Prakse testiranja programskih proizvoda

Mijo, Lučić

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:803452

Rights / Prava: Attribution-NonCommercial-ShareAlike 3.0 Unported/Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-05-07**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mijo Lučić

PRAKSE TESTIRANJA PROGRAMSKIH PROIZVODA

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mijo Lučić

Matični broj: 42281/13–I

Studij: Informacijski sustavi

PRAKSE TESTIRANJA PROGRAMSKIH PROIZVODA

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Zlatko Stapić

Varaždin, rujan 2019.

Mijo Lučić

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 potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu su obrađene aktualne prakse, metode i tehnike testiranja programskih proizvoda, pritom se fokusirajući na aplikacije za web. U istraživačkom dijelu rada obrađeno je korištenje odabranih praksi testiranja na mutlimedijskoj aplikaciji za web, koja je rađena pomoću WebRTC biblioteka. Web aplikacija koja je izrađena omogućuje korisnicima korištenje multimedijskih mogućnosti preglednika, kao što je uporaba web kamere, mikrofona, zvučnika i drugih vanjskih ulaznih i izlaznih uređaja. Primjer jednog slučaja korištenja jeste video i audio snimanje putem web kamere, te snimanje ekrana. Takva vrsta aplikacije trebala bi imati podršku raznih preglednika u različitim uvjetima i tehničkim karakteristikama. Iz tih razloga, veoma je važno upotrijebiti što više praksi testiranja web aplikacije. Odabran je agilni model razvoja programskog proizvoda i kao dio tog procesa testirana je aplikacija kroz kontinuiranu integraciju, dostavu i objavu. Također, testirana je podrška funkcionalnosti aplikacije na različitim preglednicima, što nazivamo testiranje interoperabilnosti. Između ostalog, obrađene su unificirane razine testiranja, kao što je jedinično, integracijsko testiranje i testiranje performansi.

Ključne riječi: razvoj; testiranje; web aplikacije; WebRTC, automatsko testiranje;

Sadržaj

Sadržaj iii
1. Uvod 1
2. Metode i tehnike rada 3
3. Osnove testiranja programskih proizvoda 4
3.1. Ciljevi i načela testiranja 4
3.2. Uloge u testiranju 5
3.3. Verifikacija i validacija programskog proizvoda 7
3.4. Metode i tipovi testiranja 8
3.5. Razvoj vođen testiranjem 9
3.6. Praksa ad-hoc testiranja 10
3.7. Životni ciklus programskih proizvoda 11
4. Životni ciklus testiranja programskih proizvoda 12
4.1. Testiranje kao faza u suvremenim modelima razvoja 12
4.1.1. Testiranje u vodopadnom modelu razvoja 13
4.1.2. Testiranje u agilnom modelu razvoja 13
4.1.3. Testiranje u DevOps modelu 14
5. Testiranje u kontekstu osiguranja kvalitete programskih proizvoda 17
5.1.1. Pisanje testnih planova i slučajeva 17
5.2. Testiranje performansi kao čimbenik osiguranja kvalitete 18
5.3. Standardizacija i certifikacija testiranja softvera 20
5.4. Normativne reference testiranja u programskom inženjerstvu 21
6. Razine testiranja 22
6.1. Jedinično testiranje 23
6.1.1. Dobre prakse u jediničnom testiranju 24
6.1.2. Korištenje alata za kreiranje jediničnih testova 25
6.2. Integracijsko testiranje 26
6.2.1. Testiranje web servisa 27
6.2.2. Korištenje popularnih alata za integracijsko testiranje 27
6.3. Testiranje korisničkog sučelja 28

6.3.1. Dimni testovi 29
6.3.2. Alati za UI testiranje 31
7. Testiranje u kontekstu kontinuirane integracije, isporuke i objave 32
7.1. Kontinuirana integracija....32
7.2. Kontinuirana isporuka 33
7.3. Kontinuirana objava 34
7.4. Kontinuirana integracija i dostava aplikacije putem Jenkins-a i drugih srodnih alata .35
8. Implementacija i testiranje WebRTC aplikacije....37
8.1. Pribavljanje korisničkih medija na pregledniku pomoću WebRTC-a 38
8.2. Funkcionalnosti aplikacije „SVEA“....38
8.3. Testiranje interoperabilnosti WebRTC aplikacija 42
8.3.1. Testiranje interoperabilnosti pomoću alata Kite 43
8.4. Kontinuirana integracija i dostava aplikacije „SVEA“ 45
8.5. Jedinično testiranje aplikacije u pregledniku Chrome 49
8.6. Testiranje performansi aplikacije pomoću JMeter alata 50
9. Zaključak 53
Popis literature 54
Popis slika....57
Popis tablica....58

1. Uvod

Svjedoci smo vremena kada su programski proizvodi dio naših svakodnevnih aktivnosti u mnogim aspektima života. Osiguranje kvalitete u modernom razvoju programskih proizvoda ne može bez sustavnog i predanog testiranja. Projektni rizici često su preveliki da bi se dozvolio pristup štednje na aspektu razvoja, kao što je testiranje. Svjedoci smo vremena kada testiranje kao grana unutar IT industrije raste u obliku ulaganja ogledajući se kroz potražnju na burzi rada u razvijenijim državama svijeta. Još uvijek, sama ponuda ne prati potražnju za testerima.

Jedan od mitova kojima su mnogi ljudi u branši skloni, jeste da je kontrola kvalitete jednaka testiranju. Ustvari, testiranje je samo jedna komponenta kontrole kvalitete softvera. Također, mnogi misle da je cilj testiranja otkriti i otkloniti 100% pogrešaka. Činjenica jeste da je cilj otkloniti što veći mogući broj nedostataka uz osiguravanje da program zadovoljava zahtjeve. Otrkivanje i uklanjanje svih nedostataka je nemoguće postići. Međutim, moguće je uložiti veliki napor u koncizno testiranje svih komponenti, njihove integracije i nadalje testiranje prihvatljivosti od strane klijenta.

Ono što mnogi sa vanjske perspektive ne mogu da razumiju, jeste da postoje situacije kada je testiranje veoma težak proces, ponekad i teži od samog kodiranja. Ipak, za testiranje programskih proizvoda katkada je potrebno mnogo tehničkog znanja. Testiranje je rigorozna disciplina koja zahtjeva široku lepezu vještina iz mnogih aspekata života. Također, zabluda je da automatizirani testovi eliminiraju potrebu za ručnim testovima. Ponekad je ručno testiranje neophodno. Situacija u kojoj je došlo do ogromnog nedostatka sustava se često zna pripisati testerima.

Kvaliteta softvera je odgovornost svih članova i dionika uključujući cijeli tim, a posebice u suvremenim metodama razvoja kao što je agilna metoda. Kako se razvija programsko inženjerstvo, odgovornost na procesu testiranja postaje sve bitnija. Tu činjenicu potvrđuje sve veći iznos ukupnog ulaganja u ljude i infrastrukturu koji u nekim korporacijama dostiže jednako procesu razvoja. Velika količina novih modela razvoja i testiranja proizvoda svojevrstan su aspekt mlade struke koja prolazi još uvijek kroz „dječije bolesti“ lutanja i stalnog revidiranja metoda rada.

Osnovni cilj većine projekata jesu zadovoljni korisnici. Voditeljima projekata često se čini kako su klijenti neprecizni u svojim željama. Međutim, često su klijenti mnogo oštriji nakon isporuke proizvoda kada naiđu na nekvalitetan proizvod, a posebice kada su vidljive očite objektivne pogreške u radu sustava. Iz tog aspekta, ulaganje u testiranje programskih

proizvoda još u fazi dizajniranja, je neophodno. Pritom imajući u vidu da je svrha testiranja otkriti i ukloniti što više mogućih pogrešaka.

Cilj ovog završnog rada jeste istražiti, razložiti i pojasniti relevantne metode, tipove i tehnike testiranja u suvremenom razvoju programskih proizvoda. Kako je ovo područje slabo problematizirano u domaćoj IT industriji, dolazimo do potrebe za općim pregledom i analizom testiranja programskih proizvoda. Trendovi su ubrzani i veoma dinamični i samim time zahtjeva od učesnika u razvoju softvera da se dodatno obrazuju glede testiranja, jer sve je jasnije da je taj čimbenik pomalo zapostavljen u akademskim i stručnim radovima.

U ovom radu tematizirane su sve aktualne i relevantne prakse, razine, modele, metode i tehnike testiranja programskih proizvoda. Stoga, rad je podijeljen na nekoliko glavnih cjelina. Na početku rada obrađene su sve teoretske osnove testiranja programskih proizvoda i programskog inženjerstva uopće, što uključuje ciljeve, načela, uloge, metode, tipove i određene prakse testiranja, te životni ciklus programskih proizvoda.

Kada su pojašnjenje sve osnovne kategorizacije, obrađen je životni ciklus testiranja programskih proizvoda u kojem se detaljno pojašnjava testiranje kao faza u suvremenim modelima razvoja, kao što su agilni model, DevOps i vodopadni model.

Kako je testiranje važan čimbenik u osiguranju kvalitete programa, te pisanje testnih slučajeva i planova, performansi i ostalih čimbenika kvalitete skupno su obrađeni u posebnom poglavlju. Nakon toga objašnjene su pojedine razine testiranja metodom „odozdo-prema-dole“, koje čine piramidu testiranja, kao što su jedinično, integracijsko testiranje i testiranje korisničkog sučelja. Svaka od tih razina su obrađene zasebno sa suvremenim presjekom alata i dobrih praksi korištenja tih razina u testiranju.

Suvremeni modeli razvoja imaju fazu u kojoj je testiranje nezaobilazno u kontekstu kontinuirane integracije, isporuke i objave, što je obrađeno u posebnom poglavlju. U završetku rada analizirana je implementacija i testiranje WebRTC multimedijalne aplikacije. Detaljno je objašnjena arhitektura takvih web aplikacija koje koriste vanjske medije poput web kamere i mikrofona, te na koji način se korištenje tog API-ja ponaša na različitim platformama i preglednicima, što je dobar problem za testiranje interoperabilnosti, kontinuiranu integraciju i dostavu.

2. Metode i tehnike rada

Kao osnova za pisanje završnog rada korišten je službeni predložak za pisanje radova koje je propisao fakultet. Za citiranje i referenciranje korišten je APA standardizirani stil. Za pisanje i uređivanje teksta primarno je korišten Microsoft Office Word 2016. Od alata za upravljanje referencama korištene su ugrađene funkcije Word-a i aplikacije Mendeley Desktop.

U istraživačkom dijelu rada prikazana je web aplikacija koja izrađena u svrhu istraživanja testiranja multimedijalnih aplikacija, korištene su moderne tehnologije za izradu web aplikacija, kao što su NodeJS Express, HTML5, CSS3, Bootstrap 4, JavaScript, WebRTC, express, socket.io, mocha i selenium-webdriver. Za pisanje koda korišteni jsu JetBrains WebStorm IDE i Visual Studio Code editor. Dio istraživačkog rada jeste i korištenje sustava za verzioniranje Git na servisu GitHub na način da je kreiran privatni repozitorij. Web aplikacija je testirana na lokalnom okruženju uz pomoć NodeJS lokalnog poslužitelja uz npm upraviteljem paketa. Za potrebe testiranja web aplikacije korišteni su sljedeći preglednici: Google Chrome, Mozilla Firefox, Opera te Microsoft Edge na uređajima sa operacijskom sustavu Windows 10.

Kada je u pitanju preovladajuća praksa pristupa testiranju, preovladavao je pristup testiranja „odozdo-prema-gore“ (*engl. bottom-up*), kojeg karakterizira inkrementalni način integracijskog testiranja, gdje se komponente najniže razine prve testiraju, zatim se testiraju komponente više razine. Od metoda testiranja najviše je problematiziran agilni pristup razvoju programskih proizvoda iz razloga što je najrasprostranjeniji u ovom trenutku. Također, u pojedinim djelovima rada prikazano je testiranje u DevOps okruženju te korištenje kontinuirane integracije i dostave u kontekstu testiranja programskih proizvoda.

3. Osnove testiranja programskih proizvoda

Testiranje programskih proizvoda je jedinstven proces koji se koristi kako bi se ispitao kvalitet, ispitala ispravnost i pokrivenost razvijenog proizvoda. Cilj testiranja jeste da obezbijedi što manji broj grešaka, troškova održavanja i cjelokupne cijene programskog proizvoda. Testiranje pokušava pronaći kritične dijelove softvera, te kao dio životnog ciklusa softvera, koristi se često za ispitivanje i utvrđivanje ispunjenosti zahtjeva koji su definirani od strane klijenta. Testiranje je značajno iz razloga što uvelike može prevenirati neuspjeh misije i krajnjih ciljeva programskog proizvoda.

Svjedoci smo vremena u kojem mnogi stručnjaci iz IT sektora, miješaju mnoge pojmove iz aspekta testiranja programskih proizvoda. Neke od tih pogrešaka su nedovoljna informiranost o značenju različitih pojmova. Mnogi probleme kod rada sustava svode pod pojam pogreška. Pogreška je univerzalniji pojam koji se često odnosi na engleske pojmove *fault, bug, error, defect* itd. Međutim, probleme kod rada sustava možemo svesti na preciznije pojmove.

Prema (Hrgarek, 2013) neki od njih su:

- Greška (*engl. error*) je pogreška koju je prouzrokovao programer prilikom pisanja koda.
- Propust ili unutarnja greška (*engl. fault*) je pogreška koja dovodi do jedne ili više unutarnjih pogrešaka u programu.
- Vanjska pogreška (*engl. failure*) je pogreška koja se pojavljuje kao razlika ponašanja neispravnog i ispravnog programa.
- Nedostatak ili odstupanje (*engl. defect*) je odstupanje od očekivanog rada.

Razumijevanje ovih pojmova je veoma bitna osnova za prepoznavanje nedostataka u sustavu. Zbog kolizije u različitom interpretiranju nedostataka, nastaju mnogi problemi koji za posljedicu imaju pogrešno definiran problem, a samim time i krajnje rješenje problema. Stoga je bitno da svi sudionici znaju razlučiti osnovne pojmove kod testiranja programskih proizvoda.

3.1. Ciljevi i načela testiranja

Padmini u svojoj knjizi (2006) govori o nekoliko činjenica kada je u pitanju programsko inženjerstvo, a koje se između ostalog odnose i na testiranje. Neki od njih je tvrdnja da su najbolji programeri do 28 puta bolji nego loši programeri. Te da je 80% razvoja

programskih proizvoda bazirano na intelektu, što znači da je ključ u kreativnosti ljudi. Također, tvrdi da su pogreške pri specificiranju zahtjeva najskuplje za popravak. Uklanjanje pogrešaka je najduža faza životnog ciklusa. Na kraju tvrdi kako nema jednog najboljeg pristupa pri uklanjanju pogrešaka u programima. Proces saznanja i poboljšanja prakse testiranja je dugotrajan proces koji ipak zahtjeva mnogo resursa u početku, ali svakako ima svoj pozitivan rezultat gledano strateški na duži period.

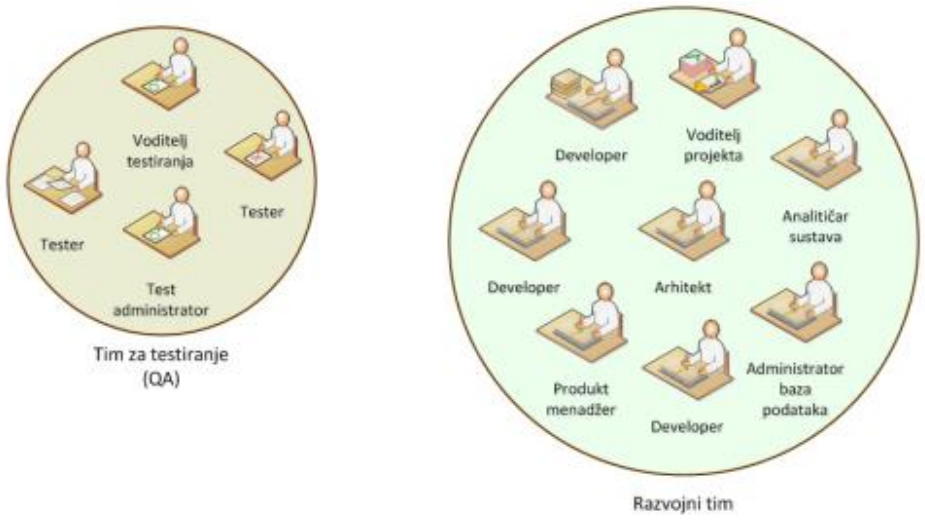
3.2. Uloge u testiranju

Svaki kvalitetan projekat mora imati kvalitetnu podjelu posla na više specijaliziranih radnika koji će svoj dio posla odraditi kvalitetno. Specijalizacija odrađivanja zadataka kada je u pitanju osiguranje kvalitete je često nužna kako bi krajnji cilj bio ispunjen. Stoga, potrebno je razdvojiti testiranje programskih proizvoda u više uloga.

U teoriji organiziranje uloga u testiranju možemo podjeliti kako navodi Hrgarek u svojoj prezentaciji (2013) na:

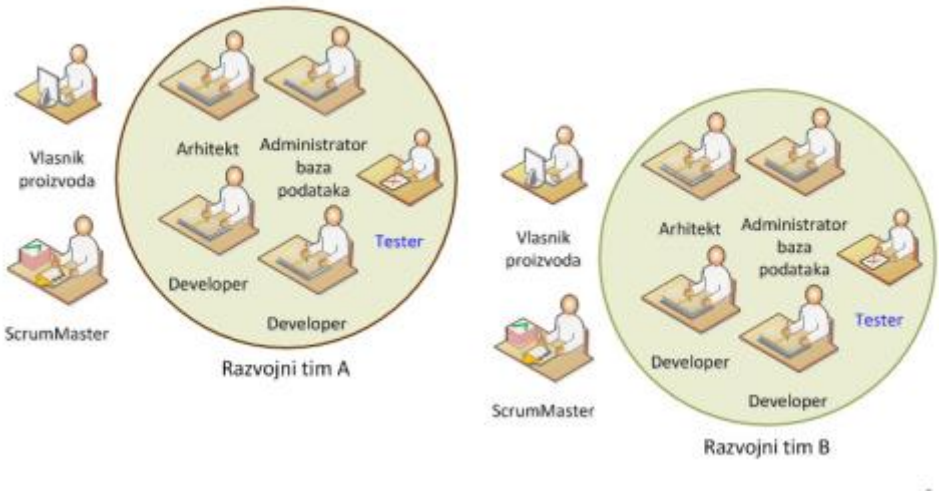
- voditelja testiranja (*engl. test manager/leader*),
- dizajnera/analitičara testova (*engl. test designer/analyst*),
- automatizera testova (*engl. test automatization engineer*),
- administratora testova (*engl. test administrator*) i
- profesionalnog testera (*engl. software tester/quality engineer*).

U ovom radu bazično će biti prikazana perspektiva dvije općenite neformalne uloge u testiranju programskih proizvoda, a to su testeri i programeri. Na slici 1 vidimo klasičan pristup organizacije testiranja.



Slika 1. Klasični pristup organizacije testiranja (Hrgarek, 2013)

U agilnom pristupu razvoju programskih proizvoda poznatom pod nazivom *Scrum*, tester je dio tima za razvijanje kao specijalizirani stručnjak za tu fazu razvoja. U posljednje vrijeme je sve veća potražnja za radno mjesto pod imenom „DevOps inženjer“ što predstavlja radnika koji ima kompetencije da upravlja cjelokupnim procesom u DevOps modelu razvoja. Kako vidimo na slici 2, ta uloga podrazumijeva široki spektar znanja u oblasti sistemskog i softverskog inženjerstva, a koja pokriva kontinuiranu integraciju, dostavu, objavu, upravljanje bazama podataka, resursima itd.



Slika 2. Organizacija testiranja u agilnom pristupu – SCRUM (Hrgarek, 2013)

Testeri se prema nazivlju kompetencija često nazivaju kao osiguravatelji kvalitete (*engl. Quality Assurance*). U drugim inženjerskim strukama često se koristi ovaj univerzalni naziv. U nekim strukama ove poslovne pozicije su višestruko bolje plaćenje, no to se mijenja i polako postaje praksa i u IT struci.

Programeri imaju svoju značajnu ulogu u testiranju programskih proizvoda. Od njih ovisi mnogo. Najprije, programeri pišu jedinične testove. Njihov zadatak jeste da naprave na osnovu testnog slučaja test koji će nedvosmisleno pokazati rezultat da li je programska jedinica zadovoljila uvjete. Samo funkcioniranje te programske jedinice najbolje poznaje onaj koji je kreirao. Stoga, uloga programera jeste da testira programski kod.

Suradnja među različitim ulogama je neminovna, posebice u veoma dinamičnim radnim okruženjima. Kako se često preklapaju paradigme razmišljanja radi prirode posla, nužno je voditi računa o objektivnom pristupu viđenja problema, gdje uloge trebaju razumijeti pristup i znanje drugoga. Uloga voditelja projekata, administratora, arhitekata ili *Scrum mastera* jeste u tome da koordiniraju odnosima i zadacima među različitim ulogama u testiranju softvera.

