Verifikacija softvera

1. Značaj kvaliteta softvera. Procesi i standardi.

IT industrija se brzo razvija i predstavlja jednu od najdinamičnijih industrija u svetu. Visok **kvalitet softvera** je ključni faktor njegove uspešnosti. Upravljanje kvalitetom softvera obuhvata principe, metode i procese koji se koriste za obezbeđivanje i unapređivanje kvaliteta softvera.

Osnovni **procesi** koji su vezani za kvalitet softvera uključuju:

- **planiranje kvaliteta softvera** - definisanje pristupa razvoju softvera koji omogućava postizanje željenog nivoa kvaliteta.
- **obezbeđivanje kvaliteta softvera** - uključivanje aspekata kvaliteta u svakodnevni razvoj softvera.
- **kontrola kvaliteta softvera** - obezbeđivanje da krajnji proizvod bude željenog kvaliteta.
- **poboljšanje kvaliteta softvera** - kontinuirano praćenje i unapređenje procesa razvoja softvera kroz analizu povratnih informacija i primenu novih metodologija razvoja.

Željeni kvalitet softvera definiše se softverskim zahtevima, ali može biti nametnut i različitim međunarodnim **standardima**. Serija standarda **ISO/IEC 25000** sadrži okvir za procenu kvaliteta softvera. Najznačajniji je standard **ISO/IEC 25010** koji definiše devet karakteristika kvaliteta softvera koje obično nazivamo **atributima kvaliteta softvera**. Standard **ISO/IEC 25023** opisuje kako se ove karakteristike kvaliteta koriste za merenje ukupnog kvaliteta proizvoda. Postoje i IEEE standardi koji se koriste, kao na primer **IEEE 730** koji daje smernice za pokretanje, planiranje, kontrolu i izvršavanje procesa obezbeđenja kvaliteta softvera ili **IEEE 1012-2016** koji definiše procese verifikacije i validacije za razvoj sistema, softvera i hardvera.

2. Značaj kvaliteta softvera. Atributi kvaliteta softvera: funkcionalna podobnost, performantnost, kompatibilnost, pouzdanost. Primeri.

(Značaj kvaliteta softvera iz 1. pitanja)

Funkcionalna podobnost je stepen u kojem softver ispunjava funkcionalne zahteve. Obuhvata naredne podatribute:

- **funkcionalna ispravnost** - softver treba da daje tačne rezultate (npr. kalkulator ne greši u sabiranju dva broja).
- **funkcionalna potpunost** - funkcije koje su očekivane specifikacijom su dostupne (npr. kalkulator ima sve predviđene funkcionalnosti kao što su sabiranje, oduzimanje, množenje i deljenje).

- **funkcionalna prikladnost** - softver ispunjava očekivane funkcionalnosti (npr. kalkulator se ponaša na očekivan način i ne treba da ima neočekivane dodatne opcije kao što je puštanje filmova).

Performantnost (efikasnost) je stepen u kojem softver zadovoljava vremenske i resursne zahteve. Obuhvata naredne podatribute:

- **vremensko ponašanje** - vreme odgovora i obrade, stopa protoka (npr. sistem za kupovinu avionskih karata treba da izvrši pretragu karata za manje od 2 sekunde u 97% slučajeva, da prikaže rezultat pretrage u roku od 3 sekunde u 95% slučajeva, obavi plaćanje i potvrdu rezervacije za manje od 5 sekundi u 99% slučajeva), da kašnjenje odgovora eksternih sistema ne bude veće od 1 sekunde i slično).
- **korišćenje resursa** - količina i vrsta korišćenih resursa (npr. kod sistema za kupovinu avionskih karata opterećenje procesora ne sme preći 70% pri normalnom radu, potrošnja RAM memorije ne sme preći 8GB po instanci aplikacije, potrošnja mrežnih resursa ne sme preći 1GB po sekundi na pojedinačnim serverima i slično).
- **kapacitet** - maksimalna ograničenja, u kontekstu broja korisnika ili veličine ulaza (npr. sistem za kupovinu avionskih karata mora da podrži najmanje 10000 istovremenih korisnika bez degradacije performansi).

Kompatibilnost je stepen u kojem softver može da funkcioniše na različitim platformama ili da deli podatke sa drugim proizvodima, sistemima i komponentama. Obuhvata naredne podatribute:

- **koegzistencijabilnost** - sposobnost deljenja okruženja i resursa sa drugim softverskim proizvodima (npr. ako korisnik primi poziv na aplikaciji za razmenu poruka dok sluša muziku, muzika treba da se pauzira, ali i da nastavi nakon poziva, tj. aplikacije ne ometaju jedne druge).
- **interoperabilnost** - sposobnost promene i korišćenja informacija sa/iz drugih aplikacija (npr. aplikacija za razmenu poruka omogućava korisniku da šalje slike iz galerije telefona, ali i sa Google Drive-a).

Pouzdanost je stepen u kojem je softver pouzdan. Obuhvata naredne podatribute:

- **zrelost** - stabilnost tokom svakodnevne upotrebe (npr. sistem za hitne službe mora biti dobro proveren i stabilan i svaka nova funkcionalnost mora biti dobro testirana pre uvođenja).
- **dostupnost** - mogućnost neprekidnog rada (npr. sistem za hitne službe mora imati mogućnost simultanog prijema velikog broja poziva, kao i dodatne infrastrukture u slučaju pada primarnog sistema).
- **tolerancija na greške** - sposobnost funkcionisanja čak i u prisustvu određenih hardverskih ili softverskih kvarova (npr. sistem za hitne službe mora da preusmeri posao ako na nekom serveru dođe do prekida napajanja).

- **sposobnost oporavka** - sposobnost vraćanja podataka i procesa u slučaju kvara (npr. sistem za hitne službe mora imati mogućnost oporavka u slučaju pada).

3. Značaj kvaliteta softvera. Atributi kvaliteta softvera: održivost. Primeri.

(Značaj kvaliteta softvera iz 1. pitanja)

Održivost predstavlja lakoću integrisanja promena u softver. Izmena softvera zahteva razumevanje softvera, pronalaženje delova koje je potrebno izmeniti, izvođenje izmena i proveru da li su izmene poremetile postojeći kod. Održivost se odnosi na lakoću svih ovih koraka. Održivost ne utiče direktno na performanse i ponašanje, ali ima uticaj na dugoročni kvalitet softvera. Metrike softvera koje su važne u kontekstu održivosti su:

- **spregnutost** - kvantitativna mera međuzavisnosti između različitih modula. Poželjna je niska spregnutost.
- **kohezija** - kvantitativna mera povezanosti funkcija ili objekta unutar istog modula. Poželjna je visoka kohezija.
- **ciklomatska složenost** - kvantitativna mera broja linearno nezavisnih putanja kontrole toka programa. Računa se po formuli:

  $$C = E - N + 2P,$$

  gde je E broj grana u grafu kontrole toka programa, N broj čvorova i P broj povezanih komponenti, gde je za većinu pojedinačnih funkcija $P = 1$. Poželjna je niska ciklomatska složenost.
- **veličina** - broj linija koda. Poželjna je manja veličina.

Refaktorisanje je proces poboljšanja dizajna postojećeg koda i predstavlja ključni deo održavanja koda. Održivost obuhvata naredne podatribute:

- **modularnost** - stepen u kom je softver logički podeljen na nezavisne i međusobno zamenljive module. Idealan modul treba da bude relativno male dimenzije, niske ciklomatske složenosti, visoke kohezije i minimalno spregnut sa ostalim modulima. Modularnost podrazumeva i standardizovane interfejse između modula, što je posebno naglašeno u savremenim softverskim arhitekturama kao što su mikroservisi (npr. aplikacija za razmenu poruka mora biti podeljena u različite module, kao što su sistem za slanje poruka, sistem za obaveštenja, sistem za korisničke postavke, sistem za autentifikaciju i tako dalje).
- **iskoristivost** - stepen u kom se komponente sistema mogu koristiti u drugim sistemima. Može se odnositi na specifikacije, dizajn, kod, podatke ili testove. Iskoristivost povećava produktivnost i kvalitet, ubrzava razvoj i smanjuje troškove i rizike u procesu razvoja (npr. poželjno je da aplikacija za razmenu poruka radi na različitim platformama, pa je potrebno komponente koje nisu specifične za platformu, kao što je sistem za prijavu, dizajnirati tako da se mogu koristiti na različitim platformama).

- **analizabilnost** - lakoća analize i razumevanja softvera. Direktno je povezana sa modularnošću i iskoristivošću. Poboljšava se i kroz postojanje dobre dokumentacije i kroz dosledno poštovanje kodnih standarda (npr. aplikacija za razmenu poruka treba da prati aktivnosti korisnika i sistema, kako bi brzo rešavala probleme koji mogu nastati).
- **izmenljivost** - lakoća implementacije izmena bez uvođenja grešaka. Ključni pokazatelj izmenljivosti je spregnutost, jer visoka spregnutost povlači potrebu za izmenama u različitim delovima koda. Dobra modularnost utiče pozitivno na izmenljivost jer smanjuje kompleksnost i ograničava uticaj izmena na lokalne delove koda (npr. aplikacija za razmenu poruka mora biti dizajnirana tako da se lako može nadograditi u skladu sa željama korisnika).
- **testabilnost** - lakoća provere da li su izmene narušile postojeće funkcionalnosti. Visoka testabilnost povlači bržu isporuku i češću povratnu informaciju o ispravnosti. Ključna je automatizacija testiranja, mogućnost automatskog generisanja testova i dostupnost test slučajeva sa unapred poznatim rezultatom izvršavanja (npr. potrebno je testirati nadogradnje koje se implementiraju po željama korisnika i pri tome pokriti sve moguće slučajeve).

4. Značaj kvaliteta softvera. Atributi kvaliteta softvera: upotrebljivost, sigurnost, bezbednost, prenosivost. Primeri.

(Značaj kvaliteta softvera iz 1. pitanja)

Upotrebljivost je stepen u kojem korisnici mogu koristiti softver. Obuhvata naredne podatribute:

- **naučljivost** - koliko je jednostavno naučiti kako softver funkcioniše (npr. korisnici bi trebalo vrlo lako da pronađu i izvrše osnovne operacije pri prvom korišćenju bankarske aplikacije).
- **operabilnost** - koliko je lako raditi sa softverom i kontrolisati ga (npr. osnovne funkcionalnosti bankarske aplikacije treba da budu dostupne u manje od četiri klika).
- **zaštita od korisničkih grešaka** - mere u okviru softvera koje sprečavaju pravljenje grešaka (npr. bankarska aplikacija upozorava korisnika ako unese neispravan broj računa).
- **estetika korisničkog interfejsa** - vizuelna prijatnost i prihvatljivost dizajna (npr. bankarska aplikacija mora imati profesionalan, moderan i intuitivan dizajn).
- **pristupačnost** - mogućnost korišćenja softvera od strane osoba sa različitim sposobnostima (npr. bankarska aplikacija mora biti pristupačna osobama sa oštećenim vidom).
- **prepoznatljivost svrhe aplikacije** - jasnoća namene softvera korisnicima (npr. početni ekran bankarske aplikacije sadrži saldo, dugme za transfer novca i istaknut logo banke).

Sigurnost je stepen u kojem softver štiti informacije i podatke. Obuhvata naredne podatribute:

- **poverljivost** - dostupnost podataka samo ovlašćenim korisnicima (npr. ako korisnik preko bankarske aplikacije pristupi svojoj banci i pregleda stanje na računu, niko drugi ne sme videti njegove informacije, podaci moraju biti šifrovani tokom prenosa i aplikacija ne sme dozvoliti snimanje ekrana).
- **integritet** - sprečavanje neovlašćenog pristupa i modifikacije podataka (npr. niko ne sme da promeni izvršene transakcije i svaka transakcija mora imati digitalni potpis koji potvrđuje da nije menjana).
- **odgovornost** - mogućnost praćenja radnji korisnika (npr. ako se korisnik žali da neko ima neovlašćen pristup njegovom računu, banka mora imati zabeležena sva korišćenja).
- **autentifikabilnost** - mogućnost dokazivanja identiteta korisnika (npr. samo pravi korisnik može pristupiti svom računu, tako što se koriste višefaktorske autentifikacije).
- **neporecivost** - nemogućnost prikupljanja informacija o određenim aktivnostima i događajima (npr. transakcije se digitalno potpisuju kriptografskim ključevima, tako da korisnik ne može tvrditi da nije izvršio transakciju).

Bezbednost je stepen u kojem proizvod, pod definisanim uslovima, izbegava stanje u kojem su ugroženi ljudski život, zdravlje, imovina ili životna sredina. Obuhvata naredne podatribute:

- **operativna ograničenost** - stepen do kog sistem ograničava svoje funkcionisanje unutar bezbednih parametara (npr. automobil pri autonomnoj vožnji mora ograničiti brzinu i manevre u skladu sa vremenskim prilikama i stanjem na putu).
- **identifikacija rizika** - stepen do kog sistem može da identifikuje tok događaja ili operacija koji mogu ugroziti bezbednost (npr. automobil mora da prepozna potencijalne opasnosti, kao što su pešaci koji iznenada prelaze ulicu i vozila koja se kreću nepredviđeno).
- **bezbednost kvara** - stepen do kog sistem može automatski da se postavi u bezbedan režim rada ili da se vrati u bezbedno stanje u slučaju kvara (npr. automobil u slučaju otkaza senzora mora automatski da se zaustavi na bezbedan način).
- **upozorenje na opasnost** - stepen do kog sistem pruža upozorenja o neprihvatljivim rizicima u operacijama ili internim kontrolama, kako bi se omogućila pravovremena reakcija (npr. automobil mora upozoriti vozača ako se pojavi situacija koju sam ne može da reši).
- **bezbednost integracije** - stepen do kog sistem može da održi bezbednost tokom i nakon integracije sa jednom ili više komponenti (npr. automobil mora održavati visok nivo bezbednosti kada se integrišu novi senzori).

Prenosivost je stepen u kom se softver može koristiti u različitim okruženjima. Obuhvata naredne podatribute:

- **prilagodljivost** - mogućnost korišćenja sa različitim hardverom, softverom ili okruženjem (npr. aplikacija za kupovinu avio karata mora da se prilagodi različitim veličinama ekrana, za mobilne uređaje, tablete, desktop računare, kao i za različita tržišta, tj. da ima opcije za različite jezike i valute).

- **instalabilnost** - mogućnost instaliranja i deinstaliranja softvera u različitim okruženjima (npr. aplikacija se na mobilnim uređajima instalira u par klikova u App Store-u ili Google Play-u, a desktop verzija je lako dostupna za Windows, Linux i macOS).
- **zamenljivost** - mogućnost zamene drugim softverskim proizvodom za istu svrhu (npr. aplikacija se može lako zameniti novom verzijom, bez gubitka podataka, pri čemu se lako uklanja stara verzija aplikacije).

5. Uticaj neispravnog softvera. Primeri neprijatnosti i materijalnih gubitaka.

Greške u softveru na različite načine utiču na korisničko iskustvo. **Neprijatnosti** su najblaži oblik posledica, kao što je iznenadno gašenje internet pregledača ili onemogućavanje uspostavljanja poziva u hitnoj situaciji. Drugi nivo uticaja ogleda se u **materijalnim gubicima**. Ove greške se najčešće javljaju u okviru poslovnog softvera i bankarskih sistema. Ovakve greške mogu dovesti i do gubitka poverenja među kupcima. Na kraju, greške u softveru mogu imati i **fatalne posledice**. Javljaju se u kritičnim sistemima, kao što je sistem za upravljanje avionima, automobilima i vozovima. Sličan rizik postoji i u medicinskim uređajima, nuklearnim elektranama, softveru svemirskih letelica i tako dalje. Zbog svega ovoga, razvoj softvera zahteva najviši stepen pažnje, rigorozno testiranje i stalnu proveru kvaliteta.

Primeri neprijatnosti i materijalnih gubitaka:

- Apple je 2012. hteo da zameni aplikaciju Google Maps na iOS-u svojom aplikacijom Apple Maps. Aplikacija je doživela veliku kritiku zbog brojnih grešaka, kao što su pogrešne lokacije gradova, znamenitosti i firmi, netačne rute, izobličene 3D mape i veliki broj nedostajućih informacija. Kao posledica, Apple je preporučio da se koriste konkurentske aplikacije, otpustio nekoliko ključnih ljudi, akcije su opale i došlo je do velikih finansijskih gubitaka.
- Kompanija Knight Capital Group je 2012. prilikom implementacije novog softvera nenamerno aktivirala stari kod, što je dovelo do generisanja velikog broja netačnih naloga, izazivajući kupovinu i prodaju akcija po izrazito nepovoljnim cenama. Reputacija kompanije je bila značajno ugrožena, a ubrzo je i bankrotirala.
- Facebook je 2019. doživeo jedan od najznačajnijih prekida, kada su korisnici širom sveta imali ograničen ili nikakav pristup Facebook-u i povezanim platformama. Kompanija nije pružila mnogo detalja o uzroku greške, zbog čega je bila kritikovana, ali se sumnja da je došlo do greške u internim sistemima za rutiranje podataka. Neki navode da je prekid trajao 14 sati, a neki da je trajao 24 sata, pa su procene materijalnih gubitaka različite.
- Kompanija Google je 2020. imala prekid usluga u trajanju od 47 minuta, što je onemogućilo korisnike širom sveta da koriste Gmail, YouTube i ostale usluge. Prema izveštaju kompanije, greška je nastala u sistemu za upravljanje kvotama skladišnog prostora, koji je nenamerno smanjio kapacitet centralnog sistema za upravljanje identitetima korisnika, pa mnogi zahtevi za autentifikaciju nisu mogli biti obrađeni. Iako je prekid trajao kratko, značajno je uticao na kompaniju.

