UNIVERZITET MB
Beograd

POSLOVNI I PRAVNI FAKULTET - BEOGRAD

TESTIRANJE SOFTVERA

- PREDAVANJA 3 -

Predavač: Prof. dr Borivoje M. Milošević

2022

Osnovni koncepti i metodologije

Testiranje softvera je proces evaluacije i provere da li softverski proizvod ili aplikacija radi ono što bi trebao raditi. Prednosti testiranja uključuju sprečavanje bagova, smanjenje troškova razvoja i poboljšanje performansi.

Malo ko može osporiti potrebu za kontrolom kvaliteta prilikom razvoja softvera. Kasna isporuka ili nedostaci softvera mogu naštetiti ugledu robne marke - što dovodi do frustriranih i izgubljenih kupaca. U ekstremnim slučajevima, bag ili kvar mogu degradirati međusobno povezane sisteme ili uzrokovati ozbiljne kvarove.

Uzmimo u obzir da Nissan mora povući više od milion automobila zbog softverskog kvara na detektorima senzora vazdušnih jastuka.

Ili, softverska greška koja je uzrokovala neuspeh lansiranja vojnog satelita vrednog 1,2 milijarde USD. Brojke govore same za sebe. Softverski kvarovi u SAD-u koštali su privredu 1,1 bilion dolara u 2016. Štaviše, utiecali su na 4,4 milijarde korisnika.

Osnovni koncepti i metodologije

Testiranje softvera je integralni deo procesa razvoja softvera, sa ciljem osiguravanja kvaliteta finalnog proizvoda. U ljudskoj prirodi je da pravimo greške. Neke greške mogu biti sitne i nebitne, dok druge mogu biti veoma ozbiljne i skupe, pa čak i dovesti do ljudskih žrtava.

Testiranje softvera je neophodno kako bi se otkrile greške načinjene u svim fazama razvoja softvera, koje se nakon toga mogu ispraviti. Otkrivanjem i ispravljanjem grešaka se obezbeđuje viši nivo kvaliteta softvera, čime se stiče poverenje i zadovoljstvo krajnjih korisnika softvera. U ovom delu fokus je na tehnikama otkrivanja grešaka u softveru.

Osnovni koncepti i metodologije

Istorija testiranja softvera:

Pojam testiranja softvera je star koliko i samo programiranje. Još od prvih napisanih programa je bilo neophodno proveriti da li se program ponaša ispravno i na takav način kako je definisano. U današnje vreme, iako možda to na prvi pogled ne izgleda tako, naš način života u potpunosti zavisi od softvera koji je ključni faktor u velikom boju sistema koje svakodnevno koristimo. Softver definiše ponašanje mrežnih rutera, bankarskih mreža, telefonskih sistema i samog Interneta. Dalje, softver je osnovna komponenta mnogih ugrađenih aplikacija koje kontrolišu rad veoma složenih sistema poput aviona, kontrole letenja, svemirskih brodova, a može se pronaći i u nešto prostijim uređajima (uslovno rečeno) poput automobila, mobilnih telefona, satova, DVD plejera, mikrotalasnih peći i druge bele tehnike.

Prema procenama, u modernom domaćinstvu postoji preko 50 procesora. Ako se posmatraju noviji modeli automobila, u svakom postoji preko 100 procesora. Na svakom od ovih procesora se izvršava softver, a optimistični krajnji korisnici smatraju da će sav taj softver da se izvršava bez grešaka. Postoje mnogi faktori koji mogu da utiču na kvalitet softvera i njegovu pouzdanost. Pažljiv dizajn i projektovanje softvera svakako mogu pomoći da softver bude pouzdan, ali osnovna metoda koja se u industriji koristi za evaluaciju softvera koji se razvija jeste testiranje softvera. Testiranje softvera, kao integralni deo razvoja softvera, ima za cilj da osigura da je kvalitet softvera na odgovarajućem nivou (engl. *Quality Assurance*).

Osnovni koncepti i metodologije

Istorija testiranja softvera:

Testiranje softvera razvija se uporedo sa razvojem softvera, koji je započeo tek nakon Drugog svjetskog rata. Informatičar Tom Kilburn zaslužan je za pisanje prvog dela softvera, koji je predstavljen 1948. god. na Univerzitetu Manchester u Engleskoj. Izvodio je matematičke proračune koristeći instrukcije mašinskog koda. Otklanjanje pogrešaka bila je glavna metoda testiranja u to vreme i tako je ostala narednih desetak godina.

Do 1980-ih, razvojni timovi gledali su dalje od izolacije i popravljanja softverskih grešaka do testiranja aplikacija u stvarnom svetu. Postavlja se teren za širi pogled na testiranje, koje je obuhvaćato proces osiguranja kvaliteta koji je bio deo životnog ciklusa razvoja softvera.

„Devedesetih godina prošlog veka došlo je do prelaza sa testiranja na sveobuhvatniji proces nazvan osiguranje kvaliteta, koji pokriva celi ciklus razvoja softvera i utieče na procese planiranja, dizajna, kreiranja i izvođenja testnih slučajeva, podršku za postojeće testne slučajeve i testiranje okruženja”, kaže Alexander Yaroshko u svom postu na web-mestu za programere uTest. “Testiranje je dostiglo kvalitativno novi nivo, što je dovelo do daljeg razvoja metodologija, pojave moćnih alata za upravljanje procesom testiranja i alata za automatizaciju testiranja.”

Osnovni koncepti i metodologije

Čuveni primeri softverskih otkaza:

Značaj testiranja softvera je možda najlakše prikazati na konkretnim primerima softverskih otkaza, kada testiranje softvera nije izvršeno na adekvatan način i kada greške nisu otkrivene na vreme. U slučaju industrijskog softvera, ove greške često dovode do velikih finansijskih gubitaka, a ponekad, na žalost, i do ljudskih žrtava. Čak i u najpovoljnijem slučaju, greške koje su prisutne u finalnom isporučenom softveru dovode do smanjivanja zadovoljstva krajnjih korisnika, što kao posledicu može imati gubitak poverenja u dati sistem i okretanje alternativnim rešenjima.

- ✓ U januaru 1990. godine, nijedan korisnik američke telekomunikacione kompanije AT&T nije mogao da uspostavi pozive na velikim rastojanjima, zbog softverske greške na relejnim svičevima.
- ✓ NASA - Mars Climate Orbiter misija je doživela ogroman neuspeh 1999. godine. U okviru svoje misije na Marsu, zbog greške u softveru letelica je nepovratno izgubljena u svemiru. Nakon duže istrage koja je usledila, otkriveno je da je napravljena greška pri konverziji imperijalnih jedinica u metrički sistem.
- ✓ Evropska Svemirska Agencija je, poput NASA, takođe je imala svoje trenutke, od kojih se izdvaja misija Ariane 5 let 501. Samo 40 sekundi nakon lansiranja 1996. godine, letelica vredna 370 miliona $ se raspala zbog softverske greške. Problem je izazvalo prekoračenje u konverziji float -> int. Inercijalni referentni sistem je radio sa 64-bitnim podacima tipa float, koje je konvertovao u 16-bitni integer, pri čemu je izazvan aritmetički overflow – šteta 10 milijardi $.

Osnovni koncepti i metodologije

Edukativni i komercijalni softver:

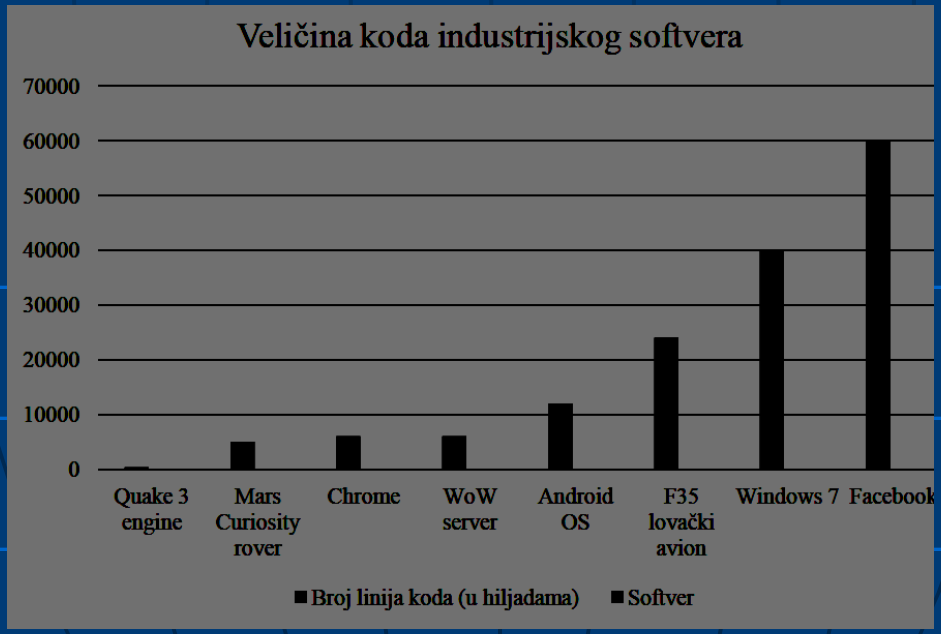
Da bi se bolje razumela potrebu za testiranjem softvera, neophodno je pojasniti pojam komercijalnog softvera i razlike u odnosu na programe sa kojima studenti dolaze u kontakt za vreme studija. Programi koji se razvijaju u okviru nastave na fakultetima se mogu definisati kao softver za edukaciju. Takav softver se programira sa ciljem demonstracije na predavanjima i vežbama, zatim ispitnim zadacima, kao i u okviru samostalnog učenja. Ono što ga karakteriše jeste da ne rešava neki konkretan problem, a samo prisustvo grešaka (iako naravno nije poželjno) nije previše zabrinjavajuće.

Komercijalni softver, sa druge strane, se razvija sa tačno definisanim ciljem, kako bi rešio neki konkretni problem krajnjih korisnika. Neophodno je da se program korektno izvršava, a greške u radu programa mogu izazvati nezadovoljstvo korisnika, finansijske gubitke, gubitke značajnih podataka, čak i ljudske žrtve. Kako bi se obezbedilo korektno izvršavanje softvera, softver se razvija po fazama, sa jasnim zahtevima i dokumentacijom, i adekvatno testira.

Osnovni koncepti i metodologije

Edukativni i komercijalni softver

Proste aplikacije za mobilne telefone mogu biti veličine nekoliko hiljada linija koda, dok su prosečne aplikacije za iPhone reda veličine 50 hiljada linija koda. Veličina kompletnog koda koji se nalazi na Curiosity roveru na Marsu je reda veličine 5 miliona linija koda. Serverski kod popularne kompjuterske igre World of Warcraft je veličine 6.5 miliona linija koda, a najnovija verzija Google Chrome je oko 7 miliona linija koda. Kompletan operativni sistem Android je veličine oko 12 miliona linija koda. Prosečna veličina ukupnog softvera koji se nalazi u modernim automobilima iznosi preko 100 miliona linija koda.
Testiranje komercijalnog softvera je neophodno. Prema procenama američkog nacionalnog instituta za standarde (NIST), softverski bagovi izazivaju gubitke u vrednosti od više desetina milijardi dolara godišnje samo u SAD.



