Marcel Kautto

Testiautomaatio – miksi yritysten kannattaa automatisoida testauksensa

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Kautto Marcel

Työn nimi: Testiautomaatio – miksi yritysten kannattaa automatisoida testauksensa

Tutkintonimike: Insinööri (AMK)

Asiasanat: testiautomaatio, testaus, laadunvarmistus, tuottavuus, ohjelmistokehitys, ohjelmistotuotanto, Robot Framework, Bittium Wireless Oy

Kansilehden kuva: Bittium Oyj

Tämän opinnäytetyön toimeksiantajana toimi Bittium Wireless Oy, joka on Kajaanissa sijaitseva ohjelmistoalan yritys. Bittium Wireless Oy on osa Bittium Oyj:tä. Bittium Oyj kehittää pääasiassa taktisen kommunikaation, lääketeknologian sekä tietoturvallisen kommunikaation laitteita sekä palveluita.

Opinnäytetyön tavoitteena oli tuoda esille testiautomaation hyötyjä taloudelliselta, laadulliselta sekä käytännölliseltä kannalta. Hyötyjä lähdettiin etsimään teoriaan perustuen myös käytännön tapauksista yrityksistä, joissa testiautomaation avulla oli joko onnistuttu vähentämään kustannuksia, tai parantamaan tehdyn ohjelmiston laatua. Lopuksi teoreettisia mahdollisuuksia pystyttiin vertaamaan käytäntöön tutkimalla niiden esiintymistä opinnäytetyön tutkimuksen aikana laaditussa testiympäristössä.

Tutkimuksen aikana huomattiin, että testiautomaatiolla on kolme tärkeintä hyötyä ohjelmistotalolle. Testausautomaatiota voidaan käyttää uudelleen, eli sen käyttö on kustannustehokkaampaa pitkällä aikavälillä. Testausautomaatio pystyy varmistamaan ohjelmiston laatua tavoilla, jotka olisivat ihmisille joko mahdottomia tai liian tehottomia testata itse. Tämän lisäksi testausautomaatio mahdollistaa jatkuvan ohjelmistokehityksen, jossa ohjelmakoodia voidaan testata sekä kehittää sykleinä.

Teoreettisen hyödyn vertaaminen sekä viitattuun käytäntöön, mutta myös työtä varten luodun automaatioympäristön kehitys paljasti, että nämä hyödyt ovat aidosti havaittavissa myös aidossa kehitysympäristössä. Samalla havaittiin, että testiautomaation tehokkuus perustuu siihen, kuinka hyvin se on osattu toteuttaa. Saatu hyöty riippui vahvasti siitä, oliko automaatio osattu ottaa projektiin mukaan jo varhain. Samoin myös testaustiimin sekä kehittäjien kommunikaatio ja luottamus testiautomaation pystyi vaikuttamaan testausautomaation kannattavuuteen projektissa. Tutkittuun tietoon perustuen päätelmäksi jää, että testiautomaatio voi tuoda aitoa taloudellista etua ohjelmistokehityksessä, mikäli sen käyttöönotto on tietoinen päätös, jossa automaation haitat ja hyödyt on hyvin kartoitettu.

KAMK • University of Applied Sciences

Abstract

Author: Kautto Marcel

Title of the Publication: Test automation – why companies should automatize testing

Degree Title: Bachelor of Engineering, ICT Engineering

Keywords: test automation, testing, quality assurance, productivity, software development, software production, Robot Framework, Bittium Wireless Ltd.

Cover picture: Bittium PLC.

This thesis was commissioned by Bittium Wireless Ltd., which is a software development company based in Kajaani. Bittium Wireless Ltd. is part of Bittium PLC. Bittium PLC. mostly develops software and devices related to tactical communications, medical technologies, and secure communications.

The primary goal of this thesis was to indicate the economical, qualitative, and functional benefits of test automation. Based on theory, these benefits were found in practical examples of companies, where test automation had been successfully used to reduce costs, or to increase the quality of their software. Finally, in order to find out whether these benefits were present in reality, these theoretical opportunities were compared to a test automation environment that was built during the writing of this thesis.

The investigation revealed that test automation has three primary benefits to a software company. It may be reused, meaning that, in the long run, utilizing test automation is more cost-effective. Test automation can additionally ensure the quality of the software in ways that would, for a human, be either impossible or not productive. Lastly, test automation enables the continuous development and testing of software. The source code may then be tested and developed in cycles.

The comparison between theory and cited practical examples, as well as the creation of an automated test environment, revealed that these benefits genuinely exist in real development environments. It was also noticed that the effectiveness of test automation is based how well it is implemented. The benefits, in turn, were heavily reliant on the speed at which it is adapted into the project, with the greatest benefits seen in early adoption. The communication between the development and testing teams, as well as the trust put in the automation by these parties was also seen as a major factor affecting the cost-effectiveness of test automation in a project. Based on the research, the conclusion is that test automation is capable of bringing genuine economical value in software development, provided that the decision to turn to automated testing is based on well researched knowledge on its flaws and strengths.

Alkusanat

Haluan kiittää Bittium Oyj:tä mielenkiintoisesta opinnäytetyöaiheesta, jonka monipuolisuus siivitti opinnäytetyön kirjoitusta.

Haluan myös omistaa pienen osan tätä opinnäytetyötä kiittääkseni Kajaanin Ammattikorkeakoulun opettajakuntaa sekä kanssaopiskelijoita, joiden tekniikan osaaminen sekä työmotivaatio auttoi jaksamaan tähän pisteeseen asti. Routaisella tiellä kulku ei aina ole ollut helppoa, mutta tärkeintä on se, että läpi on päästy.

Viimeiseksi haluan eritoten kiittää Eero Soinista sekä Eero Huuskoa opinnäytetyön tarkastustyöstä.

Sisällys

Symboliluettelo

ABS-järjestelmä	*Anti-lock brake system*. Estää pyöriä lukittumasta voimakkaan jarrutuksen aikana.
Diskontattu kassavirta	Tulevaisuudessa saatavien kassavirtojen arvo nykyhetkellä
IEEEE-ammattiliitto	*Institute of Electrical and Electronics Engineers* Kansainvälinen tekniikan alan järjestö.
Integraatiotesti	Ohjelman osien välisen kommunikaation testaus
Iteratiivinen	Toistuva. Tässä yhteydessä ohjelmistoa kehitetään jatkuvasti, eli kehitys on toistuvaa
Jatkuva kehitys ja integraatio	*engl. continuous development and integration* Ohjelmistokehityksen menetelmä, jossa usean eri tekijän luoma lähdekoodi yhdistetään osaksi yhtä ohjelmistoa
Julkaisutestaus	Yhden sovellusversion julkaisun yhteydessä tapahtuva testaus
Keskusteluhuone	*engl. chat* Verkossa toimiva palvelu, jossa käyttäjät voivat keskustella keskenään
Ketterä ohjelmistokehitys	*engl. agile software development* Joukko kehitysmenetelmiä, joille on yleistä toimivan ohjelmiston sekä suoran viestinnän tärkeys
Käyttötapaus	Luettelo toimenpiteistä, joka määrittelee, kuinka käyttäjä kommunikoi ohjelmiston kanssa

Lavasteympäristö	*engl. staging environment* Identtinen kopio koko ohjelmistosta, jolle voidaan suorittaa testejä
Lohkeileva (testi)	*engl. flaky test* Testi, joka menee läpi vain osan ajasta, myös silloin, kun sen ei pitäisi epäonnistua
Ohjelmistokehityksen v-malli	Prosessimalli, jota käytetään projektin suunnittelussa ja toteutuksessa
Ohjelmiston elämänkaari	Kehityksen sekä ylläpidon vaiheiden muodostama kuvaus ohjelmiston alusta sen loppuun
Ohjelmiston vaativuusmäärittely	Dokumentti, joka kuvaa projektin tavoitteita sekä vaatimuksia
Paraabelimainen	*engl. parabolic* Paraabelin muotoa muistuttava kehitys, eli ajan kanssa kiihtyvä
Regressio	Ohjelmiston tason heikkeneminen
Testauspäällikkö	*engl. test manager/test lead* Testaustiimin johtaja
Yksikkö- tai komponenttitesti	Testi, jossa testataan ohjelmistokoodia koko järjestelmästä erillään

1 Johdanto

Ohjelmistotestaus on suuri osa jokaisen ohjelmointialan yrityksen toimintaa. Hyvin suoritetulla testauksella voidaan vähentää kustannuksia kehitysvaiheessa, mutta myös sen jälkeen. Testausta on jo pitkän aika suoritettu manuaalisesti, keskittämällä käytännön testaukseen osaavaa henkilöstöä, joiden pääasiallinen tehtävä on löytää puutteita sekä vikoja ohjelmistopalvelusta tai -tuotteesta. Testausstrategioita ja vaatimuksia on niin paljoin kuin projektejakin – testauksessa on aina tärkeintä huomioida kyseisen projektin vaatimukset sekä kehitystavat. Vaikka tämä perinteinen testauksen linja on vieläkin tärkeä, on ala kehittynyt, ja uusia testaustapoja sekä strategioita on alkanut ilmestymään. Näillä strategioilla sekä tavoilla pyritään vähentämään kustannuksia, löytämään useampia virheitä, paikkaamaan virheitä nopeammin ja pitämään huolta laatukehityksestä. Hyvä testaus ei siis ole enää vain sitä, että ongelmia löydetään ennen tuotteen lähettämistä asiakkaalle. Siitä on tullut monimutkainen sarja erilaista laadunvarmennusta, jossa useita eri toimenpiteitä saatetaan tehdä samanaikaisesti. Testauksen haastavuuden kasvaessa myös uusia testausmetodeja tarvitaan.

Yksi näistä uusista testausmetodeista on automaattinen testaus. Automaattisessa testauksessa käytetään tietokoneella pyörivää skriptiä eli koodipätkää. Tietokone tarkistaa automaattisesti testaajan määrittelemiä tapahtumia sekä painalluksia, ja vertaa lopputulosta testaajan olettamukseen. Automaattinen testaus on vain yksi tapa vastata uusiin haasteisiin, mutta sellaisen kehittäminen ohjelmointiprojektiin voi tuoda valtavasti lisäarvoa yritykselle. Tässä tullaan keskittymään lähinnä automaattisen testauksen tapoihin parantaa laadunvarmennusta, mutta myös niitä tapoja, joilla tällainen testausstrategia voidaan ottaa oikeaoppisesti käyttöön ohjelmointiprojektissa. Käytännön esimerkeistä sekä muista yrityksistä voidaan ottaa oppia siitä, miten tästä uudesta testausstrategiasta saadaan eniten irti.

Tämä työ tehtiin perustuen työharjoitteluuni Bittium Wireless Oy:llä, jossa pääsin tutustumaan ensimmäistä kertaa testiautomaatioon. Työharjoittelun aikana pääsin toteuttamaan automaattiset testit alusta alkaen verkossa toimivalle mittausanalyysipalvelulle. Työn toteutuksessa auttoivat jo valmiiksi luodut manuaalitestitapaukset, mutta suureksi osin testitapausten rakennuksen suunnittelu oli tekijän vastuulla. Käytännön kokemus testiautomaatioympäristön pystytyksestä tuodaan myöhemmin esille vertailemalla erilaisia toimintatapoja oman, koetun tavan kanssa.

Työssä käydään läpi sovellustestauksen tärkeys, se, miksi sovelluksia ylipäänsä tulee testata. Tämän jälkeen käydään läpi niitä eri testaustapoja sekä testaussuunnitelmia, joita noudattamalla

testauksesta on tullut nopeampaa vuosien saatossa. Manuaalitestausta käsitellään osana näitä testausstrategioita. Samalla yritetään löytää niitä ongelmakohtia, joissa manuaalinen testaus vaatii automaattisen testauksen apua. Myöhemmässä vaiheessa käsitellään sitä, kuinka molemmat näistä testaustyypeistä toimivat yhdessä, mahdollistaen ketterän ohjelmistokehityksen. Automaatiotestausta käydään läpi sekä käytännön että teorian näkökannasta, miettien sen paikkaa yrityksen testausstrategiassa. Työn loppupuolella käydään läpi automaation kustannuksia, sekä sen tuomaa arvoa testaukseen. Arvon lisäystä käsitellään myös suoraan kassavirran tasolla. Lopuksi käydään läpi tapausesimerkki, joka tätä opinnäytetyötä vasten oli luotu. Tapausesimerkistä nähdään suoria viittauksia itse testauksen teoriaan, sekä käytännössä saatuja hyötyjä projektin aikana.

Yhdistämällä automaation teoreettisia hyötyjä sen luomisesta löytyvien hyötyjen kanssa, voimme havaita, kuinka suuri vaikutus automaation luomalla turvaverkolla sekä eri manuaalisen testauksen avuksi luoduilla työkaluilla on testauksen laatuun. Testausta miettiessä on hyvä lähteä ensimmäiseksi miettimään, miksi ylipäänsä testaamme sovelluksia. Testauksen tärkeyden lisääntyessä sekä vanhojen testausmetodien muuttuessa liian hitaiksi ymmärrämme paremmin, miksi automaattiset testit ovat hyvä työkalu, jota hyödyntämällä yritys voi kehittää testauskapasiteettiaan.

2 Ohjelmistotestaus yrityksissä

Vuonna 1999 Marsia kiertävä avaruusluotain on laskeutumassa Marsiin. Yhtäkkiä avaruusluotain laskeutuu 53 kilometriä lähemmäksi pintaa kuin pitäisi. Signaali menetetään, ja alusta ei enää löydetä. 193 miljoonan dollarin kehitystyö sekä yhdeksän kuukauden odotus on menetetty. Miksi? Siksi, että toinen kehitystiimeistä käytti metrijärjestelmää ja toinen kehitystiimi brittiläistä mittajärjestelmää. [1.]

Vuonna 2010 autonvalmistaja Toyota joutuu vetämään takaisin yli yhdeksän miljoonaa autoa maailmanlaajuisesti, aiheuttaen itselleen arvioiden mukaan yli kolmen miljardin dollarin kustannukset. Sovellusvirheen vuoksi auton ABS-järjestelmässä esiintyi viivettä, joka johti kolareihin. Toisin kuin aikaisemman esimerkin avaruusluotaimessa, Toyota ei selvinnyt vain lommoilla talouteen ja kunniaansa, vaan ohjelmointivirhe johti myös ihmishenkien menetykseen. [2.] Mitä yhteistä näillä tilanteilla on? Molemmat niistä ovat johtaneet suuriin, erimuotoisiin vaurioihin, ja ne ovat jääneet pysyvästi näiden yritysten historiaan. Molemmat niistä olisivat luultavasti nykyisillä testausmetodeilla sekä tarkkaavaisuudella jääneet kiinni testeissä, ilman että niistä olisi ikinä syntynyt vastaavaa jälkeä.

Yleinen sanonta ohjelmistoalalla on ”Onnistuneen ohjelmiston takana on aina huolellinen testiinsinööri” [3]. Testaustyötä voidaan pitää eräänlaisena muurina asiakkaan sekä yrityksen välillä. Huolellinen testaus ja laadunvarmennus on yritykselle yleisesti erittäin arvokasta. Sen avulla voidaan korjata ongelmia halvemmalla, nopeuttaa ohjelmistokehitystä, huomata ongelmia ennen kuin ne vaikuttavat yrityksen julkikuvaan sekä parantaa turvallisuutta. [4.] Testaustyö on kuitenkin monimutkaistunut samalla, kun projektien sekä tiimien koko on kasvanut. Käyttöön on otettu uusia testausmetodeja sekä tyylejä, joilla yritetään vastata alati kasvavaan haasteellisuuteen ohjelmistoprojekteissa. Luomalla yksityiskohtaisia rooleja voidaan yksilöiden tietotaitoa tietyn testauksen hoidosta käyttää hyväksi, mikäli yrityksen resurssit sen sallivat. Toisaalta projektin koosta riippuen voi myös olla, että yksi testaustiimi tai jopa yksilöllinen testaaja joutuu vastaamaan koko projektin tai yrityksen testauksesta. Jokainen ohjelmistoprojekti ja yritys on ainut laatuaan, ja arvostaa testausta eri tavalla. Kokenut ja ammattitaitoinen testaaja saattaa kyetä luonnollisesti tarkempaan jälkeen sekä useampaan tehtävään projektissa, mutta toisaalta on tärkeä löytää tasapaino testaajamäärän sekä projektien vaatimusten välillä. Tärkeää on kuitenkin aina osata saada irti testauksesta niin paljon kuin mahdollista, sillä sen alentava vaikutus kustannuksiin ja ylentävä vaikutus yrityksen maineeseen on projektin loppuessa, sekä sen aikana, huomattava.

