Tampereen yliopisto

Aku Lindgren

OHJELMISTOTESTAUKSEN AUTOMATISOINTI JA PARHAAT KÄYTÄNNÖT

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Tammikuu 2020

TIIVISTELMÄ

Aku Lindgren: Ohjelmistotestauksen automatisointi ja parhaat käytännöt
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Tammikuu 2020

Ohjelmistotestauksen automatisointi on välttämätöntä modernissa jatkuvan integraation ohjelmistoprojektissa, sillä tiheässä julkaisusyklissä kehittäjät tarvitsevat mahdollisimman nopean palautteen tekemästään koodista. Testien automatisointi on juurruttanut sijaansa ohjelmistokehityksessä, mutta silti monet yritykset kattavat automatisoidulla testauksella vain vähimmäismäärän, johtuen automatisoitujen testien kehitys- ja ylläpitokuluista. Tutkielman tavoite on selvittää, mitkä tekijät kasvattavat automatisointikuluja, sekä miten resursseja pystytään säästämään samalla säilyttäen testiautomaation tehokkuus.

Tutkielmassa tarkastellaan automatisoitavaksi valittavien testitasojen ja -tyyppien valintaa, sekä testitapausten priorisointia pienempiin kokonaisuuksiin. Tutkittavan aiheen nopean kehittymisen vuoksi tutkielmaan pyrittiin valita mahdollisimman uusia tutkimuksia, jotta lähdemateriaali heijastaisi nykyisiä ohjelmistokehityksen käytäntöjä.

Tutkielma osoittaa, että kaikkia testitasoja ei tule pyrkiä automatisoimaan täysin. Automatisoituja testejä suorittamalla harvoin löydetään uusia vikoja ohjelmistosta, mutta niiden avulla voidaan varmistaa, ettei aiemmin toiminut ominaisuus ole hajonnut ja näin manuaaliset testausresurssit voidaan ohjata monimutkaisempiin tehtäviin. Testiautomaatioprojektin kuluista suurin osa syntyy testien ylläpidosta, ja ylläpidon laiminlyömisestä kertyneestä teknologisesta velasta. Testien suorittaminen usein mahdollistaa hajonneiden testitapausten tunnistamisen ajoissa ja pidentää näin testitapausten elinikää.

Sisällysluettelo

1 Johdanto

Useimmat modernit ohjelmistoprojektit noudattavat jatkuvan integraation mallia (continuous integration), joka tarkoittaa menetelmää, missä ohjelmistokehittäjät tekevät lähdekoodimuutoksensa primaarisesta lähdekooditietokannasta johdetussa kehityshaarassa. Kehityshaarassa tehdyt muutokset sulautetaan runkoon, kun muutokset ovat valmiit. Kehityssykli on nopea, jonka vuoksi testauksen automatisointi on olennainen osa jokaista ohjelmistoprojektia, sillä manuaalinen testaaminen on hidasta (Cohn, 2010). Automatisoitu testaus on nykypäivänä hyvin yleistä, mutta sen käyttöönotto on erinäisistä syistä johtuen monessa yrityksessä jäänyt yksikkötestien suorittamiseen (Kasurinen et al., 2010). Osittain testausautomaation laajamittaisen käyttöönottamisen esteenä ovat siihen liittyvät kulut (Berner et al., 2005; Kasurinen et al., 2010). Tämä tutkielma selvittää mitkä ovat automatisoidun ohjelmistotestauksen parhaat käytännöt jatkuvan integraation viitekehyksessä, sekä miten testit voidaan toteuttaa tehokkaasti ja edullisesti alati kasvavissa ohjelmistoprojekteissa.