Osnovni koncepti i metodologije

Iako samo testiranje košta, kompanije mogu uštedeti milione godišnje na razvoju i podršci, naravno ako imaju dobru tehniku testiranja i QA procese na mestu.

Rano testiranje softvera otkriva probleme pre nego što proizvod izađe na tržište. Što pre razvojni timovi dobiju povratne informacije o testiranju, to će pre moći rešiti probleme kao što su:

- ✓ Arhitektonske greške
- ✓ Loše dizajnerske odluke
- ✓ Neispravna ili netačna funkcionalnost
- ✓ Sigurnosne ranjivosti
- ✓ Problemi sa skalabilnošću

Kada razvoj ostavi dovoljno prostora za testiranje, poboljšava se pouzdanost softvera i isporučuju se visokokvalitetne aplikacije sa nekoliko pogrešaka. Sistem koji ispunjava ili čak nadmašuje očekivanja kupaca dovodi do potencijalno veće prodaje i većeg tržišnog udela.

Osnovni koncepti i metodologije

Testiranje softvera tradicionalno je odvojeno od ostatka razvoja. Često se provodi kasnije u životnom ciklusu razvoja softvera nakon faze izrade ili izvođenja proizvoda. Tester može imati samo mali prozor za testiranje koda - ponekad neposredno pre nego što aplikacija izađe na tržište.

Ako se pronađu nedostaci, može biti malo vremena za ponovno kodiranje ili ponovno testiranje. Nije neuobičajeno da se softver objavi na vreme, ali uz potrebne greške i popravke. Ili tim za testiranje može ispraviti pogreške, ali propustiti datum izdavanja.

Obavljanje testnih aktivnosti ranije u ciklusu pomaže da se proces testiranja zadrži u prvom planu, a ne kao naknadna misao u razvoju. Raniji softverski testovi takođe znače da je otklanjanje nedostataka jeftinije.

Mnogi razvojni timovi sada koriste metodologiju poznatu kao kontinualno testiranje. To je deo DevOps pristupa (Agilni razvoj) – gde razvoj i operacije sarađuju tokom celog životnog ciklusa proizvoda. Cilj je ubrzati isporuku softvera uz balansiranje troškova, kvaliteta i rizika. Uz ovu tehniku testiranja, timovi ne moraju čekati da se softver izgradi pre početka testiranja. Oni mogu pokrenuti testove mnogo ranije u ciklusu kako bi otkrili nedostatke mnogo pre, kada ih je lakše popraviti.

Osnovni koncepti i metodologije

Osnovna klasifikacija testiranja:
Prema pristupu, testiranje se deli na:

Funkcionalno – testiranje bazirano na specifikaciji	Funkcionalno testiranje posmatra program kao crnu kutiju, i implementacija u ovom slučaju nije poznata. Naziva se još metode crne kutije. Softver se prosto posmatra kao funkcija koja mapira vrednosti sa ulaza na izlaz sistema. Testovi se određuju isključivo na osnovu specifikacije softvera.
Strukturno – testiranje bazirano na samom kodu	Strukturno testiranje se fokusira na samoj implementaciji programa i raspoloživom dostupnom kodu. Naziva se još **metode bele kutije**. Fokus je na izvršavanju svih programskih struktura i struktura podataka u softveru koji se testira, i na osnovu toga se određuju testovi.

Testiranje se prema nivou deli na:

Jedinično	Sprovode programeri (engl. unit testing) i odnosi se na testiranje pojedinačnih jedinica izvornog koda ili delova klase. Najmanja funkcionalna jedinica izvornog koda je najčešće jedna metoda unutar klase.
Integraciono	Glavni fokus je na verifikaciji funkcionalnosti i interfejsa između integrisanih modula.
Sistemsko	Proverava ponašanje tog sistema kao celine u odnosu na specifikaciju sistema.

Osnovni koncepti i metodologije

Vrste testiranja softvera:
Postoji mnogo različitih vrsta softverskih testova, svaki sa specifičnim ciljevima i strategijama:

Acceptance testing:	Provera funkcioniše li celi sistem kako je predviđeno.
Integration testing:	Obezbedjenje da softverske komponente ili funkcije rade zajedno.
Unit testing:	Provera da svaka softverska jedinica radi prema očekivanjima. Jedinica je najmanja komponenta aplikacije koja se može testirati.
Functional testing:	Provera funkcija emulacijom poslovnih scenarija, na temelju funkcionalnih zahteva. Testiranje crne kutije uobičajen je način provere funkcija.
Performance testing:	Testiranje kako softver radi pod različitim radnim opterećenjima. Ispitivanje opterećenja, na primer, koristi se za procenu performansi u uvjetima stvarnog opterećenja.
Regression testing:	Proverava da li nove postavke prekidaju ili degradiraju funkcionalnost. Testiranje razumnosti može se koristiti za proveru menija, funkcija i naredbi na površinskom nivou, kada nema vremena za potpuni regresijski test.
Stress testing:	Testiranje koliko opterećenja sistem može podneti pre nego što zakaže. Smatra se vrstom nefunkcionalnog testiranja.
Usability testing:	Provera koliko dobro korisnik može koristiti sistem ili web aplikaciju za dovršetak zadatka.

Osnovni koncepti i metodologije

U svakom slučaju, validacija osnovnih zahteva je kritična procena. Jednako važno, istraživačko testiranje pomaže testeru ili timu za testiranje otkriti teško predvidljive scenarije i situacije koje mogu dovesti do softverskih pogrešaka.

Čak i jednostavna aplikacija može biti podvrgnuta velikom broju i raznim testovima. Plan upravljanja testiranjem pomaže u određivanju prioriteta koje vrste testiranja pružaju najveću vrednost – sa obzirom na raspoloživo vreme i resurse. Korisnost testiranja optimizirana je izvođenjem najmanjeg broja testova kako bi se pronašao najveći broj nedostataka.

Testiranje softvera sledi uobičajeni proces. Zadaci ili koraci uključuju definisanje testnog okruženja, razvoj testnih slučajeva, pisanje skripti, analizu rezultata ispitivanja i podnošenje izveštaja o greškama. Testiranje može biti dugotrajno. Ručno testiranje ili ad-hoc testiranje može biti dovoljno za male konstrukcije. Međutim, za veće sisteme alati se često koriste za automatizaciju zadataka. Automatizirano testiranje pomaže timovima da implementiraju različite scenarije, testirati diferencijatore (kao što je premeštanje komponenti u okruženje oblaka) i brzo dobiti povratne informacije o tome šta radi, a šta ne.

Osnovni koncepti i metodologije

Dobar pristup testiranju obuhvata interfejs aplikativnog programiranja (API), korisnički interfejs i sve nivoe sistema. Isto tako, što je više testova koji su automatizovani i pokrenuti ranije, to bolje. Neki timovi izrađuju interne alate za automatizaciju testiranja. Međutim, rešenja dobavljača nude postavke koje mogu pojednostaviti ključne zadatke upravljanja testom kao što su:

Continuous testing:	Projektni timovi testiraju svaki korak čim postane dostupan. Ova vrsta testiranja softvera oslanja se na automatizaciju testiranja koja je integrisana sa procesom implementacije. Omogućuje validaciju softvera u realističnim testnim okruženjima ranije u procesu – poboljšavajući dizajn i smanjujući rizike.
Configuration management:	Organizacije centralizovano održavaju sredstva za testiranje i prate koji se softver gradi za testiranje. Timovi dobijaju pristup sredstvima kao što su kod, zahtevi, projektni dokumenti, modeli, testne skripte i rezultati testiranja. Dobri sistemi uključuju autentifikaciju korisnika i tragove revizije kako bi pomogli timovima da ispune zahteve usklađenosti uz minimalan administrativni napor.
Service virtualization:	Međutim, testna okruženja možda neće biti dostupna, naročito u ranoj fazi razvoja koda. Virtualizacija usluga simulira usluge i sisteme koji nedostaju ili još nisu dovršeni, omogućujući timovima da smanje zavisnosti i testiraju ranije. Mogu ponovo koristiti, implementirati i promeniti konfiguraciju kako bi testirali različite scenarije bez potrebe za izmenom izvornog okruženja.
Defect or bug tracking:	Praćenje nedostataka važno je i timovima za testiranje i razvojnim timovima za merenje i poboljšanje kvaliteta. Automatiziovani alati omogućuju timovima praćenje nedostataka, merenje njihovog opsega i uticaja pa i otkrivanje povezanih problema.
Metrics and reporting:	Izveštavanje i analitika omogućuju članovima tima da dele status, ciljeve i rezultate testiranja. Napredni alati integrišu metriku projekta i prikazuju rezultate na komandnoj liniji. Timovi brzo vide celokupno stanje projekta i mogu pratiti odnose između testiranja, razvoja i drugih elemenata projekta.

Nalaženje najoptimalnijeg seta za testiranje

Pokušaj automatizacije svih testova je nemoguć, a trošak pokušaja da se to učini na kraju nadmašuje korist. Za bilo koju razumno složenu aplikaciju, ne možemo testirati svaki mogući put kroz sistem, jer čak i ako postoji samo jedna petlja u aplikaciji, broj mogućih puteva postaje beskonačan. Ako zatim dodamo permutacije testnih podataka, brzo ćemo shvatiti da pokušaj testiranja svega jednostavno nije izvodljiv. Ključ je identifikovati najvažniji podskup testova - one u kojima:

» Obično nalazimo probleme

» U kojima smo videli regresije

» U kojima imamo pritužbe kupaca

» Za koje znamo da bi pojava kvara bila značajna ili čak katastrofalna

Osnovni koncepti i metodologije

Automatizacija testova:

Sistem automatizacije testiranja obično dolazi nakon upravljanja testovima, kada timovi otkriju da su svi njihovi ručno odradjeni testovi uzaludni, neefikasnii i u mnogim slučajevima, potpuno nemogući.

Uspešno stvaranje robusnog i upotrebljivog okvira za automatizaciju testiranja koji se može primeniti, može biti težak zadatak ako se nijemu nije pristupilo u samom projektu razvoja softvera. Takav proces ima potrebu da objedini zajedničku viziju, zahteve, arhitekturu, dizajn, kodiranje, a u nekim slučajevima i potvrdu da automatizacija radi za šta je bila namenjena. Bez ovih aspekata, ovo testiranje ima tendenciju da bude krhko, lomljivo, teško za održavanje, skupo za preuređivanje, i često napušteno.

IBM zagovara automatizaciju testiranja na svim slojevima aplikacije, uključujući sloj komponenti, sloj usluge i korisnički interfejs. Važno je pronaći pravu ravnotežu testova između njihovih slojeva, a industrija nalazi da prikazani postoci na sledećoj slici predstavljaju dobro polazište — zavisno od naše aplikacije.