6. Uticaj neispravnog softvera. Primeri fatalnih posledica.

(Uticaj neispravnog softvera iz 5. pitanja)

Primeri fatalnih posledica:

- Aparat za radioterapiju Therac-25 je između 1985. i 1987. najmanje šest puta dao prejaku dozu radijacije, od čega je troje ljudi umrlo. Aparat je imao dva režima - elektronski snop niskog intenziteta i fotonski snop visoke energije. Greška je nastajala kada bi tehničar izabrao jedan režim, a zatim ga promenio u drugi. Proces koji je ažurirao režim i proces koji je proveravao spremnost uređaja nisu bili dobro sinhronizovani. Dodatno, na aparatu su prethodno uklonjene hardverske sigurnosne zaštite, aparat je imao lošu dijagnostiku i kompanija koja ga je proizvodila je zanemarivala prethodne incidente. Kao rezultat, razvijeni su rigorozniji standardi za sigurnost medicinskih uređaja.
- Tokom Zalivskog rata 1991. godine, iračka raketa pogodila je američku vojnu bazu u Dahranu, pri čemu je usmrtila i povredila veći broj vojnika. Sistem za protivraketnu odbranu je imao odstupanje od približno jedne trećine sekunde nakon 100 sati rada, zbog akumulacije zaokruživanja decimalnih vrednosti u brojevima sa pokretnim zarezom. S obzirom da se balističke rakete jako brzo kreću, ovo je rezultovalo prostornim promašajem od oko 600 metara, pa mehanizam za presretanje nije aktiviran.
- Raketa Ariane 5 uništila se 37 sekundi nakon poletanja 1996. godine, pri čemu su izgubljena četiri satelita i pričinjena velika materijalna šteta. Do greške je došlo kada je sistem pokušao da konvertuje 64-bitnu vrednost brzine u 16-bitnu vrednost, što je izazvalo prekoračenje i izuzetak koji nije adekvatno obrađen. To je dovelo do pogrešnih komandi, skretanja sa putanje i aktivacije sistema za samouništenje.
- Letelica Mars Climate Orbiter nestala je prilikom ulaska u orbitu Marsa 1999. godine, jer je u orbitu ušla značajno brže nego što je planirano. Uzrok je bio propust u konverziji jedinica - deo softvera je koristio imperijalne jedinice, dok je drugi deo softvera očekivao metričke jedinice.
- Greške sa fatalnim posledicama se često dešavaju i u automobilskoj industriji. Kompanija General Motors je grešku otkrila tek nakon saobraćajnih nesreća. Greška je sprečavala aktivaciju prednjih vazdušnih jastuka i zatezača sigurnosnih pojaseva tokom sudara. Sličan problem je imala i kompanija Fiat Chrysler, koja je 2017. opozvala više od milion kamiona širom sveta, s ciljem da preventivno otkloni softverske greške koje su možda bile povezane sa nekim nesrećama.

7. Troškovi usled grešaka u softveru.

Najveći **troškovi** usled grešaka nastaju kada se greške otkriju nakon puštanja sistema u rad. To uključuje gubitke u prihodu, narušavanje reputacije, opozive proizvoda, regulatorne kazne, kao i gubitke podataka ili života. Američki nacionalni institut za standarde i tehnologiju je 2002. napravio studiju po kojoj je neispravan softver koštao američku ekonomiju 59.5 milijardi dolara godišnje. Naglašeno je i da bi rano otkrivanje grešaka moglo da uštedi 22

milijarde dolara godišnje. Konzorcijum za kvalitet informacija i softvera u svom izveštaju za 2022. godinu navodi da su troškovi usled lošeg kvaliteta softvera u Americi porasli na 2410 milijardi. U svom izveštaju navode i da su najveći troškovi proistekli iz sledećih razloga:

- **operativni softverski kvarovi** - greške u funkcionisanju sistema i nepredviđeni zastoji.
- **neuspešni razvojni projekti** - najčešće zbog lošeg upravljanja ili nedostatka kvaliteta.
- **zastareli sistemi** - održavanje i nadogradnja.
- **tehnički dug** - nepopravljive greške i neefikasnosti u kodu, najčešće posledica pravljenja brzih rešenja kada se gubi na kvalitetu jer je potrebno da nešto bude dostupno što pre.
- **sajberkriminal**

Pomenuti izveštaji uzimaju u obzir samo dostupne i javno prijavljene podatke. Kompanije, iz različitih razloga, često smanjuju i ublažavaju izveštaje o problemima koji su nastali usled grešaka u softveru. To rade kako bi zaštitile svoj ugled, izbegle pad poverenja korisnika i investitora ili kako bi izbegle pravne posledice. Greške koje dovedu do curenja podataka, pada sistema ili finansijskih gubitaka se često interno rešavaju i nikada ne dospeju u javnost. Pored toga, mnoge organizacije nemaju razvijen sistem za praćenje i merenje posledica grešaka, pa su podaci u izveštajima gotovo sigurno donji prag stvarne štete. U stvarne troškove se ubrajaju i skriveni gubici, kao što su izgubljene prilike, pad produktivnosti i frustracije korisnika.

8. Odnos verifikacije i validacije softvera. Primeri.

Verifikacija softvera obuhvata sve procese koji su potrebni da se osigura da razvijeni softver zadovoljava zadatu specifikaciju, tj. da ne sadrži defekte i da bude ispravan. Odgovara na pitanje "Da li je softver ispravno izgrađen?". Odnosi se na sve vrste provera ispravnosti rada aplikacije, uključujući i testiranje aplikacije u realnim uslovima i na različitim uređajima. **Validacija** softvera obuhvata sve procese koji su potrebni da se osigura da razvijeni softver zadovoljava korisničke potrebe. Odgovara na pitanje "Da li je izgrađen pravi softver?". Procenjuje da li aplikacija zaista ispunjava očekivanja korisnika. Aplikacija može biti tehnički potpuno ispravna, ali ako ne nudi jasnu vrednost ili ako je interfejs zbunjujući, korisnik je verovatno više neće koristiti i veoma brzo i lako će preći na konkurentsku aplikaciju.

Neka je specifikacija nekog sistema definisana sa "Aplikacija treba da prikazuje vreme". Moguće je napraviti dva različita rešenja:

- Digitalni časovnik. Verifikacija obuhvata proveru da li aplikacija ispravno prikazuje trenutno lokalno vreme, da li je vreme usklađeno sa vremenom operativnog sistema, da li je prikaz ispravan pri promeni vremenske zone, da li se vreme automatski osvežava i slično.

- Aplikacija za vremensku prognozu. Verifikacija obuhvata proveru tačnosti i konzistentnosti podataka koji su preuzeti sa odgovarajućih vremenskih servisa, da li se podaci prikazuju u odgovarajućem formatu, da li su podaci ispravni za različite lokacije i slično.

Validacija obuhvata proveru da li je razvijena aplikacija koju je korisnik želeo. Jasno je da ispravna aplikacija za vremensku prognozu nema vrednost ako je korisnik želeo digitalni časovnik.

9. Tehnike verifikacije softvera. Osnovna podela. Mogućnosti dinamičkog i statičkog pristupa verifikaciji softvera.

Tehnike verifikacije softvera su:

- **dinamička verifikacija** - ispitivanje ispravnosti softvera u toku njegovog izvršavanja. Najčešći vid je **testiranje** - izvršavanje programa sa ciljem da se pronađe što više mogućih defekata ili da se stekne dovoljno poverenja u sistem. Pravilno i sistematičko testiranje podiže nivo pouzdanosti i smanjuje verovatnoću da greške promaknu. Moderni pristupi razvoju softvera podrazumevaju da je testiranje prisutno u svakoj fazi razvoja softvera. U okviru dinamičke verifikacije, koriste se i alati za **debagovanje** i razne vrste **profajlera**.
- **statička verifikacija** - ispitivanje ispravnosti softvera bez njegovog izvršavanja, tj. analiza izvornog koda programa. Postoje različite vrste - provere i pregledi koda koje rade ljudi i **formalne metode verifikacije** softvera. Kod formalnih metoda, uslovi ispravnosti se iskazuju u terminima matematičkih tvrđenja na striktno definisanom formalnom jeziku izabrane matematičke teorije. **Formalna semantika programskog jezika** koristi matematičke modele i formalne tehnike za precizno definisanje značenja programa. Kroz formalnu semantiku moguće je modelovati izvršavanje programa nezavisno od konkretne mašine ili kompajlera, dokazivati korektnost softvera i razvijati alate za formalnu verifikaciju. S obzirom da je halting problem neodlučiv, nije moguće napraviti alat koji bi automatski analizirao kod i dao precizne informacije o njegovoj ispravnosti. Međutim, ako se odreknemo potpune preciznosti, možemo da napravimo program koji automatski, u konačnom vremenu, koristeći konačne resurse može da da veoma korisne informacije o ispravnosti programa. Preciznost može biti narušena kroz **lažna upozorenja**, tj. alat prijavljuje problem koji u realnom izvršavanju ne može da se desi, ili kroz **propuštene greške**, tj. alat ne prijavljuje problem koji u realnom izvršavanju može da se desi. Alati obično teže da imaju samo jedan od ova dva problema, a ne oba istovremeno. Formalne metode verifikacije obuhvataju i tehnike razvoja ispravnog softvera.

10. Testiranje u razvoju softvera. Cena greške u kontekstu vremena otkrivanja.

Softver nastaje kao odgovor na jasno definisane zahteve korisnika, koji određuje šta i na koji način softver treba da radi, u kojim uslovima i koje potrebe korisnika treba da zadovolji. Svako odstupanje softvera od ovih zahteva smatra se **defektom**. Oni su posledica ljudskih grešaka napravljenih u različitim fazama razvoja softvera - od analize i dizajna, preko implementacije, pa sve do testiranja i održavanja. **Testiranje** softvera predstavlja sistematsku aktivnost u okviru životnog ciklusa razvoja, koja ima za cilj da otkrije defekte pre nego što softver dospe u ruke krajnjih korisnika. Pomaže i u proceni pouzdanosti, performansi i stabilnosti softvera.

U savremenom razvoju softvera, osnovni cilj je postizanje maksimalnog profita kroz isporuku proizvoda visokog kvaliteta, uz poštovanje vremenskih i budžetskih ograničenja, pa su metode i pristupi koji omogućavaju bržu, efikasniju i pouzdaniju isporuku softvera sve važniji. Najpopularnije metodologije, kao što su agilne metodologije, zasnivaju se na iterativnom pristupu koji podrazumeva paralelno razvijanje i testiranje softverskih komponenti. Greške koje promaknu u ranim fazama razvoja mogu da izazovu lančane probleme, koji ne samo da komplikuju završne faze projekta, već mogu dovesti i do potpunog neuspeha proizvoda. Ispravljanje grešaka u početnim fazama razvoja je znatno brže, jednostavnije i jeftinije nego otklanjanje grešaka u kasnijim fazama:

- **faza analize zahteva** - greška zahteva reviziju definisanih zahteva i trošak uključuje dodatno vreme analitičara.
- **faza dizajna softvera** - greška zahteva izmenu dizajna i trošak uključuje dodatni rad dizajnera i ažuriranje opisa implementacije.
- **faza implementacije softvera** - greške programeri uglavnom brzo isprave, a cena zavisi od složenosti greške. Posebno je bitna **faza integracije komponenti**, u kojoj se greške teže otkrivaju jer uključuju više različitih delova softvera. Ispravka u tom slučaju često zahteva više članova tima ili više različitih timova.
- **faza sistemskog testiranja** - cena greške uključuje rad tima zaduženog za kvalitet softvera, prijavu greške, komunikaciju sa programerima, ponovno testiranje i praćenje kroz sistem za praćenje defekata. Ukoliko postoji korak **testiranja prihvatljivosti**, cena dodatno obuhvata i rad korisnika (kupca) koji vrši proveru da li je softver prihvatljiv i dodatno produžava završne faze projekta.
- **faza upotrebe** - greške u ovoj fazi imaju najveću cenu, jer pored tehničke ispravke dolazi i do gubitka poverenja korisnika.

11. Testiranje u razvoju softvera. Uloga testera u razvoju softvera.

(Testiranje u razvoju softvera iz 10. pitanja)

Projekti u oblasti razvoja softvera mogu se voditi na različite načine, pa je briga o kvalitetu softvera nekad odgovornost celog razvojnog tima, a nekad postoji poseban tim zadužen za tu oblast. U manjim kompanijama ovakav tim često ne postoji, dok kod većih kompanija postoji interni tim zadužen za kvalitet ili se takav tim angažuje eksterno od strane specijalizovane firme koja pruža usluge testiranja i kontrole kvaliteta. **Tester softvera** je stručnjak koji se bavi planiranjem, izvođenjem i dokumentovanjem aktivnosti testiranja softvera, sa ciljem da obezbedi očekivani kvalitet proizvoda. Njegov osnovni zadatak je sistematsko ispitivanje softvera kako bi se identifikovali defekti u funkcionisanju, performansama ili upotrebljivosti. Raspodela obaveza između programera i testera može biti organizovana na više načina:

- Programeri su istovremeno i testeri, tj. sami proveravaju svoj kod.
- Programeri i testeri rade zajedno u istom timu, blisko sarađujući.
- Programeri i testeri čine potpuno odvojene timove sa jasnim razgraničenjem odgovornosti.

Između programera i testera često nastaju problemi u komunikaciji. Pronalaženje greške testeru označava da je uspešno obavio svoj zadatak, dok programeru to znači da nije uspešno obavio svoj posao. Da bi se osoba bavila testiranjem potrebno je da poseduje osnovno razumevanje programiranja i procesa razvoja softvera. Dodatno, mora detaljno da poznaje procedure i procese testiranja, alate koji se koriste u testiranju, kao i skript programiranje.

12. Testiranje u razvoju softvera. Faze testiranja softvera: planiranje, analiza, dizajn i implementacija testova.

(Testiranje u razvoju softvera iz 10. pitanja)

Da bi se započelo testiranje nije potrebno postojanje koda, već specifikacija zahteva sistema i specifikacija zahteva softvera. Testiranje se u opštem slučaju sastoji od sledećih faza:

1. Planiranje testiranja
2. Analiza, dizajn i implementacija testova
3. Izvršavanje testova
4. Evaluacija testova
5. Zatvaranje testiranja

Planiranje testiranja je priprema za proces testiranja. Plan testiranja se prilagođava konkretnom projektu. Planiranje započinje analizom zahteva, pri čemu se razmatraju svi funkcionalni i nefunkcionalni aspekti sistema. Na osnovu analize definišu se **ciljevi testiranja**. Nakon toga, određuje se **opseg testiranja**, tj. koje komponente i koliko će biti

testirane, kao i **način testiranja**, tj. koji delovi testiranja će biti sprovedeni ručno, koji automatizovano i koji alati će se koristiti. Planiranje uključuje i **definisanje uloga i odgovornosti** članova tima. Vrši se i **procena rizika**, koja uključuje identifikaciju potencijalnih problema i njihov uticaj na kvalitet proizvoda. Planira se i vremenski okvir testiranja, uz jasno definisane **ulazne** i **izlazne kriterijume** koji određuju kada testiranje može da počne i pod kojim uslovima se smatra završenim. Neizostavan deo plana je i **metodologija za praćenje defekata**, koja definiše kako se greške prijavljuju, dokumentuju, prate tokom životnog ciklusa i na kraju zatvaraju. Konačno, planiranje uključuje i **pripremu test okruženja**, tj. obezbeđivanje potrebnog hardverskog i softverskog okruženja neophodnog za efikasno i pouzdano izvođenje testova.