2 Ohjelmistotestaus

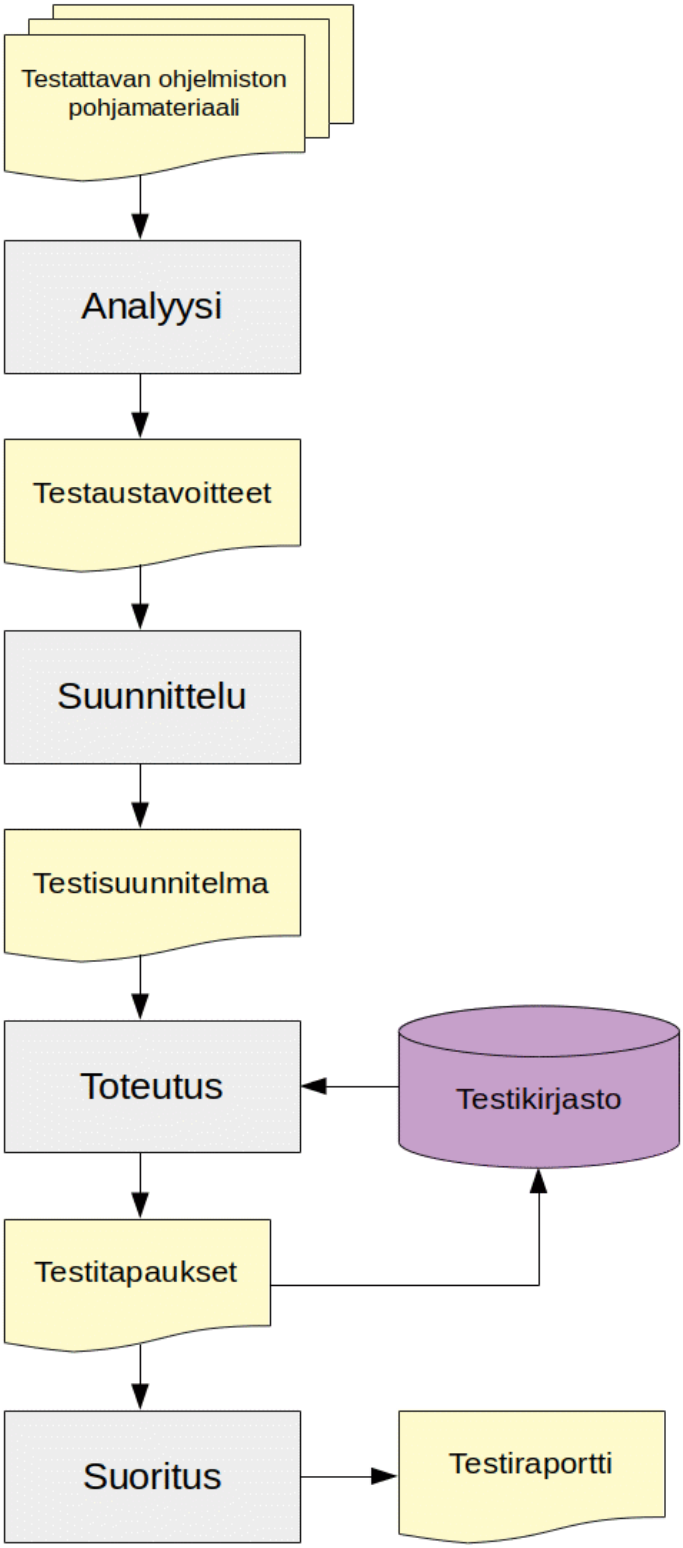
Testaus on prosessi, jonka tavoitteena on määrittää ohjelmiston laatu ja löytää siitä vikoja, näin vähentäen kuluja samalla taaten ohjelmiston laadun (Kasurinen et al., 2010). Prosessi sisältää itse testaamisen lisäksi testattavan järjestelmän analysoinnin ja testien suunnittelun, sekä tulosten tutkimisen ja raportoinnin. (International Software Testing Qualifications Board, 2019, s. 13). Alla esittelen tiivistetysti testausprosessin ja sen artefaktit, joita esittää graafisesti kuva 1.

Testin analysointivaiheessa määritellään testaustavoitteet saatavilla olevan pohjamateriaalin mukaan. Pohjamateriaalia ovat muun muassa järjestelmän vaatimukset, ja suunnitteluasiakirjat. Vaiheessa märitellään testattavat ominaisuudet ja todennäköisimmät viat.

Testisuunnittelu sisältää testaustavoitteiden kääntämisen testitapauksiksi. Testisuunnittelun yhteydessä luodaan testisuunnitelma, joka sisältää suoritettavat testitapaukset.

Testin toteutusvaiheessa luodaan tarvittavat resurssit testin suoritusta varten testauksenhallintatyökaluun, määritetään testitehtävät ja kirjoitetaan mahdolliset tarvittavat testiskriptit suoritustyökalulle.

Testin suoritusvaiheessa toteutusvaiheessa tuotetut testitehtävät suoritetaan, ja niiden pohjalta luodaan testiraportti, johon koostetaan yhteenveto suoritetuista testeistä ja niiden tuloksista. Testauksen tuloksia verrataan järjestelmän hyväksymiskriteereihin, jonka perusteella järjestelmä hylätään tai hyväksytään laadunvalvonnassa.



Kuva 1. Testausprosessi tiivistetysti.

Testitapaus on kokoelma ennalta määrättyjä syötteitä, järjestelmän esiehtoja, odotettuja tuloksia, sekä vaatimus järjestelmän tilasta suorituksen jälkeen. Testitapaukset suunnitellaan yleensä yksilöllisesti testattavalle objektille ja se voi sisältää useita askelia.

Useasta testitapauksesta muodostetaan testikokonaisuus, jolla varmistetaan testattavan järjestelmän toiminta. Testikokonaisuuden sisältö ja laajuus määrittyy testisuunnitelman mukaan.

Testikirjasto on kokoelma testitapauksia ja testikokonaisuuksia. Testikirjastosto nopeuttaa testien toteutusvaihetta, tarjoamalla valmiita testitapauksia ja -kokonaisuuksia suunnitteluvaiheen käyttöön, jolloin jokaista testitapausta ei välttämättä tarvitse kirjoittaa alusta asti uudelleen (Berner et al., 2005).

2.1 Testitasot

ISTQB:n (2019) määritelmän mukaan, testitasot ovat testausaktiviteetteja, joita organisoidaan ja hallitaan yhdessä. Tietyn testitason aktiviteetit ovat yhteydessä ohjelmistoprojektin tiettyyn vaiheeseen.