Luvun alun esimerkit ovat kauhuskenaarioita siitä, kun kaikki menee ohjelmoinnissa pieleen. Silti ne eivät ole vielä eivätkä tule pitkään aikaan olemaan historiaa. Kun projektin määräajat alkavat välkkyä kauhistuneiden ohjelmoijien silmissä ja projektin kustannukset hipovat liian kauaksi rajoja, yritykset saattavat alkaa tekemään kyseenalaistettavia päätöksiä. Nämä päätökset vielä nykyaikanakin aiheuttavat onnettomuuksia ja suuria vaurioita, siitä uusimpana tärkeänä esimerkkinä Boeing 737 -lentokonemallin surmansyöksy Etiopiassa, jonka inhimillinen hinta näkyy kuvassa 1 [5]. Onnettomuus aiheutui ennen kaikkea kustannusvähennyssyistä, mutta myös virheellisestä ajatuksenkulusta kehitysvaiheessa, jossa todellisuus ei vastannut ajateltua tilannetta. Surullista juuri tässä tapauksessa on myös se, että vaikka yritys tiesi viasta, se päätti korjata vian vasta myöhemmin. [6.]

Kuva 1. Boeing 737 -lento-onnettomuuden turmapaikka Etiopiassa [5]

On siis tärkeää yrityksille huolimatta aika- sekä kustannusrajoitteista aina pystyä toimimaan vastuullisesti. Hyvin tuotettu ohjelmistokoodi pystyy parantamaan yritysten julkikuvaa, mutta myös luomaan positiivisia käyttäjäkokemuksia, joilla on vahva positiivinen vaikutus yrityksen kassavirtaan. Hyvä testaus saattaa parhaimmillaan auttaa yritystä välttämään täyskatastrofin, mutta suurimmissa tapauksissa jälki on näkymättömämpää. Tärkeää on ymmärtää, että tämä on hyvä merkki. Mikäli ohjelmistosta ei valiteta, se usein tarkoittaa, että laatu on odotettua. Tämä laatu syntyy yhteistyöstä. Seuraavaksi kuvataan sitä, miten yhteistyö yrityksen sisällä eri testaustiimin jäsenten, mutta myös ohjelmointitiimin jäsenten välillä voi toimia.

2.1 Ohjelmistotestauksen roolit

Antamalla testaustiimille erilaisia rooleja yritys voi varmistaa, että sovelluksen testaukselle on hyvät edellytykset. Testaustiimin sisällä yksittäisillä testaajilla saattaa olla hyvin erilaista osaamista ohjelmistotestaamisesta. On tärkeää, että testitiimillä on aina johtaja. Testauspäällikön tehtävänä on usein määritellä muille testaajille sekä testi-insinööreille heidän tehtävänsä tiimin sisällä. Usein he tekevät testisuunnitelman, jonka mukaan projektin testaus etenee. Testaustiimin resurssien määritteleminen sekä hallinta on usein heidän hartioillaan, samoin kuin testauksen tilanteen selvittäminen sekä ylläpito. He tekevät raportteja testauksen tilanteesta suoraan projektipäälliköille tai muille ylemmille toimihenkilöille. [7.] Todella suurissa ohjelmistoprojekteissa heillä saattaa olla myös alempia testipäällikköjä testauksen eri osa-alueille. Testauspäällikön tehtävä on testitiimissä tärkeä: he koordinoivat koko ohjelmointitestausta ja pitävät huolta siitä, että toiminta on kannattavaa sekä tehokasta. Hyvä testauspäällikkö ei anna ajan eikä rahallisten resurssien valua hukkaan suunnittelemalla testit niin, että niistä saatava hyöty on aitoa projektin kannalta. Nykyaikaiset ohjelmistoprojektit ovat testauksen kannalta haastavia. Usein niissä on tarve jatkuvalle kehitykselle, jolloin testauspäällikön on pystyttävä tekemään hyvä suunnitelma jatkuvan kehityksen testaukseen niin, että uusia testaamattomia ominaisuuksia, tai päivityksiä, ei mene läpi asiakkaalle asti.

Testauspäällikön alla usein toimii suoraan eri rooleja suorittavia testaajia. Koska eri ohjelmistoprojektit sisältävät erilaisia vaatimuksia testaukseen, voi olla, että yksittäinen testaaja joutuu käymään läpi hyvinkin erilaisia tehtäviä. Yksittäisten testaajien sekä ohjelmistotestauksen vaiheisiin tullaan tutustumaan syvemmin luvussa 3. Yleisesti testaajien tehtäviin kuitenkin kuuluu tehdyn testaussuunnitelman läpikäynti sekä noudattaminen, testauksen suoritus sekä kommunikaatio testauksen tilasta testauspäällikön kanssa. Testaajan tehtävänä on myös ilmoittaa testitiimin johtajalle sellaisista esteistä, jotka hän huomaa ennen testausta tai testauksen yhteydessä. [7.] Tällaisen hyvän kommunikaation ylläpito ei kuitenkaan välttämättä ole aina helppoa ja vaatii toimenpiteitä testitiimiä organisoitaessa.

2.2 Testaustiimin kommunikaatio

Yksi tärkeistä työkaluista onnistuneelle ohjelmistotestaamiselle on kommunikaatio testaustiimin jäsenten välillä. Kommunikaatio testauspäällikön sekä itse testaajien välillä on kriittisen tärkeää.

Jotta tiimiä pystyttäisiin johtamaan oikein, hänen tai heidän täytyy tietää, missä tilassa eri ominaisuuksien testaus on. Kommunikaatio ei kuitenkaan ole aina helppoa, eikä hyvä testausilmapiiri rakennu itsestään. Käyttämällä tehokkaita testaus- sekä työtapoja ja rakentamalla avoimen ilmapiirin voi testaustiimi helpommin välittää ideoita sekä ajatuksia toisilleen. Kun ongelmista sekä parannusehdotuksista on helppo puhua, välittyvät ajatukset nopeammin ja eri testaustiimin jäsenten sekä johdon on helpompi saada tietoa testauksen tilanteesta.

Tapoja parantaa testaustiimin välistä kommunikaatiota on monia, mutta varmasti tärkeimpänä on pitää huomio tiimin koossa. Hyvän kokoinen tiimi ei ole liian pieni, mutta toisaalta ei myöskään liian suuri. Liian pienellä tiimillä voi mahdollisesti olla ongelmia suorittaa tarvittavaa testausta siinä ajassa, jonka projekti vaatii. Testauspaineiden kasvu näkyy välttämättä jossain vaiheessa kommunikaation heikkenemisenä. Suurten tiimien ongelma taas on usein se, että resursseja käytetään hukkaan, ja koko tiimin pitäminen ajan tasalla testauksen tilanteesta on vaikeaa, ellei mahdotonta. Työaikaa palaverien muodossa joudutaan käyttämään turhaan testauspäällikön, mutta myös muiden testaajien osalta, mikäli testauksen nykytilanteesta ei olla ajan tasalla. Samalla kannattaa keskittää huomio myös niihin testaajiin, jotka tiimissä ovat. Testaajat, jotka kaikki ovat erikoistuneet samoihin asioihin, eivät välttämättä pysty yhdessä vastaamaan modernin sovelluskehityksen haasteisiin ainakaan niin nopeasti kuin hyvin tasapainotettu testausryhmä. Yrityksen tärkeä tehtävä on pitää huolta siitä, että uudet jäsenet oppivat perustaitoja pikaisesti, mutta samalla löytävät itselleen oman tärkeän roolin testaustiimissä. Monimuotoinen, mutta keskikokoinen tiimi pitää huolta siitä, että jokainen jäsen tuo testaukseen aina parhaansa.

Kun testitiimin koko sekä osaamisen monimuotoisuus on mietitty hyvin, on jokaisella testitiimin jäsenellä aina hyvä kokemus omasta osaamisalueestaan testauspalavereissa. Palaverit ovat johtamisessa tärkeä työkalu, jossa samalla voidaan päivittää tämänhetkinen tilanne testaustiimille, mutta myöskin saada uusinta tietoa vuorotellen jokaiselta testaajalta. Tällöin, jos jollain testaajalla on eriävää tietoa, voidaan asia samalla miettiä läpi kaikkien kesken. Palaverit toimivat hyvänä lähtökohtana sekä tiimin että johtajan välille, mutta niissä voidaan, ja kannattaakin, olla testaajien sisäistä keskustelua. Hyvin suunnitelluissa palavereissa on aina tarpeeksi aikaa käydä läpi uutiset, mutta myös ottaa huomioon kaikkien testaustiimin jäsenten huomiot testauksesta. Tämän vuoksi niihin on kannattavaa panostaa aikaa. Palaverit eivät kuitenkaan ole ainut paikka koko testitiimin väliselle keskustelulle, vaan myös keskusteluhuone voi olla hyvä tapa pitää koko tiimi ajan tasalla. Keskusteluhuoneiden etu palavereihin on sen vapaamuotoisuus. Asioita on helppo ja nopea kysyä koko tiimin jäseniltä, jotka voivat vastata, mikäli siinä tilanteessa kerkeävät. Mielipiteitä sekä kehitysehdotuksia on näin helpompi käyttää läpi useammalla henkilöllä. Keskusteluhuoneista on

kuitenkin tärkeä huomata, että niiden käyttö voi vapaamuotoisuutensa vuoksi lähteä helposti kiertämään asian vierestä. Hyvä pelisääntö on, että jos keskustelu liittyy ja vaikuttaa vain yhteen vastaanottajaan, ei silloin kannata käyttää kaikkien aikaa keskusteluhuoneessa asian miettimiseen. Oikein toteutettuna nämä keskustelut voivat tehokkaasti vähentää sitä aikaa, mitä yksittäisellä testaajalla menisi joko odottaa palavereihin, tai välittää tieto johtajan kautta koko tiimille. Ne ovat hyödyllinen työkalu, jota kannattaa aidosti harkita.

Testitiimin kommunikaatio on sujuvaa, kun se edesauttaa tiimin mahdollisuutta selviytyä kaikista julkaisun tai uuden ominaisuuden testauksen tavoitteista ajallaan. Hyvä kommunikaatio edesauttaa testauksen kattavuuden lisäämistä, mutta samalla myös mahdollistaa tiimin jäsenten kommunikaation toistensa kanssa. Tällaiset vuorovaikutukset voivat synnyttää kehitysajatuksia sekä parannuksia ohjelmiston toimintaan, jotka voivat puolestaan antaa uutta suuntaa projektille. Terveellinen keskustelukulttuuri voi siis edesauttaa yrityksen tuottavuutta myös testauksen näkökulmasta. Samoin, jos testaustiimin sekä kehitystiimin välinen keskusteluyhteys on kunnossa, pystyvät testaajat tekemään parempaa yhteistyötä kehittäjien kanssa. Tämä näkyy esimerkiksi testauksen sekä kehityksen välisenä kommunikaationa, joka parhaimmillaan voi auttaa vähentämään työmäärää molemmilla puolella. Testaus oppii paremmin, mistä ja minkälaisia virheitä kehityksessä on voinut päästä syntymään, jolloin ongelmia ja kehityskohtia löydetään paremmin. Samalla kehittäjät osaavat käyttää aikaansa paremmin, kun heidän virheensä löydetään aikaisessa vaiheessa. Toisaalta myös on yrityksen kannalta hyödyllistä, että testaustiimin palaute kehityksen laatuun voi parhaimmillaan olla hyödyllistä ohjelmistokehittäjän ammattitaidolle.

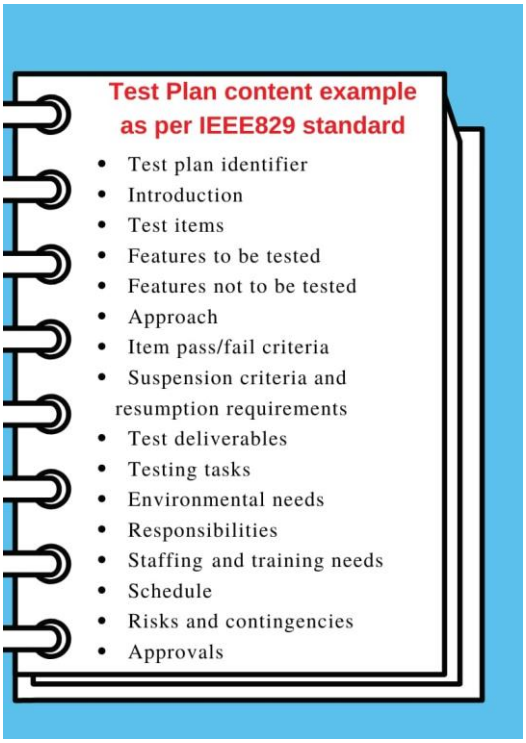
Hyvä kommunikaatio on elinehto tehokkaalle toiminnalle testaustiimissä, ja sen vaikutukset kaikkiin kehityksen osa-alueisiin ovat suuret. Hyvä kommunikaatio on vain yksi monista tavoista, joilla testaustiimin tuottavuutta, ja samalla muiden tiimien tuottavuutta, voidaan parantaa. Sen monimuotoisen vaikutuksen vuoksi sen toimivuuteen yrityksen kannattaa kuitenkin panostaa. Kommunikaation lähtökohta on kuitenkin se, että testaustiimi tietää, mitä testataan. Projektin dokumentaatio on suuri osa kommunikaation toimivuutta. Jotta projektin jäsenet voisivat keskustella sekä toimia yhdessä testatessa, on tiedettävä, mitä testauksella haetaan.

2.3 Ohjelmistotestauksen dokumentaatio

Ohjelmistoprojektit ovat usein valtavia kokonaisuuksia, joissa kokonaisuus on monen pienen osan summa. Tämän kokonaisuuden hahmottaminen on usein testauspäällikön tehtävä, mutta testaustiimin jäsenten on hyvin hankala lähteä testaamaan tuotetta, jonka toiminnallisuudesta he eivät tiedä. Epäselvästi määritellyn tuotteen tai sovelluksen testaus voi jäädä hyvin pinnalliseksi, jolloin ohjelmaan voi helposti jäädä virheitä sekä epäkohtia. Testauspäällikön kyky tuoda esille testauksen kannalta olennaiset tiedot projektista on siis kriittinen osa testauksen onnistumista. Jokaisessa projektissa on erilainen kynnys dokumentaatiolle. Itse dokumentaation luominen vie aikaa, mutta samoin kuin muulla kommunikaatiolla, sillä voidaan säästää paljon aikaa ja resursseja myöhemmissä projektin vaiheissa. Tämän kaltaisia testauksen kannalta tärkeitä dokumentteja on monia, mutta tulemme tässä käymään läpi pari niistä olennaisinta.

Ohjelmiston vaativuusmäärittelyssä kuvataan koko projektin tavoitteita sekä vaatimuksia. Tämä dokumentti on tärkeä, sillä siinä usein käydään läpi projektin toiminnallisuus käyttötapausten avulla. Yhden käyttötapauksen tarkoitus on osoittaa, kuinka järjestelmää käytettäisiin yhdessä tapauksessa. Monimutkainen järjestelmä koostuu siis useasta käyttötapauksesta. [8.] Tämän lisäksi vaativuusmäärittely usein jaetaan kahteen osioon, toiminnallisiin- ja ei-toiminnallisiin vaatimuksiin. Toiminalliset vaatimukset sisältävät tietoa siitä, mitä ohjelman kuuluu tehdä, ja kuinka eri syötteet tuottavat käyttäjälle erilaisen tulosteen tai lopputuloksen. Ei-toiminnallisiini vaatimuksiin kuuluvat resurssivaatimukset sekä laatuvaatimukset. Resurssivaatimukset määrittävät sen, paljonko asiakas käyttää projektiin rahaa. Laatuvaatimukset puolestaan määrittelevät, kuinka ohjelmisto vastaa sille esitettyihin vaatimuksiin. Laatuvaatimuksia ovat siis esimerkiksi ohjelman tietoturvallisuus tai suorituskyky. [9.] Käyttötapaukset antavat siis pohjan koko vaativuusmäärittelylle, joten niiden miettiminen myös testatessa voi olla tiimille todella hyödyllistä. Erilaisten käyttötapausten kautta voidaan yrittää lähteä asiakkaan näkökulmasta etsimään ongelmia ohjelmistosta. Käyttötapausten tunteminen on myös tärkeää siksi, että tapausten tunteminen auttaa testaustiimiä priorisoimaan niitä ongelmia, jotka he löytävät. Ongelmat, jotka aiheuttavat nopeasti tai paljon haittaa asiakkaalle heidän haluamissaan käyttötarkoituksissa ovat tärkeämpiä korjata, kuin sellaiset haitat, johon asiakas ei törmää. Hyvin kommunikoitu vaativuusmäärittely auttaa myös testaustiimin ulkopuolisia työntekijöitä tekemään parempia päätöksiä ohjelmaa valmistaessaan, jolloin virheiden väheneminen näkyy positiivisesti testaustiimin työssä.

