TAMPEREEN
AMMATTIKORKEAKOULU

OHJELMISTOJEN TESTAUS

Eero Pruuki

Opinnäytetyö
Elokuu 2017
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

PRUUKI EERO:
Ohjelmistojen testaus

Opinnäytetyö 30 sivua
Elokuu 2017

Opinnäytetyössä tutustutaan tarkemmin ohjelmistojen testaukseen WEB-kehittäjän näkökulmasta. Syvennytään lähinnä eri työkaluihin ja testaustyyleihin. Testaus periaatteet ovat samat, oli kyseessä mobiili, sulautettu tai mikä tahansa alusta jolle sovelluksia tehdään.

Asiasanat: ohjelmistojentestaus, testauskulttuuri, automaatiotestaus, ohjelman laatu

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Information Technology Degree Program
Software Engineering

EERO PRUUKI:
Testing of software

Bachelor's thesis 30 pages
August 2017

In this thesis we explore in world of software testing in point of WEB-developer. Mostly go deeper difference tools and testing styles. Principals of testing are same, said mobile, embedded system or any platform where software are made.

Key words: testing software, testing culture, automation testing, software quality

SISÄLLYS

ERITYISSANASTO tai LYHENTEET JA TERMIT (valitse jompikumpi)

TAMK	Tampereen ammattikorkeakoulu
op	opintopiste
Heartbleed	Linuxin ytimestä löytynyt tietoturva ongelma
Meltown	Intel prosessorin tietoturva ongelma
spektre	Intel prosessorin tietoturva ongelma
Linux	Tietokoneen ydin
Ubuntu	Linuxin päälle rakennettu käyttöjärjestelmä
Java	Oraclen teknologia, ohjelmointi kieli
Netbeans	Ohjelmointi työkalu
tomcat	WEB-kehityksessä ohjelmisto jonka avulla Java toimii.
apache2	Palvelin ohjelmisto
Testikattavuus	Testien laajuus
sql-injektio	Yleinen haavoittuvuus, kun syötetään tietokantaan haitallista syötettä
XSS	Cross side scriptin, eli sivun sisällä suoritaan ulkopuolista ohjelma koodia.
versionhallinta	Paikka jossa säilytään sovelluksen eri versioita.
GIT	Avoimen lähdekoodin versionhallinta sovellus
data	Tietoa tietokannassa
Palvelinarkkitehtuuri	Palvelin rakenne
bugi	Ohjelmistossa oleva ongelma
Lähdekoodi	Ohjelman koodi
Avoin lähdekoodi	Lähdekoodi on avointa, eli kaikkien saatavilla.
JavaDoc	Javan ”käyttöohje” eli tapa dokumentoida koodia.
katselmointi	Ohjelman lähdekoodin tarkastelu ja lukeminen
metodi	aliohjelma
komponentti	Pala joka metodia laajempi. Yksi kokonaisuus
Web	Internet
ide	Graafinen ohjelmointi työkalu
funktio	asia joka tekee jotain
debug	jäljentää virheitä
Python	Ohjelmointi kieli

Jython	Javaan perustuva kieli joka jäljentelee Pythonia
Geckodriver	Ajurit Gecko selain moottorille, esim FireFox ja siihen pohjautuville
SeleniumLibrary	Käytetään selaimen käytön simuloimiseen.
Jenkins	Automaatio palvelin
plugin	lisäosa
JavaScript	WEB-teknologia, ohjelmointi kieli.

1 Johdanto

Ohjelmistojen testaamisella haetaan tarkoituksella sovelluksesta virheitä. Ne pyritään löytämään mahdollisimman aikaisin, sillä bugien korjaamisen kultainen sääntö on. Mitä aikaisemmin ne löydetään, sitä halvempaa niiden korjaaminen on. Bugien metsästystä voidaan jatkaa loputtomiin, mutta yleensä määritellään taso jonka sovellus on toteutettava ennen julkaisua. Tuotanto sovellus kestää myös jonkin verran bugeja ja harvoin on sellainen tilanne että korjataan kaikkia vaikka ne olisi tiedossa. Silloin ne ovat sen verran pieniä tai harvoin ilmaantuvia että taloudellisesti korjaaminen ei ole järkevää. Pienet bugit voivat olla esimerkiksi, yksittäinen pikseli on hieman erivärinen kun pitäisi tai tavuviiva on pikselin verran liian pitkä. Vakavat bugit taas voivat kaataa ohjelman/järjestelmän tai vuotaa sovelluksen tai järjestelmän salaisuuksia (Heardbleed).

Yleisesti ottaen tietoturva ongelmat kuuluvat ohjelmisto testaukseen, tosin niiden varma testaaminen on erittäin vaikeaa koska ne voivat ilmaantua vain joillakin ohjelmisto versiolla tai jopa raudan bugista johtuen (Meltown ja spektre). Täydellistä testikattavuutta ei voida käytännössä saavuttaa, lisäksi vielä varma tietoturva testaus vaatii järjestelmän, sovelluksen ja työkalujen erittäin syvällisen tuntemuksen.

Opinnäytetyössä tutustutaan eri testausmenetelmiin ja työkaluihin. Opinnäytetyö on tehty web-koodaajan näkökulmasta, mutta periaatteet käyvät mihin tahansa ohjelmointiin. Samalla tutustutan myös muihin ohjelman laatua häiritseviin tekijöihin.

2 Ohjelmistojen testaus

2.1 Miksi

Ohjelmistojen testaamisella pyritään saamaan varmuus siitä että sovellus toimii oikein kaikissa tilanteissa. Käytännössä täydellistä testaamista ei saada aikaiseksi ja todellisuudessa viimeisen testauksen testaa aina loppukäyttäjä. Testaamisella tarkoitetaan ohjelman ”rasitusta” erilaisissa tilanteissa, katsotaan miten se käsittelee virheet, ottaen huomioon myös mahdolliset kaatumiset. Täydellistä testikattavuutta ei saada koskaan aikaiseksi, ainakaan isoissa ohjelmistoissa. Huom, testaamisella ei oteta millään tavalla huomioon koodin laatuun.