Testitasot jaotellaan neljään luokkaan, ja ne testaavat järjestelmää eri kokoisina kokonaisuuksina aina tietyn ohjelmistokomponentin yksikkötestauksesta koko järjestelmän hyväksymistestaukseen asti. Mitä matalammalla tasolla viallinen tai puutteellinen toiminnallisuus huomataan, sitä helpompi vian lähde on löytää (Berner et al., 2005; Cohn, 2010; International Software Testing Qualifications Board, 2019).

Yksikkötestaus, unit testing

Yksikkötestauksella, eli komponenttitestauksella testataan yksittäisen ohjelmistokomponentin toiminta. Tavallisesti ohjelmoijat kirjoittavat itse yksikkötestit komponenteilleen. Yksikkötestit ovat laajuudeltaan pieniä, tarkoituksena varmistaa komponentin määritelty toiminnallisuus (International Software Testing Qualifications Board, 2019).

Integraatiotestaus, integration testin

Integraatiotestaus suoritetaan kun ohjelmistoon lisätään uusia ominaisuuksia. Integraatiotestauksen tavoitteena on varmistaa, että komponentit ja niiden rajapinnat toimivat yhdessä määritellyllä tavalla, sekä löytää vikoja ennen suurien testikokonaisuuksien suorittamista. Kun olemassaolevaan lähdekoodiin tehdään muutoksia, suoritetaan regressiotestaus. Regressiotestauksella varmistetaan että muutokset eivät ole tuottaneet uusia vikoja ja aiemmin toimivaksi varmistetut ominaisuudet toimivat edelleen.

Järjestelmätestaus, system testing
Järjestelmätestauksen tarkoitus on testata järjestelmää kokonaisuutena, kun sen toiminnalle tärkeät komponentit on integroitu. Testauksella varmistetaan, että järjestelmä täyttää sille asetetut vaatimukset.

Hyväksymistestaus, acceptance testing
Korkeimmalla testitasolla järjestelmää testataan käyttäjän, asiakkaan ja mahdollisten lakien asettamien vaatimuksien puitteissa. Hyväksymistestauksen tarkoitus ei usein ole löytää vikoja ohjelmistosta, vaan testata, että työnkulku ja menettelytapojen täytäntöönpano täyttävät vaaditut kriteerit.

3 Testauksen automatisointi

Hyvin suunniteltu ja toteutettu testauksen automatisointi nopeuttaa kehitys- ja laadunvalvontaprosessia sekä säästää organisaation varoja (Berner et al., 2005; Cohn, 2010; Stresnjak & Hocenski, 2011). Berner et al. (2005) huomauttavat, että automatisoitujen testien kehittäminen on usein mittava investointi, etenkin mitä myöhemmässä kehitysvaiheessa ohjelmisto on ja organisaation tuleekin tunnistaa millä tavoin sijoitettu pääoma tuottaa yritykselle takaisin. Sijoitus ei yleensä maksa itseään välittömästi takaisin testauskuluissa, vaan sen tuottoaste muodostuu koko ohjelmistokehityksen sulavoitumisesta. Automatisoidut testit mahdollistavat lyhemmän julkaisusyklin, kun testaamiseen kuluva aika vähenee huomattavasti, ja ohjelmiston viat havaitaan aikaisemmassa vaiheessa. Testien suorittamisen lisäksi automatisoinnin yhteydessä syntyy joissain tapauksissa koko ohjelmistoprojektille hyödyllisiä työkaluja, kuten automatisoituja asennus- ja kokoonpanoprosesseja (Berner et al., 2005; Stresnjak & Hocenski, 2011).

Automatisoituja testejä sisällyttävää testausstrategiaa suunnitellessa tulee tunnistaa automatisoitujen testien funktio ohjelmistokehityksessä. Automatisoiduilla testeillä löydetään harvoin uusia vikoja ohjelmistosta, mutta ne ovat erinomaisia laadunvalvontaan ja toimintojen verifioimiseen (Kasurinen et al., 2010; Marick, 1998; Stresnjak & Hocenski, 2011), sekä havaitsemaan uudelleen esiintyviä, aiemmin tunnistettuja vikoja. Automaatiolla ei voidakaan korvata manuaalista testausta, mutta sillä voidaan vapauttaa testaajaresursseja monimutkaisempiin tehtäviin, parantaen koko testausprosessin laatua ja kattavuutta (Berner et al., 2005). Testauksen automatisointi on muodostunut moderniksi standardiksi jatkuvan integraation ja Scrum-mallin myötä, missä julkaisusykli on yhdestä neljään viikkoa pitkä.

3.1 Automatisoitavien testitasojen valinta

Kasurisen et al. (2010) suorittaman kyselytutkimuksen mukaan ylläpito- ja kehityskulut ovat yleisiä esteitä testiautomaation käyttöönottoon yrityksissä. Testaustrategiaa suunni-

teltaessa on tärkeää määritellä mitä testejä automatisoidaan ja miten usein niitä suoritetaan. Helposti perusteltavia testejä automatisoitavaksi ovat sellaiset testit, joita suoritetaan toistuvasti uudelleen, esimerkiksi jokaisen koodimuutoksen yhteydessä (Berner et al., 2005; Kasurinen et al., 2010; Marick, 1998). Modernissa jatkuvan integraation mallissa on yleistä, että yksikkötestit automatisoidaan suoritettavaksi jokaisen koodimuutoksen yhteydessä (Berner et al., 2005; Cohn, 2010). Cohnin (2010) mukaan tehokkaan Srumin edellytys on, että jokaisella ohjelmistokomponentilla on oma yksikkötestinsä, joka luodaan komponentin kehittämisen ohella. Yksikkötestit ovat nopeita suorittaa ja toisistaan eristettyjä, jolloin yhden komponentin muutos ei voi rikkoa toisen komponentin yksikkötestiä (Cohn, 2010; International Software Testing Qualifications Board, 2019).