Osnovni koncepti i metodologije

Automatizacija testova:

Prilikom određivanja ili pregleda strategije automatizacije testiranja, postavljaju se ova pitanja:

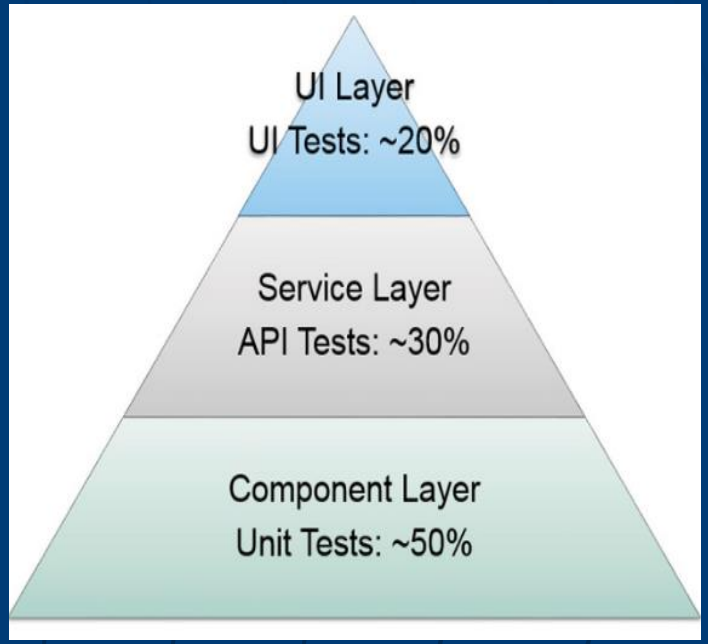
» Koliko smo delotvornii u ponovnom izvođenju postojećih testova?

» Izvodimo li većinu ili čak sve testove ručno?

» Ako automatizujemo testove, jesmo li fokusirani samo na funkcionalne testove na sloju korisničkog interfejsa, ili pokrećemo jedinice, funkcionalne testove API sloja, testove performansi, pa čak sigurnosne testove?

» Da li imamo robusni i održiv okvir za automatizaciju testiranja koji se može primeniti?

» Uključujemo li automatizirane testove u tok isporuke?



Preporučeni procenti primene automatizacije tedtiranja

Osnovni koncepti i metodologije

Automatizacija testova:

Dok programeri pišu kod, zar nema smisla pokrenuti prvi krug jediničnih testova? Izvršavanje jediničnih testova kao potvrda da programski kod ispravno radi u njihovom razvojnom okruženju uveliko bi pridonela izgradnji povjerenja tima za isporuku softvera.
Pa zašto nema toliko organizacija koje sprovode to kao politiku?

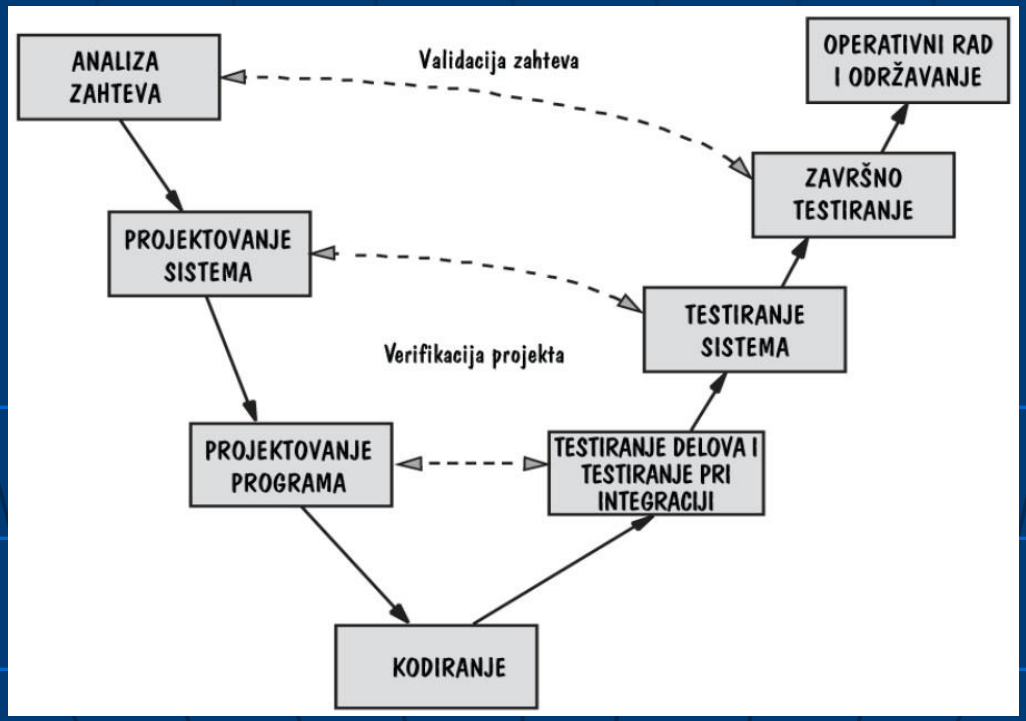
Zanemarivanje najboljih praksi koje bi mogle (tvrdim da bi trebale) biti isporučene mnogo ranije u životnom ciklusu razvoja softvera (SDLC) često rezultuje nepotrebnim kašnjenjima u kasnijim fazama i povećanim troškovima. Ako želimo isporučiti softver brže nego ikad pre uz istovremeno smanjenje poslovnog rizika, svaki član tima, uključujući programere, deli odgovornost za doprinos kvalitetu tog softvera.

Znači, neophodan je timski rad koji u jednoj metodologiji razvoja softvera izgleda:

Osnovni koncepti i metodologije

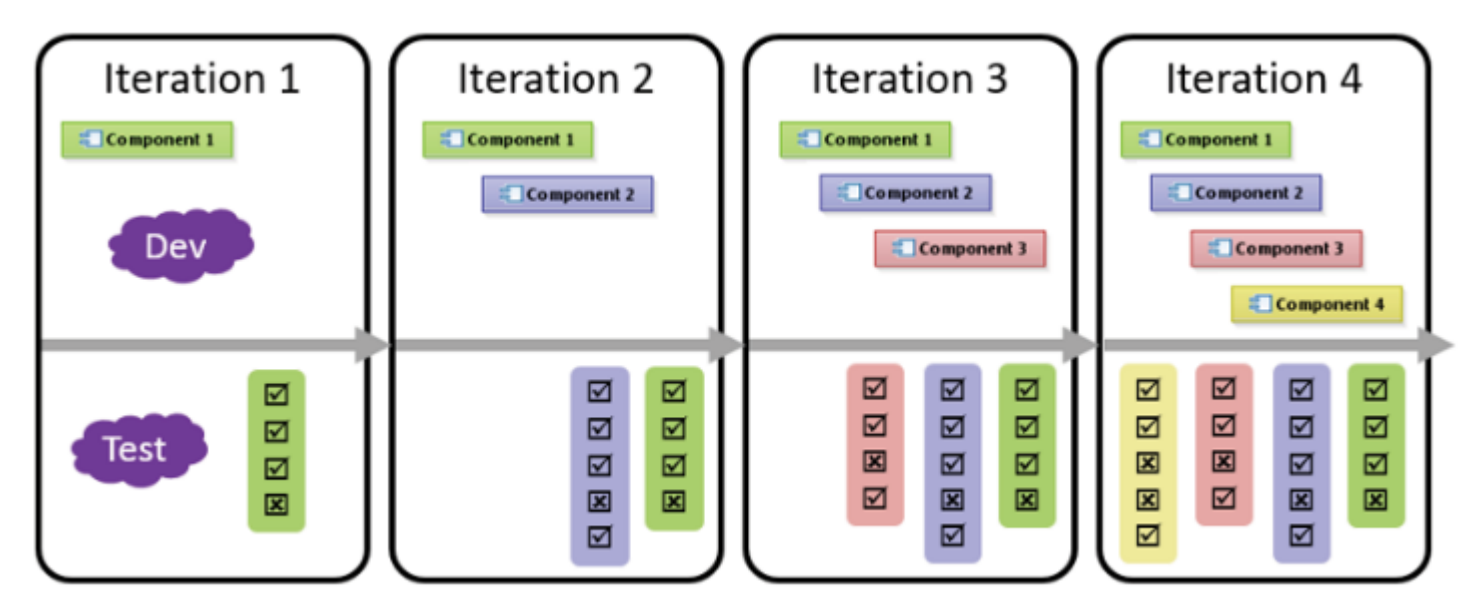
V model

V model (nemačko Ministarstvo odbrane, 1992.) je modifikovani kaskadni model koji pokazuje odnos testiranja i faza analize i projektovanja, čineći eksplicitnim neke povratne sprege koje su skrivene u kaskadnom modelu. V model sugeriše da testiranje delova i testiranje pri integraciji mogu da se koriste za verifikovanje projektovanog programa. Tokom testiranja delova i testiranja pri integrisanju programeri i članovi tima za testiranje treba da se osiguraju da su svi aspekti projektovanja ispravno implementirani u kodu. Slično tome, testiranje sistema treba da verifikuje projektovani sistem, uz potvrdu da su svi aspekti ispravno implementirani.



Osnovni koncepti i metodologije

Testiranje efikasnosti je kritičan aspekt za praćenje bržeg tempa razvoja životnih ciklusa. Optimizira se pokretanjem najmanjeg broja testova koji pronalaze najveći broj problema. Čak i timovi radeći u tradicionalnijim razvojnim životnim ciklusima, gde se ispitivanje sprovodi u jednoj fazi, utvrdili su da oni ne mogu pratiti regresijsko testiranje svaki put kad dobiju novi zadatak — sa popravcima nedostataka, promenama postojećih karakteristika, pa čak nova funkcionalnost sve u paketu u novu verziju. Slika pokazuje da se nakon samo nekoliko iteracija kvantitativno značajno povećao broj novih postavki, a time i broj testova.



Osnovni koncepti i metodologije

Analiza uticaja novih promena koda kritičan je aspekt u utvrđivanju koje regresijske testove pokrenuti. Ali bez dobre promene skupa unosa, podataka i analize koda, promene mogu biti zavaravajuće. U analizu o tome koji su pravi testovi za automatizaciju treba uključiti celi tim, od poslovanja do razvoja, do tima testiranja i tima za operacije i podršku. Svaka od ovih uloga donosi drugačiju perspektivu o tome gde stvari mogu i gde da krenu po zlu - stoga je važno uključiti ih sve. Neki primeri gde se može primeniti automatizacija testiranja uključuju sledeće:

Data-driven tests:	Testovi u kojima su složeni skupovi podataka koji pokrivaju kritične aspekte
Business logic validation:	Scenariji koji osiguravaju postizanje tačnih rezultata
Integration with a third-party system:	Testovi u kojima programeri i testeri možda nemaju potpuni pristup tom sistemu sa ''treće strane''
Low-intensity performance testing:	Izvođenje malih testova naprezanja, opterećenja, volumena ili korišćenja memorije u svim verzijama kako bi se rano identifikovale degradacije, mnogo pre sprovođenja formalnog testiranja opterećenja
Verifying error scenarios are handled:	Osiguravanje da se aplikacije ponašaju dosledno i ispravno i kada su zavisne od spoljne greške
Installation and upgrade of customer-installed software:	Testiranje komercijalno dostupnog softvera na mnogim platformama i operativnim sistemima
Scanning for security vulnerabilities:	Važni testovi kada bi finansijski ili lični podaci mogli biti ugroženi

Osnovni koncepti i metodologije

TESTIRANJE METODAMA CRNE KUTIJE:

Funkcionalno testiranje, poznato takođe pod imenom testiranje metodama crne kutije, jeste oblik testiranja softvera gde unutrašnja struktura, dizajn i implementacija softvera ili softverske jedinice koja se testira nisu poznati testeru. Softver se posmatra kao funkcija koja mapira vrednosti iz ulaznog domena sistema u izlazni. Kod ove metode vidljivi su samo ulazi i izlazi, a funkcionalnost je određena promatranjem dobijenih izlaznih podataka na temelju odgovarajućih poznatih ulaza. Prilikom testiranja, na osnovu različitih ulaznih podataka dobijeni izlazni podaci se upoređuju s unapred očekivanim te se na taj način vrši vrednovanje ispravnosti programa.

