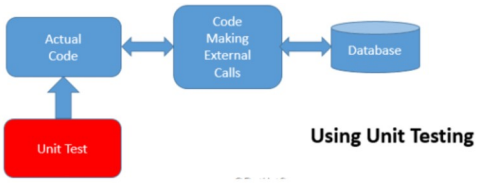
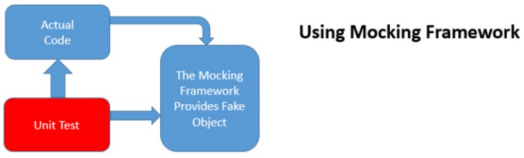
# Pitanja iz januarskog roka

## 1. Unit testovi

Jedinično testiranje je testiranje pojedinačnih komponenti softvera. Koristi se i naziv testiranje komponenti, testiranje modula, testiranje klasa. Treba da uporedi stvarnu i specificiranu funkcionalnost komponente. Osim analize funkcionalnosti, može da uključi i analizu izvornog koda komponente. Kod ovog tipa testiranja test objekti su pojedinačne softverske komponente (klasa najčešće, može da bude i uskladištena procedura, skript za bazu,…). Glavna karakteristika je da se softverska komponenta testira nezavisno i izolovano od ostatka sistema. Na ovaj način smo sigurni da verifikujemo ispravnost rada pojedinačne komponente, bez uticaja programskog koda u drugim komponentama. Detektovan problem onda definitivno ukazuje na nedostatak u testiranoj komponenti. Umesto pravih objekata koje klasa referencira, postavljaju se objekti dvojnici (*test doubles*) koji pojednostavljuju ili simuliraju ponašanje referenciranih objekata. Ovi objekti omogućuju da se test izvrši, a da bude fokusiran samo na objekat koji testira. Postoje različiti tipovi test dvojnika – Dummy Object, Test Stub, Test Spy, Mock Object i Fake Object.




Dummy objekat – objekat koji mora postojati u kodu, ali se nikada ne koristi. Najčešće postoji da popuni obaveznu listu parametara. Termin se koristi i za svaki objekat inicijalizovan podacima čiji sadržaj nije važan za test (npr. ako testiramo kreiranje nekog entiteta, sami podaci koje taj entitet sadrži nam nisu važni).

Lažni (*fake*) objekat – sadrži stvaran programski kod koji se izvršava. Jedino je zbog jednostavnosti ili brzine implementacija drugačija u odnosu na aplikaciju u produkciji i prilagođena je testiranju. Dobar primer je korišćenje *in-memory* baze podataka za potrebe testiranja.

*Stub* objekat – simulira izlaze stvarnog objekta. Na predefinisane ulaze daje predefinisane izlaze. Zna da reaguje samo na ulaze koje konkretan test predviđa.

*Mock* objekat – simulira ponašanje stvarnog objekta (za razliku od *stub* objekta koji simulira samo izlaze). Ako stvarni objekat sadrži sekvencu poziva određenih metoda, *mock* objekat ponavlja ovu sekvencu. Na ovaj način dobijamo dvojnika koji se i ponaša kao stvarni objekat.sami izlazi koji su rezultat ponašanja su i dalje simulirani.

*Spy* objekat – modifikovani stvarni objekat. Deo ponašanja je stvarno ponašanje. Deo ponašanja je simuliran zbog potreba testiranja. Korišćenjem *spy* objekata test može da dobija stvarno ponašanje jednog dela, a simulirano ponašanje drugog dela funkcionalnosti.

Način testiranja svodi se na pisanje programskog koda koji:

- poziva funkciju čiju funkcionalnost proveravamo i prosleđuje joj određene ulazne parametre
- proverava da li je rezultat funkcije očekivan za prosleđene ulazne parametre
    - rezultat može biti povratna vrednost funkcije ili

    - neki drugi efekt koji funkcija proizvodi (npr. pojava fajla na disku)

Za pisanje pojedinačnih testova programski kod testirane jedinice mora biti dostupan. Potrebno je poznavati interfejs i svrhu svake funkcije koja se testira. Iz ovih razloga jedinične testove najčešće pišu programeri.

Glavni cilj je testiranje funkcionalnosti komponente (kakve izlaze daje za zadate ulaze). Cilj se ostvaruje nizom test slučajeva od kojih svaki testira određenu kombinaciju ulaza/izlaza. Jedan deo test slučajeva moraju biti i negativni testovi koji proveravaju kako komponenta reaguje na nedozvoljene ili nepredviđene ulaze.

Strategija za jedinično testiranje najčešće podrazumeva uključivanje znanja o strukturi koda. Time ono spada u *white box* testiranje. Test slučajevi bi trebali biti napisani tako da pokriju različite putanje u programskom kodu. Nekada su jedinični testovi *black box* testiranje jer ne uzimaju u obzir strukturu testirane jedinice. Ako piše osoba koja nije programirala ili ne poznaje testiranu komponentu, ako je programski kod testa automatski generisan, ako se testovi prišu pre implementacije komponenti, kao kod *test-driven development* metodologije razvoja.

Jedinično testiranje u Javi vrši se pisanjem Java programskog koda. Koristi se *build-operate-check* šablon:

- Kreiranje test podataka – u Javi je to najčešće instanciranje objekata
- Operacija nad podacima – u Javi je to poziv funkcionalnosti objekta čije ponašanje testiramo
- Provera podataka nakon operacije – u Javi je to poređenje stanja objekata sa očekivanim stanjima

F.I.R.S.T. pravila za pisanje testova:

- Fast – testovi trebaju biti takvi da se brzo izvršavaju. Ako se ne izvršavaju brzo, neće biti često pokretani, što će smanjiti kvalitet softvera.
- Independent – testovi ne treba da zavise jedan od drugog. Jedan test ne treba da predstavlja preduslove za neki drugi test. Testovi trebaju moći da se izvršavaju nezavisno u proizvoljnom redosledu.
- Repeatable – testovi trebaju moći ponovo da se izvrše u različitim okruženjima. U svakom okruženju u kome se aplikacija razvija i koristi (produkcioni server, lokalni računar), test mora moći da se izvrši.
- Self-validating – test treba da ima jedan *boolean* rezultat (da li je prošao ili ne). Rezultat treba da se dobije izvršavanjem testa, a ne ručnim poređenjem ili subjektivnom procenom.
- Timely – testove treba pisati blagovremeno. Test treba napisati **pre** koda koji će biti testiran.

Postoje radni okviri koji pružaju pomoć u: pisanju koda za testiranje (sadrže biblioteke specijalizovane za testiranje), izvršavanju testova, izveštavanju o testovima. Najpopularniji radni okviri su Junit i TestNG.

Junit je radni okvir za testiranje Java programa. Poslednja verzija je Junit 5. Prema jednom istraživanju, Junit je najčešće uključivana eksterna biblioteka u Java projekte.

Najvažniji deo testa je verifikacija dobijenog rezultata (poređenje očekivane i stvarne vrednosti). Vrši se asertacijama. Asertacija je deo koda koji zahteva da određeni izraz bude istinit – ako to nije

slučaj, program se prekida ili se dešava izuzetak. Koristi se u testiranju pri utvrđivanju da li stvarna vrednost odgovara očekivanoj – ako to nije slučaj, test je neuspešan.

Junit predviđa različite funkcije za asertaciju: *assertTrue(), assertFalse(), assertEquals(), assertArrayEquals(), assertNull(), assertNotNull(), assertSame(), assertNotSame(), assertThat().*

Junit pruža mogućnost definisanja asertacija i putem *assertThat()* metode koja prima Matcher izraze. Prednosti ovog načina asertacije su: čitljivija sintaksa, asertacija se zadaje kroz subjekat, predikat i objekat; Matcher izrazi se mogu negirati i kombinovati konjukcijom ili disjunkcijom, čitljivije poruke o grešci kada test ne prolazi. Hamcrest projekat sadrži veliki broj Matcher izraza, najznačajniji su uključeni u Junit.

AssertJ pruža još fleksibilniji način definisanja asertacija koje koriste Matcher izraze. Različite metode za asertaciju koje vraćaju novi objekat nad kojim se može vršiti asertacija. Na ovaj način je obezbeđeno ulančavanje metoda za asertaciju. Može se koristiti IDE *code-completion* kao pomoć za korišćenje asertacije (zavisno od konkretnog objekta, različite metode za asertaciju su dostupne).

TestNG je drugi popularan radni okvir za testiranje Java programa. Razvijen je na osnovu Junit da obezbedi naprednije upravljanje testiranjem u odnosu na Junit. Junit verzije 4 je implementirao većinu funkcija koje je ranije samo TestNG obezbeđivao.

