deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka).

1. Za odabrani softver izdvojite jednu ili više funkcionalnosti sa viskom kohezijom i malom uparenošću (koje se mogu smatrati komponentama) i specificirajte način na koji se izdvojene komponente mogu višestruko upotrebiti. **Vreme izrade vežbe 45 minuta**
2. Za izabrani deo ISUM-a, u cilju povećanja rejuzibilnosti sistema proanalizirajte moguću primenu **Vreme izrade vežbe 45 minuta**
 - biblioteka klasa
 - uzoraka dizajna
 - okvira (framework-ova)

✓ Poglavlje 8

Domaći zadatak

ČETRNAESTI DOMAĆI ZADATAK

Nakon četrnaeste lekcije treba uraditi četrnaesti domaći zadatak

Na kod odabrane aplikaciju koju ste radili na nekom od predmeta koje ste prethodno slušali i položili primeniti neki od pristupa za postizanje višekratne upotrebljivosti:

- biblioteke klase
- uzorke dizajna
- rejuzibilnost klasa (npr. primenom nasleđivanja)

Domaći zadaci treba da budu realizovani u razvojnom okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ14-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ14.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templejt) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

✓ Tehnike održavanja softvera

ŠTA SMO NAUČILI U OVOJ LEKCIJI?

Kratak rezime

U ovoj lekciji se govori o osobinama višestruke primenljivosti, mogućim vrstama i pristupima koji se mogu koristiti u višestrukoj primenljivosti.

Na kraju lekcije se govori o uticaj kvaliteta softvera na proces održavanja analizom nekih od tehnika kao što su: upravljanje konfiguracijom, upravljanje izmenama, analiza uticaja, horizontalna i vertikalna pratljivost.



SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Studije slučajeva

Lekcija 15

PRIRUČNIK ZA STUDENTE

SE321 - OBEZBEDJENJE KVALITETA I TESTIRANJE SOFTVERA

Lekcija 15

STUDIJE SLUČAJEVA

- ✓ Studije slučajeva
- ✓ Poglavlje 1: Studija slučaja: OptimalSQM
- ✓ Poglavlje 2: Studija slučaja: Estimacija održavanja softvera
- ✓ Poglavlje 3: Studija slučaja: Tehnike estimacije održavanja
- ✓ Poglavlje 4: Studija slučaja: Održavanja softvera za ATM
- ✓ Poglavlje 5: Održavanja softvera po SWEBOK 2004
- ✓ Poglavlje 6: Grupna vežba
- ✓ Poglavlje 7: Individualna vežba 15
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

Šta ćemo naučiti u ovoj lekciji?

Ova lekcija se bavi se fazom održavanja softvera tako što kroz različite studije slučajeva opisuje proces, organizaciju, komponente i alate za testiranje i održavanje softvera. U pitanju je arhitektura OptimalSQM koja je primer SOA koja sadrži komponente-servise (OQT MNGR, OQT BOX, OQT MAINT, OQT OPST, OQT SIM) i dostupan je kao sveobuhvatni paket rešenja za upravljanje testiranjem i simulacijom mogućih scenarija procesa testiranja konkretne kompanije i konkretnog projekta.

Zatim je opisana studija slučaja Estimacija održavanja softvera za nastavu Courseware. Courseware je program za učenje - podržan obrazovni sadržaj ponuđen na pedagoški ispravan način studentima. Standardi za Courseware nisu široko prihvaćeni. Courseware proizveden u jednom alatu obično nije kompatibilan sa courseware-om proizvedenom u nekom drugom alatu. Pokušaji da se analizira efikasnost ili razvoj courseware-a nisu bili uspešni zbog nedostatka standardizacije.

Opisane su metrike za evaluaciju napora uloženog u razvoj programa, koje moraju podržati napor svakog podzadatka faze razvoja, pa i faze održavanja. Vođa projekta sjednjuje ove estimacije podzadataka da bi izračunao ukupni napor razvoja i održavanje Courseware softvera.

▼ Poglavlje 1

Studija slučaja: OptimalSQM

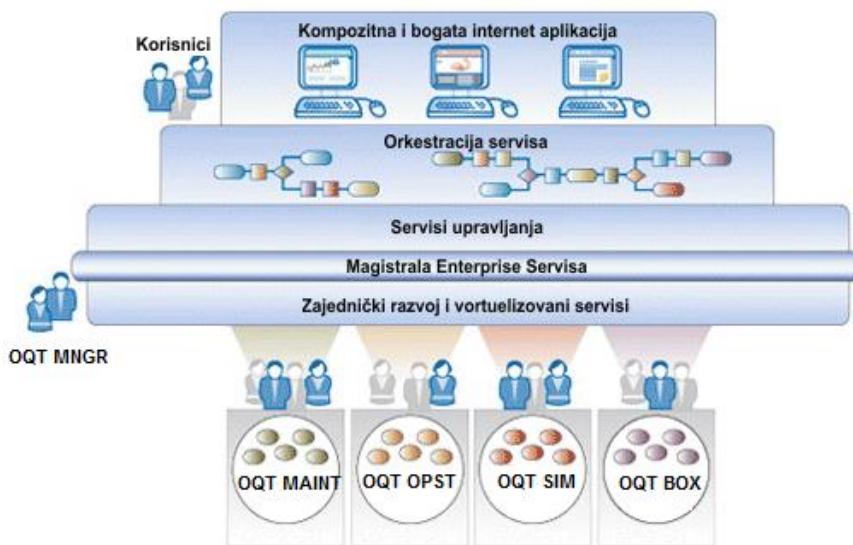
STUDIJE SLUČAJA: ARHITEKTURA OPTIMALSQM

Arhitektura OptimalSQM - SOA koja sadrži komponente-servise i dostupan je kao sveobuhvatni paket rešenja za upravljanje testiranjem i simulacijom mogućih scenarija procesa testiranja

Šta je OptimalSQM? OptimalSQM je razvojni projekat, čiji je krajnji cilj PISA ([Poslovne Inteligentne Simulacione Arhitekture](#)). Kroz realizaciju inovativne ideje OptimalSQM, kompanijama u oblasti IC tehnologija (u bankarskom, telekomunikacionom, obrazovnom sektoru), a i šire bi bio ponuđen kao softverski proizvod – Softversko okruženje PISA sa integrisanim procesima, tehnikama, softverskim aplikacijama ([EVOP Estimator Engine](#), [IOP Test Engine](#) i [IOP Maintenance Engine](#) i dr.), bazom znanja i podataka u oblasti testiranja i održavanja softvera i za neprekidnu/kontinualnu eksperimentalnu optimizaciju industrijskih procesa koji su intezivno softverski podržani.

Arhitektura OptimalSQM je primer SOA koja sadrži komponente-servise (OQT MNGR, OQT BOX, OQT MAINT, OQT OPST, OQT SIM) i dostupan je kao sveobuhvatni paket rešenja za upravljanje testiranjem i simulacijom mogućih scenarija procesa testiranja konkretne kompanije i konkretnog projekta (slika 1). MNGR je u srcu sistema, pruža integrisano i koherentno upravljanje multidisciplinarnim aspektima ispitivanja softvera i omogućava testiranje pravila u upravljanju procesom testiranja određenog tipa softvera na optimalan način, konkretne kompanije i konkretnog projekta put simulacije mogućih scenarija realizacije procesa testiranja u okviru planiranog budžeta, trajanja i atributa kvaliteta softvera koji se razvija. OptimalSQM predstavlja skup najboljih modela i tehnika iz prakse, integrisanih u optimizovan i kvantitativno rukovođen proces razvoja,

testiranja i održavanja softvera. To su pre svega ekspertske alati koji se integrišu na zahtev korisnika. PISA rešenje je servisno orijentisane arhitekture zasnovane na: 1) servisima, 2) aplikacijama koje se na zahtev korisnika uspostavljaju u cilju pružanja servisa, 3) servisno orijentisanoj infrastrukturi koja obezbeđuje standardne veze (komunikaciju) između potrebnih aplikacija i zahtevanih servisa od strane PISA korisnika koja je prikazana na Slici 1.



Slika 1.1 Konceptualni nivo PISA arhitekture [Izvor: NM SE321 -2020/2021.]

KONCEPTUALNI NIVO PISA ARHITEKTURE

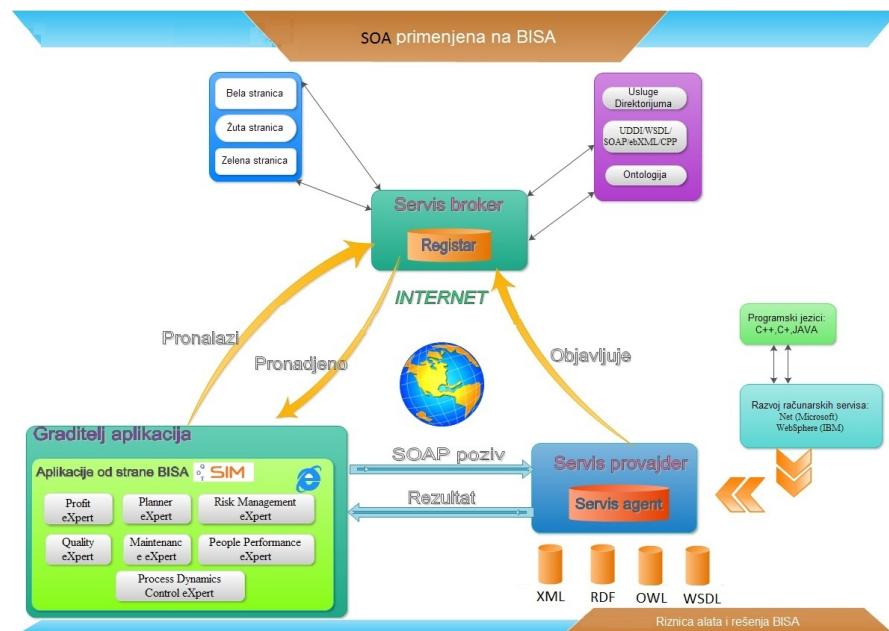
PISA je unificirano, koherentno, izbalansirano i dograđeno rešenje OptimalSQM koje treba da obezbedi potpun spektar servisa koji zadovoljavaju najviše nivoe (4 i 5 nivo zrelosti)

PISA je unificirano, koherentno, izbalansirano i dograđeno rešenje OptimalSQM koje treba da obezbedi potpun spektar servisa koji zadovoljavaju najviše nivoe (4 i 5 nivo zrelosti PRS-PTS po SEI CMM i TMM metodologiji) za razvoj kvalitetnog softvera na efikasan i efektivan tj. optimalan način.