2.1.1 Käyttäjä testaus, eli ihminen testaajana

Kun testaajani toimivat ihmiset, testien laatu riippuu täysin testaajien taito/tietämys tasosta. Jos testaajat tuntevat järjestelmän hyvin ja tuntevat tietoturvan uhat hyvin päästään kyllä todella hyvään testikattavuuteen. Kun käyttäjät tietävät tietoturva trendit kuten XSS, sql – injektio tms. he voivat jopa toteuttaa em ja muita hyökkäyksiä testattavaa sovellusta vastaan. Samalla varmistuen siitä että sovellus kestää ihan perus hyökkäyksiä. Tämä testaus tyyli vie aikaa ja testauksia voi olla vaikea toistaa.

2.1.2 Testi automaatio, eli robotti testaus

Testausautomaatiolla tarkoitetaan sitä että testaus on automatisoitu, esim. näkyviin kenttiin syötetään koneellisesti syötteitä, samalla kone tarkistaa toimiko sovellus oikein. Hieman työläs ottaa käyttöön ja ketä tahansa ei voi ruveta tekemään automaatio testausta vaan vaatii aina ammattilaista. Vaatii myös työtä testi syötteiden tekoon ja niiden ajamisen automatisointiin. Automaatio testauksella ei voi korvata käyttäjän suorittamaa testausta, mutta on loistava lisä ohjelmistojen kehityksessä ja testauksen suorittamisessa.

2.2 Miten

2.2.1 Manuaalinen, käyttäjä testaa

Ihminen sovellustaajana on yleisin testausmuoto. Sellaista sovellusta ei ole julkaistu mitä ihminen ei jossain kohtaa olisi testannut! Mutta testauksen laatu riippuu täysin käyttäjän taidoista ja kokemuksista. Annan joitakin esimerkkejä:

Petri Peruskäyttäjä, ei ymmärrä tietokoneista/tuotteesta paljoa voi työskennellä vaikka myynnissä. Jos tällainen ihminen testaa tuotetta, voisi kuvitella lähes mikä tahansa pieni yritys. Petrin testaus rajoittuu lähinnä siihen että sovellus toimii oikein oikealla testi syötteellä, eli numero kenttään syötetään numeroita tms. eikä haeta sovelluksen rajoja ja mahdollisia bugeja sen tarkemmin. Jos Petri huomaa virheen, sen raportointi saattaa olla puutteellinen ja vaikea selvittää mistä johtunut tai miten saataisiin toistumaan.

Konsta Kokenutkäyttäjä, tietää miten tietokoneet toimii ja testattavasta tuotteesta syvempi ymmärrys, saattaa olla tuotteen kehittäjä tai suunnittelija. Osaa testata myös miten sovellus toimii virhetilanteissa. Eli hakee myös tietoisesti virhetilanteita ja yrittää hakea sovelluksen rajoja toisin kuin Petri. Myös Konsta osaa raportoida virheistä huomattavasti paremmin ja ne voivat olla helposti toistettavissa.

Harri Hakkeri, ei välttämättä tiedä sovelluksesta yhtään mitään. Mutta tietokeista sitäkin enemmän. Sovelluksen toiminta silloin kun sitä käytetään oikein, ei Harria kiinnosta. Pääsääntöisesti ns. hakkeritestauksella haetaan virheitä ja ”halutaan” että sovellus toimii väärin, eli vuotaa tietoja muista käyttäjistä tai siitä mitä sen tietokantoihin on tallennettu.

Luottaminen pelkästään käyttäjä testiin saadaan lähtökohtaisesti ihan hyviä testi tuloksia mutta testikattavuus jää hieman vajaaksi, eli kaikkia mahdollisia kombinaatiota ei ole saatu testattua. Esimerkkinä vaikka kirjautumissivu jossa on kaksi kenttää käyttäjätunnus ja salasana.

Taulukko 1. Kirjautumisen testaus

Käyttäjätunnus	Salasana	Onnistuiko kirjautuminen
oikea	oikea	Kirjautuminen onnistui
oikea	väärä	Kirjautuminen ei onnistunut
väärä	oikea	Kirjautuminen ei onnistunut
väärä	väärä	Kirjautuminen ei onnistunut

Taulukon 1 mukainen testaus saavuttaisi täyden testi kattavuuden mutta käyttäjät testaisivat sitä todennäköisesti vajavaisesti, Petri testaisi lähinnä toimiiko vai ei. Konsta samanlaisesti. Ilman laajempaa testiä em. asia kirjautumissivu näyttäisi olevan kunnossa. Mutta jos se olisi huonosti suunniteltu. Esimerkiksi käyttäjän syöttämät kentät menisi suoraan kantaan ilman mitään tarkistusta esim ”SELECT * FROM user WHERE login = ’<käyttäjätunnus>’ AND password = ’<salasana>’” syöte palauttaisi käyttäjän tiedot mikäli käyttäjä tunnus ja salasana olisivat oikein. Jos halutaan koittaa rikkoa edellä mainittu kirjautumissivu, syötetään vaikka käyttätunnukseksi ja salanaksi ”’ or 1”, joka muodostaisi kyselyn ”SELECT * FROM user WHERE login = ’ ’ or 1’ AND password = ’ ’ or 1’”, jolloin sovellus saattaisi päästä kirjautumaan sisään tai palauttaa listan käyttäjistä ja salasanoista, joka tapauksessa sovellus toimisi pahasti väärin. Edellä mainittua virhettä kutsutaan myös SQL-injektioksi, joka on yksi verkon pahimmista tietoturva uhista [OWASP top 10, 2017].

2.2.2 Automaattinen

Testi automaatio taas on nopea tapa testata miten sovellus käyttäytyy eri tilanteissa. Siinä syötetään automaattisesti, eli koneellisesti, ennalta määritelty syöte erilaisiin kenttiin. Tai simuloidaan tietokoneen osoittimen käyttöä. Automaatio on loistava lisä testaus työkaluihin ja säästää paljon aikaa. Testaus automaatioon on luikuisia eri työkaloja joihin palataan myöhemmin. Testiautomaation hyöty korostuu kun automaatio testausta laajennetaan vielä automaattisella koodiin kääntämisellä ja julkaisulla. Eli kun tehdään koodi muutoksia esim. GIT-versionhallintaan, koodi testataan, käännetään ja julkaistaan automaattisesti.