Pristup crne kutije je metoda testiranja u kojoj se dolazi do testnih podataka na temelju specificiranih funkcijskih zahteva, ne uzimajući u obzir konačnu strukturu programa. Ovo testiranje naziva se još i testiranje upravljano podacima (engl. data driven), ulazno/izlazno testiranje (engl. input/output driven) ili testiranje zasnovano na zahtevima (engl. requirements based).

Pošto implementacija i struktura programa nisu poznati, program se posmatra kao crna kutija (odatle potiče i ime). Jedini izvor informacija koja se koristi za određivanje i dizajn testova jeste specifikacija zahteva programa.

Osnovni koncepti i metodologije

TESTIRANJE METODAMA CRNE KUTIJE:

Klase ekvivalencije

Podela ulaznih podataka na klase ekvivalencije je jedna od osnovnih metoda crne kutije. Može se primeniti na bilo kom nivou testiranja, i najčešće je dobar izbor odakle početi testiranje. Glavna ideja kod ove tehnike je da se ulazni domen podataka podeli na takav način da se program ponaša na isti način za sve ulazne vrednosti koje pripadaju istoj klasi ekvivalencije. Drugim rečima, sistem na isti način obrađuje sve vrednosti koje pripadaju istoj klasi ekvivalencije.

Osnovna pretpostavka od koje se polazi jeste da se skup koji predstavlja domen ulaznih podataka može podeliti u disjunktne podskupove, koji se nazivaju klase ekvivalencije. Da bi se obezbedila kompletnost, ceo skup ulaznih podataka se može predstaviti unijom svih podskupova. Pošto se pretpostavlja da će softver na isti način tretirati sve vrednosti iz iste klase ekvivalencije, dovoljno je iz svake klase testirati jednu reprezentativnu vrednost. Ukoliko se softver ponaša korektno pri testiranju jedne reprezentativne vrednosti, moguće je pretpostaviti da će i sve druge vrednosti iz te klase ekvivalencije softver obrađivati bez grešaka, pa nema previše smisla dodatno ih testirati, čime se izbegava redundansa.

Osnovni koncepti i metodologije

TESTIRANJE METODAMA BELE KUTIJE:

Strukturno testiranje (testiranje vođeno logikom (engl. logic driven testing) ili testiranje zasnovano na dizajnu (engl. design-based testing)), još poznato i kao testiranje metodama bele ili staklene kutije, je tehnika testiranja u kojoj je stvarna implementacija softvera poznata testerima. Dok se u tehnikama crne kutije tester fokusira na to šta sistem radi, u metodama bele kutije fokus je na tome kako sistem radi.

Cilj testiranja je da se izvrše sve programske strukture, kao i strukture podataka u programu. Specifikacija se ne proverava prilikom primene metoda bele kutije, već se posmatra samo kod. Strukturno testiranje se može primeniti na svim nivoima testiranja (jedinično, integraciono, sistemsko), i najčešće se primenjuje nakon metoda crne kutije, koje su bazirane na specifikaciji i verifikuju funkciju sistema.

Sami programeri često koriste strukturno testiranje na jediničnom i integracionom nivou, pošto se očekuje da su komponente koje su razvili temeljno testirane pre nego što se obavi dalja integracija tih komponenti u sistem i isporuči testerima na testiranje. Metode bele kutije su značajne i zbog toga što pomažu u određivanju mere temeljnosti testiranja, kroz procenu pokrivenosti programskog koda.

Osnovni koncepti i metodologije

TESTIRANJE METODAMA BELE KUTIJE - Pokrivanje iskaza

Pokrivanje iskaza je najosnovnija metoda bele kutije. Glavna ideja koja stoji iza ove tehnike je da nije moguće znati da li je neki iskaz ispravan ili u njemu postoji greška ukoliko se on ne izvrši. Prema tome, cilj je napraviti testove na takav način da se svaki iskaz (naredba) programa izvrši bar jedanput. Nakon testiranja se mogu identifikovati svi izvršeni iskazi, kao i oni koji nisu izvršeni zbog nekog problema, poput mrtvog koda, nekorišćenih grana i slično.

Pokrivenost iskaza je najlakše demonstrirati na konkretnom primeru. Posmatra se prosta Java metoda, koja štampa zbir dva broja, s tim što ukoliko je zbir negativan, dodaje reč negativan ispred rezultata, a ukoliko je zbir pozitivan, dodaje reč pozitivan ispred rezultata.

```
public void stampajSumu (int a, int b) {
	int rezultat = a + b;
	if (rezultat >  0)
		System.out.println ("Pozitivan " + rezultat);
	else
		System.out.println ("Negativan " + rezultat);
}
```

```
public void stampajSumu (int a, int b) {
	int rezultat = a + b;
	if (rezultat >  0)
		System.out.println ("Pozitivan " + rezultat);
	else
		System.out.println ("Negativan " + rezultat);
}
```

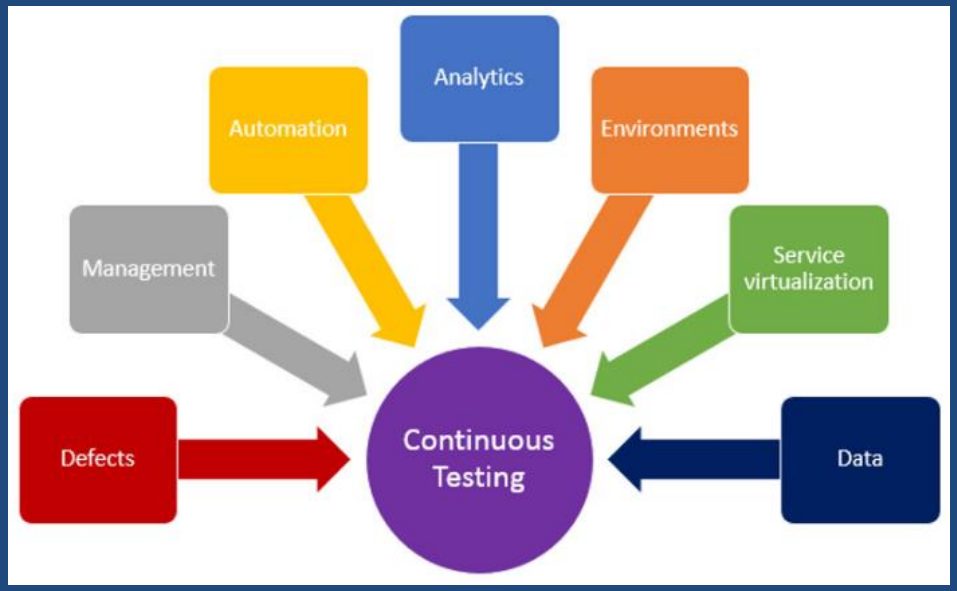
Neka je a = 3 i b = 9. Ukupan broj iskaza u kodu je 7, a testom je izvršeno 5, pa je prema opisanoj jednačini pokrivenost iskaza 5/7 = 0.71, odnosno 71%. Očigledno je da sa jednim testom nisu pokriveni svi iskazi, pošto else grana nije izvršena nijedanput.

Neka su sada a = -3 i b = -9. Drugi test izvršava 6 iskaza, pa je njegova pokrivenost 6/7 = 0.85, odnosno 85%. Međutim, zajedno, ova dva testa pokrivaju sve iskaze, pa je pokrivenost iskaza za metodu iz primera 100%.

Osnovni koncepti i metodologije

Elementi kontinualanog testiranja:

Kultura izgradnja kontinualnog testiranja zahteva ljude, vežbu, sredstva, alate i vreme. Pronalaženje prave ravnoteže odluka u svim praksama testiranja ključno je za postizanje kontinualnog testiranja. Ali pre nego počnemo implementirati kod za testiranje, jedna aktivnost koja se često zanemaruje je pregled koda. Sve što će biti uključeno u izgradnju i korišćenje tokom implementacije u okruženje (testiranje ili proizvodnja) treba pregledati tim stručnjaka okupljenih u tu svrhu. Pregledi koda moraju se efikasno sprovoditi i smatrati ih kvalitetnim; inače, tim ih može smatrati gubljenjem vremena. Recenzija procesa treba da potvrdi da novi i modifikovani softver sledi standarde organizacije i pridržava se njihove najbolje prakse.



Osnovni koncepti i metodologije

Upravljanje defektima:

Kada se primjenjuje tradicionalni pristup testiranju, defekt je početni komunikacioni kanal između testera i programera. Testeri misle “Pronašao sam bag!”, a programeri se tada slažu ili ne: slažem se da postoji problem sa kodom. Mnogo puta nedostatak nije problem u samom kodu, već u zahtevima, dizajnu ili čak samom testu. Ponekad problem leži izvan same aplikacije — u testnom okruženju, test podacima, samom testnom skriptu ili nekoj kombinaciji ovih stvari. Imati saradničko okruženje za evidentiranje i praćenje nedostataka je neophodno u svakom životnom ciklusu razvoja softvera, ali upravljanje defektima je samo vrh ledenog brega kada je u pitanju usvajanje pravog skupa pravila iz prakse.

Kada analiziramo svoj nivo truda koji trošimo na upravljanje nedostacima i greškama, postavićemo ssledeća pitanja:

» Trošimo li previše vremena na evidentiranje, trijažiranje ili analizu defekata?
» Šta je sa vremenom koje trošimo na greške koje nisu "stvarne", kao što je nesporazum između samog testa i koda?
» Šta ako bismo mogli sprečiti čitavu gamu defekata od toga da su uopšte stvoreni?