PISA rešenje je zasnovana na servisno orijentisanoj arhitekturi (**Service Oriented Architecture – SOA**) sa integrisanim ekspertskskim alatima (Profit eXpert, Planner eXpert, Risk Management eXpert, Quality eXpert, Maintenance eXpert, People Performance eXpert and Process Dynamics Control eXpert) za sve modele PRSPTS na bazi izraženog ekonomskog modela kvaliteta softvera, oceni isplativost predloženih aktivnosti obezbeđenja i kontrole kvaliteta PRS-PTS na osnovu ekonomskih parametara (ROI, BCR, CAPEX, OPEX i dr.) kao što je na slici 2 prikazano.

Planer eXpert treba na osnovu istraženih modela estimacije i predikcije veličine softvera, složenosti, trajanja razvoja, trajanja testiranja, broja potencijalnih grešaka u softveru, trajanja i cene njihove popravke tokom PRS-PTS, pruži neophodne podatke za simulaciju različitih scenarija PRSPTS iz kojih se bira optimalni scenario realizacije projekta.

Risk Management eXpert treba da u saradnji sa Profit eXpert sofverskim alatom pruži servis menadžerima dizajna i testiranja softvera u: identifikaciji, proceni efekata, plana aktivnosti smanjenja i kontrole rizika na prihvatljivom nivou, datog softverskog projekta.



Slika 1.2 Konceptualni nivo PISA arhitekture - Dijagram dugog nivoa [Izvor: NM SE321 -2020/2021.]

ULOГА KOMPONENTI QUALITY EXPERT, MAINTENANCE EXPERT I PROCESS DYNAMICS CONTROL EXPERT

Quality eXpert treba da integriše specijalizovane ekspertske alate, Maintenance eXpert treba da obezbedi servis menadžerima dizajna i testiranja softvera

Quality eXpert treba da integriše specijalizovane ekspertske alate (Quality Metrics eXpert, Test Effort Estimation eXpert, Reliability eXpert, Product release eXpert) koji obezbeđuju servis menadžerima dizajna i testiranja softvera u: izradi metrike integrisanog procesa merenja kvaliteta softvera, automatizaciji procesa planiranja zasnovanog na modelima estimacije veličine softvera, cene, broju projektanata, trajanja razvoja i testiranja, proceni i predikciji pouzdanosti softverskog rešenja tokom simulacije različitih scenarija dizajna i u toku realizacije PRS-PTS, koji treba da dovedu do doношења odluke o završetku PRS-PTS i predaje softveskog proizvoda (IS) na upotrebu.

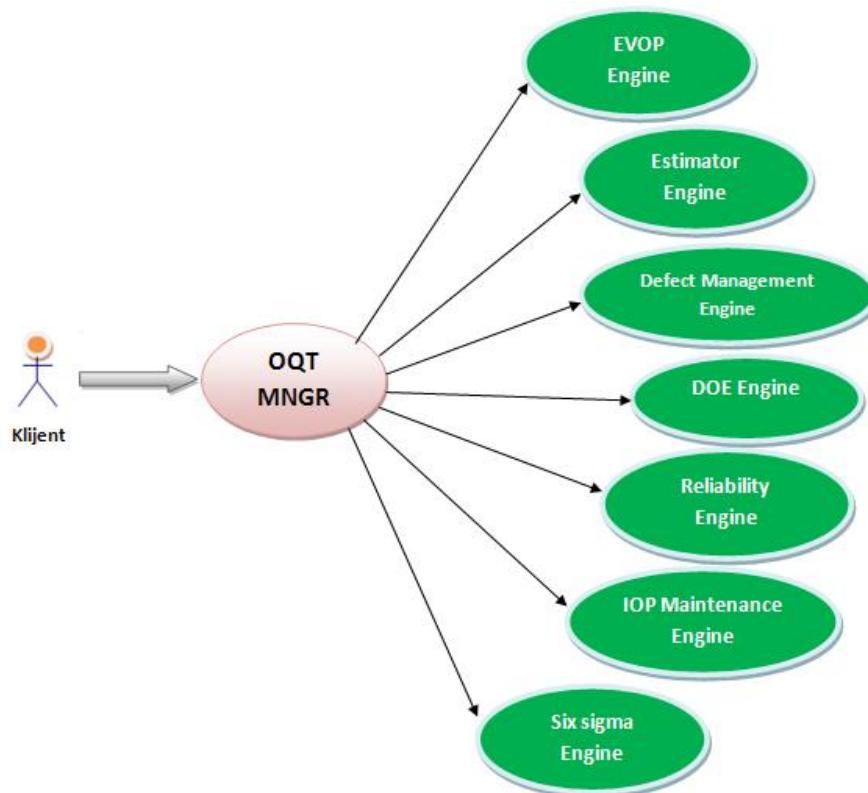
Maintenance eXpert treba da obezbedi servis menadžerima dizajna i testiranja softvera u: izradi plana i proceni troškova korektivnog, adaptivnog, perfektivnog i preventivnog održavanja softvera. Kao što smo već istakli, razvoj kvalitetnog softvera je jako složen i nepouzdan posao, ali je upravljanje složenim, dinamičkim procesom razvoja i testiranja (sa preko 100 promenljivih) još teže bez adekvatnog softverskog alata Process Dynamics Control eXpert.

Process Dynamics Control eXpert treba da identifikuje observabilne i kontrolabilne promenjive konkretnog softverskog projekta, da uspostavi kriterijume stabilnosti i optimalnosti u svakoj fazi PRS-PTS i za ceo proces. Da bi ovako realizovano softversko okruženje za optimalan razvoj kvalitetnog softvera zaista obezbedilo uspeh na konkretnom softverskom projektu tj.

dal očekivane rezultate, neophodno je stalno ulaganje u: ocenjivanje i praćenje performansi projektnog tima, podizanje stručnog kapaciteta ljudi koji realizuju projekat. Za to je odgovoran softverski alat People Performance eXpert.

ULOGA OQT MAINT KOMPONENTA

Ova komponenta razmišlja o svim rezultatima testiranja radi poboljšanja kontrole kvaliteta i upravljanja svim aspektima operacija testiranja



Slika 1.3 Dijagram slučajeva korišćenja OQT MAINT funkcija [Izvor: NM SE321 -2020/2021.]

OQT MAINT komponenta (Sl. 3) razmišlja o svim rezultatima testiranja radi poboljšanja kontrole kvaliteta i upravljanja svim aspektima operacija testiranja u korektivnom, adaptivnom i perfektivnom održavanju softvera kako u toku razvoja tako i nakon isporuke softverskog proizvoda na upotrebu

MAINT komponenta vrši unakrsne procene kvaliteta svih flota testiranja, za sve procene efikasnosti testiranja u otkrivanju i otklanjanju defekata (povećanje prinosa otkrivenih grešaka), nudeći ekstremni integritet podataka.

Osim toga, MAINT komponenta poboljšava pouzdanost softvera kroz SRE (Software Reliability Engineering) metodologiju metrike pouzdanosti softverskog proizvoda u predviđanju i proceni kritičnih faktora kao što su: stopa grešaka po fazama razvoja softvera, konačna stopa grešaka

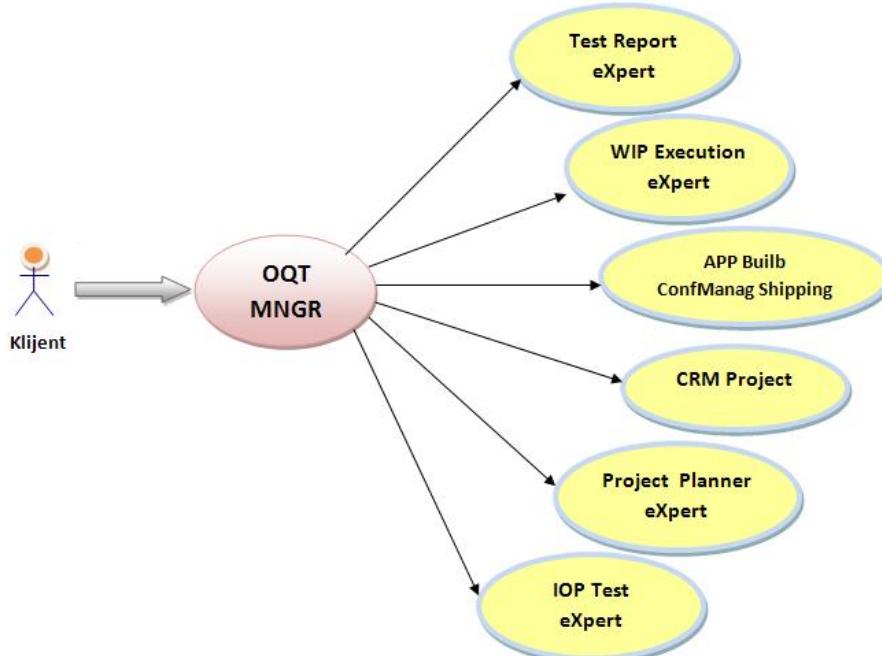
nakon 6 meseci upotrebe softvera, gustine grešaka na KSLOC ili FP metrići veličine softvera, profil greška itd.