Manuaalista testaustaakkaa vähentää merkittävästi regressiotestaamisen automatisointi (Berner et al., 2005; Kasurinen et al., 2010). Regressiotestauksella varmistetaan, että ohjelmistoon tehdyt muutokset eivät ole rikkoneet aiemmin toimiviksi varmistettuja toiminnallisuuksia (Engström et al., 2010; International Software Testing Qualifications Board, 2019). Regressiotestaus kuluttaa keskimäärin noin 80 prosenttia koko testausbudjetista, ja voi kuluttaa jopa 50 prosenttia koko ohjelmiston ylläpitobudjetista (Chittimalli & Harrold, 2009). Cohnin (2010) mukaan regressiotestauksen automatisointi on välttämätöntä modernissa Scrum-mallissa, missä regressiotestejä suositellaan suoritettavaksi jokaisen koodimuutoksen yhteydessä. Regressiotestit testaavat suurempia integroituja kokonaisuuksia kuin yksikkötestit, joten ne ovat alttiimpia hajoamiselle ja ovat hieman hitaampia suorittaa (Berner et al., 2005). Automatisoidut regressiotestit vapauttavat testaajaresurssit muihin tehtäviin, jolloin aikaa jää perusteellisempaan testaukseen ylemmissä testitasoissa.

Cohn väittää myös, että käyttöliittymätestausta tulisi automatisoida mahdollisimman vähän, sillä käyttöliittymätestien kirjoittaminen ja ylläpitäminen on erityisen työlästä. Käyttöliittymät ovat erityisen alttiita muutoksille ja usein käyttöliittymän muutos rikkoo tai mitätöi automatisoidun testin pätevyyden. Tosin Käyttöliittymätestauskehys Robot Framework on nopea ja helppo alusta kehittää käyttöliittymätestejä luonnollisella kielellä, luultavasti maksaen ylläpitokulut nopeasti takaisin testien kehittämisen ja suorittamisen nopeudella (Stresnjak & Hocenski, 2011). Käyttöliittymätestien suorittaminen on kuitenkin huomattavasti hitaampaa kuin esimerkiksi regressiotestaaminen ja siksi usein suoritettava käyttöliittymän testikokoelma tulisi pitää mahdollisimman pienenä, esimerkiksi vain perustoiminnallisuudet kattavana savutestikokoelmana.

Automatisointia suunniteltaessa on huomioitava testattavan järjestelmän kehityksen tila, sillä alituisesti muuttuvan järjestelmän testien ylläpitoon kuluu valtava määrä resursseja (Berner et al., 2005; Kasurinen et al., 2010; Marick, 1998). Automatisoitujen testien laajuus kannattaa projektin alussa pitää pienenä, jolloin ylläpidettävää on vähem-

män alati muuttuvassa ympäristössä (Berner et al., 2005; Cohn, 2010; Stresnjak & Hocenski, 2011). Stresnjak & Hocenski (2011) suosittelevat luomaan automatisointiprojektin alussa runsaasti pieniä skriptejä, jotka antavat testaajalle välittömän palautteen. Testien on tarkoitus vaatia vain pieni aikapanostus niiden luomiseen ja niillä vähennetään manuaalista testaustaakkaa heti projektin alussa. Näiden skriptien avulla tutustutaan testattavaan ympäristöön, testityökaluihin, ja luodaan pohjaa laajemmalle testikokonaisuudelle, eikä niiden hylkääminen projektin myöhemmissä vaiheissa ole menetys, sillä ne ovat maksaneet itsensä takaisin hyvin nopeasti.

3.2 Testien ylläpito

Berner et al. (2005) kertovat kokemuspohjalta tutkimuksessaan, että testien automaatio usein epäonnistuu siitä syystä, että niitä ei suoriteta tarpeeksi useasti. Testikokoelma hajoaa sellaiseen tilaan, missä testitapaukset ovat epäjohdonmukaisia ja vaikeita ymmärtää. Usein suoritetut testit auttavat huomaamaan testien vanhentumisen aikaisessa vaiheessa, jolloin sen korjaaminen on vielä kustannustehokasta ja testin elinikä pitenee. Pitkään käyttämättömänä ollut testi saattaa olla edullisempaa hylätä ja kirjoittaa alusta uudelleen, kuin korjata (Berner et al., 2005; Marick, 1998). Berner et al. (2005) esittelevät tutkimuksessan neljävaiheisen mallin, jolla automaatioprojekti usein epäonnistuu. Malli kuvaa miten automaatioprojektin *eheys*, eli kuinka luotettavasti testit toimivat, ja *ymmärrettävyys,* eli kuinka helposti testiskriptiä tutkimalla voi ymmärtää sen interaktion testattavan järjestelmän kanssa, kehittyy ajan myötä.