Vaativuusmäärittelyn ollessa kunnossa on hyvä tehdä projektista testaussuunnitelma**,** jossa käydään läpi esimerkiksi testauksen kattavuutta, työtapoja sekä testausvastuuta. Kattavuutta mietittäessä käydään läpi koko ohjelmiston elämänkaari, jonka ajalta määritellään ne vaatimukset, joita tullaan testaamaan. Samalla määritellään ne testaustyypit, joita vaatimuksen testauksen aiotaan käyttää. Tässä voidaan myös miettiä niitä ominaisuuksia, joita tullaan testaamaan, ja niitä, jotka tullaan jättämään testauksen ulkopuolelle esimerkiksi kustannussyistä. [10.] Työtapoja puolestaan kuvataan kertomalla, miten testikattavuus saavutetaan. Testaustyötapoja voidaan esimerkiksi kuvata standardien kautta. Työtapojen yhteydessä dokumentissa usein määritellään myös testauksessa tarvittavat laitteet sekä määritellään ne kriteerit, joiden mukaan testitapaukset määritellään joko läpäistyiksi tai hylätyiksi. [10.] Viimeisenä dokumentissa määritetään testausvelvollisuus. Tässä osiossa määritellään ne organisaatiot, jotka ovat vastuussa määritellyistä testauksen vaiheessa tietyssä osaa ohjelmiston elämänkaarta. Näin toimimalla, nämä organisaatiot tai ryhmät saavat etukäteen mahdollisuuden tarkentaa vaatimuksia, tai hankkia tarvittavat työkalut heille annetun testauksen suorittamiseen. [10.] Kuvassa 2 esitetään IEEEE-ammattiliiton luoman standardin mukainen esimerkki testisuunnitelman sisällöstä [11].



Kuva 2. Esimerkki testisuunnitelmasta IEEE829-standardin mukaisesti [11]

Yritysmaailman projektit ovat kuitenkin hyvin erilaisia toisistaan. Joissain projekteissa kaikille IEEE829-standardin testisuunnitelman sisällölle ei välttämättä ole tarvetta. Isoissa projekteissa voi olla, että jokaiseen eri testaukseen osa-alueeseen joudutaan tekemään oma suunnitelma,

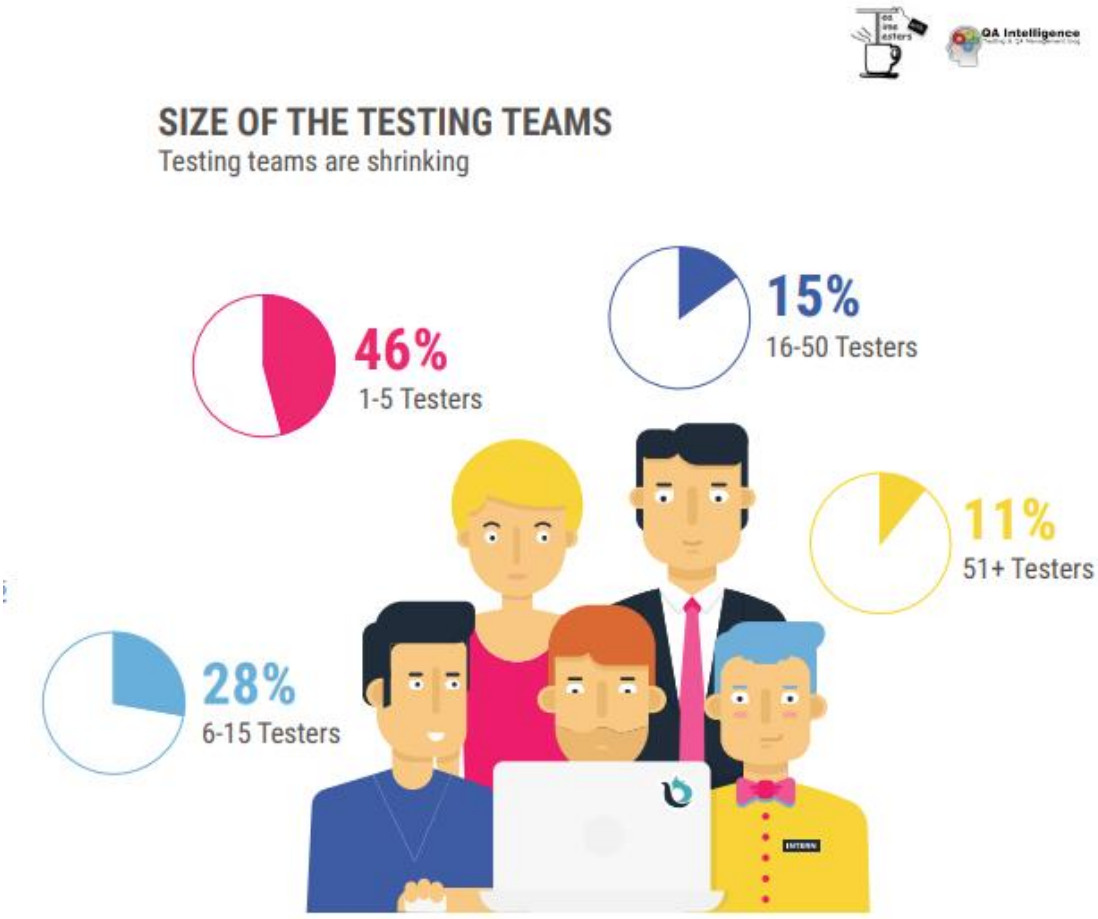
joka seuraa testistrategiaa**.** Pienemmissä projekteissa strategia saatetaan sulauttaa osaksi testisuunnitelmaa, mutta suurissa projekteissa, joissa eri osa-alueille tehdään omat suunnitelmat, on hyvä olla kaiken testaustoiminnan määrittelevä testausstrategiadokumentti. Siinä missä testisuunnitelmia on useampia, ja niitä voidaan päivittää, strategiadokumentti on usein staattinen ja sisältää usein esimerkiksi yrityksen näkökulmia testaukseen, sekä ne todisteet ja testidokumentaatio, jota testaukselta odotetaan. Siinä voidaan myös määritellä ne periaatteet, joita testauksessa kuuluu seurata. [12.] Testausstrategiadokumentin voi myös valmistaa erikseen testisuunnitelmasta, mikäli katsotaan, että siihen kuuluvia osia ei saada osaksi testisuunnitelmaa. Strategiadokumentti auttaa testausta pysymään yrityksen linjausten mukaisena.

Näiden perusdokumenttien tarkoitus on pitää huolta siitä, että testaus käytännössä noudattaa yrityksen antamia ohjeita, on aidosti hyödyllistä ohjelmiston vaatimusten näkökulmasta sekä edesauttaa testauksen menestymistä. Tietämällä etukäteen ohjelmistotuotteen vaatimukset, tarvittavat työkalut sekä testausmetodit, testaustiimi pääsee nopeasti testaukseen kiinni testausvaiheen alkaessa. Yhteiset dokumentit toimivat pelisääntöinä, joiden avulla testaustiimin laatu pysyy tasaisena. Standardoidusti tehdyt testitapaukset edesauttavat niiden luettavuutta ja toistettavuutta. Samalla projektiin jää dokumentaatiota mahdollisesti projektiin myöhemmin uusille tai vaihtuville testaajille. Epäilykset esimerkiksi testauksen tai ominaisuuden tilasta voidaan selvittää helpommin, kun testauksesta sekä vaatimuksista on tarkat jäljet saatavilla julkisesti. Hyvä dokumentaatio toisin sanoen parantaa saatavaa tietoa kaikissa eri projektin tilanteissa. Huono dokumentaatio sen sijaan poikii enemmän kysymyksiä projektin tilanteesta sekä mahdollisesti tuhlaa kaikkien projektin työntekijöiden aikaa asioita selvitellessä. Dokumentaatio on tärkeä osa työntekoa, josta on hyvä pitää huolta heti alusta alkaen. Sen ylläpitämiselle voidaan jopa antaa aikaa erikseen myöhemmin projektissa, mikäli dokumentaatio näyttää olevan projektia huomattavasti jäljessä. Vaikka dokumentaation kirjoitukseen menee aikaa, siitä hyötyy usea osapuoli useassa eri tilanteessa, jolloin saatu hyöty on helpompi realisoida**.**

Ohjelmistoprojekteissa on siis kehitetty uusia toimintamalleja sekä testausstrategioita, jotka ovat antaneet niiden monimutkaistua aikojen saatossa. Isompien tiimien ja vaatimusten nojalla uudet suunnitelmalliset työtavat ovat auttaneet suurten kokonaisuuksien hallinnassa. Seuraavaksi tullaan käsittelemään projektin testausta käytännössä testitiimin näkökulmasta. Ohjelmistoprojektin testaus käytännössä sisältää useita työvaiheita, jotka voivat helposti tehdä lommon projektin resursseihin. Oikeanlaisten työ- ja testaustapojen ylläpito voi vähentää näitä kustannuksia. Projektin työkuorman sekä vanhojen testausmenetelmien käytön tarkastelu auttaa huomaamaan automaation hyödyt käytännössä.

3 Ohjelmiston kehitys ja testaus

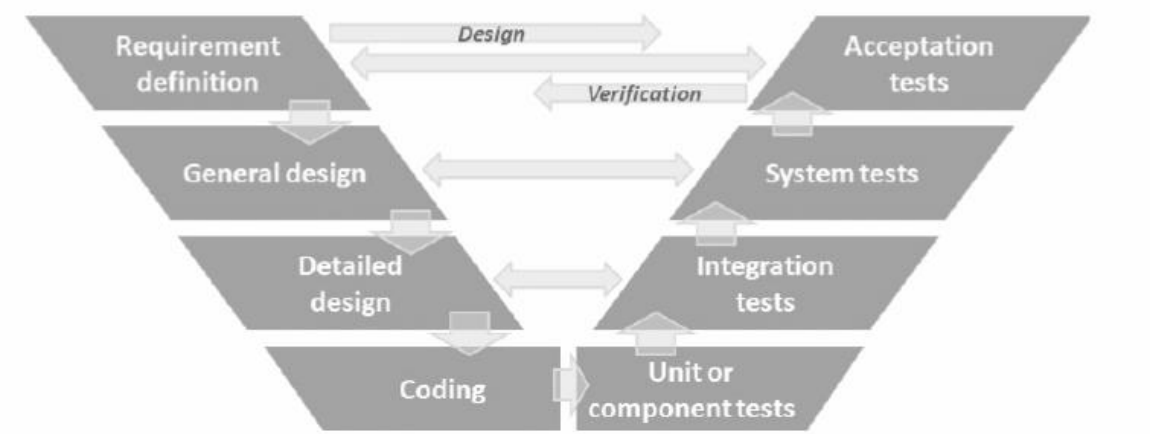
Ohjelmistotestauksessa pyritään löytämään vikoja, jotka aiheutuvat useammasta eri koodipätkästä tai komponentista. Näiden vikojen löytäminen jatkuvasti kehittyvästä sovelluksesta on lähtökohtaisesti hankalaa, sillä näiden yksittäisten osien testaamiseen ei projektilta luultavasti riitä aikaa. Yrityksissä on tästä huolimatta alettu siirtymään pienempiin testaustiimeihin, joka näkyy kuvassa 3. Ohjelmistotestaajille teetetyssä kyselyssä, jonka suoritti PractiTest & Tea-Time With Testers todetaan, että vuonna 2016 alle 15 testaajan tiimeissä työskenteli 60 % kyselyyn vastanneista, siinä missä vuonna 2018 tämä määrä oli 74 %. [13, s. 9]



Kuva 3. Testaustiimien koko kyselyihin vastanneista ohjelmistotestaajista vuonna 2018 [13, s. 9]

Trendi ei näytä alkuun seuraavan jatkuvasti monimutkaistuvien ohjelmistoprojektien vaatimuksia, mutta ymmärtämällä nykyisen ohjelmistotestauksen testaustapoja voidaan ymmärtää, kuinka testaustiimit pystyvät vastaamaan kasvavaan haasteeseen.

Vuonna 1979 amerikkalainen ohjelmistosuunnittelija Barry Boehm ehdotti kehitys- ja testausvaiheiden mallintamiseen ohjelmistokehityksen v-mallia**.** V-malli yhdistää sovelluskehityksen eri vaiheet niiden vastaaviin testausvaiheisiin. Mallin ajatuksena on esittää vaiheet niin, että tiettyjä suunnitteluvaiheita (vasemmalla) voidaan testata vain tiettyyn testivaiheeseen asti. Tämä aika on vasemmalla olevan suunnitteluvaiheen ja oikealla olevan testausvaiheen välinen aika [14.] V-malli on esitetty kuvassa 4 [14].



Kuva 4. Ohjelmistokehityksen v-malli [14]

Mallin mukaisesti vasemmalla olevat ohjelmistokehityksen vaiheet, eli järjestyksessä ylhäältä alas: vaatimusmäärittely, yleinen suunnittelu, tarkka suunnittelu sekä ohjelmointi ovat yhteydessä näihin liittyviin testausvaiheisiin oikealla. Koska testauksen mahdollinen ajankohta mallissa on suunnittelu- ja siihen kytketyn testausvaiheen ero, on hyvä aloittaa tarkastelu kaikista lyhyimmästä tarkastusvaiheesta. Ohjelmoinnin aikana syntyneet virheet tulee saada kiinni yksikkö- tai komponenttitesteissä. Näissä testeissä koodin yksittäiset osat pyritään tarkastamaan heti niiden valmistumisen jälkeen. Testaaja yrittää löytää tapoja, joilla ohjelma menee alkuperäistä ohjelmiston määrittelyä vastaan. [15.] Koska yksikkö- ja komponenttitestit vaativat testaajalle pääsyn ohjelmiston lähdekoodiin, sillä testaus on suoritettava suoraan ohjelmakoodia vasten, ennen kuin sitä on yhdistetty osaksi järjestelmää. Tämän vuoksi yksikkö- ja komponenttitestit suoritetaan useimmin suoraan ohjelmistoa koodaavan ohjelmoijan toimesta. Näiden testien suurin etu on, että niiden avulla voidaan olla varmempia yksittäisen logiikan toimivuudesta. Vaikka yksikkötestaus ei varmistakaan ohjelmiston toimimista oikein, se on tärkeä osa testejä, sillä yksittäinen virhe logiikassa voi helposti aiheuttaa myöhemmin ongelmia järjestelmää rakennettaessa. Pienten yksittäisten virheiden korjaaminen on myöhemmin hankalaa, kun yksittäiset ohjelman palaset kasautuvat yhteen.

Yksikkö- ja komponenttitestauksen jälkeen on tärkeää, että tarkka suunnitelma ohjelmiston toiminnasta on testattu integraatiotestillä**.** Integraatiotestillä tarkoitetaan testausta, jossa osittaiset ohjelmistokoodit kasataan yhteen järjestelmäksi, ja jokaisen liitoksen yhteydessä tarkistetaan, että järjestelmä vielä toimii. Joissain tapauksissa järjestelmän palaset eivät toimi yhdessä heti ohjelmointijärjestyksessä, vaan niille joudutaan rakentamaan oma testausympäristönsä. [16.] On myöskin mahdollista, että ohjelmistoa rakentaessa joudutaan tai halutaan ensin odottaa muiden osien valmistumista ennen integraatiotestien aloittamista.

