UNIVERZITET CRNE GORE
ELEKTROTEHNIČKI FAKULTET

Sanja Bjelanović

ALATI ZA TESTIRANJE I OTKRIVANJE GREŠAKA U WEB APLIKACIJAMA: STUDIJA SLUČAJA UPOTREBA SELENIUM ALATA

- master rad -

Podgorica, 2024.

UNIVERZITET CRNE GORE
ELEKTROTEHNIČKI FAKULTET

Sanja Bjelanović

ALATI ZA TESTIRANJE I OTKRIVANJE GREŠAKA U WEB APLIKACIJAMA: STUDIJA SLUČAJA UPOTREBA SELENIUM ALATA

- master rad -

Podgorica, 2024.

PODACI I INFORMACIJE O STUDENTU

Ime i prezime:	**Sanja Bjelanović**
Datum i mjesto rođenja:	04.10.1998. godine, Berane
Naziv završenog osnovnog studijskog programa i godina završetka studija:	Studijski program Primijenjenog računarstva, Elektrotehnički fakultet, Univerzitet Crne Gore, 180 ECTS kredita, 2021. godine.

INFORMACIJE O MASTER RADU

Naziv master studija:	Master studije primijenjenog računarstva
Naslov rada:	Alati za testiranje i otkrivanje grešaka u Web aplikacijama: Studija slučaja Upotreba Selenium alata
Fakultet na kojem je rad odbranjen:	Elektrotehnički fakultet

UDK, OCJENA I ODBRANA MASTER RADA

Datum prijave magistarskog rada:	17.11.2023.
Datum sjednice Vijeća na kojoj je prihvaćena tema:	01.12.2023.
Komisija za ocjenu/odbranu rada:	Prof. dr Budimir Lutovac, ETF Podgorica, predsjednik Prof. dr Nikola Žarić, ETF Podgorica, mentor Doc. dr Miloš Brajović, ETF Podgorica, član
Mentor:	Prof. dr Nikola Žarić
Datum odbrane:	*11.09.2024. godine*

Ime i prezime autora: Sanja Bjelanović, BApp

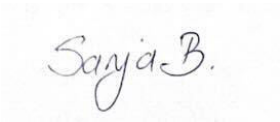
E T I Č K A I Z J A V A

U skladu sa članom 22 Zakona o akademskom integritetu i članom 18 Pravila studiranja na master studijama, pod krivičnom i materijalnom odgovornošću, izjavljujem da je master rad pod naslovom

" Alati za testiranje i otkrivanje grešaka u Web aplikacijama: Studija slučaja Upotreba Selenium alata"

moje originalno djelo.

Podnosilac izjave,



U Podgorici, dana 28.05.2024. godine

PREDGOVOR

Tvoje prijateljstvo i podrška od prvog do posljednjeg dana studiranja ostaće kao vječna uspomena na mom daljem putu.

Nedostaješ mi Blažo, i uvijek ćeš biti u mom srcu. Ovaj rad je za tebe.

IZVOD TEZE

Rad istražuje primjenu alata za testiranje softvera, sa posebnim fokusom na Selenium, kako bi se efikasno otkrile greške u web aplikacijama. Cilj je analizirati brzinu, efikasnost i preciznost automatizovanih testova u poređenju sa manuelnim pristupom. Kroz analizu karakteristika Selenium IDE i Selenium WebDriver komponenti, istraženo je kako ovi alati doprinose unaprijeđenju kvaliteta softverskih proizvoda, posebno kada je u pitanju testiranje korisničkog interfejsa web aplikacija. Uzimajući u obzir složenost testiranja i izazove u osiguranju kvaliteta softvera, ovaj rad ističe važnost kombinacije manuelnih i automatizovanih testova kako bi se postigao optimalan nivo kvaliteta i pouzdanosti softvera. Kroz detaljnu analizu prednosti i izazova u korišćenju ovih alata, ova teza doprinosi razumijevanju najboljih praksi u testiranju softvera i pruža smjernice za unaprijeđenje procesa testiranja u razvoju web aplikacija.
Ključne riječi: testiranje softvera, selenium, web aplikacije.

ABSTRACT

The paper explores the application of software testing tools, with a special focus on Selenium, to effectively identify errors in web applications. The aim is to analyze the speed, efficiency, and accuracy of automated tests compared to manual approaches. Through an examination of the features of Selenium IDE and Selenium WebDriver components, it has been investigated how these tools contribute to enhancing the quality of software products, particularly in testing the user interface of web applications. Considering the complexity of testing and challenges in ensuring software quality, this paper highlights the importance of combining manual and automated tests to achieve an optimal level of software quality and reliability. Through a detailed analysis of the advantages and challenges of using these tools, this thesis contributes to understanding best practices in software testing and provides guidelines for improving the testing process in the development of web applications.

Keywords: software testing, selenium, web applications.

SADRŽAJ

Slike

Tabele

Grafikoni

UVOD

Softverski sistemi su neizostavan dio našeg svakodnevnog života. Većina ljudi imala je iskustva sa softverom koji nije funkcionisao kako se očekivalo. Softver koji ne radi ispravno može izazvati različite probleme, uključujući gubitak novca, vremena ili poslovnog ugleda. Testiranje softvera procjenjuje kvalitet softvera i pomaže u smanjenju rizika od neuspjeha softvera u operativnom radu.

U okviru ovog rada istražićemo primjenu različitih alata za efikasno testiranje i detekciju grešaka u web aplikacijama. Poseban osvrt dat je na Selenium alat i njegovo poređenje sa ostalim analiziranim alatima.

Proces testiranja ima za cilj pronalaženje grešaka u izvornom kodu koje mogu dovesti do problema u softveru. Testiranje nije lak proces i prate ga izazovi u pokušaju da se obezbijedi kvalitet softverskog proizvoda. Kako bi pomogli ispitivačima da ostvare svoj cilj, postoji izbor različitih tehnika testiranja koje se mogu primijeniti u procesu testiranja, kao što su funkcionalno testiranje, nefunkcionalno testiranje, unit testiranje, sistemsko testiranje, integraciono testiranje itd. Kroz različite metode i alate, razvojni timovi osiguravaju da aplikacija radi kako je zamišljeno, te da se otkriju potencijalni problemi prije nego što aplikacija bude dostupna korisnicima.

Manuelno testiranje je proces testiranja softvera putem manuelnih postupaka radi identifikacije grešaka u softverskom proizvodu. Ispitivač igra ulogu krajnjeg korisnika i testira aplikaciju kako bi utvrdio da li se sistem ponaša u skladu sa zahtjevima. Međutim, postavlja se pitanje efikasnosti manuelnog testiranja u procesu osiguranja kvaliteta softvera. Upotreba alata za automatizaciju testiranja kao što je Selenium može uticati na kvalitativne karakteristike softvera, i zbog toga bi njihovi korisni efekti trebali biti razmatrani.

Upravo u ovom kontekstu, Selenium kao automatski alat, zajedno sa manuelnim alatima, postaje ključni igrač u optimizaciji testiranja web aplikacija.

Predmet istraživanja

Predmet istraživanja u radu je opis same problematike testiranja, ciljevi i metodologija. Nakon toga je fokus na tome kako Selenium kao alat za automatizaciju može da se iskoristi za efikasno testiranje i otkrivanje grešaka u web aplikacijama. Izvršena je analiza brzine Seleniuma u poređenju sa manuelnim alatima za testiranje, istražujući kako ovi pristupi doprinose unaprijeđenju kvaliteta softverskih proizvoda.

Selenium, kao alat otvorenog koda, svojom osnovnom svrhom pruža snažan interfejs za kreiranje test skripti u više programskih jezika, uključujući Python, Javu, PHP, Perl, C, NodeJS i Ruby.

Njegova sposobnost da podržava testiranje aplikacija na mnogim web pregledačima, uključujući Firefox, Chrome, Operu i Safari, čini ga sve popularnijim izborom.

U sklopu ovog istraživanja, detaljno je analizirana primjena Selenium IDE i Selenium WebDriver komponenti u automatizovanom testiranju korisničkog interfejsa web aplikacije koristeći web pregledač Chrome kao primarni alat za izvođenje testova.

Fokus je na razvoju i primjeni automatskih testova za procjenu ponašanja aplikacije.

Motivi i ciljevi istraživanja

Testiranje je u posljednih nekoliko godina doživljelo veliku popularnost u svijetu tehnologije. Svjedoci smo da ogroman broj softverskih projekata nailazi na probleme u fazi produkcije usljed grešaka u nekoj od ranijih faza razvoja.

Veoma je bitno napomenuti da je složenije softvere ili sisteme nemoguće istestirati do kraja, tj. iscrpno izvršiti testiranje softvera, za sve moguće putanje i u svakom mogućem okruženju. Zato je glavni fokus osobe koje se bavi testiranjem, da napravi inteligentan izbor malog broja test slučajeva na osnovu kojeg bi se mogla odrediti procjena kvaliteta softvera, pokriti najveći dio aplikacije i obezbijediti nesmetano funkcionisanje softvera.

Ključni razlog istraživanja je:

1. Identifikacija i razumijevanje ključnih prednosti i izazova u korišćenju Selenium-a u odnosu na manuelne alate. Zbog različitih specifikacija raznih sistema i aplikacija, potrebno je odrediti koje test strategije i tehnike su najpogodnije za testiranje.

Ključni motivi istraživanja su:

1. Interakcija manuelnog testiranja i Seleniuma putem izvršavanja skripti: Manuelno testiranje podrazumijeva da testeri interaguju sa aplikacijom na isti način kao i krajnji korisnici, istražujući njene funkcionalnosti i identifikujući greške putem interakcija. S

druge strane, Selenium automatizacija se oslanja na skripte kako bi simulirala korisničke interakcije, brzo i dosljedno izvršavajući testove koji se ponavljaju.

2. Brzina i efikasnost: Automatizovano testiranje nadmašuje manuelno testiranje kada je u pitanju brzina i efikasnost. Automatizovane skripte mogu izvršiti veliki broj test slučajeva za kratak vremenski period, dok bi čovjeku bilo potrebno mnogo više vremena za manuelno izvršenje istih testova.
3. Ponavljanje i testiranje regresije: Automatizacija se ističe u ponavljajućim zadacima, kao što je testiranje regresije. Manuelno višestruko izvršavanje istih test slučajeva može dovesti do ljudskih grešaka i nesavršenosti. Automatizacija osigurava da se testovi izvršavaju tačno onako kako su planirani svaki put.

Ciljevi istraživanja su:

1. Upoređivanje Selenium-a i manuelnih alata, radi bolje identifikacije razlika u brzini i preciznosti.
2. Istraživanje primjene Selenium IDE i Selenium WebDriver komponenti u automatizovanom testiranju, sa fokusom na Chrome pregledač.
3. Razvijanje i pisanje testova unutar alata, koji će da procijene ponašanja aplikacije koja podrazumijevaju određene operacije kao što su: pristup korisnika aplikaciji, dodavanje i izmjena podataka, pretraga i sl. Ovi ciljevi će biti od pomoći oko smjernica za buduće testiranje i rezultate tokom samog istraživanja.

Pregled dosadašnjih istraživanja

U radu [1] autori su sproveli anketu, uz medijsko sponzorstvo ToolsQA, kako bi bolje razumjeli i prikupili dokaze o izazovima sa kojima se suočavaju profesionalci za testiranje i njihove organizacije u vezi sa test automatizacijom. Oni su dobili primjerke od 2.291 ispitanika. Upitnik se sastojao od nekoliko pitanja, koja su se ticala njihovih uloga, iskustva, alata koje koriste, vrstama testiranja i izazovima sa kojima se suočavaju svakodnevno. Pored toga su i pronašli statističku analizu iz perspektive menadžera u vezi sa vrstama alata i pogled na ozbiljnost problema. Zaključak ovog istraživanja sugeriše da postoji značajan broj izazova s kojima se profesionalci za testiranje i njihove organizacije suočavaju u vezi sa automatizacijom, što ukazuje na potrebu za daljnjim razvojem alata i strategija kako bi se ti izazovi savladali.

Autori rada [2] su istraživali u industrijskom kontekstu, kvalitet i troškove izvođenja manuelnog testiranja i automatske generacije testova. Studija im se temeljila na 61 IEC 61131-3 programima iz softvera za kontrolu i na manuelnim test skriptama koje su kreirali industrijski stručnjaci. Njihovi rezultati sugerišu da automatska generacija testova moze postići sličnu pokrivenost

odluka kao manuelno testiranje koje obavljaju industrijski inženjeri, ali u znatno kraćem vremenskom periodu.

U sljedećem radu autori opisuju kratak pregled tehnika testiranja web aplikacija. Pregled je zasnovan na različitim prethodnim studijama koje su sprovedene u ovom određenom polju. Istraživanje se bazira na klasifikaciji testiranja prema vrstama testiranja i njihovom povezanom doprinosu. Istraživanje autora u osnovi obuhvata šest različitih vrsta testiranja web aplikacija: funkcionalno, sigurnosno, upotrebno, performansno, kompatibilnosno i strukturalno testiranje. Takođe, zaključuju da je glavni doprinos za sve vrste testiranja u vidu pristupa, alata i modela. [3]

U radu [4] autori raspravljaju o potrebi za specifičnim aspektima kvaliteta usluga u elektronskom okruženju i razvijaju model koji identifikuje različite dimenzije kvaliteta usluga u digitalnom kontekstu. Oni naglašavaju važnost prilagođavanja tradicionalnih metoda ocjenjivanja kvaliteta usluga kako bi odgovarale specifičnostima elektronskog poslovanja. Pored toga, istražuju ključne faktore koji utiču na percepciju kvaliteta usluga kod korisnika, uključujući brzinu usluge, sigurnost, pouzdanost, personalizaciju i jednostavnost korišćenja.

Autori u radu [5] razvijaju i istražuju E-SERVOERF pristup za procjenu kvaliteta web usluga. Rad se bazira na definisanju dimenzija unutar samog kvaliteta web stranica. Tri identifikovane dimenzije kvaliteta web servisa su percepcija rizika, web sadržaj i praktičnost usluge. Iako percepcija rizika može dovesti do povoljne percepcije kvaliteta web servisa, to ne znači da će rezultat biti zadovoljstvo korisnika. Individualne vještine računara mogu uticati na percepciju praktičnosti usluge, ali izgleda da nemaju uticaja na to kako korisnici ocenjuju kvalitet web servisa, zadovoljstvo korisnika ili namjeru za korišćenjem e-servisa. [6] Ovo je prva knjiga koja je obrađivala ideju da je dizajn važan kao i samo testiranje, pored toga što govori da je testiranje poželjan cilj, samim tim pokazuje i čitaocu kako to postići. Svako poglavlje u knjizi ilustruje kako se svaka tehnika može implementirati unutar određenog softvera i na koji način ga može poboljšati. Primjena ovih tehnika se bazirala na jedinično, integraciono, održivo i sistematsko testiranje.

Autori u radu [7] istražuju primjenu vještačke inteligencije u testiranju softvera, istražuju načine na koje mogu da potpomognu manuelnim alatima i koji su algoritmi najviše korišteni. Naročitu primjenu vidi kod ‘black box’ testiranja. Zaključuju da je veoma bitna iskorišćenost u primjeni kod optimizacije i efikasnijeg testiranja.

U radu [8] autori istražuju korišćenje algoritama mašinskog učenja za precizno testiranje neuspjelih testova. Rezultati ukazuju da su dobijene konceptualne grupe korisne u prenošenju znanja za testiranje. Iako se smatra da je potrebna ručna revizija rezultata, ovaj pristup smanjuje napor inženjera za testiranje i može ubrzati proces ispitivanja i testiranja softvera. Rad [9] je

baziran na upoređivanju manuelnog i automatizovanog testiranja. Autor smatra da analitički izvještaj sugeriše na korišćenje alata za automatizaciju. Takođe, smatra da takav izvještaj neće potcjenjivati greške u manuelnom testiranju ukoliko je tester specijalista za skriptovanje, što znači da i manuelno testiranje može osigurati visokokvalitetne web aplikacije.

Autori istražuju strategije za testiranje koje je bazirano na riziku. S.Besson je definisao princip da radi na prioritetizaciji testova na osnovu rizika koji su povezani sa različitim aspektima softvera. Autor S.Besson je razgradio strategiju za primjenu rizikom vođenog testiranja u okviru softverskog razvojnog procesa. [10]

Autor u radu [11] naglašava važnost tehničkih inspekcija i uvodi inovativan pristup koji spaja različite koncepte za 'model-based' testing. Njegova knjiga se bazira na testiranje softvera konkretno u okruženjima agilnog programiranja. U radu [12] autori su istraživali kako analiza testiranja softvera ostaje relevantna iako su se hardver i softver znatno promijenili tokom protekle tri decenije. Knjiga sadrži opis savremenih izazova, kao što su testiranje mobilnih aplikacija, kolaborativno programiranje i testiranje internet aplikacija.

Autori u radu [13] pružaju jednostavno i lako razumljivo učenje testiranja automatizacije korišćenjem Selenium WebDriver-a, koristeći Javu kao programski jezik. Ova knjiga ima za cilj da bude koristan resurs u testiranju automatizacije, posebno koristeći Selenium WebDriver.

Autori žele da istaknu u radu [14] da postoji značajan napredak u istraživanjima iz oblasti saradničkog softverskog inženjeringa. Ovo istraživanje donosi nove pristupe za efikasniju primjenu uključujući najbolje prakse, procese, alate, metrike i druge tehnike koje su dostupne za svakodnevnu upotrebu.

U radu [15] autor uvodi osnovne koncepte testiranja softvera. Takođe, razmatra problem "istraživačkog i skriptovanog testiranja" i predstavlja idealne vještine testera. Na kraju samog poglavlja, autor se bavi kratkom filozofskom diskusijom o "problemu univerzalnosti" u kontekstu testiranja softvera.

Naučne metode

Kako bi se testirala istraživačka pitanja u vezi sa datom temom mogu se koristiti različite metode, uključujući sljedeće:

1. Metoda testiranja (UI testiranje). Kreirani su testni slučajevi koji su zaduženi za testiranje korisničkog interfejsa same aplikacije. To uključuje provjeru izgleda, funkcionalnosti i ponašanje aplikacije na različitim uređajima.

2. Analiza statističkih podataka. Korišteni su podaci o vrstama testiranja, kao i vrijeme koje je potrebno za manuelno i automatizovano testiranje. Takođe, obrađena je pokrivenost testova i brzina otkrivanja grešaka.
3. Performanse testiranja: Sprovođenje serije testova upotrebom Selenium-a radi upoređivanja sa manuelnim metodama.
4. Analiza pokrivenosti. Na osnovu rezultata pokrivenosti između automatskih i manuelnih metoda idetifikovane su oblasti koje su bolje pokrivene automatskim testovima.
5. Pregled rezultata. Na osnovu analize, diskutovano je o zaključcima vezanim za rezultate između manuelnog i automatizovanog testiranja, kao i njihovim performansama.
6. Selenium u praksi: Rezultati ankete među testirima. Na osnovu ankete došli smo do zaključka koliko je Selenium kao alat popularan, efikasan i korišten kod osoba koje se bave testiranjem softvera.

Očekivani rezultati

Neki od doprinosa master rada su:

1. Smjernice za odabir metoda kod testiranja. Ovaj doprinos je značajan, jer je omogućio razvoj smjernica koje će pomoći organizacijama da bolje razumiju koji je metod testiranja najbolji za njihovu situaciju.
2. Primjeri na stvarnom projektu. Pisanje testova na praktičnim primjerima može dodatno da pomogne istraživačima da bolje razumiju implementaciju.
3. Identifikacija prednosti i nedostataka između manuelnih i automatizovanih alata.
4. Naročito značajan doprinos master rada čini upoređivanje rezultata dobijenih između manuelnog i automatskog testiranja na konkretnim scenarijima. Testiranje je obavljeno na skupu testova, a rezultati su upoređivani u odnosu na brzinu, preciznost i resurse koji su potrebni za izvršavanje. Svakako u ovom pristupu mora se spomenuti i da su manuelno i automatizovano testiranje međusobno povezani.

Struktura master rada

U prvom dijelu biće objašnjen pojam softverskog testiranja. Dublje ćemo istražiti ciljeve i motive koji stoje iza ovog procesa, sa fokusom na to kako u današnjem tehnološkom svijetu testiranje najbolje doprinosi unaprijeđenju kvaliteta web aplikacija.

U dijelu Faze softverskog testiranja je opisan detaljan pregled svake faze pojedinačno, uključujući planiranje, dizajn testova, izvršavanje i analizu rezultata. Osvrnuli smo se i na pitanje o tome kada je najpogodnije započeti proces testiranja, a biće obrađene i glavne razlike između manuelnih i automatizovanih alata.

Sekcija Analiza Selenium alata pružiće uvid u koncept alata za automatizaciju testiranja, sa naglaskom na Selenium. To podrazumijeva detaljan opis kako Selenium radi, zbog čega biramo Selenium, koji su najveći benefiti, koji tipovi testiranja mogu biti pokriveni upravo korišćenjem istog i sl. Objašnjenje će uključivati različite komponente Seleniuma, kao što su Selenium WebDriver, Selenium IDE i Selenium Grid.

U eksperimentalnom dijelu je odabrana web aplikacija na kojoj je odrađeno automatizovano testiranje. Obrađeno je nekoliko test slučajeva i upoređene su djelotvornost i brzina između manuelnog i automatizovanog testiranja. Naglašene su i glavne razlike između ova dva tipa testiranja. Samo korišćenje Seleniuma je podrazumijevalo demonstraciju različitih testova direktno u kodu, kao i prikaz rezultata uz korišćenje tabele i drugih prikladnih modela.

U završnom poglavlju ovog istraživanja, izvršena je analiza dobijenih rezultata. Pored toga, pružen je uvid u značaj efikasnog testiranja i osiguranja kvaliteta unutar samog razvoja web aplikacija.

1. TESTIRANJE SOFTVERA

Definisanje softvera je kompleksno, jer se koristi raznovrsna terminologija i koncepti. Softverski proizvod, ili kraće softver, je skup računarskih programa i pripadajuće dokumentacije koji su dizajnirani da zadovolje potrebe korisnika. On može biti razvijen za specifičnog korisnika ili za širu upotrebu na tržištu. U današnjem svijetu, kvalitet softvera je neophodan.

Česta zabluda o testiranju je da se sastoji samo od izvršavanja testova (tj. pokretanja softvera i provjere rezultata testova). Međutim, testiranje softvera uključuje i druge aktivnosti i mora biti usklađeno sa životnim ciklusom razvoja softvera. Još jedna česta zabluda o testiranju je da se fokusira isključivo na verifikaciju testnog objekta. Iako testiranje uključuje verifikaciju, odnosno provjeru da li sistem ispunjava navedene zahtjeve, takođe uključuje i validaciju, što znači provjeru da li sistem zadovoljava potrebe korisnika i drugih zainteresovanih strana u svom operativnom okruženju, [15], [16]. Sada ćemo istražiti kako modeli životnog ciklusa razvoja softvera utiču na testiranje, što će nam omogućiti bolje razumijevanje veze između razvojnog procesa i testiranja softvera.

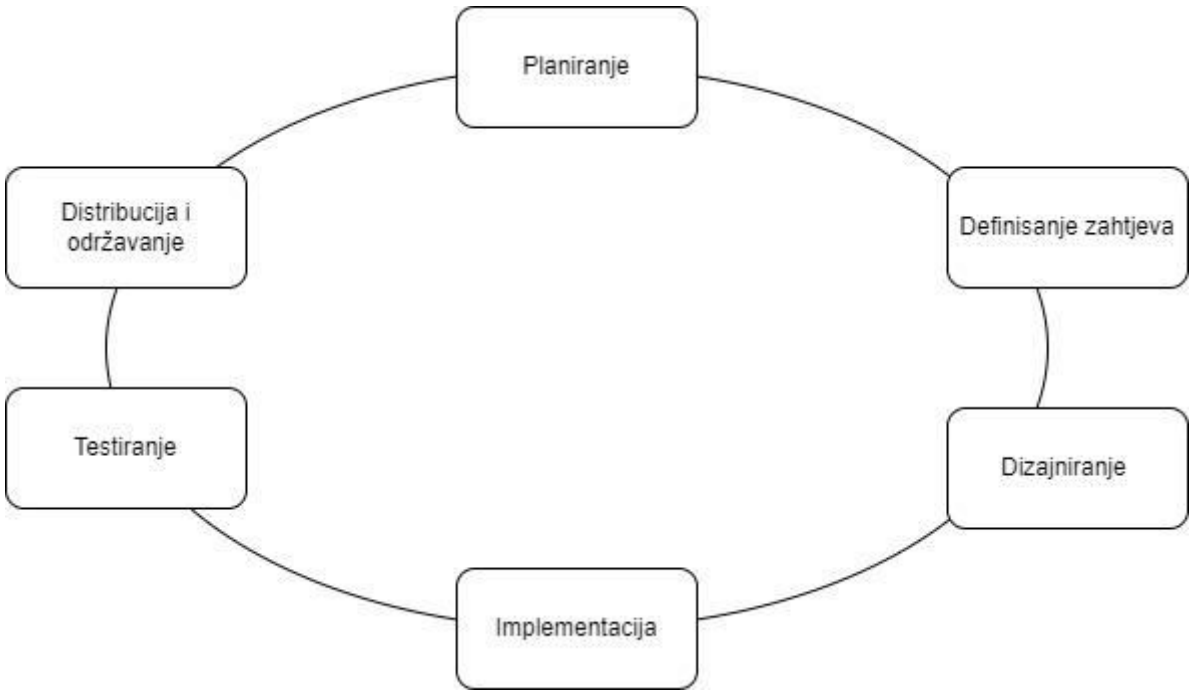
1.1 Životni ciklus razvoja softvera - SDLC

SDLC je akronim za Software Development Life Cycle, odnosno Životni ciklus razvoja softvera.

