INTERNACIONALNI UNIVERZITET TRAVNIK U TRAVNIKU
FAKULTET INFORMACIONIH TEHNOLOGIJA TRAVNIK U TRAVNIKU

ZAVRŠNI RAD

Metode i metodologije u testiranju softvera

Mentor:
prof. dr. Mladen Radivojević

Student: Jasmina Osivčić
Br. indeksa: FIT-49/18

Travnik, 2019.

1. Sadržaj

1. Sadržaj ... 2
2. Uvod ... 4
2.1. Osnovni pojmovi ... 4
2.2. Primjeri softverskih grešaka ... 5
3. Proces razvoja softvera ... 6
3.1. Životni ciklus softvera ... 6
3.2. Životni ciklus testiranja softvera ... 7
3.3. Osnovni principi testiranja ... 8
4. Razvoj softvera vođen testiranjem *(eng. Test Driven Development – TDD)* ... 9
4.1. Faze softvera vođenog testiranjem ... 9
Prva faza *(eng. Red Phase)* ... 10
Druga faza *(eng.Green Phase)* ... 10
Treća faza *(eng. Refactor Phase)* ... 10
4.2. Prednosti razvoja softvera vođenog testiranjem ... 11
5. Proces testiranja ... 12
5.1. Verifikacija i validacija ... 12
5.2. Manuelno testiranje ... 12
5.3. Automatsko testiranje ... 13
6. Nivoi testiranja ... 14
6.1. Jedinično testiranje ... 14
6.2. Integracijsko testiranje ... 14
6.2.1. Big Bang integracija ... 16
6.2.2. Inkrementalna integracija ... 16
6.3. Sistemsko testiranje ... 18
7. Tehnike testiranja ... 19
7.1. Metoda crne kutije (*eng. Black box testing*) ... 19
7.2. Metoda bijele kutije (*eng. White box testing*) ... 20
8. Tipovi testiranja ... 21
8.1. Testiranje performansi sistema (*eng. performance testing*) ... 21
8.1.1. Testiranje opterećenja (*eng. Load testing*) ... 22

8.1.2. Stres testiranje (*eng. Stress testing*) 22
8.2. Test prihvatanja od strane korisnika (UAT) 23
8.2.1. Alfa testiranje 24
8.2.2. Beta testiranje 24
8.3. Regresijsko testiranje 25
8.3.1. Tehnike regresijskog testiranja 25
8.3.2. Automatizacija regresijskog testiranja 26
8.3.3. Prednosti i mane regresijskog testiranja 26
8.4. Statičko testiranje 27
8.4.1. Neformalni pregled 28
8.4.2. Formalni pregled 28
8.4.3. Walkthrough 29
8.4.4. Tehnički pregled 29
8.4.5. Inspekcija 29
8.5. Smoke testing 30
8.6. Sanity testiranje 31
9. Zaključak 32
10. Literatura 33

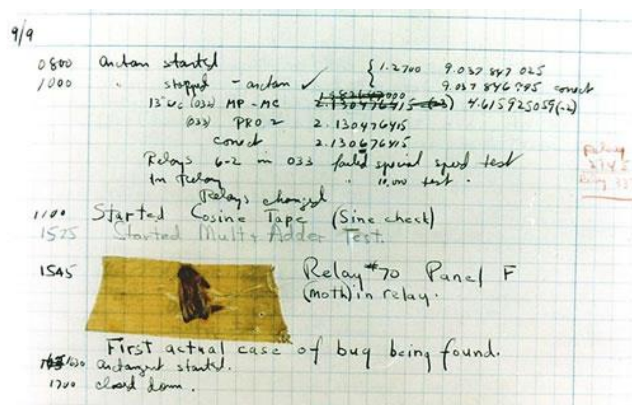
2. Uvod

2.1. Osnovni pojmovi

Da bismo uspješno razumjeli koncept testiranja softvera i izbjegli eventualne nedoumice prvo ćemo definisati osnovne pojmove.

Greška *(eng. error)* je rezultat ljudske aktivnosti, bilo da je nastala za vrijeme specifikacije zahtjeva ili tokom pisanja programa. Na primjer, ako neku od funkcionalnosti softvera pogrešno specifikujemo, to može dovesti do pogrešne implementacije ili eventualne greške u kodiranju koja će dovesti do neispravnog rada programa. Posljedica greške se naziva *defect ili bug* – ponašanje programa nije ispravno.

Pojam *bug* se pojavljuje 1945. godine kada je u relejima računara Harvard Mark II pronađen moljac. Računari su u to vrijeme zauzimali ogromne prostorije, a toplota unutrašnjih komponenti je privlačila moljce. Kada bi insekt ušao u komponente izazvao bi kratak spoj i kvar računara.



Slika 1. Prvi zabilježeni računarski *bug*[1]

Otkaz *(eng. failure)* nastaje ako sistem nije u mogućnosti da obavi funkciju koju korisnik od njega zahtijeva, jer se aktivira i izvršava defektni kod.

Testiranje je postupak izvršavanja softvera sa testovima. Osnovni ciljevi testiranja su da se pronađu otkazi ili da se provjeri ispravno ponašanje softvera. Otkaze koji se pronađu u softveru prilikom testiranjem je potrebno ispraviti. Testiranje softvera je dio procesa osiguravanja kvaliteta (*eng. Quality Assurance*), čiji je zadatak da se isporuči kvalitetan softver u određenom vremenskom roku.

Test je skup ulaznih vrijednosti, preduslova izvršavanja (*eng. preconditions*), očekivanih rezultata i stanja u kome sistem treba da bude i nakon izvršenja testa (*eng. postconditions*). Test služi da bi se ispitalo određeno ponašanje softvera.

[1] https://cdn0.tnwcdn.com/wp-content/blogs.dir/1/files/2013/09/bug.jpg

Test set se odnosi na skup svih testova koje je potrebno izvršiti na softveru. Testovi se pažljivo biraju i planiraju.

Softver tester se odnosi na profesionalca informacionih tehnologija koji je zadužen za izvršavanje jedne ili više aktivnosti testiranja. To podrazumijeva dizajn i pisanje testova, izvršavanje testova (manuelno ili automatski) i podnošenje izvještaja programerima i menadžerima.

2.2. Primjeri softverskih grešaka

Značaj testiranja softvera najlakše je prikazati kroz primjere gdje testiranje nije urađeno na adekvatan način ili greške nisu otkrivene na vrijeme. Ove greške mogu da dovedu do velikih finansijskih gubitaka.

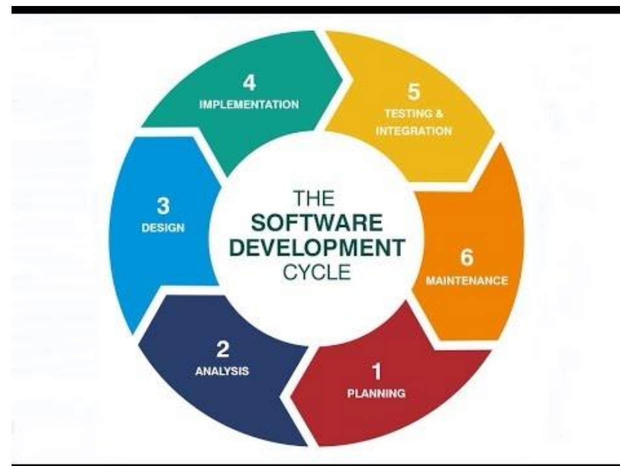
Primjeri softverskih grešaka koje su uzrokovale velike gubitke:

- 1990. godine - Svi korisnici američke telekomunikacione kompanije AT&T nisu mogli da uspostave pozive na velika rastojanja zbog softverske greške na relejnim svičevima. Greška je nastala kao posljedica ažuriranja na noviju verziju, bez adekvatnog testiranja da li nova verzija radi ispravno. Finansijski gubitak kompanije AT&T tog dana je bio preko 60 miliona dolara.
- 1999. godine - Pad Marsovog orbitera. Orbiter je prišao Marsu pod pogrešnim uglom i umjesto u orbiti završio je na površini planete. Uzrok je nekompatibilnost podataka koje su moduli slali jedan drugom, odnosno greška je nastala prilikom konverzije imperijalnih jedinica u metrički sistem, čime je uništen projekat vrijedan 327 miliona dolara.
- 1996. godine - Misija Ariana 5 let 501. Samo 40 sekundi nakon lansiranja, letjelica vrijedna 370 miliona dolara raspala se zbog softverske greške. Razvoj Ariane 5 trajao je 10 godina i uloženo je 8 milijardi dolara, a u trenutku polijetanja nosila je satelite vrijedne 500 miliona dolara.
- 2007. godine - Blokiranje aerodroma u Los Anđelesu. Zbog greške u softveru, pogrešne informacije su bile poslate u mrežu carine Sjedinjenih Američkih Država, što je dovelo do toga da 17.000 aviona bude osam sati zarobljeno na aerodromu.
- 2010. godine - Zbog greške u softveru kojim su se unosile informacije o organima koji mogu biti izvađeni iz donatora, pogrešni organi izvađeni su iz 25 donatora u Velikoj Britaniji. Softver za prikupljanje podataka se koristio od 1999. godine i pronađeno je još 400.000 grešaka.
- 2011. godine - Investicioni fond AXA je morao da plati preko 200 miliona dolara zbog štete koju je nanijela greška u softveru zbog koje su investitori izgubili investicije. Fond je znao za grešku, ali je tvrdio da je problem u tržištu i drugim faktorima.

3. Proces razvoja softvera

3.1. Životni ciklus softvera

Proces razvoja softvera se naziva životni ciklus softvera zato što opisuje „život“ softverskog proizvoda od početka njegov razvoja, pa do njegovog operativnog korišćenja i održavanja (*eng. Software Development Life Cycle*).