3.3. Verifikacija i validacija programskog proizvoda

Jedni od često korištenih pojmova koji mnoge čine zbunjenima kada ih vide zajedno, jesu verifikacija i validacija. U ovoj industriji često su predmeti rasprave i dodatnih definiranja i pojašnjenja. Validaciju na našem jeziku možemo nazivati i provjera ispravnosti ili ocjena valjanosti, dok verifikaciju možemo nazivati ovjera.

U tablici 1 uspoređene su verifikacija i validacija kroz definiciju, pojašnjenje i aktivnosti za provođenje.

Kriteriji	**Verifikacija**	**Validacija**
Definicija	„Potvrda kroz ispitivanje i pribavljanje objektivnih dokaza da li su ispunjeni specificirani zahtjevi.“ (ISTQB, 2011, str. 57)	„Potvrda kroz ispitivanje i pribavljanje objektivnog dokaza da su ispunjeni zahtjevi za određenu primjenu ili aplikaciju.“ (ISTQB, 2011, str. 57)
Pojašnjenje	Proces vrednovanja softvera tijekom ili na kraju procesa razvoja, kako bi se utvrdilo zadovoljava li navedene poslovne zahtjeve.	Pitanje koje treba postaviti jeste: „Kreiramo li mi pravi proizvod? Što znači da se ispituje da li proizvod zadovoljava potrebe korisnika i da li su zahtjevi ispravno definirani.
Aktivnosti	Pregled, prolazak i inspekcija	Testiranje

Tablica 1. Razlike između verifikacije i validacije

Verifikacija omogućava da programski proizvod zadovolji propisane zahtjeve koji su oblikovani kroz životni ciklus proizvoda. Validacija je zadovoljenje zahtjeva kupaca na kraju životnog ciklusa proizvoda. Dobra praksa testiranja je koristiti i verifikaciju i validaciju u procesu testiranja programskog proizvoda. Verifikacija je ta koja osigurava kvalitet i održavanje. U tom procesu softver mora pravilno izvršavati sve funkcije tako da neštete sustavu u smislu pogoršanja stabilnosti sustava ili okidanja dodatnih problema. Verifikacija omogućava interoperabilnost među više različitih sekcija u dokumentaciji. Prema (ISTQB, 2007, str. 29), interioperabilnost je sposobnost programskog proizvoda da surađuje s jednim ili više specificiranih komponenti ili sustava.

Ispravke pogrešaka u sustavu mogu koštati do desetinama puta više za fazu izvršavanja i održavanja nego u fazi dizajniranja. To je veoma česti slučaj u testiranju. Roy u svom članku (2019) pojašnjava kako verifikacija povećava troškove razvoja, ali kada se uzme u obzir cjelokupni životni ciklus proizvoda, onda ona zapravo smanjuje troškove.

3.4. Metode i tipovi testiranja

Metode crne, bijele i sve kutije široko su poznate u raznim industrijama, a posebno u teoriji sustava i informacije. Kako su ovo široko upotrebljivi pojmovi, potrebno je pobliže pojasniti značenje ove metode u testiranju softvera. Ove metode možemo rasporediti kroz kategorizaciju dinamičnosti tako da metode crne i bijele kutije spadaju u dinamičko testiranje, dok u statičko spadaju metode neformalog pregleda, tehničkog pregleda i inspekcije valjanosti.

Metode testiranja koje su poznate u IT industriji prema (Software Testing Fundamentals, 2017) su:

- metoda crne kutije,
- metoda bijele kutije,
- metoda sive kutije i
- metoda ad-hoc testiranja.

Metoda crne kutije je testiranje funkcionalnih ili nefunkcionalnih karakteristika bez uzimanja u obzir unutarnje strukture sustava ili komponenti. Dizajn odnosno tehnika oblikovanja testova crne kutije je procedura pomoću koje možemo izvoditi testne slučajeve, koji se temelje na analizi specifikacije komponente ili sustava bez uzimanja u obzir unutarnje strukture.

Metoda bijele kutije je testiranje zasnovano na analizi unutarnje strukture podataka i funkcije komponente ili sustava. Tehnika oblikovanja testova bijele kutije je procedura za izvođenje i odabir testnih slučajeva koji se baziraju na analizi unutarnje strukture. Ovu metodu uloga programera često može bolje oblikovati, jer poznaje dubinsko funkcioniranje komponenti odnosno njen algoritam i strukturu podataka.

Metoda sive kutije kombinira metode crne i bijele kutije. Kao što je navedeno, u metodu crne kutije unutarnja struktura komponenti je nepoznata testeru, dok u metodi bijele kutije testeru je poznata unutarnja struktura komponente ili sustava. To ustvari uključuje pristup unutarnjim strukturama podataka i algoritmima u svrhu dizajniranja testnih slučajeva,

ali i testiranje na razini klijenta, odnosno crne kutije. Ova metoda omogućuje djelomični uvid u dubinsku strukturu programa.

Tipovi testiranja prema (Software Testing Fundamentals, 2017):

- Dimno testiranje poznato kao i kreiranje verifikacijskog testiranja (*engl. Build Verification Testing*) je vrsta testiranja koja se sastoji od velikog skupa testova koji imaju za cilj osigurati rad najvažnijih funkcija sustava. Rezultati ovog tipa testiranja koriste se za odlučivanje da li je paket za objavu dovoljno stabilan za daljnje testiranje.
- Funkcionalno testiranje je tip testiranja softvera kojim se sustav testira prema specifikaciji funkcionalnosti. Te funkcionalnosti se testiraju dodavanjem ulaznih podataka i ispitivanjem izlaza. Ovaj tip osigrava da aplikacija primjereno ispunjava specificirane zahtjeve. Bavi se isključivo rezultatima obrade zahtjeva, ne i načinom obrade.
- Testiranje upotrebljivosti (*engl. usability testing*) je vrsta ispitivanja koja se vrši s aspekta krajnjeg korisnika kako bi se utvrdilo je li sustav upotrebljiv.
- Testiranje sigurnosti namjerava otkriti ranjivosti sustava i kritične točke, te utvrditi da li su resursi i podaci zaštićeni od mogućih uljeza.
- Testiranje performansi je tip testiranja kojim se želi utvrditi uspješnost sustava u smislu stabilnosti i odaziva pri određenom opterećenju rada sustava.
- Regresijsko testiranje je tip kojim se želi osigurati da nove funkcionalnosti, poboljšanja i promjene bilo kojeg tipa, ne utječu negativno na rad i stabilnost sustava.
- Testiranje usklađenosti (*engl. compliance testing*) je vrsta utvrđivanja usklađenosti sustava sa unutarnjim ili vanjskim standardima.

Kroz rad u sljedećim odjeljcima biti će detaljno pojašnjeni neki od navedenih tipova i metoda testiranja kroz primjere, te kroz različite principe testiranja koje su navedeni u prethodnim odjeljcima.

3.5. Razvoj vođen testiranjem

Razvoj vođen testiranjem ili razvoj usmjeravan testiranjem (*engl. test-driven development*) je je način razvoja programskih proizvoda gdje se testni slučajevi razvijaju i automatiziraju prije nego je razvijena aplikacija (u daljnem tekstu TDD). Primarni cilj ove tehnike razvoja jeste fokusiranje na specificiranju, a ne validaciji. Taj način tjera nas da dubinski razmislimo o zahtjevima i dizajnu sustava prije nego što napišemo konkretan kod.

TDD se oslanja na specifikaciju. Kako govori Werner takav jedan pristup ima za rezultat čistiji kod (2019), koji ima svoju svrhu, podlogu i osnovu na kojoj postoji.

TDD je tehnika u razvoju programskog proizvoda koja počinje sa pisanjem testova i koda prilagođenom onoliko koliko je potrebno da definirani testovi prođu validaciju. Prednost TDD-a jeste u tome da pruža brzu povratnu informaciju o ispravnosti proizvoda, ali može dovesti i do lažnog osjećaja sigurnosti. Još uvijek ne postoje studije koje pokazuju efektivnost ove tehnike, ali mnoge studije se slažu s time da vodi do manje neispravnih dijelova koda. Ova tehnika tjera programere ka pisanju jednostavnijeg i preciznijeg koda kako bi se postigla 100% testna pokrivenost (*engl. test coverage*). Testna pokrivenost ili analiza pokrivenosti je mjera postignute pokrivenosti određene jedinice tijekom izvršenja testa prema predodređenim kriterijima, kako bi se odredilo da li je potrebno dodatno testiranje i ukoliko jest, koji testni slučajevi su potrebni.

3.6. Praksa ad-hoc testiranja

Ad-hoc testiranje poznato i kao nasumično testiranje (*engl. monkey testing),* je metoda testiranja koja nema formalne karakteristike, kao što su planiranje i dokumentacija. Ova metoda nema formalne pripreme, odnosno ne primjenjuju se priznate tehnike oblikovanja testova i nema očekivanih rezultata. Aktivnosti provedbe testiranja vođene su vlastitom voljom onoga koji testira programskih proizvod.

Tester improvizira korake koje proizvoljno izvršava, često vođen trenutnim instiktom. Gledano sa formalnog aspekta, ova metoda ima mnoge nedostatke. Najviše iz razloga što nedostaju definirani testni slučajevi. No, ova metoda može iznenaditi sa rezultatima. Poznato je da čovjek svojim nasumičnim pristupom i vođen instiktom može otrkiti nedostatke koji se formalnim metodama teže pronalaze.

Navedena metoda često se koristi kod testiranja prihvaćanjem (*engl. acceptance testing*). Kriteriji prihvaćanja su izlazni kriteriji koje komponenta mora zadovoljiti da bi projekat u konačnici bio prihvaćen od strane klijenta ili krajnjih korisnika. Moguće je da klijenti traže izvještaje. Neformalni izvještaji se ne temelje na propisanoj proceduri. U pristupu „odozdo-prema-gore“ testiranje prihvatljivosti može biti na vrhu piramide kao posljednji čimbenik ostvarenja kvalitete.

Bez obzira na činjenicu da je zajednica godinama uspjela uspostaviti unificirane modele testiranja, *ad-hoc* testiranje je sveprisutno u praksi malih i srednje važnih projekata. Naime, mnoge organizacije ne pridavaju mnogo pažnje i resursa testiranju, jer ono može biti veoma skupo ukomponirano u neke od ustaljenih modela poslovanja. Tako da ovaj pristup

omogućuje brzi prijelaz kroz ovaj segment životnog ciklusa proizvoda. No, koliko god programsko inženjerstvo bude napredovalo u smislu metodologije i prakse testiranja, uvijek će biti primjene ad-hoc testiranja, jer ono ima svoje praktične prednosti.

Testiranje u paru (*engl. pair testing*) slično kao programiranje u paru karakterizira dvije osobe koji mogu biti različitih uloga u procesu, koji rade zajedno s ciljem pronalaska pogreški. Obično se radi na jednom računalu uz praćenje i osmišljavanje akcija za pronalazak greški. Ova tehnika je široko raspostranjena u svim tipovima organizacija, jer je veoma učinkovita u situacijama kada jedna osoba teško pronalazi grešku, jer gleda isključivo iz svoje perspektive. Programiranje i testiranje u paru pruža beneficije pametnijeg i bržeg pronalaska greški u sustavu. Kombinacija uloga može biti između testera i programera ili između testera i krajnjih klijenata. Kada se radi o više od dvije osobe, onda to nazivamo timsko testiranje (*engl. buddy testing*). Pravila su slična, s tim da ova praksa *ad-hoc* testiranja zahtjeva više kontrolnih jedinica odnosno računala u procesu otrkivanja pogrešaka.

3.7. Životni ciklus programskih proizvoda

Životni ciklus programskih proizvoda (*engl. software development life cycle*) ili samo proces razvoja programskog proizvoda definira faze u kreiranju softvera. Bazzana u svojoj knjizi navodi (2015) kako životni ciklus programskih proizvoda varira od jednog do drugog modela ovisno o vrsti modela razvoja softvera. Neki od modela su vodopadni model, spiralni model, model iterativnog i inkrementalnog razvoja, te model agilnog razvoja.

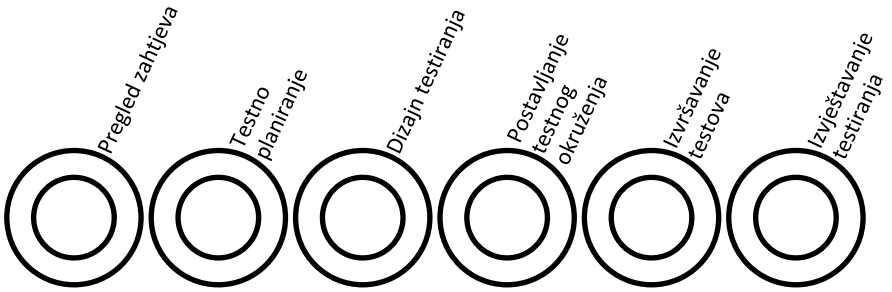
Modeli se razvijaju sa vremenom. S obzirom na trendove u programskom inženjerstvu, neki modeli koriste se u većoj mjeri zbog toga što se stalno povećavaju zahtjevi za sve većim brojem naprednih sustava u kojem testiranje dobija sve važniju ulogu. Iz tih razloga, modeli koji se više baziraju na ranom otkrivanju pogrešaka imaju prednost u modernom razvoju softvera. Najaktualniji model je agilni razvoj programskih proizvoda prema knjizi Balakrishnana o trendovima u testiranju programskih proizvoda (2017).

Kako bi primjenili dobre prakse testiranja, potrebno je najprije odabrati adekvatan model razvoja. On uvjetuje koji elementi testiranja će biti bolje odrađeni i u kojim fazama razvoja. Životni ciklus proizvoda sve više ovisi o stubovima kao što je testiranje. Dugoročno gledano, samo čvrsti i kvalitetni programi pokazuju isplativost i ispunjavaju ciljeve i misiju programskog proizvoda.

4. Životni ciklus testiranja programskih proizvoda

Životni ciklus testiranja programskih proizvoda (*engl. software testing life cycle*) definira različite faze testiranja softvera. Budući da je testiranje standardizirano, kao što razvoj softvera ima životni ciklus, testiranje također ima svoj životni ciklus, koji se ogleda kroz nekoliko faza.

Na slici 3 prikazane su faze životnog ciklusa testiranja programskih proizvoda.



Slika 3. Životni ciklus testiranja programskih proizvoda (Software Testing Fundamentals, 2017).

Kod faze pregleda zahtjeva vrše se aktivnosti pregleda i analize zahtjeva (*engl. review*). Tu se ispituje osnovna ideja dizajna sustava. Dok se testno planiranje vrši kroz dizajniranje testne specifikacije koja definira točan i precizan plan budućeg testiranja, u trećoj fazi se detaljiziraju testovi na temelju prethodno definiranih zahtjeva. U sljedećoj fazi postavlja se testno okruženje i postavljaju se potrebne pretpostavke za repliciranje ponašanja produkcijskog okruženja. Nakon toga slijedi faza izvršavanja svih testova, a zatim slijedi praćenje dnevnika rada kako bi se sastavili zadaci za poporavku tih problema.

Svaka od faza ima svoju važnost u procesu i reflektira stvarne potrebe svakog testiranja. Bitno je voditi računa o izvršenju svake od navedenih faza u životnom ciklusu testiranja. Dakako, faze životnog ciklusa testiranja mogu varirati u odnosu na prakse koje se koriste.

4.1. Testiranje kao faza u suvremenim modelima razvoja

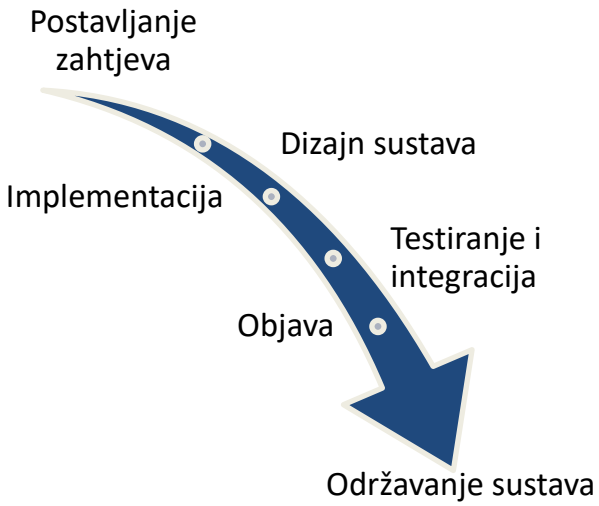
Kako testiranje softvera sve više dobija na važnosti, shodno tome softverska industrija stvara nove modele razvoja poslovnih procesa u kojima testiranje predstavlja bitnu etapu u životnom ciklusu proizvoda. Jedna tako dinamična industrija brzo stvara i širi nove

modele. Šira zajednica programskih inženjera kroz nekoliko desetljeća je uspjela definirati i promovirati nekoliko modela, od kojih se ističu modeli koje su u nastavku obrađene zasebno.

4.1.1. Testiranje u vodopadnom modelu razvoja

Vodopadni model (*engl. watterfall model*) je model životnog ciklusa programskog proizvoda u kojem svaka faza mora biti dovršena prije nego se započne slijedeća faza. Ovaj model spada u red najranijih modela koji se uveliko primjenjuje u ovoj industriji.

Faze se odvijaju linearno tako da nema preskakanja faza, što možemo vidjeti na slici 4. Ovaj model možemo okarakterizirati kao linearno-sekvencijalni model životnog ciklusa proizvoda. U ovom modelu tipično je da izlaz jedne faze djeluje kao ulaz u sljedeću fazu.



Slika 4. Vodopadni model životnog ciklusa proizvoda (autorski rad)

Testiranje i integracija kao faza u ovom procesu slijedi nakon završetka implementacije odnoso samog razvoja programa. Sve funkcionalne i nefunkcionalne jedinice sustava trebaju se integrirati u funkcionalnu cjelinu nakon njihovog pojedinačnog testiranja. U ovoj fazi moguće je objaviti integraciju komponenti na testno okruženje kako bi se testiralo ponašanje sustava. Nakon ove faze slijedi faza produkcijske objave. Posljednja etapa je svakako održavanje sustava koje ne treba biti zapostavljeno u cjelokupnom ciklusu.

4.1.2. Testiranje u agilnom modelu razvoja

Agilni model je kombinacija iterativnog i inkrementalnog procesnog modela sa fokusom na prilagodljivosti i skalabilnosti procesa. Radi se o modelu koji brzo isporučuje verziju svoga proizvoda, te ga kontinuirano poboljšava i te male promjene objavljuje. Prednost je u tome što krajnji korisnici imaju veoma brzo radnu verziju programa, tako da i

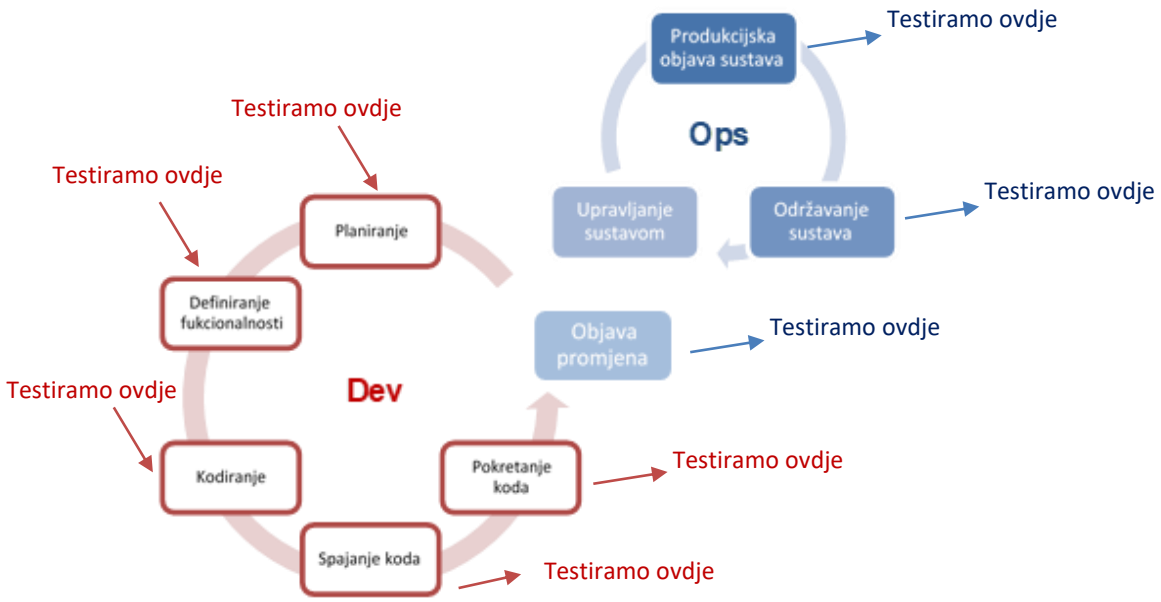
oni učestvuju neprekidno u procesu stvaranja. Cilj agilne metode je svesti razvoj i održavanje u male inkrementalne iteracije. Postoje razna vremenska određenja kada je u pitanju trajanje jedne iteracije. Ono se obično vremenski kreće od jednog do tri tjedna. Svaka od tih iteracija uključuje rad različitih uloga sa različitim specijalizacijama razvoja. Radi se o suvremenom modelu koji je veoma popularan kod svih vrsta IT organizacija. Svaka iteracija sadrži faze planiranja, analize zahtjeva, dizajna, kodiranja, testiranja i objave.

