Testiranje programskih proizvoda

Dudaš, Josip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Dudaš

TESTIRANJE PROGRAMSKIH PROIZVODA

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Dudaš

Matični broj: 0016108257

Studij: Organizacija poslovnih sustava

TESTIRANJE PROGRAMSKIH PROIZVODA

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Vjeran Strahonja

Varaždin, kolovoz 2019.

Josip Dudaš

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad obuhvaća temu testiranja programskih proizvoda kroz teoretski i praktični dio. Prvi dio rada odnosi se na približavanje pojma testiranja programskih proizvoda, njegove vrste i metode te procese i standarde testiranja. Na samom početku teoretskog dijela nalazi se uvod u temu koji čitatelje uvodi i približava im osnove testiranja. Zatim slijedi povijest testiranja koja omogućuje čitateljima lakše uživljavanje u temu.

Nakon uvoda i povijesti kreće razrada teme testiranja programskih proizvoda te otkrivanje značaja testiranja za kreiranje što boljeg konačnog proizvoda. Naglašavaju se činjenice koje su važne i na koje treba obratiti posebnu pažnju prilikom testiranja. Uz to navode se najčešće i najbolje prakse iz stvarnog svijeta.

Drugi dio rada odnosi se na praktični dio. U tom dijelu rada se primjenjuju različiti pristupi testiranju i obavlja testiranje mobilne aplikacije. Navedeni dio će obuhvatiti plan testiranja, pripremu testnih slučajeva, izradu testnih scenarija i skripta, pripremu podatka i provedbu samog testiranja. Mobilna aplikacija na kojoj je izvršeno testiranje izrađena je samo za ovaj rad te obuhvaća osnovne funkcionalnosti kako bi se omogućilo prikazivanje potrebnih testova.

Konačni cilj rada je prikazati važnost i primjenu testiranja programskih proizvoda.

Ključne riječi: testiranje programskih proizvoda, kvaliteta programa, metode testiranja, tehnike testiranja, agilno testiranje, funkcionalno testiranje, nefunkcionalno testiranje

Sadržaj

1. Uvod

Testiranje je neizostavni dio životnog ciklusa bilo kojeg proizvoda. Vrlo vjerojatno nećete sjesti u avion koji nije prije toga testiran ili čije komponente od kojih je on sagrađen nisu testirane. Ista logika se može primijeniti na programske proizvode. Testiranje programskih proizvoda je tema koja seže u razvoj svakog programskog proizvoda na svijetu. To je dio izrade programskog proizvoda koji utječe na izgled, karakteristike, sigurnost i mnogobrojne druge parametre konačnog proizvoda. Ako želimo da naš konačni proizvod bude što kvalitetniji sa što manje grešaka i nedostataka tada ulažemo napor u što detaljnije testiranje.

Testiranjem programa nećemo i ne možemo dobiti program koji radi bez ijedne greške jer takvi programi ne postoje, ali ćemo dobiti program koji će smanjiti mogućnost pojave i otkrivanja greške od strane korisnika. S jedne strane ćemo mi kao graditelji programa biti sretni jer smo ispunili zahtjeve i očekivanja korisnika, dok će s druge strane krajnji korisnici biti zadovoljni jer su svi njihovi zahtjevi ispunjeni i program obavlja zadaću koja mu je zadana.

Testiranje programskih proizvoda je proces koji obuhvaća planiranje testiranja, izradu testnih slučajeva, testiranje, prikupljanje podataka te na samom kraju analizu dobivenih rezultata i otkrivanje grešaka u radu programa. Navedenim procesom pokušava se dostići krajnji cilj, a to je otkrivanje što većeg broja grešaka, a ne potvrđivanje da program radi kako treba tj. otkrivanje što manjeg broja grešaka. Vrlo važno je postaviti ispravan cilj iz kojeg krećete u testiranje. Ako u testiranje krećete s mišljenjem kako navedenim testiranjem trebate potvrditi da program radi kako treba tada će te nesvjesno izbjeći greške. No ako u testiranje krećete s mišljenjem da vam je glavni cilj testiranja otkrivanje što većeg broja grešaka tada će to rezultirati puno boljim rezultatom. Pronaći će te veći broj grešaka koje će biti sanirane u nekom drugom postupku i konačni program bit će kvalitetniji.

Tema testiranja programskih proizvoda gotovo je neizostavna prilikom razvoja programa. Gotovo da ne postoji program na svijetu koji ne bi trebao biti testiran. Počevši od jednostavnih pa sve do kompleksnih programa. Kao primjer možemo uzeti program u koji upisujete dva broja te vam on ispisuje sumu unesenih brojeva. Na prvu navedeni program je jednostavan i možemo se zapitati da li ga ima smisla uopće testirati. No ako malo bolje razmislimo vidjet ćemo da bi mogli ispitati puno stavki vezanih uz ovaj program. Na primjer koji raspon vrijednosti brojeva možemo unijeti, da li program prihvaća druge znakove osim brojeva, da li radi sa negativnim brojevima, itd. Vidljivo je kako i mali programi mogu postati vrlo kompleksni ukoliko se ne upravlja s njima kako treba. Iz tog razloga je testiranje iznimno važno za razvoj programa.

Kada ne bi postojalo testiranje, programi bi vrlo brzo postali upotrebljivi samo za njihove tvorce (programere). Naime programeri bi izrađivali programe koji bi u njihovim glavama i njihovim rukama radili ono što oni smatraju da ti programi trebaju raditi, a ne ono što je krajnji korisnik želio i naručio.

Vrlo važno je napomenuti kako je proces testiranja iterativni postupak koji se ponavlja više puta tijekom razvoja programa. Ovaj način izvršavanja testiranja je iznimno važan kod agilnih metoda razvoja programa koje su sve zastupljenije u svijetu.

U ovom radu je opisana struktura testiranja programskih proizvoda. Prvi dio rada obuhvaća teoretsku obrada teme. Točnije obuhvaća se pojam testiranja programskih proizvoda, vrste i metode te procesi i standardi testiranja. Kako ne bi sve ostalo na pukoj teoriji drugi dio rada će obuhvatiti primjere testiranja te će se uz pomoć njih približiti obrađeno gradivo iz teoretskog dijela rada.

2. Povijest testiranja programskih proizvoda

Povijest testiranja programskih proizvoda javlja se sredinom 20. stoljeća sa razvojem prvih digitalnih računala. Nakon što je 1946. godine napravljen prvi digitalni kompjuter koji je mogao odrađivati složenije zadatke i nakon što su u svijetu shvatili mogućnosti koje im se otvaraju javilo se pitanje testiranja računalnih programa koji su izvršavani na njima. [1]

Prva greška koja se pojavila u kompjuteru dobila je naziv po engleskom nazivu za kukca (eng. bug). [1] Naime kukce privlači toplina i svjetlost, a upravo su te dvije stvari nusproizvodi prvih digitalnih računala. Pa se tako često događalo da kukci odlete na računalne komponente i stvore kratki spoj te se program ne može izvršiti. Tada su na scenu stupali ljudi koji su tražili kukce i otklanjali ih sa komponenata te tako osiguravali daljnji rad računala. Ova priča može se smatrati prekretnicom koja je pokazala kako se u izvršavanju programa mogu javiti greške koje treba otkloniti. Kako bi smanjili pojavu grešaka formiraju se prvi timovi koji služe za testiranje sustava i pronalazak grešaka.

Prva knjiga koja opisuje temu testiranja javlja se 1979. godine pod nazivom „The Art of Software Testing“ te je napisana od strane Myersa. [2] Navedena knjiga je bila prva koja je govorila u potpunosti o testiranju programskih proizvoda. Postavila je temelje modernog načina testiranja programa. Najznačajnija novina koja je proizašla iz ove knjige je testiranje metodom crne kutije koja do izlaska knjige nije bila opisana niti predstavljena javnosti.

Nakon te knjige testiranje postaje sve važnija tema u svijetu programiranja. Ljudi koji su uključeni u razvoj i održavanje programa postaju svjesni kako se za niti jedan program na svijetu ne može sa 100% sigurnošću reći kako radi bez ijedne greške. Sve više knjiga i znanstvenih članaka objavljuje se na temu testiranja programskih proizvoda. Otkrivaju se nove tehnike i metode, te usavršavaju procesi izrade testova. Također se razdjeljuje testiranje od otklanjanja grešaka (eng. debugging). Na taj se način testerima osigurava neovisnost o programu koji se testira. U samim počecima testiranja programskih proizvoda programer je bio zadužen za testiranje. Navedena praksa je danas u potpunosti okrenuta jer se pokazala dosta loša. Razlog tomu je što programeri nesvjesno izbjegavaju greške u programu ako znaju za njih. Dok su testeri s druge strane neovisni tj. oni nisu napisali kod i program koji se testira te na taj način ne izbjegavaju greške koje se nalaze u programu. [2]

Kroz povijest je vrlo lako prepoznati poveznicu između razvoja programskih proizvoda i tehnologije sa razvojem testiranja. U globalu sa svakom novom tehnologijom dolazi nova metoda i tehnika razvoja programa što u konačnici dovodi do nove metode i tehnike testiranja programa. U današnjem svijetu navedena poveznica je još izraženija jer je došlo vrijeme u kojem svakih nekoliko mjeseci dolazi do razvoja novih tehnologija koje testeri moraju ispitati.

3. Testiranje programskih proizvoda i povezani pojmovi

Kako bi što bolje razumjeli testiranje programskih proizvoda potrebno je razumjeti osnovne pojmove vezane uz proces testiranja. U daljnjem tekstu su objašnjeni najvažniji pojmovi koji se u tom procesu javljaju.

3.1. Testiranje programskih proizvoda

U svijetu postoji puno različitih definicija testiranja programskih proizvoda. Kako bi što bolje razumjeli testiranje u ovom poglavlju prikazane su dvije definicije koje dovoljno detaljno opisuju pojam testiranja. Prva definicija kaže:

„Testiranje je proces izvršavanja programa s namjerom pronalaženja pogreške.“ [3, p. 6]

Navedenom definicijom naglašava se namjera pronalaženja pogreške u radu programa. Nema spomena drugih bitnih karakteristika testiranja poput osiguravanja kvalitete, zadovoljenja zahtjeva ili neometanog rada programa. Vrlo je kratka i jasna za osobe koje su upućenije u temu testiranja te mogu izvući dublju poruku iz navedene definicije. Dok s druge strane navedena definicija bi mogla stvoriti pogrešnu ideju kod osoba koje nisu upućene u razvoj i testiranje proizvoda.

International Software Testing Qualifications Bord (ISTQB) organizacija koja se bavi certificiranjem ljudi za testiranje programskih proizvoda navodi drugu definiciju koja opisuje testiranje na opsežniji i sadržajniji način:

„Testiranje je proces koji se sastoji od svih aktivnosti životnog ciklusa razvoja proizvoda, kako statičkih tako i dinamičkih, koje se tiču planiranja, pripreme i procjene programskih i srodnih proizvoda kako bi se utvrdilo da zadovoljavaju određene zahtjeve, da su prikladni za svoju svrhu i da se otkriju njihovi nedostaci.“ [4]

Puno je sadržajnija i konkretnija od prve definicije. U ovoj definiciji naglašeno je da je testiranje uključeno u sve faze životnog ciklusa razvoja proizvoda, da se testiranjem osigurava zadovoljenje određenih zahtjeva i da se testiranjem pronalaze greške u proizvodu. Mogli bi ustvrditi da je navedena definicija pokrila sve važne stavke samog testiranja te omogućuje početniku lakše shvaćanje procesa testiranja.

3.2. Testeri

Nakon upoznavanja sa pojmom testiranja potrebno je razjasniti pojam testera. Testeri su neizostavni dio procesa testiranja. Tester je „kvalificirani stručnjak koji sudjeluje u testiranju komponente ili sustava“. [5] Naime niti jedan test ne može biti izveden bez testera, odnosno osobe koja provodi test. Testeri moraju imati dosta širok spektar znanja kako bi mogli ovladati procesom testiranja. S obzirom na vrstu testa postoje različite karakteristike koje bi testeri trebali posjedovati. Tako npr. za provođenje jediničnih i integracijskih testova testeri bi u pravilu trebali biti programeri tj. imati programerske vještine, dok s druge strane navedene vještine nisu presudne kod provođenja testova prihvaćanja. [3]

U svijetu postoji nekoliko organizacija koje se bave certificiranjem ljudi za testiranje programskih proizvoda. Među njima najpoznatije su ISTQB odnosno punim imenom International Software Testing Qualifications Bord i EUCIP odnosno punim imenom European Certification of Informatics Professionals. Svaka od tih organizacija ima svoj skup vještina kojima testeri moraju ovladati da bi postali certificirani članovi navedene organizacije. Certifikatima se zapravo potvrđuje da je dana osoba ovladala potrebnim vještinama i da samostalno može obavljati posao testera. [4] O točnim kvalifikacijama i vještinama testera više govora u poglavlju „Testne smjernice“.

3.3. Anomalije

U procesu testiranja postoji jako puno sličnih pojmova koji se često javljaju, a tiču se anomalija koje se javljaju u radu programa. Anomalije predstavljaju stanje koje odstupa od očekivanog temeljenog na specifikacijama zahtjeva, dokumentaciji, standardima, itd. Anomalije se mogu pronaći tijekom testiranja ili tijekom korištenja programskog proizvoda. Neke od anomalija su: pogreška (eng. bug), nedostatak (eng. defect), odstupanje (eng. deviation), kvar (eng. error), zastajanje/ispad (eng. failure), incident (eng. incident), propust (eng. problem). [6]

- **Pogreška, propust i nedostatak** predstavljaju „pogrešku komponente ili sustava koja može uzrokovati neizvršavanje tražene funkcije komponente ili sustava, npr. neispravna linija ili definicija podataka.“ [6]
- **Odstupanje, incident** označava događaj čije pojavljivanje zahtjeva istragu i djelovanje. [6]
- **Kvar** predstavlja „ljudsko djelovanje koje proizvodi neispravan rezultat.“ [6] Kod kvara valja istaknuti kako je ljudski faktor presudan tj. kako do kvara dolazi ljudskom pogreškom.

- **Zatajenje, ispad** predstavlja odstupanje komponente ili sustava od zahtijevanog ponašanja.

3.4. Testiranje nasuprot otklanjanja grešaka

Testiranja i otklanjanja grešaka (eng. debuging) su pojmovi koji se vrlo često miješaju u praksi. Testiranje je proces pronalaženja grešaka, dok je otklanjanje grešaka proces lociranja i otklanjanja grešaka. [3] Testeri imaju zadaću testiranja, a ne zadaću otklanjanja grešaka. Njihov primarni i jedini posao je otkrivanje što većeg broja grešaka. Dok osobe koje su zadužene za otklanjanje pogrešaka imaju obrnutu zadaću. Oni su zaduženi isključivo za lociranje i otklanjanje grešaka, a ne za njihovo pronalaženje. Osobe za otklanjanje grešaka su u većini slučajeva programeri, dok testeri mogu ali i ne moraju biti programeri. [3] U praksi se dosta često može pronaći da su svi testeri u poduzeću programeri. Navedeno situacija je dobra u određenim testovima poput testova bijele kutije (jedinični i integracijski testovi), no u većini testova to nije dobro za programski proizvod. Ona osoba koja piše kod, ne bi trebala testirati sama svoj kod.

3.5. Osiguranje kvalitete

Uz testiranje se veže pojam osiguranja kvalitete (eng. quality assurance). Navedeni pojam je dio životnog ciklusa testiranja programskog proizvoda (eng. Software Test Life Cycle). Obuhvaća skup aktivnosti koje osiguravaju provedbu procesa i postupaka u kontekstu provjere razvijenog programskog proizvoda i predviđenih zahtjeva. Vrlo je važno napomenuti kako osiguranje kvalitete ne obuhvaća sam čin testiranja već se usredotočuje na definiranje, osiguravanje i nadgledanje postupaka i procesa koji su predviđeni za osiguranje kvalitete. Možemo zaključiti kako osiguranje kvalitete zapravo diktira procesom testiranja ali ga ne provodi. [7]

3.6. Kontrola kvalitete

Kontrola kvalitete sadrži skup aktivnosti koje osiguravaju provjeru razvijenog programskog proizvoda s obzirom na zahtjeve. Aktivnosti kontrole kvalitete se mogu smatrati korektivnima jer ispravljaju moguće greške u proizvodu. Važno je da postoje dokumentirani zahtjevi te se u skladu s njima provjerava ispravnost programskog proizvoda. Kontrola kvalitete je zapravo testiranje sa ciljem pronalazaka grešaka, te je ona pod skup osiguranja kvalitete. [7]

4. Cilj testiranja

Svaki faza razvoja programskih proizvoda sadržava cilj kojem teži, tako je i kod testiranja. Mogli bismo reći da je primarni cilj testiranja otkrivanje grešaka. Navedena konstatacija je točno, no ima još skrivenih detalja koje se moraju ispuniti odnosno dostići kako bi mogli reći da je cilj testiranja ostvaren. Osim otkrivanja grešaka cilj testiranja je upravljanje kvalitetom proizvoda odnosno ispitivanje da li proizvod radi kako je zamišljen.

U tom segmentu se javljaju dva nova pojma: verifikacija i validacija programskog proizvoda. Verifikacija i validacija su procesi koji se provode kroz sve faze razvoja programskog proizvoda. To su procesi provjere i analize. Verifikacija je provjera da li je programski proizvod usklađen sa specifikacijom. [6] Odnosno jednostavnijim rječnikom možemo reći kako verifikacijom ispitujemo funkcionalne i nefunkcionalne zahtjeve proizvoda. Može se izvoditi nakon svakog iteracijskog ciklusa. Izvršava se u većini od strane programera i odgovara na pitanje: Da li gradimo proizvod na pravilan način?

Validacija je s druge strane općenita provjera usklađenosti programskog proizvoda s očekivanjima korisnika. [6] Slijedi nakon verifikacije i u globalu označava provjeru cjelokupnog proizvoda. Izvršava se od strane testera i odgovara na pitanje: Da li gradimo pravi proizvod?

Verifikaciju i validaciju možemo smatrati prvim korakom cilja kojem težimo kod testiranja. Drugi korak je pokrivenost prioriteta koje si poduzeće zada za navedeno testiranje. Za svako testiranje poduzeće mora definirati granice do kojih će ići i nakon kojih će to testiranje biti uspješno odrađeno. Navedene granice se mogu postaviti kao broj pronađenih grešaka, postotak obuhvaćenog koda, broj obuhvaćenih funkcionalnosti i slično. Kod navedenog koraka je također bitna efikasnost i efektivnost samih testova. Ne možemo biti zadovoljni ako nešto postignemo u jako dugom periodu vremena ili ako otkrijemo greške koje smo morali otkriti puno prije. Navedeni aspekt ne smijemo nikako zaboraviti.

Paretovo pravilo možemo primijeniti u testiranju kao i u mnogobrojnim drugim zanimanjima i poslovima. Navedeno pravilo za testiranje kaže kako je 80% svih programskih grešaka sadržano u 20% programskih komponenata. [8] Iz navedenog pravila bi mogli zaključiti kako proces testiranja mora biti dobro osmišljen i balansiran. Potrebno je osmisliti testove koji će se fokusirati na one dijelove programskog proizvoda koji su pogodniji greškama. Naravno da mi ne možemo znati unaprijed di se nalaze greške, ali možemo iz prethodnih testova i najboljih praksi doći do zaključka kako su određeni dijelovi programa pogodniji za razvijanje grešaka.

5. Svrha testiranja

Testiranje je proces koji se nipošto ne smije izostaviti iz razvoja programskih proizvoda. U gotovo svim industrijama i granama koje imaju kao output proizvod pa čak i uslugu mora postojati proces koji će iskušati što taj proizvod može, koje su mu granice, koliko je izdržljiv, koje radnje nad njim dovode do grešaka i slično. Kada ne bi postojao navedeni proces korisnici bi izgubili vjeru u te proizvode, ne bi ih kupovali i koristili, ne bi bili sigurni da se neće korištenjem navedenog proizvoda ozlijediti i slično. Sve te radnje bi upropastile proizvode ali i kompanije koje stoje iza toga.

Testiranje u svijetu programiranja sa pogleda programera na prvu može izgledati beskorisno. Programeri koji pišu kod koji se provjerava mogu tvrditi kako taj kod radi u svim situacijama i kako program radi na način kako je korisnik želio. Da, to može biti točno, ali i ne mora. Točnije, u većini slučajeva kod će imati greške, a finalni proizvod neće odgovarati 100% korisnikovim željama. Uz navedeno da kod i program moraju raditi savršeno testiranje pokriva dio koji se odnosni na održavanje. Većina proizvoda nije napravljena za kratkotrajno korištenje već je namijenjena za korištenje duži period. U tom periodu se može promijeniti nekoliko ljudi na tom istom projektu. Navedeni ljudi moraju biti upoznati sa kodom koji je napisan, sa funkcionalnostima koje taj proizvod pruža i moraju moći odraditi izmjene nad tim proizvodom. Kako se ne bi morali igrati sa kodom i kako ne bi morali trošiti nepotrebno vrijeme na ispravljanje svojih grešaka koje se mogu pojaviti zbog nepoznavanja koda pojavljuju se testovi. Testovi se izvrsno koriste u održavanju programskih proizvoda. Krenuvši od jediničnih testova koji novim programerima pomažu u upoznavanju i savladavanju napisanog koda, pa sve do sustavskih testova koji pomažu novim ljudima na projektu da dobiju sliku o krajnjim ciljevima koje taj proizvod mora zadovoljiti. U segmentu održavanje koda jako veliku ulogu igra dokumentiranje testova. Ne vrijedi ne bilježiti krajnje rezultate provedenih testova. S time se ne postiže nikakva dugotrajna korist za programski proizvod. Korist se u punom sjaju postiže onda kada se svaki testni slučaj i rezultat spremaju i čuvaju za buduće nadogradnje i izmjene programskog proizvoda. U svijetu tehnologije događaju se brze promjene. U tom slučaju također uskaču testovi koji mogu pomoći programerima da lakše odrede koliki dio koda moraju i smiju izbaciti, a da ostali dio koda ostane netaknut i da proizvod radi kako treba. [3]

Uz održavanje i razvoj programskih proizvoda testiranje ima svrhu osiguranja ugleda poduzeća. Naime ukoliko ne bi postojalo testiranje, poduzeće bi riskiralo svoj ugled na tržištu. Jer nitko ne može niti uz provedene testove garantirati 100% rad svog proizvoda, odnosno rad bez grešaka. A navedeno stanje se pogoršava kada se testovi ne provode ili kada se iz njihovih rezultata ne izvuku zaključci.