Slika 2. Životni ciklus softvera[2]

Razvoj softvera je podijeljen u nekoliko faza:

- Analiza i definisanje zahtjeva – u ovoj fazi se utvrđuju zahtjevi koje sistem treba da zadovolji, što se postiže saradnjom između razvojnog tima i kupaca odnosno korisnika sistema. Rezulat ove faze je lista korisničkih zahtjeva.
- Projektovanje sistema (dizajn) – u ovoj fazi se generiše projekat sistema koji daje plan rješenja odnosno arhitekturu sistema.
- Izrada softvera – faza u kojoj developeri pišu kod softvera.
- Testiranje softvera – faza u kojoj se otkrivaju i ispravljaju greške u sistemu.
- Isporuka sistema – sistem se isporučuje naručiocu, softver se instalira u radnom okruženju i vrši se obuka korisnika.
- Održavanje – dugotrajna faza u kojoj se ispravljaju greške u sistemu, otkrivene nakon njegove isporuke. Radi se na daljnjem unapređenju dijelova sistema prema zahtjevima korisnika ili promjenama u okruženju.

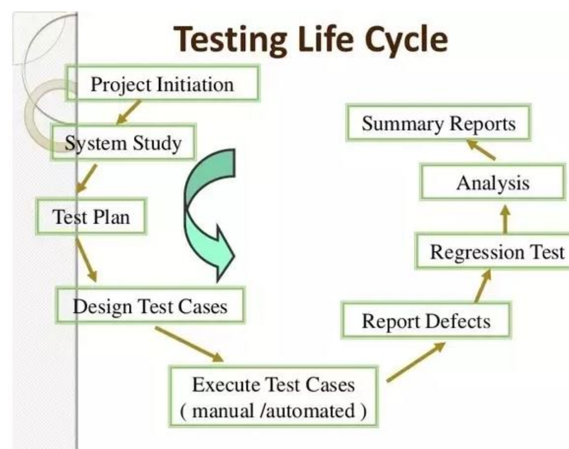
Uvođenje planiranja u proces razvoja softvera prema zadanim fazama dovelo je do nastanka tradicionalnih modela razvoja softvera. Modeli specificiraju različite faze u procesu razvoja, kao i redoslijed kojim se te faze izvršavaju.

[2] https://i.ytimg.com/vi/3Lxnn0O3xaM/hqdefault.jpg

Tradicionalne metode razvoja softvera su: kaskadni model, model vodopada (*eng. waterfall*), inkrementalni model, RAD model, spiralni i prototip model.
Tradicionalne metode razvoja softvera imale su mnoge nedostatke, što dalje vodi do novih metoda razvoja softvera. Nova metoda je agilni pristup koji negira potrebu za velikim planiranjem i obimnom dokumentacijom, a zalaže se za fleksibilniji razvoj gdje je glavni fokus na znanje i vještinu ljudi.

3.2. Životni ciklus testiranja softvera

Svaki od ovih modela razvoja softvera prati određeni životni ciklus softvera i opisuje faze u razvoju i njihov redoslijed. Tim za testiranje prati model životnog ciklusa testiranja softvera (*eng. Software Testing Life Cycle – STLC*) koji je sličan modelu razvoja softvera.



Slika 3. Životni ciklus testiranja softvera[3]

Testiranje se može podijeliti na tri faze životnog ciklusa softvera:

- Analiza zahtjeva – služi da bi se u ranim fazama razvoja softvera uklonile nejasnoće i spriječili problemi u razvoju.
- Izrada testnih scenarija – izrada testova prema korisničkim zahtjevima koje definiše produktni tim.
- Izvršavanje testova – izvršavanje napisanih testova i prijavljivanje grešaka.

S ekonomskog aspekta, sve greške pronađene prilikom testiranja predstavljaju trošak. Svaku od tih grešaka je potrebno ispraviti, što može produžiti vrijeme potrebno za izradu softvera. Što se prije greške u softveru otkriju, lakše ih je ispraviti, što uzrokuje manje troškove. Zato je vrlo bitno da testiranje počne kad i izrada specifikacija za softver.

[3] https://qph.fs.quoracdn.net/main-qimg-9cd90cdcdf317da64d9c43faa3087fee.webp

3.3. Osnovni principi testiranja

Postoji sedam osnovnih principa testiranja koji važe za bilo koji tip softvera. Ovi principi se posmatraju kao smjernice za svaki projekat.

Principi testiranja:

1. Testiranje pokazuje prisustvo defekata – testiranje ne može dokazati da sistem nema nijednog defekta već samo da pokaže da su defekti prisutni, odnosno da smanji broj neotkrivenih defekata u softveru.
2. Iscrpno testiranje nije moguće – testiranje svih mogućih kombinacija ulaza i preduslova nije moguće u praksi. Radi se analiza rizika i testovi se izvršavaju prema prioritetu.
3. Rano testiranje – testiranje treba da počne što je prije moguće da bi na vrijeme otkrivali defekte.
4. Grupisanje defekata *(eng. defect clustering)* – testiranje treba fokusirati proporcionalno broju očekivanih i pronađenih defekata po modulu. Moduli koji sadrže najviše defekata su najkritičniji i sadrže implementaciju bitnih funkcionalnosti sistema.
5. Paradoks pesticida – ukoliko iste testove stalno ponavljamo u svakoj iteraciji testiranja, ti testovi više neće moći da otkriju nove defekte. Zbog toga testove treba redovno ažurirati i dodavati nove.
6. Testiranje zavisi od konteksta – testiranje se prilagođava funkcionalnostima sistema.
7. Odsustvo grešaka ne garantuje da sistem radi kako treba – pronalaženje i ispravljanje defekata ne pomaže ako je sistem neupotrebljiv ili ako ne ispunjava zahtjeve korisnika.

4. Razvoj softvera vođen testiranjem *(eng. Test Driven Development – TDD)*

Razvoj softvera vođen testiranjem spada u agilne metode razvoja softvera. Nastao je 1999. godine kao dio *Extreme programming* metodologije.
Ova metoda je popularna u današnje vrijeme kod razvoja manjih softverskih projekata, iako sve više složenijih softverskih projekata počinje da je primjenjuje. Njene prednosti su što daje kraći i pregledniji kod i što smanjuje obim dokumentacije potrebne za projekat.

Razvoj softvera vođen testiranjem je metoda za razvoj softvera i cjelokupnih programa. Ova metoda se svodi na to da se prvo napišu testovi, te se nakon toga kreće sa implementacijom metoda koje će proći testove. Piše se jedan po jedan test. Prvo se implementira metoda koja prolazi prvi test, zatim se kreće sa pisanjem i rješavanjem drugog testa.
U dogovoru sa korisnicima i njihovim zahtjevima pišu se testovi, a implementacijom rješenja za te testove postižu se željene funkcionalnosti softvera.

Kada se napiše test, prva indikacija da test neće proći je to što će pokušati instancirati funkciju ili klasu koja još nije implementirana. Tako da, prvi korak prema rješenju takvog testa je da se napravi funkcija ili metoda koju test treba da pozove. Nakon toga se kreće sa pisanjem jednostavnih funkcionalnosti, ali s ciljem da test prođe.
Kod koji je implementiran za ovu funkcionalnost mora biti što jednostavniji i s ciljem samo da prođe test, bez trošenja vremena na sljedeću metodu koja će biti implementirana. Poenta ovog koraka je da kod bude što jednostavniji za pregled i održavanje.

Nakon što svi testovi prođu pišu se negativni testovi. Negativni testovi su testovi za koje znamo da neće proći implementirane metode. Kada prođu svi testovi vrši se refaktorizacija koda. Refaktorizacija koda vrši se tako što se ukloni duplikacija koda, prepravi se da odgovora dogovorenim pravilima za pisanje programskog koda i sl.

4.1. Faze softvera vođenog testiranjem

Metoda razvoja softvera vođenog testiranjem se dijeli na tri faze. Te faze su:

- Prva faza *(eng. Red Phase)*
- Druga faza *(eng. Green Phase)*
- Treća faza *(eng. Refactor Phase)*

Ove faze u razvoju softvera osiguravaju da imamo testove za koje ćemo pisati kod, kao i da pišemo kod koji nam je potreban da bi prošli testovi.

Prva faza *(eng. Red Phase)*

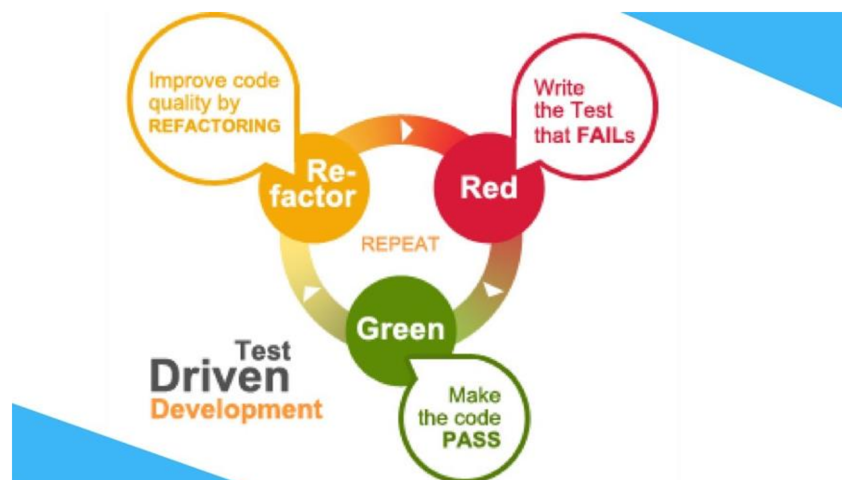
To je faza koja developerima koji tek počinju sa primjenom ove metode stvara najviše problema. Ona podrazumijeva pisanje testova, za klase i metode koje još nisu implementirane. Posljedica je da napisani testovi u ovoj fazi neće proći.