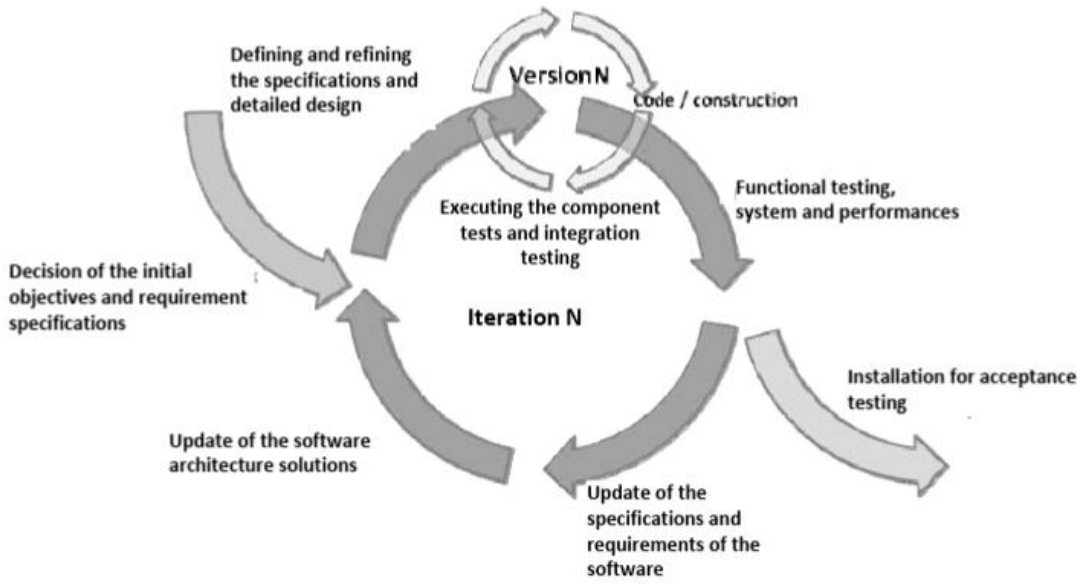
Integraatiotestien päätyttyä siirrytään koko järjestelmän yleiseen tarkistukseen. Järjestelmätestauksen tarkoitus on käydä integraatiotestien jälkeen läpi sekä koko järjestelmä, että myös siihen liittyvä dokumentaatio. Myös sellaiset osat ohjelmistoprojektia, jotka eivät sinänsä liity ohjelmistoon, voidaan tarkistaa. Esimerkkejä tarkastettavista ominaisuuksista ovat toiminnallisuus, käyttöliittymä, kuormitus sekä ohjelmiston turvallisuus. On tärkeää, että nämä testataan seikkaperäisesti järjestelmätestissä, sillä näiden ongelmien löytäminen järjestelmän käyttöönotossa, tai sen käytön aikana, aiheuttaa kustannusten huomattavaa kasvua. [16.] Tärkeää integraatiotesteissä on, että ne suoritetaan samanlaisessa ympäristössä, missä ohjelmistoa aiotaan asiakkaan puolelta käyttää. Tarkastuksen tärkeimmät tavoitteet ovat löytää kaikki jäljellä olevat epäkohdat ohjelmiston ja sen määritelmän välillä, sekä tarkistaa, että kaikki sille asetetut tavoitteet ovat toteutuneet. [17, s. 10] Järjestelmätestaus on tärkeä osa testausta, sillä siihen liittyy valtava määrä kysymyksiä kaikkien ohjelmiston toiminnan osa-alueiden kannalta. Kun ohjelmisto on näin testattu, voidaan olla huomattavasti varmempia siitä, että ohjelmiston käyttöönotto asiakkaan puolella menee suunnitellun mukaisesti.

Viimeisenä vaativuusmäärittely testataan hyväksymistestillä**,** joka ei oikeastaan nimestään huolimatta ole testi vaan eräänlainen koe. Hyväksymistesti on usein asiakkaan tekemä tarkastus ohjelmiston toiminnallisuuteen, jossa tärkeää on, että virheitä ei enää löydy. [17, s. 12.] Testin tarkoitus on testata kokonaista ohjelmistotuotetta. Tällaisessa testauksessa on järkevää käyttää loppukäyttäjää testaajana, sillä kyseessä on aito käyttötilanne. Testistä voidaan myös käyttää nimeä hyväksymiskoe, sillä testin aikana määritellään, ovatko sopimusehdot täyttyneet. Asiakkaan on tärkeää tarkastaa vaatimuksien toteutuminen, sillä usein ohjelma tulee testin jälkeen asiakkaan käyttöön. Kannattaa huomata, että mikäli tässä vaiheessa virheitä löytyy niiden korjaaminen voi tulla yritykselle hyvinkin kalliiksi. [18, s. 1–2.] Siksi on tärkeää, että tuote on testattu tarpeeksi laajasti jo ennen asiakkaalle toimittamista.

Vaikka ohjelmistokehityksen v-malli antaakin hyvän käsityksen ohjelmiston testaamisesta projektin aikana, siinä on myös joitakin puutteita, jotka on hyvä tuoda esille. Ensimmäiseksi tärkein on

huomio siitä, että v-malli voidaan käsittää niin, että testaus tapahtuu ohjelmoinnin jälkeen [14, s. 46]. Ohjelmistokehityksessä on kuitenkin tärkeää ottaa huomioon asiantuntevuus testien suunnittelussa, sekä niiden toteutuksessa. Testaus voidaan tehdä, ja usein tehdäänkin, myös kehityksen aikana. Ohjelmoijan on nopeampi ja helpompi luoda kirjoittamaansa ohjelmistopätkään yksikkötestaus ohjelmoinnin yhteydessä, kuin suunnitella testit ohjelmoinnin jälkeen. Mikäli testaus tehtäisiin erillisenä osana projektia ohjelmoinnin jälkeen, on hyvin mahdollista, että se hidastaisi ohjelmiston tuottamista huomattavasti. Syitä on useita, mutta tärkeämpinä ovat erillisen ajan varaaminen monimutkaiselle testausprosessille, ohjelmoijan kiireellisyys muiden tehtävien kanssa myöhempänä ajankohtana, sekä aika, jonka ohjelmoija tarvitsee muistellessaan uudelleen kyseisen koodipätkän vaikutusta eri järjestelmiin sekä vaatimuksia, jotka koodiin on kohdistettu.

Toisaalta voidaan myös katsoa, että v-mallin tapa katsoa ohjelmistoprojektin eri vaiheita solmuina on vanhanaikainen, ja nykyään ohjelmistokehityksen näkökulmasta usein väärä. Testauksen nopeus on ohjelmistokehityksessä tärkeää, sillä nopeammalla testauksella saadaan nopeammin tuloksia, joita voidaan lähteä tutkimaan. Ohjelmistokehityksessä onkin siirretty enemmän iteratiivisiin eli toistuviin malleihin, joissa nopea palaute vauhdittaa oikaisuja sekä korjauksissa kestävää aikaa [19.] Iteratiivinen malli on esitetty alla kuvassa 5 [14, s. 47].



Kuva 5. Kehitysvaiheet iteratiivisen mallin mukaisesti [14, s. 47]

Iteratiivinen selittää paremmin nykyisen ohjelmistokehityksen eri suunnittelu- ja testausvaiheita. Mallissa kehitys käynnistetään vasemmalta määrittelemällä sekä hiomalla ohjelmiston tarkkaa

vaatimusta. Mallissa vaaditaan päätös alkuperäisistä tavoitteista sekä vaatimusmäärittelystä ohjelmistolle. Tämän jälkeen voidaan aloittaa ohjelmiston kehityksen toisto eli iteraatio. Jokaisen iteraation ”n” yhteydessä on olemassa oma ohjelmiston versio. Ohjelmiston luonnin yhteydessä sille tehdään aikaisemmin selitetyt komponentti- sekä integraatiotestit. Tätä jatketaan, kunnes versio on useamman ominaisuuden luonnin sekä testauksen jälkeen sellaisessa vaiheessa, että sille voidaan seuraavaksi suorittaa järjestelmätestaus. Valmis tulos voidaan tässä vaiheessa lähettää asiakkaalla oikealle lähtevän nuolen mukaisesti hyväksymistestaukseen. Ohjelmistot ovat kuitenkin usein jatkokehityksen alaisia. Uudet tarpeet, ominaisuusvaatimukset, tai havaitut puutteet voivat käynnistää tarpeen jatkokehitykselle tai korjauksille. Tällöin ohjelmiston vaatimuksia joudutaan muokkaamaan, ja samalla pitämään huolta ohjelmiston arkkitehtuurin tuomisesta ajan tasalle. Tällöin kun koko ympyrä on käyty läpi, uutta toistoa voidaan alkaa ohjelmoimaan uudistettujen vaatimusten suhteen.

Iteratiivinen malli on useammilla tavoin parempi tapa kuvata nykyistä ohjelmistokehitystä, sillä se ottaa huomioon myös jatkokehityksen sekä mahdollistaa joustavamman kehityksen. Mallin avulla ohjelmiston kehityksestä tulee joustavampaa sekä samalla tehokkaampaa. Täytyy kuitenkin huomata, että testaustiimin sekä ohjelmointitiimin sisällä ei välttämättä käytetä vain ja ainoastaan näitä mainittuja testausmenetelmiä. Mallien ymmärtäminen auttaa meitä hahmottamaan suuria kokonaisuuksia ohjelmistokehityksessä, mutta käytännössä ohjelmistotestausta voidaan suorittaa myös useammalla tavalla. Seuraavaksi käsitellään ohjelmistotestausta käytännössä, ja sitä, miten ohjelmiston vikoja pystytään havaitsemaan eri tavoin.

3.1 Ohjelmistotestaus käytännössä

Tässä kappaleessa testaustiimin käytännön työtaakkaan katsotaan kuuluvan se kaikki testaus, jota ei ole tehty ohjelmoijien yksikkö- ja komponenttitestien yhteydessä. Ohjelmistokoodin testauksessa on tärkeää huomioida aina ohjelmiston kehityksen jatkuvuus. Ohjelmistotestauksen kannalta tärkeää on, että kehitys on sellaisessa vaiheessa, että ohjelmiston ominaisuuksia pystytään onnistuneesti testaamaan. Liian aikainen testaus voi johtaa turhaan testaustyöhön, mikäli testattavaa ominaisuutta kehitetään. Toisaalta myös ennen ohjelmiston koodin valmistumista voidaan suorittaa erilaisia testaamisvaiheita. Eri testausvaiheista voidaan huomata, että ne jokseenkin voidaan ajoittaa aikaisemmin mainittuun ohjelmistokehityksen v-mallin sekä iteratiivisen mallin mukaisesti. Suunnitelmien testaus voidaan laskea testaukseksi, mutta sen aikana usein itse ohjelmistotestaajien työosuus jää vain laajemmillaan vaatimusten analyysiin. Näiden vaiheiden

aikana voidaan kuitenkin jo tutkia ohjelmistoa ja kehitellä suunnitelmia tulevaa testausta varten [20]. Vaatimusmäärittelyn ja testaussuunnitelman edetessä ohjelmistotestaajien kuva tulevan ohjelmiston testaamisesta vahvistuu. Tämän vaiheen hyödyntäminen voi olla haastavaa, sillä konkreettista ohjelmistoa ei vielä ole. Toisaalta tuotteen vaatimusten sekä toiminnallisuuden tunteminen on tärkeää. Tietämys näistä kehitysvaiheista nopeuttaa testaamisen suunnittelua sekä mahdollistaa testaustiimin pääsyn suoraan testaustyöhön, kun ohjelmisto alkaa olemaan testattavassa vaiheessa. Ohjelmistotestaajat voivat myös keskittyä tuntemaan projektin testauksessa käytettävään ohjelmistoon sekä laitteistoon, palautteen antamiseen projektin suunnitelmista sekä suunnitella alustavia testejä. Projektin testauksen hidas käynnistyminen voi olla yritykselle negatiivinen kustannustekijä, mutta toisaalta hyvin suunniteltu testaus nopeuttaa testitapausten luomista, sekä parantaa luotettavuutta testaajien taitoon pystyä rakentamaan tarkkoja testejä.

Projektin ohjelmointivaiheessa testaustiimissä päästään keskittymään testaustapausten suunnitteluun. Testitapauksella tarkoitetaan kuvausta testitilanteesta, ja siinä selitetään mitä ja miten testataan. Testitapaus ohjaa suorittavaa testaajaa yhden ominaisuuden tai ehdon testaamisessa ohjelmistossa. [21.] Testitapaus koostuu yhdestä tai useammasta testivaiheesta, jossa ohjelmistotestaaja tarkistaa vaiheen esittämän väitteen paikkansapitävyyden. Sen jälkeen, kun väite on kokeiltu todistaa ohjelmistossa tapauksen määrittelemällä tavalla, testaaja merkitsee vaiheen joko onnistuneeksi (pass) tai epäonnistuneeksi (fail) [22, s. 19]. Käymällä koko testitapauksen väitteet läpi yksitellen voidaan tarkistaa, oliko yksikään vaihe epäonnistunut. Jos mikään vaiheista ei johtanut kuvattuun lopputulokseen, silloin koko testitapaus voidaan usein heti todeta myöskin epäonnistuneeksi. Testitapausten luonti on tärkeää, sillä ne auttavat tiedostamaan millä tavoin ominaisuutta on aikaisemmin testattu. Mikäli testausmetodeja tai läpimenokriteerejä vaihdettaisiin, testien tulokset saattelisivat vaihdella turhaan eri ohjelmistoversioiden välillä. Hyvin suunnitelluissa testitapauksissa ei jää varaa miettiä, onko lopputulokseen vaikuttanut testaajan erilainen tulkinta tai toiminta testin aikana. Testitapauksista puolestaan voidaan rakentaa testijoukko**,** joka koostuu useammasta testitapauksesta. Usein joukko rakennetaan esimerkiksi miettimällä, mitkä testitapaukset kuuluvat yhteen samaan ominaisuuteen tai ominaisuuksien ryhmään. Testijoukot ovat aina mahdollisimman pieniä, sillä ylisuuret testijoukot sisältävät luultavasti turhan paljon testejä, jotka puolestaan vievät ohjelmistoprojektilta liikaa resursseja ja aikaa. [22, s. 21.] Testijoukkoja voidaan myös puolestaan vielä luokitella erillisten ryhmien alle. Testien ryhmittely voi olla hyvinkin kannattavaa tehokkuuden näkökulmasta, sillä uusien ominaisuuksien testauksessa voi olla hyödyllistä kohdentaa testejä niihin ryhmiin, joihin ominaisuuksien ohjelmoinnissa tapahtuvat virheet ovat voineet vaikuttaa. Testiryhmiä voidaan myös pitää osana testauksen dokumentaatiota, sillä ne näyttävät, mitä, ja miten on aikaisemmin testattu. Samalla on hyvä aika miettiä

testaustapausten vaatimuksia, eli sitä, mitä laitteita ja työkaluja testeissä kannattaa ja pitää hyödyntää. Hyvin suunnitellut ja valmiiksi mietityt testitapaukset, jotka perustuvat projektin vaatimusmäärittelyyn, ovat loistava aloitus onnistuneelle ohjelmiston testaukselle.

Kun ohjelman kehitys on vaiheessa, jossa yksikkötestit ovat menneet läpi, ja eri komponentteja voidaan alkaa yhdistämään, on testaustiimin mahdollista alkaa testaamaan järjestelmää. On tärkeää, että viimeistään tässä vaiheessa testausympäristö on pystytetty. Testausympäristöllä tarkoitetaan ohjelmistokehityksessä sitä yhdistelmää ohjelmistoja, laitteita sekä työkaluja, joilla testaus voidaan suorittaa [23]. Tässä vaiheessa voidaan myös luoda lavasteympäristö. Lavasteympäristö on testausympäristö, jonka tarkoitus on olla mahdollisimman paljon uusimman asiakkaille tuotetun version kaltainen. Ajatuksena on, että siinä missä testiympäristössä saatetaan kokeilla uusia yksittäisiä ominaisuuksia, lavasteympäristössä voidaan kokeilla koko järjestelmän toimintaa. [23.] Tämä tulee erityisen hyödylliseksi esimerkiksi uusien ohjelmaversion julkaisujen yhteydessä. Kun tarvittava testausympäristö on pystytetty, voidaan testausta alkaa suorittamaan testaussuunnitelman ja/tai strategian mukaisesti. Ohjelmiston testaaminen on monimutkainen kokonaisuus, jossa usein testitapauksia vasten kokeillaan eri lähtöarvoja sekä verrataan ohjelman toimintaa oletettuun toimintaan. Testitapaukset voivat projektista toiseen vaihdella huomattavasti. On tärkeää, että testitapaukset pohjautuvat niihin käyttötarkoituksiin, jotka vaatimusmäärittelyssä on projektille annettu. Tällä tavoin ohjelmiston testaaminen perustuu todellisiin asiakasvaatimuksiin, ja luottamus ohjelmaan kasvaa. Usein testitapauksia lähdetäänkin purkamaan manuaalitesteillä. Manuaalitestillä tarkoitetaan ilman automaatiota toimivaa testausta, jossa ohjelmistotestaaja asettautuu loppukäyttäjän rooliin. Ohjelmiston ominaisuuksia lähdetään testaamaan yksitellen itse ohjelmistotestaajan toimesta. Manuaalisesti testatessa on huomioitava, että testin vaiheet on kuvattu yksitellen selkokielellä testitapauksen yhteydessä. [24, s. 3.] Manuaalitestaus on hyvin yleinen ohjelmistotestauksen muoto. Automaatiotestien suosiosta huolimatta se on pitänyt pintansa, sillä on olemassa testitapauksia, joissa ihmisen tekemä testaus on yksi parhaimmista tavoista varmistaa toivottu lopputulos.