• Takođe se naziva Software Development Process, odnosno Proces razvoja softvera.

• SDLC je okvir (framework) koji definiše zadatke izvršene u svakom koraku u toku razvoja softvera.

SDLC je proces po kome se realizuje softverski projekat unutar jedne organizacije. Sastoji se iz definisanog plana koji opisuje kako da se razvije, održava, zamijeni ili poboljša određeni softver. Životni ciklus definiše metodologiju za poboljšanje kvaliteta softvera i uopšteno procesa razvoja. Grafički prikaz ciklusa softverskog projekta, prikazuje ključne faze karakteristične za životni ciklus razvoja softvera, (Slika 1.) [14], [17].



Slika 1. Faze životnog ciklusa razvoja projekta

1.1.1 Faze životnog ciklusa razvoja softvera (SDLC)

Faza 1: Planiranje zahtjeva – Tokom ove faze, tim za ispitivanje posvećuje pažnju proučavanju zahtjeva iz perspektive testiranja kako bi jasno identifikovao zahtjeve klijenta. Ovaj proces omogućava ispitivačkom timu da dubinski istraži zahtjeve, prepozna ključne aspekte i stvori razumijevanje kritičnih tačaka koje će biti podvrgnute ispitivanju. Ova analiza ima za cilj ne samo prepoznavanje funkcionalnih zahtjeva, već i sagledavanje specifičnosti koje se odnose na ispitivanje, čime se osigurava temeljno razumijevanje zahtjeva kako bi se pravilno usmjerila strategija ispitivanja u daljim fazama razvoja softvera, [34].

Faza 2: Definisanje zahtjeva - Kada je analiza zahtjeva završena, sljedeći korak je da se jasno dokumentuju zahtjevi za proizvod i da budu odobreni od strane klijenta. To se radi definisanjem zahtjeva koji ulaze u pisani dokument pod nazivom SRS (Software Requirement Specification) koji se sastoji se od svih korisničkih zahtjeva za dizajniranje i razvijanje proizvoda u okviru životnog ciklusa projekta, [14].

Faza 3: Dizajniranje Arhitekture Proizvoda - Na osnovu zahtjeva navedenih u dokumentu SRS, obično se napravi više od jednog predloga dizajna za arhitekturu proizvoda koji se dokumentuje u DDS-u (Design Document Specification). Ovaj DDS se daje na razmatranje i uvid od strane svih važnih aktera projekta (project stakeholders). U toku analize DDS-a vrši se procjena rizika da li će proizvod koji nastane ispuniti sve funkcionalne, tehnološke i sigurnosne zahtjeve proizvoda. Razmatra se i modularnost dizajna, sagledava odobren budžet i rokovi, kako bi se

predvidjela sva ograničenja. Na kraju najbolji pristup u dizajnu se bira za konkretan softverski proizvod [3].

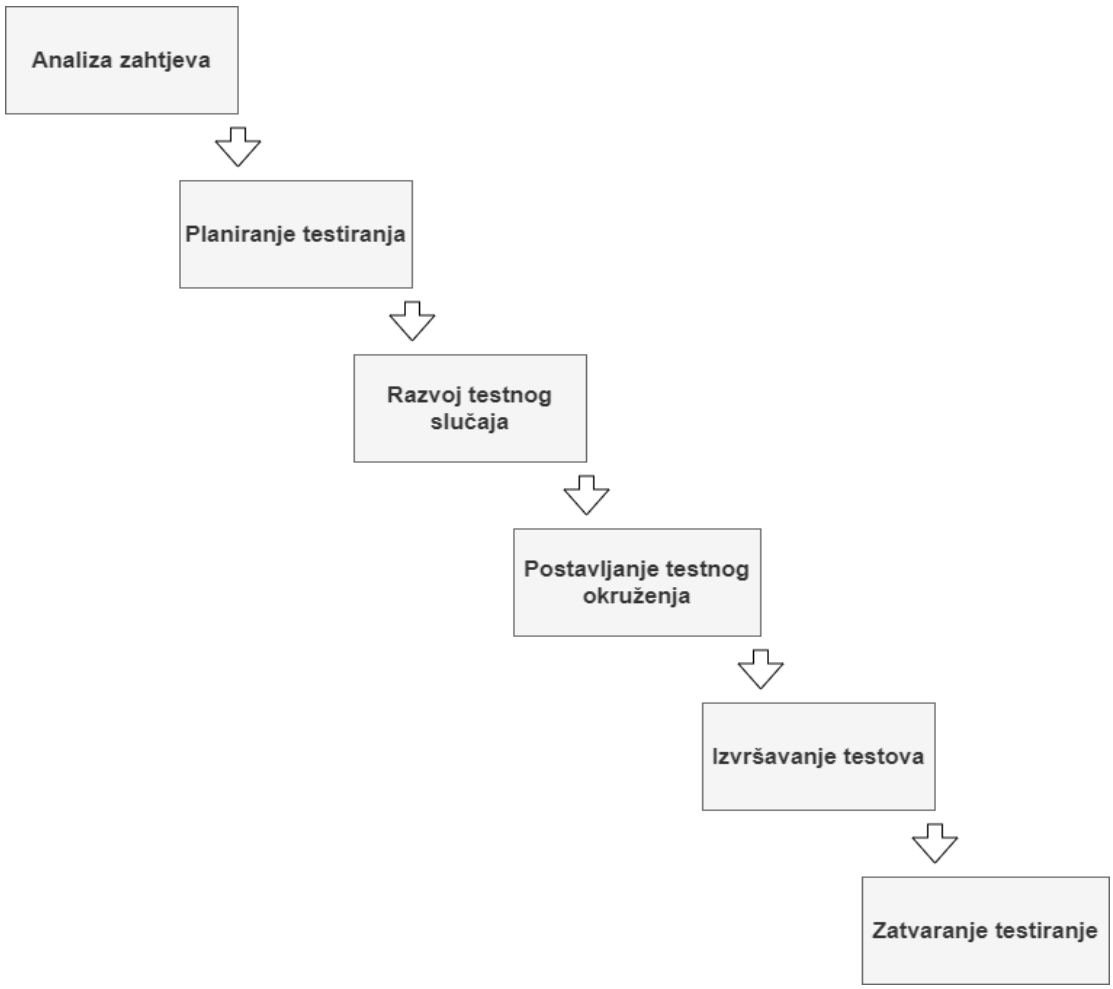
Faza 4: Implementacija - U ovoj fazi započinje stvarni razvoj proizvoda, a kao rezultat iz ove faze ciklusa, imamo napravljen softver prema svim zahtjevima koji su definisani u prethodnim fazama. Programski kod generisan je po zahtjevima opisanim u DDS dokumentu. Programeri moraju da prate uputstva za kodiranje (coding style) koja su definisana od strane njihove organizacije i programskih alata, vode računa o performansama, pouzdanosti, skalabilnosti, modularnosti i sigurnosti softverskog rješenja koje razvijaju [17], [19].

Faza 5: Testiranje proizvoda - Testiranje prožima cijeli životni ciklus razvoja, jer se u svakoj fazi razvoja vrši validacija i verifikacija napravljenog. Kao poseban korak, ova faza životnog ciklusa odnosi se na verifikaciju funkcionalnih i ne tehničkih zahtjeva softvera prije isporuke klijentima. Ova faza se odnosi na testiranje softverskog proizvoda gdje se izvještava o nedostacima i manama proizvoda, njihovo praćenje, popravljanje i ponovno testiranje, dok proizvod ne dostigne standarde kvaliteta koji su definisani u SRS dokumentu. Više o fazama samog softverskog testiranja dato je u poglavlju 1.2.

Faza 6: Distribucija na tržište i održavanje - Kada je proizvod testiran i spreman da bude distribuiran, isporučuje se klijentu. Proizvod može biti prvo objavljen u ograničenom segmentu i testiran u realnom poslovnom okruženju (UAT- User Acceptance Test). Zatim na osnovu povratnih informacija, proizvod može biti pušten takav kakav je, ili sa predloženim poboljšanjima u ciljnom tržišnom segmentu. Nakon što je proizvod pušten na tržište, slijedi faza održavanja i podrške klijentima. Postoji održavanje u garantnom roku i održavanje nakon prestanka garantnog roka. Nakon prestanka garantnog roka, klijent obično traži izmjene na softverskom rješenju u skladu sa promjenama koje se dešavaju: promjene zakonske regulative, promjene tehnologija, novi zahtjevi kupaca, poboljšanja korisničkog interfejsa, procesa rada itd [7], [17].

1.2 Faze softverskog testiranja (STLC)

Životni ciklus testiranja softvera (Software Testing Life Cycle) predstavlja redosljed određenih aktivnosti koje se sprovode tokom procesa testiranja kako bi se osiguralo da su ispunjeni ciljevi kvaliteta softvera. Suprotno svakodnevnom mišljenju, testiranje nije samo pojedinačna aktivnost, već se sastoji od niza aktivnosti koje se sprovode planirano i metodološki.



Slika 2. Faze softverskog testiranja

1.2.1 Analiza zahtjeva

Ova prva faza podrazumijeva detaljno ispitivanje i razumijevanje zahtjeva softverskog sistema kako bi se osiguralo efikasno planiranje i sprovođenje testova. Uključuje procjenu jasnoće i konzistentnosti zahtjeva kako bi se identifikovali mogući izazovi i rizici u razvoju softvera. U okviru analize zahtjeva, vrše se aktivnosti poput identifikacije zahtjeva, stvaranja testnih scenarija i određivanja kriterijuma za prihvatanje testova.

Unutar ove faze postoji nekoliko vrsta zahtjeva:

- Poslovni zahtjevi preuzeti iz poslovne dokumentacije projekta;
- Dizajnerski zahtjevi koji su detaljniji od poslovnih zahtjeva i sadrže cjelokupni dizajn potreban za izvršavanje poslovnog zahtjeva;
- Sistemski zahtjevi koji sadrže detaljan opis kao preduslov za programiranje.

Glavni kriterijum ulaska u ovu fazu je specifikacija zahtjeva. Aktivnosti koje se zatim sprovode su:

- prikupljanje podataka (poslovne analize, dizajneri, programeri),

- definisanje prioriteta testiranja,
- analiza i definisanje testova koji se mogu izvoditi,
- utvrđivanje izvodljivosti automatizovanih testova.

Kao rezultat navedenih aktivnosti je dokument koji prati sve navedene zahtjeve [8], [17].

1.2.2 Planiranje testiranja

Ova faza usmjerava razmišljanje testera i prisiljava ih da se suoče sa budućim izazovima koji se odnose na rizike, raspored, ljude, alate, troškove, napor, itd. Kao ulazni kriterijum ove faze uzima se dokument koji je dobijen kao rezultat prve faze analize zahtjeva. Planiranjem testiranja određuje se strategija testiranja [9], [19].

Tipičan sadržaj testnog plana uključuje:

- kontekst testiranja (npr. opseg, testni ciljevi, ograničenja, osnova za testiranje),
- procjena vremena koje se planira uložiti,
- procjena broja ljudi koji će biti uključeni u testiranje,
- procjena troškova testiranja,
- definisanje ograničenja u testiranju,
- definisanje rasporeda testiranja.

1.2.3 Razvoj testnog slučaja / testnog scenarija

Ova faza je ključan korak u procesu testiranja, jer pruža precizne smjernice za sprovođenje testova i ocjenu performansi softverskog sistema. Obuhvata stvaranje, provjeru i prilagođavanje test slučajeva i testnih skripti nakon završetka testnog plana. Inicijalno, podaci za ispitivanje precizno se identifikuju, zatim se pažljivo testni podaci izrađuju. Osim što u ovoj fazi tim ispisuje testne skripte, takođe priprema testne uzorke. Testni uzorci opisuju kako tačno treba sprovesti svaki test. Iako organizacije imaju različite pristupe u pisanju i dokumentaciji testnih scenarija ono što je kod svih zajedničko jeste da testni scenariji moraju biti što detaljniji. Kad je testni scenario detaljno ispisan to olakšava njegovo ponavljanje, odnosno regresiju [5], [6], [18].

1.2.4 Postavljanje testnog okruženja

Korak u kojem se definišu softverski i hardverski uslovi pod kojima će se vršiti ispitivanje softverskog proizvoda. Ova faza utvrđuje parametre pod kojima će se softver testirati kako bi se osigurala pokrivenost i tačnost rezultata ispitivanja. Odlučivanje o okruženju može se sprovoditi paralelno sa fazom razvoja testnog slučaja. Podešavanje testne okoline najčešće odrađuju timovi zaduženi za definisanje arhitekture softvera, timovi za razvoj, dizajneri, timovi zaduženi za održavanje baze podataka i sl. Nakon što se testna okolina podesi, testni tim izvršava provjeru spremnosti okoline za dalji rad.

1.2.5 Izvršavanje / Sprovođenje testova

Nakon što su sve prethodne faze završene slijedi faza testiranja. Ova faza uključuje sprovođenje testova u skladu sa testnom skriptom i upoređivanjem očekivanih i dobijenih rezultata. Ako je dobijeni rezultat nekog testnog koraka isti kao i očekivani, tad se prelazi na sljedeći. Ukoliko je dobijeni rezultat različit od očekivanog, tada tester prijavljuje grešku programeru i traži od njega ispravak iste. Nakon ispravka greške testeru preostaje još jednom da sprovede sve testne korake kako bi uvrdio da ispravak nije uticao na rad neke druge funkcionalnosti. Izvršavanje testova može biti manuelno ili automatizovano. Izvršenje testova može imati različite forme, uključujući kontinuirano testiranje ili sesije testiranja u paru.

1.2.6 Izvještavanje i zatvaranje testiranja

Ova faza je posljednja u životnom ciklusu testiranja softvera. Ona uključuje sastanak svih članova tima radi ocjene koja se temelji na osnovu rezultata i pokrivenosti testova.

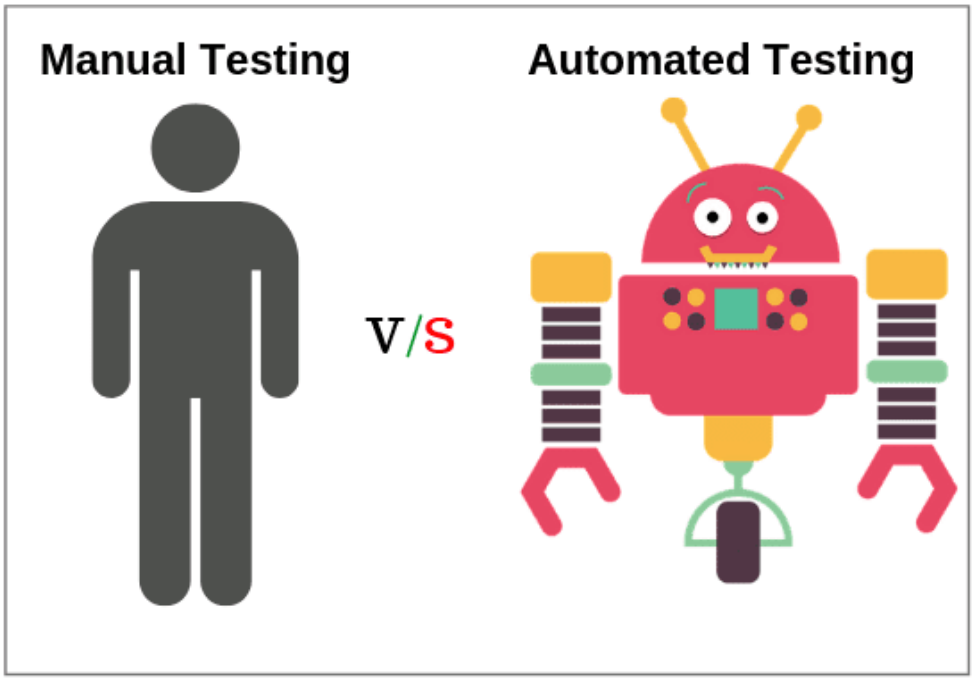
Proces zatvaranja testa se sprovodi na osnovu nekoliko faza:

- definisanje verzije softvera koja će se isporučiti klijentu,
- provjera ima li prijavljenih kritičnih grešaka koje mogu da utiču na prekid rada softvera,
- primopredaja sistema,
- arhiviranje testne dokumentacije,
- prikupljanje ideja i iskustava koja se mogu iskoristiti na drugim projektima.

Glavni rezultat ove faze su dokumenti o testovima, metrikama za ispitivanje i izvještaji o zatvaranju testa i projekta [18], [19].

2. MANUELNO I AUTOMATIZOVANO TESTIRANJE

Testiranje softvera može se sprovesti na dva načina - manuelno i automatizovano. Manuelno testiranje uključuje ručno izvršavanje testova, dok automatizovano testiranje podrazumijeva korišćenje alata i skripti za automatizaciju procesa testiranja. Svaka od ovih metoda ima svoje prednosti i ograničenja, pa je važno pažljivo odabrati odgovarajući pristup u zavisnosti od specifičnosti projekta i ciljeva testiranja [20].



Slika 3. Manuelno / Automatizovano testiranje

U nastavku, detaljno će biti obrađene ove dvije metode, pružajući uvid u njihove karakteristike, prednosti i situacije u kojima su najefikasnije.

2.1 Manuelno testiranje

Manuelno ili ručno testiranje je metoda testiranja kod koje tester priprema testne scenarije, a zatim ih sprovodi kako bi prepoznao grešku unutar softvera. Ovaj tip testiranja predstavlja najrigorozniju i najstariju metodu testiranja softvera.

Ova metoda zahtijeva posebne vještine koje su ključne za sprovođenje testiranja. Prije svega, neophodna je strpljivost, s obzirom da manuelno testiranje može biti izuzetno kompleksno, zamorno i zahtjevno u ponavljanju. Osim strpljivosti, testeri moraju posjedovati vještine kreativnosti, inovativnosti, upornosti, kao i određenu dozu otvorenosti i intuicije kako bi prepoznali potencijalne probleme. [1], [2].

Manuelno testiranje ima ključnu ulogu, posebno u ranoj fazi testiranja, jer svaki novi sistem ili aplikacija moraju prvo proći kroz manuelno testiranje prije nego što se razmotri automatsko

testiranje. Iako manuelno testiranje zahtijeva više truda, neophodno je kako bi se procijenila mogućnost automatizacije. Potpuna automatizacija testiranja softvera nije uvijek moguća, niti uvijek potpuno precizna. Iz tog razloga, manuelno testiranje ostaje nezaobilazno u testiranju softvera, čak i kao najosnovnija metoda koja igra ključnu ulogu.

Kakve su koristi manuelnog testiranja?

- Ljudska uključenost **-** Kvalifikovani tester može dobiti informacije o statusu proizvoda na osnovu ličnog iskustva. U slučaju automatizovanog testiranja, to je nemoguće jer se zasniva na prethodno razvijenim slučajevima, a promjena algoritma zahtijeva određeno vrijeme i, samim tim, određene troškove.
- Fleksibilnost - Još jedna prednost manuelnog testiranja je mogućnost improvizacije. Recimo da je klijent donio odluku da promijeni funkciju aplikacije prije izdanja. U ovom slučaju, morate prepraviti gotove softverske testove. Pri testiranju u automatizovanom režimu, to zahtijeva ažuriranje i provjeru automatskih testova, a ovaj postupak košta vremena i novca. U manuelnom režimu, ažuriranje funkcije je mnogo jednostavnije i jeftinije, [11].
- Brz početak - Da bismo pratili konkurenciju, povremeno se moraju dodavati nove funkcionalnosti unutar aplikacije. Manuelno testiranje omogućava brzo testiranje novih funkcija unutar proizvoda bez stvaranja automatskih testova.
- Efikasnost troškova - Konačni argument za manuelno testiranje je da je automatizacija skuplja. Ona zahtijeva stručnjake sa odgovarajućim kvalifikacijama i odgovarajućom naknadom. Automatski testovi moraju biti ažurirani do trenutka izdanja. Rad na stalnom unapređivanju automatskih testova negativno utiče na brzinu razvoja aplikacije i motivaciju tima uopšte [1], [16], [20].

Postoje razna uvjerenja ili zablude koje tim mogu da natjeraju da se dvoume o izboru manuelnog testiranja. Neka od najpopularnijih su:

- Testiranje povećava troškove razvoja - Smatra se da ukoliko neko ne želi da plaća testiranje u fazi razvoja, morat će da plati tehničku podršku i ispravke bagova nakon što proizvod izađe na tržište. Poenta je da rano testiranje pomaže smanjenju troškova održavanja softvera.
- Testiranje je gubitak vremena - Samo testiranje se sprovodi gotovo istovremeno s razvojem, stoga nema razloga vjerovati da ovaj proces oduzima puno vremena. Istina je da popravka grešaka identifikovanih u fazi testiranja zahtijeva vrijeme i trud, ali je lakša nego ispraviti ih nakon izdanja.
- Testiranje se vrši samo nakon faze razvoja - Očigledno je da testiranje zavisi od napisanog koda. Ali ipak, potrebno je provjeriti softverske zahtjeve i kreirati testne

artefakte, a obije procedure mogu se izvršiti prije potpunog završetka softverskog proizvoda.

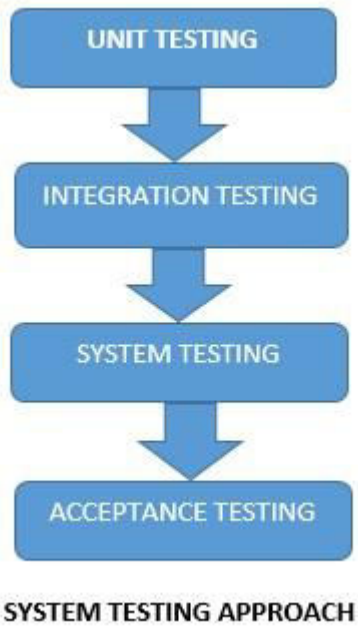
- Moguće je izvršiti iscrpno testiranje - Nemoguće je tvrditi da testiranje pokriva 100% sistema i da su svi bugovi lako predvidljivi i ispravljivi. Činjenica je da se većina problema može predvidjeti u fazi testiranja, ali potpuno iscrpno testiranje je teško ostvarivo. Cilj testiranja je pronaći i identifikovati maksimalan mogući broj bagova. Nemoguće ih je pronaći sve. Čak ni najskuplji moderni alati za automatsko testiranje ne mogu garantovati 100% zaštitu softvera od bagova [2], [19].
- Tester je jedini odgovoran za kvalitet softvera - Ovo je jedno od najčešćih zabluda vezanih za testiranje. Pokušaj pronalaženja krivca nije dobra strategija. Cijeli tim je odgovoran za kvalitet proizvoda. Timovi za praćenje takođe ukazuju na problematična područja timovima za razvoj, a njihova zajednička odgovornost je popraviti problem i predati softver bez grešaka.
- Automatizovano testiranje eliminiše potrebu za manuelnim testiranjem - Potpuna automatizacija procesa testiranja nije moguća. U različitim stepenima, uvijek je potrebno manuelno testiranje. Timovi za testiranje i osiguranje kvaliteta trebaju procijeniti koliko sistem zahtijeva manuelno i automatizovano testiranje.

2.1.1 Vrste i zahtjevi manuelnog testiranja

Manuelno testiranje mora slijediti određene parametre:

1. Svrha - Zavisno od zadatka, obavlja se funkcionalno ili nefunkcionalno testiranje. Svrha funkcionalnog testiranja je provjeriti koje funkcije su implementirane u proizvodu. Tokom nefunkcionalnog testiranja, inženjer provjerava kako softver radi kao cjelina. Testiraju se sljedeći parametri:
   - performanse - rad pod opterećenjem,
   - da li je interfejs prikladan,
   - zaštita softverskog proizvoda od hakera, neovlašćenog pristupa podacima, itd.,
   - da li postoje problemi tokom instalacije, deinstalacije ili ažuriranja softvera,
   - kompatibilnost softvera sa određenim okruženjem,
   - lokalizacija verzija softvera (lingvistički i kulturni aspekti).
2. Scenario - Može biti pozitivan ili negativan:
   - pozitivan - testira da li program odgovara očekivanom ponašanju korisnika,
   - negativan - pokazuje kako će proizvod raditi u slučaju neočekivanog ponašanja korisnika, [7], [8], [12].
3. Nivoi testiranja:

- Unit Testing - Testiraju se pojedinačni softverski moduli, jedinice softvera. Ovaj tip testiranja mogu obaviti sami programeri.
- Integration Testing – Testiranje integracije između pojedinačnih modula. Ovaj tip testiranja obavljaju testeri.
- System Testing - Čitav sistem - u skladu sa zahtjevima za softverski proizvod. Ovaj tip testiranja takođe obavljaju testeri, [20].
- Acceptance Testing - Prihvatanje je proces provjere da li softver zadovoljava kriterijume i potrebe krajnjeg korisnika. Tip koji je testiran od strane programera.