Ensimmäisessä vaiheessa luodaan pieni määrä tarkoin rajattuja automatisoituja testejä. Tässä vaiheessa testikokoelma on eheä ja helposti ymmärrettävä. Tässä vaiheessa jotkin testit tuottavat vääriä positiivisia tuloksia, ja nämä testit korjataan välittömästi.

Toisessa vaiheessa testien kehittäjät ovat oppineet automaatiotyökalujen käytön ja useita uusia testitapauksia lisätään testikokoelmaan lyhyessä ajassa. Testikokoelmat ovat edelleen suhteellisen pieniä ja helposti ymmärrettäviä, joskin nopeasti kasvavia. Ymmärrettävyytensä takia vääriä positiivisia tuloksia tuottavia testejä ei korjata välittömästi, ja syntyy kokemus, että joidenkin testitapausten tuottamat väärät tulokset ovat ”normaaleja”, sitä paitsi uusien testitapausten kirjoittaminen tuntuu nyt tärkeämmältä kuin vanhojen ylläpitäminen. Ajan myötä, ylläpidon puutteesta johtuen, testitapausten tuntemus ja luottamus niiden toimintaan hiljalleen katoaa. Toisen vaiheen lopussa testikokonaisuudet eivät ole eheitä tai hyvin ymmärrettäviä.

Kolmannessa vaiheessa testikokonaisuudet ovat rampautuneet ylläpidon puutteesta, eikä niiden suorittaminen tuota enää mielekkäitä tuloksia. Testikokokonaisuuksien saattaminen toimivaan tilaan vaatii tässä vaiheessa huomattavia määriä resursseja ja on joissain tapauksissa yritykselle liian kallista, jotta se voidaan perustellusti mahduttaa projektibudjettiin. Tämän vaiheen puolivälissä on yleensä automaatioprojektin viimeinen

mahdollisuus palauttaa testikokonaisuuden toiminnallisuus, ennen kuin sen eheys ja ymmärrettävyys vajoaa pisteeseen, josta sitä ei ole enää mahdollista tai taloudellista palauttaa.

Neljännessä vaiheessa automatisoidut testit ovat hajonneet täysin. Tässä vaiheessa projektin luonteesta riippuen määrittyy, yritetäänkö automatisoitua testausta rakentaa uudelleen.

Testien jatkuva ylläpito on tärkeää jo projektin aikaisessa vaiheessa, jolloin teknistä velkaa ei pääse kertymään. Tekninen velka tarkoittaa ratkaisuja, jotka haittaavat ohjelmistoprojektin jatkokehitystä tai sen ylläpitoa (Lehojärvi, 2017). Tässä tapauksessa viittaan teknisellä velalla rikkinäisiin testeihin, jotka eivät vastaa testattavan ohjelmiston nykytilaa, ja joiden korjaamiseen vaadittavat resurssit kasvavat sen myötä, mitä pidempään niiden korjaamista lykkää. Testejä tulee varautua päivittämään aina uuden testattavan ohjelmiston julkaisun myötä (Berner et al., 2005). Ohjelmistoon voidaan lisätä ominaisuuksia, joiden testaamista vanhat testit eivät kata, tai olemassa olevia ominaisuuksia muutetaan niin, että testit eivät sellaisenaan pysty niitä testaamaan. Myös järjestelmän vaatimuksia saatetaan muuttaa niin, että ennen virheen tuottanut syöte luetaan hyväksytysti. (Marick, 1998)

Esimerkiksi sähköpostiosoitteen syötteen tarkistava suodin on aiemmin määritelty tuottamaan varoitus väärästä sähköpostiosoitteesta, mikäli domainin pääte on *.ug.* Jos testitapaus on käyttänyt .ug-päätettä virheellisen syötteen testaamiseen ja myöhemmin suotimen asetuksia on muutettu niin, että .ug-loppuiset domainit hyväksytään, tuottaa testitapaus virheellisesti hylätyn.
Mikäli käyttöliittymää muutetaan niin, että domainin pääte valitaan pudotusvalikosta vapaan tekstikentän sijaan, ei testitapaus enää kata uuden ominaisuuden testaamista, mikä voi johtaa virheelliseen testitulokseen.

Jatkuvan integraation malli tarjoaa erinomaisen alustan suorittaa automatisoituja testejä jatkuvasti eri kokoisissa kokonaisuuksissa. Malli mahdollistaa pienempien testikokoelmien suorittamisen jokaisen kehityshaarassa tehtävän muutoksen yhteydessä. Suurempia testikokonaisuuksia voidaan suorittaa runkoon sulautuksen yhteydessä, jolloin kehitysvaiheessa kehittäjä saa nopean palautteen muutoksistaan, kun tuloksia ei tarvitse odottaa täysikokoisen testikokonaisuuden suorittamisen ajan. Pienemmän testikokonaisuuden valitsemisesta lisää luvussa 3.3.