Automaatio testaus on vaikea ottaa käyttöön, koska se vaatii aina uusien työkalujen opettelemista. Testiautomaation laatu riippuu täysin testauksen suunnittelusta, käytetyistä työkaluista ja testi syötteen sisällöstä. Myös automaatiolla saadaan testaamiseen huomattavasti enemmän erilaisia testi caseja vrt. käyttäjä käsin, koska koneellisesti voidaan syöttää helposti tuhansia kertoja erilaisia syötteitä, jota taas ihminen ei jaksaisi tehdä.

2.3 Mitä testaamisella haetaan

2.3.1 Sovelluksen toiminta virhetilanteissa

Ehkä tärkeimpänä, lähes jokaisella sovelluksella on sellaista tietoa itsellään mitä se ei saa vuotaa muualle, vaikka se joutuisi kovan kuormituksen armoille, tai peräti ulkopuolisten hyökkäyksille. Myöskään kenenkään käyttäjän ei pitäisi saada sovellusta minkään laiseen virhetilanteeseen. Kuitenkin jossain vaiheessa käyttäjälle tulee sovelluksen virhe tilanne vastaan, on sovelluksen toimittava siitä oikein, riippuen tosin virheestä.

2.3.2 Sovelluksen toiminta tehokkaasti

Testaamisella voidaan hakea myös sovelluksen rajoja. Mitä tapahtuu jos tietokannan koko kasvaa todella isoksi. Jos ajatellaan että normaalisti olisi vaikka tauluja noin 50 ja jokaisessa noin 10 000 riviä. Mitä tapahtuisi jos käyttö kasvaa vaikka sitten että rivejä olisikin joka taulussa noin 1 000 000, voiko sovellusta käyttää normaalisti. Hidastuisiko sovellus liikaa, kestääkö tietokanta, entä miten käsitellään jos on monta käyttäjää samaan aikaan? Entäpä jos kaikki käyttäjät tekisivät jotain mikä vaatii saman taulun rivin käsittelyä, voiko ne molemmat luoda kantaan rivejä? Päivitys, miten varmistutaan että jos molemmat käyttäjät tekee samaan tietokantariviin muutoksia, kumman muutokset tulevat voimaan, miten hallinnoidaan sovelluksen sisäinen keskustelu siitä että ei tule virhettä?

Sovelluksen käytön lisääntyminen ja sen testaaminen on vaikeaa. Koska kasvua ei voida ennustaa ja suuri käytön lisääntyminen vaikuttaa merkittävästi suorituskykyyn. Ei voida olettaa että sovellus jossa yleensä on vaikka muutama käyttäjä ja suhteellisen vähän dataa

kestää millään tavalla vaikkapa Facebookin kaltaisen kuorman, ongelmia tulisi joka tasolla sovelluksen suorituskyky, palvelinarkkitehtuuri tms.

2.3.3 Koodin katselmointi

Testaamisen ohella koodin katselmointi on tärkeä työkalu sovelluksen toiminnan parantamiseen. Yleensä koodin katselmoinnilla haetaan mahdollisia bugeja joita ei testaamisella saada/saatu kiinni, myöskin kohtia mitkä vaatii optimointia tai sitten vain huonosti kirjoitettua lähde koodia. Yleensä koodia katselmoi joku muu kuin tekijä koska ohjelmoija tulee sokeaksi omaan tekemiseen ja lisäksi yleensä ohjelmoija pyrkii suunnittelu ja ohjelmointi vaiheessa miettimään ratkaisunsa loppuun asti ja niiden virheet voivat jäädä kiinni katselmoinnissa.

Avoimen lähde koodien projekteissa katselmoinnin hoitaa muut projektista kiinnostuneet. Myöskin harrastajat hoitavat myös testaamisen. Suljetun lähdekoodin maailmassa yleensä projektin muut jäsenet katselmoi koodin.

Koodin katselmoinnista huolimatta aina joitakin bugeja pääsee läpi. Mutta ne virheet eivät yleensä haittaa käyttämistä normaalitilanteissa. Mutta epänormaaleissa tilanteissa saattaa jotakin dramaattista tapahtua. Katselmoinnilla haetaan myös asioita jotka helpottavat myöhempää ylläpitoa, huonoja koodauskäytäntöjä ja erikoisia ratkaisuja. Jotka eivät ole vielä ongelma mutta tulevaisuudessa saattaa muodostua isoiksikin asioiksi, kun tekijät vaihtuvat niin huono lähde koodi on ongelma ja vaatii aikaa päästä käsiksi siitä mitä tehdään ja miten tehdään. Vertaa jos koodi on selkeää ja helposti luettavaa ja ymmärrettävää.

2.3.4 Dokumentaatio

Koodin dokumentaatio voi olla periaatteessa missä vain, erillisessä dokumentin hallinta sovelluksista aina vain koodin kommentointiin. Myös eri ohjelmointi kielissä on omia dokumentointi työkaluja esim. Javassa JavaDoc. Koodin dokumentoinnin tarpeellisuudesta on monia mielipiteitä, aina siitä että se on elin tärkeää siihen että dokumentointi on turhaa. Dokumentoinnin vastustajien isoin argumentti on se että hyvin tehty koodi doku-

mentoi itse itsensä, joka kyllä osittain pitää paikkaansa, jolloin dokumentointi on pakollinen paha jota tehdään vain jos koodi on huonoa tai erikoista. Kun taas isoissa projekteissa on hyvä lukea jossain mitä teknologioita on käytetty mahdolliset luovat ratkaisut ja muita erikoisuuksia. Ja koodin kommenteilla haetaan lähinnä metodin tai aliohjelman käyttö ohjeita.

Omasta mielestä koodin kommentointi on lähtökohtaisesti turhaa. Toki ennen ohjelman alkua on hyvä kirjoitella aluksi mitä se tekee, mitä parametreja ottaa sisään ja mitä palauttaa, unohtamatta myöskin virhe tilanteita. Mutta siis yksittäisten koodi rivien kommentointi on hukkaan heitettyä aikaa, sillä muuttujien ja metodien tulee olla nimetty selkeästi. Jolloin kommenttien taso voi olla ”tässä on silmukka” tai ”alustetaan muuttajia”, jonka näkee suoraan koodista. Myöskin yleensä nähnyt kommentoitua koodia, jonka sisällyttäminen valmiiseen koodiin on erittäin paha virhe (lähtökohtaisesti turhaa ja jos tarvetta palata niin sitä varten on version hallinta). Dokumentointia varten on erillinen asiakirja ja javadoc tyylinen kommentointi, jos se ei riitä niin sovelluksen rakennetta ja nimeämiskäytäntöä on syytä miettiä uudelleen.