Na osnovu ovih podataka MAINT komponenta obezbeđuje kompletну tehničku podršku nakon puštanja softverskih proizvoda u promet, odnosno program za aktivnosti održavanja tj.za korektivno, adaptivno, perfektivno i preventivno održavanje na optimizovan način.

ULOGA OQT OPST KOMPONENTE

OQT OPST na osnovu sopstvene metrike ili usrednjene metrike iz baze merenih karakteristika tipa softverskog proizvoda koji se razvija, performansi razvojnog tima, TMM nivoa zrelosti procesa

OQT OPST komponenta (Operational Software Testing), koja je prikazana na Sl. 4, treba timu za planiranje i sprovođenje testiranja konkretnog razvijanog softvera, konkretne kompanije (Project Specific Software Testing) da omogući na osnovu stvarnih performansi konkretne kompanije i konkretnog projekta te kompanije i pronađenog optimalnog scenarija za dati projekat na bazi REZULTATA izvršenih simulacija (OQT SIM komponente) iz skupa mogućih scenarija testiranja,a pre primene u realizaciji datog konkretnog softverskog projekta, odredi karakteristike (parametre) integralnog i optimalnog PTS (IOPTS). Dakle, na osnovu sopstvene metrike ili usrednjene metrike iz baze merenih karakteristika tipa softverskog proizvoda koji se razvija, performansi razvojnog tima, zrelosti (TMM nivoa) procesa testiranja u dotoj kompaniji i sl., odredi aktivnosti i objekte testiranja u tačkama provere artefakata datog PTS (SDLC), odredi adekvatne tehnike detekcije grešaka koje obezbeđuju zahtevani kvalitet tokom razvoja softverskog proizvoda u okvirima projektnih ograničenja tj. sve parametre IOPTS.



Slika 1.4 Dijagram slučajeva korišćenja OQT OPST funkcija [Izvor: NM SE321 -2020/2021.]

DETALJNI POGLED NA PISA ARHITEKTURU

Na slici je dat slučaj upotrebe za menadžera OptimalSQM, njegov odnos prema zahtevima klijenata i proces upravljanja samim projektom

Na slici 5 dat je slučaj upotrebe za menadžera OptimalSQM. Njegov odnos prema zahtevima klijenata i proces upravljanja samim projektom koji sadrži:

- **Upravljanje kvalitetom**
- **Upravljanje rizikom**
- **Upravljanje troškovima**
- **Estimacija**
- **Metrika**



Slika 1.5 Slučaj upotrebe za menadžera OptimalSQM [Izvor: NM SE321 -2020/2021.]

FUNKCIONALNA DEKOMPOZICIJA

Slučaj upotrebe za menadžera procesa estimacije OptimalSQM-a, sadrži procese upravljanja estimacijom

Na slici 6. dat je slučaj upotrebe za menadžera procesa estimacije OptimalSQM-a, gde se predstavlja proces upravljanja estimacijom koji se sastoji iz sledećih delova:

- **Estimacija troškova proizvodnje softvera**
- **Estimacija troškova održavanja**
- **Estimacija softverskih zahteva**
- **Estimacija alata**



Slika 1.6 Slučaj upotrebe za menadžera procesa estimacije OptimalSQM [Izvor: NM SE321 -2020/2021.]

Dijagrami toka u ovom delu treba da nam pokažu tačno šta naš sistem kao i sami naš modul, treba da radi. On ne pokazuje na koji način treba da radi naša komponenta. Ovi dijagrami su takođe od velike važnosti za kasnije izračunavanje funkcionalnih tačaka i estimacije. Opšti sekvensijalni dijagram OptimalSQM i sekvensijalni dijagram toka testiranja OptimalSQM je prikazan na sledećoj slici.

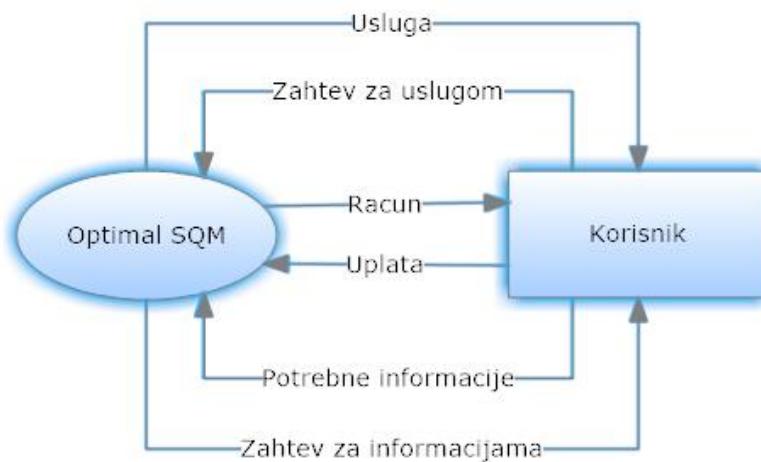
Slika 1.7 Opšti sekvensijalni dijagram OptimalSQM [Izvor: NM SE321 -2020/2021.]

Slika 1.8 Sekvensijalni dijagram toka testiranja OptimalSQM [Izvor: NM SE321 -2020/2021.]

DIJAGRAM PRVOG I DRUGOG NIVOA

Na dijagramu prvog nivoa je prikazan kompletan sistem Optimal SQM i njegov odnos prema korisniku dok se na dijagramu drugog nivoa spuštamo samo na našu komponentu

Na dijagramu prvog nivoa (slika 9) je prikazan naš kompletan sistem Optimal SQM i njegov odnos prema korisniku i komunikacija. Dakle, u osnovi, našem sistemu pristupaju eksterni klijenti. Oni na našem web sajtu zahtevaju odgovarajuću uslugu. Zatim, naš sistem na osnovu zahteva, traži potrebne informacije od njih. Na kraju, pre same usluge izdaje račun za uplatu i nakon uplate omogućava uslugu. Naravno, neke usluge će biti i besplatne ili će moći neke opcije da se probaju direktno na sajtu putem showroom-a.



Slika 1.9 Dijagram prvog nivoa OptimalSQM rešenja [Izvor: NM SE321 -2020/2021.]

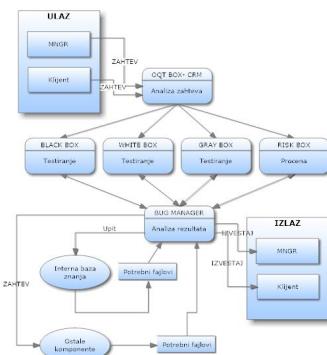
Na dijagramu drugog nivoa (Slika 10) spuštamo se samo na našu komponentu i dajemo opšti prikaz kretanja podataka unutar nje. Dakle na ulazu imamo dva interfejsa (MNGR modul i klijenta). Naša zamisao je da će svi ostali moduli unutar Optimal SQM-a komunicirati putem modula MNGR. Klijenti će na neki način moći da pristupe našem modulu, ali takođe i za te korisnike ćemo dobijati dodatne informacije iz MNGR-a koji u sebi sadrži CRM komponentu. Nakon što dobije naš modul za komunikaciju sa ovim interfejsima određen zahtev, on ga analizira i pakuje u formatiran zahtev koji se dalje šalje ka ostalim komponentama BOX-a. Dakle, na osnovu ovog zahteva pokreće se određene tehnike testiranja i moduli koji su za to zaduženi i vrši se testiranje i zatim se šalju korisniku koji ih je tražio.

Slika 1.10 Dijagram drugog nivoa OptimalSQM rešenja [Izvor: NM SE321 -2020/2021.]

DIJAGRAM TREĆEG NIVOA

Na ovom dijagramu prikazujemo i samu komunikaciju i tok informacija između samih komponenti unutar OQT BOX-a.

Na dijagramu trećeg nivoa (Slika 11) prikazujemo i samu komunikaciju i tok informacija između samih komponenti unutar OQT BOX-a. Dakle imamo kao i u prošloj dekompoziciji na početku slanje zahteva ili direktno od korisnika ili od MNGR-a ili od nekog drugog modula, ali putem MNGR-a. Naš CRM modul analizira zahtev. Formira na osnovu njega odgovaraće formatirane zahteve i pokreće određene tehnike unutar testnih modula. U slučaju da su potrebni dodatni fajlovi od nekih drugih komponenti Bug manager ih zahteva i dobija, i prosleđuje ih natrag testnim modulima kako bi izvršili testiranje. Nakon uspešnog testiranja, Bug manager analizira rezultate i generiše odgovarajući izveštaj na izlazu prema MNGR-u ili klijentu.



Slika 1.11 Dijagram trećeg nivoa OptimalSQM rešenja [Izvor: NM SE321 -2020/2021.]