3.3 Automatisoitavien testien priorisointi

Valtavien testijoukkojen ylläpitäminen vaatii jatkuvasti resursseja. Esimerkiksi regressiotestisettien koko kasvaa testattavan ohjelmiston myötä ja samalla kasvavat myös testien ylläpidon kulut (Haghighatkhah et al., 2018). Testien priorisointi tarkoittaa testitapausten järjestämistä testikokoelmassa jonkin tietyn ominaisuuden mukaan. Priorisoinnin perusteella usein valitaan vain tietty joukko testitapauksia suoritettavaksi kulloisessakin

kontekstissa ja niillä pyritään saada paljastettua mahdollisimman suuri osa ohjelmiston vioista testijoukon kärjessä olevilla testitapauksilla (Strandberg et al., 2016). Käytän tässä luvussa priorisointia terminä testijoukkojen valitsemiselle suuremmasta joukosta.

Testien priorisointia varten on kehitetty useita malleja, jotka Haghighatkhah et al. (2018) jakavat erilaisuuteen- (diversity based) tai historiaan (history based) pohjautuvaan testien priorisointiin.

Monimuotoisuuteen perustuvat testipriorisoinnin mallit suosivat testitapauksia muiden testitapausten kattavuuden perusteella. Mallilla vältetään testaamasta samaa järjestelmän osaa monella eri testitapauksella, tarkoituksena estää saman vian löytäminen usealla testitapauksella (Haghighatkhah et al., 2018). Monimuotoisuus voidaan määrittää esimerkiksi testiskriptien metodikutsuista, testitapausten nimistä tai itse tekstiskriptin sisältämästä tekstistä (Hemmati et al., 2017). Ledrun et al. (2012) kehittämässä mallissa testitapausten erilaisuus järjestetään skriptien merkkijonoja vertailemalla niiden etäisyyttä toisiinsa Manhattan-etäisyydellä mitattuna. Testikokoelman *T'* jokaiselle testitapaukselle *t* määritetään sen etäisyysarvo *d.*

$$dd(t, T', d) = min\{d(t, t_i) \mid t_i \in T', t_i \neq t\}$$

Funktio priorisoi testitapaukset, jotka ovat kauimpana testikokoelmasta, ja täten uuden testijoukon huipulla pitäisi olla kokoelma toisistaan mahdollisimman erilaisia testejä. Monimuotoisuuspriorisointi on tehokasta silloin, kun projektissa ei ole vielä paljoa dataa aiemmin suoritetuista testiajoista.

Historiaan perustuvan testipriorisoinnin on esitetty olevan tehokas tapa priorisoida häiriöitä aikaansaavia testitapauksia jatkuvan integraation ohjelmistoprojekteissa (Haghighatkhah et al., 2018; Hemmati et al., 2017; Memon et al., 2017; Strandberg et al., 2016). Menetelmässä vertaillaan aiempia testiajoja ja poimitaan niistä testitapauksia, jotka ovat aikaansaaneet häiriön testattavassa ohjelmistossa. Hemmatin et al. (2017) kehittämässä historiaan perustuvassa mallissa testitapaukset järjestetään sen perusteella, milloin ne viimeksi ovat aiheuttaneet poikkeaman ohjelmistossa. Viimeisimmässä testiajossa poikkeaman aiheuttanut testitapaus saa korkeimman arvon, sitä edeltävässä testiajossa poikkeaman aiheuttanut toiseksi korkeimman arvon ja niin edelleen. Malli priorisoi kokoelmaan lisätyt uudet testitapaukset vanhojen, epäonnistuneiden testien edelle, jolloin mahdollisesti uusia ominaisuuksia käsittelevät testit ovat testijoukon kärjessä.

Haghighatkhah et al. (2018) havaitsivat historiaan perustuvalla testipriorisoinnilla valitun testikokoelman kattavan suurimman regressiovioista. Tämä johtuu lyhyellä syklillä kehitettävän ohjelmiston luonteesta, missä uusia koontiversioita tuotetaan joissain tapauksissa useita päivässä. Haghighatkhahin et al. tutkimuksen mukaan testaushistorian ei tarvitse olla kovinkaan suuri, jotta saavutetaan tiivis seula regressiovikojen varalle. Jo

edelliseen testiajoon vertaaminen paransi testijoukon suoriutumista verrattuna satunnaiseen otantaan testitapauksia. Samassa tutkimuksessa todetaan historia- ja monimuotoisuuspriorisoinnin yhdistelmän olevan kaikkein tehokkain tapa havaita regressiot.

Riskeihin perustuvalla testauksessa suoritetaan ohjelmiston toiminnallisuuksille riskiarvio, jonka pohjalta luodaan testisuunnitelma (Amland, 2000; Storoj, 2012). Riskiarvion avulla testausresurssit kohdistetaan ohjelmiston kriittisimpiin toimintoihin (Amland, 2000). Riskiarviossa tunnistetaan mahdolliset esiintyvät viat ja arvioidaan niiden kokonaisriski. Kokonaisriskin laskemiseen Amland (2000) käytti pankille kehitettävän ohjelmiston tapaustutkimuksessaan kaavaa

$$Kr(f) = T(f) \times K(f)$$

missä $Kr(f)$ on toiminnon kokonaisriski; $T(f)$ vian esiintymisen todennäköisyys toiminnossa ja $K(f)$ on viasta aiheutuvat rahalliset kulut, jos toiminnossa esiintyy vika tuotantoon julkaistussa koodissa. Kokonaisriskin arvioimiseen tulee kuitenkin kehittää projektikohtaiset mittarit, joiden viitekehyksessä riskejä arvioidaan (Storoj, 2012). Jotkin alat, kuten lääketiede, ovat voimakkaasti säänneltyjä ja vaativat riskiarvioinnin tekemistä, jota tulisi hyödyntää ohjelmistoa kehitettäessä. Tällaisissa säännellyissä ympäristöissä riskianalyysia voidaan hyödyntää testien suunnitteluun ja priorisointiin. Testaajien tehtäväksi jää arvioida mitkä testeistä soveltuvat automatisoitavaksi ja mitkä ovat tehokkaampia ja turvallisempaa testata käsin.