Analiza, dizajn i implementacija testova rezultuju sveobuhvatnim skupom test slučajeva i procedura koji služe kao temelj za uspešno sprovođenje testiranja. **Test slučaj** je formalni dokument koji opisuje određenu situaciju testiranja kroz definisani skup ulaznih podataka i očekivane izlaze sistema. Svaki test slučaj ima za cilj da proveri konkretnu funkcionalnost softvera, njegovo ponašanje u graničnim slučajevima ili otpornost na nevalidne i neočekivane ulaze. **Procedura testiranja** definiše tačan redosled i način primene test slučajeva u kontrolisanom okruženju. **Testni okvir** je skup skripti, alata i konfiguracija koji omogućavaju automatsko pokretanje testova, simulaciju realnog okruženja i prikupljanje rezultata. **Analiza testova** podrazumeva detaljno ispitivanje mogućnosti testiranja delova softvera i identifikaciju zahteva koje sistem mora da ispuni. Podrazumeva detaljno razlaganje korisničkih priča, slučajeva upotrebe i tehničke dokumentacije, kako bi se identifikovali svi elementi koji mogu i treba da budu predmet testiranja. Na ovaj način se obezbeđuje sveobuhvatna pokrivenost svih relevantnih aspekata sistema. **Dizajn testova** podrazumeva definisanje ciljeva testiranja, identifikaciju funkcionalnosti koje treba proveriti i preciziranje uslova pod kojim će testovi biti sprovedeni. Identifikovani scenariji se razrađuju u test slučajeve i prateću dokumentaciju, čime se obezbeđuje jasan i sistematičan pristup. Obuhvata kreiranje i prioritizaciju test slučajeva, identifikaciju potrebnih test podataka i preciziranje testnog okruženja. Cilj faze je da se obezbedi dosledna i kvalitetna osnova za kasniju implementaciju i izvršavanje testova. **Implementacija testova** podrazumeva konkretizaciju dizajniranih test slučajeva, tako što se oni organizuju u procedure testiranja i pripremaju se potrebne testne dokumentacije, alati i okruženja. Ključne aktivnosti koje se sprovode su:

- **razvoj i prioritizacija procedura testiranja i razvoj test okvira** - omogućava strukturisano i automatizovano izvršavanje test slučajeva.
- **formiranje test kompleta (test suites)** grupisanjem povezanih procedura i skripti, što povećava efikasnost tokom izvršavanja.
- **raspoređivanje test kompleta u plan izvršavanja**
- **priprema test podataka i provera njihove integracije u okruženje**

13. Testiranje u razvoju softvera. Faze testiranja softvera: izvršavanje i evaluacija testova. Zatvaranje testiranja.

(Testiranje u razvoju softvera iz 10. pitanja)
(Faze testiranja softvera iz 12. pitanja)

Izvršavanje testova je faza u kojoj se praktično primenjuju test slučajevi i test procedure razvijene tokom prethodnih faza. Testovi se mogu sprovoditi ručno ili automatski, korišćenjem posebnih alata. Neophodna je organizacija testiranja kako bi se testovi izvršavali u skladu sa prioritetima i bez gubljenja resursa i vremena. Testovi se sistematski sprovode kako bi se otkrile greške, ali i kako bi se pratio status prethodno prijavljenih problema. Svaka korekcija greške zahteva ponovno izvršavanje relevantnih testova kako bi se osiguralo da popravka nije dovela do novih problema. Ova faza zahteva intenzivnu i jasnu komunikaciju između testera i programera, što omogućava bržu identifikaciju uzroka grešaka i efikasnije rešavanje problema.

Evaluacija testova obuhvata procenu kriterijuma završetka testiranja i izveštavanje. Svaka izmena može da dovede do novih grešaka, pa se definiše kriterijum završetka testiranja u odnosu na rezultate izvršavanja testova, procenat nerešenih bagova ili preostalo vreme za testiranje. Izlazni kriterijumi određuje da li je testiranje kompletirano i da li je aplikacija spremna za korišćenje u skladu sa korisničkim zahtevima. Da bi se odredilo da li su oni ispunjeni, u obzir se uzimaju rezultati testiranja dati kroz sažeti izveštaj testiranja, rezultate raznih metrika i izveštaj o defektima.

Zatvaranje testiranja nastupa kada su svi planirani testovi sprovedeni i softver je isporučen korisniku, bez daljih obaveza održavanja. Ovakva situacija je u praksi veoma retka, jer se softver nakon isporuke gotovo uvek održava, što podrazumeva i testiranje. Testiranje se može zatvoriti i ako je projekat otkazan, ako su ciljevi testiranja ispunjeni ranije nego što je planirano ili ako nema više smisla nastavljati testiranje zbog promene poslovnih prioriteta. Tokom ove faze vrši se arhiviranje test slučajeva, izveštaja i prateće dokumentacije i sprovodi se analiza procesa testiranja, uz identifikaciju praksi koje su se uspešno pokazale i praksi koje nisu bile uspešne.

14. Vrste testiranja. Testiranje jedinica koda. Primeri.

Na osnovu prirode osobina koje se testiraju, testiranje može biti:

- **testiranje funkcionalnih karakteristika** - provera da li aplikacija pravilno izvršava funkcije definisane specifikacijom zahteva.
- **testiranje nefunkcionalnih zahteva** - usmereno na performanse, sigurnost, upotrebljivost, kompatibilnost, pouzdanost i ostale tehničke i kvalitativne aspekte sistema.

Na osnovu nivoa testiranja, testiranje može biti:

- testiranje jedinica koda

- komponentno testiranje
- integraciono testiranje
- sistemsko testiranje

Na svakom nivou mogu se testirati funkcionalne i nefunkcionalne karakteristike softvera.

Testiranje jedinica koda (unit testing) predstavlja proveru ispravnosti najmanjih delova softvera koji se mogu nezavisno testirati. U OOP to su najčešće klase, u funkcionalnom programiranju funkcije, a u imperativnom programiranju procedure. Omogućava detaljan uvid u ponašanje svake pojedinačne jedinice, čime osigurava stabilnu osnovu za kasniju integraciju u veće sisteme. Jedna od najvećih prednosti testiranja jedinica koda je snažna podrška u alatima za automatsko izvršavanje i proveru rezultata testova, koji su često integrisani u savremena razvojna okruženja. Kada jedinica koda komunicira sa spoljnim resursima (npr. mreža ili baza), ta komunikacija se u test okruženju zamenjuje fiksiranim (**mock**) vrednostima, a isto važi i za komunikaciju sa drugim klasama ili komponentama. Ideja je da se sve apstrahuje kako bi se test fokusirao isključivo na ponašanje posmatrane jedinice. Ove testove najčešće pišu sami programeri tokom razvoja, što omogućava brzo otkrivanje i otklanjanje grešaka. Na primer, ako imamo funkciju koja sabira dva broja, tada možemo napisati testove koji će proveriti tačnost funkcije. Možemo proveravati sabiranje pozitivnih brojeva, sabiranje negativnih brojeva, sabiranje pozitivnih i negativnih brojeva i tako dalje.

15. Vrste testiranja. Komponentno i integraciono testiranje. Primeri.

(Vrste testiranja iz 14. pitanja)

Komponentno testiranje (component testing) proverava ispravnost komponente. **Komponenta** je skup funkcionalno povezanih jedinica koda koje zajedno obavljaju jednu logičku celinu i imaju jasno definisan interfejs. Obično se sprovodi izolovano od ostatka sistema i to odmah nakon razvoja komponente. Fokusira se na više jedinica koda i njihovu međusobnu saradnju. S obzirom da se integrišu osnovne jedinice koda, ovo je vrsta integracionog testiranja, na najnižem nivou. U praksi, mogu ga obavljati i programeri i testeri.

Integraciono testiranja (integration testing) predstavlja fazu u procesu testiranja u kojoj se proverava saradnja između više softverskih komponenti koje zajedno čine funkcionalne celine sistema. Ispituje ispravnost interfejsa i komunikacije među komponentama. Cilj je da se utvrdi da li su veze između komponenti pravilno definisane i implementirane, pa se proverava funkcionalna ispravnost u kontekstu saradnje. Mogu otkriti razne vrste problema, kao što su nekompatibilni podaci pri razmeni između modula, neusklađeni formati, tipovi ili protokoli komunikacije, pogrešno tumačenje interfejsa ili očekivanih vrednosti i greške u redosledu izvršavanja aktivnosti. Ove testove obično pišu testeri, iako je u praksi česta saradnja sa programerima. Dobra praksa je da se integraciono testiranje sprovodi inkrementalno - kako se nove komponente razvijaju i dodaju sistemu, tako se istovremeno proverava njihova integracija. Na primer, dve softverske komponente vrše računanja sa realnim brojevima, ali zbog nedovoljno precizne specifikacije jedna koristi realne brojeve

jednostruke tačnosti, dok druga koristi realne brojeve dvostruke tačnosti. Iako svaka komponenta zasebno funkcioniše ispravno u okviru svojih testova, njihova integracija dovodi do neočekivanih odstupanja u rezultatima.

16. Vrste testiranja. Sistemsko testiranje. Funkcionalno sistemsko testiranje. Regresiono testiranje. Primeri.

(Vrste testiranja iz 14. pitanja)

Sistemsko testiranje (system testing) obuhvata proveravanje sistema kao celine i ispituje da li je ponašanje sistema u skladu sa specifikacijom zadatom od strane klijenta. Ovde se zahteva potpun pristup svim delovima sistema, uključujući bazu, pristup mreži i svim hardverskim delovima sistema. Uključuje i funkcionalne i nefunkcionalne aspekte sistema, kao i istraživačko testiranje, testiranje prihvatiljvosti i instalaciono testiranje.

Funkcionalno sistemsko testiranje predstavlja proveru funkcionalnosti sistema u uslovima koji odgovaraju stvarnom korišćenju. Cilj je da se potvrdi da su sve očekivane funkcionalnosti sistema implementirane (potpunost), da svaka funkcionalnost pojedinačno radi u skladu sa zahtevima korisnika (ispravnost), kao i da ne postoje neprikladne funkcionalnosti sistema (prikladnost). Obuhvata testiranje ulaza, obrada i izlaza za različite funkcionalne scenarije, uključujući tipične i granične slučajeve upotrebe, kao i ponašanje u slučaju pogrešnih ili neočekivanih ulaza. Uspešno funkcionalno testiranje potvrđuje da sistem zadovoljava svoje osnovne zahteve i da je spreman za naredne faze verifikacije. Na primer, test za slučaj kada korisnik nema dovoljno novca za prenos sredstava u aplikaciji za banku. Preduslovi su da je korisnik prijavljen, da ima validan račun u sistemu i da na računu ima manje sredstava nego što želi da prenese. Test podaci su broj računa i iznos za prenos, a očekivani rezultat je da sistem prikazuje poruku o grešci, ne izvršava prenos i da je stanje na računu nepromenjeno.

Regresiono testiranje je proces ponovnog izvršavanja prethodno uspešno izvršenih test slučajeva s ciljem da se proveri da li novouvedene izmene u softveru nisu nenamerno dovele do regresije, tj. narušile postojeću funkcionalnost sistema ili dovele do pada performansi. Sprovodi se uvek nakon ispravki grešaka, dodavanja novih funkcionalnosti i nakon refaktorisanja. Može se sprovoditi na svakom od nivoa.

17. Vrste testiranja. Sistemsko testiranje. Istraživačko testiranje. Testovi prihvatljivosti. Instalaciono testiranje. Primeri.

(Vrste testiranja iz 14. pitanja)
(Sistemsko testiranje iz 16. pitanja)

Istraživačko testiranje podrazumeva otkrivanje neočekivanih pravaca korišćenja softvera i potencijalnih problema koji nisu bili identifikovani tokom analize i dizajna testova. Tester intuitivno istražuje aplikaciju, posmatra njeno ponašanje u različitim situacijama i na osnovu toga formuliše nove test slučajeve u realnom vremenu. Oslanja se na znanje, intuiciju i kreativnost testera. Ova vrsta testiranja ima najveći značaj kada je softver već zaokružen, tj.

kada je aplikacija dostupna u gotovo finalnom obliku. Ako se zanemari, postoji rizik da određene funkcionalnosti ostanu neproverene, pa se ova vrsta testiranja koristi kao dopuna formalnim tehnikama. Moguće je testirati i funkcionalna i nefunkcionalna svojstva sistema. Na primer, tester istražuje neočekivane tokove korišćenja u aplikaciji za deljenje fotografija koja omogućava postavljanje, komentarisanje i deljenje fotografija. Proverava pokušaj postavljanja slike u formatu koji aplikacija formalno ne podržava, brzo višestruko postavljanje iste fotografije, pokušaj učitavanja sadržaja bez pristupa internetu i promenu jezičkih podešavanja tokom aktivne sesije. Zahtev je da se u svakom od pomenutih slučajeva aplikacija ponaša stabilno, obezbedi korisniku poruke o greškama i izbegne pad sistema ili gubitak podataka.

Testovi prihvatljivosti omogućavaju korisnicima i klijentima da se sami uvere da je napravljeni softver u skladu sa njihovim potrebama i očekivanjima. Testiranje izvode i procenjuju korisnici, a razvojni tim samo pruža pomoć oko tehničkih pitanja. Obično spada u tehnike validacije softvera. Klijent može da proceni sistem na tri načina:

- **referentno testiranje** - procenjuje se da li je softver implementiran u skladu sa očekivanjima. Klijent generiše test slučajeve koji predstavljaju uobičajene uslove u kojima sistem treba da radi.
- **pilot testiranje** - instalacija sistema na privremenoj lokaciji i njegova upotreba. Testiranje se vrši simulacijom svakodnevnog rada na sistemu.
- **paralelno testiranje** - koristi se tokom razvoja, kada jedna verzija softvera zamenjuje drugu ili kada novi sistem treba da zameni stari. Ideja je paralelno funkcionisanje oba sistema, čime se korisnici postepeno privikavaju i prelaze na korišćenje novog sistema.

Instalaciono testiranje predstavlja specifičnu vrstu testiranja u kojoj se softver instalira u klijentskom okruženju, kako bi se proverila njegova sposobnost da se pravilno instalira i funkcioniše u realnim uslovima. Sistem se podešava u skladu sa tehničkim karakteristikama ciljne mašine, kao i specifičnostima sistema. Testira se i sposobnost uspostavljanja komunikacije sa spoljnim uređajima ili servisima, ako softver to zahteva. Testovi se najčešće izvode u saradnji sa krajnjim korisnicima, kako bi se osiguralo da instalacija teče bez grešaka i da sistem funkcioniše nakon instalacije. Prati se i uticaj okruženja na funkcionalne i nefunkcionalne karakteristike sistema.

18. Vrste testiranja. Nefunkcionalno testiranje. Testovi performansi. Testovi kompatibilnosti. Testovi pouzdanosti. Primeri.

(Vrste testiranja iz 14. pitanja)

Nefunkcionalno sistemsko testiranje obuhvata testiranje dinamičkih nefunkcionalnih aspekata sistema, kao što su performanse, pouzdanost, bezbednost, prenosivost, kompatibilnost i slično. Testovi se sprovode za one aspekte kvaliteta koji su prepoznati kao važni za dati sistem.

Testovi performansi proveravaju vremensko ponašanje aplikacije i njenu upotrebu resursa. Fokusira se na metrike kao što su vreme odgovora na zahteve, broj obrađenih zahteva u jedinici vremena, vreme obrade podataka, iskorišćenost memorije i slično. Čine ih:

- **testovi kapaciteta** - proveravaju ponašanje sistema pri obradi velikih količina podataka. Pažnja se posvećuje granicama sistema, tj. kako on funkcioniše kada se približi maksimalnim kapacitetima definisanim u zahtevima.
- **stres testovi** - proveravaju kako se sistem ponaša kada je izložen zahtevima van njegovog projektovanog kapaciteta, kao i kako se on oporavlja kada se opterećenje smanji.
- **testovi konfiguracije** - proveravaju rad sistema u različitim hardverskim i softverskim okruženjima. Na taj način se obezbeđuje da aplikacija funkcioniše stabilno bez obzira na konkretne kombinacije OS, drajvera, baza ili uređaja.

Na primer, želimo da testiramo ponašanje bankarske aplikacije kada veliki broj korisnika istovremeno koristi aplikaciju, kao kada je kraj meseca ili isplata plata. Pokrenemo veliki broj korisničkih sesija (npr. 10000) i izvršavamo standardne operacije kao što su prijava, pregled stanja, prenos novca i slično. Pratimo prosečno i maksimalno vreme odziva za ključne operacije, iskorišćenost resursa kao što su CPU, memorija i baza, broj uspešno izvršenih zahteva u sekundi, kao i broj neuspešnih ili odbijenih zahteva. Zahtevi su da sistem mora da podrži 8000 istovremenih korisnika sa vremenom odziva manjim od 2 sekunde, pri čemu maksimalni gubitak zahteva ne sme preći 0.01%.

Testovi kompatibilnosti proveravaju način na koji sistem komunicira sa spoljnim komponentama i sistemima. Uključuju ispitivanje koegzistencije (mogućnost rada zajedno sa drugim softverima na istom sistemu) i interoperabilnosti (sposobnost razmene podataka i funkcionalne saradnje sa drugim sistemima). Cilj je da se osigura da sistem može uspešno da funkcioniše u širem okruženju bez konflikata ili nekompatibilnosti. Na primer, želimo da proverimo da li bankarska aplikacija može da funkcioniše ispravno kada se izvršava paralelno sa drugim softverom instaliranim na korisnikovom uređaju, kao što su antivirus softver i VPN klijent. Pratimo broj zabeleženih konflikata, performanse i broj funkcionalnosti koje su otežano dostupne ili nedostupne. Zahtevi su da ne sme doći do prekida funkcionalnosti, gubitka podataka ili bezbednosnih problema, a da je maksimalan broj dozvoljenih vizuelnih anomalija jedan.