Slika 4. Prikaz levela/nivoa manuelnog testiranja (STC, 2018.)

4. Značaj testiranja

Prema važnosti testiranih funkcija, testiranje se dijeli na:

- Smoke testiranje - provjera najvažnije funkcionalnosti softverskog proizvoda;
- Testiranje kritičnog puta - provjera funkcionalnosti koje koriste tipični korisnici u svakodnevnim aktivnostima;
- Napredno testiranje - verifikacija svih deklarisanih funkcionalnosti, [3], [4].

2.1.2 Primjene manuelnog testiranja

Generalno, manuelni pristup se preporučuje za sljedeće vrste testiranja:

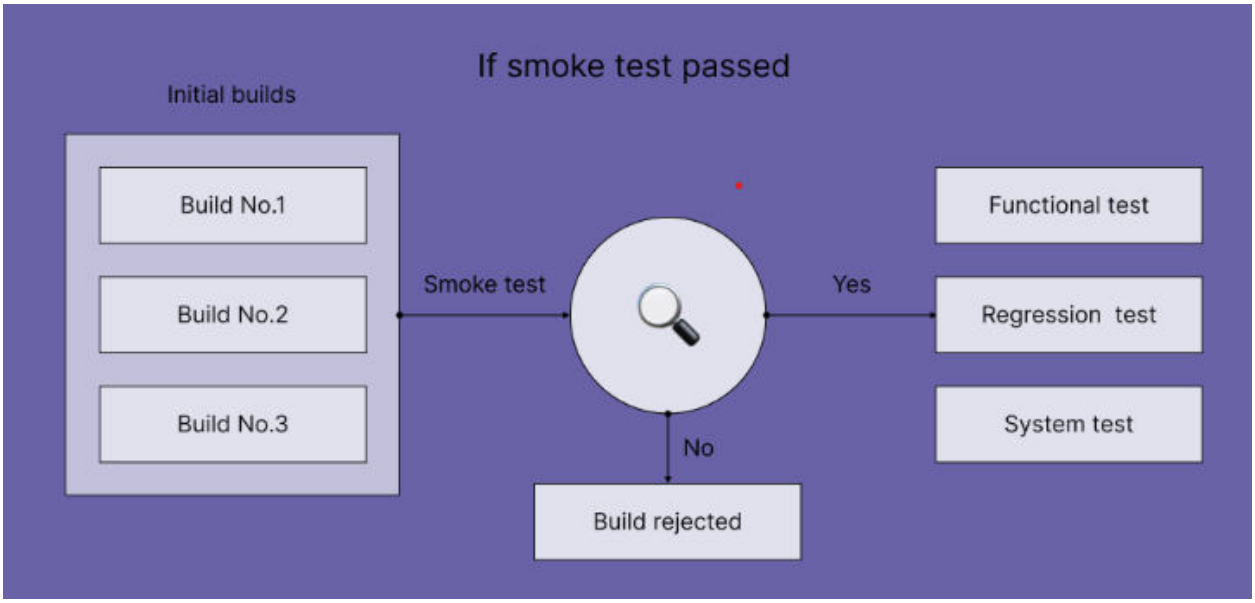
1. Istraživačko testiranje. Pri pripremi skripte, tester se oslanja isključivo na vlastito iskustvo, intuiciju i logičke zaključke. Istraživanje se obavlja brzo i bez kvalitetne dokumentacije. Cilj je brzo identifikovati kritične greške u softveru.
2. Testiranje upotrebljivosti. Prirodno, analiza softvera na upotrebljivost ne može biti automatizovana. To može obaviti samo čovjek.
3. Intuitivno testiranje ne uključuje unaprijed pripremljenu skriptu. U procesu verifikacije, inženjer se oslanja na lično iskustvo, zdrav razum i poznavanje proizvoda. Improvizacijom, tester identifikuje greške u softveru,[19], [20].

Manuelno testiranje se započinje pripremom testnih slučajeva zajedno sa početkom razvoja proizvoda. Slučajevi se mogu razlikovati po nivou detalja, ali se osiguravamo da pokrijemo sav softver, kako bismo bili spremni za dalje testiranje.

Testiranje počinje sa ‘’smoke’’ testiranjem (Testiranje dima), koji pokazuje da li je proizvod spreman za verifikaciju. Jednostavno rečeno, funkcija ili aplikacija se pokreće, i provjeravamo da li je moguće pokrenuti funkcionalno testiranje. (Slika 5.)

Smoke testiranje nema za cilj da pokrije svaki testni scenario. Umjesto toga, koncentriše se na temeljno ispitivanje najkritičnijih funkcija sistema kako bi se potvrdilo da rade kako se očekuje. To je brza i efikasna metoda za testiranje bez poduzimanja opsežnog testiranja.

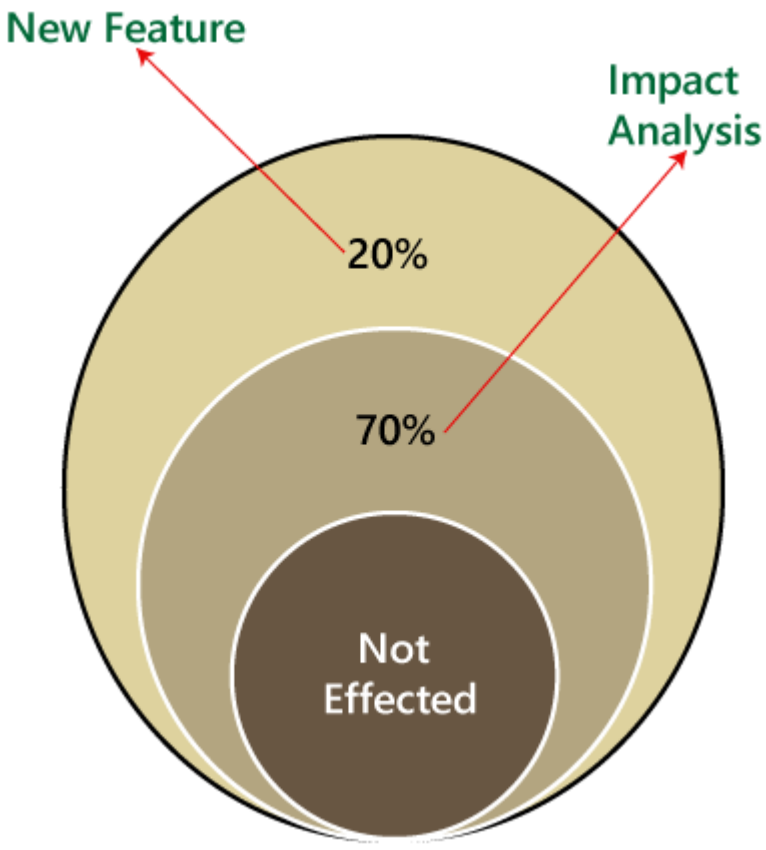
Ovo je prvi test performansi proizvoda koji pokazuje npr. da li je moguće prijaviti se, bez navođenja da li su uneseni podaci u polja validni, da li dodatne funkcionalnosti kao što su podsjetnici za lozinke rade. Ako sve ovo funkcioniše, prelazimo na dublju provjeru ovog modula - uzimamo negativne slučajeve sa graničnim vrijednostima, i procjenjujemo usklađenost sa pravilima validacije.



Slika 5. Smoke testiranje (Global App Testing, 2024.)

Sljedeća faza obuhvata niz testova regresije s ciljem pronalaženja grešaka, čak i na mjestima gdje ranije nije bilo problema. Regresija predstavlja procjenu funkcionalnosti proizvoda na standardnom skupu slučajeva tokom implementacije svakog novog modula i svake promjene. Kada developeri dodaju nove funkcionalnosti, trenutna verzija može biti oštećena ili se nova funkcija može sukobiti s postojećim, (Slika 6.).

Na primjer, dodavanje novih ekrana, a time i promjena navigacije, može poremetiti rad ili barem njegov prikaz. Probleme mogu uzrokovati i ažurirane biblioteke koju koristi aplikacija i promjena okoline u kojoj radi. Uloga testiranja regresije je veća što se aplikacija češće ažurira. Ne bi trebalo ograničiti se samo na jedan test izvršen kada je funkcionalnost već implementirana - treba provjeriti sve promjene, [16].



Slika 6. Regresiono testiranje (JavaTpoint, 2021.)

Regresiono testiranje se, naravno, sprovodi kada je softver za izdanje sa jednom ili više funkcionalnosti spreman za puštanje u produkciju, ali još uvijek treba primijeniti određene relevantne slučajeve u određenim fazama razvoja.

Posljednja faza je softver za izdanje. Ovo je beta testiranje od strane naših testera i poslovno testiranje kada proizvod ocjenjuje klijent ili određena grupa korisnika. Nakon toga, aplikacija je spremna za puštanje u produkciju. Ali rad ne završava tu - testiramo ažuriranja aplikacije i

njihovu kompatibilnost sa prethodnim verzijama, sastavljamo slučajeve za provjeru inovacija i, ako je potrebno, automatizujemo ove scenarije

2.2 Automatizovano testiranje

Kako samo ime govori, možemo pretpostaviti da se aktivnosti manuelnog/ručnog testiranja teže zamijeniti automatizovanim testiranjem. Drugim riječima, to znači da nije potreban ljudski unos kako bi se generisao test. Za razliku od manuelnog testiranja koje zahtijeva prisustvo testera kako bi interagovao sa svakim test slučajem, analizirao rezultate i izvještavao o njima, automatizovano testiranje koristi poseban softver za kontrolu izvršenja testova i upoređivanje stvarnih rezultata sa očekivanim.

Softversko testiranje je automatizovano kada postoji mehanizam za pokretanje test slučajeva bez prisustva testera. Pored toga automatizacija se opisuje kao zadatak stvaranja mehanički razumljivog prikaza manuelnog test slučaja. Ove dvije izjave jasno ukazuju da je test automatizacija usmjerena na oslobađanje testera i automatizaciju test slučajeva.

Kao što znamo, izvršavanje test slučajeva se smatra važnom procedurom tokom procesa testiranja i vrlo je vremenski zahtjevno. Doslovno rečeno, test automatizacija bi trebala štedjeti vrijeme, ali pretpostavke su da to neće biti jedina korist od automatizacije.

Zašto automatizacija?

S razvojem tehnologije, informacioni sistemi unutar organizacija postaju sve kompleksniji. Uz svaku promjenu ili unapređenje sistema, testiranje igra ključnu ulogu u isporuci proizvoda visokog kvaliteta na vrijeme.

Postoje tri ključna aspekta: brzina, kvalitet i nesigurnost u isporuci proizvoda.

Potreba za brzinom je praktično mantra informatičkog doba. Kasni proizvodi mogu izgubiti prihode, korisnike i tržišni udio. Osim toga, sa sve većom kompleksnošću tehnologije, kao i kratkim vremenom puštanja proizvoda, testiranje postaje izazovno. U smislu ciklusa razvoja softvera (SDLC), testiranje se tretira kao posljednji korak prije izlaska u proizvodnju. Međutim, vrijeme ostavljeno za testiranje često nije dovoljno, pogotovo ako se izvršava manuelno.

Kako organizacije mogu izvršiti sve moguće test slučajeve za kompleksan sistem u kratkom roku predstavlja veliku zabrinutost. Na kraju, troškovi dodatnog rada i dodatnog testiranja regresije kada se pronađu greške su skupi i teško se mogu predvidjeti. Kao rezultat toga, izlazak proizvoda može biti pogođen. Povećana kompleksnost današnjih softverskih sistema znači da postoji potencijalno neograničen broj kombinacija ulaza i događaja koji rezultiraju različitim izlazima sistema, a mnoge od ovih kombinacija često nisu obuhvaćene manuelnim testiranjem, [20], [23].

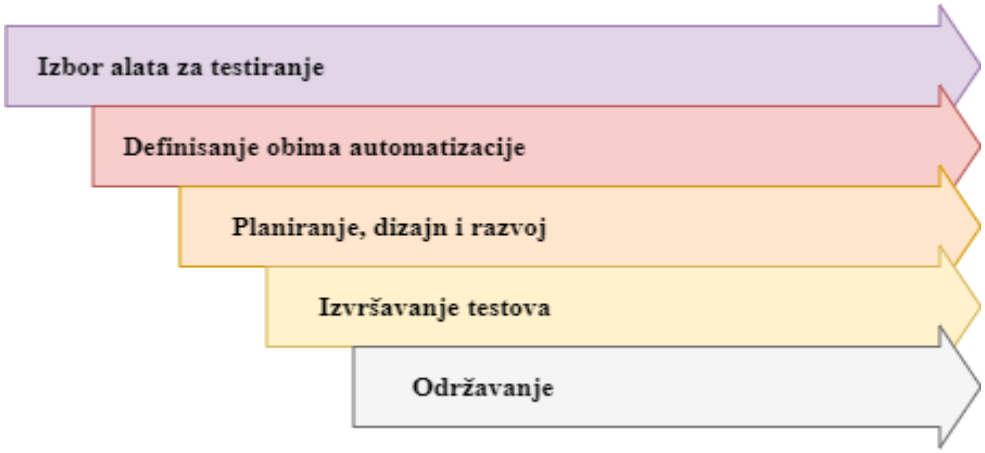
2.2.1 Prednosti automatizacije

1. Brzina - Automatizacija zahtijeva manje vremena u poređenju sa manuelnim testiranjem.
2. Ponavljanje - Ponavljanje podataka ili sadržaja može se lako postići pomoću alata za automatizaciju u kratkom vremenu.
3. Višekratna upotreba - Automatizovano testiranje može se koristiti na različitim verzijama softvera.
4. Testiranje opterećenja - Ne postoji održiva manuelna alternativa.
5. Kompletno šta - Testeri mogu kreirati testne scenarije koji pokrivaju svaki aspekt kvaliteta u softverskoj aplikaciji.
6. Pouzdanost - Automatizacija test slučajeva smanjuje ovaj rizik. Organizacija može uspostaviti proceduru koja osigurava da napisane test skripte budu ispravne.
7. Pokrivenost testa - Šira pokrivenost testiranja funkcionalnosti aplikacije.
8. Performanse - Automatizovano testiranje smanjuje vrijeme ciklusa testiranja i, samim tim, vrijeme do plasmana proizvoda na tržište, [21].

Što se tiče nedostataka automatizacije, znatno je manji broj nego kod prednosti. Neki od njih su:

- U početku ulaganje u ovakav tip testiranja moze biti skupo.
- Za razliku od manuelnog testiranja, automatizacija nije toliko lako prilagodljiva i svaka promjena u scenariju podrazumijeva promjene u samom kodu.
- Greška u koracima i programiranju alata moze dovesti do fatalnih grešaka u radu sistema.

2.2.2 Faze automatizacije



Slika 7. Faze automatskog testiranja

Postupak automatizacije sprovodi se u nekoliko koraka.

1. Prije samog početka potrebno je na temelju detaljne analize odabrati kvalitetan alat za testiranje koji će biti prilagođen potrebama softvera koji se testira. Izbor alata za automatizaciju uglavnom zavisi od tehnologije u kojoj je pisan kod za aplikaciju i od same primjene.
2. Zatim je potrebno odrediti opseg automatizacije. Definisanjem opsega postavljaju se prioriteti testiranja, kompleksnost testnih scenarija, količina podataka neophodna za izvođenje testa, itd.;
3. Nakon definisanog opsega potrebno je razraditi strategiju, plan automatizacije kao i vremenski okvir izvođenja testova. U ovoj fazi se radi kreiranje strategije automatizacije i planiranje koje sadrži sljedeće detalje:
   - odabrani alati za automatizaciju,
   - dizajn okvira i njegove karakteristike,
   - stavke automatizacije u obimu i van okvira,
   - priprema poligona za automatizaciju, odnosno testnog okruženja,
   - raspored i vremenski okvir izvršenja testova;
4. Nakon svih prethodno sprovedenih koraka slijedi izvođenje samog testa. Nakon završenog testa alat najčešće daje detaljan izvještaj o sprovedenom testu, nakon čega slijedi određivanje zadovoljavaju li dobijeni rezultati definisane kriterijume;
5. Faza održavanja predstavlja posljednji korak. Bilo kakva izmjena u specifikaciji zahtjeva promjenu scenarija, [20].

3. SELENIUM

Selenium je besplatni, *open source* alat (alat otvorenog koda) za automatsko testiranje web aplikacija, sa podrškom za različite web čitače i platforme. Kao i kod drugih alata za automatizaciju testiranja, Selenium skripte koje se pišu imaju za cilj da obavljaju sve akcije koje bi korisnik obavljao manuelno. Selenium može obavljati razne vrste automatske interakcije, ali primarna namjena i razlog zbog kog je nastao je upravo automatsko testiranje.

Priskrbljuje jedinstveni interfejs koji omogućava pisanje test skripti u programskim jezicima poput Ruby, Java, NodeJS, PHP, Perl, Python, JavaScript i C#, među ostalima. Selenium je vrlo proširiv i može se integrisati s drugim alatima i okvirima poput TestNG, JUnit, Cucumber, itd, [31], [21].

3.1 Istorija Seleniuma

Istorija Seleniuma proteže se kroz nekoliko godina i obuhvata razvoj i evoluciju skupa alatki usmjerenih na automatizaciju testiranja web stranica. Evo hronološkog pregleda ključnih događaja u istoriji Seleniuma, (Slika 8.).

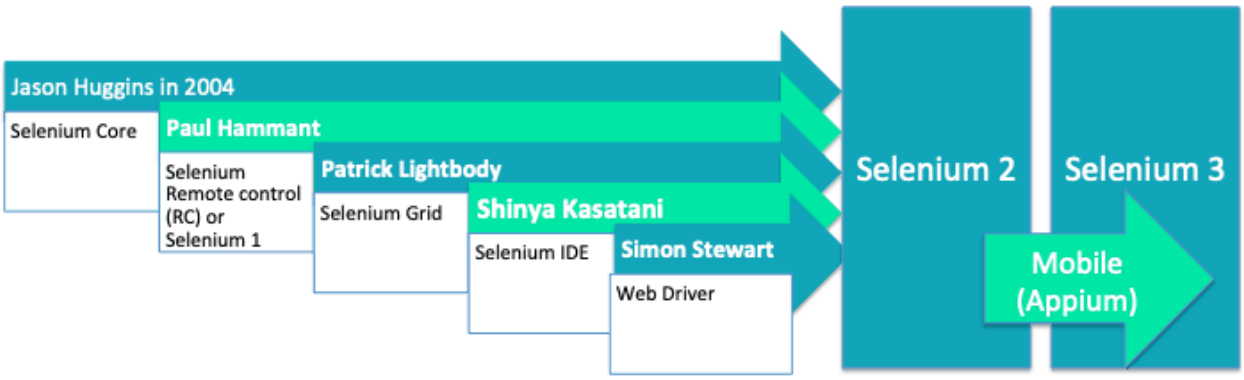
Duga istorija Seleniuma ima različite faze u kojima su ključne ličnosti značajno doprinijele rastu u softverskom testiranju.

Prva verzija Selenium-a nastala je 2004. godine kada je Džejson Hagins (Jason Huggins), za potrebe testiranja aplikacije u firmi ThoughtWorks, u kojoj je u to vrijeme radio, napisao Javascript biblioteku koju je integrisao sa Web stranama, što mu je omogućilo automatsko izvršavanje testova u različitim pretraživačima. Ta biblioteka je osnova na kojoj počiva Selenium Remote Control (poznat i kao Selenium 1), na kome je Hagins, zajedno sa drugim inženjerima, nastavio da radi.

Selenium Remote Control predstavlja prvi alat koji, korišćenjem programskog jezika po izboru, može da kontroliše pretraživač, [31], [35]. Međutim, Selenium zasnovan na Javascript-u imao je svoje nedostatke. Zbog bezbjedonosnih mjera pretraživača prema Javascript-u i sve složenijih aplikacija koje su uključivale nova ograničenja, mnoge stvari je bilo nemoguće uraditi. Sajmon Stjuart (Simon Stewart), inženjer u Guglu, počeo je 2006. godine da radi na projektu koji je nazvao WebDriver. Testeri u Guglu su intenzivno koristili Selenium i morali su da pronalaze različite načine da prevaziđu njegove nedostatke. Otuda Stjuartova ideja da napravi alat koji će direktno komunicirati sa pretraživačem, zaobilazeći Javascript. S obzirom da WebDriver pruža moćnije mogućnosti za testiranje od Selenium-a, a Selenium, sa druge strane, podržava veći broj pretraživača, tvorci Selenium-a 1 i WebDrivera-a, odlučili su 2009. godine da ova dva projekta spoje u jedan – Selenium WebDriver ili Selenium 2, ponudivši korisnicima najbolje moguće

rješenje. Paralelno, 2008. godine, Filip Hanrigu (Philippe Hanrigou), koji je takođe radio u ThoughtWorks-u, napravio je Selenium Grid koji podržava istovremeno izvršavanje više Selenium testova na bilo kom broju lokalnih ili udaljenih sistema, značajno umanjujući, na taj način, vrijeme potrebno za izvršavanje testova, [32], [33].

Naziv „Selenium" je nastao iz šale koju je napravio Džejson Hagins na račun konkurentske firme Merkjuri (Mercury). Merkjuri je u to vrijeme plasirao Mercury Interactive QuickTest Professional alat za testiranje. Hagins je htio da ponudi alternativu i za svoj alat želio ime koje to i pokazuje. Kako „mercury" u prevodu znači živa, novi alat dobio je naziv „Selenium", jer se trovanje živom liječi selenom (na engleskom „selenium").



Slika 8. Istorija Selenium-a (Ruchika Gupta, 2021.)

Selenium je od samih početaka imao veliki značaj, a danas je taj značaj još veći. Neki od ključnih razloga za korištenje Seleniuma su:

1. Podrška za jezike: Selenium omogućava kreiranje test skripti na različitim jezicima poput Ruby, Java, PHP, Perl, Python, JavaScript i C#, između ostalih.
2. Podrška za pregledače: Selenium omogućava testiranje web stranice na različitim pregledačima poput Google Chrome-a, Mozilla Firefox-a, Microsoft Edge-a, Safari-a, Internet Explorera (IE), itd.
3. Povećana razmjera: Automatizovano testiranje sa Seleniumom lako se može proširiti kako bi obuhvatilo raznolik spektar test slučajeva, scenarija i interakcija korisnika. Ova povećana razmjera osigurava maksimalno testiranje funkcionalnosti aplikacije.
4. Ponovljive test skripte: Selenium omogućava testerima kreiranje ponovljivih test skripti koje se mogu koristiti u različitim test slučajevima i projektima. Ova ponovljivost znači mnogo jer uštedi na vremenu u kreiranju i održavanju istih skripti.

5. Paralelno testiranje: Selenium podržava paralelno izvođenje testova, omogućavajući više testova da se izvršava istovremeno. Ovo pomaže u smanjenju ukupnog vremena testiranja, čineći razvojni proces efikasnijim, [35].
6. Dokumentacija i izvještavanje: Selenium pruža detaljne dnevnike izvođenja testova i izvještaje, olakšavajući praćenje rezultata testiranja i identifikaciju područja koja zahtijevaju pažnju.
7. Testiranje korisničkog iskustva: Selenium može simulirati interakcije i ponašanje korisnika, omogućavajući testerima da ocijene korisničko iskustvo i osiguraju da je aplikacija intuitivna i korisnički prijateljska.
8. Continuous Integration and Continuous Deployment (CI/CD): Selenium se može integrisati u CI/CD tokove rada kako bi automatizovao testiranje svake promjene u kodu. Ova integracija pomaže u identifikaciji i rješavanju problema ranije u razvojnom ciklusu, omogućavajući brže i pouzdanije izdanje, [22], [23].

3.2 Komponente Seleniuma

Selenium nije pojedinačan alat, već se sastoji od skupa softvera, svaki prilagođen različitim potrebama softverskog testera. Selenium ima četiri komponente koje uključuju:

1. Selenium IDE
2. Selenium RC
3. Selenium WebDriver
4. Selenium Grid

3.2.1 Selenium IDE

Selenium IDE, poznat i kao Integrated Development Environment je proširenje uglavnom, internet pretraživač Mozilla Firefox-a, ali takođe može raditi i sa Chrome pretraživačem. Selenium IDE omogućava korisnicima da snimaju, modifikuju i debaguju test skripte. Karakteriše ga visoka efikasnost s obzirom na mogućnost snimanja korisničkih akcija u redosljedu koji odgovara planiranom testiranju. Koristan za početnike u oblasti automatizovanog testiranja, budući da je od tri glavna Selenium alata najjednostavniji za upotrebu i razumijevanje komandi.