Testitapauksen tullessa loppuun merkitään saavutettu lopputulos usein testitapauksen yhteyteen käytetyssä testausohjelmistossa. Tuloksesta riippuen testi saatetaan ajaa uudelleen, tai mahdollisesti laittaa suoraan korjaukseen ohjelman kehittäjille. Testaajan tavoite on saada kaikki testijoukot ajettua niin, että mahdolliset viat saadaan ohjelmistotiimin sekä testipäällikön tietoon. Testijoukkoja sekä koko testipakettia voidaan myös joskus ajaa useampaan otteeseen, kunnes paketti on saatu läpi uusilla korjatuilla versioilla. Samalla joitain tapauksia voidaan joutua ajamaan uudelleen muutosten yhteydessä, sillä muutokset ovat voineet rikkoa myös aikaisemmin hyvin

toimiviksi todettuja ominaisuuksia. Tämän koko pitkän prosessin aikana on mahdollisuus tehdä inhimillisiä virheitä. Tämän lisäksi testauksen yhteydessä tulokset voivat erota riippuen käytetyistä ympäristöistä, testaajien omista tavoista ymmärtää testivaiheita tai esimerkiksi käytetystä ohjelmistoversiosta. Koska ohjelmistotestauksen työmäärä voi helposti näin kasvaa liiaksi projektille, on hyvä puhua ohjelmistotestauksen tehokkuuden lisäämisestä sekä niistä ongelmista, joihin uusia luovia ratkaisuja on lähdetty etsimään.

3.2 Ohjelmistotestauksen tehokkuus ja ongelmat

Otetaan aluksi yksinkertainen esimerkki testin suorittamisesta: verkkosovelluksen käynnistys tietokoneella. Testaajalle tärkeintä on aluksi vaatimusmäärittely, jonka mukaan varmastikin asiakkaan on pystyttävä käynnistämään ohjelmisto tietyillä sovelluksilla ja ehdoilla. Tässä tapauksessa ohjelmaa saatetaan haluta pystyä käyttämään usealla verkkoselaimella. Määrittelyn mukaisesti testitapaus joudutaan siis jo heti alkuun mahdollisesti ajamaan useammalla eri lähtötapauksella. Tämän lisäksi onnistunut tapaus, jossa käyttäjä pääsee käyttämään sovellusta oikeilla ehdoilla, on vain unelmatilanne. Testaajan tulee miettiä sellaisten lähtöarvojen sarjaa, joilla ohjelma voisi mahdollisesti mennä rikki. Mitä tapahtuisikaan, jos ohjelman yrittäisi käynnistää useita kertoja toistuvasti? Tai mitä tapahtuisi, jos verkkoyhteys katkeaa ohjelman käynnistyksen aikana. Testaajien päätarkoitus on löytää niitä vikatilanteita, joissa ohjelman ei pitäisi käyttäytyä sellaisella tavalla, josta voisi olla asiakkaalle tai kehittävälle yritykselle haittaa. Monimutkaisilla ehdoilla etsiminen usein tuottaa uusia yksittäisiä testejä, jotka pidentävät testitapauksia sekä vastaavasti testijoukkoja. Yksittäisen testitapauksen piteneminen voi olla huono asia, sillä pitkät testitapaukset kestävät vievät entistä enemmän resursseja, kun niitä aletaan testaamaan erilaisilla asetuksilla ja ympäristöillä. Testaustyön viemä aika siis lisääntyy moninkertaisesti mitä pidemmäksi testit muuttuvat. Tämä parabolinen työn määrän lisääntyminen täytyy pitää mahdollisimman kurissa, jotta testaustiimin resurssit riittävät mahdollisimman tarkan testauksen suorittamiseen.

Ohjelmistotestauksen tehokkuus voi olla monimutkainen asia määritellä. Toisaalta tehokasta voi olla esimerkiksi se, että ajetaan mahdollisimman paljon testitapauksia tiimille annetussa ajassa. Toisaalta testitapausten määrä ei välttämättä kerro todellisen testityön arvosta pelkän määrän perusteella. Sen sijaan voidaan katsoa esimerkiksi suoritettuja testivaiheita, sitä, kuinka paljon yksittäisiä testausvaiheita tiimi on käynyt läpi, ja montako ohjelmointivirhettä on löytynyt. Löydetyt virheet voivat olla erinomainen indikaattori testaustiimin tekemän työn tärkeydestä, mutta

niihin keskityttäessä unohdetaan täysin testauksen ylläpitävyyden luonne. Yksittäiseen testauksen tehokkuuden mittareihin nojatessa tehokkuutta on vaikea määrittää. Sitä voidaan kuitenkin kuvata testitiimin kykynä lisätä ohjelmiston laatua jaettuna työlle annetulle ajalle. Tehokas testitiimi kykenee saamaan ohjelmiston sellaiseen tilaan, että verrattuna tiimille annetuille resursseille, siitä ei keskimäärin löydy tätä työtä kalliimpia ongelmia. Paraskaan testaustiimi ei välttämättä kykene aina löytämään kaikkia pieniä vikoja, tai keksimään parannusehdotuksia ennen asiakasta. Sen sijaan, hyvä testaustiimi kykenee priorisoimaan**,** eli asettamaan testaustavoitteita tärkeysjärjestykseen.

Testaustiimin kyky käydä läpi monimutkaisia testaustapauksia sekä testijoukkoja perustuu vahvasti dokumentaation, ohjelmiston laadun sekä työnjaon lisäksi myös testaustapoihin. Erilaiset tavat kohdistaa testausta auttavat huomattavasti työkuorman vähentämisessä. Yksi suurimmista tavoista priorisoida testausta on yksinkertaisesti jättää osa testeistä ajamatta. Tämä voi olla hyvä valinta esimerkiksi tilanteessa, jossa ohjelmistoon tehdään korjaus. Korjaukset on usein keskitetty yksittäisiin ominaisuuksiin, jolloin juuri tämän ominaisuuden testaus voi olla kaikista kustannustehokkain tapa olla vakuuttunut ohjelman kunnosta. Tämä strategia voi vähentää työkuormaa huomattavasti, mutta on myös riskialtis. Testitiimi saattaa kuvitella, että korjauksen aikana on koskettu vain yhteen ominaisuuteen. Riippuen toteutuksesta, yksittäisen ominaisuuden muuttaminen voi aiheuttaa kerrannaisvaikutuksen, jossa useita eri ominaisuuksia menee korjauksen jälkeen rikki. Oletus vahingon rajautumisesta osoittautuu vääräksi, mutta sitä ei huomata testauksen yhteydessä. Kokenut testaushenkilöstö voi pystyä arvaamaan usein oikein, mutta yksikin virhe voi helposti jäädä pitkän aikaa tarkastamatta, mikäli ohjelmistolle ei tehdä suurempaa testausta säännöllisesti. Siksi vaikka tämä toimintatapa on olennainen ajan säästämiseksi, täytyy tiedostaa, että näin tehtäessä on hyvä pitää huolta myös tilanteista, joissa testauksen ohi mennä virheitä.

Tärkeää on huomata, että tavat, joissa testausta vähennetään kaiken kaikkiaan eivät ole välttämättä hyviä johtuen niistä ylimääräisistä kustannuksista, jotka syntyvät, kun ongelma selviää vasta asiakkaalla. Sen lisäksi ohjelmointivirheet ovat ongelmia, jotka kasvavat koossa ajan kanssa. Välittömästi huomatut virheet pystytään korjaamaan niin, ettei niistä aiheudu uusia, vakavampia virheitä myöhemmin ohjelmiston toiminnassa. [25, s. 17.] Tämän vuoksi testauksen määrää ei usein haluta vähentää, vaan sen sijaan keskitytään niihin testaustoimenpiteisiin, jotka parantavat löydettyjen ongelmien määrää ja raportoinnin sekä korjauksen tehokkuutta. Nämä toimenpiteet ovat yleisesti niitä hyviä kommunikaatio- sekä projektityöskentelytapoja, joiden tärkeys tuotiin jo aikaisemmin esille. Hyvän yhteistyön lisäksi sovellustestauksessa voidaan keskittää huomiota

myös raportoinnin laatuun. Mitä pidempään ongelman korjaamiseen menee, sitä kauemmin kestää saada uusi versio ulos ohjelmistosta, josta voidaan lähteä etsimään uusia ongelmia. Hyvä raportointi auttaa usein ohjelmoijia löytämään ongelmia nopeammin ja tehokkaammin. Esimerkiksi selitys testitapauksen ympäristöstä, kuvaus siitä, miten testi on ajettu, sekä yksityiskohtaiset vaiheet testin ajosta tekevät ohjelmoijan työstä helpompaa, kuin vain ilmoitus siitä, että jokin ominaisuus on rikki.

Manuaalisesti tehdyn ohjelmistotestauksen ongelmia on monia, mutta tiivistettynä voidaan sanoa, että eri tapausyhdistelmien muodostamat järjestelmät lähtevät liian nopeasti tasolle, jossa kaikkea ei voida testata. Useat tavat parantaa testauksen laatua sekä tehokkuutta ovat yksittäisiä tekijöitä, jotka vaikkakin lisäävät tehokkuutta, eivät pysty pitämään perässä vaatimusten kasvaessa. Ohjelmistotestaukseen tällä tavalla jää aukko, jossa ohjelmaa ei pystytä testaamaan perustasolla jatkuvasti tiettyinä ajankohtina. Testauksessa täytyisi voida hyödyntää jotain tapaa, jolla voisi paljastaa hyvin itse ohjelmiston manuaalisessa testaamisessa tapahtuvia virheitä, antaen tietynlaisen turvaverkon testauksen tasolle. Manuaalisen testauksen vahvuus on testaajan ammattitaito, kyky tutkia tuotetta sekä tieto ohjelman toiminnasta. Hyvä testaaja kykenee välittämään tietoa, pystyy tekemään yhteistyötä ohjelmoijien sekä muiden kanssa, ja kykenee tekemään johtopäätöksiä testauksen priorisoinnista, ja siitä, täytyykö tiettyä vikaa ylipäänsä edes korjata. Testausorganisaatiossa henkilöstömuutokset voivat aiheuttaa ongelmia, sillä uuden työntekijän kouluttaminen tarvitulle osaamisen tasolle tuotteen ominaisuuksien ja vikojen tuntemuksessa vie paljon aikaa. Henkilöstömäärän lisäys ei siis toisin sanoen ole välttämättä myöskään aina paras tapa parantaa testauksen laatua. Tarvitaan järjestelmä, jolla manuaaliseen testauksen heikkouksia voidaan paikata. Automaatiotestaus on tähän oiva ratkaisu. Seuraavassa kappaleessa käsitellään sitä, mitä automaatiotestaus on, ja miksi siitä on tullut tärkeä osa eri yritysten testausstrategiaa.

4 Testiautomaatio

Ketterän ohjelmistokehityksen yleistyessä manuaalitestauksesta on osittain tullut liian hidasta. Manuaalitestauksen lisäksi on tarvittu työkaluja, joilla testauksesta voidaan tehdä nopeampaa sekä luotettavampaa. Näiden keinojen etsimisestä on tullut tärkeää yhä useampien yritysten siirtyessä laajemmin ketterään kehitykseen [13, s. 20]. Testiautomaatiosta on tullut ratkaisu, jolla tiimien koot voidaan pitää tarpeeksi pieninä, ilman että testauksen määrää joudutaan juurikaan vähentämään.

Automaatiotestauksella tarkoitetaan sellaista testausta, jossa ihmisen sijaan eri testitapauksia käy läpi sovellus, joka on ohjelmoitu käymään läpi tietyt testivaiheet. [26.] Testiautomaatioinsinööri on puolestaan henkilö, joka kirjoittaa ohjelmistokoodia tätä sovellusta varten. Hänen tehtävänsä on ohjelmoida sovellus niin, että se kykenee varmistamaan testitapauksen onnistuneesti. Testiautomaation voi pystyttää usean eri ohjelmointikielen tai alustan päälle. Valinnassa usein vaikuttaa projektissa luodun ohjelmiston tyyppi. Esimerkiksi mobiilisovellukselle voidaan käyttää eri kieltä kuin tietokoneella toimivalle sovellukselle. Testausautomaation kirjoittaminen vie resursseja testaustiimiltä siinä missä manuaalinenkin testaus, mutta kun se saadaan pyörimään, tietokone voi testata asioita jatkuvasti ja nopeammin kuin mitä testaajat pystyvät manuaalisesti tekemään.

Käytännössä testiautomaatio on siis yhdistelmä skriptejä, jotka ajettaessa testiohjelma pystyy automaattisesti käymään läpi tehdyn sovelluksen ominaisuuksia. Nämä skriptit itsessään voidaan myös ajaa automaattisesti, jolloin suurempi osa koko testauksen työnkulusta pystytään suorittamaan automaattisesti. Automaattinen ajo mahdollistaa jatkuvan kehityksen sekä jatkuvan integraation**,** jossa testaus ei jää pullonkaulaksi. Automaattinen testaus ei kuitenkaan aikaisemmin mainitun mukaisesti rakenna itseään, vaan eri testitapaukset täytyy aina automatisoida erikseen. Siksi testiautomaation vahvuuksien sekä heikkouksien arvioiminen on tärkeää. Projektin ei ole järkevää automatisoida kaikkea, vaan automatisaation lisääminen strategiaan täytyy lisätä projektin testaukseen lisäarvoa. Ymmärtämällä automaation paikan testausstrategiassa, voi projekti saada aitoa arvoa automaattisesta testauksesta ja täten vähentää läpi pääsevien virheiden määrää.

4.1 Automaatiotestauksen vahvuudet ja heikkoudet

Mietittäessä sitä, mitä projektissa on ylipäätään järkevää lähteä automatisoimaan, täytyy tuntea, mitä automaatio oikeasti tekee. Automaation vahvuuksia ja heikkouksia voidaan esimerkiksi tarkistella kahdelta näkökannalta. Ensimmäisenä kantana on itse automaatiotestauksen omat vahvuudet käytännössä, eli mitä automaatio pystyy tai ei pysty tekemään ihmistä paremmin. Toisaalta voidaan miettiä myös automaatiotestauksen vaikutusta projektin aikatauluun sekä resursointiin. Automaation liittäminen testausstrategiaan oikealla tavalla vaatii näiden ominaisuuksien tuntemista niin, että manuaalista testausta tekevät ohjelmistotestaajat pystyvät tekemään yhteistyötä automaation kanssa sekä luottamaan automaattisen testauksen vahvuuksiin.

4.1.1 Testiautomaation vahvuudet

Automaattisen testauksen testausvauhti on yksi sen suurimmista eduista. Koska testejä ajaa tietokone, pystyy se hyvin nopeasti tarkistamaan sille tehnyt olettamukset sekä lopputuleman, ja siirtymään heti seuraavaan vaiheeseen. Etuna voidaan myös pitää tietokoneen kykyä olla tekemättä inhimillisiä virheitä itse testauksen aikana. Koska testausohjelmisto tarkistaa aina tismalleen sille määritellyt asiat, ei se vahingossa tee vääriä tulkintoja tai unohda tarkistaa toiminnallisuutta, jolloin harmaata aluetta testauksen lopputuloksessa ei ole. Testausautomaatiolla pystytään myös käytännössä luomaan testitapauksia, jotka manuaalisella testauksella eivät olisi tarpeeksi tehokkaita. Hyvänä esimerkkinä testausautomaatiossa voidaan esimerkiksi käydä läpi useita kertoja päivässä ohjelman kaikkien perustoiminnallisuuksien kokeileminen. Lyhyiden tapahtumasarjojen toisto useita kertoja ei ole automaatiolle ongelma, mutta manuaalisesti testattuna veisi liian paljon aikaa. Samalla testitapauksia tehdessä testien kehittäjä joutuu käymään läpi testausvaatimuksia ajaessaan automaatiotestejä useita kertoja. Ohjelma tulee siis jo kehityksen aikana testattua, ja testien valmistuttua voidaan olettaa, ettei ominaisuus mene enää hajalle myöhemmin ilman, että se huomataan.

Samalla automaatiotestaus voi avata uusia testitapausesimerkkejä, kuten esimerkiksi suorituskykytestauksen, jossa ohjelmaa pommitetaan usealla yhtäaikaisella komennolla tai käyttäjällä samaan aikaan. Suorituskykytestaus voi helposti mennä ohi testauksesta, sillä se vaatii, että ohjelmaa kuormitetaan huomattavasti testauksen aikana. Testausautomaatio pystyy samanaikaisesti luomaan tarvittavan paineen, mutta myös tallentamaan esimerkiksi suorituskyvyn menetyksen,

joka testin yhteydessä havaittiin automaattisesti. Näin vertailukelpoista dataa voidaan saada nopeasti, ja kerätä esimerkiksi joka päivä, jotta ohjelman toimintaa osataan tarvittaessa korjata. Tässäkin tapauksessa testausautomaatio toimii jatkuvasti taustalla, ja osaa heti varoittaa, mikäli testauksen yhteydessä jokin on mennyt pieleen. Testausautomaatio avaa siis uusia testausmahdollisuuksia, auttaa meitä virheiden nopeassa löydössä, ja helpottaa manuaalisen testauksen työkuormaa. Automaation hoitaessa kaikki itseään toistavat testivaiheet, pystyy manuaalisen testauksen työpanoksen kohdistamaan sinne, missä sitä todellisuudessa tarvitaan.