3 Testaus tyylejä

3.1 Automaatio testaus

Automaatio testauksella tarkoitetaan sitä että sovellus tai osa siitä testataan automaattisesti eli jollain työkalulla. Automaatio testaus ei voi korvata manuaalista testausta, koska se katsoo vain että sovellus toimii oikein, mutta ei ota kantaa kaikkiin virheisiin. Kuten jos on lista joka sisältö on teknisesti oikein ja muualla on sama lista jossa on jokin rajaus. Automaattisesti on erittäin vaikea huomata tällaiset virheet, puhumattakaan kirjoitus virheistä tai lokalisoinnin ongelmista.

3.1.1 Edut

Automaatio testauksella saadaan toistettua lukemattomia kertoja samat testit. Jolloin löytyvät ongelmat ovat helpompi selvittää mistä ne johtuvat. Myöskin saadaan raportti mikä kertoo miten testi meni. Automaatio testaus on myös nopea tapa testata sovellusta vrt. jos käsin tarvitsisi yrittää aina toistaa samoja asioita, myöskin automaatio testauksessa oletuksena käydään kaikki metodit lävitse.

3.1.2 Haitat

Automaatio testauksen isoimmat ongelmat ovat se että testaus vaatii ammattilaisia, eli kuka tahansa ei voi ruveta testaamaan sovellusta automaattisesti. Myöskin aloitus vaatii aina opettelua. Toisin kuin että laitettaisiin sovellus käyntiin ja ruvetaan testaamaan.

3.2 Yksikkötestaus

Yksikkötestauksella tarkoitetaan sitä että, sovellus testataan pienissä osissa eli metodi kerrallansa. Jonka jälkeen sovellusta voidaan testata kokonaisuudessaan, kun ensin on testattu että metodit varmasti toimivat. Yksikkö testausta voi tehdä myös ilman automaatio testausta. Jolloin metodille syötetään parametreina jotain syötettä ja verrataan onko se oikein. Automaattisesti taas sovellus hoitaa testauksen. Testaamiseen löytyy useita eri

kirjastoja, lähtökohtaisesti jokaiselle suositulle ohjelmointikielelle löytyy oma. Osaan ohjelmointi kielistä testaustyökalut ovat sisään rakennettuja.

Isoissa sovelluksissa yksikkötestaus on käytännössä ainut tapa millä tavalla testaan sovellusta ennen kuin se on valmis, lisäksi vielä todennäköisesti eri komponentin valmistuvat eri aikaan. Yksikkö testaamisen avulla voidaan toimittaa valmiita testattuja komponentteja, jolloin lopullinen testaus on vain katsomista että komponentit toimivat yhdessä ja oikein.

3.2.1 Edut

Saadaan nopeasti testattua yksittäisiä paloja. Eli heti kun komponentti on valmis. Sille tehdään yksikkötestit.

3.2.2 Haitat

Yksikkö testillä tulee palaute vain siitä että yksittäinen pala toimii. Se ei ota millään tavalla kantaa kokonaisuuteen

3.2.3 Testikattavuus

Sovelluksen testaamisessa pyritään aina 100 % testikattavuuteen, eli jokainen ohjelman koodirivi on testattu jolloin tiedetään että sovellus toimii jokaisessa tilanteessa oikein. Todellisessa elämässä 100 % on erittäin vaikea saavuttaa, kyseisellä hetkellä kyllä kun jokainen komponentti on testattu hyvin yhdessä ja erikseen. Mutta sovelluksen testaaminen on jatkuva operaatio, eli tilanteet muuttuvat alustat muuttuvat, uusia ongelmia löytyy ja ympärillä olevat ohjelmat muuttuvat jotka vaikuttavat alkuperäisen toimintaan. Esim selain versiot ja ohjelmointi kielet päivittyvät. Osaan muutokseen voidaan vaikuttaa ja niiden käyttöön ottoa hidastamaan, kuten palvelimen puolen ohjelmistot. Kun taas selain puoli on sellainen ”villi” maailma johon ei voi vaikuttaa, etenkin kaupallisessa tilanteessa jolloin asiakkaan selaimiin ei ole mitään vaikutusvaltaa.

3.2.4 Testauksen ongelmia

Web – kehittäjänä isoin vaikeinta on testata selain puolta, käyttöjärjestelmä ja selain eroja on paljon, jonka lisäksi osa selaimista ei noudata samoja sääntöjä, lisäksi ne toimivat hieman eritavalla eri tilanteissa. Ja jos sovellus on virheellinen niin osa selaimista pystyvät toimimaan oikein, osa taas ei.

Kaupallisessa ohjelmistokehityksessä pitää olla tieto asiakkaalla käytössä olevasta ohjelmistosta. Lisäksi osa tilanteista on sellaisia jotka esiintyvät erittäin harvoin tai erikoisessa ympäristössä, joihin normaalilla testaamisella ei päästä kiinni. Esimerkiksi jos olisi käytössä vaikka vanha versio IE – selaimesta, tai sitten asiakkaan tietokoneet ovat rajatussa verkossa, jolloin voi tulla ongelmia vaikka ohjelman varmenteiden kanssa. Olettaen että sovellus toimii internetissä ja käytetään varmenteita.

3.3 Integraatio testaus

Hieman yksikkötestausta laajempi ja siinä testataan sovelluksen laajempia komponentteja. Samalla tavalla kuin yksikkötesteissä, mutta testattava kokonaisuus koostuu useasta yksiköstä. Idea että aluksi melko suppua mutta laajennetaan koko ajan isommaksi ja isommaksi.

3.3.1 Edut

Saadaan testattua miten eri ohjelman komponentit toimii keskenään.

3.3.2 Haitat

Vaatii kehittäjiltä testaus osaamista ja ajattelu tapaa. Myöskin vanhoihin sovelluksiin vaikea liittää, testien laajuudesta johtuen.

3.4 Hyväksyttämistestaus

Testataan sitä kun sovellus on kasattu komponenteista valmiiksi ja testataan miten se kokonaisuutena toimii.

3.4.1 Edut

Ainut tapa testata että sovellus oikeasti toimii kunnolla.