Test skripte u Selenium IDE-u mogu biti generisane automatski, ali takođe mogu biti ručno kreirane, pošto svaki test predstavlja niz komandi. Set komandi omogućava testiranje različitih aspekata web aplikacija, uključujući provjeru prisustva elemenata korisničkog interfejsa na osnovu HTML tagova. Podrazumijevano, snimljene test skripte čuvaju se u HTML formatu. Međutim, korisnici imaju opciju da ih eksportuju u format nekog od podržanih programskih

jezika, kao što su C#, Java, Python ili Ruby, čime se omogućava nadogradnja funkcionalnosti uzimajući u obzir specifičnosti odabranog jezika. Selenium IDE se koristi za snimanje i izvršavanje test skripti u Mozilla Firefox pregledaču. ili u drugim pregledačima koristeći Selenium WebDriver, [23], [24].

Slika 9. Tok rada Selenium IDE (Abhishek Yadav, 2014.)

3.2.2 Selenium RC

Selenium RC je važna komponenta u Selenium testnom paketu. To je testni okvir koji omogućava testeru ili programeru da piše test slučajeve na bilo kom programskom jeziku kako bi automatizovao UI (korisnički interfejs) testove za web aplikacije protiv bilo koje HTTP web lokacije.

Rad Selenium RC-a zasniva se na njegovim komponentama. Svaka od komponenti ima posebnu ulogu u izvršavanju test skripti. Selenium RC se sastoji od sljedećih komponenata:

1. Selenium server je odgovoran za pokretanje i zatvaranje pregledača (browser-a). On interpretira i izvršava Selenium naredbe koje su poslate iz test programa. Djeluje kao HTTP proxy, presrijeće i verifikuje HTTP poruke koje se prenose od aplikacije pod testom do pregledača. Selenium naredbe u test programu pristupaju se putem jednostavnih HTTP zahtjeva Servera. Programski jezik koji je sposoban slati HTTP zahtjeve može se koristiti za automatizaciju Selenium testova u pregledačima, [22].
2. Klijentske biblioteke - Za svaki podržani jezik podržana je jedna klijentska biblioteka. Selenium klijentske biblioteke nude aplikacijski programski interfejs ili API. To je skup funkcija koje pomažu u izvršavanju Selenium naredbi u programu. Programirana funkcija unutar svakog interfejsa podržava svaku Selenium naredbu. API-ji u klijentskim bibliotekama odgovorni su za prosleđivanje naredbi Selenium serveru. Server zatim obrađuje određenu funkciju ili sprovodi testove u odnosu na aplikaciju. API-ji u klijentskim bibliotekama takođe su odgovorni za prijem rezultata i prosljeđivanje istih

programu. Rezultati se zatim mogu čuvati u promjenljivoj. Program može čak i prijaviti test kao uspješan ili neuspješan. Ispravna akcija takođe može biti preduzeta ako se dogodi neželjena greška.

Test program može biti kreiran jednostavno pisanjem programa koji pokreće neke Selenese naredbe koristeći API-je u klijentskoj biblioteci. Ako postoji Selenium test skripta koja je razvijena u IDE-u, Selenium kod može biti generisan kao alternativna opcija. IDE ima mogućnost prevođenja naredbi u pozive funkcija API-ja klijentskog drajvera.

3.2.3 Selenium WebDriver

Selenium WebDriver pruža širok spektar naredbi koje olakšavaju automatizaciju web aplikacija. Ovaj alat omogućava programerima efikasniju i sigurniju interakciju sa web stranicama. Korišćenjem WebDriver API-ja, programeri mogu pristupiti i kontrolisati web elemente na stranici, omogućavajući im lako kreiranje automatizovanih testova.

WebDriver ima nekoliko prednosti u odnosu na tradicionalne alate za automatizaciju poput Firebug-a ili Selenium IDE-a. Na primjer, omogućava kompatibilnost sa više pretraživača: testovi napisani u WebDriveru će se izvršavati na bilo kojem pretraživaču koji podržava istu verziju Seleniuma. Pored toga, WebDriver ima potpun pristup HTML DOM objektima, omogućavajući mnogo veću fleksibilnost u razvoju testnih slučajeva. Svakako, nudi poboljšanu pouzdanost i performanse izvršavanjem testova direktno na pretraživaču umjesto kroz posrednika poput Firebug-a ili Selenium IDE-a, [22], [23].

Slika 10. Selenium WebDriver arhitektura (Neha Vaidya, 2023.)

Na slici 10. ChromeDriver je drajver pretraživača, a Chrome je pretraživač. Arhitektura i funkcije ostaju iste u svim ostalim klijentskim bibliotekama, drajverima pretraživača i pretraživačima.

Selenium Client Library – Selenium WebDriver podržava više biblioteka kao što su Java, Ruby, Python, itd. Developeri su razvili jezičke veze kako bi omogućili Selenium-u da podržava više jezika. Na primjer, ukoliko koristimo drajver pretraživač u Javi, moramo koristiti veze sa jezikom Jave.

JSON WireProtocol - JSON je skraćenica od JavaScript Object Notation. Koristi se za prenos podataka između servera i klijenta na web-u. JSON Wire Protocol je REST API koji prenosi informacije između HTTP servera. Svaki BrowserDriver (kao što je FirefoxDriver, ChromeDriver, itd.) ima svoj vlastiti HTTP server, [10].

Jednostavan primjer JSON formata:

```
{“name”: “JSON”,
“topic”: “Selenium WebDriver”,
“address”: “Web Automation”,
“context”: “Example”
}
```

S obzirom na to da serveri ne razumiju programske jezike, JSON Wire Protocol koristi proces serijalizacije (pretvaranje podataka o objektu u JSON format) i de-serializacije (pretvaranje JSON formata u objekt). JSON Wire Protocol ima REST API-je koji rade preko HTTP-a.

Za svaku Selenium komandu postoji odgovarajući REST API u JSON Wire Protocolu. Selenium radi preko API naredbi, kao što su GET i POST, i funkcioniše na osnovu zahtjeva Selenium skripte koje dobije. Zahtjevi se zatim šalju na HTTP server drajvera pretraživača, kao i na pretraživače preko HTTP-a. Ovo stupa u interakciju sa elementima aplikacije i odgovor se šalje nazad u IDE kroz isti kanal.

Drajveri pretraživača (Browser Drivers) – Svaki pretraživač sadrži poseban upravljački program pretraživača. Drajveri pretraživača komuniciraju sa odgovarajućim pretraživačem bez otkrivanja unutrašnje logike funkcionalnosti pretraživača. Kada upravljački program pretraživača primi bilo koju komandu, ta komanda će se izvršiti na odgovarajućem pretraživaču i odgovor će se vratiti u obliku HTTP odgovora, [22], [25].

Putanja drajvera se postavlja na sljedeći način:

```
System.setProperty(“webdriver.chrome.driver”,“E:\\chromedriver.exe”);
```

Nakon toga se pokreće test ili program pozivanjem interakcije Selenium WebDriver-a između Biblioteke klijenata Selenium i stvarnog pretraživača.

Drajver pretraživača je ključ za primanje HTTP zahtjeva od JSON Wire Protocol-a preko njegovog HTTP servera i slanje obrađenih zahtjeva stvarnom pretraživaču. Ovdje se odvija interakcija sa elementima u svakoj operaciji, [22], [25].

Browsers - Pravi pretraživači primaju zahtjeve od svog drajvera pretraživača i rade na elementima aplikacije prema zahtjevu. Nakon toga, program pretraživača vraća podatke o izlazu klijentu preko istog kanala.

Web UI Elementi - Web stranica sadrži brojne HTML elemente (ulaz, iframe, itd.) i u Selenium WebDriver-u oni se zovu WebElements.

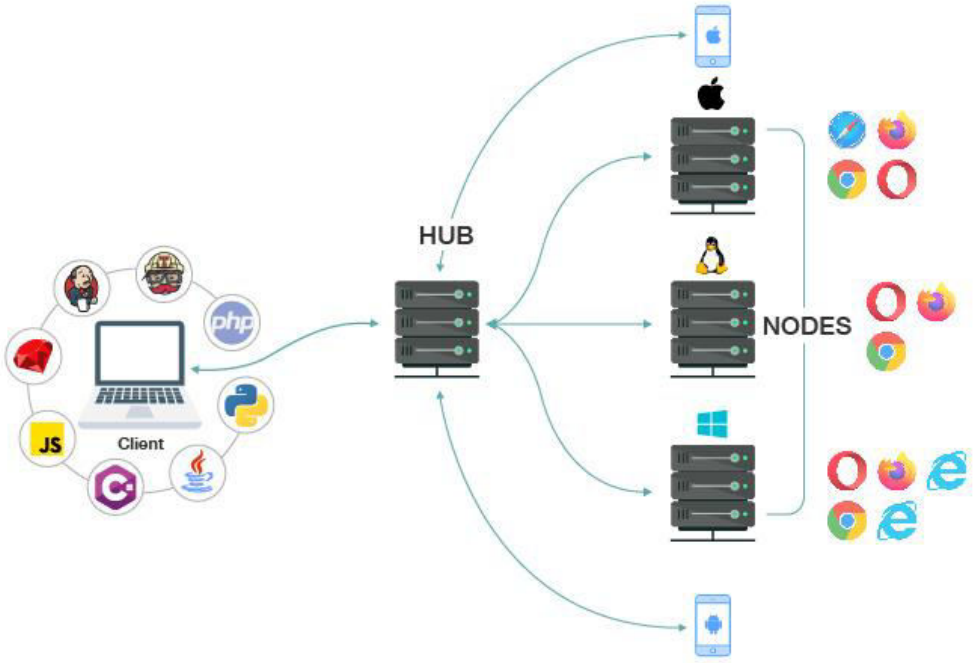
Locirajući ove elemente, WebDriver može lako obavljati zadatak i komunicirati sa Seleniumom. Zbog toga je imperativ da testeri moraju ispravno identifikovati element koji uključuje svaki test slučaj, ako ne, testni slučaj neće uspjeti zbog svoje nemogućnosti lociranja. Zahvaljujući snalažljivom WebDriver API-ju, postoje različite tehnike za lociranje UI lokatora na osnovu različitih atributa web stranice (tagname, classname, cssSelector, Xpath, ID, Name, LinkText) – DOM (Document Object Model) elemenata.

Nakon upita u DOM-u, ako je više od jednog elementa locirano i pronađeno, sistem javlja grešku. Neki od primjera elemenata na stranici i njihovih lokatora:

```
Class Name: By.className(“fruit”);
<div class="fruit"><span>Book</span></div>
ID: By.id(“bear”);
<div id="bear">...</div>
Name: By.name(“apple”);
<input name="apple" type="text"/>
```

3.2.4 Selenium Grid

Selenium Grid je komponenta unutar Selenium alata koja omogućava paralelno izvršavanje testova na više različitih pretraživača i platformi istovremeno. To omogućava brže pokretanje testova i povećava efikasnost testiranja, jer se testovi mogu distribuirati na različite čvorove u mreži. Ova funkcionalnost čini Selenium Grid izuzetno korisnim za testiranje skalabilnih web aplikacija na različitim konfiguracijama pretraživača i operativnih sistema.



Slika 11. Selenium Grid komponente (Harish Rajora, 2023.)

Selenium Grid se sastoji od arhitekture čvorišta i čvorova.

Čvor (Node):

- čvor je računar ili uređaj na kojem se izvršavaju Selenium testovi,
- svaki čvor može biti konfigurisan da podržava različite pretraživače i operativne sisteme,
- čvorovi se povezuju na čvorište kako bi čekali instrukcije za izvršavanje testova,
- kada čvor prima zahtjev za izvršavanje testa od čvorišta, on pokreće test na svom lokalnom pretraživaču i operativnom sistemu,
- čvorovi šalju rezultate testova natrag čvorištu nakon završetka izvršavanja.

Čvorište (Hub):

- čvoriste je centralni entitet u Selenium Grid-u koji upravlja distribucijom testova na različite čvorove,
- čvoriste prima zahtjeve za izvršavanje testova od korisnika ili automatizovanih skripti,
- na osnovu dostupnih kapaciteta i konfiguracija čvorova, čvoriste distribuira testove na odgovarajuće čvorove,
- prikuplja rezultate testova sa svih čvorova i prosleđuje ih korisnicima ili alatima za izvještavanje.

Na ovaj način, čvorovi omogućavaju paralelno izvršavanje testova na različitim konfiguracijama, dok čvoriste koordinira i upravlja ovim procesom, što omogućava efikasno i skalabilno testiranje web aplikacija, [22], [26].

3.3 Integracija programskog jezika Java sa Seleniumom

Selenium i Java se smatraju najboljom kombinacijom za pokretanje automatizovanih testova na različitim web pregledačima. Java se pokazala kao najpoželjniji jezik od strane profesionalaca koji svakodnevno koriste Selenium. Java je čest izbor za razvoj Selenium testova zbog svoje široke podrške, jasne sintakse i bogatih biblioteka. Nudi velike prednosti u pogledu modularnosti, ponovne upotrebljivosti koda i razumljivosti, što je čini idealnim jezikom za automatizaciju testiranja.

Neki od razloga zašto se Java toliko koristi unutar automatizacije:

- Izvršenje programa je brže u Javi u poređenju sa bilo kojim drugim programskim jezikom.
- Danas se Java više koristi od drugih jezika, pa je integracija Selenium testova sa njom relativno lakša. Pretežno se koristi u IT industriji, što rezultira ogromnom zajednicom koja je podržava, kao i masivnim repozitorijem referenci.
- Skoro 77% Selenium testera koristi Javu, što olakšava dijeljenje znanja i brzu razmjenu informacija, [13].

Java postoji već dugi niz godina, zbog čega postoji obilje već dostupnih okvira, dodataka, API-ja i biblioteka koji je podržavaju za automatizaciju testiranja. Koristi takođe dodatak JVM (Java Virtual Mashine) što je čini platformski nezavisnim jezikom. Drugim riječima, može da se koristi u bilo kojem operativnom okruženju gdje je instaliran JVM (Java Virtual Mashine).

Budući da je Java statički tipizirana, Java IDE (Integrated Development Environment) pruža puno povratnih informacija o greškama s kojima se programeri ili testeri mogu suočiti prilikom kodiranja, [13], [27].

3.4 TestNG (Testing framework for next generations)

TestNG je open-source testing framework, koji je namijenjen Java programskom jeziku, a razvio ga je Cédric Beust. Omogućava pisanje specifičnih testova, koji provjeravaju ispravnost, efikasnost i očekivano ponašanje djelova aplikacije. Inspirisan je JUnit-om i prevazilazi njegove nedostatke, jer je dizajniran da olakša testiranje do kraja. TestNG se pokazao kao moćan alat za testiranje Java aplikacija, pružajući raznolike mogućnosti za automatizaciju testova i poboljšanje efikasnosti razvojnog procesa. Sa svojim bogatim setom funkcionalnosti, fleksibilnošću i moćnim izvještavanjem, TestNG olakšava implementaciju složenih test scenarija i poboljšava kvalitet softvera.

Neke prednosti TestNG-a u odnosu na Junit koji je takođe open-source testing framework:

- anotacije su lakše za razumijevanje,

- test slučajevi (cases) mogu lakše da se grupišu,
- moguće je paralelno testiranje,
- proizvodi HTML izvještaje za implementaciju,
- generiše logove,
- podržava tri dodatna nivoa, kao što su @Before/After suite, @Before/AfterTest i Before/AfterGroup,
- koristi više Java i OO funkcija.

Koristeći TestNG, možemo da generišemo odgovarajući izvještaj i da lako saznamo koliko je test slučajeva prošlo, neuspješno ili preskočeno. Neuspjele test slučajeve možemo da izvršimo zasebno.

Primjer: Imamo 5 test slučajeva i jedna metoda je napisana za svaki od njih. (Pretpostavka da je program napisan koristeći glavnu metodu bez upotrebe TestNG.) Kada prvo pokrenemo program, uspješno se izvršavaju tri metode, a četvrta je neuspješna. Zatim, ispravimo greške prisutne u četvrtoj metodi i želimo da pokrenemo samo nju, jer se prve tri svakako izvršavaju uspješno. Sve ovo ne bi bilo moguće bez TestNG-a.

Ključne karakteristike:

- Anotacije (za laku identifikaciju metoda testa) - linije koda koje mogu da kontrolišu kako će se izvršiti metoda ispod njih. Uvijek im prethodi simbol @.

Primjer:

```
@Test(priority = 0)
public void goToHomepage() {
    driver.get(baseUrl);
    Assert.assertEquals(driver.getTitle(), "Welcome: Simon S");
}

@Test(priority = 1)
public void logout() {
    driver.findElement(By.linkText("SIGN-OFF")).click();
    Assert.assertEquals("Sign-on: Simon S", driver.getTitle());
}
```

Ovaj primjer jednostavno pokazuje, da metoda `goToHomepage()` treba da se izvrši prije `logout()` metode, jer ima niži prioritet.

- Fleksibilne konfiguracije testova - omogućava kreiranje fleksibilnih i dinamičkih konfiguracija testova, korišćenjem XML datoteka. To znači da lako možemo definisati

kako, kada i koji testovi treba da se pokrenu, što je posebno korisno za složene test scenarije, [30].

- Podrška za paralelno izvršavanje testova - značajno smanjuje vrijeme potrebno za izvršavanje velikog broja testova.
- Snažna izvještavanja - pruža detaljne izvještaje o izvršenju testova.
- Zavisnosti - podržava definisanje zavisnosti između test metoda.

TestNG okruženje se postavlja na sljedeći način:

1. Korak **-** Instalacija Java programa,
2. Korak - Podešavanje Java okruženja – gdje podešavamo varijablu Jave,
3. Korak – Preuzimanje TestNG Archive,
4. Korak **-** Podešavanje TestNG okruženja unutar IntelliJ IDEA (bez Maven-a),
5. Korak **-** Testiranje TestNG setup-a i verifikacija rezultata.

Primjer jednog inicijalnog testa:

Komanda koju pokrećemo Run TestCases, trenutno je pod nazivom Run as TestNG i biće objašnjeno u narednom koraku kako se dolazi do te promjene. Trenutno je poenta da se klikne na Run. Evo kako bi to izgledalo:

Slika 12. Prikaz pokretanja inicijalnog testa

Kada pokrenemo komandu, trebalo bi da dobijemo sledeći output:

```
I am inside setup

I am inside tearDown

I am inside test steps
```

```
Default Suite

Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
```

3.4.1 Maven i dodavanje TestNG-a koristeći Maven

Maven je moćan alat za upravljanje projektima, koji se zasniva na konceptu POM-a (Model objekta projekta) koji sadrži informacije o projektu i konfiguracione informacije za maven kako bi izgradio projekat, kao što su direktorijumi izgradnje, direktorijumi izvornog koda, ciljevi, razni plugin-i, itd. Pojednostavljuje proces izgradnje kao ANT, ali je dosta napredniji od njega. Ant je više kao build tool, a Maven se ponaša kao alat za upravljanje projektima.
To znači da možemo lako upravljati bilo kojim tipom Java projekta koristeći Maven. Ukoliko je projekat manji, sa njim se može upravljati ručno. Međutim, ako je projekat veliki ili ima više projekata, tada je efikasnije da developer ili tester taj projekat automatizuje. Maven definiše standardni način izgradnje testova, projekata i distribuciju projektnih artefakata. Pruža okvir koji omogućava jednostavno ponovno korišćenje zajedničke logike za svaki pojedinačni projekat koji prati Maven-ove standarde. Uobičajene upotrebe i ciljevi MAVEN-a su:

- Upravljanje zavisnostima: Maven automatski preuzima potrebne biblioteke i plugin-ove za naš projekat iz centralnog repozitorijuma. To znači da ne moramo ručno preuzimati i dodavati .jar datoteke u naš projekat.
- Konvencija nad konfiguracijom: Maven projekti slijede standardnu strukturu direktorijuma i formatiranje konfiguracijskih datoteka, što olakšava razumijevanje i upravljanje projektima, čak iako nismo njihov originalni autor.
- Životni ciklus izgradnje: Maven koristi unaprijed definisane životne cikluse izgradnje, (npr. clean, compile, test, package, install, deploy), koji olakšavaju definisanje i razumijevanje procesa izgradnje, [30].
- Pluginovi i ciljevi: Maven se oslanja na plugin-ove za izvršavanje zadataka. Svaki plugin može imati više ciljeva koji se mogu izvršiti.
- Smjernice: Pruža kvalitetne informacije o dokumentaciji projekta, kao i smjernice za najbolje prakse upravljanja istog.

Integracija TestNG-a putem Maven-a, omogućava automatsko upravljanje zavisnostima i izgradnjom projekta. Ovo je korisno u kontinuiranoj integraciji (CI) i kontinuiranoj isporuci (CD) okruženjima, gdje je potrebno automatizovati procese izgradnje, testiranja i isporuke.

Dodavanje TestNG-a putem Maven-a može biti jednostavniji početni korak, posebno za početnike ili manje iskusne developere. Maven automatski rješava zavisnosti, što olakšava početak rada sa TestNG-om, [28], [29].

Korišćenje testng.xml fajla omogućava veću portabilnost projekta, jer možemo lako prenijeti konfiguraciju testova između različitih razvojnih okruženja ili čak između različitih alata za izgradnju i testiranje.

4. EKSPERIMENTALNI DIO

U ovom dijelu fokus će biti na automatizovanom testiranju ključnih funkcionalnosti Doxio aplikacije koristeći Selenium.

Doxio je sistem za upravljanje dokumentima koji je dizajniran kako bi olakšao rad kompanijama koje imaju složene tokove rada i veliki broj papirnih dokumenata, ugovora i administrativnih poslova. Ovaj softver omogućava krajnjim korisnicima da digitalizuju svoju dokumentaciju. Zarad ovog istraživanja napisano je ukupno 112 testova koji pokrivaju sve ključne aspekte sistema. Ovi testovi su od velikog značaja jer osiguravaju da sistem kontinuirano funkcioniše kako je predviđeno, posebno kada su u pitanju složeni tokovi rada i obrađivanje velikog broja dokumenata. Pored toga, ovi testovi se periodično pokreću kako bi se izvršilo regresiono testiranje, što doprinosi očuvanju stabilnosti sistema tokom dužeg perioda.

Proces testiranja putem Seleniuma:

Postoji sedam osnovnih koraka prilikom testiranja pomoću Selenium alata:

1. Prvi korak za kreiranje instance webdrivera,
2. Drugo je navigacija na web stranicu,
3. Treće je pronaći HTML element na web stranici,

4. Četvrto je izvršiti radnju na HTML elementu,
5. Zatim, sljedeći korak je predviđanje odgovora pretraživača na akciju,
6. Poslednji korak je pokretanje i snimanje rezultata testa koristeći test okvir.

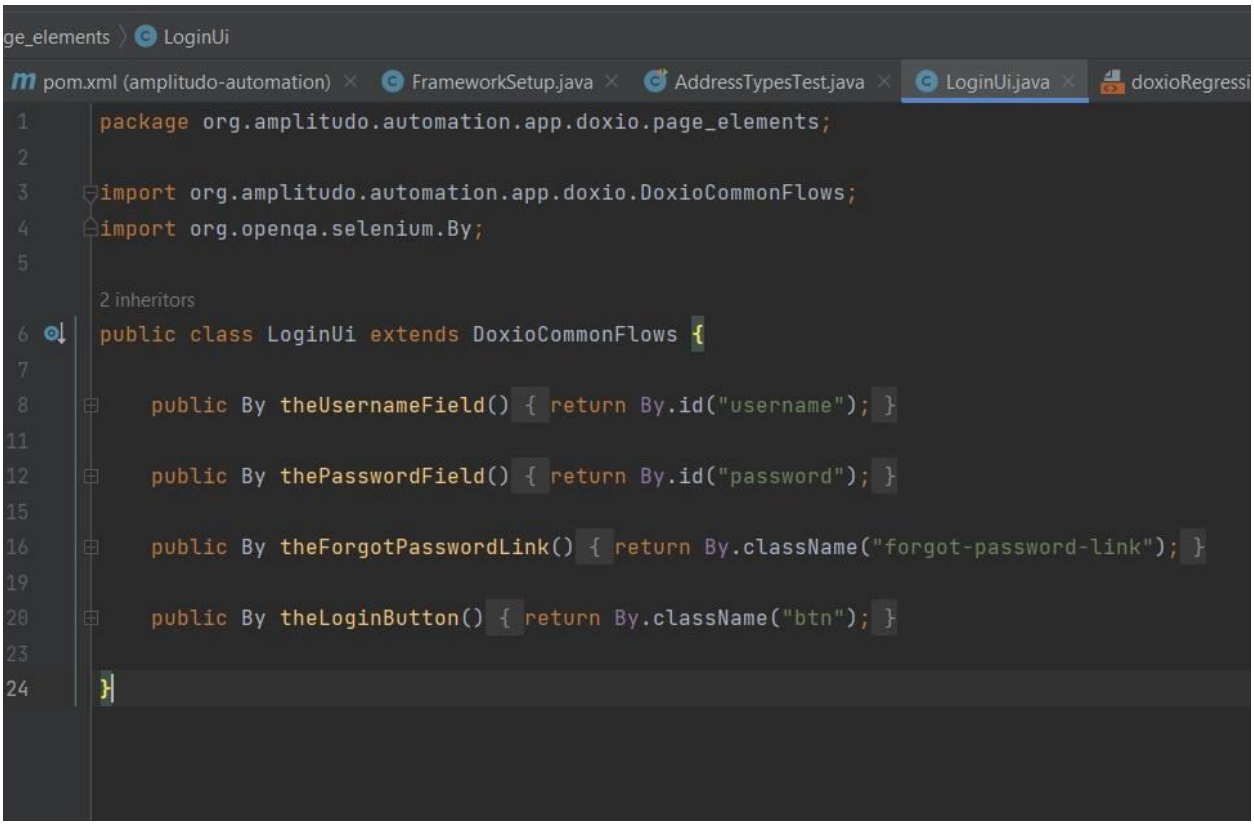
4.1 Prvi testni slučaj i njegova implementacija

Testni slučaj: LoginTest (Korisnik se uspješno prijavljuje sa ispravnim podacima na Doxio sistem i koristi Chrome pretraživač.)