OQT SIM komponenta simulira procese (scenarije) testiranje na osnovu uspostavljenih pravila i algoritama koji su bazirani na empirijskim rezultatima testiranja, kvalifikacijom potencijalne koristi ovih pravila pre primene. Zahvaljujući nadgledanju planiranja, OQT-SIM takođe proverava poboljšanje kvaliteta i efikasnosti postojećih pravila postavljenih tokom vremena, što omogućava poređenje stvarne koristi baziranih na akumulaciji informacija u realnom svetu procesa testiranja za razne vrste softverskih proizvoda, nivoa CMM i TMM zrelosti konkretne kompanije kojoj pružamo servis. OQT-SIM nudi simulaciju šablonu koji sadrže algoritme iz različitih porodica softverskih proizvoda, nivoa zrelosti softverskih kompanija, kao što su smanjenje vremena testiranja, napredna statistička kontrola procesa, kvalitet i pouzdanost, smanjenje naknadne dorade usled napravljenih grešaka u svim fazama razvoja softvera. Svaka familija stimulacije je bogata sa pravilima koji su posebna meta poslovnih potreba kao što je na slici 12 prikazano.

Slika 1.12 Prikaz mogućih scenarija na osnovu unetih parametara [Izvor: NM SE321 -2020/2021.]

▼ Poglavlje 2

Studija slučaja: Estimacija održavanja softvera

OPIS SOFTVERA ZA UČENJE - COURSEWARE

Pošto su istorijski podaci prikupljeni tokom razvoja programa, lekcija je definisana kao knjiga koja se sastoji od stranica prezentovanog materijala.

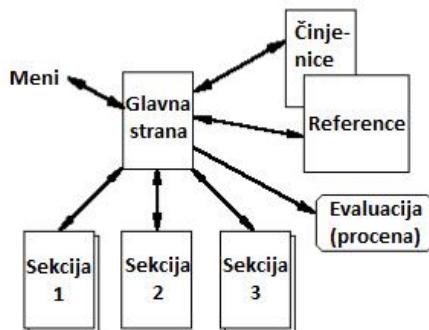
Courseware je program za učenje - podržan obrazovni sadržaj ponuđen na pedagoški ispravan način studentima. Standardi za Courseware nisu široko prihvaćeni. Courseware proizveden u jednom alatu obično nije kompatibilan sa courseware-om proizvedenom u nekom drugom alatu. Pokušaji da se analizira efikasnost ili razvoj courseware-a nisu bili uspešni zbog nedostatka standardizacije. Da bi rešila ove probleme, vazduhoplovna industrija, koja u velikoj meri ulaže u Courseware, postavila je neke standarde.

U ovoj lekciji, Courseware se definiše kao sistem pretstavljanja više grupa lekcija pomoću nekoliko podsistema. Sistem učeniku omogućava da uđe i izđe iz njega, kao i da sačuva rezultate svog testa. Lekcija je definisana autoringom softvera koji se koristi za izradu Courseware-a. Pošto su istorijski podaci prikupljeni tokom razvoja programa, lekcija je definisana kao knjiga koja se sastoji od stranica prezentovanog materijala.

Standardna lekcija (prikazana je na slici 1) sastoji se iz sledećih komponenti:

- Glavna strana, koja sadrži uvod u lekciju i navigacione kontrole,
- **Strana činjenica, koja je zbir sadržaja lekcije,**
- Strana referenci, koja sadrži razloge zašto učenje mora biti u srodstvu sa sadržajem lekcije
- Nekoliko odeljaka sadržaja (svaki od njih verovatno ima nekoliko stranica), i
- **Set pitanja povezanih sa lekcijom, koji se koriste za procenu učenikovog razumevanja iste.**

Stranice su kombinacija teksta, slika, audio i video fajlova, i hipertekstova koji pretstavljaju informacije za učenika. Za ovaj primer, definisaćemo da lekcija ima 6 strana: glavnu, činjenice, reference, tri odeljka od po jedne strane i ocenivanje sadržaja lekcije.



Slika 2.1 Standardna lekcija [Izvor: NM SE321 -2020/2021.]

ORGANIZACIONO PLANIRANJE

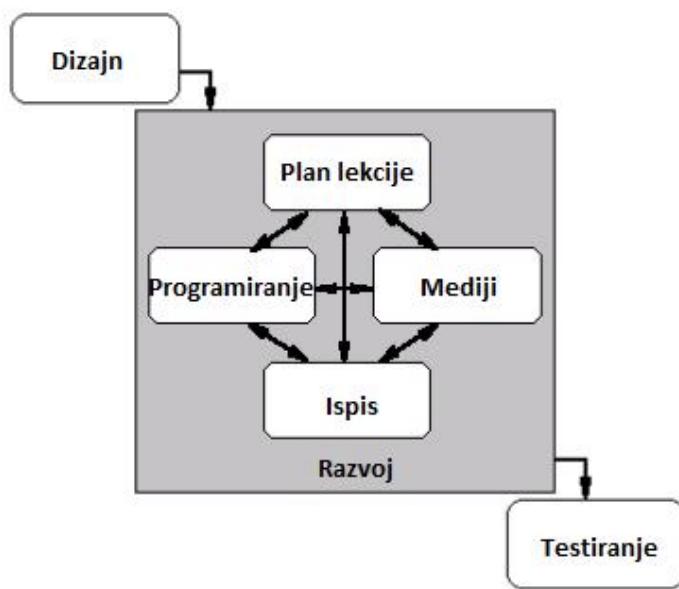
Tim koji radi na razvoju programa za učenje (Courseware-a) uključuje dodatne napore dizajnera uputstva, tehničkih urednika, specijalista za medije, programera i ulaznog osoblja.

Tim za razvoj informacionog sistema softvera, sastoji se od glavnog programera (inženjera sistema), tehničkog osoblja i rezervnog inženjera (**Pressman**). Tim koji radi na razvoju programa za učenje (Courseware-a) uključuje dodatne napore dizajnera uputstva, tehničkih urednika, specijalista za medije, programera i ulaznog osoblja.

Odgovornosti članova tima su:

- Dizajner instrukcija, koji obično vodi, planira, razvija plan lekcije i upravlja svim aktivnostima tima, a ponekad predstavlja i tehničkog urednika za drugog dizajnera;
- Tehnički urednik, koji potvrđuje jasnoću i koegzistenciju prezentovanih informacija, kao i njihovu gramatičku ispravnost;
- Specijalista za resurse, koji planira i upravlja produkcijom svih grafika, audija, animacija i videa;
- Programer, koji planira i prevodi specifikacije zahteva u sistem Courseware-a, stvara templove i održava dokumentaciju programa;
- Ulazno osoblje, koje kombinuje planove lekcija sa medijima, da bi kreirali rezultat, odnosno prezentaciju lekcije; i
- Osoblje za testiranje, koje vrši procenu preciznosti ulaznog sadržaja i medija u svakoj lekciji.

Metrike za evaluaciju napora uloženog u razvoj programa, moraju podržati napor svakog podzadatka faze razvoja. Vođa projekta sjedinjuje ove estimacije podzadataka da bi izračunao ukupni napor razvoja (Slika 2).



Slika 2.2 Razvojna faza izmenjenog modela životnog ciklusa [Izvor: NM SE321 -2020/2021.]

TEHNIKE ESTIMACIJE ODRŽAVANJA

Neke tehnike estimacije softvera zasnovane su ili na funkcionalnim tačkama ili na linijama kôda kao što smo u prethodnim lekcijama detaljno opisali.

Neke tehnike estimacije softvera zasnovane su ili na funkcionalnim tačkama ili na linijama kôda kao što smo u prethodnim lekcijama detaljno opisali. Jedan od proračuna u procesu estimacije funkcionalnih tačaka zasniva se na broju korisničkih ulaza. Za razliku od sistema baze podataka za pretraživanje, kompjuterski zasnovan kurs je stvoren da bude interaktivan, podstiče učenika da sam bira svoj put kroz kurs. Da li se onda svaki klik miša računa kao jedan ulaz (jedna od funkcionalnih tačka)? Pokušaj da se izračuna napor courseware-a ovom metodom nije uspeo. Da bi se izračunali naporovi koji su uloženi u razvoj Courseware-a potrebna je drugačija osnovna jedinica. Da bismo utvrdili estimaciju napora, na raspolaganju imamo sledeće opcije:

- estimacija kašnjenja, do kasno u projektu,
- korišćenje tehnike dekompozicije za generisanje estimacije projekta,
- razvoj empirijskog modela, ili
- sticanje automatizovanih alatki.

Parcelizacijom razvojne faze i korišćenjem istorijskih podataka kao osnovne vrednosti, metrika estimacije može da se razvije tako da je moguće predvideti napor utrošen u razvoju Courseware-a. Upotreba autoring sistema kao pomoć u razvoju Courseware-a postala je uobičajena praksa, a postoji niz autoring sistema koji su nam lako dostupni.

