Automatizacija testiranja mobilnih aplikacija

Fundak, Dora

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:112943

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni diplomski studij

AUTOMATIZACIJA TESTIRANJA MOBILNIH APLIKACIJA

Diplomski rad

Dora Fundak

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 19.09.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Dora Fundak
Studij, smjer:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. studenta, godina upisa:	D 1058, 19.09.2017.
OIB studenta:	03711649011
Mentor:	Doc.dr.sc. Emmanuel Karlo Nyarko
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva:	Doc.dr.sc. Damir Filko
Naslov diplomskog rada:	Automatizacija testiranja mobilnih aplikacija
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Opisati proces i načela testiranja mobilnih aplikacija. Na primjeru jedne aplikacije, napisane za iOS i Android uređaje, provesti odgovarajuće automatizirane napisane testove.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	19.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 26.09.2018.

Ime i prezime studenta:	Dora Fundak
Studij:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. studenta, godina upisa:	D 1058, 19.09.2017.
Ephorus podudaranje [%]:	2%

Ovom izjavom izjavljujem da je rad pod nazivom**: Automatizacija testiranja mobilnih aplikacija**

izrađen pod vodstvom mentora Doc.dr.sc. Emmanuel Karlo Nyarko

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1 UVOD 1

1.1 ZADATAK DIPLOMSKOG RADA 1

2 MOBILNI UREĐAJI 2

2.1 PAMETNI MOBITELI 2

2.2 OPERATIVNI SUSTAVI 4

2.2.1 Android 4

2.2.2 iOS 5

2.3 HARDVER 5

2.4 APLIKACIJE 6

2.4.1 Nativne aplikacije 7

2.4.2 Web aplikacije 7

2.4.3 Hibridne aplikacije 8

3 TESTIRANJE 9

3.1 KVALITETA 9

3.2 GREŠKE 10

3.3 PRINCIPI TESTIRANJA 11

3.4 DINAMIČKO I STATIČKO TESTIRANJE 12

3.5 RAZINE TESTIRANJA 14

3.5.1 Testiranje modula 16

3.5.2 Integracijsko testiranje 16

3.5.3 Testiranje sustava 17

3.5.4 Testiranje prihvatljivosti 17

3.6 VRSTE TESTOVA 18

3.6.1 Funkcijski testovi 18

3.6.2 Nefunkcijski testovi 18

3.7 RUČNO TESTIRANJE 19

4 AUTOMATIZACIJA TESTIRANJA 20

4.1 SELENIUM-WEBDRIVER 20

4.2 IZVRŠAVANJE TESTOVA 21

4.2.1 Selendroid 21

4.2.2 Ios-driver 22
4.2.3 Appium 22
4.3 PISANJE TESTOVA 24
4.3.1 JUnit 25
4.3.2 TestNG 25
4.4 FIZIČKI UREĐAJ, EMULATOR I SIMULATOR 25
5 TESTIRANJE MULTITASK APLIKACIJE 27
5.1 SPECIFIKACIJE APLIKACIJE 27
5.1.1 Ekran „Kalkulator“ 27
5.1.2 Ekran „Gorivo“ 29
5.1.3 Ekran „Bilješke“ 31
5.2 RAZVOJ APLIKACIJE 32
5.2.1 Android 32
5.2.2 iOS 34
5.3 ISTOVREMENO TESTIRANJE NA RAZLIČITIM PLATFORMAMA 37
5.3.1 Testovi 40
5.3.2 Paralelno testiranje 42
5.3.3 Izvješća testiranja 43
6 ZAKLJUČAK 44
LITERATURA 45
SAŽETAK 47
ABSTRACT 48
ŽIVOTOPIS 49
PRILOZI 50

1 UVOD

Mobilni uređaji su kroz posljednjih nekoliko godina postali sveprisutni. Od telefona i uređaja za zabavu nastali su uređaje sa širokim spektrom upotrebe. Upotrebljavaju se za komunikaciju, posao, zabavu i konzumiranje vijesti pa sve do medicinskih aplikacija, a među svim tim aplikacijama postoji velika konkurencija. Sve to je uveličalo važnost testiranja mobilnih aplikacija. Bitno je imati aplikaciju koja radi bez greške i bolje od konkurentskih aplikacija, a to se može osigurati jedino konstantnim testiranjem aplikacije. Konstantno testiranje se može izvoditi ručno, ali se sve više ide k automatizaciji testiranja gdje se sastavljaju testovi koji se konstantno izvršavaju tijekom svih faza razvojnog procesa aplikacije i osiguravaju njen rad bez grešaka.

1.1 Zadatak diplomskog rada

Zadatak ovog diplomskog rada je opisati proces i načela testiranja mobilnih aplikacija. Nakon toga će se prikazati, na primjeru napravljene mobilne aplikacije za Android i iOS uređaje, automatizacija testiranja.

Drugo poglavlje ovog rada opisuje mobilne uređaje, različite operativne sustave kao i vrste aplikacija koje postoje na njima. Treće se poglavlje bavi testiranjem. Odgovara na pitanja zašto je testiranje bitno, kako se izvodi i navodi različite vrste testiranja. U četvrtom je poglavlju pokrivena automatizacija testiranja. Navode se različite tehnologije i načini automatiziranja testiranja. Peto poglavlje sadrži proces pisanja aplikacija i testova, opis napisanih aplikacija i testova te demonstraciju automatiziranog testiranja.

2 MOBILNI UREĐAJI

Prije nego što se obradi tema testiranja aplikacija, potrebno je objasniti današnje mobilne uređaje, tj. pametne mobitele i njihovu važnost, različite dostupne operativne sustave i aplikacije. Prilikom testiranja aplikacije potrebno je znati kako funkcioniraju različiti operativni sustavi i aplikacije na njima kako bi se što učinkovitije provelo testiranje.

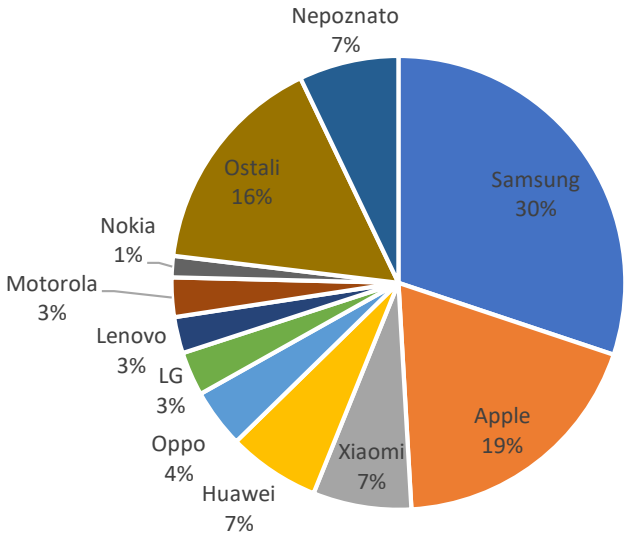
2.1 Pametni mobiteli

Prvi mobitel koji bi se mogao smatrati pametnim bio je IBM Simon iz 1994. godine [1]. Imao je ekran na dodir, mogućnost slanja i primanja email-ova, navigaciju i vijesti. Pametni mobiteli kakvi su danas prisutni počeli su se razvijati 2007. godine kada je Apple predstavio prvi iPhone.

Pametni mobiteli su deset godina stari, a svake godine postaju sve više integrirani u ljudske živote. Napreduju, razvijaju se, postaju sve raznovrsniji i nude sve više funkcija. Mobilni internet je sve brži i omogućuje brži protok sadržaja. Mobitel vibrira, svira i svijetli kako bi privukao našu pozornost, a sadržaj se konstantno mijenja i potiče korisnike na stalnu interakciju. Istraživanja su pokazala kako samo prisustvo mobitela smanjuje kognitivni kapacitet, a drugo istraživanje je pokazalo kako velika većina ispitanika doživljava fantomsko vibriranje – iluzija da je mobitel vibrirao kad zapravo nije [2].

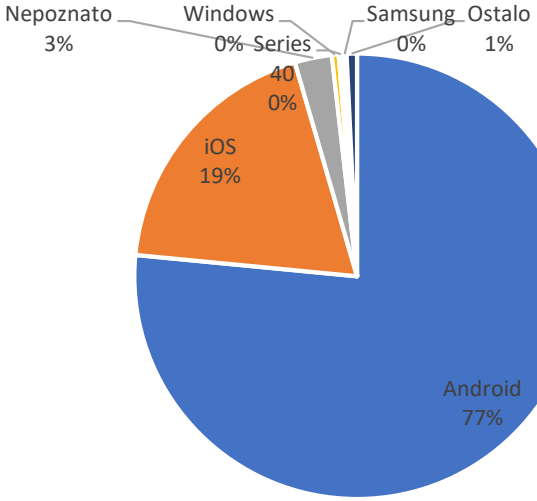
Prema istraživanju [2], 34% odraslih u UK-u pogleda mobitel u vremenu od 5 minuta nakon buđenja, a 55% u roku od 15 minuta. Pametni mobiteli su u 5 godina pretekli laptope po postotku posjedovanja u UK-u. 2012. godine laptop je posjedovalo 73% stanovništva, a pametni mobitel 52%. 2017. godine laptop je došao na 78%, a pametni mobitel na 85%, što je rast od 33% u pet godina. Sve to potvrđuje kako su mobiteli postali neizostavan dio ljudskog života. Mobilni su uređaji postali neizostavan dio modernog života do te mjere da su ljudi koji ih ne posjeduju djelomično izostavljeni iz djela modernih ekonomskih i socijalnih aktivnosti. U razvijenim zemljama 91% ljudi posjeduje bilo kakav mobitel, a 80% pametni mobitel. Oko 20% korisnika pogleda mobitel više od 50 puta na dan, što je otprilike svakih 20 minuta budnih sati [3].

Skoro je 384 milijuna pametnih mobitela prodano u prvom kvartalu 2018. godine. Potražnja za najnovijim high-end mobitelima opada jer kupovina takvog mobitela donosi minimalno unaprjeđenje u odnosu na malo starije modele, a potražnja za jeftinijim pametnim telefonima raste jer oni postaju sve kvalitetniji. Udio različitih proizvođača može se vidjeti na grafu 2.1.



Graf 2.1. Zastupljenost proizvođača mobilnih uređaja [4]

Najrasprostranjeniji su Samsung mobilni uređaji koje slijedi Apple, pa Huawei i Xiaomi [4]. Samsung, Xiaomi i Huawei kao temelj operativnog sustava upotrebljavaju Android. Iz te se podjele može doći do zaključka da su Android i iOS najrasprostranjeniji operativni sustavi što je prikazano na grafu 2.2.



Graf 2.2. Zastupljenost operativnog sustava na mobilnim uređajima [4]

Iz grafa se vidi kako su iOS i Android najraširenij sustavi na mobilnim uređajima i zajedno čine preko 95% uređaja u svijetu. Postoji jako puno mobilnih uređaja s različitim operativnim sustavima i njihovim verzijama pa zbog toga testiranje postaje komplicirano. Aplikacija se može ponašati drugačije na različitim uređajima, ali se to sve ne može testirati. Zato je bitno uzeti u obzir statistike o tome koje uređaje rabi najviše korisnika aplikacije i na njima provesti testiranja.

2.2 Operativni sustavi

Najrasprostranjeniji operativni sustavi su Android i iOS pa će se u ovome radu koncentrirati na testiranje aplikacija na ta dva operativna sustava. U tablici 2.1. prikazane su sličnosti i razlike između ta dva operativna sustava i njihove karakteristike.

Tablica 2.1. Karakteristike operativnih sustava [5].

Specifikacija	Android	iOS
Kompanija	Google	Apple
OS obitelj	Linux	OS X, Linux
Napisan u	C, C++	C, C++
Aplikacije se pišu u	Java	Objective-C, Swift
Proizvođači uređaja	Samsung, LG, Huawei, Xiaomi, Lenovo…	Apple
Mogućnost preinaka	da	ne
Ekran	ekran na dodir	ekran na dodir
Glasovne naredbe	Google now	Siri
Trgovina aplikacijama	Google Play, Amazon, …	App store

Može se vidjeti da platforme imaju slične, ako ne i jednake, funkcionalnosti (prepoznavanje glasa, video pozivi, mape, e-mail, kalendar…), ali u svojoj jezgri su drugačiji.

2.2.1 Android

Android je besplatan open-source operativni sustav za mobilne uređaje zasnovan na Linux kernel-u, a od 2005. godine je u vlasništvu Google-a. Google definira Android kao prvu potpuno otvorenu platformu za uređaje. Smatraju da platforma sadrži sav potreban softver bez zapreka za inovaciju. Zbog toga što je open-source, Android upotrebljava većina proizvođača mobilnih uređaja kao bazni OS. Posljednja verzija Android operativnog sustava izašla je sredinom 2017. godine i naziva se Android Oreo 8.0, a kasnije je izašla i nadogradnja 8.1. Iako je to posljednja

verzija sustava, ona nije među najzastupljenijima. Android ima problem jer podržava jako puno različitih uređaja, a ti uređaji ne mogu ići u korak s razvojem operativnog sustava. Zbog toga je trenutno najrasprostranjeniji Nougat 7.1 i 7.0, nakon njega Marshmallow 6.0 [6].

Aplikacije za Android pišu se u Javi, kompajliraju u Java bajt kod, a nakon toga u Dalvik ili ART bajt kod. Dalvik upotrebljava točno na vrijeme (engl. *just in time*) kompajliranje, što znači da se kod kompajlira netom prije izvršavanja zbog čega upotrebljava manje resursa. ART upotrebljava prijevremeno kompajliranje (engl. *ahead of time*), što znači da se aplikacija kompajlira samo jednom prilikom instalacije. ART-om se smanjuje sporost pokretanja aplikacije. Sav pristup hardveru i nekim sistemski funkcionalnostima ide preko ART/Dalvik-a. Time se osigurava da se programeri aplikacija ne moraju brinuti o hardveru [7].

2.2.2 iOS

Drugi najpopularniji mobilni operativni sustav, iOS, razvio je Apple 2007. godine. Sustav je u potpunosti zatvoren i mogu ga upotrebljavati samo Apple proizvodi. Kako bi programer mogao pisati aplikacije za iOS, uređaje on mora posjedovati računalo s macOS-om jer je razvojna okolina (Xcode) dostupna samo za korisnike Apple proizvoda. Aplikacije za iOS pišu se u Objectiv-C-u ili u Swift-u. Swift se kompajlira u strojni kod i najbolje funkcionira kad se upotrebljava prijevremeno kompajliranje. Operativni sustav iOS se konstantno unapređuje i izdaju se nove verzije. Trenutno je posljednja verzija iOS 11.4. Za razliku od Androida, posljednja verzija iOS-a je i najrasprostranjenija. Razlog tome je što Apple podržava samo svoje proizvode i rade kompatibilnost prema starijim verzijama za nove verzije operativnog sustava tako da ih i stariji uređaji mogu rabiti [8].