Radi bolje preglednosti i čitljivosti koda unutar frameworka, posebno u kontekstu testiranja UI-a, razdvojili smo funkcionalnosti na tri ključne komponente ili metode: UI, Flow i Test. Svaka od ovih komponenti je odvojena metoda koja nasleđuje funkcionalnosti jedna od druge.

Na taj način, razdvajamo odgovornosti i jasno definišemo ulogu svake komponente:

- UI se odnosi na elemente korisničkog interfejsa i njihovo identifikovanje (putem lokatora pristupamo elementima);
- Flow predstavlja tok ili proceduru izvršavanja određenih radnji unutar aplikacije;
- Test sadrži konkretne testne slučajeve koji proveravaju ispravnost funkcionalnosti. Ovaj pristup omogućava bolje razumijevanje i održavanje koda.



Slika 13. Prikaz LoginUi metode

Opis LoginUi metode:

Ovde je dat primer klase LoginUi koja predstavlja dio strukture za upravljanje elementima stranice u Doxio aplikaciji:

- `package org.amplitudo.automation.app.doxio.page_elements;` – Dio koji predstavlja deklaraciju paketa u kojem se nalazi klasa LoginUi. Paket je organizaciona jedinica koja grupiše srodne klase i interfejse kako bi se olakšala njihova organizacija i upravljanje.
- `import org.amplitudo.automation.app.doxio.DoxioCommonFlows` – Ovaj red uvozi klasu DoxioCommonFlows iz paketa org.amplitudo.automation.app.doxio, što omogućava pristup njenoj funkcionalnosti unutar klase LoginUi.
- `import org.openqa.selenium.By` – Uvoz klase By iz paketa org.openqa.selenium, koja omogućava identifikaciju HTML elemenata na web stranici putem različitih selektora (npr. ID, klasa, CSS selektor).
- `public class LoginUi extends DoxioCommonFlows {` – Definicija klase LoginUi koja nasleđuje funkcionalnosti iz klase DoxioCommonFlows. Nasleđivanje omogućava da nova klasa dobije sve metode i osobine roditeljske klase.
- `public By theUsernameField() {return By.id("username")}` - Metoda theUsernameField() koja vraća selektor za polje za unos korisničkog imena. Ovde se koristi selektor "id" kako bi se pronašao odgovarajući HTML element na osnovu njegovog identifikatora.
- `public By thePasswordField() {return By.id("password")}` – Metoda thePasswordField() koja vraća selektor za polje za unos lozinke. Koristi se isti princip kao i za prethodni metod, koristeći "id" selektor.
- `public By theForgotPasswordLink() {return By.className("forgot-password-link")}`
- `public By theLoginButton() {return By.className("brt")}` - Metode koje su slične kao i prethodne i vraćaju selektore za dugme za pronalaženje odgovarajućeg HTML elementa.

Ovaj primer ilustruje kako se u klasi LoginUi definišu različite metode koje omogućavaju identifikaciju različitih elemenata na stranici Doxio aplikacije, koristeći različite selektore kao što su ID, klasa itd.

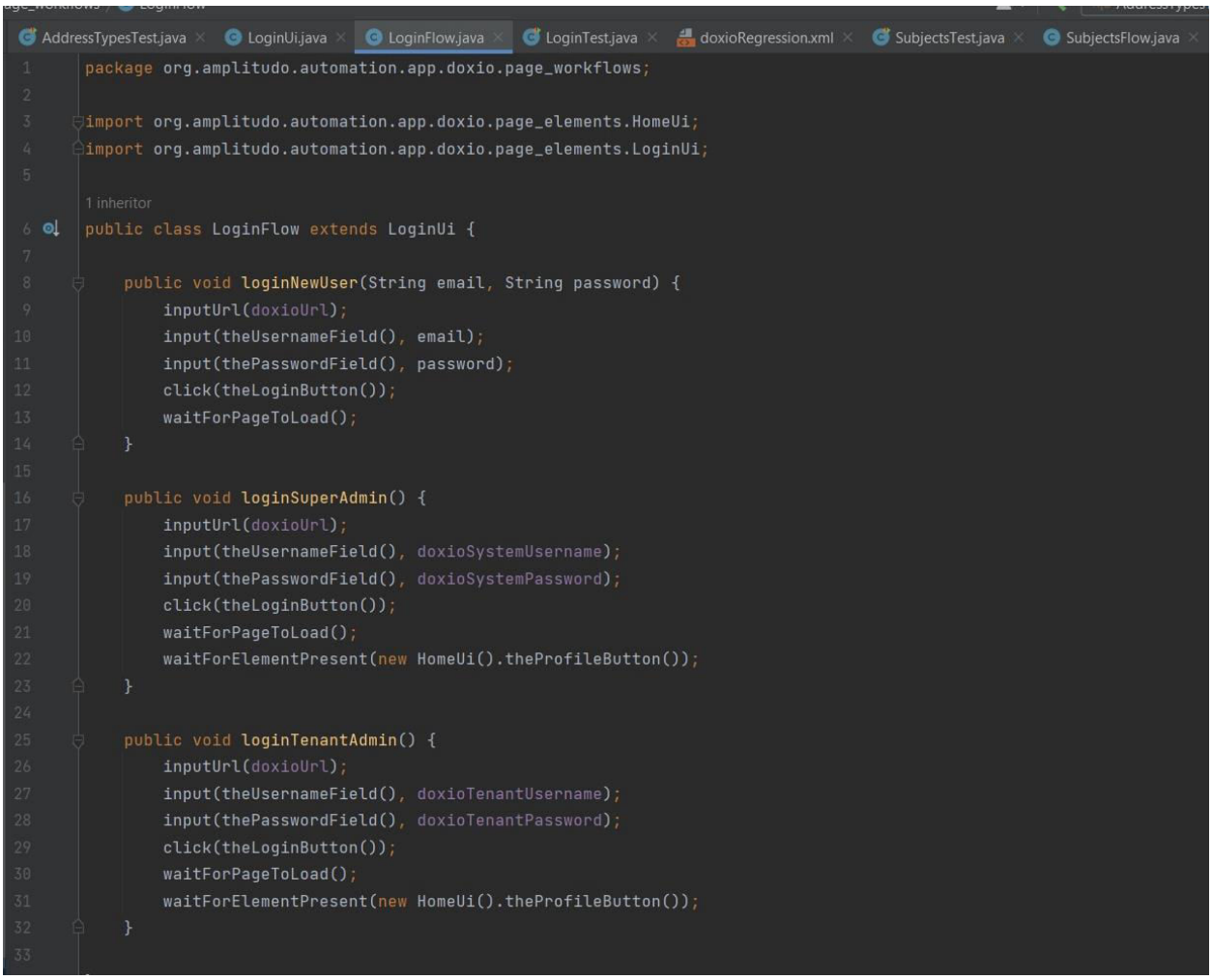
Prije izvođenja radnje na bilo kojem elementu, moramo izvršiti sljedeće korake.

1. Pregledamo element,
2. Identifikujemo/lociramo element,

3. Pronađemo element.

Da bismo pregledali elemente, kliknemo desnim klikom na element i na Inspect element, koji će dati izvorni kod tog elementa i klikom na traženi element vidjećemo izvor tog elementa. U ovom slučaju smo koristili ‘ID’ od svakog elementa, (Slika 14.).

Slika 14. Primjer pronalaska elemenata putem selektora na Login stranici



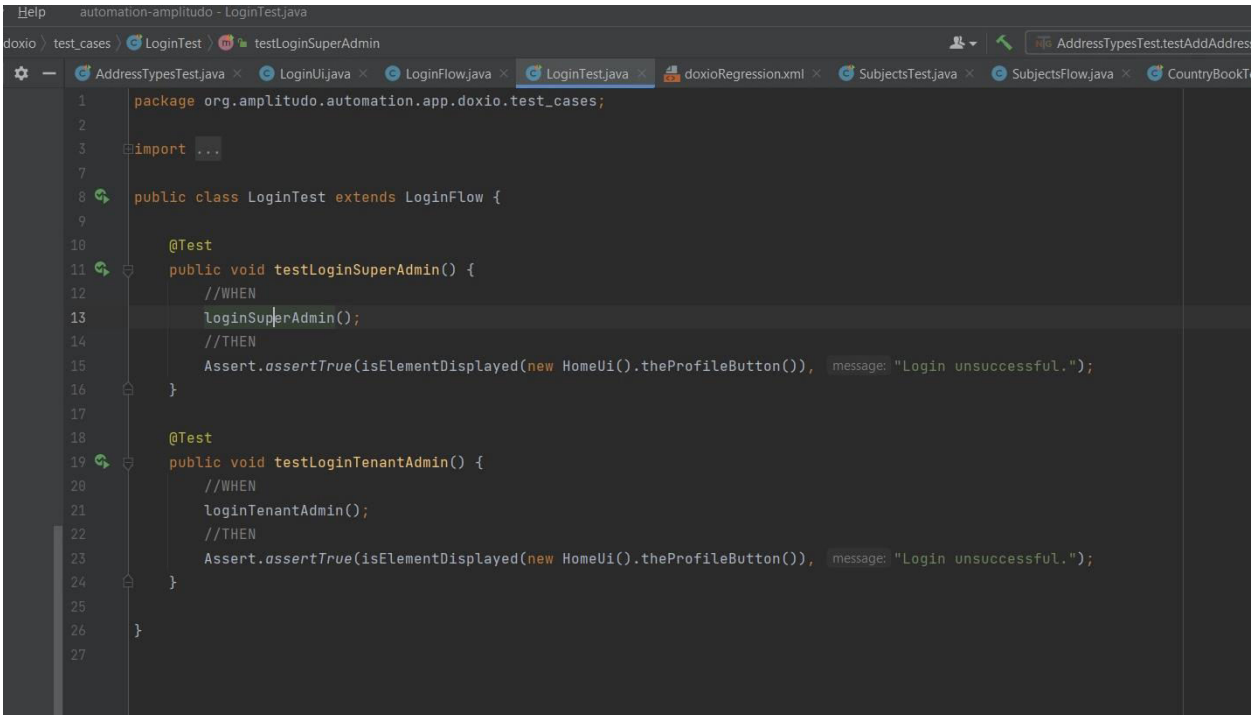
Slika 15. Prikaz metode LoginFlow

Opis LoginFlow metode:

Na slici iznad vidimo tri metode koje su slične ali se različiti tipovi korisnika loguju. Naime, imamo logovanje običnog user-a, super admina i tenant admina. Obradićemo loginTenantAdmin metodu, jer sve isto funkcioniše i za druge dvije, samo su različiti kredencijali za pristup.

- Na početku ponovo imamo import paketa, koji nam omogućavaju da pozovemo neku metodu unutar Flow-a. Objašnjenje je dato na samom početku kod LoginUi metode.
- `inputUrl(doxioUrl):` Metoda koja se koristi za unošenje URL adrese Doxio aplikacije. Ova adresa predstavlja lokaciju na kojoj se aplikacija nalazi i gdje korisnik može pristupiti prijavljivanju.
- `input(theUsernameField(),doxioTenantUsername):` Metoda za unos korisničkog imena TenantAdmin-a, koristeći polje za unos korisničkog imena definisano u klasi LoginUi.
- `input(thePasswordField(),doxioTenantPassword):` Metoda za unos lozinke TenantAdmin-a, koristeći polje za unos lozinke definisano u klasi LoginUi.

- `click(theLoginButton()):` Metoda za klik na dugme za prijavu, koje je takođe definisano u klasi LoginUi.
- `waitForPageToLoad():` Metoda koja čeka da se stranica učita nakon što se izvrši prijava.
- `waitForElementPresent(newHomeUi().theProfileButton()):` Metoda koja čeka da se pojavi dugme za profil na početnoj stranici (Home), što označava da je korisnik uspješno prijavljen.



Slika 16. Prikaz LoginTest testne metode

Opis LoginTest metode:

Kao i kod prethodne metode, i ovdje imamo dva test slučaja koja su identična, samo ih kredencijali korisnika odvajaju, tako da ćemo detaljno objasniti jednu od njih. Ovdje imamo JUnit test slučaj pod nazivom `testLoginTenantAdmin.

- `public class LoginTest extends LoginFlow{:` Ovaj red definiše klasu LoginTest koja nasleđuje funkcionalnosti iz klase LoginFlow. Nasleđivanje omogućava pristup metodama definisanim u klasi LoginFlow.
- `@Test:` Oznaka da je metoda testLoginTenantAdmin() testna metoda koja će biti izvršena kao dio testnog okruženja.

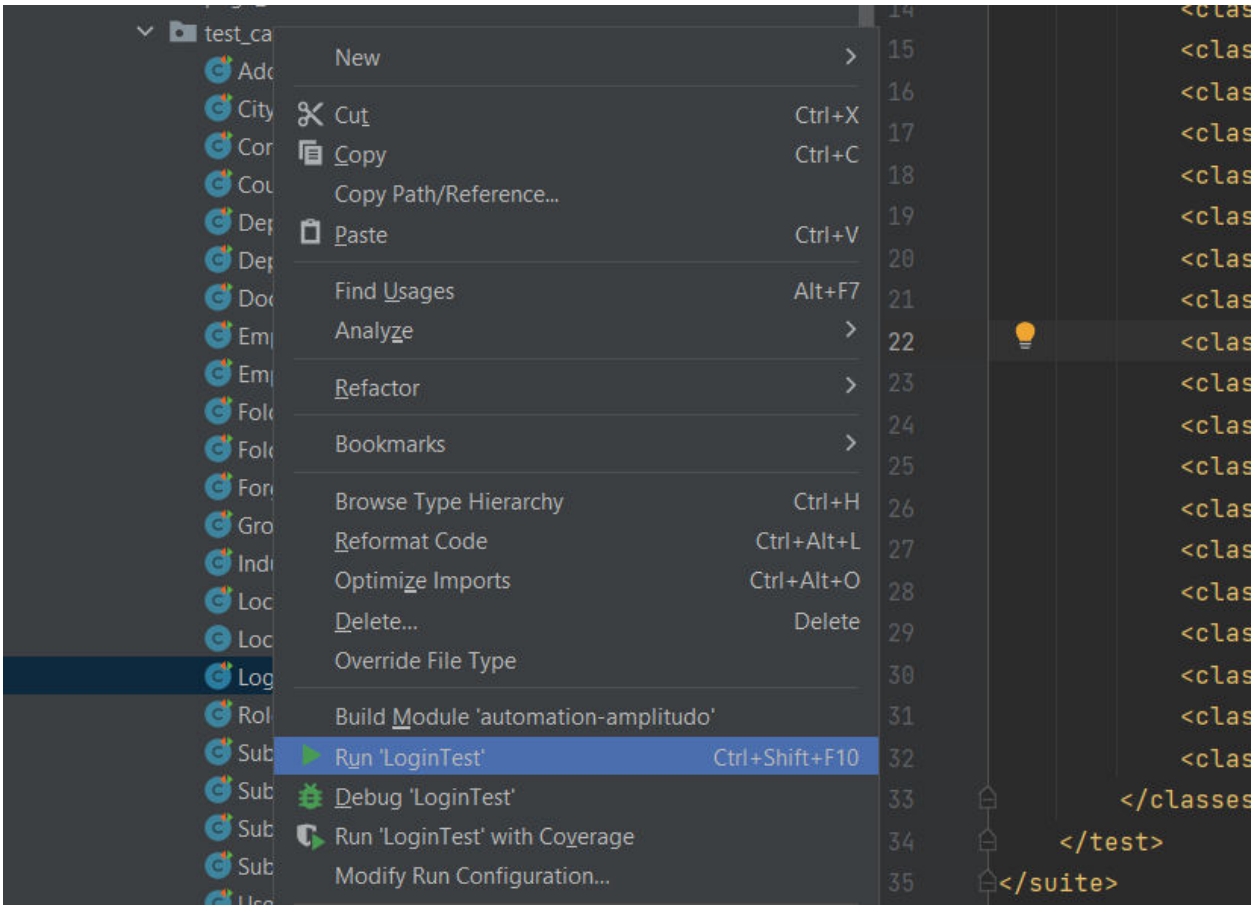
Test je podijeljen u dva logička dijela: akcija (`WHEN`) i verifikacija (`THEN`).

- `WHEN` dio sadrži poziv `loginTenantAdmin()`, gdje se vrši proces prijavljivanja korisnika kao TenantAdmin.

- `THEN` segment, tu se koriste se asertacije za verifikaciju uspješnosti operacije. Asertacija `Assert.assertTrue(isElementDisplayed(new HomeUi().theProfileButton()),"Login unsuccessful.");`: Ovaj red provjerava da li je dugme za profil vidljivo na početnoj stranici nakon prijave.

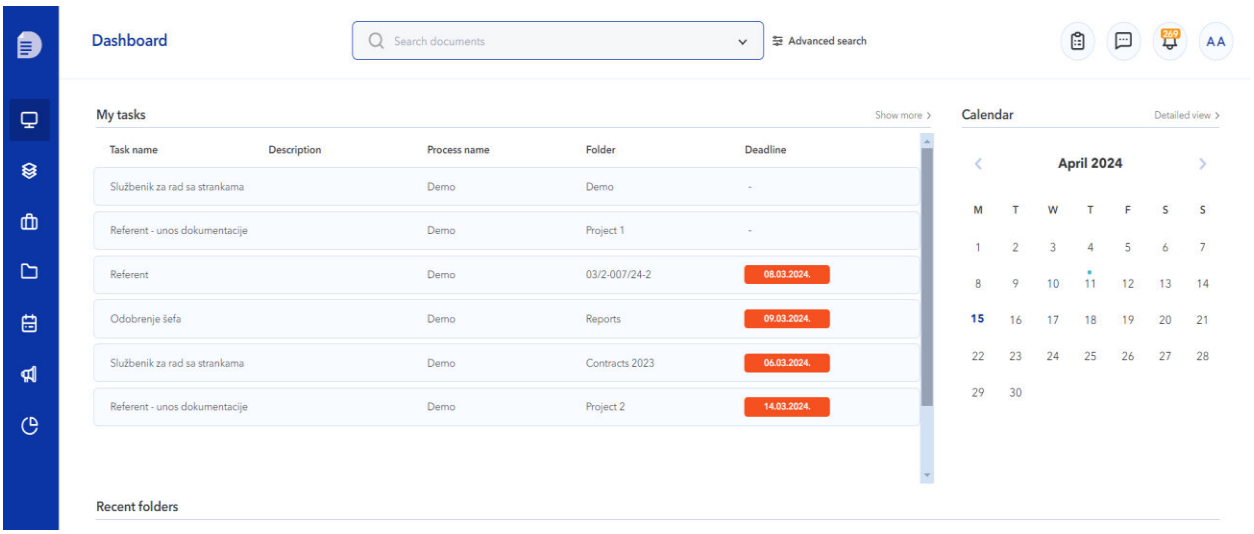
Ako dugme nije vidljivo, test će biti označen kao neuspješan, a kao dodatak se ispisuje poruka "Prijavljivanje nije uspešno." kako bi se lakše identifikovala greška u slučaju da test ne uspije.

Paket testova se može izvršiti pomoću Maven-a ili Junit-a u zavisnosti od obima testa. U ovom slučaju imamo jedan test i on se pokreće na sljedeći način: Kliknemo desnim tasterom miša na test slučaj, izaberemo Run i izaberemo JUnit test koji želimo da se izvrši. (Slika 17.).



Slika 17. Pokretanje Login testa

Nakon izvršenja, program pokreće test slučaj sa obaveštenjem sa konzole i imamo sljedeći prikaz na stranici. Test je uspješno prošao. (Slika 18.)



Slika 18. Prikaz dashboard-a nakon uspješno izvršenog Login testa

4.2 Drugi testni slučaj i njegova implementacija

Testni slučaj: Employee test (Dodavanje zaposlenih unutar sistema).

```
2 inheritors
public class EmployeesUi extends DoxioCommonFlows {

    /*
     * -------------------- GENERAL INFORMATION -------------------- //
     */

    public By theEmployeePrimaryIdentifier() {
        return By.xpath( name: "//input[contains(@class,'employeePrimaryIdentifier')]");
    }

    public By theEmployeeFirstNameField() { return By.xpath( name: "//input[contains(@class,'employeeFirstName')]"); }

    public By theEmployeeLastNameField() { return By.xpath( name: "//input[contains(@class,'employeeLastName')]"); }

    public By theEmployeeEmailAddressField() { return By.xpath( name: "//input[contains(@class,'employeeEmail')]"); }

    public By theEmployeesUserAccountField() { return By.id("user"); }

    /*
     * -------------------- COMMON ELEMENTS -------------------- //
     */

    public By theAddEmployeeButton() { return By.className("add-client-btn"); }

    public By theEmployeeSaveChangesButton() { return By.className("save-icon"); }

    public By theEmployeeUndoChangesButton() { return By.className("cancel-icon"); }

    public By theEmployeeEditChangesButton() { return By.className("edit-icon"); }

    public By theEmployeeDeleteEntryButton() { return By.className("delete-icon"); }

    public By theEmployeeEditButton() { return By.className("edit-button"); }

    public By theEmployeeBackButton() { return By.className("return-btn"); }

    /*
     * -------------------- ADDRESS -------------------- //
```

Slika 19. Prikaz EmployeesUi metode

- `By theEmployeePrimaryIdentifier()`: vraća lokator za primarni identifikator zaposlenog.

- `By theEmployeeFirstNameField()`: vraća lokator za polje imena zaposlenog.

- `By theEmployeeLastNameField()`: vraća lokator za polje prezimena zaposlenog.

- `By theEmployeeEmailAddressField()`: vraća lokator za polje e-mail adrese zaposlenog.

- `By theEmployeesUserAccountField()`: vraća lokator za polje korisničkog naloga zaposlenog.

- `By theAddEmployeeButton()`: vraća lokator za dugme za dodavanje zaposlenog.

- `By theEmployeeSaveChangesButton()`: vraća lokator za dugme za snimanje promjena.

- `By theEmployeeUndoChangesButton()`: vraća lokator za dugme za poništavanje promjena.

- `By theEmployeeEditChangesButton()`: vraća lokator za dugme za uređivanje promjena.

- `By theEmployeeDeleteEntryButton()`: vraća lokator za dugme za brisanje unosa zaposlenog.

- `By theEmployeeEditButton()`: vraća lokator za dugme za uređivanje zaposlenog.

- `By theEmployeeBackButton()`: vraća lokator za dugme za povratak na prethodni ekran.

```
21      }
22
23      public void addEmployeeWithAllFields(Company company, Person person, RandomData random) {
24          navEmployees();
25          click(theAddEmployeeButton());
26          enterEmployeeAllGeneralInfoFields(company, person);
27          click(theEmployeeSaveChangesButton());
28          click(theAddAddressButton());
29          enterEmployeeAllAddressFields(random);
30          click(theEmployeeSaveChangesButton());
31          hoverAndClickRandom(theAddContactButton());
32          enterEmployeeContactInformation(company);
33          click(theEmployeeSaveChangesButton());
34          click(theAddDepartmentButton());
35          enterEmployeeDepartmentAndWorkplaceField();
36          click(theEmployeeSaveChangesButton());
37          click(theEmployeeBackButton());
38      }
```

Slika 20. Prikaz AddEmployeeWithAllFields Flow metode

Opis AddEmployeeWithAllFileds Flow metode:
U ovom primjeru addEmployeeWithAllField je naziv metode koja služi za dodavanje zaposlenih unutar sistema. Obuhvata dodavanje zaposlenih sa popunjavanjem svih polja unutar određene forme. Unutar zagrada, Company company i Person person su parametri metode.

‘Company company’: Ovaj parametar predstavlja objekat tipa Company, što znači da metoda addEmployeeWithAllFields očekuje da se proslijedi instanca klase Company kao argument prilikom poziva. Klasa Company predstavlja informacije o kompaniji u sistemu.

‘Person person’: Ovaj parametar predstavlja objekat tipa Person, što znači da metoda addEmployeeWithAllFields očekuje da se proslijedi instanca klase Person kao argument prilikom poziva. Klasa Person predstavlja informacije o osobi koja se dodaje kao zaposleni u kompaniju.

`navEmployees()`: navigira do sekcije za upravljanje zaposlenima.

- `click(theAddEmployeeButton())`: inicira akciju dodavanja novog zaposlenog.