Druga faza *(eng.Green Phase)*

Sve dok testovi ne prolaze, traje prva faza. Da bi prešli u drugu fazu potrebno je implementirati kod potreban za testove da bi prošli. U ovoj situaciji je ponekad teško odlučiti kako napisati najmanje koda, pa je dobra praksa da se napiše najjednostavniji kod da bi test prošao.

Treća faza *(eng. Refactor Phase)*

U ovoj fazi se vodi računa o kvaliteti koda, jednostavnosti održavanja i jednostavnosti čitanja koda. Refaktorizacija podrazumijeva mijenjanje koda uz uvažavanje navedena tri atributa. Testovi u ovoj fazi služe da bi developeri vidjeli da promjene na kodu ne mijenjaju dosadašnju prolaznost testova. Dok testovi prolaze znamo da je kod u redu.



Slika 4. Faze razvoja softvera vođenog testiranjem[4]

Nakon što se završi faza refaktorizacije počinje se ponovo s prvom fazom i novim testom.

[4] https://i.ytimg.com/vi/T38L7A0xP-c/maxresdefault.jpg

4.2. Prednosti razvoja softvera vođenog testiranjem

Prednosti ovog modela razvoja softvera su:

- Osigurava se kvalitetan kod od samog početka. Piše se samo kod na osnovu kojeg će proći testovi. Ovim pristupom se postiže da su zahtjevi korisnika ispunjeni.
- Softver odgovara na sve zahtjeve klijenta. Zahtjevi su pisani kao testovi i vrši se implementacija funkcionalnosti prema napisanim testovima.
- Prave se jednostavniji programi i korisnički interfejsi *(eng. Application Interface - API).* Developer koji piše program i pravi korisnički interfejs je prva osoba koja ih koristi, te na osnovu toga može da vidi da li imaju smisla ili je potrebno poboljšanje.
- U finalnim verzijama skoro da nema nekorišćenog koda.
- Testovi osiguravaju da sve potencijalne promjene u kodu ili dodavanje novih metoda neće narušiti postojeće funkcionalnosti.
- Ova metoda ima vrlo malo dekefata. Ako se uoči neki problem ili *bug*, kod se ispravlja i piše se poseban test za taj *bug* da bi se osiguralo da se taj problem neće više pojavljivati.

5. Proces testiranja

5.1. Verifikacija i validacija

Testiranje nekog softvera se sastoji od verifikacije i validacije.
Ciljevi procesa verifikacije i validacije su:

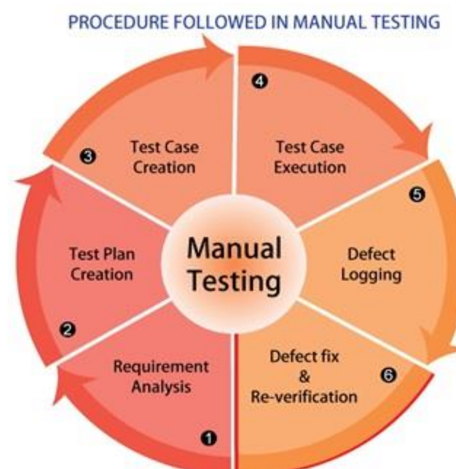
- Pronaći greške u sistemu
- Procjena da li je sistem upotrebljiv, odnosno da li izvršava zahtjevane funkcionalnosti

Postupkom verifikacije provjeravamo da li je sistem pravilno implementiran, odnosno provjerava se da li je sistem dobro napravljen.
Postupkom validacije provjeravamo da li taj sistem ispunjava zahtjeve koji se nalaze u specifikaciji sistema, odnosno da li sistem odgovara potrebama i željama korisnika.

Verifikacija i validacija se provode kroz cijeli životni ciklus softvera, neovisno o njegovoj kompleksnosti, veličini i sl.
Proces verifikacije i validacije se sastoji od tehnika kontrole softvera i testiranja softvera. Kontrola i testiranje softvera su komplementarne tehnike.

5.2. Manuelno testiranje

Manuelno testiranje podrazumijeva ručno izvođenje testova. Osoba koja izvodi testove, tester, prati određene korake kako bi verifikovao neki dio sistema ili dio koda koji se testira. Test je uspješan ako se izvrši nad softverom i dobiju se rezultati koji su isti kao očekivani. Popularna metoda je metoda destruktivnog testiranja gdje se test smatra uspješnim ako se njegovi rezultati nad softverom ne slažu sa očekivanim, odnosno test je uspješan ako otkrije postojanje grešaka u softveru.



Slika 5. Proces manuelnog testiranja[5]

[5] http://www.professionalqa.com/assets/images/manual-testing.jpg

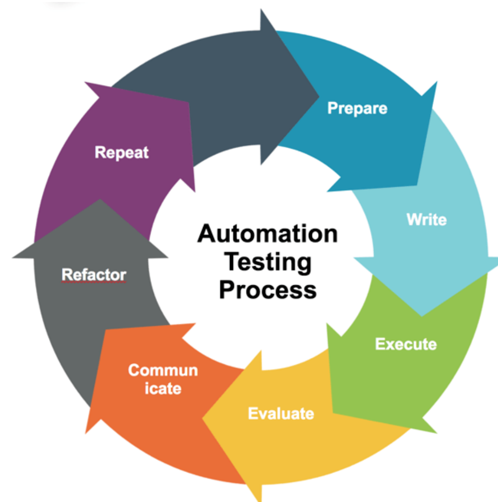
Nedostaci manuelnog testiranja:

- Manje pouzdano – jer ne postoji strogo mjerilo da li su aktuelni i očekivani rezultati u korektnom odnosu već se oslanja na mišljenje testera.
- Visok rizik – manuelno testiranje ima visok rizik propusta i grešaka.
- Nepotpuna pokrivenost – testiranje je vrlo zahtjevan proces ako uključuje testiranje na više platformi, operativnih sistema, servera, klijenata, protokola ili poslovnih procesa.
- Rokovi – Ograničeni resursi u toku manuelnog testiranja sprečavaju efikasno testiranje što dovodi do probijanja rokova. Skup i dugutrajan proces.

5.3. Automatsko testiranje

Za razliku do manuelnog testiranja, automatsko testiranje podrazumijeva postojanje koda koji je pisan tako da bi se automatizovali koraci za izvršenje određenog testa. Automatizacija testova se može pisati u bilo kojem programskom jeziku, a napisani kod se naziva testna skripta.

Prednost automatskog testiranja je što se testovi izvršavaju dosta brže nego manuelno, što je vrlo bitno za regresijsko testiranje.



Slika 6. Proces automatskog testiranja[6]

Prednosti automatskog testiranja:

- Brzina – brži su od manuelnih testova zbog čega se postiže ušteda vremena.
- Pouzdanost – eliminiše se opasnost od ljudske greške.
- Ne zahtijeva prisustvo testera.

Nedostaci automatskog testiranja:

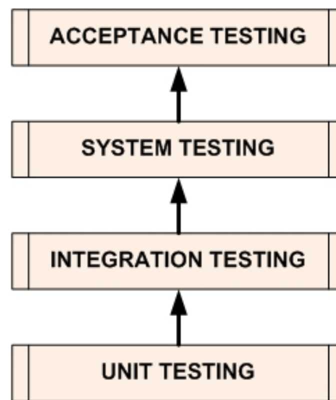
- Automatski razvoj testova je skuplji proces od kompletnog ciklusa manuelnog testiranja.
- Zahtijeva dodatne stručne testere.

[6] https://www.joecolantonio.com/wp-content/uploads/2018/11/AutomationTestingProcess.png

6. Nivoi testiranja

Testiranje softvera ima više nivoa:

- Jedinično *(eng. Unit testing)*
- Integracijskog
- Sistemskog testiranja



Slika 7. Nivoi testiranje softvera[7]

6.1. Jedinično testiranje

Jedinično testiranje je testiranje jedinice programskog izvornog koda. Pod jedinicom se smatra najmanji dio softvera koji se može testirati.
U objektno orijentisanom programiranju, kao jedinica se posmatra jedna metoda ili klasa. Jedinično testiranje obično izvode developeri za dio koda koji su napisali kako bi provjerili da li napisane funkcije rade kako treba. Jedna funkcija može da ima više testova.
Jedinično testiranje ne može provjeriti funkcionalnost softvera, ali može osigurati da komponente sistema mogu raditi svaka za sebe.

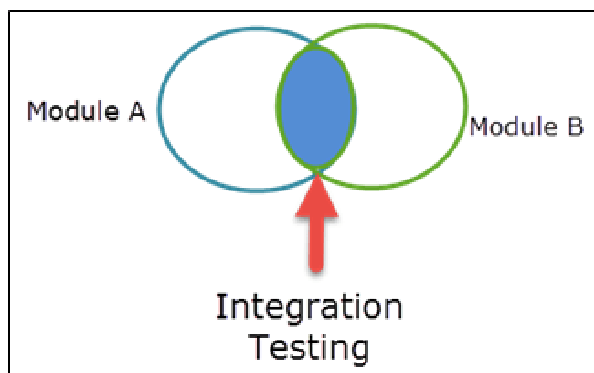
6.2. Integracijsko testiranje

Faza integracijskog testiranja dolazi na kraju jediničnog testiranja. Integracijsko testiranje je u praksi često zanemareno i ne obavi se na adekvatan način što dovodi do ozbiljnih problema.

Integracijsko testiranje je faza u kojoj se pojedinačni modeli i jedinice spajaju i testiraju zajedno kao cjelina. Komponente koje ulaze u integracijsko testiranje su već prošle jedinično testiranje. Ove komponente se grupišu i nad njima se izvršavaju posebno pripremljeni testovi. Ovo testiranje je važno iz razloga da se izbjegnu svi potencijalni problemi koji mogu nastati prilikom spajanja komponenti u cjelinu.