6. Standardi

U svijetu postoji puno organizacija koje se bave razvojem i implementacijom standarda u području upravljanja kvalitetom programskih proizvoda. Navedene organizacije postavljaju pravila, ograničenja i najbolje prakse koje treba slijediti kako bi finalni proizvod bio što kvalitetniji. Prema međunarodnoj organizaciji za standardizaciju (eng. International Organization for Standardization – ISO) definicija standarda glasi: "Standardi su dokumentirane smjernice koje sadrže sporazume o proizvodima, praksama ili operacijama od strane nacionalnih ili međunarodno priznatih industrijskih, profesionalnih, trgovinskih udruženja ili vladinih tijela". [9]

Svako poduzeće koje se odluči na dobivanje certifikata iz upravljanja kvalitetom mora poštivati upute koje im nadležna organizacija savjetuje. Ukoliko poduzeće ispuni sve uvjete pojedinog standarda za uzvrat dobiva certifikat koji služi kao potvrda upravljanja kvalitetom. Navedeni certifikati su vrlo cijenjeni u poslovnom svijetu jer se njima dokazuje da poduzeće slijedi najbolje prakse i da upravlja kvalitetom na pravilan način. Korisnici programskih proizvoda više vjeruju poduzećima koje slijede standarde i primjenjuju najbolje prakse u razvoju proizvoda. S druge strane poduzećima standardi olakšavaju izvedbu procesa i aktivnosti u upravljanju kvalitetom.

Međunarodna organizacija za standardizaciju zajedno sa međunarodnom elektrotehničkom komisijom (eng. International Electrotechnical Commission – IEC) ima nekoliko standarda koji se odnose na upravljanje kvalitetom programskih proizvoda. U daljnjem tekstu su objašnjeni navedeni standardi i njihove pod karakteristike.

6.1. ISO/IEC 9126

ISO/IEC 9126 je standard koji se odnosi na programsko inženjerstvo i kvalitetu proizvoda. Navedeni standard se sastoji od:

1. Modela kvalitete (eng. quality model)
2. Eksternih mjerenja (eng. external metrics)
3. Internih mjerenja (eng. internal metrics)
4. Mjerenja kvalitete u upotrebi (eng. quality in use metrics) [10]

Model kvalitete je dio u kojem se predstavlja navedeni standard i što se može očekivati od njega. Opisani su dijelovi od kojih je sačinjena kvaliteta. Pa tako kaže da se kvaliteta može podijeliti i promatrati kroz dva odvojena dijela. Prvi dio su interne i eksterne kvalitete, dok su drugi dio kvalitete u korištenju.

Interne i eksterne kvalitete se dalje dijele na karakteristike funkcionalnosti, pouzdanosti, upotrebljivosti, učinkovitosti, održivosti i prenosivosti. Svaka od tih karakteristika razlaže se na pod karakteristike i metrike. Pod karakteristike objašnjavaju i detaljiziraju svaku od karakteristika s obzirom da li se odnose na interne ili eksterne kvalitete. Dok metrike definiraju točne algoritme, formule pomoću kojih se izvršava mjerenje kvalitete. [10]

Interne kvalitete su one koje sagledavaju proizvod iznutra, dok eksterne kvalitete sagledavaju proizvod izvana. Interne kvalitete u programskim proizvodima se odnose na kod i način kako je program napravljen dok se eksterne kvalitete odnose na rezultate testova koji se izvode nad programskim proizvodom. [10]

Za razliku od internih i eksternih kvaliteta kvalitete u korištenju predstavljaju korisnikov pogled na kvalitetu programskog proizvoda i to u točno određenom okruženju i kontekstu. Navedena kvaliteta mjeri u kojoj mjeri korisnici mogu ostvariti svoje ciljeve korištenjem navedenog proizvoda. Ona se dalje dijeli na karakteristike efektivnosti, produktivnosti, sigurnosti i zadovoljstva. [10]

6.2. ISO/IEC 25000 – SquaRE

ISO/IEC 25000 standard je poznatiji pod imenom SQuaRE. Nastao je kao rezultat razvoja drugih sličnih standarda: ISO/IEC 9126 (kvaliteta programskih proizvoda) i ISO/IEC 14598 (procjena programskih proizvoda). Navedeni standard sastoji se od nekoliko pod standarda koji su usmjereni ka kvaliteti programskih proizvoda. Točnije on se bavi specifikacijama, mjerenjem i vrednovanjem zahtjeva za kvalitetu softverskih proizvoda, te je odvojen i različit od upravljanja kvalitetom procesa definiranih u ISO 9000 obitelji standarda. [11]

Sastoji se od pet glavnih pod standarda:

1. ISO/IEC 2500n – odjel upravljanja kvalitetom (eng. quality management division),
2. ISO/IEC 2501n – odjel za model kvalitete (eng. quality model division),
3. ISO/IEC 2502n – odjel za mjerenje kvalitete (eng. quality measurement division),
4. ISO/IEC 2503n – odjel za zahtjeve kvalitete (eng. quality requirements division),
5. ISO/IEC 2504n – odjel za ocjenu kvalitete (eng. quality evaluation division) [11]

6.3. ISO/IEC 29119

Razvoj ISO/IEC/IEEE 29119 standarda je započet 2007. godine, a njegovi dijelovi vezani za koncepte i definicije, procese testiranja i testnu dokumentaciju postaju službeni međunarodni standardi 2013. godine, dok dio vezan za tehnike testiranja postaje službeni 2015. godine. [12]

Svrha ISO/IEC/IEEE 29119 serije standarda je definiranje međunarodno priznatog standarda za testiranje programskog proizvoda kojeg može koristiti bilo koja organizacija prilikom obavljanju bilo kojeg oblika testiranja programskog proizvoda. [13] Navedena serija standarda sastoji se od pet dijelova od kojih svaki ima svoju jedinstvenu svrhu.

Dijelovi:

1. ISO/IEC/IEEE 29119-1 - Koncepti i definicije (eng. concepts and definitions)
2. ISO/IEC/IEEE 29119-2 - Testni procesi (eng. test processes)
3. ISO/IEC/IEEE 29119-3 - Testna dokumentacija (eng. test documentation)
4. ISO/IEC/IEEE 29119-4 - Tehnike testiranja (eng. test techniques)
5. ISO/IEC/IEEE 29119-5 - Testiranje temeljem ključnih riječi (eng. keyword-driven testing)

6.3.1. ISO/IEC/IEEE 29119-1

Prvi dio iz serije standarda naziva se ISO/IEC/IEEE 29119-1 i sadržava koncepte i definicije najvažnijih pojmova. Navedeni dio olakšava upotrebu drugih dijelova standarda ISO/IEC/IEEE 29119 na način da objašnjava najvažnije koncepte i definicije na kojima se drugi dijelovi grade. Uz teorijsku podlogu navedeni dio sadržava i primjere iz prakse, te tako pruža lakše vladanje standardima. Bez ovog dijela bilo bi teško savladati koncepte koji se koriste u drugim dijelovima. [9]

6.3.2. ISO/IEC/IEEE 29119-2

ISO/IEC/IEEE 29119-2 drugi je dio iz serije standarda i odnosi se na testne procese. On sadržava opise procesa testiranja programskih proizvoda na organizacijskoj razini, razini upravljanja testovima i razini dinamičkog testiranja. Testni procesi se odnose na dinamička testiranja, funkcionalna i nefunkcionalna testiranja, ručna i automatizirana testiranja, te na skriptna i neskriptna testiranja. Procesi definirani u navedenom dijelu mogu se koristiti u bilo kojem modelu razvoja programskih proizvoda. [9]

6.3.3. ISO/IEC/IEEE 29119-3

ISO/IEC/IEEE 29119-3 dio odnosi se na testnu dokumentaciju, točnije na predloške i primjere testne dokumentacije. Navedenim predlošcima testne dokumentacije pokriveni su svi dijelovi životnog ciklusa razvoja programskog proizvoda. Za ovaj dio je specifično to kako se testna dokumentacija može prilagoditi organizaciji koja ju slijedi. [14] Testna dokumentacija je podijeljena na tri tipa:

1. Organizacijska testna dokumentacija (pravila testiranja, testne strategije)
2. Projektna testna dokumentacija (plan testiranja projekta, testna projektna izvješća)

3. Dokumentacija o testnim razinama (testni plan, testna specifikacija, testni rezultati, izvješća o anomalijama, izvješća o statusima testnih razina, izvješća o testnoj okolini, izvješća o završetku testne razine) [9]

6.3.4. ISO/IEC/IEEE 29119-4

ISO/IEC/IEEE 29119-4 je četvrti dio iz serije standarda ISO/IEC/IEEE 29119, te on definira tehnike testiranja koje se mogu koristiti tijekom dizajna i provedbe procesa testiranja definiranih u ISO/IEC/IEEE 29119-2. Namijenjen je testerima, testnim upraviteljima, programerima i svim onima koji su zaduženi za upravljanje i provedbu procesa testiranja. [15] Tehnike opisane u ovom dijelu standarda mogu biti učinkovit način za razvoj testnih slučajeva. [14] Testni slučajevi se koristite za izvođenje dokaza da su svi zahtjevi navedeni u testu uspješno ispunjeni ili postoje određeni nedostatci koje je potrebno ispraviti.

6.3.5. ISO/IEC/IEEE 29119-5

Zadnji dio iz serije standarda ISO/IEC/IEEE 29119 definira konzistentna rješenja za testiranje pomoću ključnih riječi. Testiranje pomoću ključnih riječi je tehnika koja uključuje opisivanje testnih slučajeva na temelju unaprijed definiranog skupa ključnih riječi. Ključne riječi temelje se na nizu radnji koje je potrebno izvršiti pomoću koraka opisanih u testnom slučaju. [14] Navedena tehnika testiranja je lako razumljiva jer su ključne riječi pisane na prirodnom jeziku.

7. Tipovi testiranja

U ovom poglavlju opisat će se dva osnovna tipa testiranja koji se pojavljuju u životnom ciklusu testiranja programskih proizvoda.

7.1. Ručno testiranje

Prvi i najstariji tip testiranja je ručno testiranje. Ručno testiranje se kao što ime kaže obavlja ručno bez korištenja automatiziranih alata ili skripata. Kod ručnog testiranja testeri su u ulozi krajnjih korisnika, te ispituju programski proizvod na način kako bi ga krajnji korisnik koristio. Testeri ručnim testiranjem pokušavaju pronaći neočekivana ponašanja i greške u radu programa.

Ručno testiranje se dalje može podijeliti na faze. Pa tako imamo jedinično testiranje, integracijsko testiranje, testiranje sustava i testiranje prihvatljivosti. Svaku od tih faza testeri ispituju na način kako bi krajnji korisnik to odradio. Navedena činjenica dovodi do toga da testeri potrebni za ručno testiranje ne trebaju imati znanja pisanja i čitanja koda. Kod ručnog testiranja testeri se koriste testnim planovima, testnim slučajevima odnosno testnim scenarijima. [16]

Ručno testiranje je iznimno korisno kod postavljanja dobre podloge za automatizirana testiranja odnosno automatizirane testove. Ručnim testiranjem i ljudskom radoznalošću moguće je otkriti greške koje automatizirani testovi ne bi mogli. Dok su automatizirani testovi unaprijed napisani i točno definirani, ručni testovi zajedno sa ljudskom radoznalošću mogu zaviriti u određene sfere programskog proizvoda koje nisu predviđene. Uz to što je rezultat ručnog testiranja nepredvidiv ručno testiranje također ima prednost obavljanja testova od strane bilo koje osobe unutar poduzeća.

7.1.1. Zašto koristiti ručno testiranje?

Oracle navodi sljedeće razloge zašto koristiti ručno testiranje:

- Vrijeme – testni timovi možda neće imati vremena za naučiti testne alate, pisati skripte ili pronaći alternative ručnim testovima
- Složenost aplikacija – ako je složenost programskog proizvoda velika tada vjerojatno nije prikladno koristiti automatizirane testove
- Skup vještina – određeni testeri nemaju potrebni skup vještina kako bi mogli koristiti alate za automatizirano testiranje
- Trošak – skupoća početnog ulaganja u nabavu alata za automatizaciju testova

- Sigurnost – pojedini testeri i upravitelji kvalitetom nemaju povjerenja u alate za automatizirano testiranje
- Svijest – uvijek postoje alternative automatiziranom testiranju [17]

7.1.2. Kada je ručno testiranje bolje od automatiziranog?

U nekim situacijama je puno bolje koristiti ručno testiranje naspram automatiziranog testiranja. Prva stvar koja se nameće kao veliki razlog zbog kojeg je bolje koristiti ručno testiranje je subjektivnost. Za pojedine funkcionalnosti potrebna je ljudska subjektivnost. Naravno da programi nemaju tu subjektivnost i ne mogu to odraditi umjesto ljudi. Na primjer ljudska subjektivnost je glavni kriterij kod provjere upotrebljivosti i osjećaja korištenja programskog proizvoda. [17]

Uz ljudsku subjektivnost razlog za korištenje ručnog testiranja naspram automatiziranom je kompleksnost programskog proizvoda. Ukoliko je programski proizvod kompleksan i iznimno važan za poduzeće koje ga razvija, dobro je promisliti zašto bi koristili automatizirane testove. Možda je bolje iskoristiti testere i njihovo znanje za detaljno testiranje navedenog programskog proizvoda.

I zadnji razlog kojeg je potrebno istaknuti su promjene u novim funkcionalnostima. U fazi razvoja novih funkcionalnosti nije pametno koristiti automatizirane testove koji su striktni i ispituju točno određeni skup slučajeva po određenim uvjetima. Kako su nove funkcionalnosti još u razvoju tako i slučajevi variraju i nije potrebno gubiti vrijeme na pisanje skripti.

7.2. Automatizirano testiranje

Drugi tip testiranja je automatizirano testiranje. Automatizirano testiranje je testiranje u kojem testeri pišu testne skripte i koriste različite alate i programe za testiranje programskih proizvoda. Svrha i cilj automatiziranog testiranja je obaviti testove što brže i jednostavnije pri tome pokriti što je više moguće slučajeva i iskoristiti testove nekoliko puta. [16]

Automatizirano testiranje se ne može koristiti u svim prilikama. Ono se najčešće koristi za provjeru grafičkog korisničkog sučelja, validaciju polja poput unosa podataka za prijavu, vezu s bazom podataka, itd. Svaki od tih testova bi se mogao izvršiti ručno no taj način bi oduzeo previše vremena i novca. Važno je istaknuti kako automatizirano testiranje u kratkom vremenskom periodu košta znatno više od manualnog odnosno ručnog testiranja. Ukoliko automatizirano testiranje gledamo kroz duži period vremena tada možemo doći do zaključka kako automatizirano testiranje košta manje. Razlog toj varijaciji je to što početno ulaganje kod automatiziranog testiranja košta nešto više od početnog ulaganja kod ručnog testiranja. Prvenstveno se to odnosi na nabavu programa i alata koji se koriste za provedbu

automatiziranih testova. U trošak također ulazi i obrazovanje ljudi za korištenje programa i alata. Jer bez znanja navedeni programi i alati nemaju primjenu tj. ne donose nikakvu vrijednost poduzeću. [16]

7.2.1. Kada je automatizirano testiranje bolje od ručnog?

U određenim situacijama je bolje koristiti automatizirano testiranje naspram ručnog testiranja. Prva takva situacija je korištenje automatiziranog testiranja prilikom provođenje regresijskih testova. Regresijski testovi se provode onda kada su dodane nove funkcionalnosti na postojećem programskom proizvodu. Te se njima provjerava da li je došlo do grešaka u postojećim ne diranim funkcionalnostima. [17]

Također automatizirano testiranje je vrlo korisno prilikom statičkih i ponavljajućih testova. Zadaci su u takvim testovima ponavljajući, nepromjenjivi i monotoni, te su idealni kandidat za automatizaciju. [17]

Nadalje automatizirano testiranje je korisno kod testiranja pomoću podataka. Testiranjem pomoću podataka se provjerava funkcionalnost s mnogo različitih ulaza i velikim skupom podataka. Primjer su prijave, pretraživanja, registracije i slično. [17]

I zadnji razlog kada je bolje koristiti automatizirano testiranje naspram ručnog su testiranja opterećenja i performansi programskog proizvoda. [17]

7.2.2. Alati za automatizirano testiranje

Postoje mnogobrojni alati za provođenje automatiziranog testiranja. Ne postoji univerzalni alat koji bi pokrio sve tipove programa, platforme i skriptne jezike. Prema tome svaki alat ima prednosti i mane, te je potrebno pomno izabrati onaj alat koji odgovara programskom proizvodu kojeg razvijamo i okruženju u kojem se navedeni proizvod razvija.

7.2.2.1. Selenium

Prvi alat za automatizirano testiranje koji je potrebno spomenuti je Selenium. Selenium je alat otvorenog koda koji se koristi za testiranje web aplikacija putem preglednika na različitim platformama. Besplatan je i podržan je od velikog broja internetskih preglednika. [18] Svrstava se u grupu alata za funkcionalno testiranje. Sastoji se od četiri komponente:

1. Selenium IDE - koristi se kao alat za izradu testnih skripti i nije potreban programski jezik kako bi izradili testne skripte
2. Selenium Remote Control - omogućuje korisnicima korištenje programskog jezika
3. Selenium Web Driver – omogućuje stabilnu komunikaciju između testnih skripti i web preglednika

4. Selenium Grid - pomaže u provedbi paralelnih testova na različitim preglednicima pomoću Selenium Remote Control-a [19]

7.2.2.2. SoapUI

Drugi alat je SoapUI. Navedeni alat na svojoj web stranici kaže kako je SoapUI najrašireniji alat za funkcionalno testiranje aplikacijskog programskog sučelja (eng. application program interface – API) na svijetu. [20] SoapUI omogućuje funkcionalno testiranje SOAP (eng. simple object access protocol) web servisa i REST (eng. representational state transfer) aplikacijskog programskog sučelja, pokrivenost jezika opisa web servisa (eng. web services description language) i izmjenu testova. [19]

7.2.2.3. Katalon Studio

Treći alat se naziva Katalon Studio, namijenjen je automatiziranom testiranju web aplikacija, mobilnih aplikacija i aplikacijskog programskog sučelja. Jednostavan je za korištenje i besplatan. Uz alat dolazi i analitički alat koji omogućuje bolji pogled rezultata provedenih testova i organizaciju budućih testova. [21]

7.2.2.4. Apache JMeter

Kako ne bi spomenuli samo alate za funkcionalno testiranje sljedeći alat se koristi kao alat za testiranje mjerenje performansi. Naziva se Apache JMeter i u potpunosti je napisan u Java programskom jeziku. Koristi se za testove mjerenja performansi i testove opterećenja. Prva namjena mu je bila testiranje web aplikacija, no danas se njegova namjena proširila i na ostale testne funkcije poput SOAP i REST web servisa, java objekte, razne protokole i slično. [22]

8. Proces testiranja

U svijetu testiranja ne postoji jedinstveni proces testiranja koji vrijedi za sve projekte razvoja programskih proizvoda. Proces testiranja i njegovi pod procesi variraju od projekta do projekta. Iako procesi testiranja variraju u ovom poglavlju će biti definiran i detaljnije opisan proces koji ISTQB navodi kao temeljni proces testiranja. Naime može se definirati temeljni proces na koji se naknadno od projekta do projekta dodaju pod procesi i aktivnosti. Na taj se način dobiva jedinstveni proces testiranja za pojedini projekt.