Testovi pouzdanosti proveravaju stabilnost softvera i sposobnost da funkcioniše bez grešaka tokom određenog vremenskog perioda, u realnim ili simuliranim uslovima rada, kao i sposobnost da se oporavi nakon što dođe do greške, bilo automatski ili uz minimalnu intervenciju korisnika. Na primer, testiramo da li bankarska aplikacija može da radi 72h bez prekida, tokom kojih se konstantno generišu korisnički zahtevi, uz periodične simulacije grešaka i poremećaja u mrežnom okruženju. Pratimo broj padova ili prekida rada, vremensko trajanje oporavka posle greške i prisustvo curenja memorije ili degradacije performansi. Zahtevi su da ne sme doći ni do jednog sistemskog pada, da svaki oporavak mora biti kraći od 60 sekundi i da memorija ne sme rasti progresivno bez oslobađanja.

19. Vrste testiranja. Nefunkcionalno testiranje. Testovi upotrebljivosti. Testovi bezbednosti. Testovi sigurnosti. Testovi prenosivosti. Primeri.

(Vrste testiranja iz 14. pitanja)
(Nefunkcionalno testiranje iz 18. pitanja)

Testovi upotrebljivosti proveravaju koliko je softver lak za upotrebu krajnjim korisnicima. Ova vrsta testiranja je posebno važna za aplikacije sa korisničkim interfejsom, jer ima direktan uticaj na korisničko iskustvo i prihvatanje proizvoda. Proverava se koliko brzo novi korisnik može da nauči da koristi sistem bez pomoći, da li korisnik može da izvrši zadatke u prihvatljivom broju koraka, da li sistem omogućava korisniku da lako prepozna i ispravi greške, da li su elementi grafičkog korisničkog interfejsa intuitivno raspoređeni, da li korisnici subjektivno ocenjuju sistem kao prijatan za korišćenje, da li je upotrebljiv za osobe sa različitim oblicima invaliditeta i slično. Testiranje se obično sprovodi kroz posmatranje korisnika dok izvršavaju tipične zadatke, uz praćenje vremena, broja grešaka i nivoa frustracije. Koriste se i upitnici i intervjui nakon testiranja kako bi se saznao subjektivni utisak korisnika. Na primer, želimo da testiramo koliko brzo novi korisnik može da izvrši transfer sredstava koristeći bankarsku aplikaciju. Pratimo vreme koje je potrebno, broj pokušaja i grešaka, broj pitanja koja korisnik postavi i subjektivnu ocenu korisnika. Zahtev je da korisnik obavi transakciju bez pomoći za manje od 5 minuta, sa najviše jednom greškom.

Testovi bezbednosti proveravaju zaštitu ljudi, opreme i okruženja od neželjenih efekata korišćenja softvera. Posebno je važno u kritičnim sistemima, kao što su medicinski uređaji, automobilski sistemi, vazduhoplovstvo i slično. Na primer, želimo da proverimo kako sistem autonomne vožnje reaguje u slučaju kada pri brzini od 50km/h na put iznenada ulazi pešak na udaljenosti od 15 metara. Pratimo vreme reakcije sistema, efikasnost kočenja i aktivaciju dodatnih sistema, kao što su upozorenja. Zahtev je da se vozilo mora zaustaviti pre kontakta sa pešakom u 99% slučajeva.

Testovi sigurnosti proveravaju da li su određene funkcionalnosti sistema dostupne isključivo onim korisnicima kojima su namenjene. Proveravaju se dostupnost, integritet i poverljivost podataka. Na primer, želimo da testiramo da li je bankarska aplikacija otporna na pokušaj neautorizovanog pristupa putem napada grubom silom na formu za logovanje. Zahtev je da aplikacija mora da blokira pristup nakon najviše 3 neuspešna pokušaja i onemogući dalju automatizovanu verifikaciju lozinki, pri čemu napad mora biti zabeležen u sistemskim logovima.

Testovi prenosivosti proveravaju sposobnost softvera da funkcioniše u različitim okruženjima. Cilj je da se potvrdi da softer može lako da se instalira, koristi i radi ispravno na više platformi, OS, hardverskih konfiguracija ili pregledača, kao i da može da se u potpunosti ukloni sa sistema bez ostavljanja tragova ili neželjenih podataka. Na primer, želimo da testiramo da li bankarska aplikacija funkcioniše dosledno na različitim OS i verzijama uređaja. Instaliramo i pokrećemo aplikaciju na svakom test uređaju i OS, prijavljujemo korisnika i proveravamo prikaz početnog ekrana i menija, kao i vizuelnu usklađenost. Zahtev

je da aplikacija mora nesmetano da radi na svim platformama, bez bilo kakvih padova ili vizuelnih nepravilnosti.

20. Tehnike testiranja. Karakteristike dobrog skupa testova. Pokrivenost testiranjem. Podela na tehnike testiranja.

Tehnike testiranja imaju za cilj da pruže sistematski odgovor na pitanje kako identifikovati reprezentativni skup test slučajeva, odnosno testova. **Reprezentativni skup testova** treba da obuhvati slučajeve koji najbolje odražavaju stvarne uslove rada softverskog sistema, ali i one koji imaju povećanu verovatnoću da otkriju potencijalne slabosti u implementaciji. Dodatno, treba da omogući efikasno balansiranje između obima testiranja i potrošnje resursa.

Dobro definisan skup testova predstavlja osnovu za kvalitetno testiranje koje doprinosi visokom kvalitetu softvera. Reprezentativni skup testova treba da ima naredne karakteristike:

- **visok potencijal otkrivanja grešaka**
- **relativno mala veličina**
- **relativno velika brzina izvršavanja**
- **pruža visok stepen poverenja u pouzdanost softvera**

Pokrivenost testiranjem (test coverage) je metrika koja pomaže da se proceni koliko su testovi sveobuhvatni i koliko dobro proveravaju funkcionalnost, strukturu i ponašanje sistema. Koristi se za procenu kvaliteta softvera i pomaže u odluci da li je softver spreman za isporuku. Definiše se kao procenat elemenata sistema koji su obuhvaćeni testiranjem u odnosu na sve elemente te vrste. Postoje razne vrste pokrivenosti:

- **pokrivenost zahteva** - pokazuje koliki procenat zahteva je proveren testiranjem u odnosu na ukupan broj zahteva korisnika koji su definisani kroz specifikaciju. Potpuna pokrivenost podrazumeva da je svaki zahtev testiran.
- **pokrivenost funkcionalnosti** - pokazuje koliki procenat funkcionalnosti je proveren testiranjem u odnosu na ukupan broj funkcionalnosti sistema. Potpuna pokrivenost podrazumeva da je svaka funkcionalnost testirana.
- **pokrivenost koda** - obuhvata više vrsta pokrivenosti, kao što je pokrivenost naredbi - broj izvršenih naredbi u okviru testiranja podeljen sa ukupnim brojem naredbi u projektu.

Tehnike testiranja se dele na:

- **testiranje crne kutije** - generisanje test primera bez razmatranja interne strukture koda, već isključivo na osnovu specifikacije. Fokusira se na ponašanje sistema iz korisničkog ugla. Prednost je mogućnost potpunog razdvajanja programera i testera, pa ga obično obavljaju testeri. Oni sistemu pružaju ulaze, a zatim proveravaju izlaze u odnosu na specifikaciju.

- **testiranje bele kutije** - generisanje test primera na osnovu interne strukture i logike izvornog koda, pri čemu se posebno proučava tok izvršavanja programa. Obično ga obavljaju programeri tokom faze razvoja softvera, npr. kroz testiranje jedinica koda. Skuplje je i zahtevnije od testiranja crne kutije zbog potrebe za dubinskim uvidom u rad sistema, pa se najčešće primenjuje kada je potrebna visoka pouzdanost i gde su greške posebno skupe.
- **testiranje sive kutije** - prelaz između tehnika crne i bele kutije. Postoji uvid u unutrašnju strukturu sistema, ali ne u istoj meri kao kod tehnika bele kutije. Vrše ga i programeri i testeri, na primer kroz komponentno i integraciono testiranje.

21. Tehnike testiranja. Testiranje metodama crne kutije. Isprobavanje svih mogućih ulaza. Metod klasa ekvivalencije. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Testiranje metodama crne kutije iz 20. pitanja)

Testiranje crne kutije teorijski se može uraditi **isprobavanjem svih mogućih ulaza**, ali već za neke trivijalne programe nije moguće koristiti ovu tehniku. Na primer, kvadratna jednačina $ax^2 + bx + c = 0$ ima dva rešenja, pa ako su koeficijenti tipa int32 onda je broj različitih test primera $2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{96}$. Dodatno, potrebno je razmotriti i moguće neispravne ulaze kojih može biti mnogo. Na primer, neispravan ulaz za starost osobe može biti negativan broj ili neka proizvoljna reč. Dakle, test slučajevi se mogu podeliti u dve osnovne kategorije - test slučajevi koji odgovaraju validnim ulazima i test slučajevi koji odgovaraju nevalidnim ulazima.

Testiranje pomoću klasa ekvivalencije je tehnika koja smanjuje broj test slučajeva na prihvatljiv nivo, a da se pri tome zadrži zadovoljavajuća pokrivenost ulaznog prostora podataka. **Klase ekvivalencije** su skupovi ulaznih podataka koji se obrađuju na isti način od strane sistema. Mogu biti **validne** ili **nevalidne**, u zavisnosti od toga da li bi sistem trebalo da prihvati te vrednosti za dalju obradu ili ih odbaci. Na primer, ako se očekuje da korisnik unese broj godina između 18 i 65, validne klase bi pokrivale sve vrednosti unutar tog opsega, a nevalidne klase vrednosti ispod 18 i iznad 65. Pretpostavke ove tehnike su:

- Ako jedan test slučaj iz određene klase ekvivalencije otkrije grešku, velika je verovatnoća da bi i svi ostali test slučajevi iz iste klase otkrili istu tu grešku.
- Ako jedan test slučaj iz određene klase ekvivalencije ne otkrije grešku, pretpostavlja se da ni ostali test slučajevi iz te klase ne bi otkrili grešku.

Na osnovu ovih pretpostavki, testiranje se može racionalizovati izborom po jednog predstavnika iz svake klase. Ključni koraci ove tehnike su:

1. Identifikovati sve validne klase ekvivalencije za ulazne vrednosti.
2. Izabrati po jedan test slučaj za svaku validnu klasu.
3. Identifikovati sve nevalidne klase ekvivalencije za ulazne vrednosti.
4. Izabrati po jedan test slučaj za svaku nevalidnu klasu.

Na primer, sistem za zapošljavanje donosi odluku na osnovu starosti i to tako što ne zapošljava osobe mlađe od 16 ili starije od 55, pri čemu osobe od 16 do 18 može zaposliti sa pola radnog vremena, a osobe od 19 do 55 sa punim radnim vremenom. Validne klase ekvivalencije mogu biti 0-15, 16-17, 18-55, 56-99, a nevalidne klase vrednosti ispod minimalne vrednosti i vrednosti iznad maksimalne vrednosti. Za svaku od klasa biramo po jednog predstavnika i na taj način smanjujemo broj test slučajeva sa oko 100 na svega par test slučajeva.

22. Tehnike testiranja. Testiranje metodama crne kutije. Metod klasa ekvivalencije. Metod graničnih vrednosti. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Testiranje metodama crne kutije iz 20. pitanja)
(Metod klasa ekvivalencije iz 21. pitanja)

Testiranje graničnih vrednosti potiče od ideje da se veliki broj grešaka javlja upravo na granicama definisanih klasa ekvivalencije. Fokusira se na vrednosti neposredno iznad i ispod granica. Kod celobrojnih vrednosti to su uzastopne vrednosti ($n-1$ i $n+1$, ako je n granica), dok kod realnih razlika može biti definisana brojem značajnih decimala. Kada imamo realne brojeve visoke preciznosti ili višedimenzionalne ulaze, identifikacija relevantnih granica i test tačaka zahteva dodatnu pažnju. Ključni koraci ove tehnike su:

1. Identifikovati klase ekvivalencije na osnovu ulaznih podataka.
2. Odrediti tačne granice svake identifikovane klase.
3. Za svaku granicu, definisati sledeće test tačke:
    - tačka na samoj granici.
    - tačka neposredno ispod granice.
    - tačke neposredno iznad granice.

Tačke koje se nalaze ispod i iznad granice mogu pripadati klasama koje su već obuhvaćene testovima, pa treba obratiti pažnju da se testovi ne dupliraju. Na primer, kod sistema za zapošljavanje možemo uzeti iste klase ekvivalencije i za svaku od granica (0, 15, 16, 17, 18, 55, 56 i 99) odrediti vrednost na granici, vrednost ispod granice i vrednost iznad granice. Unija svih slučajeva i predstavnika klasa čine konačni skup za testiranje. Može se desiti da i u tom skupu postoje preklapanja. Testovi koji pripadaju istoj klasi ekvivalencije, a nisu granične vrednosti moguće je ukloniti, bez gubitka pokrivenosti.

23. Tehnike testiranja. Karakteristike dobrog skupa testova. Tabele odlučivanja. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Karakteristike dobrog skupa testova iz 20. pitanja)

Tabele odlučivanja pružaju pregled složenih poslovnih pravila u strukturisanom i lako čitljivom obliku. Koriste se kao sredstvo za analizu i dizajn test scenarija u složenim informacionim sistemima. Organizovane su u redove i kolone. Redovi se dele u dve grupe - prvu grupu čine uslovi nad ulazima, dok drugu grupu čine moguće akcije koje sistem treba da izvrši. Kolone predstavljaju pravila - svaka kolona odgovara jedinstvenoj kombinacija vrednosti uslova i opisuje koje se akcije u tom slučaju preduzimaju. Uslovi mogu biti binarni (npr. da/ne) ili viševrednosni (npr. malo/srednje/veliko). Kada su uslovi binarni iz svakog pravila se obično izvodi jedan test slučaj, a kada su viševrednosni može se izvesti više test slučajeva, u zavisnosti od željenog nivoa pokrivenosti. U slučajevima kada više pravila dovode do iste akcije, bez obzira na vrednost nekog uslova, moguće je izvršiti objedinjavanje pravila. U takvim slučajevima, uslov koji ne utiče na ishod označava se simbolom "--" i naziva **nebitnim**. Test slučajevi izvedeni iz tabele odlučivanja mogu se dodatno kombinovati sa drugim tehnikama testiranja, što povećava preciznost testiranja.

Na primer, bankarski sistem vrši odluku da li će izvršiti uplatu na osnovu podataka o sredstvima na računu i dozvoljenom minusu korisnika. Tabela odlučivanja je:

	Pravilo 1	**Pravilo 2**	**Pravilo 3**
Uslovi			
Dovoljno sredstava na računu	**Da**	**Ne**	**Ne**
Dozvoljen minus	**--**	**Da**	**Ne**
Akcije			
Isplata odobrena	**Da**	**Da**	**Ne**

Prvo pravilo predstavlja slučaj kada korisnik ima dovoljno sredstava na računu, pri čemu nije bitno da li je minus dozvoljen ili ne. Iz drugog pravila se može izvesti slučaj kada korisnik nema dovoljno sredstava, ali mu je dozvoljen minus. U oba slučaja, očekivana akcija je da je isplata odobrena. Treće pravilo označava slučaj kada korisnik nema dovoljno sredstava i minus mu nije dozvoljen, a očekivana akcija je da isplata neće biti odobrena.

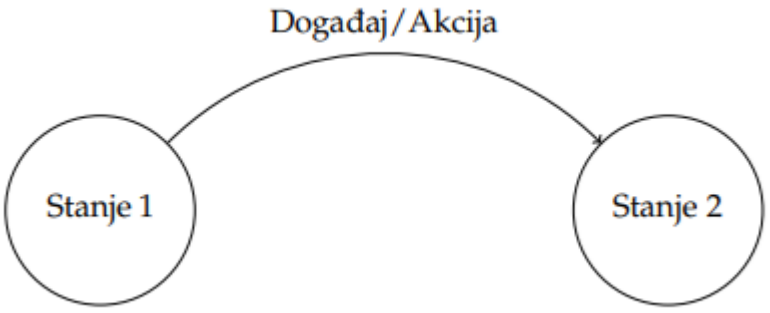
24. Tehnike testiranja. Karakteristike dobrog skupa testova. Dijagrami stanja. Tabele stanja. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Karakteristike dobrog skupa testova iz 20. pitanja)

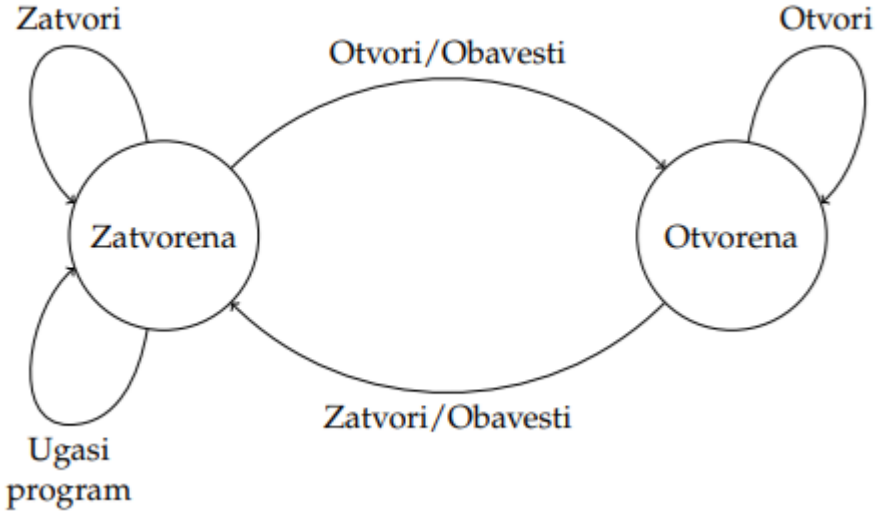
Sistemi koji reaguju na spoljašnje događaje i čije ponašanje zavisi od prethodno izvršenih akcija mogu se efikasno modelovati pomoću **konačnih automata**. U svakom trenutku, takav sistem se nalazi u jednom od konačno mnogo mogućih stanja i pasivno čeka neki ulazni događaj. Kada se događaj dogodi, sistem prelazi u novo stanje, pri čemu se često izvršava i neka akcija. **Dijagrami stanja** predstavljaju jedan od najvažnijih načina prikaza konačnog

automata, jer kompaktno i pregledno opisuju složene zahteve sistema i njegov način interakcije sa spoljašnjim svetom. Osnovni elementi dijagrama stanja su:

- **stanje** - ponašanje ili konfiguracija sistema. Čuva informaciju o prošlim događajima i određuje reakciju na buduće.
- **prelaz** - promena iz jednog stanja u drugo, inicirana događajem.
- **događaj** - spoljašnji ili unutrašnji signal koji izaziva prelaz.
- **akcija** - operacija koju sistem izvršava kao odgovor na prelaz (npr. promena izlaza ili logovanje).



