LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ANDROID MOBILO LIETOTŅU TESTĒŠANAS AUTOMATIZĀCIJA

BAKALAURA DARBS

Autors: **Pāvels Lazučonoks**
Studenta apliecības Nr.: pl12005
Darba vadītāja: Dr.dat. Zane Bičevska

RĪGA 2016

ANOTĀCIJA

Kopā ar mobilo lietotņu skaitu, pieaug arī testētāju darbs, tāpēc bieži vien testētāji izvēlās automatizēt daļu no testēšanas procesa, tāpēc, šī darba mērķis ir apskatīt Android mobilo lietotņu testu automatizācijas iespējas un alternatīvas, lai izvēlētos labāko Android lietotņu testu automatizācijas rīku.

Teorētiskajā darba daļā ir aprakstīti mobilo lietotņu melnās kastes būtiskākie paveidi, to specifika. Apkopota informācija par testu automatizāciju un apskatīti lietotāja saskarnes testu automatizācijas rīku veidi un definēti to izvēles kritēriji.

Praktiskajā daļa detalizēti tika apskatīti četri populārākie Android mobilo lietotņu testēšanas rīki, un veikta rīku analīze uz konkrēta piemēra, salīdzinātas testu automatizācijas iespējas, testu izpildes ātrums un izvēlēto rīku trūkumi. Gala rezultātā tika apkopota iegūta informācija par rīkiem un izvēlēts labākais.

Atslēgvārdi: testēšana, mobilās lietotnes, Android, automatizācija, automatizēta testēšana.

ABSTRACT

Android mobile application test automation

Along with the count of mobile devices grows the work amount of software testers. Because of that software testers often choose to automate some parts of testing process. The goal of this paper is to analyze possibilities and alternatives within automated Android mobile application testing and choose the best test automation tool for Android applications.

The theoretical part this work describes the most important types of black box mobile testing and their specifics. Author summarizes information about test automation, examines user interface test automation tool types and defines selection criteria for these tools.

In practical part four most popular Android mobile testing tools have been reviewed and analyzed on a specific example. Test automation possibilities, test execution speed and flaws of the chosen tools are compared. Finally, all gathered information of this has been summarized and the best test automation tool has been chosen.

Keywords: testing, mobile apps, Android, automation, automated testing.

SATURA RĀDĪTĀJS

APZIMĒJUMI UN SKAIDROJUMI

OS – operētājsistēma, jeb programmu komplekss, kas vada datu organizēšanu un programmu izpildi datorā, nodrošina aparatūras un programmatūras kopdarbību, resursu racionālu izmantošanu, kā arī sadarbību ar lietotāju

SDK - palīgprogrammu (rutīnu) kopa, kas lietojumprogrammu izstrādātājam atvieglo programmu veidošanu, ievērojot konkrētās to darbības vides īpatnības

GUI - Displeja formatēšanas veids, kas ļauj lietotājam izvēlēties komandas, palaist programmas, kā arī apskatīt datņu sarakstus un citus objektus, norādot to piktogrāfiskos attēlus (ikonas)

GPS - ASV pavadoņu navigācijas sistēma

NFC - protokolu kopums, kas ļauj sūtīt datus starp mobilajām ierīcēm tuvā attālumā (līdz 10 cm) ar radiosakaru starpniecību

Bluetooth - maza darbības rādiusa bezvadu datortīklu standarts

OCR - optiska rakstzīmju pazīšana, grafisku rakstzīmju atpazīšana ar optiskiem līdzekļiem.

CSS - īpaša stila lapas valoda, ko lieto, lai aprakstītu izskatu iezīmēšanas valodā veidotiem dokumentiem

XPath - XML Path Language - XML dokumentu elementu adresācijas valoda

ID – unikāls objekta vai personas identifikators.

WP – Windows Phone.

C# - kompānijas Microsoft izstrādāta daudzparadigmu programmēšanas valoda.

JS – JavaScript - skriptu valoda, kas ļauj globālā tīmekļa izstrādātājiem veidot interaktīvas vietnes.

PHP - programmēšanas valoda, kas galvenokārt tiek izmantota dažādu tīmekļa aplikāciju izstrādei un uzturēšanai.

API – application programming interface - lietojumprocesos izmantojama pilna operētājsistēmas funkciju specifikācija, kā arī šo funkciju izmantošanas procedūru apraksts.

USB - ārējās kopnes standarts, kas nodrošina datu pārraides ātrumu 12 megabiti sekundē.

PNG - bitkartētu grafikas datņu formāts.

VNC – attālinātas piekļuves sistēma, kas izmanto RFB protokolu.

TestNG - testēšanas satvars lietotnēm, kas rakstītas Java programmēšanas valodā.

JUnit - testēšanas satvars lietotnēm, kas rakstītas Java programmēšanas valodā.

APK – Android lietotņu pakotnes datņu formāts.

IEVADS

Mobilie telefoni cilvēcei ir pazīstami jau kopš 1980. gadu sākuma. Kopš tā laika, protams, ierīces ir krietni mainījušās, tomēr lielākas pārmaiņas nāca 2007. gadā, kad *Apple* prezentēja savu pirmo *Apple iPhone* mobilo viedtālruni. Brīdi, kad Stīvs Džobs prezentēja jaunu *iPhone,* daudzi cilvēki uzskata par vēsturisku, jo tieši šajā momentā tika noteikts jauns mobilo ierīču standarts, kas tiek joprojām izmantots. Un, sākot no tā brīža, mobilo viedtālruņu tirgus sāka tikai augt [1].

Tagad, astoņus gadus vēlāk, skārienjūtīgie viedtālruņi un planšetdatori ir kļuvuši par visuresošiem, gandrīz katram cilvēkam, sākot no bērna un beidzot ar vecāku paaudzi, lietojumā ir kaut viena skārienjūtīga „gudra" ierīce. Vairāk nekā 2.5 miljoni mobilo lietotņu šo gadu laikā tika izstrādāti un ir pieejami lejuplādēšanai no dominējušo ierīču lietotņu veikaliem: *Android Play Store* – 1,6+ miljoni, *iOS* store – 1,5+ miljoni, *WP Store* – 340 tūkstoši [2], un šīs skaitlis tikai aug.

Kopā ar mobilo lietotņu skaitu palielināšanos, palielinās arī konkurence starp lietotņu izstrādātajiem, tāpēc mobilo lietotņu izstrādātāji mēģina nodrošināt labu lietotnes kvalitāti. Līdz ar to, palielinās mobilo lietotņu testētāju darbs un kvalitātes pārbaudes process paliek apjomīgāks, un sarežģītāks. Arvien grūtāk un ilgāk paliek pārbaudīt mobilo lietotņu atbilstību kvalitātes prasībām manuāli, tāpēc daudzi mobilo lietotņu testētāji testēšanas procesu pēc iespējas mēģina automatizēt.

Pamatā eksistē divas automatizētas testēšanas pieejas: testēšana koda līmenī un testēšana lietotāja saskarnes līmenī, jeb GUI līmeņa testēšana. Pie pirmā līmeņa attiecas vienībtestēšana, kad tiek pārbaudītas koda atsevišķo daļu funkcionalitāte, bet otrajā gadījumā tiek pārbaudīts gatavas lietotnes funkcionalitāte caur lietotāja saskarni. Šajā darbā lielāka uzmanība tiks veltīta tieši otrai testu automatizācijas pieejai, kad tiek imitēta gatavas mobilās lietotnes lietošana un tiek pārbaudīta tās funkcionalitāte.

Veicot GUI līmeņa funkcionalitātes testēšanu, tiek imitēti reāli lietotnes darbības gadījumi ar mērķi pārbaudīt vai lietotnes funkcionalitāte atbilst prasībām – tiek salīdzināts paredzamais rezultāts ar esošo. Tiek izveidoti testa gadījumi, kas atbilst lietotnes reālajiem lietošanas gadījumiem. Var teikt, ka lietotne tiek testēta tā kā to lietos potenciālais lietotnes lietotājs, tāpēc veicot lietotnes testēšanu GUI līmenī var novērtēt lietotnes kvalitāti no reāla lietotāja skata.

Tā, ka Android mobilā operētājsistēmā ir visvairāk izplatīta pasaulē un Android lietotņu skaits ir lielāks par citu mobilo opērētājsistēmu lietotņu skaitu, darba autors ir nolēmis izpētīt tieši Android mobilo lietotņu testu automatizācijas metodes un iespējas, tāpēc darbā gaitā tiks apskatīta un izanalizēta informācija par mobilo lietotņu automatizēto testēšanu, kā arī veikts pētījuma par Android mobilo lietotņu testu automatizācijas rīkiem un satvariem.

Darbs sastāv no sekojošam daļām:

Pirmajā nodaļā tiek apskatīti mobilo lietotņu veidi – konkrētas platformas, tīmekļa un hibrīda lietotnes, tiek apkopoti katra mobilas lietotnes veida priekšrocības un trūkumi.

Otrajā nodaļā ir apkopoti mobilo lietotņu testēšanas paveidi, tādi, kā funkcionāla testēšana, lietojamības testēšana, baterijas patēriņa testēšana, veiktspējas testēšana, stresa un pārtraukumu testēšana, uzstādīšanas, atinstalēšanas un atjaunināšanas testēšana, to specifika. Kā arī izpētīts vai iespējams šos paveidus automatizēt.

Trešajā darba nodaļā tiek definēts kas ir automatizēta testēšana kopumā, tas pielietošanas priekšrocības un trūkumi, kā arī definēts ko un kad ir jāautomatizē testējot mobilās lietotnes.

Ceturtajā nodaļā ir apskatīts, kas ir mobilo lietotņu grafiskās lietotāja saskarnes rīki kopumā, to veidi pēc lietotāja saskarnes elementu atpazīšanas pieejas, kā arī definēti automatizācijas rīku izvēles kritēriji.

Piektajā darbā daļā tiek apskatīti reāli Android mobilo lietotņu funkcionālas testēšanas rīki un to pielietojums uz konkrēta piemēra. Tiek apkopota informācija par katra rīka testu automatizācijas iespējam, noteikts iepriekšdefinētas testu kopas izpildes laiks, definēti rīku trūkumi un konstatētas problēmas, galā tika izvēlēts piemērotākais Android mobilo lietotņu testu automatizācijas rīks. Kā arī tika piedāvāti mobilo lietotņu baterijas testu un stresa un patraukuma testu automatizācijas rīki.

Pielikumā ir pievienoti testu automatizācijas rīkos autora izveidoti testu skripti.

1. MOBILO LIETOTŅU VEIDI

Pirms uzsākt mobilo lietotņu testēšana, manuālo vai automatizēto, testētājam ir jāzin, kāda veida lietotne viņam jātestē, jo atkarība no lietotnes veida , ir jāizvēlas piemērotas testēšanas metodes un rīki.

Mobilā lietotne ir lietojumprogrammatūra, kas darbojas viedtālruņos, planšetdatoros, e-lasītājos, un citās gudras ierīcēs, kuram ir sava operētājsistēma [3]. Izšķir trīs mobilo lietotņu veidus, kuri atšķiras pēc izstrādes veida:

- Konkrētas platformas lietotnes
- Tīmekļa lietotnes
- Hibrīda lietotnes

Šajā sadaļā autors dod priekšstatu par katru lietotnes veidu, kā arī apkopos to trūkumus un priekšrocības.

1.1. Konkrētas platformas lietotnes

Konkrētas platformas lietotnes tiek programmētas kādas platformai specifiskā programmatūras izstrādes komplektā, izmantojot specifisku programmēšanas valodu, un ir paredzētas vienai konkrētai mobilajai platformai. Piemēram, lietotnes, kas ir paredzētas *Android* tiek programmētas lietojot *Java* vai *C++* programmēšanas valodas, *iOS* lietotnes tiek programmētas lietojot *Objective-C* vai *Swift,* un *C#, Visual Basic* vai *C++* tiek izmantots priekš *Windows Phone* lietotnēm [4].

Konkrētas platformas lietotnēs var viegli pieslēgt un izmantot ierīces resursus un aprīkojumu – *GPS, NFC, Bluetooth,* kompasu un citus. Kā arī šīs lietotnēs var viegli piekļūt pie kontaktu sarakstu, ierīces galerijas vai multivides failiem. Bieži vien šīs lietotnes ir labi optimizētas mobilām platformām.

Konkrētas platformas lietotņu trūkumus un priekšrocības var redzēt tabulā 1.1.

Tabula 1.1.

Konkrētas platformas lietotnes trūkumi un priekšrocības

Priekšrocības	Trūkumi
laba veiktspēja, pateicoties labai optimizācijai	diezgan paliela izstrādes cena
datu uzglabāšana bezsaistē	ilgs apstiprināšanas process
labas lietojamības nodrošināšana,	katrai platformai ir jāizstrādā sava lietotne lietojot citu programmēšanas valodu
specifiskas platformas aprīkojumu un programmatūru atbalsts	

1.2. Tīmekļa lietotnes

Tīmekļa lietotnes ir mājaslapa, kura ir pieejama no mobilās ierīces pārlūka. Mājaslapa tiek speciāli optimizēta priekš mobilās ierīces pārlūka un ir neatkarīgas no mobilās ierīces platformas [5].

Šī veida lietotnes tiek izstrādātas izmantojot tradicionālas tīmekļa izstrādes tehnoloģijas – *HTML, CSS, JavaScipt*, kā arī servera puse tiek izstrādāta lietojot *Node.js, PHP, ASP.net* vai citas tīmekļa tehnoloģijas.

Tīmekļa lietotnes var izmantot lietotnes kameru, GPS vai akselerometru. Šīs lietotnes netiek instalētas uz mobilo ierīci un ir atkarīgi no tīkla pieslēguma, tāpēc nevar nodrošināt tādu veiktspēju kā konkrētas platformas lietotnes.

Tīmekļa lietotņu priekšrocības un trūkumus var redzēt tabulā 1.2.

Tabula 1.2.

Tīmekļa lietotņu trūkumi un priekšrocības

Priekšrocības	Trūkumi
ātrāks un lētāks izstrādes process	nav pieejami bezsaistē
nav nepieciešamības glabāt atmiņā	lietotnes nav pieejas lietotņu veikalā
atjauninājumu process ir ātrs un viegls	mazāka ērtība, salīdzinājuma ar konkrētas platformas lietotnēm
neatkarība no mobilās platformas	

1.3. Hibrīda lietotnes

Hibrīda lietotnes, kā jau var saprast pēc nosaukuma, apvieno divus izstrādes paņēmienus – tīmekļa vietnes izstrādi un konkrētas mobilās platformas lietotnes izstrādi. Hibrīda izstrāde pilnībā neatbrīvojās no lietotnes piesaistes pie konkrētas platformas, bet tā nodrošina kopīgu koda un loģikas izmantošanu starp dažādām platformām.

Kad tīmekļa daļa ir izveidota, izstrādātāji uzkompilē šī koda bāzi dažādos konkrētos platformas formātos: priekš *Android, iOS, BlackBerry* vai *Windows Phone*.

Bieži vien, drošība un veiktspēja ir sliktāka, nekā konkrētas platformas lietotnēm.

Hibrīda lietotņu trūkumi un priekšrocības ir apkopoti tabulā 1.3.

Tabula 1.3.

Hibrīda lietotnes trūkumi un priekšrocības

Priekšrocības	Trūkumi
Vienota koda bāze vairākām platformām	Platformas īpatnības nevar tikt izmantotas izstrāde, jo var būt nepieejamas citā platformā
Ātri ieviešami atjauninājumi	Izstrādāt vienādas dizaina vadlīnijas dažam platformām ir grūti
	Zemāka ātrdarbība

Kad ir noskaidrots lietotnes veids un ir zināma ierīces platforma, var ķerties pie testēšanas un testu automatizācijas rīka izvēles.

2. MOBILO LIETOTŅU TESTĒŠANAS PAVEIDI UN TO SPECIFIKA

Testēšana ir "programmatūras un aparatūras darbības pārbaude, izmantojot testdatus. Testēšanas mērķis var būt defekta atklāšana, tā atrašanās vietas lokalizēšana vai arī testējamā objekta dinamisko parametru noskaidrošana [6]." Veicot melnās kastes mobilās lietotnes testēšanu, tiek pārbaudīta lietotnes kvalitāte - lietojamība, baterijas patēriņš lietotnes lietošanas laikā, tās funkcionalitāte un atbilstība iepriekšdefinētām prasībām, tās veiktspēja.

Mobilo lietotņu testēšanas process pēc būtības ir līdzīgs datora un tīmekļa lietotņu testēšanai – tiek pielietotas tas pašas testēšanas metodes, veidi un līmeņi, tomēr veicot mobilo lietotņu testēšanu, testētājam ir jārēķinās ar mobilo ierīču specifiskām īpatnībām: ierīču un operētājsistēmas versiju sadrumstalotība, neliels ekrāna izmērs, dažāda veida pārtraukumi, ierīces specifiskais tehniskais nodrošinājums un veiktspējas ierobežojumi.

Lai vieglāk būtu orientēties, kas un kā tiek testēts veicot mobilo lietotņu testēšanu, šajā sadaļa darba autors ir apkopojis informāciju par mobilo lietotņu svarīgāko testēšanas paveidu īpatnībām, kā arī iedalījis, kāds no testēšanas paveidiem var tikt automatizēts un/vai testēts manuāli.

2.1. Funkcionālā testēšana

Bieži vien, minot vārdu "testēšana", tiek minēta tieši lietotņu funkcionāla testēšana, jo tieši šīs testēšanas veidam tiek veltīts visvairāk testēšanas laika. Pēc 2014 gadā *Keynote* veikta pētījumā, kurā piedalījās vairāk nekā 1600 mobilo lietotņu izstrādātāji un testētāji, tika secināts, ka mobilo lietotņu testēšanā lielāka uzmanība tiek veltīta tieši funkcionālai testēšanai – 47% no visa testēšanas procesa [7].

Funkcionālā testēšana ir viena no svarīgākām lietotnes testēšanas daļām, jo testējot jebkuru lietotni, gan mobilo, gan parasto, ir jāpārbauda lietotnes funkcionalitāte – vai visas lietotnes funkcijas, prasības un specifikas ir īstenotas korekti un atbilst prasībām.

Veicot mobilo lietotņu funkcionālo testēšanu ir jārēķinās ar to, ka visi moderni viedtālruņi un planšetdatori ir aprīkoti ar vienu vai vairākām kamerām, mikrofonu un skaļruņiem, dažāda tipa sensoriem, piemēram, žiroskops, akselerometrs vai gaismas sensors, un skārienekrānu, tas arī atšķir parasto lietotņu testēšanu no mobilo lietotņu testēšanas. Līdz ar to, testētāja uzdevums ir pārbaudīt lai lietotnē korekti sadarbotos ar ierīces aprīkojumu. Tas arī atšķir parasto lietotņu testēšanu no mobilo lietotņu testēšanas.