8.1. Planiranje i kontrola

Prvi pod proces procesa testiranja programskog proizvoda odnosi se na planiranje i kontrolu. Kao i svaki drugi projekt tako i projekt testiranja sadrži proces planiranja kao prvi korak realizacije tog projekta. Iz iskustva se može zaključiti kako je pametno krenuti u realizaciju nečega sa detaljnim opisom aktivnosti, resursa i zaduženja između članova. Isto to vrijedi i za proces testiranja. Prvo je potrebno donijeti detaljni plan testiranja koji će pomoći u kontroli testiranja. Kao i kod drugih planova, plan testiranja je potrebno tijekom testiranja kontrolirati, ažurirati i ispravljati. [23]

8.1.1. Testni plan

U planu testiranja je potrebno definirati misiju i ciljeve koji se nastoje dostići testiranjem. Misija mora govoriti o tome zašto izvršavamo navedene testove, a ciljevi moraju usmjeravati testere u pravcu koji je zacrtan testiranjem. Uz misiju i ciljeve testni plan mora sadržavati resurse koji će biti korišteni tijekom testiranja. Resursi testiranja odnose se na alate, ljude, i druge potrebne stvari bez kojih ne bi bili u mogućnosti izvesti testiranje. Uz resurse potrebno je definirati strategiju testiranja. Strategija objašnjava kako i na koji način će se obaviti testiranje. Na koje aktivnosti će se pripaziti tijekom testiranja, na što će biti skrenuta posebna pozornost, koji resursi će se koristiti na pojedinim fazama i slično. Nakon što se definiraju resursi i strategija potrebno je definirati vrijeme i budžet procesa testiranja. Vrijeme je stavka koju svakako ne treba zaboraviti i ispustiti iz testnog plana. Vremensko razdoblje izvedbe može znatno utjecati na krajnje rezultate testova. Pre kratko vremensko razdoblje tjera testere na brže otkrivanje grešaka što može uzrokovati manjim brojem otkrivenih grešaka. Dok dugo vremensko razdoblje može uzrokovati opuštanje i pomanjkanje volje za traženje grešaka, te u konačnici također rezultirati manjim brojem otkrivenih grešaka. Ne postoji točno definirano vremensko razdoblje potrebno za odvijanje procesa testiranja, ali je vrlo važno pomno isplanirati navedeni dio. Uz vrijeme se veže novac, tako da je budžet sljedeća stavka koja se

mora uzeti u obzir tijekom planiranja testiranja. Uz navedene stavke, pod proces planiranja i kontrole može sadržavati još niz stavki koje treba isplanirati kako bi se krenulo sa analizom i dizajnom. [23]

8.2. Analiza i dizajn

U pod procesu analize pristupa se analiziranju, preispitivanju planiranih radnji. Na taj se način donosi osnova za testiranje koja se u pod procesu dizajna detaljnije razrađuje i gradi. Iz pod procesa analize i dizajna izlaze testni slučajevi koji se kasnije implementiraju i izvršavaju. Kako u daljnje pod procese ne bi ulazili nadrealni testni slučajevi potrebno je detaljno preispitati što bi se trebalo testirati i utvrditi da li je to izvedivo odnosno testabilno. Uz provjeru testabilnosti potrebno je u navedenom pod procesu odrediti rizik. Strategiju testiranja koja se donijela u planu testiranja potrebno je preispitati i u nju uračunati rizik koji se može dogoditi. Strategija definira tehnike testiranja koje će se izvršavati te je s toga potrebno preispitati svaku tehniku koja će biti provedena i vidjeti prednosti i nedostatke koje ona donosi. [23]

U pod procesima analize i dizajna moraju se definirati logični i konkretni testni slučajevi kako ne bi došlo do problema u daljnjem procesu testiranja. Testni slučajevi se moraju razraditi do najsitnijih detalja.

8.2.1. Testni slučajevi

„Skup ulaznih vrijednosti, preduvjeta izvršenja, očekivanih rezultata i uvjeta izvršenja razvijenih za određeni cilj ili testni uvjet, kao što je vježbanje određenog programskog puta ili provjera sukladnosti s određenim zahtjevom.“ [23]

Testni slučajevi nisu predefinirani te se mijenjaju s obzirom na projekt za koji se koriste. Ali mogu se nabrojiti neke osnovne komponente koje svaki testni slučaj mora imati, a to su:

- Identifikacijska oznaka testnog slučaja
- Verzija programskog proizvoda
- Datum testiranja
- Dio programskog proizvoda koji se testira
- Svrha
- Pretpostavke
- Preduvjeti
- Koraci provedbe testa
- Očekivani rezultat
- Stvarni rezultat [24]

8.3. Implementacija i izvršenje

Nakon što se završi sa pod procesima analize i dizajna čiji rezultat moraju biti testni slučajevi, potrebno je pristupiti izvršenju testnih slučajeva odnosno izvršavanju testova. Prvo se kreće sa implementacijom testnih slučajeva koji u ovoj fazi prelaze iz logičkih u stvarne testne slučajeve. Na taj način se uzimaju sve prethodno dogovorene stvari u obzir. Nakon implementacije kreće se sa izvršavanjem testova na definiranoj okolini i uz pomoć resursa koji su definirana tijekom faze planiranja. Vrlo važno je pratiti smjernice i strategiju koja je također donesena u fazi planiranja kako ne bi došlo do odstupanja. Često se u praksi događa kako testeri prilikom implementacije i izvršenja testova odlutaju od dogovorenih smjernica, te u konačnici ne postignu dogovorene ciljeve. [23]

Potrebno je pratiti razvoj implementacije i izvršenja kako se navedene stvari ne bi događale. Uz to je potrebno bilježiti rezultate svake provedene aktivnosti, te na taj način ostavljati što je više moguće traga. Tragovi su iznimno korisni u sljedećem pod procesu kada dolazi evaluacija rezultata. Također je u ovim pod procesima vrlo važno ne zaboraviti glavni cilj svakog testiranja, a to je otkrivanje grešaka. Potrebno je otkriti što je veći broj grešaka.

8.4. Evaluacija i kreiranje izvještaja

Nakon što odradimo testove slijedi evaluacija dobivenih rezultata. Potrebno je vrlo pažljivo proći sve tragove koje smo bilježili u prethodnim pod procesima implementacije i izvršenja. Svaki trag i rezultat je potrebno usporediti i ocijeniti sa predviđenim kriterijima koje smo si zadali tijekom planiranja. Konačnim ocjenama možemo odrediti da li je provedeno testiranje uspješno ili neuspješno odrađeno. Da li je potrebno odraditi dodatne testove ili možemo prijeći na kreiranje izvještaja. [23]

Ako testiranje ocijenimo uspješnim krećemo sa kreiranjem izvještaja koji prikazuju koje rezultate smo dobili, koliko grešaka smo uočili, di se nalaze tj. u kojim testovima smo ih našli, koji utjecaj imaju na daljnji razvoj i slično. Također se u ovoj fazi uzimaju vrijeme i novac u razmatranje te se na temelju te dvije stvari mogu definirati grafovi koji će prikazati koliko nas je sve to koštalo.

8.5. Zatvaranje

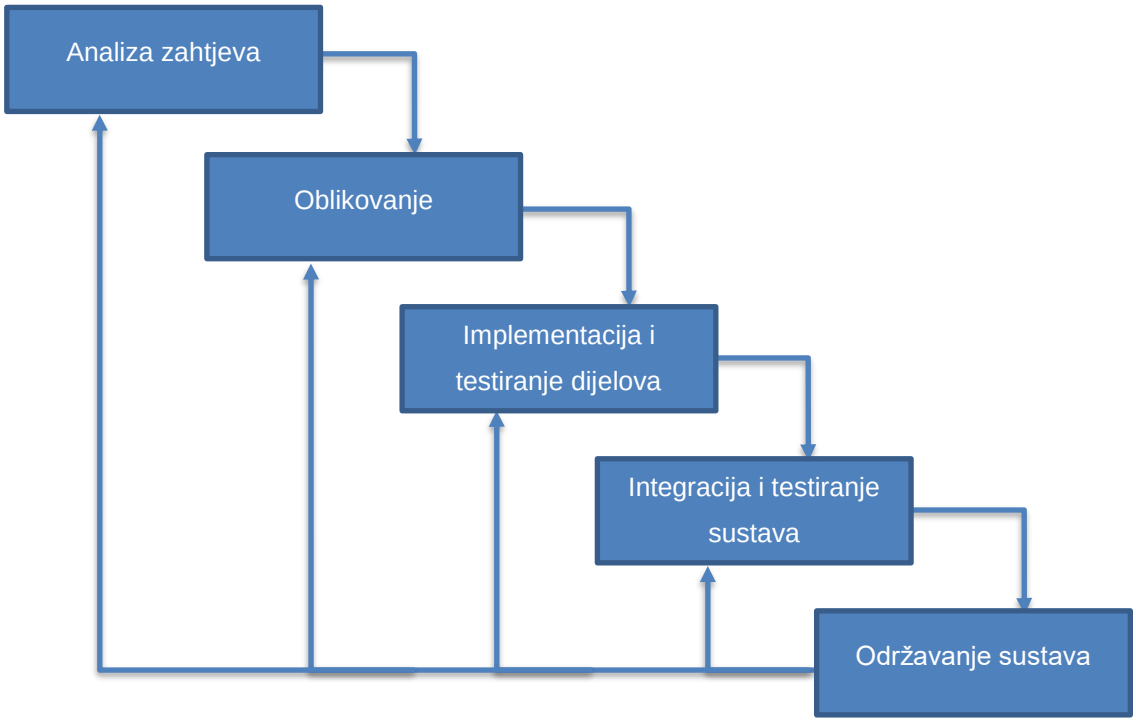
Pod proces zatvaranja sadrži skup aktivnosti koje se odnose na konačno zatvaranje testiranja. Nakon tih aktivnosti testiranje je gotovo. Ova faza se može nazvati i dokumentiranje jer se u njoj odvija proces arhiviranja testova i dokumentiranja svih njihovih stavki. Potrebno je sve rezultate spremiti i iz njih izvući što je više pouka za buduća testiranja kao i za budući razvoj programskih proizvoda. [23]

9. Proces testiranja s obzirom na metodologiju razvoja

Proces testiranja programskih proizvoda ovisi o metodologijama razvoja programskih proizvoda. Naime svaka od metodologija razvoja izvršava proces testiranja na drugačiji način. Bilo da proces testiranja uključuje nakon svake faze razvoja proizvoda ili da testiranje radi na kraju sveukupnog razvoja ili kombinirano. Postoji puno metodologija razvoja programskih proizvoda, a u ovom poglavlju će biti obuhvaćene tri metodologije: vodopadni model, V-model i agilni/iterativni model.

9.1. Vodopadni model

Vodopadni model razvoja karakteriziraju odvojene i specifične faze specifikacije i razvoja. Svaka od faza vodopadnog modela mora se dovršiti prije prelaska na sljedeću fazu. Navedena situacija je dosta zamršena jer ne postoji povratna veza između susjednih faza. Vodopadni model se smatra starijim modelom razvoja programskih proizvoda. Faze koje obuhvaća navedeni model su: analiza zahtjeva, razvoj i oblikovanje sustava i programske potpore, implementacija i testiranje modula, integracija i testiranje sustava i na kraju uporaba sustava i njegovo održavanje. [25] [26]



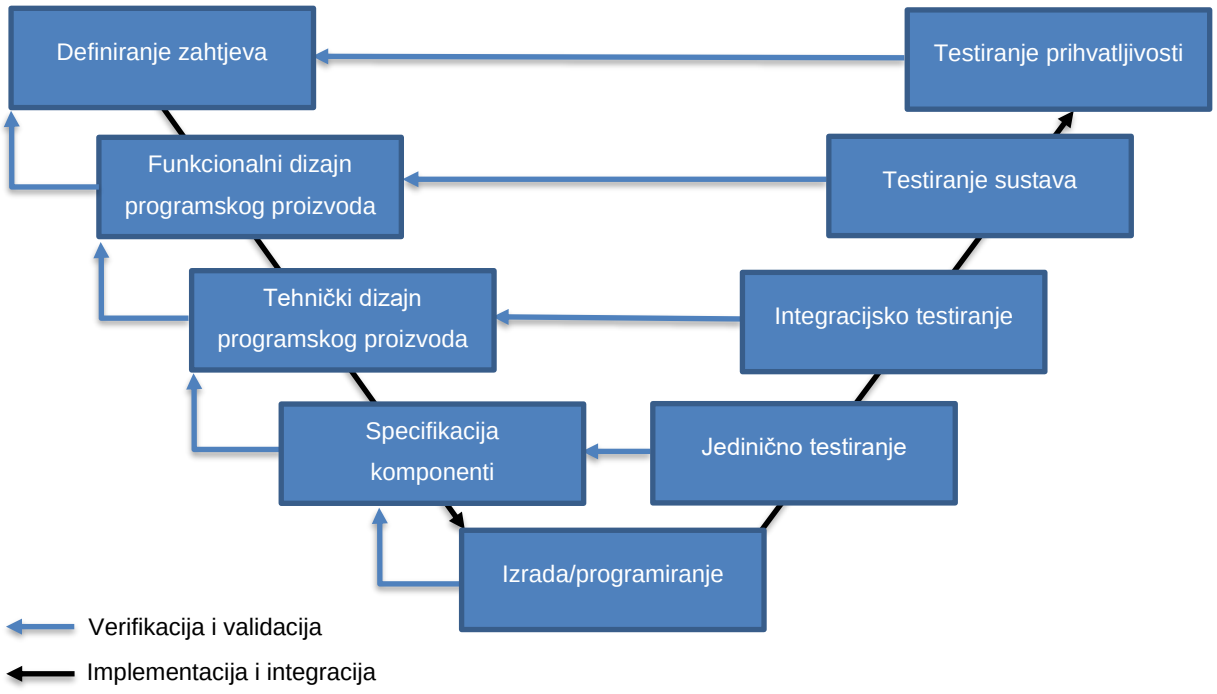
Slika 1 Vodopadni model razvoja programskih proizvoda [23]

Na prethodnoj je slici moguće vidjeti vodopadni model i faze od kojih je on sastavljen. U vodopadnom modelu testiranje je uključeno nakon faze implementacije. Naime kada je završena faza implementacija određenog dijela slijedi testiranje navedenog dijela. Na taj isti način počiva i testiranje sustava u cjelini. Nakon što se cijeli sustav odnosno programski proizvod integrira tada slijedi faza testiranja tog sustava. Nedostaci koji se javljaju kod vodopadnog modela odnose se na korisničke zahtjeve koji moraju biti jako detaljno definirani i eventualne promjene moraju biti svedene na minimum jer u protivnom može doći do problema u razvoju. Tako je i kod testiranja u ovoj metodici razvoja. Ako otkrijemo preveliki broj grešaka dolazi do narušavanja redoslijeda izvođenja i dolazi do problema. Testiranje se izvodi jednokratno na kraju implementacije te tako smanjuje mogućnost otkrivanja većeg broja grešaka.

9.2. V-model

Kako se u standardnom vodopadnom modelu testiranje izvodi samo prije produkcije programskog proizvoda odnosno na kraju razvoja, pojavila se potreba za modifikacijom standardnog modela. Modifikacija koja je nastala naziva se V-model i ona podržava testiranje nakon svake faze životnog ciklusa razvoja programskog proizvoda. Glavna ideja V-modela je izjednačavanje važnosti zadataka razvoja i testiranja. [23]

V-model ima nekoliko verzija. Od literature do literature varira broj razina koje obuhvaća, ali sve verzije imaju istu osnovu. Osnova koja definira V-model kaže kako lijeva grana definira razvojni proces dok desna grana definira procese integracije i testiranja. [23]



Slika 2 Osnovni V-model [25]

Prva faza razvoja odnosi se na prikupljanje, specificiranje i analizu korisničkih zahtjeva. Za provjeru zadovoljava li programski proizvod zahtjeve korisnika provode su testovi prihvatljivosti. Analizom njihovih rezultata utvrđuje se u kojoj mjeri su zahtjevi ispunjeni.

Druga faza je razvoja odnosi se na funkcionalni dizajn programskog proizvoda, točnije dizajniraju se funkcionalnosti programskog proizvoda. Za navedenu fazu provodi se testiranje programskog proizvoda koje provjerava zadovoljava li sustav kao cjelina definirane funkcionalne zahtjeve.

Treća faza odnosi se na tehnički dizajn programskog proizvoda. Uključuje osmišljavanje implementacije programskog proizvoda do najsitnijih detalja. Razrađuje se arhitektura programskog proizvoda. Za njenu provjeru koriste se integracijski testovi koji provjeravaju da li grupe komponenata odnosno jedinica komuniciraju na način koji je definiran u ovoj fazi.

Četvrta faza odnosi se na specifikaciju komponenti (najmanjih jedinica od kojih je sagrađen programski proizvod). U ovoj fazi definiraju se podsustavi, njihovi zadaci, ponašanje, struktura i sučelja prema drugim podsustavima. [23] Za provjeru ove faze koriste se jedinični testovi.

I zadnja faza je programiranje odnosno izrada programskog proizvoda. Uključuje kodiranje, izradu programskog proizvoda prema specifikacijama iz prethodnih faza.

9.3. Agilni/iterativni model

Zadnjih godina agilni model razvoja programskih proizvoda postaje sve popularniji. Postaje sve popularniji iz razloga što omogućava brz odziv na korisničke zahtjeve. Kako je vodopadni model dosta striktan oko novih korisničkih zahtjeva razvio se novi model koji omogućuje iterativni razvoj programa uz stalne promjene koda. Navedena metodologija se više koristi kod manjih i srednjih projekata kod kojih postoji stalni kontakt sa klijentima koji dodaju nove zahtjeve. [3, p. 175]

Agilni model se zapravo sastoji od grupe modela koji počivaju na jednakim načelima razvoja u inkrementima. Neki od modela su:

- ekstremno programiranje,
- SCRUM,
- čist razvoj programske potpore,
- razvoj zasnovan na ispitivanju. [3, p. 177]

Proces testiranja kod navedenih modela je uključen u svaku od provedenih iteracija. Na taj se način isporučuju funkcionalnosti proizvoda koji su ispitane i spremne za korisničku upotrebu. Ne događa se situacija u kojoj prvo napravite gotov proizvod pa tek onda idete u testiranje tog proizvoda. U ovim metodama razvoja iteracije pomažu u otkrivanju većeg broja grešaka na način da sagledavaju manje inkremente proizvoda. Navedena značajka dosta

pridonosi korištenju navedenih metoda u stvarnom svijetu u kojem poduzeća koja kreiraju programske proizvode nastoje napraviti proizvode sa što manje grešaka.

Uz to je važno spomenuti kako stalna komunikacija sa klijentima pridonosi manjem broju grešaka. U tom slučaju ste u mogućnosti bolje ispitati želje i zahtjeve klijenata i na taj način izraditi bolji proizvod.

10. Testne metode

Testiranje programskih proizvoda sadrži tri metode izvođenja testiranja. Metode su formirane oko sadržaja koji je vidljiv/nevidljiv testerima prilikom izvođenja testiranja, te dijelovima koji se provjeravaju. Prva metoda koja je objašnjena u ovom poglavlju je metoda crne kutije. Metodu crne kutije karakterizira tester koji nema pristup unutarnjoj strukturi programskog proizvoda i provjera ulaza i izlaza testnog objekta. Metoda koja je suprotna metodi crne kutije je metoda bijele kutije. U navedenoj metodi tester ima pristup unutarnjoj strukturi programskog proizvod, te provjerava logiku i strukturu koda. I zadnja metoda je metoda sive kutije koja je nastala miješanjem prve dvije.

10.1. Metoda crne kutije

Prilikom testiranja metodom crne kutije unutarnja struktura programskog proizvoda je nepoznata. Testeri za definirane inpute provjeravaju da li će dobiti očekivane outpute. Prilikom izvođenja testiranja nisu upoznati sa unutarnjom logikom i radnjama koje se odvijaju nad inputom, već su isključivo fokusirani na output. Metoda crne kutije orijentirana je na specifikacije odnosno definirane zahtjeve proizvoda. Promatra se zadani input i dobiveni output te njihova usklađenost sa specifikacijama, zahtjevima definiranim na početku procesa testiranja. [23]

Prilikom testiranja metodom crne kutije važno je definirati razumni skup testnih slučajeva. Pod razumnim skupom se misli skup ulaznih podataka koji reprezentira gotovo sve moguće inpute. U praksi je ne moguće ispitati svaki mogući input jer bi navedeni testovi trajali pre dugo, već se pokušavaju razumno definirati skupovi podataka i samo određene podatke iz tih skupova isprobati kao input. Na taj se način smanjuje broj inputa, a s druge strane pokrivaju se svi skupovi podataka koji se mogu unijeti kao input.

Kako bi odredili navedene skupove podataka koristimo se sljedećim tehnikama:

1. Particioniranje klasa ekvivalencije (eng. equivalence class partitioning)
2. Analiza graničnih vrijednosti (eng. boundary value analysis)
3. Testiranje tranzicijskog stanja (eng. state transition testing)
4. Tehnike temeljene na logici (eng. logic-based techniques)
5. Testiranje temeljeno na slučajevima upotrebe (eng. use-case based testing) [23]

Nabrojane tehnike biti će opisane u poglavlju testne tehnike.

10.1.1. Prednosti

Metoda crne kutije ima mnogobrojne prednosti:

- Pristup kodu i unutarnjoj strukturi nije potreban
- Vrlo lako odrediti da li se greška dogodila (usporedba dobivenog outputa sa specifikacijama/zahtjevima)
- Laka izvedba testova
- Nije potrebno znanje programiranja tj. razumijevanja koda
- Dobro strukturiran i pogodan za testiranje velikih segmenata programskih proizvoda [3]

10.1.2. Nedostaci

Također sadrži i nedostatke:

- Nedovoljno znanje testera o samom programskog proizvodu koji se testira
- Pogađanje grešaka u dosta slučajeva, jer tester ne vidi i ne poznaje unutarnju strukturu i ne može ciljano testirati dijelove sklonije pogreškama
- Velika količina testnih slučajeva [3]

10.2. Metoda bijele kutije

Testiranje metodom bijele kutije se za razliku od testiranja metodom crne kutije zadire u unutarnju strukturu programskog proizvoda. Testeri su u mogućnosti sagledati i testirati programski proizvod vodeći se unutarnjom strukturom proizvoda. Kako bi tester testirao metodom bijele kutije on mora poznavati programiranje. Na taj se način omogućuje korištenje metode bijele kutije. Upravo gledanjem, promatranjem strukture koda tester pronalazi i prijavljuje greške. Iako se testiranje ne bavi izmjenom koda u nekim slučajevima i tehnikama metode bijele kutije kod se može mijenjati i tako lakše pronaći greške u radu programa.