Testovi se mogu grupisati u Test Suite:

- U istom *suite* treba da se nađu logički srodni testovi
- Ovi testovi se onda mogu izvršavati u grupi.

Test Suite se definiše kao XML fajl. Sadrži niz testova koji se izvršavaju. Za svaki test se navode klase čije @Test metode test sadrži.

Test se može dodeliti određenoj grupi (kategoriji) i moguće je onda izvršiti testove samo iz jedne grupe. Dodeljivanje testa u grupu: *@Test(groups = {„postgres-test“})*. Jedan test može biti u više grupa. Ako se anotacija stavi nad klasom, onda svi testovi iz te klase pripadaju toj grupi. Izvršavanje određene grupe testova – u *test suite* se za test navodi na koje se grupe odnosi.

Moguće je pri izvršavanju testova određene metode izvršiti na odgovarajućim mestima u životnom ciklusu izvršavanja testova:

- @BeforeSuite, @AfterSuite – metoda se izvršava pre ili posle izvršavanja *suite*
- @BeforeMethod, @AfterMethod – metoda se izvršava pre ili posle svake @Test metode
- @BeforeTest, @AfterTest – metoda se izvršava pre ili posle *test* taga u *test suite*
- @BeforeGroups, @AfterGroups – metoda se izvršava pre ili posle izvršavanja određene grupe
- @BeforeClass, @AfterClass – metoda se izvršava pre prve ili nakon poslednje @Test metode u klasi

Test može da definiše da očekuje da se pri pozivu metode desi određeni izuzetak. Test prolazi ako se takav izuzetak desi. Ovo se koristi najčešće za negativne testove kada verifikujemo ponašanje na nevalidne ulaze. Može se i postaviti i da očekuje više izuzetaka (bilo koji izuzetak ako se desi, test prolazi).

Moguće je definisati maksimalno vreme za koje test mora biti izvršen da bi test bio uspešan (vreme se navodi u milisekundama). Koristi se kod testiranja performansi. Za određene testove se može definisati da se uopšte ne izvršavaju (upotrebom @Test(enabled = false) anotacije). Može se definisati da izvršavanje određenog testa zavisi od uspešnog izvršavanja drugih testova. Test se ne izvršava ako zavisni testi testovi nisu prošli. Zavisnost od pojedničanih testova: *@Test(dependsOnMethods = {„test1“, „test2“})*. Zavisnost od cele grupe: *@Test(dependsOnGroups = {„group1“, „group2“})*.

Testovi mogu u toku izvršavanja da dobijaju parametre. Ako se pošalje lista parametara, TestNG će kreirati više instanci testova, pri čemu će svaka instanca dobijati naredni iz liste. Dva načina prosleđivanja parametara: iz XML fajla, iz programskog koda putem @DataProvider anotacije.

Mocking mehanizam omogućuje simulaciju ponašanja objekata koje testirani objekat koristi:

- da bi se testirani objekat testirao u izolaciji, važno je da referencirani objekti ne unose grešku
- potrebno je simulirati da referencirani objekti uvek rade ispravno.

Umesto pravih objekata koje klasa referencira, postavljaju se objekti dvojnici (*test doubles*) koji simuliraju ponašanje referenciranih objekata.

Mockito – najpopularniji radni okvir za implementaciju *mocking* mehanizma u testiranju Java programa. Omogućuje kreiranje objekta dvojnika:

- objekat ima isti interfejs kao originalni
- metode se ne izvršavaju nad stvarnim objektom, nego vraćaju simulirani rezultat koji mi odredimo

*Mockito* sintaksno ne razlikuje *stub* od *mock* objekata. Objekti se kreiraju metodom *mock()*, kontekst korišćenja pravi razliku između ova dva tipa. *Mock* objekat se kreira tako što se statičkoj *mock()* metodi prosleđuje klasa čiji lažni objekat kreiramo. Za metode *mock* objekta se mora definisati ponašanje – koji rezultat određena metoda vraća za određene izlaze, metode za koje se ne navede lažno ponašanje vraćaju *null*. U terminologiji testiranja, ovo odgovara *stub* objektu. Pri izvršavanju testa, *mock* objekat će se ponašati kako je specificirano. Metoda *mock* objekta može da simulira izbacivanje određenog izuzetka za neke ulaze.

Osim *mock* objekata *Mockito* omogućuje i kreiranje *spy* objekata. *Spy* objekat je objekat kod koga se pozivaju stvarne metode, osim onih za koje je eksplicirno navedeno simulirano ponašanje. Kod *mock* objekta metode za koje se ne navode ponašanje vraćaju *null*, a kod *spy* objekta pozvaće se stvarna metoda.

*Mockito* omogućuje i tehnike *white-box* testiranja korišćenjem lažnih objekata. *Mockito* obezbeđuje informacije koliko puta je metoda lažnog objekta pozivana. Na ovaj način se, osim stvarnog rezultata (kao kod *black-box* testiranja) može proveriti i da li testirani objekat referencira druge objekte na odgovarajući način. Npr. ako smo putem *mock* objekta simulirali snimanje u bazu, možemo proveriti da li je metoda za snimanje u bazu pozvana tačno jednom. Ako se *mock* objekat ovako koristi, to odgovara *mock* objektu u terminologiji testiranja.

Koristi se *verify()* metoda kojoj prosleđujemo:

- koji *mock* objekat verifikujemo

- koju metodu tog objekta verifikujemo
- broj očekivanih poziva te metode

Broj očekivanih poziva se zadaje kao:

- konkretan broj – *times(3)*
- implicitno bez navođenja – tada se smatra da se zahteva jedan poziv
- nikad – *never()*
- minimalan/maksimalan broj poziva – *atLeastOnce(), atLeast(3), atMost(6)*

Privatnu metodu nije moguće pozvati direkrno iz testa. Moguća rešenja:

- Ne testirati – nije dobro jer je važno taj kod izolovano testirati
- Promeniti modifikator pristupa metodi – nije dobro jer je metoda sa razlogom privatna i ne treba menjati logiku enkapsulacije zbog testova
- Ugraditi test klasu unutar klase koja se testira – nije dobro, jer time kod za testiranje nije odvojen od produkcionog koda.
- Korišćenje refleksije – jedino prihvatljivo rešenje. Refleksijom se može pristupiti i privatnim metodama.

## 2. Angular testiranje

Potrebno je izvršiti jedinično testiranje Angular klijentske aplikacije. To podrazumeva testiranje svih komponenata u klijentskoj aplikaciji. Testovi treba da izolovano testiraju svaku komponentu.

Angular sadrži *Testing utilities* – skup klasa i funkcija koje olakšavaju testiranje. Omogućuju da testirana Angular komponenta komunicira sa Angular radnim okvirom. Test objekti se testiraju u Angular okruženju.

Neke testove je moguće realizovati i kao potpuno izolovane testove nezavisno od Angulara – ručno se instanciraju objekti, umesto da se injektuju od Angulara.

TestBed – klasa koja omogućuje da se test izvrši u Angular okruženju. Čini dostupnim objekte čijim životnim ciklusom upravlja Angular. *ConfigureTestingModule* metoda – uspostavlja inicijalno test okruženje. Pri inicijalizaciji se šalje *TestModuleMetadata* objekat koji predstavlja modul koji će test okruženje sadržati. *Inject* metoda – preuzima objekat iz okruženja. Uvek treba uzimati objekat iz okruženja umesto ručnog instanciranja jer druge komponente u okruženju koriste objekat iz okruženja, pa je samo korišćenjem tog objekta moguće imati ispravan test. *CreateComponent* metoda – kreira Angular komponentu unutar okruženja u skladu sa konfiguracijom okruženja, vraća objekat klase *ComponentFixture*.

ComponentFixture – omogućuje pristup Angular komponenti i njenoj DOM reprezentaciji. *ComponentInstance* atribut – instanca same komponente na koju se objekat odnosi. *DebugElement* atribut – objekat klase DebugElement koja predstavlja reprezentaciju elementa DOM stabla za potrebe testiranja. *DetectChanges* metoda – inicira poziv Angular ciklusa koji detektuje promene u objektima i sinhronizuje ih sa HTML komponentama. Ako test postavi vrednost u objekat, pre provere da li se ta vrednost nalazi u HTML komponenti, potrebno je pozvati *detectChanges*. Koristi se i za inicijalizaciju komponente jer automatski poziva *ngOnInit*. *WhenStable* metoda – vraća

*Promise* objekat koji se razrešava kada je *ComponentFixture* stabilan, tj. kada je ciklus detekcije promena završen uključujući završetak svih asinhronih aktivnosti u ciklusu. Koristi se da test sačeka da se promene detektuju i postave u komponente.