Testiautomaatio voi myös näkyä eri testaajien sekä ohjelmoijien työkuvassa positiivisesti. Kehittäjä voi käyttää ohjelmistoa yksikkötestauksen aikana, joka kertoo hänelle, mikäli ohjelmistosta löytyy automaattisesti virheitä. Tämä vähentää helposti löytyvien virheiden määrää jo ennen koodin päästämistä testaustiimin käsiin. Myös testaustiimin sisällä testiautomaatiosta voi olla hyötyä manuaalitestaajille, kun automaatiotestien tulokset auttavat heitä nopeammin ongelmien jäljille testiajon jälkeen. Kun testausautomaation avulla voidaan pyörittää päivittäisiä testejä**,** saadaan tieto hajonneista ominaisuuksista heti seuraavaksi päiväksi. Testausautomaation avulla voidaan myös tehdä esimerkiksi automaattisia kooditarkastuksia päivitysten yhteydessä. Automaattinen laaduntarkastus testauksen kaikilla eri tasoilla antaa mahdollisuuden nopeuttaa koko ohjelmistokehityssykliä sen kaikissa vaiheissa.

4.1.2 Testiautomaation heikkoudet

Testiautomaatio täytyy aina luoda projektin mukaiseksi. Mikäli automatisoitu testitapaus ei tarkista jotain ominaisuutta tai erikoistapausta, ei se pysty itse siitä syntyvää virhettä huomaamaan. Automaatio täytyy aina suunnitella niin, että se pystyy ottamaan huomioon eri rajatapaukset ohjelman toiminnassa. Siinä missä manuaalisessa testauksessa ihminen voi arvioida esimerkiksi onnistumisen asteen, automaatiotesti tulee aina sanomaan suoraan, onnistuiko testi vai ei. Useimmiten tieto virheen synnystä tai asteesta voi olla hyvä tapa lähteä purkamaan ongelman kriittisyyttä tai lähtöperää. Automaattitestauksen yhteydessä on toisin sanoen usein hyvä lähteä suorittamaan myös manuaalinen nopea tarkistus virheen löytymisen yhteydessä.

Automaatiotesteissä testitapaukset voivat olla lohkeilevia, joka tarkoittaa testiä, jonka lopputulos on näennäisesti satunnainen. [27.] Näille testeille on ominaista, että ne voivat satunnaisesti epäonnistua, mutta uudelleen ajettaessa tarpeeksi monta kertaa, mennä läpi. Lohkeilevat testit ovat

testauksen kannalta ongelmallisia, koska jos lopputuloksesta ei voida olla varma, ei silloin myöskään tiedetä, milloin testi on oikeasti epäonnistunut, ja sovellus korjauksen tarpeessa. Lohkeilevia testejä ei voi jättää automaatioon, vaan ne täytyy korjata. Manuaalitestaajien on luotettava siihen, että automaatio oikeasti löytää viat kaikesta, mikä sen testattavaksi on tuotu. Lohkeilevat testit vaikuttavat negatiivisesti tähän luottamukseen, ja vaikeuttavat uskoa automaatiotestauksen kykyyn käydä läpi testitapauksia.

Testausautomaatiolla pystytään ajan kuluessa saavuttaa testauskulujen huomattavaa pienentymistä, mutta lyhyellä aikavälillä kyseessä on projektille kallis investointi, joka vaatii myös osaamista käyttöönoton yhteydessä. Projektin työtapojen ei ainoastaan testauksen, mutta myös raportoinnin sekä kehityksen osalta on oltava valmiit muutokseen, kun testausautomaatio aiotaan saada kehitykseen mukaan. [28, s. 28.] Automaatioskriptien kirjoittajat joutuvat käyttämään huomattavan osan työstään automaation luontiin sekä ylläpitoon. Varsinkin kehitysvaiheessa tässä on myös mahdollista tehdä virheitä, jotka aiheuttavat kulujen nousua ennen kuin automaation hyötyjä aletaan konkretisoimaan projektissa. Tämän vuoksi automaation kehityksessä on tärkeää suunnitella hyvin, miksi ja miten automatisaatiota aiotaan tuoda osaksi ohjelmistokehitystä.

Automaatiotestauksella ei myöskään kokonaan päästä irti manuaalitestauksesta, sillä automaatiotestaus on kaikista eniten vain uusi mahdollisuus testata tehokkaammin vanhoja testitapauksia. Vaikka automaatiotestaus avaakin mahdollisuuksia uusille testeille, se ei kuitenkaan poista manuaalisen testauksen tärkeyttä. Manuaalinen testaus on siitä sekä hyvä, että huono, että ihminen pystyy tekemään johtopäätöksiä näkemästään. Visuaaliset sekä tulkinnanvaraiset tulokset, esimerkiksi käyttöliittymän ulkonäkö, eivät ole hyvin testattavissa automaatiotesteillä. Hyvällä yhteistyöllä testiautomaatio tuo suunnatonta lisäarvoa projektille, mutta parhaimmillaan se toimii rinta rinnan manuaalitestauksen kanssa. [28, s. 28.]

4.2 Automaatiotestaus osana testausstrategiaa

Automaatiotestaamisella on useita vahvoja puolia, mutta myös heikkouksia, jotka joudutaan ottamaan huomioon yrityksen testistrategiaa mietittäessä. Testausstrategia voi esimerkiksi koostua siitä, paljonko testausta halutaan automatisoida, ja montako manuaali sekä automaatiotestaajaa tiimissä on. Kun projektin vaatimusmäärittely on tehty, on helpompi alkaa miettimään, miten eri ominaisuuksia tullaan testaamaan. Hyvä testausstrategia ottaa huomioon aina automaatiotestit

tavalla tai toisella. Siinä missä joissain pienissä projekteissa on tarpeen vain käyttää yksikkötestausta nopeuttavaa ohjelmistoa, isommissa projekteissa voi olla hyvä rakentaa kokonainen automaatiotestausympäristö ohjelmistotestauksen avuksi. Tämän jaon harkinta testauksen yhteydessä on kannattavaa, sillä mikäli testausta joudutaan myöhemmin muuntamaan automaattiseksi se voi tulla loppujen lopuksi kalliiksi. Tarpeeksi aikaisin tehty automaatiotestaus voi alusta alkaen auttaa välttämään helposti havaittavia ohjelmistovikoja, sekä parantaa ohjelmiston laatua esimerkiksi saamalla kiinni ongelmia huomattavasti manuaalista testausta nopeammin.

Kun päätös automaatiotestauksen lisäämisestä testausstrategiaan on tehty, on hyvä miettiä, mitä testausta lähdetään tuomaan automaatioon ensimmäisenä. Tässä tärkeää on tietää automaation vahvuus. Automaatiolla on kaikista helpointa tarkistaa olettamuksia [29]. Testien suorittaminen eri ympäristöissä ei usein ole automaattisille testeille ongelma, sillä testitapaukset voidaan ajaa eri syötteillä sekä oletusarvoilla kerta toisensa jälkeen. Normaalisti hitaat sekä työläät, itseään toistavat testit sujuvat automaatiolta parhaiten. Työläiden testitapausten automatisaatio voi tuoda huomattavaa lisäarvoa projektille, sillä juuri niistä voidaan poistaa eniten työtä manuaalisen testauksen osalta. Toinen hyvä tapa lähteä automatisoimaan testitapauksia on miettiä, mitä manuaalitestauksessa tällä hetkellä tarvitaan. Automaatiolla voidaan esimerkiksi simuloida useamman käyttäjän tapauksia, joka ilman useamman testaajan apua olisi mahdotonta manuaalisessa testauksessa. Näitä testejä suunnitellessa on automaatiotestauskehittäjien tärkeä kommunikoida muiden testaajien kanssa, jotta manuaalisen testauksen puutteita voidaan kartoittaa automaation kehitystä varten. Kolmantena voidaan miettiä projektille tärkeitä vaiheita, joiden testaus on kaikista hyödyllisempää. Esimerkkeinä tällaisesta testauksesta voitaisiin esimerkiksi pitää julkaisutestausta, jossa koko järjestelmän uuteen versioon tehdään perinpohjainen testaus. Julkaisutestauksessa on tärkeää pystyä todistamaan, että tätä versiota pystytään käyttämään asiakkaan toimesta normaalisti. [30.] Julkaisutestauksen yhteydessä automaatiotestauksen luoma turvaverkko voi olla todella tärkeää. Koska uudet muutokset ovat voineet rikkoa jotain aikaisemmin toiminutta, koko ohjelmiston toiminta joudutaan käymään läpi. Mikäli testiautomaatiolla on tehty perustason testaus kaikkiin ohjelmiston toimintoihin, sen ajaminen tämän julkaisun yhteydessä voi auttaa varmistumaan siitä, ettei mitään olennaista ole rikki. Joissain tapauksissa tällä testauksella voidaan saada kiinni elintärkeitä vikoja, jotka ovat rikkoneet jotain, mitä manuaalisessa testauksessa ei ole tajuttu testata. Toisaalta myös uusien korjauksien jälkeen nämä testit on hyvä ajaa, sillä testien ajaminen automaattisesti ei juurikaan vie resursseja verrattuna siihen, että kaikki testit ajettaisiin manuaalisesti.

Testausstrategiassa kuuluu kuitenkin myös ottaa huomioon, että itse automaatiotestien suunnitteleminen, toteuttaminen, ylläpito sekä valvominen voi viedä yhden tai useamman testaustiimin jäsenen täysiaikaisen huomion. Kun ohjelmaan kehitetään uusia ominaisuuksia, muokataan vanhaa toiminallisuutta, tai havaitaan virheitä, on mahdollista, että testausta joudutaan päivittämään. Testiautomaatio ei ole projektille ilmaista, mutta se antaa sille arvoa, jota aikaisemmin ei ole pystytty saamaan. Automaation hyöty kasvaa ajan kasvaessa, sillä kaikki aikaisemmat testitapaukset voidaan aina myöhemmin ajaa, samalla varmistuen siitä, että regressiota, eli taantumista, eli ole tapahtunut. Automaatiotestien edistyessä projektin manuaalisia resursseja voidaan alkaa enemmin määrin siirtämään sellaisiin toimiin, joissa ihmissilmää ei voida korvata. Manuaalisesta testauksesta voidaan saada suurenevin määrin tutkivaa, uutta testausta, jossa testauksesta saadaan esimerkiksi parannusehdotuksia liittyen ohjelman toimintaan, käyttöliittymän helppokäyttöisyyteen tai koko ohjelman käyttökokemukseen. Automaatiolle voidaan jättää suurin osa mustavalkoisista tarkistustilanteista, joita ohjelmassa täytyy varmistaa, ja muut, ihmisen harkintakykyä vaativat testit, jätetään manuaaliseksi.

Koska automaatiotestit eivät koskaan voi korvata täysin manuaalista testausta, voidaan sanoa, että hyvässä testausstrategiassa ne antavat lisäarvoa toisiinsa. Manuaalinen testaus toimii edelläkävijänä, jossa testitapauksia suunnitellaan sekä toteutetaan pari kertaa suoritettavaksi testaajien toimesta. Automaatiotestauksesta huolehtiva tiimi pystyy sen jälkeen harkitsemaan, mikäli näitä testejä halutaan sekä ylipäätään pystytään automatisoimaan. Tätä sykliä jatkamalla usein automaatiotestaus alkaa suuremmaksi osaksi pystyä tarkistamaan enemmän, kuin vain ohjelman perustoiminnallisuuden. Se pystyy vastaamaan myös rajatapaustarkistuksista sekä ohjelmoijien koodien laadun varmennuksesta, jo ennen niiden liittämistä osaksi koko järjestelmää. Resursseja hyvin järjestämällä tätä etuasemaa voidaan käyttää siihen, ettei manuaalitestaukseen tarvitse enää käyttää niin paljon aikaa sekä rahaa. Tämän ohjelmistokehitysympäristön murroksen hyödyistä on olemassa aitoja tapauksia, jotka auttavat ymmärtämään, miten nämä hyödyt konkretisoituvat kassavirrassa ajan kuluessa.

4.3 Automaation taloudellinen arvo

Automaatiotestauksen taloudellinen hyöty projektissa on lähtökohtaisesti hyvin vaikea arvioida. Testaustapausten luomiseen käytetty aika aiheuttaa ensinnäkin alussa vain kustannuksia. Nämä kustannukset voivat väärin toteutetussa automaatiossa jatkua ilman, että automaatiosta saadaan

muuta kuin laadullista hyötyä. Taloudellisen arvon mittaaminen laadusta on hankalaa, sillä vertaaminen automaation puuttumiseen on hyvin hankalaa. Kun automaation arvoa lähdetään tutkimaan, huomataan, että sillä on lähinnä kolme arvoa tuovaa tekijää.

4.3.1 Automaatio tekee kehityksestä tuottavampaa

Testausautomaation tuoma laadun automaattinen tarkastus aiheuttaa sen, että rikkinäinen koodi huomataan nopeammin. Joissain tapauksissa virheet eivät edes pääse testaustiimille asti, jolloin testauksessa voidaan keskittyä muuhun. Mitä nopeammin virhe huomataan, sitä vähemmän vaurioita se aiheuttaa. Automaatioskriptien tuoma turvaverkko julkaisujen sekä päivittäisten tarkastusten yhteydessä auttaa huomaamaan ongelmia, jotka normaalisti saattaisivat mennä tutkan alta. Mikäli samankaltaiseen testaukseen on käytetty ylimääräisiä manuaalisia resursseja, niitä voidaan alkaa automaation kehittyessä siirtämään sellaisiin testeihin, joita ei vielä ole automatisoitu, tai ylipäänsä kannata automatisoida. Itseään toistavat testitapaukset vähenevät, elleivät jopa poistu manuaalitestaajien hartioilta. Näin sekä testaajien että ohjelmistokehittäjien työaikaa ei enää valu hukkaan hitaiden testaustapausten varmistamisessa.

4.3.2 Automaatio avaa uusia ovia kehitykselle

Testiautomaation valttikortti on aina nopeus. Kun vanhoja testitapauksia aletaan kirjoittamaan automaattiseksi, tietokone pystyy käymään niitä läpi huomattavasti manuaalitestausta nopeammin. Automaatio avaa polun kohti jatkuvaa kehitystä sekä integraatiota, kun testaus ei enää toimi pullonkaulana projektissa, jolloin kehitysnopeus kasvaa. Tämän lisäksi tuloksia saadaan nopeammalla syötteellä, kun automaatiotestit voidaan ajaa esimerkiksi normaalin työajan ulkopuolella. Jatkuvan kehityksen sekä integraation mahdollisuus on juuri se, mikä on tehnyt nykyisistä ohjelmistoprojektien kehitysnopeuksista ylipäänsä todellisuutta [31].

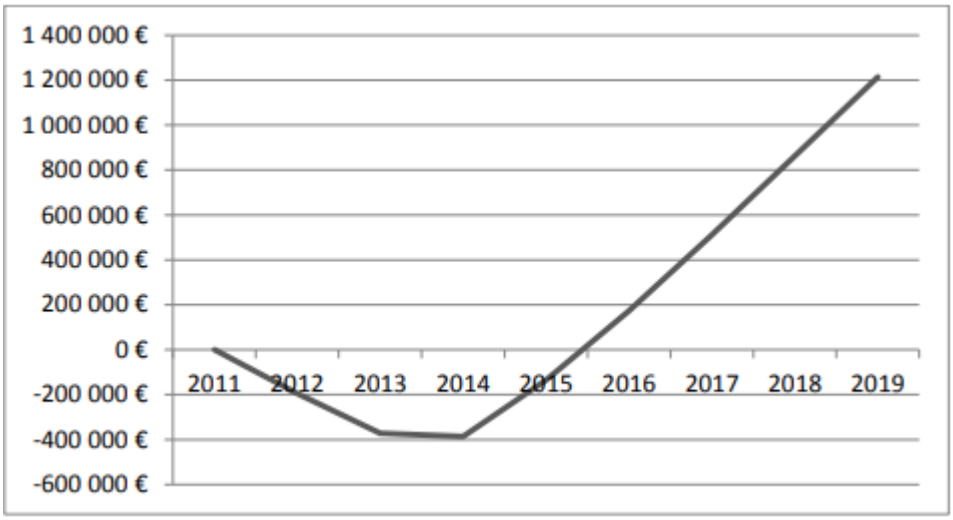
4.3.3 Automaatiolla voidaan vähentää kustannuksia

Tärkeää on kuitenkin huomata, että suorien kustannusten väheneminen itse testaamisen vähentämisessä ei välttämättä ole aina kannattavaa. Testauksen kehittämisessä kannattaa aina liikkua

askel kerrallaan. Automaatiotestien parantuessa ja löytäessä ongelmia entistä useammin säästetään jo kustannuksia niissä virheissä, jotka eivät pääse asiakkaalle asti. Testejä ei jää enää ajamatta ajan vuoksi. Joissain tapauksissa testit voidaan automatisoida niin hyvin, että manuaalitestaajat voivat alkaa siirtyä tutkimuspohjaiseen testaukseen. On kuitenkin hyvä, että manuaalitestaus on olemassa myöskin automaation laadun varmentamiseksi. Joissain tapauksissa automaatio voidaan kuitenkin saada toimimaan niin tehokkaasti, että osa manuaalitestauksen resursseista on jopa mahdollista siirtää toiseen projektiin, joka on kehityksessä aikaisemmassa vaiheessa. Löytämällä hyvän manuaalitestauksen sekä automaatiotestauksen tasapainon, projektissa voidaan saada kaikki hyöty irti molemmista. Kustannusten vähentämisen takana on kuitenkin useita hyviä valintoja, jotka liittyvät sekä automaation toteuttamiseen, että myös siihen tapaan, jolla testauksessa sekä koko projektissa voidaan luottaa testiautomaatioon. Kun testiautomaatioon luotetaan, se voi aidosti vähentää testauksen kustannuksia sekä parantaa koko testauskehityksen tuottavuutta.