4.1.3. Testiranje u DevOps modelu

Ghahrai u svom članku (2018) pojašnjava da je DevOps novi pojam u testiranju koji postaje veliki trend u posljednih nekoliko godina. DevOps je u suštini udruživanje programerskih i sistemskih praksi u razvoju softvera.

Prema (Clokie, 2017) na neki način ovaj model ide korak naprijed naspram agilnog modela zato što približava aktivnosti objave i implementacije onima u razvoju i testiranju programskog proizvoda. To u praksi znači da je razvojni tim odgovoran za razvoj, testiranje i objavu i održavanje proizvoda kojeg stvaraju. DevOps je koncept koji je veoma brzo postao česta tema među programskim inženjerima i ubrzo postao veoma korištena metoda. Broj konferencija i seminara na ovu temu u posljednjih nekoliko godina potvrđuje važnost ovog modela i njegovo ubrzano širenje.

Testiranje u DevOps modelu zatvara cijeli ciklus razvoja softvera i životni ciklus objave i isporuke, što možemo zaključiti posmatrajući sliku 5. Testeri se više ne fokusiraju samo na funkcionalno testiranje i verifikaciju. DevOps proširuje zadatke testerima i programerima u razvoju. Testeri su uključeni u testiranje perfomansi, sigurnosti sustava, te imaju zadatak da prate analitiku perfomansi, dnevnike rada, dnevnike pogreški itd.

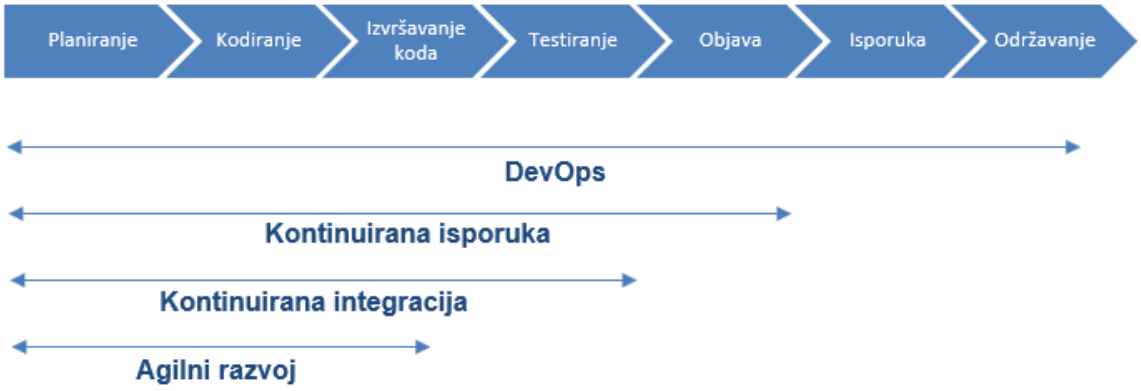


Slika 5. Kontinuirano testiranje u DevOps modelu (Isaac, 2018)

Tester Dan Ashby je u svom blogu za testiranje u DevOps (2016) napisao: „Možete vidjeti da se ljudi bore shvatiti gdje se testiranje uklapa u model koji ga uopće ne spominje. Za mene se testiranje uklapa u svaku etapu u ovom modelu."

Prednosti ovog modela u odnosu na druge se očituju u nekoliko parametara, a neki od njih su poboljšana kvaliteta aplikacije i perfomansi, veći prihodi i manji utrošak vremena u poboljšanju resursa. Ono što dokazuju dosadašnje prakse velikih korporacija koje koriste ovaj model, jeste da se DevOps modelom smanjuje broj projekata koji prelaze inicijalne rokove projekta.

Koliko ovaj model pokriva širok dijapazon razvoja jednog proizvoda, slikovito se može vidjeti na slici 6.



Slika 6. Usporedba etapa razvoja kod suvremenih modela (Shanmugam, 2019)

Zadaće u radu DevOps-a možemo podijeliti po nazivu na zadaće „dev-a“ i „ops-a“. „Dev“ obično ima zadatak brinuti se o perfomansama sustava, tako da je njegova zadaća vršiti kontinuirano testiranje perfomansi. On ima pristup svim elementima sustava, kao što je lokacija objavljene aplikacije, baza podataka, logovi, eksterni servisi itd. Također, njegovo zaduženje jeste stvarati i predočiti analitiku klijentima kako bi dobili pravovremene informacije o funkcioniranju sustava. „Dev“ bi trebao brinuti o kvaliteti koda na način da prati „urednost“ koda i sve drugo što ga čini kvalitetnim. Vodi računa da se kod pravilno komentira, te dokumentira u tehničkoj dokumentaciji. On se na neki način bavi i otklanjanjem pogrešaka u kodu.

Zadaća „ops“ uloge kod ovog modela jeste u tome da brine o dostupnosti aplikacije svim potrebnim korisnicima na pravilan način. Zadužen je da se bavi osnovnim stavkama optimizacije za tražilice (*engl. Search Engine Optimization*) i drugim segmentima digitalnog marketinga. Testiranje perfomansi je zadaća obje strane. Treba voditi računa o popunjenosti memorije, perfomansama procesora na poslužiteljima, te drugim parametrima perfomansi sustava. „Ops“ se bavi analitikom performansi te na neki način prati razvoj kroz vrijeme. To mu pomaže da na vrijeme može reagirati i donositi bitne odluke.

5. Testiranje u kontekstu osiguranja kvalitete programskih proizvoda

Ispitivanje ili kontrola kvalitete programskih proizvoda (*engl. Software Quality Control*) je skup aktivnosti za osiguravanje kvalitete u programskih proizvodima. Kako kaže (Banjavčić, 2017), njen cilj je osigurati da programski proizvodi zadovoljavaju sve zahtjeve i specifikacije.

Kontrola kvalitete programskih proizvoda uključuje dvije glavne aktivnosti, a to su pregledi (*engl. reviews*) i testiranje. Aktivnosti testiranja provode se kroz razine, kao što su jedinično, integracijsko, sustavsko i testiranje prihvatljivosti.

Pregledi uključuju niz podaktivnosti, a ključne su:

- pregled zahtjeva (*engl. requirement review*),
- dizajn pregleda (*engl. design review*),
- pregled koda (*engl. code review*),
- pregled plana razmjene (*engl. deployment plan review*),
- pregled testnog plana (*engl. test plan review*) i
- pregled testnih slučajeva (*engl. test cases review*).

Na web stranici (Software Testing Fundamentals, 2017) možemo pročitati kako se kontrola kvalitete programskog proizvoda mjeri na više načina. Mjerenje pogrešaka (*engl. defect age*) u jedinici vremena i po fazama. Gustoća pogrešaka (*engl. defect density*) je broj potvrđenih nedostataka otkrivenih u komponentama tijekom određenog razdoblja razvoja podijeljen s veličinom komponente. Učinkovitost otkrivanja pogrešaka (*engl. defect detection efficiency*) je broj nedostataka otkrivenih tijekom faze otkrivanja, podijeljen s ukupnim brojem otrkivenih pogrešaka. Trošak kvalitete (*engl. cost of quality*) je mjera koja zbraja trošak kontrole kvalitete i trošak neuspjeha kontrole. Drugim riječima, sažima troškove vezane za prevenciju i otkrivanje nedostataka i troškove zbog kojih se pojavljuju pogreške.

5.1.1. Pisanje testnih planova i slučajeva

Za svakog testera bitno je znati na koji način pisati testne planove i testne slučajeve (*engl. test cases*), koji su bazična podloga za uspješan proces testiranja programskog proizvoda. Sveobuhvatan proces testiranja ne može bez definiranja testnih slučajeva. Proces razvijanja testnih slučajeva može pomoći u pronalaženju problema i nedostataka u definiranim zahtjevima ili dizajnu aplikacije. Format testnih slučajeva obično definiraju alati

koji se koriste za upravljanje testiranja. Stoga, postoje varijacije kod procedura za izvršavanje testnih slučajeva.

Specifikacija testnih slučajeva je dokument koji specificira skup testnih slučajeva, a sadrži ciljeve, ulaze, izlaze, akcije, očekivane rezultate, preduvjete i tokene za izvršenje pojedine jedinice. Ta pojedina jedinica testnog slučaja je skup ulaznih vrijednosti, preduvjeta za izvršavanje, očekivanih rezultata, uvjeta, petlji, spajanja i drugih okolnosti testiranja jedne funkcionalnosti komponente. Testni slučaj je ustvari skup uvjeta i varijabli pod kojima tester određuje da li komponenta zadovoljava zahtjeve i ispunjava zahtjeve rada. Proces izrade i ažuriranja testnih slučajeva pomaže pri pronalaženju problema u definiranim zahtjevima ili dizajniranju aplikacije.

Testni plan je dokument koji opisuje opseg i aktivnosti testiranja softvera. To je osnova za formalno testiranje bilo kojeg softvera. Razlikujemo dva tipa testnih planova, master test plan se odnosi na višestruke razine testiranja i testni plan faza koji opisuje jednu fazu testiranja. Radi se dokumentu koji opisuje doseg, pristup, resurse i raspored planiranih aktivnosti testiranja. Kako navodi Software Testing Fundamentals na svojoj web stranici (2017) testni plan specificira detalje i pojedine elemente testiranja, svojstva, varijable, uloge izvršenja, ulazne i izlazne kriterije itd.

Testni plan možemo definirati na nivou svake razine, koje poznajemo kao jedinično, integracijsko i testiranje prihvatljivosti, odnosno testiranje korisničkog sučelja. Tako je testni plan poželjno praviti za svaku od navedenih razina.

5.2. Testiranje performansi kao čimbenik osiguranja kvalitete

Ispitivanje performansi je vrsta testiranja koja pokušava odrediti i prikazati parametre stabilnosti i kvalitete odaziva performansi sustava. Performanse znaju veoma često zadati „glavobolju“ administratorima sustava ukoliko nisu vršili pravovremeno i učestala testiranja performansi, osobito onda kada se čini da je sustav stabilan i da obavlja svoje zadatke uredno.

Najvažnije za sustav jeste da su performanse sustava skalabilne. To je garant da će sustav raditi na dugi period prema strategiji životnog ciklusa softvera. Također, jedan od bitnih principa jeste ne oslanjati se samo na neke alate. Potrebno je osigurati zamjenska rješenja i drugačije algoritamske postavke za testiranje performansi sustava. Uvijek treba

imati alternativno praćenje perfomansi. Prosječni rezultati mogu biti najispravniji pokazatelj realnog stanja.

Kod ovog tipa testiranja nužno je postaviti okolinu za testiranje što sličnijom produkcijskom okruženju. Testno okruženje mora u svakom segmentu simulirati okolinu produkcije. Također, treba osigurati da se sve promjene na produkciji, pa čak i one koje su označene kao male dorade i ispravke problema, treba najprije isprobati na testnom okruženju. Testna okruženja mogu imati više svojih instanci s obzirom na to da produkcijski sustav može biti modularan.

Postoji nekoliko tipova testiranja performansi, a nekoliko veoma važnih su:

- testiranje opterećenja (*engl. load testing*),
- testiranje otpornosti (*engl. stress testing*),
- testiranje izdržljivosti (*engl. endurance testing*) i
- testiranje učinka (*engl. spike testing*).

Testiranje opterećenja je vrsta testiranja učinkovitosti koja evaulira ponašanje komponente ili sustava s povećanim opterećenjem. To se može simulirati kroz povećani broj paralelnih korisnika koji aktivno koriste komponentu da bi se utvrdilo opterećenje koje može podnijeti ta komponenta ili sustav.

Testiranje otpornosti ili testiranje stresa je tip testiranja perfomansi koji se provodi kako bi se procijenilo ponašanje sustava u cjelini prema predviđenim granicama opterećenja. Klasični primjer ovakvog tipa testa je premještaj kompletnog sustava na druge fizičke lokacije, gdje se treba ispitati rad pri promjeni fizičkog okruženja.

S druge strane, testiranje izdržljivosti je vrsta testiranja perfomansi koja se provodi kako bi se procijenilo ponašanje sustava kada se postiže veliko radno opterećenje sustava na duži period i jačim intezitetom gdje se granice izdržljivosti često probijaju.

Testiranje učinka se provodi kako bi se procijenilo ponašanje sustava kada se veliko opterećenje značajno poveća i naglo dogodi. Sustav može veoma neočekivano da djeluje kada se dogode velike promjene u opterećenju. Zbog toga svi tipovi testiranja perfomansi imaju svoju ulogu u konačnom osiguranju i stabilnosti sustava.

5.3. Standardizacija i certifikacija testiranja softvera

Kako je programsko inžinjerstvo jedno od najmlađih inženjerskih znanosti s vremenom se pojavila potreba za što većom standardizacijom. Kako bi se izbjegao „nered“ koji je bio karakterističan za mladu znanost u svom nastajanju, postavljanje principa, načela i metoda rada bilo je neizbježno, jer se radi o potrebi za što kvalitetnijim proizvodima i uslugama iz ovog sektora. Tako je uporaba informacijskih tehnologija glavna karakteristika modernog društva i neizostavan dio svih velikih industrija. Da bi se osigurala kvaliteta neophodna je standardizacija, a s vremenom i ispitivanje kompetencija ljudi u toj industriji, tzv. certifikacija.

Prema (Padmini, 2006) neke od najraširenijih certifikacija iz domene testiranja softvera su:

- CSQE - Certificirani inženjer prema američkoj društvenoj organizaciji za kvalitetu ASQ (American Society for Quality) podrazumijeva razvoj i implementaciju kvalitete softvera, pregled softvera, testiranje, provjeru valjanosti te provodi procese i metode razvoja i održavanja softvera (ASQ, 2019).
- CSQE – Certifikacija za testiranje softvera od strane QAI (Quality Assurance Institute) namijenjena je uspostavljanju standarda za početnu kvalifikaciju i pružanje okvira za funkciju testiranja putem agresivnog obrazovnog programa (CSQE, 2019).
- CSTA – Stjecanje kvalifikacije certificiranog analitičara kvalitete softvera (CSQA) od strane QAI (Quality Assurance Institute) ukazuje na profesionalnu razinu kompetencije u načelima i praksama osiguranja kvalitete u IT struci (CSQA, 2019).
- ISTQB je skraćenica od Međunarodni odbor za osposobljavanje testera softvera (*engl. International Software Testing Qualifications Board*). To je međunarodna organizacija koja se bavi certificiranjem i standardizacijom testiranja softvera. Organizacija prati i definira trendove u programskom inženjerstvu te u korelaciji sa drugim organizacijama vrši certificiranje u ovoj branši. Veliki je doprinos ove organizacije u teoriji testiranja softvera (ISTQB, 2011).

Iako se radi o mladoj grani u IT industriji, danas postoje veoma kvalitetne organizacije koje vrše certifikaciju za kompetencije testiranja programskih proizvoda. U razvijenim tržištima ovaj tip certifikacije se sve više traži kod angažiranja i zapošljavanja stručnjaka, dok u našem tržištu ne postoji još uvijek certifikacija za ovu granu. Svakako, zadatak šire IT zajednice da uključi u obzir certifikaciju kod zapošljavanja testera.

5.4. Normativne reference testiranja u programskom inženjerstvu

Prema (ISO/IEC JTC1/SC7, 2014) postoji više svjetskih standarda odnosno normativnih referenci koji se tiču testiranja programskih proizvoda, od kojih su:

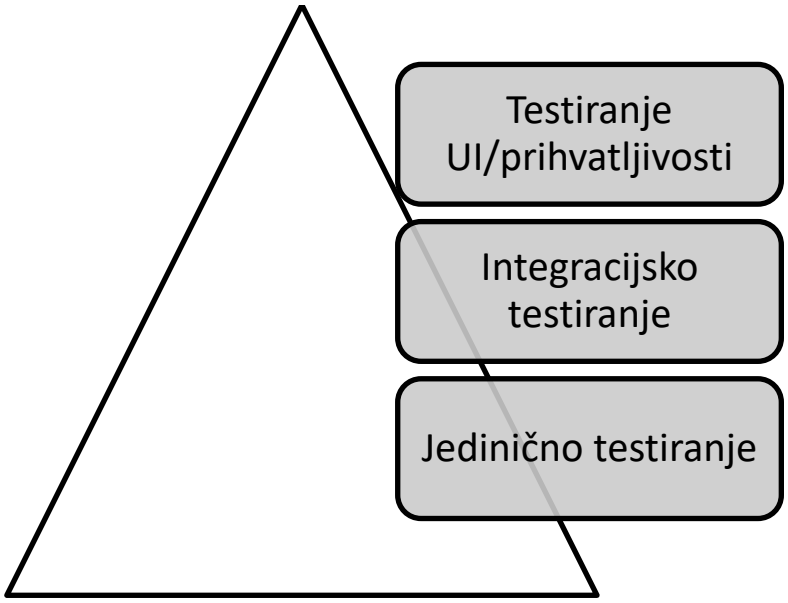
- ISO 9000:2005 – Quality Managment System – Fundamentals and vocabulary
- ISO/IEC 29119-1: Concepts & Definitions (rujan 2013)
- ISO/IEC 29119-2: Test Processes (rujan 2013)
- ISO/IEC 29119-3: Test Documentation (rujan 2013)
- ISO/IEC 29119-4: Test Techniques (2014)
- ISO/IEC 29119-5: Keyword Driven Testing (2015)

Ovi standardi su nastali iz potrebe standardizacije ponajprije zbog velikih sustava kojima je osiguranje kvalitete veoma bitno i gdje se ulažu veliki resursi kako bi proizvod bio što manje rizičan u svojoj primjeni. Velike industrije koje koriste informacijske sustave sa osjetljivim podacima ulažu znatne resurse u istraživanje modernih koncepata i praksi, te ih spojili sa IT industrijom u standardizirani program, ponajprije kroz ISO standardizaciju.

6. Razine testiranja

Većina programskih proizvoda sastoji se od tri razine: korisničko sučelje ili prezentacijska razina, servisi i programska logika. Ovisno o metodologiji, tih razina može biti više od tri. Svaka od tih razina se testira zasebno. Te razine čine osnovnu piramidu testiranja.

Piramida testiranja prema pristupu „odozdo-prema-gore“ (*engl. bottom-to-up*) ima tri ili više razina, što možemo vidjeti na slici 7. Prema tom pristupu prvi korak je izrada i izvršavanje jediničnih testiranja, zatim integracijsko testiranje te na kraju testiranje korisničkog sučelja ili po nekim autorima testiranje prihvatljivosti. Moguće je ovoj piramidi testiranja dodati i sustavsko testiranje.



Slika 7. Piramida automatskog testiranja (autorski rad)

Gledano po principu „odozdo-prema-gore“ razine testiranja koji su standardizirani prema (Software Testing Fundamentals, 2017):

- jedinično testiranje,
- integracijsko testiranje,
- sustavsko testiranje i
- testiranje prihvatljivosti.

Jedinično testiranje podrazumijeva testiranje svake jedinice kao osnovne komponente u cilju određivanja ispravnosti. Integracijsko testiranje karakterizira testiranje svih povezanih komponenti od manje do veće razine apstrakcije. Cilj je da sve komponente funkcioniraju kao cjelina.

Testiranje ili provjera sustava (*engl. system testing*) je proces testiranja integriranog sustava u svrhu provjere da li cjelokupni sustav odgovara specificiranim zahtjevima. Provjerava se cjelokupni integrirani softver, kako navodi (Jorgensen, 2013) u svojoj knjizi.

Testiranje prihvatljivosti je formalno testiranje programskog proizvoda na osnovu korisničkih potreba, zahtjeva, potreba poslovnog procesa koja se provodi, da bi se na kraju utvrdilo da li sustav zadovoljava potrebe. Kriterije prihvatljivosti postavlja klijent koji donosi krajnju ocjenu. Više o pojedinim razinama testiranja u nastavku.

6.1. Jedinično testiranje

Jedinično testiranje (*engl. unit testing*) je razina testiranja programskih proizvoda gdje se ispituju pojedine jedinice ili komponente sustava. Cilj i svrha jeste provjeriti da li svaka jedinica radi kako je zamišljeno i definirano u specifikaciji zahtjeva. Jedinica odnosno komponenta je najmanji dio koji se testira. Računalnim jezikom govoreći, jedinica ima obično jedan ili nekoliko ulaza i jedan izlaz. U proceduralnom programiranju, jedinica se može označiti kao funkcija, postupak ili cijelina u programu. U objektno-orijentiranom programiranju, najmanja jedinica je metoda koja može pripadati klasi, apstraktnoj klasi, sučelju ili izvedenoj klasi. Također, postoje prakse u kojima se moduli aplikacije tretiraju kao jedinice koje se testiraju. Takva praksa ima mana u slučaju da postoji mnogo pojedinačnih jedinica unutar modula koje se mogu zasebno tretirati.

Jedinično testiranje izvodi se pomoću metode ispitivanje bijele kutije. Ujedno, ono je prva razina testiranja gledano pristupom „odozdo-prema-gore“ (*engl. bottom-up testing*). Ovu razinu testiranja obavljaju programeri koji prethodno kreiraju komponente aplikacije. Uloga testera u ovoj razini je minimalna ili nepostojeća, ali je važno da bazično poznavaju proces i svrhu ove razine testiranja. Također, mnogo im govori na čemu trebaju bazirati svoje eksplorativno testiranje (*engl. exploatory testing*). Postoje prakse u kojima programeri i testeri u paru pišu i izvršavaju jedinične testove.