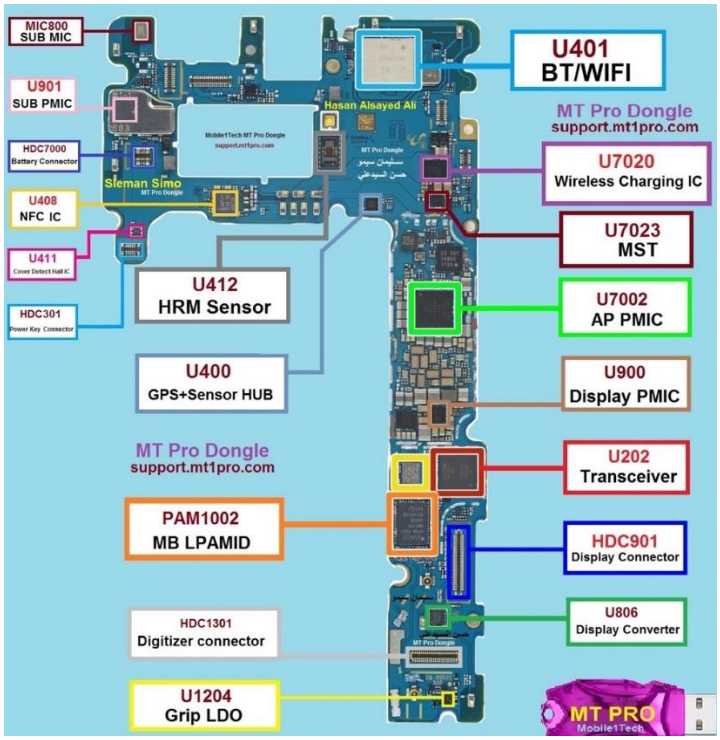
2.3 Hardver

Pametni su mobiteli u svojoj srži minijaturna računala. Sadrže jednake osnovne dijelove kao oni, na primjer memoriju i procesor. Razlika je u tome što se u mobitelima nalazi puno senzora [5] (npr. za osvjetljenje, udaljenost, ubrzanje, nagib…), uglavnom imaju dvije kamere, različite tehnologije za povezivanje (NFC, bluetooth, GPS, WiFi, podatkovni promet…) i, ovisno o proizvođaču, različite opcije za povezivanje (audio jack, mini USB, USB type-C, lightning…).

Najveći dio pametnog mobitela je ekran na dodir. Postoje dvije vrste ekrana na dodir. Prva je rezistivni ekran koji se sastoji od dva sloja konduktivnog materijala između kojih je praznina koja se ponaša kao otpornik. Kada se ekran dodirne, ta se dva sloja dodirnu i na tom se mjestu provodi struja. Druga vrsta je kapacitivni ekran koji se sastoji od sloja stakla presvučenim tankim

slojem konduktora. S obzirom na to da je i ljudsko tijelo konduktor, dodirivanje ekrana iskrivljuje elektrostatičkog polja koji se mjeri kao kapacitet [9].

Tipični se pametni mobitel sastoji od : matične ploče, CPU, GPU, memorije, različite antene za različite tehnologije, senzori, baterija itd., a to je prikazano na slici 2.1.



Slika 2.1. Arhitektura mobilnog uređaja [31]

Prilikom testiranja bitno je znati na kojem se hardveru testira jer o njemu ovisi i ponašanje aplikacije. Uglavnom su aplikacije za iOS i Android drugačije napravljene iako izgledaju kao ista aplikacija.

2.4 Aplikacije

Nakon hardvera, razlog zbog kojega su aplikacije na uređajima drugačije je to što obje platforme imaju smjernice za dizajn koje se razlikuju. Svaka platforma ima drugačiji preporučeni izgled aplikacije, tijek aplikacije i način rada. Te se smjernice moraju poštovati kako bi aplikaciju bilo moguće prodavati u trgovini sustava. Bitno je pratiti smjernice jer su korisnici naviknuti na određeni „osjećaj“, pokrete i naredbe u aplikaciji, a kao što je rečeno korisnik će vrlo brzo obrisati aplikaciju ako se u njoj ne snalazi.

Zbog naglog razvoja uređaja i njihove sveprisutnosti korisnici su postali izbirljivi. Postoji bezbroj različitih aplikacija za jednu istu funkciju. Ako se u App store tražilicu na iPhone-u upiše „kalendar“ dobit će se bezbroj aplikacija koje obavljaju tu funkciju; isto vrijedi i za Play Store na

Androidu. Prema [10] 52% korisnika obriše mobilnu aplikaciju ili napusti web stranicu ako prilikom prvog korištenja naiđe na problem ili se ne mogu snaći. 44% korisnika upotrebljava manje od 5 različitih aplikacija dnevno. Korisnicima je najbitnije kako aplikacija radi (bitno je brzo učitavanje), je li dostupna i koliko je intuitivna. Zbog toga je bitno testirati aplikaciju i pobrinuti se da radi kako je očekivano i da je kvalitetna jer je izbor aplikacija beskonačan.

Kako postoje različiti uređaji i različiti operativni sustavi za njih, postoje i različiti tipovi aplikacija. Aplikacija može biti nativna, hibridna ili web. Po samom izgledu aplikacije i njenom ponašanju ne može se znati je li ona nativna ili hibridna, a web aplikaciju se može prepoznati po tome što se prilikom njenog pokretanja otvara web pretraživač.

2.4.1 Nativne aplikacije

Nativne su aplikacije programirane u jeziku koji je specifičan za mobilni operativni sustav. To znači Java za Android i Swift ili Objectiv-C za iOS. One mogu upotrebljavati sve biblioteke i API-je specifične za sustav čime im se omogućuje pristup svim dostupnim resursima sustava i uređajima. Zbog toga su aplikacije brže jer mogu upotrebljavati i donekle upravljati resursima CPU-a i GPU-a direktno. S obzirom da je nativna aplikacija pravljena za točno određeni sustav, korisnici mogu rabiti sve standardne pokrete specifične za taj sustav, a i aplikacija je napravljena po smjernicama za dizajn tog sustava. Takva aplikacija može raditi i kad uređaj nije spojen na Internet [5].

Negativna strana nativnih aplikacija je to što se takva aplikacija može upotrebljavati samo na tom specifičnom sustavu kojemu je aplikacija nativna. Potrebno je praviti potpuno novu aplikaciju za drugi sustav.

2.4.2 Web aplikacije

Mobilna web aplikacija je zapravo web stranica kojoj je pristupa preko pretraživača na mobitelu. Razvijene su upotrebom JavaScript-a, HTML-a (najčešće HTML5) i CSS-a. Takve aplikacije nisu pohranjene na mobitelu i može im se pristupiti samo preko interneta, što znači da nema instalacije i aplikacija se može jednostavno ažurirati. One ne mogu pristupiti većini hardvera pa ne mogu upotrebljavati kameru ni senzore, ali mogu dobiti GPS lokaciju ili davati obavijesti. Takve aplikacije su responzivne, mogu prikazivati videozapise i slike, ali nemaju sve moguće funkcionalnosti koje nativne aplikacije imaju. One se brzo i jednostavno razvijaju i mogu se pokretati na svim sustavima [11].

Problem se može pojaviti kod web pretraživača jer različiti pretraživači mogu prikazivati aplikaciju drugačije pa je zbog toga potrebno testirati aplikaciju na različitim pretraživačima koje

korisnici rabe za pristup aplikaciji. Ako korisnik ima sporu internetsku vezu, aplikaciji će trebati više vremena kako bi se učitala jer se sav sadržaj prvo treba skinuti s interneta na mobitel kako bi se mogao prikazati.

2.4.3 Hibridne aplikacije

Hibridne su aplikacije kombinacija nativne i web aplikacije. Razvija se tako da se web aplikacija omota nativnom aplikacijom kako bi korisnici dobili osjećaj da upotrebljavaju nativnu aplikaciju, a ne web stranicu. Takve se aplikacije mogu upotrebljavati na Androidu, iOS-u i Windows-u, a omogućuju korištenje hardvera uređaja koji inače nije dostupan web aplikacijama.

Nedostatci takvih aplikaciju su njihova sporost jer su ovisne o serveru i jer nisu pisane za točno određenu arhitekturu. Problem je i što se ne mogu jedinstvenim dizajnom pokriti smjernice svih sustava [5].

3 TESTIRANJE

Ljudi su stalno u interakciji sa softverom. Rabe ga u svakodnevnom životu, od bankarstva i automobila do pametnih hladnjaka. Neki od problema s kojima se ljudi redovno susreću su greške prilikom plaćanja karticom, sporo učitavanje web stranice, neke opcije aplikacije koji ne radi kako bi trebale raditi itd. Većina tih problema su softverski problemi.

Nemaju svi sustavi jednaku važnost i greške nemaju jednak utjecaj na ljude. Neki problemi su trivijalni, dok neki mogu koštati novaca i reputacije, a mogu rezultirati ozljedom ili smrću [12]. Na primjer, može doći do pogreške prilikom prikaza podataka iz baze. Ako neka web stranica krivo prikazuje opis tvrtke, ne može se ništa loše dogoditi, jedino mogu izgledati neprofesionalno. Greška prikaza podatka kod medicinske aplikacije, koja nadzire razinu šećera u krvi i predlaže dozu inzulina pacijentima, je jako rizična i bitno je da do pogreške nikada ne dođe.

Testiranje je potrebno jer svi rade pogreške, a testira se kako bi aplikacija radila bez pogrešaka za što više korisnika. Neke pogreške mogu biti neprimjetne, ali neke mogu biti skupe i opasne. Svaki se proizvod treba provjeravati i testirati tijekom razvoja jer uvijek nešto može poći po krivu. Testiranje aplikacije je potrebno izvoditi tijekom razvoja aplikacije kako bi se greške što prije otkrile i kako bi se smanjila mogućnost pogrešaka kod završene aplikacije. Testirati se može sve u aplikaciji, od korisničkog sučelja do tijeka aplikacije [12].

Testiranje se može vršiti i kako bi se zadovoljili kriteriji regulatornog tijela ili zadani standardi. Na primjer, *US Food and Drug Administration* (FDA) u Sjedinjenim Američkim Državama nadzire medicinske aplikacije i stavlja pred njih različite zahtjeve koje aplikacija mora zadovoljiti kako bi bila odobrena od njihove strane. Korisnicima u Americi je bitno da su proizvodi odobreni od strane te regulatorne agencije pa ako se želi poboljšati šansu uspjeha nekog proizvoda na tom tržištu, bitno je dobiti odobrenje od FDA-a.

3.1 Kvaliteta

Testiranje je proces analize programskog koda s ciljem otkrivanja informacija o ispravnosti i kvaliteti. Osigurava pouzdanost, ispravnost i otkrivanje pogrešaka. Aplikacija je kvalitetna ako je napravljena po dobro definiranim specifikacijama koje zadovoljava. Iz toga proizlaze verifikacija i validacija aplikacije. Verifikacija se odnosi na postupak provjere ponaša li se program prema specifikacijama i regulacijama te odgovara na pitanje „radimo li proizvod dobro?“. Validacija je postupak koji provjerava zadovoljava li program zahtjeve i odgovara na pitanje „radimo li dobar proizvod?“ [13].

Međunarodni standard ISO/IEC 25010:2011 [14] za kvalitetu softvera definira kvalitetu kao „stupanj u kojem skup bitnih značajki zadovoljava zahtjeve“. Definiraju šest karakteristika kvalitete softvera:

1. Funkcionalnost

Zadovoljavanje specifikacija i standarda, siguran i prikladan za upotrebu.

2. Pouzdanost

Mogućnost rada u određenim uvjetima kroz neki vremenski period. Podrazumijeva i mogućnost oporavka sustava ako dođe do greške.

3. Prikladnost za upotrebu

Jednostavan za shvatiti i upotrebljavati, privlačan korisnicima.

4. Učinkovitost

Softver se treba ponašati prikladno resursima koje troši.

5. Održivost

Softver treba biti moguće prepraviti, održavati, testirati i analizirati.

6. Prenosivost

Softver se mora moći upotrebljavati u različitim okruženjima.

3.2 Greške

Greške (engl. *bugs*), defekti ili mane su neki od naziva za probleme koji se pojavljuju u aplikaciji. Zatajenje aplikacije se događa kad aplikacija nije u skladu sa zadanim specifikacijama ili kad specifikacije ne opisuju dobro njenu funkciju. Svaka aplikacija ima zadane specifikacije koje treba zadovoljiti. Specifikacije definiraju kako se aplikacija treba ponašati, što će raditi i što neće raditi. Greška prema [15] nastaje kad:

1. aplikacija ne radi što specifikacije kažu da bi trebala raditi
2. aplikacija radi ono što specifikacije kažu da ne bi trebala raditi
3. aplikacija radi nešto što nije navedeno u specifikacijama
4. aplikacija ne radi nešto što nije navedeno u specifikacijama, a trebalo bi biti
5. aplikacija je spora, nepregledna ili ju je teško rabiti

Greške se mogu rangirati prema ozbiljnosti i prioritetu. Prema ozbiljnosti mogu biti: kritične, značajne, niže ili kozmetičke. Kritične greške onemogućuju rad aplikacije u potpunosti i bitno ih je što prije ispraviti. Značajne greške označavaju nerad neke funkcionalnosti aplikacije, ali se može zaobići greška koja se dobije. Niže greške su one koje ne utječu na rad aplikacije, a kozmetičke greške su odudaranje od zadanog dizajna [16].

3.3 Principi testiranja

Prema [12] i [15], postoje principi testiranja koji su sastavljeni tijekom proteklih pola stoljeća testiranja i smatraju se generalnim smjernicama prilikom svih vrsta testiranja.

1. Testiranje dokazuje prisutnost grešaka

Testiranje dokazuje prisutnost grešaka, ali ne može dokazati da greške ne postoje. Testiranje smanjuje vjerojatnost postojanja neotkrivenih grešaka, ali ako greške nisu pronađene, to ne znači da one ne postoje.

2. Nemoguće je testirati cijelu aplikaciju

Nemoguće je testirati sve moguće unose, prethodna stanja i rezultate. Umjesto sveobuhvatnog testiranja, radi se testiranje bitnih dijelova aplikacije i čestih akcija koji utječu na doživljaj korisnika. Redundantni i nepotrebni testovi se zanemaruju kako bi se uštedilo vrijeme.

3. Rano testiranje

Potrebno je što ranije tijekom razvojnog procesa početi s testiranjem aplikacije kako bi se odmah pronašle i ispravile greške.

4. Imunizacija aplikacije

Ako se konstantno izvršava isti set testova, taj set testova neće pronaći nove greške jer će one biti ispravljene. Kako bi se izbjeglo nepronalaženje novih grešaka, testove je potrebno evaluirati i sukladno evaluaciji promijeniti. Potrebno ih je usmjeriti na neke druge dijelove aplikacije.

5. Testiranje ovisi o kontekstu

Testiranju se pristupa ovisno o kontekstu. Različite vrste aplikacije se testiraju na drugačije načine.

6. Zabluda o nepostojanju grešaka

Pronalazak i popravljanje grešaka ne znači da će korisnici prihvatiti aplikaciju. Broj grešaka nije najbitnije obilježje aplikacije. Ako je ona nestabilna, spora i ne zadovoljava korisnikove zahtjeve, neće biti uspješna.

Kao dodatni principi koje [15] navodi mogu se spomenuti i:

7. Što se više grešaka pronađe, to više grešaka postoji

Ako se prilikom testiranja pronađe greška, postoji velika vjerojatnost da postoji još nekoliko takvih pogrešaka ili pogrešaka oko te pronađene. Razlog tome je što programeri ponavljaju pogreške ili što pronađena pogreška može biti samo nagovještaj većih problema u pozadini aplikacije.

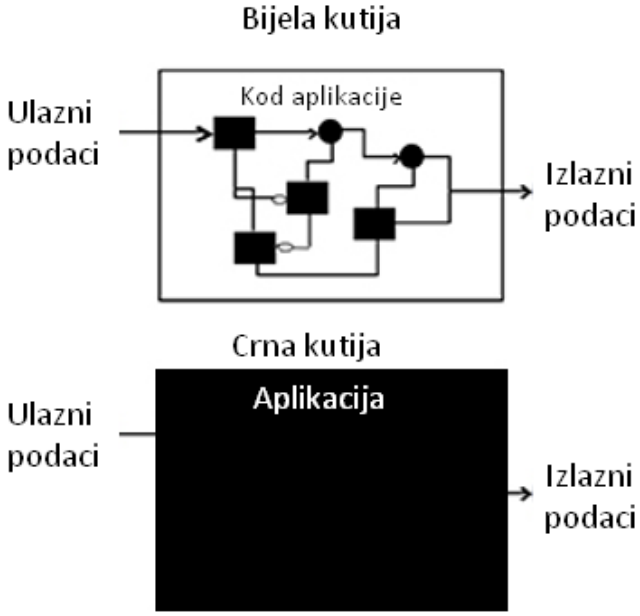
8. Neće sve greške biti ispravljene

Zbog nedostatka vremena, neisplativosti ispravka ili iz razloga što pronađena greška zapravo nije greška neće sve greške biti ispravljene.

3.4 Dinamičko i statičko testiranje

Testiranje aplikacije može se podijeliti u dvije skupine, dinamičko i statičko testiranje. Kod statičkog testiranja se ne izvršava aplikacija nego se gleda sam kod. Testeri i programeri bi trebali tijekom razvoja aplikacije raditi evaluaciju koda (engl. *code review*), odnosno netko tko nije napisao kod bi trebao pogledati taj kod i odlučiti prati li on smjernice kodiranja i je li on prihvatljiv. Druga vrsta evaluacije se može raditi prije razvoja aplikacije kontroliranjem zadanih specifikacija kako se tijekom razvoja ne bi morale mijenjati specifikacije [12]. Tijekom statičke faze, kod se može provjeravati pomoću alata kako bi se osiguralo da prati zadane smjernice formata kodiranja [5].

Kod dinamičkog se testiranja kod izvršava i nadzire se rad aplikacije. Dinamičko se testiranje može podijeliti u dvije skupine testiranja: bijela kutija (engl. *white box*) i crna kutija (engl. *black box*) testiranje [5] prikazanih na slici 3.1.



Slika 3.1 Bijela i crna kutija načini testiranja

Bijela kutija testiranje opisuje testiranje prilikom kojega se poznaje unutarnja struktura metoda ili klasa. Cilj takvog testiranja je provjeriti moguće putanje, uvjete i petlje kako bi se aplikacija osigurala od rušenja, beskonačnih petlji, nedostatka memorije, kako bi se otkrili neki sigurnosni nedostaci itd. Takvo testiranje uglavnom vrše sami programeri prilikom pisanja koda i ako pronađu greške, odmah ih ispravljaju. Naziva se i testiranje vođeno logikom (engl. *logic-driven*) jer se testni scenariji izvode iz logike aplikacije i njenog ponašanja [17].

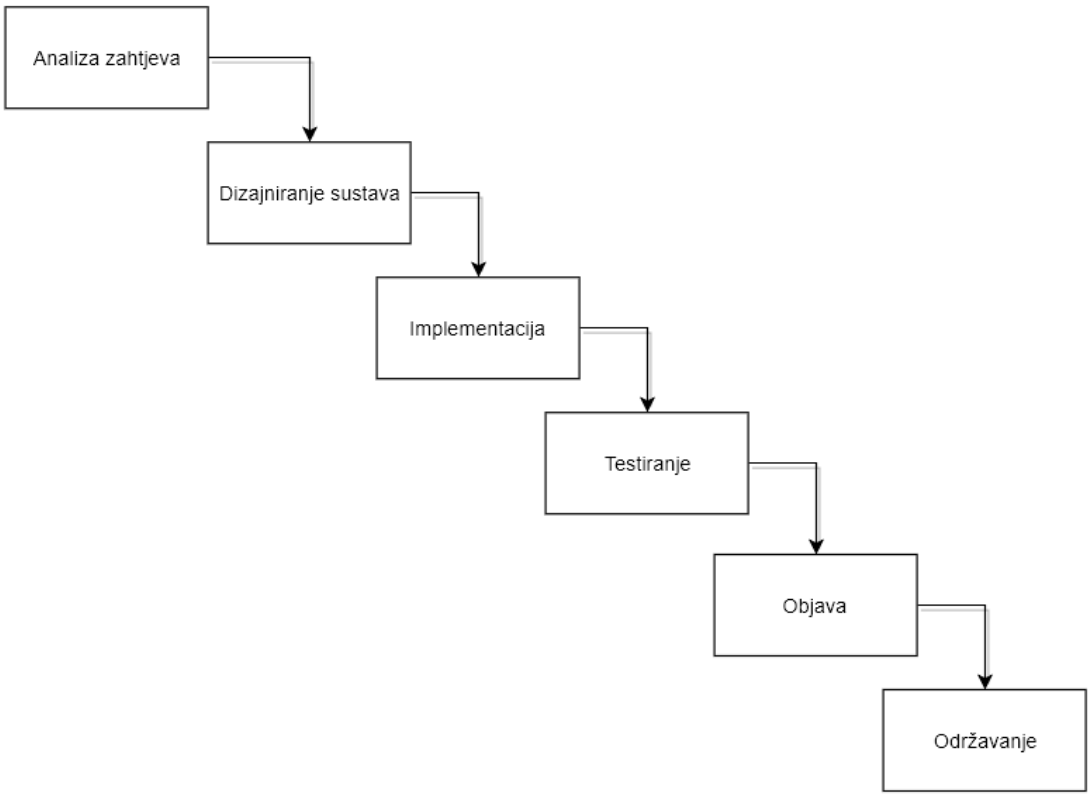
Crna kutija testiranje opisuje testiranje prilikom kojega se ne poznaje unutarnja struktura metoda i klasa. Takvo testiranje uglavnom rade testeri koji znaju što aplikacija treba raditi, ali ne i kako. U ovome slučaju testni scenariji se pišu iz specifikacija. Takva vrsta testiranja još se naziva i testiranje vođeno podatcima (engl. *data-driven*) jer se testiraju uneseni i dobiveni podatci [17]. Razlike između crne i bijele kutije načina testiranja prikazane su u tablici 3.1.

Tablica 3.1. Razlike između bijele i crne kutije načina testiranja

Crna kutija	Bijela kutija
Testeru nije poznata unutarnja struktura ni kod aplikacije	Tester zna unutarnju strukturu i kod aplikacije
Sistemsko i testiranje prihvatljivosti	Testiranje modula
Testove izvode testeri	Testove izvode programeri
Nije potrebno poznavanje programskog jezika u kojemu je aplikacija pisana	Potrebno je dobro poznavanje programskog jezika u kojemu je aplikacija pisana

3.5 Razine testiranja

Postoji puno modela po kojima se razvijaju aplikacije. Odabir modela ovisi o aplikaciji i njenim ciljevima. Neke metode se zasnivaju na brzini i zaradi, a neke na kvaliteti i dokumentiranju. Testiranje je uvijek uključeno u razvojni ciklus, ali svaka metoda osigurava drugačije mjesto testiranju. V-model razvoja aplikacije je razvijen jer su tradicionalni načini razvoja, kao na primjer model vodopad, imali problema s pravovremenim otkrivanjem grešaka [12]. Na slici 3.2 prikazan je razvojni model vodopad.

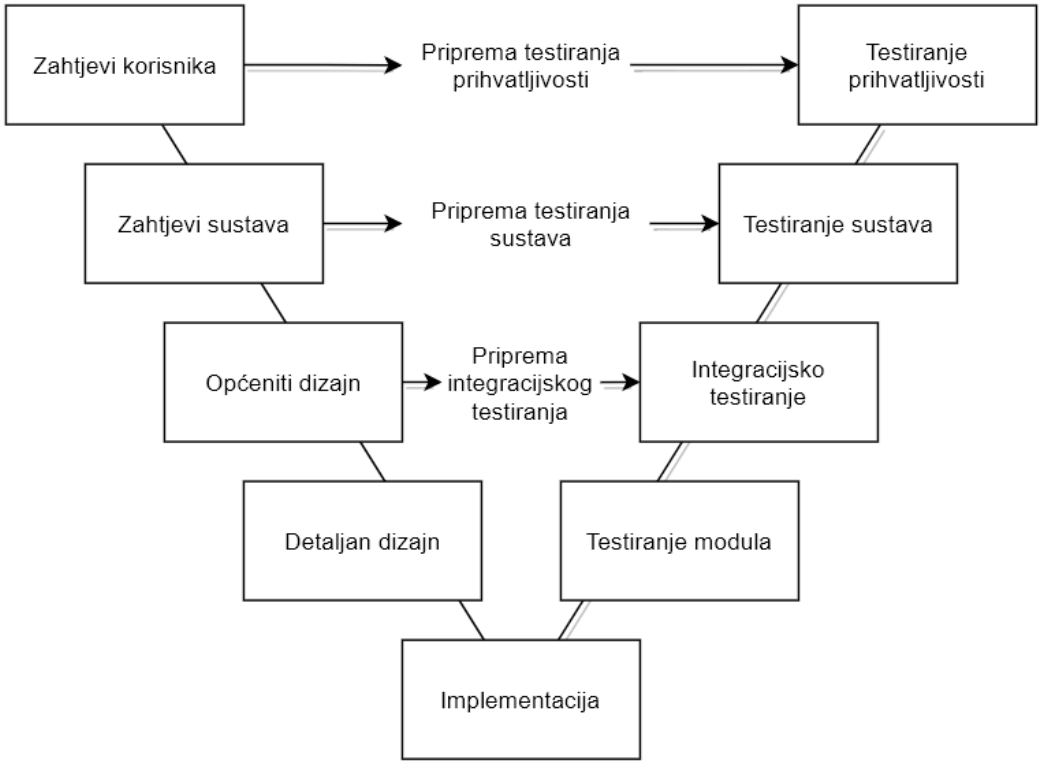


Slika 3.2. Model vodopad

Razvoj softvera metodom vodopad se sastoji od šest ključnih faza. Razvoj započinje analizom zahtjeva tijekom koje se skupljaju sve potrebne informacije o proizvodu, željama klijenta i moguće funkcionalnosti. Nakon toga se smišlja dizajn sustava. Nakon toga dolazi razdoblje implementacije gdje se pravi cijela aplikacija. Nakon završetka aplikacije, ona se testira i objavljuje kao gotov proizvod. Tada se aplikacija održava. Takav način razvoja aplikacije je linearan sekvencijalni i ne dolazi do preklapanja faza razvoja. Probleme kod takvog načina rada je što testiranje dolazi tek pred kraj razvojnog procesa i greške se kasno otkrivaju [18].

Jedna od glavnih smjernica V-modela je da se s testiranjem počne što ranije tijekom razvojnog procesa što je i jedan od ranije spomenutih principa testiranja. Testiranje bi se trebalo

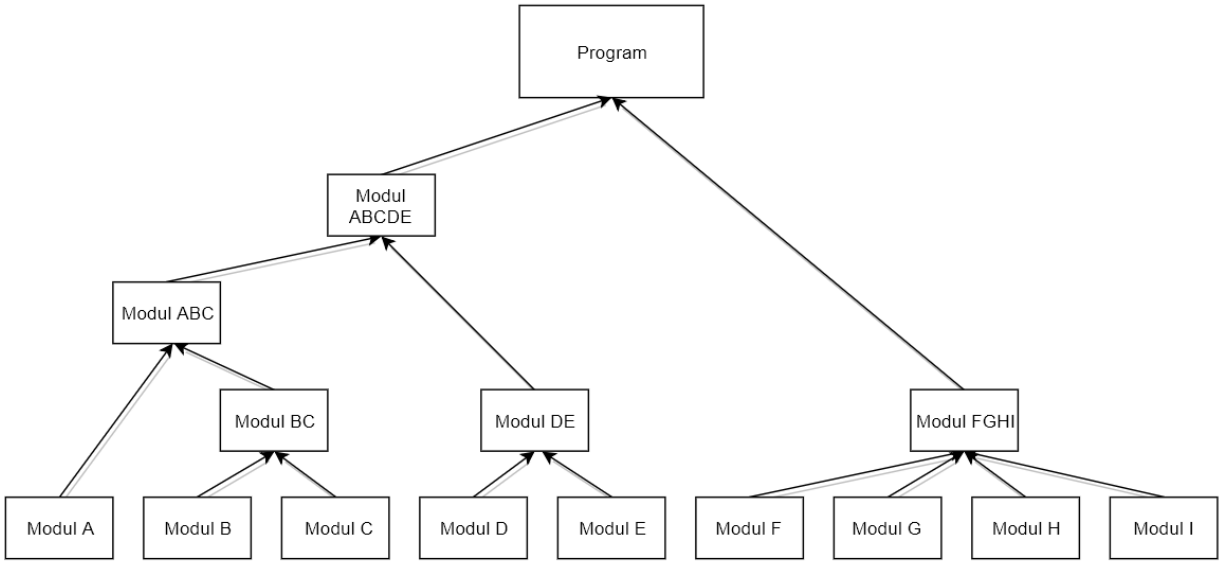
izvoditi paralelno s razvojem aplikacije u svim fazama razvoja. Slika 3.3 prikazuje V-model razvoja i testiranja aplikacije.



Slika 3.3. V-model

Navedene su razine testiranja: testiranje komponenti, integracijsko testiranje, sistemsko testiranje, testiranje prihvaćenosti koje će detaljnije biti objašnjene. Te se razine mogu kombinirati s nekim drugim modelima razvoja ili se ne moraju sve izvršavati. Kao što jedan od principa testiranja govori, testiranje ovisi o kontekstu i vrsti aplikacije.

Aplikacija se može simbolički prikazati kao na slici 3.4.



Slika 3.4. Simbolički prikaz rastava aplikacije na module

Aplikacija se može prikazati kao puno malih komponenti ili modula koji se spajaju ili imaju interakciju. Moduli su najmanje jedinice u aplikaciji, kao na primjer funkcije, metode ili objekti. Sve komponente zajedno tvore konačnu aplikaciju i sve njene funkcionalnosti.

3.5.1 Testiranje modula

Naziva se još unit, testiranje komponenti ili programsko testiranje. Kod ovog se testiranja testira rad modula, objekta, klase ili metode unutar aplikacije koji se može odvojeno testirati, odnosno najmanji izvodivi dio aplikacije. Testira se odvojeno od ostatka aplikacije. Može uključivati testiranje funkcionalnosti ili karakteristika. Na primjer, može se testirati uporaba resursa, performanse ili jednostavno daje li komponenta očekivane rezultate. Radi se prije testiranja cijele aplikacije. Najčešće ih pišu i izvode sami programeri nakon što završe pisanje najmanje jedinice u programu. Uglavnom su izvode u razvojnom okruženju uz dostupne *debugging* opcije. Greške pronađene prilikom testiranja komponenti se odmah ispravljaju i nema nikakvih posljedica. Takva se vrsta testiranja ubraja pod testiranje bijele kutije [17].

3.5.2 Integracijsko testiranje

Testira povezanost različitih modula i njihovu interakciju s drugim dijelovima aplikacije. Postoji nekoliko načina za izvršavanje integracijskog testiranja. Jedan od njih je veliki prasak (engl. *big bang*) gdje se sve komponente integriraju u aplikaciju odjednom i tada se testiraju. To

je ekstreman način implementacije komponenata, ali prednost takvog testiranja je što su u trenutku testiranja sve komponente gotove, ali negativno je što se odjednom napravi puno promjena koje zahtijevaju velik broj testova i puno vremena za njihovo izvršavanje. Tada je teže otkriti izvor nastale greške [12].

Postoji i drugi način, inkrementalno testiranje, koje podrazumijeva integraciju jedne po jedne komponente. Prednost takvog načina rada je što se jednostavno može otkriti uzrok greške jer nema puno novih komponenata, a nedostatak je što nisu sve promjene u aplikaciji napravljene pa se moraju pisati zamjenski kodovi [12]. Inkrementalnom se testiranju može pristupiti na dva načina. Od gore prema dolje i od dolje prema gore. Od dolje prema gore metoda znači da se testiraju niže komponente u odnosu na više komponente, odnosno module. Od gore prema dolje pristup testira od cijele aplikacije prema nižim komponentama.

3.5.3 Testiranje sustava

Testiranje sustava se može još nazvati i *release* testiranje jer se izvršava pred kraj razvojnog procesa, odnosno prije objavljivanja aplikacije. Svrha je usporedba aplikacije sa zadanim funkcijskim i nefunkcijskim zahtjevima. Može uključivati testiranje usporedbom sa specifikacijama, poslovnim procesima, primjerima uporabe (engl. *use cases*) itd. Aplikacije se testira u potpunosti sa svim svojim komponentama. Uobičajeno se da testeri nemaju znanja o detaljima implementacije (crna kutija). Testeri bi trebali biti neovisni što povećava vjerojatnost pronalaženja grešaka i ne dolazi do sukoba interesa razvoja i testiranja [17].

3.5.4 Testiranje prihvatljivosti

Provjerava ponašanje sustava u odnosu na zahtjeve naručitelja. Najčešće se provodi zajedno s timom naručitelja. Cilj je pokazati da sustav zadovoljava zahtjeve i da je spreman za upotrebu. Prilikom takvog testiranja aplikacija se daje korisnicima ili nekim stakeholderima na testiranje koji u kontroliranim uvjetima isprobavaju i testiraju aplikaciju [12].

Prva faza testiranja prihvatljivosti se naziva alfa testiranje. Takvo se testiranje odvija u kontroliranim uvjetima gdje odabrani korisnici isprobavaju aplikaciju i bilježe se njihove reakcije i problemi na koje nailaze. Druga faza je beta testiranje gdje se aplikaciju učini dostupnom dijelu pravih korisnika koji rabe aplikaciju u nekontroliranim uvjetima. Ti korisnici šalju svoje mišljenje, reakcije i probleme programerima koji tada ispravljaju pronađene greške.

3.6 Vrste testova

Vrste testova postoje kako bi se definirali ciljevi testova na određenoj razini. Potrebno je definirati vrstu testa jer testiranje jedne komponente i njene ispravne funkcionalnosti ne znači da je ta komponenta bez greške kod ostalih vrsta testova. Jednostavnije je testove podijeliti po ciljevima koje trebaju ostvariti. Cilj nekog testa može biti testiranje funkcionalnosti, ali isti tako može biti i neka nefunkcijska karakteristika, na primjer provjera sigurnosti ili provjera korištenja resursa. Po svome cilju testovi se mogu podijeliti na funkcijske i nefunkcijske [12].

3.6.1 Funkcijski testovi

Svrha funkcijskog testiranja je provjera radi li aplikacija ono što je zadano u specifikacijama. Može se reći da provjerava „što aplikacija radi“. Funkcijsko se testiranje izvodi tako da se napravi neka akcija ili unos u aplikaciji i dobiveni se rezultat uspoređuje s očekivanim. Provjerava se je li rezultat jednak očekivanom, ako nije došlo je do greške. Takvo se testiranje može izvoditi kao crna ili bijela kutija, testiranje modula, integracijsko testiranje, testiranje sustava ili testiranje prihvatljivosti [19].

Funkcijsko se testiranje izvodi s krajnjim korisnicima na umu pokušavajući upotrebljavati aplikaciju kao što bi je korisnik upotrebljavao. Testni scenariji se pišu uzimajući u obzir ponašanje korisnika i njihova moguća očekivanja od aplikacije i ne uključuje testiranje performansi. Zato postoji druga vrsta testova, nefunkcijski testovi [19].

3.6.2 Nefunkcijski testovi

Bitne karakteristike aplikacije su njena sigurnosti, performanse, prenosivost itd., a to nisu funkcije koje se testiraju funkcionalnim testiranjem. Kod takvog testiranja se ne testira ispravnost rada aplikacije nego neke općenitije karakteristike.

Jedan od nefunkcijskih testiranja je testiranje sigurnosti. Tu se provjerava štiti li aplikacija korisnikove podatke. Pod testiranje sigurnosti se ubrajaju provjera autentifikacije i autorizacije. Autentifikacija je proces u kojemu aplikacija traži od korisnika na primjer korisničko ime i lozinku kako bi korisnik mogao pristupiti svojim podatcima. Bitno je da nitko osim osobe s pravom kombinacijom korisničkog imena i lozinke ne može pristupiti podatcima. Autorizacija govori o privilegijama i pravima pristupa koje određeni korisnik može imati. Bitno je osigurati da korisnici nemaju ovlasti nad aplikacijom ili podatcima za koje nisu autorizirani [16].

Testiranje upotrebljivosti se radi kako bi se ocijenilo koliko je jednostavno naučiti se upotrebljavati je i rabiti je. Uglavnom se izvode sa samim korisnicima ili s ljudima koji nisu nikada

bili u dodiru s aplikacijom. Izvode se kao testovi crne kutije. Ova vrsta testiranja je bitna jer su osobe koje razvijaju aplikaciju dobro upućene u tehnologiju i neke akciju su intuitivne, dok s druge strane korisnici ne moraju nužni biti upućeni u moderne tehnologije [20].

Testiranje prenosivosti se izvodi tako da se promijeni okruženje aplikacije i nadziru se njene performanse. Pod promjenom okruženja misli se na promjenu operativnog sustava, uređaja ili Internet preglednika. Za aplikaciju je bitno da radi u različitim okruženjima jer i korisnici upotrebljavaju različito okruženje od onoga koje rabe programeri. Kako bi se takvi testovi uspješno provodili, bitno je prilikom razvoja aplikacije imati na umu prenosivost i različita okruženja [16].

Testiranje integriteta podataka je jedan od bitnijih vrsta testova. Integritet označava da su podatci potpuni i cjeloviti. Takvim se testovima provjerava da spremanje, prijenos ili dohvaćanje podataka ne utječu na same podatke. Utvrđuje se postoje li neke greške zbog kojih može doći do korupcije podataka ili neovlaštenog pristupa njima [16].

Još neki od nefunkcijskih testova su testiranje izdržljivosti gdje se testira kako sustav podnosi velika opterećenja, testiranje internacionalizacije i lokalizacije gdje se provjerava reagira li aplikacija u skladu s postavljenom lokacijom ili jezikom, itd.

3.7 Ručno testiranje

Ručno (engl. *manual*) testiranje je najstarija verzija testiranja. Podrazumijeva testera koji je upućen u aplikaciju i zna koje sve funkcionalnosti ona pruža. Tester sastavlja testne scenarije koji trebaju biti napisati tako da provjeravaju aplikaciju i njene rubne slučajeve. Nakon toga, tester izvršava testove i označava jesu li prošli (dogodilo se što je očekivano) ili su pali (dogodilo se nešto neočekivano što se nije trebalo dogoditi). Ako je neki test pao, bitno je imati zapis o tome što se točno dogodilo i koje su akcije dovele do tog pada kako bi programeri znali ponoviti korake i vidjeti što je pošlo po krivu te popraviti grešku.

Problem kod ručnog testiranja je što zahtijeva puno vremena i ljudskih resursa, a što aplikacija postaje veća i kompliciranija, problem raste. Ručno testiranje se ne smatra konzistentnim ni ponovljivim jer svaki ručni tester može pristupiti testnom scenariju ili testu drugačije. Razlike u brzini izvođenja nekih operacija ili mjestu klika mogu rezultirati drugačijim rezultatima [21].

Kako bi se doskočilo problemima ručnog testiranja, preporučuje se automatsko testiranje gdje se pišu skripte koje izvršavaju zadane testove. Automatskom testiranju će biti posvećeno cijelo sljedeće poglavlje.

4 AUTOMATIZACIJA TESTIRANJA

Automatizacija testiranja ima puno prednosti za testiranje softvera. Poboljšava rezultate i kvalitetu, povećava pouzdanost i smanjuje odstupanja u rezultatima, ubrzava proces, povećava pokrivenost testovima, a u konačnici može povećati sveukupnu kvalitetu softvera. Dobra automatizacija može dovesti do ubrzanja objavljivanja aplikacije, poboljšati kvalitetu objave, povećati pokrivenost testovima, smanjiti troškove testiranja i omogućiti ranije otkrivanje grešaka ako se pravilno testira [21].

Jedan od glavnih problema kod automatizacije testiranja je skalabilnost i održavanje testova. Softveri i aplikacije se vremenom mijenjaju. Mijenja im se dizajn i funkcije pa se može dogoditi da stari testovi više nisu prikladni za novu verziju aplikacije. Drugi problem je niska pokrivenost automatiziranih testova. Kao što je spomenuto, aplikacija se mijenja i stari testovi prestaju biti aktualni. Potrebno ih je nadopuniti ili promijeniti, što zahtijeva ljudske resurse [21].

Automatizacija testiranja je proces u kojem se upotrebljavaju alati za testiranje softvera. Kreiraju se skripte koji se izvršavaju i na kraju se rezultat skripte uspoređuje s očekivanim rezultatom. Automatizacija testova se vrši pomoću alata za automatizaciju testiranja. Alat za automatizaciju testiranja je softverska aplikacija koja omogućuje analizu aplikacije i izvršavanje testova [16].

Prvi bi trebali biti automatizirani testovi koje je potrebno konstantno ponavljati i testovi koji zahtijevaju puno podataka za izvršavanje. To su na primjer testovi koje je teško izvoditi ručno, koji provjeravaju rad aplikacije na različitim platformama ili povjeravaju GUI. Ne bi trebalo automatizirati testove koji će se izvršiti samo jednom i nikada više, kao ni testove koji testiraju granice aplikacije i izmišljaju se na mjestu (ad hoc) ili testovi čiju prolaznost procjenjuje čovjek.

4.1 Selenium-WebDriver

Selenium je nastao 2004. godine kada je Jason Huggins, radeći ručno testiranje web stranice, došao na ideju kako bi se repetitivni testovi mogli automatizirati i sami izvoditi. Napisao je JavaScript biblioteku koja je omogućavala interakciju koda s internetskom stranicom i ponovljivo izvršavanje testova na različitim internetskim preglednicima. Ta je biblioteka postala osnova za današnje Selenium alate. S vremenom su internetski preglednici počeli ograničavati JavaScript i zbog toga je Selenium postajalo sve teže upotrebljavati. 2006. godine Google-ov inženjer Simon Stewart je počeo raditi na projektu koji je nazvao WebDriver i trebao je služiti kao zamjena za tadašnji Selenium. WebDriver je bio zamišljen kao alat za testiranje koji direktno

komunicira s preglednikom izbjegavajući ograničenja JavaScript-a. 2008. godine Selenium i WebDriver su spojeni u jedan projekt pod nazivom Selenium 2.0 [22].

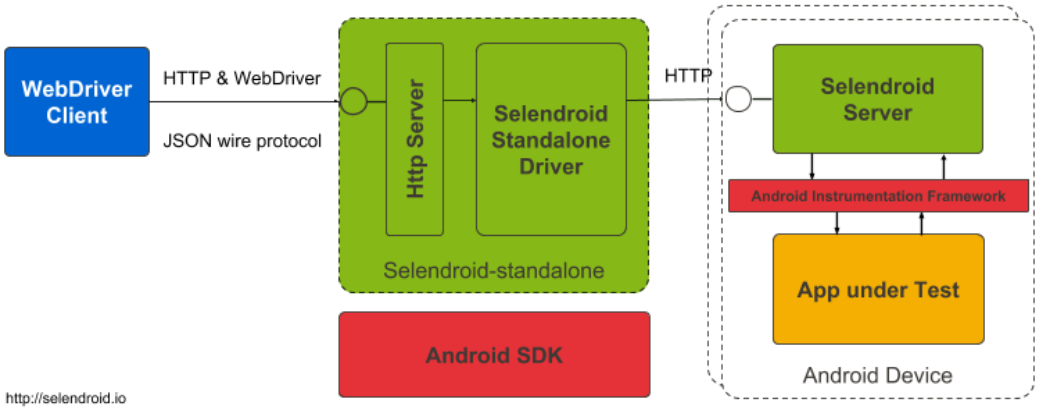
Jedan od alata koje Selenium nudi je Selenium IDE koji omogućuje pravljenje jednostavnih i brzih testova pritiskanjem po stranici i spremanjem tih akcija kao jedan test, ali se on ne preporučuje za automatizaciju testiranja svih testova. Preporučuje se pisanje skripti uporabom Selenium-WebDriver-a. Selenium-WebDriver podržava sve popularne Internet preglednike (Chrome, Firefox, Safari, Opera, IE,…), ali i mobilne platforme iOS i Android uz pomoć nekih dodatnih alata (Appium, Selendroid ili ios-driver). Podržava pisanje testova u Javi, Pyhtonu, Rubyju i C# [22]. U ovom će se projektu upotrebljavati Javu za pisanje testova, a od dodatnih alata koristit će se Appium.

Aplikacijsko programsko sučelje (API, engl. *Application Programming Interface*) se sastoji od tehničkih specifikacija koje određuju kako se informacije razmjenjuju između programa i sučelja napisanog po tim specifikacijama i objavljenog za upotrebu. Aplikacije pomoću API-ja međusobno komuniciraju i razmjenjuju podatke [23]. WebDriver Selenium-u služi kao API za interakciju s preglednikom i upravljanje njime. Kako bi se Selenium 2.0 rabio u projektu, potrebno ga je uvesti. Najjednostavniji način za to učiniti je pomoću Maven-a. Maven je alat koji služi za automatsko skidanje ovisnosti i njihovo uvoženje u projekt uzimajući u obzir verzije. Svaki malo bolji program za uređivanje koda ima opciju uporabe Maven-a. U ovome će se projektu upotrebljavati IntelliJ razvojno okruženje za pisanje testova [22].

4.2 Izvršavanje testova

4.2.1 Selendroid

Selendroid je okvir za automatizaciju testova za nativne, hibridne i web Android aplikacije. Oslanja se na Selenium WebDriver. Testove je moguće izvršavati na emulatorima i fizičkim uređajima. Nije potrebno modificirati aplikaciju koja se testira i moguće je izvršavati više od jednog testa u isto vrijeme. Sastoji se od četiri glavne komponente: WebDriver klijenta, Selendroid servera, AndroidDriver applikacije i samostojećeg Selendroida [24]. Na slici 4.1. je prikazana povezanost tih komponenti.



Slika 4.1. Arhitektura Selendroid-a [24]

WebDriver klijent je je Java klijentska biblioteka zasnovana na Selenium Java klijentu. Sasmostojeći Selendroid se upotrebljava za instalaciju servera i aplikacije pod testom na uređaj, služi za komunikaciju između klijenta i servera. Selendroid server je server koji se izvršava uz aplikaciju na mobilnom uređaju. AndroidDriver je aplikacija na mobilnom uređaju koja omogućuje pristup elementima mobilne aplikacije pod testom [24].

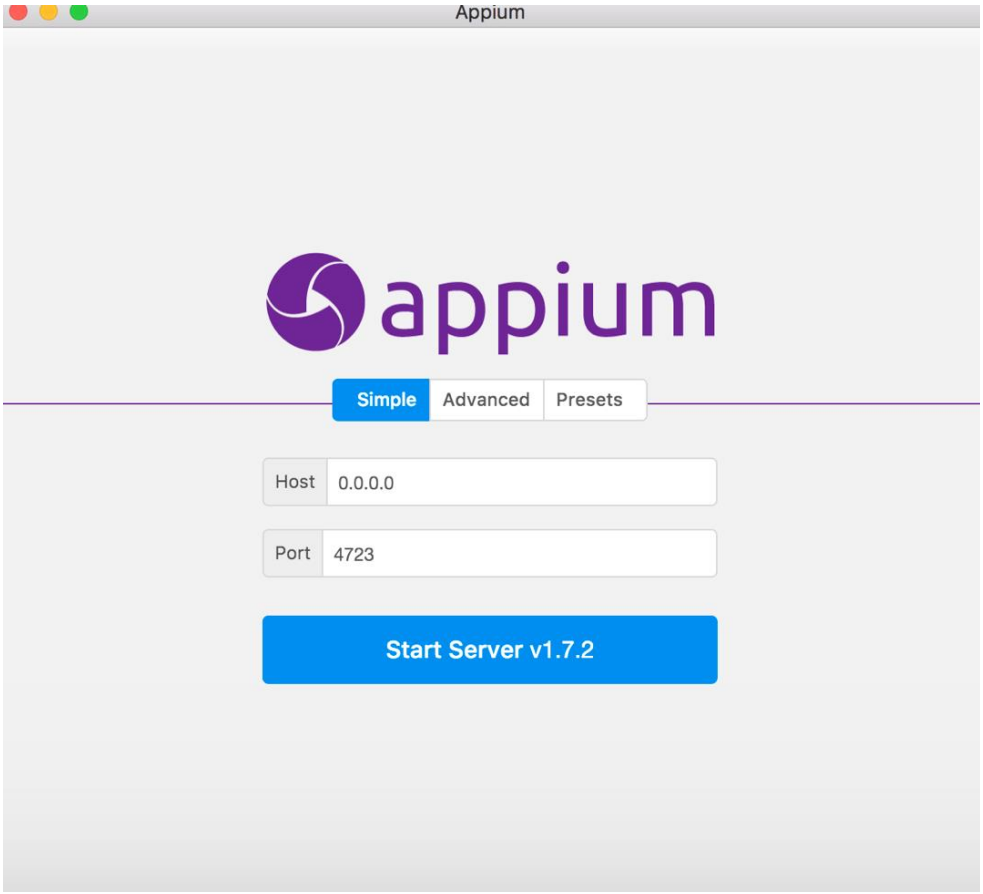
4.2.2 Ios-driver

Ios-driver služi za automatizaciju testiranja aplikacija na iOS mobilnim uređajima. Omogućuje testiranje nativnih, hibridnih i web aplikacija na emulatorima i fizičkim uređajima. Upotrebljava pristup skoro identičan Selendroid-u samo za iOS arhitekturu. Implementira JSON Wire protokol kako bi komunicirao s iOS aplikacijom [25].

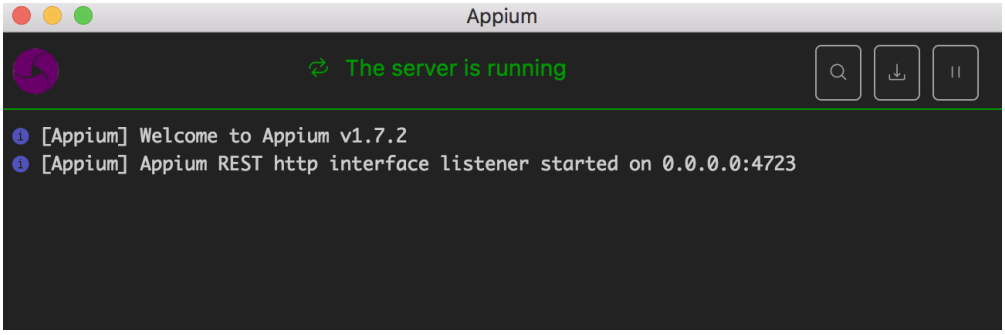
4.2.3 Appium

Appium je alat otvorenog koda za automatizaciju nativnih, web i hibridnih aplikacija za iOS, Android i Windows platforme, što znači da omogućuje višeplatformsko (engl. *crossplatform*) testiranje. Višeplatformsko znači da se isti kod (u ovom slučaju test) može upotrijebiti za različite platforme [26]. U pozadini za iOS upotrebljava XCUITest i UIAutomation okvire. Ti su okviri za testiranje odobreni i napravljeni od strane Apple-a za pisanje testova uporabom Xcode razvojnog okruženja tijekom samog razvoja aplikacije, dakle za testiranje modula. Za Android upotrebljava UiAutomator. Upotrebljavaju se WebDriver i JSON Wire protokol koji je proširen u Mobile JSON Wire protokol. Proširenje je napravljeno dodavanjem novih, za mobilne uređaje specifičnih, ponašanja i načina lociranja kao na primjer informacije vezane za rotaciju uređaja. Appium je zapravo server napisan u Node.js-u koji prima zahtjev od klijenta, izvršava taj zahtjev na mobilnom uređaju i šalje rezultat i odgovor klijentu. Automatizacija se izvršava kao sjednica (engl. *session*). Za pokretanje sjednice šalje se zahtjev s ključnim riječima i vrijednostima kojima se

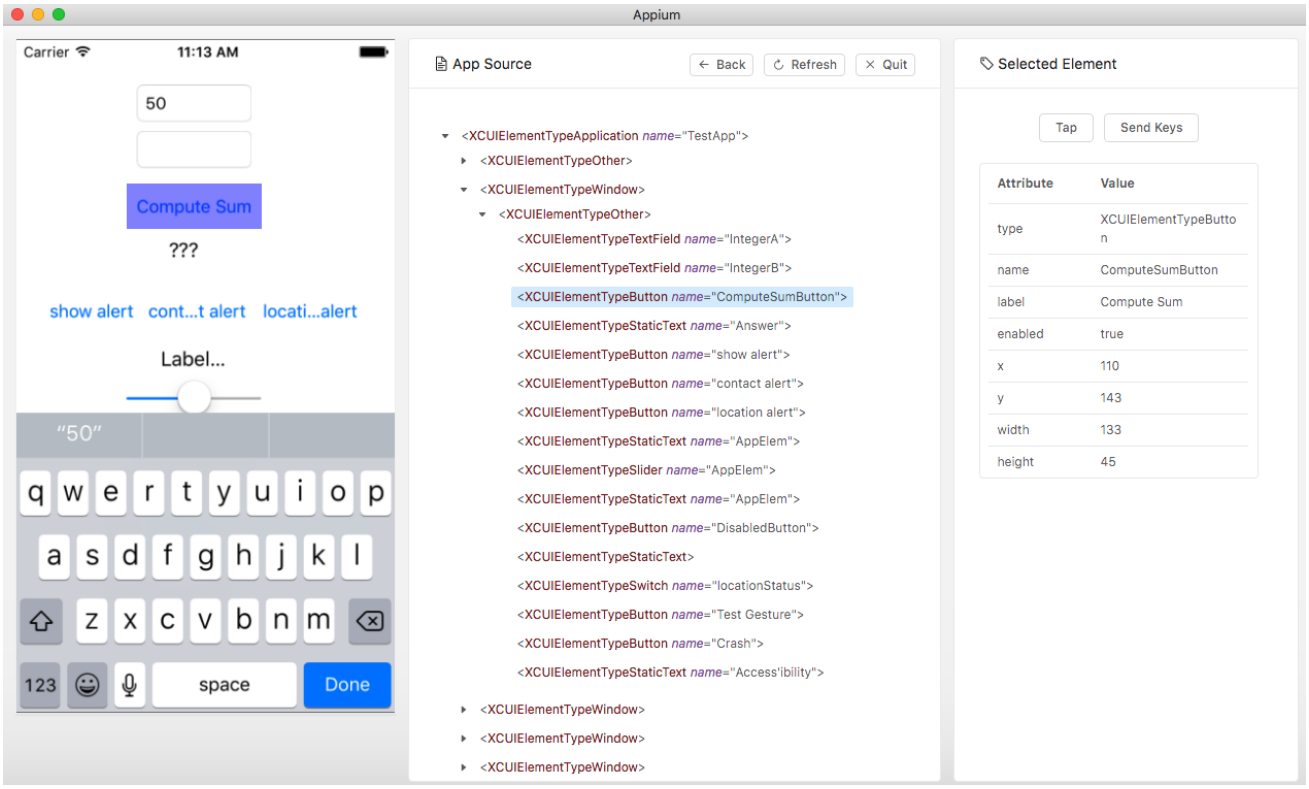
postavlja sjednica [26]. Na slici 4.2. prikazana je Appium Desktop aplikacija za računala. Aplikacija služi kao grafičko sučelje za Appium server (prikazan na slici 4.3.), ali i pruža alat, Inspector (prikazan na slici 4.4.), za detaljan pregled elemenata aplikacije i ispis njihovih oznaka.



Slika 4.2. Početni prozor Appium Desktop aplikacije



Slika 4.3. Pokrenut Appium server



Slika 4.4. Appium Inspector [27]

Appium koristi AppiumDriver klasu koja nasljeđuje RemoteWebDriver klasu. RemoteWebDriver je Selenum klasa koja implementira WebDriver sučelje (engl. *interface*), a koristi se za izvršavanje testova na udaljenim uređajima. Radi na principu server-klijent. AppiumDriver sadrži dodatke koji se odnose samo na mobilne uređaje i omogućuje izvršavanje testova korištenjem Appium servera, a dijeli se na AndroidDriver i IOSDriver. Svaki od tih drivera nasljeđuje AppiumDriver i sadrži dodatke specifične za platformu kojoj su namijenjeni. Za svaki je operativni sustav potrebno koristiti *driver* namijenjen točno tom operativnom sustavu [36]. Uz AppiumDriver postoje i driveri za svaki od popularniji Internet preglednika – ChromeDriver, FirefoxDriver, itd.

4.3 Pisanje testova

Java je programski jezik koji se upotrebljava za različite projekte i u različite svrhe. Kako bi se Java upotrebljavala za testiranje, potrebno je upotrijebiti neki od okvira (engl. *framework*) za Javu za testiranje. Ti okviri donose nove funkcionalnosti Javi i WebDriver-u te olakšavaju upravljanje testovima, njihovo pokretanje i izvršavanje. Omogućavaju selektivno ili sekvencijalno izvršavanje testova, generiranje rezultata, ispis koraka testa, spremanje slike s ekrana u

specificiranim trenucima i mnoge druge mogućnosti. Spomenut će se i usporediti dva dostupna okvira: JUnit i TestNG jer su to jedni od najpopularnijih i najkorištenijih okvira.

4.3.1 JUnit

JUnit je okvir za pisanje testova, a kao testovi koji se mogu najbolje izvesti pomoću njega navode se testovi modula. Besplatan je i otvorenog je koda. Bitan je kod testiranja vođenog razvojem jer omogućuje pisanje testova i prije pisanja samog koda aplikacije. Omogućuje definiranje testnih metoda, izvođenje testova, uspoređivanje dobivenih i očekivanih rezultata. Testne klase je moguće povezati i izvoditi ih jedne za drugom [28].

4.3.2 TestNG

TestNG je okvir za pisanje testova nastao kao poboljšanje JUnite-a tako što su dodane nove funkcionalnosti. Postaje sve popularniji i prikladan je za pisanje svih vrsta testova. Neke od dodanih funkcionalnosti su mogućnost izvršavanja koda prije i poslije izvršavanja grupe testova. Omogućuje označavanje testa pripadanjem u neku grupu pa se tako testovi mogu označiti nekom oznakom i onda se kasnije mogu izvršiti samo testovi s tom oznakom. TestNG pruža mogućnost izvršavanja istog testa paralelno na više niti. Još jedna od mogućnosti koje JUnit nema je prihvaćanje unosnih podataka iz XML, CSV ili običnog tekstualnog dokumenta. TestNG ima mogućnost preskakanja testova koji ovise o nekim drugim testovima ako su ti testovi pali [29].

JUnit je stariji i zbog toga ima više korisnika i više materijala kojima se korisnici mogu poslužiti, ali TestNG postaje sve popularniji i zajednica korisnika je aktivna. Zbog svih mogućnosti koje TestNG pruža, a JUnit nema, za ovaj će se projekt upotrebljavati TestNg.

4.4 Fizički uređaj, emulator i simulator

Testiranje na fizičkom uređaju je najlogičnije jer se aplikacije prave za fizičke uređaje i korisnik će ih tako upotrebljavati. Potrebno je znati ograničenja fizičkog uređaja i napraviti aplikaciju koja radi u skladu s njima. Problem kod takvog testiranja je što postoji puno različitih uređaja (pogotovo za Android) s različitim hardverom, a i softverom kao što je prikazano u potpoglavlju 2.1. Programer ne može testirati aplikaciju na svim mogućim uređajima da bi se uvjerio da ona radi. Zbog toga postoje emulatori i simulatori.

Emulatori su programi koji kompajliraju kod aplikacije tako da se aplikacija može izvršavati na računalu, a ne na mobilnom uređaju. Emulatori se ponašaju kao fizički uređaji i oponašaju njihov hardver i operativni sustav. Nedostatak emulatora je što se aplikacija zapravo izvršava na računalu pa taj emulirani virtualni uređaj nema senzore ni hardver kao što su kamera

ili mikrofon [5]. Neki emulatori nude mogućnost i upravljanja senzorima. Na primjer Genymotion [30], Android emulator, ima mogućnost davanja GPS lokacije uređaju, upotrebljavanje kamere i mikrofona (ako ih računalo ima) i još neke mogućnosti.

Simulatori su jednostavniji softveri koji simuliraju samo dio rada fizičkog uređaja i hardvera. Jednostavniji su za uporabu i „lakši“ za računalo na kojemu se izvršavaju. Nema nikakve mogućnosti interakcije s hardverom [5].

Najbolja je praksa upotrebljavati emulator ili simulator na početku razvoja kako bi programer bio siguran da se aplikacija izvršava kao što je zamišljeno. Oni uglavnom dolaze besplatno u sklopu razvojnog okruženja. Za testiranje je bitno testirati aplikaciju na pravom uređaju (čak i kada se ne mogu testirati svi mogući uređaji) jer simulatori i emulatori ne daju pravu sliku o izvršavanju aplikacije. Emulator i simulator pružaju kontrolirano okruženje u kakvom aplikacija nikad neće biti izvršavana kod korisnika.

5 TESTIRANJE MULTITASK APLIKACIJE

Kako bi se prikazalo razvijanje testova koji se mogu izvršavati na dvije različite mobilne platforme, Android i iOS, potrebno je razviti jednaku nativnu aplikaciju za oba sustava. Razvoj nativnih aplikacija je odabran kako bi pratio trendove razvoja mobilnih aplikacija u svijetu. Svaka aplikacija koja želi uspjeti i imati što više korisnika treba imati aplikaciju za oba operativna sustava. Neke velike kompanije, kao na primjer Facebook, razvijaju hibridnu aplikaciju upotrebljavajući svoj programski jezik React-Native, dok druge, na primjer Niantic s aplikacijom Pokemon Go, razvijaju dvije paralelne nativne aplikacije. Cilj je pružiti korisnicima na oba sustava što sličnije iskustvo. Da bi se osigurala kvaliteta aplikacija potrebno ih je testirati. Najbolji način za testiranje takvih aplikacija je automatizacija testiranja uz testove pisane za obje platforme simultano, tj. pisanje jedne skripte koja se može izvršavati na više različitih platformi.

5.1 Specifikacije aplikacije

MultiTask aplikacija je zamišljena kao aplikacija koja ima različite funkconalnosti na kojima se može prikazati raspon mogućnosti testiranja korištenjem Appiuma. Aplikacija je podijeljena na tri djela: kalkulator, računanje potrošnje goriva i bilješke. Na donjem djelu ekrana aplikacije nalazi se izbornik s tri ponuđena izbora „Kalkulator“, „Gorivo“ i „Bilješke“. Pritiskom na svaki od tih izbora otvara se odabrani ekran (Slike 5.1.-5.3.).

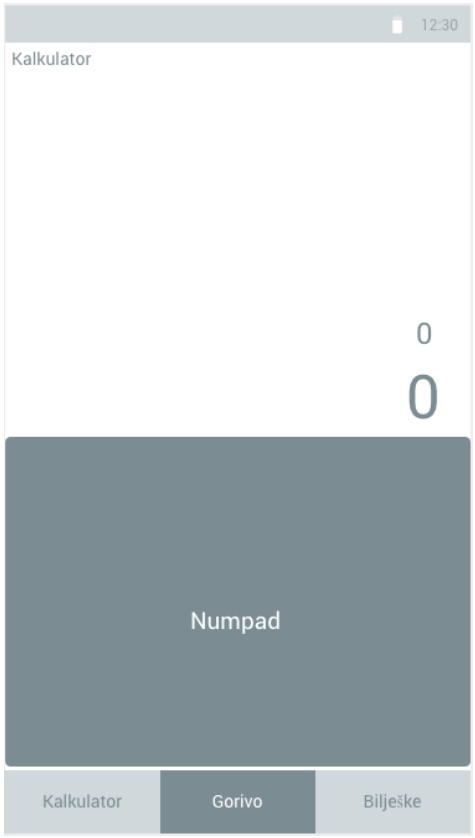
5.1.1 Ekran „Kalkulator“

Na ekranu Kalkulator nalaze se gumbi s brojevima, računskim operacijama i dodatnim opcijama brisanja jedne znamenke, brisanja svih podataka, jednakosti i decimalnog zareza. Funkcionalne specifikacije kalkulatora prikazane su u tablici 5.1.

Tablica 5.1. Funkcionalne specifikacije ekrana Kalkulator.

Akcija	Preduvjet	Očekivani rezultat
Pritisak gumba s brojem	Prazan ekran	Prikazuje se pritisnuti broj na ekranu
	Ekran sadrži broj	Pritisnuti broj se pokazuje desno od postojećeg broja
Pritisak na gumb decimalnog zareza	Prazan ekran	Ne dogodi se ništa
	Ekran sadrži broj	Decimalni se zarez prikazuje desno od postojećeg broja
	Broj već sadrži decimalni zarez	Ne dogodi se ništa
Pritisak na gumb računske operacije	Prazan ekran	Ne dogodi se ništa
	Ekran sadrži broj	Broj se pomakne na gornji ekran, operacija se prikaže na glavnom ekranu
Pritisak na gumb jednakosti	Unesena dva broja i računska operacija	Rezultat operacije se prikazuje na gornjem ekranu, glavni ekran ostaje prazan
Pritisak na gumb „C“	Prazan ekran	Ne dogodi se ništa
	Ekran sadrži broj	Obriše se posljednji unesen broj
Pritisak na gumb „AC“		Brišu se svi podaci s ekrana

Slika 5.1. prikazuje *wireframe* ekrana kalkulatora koji služi kao predložak za izradu izgleda aplikacije.



Slika 5.1. Dizajn ekrana „Kalkulator“

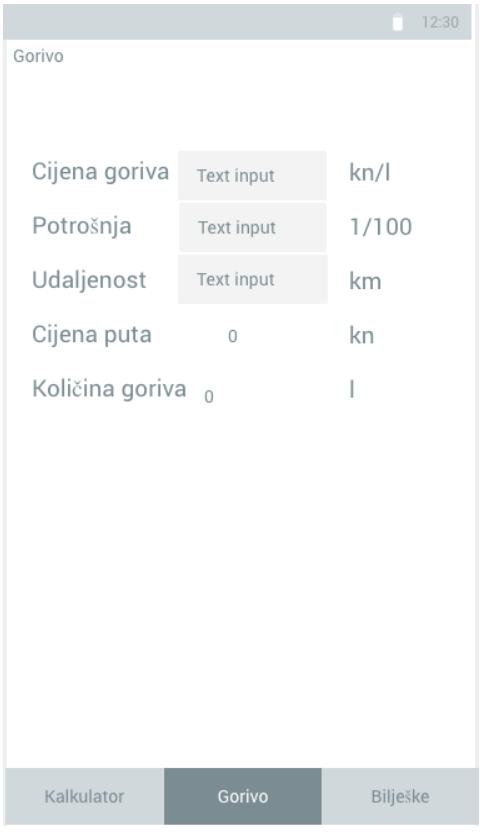
5.1.2 Ekran „Gorivo“

Na ekranu Gorivo nalaze se tri polja za unos cijene goriva, potrošnje i udaljenosti. Pritiskom na polje za unos prikazuje se brojčana tipkovnica kojom se unose brojevi. Unosom cijene goriva, potrošnje i udaljenosti dobiva se izračun cijene puta i količina potrošenog goriva. Unosom potrošnje i udaljenosti dobiva se potrebna količina goriva bez cijene puta. Funkcionalne specifikacije ekrana prikazane su u tablici 5.2.

Tablica 5.2. Funkcionalne specifikacije ekrana Gorivo

Akcija	Preduvjet	Očekivani rezultat
Pritisak na polja za unos		Prikaz numeričke tipkovnice
Unos broja numeričkom tipkovnicom	Odabrano polje za unos	Broj je upisan u polje za unos
Završetak unosa brojeva u polja	Sva tri polja sadrže brojeve	Prikazuje se rezultat računanja po formulama: $količina\ goriva = \frac{udaljenost \cdot potrošnja}{100}$ $cijena\ puta = količina\ goriva\ \cdot cijena$
	Polja Potrošnja i udaljenost sadrže brojeve	Prikazuje se rezultat računanja Količine goriva: $količina\ goriva = \frac{udaljenost \cdot potrošnja}{100}$

Slika 5.2. prikazuje *wireframe* ekrana kalkulatora koji služi kao predložak za izradu izgleda aplikacije.



Slika 5.2. Dizajn ekrana „Gorivo“

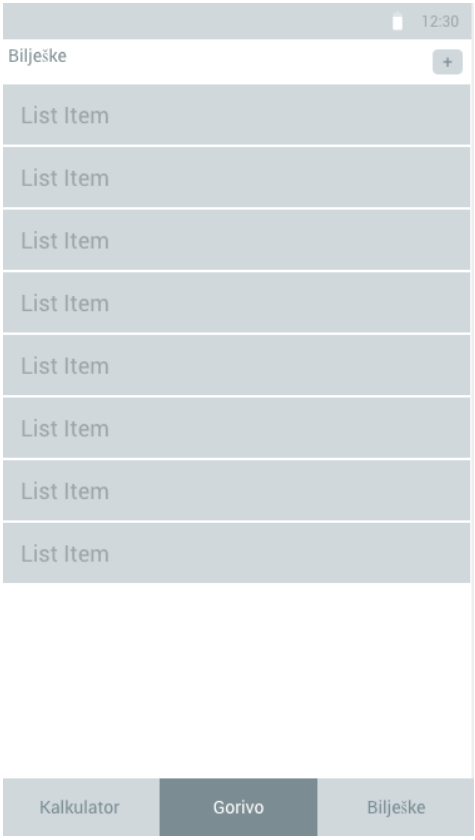
5.1.3 Ekran „Bilješke“

Ekran Bilješke omogućuje korisniku dodavanje i brisanje zapisa. Bilješka se dodaje pritiskom na za to predviđeni gumb, a briše, ovisno o sustavu, pritiskom na bilješku ili pomicanjem bilješke u lijevo. Bilješke trebaju ostati spremljene i kada korisnik napusti aplikaciju. Funkcionalne specifikacije ekrana s bilješkama prikazane su u tablici 5.3.

Tablica 5.3. Funkcionalne specifikacije ekrana Bilješke

Akcija	Preduvjet		Očekivani rezultat
Pritisak gumba „+“			Prikazuje se *pop-up* koji omogućuje unos teksta i tekstualna tipkovnica
	Otvoren pop-up		Uneseni se tekst prikazuje u polju za unos teksta
Pritisak na gumb „Save“	Otvoren pop-up Unesen tekst		Uneseni tekst se sprema i prikazuje na listi
Pritisak na gumb „Cancel“	Otvoren pop-up Unesen tekst		*Pop-up* se zatvara i tekst nije spremljen
Pritisnuti i povući bilješku iz liste malo na lijevo	iOS	Postoji bilješka na listi	Prikazuje se „Delete“ dugme na čiji se pritisak briše bilješka
Pritisnuti i povući bilješku iz liste na lijevo	iOS	Postoji bilješka na listi	Briše se bilješka
Pritisnuti bilješku	Android	Postoji bilješka na listi	Prikazuje se *pop-up* koji nudi opciju brisanja bilješke pritiskom na tipku „Yes“ ili odustajanje pritiskom na tipku „No“

Slika 5.3. prikazuje *wireframe* ekrana kalkulatora koji služi kao predložak za izradu izgleda aplikacije.

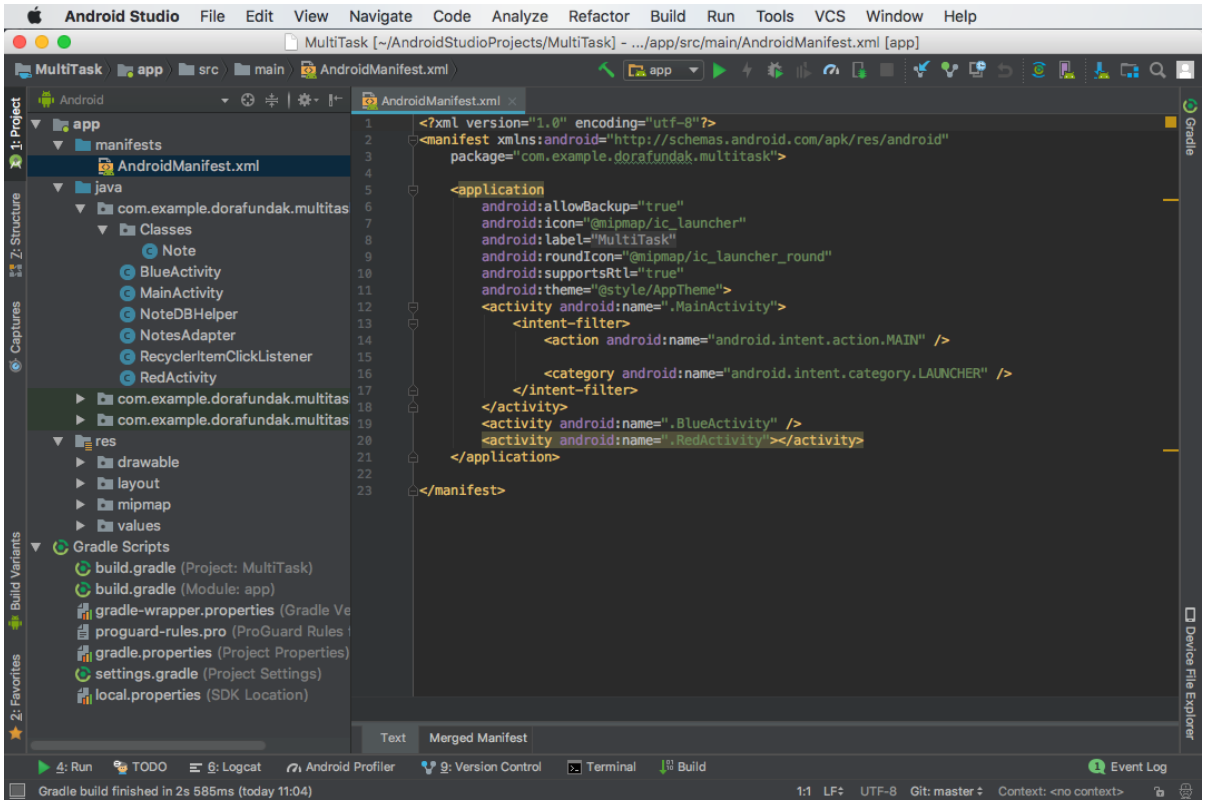


Slika 5.3. Dizajn ekrana „Bilješke“

5.2 Razvoj aplikacije

5.2.1 Android

Razvoj nativne aplikacije za Android odvija se u razvojnom okruženju Android Studio. Ono pruža mogućnost razvoja aplikacije u Java ili Kotlin programskom jeziku. Android Studio se zasniva na IntelliJ IDEA razvojnom okruženju. Uz mogućnosti uređivanja i prikazivanja koda, Android Studio pruža opcije specijalizirane za razvoj mobilne aplikacije. Upotrebljava Gradle za *buildanje* aplikacije, dolazi s emulatorom koji ima skoro sve opcije kao i pravi uređaj, moguće je pokrenuti samo promijenjeni dio aplikacije bez stvaranja novog APK, ima GitHub integraciju i mnoge druge pogodnosti [32]. Razvojno okruženje Android Studio prikazano je na slici 5.4.



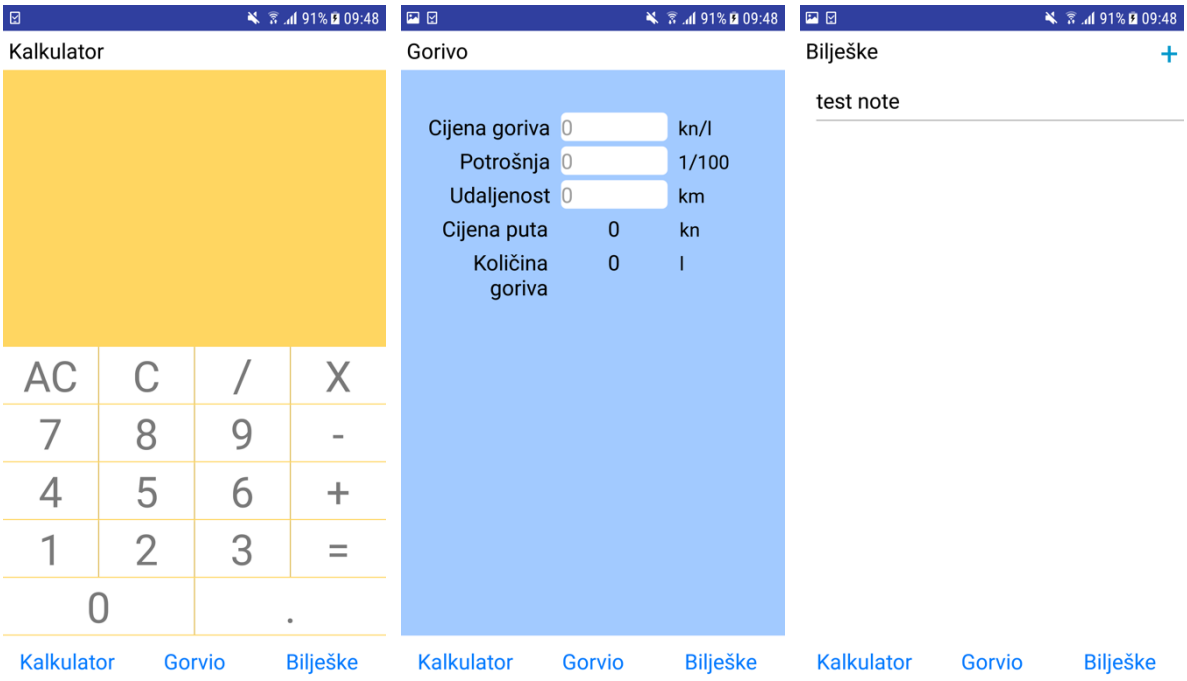
Slika 5.4. Android Studio na macOS-u

Projekt u Android Studiju sadrži module s izvornim kodom aplikacije. Glavna podjela je na app (koji sadrži: manifest, java i res) te Gradle Scripts. Struktura MultiTask aplikacije u Android Studiju vidi se na slici 5.4 u lijevom izborniku. Manifest sadrži AndroidManifest.xml koji u XML opisnom jeziku sadrži opis aplikacije i svake njene komponente. U manifestu je navedeno koje će sve dozvole aplikacija tražiti (npr. pristup internetu), definira strukturu aplikacije popisom komponenti s navedenim aktivnostima, uslugama i servisima. Naveden je potreban hardware i software za izvršavanje aplikacije. Detaljniji opis aplikacije nalazi se u Gradle Scripts skripti gdje se navodi trenutna verzija aplikacije i koje se sve dodane biblioteke upotrebljavaju.

Java datoteka sadrži izvorni kod aplikacije napisan u Javi. Tamo se nalaze sve metode, klase, sučelja (engl. *interface*), *activity*-ji i ostalo. Activity je klasa koja predstavlja jedan ekran aplikacije, a početni ekran je zadan MainActivity klasom. Uz MainActivity klasu, napravljene su BlueActivity i RedActivity koje predstavljaju dodatna dva ekrana potrebna aplikaciji.

Res folder sadrži sve resurse -slike, izglede, stilove koji se odvajaju od ostatka koda. U folderu *layout* se nalaze resursi za definiranje izgleda aplikacije napisani XML opisnim jezikom. Svaki *layout* ima korijenski element koji definira orijentaciju i raspored elemenata koji su u njega ugniježđeni. Ugniježđeni elementi mogu biti tipa *Button*, *TextView*, *EditText*, *ImageView*, itd.

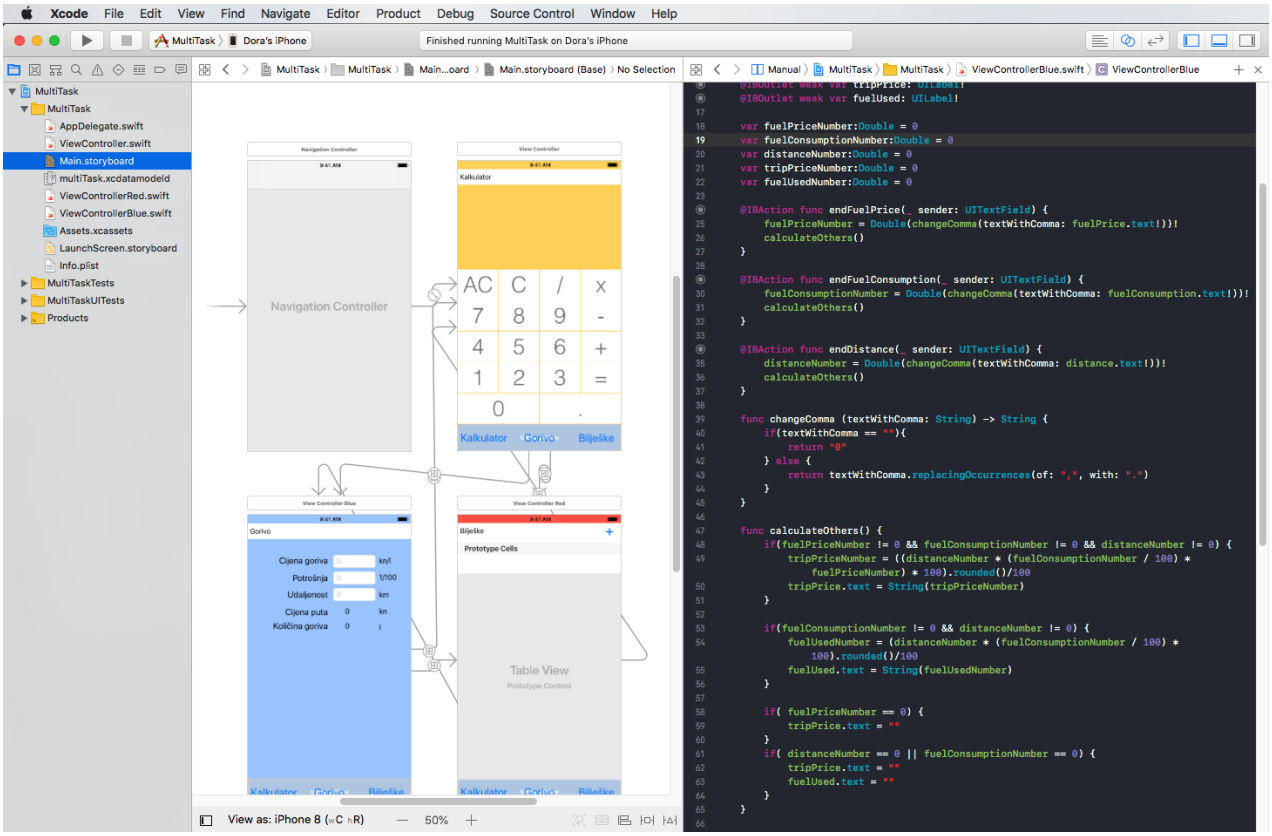
Svaki od tih elemenata ima različite karakteristike i uporabu, a njihovo se ponašanje i izgled mogu definirati dodatnim parametrima. Izgled gotove Android aplikacije prikazan je na slici 5.5.



Slika 5.5. Tri glavna ekrana Android aplikacije na Samsung Galaxy S6 mobilnom uređaju

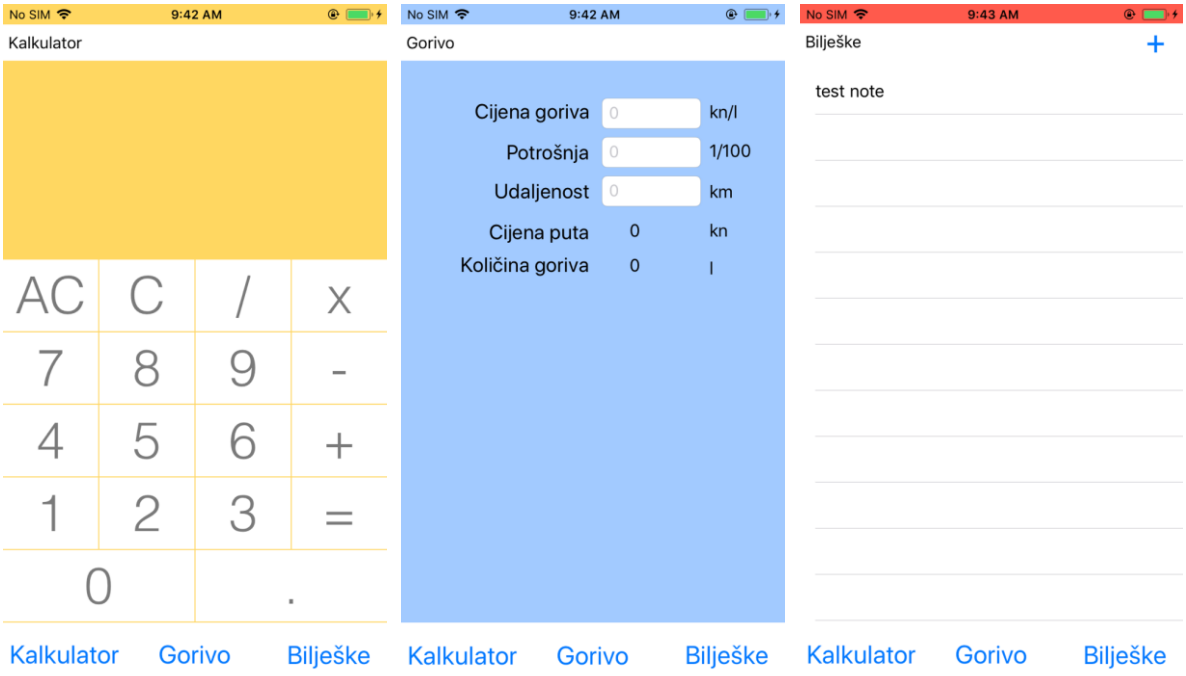
5.2.2 iOS

Razvoj nativne aplikacije za iOS odvija se u razvojnom okruženju Xcode. Xcode je IDE za macOS i omogućuje razvoj aplikacija za macOS, iOs, watchOS i tvOS. Podržava razvoj u C, C++, Objective-C, Swiftu i još nekim programskim jezicima. Omogućuje pisanje i uređivanje koda, sadrži katalog dostupnih elemenata za izradu izgleda aplikacije, dolazi sa simulatorom bilo kojeg Apple mobilnog uređaja i moguće je skaliranje verzije operativnog sustava na simuliranom uređaju te ima integriran GitHub [33]. Xcode razvojno okruženje prikazano je na slici 5.6., a kod je pisan u Swift programskom jeziku.



Slika 5.6. Xcode na macOS-u

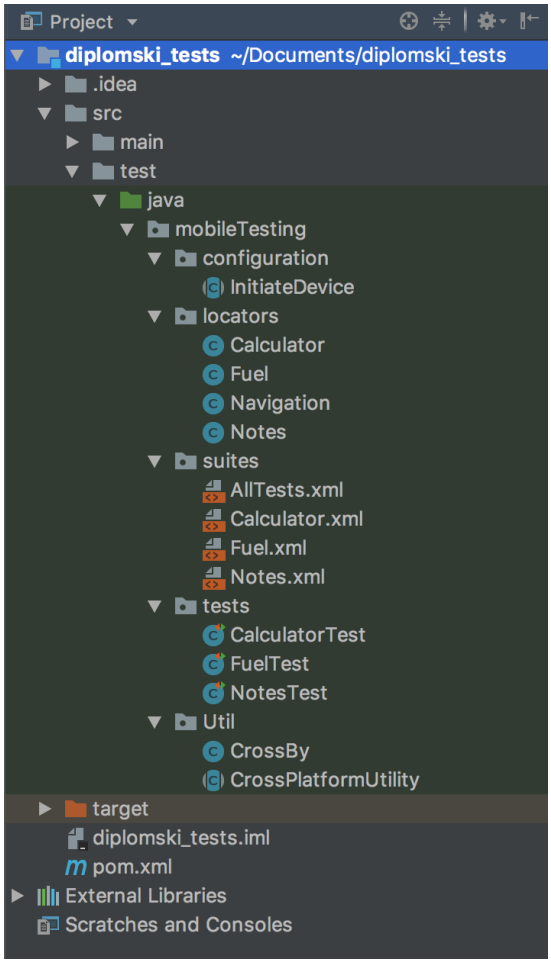
Xcode projekt je repozitorij svih dokumenata i resursa potrebnih za izradu aplikacije. AppDelegate.swift dokument sadrži metode koje se pozivaju kada dođe do neke promjene na uređaju (npr. dolazak notifikacije ili poziva). ViewController.swift upravlja zadanim ekranom, a svaki ekran ima jedan ili više svojih kontrolera. Main.storyboard je vizualna reprezentacija izgleda aplikacije i prikazuje povezanost različitih scena, odnosno ekrana. Kroz Main.storyboard se mogu dodavati elementi ekranu; na primjer label, button, text field, slider, switch… Ti se elementi na jednostavan način mogu povezati s metodom u kontroleru tako da se označi element i pritisne tipka *control* na tipkovnici. Tada se pojavi poveznica koja se može dovući do djela koda kontrolera gdje se otpusti. Na tom se mjestu pojavi izbornik u kojem se može odabrati što se želi učiniti s elementom. Moguće je stvoriti varijablu ili metodu vezanu uz taj element. Datoteka Main.storyboard sadrži izgled aplikacije i povezanost ekrana, a može se vidjeti na lijevoj strani slike 5.6. Xcode projekt MultiTask ima tri kontroler datoteke. Svaki kontroler upravlja radom jednog ekrana. Izgled gotove iOS aplikacije prikazan je na slici 5.7.



Slika 5.7. Tri glavna ekrana aplikacije na iPhone 8 mobilnom uređaju

5.3 Istovremeno testiranje na različitim platformama

Prilikom stvaranja novog projekta u IntelliJ-u potrebno je odabrati Maven i Java verziju (u ovom je projektu korištena Java 10.0.1) nakon čega se dobiva prazan projekt. Kako bi se jedan test mogao izvršavati na obje platforme, potrebno je napraviti vlastiti *framework* upotrebom i nadogradnjom postojećih metoda. Izgled gotovog projekta i njegovih datoteka i dokumenata prikazan je na slici 5.8.



Slika 5.8. Struktura projekta za testiranje

Paket configuration sadrži datoteku InitiateDevice unutar koje se konfigurira testiranje. Prije pokretanja testova potrebno je postaviti željene sposobnosti (engl. *Desired Capabilities*) koje su ključne riječi i vrijednosti koje Appium klijent šalje serveru, a time se opisuje sjednica (engl. *session*), korišteni uređaji, operativni sustavi, itd. Iz razloga što se testovi izvršavaju na dvije različite platforme potrebno je navesti drugačije vrijednosti za svaku od njih. Postavljanje *Desired Capabilities*-a za obje platforme prikazano je u programskom kodu 5.1.

```
@BeforeSuite(alwaysRun = true)
   public void setUp() throws Exception {
      DesiredCapabilities capabilities = new DesiredCapabilities();
      switch (platform) {
         case IOS:
            capabilities.setCapability("deviceName", "iPhone 8");
            capabilities.setCapability("udid", UDID);
            capabilities.setCapability("platformName", "iOS");
            capabilities.setCapability("automationName", "XCUITest");
            capabilities.setCapability("bundleId", "dfundak.MultiTask");
            iosDriver = new IOSDriver(new URL("http://0.0.0.0:4723/wd/hub"), capabilities);
            break;
         case ANDROID:
            capabilities.setCapability("deviceName", "Galaxy S6");
            capabilities.setCapability("platformName", "ANDROID");
            capabilities.setCapability("appPackage", "com.example.dorafundak.multitask");
            capabilities.setCapability("appActivity", "MainActivity");
            androidDriver = new AndroidDriver(new URL("http://0.0.0.0:4723/wd/hub"),
          capabilities);
            break;
      }
}
```

Programski kod 5.1. Dio koda u kojem se inicijalizira driver

Naredba @BeforeSuite je TestNG naredba [34] koja označava da će funkcija označena tom naredbom biti izvršena prilikom svakog pokretanja, prije svih testova. Postoji nekoliko takvih naredbi koje označavaju pokretanje prije ili poslije određene klase, metode ili grupe testova. Kroz cijeli se projekt upotrebljava switch-case Java naredba kako bi se provjerilo na kojem se operativnom sustavu test izvodi i kako bi se izvršile prikladne naredbe. Postoji statična varijabla *platform* u koju se pohranjuje odabrana platforma odabirom enum varijabla koja ima dvije vrijednosti IOS i ANDROID. U navedenom se kodu, prilikom izvršavanja testova, serveru šalju podatci o korištenom uređaju (iPhone 8 ili Galaxy S6), ime platforme, *bundleId* i *appPackage* koji imaju sličnu funkciju - jedinstvena identifikacija aplikacije na uređaju. UDID je jedinstvena

oznaka iOS mobilnog uređaja. Postoji još mnogo dodatnih oznaka koje se mogu poslati serveru, a mogu se pronaći u literaturi [35].

Pomoću Appium Inspector-a može se dobiti lista svih elemenata prikazanih na ekranu aplikacije u hijerarhijskom XML obliku (primjer na slici 4.4.). Elementi mogu imati oznake: class name, id, name, value, XPath,… Više elemenata na ekranu mogu imati jednak value, id ili name, a XPath je jedinstvena oznaka. Upotreba XPath-a se ne preporučuje osim ako se ne može na drugi način pristupiti elementu jer je takav način pristupanja najsporiji od svih ostalih. Mapa *locators* sadrži klase koje u sebi sadrže oznake elemenata koje su dobivene pregledom ekrana Appium Inspector-om. Android i iOS imaju drugačiji način označavanja elemenata pa se zbog toga koristi CrossBy klasa. Oznake elemenata se pohranjuju u obliku prikazanom unutra programskog koda 5.2.

```
By priceField = CrossBy.id("FuelPrice", "com.example.dorafundak.multitask:id/fuelPrice");
```

Programski kod 5.2. Oznaka elementa

gdje je prva oznaka za iOS, a druga za Android. CrossBy metoda je napisana kako je prikazano u programskom kodu 5.3.

```
public static By id(String IOSId, String androidId) {
    switch (getPlatform()) {
        case IOS:
            return By.id(IOSId);
        case ANDROID:
            return By.id(androidId);
        default:
            return null;
    }
}
```

Programski kod 5.3. CrossBy metoda u klasi

Kad se pozove određeni element, u ovom slučaju priceField, vrati se oznaka za trenutnu platformu. Na taj se način olakšava pohrana oznaka elemenata i njihovo pozivanje.

Sve akcije koje se mogu izvoditi nad elementima postoje u sklopu IOSDriver-a i AndroidDrivera, a specifične su za platformu i zahtijevaju upotrebu drivera te platforme. Zbog toga je potrebno napisati metode za često upotrebljavane akcije kako bi se omogućilo jednostavno pisanje testova za obje platforme. Te se metode nalaze u klasi CrossPlatformUtility, a neki od

metoda su: getElement, locateElement, sendKeys, getElementText itd. Kod getElement metode prikazan je u programskom kodu 5.4.

```
public static WebElement getElement(By identifier) {
   switch (getPlatform()) {
      case IOS:
       new WebDriverWait(getIOSDriver(), time)
                .until(ExpectedConditions.visibilityOfElementLocated(identifier));
           return getIOSDriver().findElement(identifier);
      case ANDROID:
       new WebDriverWait(getAndroidDriver(), time)
                .until(ExpectedConditions.visibilityOfElementLocated(identifier));
           return getAndroidDriver().findElement(identifier);
}
```

Programski kod 5.4. getElement metoda

Navedena metoda vraća traženi element. *WebDriverWait* čeka zadano vrijeme *time* i provjerava je li traženi element prikazan na ekranu bez vraćanja greške unutar tog vremenskog okvira. Kad zadano vrijeme prođe, ako element nije pronađen na ekranu, vraća grešku. Kad bi se upotrebljavala samo findElement naredba, bez prethodnog čekanja, testovi bi često padali jer se ekran ne bi stigao učitati, a test bi to protumačio kao da se element ne nalazi na zadanom ekranu.

