Automatizirano testiranje softvera

Krajačić, Rudolf

Undergraduate thesis / Završni rad

2023

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:659755

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Rudolf Krajačić

AUTOMATIZIRANO TESTIRANJE SOFTVERA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Rudolf Krajačić

Matični broj: 0016138891

Studij: Poslovni sustavi

AUTOMATIZIRANO TESTIRANJE SOFTVERA

ZAVRŠNI RAD

Mentor/Mentorica:

Dr. sc. Marko Mijač

Varaždin, rujan 2023.

Rudolf Krajačić

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Ovaj završni rad istražuje automatizirano testiranje softvera i njegovu potrebu u provjeri kvalitete sustava i aplikacija. Na početku rada objašnjava se princip testiranja, kako manualnog tako i automatiziranog, zatim se ističu prednost i važnosti koje automatizirano testiranje ima za razliku od manualnog testiranja. U radu se spominju alati automatiziranog testiranja, na koji način i zašto su odabrani neki od navedenih za praktični dio ovog završnog rada, i kroz stvarni primjer programa način na koji su ti alati korišteni. Ovu temu izabrao sam iz razloga što me zanima automatizirano testiranje i u vrijeme pisanja završnog rada odrađivao sam studentski posao na kojem jedan dio zaduženja uključuje ovakvu vrstu testiranja u programu IntelliJ uz pomoć okvira Selenium. Automatizirano testiranje softvera provoditi će se na programu koji je prethodno izrađen i testiranja su provedena na tri razine koje čine jedinično testiranje, integracijsko testiranje i testiranje „s kraja na kraj“. Teorijski dio rada napisan je prema mnogim izvorima literature, ali praktični dijelovi pisani su prema vlastitom iskustvu koje je stečeno kroz praksu.

Ključne riječi: automatizirano testiranje, integracijsko testiranje, jedinično testiranje, testiranje od kraja do kraja, IntelliJ, Selenium, Mockito, JUnit

Sadržaj

1. Uvod

Softver je u današnje vrijeme postao nezaobilazan dio ljudske svakodnevice, koliko s obzirom na digitalizaciju i modernizaciju, toliko i zbog globalne pandemije koja je onemogućila fizičke kontakte s drugima tako da je softver postao važan alat za komunikaciju, što pokazuje i podatak da je 45% ljudi diljem svijeta potrošilo više vremena na društvenim mrežama nego prije [1]. Sve više poduzeća fokusira se na aplikacijske softvere s ciljem pružanja usluge i rješenja za različite poslovne izazove, tako postaje važno omogućiti da su softverski proizvodi funkcionalni, kvalitetni i pouzdani. Upravo to je i razlog da automatizirano testiranje softvera postaje bitnija komponenta pri razvijanju i održavanju softvera. U završnom radu primijenjene su tri razine testiranja – jedinično, integracijsko i testiranje s kraja na kraj uz korištenje prikladnih alata kao što su JUnit, Spring Test, Mockito i Selenium te je krajnji cilj implementirati efikasno i dobro obuhvaćeni okvir (eng. *Framework*) za automatizirano testiranje.

Jedinično testiranje (eng. *Unit testing*) fokusira se na ispitivanje funkcionalnosti i ispravnosti malih dijelova programa – jedinica. Za ovaj dio testiranja koristit će se Mockito, popularan okvir za razvijanje i izvršavanje testnih jedinica u Java programskom jeziku.

Integracijsko testiranje (eng. *Integration testing*) je sljedeća razina čiji je cilj testiranje integracije između više različitih komponenti softvera. Ovdje će se koristiti JUnit, programski okvir za testiranje Java aplikacija i uz to pruža podršku za integracijska testiranja. JUnit se koristi s mnogim bibliotekama i alatima za testiranje Spring Boot aplikacijskih programa. Uz JUnit koristit će se i Spring Test alat koji će služiti za izvršavanje testova integracije koji sadrže bazu podataka i ostale dijelove aplikacije.

U konačnici, testiranje s kraj na kraj (eng. *End-to-end testing – E2E*) podrazumijeva vrstu testiranja korisničkog toka u aplikaciji, mogućnost testiranja svih interakcija između aplikacijskih komponenti. Za izvršavanje E2E testova koristit će se okvir Selenium, popularan kod testiranja preglednika.

Pisanjem ovog završnog rada istražit ćemo potrebu i prednosti automatskih testova u pogledu na sve tri razine testiranja, primjenom alata JUnit, Mockito, Spring Boot i Selenium analizirati ćemo njihovu implementaciju pri razvoju testnih scenarija. Naravno postoje i mogući problemi s kojima bi se mogli susresti prilikom pisanja automatskih testova te ćemo ponuditi smjernice kako uspješno testirati softver.

2. Osnove testiranja

Početak razvoja počinje istraživanjem i razumijevanjem korisničkih zahtjeva i završava testiranjem implementiranog proizvoda te njegovom isporukom. Prvotni koraci su bitni za razvoj korisniku prihvatljivog proizvoda, krajnja faza bitna je za dobivanje povjerenja svih korisnika [2].

Prije nego što započnemo s automatskim testiranjem softvera, potrebno je utvrditi i objasniti što je uopće testiranje softvera. Prema International Business Machines Corporation-u (IBM) „testiranje softvera je proces procjene i potvrde da softverski proizvod ili aplikacija radi ono što je predviđeno. Prednosti testiranja uključuju sprječavanje grešaka, smanjenje troškova razvoja i poboljšanje performansi“ [3].

Desai i Srivastava [4] navode da postupak sprječavanja grešaka ima ključnu ulogu u sprječavanju programskih pogrešaka. Njegova uspješnost ovisi o temeljnom testiranju softvera, analizirajući nedostatke u svakoj fazi kako bi smo razvili tehnike, smjernice i metode koje će spriječiti njihovo ponavljanje u budućim fazama, uz to neprestano unaprjeđujući postupak prevencije oštećenja identificiranjem uzročnih čimbenika i izradom kratkoročnih te dugoročnih akcijskih planova. Kratkoročni planovi provode se odmah kako bi se brzo riješili problemi, dok dugoročni planovi uključuju promjene u procesu. Ove promjene procesa se koriste u budućim projektnim testiranjima.

Testiranje je bitno jer greška (eng. *Bug*) može dovesti do skupih, pa čak i opasnih problema, koji potencijalno mogu biti odgovorni za novčane, ali i ljudske gubitke kojih je u prošlosti bilo mnogo [5], [6]:

- Davne 1985. godine Therac-25, radijacijski uređaj zbog softverske greške je pacijentima davao i do 100 puta veće doze radijacije nego što je normalno, što je rezultiralo smrću 3 osobe i još 3 su ozlijeđene.
- U travnju 1999. godine, softverska greška na gornjem dijelu rakete Titan IV nije primijećena prije lansiranja, što je uzrokovalo štetu od 1.23 milijardi dolara, najskuplja nesreća u 50 godina lansiranja iz Cape Canaveral baze [42].
- 2015. godine londonski terminal Bloomberg srušio se zbog softverske greške koja je utjecala na preko 300.000 trgovaca na financijskom tržištu i tamošnje vlasti su bile prisiljene na odgodu prodaje državnog duga vrijednog tri milijarde funti.
- Tvrtka Nissan 2023. godine povukla je više od milijun automobila modela Note, Leaf, Kick i Serena zbog iznenadnog ubrzanja prilikom gašenja tempomata, ali i potencijalnog strujnog udara koji može ugasiti motor prilikom vožnje.

- Starbucks, popularna američka tvrtka koja prodaje napitke od kave, bila je prisiljena zatvoriti skoro 60% trgovina u Sjedinjenim Američkim Državama i Kanadi zbog kvara na njihovom elektronskom terminalu (eng. *Point Of Sale – POS*).

2.1. Principi testiranja

Spominjući navedene softverske greške, možemo zaključiti da je potreba za testiranjem iznimno bitna, a principi koji se koriste kod testiranja su sljedeći [7], [8]:

- Potpuno testiranje nije moguće – vrlo je teško testirati sve module i njihove značajke s svim vrstama kombinacija ulaznim podataka kroz testni proces, stoga se umjesto iscrpnog testiranja koje zahtjeva ogromne napore, testiranje nekoliko kombinacija provodi na temelju prioriteta. Razlog tome su rokovi koji ne dopuštaju takve tipove testiranja.
- Klasteriranje grešaka – ovaj princip govori da kroz testiranje možemo najviše grešaka otkriti unutar malog broja modula, razlozi za to su različiti, poput kompleksnosti modula, kompleksnosti programiranja i slično. Ovakve vrste aplikacije prate Paretov princip (eng. *Pareto Principle*) koji u svijetu testiranja glasi otprilike: 80% grešaka nalazi se u 20% modula aplikacije. Iako i ovaj pristup ima manu, a to je da se ponovnim izvođenjem istih testova u konačnici nećemo pronaći nove greške.
- Paradoks pesticida – definicija ovog principa objašnjava da izvođenjem istih testova iznova za redom tijekom određenog vremenskog perioda, testovi neće otkriti nikakve nove greške u programskom kodu. Kako bi spriječili ovu pojavu, testne slučajeve je potrebno kontrolirati, dodati nove i drugačije testne slučajeve da bi mogli pronaći nove druge greške.
- Testiranje prikazuje prisutnost nedostataka – testiranje ne govori o odsutnosti grešaka, testiranje govori da su greške prisutne. To govori da testiranje softvera smanjuje vjerojatnost neotkrivenih mana koje su prisutne u programu, ali nikad nije moguće sasvim sigurno reći da greške ne postoje. Čak i ako uspijemo postići da proizvod radi bez skoro i jedne greške, što ako program ne zadovoljava zahtjeve i potrebe koje su korisnici željeli? To nas dovodi do sljedećeg principa.
- Savršen-neupotrebljiv program – postoji mogućnost da je program gotovo savršen, ali nema koristi od njega. Takva situacija se javlja ako se sustav testira, ali za krive potrebe. Testiranje je proces koji zahtjeva osim nalaženja grešaka, uz to i provjeru zadovoljava li softver potrebe klijenta. S toga pronalaženje grešaka i ispravljanje grešaka ne pridonose cjelokupnom sustavu ako je sustav u konačnici neupotrebljiv i nezadovoljavan potrebe i zahtjeve.

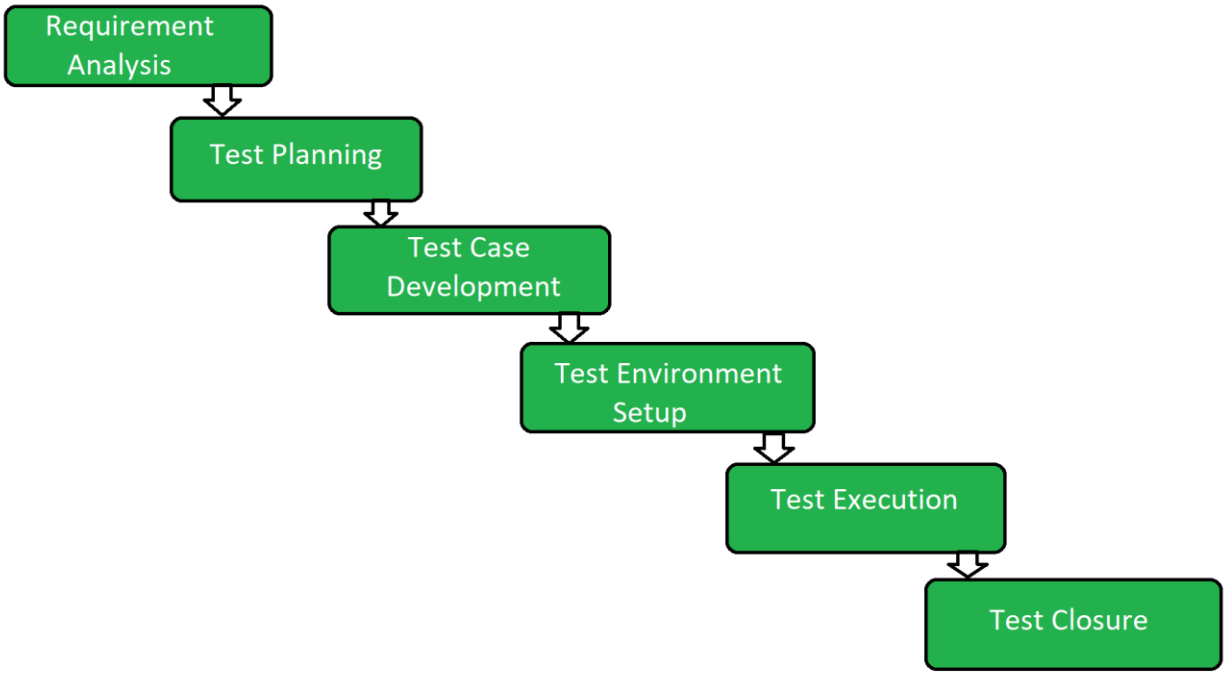
- Rano testiranje – proces testiranja morao bi započeti čim ranije u životnom ciklusu razvoj softvera (eng. *Software Development Life Cycle – SDLC*). Takvim pristupom se greške pronađu u ranim fazama, kad ih je puno jeftinije ispraviti. Preporuka za početak testiranja softvera je onog trenutka kad su zahtjevi procesa definirani.
- Testiranje ovisi o kontekstu – postoje različiti načini testiranja ovisno o situacijama u kojima se nalazimo i koje trebamo provesti kroz testni scenarij, stoga će se testiranje web stranica razlikovati u načinu testiranja operacijskog sustava satelita. Potrebni su drugačiji pristupi testiranju, drugačije metodologije, tehnike i vrste testiranja ovisno o kakvom je programu riječ.

2.2. Životni ciklus testiranja

Životni ciklus testiranja softvera (eng. *Software Testing Life Cycle - STLC*) je pristup testiranju programske aplikacije čiji je cilj da aplikacija zadovolji potrebe korisnika te da nema grešaka. To je proces koji prati određeni niz koraka ili faza, pri čemu svaka faza ima specifične ciljeve i zadatke. STLC se koristi da program bude na visokoj razini kvalitete, da ga čini pouzdanim te da zadovoljava potrebe korisnika [9].

STLC se sastoji od nekoliko faza, a to su:

1. Analiza zahtjeva
2. Planiranje testiranja
3. Razvoj testnih slučajeva
4. Priprema testne okoline
5. Izvođenje testova
6. Zatvaranje testova



Slika 1: Faze životnog ciklusa testiranja softvera (Izvor: [9])

2.2.1. Faza analize zahtjeva

Početak STLC-a počinje fazom analize zahtjeva, u kojoj se testeri analiziraju zahtjeve klijenata koji su unutar ciklusa razvoja softvera (eng. *Softvare Development Life Cyrcle – SDLC*), nakon pregleda zahtjeva, testni tim izrađuje testni plan kako bi se vidjelo da li program zadovoljava potrebe ili ne [10]. Aktivnosti koje se odvijaju tijekom ove faze su sljedeće [9]:

- Priprema liste zahtjeva i nejasnoća ako postoje prilikom razgovora sa voditeljem projekta, sistemskim analitičarima i klijentima.
- Potrebno je napraviti listu svih testova koji se provode poput funkcijskih testova, testova performanse i testova sigurnosti.
- Napraviti listu detalja testnih okruženja koja se sastoji od potrebnih alata za izvađanje testnih slučajeva.

2.2.2. Faza planiranja testiranja

Nakon što je završena faza analize zahtjeva, na red dolazi bitna faza, a to je planiranje testiranja. U njoj voditelj testera određuje izračunava koliko je truda i novaca potrebno za izradu testiranja. Detalji faze planiranje testiranja podrazumijeva stvari poput [9]:

- Pripreme testnih slučajeva i kontrola procedure.

- Podjele zadataka i ovlasti u timu.
- Nabrajanja mogućih rizika.
- Definiranja cilja i opseg programa.
- Definiranja metoda korištenih prilikom testiranja
- Pregleda procesa testiranja.
- Određivanja testne okoline.

2.2.3. Faza razvoja testnih slučajeva

Razvoj testnih slučajeva uključuje kreiranje testnih slučajeva te testnih skripti nakon što je testni plan spreman za korištenje. Testni podaci se u početku validiraju, da bi ih potom stvorili i pregledali i u konačnici se prepravljaju na osnovi preduvjeta. Tek tada testni tim počinje s procesom kreiranja testnih slučajeva za pojedine jedinice. Ova faza uključuje radnje poput [11]:

- Kreiranja testnih slučajeva ili automatskih skripti.
- Pregled i izrada bazičnih testnih skripti.
- Unos testnih podataka, ako je testna platforma dostupna.

2.2.4. Faza pripremanja testne okoline

Također bitan dio STLC-a, jer se ovdje odabiru uređaji i softver na kojima će se program testirati. Pripremanje testne okoline je zasebna aktivnost koja se može odvijati u isto vrijeme kao i faza razvoja testnih slučajeva. U ovaj dio procesa nije uključen tim testera, već postavljanje testne okoline odrađuju programeri ili klijenti [10]. Zadaci ove faze sadrže [11]:

- Pripremu testne okoline.
- Pripremu ulaznih parametara.
- Provođenje dimnih testova (eng. *Smoke test*).

2.2.5. Faza provođenja testova

Testiranje dolazi na red tek nakon uspješnog provođenja testnog plana, te sada testni tim može započeti sa razvojem i izvođenjem testnih slučajeva. Testeri detaljno opisuju testne slučajeve, a po potrebi i ulazne parametre. Napravljeni testni slučajevi se pregledavaju od strane voditelja tima kvalitete [10]. Zadaci ove faze sadrže [9]:

- Pripremu ulaznih parametara.
- Izvođenje testova.
- Praćenje grešaka.

- Analizu rezultata testova.
- Ponovno testiranje grešaka.
- Izradu testne dokumentacije.

Bitno je napomenuti da je provođenje testova ponavljajući proces, što znači da ga je potrebno ponavljati onoliko puta dok otkrivene greške nisu ispravljene i sve dok se program ne čini dobar za isporuku [9].