Stranica je jedinica za prezentaciju informacije u autoring sistemu, što je standard postavljen od strane kompanije koja nas snabdeva istorijskim podacima. Stranica se sastoji od teksta, hiperteksta i svih povezanih medijskih vrsta. Metoda predstavljene estimacije će odrediti

napor razvoja izračunavanjem napora koji je utrošen na stvaranju jedine stranice. Faza razvoja je podeljena na sledeće pod-zadatke:

- **Dizajn (De):** Stvaranje i tehničko uređivanje nastavnih planova. Ukupan napor za dizajn (De) jednak je zbiru stvaranja lekcija (Le) i tehničke obrade (Tee).
- **Programiranje (Pe):** Promene u šablonu koje su definisane u nastavnom planu, preliminarno testiranje i ispravke u svoj dokumentaciji.
- **Mediji (Me):** Razvoj grafike, audija, videa kao što je definisano u nastavnom planu. Ukupan napor za medijsku razvoj (Me) jednak je zbiru grafičkog napora (Ge), audio napora (Ae) i video napora (Ve).
- **Ispis (Ee):** Ubacivanje sadržaja lekcije u šablon i koordinisanu medijsku prezentaciju.
- **Testiranje (Te):** Obezbeđuje tačnost prezentacije lekcije kao individualne jedinice.

Ukupni napor je kompozitivna metrika koja se sastoji od zbiru napora utrošenog u razvoj svih pod-zadataka: $\text{Ukupni napor} = \text{De} + \text{Pe} + \text{Me} + \text{Ee} + \text{Te}$

UPOTREBA ISTORIJSKIH PODATAKA

Nedostatak istorijskih mernih podataka često dovodi do postavljanja proizvoljnog rasporeda za buduće projekte, i često je ključ neuspeha razvijenog softvera.

Nedostatak istorijskih mernih podataka često dovodi do postavljanja proizvoljnog rasporeda za buduće projekte, i često je ključ neuspeha razvijenog softvera. Uključivanjem istorijskih podataka kao polazne osnove u metrikama estimacije, prikazanih u ovom radu, dobija se solidna empirijska osnova.

Prikupljanje istorijskih podataka upošljava opšti sistem izveštavanja, odnosno, "vreme" nas je izveštavalo o razvoju medija, ali nije bilo podeljeno po tipu medija.

Razgovori sa medijskim osobljem i pregled njihovih ličnih vremenskih dnevnika bili su neophodni da bi se odredio napor za svaku vrstu medija (audio, grafiku i video). Razgovori sa osobljem koje je zaduženo za dizajn su takođe neophodni, da bi se utvrdilo prosečno vreme koje je potrebno tehničkom uredniku.

✓ Poglavlje 3

Studija slučaja: Tehnike estimacije održavanja

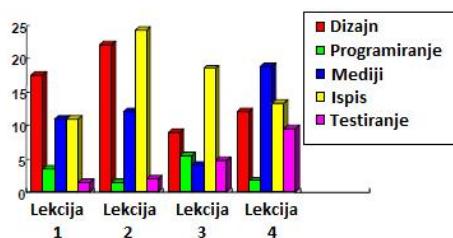
KOLIČINA NAPORA - UPOTREBA ISTORIJSKIH PODATAKA

Količina napora prijavljena za podzadatke razvoja i održavanja varira od časa do časa. Razlozi za ovolika variranja u rasponu su količina iskustva osoblja, broj i vrsta razvijenih medija

Količina napora prijavljena za podzadatke razvoja i održavanja varira od časa do časa, kako je predstavljeno na slici 1. Razlozi za ovolika variranja u rasponu su količina iskustva osoblja, broj i vrsta razvijenih medija, teškoće u koordinaciji tipova medija, kao i poteškoće stvara ili dobijanja medijskih resursa.

Iako su istorijski podaci obezbeđeni iz više lekcija, neki od podataka nisu uračunati iz sledećih razloga:

- netačno izveštavanje zbog upotrebe neodgovarajućih kodova izveštavanja,
- primena različitih sistema izveštavanja u sred razvoja lekcije, ili
- lekcija je sastavljena od grupe podlekcijsa, a samim tim, nije u skladu sa definicijom standardne lekcije



Slika 3.1 Prijavljeni razvojni napori (jedinica: utrošeni sati) [Izvor: NM SE321 -2020/2021.]

Izračunavanje proseka prijavljenog napora utrošenog na razvoj N strana, (osnova za izračunavanje napora za svaki podzadatak je predstavljena u tabeli na slici 2).

Primenom formula iz prethodne sekcije ove lekcije koji su sistematizovani u tabeli na slici 3, vrši se izračunavanje količine napora.

Pod-zadaci	Prosek utrošenih sati
Dizajn Tehničko uređivanje	2.25 / strana + 4.0 * # recenzije
Audio resurs	0.25 / audio fajl
Grafički resurs	1.0 / grafika
Vidio resurs	4.0 za 20 sekundi videa
Ispis	1.25 / strana
Programiranje	0.5 / strana
Testiranje	0.25 / strana

Slika 3.2 Prosečno vreme razvoja strane, zasnovano na istorijskim podacima [Izvor: NM SE321 -2020/2021.]

Slika 3.3 Izračunavanje proseka prijavljenog napora [Izvor: NM SE321 -2020/2021.]

METRIKA ESTIMACIJE POJEDINIХ AKTIVNOST

Korišćenjem ovih proseka kao osnovne jedinice, razvijena je metrika estimacije za svaku aktivnost, koja se odnosi na razvoj standardne lekcije.

Korišćenjem ovih proseka kao osnovne jedinice, razvijena je metrika estimacije za svaku aktivnost, koja se odnosi na razvoj standardne lekcije.

$$\text{Ukupni napor} = D_e + A_e + G_e + V_e + E_e + P_e + T_e$$

Za lekciju koja ima N=6 strana, kod kojih se na svakoj strani nalazi po jedan audio fajl, jedan grafički fajl i dve interne recenzije, estimacija napora izgleda ovako:

$$D_e + A_e + G_e + V_e + E_e + P_e + T_e$$

$$D_e = [2.25 * 6] + [4.0 * 2] = 13.5 + 8.0 = 21.5$$

$$A_e = 0.25 * 6 = 1.5 \quad G_e = 1.0 * 6 = 6.0 \quad V_e = 4.0 * 0 = 0$$

$$E_e = 1.25 * 6 = 7.5 \quad P_e = 0.5 * 6 = 3.0 \quad T_e = 0.25 * 6 = 1.5$$

$$\text{Ukupni napor} = 41.0 \text{ neprilagođenih utrošenih sati.}$$

Ova procena ne uključuje nekoliko zadataka koji se obavljaju izvan pojedinačne lekcije, poput uključivanja u ove lekcije u sistem kursa, vreme instalacije, testiranje kursa u celini nakon što se završe svi dodaci lekcije, kao i niz drugih zadataka koji nisu direktno povezani sa razvojem jedne lekcije. Treba napomenuti da su ove metrike takođe pod uticajem faktora prilagođavanja troškova (objašnjeno u sledećem delu), što se odražava na estimaciju neprilagođenih utrošenih sati.

FAKTORI PRILAGOĐAVANJA TROŠKOVA

To su spoljni faktori koji utiču na napor utrošen za izradu finalne lekcije

Baš kao i u drugim softverskim sistemima, i ovde postoje spoljni faktori koji utiču na napor utrošen za izradu finalne lekcije, kao što je:

- iskustvo svakog člana tima, ponaosob,
- sposobnost tehničkih stručnjaka, informacije, izvori i
- postojanje ili nedostatak šablonu.

Svaka aktivnost, opisana u sledećoj tabeli na slici 4, koja je povezana sa fazom razvoja ima sopstvena ograničenja (faktore prilagođavanja troškova) koja utiču na estimaciju napora razvoja.

Dizajn [Cf _d]	= DE * IA * SME * IR
Programiranje [Cf _p]	= TA * PE * PC
Audio [Cf _a]	= TE * RA
Grafike [Cf _g]	= TE * RA
Video [Cf _v]	= TE * RA
Ispis [Cf _e]	= EE * EC
Testiranje [Cf _t]	= DL

Slika 3.4 Faktori koji utiču na napor utrošen za izradu finalne lekcije [Izvor: NM SE321 -2020/2021.]

Faktori prilagođavanja troškova koji utiču na specifične zadatke dati su u sledećoj tabeli na slici 5.

Slika 3.5 Faktori prilagođavanja troškova [Izvor: NM SE321 -2020/2021.]

PREGLED VREDNOSTI FAKTORA TROŠKOVA

Vrednosti dodeljene faktorima prilagođavanja troškova postavljene su u dve oblasti sa početnom vrednošću 1,0 (ne odražavajući nikakav efekat na obračun napora).

Vrednosti dodeljene ovim faktorima prilagođavanja troškova postavljene su u dve oblasti sa početnom vrednošću 1,0 (ne odražavajući nikakav efekat na obračun napora). Za prilagođavanje faktorima koji imaju veći uticaj na stvarne napore (Tabela na slici 6), opseg je -0.4 do 1.6 u koracima od 0.1. Za ostatak, opseg je -0.5 do 1.5 u koracima od 0.05. Na primer, iskusnom dizajneru će DE biti recimo 0.75, jer smo mogli da očekujemo da će zadatak dizajna biti završen za manje vremena nego što bi to bio slučaj da je u pitanju prosečni ili manje iskusni dizajner.

Vrednost faktora troškova	Faktor prilagođavanja troškova
$1.0 \pm (0.1 \text{ to } 0.6)$	Sposobnost resursa Kompleksnost ispisa Kompleksnost kursa Uticaj recenzija dizajna Prilagođavanje šablonu
$1.0 \pm (0.05 \text{ to } 0.5)$	Iskustvo u dizajnu Dostupnost informacije Dostupnost SME Iskustvo u programiranju Iskustvo sa alatom Iskustvo u video produkciji Iskustvo ispisa Detaljni nivo (testiranja)

Slika 3.6 Pregled vrednosti faktora troškova [Izvor: NM SE321 -2020/2021.]