3.4.2 Haitat

Työlästä, loputtomasti variaatioita, täydellinen testaus on mahdotonta. Esim, web-kehityksessä on lukuisia selaimia ja alustoja jotka pitää ottaa huomioon. Myöskin perus sovelluksessa variaatioiden määrä on loputon.

3.5 Käyttäjä testaus

Käyttäjä testauksella tarkoitetaan sitä että joku käyttäjä testaa, eli käyttää sovellusta testimielessä. Raportoi bugeista ja ongelmista.

3.5.1 Edut

Nopea aloittaa. Yleensä käyttäjätestauksella saadaan hyvä kokonaiskuva sovelluksetta. Kuka tahansa pystyy aloittamaan manuaalisen testauksen aloittamalla sovelluksen käytön sillä ajatuksella että kartoittaa onko siitä niin sanottuun oikeaan käyttöön. Tällä tavalla voidaan lähes mitä tahansa sovellusta testata. Myöskin äärettömän tärkeää sovelluksen lopullisessa testauksessa, yleensä loppukäyttäjä tai asiakas tekee viimeisen testauksen. Voidaan myös soveltaa laajemmin, esim pilotointi tai linux käyttöjärjestelmän testaus, jossa katsotaan sopiiko se omiin käyttötarkoituksiin.

3.5.2 Haitat

Laajaa testikattavuutta on hyvin vaikea saada manuaalisessa testauksessa. Myöskin raportointi on hieman ongelmallista ja virheiden toistaminen hankalaa. Johtuen siitä että joku konkreettisesti syöttää sovellukselle jotain syötettä ja katsoo toimiiko se oikein. Voi olla myös tilanteita että käyttäjä ei huomaa tai tiedä että on virhe. Jolloin on sellainen tilanne että käyttäjä ei osaa raportoida siitä eteenpäin. Manuaalinen testaus on myös hidasta jolloin sen tekiminen vaatii aikaa ja rahaa. Hitaus johtuu ihmisen hitaudesta plus siitä että hyvän testikattavuuden saavuttamiseksi täytyy tietää tarkkaan millaista syötettä sovellukselle syöttää.

Käyttäjätestauksen isoin ongelma on toistettavuus, ellei seuraa erittäin tarkkaan laadittua testisuunnitelmaa. Ihmisten ominaisuudesta johtuen käy hyvin helposti siten että ei enää muista miten johonkin tilanteeseen on päästy, eikä ihme sillä jonkin virhetilanteen on saattanut laukaista jokin toiminto mikä on tapahtunut vahingossa ja huomaamatta. Eli sen toistaminen on käytännössä mahdotonta.

3.5.3 Black box -testaus

Mustalaatikko eli black box -testaus tarkoittaa nimensä mukaan sitä että sovellusta testaan kuin ”mustaa laatikkoa”, siis sen toiminnasta ei tiedetä mitään. Testaaja vain katsoo miten se toimii miten toiminnot onnistuu. Tällä tavalla voidaan simuloida loppukäyttäjää kaikista parhaiten. Koska testaaja käyttää sovellusta eri tavalla kuin suunniteltu ja luottaa omaan intuitioonsa ja ennalta opittuihin asioihin.

3.5.4 White box -testaus

Valkoinenlaatikko eli white box -testauksessa, testaaja tietää tarkkaan mitä testaa ja miten sen pitäisi toimia. Myöskin tietää tarkkaan mitä pitää testata. Testaaja voi olla vaikka tuotteen tekijä.

3.5.5 Grey box-testaus

Harmaalaatikko eli grey box -testauksessa, on whitebox ja blackbox testauksen välillä. Eli jonkin tietoa ja taitoa, mutta ei syvällistä tietämystä testattavaan tuotteeseen, esimerkiksi ohjelmoijan työkaveri.

4 Testaus Työkaluja

4.1 JUnit

Yksikkö testaus kirjasto java -ohjelmeinti kielille. Se on avoimen lähkoodin sovellus, eli sen taustalla ei ole suurta yksittäistä jakelijaa, vaan iso joukko kehittäjiä. JUnit:n avulla voidaan myös määritellä aika rajoja mitä sovelluksella saa kestää, esimerkiksi voitaisiin määritellä kuinka kauan tietokantakysely saa kestää.

JUnit voi myös lisätä vanhaan sovellukseen. Suurta kipuilua ei ole, kunhan pitää mielessä että ihan kaikkea ei voida automaattisesti testata.

4.1.1 Käyttö

Normaalien java pakettien lisäksi luodaan Testi paketit. Riippuen Idestä, mutta netbean-sissä se tehdään klikkaamalla java -tiedostoa oikealla, valitsemalla Tools -> Create test. Sen jälkeen kirjoitetaan ja ajetaan testit.

Virheelliset testit ovat sellaisia joissa on heitetty exception tai menty fail() functioon. Se on JUnit oma sovellus joka kertoo virheestä.

Testi tiedostoja voi myös debugata samanlaisesti kuin normaalia sovelluksen käyttöä. Joka helpottaa paljon etenkin testauksen alussa ja opettelussa.