- `enterEmployeeAllGeneralInfoFields(company, person)`: popunjava sve opšte informacije zaposlenog koristeći podatke iz objekata `company` i `person`.

- `click(theEmployeeSaveChangesButton())`: čuva sve unesene promjene.

- `click(theAddAddressButton())`: otvara formu za dodavanje adrese.

- `enterEmployeeAllAddressFields(random)`: popunjava polja adrese sa nasumično generisanim podacima.

- `hoverAndClickRandom(theAddContactButton())`: stavlja kursor na dugme za dodavanje kontakta i klika na nasumično odabrani kontakt.

- `enterEmployeeContactInformation(company)`: unosi kontakt informacije za kompaniju.

- `click(theAddDepartmentButton())`: otvara formu za dodavanje odjeljenja.

- `enterEmployeeDepartmentAndWorkplaceField()`: popunjava informacije o odjeljenju i radnom mjestu zaposlenog.

- `click(theEmployeeBackButton())`: vraća korisnika na prethodni ekran.

Slika 21. JUnit test metod `testAddEmployeeWithAllFields`- testna metoda za dodavanje zaposlenog

U testu za dodavanje zaposlenog, prvo kreiramo novu instancu klase Company pomoću konstruktora new Company (). Zatim Person pomoću konstruktora new Person () i random pomoću konstruktora newRandomData (). Ovi koraci su neophodni kako bismo imali validne objekte kompanije sa kojima izvršavamo test. Nakon inicijalizacije, ovi objekti će biti korišćeni za simuliranje scenarija dodavanja zaposlenog u određenu kompaniju."

- ‘GIVEN’ priprema okruženje za proces prijavljivanja korisnika kao TenantAdmin, što omogućava nesmetano izvršavanje glavne testne metode."

- `goLoginTenantAdmin()`: izvršava proces prijave u aplikaciju kao administator (tenant).

-`WHEN` dio sadrži poziv ` addEmployeeWithAllFields(company, person, random)`: gdje poziva metod definisan na prvoj slici za dodavanje zaposlenog sa svim poljima.

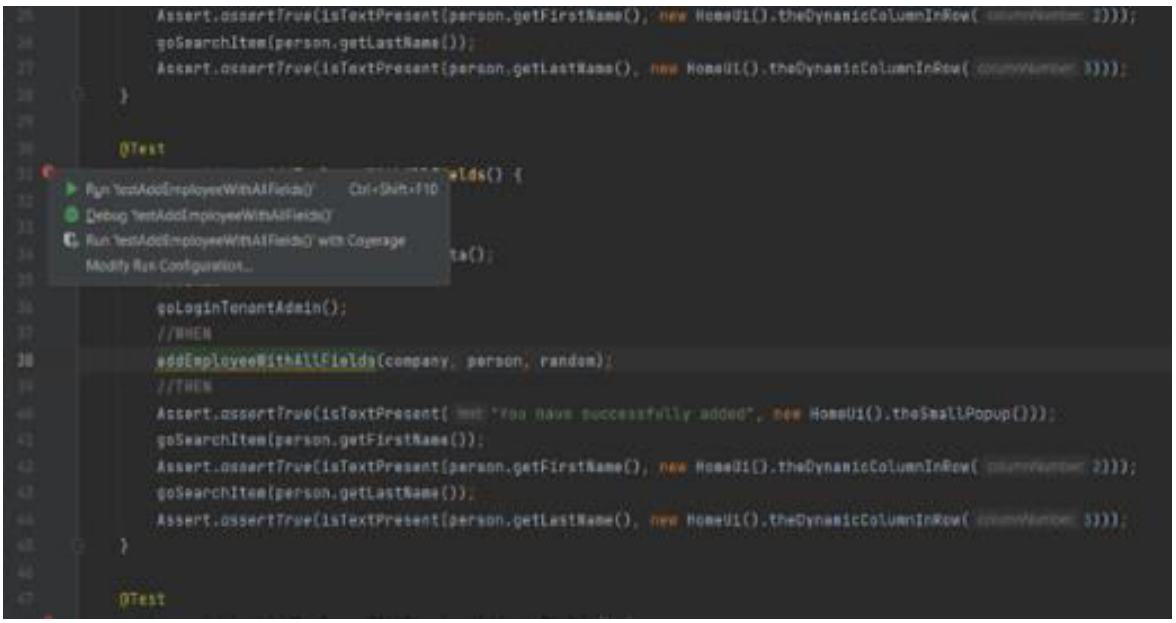
- `THEN` segment, tu se koriste se asertacije za verifikaciju uspešnosti operacije.

- `Assert.assertTrue(isTextPresent(...))`: provjerava da li je tekst prisutan na stranici koristeći asercije, što je ključni dio testiranja da bi se potvrdilo da su operacije uspešno izvršene.

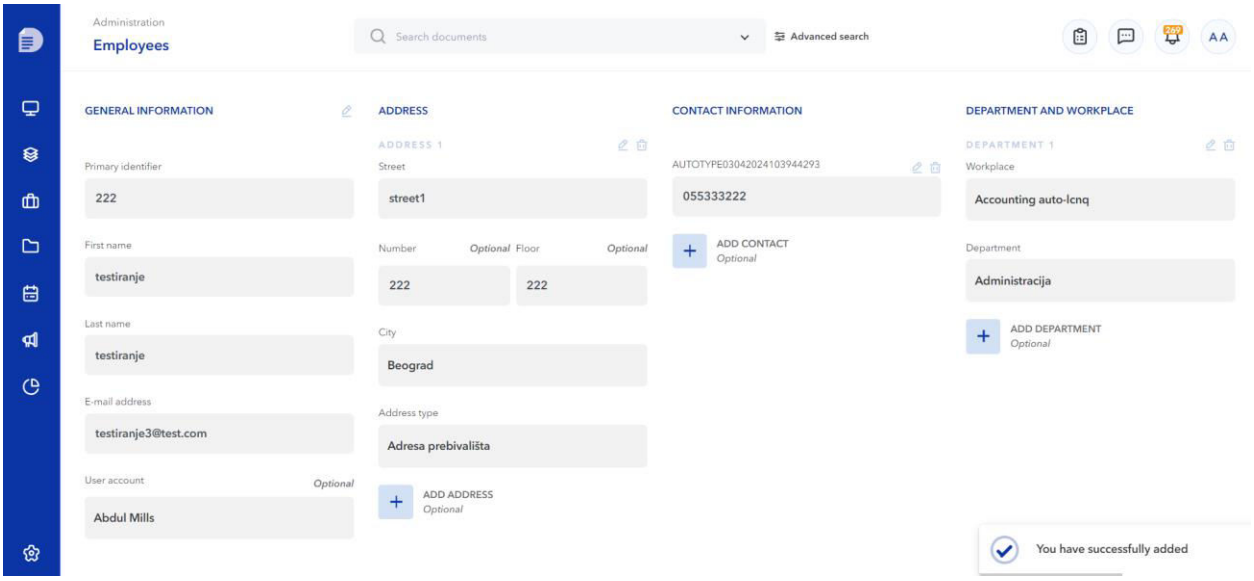
"goSearchItem(person.GetFirstName()) izvršava sljedeću funkciju goSearchItem() koja se koristi za pretragu određenog elementa unutar aplikacije. Kao argument ove funkcije, prosleđujemo ime osobe (person.GetFirstName()) koje se dobija pozivanjem metode GetFirstName() nad objektom person. Ovaj korak obično predstavlja akciju korisnika u aplikaciji, gde se korisnik traži na osnovu svog imena kako bi se pronašli odgovarajući podaci ili informacije."

Nakon ove metode ponovo imamo asertaciju, gdje provjeravamo da li je korisnik kojeg smo pretražili u tabeli pomoću ‘`goSearchItem(person.GetFirstName())`’ metode uspješno prikazan u tabeli, što samim tim provjerava i da li je dodavanje uspješno izvršeno.

Kada imamo više metoda označenih sa @Test anotacijom unutar jednog glavnog testa (klase koja sadrži testove) i želimo da pokrenemo samo jedan od njih, u ovom slučaju test ‘AddEmployeeWithAllFields’ to radimo na sljedeći način. Svaka metoda koja predstavlja test ima svoje dugme za pokretanje testa, koje je također označeno kao "Run". Klikom na to dugme, izabrani test će biti pokrenut, a zatim u sljedećem prikazu (Slika 23.) vidimo kako se taj izvršeni test prikazuje na sistemu.

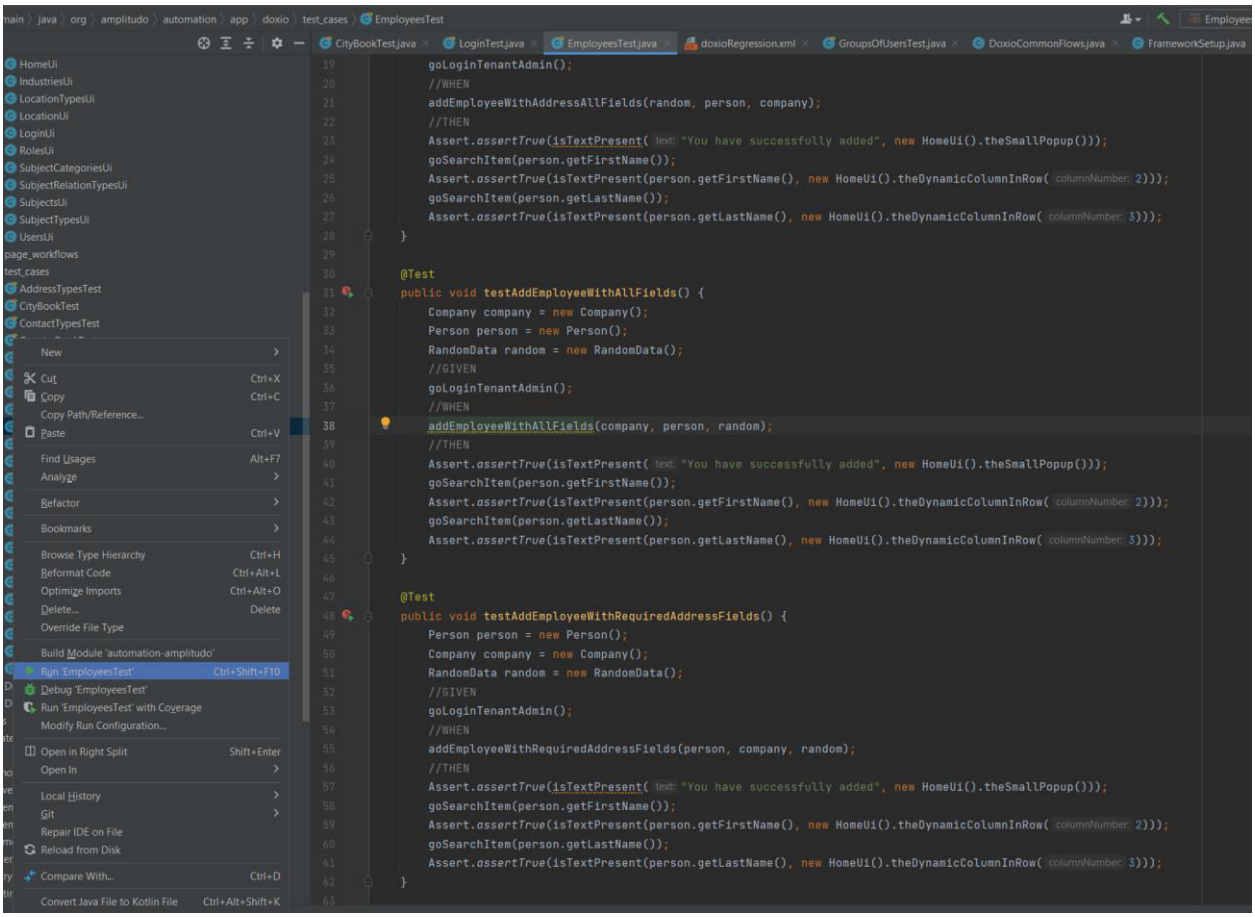


Slika 22. Prikaz pokretanja jednog testa unutar klase testova



Slika 23. Prikaz ekrana nakon uspješno završenog 'AddEmployeeWithAllFields' testa

U slučaju da želimo da pokrenemo sve testove odjednom, koji su unutar glavne metode, to bismo radili na način da kliknemo na glavni test i izaberemo opciju 'Run As', a zatim izaberemo 'Junit Test'. (Slika 24.)



Slika 24. Pokretanje svih testova unutar glavne metode

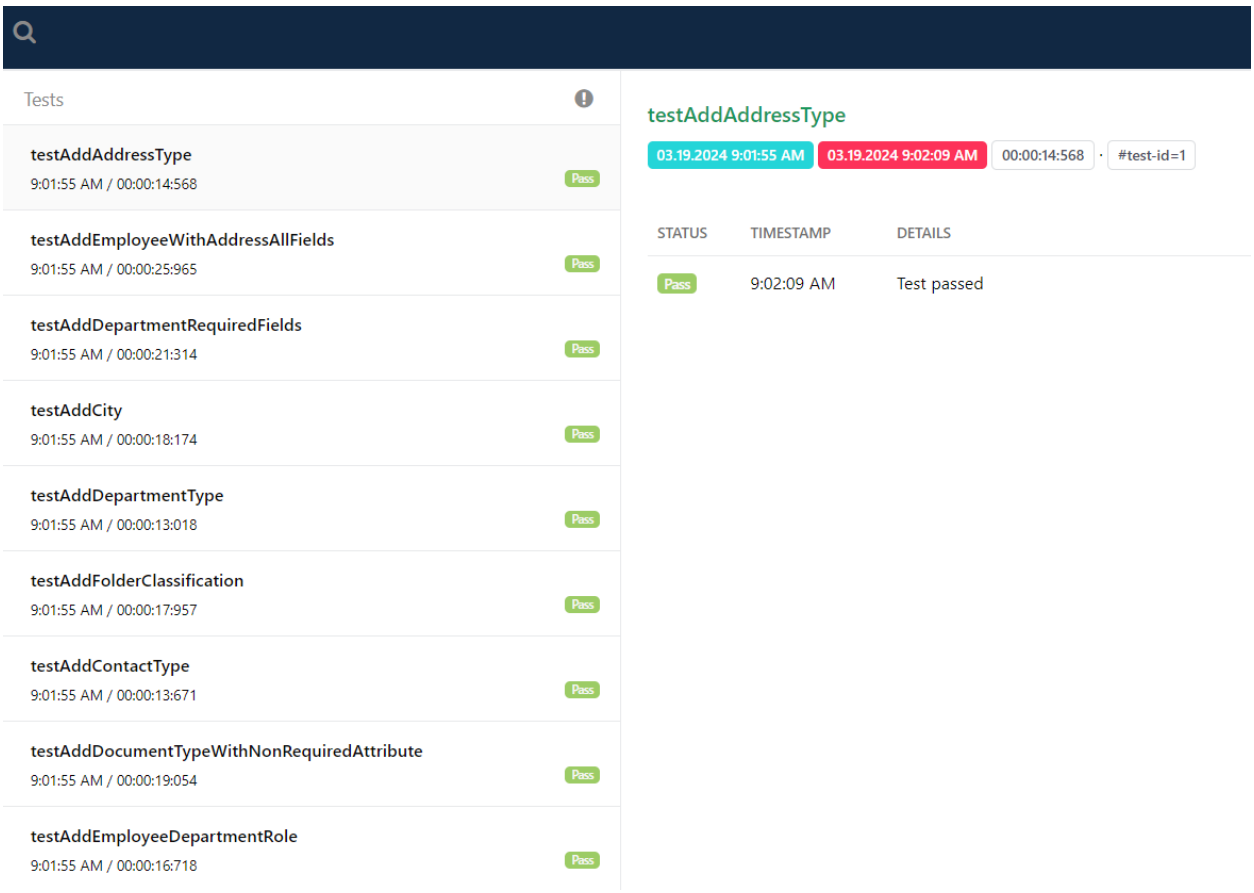
4.3 Rezultati pokrenutih testova pomoću Seleniuma

Iako je sprovedeno kompletno testiranje koje je obuhvatilo 112 različitih testova, iz praktičnih razloga fokus je na detaljnu analizu samo dva testa. Ova dva testa, koji obuhvataju proces prijave korisnika i dodavanje novih zaposlenih u sistem, odabrana su kako bismo bolje objasnili metodologiju i funkcionalnosti sistema. Ovaj pristup nam omogućava da dublje istražimo ove ključne funkcionalnosti, uzimajući u obzir obim koda i opsežnost testova. Na taj način, iako ne obuhvatamo sve aspekte aplikacije, pružamo detaljan uvid u ključne procese i procedure unutar sistema putem Selenium alata.

Kada pokrećemo testove u Seleniumu, jedan od rezultata može biti generisanje 'index.html' datoteke. Ova datoteka je obično glavni izvještaj koji sadrži detaljne informacije o svim izvršenim testovima.

Komponenta unutar Seleniuma koja generiše ovakve izvještaje je obično TestNG (Test Next Generation). Kada se testovi pokrenu kroz TestNG okvir, on automatski generiše 'index.html' izvještaj unutar framework-a koji sadrži informacije o testovima koji su prošli, kao i onima koji nijesu, potrebnom vremenu za izvršavanje svakog testa, logovima, kao i ukupnim rezultatima.

Ovaj 'index.html' izvještaj obično pruža preglednu i lako razumljivu prezentaciju rezultata testova, što olakšava timovima da identifikuju probleme i prate napredak testiranja.



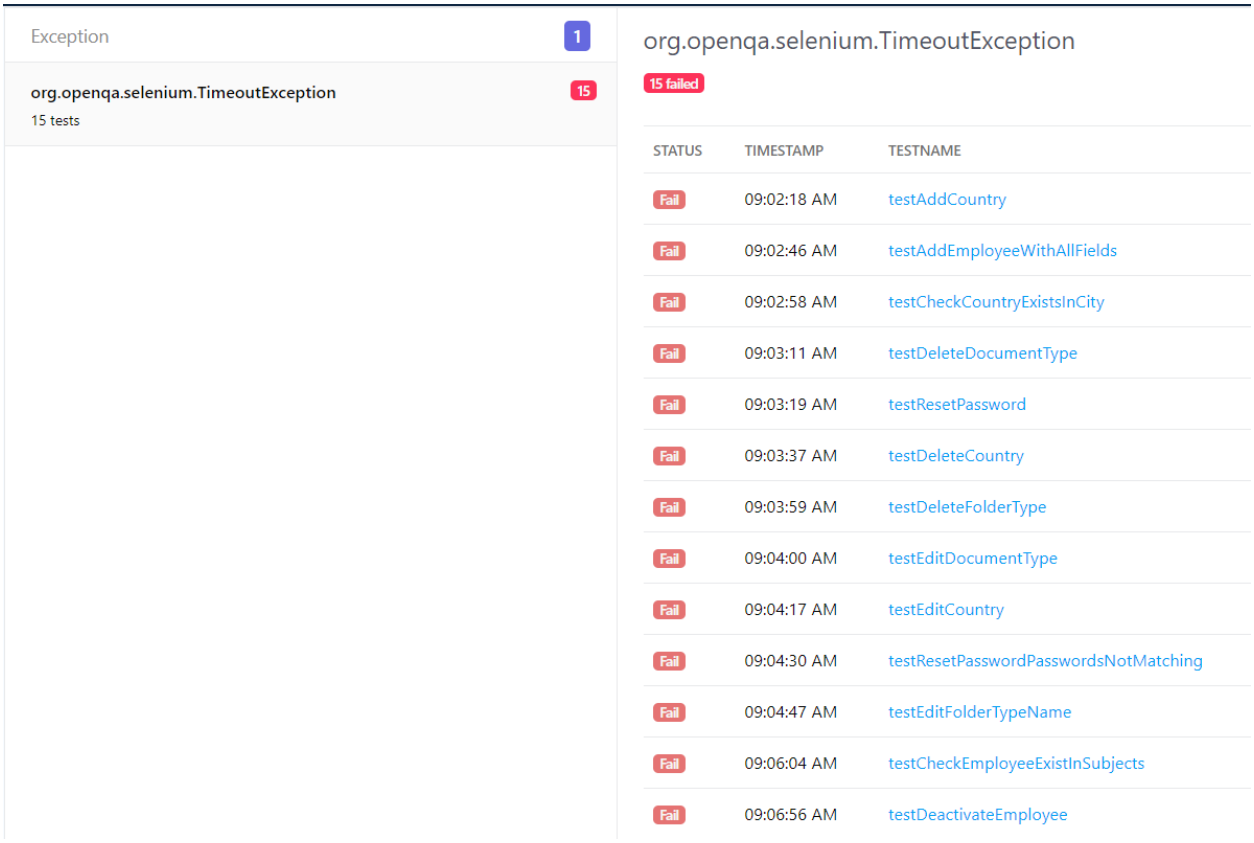
Slika 25. Prikaz prve stranice izvještaja (nije prikazan ukupan broj testova, zbog bolje preglednosti)

1. Glavni panel sa listom testova (lijeva strana):

- Svaki test koji je pokrenut ima svoj naslov, npr: ‘testAddAddressType’, ‘testAddEmployeeWithAllFields’ i sl.
- Vrijeme pokretanja je navedeno pored svakog testa (npr., 9:01:55 AM).
- Trajanje svakog testa je takođe navedeno u formatu hh:mm:ss:ms (npr., 00:00:14:568).
- Desno od trajanja, nalazi se status svakog testa, u ovom slučaju "Pass" što znači da je test uspješno prošao.

2. Detalji selektovanog testa (desna strana):

- Ovo je detaljni prikaz za test koji je trenutno otvoren u listi testova, u ovom slučaju "testAddAddressType".
- Prikazuje status "Pass", dva vremenska prikaza (početak i kraj: 03.19.2024 9:01:55 / 9:02:09 AM), ukupno trajanje testa (00:00:14:568), i identifikator testa (#test-id=1).
- Postoji i poruka "Test passed" što potvrđuje da je test uspješno izvršen.

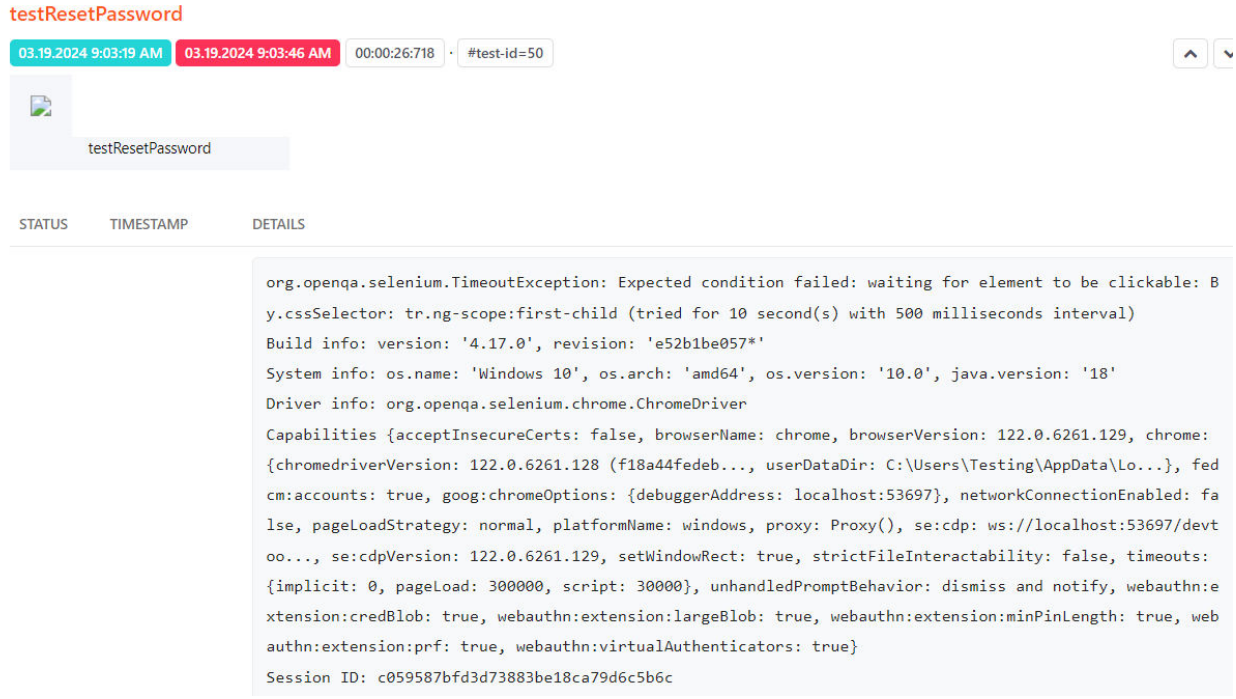


Slika 26. Prikaz druge stranice izvještaja (Ovdje imamo prikazane testove koji nijesu prošli)

1. 'org.openqa.selenium.TimeoutException' (lijeva strana slike) označava da je došlo do greške u izvršavanju testova. TimeoutException je vrsta izuzetka koji se javlja kada Selenium nije mogao pronaći ili izvršiti određenu radnju unutar očekivanog vremenskog ograničenja.
2. Panel sa listom testova koji nijesu prošli (desna strana):

- Na početku imamo prikaz statusa koji je ‘Fail’, što znači da test nije prošao.
- Pored toga imamo ‘Timestamp’ koji prikazuje vrijeme kad je test pokušao da se izvrši i iz nekog razloga nije prošao uspješno. Npr. (09:02:18 AM)
- Nakon toga imamo i ‘TestName’, koji nam ustvari vraća naziv testa koji nije prošao. Npr. ‘TestAddCountry’, ‘TestResetPassword’ i sl.