Zadatak dizajna je podeljen u dva koraka: dizajn lekcije i tehničku obradu. Deo koji se odnosi na tehničko uređivanje zadatka dizajna ne utiče na faktore prilagođavanja troškova o kojima smo govorili.

Izračunavanje ovih vrednosti vrši se prema formulama iz prethodnih tabela su sistematizovani u tabeli na slici 7.

Pod-zadatak		Vrednost faktora troškova
Dizajn	$[Cf_d] = DE * IA * SME * IR = 1.1 * 1.0 * 1.0 * 1.2$	= 1.32
Programiranje	$[Cf_p] = TA * PE * PC = 0.9 * 0.95 * 1.0$	= 0.855
Audio	$[Cf_a] = TE * RA = 0.95 * 1.0$	= 0.95
Grafika	$[Cf_g] = TE * RA = 0.95 * 1.1$	= 1.045
Ispis	$[Cf_e] = EE * EC = 1.0 * 1.2$	= 1.2
Testiranje	$[Cf_t] = DL = 0.9$	= 0.9

Slika 3.7 Izračunavanje ovih vrednosti [Izvor: NM SE321 -2020/2021.]

REZIME METRIKE ESTIMACIJE ZA COURSEWARE RAZVOJ

Primenom faktora prilagođavanja troškova, prema usvojenim vrednostima za uticaj na stvarne napore tj. na svaki napor pod-zadataka, dobijaju se konačne vrednosti napora u ovom primeru

Primenom faktora prilagođavanja troškova, prema usvojenim vrednostima za uticaj na stvarne napore tj. na svaki napor pod-zadataka, dobijaju se konačne vrednosti napora u ovom primeru obračuna dati su u na slici 8. Procena ukupno potrošenih sati uračunavajući i korekcione faktore: 46.42

Rezime metrike estimacije za Courseware razvoj (u utrošenim satima) prikazane su u tabeli na slici 9, gde je N= broj strana web aplikacije.

Pod-zadatak		N sati*	Faktor troškova		I sati*
Dizajn	+ (2.25 * 6) =	13.5	* 1.32≈	17.82	+ 25.82
Uređivanje				8.0	
Programiranje	(0.5 * 6) =	3.0	* 0.855≈	2.56	2.56
Audio	(0.5 * 6) =	1.5	* 0.95≈	1.42	1.42
Grafika	(1.0 * 6) =	6.0	* 1.045≈	6.27	6.27
Ispis	(1.25 * 6) =	7.5	* 1.2≈	9.0	9.0
Testiranje	(0.25 * 6) =	1.5	* 0.9≈	1.35	1.35
	sabrano	33.0		38.42	46.42
N sati*=Neprilagođeni sati			I sati*=ukupni izračunati sati		

Slika 3.8 Procena ukupno potrošenih sati uračunavajući i korekcione faktore [Izvor: NM SE321 -2020/2021.]

Slika 3.9 Rezime metrike estimacije za Courseware razvoj [Izvor: NM SE321 -2020/2021.]

Mogu se izvesti sledeći zaključci:

- **Na početku faze održavanja je potrebno najveće angažovanje jer je potrebno izvršiti obuku korisnika, otpočeti njegovo korišćenje, otkloniti zaostale bagove, uspostaviti i poboljšati proces održavanja.**
- **Nakon prve faze opada intenzitet angažovanja (ako je bilo uspešno) jer su otklonjeni početni problemi i tzv. "dečije bolesti", tj. često se koristi i izraz "uhodavanje sistema".**
- **No, tokom vremena dolazi do evolucije okruženja u kome se koristi softver jer se menjaju i inoviraju drugi sistemi koji su u vezi s posmatranim. Sem toga, Evoluira i šire okruženje (ne samo tehničko) tj. menjaju se zahtevi korisnika, zakonske, etičke i druge norme, i drugo.**
- **Stoga je potrebno izvršiti prilagođenje sistema novim uslovima. Tokom vremena to postaje sve skuplje i skuplje, tako da se u jednom trenutku donosi odluka o povlačenju proizvoda iz upotrebe (eng. Retirement - "penzionisanje")**
- **Održavanje je integralni deo životnog veka softvera i ima za cilj povećanje rentabilnosti ulaganja u softver i opremu obezbeđivanjem što duže operativnosti istih.**

▼ Poglavlje 4

Studija slučaja: Održavanja softvera za ATM

ATM - ORGANIZACIJA TESTIRANJA

U suštini, ATM predstavlja samo interfejs (vezu) između klijenta i informacionog sistema banke – koji obavlja sve značajne operacije.

ATM je deo vrlo složenog sistema banke. Predmet ovog plana testiranja neće biti veza ATM-a i banke, kao ni bančin informacioni sistem (IS). Veza sa bankom će, u prvim fazama razvoja, biti simulirana raznim stabovima, a IS banke će biti analiziran samo do nivoa potrebnog da bi se utvrdili tipovi podataka i struktura baze koja pohranjuje podatke o klijentima i njihovim računima. Takođe, neće biti testirana ni hardverska ispravnost ATM-ova, jer se prepostavlja da je to obavio proizvođač pre isporuke.

Ovaj plan obuhvata samo GEATM, kao softver koji opslužuje ATM, kao i zahteve koji su definisani u početnoj fazi projekta.

Ciljevi koji se žele postići testiranjem su da:

– Sve zahtevane funkcionalnosti GEATM-a budu implementirane i ispravno funkcionišu, u skladu sa zahtevima klijenta;

– Softver ima zahtevane performanse;

– Je softver pouzdan, robustan i lako se oporavlja nakon otkaza;

– Je softver lak za korišćenje i održavanje;

– Je softver bezbedan;

– **Ima pravilnu interakciju sa potrebnim spoljnjim elementima (hardver i softver).**

Ono što sledi je lista mogućnosti koje će biti testirane, kao i oznake datoteka sa izvornim kodom.

– Podizanje sistema, **Gašenje sistema, GUI – Interfesi sa spoljašnjim svetom (prema IS banke), Rad sa računima (transferi)**, Rad sa karticama, Autentifikacija (sigurnost, zadržavanje kartice i sl.), Izdavanje novca, Performanse (vreme odziva na korisničku akciju, dostupnost i sl.), Beleženje loga upotrebe bankomata, Podešavanje iznosa u kojima se može podići novac.

Mogućnosti i funkcije koje neće biti testirane: Kako je već opisano u uvodu, predmet ovog testiranja će biti samo softver koji upravlja ATM mašinom. Prema tome, ovim planom neće biti obuhvaćen sam hardver ATM mašine (osim do nivoa neophodnog da bi se utvrdila kompatibilnost koda sa konkretnom platformom), niti ostale komponente informaciono – komunikacionog sistema čiji je ATM mašina deo. Pretpostavka je da su ove komponente sistema ispravne.

KRITERIJUMI ZA ULAZ/IZLAZ IZ FAZE TESTIRANJA

Faza testiranja će se smatrati završenom ukoliko su odrađeni svi planirani testovi.

Da bi neki softverski modul uopšte dospeo do faze testiranja, mora zadovoljiti sledeće kriterijume:

- Sav kod se mora ispravno kompajlirati;
- Svi testovi razvojnih timova moraju biti uspešno izvršeni;
- Korisnički zahtevi se ne mogu menjati;
- Kod je zamrznut.

Nakon ovoga, SUT dobija Prošao/Nije prošao i u zavisnosti od toga preduzimaju se odgovarajući koraci.

Steer IT rangira greške/defekte od 1 do 4, gde 1 predstavlja grešku koja ima katastrofalan uticaj na sistem/korisnike, a 4 predstavlja grešku koja ima minimalni efekat na sistem/korisnike. Koje greške spadaju u koju kategoriju zavisi od konkretnog SUT-a. Precizna kategorizacija grešaka, kao i definicije pojmove „greška“, „defekat“ i „otkaz“. S obzirom na kritičnost sistema u kom GEATM treba da funkcioniše, kriterijumi prolaznosti će biti vrlo visoki.

Ovo znači sledeće:

- Broj grešaka kategorije 1=0;
- Broj grešaka kategorije 2=0;
- Broj grešaka kategorije 3=0;
- Broj grešaka kategorije 4<3 po modulu;
- Broj uspešno obavljenih testova=99 %;
- Svi planirani testovi moraju biti izvršeni.

Za ovaj softver će se smatrati prihvatljivim ukoliko se utvrdi postojanje najviše 2 greške kategorije 4 u pojedinačnom modulu. Ovo znači da se u tom slučaju testiranje može nastaviti. Međutim, svi otkazi/anomalije i defekti koji su ih izazvali moraju biti zabeleženi i prosleđeni odgovarajućoj referenci. Svi otkazi će biti predočeni razvojnom timu i imaće prioritet pri kasnijim opravkama, koje će biti propraćene regresionim testovima. Ukoliko neki SUT ne ispunji gore navedene kriterijume, smatraće se da nije prošao testiranje i vratiće se na doradu onom timu za koji se utvrdi da je odgovoran za grešku.

ATM - POTREBNO OSOBLJE I OBUKA

Usled vrlo ograničenog vremenskog roka, u procesu testiranja će se koristiti samo oni alati sa kojima su članovi tima već familijarni ili čije korišćenje može biti brzo naučeno.

Budući da je naša organizacija na 3. TMM nivou, svo naše osoblje je već dobro upoznato i ima iskustva sa:

- Opštim tehnikama razvoja i testiranja
- Svim razvojnim i alatima za automatizovano testiranje koji mogu biti korišćeni.