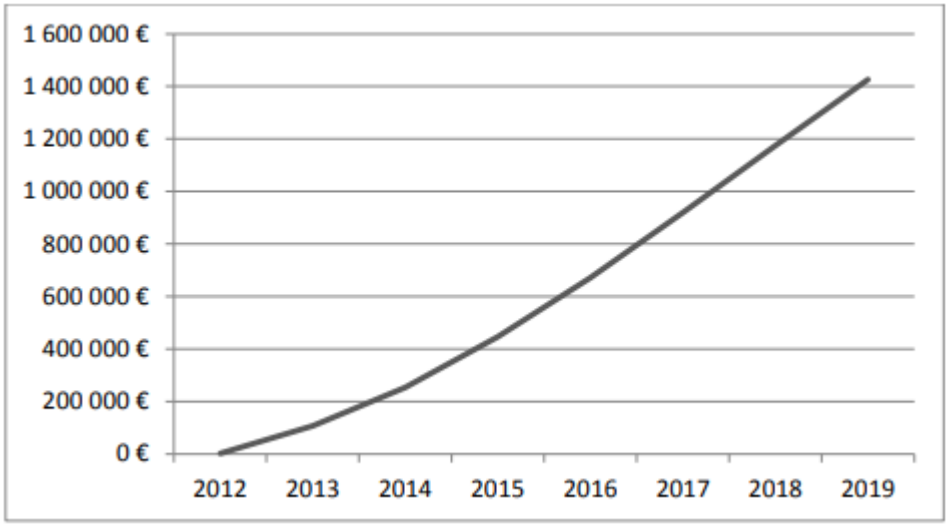
4.3.4 Hyötyjen mittaluokka

Kuten aikaisemmin mainittiin, automaation hyöty ei välttämättä näy suoraan testauksen budjetoinnissa. Sen sijaan, koko kehityksen hinta voi siirtyä alaspäin, vaarallisia virheitä löydetään nopeammin sekä enemmän, ja loppujen lopuksi koko kehityksen laatu paranee, jolloin asiakkaiden luottamus tuotteeseen kasvaa. Tämä tekee automaatiosta mainitusti vaikean mitattavan, mutta hyviä esimerkkejä sen toiminnasta voidaan havaita yritysmaailmassa. Diplomityössään Antti Laapas kuvaa sitä taloudellista hyötyä, jota testikattavuuden lisääminen, aikaisemmin löydettyjen virheiden sekä toistetun työn välttäminen on tuonut eräälle projektille. [32] Tämä hyöty esitetään diskontatun kassavirran**,** eli tulevaisuudessa saatavien rahavirtaerien arvioinnista nykyhetkellä [33]. Tämä kassavirran muutos ajan funktiona esitetään kuvassa 6 [32, s. 100].



Kuva 6. Testiautomaation vaikutus rahavirran kehitykseen eräässä ohjelmistoprojektissa [32, s. 100]

Tärkeää on huomata, että testiautomaation tuonti projektiin aiheuttaa ensimmäisen parin vuoden aikana pienen lommon yrityksen kassavirtaan. Alkuinvestointi kuitenkin pian muuttuu paraabelimaiseksi kasvuksi, jossa testiautomaation edut alkavat säästämään kustannuksia eri tavoin sekä laadun lisääntymisen että myös manuaalitestauksen ajan säästön näkökulmasta. Projektin koosta riippuen testausautomaatio maksaa itsensä takaisin eri ajassa, mutta tärkeää on, että parabolinen kasvu projektin kehityksessä on vahva tuki automaatiotestauksen puolesta. Mitä paremmin projekti menestyy markkinoilla sekä hyötyy automaatiosta, sitä kovempaa kasvua tulevaisuudessa voidaan olettaa automaatiotestauksen antavan projektille. Tärkeää on myös, että tässä esimerkissä testausautomaatio tuotiin projektiin myöhemmin sen aloituksen jälkeen. Koska kehitystyö tapahtui keskellä projektia, siitä syntyneet kustannukset ovat laskettu olevan keskimääräistä suurempia. [32, s. 98.] Automaatiotestauksessa voidaan saada vielä isompia säästöjä, mikäli kaikki testaus alusta alkaen tehdään automaatio huomioiden. Tällaisissa projekteissa Laapas on kuvannut olevan samanlaista kasvukehitystä, mutta ilman laskettua haittaa projektin kassavirralle. Esimerkki vastaavasta projektista voidaan nähdä kuvassa 7 [32, s. 101].



Kuva 7. Kassavirta projektissa, jossa automaatio on huomioitu heti alusta alkaen [32, s. 101]

Näiden tapausesimerkkien pohjalta voidaan siis todeta, että automaatiolla voidaan saada huomattavaa lisäarvoa myös silloin, kun projektia ei alun perin ole mietitty automaation näkökulmasta. Automaation tuoma lisäarvo on aidosti tärkeää kehityksen kannalta rahallisesti mitattuna. Tämä uusii kehityssuunta auttaa yrityksiä tekemään ohjelmistoja nopeammin, turvallisemmin sekä laadukkaammin pitkässä juoksussa. Automaatiosta on tullut tulevaisuuden tapa testata.

5 Esimerkkitapaus

Opinnäytetyön yhteydessä on suoritettu harjoittelu- sekä työajalla testiautomaatio eräässä Bittium Wireless:n ohjelmistoprojektissa. Tämän projektityön aikana tähän ohjelmistoon kehitettiin testiautomaatioympäristö, jossa ohjelman toiminnallisuutta voidaan testata skriptien avulla. Työssä aloitettiin manuaalitestauksesta, jonka aikana projektin eri osa-alueet tulivat tutuiksi. Manuaalitestaus oli hyvä tapa myöskin oppia projektin tavoitteet, haasteet sekä jo valmiiksi luodut testitapaukset sekä tuotteen toiminta. Manuaalitestauksesta oli hyvä väylä lähteä rakentamaan projektiin automaatiota, jossa aikaisemmin luotuja manuaalisia testejä käytettiin referenssinä. Projekti oli tähän opinnäytetyöhön hyvä, sillä tähän ohjelmistoon ei aikaisemmin ollut tehty mitään automaatiotestejä. Tämä antoi mahdollisuuden alkaa luomaan ympäristöä alusta alkaen itse, joka antoi korvaamatonta kokemusta automaatioympäristön suunnittelusta.

5.1 Projektin esittely

Bittium MedicalSuite on Bittiumin kehittämä etäseurantapalvelu, jonka pääasiallinen tarkoitus on mahdollistaa yhteistyö kardiologien, klinikkojen sekä palveluntarjoajien välillä [34]. Käytännössä tuote on verkossa toimiva palvelu, jossa voidaan ylläpitää eri lääketieteellisiä mittauksia, jotka on tehty Bittiumin tarjoamalla Faros -mittalaitteella. Palvelussa on olemassa eri käyttäjätasoja, jotka määrittelevät, mitä kyseinen käyttäjä voi tehdä. Nämä tasot ovat klinikka, diagnostikko, teknikko sekä ylläpitäjä. Yhdessä nämä neljä eri roolia muodostavat työnkulun, jossa klinikka voi luoda mittauksen, johon potilas tuottaa dataa. Mittauksen ollessa valmis se voidaan siirtää diagnostikolle tarkastettavaksi. Diagnostikko puolestaan voi analysoida datan, tehdä päätöksiä datasta ja sen jälkeen antaa raportin, joka palvelun kautta voidaan sitten klinikan puolelta antaa potilaalle.

MedicalSuite sisältää useita eri työkaluja, joilla eri mittauksen kaaren hallinta tehdään. Esimerkiksi MedicalSuite Backoffice on ylläpitäjien työkalu, josta esimerkiksi käyttäjädataa sekä rahavirtaan liittyvää dataa voidaan hallita. MedicalSuite Mobile on puolestaan puhelinsovellus, joka toimii käyttöliittymänä potilaalle, mutta myös MedicalSuite Centerin sekä Mobilen välisenä yhteysketjuna. Tuoteperheen laajuudesta johtuen koko palvelun testaus tulee vaatimaan eri testausmenetelmiä, sillä kaikki tuotteet, vaikkakin ovat osa MedicalSuiten palvelua, eivät toimi samalla alustalla tai edes käyttäjällä. Koko tuotteen testaaminen vaatii siis useamman henkilön testaustyötä,

sekä eri käyttötapausten tarkkaa miettimistä. Koska projektin alussa työ oli enimmäkseen manuaalitestausta, siitä saatiin hyvä kuva sille, miksi tuotteen testaaminen vie niin paljon aikaa. Manuaalitestauksen ajalliset haasteet näkyivät lähinnä järjestelmän testaamisessa sen kaikilla eri oletusarvoilla. Sovellusta tuli pystyä käyttämään useammalla eri verkkoselaimella, kännykkämallilla sekä eri asetuksilla itse mittaukselle. Manuaalitestitapausten määrä on niin valtava, että kaiken tarkka testaaminen käy mahdottomaksi. Vaikka manuaalisen testauksen tärkeys tässä projektissa on ilmiselvä johtuen ohjelmiston käyttäjäläheisyydestä, oli selvää, että projektissa tarvitaan myös automaattisia testejä helpottamaan testauksen työkuormaa.

5.2 Projektin automaatiotestaus

Projektissa päätettiin automaatiotestauksen osalta keskittyä pääosin käyttöliittymän testaukseen. Mobiilisovelluksen testaus on haastavaa johtuen tämän projektin vaatimuksista laitteiston osalta. Kaikista suurin hyöty nähtiin sille, että käyttöliittymän toimivuuteen saadaan lisää luotettavuutta, kun uusia ominaisuuksia sekä korjauksia tuodaan sovellukseen. Käyttöliittymä on keskeinen osa koko järjestelmän toimintaa, jonka vakauden ylläpito on keskeistä kaikkien eri käyttäjätyyppien kesken.

Käyttöliittymän testauksessa päätettiin käyttää Robot Framework -ohjelmistoa, yhdistettynä Selenium -sovelluskirjastoon. Robot Framework on avoimen lähdekoodin ohjelmisto, jonka koodi on ihmiselle helposti luettavaa. Yksinkertainen ohjelmointikieli sekä mahdollisuus ladata lisäominaisuuksia Robot Frameworkiin tekivät siitä hyvän valinnan tälle projektille. Yksi näistä lisäominaisuuksista oli Selenium, jonka avulla Robot Framework kykenee löytämään verkkosivuilta eri käyttöliittymäelementtejä, ja kertomaan, voidaanko esimerkiksi tiettyä nappia painaa testissä. Robotissa ohjelmointi perustuu avainsanoihin, joita yhdistämällä luodaan yksi testitapaus. Testeissä avainsanana voisi esimerkiksi olla, että käyttäjän täytyy pystyä painamaan nappia. Tämä avainsanapohjainen ohjelmointi antaa robotille sen kiitettävän luettavuuden myös projektin muille testaajille. [35.]

Manuaalitestauksesta saadun tiedon perusteella pystyttiin helposti alkaa lajittelemaan testitapauksia niihin, jotka aikaisemman tiedon valossa oli helppo tehdä automaattiseksi, ja taas sellaisiin, joita ei välttämättä haluta heti automatisoida. Näistä automatisoitavista testeistä valittiin esimerkiksi eri selainten testit, sillä automaatiotestit pystyvät nopeasti vaihtamaan selainta ja aja-

maan koko testitapauksen uudelleen eri selaimella. Tämän lisäksi esimerkiksi kaikki testitapaukset, jossa samaa toiminnallisuutta testattiin eri lähtöasetelmilla, oli helposti automatisoitavissa. Helposti automatisoitavissa tapauksissa oli kuitenkin syytä kiinnittää huomiota itse testin sisältöön. Käyttöliittymätestauksessa voitiin helposti tarkistaa nappien sekä kuvien näkyvyys, ja olemassaolo, mutta ei sitä, oliko kuva visuaalisesti näytetty oikein. Samoin myös testitapauksissa oli tärkeää määrittää, milloin tapaus epäonnistuisi. Pelkästään se, että elementtien olemassaolo tarkistetaan, ei esimerkiksi takaa siitä, että niissä olevat arvot olisivat oikeanlaiset. Robot Framework pystyi arvioimaan annetun datan perusteella, oliko esimerkiksi syntymäpäivästä laskettu ikä oikein, mutta ei sitä, oliko eri kieliasetuksen käännös oikeanlainen. Kaikki testitapaukset perustuvat aina testiautomaation luojan olettamukseen siitä, mikä on testissä oikea lopputulos, ja mikä väärä. Tämän vuoksi oli asioita, jotka oli hyvä laittaa jo valmiiksi listalle manuaalitestausta varten.

Automaatiotestauksen heikkoudet kehitysvaiheessa olivat ne testitapaukset, jotka vaativat ihmissilmän harkintakykyä testin tuloksesta. Automaatiossa huomattiin usein, että manuaalisia testitapauksia joutuu muokkaamaan ja myös ajamaan manuaalisesti joiltain osilta johtuen niiden tulkinnanvaraisuudesta. Kaikki tapaukset, jossa kuvailtiin ohjelman visuaalisuutta, käännösten tarkkuutta tai esimerkiksi mittausdatan visuaalista oikeellisuutta, eivät olleet helposti testattavissa. Heikkouksienkin osalta automaatiosta oli kuitenkin hyötyä. Kun nämä automaation heikot osa-alueet tulivat ilmi, saatiin manuaalitestaus keskitettyä näiden ominaisuuksien testaamiseen. Automaatiolle jäi paikka korvaamassa kaiken tylsän työn, kuten eri näppäinyhdistelmien tai rajatapauksien testaamisen.

Automaation edetessä hyöty kasvoi huomattavasti. Automaatiosta saatiin luotettavampaa sekä tarkempaa, kun testaustapausten määrä alkoi kasvamaan. Käyttöliittymän testauksen lisäksi automaatiosta saatettiin joskus hakea myös manuaalitestaajille suoraa apua esimerkiksi tarvittavan mittausdatan luomisessa. Automaatio toimi myös työkaluna manuaalitestaajille, kun se saatiin jokaisen testaajan käyttöön tarvittaessa. Toteutettuna manuaalitestaajien käyttöön, automaatiotestit ovat usein osa skriptiä, jonka ajamalla kuka tahansa testaaja voi ajaa tietyt testit haluamaansa ympäristöä varten. Mikäli testaajalla ei riitä aika ajaa kaikkia testitapauksia, hän voi keskittyä uusiin ominaisuuksiin, ja jättää vanhojen ominaisuuksien testaaminen automaation varaan. Kun uusia päivityksiä julkaistaan, vanhat tapaukset automatisoidaan niin, että tarvittaessa ne voidaan aina tarkistaa päivitysten yhteydessä. Tämän lisäksi automaatio saatiin ajamaan joka yö niin, että mikäli yksikin tarkistus epäonnistui, siitä lähtisi ilmoitus aamulla heti automaatiotiimille, jotka voivat tarkistaa ongelman. Mikäli virhe voidaan toistaa, siitä voidaan heti ilmoittaa kehittäjille,

jotka saavat ongelman korjattua mahdollisimman pikaisesti. Näin luottamus ohjelmiston toimintaan kaikilla ajanhetkillä kasvoi sekä testaus, että myös kehitys sekä johtopuolella.