Vidljivo je da su naredbe za obje platforme jednake i metoda vraća jedan WebElement, ali je potrebno upotrebljavati driver za odabranu platformu. U ovom se slučaju može upotrebljavati WebElement jer iOS i Android driveri proširuju (engl. *extend*-aju) WebElement klasu. Neke od metoda koje su jednake na oba drivera preko WebElementa su: click, sumbit, sendKeys, isEnabled, getText itd. Isto tako, postoje naredbe koje su karakteristične za platformu i zbog toga je poželjno razdvojiti driver odmah na početku, a ne upotrebljavati WebDriver koji ne sadrži metode specifične za mobilne uređaje, kao na primjer naredbu *swipe*.

5.3.1 Testovi

Testove je nabolje podijeliti u logičke skupine po dijelu aplikacije koju testiraju ili po vrsti testova. U projektu su testovi podijeljeni na ekrane na koje se testovi odnose, pa postoje tri klase s testovima: CalculatorTest, FuelTest i NotesTest. Za svaki se ekran u aplikaciji izvodi integracijsko testiranje elemenata. Primjer testa prikazan je u programskom kodu 5.5.

```
@Test(description = "Addition test 72+3=75")
   public void additionTest() {
      locateElementClick(Calculator.button7);
      locateElementClick(Calculator.button2);
      verifyElementContainsText(Calculator.currentNumberDisplay, "72");
      locateElementClick(Calculator.plusButton);
      verifyElementContainsText(Calculator.lastNumberDisplay, "72");
      verifyElementContainsText(Calculator.currentNumberDisplay, "+");
      locateElementClick(Calculator.button3);
      verifyElementContainsText(Calculator.currentNumberDisplay, "3");
      verifyElementContainsText(Calculator.lastNumberDisplay, "72");
      locateElementClick(Calculator.equalsButton);
      verifyElementContainsText(Calculator.lastNumberDisplay, "75");
   }
```

Programski kod 5.5. Test provjere operacije zbrajanja

Test naveden u programskom kodu 5.5. se izvršava na „Kalkulator“ ekranu, a provjerava ispravnost gumbova i ispisa rezultata. Metoda *locateElementClick(Calculator.button7)* na ekranu locira gumb broja sedam i klikne ga. Metoda: *verifyElementContainsText(Calculator.currentNumberDisplay, "72")*na ekranu locira prostor za prikaz brojeva i provjerava sadrži li taj prostor ispravnu vrijednost. Kad bi ispis bio kriv, na primjer samo „7“, došlo bi do greške i test ne bi prošao.

Testovi se mogu izvršavati jedan za drugim tako što se grupiraju u grupe testova zvane *suite*. *Suite* se definira u XML datoteci kao što je prikazano na primjeru programskog koda 5.6.

```
<suite name="Notes">
   <test name="Notes Test">
      <classes>
         <class name="mobileTesting.tests.NotesTest">
            <methods>
               <include name="addNote"/>
               <include name="deleteNote"/>
               <include name="cancelAddNote"/>
            </methods>
         </class>
```

```
    </classes>
  </test>
</suite>
```

Programski kod 5.6. Test suite testova za ekran „Bilješke“

Kad se pokrene *suite*, izvršavaju se svi testovi s popisa jedan za drugim bez prestanka ili ikakvog potrebnog unosa. *Suite* bi trebao sadržavati testove povezne logičkim smislim. Na primjer, može sadržavati sve testove koji provjeravaju korisničko sučelje ili sve testove koji provjeravaju ispravnost prikazanih podataka (engl. *data integrity*). Kod takvog načina izvršavanja testova dolazi do izražaja korisnost @Before naredbi. Te se naredbe uglavnom upotrebljavaju kako bi se aplikacija dovela u željeno stanje prije početka testa. U slučaju testa iz 5.6. primjera, bitno je da se aplikacija nalazi na ekranu „Notes“ pa se upotrebljava oznaka @BeforeClass za metodu before() unutra koje se prije izvršavanja bilo kojeg testa iz klase NotesTest navigira do ekrana „Bilješke“.

Kao što se i testovi mogu grupirati u *suite*, može se i napraviti *suite* od drugih *suite*-ova što je prikazano u programskom kodu 5.7.

```
<suite name="All tests">
  <suite-files>
    <suite-file path="Calculator.xml"></suite-file>
    <suite-file path="Fuel.xml"></suite-file>
    <suite-file path="Notes.xml"></suite-file>
  </suite-files>
</suite>
```

Programski kod 5.7. Test suite svih suite-ova

Ako su svi testovi dobro predefinirani i napisani tako da ne ovise o drugim testovima, *suite* sa svim testovima je izvrstan način za testiranje cijele aplikacije.

5.3.2 Paralelno testiranje

Testovi su napisani tako da se mogu izvršavati na različitim mobilnim operativnim sustavima, a kako bi se uštedilo vrijeme, testovi mogu biti istovremeno pokrenuti na nekoliko različitih uređaja. Uređaji ne moraju biti istog tipa sustava. Paralelno izvršavanje testova se postiže tako da se Appium server pokrene u terminalu na određenom portu za određeni uređaj, a to je prikazano u programskom kodu 5.8.

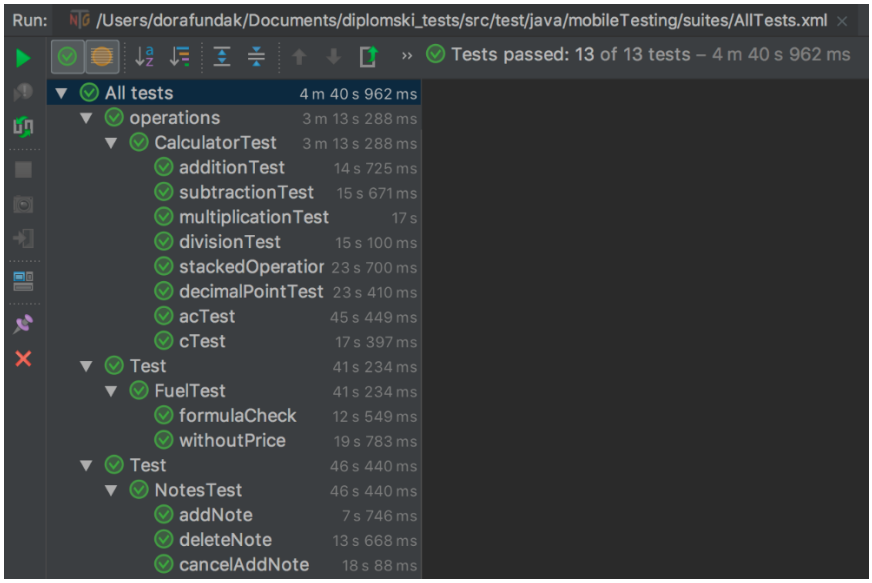
```
$ appium -U [UDID] -p 4723
```

Programski kod 5.8. Naredba za pokretanje Appium servera

Za pokretanje drugog servera, za drugi uređaj, potrebno je otvoriti novu konzolu i istom naredbom pokrenuti server na neki drugi port s identifikacijom drugog uređaja. Te se promjene porta trebaju napraviti i u *DesiredCapabilities* postavkama. Nakon toga se testovi mogu pokrenuti na oba uređaja i izvršavati neovisno jedni od drugih.

5.3.3 Izvješća testiranja

Nakon izvršavanja testova potrebno je pogledati i spremiti rezultate testiranja. Test može proći (engl. *pass*), pasti (engl. *fail*) ili biti preskočen. IntelliJ ima integrirano izvještavanje o uspješnosti testa. Na slici 5.9. je prikazano izvješće nakon što su se izvršili svi testovi navedeni u programskom kodu 5.7.



Slika 5.9. Rezultat AllTests *suite-a*

Prolazak testa znači da aplikacija zadovoljava sve uvjete zadane testom. Pad testa znači da aplikacija ne prati predviđen tok događanja ili da neki elementi ne zadovoljavaju uvjete. IntelliJ pad prikazuje kao grešku u kodu i ispisuje točno mjesto na kojemu je došlo do greške.

6 ZAKLJUČAK

Mobilni uređaji su sveprisutni i bitno je osigurati da aplikacije za njih rade bez grešaka. Da bi se osigurala kvaliteta aplikacije, aplikaciju je potrebno testirati, a najsigurniji način testiranja je konstantno testiranje i provjeravanje. Konstantno testiranje se može izvoditi ručno, ali je bolje rješenje automatizacija testiranja. Jednom napisani, automatizirani se testovi mogu konstantno izvršavati tijekom svih faza razvoja aplikacije. Kod aplikacija koje imaju verzije za različite operativne sustave, testiranje zahtjeva puno više vremena jer je potrebno testirati obje aplikacije na različitim uređajima. Rješenje tog problema je višeplatformsko testiranje. Ideja iza takvih testova je napisati jedan test koji se uz pomoć alata (u ovom slučaju Appiuma) i napisanih testnih okvira može izvršavati neovisno o operativnom sustavu uređaja. Kako bi se ubrzao proces testiranja, moguće je istovremeno testirati više uređaja – paralelno testiranje. Kad se svi testovi izvrše, dobije se izvješće o uspješnosti testiranja.

LITERATURA

[1] Softpedia, https://news.softpedia.com/news/Did-You-Know-The-First-Modern-Smartphone-was-the-IBM-Simon-470537.shtml (lipanj, 2018.)

[2] Deloitte, State of the smart, Consumer and business usage patterns, Global mobile Consumer Survey 2017: UK Cut, https://www.deloitte.co.uk/mobileuk/assets/img/download/global-mobile-consumer-survey-2017_uk-cut.pdf (lipanj, 2018.)

[3] Deloitte, Global mobile consumer trends, 2nd edition, https://www2.deloitte.com/content/dam/Deloitte/us/Documents/technology-media-telecommunications/us-global-mobile-consumer-survey-second-edition.pdf (lipanj, 2018.)

[4] StatCounter, http://gs.statcounter.com/os-market-share/mobile/worldwide (lipanj, 2018.)

[5] D., Knott, Hands-on mobile app testing : a guide for mobile testers and anyone involved in the mobile app, Pearson Education, Inc, United States of America, 2015.

[6] Developer Android, https://developer.android.com/about/dashboards/ (lipanj, 2018.)

[7] M., Haris, B., Jadoon, M., Yousaf, F. H., Khan, Evolution of Android operating system: a review, 2nd International Conference on Advanced Research, Australia, 2017.

[8] Apple Developer, https://developer.apple.com/support/app-store/ (lipanj, 2018.)

[9] The Interaction Design Foundation, https://www.interaction-design.org/literature/article/the-anatomy-of-a-smartphone-things-for-designers-to-consider-for-mobile-development (lipanj, 2018.)

[10] Appdynamics, The App Attention Indeks, A 2017 study examining the impact of app performance on consumer behavior and business outcomes, http://docplayer.net/67929934-The-app-attention-index-a-2017-study-examining-the-impact-of-app-performance-on-consumer-behavior-and-business-outcomes.html (lipanj, 2018.)

[11] N., Verma, Mobile Test Automation with Appium, Packt Publishing Ltd., Mumbai, 2017.

[12] D., Graham, E., van Veenendaal, I., Evans, R., Black, Foundations of software testing, ISTQB certification, International Thomson Business Press, London, 2006.

[13] Ask the Standards Experts, https://asqasktheexperts.com/2012/06/12/qualification-verification-and-validation/ (lipanj, 2018)

[14] ISO/IEC 25010:2011, https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (lipanj, 2018)

[15] R., Patton, Software Testing, Sams Publishing, Indiana, 2001.

[16] Software testing revealed, Drugo izdanje, https://www.test-institute.org/Software_Testing_Books_International_Software_Test_Institute.php (lipanj, 2018)

[17] G., Myers, T., Badgett, C., Sandler, The art of software testing, Treće izdanje, John Wiley & Sons, Inc., Kanada, 2011.
[18] SDLC - Waterfall Model, https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (lipanj, 2018.)
[19] What is Functional Testing? Types, Tips, Limitations & More, https://stackify.com/functional-testing-types-tips-limitations/ (lipanj, 2018)
[20] What is usability testing?, https://www.experienceux.co.uk/faqs/what-is-usability-testing/ (lipanj, 2018.)
[21] H., Q., Nguyen, M., Hackett, B., K., Whitelock, Happy about global software test automation, Happy about, Kalifornija, 2006.
[22] SeleniumHQ, https://www.seleniumhq.org/docs/01_introducing_selenium.jsp (lipanj, 2018)
[23] Application program interface (API), https://searchmicroservices.techtarget.com/definition/application-program-interface-API (lipanj, 2018)
[24] Selendroid, http://selendroid.io/ (lipanj, 2018)
[25] ios-driver https://ios-driver.github.io/ios-driver/ (lipanj, 2018)
[26] Appium, http://appium.io/ (lipanj, 2018)
[27] Appium Desktop, https://github.com/appium/appium-desktop (lipanj, 2018)
[28] JUnit - Overview, https://www.tutorialspoint.com/junit/junit_overview.htm(lipanj, 2018)
[29] JUnit vs. TestNG: Which Testing Framework Should You Choose?, https://dzone.com/articles/junit-vs-testng-which-testing-framework-should-you (lipanj, 2018)
[30] Genymotion, https://www.genymotion.com/desktop/ (lipanj, 2018)
[31] Samsung Note 8 Layout, https://mobil1tech.blogspot.com/2018/02/samsung-note-8-sm-n950u-pcb-layout.html?view=flipcard (kolovoz, 2018)
[32] Developer Android, https://developer.android.com/studio/intro/ (rujan, 2018.)
[33] Xcode IDE, Apple Developer https://developer.apple.com/xcode/ide/ (rujan, 2018.)
[34] TestNG, http://testng.org/doc/documentation-main.html (rujan, 2018.)
[35] Appium, capabilities, http://appium.io/docs/en/writing-running-appium/caps/#general-capabilities (rujan, 2018.)
[36] Appium driver, https://discuss.appium.io/t/what-is-the-use-or-difference-between-androiddriver-iosdriver-appiumdriver-and-remote-webdriver/8750/2 (rujan, 2018.)

SAŽETAK

Testiranje je bitan dio razvoja mobilnih aplikacija. Kako bi se skratilo vrijeme testiranja, a povećala pouzdanost, predlaže se automatizacija testiranja. Automatizacija testiranja se provodi pisanjem skripti (testova) koje provjeravaju različite aspekte aplikacije. U radu je prikazana automatizacija testiranja iste aplikacije napravljene za dva različita operativna sustava. Napisan je *framework*, upotrebom i nadogradnjom postojećih metoda, koji podržava istovremeno testiranje na različitim platformama.

Ključne riječi: Android, automatizacija testiranja, iOS, mobilne aplikacije, QA

ABSTRACT

Mobile applications test automation

Testing is an important part of mobile application development. Test automation is recommended to shorten the testing time and to increase reliability. Test automation is done by writing scripts (tests) designed to verify different aspects of the application under test. Automation of an application written for two different operating systems is shown in this paper. The framework was written, using existing methods and extending them, which supports simultaneous testing on different platforms.

Key words: Android, test automation, mobile applications, iOS, QA

ŽIVOTOPIS

Dora Fundak rođena je 30. siječnja 1995. godine u Osijeku. Završila je osnovnu školu Miroslava Krleže u Čepinu nakon koje je upisala III. gimnaziju Osijek. Tijekom srednje škole sudjelovala je na županijskim natjecanjima iz informatike, Hrvatskim otvorenim natjecanjima u informatici (HONI) i ACSL All-Star natjecanjima. Maturirala je 2013. godine i upisala preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. 2016. godine završava preddiplomski studij elektrotehnike te upisuje diplomski studij na istom fakultetu. Opredjeljuje se za smjer komunikacije i informatika, izborni blok mrežne tehnologije.

_____

Dora Fundak

PRILOZI

1. Programski kod Android mobilne aplikacije
2. Programski kod iOS mobilne aplikacije
3. Programski kod testova