Tim za testiranje će činiti četiri člana naše organizacije, QA menadžer projekta i stručni konsultant koji je stalni saradnik naše organizacije. Na testiranju će puno radno vreme biti angažovani samo zaposleni iz naše organizacije, dok će ostali biti uključeni u testiranje po potrebi, najviše pola radnog vremena. Ukoliko se ukaže potreba moguće je angažovanje dodatne radne snage iz redova pripadnika naše organizacije. O ovome odluku donosi test menadžer u dogовору са menadžerom projekta.

Stručni konsultant za testiranje dužan je da timu za testiranje predoči najsavremenije tehnike i metode testiranja softvera i uputi na najkritičnije tačke projekta na koje posebno treba obratiti pažnju prilikom testiranja.

Takođe, odobravanjem ovog plana, verifikuje da su u procesu testiranja korišćeni najbolji raspoloživi alati i tehnike.

Menadžer projekta pravi glavni plan realizacije projekta, i dodeljuje resurse (vreme, novac i sl.) timu za testiranje. Ukoliko bude potrebno, donosi odluke o prekidanju testiranja, produženju roka za završetak testiranja i sl.

QA menadžer prvenstveno je zadužen za nadgledanje svih faza implementacije, usmeravanje tima za osiguranje kvaliteta i redovno izvještavanje menadžera projekta o rezultatima sprovedenih testova.

Test menadžer izrađuje plan testiranja raspolažući resursima koji su mu dodeljeni glavnim planom realizacije projekta. Ostale aktivnosti koje obavlja su interakcija sa QA menadžerom i menadžerom projekta, upravljanje timom koji obavlja testiranje i kontrola produktivnosti članova. Takođe, u obaveze test menadžera spada i definisanje test slučajeva koje će kasnije sprovoditi testeri pod njegovim vođstvom.

Testeri će, u prvim fazama projekta, biti zaduženi za sprovođenje modularnog testiranja, a kasnije i za realizaciju sistemskih testova. Prilikom svih navedenih aktivnosti i za svaki urađeni test će sastaviti Izveštaj o testiranju, a u slučaju pojave bilo kakvih nepravilnosti u radu SUT-a napraviće i Izveštaj o greškama.

ATM - PROJEKTOVANJE SOFTVERA

Cilj projekta obuhvata povećanje produktivnosti rada ATM-a u smislu implementaciju veoma sofisticiranih algoritama koji će se koristiti pri pretraživanju baze podataka

Za svrhe ovog primera, pretpostavljamo da uvodimo software za podršku ATM mašine u hipotetičku kompaniju koju ćemo nazvati "Cocomo d.o.o". Pretpostavljamo da firma "Cocomo" poseduje centralu, odakle se vrši upravljanje i raspodela zadataka. Takođe ćemo pretpostaviti da imamo 4 objekata koja će se baviti sprovođenjem zadataka, uključujući sektore za instalaciju i distribuciju rešenja, dobijenih od glavnog tela firme, centrale firme.

Naravno, cilj firme je, između ostalog, i proširivanje delovanja na nivou države uz tendenciju proširenja i van granica iste. Pored toga, "Cocomo" ima za cilj uspostavljanje saradnje sa firmama iz sfere proizvodnje softverskih rešenja za banke.

"Cocomo" firma je na početku svog rada koristila softverska rešenja zasnovana na minimumu usluga koje su ta rešenja pružala, pružajući šturu i nekompletну podršku korisnicima ATM

mašina. Stoga, usled tendencije povećanja obima posla u sferi bankarstva, "Cocomo" se odlučila na implementaciju efikasnijih softverskih rešenja poput GEATM softverskog proizvoda, kao podršku svom poslovanju, a sve u cilju zadovoljenja korisničkih zahteva.

Takođe ćemo pretpostaviti da firma "Cocomo" ima u planu da postavlja svoje ATM-ove na 20 lokacija u jednom gradu.

GEATM je veoma složen i sofisticiran softverski proizvod na bazi Sun-ove Java 1.5 platforme koji ima za cilj povećanje efikasnosti rada ATM mašina i pridobijanje velikog broja korisnika. Cilj projekta obuhvata povećanje produktivnosti rada ATM-a u smislu da GEATM ima za implementaciju veoma sofisticirane algoritme, koji će se koristiti pri pretraživanju baze podataka kao i pri ažuriranju stanja podataka sa pojedinačnih računa.

ATM - ODRŽAVANJE SOFTVERA

Primer održavanja softvera prikazan je na ATM uređaju organizovanom korišćenjem strategija uzorka (pattern) je jedan od dizajn obrazaca.

Primer održavanja softvera prikazan je na ATM uređaju organizovanom korišćenjem strategija uzorka (pattern) je jedan od dizajn obrazaca.

Ovaj uzorak se koristi kada postoji višestruki algoritam za određeni zadatak i u situaciji kada se klijent odluči za primenu koja će se koristiti u toku rada. Strategija uzorka je takođe poznata kao Policy obrazac. Definiše više algoritama i dozvoljava da klijent aplikacija prođe algoritam koji se koristi kao parametar. Jedan od najboljih primera ovog obrasca je Collections.sort () metod koji koristi komparator parametar. Na osnovu različitih implementacija COMPARATOR interfejsa, objekti koji su uzeti su upoređeni na različite načine. Na ovom primeru, je prikazano sprovođenje jednostavne Korpe gde postoje dve strategije plaćanja:

- koristeći kreditnu karticu ili
- koristeći PayPal.

Pre svega će se stvoriti interfejs za strategiju uzorak, u ovom slučaju u svrhu plaćanja iznosa koji je prošao kao argument .

Za detaljan uvid u kompletan dokument potrebno je preuzeti "ATM dokument" koji je dat nakon ovih vežbi.

```
package com.journaldev.design.strategy;

public interface Korpa {
    public void placanje (int suma);
}
```

Slika 4.1 Primer programskog koda [Izvor: NM SE321 -2020/2021.]

Slika 4.2 Test PayPal dela [Izvor: NM SE321 -2020/2021.]

✓ Poglavlje 5

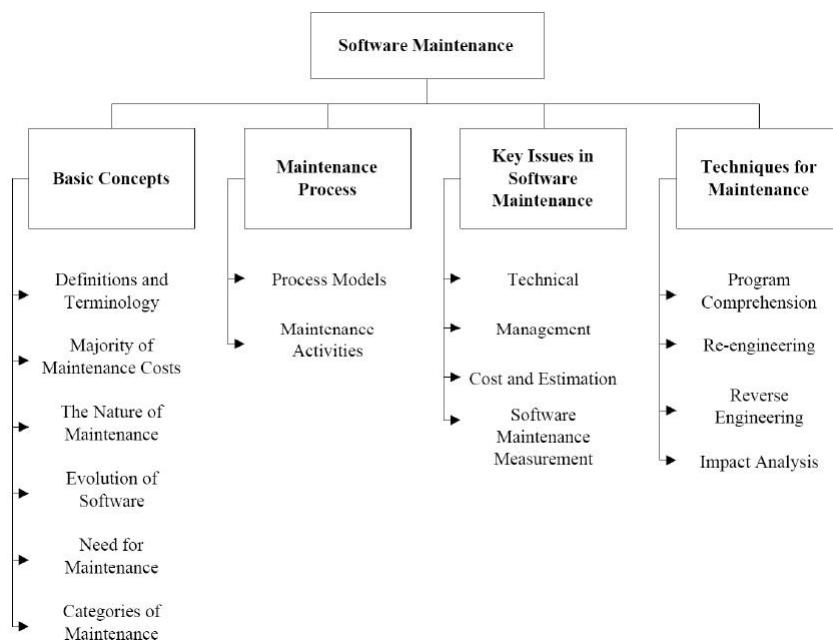
Održavanja softvera po SWEBOK 2004

OBLASTI ODRŽAVANJA SOFTVERA PO SWEBOK 2004

To su: Osnovni koncepti održavanja; Proces održavanja; Osnovni aspekti održavanja; Tehnike održavanja

Oblasti održavanja softvera su dekompozicija softverskog inženjerstva na teme koje su "generalno prihvaćene" u svetu održavanja softvera a one su (Slika 1):

- Osnovni koncepti održavanja
- Proces održavanja
- Osnovni aspekti održavanja
- Tehnike održavanja



Slika 5.1 Oblasti održavanja softvera [Izvor: NM SE321 -2020/2021.]

Teme su u skladu s aktuelnom situacijom u literaturi i standardima softverskog inženjerstva. Sledeća slika je preuzeta iz ref SWEBOK 2004, i na njoj su dati dalje pod oblasti navedenih oblasti.

Održavanje softvera definisano je i u IEEE 1219 standardu za Održavanje softvera kao:

Modifikacija programskog proizvoda nakon isporuke u cilju otklanjanja defekata, poboljšanja performansi ili drugih atributa, ili u cilju prilagođenja proizvoda izmenjenom okruženju.

Isti standard takođe definiše i aktivnosti održavanja pre isporuke softverskih proizvoda. ISO / IEC 12207 standard za životni vek procesa, u suštini opisuje održavanje kao jedan od primarnih procesa životnog veka i opisuje održavanje kao:

"MODIFIKACIJA KODA I PRIPADAJUĆE DOKUMENTACIJE USLED OTKLANJANJA PROBLEMA ILI POTREBE ZA POBOLJŠANJEM"

Cilj je da se postojeći softverski proizvod menja, uz očuvanje integriteta.