Dijagrami stanja su vrsta usmerenog grafa, pa se testiranje može zasnivati na njegovom obilasku. Na primer, možemo zahtevati da svaki prelaz bude ispitan bar jednom, što predstavlja dobar kompromis između pokrivenosti i obima testova. Može se zahtevati i pokrivenost svih stanja ili svih mogućih putanja kroz dijagram. Na primer, softverski sistem za maloprodaju ima ugrađenu opciju za otvaranje i zatvaranje kase. Njegov dijagram stanja može biti:



Test slučajevi mogu biti:

- Otvori, zatvori, ugasi program.
- Otvori, otvori, zatvori, zatvori, ugasi program.

Konačni automat koji modeluje sistem može se prikazati i **tabelama stanja**. Njihov prednost je sistematični pristup, jer prikazuju sve moguće kombinacije stanja i događaja. Na taj način mogu se uočiti i neke situacije u kojima ponašanje sistema nije definisano, što može da

spreči pojavu grešaka. Iz svakog reda može se izvesti jedan test slučaj. Njihova mana je to što postaju nepraktične ako postoji veliki broj mogućih stanja i događaja. Na primer, tabela za prethodni primer može biti:

Trenutno stanje	Događaj	Akcija	Naredno stanje
Zatvorena	**Otvori**	**Obavesti**	**Otvorena**
Zatvorena	**Zatvori**	**-**	**Zatvorena**
Zatvorena	**Ugasi program**	**-**	**Zatvorena**
Otvorena	**Otvori**	**-**	**Otvorena**
Otvorena	**Zatvori**	**Obavesti**	**Zatvorena**
Otvorena	**Ugasi program**	**-**	**Nedefinisano**

Na osnovu tabele lako uočavamo slučaj za koji stanje nije definisano i u tom slučaju možemo upozoriti korisnika i sprečiti gašenje programa pre zatvaranja kase ili prosto automatski zatvoriti kasu.

25. Tehnike testiranja. Testiranje metodama bele kutije. Putanja i broj putanja u programu. Pojam i vrste pokrivenosti. Njihovi odnosi. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Testiranje metodama bele kutije iz 20. pitanja)

Cilj testiranja bele kutije je ispitivanje različitih putanja kroz kod programa. **Putanja** je konkretan niz naredbi koje se izvršavaju tokom jednog prolaska kroz program, u zavisnosti od ulaznih podataka i kontrole toka programa. Ključni elementi koji utiču na formiranje putanja su:

- uslovna grananja (npr. naredbe if i switch)
- petlje (npr. naredbe while i for)
- skokovi (npr. naredbe break i return)
- pozivi funkcija
- izuzeci i obrada izuzetaka (npr. naredbe try i catch)

Analogno ispitivanju svih kombinacija ulaza kod testiranja crne kutije, može se zahtevati ispitivanje svih putanja kroz program. Broj mogućih putanja eksponencijalno raste sa porastom složenosti programa, pa potpuno testiranje svih putanja u praksi najčešće nije izvodljivo. Na primer, sledeći program ima 2^3 putanja:

```
int pozitivni(int a, int b, int c) {
    int brojac = 0;
    if(a > 0)
        brojac++;
    if(b > 0)
        brojac++;
    if(c > 0)
        brojac++;
    return brojac;
}
```

Slično, ako bismo za niz od n brojeva računali koliko njih je pozitivno, imali bismo 2^n različitih putanja. Ako bismo isto radili za sve brojeve koji se učitavaju sa standardnog ulaza, broj putanja zavisi od broja unetih elemenata i nije ograničen.

Zbog velikog broja mogućih putanja, koriste se **metrike pokrivenosti koda** kako bi se obezbedio balans između kvaliteta i troškova testiranja. One pomažu u identifikaciji delova sistema koji su testirani i onih koji nisu, čime omogućavaju donošenje odluka o daljem razvoju testne infrastrukture. Pre početka testiranja, određuje se odgovarajući nivo i vrsta pokrivenosti. Nakon toga, analizira se struktura i logika izvornog koda, kreiraju test primeri na osnovu relevantnih putanja i odabranih metrika pokrivenosti, i na kraju se testovi izvršavaju. Najčešće korišćene metrike pokrivenosti koda su:

- **pokrivenost putanja** - pokazuje koliki je procenat putanja u programu koje su izvršene tokom testiranja. Potpuna pokrivenost znači da je svaka moguća putanja kroz program izvršena bar jednom.
- **pokrivenost naredbi** - pokazuje koliki je procenat naredbi u programu koje su izvršene tokom testiranja. Potpuna pokrivenost se postiže kada je svaka naredba izvršena bar jednom. Najčešće se koristi u praksi jer je jednostavna za implementaciju i brzo se izračunava, ali ona ne odražava složenost logike programa i ne garantuje da su sve bitne situacije obuhvaćene testovima.
- **pokrivenost grana (odluka)** - pokazuje koliki je procenat ishoda odluka testirano u odnosu na ukupan broj odluka u programu. Za potpunu pokrivenost, svaka odluka u kodu mora biti evaluirana bar jednom kao tačna i bar jednom kao netačna.
- **pokrivenost uslova** - pokazuje koliki je procenat pojedinačnih logičkih uslova unutar složenijih izraza dobio i tačnu i netačnu vrednost tokom testiranja. Za potpunu pokrivenost, svaki uslov u svakoj odluci u kodu mora biti evaluiran bar jednom kao tačan i bar jednom kao netačan.
- **pokrivenost višestrukih uslova** - pokazuje koliki je procenat mogućih kombinacija vrednosti pojedinačnih uslova u okviru svake odluke izvršen tokom testiranja. Za potpunu pokrivenost, svaka moguća kombinacija vrednosti pojedinačnih uslova za svaku odluku mora biti izvršena bar jednom.

- **pokrivenost funkcija** - pokazuje koliki je procenat funkcija izvršen tokom testiranja. Za potpunu pokrivenost, svaka funkcija mora biti bar jednom pozvana. Korisna je za osnovnu proveru, jer ako neka funkcija nikad nije pozvana, to znači da taj deo softvera uopšte nije pokriven testovima.

Na primer, neka imamo sledeći kod:

```
if(a > 0 && b > 0) {
    a++;
    b++;
}
```

Test $a = 1,\ b = 2$ pokriva sve naredbe, ali ne pokriva sve odluke (nedostaje odluka kada uslov nije ispunjen), ne pokriva sve uslove (nedostaju uslovi da su vrednosti oba podizraza netačne), ne pokriva sve višestruke uslove (nedostaju još tri kombinacije vrednosti podizraza) i ne pokriva sve putanje (nedostaje putanja koja prolazi samo kroz liniju 1). Testovi $a = 1,\ b = 2$, $a = -1,\ b = 2$, $a = -1,\ b = -2$ i $a = 1,\ b = -2$ pokrivaju sve naredbe, sve odluke, sve uslove, sve višestruke uslove i sve putanje.

Odnosi metrika:

- Potpuna pokrivenost naredbi podrazumeva potpunu pokrivenost funkcija.
- Potpuna pokrivenost odluka podrazumeva potpunu pokrivenost naredbi.
- Potpuna pokrivenost putanja podrazumeva potpunu pokrivenost odluka.
- Potpuna pokrivenost višestrukih uslova podrazumeva potpunu pokrivenost uslova i potpunu pokrivenost odluka.

26. Tehnike testiranja. Testiranje metodama bele kutije. Pojam i vrste pokrivenosti. Preporučen nivo pokrivenosti. Alati za merenje pokrivenosti. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Testiranje metodama bele kutije iz 20. pitanja)
(Pojam i vrste pokrivenosti iz 25. pitanja)

Ne postoji univerzalan odgovor na pitanje **koji nivo pokrivenosti je dovoljno dobar**, već on zavisi od prirode softvera, konteksta upotrebe, kritičnosti i ciljeva testiranja. Ipak, postoje neke smernice. Prag za pokrivenost naredbi je od 80% do 90%, iako se često potpuna pokrivenost postavlja kao cilj. Ovaj prag predstavlja kompromis između efikasnosti i troškova testiranja. Čak ni potpuna pokrivenost ne garantuje da grešaka neće biti, jer možda nisu obuhvaćeni svi logički putevi, uslovi ili granični slučajevi. Sa druge strane, nizak nivo pokrivenosti ukazuje na povišen rizik da greške mogu ostati neotkrivene. Pokrivenost koda testovima se menja tokom razvoja, pa je potrebno redovno merenje i praćenje pokrivenosti.

Alati za merenje pokrivenosti koda koriste se za analizu koji delovi koda su bili izvršeni tokom testiranja. Većina alata funkcioniše na osnovu **instrumentacije** - proces u kom se u kod automatski umeću instrukcije koje beleže rezultate izvršavanja. Može da se vrši pre ili za vreme kompilacije, kao i nakon kompilacije, na nivou bajtkoda ili izvršnog koda. Izvršavanjem instrumentovanog koda nad test ulazima, prikupljaju se podaci o tome koje su linije, grane, uslovi ili funkcije izvršeni. Na osnovu tih informacija generiše se izveštaj o pokrivenosti, koji pomaže programerima da identifikuju nedovoljno testirane delove sistema. Izveštaji se često vizuelizuju kroz procente pokrivenosti i označene delove koda, kako bi se lakše uočili delovi koji zahtevaju dodatne testove.

27. Tehnike testiranja. Testiranje metodama bele kutije. Izbor relevantnih testova i putanja. Testiranje osnovnih putanja. Testiranje na osnovu grafa toka podataka. Primeri.

(Tehnike testiranja iz 20. pitanja)
(Testiranje metodama bele kutije iz 20. pitanja)

Izbor relevantnih testova je jedan od ključnih izazova u testiranju metodama bele kutije. Dobar izbor treba da obezbedi visok nivo pokrivenosti uz minimalan broj test primera. U praksi, preporučuje se korišćenje više jednostavnih i kratkih putanja koje se međusobno razlikuju u malim detaljima, umesto jedne veoma složene putanje, što olakšava otkrivanje grešaka i održavanje testova. Na primer, petlje mogu potencijalno generisati beskonačan broj putanja, pa se biraju reprezentativni slučajevi kao što su nulta iteracija, slučaj kada se petlja izvršava jednom, slučaj kada se petlja izvršava dva puta, kao i izvršavanje petlje $n-1$ i n puta ako je poznat gornji limit n.

Testiranje osnovnih putanja je tehnika koja osigurava da su sve logičke grane u programu ispitane bar jednom. Zasniva se na analizi grafa toka kontrole i uključuje sledeće korake:

1. Izgradnja grafa kontrole toka programa.
2. Izračunavanje ciklomatične kompleksnosti grafa - C.
3. Odabir skupa od C linearno nezavisnih osnovnih putanja.
4. Kreiranje test primera za svaku osnovnu putanju.
5. Izvršavanje testova i analiza rezultata.

Testiranje osnovnih putanja garantuje pokrivenost svih naredbi i svih grana, jer je skup osnovnih putanja konstruisan tako da pokriva svaki čvor i svaku logičku odluku u grafu toka kontrole. Glavni nedostatak ove tehnike je to što pretpostavlja da su sve osnovne putanje dostižne i izvodljive, dok u praksi neke putanje mogu biti logički ili praktično nedostižne.

Testiranje na osnovu grafa toka podataka ispituje ispravnost životnog ciklusa promenljivih i upotrebe promenljivih u različitim delovima programa. Tipične anomalije upotrebe podataka su upotreba neinicijalizovane promenljive, neiskorišćena promenljiva, ponovno definisanje, upotrebe nakon oslobađanja resursa i tako dalje. Tehnika identifikuje mesta u kodu gde se promenljive definišu i koriste, a zatim kreira test primere koji pokrivaju tok izvršavanja

programa koji prolazi kroz date definicije i korišćenja. Posebno je korisna za otkrivanje suptilnih grešaka koje nisu nužno vidljive kroz standardne tehnike testiranja, pa se koristi u bezbednosno kritičnim i složenim sistemima.

28. Tehnike testiranja. Problem proročišta. Metamorfno testiranje. Primeri.

(Tehnike testiranja iz 20. pitanja)

Proročište (test oracle) je mehanizam ili znanje koje omogućava testeru da odredi: "Očekivani rezultat za ulaz U je izlaz I" i da uporedi stvarni izlaz sa očekivanim. **Problem proročišta (oracle problem)** nastaje kada takav mehanizam ne postoji ili ga je teško definisati. Izražen je u oblastima kao što su računarska grafika, konstrukcija kompilatora i mašinsko učenje. Na primer, želimo da testiramo da li C kompajler ispravno generiše mašinski kod i ispravno optimizuje kod. To je problem jer najčešće ne postoji formalna specifikacija očekivanog izlaza za proizvoljni ulazni program. Takođe, ručna provera mašinskog koda je izuzetno teška, a automatizovana provera semantičke ekvivalencije izvornog i prevedenog programa je u opštem slučaju nerešiv problem. Problem proročišta se može rešavati na različite načine. Prvi način je korišćenje alternativnih implementacija ili ranijih verzija softvera za poređenje. Ukoliko su rezultati različiti, onda bar jedno od implementacija ima grešku. Ovo se ne može uvek primeniti jer često ne postoji više implementacija istog algoritma. Na primer, možemo rešiti problem testiranja C kompajlera tako što izvorni kod prevodimo pomoću više različitih kompajlera i proverimo da li dobijamo isti izlaz. Čak i u ovom slučaju ne možemo tvrditi da je rezultat ispravan, jer bismo možda za neke druge ulaze dobili različite izlaze. Dodatno, moguće je i da svi kompajleri imaju istu grešku.

Metamorfno testiranje je drugi način za rešavanje problema proročišta koji koristi poznavanje osobina sistema koji se testira - metamorfna svojstva. **Metamorfno svojstvo (relacija)** opisuje kako bi izlaz programa trebalo da se promeni kada bi se ulaz izmenio na određeni način. Ideja je da se na osnovu dva ulaza koja su u nekom odnosu odredi odnos između odgovarajućih izlaza. Na primer, ako ne možemo da izračunamo standardnu devijaciju za veoma dugačak niz brojeva, možemo koristiti sledeća svojstva: permutacija elemenata ne utiče na standardnu devijaciju i množenje svake vrednosti sa -1 ne utiče na standardnu devijaciju. Testiranjem se proverava da li se, za zadate ulaze, izlazi menjaju na očekivani način. Ukoliko se javi odstupanje, detektuje se greška. Na primer, za pomenuti niz napravimo četiri permutacije i proverimo da li daju isti rezultat ili proverimo da li se dobija isti rezultat za početni niz i niz u kojem su sve vrednosti pomnožene sa -1. Metamorfno testiranje omogućava detekciju grešaka kad tradicionalne metode nisu primenljive, ali je za njegovu uspešnu primenu potrebno dobro razumevanje domena problema, identifikacija relevantnih i pouzdanih metamorfnih svojstava, kao i automatizacija generisanja izvedenih testova i njihove verifikacije. Jedan od najkorisnijih tipova metamorfnih relacija je relacija ekvivalencije koja podrazumeva da transformisani ulaz mora proizvesti potpuno isti izlaz kao i original. Zbog svoje prirode, omogućava detektovanje širokog spektra grešaka. Na primer, ako imamo klasifikator slika, jedan primer metamorfnog odnosa je da ako se ulazna slika rotira za ugao do 5 stepeni, očekuje se da klasifikacija ostane nepromenjena. Sada možemo

rotirati originalne slike i porediti rezultate. U praksi, relacija ekvivalencije nije uvek dostupna, pa se koriste slabije forme, kao što su relacije očuvanja određenih karakteristika rezultata ili relacije koje predviđaju smer promene izlaza. Na primer, ako imamo sistem koji predviđa cenu nekretnine na osnovu kvadrature, lokacije i broja soba, možemo reći da u slučaju povećanja kvadrature, pri čemu broj soba i lokacija ostaju isti, očekujemo povećanje predviđene cene.