2.2.6. Faza zatvaranja testiranja

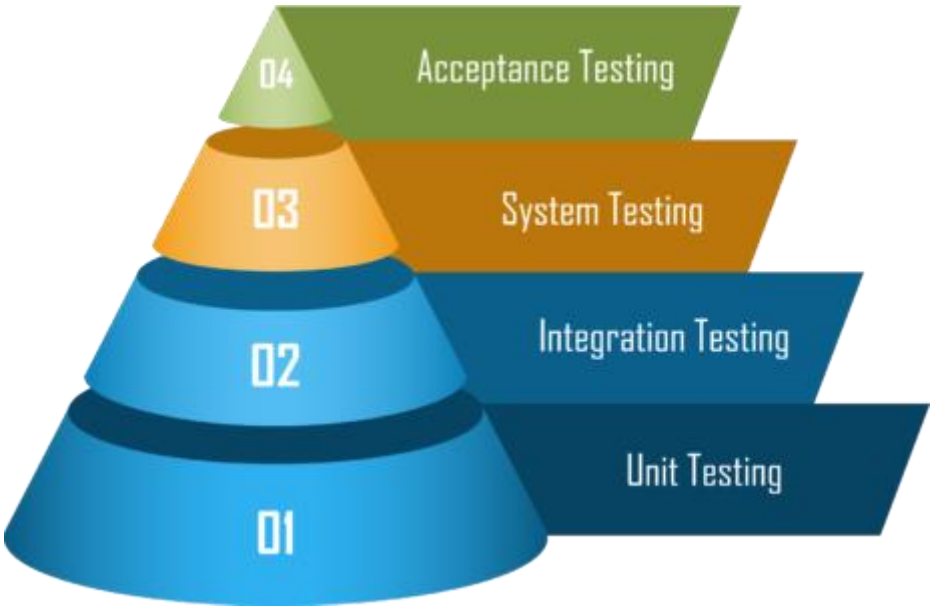
Ujedno i posljednja faza u procesu izvođenja testova koja uključuje završavanje izvještaja testiranja, prikupljanje rezultata testiranja i stvaranje testne dokumentacije. Sve navedene stavke se analiziraju i raspravljaju unutar sastanka članova tima s ciljem uzimanja pouka iz trenutnog projekta i pronalaska načina za bolju implementaciju u budućnosti. Ova faza također ima svoje završne aktivnosti poput [11]:

- Ekspertize ciklusa završavanja na temelju utrošenog vremena, uloženog novca, korištene tehnologije, kvalitete odrađenog posla.
- Dokumentiranja naučenog na projektu.
- Pripreme izvještaja o zatvaranju testiranja.
- Analize testnih rezultata za otkrivanje grešaka na projektu.

3. Razine testiranja

Prema Certified Tester Foundation Level Syllabusu, grupe testnih aktivnosti koje su napravljene i održavane zajedno nazivaju se razina testiranja. Svaka razina je primjer testnog procesa, izvedenog u ovisnosti o softveru koji se nalazi na datoj razini testiranja. Razine testiranja povezane su s drugim aktivnostima koje su dio životnog ciklusa razvoja softvera (eng. *Software Development Life Cycle – SDLC*). Postoje četiri razine testiranje softvera [12]:

1. Jedinično testiranje.
2. Integracijsko testiranje.
3. Sistemsko testiranje.
4. Testovi prihvaćanja.



Slika 2: Razina testiranja (Izvor : [13])

3.1. Jedinično testiranje

Jedinica (eng. Unit) je najmanji dio aplikacije ili sustava koji je moguće testirati, te ovakva vrsta testiranja pomaže testiranju svakog modula zasebno. Cilj je testiranje svakog dijela softvera na način da ga se razdvoji i tada se provjerava ispunjavaju li komponente svoje zadatke. Često se provodi zasebno u odnosu na ostali sustav, zavisno o SDLC-u, koji može zahtijevati potrebu za oponašateljima objekata (eng. *Mock object*), virtualizaciju servisa i različitim upravljačima. Značajke ovakvog testiranja uključuju [13]:

- Smanjenje rizika.
- Određivanje ponaša li se komponenta na način kako je dizajnirana.
- Pronalaženje grešaka u komponentama.
- Kontroliranje grešaka kako ne bi otišle na više razine.

Jedinične testove najčešće pišu programeri koji su pisali programski kod, a neki od alata koji se koriste za njihovo provođenje su JUnit, Mockito, NUnit, TestNG itd. [14].

Prednosti jediničnih testova prema Java Pointu su [15]:

- Ovakvo testiranje koristi modulski pristup pri čemu svaki dio koda može biti testiran bez čekanja na red da drugi testovi budu izvršeni.
- Programeri se fokusiraju na dobivenu funkcionalnost jedinice i kako ona radi te kako izgleda u jediničnim testovima kako bi mogli razumjeti jedinično sučelje.
- Unit testovi omogućavaju programerima refaktoriranje koda nakon određenog vremenskog perioda kako bi se uvjerili da modul još uvijek radi bez problema.

Jedinični testovi imaju i neke nedostatke [15]:

- Test ne može prepoznati integracijske te greške više razine jer se temelji na testiranju jedinica koda.
- Zbog fokusiranja na jedinicu, za testiranje cijelog softvera potrebno ga je koristiti s drugim vrstama testiranja.
- Nije moguće evaluirati sve putanje izvršavanja, tako da nije moguće uhvatiti svaku grešku s jediničnim testiranjem.

3.2. Integracijsko testiranje

Integracijsko testiranje je druga razina testiranja softvera. U ovoj vrsti testiranja jedinice i pojedinačne komponente se spoje i testiraju u grupi. Svrha integracijskog testiranja je provjeriti točnost komunikacije između svih modula. Postoji više pristupa integracijskog testiranja [16]–[18]:

- Inkrementalno integracijsko testiranje – testiranje se provodi integracijom dva ili više modula koji su logički povezani jedan na drugog, koje se zatim testira kako bi vidjeli pravilan način rada programa. Postoje čak tri različita pristupa inkrementalnom testiranju, a to su:
  - Integracijsko testiranje od vrha prema dnu (eng. *Top-down*) - metoda u kojoj integracija kreće odozgo prema dolje, prateći tijek aplikacije. Više razine modula testirane su prije donjeg dijela modula. Prednost ovog pristupa je mogućnost

integracije je jednostavnost primjene te je ovisnost o drugim dijelovima aplikacije vrlo mala.

  - Integracijsko testiranje od dna prema vrhu (eng. *Bottom-up*) – proces u kojem se testiranje i integriranje komponenti radi od najnižeg modula u aplikaciji prema najvišem modulu. Izvođenje traje sve dok i posljednji modul ne prođe kroz testiranje. Prednosti su mu visok postotak uspjeha, brzina i učinkovitost.

- Integracija velikog praska (eng. *Big bang*) – ovaj primjer integracijskog testiranja funkcionira na način da sve komponente integrira odjednom, koje zatim testira kao cjelinu. U slučaju da sve komponente koje testiramo nisu dovršene test se neće izvršiti. Prednosti su mu da je prikladan za manje sustave, ali mu je glavni nedostatak to što će tijekom provođenja testiranja biti gubljenja vremena, s obzirom da je neophodno čekati da se svi moduli implementiraju prije početka testiranja.

3.3. Sistemsko testiranje

Treće po redu testiranje je sistemsko testiranje, koje se provodi na cijelom sustavu i provjerava radi li, prema unaprijed definiranim zahtjevima, sustav kao cjelina. Testeri provode sistemska testiranja kako bi provjerili funkcijske i ne funkcijske zahtjeve sustava nakon što su zasebni moduli i komponente spojeni zajedno. Sistemsko testiranje je kategorija testiranja crne kutije (eng. *Black Box testing*), tako da se testiranja provode samo na vanjskim funkcionalnim dijelovima. U prvi fokus se stavlja testiranje cijelog sustava s ciljem provjere svih komponenti sustava, rade li ispravno [19].

Primjena ovog testiranja je poslije integracijskog testiranja, ali prije testiranja prihvatljivosti. Sistemsko testiranje provode testni timovi softvera na dnevnoj bazi kako bi osigurali da sustav radi nesmetano u bitnim situacijama izrade [18].

Budući da postoji više od 50 vrsti različitih sistemskih testiranja, moguće je pronaći vrstu po potrebi korisnika, makar se u praksi koristi samo nekolicina unutar testnih timova [19].

3.4. Testovi prihvaćanja

Zadnja, četvrta razina testiranja softvera je testiranje prihvaćanja, čiji je zadatak procijeniti zadovoljava li program korisničke zahtjeve prema prethodnom dogovoru. Važno je napomenuti da ovu vrstu testiranja odrađuju klijenti ili krajnji korisnici.

Glavni cilj je potvrditi tok aplikacije s kraja na kraj. Test prihvaćanja se ne fokusira na kozmetičke greške, pravopisne greške ili sustavno testiranje, ono se provodi na posve drugačijem testnom okruženju u čijoj su bazi podaci koji bi bili u produkcijskoj verziji aplikacije. Na neki način ovo je testiranje crne kutije, gdje je uključeno nekoliko krajnjih korisnika [20].

Stoga je vrlo važno provesti testove prihvaćanja, a ako ih testni tim izostavi, postoji velika šansa da se krajnji proizvod ne slaže sa zahtjevima korisnika. Nakon što je sistemsko testiranje provedeno, testeri provode testove prihvaćanja iz sljedećih razloga [20]:

- Da bi osigurali rad softvera na način koji je dogovoren sa klijentom.
- Potvrda da softver odgovara trenutnom tržištu i da se može mjeriti sa konkurencijom.
- Dobivanje pouzdanja u proizvod koji se sprema za plasiranje na tržište.

Prednosti koje ovi testovi donose [21]:

- Iz razloga što su korisnici u ovom slučaju testeri, služi kao pomoć projektnom timu kako bi znali buduće zahtjeve od njih samih.
- Testovi su automatizirani.
- Korisniku je lakše objasniti kakve promjene i napretke želi u softveru.
- Cijeli proizvod se može testirati.
- Uključivanjem korisnika u testove dobivamo njihovo zadovoljstvo i pouzdanje u proizvod.

Ali pak s druge strane imamo i neke nedostatke testiranja prihvaćanja [21]:

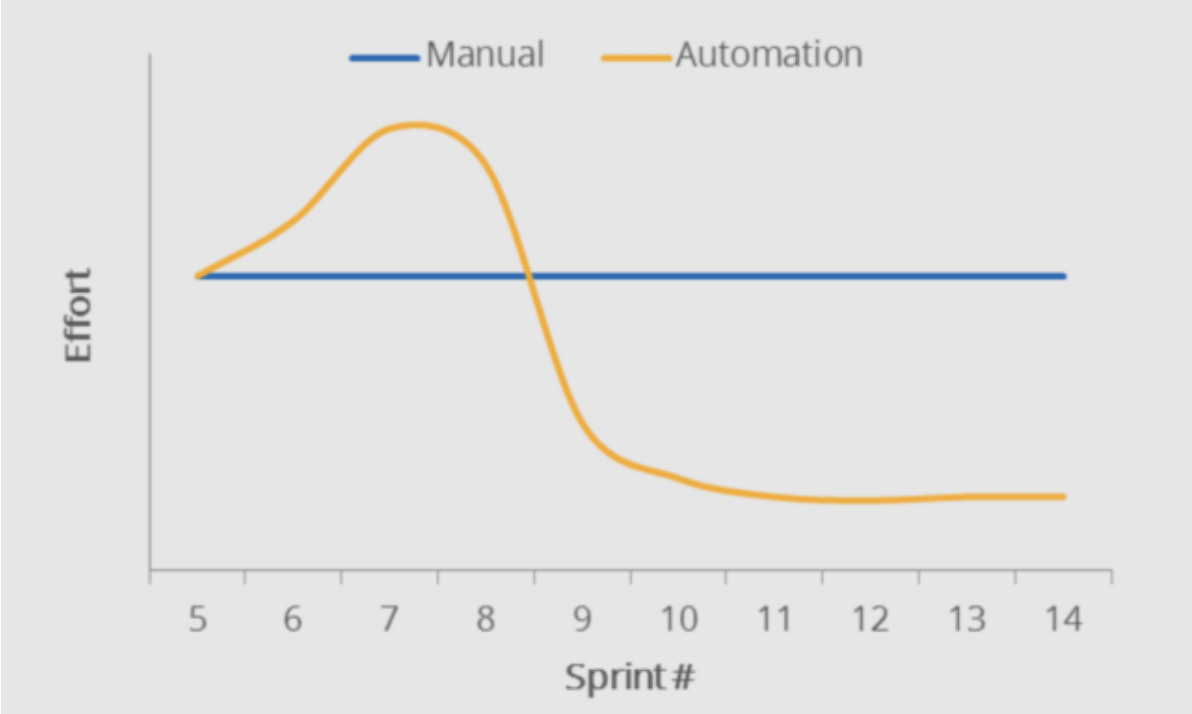
- Korisnici moraju imati barem osnovno znanje o proizvodu koji testiraju.
- Programeri ne sudjeluju u ovoj vrsti testiranja.
- Povratne informacije na testiranje mogu potrajati, zbog velikog broja ljudi koji testiraju proizvod, isto tako mišljenja korisnika su drugačija jedna od drugih.

4. Automatizirano testiranje

Automatizirano testiranje je vrsta testiranja u kojoj alat ili okvir testiranja izvršava zadane testne skripte za razliku od ručnog testiranja koje provodi tester korak po korak [22].

Za uspješnu izradu softvera potrebno je ponavljati određene testove i po nekoliko puta, a upotrebom automatskih testova postoji mogućnost snimanja testova te pregledavanje istih nakon završetka izvođenja. Jednom kada su automatizirane testne skripte, više nema potrebe da tester posreduje, alat sve odrađuje sam [23].

Automatizacija drastično smanjuje vrijeme potrebno za testiranjem, što navodi sve veći broj poduzeća da automatiziraju svoje testove. Usporedbe radi, za pokretanje procesa automatskih testova u odnosu na ručne testove potrebno je više napora, ali poslije toga automatizacija ušteđuje puno više vremena i truda za provedbu [22].

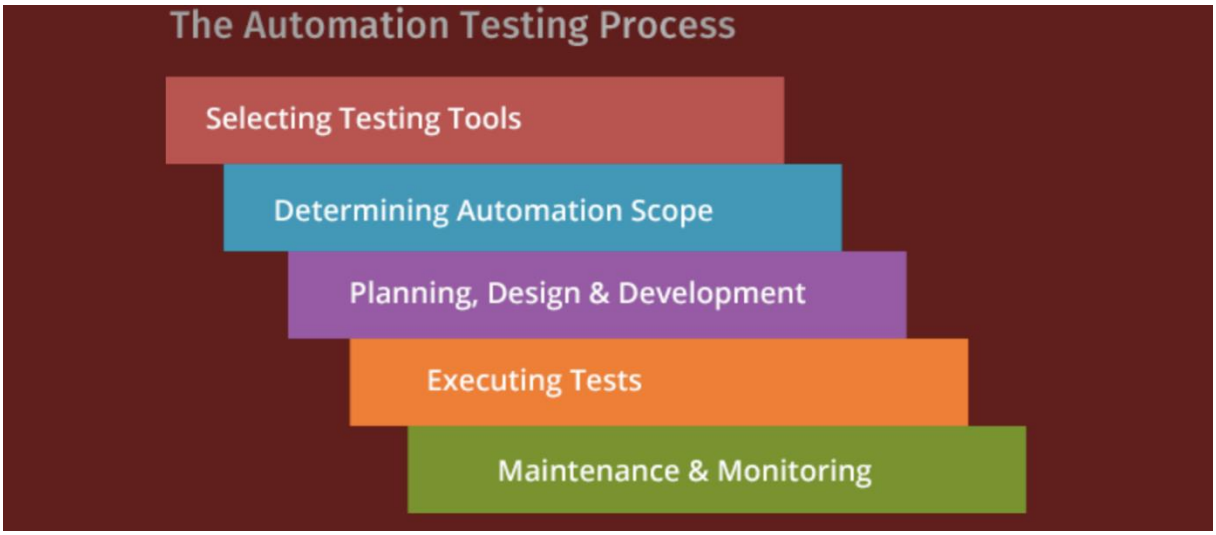


Slika 3: Uloženi trud u određenom vremenskom periodu (Izvor: [24])

Automatizirano testiranje ima nekoliko faza od kojih se sastoji [23]:

1. Odabir alata – prva, bitna faza kod automatiziranog testiranja je odabir alata, ovisno o tehnologijama koje su korištene za izradu softvera.

2. Definirati opseg automatizacije – odnosi se na područje aplikacije koje ulazi u testiranja, a način određivanja područja moguće je definirati preko značajki koje su bitne za odrađivanje poslova, scenariji s velikom količinom podataka, mogućnost korištenja testnih slučajeva u drugim preglednicima itd.
3. Planiranje, dizajniranje i izrada – u ovoj fazi izrađuje se plan i strategija automatiziranog testiranja, tu se nalaze detalji poput alata koji je odabran za testiranje, pripremanje testnog okruženja, zakazivanje datuma izvršenja testova itd.
4. Izvođenje testova – testne skripte koje se automatski odrađuju pune se podacima za testiranje, nakon provedbe testova alat izdaje izvještaj.
5. Održavanje – zadnja faza u automatiziranom procesu, služi za provjeru rade li novo dodane funkcionalnosti u softveru ispravno. Održavanje se provodi nakon dodavanja novih automatskih testova kako bi se poboljšala efektivnost automatskih skripti.



Slika 4: Proces automatiziranog testiranja (Izvor: [25])

Automatizirano testiranje nudi različite prednosti, prvo bi bila ušteda vremena, jer zbog njihovog brzog izvođenja testerima omogućava manju potrebu nadziranja testova. To govori i podatak da je automatizirano testiranje čak 70% brže od ručnog testiranja. Testne skripte je moguće ponovno iskoristiti u drugim slučajevima. Također, odlični su kod testova koje je potrebno ponavljati jer nisu toliko naporni za testera kao kod ručnog testiranja [23]. No za razliku od ručnog testiranja, automatizirano testiranje je inicijalno skupo implementirati, alati mogu biti skupi. Uz to, za svaku promjenu u softveru potrebno je ispraviti testni slučaj po novim karakteristikama. Nakraju, nije primjeren za testiranja prihvaćanja i testiranje korištenja gdje su potrebni stvarni korisnici [26].

5. Alati za automatizirano testiranje softvera

Uspjeh u bilo kojem automatiziranom testiranju ovisi o odabiru odgovarajućeg alata za automatizaciju. Odabir pravog testnog alata za projekt jedan je od najboljih načina postizanja produktivnost projekta i uštede troškova [27]. Postoje mnoge vrste testnih alata koje mogu biti uzete u obzir prilikom odabira alata, a to mogu biti:

- Alati otvorenog koda (eng. *Open-source tools*)
- Komercijalni alati (eng. *Commercial tools*)
- Prilagođeni alati (eng. *Custom tools*)

5.1. Alati otvorenog koda

Alati otvorenog koda su softverski besplatni alati čiji je izvorni kod dostupan javnosti na mogućnost slobodnog korištenja, distribucije i modifikacije. Dostupni su za bilo koju fazu u procesu testiranja, od upravljanja testnim slučajevima pa sve do praćenja grešaka [27].