Cilj je da se postojeći softverski proizvod menja, uz očuvanje integriteta. ISO / IEC 14764 Međunarodni standard za Održavanje softvera, definiše održavanje softvera, slično kao ISO / IEC 12207, i stavlja naglasak na aktivnosti održavanja pre isporuke, recimo - planiranje.

Prema ISO / IEC 12207 osnovne aktivnosti održavanja softvera su:

- implementacija procesa,
- analiza problema i modifikacija,
- implementacija modifikacija,
- recenzije održavanja / prihvaćanje,
- migracija (preseljenje),
- "penzionisanje" (povlačenje iz upotrebe).

Održavanje može imati širi spektar aktivnosti od razvoja, s više izmena za praćenje i kontrolu. Otuda, upravljanje konfiguracijom softvera je važan aspekt razvoja i održavanja. Kontakt sa proizvođačem je veoma bitan za održavanje, jer treba preuzeti softver, dokumentaciju, alate za održavanje i izvršiti obuku. Jako je bitno da se ovo isplanira i uradi blagovremeno.

Promene u okruženju u kojima korisnik deluje, uzrokuju promenu uslova rada softvera što dovodi do evolucije zahteva za softver. Otuda, razvoj nove verzije softvera ima sve karakteristike kao i razvoj početne verzije, sem što sada postoji i prethodna verzija, što može znatno komplikovati situaciju.

PROCENA TROŠKOVA ODRŽAVANJE SOFTVERA

*Procena troškova održavanje softvera je važan aspekt planiranja.
Trošak održavanja predstavlja znatan deo ukupnih troškova i stoga treba da bude pažljivo analizirana njegova struktura*

Troškovi održavanja softvera zavise od nekoliko faktora kao na primer:

- **Ograničeno poznavanje održavanog softvera** je bitan faktor jer se od 40% do 60% rada posvećeno razumevanju softvera koji se želi modifikovati. Softverski tim zadužen za održavanje, poseduje ograničeno poznavanje softvera i mora ga naučiti uz korišćenje.
- **Testiranje** je takođe bitan faktor jer ga treba izvršiti nakon modifikacija. Stoga se primenjuju tehnike regresionog testiranja kao i tehnike selektivnog testiranja radi smanjenja troškova.
- **Analiza posledica** (*impact analysis*) je veoma važna u cilju smanjenja rizika.
- **Lakoća održavanja** (*Maintainability*) je svojstvo softvera koji pokazuje lakoću s kojom softver može biti održavan, poboljšan, prilagođen ili korigovan u skladu s postavljenim zahtevima. Ona se može poboljšati kroz dizajn i kodiranje, pa u tu svrhu treba da bude deo specifikacije sistema.

Postoji stalno pitanje ulaganja u održavanje, jer se neposredno povećavaju troškovi, a korist nije uvek očigledna.

Tim koji razvija softver neće uvek preuzeti i održavanje, pa je potrebno oformiti poseban tim. **Outsourcing** održavanja se veoma često primenjuje: U oba slučaja treba pažljivo proanalizirati dobiti i troškove pri izboru rešenja. Slično se može reći i za organizaciju tima za održavanje.

Procena troškova održavanje softvera je važan aspekt planiranja. Trošak održavanja predstavlja znatan deo ukupnih troškova, i stoga treba da bude pažljivo analizirana njegova struktura, kao i faktori koji utiču na lakoću održavanja (*maintainability*). Analiza posledica (*impact analysis*) identificuje koji su sve sistemi zavisni i omogućava procenu troškova modifikacija.

Procena troškova se može vršiti na bazi iskustva, ili primenom parametarskih modela (kao što je COCOMO). Softverska merenja se takođe mogu koristiti za dobijanje podataka o troškovima.

✓ Poglavlje 6

Grupna vežba

REKAPITULACIJA TESTIRANJA, ORGANIZACIJE I ODRŽAVANJA SOFTVERA

Na osnovu prethodnih vežbi potrebno je rekapitulirati i demonstrirati proces testiranja, organizacije i održavanja softvera.

Za odabrani deo ISUM-a namenjen studentima (pregled predispitnih obaveza, prijava ispita, uvid u finansije, pregled položenih ispita, biblioteka), svaki student treba da uz pomoć asistenta a na osnovu do sada stečenog znanja iz predmeta da predlog:

1. Adekvatnog procesa testiranja, a time i aktivnosti, kriterijume, resurse testiranja.
2. Adekvatnog procesa održavanja, a time aktivnosti, kriterijume, resurse održavanja.

Vreme trajanja 45 minuta.

✓ Poglavlje 7

Individualna vežba 15

ODBRANA PROJEKTA

U okviru individualnih vežbi u 15. nedalji predavanja predviđena je odbrana projekata

U okviru individualnih vežbi u 15. nedalji predavanja predviđena je odbrana projekata. **Vreme trajanja 90 minuta**

✓ Poglavlje 8

Domaći zadatak

PETNAESTI DOMAĆI ZADATAK

Nakon petnaeste lekcije potrebno je uraditi petnaesti domaći zadatak.

Za odabranu aplikaciju koju ste radili na nekom od predmeta koje ste prethodno slušali i položili dajte predlog:

1. Adekvatnog procesa testiranja, a time i aktivnosti, kriterijume, resurse testiranja.
2. Adekvatnog procesa održavanja, a time aktivnosti, kriterijume, resurse održavanja.

Domaći zadaci treba da budu realizovani u razvojnom okruženju koje je definisano domaćim zadatkom i da predstavljaju jedinstveno rešenje svakog studenta.

Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

DOMAĆI ZADATAK - UPUTSTVO

Prilikom izrade domaćeg zadatka potrebno je pridržavati se uputstva za izradu.

Domaći zadatak se imenuje: SE321-DZ15-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br. Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta:

sara.nikolic@metropolitan.ac.rs (za studente u Beogradu i internet studente)
jovana.jovic@metropolitan.ac.rs (za studente u Nišu)
sa naslovom (subject mail-a) SE321-DZ15.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Forma (templet) domaćeg zadatka je data na sledećim stranicama. Obim rešenja domaćeg zadatka je od 3 do 6 strana (tekst, slike, programski kod i sl.).

▼ Zaključak

ZAKLJUČAK

Šta smo naučili u ovoj lekciji?

U ovoj lekciji su opisane funkcije i servisi kao i sama arhitektura OptimalSQM koja je primer SOA. OptimalSQM rešenje sadrži komponente-servise (OQT MNGR, OQT BOX, OQT MAINT, OQT OPST, OQT SIM) i dostupan je kao sveobuhvatni paket rešenja za upravljanje testiranjem i simulacijom mogućih scenarija procesa testiranja konkretne kompanije i konkretnog projekta. Takođe je studiran slučaj softvera – *Courseware*, koji se definiše kao sistem od više grupa lekcija pomoći nekoliko podsistema. Sistem učeniku omogućava da uđe i izade iz njega, kao i da sačuva rezultate svog testa. Lekcija je definisana autoringom softvera koji se koristi za izradu Courseware-a. Pošto su istorijski podaci prikupljeni tokom razvoja programa, lekcija je definisana kao knjiga koja se sastoji od stranica prezentovanog materijala.

U lekciji su u okviru rezimea o održavanju softvera, opisani i faktori od kojih zavise troškovi održavanja kao što su na primer:

- Ograničeno poznavanje održavanog softvera koje je bitan faktor, jer je od 40% do 60% rada posvećeno razumevanju softvera koji se želi modifikovati. Softverski tim za održavanje ima ograničeno poznavanje softvera i mora ga naučiti uz korišćenje.
- Testiranje koje je takođe bitan faktor, jer ga treba izvršiti nakon modifikacija. Stoga se primenjuju tehnike regresionog testiranja kao i tehnike selektivnog testiranja radi smanjenja troškova.
- Analiza posledica (**impact analysis**) koja je takođe veoma važna u cilju smanjenja rizika.

LITERATURA ZA LEKCIJU 15

U izradi ove lekcije korišćena je navedena literatura.

Obavezna literatura:

1. P. Grubb and A.A. Takang, Software Maintenance: Concepts and Practice, 2nd ed., World Scientific Publishing, 2003. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
2. S. Naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley-Spektrum, 2008. (Link: <https://www.pdfdrive.com/> ili <http://it-ebooks.info/book/>)
3. Kan S.H., Metrics and Models in Software Quality Engineering, second edition, Addison Wesley, 2002. (Link:<https://www.pdfdrive.net/> ili <http://it-ebooks.info/book/>)
4. V. B. Spasojević, i ostali, Organizacija sistema kvaliteta i alati kvalitet, Industrija 2004, vol. 32, br. 4, str. 91-107

Dopunska literatura:

1. Samir Lemeš, Nevzudin Buzadija, STANDARDNI MODELI ODRŽAVANJA SOFTVERA, 5. Konferencija „ODRŽAVANJE - MAINTENANCE 2018“ Zenica, B&H, 10. – 12. Maj 2018.
2. Lj. Lazić , Software Quality & Testing Metrics, WSEAS 7th WSEAS EUROPEAN COMPUTING CONFERENCE (ECC '13), Dubrovnik, Croatia, June 25-27, 2013 . (link: <http://www.wseas.org/wseas/cms.action?id=4102>

Veb lokacije:

1. <http://www.qsm.com/>
2. <https://www.computersciencezone.org/software-quality-assurance/>
3. <http://www.pisa.rs>