*DebugElement* – omogućuje testu pristup elementima HTML DOM stabla. *Query* metoda – omogućuje preuzimanje jednog elementa po nekom selektoru *debugElement.query(By.css(‚fieldX‘));*. *QueryAll* metoda – slično kao *query*, samo što vraća sve elemente koji odgovaraju selektoru. *NativeElement* atribut – predstavlja sam HTML element u browseru.

Testiranje HTTP komunikacije standardno se vrši postavljanjem test dvojnika na mesto komponente koja komunicira sa backendom. *HttpTestingController* se koristi kao dvojnik – ne vrši stvarne mrežne pozive, možemo simulirati i pratiti mrežne pozive. Praćenje HTTP poziva – proveravamo da li je poziv ka tom URL upućen i dobijamo *mock* objekat za simuliranje odgovora. *Const mockHttp = httpTestingController.expectOne(‚api/students‘); expect(mockHttp.request.method).toBe(‚GET‘);* Simuliranje HTTP odgovora vrši se metodom *flush* i specificiranjem lažnog odgovora.

Za izolovano testiranje komponente potrebno je postaviti test dvojnike na mesto objekata koje komponenta koristi. Možemo koristiti *Jasmine* podršku za *spy* objekte, oni omogućuju vraćanje predefinisanih izlaza na predefinisane ulaze i „špijuniranje“ da li su i kako objekti korišćeni. Za komplikovanije test dvojnike, možemo napraviti poseban objekat koji simulira stvarno ponašanje.

Komunikacija sa HTML komponentama:

- Da bi se inicirao *binding* podataka od objekta ka komponenti – *detectChanges* metoda iz klase *ComponentFixture*
- Da bi se inicirao *binding* od komponente ka povezanom objektu – *dispatchEvent* metoda nad *nativeElement* objektom u testu.

### 3. Tick funkcija

*Tick* funkcija – unutar *fakeAsync* funkcije možemo simulirati protok vremena dok god se sve trenutno nerešene asinhrone aktivnosti ne završe.

### 4. Asinhroni pozivi

*waitForAsync* funkcija omogućuje jednostavnije testiranje funkcionalnosti koje se izvršavaju asinhrono. Kod unutar *waitForAsync* funkcije se izvršava unutar specijalne asinhrone test zone. Ova funkcija presreće i evidentira sve *Observable* objekte kreirane unutar nje (koristi se za detekciju kada je *ComponentFixture* stabilan pozivom *whenStable* metode.

*FakeAsync* – dodatno pojednostavljenje u odnosu na *async* funkciju za testiranje asinhronog koda. Omogućuje linearno pisanje koda, bez reakcije na razrešavanje *Observable* objekta putem *subscribe* metode. *Tick* funkcija – unutar *fakeAsync* funkcije možemo simulirati protok vremena dok god se sve trenutno nerešene asinhrone aktivnosti ne završe. Nakon poziva funkcije, asinhrone aktivnosti su završene, pa ne mora kod da bude u funkciji *subscribe* na *Observable* objektu.

5. **JPA**

Kod testiranja serverskog dela Spring veb aplikacije, za testiranje JPA sloja možemo koristiti realnu bazu podataka. Druga varijanta je korišćenje testne *in-memory* baze:

- Postavlja se anotacija @DataJpaTest
- Koja konkretno baza podataka će biti korišćena zavisi od biblioteke koja je dodata u CLASSPATH aplikacije (ako koristimo Maven, dodamo je u *dependencies*). Komunikacija se odvija preko klase *TestEntityManager* koja se injektuje.

*@Autowired*

*private TestEnitityManager entityManager;*

6. **Test Entity Manager**

TestEntityManager pruža podskup metoda EntityManager-a koje su korisne za testove, kao i pomoćne metode za česte zadatke testiranja kao što su *persist* ili *find*. TestEntityManager omogućava korišćenje EntityManager-a u testovima. Spring Repository je apstrakcija iznad EntityManager-a; štiti programere od detalja na nižem nivou JPA i donosi mnogo praktičnih metoda. Ali Spring omogućava korišćenje EntityManager-a kada je potrebno u kodu aplikacija i testovima.

TestEntityManager je deo JPA (Java Persistence API) i koristi se u JPA testovima za simuliranje okruženja za upravljanje entitetima. (EntityManager). TestEntityManager pruža slične funkcionalnosti kao EntityManager, ali je dizajniran za upotrebu u testnom okruženju kako bi se olakšalo pisanje i izvođenje testova koji uključuju JPA. Evo nekoliko ključnih stvari koje TestEntityManager omogućava:

- **Simuliranje baze podataka** – TestEntityManager nam omogućava stvaranje i upravljanje priveremenom *in-memory* bazom podataka za potrebe testiranja. Ovo je korisno da bi se izbeglo povezivanje sa pravom bazom podataka tokom testiranja, što može biti sporije i komplikovanije.
- **Manipulacija entitetima** – možemo koristiti TestEntityManager za upis, dobavljanje, ažuriranje i brisanje entiteta iz privremene baze podataka unutar testa. To omogućava simulaciju različitih scenarija upisa i dobavljanja podataka
- **Proveravanje rezultata** – TestEntityManager omogućuje proveru očekivanih rezultata tokom testiranja. Na primer, možemo proveriti da li je određeni entitet upisan ili ažuriran u bazi podataka ili možemo proveriti rezultate upita.
- **Transakcije** – TestEntityManager podržava transakcije, što nam omogućava simuliranje trasnakcijskog ponašanja u testovima. Možemo započeti, potvrditi ili poništiti transakcije kako bismo testirali kako se naša aplikacija ponaša u različitim scenarijima.

Sve u svemu, TestEntityManager pruža alate i funkcionalnosti neophodne za jednostavno i učinkovito testiranje JPA entiteta i povezanih funkcionalnosti unutar naše aplikacije.

## 7. Jasmine

Jasmine je radni okvir za testiranje JavaScript koda. Omogućuje razčličite funkcije za evaluiranje vrednosti izraza, kao i rad sa test dvojnicima. Može se koristiti i u kombinaciji sa drugim radnim okvirima.

Jasmine konfiguracija:

- *npm install –save-dev jasmine*
- *npx jasmine init*
- U package.json dodati: *„scripts“: {„test“: „jasmine-browser-runner runSpecs“}*
- Startovanje testova: *npm test*

Jasmine suite – osnovna jedinica grupisanja testova je *suite*. Definiše se pozivom funkcije *describe*. Funkcija prima naziv *suite*-a i funkciju u kojoj se pišu testovi koji pripadaju tom *suite*

*describe(„Suite name“, function() {…*

Obzirom da se testovi pišu u običnoj JavaScript funkciji, ona može da sadrži proizvoljne promenljive koje su dostupne i testovima.

Jasmine testovi – koristi se termin *spec*. Jasmine je napravljen kao *behaviour-driven development* radni okvir. *Spec* treba da specificira očekivano ponašanje. Definiše se funkcijom *it* koja prima naziv *spec*-a i funkciju koja sadrži kod kojim se utvrđuje da li je specificirano ponašanje realizovano *it(„should do something“, function() {*. Naziv *suite-a* zajedno sa nazivom *spec*-a trebali bi da formiraju rečenicu prirodnog jezika kojom se objašnjava traženo ponašanje.

Jasmine Matchers omogućuju poređenje stvarnog i očekivanog ponašanja. Vraćaju izraz koji je tačan ili netačan. Ako *matcher* daje netačnu vrednost, test nije uspešno prošao. *expect(counter).toBe(2);* Podržane su različite matcher funkcije: poređenje vrednosti, pronalaženje vrednosti, očekivanje izuzetka…

Inicijalizacija i završetak testova – Jasmine omogućuje pisanje funkcija koje će se izvršavati pre ili posle svakog *spec*-a. Definišu se kao parametar Jasmine funkcija *beforeEach* i *afterEach.*

Ugnježdavanje testova – moguće je ugnježdavati *describe* blokove tako da se dobija struktura tipa stabla (prvi *describe* blok je koren, dok su *it* blokovi listovi). Na ovaj način se hijerarhijski pozivaju *beforeEach* i *afterEach* funkcije ukoliko postoje. Kreće se niz stablo od korena do lista i izvršavaju se svi *beforeEach* blokovi, zatim *it* blok, pa onda u obrnutom redosledu *afterEach* blokovi.