5.3 Automaation jatkokehitys

Automaatiotestaus on harvoin valmis, sillä uudet ominaisuudet halutaan aina automatisoida siinä missä mahdollista. Automaatiota tullaan kehittämään jatkossa niin, että se kattaa useamman osan testitapauksista. Tämän lisäksi automaatiossa tärkeänä nähdään, että mahdolliset testitulokset ovat selviä. Yksi kehityskohteista on luottamus automaatioon. Tätä luottamusta saadaan näyttämällä visuaalisesti, mitä testeissä oikeasti tapahtuu, ja kuinka paljon koko prosessista on testattu. Visuaaliset tulokset sekä pitkän ajan tarkastelu tulosten tarkkuuden selvittämiseksi lisäävät ennestään luottoa automaatiotestien tuloksiin. Automaation ongelmaksi usein muodostuva pirstaleisuus ei projektissa ole vielä ollut suuri ongelma, mutta se täytyy aina jatkokehityksessä pitää mielessä uusia testejä luotaessa. Pirstaleiset testit ovat suuri ongelma testien luotettavuuteen, ja tämän vuoksi niiden välttäminen on ensiarvoisen tärkeää. Projektissa tullaan varmasti vielä tulevaisuudessakin törmäämään tilanteisiin, jossa automaatiosta halutaan suoraan dataa tai suorituskyvyn mittaamista. Virheettömästi toimiva, luotettava testiautomaatio, jonka tarkastamat kohdat sekä ohittamat kohdat ovat hyvin dokumentoitu antavat loistavan lisän tämän projektin kehitykseen tulevaisuudessa.

5.4 Projektin automaatiokehityksen arviointi

Automaatiosta voidaan lyhyen tapausesimerkin perusteella löytää monia samoja syitä, kuin miksi sitä pidetään teoriassa hyvänä lisänä sovellustestaukseen. Koodin vanhojen ominaisuuksien toiminallisuuden turvaaminen, uudet testitapaukset, yhteistyö manuaalitestauksen kanssa sekä automaattiset testiajot ovat kaikki hyödyllisiä työkaluja ohjelmistotestauksessa. Niiden ansiosta ongelmia saadaan korjattua sekä havaittua nopeammin kuin ennen. Samalla manuaalitestauksen työkaluvarasto on kasvanut, kun skriptit sekä ohjelmointi auttavat eri tilanteiden hallinnassa. Projektista voi selvästi havaita, että automaatiotestaus on hyödyllistä koko projektin testaukseen, eikä se ole vain ja ainoastaan oma osa-alueensa. Luotettavuutta ei voi liikaa korostaa, sillä automaatiotestien käyttöönotossa vaaditaan aina se, että muut testaajat voivat luottaa siitä saatuihin lopputuloksiin. Tässä projektissa luottamus on kasvanut hyvin, mutta jotta automaatiotestaus olisi kannattavaa myös jatkossa, on syytä panostaa siihen, ettei luottamus heikkene.

Automaatiotestauksen lisäys projektiin vie yhden ylimääräisen automaatioinsinöörin työn, mutta samalla projektin testaustaakan kasvaessa projektiin oli kuitenkin pakko hankkia lisää käsipareja. Tästä voidaan päätellä, että omalla tavallaan testiautomaatio on ollut pelkkää plussaa. Projektissa on saatu uudelleenajettavia, luotettavia testejä, jotka varmistavat ohjelmiston toiminnan pohjan. Tähän asti ohjelmistossa on löydetty hyvin vikoja, jotka on saatu korjattua. Automaatio itsessään on myös useita kertoja huomannut palvelun ongelmia, kuten esimerkiksi sen, että tietokanta on alhaalla. Näiden huomaus pikaisesti yöllä suoritettujen testiajojen jälkeen on antanut testaustiimille mahdollisuuden päästä pikaisesti testaamaan muita ongelmia. Testien uudelleenkäytettävyyden, projektille tuotujen uusien testimahdollisuuksien sekä nopeammin löydettyjen ongelmien vuoksi voidaan todeta, että automaatiosta on ollut positiivista vaikutusta myös kassavirran osalta. Tämä robottitestien uudelleenkäytettävyys tulee maksamaan itsensä myös takaisin tulevaisuudessa useita kertoja, kun kehitys jatkuu testitapausten osalta. Kaikki automaatioon laitettu jää voimaan, ja vaikka pieniä korjauksia joskus vaaditaan testien ylläpitoon, testitapausten kehitys pitää huolen siitä, ettei ohjelman päivittyessä manuaalitestaukselta vahingossa jää tarkastamatta sitä yhtä ominaisuutta, joka juuri silloin oli mennyt hajalle.

Kehityksen aikana löytyneitä parannusehdotuksia on kuitenkin tulevaisuutta varten. Koska työnkuva oli tekijälle uusi testien kehittämisen alkuvaiheilla, ei luottamus testituloksiin sekä testien helppo muokkautuvuus ollut aluksi tarpeeksi hyvin huomioitu. Tulevaisuudessa testien kehitys tullaan aina tekemään niin, että niitä voidaan myöhemmin helposti muokata mahdollisimman paljon. Toiminta usealla eri testiympäristöllä olisi pitänyt ottaa heti alusta huomioon, jotta testien korjaukseen ei olisi mennyt niin paljon aikaa. Samalla vanhentuneet kehitysstandardit ovat alkaneet muuttua parempaan päin, joten automaation kehityksessä on tärkeää, että nämä standardit pidetään mukana. Joissain tapauksissa Robot Frameworkia oli käytetty myös sellaisissa tapauksissa, jossa se ei ollut paras työkalu tilanteeseen. Automaation kehityksessä on hyvä tulevaisuudessa pitää myös mielessä se, että oikein valittu työkalu tekee tehtävän suorituksesta huomattavasti helpompaa. Kehityksen aikana myös dokumentaation tärkeys on korostunut, ja sen puute aikaisemmista testeistä on muuttunut parempaan päin uusien testitapausten myötä. Vanhat, huonommin dokumentoidut testit olisivat kuitenkin hyvä päivittää, jotta mahdollisesti myöhemmin automaatiota jatkokehittävät henkilöt pääsisivät nopeammin kiinni sen toimintaan.

6 Yhteenveto

Testiautomaation tuominen osaksi ohjelmistoprojektia on iso päätös, jolla on kuitenkin useita valtavan positiivisia vaikutuksia testauksen laatuun. Hyvin toteutettu automaatio ottaa huomioon sekä projektin vaatimukset, että myös automaation hyödyt sekä haitat. Samalla tärkeää on myös koko organisaation muutos. Koko testiympäristön on otettava automaatio vastaan, mikäli halutaan pysyviä positiivisia muutoksia testauksen laadussa. Nämä kaikki seikat saattavat vaikuttaa yritykselle pelottavalta, kun mietitään sitä, minkälaisia kustannuksia ja riskitekijöitä testiautomaation käyttöönottoon liittyy.

Tärkeintä on ymmärtää, että testiautomaatio ei ole vain sitä, että ohjelmistolle tehdään erityisiä automaatiotestejä testi-insinöörien toimesta. Automaatio on myös ohjelman läpikäyntiä nopeiden, valmiiksi luotujen työkalujen avulla. Se on testauksen malli, jossa tietokonetta käytetään testauksen avustajana. Testausautomaation tuonti projektiin voi alkaa hyvinkin pienistä askeleista. Aletaan tarkistamaan koodi automaattisilla työkaluilla, luomaan automaattisia testiraportteja ja sitten lopulta suunnittelemaan ensimmäistä täysin automatisoitua testitapausta. Nämä työkalut ovat kaikki kehittyneet paljon viimeisen vuosikymmenen aikana, ja niiden lisäys projektiin on helpompaa kuin koskaan. Testausta helpottavia työkaluja kannattaa kokeilla, sillä niiden kehitys on, kuten edellä mainittu, mahdollistanut entistä nopeammin tahdin sovellusten kehittämisessä.

Tapausesimerkistä voidaan jo päätellä, ettei automaation lisäys projektiin vaadi aina kokonaista tiimiä. Nykyajan ohjelmistoilla pienetkin testaustiimit, myös yksittäiset henkilöt, voivat kehittää testiautomaatiota projekteissa. Saatu hyöty tämän yhden testaajan panoksella on huomattava kaikista eri testauksen näkökulmista. Hyvässä projektissa tämä uusi testaustyyppi lisätään osaksi vahvaa, osaavaa manuaalisen testauksen tiimiä, jotka itsekin hyötyvät uusista luoduista työkaluista sekä automaation tarkkuudesta. Automaatiosta tulee nopeasti osa hyvän ohjelmistotestaajan työkalupakkia, josta saatu hyöty voidaan valjastaa koko yrityksen käyttöön. Ohjelman ongelmat löytyvät nopeammin, virheistä koituneet kustannukset pienenevät. Vähemmän vakavia ongelmia pääsee läpi testauksesta, ja yrityksen maine sekä yrityskuva paranevat. Testausautomaatio ei ole nollasummapeli, vaan se antaa kaikille jotain. Liittyminen osaksi automaatiota hyödyntävien yritysten joukkoa vaatii rohkeutta, mutta automaatiosta usein tulee irrottamaton osa sovellustestauksen toimivuutta. Tekoälyn kehittyessä pidemmälle automatisoidut testit tulevat tekemään vielä useammasta nykyään manuaalisesta testistä nopeampaa, tarkempaa sekä tehokkaampaa, ja tästä saatua hyötyä ei kukaan halua ohittaa.

Lähteet

1. Harley, N. (29.5.2018). 11 of the most costly software errors in history. Viitattu 25.3.2021. Saatavilla osoitteessa https://raygun.com/blog/costly-software-errors-history/

2. Celerity. The true cost of a software bug. Viitattu 25.3.2021. Saatavilla osoitteessa https://www.celerity.com/the-true-cost-of-a-software-bug

3. AMCAT. (1.8.2018) Testing Engineer – Fresher/Entry Level. Viitattu 25.3.2021. Saatavilla osoitteessa https://www.aspiringminds.com/hr-insights/featured-profiles/testing-engineer-fresher-entry-level/

4. Lakshay, S. (3.9.2019) Why is Testing Necessary? Viitattu 25.3.2021. Saatavilla osoitteessa https://www.toolsqa.com/software-testing/istqb/why-is-testing-necessary/

5. Stacey K, Waldmeir P. (17.3.2019) FAA defends safety procedures after Boeing 737 Max crashes. Financial Times. Viitattu 25.3.2021. Saatavilla osoitteessa https://www.ft.com/content/fcd85bde-48d0-11e9-bbc9-6917dce3dc62

6. Gates, D. (22.11.2020) Q&A: What led to Boeing's 737 MAX crisis. The Seattle Times. Viitattu 28.4.2021. Saatavilla osoitteessa https://www.seattletimes.com/business/boeing-aerospace/what-led-to-boeings-737-max-crisis-a-qa/

7. International Software Test Insitute. Software Testing Roles and Responsibilities. Viitattu 4.4.2021. Saatavilla osoitteessa https://www.test-institute.org/Software_Testing_Roles_And_Responsibilities.php

8. Armour F, Miller G. Advanced Use Case Modeling: Software Systems. Addison-Wesley; 2000. ISBN-13: 9780201615920

9. Luukkainen, M. (2020). Ohjelmistotuotanto, osa 2. Viitattu 9.5.2021. Saatavilla osoitteessa https://ohjelmistotuotanto-hy.github.io/osa2/

10. Rice, Randall W. (20.10.2020). Viitattu 9.5.2021. Write a test plan. Saatavilla osoiteessa https://www.practitest.com/qa-learningcenter/best-practices/write-a-test-plan/

11. Cania, L. (23.10.2019). Cania Consulting. Viitattu 4.4.2021. Saatavilla osoitteessa https://cania-consulting.com/2019/10/23/test-planning-and-test-plan/

12. Omar, A. (8.5.2018). Understand the difference between Test Plan and Test Strategy document. Viitattu 4.4.2021. Saatavilla osoitteessa https://medium.com/@anujomar1a2/understand-the-difference-between-test-plan-and-test-strategy-document-cdbf35ba40df

13. Montvelisky, J & Bhamere, L. (2018). The State of Testing™ Report 2018. PractiTest & Tea-Time with Testers. Viitattu 5.4.2021. Saatavilla osoitteessa https://qablog.practitest.com/wp-content/uploads/2018/07/2018_state_of_testing_report_1.2.pdf

14. Homès B, Homès B. Fundamentals of Software Testing. Hoboken: John Wiley & Sons, Incorporated; 2012. s. 45;47. ISBN-13: 9781118603093

15. Myers GJ, Sandler C, Badgett T. The Art of Software Testing. Hoboken: John Wiley & Sons, Incorporated; 2011. s. 111. ISBN-13: 9781118133132

16. Joensuun yliopisto, tietojenkäsittelytieteen laitos. Integrointi- ja järjestelmätestaus. Viitattu 5.4.2021. Saatavilla osoitteessa http://cs.joensuu.fi/tSoft/testaus.htm#integrointitesti

17. Savonia-ammattikorkeakoulu. Ohjelmiston testaus ja laatu – testaustasot. Viitattu 5.4.2021. Saatavilla osoitteessa http://webd.savonia.fi/home/ktrasse/muut/testaus_laatu/testaus_3.pdf

18. Katara M. (2011). Ohjelmistojen testaus. Tampereen teknillinen yliopisto. Viitattu 5.4.2021. Saatavilla osoitteessa http://www.cs.tut.fi/~tie21201/s2011/luennot/OHJ-3060_2011_110-170.pdf

19. Holm J. (4.4.2011). Dissecting the V Model. Viitattu 11.4.2021. Saatavilla osoitteessa https://jockeholm.wordpress.com/2011/04/04/dissecting-the-v-model/

20. Rana K. (11.5.2020). What is Software Testing? Viitattu 11.4.2021. Saatavilla osoitteessa https://artoftesting.com/what-is-software-testing

21. Brush, K. (2020). Test Case. Viitattu 1.5.2021. Saatavilla osoitteessa https://searchsoftwarequality.techtarget.com/definition/test-case

22. Pollari, J. (2014). Ohjelmistotestaus. Opinäytetyö. Saatavilla 11.4.2021 https://www.theseus.fi/bitstream/handle/10024/85400/Pollari_Jukka.pdf?sequence=1&isAllowed=y

23. Testim. (7.11.2019). What Is a Test Environment? A Guide to Managing Your Testing. Viitattu 12.4.2021. Saatavilla osoitteessa https://www.testim.io/blog/test-environment-guide/

24. Honka, J. (12.12.2017). Graafisten käyttöliittymien testaus ja testausviitekehykset. Kandidaattityö. Viitattu 14.4.2021. Saatavilla osoitteessa http://urn.fi/URN:NBN:fi:jyu-201712144703

25. Padmini, C. (2016). Beginners Guide To Software Testing. Viitattu 16.4.2021. Saatavilla osoitteessa https://www.softwaretestingclass.com/wp-content/uploads/2016/06/Beginner-Guide-To-Software-Testing.pdf

26. Qentinel. Mitä on testausautomaatio, ja miten teet siitä liiketoimintasi vauhdittajan? Viitattu 17.4.2021. Saatavilla osoitteessa https://value.qentinel.com/fi/mita-on-testausautomaatio

27. GitLab Docs. Flaky tests. Viitattu 18.4.2021. Saatavilla osoitteessa https://docs.gitlab.com/ee/development/testing_guide/flaky_tests.html

28. Partanen, P. (2009). Laadukkaan ohjelmistotestauksen piirteet. Tutkintotyö. Viitattu 18.4.2021. Saatavilla osoitteessa http://urn.fi/URN:NBN:fi:amk-201003064971

29. Tricentis Team. (23.12.2018). The Myth of Automated Exploratory Testing. Viitattu 18.4.2021. Saatavilla osoitteessa https://www.tricentis.com/blog/the-myth-of-automated-exploratory-testing/

30. Software Engineering, Software Testing. Viitattu 19.4.2021. Saatavilla osoitteessa https://cs.ccsu.edu/~stan/classes/CS410/Notes16/08-SoftwareTesting.html

31. Sacolick, I. (17.1.2020). What is CI/CD? Continuous integration and continuous delivery explained. InfoWorld. Viitattu 19.4.2021. Saatavilla osoitteessa https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

32. Laapas, A. (2014). Cost-benefit analysis of using test automation in the development of embedded software. Pro-gradu-työ. Viitattu 19.4.2021. Saatavilla osoitteessa http://urn.fi/URN:NBN:fi-fe2014060526268

33. Pankkiasiat.fi. Diskontattu kassavirta. Viitattu 19.4.2021. Saatavilla osoitteessa

https://pankkiasiat.fi/diskontattu-kassavirta

34. Bittium. Bittium MedicalSuite™ - Remote Monitoring Service Platform. Viitattu 22.4.2021. Saatavilla osoitteessa https://www.bittium.com/medical/bittium-medicalsuite

35. Robot Framework Foundation. Robot Framework Introduction. Viitattu 25.4.2021. Saatavilla osoitteessa https://robotframework.org/