Eksplorativno testiranje je neformalna tehnika, kod koje tester aktivno kontrolira oblikovanje testova kada su ti testovi provedeni. Testeri koriste te informacije za unaprijeđenje procesa testiranja u budućnosti. Tijek izvođenja jediničnog testiranja u cjelokupnom smislu izvodi se kroz pisanje plana, slučajeva jedinica i pisanja koda za ispitivanje valjanosti pojedinih komponenti.

Testiranje jedinica sustava povećava povjerenje u kvalitetu koda odnosno u njegovu skalabilnost. Skalabilnost je sposobnost programskog proizvoda za nadograđivanje kako bi podržao dodatno opterećenje i nadogradnju. Skalabilno testiranje je ispitivanje sposobnosti

programskog proizvoda za podnošenje dodatnog opterećenja. Ukoliko se jedinični testovi pišu precizno i kvalitetno, moguće je odmah nakon promjena i dodatnih opterećenja, „uhvatiti“ i prepoznati pogreške na način da je poznat okvirni uzrok i mjesto nastanka u kodu. Također, ukoliko su promjene već napravljene, a jedinice su malo međusobno ovisne, moguće je lakše i brže napisati jedinične testove te je neželjeni učinak promjena u kodu značajno manji. Bushnev u svom članku navodi kako je između ostalog dobra praksa pisanja koda svakako pisati dijelove koda što manje ovisne jedne od drugih (2014).

Mnoge su prednosti prve razine testiranja. Jedna od njih svakako je ponovna iskoristivost komponenti. Kako bi se moglo sustavno pristupiti pisanju jediničnih testova, kod treba biti što više modularan sa jasnom definicijom, svrhom i jasnošću. To su dobre pretpostavke za ponovno iskorištavanje koda. Jedinično testiranje na neki način „nagrađuje“ dobro pisani kod, ali i tjera programera da piše što čišći kod, kako opisuje Martin u svojoj poznatoj knjizi „*Clean Code*“ (2002). U principu, ova razina omogućuje brži razvoj. Kada se programeri služe uobičajenim pristupom kao što je često postavljanje točaka prekida (*engl. breakpoints*), ulaze u tzv. vrtlog vremena posebno kada je struktura programa kompliciranija. Jedinično testiranje može ubrzati cijeli proces zbog toga što je sigurniji pristup napisati i pokrenuti test dinamički. Također, pisanje ovih testova poboljšava samo razumijevanje vlastitog koda i procedura koje program izvršava.

Napor koji je potreban za pronalaženje i popravljanje nedostataka pronađenih tijekom jediničnog testiranja je veoma manji u odnosu na nedostatke otkrivene u višim razinama aplikacije. Samim time i cjelokupni trošak razvoja je manji. Pod troškove razvoja smatra se uloženo vrijeme, ljudski i drugi resursi. Veoma loša praksa je suočiti se sa pogreškama tijekom aktivnog korištenja ili tijekom testiranja prihvaćanja (*engl. acceptance testing*). Ispravljanje pogrešaka je jednostavnije. Kada pojedini test ne uspije, potrebno je samo najnovije izmjene popraviti, pod uvjetom da su u prethodnim iteracijama sve faze testiranja pravilno završene.

6.1.1. Dobre prakse u jediničnom testiranju

Prvo što treba odlučiti kod pisanja jediničnih testova, jeste odabrati odgovarajući alat za testiranje koji je podržan za programski jezik koji se koristi. Alati za testiranje (*engl. frameworks*), slobodne biblioteke ili paketi omogućavaju programerima testiranje jedinica u programskom jeziku kojim programiraju same jedinice. Tako da većina programskih jezika ima određenu vrstu podrške za jedinično testiranje. Primjerice, testiranje aplikacija u C# omogućeno je kroz Visual Studio IDE kroz integrirane biblioteke.

Nema potrebe stvarati slučajeve testiranja. Formalno za testne slučajeve se piše specifikacija na početku procesa testiranja odnosno u fazi planiranja. Specifikacija testnih slučajeva je dokument koji specificira skup ciljeva, ulaza, izlaza i akcija jedinice koja se testira. Dobra praksa je usredotočiti se na jedinice koje uveliko utječu na ponašanje sustava na tzv. kritične točke sustava. Jediničnim testiranjem cilj je postići izolaciju odnosno izdvajanje razvojnog i testnog okruženja programa.

Dobra je praksa korištenje podataka koji će se koristiti u produkciji omogućava rano otrkivanje pogrešaka pogotovo onih koji nisu specifični za programsku logiku nego na izuzetke korištenja. To se posebice odnosi na ulazne podatke. Ukoliko se neki nedostatci slučajno otkriju ručnim testiranjem ili *ad-hoc* metodom, poželjno je napisati test, pa zatim otkloniti pogrešku, jer time preveniramo mogućnost ponavljanja te ili slične pogreške. Karakteristika programera jeste da žele što više stvari automatizirati. Iz tog razloga ne vole ponavljanje procesa koji stvaraju nelagodu. To se postiže kontinuiranom nadogradnjom i poboljšanjem pisanja koda.

Kao što sam napomenuo, treba težiti pisanjem testnih slučajeva koji su međusobno neovisni. Primjerice, ako klasa ovisi o bazi podataka, ne pišite slučaj koji ispituje klasu interakcijom sa bazom podataka. Umjesto toga, izradite apstraktnu klasu ili sučelje oko te veze sa bazom podataka i implementirajte ta sučelja tako što ćete je postaviti u ulogu posrednika. Konačni cilj je pokriti sve putanje kroz jedinicu. Posebnu pozornost treba obratiti na uvjetnje petlje.

U konačnici, verzioniranje koda je veoma korisno i u danjašnjem modernom programiranju gotovo pa neizbježno kada su u pitanju veće aplikacije i više učesnika u razvoju. Korištenje sustava za upravljanje verzijama koda kao što je Git, uvelike poboljšava učinkovitost programiranja i predstavlja dobru praksu.

6.1.2. Korištenje alata za kreiranje jediničnih testova

Postoji cijeli niz kvalitetnih alata koji nude široku lepezu mogućnosti za jedinično testiranje. Neki od popularnih alata za jedinično testiranja za popularne programske jezike:

- xUnit.net – besplatni alati za testiranje C# .NET aplikacija (poveznica za preuzimanje: https://xunit.net/).
- JUnit i NUnit – besplatni okviri/alati za jedinično testiranje aplikacija pisanih u Java programskom jeziku (poveznica za preuzimanje: https://junit.org/junit5/).
- PHPUnit – okvir/alat za PHP jezik (poveznica za preuzimanje: https://phpunit.de/).

- Karma i Jasmine – besplatni alati za testiranje web aplikacija u JavaScript-u (poveznica za preuzimanje: https://karma-runner.github.io/latest/index.html).

Kako broj programskih jezika ubrzano raste, tako se pojavljuje sve više alata za jedinično testiranje. Posebice, popularni programski jezici imaju nekoliko alata podrške za jedinično testiranje, kao što je primjerice C#.

6.2. Integracijsko testiranje

U drugu razinu testiranja spada integracijsko testiranje koje predstavlja spajanje ključnih točki sustava kroz razne slojeve aplikacije. Integraciju kao pojam možemo definirati kao proces kombiniranja komponenti u veće cjeline. Radi se o testiranju koje se provodi radi utvrđivanja pogrešaka u sučeljima i u interakcijama između integriranih komponenti.

Integracijsko testiranje je razina testiranja gdje se pojedine jedinice (*engl. functional units*) odnosno funkcionalne jedinice sustava kombiniraju i testiraju kao skupina kako bi se ispitale kritične točke spajanja. Svrha ove razine jeste otkrivanje pogreški u procesu interakcije između spojenih funkcionalnih jedinica. Testni upravljački programi ili komponente (*engl. tests drivers*) često se koriste kao pomoćni alati kod integracijskog testiranja. Komponentno integracijsko testiranje se provodi s ciljem pronalaženja grešaka u sučeljima i interakciji između integriranih komponenti.

Svaka metoda testiranja kao što su metoda crne, bijele i sive kutije primjenljiva je na ovoj razini testiranja. Metoda ponajviše ovisi od preciznosti definiranja funkcionalne jedinice koja se spaja u cijelinu. Pripremni koraci za uspostavljanje integracijskog testiranja su razrada testnog plana i testnih slučajeva kroz iteracije aktivnosti pripreme, pregleda, izmjene i izrade. Jedan od savjeta za kvalitetnije izvršavanje integracijskih testova je detaljno definiranje interakcije između više jedinica. Dokumentacija o samim jedinicama je veoma bitna, kao i osigurano funkcioniranje tih jedinica prema zahtjevima, kako bi integracijski testovi imali smisla. Dakle, prije testiranja integracije treba testirati svaku pojedinu komponentu.

Također, u programskom inženjerstvu poznat je i pojam integracijskog testiranja sustava koje testira sučelja eksternih organizacija koje se integriraju sa sustavom. To je veoma zahtjevan zadatak s obzirom da je teže testirati eksterne resurse kada su projektirani i razvijeni od strane drugih organizacija. U ovom slučaju mnogi se moraju osloniti na postojanje kvalitetne tehničke i korisničke dokumentacije tih sučelja. Koliko god je moguće, treba omogućiti što automatiziraniji proces izvršenja testova, posebno kada se koristi pristup „odozdo-prema-gore“.

Integracijski testovi je jedan od tipova koji kombinira više od jedne strane, odnosno u ovom slučaju krajnjih točaka, korisničkog sučelja i poslužiteljske strane. Oni su bitni iz mnogo razloga. Jedan od njih je uloga kritične uloge u otkrivanju pogrešaka i iznimki u niskom nivou realizacije koje smo propustili u slojevima jediničnog testiranja. Integracijski testovi su važna spona i garant stabilnosti između ostalih razina testiranja.

U praksi integracijskog testiranja uloga programera je testirati funkcionalne jedinice *back-end* web servisa pomoću jednostavnih HTTP zahtjeva. Ova razina testiranja prvenstveno je zadatak programera i onih koji poznaju ponašanje i teoriju HTTP zahtjeva odnosno web servisa. Za ovaj proces testiranja obično je dostupno korisničko sučelje nalik onom krajnjem korisniku kao kod testiranja korisničkog sučelja. Testeri i programeri u ovom slučaju idu dublje u samu arhitekturu sustava tako što će odrediti točke slanja i primanja i time simulirati i izvršavati zahtjeve. Naime, radi se o višem sloju apstrakcije. Potrebno je „zagristi ispod površine“. Svakako je bezbolnije na toj razini otrkiti probleme u izvržavanju zahtjeva.

6.2.1. Testiranje web servisa

Kod razvijanja programskih proizvoda kao što su web aplikacije, važan segment funkcioniranja su web servisi, poznati i kao `RESTfull` servisi, `SOAP`, `Web API` itd. Oni predstavljaju programske jedinice koje se pokreću i izvršavaju na web poslužiteljima i odgovaraju na `HTTP` zahtjeve od strane klijentske strane. Svakako, osnovni uvjet za testiranje web servisa je dubinsko poznavanja kako web tehnologije rade. Osnovna arhitektura interneta je ostala ista, tako da su osnove na kojima počivaju web servisi poprilično standardizirani i učestali u svojoj primjeni. Potrebno je poznavati osnove mreža računala, a kroz njih specifično rad aplikacijskog, prezentacijskog, transportnog i mrežnog sloja internetske mreže. Pod osnovne `HTTP` zahtjeve smatramo zahtjev primanja (`GET`), zahtjev slanja (`POST`), ažuriranja (`PUT`) i brisanja (`DELETE`).

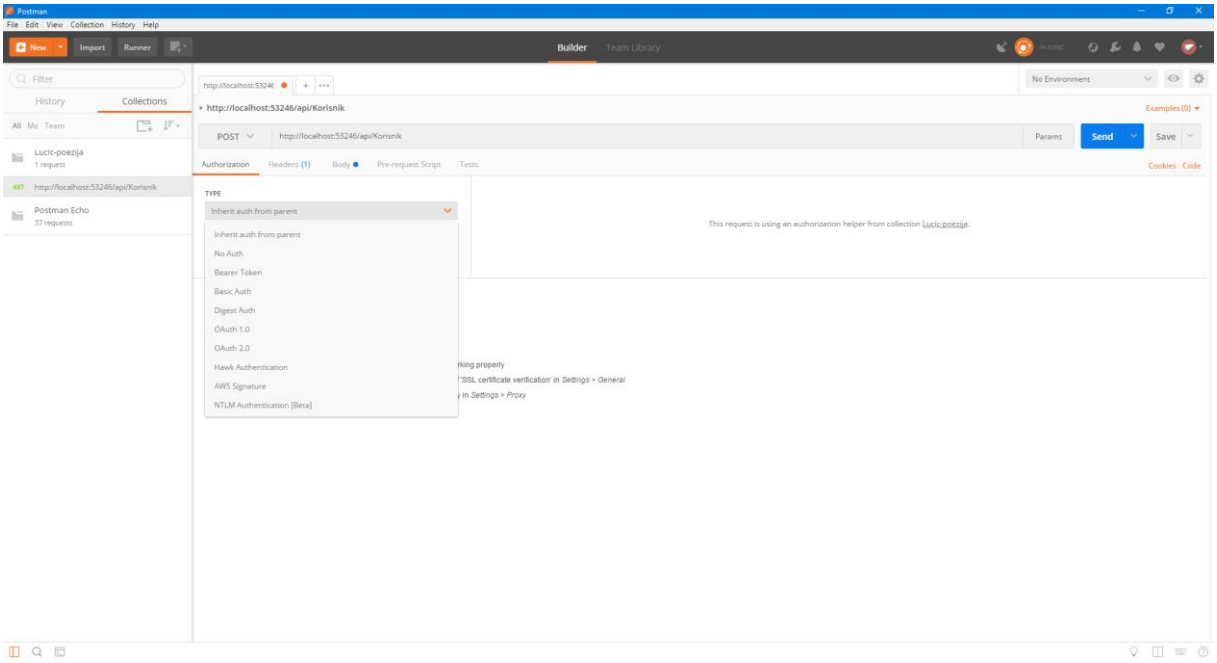
6.2.2. Korištenje popularnih alata za integracijsko testiranje

Postman spada u red veoma popularnih i široko korisnih alata koji pomaže programerima i testerima pri izradi i testiranju API-ja, web servisa i RESTful servisa. Ovaj alat se koristi u mnoge svrhe, a najčešća je u integracijskom testiranju.

Postoji nekoliko vrsta web servisa i u zavisnosti arhitekture tih servisa, te postoje razni alati koji su specijalizirani odnoso korisni za integracijsko testiranje, kao npr. SoapUI za testiranje Soap web servisa. Ono što treba napomenuti jeste korisnost dinamičke dokumentacije za API kao što je Swagger, koji omogućava da se testiraju servisi putem web

aplikacije, neovisno od pristupa sa korisničkog sučelja aplikacije. Ono što treba napomenuti, kada je u pitanju učestala praksa programera, jeste upravo fokus na integracijskom testiranju web servisa kroz sustavno praćenje šta se događa sa ulaznim i izlaznim modelima, parametrima i ostalim rezultatima zahtjeva.

Postman se koristi na jednostavan način tako što se definira tijelo, zaglavlje, autentifikacija i ostale postavke zahtjeva. Moguće je vjerno simulirati zahtjeve iz aplikacije. Stoga, Postman spada u red najkorištenijih alata kod testiranja programskih proizvoda, uopće. Jedan dio problema koji su zanemareni kod integracijskog testiranja, mogu se otkriti kod testiranja korisničkog sučelja. Na slici 8 možemo vidjeti kako izgleda definiranje `POST` zahtjeva u Postman alatu.



Slika 8. Postman - alat za testiranje API-ja (snimak zaslona)

6.3. Testiranje korisničkog sučelja

Testiranje korisničkog sučelja, često nazvano UI testiranje spada u red automatiziranih testova visoke razine apstrakcije. Pišu se skripte koje testiraju aplikacije na isti način na koji to rade krajnji korisnici. Oni klikaju, odabiru i obavljaju sve druge radnje koje svaki prosječni korisnik radi prilikom korištenja web aplikacije. U suštini, ovi testovi su skriptna reprezentacija onoga što rade korisnici. Svrha svakog programskog proizvoda treba biti takva da su krajnji korisnici zadovoljni. Sve ono što se nalazi u pozadini, nije bitno ukoliko su oni nezadovoljni cjelokupnim proizvodom ili uslugom. Taj univerzalni princip tjera testere i programere da koriste ovom vrstom testiranja. S druge strane, sama suradnja testera i

programera je veoma bitna da li kroz dobre prakse dizajniranja informacijskog sustava ili kroz svakodnevnu komunikaciju. Većina timova ima česte probleme u oba slučaja.

U piramidi testiranja programskih proizvoda često se postavlja na vrh same piramide testiranja. U današnjem programskom inženjerstvu kada se spomene korisničko sučelje često se podrazumijeva grafičko korisničko sučelje, jer sve veći broj aplikacija u potpunosti prelazi u grafičku reprezentaciju umjesto tekstualno orijentiranih instrukcija. Alternativa tome su konzolne aplikacije.

Ovi testovi u suštini prolaze kroz sve slojeve aplikacije „od-kraja-do-kraja“. To znači da prolaze kroz sve različite dijelove, kao što je korisničko sučelje, niže slojeve servisa i integracije sa samom bazom podataka. Prezentacijski sloj često može otrkiti mnoge nedostatke nižih slojeva aplikacije. On na neki način može biti zamjena za testiranje prihvatljivosti aplikacije.

6.3.1. Dimni testovi

Jedan od pojmova koji se koristi kod ove razine testiranja jeste „dimni test“ (*engl. smoke test*). Također, poznat i kao verifikacijsko testiranje izgradnje (*engl. Build Verification Testing*). To su testovi koji provjeravaju osnovne funkcije informacijskog sustava tako što vidi da li su pokrenuti i funkcionalni. Oni osiguravaju da najvažnije funkcije sustava rade stabilno. „Skup svih definiranih/planiranih testnih slučajeva koji pokrivaju glavnu funkciju komponente/sustava u svrhu provjere ključnih funkcija programa, ali bez zamaranja sa finim detaljima. Dnevno izdanje i dimni test spadaju u najbolje prakse širom industrije.„ (ISTQB, 2011.).

Izraz „dimno testiranje“ došao je kao aluzija na sličan tip hardverskog testiranja u kojem uređaji prolaze kroz test gdje je cilj da se „ne zapali“, odnosno ispita da li ostaje funkcionalan nakon prvog uključivanja. Prema (Rasmusson, 2016., str. 21) korisno je zato što:

- provjerava je li aplikacija ispravno isporučena,
- provjerava je li okruženje ispravno postavljeno,
- i jesu li svi dijelovi arhitekture ispravno povezani i postavljeni pravilno.

Dimni testovi čine UI testove korisnim u cjelokupnom procesu osiguravanja kvalitete programskog proizvoda. Primjerice, ukoliko želimo testirati mogućnost prijavljivanja na neku web aplikaciju unošenjem korisničkog imena, lozinke i klika na gumb, preusmjeravanja na stranicu dobrodošlice, trebamo dobro razmisliti i zapisati korake ponašanja sustava sa strane

krajnjeg korisnika. Kada zapišemo korake, potrebno je pretvoriti ih u test koji će to pravilno ispitati.

Mnogi programski jezici imaju već dostupne biblioteke koje olakšavaju ovaj proces. Ukoliko se radi o slučaju korištenja koji se često koristi u raznim sličnim sustavima, poželjno je potražiti rješenja koja su već usvojena kao uzorak dizajna ili kao dobra praksa. U osnovi, UI testovi se mogu bazirati na ključnim alatima za prikaz web aplikacija, kao što su HTML (*engl. Hypertext Markup Language*) i CSS (*engl. Cascading Style Sheets*). Za primjer prijavljivanja na sustav, kroz ovaj tip testiranja potrebno je ispitati dohvaćanje elemenata kroz njihov tip podataka i jedinstveni indeks.

Kada su u pitanju moderni principi UI testiranja, treba obratiti pažnju na nekoliko važnih principa. Jedan od najvažnijih svakako je ne oslanjati se samo na UI testove, kako kaže Blazedemo (2017). Zatim, treba koristiti okvire za testiranje ponašanja sustava (*engl. behavior-driven development*).

Bazični okvir za uspješno testiranje jeste korištenje uzoraka dizajna i principa testiranja. Ono što mnogi zaborave u želji da pokriju što šire mogućnosti korištenja programa, jeste da pokušavaju praviti i pokretati sve testove na svim pretraživačima istovremeno, što je pogrešna praksa iz više razloga. Jedno su ograničeni resursi, a drugo nemogućnost detaljnog uvida u pojedinosti rezultata testova na pojedinim pretraživačima.

Između ostalog, potrebno je voditi računa o prenosivosti alata za automatsko testiranje. Korištenje standardiziranih konvecija imenovanja testova je jedan od važnih principa kvaltietnog dizajna testova. Spremanje slika ekrana prilikom neuspješnih provjera, jedan je od trikova koji mogu olakšati daljni razvoj testova. Iako je dodavanje komentara veoma korisno u razvoju programskih proizvoda, dobra je praksa pisati pojednostavljene testove. Pored toga, svi testovi treba što manje ovisiti jedan od drugog. Zatim, treba težiti prema korištenju dinamičkih procedura baziranim na podacima. U programskom inženjerstvu uvijek treba težiti automatizaciji procesa, jer ono na duže staze uvijek daje bolje rezultate od ručnog statičkog procesa. Korisno je postaviti detaljno izvještavanje kroz dnevnike rada.