Jasmine omogućuje kreiranje *spy* objekta koji predstavlja test dvojnika. Ovaj objekat ima dvostruku ulogu: može da simulira ponašanje objekta, omogućuje praćenje operacija izvršenih nad objektom.

## 8. White box

Kod white box testiranja testovi se dizajniraju na osnovu uvida u strukturu test objekta (analizira se programski kod objekta). Testiranje analizira i tok programa unutar test objekta. Tačka posmatranja je unutar objekta. Tačka upravljanja može biti unutar objekta kada se objekat modifikuje za potrebe testiranja da bi se izvršio željeni tok programa.

Razlika između white box i black box testiranja:

- white box se koristi u testovima nižeg nivoa – jedinični i integracioni testovi

- black box se koristi za e2e testove, može i za integracione
- black box je isključivi način testiranja kod tehnika koje kreću razvoj programa od testova – *test-driven development*
- gray box tehnike su tehnike testiranja koje uključuju principe i white box i black box testiranja

Osnova za testiranje je izvorni kod test objekta (zbog toga se zove i testiranje zasnovano na strukturi ili kodu). Uzima u obzir koji delovi izvornog koda test objekta se izvršavaju pri testiranju. Dobar test bi trebao da izvrši sve delove izvornog koda bar jednom. Pri dizajnu test slučajeva se uzima u obzir koji delovi koda će se izvršiti pri pokretanju test slučajeva. Očekivani rezultat se određuje na osnovu specifikacije zahteva, a ne analizom koda jer kod može da ima grešku, a upravo je svrha testiranja da se te greške pronađu.

Tehnike white box testiranja:

- Testiranje naredbi
- Testiranje grana
- Testiranje uslova
  - jednostavno testiranje ulsova
  - višestruko testiranje uslova
  - minimalno višestruko testiranje uslova
- Testiranje putanja

Za lakšu analizu izvornog koda, kod se transformiše u graf toka programa. Naredbe programa su čvorovi grafa. Tok programa je predstavljen granama grafa. Sekvenca naredbi koje bezuslovno slede jedna drugu se predstavlja kao jedan čvor.

**Testiranje naredbi** – analizira se svaka naredba u izvornom kodu test objekta. Test slučajevi treba da izvrše određeni procenat svih naredbi (ili sve naredbe). Pokrivenost testa (*coverage*) utvrđuje koliko je naredbi izvršeno testovima. Naziva se i C0 (C-zero) pokrivenost. Foksuiranje samo na pokrivenost svih naredbi ne pokriva dovoljno moguće tokove programa (ne pokriva izvršavanje grana koje nemaju naredbe). Ipak, čak i za ove testove je teško obezbediti 100% pokrivenost. Neke izuzetke u kodu je teško izazvati.

**Testiranje grana** – naprednija tehnika od testiranja naredbi. Analiziraju se grane u grafu toka programa. Zavisno od uslovnih izraza, prati se koje grane toka programa se izvršavaju. Pokrivenost se određuje u odnosu na procenat grana izvršenih u okviru testa. Naziva se i C1 pokrivenost. Za razliku od testiranja naredbi, potrebno je pokriti i grane bez naredbi (npr. if naredbu bez else dela). Pri dizajnu test slučajeva vodi se računa o svim granama toka programa. Neke grane se izvršavaju i više puta – jeste redundantno, ali je neophodno jer različiti tokovi imaju neke zajedničke delove. Predstavlja stroži kriterijum u odnosu na testiranje naredbi – obično zahteva kreiranje više test slučajeva, može da utvrdi nedostajuće naredbe u praznim granama, oštrija tehnika od testiranja naredbi (potpuna pokrivenost grana garantuje i potpunu pokrivenost naredbi, ali ne važi obrnuto). Testiranje grana ne razlikuje koliko puta se grana izvršava. Pokrivenost se meri samo na osnovu toga da li je grana uopšte izvršena.

**Testiranje uslova** – analiziraju se konkretni uslovni izrazi koji uzrokuju prelazak toka programa na određenu granu. Za razliku od testiranja grana koje se fokusira na proveru grana, testiranje uslova se bavi time šta dovodi do toga da program pređe u određenu granu. Uslov može biti složen – više prostih uslova povezanih logičkim operatorima. Prost uslov je onaj koji daje logičku vrednost kao rezultat i ne sadrži logičke operatore, sadrži relacione operatore koji daju logički rezultat.

*Jednostavno testiranje uslova* – kreiraju se testovi tako da se za svaki prost uslov proveri ponašanje programa za oba moguča rezultata (i *true* i *false*). Uslov jednostavnog testiranja uslova je da su oba prosta rezultata pri testiranju imala i rezultate *true* i *false*. Ovakvo testiranje će proveriti samo jednu od dve moguće grane toka programa:

- ovakav kriterijum testiranja je blaži u odnosu na testiranje grana jer ne mora proveriti sve grane
- ne gleda se rezultat kompletnog uslova, nego samo rezultati prostih uslova

*Višestruko testiranje uslova* – zahteva da testovi provere sve moguće kombinacije rezultata prostih uslova, za razliku od jednostavnog testiranja uslova koje ne gleda kombinacije, nego samo rezultat pojedinačnog prostog uslova. Ovaj kriterijum testiranja sigurno pokriva sve moguće grane. Ako proverimo sve moguće kombinacije uslova, svaka grana će biti pokrivena nekom od kombinacija. Ovo je najstroži kriterijum, ali težak za implementaciju jer za komplikovanije uslove postoji prevelik broj mogućih kombinacija ($2^n$ ako je n broj prostih uslova). Nekada ne postoje ulazni podaci za određene kombinacije uslova.

*Minimalno višestruko testiranje uslova* – optimizacija višestrukog testiranja uslova. Ne proveravaju se sve moguće kombinacije prostih uslova. Testovi uključuju samo one moguće kombinacije u kojima promena rezultata prostog uslova menja rezultat složenog uslova:

- time se verifikuje da li je prost uslov dobro implementiran
- ako nije dobro implementiran, složeni uslov će dati drugačiji rezultat

Potrebno je manje test slučajeva nego kod klasičnog višestrukog testiranja uslova. Pri dizajnu test slučajeva analizira se koji ulazni podaci daju koje rezultate i kombinacije prostih uslova. Pokrivenost se računa kao procenat izvršenih u odnosu na sve moguće logičke vrednosti uslova. Preporučeni način testiranja uslova je kroz minimalno višestruko testiranje uslova:

- uzima u obzir kompleksnost uslova
- razuman broj test slučajeva
- ne daje manju pokrivenost od testiranja izraza i grana.

Voditi računa da uslov može da sadrži logičke promenljive koje su ranije u kodu dobile rezultat nekog složenog uslovnog izraza – test ne bi trebao da gleda samo uslov u kojem se rezultat promenljive koristi, nego i ranije dodele vrednosti logičkim promenljivim. Voditi računa o načinu evaluacije uslova pri izvršavanju programa. Ne proverava se prost uslov čiji rezultat ne utiče na konačan rezultat uslova (npr. drugi operand u konjukciji, ako je prvi *false*).

**9. Šta je test bed (test okruženje) i kako on izgleda u tehnologijama koje smo radili (Junit/TestNG, Angular, Selenium, Spring)?**

Za Spring pričati o kontekstu, test rest template-u, test entity manager-u i profilima.

Test bed (ili testno okruženje) je skup resursa, konfiguracija i softverskih alata koji su postavljeni kako bi podržali izvođenje testova. Test bed omogućuje razvojnim timovima da izvrše testiranje softvera u kontrolisanom i predvidivom okruženju, što pomaže u otkrivanju grešaka i nedostataka u kodu pre nego što se softver pusti u produkcijsko okruženje.