„Open-source“ okviri omogućuju korisnicima pristup izvornom kodu i njegovu izmjenu pod određenim uvjetima. Iako se termin "open-source" često koristi kao sinonim za "besplatni softver", uvijek se preporučuje provesti temeljito istraživanje jer često dolaze s skrivenim troškovima. „Open-source“ alati uz to nude i komercijalne usluge podrške oko njihove osnovne strukture kako bi pružili sigurnosnu mrežu organizacijama koje ih žele prihvatiti, tzv. „freemium“ model. Prednosti koje omogućuju „open-source“ alati su sljedeće [28]:

- Ušteda troškova – „open-source“ alati u odnosu na druge nude pristup njihovim mogućnostima s nikakvim ili minimalnim troškovima što smanjuje trošak projekta.
- Otvorenost modificiranju – većina okvira nudi modificiranje njihove osnovne strukture kako bi korisnik mogao zadovoljiti potrebe svojeg projekta. Neki okviri možda to ne omogućavaju, imaju par jeftinih servisa oko osnovnog dizajna te pružaju mogućnost pristupa i personalizacije prema potrebi korisnika.
- Puno korisnika – otvoreni alati često imaju aktivnu zajednicu korisnika i razvojnih programera, te za njih postoje podrške, dijeljenje znanja i rješavanje mogućnih problema.
- Kontinuirani razvoj – mnogi otvoreni alati imaju česte nadogradnje i poboljšanja, što ih čine dobrim za korištenje s obzirom da se sve greške poprave vrlo brzo prilikom prijave, a nove značajke se redovito dodaju.

Nedostaci koje „open-source“ alati imaju su [29]:

- Teško korištenje – neki alati mogu biti komplicirani za postavljanje i korištenje, te nemaju „user-friendly“ sučelje koje su korisniku poznate. Takve stvari mogu utjecati na produktivnost i onemogućiti testera da koristi alat.
- Nedostatak podrške – otvoreni alati ne dolaze uvijek s opcijama podrške i u slučaju nedostatka podrške ili stručnosti može otežati ostvarenje koristi u datom trenutku.
- Skriveni troškovi – vrlo često otvoreni okviri znaju imati skrivene troškove kako bi se korisniku omogućio potpuni pristup alatu, što u konačnici može završiti skuplje nego je korisnik zamislio.
- Sigurnost - provjera otvorenih alat za sigurnosne ranjivosti je ključna jer one mogu dovesti do zloupotrebe. Zbog prirode otvorenosti koda ovih alata bilo tko ima mogućnost prepravljanja koda čime je rizik preuzimanja tog alat puno veći.

Otvoreni alati imaju i nedostataka, ali su nedostaci nadmašeni prednostima koje donose prilagodba alat potrebama projekta i slobodno korištenje.

5.2. Komercijalni alati

Komercijalni alati se prodaju ili licenciraju korisnicima kako bi mogli koristiti njihove funkcionalnosti i mogućnosti, a za razliku od „open-source“ alata, kod komercijalnih alata nije javno dostupan. Za ove alate korisnici moraju platiti određeni iznos kako bi ih mogli koristiti ili pak kupiti licencu za njihovo korištenje. Neke od prednosti koje komercijalni alati nude su [29]:

- Sigurnost i pouzdanost – ovi alati prolaze mnoge testove sigurnosti i pouzdanosti prije nego budu stavljeni na tržište.
- Korisnička podrška – komercijalni alati imaju dobru pokrivenost korisničkom podrškom u slučaju da se dogodi bilo kakva greška, uz to ima dokumentaciju za korištenje alata koja je lagana za razumjeti.
- Jednostavno korištenje – mnogi komercijalni alati imaju ugrađene tokove rada koji olakšava kretanje i korištenje alata.

Nedostaci ovih alata su sljedeći [30]:

- Cijena – komercijalni alati mogu biti iznimno skupi iako su u današnjici su većinom temeljeni na pretplati, svota koju je potrebno izdojiti za njih je velika.
- Slaba fleksibilnost – ovi alati napravljeni su za točno određene zadatke, te je za potrebe izvođenja drugih poslova potrebno primijeniti neke druge alate.

5.3. Prilagođeni alati

Prilagođeni alati su softverski alati koje su tvrtke napravile kako bi zadovoljile posebne zahtjeve i potrebe prilikom izrade određenih softverskih projekata. U dogovoru sa voditeljem testiranja mogu se izraditi ovi alati ukoliko tim programera ima dovoljno dobro znanje [27]. Prednosti koje donose prilagođeni alati su [28]:

- Prilagodljivost – alati se mogu promijeniti i prilagoditi svakom projektu za koji se koriste, također moguće je odmah dodati ili maknuti nepotrebne značajke.
- Kontrola – testeri imaju potpunu kontrolu nad softverskim kodom budući da oni posjeduju IP adresu alata.
- Odlična podrška – u svakom trenutku korisnicima alata dostupna je stručna podrška.
- Iskoristivost – prilagođene alate je moguće upariti s drugim alatima koji su potrebni, uz to moguće ih je iskoristiti za buduće, slične projekte.

Nedostaci koji su prisutni kod prilagođenih alata su sljedeći [28], [29]:

- Vremenski zahtjevno – kao i svaki drugi program, prilagođeni alat mora biti dizajnirano, razvijeno i testirano kako bi bili sigurni da funkcionira, što ga čini dugotrajnim, zamornim i vremenski zahtjevnim procesom.
- Cijena – ovisno o značajkama i sigurnosnim provjerama koje se ugrade, izrada prilagođenog alata može biti iznimno skupa, zato je za izradu prilagođenog alata potrebno imati određenu količinu novaca da bi se alat mogao izraditi na najbolji mogući način.

Tablica 1: Karakteristike top 5 automatizacijskih alata

Alat	**Selenium**	**Katalon Studio**	**Appium**	**TestComplete**	**Cypress**
Vrsta aplikacije koja se testira	Web	Web, API, mobilne aplikacije, računalne aplikacije	Mobilne aplikacije (Android, iOS)	Web, API, računalne aplikacije	Web

Podržane platforme	Windows, macOs, Linux, Solaris	Windows, macOs, Linux	Windows, macOS	Windows	Windows Linux OS X
Konfiguracija	Potrebno programiranje	Lagana	Potrebno programiranje	Lagana	Potrebno programiranje
„Low-code“ i skriptni način	Skriptni načina	Oba način	Skriptni način	Oba načina	Skriptni način
Podržani skriptni jezici	Java, C#, Python, JavaScript, PHP, Ruby, Perl	Java, Groovy	Java, C#, Python, JavaScript, PHP, Ruby, Perl	JavaScript, Pyton, VBScript, Delphi, C++, C#	JavaScript
Napredni testni izvještaji	Ne	Da	Ne	Ne	Da
Cijena	Besplatno	Besplatno, Premium način od 167$ mjesečno	Besplatno	Od 3253€	Besplatno do 500 testnih rezultata, poslije od 67$ mjesečno

(Izvor: Prilagođeno prema: [31, str. 1])

5.4. Alati za E2E testiranja

5.4.1. Selenium

Selenium je besplatan „open-source“ alat koji služi za automatizirano testiranje web-aplikacija unutar različitih preglednika. Prvu inačicu ovog alata napravio je Jason Huggins 2004. godine, gdje kao zaposlenik tvrtke ThoughtWorks izrađuje JavaScript alat pod nazivom JavaScriptTestRunner koji je služio za testiranje interne aplikacije za vrijeme i troškove. S obzirom da je ova savjetodavna tvrtka radila po agilnoj metodi, automatizirano testiranje svih aplikacija je bio ključni stil rada ThoughtWorks-a [32]. Shvativši potencijal ovog alata da pomogne automatizirati i ostale web-aplikacije, pretvoren je u program otvorenog koda te je kasnije dobio ime Selenium Core [33].

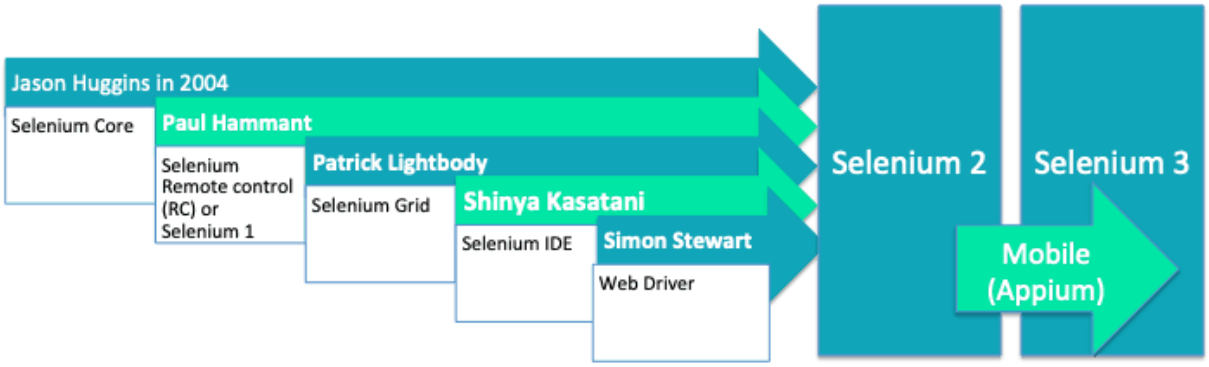
Problem nastaje zbog politike istog izvora (eng. *Same-Origin Policy*) koji zabranjuje JavaScript kodu da pristupi elementima domene koja nije s istog izvora. Iz tog razloga su testeri morali instalirali lokalno Selenium Core i web servis koji je sadržavao web aplikaciju koju su testirali kako bi imale istu domenu. Tako je Paul Hammant, također zaposlenik ThoughtWorks-a te iste godine je odlučio napraviti server koji će se ponašati kao HTTP posrednik (eng. *proxy*) koji služi kao varka pregledniku kako bi mislio da su Selenium Core i web aplikacija testirani preko iste domene. Takav pristup je nazvan Selenium daljinski upravljač (eng. *Selenium Remote control - Selenium RC*) ili Selenium 1 [34].

U priču se uključuje i Patrick Lightbody koji je imao ideju pod nazivom „Hosted QA“. Radi se o paralelnom testiranju koje se koristi u različitim preglednicima i operacijskim sustavima, čiji je cilj smanjivanje vremena koje je potrebno da bi se testovi izvršili i donijeli bolju učinkovitost. Patrick je zbog svoje ideje odlučio dati otkaz te je isprogramirao sustav za Hosted QA koji je snimao zaslon preglednika u različitim situacijama te istovremeno pružao uslugu klijentima [32].

Japanski programer web aplikacija Shinya Kasatani postao je zainteresiran u Selenium te je kreirao Selenium integrirano razvojno okruženje (eng. *Selenium Integrated Development Environment - Selenium IDE*) čija je funkcionalnost implementirati izvorni kod Seleniuma u ekstenziju preglednika Firefox, te s njim ima mogućnost snimanja testova i pretvara ih u skriptu za ponovno korištenje. Kasatani je donirao Selenium IDE u Selenium projekt 2006. godine [34].

Simon Stewart, softverski inženjer ThoughtWork tvrtke 2006.godine izradio je WebDriver, alat koji omogućuje testerima provođenje testova unutar bilo kojeg preglednika, ali ima kontrolu nad preglednikom na sistemskoj razini. Imao je bolji API od tadašnjeg Selenium

RC-a i to se pokazalo vrlo obećavajuće [34]. Stewart je svoj alat prezentirao na Googleovoj konferenciji u New Yorku 2007. godine i ubrzo poslije toga započeo je raditi na sklapanju njegovog alata s Selenium RC. To se ostvarilo 2008. godine kad je kreiran alat Selenium 2, kojem je WebDriver bila jezgra [33].



Slika 5: Povijest Seleniuma (Izvor: [33])

5.4.2. Selenium komponente

Kada govorimo o Seleniumu, moramo napomenuti da se ne radi o jednom alatu, već o skupu softvera od kojih svaki zadovoljava potrebe za određeni testni scenarij. Selenium se sastoji od četiri komponente koje uključuju:

- Selenium IDE – najjednostavniji okvir u Selenium paketu, radi se o ekstenziji za Mozilla Firefox ili za Chrome, nisu potrebne nikakve programerske sposobnosti za rad s ovim okvirom jer sve testove provodi preko snimanja i reprodukcije. Upravo zbog te jednostavnosti trebao bi se koristiti samo kao prototip alat [34]. Ali Selenium IDE nije koristan testerima zbog njegovih nedostataka poput sporog izvođenja automatskih testova te ne može biti upotrjebljen kod složenijih aplikacija gdje je potrebna podrška za uvjete i iteracije [33]. Uz to ne može provoditi testiranja nad bazom, ne podržava „data-driven“ testiranja, nije moguće provesti detaljni izvještaj o testiranju i nije moguće izvesti skripte u WebDriver [35].
- Selenium RC – dugo vremena i glavni okvir cijelog Selenium projekta, uz to je bio i prvi automatizacijski alat za testiranje web-aplikacija koji je dozvoljavao korisnicima da koriste programski jezik koji žele poput Jave, C#, PHP, Python, Ruby, Perl [33]. Za razliku od Selenium IDE-a bio je brži, podržavao je testiranje temeljeno na podacima i imao je mogućnost slikanja i snimanja zaslona. S druge strane imao je kompliciranu

arhitekturu što ga je činilo sporim, API-i su slabo objektno-orijentirani i potrebno je imati dobro početno znanje programiranja [36], [37].

- Selenium Grid – alat koji zajedno s Selenium RC-om odrađuje paralelne testove preko različitih računala i različitih preglednika u istom vremenskom tijeku [34]. Sastoji se od dvije komponente:
    - „Hub“ – server koji dozvoljava pristup zahtjeva sa WebDriver klijenta čije JSON testne naredbe šalje na udaljene upravljače na čvorovima (eng. *Node*). Prima instrukcije od klijenta i izvodi ih udaljeno na različitim čvorovima odjednom.
    - „Node“ – udaljeni uređaj koji ima svoj operativni sustav i udaljeni WebDriver, prima zahtjeve od hub-a u obliku JSON testnih naredbi koje potom izvršava pomoću WebDrivera.

  Prednosti su mu da je odličan za izvršavanje puno testnih skupova koji trebaju biti gotovi u što kraćem roku, ali potreba za iskusnim programerima i cijena koja je potrebna da se nabave dodatni „node-ovi“ su neki od nedostataka [38].

- Selenium WebDriver – poboljšana verzija Selenium RC-a, dodana u Selenium kako bi se nadišla ograničenja s kojima se Selenium RC našao. WebDriver također podržava razne programske jezike i platforme kako bi zadovoljio veću skupinu potreba, ali i zahtjeva znanje barem jednog programskog jezika [39]. Donosi moderniji i stabilniji pristup automatiziranom testiranju preglednika, te se ne pouzda na JavaScript za automatizirano testiranje, već kontrolira preglednik direktno komunicirajući s njime [34]. Neke od prednosti su mu: jednostavnija instalacija nego Selenium RC-a, realnija interakcija u pregledniku, brže vrijeme izvođenja od IDE i RC-a. Nedostaci koje ima su: zahtjeva znanje programiranja, instalacija je kompliciranija od Selenium IDE-a, nema ugrađen mehanizam za generiranje testnih rezultata [39].

Slika 6: Selenium komponente (Izvor: [35])

5.4.3. Katalon Studio

Katalon Studio je alat namijenjen za automatizirano testiranje web, API te mobilnih i Windows aplikacija. Dizajniran je za kreiranje i ponovo korištenje upotrijebljenih testnih skripti bez programiranja. Uz to još ima mogućnost testiranja automatskih testova nad elementima korisničkog sučelja, pa čak i skočne prozore i iFrameove, te je podržan je na Windowsima, Linuxu i macOS-u. Kako je sve ugrađeno u alat, korisnik se može usredotočiti na testne aktivnosti i potrošiti manje vremena na stvaranje, izvođenje i održavanje testova [40].

Prva verzija Katalon Studia puštena je u javnost u siječnju 2015. godine kao besplatni alat tvrtke Katalon Inc., a u listopadu 2019. godine dostupna je nova verzija Katalon Studio Enterprise koja nudi novije opcije poput izrade testnih profesionalnih testnih klasa, napredno izvršavanje testova i dinamički tokovi testiranja. Tada je predstavljen također i Katalon Runtime Engine, dodatak unutar Katalon Studio alata koji korisnicima omogućuje zakazivanje i izvođenje automatskih testova unutar sučelja naredbenog retka (eng. *command-line-interface – CLI*). Ovaj dodatak može se koristiti za planiranje testova, integraciju s CI/CD sustavom ili za izvršavanje testova na virtualnim „kontejnerima“ kao što je Docker [41].

Glavna prednost koju Katalon ima je lakoća implementacije i širi set integracije u odnosu na Selenium. Iduća stavka je da ima dvostruko sučelje za pisanje testova za korisnike

s različitim programskim znanjem što testerima s manje programskog znanja omogućava jednostavnije korisničko sučelje koje ne zahtjeve pisanje koda. Iskusniji korisnici imaju pristup pisanju koda s isticanjem sintaksi i mogućnost provjere grešaka. Katalon podržava lokalno i udaljeno testiranje, također i paralelno i sekvencijalno pokretanje testova, a radi na Groovy programskom jeziku [41].

Bitne značajke kod Katalon Studia su [42]:

- Jednostavnost i lakoća implementacije – kompaktni paket Katalon Studia sadrži gotovo sve što je potrebno za postavljanje ovog automatizacijskog alata.
- Brza i lagana instalacija – kako i instalaciju, ovaj alat nudi vrlo laganu konfiguraciju svojeg proizvoda. Nudi unaprijed pripremljene testne skripte i predloške kao što su biblioteke i objektni repozitoriji.
- Dostupna besplatna verzija – alat ima potpuno besplatnu verziju što mnogi korisnici često gledaju kao prednost. Besplatna verzija se preporuča za individualnog korisnika, dok se timovima preporučuje da koriste Enterprise verziju alata koja im daje više značajki i mogućnosti.

Kako svaki proizvod ima dobrih strana, tako ima i loših strana, kod Katalon Studia to su sljedeće stvari [41]:

- Mala zajednica – iako je proizvod napravljen 2015. godine, korisnika je manje nego kod većih konkurenata. Usporedbe radi, vezano za Katalon Studio na Stack Overflowu postoji 785 aktivnih pitanja, dok za Selenium WebDriver postoji čak 155.922 pitanja.
- Nedovoljno programskih jezika – samo jedan jezik je podržan u Katalonu, a to je Groovy. On je dio Jave, tako da će ga korisnici s Java znanjem moći koristiti.
- Zatvoreni alat – za razliku od Seleniuma koji je open-source alat i korisnici ga mogu prilagođavati svojim potrebama te koristiti zajedničke pakete (eng. *Community-packages*), Katalon ima zatvoren izvor koda što dovodi do manjeg broja programera u zajednici.
- Problemi performansa – korisnici su prijavljivali probleme s zaleđivanjem i usporavanjem rada programa, mobilno testiranje traje duže od drugih testiranja zbog potrebe za snimanjem ekrana i pisanjem koda.