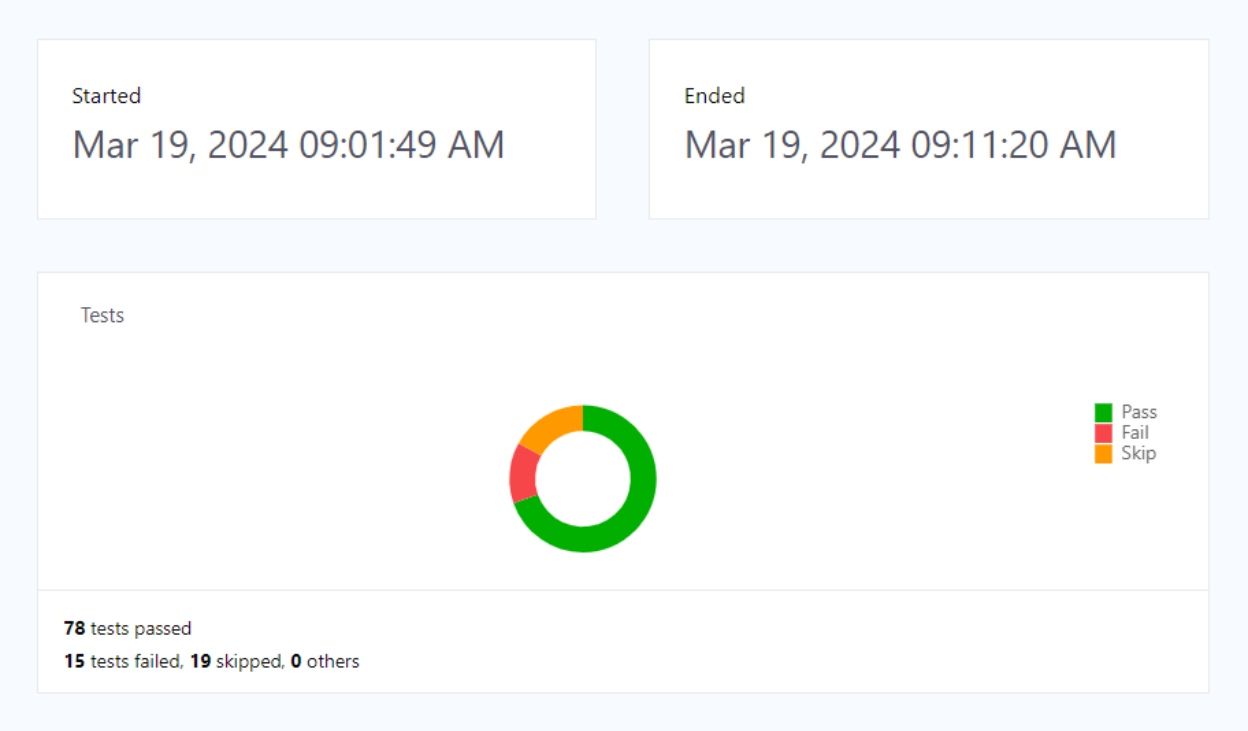
Ukoliko kliknemo na naziv testa koji je označen kao neuspješan, otvoriće se detaljni prikaz koji pruža dublji uvid u razloge neuspjeha tog testa. Ova funkcionalnost omogućava timovima za testiranje da detaljno istraže uzroke neuspjeha i identifikuju korake koje treba preduzeti kako bi se problemi riješili i testovi uspješno izvršili u budućnosti, (Slika 27.).



Slika 27. Detaljniji prikaz testa koji je označen kao neuspješan

Prva linija koda “org.openqa.selenium.TimeoutException: Expected condition failed: waiting for element to be clickable: By.cssSelector: tr.ng-scope:first-child (tried for 10 second(s) with 500 milliseconds interval)“ precizno identifikuje da je došlo do ‘TimeoutException’ greške prilikom čekanja da element postane klikabilan na stranici. Takođe, navodi se i CSS selektor traženog elementa ("tr.ng-scope:first-child"), kao i vremensko ograničenje u kome je Selenium pokušavao da pronađe element (10 sekundi, sa intervalom od 500 milisekundi između pokušaja).

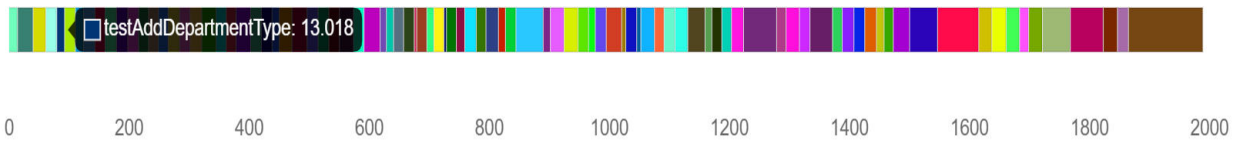
S obzirom na ovu grešku, možemo pretpostaviti da bi rješenje za ovaj problem uključivalo prilagođavanje testa kako bi se osiguralo da se element koji se očekuje postane klikabilan unutar očekivanog vremenskog perioda. Ovo može značiti prilagođavanje vremenskih ograničenja ili implementaciju dodatnih provjera kako bi se osiguralo da se element pojavi na stranici prije nego što se pokuša njegovo klikanje. Takođe, može biti korisno pregledati strukturu stranice i provjeriti da li postoje promjene koje bi mogle uticati na identifikaciju ili interakciju sa traženim elementom.



Slika 28. Prikaz posljednje stranice izvještaja

Na slici iznad imamo prikaz i posljednje stranice izvještaja, gdje su rezultati testiranja za određeni vremenski period. Testiranje je započeto 19. marta 2024. godine u 09:01:49 ujutro, a završilo se u 09:11:20AM. Ovdje primjećujemo da je testiranje završeno u rekordnom roku, tačnije za 9 minuta i 31 sekundu, s obzirom da je pokrenuto 112 testova u isto vrijeme. Od toga ukupno je 78 testova koji su prošli, dok je 15 testova neuspjelih. Takođe, vidimo da je 19 testova preskočeno, dok nije bilo drugih slučajeva.

Pored ovakvog prikaza izvještaja takođe je generisan i ‘Timeline’ prikaz:



Slika 29. Prikaz vremenske crte ili timeline izvršenih testova

Ova slika prikazuje vremensku crtu ili "timeline" koji predstavlja raspored izvršavanja testova u okviru testnog suite-a. Svaki segment na vremenskoj crti predstavlja jedan test, a različite boje predstavljaju različite testove ili grupe testova.

Vremenska skala je označena brojevima (npr., 0, 200, 400, ... 2000), koji predstavljaju milisekunde, sekunde ili neku drugu jedinicu vremena. Ovaj prikaz je veoma bitan jer govori da

se svaki test izvršava sekventno, jedan za drugim, i da svaki test zauzima određenu količinu vremena na crti. Kursor miša trenutno stoji na jednom testu 'testAddDepartmentType' koji je trajao 13.018 jedinica vremena. Ovo može biti korisno za vizuelno predstavljanje trajanja testova i identifikovanje bilo kojih testova koji značajno duže traju od ostalih, što bi moglo ukazivati na potencijalne probleme ili oblasti za optimizaciju.

4.3.1 Upoređivanje i rezultati – Vrijeme i brzina potrebna za izvršavanje testova (Manuelno i automatizovano testiranje)

Manuelno izvršavanje testova najviše zavisi od kompleksnosti testnog slučaja. Obzirom na to da su svi testovi različiti, teško se može reći sa sigurnošću koliko bi trajanje bilo, ali u prosjeku se može reći da se minimum duplo brže izvršavaju automatski testovi od manuelnih.

Iz ovog razloga biće odrađeno poređenje u brzini kroz par konkretnih scenarija sa sistema od ukupnih 112 testova koji su napisani za automatizaciju testiranja. Počećemo sa 5 testova, zatim ćemo povećati broj testova na 10, pa na 15 i na kraju na 20, jer se razlike u vremenu izvršavanja povećavaju sa povećanjem broja testova.

Broj testova	Manuelno izvršavanje(hh:mm:ss:ms)	Automatizovano izvršavanje(hh:mm:ss:ms)
5	00:05:09:44	00:00:40:18
10	00:09:58:74	00:01:25:02
15	00:15:33:07	00:01:55:01
20	00:21:22:13	00:02:22:14

Tabela 1. Tabela sa rezultatima gdje je prikazana razlika u brzini izvršavanja testova između manuelnog i automatizovanog testiranja

U odnosu na tabelu iznad možemo da ustanovimo koliko je automatizovano testiranje, odnosno izvršavanje testnih slučajeva brže od manuelnog. ako se povećava broj testova, vreme izvršavanja se povećava i u manuelnom i u automatizovanom okruženju. Vrijeme potrebno za izvođenje testova je zabilježeno i uočeno je da je za manuelno testiranje potrebno više vremena u odnosu na automatsko testiranje.

Veoma je bitno da se razlika u vremenu između manuelnog i automatizovanog izvršavanja povećava sa povećanjem broja testova. Na početku smo imali situaciju gdje je svih 112 testova sa sistema pokrenuto, što znači da bi manuelno testiranje oduzelo znatno više vremena ako

bismo radili testiranje regresije, odnosno testiranje čitavog sistema. Takođe, bitno je spomenuti da je automatizovano testiranje više bazirano na testiranju happy path-a sistema. Ovo znači da tim putem osiguravamo glavne funkcionalnosti ili korake koje korisnici najčešće koriste, a koji bi trebali da prođu bez grešaka ili problema. U ovom slučaju samo za manuelno testiranje bi nam trebalo više resursa, uključujući ljudske resurse i vrijeme, dok automatizacija omogućava bolje iskorištenje hardverskih resursa.

Parametri	Manuelno testiranje	Automatizovano testiranje
1. Vremenski zahtjevno	DA	NE
2. Snimanje testnih slučajeva	NE	DA
3. Automatizovane testne skripte	NE	DA
4. Dostupnost alata	NE	DA
5. Veća preciznost	NE	DA
6. Brže vrijeme obrade	NE	DA
7. Istraživačko testiranje	DA	NE
8. Veći inicijalni troškovi/ulaganje	NE	DA
9. Veća pouzdanost	NE	DA
10. Isplativo za regresiono testiranje sa manjim obimom	DA	NE
11. Isplativo za regresiono testiranje sa većim obimom	NE	DA
12. Paralelno izvršavanje	NE	DA

Tabela 2. Poređenje manuelnog i automatizovanog testiranja u odnosu na dostupne parametre

Tabela 2. prikazuje razliku u korišćenju parametara u odnosu na izvršene testove, i zaključeno je da automatsko testiranje ima prednosti nad manuelnim testiranjem. Kroz samostalno mjerenje vremena za manuelno testiranje, analizirane su razlike u parametrima kao što su tačnost, brzina, pouzdanost i efikasnost između ova dva tipa testiranja.

4.4 Performanse Seleniuma

U slučaju gdje imamo prikazan izvještaj sa rezultatima automatizovanog testiranja, unutar kojeg je istovremeno pokrenuto ukupno 112 testnih slučajeva je upravo riječ o paralelizaciji ili paralelnom izvršavanju testova. Paralelizacija ima za cilj poboljšanje performansi testa izvođenjem testova istovremeno, dok serijalizacija osigurava da se testovi izvršavaju jedan za drugim. Izbor ove metode zavisi od faktora kao što su potrebe testa, raspoloživi resursi i brzina izvršenja.

Na koji način pokrećemo paralelno testiranje?

U ovakvim slučajevima, možemo koristiti atribut "parallel". Atribut "parallel" tag-a "suite" prihvata četiri vrijednosti:

1. tests - Svi test slučajevi unutar <test> taga u XML datoteci za testiranje će se pokrenuti paralelno,
2. classes - Svi test slučajevi unutar jedne Java klase će se pokrenuti paralelno,
3. methods - Svi metodi sa @Test anotacijom će se izvršavati paralelno,
4. instances - Test slučajevi u istom primjeru će se izvršavati paralelno, ali će dva metoda iz dva različita primjera raditi u različitim nitima.

Paralelno testiranje nudi nekoliko prednosti, uključujući:

1. Smanjenje vremena izvršenja: Pokretanjem testova istovremeno na više niti ili procesa, paralelno testiranje značajno smanjuje ukupno vrijeme izvršenja. To omogućava brži povratnih informacija o stabilnosti i kvalitetu testiranog softvera.
2. Povećana pokrivenost testova: Sa paralelnim testiranjem, više test slučajeva može se izvršiti istovremeno, što dovodi do povećane pokrivenosti testova. To pomaže u identifikaciji problema i grešaka u različitim scenarijima i konfiguracijama, [8].
3. Efikasnost testiranja: Paralelno testiranje maksimizira iskorištenje resursa distribuiranjem testova na više mašina, niti ili procesa. Optimizira dostupne hardverske resurse i minimizira vrijeme neaktivnosti, poboljšavajući efikasnost testiranja.
4. Rana detekcija grešaka: Pokretanje testova paralelno omogućava bržu detekciju grešaka i problema. Greške se mogu identifikovati i adresirati ranije u razvojnom ciklusu, što dovodi do bržeg rješavanja i smanjenja ukupnih troškova.
5. Povećana skalabilnost: Paralelno testiranje je visoko skalabilno i lako može nositi sa velikim brojem test slučajeva. Kako broj testova raste, paralelna izvršenja osiguravaju da testiranje ostane efikasno bez žrtvovanja brzine ili tačnosti.

6. Pouzdani rezultati: - Paralelno testiranje pomaže u identifikaciji problema vezanih za konkurentnost, kontenciju resursa i paralelno izvršenje. Pokretanje testova istovremeno pruža vrijedne uvide u potencijalne probleme koji mogu biti očigledni u sekvencijalnom izvršenju.

Za razliku od paralelizacije, Serijalizacija radi na principu da se testni slučajevi izvršavaju redom, što potencijalno može da dovede do toga da više vremena treba da se izvrši svaki test. Koristi resurse na kontrolisaniji način od paralelizacije, izbjegavajući moguće konflikte i uslove prilikom izvršavanja testova. Serijalizacija je dosta pogodna za testne scenarije koji su zavisni od nekih drugih, ili za specifične testove kojima je veoma bitan redosljed izvršavanja [23], [24].

Primjer Paralelizacije:

```
public class ParallelExecutionDemo {

    public static void main(String[] args) {

        TestNG testng = new TestNG();

        testng.setTestClasses(newClass[]{  ParallelExecutionTests.class
});

        testng.setParallel("methods");

        testng.run();}}
```

Primjer Serijalizacije:

```
import org.testng.TestNG;

public class SerializationExecutionDemo {

    public static void main(String[] args) {

        TestNG testng = new TestNG();

        testng.setTestClasses(newClass[]{
SerializationExecutionTests.class });

        testng.run();}}
```

Glavne razlike između koda za paralelizaciju i serijalizaciju:

U kodu za paralelizaciju (ParallelExecutionDemo), postavljamo paralelno izvršenje korištenjem metode setParallel("methods"). Ovaj dio koda omogućava izvršenje test metoda istovremeno.Za

serijalizaciju (SerializationExecutionDemo), koristimo testnu klasu, koja je glavna da se prepozna da se testovi izvršavaju jedan za drugim [23].

U oba slučaja, koristimo TestNG biblioteku za konfigurisanje i izvršavanje testova.Takođe u oba primjera, metoda setTestClasses koristi se za postavljanje test klasa koje će biti izvršene.

Ukratko, kod za paralelizaciju omogućava efikasnije izvršavanje testova istovremeno, dok serijsko izvršava testove jedan za drugim, potencijalno rezultirajući dužim vremenom izvršenja, [24].

4.5 Perspektiva testera: Procjena Tehničkog iskustva i stavova o automatizaciji testiranja

Sa sve većim i kompleksnijih sistemima i aplikacijama, testeri se suočavaju sa izazovima u pronalaženju efikasnih metoda testiranja. Jedan od ključnih aspekata ovog procesa je svakako automatizacija testiranja. Međutim, pored benefita automatizacije, testeri se suočavaju sa nizom izazova i prepreka prilikom implementacije ovog pristupa. Ovi izazovi uključuju složenost postavljanja automatizovanih testova, održavanje test skripti, nedostatak stručnosti u korišćenju alata za automatizaciju i slično. U ovom istraživanju, "ispitanici" se odnose na QA testere koji su učestvovali u anketi, pružajući svoje iskustvo i uvide u procese testiranja softvera. Njihova angažovanost i odgovori doprinose dubljem razumijevanju perspektiva unutar ove specifične zajednice stručnjaka.

Aspekti ankentnog istraživanja:

Anketa je kreirana sa ciljem da istraži:

- Nivo tehničkog iskustva ispitanika u oblasti testiranja softvera.
- Stavove ispitanika o važnosti automatizacije testiranja u razvoju softvera.
- Učestalost korištenja automatizacije testiranja u njihovim poslovima.
- Glavne prednosti i izazove koje testeri doživljavaju prilikom implementacije automatizacije testiranja.
- Očekivanja u pogledu doprinosa automatizacije testiranja kvaliteta softvera.

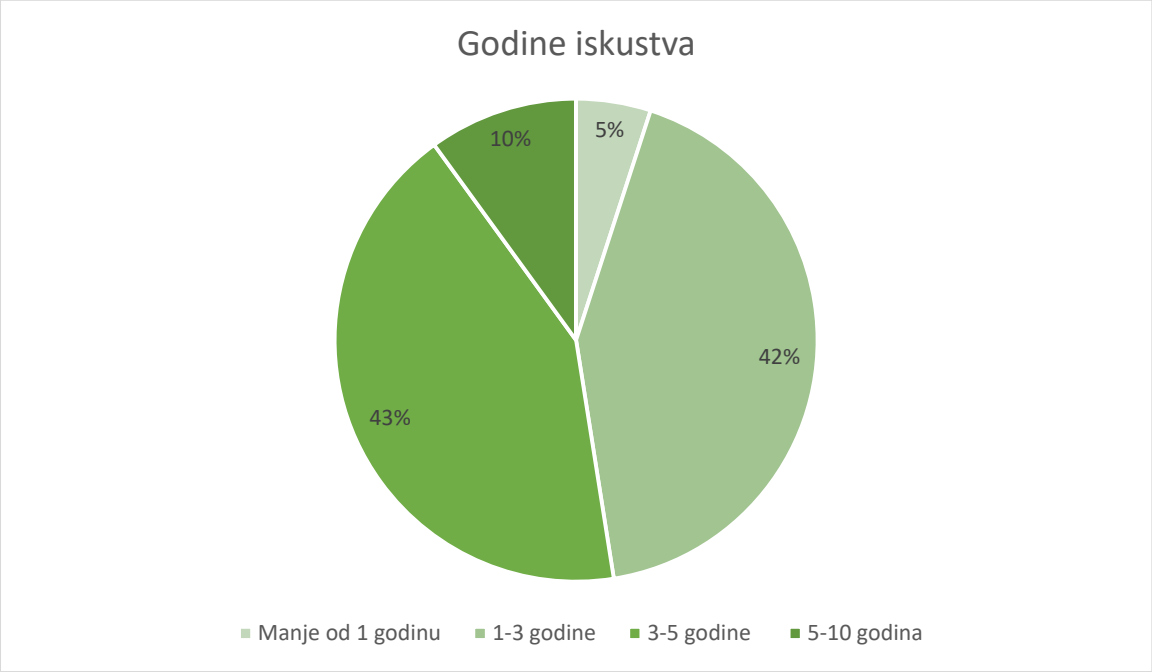
U narednim tabelama i grafikonima biće prikazani rezultati ankete na osnovu datih odgovora.

1. Godine iskustva u oblasti testiranja softvera

Prema tabeli koja prikazuje godine iskustva u testiranju, broj ispitanika sa iskustvom između 1 i 3 godine se podudara sa brojem ispitanika koji imaju iskustvo između 3 i 5 godina, pri čemu nijedan ispitanik nema preko 10 godina iskustva.

Godine iskustva				
Manje od 1 godinu	1-3 godine	3-5 godine	5-10 godina	Više od 10 godina
2	17	17	4	0

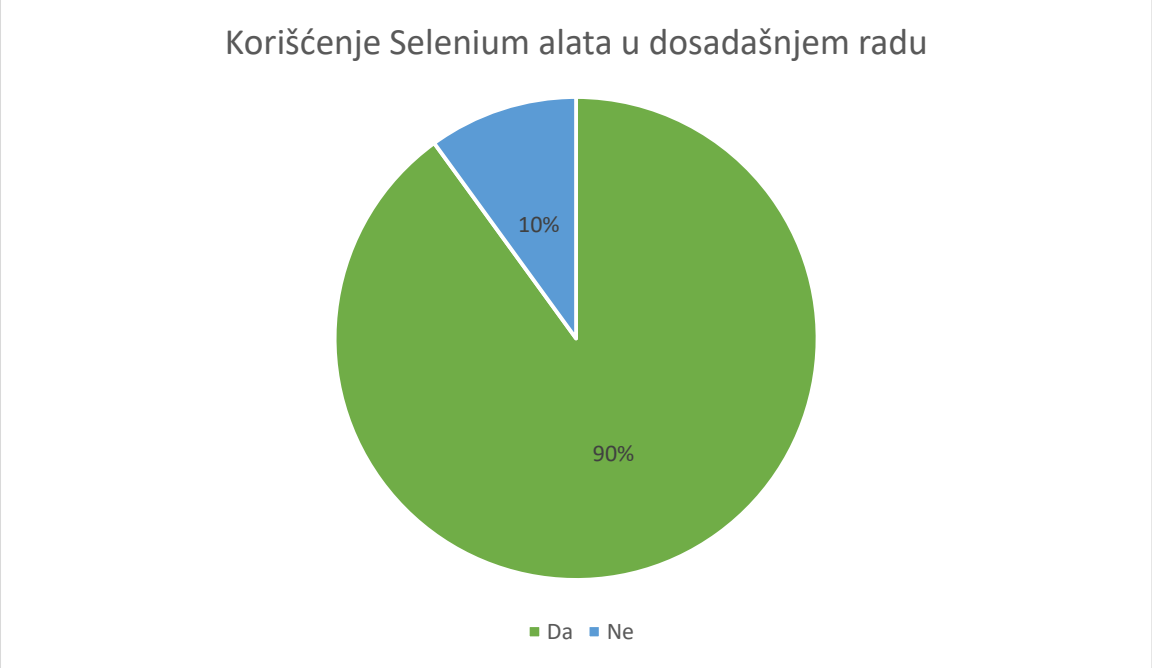
Tabela 3. Prikaz odgovora o godinama iskustva u testiranju softvera učesnika ankete na uzorku od 40 odgovora



Grafik 1. Procentualni prikaz odgovora o godinama iskustva u testiranju

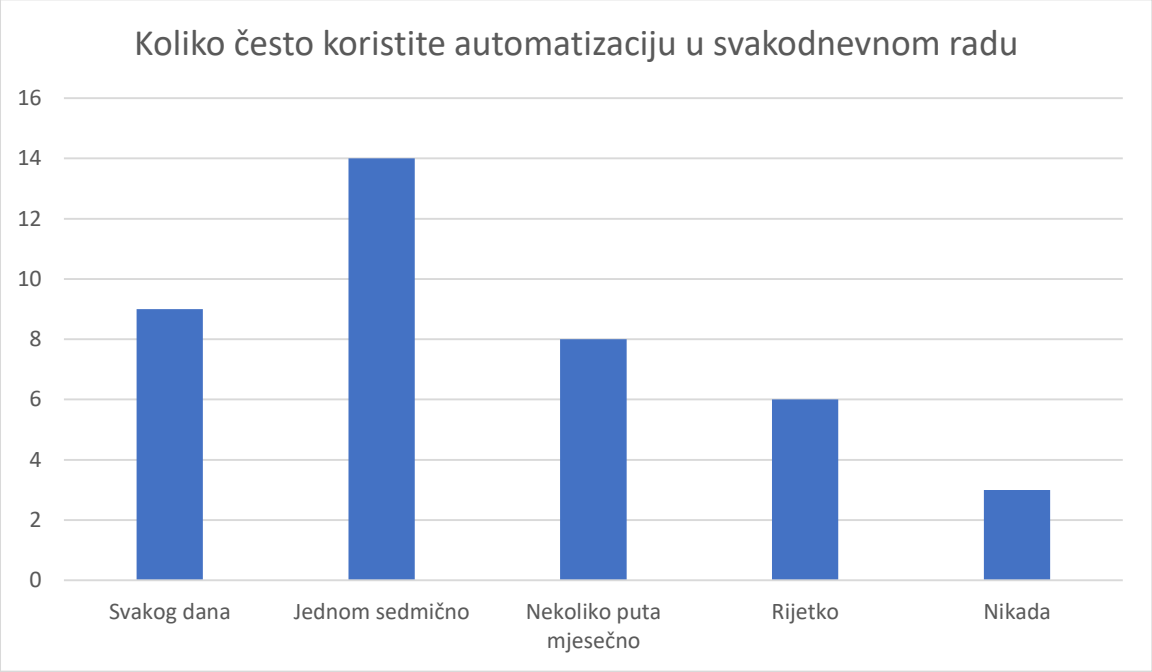
2. Da li ste koristili alate za automatizaciju testiranja, poput Seleniuma, u svom dosadašnjem radu?