[7] http://softwaretestingfundamentals.com/wp-content/uploads/2011/01/software_testing_levels1.jpg

Cilj integracijskog testiranja je detaljno testiranje integracija i interfejsa između komponenti, a na najvišem nivou uključuje i interakciju sa drugim komponentama sistema (operativni sistem, sistem fajlova, hardverski i softverski interfejsi).



Slika 8. Integracijsko testiranje[8]

Tehnike integracijskog testiranja:

- Big Bang integracija
- Integracija od vrha prema dnu (*eng. Top-down integration*)
- Integracija od dna prema vrhu (*eng. Bottom-up integration*)
- Sendvič integracija
- Integracija po grafu poziva

Neke komponente u softveru mogu biti nedovršene ili nedostupne za testiranje. Tako da se stvarne komponente u test okruženju zamjenjuju sa simuliranim komponentama koje odgovaraju na pozive na isti način kao i realne komponente.
Ove komponente, prema tome da li pozivaju ili su pozivane dijelimo na drajvere ili stabove.

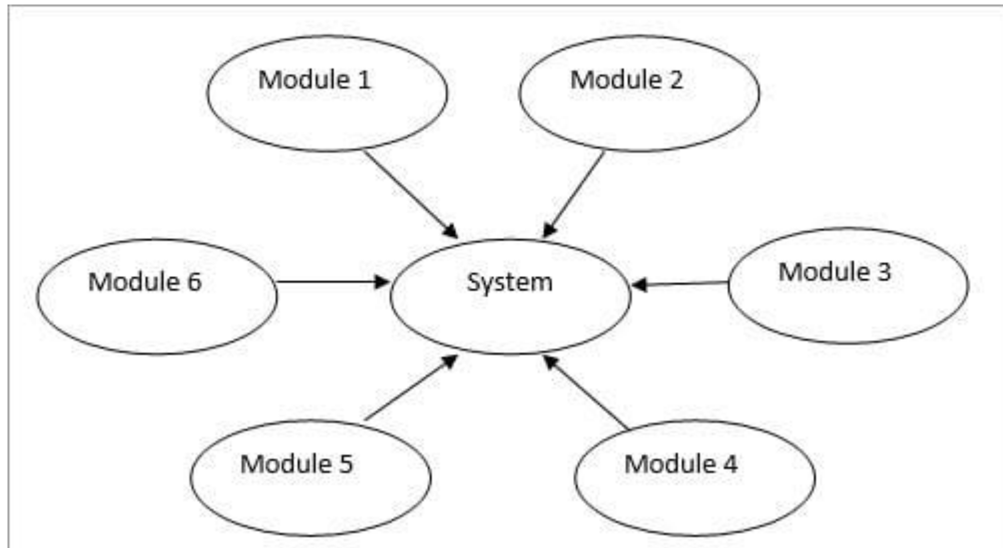
Drajveri (*eng. drivers*) su komponente koje simuliraju rad modula višeg nivoa, koje pozivaju druge komponente softvera i očekuju neki odgovor. Drajveri se koriste u integraciji od dna prema vrhu, jer ova tehnika prvo implementira i integriše module na dnu hijerarhije, dok moduli višeg nivoa još nisu implementirani.

Stub (*eng. stub*) je model koji simulira ponašanje modula nižeg nivoa, simulira rad realnih komponenti koje primaju pozive i vraćaju iste tipove rezultata kao i realne komponente. Stubovi se koriste u integraciji od vrha prema dnu jer se prvo implementiraju i integrišu modeli viših nivoa hijerarhije. Da bi se mogli testirati viši nivoi, moduli nižih nivoa koji još nisu implementirani se zamjenjuju stubovima.

[8] https://www.guru99.com/images/3-2016/032816_1230_SystemInteg1.png

6.2.1. Big Bang integracija

Kod Big Bang integracije, pojedinačne komponente sistema se razvijaju odvojeno i testiraju na jediničnom nivou. Kad se komponente spoje i integrišu, nakon završetka razvoja i jediničnog testiranja, sistem se testira kao cjelina.



Slika 9. Big Bang integracijski pristup[9]

Nakon spajanja više modula odjednom, pojavit će se dosta defekata i problema. Problemi koji postoje na interfejsima između modula se uoče vrlo kasno, na samom kraju razvoja sistema.
Pronađene probleme je vrlo teško izolovati zbog velikog broja komponenti koje su istovremeno integrisane, a neke kritične defekte je tek moguće otkriti u produkciji.

Big Bang integracija nije najbolji primjer načina na koji treba pristupiti testiranju sistema. Integracija komponenti se vrši s nadom da će sve komponente raditi savršeno nakon što se spoje u cjelinu.
Tako da, možemo reći da je ovaj način integracijskog testiranja primjenjiv na jako male sisteme koji se sastoje od nekoliko komponenti. U slučaju ozbiljnih i kompleksnih sistema se ne može upotrebljavati.

6.2.2. Inkrementalna integracija

Kod ovog načina testiranja svaka komponenta se testira jedinično, i nakon toga se komponente integrišu inkrementalno i testiraju detaljno da se osigura ispravnost interakcije i interfejsa između komponenti.
Inkrementalna integracija znači da se komponente dodaju inkrementalno, jedna po jedna. Svako dodavanje komponenti zahtijeva detaljno testiranje prije dodavanja sljedeće komponente.

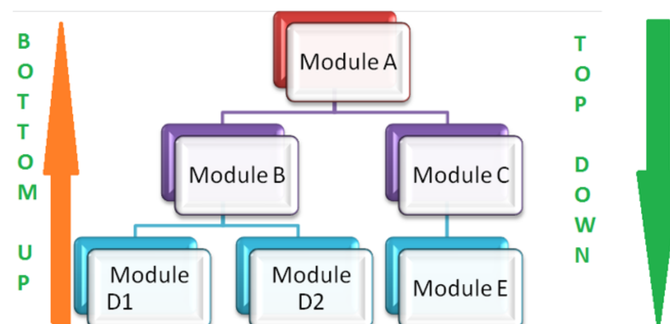
[9] https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/05/Big-bang-approach.jpg

Integrisane komponente se testiraju kao grupa, s ciljem da se osigura da su ispravno integrisane i da informacije ispravno idu od jedne komponente do druge.

Mana ove integracije je to što je često potrebno upotrebljavati stubove i drajvere, da bi se zamijenile komponente koje još nisu implementirane ili integrisane.

Tipovi inkrementalne integracije su:

- Od vrha prema dnu (*eng. top-down integration*) - testira se integracija na najvišem nivou prvo, zatim se spušta sve do najnižeg nivoa pri čemu se koriste stubovi, kojima se zamjenjuje nepostojeći ili netestirani dijelovi s nižih nivoa.
- Od dna prema gore (*eng. bottom-up integration*) - testira se integracija na najnižim nivoima i diže se do najvišeg nivoa. Ovdje su svi dijelovi sistema testirani i implementirani za svaki naredni nivo integracije.



Slika 10. Tipovi inkrementalne integracije[10]

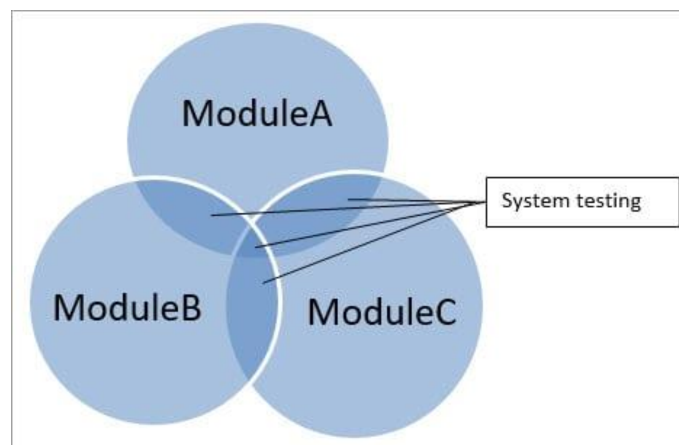
[10] https://bitwaretechnologies.com/wp-content/uploads/2017/02/FRRE.png

6.3. Sistemsko testiranje

Sistemsko testiranje je testiranje sistema kao cjeline, nakon što se uspješno završi kompletna integracija.
Svrha sistemskog testiranja je verifikacija da softver kao cjelina ispunjava zahtjeve koji su definisani u specifikaciji zahtjeva sistema. Sistemsko testiranje obuhvata testove koji se definišu na osnovu specifikacije zahtjeva sistema i obično ga obavljaju najiskusniji testeri. Ponekad se za ovaj zadatak angažuje nezavisan tim testera, da bi se obezbjedila potpuna objektivnost pri sistemskom testiranju.

Fokus sistemskog testiranja je kompletno integrisan softver, da bi se provjerilo da li sve komponente rade zajedno kao cjelina (*eng. end to end test*). Testiranje s kraja na kraj znači da se softver testira u realnom scenariju upotrebe.
Testiranje se obavlje iz korisnične perspektive, sa dodatnom provjerom svakog ulaznog podatka i odgovarajućeg izlaza.
Sistemsko testiranje ne mora uvijek biti funkcionalno testiranje, ponekad je potrebno provjeriti i nefunkcionalne zahtjeve kao što su performanse sistema, skalabilnost, pouzdanost i test opterećenosti.

Pod sistemsko testiranje se može svrstati i testiranje prihvatanja od strane klijenta (*eng. user acceptance testing* ili skraćeno *acceptance testing*).
Ovu vrstu testiranja vrši klijent koji je naručio softverski sistem, s ciljem da bi verifikovao da li sistem sadrži sve što je specifikovano u zahtjevima i da li je implementirano na odgovarajući način.