Osnovni koncepti i metodologije

Upravljanje testovima:
Jedna od sledećih praksi koju testni timovi obično usvajaju je upravljanje tetestiranjme, što uključuje sledeće:

» Planiranje testiranja
» Identifikovanje potrebnih testova
» Kreiranje test slučajeva i skripti
» Prikupljanje postojećih testova
» Izvođenje testova
» Identifikovranje suvišnih, duplih testova i testova koji se preklapaju
» Praćenje i izveštavanje o napretku testiranja

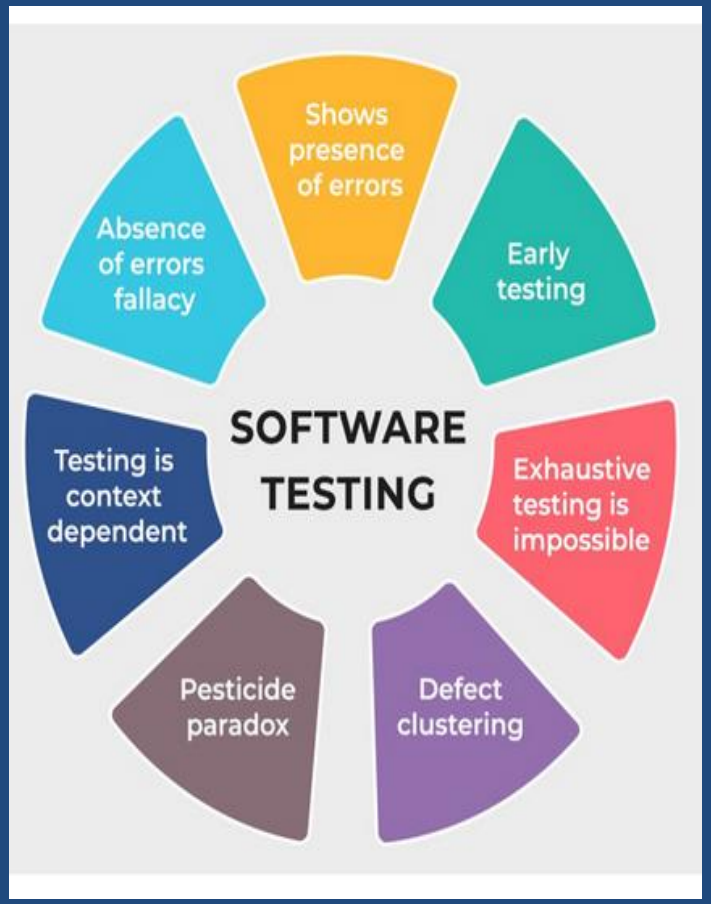
Ova praksa upravljanja testiranjem pomaže i timovima i menadžmentu da na prvi pogled razumeju prolaze li testovi, jesu li uspešni ili su blokirani. Pitanja koja možemo postaviti u ovom području uključuju:

» Trošimo li vreme na ručnu izradu izveštaja o statusu i sakupljanju rezultata izvršenja testa?
» Imamo li alat koji pruža rezultate izvršenja testa u u stvarnom vremenu i omogućuje nam da istražujemo po potrebi?
» Kako znamo jesmo li u okviru rasporeda našeg testa (ili iza, ili čak ispred!)?

Osnovni koncepti i metodologije

Postoje sedam osnovnih principa testiranja:

1. Testiranje pokazuje prisustvo defekata – testiranje može da pokaže da su defekti prisutni, ali ne može da dokaže da u sistemu nema nijednog defekta.
2. Iscrpno testiranje nije moguće – testiranje svih mogućih kombinacija ulaza i preduslova u praksi nije moguće za bilo koji netrivijalni sistem. Radi se analiza rizika.
3. Rano testiranje – kako bi se što pre otkrili defekti, testiranje treba da počne što je pre moguće.
4. Grupisanje defekata (engl. *defect clustering*) – testiranje treba fokusirati proporcionalno broju očekivanih i kasnije pronađenih defekata (gustini defekata) po modulima.
5. Paradoks pesticida – ukoliko se isti testovi ponavljaju u svakoj iteraciji testiranja, na kraju taj isti skup testova više neće moći da pronađe nijedan novi defekt.
6. Testiranje zavisi od konteksta. Testiranje se različito izvršava u različitim kontekstima.
7. Odsustvo grešaka ne garantuje da sistem radi kako treba – pronalaženje i ispravljanje defekata ne pomaže u slučaju da je sistem neupotrebljiv, ili da ne ispunjava želje i očekivanja korisnika.



Osnovni koncepti i metodologije

ALATI ZA TESTIRANJE SOFTVERA:

Pored raznih tehnika i metoda testiranja softvera razvijeni su i mnogi alati koji olakšavaju ili potpuno automatizuju proces testiranja. Gotovo da je nemoguće učiniti proces testiranja jasno definisanim i ponovljivim bez prednosti koje alati za testiranje nude.

Osnovna podela testiranja podrazumeva manuelno i automatsko testiranje. Naravno, manuelno testiranje se ne odigrava u potpunosti ručno. Oba oblika testiranja koriste alate koji omogućavaju efikasnije testiranje.

Razvoj automatskih alata za testiranje je reakcija na napredovanje web baziranih, klijent/server i alata za razvoj softvera koji su omogućili ubrzani razvoj aplikacija sa znatno boljim funkcionalnostima. Sektori za testiranje dolaze u dodir sa softverom koji je jako unapređen, ali i kompleksan. Novi alati za testiranje se razvjaju kao podrška procesu obezbeđenja kvaliteta.

Osnovni koncepti i metodologije

Postoji nekoliko kategorija alata po ciljevima i karakteristikama:

Alati funkcija/regresija	- pomažu u testiranju grafičkog korisničkog interfejsa. Neki pomažu i kod drugih tipova interfejsa. Npr. web test alati koji testiraju kroz pretraživač (capture/replay) alati.
Alati za test dizajn/podatke	- pomažu stvaranju test case-ova i generisanju test podataka.
Load/performance alati	- alati za test opterećenja i performansi.
Alati proces/menadžment	- pomažu organizovanje i izvršavanje paketa testova na nivou komandne linije, API-ja ili protokola.
Unit test alati	- ovi alati, okviri i biblioteke podržavaju unit testiranje, koje se obično izvršava od strane developera, obično korišćenjem interfejsa ispod javnog.
Alati za test implementacije	- pomažu testiranju u vreme izvršavanja.
Alati procene testa	- pomažu procenu kvaliteta testova. Uključuju alate za pokriće koda.
Statistički test analajzeri	- analiziraju programe bez pokretanja istih. Metrički alati spadaju u ovu kategoriju.
Alati za upravljanje defektima	- alati koji prate softverski proizvod i defeke i upravljaju zahtevima za njihovo poboljšanje.
Alati za praćenje performansi/usaglašenosti aplikacije	- mere i maksimiziraju vrednosti u IT životnom ciklusu isporuke, kako bi obezbedili da aplikacije zadovolje nivo kvaliteta, performanse i ciljeve dostupnosti.
Alati runtime analize	- analiziraju programe dok su pokrenuti.

Osnovni koncepti i metodologije

Firma **RationalRose** nudi vrlo razrađeno rešenje automatizacije procesa testiranja u vidu niza alata različitih namena, koji sarađuju međusobno i sa ostalim razvojnim alatima za druge faze razvojnog ciklusa softvera:

NAZIV	DELOVANJE
Rational Administrator	- služi za administraciju repozitorijuma svih artefakata koji se odnose na projekat i aktivnost testiranja, pravljenje, ažuriranje projekata, dodela prava pristupa pojedinim članovima tima itd.
RationalTestManager	- prvenstveno se koristi za organizaciju i upravljanje procesom testiranja, pravljenje test planova, dizajn testova i evaluaciju rezultata testiranja.
Rational Robot	- implementacija testa kroz automatizovano funkcionalno i performansno testiranje zasnovano na test skriptovima, sa funkcijom snimanja korisnikovih akcija nad aplikacijom i autogenerisanja skripta.
Rational TestFactory	- analiziranje strukture aplikacija (prvenstveno GUI, Java, C++, VB) radi podrške implementacije testa kroz automatizovano generisanje test skriptova.
Rational QualityArchitect	- specijalizovan za testiranje komponentnih tehnologija srednjeg sloja, EJB i COM.
Rational ManualTest	- alat za implementaciju manuelnog testa.
Rational SiteCheck	- alat za ispitivanje i kontrolisanje strukture i funkcije web aplikacija.
Rational Purify	- detektuje greške u izvršavanju i curenje (leakage) memorije.
Rational Quantify	- sakupljanje i analiza podataka o performansama
Rational PureCoverage	- podaci o pokrivenosti koda (white box testiranje).

Osnovni koncepti i metodologije

Postoji višestruka korist od upotrebe alata u procesu testiranja:

NAZIV	DELOVANJE
Sistematika procesa:	Alati "vode" kroz određene aktivnosti i na sistematizovan način hijerarhijski organizuju i memorišu sve neophodne artefakte.
Poboljšano upravljanje:	U svakom trenutku moguće je proceniti u kom procentu je testiranje obavljeno i šta je još ostalo da se uradi.
Produktivnost:	Automatizovano testiranje obezbeđuje da se veći broj testova sprovodi u jedinici vremena nego manuelno.
Ponovljivost testa (podrška regresionom testiranju):	kroz formalno implementiranje testova kroz test skriptove obezbeđeno je da se test više puta može ponoviti pod identičnim uslovima.

Osnovni koncepti i metodologije

Dibageri:

- Dibageri daju napredne funkcije kao što su pokretanje programa korak po korak (single-stepping), zaustavljanje ili pauziranje izvođenja programa na takozvanom *breakpointu* tokom određenih zbivanja, a neki čak i mogućnost menjanja programa dok se izvodi. Iste funkcije koje čine debugger korisnim za rješavanje bugova čine ga i pomagalom pri razbijanju softverske zaštite, tj. krakiranju program koji nije besplatan u svrhe da se može koristiti neograničeno bez ikakvog plaćanja ili kupovanja. Korisni su i za testiranje performansi programa. Pojedini dibageri rade samo sa specifičnim programskim jezikom dok drugi mogu raditi sa više njih.
- Većina popularnih dibagera daje samo jednostavni komandnolinijski interfesj (command-line interface - CLI), često iz razloga da maksimiziraju portabilnost i minimaliziraju trošenje sistemskih resursa računara. Ipak, popravljanje grešaka u programu preko grafičkog interfejsa (GUI) dibagera se često smatra jednostavnijim i produktivnijim.

Neki od poznatih debuggera su:

- OllyDbg
- SoftICE
- Java Platform Debugger Architecture
- CodeView
- WinDbg
- Eclipse debugger
- Microsoft Visual Studio Debugger

Osnovni koncepti i metodologije

Dibageri:

Otkrivanje i uklanjanje grešaka u programu (eng. Debugging):

Kada se govori o korektnosti/ispravnosti programa, obično se razlikuju:

- ✓ sintaksna ispravnost programa i
- ✓ semantička ispravnost programa

Sintaksna ispravnost programa podrazumeva da je program zapisan u skladu sa specifikacijom programskog jezika, tj. da su sve naredbe programa ispravne jezičke konstrukcije u odnosu na gramatiku programskog jezika. Tako npr,

```
If n > 0
Then n = n + 1
Else n = n - 1
```

je primer sintaksno neispravne konstrukcije u programskom jeziku VBA, dok je

```
If n > 0
Then n = n + 1
Else n = n - 1
End If
```

sintaksno ispravna naredba programskog jezika VBA.

Osnovni koncepti i metodologije

Dibageri:

Čak i kada je program sintaksno ispravan, to još uvek ne znači da on i radi ono za šta je napisan. U tom slučaju program sadrži greške logičke prirode, tj. programer je tokom pisanja programa pogrešno protumačio značenje (semantiku) pojedinih naredbi koje je napisao.