Ta vrsta testiranja omogućuje programerima bolje razumijevanje suštine rada aplikacije sa strane krajnjih korisnika. To znanje je neophodno kako bi programeri testerima omogućili da aplikacija bude što više pogodna za pisanje i nadogradnju UI testova. A sama uloga testera je da pokrije što veći raspon testiranjem „od-kraja-do-kraja“ korištenjem UI testova. Treba biti spreman potrošiti dodatno vrijeme i resurse na mnoge „lukave dijelove“ aplikacije, korištenjem drugih vrsta testiranja. Stoga, UI testiranje često nije dovoljno, ali zato pokriva najširi spektar.

6.3.2. Alati za UI testiranje

Danas postoji niz kvalitetnih i naprednih alata za UI testiranje. Razvoj ovih alata polako mijenja paradigmu, širinu i mogućnosti ove veoma korisne razine testiranja. Neki od tih alata se koriste neovisno o platformi, tehničkim uvjetima, programskim jezicima itd. Postoji niz specijaliziranih alata, od kojih se ističu Selenium i Apache JMeter kao najpopularniji okviri za automatskoh testiranje.

Neki od najpopularnijih alata za automatsko testiranje (Kazeeva, 2018):

- Selenium IDE i Selenium Web Driver (poveznica za preuzimanje: https://www.seleniumhq.org/selenium-ide/)
- Carina (poveznica na GitHub repozitorij: https://github.com/qaprosoft/qps-infra)
- Google EarlGrey (poveznica na GitHub repozitorij: https://github.com/google/EarlGrey)
- Cucumber (poveznica na instalaciju: https://cucumber.io/)
- Watir (poveznica na instalaciju: http://watir.com/)
- Appium (poveznica na GitHub repozitorij: https://github.com/appium/appium-desktop/releases/tag/v1.13.0)
- RobotFramework (poveznica na instalaciju: https://robotframework.org)
- Apache JMeter (poveznica na instalaciju: https://jmeter.apache.org/download_jmeter.cgi)
- Gauge (poveznica na instalaciju: https://gauge.org/getting-started-guide/we-start/)
- i Robotium (poveznica na GitHub repozitorij: https://github.com/RobotiumTech/robotium)

Selenium je skup alata (*engl. framework*) za automatsko testiranje korisničkog sučelja web aplikacija. On pokriva široki spektar korisnih mogućnosti za kvalitetno testiranje aplikacija na svim široko raprostranjenim pretraživačima. Selenium ima dobru podršku za pisanje testova pomoću različitih platformi. On je svakako najpopularniji i najrasprostranjeniji alat za automatsko UI testiranje. Dio Selenium paketa su Selenium IDE i Selenium Web driver. To je skup alata i dodataka aktulnim preglednicima, kao što su Chrome, Firefox, Edge i drugi.

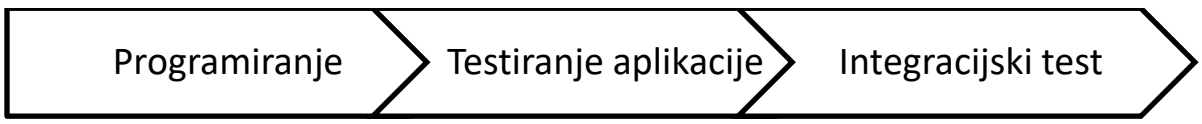
7. Testiranje u kontekstu kontinuirane integracije, isporuke i objave

Pojmovi kao što su kontinuirana integracija, dostava i objava često idu zajedno iz razloga što moderni alati objedinjuju procese tj. nude sve te mogućnosti kroz jedan alat (Maynard, 2018). Mnogim velikim, srednjim i malim razvojnim timovima nemoguće je zamisliti proces bez kontinuirane integracije, dostave i/ili objave. Iako su ovi principi veoma mladog datuma, oni su široko prihvaćeni. Postavlja se pitanje gdje je mjesto testiranja programskih proizvoda kod kontinuirane integracije, dostave i objave. Testiranje je ustvari sastavni dio procesa svih ovih metoda rada. Kontinuirana integracija i objava je primjenjiva u kombinaciji raznih modela razvoja i kao takva testiranje podiže na višu razinu.

Kontinuirana integracija i kontinuirana dostava dvije su moderne prakse razvoja softvera. Kontinuirana integracija proces je automatizacije izrade i testiranja koda svaki put kada neki član tima izvrši promjene u kodu kojeg prepoznaje sustav za verzioniranje koda.

7.1. Kontinuirana integracija

Kontinuirana integracija (*engl. Continuous integration*) je praksa kod testiranja programskih proizvoda koja omogućuje integraciju izvornog koda neprekidno u više iteracija i instanci. Svrha kontinuirane integracije jeste brzo otrkivanje pogreški i lakše pronalaženje izvora odnosno okidača pogreške. Proces kontinuirane integracije se okida pri završetku kodiranja neke određene jedinice, poboljšanja ili nadogradnje, te pokreće skripte za spajanje postojećih i novih dijelova u koda u funkcionalnu cjelinu i sustav. Kako vidimo na slici 9, taj proces se može razložiti u nekoliko etapa. Najprije, potrebno je prije rada na nekom programerskom zadatku stvoriti `branch`, odnosno odvojenu instancu koda u sustavu za verzioniranje Git. Prilikom završetka kodiranja na nekoj funkcionalnoj jedinici, potrebno je spremiti i objaviti promjenu. Nakon toga slijedi proces spajanja glavne verzije koda `master` sa tom odvojenom jedinicom koda. Alati za kontinuiranu integraciju prepoznaju te promjene te izvještavaju o problemima u točkama spajanja jedinica. Ukoliko je integracija pala na testiranju, programer ima za zadatak riješiti taj problem spajanja, te ponovo pokrenuti test za integraciju, a nakon toga dostavu i objavu.



Slika 9. Prikaz procesa kontinuirane integracije (autorski rad)

Neke od dobrih praksi korištenja kontinuirane integracije prema (DevOpsZone, 2019) su:

- Održavanje i pravilno korištenje Git repozitorija.
- Automatiziranje pokretanja svih modula projekta.
- Provjeravanje ispravnosti izvornog koda.
- Konfiguriranje okruženja pokretanja prema potrebama i mogućnostima.

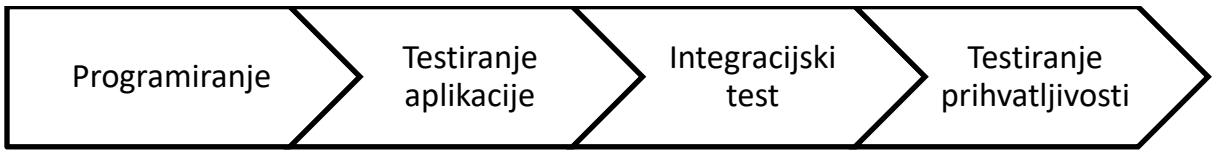
Kontinuirana integracija se može ugraditi u svaki od suvremenih modela razvoja i može biti dio koji osigurava kvalitetnu integraciju cjelokupnog procesa. U modernom agilnom DevOps modelu, kontinuirana integracija sudjeluje u kompliciranom, ali i detaljnom procesu razvoja.

7.2. Kontinuirana isporuka

Kontinuirana isporuka ili dostava (*engl. Continuous Delivery*) na neki način objedinjuje sve pakete distribucije sustava u jedan koji će se na kraju izvršiti. Može se smatrati produžetkom kontinuirane integracije i postupak je automatskog pokretanja aplikacije nakon uspješne integracije. Cilj je minimizirati vrijeme izvođenja, vrijeme koje je proteklo između pisanja novih linija koda i postojećeg koda koji koriste aktivni korisnici u produkciji.

Kontinuirana isporuka sprema pakete za objavu na osnovu promjena, nove funkcionalnosti, promjene konfiguracije, ispravke greški u kodu ili pak kao paket instalacije krajnjem klijentu. Cilj ovog procesa je proizvesti kompletne pakete promjene na siguran, brz i održiv način. Bilo da se radi o složenim aplikacijama, velikom distribuiranom sustavu, kontinuirana isporuka pravilnom konfiguracijom može olakšati proces koji bi inače mogao trajati mnogo tuže i sa puno više rizika. Bez obzira što na istom repozitoriju može raditi stotine programera te praviti svakodnevne promjene, ovaj proces je u mogućnosti upravljati tim promjenama na način da osigura sigurnu isporuku.

Na slici 10 možete vidjeti tijek procesa uobičajene kontinuirane dostave ili isporuke.

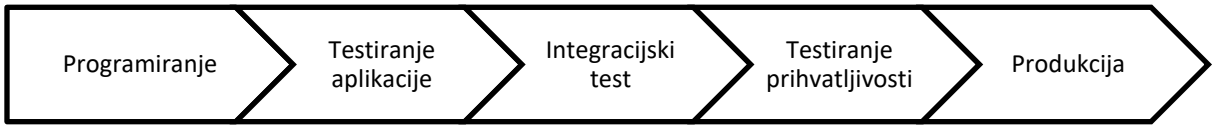


Slika 10. Prikaz procesa kontinuirane dostave ili isporuke (autorski rad)

Svi koraci kontinuirane integracije su ujedno i dio procesa kontinuirane dostave paketa. S tim, da je novi korak upravo testiranje prihvatljivosti. Treba napomenuti kako kontinuirana dostava ustvari čini presliku pristupa „odozdo-prema-dole“ u suvremenoj varijanti piramide testiranja. To možemo utvrditi koracima koji od programiranja, odnosno testiranja jedinica, preko integracijski testova i testova prihvatljivosti zatvara se piramida testiranja u navedenom pristupu. Svakako, kontinuirana dostava dio je procesa sljedeće razine, a to je kontinuirana objava, koji na kraju objedinjuje promjene i integraciju u paket koji se distribuira na testna i produkcijska okruženja.

7.3. Kontinuirana objava

Treći korak koji slijedi nakon uspješne kontinuirane integracije i isporuke je postavljanje kontinuirane objave programa (*engl. Continuous deployment*). To u prijevodu znači objava na produkciju odnoso na okruženje koje aktivno koriste klijenti. Naravno, objava na produkciju kao automatiziran proces može biti rizična, ali svakako olakšava i ubrzava posao. Ručna objava je veoma stresna i naporna i te ima mogućnost velikog postotka greške. Kontinuirana objava rješava te probleme automatskim prepoznavanjem promjena u glavnom produkcijskom repozitoriju, te sprema samo nastale promjene u pakete koji se spremaju na produkcijsko okruženje. U nastavku možete vidjeti tijek procesa koje uključuje kontinuirana objava uključujući prethodne etape kontinuirane integracije i isporuke. Na slici 11 vidimo tijek procesa kontinuirane objave.



Slika 11. Prikaz procesa kontinuirane objave (autorski rad)

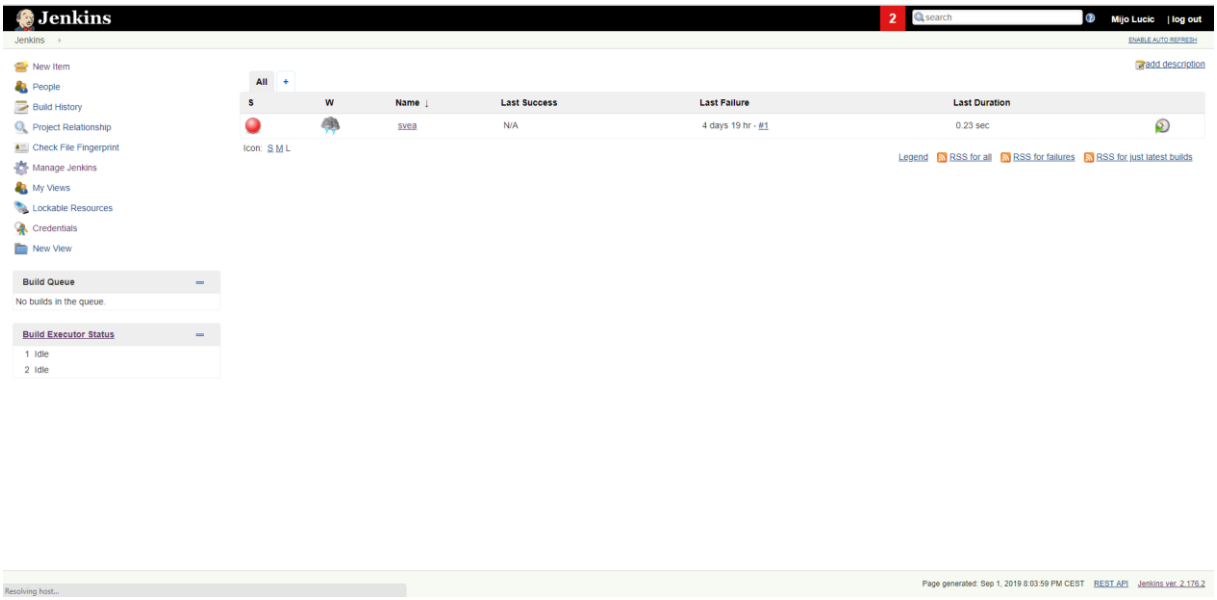
Kontinuirana objava jedan je dio, proces, koji se može ili ne mora provoditi kao dio cjelovitog DevOps modela. To znači da prije nego što počnete koristiti kontinuiranu objavu morat ćete se prebaciti na DevOps model razvoja.

Najbolji dio DevOps-a je u tome što on okuplja sve timove koji bi na kraju mogli komunicirati s novim projektom i omogućava im da istovremeno surađuju, daju i primaju povratne informacije i ubrzavaju ciklus razvoja i implementacije. Poput agilne metodologije, DevOps se može činiti suviše kompliciranim za primjenu za mnoge organizacije.

Mnogi su benefiti korištenja ovih praksi, a neke od vidljivijih je svakako smanjenje pogrešaka u kodu, manje slijepog pregleda koda i analiziranja rada funkcionalnosti od strane testera, te su ukupni troškovi testiranja manji. Kontinuiranom objavom proces je manje rizičan i postaje jednostavniji za praćenje. Krajnji korisnici postaju zadovoljniji uslogom, jer imaju brzi uvid u razvoj i nova poboljšanja sustava.

7.4. Kontinuirana integracija i dostava aplikacije putem Jenkins-a i drugih srodnih alata

Jenkins je veoma popularan besplatni poslužitelj za kontinuiranu integraciju, dostavu i objavu baziran na Java platformi. Pruža stotine korisnih dodataka koji podržavaju izgradnju, implementaciju i automatizaciju bilo kojeg programskog proizvoda, bez obzira u kojem jeziku pisan. Omogućuje laganu i jednostavnu instalaciju i postavku platforme koja će automatizirati proces testiranja kroz primjenu kontinuirane integracije, dostave i objave. Funkcionira na principu stvaranja paketa projekta kojeg objavljuje, povezan sa izmjenama koje se evidentiraju kroz sustav za verzioniranje. Na slici 12 možemo vidjeti radnu ploču alata u kojem je lista svih kreiranih *pipeline-a*.



Slika 12. Jenkins – popularni alat za kontinuiranu integraciju i objavu (snimka zaslona)

Danas nam je dostupan čitavi niz veoma kvalitetnih alata koji omogućavaju kontinuiranu objavu, isporuku i objavu. Možemo reći da je Jenkins najpopularniji.

Ovo je popis popularnih alata za kontinuiranu integraciju i dostavu (Guru99, 2019):

- Jenkins - (poveznica na instalaciju: https://jenkins.io/download/)
- Microsoft Visual Stuido Team Services - (poveznica na stranicu: https://azure.microsoft.com/en-us/services/devops/?nav=min)
- Atlassian Bamboo – (poveznica na instalaciju: https://www.atlassian.com/software/bamboo)
- GitLab - (poveznica na instalaciju: https://about.gitlab.com/stages-devops-lifecycle/)
- Jetbrains TeamCity – (poveznica na instalaciju: https://www.jetbrains.com/teamcity/download/)
- Travis CI – (poveznica na instalaciju: https://travis-ci.org/)
- CircleCI – (poveznica na instalaciju: https://circleci.com/)

Principi kontinuirane integracije, dostave i objave se na neki način ugrađuju u razne modele razvoja programskog proizvoda. Popularnost i široka lepeza uporabe ovih principa dokazuje kako je ideja poboljšanja razvoja programskog proizvoda u zamahu industrije koja se ne libi brzo prihvatati dobra riješenja, a testiranje kao čimbenik u tim procesima jeste na neki način i cilj. U narednom odjeljku kroz primjer će biti prikazan detaljni proces kontinuirane integracije i objave kroz stvarnu aplikaciju.

8. Implementacija i testiranje WebRTC aplikacije

WebRTC je besplatni okvir (*engl. framework*) koji omogućava komunikaciju u realnom vremenu putem web preglednika (*engl. real-time comunications*). Uključuje osnovne elemente za kvalitetnu komunikaciju putem web aplikacija, kao što su audio i video komponente korištene za snimanje, reprodukciju i streaming zvuka i video zapisa. WebRTC komponente se implementiraju preko preglednika koristeći dostupni JavaScript API. Programeri koriste te komponente kako bi postigli razne multimedijalne mogućnosti u aplikacijama za web i mobitele. Svrha nastanka WebRTC-a je standardizacija multimedijalnih mogućnosti preglednika bazirajući se na API razini W3C (*engl. World Wide Web Consortium*).

Postoje mnoge prednosti korištenja WebRTC API-ja za izradu aplikacija, kao što je video i audio snimanje, video-chat i ostale multimedijalne funkcionalnosti. Ključne prednosti se ogledaju kroz to što su temeljne tehnologije za web, kao što su HTML, CSS, HTTP, TCP/IP protokoli, otvoreni i slobodni za daljnu nadogradnju i korištenje. Snaga te web zajednice jeste u tome što omogućuje svima da se uključe i doprinesu općoj koristi. Trenutno, ne postoji bolje besplatno rješenje za izradu visoko kvalitetnih aplikacija koje se služe komunikacijom u pregledniku. Ono što čini globalnu prednost je integracija sa raznim audio i video uređajima koji su prisutni kod mnogih koji se koriste Internet tehnologijama, drugim riječima krajnjim točkama internetske mreže. Sa sigurnosnog i mrežnog aspekta, WebRTC uključuje `NAT` i *firewall* tehnologiju koristeći razne podrške, kao što `je RTP-over-TCP`, `STUN` i `ICE`, te podrška za *proxy* poslužitelje.

WebRTC se gradi na podrškama za preglednike. On sažima signalizaciju nudeći signalni stroj koji izravno prenosi `PeerConnection`. Stoga, programerima se nudi mogućnost odabira protokola za njihove potrebe ograničavajući ih minimalno. Ovaj okvir ima dobro razrađenu i kompleksnu arhitekturu, koja u konačnici programerima omogućuje izradu bogatih multimedijalnih web aplikacija u stvarnom vremenu bez potrebe za raznim dodacima i instalacijama. Opća svrha je omogućiti svima snažne RTC aplikacije da rade jednako na više web i mobilnih preglednika, te na više platformi neovisno o razini mrežnih protokola. Arhitektura u osnovi bazira se na dva sloja: `WebRTC C++ API` namijenjen razvoju web preglednika i `WebRTC Web API` namijenjen web programerima za razvoj web aplikacija.

8.1. Pribavljanje korisničkih medija na pregledniku pomoću WebRTC-a

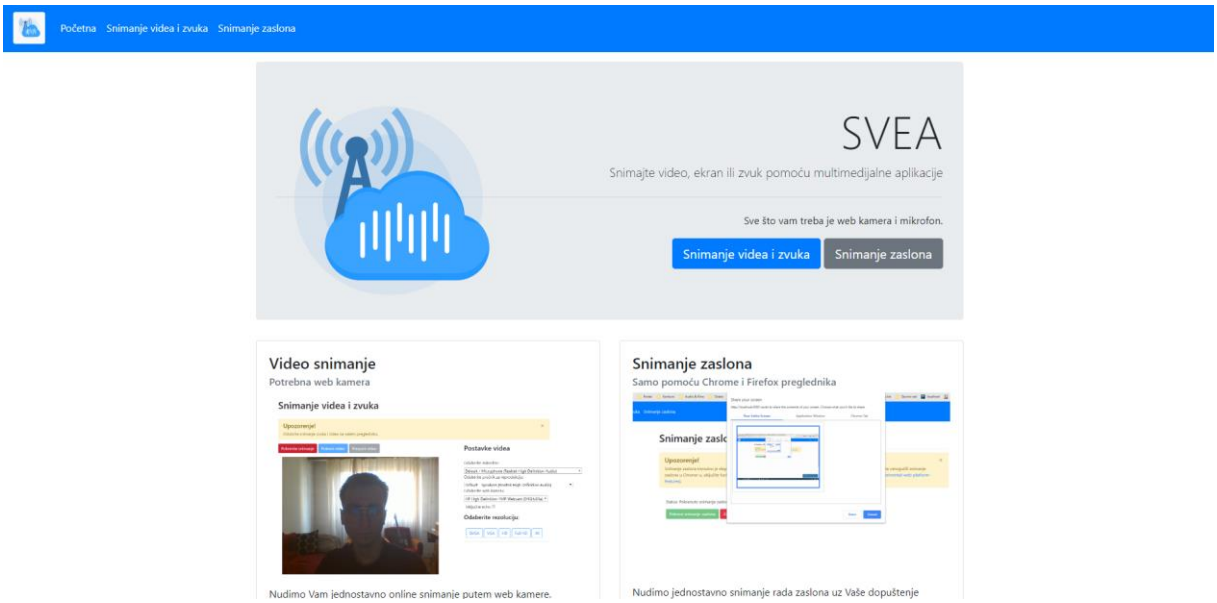
Pribavljanje korisničkih medija (*engl. getUserMedia*) predstavlja sinkronizirano strujanje medija poput web kamere, mikrofona i slično. Jedna od osnovnih uporaba WebRTC API-ja je kroz implementaciju `MediaStream` komponente koja dobiva pristup podatkovnim tokovima, kao što je web kamera i mikrofon. Svakako, korisnik mora odobriti korištenje tih uređaja. Preglednici na različite načine upravljaju dozvolama korisnika o korištenju tih medija.