Mobilo lietotņu funkcionāla testēšana pārsvara tiek veikta caur grafisko lietotāja saskarni, kad tiek definēti konkrēti lietotnes lietošanas gadījumi, un testētājs, lietotnē izpildot iepriekšdefinētus lietošanas gadījumā soļus, nosaka, vai saņemtais rezultāts atbilst sagaidāmam. Tādā veidā tiek definēti dažāda veida testa piemēri, kas atbilst lietotnes reāliem lietošanas gadījumiem. Var teikt, ka funkcionalitāte tiek pārbaudīta tādā veidā, kā to izmantos lietotnes reāls lietotājs. "Veicot tāda veida testēšanu lietotnes izstrādes agrīnā stadijā, palielinās izstrādātāju un testētāju produktivitāte, palielinās koda kvalitāte un samazinās risks atrast kļūdas izstrādes beigu posmā [8]."

Funkcionālo testēšanas procesu ir iespējams automatizēt veicot, kā jau iepriekš bija minēts, lietotāja saskarnes līmeņa funkcionālas testēšanas automatizāciju, kad tiek izveidoti reāli lietotnes lietošanas gadījumi, un tiek automātiski pārbaudīta lietotnes funkcionalitāte caur grafisko lietotāja saskarni.

2.2. Lietojamības testēšana

Mobilo lietotņu labas lietojamības izveide, kā jebkurai lietotnei ar grafisko lietotāja saskarni, ir viens no svarīgākiem izstrādes mērķiem, no kura var būt atkarīga lietotnes reputācija, jo slikti saprotamas un nelietojamas lietotnes, pat ar labu funkcionalitāti, lietotāji nelietos. Apmeklējot kādu mājaslapu vai ieslēdzot lietotni, lietotajam nav jātērē laiks tās saskarnes pētīšanai un saprašanai, jo ir pieejamas vairākas alternatīvas, kas atbilst lietotāja prasībām [9].

Tieši lietotnes lietojamība veido lietotājam pirmo iespaidu par lietotni. Intuitīvi nesaprotama lietotne vai nekorekti attēlojami lietotnes saskarnes elementi bieži vien noved pie tā, ka lietotāji atsākas no lietotnes izmantošanas un meklē alternatīvas. Tāpēc ir svarīgi pārliecināties, ka lietotne ir pašsaprotama, viegli lietojama un vizuāli acīm patīkama: visi elementi ir izveidoti vienota stilā – vienāds izmērs, fonts, krāsas un atstarpes.

Veicot lietojamības testēšanu ir grūti definēt prasības, attiecībā pret kurām tiek veikta testēšana, tomēr dažādi lietojamības pētīšanas eksperti un organizācijas ir izstrādājuši lietotņu lietojamības kvalitātes radītājus un vadlīnijas, balstoties uz kurām var tikt novērtēta lietotnes lietojamība. Kā piemēru var minēt ISO 9241-11 standartu, kur lietojamības tiek novērtēta pēc trim rādītājiem: rezultativitāte, efektivitāte un apmierinātība [10], un Džeikoba Nīlsena definētie lietojamības kvalitātes komponenti: atgūstamība, efektivitāte, neaizmirstamība, kļūdas un apmierinātība [9]. Zem katra kvalitātes komponenta, ir konkrēts jautājums, atbildot uz kuru, testētājs spēs secināt par testējamas lietotnes kvalitāti.

Bieži vien, nopietnākai mobilo lietotņu lietojamības testēšanā tiek piesaistīti cilvēki, kuri nav saistīti ar lietotnes izstrādi, tas var būt potenciāls lietotnes lietotājs vai arī eksperti, ar pieredzi lietojamības testēšanā, ar mērķi, uzzināt un izpētīt viņu viedokli par testējamo lietotni. Šī pieeja tiek izmantota laboratorijas eksperimentu metodē, kad lietotājs veic lietotnes izpēti tā saucamā laboratorija testētāju uzraudzībā, vai arī laukā pētījumos, kad testētājam, pēc lietotnes lietošanas tiek uzdoti ar lietojamību saistīti jautājumi. Pielietojot šīs metodes, vēl izstrādes sākumā var definēt lietotāju prasības [11].

Mobilo lietotņu lietojamības testēšana ir jāveic manuāli, jo automatizācijas rīki nespēj patstāvīgi analizēt lietotnes izskatu un elementu lietotājam ērtu un saprotamu attēlojumu.

2.3. Baterijas patēriņa testēšana

Testējot mobilo lietotni, noteikti ir jāveic baterijas patēriņā testēšana, jo ja lietotne patērē pārāk daudz enerģijas, lietotājs noteikti to neizmantos. Pārsvarā, mobilo lietotņu testēšanā notiek divos variantos: kad baterija ir pilnībā uzlādēta un ir zems baterijas uzlādes līmenis.

Pilnībā uzlādētas baterijas testēšana notiek pēc sekojoša scenārija: baterija ir pilnībā uzlādēta, testējama lietotne ir uzstādīta un atvērta, ierīce atrodas gaidstāvē (lietotne strādā priekšplānā) [1]. Testētāja uzdevums ir pārbaudīt cik procentu baterijas lādiņa tiek patērēts lietotnes lietošanas laikā. Bieži vien šī testēšana tiek veikta lietotnes reālas lietošanas apstākļos, kad testētājs veic noteiktas darbības lietotnē, imitējot reālos lietošanas gadījumus. Lietošanas gadījumi tiek atkārtoti pēc noteikta laikā, piemēram pēc katram 15 minūtēm, un pirms katra lietošanas gadījuma atkārtojuma tiek piefiksēts baterijas patēriņš. Tāda veidā var noteikt, cik liels ir baterijas patēriņš lietotnes noteiktā lietošanas laikā.

Pirms sākt testēt, nedrīkst aizmirts izslēgt visas citas lietotnes, kas var patērēt baterijas lādiņu, ka arī ir jāizslēdz jauninājumi, kas pēkšņi var sākties testēšanas laikā. Tieši tāds pats tests var atkārtot ar lietotni fona režīmā.

Zemas baterijas uzlādes testēšana notiek līdzīgi pilnībā uzlādētas baterijas testēšanai, ar izņēmumu, ka testēšanas sākumā ierīces baterija uzlādes līmenis ir zems, t.i. 10% vai 15%. Tādā uzlādes līmenī, mobilā ierīce izslēdz ierīces īpašas funkcijas, ka GPS, tīkla pieslēgumu un dažādus sensorus, kas patērē enerģiju un saīsina ierīces darbības laiku. Jā lietotne izmanto šīs ierīces iespējas, ir nepieciešams pārbaudīt, kā tā uzvedīsies zema uzlādes līmenī: jāpārbauda, vai lietotne nenobrūk, neiesalst [1].

Bateriju testēšanu var veikt gan manuāli, gan automatizēti, tomēr pēc autora pieredzes, labāk ir izmantot kādu automatizācijas rīku, jo testējot manuāli, testētājam ir precīzi jāseko laikam kad

jāveic mērījums, bet tas ne vienmēr sanāk, it īpaši ja testētājs paralēli nodarbojas ar vairāku momentu testēšanu.

2.4. Veiktspējas testēšana

Veiktspējas testēšana ir viens no galvenajiem aspektiem jebkurās programmatūras izstrādē, it īpaši uz mobilām ierīcēm, jo mūsdienās lietotnes darbošanās ātrums ir viens no svarīgākiem rādītājiem, kuram pievērš uzmanību lietotājs.

Testējot veiktspēju, vairāk jākoncentrējas uz lietotnes ātrdarbību. Piemēram, ir jāpārbauda, cik ilgi ielādējas lietotne, lietotnes bildes vai animācijas, teksts, cik ātri lietotne atbild uz lietotāja pieprasījumiem. Ja testētājam liekas, ka kaut kas ielādējas pārāk ilgi, to noteikti, ir jāpiefiksē.

Tieši ātrdarbība ir svarīga, jo pēc veikta pētījuma, 60% mobilo ierīču lietotāju atsakās no mobilas lietotnes vai mājaslapas lietošanas, ja tā neielādējas trīs sekunžu laikā [12]. Līdz ar to, mobilām lietotnēm ir jābūt labi optimizētam, lai atbilstu lietotāju prasībām.

Veiktspējas testēšana noteikti ir jāveic pirms katras testējamas lietotnes jaunās versijas izlaišanas, jo jaunās relīzes kandidātam nedrīkst būt lēnākam par iepriekšējo, pretējā gadījumā lietotne zaudēs savu reputāciju.

Veiktspējas testēšanu var veikt gan manuāli, gan izmantojot profilētāju, kas mēra funkciju izpildes laiku. Profilētāju var izmantot paši izstrādātāji.

2.4.1. Stresa un pārtraukumu testēšana

Viens, no veiktspējas testēšanas veidiem, kas bieži tiek pielietots mobilo lietotņu testēšanā ir stresa un pārtraukumu testēšana. Šie divi testēšanas veidi tiek veikti kopā, jo bieži vien, mobilās lietotnēs tieši dažāda veida pārtraukumi var izraisīt stresa situāciju un noslogot sistēmu.

Stresa testēšana ir mobilo lietotņu testēšanas daļa, kura tiek veikta, lai novērtētu sistēmas darbību atbilstoši vai ārpus noteikto prasību limitiem [13], tas nozīmē, ka tiek pārbaudīts, kas notiek ar lietotni stresa situācijā, cik lietotne ir izturīga robežslodzē.

Savukārt, patraukumu testēšana ir testēšanas veids, kurā tiek pārbaudīta lietotnes uzvedība pārtraukuma gadījumā. Pār pārtraukumu mobilajā lietotne tiek uzskatīta zvanu vai īsziņu saņemšana vai sūtīšana, pašpiegādes (push) paziņojumu saņemšana, kameras vai citu lietotņu ieslēgšana [14], ierīces pogu spiešana. Bieži vien, tāda veida pārtraukumi arī ir stresa situācijas cēloņi.

Stresa un pārtraukumu testēšanu ir svarīgi veikt, jo mobilo ierīču viena no primārām funkcijām ir tieši zvanīšana un īsziņu sūtīšana vai saņemšana, līdz ar to, tāda veida pārtraukumi var notikt jebkurā noteiktas lietotnes lietošanas laikā un sistēma var tikt noslogota negaidīti, tāpēc ir svarīgi pārliecināties, ka stresa vai pārtraukuma rašanas gadījumā saskarnes elementi nepazūd, lietotne nenobrūk vai neiesalst, kā arī jāpārbauda, vai netiek pazaudēta informācija. Tātad, veicot šī veida testēšanu, testētājs varēs secināt par lietotnes veiktspēju un stabilitāti.

Stresa un pārtraukumu testēšana manuāla testēšana ir ļoti laikietilpīgs process, kas nevar nodrošināt ātru ieejas datu ģenerēšanu un ievadīšanu, līdz ar to, šīs process bieži vien tiek automatizēts ar dažādu rīku palīdzību.

2.5. Uzstādīšanas testēšana

Uzstādīšanas process rāda pirmo iespaidu par mobilo lietotni. Ja instalācija būs neveiksmīga, diez vai lietotājs mēģinās vēlreiz uzstādīt lietotni, tāpēc ir svarīgi pārbaudīt, lai uzstādīšanas process notiktu bez kļūdām.

Lai pārliecinātos, ka lietotne būs veiksmīgi uzstādīta, testētājam ir jāpārbauda vairāki gadījumi, kas varētu rasties mobilās lietotnes uzstādīšanas laikā. Iespējamie testēšanas varianti [1]:

- Jāpārbauda, ka lietotni drīkst veiksmīgi uzstādīt, gan ierīces atmiņā, gan atmiņās kartē;
- Jāpārbauda, kā lietotne uzstādīsies ar dažādiem tīkla pieslēguma veidiem (WiFi vai datu tīkls);
- Uzstādīšanas laikā, mainīt tīkla pieslēgumu, no WiFi uz 3G/4G, vai izslēgt tīkla savienojumu;
- Uzstādīšanas laikā pārslēgties uz citu lietotni;
- Mēģināt uzstādīt lietotni, kad ierīcē trūkst atmiņas;
- Uzstādīt lietotni caur datu kabeli izmantojot datoru .

Veicot šīs darbības, jāvēro, vai lietotne nenobrūk vai neiesalst. Uzstādīšanas testēšana tiek veikta tikai manuāli.

2.5.1. Atinstalēšanas testēšana

Kad ir veikta uzstādīšanas testēšana, testētājam noteikti ir jāpārbauda arī pretējais process – lietotnes atinstalēšana. Veicot šī veida testēšanu, ir jāpārliecinās, lai pēc lietotnes atinstalēšanas,

uz ierīces nepaliek lietotnes dati un lietotne tiek izdzēsta pilnībā, kā arī jāpaplicinās, lai atinstalēšanas process neiesalst vai nenobrūk, vai arī neparadās nevēlamas kļūdas.

Pēc veiksmīgas atinstalēšanas, veicot uzstādīšanas testēšanu pa jaunam, visiem datiem, kas bija saglabāti pirms atinstalēšanas, nav jābūt pieejamiem – lietotnei ir jāuzvedas tā, it kā būtu pirmo reizi uzstādīta, vienīgi, ja dati pirms tam nebija saglabāti mākonī.

Šīs process, tāpat ka uzstādīšanas testēšana tiek veikta manuāli.

2.5.2. Atjaunināšanas testēšana

Kopā ar lietotnes uzstādīšanu un atinstalēšanu, mobilo lietotņu lietotājiem nākas atjaunināt lietojamo lietotni, ja tās izstrādātājs ir izveidojis tās jaunāko versiju. Līdz ar to, testētāja uzdevums ir pārbaudīt, lai lietotnes atjaunināšanas procesā nerodas kļūdas [1], kā arī, vai atjaunināšana notiek veiksmīgi – lietotne tiek veiksmīgi atjaunināta, vai nenotiek datu zudums, pēc atjaunināšanas.

Atjaunināšanas testēšana ir jāveic ne tikai atjauninot lietotni no iepriekšējas versijas, bet arī jāpārbauda, vai lietotne tiek veiksmīgi atjaunināta arī no vecākām versijām, jo bieži vien gadās, ka lietotājs kādu laiku neatjaunina lietotni.

Atjaunināšanas testēšana, tāpat ka uzstādīšanas testēšana jāveic tikai manuāli.

3. MOBILO LIETOTŅU AUTOMATIZĒTA TESTĒŠANA

Šodien automatizēta gan mobilo, gan tīmekļa, gan "parasto" lietotņu testēšana tiek plaši izmantota testēšanas procesā un testu automatizācijas popularitāte tikai aug. Līdz ar to, palielinās pieprasījums pēc testu automatizācijas speciālistiem.

Runājot par definīciju, automatizēta testēšana, ir testēšanas process, kurā tiek izmantota speciala programmatūra un rīki, kas automatizē testpiemēru izpildi un novērtēšanu [15], salīdzina saņemto rezultātu ar paredzamo. Var teikt, ka automatizācijas rīki imitē mobilās lietotnes reāla lietotāja darbības. Atkarībā no rīka, eksistē arī tādi, kas palīdz novērtēt mobilas lietotnes veiktspēju, veikt automatizēto baterijas testēšanu un simulēt stresa situāciju, nodrošināt ātru ieejas datu ģenerēšanu, un ievadīšanu.

Testu automatizācijas pielietošanas mērķis ir optimizēt testēšanas procesu, mazinot rutīnas darbu, līdz ar to samazinot izmaksas un testēšanas laiku, it īpaši ja projekts ir samērā liels un lietotnes jaunas versijas tiek veidotas samērā bieži.

Pārsvarā testu automatizācija nav vērsta uz jaunu kļūdu meklēšanu, bet gan uz sistēmas prasību atbilstības pārbaudi. Parasti, tiek automatizēti regrestestēšanas testu kopas, tas nozīmē veicot automatizēto testēšanu tiek pārbaudītas sistēmas vecās kļūdas, vai tās ir veiksmīgi izlabotas, kā arī vai pēc veco kļūdu izlabošanas nav paradījušas jaunas kļūdas.

Šajā darba daļā autors mēģinās apkopot automatizētas testēšanas priekšrocības un trūkumus, kā arī saprast - kas un kad ir jāautomatizē testējot mobilās lietotnes.

3.1. Automatizētas testēšanas priekšrocības

Testu automatizācijai ir vairākas priekšrocības, kas varētu kļūt par iemeslu, iekļaut to testēšanas procesā. Šajā sadaļā darba autors ir apkopojis un izvirzījis galvenās automatizētas testēšanas priekšrocības.

Pirmkārt, testu automatizācija ilgtermiņa projektā noteikti samazina testēšanas laiku un izmaksas, jo testēšana notiek automātiski, bez testētāja iesaistīšanas, tas nozīmē, ka testēšana var notikt arī tad, kad testētājs ir aizņemts ar citu darbu. Automatizēta testēšana notiek daudz ātrāk nekā manuāla, jo automatizācijas rīks spēj daudz ātrāk izpildīt komandas, "spiest podziņas" un ievadīt nepieciešamo informāciju. Līdz ar to, testēšanas un izstrādes process paātrinās, jo programmētāji tiek ātrāk informēti par iespējamām kļūdām lietotnē.

Kā arī, testi var tikt atkārtoti neierobežoti daudz reižu un tādā veidā, testētājam nav nepieciešams veikt rutīnas darbu.

Automatizējot testus, testēšanu var veikt uzreiz uz vairākām ierīcēm, tas nozīmē, ka palielinās ne tikai testēšanas ātrums un kvalitāte, bet arī ierīču pārklājumu, jo testēšana var notikt uz ierīcēm ar dažādu operētājsistēmu versijām un dažāda izmēra ekrāniem, kas mobilo lietotņu izstrāde ir ļoti svarīgs moments.

Palielinās arī testa gadījumu pārklājums, jo automatizācijas rīks var izpildīt lielu skaitu testēšanas gadījumu un to dažādas kombinācijas [16].

Nedrīkst arī aizmirt par cilvēcisko faktoru. Veicot ikdienā rutīnas darbu, cilvēks neuzmanības dēļ var nepamanīt kādu svarīgu un ne tik svarīgu kļūdu. Savukārt testu automatizācija tādu iespēju samazina, jo automatizācijas rīks precīzi izpilda ievadītas komandas, vienīgi, ja komanda tika padota kļūdaini [17].

Svarīgs moments ir arī tas, ka daži automatizācijas rīki var izpildīt darbības, kuras vienam testētājam būtu grūti vai neiespējami atkārtot. Kā piemēru mobilo lietotņu testēšanā var minēt stresa un pārtraukumu testēšana. Testētājam būtu grūti vienam ātri izveidot slodzi uz ierīci - spiest vairākas pogas un sūtīt dažāda veida paziņojumus, ātri aizpildīt dažādus ievadlaukus vai arī pievienot kontaktu sarakstā vairākus simtus kontaktu. Tāpēc šeit arī tiek izmantoti automatizācijas rīki, kas to imitē un izpilda dažu sekunžu laikā.