29. Načini testiranja. Manuelno testiranje.

Osnovne aktivnosti pri testiranju su definisanje test primera koji će biti predmet provere i sprovođenje postupka izvršavanja i evaluacije tih test primera. Obe ove aktivnosti mogu biti obavljene manuelno ili automatski, pa teorijski postoje četiri moguće kombinacije. Ipak, u praksi se automatsko generisanje testova i manuelno sprovođenje testiranja obično ne koristi, dok se ostale kombinacije koriste.

Manuelno testiranje podrazumeva da se obe aktivnosti sprovode manuelno, tj. da testeri smišljaju test slučajeve i samostalno sprovode testiranje i ocenjuju rezultate. Nezamenljivo je u delovima verifikacije koji zahtevaju ljudsku percepciju, razumevanje konteksta i subjektivnu procenu kvaliteta. Primer je istraživačko testiranje, gde automatizacija ne može da zameni istraživanje softvera koje vrši tester. Dodatno, ono je ključno u procesu validacije, gde je cilj utvrditi da li softver zaista zadovoljava potrebe krajnjih korisnika. Automatizovani alati mogu potvrditi prisustvo odgovarajućih elemenata, ali ne i njihovu funkcionalnu ili estetsku vrednost. Manuelnim testiranjem se proveravaju i svojstva koja se nalaze na preseku validacije i verifikacije. Primer je testiranje prihvatljivosti koje obično vrše sami korisnici, gde se između ostalog daju i subjektivne ocene o korisničkom iskustvu i interfejsu. Subjektivno korisničko iskustvo je ključno i kod aplikacija u realnom vremenu i interaktivnih aplikacija, poput video igara, sistema za virtuelnu realnost i simulacija.

30. Načini testiranja. Automatsko izvršavanje i evaluacija testova. Kontinuirana integracija.

(Načini testiranja iz 29. pitanja)

Iako manuelno testiranje omogućava uvid u korisničko iskustvo i subjektivne aspekte sistema, ono je često sporo i podložno greškama. Automatizovani testovi smanjuju potrebu za ručnim intervencijama i ubrzavaju razvoj softvera. Najčešći oblik automatizovanog testiranja je izvršavanje testova jedinica koda. Oni se obično automatski pokreću prilikom svake izmene u kodu. Postoji veliki broj okruženja za automatsko izvršavanje testova, po obrascu xxxUnit, gde xxx označava programski jezik (npr. CppUnit, JUnit, PyUnit). Pored alata za izvršavanje testova, postoje i alati koji automatizuju čitav proces testiranja, uključujući upravljanje test slučajevima, integraciju sa repozitorijumima koda, praćenje defekata i generisanje izveštaja.

Kontinuirana integracija (Continuous Integration - CI) predstavlja ključni proces u razvoju softvera, posebno u timovima gde više članova svakodnevno unosi izmene u zajednički kodni repozitorijum. Cilj je da se svaka izmena odmah objedini sa trenutnim stanjem projekta, automatski izgradi i testira, kako bi se greške otkrile što ranije. Prilikom svakog objedinjavanja koda, sistem prolazi kroz sledeće faze:

- **integracija** - sve trenutne izmene spajaju se u jedinstvenu verziju projekta.
- **izgradnja** - kod se kompajlira i pakuje u izvršni fajl ili instalacioni paket.
- **testiranje** - automatski test primeri se izvršavaju kako bi se proverila ispravnost sistema.
- **arhiviranje** - rezultujući artifakti se verzionišu i čuvaju za buduću potrebu.
- **primena** - sistem se učitava u okruženje za testiranje ili integraciju, gde može biti pokrenut i dostupan za evaluaciju.

Ovakav pristup doprinosi ranom otkrivanju grešaka, smanjenju troškova projekta, kraćem vremenu razvoja i nižem riziku prilikom isporuke novih verzija softvera.

31. Načini testiranja. Automatsko generisanje test primera.

(Načini testiranja iz 29. pitanja)

Automatsko generisanje test primera je moguće samo za određene vrste testiranja u kojima se mogu formalno definisati kriterijumi pokrivenosti i ponšanja sistema. Na primer, to su testovi usmereni na otkrivanje grešaka kao što su deljenje nulom, korišćenje neinicijalizovanih promenljivih ili pristup nedozvoljenim memorijskim lokacijama. Automatizacija u ovim situacijama ima prednost jer omogućava sistematski generisanje velikog broja ulaznih podataka uz minimalni ljudski napor. Posebno je korisno za otkrivanje ekstremnih i graničnih slučajeva. Na primer, alati za fuzzing mogu nasumično, ali ciljano, generisati razne kombinacije ulaza, uključujući i neke neuobičajene, kako bi izazvali neželjeno ili nepredviđeno ponašanje softvera. Slično, statička analiza može identifikovati potencijalne probleme u kodu, na osnovu čega se mogu automatski generisati test primeri koji testiraju te kritične tačke. Kada se radi o testiranju složenih funkcionalnosti koje zavise od višeslojnih uslova ili korisničkih odluka, automatizovano generisanje test primera je veoma izazovno, a nekad i neizvodljivo. Zbog toga se u praksi automatizovano generisanje testova najefikasnije koristi kao dopuna manuelnom procesu.

32. Debagovanje. Veza izvršivog koda i debagera. Režim prevođenja za upotrebu. Primeri.

Debagovanje je proces u razvoju softvera koji ima za cilj pronalaženje greške, odnosno uzroka defekta u programu. **Debager** je alat koji se koristi za praćenje izvršavanja programa radi boljeg razumevanja ponašanja programa i lakšeg pronalaženja greške u programu. Kako bi informacije koje debager pruža bile precizne i razumljive, neophodna je podrška kompajlera i linkera, koji obezbeđuju povezivanje izvršivog koda sa izvornim, kao i razne druge korisne podatke. Debageri se često integrišu u razvojna okruženja koja im obezbeđuju

grafički korisnički interfejs, koji olakšava upotrebu debagera. Ipak, to je samo sredstvo interakcije sa debagerom. Sama funkcionalnost debagera ostaje nezavisna od načina na koji se korisniku prikazuje.

Tokom kompilacije dostupne su mnoge opcije i podešavanja koja omogućavaju precizno upravljanje načinom prevođenja izvornog koda u izvršivi program. **Režim kompilacije** je skup opcija kompilacije koje se često zajedno koriste sa određenim ciljem. Postoje:

- **režim prevođenja za upotrebu (release mode)** - režim prevođenja koji ima za cilj dobijanje optimizovane izvršive verzije namenjene krajnjem korisniku. Dobija se release verzija programa.
- **režim prevođenja za pronalaženje grešaka (debug mode)** - režim prevođenja koji ima za cilj dobijanje izvršive verzije programa namenjene programeru radi lakšeg otkrivanja grešaka. Dobija se debug verzija programa.
- **kombinovani režim** - kombinacija prethodna dva režima. Koristi se kada je potrebno naći grešku u programu, a režim za pronalaženje grešaka ne daje željene rezultate.

Debager se može koristiti za svaku izvršivu verziju programa, ali se najviše informacija dobija za debug verziju, dok se za release verziju dobija samo uvid u asemblerski kod.

Primarni cilj prevođenja u **režimu za upotrebu** je postizanje visoke efikasnosti izvršavanja kroz različite strategije optimizacija koje sprovodi kompajler, kao što su uklanjanje mrtvog koda, umetanje funkcija ili preuređivanje instrukcija. Primarno, optimizacije se odnose na brzinu izvršavanja, ali za neke aplikacije su neophodne optimizacije koje se odnose na smanjivanje veličine izvršive datoteke i upotrebe memorije u fazi izvršavanja. Tada je u pitanju prevođenje u **režimu za upotrebu sa minimalnom veličinom**. Sprovedene optimizacije smanjuju mogućnost povezivanja izvršivog koda sa izvornim datotekama. Neki delovi koda se mogu potpuno izostaviti, promeniti redosled ili transformisati do neprepoznatljivosti, pa se ova verzija prevođenja gotovo nikada ne koristi za analizu i dijagnostiku grešaka. Na primer, gcc kompajler nema jedinstvenu opciju za prevođenje za upotrebu, već se uključuju opcije visokog nivoa optimizacije (-O2 ili -O3), dok se za režim za upotrebu sa minimalnom veličinom koristi opcija -Os. Opcija -DNDEBUG definiše makro NDEBUG koji onemogućava funkciju assert() čija je upotreba karakteristična u fazi otklanjanja grešaka.

33. Debagovanje. Veza izvršivog koda i debagera. Režim prevođenja za pronalaženje grešaka. Formati za predstavljanje pomoćnih informacija. Primeri.

(Debagovanje iz 32. pitanja)
(Veza izvršivog koda i debagera iz 32. pitanja)

Primarni cilj prevođenja u **režimu za pronalaženje grešaka** je pravljenje izvršive verzije koda koja za svaku instrukciju u fazi izvršavanja omogućava preciznu povezanost sa odgovarajućim linijama izvornog koda. Kompajler u ovom režimu generiše dodatne

informacije što omogućava debageru da uspostavi ove veze. U ovom režimu su optimizacije isključene ili svedene na minimum kako bi veza bila preciznija. Kao posledica, debug izvršivi fajl je obično znatno veći nego release fajl. Takođe, izvršavanje u ovom režimu može biti sporije i manje memorijski efikasno, ali to je prihvatljivo u fazi razvoja. Na primer, gcc kompajler nema jedinstvenu opciju za prevođenje za pronalaženje grešaka, već se uključuju opcije niskog nivoa optimizacije (-O0) i opcija koja uključuje upisivanje informacija za debagovanje u izvršivu datoteku (-g).

Formati za predstavljanje pomoćnih informacija za debagovanje preciziraju na koji način kompajler može da zapiše podatke kao što su nazivi funkcija i promenljivih, tipovi podataka i slično. Najpoznatiji formati su:

- **DWARF** - nezavisan je od formata izvršive datoteke, ali se najčešće koristi za izvršive i povezane datoteke (**ELF**) u okviru UNIX operativnih sistema. Može se koristiti za programe pisane u jezicima C, C++, Fortran i mnoge druge, a po potrebi se može i proširiti za specifične funkcionalnosti nekih drugih jezika. Metapodaci se umeću direktno u izvršivi kod, pa je debug verzija programa veća u odnosu na release verziju. Na primer, kompajler gcc i Clang koriste ovaj format i njih mogu da koriste debageri gdb i lldb.
- **Microsof CodeView** - format koji koristi operativni sistem Windows. Koristi se za programe pisane u jezicima C, C++ i jezicima .NET platforme. Metapodaci se distribuiraju odvojeno od izvršivog koda, u okviru datoteka sa ekstenzijom pdb. Oni se koriste po potrebi, tako što se učitavaju u debager. Na primer, kompajler MSVC koristi ovaj format i njega mogu da koriste debageri WinDbg i Visual Studio Debugger.

34. Debagovanje. Veza izvršivog koda i debagera. Kombinovani režimi prevođenja. Primeri.

(Debagovanje iz 32. pitanja)
(Veza izvršivog koda i debagera iz 32. pitanja)

Neke greške se mogu ispoljavati isključivo u release verziji. Jedan od mogućih uzroka je to što debug verzija sadrži dodatne informacije, kao i inicijalizaciju memorije koju release verzija ne poseduje. Na taj način se greške mogu zamaskirati, iako su prisutne u izvornom kodu. Moguće je i da dođe do greške u kompajleru prilikom nekih optimizacija. U oba slučaja, debagovanje debug verzije ne može otkriti uzrok greške, a debagovanje release verzije je otežano zbog nepostojanja veze između izvornog i izvršivog koda. Zbog toga se sve više ulaže u unapređenje mogućnosti za debagovanje optimizovanog koda, a problemu se pristupa upotrebom **kombinovanog režima** prevođenja, gde se optimizacije kombinuju sa zadržavanjem debag informacija. Na primer, kombinovani režim kod gcc kompajlera bi podrazumevao viši stepen optimizacije (npr. -O2), opciju koja uključuje informacije za debagovanje (-g) i opciju -DNDEBUG kako bi izvršavanje više ličilo režimu prevođenja za upotrebu.

35. Debagovanje. Veza izvršivog koda i debagera. Anti-debagovanje.

(Debagovanje iz 32. pitanja)
(Veza izvršivog koda i debagera iz 32. pitanja)

Mogućnost debagovanja izvršive verzije programa nije uvek poželjna, pogotovo u kontekstu zaštite intelektualne svojine, bilo radi zaštite od neovlašćenog kopiranja ili radi sprečavanja obrnutog inženjeringa. Nakon generisanja izvršive verzije programa, moguće je primeniti specijalizovane alate koji modifikuju izvršivi kod sa ciljem otežavanja i ometanja debagovanja. Ove tehnike nazivaju se **anti-debagovanje**. Koriste se i u malicioznim programima tako da otežaju prepoznavanje od strane antivirusa i sistema za detekciju pretnji. Tehnike mogu uključivati detekciju prisustva debagera, izmenu toka izvršavanja u slučaju da je debager prisutan, korišćenje specifičnih instrukcija koje se ponašaju drugačije u prisustvu nadgledanja, kao i tehnike **obfuskacije** koja ne menja funkcionalnost programa, već način na koji je on predstavljen. To može biti preimenovanje simbola, dodavanje beskorisnog ili mrtvog koda, šifrovanje delova koda ili podataka i slično. Anti-debagovanje predstavlja balans između zaštite i funkcionalnosti - prekomerna zaštita može negativno uticati na stabilnost, efikasnost i održivost.

36. Debagovanje. Vrste debagovanja. Interaktivno debagovanje. Implementacija interaktivnog debagovanja, tačaka prekida i tačaka posmatranja. Primeri.

(Debagovanje iz 32. pitanja)

Vrste debagovanja su:

- **interaktivno debagovanje** - debager započinje proces i prati njegovo izvršavanje.
- **udaljeno debagovanje** - debager prati izvršavanje procesa koji je već započeo sa radom.
- **debagovanje nakon prekida izvršavanja programa (post-mortem debagovanje)** - debager analizira stanje programa nakon što je on završio sa radom.

Interaktivno debagovanje podrazumeva dinamičku analizu programa u realnom vremenu uz aktivno učešće programera, koji koristi debager da bi pratio i kontrolisao tok izvršavanja aplikacije. Omogućava dvosmernu komunikaciju sa debagerom, putem komandne linije ili grafičkog korisničkog interfejsa. Osnovni mehanizam interaktivnog debagovanja su **tačke prekida (breakpoints)**, koje omogućavaju da se izvršavanje programa automatski pauzira na odabranoj liniji koda. Kada se izvršavanje zaustavi, debager omogućava inspekciju promenljivih, sadržaja steka, registara i ostalih komponenti stanja programa. Nakon analize, izvršavanje može biti nastavljeno korak po korak (**step**) ili do sledeće tačke prekida (**next**). Pored običnih prekida, moguće je postavljati i **uslovne tačke prekida** koje se aktiviraju samo kada je ispunjen određeni logički uslov. Debageri omogućavaju i postavljanje **tačaka posmatranja (watchpoints)** nad određenim lokacijama u memoriji. Kada se sadržaj

označene memorijske lokacije promeni, debager pauzira izvršavanje, čime se olakšava praćenje neželjenih ili neočekivanih promena podataka. Ova tehnika je posebno korisna kod analize grešaka izazvanih nepredviđenim upisima u memorija, kao što je prekoračenje bafera. Debageri nekad pružaju i naprednije opcije, kao što je **hot code replacement** - izmena koda u toku izvršavanja. To omogućava programeru da prilagodi ili ispravi deo programa bez potpunog zaustavljanja i ponovnog pokretanja izvršavanja. Još jedna napredna tehnika je **reverse debugging**, koja omogućava programeru da prati izvršavanje koda unazad, analizirajući operacije koje su dovele do određenog stanja.

Implementacija:

- Debageri su sistemski zavisni alati jer pomoću sistemskih poziva mogu upravljati procesima. Na primer, na sistemu Linux važna je funkcija ptrace kojom jedan proces (najčešće debager) može da kontroliše drugi proces i da upravlja njegovim unutrašnjim stanjem. Debageri je koriste da zaustavljaju program, posmatraju memoriju programa i da je menjaju.
- Kada se postavi tačka prekida, debager zameni odgovarajuću instrukciju instrukcijom prekida, pri čemu čuva originalnu instrukciju. Kada se naiđe na instrukciju prekida, desi se hardverski izuzetak, OS zaustavlja proces i obaveštava debager o tome. Debager proverava da li je prekid u listi očekivanih prekida koje je postavio. Ako je greška u originalnom kodu, onda se dopusti da se ta greška i izvrši. Ako je u pitanju tačka prekida koju je postavio debager, on na tom mestu omogućuje uvid u sve vrednosti registara procesa, stanje memorije, kao i pročitane informacije o procesu povezane sa informacijama o izvornom kodu koje su generisane od strane kompajlera i linkera prilikom prevođenja. Kada korisnik želi da nastavi sa izvršavanjem, debager zameni instrukciju prekida originalnom instrukcijom, izvrši se, zameni ponovo originalnu instrukciju instrukcijom prekida i prepusti kontrolu programu. Ako je u pitanju uslovna tačka prekida, debager proverava i uslov i ako on nije ispunjen preskače zaustavljanje procesa.
- Ako je dostupna podrška hardvera, biće podignut izuzetak kada se vrednost na nekoj memorijskoj lokaciji promeni, što omogućava implementaciju tačaka posmatranja. Ako podrška hardvera ne postoji ili se zahteva praćenje većeg broja vrednosti nego što je dostupno hardverski, debager mora da izvršava instrukciju po instrukciju i da za svaku proverava šta se dešava na traženim memorijskim lokacijama, što usporava izvršavanje.