Evo kako test bed izgleda u nekima od tehnologija koje smo radili:

- Junit/TestNG:
  - Test bed u Junit ili TestNG obično uključuje konfiguraciju testnih klasa, metoda i paketa.
  - Koristi se za pisanje i izvođenje Unit testova u Java aplikacijama.
  - Uključuje biblioteke za asertacije (provera očekivanih rezultata), upravljanje testovima i generisanje izveštaja.
- Angular:
  - Test bed u Angular aplikacijama može uključivati konfiguraciju za pokretanje testova komponenti, usluga, direktiva, itd.
  - Koristi se za pisanje i izvođenje testova korisničkog interfejsa, integracionih testova i jediničnih testova u Angular aplikacijama.
  - Uključuje alate poput Karma test runner-a, Jasmine za pisanje testova, te alate za mocking i simulaciju HTTP zahteva.
  - TestBed je klasa koja omogućuje da se test izvrši u Angular okruženju. Čini dostupnim objekte čijim životnim ciklusom upravlja Angular.
  - *ConfigureTestingModule* metoda uspostavlja inicijalno test okruženje. Pri incijalizaciji se šalje TestModuleMetadata objekat koji predstavlja modul koji će test okruženje sadržati.
  - *Inject* metoda preuzima objekat iz okruženja. Uvek treba preuzimati objekat iz okruženja umesto ručnog instanciranja jer druge komponente u okruženju koriste objekat iz okruženja, pa je samo korišćenjem tog objekta moguće imati ispravan test.
  - *CreateComponent* metoda kreira Angular komponentu unutar okruženja u skladu sa konfiguracijom okruženja. Vraća objekat klase *ComponentFixture*.
- Selenium:
  - Test bed u Seleniumu uključuje konfiguraciju WebDrivera, koje je suštinski interfejs za automatizaciju web browsera.
  - Koristi se za pisanje i izvođenje testova funkcionalnosti web aplikacija.
  - Uključuje biblioteke ili okvire za pisanje testova (poput TestNG ili Junit za Java, ili slične za druge jezike), kao i Selenium WebDriver API za interakciju sa web elementima.
- Spring:
  - Test bed u Spring aplikacijama može uključivati konfiguraciju Spring kontejnera, beanova i drugih resursa potrebnih za izvođenje testova.

- Koristi se za pisanje i izvođenje integracionih testova, unit testova i testova komponenti Spring aplikacije.
- Uključuje biblioteke ili okvire za kreiranje Spring aplikacija (poput Spring Test Framework-a), kao i alate za simulaciju spoljnih servisa ili baza podataka (korišćenjem mockova ili stvaranjem privremenih baza podataka za testiranje).

U svakom od ovih okvira i tehnologija, test bed je ključni deo procesa testiranja koji omogućava razvojnim timovima da provere ispravnost i pouzdanost svojih aplikacija pre nego što se one dostave korisnicima.

**10. Razlika između manuelnih i automatskih testova.**

Statičko testiranje podrazumeva verifikaciju test objekta analizom njegovog sadržaja, za razliku od dinamičkog testiranja koje verifikaciju vrši izvršavanjem test objekta korišćenjem određenih test podataka. Može biti:

- manuelno – učesnici u programu vrše analizu programa
- automatsko – softverski alati vrše analizu programa, moguće samo za mašinski čitljive test objekte sa formalnom strukturom.

Cilj statičkog testiranja je pronalaženje nedostataka u funkcionalnostima, ali i dizajnu aplikacija. Rano otkrivanje ovih nedostataka smanjuje troškove kasnijih aktivnosti u životnom ciklusu aplikacije. Statičkim testiranjem se može utvrditi da arhitektura nije proširiva, specifikacija zahteva nije ispoštovana, da implementacija nije usklađena sa standardima… Dodatno, neki nedostaci u programu se teško otkrivaju dinamičkim testiranjem, npr. nedeterministička ponašanja, konkurentni problemi, *data race*.

Manuelna evaluacija odnosi se na analiziranje dokumenata u projektu od strane učesnika. Dokument je najčešće sam kod, ali može biti i specifikacija zahteva, model sistema, model podataka, plan aktivnosti… Različite tehnike analize razlikuju se po intenzitetu, formalnosti, potrebnim resursima i ciljevima. Ne postoji jedinstvena terminologija za ove različite tehnike analize.

*Pregled* je generalni termin koji se ovde koristi za sve tehnike manuelnog statičkog testiranja. U praksi se najčešće koristi *review* ili *inspection*. U pregledu najčešće učestvuje više ljudi (*peer review*). Moguće prednosti pregleda su da kao i svako otkrivanje nedostataka, poboljšava dalji proces razvoja (što se ranije nedostaci utvrde, tim bolje), kvalitetnije rešenje jer je u razvoj rešenja uključeno više ljudi, učesnici uče jedni od drugih, formalizacija i dokumentovanje rešenja (autor mora da iznese rešenje tako da drugim učesnicima bude čitljivo i razumljivo, kao i da obrazloži odluke donete pri razvoju), odgovornost se raspoređuje na sve učesnike. Iako troši resurse, pregled je neizostavan u razvoju jer značajno smanjuje broj kasnijih nedostataka i troškove otklanjanja. Mogući problemi u pregledu su uglavnom psihološke prirode, autor može imati utisak da je predmet analize i kritike on i njegov rad, a ne dokument. Problemi se mogu ublažiti jasnim ciljem i dobrim upravljanjem svakim pregledom i dobrim izborom učesnika u pregledu. Iako postoje različiti tipovi pregleda, obično postoje sledeće faze (implicitno ili eksplicitno izražene):

- planiranje
- informisanje

- samostalna priprema
- sastanak
- ispravke
- revizija

Planiranje pregleda može biti na generalnom nivou ili na nivou jednog pregleda. Na generalnom nivou, na nivou projekta se mora doneti odluka koji dokumenti se pregledaju, mora se u plan projekta uvrsiti i dinamika i troškovi tih pregleda. Na nivou jednog pregleda treba pripremiti dokumente koji su predmet pregleda, izabrati učesnike u pregledu, obezbediti da je dokument spreman za pregled (da je završen na predviđenom nivou za tu fazu projekta), odrediti mesto i vreme sastanka i druge detalje potrebne za održavanje pregleda. Pre održavanja sastanka potrebno je: obavestiti učesnike o održavanju, temi i cilju pregleda, obezbediti učesnicima pristup dokumentima koji su tema pregleda, dati učesnicima dodatne informacije o dokumentima, ukoliko nisu potpuno upućeni. Ovo je moguće realizovati pisanim pozivom ili kratkim grupnim sastankom. Svaki učesnik bi trebao da se pripremi za sastanak analizom dokumenata koji su tema pregleda i trebao bi da pripremi komentare, zamerke i pitanja za druge učesnike. Sastanak je centralna aktivnost u pregledu. Istovremeno je i najveći izazov – potrebno je doneti odluke kroz diskusiju učesnika koji imaju različite stavove (nekad čak i ciljeve). Neophodno je sastanak unapred vremenski limitirati da bi dao rezultat. Važno je da jedna osoba vodi sastanak, ovo obezbeđuje da sastanak ne odstupi od cilja i da komunikacija bude konstruktivna i sa što manje konflikta, korisno je voditi beleške. Sastanak se mora završiti zaključkom – donosi se odluka da li se dokument prihvata, ispravlja ili odbacuje. Nakon sastanka ili na samom sastanku donosi se odluka o izmenama u pregledanom dokumentu, najčešće je autor odgovoran da izvrši dogovorene izmene u dokumentu. Nakon ispravki neophodno je izvršiti reviziju nove verzije dokumenta. Mora se obaviti novi pregled, ali procedura drugog pregleda može biti nešto kraća sa fokusom samo na modifikovane delove. Potrebno je evaluirati sam proces pregleda kako bi naredni pregled ispravio nedostatke u proceduri i identifikovati faktore koji mogu naredne pregled učiniti korisnijim i efikasnijim. Možemo identifikovati nekoliko klasa učesnika:

- Menadžer:
  - Upravlja procesom verifikacije sistema.
  - Upravlja pregledima na nivou generalnog plana i mesta pregleda u dinamici projekta.
  - Ne mora da učestvuje u pregledima jer često nije upućen u tehničke detalje koji se na pregledima analiziraju.
- Moderator:
  - Zadužen je za sprovođenje pregleda.
  - Vodi sve faze pregleda.
  - Evidentira dokumentaciju u pregledu.
  - Potrebno je da ima društvene veštine zbog usklađivanja učesnika u pregledu.
  - Trebao bi da zauzme neutralan stav na samom pregledu.
- Autor:

    - Osoba koja je kreirala dokument koji je predmet pregleda.
    - Kod timskog rada, može biti delegiran jedan od od autora da učestvuje u pregledu ili prisustvuju svi.
    - U pravilu najteže prihvata pregled jer se njegov rad evaluira.
- Recenzenti:
    - Osobe koje vrše pregled dokumenta.
    - Identifikuju nedostatke u dokumentu.
    - Moraju da imaju tehničko znanje.
    - Mogu predstaviti različite aspekte projekta (ekonomski aspekt, arhitekturu, dizajn, zahteve,…).
- Beležnik:
    - zadužen da evidentira informacije iz pregleda (probleme, stavove, odluke, zaključke)

U praksi često isti ljudi preuzimaju različite tipove uloga (npr. menadžer je istovremeno i moderator, ali i recenzent koji aktivno učestvuje u donošenju odluka u pregledu. Takođe, za vođenje beležaka može biti zadužen neko od članova tima (važno je da svi imaju uvid u zaključke pregleda).