5.4.4. Appium

Appium je „open-source“ projekt i sustav povezanih softvera koji je napravljen za olakšanu automatizaciju korisničkog sučelja na raznim platformama koje uključuju mobilne aplikacije, web preglednike, računalne aplikacije pa čak i za televizore [43].

U članku [44] spominje se izum Appium-a. Godine 2011., Dan Cuellar stvorio je iOSAuto, danas poznatiji kao Appium, s željom da smanji duljinu testnog prolaza (eng. *Test pass*) na iOS platformi. Postojala je mogućnost manje testiranja, ali to bi dovelo do dodatnog rizika. Istražujući alate koji su bili dostupni shvatio je da svaki od njih ima neke nedostatke, pa tako i Apple-ov alat UIAutomation čiji su testovi morali biti pisani u JavaScript-u i nisu imali mogućnost implementacije i otkrivanje grešaka. U konačnici je uspio uz pomoć Apple-ovog alata prikazati sekvencijalno poredane tekstualne datoteke da primaju naredbe, koristeći „eval()“ funkciju izvršio ih je i zatim pohranio na disk pomoću Pythona. Kod je prepravio u C# da bi zatim implementirao sintaksu u stilu Seleniuma za pisanje sekvencijski poredane JavaScript naredbe.

Ovaj alat koristi se većinom za automatizirano testiranje softvera, gdje pomaže odrediti da li funkcionalnosti aplikacije koja se testira radiju kako bi trebale. U odnosu na druge vrste softverskog testiranja, UI automatizirano testiranje omogućava testerima pisanje koda koje ih vodi kroz scenarij u stvarnom UI aplikacije čime se što je bliže moguće imitira situacija u stvarnom svijetu pri čemu se omogućuju raznorazne koristi automatizacije, uključujući brzinu, mjerljivost i konzistenciju [43].

Appium podržava sve jezike koje imaju Selenium biblioteke, kao što su Java, JavaScript, PHP, Ruby, Python, C#, ali ne podržava testiranje Androida ispod verzije 4.2. Uz to nema podrške za pokretanje Appium Inspectora na Windows sustavima [45].

5.4.5. TestComplete

TestComplete je automatizirani alat kreiran od strane SmartBear poduzeća za testiranje web, mobilnih i računalnih aplikacija. Uz pomoć ovog alat moguće je kreirati, upravljati i izvršavati testove. Jezici koje podržava su VBScript, Python, JavaScript, C++ i DelphiScripta . U ovom alatu postoji mogućnost reproduciranja izvođenja i kreiranja testova putem snimanja. Snimanjem ručnog testiranja procesa, moguće je reproducirati i obraditi ga kao automatizirano testiranje, jedinično testiranje te regresijsko testiranje [46]. Značajke koje TestComplete ima su sljedeće [47]:

- Distribuirano testiranje – moguće je izvoditi više automatskih testova na različitim uređajima.

- Testiranje snimanja i reprodukcije – snima samo ključne radnje koje su bitne za ponovno pokretanje testa i miče sve nepotrebne radnje.
- Test Visualizer – alat unutar TestComplete-a koji pomaže snimati zaslon automatski kod izvođenja testa, omogućava brzu usporedbu između očekivanog i dobivenog ekrana prilikom testiranja.
- Mogućnost čitanja metoda i opcija internih objekata – alat čita imena mnogih elemenata poput .NET, Java, Visual Basic aplikacije, Delphi i sličnih.

Prednosti TestComplete alata [46]:

- Lagano korištenje – sa svojim jednostavnim integracijskim značajkama, ovo je lagano dostupan i koristan alat, korisnici koji ne znaju niti jedan programski jezik mogu koristiti TestComplete.
- Prilagodljiv – u slučaju da korisnik nije zadovoljan rezultatom koji alat za uređivanje izradi, u ovom alatu moguće je pisati i mijenjati skriptu ručno.
- Stalna unaprjeđenja – korisnici mogu uvijek očekivati odličnu korisničku podršku, visoku razinu održavanja i konstantna ažuriranja alata.

Nedostaci koji su prisutni u TestComplete-u [47]:

- Licence za TestComplete su skupe, cijena za Base vrstu alata počinje od € 3253
- Alat je podržan samo na Windows sustavu, ne radi na macOS-u.
- Potrebni su ručni koraci za ažuriranje testnih slučajeva.
- Nema dovoljno podrška preko otvorene zajednice.
- Službena dokumentacija nije jednostavna.

5.4.6. Cypress

Cypress je alat za testiranje front-end dijela weba, baza mu je JavaScript. Zadatak mu je riješiti poteškoće s kojima se susreću programeri ili osiguravatelji kvalitete (eng. *Quality Assurance – QA*) tijekom testiranja aplikacija. Alat je prilagođen programerima za korištenje i primjenjuje model objekata dokumenta (eng. *Document Object Model – DOM*) te se izvodi direktno u pregledniku. Uz to Cypress ima interaktivni testni provoditelj koji služi za pokretanje testova u kojem se izvršavaju sve naredbe. Ovaj alat korisnici mogu primjenjivati za jedinično testiranje, integracijsko testiranje i E2E testiranja [48]. Često ga se uspoređuje sa Seleniumom, ali ovaj alat je arhitekturalno i bitno drugačiji na način da se u Cypressu testiranje izvodi u istom ciklusu kao i aplikacija koja se testira, dok se kod drugih testiranja izvode van

preglednika gdje se naredbe izvršavaju putem mreže. Ovaj alat nije ograničen kao Selenium, to mu daje prednost da su testovi pisani u Cypressu pouzdaniji, provode se brže te ih je lakše pisati [49]. Kako je Cypress lokalno pohranjen na računalo, ima mogućnost pristupanju operacijskom sustavu kako bi mogao obavljati automatske testove, uz to uključuje i neke druge mogućnosti poput snimanja i slikanja zaslona, obavljanja operacija na mreži te operacija nad općim sustavom [50].

Cypress je napravljen s ciljem kako bi razvoj i testiranje bili jedan paralelni proces i u tome pomažu arhitekturalna poboljšanja ovog alata koja omogućuju testerima izvođenje razvoja vođenog testovima (eng. *Test-Driven Development – TDD*) za cjelokupno E2E testiranje [48].

Cypress ima značajke koje niti jedan drugi testni okvir nema, a to su [51]:

- Automatsko čekanje – kod ovog alat nije potrebno koristiti naredbe za čekanje kao kod Seleniuma, Cypress automatski čeka naredbe prije nego nastavi s testom.
- Brzina – Cypress je brz iz razloga što testove provodi u pregledniku.
- Jednostavno otklanjanje grešaka – mogućnost ispravka greški je dostupna odmah u razvojnom alatu, greške koje postoje su uočljive te ih je moguće vrlo brzo ispraviti.
- Spies, Stubs i Clocks – koncepti Cypressa koji se koriste za lakše testiranje određenih dijelova aplikacije.
    - „Spies“ predstavlja objekte koji imaju mogućnost praćenja i bilježenja pozvanih metoda ili funkcija unutar programskog koda, koristi se kod za provjeru koliko se koja metoda puta poziva tokom izvođenja testa.
    - „Stubs“ služi kao dopuna za stvarne komponente kojima se simulira njihovo ponašanje u određenim uvjetima, koristi se za testiranje bez stvarnog spajanja s resursima.
    - „Clocks“ su alati koji kontroliraju vrijeme u testnom okruženju, korisna stvar tijekom testiranja situacija u različitim vremenskim intervalima ili za simuliranje vremenskih ovisnosti.
- Upravljanje mrežnim prometom – jednostavnije kontroliranje, mijenjanje te testiranje krajnjih slučajeva bez potrebe korištenja servera.
- Snimanje zaslona – slike zaslona su automatski okinute ako dođe do pada testa, ali je i snimanje zaslona moguće kroz cijeli testni skup.

Neki od nedostataka Cypressa su sljedeći [49]:

- Nije moguće pokretanje u više od jednog preglednika istovremeno.
- JavaScript je jedini podržani jezik za kreiranje testnih slučajeva.
- Slaba podrška za unutarnji okvir (eng. *Inline Frame – iFrame*).

5.5. Alati za jedinično testiranje

5.5.1. JUnit

JUnit je besplatan okvir za jedinično testiranje u Java programskom jeziku, zbog mogućnosti pisanja ponavljajućih testova vrlo je koristan Java programerima. Ovaj okvir razvili su Erich Gamma i Kent Beck prilikom leta avionom iz Zuricha prema Atlanti 1997. godine [52].

Ovaj „framework“ daje mogućnost testiranja dinamičkih testova uz pisanje malo koda, ali svejedno su testovi čisti i pregledni. Osim proširenja, JUnit uključuje i jako dobre sofisticirane značajke testiranja poput prosljeđivanja parametara testnim metodama i mogućnost umetanja ovisnosti (eng. *dependencies*) [14].

Način integracije JUnit okvira i programskih alata za Java jezik poput Eclipse IDE ili IntelliJ IDEA je vrlo jednostavan, također moguće ga je povezati sa platformama za testiranje softvera poput ScalaTesta, LambdaTesta, TestContainera itd. Zadnja objavljena verzija ovog okvira je JUnit 5 i sadrži najnovije značajke Jave [14].

Značajke koje ima su mnoge, od dugog popisa liste anotacija koje se mogu izvršavati i identificirati, pa do Assertion metoda koje pomažu u provjeri očekivanog rezultata. Ima svoj ugrađeni predložak pomoću kojeg je moguće vrlo brzo implementirati i pokrenuti test. Uz to što ima mogućnost jednostavnog stvaranja i izvođenja testova, JUnit nudi programeru detaljan izvještaj o testu [53].

5.5.2. NUnit

Nunit je okvir za testiranje jedinica unutar svih .NET programskih jezika. U počecima je prenesen iz JUnita, ali u trenutnoj produkcijskoj verziji 3 je potpuno izmijenjen i stavljeno je puno novih značajki za širu primjenu .NET platforme [54]. Napisan je u C# jeziku i može testirati kod bilo kojeg .NET jezik, uključujući i C#, VB.NET i F#. NUnit sadrži nekoliko Assertion

metoda kojima se provjerava ispravan rad koda koji se testira. U sebi ima i atribute kojima se može kontrolirati izvođenja testova [55].

Značajke koje NUnit ima [56]:

- Lagan za naučiti.
- Sličan je Junit okviru, što znači da ima slične značajke.
- Velik izbor anotacija čine ovaj način testiranja bržim.
- Podržava jedinično, integracijsko i E2E testiranja.
- Ima mogućnost izvađanja paralelnih testova.
- Podržava način za otklanjanje grešaka.
- Dobra podrška zajednice.

5.5.3. TestNG

Test Next Generation je puni naziv ovog „frameworka“ za testiranje u Java jeziku. Cedric Beust napravio je TestNG u 2004. godini na temelju JUnita, ali njegova zamisao bila je jači i fleksibilniji testni okvir. TestNG je „open-source“ projekt koji je izdan pod licencom Apache Licence 2.0 [57]. Podržava vrlo napredne značajke za pisanje testova poput grupiranja testnih slučajeva uz definiranje redoslijeda i parametrizaciju. Osim toga, pruža automatizacijske sposobnosti poput stvaranja prilagođeni testnih izvještaja i skripte koje su lakše čitljive [14].

Kao i NUnit, ovaj okvir pokriva više kategorija testiranja poput jediničnog, funkcijskog, integracijskog te E2E testiranja. Inspiriran je po okvirima NUnit i JUnitu te ima slične anotacije kao i oni, ali i svoje dodatne funkcionalnosti poput zapisivanja prilikom izvođenja testova te višenitnost (eng. *Multithreading*) [14].

Glavne značajke TestNG okvira su sljedeće [57]:

- Anotacije – koristi anotacije za testove, davanje prioriteta te konfiguraciju drugih aspekata kako bi se test trebao odvijati.
- Paralelni testovi – ima jako dobro podržavanje za paralelne testove, čime se smanjuje vrijeme testiranja.
- Fleksibilno izvješće – zbog svojeg načina izvještaja, ima dobar način ispravljanje grešaka ili može poslužiti kao dokumentacija dioničarima (eng. *Stakeholders*).
- Testiranje vođeno podacima (eng. *Data-driven testing – DDT*) – TestNG omogućuje jednostavnu konfiguraciju testova, što je korisno kod DDT-a.
- Unos parametara – TestNG dopušta unos parametara u testove, što se može koristiti za konfiguraciju ili pružanje podataka testiranju.

- Ugrađena podrška grupiranja testova – u ovom okviru moguće je grupirati testove, to nam omogućuje pokretanje podskup testova.

5.6. Alati za integracijsko testiranje

5.6.1. Citrus

Citrus je popularan alat integracijskog testiranja, široko korišten za implementaciju softvera. Prva verzija ovog okvira bila je javnosti dostupna 2006. godine od strane tvrtke ConSol i od tada neprekidno rade na poboljšanju. ColSol koristi Citrus za sve svoje procesne integracijske aplikacije (eng. *Enterprise Application Integration - EAI*) pri čemu izvodi više tisuća testova svakodnevno. Podržava programski jezik Java, a radi na platformama poput weba, Kafke itd. [58], [59].

Citrus pruža okvir za testiranje interakcije između različitih dijelova sustava, od web usluga, baze podataka, sustava razmjene poruka i slično. Dopušta programerima stvaranje integracijskih testova koji provjeravaju E2E funkcionalnosti njihovih aplikacija. Citrus podržava više prijenosnih protokola čime postaje prilagodljiv za upotrebu u različitim testnim scenarijima. Programeri uz pomoć Citrusa imaju mogućnost automatiziranja integracijskog procesa, tako štedeći vrijeme i osiguravajući kvalitetu proizvoda. U cjelini, ovo je moćan alat za integracijsko testiranje s mogućnošću pomaganja programerima pronalaska i odstranjivanja problema prije nego što dođu u produkcijsku aplikaciju [58].

Glavne značajke alata Citrus [58]:

- Napravljen je za testiranje sustava poruka, to ga čini pravim izborom kod testiranja sučelja za razmjenu poruka i web servisa.
- Fleksibilnost testiranja ovim alatom omogućava testerima konfiguraciju testova koristeći XML, Java ili Groovy.
- Citrus se spaja s alatima kontinuirane integracije (eng. *Continuous Integration – CI*) kao što su Jenkins i Bamboo, što olakšava integraciju testova u proces isporuke.

5.6.2. Tessy

Tessy je alat napravljen za integracijsko i jedinično testiranje sa ISO 26262 certifikatom. Idealan je za automatska testiranja sa visokom razinom sigurnosti. Koristi se jedinična i integracijska testiranja unutar ugrađenih sustava (eng. *Embedded systems*), također i za regresijska testiranja. Podržava vodeće platforme mikro kontrolera kao što su TI, Microchip, Infeneon i slični. Tessy pomaže testerima i programerima na način da brine o pokrivenosti i dosljednosti programskog koda [60], [61].

Tessy brine o pokrivenosti koda aplikacije, te imamo mogućnost kreiranja testnih slučajeva uz pomoć uređivača klasifikacijskog stabla (eng. *Classification Tree Editor – CTE*), također postoji mogućnost korištenja uređivača testnih podataka (eng. *Test Data Editor – TDE*) za izmjenu testnih podataka [62].

Značajke Tessy alata [60], [61]:

- Mogućnost slijeđenja podataka
- Podrška za skoro sve mikro kontrolere.
- Podržava regresijsko testiranje, što je bitno kod softvera zbog čestog ažuriranja.
- Pokrivenost koda čini ovaj alat jako pouzdanim.
- Jezici poput C/C++ su podržani u Tessy.
- Stvara testni izvještaj na temelju testnih rezultata.

5.6.3. FitNesse

Također okvir otvorenog koda, služi za integracijsko testiranje web aplikacija koji olakšava komunikaciju između programera, testera i dioničara. Kako ovaj okvir omogućava formuliranje i izvođenje integracijskih testova u stilu „wiki“ stranice, korisnici koji nisu upoznati s tehnologijom ga također mogu razumjeti i koristiti [63].

Dobra stvar FitNessea je njegova kompatibilnost s različitim testnim okvirima kao JUnit i NUnit, isto tako i njegova sposobnost izrade izvještaja u različitim formatima kao što su HTML, JUnit i JSON. To olakšava korištenje okvira kojeg programer „voli“ koristiti. FitNesse također nudi i nekoliko značajki provjere, poput testiranja web servisa i automatiziranja testova prihvaćanja [64].

Automatski testovi mogu biti pisani u jezicima poput Jave ili .NET-a. FitNesse radi dobro s CI alatima, na taj način omogućavajući da se testovi izvode kao dio CI procesa. Ovo omogućava redovno izvođenje testova prihvaćanja i brzu povratnu informaciju o radu sustava.

Prednosti FitNesse alata su sljedeće:

- Okvir otvorenog koda
- Jednostavna instalacija
- Podržava programske jezike poput Jave, C#, Pythona.

Nedostatak koji se javlja kod FitNesse-a je sporo izvođenje testova [63].

6. Primjer provedbe automatiziranog testiranja

6.1. Opis testne aplikacije

Web aplikacija koja se testira u ovom završnom radu je eRačunPlus, već izrađena aplikacija na kojoj postoji mogućnost kreiranja i filtriranja e-Računa. Aplikacija se sastoji od ukupno 3041 linije koda, od čega je 670 linija napravljeno za poslužiteljsku stranu aplikacije (eng. *Backend*), dok je 2371 linija koda napisana za klijentsku stranu (eng. *Frontend*) aplikacije. Testiranja koja će se provoditi nad ovom aplikacijom su sljedeća:

- E2E testiranje
- Jedinično testiranje
- Integracijsko testiranje

Za izradu poslužiteljske strane dijela ove aplikacije korišten je program Eclipse IDE, alat otvorenog koda koji služi za izradu Java aplikacija. Ovaj alat podržava i druge programske jezike, a neki od njih su Fortran, C/C++, JavaScript, PHP, Ruby, Scala [65].

Za izradu klijentske strane aplikacije korišten je Visual Studio Code, besplatan uređivač izvornog koda, dostupan za Web, Windows, Linux, macOS i za Rasberry Pi OS. Alat dolazi s već unaprijed ugrađenom podrškom za JavaScript, Node.js te TypeScript, ali ima mogućnost podržavanja drugih programskih jezika kao što su C++, C#, Java, Python, PHP. Uz to ima mogućnost spajanja na „oblak“ (eng. *Cloud*) [66].

Kao baza podataka korišten je H2, alat otvorenog koda koji može biti ugrađen u Java aplikacije ili biti pokrenut u klijent-poslužitelj (eng. *Client-server*) načinu rada. Uglavnom se H2 koristi kao baza podataka memorije, te se podaci neće trajno spremati na disk. Ne preporučuje se za produkcijsku okolinu upravo zbog svoje integrirane baze podataka, služi za razvoj i testiranje projekata [67].