Googlen kehittämässä koneoppimista hyödyntävässä Efficacy Presubmit Service -järjestelmässä (EPS) suuresta testikirjastosta suoritetaan koodimuutosten yhteydessä vain ne testit, jotka kohdistuvat muutettujen komponenttien testaamisen (Memon et al., 2017; Spragins, 2016). Periaate on sama kuin muutettuun koodiin kohdistetussa priorisoinnissa, eli testikokoelmaan pyritään valitsemaan vain ne testit jotka testaavat vain muutettuja komponentteja (Hemmati et al., 2017; Strandberg et al., 2016), mutta Googlen käyttämässä mallissa huomioon otetaan myös testien suoritushistoria, painottaen erittäin vahvasti deterministisesti hylättyjä tuloksia tuottavia testejä, sillä Spraginsin (2018) mukaan onnistuneiden testien suorittaminen ei anna hyödyllistä palautetta ohjelmistokehittäjille. Testimenetelmän kehittämisen perustana on Googlen valtava lähdekooditietokanta, noin 86 teratavua vuonna 2015, jonka testaamiseen on kehitetty yksi maailman suurimpia testikirjastoja (Levenberg & Potvin, 2016; Spragins, 2016). Jo testien valtavasta määrästä johtuen perinteiset priorisointimenetelmät eivät tarjonneet testeille riittävää herkkyyttä ja tehokasta suoritusaikaa (Memon et al., 2017). EPS suoritetaan Googlen monoliittisen versiohallinnan presubmit vaiheessa, eli ennen kuin koodimuutokset sulautetaan primaariseen lähdekooditietokantaan antaen kehittäjälle palautteen mahdollisimman nopeasti. Kun muutettu lähdekoodi sulautetaan primaariseen lähdekoodiin, suoritetaan huomattavasti laajempi testikokoelma, jonka tarkoituksena on varmistaa, että EPS ei karsinut minimoidusta testikokoelmasta virheen tuottavia testejä pois (Spragins, 2016).

4 Yhteenveto

Automatisoitujen testien kehittämiseen ja ylläpitoon vaadittava rahallinen ja henkilötyötuntien panostus ovat monelle projektille liian suuri sijoitus (Berner et al., 2005; Kasurinen et al., 2010). Automaatioprojekti onkin helpoin aloittaa heti projektin alussa, jolloin pienilläkin testeillä saadaan vähennettyä manuaalista testaustaakkaa, kehittäjät saavat nopeaa palautetta koodistaan, ja samalla testaustiimi kartuttaa tuntemustaan testattavasta järjestelmästä (Stresnjak & Hocenski, 2011). Vaikka automaatiota ei olisi aloitettu heti projektin alkuvaiheessa, on tehokkaaksi osoitettu tapa aloittaa automaation luominen ”pohjalta ylös”, eli aloittaen yksikkötestien automatisoinnista ja lisäten vähitellen regressiotestejä testikirjastoon (Berner et al., 2005; Cohn, 2010). Bernerin et al. (2005) mukaan jos jokin testi pitää suorittaa yli kymmenen kertaa, se on hyvä kandidaatti automatisoitavaksi.

Testiautomaation suurimmat kulut syntyvät testien ylläpitämisestä (Chittimalli & Harrold, 2009). Testejä ei tulisi päästää ajautumaan vanhentuneeseen tilaan mistä niitä on työlästä palauttaa takaisin toimintaan, vaan hajoavat testit tulisi korjata mahdollisimman nopeasti, ettei teknistä velkaa pääse kertymään niin paljon, että se rampauttaa koko automaatioprojektin (Berner et al., 2005). Testien suorittaminen tiheästi nopeuttaa hajoavien testien huomaamista. Jatkuvassa integraatiossa voidaan suorittaa jokaisen koodimuutoksen yhteydessä jokin kokoelma testitapauksia, mikä tarkoittaa kehitystiimin koosta riippuen kymmenistä satoihin testisuorituksiin päivässä (Berner et al., 2005; Memon et al., 2017).