Razlikujemo dve grupe pregleda:

- pregledi namenjeni analizi karakteristika projekta – da li su funkcionalni i nefunkcionalni zahtevi realizovani
- pregledi namenjeni analizi samog procesa razvoja projekta – analiziraju kvalitet upravljanja projektom, ispunjenost plana projekta i sl.

Iako nema opšteprihvaćene klasifikacije, u nastavku je jedna moguća klasifikacija tipova pregleda namenjenih analizi karakteristika projekta.

Prolazak kroz dokument (*walkthrough*) je neformalan pregled sa ciljem nalaženja nedostataka u dokumentu, vrši se kroz grupni sastanak. Cilj je i informisanje učesnika o delovima projekta. Vrši se bez posebne pripreme i planiranja. Nema formalnu proceduru, svodi se na razmenu mišljenja sa kolegama, brzo i lako se organizuje.

Inspekcija je najviše formalan tip pregleda. Prati unapred određenu proceduru: uloge su striktno podeljene, recenzenti imaju spisak stavki koje proveravaju. Pregled fokusiran na predefinisane aspekte. Cilj je usko postavljen na pronalaženje nedostataka u dokumentu. Prikupljeni podaci se formalno evidentiraju i mogu se koristiti za poboljšanje procesa razvoja i naredne inspekcije.

Tehnički pregled je fokusiran na tehničke aspekte sistema: usklađenost sa specifikacijom, ispunjenost svrhe zbog koje se razvija, usklađenost sa standardima. Osnov za pregled je isključivo zvanična specifikacija zahteva. Recenzenti poseduju tehničko znanje. Procedura može biti više ili manje formalno sprovedena. Kod formalnog tehničkog pregleda, svi rezultati se formalno evidentiraju.
Svaki pregled dokumenta od strane osobe koja nije autor može se smatrati neformalnim pregledom. Različite faze pregleda su najčešće samo implicitno izražene. Ne mora da uključuje sastanak.

Izbor tipa pregleda zavisi od konkretnog slučaja. Neke smernice za izbor:

- da li rezultati trebaju biti formalno dokumentovani
- koliko je komplikovano organizovati grupni lični sastanak sa učesnicima
- koliko tehničkog znanja je potrebno da učesnici poseduju
- kakav je odnos resursa potrebnih za pripremu pregleda i koristi od rezultata pregleda
- da li je predmet pregleda formalno specificiran da bi mogao biti analiziran programski
- koliko je menadžment projekta obezbedio resursa za preglede.

Cilj programske statičke analize je pronalaženje nedostataka, ali i generalno unapređenje kvaliteta sistema (isto kao i kod manuelne analize). Ovog puta analizu vrši programski alat. Većina alata koji se koriste pri razvoju aplikacije (editori koda, grafički alati za modelovanje, …) vrše analizu ispravnosti dokumenata – korisnik dobija komentare, upozorenja i greške. Koristi se u kombinaciji sa manuelnim pregledima. Pre pregleda, dokument treba biti programski pregledan kako bi se smanjio broj nedostataka za ručnu analizu. Programska analiza ima svoja ograničenja – većina nedostataka se može uočiti samo dinamičkim testiranjem (izvršavanjem test objekta). Dokument mora biti mašinski čitljiv i imati strukturu. I pored stalnog usavršavanja, neke nedostatke i dalje može da utvrdi samo čovek. Ovo se posebno odnosi na to kada se sa nivoa tehničkih detalja pređe na širu sliku (dizajn šabloni, arhitektura, proširivost, lakoća održavanja, …). Kompajler je najznačajniji alat za programsku analizu koda – prepoznaje sintaksne greške, ali i druge nedostatke (nekorišćene promenljive, nedostupan kod, …). Pored kompajlera, koriste se i posebni alati, tzv. analizatori. Programskom statičkom analizom mogu se utvrditi: sintaksni nedostaci, odstupanja od konvencija, pravila i standarda (pravila su često ugrađena u IDE za razvoj programa), nedostaci u kontroli toka programa, nedostaci u toku podataka u programu.

Analiza toka programa utvrđuje nepravilnosti u toku programa. Izgrađuje graf toka programa. Može da:

- utvrdi da program ni u jednom scenariju neće izvršiti određeni deo koda (*unreacahble code*)
- utvrditi da tok programa nije u skladu sa konvencijama, npr. više tačaka izlaza iz funkcije
- identifikuje potencijalne neželjene greške (npr. nedostatak *break* izraza u *case* labeli *switch* izraza).

Programski se može proceniti i kompleksnost koda:

- prebrojavanjem broja linija koda
- izračunavanjem ciklomatskog broja.

Ciklomatski broj daje informaciju o kompleksnosti strukture koda (na osnovu broja mogućih grana u izvršavanju koda). Računa se na osnovu grafa toka programa formulom broj grana u grafu, broj čvorova u grafu.

SonarQube je alat za programsku analizu kvaliteta koda koji prepoznaje odstupanje od: standarda i konvencija, potencijalne nedostatke, kompleksnost koda, dupliranje koda. Podržava različite programske jezike, uključujući i Java i JavaScript.

SonarLint je Eclipse plugin za SonarQube. Omogućuje prikaz osnovnih rezultata Sonar analize u Eclipse IDE. Potrebno je imati instaliran SonarQube server i povezati se na njega u konfiguraciji SonarLint u Eclipse.

11. **Kako bismo razdelili koje testove bismo radili manuelno, a koje automatski u nekom projektu?**

Razlikovanje između testova koji bi trebalo da se rade manuelno i onih koji bi trebalo da se izvršavaju automatski u projektu zavisi od različitih faktora, uključujući vrstu aplikacije, složenost, resurse tima, prioritete i druge faktore. Evo nekoliko smernica koje mogu pomoći u donošenju odluke:

Testovi koji se obično rade manuelno:

1. **Istraživačko testiranje**: Testiranje koje zahteva ljudsku kreativnost i intuiciju, poput istraživanja korisničkog iskustva, provere vizuelnog dizajna ili otkrivanja potencijalnih grešaka u korisničkim scenarijima.

2. **Upotrebljivost i korisničko iskustvo**: Testiranje korisničkog interfejsa, korisničkog iskustva i usklađenosti sa dizajnom, kao i provođenje korisničkih testova sa stvarnim korisnicima.

3. **Testiranje na različitim uređajima i pregledačima**: Testiranje kompatibilnosti sa različitim uređajima (računari, tableti, mobilni telefoni) i pregledačima.

4. **Testiranje lokalizacije i internacionalizacije**: Provera lokalizacije (jezičke podrške) i internacionalizacije (prilagođavanje kulture).

5. **Performanse i opterećenje**: Testiranje performansi aplikacije pod opterećenjem, kao i praćenje performansi u stvarnom vremenu.

6. **Bezbednosno testiranje**: Testiranje bezbednosti aplikacije, identifikovanje ranjivosti i provere sigurnosnih mehanizama.

Testovi koji se obično izvršavaju automatski:

1. **Unit testovi**: Testiranje pojedinačnih komponenti i funkcija kako bi se osigurala tačnost i ispravnost njihovog ponašanja.

2. **Integracioni testovi**: Testiranje integrisanog sistema ili više komponenti kako bi se proverila njihova interoperabilnost i ispravno funkcionisanje.

3. **Regresioni testovi**: Automatizacija testova koji proveravaju da li su promene u kodu uzrokovale nove greške ili su promenile postojeće funkcionalnosti.

4. **Testiranje API-ja:** Automatizacija testova koji proveravaju funkcionalnost API-ja i tačnost odgovora.

5. **Testiranje performansi**: Automatizacija testova koji prate performanse aplikacije tokom vremena i identifikuju probleme performansi.

6. **Code coverage testovi**: Automatizacija testova koji prate pokrivenost koda testovima i identifikuju delove koda koji nisu pokriveni testovima.

Ova podela nije stroga i može se prilagoditi specifičnostima vašeg projekta. Idealno je kombinovati oba pristupa tamo gde je to potrebno kako biste osigurali adekvatan nivo testiranja i kvaliteta vaše aplikacije.