6.2. Testiranje s kraja na kraj

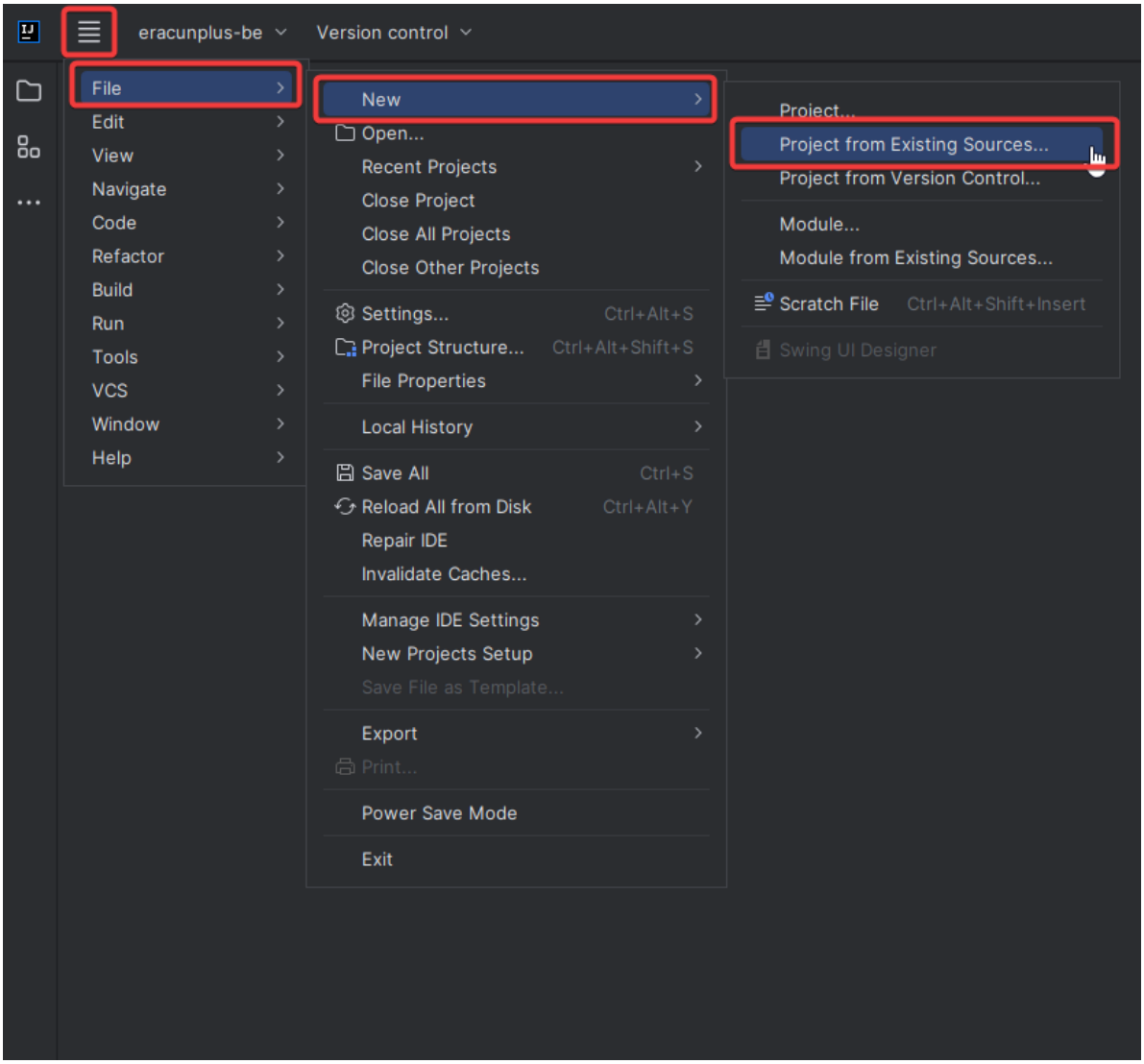
Za testiranje E2E dijela aplikacije koristit će se Selenium okvir unutar programa IntelliJ IDEA. Testiranje će se provoditi unutar backend dijela projekta, te je za to potrebno klonirati projekt s GitHub-a. Nakon uspješnog kloniranja i za mogućnost korištenja Selenium WebDrivera potrebno je preuzeti i instalirati sljedeće alate:

- Geckodriver – mehanizam Firefox web pretraživača, razvijen od strane Mozilla Corporationa. Geckodriver je veza između Selenium testova i Firefox preglednika,

„proxy“ za korištenje W3C WebDriver kompatibilnih klijenata koji im dopušta interakciju s preglednicima koji imaju se osnivaju na Gecko mehanizamu [68].

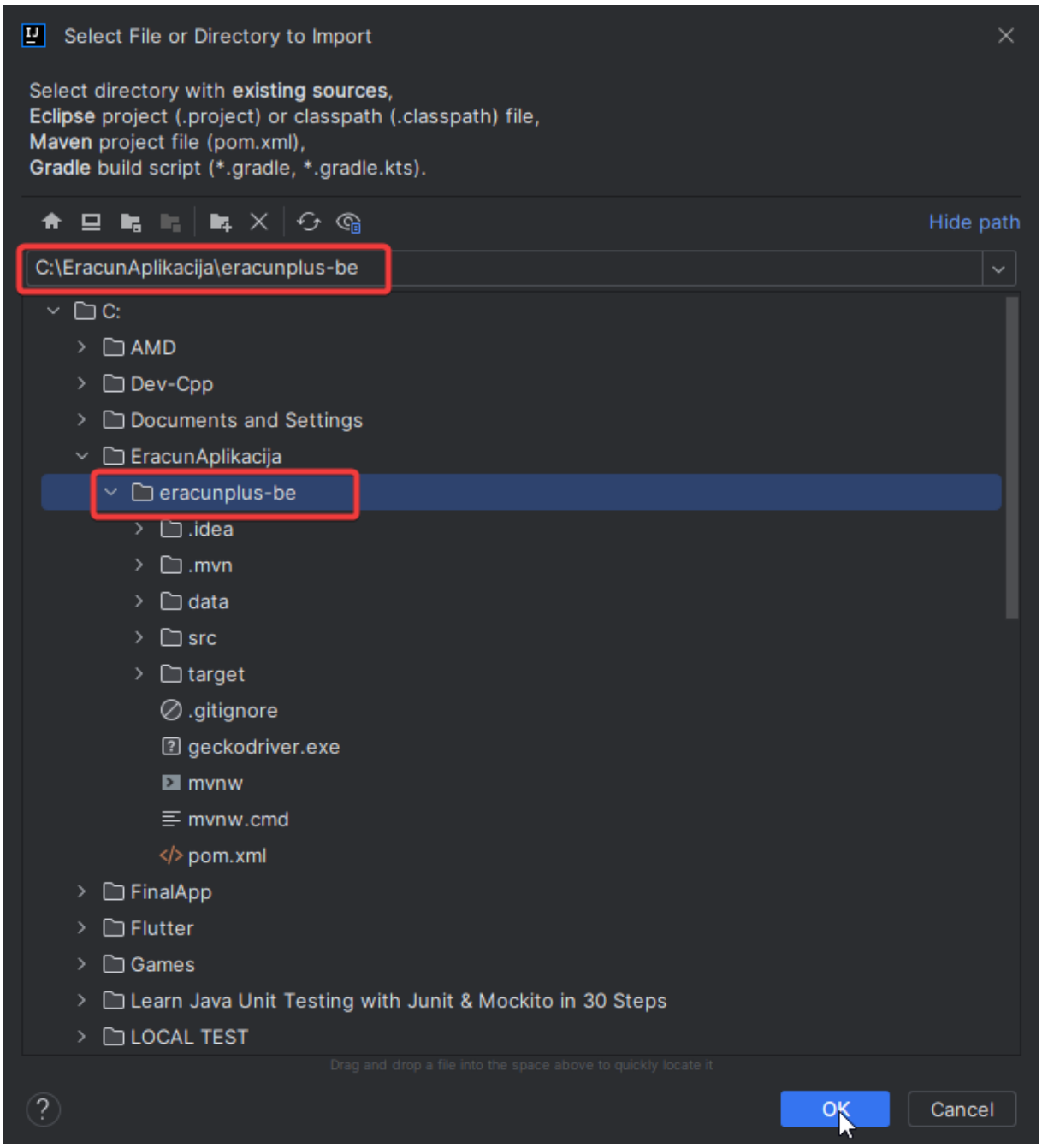
- IntelliJ IDEA Community Edition 2023.1 (kraće IntelliJ).
- Selenium Java WebDriver.
- SelectorsHub – dodatak unutar Mozilla Firefoxa za automatizirano generiranje XPath, CSS selektora te jQuery i JS putanje. Također je moguće ručno unošenje XPath i CSS selektora za sve web elemente [69].

Nakon instaliranja ovih komponenti, možemo započeti s konfiguracijom Seleniuma unutar IntelliJ alata. Kreiramo novi projekt iz postojećih izvora na način da u alatnoj traci odaberemo *File>New>Project from Existing Sources* (slika 3).



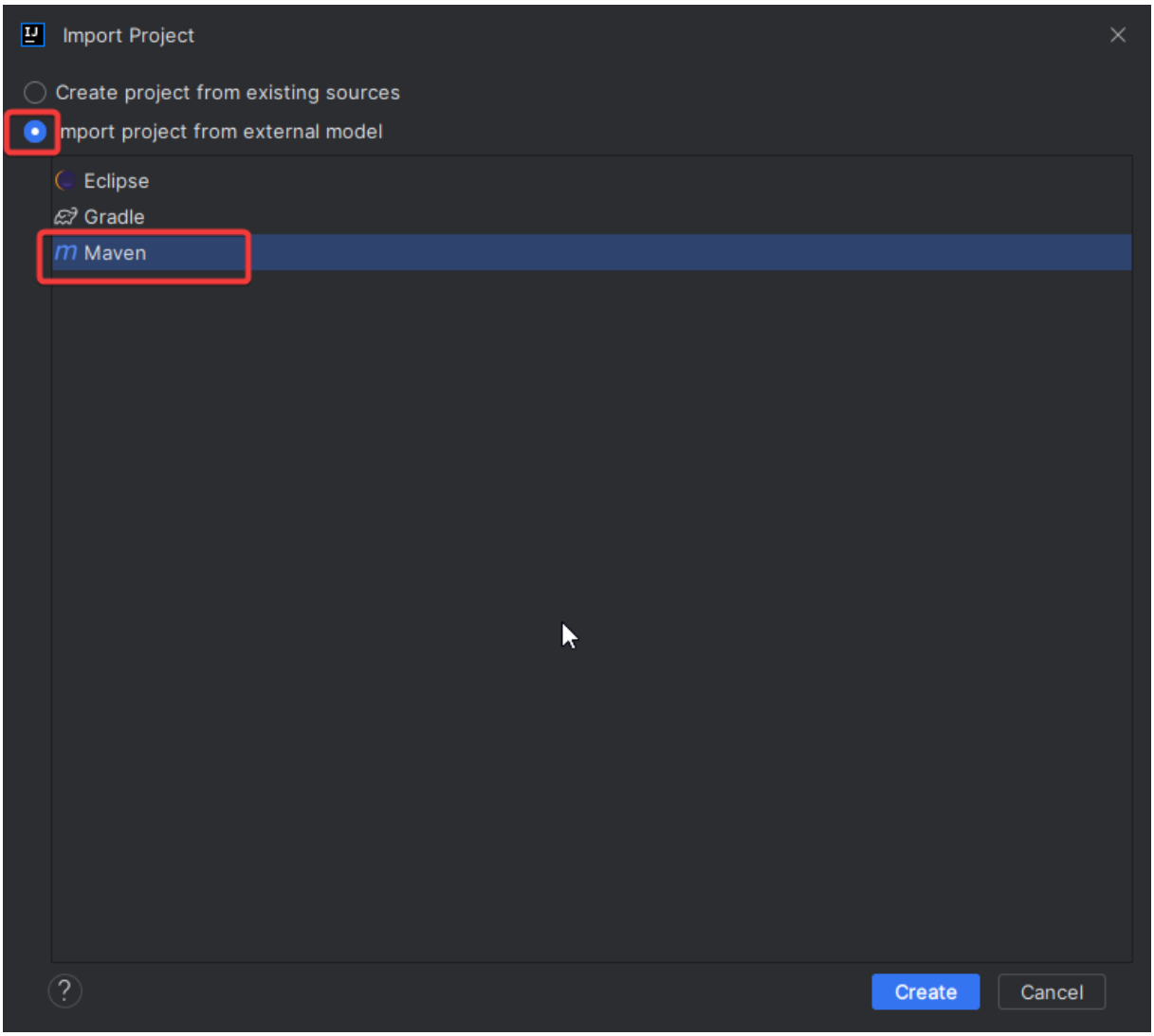
Slika 7: Stvaranje projekta unutar IntelliJ IDEA alata [Autorski rad]

Otvara nam se prozor u kojem odabiremo mapu u kojoj se nalazi aplikacija, odaberemo putanju do projekta i kliknemo gumb „OK“.



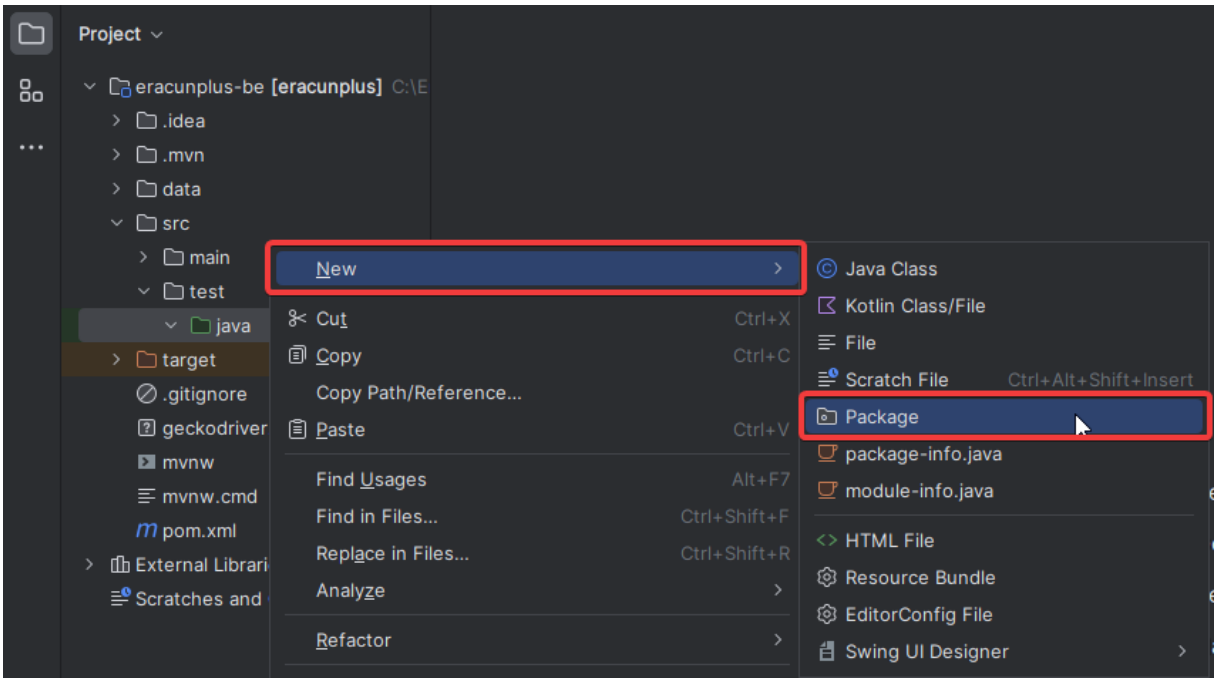
Slika 8: Odabir mape u kojoj se nalazi projekt [Autorski rad]

U novom prozoru potrebno je kliknuti na „Import project from extrenal model“ i odabrati „Maven“.



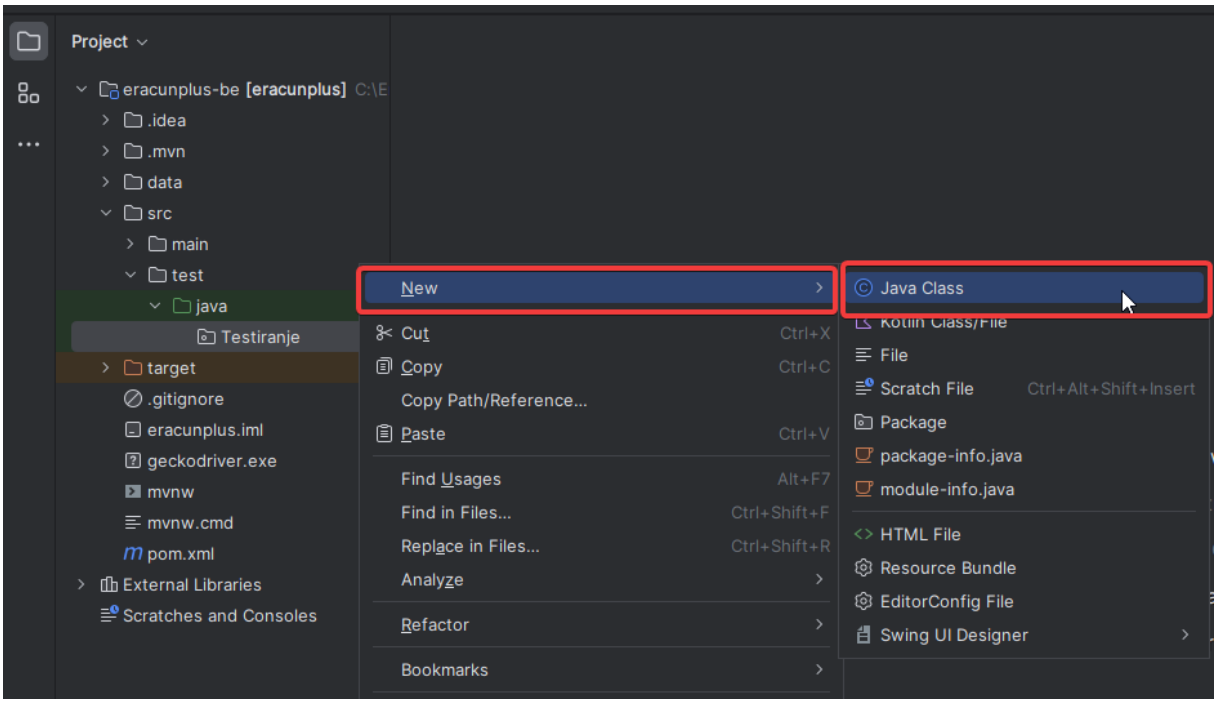
Slika 9: Uvoz projekta [Autorski rad]

Sljedeći korak je dodati pakete unutar projekta, točnije unutar direktorija „test“. Desnom tipkom miša kliknemo na „java“ i odaberemo *New>Package* i imenujemo paket „testiranje“.



Slika 10: Dodavanje paketa [Autorski rad]

Zatim moramo napraviti novu Java klasu, na način da desnim klikom na napravljenom paketu odaberemo *New>Java Class* te je nazovemo KreiranjeRacunaTest.

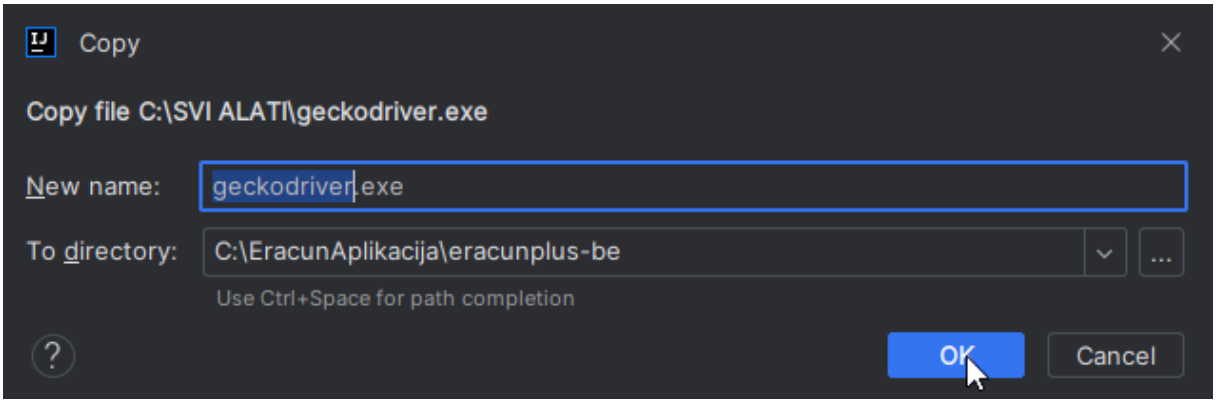


Slika 11: Kreiranje Java klase [Autorski rad]

Nakon ovih koraka klasa u kojoj testiramo izgleda ovako.

Slika 12: Prva klasa unutar projekta [Autorski rad]

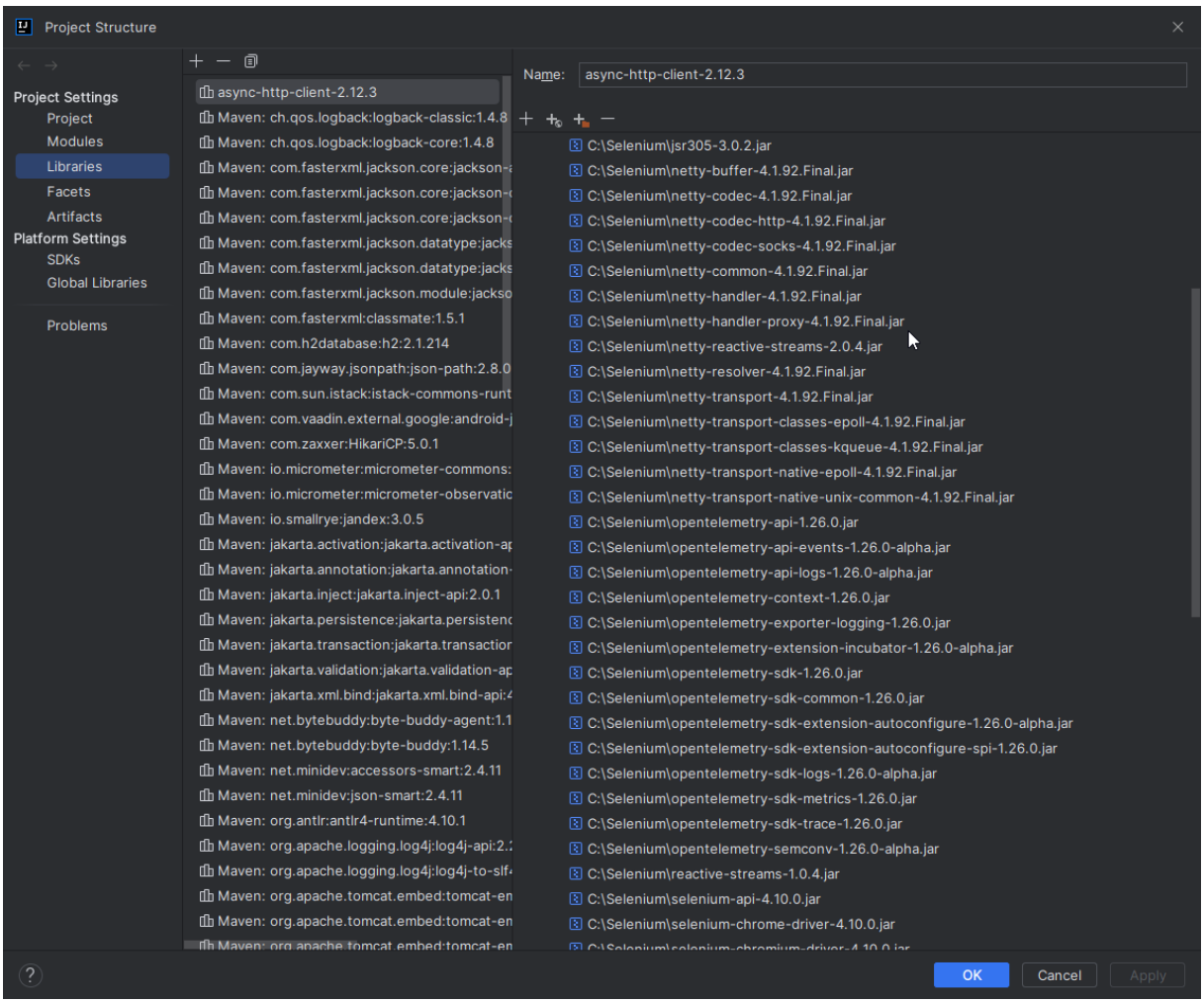
Sad je vrijeme za dodavanje Geckodrivera u projekt, za to je potrebno pronaći mjesto gdje je driver preuzet, kopirati ga te unijeti u IntelliJ aplikaciju na način da kliknemo na „eracunplus-be“ i pomoću prečice na tipkovnice crtl+v zalijepiti Geckodriver.



Slika 13: Dodavanje Geckodrivera u projekt [Autorski rad]

Prije kraja potrebno je dodati i Selenium biblioteke unutar projekta. Desnim klikom miša na „eracunplus-be“ i kliknemo na ikonicu ≡ > *File>Project Structure…* Otvara se novi prozor „Project Structure“ i potrebno je kliknuti na „Libraries“. Unutar „Libraries“ kliknuti na znak „+“ te odabrati Java biblioteku. U novom prozoru odabrati sve .jar datoteke koji se nalaze u Selenium

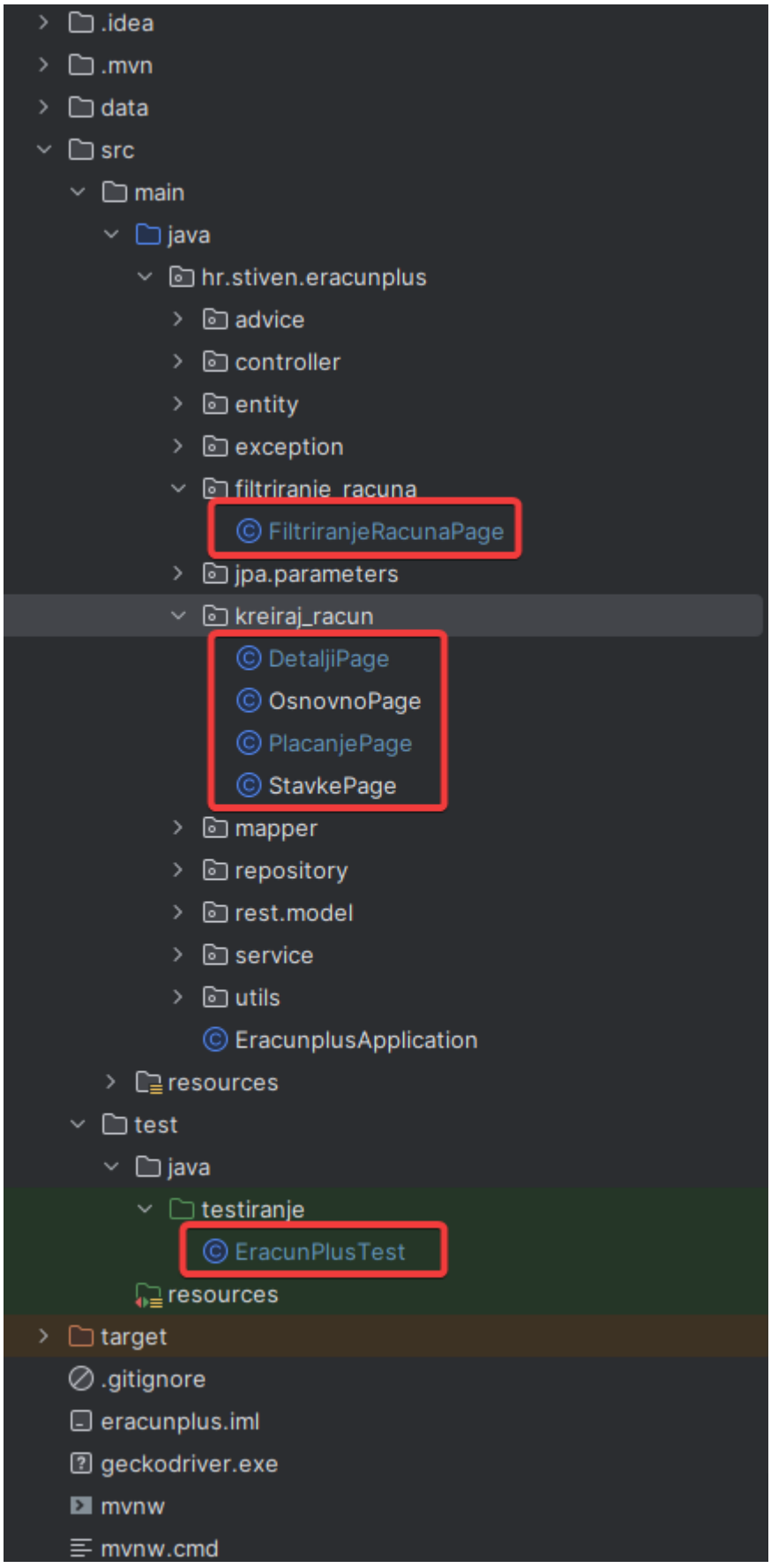
WebDriver datoteci te kliknuti gumb „OK“. U prozoru „Project Structure“ kliknuti na „Apply“ te zatim „OK“. Sad je naš projekt spreman za Selenium testiranje u Mozilla pregledniku.



Slika 14: Dodavanje Selenium .jar datoteka u projekt [Autorski rad]

6.2.1. Implementacija E2E testova

Izrada testova odrađena je primjenom uzorka Page Object Model-a (eng. *Page Object Model - POM*) i Page Factorya. POM je uzorak u kojem je svaka stranica na webu izrađena kao posebna klasa, dok je Page Factory klasa od strane Selenium WebDrivera za podržavanje uzorka Page Object dizajna [70]. Ovdje se koristi anotacija @FindBy, te se metodom initElements inicijaliziraju web elementi. U ovom projektu klase izgledaju ovako:



Slika 15: Klase po uzoru na Page Object Model [Autorski rad]

Klasa „OsnovnoPage“ sadrži implementaciju: dohvaćanja „headera“ stranice te dolazak na ekran „Kreiraj račun“. Uz to ovdje se unose podaci koji su nužni za mogućnost kreiranja e-Računa. Sljedeća po redu je klasa „DetaljiPage“ te je u nju implementiran dolazak na ekran „Detalji“, dohvaćanje headera tog ekrana te također unos podataka vezanih uz detalje računa.

U klasi „StavkePage“ nalazi se implementacija dolaska na navedeni ekran, klika na gumb za dodavanje stavke računa te također unos potrebnih podataka za kreiranje stavke e-Računa. Klasa „PlacanjePage“ sastoji se od dolaska na ekran Plaćanje, odabira načina plaćanja e-Računa te nakraju klika na gumb Spremi račun. Ove stavke se nalaze u posebnom paketu pod nazivom „kreiraj_racun“ dok se klasa „FiltriranjeRacunaPage“ nalazi u drugom paketu iz razloga što nisu dijelom iste stranice na platformi. „FiltriranjeRacunaPage“ sastoji se od dolaska na stranicu „Računi“, klika na padajući izbornik valute, odabira valute i u konačnici klika na gumb „Pretraži“.

„EracunPlusTest“ je klasa u kojoj se pišu testni slučajevi i priprema se preglednik u kojem se odvija testiranje. Tako će se u njoj inicijalizirati svi objekti iz „Page“ klasa i akcije koje se provode unutar tih klasa. U nastavku ću opisati jednu od navedenih klasa i pobliže objasniti implementaciju koda.

6.2.2. Klasa „FiltriranjeRacunaPage“

Unutar klase „FiltriranjeRacunaPage“ napravljena je implementacija u kojoj se:

- Klikne na gumb „Računi“
- Klikne na padajući izbornik „Valuta“
- Odabire se valuta „EUR“
- Klikne na gumb „Pretraži“

Kod izgleda ovako:

```
package hr.stiven.eracunplus.filtriranje_racuna;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import java.time.Duration;
```

Ovdje se nalaze biblioteke potrebne za provođenje testova.

```
public class FiltriranjeRacunaPage {
   WebDriver driver;
   @FindBy(xpath =
"/html[1]/body[1]/div[1]/div[1]/main[1]/form[1]/div[1]/div[1]/div[1]/div[1]
/div[7]/div[1]/div[1]/div[1]")
   WebElement ddValutaFilter;
   @FindBy(xpath =
"/html[1]/body[1]/div[1]/div[1]/div[1]/div[1]/ul[1]/li[1]/a[1]/div[2]/span[
1]")
```

- @FindBy anotacija služi za određivanje lokacije objekta WebElementa, u ovom slučaju koristimo xpath elementa. XPath predstavlja putanju do određenog HTML elementa na stranici. U ovom slučaju ovo je putanja do varijable „btnRacuni“.

```
WebElement btnRacuni;
```

- „btnRacuni“ je varijabla WebElementa, što je sučelje u Seleniumu koje omogućava interakciju s web elementima stranice.

```
@FindBy(xpath = "//li[normalize-space()='EUR']")
WebElement btnFilterEUR;

@FindBy(xpath = "//button[contains(text(),'Pretra i')]")
WebElement btnPretraziFIlter;

public FiltriranjeRacunaPage(WebDriver driver){
   this.driver = driver;
   PageFactory.initElements(driver, this);
}
```

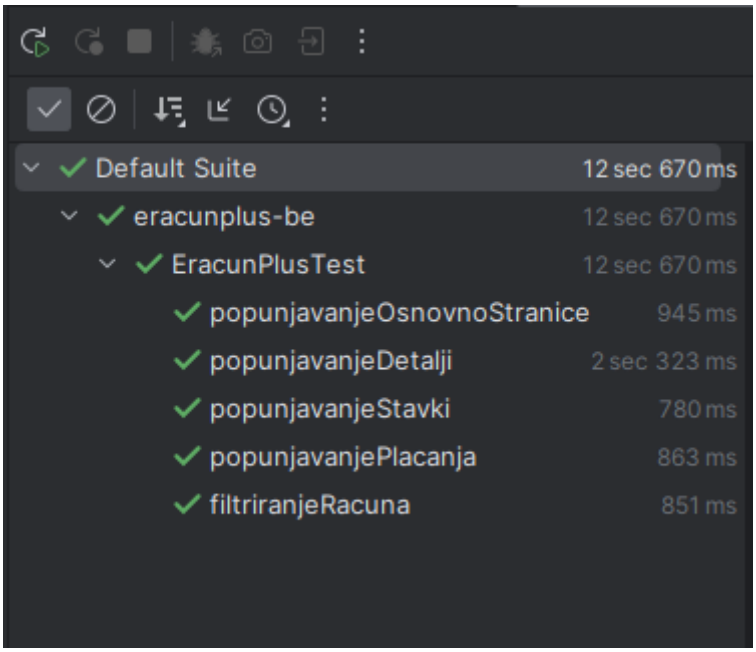
- Unutar „FiltriranjeRacunaPage“ konstruktora potrebno je instancirati elemente koji se koriste unutar ove klase. Konstruktor prima WebDriver kao argument, koji služi za interakciju s web stranicom. Dio koda koji je bitan je „PageFactory.initElement(driver,this);“ jer se ovdje inicijaliziraju svi elementi koje sa stranice i spremi su za korištenje.

```
public void klikRacuni(){ btnRacuni.click();}
```

- U metodi „klikRacuni“ se simulira klik na gumb „btn.Racuni“.

```
public void odabirValuteFilter(){ddValutaFilter.click();}
public void odabirEURFilter(){btnFilterEUR.click();}
public void pretraziFilter(){btnPretraziFIlter.click();}
}
```

U konačnici možemo vidjeti da su svi testovi prošli.



Slika 16: Uspješno izvršeni testovi [Autorski rad]

6.3. Jedinično testiranje

Jedinično testiranje aplikacije također se odvija unutar programa IntelliJ IDEA, testiranjem poslužiteljskog dijela projekta. Nakon uspješnog uvoza projekta u IntelliJ potrebno je preuzeti i instalirati sljedeće alate:

- Mockito – okvir za testiranje aplikacija. Njegova svrha je olakšati jedinično testiranje stvaranjem oponašatelja (eng. *Mock*), koji oponašaju ponašanje stvarnih objekata programa.
- JUnit 5

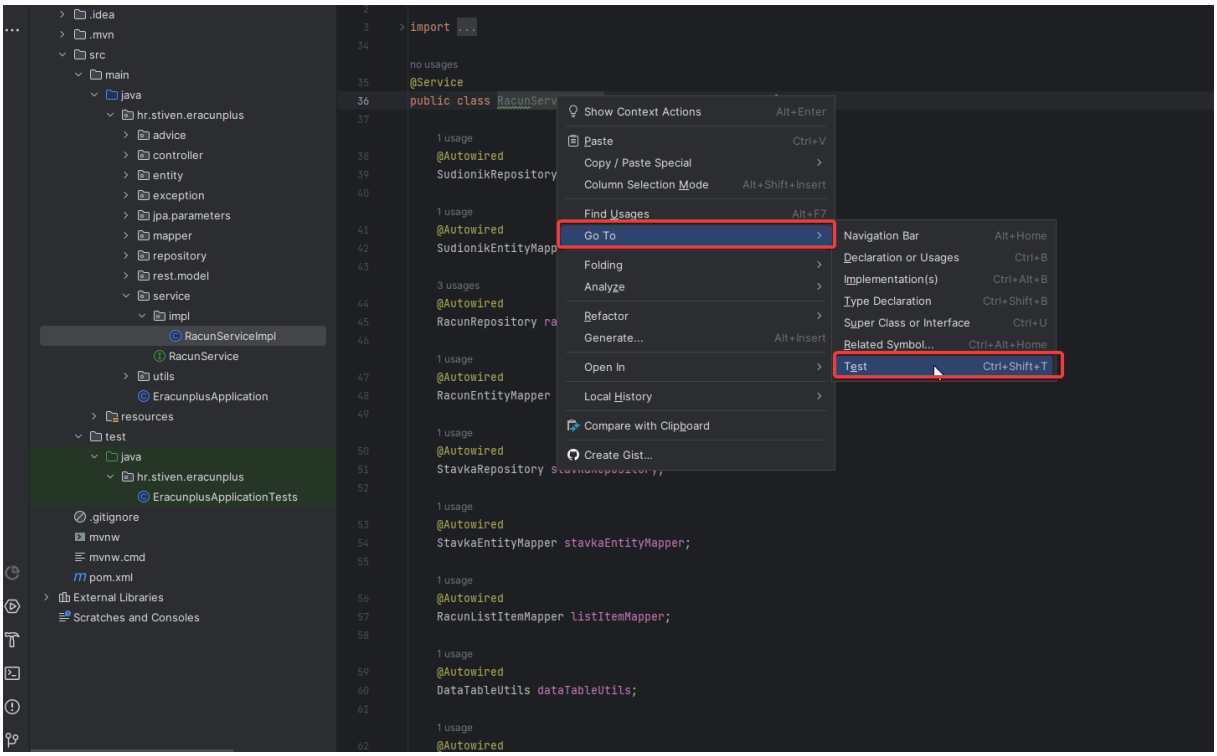
Sljedeći korak je umetanje ovisnosti Mockita i JUnita u pom.xml datoteku. To izgleda ovako.

```xml
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
```

Slika 17: Dodavanje ovisnosti u pom.xml datoteku [Autorski rad]

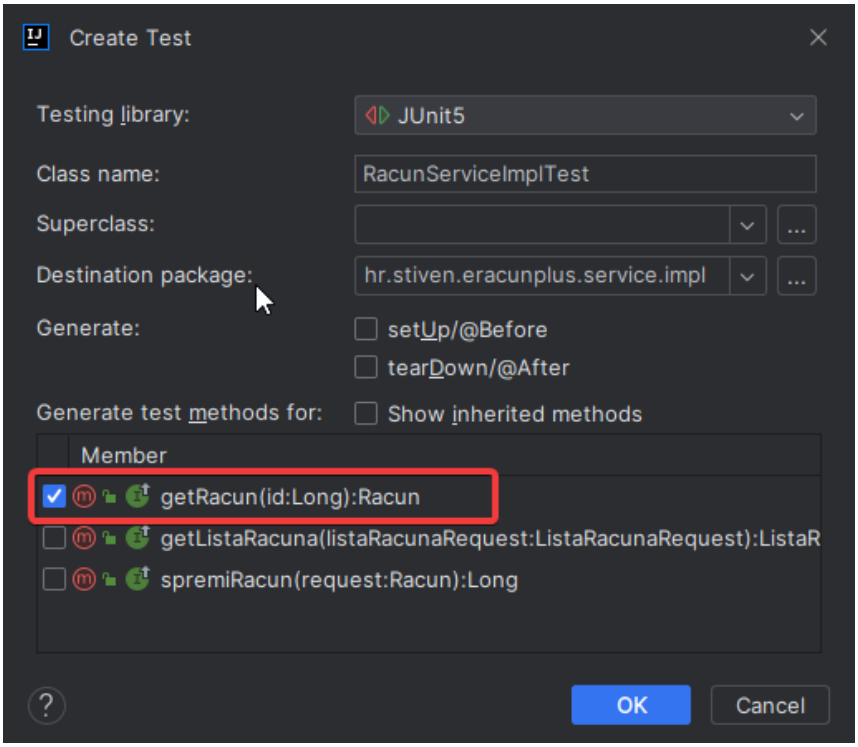
Kao i za Selenium, ovdje je također potrebno dodati Mockito .jar datoteku u projekt unutar dijela „Libraries“. Sada možemo započeti sa stvaranjem testnih klasa i njihovom implementacijom.

Testne klase radimo nad klasom „RacunServiceImpl“ u kojoj se nalaze metode „getRacun“, „getListaRacuna“, „spremiRacun“, „spremiStavke“, „spremiRacun“ te „spremiSudionika“. Način na koji kreiramo testnu klasu je da unutar klase „RacunServiceImpl“ pritisnemo desni klikom miša. U izborniku kliknemo na *„Go To“* > *„Test“*



Slika 18: Kreiranje testne klase [Autorski rad]

U ovom radu odabrana je metoda „getRacun“ nad kojom će se provoditi jedinična testiranja. Ovim korakom završava priprema i projekt je spreman za implementaciju jediničnih testiranja.



Slika 19: Odabir metoda koje će se testirati [Autorski rad]

6.3.1. Implementacija jediničnih testova

Testovi jedinica su napravljeni na sljedeći način. Potrebno je konfigurirati dvojnike objekta i definirati ih, zatim odrediti što je očekivani rezultat kada se pozove određena metoda iz objekta dvojnika. U konačnici potrebno je provjeriti da li je očekivani rezultat jednak stvarnom rezultatu.

6.3.2. Klasa „RacunServiceImplTest“

Unutar klase „RacunServiceImplTest“ implementirano je sljedeće:

- Test „getRacunByID“ – dohvaća se racun prema unesenom parametru.
- Test „getRacunNotFoundException“ – dohvaća se iznimka (eng. *Exception*) ako racun nije pronađen.

Kod implementacije izgleda ovako:

```
package hr.stiven.eracunplus.service.impl;
import hr.stiven.eracunplus.entity.RacunEntity;
import hr.stiven.eracunplus.entity.StavkaEntity;
import hr.stiven.eracunplus.entity.SudionikEntity;
import hr.stiven.eracunplus.exception.GloballyHandledException;
import hr.stiven.eracunplus.mapper.RacunMapper;
import hr.stiven.eracunplus.repository.RacunRepository;
import hr.stiven.eracunplus.rest.model.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
```

- Ovdje se nalaze biblioteke potrebne za provođenje testova.

```
@ExtendWith(MockitoExtension.class)
```

- Ovdje inicijaliziramo Mockito okvir za stvaranje i umetanje objekta dvojnika u JUnit 5 testnu klasu.

```
public class RacunServiceImplTest {
   private Long testIdRacuna;
   private Racun racun;
   private RacunEntity racunEntity;
   @Mock
   private RacunMapper racunMapper;
   @Mock
   private RacunRepository racunRepository;
```

- @Mock anotacijom Mockito stvara lažni objekt nad tom klasom. Ti objekti simuliraju stvarne objekte te nam daju kontrolu nad njima tijekom testiranja.

```
@InjectMocks
private RacunServiceImpl racunServiceImpl;
```

- @InjectMock anotacijom Mockito automatski stavlja objekte koji su označeni s @Mock anotacijom u klasu čija se metoda testira.

```
  @BeforeEach
```

- @BeforeEach anotacija označuje da se „setUp“ metoda pokreće prije svake od testnih metoda „getRacunByID“ i „racunNotFoundException“.

```
  public void setUp(){
        testIdRacuna = 1L;
        SudionikEntity sudionikPrimateljEntity = new SudionikEntity(1L,
"HEP", "OIB123", "Vilka Novaka 24", "Hrvatska", "42000", "Varazdin",
"Jozo", "mail.com", "0987654321", "PIB", "IBT", "RBT", "Gradevina");
        SudionikEntity sudionikIzdavateljEntity = new SudionikEntity(2L,
"FINA", "OIB2123", "Korutaska 2", "Hrvatska", "10000", "Zabok", "Ivo",
"fina.com", "123123123", "PIB2", "IBT2", "RBT2", "Banka");
        List<StavkaEntity> stavkaEntities = new ArrayList<>();
        racunEntity = new RacunEntity(testIdRacuna,
sudionikIzdavateljEntity, sudionikPrimateljEntity, "RB 1",
LocalDate.of(2023,2,10), LocalDate.now(), 2, "Nova roba", stavkaEntities,
2, "Kartica 12345", "Racun P", "HR IBAN 123", "swift", "Referenca",
"Napomena dodatak", "Nema zahtjeva", "Ostale informacije");
        Sudionik sudionikIzdavatelj = new Sudionik("FINA", "OIB2123",
"Korutaska 2", "Hrvatska", "10000", "Zabok", "Ivo", "fina.com",
"123123123", "PIB2", "IBT2", "RBT2", "Banka");
        Sudionik sudionikPrimatelj = new Sudionik("HEP", "OIB123", "Vilka
Novaka 24", "Hrvatska", "42000", "Varazdin", "Jozo", "mail.com",
"0987654321", "PIB", "IBT", "RBT", "Gradevina");
        List<Stavka> stavkaList = new ArrayList<>();
```

```
        racun = new Racun(sudionikIzdavatelj, sudionikPrimatelj, "Broj
Racuna 123", LocalDate.of(2023, 6,12), LocalDate.of(2023,8,21), 2, "Opis je
dodan", stavkaList, 1, "4848 1232 2222", "RacunP", "HR123123123", "Swift",
"Referenciranje", "Nema dodatne napomene", "Zahtjev", "Informacije" );
    }
```

- U metodi „setUp“ varijabli „testIdRacuna“ dali smo vrijednost 1, inicijalizirali smo i popunili podacima objekt „sudionikPrimateljEntity“ koji ima tip „SudionikEntity“, isto to smo napravili za objekt „sudionikIzdavateljEntity“, također tipa „SudionikEntity“. Uz to, napravili smo varijablu „stavkaEntities“ tipa „List<StavkeEntity>“. Ovi objekti su nam potrebni kako bi mogli kreirati objekt „racunEntity“. Za objekt „racun“ potrebni su objekt „sudionikIzdavatelj“ tipa „Sudionik“, objekt „sudionikPrimatelj“ također tipa „Sudionik“, uz to nam je potrebna i „stavkaList“ koja je varijabla tipa „List<Stavka>“. Nakon što objekt „racun“ popunimo ostalim potrebnim podacima krećemo s implementacijom testnih metoda.

```
    @Test
```

- @Test označava testnu metodu unutar koje se provodi testiranje. U ovom slučaju to su „getRacunById“ i „racunNotFoundException“.

```
public void getRacunByID() {
when(racunRepository.findById(testIdRacuna)).thenReturn(Optional.of(racunEn
tity));
   when(racunMapper.toRacun(racunEntity)).thenReturn(racun);
   assertEquals(racun, racunServiceImpl.getRacun(testIdRacuna));
}
```

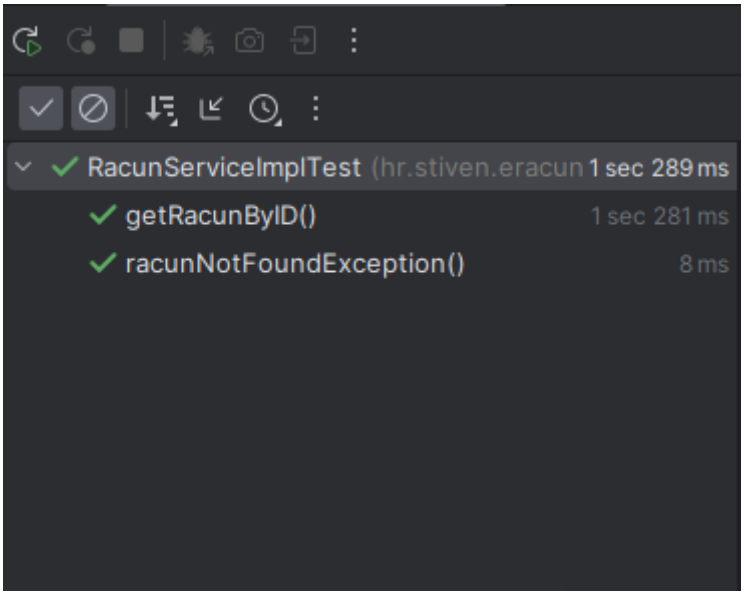
- U testnoj metodi „getRacunByID“ definirati ćemo ponašanje dvojnika tijekom testa when/thenReturn metodom. Navedeni kod radi sljedeće: nad objektom dvojnika „racunMapper“ pozivamo metodu „findById“ kojoj proslijedimo parametar „testIdRacuna“ s vrijednošću 1. Specifičnost when/thenReturn metode je da poziva metodu „findById“ i vraća zadanu vrijednost. U konačnici metoda „assertEquals“ provjerava da li se očekivana vrijednost i vrijednost pozvane stvarne metode podudaraju.

```
    @Test
public void racunNotFoundException() {
        Exception thrownException =
assertThrows(GloballyHandledException.class, () ->
racunServiceImpl.getRacun(testIdRacuna));
        assertEquals("Nije pronaden racun s dobivenim identifikatorom",
```

```
thrownException.getMessage());
    }
}
```

- U testnoj metodi „racunNotFoundException“ dohvaćamo grešku koja se javlja ako ne postoji racun sa zadanim id-em. Dakle ako je greška „GloballyHandledException“ dohvaćena, tada je test prošao. Naposlijetku provjeravamo ako je očekivana poruka jednaka poruci koji baca greška.

Svi testovi su uspješno prošli, što možemo vidjeti na slici 20.



Slika 20: Jedinični testovi su prošli [Autorski rad]

6.4. Integracijsko testiranje

Integracijska testiranja su također implementirana unutar IntelliJ programa uz pomoć Spring Boot-a koji nam uz pomoć @SpringBootTest anotacije omogućava da se testira cijela aplikacija. Ponovno je potrebno napraviti uvoz projekta u IntelliJ program i imati sljedeće ovisnosti.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Slika 21: Ovisnosti za Spring Boot [Autorski rad]

Zatim izrađujemo testnu klasu „RacunServiceIntegrationTest“ nad klasom „RacunServiceImpl“ i „SudionikServiceTest“ nad klasom „SudionikService“. Pošto smo taj korak napravili u dijelu jediničnih testova, sad prelazimo na implementaciju integracijskih testova.

6.4.1. Implementacija integracijskog testa

Napravljeni su integracijski testovi servisnog sloja aplikacije, konkretnije nad „RacunServiceImpl“ i „SudionikService“ klasama. Testira se ispravan dohvat računa slanjem zahtjeva do repozitorija „RacunRepository“ te da je dobiven očekivani rezultat. U drugom testu se ispituje dohvaća li se očekivani rezultat slanjem zahtjeva prema repozitoriju „SudionikRepository“.

6.4.2. Klasa „RacunServiceIntegrationTest“

Unutar testne klase implementirano je sljedeće:

- Test „getRacunById_forSavedRacun“ – dohvaća se spremljeni račun.

Kod implementacije je sljedeći:

```
package hr.stiven.eracunplus.service.impl;
import hr.stiven.eracunplus.entity.RacunEntity;
import hr.stiven.eracunplus.entity.StavkaEntity;
import hr.stiven.eracunplus.entity.SudionikEntity;
import hr.stiven.eracunplus.repository.RacunRepository;
import hr.stiven.eracunplus.rest.model.Racun;
import jakarta.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.boot.test.context.SpringBootTest;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.BDDAssertions.then;
```

- Potrebne biblioteke za provođenje integracijskih testova

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)
```

- Ova anotacija služi za sprječavanje podrazumijevanog pokretanja WebEnviroment okoline, te WebEnvironment postavljamo na NONE

```
@Transactional
```

- Ova anotacija služi za čišćenje podataka nakon izvođenja testa

```
public class RacunServiceImplIntegrationTest
{
    @Autowired
```

- @Autowired anotacija služi za automatizirano ubacivanje ovisnosti, tako da program sam odabire „bean“ ovisnosti. Dakle ova anotacija će pokušati pronaći odgovarajući „bean“ ove klase i dodijeliti ga varijabli „racunRepository“.

```
    private RacunRepository racunRepository;
    @Autowired
    private RacunServiceImpl racunServiceImpl;

   @Test
```

- Anotacija za provođenje testa

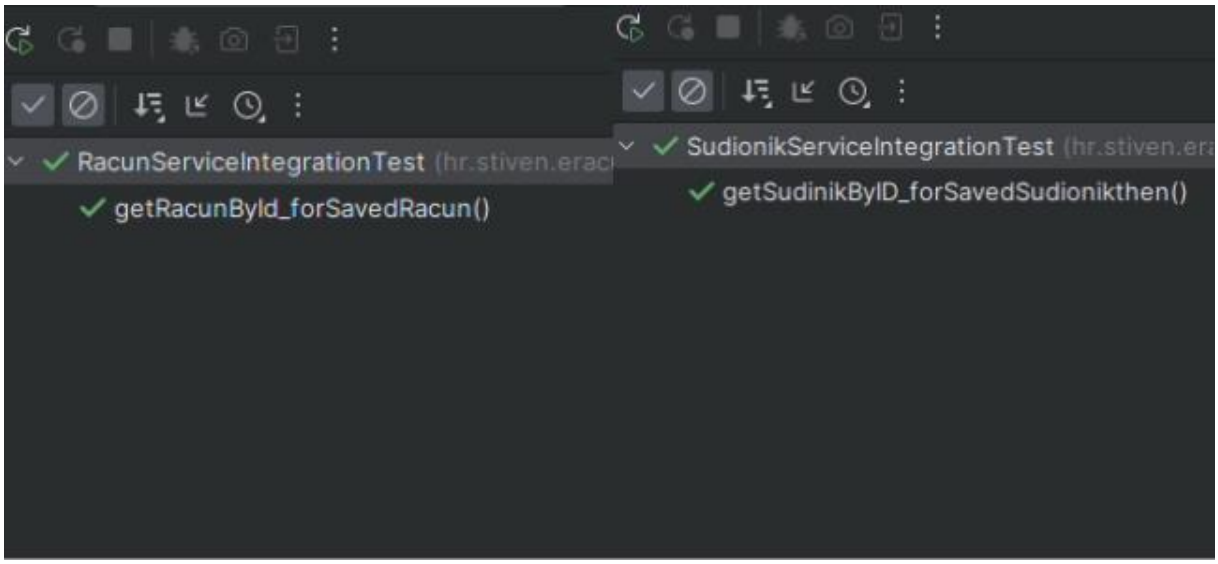
```
   void getRacunById_forSavedRacun()
   {
       Long id = 1L;
       SudionikEntity sudionikPrimateljEntity = new SudionikEntity(
1L,"HEP","OIB123","Vilka Novaka 24", "Hrvatska", "42000", "Varazdin",
"Jozo", "mail.com", "0987654321", "PIB", "IBT", "RBT", "Gradevina");
       SudionikEntity sudionikIzdavateljEntity = new SudionikEntity(2L,
"FINA", "OIB2123", "Korutaska 2", "Hrvatska", "10000", "Zabok", "Ivo",
"fina.com", "123123123", "PIB2", "IBT2", "RBT2", "Banka");
       List<StavkaEntity> stavkaEntities = new ArrayList<>();

       RacunEntity racunEntity = racunRepository.save(new RacunEntity(id,
sudionikIzdavateljEntity, sudionikPrimateljEntity, "RB 1",
LocalDate.of(2023,2,10), LocalDate.now(), 2, "Nova roba", stavkaEntities,
2, "Kartica 12345", "Racun P", "HR IBAN 123", "swift", "Referenca",
"Napomena dodatak", "Nema zahtjeva", "Ostale informacije"));
       Racun racun = racunServiceImpl.getRacun(racunEntity.getId());

       then(racunEntity.getId()).isNotNull();
       then(racun.brojRacuna()).isEqualTo("RB 1");
   }
}
```

- U testnoj metodi „getRacunByID_forSavedRacun“ inicijaliziramo objekt „sudionikPrimateljEntity“ i „sudionikIzdavateljEntity“ iz tipa „SudionikEntity“ te ih popunimo podacima. Zatim objekt „racunEntity“ sprema u „racunRepository“ pomoću metode „save()“. Tada u objekt „racun“ koji je tipa „Racun“ spremamo račun koji je dohvaćen preko metode „getRacun()“.

U konačnici oba testa su uspješna.



Slika 22: Uspješno izvedeni integracijski testovi [Autorski rad]

7. Zaključak

Tema ovog završnog rada bila je prikazati primjenu automatiziranog testiranja softvera prema piramidi testiranja koja uključuje jedinično testiranje, integracijsko testiranje i testiranje s kraja na kraj. U početku rada objašnjeno je zašto postoji potreba za testiranjem softvera te koji su principi kojih se držimo prilikom samog testiranja. Kroz rad spomenut je i životni ciklus testiranja softvera te je navedeno i pojašnjeno šest faza koje čine navedeni ciklus. Pojašnjeno je i automatizirano testiranje, navedeni su koraci od kojih se sastoji te prednosti i nedostaci ovakvog načina testiranja. Navedene su razine testiranja koja se provodi postoje, počevši od jedinične razine testiranja pa sve do testova prihvaćanja. U radu su spomenuti brojni alati za provođenje automatiziranog testiranja, podijeljeni u tri skupine: alati otvorenog koda, komercijalni alati i prilagođeni alati. Također, navedeni su najčešće korišteni alati za korištenje kod jediničnih testova, integracijskih testova i testova s kraja na kraj.

Praktični dio testiranja odrađen je nad aplikacijom pod nazivom e-RačunPlus, u kojoj se mogu kreirati i pretraživati e-računi. Jedinična testiranja odrađena su IntelliJ IDEA programu, koristeći JUnit 5 i okvir za testiranje Mockito. Unutar testnih klasa kreirajući lažne objekte korištene su metode klasa aplikacije, i u konačnici provjerili odgovaraju li očekivani podaci stvarnim dobivenim podacima. Integracijska testiranja provedena su također unutar IntelliJ alata, koristeći Spring Boot okvir u kojem se nalazi anotacija @SpringBootTest kojom se pokreće cijela aplikacija i njeni stvarni podaci. Tako je dio testiranja s kraja na kraj obuhvaćao Selenium testiranje korisničkog sučelja, koje je implementirano unutar aplikacije IntelliJ IDEA pomoću Selenium WebDrivera i GeckoDrivera, koji su pokrenuli aplikaciju u Mozilla Firefox pregledniku te je program sam popunjavao zadana polja, klikao na određeni gumb i mijenjao ekrane unutar web stranice. Testovi s kraja na kraj imali su najdulje vrijeme izvođenja, što zbog vremena koje je potrebno da se stranica i njezini elementi učitaju, ali također i vrijeme koje je potrebno da program popuni zadana polja, klikne na određeni gumb ili odabere dan i mjesec unutar kalendara.

Automatizirano testiranje softvera, zanemarivši početno postavljanje i implementaciju testova, uistinu olakšava proces testiranja. Razlog tome je nepotrebno ručno provjeravanje da li su ispravni. Jednom napisani automatski testovi mogu poslužiti u trenutnom, ali i u budućem projektu.

Popis literature

[1] „Digitalizacija u vrijeme koronavirusa: Kako je nemoguće postalo moguće“, *Apsolon*. https://apsolon.com/eu-fondovi-2021-2027/digitalizacija-u-vrijeme-koronavirusa/ (pristupljeno 02. rujan 2023.).

[2] „Manual Testing vs Automation Testing: Which One Should You Choose?“, *Testsigma Blog*, 16. studeni 2022. https://testsigma.com/blog/manual-testing-vs-automation-testing-which-one-should-you-choose/ (pristupljeno 02. rujan 2023.).

[3] „What is Software Testing and How Does it Work? | IBM“. https://www.ibm.com/topics/software-testing (pristupljeno 02. rujan 2023.).

[4] S. DESAI i A. SRIVASTAVA, *SOFTWARE TESTING : A Practical Approach*. Phi Learning, 2016. [Na internetu]. Dostupno na: https://books.google.hr/books?id=B4sQDAAAQBAJ

[5] „What is Software Testing? Definition“, 05. kolovoz 2023. https://www.guru99.com/software-testing-introduction-importance.html (pristupljeno 02. rujan 2023.).

[6] „Washingtonpost.com: Rocket Failures Shake Space Industry“. https://www.washingtonpost.com/wp-srv/national/daily/may99/rockets11.htm (pristupljeno 02. rujan 2023.).

[7] „7 Principles of Software Testing with Examples“, 08. srpanj 2023. https://www.guru99.com/software-testing-seven-principles.html (pristupljeno 02. rujan 2023.).

[8] „Software Testing Principles - javatpoint“. https://www.javatpoint.com/software-testing-principles (pristupljeno 02. rujan 2023.).

[9] „Software Testing Life Cycle (STLC)“, *GeeksforGeeks*, 10. svibanj 2019. https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/ (pristupljeno 02. rujan 2023.).

[10] „Software Testing Life Cycle - javatpoint“. https://www.javatpoint.com/software-testing-life-cycle (pristupljeno 02. rujan 2023.).

[11] „STLC (Software Testing Life Cycle) Phases, Entry, Exit Criteria“, 05. kolovoz 2023. https://www.guru99.com/software-testing-life-cycle.html (pristupljeno 02. rujan 2023.).

[12] „CTFL-2018-Syllabus.pdf“. https://astqb.org/assets/documents/CTFL-2018-Syllabus.pdf (pristupljeno 02. rujan 2023.).

[13] „Levels of Software Testing | Four Testing Levels Explained“, *Edureka*, 05. kolovoz 2019. https://www.edureka.co/blog/software-testing-levels/ (pristupljeno 02. rujan 2023.).

[14] T. Q. Lead, „18 Best Unit Testing Tools In 2023“, *The QA Lead*. https://theqalead.com/tools/best-unit-testing-tools/ (pristupljeno 02. rujan 2023.).

[15] „Unit Testing - javatpoint“. https://www.javatpoint.com/unit-testing (pristupljeno 02. rujan 2023.).

[16] „Integration Testing - javatpoint“. https://www.javatpoint.com/integration-testing (pristupljeno 02. rujan 2023.).

[17] „Integration Testing: What is, Types with Example“, 05. kolovoz 2023. https://www.guru99.com/integration-testing.html (pristupljeno 02. rujan 2023.).

[18] Administrator, „What is integration testing? Types, Process & Implementation“, *https://www.zaptest.com/*. https://www.zaptest.com/what-is-integration-testing-deep-dive-into-the-types-process-implementation (pristupljeno 02. rujan 2023.).

[19] C. Singureanu, „System Testing - Types, Process, Tools & More!“, *https://www.zaptest.com/*. https://www.zaptest.com/what-is-system-testing-types-tools (pristupljeno 02. rujan 2023.).

[20] „What is Acceptance Testing? (Importance, Types & Best Practices)“, *BrowserStack*. https://browserstack.wpengine.com/guide/acceptance-testing/ (pristupljeno 02. rujan 2023.).

[21] „Acceptance Testing | Software Testing“, *GeeksforGeeks*, 29. travanj 2019. https://www.geeksforgeeks.org/acceptance-testing-software-testing/ (pristupljeno 02. rujan 2023.).

[22] „What is Automation Testing: Benefits, Strategy, Tools | BrowserStack“. https://www.browserstack.com/guide/automation-testing-tutorial (pristupljeno 02. rujan 2023.).

[23] „What is Automation Testing? Test Tutorial“, 05. kolovoz 2023. https://www.guru99.com/automation-testing.html (pristupljeno 02. rujan 2023.).

[24] „automation-test-640x378.png (PNG slika, 640 × 378 piksela)“. https://browserstack.wpenginepowered.com/wp-content/uploads/2022/08/automation-test-640x378.png (pristupljeno 02. rujan 2023.).

[25] „Automation Testing Tutorial: Everything You Need to Know in 2021“, *UTOR*, 25. svibanj 2021. https://u-tor.com/topic/automation-testing-tutorial (pristupljeno 02. rujan 2023.).

[26] STF, „Automated Testing“, *Software Testing Fundamentals*, 30. rujan 2012. https://softwaretestingfundamentals.com/automated-testing/ (pristupljeno 02. rujan 2023.).

[27] „How to Select Best Automation Testing Tool?“, 12. kolovoz 2023. https://www.guru99.com/testing-automation-why-right-tools-are-necessary-for-testing-success.html (pristupljeno 02. rujan 2023.).

[28] „Open Source vs Commercial vs Custom Based Test Automation Suite“, *Qentelli*. https://www.qentelli.com/thought-leadership/insights/open-source-vs-commercial-vs-custom-based-test-automation-suite (pristupljeno 02. rujan 2023.).

[29] „Disadvantages of open source software | nibusinessinfo.co.uk“. https://www.nibusinessinfo.co.uk/content/disadvantages-open-source-software (pristupljeno 02. rujan 2023.).

[30] C. S. Dasagrandhi, „Open Source vs Commercial Test Automation Tools“. https://blog.vsoftconsulting.com/blog/open-source-vs-commercial-test-automation-tools (pristupljeno 02. rujan 2023.).

[31] „Top 15 Automation Testing Tools 2023 | Katalon“, *katalon.com*. https://katalon.com/resources-center/blog/automation-testing-tools (pristupljeno 02. rujan 2023.).

[32] „History“, *Selenium*. https://www.selenium.dev/history/ (pristupljeno 02. rujan 2023.).

[33] R. Gupta, „Selenium Testing Automation : Overview and History“, *Webomates*, 07. siječanj 2021. https://www.webomates.com/blog/software-testing/selenium-testing/ (pristupljeno 02. rujan 2023.).

[34] „What is Selenium? Introduction to Selenium Automation Testing“, 08. srpanj 2023. https://www.guru99.com/introduction-to-selenium.html (pristupljeno 02. rujan 2023.).

[35] „What is Selenium: Getting Started with Automation Testing“, *Simplilearn.com*. https://www.simplilearn.com/tutorials/selenium-tutorial/what-is-selenium (pristupljeno 02. rujan 2023.).

[36] M. Technologies, „▷ What Is Selenium RC : A Step-by-Step Guide for 2023“, *Mindmajix*, 22. travanj 2021. https://mindmajix.com/selenium/what-is-the-use-of-selenium-remote-control (pristupljeno 02. rujan 2023.).

[37] „What is Selenium RC : Difference from Webdriver“, *BrowserStack*. https://browserstack.wpengine.com/guide/selenium-rc-tutorial/ (pristupljeno 02. rujan 2023.).

[38] „Selenium Grid Tutorial : Learn Basics & How to Set It Up“, *BrowserStack*. https://browserstack.wpengine.com/guide/selenium-grid-tutorial/ (pristupljeno 02. rujan 2023.).

[39] „Selenium Webdriver Tutorial in Java with Examples“, *BrowserStack*. https://browserstack.wpengine.com/guide/selenium-webdriver-tutorial/ (pristupljeno 02. rujan 2023.).

[40] „Top 10 free open-source testing tools, Framework & Libraries“, *katalon.com*. https://katalon.com/resources-center/blog/open-source-testing-tools (pristupljeno 02. rujan 2023.).

[41] „The Good and the Bad of Katalon Studio Automation Testing Tool“, *AltexSoft*. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-katalon-studio-automation-testing-tool/ (pristupljeno 02. rujan 2023.).

[42] M. Technologies, „▷ Katalon Studio Tutorial | A Beginners Guide to Katalon Studio“, *Mindmajix*, 22. travanj 2021. https://mindmajix.com/katalon-studio-tutorial (pristupljeno 02. rujan 2023.).

[43] „Appium Documentation - Appium Documentation“. http://appium.io/docs/en/2.0/ (pristupljeno 02. rujan 2023.).

[44] „Appium Project History - Appium Documentation“. http://appium.io/docs/en/2.0/intro/history/ (pristupljeno 02. rujan 2023.).

[45] „APPIUM Tutorial for Android & iOS Mobile Apps Testing“, 15. srpanj 2023. https://www.guru99.com/introduction-to-appium.html (pristupljeno 02. rujan 2023.).

[46] M. Technologies, „TestComplete Tutorial for Beginners | Learn Software Testing“, *Mindmajix*, 22. travanj 2021. https://mindmajix.com/testcomplete-tutorial (pristupljeno 02. rujan 2023.).

[47] „Introduction to Test Complete | Components, Features & Benefits“. https://www.knowledgehut.com/blog/software-testing/testcomplete-introduction (pristupljeno 02. rujan 2023.).

[48] „Cypress vs Selenium: Key Differences“, *BrowserStack*. https://browserstack.wpengine.com/guide/cypress-vs-selenium/ (pristupljeno 02. rujan 2023.).

[49] „Why Cypress? | Cypress Documentation“. https://docs.cypress.io/guides/overview/why-cypress (pristupljeno 02. rujan 2023.).

[50] „Key Differences | Cypress Documentation“. https://docs.cypress.io/guides/overview/key-differences (pristupljeno 02. rujan 2023.).

[51] E. Kinsbruner, „What is Cypress Testing? What It Is and How to Get Started | Perfecto“. https://www.perfecto.io/blog/cypress-testing (pristupljeno 02. rujan 2023.).

[52] R. Kumar, „What is Junit and How it works? An Overview and Its Use Cases“, *DevOpsSchool.com*, 29. travanj 2022. https://www.devopsschool.com/blog/what-is-junit-and-how-it-works-an-overview-and-its-use-cases/ (pristupljeno 02. rujan 2023.).

[53] „JUnit Tutorial For Beginners - What Is JUnit Testing?“, *Software Testing Help*, 30. lipanj 2023. https://www.softwaretestinghelp.com/junit-tutorial/ (pristupljeno 02. rujan 2023.).

[54] „NUnit.org“. https://nunit.org/ (pristupljeno 02. rujan 2023.).

[55] „Selenium NUnit Tutorial: A Comprehensive Guide With Examples and Best Practices“. https://www.lambdatest.com/learning-hub/nunit-tutorial (pristupljeno 02. rujan 2023.).

[56]„NUnit vs xUnit vs MSTest: Which is better? Detailed Comparison“, *Testsigma Blog*, 19. prosinac 2022. https://testsigma.com/blog/nunit-vs-xunit/ (pristupljeno 02. rujan 2023.).
[57]„TestNG: A Comprehensive Guide | HeadSpin“. https://www.headspin.io/blog/testng-a-comprehensive-guide (pristupljeno 02. rujan 2023.).
[58]„Top 10 Integration Testing Tools in Software Testing“, *Testsigma Blog*, 23. lipanj 2023. https://testsigma.com/tools/integration-testing-tools/ (pristupljeno 02. rujan 2023.).
[59]„Citrus Framework“. http://citrusframework.org/docs/welcome/ (pristupljeno 02. rujan 2023.).
[60]„What is Tessy Tool in Software Testing | Software Testing Tool“, *Tutorials Link*. https://tutorialslink.com/Articles/What-is-Tessy-Tool-in-Software-Testing-Software-Testing-Tool/3246 (pristupljeno 02. rujan 2023.).
[61]„Software Testing - Integration Testing Tool“, *GeeksforGeeks*, 22. ožujak 2022. https://www.geeksforgeeks.org/software-testing-integration-testing-tool/ (pristupljeno 02. rujan 2023.).
[62]„TESSY - Test System - Razorcat Development GmbH“. https://www.razorcat.com/en/product-tessy.html (pristupljeno 02. rujan 2023.).
[63]R. Tasnim, „Top 10 Integration Testing Tools You Should Try In 2023“, *Software Testing Stuff*, 14. lipanj 2023. https://www.softwaretestingstuff.com/integration-testing-tools/ (pristupljeno 02. rujan 2023.).
[64]„Top 15 Best System Integration Testing Tools To Write Integration Tests“, *Vietnam Software Outsourcing - MOR Software*. https://morsoftware.com/blog/integration-testing-tools (pristupljeno 02. rujan 2023.).
[65]„Eclipse IDE Software Reviews, Demo & Pricing - 2023“. https://www.softwareadvice.com/ide/eclipse-ide-profile/ (pristupljeno 02. rujan 2023.).
[66]M. Heller, „What is Visual Studio Code? Microsoft’s extensible code editor“, *InfoWorld*, 08. srpanj 2022. https://www.infoworld.com/article/3666488/what-is-visual-studio-code-microsofts-extensible-code-editor.html (pristupljeno 02. rujan 2023.).
[67]„H2 Database - Introduction“. https://www.tutorialspoint.com/h2_database/h2_database_introduction.htm (pristupljeno 02. rujan 2023.).
[68]„GeckoDriver vs Marionette: Differences“, *BrowserStack*. https://browserstack.wpengine.com/guide/geckodriver-vs-marionette/ (pristupljeno 02. rujan 2023.).
[69]„SelectorsHub“. https://chrome.google.com/webstore/detail/selectorshub/ndgimibanhlabgdgjcpbbndiehljcpfh (pristupljeno 02. rujan 2023.).

[70]„Page Object Model and Page Factory in Selenium“, *BrowserStack*. https://browserstack.wpengine.com/guide/page-object-model-in-selenium/ (pristupljeno 02. rujan 2023.).

.

Popis slika

Popis tablica