3.2. Automatizētas testēšanas trūkumi

Neskatoties uz automatizētas testēšanas vairākām priekšrocībām, automatizētai testēšanai ir arī vairāki ierobežojumi un trūkumi, kurus jāņem vērā.

Ir jāsāk no tā, ka testu automatizācija ir ilgs un darbietilpīgs process, kas sākas no manuālas testēšanas, jo tikai pēc paveiktas manuālas testēšanas testētājs var korekti "paradīt" rīkam, ko un kā tam jātestē, un kad viss ir saplānots, testētājam ir jāievada katrs atsevišķs testēšanas gadījums rīkā.

Testu automatizācija ir arī samērā dārga, jo tas prasa jaunu cilvēku piesaisti komandā un viņu apmācību, kā arī jaunas aparatūras un programmatūras iegādi [17].

Nākošais automatizētas testēšanas trūkums ir tas, ka mainoties lietotnes dizainam, testētājam ir jāpapildina un jāpārstrādā automatizētie testi, līdz ar to, testētājam bieži vien ir "jātestē testi", jeb jāpārbauda ievadīto skriptu pareizība.

Ka trūkumu var arī minēt to, ka testu automatizāciju veic cilvēki, līdz ar to ir jāpieņem tas fakts, ka testēšanas skripta tiks pieļauta kļūda, kura varētu ietekmēt lietotnes kvalitāti vai izstrādes procesu.

Nedrīkst arī aizmirst, ka testēšanu veic rīks, līdz ar to, veicot testēšanu var izlaist kādu nelielu kļūdu kuras pārbaude nebija iekļauta skriptā.

Vel viens, un ,manuprāt, būtisks trūkums ir tas, ka veicot testu automatizēšanu, testētājam ir jābūt kodēšanas iemaņām, jo testu automatizēšana ir sava veidā programmēšana, līdz ar to, ne katrs testētājs spēs automatizēt testus.

Apkopojot automatizētas testēšanas priekšrocības un trūkumus, var secināt, ka tomēr automatizēta testēšana nav manuālas testēšanas aizvietojums, jo testēšanas rīki nav "aprīkoti" ar mākslīgo intelektu un nespēj aizvietot dzīvu cilvēku, tomēr pareizi kombinējot šīs metodes, var noteikti optimizēt testēšanas procesu, un palielināt izstrādājamo lietotņu kvalitāti.

3.3. Mobilo lietotņu testu automatizācijas pielietojums

Automatizācijas process atvieglo testētāja darbu, jo darbs tiek pildīts automātiski, kā arī palīdz notestēt tas lietotnes, daļas, kuras testētājs fiziski nespēj izpildīt, tomēr nedrīkst aizmirst, ka visu testēšanas procesu nedrīkst un nevajag automatizēt. Līdz ar to, rodas jautājums, ko tieši ir jāautomatizē?

Atrast konkrēto atbildi uz doto jautājumu, kas derētu jebkuras mobilās lietotnes testēšanai nevar, jo testēšanas stratēģija katram mobilas lietotnes tipam un mobilai platformai atšķirsies. Tomēr pielietojot testu automatizāciju savā projektā, ir ieteicams automatizēt tieši sekojošo [18]:

- kritiskas lietotnes daļas, kuram ir svarīga nozīme mobilajā lietotnē (piem. reģistrācija vai pieteikšanās),
- tos testēšanas gadījumus, kurus ir jāatkārto daudzkārt,
- testēšanas gadījumi, kurus ir grūti vai ilgi izpildīt manuāli,
- pieņemšanas kritēriji, kas varētu ātri sniegt atbildi par lietotnes atbilstību prasībām,
- regresijas testu komplektu, lai pēc jaunas versijas izlaišanas, testētājs ātri varētu noteikt, vai lietotne "nav palikusi sliktāka",
- stresa un pārtraukumu situācijas, jo testētājam pašam būs grūti noslogot ierīci vai ātri izraisīt dažāda veida pārtraukumus lietotnes darbībā,
- baterijas uzlādes patēriņu, jo manuāli veicot baterijas testēšanu, testētājam ir visu laiku jāseko līdzi baterijas lādiņam un jāpiefiksē patēriņš.

Līdz ar to, var redzēt, ka, pirms uzsākt testu automatizāciju, ir kārtīgi jāsaplāno testēšanas stratēģija, lai varētu novērtēt kurus lietošanas gadījumus ir jāautomatizē, un vai ir jāautomatizē vispār. Tomēr noteikti var secināt, ka automatizēta testēšana vairāk der ilgtermiņa projektiem ar lielu funkciju klāstu.

4. MOBILO LIETOTŅU AUTOMATIZĀCIJAS RĪKI

Tā, ka palielinās pieprasījums pēc mobilo lietotņu testu automatizācijas, palielinās arī pieprasījums pēc dažāda veida automatizācijas rīkiem. Automatizācijas rīks ir speciāla lietojumprogramma, kura tiek izmantota testu automatizēšanai un automatizēto testu izpildei.

Tagad eksistē vairāki mobilo lietotņu automatizācijas satvari, kuri ietver sevī automatizācijas rīkus. Priecē arī tas, ka daudzi šie satvari ir pieejami bezmaksas un ir atvērta pirmkoda, līdz ar to, tos ir viegli konfigurēt pēc paša testētāja vai kompānijas, kurā viņš strādā, gribas.

Katrs mobilo lietotņu automatizācijas rīks atšķiras ar lietotāja saskarnes elementu atpazīšanas pieeju, mobilo operētājsistēmu un mobilo lietotņu veidu atbalstu, testu automatizācijas veidu, atbalstāmām programmēšanas valodām un citiem parametriem. Tāpēc pirms uzsākt testu automatizāciju ir svarīgi definēt prasības automatizētas testēšanas rīkam.

Šajā darba daļā autors ir apkopojis mobilo lietotņu automatizācijas rīku izvēles kritērijus.

4.1. Grafiskās lietotāja saskarnes testēšanas automatizācijas rīku veidi

Izšķir vairākus mobilo lietotņu automatizācijas rīku veidus pēc lietotāja saskarnes elementu atpazīšanas pieejas: attēlu atpazīšanas pieeja, tekstu atpazīšanas pieeja, objektu atpazīšanas pieeja, darbības ierakstīšanas un izpildīšanas pieeja. Pirms izvēlēts konkrētu testu automatizācijas rīku, testētājam ir noteikti jāzin katras lietotāja saskarnes elementu atpazīšanas pieejas vājas un stipras puses, jo tieši šīs varētu kļūt par noteicošo rīka izvēlē.

4.1.1. Attēlu atpazīšanas pieejas rīki

Rīki, kuri izmanto attēlu atpazīšanas pieeju, lai izpildītu automātiskas darbības, salīdzina rīkā ievadīto attēlu ar lietotnē esošiem. Veidojot skriptu, tiek izveidots kāda elementa ekrānuzņēmums, kas tiek ierakstīts skriptā. Kad skripts tiek palaists, rīks ar attēlu atpazīšanu salīdzina redzamo mobilās ierīces ekrānā ar skripta saglabātiem attēliem, un ja šī ievadītais attēls tiek atrasts ekrānā, tad tiek izpildītas skriptā ievadītas darbības.

Šāda veida rīki ir ļoti noderīgi, ja mobilās lietotnes lietotāja saskarnes dizains ir nemainīgs vai netiek mainīts ļoti bieži, it īpaši, ja vairākās mobilajās platformās tiek izmantoti vienādi elementi – pogas, dažāda veida logo vai bildes, tad rīkus, kuri ir balstīti uz attēlu atpazīšanu, var pielietot dažādplatformu mobilo lietotņu testēšanai. Kā arī, tāda veida rīki ļauj viegli veikt testu

automatizāciju izmantojot tika lietotnes lietotāja saskarnes elementu attēlus, līdz ar to, testētājam nav nepieciešams pārzināt visu elementu klases, indeksus vai ID [19].

Tomēr attēlu atpazīšanas rīkiem ir arī savi trūkumi – tie ir ļoti jūtīgi pret dažāda veida izmaiņām. Piemēram, mainot ierīces orientāciju no portretorientācijas un ainavorientāciju, skripts noteikti nenostrādās, tāpēc katram režīmam ir jāraksta atsevišķs skripts. Kā arī dažāda ekrāna izmēra ierīcēm šīs rīku veids nederēs.

Vēl viens no trūkumiem ir tas, ka šos rīkus nav iespējams testēt lietotnes dažādas valodas, vai arī katram tulkojumam ir jāveido savs skripts, jo mainot valodu, mainās arī uzņemtais skripta attēls. Tagad daudzi mobilo lietotņu izstrādātāji lietotnēs piedāvā vairākas valodas, līdz ar to šis fakts varētu kļūt par noteicošo rīku tipa izvēlē.

Attēlu atpazīšanas pieeja tiek izmantota sekojošos Android mobilo lietotņu testu automatizācijas rīkos:

eggPlant [20] – kompānijas TestPlant izstrādātais testu automatizācijas maksas rīks, kas ir paredzēts tieši mobilo lietotņu testu automatizācijai,

Sikuli [21] – bezmaksas rīks kas ir paredzēts gan datora, gan mobilo lietotņu testu automatizācijai, tomēr, lai veiktu testu uz reālas Android ierīces, ir nepieciešama ierīces modifikācija.

Ranorex [22] – maksas testu automatizācijas rīks, kas ir paredzēts gan datora, gan mobilo lietotņu testu automatizācijai.

4.1.2. Objektu atpazīšanas pieejas rīki

Rīki, kas izmanto objektu atpazīšanas pieeju, lietotāja saskarnes elementus atpazīst caur saskarnes elementu koku. Pie elementiem var piekļūt caur XPath, ID vai klases nosaukumu [23].

Objektu atpazīšanas rīki ir ļoti izplatīti dažādu mobilo lietotņu testēšanā – tīmekļa, hibrīda vai konkrētas platformas lietotņu, jo tāda objektu atpazīšanas pieeja ļauj bez grūtībām piekļūt pie jebkādiem lietotāja saskarnes elementiem: pogām, ievadlaukiem, skatiem un citiem, ja šiem elementiem ir definēts konkrēts ID.

Pateicoties tam, ka objektu ID un klases pārsvarā ir nemainīgas, šī veida rīki ļauj veikt automatizētu testēšanu pat pēc dizaina izmaiņām, ja lietotāja saskarnes elementu atrāšanas vietas tiek mainītas. Tas nozīmē arī to, ka dažāda izmēra un izšķirtspēju ekrāniem var izmantot vienu un to pašu skriptu.

No tā var secināt, ka šī veida automatizācijas rīki ir vispiemērotākie testu automatizācijai, jo ka jau iepriekš minēju, tie nav atkarīgi no lietotāja saskarnes dizaina izmaiņām, ekrāna orientācijas

maiņas, izšķirtspējas un lieluma, līdz ar to, testētājs var daudz ātrāk un vieglāk veikt testu automatizāciju.

Pazīstamāki Android testu automatizācijas objektu atpazīšanas rīki ir:

Appium [24] – atvērta pirmkoda dažādplatformu testu automatizācijas rīks, kas ir paredzēts konkrētas platformas, hibrīda un arī tīmekļa mobilo lietotņu testēšanai.

Calabash [25] – bezmaksas mobilo lietotņu testu automatizācijas rīks, kas ir paredzēts hibrīda un konkrētas platformas mobilo lietotņu testēšanai. Eksistē šī rīka divas versijas: Android lietotņu testēšanai un iOS lietotņu testēšanai.

Selendroid – "Selenium priekš Android" [26], Android mobilo lietotņu testu automatizācijas rīks, kura pamatā tiek izmantotas Selenium bibliotēkas. Rīks ir paredzēts visiem mobilo lietotņu veidu testēšanai.

Espresso [27] – Google izstrādāts atvērta pirmkoda Android lietotņu testu automatizācijas rīks. Rīks der konkrētas platformas, hibrīda un tīmekļa lietotņu testēšanai.

Robotium [28] – atvērta pirmkoda Android lietotņu testu automatizācijas rīks, kas paredzēts konkrētas platformas un hibrīda lietotņu testēšanai.

4.1.3. Koordinātu atpazīšanas pieejas rīki

Koordinātu atpazīšanas balstītie rīki darbojas izmantojot x un y ass koordinātes. Koordinātes tiek izmantotas, lai atzīmētu lietotāja saskarnes elementa atrašanas vietu un tādā veidā rīks sāk darboties, spiežot x un y ass noradītās vietās.

Šīm automatizācijas rīku veidam ir līdzīgi trūkumi, kā attēlu atpazīšanas rīkiem, jo mainoties dizainam, mainās arī elementu koordinātes un skriptu ir jāpārveido ievadot elementu jaunas koordinātes. Arī dažādu izmēru ekrāniem ir jāievada savas koordinātes, līdz ar to testu automatizācijas darbs paliek apjomīgāks [23].

Līdzīgas ir arī priekšrocības, īpaši ja dažādās platformās lietotnes elementi atrodas vienā vietā, tad šos rīkus noteikti var izmantot dažādplatformu mobilo lietotņu testēšanā.

Koordinātu atpazīšanas balstīta pieeja var tikt pielietota arī vairākos objektu atpazīšanas rīkos: Appium, Calabash, Selendroid, Robotium.

4.1.4. Tekstu atpazīšanas pieejas rīki

Tekstu atpazīšanas rīki pārsvarā izmanto OCR tehnoloģijas, jeb optisko rakstzīmju atpazīšanu, lai lietotnē atrastu dažādu lietotnes elementos ierakstīto tekstu (pogu, ievadlauku u.c.), kas ir rakstīts šo elementu iekšā, tādā veidā elementi tiek pārveidoti un atpazīti ka teksts [29].

Tekstu atpazīšanas rīku priekšrocība ir tas, ka vienu skriptu var izmantot vairāku izšķirtspēju, ekrānu izmēru un ekrānu orientāciju testēšanā, jo šie parametri nekā neietekmē mobilās lietotnes elementu tekstu. Tomēr, ja lietotnes elementos nav nekāda teksta, tad elementu nav iespējams identificēt, līdz ar to testēšana paliek neiespējama.

Izvēloties šī tipa automatizācijas rīkus ir jārēķinās arī ar to, ka tie ir diezgan lēni, jo visa ekrānā attēlojamais lietotnes skats tiek skenēta, un tiek meklēti visi tajā esošie teksta elementi [30].

Šī lietotāja saskarnes elementu atpazīšanas pieeja, tāpat ka koordinātu atpazīšanas balstīta pieeja, tiek izmantota vairākos populārākos Android mobilo lietotņu testu automatizācijas rīkos: Appium, Selendroid, Robotium, eggPlant, Calabsh.

4.1.5. Darbību ierakstīšanas un izpildīšanas rīki

Kā atsevišķu mobilo lietotņu testēšanas rīku veidu ir vērts pieminēt darbību ierakstīšanas un izpildīšanas rīkus.

Darbību ierakstīšanas un izpildīšanas rīki, ka jau pēc nosaukuma var saprast, spēj ierakstīt testētāja darbības – pogu nospiešana, teksta ievadīšana, ritināšana – automātiski tiek izveidots skripts ar testētāja darbībām, kas ļauj šīs darbības atkārtot, kad tests tiek palaists [31].

Šos automatizācijas rīkus ir visvieglāk apgūt, jo tie ir pārsvara neprasa nekādas programmēšanas iemaņas un ir intuitīvi saprotamas. Tomēr, neskatoties uz to, ka viss izklausās tik viegli, darbību ierakstīšanas un izpildīšanas rīkiem ir ļoti daudz trūkumu.

Skriptu izpilde ir ļoti atkarīga no UI, ekrāna orientācijas, izmēra un izšķirtspēju izmaiņām, līdz ar to, skriptus ir grūti izpildīt uz dažāda veida ierīcēm.

Kā arī, viena no nozīmīgam problēmām, ir tas, ka testus, pēc skriptu izveidošanas, ir jāpārstrādā, jo gadās, ka testētāja ierakstītu darbību atkārtojums notiek pārāk ātri, kad testējamais skats vai lapa vēl nav atvērti, līdz ar to, testu automatizēto testu rezultāti var būt kļūdaini.

Šī pieeja tiek izmantota jau minētā eggPlant rīka, Robotium rīka maksas papildinājumā Robotium Recorder [32], Ranorex, kā arī RERAN [33] bezmaksas atvērta pirmkoda Android lietotņu testēšanas rīks.

4.2. Mobilo lietotņu testu automatizācijas rīku izvēles kritēriji

Kā jau iepriekš tika minēts, mobilo lietotņu automatizācijas rīki atšķiras pēc lietotāja saskarnes elementu atpazīšanas pieejas, mobilo operētājsistēmu un mobilo lietotņu veidu atbalsta, testu automatizācijas veida, atbalstāmām programmēšanas valodām un citiem parametriem, līdz ar to, pirms izvēlēties rīku ir svarīgi apzināties rīka iespējas. Darba autors ir apkopojis kritērijus, pēc kuriem būs vieglāk izvēlēties nepieciešamo rīku.

Pirmais, un svarīgākais kritērijs, kam ir jāpievērš uzmanība, ir kāda veida mobilā lietotne tiks testēta, jo ne vienmēr automatizācijas rīks spēj nodrošināt testu automatizāciju visiem trim lietotņu veidiem. Tikai hibrīda vai konkrētas platformas testēšanai derēs Calabash un Robotium rīki, tomēr šie rīki nenodrošina tīmekļa lietotņu automatizēto testēšanu. Savukārt, Appium, eggPlant, Selendroid rīki nodrošina visu triju mobilo lietotņu veidu testu automatizāciju, tajā skaitā arī tīmekļa.

Otrais svarīgākais kritērijs ir kādas mobilās platformas tiek atbalstītas – iOS, Android, Windows Phone, BlackBerry. Ir vairāki rīki, kas nodrošina dažādplatformu lietotņu testēšanu – egPlant, Appium, Calabsah, kas varētu būt par labu risinājumu rīka izvēlē. Tikai Android mobilo lietotņu testēšanai, kas autoram ir arī nepieciešams, ir paredzēti Selendroid un Robotium rīki.