Otkrivanje i ispravljanje semantičkih grešaka je daleko teže od otkrivanja i ispravljanja sintaksnih grešaka. Popularni engleski naziv za semantičku grešku u programu je *bug* ("buba"). Iako se poreklo tog naziva pripisuje anegdoti o pronađenom insektu kao uzroku kvara jednog od prvih elektronskih računara tokom 40-tih godina XX veka, termin *bug*, u značenju "kvar, nefunkcionisanje", se koristio u engleskom jeziku i pre pojave elektronskih računara, još u XIX veku za opis mehaničkih kvarova.

Značaj otkrivanja i ispravljanja semantičkih grešaka je vremenom doveo do razvoja posebnih softverskih alata čiji je to zadatak. Da bi se olakšao proces razvoja softvera, danas za razne programske jezike postoje integrisana razvojna okruženja (eng. Integrated Development Environment, IDE) koja u sebi objedinjuju alate za:

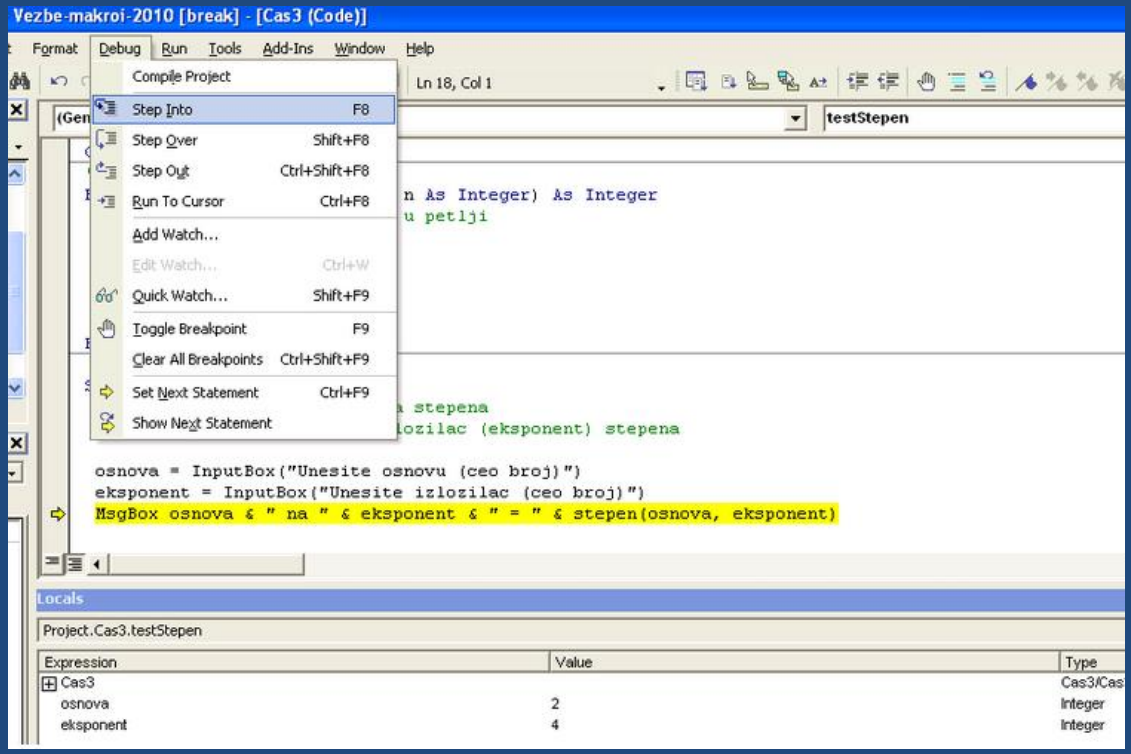
- ✓ unos i uređivanje teksta (naredbi) programa pomoću tastature (eng. text editor)
- ✓ prevođenje/intepretiranje programa
- ✓ otkrivanje i ispravljanje sintaksnih grešaka
- ✓ otkrivanje i ispravljanje semantičkih grešaka (eng. debugger)

Osnovni koncepti i metodologije

Dibageri:

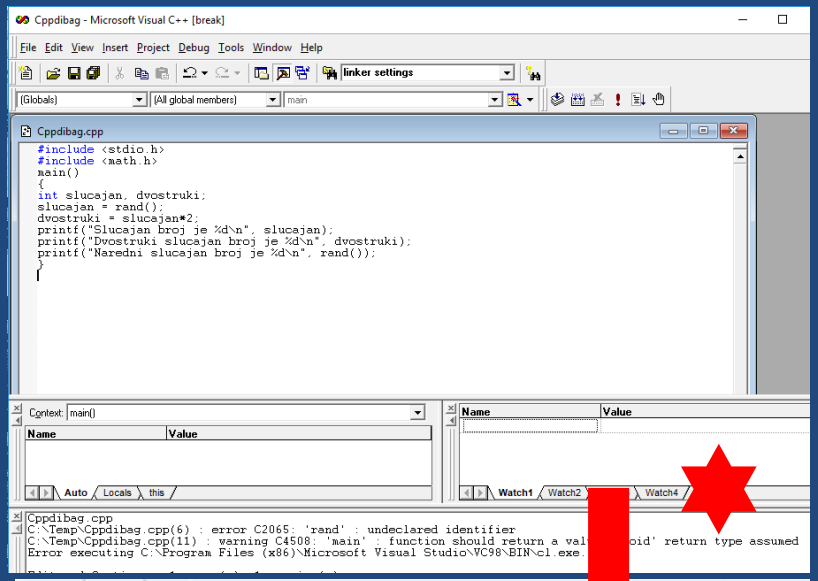
Za programski jezik VBA postoji okruženje (Visual Basic Editor, VBE) koje sadrži ugrađen debugger. Ova opcija testiranja softvera postoji u svim višim programskim jezicima. Jedan od osnovnih načina korišćenja debugger-a je:

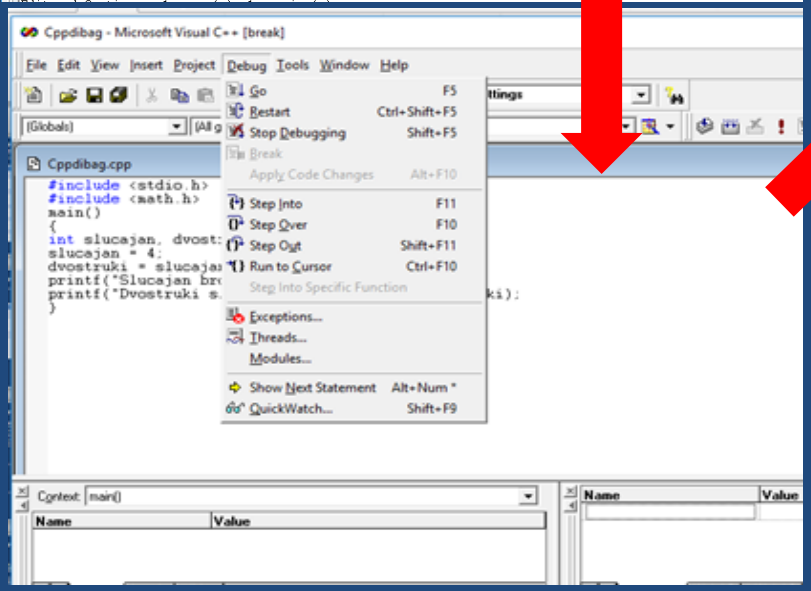
- ✓ Postepeno izvršavanje jedne po jedne naredbe programa (meni Debug > Step Into (F8))
- ✓ Uz praćenje promene vrednosti svih promenljivih tekućeg potprograma (tj. procedure ili funkcije čija se naredba izvršava). Imena i vrednosti promenljivih tekućeg potprograma su dostupne u posebnom prozoru Locals Window ("prozor sa lokalnim promenljivama"), koji postaje vidljiv aktiviranjem istoimene opcije iz menija View.

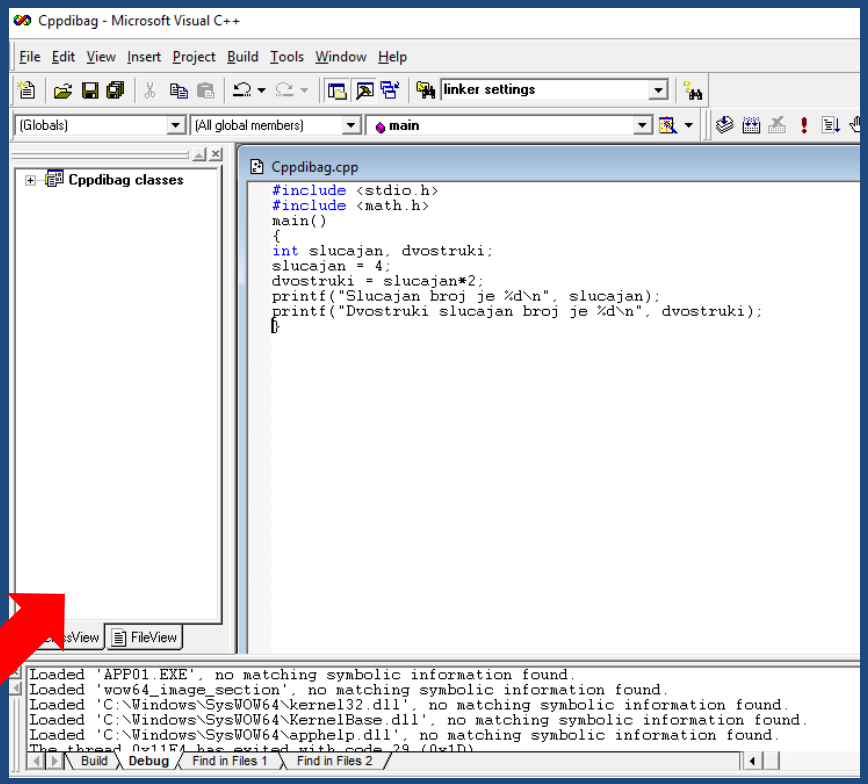


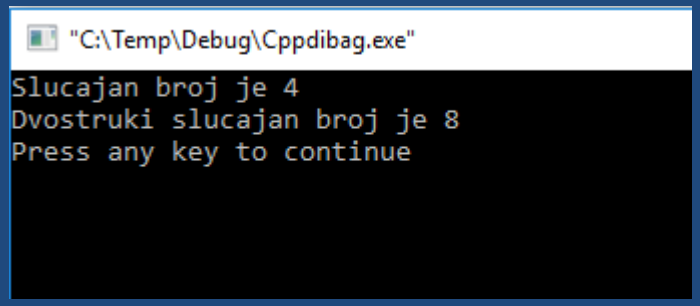
Osnovni koncepti i metodologije

Dibageri: U programskom jeziku C pokretanje dibagera bi izgledalo:







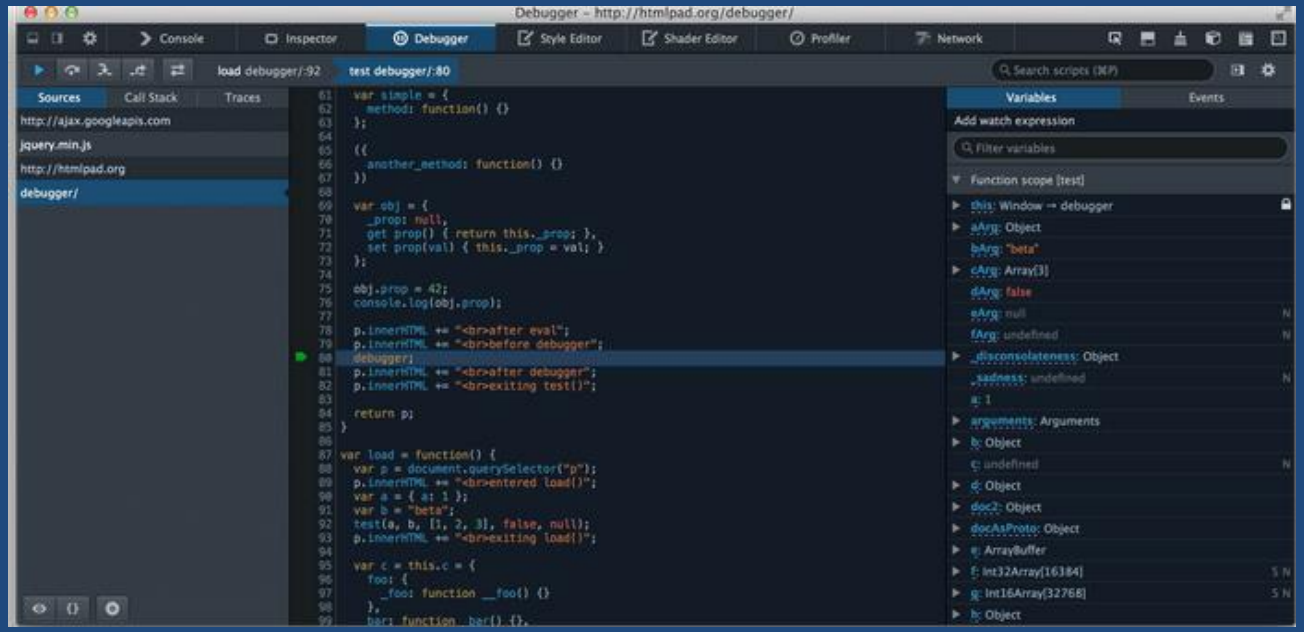


Osnovni koncepti i metodologije

Dibageri:
Nareba za ispravljanje pogrešaka poziva sve dostupne funkcije za otklanjanje pogrešaka, kao što je postavljanje tačke prekida. Ako nije dostupna funkcija za otklanjanje pogrešaka, ova izjava nema učinka. Sledeći primer pokazuje kod u koji je umetnut izraz za ispravljanje grešaka, i za pozivanje programa za ispravljanje grešaka (ako postoji) kada se funkcija pozove. U programskom jeziku **JavaScript** pokretanje dibagera bi izgledalo:

```
function potentiallyBuggyCode()
{ debugger;
// do potentially buggy stuff to examine, step through, etc. }
```

Kada se program za ispravljanje grešaka pozove, izvršenje se pauzira u naredbi za ispravljanje grešaka. To je poput tačke prekida u izvornoj skripti.



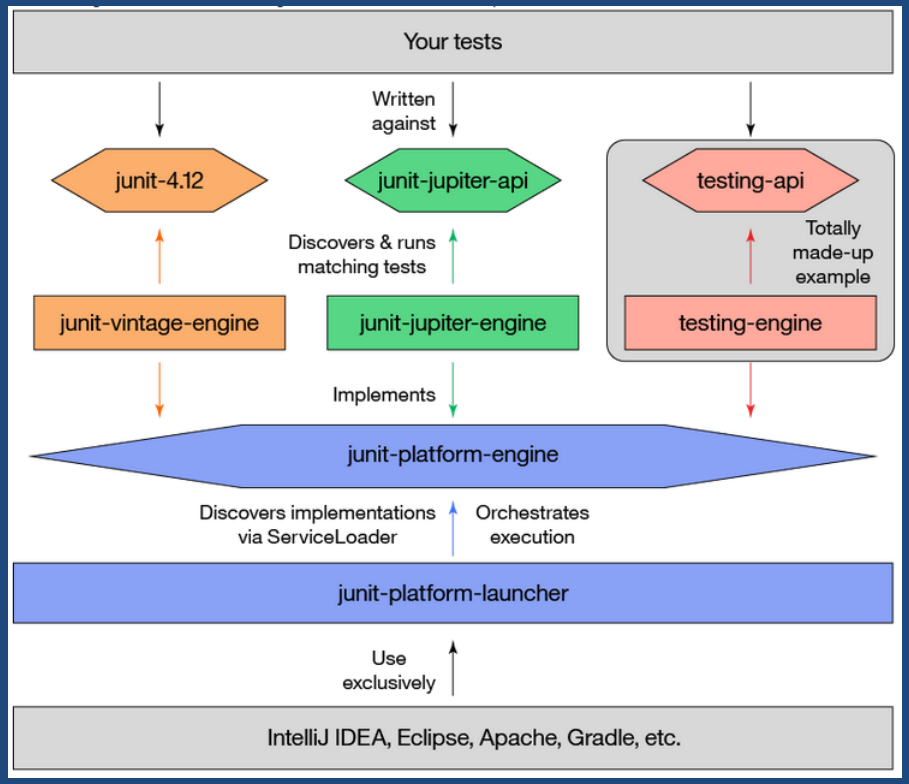
Osnovni koncepti i metodologije

JUNIT FRAMEWORK ALAT:

- JUnit je mali, ali moćan Java *framework* za kreiranje i izvršavanje automatskih jediničnih testova u programskom jeziku Java. Jedinično testiranje podrazumeva da se testira mali deo programa – metoda, klasa ili nekoliko manjih klasa koje čine jednu komponentu i slično, u izolaciji od drugih delova koda. Na ovaj način se brzo može uočiti ukoliko nešto nije u redu sa kodom.

- JUnit se često koristi u klasičnom testiranju, kao sredstvo za automatizaciju testiranja. Sa druge strane, JUnit ima ključnu ulogu u *test-driven* razvoju programa, i promoviše ideju *prvo testiraj pa onda kodiraj*, gde se podrazumeva da programer paralelno sa pisanjem koda piše i jedinične testove.

- Ovaj pristup se zasniva na ideji da se pre programiranja određene jedinice napiše jedinični test koji će verifikovati da li je ta jedinica ispravno programirana. Na ovaj način se programiranje svodi na naizmenične cikluse pisanja jediničnih testova i programiranja samih jedinica, što kao rezultat obično ima povećanje produktivnosti programera i stabilnosti programskog koda, i smanjenje vremena koje se kasnije troši na dibagovanje programa.



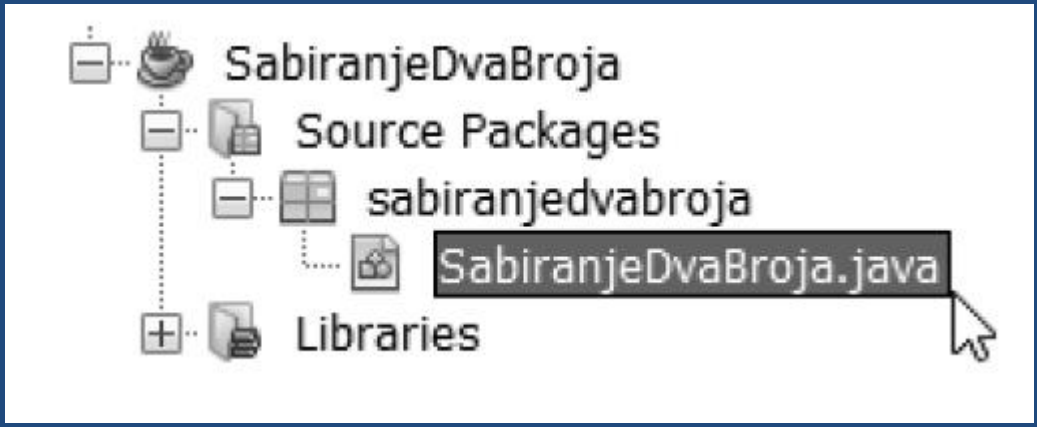
Osnovni koncepti i metodologije

Osnove JUnit alata:

- Osnove JUnit alata najlakše je pokazati na konkretnim primerima. Podrazumeva se da čitalac poznaje osnove programskog jezika Java. Posmatramo klasu SabiranjeDvaBroja, koja ima implementiranu metodu saberi u sledećem obliku:

```
package sabiranjedvabroja;
public class SabiranjeDvaBroja
{public static int saberi (int prvi, int drugi)
{return prvi + drugi;}}
```

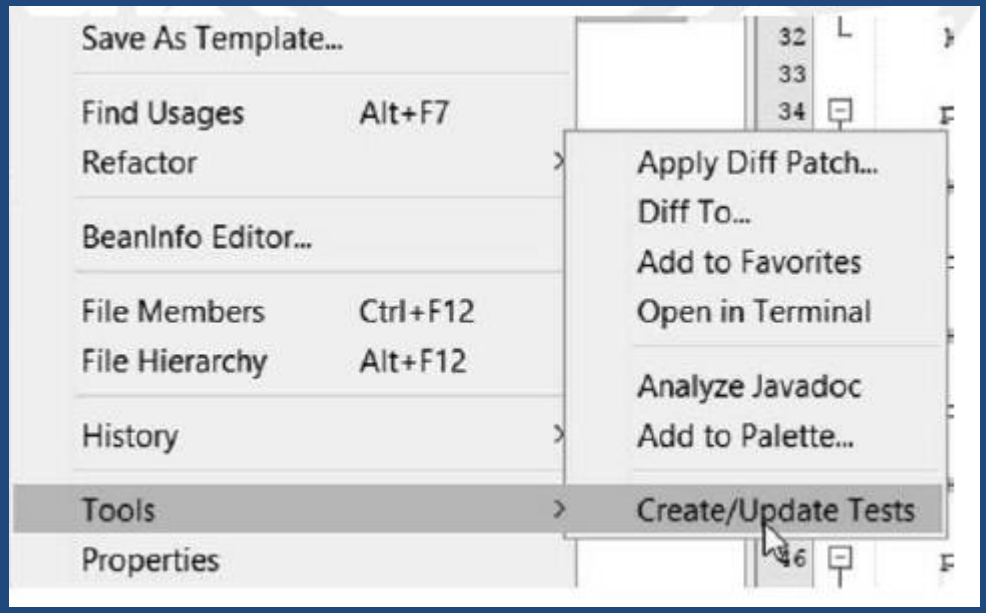
Hijerarhija klasa ovog programa u okviru NetBeans je:



Osnovni koncepti i metodologije

Osnove JUnit alata:

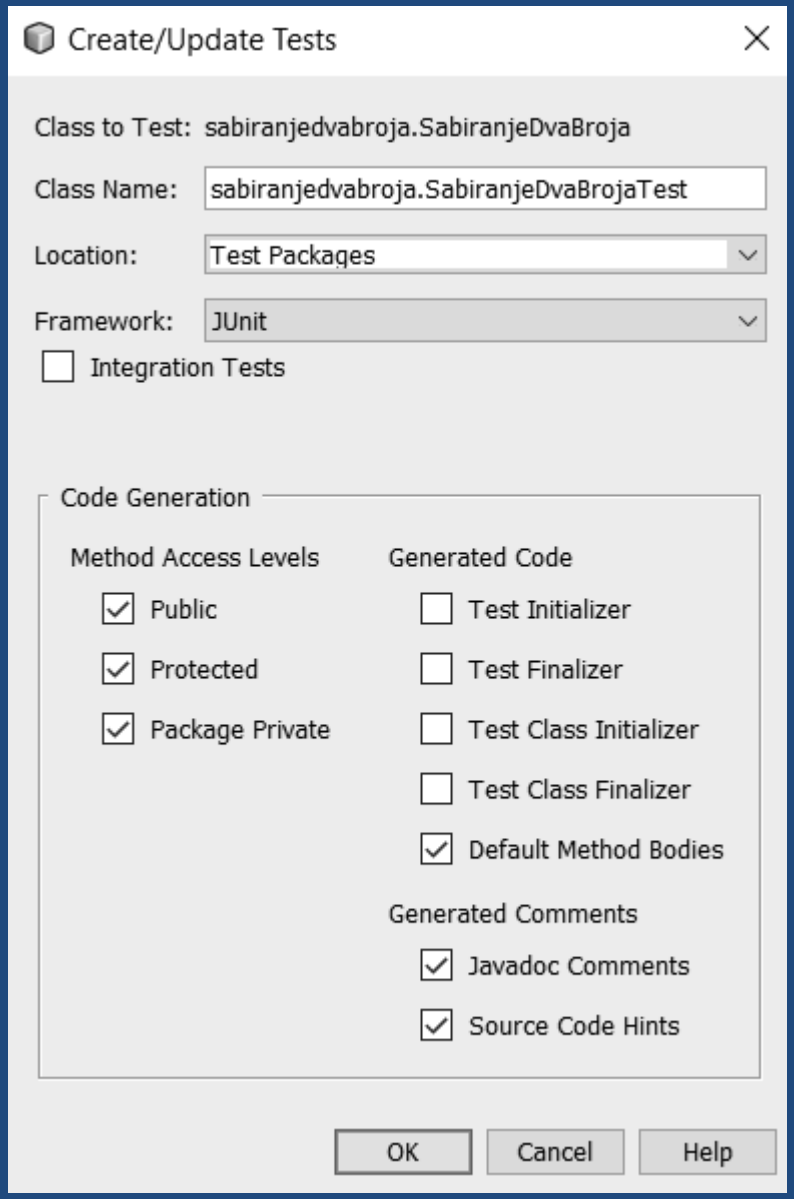
Uočava se klasa za koju je potrebno napisati JUnit test, u ovom slučaju je to klasa SabiranjeDvaBroja.java, koja se nalazi u okviru projekta pod imenom SabiranjeDvaBroja, u *Source Packages*, unutar paketa sabiranjedvabroja. Desnim klikom na ovaj java fajl se otvara meni sa dodatnim opcijama, od kojih je potrebno odabrati opciju Tools, a zatim opciju Create/Update Tests, kao što je prikazano na slici



Osnovni koncepti i metodologije

Osnove JUnit alata:

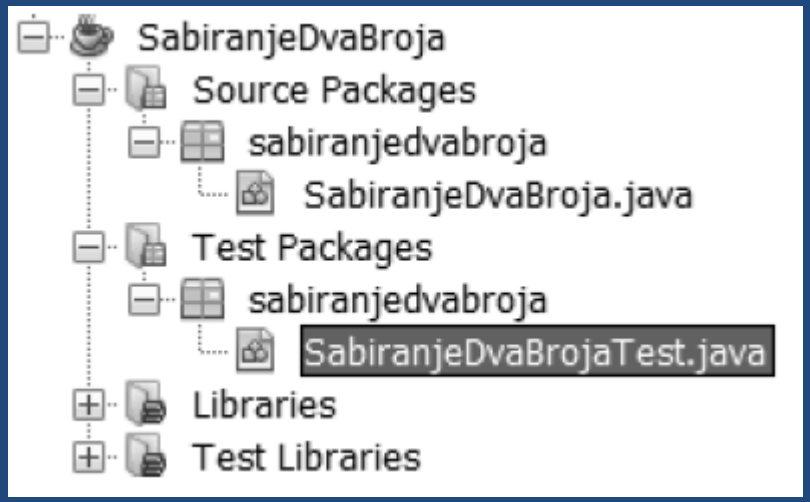
- ✓ Klikom na Create/Update Tests opciju dobija se dijalog sa dodatnim podešavanjima, prikazan na slici.
- ✓ Selekcijom odgovarajućih opcija moguće je automatski dodati inicijalizaciju i finalizaciju testa ili test klase, kao i dodati *default* telo test metoda.
- ✓ Moguće je automatski generisati Javadoc komentare za kasnije kreiranje Javadoc dokumentacije, kao i hintove za izvorni kod.
- ✓ U ovom početnom primeru selektovana je opcija *Default Method Bodies*, koja će automatski za svaku metodu klase koja se testira dodati jedan test u test klasi, sa *default* telom.



Osnovni koncepti i metodologije

Osnove JUnit alata:

- ✓ Klikom na taster OK, generiše se odgovarajuća test klasa, čije podrazumevano ime je ime klase koja se testira sa dodatim Test na kraju, u ovom slučaju SabiranjeDvaBrojaTest.java. Ova klasa se nalazi u automatski generisanom folderu Test Packages, unutar paketa pod istim imenom kao i paket u kome se osnovna klasa nalazi u Source Packages, prikazano na slici.



- ✓ Automatski generisana test klasa je prikazana na donjoj slici. Za svaku metodu klase SaberiDvaBroja su automatski generisani testovi – pošto imamo samo jednu metodu saberi, generisana je jedna metoda testSaberi u test klasi. Automatski kreirano telo metode svakog generisanog testa (ukoliko je pri kreiranju selektovana opcija *Default Method Bodies*) je samo predviđeno da bude vodilja, i mora biti modifikovano kako bi to zaista bio test.

```
package sabiranjedvabroja;

import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author mzivkovic
 */
public class SabiranjeDvaBrojaTest {

    public SabiranjeDvaBrojaTest() {
    }

    /**
     * Test of saberi method, of class SabiranjeDvaBroja.
     */
    @Test
    public void testSaberi() {
        System.out.println("saberi");
```

Osnovni koncepti i metodologije

Osnove JUnit alata:

Na početku test klase može se uočiti import dva paketa:

```
import org.junit.Test;
import static org.junit.Assert.*;
```

Prvi import služi da obezbedi dostupnost klasa JUnit okruženja, poput klase Test. Drugi import služi da se sve provere (engl. *asserts*) pišu bez prefiksa klase, u obliku **assertEquals ()** umesto **Assert.assertEquals ()**. Ove metode su definisane kao statičke u klasi Assert. Sama test klasa koja se piše ne treba da nasleđuje ništa, pošto se koristi Java mehanizam refleksije.

Rezultat dobijen izvršavanjem dela koda koji je predmet testa (u ovom slučaju poziv saberi () metode iz klase SabiranjeDvaBroja) se unutar test metode testSaberi () proverava pozivom jednog od assert metoda definisanih u okruženju JUnit. U ovom primeru, koristi se assertEquals metoda, koja je u JUnit-u definisana na sledeći način:

assertEquals(expected, actual) Ukoliko se vrednost expected (očekivana vrednost) ne poklapa sa actual (vrednost koja se dobija izračunavanjem u kodu koji se testira), izbacuje se izuzetak tipa **java.lang.AssertionError**. Ovaj izuzetak dovodi do prekidanja izvršavanja trenutne test metode, status testa se automatski postavlja na Failed, i nastavlja se sa izvršavanjem ostalih test metoda.

Nakon neophodnih izmena tela metode testSaberi (), ona ima sledeći oblik:

```
@Test
public void testSaberi() {
int prvi = 12; int drugi = 15;
int expResult = 27;
int result = SabiranjeDvaBroja.saberi(prvi, drugi);
assertEquals(expResult, result);}
```

Osnovni koncepti i metodologije

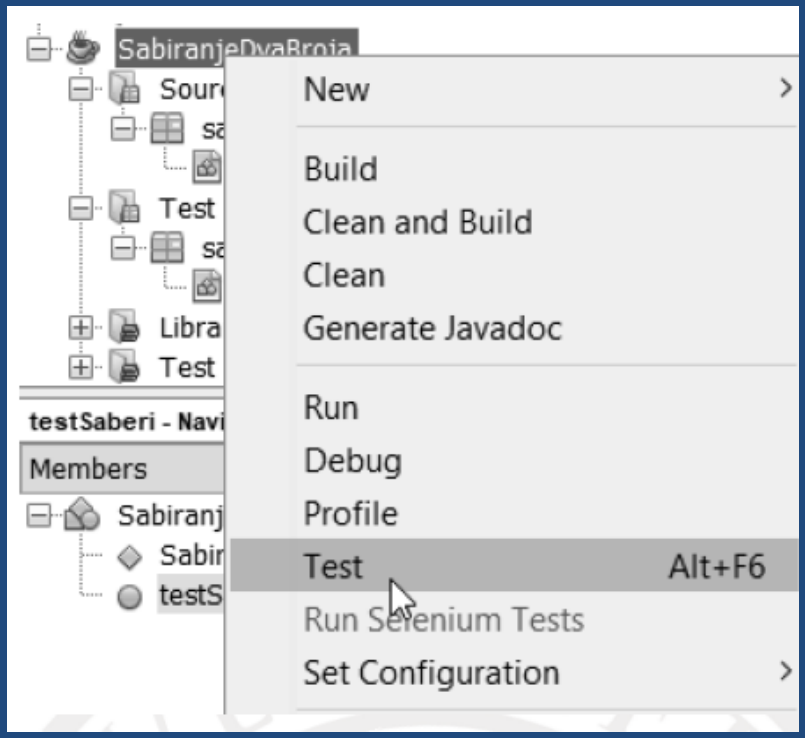
Osnove JUnit alata:

Aktuelni rezultat se dobija pozivom metode koja se testira – metoda saberi iz klase SabiranjeDvaBroja sa definisanim vrednostima za sabirke. Na kraju se, uz pomoć assertEquals metode, proverava da li se očekivani rezultat (prvi parametar metode assertEquals) poklapa sa aktuelnim rezultatom dobijenim pozivom metode koja se testira i izračunavanjem povratne vrednosti (drugi parametar metode assertEquals). Ukoliko se ove dve vrednosti poklapaju, rezultat testa će biti *Passed*, u suprotnom će imati vrednost *Failed*.

Moguće je pisati test i u skraćenom obliku, bez kreiranja privremenih promenljivih
prvi, drugi, expResult i result, u obliku:

```
@Test
public void testSaberi() {
assertEquals(27, SabiranjeDvaBroja.saberi(12, 15));
}
```

Kada su testovi napisani, možemo ih izvršiti desnim klikom na projekat, pa klikom na test (ili uz pomoć skraćenice Alt + F6), kao što je prikazano na slici.



HVALA NA PAŽNJI