Suuren testikirjaston suorittaminen jokaisen koodimuutoksen yhteydessä ei yleensä ole tarpeellista, eikä edes ajallisesti mahdollista. Testitapauksia priorisoimalla testit järjestetään sen mukaan, miten todennäköisesti ne löytävät vikoja muutetusta koodista. Jatkuvassa integraatiossa tehokkaaksi tavaksi on todistettu historiaan pohjautuva priorisointi, eli testien suoritushistorian tarkastelulla saatu järjestys kaikkein todennäköisimmin epäonnistuvista testitapauksista (Haghighatkhah et al., 2018). Priorisoiduista testitapauksista voidaan kuratoida pienempiä testikokonaisuuksia suoritettavaksi esimerkiksi kehityshaaran koodimuutoksien yhteydessä, jolloin kehittäjän ei tarvitse odottaa suuren testikokonaisuuden suorittamista loppuun, ennen kuin saa palautteen koodistaan (Memon et al., 2017; Spragins, 2016). Laajemmat testikokonaisuudet voidaan suorittaa joka yö, tai kun kehityshaara sulautetaan päähaaraan.

Googlen koneoppimista hyödyntävä EPS-järjestelmä on Googlen sisältä tulevien kokemusten mukaan erittäin tehokas valtavien testikirjastojen priorisointiin (Memon et al., 2017; Spragins, 2016). Tällä hetkellä koneoppimisesta ohjelmistotestauksen apuna käyttävää tutkimusdataa on hyvin vähän Googlen ulkopuolella, mutta Memonin et al.

(2017) tutkimuksesta voidaan päätellä, että vastaavan järjestelmän kehittäminen ja käyttöönottaminen vaatii valtavasti resursseja, sekä selkeän infrastruktuurin ja ohjelmistokehitysprosessin, jotta oppivaan algoritmiin voidaan syöttää aineistoa.

5 Lähdeluettelo

Amland, Ståle. Risk-Based Testing: Risk Analysis Fundamentals and Metrics for Software Testing Including a Financial Application Case Study. *Journal of Systems and Software* 53, no. 3 (September 15, 2000): 287–95. https://doi.org/10.1016/S0164-1212(00)00019-4.

Berner, Stefan, Roland Weber, and Rudolf K. Keller. Observations and Lessons Learned from Automated Testing. In *Proceedings of the 27th International Conference on Software Engineering*, 571–579. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005. https://doi.org/10.1145/1062455.1062556.

Chittimalli, Pavan Kumar, and Mary Jean Harrold. Recomputing Coverage Information to Assist Regression Testing. *IEEE Transactions on Software Engineering* 35, no. 4 (July 2009): 452–69. https://doi.org/10.1109/TSE.2009.4.

Cohn, Mike. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Signature Series. Upper Saddle River (NJ): Addison-Wesley, 2010.

Engström, Emelie, Per Runeson, M. A. Babar, M. Vierimaa, and M. Oivo. A Qualitative Survey of Regression Testing Practices. In *Product-Focused Software Process Improvement/Lecture Notes In Computer Science*, 6156:3–16, 2010. https://doi.org/10.1007/978-3-642-13792-1_3.

Haghighatkhah, Alireza, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. Test Prioritization in Continuous Integration Environments. *Journal of Systems and Software* 146 (December 1, 2018): 80–98. https://doi.org/10.1016/j.jss.2018.08.061.

Hemmati, Hadi, Zhihan Fang, Mika V. Mäntylä, and Bram Adams. Prioritizing Manual Test Cases in Rapid Release Environments. *Software Testing, Verification and Reliability* 27, no. 6 (2017): e1609. https://doi.org/10.1002/stvr.1609.

International Software Testing Qualifications Board. (2019). Foundation Level Syllabus. https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html, haettu 19.3.2020.

Kasurinen, Jussi, Ossi Taipale, and Kari Smolander. Software Test Automation in Practice: Empirical Observations. Research Article. Advances in Software Engineering, 2010. https://doi.org/10.1155/2010/620836.

Ledru, Yves, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing Test Cases with String Distances. *Automated Software Engineering* 19, no. 1 (March 1, 2012): 65–95. https://doi.org/10.1007/s10515-011-0093-0.

Lehojärvi, Jaana. Tekninen velka – tunnusta, tunnista ja rajoita – Sytyke ry. Blog. *Sytyke* (blog), 2017. http://www.sytyke.org/tapetilla/tekninen-velka-tunnusta-tunnista-ja-rajoita/, haettu 31.3.2020.

Levenberg, Rachel Potvin, Josh. Why Google Stores Billions of Lines of Code in a Single Repository, 2016. https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext.

Marick, B. (1998, toukokuu). When Should a Test Be Automated? 11th Int'l Software/Internet Quality Week. San Fransisco, California, USA

Memon, Atif, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-Scale Continuous Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 233–42, 2017. https://doi.org/10.1109/ICSE-SEIP.2017.16.

Spragins, Peter. Efficacy Presubmit. *Google Testing Blog* (blog), 2016. https://testing.googleblog.com/2018/09/efficacy-presubmit.html.

Storoj, Oxana. Risk Based Software Testing. *Proceedings of International Conference on Application of Information and Communication Technology and Statistics in Economy and Education (ICAICTSEE)*, 2012, 182.

Strandberg, Per Erik, Daniel Sundmark, Wasif Afzal, Thomas J. Ostrand, and Elaine J. Weyuker. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 12–23, 2016. https://doi.org/10.1109/ISSRE.2016.23.

Stresnjak, S., & Hocenski, Z. (2011). Usage of Robot Framework in Automation of Functional Test Regression. ResearchGate. https://www.researchgate.net/publication/268185107_Usage_of_Robot_Framework_in_Automation_of_Functional_Test_Regression