Ir jāpievērš uzmanība arī tam, vai pirms automatizēt testus ir nepieciešams modificēt pašu lietotni, jo vairākiem rīkiem, piemēram, Calabash, lai tos izmantotu, ir nepieciešams pievienot lietotnes kodā jāpievieno dažādu bibliotēku atbalstu. Mēdz gadīties, ka pirms testēšanas ir nepieciešams modificēt lietotni ieslēdzot atkļūdošanas režīmu, līdz ar to, ir nedaudz modificē lietotnes kods. Šajā gadījumā, protams, vieglāk būtu izmantot tos automatizācijas rīkus, kur koda modificēšana nav nepieciešama – Appium, eggPlant, RERAN, Selendroid.

Nākošais, kam jāpievērš uzmanība, ir paša automatizācijas rīka veids, jeb lietotāja saskarnes elementu atpazīšanas pieeja. Kā jau iepriekš minēju, iedala četras elementu atpazīšanas pieejas – koordinātu atpazīšana, objektu atpazīšana, attēlu atpazīšana, tekstu atpazīšana. Pirms izvēlēties konkrēta veida rīku, ir jāsalīdzina katra rīku veida priekšrocības un trūkumus. Pēc autora veikta pētījuma, autors ir konstatējis, ka tomēr labāk ir pieturēties pie objektu atpazīšanas pieejas, kuras tiek pielietota – Appium, Calabash, Robotium un Selendroid rīkos. Tāda veida pieeja ir neatkarīga no lietotnes dizaina izmaiņām un ierīces orientācijas.

Vēl viens, manuprāt, svarīgs moments ir kādas programmēšanas valodas rīks atbalsta. Bieži vien testu automatizācijas rīki atbalsta tikai vienu konkrētu programmēšanas valodu, tāpēc ir jāizvēlas tādu rīku, kurā testētājs spēs rakstīt skriptus. Jā testētājs ir pazīstams ar Java programmēšanas valodu, tam derēs lielāka daļa rīku, kas paredzēti Android lietotņu testu

automatizācijai – Robotium, Selendroid, Espresso. Jā ir vēlme veikt rīku automatizāciju dabiskā valodā kopā ar Ruby programmēšanas valodu, tad noteikti ir jāizvēlas Calabash rīks. Protams, ir arī risinājumi, kur rīki atbalsta vairākas programmēšanas valodas – Appium, kas nodrošina pārsvara visu populāro programmēšanas valodu atbalstu.

Svarīgi ir arī tas, lai rīka izmantošanai, nav jāmodificē mobilā ierīce, jo, piemēram, veikt testu automatizāciju izmantojot Sikuli rīku var veikt tikai uz modificētam Android ierīcēm ar superlietotāja tiesībām. Ieteicams veikt testēšanu un nemodificētas ierīces, jo tad testēšana ir tuvu realitāte. Pēc autora pieredzes var teikt, la bieži vien funkcija kas strādā uz modificētas lietotnes, galu galā nekorekti strādās uz nemodificētas.

Būtu ieteicams arī, lai rīks spētu veikt testēšanu vienlaikus uz vairākām ierīcēm, tādā veidā, varēs nodrošināt lielāku ierīču un mobilo operētājsistēmu pārklājumu, kas Android lietotņu testēšana ir svarīgs, jo jau tagad eksistē vairāk nekā 1.4 miljardi Android ierīču ar dažādam operētājsistēmas versijām. Tādu iespēju nodrošina Calabash, Robotium, Selendroid un eggPlant rīki.

Svarīgi ir tas, lai rīks nodrošinātu pietiekami saprotamu atskaiti testa beigās. Būtu labi izvēlēties tādu rīku, kas varētu veikt ekrānuzņēmumus kļūdu gadījumā.

Protams, testēšana ir svarīgs arī testu izpildes laiks, tāpēc izvēloties rīku, būtu ieteicams izvēlēties pēc iespējas ātrāko, lai testētājs varētu ātrāk saņemt testu rezultātus, kas savukārt veicinās ātrāku izstrādes procesu.

Svarīgs moments rīku izvēlē ir arī rīku cena, jo bieži vien tieši izmaksas ir noteicošais faktors. Labi ir tas, ka daudzi mobilo lietotņu testu automatizācijas rīki ir pieejami bezmaksas, kā arī ir atvērta pirmkoda rīki, kurus var pārveidot pēc uzņēmuma vai testēšanas komandas gribas un vajadzībām.

Ir jāpievērš uzmanība arī tam, cik labi rīks ir dokumentēts, jo bieži vien, bezmaksas rīki ir skopi dokumentēti, līdz ar to, testētājam, tērējot laiku, ir jāmeklē papildus informāciju par rīka konfigurēšanu un testu automatizēšanu. Labu detalizētu dokumentāciju ir pieejami Robotium, Selendroid, eggPlant rīkiem.

Ņemot vērā visus augšā minētus kritērijus katrs testu automatizācijas speciālists vai testētāju komanda varēs viegli izvēlēties piemērotu automatizācijas rīku, kas atbilst viņu prasībām.

5. ANDROID MOBILO LIETOTŅU TESTU AUTOMATIZĀCIJAS RĪKU ANALĪZE

Kā tika minēts daļā 2.1., vairāki mobilo lietotņu testēšanas paveidi var tikt automatizēti. Šajā darbā daļā autors mēģinās veikt Android mobilo lietotņu lietotāja saskarnes testēšanas rīku analīzi pēc iepriekšdefinētiem kritērijiem, kā arī sniegt informāciju par mobilo lietotņu baterijas testēšanas automatizācijas iespējam, un stresa un pārtraukumu testēšanas automatizāciju.

5.1. Testējama lietotne

Lai veiktu testēšanu un testu automatizāciju, autoram bija nepieciešama testējama lietotne. Izvēloties lietotni testēšanai, autors balstījās uz sekojošam prasībām:

- Lietotnei ir jābūt pieejamai Android ierīcēm;
- Tai ir jābūt atvērta pirmkoda bezmaksas lietotnei, lai to varētu brīvi lejuplādēt;
- Lietotnē ir jābūt pietiekami liela funkcionalitāte, kas nodrošinātu plašākas testēšanas iespējas un neierobežotu testa gadījumu izvēlē.

Tā, ka autoram ir pieredze zīņapmaiņas lietotņu testēšanā, lai patērētu mazāk laika izvēlētas lietotnes izpētei, tika izskatīti vairākas zīņapamaiņas mobilās lietotnes. Testēšanai autors izvēlējas Telegram Messenger [34] 3.7.0 versijas lietotni.. Tā ir bezmaksas atvērta pirmkoda teksta īsziņu un multivides apmaiņas dažādplatoformu mobilā lietotne, kas paredzēta WP, iOS un Android ierīcēm, līdz ar to, tā pilnībā atbilst autora izvirzītam prasībām.

5.2. Grafiskās lietotāja saskarnes testēšanas automatizācijas rīki

Kā jau iepriekš tika minēts, grafiskās lietotāja saskarnes līmeņa testēšana, ir process, kurā gaitā tiek pārbaudīta lietotnes funkcionalitāte caur lietotnes grafisko lietotāja saskarni, tādā veidā salīdzinot esošo rezultātu ar paredzamo. Protams, eksistē daudz dažādu lietotāja saskarnes testēšanas automatizācijas rīku, tāpēc, lai demonstrētu dažādas testu automatizācijas iespējas un paņēmienus, autors ir nolēmis izvēlēties tādus rīkus, kur testu automatizācijas process atšķiras.

Pētot vairākus avotus un materiālus [35,36,37], darba autors ir izvēlējies 4 atšķirīgus mobilo lietotņu testēšanas automatizācijas rīkus, kuri ir paredzēti Android vai dažādplatformu mobilo lietotņu testēšanai:

- Appium – iOS, Android
- Calabash – iOS, Android
- Robotium Recorder - Android
- eggPlant – iOS, Android

Testēšanai tika izmantota Sony Xperia SP Android 4.3 mobilā ierīce, ka papildus ierīce tika izmantota LG Spirit C70 Android 5.0.1. Rīku analīzei un testu izpildes laika mērīšanai tika izveidota testa kopa, kas satur sekojošus testēšanas gadījumus un soļus:

Tabula 5.1.

Telegram lietotnes testēšanas gadījumi

Testēšanas gadījums	Soļi
Nekorekta telefona numura ievadīšana	• Atvērt lietotni • Nospiest pogu “Start Messaging” • Ievadīt telefona numuru ievadlaukā, kura ciparu skaits neatbilst prasītām • Nospiest “Done” pogu • Nospiest “Ok” uznirstošā laukā
Sveša telefona numura ievadīšana	• Ievadīt telefona numuru, kas nav šīs ierīces telefona numurs • Nospiest “Done” pogu • Nospiest “Wrong number?” pogu
Pieteikšanās lietotnē	• Ievadīt dotas ierīces telefona numuru ievadlaukā • Nospiest “Done” pogu • Sagaidīt īsziņu ar pieteikšanās kodu
Īsziņās sūtīšana citam lietotajam	• Nospiest “Menu” pogu • Nospiest “Contacts” pogu • Atrast lietotāju kontaktu sarakstā • Uzsākt jaunu ziņojumapmaiņas sesiju, uzspiežot uz atrasto lietotāju • Uzrakstīt tekstu ievadlaukā • Nospiest “Send” pogu

Atteikšanās no lietotnes	• Nospiest “Menu” pogu • Nospiest “Settings” pogu • Nospiest “Papildus” pogu • Nospiest “Log out” • Nospiest “Ok” uznirstošā laukā

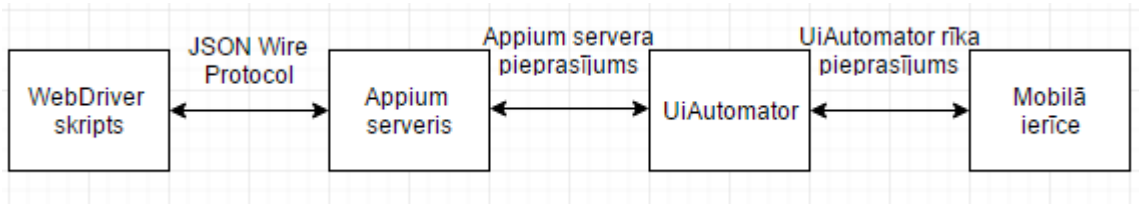
Pēc tabulā 5.1 aprakstītiem testa gadījumiem un soļiem tika izveidoti testu automatizācijas skripti.

5.2.1. Appium rīks

Appium ir atvērta pirmkoda bezmaksas testu automatizācijas rīks, kas ir paredzēts tieši mobilo lietotņu testēšanas automatizācijai, gan konkrētas platformas, gan hibrīda, gan tīmekļa lietotnēm. Appium atbalsta Android, iOS un Firefox OS mobilās operētājsistēmas.

Rīkā tiek izmantots Node.js serveris savienojumam ar mobilo ierīci, un Selenium WebDriver testu izpildīšanai, līdz ar to, Appium rīks atbalsta visas WebDriver atbalstāmas programmēšanas valodas: Java, JS, C#, Haskell, Objective-C, Perl, PHP, Python, R, Ruby.

Testu izpilde notiek sekojoši: izmantojot *JSON Wire Protocol* Appium serverim tiek nosūtītas komandas. Appium serveris pārstrādā saņemtas komanda, izveido automatizācijas sesiju, un savienojas ar UiAutomator automatizētas testēšanas rīku. UiAutomator rīks nosūta pieprasījumu ierīcei, kur tiek izpildītas nosūtītas komandas [38]. Tad no ierīces tiek sūtīta atbilde ar soļu izpildes rezultātiem. Kad serveris ir saņēmis atbildi, komandu izpildes rezultāti tiek attēloti konsolē. Shematiski testu izpildes secību var redzēt attēlā 5.1.



5.1. att. **Appium rīka soļu izpildes secības**

5.2.1.1. Testu automatizācija

Appium rīks nodrošina tikai manuālu testu automatizāciju. Testu automatizācija tiek veikta *Eclipse Mars* programmēšanas vidē. Kā iepriekš tika minēts, testu pierakstīšanu var veikt vairākas

programmēšanas valodas, autors izvēlējas Java programmēšanas valodu, jo iepriekš bija pieredze darbā ar Java.

Sākumā, pirms veikt testu automatizāciju, ir nepieciešams pieslēgs visas nepieciešamas bibliotēkas, kuras ir pieejamas Appium rīka mājaslapā. Pēc nepieciešamo bibliotēku pieslēgšanas, ir nepieciešams izveidot savienojumu starp Appium rīku un mobilo ierīci, noradot ierīces vārdu, operētājsistēmu, tas versiju, mobilās lietotnes atrašanas vietu datorā un savienojuma URL, lai rīks varētu pieslēgties pie ierīces. Autora izmantotas pieslēgšanas konfigurācijas var redzēt attēla 5.2:

```
AndroidDriver driver;

//Pieslegsanas konfiguracijas
@Test
public void AndroidConfig() throws MalformedURLException{

    DesiredCapabilities capability = new DesiredCapabilities();
    capability.setCapability("deviceName", "C5303");
    capability.setCapability("platformName", "Android");
    capability.setCapability("deviceVersion", "4.3");

    File file = new File("C:\\Users\\Pavel\\workspace\\AutoTest\\apks\\telegram3-7-0.apk");
    capability.setCapability("app", file.getAbsolutePath());

    driver = new AndroidDriver(new URL("http://192.168.1.2:4722/wd/hub"), capability);
    }
```

5.2. att. **Appium rīka pieslēgšanas konfigurācijas mobilajai ierīcei**

Uzreiz pēc pieslēgšanas konfigurēšanas var uzsākt testu rakstīšanu. Testu izpildei tika izmantots TestNG satvars, tāpēc uzsakot testu automatizāciju, lai rīks spētu identificēt testu, pirms testa pierakstīšanas ir jāpievieno rinda @Test, pēc kuras jādefinē tests ka funkciju, kurā iekšā tiek definētas darbības, kuras rīkam ir jāveic lietotnē.

Autora izveidotais kods testa gadījumam “Īsziņās sūtīšana citam lietotajam” ir redzams attēlā 5.3.

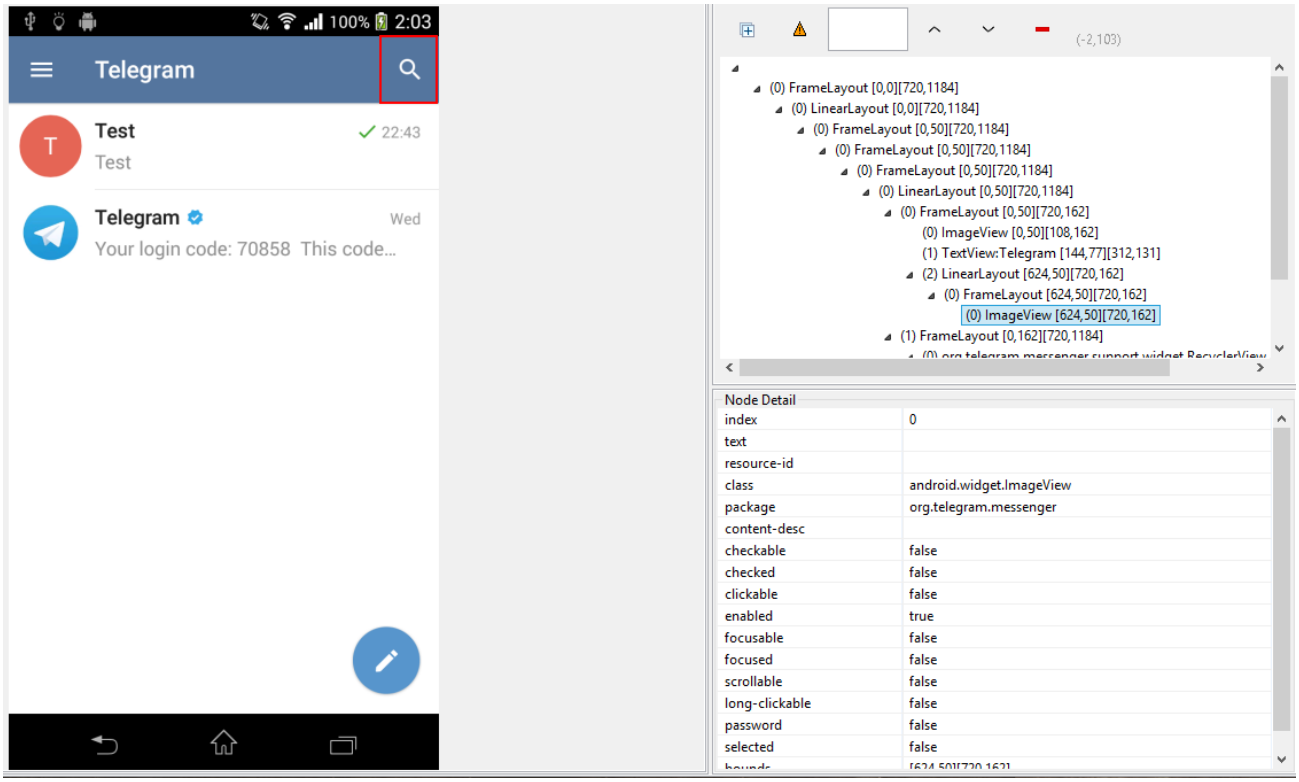
```
@Test
public void WriteMessage() throws InterruptedException{

    WebDriverWait wait = new WebDriverWait(driver, 50);
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//android.widget.TextView[contains(@text, 'Telegram')]")));
    driver.findElement(By.className("android.widget.ImageView")).click();   //Nospiest "Menu" pogu
    driver.findElement(By.name("Contacts")).click(); //Nospiest "Konktakti" pogu

    //gaidit kamer atversies kontaktu skats
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//android.widget.TextView[contains(@text, 'Contacts')]")));
    driver.findElement(By.xpath("//android.widget.FrameLayout[@index='2']")).click(); //izveleties "Test" kontu

    //ierakstit ievadlaukaa tekstu "Test"
    driver.findElement(By.xpath("//android.widget.EditText[contains(@text, 'Message')]")).sendKeys("Test");

    //Nospiest "Send" pogu
    driver.findElement(By.xpath("//android.widget.FrameLayout/android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/"
            + "android.widget.FrameLayout[1]/android.widget.FrameLayout[1]/"
            + "android.widget.LinearLayout[1]/android.widget.FrameLayout[2]/"
            + "android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/"
            + "android.widget.FrameLayout[@index='1']/android.widget.ImageView")).click();

    driver.findElement(By.className("android.widget.ImageView")).click(); //atgriezties uz sakuma skatu
}
```

5.3. att. **Testa automatizācijas piemērs Appium rīkā izmantojot TestNG**

Kodā tiek izpildītas visas darbības, kas ir definētas tabulā 5.1. “Īsziņās sūtīšana citam lietotajam” testa gadījumam. Pārējo testa gadījumu autora izveidoto kodu Appium rīkam var redzēt pielikumā 1.