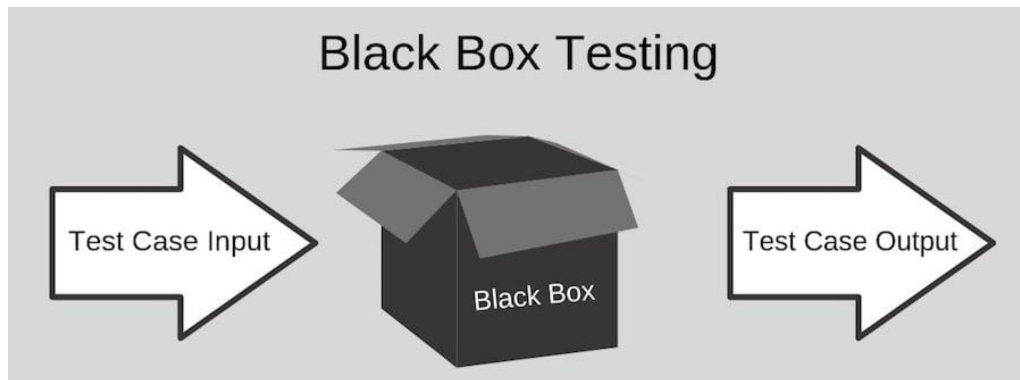
Slika 11. Sistemsko testiranje[11]

[11] https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/05/system-testing-example.jpg

7. Tehnike testiranja

7.1. Metoda crne kutije (*eng. Black box testing*)

Metoda crne kutije ili funkcionalno testiranje je oblik testiranja softvera gdje unutrašnja struktura, dizajn i implementacija softvera ili softverskih jedinica koje se testiraju nisu poznate testeru. Kako implementacija i struktura softvera nije poznata, softver se posmatra kao crna kutije po čemu je ovaj metod i dobio ime.



Slika 12. Metoda crne kutije[12]

Jedina poznata informacija koju tester ima za određivanje i dizajn testova je specifikacija zahtjeva programa.
Metodom crne kutije se mogu otkriti nepostojeće funkcionalnosti ili pogrešne implementacije u softveru, greške u ponašanju softvera kao i problemi sa performansama sistema.

Funkcionalno testiranje se može primijeniti na svim nivoima testiranja (jediničnom, integracijskom i sistemskom nivou). Uticaj nivoa testiranja na metod crne kutije se ogleda samo u kompleksnosti izvršavanja ove metode.

Glavna prednost metode crne kutije je u tome što testovi koji se koriste su nezavisni od implementacije softvera, jer se temelje isključivo na specifikaciji sistema. Testovi su upotrebljivi i u slučaju kada dođe do promjena u implementaciji softvera.
Pošto se ovi testovi pišu na osnovu specifikacije sistema, mogu se razvijati i paralelno s razvojem softvera, odnosno može se početi s njihovim dizajniranjem čim se završi specifikacija sistema.
Još jedna prednost ove metode je što se testovi izvršavaju sa korisničke tačke gledišta.

Mana metode crne kutije je to da neke implementirane dijelove sistema nije moguće pokriti testovima, odnosno implementirane su i funkcionalnosti koje odstupaju od specifikacije sistema.

---

[12] https://phoenixnap.com/blog/wp-content/uploads/2018/11/black-approach.jpg

Najveći problem ove metode je nejasna i nekompletna specifikacija softvera, u slučaju kad krajnji korisnik ne formuliše dobro svoje potrebe, čime se značajno otežava dizajn testova.

Tehnike testiranja crne kutije su:

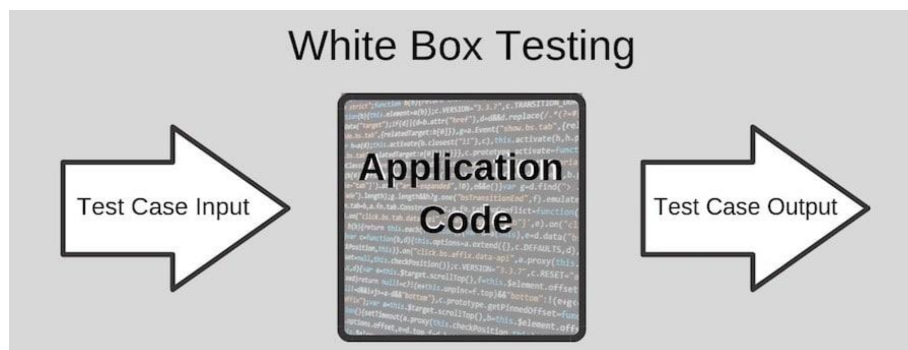
- Klasa ekvivalencije
- Analiza graničnih vrijednosti
- Tabela odlučivanja i uzročno-posljedični graf
- Testiranje zasnovano na modelu stanja

7.2. Metoda bijele kutije (*eng. White box testing*)

Strukturno testiranje, poznato kao metoda bijele ili staklene kutije, je tehnika testiranja gdje je testerima poznata implementacija softvera.
Za razliku od modela crne kutije koji je fukusiran na to šta softver radi, model bijele kutije je fokusiran na to kako softver radi.

Cilj testiranja ovim modelom je da se izvrše sve programske strukture i sve strukture podataka u softveru. Ovdje se provjerava samo kod, a ne specifikacija. Ovaj metod se može primijeniti na svim nivoima testiranja, ali se najčešće primjenjuje nakon metode crne kutije.

Metod bijele kutije koriste i sami developeri, jer se očekuje da su implementirane komponente temeljno testirane prije nego se integrišu u softver, odnosno daju testerima na detaljno testiranje.



Slika 13. Metoda bijele kutije[13]

Pokrivenost testiranja (*eng. coverage*) je mjera do koje je neki softver ili određena softverska komponenta istestirana nekim skupom testova, u smislu procenta obuhvaćenih stavki.
U slučaju da pokrivenost nije 100%, potrebno je proširiti skup testova s novim testovima. Pokrivenost se povećava tako da se sistemski pišu novi testovi da bi se pokrili svi dijelovi softvera, odnosno da se svi dijelovi softvera izvrše bar jednom.

[13] https://phoenixnap.com/blog/wp-content/uploads/2018/11/white-testing-diagram.jpg

8. Tipovi testiranja

8.1. Testiranje performansi sistema (*eng. performance testing*)

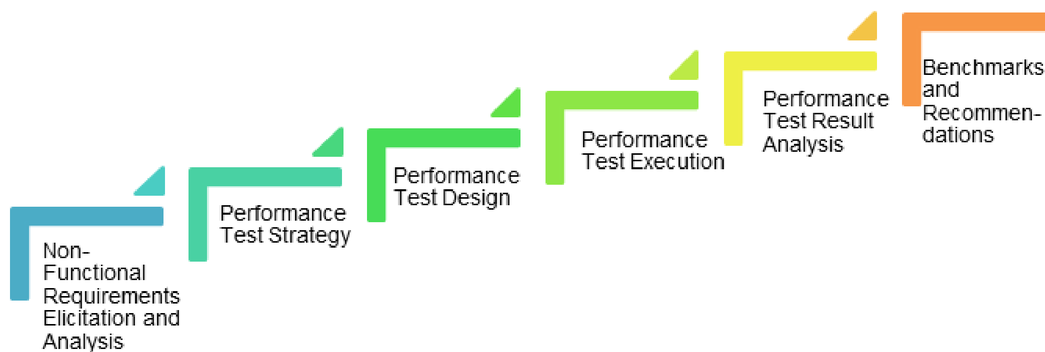
Performanse obuhvataju vrijeme odaziva softvera, pouzdanost, upotreba resursa i skalabilnost. Testiranje performansi je tip testiranja koji verifikuje da se sistemski softver ponaša na odgovarajući način pod nekim očekivanim opterećenjem.

Cilj testiranja performansi nije pronalazak novih defekata, već da se eliminišu petencijalni problemi koji utiču na performanse sistema.

Kod testiranje performansi posmatraju se:

- Brzina – provjerava se brzina odaziva softvera
- Skalabilnost – testira se maksimalno korisničko opterećenje pod kojim softver može da radi
- Stabilnost – provjerava se stabilnost softvera pod različitim opterećenjima

Testiranje performansi može da otkrije greške koje nije pokazalo funkcionalno testiranje, a to omogućava da se softver popravi i poboljša prije nego se pusti u upotrebu.
Bez testiranja softvera prije puštanja u upotrebu, mogu da se pojave razni problemi. Ti problemi mogu da budu spor rad softvera u slučaju da ima više korisnika, nekonzistentnosti prilikom pristupa i sveukupne loše upotrebljivosti softvera.



Slika 14. Životni ciklus performance testiranja[14]

Testiranje performansi se može grupisati na više tipova:

- Testiranje opterećenja (*eng. load test*)
- Stres test (*eng. stress test*)
- Test izdržljivosti (*eng. endurance test*)
- Testiranje skalabilnosti (*eng. scalability testing*)

[14] https://qainsights.com/wp-content/uploads/2012/11/Performance-Testing-Life-Cycle-QAInsights.png

Najčešći problemi koji se otkrivaju testiranjem performansi:

- Predugo vrijeme pokretanja sistema – potrebno je da vrijeme pokretanja sistema bude što je moguće manje.
- Loše vrijeme odziva – posmatra se vrijeme koje je potrebno od trenutka kada korisnik izvrši unos u sistem, da sistem vrati odgovor na taj korisnički zahtjev. Ovo vrijeme bi trebalo da bude jako kratko, jer može se desiti da korisnik izgubi interesovanje ako dugo čeka na odgovor.
- Loša skalabilnost – ogleda se u slučajevima kada aplikacija ne može da podrži očekivani broj korisnika. Test opterećenosti se koristi da bi se provjerilo da sistem može da podrži predviđeni broj korisnika.
- Uska grla – ograničenja sistema koja utiču na performanse sistema. Greške mogu da budu u kodu, ali mogu da budu i hardverske (nedovoljna procesorska moć, nedostatak memorije, loša mrežna infrastruktura i sl.)

8.1.1. Testiranje opterećenja (*eng. Load testing*)

Testiranjem opterećenja određuju se performanse sistema u okviru realnih uslova upotrebe. Ovo testiranje spada u testiranje nefunkcionalnih zahtjeva.
Određuje se ponašanje sistema pod normalnih opterećenjima, kao i pod najvećim očekivanim opterećenjem. Identifikuje se maksimalni operativni kapacitet, uska grla ako postoje i koja komponenta izaziva opadanje performansi.
U slučaju kada se opterećenje podigne iznad razumnog nivoa, test opterećenja postaje stres test. Ovaj oblik testiranja najčešće se primjenjuje za klijent/server web sisteme.

Test opterećenja određuje:

- Maksimalni operativni kapacitet sistema
- Da li postojeća infrastruktura zadovoljava potrebe sistema
- Održivost sistema u slučaju povećanog korisničkog opterećenja (*eng. peak load*)
- Broj konkurentnih korisnika koje sistem može da izdrži

8.1.2. Stres testiranje (*eng. Stress testing*)

Stres test je vrsta testiranja u kom se sistem stavlja pod ekstremna opterećenja i posmatra se njegovo ponašanje za vrijeme ogromnog broja zahtjeva ili obrade podataka.

Cilj stres testa je da sistem izbaci van normalnog maksimalnog operativnog kapaciteta i dovede do tačke pucanja. Prilikom određivanja tačke pucanja, određuju se i sigurnosna ograničenja sistema, koja se onda upoređuju sa zahtjevima iz specifikacija. Na taj način se određuje stabilnost i pouzdanost sistema, kao i da li sistem ispunjava zahtjeve iz specifikacije sistema.

Ovdje je vrlo važno da se utvrdi kada sistem tačno puca, odnosno da se utvrdi oblik otkaza sistema. Sistem ne bi trebalo da u potpunosti otkaže ni u slučaju ekstremnog opterećenja.

Svaki stres test se treba fokusirati na sljedeće:

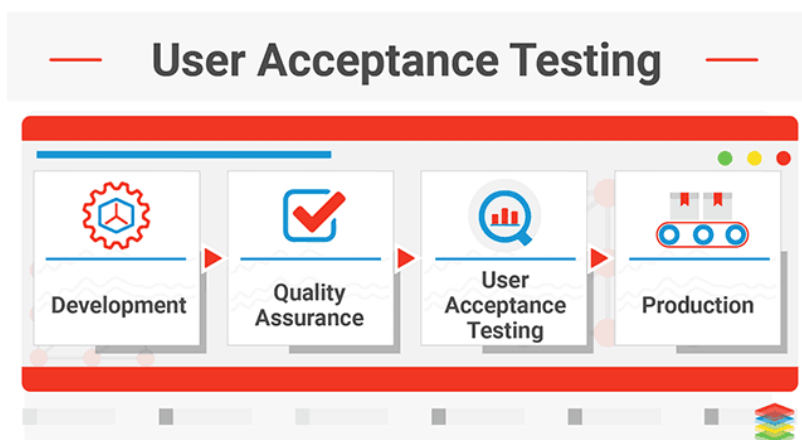
- Provjera da li sistem radi pod ektremnim opterećenjima
- Provjera da li sistem prikazuje odgovarajuće poruke u slučaju kada je opterećenje toliko da ne može da odgovori na zahtjev korisnika
- Ako sistem potpuno otkaže, dolazi do gubitka podataka, novca i povjerenja korisnika

U stres testiranju, analizira se i ponašanje sistema nakon otkaza. Za uspješno izvršen stres test, potrebno je da sistem ispiše poruku o grešci i da bude bez gubitka kritičnih podataka. Nakon otkaza sistema, potrebno je i verifikovati uspješan oporavak sistema.

8.2. Test prihvatanja od strane korisnika (UAT)

Test prihvatanja od strane korisnika (*eng. User acceptance testing* ili *Acceptance testing*) je testiranje koje vrši klijent kada je sistem spreman za isporuku, nakon što se ispravila većina defekata pronađena u sistemskoj fazi testiranja.

Cilj ovog testiranja je da klijent stekne povjerenje u sistem koji je implementiran. Provjeravaju se funkcionalnosti prema specifikaciji zahtjeva i određuje se da li sistem ispunjava potrebe krajnjih korisnika.



Slika 15. UAT[15]

Ovo testiranje vrše krajnji korisnici sistema da bi osigurali da sistem ima implementirane sve funkcionalnosti koje su dogovorene. Testiranje se vrši u finalnoj fazi projekta, prije puštanja sistema u produkcijsko okruženje.

[15] https://images.xenonstack.com/insights/xenonstack-what-is-user-acceptance-testing.png

Test prihvatanja od strane korisnika je obavezan zbog:

- U slučaju da su developeri pogrešno protumačili zahtjeve i samim tim pogrešno implementirali funkcionalnosti
- Ponekad promjene koje se naknadno definišu , ne budu iskomunicirane s developerima odnosno ne budu implementirane.

U praksi test prihvatanja od strane korisnika ima dvije faze, a to su alfa i beta testiranje.

8.2.1. Alfa testiranje

Cilj alfa testiranje je da se identifikuju svi potencijalni problemi prije nego se softver pošalje klijentima, odnosno prije nego se počne sa beta testiranjem. Alfa testiranje se vrši u kontrolisanim uslovima, developeri i testeri sarađuju da bi simulirali ponašanje krajnjih korisnika i prate sve uočene problema.
Testeri u fazi alfa testiranja simuliraju krajnje korisnike upotrebom raznih tehnika, s ciljem da izvrše i provjere sve akcije koje bi i korisnik mogao napraviti.

Alfa testiranje bi trebalo da izvode timovi nezavisnih testera, koji nisu učestvovali u ranijim fazama testiranja, radi objektivnosti pri testiranju.

8.2.2. Beta testiranje

Beta testiranje (*eng. Field test*) ili testiranje na terenu se izvodi na lokaciji klijenta. Ovo testiranje vrše stvarni korisnici sistema u stvarnim uslovima.
Beta verzija softvera se isporučuje korisnicima, koji ga instaliraju i koriste u realnim uslovima.

Cilj beta testiranja je da stvarni korisnici u realnom okruženju otkriju greške i propuste iz korisničke perspektive. Beta testiranje smanjuje rizik od neupjeha softvera i generalno poboljšava kvalitet softvera.
Ova faza testiranja je finalna prije nego što se softver isporuči svim korisnicima.

Prednost beta testiranja je što od korisnika dobijamo povratne informacije.

8.3. Regresijsko testiranje

Softver koji se razvija po iterativnom modelu, podrazumijeva da se u toku svake iteracije implementira dio funkcionalnosti sistema, koje se kasnije integrišu i testiraju tako da je fokus usmjeren na nove dijelove koji su dodani.
Dešava se da, kada integrišemo nove komponente u sistem, komponente koje su do sada radile u prethodnim iteracijama prestanu da rade.

Regresijsko testiranje je tip testiranja softvera čiji je cilj da potvrdi da nove promjene u kodu softvera nisu uzrokovale greške u već postojećim funkcionalnostima sistema.
Ovaj tip testiranja podrazumijeva ponavljanje testova koji su već izvršeni u prethodnim iteracijama, da bi bili sigurni da postojeće funkcionalnosti rade ispravno i nakon implementacije novih komponenti.

Regresijsko testiranje je potrebno u slučaju:

- Novih zahtjeva u specifikaciji softvera
- Dodavanja novih funkcionalnosti
- Uklanjanja postojećih funkcionalnosti
- Ispravke defekata
- Optimizacije u svrhu poboljšanja performansi sistema

8.3.1. Tehnike regresijskog testiranja

Regresijsko testiranje spada u aktivnosti održavnja softvera, a to podrazumijeva poboljšanja funkcionalnosti i sistema kao cjeline, ispravljanje grešaka, optimizaciju ili eventualno brisanje neke od funkcionalnosti sistema.
Svaka od navedenih akcija može da dovede do nepravilnog rada sistema, pa je regresijsko testiranje obevezno.

Regresijsko testiranje se može obaviti na više načina:

- Potpuno regresijsko testiranje – uz nove testove za novu funkcionalnost, testiraju se i postojeći testovi iz prethodnih iteracija. Ovo je vrlo skup način što se tiče potrebnog vremena i resursa, pa se zbog toga neophodno smanjiti broj testova.
- Parcijalno regresijsko testiranje – biraju se i testiraju samo određeni dijelovi grupe testova, a ne svi testovi iz skupa. Odabir testova se vrši tako da se fokusiramo na prethodno implementirane funkcionalnosti koje imaju zajedničkih osobina sa funkcionalnostima koje se implementiraju u trenutnoj iteraciji.
- Prioritizacija testova – određuje se prioritet za testove na osnovu poslovnog uticaja (*eng. business impact*), kitičnih i često korištenih funkcionalnosti. Ovim načinom značajno smanjujemo broj testova za regresijsko testiranje.

Da bi se napravio dobar skup testova za regresiju, potrebno je da se poštuju sljedeća pravila prilikom odabira testova. Regresijski testovi treba da obuhvate:

- Testove koji su otkrili defekte u prethodnim iteracijama
- Funkcionalnosti sistema koje će najviše biti u upotrebi kod korisnika
- Testove koji verifikuju osnovne funkcionalnosti
- Testove funkcionalosti sa zadnjim izmjenama
- Komplenksne testove
- Testove koji provjeravaju granične vrijednosti

8.3.2. Automatizacija regresijskog testiranja

Ako softver koji se razvija ima dosta izmena, troškovi testiranja su veliki.
Manuelno testirenje nije pogodno jer produžava vrijeme potrebno za testiranje i povećava troškove. Manuelno regresijsko testiranje s vremenom postaje naporno i dosadno jer se stalno ponavljaju isti testovi.

Regresijsko testiranje je zbog toga najbolje automatizovati i uvijek se teži ka primjeni nekog alata za automatizaciju testiranja.
Obim automatizacije zavisi od broja testova koji su ponovo primjenjivi (*eng. reusable*) u višestrukim iteracijama i ciklusima regresijskog testiranja.

Kako se stalno pokreće isti skup automatskih testova, proces testiranja može postati statičan. Iz tog razloga potrebno ih je redovno ažurirati, da bi se izbjegli negativni povratni efekti.

8.3.3. Prednosti i mane regresijskog testiranja

Prednosti regresijskog testiranja:

- Osigurava da neka skorija promjena u softveru nije uticala na već postojeće funkcionalnosti
- Verifikuje da se greške iz prethodnih iteracija ne ponavljaju
- Može se automatizovati u nekoj mjeri
- Poboljšava kvalitet proizvoda

Mane regresijskog testiranja:

- Ako se regresijsko testiranje vrši manuelno, vrlo je skupo i dugotrajno
- Svaka promjena koda zahtjeva regresijsko testiranje, jer može imati negativan uticaj na druge komponente
- Ako se skup regresijskih testova ne ažurira na vrijeme, proces postaje statičan

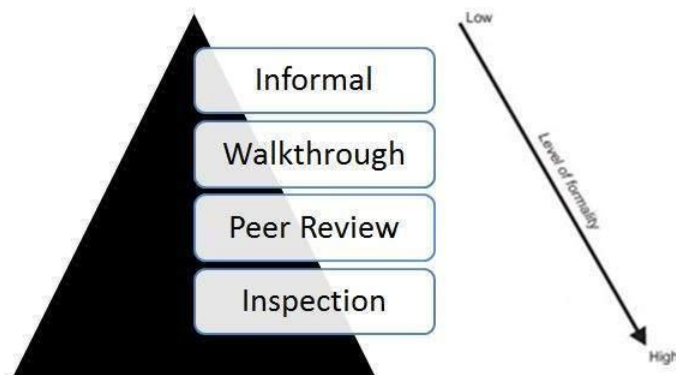
8.4. Statičko testiranje

Statičko testirnaje je skup tehnika kojim se poboljšava kvaliteta softvera, kao i efikasnost i produktivnost samog procesa razvoja softvera.
Ovaj tip testiranja se bazira na procesu statičkog pregleda (*eng. review*), s ciljem da se defekti pronađu što ranije u procesu razvoja softvera.

Statičko testiranje je testiranje softvera bez njegovog pokretanja ili izvršavanja. Statičko testiranje se može vršiti na kodu, dizajnu, modelima, funkcionalnim zahtjevima, specifikaciji zahtjeva i arhitekturi softvera.

Tipovi statičkog testiranja su:

- Neformalni pregled
- Formalni pregled
- Walkthrough
- Tehnički prgled
- Peer review
- Inspekcija



Slika 16. Tipovi statičkog testiranje prema nivou formalnosti[16]

Statičko testiranje može da počne vrlo rano u razvoju softvera, što omogućava da se potencijalne greške uoče na vrijeme. Pronađene greške je vrlo lako ispravit po pitanju uloženog vremena i troškova.

Tipovi defekata pronađeni statičkim testiranjem:

- Nejasna i nepotpuna specifikacija zahtjeva
- Greške u arhitekturi
- Greške u dizajnu
- Problemi u specifikaciji interfejsa između komponenti
- Neusklađenost s zahtjevanim standardima
- Prekomplikovan kod

[16] https://www.tutorialspoint.com/software_testing_dictionary/images/static_testing.jpg

8.4.1. Neformalni pregled

Neformalni pregled (*eng. Infromal review*) se koristi u ranim fazama razvoja dokumenata za projekat. Pod dokumentima posmatramo specifikaciju zahtjeva, dizajn i arhitekturu sistema, listu testova i sl.

Cilj neformalnog pregleda je da se taj dokument prezentuje široj publici, kao što su menadžment, klijenti, tim programera i tim testera.

U ranim fazama je dovoljno dvoje ljudi za razvoj dokumenta, a kasnije se uključuje sve više ljudi. S više ljudi organizuju se sastanci, gdje svi prisutni mogu dati svoje mišljenje, s ciljem da se poboljša kvaliteta dokumenta i otkriju defekti.

8.4.2. Formalni pregled

Ovaj proces je strukturisan i regulisan i sastoji se od šest osnovnih faza:

- Planiranje – počinje zahtjevom za pregled koji autor upućuje moderatoru. Moderator određuje datum i vrijeme sastanka. Moderator onda vrši provjeru da li je dokument spreman za pregled (*eng. entry criteria*) i definiše formalni izlazni kriterij (*eng. exit criteria*). U slučaju da dokument ne prođe ulazni kriterij vraća se autoru na doradu.
- *Kick-off* – ovo je opcionalni korak u procesu. Cilj ovog sastanka je prezentovanje kratkog uvoda o ciljevima pregleda i dokumentima svima uključenim u sastanak.
- Priprema – svaki član tima pregleda dokument. Koriste se odgovarajuće procedure i pravila da bi verifikovali sve potrebne stavke dokumenta.
- Formalni sastanak – sastoji se od faze logovanja, faze diskusije i faze odlučivanja. U fazi logovanja se evidentiraju svi potencijalni defekti. Ako neki problem zahtjeva diskusiju to se radi u fazi diskusije, što se evdentira za kasnije praćenje. Na kraju, u fazi odlučivanja donosi se odluka o dokumentu koji se pregled. Ako dokument ne ispunjava izlazni kriterij on se vraća autoru i pregleda ponovo kad se ispravi.
- Ispravka – ako je pronađeni broj defekata izvan dozvoljenog nivoa, dokument mora biti ispravljen.
- *Follow-up* – moderator provjerava da li je autor poduzeo određene akcije nad prijavljenim defektima.

Navedene faze se izvršavaju u zadatom rasporedu i ovaj tip pregleda se obavezno dokumentuje.
Ovu strukturu procesa prate svi oblici formalnog pregleda, a svi učenici imaju svoje uloge i zaduženja.

8.4.3. Walkthrough

Walkthrough spada u neformalni proces. Ovaj sastanak vodi sam autor dokumenta.
Autor sa svim učesnicima sastanka prolazi kroz dokument, da se postigne razumijevanje teme i da bi se prikupile povratne informacije.
Ovaj tip pregleda je pogodan za ljude koji nisu stručni u razvoju softvera, odnosno klijente za koje se softver pravi.

Cilj *walkthrough* procesa je:

- Prezentacija dokumenta da bi se dobile povratne informacije o temi dokumentacije
- Prenošenje znanja
- Evaluacija sadržaja dokumenta
- Diskusija o predloženim rješenjima

8.4.4. Tehnički pregled

Tehnički pregled spada u formalni proces, a vodi ga moderator ili tehnički ekspert.
Ovaj proces se obavlja u formi *peer review*, bez prisustva menadžera.
Peer review je tip pregleda, gdje dokument pregleda autor i jedan ili više kolega, da se evaluira tehnični sadržaj i kvalitet dokumenta.

Ciljevi tehničkog pregleda su:

- Verifikacija upotrebe tehničkih koncepata
- Procjena i provjera upotrebe tehničkih koncepata
- Upoznavanje učesnika sa detaljnim tehničkim sadržajem dokumenta

8.4.5. Inspekcija

Inspekcija je najformalniji tip pregleda. Ove preglede vode trenirani moderatori.
Dokumenti se za vrijeme inspekcije pripremaju i detaljno pregledaju od strane inspektora prije sastanka, a defekti koje pronađe se obavezno dokumentuju u listu.

8.5. Smoke testing

Smoke test je oblik testiranja softvera koji provjerava da li osnovne funkcionalnosti softvera rade na unaprijed određen način. Ovaj oblik testiranja nije detaljan, niti ulazi u dubinu funkcionalnosti. Izvršava se mali broj testova samo da bi se provjerilo da li osnovne funkcije softvera rade. Ovaj oblik testiranja se koristi kao osnovna provjera da li je softver dovoljno stabilan i spreman za dalje testiranje.
Termin smoke test odnosno testiranje da li ima dima, vuče svoje korjene iz testiranja hardverskih komponenti. Prilikom prvog uključivanja hardvera, prvo se vizuelno provjerava da li ima dima ili varnica da bi se utvrdilo da li hardver ima neke krititčne mane u spojevima.

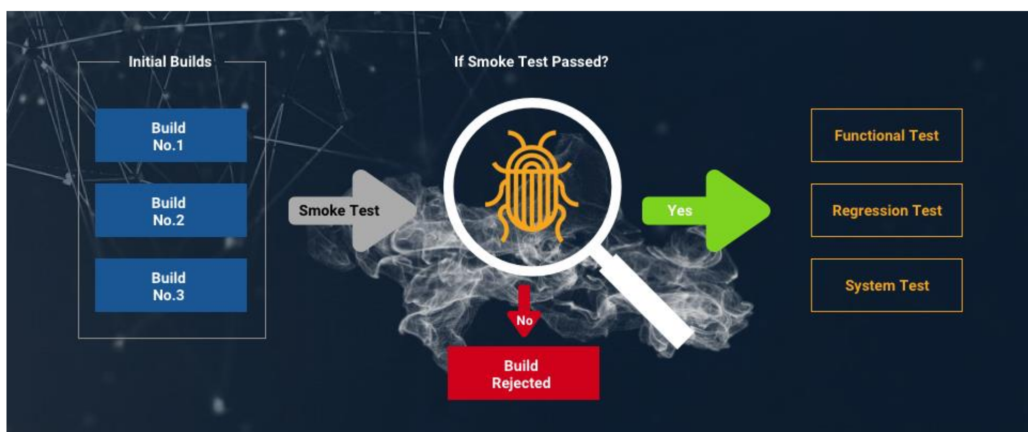
Smoke test može da otkrije u ranoj fazi da li softver ima neki katastofalni problem (*engl. showstopper*). Ako problem postoji nema neke svrhe počinjati dalje detaljnije testiranje. Ovo testiranje je u domenu developera, koji na ovaj način provjeravaju stabilnost softvera.
Pošto smoke test ne obuhvata detaljno testiranje, obično se radi samo srećna putanja (*eng. happy path*), odnosno prolazi se kroz osnovne funkcionalnosti i korisničke scenarije sa validnim ulaznim podacima.

Prednosti smoke testiranja:

- Kritične defekte je moguće uočiti u ranoj fazi testiranja
- Utvđuje se uticaj defekata iz prethodnih iteracija na funkcionalnosti softvera
- Mali broj testova
- Kratko vrijeme testiranja

Mane smoke testiranja:

- Nije dovoljno detaljno
- Nije moguće otkriti sve kritične bugove zbog malog broja testova koje obuhvata
- Ne obuhvata negativne scenarije ili testiranje sa neispravnim podacima



Slika 17. Smoke test[17]

[17] https://www.google.com/url?sa=i&source=images&cd=&ved=2ahUKEwjh_Yrb9J3jAhWxM-wKHf41B_AQjRx6BAgBEAU&url=https%3A%2F%2Freqtest.com%2Ftesting-blog%2Fsmoke-testing-2%2F&psig=AOvVaw2nDTDBm8ceyNOl0jZmpVCa&ust=1562420638385901

8.6. Sanity testiranje

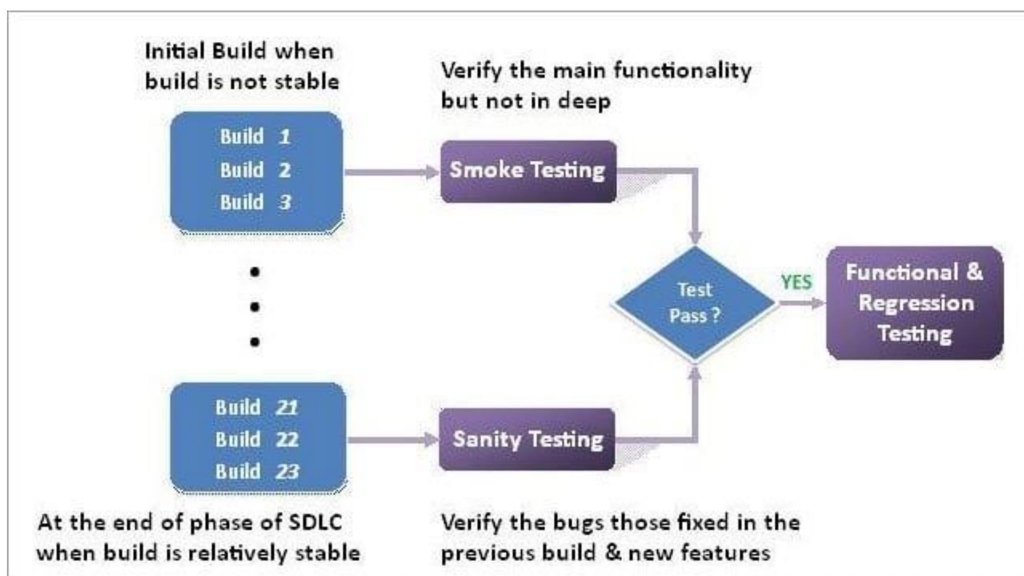
Test zdravog razuma (*eng. Sanity test*) je sličan smoke testu. Ovi termini se često miješaju u testiranju softvera. Iako razlike između ova dva tipa nisu velike, one ipak postoje.

Smoke test se koristi za testiranje builda softvera da bi se verifikovao ispravan rad osnovnih funkcionalnosti i izvršava se prije bilo kojeg detaljnog testiranja, kako testeri ne bi gubili vrijeme u slučaju nekih kritičnih problema.
Sanity test se izvršava na relativno stabilnim softverom. Ovaj tip testiranja vrše testeri nakon primanja builda u kome postoje izmjene u kodu ili je dodana nova funkcionalnost.

Cilj sanity testiranja je da se utvrdi da li dodana funkcionalnost radi kako treba i da osnovne funkcionalnosti softvera nisu ugrožene. Ako sanity test ne prođe, build se odbija i ne počinje se dalje testiranje, čime se štedi vrijeme i smanjuju troškovi.

I smoke i sanity testiranje služe da bi se utvrdilo da li je softver dovoljno stabilan za detaljno testiranje, odnosno da se uštedi vrijeme u slučaju da postoji neka kritična greška. Oba tipa testiranja se obavljaju nad istim buildom, prvo se izvršava smoke a nakon njega sanity.

Ovi tipovi testiranja se mogu izvršiti i manuelno ili nekim alatom za automatizaciju. U slučaju da se koristi neki alat za automatizaciju, najčešće se pokretanje tih testova inicijalizira s istim procesom koji pravi sam build.



Slika18. Sanity test[18]

[18] https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/02/SmokeTesting.jpg

9. Zaključak

Testiranje je sastavni dio životnog ciklusa svakog softvera. Da bi povećali kvalitetu i smanjili greške u razvoju softvera neophodno je testiranje u svim fazama izrade softvera.
Iako se trenutno vodi rasprava koje je bolje automatsko ili manuelno testiranje, iz navedenog možemo zaključiti da svaki oblik testiranja ima svojim prednosti i mana. Ali neosporno je da za uspješno testirnje potrebo je uključiti oba oblika testiranja.

Dok govoreći o tipovima testiranja koje treba primjeniti na razvoju nekog softvera zavisi od razvojnog tima koji bi trebao odlučiti koji od navedenih tipova najbolje odgovara njihovom sistemu rada. U ovom radu si navedeni najvažniji tipovi testiranja, njihova podjela i definicija.
Za uspješno testiranje softvera potrebno je da si članovi tima sarađuju i iznose svoje mišljenje i ideje.

10. Literatura

[1] Manger, R. : Softversko inženjerstvom, Zagred, 2005
[2] Mayers, G. : The Art of Software Testing, Word Association, New Jersey, 2004
[3] Živković, M. : Testiranje softvera, Beograd, 2018

Internet sajtovi:
[1] https://www.guru99.com/alpha-beta-testing-demystified.html
[2] https://www.guru99.com/smoke-sanity-testing.html
[3] https://dtc.rs/sr/testiranje-softvera
[4] https://en.wikipedia.org/wiki/Test-driven_development
[5]https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/

Popis slika:
Slika 1. Prvi zabilježeni računarski bug, preuzeto sa: https://cdn0.tnwcdn.com/wp-content/blogs.dir/1/files/2013/09/bug.jpg
Slika 2. Životni ciklus softvera, preuzeto sa: https://i.ytimg.com/vi/3Lxnn0O3xaM/hqdefault.jpg
Slika 3. Životni ciklus testiranja softvera, preuzeto sa: https://qph.fs.quoracdn.net/main-qimg-9cd90cdcdf317da64d9c43faa3087fee.webp
Slika 4. Faze razvoja softvera vođenog testiranjem, preuzeto sa: https://i.ytimg.com/vi/T38L7A0xP-c/maxresdefault.jpg
Slika 5. Proces manuelnog testiranja, preuzeto sa: http://www.professionalqa.com/assets/images/manual-testing.jpg
Slika 6. Proces automatskog testiranja, preuzeto sa: https://www.joecolantonio.com/wp-content/uploads/2018/11/AutomationTestingProcess.png
Slika 7. Nivoi testiranje softvera, preuzeto sa: http://softwaretestingfundamentals.com/wp-content/uploads/2011/01/software_testing_levels1.jpg
Slika 8. Integracijsko testiranje, preuzeto sa: https://www.guru99.com/images/3-2016/032816_1230_SystemInteg1.png
Slika 9. Big Bang integracijski pristup, preuzeto sa: https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/05/Big-bang-approach.jpg
Slika 10. Tipovi inkrementalne integracije, preuzeto sa: https://bitwaretechnologies.com/wp-content/uploads/2017/02/FRRE.png
Slika 11. Sistemsko testiranje, preuzeto sa: https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/05/system-testing-example.jpg
Slika 12. Metoda crne kutije, preuzeto sa: https://phoenixnap.com/blog/wp-content/uploads/2018/11/black-approach.jpg
Slika 13. Metoda bijele kutije, preuzeto sa: https://phoenixnap.com/blog/wp-content/uploads/2018/11/white-testing-diagram.jpg

Slika 14. Životni ciklus performance testiranja, preuzeto sa: https://qainsights.com/wp-content/uploads/2012/11/Performance-Testing-Life-Cycle-QAInsights.png
Slika 15. UAT, preuzeto sa: https://images.xenonstack.com/insights/xenonstack-what-is-user-acceptance-testing.png
Slika 16. Tipovi statičkog testiranje prema nivou formalnosti, preuzeto sa: https://www.tutorialspoint.com/software_testing_dictionary/images/static_testing.jpg
Slika 17. Smoke test, preuzeto sa: https://www.google.com/url?sa=i&source=images&cd=&ved=2ahUKEwjh_Yrb9J3jAhWxM-wKHf41B_AQjRx6BAgBEAU&url=https%3A%2F%2Freqtest.com%2Ftesting-blog%2Fsmoke-testing-2%2F&psig=AOvVaw2nDTDBm8ceyNOl0jZmpVCa&ust=1562420638385901
Slika18. Sanity test, preuzeto sa: https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/02/SmokeTesting.jpg