12. **Kako funkcioniše JaCoCo (tačno opisati kako odredi *coverage*)?**

JaCoCo je biblioteka za merenje pokrivenosti Java koda testovima. Meri pokrivenost naredbi (C0) i grana (C1), metoda i klasa. Potrebno je dodati JaCoCo u Maven zavisnosti. Potrebno je konfigurisati da se pri izgradnji projekta aktivira JaCoCo plugin. Pokretanjem Maven izgradnje pokreće se i JaCoCo plugin koji generiše izlazni JaCoCo fajl sa podacima o pokrivenosti koda – mvn clean test, dobija se target/jacoco.exec. Ovo je binarni fajl koji je potrebno procesirati nekim alatom da bismo dobili podatke čitljive za čoveka. Sonar može da procesira JaCoCo izlazne fajlove. Potrebno je pokrenuti sonar cilj korišćenjem maven alata mvn sonar:sonar. Kroz Sonar GUI se mogu videti podaci o pokrivenosti koda (na http://localhost:9000 za lokalno instalirani Sonar).

JaCoCo, skraćenica od Java Code Coverage, je alat za merenje pokrivenosti koda koji se često koristi u Java razvoju. On funkcioniše tako što instrumentira Java bajt kod kako bi pratio koje linije koda su izvršene tokom izvršavanja testova. Ovaj alat može da prati pokrivenost linija koda, grananja, instrukcija i metoda. Evo kako JaCoCo određuje pokrivenost koda:

- **Instrumentacija koda**: Pre izvršavanja testova, JaCoCo intrumentira Java bajt kod. To znači da umeće dodatni kod (instrumentaciju) u izvorni kod koji prati izvršavanje svake linije koda, grananja, instrukcije i metode.
- **Izvršavanje testova**: Nakon instrumentacije, testovi se izvršavaju kao i obično. Tokom izvršavanja, JaCoCo prati koje delove koda su izvršeni, koristeći dodatni instrumentirani kod.
- **Generisanje izveštaja**: Nakon izvršavanja testova, JaCoCo generiše izveštaj o pokrivenosti koda. Ovaj izveštaj sadrži detaljne informacije o tome koliko je svaka linija koda (ili grananja, instrukcije, metode) izvršena tokom testiranja. Na osnovu ovih informacija, JaCoCo izračunava procenat pokrivenosti za različite delove koda.
- **Pokrivenost koda**: Konačno, JaCoCo pruža različite metrike pokrivenosti koda, uključujući:
  - **Pokrivenost linija koda**: Procenat linija koda koje su izvršene tokom testiranja.
  - **Pokrivenost grananja**: Procenat grana (uslovnih izraza) koje su izvršene.
  - **Pokrivenost instrukcija**: Procenat instrukcija koje su izvršene.
  - **Pokrivenost metoda**: Procenat metoda koji su pozvani tokom testiranja.

  Ove metrike omogućavaju programerima da procene koliko dobro su testovi pokrili njihov kod i da identifikuju delove koda koji nisu dovoljno testirani ili potpuno nepokriveni.

**13. Šta uvodi novo Angular testiranje?**

Angular framework uvodi nekoliko novih pristupa i alata za testiranje, koji olakšavaju razvoj, održavanje i izvršavanje testova za Angular aplikacije. Evo nekih od glavnih novina u Angular testiranju:

1. **Angular Test Bed**: Angular Test Bed je sredstvo za konfigurisanje i izvršavanje testova za Angular komponente, direktive i servise. Omogućava simuliranje Angular okruženja u kojem se komponente izvršavaju, što olakšava izolovano testiranje komponenti.

2. **Jasmine i Karma**: Angular koristi Jasmine kao osnovni okvir za pisanje testova. Jasmine je fleksibilan i moćan okvir koji podržava pisanje čistih, izražajnih testova. Karma je alat koji se koristi za automatsko pokretanje testova u različitim pregledačima i okruženjima, čime se osigurava konzistentnost i pouzdanost testova.

3. **Angular CLI Integration**: Angular CLI (Command Line Interface) pruža integraciju za kreiranje, konfigurisanje i izvršavanje testova. Može automatski generisati osnovne testove za nove komponente ili servise, olakšavajući tako početak sa pisanjem testova.

4. **Mocking Dependencies**: Za testiranje komponenti koje zavise od drugih komponenti ili servisa, Angular omogućava korišćenje mock-ova kako bi se simulirali ovi zavisni delovi. Ovo olakšava izolovano testiranje komponenti i povećava brzinu izvršavanja testova.

5. **Protractor za End-to-End (E2E) Testiranje**: Protractor je alat za automatizaciju E2E testiranja Angular aplikacija. On koristi Selenium WebDriver za simuliranje interakcija korisnika sa aplikacijom u stvarnom pregledaču, omogućavajući testiranje kompletnih putanja kroz aplikaciju.

Ove funkcionalnosti zajedno čine Angular testiranje efikasnim i moćnim alatom za razvoj visoko kvalitetnih Angular aplikacija.

### 14. Statičko automatsko vs statičko manuelno testiranje

Statičko automatsko testiranje i statičko manuelno testiranje su dva različita pristupa u testiranju softvera. Evo kako se oni razlikuju:

1. **Statičko automatsko testiranje**:

- **Automatizacija**: U ovom pristupu, alati i skripte se koriste za izvršavanje testova bez direktnog učešća korisnika. Ovi testovi se izvršavaju na izvornom kodu ili na kompajliranom kodu softvera.

- **Prednosti**: Automatizacija može biti brža i konzistentnija od ručnog testiranja. Može se lako integrisati u kontinuirane procese integracije i isporuke (CI/CD).

- **Ograničenja**: Automatski testovi mogu biti teži za postavljanje i održavanje, posebno za promene u softveru. Nisu svi testovi pogodni za automatizaciju, kao što su testiranja korisničkog interfejsa sa složenom logikom.

2. **Statičko manuelno testiranje**:

- **Ručno izvršavanje**: U ovom pristupu, testovi se izvršavaju ručno od strane ljudskih korisnika koji pregledaju softverske artefakte (kôd, dokumentaciju, korisnički interfejs itd.) kako bi identifikovali probleme ili greške.

- **Prednosti**: Ručno testiranje može biti fleksibilno i intuitivno, posebno za istraživačko testiranje gde se traže neočekivane greške. Može biti korisno za delove softvera gde je teško ili nemoguće automatizovati testove.

- **Ograničenja**: Ručno testiranje može biti sporo i skupo, posebno za ponovljive zadatke. Postoji veća mogućnost ljudske greške. Nije lako pratiti napredak i rezultate testiranja.

U praksi, često se koristi kombinacija oba pristupa. Automatsko testiranje se koristi za rutinske, ponovljive testove koji mogu biti lako automatizovani, dok se ručno testiranje koristi za složenije scenarije, istraživačko testiranje i za proveru korisničkog iskustva. Ova kombinacija omogućava efikasno i sveobuhvatno testiranje softvera.

15. **Kako JaCoCo izvršava *code coverage*, da li to radi statički ili dinamički, kako on zna šta je *cover-ovao*?**

JaCoCo izvršava code coverage dinamički. To znači da prati izvršavanje koda tokom stvarnog izvršavanja testova, a ne analizira kod statički pre izvršavanja.

Evo kako JaCoCo funkcioniše u praćenju code coverage tokom izvršavanja testova:

1. **Instrumentacija koda**: Pre izvršavanja testova, JaCoCo instrumentira Java bajt kod dodajući dodatni kod koji prati izvršavanje svake linije koda, grananja, instrukcije i metode.

2. **Izvršavanje testova**: Nakon instrumentacije, testovi se izvršavaju. Tokom izvršavanja, dodatni kod ubačen od strane JaCoCo beleži koje linije koda su izvršene i koje nisu.

3. **Generisanje izveštaja**: Nakon završetka izvršavanja testova, JaCoCo koristi informacije prikupljene tokom izvršavanja da bi generisao izveštaj o pokrivenosti koda. Ovaj izveštaj sadrži detaljne informacije o tome koliko je svaka linija koda (ili grananja, instrukcije, metode) izvršena tokom testiranja.

Dakle, JaCoCo ne zna unapred šta će biti izvršeno; umesto toga, on beleži stvarno izvršavanje tokom testiranja. To mu omogućava da pruži precizne informacije o tome koji delovi koda su pokriveni testovima i koji nisu.

16. **Black box metode**

Black box metode su tehnike testiranja softvera koje se fokusiraju na testiranje funkcionalnosti softvera bez obzira na interne strukture ili detalje implementacije. Ove metode se nazivaju "black box" (crna kutija) zato što se softver posmatra kao zatvorena kutija, čije se funkcionalnosti ispituju samo na osnovu ulaznih podataka i očekivanih izlaznih rezultata, a ne na osnovu interne logike ili implementacije.