Kodā lietotāju saskarnes elementu identificēšana notiek pēc klases nosaukuma, atslēgvārdiem un XPath vaicājuma. Lietotāja saskarnes elementu definēšanai tika izmantots *uiautomatorviewer* rīks, kas ir iekļauts Android programmizstrādes komplektā. *Uiautomatorviewer* rīks nodrošina Android ierīču ērtu ekrānā attēloto lietotāja saskarnes komponenšu skenēšanu un analīzi [39].

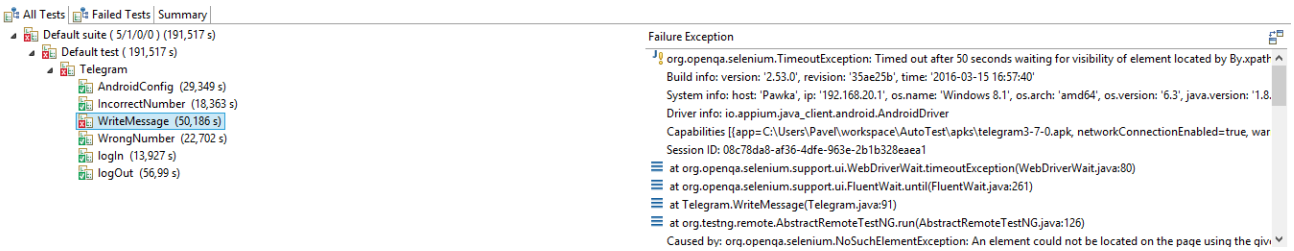
Rīks ir viegls izmantošanā, vienīgais, kas ir jāizdara testētājam, ir jāpieslēdz mobila ierīce pie datora caur USB vadu, jāpalaiž testējama lietotne un jānospiež “Device Screenshot” poga. Tad rīks izveido mobilā ierīces ekrānuzņēmumu un identificēs katru lietotāja sakasrnes elementu. Testētājam būs pieejama XML elementu izkārtojuma hierarhija un klases, kā arī vairāku elementu ID un klases. Izmantojot šo informāciju, autors spēja viegli identificēt un piekļūt katram lietotnes lietotāja saskarnes elementam. Līdz ar to, lielu grūtību ar testu automatizāciju nebija.



5.4. att. **Uiautomatorview rīka ekrānuzņēmums**

Attēlā 5.4. ir redzams *Uiautomatorviewer* rīka darbības piemērs ar izveidoto ierīces ekrānuzņēmums, kurā var redzēt testējamo lietotni Telegram. Rīkā ir izveidota XML elementu hierarhija, kura ir attēlota labajā augšējā stūrī, kā arī dota informācija par ekrānā izvēlēto elementu – zem XML elementu hierarhijas.

Pēc testu izpildes, testētājam ir pieejami testu rezultāti. Autora saņemto automatizētas testu kopas izpildes rezultāta piemēru var redzēt attēlā 5.5.



5.5. att. **Automatizēto testu izpildes rezultātu piemērs**

Testa rezultātu informācija ir vienkāršota: testētājam ir pieejami testu izpildes rezultāti (izpildītas vai nav izpildīts) un izpildes laiks. Neveiksmīgi izpildītiem testiem tiek noradīts iemesls, kāpēc tests netika izpildīts. Papildus koda var izveidot funkciju, kas veiks ekrānuzņēmumu uzņemšanu kļūdas rašanas gadījumā.

Runājot par testu ātrdarbību, testu kopas izpildes vidējais laiks normālos apstākļos – bez jebkāda veida traucējumiem un pārtraukumiem – ir 119 sekundes.

5.2.1.2. Grūtības un problēmas rīka izmantošanā

Veicot testu automatizāciju, izmantojot Appium rīku, autors ir sastapies ar vairākām problēmām un grūtībām.

Jau no paša rīka izmantošanas sākuma autoram rādās problēmas rīka konfigurēšanā. Tā, ka oficiālajā dokumentācija informācija par rīka konfigurēšanu Windows operētājsitēmā ir dota minimāli, nācās meklēt vairākus citus avotus, lai iedarbināt rīku.

Uzsākot automatizētu testēšanu tika novērots, ka rīks laiku pa laikam mēdz izbeigt automatizēto testu uzreiz pēc palaišanas, rezultātā rādot, ka tests ir neveiksmīgs. Šo problēmu var atrisināt, restartējot Appium rīku.

Vēl viena, un manuprāt nozīmīga rīka problēma ir tas, ka rīks ir paredzēs samēra jaunu mobilo operētājsistēmu versijām - Android versijām ar API 17 un lielāku, tas nozīmē, ka testēšanu var veikt tikai sākot no Android 4.2 versijas un jaunākām. Tādā gadījumā, lai veiktu testēšanu uz vecākam operētājsistēmas versijām, būs nepieciešam meklēt papildus rīku.

5.2.1.3. Secinājumi par rīku

Rīka pielietošanai testētājam ir noteikti ir jābūt programmēšanas prasmēm, lai spētu pietiekami labi un korekti automatizēt testus. Testētājam bez programmēšanas iemaņām var rasties grūtības testu automatizācijā. Tomēr ja testētājs ir apguvis kādu no populāram programmēšanas valodām, Appium rīks tam noteikti derēs, pateicoties tā atbalstāmam programmēšanas valodu skaitam. Testētājs var brīvi veikt testu automatizāciju viņam zināmā programmēšanas valodā, netērējot laiku jaunas valodas apgūšanai.

Neskatoties uz to, ka rīkā tika testētā Android mobilā lietotne, Appium rīks ir piemērots arī iOS mobilām ierīcēm, un testa koda bāze gan Androim, gan iOS lietotnēm ir vienāda, kas liecina par rīka unikalitāti.

Neskatoties visām priekšrocībām, Appium rīka testēšanas laikā autors ir konstatējis, ka rīks ir diezgan nestabils, jo tika novēroti vairāki gadījumi, kad automatizēto testu vienkārši nevarēja uzsākt, vai arī tests beidzās mirkli pēc uzsākšanas.

5.2.2. Calabash rīks

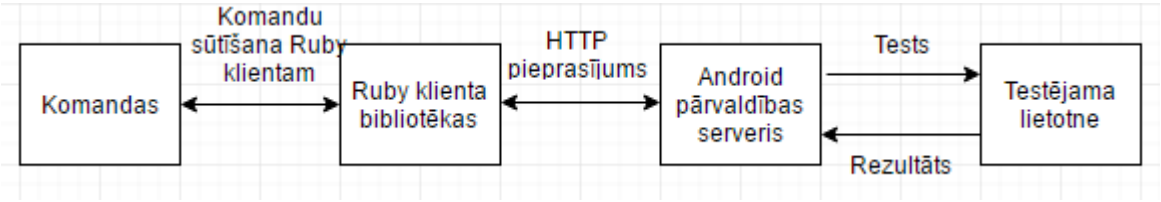
Calabash ir bezmaksas dažādplatformu mobilo lietotņu testēšanas rīks, kas ir paredzēts konkrētas platformas un tīmekļa lietotņu testēšanai iOS un Android operētājsistēmās. Testēšanu var veikt gan uz reālām ierīcēm, gan uz emulatoriem.

Lietotāja saskarnes elementu atpazīšanai rīkā tiek izmantotas trīs pieejas – objektu atpazīšana, tekstu atpazīšana, un koordinātu atpazīšanas pieeja.

Testu automatizācija notiek Gherkin dabiskā valodā, kuru izmanto Cucumber testa soļu definēšanai. Soļu darbības, savukārt, tiek definētas Ruby programmēšanas valodā.

Android mobilo lietotņu testu automatizācijai, Calabash izmanto Android pārvaldības serveri, kas balstās uz *ActivityInstrumentationTestCase2* klasi, kas ir paredzēts Android lietotnes aktivitāšu testēšanai [40]. Pārvaldības serveris tiek automātiski uzstādīts uz mobilo ierīci testēšanas sākumā kā atsevišķa lietotne.

Komandas, definētas Gherkin valodā, pirms sūtīšanas serverim, tiek tulkotas Ruby klienta bibliotēkās, tad caur HTTP pieprasījumu, komandas tiek nosūtītas serverim, caur kuru arī tiek izpildītas. Pēc testu komandu izpildes, serveris saņem atbildi par to izpildi, kuri tālāk tiek atgriezti testētājam. Vizuāli testu izpildes secību var redzēt attēlā 5.6.



5.6. att. **Calabash rīka soļu izpildes secības**

5.2.2.1. Testu automatizācija

Calabash testu automatizācija, ka jau bija minēts, notiek izmantojot Gherkin un Ruby programmēšanas valodu. Gherkin valodā notiek testa soļu pierakstīšana dabiskā valodā, un izmantojot Ruby notiek šo soļu definēšana. Soļus iespējams definēt jebkurā no 60 Cucumber atbalstāmā dabiskām valodām, kurā skaitā ir arī latviešu. Testa kopas automatizāciju autors ir veicis angļu valodā, tomēr ir izmēģinājis arī latviešu valodas testa gadījuma pierakstu. Testa automatizācijai tika izmantots teksta redaktors Atom [41]. Visas darbības ar rīku tiek veiktas caur komandrindu.

Lai Calabash rīks spētu atrast testu kopu, ir jādefinē tās nosaukums – *Feature: Testa kopas nosaukums,* latviski tas būtu – *Fīča: Testa kopas nosaukums*.

Katru atsevišķu testa gadījumu ir jāsak ar tā nosaukumu - *Scenario: Scenārija nosaukums*, kur mēs nosākam, ka tas ir jauna scenārija sākums. Latviski tas būtu jādefinē sekojoši - *Scenārijs: Scenārija nosaukums*.

Kad ir definēts scenārijs, var sākt ar scenārija soļu definēšanu. Kā jau iepriekš minēju, scenāriju apraksts notiek dabiskā valodā. Calabash rīkā iepriekšizveidoti dabiskās valodas soļu definējumi ir atrodami pirmkoda mapē "steps" [42], vai arī tos var definēt patstāvīgi datnē *calabash_steps.rb*, kas atrodas Calabash rīka *step_definitions* mapē.

Autora izveidoto soļa definīcijas piemēru var redzēt attēlā 5.7.

```
Then /^I press imageview number (\d+)$/ do |index|
  tap_when_element_exists("android.widget.ImageView index:#{index.to_i-1}")
end
```

5.7. att. **Calabash soļu definēšanas piemērs**

Soļa definīcijā tiek dots soļa pieraksts dabiskā valodā, kā tas tiks uzrakstīts testa scenārijā. Nākošā rindā tiek aprakstītas darbības, šajā gadījumā – atrast elementu ar klasi *android.widget.ImageView* un aprakstā padoto indeksu. Šī soļa izsaukuma piemērs izskatās sekojoši: *"Then I press imageview number 1*", tas nozīmē, ka tiks atrast *ImageView* elements ar indeksu 0, kurš pēc kārtas ir pirmais šīs klases elements.

Autora izveidotais kods testa gadījumam "Īsziņās sūtīšana citam lietotajam" ir redzams attēlā 5.8. Pārējo testa piemēru definējumu var redzēt pielikumā 2.

```
Scenario: I can write message to another user
   When I see text "Telegram"
   Then I press image number 2
   Then I press textview number 7
   Then I press SimpleTextView number 1
   Then I enter "Test" text into message input field number 1
   Then I press image number 4
   Then I press image number 2
   Then I see the text "Telegram"
```

5.8. att. **Calabash testa gadījuma pieraksta piemērs**

Attēla 5.8. var redzēt, ka testa soļu aprakstam tiek izmantota angļu valoda, līdz ar to testa soļi ir viegli saprotami jebkuram angliski runājošam cilvēkam.

Autors arī ir veicis testa gadījuma "Īsziņās sūtīšana citam lietotajam" automatizāciju, pierakstot to latviešu valodā, pirms tam definējot testa soļus latviski. Lai paziņotu rīkam, kādā valodā tiks padoti testa soļi, pirms *Fīčas* nosaukuma ir jādefinē rīka soļu pieraksta izmantojama

valoda, piemēram - *# language: lv*. Lai uzzināt Cucumber atbalstāmas valodas un to saīsināto nosaukumu, komandrinda jāievada *cucumber --i18n help.*

Testa gadījuma "Īsziņās sūtīšana citam lietotajam" soļu definējuma piemērs latviešu valodā var redzēt attēlā 5.9.

```
# language: lv

Fīča: Īsziņas rakstīšana

Scenārijs: Kā pieteiks lietotājs es varu rakstīt īsziņas
  Ja Es redzu tekstu "Telegram"
  Tad Es nospiežu uz attēlu ar numuru 2
  Tad Es nospiežu uz textview elementu ar numuru 7
  Tad Es nospiežu uz SimpleTextView elementu ar numuru 1
  Tad Es ievadu "Test" tekstu īsziņas ievadlaukā ar numuru 1
  Tad Es nospiežu uz attēlu ar numuru 4
  Tad Es nospiežu uz attēlu ar numuru 2
  Tad Es redzu tekstu "Telegram"
```

5.9. att. **Calabash testa piemēra pieraksts latviešu valodā**

Pēc testu kopas izpildes, komandrindā tiek izvadīts testu izpildes rezultāts. Autora saņemta testa izpildes rezultāta piemēru var redzēt attēla 5.10.

```
  Scenario: I can logout                  # features/my_first.feature:41
    When I see text "Telegram"            # features/step_definitions/calabash_steps.rb:3
    Then I press imageview number 2       # features/step_definitions/calabash_steps.rb:8
    Then I press textview number 9        # features/step_definitions/calabash_steps.rb:20
    Then I press imageview number 1       # features/step_definitions/calabash_steps.rb:8
    Then I press textview number 2        # features/step_definitions/calabash_steps.rb:20
    Then I press view with id "button1" # calabash-android-0.7.3/lib/calabash-android/steps/press_button_steps.rb:13
    Then I see the text "Telegram"        # calabash-android-0.7.3/lib/calabash-android/steps/assert_steps.rb:1

5 scenarios (5 passed)
36 steps (36 passed)
1m45.788s
```

5.10. att. **Calabash testu izpildes rezultātu piemērs**

Testu izpildes rezultātos tiek paziņots par veiksmīgi izpildītiem, izlaistiem vai neveiksmīgi izpildītiem testa gadījumiem un soļiem. Gadījumā, ja solis netiek izpildīts, tiek uzņemts ekrānuzņēmums, pēc kura vieglāk var saprast kur ir notikusi kļūda, līdz ar to, ir vieglāk atkārtot šo kļūdu un piefiksēt.

Testa gadījuma izpildes atskaiti var arī saņemt HTML formātā, norādot to testu palaišanas sākumā, pievienojot – "*--format html --out reports.html*".

5.11. att. **Calabash HTML atskaite**

Atskaitē tiek izvadīta tieši tāda pati informācija, kā komandrindā, tomēr tāda veida atskaite ir daudz patīkamāka acīm un labāk pārskatāmāka.

Elementu meklēšana tika veikta izmantojot Calabash Ruby API, caur kuru notiek pieslēgšana testa serverim, kurš atrodas mobilajā ierīcē un caur kuru notiek sadarbība ar lietotni. Pieslēdzoties testa serverim ir iespējams identificēt ekrānā redzamos UI elementus, kuri tiek izvadīti komandrindā. Līdzīgi, ka *UIAutomatorViewer* rīkā, testētājam tiek paziņota elementu ID, klases nosaukums, koordinātes un citi elementa parametri, tomēr vairākas reizes autors ir konstatējis, ka Calabash Ruby API identificētie lietotnes lietotāja saskarnes elementu klases atšķiras no *UIAutomatorViewer* rīkā elementu klasēm, kā arī elementu indeksi abos rīkos nesakrīt. Līdz ar to, *UIAutomatorViewer* rīks Calabsh testu automatizācijai neder.

Testu kopas vidējais izpildes laiks ir 113 sekundes.

5.2.2.2. Grūtības un problēmas rīka izmantošanā

No paša sākuma autoram radās problēmas ar rīka konfigurēšanu. Oficiālajā dokumentācija, līdzīgi ka Appium rīkam, trūka detalizācijas, tāpēc autoram nācās meklēt papildus avotus ar Calabash rīka konfigurācijas aprakstu [43].

Otra problēma, ar kuru sastapās autors, ir lietotāja saskarnes elementu identificēšana. Kaut arī IRB nodrošina iespēju atpazīt lietotāja saskarnes elementus, autors vairākas reizes ir sastapies ar to, ka ievietojot kodā atrasto elementa klases nosaukumu vai indeksu, rīks nespēja to elementu korekti identificēt. *UIAutomatorViewer* rīks šajā gadījumā arī nepalīdzēja, jo bieži vien elementa

klase, kas tika dota *UIAutomatorViewer* rīkā nederēja. Lai atrisinātu šo problēmu, vairākas reizes elementus nācās meklēt pēc atlasīšanas metodes.

Vēl viena būtiska problēma ir tas, ka katra testu kopas palaišanas reizē, rīks no jauna uzinstalē lietotni uz mobilās ierīces. Autoram sanāca atrisināt šo problēmu, izmainot testu sākšanas nostatījumus *app_installation_hooks.rb* datnē. Tas bija nepieciešams, jo veicot katras jaunas testu kopas testēšanu, tika tērēts laiks lietotnes uzstādīšanai – apmēram 1 minūte.

Pēdējā problēma bija saistīta ar to, ka katra jauna testa izpildes sākuma, lietotne tika pārstartēta. Līdz ar to, katru jaunu testa kopu ir jāsāk ar pieteikšanās sistēmā, kas ietekmē automātisku testu izpildes laiku.

5.2.2.3. Secinājumi par rīku

Pateicoties Cucumber testu automatizācija notiek samērā viegli. Kā arī testu automatizācijas kļūdas rašanas gadījumā, rīks piedāvā problēmas risinājumu, piemēram, ja tests Gherkin valodā ir definēts nepareizi, testētājam automātiski tiek piedāvāts uzģenerēts Ruby programmēšanas valodā kods, kas definē neatrasto/ nepareizi definēto soli.

Rīks ir samērā viegli apgūstāms, ja iepriekš bija pieredze darbā ar komandrindu. Tomēr ja komandrindā iepriekš nebija strādāts, internetā ir pieejami vairāki materiāli, kuros ir pietīkoši saprotami izskaidrots, kā jāstrādā ar Calabash rīku komandrindā.

Runājot par testu izpildes ātrumu, rīks nav pārāk ātrs, izpildes laiks ir vidējais.

Lielāka šī rīka problēma, ir lietotāja saskarnes elementu definēšana, kas laiku pa laikam var nesakrist ar to reālo nosaukumu, līdz ar to testētājam ir jātērē laiks, lai atlasītu visas iespējamus elementa identificēšanas variantus.

Kopumā rīks liekas diezgan stabils, nekādu traucējumu rīka darbība netika novērots.