Od 40 učesnika/ce koji su dali odgovor na ovo pitanje 36 je odgovorilo 'Da' dok je 'Ne' odgovorilo 4 učesnika/ca.



Grafik 2. Korišćenje Selenium alata u dosadašnjem radu

3. Koliko često koristite automatizaciju testiranja u svakodnevnom radu?



Grafik 3. Koliko je česta upotreba automatizacije u svakodnevnom radu

Analizom odgovora na pitanje o korištenju automatizacije testiranja u svakodnevnom poslu, primjećujemo raznolikost u odgovorima ispitanika. Većina ispitanika (14) navodi da jednom sedmično koristi automatizovano testiranje, što ukazuje na široku integraciju automatizovanih procesa u njihov radni tok. Zatim, 9 ispitanika izjavljuje da koristi automatizaciju testiranja svakog dana, dok 8 ispitanika navodi da je to nekoliko puta mjesečno. Manji broj ispitanika (6) izjavljuje da automatizaciju koristi rijetko, dok je mali broj (3) rekao da je nikada ne koristi. Ovi podaci pokazuju raznolikost u učestalosti korištenja automatizacije među QA testerima, sa većinom koji redovno primjenjuju ovaj pristup kako bi poboljšali efikasnost i preciznost u procesu testiranja softvera.

4. Koji su, prema vašem mišljenju, glavni benefiti korištenja automatizacije testiranja u razvoju softvera?

Četrdeset učesnika/ca ankete je dalo odgovor na ovo pitanje u vidu tekstualnog obrazloženja a primjeri koji se najčešće prožimaju kroz odgovore jesu:

- Brzina izvođenja testova, povećana pokrivenost testova.
- Povećana pouzdanost, reproduktivnost testova, poboljšana produktivnost razvojnog tima.
- Skabilnost testova, minimizacija ljudske greške, brzina regresije.
- Ubrzava procese razvoja softera, znatno smanjuje vrijeme testiranja i što je najvažnije poboljšava kvalitet samog proizvoda.
- Olakšava i ubrzava testiranje koje bi inače moralo da se obavlja manuelno.

- Povećana efikasnost, bolja pokrivenost testova, brže dovođenje proizvoda na tržište.
- Ukoliko je potrebno testiranje cijelog sistema, olakšaće posao.
- Veliko olakšanje prilikom regresionog testiranja ili retest-a cijelog sistema.
- Ušteda vremena prilikom testiranja novih funkcionalnosti na sistemu.

5. Koje biste izazove ili prepreke naveli kao najčešće prilikom implementacije automatizacije testiranja?

Analizom odgovora na pitanje o izazovima i preprekama prilikom implementacije automatizacije testiranja, identifikovani su neki uobičajeni faktori koji se najčešće ističu.

- Setovanje frameworka.
- Integracija sa postojećim procesima i alatima, održavanje testova.
- Vrijeme potrebno za inicijalnu postavku frameworka, odabir frameworka za automatizaciju i pronalaženje kadra za automatizaciju.
- Izazov - selektovanje djelova aplikacije koje treba automatizovati, a što se tiče prepreka vrijeme koje je potrebno da se ispišu svi testovi je obično ograničavajući faktor.
- Programerske vještine.
- Složenost održavanja skripti, zavisnost od UI elemenata, upravljanje podacima za testiranje.

Obzirom na to da postoji niz zajedničkih faktora koji se ističu, ovi odgovori naglašavaju potrebu za adekvatnim planiranjem, resursima i podrškom kako bi se prevazišli izazovi u implementaciji automatizacije testiranja.

6. Koliko smatrate da automatizacija testiranja doprinosi kvalitetu softvera?

Na osnovu rezultata ankete o doprinosu automatizacije testiranja kvalitetu softvera, uočava se visok stepen saglasnosti među ispitanicima. Većina ispitanika, tačnije 21 od 40, ocenjuje da automatizacija testiranja izuzetno doprinosi kvalitetu softvera, dok je 19 ispitanika dalo ocenu 4, što ukazuje na visok nivo povjerenja u doprinos ovog procesa. Ovi podaci pokazuju prihvatanje i pozitivan stav prema korišćenju automatizacije testiranja među QA testerima.

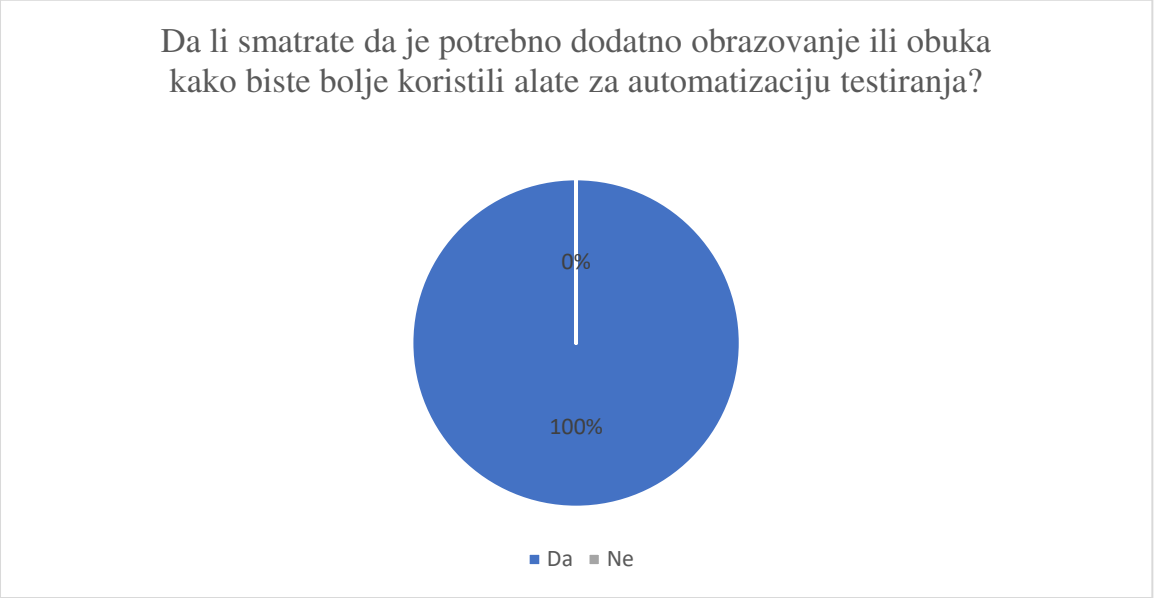
Ovakva visoka ocjena doprinosa automatizacije pokazuje da je ova praksa postala neizostavan dio procesa razvoja softvera, pa se smatra ključnim faktorom za postizanje visokog kvaliteta softverskih proizvoda.

Grafik 4. Prikaz odgovara na pitanje 'Koliko automatizicija doprinosi kvalitetu softvera'

7. Da li smatrate da je potrebno dodatno obrazovanje ili obuka kako biste bolje koristili alate za automatizaciju testiranja?

Na osnovu rezultata ankete, koji pokazuju da je 100% ispitanika odgovorilo potvrdno na pitanje o potrebi dodatnog obrazovanja ili obuke za bolje korišćenje alata za automatizaciju testiranja, jasno je da postoji široko rasprostranjeno mišljenje u neophodnost kontinuiranog usavršavanja u ovoj oblasti. Ovakav pristup ukazuje na prepoznavanje kompleksnosti alata za automatizaciju i svijest o važnosti nadogradnje vještina kako bi se iskoristile sve mogućnosti koje ovi alati pružaju.

S obzirom na dinamičnu prirodu tehnološkog razvoja i brze promene u oblasti softverskog inženjerstva, ova podrška dodatnom obrazovanju ukazuje na želju i spremnost QA testera da prate najnovije trendove i tehnologije kako bi unapredili svoje vještine i znanje u radu. Ovi rezultati potvrđuju važnost ulaganja u profesionalni razvoj i obrazovanje.

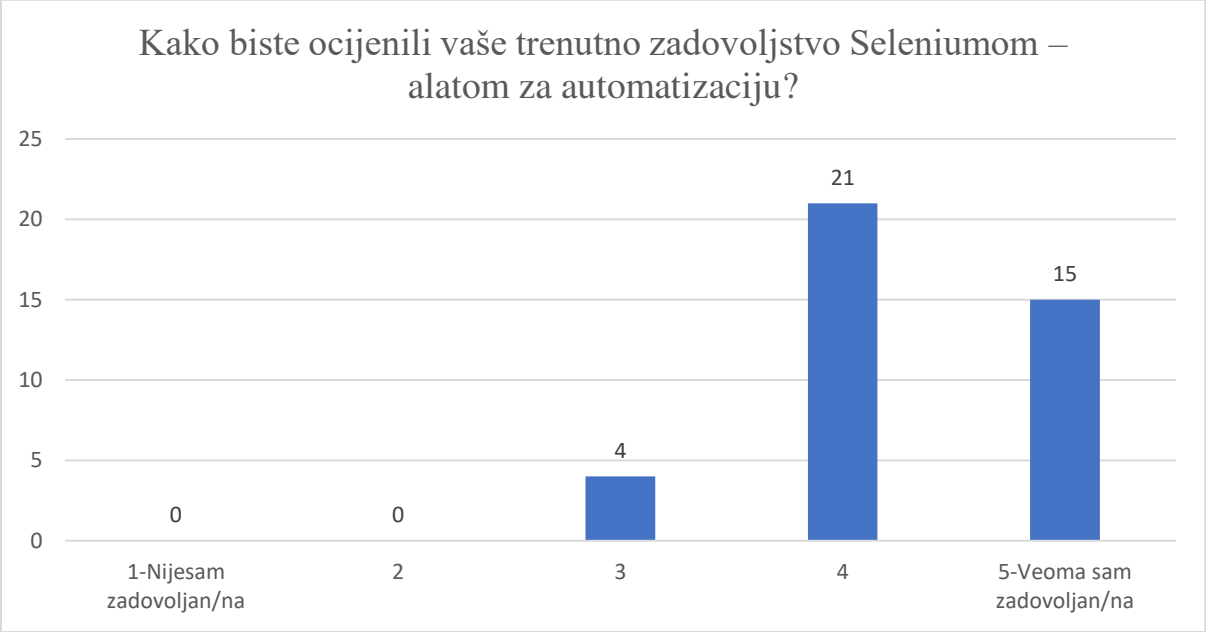


Grafik 5. Odgovori ispitanika na pitanje 'Da li smatrate da je potrebno dodatno obrazovanje ili obuka kako biste bolje koristili alate za automatizaciju testiranja'.

8. Kako biste ocijenili vaše trenutno zadovoljstvo Seleniumom – alatom za automatizaciju?

Na pitanja o trenutnom zadovoljstvu Seleniumom – kao alatom za automatizaciju testiranja, primjetna je visoka ocjena zadovoljstva među ispitanicima. Većina ispitanika, tačnije 15 od 40, dala je najvišu ocjenu 5, što ukazuje na veoma visok nivo zadovoljstva. Dodatnih 21 ispitanik ocijenilo je alat ocjenom 4, što sugeriše na opšte zadovoljstvo, dok je manji broj ispitanika, tačnije 4, dalo ocjenu 3.

Ovi rezultati govore da većina QA testera ima pozitivno mišljenje o alatima za automatizaciju testiranja koje trenutno koriste, sa naglaskom na Selenium, pri čemu većina ocenjuje zadovoljstvo kao veoma visoko.



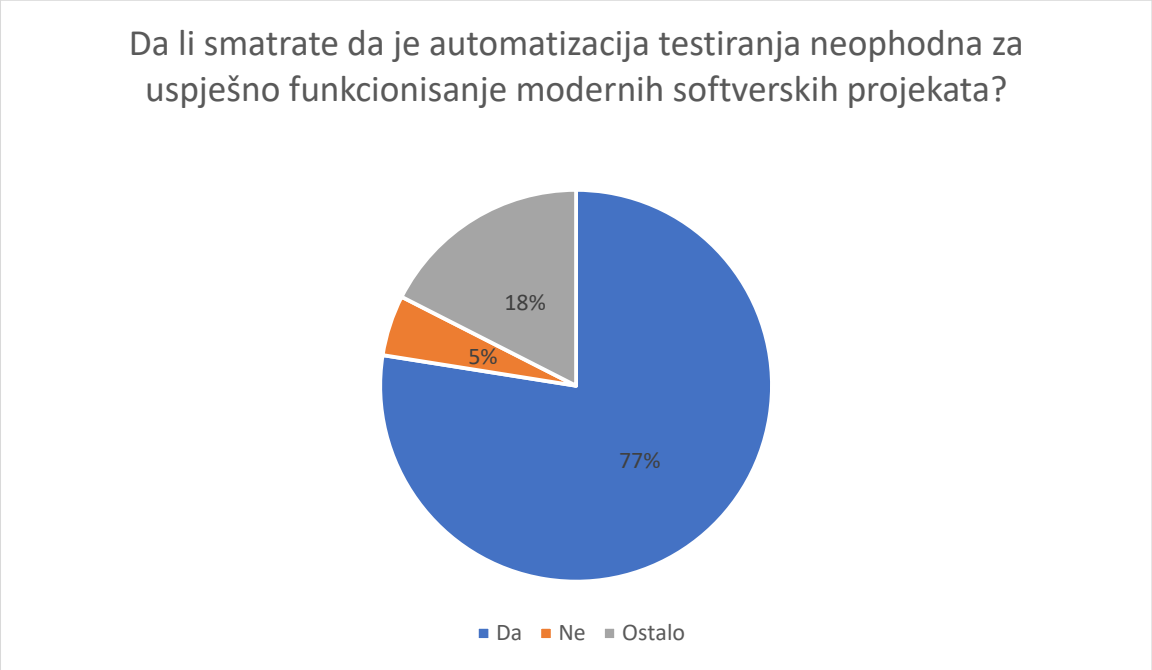
Grafik 6. Odgovori ispitanika na pitanja 'Kako biste ocijenili vaše trenutno zadovoljstvo Seleniumom – alatom za automatizaciju'

9. Da li smatrate da je automatizacija testiranja neophodna za uspješno funkcionisanje modernih softverskih projekata?

Na osnovu analize odgovora ispitanika na pitanje o neophodnosti automatizacije testiranja za uspešno funkcionisanje modernih softverskih projekata, ističe se značajan stepen saglasnosti među većinom učesnika, tj.QA testera. Čak 31 od 40 ispitanika je sklono mišljenju da je automatizacija testiranja imperativna za efikasno vođenje savremenih softverskih sistema.

Ovo potvrđuje široko prihvatanje automatskih procesa u testiranju kao ključnog faktora za postizanje visokog nivoa kvaliteta. Međutim, dvije osobe su izrazile suprotno mišljenje, tvrdeći da automatizacija nije obavezna, dok je sedam ispitanika iznijelo odgovore u kategoriji "Ostalo".

Pod ovom kategorijom ispitanici su isticali u svojim odgovorima da automatizacija može biti korisna, ali da nije nužno obavezna, ili da može biti implementirana u određenom kontekstu. Takođe bilo je usaglašenih mišljenja da automatizacija može biti korisna za veće sisteme i projekte, kao i da je ogroman plus ako se koristi u kombinaciji sa manuelnim testiranjem.



Grafik 7. Procentualni prikaz odgovora ispitanika na pitanje 'Da li smatrate da je automatizacija testiranja neophodna za uspješno funkcionisanje modernih softverskih projekata.

10. Koje biste dodatne funkcionalnosti ili mogućnosti voljeli vidjeti kod alata za automatizaciju testiranja, posebno kad je u pitanju Selenium?

Na pitanje o dodatnim funkcionalnostima ili mogućnostima kod Seleniuma, ispitanici su dali odgovor na ovo pitanje u vidu tekstualnog obrazloženja a odgovori koji se najčešće prožimaju jesu sljedeći:

- Voljeo bih da vidim funkciju koja sama izvlači UI elemente, kako bi nam olakšao taj dio prilikom automatizacije(npr. Selectors hub plugin koji ce sam izvući sve elemente za input polja, checkboxove, radio buttons itd.)
- Detaljniji report prilikom izvršavanja testova, step-by-step screen-shotovi po defaultu, paralelizacija na posebnim thread-ovima koji ne bi opteretili sistem sa previse zahtjeva.
- Bilo bi dobro da se uključi vještačka inteligencija i neka vrsta virtuelnog asistenta.
- Pametnije otkrivanje promjena u UI elementima, poboljšana analitika i izvještavanje.

Na osnovu prikupljenih odgovora, zaključak je da postoji izražena potreba za unapređenjem određenih funkcionalnosti Selenium alata, kao što su automatsko izvlačenje UI elemenata, detaljniji izveštaji prilikom testiranja, integracija veštačke inteligencije, pametnije otkrivanje promjena u UI elementima i slično. Ovi zahtjevi ukazuju na važnost kontinuiranog razvoja alata kako bi se efikasnije odgovorilo na potrebe modernih softverskih projekata.

ZAKLJUČAK

Testiranje softvera je ključna aktivnost za postizanje visokog kvaliteta softverskih rješenja. U ovom radu je demonstrirano kako uključivanje testiranja od samog početka razvoja softvera može doprinijeti boljim performansama i stabilnijim softverskim okruženjem. Kroz implementaciju Java test paketa sa Selenium Frameworkom, pokazano je da automatizacija testiranja značajno doprinosi preciznosti i efikasnosti, posebno u regresivnom i funkcionalnom testiranju

Jedan od glavnih doprinosa ovog rada je detaljna analiza i implementacija automatizovanog testiranja pomoću Selenium alata. Eksperimentalni dio je pokazao da automatizacija, iako inicijalno zahtjevna u smislu vremena i resursa, pruža dugoročne rezultate u pogledu smanjenja troškova, boljeg upravljanja vremenom i poboljšanja kvaliteta softvera. Rezultati su pokazali da automatizovano testiranje nadmašuje manuelno testiranje u pogledu tačnosti i pokrivenosti testnih slučajeva.

Za buduća istraživanja, preporučuje se dalji razvoj i unapređenje alata za automatizaciju, kao i istraživanje novih metodologija koje mogu dodatno optimizovati proces testiranja. Takođe, korisno bi bilo istražiti integraciju automatizovanog testiranja sa drugim aspektima razvoja softvera, kao što su kontinuirana integracija i kontinuirana isporuka (CI/CD), kako bi se postigla još veća efikasnost u razvojnim procesima. Iako je automatizacija testiranja inicijalno zahtjevna, njene dugoročne prednosti čine je neophodnim dijelom svakog ozbiljnog razvojnog procesa. Ovaj rad doprinosi razumijevanju i praktičnoj primjeni automatizacije testiranja, otvarajući put za dalja unaprijeđenja i istraživanja u ovoj oblasti.

LITERATURA

[1] O. Howard, 2018. "*Challenges and Opportunities in Test Automation*," A Survey

[2] E. P. Enoiu, D. Sundmark, A. Causevic, and P. Petterson, 2016. "*A Comparative Study of Manual and Automated Testing for Industrial Control Software*"

[3] Lj. Lazić and N. Mastorakis, "OptimalSQM: Integrated and Optimized Software Quality Management," *WSEAS Transactions on Information Science and Applications*, vol. 6, no. 10, pp. 1636-1664, 2009.

[4] J. Santos, "E-service quality: a model of virtual service quality dimensions," *Managing Service Quality*, vol. 13, no. 3, pp. 233-246, 2003.

[5] U. Godwin, K. Bagchi, and P. Kirs, "Assessing Web Service Quality Dimensions: The E-SERVPERF Approach," *Issues in Information Systems*, vol. IX, no. 2, 2008.

[6] B. Beizer, Software Testing Techniques, *2nd ed. International Thomson Computer Press*, 1990

[7] R. Lima, I. P. de Viana do Castelo, A. M. Rosado da Cruz, and J. Ribeiro, "Artificial Intelligence Applied to Software Testing: *A Literature Review*," June 2020.

[8] J. Kahles, A. Jung, and T. H. June 2019. "Automating Root Cause Analysis via Machine Learning in Agile Software Testing Environments,".

[9] P. Sudha, 2018. "Measuring Effectiveness of Software Quality by Comparing Manual Testing and Selenium," *International Journal of Scientific Research and Management Studies.*

[10] S. Besson, 2005. "A Strategy for Risk-Based Testing," Real-world Expertise for Software and IT Professionals.

[11] Gg P. C. Jorgensen, 2013. Software Testing: A Craftsman's Approach, *4th ed. AUERBACH*.

[12] G. J. Myers, C. Sandler, and T. Badgett, 2011. The Art of Software Testing, 3rd ed. Wiley Publishing.

[13] N. Garg, 2014. Test Automation using Selenium WebDriver with Java: *Step by Step Guide, 1st ed. AdactIn Group Pty Ltd.*

[14] M. Ivan, J. G. Andree, and V. H. 2010. "Collaborative Software Engineering: *Challenges and Prospects," Springer Link*, pp. 389–403.

[15] A. R. Adam, 2018. Fundamentals of Software Testing, *Springer Link,* pp. 1-52.

[16] Lemke, G. 2018. *The Software Development Life Cycle and Its Application*. Eastern Michigan University DigitalCommons@EMU Senior Honors Theses & Projects. Dostupno na: https://commons.emich.edu/honors/193 [pristupljeno: oktobar, 2023.]

[17] Rajani Devi, T. 2012. "Importance of Testing in Software Development Life Cycle." *International Journal of Scientific & Engineering Research*, 3(5), ISSN 2229-5518.

[18] Risener, K. 2022. "A Study of Software Development Methodologies." *Computer Science and Computer Engineering Undergraduate Honors Theses*. Dostupno na: https://scholarworks.uark.edu/csceuht/103 Posjeta: 12.02.2024.

[19] Tuteja, M. i Dubey, G. 2012. "A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models." *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, issue 3, ISSN: 2231-2307.

[20] Kumari, B., Chauhan, N., & Vedpal. (2018) "A COMPARISON BETWEEN MANUAL TESTING AND AUTOMATED TESTING." *JETIR*, December 2018, Volume 5, Issue 12. Dostupno na: http://www.jetir.org [pristupljeno: novembar 2023.]

[21] Singh, R. 2023. *A Definitive Guide to Mastering Selenium WebDriver Automation Effectively*. June 27, 2023.

[22] "Chrome extension" *(2024) Selenium Full Material Updated Greens. Dostupno na: https://greenstechnologys.com/Selenium%20Full%20Material%20Updated%20Greens.pdf* [pristupljeno: novembar 2023.]

[23] Singh, N. 2016. "*Selenium Ide Basic*." Uploaded by Narottam Singh on Jun 10, 2016.

[24] Vaidya, N. (2023) "*All You Need to Know About Selenium WebDriver Architecture*." Last updated on Jan 12, 2023. Dostupno na: https://www.edureka.co/blog/selenium-webdriver-architecture/ [pristupljeno: decembar 2023.]

[25] Selenium 2023. *"Getting Started with Selenium Grid."* Last modified May 25, 2023. Dostupno na: https://www.selenium.dev/documentation/grid/getting_started/ [pristupljeno: decembar 2023.]

[26] *itlearn360* 2020. *New Selenium Session Java*. Dostupno na: https://www.itlearn360.com/wp-content/uploads/2020/11/New_Selenium-Session-Java-.pdf [pristupljeno: decembar 2023.]

[27] *Apache Software Foundation* (2024) "Apache Maven Surefire Plugin." Dostupno na: https://maven.apache.org/surefire/maven-surefire-plugin/ [pristupljeno: oktobar 2023.]

[28] Marbaise, K. H. 2019. *Maven Unit and Integration Test Guide*. Dostupno na: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://khmarbaise.github.io/maui/maui.pdf

[29] Forrester 2023. *The Forrester Wave™ Software Composition Analysis,* Q2 2023.

[30] Smartbear 2018. *What is Automated Testing?* Smartbear. Dostupno na: https://smartbear.com/learn/automatedtesting/what-is-automated-testing/ [pristupljeno: novembar 2023.]

[31] Selenium 2018. *What is Selenium?* Selenium Dostupno na: https://www.seleniumhq.org/. [pristupljeno: novembar 2023.]

[32] Albarka Umar, M. (2020) "*Comprehensive Study of Software Testing: Categories, Levels, Techniques, and Types*." Jun 2020.

[33] Kaner, C., Falk, J., & Nguyen, H. Q. 1999. *Testing Computer Software* (2nd ed.). Wiley.

[34] Beizer, B. 1990. *Software Testing Techniques* (2nd ed.). Van Nostrand Reinhold.

[35] Rajkumar, S. M. 2018. *Automated Software Testing: Introduction, Management, and Performance*. CRC Press.