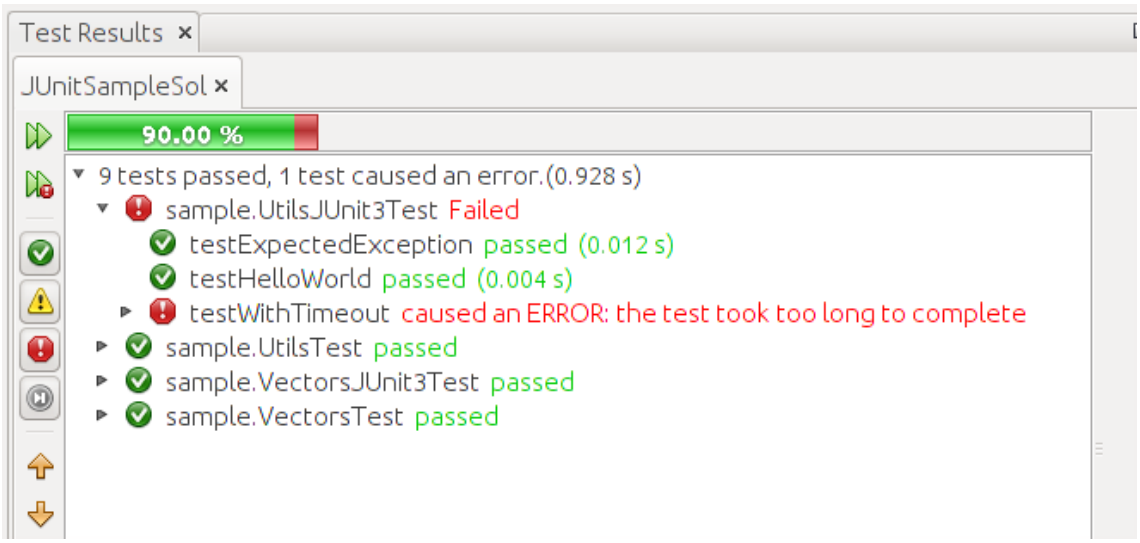
4.1.2 Testi casejen kirjoittaminen

Kun kirjoitetaan testi tiedostoja. Määritellään etukäteen miten niiden pitäisi käyttäytyä. JUnitista löytyy ainakin assertTrue, assertFalse ja assertEquals joiden avulla määritellään se miten sovellus käyttäytyy. Eli jos määritellään assertTrue(Joku boolean ehto) esimer-kiksi assertTrue(tulos == 3); testi menee läpi jos tulos on 3. Kun taas jos olisi määritelty

assertFalse(tulos == 3); testi ei mene läpi, jos tulos olisi ollut 3. AssertEquals sin avulla voidaan määritellä merkkejä että ovatko ne samat.

4.1.3 Tulokset ja niiden lukeminen

JUnit palauttaa erittäin selkeästi, mikä kohta ei mennyt testeistä lävitse. Samalla myös lasketaan myös onnistumisprosentin.



Kuva JUnit:n palautteesta. Kuvasta nähdään että yksi testi ei mennyt läpi.

4.1.4 Ongelmat

JUnit käytössä huomattiin muutamia ongelmia. Testatessa web-sovelluksen java backend tiedostoja, ongelmaksi tuli kanta testaaminen silloin kun tietokantaa tarvitaan. Kun yksikkö testaamisen idea on testata yhtä osaa kerrallansa tyylillä mitä palauttaa ja mitä pitäisi palauttaa. Niin tietokanta luo niin sanotun liikkuvan maalin johon ei voi oikeastaan osua. Yksinkertaisesti yksikkö testaus on hankalaa kun mukana on tietokanta. Ongelmia ovat siis tietokannan määrittely, koska sovellus on pilkottu testauksessa osiin, tietokannan käyttäminen vaatii sen että tietokanta määritellään erikseen jokaisella testattavalla palasella.

4.2 Robot framework

Automatisoitu työkaluhyväksymistestaukseen, jonka avulla voidaan ajaa ennalta määriteltyjä testejä erilaisiin sovelluksiin. Robot framework on julkaistu Apache Licence 2.0 – lisenssillä.

4.2.1 Komponentit

Python tai Jython, vaaditan Robot Frameworkin käyttämiseen.

Geckodriver, vaaditaan kun ajetaan testi-caseja firefoxssa Robot Frameworkilla. Väärä versio aiheutti TimeOut Exceptionin. Eli linkki SeleniumLibraryn ja selaimen välillä.

SeleniumLibrary, apukirjasto jota käytetään selaimien komentojen ajamiseen, eli kenttien kirjoittamiseen ja nappien painamiseen.

4.2.2 Asennus

Asennus oli aluksi yllättävän haasteellista. Isoin ongelma oli eri komponenttien versio riippuvuudet, joista ei ollut mitään selkeää listaa vaan mentiin yritys ja erehdys linjalla. Kun versio riippuvuudet oli ratkaistu, niin Demon ajaminen oli todella helppoa.

```
sudo apt-get install python
sudo pip install robotframework
sudo pip install robotframework-selenium2library


wget https://github.com/mozilla/geckodriver/releases/down-
load/v0.20.0/geckodriver-v0.20.0-linux64.tar.gz
sudo sh -c 'tar -x geckodriver -zf geckodriver-v0.20.0-linux64.tar.gz -O >
/usr/bin/geckodriver'
sudo chmod +x /usr/bin/geckodriver
geckodriver-v0.20.0-linux64.tar.gz
rm geckodriver-v0.20.0-linux64.tar.gz
```

Yllä olevilla komennoilla onnistuu Robot Frameworkin ja niiden riippuvuuksien asentaminen Linux tietokoneeseen.

4.2.3 Käyttö

Voidaan ajaa komentoriviltä tai automaattisesti esim. Jenkins –plugin kautta. Tässä tapauksessa ajetaan testit manuaalisesti komentoriviltä.