5.2.3. Robotium rīks

Robotium ir atvērta pirmkoda automatizētas testēšanas rīks, kas ir paredzēts Android mobilo lietotņu melnas kastes testu automatizācijai [44]. Robotium ir paredzēts konkrētas platformas un hibrīda mobilo lietotņu testēšanai. Testu automatizācija jāveic Java programmēšanas valodā.

Testējamas lietotnes lietotāja saskarnes elementu atpazīšanai, tāpat kā Appium rīkā, tiek izmantota objektu atpazīšanas pieeja, teksta un koordinātu atpazīšanu balstīta pieeja.

Rīkam ir pieejama detalizēta dokumentācija, kur solis pa solim tiek aprakstīta rīka konfigurēšana Eclipse un Android Studio programmēšanas vidēm.

Robotium rīks, tapāt ka Calabash rīks, automātisko testu izpildei izmanto Android pārvaldības serveri, kas balstās uz *ActivityInstrumentationTestCase2* klasi. Palaižot automātisku testu izpildi, ierīcē tiek izveidots pārvaldības serveris, kā atsevišķa lietotne, kurš tālāk jau izpilda automatizēto testu soļus testējamā lietotnē, atgriežot testa soļu izpildes rezultātus, kuri testētājam tiek attēloti konsolē.

5.2.3.1. Testu automatizācija

Robotium rīks nodrošina manuālu testu automatizāciju, tomēr ir pieejams papildus maksas rīks Robotium Recorder, kas automātiski ģenerē testa kodu atkārtojot lietotāja darbības.

Testa koda veidošanu var veikt Eclipse vai Android Studio izstrādes vīdēs. Autors izvēlējas veikt testu automatizāciju Eclipse vidē, jo kā jau iepriekš bija rakstīts, autoram bija pieredzē darbā šajā izstrādes vidē.

Pirms uzsākt testēšanu, ir jāizveido Android testēšanas projekts, kur AndroidManifest.xml failā ir jānorāda testējamas lietotnes pakotnes nosaukums - *org.telegram.messenger*. Kad tas ir izdarīts, ir jāizveido jauna pakete, caur kuru notiks savienojums ar testējamo lietotni, tajā noradot lietotnes palaišanas klases nosaukums - *org.telegram.ui.LaunchActivity*.

Kad viss ir pareizi ievadīts un sakonfigurēts, var ķertie pie testu automatizācijas. Robotium rīkā testēšanai tiek izmantota *Solo* klase, tāpēc ir jādefinē klases objektu, lai turpmāk izsauktu metodes, kas veic noteiktas darbības mobilajā ierīc ē.

Definējot testu, tā nosaukumam ir jāsākas ar vārdu *test,* piemēram, testWriteMessage, lai JUnit spētu atpazīt doto metodi ka testu. Autora izveidotais testa gadījuma "Īsziņās sūtīšana citam lietotajam" kods ir redzams attēla 5.12. Pārējo testa gadījumu automatizācijas kodu var redzēt pielikumā 3.

```
public void test_writeMessage() {

    solo.waitForActivity("LaunchActivity"); //gaidam kamer ieslegsies lietotne
    solo.clickOnImage(1); //nospiezam "Menu" pogu
    solo.clickOnText("Contacts"); //izvelamies kontaktu sarakstu
    solo.clickInList(3); //izvelamies kontaktu, kuram reakstisim iszinu
    solo.clickOnText("Message"); //nospiezam uz ievadlauku
    solo.enterText(0, "Test"); //ievadam ievadlauka tekstu "Test"
    solo.clickOnImage(3); //nospiezam "Send" pogu
    solo.clickOnImage(1); //izejam no zinapmajnas
}
```

5.12. att. **Testa automatizācijas piemērs Robotium rīkā izmantojot JUnit**

Objektu atpazīšanai var neizmantot papildus rīki, jo objektus samērā viegli identificēt, pateicoties tam, ka Robotium rīks patstāvīgi piešķir indeksus lietotāja saskarnes elementiem sākot no augšējā kreisa stūra līdz apakšējam labajam stūrim. Līdz ar to, objektu ir viegli identificēt, saskaitot to kārtas numuru starp vienādiem elementiem. Tomēr, ja ir nepieciešams noskaidrot elementa klasi vai ID var izmantot jau iepriekš minēto *UiAutomatorViewer* rīku.

Vēl viena iespēja veikt testu automatizāciju Robotium rīkā ir izmantot papildinājumu Robotium Recorder – maksas rīks, kas automātiski veic testu automatizāciju Robotium rīkam. Lai izpētītu rīku, autors ir izmantojis rīka bezmaksas izmēģinājumversiju. Lai uzsāktu testu automatizāciju, rīkam ir jānorada testējamas lietotnes APK datni vai projektu izstrādes vidē. Kad lietotne ir norādīta, nospiežot "Start Recording" pogu var uzsākt automātisku testu izvedi.

Testu automātiska izveide notiek sekojoši: testētājs veic nepieciešamas darbības mobilajā ierīcē. Šīs darbības tiek attēlotas Robotium Recorder rīka logā. Kad darbības, kurās ir nepieciešams rīkam atkārtot ir ierakstītas, ir jānospiež poga "Stop Recording". Pēc ierakstīšanas, testētājs var dzēst kādu no darbībām. Jā visas ierakstītas darbības atbilst prasītām, testētājs saglabā testu – ierakstītas darbības tiek pārveidotas kodā.

Automātiski izveidota koda piemērs Robotium Recorcder rīkā ir redzams attēla 5.13.

```
public void testRun() {
    // Wait for activity: 'org.telegram.ui.LaunchActivity'
    solo.waitForActivity("LaunchActivity", 2000);
    // Click on ImageView
    solo.clickOnView(solo.getView(android.widget.ImageView.class, 1));
    // Click on Contacts
    solo.clickInList(7, 0);
    // Click on UserCell
    solo.clickInList(3, 0);
    // Click on Empty Text View
    solo.clickOnView(solo.getView(android.widget.EditText.class, 0));
    // Enter the text: 'Test '
    solo.clearEditText((android.widget.EditText) solo.getView(android.widget.EditText.class, 0));
    solo.enterText((android.widget.EditText) solo.getView(android.widget.EditText.class, 0), "Test ");
    // Click on ImageView
    solo.clickOnView(solo.getView(android.widget.ImageView.class, 3));
    // Click on ImageView
    solo.clickOnView(solo.getView(android.widget.ImageView.class, 1));
}
```

5.13. att. **Automātiski izveidots tests Robotium Recorder rīkā**

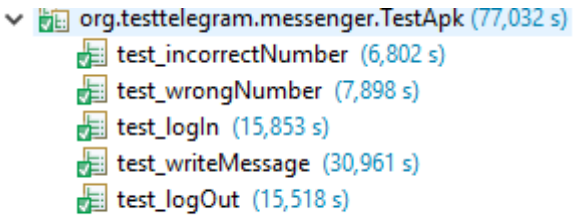
Var redzēt, ka atšķirībā no autora manuāli rakstīta koda, Robotium Record rīks elementus meklē tikai izmantojot objektu atpazīšanas pieeju, savukārt autors kombinēju objektu atpazīšanu ar tekstu atpazīšanas pieeju. Automātiski izveidots tests precīzi atkārtojat visas prasītas darbības. Pēc vairāku automātiskas testu automatizācijas mēģinājumiem, autors ir konstatējis ka Robotium Recorder rīks precīzi spēj atpazīt UI elementus un atkārtot testētāja lietotnē veiktas darbības.

Kad ir automatizēti visi nepieciešami testa gadījumi, ir jāizveido testkomplekts. Tas ir nepieciešams tādēļ, ka JUnit satvarā testi tiek izpildīti alfabētiskā secībā, tāpēc, lai noteikt testētājam nepieciešamo secību, ir jāizveido testkomplekts, kurā tiek noradīta testu izpildes secība. Autora izveidotais testkomplekts ir redzams attēla 5.14.

```
public static TestSuite suite(){
    TestSuite t = new TestSuite();
    t.addTest(TestSuite.createTest(TestApk.class, "test_incorrectNumber"));
    t.addTest(TestSuite.createTest(TestApk.class, "test_wrongNumber"));
    t.addTest(TestSuite.createTest(TestApk.class, "test_logIn"));
    t.addTest(TestSuite.createTest(TestApk.class, "test_writeMessage"));
    t.addTest(TestSuite.createTest(TestApk.class, "test_logOut"));
    return t;
}
```

5.14. att. **Robotium testkomplekta izveide**

Tiek izveidota metode ar nosaukumu suite(), lai JUnit varētu noteikt, ka tas ir testkomplekts. Metodē tiek izveidots jauns testkomplēkts, kurā tiek pievienotas testa piemēri nepieciešamā secībā. Galā tiek atgriezts šīs testkomplekts.



5.15. att. **Robotium testa rezultāta piemērs**

Pēc testu izpildes, testētājam tiek atgriezti testu izpildes rezultāti, kura var redzēt vai tests tika vai netika veiksmīgi izpildīts, vai arī radās kāda kļūda notika testa izpildes laikā. Katram neizpildītam testa gadījumam ir pieejams kļūdu žurnāls, kur var redzēt testa neizpildīšanas cēloni. Autora saņemta testa rezultāta piemēru var redzēt attēlā 5.15.

Veicot testa kopas izpildes laika mērījumu, testu kopas izpildes vidējais laiks ir 78 sekundes.

5.2.3.2. Grūtības un problēmas rīka izmantošanā

Pirmā grūtība, ar ko sastapies autors ir tas, ka pirms uzsākt lietotnes automatizēto testēšanu, ir nepieciešams pārveidot testējamas lietotnes APK failu, ieslēdzot lietotnē atkļūdošanas režīmu. Atrisinājums šai problēmai tika atrast Robotium rīka dokumentācijā, kur tika piedāvāts šīs problēmas divi atrisinājuma veidi: manuāli, caur komandrindu, un automātiski, izmantojot *re-sign* lietotni.

Kā nelielu trūkumu var arī minēt, ka rīks automātiski nespēj atbloķēt mobilās ierīces ekrānu, tomēr šī problēma ir atrisināma ar Android ierīču iebūvēto "Palikt nomodā" funkciju, kas ir viena no "Izstrādātājiem paredzētam iespējam".

5.2.3.3. Secinājumi par rīku

Kopumā par rīku palika labs iespaids. Vairāku lietošanas dienu laikā autors ir konstatējis, ka Robotium rīks ir pietiekami stabils, jo nekādu traucējumu vai nopietnu problēmu rīka izmantošana autors nav novērojis.

Testu automatizācija ir samēra vienkārša, jo visas komandas un metodes ir intuitīvi saprotami angliski runājošiem cilvēkiem, kā arī objektu identifikācija ir samēra viegla pateicoties Robotium rīka elementu indeksēšanai. Pateicoties tam, nav obligāti jāzin lietotāja saskarnes meklējamo elementu ID.

Robotium Recorder arī ir diezgan spēcīgs papildinājums, kas spēj precīzi veikt automatizēto testu automatizāciju, līdz ar to testētājam ir nepieciešamas minimālas zināšanas par testējamo lietotni, un pateicoties tam testu automatizācijas laiks samazinās.

Testu izpildes laiks arī ir samēra ātrs.

5.2.4. eggPlant rīks

eggPlant rīks ir kompānijas TestPlant izstrādātais testu automatizācija rīks kas ir paredzēts mobilo lietotņu grafiskas saskarnes testēšanai. Rīks izmanto attēlu atpazīšanas pieeju lietotāja saskarnes elementu atpazīšanai, līdz ar to, rīks ir neatkarīgs no ierīces platformas, testus var veikt uz iOS, Android, WP, BlackBerry ierīcēm [20]. Kā arī testēšanu var veikt uz visiem lietotņu veidiem – hibrīda, konkrētas platformas un tīmekļa. Testu automatizāciju veic SenseTalk skriptēšanas valodā.

Rīka tiek izmantots VNC sistēma mobilās ierīces ekrāna attēla saņemšanai un kontrolei datorā. Tāpēc uz testējamās ierīces un datora, no kura tiks veikta testēšana, ir jābūt uzstādītam VNC serverim.

eggPlant ir maksas rīks, kura cena ir atkarīga no tā, cik daudz cilvēku rīku izmantos. Vienam rīka lietotājam tās izmaksās – 5730 eiro/gadā, un grupas lietojumam – 9550 eiro/gadā. Licence tiek iegādāta vienam datoram, tomēr testētāji var izmantot šo rīku attālināti no savam ierīcēm vai datoriem.

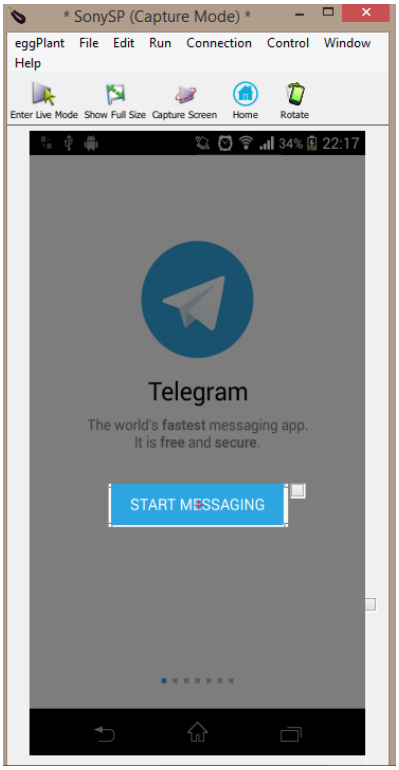
TestPlant mājaslapā ir pieejamas vairākas eggPlant rīka versijas un modifikācijas. Darba autors savam pētījumam ir izmantojis eggPlant Functional v16.01, kurš ir pieejams bezmaksas piecu dienu izmēģinājumam.

Runājot par dokumentāciju, rīkam ir pieejama diezgan plaša dokumentācija, izlasot kuru, nerodas jautājumu par rīka konfigurēšanu un izmantošanu.

5.2.4.1. Testu automatizācija

eggPlant rīkā testu automatizāciju var veikt vairākos veidots: manuāla skriptu izveide, automātiska skriptu ģenerēšana un tabulveida skriptu izveide.

Pirms uzsākt manuālo testu automatizāciju, testētājam ir jāizveido visu testā izmantojamo elementu attēlus. Lai to izdarītu eggPlant rīks dublē mobilās ierīces ekrānu datorā, līdz ar to, testētājam, ieslēdzot tveršanas režīmu, testētājs var viegli izveidot meklējamo elementu attēlus. Attēlu tveršanas process notiek ir redzams attēla 5.16.



5.16. att. **eggPlant rīka attēla tveršanas piemērs**

Testētājs izdala nepieciešamo objektu un nospiežot pogu “Capture image”, saglabājot to PNG formātā.

Kad nepieciešamie attēli ir izveidoti, notiek pati skriptu rakstīšana – testētājs norāda komandu, kuru ir jāizpilda rīkam un pievieno elementa bildi, kam šī komanda ir domāta. Autora izveidota skripta piemēru var redzēt attēlā 5.17.

```
Click "image0029" //nospiest "Menu" pogu
Click "Contacts" //atvert kontaktu sarakstu
WaitFor 6.0, "image0030_new" // gaidīt kamēr ieladēsies kontaktu skats
Click "image0030_new" // izvēlēties kontaktu no saraksta
WaitFor 8.0, "image00331_new" //gaidīt, kamēr atversies ziņapmaiņas skats
Click "Message" //izvēlēties ievadlauku
TypeText "Test" //uzrakstit tekstu "Test" ievadlaukā
Click "image0031" //nospiest "Send" pogu
```

5.17. att. **eggPlant skripta piemērs**

Skriptā ir aprakstītas sekojošas darbības: nospiest pogu kas redzama attēla "image0029.png", atvērtā logā nospiest pogu "Contacts" un uzgaidīt kamēr ielādēsies kontaktu skats. Tad nospiest uz elementu, kas ir redzams attēlā "image0030_new.png" – testa kontakts, uzgaidīt kamēr paradīsies zīņapmaiņas skats, kas ir redzams attēlā "image00331_new.png", uzspiest uz ievadlauku, kas redzams "Message.png" un ievadīts tekstu "Test". Kad teksts ir ievadīts, nospiest "Send" pogu.

Var redzēt, ka manuāls skriptu rakstīšanas process ir samērā viegls, jo rīkā izmantojama skriptēšanas valoda SenseTalk ir līdzīga angļu valodai. Pārējo testa gadījumu definējumu var redzēt pieliku 4.

eggPlant Funcional rīkā ir pieejams "Turbo Capture" režīms, kad testu automatizācija notiek automātiski. Izvēloties šo funkciju, testētājam ir jāatkārto visas darbības, kuram jābūt automatizētam. Darbības ir jāveic mobilās ierīcēs datorā dublējamā ekrānā, kas redzams attēlā 5.16. Kad testētājs ir veicis visas darbības un pārtraucis "Turbo Capture" režīmu, rīks automātiski izveido nospiesto attēlu bildes un automātiski uzģenerē visas testētāja veiktas darbības lietotnē. Šī iespēja ir ļoti noderīga testētājiem, bez programmēšanas prasmēm, tomēr automātiski izveidots skripts vai uzņemti attēli ir neprecīzi un testētājam ir jāpieliek roka, lai to modificētu.

Trešā testu automatizācijas iespēja ir tabulveida testu izveido. Testētājs izveido jaunu tabulu, kurā tiek izvēlēta darbība, kura jāveic rīkam, kā arī tiek izvēlēt attēls, kam šī darbība ir piemērota. Autora izveidotais piemērs ir redzams attēlā 5.18.

Step	Action	Arguments	Expected Value	Actual Value	Pass / Fail	Comment
1	Click	STARTMESSAGING.png	N/A	N/A		
2	TypeText	1	N/A	N/A		
3	Click	"image0024"	N/A	N/A		
4	WaitFor	8, "TelegramInvalidphonenumberOK"	N/A	N/A		
5	Click	"OK"	N/A	N/A		
6	Click	"delete"	N/A	N/A		
*						

5.18. att. **eggPlant tabulveida skriptu izveda**

Kad ir izveidoti atsevišķi testu skripti, var veidot testu kopas, kas sastāv no vairākiem skriptiem, kur pēc testu kopas izpildes var redzēt katra atsevišķa testa izpildes rezultātu.

Run	Script	Co...	Status
☑	Incorrect Number.script	--	SUCCESS
☑	WrongNumber.script	--	SUCCESS
☑	LogIn.script	--	SUCCESS
☑	WriteMessage.script	--	SUCCESS
☑	LogOut.script	--	FAILURE

5.19. att. **eggPlant testu kopas piemērs**

Kā arī, pēc katra automatizēto testu izpildes var redzēt katra atsevišķa testa detalizētus rezultātus, kā tas attēlots attēlā 5.20.