Temelj testiranja metodom bijele kutije počiva na izvršenju svakog dijela koda testnog objekta barem jednom. Testni slučajevi se definiraju promatrajući protok podataka, slijed izvršavanja koda i analizirajući logiku programskog proizvoda. Nakon što se testni slučajevi definiraju kreće se u izvršavanje. Iako bi se možda očekivalo kako se rezultati u ovoj metodi vrednuju preko koda, istina je drugačija. Rezultati se vrednuju preko početno definiranih zahtjeva i specifikacija. [23]

Tehnike metode bijele kutije su sljedeće:

1. Testiranje iskaza (eng. statement testing)
2. Testiranje odluka/grana (eng. decision/branch testing)
3. Test uvjeta (eng. test of conditions)
4. Testiranje putanja (eng. path testing) [23]

10.2.1. Prednosti

Prednosti testiranja metodom bijele kutije su:

- Poznavanje koda od strane testera
- Lakše definiranje dijelova koda koji će se testirati (testiranje rizičnijih dijelova koda)
- Fokusiranje na dijelove koda koji su skloniji greškama
- Otkrivanje grešaka u ranijim fazama razvoja
- Lakše definiranje testnih podataka [3]

10.2.2. Nedostaci

Nedostaci testiranja bijelom kutijom su:

- Troškovi testiranja su veći
- Testeri koji su dobro upoznati sa unutarnjom strukturom programskog proizvoda
- Nemogućnost ispitivanja svih dijelova koda [3]

10.3. Metoda sive kutije

Metoda sive kutije je metoda koja je nastala mješavinom metoda bijele i crne kutije. Kod testiranja metodom sive kutije tester poznaje dio strukture koda i unutarnje logike programskog proizvoda, ali ne pozna cijelu strukturu niti logiku. Testiranje metodom bijele kutije traži od testera da poznaje cjelokupnu strukturu koda, te da je u mogućnosti ispitati sve dijelove koda. Dok testiranje metodom crne kutije testeri ispituju vanjštinu tj. inpute i outpute, dok u isto vrijeme nisu upoznati sa unutarnjom strukturom programskog proizvoda.

Glavno obilježje testiranja metodom sive kutije je ograničeno znanje o unutarnjem funkcioniranju programskog proizvoda. Metoda sive kutije naziva se još i prozirno testiranje. Razlog tomu je djelomično poznavanje strukture. Najčešće se koristi za testiranje web aplikacija. Tijekom testiranja metodom sive kutije testeri imaju pristup dokumentima dizajna i bazi podataka. Na taj način testeri bolje pripremaju testne podatke i testne scenarije. Kod testiranja metodom sive kutije testeri su ljudi različitih znanja i vještina. Mogu biti krajnji korisnici, programeri, upravitelji kvalitetom i slično. [27]

Tehnike metode sive kutije su:

1. Matrično testiranje (eng. matrix testing)
2. Regresijsko testiranje (eng. regression testing)
3. Testiranje uzoraka (eng. pattern testing)
4. Testiranje ortogonalnog niza (eng. orthogonal array testing) [27]

10.3.1. Prednosti

Prednosti metode sive kutije su:

- Iskorištavanje prednosti metoda crne i bijele kutije
- Ne oslanjanje na kod, već na definiciju sučelja i funkcionalne zahtjeve
- Test se izvršava sa gledišta korisnika, a ne dizajnera, programera [27]

10.3.2. Nedostaci

Nedostaci metode sive kutije su:

- Ne poznavanje cijele strukture i logike programskog proizvoda
- Ne pokrivanje svih mogućih scenarija [27]

11. Tehnike testiranja pomoću metoda crne kutije

Testiranje metodom crne kutije je testiranje uz pomoću specifikacija/zahtjeva programskog proizvoda. Naziva se još i testiranje vođeno ulazima i izlazima. Cilj navedene metode je pronaći područja programskog proizvoda u kojima se program ne ponaša prema definiranim zahtjevima. [3] U poglavlju 10.1. je detaljnije objašnjeno testiranje metodom crne kutije. U ovom poglavlju opisuju se tehnike koje se koriste za određivanje podataka koji će ući u testiranje i određivanje veličine i zahtjeva pojedinog testnog slučaja.

11.1. Particioniranje klasa ekvivalencije

Prva tehnika naziva se particioniranje klase ekvivalencije (eng. equivalence class partitioning). Navedena tehnika se koristi kako bi se podaci sličnih ili istih karakteristika grupirali u klase ekvivalencije. Pretpostavka grupiranja podataka u klase je da ih testni objekt obrađuje na isti način. Svaki podatak kojeg testni objekt na jednaki način obrađuje i njime upravlja stavlja se u klasu sa sličnim ili istim podacima. Na taj se način smanjuje potreban broj testnih podataka za pojedini testni slučaj. Više nije potrebno ispitati sve moguće ulazne podatke već je moguće izabrati jednog predstavnika klase i njega testirati. Pretpostavka je da će za bilo koju drugu ulaznu vrijednost iste klase ekvivalencije testni objekt pokazati isto ponašanje. Ista praksa se mora ponoviti za sve netočne vrijednosti. Prvo ih je potrebno grupirati u klase, a zatim ispitati jednog predstavnika. [3]

Jednostavan primjer klase ekvivalencije za program koji uzima dva broja te ispisuje njihov zbroj prikazan je u tablici 1. Navedeni program vraća rezultat samo pozitivnim brojevima u rasponu 0 < x <= 1.000.000 dok su ostali brojevi, slova i ostali interpunkcijski znakovi zabranjeni.

Tablica 1 Particioniranje podataka u klase ekvivalencije

Ulazni uvjet	**Klasa ekvivalencije sa validnim podacima**	**Klasa ekvivalencije sa ne validnim podacima**
Unos brojeva	0 < x >= 1.000.000 (predstavnik: 500)	x <= 0 (predstavnik: -500)
		x > 1.000.000 (predstavnik: 1.000.001)
		Slova i ostali znakovi (predstavnik: „a?“)

11.2. Analiza graničnih vrijednosti

Druga tehnika metode crne kutije je analiza graničnih vrijednost. Navedena tehnika za razliku od klasa ekvivalencije uzima podatak koji je na granici klase, a ne proizvoljni podataka iz navedene klase. Na taj se način pokrivaju rubni uvjeti klase te se povećava mogućnost pronalaska greške. Uz rubne uvjete ovu tehniku karakterizira razvoj testnih slučajeva za izlazne rezultate odnosno izlazne klase ekvivalencije. [3]

U testnim slučajevima se uzimaju tri vrijednosti: prva vrijednost koja je do rubne vrijednosti klase, druga vrijednost koja je predstavlja rubnu vrijednost klase i treća vrijednost koja je izvan ruba klase ekvivalencije. Svaka od tih vrijednosti tvori jedan testni slučaj koji se mora izvršiti. Na taj se način osigurava provjera valjanih i nevaljanih podataka kao ulaznih i izlaznih parametara.

Tehnika analize graničnih vrijednosti je nešto zastupljenija tehnika i bolje prihvaćana iz razloga što su rubni slučajevi često nedovoljno i neadekvatno objašnjeni te u puno slučajeva dovode do grešaka. Naravno da se postavlja pitanje „Što ako klasa ekvivalencije nema točno definiran rub?“. U tim slučajevima nije moguće ispitati granične vrijednosti već se odabire jedna vrijednost unutar klase ekvivalencije i jedna vrijednost izvan klase ekvivalencije.

Sljedeća tablica prikazuje tehniku analize rubnih slučajeva na primjeru iz prošlog podpoglavlja.

Tablica 2 Analiza graničnih vrijednosti

Rubni uvjet	**Vrijednost unutar klase**	**Vrijednost na rubu klase**	**Vrijednost izvan klase**	**Napomena**
X > 0	1	0	-1	0 i -1 dovode do greške
X <= 1.000.000	999.999	1.000.000	1.000.001	1.000.001 dovodi do greške

11.3. Testiranje tranzicijskog stanja

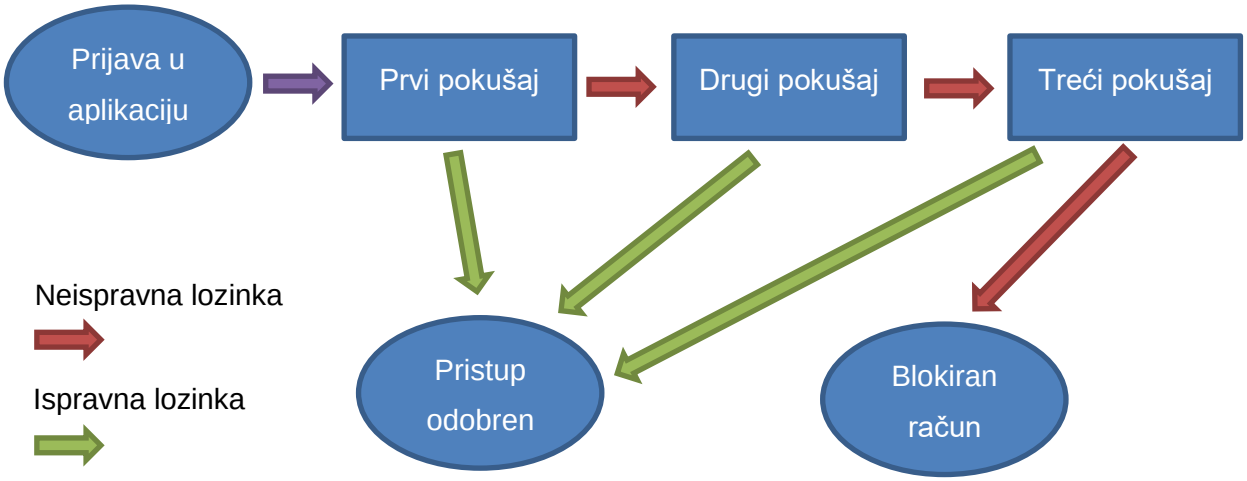
Tehnike klase ekvivalencije i analize graničnih vrijednosti ne uzimaju u obzir povijest, slijed i kombinacije ulaznih podataka, već uzimaju trenutni podatak izabran iz klase ekvivalencije. Na taj se način ne može simulirati i testirati programski proizvod na način kako ga koristi korisnik. Kako bi se to izbjeglo koristi se tehnika testiranja tranzicijskog stanja. Navedena tehnika koristi dijagrame i tablice tranzicije stanja kako bi objasnila koncepte na koji se način ulazni podaci transformiraju od ulaza do izlaza.

Kako bi se definirao testni slučaj pomoću tehnike tranzicijskog stanja potrebno je definirati:

- početno stanje testnog objekta,
- ulaze,
- očekivani ishod,
- očekivano završno stanje. [23]

Uz navedene informacije potrebno je za svako tranzicijsko stanje pojedinog testnog slučaja definirati:

- stanje prije tranzicije,
- inicijacijski događaj koji pokreće tranziciju,
- očekivanu reakciju izazvanu tranzicijom,
- sljedeće očekivano stanje. [23]



Slika 3 Dijagram tranzicijskog stanja [26]

Primjer na slici prikazuje dijagram tranzicijskog stanja prilikom prijave u aplikaciju. Možemo iščitati kako aplikacija nakon unosa točne lozinke daje odobrenje za ulazak korisniku, dok prilikom unosa netočne lozinke aplikacija nudi još dva pokušaja za prijavu. Ukoliko se tri puta za redom unese kriva lozinka tada će se račun blokirati. Navedeno testiranje se ne bi moglo ispitati putem tehnike klase ekvivalencije jer ona ne uzima u obzir stanja i povijest podataka prilikom izvršavanja testnog slučaja.

Iz ovog primjera možemo napraviti četiri testna slučaja. Prvi koji bi obuhvatio unošenje točne lozinke u prvom pokušaju. Zatim drugi testni slučaj u kojem bi se prvo unijela netočna lozinka, a zatim točna. Treći testni slučaj bi bio unošenje prva dva puta netočne lozinke te tek onda točnu lozinku. I zadnji testni slučaj bi bio unos tri puta za redom netočne lozinke. [26]

11.4. Testiranje temeljeno na slučajevima korištenja

Sljedeća tehnika testiranja bazira se na slučajevima korištenja. Kod navedene tehnike koriste se dijagrami slučajeva korištenja koji opisuju interakciju korisnika sa sustavom. Testerima se omogućuje kreiranje testnih slučajeva kroz slučajeve korištenja opisanih u dijagramima.

„Model slučajeva korištenja omogućava opis osnovnih funkcionalnih cjelina i njihovog ponašanja, vanjskih učesnika (uloga) i njihove interakcije sa sustavom, čiji je cilj ostvarivanje poslovnih ciljeva i slučajeva za testiranje, na razini sustava.“ [28]

Kako se modelom slučajeva korištenja opisuje interakcija korisnika i sustava najbolje je navedenu tehniku koristiti kod testiranja prihvatljivosti i testiranja sustava. Nedostaci koji pojavljuju u navedenoj tehnici su vezani za detalje koji su obuhvaćeni modelima slučajeva korištenja. Modeli slučajeva korištenja ne idu u detalje interakcije između korisnika i sustava što može dovesti do slijepe ulice kod testiranja navedenom tehnikom. Testeri nisu u mogućnosti odrediti daljnje testne slučajeve za testiranje činjenica koje nisu obuhvaćene modelom.

12. Tehnike testiranja pomoću metoda bijele kutije

Glavna razlika koja odvaja metode bijele kutije od metoda crne kutije je pristup unutarnjoj strukturi testnog objekta. Navedene metode nazivaju se još i metode testiranja na temelju koda. Cilj i temelj metoda bijele kutije je izvršavanje svakog dijela koda testnog objekta barem jednom. O metodi bijele kutije možete više pročitati u podpoglavlju 10.2, a u ovom poglavlju će biti opisane neke od tehnika koje se koriste pri izradi testnih slučajeva prilikom korištenja metoda bijele kutije.

12.1. Testiranje i pokrivenost iskaza

Prva tehnika koja se spominje i koristi u metodama bijele kutije je testiranje i pokrivenost iskaza. Navedenom tehnikom se nastoji pokriti svaki dio koda testnog objekta. Potrebno je definirati testne slučajeve koji će pokriti svaki dio koda odnosno izvršiti svaki dio koda u programskom proizvodu. Ukoliko se dogodi da nakon izvršenih svih testnih slučajeva postoji dio koda koji se nije niti jednom izvršio tada taj dio koda nazivamo mrtvim i njega treba ukloniti. [29]

Tehnika testiranja i pokrivenosti iskaza smatra se najjednostavnijima za primjenu, te se ujedno smatra i najmanje efikasnom tehnikom. Najmanje je efikasna iz razloga što je testeru dosta lagano napisati i narediti testne slučajeve kako bi izvršio svaki dio koda barem jedanput. [30]

12.2. Testiranje odluka/grana

Druga tehnika testiranja naziva se testiranje odluka/grana. Temelji se na testiranju odluka /grana koje postoje unutar testnog objekta. Za razliku od prethodne tehnike navedena tehnika ne uzima u obzir iskaze i njihovu pokrivenost već se samo temelji na pokrivenosti odluka /grana. Svaka odluka/grana može se izvršiti sa true ili false vrijednosti. Svaka od grana se mora izvršiti barem jednom kako bi test prošao u protivnom test pada i rezultira pronalaskom greške. Tehnikom testiranja odluka/grana pokrivaju se odluke: IF i SWITCH, te petlje: FOR, WHILE i DO WHILE. [30]

12.3. Testiranje uvjeta

Testiranje uvjeta je treća tehnika testiranja metodom bijele kutije. Navedena tehnika pokriva uvjete i njihovo zadovoljenje. Mogli bi reći kako je testiranje uvjeta pod tehnika tehnike testiranja odluka/grana. Kod složenih uvjeta razmatraju se elementarni uvjeti nezavisno jedan od drugome. Radi toga testiranje uvjeta može dati drugačije rezultate nego testiranje odluka/grana. No ukoliko se testni objekt sastoji samo od jednostavnih atomskih izraza tada će rezultat testiranja odluka/grana biti isti kao i kod testiranje uvjeta. [30]

12.4. Testiranje putanja

Testiranje putanja je četvrta tehnika testiranja pomoću metoda bijele kutije. Navedenom tehnikom nastoji se pokriti putanje od startnog do završnog čvora u grafu toka kontrole. Potrebno je pokriti bazični skup putanja. To bi u prijevodu značilo da je potrebno napraviti dovoljan broj testnih slučajeva koje će obuhvatiti linearno nezavisan skup putanja. Nezavisna putanja je on putanja koja se od svih drugih putanja razlikuje bar po jednom svom segmentu. [30]

13. Vrste testiranja

Postoje mnogobrojne podjele testiranja prema vrstama. Kada bi išli razlagati svaku podjelu mogli bi vidjeti kako većina njih ima slične značajke, te se na temelju tih značajki podjela testiranja može podijeliti na dvije generalno najčešće podjele. Prva podjela testiranja obuhvaća sve vrste testiranja koje se obavljaju u životnom ciklusu razvoja programskog proizvoda. Prema toj podjeli postoji četiri vrste testiranja:

- Jedinično testiranje (testiranje komponenti)
- Integracijsko testiranje (testiranje više komponenti od jednom)
- Testiranje sustava (testiranje cjelokupnog sustava)
- Testiranje prihvatljivosti [23]

Uz podjelu testiranja prema životnom ciklusu razvoja programskog proizvoda postoji generička podjela testiranja koja na općenitiji način objašnjava vrste testiranja. U nastavku slijedi objašnjenje svake od podjela sa pripadajućim vrstama testiranja.

13.1. Vrste testiranja prema životnom ciklusu razvoja programskog proizvoda

Životni ciklus razvoja programskog proizvoda razlikuje se među metodologijama, ali se može reći kako velika većina metodologija prolazi kroz četiri vrste testiranja. Prva vrsta testiranja koja se koristi u životnom ciklusu razvoja je jedinično testiranje, nakon toga slijedi integracijsko testiranje, pa testiranje sustava, testiranje prihvatljivosti i na kraju testiranje novih verzija proizvoda. Vrlo lako možemo zaključiti kako navedene vrste testiranja slijede obujam projekta tijekom njegovog razvoja. Integracijsko testiranje ima zadaću ispitati najmanje jedinice koda. Navedeno testiranje je iz tog razloga prvo po redu. Nakon toga je potrebno testirati nešto veće jedinice koda, te slijedi integracijsko testiranje. I onda kada smo gotovi sa velikom većinom posla oko razvoja slijedi testiranje sustava i testiranje prihvatljivosti. Kako je razvoj većine programskih proizvoda dugoročan tako postoji testiranje novih verzija proizvoda koje osigurava da nove funkcionalnosti, izmjene budu sigurne sa što manje grešaka. U nastavku slijedi detaljno objašnjenje svake vrste zasebno. [23]

13.1.1. Jedinično testiranje

Jedinično testiranje je prva vrsta testiranja prema podjeli na životni ciklus razvoja programskih proizvoda. Naziva se još i komponentno testiranje jer ispituje najmanje jedinice/komponente sustava. Izvršava se od strane programera koji su napisali kod. Iz tog

slijedi da je potrebno znanje programiranja za izvršavanje navedenih testova. Cilj jediničnog testiranja je izolirati najmanje dijelove programskog proizvoda i ispitati ih da li su točni u smislu zahtjeva i funkcionalnosti.

Jedinice koda se mogu nazivati moduli, jedinice, klase, funkcije, metode i itd. Naziv jedinice ovisi o programskom jeziku koji se koristi za razvoj programskog proizvoda. Vrlo važno je reći kako navedeno testiranje pripada testiranju pomoću metode bijele kutije. Razlog tomu je dostupnost unutarnje strukture i logike programa testeru tijekom testiranja. [23, pp. 42-50] Testni slučajevi se grade na način da se svaka linija koda od koje je jedinica sastavljena izvrši bar jednom.

Kako se jediničnim testiranjem ispituju najmanje jedinice koda tako se osigurava manja vjerojatnost pojavljivanja grešaka kod sastavljanja jedinica u veće komponente sustava. Uz to što se smanjuje vjerojatnost pojavljivanja grešaka jediničnim testiranjem dobivamo sustav koji je testabilan na najmanjoj razini razvoja. U prijevodu jediničnim testovima možemo lakše implementirati buduće promjene na kodu, na način da ne poremetimo ostali kod. [23, pp. 42-50]

Vrlo je važno detaljno ispitati najmanje jedinice jer su one temelj na kojem se program razvija, možemo ih usporediti sa temeljem kod zgrada i kuća. Ako one ne valjaju i ne izvršavaju svoje zadaće tada neće ni cijela struktura biti stabilna i izvršavati svoju zadaću. Također jediničnim testiranjem se pronalaze greške u ranijim fazama životnog ciklusu razvoja programskog proizvoda što rezultira manjim troškovima razvoja. Kao što je već spomenutu iznimno je korisno pronaći greške što ranije. Prvi razlog su manji troškovi razvoja, a drugi je manja vjerojatnost pojavljivanja grešaka u kasnijim fazama razvoja.