37. Debagovanje. Vrste debagovanja. Udaljeno debagovanje. Debagovanje nakon prekida izvršavanja programa. Primeri.

(Debagovanje iz 32. pitanja)
(Vrste debagovanja iz 36. pitanja)

Udaljeno debagovanje podrazumeva interaktivno debagovanje pri čemu se aplikacija izvršava na jednom sistemu (**ciljni sistem**), dok se proces debagovanja obavlja na drugom sistemu (**udaljeni sistem**). Povezivanje se ostvaruje preko mreže koristeći odgovarajuće protokole i servise. Omogućava da debager sa udaljenog računara upravlja izvršavanjem

programa na ciljnom sistemu, postavlja tačke prekida, ispituje vrednosti promenljivih i slično. Naročito je korisno kada ciljni sistem ima ograničene resurse i ne može lokalno izvršavati debager, što je čest slučaj kod uređaja sa ugrađenim računarom (embedded systems), kao i prilikom razvoja sistema kojima nije lako obezbediti direktan fizički pristup. Na primer, korišćenjem debagera gdb na udaljenom sistemu i servisa gdbserver na ciljnom sistemu, moguće je uspostaviti vezu između razvojne stanice i ciljnog sistema. Da bi debagovanje bilo moguće, ciljna i udaljena arhitektura moraju biti iste ili debager na udaljenom sistemu mora imati podršku za arhitekturu ciljnog sistema.

Debagovanje nakon prekida izvršavanja programa (post-mortem debagovanje) analizira ponašanje programa nakon što je njegovo izvršavanje već prekinuto, najčešće usled greške. Oslanja se na informacije koje su zabeležene u trenutku prekida, bez mogućnosti ponovnog pokretanja aplikacije u istom stanju. Posebno je važno u produkcionim okruženjima, gde se greške moraju analizirati bez direktnog ponovnog izvršavanja programa u identičnim uslovima. Za post-mortem debagovanje ključnu ulogu imaju **core dump datoteke** koje sadrže snimak memorije procesa u trenutku pada, kao i relevantne informacije (sadržaj steka, registara, segmenta podataka i koda). Za generisanje ovih datoteka potrebna su dodatna podešavanja u okviru OS, jer je podrazumevano opcija najčešće isključena. Tehnike post-mortem analize uključuju pregled sadržaja memorije i registara, rekonstrukciju stek poziva funkcija, utvrđivanje poslednjih instrukcija koje su se izvršile, uvid u vrednosti promenljivih u trenutku pada i analizu sistemskih logova i izlaznih datoteka.

38. Debagovanje. Primeri debagera.

(Debagovanje iz 32. pitanja)

Upotreba debagera u različitim jezicima ima mnogo zajedničkih osobina - uobičajene funkcionalnosti uključuju postavljanje i uklanjanje tačaka prekida, koračanje naredbu po naredbu, ispitivanje vrednosti promenljivih i slično. Sintaksa i interfejs se mogu razlikovati, ali principi ostaju isti. Primeri debagera:

- **gdb** - omogućava sve tri vrste debagovanja, a multiarch gdb omogućava pokretanje i analizu izvršivih datoteka namenjenih procesorskim arhitekturama koje su različite od one na kojoj se debagovanje izvršava. Napisan je u C-u i razvija se kao deo GNU projekta. Koristi se za različite programske jezike, kao što su Ada, C, C++, Objective-C, Pascal, Go, Rust i Java. Može se integrisati u veliki broj razvojnih okruženja i dostupan je na velikom broju operativnih sistema.
- **lldb** - razvija se kao deo projekta LLVM i omogućava interaktivno debagovanje. Implementiran je u jeziku C++ i podržava jezike kao što su C, C++, Objective-C i Swift. Može se integrisati u različita razvojna okruženja, a prvenstveno je namenjen Unix operativnim sistemima, ali ima podršku i za Windows.

- **WinDbg** i **Visual Studio Debugger** - razvija ih kompanija Microsoft i namenjeni su za rad na OS Windows. Imaju podršku za jezike C, C++ i jezike zasnovane na .NET platformi kao što su C# i F#. Podržavaju interaktivno debagovanje i post-mortem analizu. Visual Studio Debugger je integrisan u razvojno okruženje Visual Studio i pretežno je namenjen razvoju korisničkih aplikacija. WinDbg je manje poznat, ali je napredniji i pogodan za sistemsko debagovanje.
- **jdb** - namenjen je za Javu i deo je standardne Java distribucije. Koristi se na sličan način kao gdb. Može se integrisati u razvojna okruženja kao što je IntelliJ IDEA. Za Javu se mogu koristiti i neki drugi debageri u kombinaciji sa specifičnim podešavanjima i prevodiocima koji generišu izvršivi kod.
- **pdb** - modul programskog jezika Python koji omogućava interaktivno debagovanje iz terminala ili integraciju u razvojne alate.

39. Debagovanje. Otvoreni problemi.

(Debagovanje iz 32. pitanja)

Iako je debagovanje ključna tehnika za analizu izvršavanja programa, primenljivost debagera je ograničena i postoje razni problemi:

- **višenitne aplikacije** - greške mogu zavisiti od redosleda izvršavanja zbog konkurentnog izvršavanja niti, pa debageri često ne uspevaju da precizno upravljaju svim nitima istovremeno. Čak i pokretanje aplikacije u debageru može promeniti ponašanje programa i redosled izvršavanja niti. Debageri su i dalje efikasni za praćenje pojedinačnih niti, a za višenitne sisteme se koristi kombinacija raznih tehnika kao što su logovanje, profajliranje, upotreba sanitajzera i statička analiza.
- **distribuirane aplikacije** - njihovo izvršavanje uključuje više nezavisnih komponenti koje rade na različitim fizičkim mašinama, za šta tradicionalni debageri nisu dizajnirani.
- **sistemi sa ugrađenim računarom** i **aplikacije koje rade u realnom vremenu** - strogi vremenski zahtevi i ograničeni resursi ne dozvoljavaju pauziranje programa ili umetanje dodatnog koda za analizu. Korišćenje debagera nekad može dovesti do narušavanja funkcionalnosti sistema.
- **release aplikacije** - optimizacije koje vrši kompajler menjaju strukturu izvršivog koda pa se narušava mogućnost povezivanja izvornog i izvršivog koda.
- **veoma veliki i kompleksni sistemi** - količina podataka i broj međusobno povezanih komponenti prevazilaze mogućnosti praćenja putem debagera.

Zbog svih navedenih problema, debagovanje se u praksi najčešće kombinuje sa logovanjem, automatskim testiranjem, statičkom analizom i slično.

40. Debagovanje. Štampanje umesto debagera. Primeri.

(Debagovanje iz 32. pitanja)

Štampanje vrednosti promenljivih predstavlja jednu od prvih tehnika debagovanja koju većina programera usvaja. Glavna prednost je jednostavnost i intuitivnost ove tehnike koja ne zahteva poznavanje dodatnih alata. Podrazumeva umetanje naredbe za štampanje u ključne delove koda, kako bi se prikazale trenutne vrednosti promenljivih, pratila kontrola toka i uočili neočekivani rezultati tokom izvršavanja. Na primer, ako program proizvodi netačan rezultat, programer može umetnuti štampanje unutar petlje ili grane uslova kako bi ispratio kako se vrednosti promenljivih menjaju. Uprkos dostupnosti savremenih i moćnih alata za debagovanje, štampanje je i dalje široko rasprostranjeno. Glavne mane ove tehnike su:

- Umetanje funkcije za štampanje zahteva ponovno prevođenje programa, što može biti vremenski zahtevno.
- Umetanje funkcije za štampanje može promeniti raspored memorije programa i time prikriti ili izmeniti ponašanje greške.
- Štampanjem se ne mogu pratiti svi relevantni aspekti stanja programa, kao što su sadržaji registara, vrednosti pokazivača ili stanje steka.
- Štampanje ne omogućava zaustavljanje programa niti interaktivno ispitivanje stanja promenljivih.

Ipak, štampanje može biti opravdano ako za određenu arhitekturu ili platformu ne postoji debager ili ako debager svojim prisustvom menja ponašanje programa.

41. Instrumentacija.

Instrumentacija je proces dodavanja instrukcija u program kako bi se, tokom njegovog izvršavanja, pratile određene pojave i prikupljali podaci o njegovom ponašanju. Koristi se u jednoj vrsti profajlera i u sanitajzerima. Dobra instrumentacija ispunjava sledeće osobine:

- prikuplja samo potrebne podatke - premalo informacija je beznačajno, a previše informacija može usporiti program i obradu samih podataka.
- ne utiče na funkcionalnost programa - ako utiče na funkcionalnost, onda prikupljeni podaci ne oslikavaju program na pravi način.
- ne usporava previše rad programa - može dovesti do praktične neupotrebljivosti programa.
- ne povećava previše upotrebu memorije - može dovesti do praktične neupotrebljivosti programa.

Instrumentaciju može vršiti programer dodavanjem i inkrementiranjem brojača na željenim mestima u kodu, a može se vršiti i automatski. Automatsku instrumentaciju može vršiti kompajler i/ili linker tokom prevođenja i linkovanja, specijalizovani alat na izvršivom

programu ili specijalizovani alat tokom izvršavanja programa. Instrumentacija se može vršiti prilikom proizvoljne vrste prevođenja, kao i nad proizvoljnom izvršivom verzijom programa. Ipak, za dinamičko otkrivanje grešaka poželjno je debug prevođenje, dok je za praćenje performansi neophodno release prevođenje.

42. Profajliranje. Upotreba profila.

Proces optimizacije predstavlja sastavni deo razvoja softvera, jer obezbeđuje da implementacija ispunjava zahteve u pogledu brzine izvršavanja i potrošnje resursa. Kako bi se precizno identifikovali delovi koda koji zahtevaju poboljšanje, koriste se alati koji generišu detaljne informacije o ponašanju programa tokom izvršavanja - **profajleri**. **Profajliranje** je vid dinamičke analize programa kojim se tokom izvršavanja programa, nad unapred definisanim skupom ulaznih podataka, prikupljaju detaljni podaci o ponašanju programa u realnim uslovima. Rezultat ovog procesa je skup podataka poznat kao **profil** programa. On može da obuhvata različite informacije, kao što su frekvencije poziva funkcija, procenat ukupnog vremena utrošenog u pojedinačnim delovima koda, informacije o korišćenju memorije i tako dalje. Profajliranje može biti implementirano softverski, hardverski ili kao kombinacija oba pristupa. Za neke vrste merenja, kao što je broj promašaja u keš memoriji, hardverska podrška je neophodna. Profajliranje se može zasnivati na instrumentaciji ili uzorkovanju, gde se povremeno beleže karakteristični podaci o stanju sistema. Obe tehnike imaju svoje prednosti i mane.

Profil se može upotrebljavati na različite načine. Najčešća upotreba je identifikacija **vrućih delova koda (hot spots)** - delovi koda koji se najčešće izvršavaju ili najviše doprinose ukupnom vremenu izvršavanja. Profil se može koristiti i za procenu pokrivenosti koda testovima, detektovanje curenja memorije i raznih grešaka u kodu. Optimizaciju na osnovu profila može izvršiti programer ili prevodilac, automatski. Takve optimizacije koje vrši prevodilac nazivaju se **optimizacije vođene profilima**. Postoje dve vrste:

- **optimizacija u fazi kompilacije pre izvršavanja programa** - zahteva da je profil dostupan, što znači da ovoj kompilaciji prethodi jedna kompilacija i izvršavanje programa sa ciljem prikupljanja profila. Prikupljanje profila može biti vremenski i memorijski zahtevno. Zbog toga se nekada koriste **statički profajleri**, tj. predviđanje profila metodama mašinskog učenja na osnovu karakteristika koda.
- **optimizacija u fazi kompilacije tokom izvršavanja programa** - koristi profil koji se dobija tokom izvršavanja da bi se donele odluke o tome da se neki delovi koda kompajliraju, umesto interpretiraju. Karakteristična je za kompilaciju u toku izvršavanja (JIT).

43. Profajliranje. Kvalitet profila.

(Profajliranje iz 42. pitanja)

Da bi se donosile odluke o optimizaciji na osnovu profila, važno je da izvršavanje u okviru kojeg se vrši profajliranje reflektuje realnu upotrebu programa. U suprotnom, mogu se

propustiti prilike za optimizaciju programa ili se mogu doneti pogrešne odluke. Na primer, ako funkciju koja se u praksi primenjuje nad velikim nizovima profajliramo sa malim nizovima, onda ne možemo uočiti problem sa performansama koji može nastati u praksi. Ili, ako imamo grananje gde se u praksi if grana izvršava 20 puta češće od else grane, ali nas ulazni podaci za profajliranje vode u else granu, može se doći do zaključka da treba optimizovati else granu. To neće dati vidljive rezultate, jer se ta grana retko izvršava. Prilikom optimizacije na osnovu profila treba pronaći ravnotežu između vremena utrošenog na prikupljanje podataka i koristi od dobijenih informacija. U početnim fazama optimizacije koriste se jednostavnije i manje precizne metode kako bi se identifikovali najveći problemi, a kasnije se koriste preciznije tehnike za otkrivanje suptilnijih problema.

44. Profajliranje. Profajliranje uzimanjem uzoraka. Primeri.

(Profajliranje iz 42. pitanja)

Profajler zasnovan na uzorkovanju periodično prekida izvršavanje programa, najčešće u fiksnim vremenskim intervalima, koristeći prekide OS. Prilikom svakog prekida, profajler snima stek poziva svake niti. Prikupljeni podaci se agregiraju tako da budu pogodni za analizu. Na osnovu dovoljno velikog broja uzoraka moguće je napraviti preciznu procenu koji delovi koda se najčešće izvršavaju ili gde se troši najviše vremena. Ova tehnika pruža statističku aproksimaciju, pa ako se neka funkcija izvršava vrlo brzo i retko, moguće je da neće biti obuhvaćena uzorkovanjem. Tehnika omogućava programu da se izvršava gotovo punom brzinom, pa je pogodna za velike i složene aplikacije, kao i aplikacije u realnom vremenu. Dodatno, nije potrebno ponovno prevođenje izvornog koda već se vrši nad izvršivim datotekama. Primer ovog tipa profajlera je **perf** koji je uveden kao deo Linux kernela 2009. godine. Cilj ovog alata je bio da zameni starije i manje fleksibilne alate za profajliranje uzorkovanjem, pa se razvija zajedno sa Linux kernelom. Pruža statistički pregled vremena izvršavanja i resursa koje program koristi. Karakterišu ga mali troškovi upotrebe i visoka preciznost. Omogućava primenu na nivou pojedinačnih procesa, ali i na nivou celog sistema. Rezultati mogu biti prikazani tekstualno ili u kombinaciji sa vizuelnim alatima za dublju analizu. Još jedan primer je **plameni graf (flame graph)**. Daje vizuelizaciju koja se koristi za analizu performansi softvera. Generiše se na osnovu rezultata rada profajlera. Predstavlja interaktivan prikaz poziva metoda tokom izvršavanja programa. Svaka traka odgovara pozivu jedne metode, a svaki "plamen" odgovara jednom nizu poziva metoda. Širina trake proporcionalna je vremenu koje je program proveo u toj metodi i metodama koje su iz nje pozvane.

45. Profajliranje. Instrumentaciono profajliranje. Profajliranje putanja, grana i blokova.

(Profajliranje iz 42. pitanja)

Instrumentacijom se mogu pratiti različite karakteristike izvršavanja programa. Profili koji se koriste za određivanje delova koda koji se najčešće izvršavaju mogu se dobiti na osnovu sledećih vrsta profajliranja:

- **profajliranje putanja**
- **profajliranje blokova**
- **profajliranje grana**

Instrumentacija se može vršiti u fazi kompilacije i u tom slučaju se profajliranje vrši izvršavanjem instrumentovanog programa. Instrumentacija se može vršiti i u fazi izvršavanja programa i u tom slučaju profajler dinamički ubacuje dodatne instrukcije u program.

Profajliranje putanja je vid profajliranja kojim se dobijaju informacije o najčešće korišćenim putanjama kroz program. Ova vrsta profila sadrži i informacije o profilima grana i blokova. Zahteva kompleksne algoritme i najviše utiče na performanse izvršavanja tokom profajliranja.