Run Date	Duration	Succ...	Errors	Warnings	Exceptions
04.05.16 01:48:49	0:00:18	1			
04.05.16 01:43:46					
04.05.16 01:33:34					
04.05.16 01:29:03	0:02:07	1			
04.05.16 00:54:31	0:00:12	1			
04.05.16 00:52:53					
04.05.16 00:48:43	0:00:06	1			
04.05.16 00:47:14	0:00:07	1			
04.05.16 00:46:34					

Statistics - LogOut

First Run	04.05.16 00:25:20	
Modified	04.05.16 00:48:43	
Runs	Success	Failure
8	5	3
	62.5%	37.5%
Avg. Time	0:00:34	

Reset Stats

5.20. att. **Testa izpildes rezultātu piemērs**

Testu rezultātos var redzēt informāciju par iepriekšējam testa izpildes reizēm, kā arī testa izpildes statistiku, kas noteikti palīdzēs sekot kļūdu labošanas dinamikai. Par katru saglabāto testu ir iespējams arī saņemt žurnālfailus, kas palīdzēs mobilo lietotņu testētājiem un programmētājiem ātrāk atrast kļūdas rašanas iemeslu, un ari ekrānuzņēmumus.

Runājot par testu izpildes ātrdarbību, testu kopas izpildes laiks normālos apstākļos, bez jebkādiem traucējumiem ir 58 sekunde.

5.2.4.2. Grūtības un problēmas rīka izmantošanā

Īpašu problēmu un grūtība rīka izmantošanā autors nav konstatējis. Problēma vairāk ir saistīta ar elementu atpazīšanas pieeju, kura tiek pielietota rīka – veicot testu automatizāciju, testētājam ir jāveic dubulta darbs: automatizēt testus gan portretorientācijas, gan ainavorientāciju. Skripts gandrīz netiek mainīts, tiek mainīti attēli, pēc kuriem notiek elementu atpazīšana. Kā arī rodas grūtības, ja uz lietotnē ir izveidoti vienādi attēlojami lietotāja saskarnes elementi.

Par rīka trūku var minēt to, ka rīks, tāpat ka Robotium rīks, eggPlant nespēj atbloķēt mobilās ierīces ekrānu, kā jau minēju, šī problēma ir viegli atrisināma ar Android ierīču iebūvēto "Palikt nomodā" funkciju.

5.2.4.3. Secinājumi par rīku

Secinājumā par eggPlant rīku var teikt, ka rīks ir intuitīvi saprotams un viegli izmantojams. Pateicoties tam, ka rīkā tiek pielietota attēlu elementu atpazīšanas pieeja, testu automatizāciju var veikt cilvēks bez programmēšanas prasmēm.

Testu automatizācijas process ir ļoti vienkāršs pateicoties rīka papildus iespējām: "Turbo Capture Session" un tabulveida testu izveide. Testu atskaites ir diezgan informatīvas, kur testētājs var konkrēti redzēt kurā brīdī ir notikusi kļūda.

Runājot par stabilitāti, rīka izmantošanas laikā nerādās nekādi traucējumi, kas varētu ietekmēt testu automātisko veikšanu. Kā arī testi tiek samērā ātri izpildīti.

Autors ir konstatējis, ka neskatoties uz rīka nelieliem trūkumiem, eggPlant rīks ir labs risinājums, testu automatizēšanas rīka izvēlē.

5.2.5. Kopsavilkums par testu automatizācijas rīkiem

Lai analīzes un salīdzinājuma rezultāti būtu labāk pārskatāmāki, autors ir izveidojis tabulu, kurā tika apkopota pamata informācija par rīkiem pēc iepriekšdefinētiem kritērijiem.

Tabula 5.2.

Testēšanas rīku salīdzinājuma rezultāti

	Appium	**Calabash**	**Robotium**	**eggPlant**
OS atbalsts	Android , iOS, Firefox OS	Android, iOS	Android	Android, iOS, Blackberry, WP
Lietotnes veidu atbalsts	Konkrētas platformas, tīmekļa, hibrīda	Konkrētas platformas	Konkrētas platformas, hibrīda	Konkrētas platformas, tīmekļa, hibrīda
Elementu atpazīšanas pieeja	Objektu atpazīšana, tekstu atpazīšana	Objektu atpazīšana, tekstu atpazīšana, koordinātu atpazīšana	Objektu atpazīšana, tekstu atpazīšana, koordinātu atpazīšana	Attēlu atpazīšana, tekstu atpazīšana
Ierīces atbloķēšanas no gaidstāves	Jā	Jā	Nē	Nē
Programmēšanas valodas atbalsts	Ruby, C#, Java, JS, Objective C, PHP, Python, Perl, Clojure	Gherkin, Ruby	Java	SenseTalk
Testu veikšana uz reālām ierīcēm	Jā	Jā	Jā	Jā
Atskaišu ģenerēšana	Atskaitēs tiek dota minimāla informācija par testu rezultātiem	Pietiekami detalizētas atskaites, kurās var redzēt kurā solī ir notikusi kļūda, ka arī kļūdas gadījuma tiek uzņemts ekrānuzņēmums	Atskaitēs tiek dota minimāla informācija par testu rezultātiem	Detalizētas testu rezultātu atskaites ar ekrānuzņēmumiem

Vairāku ierīču testēšana vienlaikus	Nē	Jā	Jā	Jā
Testu kopas izpildes laiks	119 sekundes (1 min 59 sek)	113 sekundes (1 min 53 sek)	78 sekundes (1 min 18 sek)	58 sekunde (0 min 58 sek)
Rīka cena	Bezmaksas – atvērta pirmkoda rīks	Bezmaksas – atvērta pirmkoda rīks	Bezmaksas – atvērta pirmkoda rīks. Recorder - 295 usd/ gadā	Komandas licence – 9650 eiro/gadā, Viena lietotāja licence – 5790 eiro/gadā
Dokumentācija	Dokumentācija nav pārāk informatīva, ir vairākas neskaidrības, kuras varētu aprakstīt detalizētāk	Dokumentācija nav pārāk informatīva, ir vairākas neskaidrības, kuras varētu aprakstīt detalizētāk	Informatīva dokumentācija ar detalizētu konfigurācijas soļu aprakstu	Pieejama plaša dokumentācija

Pētot izvēlētos testu automatizācijas rīkus un veicot izvēlētas lietotnes testa automatizāciju, autors ir secinājis, no testēšanā izmantotajām lietotāja saskarnes elementu atpazīšanas pieejam, vislabākais tomēr ir objektu atpazīšanas pieeja:

- Pirmkārt, testētājam nav jāveic dubulta darbs, un testēšanu var veikt gan portretorientācijā, gan ainavorientācijā;
- Otrkārt, šī pieeja noliedz iespēju vienādu elementu atrašanai, kā tas var notikt izmantojot teksta vai attēlu atpazīšanas pieeju, jo lietotāja saskarnes elementi tiek meklēti pēc universāla indeksa, klases vai ID.

Ka labākus rīkus no izvēlētiem četriem, autors ir izvēlējies Robotium un eggPlant rīkus. Ar šiem rīkiem radās vismazāk problēmu, abi rīki ir labi dokumentēti, kā arī testu izpilde notiek samēra ātri. Tomēr, ka iepriekš autors ir minējis, priekšroka ir rīkiem ar objektu atpazīšanas pieeju, kas ir tiek pielietota tieši Robotium rīka. Par noteicošo kritēriju kļuva riku cenas, var redzēt, ka eggPlant ir pietiekami dārgs rīks – 5790 vai 9650 eiro/gadā, līdz ar to, priekšroka ir bezmaksas alternatīvam, jo pat pērkot Robotium rīka papildinājumu Recorder, tas izmaksātu daudz mazāk.

Balstoties uz veikto salīdzinājumu, autors ir konstatējis, ka labākais rīks konkrētas platformas un hibrīda Android mobilo lietotņu testēšana ir Robotium rīks.

Ja autoram rastos nepieciešamība izvēlēties rīku tīmekļa lietotņu testēšanā, neskatoties uz autora konstatētam rīka problēmām, no četriem izvēlētiem rīkiem, autors izvelētos Appium rīku, jo,

- pirmkārt, tas ļauj testēt tīmekļa lietotnes;
- otrkārt, tas ir bezmaksas rīks,
- treškārt, testēšanā var izmantot Selenium WebDriver bibliotēkas, kuru sākuma uzdevums bija tieši tīmekļa automatizēta testēšana.

Līdz ar to, ka labāku variantu hibrīda un konkrētas platformas Android lietotņu rīku autors piedāvā Robotium rīku, un ka tīmekļa lietotņu testēšanas rīku – Appium rīku.

5.3. Baterijas testēšanas rīki

Vairākās jaunākajās mobilajās ierīcēs Android platformā jau iepriekš ir ieprogrammēts rīks, kas norāda, cik baterijas patērē katra lietotne. Šīs rīks ir ļoti noderīgs, un pārējo mobilo platformu izstrādātajiem būtu vērts izveidot līdzīgo rīku, lai nav jāmeklē un jāuzstāda papildus baterijas pārbaudes rīkus. Izmantojot automatizācijas rīkus, par kurām minēju sadaļā 5.2. baterijas enerģijas patēriņa izsekošanu var automatizēt

Ja Android lietotņu baterijas patēriņa testēšanai ir nepieciešams kāds nopietnāks rīks, viens no variantiem vārētu būt JouleUnit. Tas ir atklāta pirmkoda rīks, kura izmantošanai ir nepieciešams Eclipse vide. Šīs rīks meklē enerģijas lieko tērēšanu lietotnes darbošanās laikā, ka arī, mēra CPU, WiFi vai ekrāna enerģijas patēriņu. Tajā var redzēt enerģijas patēriņa izmaiņas dažādos testēšanas laika posmos. Kā trūkumu var minēt, ka šīs rīks ir paredzēts programmētājiem, vai baltas kastes testētājiem, kad ir pieejams mobilās lietotnēs pirmkods, līdz ar to, nav iespējams veikt baterijas enerģijas patēriņa mērīšanu melnas kastes testēšanā.

Viens no iespējamiem baterijas testēšanas rīka izvēles risinājumiem, kuru piedāvā darba autors, ir Monsoon PowerMonitor [45] ierīce, kura ir paredzēta tieši mobilo ierīču baterijas testēšanai un ļauj mērīt baterija patēriņu uz jebkāda veida litija baterijām. Vienīgas grūtības sagādā ierīces ar iebūvēto bateriju.

Ierīce ar vadu palīdzību tiek pievienots pie ierīces baterijas, tāpēc ierīce saņem enerģiju pa tiešo caur PowerMonitor ierīci. Pateicoties tam, baterijas patēriņa mērījumi ir precīzāki, nekā mobilo ierīču lietotņu rādītāji. Pie datora ierīce tiek pievienota caur USB portu. Datorā tiek izmantota PowerTool lietotne, kurā tiek attēloti visi PowerMonitor ierīces radītājus. Šo ierīces

izmantošanu var kombinēt ar testu automatizācijas rīku izmantošanu, simulējot testējamas ierīces reālo lietošanas gadījumu, tādā veidā, var izsekot baterijas panteriņu reālos lietošanas apstākļos.

Automatizācijas faila satura piemērs: start PowerToolCmd.exe /savefile=battery_test_name.pt4 /trigger=ATD1800A /vout=3.8 /keeppower /noexitwait, kur mēs norādām rezultātu saglabāšanas faila nosaukumu, testēšanas procesa ilgumu un mobilās ierīces baterijas voltāžu.

Vienīgais šī rīka trūkums, ir tā cena – 771 USD, tomēr ja kompānija tiešām vēlas veikt kvalitatīvu baterijas testēšanu, šī ierīce noteikti noderēs.

5.4. Stresa un pārtraukumu testēšanas rīki

Stresa un pārtraukumu testēšanā var izmantot dažādus automatizācijas rīkus, kas spēj ģenerēt nejaušas ierīces lietošanas gadījumus. Darba autors šiem mērķiem iesaka izmantot Monkey rīku, kas ir paredzēts Android ierīcēm, un ir iekļauts Android SDK, līdz ar to, nav nepieciešams lejuplādēt un instalēt papildus rīkus un bibliotēkas.

Rīks ģenerē nejaušus ierīces izmantošanas gadījumus, piemēram, ka uzklikšķināšana uz ekrāna, skāņas ieslēgšana un izslēgšana, tīkla pieslēguma ieslēgšana vai izslēgšana, ierīces rotāciju un citas lietotnes izmantošanas situācijas, kas varētu rasties ierīces izmantošanā. Pateicoties tam, ka darbības tiek ģenerētas pietiekami ātri, tas noslogo sistēmu, radot stresa situāciju.

Rīks tiek izmantots ar komandrindas palīdzību, lai to palaistu ir nepieciešams norādīt ceļu līdz mūsu lietotnei, un jānorāda, cik daudz nejaušu komandu jāuzģenerē.

Autora izveidotais rīka izsaukuma piemērs Telegram lietotnei: *adb shell monkey -p org.telegram.messenger -v 1500.*

Skripta palaišanas rezultātā, tiek uzģenerētas patvaļīgas 1500 darbības, kuras tiek aprakstītas komandrindā.

Testa laikā, testētājs var novērot, kā uzvedas lietotne stresa un patraukuma rašanas situācija: vai nav notikusi informācijas pazušana, vai lietotne turpina strādāt, nenobrūk - un no tā var secināt par dotas lietotnes stabilitāti.

SECINĀJUMI

Darba gaitā autors deva priekšstatu par mobilo lietotņu veidiem, izvirzīja katra veida priekšrocības un trūkumus. Balstoties uz informācijas avotiem un autora mobilo lietotņu testēšanas pieredzi, tika definētas mobilo lietotņu būtiskākie testēšanas paveidi un to specifika mobilo lietotņu testēšanā. Tika veikts pētījums par mobilo lietotņu testu automatizācijas iespējam, automatizācijas priekšrocībām un trūkumiem.

Pēc veiktā pētījuma autors ir secinājis, ka automatizēta testēšana ir vairāk piemērota lieliem projektiem ar pietiekami lielu funkcionalitāti un atbalsta laiku, tāpēc ka pats testu automatizācijas process ir samēra laikietilpīgs, jo katra testa piemēra automatizāciju ir rūpīgi jāplāno, kā arī katra automatizēto testpiemēra soļa korekto izpildi ir jāpārbauda lietotnē.

Salīdzinot Android mobilo lietotņu testu automatizācijas iespējas un alternatīvas, autors ir konstatējis, ka labākais, no izvēlētiem konkrētas platformas un hibrīda Android mobilo lietotņu testēšanas rīkiem ir Robotium rīks, kurš neskatoties uz nelieliem trūkumiem, salīdzinājumā ar pārējiem izvēlētiem rīkiem, ir samērā ātrs, pieejams bezmaksas lietošanā, labi dokumentēts un viegli apgūstams.

Tika konstatēts, ka no rīku piedāvātām lietotāja saskarnes elementu atpazīšanas pieejam, labāka pieeja ir objektu atpazīšanas pieeja, kas ir neatkarīga no izmaiņām dizainā, un, atšķirībā no pārējam pieejam, nodrošina lietotāja saskarnes objektu korektu atpazīšanu.

Veicot testu automatizāciju, tika secināts, ka katra testa piemērā automatizēšanai ir jāvelta pietiekams laiks plānošanai, uzreiz ķeroties pie testu automatizācijas, var tikt izlaisti būtiskie momenti, kuri manuālā testēšanās laikā netiek izskatīti.

Automatizēta testēšana ir ļoti noderīga mobilo lietotņu baterija testēšanā, ka arī stresa un pārtraukumu testēšanā, kur bez automatizācijas rīku palīdzības ir samērā grūti noslogot sistēmu, lai tā strādātu robežslodzē.

Kopumā, var droši teikt, ka mobilo lietotņu automatizēta testēšana nevar aizvietot manuālo testēšanu, tomēr mobilo lietotņu testēšanā ir vairāki momenti, kur bez automatizācijas rīkiem ir pietiekami grūti iztikt.

NOBEIGUMS

Veicot darbu, autors ir apskatījis Android mobilo lietotņu testu automatizācijas iespējas un alternatīvas, kā arī ieguva priekšstatu par automatizēto testu kopumā. Tāpat papildināja savas zināšanas par mobilo lietotņu, gan manuālo, gan automatizēto, testēšanu kopumā.

Darbā tika apskatīti reāli Android lietotņu testu automatizācijas rīki, un tika veikta iepriekšdefinētas testa kopas automatizācija uz konkrētas lietotnes piemēra, kas autoram kļuva par labu sākumpunktu, ne tikai Android, bet arī citu mobilo operētājsistēmu lietotņu testu automatizācijas apgūšanai.

Pētījuma procesā autors iepazinās ar vairākām testu automatizācijas rīkiem un paņēmieniem, kas noteikti noderēs autora turpmākajā darbā par testētāju. Darbā veikto pētījumu noteikti varēs vēlāk izmantot, kā atbalstu Android lietotņu testu automatizācijas rīka izvēlē.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Pāvels Lazučonoks, *Mobilo Lietotņu testēšana*, Kursa darbs, 2015.
2. Number of apps available in leading app stores as of July 2015 [tiešsaiste]. [atsauce 24.03.2016.]. Pieejams: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores
3. What's a mobile app? [tiešsaiste]. [atsauce 03.04.2016.]. Pieejams: http://www.contentious.com/2011/03/02/whats-a-mobile-app/
4. A. Singh, "Top 6 Programming Languages for Mobile App Development" [tiešsaiste]. [atsauce 03.04.2016.]. Pieejams: https://www.linkedin.com/pulse/top-6-programming-languages-mobile-app-development-akshatha-singh?articleId=7603736045158150836
5. What is the Mobile Web [tiešsaiste]. [atsauce 03.04.2016.]. Pieejams: https://www.techopedia.com/definition/23588/mobile-web
6. Akadēmiskā terminu datubāze AkadTerm. [tiešsaiste]. [atsauce 06.04.2016.]. Pieejams: http://termini.lza.lv/
7. Keynote, "The State of Mobile Software Quality Full Report," 2014, 7 lpp, [tiešsaiste]. [atsauce 06.05.2016.]. Pieejams: http://www.keynote.com/docs/reports/Keynote_MSQ_Survey_2014_Full_Report.pdf
8. TestObject, "Mobile App Testing. Main challenges, different approaches, one solution," 2015, 12 lpp, [tiešsaiste]. [atsauce 12.04.2016.]. Pieejams: https://testobject.com/wp-content/uploads/ebook_mobile_app_testing.pdf
9. Usability 101: Introduction to Usability. [tiešsaiste]. [atsauce 20.05.2016.]. Pieejams: https://www.nngroup.com/articles/usability-101-introduction-to-usability/
10. Usability – ISO 9241 definition. [tiešsaiste]. [atsauce 20.05.2016.]. Pieejams: https://www.w3.org/2002/Talks/0104-usabilityprocess/slide3-0.html
11. F. Nayebi, J.M. Desharnais, A. Abran, "The State of the Art of Mobile Application Usability Evaluation," *25th IEEE Canadian Conference on Electrical and Computer Engineering*, 2012, doi: 10.1109/CCECE.2012.6334930, 3 lpp.
12. uTest, 'The Essential Guide to Mobile App Testing" [tiešsaiste]. [skatīts 14.04.2016.]. Pieejams: http://core.ecu.edu/STRG/materials/uTest_eBook_Mobile_Testing.pdf
13. Terminu vārdnīca (Moodle rīks). [tiešsaiste]. [atsauce 17.04.2016.]. Pieejams: http://estudijas.lu.lv/mod/glossary/view.php?id=47882
14. G.J. Myers, T. Badgett, C. Sandler, *The Art of Software Testing Third Edition*, JohnWiley & Sons, 2012, 124 lpp.