13.1.2. Integracijsko testiranje

Nakon jediničnog testiranja slijedi integracijsko testiranje. Ono se razlikuje od jediničnog po obujmu jedinica koje ulaze u samo testiranje. Integracijsko testiranje obuhvaća nekoliko komponenti (jedinica) koje se zajedno testiraju. Jedan od preduvjeta testiranju su pojedinačno testirane komponente koje ulaze u integracijsko testiranje. Ukoliko nisu pojedinačno odnosno jedinično testirane komponente tada nije pametno preći na integracijske testove. U suprotnom bi se mogle preskočiti greške koje postoje u pojedinačnim komponentama, a u integracijskim testovima te greške nisu vidljive. [23, pp. 50-58]

Cilj integracijskih testova je otkriti greške u sučeljima i interakciji između komponenti. Komponente zasebno mogu raditi bez greške, ali u interakciji može doći do slanja pogrešnih podataka, pokretanja pogrešnih funkcija i slično. Kako se programski proizvod gradi tako je potrebno graditi i integracijske testove. Svaka novo nadodana komponenta programa povećava složenost konačnog programa.

Integracijsko testiranje koje obuhvaća unutarnje komponente sustava naziva se užim integracijskim testiranjem, dok integracijsko testiranje koje obuhvaća neke od komponenti kao što su: baze podataka, datotečni sustavi ili mrežni sustavi naziva se šire integracijsko testiranje. Vrlo važno je napomenuti kako se kod jediničnog testiranja koriste lažni, zamaskirani podaci, dok se kod integracijskog testiranja koriste pravi podaci tj. podaci kojima raspolaže korisnik. Integracijski testovi se razlikuju od jediničnih prema sljedećim karakteristikama:

- Koriste se stvarne komponente koje programski proizvod koristi u produkciji
- Potrebno je puno više koda i obrade podataka
- Potrebno je puno više vremena za izvršavanje testova [31]

Kod integracijskog testiranja postoje četiri strategije kreiranja testnih planova:

- Integracija od dole prema gore (eng. bottom-up)
- Integracija od gore prema dole (eng. top-down)
- Ad hoc integracija
- Integracija pomoću kralježnice [23, pp. 50-58]

Kod strategije „od dole prema gore“ testovi počinju sa elementarnim komponentama programskog proizvoda koje smiju pozivati samo funkcije operacijskog sustava. Veći testni podsustavi kod navedene strategije sastavljaju se od već ispitanih komponenti koje se zatim testiraju. Glavni nedostatak ove strategije je to što testeri moraju simulirati komponente viših razina. [23, pp. 50-58]

Kod strategije „od gore prema dole“ testovi počinju sa testiranjem komponenata viših razina, te tek nakon njih slijedi testiranje komponenti nižih razina. Komponente viših razina su zapravo komponente koje pozivaju sve ostale komponente sustava. Pomoću ove strategije se omogućava da komponente više razine budu dio glavne testne okoline te na taj način testni usmjerivači nisu potrebni. [23, pp. 50-58]

Kod „ad hoc“ strategije integracija komponenti ide prema tome kako su napravljene. Na taj se način štedi vrijeme jer se komponente dodaju kako se razvijaju, te nema odstupanja u vremenima testiranja.

I kod zadnje strategije integracije pomoću kralježnice prvo se definiraju komponente „kralježnice“ koje se prvo testiraju, a zatim se na „kralježnicu“ integriraju ostale komponente. Prednost ovakvog načina što se komponente mogu integrirati prema bilo kojem redu, a glavni nedostatak je što ukoliko ne definiramo „kralježnicu“ ne možemo testirati ništa drugo. [23, pp. 50-58]

13.1.3. Testiranje sustava

Nakon što završimo testiranje pojedinačnih i više povezanih komponenti sustava krećemo sa testiranjem sustava kao cjeline. Navedeno testiranje je prvo testiranje u kojem se sustav kao cjelina testira. Cilj testiranja sustava je otkriti da li sustav kao cjelina odgovara propisanim korisničkim zahtjevima i standardima kvalitete. [29] Potrebno je utvrditi da li cjelokupni sustav zadovoljava funkcionalne i nefunkcionalne zahtjeve koje je klijent zatražio na početku projekta.

Izvršava se primjenom metode crne kutije. Unutarnja struktura sustava nije poznata testerima prilikom izvršavanja testnih slučajeva. Testeri su uglavnom neovisne osobe, tj. osobe koje ne poznaju logiku i strukturu koda, te nisu sudjelovali u razvoju. Testiranje sustava izvodi testove na programskom proizvodu u sličnoj okolini kao što je produkcijska. Na taj se način provjerava kako proizvod reagira u gotovo istoj okolini u kojoj će ga korisnici koristiti. Detaljno se prolaze i ispituju upute za korištenje programskog proizvoda, sistemska dokumentacija koja objašnjava arhitekturu programskog proizvoda, materijali vezani za trening korisnika (kod složenih programskih proizvoda) i slično. Iako je vrlo jasno što se pokušava provjeriti kod testiranja sustava, vrlo je teško izvršiti navedeno testiranje iz razloga što su u velikoj većini zahtjevi nepotpuni, nejasni ili uopće nisu definirani. Na taj način testeri ne mogu znati kako bi sustav trebao reagirati i što bi trebao raditi u određenim situacijama, te tako dolazi do ne otkrivanja grešaka.

Ukratko testiranjem sustava se testiraju, verificiraju i validiraju poslovni zahtjevi i aplikacijska arhitektura. Potrebno je imati detaljno razrađene i definirane zahtjeve kako bi u testiranje imalo učinka i kako bi se pronašao što veći broj grešaka.

13.1.4. Testiranje prihvatljivosti

Testiranje prihvatljivosti je zadnje testiranje u životnom ciklusu razvoja programskog proizvoda. Posljednji korak prije puštanja programskog proizvoda u produkcijsku okolinu. Prema ISTQB-u testiranje prihvatljivosti je

„formalno testiranje s obzirom na potrebe korisnika, zahtjeve i poslovne procese koji se provode kako bi se utvrdilo zadovoljava li sustav kriterije prihvatljivosti i kako bi se omogućilo korisniku, kupcima ili drugom ovlaštenom subjektu da odredi hoće li sustav prihvatiti ili ne.“ [5]

Cilj testiranja prihvatljivosti je ustvrditi da li programska struktura odgovora poslovnim zahtjevima i utvrditi da li je program spreman odnosno prihvatljiv za isporuku. Metoda crne kutije se koristi prilikom testiranja. Redoslijed ispitivanja se ne definira unaprijed već se definira ad-hoc kako proces teče.

U literaturi se navodi puno različitih podjela testiranja prihvatljivosti na tipove. Dok neki autori navode samo dva osnovna tipa testiranja prihvatljivosti interno (alfa) i eksterno (beta) testiranje, postoje autori koji dijele testiranje prihvatljivosti na više tipova:

- korisničko testiranje prihvatljivosti (eng. user acceptance testing)
- testiranje prihvatljivosti ugovora (eng. contract acceptance testing)
- operativno testiranje prihvatljivosti (eng. operational acceptance testing)
- testiranje prihvatljivosti s obzirom na okolinu (eng. field acceptance testing) - sastoji od alfa i beta testiranja [23, pp. 61-64]

13.1.4.1. Korisničko testiranje prihvatljivosti

Korisničkim testiranjem prihvatljivosti provjerava se da li programski proizvod ispravno radi sa korisničke strane. U ovom testiranju se korisnici smatraju krajnji korisnici kojima je programski proizvod namijenjen i stoga se testiranje provodi iz perspektive krajnjih korisnika, a ne perspektive klijenata/korisnika koji su ugovorili programski proizvod. Važno je odjeliti navedene perspektive jer nisu isti zahtjevi korisnika koji je ugovorio proizvod i krajnjih korisnika koji će navedeni proizvod koristiti. [23, pp. 61-64]

13.1.4.2. Testiranje prihvatljivosti ugovora

Testiranje prihvatljivosti ugovora je testiranje koje obuhvaća prolazak svih zahtjeva koji se nalaze na ugovoru između proizvođača i klijenta te utvrđivanje da li su navedeni zahtjevi implementirani unutar programskog proizvoda. Testiranje zahtjeva navedenih u ugovoru se već provelo kod testiranja sustava pa je si je onda dobro postaviti pitanje zašto to ponavljati. Razlog ponavljanja je to što kod testiranja prihvatljivosti korisnik ima glavnu ulogu te se sa njegove strane odnosno iz njegovog gledišta moraju izvesti testovi, dok su kod testiranja sustava okolina i testeri različiti i nisu postavljeni u perspektivu korisnika. [23, pp. 61-64]

13.1.4.3. Operativno testiranje prihvatljivosti

Operativno testiranje prihvatljivosti uključuje testiranje nefunkcionalnih zahtjeva. To mogu biti zahtjevi oporavka, dostupnosti, održavanja, prestanka rada i slično. Operativnim testiranjem se omogućava stabilnost programskog proizvoda. [32]

13.1.4.4. Interno - alfa testiranje

Interno i eksterno testiranje prihvatljivosti spada u testiranje prihvatljivosti s obzirom na okolinu. Kako bi proizvođača programskog proizvoda koštalo jako puno kreiranje različitih okruženja za pokrivanje svih mogućih scenarija tada se javlja testiranje s obzirom na okolinu. U navedenom tipu testiranja postoje interno i eksterno testiranje odnosno alfa i beta testiranje. U internom/alfa testiranju sudjeluju osobe iz poduzeća koje nisu sudjelovale u projektu razvoja.

Na taj se način po prvi puta programski proizvod testira sa korisnicima koji ne znaju puno o proizvodu. Provode se različiti testni slučajevi koje ne bi osobe koje su radile na projektu uzimale u obzir. Kako alfa testeri ispituju i koriste programski proizvod tako i prijavljuju greške i daju povratne informacije razvojnim timovima. Povratne informacije i prijavljene greške su od krucijalne važnosti za razvojne timove jer dobivaju informacije od korisnika koji nisu sudjelovali u razvoju i na taj način mogu poboljšati proizvod. [32]

13.1.4.5. Eksterno - beta testiranje

Eksterno testiranje naziva se još i beta testiranje. Ono se provodi izvan poduzeća sa testerima koji nisu zaposlenici poduzeća te također nisu sudjelovali u razvoju programskog proizvoda. Beta testiranje isključivo slijedi nakon alfa testiranja. U alfa testiranju sudjeluje manje ljudi od beta testiranja, ali je važno prvo napraviti alfa testiranje kako bi se smanjio broj grešaka prije izdavanja krajnjim korisnicima na testiranje. Kao i kod alfa testiranja svaki od testera (krajnjih korisnika) daje povratne informacije o proizvodu i greškama koje je zamijetio. Zatim razvojni tim popravlja i poboljšava programski proizvod i tako tvori stabilniji proizvod. Kod beta testiranja je istaknuta sloboda i ne predviđenost akcija koje korisnik izvršava nad proizvodom. Navedeno dovodi do otkrivanja većeg broja grešaka i zamjećivanja najsitnijih detalja. [32]

13.1.5. Testiranje novih verzija programskih proizvoda

Kako je većina programskih proizvoda napravljena za dugoročno korištenje tako je nastala potreba za ažuriranjem proizvoda, dodavanjem novih funkcionalnosti, sigurnosnih zakrpa, izmjenama tehnologije i slično. Svaka nova izmjena je zapravo nova verzija proizvoda i svakoj od njih je potrebno testiranje. Nitko ne želi koristiti novu verziju proizvoda ako nije prethodno testirana.

Prva pod vrsta testiranja novih verzija je testiranje izmjena nastalih održavanjem programskog proizvoda. Održavanjem programskog proizvoda ispravljaju se greške, nadodaju nove linije koda koje su potrebne kako bi programski proizvod slijedi tehnologiju, dodaju se zakrpe i slično. Svaku promjenu koja se napravi potrebno je ispitati i utvrditi da li je dovela do novih grešaka. Uz testiranje promjena nastalih održavanjem javlja se testiranje nakon daljnjeg razvoja programskog proizvoda. Ako je projektni menadžment definirao promjene koje se trebaju napraviti na proizvodu tada će se izdati nova verzija programskog proizvoda sa novom promjenama (funkcionalnostima). Kako bi korisnici ostali lišeni grešaka tada slijedi testiranje novih promjena. [23, p. 65]

I zadnje testiranje novih verzija može obuhvatiti testiranje kod inkrementalnog razvoja. Ukoliko se proizvod radi u inkrementima i tako izdaje korisnicima potrebno je nakon svakog razvijenog inkrementa izvršiti testiranje. SCRUM metodologija razvoja je idealni primjer razvoja

u inkrementima. Proizvod ne mora u cijelosti biti gotov kako bi se pustio u produkciju već se razvija u inkrementima i nakon svakog inkrementa šalje u produkciju.

13.2. Generičke vrste testiranja

Nakon vrsta koje slijede životni ciklus razvoja programskog proizvoda slijedi podjela testiranja na generičke vrste. Generičke vrste testiranja dijele se na:

- Funkcionalno testiranje
- Nefunkcionalno testiranje
- Testiranje strukture programskog proizvoda
- Testiranje povezano sa promjenama [23]

13.2.1. Funkcionalno testiranje

Funkcionalno testiranje je testiranje koje se provodi metodama crne kutije. Uključuje sve vrste testova koje provjeravaju programski sustav kroz postavljanje ulaza i provjeru izlaza. Samo ime kaže da se ovim testiranjem provjeravaju funkcionalnosti koje proizvod ima. Prema tome testovi imaju zadatak provjeriti ulaze i izlaze sa definiranim funkcionalnim zahtjevima. Funkcionalni zahtjevi definiraju kako će se programski proizvod ponašati, odnosno što očekivati kao izlaz za određeni ulaz.

Kod funkcionalnog testiranja prvi korak je odrediti funkcionalnosti koje ulaze u razmatranje odnosno testiranje. Nakon toga je potrebno definirati testne podatke koji će se koristiti prilikom izvođenja funkcionalnih testova. Navedeni korak bi se mogao zvati i definiranje ulaznih podataka. Nakon ulaza definirati izlaze koji se očekuju na temelju testnih podataka i specifikacija programskog proizvoda. Nakon toga je potrebno razraditi testne scenarije. Nakon što su testni scenariji gotovi, prelazi se na izvođenje testnih slučajeva i na samom kraju usporedba stvarno dobivenih sa očekivanim rezultatima. [24]

13.2.2. Nefunkcionalno testiranje

Kod programskih proizvoda je iznimno bitno da budu lagani za korištenje u smislu da korisnik ne mora voditi brigu o brzini, sigurnosti, pouzdanosti i drugim faktorima koji utječu na korištenje proizvoda. Kada se eliminiraju ti faktori korištenje proizvoda postaje puno zabavnije i jednostavnije za korisnika. Nefunkcionalno testiranje se bavi navedenim faktorima te se ono razlikuje od funkcionalnog testiranja po tome što u razmatranje ne ulaze direktno funkcionalnosti već atributi koji opisuju ponašanje funkcionalnosti ili atributi koji opisuju ponašanje sustava u cjelini. Koliko dobro navedena funkcionalnost ili sustav radi te kakva je kvaliteta izvedbe.

Testiranjem nefunkcionalnih zahtjeva programskog proizvoda postiže se bolja kvaliteta, bolje performanse i ugodnije korištenje što u konačnici dovodi do većeg zadovoljstva korisnika. ISO u svom standardu 9126 navodi kako zahtjevi nefunkcionalnog testiranja obuhvaćaju pouzdanost, upotrebljivost, učinkovitost, kompatibilnost i sigurnost.

Svrha funkcionalnog testiranja je utvrditi da funkcionalnosti odgovaraju zahtjevima korisnika, ali ne i da li se ti zahtjevi izvode na način da budu ugodni za korištenje. Kao primjer možemo uzeti prijavljivanje u sustav. Funkcionalni zahtjev je bio da se prijavljivanje obavi upisom emaila i lozinke te da se potom otvori početna stranica. Funkcionalno testiranje je potvrdilo kako su svi zahtjevi funkcionalnosti ispunjeni te da se provjerom podataka korisnika otvorila početna stranica. Nedugo nakon toga kada su se korisnici susreli sa tom funkcionalnosti došli su do zaključka kako prijavljivanje traje predugo, upotrebljivost odnosno preglednost je dosta loša i sigurnost upisanih podataka je upitna. Kako se ne bi događale takve stvari potrebno je provoditi nefunkcionalno testiranje.

Mayers u svojoj knjizi „The art of software testing“ navodi kako je nefunkcionalnim testiranjem potrebno obuhvatiti:

- Testiranje performansi – brzina obrade i vrijeme odziva
- Testiranje volumena – ponašanje sustava ovisno o količini podataka
- Testiranje opterećenja – ponaša sustava prilikom većeg broja zahtjeva, korisnika
- Testiranje sigurnosti – neovlašteni pristup podacima ili sustavu
- Testiranje robusnosti – ispitivanje pravilnog rukovanja iznimkama, greškama, odzivu na greške i slično
- Testiranje pouzdanosti – prosječno vrijeme između kvarova, broj grešaka u određenom vremenu
- Testiranje različitih konfiguracija – ponašanje na različitim operacijskim sustavima, hardverskim komponentama i slično
- Testiranje kompatibilnosti – uvoz, izvoz podataka, konverzije podataka i slično
- Testiranje upotrebljivosti – lakoća, jednostavnost, razumljivost korištenja programskog proizvoda
- Testiranje dokumentacije – provjera dokumentacije i detaljnost opisa
- Testiranje održivosti – ispitivanje održivosti, ažurnosti sustava

I na kraju je važno spomenuti kako su zahtjevi nefunkcionalnog testiranja više subjektivne nego objektivne prirode te se često izostavljaju detalji koji bi te zahtjeve prebacili na objektivnu stranu. Dobar primjer toga je zahtjev koji kaže kako bi sustav odnosno programski proizvod morao biti brz, ali opis brz nije točno specificiran. Za neke korisnike je 2 sekunde brzo, a za neke je 2 sekundi pre sporo. Prema tome vrlo je važno ići što preciznije i

dublje moguće u razgovore sa korisnicima i njihovom percepcijom zahtjeva. Nadalje se neki zahtjevi često podrazumijevaju pa ih testeri preskoče i ne ispitaju.

13.2.3. Testiranje strukture programskog proizvoda

Odmah po nazivu testiranja strukture programskog proizvoda moguće je zaključiti kako se radi o vrstama koje se izvode pomoću metoda bijele kutije. Struktura i logika programskih proizvoda je poznata tijekom testiranja. Cilj je proći svaki strukturni element barem jedanput te na taj način osigurati da svi strukturni elementi imaju svoju svrhu. Provjerava se tijek kontrole komponenata, hijerarhija poziva procedura i struktura izbornika. Ova vrsta testiranja s najčešće koristi u jediničnim i integracijskim testovima. [23]

13.2.4. Testiranje povezano sa promjenama (regresijsko testiranje)

Zadnja generička vrsta testiranja je testiranje povezano sa promjenama. Svaki programski proizvod u svom životnom ciklusu dobiva nadogradnje, promjene koje su neophodne za njegov daljnji rad. Kako se ne bi dogodilo da se novim nadodanim promjenama događaju greške u programskom proizvodu provodi se testiranje povezano sa promjenama.

Uz provjeru novog koda koje su promjene donijele potrebno je također provjeriti i stari kod odnosno ostale dijelove programskog proizvoda. Na taj se način osigurava da nadodane promjene ne utječu na dijelove koji nisu promijenjeni. Navedeno testiranje naziva se regresijsko testiranje. [24]

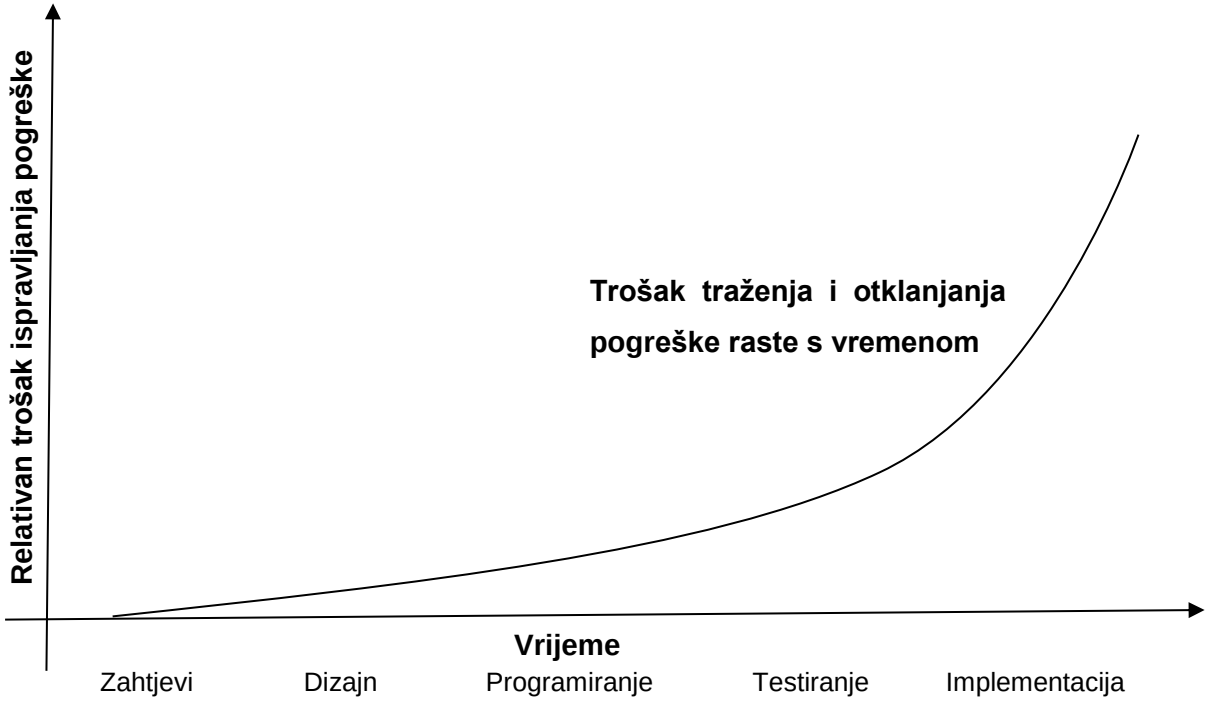
Pomoću navedenih testova postiže se bolja kontrola nad greškama, lakše upravljanje novim verzijama, promjenama koje se implementiraju na proizvodu i ublažavanje rizika. Regresijsko testiranje provodi se na ostalim vrstama testiranja poput funkcionalnog, nefunkcionalnog i strukturnog testiranja.