Profajliranje blokova daje informacije o ukupnom broju izvršavanja svakog bloka u programu. Blok može biti funkcija ili deo koda u kome se ne nalaze instrukcije grananja ili skokova. Naivni algoritam bi podrazumevao da se u svaki blok umetne brojač, ali ovo poprilično usporava program i opterećuje memoriju. Ova vrsta profajliranja ne daje informacije o tome koje su putanje kroz program najčešće, kao ni koji su prelazi između blokova česti.

Profajliranje grana daje informacije o prelascima koji se ostvaruju instrukcijama grananja ili skoka koje prebacuju tok izvršavanja iz jednog bloka u drugi. Mogu se dobiti i podaci koji se dobijaju profajliranjem blokova, jer se broj izvršavanja bloka može dobiti tako što se sumiraju brojači svih grana koje ulaze u blok. Naivni algoritam bi podrazumevao da se za svaku naredbu skoka umetne brojač, ali to previše opterećuje program. Ako se instrumentacija radi u fazi kompilacije, moguće je primeniti **Knutov algoritam**:

1. Konstruisati graf kontrole toka programa, u kojem svaki čvor predstavlja blok instrukcija, a grana naredbu skoka ili grananja.
2. Za dati graf izračunati razapinjuće stablo.
3. Granama koje ne pripadaju stablu dodati brojač.

Umesto svake grane, instrumentuje se $e - v + 1$ grana, gde je v broj čvorova, a e broj grana grafa. Isti broj grana se može instrumentovati na više načina, jer algoritam koji traži razapinjuće stablo može dati različite rezultate u zavisnosti od redosleda obilaska grana. Broj izvršavanja grana koje ne sadrže brojač može se izračunati na osnovu profajliranih vrednosti jer je zbir izlaznih vrednosti grana iz jednog bloka jednak zbiru ulaznih vrednosti grana u taj blok. Optimalno razapinjuće stablo je ono kod kog se grane najveći broj puta izvršavaju, jer se te grane neće instrumentovati i neće opterećivati program. Ipak, to je informacija koja se profajliranjem i traži pa je teško izabrati takvo stablo.

46. Profajliranje. Instrumentaciono profajliranje. Smanjivanje troškova instrumentacije.

(Profajliranje iz 42. pitanja)
(Instrumentaciono profajliranje iz 45. pitanja)

Instrumentacija omogućava dobijanje preciznog profila, ali značajno troši vremenske i memorijske resurse pa postoje razni pristupi smanjivanju troškova uz smanjivanje preciznosti profila. Prvi pristup podrazumeva naizmenično izvršavanje instrumentovanog i originalnog koda. U kod se umeću provere na osnovu kojih se odlučuje da li se izvršava instrumentovani kod ili se ostaje u originalnom kodu. Potrebno je da postoje dve verzije koda - instrumentovani kod, koji se naziva i **duplirani kod**, kao i originalni kod, koji se naziva **proveravajući kod** jer se u njemu ispituje uslov koji, ako je ispunjen, omogućava prelazak u duplirani kod. Kada izvršavanje pređe u duplirani kod, ono tu ostaje ograničeno vreme nakon čega se vraća u proveravajući kod. Prelazak se može vršiti na osnovu fiksiranih vremenskih intervala ili brojača. U prvom slučaju, kada istekne odgovarajući vremenski interval, sledeći prelazak se dešava kada se ponovo ispita uslov prelaska. U drugom slučaju, čuva se brojač koji se dekrementira i kada dođe do nule, prelazi se u duplirani kod i resetuje brojač. Drugi pristup je najčešće precizniji. Ovaj pristup smanjivanja troškova instrumentacije povećava potrebnu memoriju i vreme kompilacije. Postoje i verzije koje ne prave kopiju celog koda, već samo onih delova koji su vezani za instrumentaciju, što smanjuje memoriju i vreme kompilacije.

47. Profajliranje. Instrumentaciono profajliranje. Instrumentacija u fazi izvršavanja. Valgrind i faze transformacije koda.

(Profajliranje iz 42. pitanja)
(Instrumentaciono profajliranje iz 45. pitanja)

Valgrind je platforma koja omogućava izvršavanje programa, njegovu instrumentaciju u fazi izvršavanja i snimanje izveštaja koji nastaju prilikom analize izvršavanja programa. Koristi se kao osnova za pravljenje alata za dinamičku analizu programa - profajlera i alata za dinamičku detekciju grešaka. Svi Valgrind alati rade na istoj osnovi koja obuhvata podršku za izvršavanje i snimanje rezultata analize, kao i interfejs ka definisanju instrumentacije.

Valgrind deli originalni kod u sekvence koje se nazivaju **osnovni blokovi**. To su linearne sekvence mašinskog koda, na čiji se početak dolazi nekom naredbom skoka, koje u sebi ne sadrže grananja, pozive funkcija ili skokove, i koje se završavaju skokom, pozivom funkcije ili naredbom povratka u funkciju pozivaoca. Veličina osnovnog bloka ograničena je na maksimalno šezdeset mašinskih instrukcija. Valgrind dopunjava mašinski kod kodom koji vrši instrumentaciju, pojedinačno po osnovnim blokovima, neposredno pre prvog izvršavanja samog bloka. Faze transformacije koda su:

1. **Izgradnja optimizovane međureprezentacije** - disasembliranje (prevođenje mašinskog koda u međureprezentaciju) i standardne optimizacije programskih prevodilaca.

2. **Instrumentacija** - vrši se nad generisanom međureprezentacijom. Blokovi se prosleđuju alatu koji može proizvoljno da ih transformiše.
3. **Prevođenje u izvršivi kod** - obuhvata optimizacije, izgradnju stabla, odabir instrukcija, alokaciju registara i asembliranje. Druga faza optimizacije je jednostavnija od prve. Nakon nje se od međureprezentacije kreira stablo radi lakšeg odabira instrukcija. Prilikom odabira instrukcija, koriste se virtuelni registri koji se određuju u fazi alokacije registara. Na kraju, kod se prevodi u mašinski i smešta u memoriju.

Sve faze obavlja Valgrind, osim faze instrumentacije koju obavlja odgovarajući alat. Valgrind troši najviše vremena na proces transformacije koda, kao i pronalaženje i izvršavanje transformisanog koda.

48. Profajliranje. Instrumentaciono profajliranje. Valgrind i najpoznatiji Valgrind alati. Cachegrind i Callgrind.

(Profajliranje iz 42. pitanja)
(Instrumentaciono profajliranje iz 45. pitanja)
(Valgrind iz 47. pitanja)

Najpoznatiji Valgrind alati su:

- **Memcheck** - detektor memorijskih grešaka.
- **Massif** - detektor memorijskih grešaka praćenjem upotrebe dinamičke memorije.
- **Cachegrind** - profajler keš memorije.
- **Callgrind** - profajler funkcija.
- **Helgrind** i **DRD** - detektori grešaka višenitnog izvršavanja.

Cachegrind je alat Valgrind platforme namenjen profajliranju keš memorije i izvršavanja grana. Simulira memorijski podsistem procesora i prati pristupe softverskom kešu tokom izvršavanja programa. Uključuje keš prvog nivoa (L1), koji je podeljen na deo za instrukcije (I1) i deo za podatke (D1), kao i keš drugog nivoa (L2). Na mašinama sa tri ili više nivoa keša, Cachegrind modeluje prvi i poslednji nivo. Prikuplja detaljnu statistiku o pristupima memoriji i kešu na nivou celog programa, ali i na nivou pojedinačnih funkcija. Prati ukupan broj izvršenih instrukcija, čitanja i pisanja u memoriju, kao i broj promašaja pa omogućava identifikaciju uskih grla u performansama vezanim za efikasnost upotrebe keš memorije.

Callgrind je alat Valgrind platforme namenjen profajliranju poziva funkcija i generisanju grafa poziva funkcija, gde se prikazuju odnosi između pozivaoca i pozvanih funkcija, broj izvršenih instrukcija i broj poziva. Dodatno, može da analizira upotrebu keš memorije i grananja na sličan način kao Cachegrind. Cachegrind meri događaje koji nastaju unutar same funkcije (**ekskluzivni troškovi**), dok Callgrind propagira troškove funkcija kroz sve njene pozive (**inkluzivni troškovi**). Zahvaljujući prikazu grafa poziva, mogu se pratiti putanja od početka rada programa i identifikovati funkcije sa najvećim ukupnim troškovima.

49. Dinamička detekcija grešaka. Sanitajzeri. Adresni i memorijski sanitajzer.

Alati za **dinamičku detekciju grešaka** koriste instrumentaciju kako bi omogućili automatsku detekciju grešaka u kodu, najčešće u radu sa memorijom i nitima. Značajno doprinose stabilnosti, bezbednosti i pouzdanosti softvera. Njihova glavna prednost je to što automatizuju proveru velikog broja potencijalno problematičnih situacija. To omogućava programeru da se fokusira na otklanjanje grešaka, umesto na njihovo traženje. Obično se primenjuju u ranim fazama razvoja i testiranja.

Sanitajzeri vrše automatsku instrumentaciju i drugačiju organizaciju memorije u fazi kompilacije, tako da se u fazi izvršavanja omogućava automatska detekcija grešaka. Koriste se samo u testnom okruženju, jer značajno povećavaju vreme izvršavanja i potrošnju memorije. Najpoznatiji sanitajzeri su adresni sanitajzer, sanitajzer memorije, sanitajzer niti i sanitajzer nedefinisanog ponašanja.

Adresni sanitajzer (ASan) je namenjen dinamičkoj detekciji grešaka u upravljanju memorijom. Instrumentuje program tokom kompilacije, ubacujući dodatne provere pri svakom pristupu memoriji i preuređuje način organizacije memorijskog prostora kako bi omogućio precizno otkrivanje grešaka. Dostupan je u okviru kompajlera GCC, Clang/LLVM, Xcode i MSVC. Detektuje greške kao što su čitanje ili pisanje izvan granica alocirane memorije, korišćenje memorije nakon njenog oslobađanja, dvostruko oslobađanje memorije i curenje memorije. Pored instrumentacije svih pristupa memoriji, alokacije memorije se proširuju **crvenim zonama** koje okružuju svaki blok memorije i služe za otkrivanje prekoračenja granica. Status svakog bajta memorije čuva se u posebnoj memoriji (**shadow memory**) koja omogućava efikasnu detekciju grešaka. Glavne prednosti alata ASan su njegova jednostavnost i pouzdanost - program se prevodi uz dodatak odgovarajuće opcije, a greške u pristupu memoriji se otkrivaju prilikom prvog nepravilnog pristupa. Ipak, programi instrumentovani ovim sanitajzerom zahtevaju značajno više memorije i sporije se izvršavaju.

Memorijski sanitajzer (MSan) je namenjen za otkrivanje korišćenja promenljivih kojima nisu inicijalizovane vrednosti u programima napisanim u jezicima poput C i C++. Korišćenje neinicijalizovanih podataka može dovesti do nepredvidljivog ponašanja i takve greške su često teške za detekciju. Alat se razvija u okviru LLVM projekta i može se koristiti kao opcija kompajlera Clang/LLVM. Instrumentuje program tokom kompilacije, ubacujući dodatne provere koje prate inicijalizovanost svake promenljive u memoriji. Održava poseban deo memorije (shadow memory) u kojoj pamti status svakog bajta korisničke memorije, označavajući da li je njegova vrednost inicijalizovana. Program prekida izvršavanje čim se otkrije čitanje neinicijalizovane vrednosti. Glavna prednost alata MSan je precizno otkrivanje ove klase grešaka koje su inače teško uočljive testiranjem ili debagovanjem. Ipak, njegova upotreba povećava potrošnju memorije i usporava vreme izvršavanja, čak i više nego alat ASan. Dodatno, zahteva da sve biblioteke sa kojima se program linkuje budu kompajlirane sa podrškom za ASan, jer u suprotnom može generisati lažno pozitivne rezultate.

50. Dinamička detekcija grešaka. Sanitajzeri. Sanitajzer niti i nedefinisanog ponašanja.

(Dinamička detekcija grešaka iz 49. pitanja)
(Sanitajzeri iz 49. pitanja)

Sanitajzer niti (TSan) je namenjen otkrivanju grešaka u višenitnim programima, odnosno detekciji nesinhronizovanog pristupa podacima. Ovakve greške je teško naći jer se često ispoljavaju samo u specifičnim scenarijima. Dostupan je u okviru kompajlera GCC i Clang/LLVM. Instrumentuje program tokom kompilacije i prati sve pristupe deljenim promenljivama tokom izvršavanja. Kada otkrije da dve ili više niti pristupaju istoj promenljivoj, a bar jedna od njih vrši upis, bez adekvatne sinhronizacije, sanitajzer prijavljuje grešku. Glavna prednost alata TSan je što otkriva klasu grešaka koje se inače teško otkrivaju. Ipak, programi instrumentovani ovim sanitajzerom se izvršavaju značajno sporije i potrošnja memorije je veća.

Sanitajzer nedefinisanog ponašanja (UBSan) je namenjen otkrivanju ponašanja koja su nedozvoljena ili nedefinisana prema standardu programskog jezika i mogu dovesti do ozbiljnih problema. Dostupan je u okviru kompajlera GCC i Clang/LLVM. Instrumentuje program tokom kompilacije i dodaje provere za razne vrste operacija koje mogu izazvati nedefinisano ponašanje kao što su prekoračenje opsega celobrojnih tipova, nedozvoljene konverzije, pristup nepravilno poravnatim memorijskim lokacijama, dereferenciranje null pokazivača ili operacija šiftovanja koja vodi do prekoračenja. Glavna prednost alata UBSan je mogućnost otkrivanja grešaka koje na određenim arhitekturama i u određenim uslovima prolaze bez vidljivih posledica, kao i to što ima minimalan uticaj na performanse u poređenju sa drugim sanitajzerima. Preporučuje se njegovo korišćenje zajedno sa drugim sanitajzerima kako bi se obuhvatio što širi spektar grešaka.

51. Dinamička detekcija grešaka. Valgrind i najpoznatiji Valgrind alati. Memcheck i Massif.

(Dinamička detekcija grešaka iz 49. pitanja)
(Valgrind iz 47. pitanja)
(Najpoznatiji Valgrind alati iz 48. pitanja)

Memcheck je najpoznatiji i najčešće korišćeni alat Valgrind platforme. Služi za detekciju memorijskih grešaka. Program koji se izvršava pod kontrolom ovog alata je mnogo sporiji u odnosu na samostalno izvršavanje, što je posledica dinamičke transformacije koda. Izlaz programa dopunjen je izveštajima koje generiše Memcheck i koji se ispisuju na standardni izlaz za greške. Za programe pisane u jezicima C i C++ Memcheck otkriva greške kao što su upisivanje podataka van opsega hipa ili steka, pristupanje oslobođenoj memoriji, neispravno oslobađanje memorije (npr. duplo oslobađanje ili neupareno korišćenje funkcija malloc/new i free/delete), curenje memorije, korišćenje neinicijalizovanih vrednosti i slično.

Massif je alat Valgrind platforme namenjen analizi korišćenja memorije u hip segmentu programa. Može da detektuje neke probleme koje Memcheck ne može. To su problemi kada memorije formalno nije izgubljena, pokazivač na nju postoji, ali se više nikada ne koristi tokom izvršavanja. Programi sa ovom vrstom curenja memorije bespotrebno zauzimaju resurse. Massif pruža detaljne informacije o tome koji su delovi programa odgovorni za alokaciju memorije, što olakšava pronalaženje uzroka problema.

52. Dinamička detekcija grešaka. Valgrind i najpoznatiji Valgrind alati. Helgrind i DRD.

(Dinamička detekcija grešaka iz 49. pitanja)
(Valgrind iz 47. pitanja)
(Najpoznatiji Valgrind alati iz 48. pitanja)

Helgrind i **DRD** su alati Valgrind platforme namenjeni otkrivanju grešaka u sinhronizaciji pri korišćenju POSIX modela niti. Oba alata imaju značajan broj preklapajućih detekcija, ali koriste različite algoritme za analizu izvršavanja i otkrivaju delimično različite tipove grešaka. Otkrivaju mrtvo blokiranje (deadlock) koje nastaje kao posledica pogrešnog redosleda zaključavanja i pristupa zajedničkoj promenljivoj bez adekvatne sinhronizacije. Alat DRD može da detektuje i zadržavanje katanca (lock-holding) i lažno deljenje (false sharing). Greške koje alati mogu otkriti su:

- **pogrešno otključavanje muteksa** - prosleđivanje nezaključanog, nevažećeg ili muteksa koji je zaključala druga nit.
- **nepravilno rukovanje zaključanim muteksom** - uništavanje nevažećeg ili zaključanog muteksa i dealokacija memorije koja sadrži zaključan muteks.
- **pogrešno korišćenje funkcije *pthread_cond_wait*** - prosleđivanje nezaključanog, nevažećeg ili muteksa koji je zaključala druga nit.
- **greške u radu sa barijerama *pthread_barrier*** - nevažeća ili dvostruka inicijalizacija i čekanje na objekat koji nikada nije inicijalizovan.