15. Programmatūras testēšanas rokasgrāmata vadītājiem Versija 1.0 [tiešsaiste]. [atsauce 21.04.2016.]. Pieejams: http://squalio.com/assets/upload/userfiles/files/SQ_Testesanas_Rokasgramata_Master.pdf
16. S. Butt, "Benefits of Automation Testing" [tiešsaiste]. [atsauce 24.05.2016.]. Pieejams: http://red-badger.com/blog/2013/02/01/benefits-of-automated-testing/
17. Mark Chatham, *Selenium By Example – Volume 3: Selenium WebDriver*, Lulu.com, 2014, 88-89 lpp.
18. Daniel Knott, *Hands-On Mobile App Testing*, Leanpub, 2015, 179 lpp.
19. G. Mackeown, "Not yet convinced by image based test automation? Here are six arguments that may change your mind…" [tiešsaiste]. [atsauce 28.04.2016.]. Pieejams: http://www.testplant.com/2015/03/20/not-yet-convinced-by-image-based-test-automation-here-are-six-arguments-that-may-change-your-mind/
20. eggPlant Tools For Software Application Testing by TestPlant [tiešsaiste]. [atsauce 06.05.2016.]. Pieejams: http://www.testplant.com/eggplant/testing-tools/
21. Sikuli Script [tiešsaiste]. [atsauce 28.04.2016.]. Pieejams: http://www.sikuli.org/
22. Test Automation for GUI Testing | Ranorex [tiešsaiste]. [atsauce 28.04.2016.]. Pieejams: http://www.ranorex.com/
23. An introduction to object recognition testing [tiešsaiste]. [atsauce 01.05.2016.]. Pieejams: https://smartbear.com/learn/automated-testing/intro-to-object-recognition/
24. Appium: Mobile App Automation Made Awesome [tiešsaiste]. [atsauce 30.04.2016.]. Pieejams: http://appium.io/
25. Calaba.sh - Automated Acceptance Testing for iOS and Android Apps [tiešsaiste]. [atsauce 30.04.2016.]. Pieejams: http://calaba.sh/
26. Selendroid: Selenium for Android [tiešsaiste]. [atsauce 30.04.2016.]. Pieejams: http://selendroid.io/
27. Espresso [tiešsaiste]. [atsauce 30.04.2016.]. Pieejams: http://google.github.io/android-testing-support-library/docs/espresso/
28. Robotium [tiešsaiste]. [atsauce 30.04.2016.]. Pieejams: http://www.robotium.org
29. S. Wagner, "Improved Mobile App Testing with Text Recognition And Matching" [tiešsaiste]. [atsauce 01.05.2016.]. Pieejams: https://testobject.com/blog/2013/11/text-recognition-and-matching-for-powerful-app-testing.html
30. Testing with Optical Character Recognition [tiešsaiste]. [atsauce 01.05.2016.]. Pieejams: http://eng.wealthfront.com/2015/07/02/testing-with-optical-character-recognition-ocr/
31. What Are Advantages and Weaknesses of a Capture/Playback tools [tiešsaiste].

[atsauce 03.05.2016.]. Pieejams: http://qatestlab.com/knowledge-center/QA-Testing-Materials/what-are-advantages-and-weaknesses-of-a-capture-playback-tool/

32. Robotium Recorder – Robotium Tech [tiešsaiste]. [atsauce 12.05.2016.]. Pieejams: http://robotium.com/products/robotium-recorder
33. RERAN - Record and Replay for Android [tiešsaiste]. [atsauce 03.05.2016.]. Pieejams: http://www.androidreran.com/
34. Telegram Messenger [tiešsaiste]. [atsauce 05.05.2016.]. Pieejams: https://telegram.org/
35. 12 Best Mobile App Testing Frameworks [tiešsaiste]. [atsauce 05.05.2016.]. Pieejams: https://iprodev.com/12-best-mobile-app-testing-frameworks/
36. D. Agarwal, "Top 11 Mobile Automation Tools" [tiešsaiste]. [atsauce 05.05.2016.]. Pieejams: https://testingmobileapps.wordpress.com/2016/02/06/top-11-mobile-automation-tools/
37. Whats is the best mobile UI Automation tool for testing apps on real devices for iOS And Android & iOS [tiešsaiste]. [atsauce 05.05.2016.]. Pieejams: https://www.quora.com/What-is-the-best-mobile-UI-Automation-tool-for-testing-apps-on-real-devices-for-iOS-and-Android-iOS
38. Appium: A Cross-Browser Mobile Automation Tool [tiešsaiste]. [atsauce 23.05.2016.]. Pieejams: https://www.3pillarglobal.com/insights/appium-a-cross-browser-mobile-automation-tool
39. Testing Support Library [tiešsaiste]. [atsauce 07.05.2016.]. Pieejams: http://developer.android.com/intl/ru/tools/testing-support-library/index.html
40. J. Maturana Larsen, "An Overview of Calabash Android" [tiešsaiste]. [atsauce 15.05.2016.]. Pieejams: http://blog.lesspainful.com/2012/03/07/Calabash-Android/
41. Atom: A hackable text editor for the 21st Century. [tiešsaiste]. [atsauce 15.05.2016.]. Pieejams: https://atom.io/
42. Automated Functional testing for Android based on cucumber [tiešsaiste]. [atsauce 16.05.2016.].Pieejams: https://github.com/calabash/calabashandroid/tree/master/ruby-gem/lib/calabash-android/steps
43. M. Poschenrieder, "A Beginner's Guide to Automated Mobile App Testing" [tiešsaiste]. [atsauce 15.05.2016.]. Pieejams: https://blog.testmunk.com/tutorial-for-automated-mobile-app-testing-calabash/
44. Hrushikesh Zadgaonkar, *Robotium Automated Testing for Android,* Packt Publishing, 2013, 5 lpp.
45. Power Monitor [tiešsaiste]. [atsauce 24.05.2016.]. Pieejams: https://www.msoon.com/LabEquipment/PowerMonitor/

PIELIKUMI

1. pielikums

Appium rīkam autora izveidotais kods

```
public class Telegram {

	AndroidDriver<WebElement> driver;

	//Pieslegsanas konfiguracijas
	@Test
	public void AndroidConfig() throws MalformedURLException {

		DesiredCapabilities capability = new DesiredCapabilities();
		capability.setCapability("deviceName", "C5303");
		capability.setCapability("platformName", "Android");
		capability.setCapability("deviceVersion", "4.3");

		File file = new
File("C:\\Users\\Pavel\\workspace\\AutoTest\\apks\\telegram3-7-0.apk");
		capability.setCapability("app", file.getAbsolutePath());

		driver = new AndroidDriver(new
URL("http://192.168.47.1:4722/wd/hub"), capability);
		}

	@Test
	public void IncorrectNumber() throws InterruptedException{


		WebDriverWait wait = new WebDriverWait(driver, 50);

	wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("org.tel
egram.messenger:id/start_messaging_button")));

	driver.findElement(By.id("org.telegram.messenger:id/start_messaging_but
ton")).click();

	driver.findElement(By.xpath("//android.widget.EditText[@index='2']")).s
endKeys("1");

	driver.findElement(By.className("android.widget.ImageView")).click();

	wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("android
:id/message")));
		driver.findElement(By.id("android:id/button1")).click();

	driver.findElement(By.xpath("//android.widget.EditText[@index='2']")).c
lear();

	}


	@Test
	public void WrongNumber() throws InterruptedException{


		WebDriverWait wait = new WebDriverWait(driver, 50);

	driver.findElement(By.xpath("//android.widget.EditText[@index='2']")).s
endKeys("11111111");

	driver.findElement(By.className("android.widget.ImageView")).click();
```

```
      wait.until(ExpectedConditions.visibilityOfElementLocated(By.name("Phone
verification")));

      driver.findElement(By.xpath("//android.widget.TextView[contains(@text,
'Wrong number?')]")).click();

      driver.findElement(By.xpath("//android.widget.EditText[@index='2']")).c
lear();

      }

//Testa piemers veiksmigai sistemas pieteikshanai
@Test
public void logIn() throws InterruptedException{


                  WebDriverWait wait = new WebDriverWait(driver, 50);

      driver.findElement(By.xpath("//android.widget.EditText[@index='2']")).s
endKeys("29667817");

      driver.findElement(By.className("android.widget.ImageView")).click();

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//an
droid.widget.TextView[contains(@text, 'Telegram')]")));
            }

@Test
public void WriteMessage() throws InterruptedException{

                  WebDriverWait wait = new WebDriverWait(driver, 50);

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//an
droid.widget.TextView[contains(@text, 'Telegram')]")));

      driver.findElement(By.className("android.widget.ImageView")).click();
      //Nospiest "Menu" pogu
                  driver.findElement(By.name("Contacts")).click(); //Nospiest
"Konktakti" pogu

                  //gaidit kamer atversies kontaktu skats

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//an
droid.widget.TextView[contains(@text, 'Contacts')]")));

      driver.findElement(By.xpath("//android.widget.FrameLayout[@index='2']")
).click(); //izveleties "Test" kontu

                  //ierakstit ievadlaukaa tekstu "Test"

      driver.findElement(By.xpath("//android.widget.EditText[contains(@text,
'Message')]")).sendKeys("Test");

                  //Nospiest "Send" pogu

      driver.findElement(By.xpath("//android.widget.FrameLayout/android.widge
t.LinearLayout[1]/android.widget.FrameLayout[1]/"
                              +
"android.widget.FrameLayout[1]/android.widget.FrameLayout[1]/"
                              +
"android.widget.LinearLayout[1]/android.widget.FrameLayout[2]/"
```

```
                                    +
"android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/"
                                    +
"android.widget.FrameLayout[@index='1']/android.widget.ImageView")).click();


      driver.findElement(By.className("android.widget.ImageView")).click();
//atgriezties uz sakuma skatu
            }


            @Test
            public void logOut(){

                  WebDriverWait wait = new WebDriverWait(driver, 50);

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//an
droid.widget.TextView[contains(@text, 'Telegram')]")));

      driver.findElement(By.className("android.widget.ImageView")).click();

      wait.until(ExpectedConditions.presenceOfElementLocated(By.className("or
g.telegram.messenger.support.widget.RecyclerView")));

      driver.findElement(By.xpath("//android.widget.TextView[contains(@text,
'Settings')]")).click();

      driver.findElement(By.xpath("//android.widget.LinearLayout[@index='1']/
android.widget.FrameLayout/android.widget.ImageView")).click();
                  driver.findElement(By.name("Log out")).click();

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("android
:id/contentPanel")));
                  driver.findElement(By.id("android:id/button1")).click();

      wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("org.tel
egram.messenger:id/start_messaging_button")));

            }
}
```

```
Scenario: When I enter incorrect phone number an error appears
  When I see text "START MESSAGING"
  Then I press view with id "start_messaging_button"
  Then I enter "1" phone number into field 1
  Then I press imageview number 1
  Then I see the text "Invalid phone number"
  Then I press view with id "button1"
  Then I clear input field 1

  Scenario: I can change my phone number if was entered wrong number
    When I see text "START MESSAGING"
    Then I press view with id "start_messaging_button"
    Then I enter "11111111" phone number into field 1
    Then I press imageview number 1
    Then I see the text "Phone verification"
    Then I press textview number 5
    Then I clear input field 1

    Scenario: I can successfully login using my phone number
      When I see text "START MESSAGING"
      Then I press view with id "start_messaging_button"
      Then I enter "29667817" phone number into field 1
      Then I press imageview number 1
      Then I see the text "Phone verification"
      Then I wait for 10 seconds
      Then I see the text "Telegram"
      Scenario: I can write message to another user
        When I see text "Telegram"
        Then I press imageview number 2
```

Then I press textview number 7

Then I press SimpleTextView number 1

Then I enter "Test" text into message input field number 1

Then I press imageview number 4

Then I press imageview number 2

Then I see the text "Telegram"

Scenario: I can logout

When I see text "Telegram"

Then I press imageview number 2

Then I press textview number 9

Then I press imageview number 1

Then I press textview number 2

Then I press view with id "button1"

Then I see the text "Telegram"

```
public class TestApk extends ActivityInstrumentationTestCase2 {

	private static final String LAUNCHER_ACTIVITY_FULL_CLASSNAME =
"org.telegram.ui.LaunchActivity";

	private static Class launcherActivityClass;
	static {
		try {
			launcherActivityClass = Class
			.forName(LAUNCHER_ACTIVITY_FULL_CLASSNAME);
		}
		catch (ClassNotFoundException e) {
			throw new RuntimeException(e);
		}
	}


	public TestApk() throws ClassNotFoundException {
		super(launcherActivityClass);
	}

	private Solo solo;

	@Override
	protected void setUp() throws Exception {
		solo = new Solo(getInstrumentation(), getActivity());
	}

	public void test_incorrectNumber() {

		solo.clickOnView(solo.getView("org.telegram.messenger:id/"
				+ "start_messaging_button"));
		solo.enterText(1, "1");
		solo.clickOnImage(0);
		solo.waitForText("Invalid phone number");
		solo.clickOnView(solo.getView("android:id/button1"));
		solo.clearEditText(1);
	}

	public void test_wrongNumber() {

		solo.clickOnView(solo.getView("org.telegram.messenger:id/"
				+ "start_messaging_button"));
		solo.enterText(1, "11111111");
		solo.clickOnImage(0);
		solo.waitForText("Phone verification");
		solo.clickOnText("Wrong number?");
		solo.waitForText("Your phone");
		solo.clearEditText(1);
	}



	public void test_logIn() {

		solo.clickOnView(solo.getView("org.telegram.messenger:id/"
				+ "start_messaging_button"));
```

```
            solo.enterText(1, "29667817");
            solo.clickOnImage(0);
            solo.waitForText("Phone verification");
            solo.sleep(10000);
            solo.waitForText("Telegram");
      }

      public void test_writeMessage() {

            solo.waitForActivity("LaunchActivity"); //gaidam kamer ieslegsies
lietotne
            solo.clickOnImage(1); //nospiezam "Menu" pogu
            solo.clickOnText("Contacts"); //izvelamies kontaktu sarakstu
            solo.clickInList(3); //izvelamies kontaktu, kuram reakstisim
iszinu
            solo.clickOnText("Message"); //nospiezam uz ievadlauku
            solo.enterText(0, "Test"); //ievadam ievadlauka tekstu "Test"
            solo.clickOnImage(3); //nospiezam "Send" pogu
            solo.clickOnImage(1); //izejam no zinapmajnas
      }

      public void test_logOut() {

            solo.waitForText("Telegram");
            solo.clickOnImage(1);
            solo.clickOnText("Settings");
            solo.clickOnImage(0);
            solo.clickOnText("Log out");
            solo.waitForText("Telegram");
            solo.clickOnButton(1);
      }

      public static TestSuite suite(){
            TestSuite t = new TestSuite();
            t.addTest(TestSuite.createTest(TestApk.class,
"test_incorrectNumber"));
            t.addTest(TestSuite.createTest(TestApk.class,
"test_wrongNumber"));
            t.addTest(TestSuite.createTest(TestApk.class, "test_logIn"));
            t.addTest(TestSuite.createTest(TestApk.class,
"test_writeMessage"));
            t.addTest(TestSuite.createTest(TestApk.class, "test_logOut"));
            return t;
      }

      @Override
      public void tearDown() throws Exception {
            solo.finishOpenedActivities();
      }
}
```

4. pielikums

eggPlant Functional rīkam autora

izveidotais kods

IncorrectNumber.script

```
Click "STARTMESSAGING.png" //nospiest pogu START MESSAGING

TypeText "1" //ierakstīt ciparu "1"

WaitFor 8.0, (Text:"1") //gaidīt, kamēr teksts tiek ierakstīts

Click "image0024" //nospiest Done pogu

WaitFor 8.0, "TelegramInvalidphonenumberOK" //sagaidam paziņojumu par kļūdu

Click "image0025" //nospiest OK pogu

TypeText Backspace //izdzēst iepriekš ierakstitu ciparu
```

WrongNumber.script

```
TypeText "11111111"

WaitFor 8.0, "1111.png"

Click "image0026"

Click "Wrongnumber_0001"

DoubleClick "image0027"

TypeText Backspace
```

LogIn.script

```
TypeText "29667817"

WaitFor 8.0, "37129667817"

Click "image0028"

WaitFor 20.0, "image0029_new"
```

WriteMessage.script

```
Click "image0029"  //Nospiest Menu pogu

Click "Contacts" //izvēlēties kontaktu skatu

WaitFor 6.0, "image0030_new" //gaidām kamēr ielādēsies kontakti
```

Click "image0030_new" //izvēlāmies testa kontaktu

WaitFor 8.0, "image00331_new" //sagaidām kamēr ielādēsies ziņapmaiņās skats

Click "Message" //izvēlamies ievadlauku

TypeText "Test" //ierakstam tekstu “Test”

Click "image0031" //nosūtam īsziņu

LogOut.script

Click "image0029" //nospiest Menu pogu

Click "Settings" //nospiest “Settings” pogu

Click "image0034" //nospiest “Menu” pogu

Click "Logout" //nospiest “Logout”

WaitFor 8.0, "TelegramAreyousureyouwanttolog" //gaidam brīdinājuma paziņojumu

Click "Ok" //nospiest “Ok” pogu

Bakalaura darbs „Android mobilo lietotņu testēšanas automatizācija” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____ Pāvels Lazučonoks

Rekomendēju/nerekomendēju darbu aizstāvēšanai
Vadītāja: Dr.dat. Zane Bičevska _____ __.05.2016.

Recenzents: Dr.dat. Vineta Arnicāne

Darbs iesniegts Datorikas fakultātē __.05.2016.
Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē
_____. prot. Nr. _____.

Komisijas sekretārs(-e): _____