Metoda `getUserMedia()` uzima parametar objekta `MediaStreamConstraints` i vraća varijablu koja se poziva u `MediaStream` objektu. Svaki `MediaStream` ima oznaku, koja je kriptirana u obliku lozinke. Nizovi `MediaStreamTracks` vraćaju metode `getAudioTracks()` i `getVideoTracks()`. `RTCPeerConnection` je WebRTC komponenta koja obrađuje komunikaciju podataka strujanjem između krajnjih točaka (*engl. peers*).

8.2. Funkcionalnosti aplikacije „SVEA“

Za potrebe istraživanja napravljena je web aplikacija koja nudi korisnicima mogućnost snimanja audio i video zapisa, preuzimanja, konfiguriranja postavki vanjskih medija, snimanje zaslona itd. Te funkcionalnosti objedinjuju većinu mogućnosti WebRTC okvira u jedinstvenu aplikaciju koja svodi više korisnih alata u jednu web aplikaciju. Naziv aplikacije „SVEA“ sastavljen je od prvih slova „aplikacija za snimanje videa i ekrana“. Sama aplikacija na neki način predstavlja hibrid poznatih funkcionalnosti koje možemo koristiti instalacijom dodataka za preglednike i slično. Svrha aplikacije da smanji potrebe za instaliranjem pregledničkih dodatka za standardne multimedijske potrebe. Izvorni kod aplikacije možete vidjeti na GitHub repozitoriju https://github.com/mijlucic/zavrsnirad.git. Na slici 13 je prikazana početna stranica aplikacije koja opisuje osnovne funkcionalnosti i ima poveznice na stranice tih funkcionalnosti.

Aplikacija zahtjeva dozvolu od korisnika za pristup izvorima zvuka i videa, kao što je korištenje web kamere i mikrofona. Ukoliko se koristi Chrome preglednik, za funkcionalnost snimanja zaslona potrebno je u pregledniku odobriti eksperimentalnu mogućnost snimanja zaslona, jer koristimo `getUserMedia` biblioteku, koja uz pomoć WebRTC-a omogućava konfigurabilno snimanje zaslona visoke rezolucije.

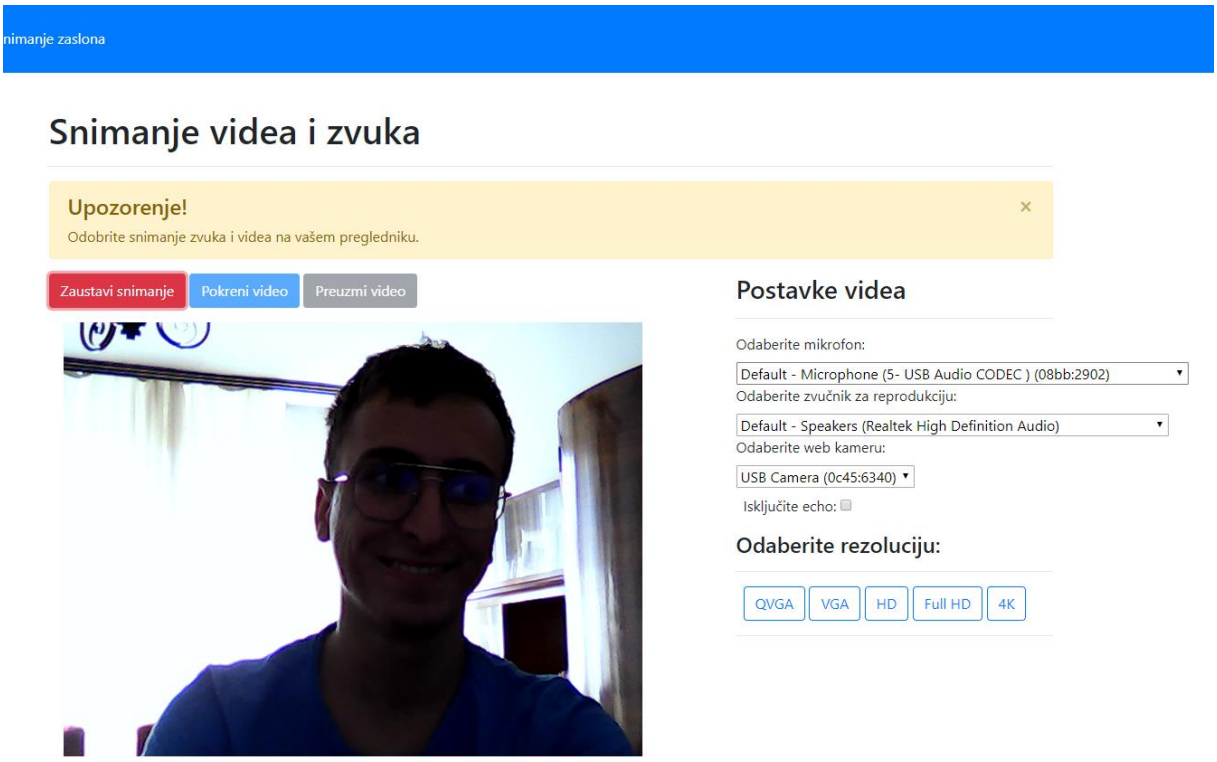


Slika 13. Slika ekrana web aplikacije koja služi za istraživanje testiranja multimedijalnih funkcionalnosti (snimak zaslona)

Aplikacija „SVEA“ nudi nekoliko funkcionalnosti kao što su:

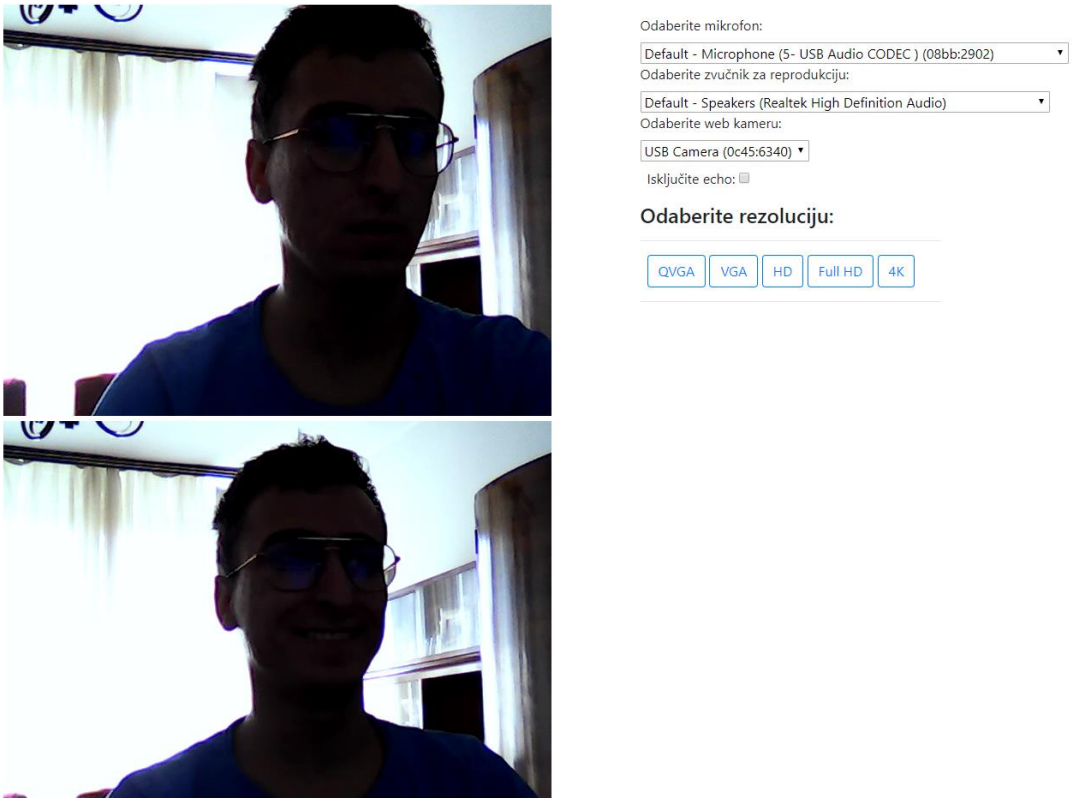
- odabir web kamere,
- odabir rezolucije videa za snimanje,
- odabir mikrofona za snimanje,
- odabir zvučnika za reprodukciju,
- snimanje video i audio snimka,
- preuzimanje video snimka,
- snimanje ekrana i
- pregled i preuzimanje snimka zaslona u .webm formatu.

Na slici 14 prikazana je stranica u kojem je realizirana funkcionalnost snimanja videa i zvuka.



Slika 14. Prikaz opcija za snimanje videa putem web kamere (snimak zaslona)

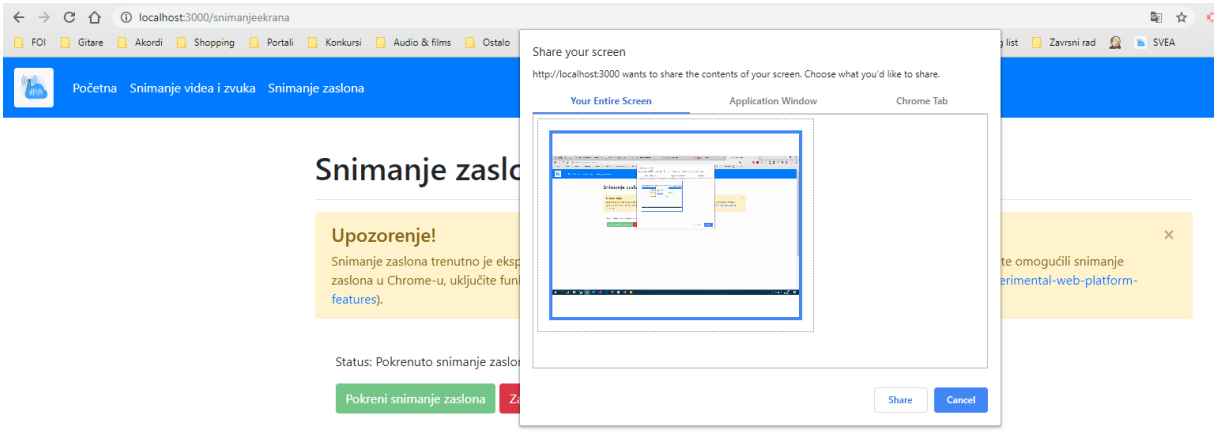
Prva funkcionalnost aplikacije je snimanje zvuka i videa pomoću web kamere, mikrofona i drugih sličnih medija. Kada se video i zvuk snime, moguće je prikazati snimljeni video odmah ispod žive kamere, te je moguće preuzeti video u .webm formatu, koji je namijenjen za prikaz videa na web i desktop aplikacijama. Konfigurabilnost se ogleda u mogućnosti biranja rezolucije, izvora zvuka i videa te odabir zvučnika za reprodukciju zvuka. Između ostalog, kako vidimo na slici 15 omogućene su specifične opcije kao što je uklanjanje mikrofonije i eha kod živog videa. Sve opcije kada se promjene, automatski ažuriraju živi video. Kada se snimanje videa zaustavi, klikom na gumb „Pokreni video“ ispod se prikaži snimljeni video kojeg je moguće preuzeti u .webm formatu.



Slika 15. Prikaz živog iznad i snimljenog videa ispod (snimak zaslona)

Kod realizacije ovih funkcionalnosti korištena je `getUserMedia()` metoda WebRTC API-ja za pribavljanje medija, a to su web kamere i mikrofoni. Aplikacija uspješno upravlja različitim vrstama web kamere i mikrofona priključenih za kompjuter, te prepoznaje osnovne tehničke karakteristike svake od tih uređaja.

Druga funkcionalnost aplikacije „SVEA“ jeste snimanje zaslona kojeg omogućuje web preglednik uz mogućnost odabira prozora, kartice ili samo dijela zaslona. Na slici 16 možemo vidjeti kako izgleda snimanje zaslona u Chrome pregledniku. On omogućuje snimanje cijelog zaslona, same kartice ili pak cijelog prozora. Klikom na gumb „Podijeli“, WebRTC snima zaslon i sprema ga u datoteku .webm formata. Kada se odobri snimanje ekrana, te nakon zaustavljanja snimanja, moguće je preuzeti tu datoteku.



Slika 16. Snimanje zaslona u aplikaciji „SVEA“ na pregledniku Chrome (snimak zaslona)

8.3. Testiranje interoperabilnosti WebRTC aplikacija

Jedna od važnih zadaća konzorcija u ostvarivanju misije WebRTC-a, jeste da se osigura često automatsko testiranje WebRTC API-ja. Kako se web preglednici često poboljšavaju i tehnologije izrade aplikacije ubrzano razvijaju, pojavljuje se potreba za učestalim testiranje.

Aplikaciju koja je izrađena u svrhu istraživanja, testirana je na operacijskom sustavu Windows 10 i na web preglednicima Chrome, Firefox i Edge. Prema mnogim istraživanjima u posljednjih 10 godina, uz neka odstupanja, vodeći preglednici u svijetu su Chrome i Firefox. Ta dva preglednika interoperabilni su, ali s obzirom na svoje specifičnosti i mala odstupanja od standarda kojeg propisuje W3C, ti preglednici zahtijevaju mali stupanj prilagodbe.

Tako da API za pribavljanje korisničkih medija `getUserMedia` (prema W3C) je u Chrome `webkitGetUserMedia`, a kod Firefox-a, `mozGetUserMedia`. Svakako, pri korištenju tih API-ja treba provjeriti da li su aktualni i imaju li trenutnu podršku za preglednik koji se želi koristiti. WebRTC adapter je dostupan preko *GitHub-a*, hostinga za verzioniranje koda. Taj adapter preko svoje biblioteke *polyfill* olakšava rad sa zastarjelim tehnologijama, te omogućuje programerima korištenje naziva prema W3C standardima.

Također, treba napomenuti da se sve te mogućnosti mogu koristiti i na mobilnim uređajima kroz nativni API. Za izradu aplikacija koja će imati tu podršku, potrebno je zadovoljiti neke posebne uvjete. Ovaj API nije u potpunosti zaživio i ima veoma mali broj korisnika. Čini se kako ovaj segment uveliko utječe na to da velike korporacije žele nametnuti svoja rješenja za multimedijalne aplikacije u realnom vremenu, bazirajući se više na efikasnosti i efektivnosti tehnologije nego na kvalitetnoj podršci na raznim uređajima. Tako

da velike softverske kompanije koje upravljaju programskim jezicima, nude svoja rješenja koja uspjevaju biti konkurentna WebRTC aplikacijama. Segment testiranja takvih aplikacija je veoma bitan u tom segmetu W3C konzorcij uspijeva osigurati kvalitetno kontinuirano testiranje ove tehnologije. Šira standardizacija tehnologije je uvijek u prednosti kada je u pitanju osiguranje kvalitete proizvoda.

Na primjeru aplikacije „SVEA“ utvrđen je različito upravljanje snimanja zaslona, što se ogleda i kroz postavke preglednika. U okviru tehničke realizacije, prilikom svakog pozivanja nekog medija, treba imati u obzir rad aplikacije na konvencionalnim preglednicima.

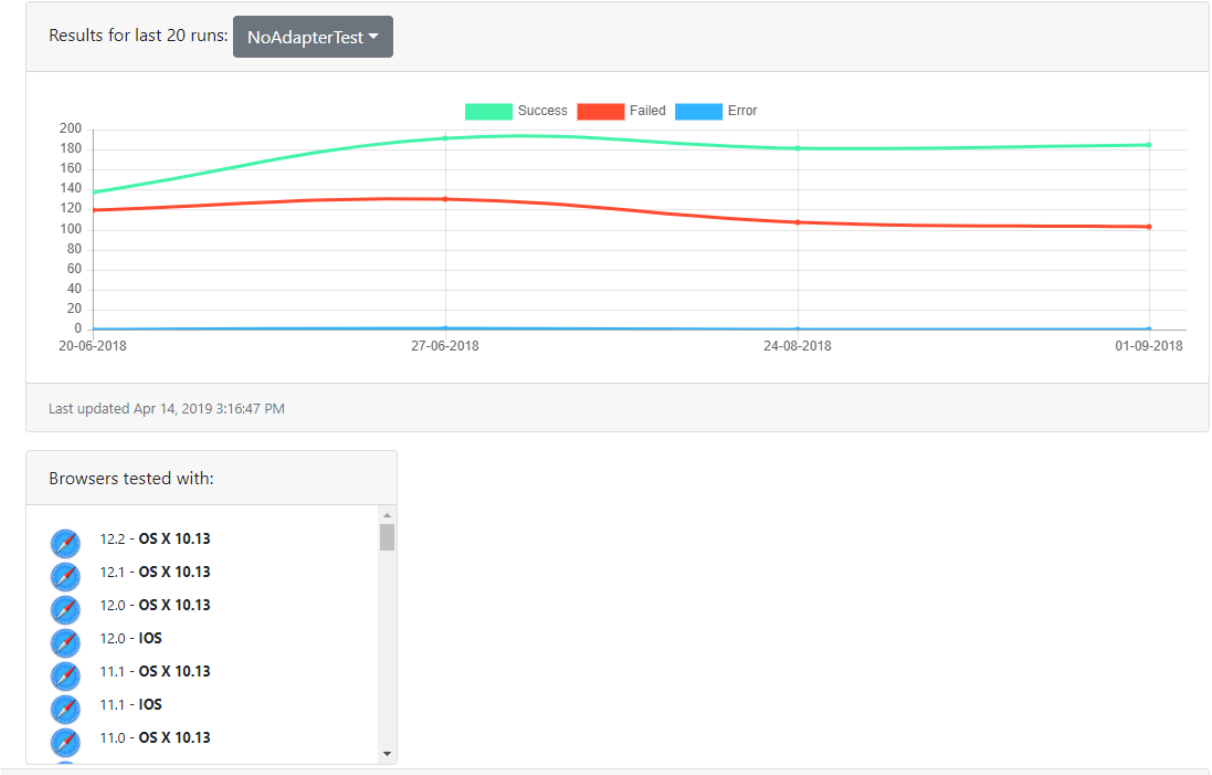
8.3.1. Testiranje interoperabilnosti pomoću alata Kite

KITE (*engl. Karoshi Interoperability Testing Engine*) je besplatni alat za testiranje interoperabilnosti između različitih preglednika. Interoperabilnost predstavlja mogućnost paralelnog korištenja aplikacije na dva ili više različitih preglednika. KITE alat uveliko olakšava testiranje interoperabilnosti, jer prikazuje putem grafičkog prikaza rezultate prema tipu tehničke funkcionalnosti. Alat je dizajniran tako da bude ponovno iskoristiv, jednostavan za korištenje i nadogradnju automatiziranog okruženja za testiranje. Za sada, on dobro radi na stabilnim verzijama najpopularnijih preglednika, kao što su Chrome, Firefox, Opera, Safari i Edge koji se pokreću na operacijskim sustavima Windows, Linux ili MacOS. Testiranje za mobilne uređaje još uvijek je u fazi razvoja. Nativne aplikacije sve više uzimaju maha, ali one zahtjevaju drugačiji arhitekturalni pristup testiranju, a za to je zadužena suradnja među velikim softverskim kompanijama i konzorciju koji je kreirao WebRTC.

Prema (WebRTC, 2018) postoje javno dostupni testovi putem alata Kite koji ispituju interoperabilnost po kriteriju interoperabilnosti preglednika na različitim operacijskim sustavima:

- `WebAudioInputTest` – provjerava integraciju `WebAudio API-ja`.
- `VP8SDPNegotiationTest` – provjerava sposobnost koristeći samo `vp8` kodek.
- `SimulcastTest` – provjerava može li preglednik koristiti simulcast za WebRTC.
- `RebroadcastTest` – provjerava sposobnost prijenosa primljenog strujanja na drugu međusobnu vezu.
- `NoAdapterTest` – provjerava interoperabilnost između preglednika bez adapter.js datoteke.

- `MultiStreamTest` – provjerava sposobnost korištenja višestrukih strujanja preko WebRTC-a.
- `MediaRecorderAPITest` – provjerava API za snimanje medija.
- `IceConnectionTest` – provjerava `ICEConnection` stanje između dva preglednika koji komuniciraju putem appr.tc biblioteke.
- `H264SDPNegotiationTest` – provjerava sposobnost preglednika za rad koristeći samo H-264 medijski kodek.
- `CanvasStreamToPcTest` – provjerava `MediaCapture()` funkciju na medijskom platnu (*engl. canvas*).



Slika 17. Rezultat posljednjih 20 izvršenja testa NoAdapterTest (snimak zaslona)

Na slici 17 možemo vidjeti snimak ekrana u kojem su prikazani rezultati testa `NoAdapterTest`, koji prikazuje rezultat interoperabilnosti preglednika bez adapter.js datoteke, koja se obično uključuje kod korištenja WebRTC-a u JavaScript jeziku. Test je izvršen u nekoliko navrata u 2018. godini. Graf prikazuje odnos broja uspješnih ispitivanja u odnosu na neuspješne po datumu izvršenja navedenog testa. Rezultat zadnjih deset pokretanja jeste 185 uspješnih i 103 neuspješna provjeravanja.

Proučavajući rezultate testiranja na KITE Dashboardu (KITE, 2019) možemo zaključiti kako se konzorcij brine da postoji kvalitetna provjera interoperabilnosti i pruža nadogradnju testova vezanih za specifičnosti multimedijskih protokola koji se koriste za pribavljanje i

strujanje medija. KITE Dashboard nudi desetine testova na stotine verzija različitih preglednika što je zaista značajan i pohvalan broj. Za svaki test se može vidjeti različiti tipovi grafova, te lista verzija preglednika na kojima je pokrenut test.

8.4. Kontinuirana integracija i dostava aplikacije „SVEA“

Kao što je navedeno u prethodnim poglavljima, moderne prakse sve više uključuju kontinuiranu integraciju, dostavu i objavu aplikacija kako bi objavljivanje promjena bilo jednostavnije i učinkovitije. Tako je u ovom odjeljku prikazan proces konfiguriranja i postavljanja alata Jenkins na lokalnom okruženju za kontinuiranu integraciju, dostavu i objavu aplikacije.

Korake uspostavljanja kontinuirane integracije i dostave aplikacije „SVEA“ možemo razložiti na:

- Instalaciju i konfiguraciju Jenkins-a na lokalnom okruženju.
- Povezivanje sa sustavom za verzioniranje (Git).
- Postavljanje testnog i produkcijskog okruženja.
- Postavljanje radnika.
- Postavljanje pipeline.
- Pokretanje i upravljanje Jenkins radnicima.

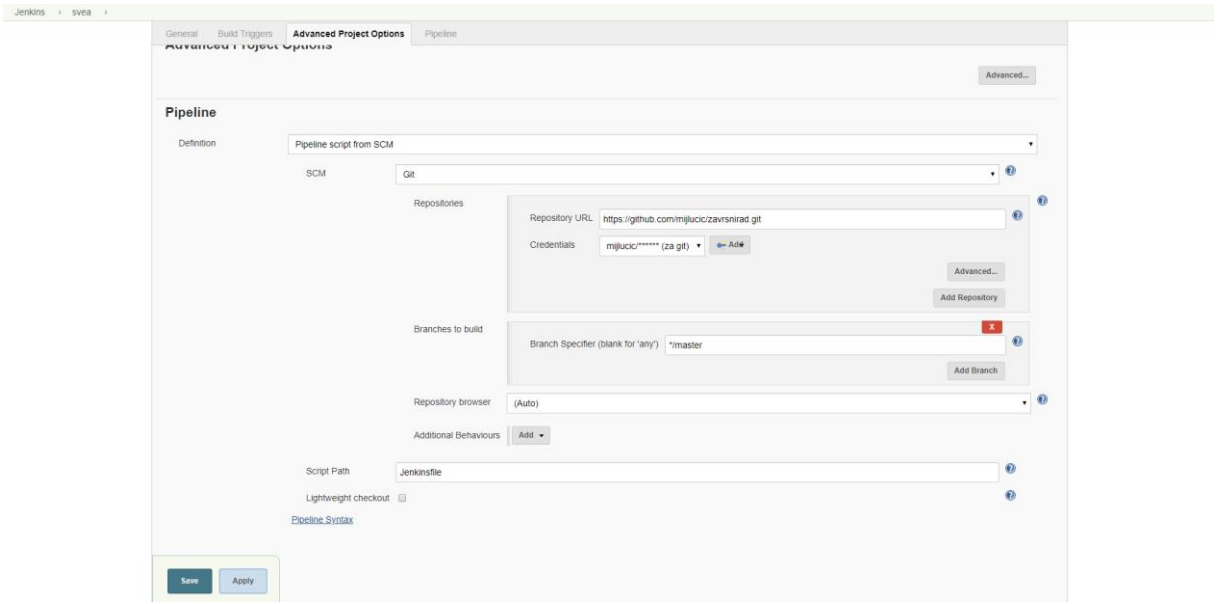
Svakako, prvi korak je preuzeti i instalirati Jenkins na operacijski sustav, u ovom slučaju Windows 10. Na lokalnom okruženju Jenkins aplikacija je instalirana i postavljena na URL http://localhost:8080/. Najprije, potrebno je pokrenuti Jenkins komandom sa administratorskim pravima `java -jar jenkins.war` na lokaciji `C:\Program Files (x86)\Jenkins`. Za uspješnu konfiguraciju, potrebno je podesiti globalne varijable okruženja na Windows-u na način da se definiraju ispravne putanje Java SE, NodeJs-a, Git-a i drugih eksternih programa koje koristi Jenkins za obavljanje zadataka. Prilikom prvog pokretanja komande, Jenkins nudi po koracima osnovne postavke okruženja kako bi se kreirao i postavio prvi radnik i *pipeline*.

Drugi korak jeste instalacija sustava za verzioniranje, ukoliko ne postoji. Zatim, treba podesiti korisnički račun, te instalirati plugin za Git na Jenkins plugin manager-u. Također, potrebno je i druge dodatke instalirati i konfigurirati. Za osnovno korištenje, dovoljno je instalirati preporučene dodatke koje prikazuje Jenkins. Jedan od zanimljvih dodataka je Slack, koji omogućuje povezivanje poznatog alata za brzu komunikaciju u projektnim timovima, tako što korisnike obavještava o izvršavanju testova na Jenkins-u. Povezivanje

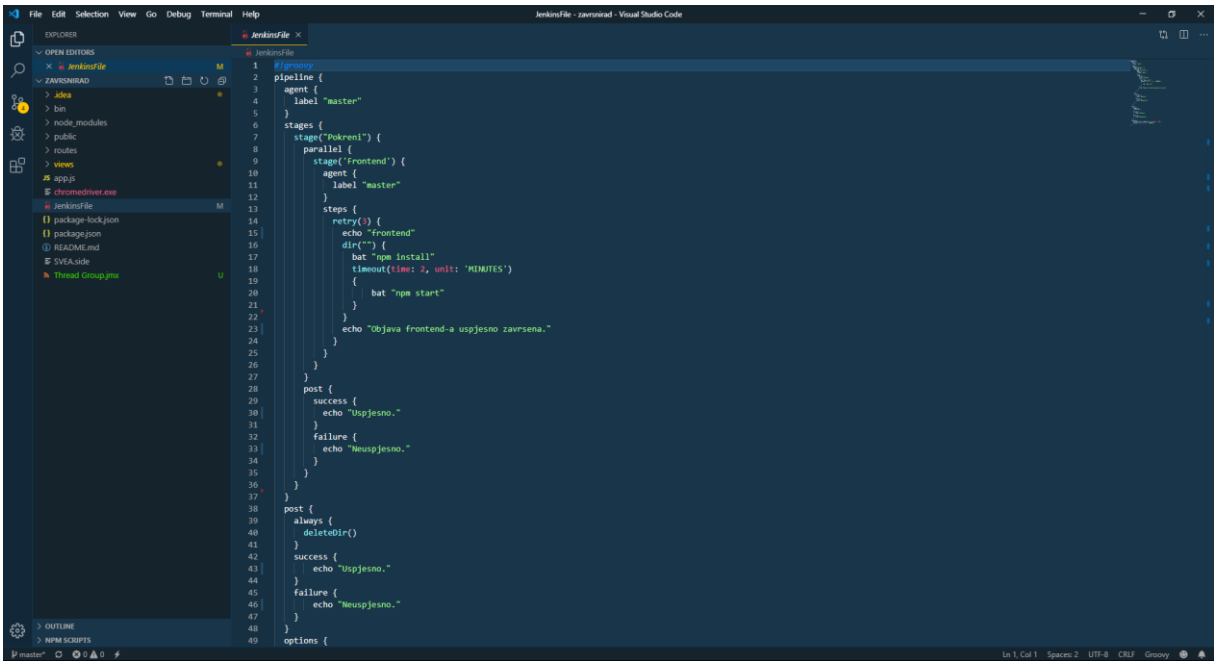
Slack-a i Jenkins-a može u početku biti komplicirano. No, ostvaruje cilj pravovremenog obavještenja korisnika o rezultatima kontinuirane integracije i objave.

Kada se pokrene aplikacija putem preglednika, na početku se dodaje novi radnik. Početna konfiguracija već ima glavnog radnika, koji u suštini može raditi posao izvršenja na istom okruženju gdje je instaliran Jenkins. Radi se o radniku pod nazivom „`master`“, koji po pokrenutom Java servisu, pokreće izvršenje testa.

Najvažniji i najspecifičniji korak jeste definiranje pipeline-a. U slučaju aplikacije „SVEA“, konfiguriran je *pipeline*, koji u osnovi gleda Jenkinsfile koji je dostupan u paketu kojeg prepoznaje sustav za verzioniranje. Na slici 18 vidimo konfiguraciju Git putanje, autentifikaciju i određeni `branch`, koji se prati kod objavljenih promjena, kako bi radnik izvršio testove.

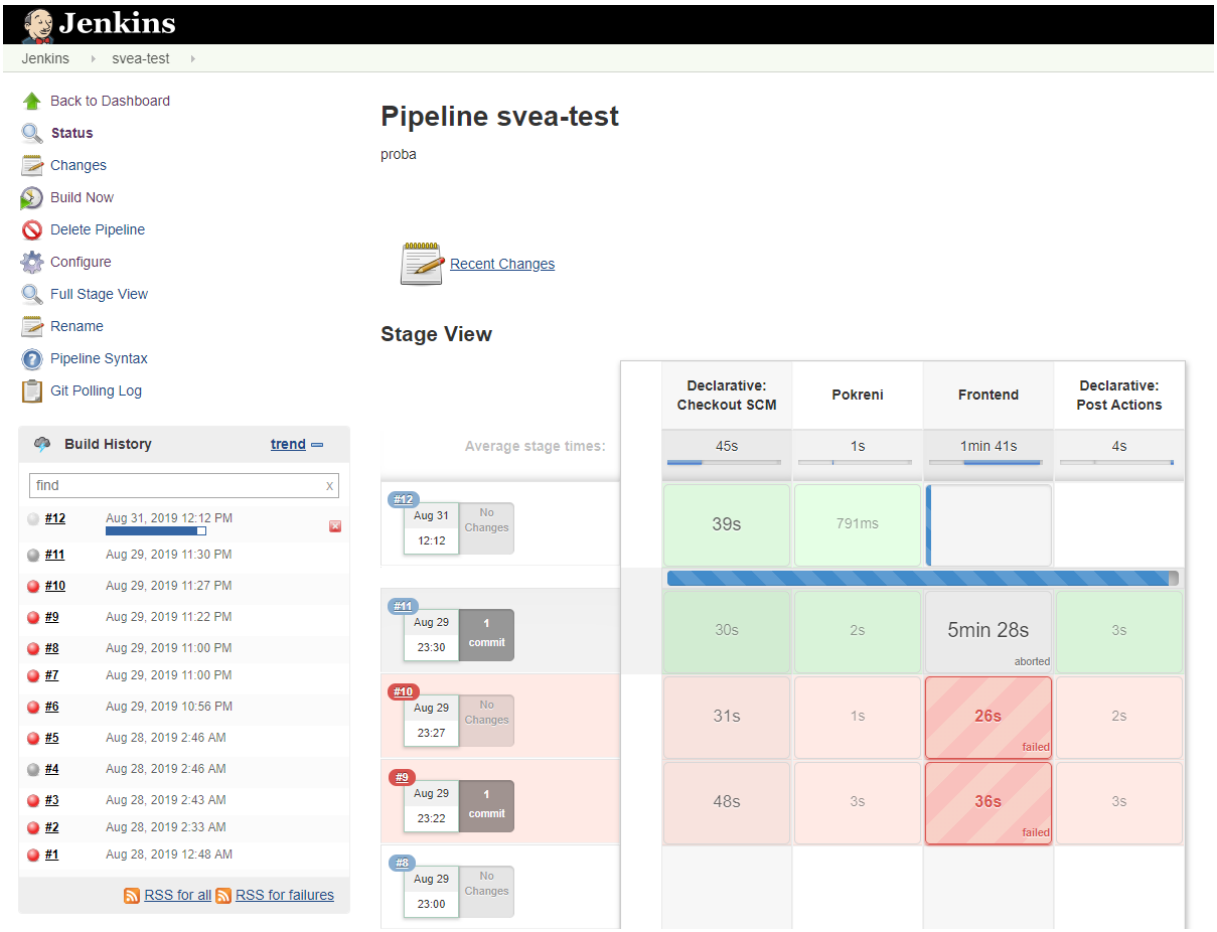


Slika 18. Konfiguracija pipeline za aplikaciju „SVEA“ (snimak zaslona)



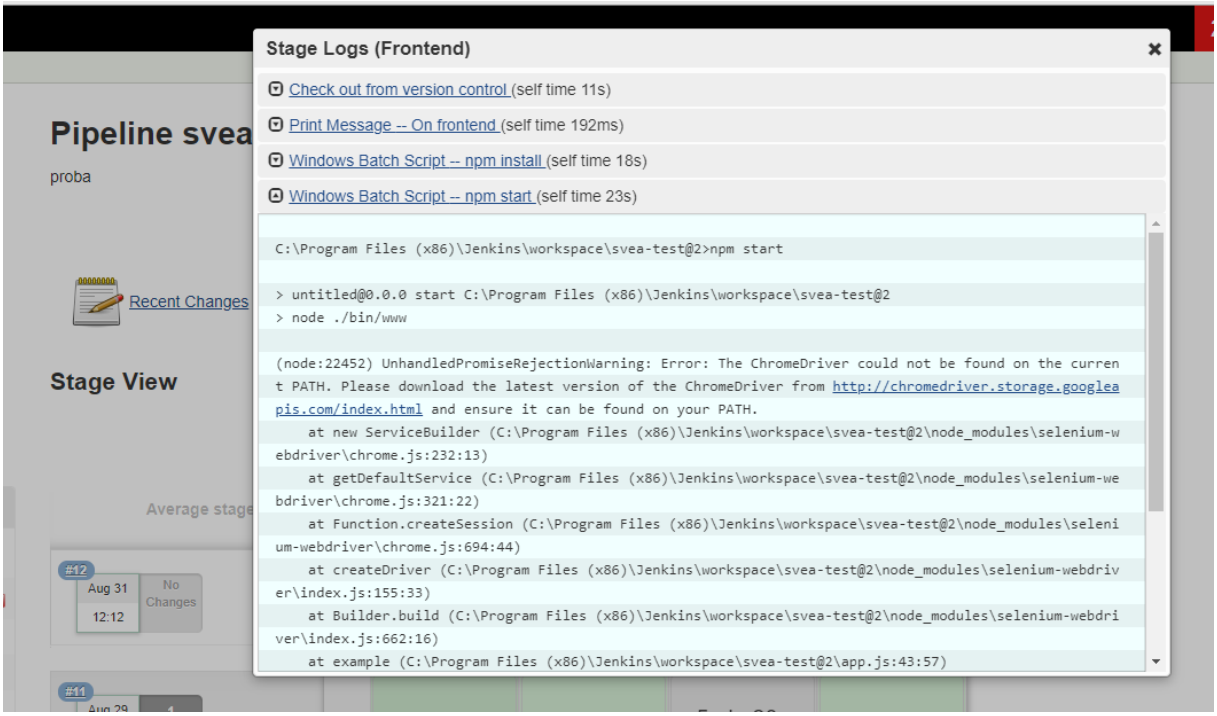
Slika 19. Jenkinsfile koraci za izvšenje pipeline-a (snimak zaslona)

U datoteci `Jenkinsfile` definiraju se koraci izvršenja. Kako vidimo na slici 19, prvi korak je prepoznavanje promjena u `master branch-u` sustava za verzioniranje. Kada se utvrdi koje promjene su izvršene od posljednjeg izvršenja, nastupa korak izvršenja komandi za objavu frontend-a aplikacije. U ovom slučaju radi se o Node.js aplikaciji. Stoga, komanda za instaliranje node dodataka se prva izvršava `npm install`. Nakon toga slijedi izvršenje komande `npm start`. Postavljen je *timeout* od 2 sekunde za prekidanje pokrenutog programa, jer ova komanda postavlja aplikaciju pokrenuto sve dok nema definiranog prekida. Za potrebe demonstracije, aplikacija se gasi nakon dvije minute. Primjer jednog uspješnog pokretanja (*engl. build*) možemo vidjeti na slici 20.



Slika 20. Prikaz izvršavanja pipeline kontinuirane integracije (snimak zaslona)

Dnevnik izvršenja, odnosno konzolni dnevnik prikazuje detaljno izvršavanje svih komandi i rezultata, te ispisuje eventualne pogreške pri izvršavanju testa. Na slici 21 vidimo upozorenja i pogreške kod pokretanja komande za pokretanje Node aplikacije.



Slika 21. Prikaz dnevnika izvšenih komandi kod pokretanja aplikacije (snimak zaslona)

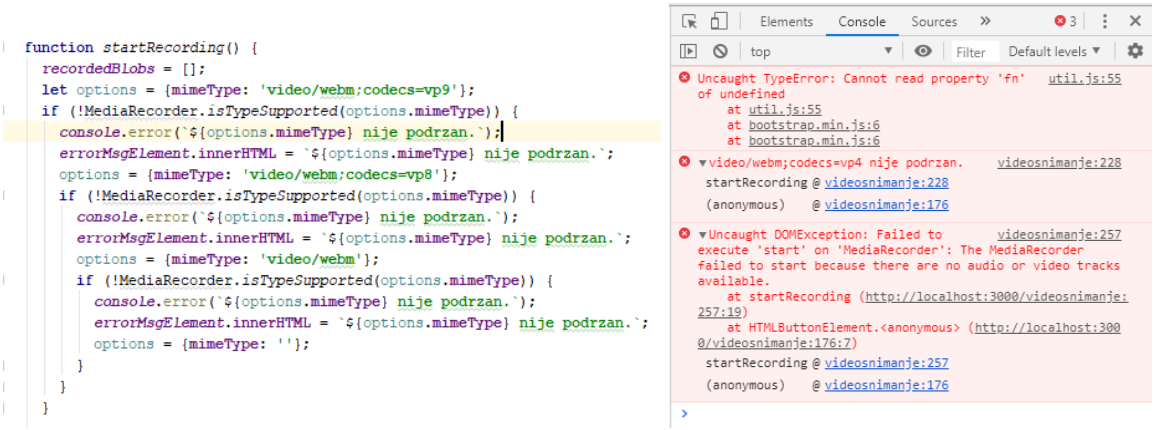
Kao što je navedeno, Jenkins prepoznaje promjene i vrši integraciju i objavu. Kada otvorimo pipeline možemo vidjeti prikaz posljednjih nekoliko pokretanja, vrijeme i datum izvršenja, vrijeme po koracima izvršenja, nazive branch-eva itd. Preporučuje se pratiti *pipeline* kod svake bitnije promjene u kodu. Na kraju moguće je izvesti rezultate pokretanja kroz razne izvještaje koji se mogu izvesti u neku datoteku. Tu se ogleda prednost ovog alata.

8.5. Jedinično testiranje aplikacije u pregledniku Chrome

Koristeći *Developer tools* opcije u pregledniku možemo jednostavno pisati jedinične testove pojedinih komponenti aplikacije. U slučaju multimedijalne aplikacije kao što je „SVEA“, gdje ne postoje web servisi za obradu podataka na udaljeni poslužitelj i baze podataka, nema potrebe vršiti integracijsko testiranje kada nema web servisa na bazu podataka. No, kako se radi o Node.Js aplikaciji, GET zahtjevi postoje kod osnovnih funkcionalnosti aplikacije, kao što je snimanje videa, zvuka i zaslona. Karakter aplikacije je takav da je potrebno koristiti sve moguće načine izvještavanja o radu pojedinih funkcija u kodu kada je u pitanju pokretanje i konfiguracija snimanja putem WebRTC biblioteke. Jedan od jednostavnijih i sigurnijih načina jeste direktno zapisivanje pogrešaka i uspješnog izvršavanja pojedinih akcija.

Na primjeru ove aplikacije, korištene su beneficije konzole web preglednika za praćenje uspješnosti izvršenja pojedinih komponenti. Kod većine metoda za pokretanje

medija, postavljen je bazični sustav za praćenje pogreški putem konzole. Recimo, u metodi `startRecording()` koja služi za pokretanje kamere i provjeru ispravnosti medija, postavljene su linije koda `console.error()` za prikaz nepodržanih tipova medija. Primjer koda u kojem se poziva `console.error()` metoda na lijevoj strani slike 22, dok je na desnoj strani slike rezultat u konzoli Chrome preglednika.



Slika 22. Dio koda za pokretanje video snimanja lijevo i rezultat testiranja u konzoli preglednika desno (snimci zaslona)

Široka je lepeza mogućnosti koje pruža konzola web preglednika, a ponajviše ogledajući se kroz DOM strukturu web aplikacije. Ovaj primjer u kodu prikazuje učestalo korištenje konzole kod izvršavanja ključnih funkcija aplikacije. Svakako, ovaj tip testiranja služi programeru za što ranije otkrivanje konkretnih pogrešaka.

8.6. Testiranje performansi aplikacije pomoću JMeter alata

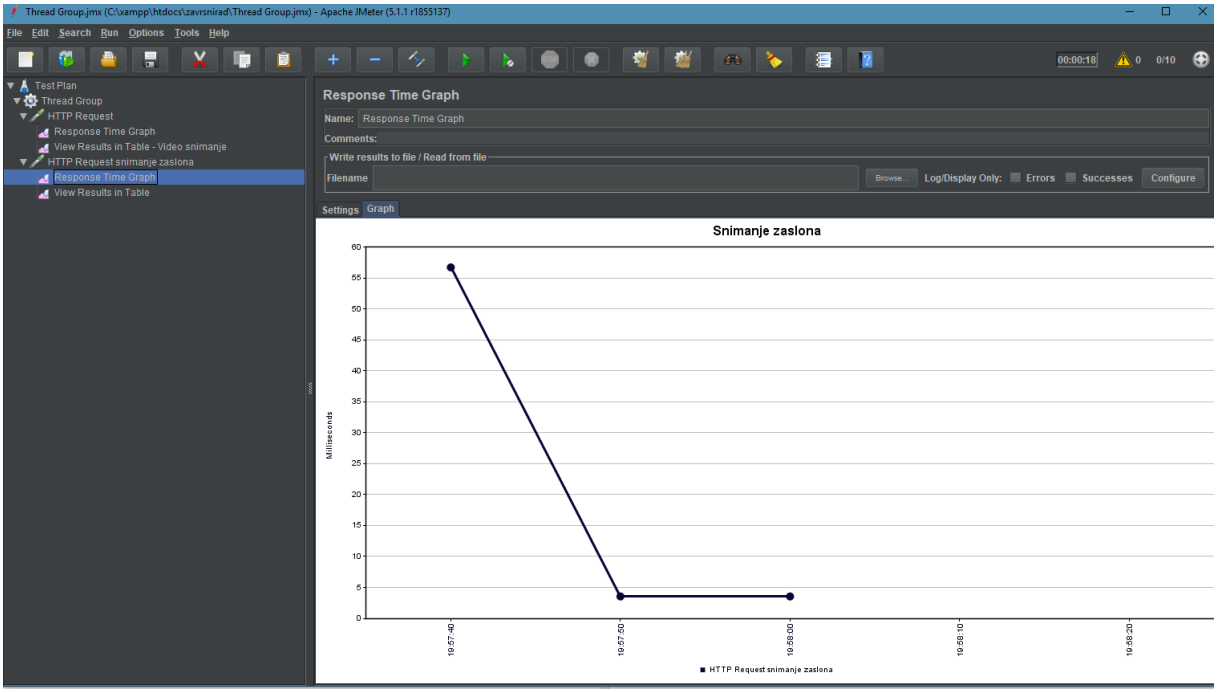
Testiranje opterećenja (*engl. load testing*) je vrsta testiranja učinkovitosti koja pokazuje ponašanje funkcionalne jedinice ili sustava s povećanim opterećenjem, kao što je npr. povećani broj paralelnih korisnika ili transakcija na bazu podataka, a sve s ciljem da se utvrdi ponašanje sustava pod tim povećanim opterećenjem. Testiranje opterećenja možemo povezati i sa pojmovima testiranja performansi i stresnog testiranja.

Apache JMeter je popularni alat za testiranje web aplikacija, koji se većinom koristi za testiranje opterećenja, analizu i mjerenje performansi raznih aplikacija, ponajviše aplikacija za web. Također, koristi se kod analize rada baze podataka, testiranja rada FTP protokola, LDAP, JMS, HTTP zahtjeva i drugih specifičnih protokolarnih zahtjeva na poslužitelj i slično. Ovaj alat podržava promjenljivu parametrizaciju, konfiguracijske varijable i generira veliki broj različitih izvještaja o izvršavanju komponenti ili sustava. Za instaliranje i korištenje JMeter-a

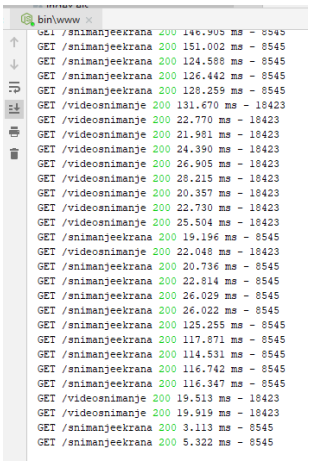
potrebno je imati instaliran Java JDK. JMeter GUI se pokreće otvaranjem `jmeter.bat` datoteke. Moguće je koristiti konzolu i CLI za pokretanje testova opterećenja.

Cilj testiranja opterećenja WebRTC aplikacije jeste prikazati parametre odaziva sustava kada sa više preglednika se pokreću zahtjevi za snimanje videa i zvuka. Kako se radi o GET zahtjevima na Node lokalni poslužitelj, u kreiranju testa treba postaviti dva `HTTP request` na http://localhost:3000/videosnimanje i http://localhost:3000/snimanjeekrana.

Na primjeru aplikacije „SVEA“ imamo dva glavna zahtjeva koje smo postavili na alatu JMeter. Kao rezultat, alat prikazuje grafikon i prikaz redaka u tablici izvršenja zahtjeva. Konfiguracija ovih testova je takva da se simulira „napad“ 10 različitih korisnika vremenske razlike od 2 sekunde, tako da grafikon prikazuje na istu komponentu po svakoj stotinci vremena. Dakako, program izvršava te akcije kroz više thread-ova. Na slici 23 vidimo rezultate kroz grafikon, dok na slici 24 prikazan je dnevnik izvršenja GET zahtjeva sa vremenskim odazivom poslužitelja.

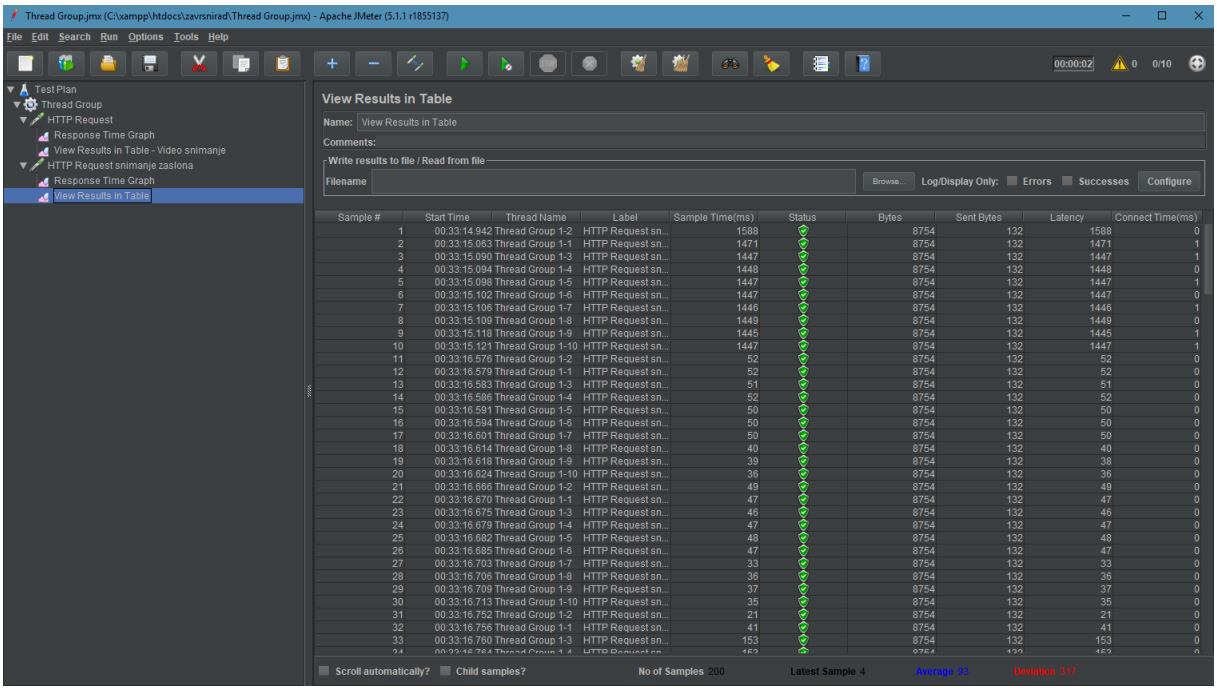


Slika 23. Grafik rezultata testiranja opterećenja GET zahtjeva snimanja zaslona u JMeter alatu (snimak zaslona)



Slika 24. Prikaz dnevnika izvršenja zahtjeva od 10 korisnika u isto vrijeme i odaziva aplikacije (snimak zaslona)

Na slici 25 vidimo prikaz tablice sa izvršenim zahtjevima 10 korisnika kod snimanja zaslona. U tablici su prikazani stupci o vremenu kašnjenja, statusu, vremenu, oznaci i drugim tehnički podaci o svakom zahtjevu. Po rezultatima možemo zaključiti kako su uspješno izvršeni svi zahtjevi sa relativno malo kašnjenja i uporabe memorije računala, bez obzira na povećano opterećenje.



Slika 25. Rezultati testiranja snimanja zaslona prikazani na tablici JMeter alata (snimak zaslona)

9. Zaključak

Cilj ovog završnog rada bio je dati sveobuhvatan prikaz praksi kod testiranja programskih proizvoda, pritom obuhvatiti sve relevantne i aktualne metode koje su prisutne u realnom IT sektoru. Dodatni cilj ovog rada bio je pružiti kvalitetan i sveobuhvatan uvod u sve ono što podrazumijevamo za testiranje programskih proizvoda. Specifičnost rada jeste u tome da pruža širok pojmovni opseg kada je u pitanju testiranje. Pružen je uvid u međusobnu povezanost metoda i praksi koji su standardizirani i poznati kao rasprostranjeni u realnom sektoru.

Svjedoci smo vremena kada se prakse često mijenjaju, preispituju i zamjenjuju potpuno novim do sada nepoznatim praksama. Takav je slučaj i kod testiranja programskih proizvoda. Period koji nastupa ponudit će nam nova rješenja i svakako više staviti u fokus ovu fazu životnog ciklusa programskog proizvoda. Ono što je značajno jeste da se pojačava svijest kod programera i svih dionika u stvaranju i održavanju sustava kada je u pitanju važnost i uloga kontinuiranog testiranja u životnom ciklusu programskog proizvoda.

Rezultati istraživačkog djela rada pokazuju kako su alati za testiranje veoma dobro smišljeni i razrađeni, te kako daju rezultate koji zaista mogu biti relevantni u cjelokupnom viđenju testiranja programskih proizvoda. WebRTC multimedijalne aplikacije su specifične po svojoj arhitekturi te imaju određene nedostatke u prikazivanju uspješnosti određenih funkcija, jer se ne radi o klasičnim HTTP zahtjevima na bazu podataka, nego se koriste posebni protokoli za izvršavanje multimedijskih funkcionalnosti. Dakako, postoje i ograničenja u istraživanju ovakve široke teme, a ona se ogledaju u tome da su prakse testiranja iako mnogo unificirane, uveliko ovisne o skupu tehnologija i alata koji se koriste u razvoju programskih proizvoda. Specifičnosti programskih jezika i okruženja, a samim time i metoda razvoja uveliko određuju smjer i format testiranja. Stoga je testiranje aplikacije „SVEA“ bilo definirano prema specifičnosti NodeJS skriptnog programskog jezika.

Ovim radom stvorena je osnova za istraživanje pojedinih praksi, osobito suvremenih, kao što je testiranje u kontekstu kontinuirane integracije, dostave i objave. Zasigurno, predmet istraživanja u sljedećih nekoliko godina biti će tesitranje u kontekstu suvremenih modela, kao što su DevOps i agilni razvoj. Svaka od razina testiranja je dovoljno široka i kompleksna da zahtjeva poseban osvrt i detaljnu analizu, jer praksa pokazuje kako se te razine proširuju u funkcionalnosti testiranja, te pokrivaju širi spektar testnih slučajeva. Ti trendovi već sada uveliko pomjeraju cjelokupnu industriju u neki drugi smjer. Testiranje postaje sve važnije u procesu stvaranja i održavanja programskog proizvoda, a samim time i povećava cijenu usluga testiranja.

Popis literature

Alan B. Johnston, D. C. (2014). *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition.* St. Louis, MO, SAD: Digital Codex LLC, 3. izdanje.

ASQ. (kolovoz 2019). *SOFTWARE QUALITY ENGINEER CERTIFICATION CSQE*. Dohvaćeno iz ASQ: https://asq.org/cert/software-quality-engineer

Balakrishnan, H. M. (2017). *Trends in Software Testing.* Singapore: Springer.

Banjavčić, J. (2017). *Metode vrednovanja kvalitete programskih proizvoda.* Pula: Sveučilište Jurja Dobrile u Puli.

Bazzana, G. Y. (2015 - 2016.). *ISTQB Worldwide Software Testing Practices Report.* Brisel, BEL: International Software Testing Qualifications Board [ISTQB].

Bushnev, Y. (14. Studeni 2017). *Top 15 UI Test Automation Best Practices You Should Follow*. Dohvaćeno iz BlazeMeter: https://www.blazemeter.com/blog/top-15-ui-test-automation-best-practices-you-should-follow

Certifications, S. (kolovoz 2019). *CSQA*. Dohvaćeno iz Software Certifications: https://www.softwarecertifications.org/csqa/

Certifications, S. (kolovoz 2019). *CSTE*. Dohvaćeno iz Software Certifications: https://www.softwarecertifications.org/cste/

Clokie, K. (2017). *A Practical Guide to Testing in DevOps.* Leanpub.

Contributor. (4. Srpanj 2017). *13 Best Practices of Successful Software Testing Projects*. Dohvaćeno iz DevOps: https://devops.com/13-best-practices-successful-software-testing-projects/

Dan, & Ashby, D. (17. lipanj 2019). *Continuous testing in DevOps*. Dohvaćeno iz Dan Ashby: https://danashby.co.uk/2016/10/19/continuous-testing-in-devops/

FOI. (studeni 2017). *Preporuke citiranja i referenciranja primjenom stila refernciranja APA.* Varaždin: Sveučilište u Zagrebu, Fakultet organizacije i informatike.

Fundamentals, S. T. (n.d.). *Software Quality Control*. Dohvaćeno iz Software Testing Fundamentals: http://softwaretestingfundamentals.com/software-quality-control/

Ghahrai, A. (2. prosinac 2018). *Testing In DevOps - Strategies, Tools and Practices*. Dohvaćeno iz Testing excellence: https://www.testingexcellence.com/testing-in-devops/

Guru99. (kolovoz 2019). *Guru99*. Dohvaćeno iz 15 Best Jenkins Alternatives in 2019: https://www.guru99.com/jenkins-alternative.html

Hrgarek, N. (2013). Testiranje i kvaliteta softvera. Varaždin.

Isaac, A. (28. studeni 2018). *DevOps in 3 sentences*. Dohvaćeno iz Dev: https://dev.to/ashokisaac/devops-in-3-sentences-17c4

ISO/IEC JTC1/SC7, W. G. (2014). *The International Software Testing Standard*. Dohvaćeno iz Software Testing Standard: http://www.softwaretestingstandard.org/

ISTQB. (2016). *Certification path*. Dohvaćeno iz ISTQB: https://www.istqb.org/certification-path-root.html

Jorgensen, P. C. (2013). *Software Testing: A Craftsman's Approach, Fourth Edition.* Boca Raton, FL, SAD: Auerbach Publications.

Kazeeva, A. (14. Ožujak 2018). *10 Best Open Source Test Automation Frameworks for Every Purpose*. Dohvaćeno iz DZone: https://dzone.com/articles/10-best-open-source-test-automation-frameworks-for

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices.* New Jersey, USA: Pearson Education.

Maynard, C. (n.d.). *Software Testing for Continuous Delivery*. Dohvaćeno iz Atlassian: https://www.atlassian.com/continuous-delivery/software-testing

Myers, G. J. (2004). *The Art of Software Testing.* Canada: John Wiley & Sons, Inc.

Padmini, C. (2006). *Beginners Guide To Software Testing.* Lulu Enterprises.

Pezze, M., & Young, M. (2008). *Software testing and analysis: Process, Principles, and Techniques.* Hoboken, USA: John Wiley & Sons Inc.

Pinterest. (kolovoz 2019). *Agile DevOps*. Dohvaćeno iz Pinterest: https://www.pinterest.com/pin/218002438199346238/

Rasmusson, J. (2017). *The Way of the Web Tester.* Raleight, NC, SAD: The Pragmatic Bookshelf.

Ristic, D. (2015). *Learning WebRTC.* Birmingham, VB: Packt Publishing.

Roy, A. (18. lipanj 2019). *Test Verification and Validation in Website Testing*. Dohvaćeno iz DZone: https://dzone.com/articles/test-verification-vs-validation-in-website-testing

Selenium Documentation. (13. Kolovoz 2018). Dohvaćeno iz SeleniumHQ: https://www.seleniumhq.org/docs/

Sergiienko, A. (2015). *WebRTC Cookbook.* Birmingham, VB: Packt Publishing.

Shanmugam, S. (19. lipanj 2019). *What Is Continuous Deployment? Everything you need to know*. Dohvaćeno iz DevOpsZone: https://dzone.com/articles/what-is-continuous-deployment-everything-you-need

Šljanić, S. (7. prosinac 2016). *Uspesan QA tim - Moje iskustvo u QA*. Dohvaćeno iz TNation: https://www.tnation.eu/blog/uspesan-qa-tim-moje-iskustvo-u-qa/

WebRTC. (2016). *Architecture*. Dohvaćeno iz WebRTC: https://webrtc.org/architecture/

WebRTC, K. (kolovoz 2019). *KITE Dashboard*. Dohvaćeno iz KITE: https://kiteboard.herokuapp.com/dashboard

Werner, M. (2019). Automated Testing. *DZone*.

Popis slika

Slika 1. Klasični pristup organizacije testiranja (Hrgarek, 2013) 5
Slika 2. Organizacija testiranja u agilnom pristupu – SCRUM (Hrgarek, 2013) 6
Slika 3. Životni ciklus testiranja programskih proizvoda (Software Testing Fundamentals, 2017). 12
Slika 4. Vodopadni model životnog ciklusa proizvoda (autorski rad) 13
Slika 5. Kontinuirano testiranje u DevOps modelu (Isaac, 2018) 15
Slika 6. Usporedba etapa razvoja kod suvremenih modela (Shanmugam, 2019) 16
Slika 7. Piramida automatskog testiranja (autorski rad) 22
Slika 8. Postman - alat za testiranje API-ja (snimak zaslona) 28
Slika 9. Prikaz procesa kontinuirane integracije (autorski rad) 32
Slika 10. Prikaz procesa kontinuirane dostave ili isporuke (autorski rad) 34
Slika 11. Prikaz procesa kontinuirane objave (autorski rad) 34
Slika 12. Jenkins – popularni alat za kontinuiranu integraciju i objavu (snimka zaslona) 35
Slika 13. Slika ekrana web aplikacije koja služi za istraživanje testiranja multimedijalnih funkcionalnosti (snimak zaslona) 39
Slika 14. Prikaz opcija za snimanje videa putem web kamere (snimak zaslona) 40
Slika 15. Prikaz živog iznad i snimljenog videa ispod (snimak zaslona) 41
Slika 16. Snimanje zaslona u aplikaciji „SVEA“ na pregledniku Chrome (snimak zaslona) ...42
Slika 17. Rezultat posljednjih 20 izvršenja testa NoAdapterTest (snimak zaslona) 44
Slika 18. Konfiguracija pipeline za aplikaciju „SVEA“ (snimak zaslona) 46
Slika 19. Jenkinsfile koraci za izvšenje pipeline-a (snimak zaslona) 47
Slika 20. Prikaz izvršavanja pipeline kontinuirane integracije (snimak zaslona) 48
Slika 21. Prikaz dnevnika izvšenih komandi kod pokretanja aplikacije (snimak zaslona) 49
Slika 22. Dio koda za pokretanje video snimanja lijevo i rezultat testiranja u konzoli preglednika desno (snimci zaslona) 50
Slika 23. Grafik rezultata testiranja opterećenja GET zahtjeva snimanja zaslona u JMeter alatu (snimak zaslona) 51
Slika 24. Prikaz dnevnika izvršenja zahtjeva od 10 korisnika u isto vrijeme i odaziva aplikacije (snimak zaslona) 52
Slika 25. Rezultati testiranja snimanja zaslona prikazani na tablici JMeter alata (snimak zaslona) 52

Popis tablica

Tablica 1. Razlike između verifikacije i validacije 7