Evo nekoliko često korišćenih black box metoda:

1. **Funkcionalno testiranje**: Ova metoda se fokusira na testiranje da li softver radi prema specifikacijama funkcionalnosti koje su korisnicima obećane. Testovi se sprovode na osnovu ulaznih podataka i očekivanih izlaznih rezultata, bez obzira na to kako softver postiže te rezultate.

2. **Testiranje graničnih vrednosti**: Ova metoda se koristi za testiranje ponašanja softvera na ivicama i van granica dozvoljenih vrednosti. Fokusira se na testiranje krajnjih vrednosti i vrednosti koje su blizu granica da bi se otkrile greške koje se mogu pojaviti u ekstremnim uslovima.

3. **Testiranje ispravnosti**: Ova metoda se koristi za testiranje tačnosti izlaza softvera u odnosu na očekivane rezultate. Testovi se fokusiraju na proveru ispravnosti izlaza na osnovu različitih kombinacija ulaznih podataka.

4. **Testiranje korisničkog interfejsa**: Ova metoda se koristi za testiranje korisničkog iskustva i funkcionalnosti korisničkog interfejsa softvera. Testovi se fokusiraju na proveru interaktivnosti, navigacije i estetike korisničkog interfejsa.

5. **Testiranje kompatibilnosti**: Ova metoda se koristi za testiranje kompatibilnosti softvera sa različitim operativnim sistemima, pregledačima, uređajima ili drugim aplikacijama sa kojima softver može da interaguje.

Black box metode omogućavaju testiranje softvera na osnovu korisničkih zahteva i očekivanja, što ih čini korisnim alatima za osiguravanje kvaliteta softvera bez potrebe za detaljnim poznavanjem interne implementacije.

**17. Spring Boot test podrška**

Spring Boot pruža široku podršku za testiranje kako bi olakšao razvoj i održavanje Spring aplikacija. Ovde su neke od ključnih karakteristika i alata za testiranje u Spring Boot-u:

1. **Spring Test Framework**: Spring Boot se oslanja na Spring Test Framework koji omogućava jednostavno pisanje testova za Spring aplikacije. Ovaj framework pruža anotacije poput `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`, koje olakšavaju konfigurisanje konteksta aplikacije za testiranje.

2. **Embedded Test Environment:** Spring Boot omogućava pokretanje testova u ugrađenom okruženju (embedded environment), što znači da aplikacija može biti pokrenuta unutar testa, eliminirajući potrebu za spoljnim serverima ili baza podataka tokom testiranja.

3. **Spring Boot Test Slice Annotations**: Spring Boot pruža test slice anotacije poput `@WebMvcTest`, `@DataJpaTest`, `@RestClientTest`, koje omogućavaju selektivno učitavanje samo delova aplikacije relevantnih za testiranje određene komponente.

4. **Mocking Support:** Spring Boot olakšava mockovanje zavisnosti u testovima pomoću integrisanih alata poput Mockito i MockMVC. Ovo omogućava izolovano testiranje komponenti bez stvarnih spoljnih zavisnosti.

5. **Testiranje REST servisa**: Spring Boot pruža `TestRestTemplate` i `MockMvc` za testiranje REST kontrolera. Ovi alati omogućavaju slanje HTTP zahteva ka REST endpointima i proveru odgovora.

6. **Testiranje baza podataka**: Spring Boot podržava testiranje baza podataka pomoću anotacija poput `@DataJpaTest` i `@AutoConfigureTestDatabase`, koje omogućavaju brzo konfigurisanje baza podataka samo za potrebe testiranja.

7. **Testiranje konfiguracija**: Spring Boot omogućava testiranje konfiguracija aplikacije korišćenjem anotacija poput `@TestConfiguration` i `@MockBean`, što omogućava testiranje kako aplikacija reaguje na različite konfiguracije.

8. **Integracija sa popularnim alatima za testiranje**: Spring Boot se lako integriše sa popularnim alatima za testiranje kao što su JUnit, AssertJ, Hamcrest, Selenium, i drugi.

Ove karakteristike čine Spring Boot moćnim alatom za testiranje koji olakšava razvoj robustnih, pouzdanih i visoko kvalitetnih Spring aplikacija.

**18. Šta je Angular uveo u testiranje, a šta je od Jasmine?**

Angular je unapredio testiranje svojih aplikacija na nekoliko načina, uključujući uvođenje nekih novih konvencija i alata, dok se Jasmine, kao testni okvir, koristi u Angular okruženju za pisanje i izvršavanje testova. Evo kako oba doprinose testiranju u Angular:

**Angular:**

1. **Angular Test Bed**: Angular Test Bed je sredstvo koje omogućava konfiguraciju i izvršavanje testova za Angular komponente, direktive i servise. Pruža simuliranje Angular okruženja u kojem se komponente izvršavaju, što olakšava izolovano testiranje komponenti.

2. **Komponentno orijentisano testiranje**: Angular je komponentno orijentisan framework, što olakšava testiranje pojedinačnih komponenti. Testiranje se obično vrši na nivou komponenti koristeći Angular Test Bed, što omogućava brzo i efikasno testiranje pojedinačnih delova aplikacije.

3. **Integracija sa Angular CLI**: Angular CLI (Command Line Interface) pruža integraciju za kreiranje, konfigurisanje i izvršavanje testova. Može automatski generisati osnovne testove za nove komponente ili servise, olakšavajući početak sa pisanjem testova.

**Jasmine:**

1. **Testiranje na nivou jedinica**: Jasmine je testni okvir koji se često koristi u Angular okruženju za pisanje testova na nivou jedinica (unit tests). Omogućava definisanje specifikacija (spec) koje opisuju ponašanje pojedinačnih delova koda.

2. **Čista sintaksa**: Jasmine pruža čistu i izražajnu sintaksu za pisanje testova. Testovi se pišu koristeći funkcije i metode kao što su `describe`, `it`, `expect`, što olakšava čitanje i razumevanje testova.

3. **Asinhrono testiranje**: Jasmine podržava testiranje asinhronih operacija koristeći funkcije poput `beforeEach`, `afterEach` i `done`, kao i asinhronih verzija funkcija `it` i `expect`.

Kombinacija Angular Test Bed-a i Jasmine-a omogućava programerima da efikasno testiraju Angular aplikacije na nivou komponenti, servisa i drugih delova aplikacije. Ova kombinacija olakšava održavanje visokog nivoa kvaliteta i pouzdanosti Angular aplikacija.

**19. Kako bismo testirali običan JavaScript kod?**

Postoje različiti pristupi i alati koji se mogu koristiti za testiranje običnog JavaScript koda. Evo nekoliko načina:

1. **Unit testiranje pomoću testnih okvira**: Koristite testne okvire poput Jasmine, Mocha, Jest ili QUnit za pisanje i izvršavanje unit testova za funkcije i module vašeg JavaScript koda. Ovi okviri omogućavaju definisanje testova, izvršavanje testova i proveru očekivanih rezultata.

2. **Testiranje korisničkog interfejsa (UI testiranje)**: Koristite alate poput Selenium, Puppeteer ili TestCafe za testiranje korisničkog interfejsa vaše veb aplikacije. Ovi alati omogućavaju automatizaciju interakcija korisnika sa veb stranicom ili veb aplikacijom i proveru očekivanih rezultata.

3. **Testiranje API-ja (Backend testiranje)**: Koristite alate poput Postman, SuperTest ili Axios za testiranje API-ja i server-side funkcionalnosti vaše aplikacije. Ovi alati omogućavaju slanje HTTP zahteva ka API-ju i proveru odgovora.

4. **Mocking i stubiranje**: Koristite biblioteke poput Sinon.js za mockiranje i stubiranje zavisnosti kako biste izolovali delove koda za testiranje. Ovo je korisno kada želite testirati funkcije koje zavise od spoljnih resursa kao što su AJAX pozivi, vreme, ili baza podataka.

5. **Code coverage alati:** Koristite alate poput Istanbul ili JaCoCo za praćenje pokrivenosti koda testovima. Ovi alati vam omogućavaju da identifikujete delove koda koji nisu pokriveni testovima i da poboljšate pokrivenost testovima.

6. **Integracija sa CI/CD**: Integrirajte vaše testove u proces kontinuirane integracije i isporuke (CI/CD) kako biste automatski izvršavali testove pri svakoj promeni koda i osigurali da aplikacija ostane stabilna i pouzdana.

Ovi pristupi i alati mogu se kombinovati i prilagoditi prema potrebama i zahtevima vašeg projekta kako biste osigurali visok nivo kvaliteta i pouzdanosti vašeg JavaScript koda.