4.2.4 Testi casejen kirjoittaminen

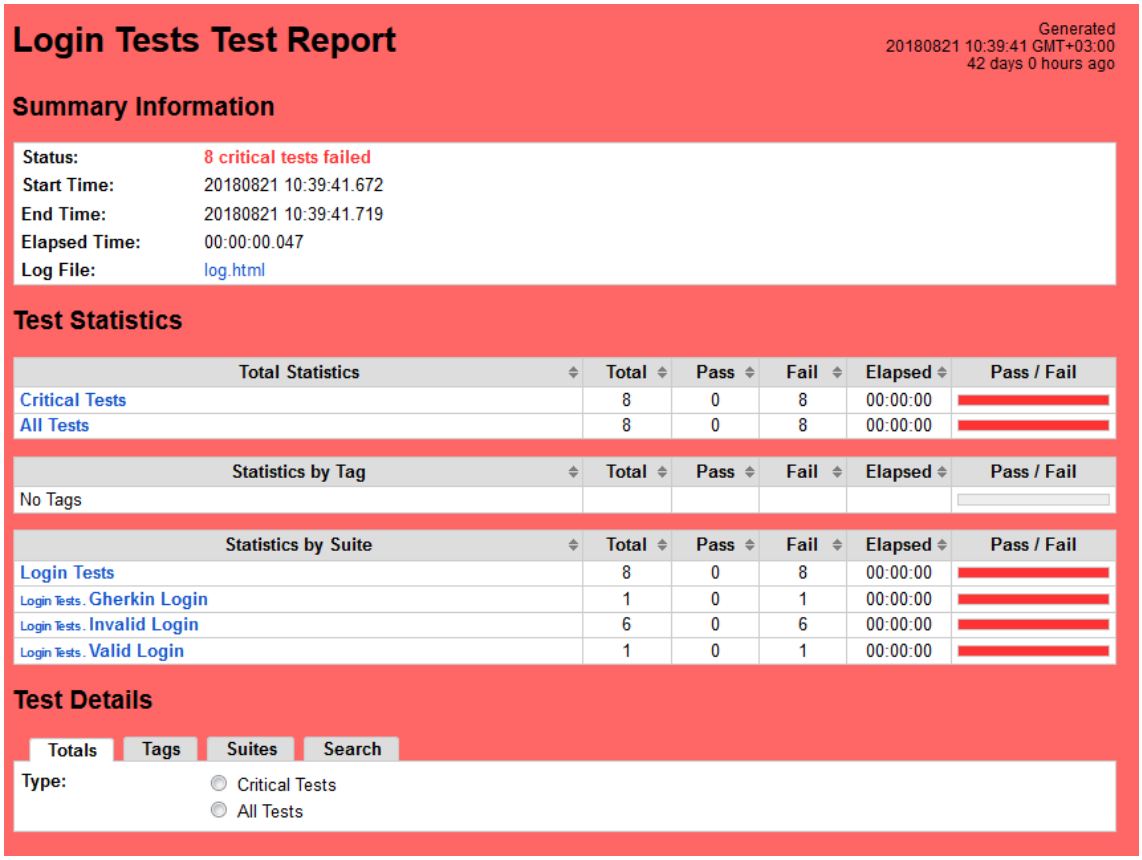
Robot frameworkin testi caset ovat melko yksinkertaisia kirjoittaa. Ne voidaan kirjoittaa jopa sillä tavalla että ihminen jolla ei ole mitään käsitystä Robot frameworkista pystyy jollain tavalla ymmärtämään mitä se tekee esim. ”Open Browser To Login Page”. Edellä mainitusta esimerkistä kuka tahansa pienen englaninkielen avulla pystyy päättelemään että avataan selain kirjautumis -sivulle

Idea on se että kirjoitetaan avainsanojen avulla testit. Eli tyylillä avaa selain, kirjoita kenttiin haluttu syöte, paina nappulaa jne. ja lopuksi mitä halutaan tapahtuvan, esim. avataan selain, mennään johonkin sivulle ja kirjaudutaan sisään, päästiinkö vai eikö? Mitä odottettiin? Robot Framework muodostaa annettujen tietojen peruteella tulos sivun joka kertoo miten testit meni. Kuinka moni hyväksyttiin ja moniko meni hylkyyn. Sille voi myös määritellä kriittisiä testejä.

4.2.5 Tulokset ja niiden lukeminen

Robot frameworki muodostama tulos sivu ja komentorivi palaute ovat selkeitä. Molemmissa kerrotaan mitä meni pieleen, sivusto on huomattavasti laajempi ja sen avulla pääsee myös katselemaan logeja helposti. Kun taas komentorivi kertoo vain että läpi meni ja tai ei ja jotain yksinkertaista virhettä, jos tarvitsee tarkemmin tutkia virheitä. Täytyy kaivaa logit auki ja tutkia niitä.

Lähtökohtaisesti riittää komentorivin pikapalaute, etenkin jos testit onnistuivat. Jos ei niin verkkosivut antavat hieman laajemman kuvan testauksesta.



Login Tests Test Report

Generated
20180821 10:39:41 GMT+03:00
42 days 0 hours ago

Summary Information

Status:	8 critical tests failed
Start Time:	20180821 10:39:41.672
End Time:	20180821 10:39:41.719
Elapsed Time:	00:00:00.047
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	8	0	8	00:00:00	
All Tests	8	0	8	00:00:00	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

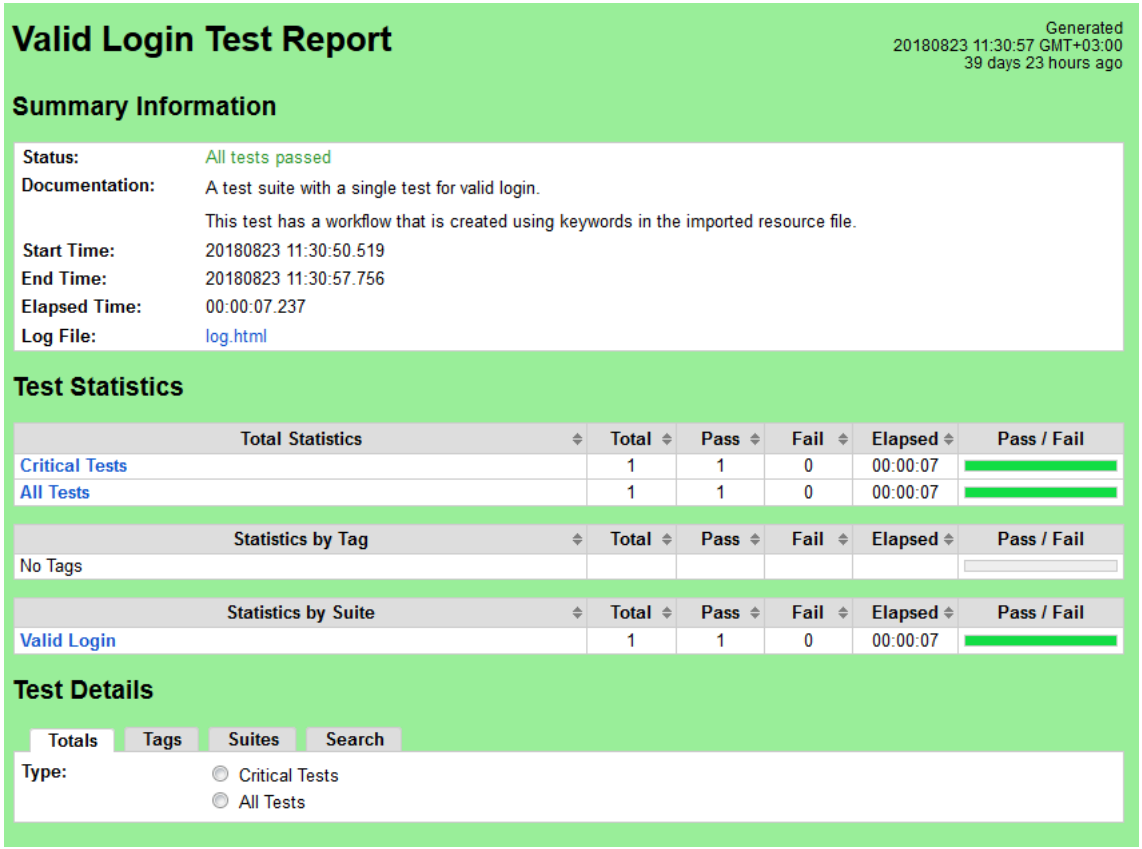
Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Login Tests	8	0	8	00:00:00	
Login Tests. Gherkin Login	1	0	1	00:00:00	
Login Tests. Invalid Login	6	0	6	00:00:00	
Login Tests. Valid Login	1	0	1	00:00:00	