14. Testne smjernice

U ovom poglavlju su opisane smjernice koje se savjetuju pri testiranju programskih proizvoda. Navedene smjernice će dati testerima dodatne informacije što bi trebali raditi, a što izbjegavati. Smjernice imaju sličnu ulogu kao standardi jer usmjeravaju ljude što bi trebali raditi, a što ne. Jedina razlika između smjernica opisanih u ovom poglavlju i standarda je u tome što se slijeđenjem standarda dobiva certifikat koji potvrđuje da ste izvršili određene testove na način koji je opisan tim standardom, dok smjernice savjetuju testere, odnosno ne naređuju da se tako mora napraviti.

14.1. Početi testiranje što ranije

Prva smjernica govori o tome kako je potrebno početi testirati što ranije. Testiranje se ne bi smjelo izvršavati kada je proizvod već finiširan, već obrnuto. Potrebno je testiranje početi što ranije u procesu razvoja programskog proizvoda. Na taj se način štedi novac, vrijeme i otkriva veći broj grešaka koje bi inače završile u finalnom proizvodu. [8]



Slika 4 Trošak ispravljanja pogrešaka u različitim fazama razvoja programa [8, p. 8],

Na prethodnoj slici 1 može se vidjeti graf koji u prikazuje vrijeme i trošak ispravljanja pogreške po fazama razvoja programskog proizvoda. Vidljivo je kako krivulja raste kako vrijeme prolazi, tj. trošak ispravljanja grešaka raste gotovo eksponencijalno s vremenom. U

prvim ranim fazama razvoja proizvoda otklanjanje pogreške košta najmanje dok ista pogreška otkrivena u završnim fazama razvoja može koštati i nekoliko puta više.

14.2. Dokazati da program sadržava greške

Sljedeća smjernica govori o tome da testiranje mora rezultirati otkivanjem grešaka u radu programa. Testiranje je proces izvršavanja programa s namjerom pronalaženja pogrešaka. [3] Ne smije se ulaziti u testiranje s ciljem provjere da program radi kako treba. Ukoliko ulazimo u testiranje s tom namjenom vrlo vjerojatno nećemo pronaći onoliki broj grešaka koji bi pronašli s ciljem koji kaže da testiranje rezultira pronalaženjem grešaka. Navedena smjernica je jedna od najvažnijih u svijetu testiranja. Ona se spominje u jako puno literature i smatra se kamenom temeljcem u procesu testiranja.

14.3. Testiranje ovisi o kontekstu

Testiranje nije isto u svim situacijama, ono ovisi znatno o kontekstu samog testiranja. Na primjer nećemo jednako pristupiti testiranju ako testiramo mobilnu aplikaciju ili web aplikaciju. Svaka tehnologija u kojoj se razvijaju programski proizvodi ima svoje načine i najbolje prakse testiranja. Uz tehnologije vrlo važno je proučiti koji testovi najbolje djeluju za pojedine metode razvoja programskih proizvoda. Vodopadni model razvoja ima različitiji način kreiranja testova od agilnog modela. Također se ciljevi pojedinih vrsta testova razlikuju jedan od drugog. Pa tako jedinični i integracijski testovi imaju cilj osigurati da kod slijedi dizajn odnosno da pojedine jedinice rade na pravilan način te komuniciraju ispravno. Dok sustavski testovi imaju cilj osigurati da program radi na način kako je kupac želio. [8]

Vrsta testa te pristup koji će se koristiti ovisi o nizu čimbenika kao što su: vrsta sustava, standardi, korisnički zahtjevi, razina i vrsta rizika, cilj ispitivanja, postojanje dokumentacije, znanje testera, vrijeme, budžet, životni ciklus razvoja, itd.

14.4. Testni plan

Kao i u drugim fazama razvoja programskog proizvoda tako i u fazi testiranja postoji plan koji se donosi i slijedi tijekom te faze. Plan testiranja obuhvaća opseg samog testiranja, ciljeve koji se trebaju postići, metode, tehnike, vrste, okruženje i alate koji će se koristiti, zatim zaduženja, rizike koji se mogu pojaviti i razine koje će se obuhvatiti testiranjem.

Ukratko plan je sveobuhvatni dokument koji usmjerava uključene kako i na koji način će se provoditi testovi. Nažalost, u praksi se dešava da testovi uopće nemaju planove, te takvi testovi postaju teški za provođenje, vlada konfuzija među uključenima i rezultati su znatno

slabiji od pomno planiranih testova. Pomoću testnih planova se također poboljšava efikasnost i efektivnost testova jer se skraćuje vrijeme provođenja testova i povećava broj pronađenih grešaka. [8]

14.5. Testeri različitih karakteristika

Testeri, odnosno osobe zadužene za provođenje testova moraju imati određene karakteristike kako bi mogli provesti testiranje. Vrlo važno je pomno odabrati i rasporediti testere po pojedinim testovima. Ovisno o vrsti testa i testnoj razini testeri se razlikuju. Na slici broj 2 je moguće vidjeti kako završne testove provode upravitelji kvalitete. Oni moraju provjeriti da li programski proizvod prije preuzimanja od strane korisnika zadovoljava sve potrebne karakteristike koje je korisnik zatražio. Navedene testove ne bi trebao provoditi npr. programer. Programeri nemaju potrebne karakteristike, znanja i vještine kako bi mogli primijetiti što odgovora korisnikovim zahtjevima, a što ne. S druge strane jedinične testove nikako ne mogu provoditi npr. upravitelji kvalitete jer oni ne znaju kodirati i ne bi prepoznali moguće pogreške u kodu. Također su na slici spomenuta još tri testa i njihovi izvršitelji, pa tako imamo testova prihvaćanja koje provode korisnici/kupci, dok integracijske i sustavske testove provode neovisni specijalizirani testeri. [8]

Slika 5 Razine testiranja programskog proizvoda [8, p. 9]

14.6. Evaluacija i pregled testova

Evaluacija i pregled testova je princip koji slijedi pri samom kraju procesa testiranja. Naime testovi nisu kompletirani i nemaju velik značaj ako se ne evaluiraju i ako se ne dokumentira njihov rezultat. Prvo je potrebno analizirati testne rezultate i izvući zaključke o provedenom testu. Potrebno je sagledati što je pozitivnog, a što negativnog pronađeno. Što se mora i može unaprijediti, što je dobro odrađeno i slično. Nakon što se pomno analiziraju rezultati, potrebno je dokumentirati navedeno. Na taj se način osigurava postojanost testova za buduće nadogradnje, izmjene ili preglede programskog proizvoda. Novi programeri, testeri ili neki drugi učesnici na tom projektu imati će uvid u provedene testove, te će moći lakše detektirati novo nastale greške, implementirati nove funkcionalnosti i ispraviti pogreške. Uz pregled potrebno je evaluirati odrađen posao te tako zaključiti testove. Odrediti da li su provedeni testovi uspješno ili manje uspješno odrađeni. [8]

14.7. Nepostojanje pogrešaka

I zadnji princip koji valja istaknuti je zabluda o nepostojanju pogrešaka. Naime možemo provesti jako puno testova koji će rezultirati nepostojanjem pogrešaka, no to i dalje ne mora značiti kako taj programski proizvod radi kako treba. Ukoliko provedemo samo jedinične testove i ustvrdimo kako programski proizvod radi bez greške, bez zastoja i jako brzo. To ne mora značiti da taj proizvod odgovora korisničkim zahtjevima. Vrlo važno je da osim pozitivnog programskog testiranja kojim potvrđujemo da sustav radi ono što treba raditi, također provedemo i negativno testiranje kojim potvrđujemo da sustav ne radi ono što ne bi trebao raditi. [3]

15. Primjeri

Kako bi što lakše razumjeli testiranja programskih proizvoda napravio sam Android mobilnu aplikaciju naziva „Sport analytic“. Uz pomoć navedene aplikacije biti će prikazan plan testiranja, priprema testnih slučajeva, izrada testnih scenarija, priprema podatka i provedba samog testiranja uz pomoć različitih vrsta testiranja.

„Sport analytic“ aplikacija je zamišljena kao aplikacija koja omogućava lakšu evidenciju iznajmljenih i rezerviranih sportskih i ostalih ljetnih rekvizita. Postoji radno mjesto pod nazivom „Sportski radnik“ čija je zadaća iznajmljivanje sportskih rekvizita, ležaljki, suncobrana i brodova. Navedena aplikacija je namijenjena upravo „Sportskim radnicima“ koji prilikom iznajmljivanja koriste papire i olovku te na taj način bilježe rezervirane i iznajmljene rekvizite. Postojeći način je staromodan, spor, ne ekološki i često dolazi do krive komunikacije između radnika i njihovih nadležnih. Kako bi se ubrzao način bilježenja rezerviranih i iznajmljenih rekvizita, kako bi se smanjio na minimum nesporazum između radnika i njihovih nadležnih, te kako bi svi zapisi ostali pohranjeni na jednom mjestu s mogućnošću bržeg pretraživanja napravio sam navedenu aplikaciju.

Aplikacija je izrađena u Kotlin programskom jeziku verzije 1.2 [33], te je namijenjena za Android operacijski sustav. Lokalna baza podataka koja se nalazi na mobilnom uređaju je SQLite baza podataka [34], dok je baza podataka na serveru MySQL baza podataka [35]. Za izradu aplikacijskog programskog sučelja (eng. application programming interface, API) korišten je PHP skriptni jezik [36]. Kod same aplikacije možete pronaći na https://github.com/JosipDudas/sport_analytic_kotlin_android.git, dok kod web servisa možete pronaći na https://github.com/JosipDudas/sport_analytic_webservice.git.

U nastavku poglavlja slijede primjeri testova napravljenih nad „Sport analytic“ aplikacijom.

15.1. Jedinično testiranje

Prvo na redu testiranje prema životnom ciklusu proizvoda je jedinično testiranje. Jedinično testiranje obuhvaća testiranje najmanjih jedinica koda koje se mogu i imaju svrhu testirati. Primjer koji će biti detaljnije opisan u nastavku tiče se testiranja LoginViewModel klase koja obuhvaća logiku LoginActivity klase. Navedeno testiranje podijeljeno je na testne slučajeve koji obuhvaćaju testiranje pojedinih funkcija unutar klase. Svaka od funkcija biti će ispitana na način da se pokriju svi mogući uvjeti unutar funkcije. Namjera je pokriti sve uvjete i na taj način osigurati da funkcije odrađuju posao na predviđeni način. Važno je napomenuti kako se neki od testova u nastavku mogu smatrati integracijskim testovima iz razloga što se unutar funkcije poziva neka druga funkcija.

Prvi korak kod procesa testiranja je izrada testnog plana, a zatim ide analiza i dizajn testova tj. testnih slučajeva koji se potom izvršavaju i zapisuje se njihov rezultat, te na samom kraju uspoređuju dobiveni rezultati sa očekivanim rezultatima kako bi se utvrdilo da li je test prošao ili pao.

Testni plan:

Tablica 3 Testni plan (jedinično testiranje)

Svrha	Ispitati funkcije, tj. najmanje jedinice koda koje se mogu testirati.
Cilj	Ispitati da funkcionalnost prijave u aplikaciju radi ispravno. Ispitati sve funkcije „LoginViewModel“ klase.
Resursi	Kod „Sport analytic“ aplikacije, Android studio 3.4 [37], Mockito 2 testni okvir (eng. framework) [38], JUnit 4 testni okvir [39], programer
Zaduženja	Josip Dudaš – izrada plana testiranja Josip Dudaš – izrada testnog slučaja Josip Dudaš – provođenje testiranja
Strategija - aktivnosti	Jedinično testiranje, testiranje metodom bijele kutije, automatizirano testiranje

Test_1:

Tablica 4 Testni slučaj: onLogin-1A

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „onLogin()“
Oznaka testnog slučaja	onLogin-1A
Testni slučaj	Provjera onLogin() funkcije na unos validnog emaila i lozinke
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studija 3.4 [37], uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „login_success_loginIsSuccessTrue“ koji će pozvati „onLogin()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Proslijediti validni email i lozinku navedenoj funkciji 5. Pokrenuti jedinični test
Očekivani rezultat	Prijava je uspješno odrađena, varijabla „loginIsSuccess“ je postavljena na true, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ nije niti jednom pozivana i mjenjana
Stvarni rezultat	Prijava je uspješno odrađena, varijabla „loginIsSuccess“ je postavljena na true, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ nije niti jednom pozivana i mjenjana
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun login_success_loginIsSuccessTrue() {
    runBlocking {
        // given
        val id = UUID.randomUUID().toString().toUpperCase()
        val getLoginResponse = GetLoginResponse()
        getLoginResponse.id = id
        getLoginResponse.firstname = "Test"
        getLoginResponse.lastname = "Test"
        getLoginResponse.password = "test"
        getLoginResponse.email = "test@foi.hr"
        getLoginResponse.position = "Admin"
        getLoginResponse.address = "Test"
        getLoginResponse.sex = "F"
        getLoginResponse.company_id = id
        getLoginResponse.status = true
        getLoginResponse.message = "Test"
        classUnderTest.email.value = "test@foi.hr"
        classUnderTest.password.value = "test"
        val mockUserDao = mock<UserDao>()

        // when
        whenever(mockSportAnalyticService.userLogin(any(),
any())).thenReturn(Calls.response(getLoginResponse))
        whenever(mockConnector.userDao()).thenReturn(mockUserDao)
        classUnderTest.login()

        // then
        verify(errorObserver, never()).onChanged(any())
        verify(mockPreferences, times(1)).setUser(any())
        verify(mockConnector.userDao(), times(1)).insertUser(any())

        argumentCaptor<Boolean>{
            verify(progressObserver, times(2)).onChanged(capture())
            assertEquals(true, allValues[0])
            assertEquals(false, allValues[1])
        }

        argumentCaptor<Boolean>{
            verify(loginIsSuccessObserver, times(1)).onChanged(capture())
            assertEquals(true, allValues[0])
        }
    }
}
```

Test_2:

Tablica 5 Testni slučaj: onLogin-1B

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „onLogin()“
Oznaka testnog slučaja	onLogin-1B
Testni slučaj	Provjera onLogin() funkcije prilikom odstupanja (eng. exception)
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studija 3.4 [37], uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „login_exception_errorMessageShow“ koji će pozvati „onLogin()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Kreirati odstupanje (eng. exception) prilikom poziva API-a 5. Pokrenuti jedinični test
Očekivani rezultat	Prijava je nije uspješno odrađena, varijabla „loginIsSuccess“ nije niti jednom pozivana i mijenjanja, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ je mijenjanja jedanput
Stvarni rezultat	Prijava je nije uspješno odrađena, varijabla „loginIsSuccess“ nije niti jednom pozivana i mijenjanja, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ je mijenjanja jedanput
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun login_exception_errorMessageShow() {
    runBlocking {
        // given

        // when
        whenever(mockSportAnalyticService.userLogin(any(),
any())).thenReturn(Calls.failure(IOException()))
        classUnderTest.login()

        // then
        verify(loginIsSuccessObserver, never()).onChanged(any())
        verify(mockPreferences, never()).setUser(any())

        argumentCaptor<IOException>{
            verify(errorObserver, times(1)).onChanged(capture())
        }

        argumentCaptor<Boolean>{
            verify(progressObserver, times(2)).onChanged(capture())
            assertEquals(true, allValues[0])
            assertEquals(false, allValues[1])
        }
    }
}
```

Test_3:

Tablica 6 Testni slučaj: onLogin-1C

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „onLogin()“
Oznaka testnog slučaja	onLogin-1C
Testni slučaj	Provjera onLogin() funkcije nakon što je API odgovor bio tipa „false“
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „login_responseIsFailed_errorMessageShow“ koji će pozvati „onLogin()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Kreirati API odgovor koji vraća „status = false“ 5. Pokrenuti jedinični test
Očekivani rezultat	Prijava je nije uspješno odrađena, varijabla „loginIsSuccess“ je jednom pozvana i postavljena na false, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ je mijenjanja jedanput
Stvarni rezultat	Prijava je nije uspješno odrađena, varijabla „loginIsSuccess“ je jednom pozvana i postavljena na false, varijabla „progress“ je prvo postavljena na true a zatim na false, varijabla „error“ je mijenjanja jedanput
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun login_responseIsFailed_errorMessageShow() {
    runBlocking {
        // given
        val id = UUID.randomUUID().toString().toUpperCase()
        val getLoginResponse = GetLoginResponse()
        getLoginResponse.id = id
        getLoginResponse.firstname = "Test"
        getLoginResponse.lastname = "Test"
        getLoginResponse.password = "test"
        getLoginResponse.email = "test@foi.hr"
        getLoginResponse.position = "Admin"
        getLoginResponse.address = "Test"
        getLoginResponse.sex = "F"
        getLoginResponse.company_id = id
        getLoginResponse.status = false
        getLoginResponse.message = "Test"
        classUnderTest.email.value = "test@foi.hr"
        classUnderTest.password.value = "test"
        val mockUserDao = mock<UserDao>()

        // when
        whenever(mockSportAnalyticService.userLogin(any(),
any())).thenReturn(Calls.response(getLoginResponse))
        whenever(mockConnector.userDao()).thenReturn(mockUserDao)
        classUnderTest.login()

        // then
        verify(mockPreferences, never()).setUser(any())
        verify(mockConnector.userDao(), never()).insertUser(any())

        argumentCaptor<Boolean>{
            verify(progressObserver, times(2)).onChanged(capture())
            assertEquals(true, allValues[0])
            assertEquals(false, allValues[1])
        }

        argumentCaptor<Boolean>{
            verify(loginIsSuccessObserver, times(1)).onChanged(capture())
            assertEquals(false, allValues[0])
        }

        argumentCaptor<IllegalStateException>{
            verify(errorObserver, times(1)).onChanged(capture())
        }
    }
}
```

Test_4:

Tablica 7 Testni slučaj: isFormValid -1A

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „isFormValid“
Oznaka testnog slučaja	isFormValid -1A
Testni slučaj	Provjera isFormValid () funkcije sa proslijeđenim validnim podacima emaila i lozinke
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „isFormValid_true_enableLoginButtonPostOnTrue“ koji će pozvati „isFormValid()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Postaviti email vrijednost na „test@foi.hr“ 5. Postaviti password vrijednost na „test“ 6. Pokrenuti jedinični test
Očekivani rezultat	Varijabla „enableLoginButton“ postavljena je na true
Stvarni rezultat	Varijabla „enableLoginButton“ postavljena je na true
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun isFormValid_true_enableLoginButtonPostOnTrue() {
    // given
    val email = "test@foi.hr"
    val password= "test"
    classUnderTest.email.value = email
    classUnderTest.password.value = password

    // when
    classUnderTest.isFormValid()

    // then
    argumentCaptor<Boolean>{
        verify(enableLoginButtonObserver, times(1)).onChanged(capture())
        assertEquals(true, allValues[0])
    }
```

Test_5:

Tablica 8 Testni slučaj: isFormValid -1B

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „isFormValid“
Oznaka testnog slučaja	isFormValid -1B
Testni slučaj	Provjera isFormValid () funkcije sa proslijeđenim ne validnim emailom
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „isFormValid_emailIsIncorrect_enableLoginButtonPostOnFalse“ koji će pozvati „isFormValid()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Postaviti email vrijednost na „test“ 5. Postaviti password vrijednost na „test“ 6. Pokrenuti jedinični test
Očekivani rezultat	Varijabla „enableLoginButton“ postavljena je na false
Stvarni rezultat	Varijabla „enableLoginButton“ postavljena je na false
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun isFormValid_emailIsIncorrect_enableLoginButtonPostOnFalse() {
    // given
    val email = "test"
    val password= "test"
    classUnderTest.email.value = email
    classUnderTest.password.value = password

    // when
    classUnderTest.isFormValid()

    // then
    argumentCaptor<Boolean>{
        verify(enableLoginButtonObserver, times(1)).onChanged(capture())
        assertEquals(false, allValues[0])
    }
}
```

Test_6:

Tablica 9 Testni slučaj: isFormValid -1C

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „isFormValid“
Oznaka testnog slučaja	isFormValid -1C
Testni slučaj	Provjera isFormValid () funkcije sa neproslijeđenom vrijednosti emaila
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „isFormValid_emailIsIncorrect_enableLoginButtonPostOnFalse“ koji će pozvati „isFormValid()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Postaviti password vrijednost na „test“ 5. Pokrenuti jedinični test
Očekivani rezultat	Varijabla „enableLoginButton“ postavljena je na false
Stvarni rezultat	Varijabla „enableLoginButton“ postavljena je na false
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun isFormValid_emailOrPasswordIsEmpty_enableLoginButtonPostOnFalse() {
    // given
    val password= "test"
    classUnderTest.password.value = password

    // when
    classUnderTest.isFormValid()

    // then
    argumentCaptor<Boolean>{
        verify(enableLoginButtonObserver, times(1)).onChanged(capture())
        assertEquals(false, allValues[0])
    }
}
```

Test_7:

Tablica 10 Testni slučaj: onRegisterClick-1A

Oznaka testnog scenarija	Prijava
Testni scenarij	Provjera funkcije „onRegisterClick“
Oznaka testnog slučaja	onRegisterClick-1A
Testni slučaj	Provjera onRegisterClick () funkcije
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Mockito 2 biblioteka (eng. framework) [38] i JUnit 4 testni okvir [39]
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginViewModelUnitTest.kt“, unutar datoteke „com.dudas.sportanalytic (test)“ 3. Kreirati jedinični test pod nazivom „onRegisterClick_registerButtonIsClicked_startRegistrationActivityPostTrue“ koji će pozvati „onRegisterClick ()“ funkciju koja se nalazi unutar „LoginViewModel“ klase 4. Pokrenuti jedinični test
Očekivani rezultat	Varijabla „startRegistrationActivity“ postavljena je na true
Stvarni rezultat	Varijabla „startRegistrationActivity“ postavljena je na true
Prošao/Pao	Prošao

Kod jediničnog testa:

```
@Test
fun
onRegisterClick_registerButtonIsClicked_startRegistrationActivityPostTrue()
{
    // given

    // when
    classUnderTest.onRegisterClick()

    // then
    argumentCaptor<Boolean>{
        verify(startRegistrationActivityObserver,
times(1)).onChanged(capture())
        assertEquals(true, allValues[0])
    }
}
```

15.2. Integracijsko testiranje

Nakon jediničnih testova slijede integracijski testovi. Kod testiranja Android mobilnih aplikacija teško je razgraničiti razliku između jediničnih i integracijskih testova. Svi jedinični testovi koji spadaju u kategoriju srednjih testova (testovi srednje veličine) mogu se smatrati integracijskim testovima. Takvi testovi u sebi ispituju interakciju između malih jedinica koda. Prema navedenoj konstataciji možemo reći kako jedinični testovi iz prethodnog poglavlja nisu svi u potpunosti jedinični jer ispituju komunikaciju više jedinica koda odnosno ovom slučaju funkcija. Ali kako ne postoji točno definirano što su integracijski testovi, odlučio sam staviti UI testove odnosno testove korisničkog sučelja (eng. user interface). Navedeni testovi ispituju korisničko sučelje i komunikaciju između integriranih jedinica koda. U nastavku slijedi primjer provedenog testiranja.

Testni plan

Tablica 11 Testni plan integracijskih testova

Svrha	Ispitati interakciju između integriranih dijelova aplikacije.
Cilj	Ispitati da funkcionalnost prijave u aplikaciju radi ispravno te da integrirani sustavi rade na predviđen način. Predviđen način bi bio onaj u kojem registrirani korisnici prilikom unosa ispravnih podataka dobivaju pristup aplikaciji, dok ne registrirani i netočno upisani podaci rezultiraju ispisom pogreške prilikom prijave.
Resursi	Kod „Sport analytic“ aplikacije, Android studio 3.4.2 [37], Espresso 3.2.0 biblioteka, JUnit 4 testni okvir (eng. framework), Android emulator Android 9 (API level 28), programer
Zaduženja	Josip Dudaš – izrada plana testiranja Josip Dudaš – izrada testnog slučaja Josip Dudaš – provođenje testiranja
Strategija - aktivnosti	Integracijsko testiranje, testiranje metodom sive kutije, automatizirano testiranje

Test_1:

Tablica 12 Testni slučaj: onLogin-1A

Oznaka testnog scenarija	Prijava-UI
Testni scenarij	Provjera prijave unutar aplikacije
Oznaka testnog slučaja	Prijava-1A
Testni slučaj	Provjera funkcionalnosti prijave prilikom unosa validnog emaila i lozinke
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Espresso 3.2.0 biblioteka i JUnit 4 testni okvir (eng. framework) [39], Android emulator Android 9 (API level 28)
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginActivityUITest.kt“, unutar datoteke „com.dudas.sportanalytic (androidTest)“ 3. Kreirati test korisničkog sučelja (eng. user interface) pod nazivom „login_validEmailAndPassword_SuccessLogin“ 4. Test mora ispitati: • Edit view pod nazivom user_email (nazočnost i dostupnost) • Edit view pod nazivom user_password (nazočnost i dostupnost) • Button pod nazivom user_login (nazočnost i dostupnost) • Text view pod nazivom txt_registration (nazočnost) • Text view pod nazivom txt_register (nazočnost i dostupnost) • Očistiti user_email polje i upisati „josdudas@gmail.com“ • Provjeriti da li se user_login može kliknuti • Očistiti user_password polje i upisati „josip“ • Kliknuti na user_login • Provjeriti da li je otvorena MainActivity klasa • Provjeriti da li je ispisana poruka da lokacija nije odabrana • Provjeriti podatke spremljenog korisnika, točnije njegov email • Otvoriti ladicu (eng. drawer) • Pričekati sekundu radi animacije • Kliknuti na logout (odjaviti se iz aplikacije) • Provjeriti da li se LoginActivity klasa pokrenula 5. Pokrenuti test

Očekivani rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tipka za prijavu u aplikaciju je dostupna tek nakon upisa validnih podataka, • nakon klika na tipku za prijavu prijava je uspješno odrađena, • MainActivity klasa je pokrenuta, • ispisana je poruka da lokacija nije odabrana, • korisnički podaci su korektno spremljeni i • odjava je izvršena uspješno
Stvarni rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tipka za prijavu u aplikaciju je dostupna tek nakon upisa validnih podataka, • nakon klika na tipku za prijavu prijava je uspješno odrađena, • MainActivity klasa je pokrenuta, • ispisana je poruka da lokacija nije odabrana, • korisnički podaci su korektno spremljeni i odjava je izvršena uspješno
Prošao/Pao	Prošao

Kod napravljenog testa:

```
@Test
fun login_validEmailAndPassword_SuccessLogin() {
    // Given

    //When
    onView(withId(R.id.nd_email)).check(matches(isDisplayed()))
    onView(withId(R.id.nd_email)).check(matches(isEnabled()))

    onView(withId(R.id.nd_password)).check(matches(isDisplayed()))
    onView(withId(R.id.nd_password)).check(matches(isEnabled()))

    onView(withId(R.id.nd_login)).check(matches(isDisplayed()))
    onView(withId(R.id.nd_login)).check(matches(not(isEnabled())))

    onView(withId(R.id.txt_registration)).check(matches(isDisplayed()))

    onView(withId(R.id.txt_register)).check(matches(isDisplayed()))
    onView(withId(R.id.txt_register)).check(matches(isEnabled()))


onView(withId(R.id.nd_email)).perform(clearText(),typeText(VALID_EMAIL))
    onView(withId(R.id.nd_login)).check(matches(not(isEnabled())))

onView(withId(R.id.nd_password)).perform(clearText(),typeText(VALID_PASS))
    onView(withId(R.id.nd_login)).check(matches(isEnabled()))

    onView(withId(R.id.nd_login)).perform(click())

    intended(hasComponent(ComponentName(intentsTestRule.activity,
MainActivity::class.java)))


onView(withText(R.string.location_is_unknown)).check(matches(isDisplayed())
)

    // Then
    assertEquals(VALID_EMAIL,
intentsTestRule.activity.preferences.getUser()!!.email)

    onView(withContentDescription("Navigate up")).perform(click())

    Thread.sleep(1000)

    onView(withText(R.string.logout)).perform(click())

    intended(hasComponent(ComponentName(intentsTestRule.activity,
LoginActivity::class.java)))
}
```

Test_2:

Tablica 13 Testni slučaj: onLogin-1B

Oznaka testnog scenarija	Prijava-UI
Testni scenarij	Provjera prijave unutar aplikacije
Oznaka testnog slučaja	Prijava-1B
Testni slučaj	Provjera funkcionalnosti prijave prilikom unosa podataka neregistriranog korisnika
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Espresso 3.2.0 biblioteka i JUnit 4 testni okvir (eng. framework) [39], Android emulator Android 9 (API level 28)
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginActivityUITest.kt“, unutar datoteke „com.dudas.sportanalytic (androidTest)“ 3. Kreirati test korisničkog sučelja (eng. user interface) pod nazivom „login_invalidEmailAndPassword_unsuccessfulLogin“ 4. Test mora ispitati: • Edit view pod nazivom user_email (nazočnost i dostupnost) • Edit view pod nazivom user_password (nazočnost i dostupnost) • Button pod nazivom user_login (nazočnost i dostupnost) • Text view pod nazivom txt_registration (nazočnost) • Text view pod nazivom txt_register (nazočnost i dostupnost) • Očistiti user_email polje i upisati „test@gmail.com“ • Provjeriti da se user_login ne može kliknuti • Očistiti user_password polje i upisati „test“ • Provjeriti da li se user_login može kliknuti • Kliknuti na user_login • Provjeriti da li je ispisana poruka o tome da su lozinka i email krivo uneseni i da taj korisnik ne postoji • Provjeriti da li su spremljeni bilo kakvi podaci o korisniku 5. Pokrenuti test
Očekivani rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje,

	• tipka za prijavu u aplikaciju je dostupna tek nakon upisa validnih podataka, • nakon klika na tipku za prijavu poruka o neuspjeloj prijavi je ispisana
Stvarni rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tipka za prijavu u aplikaciju je dostupna tek nakon upisa validnih podataka, nakon klika na tipku za prijavu poruka o neuspjeloj prijavi je ispisana
Prošao/Pao	Prošao

Kod napravljenog testa:

```
@Test
fun login_invalidEmailAndPassword_unsuccessfulLogin() {
    // Given

    //When
    onView(withId(R.id.user_email)).check(matches(isDisplayed()))
    onView(withId(R.id.user_email)).check(matches(isEnabled()))

    onView(withId(R.id.user_password)).check(matches(isDisplayed()))
    onView(withId(R.id.user_password)).check(matches(isEnabled()))

    onView(withId(R.id.user_login)).check(matches(isDisplayed()))
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

    onView(withId(R.id.txt_registration)).check(matches(isDisplayed()))

    onView(withId(R.id.txt_register)).check(matches(isDisplayed()))
    onView(withId(R.id.txt_register)).check(matches(isEnabled()))


onView(withId(R.id.user_email)).perform(clearText(),typeText(INVALID_EMAIL)
)
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

onView(withId(R.id.user_password)).perform(clearText(),typeText(INVALID_PAS
S))
    onView(withId(R.id.user_login)).check(matches(isEnabled()))

    onView(withId(R.id.user_login)).perform(click())


onView(withText(R.string.invalid_email_or_password)).inRoot(RootMatchers.wi
thDecorView(Matchers.not(Is.`is`(intentsTestRule.activity.window.decorView)
))).check(matches(isDisplayed()))

    // Then
    assertEquals(null, intentsTestRule.activity.preferences.getUser())
}
```

Test_3:

Tablica 14 Testni slučaj: onLogin-1C

Oznaka testnog scenarija	Prijava-UI
Testni scenarij	Provjera prijave unutar aplikacije
Oznaka testnog slučaja	Prijava-1C
Testni slučaj	Provjera funkcionalnosti prijave prilikom unosa nepravilnog emaila
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Espresso 3.2.0 biblioteka i JUnit 4 testni okvir (eng. framework) [39], Android emulator Android 9 (API level 28)
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginActivityUITest.kt“, unutar datoteke „com.dudas.sportanalytic (androidTest)“ 3. Kreirati test korisničkog sučelja (eng. user interface) pod nazivom „login_invalidEmailFormat_loginButtonDisabled“ 4. Test mora ispitati: • Edit view pod nazivom user_email (nazočnost i dostupnost) • Edit view pod nazivom user_password (nazočnost i dostupnost) • Button pod nazivom user_login (nazočnost i dostupnost) • Text view pod nazivom txt_registration (nazočnost) • Text view pod nazivom txt_register (nazočnost i dostupnost) • Očistiti user_email polje i upisati „test@foi“ • Provjeriti da se user_login ne može kliknuti • Očistiti user_password polje i upisati „test“ • Provjeriti da se user_login ne može kliknuti • Provjeriti da li su spremljeni bilo kakvi podaci o korisniku 5. Pokrenuti test
Očekivani rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tipka za prijavu u aplikaciju nije dostupna jer nisu uneseni ispravni podaci
Stvarni rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje,

	• tipka za prijavu u aplikaciju nije dostupna jer nisu uneseni ispravni podaci
Prošao/Pao	Prošao

Kod napravljenog testa:

```
@Test
fun login_invalidEmailFormat_loginButtonDisabled() {
    // Given

    //When
    onView(withId(R.id.user_email)).check(matches(isDisplayed()))
    onView(withId(R.id.user_email)).check(matches(isEnabled()))

    onView(withId(R.id.user_password)).check(matches(isDisplayed()))
    onView(withId(R.id.user_password)).check(matches(isEnabled()))

    onView(withId(R.id.user_login)).check(matches(isDisplayed()))
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

    onView(withId(R.id.txt_registration)).check(matches(isDisplayed()))

    onView(withId(R.id.txt_register)).check(matches(isDisplayed()))
    onView(withId(R.id.txt_register)).check(matches(isEnabled()))


onView(withId(R.id.user_email)).perform(clearText(),typeText(INVALID_EMAIL_
FORMAT))
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

onView(withId(R.id.user_password)).perform(clearText(),typeText(INVALID_PAS
S))
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

   // Then
    assertEquals(null, intentsTestRule.activity.preferences.getUser())
}
```

Test_4:

Tablica 15 Testni slučaj: onLogin-1D

Oznaka testnog scenarija	Prijava-UI
Testni scenarij	Provjera prijave unutar aplikacije
Oznaka testnog slučaja	Prijava-1D
Testni slučaj	Provjera funkcionalnosti prijave ukoliko korisnik nije registriran, dostupnost registracije
Preduvjeti	Kod „Sport analytic“ aplikacije otvoren unutar Android studia, uključena Espresso 3.2.0 biblioteka i JUnit 4 testni okvir (eng. framework) [39], Android emulator Android 9 (API level 28)
Testni koraci	1. Otvoriti kod aplikacije 2. Otvoriti klasu pod nazivom „LoginActivityUITest.kt“, unutar datoteke „com.dudas.sportanalytic (androidTest)“ 3. Kreirati test korisničkog sučelja (eng. user interface) pod nazivom „login_registerClick_openRegisterActivity“ 4. Test mora ispitati: • Edit view pod nazivom user_email (nazočnost i dostupnost) • Edit view pod nazivom user_password (nazočnost i dostupnost) • Button pod nazivom user_login (nazočnost i dostupnost) • Text view pod nazivom txt_registration (nazočnost) • Text view pod nazivom txt_register (nazočnost i dostupnost) • Očistiti user_email polje i upisati „test@gmail.com“ • Očistiti user_password polje i upisati „test“ • Kliknuti na txt_register • Provjeriti da li je otvorena RegisterationActivity klasa • Provjeriti da li su spremljeni bilo kakvi podaci o korisniku 5. Pokrenuti test
Očekivani rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tekstualno polje registracija je prisutno, • nakon klika na tekstualno polje za registraciju otvorila se RegistrationActivity klasa

Stvarni rezultat	• svi resursi unutar LoginActivity klase su prisutni i sadržavaju dogovoreno stanje, • tekstualno polje registracija je prisutno, • nakon klika na tekstualno polje za registraciju otvorila se RegistrationActivity klasa
Prošao/Pao	Prošao

Kod napravljenog testa:

```
@Test
fun login_registerClick_openRegisterActivity() {
    // Given

    //When
    onView(withId(R.id.user_email)).check(matches(isDisplayed()))
    onView(withId(R.id.user_email)).check(matches(isEnabled()))

    onView(withId(R.id.user_password)).check(matches(isDisplayed()))
    onView(withId(R.id.user_password)).check(matches(isEnabled()))

    onView(withId(R.id.user_login)).check(matches(isDisplayed()))
    onView(withId(R.id.user_login)).check(matches(not(isEnabled())))

    onView(withId(R.id.txt_registration)).check(matches(isDisplayed()))

    onView(withId(R.id.txt_register)).check(matches(isDisplayed()))
    onView(withId(R.id.txt_register)).check(matches(isEnabled()))


onView(withId(R.id.user_email)).perform(clearText(),typeText(INVALID_EMAIL)
)

onView(withId(R.id.user_password)).perform(clearText(),typeText(INVALID_PAS
S))

    onView(withId(R.id.txt_register)).perform(click())

    intended(hasComponent(ComponentName(intentsTestRule.activity,
RegistrationActivity::class.java)))

    // Then
    assertEquals(null, intentsTestRule.activity.preferences.getUser())
}
```

15.3. Testiranje sustava

Nakon izvršenih integracijskih testova u procesu testiranja prema životnom ciklusu razvoja proizvoda slijedi testiranje sustava. Kako teče životni ciklus razvoja proizvoda tako proizvod biva sve veći i obuhvatniji. Nakon što smo u jediničnim testovima pokrili i ispitali najmanje jedinice koda, te u integracijskim testovima pokrili spajanje određenih jedinica koda u malo veće cjeline odnosno u ovom slučaju funkcionalnost prijave. U testovima sustava slijedi ispitivanje sustava kao cjeline. Glavni cilj navedenih testova je pronaći greške koje jedino mogu biti vidljive kada se cjelokupni sustav testira. Kako aplikacija koja se testira u ovim primjerima nije napravljena za stvarnog kupca, nemoguće je napraviti realnu situaciju koja se događa u pravom svijetu. Prema tome u testiranju sustava pokrit će se testiranje performansi aplikacije prema definiranim zahtjevima na početku kreiranja aplikacije.

15.3.1. Testiranje performansi

Testiranje performansi biti će izvršeno uz pomoć automatiziranog testa izvršenog uz pomoć Firebase Test Lab [40] testne infrastrukture u kojem će se provjeriti performanse aplikacije na tri različita Android uređaja. U obzir će ući tri performanse: iskorištenost procesora, zauzeće radne memorije i broj skinutih podataka putem Interneta. Svaki od uređaja će imati različitu verziju Android operacijskog sustava.

Testni plan

Tablica 16 Testni plan testiranja performansi

Svrha	Ispitati performanse aplikacije na Android uređajima različitih verzija Android operacijskog sustava.
Cilj	Potvrditi kako performanse aplikacije odgovaraju željenim karakteristikama performansi.
Resursi	SportAnalytic aplikacija, Firebase korisnički račun [41], Test Lab Robo test [42], Robo skripta [42]
Zaduženja	Josip Dudaš – izrada plana testiranja Josip Dudaš – izrada testnog slučaja Josip Dudaš – provođenje testiranja
Strategija - aktivnosti	Testiranje sustava, testiranje performansi, automatizirano testiranje

Tablica 17 Skala prihvatljivih performansi

Skala prihvatljivih performansi			
	Poželjno	**Prihvatljivo**	**Neželjeno**
Procesor (%)	< 25	25 > 40	> 40
Memorija (KiB)	< 200 000	200 000 < 300 000	> 300 000
Mreža (B/s)	< 50 000	50 000 < 100 000	> 100 000

Test_1:

Tablica 18 Testni slučaj: Performanse-API_21

Oznaka testnog scenarija	Performanse-1
Testni scenarij	Provjera performansi
Oznaka testnog slučaja	Performanse-API_21
Testni slučaj	Provjera performansi aplikacije na Android verziji 5.0 (API 21)
Preduvjeti	Popis željenih performansi, SportAnalytic aplikacija, Firebase korisnički račun [41], Test Lab Robo test [42], Robo skripta [42]
Testni koraci	1. Preuzeti Robo skriptu sa linka: https://drive.google.com/file/d/1VyiMalG5akR2DVsegp2yV8CMEWd8oSw1/view?usp=sharing 2. Preuzeti aplikaciju sa linka: https://drive.google.com/file/d/1ILoxSZtcNiBE6OP_kRGzzWBLjiWKZadF/view?usp=sharing 3. Prijaviti se na Firebase račun 4. Otići na „Go to console“ 5. Odabrati SportAnalytic projekt 6. Otići na opciju „Test Lab“ u lijevom izborniku 7. Odabrati „Run a test“ 8. Odabrati „Run a Robo test“ 9. Učitati aplikaciju na Firebase TestLab 10. Učitati Robo skriptu na Firebase TestLab 11. Kliknuti „Continue“ 12. Odabrati Android uređaj sa verzijom 5.0 (API level 21) 13. Kliknuti na „Run tests“ 14. Usporediti dobivene rezultate sa željenim performansama

Očekivani rezultat	• Robo test je uspješno izvršen • Nisu pronađene greške u aplikaciji • Performanse odgovaraju definiranim u tablici 17 Skala prihvatljivih performansi
Stvarni rezultat	• Robo test je djelomično izvršen • Aplikacija se srušila, tj. pronađena je greška „java.lang.NullPointerException: Attempt to read from field 'android.app.Activity android.app.ActivityThread$ActivityClientRecord.activity' on a null object reference“ • Kako se test nije izvršio do kraja i kako se pojavila greška prilikom izvođenja Robo skripte test nije prošao • Performanse nije potrebno gledati jer je test pao na 13. koraku koji je potreban kako bi se performanse gledale i uzimale u obzir
Prošao/Pao	Pao

Kako je test pao rezultati performansi se ne mogu uzeti u obzir. No svejedno na sljedećoj slici možete pogledati rezultate postignutih performansi onog dijela testa koji se uspio izvršiti:

Slika 6 Rezultati provjere performansi aplikacije na Android verziji 5.0 (API 21)

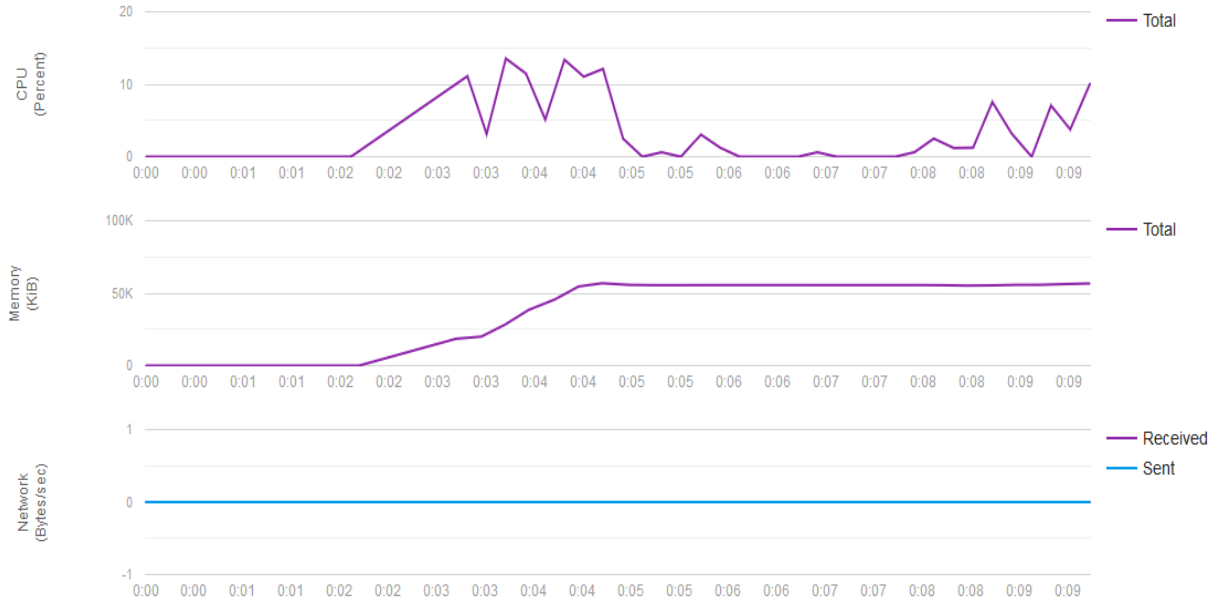
Test_2:

Tablica 19 Testni slučaj: Performanse-API_23

Oznaka testnog scenarija	Performanse-1
Testni scenarij	Provjera performansi
Oznaka testnog slučaja	Performanse-API_21
Testni slučaj	Provjera performansi aplikacije na Android verziji 6.0 (API 23)
Preduvjeti	Popis željenih performansi, SportAnalytic aplikacija, Firebase korisnički račun [41], Test Lab Robo test [42], Robo skripta [42]
Testni koraci	1. Preuzeti Robo skriptu sa linka: https://drive.google.com/file/d/1VyiMalG5akR2DVsegp2yV8CMEWd8oSw1/view?usp=sharing 2. Preuzeti aplikaciju sa linka: https://drive.google.com/file/d/1ILoxSZtcNiBE6OP_kRGzzWBLjiWKZadF/view?usp=sharing 3. Prijaviti se na Firebase račun 4. Otići na „Go to console“ 5. Odabrati SportAnalytic projekt 6. Otići na opciju „Test Lab“ u lijevom izborniku 7. Odabrati „Run a test“ 8. Odabrati „Run a Robo test“ 9. Učitati aplikaciju na Firebase TestLab 10. Učitati Robo skriptu na Firebase TestLab 11. Kliknuti „Continue“ 12. Odabrati Android uređaj sa verzijom 6.0 (API level 23) 13. Kliknuti na „Run tests“ 14. Usporediti dobivene rezultate sa željenim performansama
Očekivani rezultat	• Robo test je uspješno izvršen • Nisu pronađene greške u aplikaciji • Performanse odgovaraju definiranim u tablici 17 Skala prihvatljivih performansi
Stvarni rezultat	• Robo test je djelomično izvršen • Aplikacija se srušila, tj. pronađena je greška „java.lang.NullPointerException: Attempt to read from field 'android.app.Activity

	android.app.ActivityThread$ActivityClientRecord.activity' on a null object reference“ • Kako se test nije izvršio do kraja i kako se pojavila greška prilikom izvođenja Robo skripte test nije prošao • Performanse nije potrebno gledati jer je test pao na 13. koraku koji je potreban kako bi se performanse gledale i uzimale u obzir
Prošao/Pao	Pao

Kako je test pao rezultati performansi se ne mogu uzeti u obzir. No svejedno na sljedećoj slici možete pogledati rezultate postignutih performansi onog dijela testa koji se uspio izvršiti:



Slika 7 Rezultati provjere performansi aplikacije na Android verziji 6.0 (API 23)

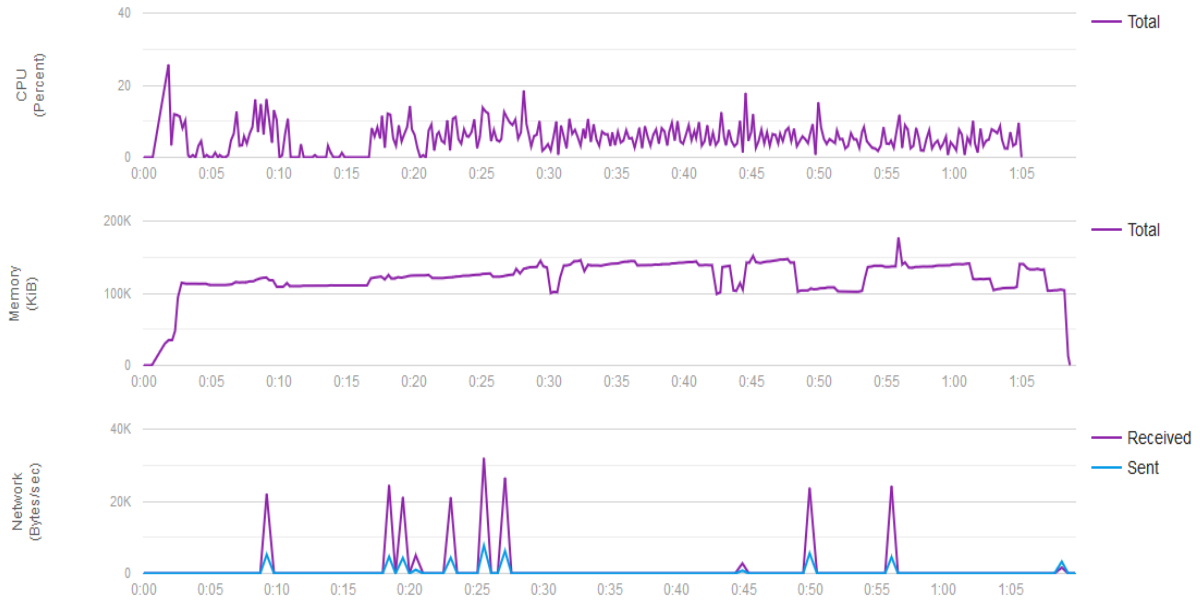
Test_3:

Tablica 20 Testni slučaj: Performanse-API_26

Oznaka testnog scenarija	Performanse-1
Testni scenarij	Provjera performansi
Oznaka testnog slučaja	Performanse-API_21
Testni slučaj	Provjera performansi aplikacije na Android verziji 8.0 (API 26)
Preduvjeti	Popis željenih performansi, SportAnalytic aplikacija, Firebase korisnički račun [41], Test Lab Robo test [42], Robo skripta [42]
Testni koraci	15. Preuzeti Robo skriptu sa linka: https://drive.google.com/file/d/1VyiMalG5akR2DVsegp2yV8CMEWd8oSw1/view?usp=sharing 16. Preuzeti aplikaciju sa linka: https://drive.google.com/file/d/1ILoxSZtcNiBE6OP_kRGzzWBLjiWKZadF/view?usp=sharing 17. Prijaviti se na Firebase račun 18. Otići na „Go to console“ 19. Odabrati SportAnalytic projekt 20. Otići na opciju „Test Lab“ u lijevom izborniku 21. Odabrati „Run a test“ 22. Odabrati „Run a Robo test“ 23. Učitati aplikaciju na Firebase TestLab 24. Učitati Robo skriptu na Firebase TestLab 25. Kliknuti „Continue“ 26. Odabrati Android uređaj sa verzijom 8.0 (API level 26) 27. Kliknuti na „Run tests“ 28. Usporediti dobivene rezultate sa željenim performansama
Očekivani rezultat	• Robo test je uspješno izvršen • Nisu pronađene greške u aplikaciji • Performanse odgovaraju definiranim u tablici 17 Skala prihvatljivih performansi
Stvarni rezultat	• Robo test je uspješno izvršen • Nisu pronađene greške u aplikaciji • Performanse odgovaraju definiranim u tablici 17 Skala prihvatljivih performansi

Prošao/Pao	Prošao

Rezultati performansi:



Slika 8 Slika 7 Rezultati provjere performansi aplikacije na Android verziji 8.0 (API 26)

16. Zaključak

Kroz rad mogli smo zaključiti kako je testiranje programskih proizvoda tema koja je usko povezana sa razvojem programskih proizvoda te bi mogli bi reći da je ona neizostavni dio razvojnog procesa. Testiranje se može povezati sa razvojem bilo kojeg proizvoda na svijetu. Korisnici proizvoda ne bi imali povjerenja u proizvode bez prethodnih testiranja. Uz povjerenje korisnika, testiranjem se utječe na željeni izgled, željene karakteristike i prijeko potrebnu sigurnost proizvoda. Naravno da testiranje nije mađioničarski trik kojim se jednostavno postiže željeni izgled, karakteristike, struktura već testiranje pomaže razvojnom timu da se ispitaju dogovoreni izgled, karakteristike, struktura proizvoda sa napravljenim i tako uoče greške.

Važno je postaviti pravi cilj testiranja koji kaže da je cilj otkriti što veći broj grešaka koje proizvod ima te na taj način umanjiti vjerojatnost pojave istih u krajnjem programskom proizvodu koji se isporučuje kupcu. Nepravilno je postaviti si cilj koji kaže da je testiranjem potrebno potvrditi da proizvod radi na pravilan način. Tim ciljem ćemo si umanjiti vjerojatnost pronalaska grešaka i smanjiti kvalitetu finalnog programa. Testiranjem se ne možemo postići nepostojanje grešaka u finalnom proizvodu, ali možemo smanjiti vjerojatnost pojavljivanja navedenih grešaka. Za niti jedan programski proizvod na svijetu ne možemo reći kako on nema niti jednu grešku, jer proizvodi bez grešaka ne postoje. U primjeru testiranja na testnoj aplikaciji Sport Analytic moguće je vidjeti kako su svi prethodni testovi (jedinični i integracijski) prolazili u redu i nisu pronalazili niti jednu grešku. Ali u zadnjoj kategoriji testova odnosno testova performansi test je otkrio kako se ne može prijaviti u aplikaciju na Android uređajima sa starijom verzijom Android operacijskog sustava. Iz navedenog primjera se može zaključiti kako i u slučaju provedbe određenih testova koji su prošli ne možemo reći kako taj proizvod nema grešaka u svom radu. Navedena situacija bi se pogoršala ukoliko ne bi provodili nikakve testove.

Kroz primjere ali i kroz teoretski dio rada lako je zaključiti kako je testiranje proces koji se sastoji od jako puno pod koraka. Testiranje nije jednokratni i brzi proces već on traje. Iz tog razloga je dobro prvo isplanirati cjelokupni proces testiranja prije samog izvođenja. U procesu planiranja je potrebno uzeti u obzir sve stavke, aktivnosti, resurse, strategije koje će se koristiti u izradi, provedbi i evaluaciji testova. Nakon planiranja je potrebno krenuti sa razradom testnih scenarija i testnih slučajeva koji će se kasnije izvršiti. Prilikom razrade je potrebno voditi računa o svim detaljima. Nakon što su testni slučajevi razrađeni slijedi izvršavanje testova od strane testera. Testeri su osobe bez kojih ne bi bilo moguće provesti testiranja. Iz tog razloga ih je potrebno jako pomno birati. Testeri moraju biti različitih znanja i vještina, jer će se samo tako

postići raznovrsnost prilikom testiranja. I nakon što testeri obave svoj posao testiranja slijedi prikupljanje podataka i usporedba sa očekivanim rezultatima.

Kako testiranje ne bi bilo usmjereno na samo određena područja u svijetu postoje mnogobrojne metode, vrste i tehnike uz pomoć kojih se izvršava testiranje. Na taj se način postiže pokrivenost različitih dijelova programa, od njihove strukture pa sve do korisničkog sučelja. Prilikom osmišljavanja testova potrebno je voditi računa o metodama, vrstama i tehnikama koje će se koristiti jer izborom manje pogodnih metoda, vrsta i tehnika može rezultirati manjem brojem otkrivenih grešaka koje će ući u finalni programski proizvod.

Važna napomena koju većina testera naglašava je rano otkrivanje grešaka. Što je greška otkrivena u kasnijim fazama životnog ciklusa programskog proizvoda to je cijena njenog otklanjanja veća. Iz toga se može zaključiti kako je potrebno što ranije u razvoju programskog proizvoda pokušati naći što je više moguće grešaka kako bi konačni proizvod bio što bolji i kako bi koštao manje.

Popis literature

[1] F. McCown, "A Short History of Computing", Harding UniversityComputer Science Dept, 2013. [Na internetu]. Dostupno: https://www.harding.edu/fmccown/short-history-of-computing.pdf. [Pristupano 27 04 2019].

[2] J. Meerts i D. Graham, „The History of Software Testing“, [Na internetu]. Dostupno: http://www.testingreferences.com/testinghistory.php. [Pristupano 29 04 2019].

[3] G. J. Myers, T. Badgett i C. Sandler, The art of software testing - Third edition, New Jersey: Hoboken, N.J., USA : John Wiley & Sons, 2012.

[4] ASTQB, „American Software Testing Qualifications Board“, [Na internetu]. Dostupno: https://astqb.org/what-is-software-testing/. [Pristupano 29 04 2019].

[5] International Software Testing Board, „Standard Glossary of Terms used in Software“, [Na internetu]. Dostupno: https://astqb.org/resources/glossary-of-software-testing-terms/. [Pristupano 29 04 2019].

[6] V. Kirinić, "TC09: Verifikacija, validacija i testiranje programske opreme", nastavni materijali na predmetu Kvaliteta i mjerenja u informatici [Moodle], Sveučilište u Zegrebu, Fakultet organizacije i informatike, Varaždin, 2018.

[7] AltexSoft, „Quality Assurance, Quality Control and Testing — the Basics of Software Quality Management“, AltexSoft, 2018. [Na internetu]. Dostupno: https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/. [Pristupano 14 07 2019].

[8] S. U. Farooq, Software Testing – Goals, Principles, and Limitations, Department of Computer Sciences University of Kashmir, India, 2010.

[9] S. Reid, „ISO/IEC/IEEE 29119 The New International Software Testing Standards“, Testing Solutions Group, 2013. [Na internetu]. Dostupno: https://www.bcs.org/upload/pdf/sreid-120913.pdf. [Pristupano 10 5 2019].

[10] ISO/IEC, International Standard ISO/IEC 9126, ISO/IEC, 2019.

[11] ISO/IEC, Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, ISO/IEC, 2005.

[12] WG26, „Software testing standard“, [Na internetu]. Dostupno: http://softwaretestingstandard.org/aboutWG26.php. [Pristupano 9 5 2019].

[13] I. O. f. Standardization, „International Organization for Standardization“, ISO, 9 2013. [Na internetu]. Dostupno: https://www.iso.org/standard/45142.html. [Pristupano 10 5 2019].

[14] „5 Latest Software Testing Standards“, testbytes, 4 10 2017. [Na internetu]. Dostupno: https://www.testbytes.net/blog/software-testing-standards/. [Pristupano 10 5 2019].

[15] I. O. f. Standardization, „ISO/IEC/IEEE 29119-4:2015 Software and systems engineering -- Software testing -- Part 4: Test techniques“, International Organization for Standardization, 2 2015. [Na internetu]. Dostupno: https://www.iso.org/standard/60245.html. [Pristupano 10 5 2019].

[16] C. B. Testing, „Moving From Manual to Automated TestingA Tester’s Journey“, [Na internetu]. Dostupno: https://crossbrowsertesting.com/crossbrowsertesting/media/pdfs/moving-from-manual-to-automated-testing.pdf. [Pristupano 12 5 2019].

[17] A. D. F. Joe Fernandes, When to Automate Your Testing (and When Not To), Oracle.

[18] SeleniumHQ, „SeleniumHQ“, [Na internetu]. Dostupno: https://www.seleniumhq.org/about/. [Pristupano 13 5 2019].

[19] M. V. S. G. A. K. R. Jyotsna, „Automated Testing: An Edge Over Manual Software Testing“, International Journal of Trend in Scientific Research and Development, pp. 710-713, 2017.

[20] SoapUI, „SoapUI“, SmartBear, [Na internetu]. Dostupno: https://www.soapui.org/open-source.html. [Pristupano 13 5 2019].

[21] Katalon, „About us“, Katalon, [Na internetu]. Dostupno: https://www.katalon.com/about-us/. [Pristupano 13 5 2019].

[22] A. S. Foundation, „Apache JMeter“, Apache Software Foundation, [Na internetu]. Dostupno: https://jmeter.apache.org/. [Pristupano 13 5 2019].

[23] H. S. A. S. Tilo Linz, Software testing foundations - 4th edition, Rocky Nook, 2014.

[24] tutorialspoint.com, Software testing tutorial, tutorialspoint.com.

[25] A. Jović, M. Horvat, D. Ivošević i N. Frid, Procesi programskog inženjerstva, FER, 2015.

[26] D. QA, „Test Design Techniques: State Transition Testing“, Derisk QA, 29 06 2018. [Na internetu]. Dostupno: http://www.deriskqa.com/Article-State-Transition-Testing.html. [Pristupano 04 06 2019].

[27] V. P. Shivani Acharya, „Bridge between Black Box and White Box – Gray Box“, International Journal of Electronics and Computer Science Engineering, pp. 175-185.

[28] V. Strahonja, „UML Slučajevi korištenja“, nastavni materijali na predmetu Programsko inženjerstvo [Moodle], Varaždin, 2015.

[29] H. Babbar, „Software testing: Techniques and test cases“, International journal of research in computer applications and robotics, pp. 44-53, 03 2017.

[30] D. Bojić i D. Drašković, Testiranje softvera, Beograd: Univerzitet u Beogradu, Elektrotehnički fakultet, Katedra za računarsku tehniku i informatiku, 2012.

[31] Microsoft, „Integration tests in ASP.NET Core“, Microsoft, 25 02 2019. [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-2.2. [Pristupano 04 05 2019].

[32] I. Hooda i R. S. Chhillar, „Software Test Process, Testing Types and Techniques“, International Journal of Computer Applications , 13 01 2015.

[33] A. 2. license, „Using Kotlin for Android Development“, JetBrains and Google, [Na internetu]. Dostupno: https://kotlinlang.org/. [Pristupano 14 07 2019].

[34] S. Consortium, „SQLite“, SQLite Consortium, [Na internetu]. Dostupno: https://www.sqlite.org/index.html. [Pristupano 14 07 2019].

[35] Oracle, „MySQL“, Oracle, [Na internetu]. Dostupno: https://www.mysql.com/. [Pristupano 14 07 2019].

[36] P. Group, „PHP“, PHP Group, [Na internetu]. Dostupno: https://php.net/. [Pristupano 14 07 2019].

[37] G. Developers, „Android Studio“, Google, [Na internetu]. Dostupno: https://developer.android.com/studio. [Pristupano 14 07 2019].

[38] „Mockito“, [Na internetu]. Dostupno: https://site.mockito.org/. [Pristupano 14 07 2019].

[39] JUnit, „JUnit 4“, JUnit, 25 11 2018. [Na internetu]. Dostupno: https://junit.org/junit4/index.html. [Pristupano 14 07 2019].

[40] G. Developers, „Firebase Test Lab“, Google, [Na internetu]. Dostupno: https://firebase.google.com/docs/test-lab. [Pristupano 14 07 2019].

[41] G. Developers, „Firebase“, Google, [Na internetu]. Dostupno: https://firebase.google.com/. [Pristupano 14 07 2019].

[42] G. Developers, „Firebase Test Lab Robo Test“, Google, [Na internetu]. Dostupno: https://firebase.google.com/docs/test-lab/android/robo-ux-test. [Pristupano 14 07 2019].

[43] L. M. I. B. B. Dolores R. Wallace, Reference Information for the Software, DIANE Publishing, 1996..

[44] N. Jenkins, A Software Testing Primer, Creative Common, 2008..

[45] P. C. Jorgensen, Software testing A Craftsman's Approuch Fourth Edition, 6000 Broken Sound Parkway NW, Suite 300, USA: CRC Press, 2014.

Popis slika

Popis tablica

Prilozi

1. „Sport analytic“ android aplikcija (Kotlin) - https://github.com/JosipDudas/sport_analytic_kotlin_android.git
2. Web servisi - https://github.com/JosipDudas/sport_analytic_webservice.git