Aleksi Mäkeläinen

TOIMINNANOHJAUSJÄRJESTELMIEN TESTAAMINEN

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Mäkeläinen, Aleksi
Toiminnanohjausjärjestelmien testaaminen
Jyväskylä: Jyväskylän yliopisto, 2020, 28 s.
Tietojärjestelmätiede, kandidaatin tutkielma
Ohjaaja: Kyppö, Jorma

Toiminnanohjausjärjestelmät ovat yritysten liiketoimintaprosesseja tukevia tietojärjestelmiä, jotka kokoavat yrityksen prosessien tuottamat tietovirrat yhteen keskitettyyn tietojärjestelmään. Niiden kehitys on mukaillut liiketoimintaympäristössä tapahtuneita yleisiä muutoksia. Näistä yksi tällä hetkelläkin tapahtuva muutos on ohjelmistojen siirtyminen tarjottavaksi pilvipalveluina. Pilvipalveluna tuotettua ohjelmistoa ylläpitää sen kehittänyt yritys omilla palvelimillaan, jolloin sen ylläpitovastuu siirtyy ohjelmiston käyttäjältä sen tuottajalle. Ylläpitokustannusten vähentämisen takia ohjelmistotuottajat pyrkivät yhtenäistämään ylläpidossa olevia järjestelmäversioita pakottamalla niiden käyttäjät päivittämään ohjelmistojaan säännöllisesti. Nykyään uudet toiminnanohjausjärjestelmät toimitetaan pääsääntöisesti pilvipalveluina, jolloin niiden päivitystiheys yleistyy verrattuna aikaan, jolloin niitä ylläpidettiin niitä käyttävien organisaatioiden toimesta. Toiminnanohjausjärjestelmien keskeisen roolin vuoksi päivitysten mahdollisesti aiheuttamilla virheillä voi olla merkittävä vaikutus yrityksen toimintaan. Ohjelmistotuotannon tärkein laadunvalvontamenetelmä on testaus, joka on keskeisin keino virheiden havaitsemiseen ohjelmistossa. Tämä tutkielma on kirjallisuuskatsaus, joka lähdeaineiston perusteella pohtii toiminnanohjausjärjestelmien testaamiseen vaikuttavia asioita ja kuinka toiminnanohjausjärjestelmien testaaminen muuttuu pilvipalveluiden yleistyessä. Tutkielman perusteella voidaan todeta, että pilvipalveluna toteutettuihin toiminnanohjausjärjestelmiin siirtyminen tulee oletettavasti lisäämään testauksen vaatimia resursseja. Syynä tähän voidaan nähdä toiminnanohjausjärjestelmien keskeinen rooli yritysten liiketoimintaprosessien mahdollistajana, joka vaatii koko järjestelmän laajuisia monimutkaisia testejä.

Asiasanat: Toiminannanohjausjärjestelmä, pilvipalvelut, testaaminen, regressiotestaaminen

ABSTRACT

Mäkeläinen, Aleksi
Testing of Enterprise resource planning systems
Jyväskylä: University of Jyväskylä, 2020, 28 pp.
Information systems, Bachelor's thesis
Supervisor: Kyppö, Jorma

Enterprise resource planning (ERP) systems are information systems that support the business processes of the organizations using them by gathering all the data created by the business processes of the organization to one centralized information system. They have evolved as a part of broader changes in the business environment and technology. Recently organizations have started using more software as a service, and following this trend, almost all new ERP systems are delivered as a service. Software that is used as a service is software that is maintained by the software vendor and is located on the vendor's servers. To reduce upkeep costs of the software the software vendors try to minimize the number of different versions they need to maintain and sometimes they force the users to update their systems. This leads to increased updates of ERP software compared to before when the ERP software was maintained by the organization using it. Updating software has a change of introducing different malfunctions, and because of the role of ERP systems in the day to day operations of the organization, they must work properly. Testing is the most important quality assurance method and an essential tool for finding these possible issues in the software. This thesis is a literature review that focuses on testing of ERP systems and the changes that software as a service will introduce to them. As a result of the review, one can say that the trend of software as a service should lead to an increase in the resources needed for testing. ERP systems are complex systems which means that conducting system-wide tests for them is complicated, and as a result of increased updates, the organizations will need to test the systems more often.

Keywords: Enterprise resource planning system, software as a service, testing, regression testing

KUVIOT

SISÄLLYS

1 JOHDANTO

Toiminnanohjausjärjestelmät ovat yrityksen lisäarvoa tuottavia prosesseja tukevia järjestelmiä, jotka integroivat yrityksen liiketoimintaprosessit yhteen keskitettyyn tietojärjestelmään. Yritysten jokapäiväisen toiminnan kannalta ne ovat erittäin kriittisiä järjestelmiä, joiden ongelmat voivat heijastua koko yrityksen toimintaan. Toisaalta hyvin onnistunut toiminnanohjausjärjestelmäprojekti ja sitä kautta hyvin yrityksen toimintoja tukeva toiminnanohjausjärjestelmä voi tuoda yritykselle merkittäviä säästöjä (Umble, Haft & Umble, 2003).

Mukaillen yleistä teknologista kehitystä myös useat uudet ja päivitettävät toiminnanohjausjärjestelmät toteutetaan nykyään pilvipalveluna, jossa ohjelmisto sijaitsee ohjelmiston tuottajan ylläpitämillä palvelimilla. Pilvipalvelussa ohjelmiston samaa ja hyvin standardoitua koodipohjaa jaetaan samaan aikaan usealle eri asiakkaalle (Sun, Zhang, Guo, Sun & Su, 2008). Tämä mahdollistaa samalla järjestelmätoimittajille tehokkaamman keinon oman tuotteensa jakamiseen. Järjestelmätoimittajat pyrkivätkin yhtenäistämään ylläpitämiään järjestelmäversioita, jotta he pystyvät vähentämään ohjelmistoista aiheutuvia ylläpitokustannuksia. Esimerkiksi ohjelmistovirheiden korjaus voidaan tehdä kaikille asiakkaille samaan aikaan. Tämän takia järjestelmätoimittajat myös pyrkivät samalla yhtenäistämään ylläpidossa olevia järjestelmäversioita mahdollistaen vielä laajempien hyötyjen saavuttamisen. Vaikka ohjelmistojen käyttäjät pystyvät pääsääntöisesti päättämään ohjelmistopäivitysten asentamisesta, vaativat useat järjestelmätoimittajat kuitenkin asiakkaitaan päivittämään ohjelmistoversionsa säännöllisesti.

Tämän kehityksen osana toiminnanohjausjärjestelmien ohjelmistopäivitysten määrä lisääntyy merkittävästi samalla lisäten testaamiseen käytettävää aikaa. Ennen toiminnanohjausjärjestelmät eivät sijainneet ohjelmiston tuottajan palvelimilla ja niiden päivittämiselle ei välttämättä ole ollut pakottavaa tarvetta. Nykyään kuitenkin esimerkiksi Microsoft päivittää omaa pilvipohjaista toiminnanohjausjärjestelmää kahdeksan kertaa vuodessa pakottaen toiminnanohjausjärjestelmien käyttäjät päivittämään vähintään joka neljäs uusi ohjelmaversio (Microsoft, 2019).

Tämän tutkielman tutkimuskysymys on

- Kuinka toiminnanohjausjärjestelmien monimutkaisuus ja pilvipalveluiden yleistyminen vaikuttavat testaamiseen?

Lisäksi avustavina tutkimuskysymyksinä toimivat käsitteiden määritelmät

- Mikä on toiminnanohjausjärjestelmä?
- Mitä on testaaminen?

Tutkimus toteutettiin kirjallisuuskatsauksena hyödyntäen Templierin ja Paren (2015) menetelmäviitekehystä. Kirjallisuutta haettiin JYKDOK –tietokantaa ja Google Scholar –palvelua hyödyntäen alla olevilla avainsanoilla ja niiden yhdistelmillä.

- enterprise resource planning system (ERP)
- software as a service (SaaS)
- testing
- software testing
- regression testing

Lähteiden valinnassa pyrittiin painottamaan julkaisun ajankohtaisuutta ja viittausmäärää. Lisäksi lähteet pyrittiin valitsemaan lähes pelkästään korkeatasoisista julkaisukanavista. Testaamisen lähdemateriaalia valitessa ajankohtaisuutta painotettiin vähemmän, koska teknologian kehityksestä huolimatta useat vanhat julkaisut ovat vielä ajankohtaisia. Voidaan jopa sanoa, että ohjelmistojen ollessa ennen paljon yksinkertaisempia on niistä myös helpompi hahmottaa testaamisen perusperiaatteita. Toisaalta toiminnanohjausjärjestelmien testaamisesta löytyvän lähdemateriaalin valinnassa jouduttiin käyttämään löysempiä kriteereitä, sillä siitä on verrattain vähemmän tutkimusmateriaalia kuin muista tutkielman aihealueista.

Tutkielma koostuu johdannosta ja kolmesta sisältöluvusta. Tämän lisäksi viimeisessä kappaleessa esitetään tutkielman yhteenveto ja pohditaan sen tuloksia. Ensimmäisessä sisältöluvussa määritellään toiminnanohjausjärjestelmän käsite ja perehdytään toiminnanohjausjärjestelmien kehitykseen ja pohditaan niiden muuttumista yleisten teknologisten muutosten mukana. Toisessa sisältöluvussa määritellään testaamisen käsite ja pohditaan sen tavoitteita ja käytännön toteutusta. Lisäksi perehdytään eroihin testaamisessa ohjelman eri arkkitehtuuritasoilla. Kolmas ja viimeinen sisältöluku keskittyy toiminnanohjausjärjestelmien testaamiseen.

2 TOIMINNANOHJAUSJÄRJESTELMÄT

Tässä luvussa määritellään toiminnanohjausjärjestelmän käsite. Tämän lisäksi luvussa käydään läpi toiminnanohjausjärjestelmien kehitys osana liiketoimintaympäristön muutosta, joka on johtanut toiminnanohjausjärjestelmien siirtymiseen tuotettavaksi pilvipalveluina.

2.1 Toiminnanohjausjärjestelmät yrityksissä

Toiminnanohjausjärjestelmät ovat pohjimmiltaan integroituja tietojärjestelmiä, jotka tukevat yritysten lisäarvoa tuottavia prosesseja. Toisin sanoen ne ovat laajoja tietojärjestelmäkokonaisuuksia, jotka yhdistävät taloushallinnon, toimitusketjun hallinnan ja asiakkuudenhallinnan prosessit yhteen järjestelmään (Ruivo, Oliviera & Neto, 2012). Mahdollisista prosesseista, joita toiminnanohjausjärjestelmät voivat yhdistää, on Gerrard (2007) maininnut tuotannonhallinnan, toimitusketjunhallinnan, taloushallinnon, projektienhallinnan, henkilöstöhallinnon sekä asiakkuudenhallinnan. Tämän lisäksi ne voivat hänen mukaansa toimia yrityksen tietovarastona (Gerrard, 2007). Yhdistettäviä ohjelmamoduuleita tai käyttötarkoituksia voi olla tätäkin enemmän, mutta tärkeimpänä toiminnanohjausjärjestelmää määrittelevänä tekijänä voidaan pitää niiden toimintaa yritysten eri prosessien tietovirtoja yhdistävänä kokonaisuutena. Yksittäiset prosessit tai niiden osat voidaan suorittaa erillisissä järjestelmissä, jolloin niiden tuottama tieto siirretään toiminnanohjausjärjestelmään.

Al-Shardan ja Ziani (2015) ovat antaneet esimerkin mahdollisesta toiminnanohjausjärjestelmän moduulikokoonpanosta (kuvio 1). Kuvio on yksinkertaisuudessaan hyvä esimerkki toiminnanohjausjärjestelmän yhdistävästä luonteesta, mutta sitä tutkiessa tulisi huomioida, että jokainen ulkoinen moduuliyksikkö voi pitää sisällään useita eri prosesseja ja alimoduuleita. Riippuen yrityksen liiketoimintaprosesseista sen ei myöskään tarvitse käyttää jokaista toiminnanohjausjärjestelmän tarjoamaa ohjelmamoduulia.

KUVIO 1 Toiminnanohjausjärjestelmien moduulit (Al-Shardan & Ziani, 2015, s. 95)

Klausin, Rosemannin ja Gablen (2000) mukaan toiminnanohjausjärjestelmä on konseptina kolmeosainen. Heidän mukaan se voidaan ensinnäkin nähdä tietokoneohjelmana. Toiseksi se on tavoitetila, jossa kaikki organisaation prosessit ja data ovat yhdessä kokoavassa rakenteessa. Kolmanneksi heidän mukaansa toiminnanohjausjärjestelmä voidaan nähdä infrastruktuurin avainelementtinä, joka tarjoaa yritykselle ratkaisun (Klaus, Rosemann & Gable, 2000). Johansson ja Ruivo (2013) ovat taas määritelleet toiminnanohjausjärjestelmän yhdistäväksi ohjelmistopaketiksi, jossa on yksi jaettu tietokanta, joka tukee liiketoimintaprosesseja yrityksen koosta riippumatta. He lisäävät, että toiminnanohjausjärjestelmät koostuvat useasta erilaisesta funktionaalisesta moduulista, jotka mukailevat yrityksen osastojen rakennetta (Johansson & Ruivo, 2013).

Yhdistävästä luonteestaan johtuen toiminnanohjausjärjestelmien kriittisyys yritysten liiketoimintaprosessien mahdollistajana on erittäin suuri. Ongelmat toiminnanohjausjärjestelmien toiminnassa voivat johtaa suuriin taloudellisiin menetyksiin. Esimerkiksi Vänskä (2017) mainitsee Oriolan uuden toiminnanohjausjärjestelmän käyttöönoton ongelmien johtaneen miljoonien eurojen tappioihin ja lääkkeiden toimitusongelmiin. Toisaalta Hunton, Lippincott ja Recki (2003) mainitsevat uuden toiminnanohjausjärjestelmän käyttöönotto voivan tuoda yritykselle hyvin merkittäviä taloudellisia hyötyjä. Umblen, Haftin ja

Umblen (2003) mukaan hyvin onnistunut toiminnanohjausjärjestelmäprojekti voi tuoda merkittäviä säästöjä luomalla parempia kysyntäennusteita, nopeuttamalla tuotantosyklejä ja parantamalla asiakaspalvelua. Toiminnanohjausjärjestelmien käyttöönotosta ovat Hunton, Lippincott ja Reck (2003) todenneet, että se ei aina tuota suoraan havaittavissa olevia hyötyjä, mutta vertaillessaan toiminnanohjausjärjestelmiä käyttäviä yrityksiä sellaisiin, jotka eivät niitä käytä, he ovat todenneet käyttöönoton hyötyjen olevan merkittäviä. Ero yritysryhmien välillä selittyy pääosin sillä, että toiminnanohjausjärjestelmiä käyttämättömien yritysten suoriutuminen markkinoilla on selvästi huonompaa. (Hunton, Lippincott & Reck, 2003).

2.2 Toiminnanohjausjärjestelmien kehittyminen

Toiminnanohjausjärjestelmät ovat kehittyneet ensin tuotantoyritysten tarpeisiin mukaillen liiketoimintaympäristössä ja teknologian kehittymisessä tapahtuneita yleisiä muutoksia. Tämän kehityksen alkuaikoina järjestelmiä kutsuttiin tuotannonohjausjärjestelmiksi, joita käyttivät tuotantoyritykset. Tuotannonohjausjärjestelmät keskittyivät lähinnä isojen varastomassojen tehokkaaseen hallintaan, koska vielä koska vielä 1960-luvulla yrityksillä oli varaa pitää tuotteita varmuuden vuoksi varastossa. (Umble, Haft & Umble, 2003). Keskeisenä tuotannonohjausjärjestelmien tavoitteena oli yritysten aineellisten resurssien hallinnointi.

Tuotannonohjausjärjestelmät kehittyivät lisääntyneiden toiminnallisuuksien myötä vaiheittain toiminnanohjausjärjestelmiksi, jotka keskittyvät aineellisten resurssien hallinnan lisäksi myös yritysten aineettomien resurssien hallintaan. Tätä kehitystä on ajanut sekä tietotekniikan yleistyminen että yritysten toimintaympäristöissä tapahtunut kasvava kilpailu. 1980-luvulla tuotannonohjausjärjestelmät alkoivat kattamaan taloushallinnon prosesseja ja 1990-luvun alussa ne kattoivat jo koko yritysten erilaisten resurssitarpeiden hallinnoinnin, eivätkä ne enää keskittyneet pelkästään aineellisten resurssien hallintaan (Umble, Haft & Umble, 2003). Tuotantoyritykset eivät täten enää olleet ainoita yrityksiä, jotka näistä tietojärjestelmistä pystyivät hyötymään. Samalla näiden yritysten prosesseja yhdistävien järjestelmien nimeksi vakiintui toiminnanohjausjärjestelmä.

Perinteisesti toiminnanohjausjärjestelmät ovat olleet niin sanottuja on-premise ratkaisuja, joita on ylläpidetty ja hallinnoitu toiminnanohjausjärjestelmää käyttävän organisaation toimesta (Johansson & Ruivo, 2013). Viimeisen muutaman vuoden aikana ohjelmistojen ja palveluiden toimittaminen pilvipalveluina on yleistynyt merkittävästi. Tämä on myös heijastunut toiminnanohjausjärjestelmien toimittamiseen, ja kolme suurinta toiminnanohjausjärjestelmätoimittajaa Microsoft, Oracle ja SAP ovatkin siirtyneet tarjoamaan pääsääntöisesti pilvipohjaisia ratkaisuja (Elbahri ym., 2019). Pilvipalvelut voidaan jakaa kolmeen eri kategoriaa, joissa tuotetaan infrastruktuuri, alusta tai ohjelmisto palveluna. Näistä käytetään yleisesti termejä IaaS (Infrasturcture as a Sercive),

PaaS (Platform as a Service) ja SaaS (Software as a service). Toiminnanohjausjärjestelmien tapauksessa puhutaan ohjelmiston tarjoamisesta palveluna.

Xin ja Levinan (2008) mukaan pilvipalveluna tuotettu ja tarjottu ohjelma on ohjelma, jonka sama koodipohja on käytössä usealla eri asiakkaalla, ja sitä jaetaan palveluna internetin välityksellä. Ohjelmaa ylläpidetään jatkuvasti sen tuottaman ohjelmistoyrityksen toimesta (Seethamraju, 2015). Saas-mallissa asiakas ei osta perinteiseen tyyliin ohjelmistoon kertalisenssiä vaan maksaa siitä kuukausittaista käyttöhintaa esimerkiksi käyttäjälukumäärän mukaan. Koska ohjelmistotuottajan ei tarvitse ylläpitää kuin yhtä yhteistä koodipohjaa on tuottajan mahdollista tehokkaamman ohjelmistotuotannon ja lisensointimalli mahdollistaa toimittajaorganisaatiolle tasaisemman tulovirran. Näiden lisäksi Xi ja Levina (2008) mainitsevat toimittajan hyödyksi sen, että heidän ei tarvitse enää olla vastuussa yrityskohtaisten muutosten ylläpidosta. Saas-mallin voidaankin sanoa yrittävän toistaa perinteisessä tuotannossa suuressa roolissa olevia skaalaetuja (Sun ym., 2008)

Ohjelmistotuottajien Saas-mallista saamat hyödyt näkyvät ohjelmistojen kuluttajille halvempina hintoina. Erityisesti toiminnanohjausjärjestelmistä puhuttaessa Peng ja Gala (2014) toteavat, että pilvipohjaisien toiminnanohjausjärjestelmien ylläpidon ollessa kolmannen osapuolen vastuulla keventää se päivitysten asennuksesta aiheutuvaa kuormitusta organisaation sisäisille IT-palveluille, joka mahdollistaa järjestelmien päivittämisen useammin. Verrattuna vanhoihin on-premise järjestelmiin he toteavat, että niiden käyttöönotto sekä päivittäminen taas voi olla hyvinkin kallista ja työlästä. (Peng & Gala, 2014).

3 TESTAAMINEN

Tässä luvussa määritellään testaamisen käsite ja käydään läpi testauksen eri vaiheet. Tämän lisäksi käydään läpi kuinka ohjelman arkkitehtuurin taso, jossa testausta tehdään, vaikuttaa testaamiseen. Lopuksi käsitellään järjestelmän uudelleentestaamista, josta käytetään yleisesti nimitystä regressiotestaaminen.

3.1 Testaamisen käsite

Testaaminen on keskeinen osa ohjelmistokehitystä ja sen keskeisin laadunvalvonnan väline. Yksinkertaisimmillaan se on ohjelman suorittamisen seuraamista, jotta voidaan varmistua ohjelman toimivan suunnitellulla tavalla ja samalla tunnistaa mahdollisia virheitä (Bertolino, 2007). Sawant, Bari ja Chawan (2012) ovat määritelleet testaamisen prosessiksi, jonka tarkoituksena on löytää ohjelmistosta mahdollisia virheitä ja täten varmistua sen laadusta. Whittaker (2000) erottaa testaamisen toisesta ohjelmiston laadunvarmistusmenetelmästä, koodin tarkistuksesta, toteamalla testaamisen perustuvan koodin suorittamiseen. Koodin tarkistuksessa ohjelman lähdekoodia luetaan ja analysoidaan sitä kuitenkaan suorittamatta (Whittaker, 2000).

Goodenough ja Gerhart (1975) ovat todenneet ideaalin testin olevan sellainen, joka suoritetaan hyväksytysti vain, mikäli ohjelmisto ei sisällä yhtäkään virhettä. Heidän tutkimuksensa jälkeen ohjelmistot ja tietotekniikka on monimutkaistunut valtavasti ja nykyään voitaneen pitää mahdottomana koko ohjelmiston kattavien ideaalien testien rakentamista. Toisaalta juuri tuon ajan ohjelmien yksinkertaisuus tuo selkeämmin esille ohjelmistotestauksen perusperiaatteita, koska ne eivät huku valtavien ohjelmistokoodimassojen sekaan.

Goodenough ja Gerhart (1975) ovat jakaneet ohjelmistossa esiintyvät virheet kahteen eri kategoriaan, logiikkavirheisiin ja suorituskykyvirheisiin. Suorituskykyvirheeksi kutsutaan virhettä, jonka vuoksi ohjelmaa suoritetaan epäsuotuisalla tavalla, joka johtaa ohjelman suorituskyvyn heikentymiseen. Tietokoneiden suorituskyvyn eksponentiaalisen kasvun myötä suorituskykyvirheiden

merkitsevyyden voidaan olettaa pienentyneen. Toisaalta mikäli tietokoneiden suorituskyvyn kasvu tulevaisuudessa hidastuu voivat nämä ongelmat tulla uudestaan ajankohtaisiksi. Huolimattoman resurssien käytön voidaankin jo huomata olevan suhteellisen yleistä, joka vielä tällä hetkellä piiloutuu tietokoneiden suorituskyvyn kasvamisen taakse.

Logiikkavirheet voidaan jakaa Goodenoughin ja Gerhartin (1975) mukaan neljään eri alakategoriaan ylätason tarpeiden suunnittelusta varsinaiseen ohjelmoimiseen riippuen siitä, missä kohtaa toiminnallisuuden toteutusta se on ilmennyt. (kuvio 2). Testaamisen tulisi pystyä löytämään virheitä jokaisesta logiikkavirhekategoriasta. (Goodenough & Gerhart, 1975). Testausta suunniteltaessa ja testitapauslistoja rakennettaessa tulisikin pyrkiä ottamaan huomioon kaikki nämä kategoriat.

KUVIO 2 Logiikkavirheet mukaillen Goodenoughia ja Gerhartia (1975)

Sawant, Bari ja Chawan (2012) ovat löytäneet ohjelmistotestaukselle viisi tavoitetta, joiden perusteella testaamista tulisi tehdä. Ensinnä testauksen pitäisi pystyä määrittelemään toimiiko ohjelma niin kuin on määritelty. Toiseksi testauksen tulisi olla tehokasta huomioiden kustannukset ja aikatauluhaasteet. Kolmanneksi testauksen tulisi pystyä tasapainottamaan määrittelyt, tekniset haasteet sekä käyttäjien odotukset. Neljänneksi testaus tulisi dokumentoida hyvin, jotta jälkikäteen on mahdollista todeta testien olevan tehdyt, ja uusia testejä tarvitse enää tehdä uudelleen. Viimeiseksi ja tärkeimmäksi tavoitteeksi he listaavat sen, että testaamisen tulisi olla tarkoituksenmukaista. Testaajien tulisi

tietää miksi testataan, mikä on sen päämäärä, ja mitkä ovat mahdolliset testauksen lopputulokset (Sawant, Bari & Chawan, 2012).

Virheiden määrää voidaan vähentää kattavalla testaamisella, mutta johtuen nykyajan ohjelmistojen laajuudesta virheitä esiintyy lähes aina julkaistuissa ohjelmistoissa huolimatta testauksen määrästä ja laadusta. Esiintyneisiin virheisiin Whittaker (2000) on löytänyt neljä syytä. Hänen mukaansa käyttäjä on voinut suorittaa testaamatonta koodia tai toisaalta suorittaa sitä eri järjestyksessä kuin testaaja. Toisaalta käyttäjän määrittelemät syöttöarvot ohjelmalle ovat myös voineet olla sellainen yhdistelmä, jota ei ole testattu. Viimeisenä mahdollisena virheen löytämisen syynä hän mainitsee mahdollisuudeksi sen, että käyttäjän järjestelmäkokoonpano on ollut sellainen, jota ei ole testattu. (Whittaker, 2000).

Whittakerin (2000) mainitsemista syistä voidaan olettaa viimeiseksi mainitun muuttuvan harvinaisemmaksi pilvipalveluiden käytön yleistyessä yrityksissä. Aikaisemmin ohjelmistotuottajilla ei ole ollut suoraa kontrollia järjestelmäympäristöistä, joissa heidän ohjelmaansa suoritetaan. Samalla käytettävien ohjelmistoversioiden määrä on saattanut olla suuri, mikä on muodostanut yhdessä järjestelmäympäristöjen erilaisuuden kanssa erittäin suuren määrän testattavia kokoonpanoja. Pilvipalveluina käytettäviä ohjelmia kuitenkin suoritetaan ohjelmistotuottajien omilla palvelimilla, jolloin ohjelmistotuottaja pystyy itse määrittelemään käyttöympäristöt. Tämän lisäksi ohjelmistotuottaja pystyy mahdollisesti pakottamaan käyttäjiä uudempaan versioon poistaen tarpeen testata vanhoja ohjelmistoversioita.

Nykyajan ohjelmistojen monimutkaisuudesta johtuen Goodenoughin ja Gerhartin (1975) mainitsemia ideaalisia testejä ei käytännöstä pystytä tekemään ennen ohjelmiston julkaisemista. Tällöin ohjelmistosta löydetyt virheet tulee korjata. Whittaker (2000) on antanut ohjelmiston korjaamiselle neljä mahdollista lopputulosta (kuvio 3). Havaittu alkuperäinen virhe voidaan hänen mukaansa onnistua korjaamaan tai se on vielä korjauksenkin jälkeen ohjelmistossa. Riippumatta korjauksen onnistumisesta on myös mahdollista rikkoa samalla jokin toinen toiminnallisuus (Whittaker, 2000).

Korjaa virheen Ei riko muita toiminnallisuuksia	Ei korjaa virhettä Ei riko muita toiminnallisuuksia
Korjaa virheen Rikkoo toisen toiminnallisuuden	Ei korjaa virhettä Rikkoo toisen toiminnallisuuden

KUVIO 3 Virheen korjauksen mahdolliset seuraukset mukaillen Whittakeria (2000)

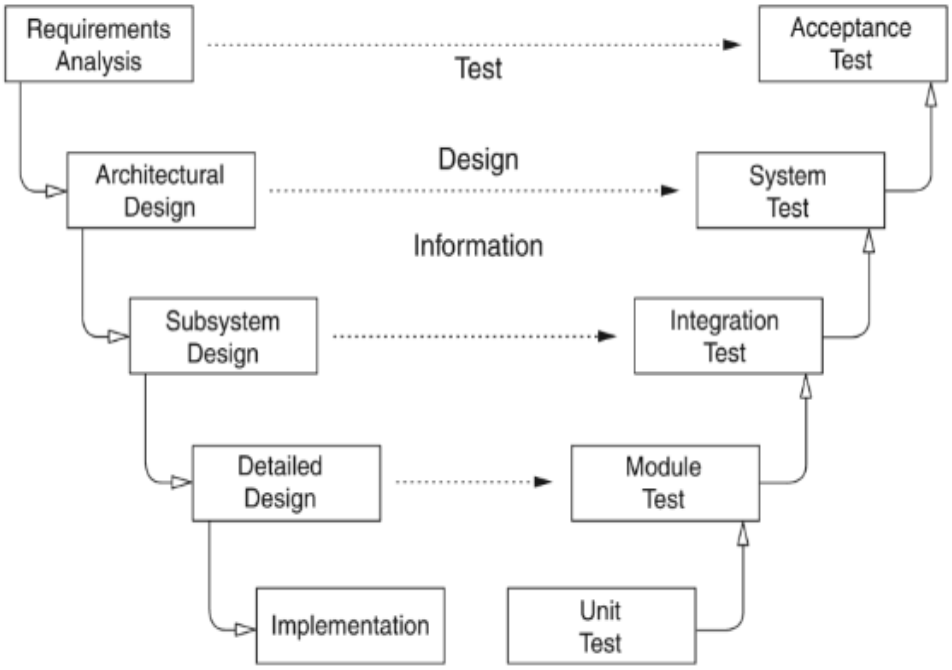
3.2 Testauksen vaiheet

Testaus voidaan jakaa yleisesti kolmeen eri kokonaisuuteen. Nämä kolme kokonaisuutta ovat ohjelmaan syötettävät arvot ja niistä saatavat ulostuloarvot, testattava ohjelma ja testioraakkeli. Testioraakkeliksi kutsutaan konetta tai ihmistä, joka annettujen syöttöarvojen perusteella osaa päätellä onko ohjelman suorituksen jälkeen saadut ulostuloarvot oikeat (Shahamiri, Kadir & Mohd-Hashim, 2009). Testattavalle ohjelmalle annetaan syöttöarvot, joista ohjelman suorituksen jälkeen saadaan näihin syöttöarvoihin liittyvät ulostuloarvot. Testioraakkelin tehtävänä on tämän jälkeen määritellä toimiiko järjestelmä oikein.

Testaus eroaa merkittävästi riippuen siitä, millä ohjelmistoarkkitehtuurin tasolla sitä tehdään. Osalle näistä testaustyypeistä on vakiintunut yleinen nimitys, ja esimerkiksi yksikkötestaus on käsitteenä laajasti tunnettu. Whittakerin (2000) mukaan yksikkötestauksessa ohjelman komponentteja testataan erikseen erillään muusta järjestelmästä. Tämän lisäksi Whittaker (2000) on jaotellut testaamisen integraatiotestaamiseen ja järjestelmätestaamiseen. Integraatiotestauksessa aiemmin testattuja yksittäisiä komponentteja testataan isompana kokonaisuutena, jotta varmistutaan niiden yhteistoiminnasta. Integraatiotestaus siis keskittyy yksittäisten komponenttien välisen tiedonsiirron oikeellisuuden varmistamiseen.

Järjestelmätestauksen Whittaker (2000) on jaotellut kahteen eri kategoriaan, funktionaaliseen ja rakenteelliseen testaamiseen. Funktionaalinen järjestelmätestaus testaa koko järjestelmää välittämättä testattavana olevan ohjelmiston koodipohjasta keskittyen vain tiettyjen käyttötapausten suorittamiseen hyväksytysti. Tässä testaustavassa ei siis ollenkaan tarkastella kyseisen ohjelmiston koodia. Rakenteellinen testaus taas keskittyy kokonaan ohjelman koodin analysointiin koodin rakenteesta johtuvien ongelmien, esimerkiksi suorituskykyongelmien, havaitsemiseen. (Whittaker, 2000). Funktionaalista järjestelmätestausta voi myös suorittaa asiakas, joka tulee käyttämään ohjelmistoa. Tällöin siitä käytetään nimitystä hyväksyntätestaus.

Amman ja Offutt (2016) ovat määritelleet testaustyypit yleisesti V-malliksi kutsutun mallin mukaan. He käsittelevät hyväksyntätestausta omana testaustyyppinään sekä määritelevät integraatiotestausta alemmaksi tasoksi moduulitestauksen, jossa pienempää määrää komponentteja testataan keskenään moduuleittain. (kuvio 4). Yhteistä Whittakerin (2000) sekä Ammanin ja Offutin (2016) käsityksestä erityyppisistä testaamisista on niiden siirtyminen alemman tason yksikkötestauksesta kohti koko järjestelmän toimintaa koskevaa testausta.



KUVIO 4 Ohjelmistotestauksen eri tasot (Amman & Offutt, 2016, s. 23.)

Oraakkeliongelmaksi kutsutaan sitä haastetta, miten testioraakkeli pystyy erottamaan halutun ulostuloarvon virheellisestä ulostuloarvosta. Testauksen syöttöarvojen ja testattavan osa-alueen toiminta ovat hyvin erilaisia eri testauksen tasoilla, ja sitä kautta myös oraakkeliongelman laajuus muuttuu. Mitä ylemmälle tasolle ohjelman arkkitehtuurissa päädytään, sitä monimutkaisemmat ovat mahdollisten syöttöarvojen yhdistelmät. Samalla, kun syöttöarvojen yhdistelmät lisääntyvät ja järjestelmän toiminta monimutkaistuu, vaaditaan testioraakkelilta enemmän, jotta se pystyy määrittelemään testituloksen perusteella toimiiko ohjelma halutulla tavalla.

Testioraakkeli voi olla joko siihen tarkoitukseen rakennettu ohjelma tai ihminen. Mitä monimutkaisempi testattava järjestelmä on, sitä todennäköisempää on ihmisen toimiminen testioraakkelina. Alemmilla arkkitehtuuritasoilla syöttöarvojen automatisointi sekä niiden virheiden havaitsemisen kannalta optimaalisten syöttöarvojen valitsemin on jo laajasti käytössä ja yksikkötestien automatisointi onkin tuttua jokaiselle ohjelmoijalle. Korkeammilla arkkitehtuuritasoilla testioraakkelina toimii käytännössä aina ihminen, jonka lisäksi sama henkilö on yleensä vastuussa syöttöarvojen määrittelemisestä. Baresi ja Young (2001) ovat maininneet ihmisoraakkelin käytön hyödyksi sen, että ihminen on parempi määrittämään testien oikeellisuuden puutteellisesta ohjeistuksesta huolimatta. Toisaalta huonoina puolina heidän mukaansa ovat ihmisoraakkelin huomattavasti korkeammat kustannukset ja tarkkuus etenkin suuria testimääriä käsiteltäessä. (Baresi & Young, 2001).

3.3 Regressiotestaaminen

Regressiotestaaminen on testaamista, joka toteutetaan ohjelmiston päivityksen yhteydessä, ja siinä käydään uudelleen läpi ohjelmiston kehitysvaiheen testitapaukset (Whittaker, 2000). Wieczorek, Stefanescu ja Schieferdecker (2008) ovat määritelleet sen ohjelmiston testaamiseksi koodin lisäämisen jälkeen, jotta voidaan varmistua, että lisätty koodi ei riko mitään aikaisempaa toiminnallisuutta. Regressiotestausta tehdään jo ohjelmiston kehitysvaiheessa eri kehitysversioiden päivitysten yhteydessä. Kattava testitapauslista mahdollistaa suunnitelmallisen ohjelman laadunvalvonnan.

Jo pieni muutos yhdessä osassa ohjelmistoa voi Ammanin ja Offuttin (2015) mukaan mahdollistaa muutosten ilmaantumisen muihin osiin ohjelmistoa, vaikka ne eivät suoraan liittyisikään toisiinsa. Heidän mukaansa regressiotestaamisen ensisijainen päämäärä onkin näiden ongelmien löytäminen (Amman & Offutt, 2015). Leungin ja Whiten (1989) mukaan regressiotestaus sisältää muunnellun ohjelmiston testaamista testitapauksilla varmistaen sen, että ohjelma toimii muutostenkin jälkeen määritellysti. Näiden tapausten vuoksi onkin tärkeää, että ohjelma on määritelty riittävän tarkasti, jolloin nämä ongelmat on mahdollista havaita.

Leung ja White (1989) mainitsevat regressiotestauksen jakautuvan kahteen erilaiseen tyyppiin, jotka ovat korjaava ja progressiivinen regressiotestaaminen. Korjaava regressiotestaamista tehdään koodin lisäämisen jälkeen, kun sen alkuperäinen toimintalogiikka ei ole muuttunut, jolloin se toimii ohjelmiston alkuperäisen määrittelyn mukaan. Tällöin vanhoja testitapauksia voidaan käyttää hyväksi testaamisessa. Progressiivinen regressiotestaaminen taas sisältää päivityksen yhteydessä muuttuneen määrittelyn, jolloin ohjelmiston voidaan olettaa käyttäytyvän eri tavalla. (Leung & White, 1989). Tämän jaottelun perusteella vanhojen testitapausten käyttäminen on paljon helpompaa korjaavan regressiotestaamisen tapauksessa.

Wongin, Horganin, Londonin ja Agrawalin (1997) mukaan regressiotestaamisessa voidaan suorittaa joko kaikki edellisen ohjelmaversion testaamista varten tehdyt testitapaukset tai vain tietty osa niistä. Ketterän kehityksen yleistyessä ja ohjelmistojen koon kasvaessa testitapausten kokonaismäärä voi nykyään kasvaa hyvinkin suureksi, jolloin koko testitapauslistan läpikäyminen voi muodostua hyvin kalliiksi ja aikaa vieväksi. Tähän vaikuttaa erityisesti se, millä tasolla ohjelmistoarkkitehtuuria testausta tehdään, koska eri ohjelmistoarkkitehtuurien tasolla tapahtuvassa testaamisessa on yleensä käytettävissä eri määrä automaatiota. Pääsääntöisesti alemman mitä alempana ohjelman arkkitehtuuria testataan, sitä enemmän automaatiota on käytettävissä. Regressiotestaamisen haasteeksi voidaankin todeta olevan se, kuinka on mahdollista poimia alkuperäisestä testitapauslistasta sellainen otos, jolla pystytään havaitsemaan mahdolliset virheet mahdollisimman tehokkaasti ja täten vähentämään ylemmän arkkitehtuuritasojen testauksen viemää resurssimäärää.

Rothermel, Untch, Chu ja Harrold (2001) ovat todenneet, että regressiotestaaminen voi aiheuttaa jopa puolet julkaistun ohjelmiston ylläpitokustannuksista. Regressiotestaamisen tavoitteena voidaankin pitää näiden kustannusten minimointia kuitenkin samalla pitäen huolen tarpeeksi kattavien testien suorittamisesta. Kustannusten minimoinnin ovat myös Sawant, Bari ja Chawan (2012) määritelleet yhdeksi testauksen tavoitteeksi, ja tämän lisäksi heidän mielestään testaus tulisi dokumentoida hyvin. Testauksen kattava dokumentointi mahdollistaa vanhojen testitapausten toistamisen helpommin. Yoo ja Harman (2012) mainitsevat regressiotestaamisen kulujen vähentämisen mahdollisiksi ratkaisuiksi testitapausten minimoinnin, testitapausten valitsemisen ja testitapausten priorisoinnin.

Testitapauksen minimoinnissa pyritään löytämään mahdollisimman kattava ja tehokas otos testitapausjoukosta. Siinä pyritään löytämään ja poistamaan sellaisia testitapauksia, jotka ovat vanhentuneet tai eivät enää ole oleellisia (Yoo & Harman, 2012). Testitapausten valitsemisessa tavoitteena on sama lopputulos, mutta se eroaa minimoinnista käytännön toteutukseltaan. Yoon ja Harmanin (2012) mukaan minimoinnissa pyritään ennalta määriteltyjen kriteerien perusteella määrittelemään valittu otos, kun taas valitseminen keskittyy enemmän ohjelmistoversioiden välisiin eroihin ja pyrkii tätä kautta valitsemaan halutut testitapaukset.

Rothermel ym. (2001) ovat todenneet, että testitapausten valitsemisen ja minimoinnin seurauksena virheiden havaitseminen ei yleensä merkittävästi heikkene. Heidän mukaansa tämä ei kuitenkaan aina pidä paikkaansa. He ovat ehdottaneet minimionnin ja valitsemisen sijaan ratkaisuksi testitapausten priorisointia, jossa testitapauksille määritetään tärkeysjärjestys jonkun ennalta määritellyn kriteerin perusteella. Priorisoinnissa testitapauksia ei poisteta testitapauslistalta, mutta mikäli testaus jostain syystä keskeytyy ennenaikaisesti esimerkiksi aikataulu- tai kustannussyistä pystytään ainakin kriittisimmät testitapaukset käymään läpi (Rothermel ym., 2001).

4 Toiminnanohjausjärjestelmien testaaminen

Tämä luku käsittelee toiminnanohjausjärjestelmien testaamiseen vaikuttavia asioita. Luvussa tarkastellaan testaamisen lisäksi toiminnanohjasjärjestelmien räätälöintiä, jossa toiminnanohjausjärjestelmän toimintaa muutetaan asiakasyrityksen toiveita vastaavaksi ohjelman koodia muuttamalla. Lopuksi käsitellään toiminnanohjausjärjestelmien regressiotestaamista.

4.1 Toiminnanohjausjärjestelmien räätälöinti

Yritysten liiketoimintaprosessit voivat erota hyvin paljon toisistaan ja samalla eri yrityksillä on erilaisia tarpeita, joita toiminnanohjausjärjestelmän käytöllä pyritään täyttämään. Nämä eroavaisuudet johtuvat toiminnanohjausjärjestelmien käytöstä eri aloilla ja eri maissa, jonka seurauksena liiketoiminnan prosessien vaatimukset ja säännökset voivat olla hyvinkin erilaisia. Tämä lisäksi toiminnanohjausjärjestelmiä käyttää nykyään hyvin suuri joukko erikokoisia yrityksiä, kun vielä 1990-luvun alussa toiminnanohjausjärjestelmät olivat lähinnä vain suurimpien yritysten käytössä.

Uudet toiminnanohjausjärjestelmät tuotetaan pääasiassa pilvipalveluina, joissa ohjelmiston samaa koodipohjaa jaetaan usealle eri asiakkaalle. Tästä käytetään myös nimitystä multitenancy arkkitehtuuri. Tässä arkkitehtuurissa ohjelmiston toimittaja pyrkii tekemään mahdollisimman standardimaisen ohjelman, jonka ohjelmakoodia ei tarvitse muuttaa asiakkaiden vaatimusten takia (Sun ym., 2008). Asiakkailla on kuitenkin omia vaatimuksia ohjelman toiminnan suhteen, joka johtaa siihen, että ohjelmistosta löytyy laajat parametrit, joilla ohjelmiston toimintaa voidaan hallita. Parametreja on tehty myös yksittäisten maiden säännösten tarpeisiin. Tätä parametrien kautta tapahtuvaa toiminnanohjausjärjestelmän toiminnan muuttamista ilman ohjelmakoodin lisäämistä kutsutaan konfiguroinniksi (Al-Shardan & Ziani, 2015).

Parametrien ollessa riittämättömät voivat asiakkaat halutessaan muuttaa toiminnanohjausjärjestelmän toimintaa lisäämällä siihen ohjelmakoodia, jonka

lisäämistä ei yleensä tee ohjelmiston alkuperäinen toimittaja. Tästä toiminnanohjausjärjestelmän toiminnan muutoksesta koodia lisäämällä käytetään nimitystä kustomointi tai räätälöinti. (Al-Shardan & Ziani, 2015). Tämän lisäksi Dittrich, Vaucoleur ja Giff (2009) pitävät myös muiden järjestelmien liittämistä toiminnanohjausjärjestelmään räätälöintinä.

Parthasarathy ja Sarma (2016) ovat tutkimuksessaan todenneet, että yleensä mahdollisimman vähän räätälöity toiminnanohjausjärjestelmä on kustannustehokkuudeltaan selvästi parempi. Syyksi toiminnanohjausjärjestelmän räätälöinnille he kuitenkin mainitsevat todennäköisesti sen, että erityisesti suurissa yrityksissä liiketoimintaprosessien muuttaminen koetaan sisäisesti hankalammaksi toteuttaa kuin toiminnanohjausjärjestelmän räätälöinti (Parthasarathy & Sarma (2016). Voidaan yleisesti todeta, että toiminnanohjausjärjestelmien räätälöinnit kasvattavan järjestelmän monimutkaisuutta. Nykyään toiminnanohjausjärjestelmät ovat tämän lisäksi yhä useammin käytössä pilvipalveluina. Tällaisena tuotetun ohjelman toimittaja ei enää ole vastuussa asiakaskohtaisten muutosten toimivuudesta (Xi & Levina, 2008). Tämän voidaan olettaa lisäävän pilvipalveluina toimitettujen ja paljon räätälöityjen toiminnanohjausjärjestelmien testaamiseen kuluvaa aikaa.

4.2 Toiminnanohjausjärjestelmien testaaminen

Gerrard (2007) on tunnistanut toiminnanohjausjärjestelmien käyttöönottoprojekteista seitsemän vaihetta. Vaiheisiin kuuluvat ohjelman määrittely, ohjelman räätälöinti, infrastruktuurin kehitys, yhdistäminen vanhoihin järjestelmiin, datamigraatio, toiminnan muutos ja testaaminen. Hän huomauttaa, että toiminnanohjausjärjestelmäprojektien ollessa laajoja ja yrityksille kriittisiä niiden testaamiseen käytetty aika lähentelee puolta projektin kokonaiskustannuksista. Hänen mukaansa tutkimusta aiheesta on kuitenkin verrattain vähän (Gerrard, 2007). Toisaalta toiminnanohjausjärjestelmien testaamisessa voidaan hyödyntää samoja periaatteita kuin muidenkin ohjelmistojen testauksessa (Al-Hossan & Al-Mudimigh, 2011).

Wieczorekin ja Stefanescun (2010) mukaan näistä ohjelmistotestauksen periaatteista ymmärretään parhaiten alemman tason yksikkötestaaminen. Voidaan kuitenkin olettaa, että toiminnanohjausjärjestelmien testaamisessa ylempien arkkitehtuuritasojen testaaminen muodostaa tärkeämmän kokonaisuuden ja on tämän takia isommassa roolissa johtuen niiden roolista yrityksen prosesseja tukevana järjestelmänä. Wieczorek, Kozyra, Schur ja Rothi (2012) ovatkin todenneet toiminnanohjausjärjestelmien testaamisen tärkeimmän tavoitteen olevan yrityksen liiketoimintaprosessien testaaminen. Liiketoimintaprosessien testaaminen tarkoittaa järjestelmän laajuista testaamista ja se voidaan olettaa tehtävän manuaalisesti, koska yritysten liiketoimintaprosessit eroavat toisistaan merkittävästi riippuen yrityksestä ja sen toimialasta. Tällöin sen tekee käytännössä yrityksen kyseisen alueen erikoisosaaja tai ulkopuolinen konsultti. Tehtä-

essä koko järjestelmää käsitteleviä laajoja testejä voidaan olettaa niiden käytännön toteutuksen vievän paljon resursseja.

Whittaker (2000) on jakanut testaamisen arkkitehtuuritasoittain yksikkötestaukseen, integraatiotestaukseen ja järjestelmätestaukseen. Ammanin ja Offutin (2015) käyttämä jako on hyvin samanlainen, tosin he ovat lisänneet yksikkötestauksen ja integraatiotestauksen väliin moduulitestauksen ja lisäksi he pitävät asiakkaan suorittamaa hyväksyntätestausta aina osana järjestelmätestausta. Verrattuna näihin kahteen yleisesti ohjelmistoja käsittelevään testausjakoon Gerrard (2007) on tunnistanut toiminnanohjausjärjestelmäprojektien sisältävän yksikkötestaukset, alijärjestelmien integraatiotestaukset, järjestelmien väliset integraatiotestaukset ja hyväksyntätestaukset. Näiden jakojen perusteella toiminnanohjausjärjestelmien testausta voitaneen käsitellä kolmena erilaisena kokonaisuutena. Ensimmäinen kokonaisuus sisältää alimman tason yksikkötestaamisen, toinen kokonaisuus on erilaisten komponenttien ja järjestelmien yhteistestaaminen ja viimeinen kokonaisuus on koko järjestelmän laajuinen hyväksyntätestaus.

Yksikkötestaus on ohjelmiston yksittäisten komponenttien testaamista erillään muusta järjestelmästä (Whittaker, 2000). Uudet toiminnanohjausjärjestelmät toimitetaan nykyään poikkeuksetta pilvipalveluina, jolloin ohjelmakoodia ylläpitää ohjelmiston toimittaja omalla palvelimellaan (Elbahri ym., 2019). Gerrardin (2007) mukaan ohjelmistojen tuottajilla on omat tapansa testata ohjelmistoja, mutta ne keskittyvät lähinnä yleisesti ohjelman toimintaan. Pilvipalvelumallin myötä ohjelmiston toimittaja ei ole vastuussa mahdollisista asiakasyrityksen tekemistä räätälöinneistä, jolloin niiden yksikkötestaaminen on asiakasyrityksen vastuulla. Yksikkötestaus on tärkeä osa myös toiminnanohjausjärjestelmien testaamista, mutta toiminnanohjausjärjestelmien kohdalla se ei merkittävästi eroa muiden ohjelmistojen testaamisesta. Yksikkötestauksen keskittyessä määriteltyjen algoritmien oikeanlaiseen toteutukseen niiden havaitsemien ongelmien korjaaminen on toiminnanohjausjärjestelmien laajuuden huomioon ottaen suhteellisen yksinkertaista.

Siirryttäessä yksikkötestaamisen jälkeen ohjelmistoarkkitehtuurin ylemmille tasoille toiminnanohjausjärjestelmien monimutkaisuus alkaa vaikuttamaan merkittävästi testaamiseen. Komponenttien ja järjestelmien integraatiotestauksessa erilaisten komponenttien, moduulien ja alijärjestelmien määrä voi kasvaa huomattavan suureksi. Toiminnanohjausjärjestelmät voivat Schliesserin (2007) mukaan olla yhteydessä useisiin erilaisiin legacy järjestelmiin tai jopa muihin toiminnanohjausjärjestelmiin. Toiminnanohjausjärjestelmää pienempien ohjelmakokonaisuuksien tapauksissa eri komponentit voivat sijaita saman ohjelman sisällä tai niistä on korkeitaan muutama yhteyksiä eri järjestelmiin.

Kun kaikkien alijärjestelmien ja moduulien yhteydet on saatu testattua, on viimeisenä toiminnanohjausjärjestelmien testausvaiheena hyväksyntätestaus. Whittaker (2000) on määritellyt hyväksyntätestauksen koko järjestelmän laajuiseksi funktionaaliseksi testaukseksi, jonka suorittaa ohjelmistoa käyttävä asiakas. Tässä vaiheessa testataan toiminnanohjausjärjestelmän kautta kulkevien liiketoimintaprosessien toimivuus. Wieczorek ym. (2012) pitävät hyväk-

syntätestausta merkittävämpänä toiminnanohjausjärjestelmän laadunvarmistuskeinona.

Hyväksyntätestaus vaatii useiden eri järjestelmien ja liiketoimintaprosessien tuntemista, joten voidaan olettaa, että se tehdään lähes aina ihmisen toimesta. Lisäksi testaukseen osallistuu useampi eri liiketoimintaprosessien asiantuntija, sillä jo yksi parametrimuutos järjestelmässä voi vaikuttaa siihen miten usea eri liiketoimintaprosessin perusteella rakennettu käyttötapaus toimii. (Schliesser, 2007). Tämän lisäksi toiminnanohjausjärjestelmien räätälöinti vaikuttaa järjestelmien monimutkaisuuteen ja tätä kautta se vaikuttaa testaamisessa käytettäviin syöttöarvoihin sekä sitä kautta lisää testioraakkelilta vaadittua ymmärrystä järjestelmän toiminnasta.

4.3 Toiminnanohjausjärjestelmien regressiotestaaminen

Toiminnanohjausjärjestelmät ovat perinteisesti olleet niin sanottuja on-premise ratkaisuja, jolloin niitä ylläpidetty ohjelmaa käyttävän organisaation toimesta (Johansson & Ruivo, 2013). Samalla ohjelmaa käyttävällä organisaatiolla on ollut suurempi mahdollisuus vaikuttaa järjestelmän päivitysten yleisyyteen ja niiden laajuuteen. Ohjelmaa ei välttämättä ole tarvinnut päivittää useaan vuoteen, jos ohjelmaa käyttävä organisaatio ei ole nähnyt sitä tarpeelliseksi. Tämän takia toiminnanohjausjärjestelmiä ei ole testattu niiden käyttöönottoprojektien jälkeen kovinkaan useasti.

Pilvipalveluna tuotetussa toiminnanohjausjärjestelmässä ohjelmaa käyttävä organisaatio ei enää voi itse päättää päivitysten ajankohdasta vaan toiminnanohjausjärjestelmien toimittajat määrittelevät milloin järjestelmä tulisi päivittää (Bjelland & Haddara, 2018). Ohjelmistotuottajat pyrkivät vähentämään tuettujen ohjelmistoversioiden määrää, joka helpottaa ohjelmistojen ylläpitoa niiden näkökulmasta. Tämä käytännössä johtaa päivitysten yleistymiseen, joka taas johtaa siihen, että toiminnanohjausjärjestelmiä tulee tulevaisuudessa regressiotestata useita kertoja vuodessa.

Ohjelmistotuottajien suorittamalla alemmalla arkkitehtuuritasolla, jossa luodaan yksikkötestejä sekä moduulien välisiä yksinkertaisia testejä, voidaan käyttää laajasti hyödyksi vanhoja testitapauksia. Yksikkötestien tapauksessa ne on myös yleensä automatisoitu. Ylemmillä arkkitehtuuritasoilla toiminnanohjausjärjestelmien testaaminen, ja sitä kautta myös regressiotestaaminen, on työlästä ja hyvin usein manuaalista. Oman haasteensa toiminnanohjausjärjestelmien testaamiseen tuottaa se, että testauksessa testitapausten tulisi olla toistettavissa mahdollisimman samanlaisina (Wieczorek, Stefanescu & Schiefendecker, 2008). Johtuen toiminnanohjausjärjestelmien monimutkaisuudesta niiden testauksessa ei välttämättä voi käyttää samoja esimerkkitapahtumia useaan kertaan, vaan ne tulee joka kerta luoda uudelleen. Tämä mahdollistaa inhimilliset virheet.

Pilvipalveluina toimitettujen toiminnanohjausjärjestelmien järjestelmänlaajuinen regressiotestaaminen on luonteeltaan lähimpänä Leungin ja Whiten

(1989) määrittelemää korjaavaa regressiotestaamista, jossa ohjelmiston määritelty toimintalogiikka ei muutu, ja vanhoja testitapauksia voidaan käyttää hyödyksi. Kuitenkin ottaen huomioon kuinka laajasta testauskokonaisuudesta on kyse, toiminnanohjausjärjestelmän tapauksessa voidaan olettaa, että kaikkien organisaatioiden tapauksessa vanhat testitapaukset eivät ole testaamisen kannalta tarpeeksi kattavia. Dittrichin, Vaucoleurin ja Giffin (2009) mukaan organisaatiot eivät aina priorisoi testaamista ja sen laajuus määräytyy myös organisaation halusta panostaa testaamiseen. Tämän takia alkuperäiset testitapaukset eivät välttämättä ole kovinkaan kattavia suhteessa siihen, miten helposti virheitä voi ohjelmistoihin tulla ja kuinka merkittävässä roolissa toiminnanohjausjärjestelmät ovat yrityksen jokapäiväisessä toiminnassa.

Toiminnanohjausjärjestelmien regressiotestaamista tuleekin käsitellä eri tavalla kuin muiden yksinkertaisempien ohjelmistojen. Regressiotestaamisessa pyritään joko valitsemaan tai minimoimaan varsinaisessa testaamisessa käytettyä testitapauslistaa mahdollisimman kattavaksi (Yoo & Harman, 2012). Tämän lisäksi Rothermelin ym. (2001) mukaan testitapauksia on mahdollista priorisoida. Toiminnanohjausjärjestelmien tapauksessa tulisi kuitenkin huomioida, että varsinainen testitapauslista ei alun perinkään välttämättä ole tarpeeksi kattava, jolloin niiden regressiotestaamista ei voida tarkastella samojen kriteerien kautta. Toiminnanohjausjärjestelmien testaamisessa tulisikin ensin kiinnittää huomiota kattavien testitapausten tekemiseen.

5 Pohdinta ja yhteenveto

Tutkielmassa perehdyttiin toiminnanohjausjärjestelmiin, testaukseen ja tämän lisäksi toiminnanohjausjärjestelmien testaamiseen. Tutkielmassa pyrittiin kirjauskatsauksen avulla selvittämään, miten toiminnanohjausjärjestelmien monimutkaisuus sekä niiden siirtyminen pilvipalveluiksi vaikuttavat niiden testaukseen. Kirjallisuuskatsaus toteutettiin perustuen tietokannoista haettuihin lähdemateriaaleihin.

Ensimmäisessä sisältöluvussa määriteltiin käsite toiminnanohjausjärjestelmä ja perehdyttiin sen rooliin yritystä tukevana järjestelmänä. Tämän lisäksi käsiteltiin toiminnanohjausjärjestelmien kehitystä osana liiketoimintaympäristön muutosta ja niiden nykyistä siirtymistä tuotettavaksi pilvipalveluina. Suurimmat toiminnanohjausjärjestelmätoimittajat tarjoavatkin nykyään lähes ainoastaan pilvipalveluina tuotettuja toiminnanohjausjärjestelmiä (Elbahri ym., 2019). Toinen sisältöluku käsitteli testaamista ja siinä käytiin läpi testaamisen eri vaiheet ja se kuinka testaaminen eroaa eri ohjelmistoarkkitehtuurin tasoilla. Luvussa perehdyttiin myös regressiotestaamiseen ja kuinka on mahdollista valita alkuperäisistä testaustapauksista kattava testitapausjoukko regressiotestaamista varten. Kolmas sisältöluku käsitteli toiminnanohjausjärjestelmien testaamista. Luvussa käsiteltiin toiminnanohjausjärjestelmien räätälöintiä sekä verrattiin toiminnanohjausjärjestelmien testaamista ja regressiotestaamista aikaisemmassa luvussa pohdittujen asioiden perusteella.

Tutkielman tulokseksi voidaan lähdekirjallisuuden perusteella todeta, että pilvipohjaisiin toiminnanohjausjärjestelmiin siirtyminen tulee oletettavasti lisäämään toiminnanohjausjärjestelmiä käyttäville organisaatiolle aiheutuneita päivityskustannuksia, jotka johtuvat lisääntyneiden päivitysten aiheuttamasta testaamisesta. Vaikka toiminnanohjausjärjestelmien päivitysten testaamisen kustannuksista ei löytynyt paljoa tutkittua tietoa, voidaan oletuksia tehdä sekä toiminnanohjausjärjestelmäprojekteista saadusta tiedosta että muiden ohjelmistojen ylläpidon kustannuksista. Gerrardin (2007) mukaan yleensä toiminnanohjausjärjestelmäprojekteissa testaamiseen kuluu lähes puolet käytettävissä olevasta budjetista, johtuen järjestelmien kriittisyydestä yrityksille. Vaikka toiminnanohjausjärjestelmiä ei ole ennen yleisesti regressiotestattu, voidaan sen kus-

tannusten olettaa olevan samanlaiset kuin muillakin ohjelmistoilla. Tämä ohjelmiston uudelleentestaaminen voi muiden ohjelmistojen tapauksessa Rothermelin ym., (2001) mukaan aiheuttaa jopa puolet julkaistun ohjelmiston ylläpitokustannuksista. Tämän lisäksi toiminnanohjausjärjestelmiä joudutaan suorittamaan monia koko järjestelmää koskevia testejä, jolloin testioraakkelina toimii yleensä ihminen. Ihmisen käyttäminen nostaa Baresin ja Youngin (2001) mukaan testaamisen kustannuksia merkittävästi. Näiden perusteella voitaneen olettaa toiminnanohjausjärjestelmien regressiotestaamisen aiheuttavan huomattavia kustannuksia niitä käyttäville organisaatioille.

Tutkielmaa tarkasteltaessa tulisi kuitenkin huomioida, että se on suppea kirjallisuuskatsaus ja sen lisäksi toiminnanohjausjärjestelmien siirtyminen pilvipalveluiksi on kohtalaisen uusi ilmiö. Johtuen toiminnanohjausjärjestelmien vaatiman testauksen vähäisyydestä jouduttiin sitä käsittelevään osioon valitsemaan aineisto verrattain kapeasta lähdemateriaalista. Tämän lisäksi pilvipalvelukehitys on viime vuosina ollut hyvin nopeaa, joten sen olemuksen voidaan olettaa jo muuttuneen. Toisaalta toiminnanohjausjärjestelmät ja testaus käsitteinä ovat hyvin laajasti tutkittuja aihealueita.

Jatkotutkimusaiheina toiminnanohjausjärjestelmien testaamiseen voidaan nähdä useita, sillä mahdollisella testauksen tehostamisella voidaan saada merkittäviä taloudellisia hyötyjä. Toiminnanohjausjärjestelmien regressiotestaaminen tulee yleistymään, mutta sitä ei vielä ole tutkittu kovinkaan paljon. Tutkimusaiheina voidaan nähdä esimerkiksi keskittyminen toiminnanohjausjärjestelmien oraakkeliongelman erityispiirteisiin tai siihen kuinka testeihin olisi mahdollista luoda hyvät syöttöarvot. Regressiotestausta tutkittaessa voitaisiin tutkia sitä kuinka yritykset tällä hetkellä testaavat omia järjestelmiään ja sitä kuinka kattavat testitapauslistat yrityksillä tällä hetkellä on.

LÄHTEET

Al-Hossan, A., & Al-Mudimigh, A. S. (2011). Practical guidelines for successful ERP testing.

Al-Shardan, M. M., & Ziani, D. (2015). Configuration as a service in multi-tenant enterprise resource planning system. *Lecture Notes on Software Engineering*, *3*(2), 95.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Baresi, L., & Young, M. (2001). *Test oracles*. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, USA.

Bertolino, A. (2007, May). Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering* (pp. 85-103). IEEE Computer Society.

Bjelland, E., & Haddara, M. (2018). Evolution of ERP systems in the cloud: A study on system updates. *Systems*, *6*(2), 22.

Dittrich, Y., Vaucouleur, S., & Giff, S. (2009). ERP customization as software engineering: knowledge sharing and cooperation. *IEEE software*, *26*(6), 41-47.

Gerrard, P. (2007, September). Test methods and tools for ERP implementations. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)* (pp. 40-46). IEEE.

Goodenough, J. B., & Gerhart, S. L. (1975). Toward a theory of test data selection. *IEEE Transactions on software Engineering*, (2), 156-173.

Elbahri, F. M., Al-Sanjary, O. I., Ali, M. A., Naif, Z. A., Ibrahim, O. A., & Mohammed, M. N. (2019, March). Difference Comparison of SAP, Oracle, and Microsoft Solutions Based on Cloud ERP Systems: A Review. In *2019 IEEE 15th International Colloquium on Signal Processing & Its Applications (CSPA)* (pp. 65-70). IEEE.

Hunton, J. E., Lippincott, B., & Reck, J. L. (2003). Enterprise resource planning systems: comparing firm performance of adopters and nonadopters. *International Journal of Accounting information systems*, *4*(3), 165-184.

Johansson, B., & Ruivo, P. (2013). Exploring factors for adopting ERP as SaaS. *Procedia Technology*, *9*, 94-99.

Klaus, H., Rosemann, M., & Gable, G. G. (2000). What is ERP?. *Information systems frontiers*, *2*(2), 141-162.

Leung, H. K., & White, L. (1989, October). Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989* (pp. 60-69). IEEE.

Microsoft. Haettu 7.10.2019 osoitteesta https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/fin-ops/get-started/public-preview-releases

Parthasarathy, S., & Sharma, S. (2016). Efficiency analysis of ERP packages—A customization perspective. *Computers in Industry*, *82*, 19-27.

Peng, G. C. A., & Gala, C. (2014). Cloud ERP: a new dilemma to modern organisations?. *Journal of Computer Information Systems*, *54*(4), 22-30.

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, *27*(10), 929-948.

Ruivo, P., Oliveira, T., & Neto, M. (2012). ERP use and value: Portuguese and Spanish SMEs. *Industrial Management & Data Systems, 112*(7), 1008-1025.

Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012). Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA)*, *2*(3), 980-986.

Schliesser, S. (2007, October). An approach to ERP testing using services. In *IEEE International Conference on Software-Science, Technology & Engineering (SwSTE'07)* (pp. 14-21). IEEE.

Seethamraju, R. (2015). Adoption of software as a service (SaaS) enterprise resource planning (ERP) systems in small and medium sized enterprises (SMEs). *Information systems frontiers*, *17*(3), 475-492.

Shahamiri, S. R., Kadir, W. M. N. W., & Mohd-Hashim, S. Z. (2009, September). A comparative study on automated software test oracle methods. In *2009 Fourth International Conference on Software Engineering Advances* (pp. 140-145). IEEE.

Sun, W., Zhang, X., Guo, C. J., Sun, P., & Su, H. (2008, September). Software as a service: Configuration and customization perspectives. In *2008 ieee congress on services part ii (services-2 2008)* (pp. 18-25). IEEE.

Templier, M., & Paré, G. (2015). A framework for guiding and evaluating literature reviews. *Communications of the Association for Information Systems*, *37*(1), 6.

Umble, E. J., Haft, R. R., & Umble, M. M. (2003). Enterprise resource planning: Implementation procedures and critical success factors. *European journal of operational research*, *146*(2), 241-257.

Vänskä, O. Haettu 7.10.2019 osoitteesta https://www.tivi.fi/uutiset/laaketukkuri-oriola-kiistaa-ohjelmisto-ongelmat-sap-siirtymaa-testattiin-huolella-silti-meni-pieleen/101b2168-4523-3ab3-b3ac-ea7b54980104

Whittaker, J. A. (2000). What is software testing? And why is it so hard?. *IEEE software*, *17*(1), 70-79.

Wieczorek, S., Kozyura, V., Schur, M., & Roth, A. (2012, March). Practical model-based testing of user scenarios. In *2012 IEEE International Conference on Industrial Technology* (pp. 306-311). IEEE.

Wieczorek, S., Stefanescu, A., & Schieferdecker, I. (2008, April). Test data provision for ERP systems. In *2008 1st International Conference on Software Testing, Verification, and Validation* (pp. 396-403). IEEE.

Wieczorek, S., & Stefanescu, A. (2010, March). Improving testing of enterprise systems by model-based testing on graphical user interfaces. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems* (pp. 352-357). IEEE.

Wong, W. E., Horgan, J. R., London, S., & Agrawal, H. (1997, November). A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering* (pp. 264-274). IEEE.

Xin, M., & Levina, N. (2008, December). Software-as-a-service model: Elaborating client-side adoption factors. In *Proceedings of the 29th International Conference on Information Systems, R. Boland, M. Limayem, B. Pentland,(eds), Paris, France*.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, *22*(2), 67-120.