Test Details

Totals | Tags | Suites | Search

Type: ○ Critical Tests ○ All Tests

Kuva palautesivusta, testit eivät menneet lävitse



Valid Login Test Report

Generated
20180823 11:30:57 GMT+03:00
39 days 23 hours ago

Summary Information

Status:	All tests passed
Documentation:	A test suite with a single test for valid login.
	This test has a workflow that is created using keywords in the imported resource file.
Start Time:	20180823 11:30:50.519
End Time:	20180823 11:30:57.756
Elapsed Time:	00:00:07.237
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:07	
All Tests	1	1	0	00:00:07	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Valid Login	1	1	0	00:00:07	

Test Details

Totals | Tags | Suites | Search

Type: Critical Tests / All Tests

Kuva palautesivusta, kun testit menivät lävitse.

4.2.6 Ongelmat

Isoihin ongelmiin ei törmätty johtuen Robot Frameworkin tavasta toimia simuloida selaimen painalluksia. Eli se vain simuloi ihmisen toimintoja. Ainut ongelma mihin törmättiin, oli eri komponenttien yhteensopivuus.

5 Muita ohjelmisto kehityksen työkaluja ja menetelmiä

5.1 Dev Ops

DevOps:lla pyritään automatisoida ohjelmiston kehitykseen, testaamiseen ja ylläpitoon liittyvää työtä. Esim. Kun tehdään git-commit, suorittaan automaattisesti testejä, käännetään ohjelma ja ajetaan tuotantoon. Näin säästetään paljon aikaa ja saadaan nopeasti ajettua uusia muutoksia tuotantoon pienellä vaivalla.

5.1.1 Jatkuva Integrointi (CI)

Idea on että kehittäjät synkronoivat jatkuvasti versionhallinnan kanssa ja pitävät paikalisen kehitys ympäristön ajan tasalla tuotantoversion kanssa.

Jatkuva integrointi pitää olla mukana alusta saakka. Muuten tulee ongelmia integroinnin kanssa jos monet kehittäjät ovat tehneet samaan aikaan muutoksia samoihin tiedostoihin.

5.1.2 Jatkuva toimitus (CD)

Eli muutokset toimitaan pienissä paloissa sitä mukaan kun ne ovat valmiita. Eli tehdään mieluummin monta pientä päivitystä kuin yksi iso.

5.2 Huonoja tapoja

Alla on esimerkkejä siitä millä tavalla sovelluskehityksessä tehdään painajaismaista. Oletuksena on se että on yksi päähaara josta pienillä muokkauksilla saadaan erituotteitta.

5.2.1 Kirsikan poiminta (Cherry picking)

Uutta tuotetta tehdessä otetaan pohjaksi jokin muu kuin päähaarassa oleva, kuten aikaisempi asiakkaan tuote. Kun kirsikan poimintaa harrastetaan pitkään, ollaan pisteessä jossa

tuotteita ei ole yksi vaan lukuisia. Käytännössä mahdoton yhdistää niitä takaisin yhdeksi tuotteeksi,

5.2.2 Haarautuminen

Kun annetaan joidenkin tuotteiden kehittyä oma itsenään liian kauan. Ollaan pisteessä jolloin päähaaran kanssa synkronointi on haasteellista. Tai kun tuotteen muutoksia ei viedä ajoissa päähaaraan. Riskit kirsikan poimintaan kasvaa. Ylläpito vaikeutuu huomattavasti.

6 Pohdinta

Ohjelmistojen testauksessa omasta mielestäni paras lähtökohta on sellainen ”Riittävän lähellä (Close enough)” -ajattelu, täydellisyys on sellainen mihin pitää pyrkiä mutta samalla tiedostaa että se ei saa olla itseisarvo. Esimerkiksi kaupallisessa WEB-kehityksessä pitää ottaa eri selaimet huomioon, mutta vahat ja harvinaiset voi jättää suosiolla testaamatta, Firefox, chrome, Intenet Explorer ja edge riittävät kyllä hyvin. Sitä paitsi suurin osa harvinaisimmista selaimista pohjautuu edellä mainittuihin selaimiin, eli suurella todennäköisyydellä sovellus toimii ihan hyvin.

Toinen vahva mielipide on se että lähtökohtaisesti valmiita sovelluksia ei ole, poikkeuksena kaikki mitä ei enää tueta, esimerkkinä WIN 98, on valmis. Samoin lukuisat opiskelijoiden harjoitustyöt, mutta niillä on yhteistä se, että niitä toivottavasti ei ole missään tuotanto/kriittisessä käytössä. Toki niitä voi ajaa hiekkalaatikossa tai harrastekäytössä. Eli päästään taas siihen että testaus ei lopu koskaan. Myöskin jatkuvasti eri komponentin vanhentuvat ja päivittyvät, esim java tai lukuisat JavaScript -kirjastot. Jolloin niitä täytyy päivittää ja testaus jatkuu. Myöskin tietoturva luo koko ajan uusia haasteita. Mitä haavoittuvuuksia on löytynyt ja miten ne paikataan?

Lähteet

OWASP, 2017 https://www.owasp.org/index.php/Top_10-2017_Top_10

JUnit javadoc, 2018 https://www.junit.org/junit4/javadoc/latest/index.html

Software Testin Tutorial, Tutorials Point (I) Pvt. Ltd 2016, Tulostettu 8.4.2018 https://www.tutorialspoint.com/software_testing/software_testing_tutorial.pdf

Robot Framework. Luettu 27.08.2018. http://robotframework.org

Atlassian, Continuous integration vs. continuous delivery vs. continuous deployment. Luettu 27.08.2018. https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd