Timo Pyykkö

OHJELMISTOTESTAUS SIIRRYTTÄESSÄ PERINTEISISTÄ OHJELMISTOKEHITYSMENETELMISTÄ SCRUMIIN

Tietojärjestelmätieteen
pro gradu -tutkielma
18.1.2010

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Pyykkö, Timo Antero
Tietojärjestelmätieteen pro gradu -tutkielma / Timo Pyykkö
Jyväskylä: Jyväskylän yliopisto, 2010.
166 s.

Pro gradu -tutkielma

Ohjelmistotestauksesta muodostuu usein yksi vaikeimmista hallittavista asioista ketteriin menetelmien käyttöön siirryttäessä. Myös itse ketterän menetelmän käyttöönotto on haastava ja pitkä prosessi.

Tutkimuksen tavoitteena on selvittää mitä haasteita ja muutostoimenpiteitä ohjelmistotestaukseen on odotettavissa, kun projekteissa tapahtuvaa ohjelmistokehitystä aletaan tehdä perinteisten ohjelmistokehitysmenetelmien sijaan Scrumia käyttäen. Tutkielmassa käsitellään myös ohjelmisto-organisaatioissa tapahtuvaa ketterien menetelmien pilotointia ja käyttöönottoa, joilla on olennainen vaikutus testauksen muutosprosessiin.

Vastauksia tutkimusongelmaan etsitään kirjallisuuskatsauksen pohjalta ja tapaustutkimuksen avulla luomalla katsaus toimeksiantajaorganisaation kahdessa projektissa toteutettuun Scrum-pilotointiin. Lähdekirjallisuudesta kerätään tietoa, vaihtoehtoja ja muiden esittämiä yleisiä kokemuksia ketterien menetelmien pilotoinnista ja testauksesta. Lähdekirjallisuudesta tehtyjä havaintoja verrataan toimeksiantajaorganisaation kahden projektin tuloksiin, jotka on puolestaan johdettu Scrumia pilotoineista projekteista havaintojen ja kyselyjen avulla.

Pilottiprojekteista saadut tutkimustulokset ovat samansuuntaisia kirjallisuuden kanssa, joka kertoo ketterän menetelmän käyttöönottoon ja Scrumissa suoritettavaan testaukseen liittyvistä haasteista. Ketterien menetelmien onnistunut ja tehokas käyttöönotto kestää kauan ja edellyttää huolellista valmistelua ja suunnittelua. Käyttöönottoa edeltävään pilotoinnin onnistumiseen vaikuttaa niin projektin luonne, käytettävä teknologia, tiimin koostumus kuin asiakaskin. Testaus ja laadunvarmistus Scrumissa kuuluu koko tiimille. Tutkiva testaus ja testauksen automatisointi ovat tärkeimpiä testauksen peruspilareita Scrumissa.

AVAINSANAT: ketterät menetelmät, ohjelmistokehitys, pilotointi, Scrum, testaus

ABSTRACT

Pyykkö, Timo Antero
Master's Thesis in Information Systems Science / Timo Pyykkö
Jyväskylä: University of Jyväskylä, 2010.
166 p.

Master's Thesis

When adopting agile methods, software testing is often one of the most difficult issues to control. Moreover, adoption of the agile method itself is a challenging and long-term process, in which a number of issues have to be taken into account.

The aim of this study is to find out the challenges and changes needed in software testing when software is developed with Scrum instead of traditional software development methods. In addition, the piloting and adoption of agile methods in software organizations are examined, both of which have a substantial effect on change process in testing.

In order to answer the research question, a literature review and a case study which examined Scrum pilot projects of the commission organization were performed. Literature is used in order to gather information, alternatives and common experiences of Scrum piloting and testing. Findings found are compared to results of two projects of commission organization. The results of pilot projects have been gathered from Scrum pilot projects via observing and inquiries.

The results obtained from pilot projects are consistent with the literature related to adoption agile methods and Scrum testing challenges. Successful and efficient adoption of an agile method takes a long time and demands careful preparation and planning. The success rate of piloting performed before full adoption depends on the characteristics of the project, technology in use, team structure and customer. In Scrum the responsibility of testing and quality assurance belongs to everybody. Exploratory testing and test automation are the key pillars of testing in Scrum.

KEYWORDS: agile methods, piloting, Scrum, software development, testing

SISÄLTÖ

1 JOHDANTO

Ohjelmistoliiketoiminnassa mukana olevilta organisaatioilta vaaditaan koko ajan enemmän yhä lyhyemmässä ajassa. Kehitettävien ohjelmistojen sisältämät toiminnallisuudet ja vaatimukset muuttuvat projektin aikana yhä useammin ja nopeammin. Lisäksi ohjelmiston koko ja kompleksisuus kasvaa. Samaan aikaan järjestelmän tilaajat vaativat, että ostettujen järjestelmien osakokonaisuudet ja muut toimitukset tulee olla täysin testattu. Järjestelmien tulee myös vastata toiminnallisuuksiltaan loppukäyttäjien todellisia tarpeita ja annettuja vaatimuksia.

Kyky joustaa, toimia dynaamisesti ja pystyä toimittamaan toimivia ohjelmistoja tai niiden osakokonaisuuksia nopeasti pakottaa ohjelmistoalan organisaatiot muuttamaan toimintatapojaan, jotta ne kykenisivät vastaamaan asiakkaan vaatimuksiin ja tarpeisiin. Yhä useammat organisaatiot siirtyvät käyttämään ketteriä menetelmiä tai ainakin kokeilevat niiden soveltuvuutta omassa toimintaympäristössään. Lovelandin, Millerin, Prewittin ja Shannonin (2005, 339) mukaan ketterien ohjelmistokehitysmenetelmien tavoite on toimittaa toimivaa ohjelmistoa aikaisin ja usein. Menetelmien vahvuutena on kyky käsitellä muuttuvat vaatimukset läpi kehitys- ja testausvaiheiden (Loveland ym. 2005, 339). Ketterissä menetelmissä käytössä olevat testausmenetelmät ja -käytännöt eivät kuitenkaan ole tarpeeksi kattavia, vaan niiden kehittämiselle ja tutkimiselle on tarvetta.

Testaus on tärkeä osa ohjelmistokehitystä. Perinteisten ohjelmistokehitysmenetelmien testauskäytännöt ja -lähestymistavat sisältävät lukuisia aktiviteetteja, jotka sijoittuvat integraatio- ja testausvaiheeseen ohjelmistokehitysprojektin lopussa. Tällöin testausvaiheen kesto ei useinkaan ole ennustettavissa ja omat haasteensa testaukseen tuovat myös alati muuttuvat vaatimukset. Tiukasti aikaan sidotut ketterät menetelmät edellyttävät testaukseen ja laadunvarmistukseen erilaista lähestymistapaa. Tästä huolimatta useimmat ketterät menetelmät

eivät juurikaan ota kantaa siihen, kuinka testausta tulisi tarkalleen ottaen tehdä. Useimmat ketterät menetelmät sisältävät suuren määrän hyviä käytäntöjä kehittäjille, sisältäen muun muassa automatisoidun yksikkötestauksen. Kuitenkin, ohjeistuksen kattavan testauksen suorittamiseen voidaan sanoa olevan puutteellista. Tämä johtaa siihen, että organisaation on itse päätettävä tai kehitettävä sovellettavat testauskäytännöt esimerkiksi perinteisten ohjelmistokehitysmenetelmien testauskäytäntöjen pohjalta. Testauskäytäntöjen yhdistäminen ketterien menetelmien kanssa voi kuitenkin olla haasteellista. (Itkonen, Rautiainen & Lassenius 2005).

Organisaation tai kehitystiimin miettiessä ketterien menetelmien käyttöönottoa on hyvä pysähtyä miettimään, kuinka esimerkiksi teknologia tukee tai hidastaa uusia työtapoja ja etenkin testausta. Crispin ja Gregory (2009, 35) esittävät, että kun ohjelmistokehitysorganisaatiot ottavat käyttöön ketterän menetelmän, kestää muutosprosessi yleensä pisimpään testaus- tai laadunvarmistustiimeillä. Perinteiset testaustoimenpiteet, kuten metriikkojen ylläpito ja seuranta sekä testisuunnitelmien kirjoitus eivät näytä soveltuvan ketteriin projekteihin kovinkaan hyvin (Crispin & Gregory 2009, 35).

Tutkielmassa seurattiin toimeksiantajaorganisaation kahden eri projektin Scrum-pilotointia testausnäkökulmaa painottaen. Tutkijan motiivina oli saada tiiminsä testaus pitkällä tähtäimellä mahdollisimman tehokkaaksi ja toimivaksi. Tämän mahdollistamiseksi oli selvitettävä, mitä mahdollisuuksia ja vaihtoehtoja testauksen suorittamiseen on projekteissa, joissa käytössä olevana ohjelmistokehitysmenetelmänä on Scrum. Tutkielmassa käsitellään kirjallisuudesta löytyviä Scrum-testauksen toteutuksesta kertovia tapoja, joita sekä sovelletaan että testataan toimeksiantajaorganisaation pilottiprojekteissa. Tutkimuksen tavoitteena oli selvittää mitä haasteita ja muutostoimenpiteitä ohjelmistotestaukseen oli odotettavissa, kun projekteissa tapahtuvaa ohjelmistokehitystä alettiin tehdä perinteisten ohjelmistokehitysmenetelmien sijaan Scrumia käyttäen. Oman haasteensa Scrum-pilotoinnille asetti projekteissa käytetty teknologia, IBM:n

Lotus Notes ja Domino. Tutkielman kirjoittaja osallistui aktiivisesti molempiin projekteihin testaajan roolissa suorittaen niissä samanaikaisesti osallistuvaa havainnointia. Täten ketterien menetelmien testausta käsitellään tutkielmassa ensisijaisesti testaajan näkökulmasta, minkä vuoksi esimerkiksi ketteriin menetelmiin kuuluva kehittäjien suorittamaa yksikkötestaus on jätetty työssä vähemmälle huomiolle.

Alkuhaastatteluista saatujen tulosten perusteella valittiin testauksen kehittämiskohteet, jotka nekin johtivat osaltaan päätökseen lähteä toteuttamaan Scrum-pilotteja. Alkuhaastatteluissa ilmenneitä testauksen epäkohtia, pilottien aikana suoritettua havainnointia ja Scrum-pilottien päätteeksi tehtyjen kyselyjen tuloksia vertaillaan keskenään kohdassa 7.4.

Tutkielman alussa luvussa kaksi esitellään perinteiset ohjelmistokehitysmenetelmät yleisellä tasolla ja määritellään niissä käytettävän ohjelmistotestauksen tunnuspiirteet. Luvussa kolme käydään läpi ketterien menetelmien määritelmä, tarkoitus, soveltuvuus ja testaus. Lisäksi luvussa luodaan käytettävästä ohjelmistokehitysmenetelmästä riippumaton katsaus testaajaan rooliin.

Koska Scrum on toimeksiantajaorganisaation piloteissa kokeiltava menetelmä, käsitellään se erikseen luvussa neljä. Luvussa viisi paneudutaan ketterien menetelmin pilotointiin ohjelmisto-organisaatiossa: miten ketterä menetelmä otetaan hallitusti käyttöön, mitä sudenkuoppia sen käyttö pitää sisällään ja miten ne voidaan välttää sekä miten valitaan järkevä pilottiprojekti ja sen ajankohta. Lisäksi luvussa esitellään lähdekirjallisuudesta löydettyjä ketterän menetelmän käyttöönoton kokemuksia.

Luvussa kuusi esitellään tutkielman empiirisen osuuden kohde, tutkimuksen menetelmät ja sen toteuttaminen. Tutkimustulokset, jotka on johdettu kirjallisuuskatsauksen lisäksi alkuhaastatteluista, havainnoinnista ja loppukyselyistä, esitellään luvussa seitsemän. Luku sisältää myös vastauksen tutkimusongel-

maan. Pohdinta ja johtopäätökset ovat luvussa kahdeksan ja tutkielma päättyy yhteenvetoon luvussa yhdeksän.

Tutkielman päälähteenä on käytetty Lisa Crispinin ja Janet Gregoryn kirjaa Agile Testing - A Practical Guide for Testers and Agile Teams. Kirjaa voidaan pitää tutkielman kirjoittamisajankohtana yhtenä tärkeimmistä ketterää testausta käsittelevän kirjallisuuden teoksista. Toisena tutkielman tärkeänä lähteenä on käytetty Allan Kellyn kirjaa Changing Software Development – Learning to become agile. Kirja käsittelee ketterien menetelmien käyttöönottoa organisaatioissa lähestyen aihepiiriä tyypillisimpien käyttöönotossa havaittujen ongelmien näkökulmasta.

Tutkielman aihe sisältää lukuisia ohjelmistoalaan vakiintuneita, lainasanoista johdettuja termejä. Termien suomentamiseen on käytetty apuna kirjallisuudesta löytyneitä esimerkkejä niiltä osin, kuin lähteiden suomennokset on koettu järkeviksi ja yleisesti hyväksytyiksi sanojen vastineiksi. Muissa tapauksissa tutkija on suomentanut termit itse. Osaa termeistä, kuten esimerkiksi inkrementtiä, ei ole katsottu mielekkääksi lähteä suomentamaan, sillä lukijalla oletetaan olevan perustietämys aihepiiriin ja siinä käytettyyn sanastoon. Merkittävimmistä käsitteistä ohjelmistovirheellä tarkoitetaan *bugia*, pyrähdyksellä *sprinttiä* ja kehitettävällä toiminnallisuudella puolestaan kehittäjän pienintä mahdollista toteutettavaa osakokonaisuutta, *issueta* tai *taskia*.

2 PERINTEISET OHJELMISTOKEHITYSMENETELMÄT JA TESTAUS

Tässä luvussa esitellään lyhyesti muutamia kirjallisuudessa perinteisiksi luokiteltuja ohjelmistokehitysmenetelmiä. Menetelmistä käsitellään tässä yhteydessä vain niitä, joiden tunnuspiirteistä yhtä tai useampaa on käytetty tiimissä, johon tutkimuksen pilottiprojektit kuuluvat. Lisäksi luvussa määritellään ohjelmistotestauksen käsite sekä sen integroituminen ohjelmistokehitykseen. Luvussa pyritään vastaamaan seuraaviin kysymyksiin: mitä testauksella tarkoitetaan, miksi sitä tarvitaan ja miten testausta suoritetaan perinteisissä ohjelmistokehitysmenetelmissä.

2.1 Perinteiset ohjelmistokehitysmenetelmät

Ennen ketteriä prosessimalleja kehitettyjä ohjelmistokehitysmenetelmiä kutsutaan suunnitelmakeskeisiksi (*plan-driven*) ohjelmistokehitysmenetelmiksi. Ohjelmistokehitysmenetelmän suunnitelmakeskeisyydellä tarkoitetaan menetelmän sisältävän paljon suunnittelutyötä ennen varsinaisen toteutuksen alkamista. Ohjelmiston kehitys nähdään elinkaarena, joka alkaa tarpeiden määrittelystä ja päättyy ylläpitoon. Suunnitelmakeskeisyys ei kuitenkaan välttämättä sulje pois menetelmän tai viitekehyksen iteratiivisuutta: esimerkiksi spiraalimalli on sekä iteratiivinen että suunnittelukeskeinen.

Ohjelmiston elinkaarella tarkoitetaan aikaa, joka kuluu ohjelmiston kehittämisen aloittamisesta sen poistamiseen käytöstä. Vaihejakomallilla puolestaan tarkoitetaan tapaa, jolla ohjelmiston kehitystyö tai koko elinkaari jaetaan vaiheisiin. (Haikala & Märijärvi 2006, 36.) Kaikkia edellisessä kappaleessa määritettyyn joukkoon kuuluvia kehitysmenetelmiä kutsutaan tässä tutkielmassa perinteisiksi ohjelmistokehitysmenetelmiksi. Niillä tarkoitetaan siis yleisesti ottaen vesiputousmallin, V-mallin, Evo-mallin, spiraalimallin ja/tai protoilumallin tunnuspiirteitä sisältäviä menetelmiä, jotka eivät ole ketteriä ja joissa suunnitte-

lua ja dokumentaatiota pidetään äärimmäisen tärkeinä asioina. Seuraavaksi esitellään näitä suunnitelmakeskeisiä ohjelmistokehitysmenetelmiä ja elinkaarimalleja, joita on käytetty myös toimeksiantajaorganisaation tiimissä kokonaisuudessaan tai osittain ennen Scrum-pilotointia.

2.1.1 Vesiputousmalli

Yhdysvaltalainen IT-tiedemies Winston Royce esitteli vuonna 1970 artikkelissaan sittemmin maailman kuuluisimmaksi muodostuneen ohjelmistokehityksessä käytettävän prosessin, vesiputousmallin. Roycen mukaan ohjelmistotuotantoa ei kuitenkaan tulisi tehdä mallin mukaisesti ja hän onkin kritisoinut esittelemäänsä vesiputousmallia sekä kyvyttömyydestä käsitellä muutoksia että joustamattomuudesta iteratiivisuuden suhteen. Kuitenkin hyvin pian Roycen artikkeliin viitattiin virheellisesti vesiputousmallin alkuperäisteoksena, vaikka Royce kannatti todellisuudessa iteratiivista kehitystyötä. (Larman & Basili 2003.)

Vesiputousmalli on erikoistapaus yleisestä ongelmanratkaisumallista: se kehottaa analysoimaan ratkaistavan ongelman, suunnittelemaan ratkaisun, toteuttamaan sen sekä testaamaan ratkaisua (Haikala & Märijärvi 2006, 41). Vesiputousmallissa ohjelmistokehitys on jaettu selkeisiin osiin, jotka ovat vaatimusanalyysi, määrittelyvaihe, suunnitteluvaihe, toteutusvaihe ja testausvaihe, jonka jälkeen tuote on valmis siirrettäväksi tuotantoon. Tuotteen elinkaari jatkuu vielä tämänkin jälkeen mahdollisella ylläpitovaiheella. (Royce 1970.)

Vesiputousmallin vahvuutena on sen perusteellisuus, joka auttaa tuottamaan erittäin vakaata koodia. Mallin selvimpänä heikkoutena pidetään ohjelmiston tuotantoon saamisen vaikeutta, jonka nähdään johtuvan vaiheiden keskinäisestä riippuvaisuudesta: Kaikki vaiheet eivät todennäköisesti tule ajoittumaan juuri niin kuin on suunniteltu. Tämä aiheuttaa usein ongelmia tilanteissa, joissa vaiheen pitäisi päättyä ja testaus on silti kesken. (Loveland ym. 2005, 44.) Vesiputousmalli perustuu vahvaan dokumentaatioon, joten se soveltuu projektei-

hin, joiden vaatimukset ovat hyvin selvillä ennen projektin alkua ja joihin ei tule juurikaan muutoksia (Haikala & Märijärvi 1997, 27–28).

Loveland ym. (2005, 44) esittävät vesiputousmallin vaativan meneillään olevan vaiheen valmistumista, ennen kuin seuraava vaihe voi alkaa. Malli ei kuitenkaan ole yksisuuntainen: on tyypillistä, että menossa olevasta vaiheesta joudutaan palaamaan edeltäneeseen vaiheeseen yhden tai useamman kerran johtuen erilaisista muutoksista tai vastaan tulleista haasteista (Haikala & Märijärvi 1997, 27–28).

Yhteenvetona voidaan siis todeta, että vesiputousmallissa edetään käytetyn ajan suhteen vaiheesta toiseen lineaarisessa järjestyksessä. Keskeinen idea on, että tietyllä ajanhetkellä keskitytään ainoastaan yhteen prosessivaiheeseen kerrallaan. Kun työ on valmis, siirrytään seuraavaan vaiheeseen tai palataan edelliseen, mikäli aiempaa työtä on tarpeen muuttaa – olettaen kuitenkin suunnan olevan aina eteenpäin.

2.1.2 V-malli

V-mallissa ohjelmiston toteuttaminen lähtee liikkeelle suunnittelusta. Ensin tehdään vaatimusmäärittely. Tämän jälkeen siirrytään suunnitteluvaiheeseen, jonka aikana tuote suunnitellaan aloittaen laajemmasta kokonaisuudesta siirtyen vaiheittain tarkempiin, yksittäisiin toimintoihin. Suunnittelun valmistuttua tehdään varsinainen tuotteen toteutus. V-mallissa korostetaan verifioinnin ja validoinnin merkitystä omina työvaiheinaan, ja niitä molempia on tarkoitus tehdä koko tuotteen kehittämisen ajan. (Boehm 1979.)

Toteutusta seuraa testaus, joka sisältää tietyn suoritusjärjestyksen: ensin testataan tuotteen pienet ja yksittäiset toiminnot ja niiden myötä testauskohdetta laajennetaan suurempiin kokonaisuuksiin, päätyen lopuksi koko tuotteen järjestelmätestaukseen ja asiakkaan hyväksyntätestaukseen. V-mallissa testaus ei ole enää erillinen työvaihe, vaan se on integroitu kaikkiin prosessin osavaiheisiin.

(Gilb 1993, 38.) Testauksessa on neljä eri vaihetta, joita ovat yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus (Gilb 1993, 38; ks. myös Boehm 1979; Haikala & Märijärvi 2006, 288–289).

Gilb (1993, 38) on tutkinut eri ohjelmistokehitysmenetelmien vahvuuksia ja pitää V-mallia vesiputousmallia tehokkaampana. Perusteluna tähän Gilb (1993, 38–39) pitää V-mallin kykyä mahdollistaa muun muassa testauksen suunnittelun ja testitapausten kirjoittamisen aloittamisen heti projektin alussa, jopa ennen toteutuksen alkamista.

2.1.3 Protoilumalli

Haikalan ja Märijärven (2006, 42) mukaan prototyyppilähestymistavalla voidaan tarkoittaa lähes mitä tahansa työskentelymallia, jossa jotain tuotteen piirrettä kokeillaan ennen varsinaisen tuotteen rakentamista. Haikala ja Märijärvi (2006, 42) jatkavat toteamalla prototyyppien soveltuvan erityisesti uuden teknisen ratkaisun vaatiman kokeilun tekemiseen tai epäselvien asiakasvaatimuksien etsimiseen.

Protoilumallin taustalla on olettamus, jonka mukaan iso osa projektiin liittyvistä avoimista ongelmista selviää vasta tuotteen ensimmäisen valmistumisen ja sen asiakkaalle esittelemisen myötä. Täten ensimmäinen malli tai luonnos voi olla erittäin pelkistetty, jotta malli saataisiin nopeasti asiakkaan kokeiltavaksi. Kun asiakkaalla on käytettävissään konkreettinen tuote, jota hän voi kokeilla ja kommentoida, on vaatimusten esittäminen helpompaa, olipa protomalli miten alustava ja keskeneräinen tahansa. (Haikala & Märijärvi 2006, 42–44).

Ongelmana protoilussa Haikala ja Märijärvi (2006, 43) pitävät tilanteita, joissa asiakas luulee lähes oikealta näyttävän tuotteen tai järjestelmän olevan jo käytännöllisesti katsoen valmis, vaikka käytännössä valtaosa toteutuksesta on vielä tekemättä. Täten prototyypistä ei kannata välttämättä tehdä mahdollisimman

viimeistellyn näköistä ja tuntuista – asiakkaankin on huomattava, että järjestelmä on vielä keskeneräinen (Haikala & Märijärvi 2006, 43).

2.1.4 Evo-malli

Evo-mallin ideana on rakentaa ensimmäisessä projektissa ydinjärjestelmä, jota seuraavissa projekteissa tai kehitysjaksoissa kehitetään eteenpäin. Jokaisen jatkossa tehtävän kehitysvaiheen tuloksena järjestelmästä syntyy uusi versio, jota kehitetään edelleen seuraavilla kierroksilla. Evo-mallin tapaisia kehitysmalleja sanotaan joskus myös inkrementaalisiksi malleiksi. Tällä tarkoitetaan yleensä ohjelmistokehitystä, jossa lopputuotetta kehitetään pienehköinä inkrementteinä yhden projektin sisällä. Projektissa määritetty lopputulos saadaan aikaan esimerkiksi kolmella inkrementtikierroksella, joista jokaisen tuloksena on toimiva järjestelmä. (Haikala & Märijärvi 2006, 45.)

Haikalan ja Märijärven (2006, 47) mukaan kaikkeen Evo-mallin tapaiseen ohjelmistokehitykseen liittyy yksi vakava ja yleinen hallinnallinen ongelma: kun asiakas on saanut kehitettävästä ohjelmistosta uuden version, voi käydä niin, että projektiryhmän aika kuluu asiakkaan ongelmien ratkomiseen ja virheiden korjailuun, jolloin seuraavan inkrementin kehitys pysähtyy kokonaan tai ainakin hidastuu. Toisena sudenkuoppana Haikala ja Märijärvi (2006, 47) pitävät liiallisen inkrementaalisuuden käyttöä, joka voi johtaa ohjelman pirstoutumiseen ja kokonaisarkkitehtuurin rapautumiseen.

2.1.5 Spiraalimalli

Amerikkalainen ohjelmistoinsinööri ja professori Barry Boehm esitteli vuonna 1988 spiraalimallin. Spiraalimalli muistuttaa Evo-mallia, eli sen vaiheet etenevät kierroksittain spiraalimaisesti samoja vaiheita toistaen. Mallista on havaittavissa myös niin protoilun kuin vesiputousmallinkin piirteitä. Jokaisen kierroksen tuloksena ei kuitenkaan ole välttämättä toimiva tuote: tulokset voivat olla esi-

merkiksi määrittelyvaiheen dokumentaatiota, jotka toimivat syötteinä spiraalin seuraavalle kierrokselle. (Boehm 1988.)

Spiraalimalli korostaa erityisesti riskien hallintaa ja riskit käydäänkin läpi spiraalin jokaisella kierroksella. Spiraalimallin käytön tarjoama suurin etu on sen kyky pystyä sisällyttämään olemassa olevien ohjelmistokehitysmallien hyviä puolia minimoimalla mallien huonojen puolien mahdollisia vahinkotekijöitä. (Boehm 1988.)

2.2 Ohjelmistotestauksen määritelmä, tarkoitus ja tavoitteet

Ohjelmistotestaus on merkittävä osa ohjelmiston elinkaarta. Testauksen määritelmiä ja tarkoituksia tavoitteineen on vuosien saatossa esitetty lukuisia Niin kuin aina, osa määritelmistä on myös hieman toisistaan poikkeavia, ja seuraavaksi esitelläänkin ohjelmistotestauksesta esitettyjä erilaisia määritelmiä ja näkemyksiä.

Dijkstra (1972) on todennut, että testaus kykenee näyttämään ohjelmistovirheiden olemassaolon, mutta testauksella ei koskaan pystytä osoittamaan niiden olemattomuutta. Adrionin, Branstadin ja Cherniavskyn (1982) mukaan testaus on ”ohjelman käyttäytymisen tutkimista suorittamalla ohjelmaa testiaineistoilla”. Hennellin, Hedleyn ja Riddellin (1984) mielestä testauksen tarkoituksena ei ole löytää virheitä, vaan vakuuttua siitä, että niitä ei ole. Hetzelin (1988, 26) päätelmä testauksesta on, että se on mitä tahansa tekemistä, jossa arvioidaan ohjelman tai järjestelmän ominaisuuksia ja kyvykkyyttä, ja päätellään tulosten yhtenevyys vaatimuksiin. Myers (2004) on puolestaan esittänyt testauksen olevan ohjelman suorittamista siten, että aikomuksena on löytää virheitä.

Craig ja Jaskiel (2002, 4) toteavat Myersin ja Hetzelin määritelmien olevan puutteellisia testauksen ulottuvuuden ja rajojen osalta. Täten Craig ja Jaskiel (2002, 4) päätyvätkin määrittelemään testauksen olevan ohjelmistokehityksen rinnak-

kainen elinkaariprosessi, joka käyttää ja ylläpitää testiaineistoa ja sen työkaluja, jolloin testattavana olevan ohjelmiston laatua voidaan mitata ja parantaa. Haikala ja Märijärvi (2004, 282) määrittelevät ohjelmistotestauksen olevan suunnitelmallista virheiden etsimistä ohjelmaa tai sen osaa suorittamalla. Ohjelmistotestaus voi olla myös suunnittelematonta, sillä esimerkiksi tutkiva testaus (*exploratory testing*) ei välttämättä noudata mitään ennalta määrättyä kaavaa tai suunnitelmaa.

Black (2007, 6) puolestaan lähtee ohjelmistotestauksen määrittelyssä liikkeelle siitä, mitä ohjelmistotestaus ei ole, päätyen seuraavanlaiseen tulokseen: Ohjelmistotestaus ei sen ole todistamista, että ohjelmistossa ei enää esiinny virheitä, eikä se myöskään ole sitä, että löydetään kaikki virheet. Testaustiimin on mahdotonta saavuttaa sellaista tehtävää (Black 2007, 6). Samaan tulokseen mahdottomuudesta todistaa ohjelmiston virheettömyys testauksen avulla on tullut myös Beizer (1990, 24).

Lovelandin ym. (2005, 6) mukaan ohjelmistotestaajan perimmäisenä tarkoituksena on varmistaa, että löydettyjen ohjelmistovirheiden joukossa ovat juuri ne pahimmat virheet, joilla on merkitystä. Testauksen tarve voidaan ymmärtää myös Pezzén ja Youngin (2008, 29) toteamasta ihmisten inhimillisyydestä: Ohjelmistokehittäjinä toimivat ihmiset tekevät erehdyksiä (*errors*), jotka johtavat virheisiin (*faults*) ohjelmistossa. Virheet saattavat johtaa häiriöihin (*failure*), mutta virheitä sisältävä ohjelmisto ei välttämättä aiheuta häiriöitä jokaisella sen käyttökerralla. (Pezzé & Young 2008, 29.)

Testauksen ja sen suunnittelun – osina laadunvarmistusta – tulisi keskittyä ohjelmistovirheiden ennaltaehkäisyyn. Siinä määrin kuin testaus ja testauksen suunnittelu eivät estä itse ohjelmistovirheitä, niiden pitäisi kyetä löytämään ohjelmistovirheiden aiheuttamat oireet. Toteutumaton ohjelmistovirhe on parempi kuin syntynyt, havaittu ja korjattu ohjelmistovirhe, sillä mikäli ohjelmistovirhe onnistutaan ennaltaehkäisemään, korjausta tarvitsevaa ohjelmistokoodia ei ole. Tällöin myöskään korjauksen toimivuutta ei tarvitse vahvistaa uudel-

leentestauksella, minkä lisäksi vältytään myös aikataulumuutoksilta. (Beizer 1990, 3.)

Testien tehokas suunnittelu on yksi parhaista ja tunnetuista ohjelmistovirheiden ehkäisymenetelmistä: ajatustyö, joka kehittäjän täytyy tehdä luodakseen hyödyllisen testitapauksen voi auttaa löytämään ja eliminoimaan ohjelmistovirheitä ohjelmistokehityksen joka vaiheessa, määrittelystä toteutukseen. Ideaalinen testaustoiminta olisi niin tuloksellista ohjelmistovirheiden ennaltaehkäisyn suhteen, että varsinaiselle testauksen suorittamiselle ei olisi tarvetta – kaikki ohjelmistovirheethän olisi löydetty ja korjattu jo testauksen suunnitteluvaiheessa. (Beizer 1990, 3–4.)

Edellä mainittua Beizerin esittämää ideaalista tilannetta ei ole kuitenkaan mahdollista tavoittaa, sillä inhimillisiltä virheiltä on mahdotonta välttyä kokonaan. Siinä määrin kun testaus epäonnistuu yhdessä tärkeimmistä tavoitteistaan, ohjelmistovirheiden ennaltaehkäisemisessä, sen täytyy päästä toissijaiseen päämääräänsä, ohjelmistovirheiden löytämiseen. Ohjelmistovirheet eivät ole aina ilmeisiä ja silmiinpistäviä. Ohjelmistovirhe osoitetaan odotetun lopputuloksen poikkeamalla. Jotta poikkeama havaittaisiin, on oltava olemassa dokumentaatio jonka avulla voidaan määritellä odotetut testien suorituskomennot. Yksi ohjelmistovirhe voi näyttäytyä monena eri oireena – ja päinvastoin: eri ohjelmistovirheillä voi olla sama ilmenemismuoto ohjelmistossa. Näin ollen oireet ja niiden aiheuttajat voidaan selvittää vain käyttämällä pieniä, yksityiskohtaisia testejä. (Beizer 1990, 4.)

Beizer (1990, 24–26) toteaa Mannaan ja Waldingeriin (1978) tukeutuen testauksella olevan mahdotonta todistaa ohjelmiston olevan täysin virheetön. Syiksi tähän Manna ja Waldinger (1978) pitävät seuraavia teoreettisia esteitä:

- ”Emme voi koskaan olla varmoja siitä, että määrittelydokumentaatio on oikeellinen.”

- ”Ei ole olemassa verifiointijärjestelmää, joka pystyy verifioimaan jokaisen ohjelmiston.”

- ”Emme voi koskaan olla varmoja verifiointijärjestelmän oikeellisuudesta.”

Testauksen tarkoituksen määrittämisessä voidaan käyttää Beizerin (1990, 4) esittämää asenteelliseen edistymiseen pohjautuvaa vaihejakoa:

- Vaihe 0: Testauksella ja virheenpoistolla (*debugging*) ei ole mitään eroa.

- Vaihe 1: Testauksen tarkoitus on näyttää toteen, että ohjelmisto toimii, kuten sen on tarkoitus toimia.

- Vaihe 2: Testauksen tarkoitus on osoittaa, että ohjelmisto ei toimi oikein.

- Vaihe 3: Testauksen tarkoitus ei ole todistaa mitään, vaan pienentää havaitut riskit toimimattomasta ohjelmistosta hyväksyttävälle tasolle.

- Vaihe 4: Testaus ei ole pelkkä toiminto, vaan myös ajattelutapa joka johtaa vähäriskisen ohjelmiston syntyyn ilman, että testaukseen tarvitsee käyttää suunnattomasti resursseja. (Beizer 1990, 4.)

2.3 Testaus perinteisissä ohjelmistokehitysmenetelmissä

Yleisen virheellisen harhaluulon mukaan perinteisten ohjelmistokehitysmenetelmien ohjelmistotestaus perustuu absoluuttisiin vaatimuksiin. Tosiasiassa testauksen ollessa käynnissä se suoritetaan monesti muuttuville tai jopa jo vanhentuneille vaatimuksille. Testaajan työnkuva perinteisissä ohjelmistokehitysmenetelmissä on ketterien menetelmien testausta suoraviivaisempaa. Testaajan tehtävänä on yksinkertaisesti varmistaa testattavan järjestelmän vaatimusten ja muun olennaisen dokumentaation avulla se, että ohjelmisto toimii kuten pitääkin. Hieman kärjistäen testaajan ei siis tarvitse huolehtia siitä, onko dokumentaation sisältö sitä, mitä asiakas oikeasti halusi – jos kaikki dokumentaation vaa-

timukset toteutuvat testauksessa, ohjelmiston laadun oletetaan olevan hyvää. (Agile 2009.)

Black (2007, 24–25) toteaa V-mallin olevan yleensä luonteeltaan aikataulu- ja budjettivetoinen. Aikataulujen pettäessä tai rahojen loppuessa on testaus yleensä osa-alue josta luovutaan ja tingitään, sillä suurin osa testauksesta suoritetaan projektin loppuvaiheessa. Suurissa projekteissa mahdollisuudet tarkkaan aikataulun suunnitteluun kuukausiksi tai jopa vuosiksi eteenpäin ovat erittäin pienet. (Black 2007, 24–25.)

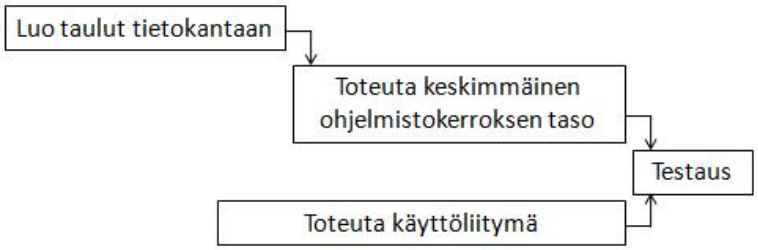
V-mallia käyttävien projektien muuttuminen testauskatastrofiksi projektin lopussa on mahdollista, mutta se voidaan kuitenkin estää. Ratkaisuksi Black (2007, 25) esittää seuraavia toimenpiteitä: Testauksen suunnittelu-, analysointi- ja toteutusvaiheet kannattaa aloittaa projektissa aikaisessa vaiheessa, jolloin virheiden aikainen havaitseminen ja korjaaminen on helpompaa. Ohjelmistokehityksessä tulisi pitää huolta myös siitä, että testiajojen ajanjaksot menevät sykleissä – tällöin saadaan aikaa virheenkorjauksia ja uusien ohjelmistoversioiden uudelleentestausta varten. Ominaisuuksien siirtäminen myöhempiin julkaisuihin on testausvaiheen päivämäärien siirtämistä parempi ratkaisu. (Black 2007, 25.)

Perinteisten ohjelmistomenetelmien testauksessa on kuitenkin useita haasteita. Vaikuttaisi siltä, että yleinen ongelma perinteisten ohjelmistomenetelmien testauksessa on sen jääminen yhdeksi viimeisimmistä suoritettavista asioista projektissa. Kaikissa menetelmissä tämä ongelma ei kuitenkaan realisoidu: Haikala ja Märijärvi (2006, 45) toteavat Evo-mallin jokaisen inkrementin tuloksena olevan uusilla ominaisuuksilla kasvatettu järjestelmä. Näin meneteltäessä esimerkiksi kriittisten suunnitelmien käyttökelpoisuutta päästään testaamaan jo projektin alkuvaiheessa (Haikala & Märijärvi 2006, 45).

Crispinin ja Gregoryn (2009, 10) mielestä perinteisiä ohjelmistokehitysmalleja käyttävien tiimien testaajien on käytettävä paljon aikaa vaatimusdokumentaati-

on läpikäymiseen, että testaussuunnitelmat olisi ylipäätään mahdollista kirjoittaa. Tämän jälkeen testaajien on kuitenkin odotettava kauan, ennen kuin ohjelmisto, jota varten testit kirjoitettiin, saadaan testattavaksi (Crispin & Gregory 2009, 10). Esimerkiksi vesiputousmallia käytettäessä ongelmia saattaa esiintyä juuri vaatimusmäärittelyn ja testausvaiheen välillä olevan pitkän aikajakson takia: projektin loppuvaiheessa löytyvät virheet johtavat helposti projektin viivästymiseen ja kustannusten ylittymiseen.

Cohnin (2006b, 13–14) mukaan aikaisessa vaiheessa aloitettua testausta hankaloittaa testauksen aloittamiseen vaadittavien asioiden keskinäiset riippuvuussuhteet. Kuvio 1 osoittaa testauksen aloittamisen edellyttävän samanaikaista yhtymäkohtaa keskimmäisen ohjelmistokerroksen toteutuksen, siihen vaikuttavan tietokannan taulujen lisäämisen, käyttöliittymän toteutuksen ja testaustausresurssien saatavuuden suhteen. Testaus aloitus viivästyy, mikäli yksikin edellä mainituista tekijöistä etenee hitaammin kuin oli suunniteltu. (Cohn 2006b, 14.) Samantyyppisiä testaukseen liittyviä haasteita on tunnistanut myös Glass (2003, 74), jonka mukaan suurin syy ohjelmistoprojektien ongelmiin ovat puutteet vaatimusmäärittelyssä. Koska vaatimusmäärittelyä on suurissa ohjelmistoprojekteissa mahdotonta saada absoluuttisen todenmukaiseksi ennen toteutusvaihetta, ollaan siis helposti ongelmissa (Glass 2003, 74). Myöhemmin esiteltävät ketterät menetelmät lähtevät siitä ajatuksesta, että muuttumatonta vaatimusmäärittelyä ei edes voida tehdä projektin alussa.



KUVIO 1. Testaukseen vaikuttavat, keskinäisessä riippuvuussuhteessa olevat tehtävät (Cohn 2006b, 13).

3 KETTERÄT MENETELMÄT JA TESTAUS

Nopeasti muuttuvat ympäristöt vaihtuvine vaatimuksineen ja tiukkoine aikatauluineen vaativat joustavampaa ohjelmistokehitystä, kuin mihin perinteiset ohjelmistokehitysmenetelmät monesti kykenevät. Jatkuvat muutokset pakottavat ohjelmistokehitysorganisaatiot ja tiimit toimenpiteisiin kilpailukyvyn säilyttämiseksi, sidosryhmien mieltymyksien toteuttamiseksi, uusien teknologioiden käyttöönottojen ja nopeiden markkinatoimitusten mahdollistamiseksi. Kaikki nämä yhdessä haastavat toden teolla perinteisten, suunnitelmakeskeisten ohjelmistokehitysmenetelmien käytön. (Cao & Ramesh 2007.)

Yksi ratkaisu edellä esitettyihin ohjelmistokehityksen haasteisiin on ketterät menetelmät. Ne pyrkivät minimoimaan riskejä, suosivat suoraa viestintää, hyväksyvät pitkin projektia tulevat muutokset ja pitävät toimivaa ohjelmistoa dokumentaatiota tärkeämpänä. Ketterien menetelmien tehokas käyttö sisältää kuitenkin haasteita, joista yksi vaikeimmista on ohjelmistotestaus.

Tässä luvussa luodaan katsaus ketterien menetelmien määritelmään, soveltuvuuteen ja testaukseen. Lisäksi luvussa vertaillaan ohjelmistokehitysmenetelmiä ja tutkitaan testaajan roolia yleisesti. Ketteristä menetelmistä Scrumia ei käydä läpi tässä luvussa, vaan se esitellään erikseen luvussa 4.

3.1 Yleistä

Monet perinteiset ohjelmistokehitysmenetelmät perustuvat tarkkaan määrittelyyn olettaen, ettei vaatimuksiin tulisi enää myöhemmissä vaiheissa muutoksia. Ohjelmistoprojektille on kuitenkin tyypillistä, että sen edetessä asiakkaalta tulee muutospyyntöjä. Tämä johtaa siihen, että muutoksia tulisi kyetä ottamaan vastaan ja hallitsemaan.

Ketterien menetelmien tärkeimpänä pidetyt asiat on kiteytetty neljään ketterän kehityksen arvoon. Kaksitoista ketterien menetelmien periaatetta puolestaan ilmaisevat tavat arvojen realisoimiselle. Käytännöt taas kertovat, miten periaatteita voidaan noudattaa ja toteuttaa käytännössä. Nämä kolme eri ketteriä menetelmiä kuvaavaa osa-aluetta esitellään seuraavaksi, aloittaen neljästä ketterän menetelmän arvosta Agile Manifestoa (2001) lainaten:

> ”Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan:
>
> 1. Yksilöitä ja vuorovaikutusta enemmän kuin prosesseja ja työkaluja
> 2. Toimivaa sovellusta enemmän kuin kokonaisvaltaista dokumentaatiota
> 3. Asiakasyhteistyötä enemmän kuin sopimusneuvotteluja
> 4. Muutokseen reagoimista enemmän kuin suunnitelman noudattamista
>
> Vaikka oikealla puolellakin on arvoa, arvostamme vasemmalla puolella olevia asioita enemmän.”

Ketterien menetelmien 12 periaatetta esitellään liitteessä 1. Ketterien menetelmien käytäntöjä on olemassa lukuisia ja jotkin käytännöt esiintyvät eri menetelmissä eri nimillä. Seuraavaksi esitellään muutamia yleisimpiä ja tunnetuimpia käytäntöjä.

Refaktorointi (*refactoring*) on menetelmä ohjelmakoodin laadun ja ylläpidettävyyden parantamiseksi koodin sisäistä rakennetta muuttamalla. Ohjelman rajapinnat tai käyttäjälle näkyvä toiminnallisuus ei muutu refaktoroitaessa. (Crispin & Gregory 2009, 496.)

Jatkuva integrointi (*continuous integration*) on prosessi, jossa ohjelmistoa koostetaan ja integroidaan jatkuvasti (Crispin & Gregory 2009, 486). Perinteisissä kehitysmalleissa osakokonaisuuksien integroiminen yhteen sijoittuu yleensä projektin loppupuolelle ja se tehdään kertaluonteisena toimenpiteenä. Crispinin ja Gregoryn (2009, 487) mielestä jatkuva integrointi on yksi ketterien menetelmien

tärkeimmistä asioista. Jatkuvan integroinnin suorittamiseen tarvittavat edellytykset ja ympäristöt tulisi siis hoitaa kuntoon hyvissä ajoin (Crispin & Gregory 2009, 487).

Testivetoisella kehityksellä (*test-driven development*, jäljempänä TDD) tarkoitetaan nimensä mukaisesti sitä, että ohjelmistoa kehitetään testilähtöisesti. TDD:tä käyttävä kehittäjä kirjoittaa pienen yksikkötestin toteutukselle, jonka aikoo toteuttaa. Testin ollessa valmis sitä suoritetaan ja toteutusta rakennetaan ja muutetaan niin kauan, että testi menee onnistuneesti läpi. (Crispin & Gregory 2009, 498.)

Hochmüller ja Mittermeir (2008) kyseenalaistavat ketterien menetelmien pitämisen kokonaan uutena asiana esittäen, että jotkut perinteisten ohjelmistokehitysmenetelmien puolustajat väittävät ketteryyden olevan vain uusi käsite uudelleenkeksityille, vanhoille käytännöille. Samaa mieltä ovat myös Larman ja Basili (2003) esittäessään olevan olemassa prosessimalleja, jotka vaikuttivat vuosia sitten olevan erittäin varteenotettavia vaihtoehtoja suunnitelmakeskeisille ohjelmistokehitysmalleille. Hochmüller ja Mittermeir (2008) pitävätkin inkrementaalisen ohjelmistokehityksen ja protoilun yhdistelmää yhtenä todennäköisenä vaihtoehtona ketterille menetelmien esi-isäksi ja suunnannäyttäjäksi. Useimmat ketteristä menetelmistä tarjoavat prosessikuvauksen lisäksi myös kattavan kokoelman suositeltuja käytäntöjä ja tekniikoita, joiden kuvaamisen myötä ketterien menetelmien käytön soveltamisesta halutaan tehdä helpompaa ja tehokkaampaa. (Hochmüller & Mittermeir 2008.)

Ohjelmistokehitysmenetelmiä kohtaan on vuosien saatossa esitetty kritiikkiä, eivätkä siltä välty ketterät menetelmätkään. Tichyn (2004) mukaan ketterien menetelmien kannattajat pitävät monesti ohjelmiston dokumentaatiota ja määrittelyjä asioina, jotka eivät tuota mitään lisäarvoa. Tämän seurauksena on tehty johtopäätöksiä siitä, että ketterä ohjelmistokehitys ei olisi mitään uutta vaan pelkästään anarkistinen vastareaktio kohti byrokraattisia, raskaita ohjelmisto-

kehitysmalleja, jotka vaativat kehittäjiltä runsaasti erinäisiä tuotoksia projektin eri osapuolille. (Tichy 2004.)

3.1.1 Ketterille menetelmille asetettuja kriteerejä

Ketterien menetelmien käyttöön tarvitaan muutakin kuin tietämystä niiden luonteesta, periaatteista ja käytännöistä. Lopputuloksen kannalta ratkaisevassa asemassa ovat tiimin koostumus ja asenne. Daviesin ja Sedleyn (2009, 123) mukaan ketteriä menetelmiä käyttävän tiimin tulee oppia työskentelemään yhteen siten, että tiimin tavoitteet täyttyvät. Tämä voi edellyttää sekä toteutettaviksi haluttavien toiminnallisuuksien että niiden vaatiman testauksen ymmärtämistä. Tiimin on työskenneltävä yhdessä sekä varmistettava, että kaikki edellä mainitut asiat täyttyvät ja tulevat tehtyä. (Davies & Sedley 2009, 123.) Ketterän tiimin ideaalisesta koosta ja koostumuksesta kerrotaan enemmän luvussa 5.5.3.

Millainen ohjelmistokehitysmenetelmän on oltava, että sitä voidaan sanoa ketteräksi? Abrahamssonin, Salon, Ronkaisen ja Warstan (2002, 98–99) mielestä ketterästä ohjelmistokehityksestä on kyse silloin, kun siitä on tunnistettavissa inkrementaalisuus, suoraviivaisuus, sopeutumiskyky viime hetken muutoksiin sekä laajamittainen yhteistyö ja kommunikaatio asiakkaiden ja kehittäjien välillä. Tichy (2004) on puolestaan esittänyt, että ketterät menetelmät korostavat aikaista ja jatkuvaa ohjelmiston tuotantoon saamista, toivottavat muutokset tervetulleiksi ja arvostavat asiakkaalta aikaisessa vaiheessa saatua palautetta. Ketterät menetelmät pyrkivät vähentämään tehottomuutta, byrokratiaa ja kaikkea sitä, millä ei ole todellista arvoa ohjelmiston kehitykseen. Käytössä olevat resurssit suunnataan siis varsinaiseen konkreettiseen ohjelmiston toteutukseen. (Tichy 2004.)

Crispin ja Gregory (2009, 46) ottavat kantaa siihen, milloin tiimit voivat aidosti kutsua itseään ketteriksi: tiimi, joka ei noudata ketterien menetelmien perusarvoja ja periaatteita, ei ansaitse ketteriä menetelmiä käyttävän leimaa itselleen. Myöskään se, että ohjelmistosta julkaistaan kuukausittain uusi versio ja doku-

mentointia väheksytään, ei vielä rinnasta toimintaa ketterään ohjelmistokehitykseen (Crispin & Gregory 2009, 46).

3.1.2 Ketterien menetelmien soveltuvuus

Seuraavaksi tarkastellaan sitä, katsotaanko ketteriä menetelmiä voitavan käyttää kirjallisuuden perusteella millaisissa projekteissa tahansa. Ketterien menetelmien skaalautumisella tarkoitetaan tässä yhteydessä menetelmän soveltamista, räätälöintiä ja muokkaamista niin pieniin kuin suuriinkin ohjelmistoprojekteihin.

Ketterien menetelmien soveltuvuudesta laajoihin ohjelmistoprojekteihin on tehty tapaustutkimuksia ja kirjoitettu useita artikkeleja (ks. esim. Gat 2006; Miller & Carter 2007). Kaikille tutkimuksista saaduille tuloksille on yhteistä ainoastaan toteamus: on olemassa haasteita, joita tulisi ratkaista.

Boehm ja Turner (2003a) esittävät ketterien menetelmien olevan täysin kykenemättömiä skaalautumaan laajoihin, kompleksisiin projekteihin. Turkin, Francen ja Rumpen (2002) mielestä ketterissä menetelmissä on rajoitteita koskien niin hajautettua ohjelmistokehitystä, turvallisuuskriittisten ohjelmistojen kehittämistä kuin suuria projektitiimejäkin. Turkin ym. (2002) mukaan ylläpidettävien viestintäkanavien määrä voi rajoittaa suurten ohjelmistokehitystiimien tehokkuutta. Suuret tiimit hyötyvät enemmän suunnitelmakeskeisten ohjelmistokehitysmenetelmien käytöstä, jolloin ohjelmistokehitys nojaa enemmän dokumentaatioon ja suunnitteluun, jotka hoidetaan useiden virallisten organisaatiossa käyttämien viestintäkanavien kautta. (Turk ym. 2002.)

Kirjallisuus ei anna ketterien menetelmien soveltuvuuteen ja skaalautuvuuteen yksiselitteistä vastausta. Hollerin (2006) mukaan on olemassa useita viitteitä siitä, että skaalautumisen ongelmat eivät ole ohjelmistokehitysmenetelmäspesifisiä. Mitä laajempi projekti on kyseessä, sitä suurempi mahdollisuus sillä on epäonnistua: projektiin osallistuvien ihmisten lukumäärän noustessa kasva-

vat myös mahdollisuudet väärin ymmärretystä viestinnästä tai kokonaisuuden kompleksisuudesta johtuvien riskien toteutumiseen. Ketterät menetelmät yksinkertaisesti hyväksyvät nämä realiteetit ja siitä johtuen suosittelevat sovelluskohteeksi pienempiä projekteja lyhyempine aikajaksoineen ja pienempine tiimeineen. Tämä ei kuitenkaan tarkoita sitä, että ketteriä menetelmiä käyttävien organisaatioiden ja tiimien tulisi välttää laajoja projekteja. Ketterien menetelmien ratkaisu suurten projektien hallintaan on pilkkoa projektit osaprojekteiksi tai -kokonaisuuksiksi, joita hallinnoivat ja johtavat erilaisista osaajista koostuvat tiimit itse (*cross-functional teams*). (Holler 2006.)

Muutoksenhallinta on tärkeä osa ohjelmistoprojekteja, ja sen tärkeys kasvaa ja hallinta vaikeutuu projektin koon kasvaessa. Perinteisiä ohjelmistokehitysmenetelmiä käyttävissä suurissa projekteissa epäselviä ja tulkinnanvaraisia muutospyyntöjä (*change request*) varten on olemassa muutoksenhallintaryhmä. Ketterissä menetelmissä kyseistä menettelyä ei ole, ja tyypillisesti uudet, asiakkaalta tulleet vaatimukset ja toiminnallisuudet otetaan mukaan projektiin automaattisesti toteuttamalla ne viimeistään seuraavassa pyrähdyksessä. Tällöin muutoksia otetaan mukaan ilman minkäänlaista analyysiä mahdollisista vaikutuksista ohjelmiston arkkitehtuuriin tai muihin toiminnallisuuksiin.

Beavers (2007) esittää artikkelissaan BMC Softwaren kohtaamia ongelmia uusien vaatimusten mukaan ottamisesta kehitettävään ohjelmistoon. BMC Software näki tarpeelliseksi ottaa Scrum-projektia tekevään tiimiin mukaan henkilö, jonka rooli oli toimia vaatimusarkkitehtinä. Vaatimusarkkitehdin työtehtäviin kuului uusien toiminnallisuuksien ja ominaisuuksien yksityiskohtainen ja huoliteltu saattaminen selkeiksi vaatimuksiksi. Tavoitteena uuden roolin perustamisessa oli pienentää kuilua uusien ominaisuuksien karkean kuvauksen ja yksityiskohtaisesti selitetyn vaatimuksen välillä – tavoitteen toteutuessa vaatimuksia olisi helpompi priorisoida ja hallinnoida. Uuden roolin mukaan tuominen onnistui, sillä BMC Softwaren vaatimustenhallinta parani. Kuilu ominaisuuksien ja yksityiskohtaisten vaatimusten välillä pieneni. Toinen suuri saavutettu etu

uudessa käytännössä oli kehittäjätiimille tarjolla oleva mahdollisuus vaikuttaa vaatimuksiin ja niiden prioriteetteihin. Vaatimusarkkitehdin roolin perustaminen johti myös yhteistyön ja kommunikaation paranemiseen tuotteen omistajien ja kehitystiimien välillä. (Beavers 2007.)

Abrahamssonin, Warstan, Siposen ja Ronkaisen (2003) mukaan ketterissä menetelmissä on puutteita myös projektinhallinnan suhteen: tutkimuksesta saadut tulokset osoittavat, että ketterät ohjelmistokehitysmenetelmät kattavat kyllä vaiheita ohjelmistokehityksen elinkaaresta mutta useimmat niistä eivät tarjoa riittävää tukea projektinhallintaan. Abrahamsson ym. (2003) jatkavat esittämällä Scrumin olevan eksplisiittisesti suunnattu ketterien menetelmien ohjelmistokehityksen projektinhallintaan, mutta sitä tulisi käyttää jonkin toisen, täydentävän ohjelmistokehitysmenetelmän kanssa. Schwaber ja Beedle (2002, 2) esittävät Extreme Programmingin (jäljempänä XP) olevan hyvä Scrumia täydentävä ohjelmistokehitysmenetelmä. Myös Waters (2008) pitää Scrumin ja XP:n samanaikaista käyttöä mahdollisena.

Scrumin ja XP:n lisäksi muutkin ketterät menetelmät suhtautuvat projektinhallintaan ja päätöksentekoon hieman toisistaan poikkeavasti. Abrahamssonin ym. (2003) mukaan Adaptive Software Development (ASD) kehottaa projektipäälliköitä joustamaan projekteissa oleviin muutoksiin. Toimintolähtöistä kehitystä (Feature Driven Development, FDD) käytettäessä projektipäälliköt ovat puolestaan oikeutettuja päättämään projektin laajuuteen, aikatauluun ja henkilöstöön liittyvistä asioista (Abrahamsson ym. 2003).

Edellä esitetyn perusteella voidaan todeta, että kullakin ketterillä menetelmillä on erilaiset vastaukset projektinhallintaan. Yleistä ohjeistusta projektipäälliköt eivät saa edes Agile Manifeston (2001) pääperiaatteista – ne on kohdistettu ihmisille, jotka ovat tekemisissä varsinaisen toteutuksen kanssa, kuten ohjelmistokehittäjille, testaajille ja arkkitehdeille.

Ohjelmistokehityksessä on jo kauan sitten huomattu tarve pystyä mittaamaan projekteja ja ohjelmistokehitysprosesseja. Mittaamiseen on olemassa metriikoita, joiden avulla voidaan päätellä esimerkiksi kehitettävän ohjelmiston kustannuksia (Kemerer 1987). Kemererin (1987) mukaan ohjelmiston tuottamisen kustannusten selvittämiseen käytettävät mallit pystyvät jopa 88 prosentin tarkkuuteen silloin, kun malli on kalibroitu huolellisesti sitä käyttävän organisaation tarpeisiin. Metriikoiden käytöstä ei olla kuitenkaan yksimielisiä, sillä Gilbin (2006) mielestä ketterät menetelmät eivät tue liiketoimintaan liittyvien metriikoiden hallintaa: kustannuksia on mahdotonta arvioida suhteessa liiketoiminnallisiin etuihin, jotka saadaan ketterien menetelmien käytöstä.

Aktiivisten ja tasaisesti etenevien ohjelmistoprojektien lisäksi on olemassa projekteja, jotka ovat luonteeltaan pitkäkestoisia ja joissa uusia toiminnallisuuksia ei ole koko ajan tarvetta tai edes mahdollista toteuttaa. Järjestelmä on kenties ollut tuotannossa jo vuosien ajan, mutta sitä ylläpidetään ja kenties myös jatkokehitetään satunnaisesti.

Haikala ja Märijärvi (2006, 41) määrittelevät ylläpidon olevan asiakkaan ongelmien ratkomista, virheiden korjaamista, ohjelman muuttamista vaatimusten muuttuessa sekä uusien piirteiden lisäämistä. Ylläpito voidaan jakaa karkeasti korjaavaan (*corrective*), adaptiiviseen (*adaptive*) ja täydentävään (*perfective*) ylläpitoon. Korjaavassa ylläpidossa korjataan ohjelman virheitä ja adaptiivisessa ylläpidossa muutetaan ohjelmaa, koska ympäristön vaatimukset ovat muuttuneet. Täydentävässä ylläpidossa puolestaan parannellaan ohjelmaa muuttamalla tai lisäämällä sen toiminnallisuuksia. (Haikala & Märijärvi 2006, 41.) Scrumia soveltuvampi ohjelmistokehitysmenetelmä edellä esitettyihin ylläpito- tai jatkokehitysprojekteihin voisi olla uusi menetelmä Scrumban, jota ei kuitenkaan aiherajauksen vuoksi esitellä tässä yhteydessä tarkemmin.

3.2 Ohjelmistokehitysmenetelmien vertailua

Pezzén ja Youngin (2008, 376) mukaan käytettävästä ohjelmistokehitysprosessista riippumatta säilyy voimassa seuraava perusolettamus: ohjelmistovirheen korjaamisen aiheuttamat kustannukset nousevat funktiona sille ajalle, joka kuluu ohjelmistovirheen syntymisestä (virheellisen koodin kirjoittaminen) sen havaitsemiseen. Tästä huolimatta, tai juuri tästä syystä, on tärkeää löytää organisaatiolle, tiimille tai projektille sopivin ohjelmistokehitysmenetelmä, jota käyttämällä aika edellä mainitulle ohjelmistovirheen elinkaarelle on mahdollisimman pieni. (Pezzé & Young 2008, 376.)

Seuraavaksi esitellään muutamia eroavaisuuksia, joita ketterien ja perinteisten ohjelmistokehitysmenetelmien vertailu nostaa esille. Ihanteellinen *toimintaympäristö* ketterien menetelmien ohjelmistokehityksen mukavuusalueeksi on dynaaminen ympäristö vaihtelevine vaatimuksineen ja teknologioineen. Perinteiset ohjelmistokehitysmenetelmät puolestaan ovat hyviä valintoja suhteellisen vakaaseen toimintaympäristöön. (Cao & Ramesh 2007.) Kuten taulukosta 1 voidaan todeta, perinteiset ja ketterät menetelmät vastaavat ohjelmistokehityksen eri osa-alueisiin monin tavoin erilailla.

Arvot, joita ketterät menetelmät nostavat esille, ovat vuorovaikutus, yhteistyö ja sopeutumiskyky, kun taas perinteisissä menetelmissä pidetään tärkeänä suunnittelua, ennustettavuutta, tarkkailua ja kontrollia. Ketterien menetelmien *oletamuksiin* kuuluu vaatimusten esiintulo läpi projektin, kun taas perinteiset menetelmät edellyttävät täydellistä ja tarkkaa vaatimusmäärittelyä ennen kehitysvaiheen alkamista. Ketterät menetelmät pitävät muutoksia väistämättömänä, joten niiden esiintyminen hyväksytään. Perinteisiä ohjelmistokehitysmenetelmiä käytettäessä puolestaan tarvitaan ponnisteluja muutoksen kokonaisuuden hallinnoimiseen, mistä syystä muutosten määrää pyritään minimoimaan. (Cao & Ramesh 2007.)

TAULUKKO 1. Perinteisten ja ketterien menetelmien vertailua (Nerur, Mahapatra & Mangalaraj 2005).

	Traditional	Agile
Fundamental Assumptions	Systems are fully specifiable, predictable, and can be built through meticulous and extensive planning.	High-quality, adaptive software can be developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change.
Control	Process centric	People centric
Management Style	Command-and-control	Leadership-and-collaboration
Knowledge Management	Explicit	Tacit
Role Assignment	Individual—favors specialization	Self-organizing teams—encourages role interchangeability
Communication	Formal	Informal
Customer's Role	Important	Critical
Project Cycle	Guided by tasks or activities	Guided by product features
Development Model	Life cycle model (Waterfall, Spiral, or some variation)	The evolutionary-delivery model
Desired Organizational Form/Structure	Mechanistic (bureaucratic with high formalization)	Organic (flexible and participative encouraging cooperative social action)
Technology	No restriction	Favors object-oriented technology

Edellä esitettyjen eroavaisuuksien tuloksena *ohjelmistokehityskäytäntöjen toteuttaminen* eroaa ketterien ja perinteisten ohjelmistokehitysmenetelmien välillä – huolimatta siitä, että jokaisella käytännöllä on pitkä historia ohjelmistotuotannossa. Ketterät menetelmät ovat vieneet jotkin käytännöistä äärimmäisyyksiin verrattuna perinteisten menetelmien käytäntöihin. Tästä hyvänä esimerkkinä on ketterien menetelmien tapa toimittaa ja julkaista ohjelmistoa: huolimatta siitä, että ohjelmistoja julkaistaan yleensä versioittain (*release*), niitä pyritään julkaisemaan paljon ja pienemmissä erissä. (Cao & Ramesh 2007.)

Vesiputousmalli ei tue vaatimusten muuttumista, joten uudelleen tehtävän työn kustannukset ovat korkeita. Vesiputousmallia käytettäessä asiakas kylläkin saa selkeän kokonaisarvion ja aikataulun jo projektin alussa, mahdollisesti jo ennen sen alkamista, mutta kustannusarviot ohjelmistoalalla ovat aina karkeita ja täten vain suuntaa-antavia. Todellisuudessa aiemmin esitetty arvio voi heittää moninkertaisesti verrattuna käytännön toteutumaan. Aikataulujen pettäessä testaus on yleensä eniten kärsivä osa-alue, sillä aikaa kurotaan umpeen juuri

testausvaiheesta. Testauksen vähälle jääminen puolestaan vaikuttaa ohjelmiston laatuun.

Smithin ja Sidkyn (2009, 33) mukaan yksi Scrumin selkeimmistä vahvuuksista on sen kyky tuottaa toteutettavan järjestelmän ominaisuuksia ja toiminnallisuuksia juuri niillä syklinopeuksilla ja siinä järjestyksessä, joiden avulla voidaan maksimoida liiketoiminnasta saatavat hyödyt. Lisäksi Scrumin puuttumattomuus pyrähdyksen aikana tapahtuviin käytäntöihin saa aikaan joustavuutta toteutukseen, kun tekemisiä ei ohjailla liikaa. Päivittäiset tapaamiset tuovat projektin tilanteeseen säännöllisesti läpinäkyvyyttä. (Smith & Sidky 2009, 33.) Scrumin ominaisuuksia ja vahvuuksia verrattuna muutamiin muihin ohjelmistokehitysmenetelmiin on kuvattu taulukossa 2.

TAULUKKO 2. Scrum verrattuna muutamiin muihin ohjelmistokehitysmenetelmiin (Agraval 2008).

	Waterfall	**Spiral**	**Iterative**	**SCRUM**
Defined processes	Required	Required	Required	Planning & Closure only
Final product	Determined during planning	Determined during planning	Set during project	Set during project
Project cost	Determined during planning	Partially variable	Set during project	Set during project
Completion date	Determined during planning	Partially variable	Set during project	Set during project
Responsiveness to environment	Planning only	Planning primarily	At end of each iteration	**Throughout**
Team flexibility, creativity	Limited - cookbook approach	Limited - cookbook approach	Limited - cookbook approach	**Unlimited during iterations**
Knowledge transfer	Training prior to project	Training prior to project	Training prior to project	**Teamwork during project**
Probability of success	Low	Medium low	Medium	**High**

Sutherland (2001) on esittänyt, ettei käytettävä teknologia rajoittaisi Scrumin käyttöä. Vaikka Scrum on vain projektinhallinnan viitekehys, on sen toimivuudessa varmasti eroja eri konteksteissa. Testauksen automatisointia pidetään ket-

terissä menetelmissä tärkeänä. Kun teknologiana on IBM:n Lotus Notes ja Domino, ei yksikkötestaus ole automatisoitavissa kovinkaan helposti.

Smith ja Sidky (2009, 33–34) näkevät Scrumissa myös heikkouksia. Ensimmäiseksi mainitaan Scrumin ideologia: Scrum-tiimi on itseohjautuva ryhmä, jonka yksilöt pystyvät tekemään kaikkia projektin toteutukseen liittyviä asioita. Ryhmä erikoisasiantuntijoita voi olla vaikeaa muuttaa tällaiseksi yleispäteväksi ryhmäksi. Toisekseen Scrum-tiimin menestys lepää liikaa Scrum-mestarin varassa, jolloin prosessin riippuvuus yhdestä yksilöstä on liian suuri. (Smith & Sidky 2009, 33–34.)

3.3 Ketterien menetelmien ohjelmistotestaus

Ketterät menetelmät korostavat ja painottavat testaajien aikaista osallistumista ohjelmiston kehitystyöhön ja sitä kautta saatua nopeaa palautetta, mutta kehittäjien ja laadunvarmistuksen välisestä suhteesta ei ole juurikaan kerrottu kirjallisuudessa (Williams & Cockburn 2003). Ketterät menetelmät eivät myöskään juuri sisällä yksikohtaisesti selitettyjä testauskäytäntöjä (Itkonen ym. 2005).

Scrumia käyttäessä hyväksytään se tosiasia, että kaikkia asiakkaan tai loppukäyttäjän tarpeita ei ole mahdollista ennustaa. Tämän myötä testaaja ei voi vaatia itselleen täydellistä, ajan tasalla olevaa vaatimusdokumentaatiota testausta varten. Vaatimusten kerääminen tehdäänkin kattavan dokumentaation luomisen sijaan monesti enemmänkin vaatimuksista keskustelemalla – keskustelut tuotteen omistajan ja tiimin kanssa ovat usein testaajan ensisijainen tapa selvittää kehitettävien ominaisuuksien tarkka toiminnallisuus. (Cohn 2009, 149.)

Daviesin ja Sedleyn (2009, 123) mielestä on yleistä, että tiimit aliarvioivat ajan, joka kuluu testaukseen ja siinä löytyvien ohjelmistovirheiden korjauksiin. Testaus ketterissä menetelmissä ei ole yhden ihmisen tehtävä, vaan vastuu siitä kuuluu koko tiimille. Jokaisella tiimin jäsenellä on erilaisia taitoja myötävaikut-

taa kehitettävän järjestelmän osakokonaisuuksien ja toiminnallisuuksien valmistumiseen, joka siis ketterissä menetelmissä sisältää myös huolellisen testauksen. (Davies & Sedley 2009, 123.)

Vaikka ketterien menetelmien pyrähdykset ovatkin lyhyitä, ei testauksen pyrähdykseen sijoittuminen ole yhdentekevää. Cohnin (2009, 308–309) mukaan ketterää menetelmää käyttävä tiimi voi kohdata haasteita, mikäli se jättää testauksen pyrähdyksen loppuun. Syynä tähän pidetään haasteita parantaa olemassa olevan toteutuksen laatua sen jälkeen, kun toteutus on jo valmis. Asioiden toimivuus voidaan vahvistaa vasta testaamisen myötä, joten mikäli testausta ei ole, on mahdollista, että samoja virheitä toistetaan monta kertaa ennen kuin ne huomataan. Testauksen puuttuessa toteutusvaiheesta myös palautteen antaminen vaikeutuu. (Cohn 2009, 308–309.) Vaikuttaisi siis olevan niin, että testaustyötä, niin suunnittelua kuin varsinaista testaustakin, tulisi siis tehdä tasaisesti läpi pyrähdyksen.

Adams ja Neustel (2006, 4) esittävät pyrähdys N:ssä tehtävän toteutuksen testauksen suoritettavaksi pyrähdyksen N+1 aikana, pyrähdyksessä N+1 tehtävän toteutuksen testauksen suoritettavaksi pyrähdyksen N+2 aikana ja niin edelleen. Kyseinen menettely takaisi toteuttajille työrauhan, mutta samalla se jakaa Scrum-tiimin kahtia. Tämä ei ole välttämättä hyvä asia, sillä tiimin eri rooleissa toimivien henkilöiden työvaiheiden ajoituksista tulisi päästä eroon, sillä se voi lisätä eriarvoisuutta. Lisäksi pyrähdyksessä N kehitettyjen toiminnallisuuksien valmistumisesta ja toimivuudesta ei saada varmuutta niin nopeasti kuin siihen olisi tarvetta. Kokonaisuudesta jää pois myös kehittäjien välittömästi saama palaute.

Ketteriä menetelmiä käyttävän projektin sidosryhmät voidaan ajatella myös yhdeksi joukoksi ihmisiä, jotka työskentelevät yhteisen päämäärän hyväksi. Davies ja Sedley (2009, 123–125) esittävät, että jonkinlaista luokittelua ja kuvausta projektitiimin eri rooleissa toimivien henkilöiden ja muiden sidosryhmien

välille on kuitenkin mahdollista tehdä myös testauksen suhteen. Näitä esitellään seuraavaksi Daviesin ja Sedleyn (2009, 123–125) mukaan.

Kehittäjien on varmistettava, että heidän kirjoittamansa koodi läpäisee toiminnallisuudelle asetetut hyväksymiskriteerit, ennen kuin koodi ja sen sisältämä toiminnallisuus menevät eteenpäin testattavaksi. Näin pyritään välttämään asiakkaiden ja testaajien ajan hukkaamista. Kehittäjiä on hyvä rohkaista automatisoimaan testauksesta niin paljon kuin mahdollista – siitä huolimatta, että itse tehtyjen ohjelmistovirheiden löytämiselle ollaan monesti sokeita. (Davies & Sedley 2009, 123–124.)

Asiakkaat ovat itse parhaiten perillä ympäristöstä, jossa heille tehtävää ohjelmistoa tullaan aikanaan käyttämään. Testaamisen kontekstissa asiakkaan fokus kohdistuu yleensä siihen, voiko käyttäjä saavuttaa tai suorittaa käyttäjätarinan (*user story*) tai toiminnallisuuden kuvauksen. On kuitenkin otettava huomioon, että käyttäjiltä voi jäädä helposti huomaamatta testauksessa erilaiset reunaehtoja sisältävät tilanteet ja toimenpiteet, joista syntyneet virhetilanteet järjestelmän tulisi kyetä käsittelemään hallitusti. Asiakkaan testausta tulisi kannustaa tarjoamalla heille testattavaksi ohjelmiston uusin versio aina, kun sille on tarvetta. (Davies & Sedley 2009, 124.)

Testaajat ovat parhaimmillaan turmiollisessa testauksessa (*destructive testing*), jossa järjestelmä pyritään saamaan virhetilaan käyttämällä sitä väärin tai toisin kuin on suunniteltu. Testaajat auttavat tiimiä suoriutumaan käyttäjätarinoiden testeistä ja vahvistavat näiden läpäisyn. Testaajat tarvitsevat monesti kehittäjien apua testauksen automatisoinnin toteuttamisessa. (Davies & Sedley 2009, 124.)

Ulkoiset tiimit, mikäli sellaisia on, suorittavat mahdollisesti erikoisempia testauksen muotoja kuten turvallisuustestaus, käytettävyystestaus tai eri ohjelmistoalustojen (*platform*) välinen testaus. Näiden testausten myötä löytyneet puutteet ja ongelmat ja niiden tarvitsemat korjaustoimenpiteet kannattaa ottaa huomi-

oon seuraavia pyrähdyksien aikatauluja suunnitellessa. (Davies & Sedley 2009, 124.)

Ketterissä menetelmissä testaajan työkuorma projektissa, tai mahdollisesti useissa projekteissa samanaikaisesti, on ainakin teoriassa melko hyvin ennustettavissa pyrähdysten kiertokulun ansiosta. Tästä huolimatta esimerkiksi yksittäisten toiminnallisuuksien muuttuessa testaajan on kyettävä vastaamaan äkkinäisiin testaustarpeisiin – on pystyttävä testaamaan mitä vaan ja milloin tahansa niin pienillä etukäteisvalmisteluilla kuin mahdollista.

Ketterissä menetelmissä suoritettavan testauksen haasteisiin törmätään kirjallisuuden perusteella varsin usein. Cohn (2009, 149) pitää pyrähdyksiä yhtenä suurimmista testaajan kohtaamista muutoksista ketteriin menetelmiin siirtyessä. Mikäli jokaista pyrähdystä ajatellaan omana pienenä projektinaan, pidetään testauksen ajankohta tällöin luonnollisesti kunkin pyrähdyksen lopussa. Tämä ei kuitenkaan ole Cohnin (2009, 149) mielestä hyvä ratkaisu, sillä testaus pääsee todennäköisesti alkamaan erittäin myöhään pyrähdyksen ja sen myötä muodostuu itse asiassa jokaiseen pyrähdykseen vesiputousmallinen ohjelmiston kehitysvaihe.

Nopeasti ja alati muuttuvassa ympäristössä on tärkeää, että testaaja tekee viisaita päätöksiä siitä, mitä testata kussakin vaiheessa ja milloin. On pelkkää ajanhukkaa, jos testaaja testaa tietämättään keskeneräisiä ominaisuuksia tai tekee kehittäjille vikaraportteja löydöksistä, jotka ovatkin oikeasti ominaisuuksia. Ajan tasalla pysymisen edellytyksiä ovat täten ainakin jatkuva kommunikointi kehittäjien kanssa sekä aktiivinen aamupalavereihin osallistuminen. Jotta testattava ohjelmisto tulisi tutuksi edes jossakin määrin, on sitä mahdollisuuksien mukaan järkevää käyttää tutustumismielessä, jolloin pääpaino ei ole välttämättä ohjelmistovirheiden löytämisessä. Ratkaisuksi tähän sopiikin tutkiva testaus.

Crispinin ja Gregoryn (2009, 39–40) mukaan testaajat, jotka eivät muuta lähestymistapaansa testaukseen, tulevat kohtaamaan vaikeita aikoja työskennelles-

sään kiinteästi osana kehitystiimiä. Lisäksi testaajat, jotka ovat tottuneet tekemään pelkästään manuaalista testausta käyttöliittymän kautta, eivät välttämättä ymmärrä automatisoitua lähestymistapaa joka on olennainen osa ketteriä menetelmiä. (Crispin & Gregory 2009, 39–40.)

Seuraavaksi pohditaan sitä, miten ketterät menetelmät voisivat mukautua perinteisten ohjelmistokehitysmenetelmien testaustasoihin. Koska ketterissä menetelmissä testausta tehdään läpi projektin, on esimerkiksi järjestelmätestauksen määritelmä häilyvä: sitä voidaan nähdä suoritettavan pitkin projektia, sillä jokaisen pyrähdyksen myötä valmistuu tuotantoon kelpuutettavaa ohjelmistoa. Toisaalta laajamittainen järjestelmätestaus on tehtävä projektin viimeisen pyrähdyksen jälkeen, kun kaikki osakokonaisuudet on lopullisesti integroitu yhteen. Osalla yleisistä ketterien menetelmien käytännöistä, kuten luvussa 3.1 esitellyllä refaktoroinnilla on tekemistä myös testauksen kanssa sen vaikuttaessa laatuun hyvin hoidettuna.

Yksikkötestaus. Yksikkötestaus vahvistaa järjestelmän pienen, yksittäisen osakokonaisuuden toimivuuden (Crispin & Gregory 2009, 499). Yleisenä yksikkötestausmenetelmänä ketterissä menetelmissä käytetään TDD:tä. Siinä kehittäjä kirjoittaa testin pienelle toiminnallisuudelle ennen sen toteuttamista. Ensin testi epäonnistuu, ja sitten kehittäjä jatkaa toteutusta niin kauan, että testi menee läpi onnistuneesti. (Crispin & Gregory 2009, 5.)

Järjestelmätestaus. Schwaber (1995) mainitsee järjestelmätestausta suoritettavan Scrumin pyrähdyksiä sisältävän, aktiivisen kehitysvaiheen jälkeisessä sulkemisvaiheessa. Koska ketterien menetelmien luonteeseen kuuluu jatkuva testaus, voidaan järjestelmätestausta nähdä suoritettavan jo aiemminkin, pyrähdysten aikana. Tästä huolimatta kehitetylle järjestelmälle suoritettava kokonaisvaltainen testaus, jossa voidaan varmistua osakokonaisuuksien yhteentoimivuudesta, on mahdollista tehdä vasta järjestelmän osakokonaisuuksien integroimisen jälkeen.

Hyväksyntätestaus. Crispin ja Gregory (2009, 493) esittävät ketterien menetelmien hyväksyntätestauksen määrittävän liiketoiminnallisen arvon tai tehtävän kullekin toiminnallisuudelle, joka järjestelmään toteutetaan. Hyväksyntätestit voivat kohdistua toiminnallisiin ja ei-toiminnallisiin eli niin sanottuihin laadullisiin vaatimuksiin kuten esimerkiksi järjestelmän suorituskykyyn ja luotettavuuteen. Hyväksyntätestausta voidaan käyttää niin liiketoimintaan kuin teknologiaan liittyviin asioihin. (Crispin & Gregory 2009, 493.)

Muut testaustyypit. Ei-toiminnallisia vaatimuksia voivat olla järjestelmän suorituskyvyn lisäksi myös turvallisuuteen tai käytettävyyteen asetetut vaatimukset. Suorituskykytestaus, käytettävyystestaus, turvallisuustestaus ynnä muut testaustasot ovat ketterissä menetelmissä yhtä tärkeitä kuin esimerkiksi vesiputousmallin mukaisesti kehitettävälle ohjelmistollekin. Crispin ja Gregory (2009, 14) suosittelevat edellä mainittuja testaustoimenpiteitä tehtävän niin aikaisessa vaiheessa kuin mahdollista, jolloin saadut tulokset ohjaavat suunnittelua ja toteutusta oikeaan suuntaan.

3.3.1 Testauksen automatisointi

Ohjelmistokehitysprojekteissa trendinä ovat kehitettävään ohjelmistoon kohdistuvien vaatimusten lisääntyminen, viime hetken muutokset ja asiakkaan toimittajaa kohtaan asettamat odotukset vastata muutoksiin nopeasti. Tämä johtaa luonnollisesti siihen, että ohjelmistoa on kyettävä testaamaan yhä enenevässä määrin ja lyhyemmässä ajassa. Kun ohjelmistojen koko ja kompleksisuus kasvavat, on selvää, etteivät manuaaliset prosessit tule takaamaan nopeita ja toistettavia tuloksia, jolloin ketterien menetelmien käyttöönotto voi kuulostaa houkuttelevalta automatisoinnin ollessa yksi menetelmien keskeisistä piirteistä.

Testauksen automatisointia pidetään elintärkeänä ketterissä menetelmissä jatkuvan palautteen saamiseksi (Crispin & Gregory 2009, 274–275; Cohn 2009, 311). Automatisoidun testauksen avulla voidaan säästää aikaa, tehostaa toteutusvaihetta ja saada testaukseen kattavuutta. Käyttämällä aikaa testien suunnit-

teluun voidaan välttää helpommin tilanteet, joissa on automatisoitu testejä, joiden avulla ei löydetä tehokkaasti virheitä. (Kaner, Bach & Pettichord 2002, 93.)

Crispin ja Gregory (2009, 258) näkevät yleisimmäksi syyksi testien automatisointiin sen, että kaikkien välttämättömien testien suorittamiseen manuaalisesti ei yksinkertaisesti ole aikaa. Craig ja Jaskiel (2002, 221) suosittelevat välttämään testauksen automatisointia niihin järjestelmän toiminnallisuuksiin, jotka muuttuvat usein. Perusteluna Craig ja Jaskiel (2002, 221) esittävät ylläpidettävyyden: usein muuttuvien toiminnallisuuksien automatisointi vie liikaa resursseja suhteessa siitä saataviin hyötyihin. Craigin ja Jaskielin esittämä mielipide on hieman ristiriidassa ketterien menetelmien automatisoinnin soveltamiskohteiden kanssa: testauksen automatisointi on ketterien menetelmien tärkeimpiä osa-alueita ja ketterissä menetelmissä jos missä kehitettävä järjestelmä ominaisuuksineen muuttuu jatkuvasti.

Testauksen automatisointi ei rajoitu tai kohdistu ketterissä menetelmissä ideaalitilanteessa yhteen ohjelmistokehityksen tasoon, kuten esimerkiksi yksikkötestaukseen. Cohnin (2009, 311–312) mukaan tehokas testauksen automatisointistrategia kattaa kolme tasoa: yksikkötestauksen, sovelluksen toimintoja sisältävän palvelukerroksen testauksen ja käyttöliittymätason testauksen. Eniten Cohnin (2009, 311–312) mielestä tulisi panostaa yksikkötestauksen automatisointiin, sillä se on testauksen automatisoinnin peruspilari. Automatisoitu käyttöliittymätestaus on puolestaan pienemmässä roolissa, sillä se on aikaa vievää ja sen ylläpito on kallista, sillä pienikin muutos käyttöliittymän toteutuksessa voi pilata useita testejä (Cohn 2009, 312).

Regressiotestauksella tarkoitetaan edelliselle ohjelmistoversiolle ajettujen testitapausten uudelleensuorittamista. Tavoitteena on löytää uusien toiminnallisuuksien ja tehtyjen korjausten mahdollisesti aiheuttamat ohjelmistovirheet (Pezzé & Young 2008, 29). Crispin ja Gregory (2009, 261) pitävät testauksen automatisointia regressiotestauksen turvaverkkona. Myös Tichi (2004) sekä Craig ja Jaskiel (2002, 217) pitävät automatisoitua regressiotestausta yhtenä ketterien

menetelmien avaintekijöistä. Regressiotestauksen automatisoinnin tehokkuudesta ei kuitenkaan olla täysin yhtä mieltä, sillä Kanerin ym. (2002, 101) mukaan automatisoidut regressiotestit löytävät yllättävän tehottomasti ohjelmistovirheitä.

Cohn (2009, 315) toteaa Scrumin käyttöön siirtyneiden tiimien sortuvan alussa yleiseen virheeseen: Pyrähdyksessä toteutettuja toiminnallisuuksia vastaavat automatisoidut testitapaukset toteutetaan vasta myöhemmissä pyrähdyksissä. Kun testit automatisoidaan koodin kirjoittamisen jälkeen, menetetään automatisoinnin arvosta iso osa. Täten automatisoidut testit ovat hyödyllisimpiä aktiivisella kirjoitushetkellä, jolloin koodi muuttuu jatkuvasti. Tällöin automatisoinnista saavutetaan maksimaalinen hyöty, sillä suurimman arvon saamiseksi myös kustannukset ovat alhaisimmat. (Cohn 2009, 315.)

Testauksen automatisoinnilla on helpompi ennustaa automaattisen testiskriptin suorittamisen vaatima työmäärä kuin manuaalisen testaamisen vaatima resurssitarve. Automatisoimalla regressiotestaus puolestaan varmistetaan sen kuuluminen mukaan pyrähdyksiin – manuaalinen regressiotestaus on helposti osa-alue, josta projektissa tingitään tiukan aikataulun vuoksi. Testauksen automatisoinnin ulkopuolelle jäävät ei-toiminnalliset vaatimukset ja käytettävyyteen liittyvät käyttäjätarinat, kuten esimerkiksi seuraava: ”Peruskäyttäjä kykenee vaivatta tekemään järjestelmällä monipuolisia hakuja.” Myös Crispinin ja Gregoryn (2009, 285) mielestä käytettävyystestaus kannattaa suorittaa manuaalisesti.

Wellens (2008) käsittelee artikkelissaan testauksen automatisoinnin haasteita ja kompastuskiviä. Monien testaustyökalut mainostetaan olevan tallenna ja toista -työkaluja (*record & playback*). Tällainen työkalun nimeäminen johtaa kuitenkin helposti harhaan: automatisointityökalun ymmärretään kykenevän nauhoittamaan ja toistamaan testitapaukset vaivattomasti ja nopeasti ilman ongelmia. Kuitenkin, käytännössä harva työkalu ymmärtää sujuvasti eri ympäristöjen

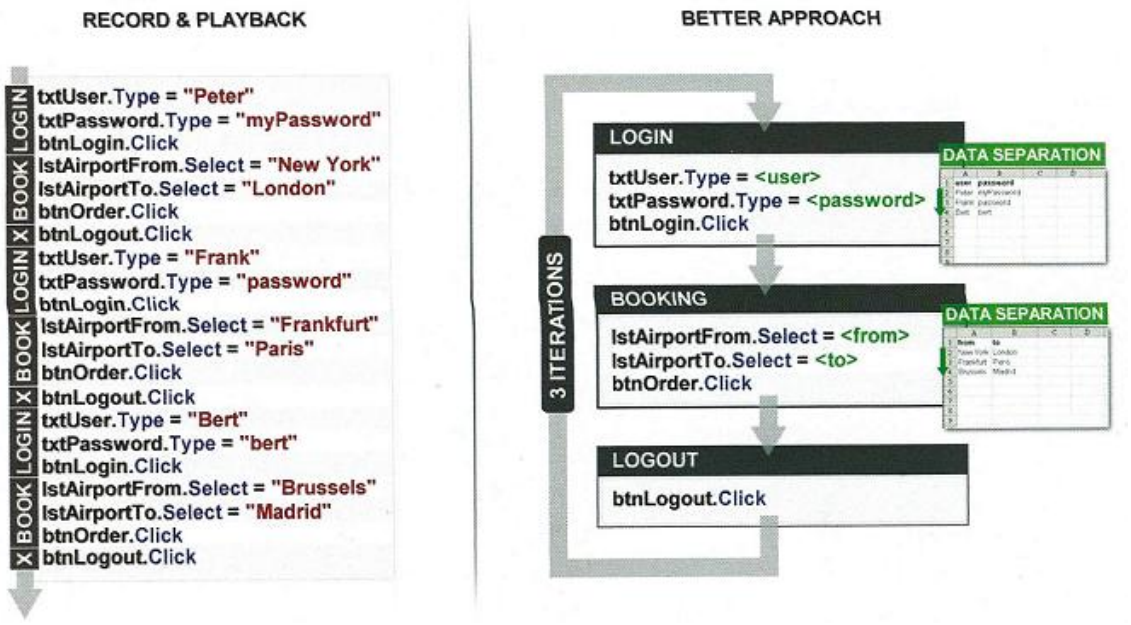
teknologioita, minkä vuoksi testitapausten ylläpito on aikaa vievää. (Wellens 2008.)

Karhu, Repo, Taipale ja Smolander (2009) ovat tutkineet testauksen automatisointiin vaikuttavia tekijöitä. Tutkimukseen valittiin erityyppisiä organisaatioita: mukana oli niin tuotekeskeistä ohjelmistokehitystä, räätälöityä järjestelmäkehitystä kuin testauspalvelujakin tarjoavia organisaatioita. Tulosten mukaan testauksen automatisoinnin laaja käyttö on mahdollista silloin, kun testattavat järjestelmät eivät ole kompleksisia ja ne eivät ole riippuvuuksia kolmannen osapuolen ohjelmistoista. Myös alhainen kynnys automatisointiin ryhtymiseen, vankkana perustana oleva teknologia ja uudelleenkäytettävyys tukevat testauksen automatisoinnin käyttöä. (Karhu ym. 2009.)

Testauksen automatisoinnin suurimmat saavutetut hyödyt ovat laadun parantuminen paremman testauskattavuuden myötä sekä mahdollisuus suorittaa testausta enemmän vähemmässä ajassa (Karhu ym. 2009). Tästä huolimatta Karhun ym. (2009) mielestä paras testikattavuus saavutetaan silloin, kun automatisoitujen testien lisäksi testitapausten valinnassa otetaan mukaan myös manuaalista testausta. Suurimmat testauksen automatisoinnin haittapuolet ovat kustannukset, jotka tulevat automatisoinnin toteutuksesta, ylläpidosta ja koulutuksista. Haasteeksi koetaan myös automatisoidun testauksen viemät resurssit: testit ovat harvoin pysyvästi valmiita, eli niiden ylläpitoon vaaditaan henkilön työpanosta. (Karhu ym. 2009).

Seuraavaksi esitettävä Wellensin (2008) esimerkki osoittaa sen, että automatisoidut testit kannattaa suunnitella huolellisesti. Esimerkkinä testauksen automatisoinnista on lennonvarausjärjestelmän testitapaus, jossa kirjaudutaan vuorotellen järjestelmään kolmella eri käyttäjätilillä, varataan lento pisteestä A pisteeseen B ja kirjaudutaan ulos. Sen sijaan, että nauhoitus tehtäisiin staattisesti ja yhdeksi pitkäksi kokonaisuudeksi, Wellens (2008) esittää paremmaksi vaihtoehdoksi luoda lyhyempi skripti, joka sisältää kolme osiota: kirjautumisosion käyttäjätunnuksineen ja salasanoineen, varsinaisen varauksen tekemisen syöt-

teineen ja uloskirjautumisen. Tämän lisäksi vaihdetaan staattiset tiedot dynaamisiksi arvoiksi, jotka haetaan tietovarastosta ajonaikaisesti. Näin on saatu automatisoitu testi pienemmällä määrällä koodia ja samalla testin ylläpidettävyys on manuaalista testausta parempi (KUVIO 2). (Wellens 2008.)



KUVIO 2. Automatisoidun testitapauksen osittaminen ja dynaamisuus (Wellens 2008, 22).

3.3.2 Tutkiva testaus

Vaikuttaisi siis siltä, että ketterien menetelmien testaukseen ei välttämättä ole saatavilla ajan tasalla olevaa järjestelmää kuvaavaa dokumentaatiota. Lisäksi yksityiskohtaisten testitapausten kirjoittamiseen ei välttämättä ole aikaa, tai vaikka olisikin, niin kirjoitettujen testitapausten käyttöarvo olisi vähäistä. Seuraavaksi esitellään tutkivaa testausta, joka vaikuttaisi sopivan luonteensa puolesta yhdeksi ratkaisuvaihtoehdoksi näihin haasteisiin.

Bach (2002) määrittelee tutkivan testauksen samanaikaisesti tapahtuvaksi oppimiseksi, testien suunnitteluksi ja niiden suorittamiseksi. Hän laajentaa määritelmää esittämällä tutkivassa testauksessa testaajan aktiivisesti kontrolloivan testien muotoa niitä suoritettaessa keräten samalla informaatiota suunnitellakseen uusia ja parempia testitapauksia (Bach, 2002b). Loveland ym. (2005, 339)

puolestaan käyttävät tutkivasta testauksesta nimitystä taiteellinen testaus. Tällä he halunnevat korostaa lukijalle tutkivan testauksen toteutuksen ja sisällön olevan tutkivaa testausta suorittavan henkilön intuition varassa. Lovelandin ym. (2005, 240) mukaan testaaja tietää testattavasta ohjelmistosta tai sen osakokonaisuudesta aina enemmän testauksen lopussa kuin sen alkuvaiheessa.

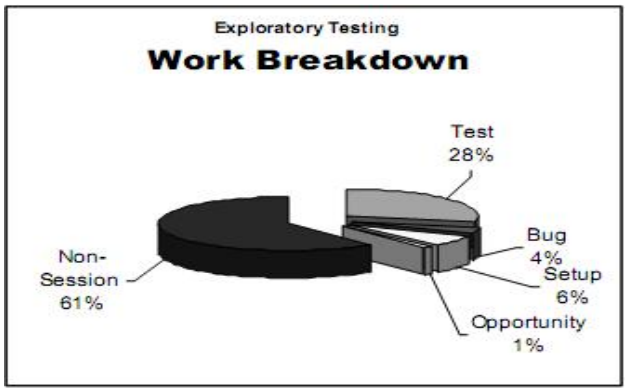
Haasteena tutkivassa testauksessa on löytää vastaus siihen, missä menee dokumentoinnin järkevä raja ilman, että tutkivan testauksen ominaispiirre, vapaus, kärsii. Jonkinlaista dokumentaatiota olisi hyvä tehdä, sillä projektitiimin on oltava tietoisia siitä, mitä on jo testattu ja pystyttävä täten arvioimaan tuloksia. Kun saman ohjelmiston testaukseen käytetään tutkivaa testausta useamman henkilön toimesta mahdollisesti jopa samanaikaisesti, voi testauksen kattavuuden ja kokonaistilanteen seuranta olla haasteellista.

Bach (2000) on tunnistanut edellä kuvatun ongelman tutkivan testauksen seurannasta ja kehittänyt lähestymistavan mitata ja hallita tutkivaa testausta. Bach (2000) käyttää menetelmästä nimeä sessiopohjainen testauksen hallinta (*session-based test management*). Sen ideana on helpottaa testauksen seurantaa saamalla testauksesta konkreettisia tuloksia ja raportteja (Bach 2000).

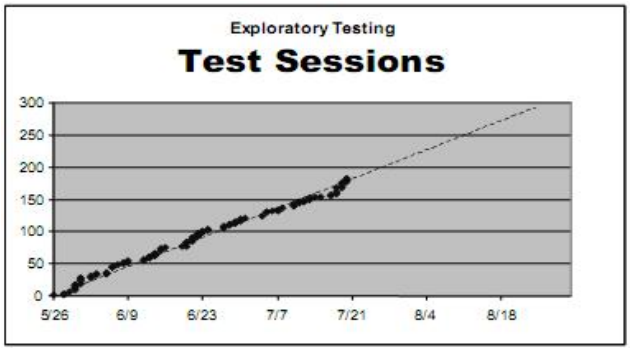
Sessiopohjainen testauksen hallinta voidaan käsittää rakenteelliseksi tutkivaksi testaukseksi, mikä saattaa kuulostaa ristiriitaiselta tutkivan testauksen vapautta ajatellen. Rakenteellisuus ei kuitenkaan tarkoita sellaista testauksen ennalta suunnittelua, joka rajoittaisi tutkivan testauksen valttikortteja, eli joustavuutta ja mahdollisuutta luovuuteen. Rakenteellisuudella tarkoitetaan sitä, että on olemassa joukko odotuksia sille, millaista testaustyötä tullaan tekemään, ja kuinka se raportoidaan. (Bach 2000.)

Sessiopohjaista testauksen hallintaa sovellettaessa testaus tehdään sessioissa, joiden kesto vaihtelee 45 minuutista useisiin tunteihin. Sessio on ominaisuuksiltaan tarkoituksenmukainen, raportoiva ja yhtäjaksoinen. Tarkoituksenmukaisuudella tarkoitetaan session sisältävän määrittelyn siitä, mitä testataan tai mitä

ongelmia erityisesti etsitään. Session on oltava raportoitava, eli siitä syntyy lyhyt kuvaus ikään kuin muistioksi. Yhtäjaksoisella sessiolla puolestaan halutaan varmistua siitä, ettei testausjaksoa keskeytä esimerkiksi palaverit tai puhelut. Sessio voi sisältää esimerkiksi ohjelmiston toiminnon tarkastelua, tietyn ongelman paikannusta tai ohjelmistovirhekorjausten toimivuuden tarkastamista. Jokainen sessio käydään suullisesti läpi. Kun testaussessioita katselmoidaan ja niiden sisältöä tallennetaan asiakirjoiksi, tarjoavat ne mahdollisuuden arvioida jatkossa samantyyppisten testaustöiden työmääräarviot. Tutkiva testaus siis jakaantuu moniin erilaisiin tehtäviin (KUVIO 3). Kuten kuvion 4 perusteella voidaan todeta, ennustettavuus ja suunnittelu paranevat aiemmin toteutuneiden testaussessioiden seurannan avulla. (Bach 2000.)



KUVIO 3. Testaajan työajan jakautuminen (Bach 2000).



KUVIO 4. Testaussessioiden ennustaminen aiemmin toteutuneiden perusteella (Bach 2000).

Bachin idea tutkivan testauksen seurantaan on hyvä, sillä ohjelmistoprojekteissa on aina tärkeää pystyä mittaamaan ja seuraamaan tehdyn työn etenemistä ja tehokkuutta. Tästä huolimatta herää helposti kysymys siitä, missä määrin seu-

rantaa voidaan tehdä siten että tutkiva testaus voitaisiin aidosti kokea intuition varassa tapahtuvaksi suunnittelemattomaksi testaukseksi.

Tutkivaa testausta pidetään yleisesti ohjelmistovirheiden löytämisen menetelmänä. Cohnin (2009, 314) mukaan tutkivan testauksen suorittamisella voidaan tarkastella myös automatisoitujen testitapausten kattavuutta. Lisäksi voidaan auttaa tiimiä löytämään asioita, jotka ovat aiemmissa vaiheissa vaikuttaneet hyviltä ideoilta – mutta joiden toteutuksen myötä onkin havaittu tarve muuttaa suunniteltua toteutusta järkevämmäksi (Cohn 2009, 314).

Kun mietitään luvussa 3.1 esitetyn TDD:n periaatetta, voitaneen tutkivan testauksen luonnetta ja ideologiaa tarkasteltaessa todeta sen olevan puolestaan testivetoista testausta. Tätä tukee ainakin Bach (2001) esittäessään tutkivassa testauksessa edellisen ajetun testin tuloksen vaikuttavan seuraavan testin kohteeseen ja luonteeseen. Tutkivaa testauta ei voida varsinaisesti pitää uutena menetelmänä, sillä sitä lienee harjoitettu enemmän tai vähemmän tiedostamatta tilanteissa, joissa on kehitystiimin toimesta ” kokeiltu vaan nopeasti” ohjelmaa tai sen tiettyä toiminnallisuutta ilman minkäänlaista dokumentaatiota.

Yksi tutkivan testauksen selkeimmistä vahvuuksista on sen vapaus: testaajalle tarjotaan mahdollisuus suorittaa ennalta määräämättömiä operaatioita intuitiivisesti. Toisekseen se on erinomainen testausstrategia käytettäväksi myös suunnitelmakeskeisissä menetelmissä aiemmin löydettyjen ohjelmistovirheiden korjauksien läpikäymiseksi nopeasti ja tehokkaasti, mikäli läpikäytäviä asioita on rajallinen määrä.

Haasteena tutkivassa testauksessa voitaneen pitää tilanteita, joissa testauksen kohteena on kompleksinen järjestelmä, jossa käyttäjän suorittaman toimenpiteen tulos ei ilmene suoraan esimerkiksi käyttöliittymästä. Esimerkkinä tästä voidaan pitää Notes-tietokannan asetusten muuttamista, jonka seurauksena käyttöliittymässä pitäisi mahdollisesti tapahtua muutoksia, mutta testaajalla ei ole asiasta täyttä varmuutta. Asian selvittäminen vaatii joko kehittäjän apua tai

ohjelmistovirheraportin tekemistä, joista jälkimmäinen lienee huono vaihtoehto – etenkin jos ohjelmisto toimii kuten pitääkin. Kolmantena vaihtoehtona on toki dokumentaation tutkailu, mutta jos kyseessä on ketteriä menetelmiä käyttävä projekti, on todennäköisyys melko pieni sille, että testaaja löytää ajan tasalla olevan dokumentaation josta saa vastauksen ongelmaansa.

Tutkiva testaus tuo testaajalle myös muita haastavia tilanteita, joista yksi on testaajan vastuu pysyä testauksessa järkevällä tasolla: järjestelmän testauksessa ei kannata mennä teknisesti liian pitkälle. Tämä on otettava huomioon ainakin tilanteissa, joissa käytettävissä oleva aika on vähissä: oletusarvoisesti testaus kannattaa kohdistaa tärkeimpiin ja useimmin käytettyihin toimintoihin, jotka tuskin sijaitsevat järjestelmän uumenissa. Tutkiva testaus voidaan nähdä myös suoraan johdannaiseksi Agile Manifestosta, sillä tutkivan testauksen myötä jos minkä voidaan tarjota ketterää muutoksiin reagoimista ja edistää toimivaa ohjelmistoa, dokumentoinnin ja suunnitelman painottamisen sijaan. Itkosen ym. (2005) mielestä tutkiva testaus on tehokas tapa sovellusten testaamiseen loppukäyttäjän näkökulmasta ja sitä käyttämällä pystytään löytämään ohjelmistovirheitä tehokkaasti suhteessa testaukseen käytettyyn aikaan.

Sen sijaan, että testatessa noudatettaisiin kirjoitettua testaussuunnitelmaa liian tarkasti, kannattaa testaajan kiinnittää huomiota suoritettujen toimintojen ympärillä tapahtuviin muihinkin asioihin ja yksityiskohtiin. Intuitio ja sen mukaan käyttäytyminen on asia, jota ei voida opettaa koneelle. Vaikka tutkiva testaus perustuu pitkälti intuition varassa toimimiseen, se ei sulje pois työkalujen käyttöä tutkivan testauksen apuna: hyvä tutkivan testauksen työkalu ei korvaa ihmisen vuorovaikutusta, vaan tehostaa havainnointia. Tämän seurauksena testaajien on esimerkiksi helpompaa jäljittää ja selvittää toimenpiteet, jotka vaaditaan suoritettavan vaikeasti toistettavien ohjelmistovirheiden esiin saamiseksi. (Crispin & Gregory 2009, 210–211.)

Itkonen, Mäntylä ja Lassenius (2007) ovat verranneet testitapauspohjaisen ja tutkivan testauksen välistä ohjelmistovirheiden löytämisen tehokkuutta. Tut-

kimuksen tulosten perusteella testitapauspohjainen testaus tuotti huomattavasti enemmän virheellisiä eli aiheettomia ohjelmistovirheraportteja kuin tutkiva testaus. Toisena tutkimustuloksena havaittiin, että ennalta suunniteltujen testitapausten käyttämisestä ei ollut hyötyä virheiden löytämisen tehokkuuden kannalta. Näiden kahden edellä mainitun testausmenetelmän välillä ei ollut eroa löydettyjen ohjelmistovirheiden tyyppiin, vakavuuteen tai havaitsemisvaikeuteen. (Itkonen ym. 2007.)

3.3.3 Kuka testaa?

Ketterissä menetelmissä testaus ja kehitettävän tuotteen laadunvarmistus ei ole pelkästään testaajan vastuulla, vaan siitä huolehtii koko tiimi (Cohn 2009, 323; Crispin & Gregory 2009, 15). Crispin ja Gregory (2009, 105) jatkavat toteamalla, että testaajat kyllä ottavat päävastuun etenkin liiketoiminnallisten toiminnallisuuksien testaamisesta, mutta kehittäjät ovat aktiivisesti mukana suunnittelemassa ja automatisoimassa testejä.

Tuomikoski ja Tervonen (2009) ovat tutkineet testauksen mukaan ottamista Scrumiin. Kohlin vuonna 2005 kirjoittaman artikkelin rohkaisemana he päättivät kokeilla tutkivan testauksen tehokkuutta tapauksessa, jossa sitä käyttäisi koko tiimi. Käytännön toteutus testaukseen hoidettiin siten, että yksi testaajista järjesti tutkivan testauksen session koko tiimille pyrähdyksen lopussa. Aluksi testaaja piti lyhyen esittelyn tutkivan testauksen ideologiasta. Löytyneitä ohjelmistovirheitä tai sellaisina pidettyjä toiminnallisuuksia kirjattiin järjestelmän sijaan muistilapuille. Muutaman tunnin pituisen testaussession päätteeksi löydöksiä tai sellaisiksi luultuja sisältäneet laput kerättiin, ja ohjelmiston sen hetkisestä valmiudesta ja tilasta keskusteltiin lyhyesti tiimin kesken. Tilaisuuden lopuksi testaaja kävi kaikki laput läpi, erotellen oikeat ohjelmistovirheet ja käytettävyysongelmat sekä poistaen samalla toisteiset löydökset. (Tuomikoski & Tervonen 2009.)

Tuomikoski ja Tervonen (2009) toteavat testaussession menneen erittäin hyvin: osallistujien mielestä sessio oli hyvää vaihtelua normaaleihin päivärutiineihin ja se kannusti keskusteluun ja avoimuuteen. Löydettyjen ohjelmistovirheiden määrä oli yllätys positiivisessa mielessä: virheitä löydettiin enemmän kuin aikaisemmin, mikä kertoi testauksen tehokkuudesta. Testaukseen osallistuneilla kehittäjillä oli erittäin hyvä ymmärrys koodin heikoista kohdista, ja he kohdistivatkin testauksen juuri näihin alueisiin. Session päätteeksi monet siihen osallistuneet olivat sitä mieltä, että testaukseen tulisi jatkossakin kiinnittää enemmän huomiota. (Tuomikoski & Tervonen 2009.)

3.3.4 Ketterän testauksen haasteita ja avoimia kysymyksiä

Itkonen ym. (2005) vertailevat artikkelissaan ketterien ja perinteisten ohjelmistokehitysmenetelmien testausta ja toteavat perinteisten menetelmien testauskäytäntöjen soveltamisen ketteriin menetelmiin tuovan haasteita. Artikkelissa osoitetaan ketterien menetelmien yleisten pääperiaatteiden ja luonteen sekä perinteisten ohjelmistokehitysmenetelmien testauskäytänteiden ristiriidat. Tätä avataan tarkemmin sekä vertailemalla ketterien menetelmien pääperiaatteita perinteiseen ohjelmistotestaukseen että asettamalla ketterien ja perinteisten ohjelmistokehitysmenetelmien testauskäytännöt samalle viivalle. (Itkonen ym. 2005.)

Perinteiset testauskäytännöt eroavat monista ketterien menetelmien testauksen yleiskuvasta. Perinteisten ohjelmistokehitysmenetelmien testaukseen kuuluu monesti testauksen riippumattomuus ja sen muusta ohjelmistokehityksestä erillään pitäminen, kun taas ketterissä menetelmissä kehittäjät kirjoittavat testejä omalle koodilleen. Lisäksi testaaja on ketterissä menetelmissä osa kehitystiimiä, ja monissa tilanteissa roolien rajat ovat häilyviä. (Itkonen ym. 2005.)

Perinteisten ohjelmistokehitysmenetelmien testauksen tarkoituksena on löytää virheitä käyttämällä ohjelmaa keskittyneesti ja määrätietoisesti. Ketterien menetelmien testaus on yleensä rakentavaa, eli laatu rakennetaan tuotteen sisään. It-

konen ym. (2005) toteavat ketterissä menetelmissä luotettavan testauksen automatisointiin erittäin paljon. Perinteisten menetelmien testausammattilaiset näkevät tärkeämmäksi katselmoida testauksen tuloksia saadakseen selville löydettyjä virheitä ja oppiakseen niistä. Ketterien menetelmien automatisoitu testaus ei ole perinteisten menetelmien testaajien mielestä kovin tehokas tapa paljastaa ohjelmistokoodin virheitä. (Itkonen ym. 2005.)

Itkonen ym. (2005) ovat tutkineet haasteita, kun perinteisten ohjelmistokehitysmenetelmien testauskäytäntöjä sovitetaan yhteen ketterien menetelmien periaatteiden kanssa. Saatujen tulosten yhteenveto esitetään taulukossa 3. Siinä esitetyn ensimmäisen löydöksen mukaan ketterien menetelmien periaatteena on toimivan ohjelmiston toimitus lyhyin väliajoin. Tämä tuo haasteita perinteisten ohjelmistokehitysmenetelmien testauskäytäntöihin, sillä testaukselle jää vähän aikaa ja pyrähdykset ovat pituudeltaan kiinteitä aikajaksoja. Vaikka pyrähdykseen kuuluvassa testauksessa löydettäisiinkin runsaasti ohjelmistovirheitä, on silti mentävä eteenpäin kohti seuraavaa pyrähdystä.

Toisekseen ketterissä menetelmissä muutokset ovat arkipäivää, joten perinteisen testauksen suunnittelu on hankalaa ilman kunnollista dokumentaatiota. Kolmantena ristiriitana on kommunikaatio ja tiedon kulkeminen: ketterissä menetelmissä uskotaan kasvotusten tapahtuvaan kommunikointiin, jolloin osapuolet ovat tiiviisti tekemisissä toistensa kanssa. Tällöin perinteisten testausmenetelmien käyttämää dokumentaatiota ei välttämättä ole olemassa ja yksityiskohdat testausten odotetuista tuloksista ovat lähinnä hiljaista tietoa. (Itkonen ym. 2005.)

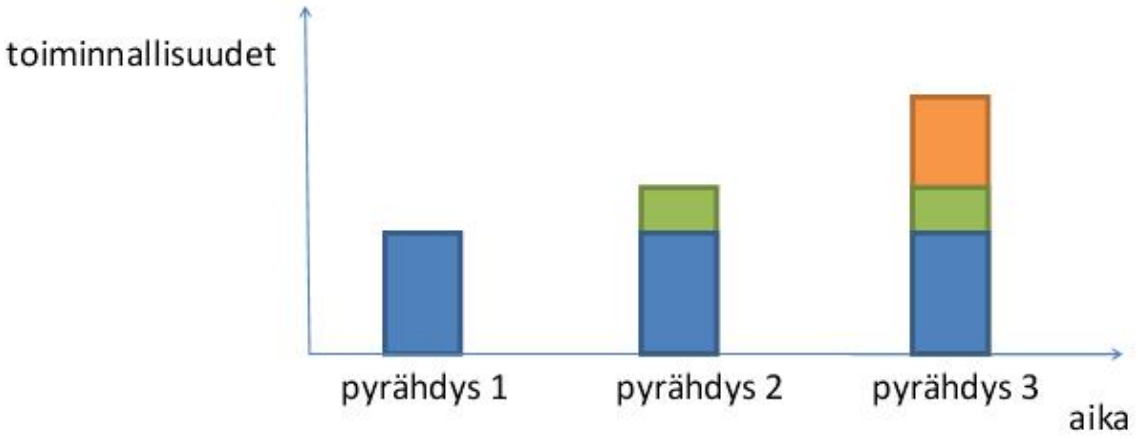
Neljänneksi, jos edistymistä mitataan toimivalla ohjelmistolla, ei testausta voida jättää pyrähdyksen lopun vaiheeksi. Syynä tähän on edistymisen ehto: testauksen on tuotettava ajoissa tietoa toteutuksen sen hetkisestä laadusta mahdollistaakseen päättelyt siitä, onko toteutus toimiva kokonaisuus vai ei. Edellä mainittujen lisäksi, viidentenä, ketterien menetelmien ajatus yksinkertaisuudesta

voi johtaa monimutkaisten ja aikaa vievien testauskäytäntöjen pois jättämiseen. (Itkonen ym. 2005.)

TAULUKKO 3. Ketterien menetelmien periaatteiden tuomat haasteet perinteisten ohjelmistomenetelmien testauskäytäntöihin (Itkonen ym. 2005).

Ketterien menetelmien periaate	**Haaste perinteisen testauksen näkökulmasta**
Arvokasta, merkityksellistä ohjelmistoa valmistuu pienissä erissä, mutta usein	Testauksen vähäinen aika pyrähdyksissä ja aikarajojen joustamattomuus
Muutokset hyväksytään, vaikka ne tulisivat ilmi missä tahansa kehitysvaiheessa	Testaus ei voi perustua valmiisiin ja luotettaviin määrityksiin ja dokumentaatioon
Usko kasvotusten tapahtuvaan kommunikointiin	Yksityiskohtainen informaatio testien odotetuita tuloksista on hiljaista tietoa: se on kehittäjien ja liiketoiminnasta vastaavien ihmisten päässä
Toimiva ohjelmisto on edistymisen pääasiallinen mittari	Laatuun liittyvä informaatio vaaditaan pyrähdyksen alussa läpi kehityksen
Yksinkertaisuus on ehdotonta	Testauskäytännöistä luovutaan helposti, sillä niitä ei pidetä riittävän yksinkertaisina

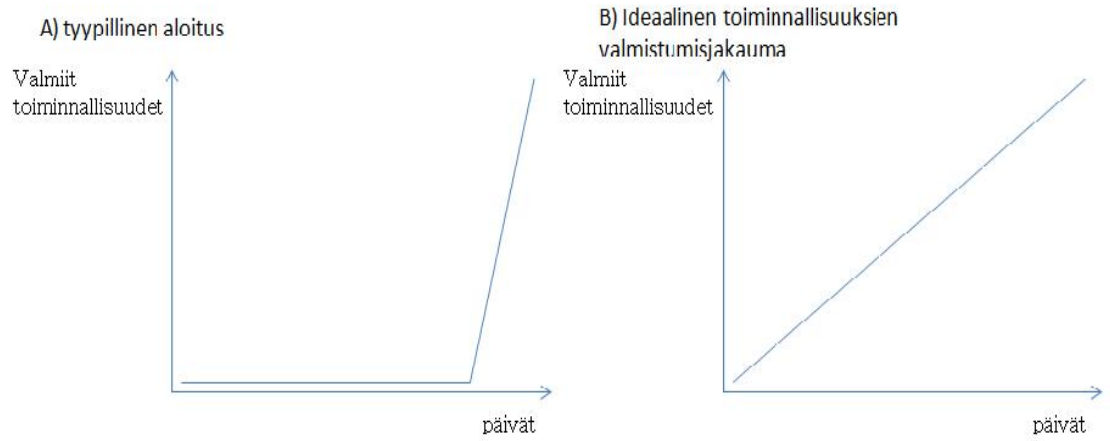
Kuvio 5 havainnollistaa yhtä ketterien menetelmien haastetta, joka realisoituu puutteellisen testauksen automatisoinnin myötä. Kuvion 5 palkkien kukin eri väri edustaa eri pyrähdysten toiminnallisuuksia. Kun projekti etenee, kehitettävän järjestelmän toiminnallisuuksien määrä kasvaa pyrähdysten myötä. Tällöin vaarana on tilanne, jossa perustoiminnallisuuksia, jotka on tärkeää testata jokaisen pyrähdyksen aikana, ei saada automatisoitua. Tämä puolestaan johtaa usein siihen, että toiminnallisuuksien määrä on lopulta niin suuri, ettei niitä käytännössä ehditä testaamaan pyrähdyksen aikana.



KUVIO 5. Testaustyön kasvaminen projektin edetessä.

Ketterissä menetelmissä muutosta tapahtuu koko ajan. Tämä voi olla testaajalle sekä turhauttavaa että henkisesti raskasta. Kuitenkin, muutosten vastapainoksi, ketterien menetelmien testaus helpottaa ajankäytön suunnittelua: lähtökohtaisesti testausta suoritetaan jokaisessa pyrähdyksessä ennalta sovitun määrän verran. Tämä eroaa kuvitteellisen projektin vesiputousmallisesta testausjaksosta, joka ei koskaan toteutunut, koska ohjelmiston toteutus venyi ja aikataulut johtivat testaustuntien radikaaliin vähenemiseen. Kuten yllä esitetyn kuvion 5 avulla havainnollistettiin, pyrähdysten testaukseen varattua tuntimäärää on todennäköisesti tarpeen lisätä projektin etenemisen myötä.

Cohnin (2009, 207) mielestä on varsin tavanomaista, että pyrähdykseen valitut kehitettävät toiminnallisuudet valmistuvat vasta pyrähdyksen lopussa. Tällaisissa projekteissa työskenteleville testaajille ei välttämättä ole juurikaan testattavaa pyrähdyksen alkuvaiheilla, kun taas viimeisten päivien aikana kaikki toiminnallisuudet pitäisi pystyä testaamaan nopeasti ja tehokkaasti. Yhtenä ratkaisuna edellä esitettyyn haasteeseen Cohn (2009, 207–208) neuvoo luomaan sellaisen taulukon, joka havainnollistaa pyrähdykseen valittujen toiminnallisuuksien valmistumiset. Tämän myötä tiimin jäsenet huomaavat pian orastavan ongelman ja toivon mukaan alkavat etsiä itseohjautuvasti ratkaisuvaihtoehtoja siihen, kuinka pyrähdyksen toiminnallisuudet saataisiin nopeammin valmiiksi (Cohn 2009, 207). Näitä molempia skenaarioita on havainnollistettu alla olevassa kuviossa 6.



KUVIO 6. Pyrähdyksessä valmistuvat toiminnallisuudet (Cohnia 2009, 208 mukaillen).

3.4 Testaajan rooli

Pettichordin (2000) mukaan monet ohjelmistokehittäjät eivät ymmärrä, kuinka vaikeaa järjestelmien kattava ja tehokas testaaminen voi olla. Testaaminen vaatii kärsivällisyyttä ja joustavuutta. Testaajalla tulee olla ammattitaitoa sekä järjestelmän yksityiskohtien ymmärtämiseen että sen kokonaiskuvan hahmottamiseen. Monet testaajat turhautuvat työskennellessään testausta helppona työnä pitävien kehittäjien kanssa. (Pettichord 2000.)

Patton (2006, 19) esittää ohjelmistotestaajan tavoitteiksi löytää ohjelmistosta virheitä mahdollisimman aikaisessa vaiheessa sekä varmistua siitä, että virheet korjataan. Huomionarvoista on kuitenkin se, että virheen korjaamisella ei välttämättä tarkoiteta aina ohjelmiston kooditason muokkaamista oikeaksi, vaan kyseessä voi olla vaikkapa kommentin lisääminen käyttöohjeeseen tai lisäkoulutuksen järjestäminen asiakkaalle (Patton 2006, 19).

Ohjelmistoa testatessaan testaajan on toisinaan hyvä asettua kehitettävän ohjelmiston loppukäyttäjän tai asiakkaan rooliin. Hyvältä testaajalta voidaan odottaa osaamista hyödyntää järjestelmän dokumentaatiosta ilmenevät vaatimukset, mutta myös valmiutta suunnitella ja toteuttaa järjestelmän kokonaisvaltainen testaus myös ilman dokumentaatiota. Lisäksi kyky päättää tapauskohtaisesti

milloin testausta on tehty riittävästi, on tärkeää. Testaajan miettiessä esimerkiksi hakutoiminnon testaamista syöttämällä hakukenttään satoja merkkejä, reagoi kehittäjä tilanteeseen todennäköisesti seuraavalla tavalla: ”No mutta ei kukaan koskaan tule tekemään noin, ja ei sen edes tarvitse toimia kyseisellä tavalla! … ja sitä paitsi se toimii minun koneellani…”

3.4.1 Hyvän ohjelmistotestaajan vaatimukset

Seuraavaksi listataan muutamia hyvän ohjelmistotestaajan ominaisuuksia Pattonin (2006, 20) mukaan:

- Testaajat ovat tutkijoita. Heitä ei pelota heittäytyä tuntemattomiin tilanteisiin. He rakastavat tilanteita, joissa he saavat uuden ohjelmiston tai sen osan, pääsevät asentamaan sen tietokoneelleen sekä seuraamaan, mitä tapahtuu.

- He ovat ongelmanratkaisijoita ja hyviä hahmottamaan miksi jokin asia ei toimi. He rakastavat ongelmia, ”pähkinöitä”.

- He ovat periksiantamattomia. Ohjelmistotestaajat jatkavat yrittämistä loputtomiin. Ohjelmistovirhe, jonka he näkevät, saattaa hävitä tai olla vaikeaa toistaa. Ohjelmistotestaaja ei luovuta, vaan yrittää kaikin voimin toistaa vian.

- He ovat luovia. Pelkän itsestään selvyyksien testaaminen ei ole ohjelmistotestaajille tarpeeksi. Heidän tehtävänsä on ajatella luovasti ja löytää virheitä myös tarkastelemalla asioita odottamattomista näkökulmista.

- Testaajat ovat hillitysti perfektionisteja. He pyrkivät täydellisyyteen tietäen kuitenkin milloin täydellinen tulos on saavuttamattomissa, jolloin he ovat tyytyväisiä päästyään niin lähelle kuin mahdollista.

- Testaajilla on hyvä arviointikyky. Heidän on tehtävä päätöksiä siitä, mitä he testaavat, kuinka kauan siinä menee ja onko tarkasteltava ongelma oikeasti ohjelmistovirhe vai ei.

- He ovat hienotunteisia ja diplomaattisia. Ohjelmistotestaajat ovat henkilöitä, jotka tuovat muille yleensä huonoja uutisia. Heidän tehtävänään on kertoa ohjelmistokehittäjälle, että heidän kehittämässään toteutuksessa on ongelma. Hyvä ohjelmistotestaaja osaa ilmaista asian oikeaoppisesti ja tietää kuinka työskennellä sellaisten ohjelmoijien kanssa, jotka eivät aina ole hienotunteisia ja diplomaattisia.

- He ovat suostuttelukykyisiä. Virheitä, jotka testaajat löytävät, ei aina pidetä tarpeeksi vakavana, että niitä korjattaisiin. Testaajien tulee olla hyviä ilmaisemaan pyrkimyksensä selvästi, perustelemaan miksi ohjelmistovirhe tulisi ehdottomasti korjata ja seuraamaan korjauksen läpivientiä.

Yleisesti ajatellaan, että ohjelmistotestaus on prosessi, joka perustuu sellaisten testitapausten suorittamiseen, jotka on huolellisesti suunniteltu käyttäen jotain testitapausten suunnittelutekniikkaa. Juristo, Moreno ja Vegas (2004) ovat asiasta eri mieltä esittäessään testaajien taitojen ja tietämyksen olevan tärkeässä roolissa myös testauksen suoritusvaiheessa. Testaajien taidoilla ja testitapausten suunnittelutekniikoilla on yhtä suuri vaikutus testauksesta saataviin tuloksiin (Juristo ym. 2004).

Pettichordin (2000) mukaan ohjelmistokehitys vaatii erilaisia kykyjä ja rooleja ja hänen mielestään tehokkaassa tiimissä testaajat ja kehittäjät täydentävät toisiaan, tuottaen perspektiivejä ja taitoja, joita toiselta osapuolelta saattaa puuttua. Pettichord (2000) pitää hyvinä testaajina henkilöitä, joilla on monia sellaisia luonteenpiirteitä, jotka ovat täysin vastakkaisia hyvillä kehittäjillä tarvittaviin luonteenpiirteisiin.

Esimerkkinä täydentävistä taidoista ja luonteenpiirteistä ovat muun muassa asenne ajoittain toistuvia työtehtäviä kohtaan: hyvät testaajat ymmärtävät ja oppivat hyväksymään sen, että testaus voi toisinaan olla itseään toistavaa. Monet kehittäjät puolestaan inhoavat toistuvia työtehtäviä ja pyrkivät välttelemään niitä esimerkiksi automatisoinnin avulla. Erilainen asenne ja pohdinta testaajan ja ohjelmistokehittäjän välillä nousee todennäköisesti esille myös kehittäjän löytämää ohjelmistovirhettä analysoitaessa. Ohjelmistokehittäjän mielenkiinto voi kohdistua ohjelmistovirheen esiintymistodennäköisyyteen loppukäyttäjällä. Testaajaa tämä ei kiinnosta: tärkeämpää on löydetyn ohjelmistovirheen vakavuus ja sen korjaaminen. (Pettichord 2000.)

3.4.2 Testaajien oikeudet

Crispin ja House (2002, 31–32) ovat esittäneet testaajien oikeuksia, joiden avulla heidän on mahdollista yltää yhtä hyviin suorituksiin kuin kaikkien muidenkin XP-tiimin jäsenten. Koska testaajat katsovat kehitettävää ohjelmistoa monesti asiakkaan näkökulmasta, heidän oikeutensa peilautuvat osittain asiakkaiden oikeuksista (Crispin & House 2002, 31). Seuraavaksi esitellään testaajien oikeuksia Crispinin ja Housen (2002, 31–32) mukaan.

- *”Testaajalla on oikeus nostaa esiin laatuun ja prosessiin liittyviä asioita ja ongelmia milloin tahansa.*

- *Testaajalla on oikeus esittää kysymyksiä asiakkaille ja kehittäjille ja saada vastauksia kohtuullisessa ajassa.*

- *Testaajalla on oikeus kysyä ja saada apua keneltä projektitiimin jäseneltä tahansa, sisältäen kehittäjät, esimiehet ja asiakkaat.*

- *Testaajalla on oikeus tehdä ja päivittää omiin tehtäviinsä liittyviä arvioita ja saada nämä sisällytetyksi käyttökuvausten arviointeihin.*

- *Testaajalla on oikeus työkaluihin joita hän tarvitsee selviytyäkseen työtehtävistäsi kohtuullisessa ajassa.*

- *Testaajalla on oikeus odottaa itsensä lisäksi myös projektitiimiltään vastuullisuutta laadusta huolehtimiseen."*

3.4.3 Ketterän testaajan periaatteet

Crispin ja Gregory (2009, 21–31) esittelevät kymmenen, osittain Agile Manifeston periaatteista johdettua ketterän testauksen periaatetta, joita he pitävät tärkeänä. Mikäli ketterässä tiimissä toimiva testaaja pyrkii noudattamaan näitä ohjeita, tulee hänen muun muassa olla itseohjautuva ja suosittava kasvotusten tapahtuvaa kommunikointia. Lisäksi muutoksiin on kyettävä reagoimaan nopeasti ja pyrittävä jatkuvaan tekemisen parantamiseen. (Crispin & Gregory 2009, 23–24, 29.)

Crispinin ja Gregoryn esittämät periaatteet ovat hyviä ohjenuoria testaajalle ketteriä menetelmiä käyttävään projektiin. Periaatteet sisältävät testaajille suunnattuja suosituksia ja oikeuksia, joista molemmista on hyvä pitää kiinni. Tutkielman aihe- ja tilarajoitusten vuoksi periaatteisiin ei perehdytä tässä yhteydessä tarkemmin. Sen sijaan seuraavaksi siirrytään käsittelemään tutkimuksen kannalta olennaista Scrum-ohjelmistokehitusmenetelmää.

4 SCRUM

Kuten jo johdannossa todettiin, toimeksiantajaorganisaation pilottiprojekteissa käytetty ketterä menetelmä oli Scrum. Tässä luvussa käydään läpi Scrumin prosessia, rooleja, käytäntöjä ja tapaamisia. Tässä yhteydessä ei kuitenkaan käsitellä vielä Scrumin käyttöönottoa, sillä ketterien menetelmien käyttöönottoon perehdytään tarkemmin luvussa 5.3.

4.1 Yleistä

Termi Scrum juontaa juurensa alun perin rugbysta, jossa sillä tarkoitetaan aloitusryhmitystä. Termiä käytettiin kirjallisuudessa tiettävästi ensimmäisen kerran Hirotaka Takeuchin ja Ikujiro Nonakan vuonna 1986 julkaisemassa artikkelissa, jossa esitellään joustava, nopea ja itseohjautuva ohjelmistokehitysprosessi Japanista. (Schwaber & Beedle 2002, 2.)

Scrum on kehitetty ohjelmistojen ja järjestelmien kehityksen prosessinhallintaan. Se on empiirinen lähestymistapa, joka käyttää teollisuuden prosessinhallintateorian ideoita järjestelmien kehittämiseen. Tuloksena on ohjelmistokehitykseen soveltuva lähestymistapa, joka sisältää joustavuutta, sopeutumiskykyä ja tuottavuutta. (Schwaber & Beedle 2002, 1–2.) Scrum ei määrittele mitään spesifisiä ohjelmistokehitystekniikoita ohjelmistojen toteutukseen tai testaukseen, vaan keskittyy siihen, kuinka tiimin jäsenten tulisi toimia voidakseen tuottaa ohjelmistoja ja järjestelmiä joustavasti alati muuttuvassa kilpailuympäristössä (Abrahamsson ym. 2002, 27). Schwaber ja Beedle (2002, 2) esittävät, että kun projekti tehdään Scrumia käyttäen, on kehitettävän järjestelmän toimivia toiminnallisuuksia valmiina jo kuukauden sisällä projektin aloitusajankohdasta.

Scrumin keskeisimmän käsityksen mukaan ohjelmistokehitykseen liittyy useita ympäristöön liittyviä ja teknisiä tekijöitä ja muuttujia, kuten vaatimukset, aikataulut, resurssit ja teknologia. Muutokset tekijöissä projektin aikana ovat to-

dennäköisiä. Näistä syistä aiheutuvien huonon ennustettavuuden ja kompleksisuuden vuoksi ohjelmistokehitysprosessilta vaaditaan joustavuutta, jotta se kykenisi vastaamaan kehitysprojektiin kohdistuviin muutoksiin ja haasteisiin. (Abrahamsson ym. 2002, 27.) Scrumin avulla voidaan parantaa organisaation ohjelmistokehitystä, sillä Scrum sisältää useita aktiviteetteja, jotka tähtäävät yhdenmukaisesti tunnistamaan mitä tahansa niin ohjelmistokehitysprosessissa kuin käytännöissäkin esiintyviä puutteita tai esteitä (Abrahamsson ym. 2002, 28).

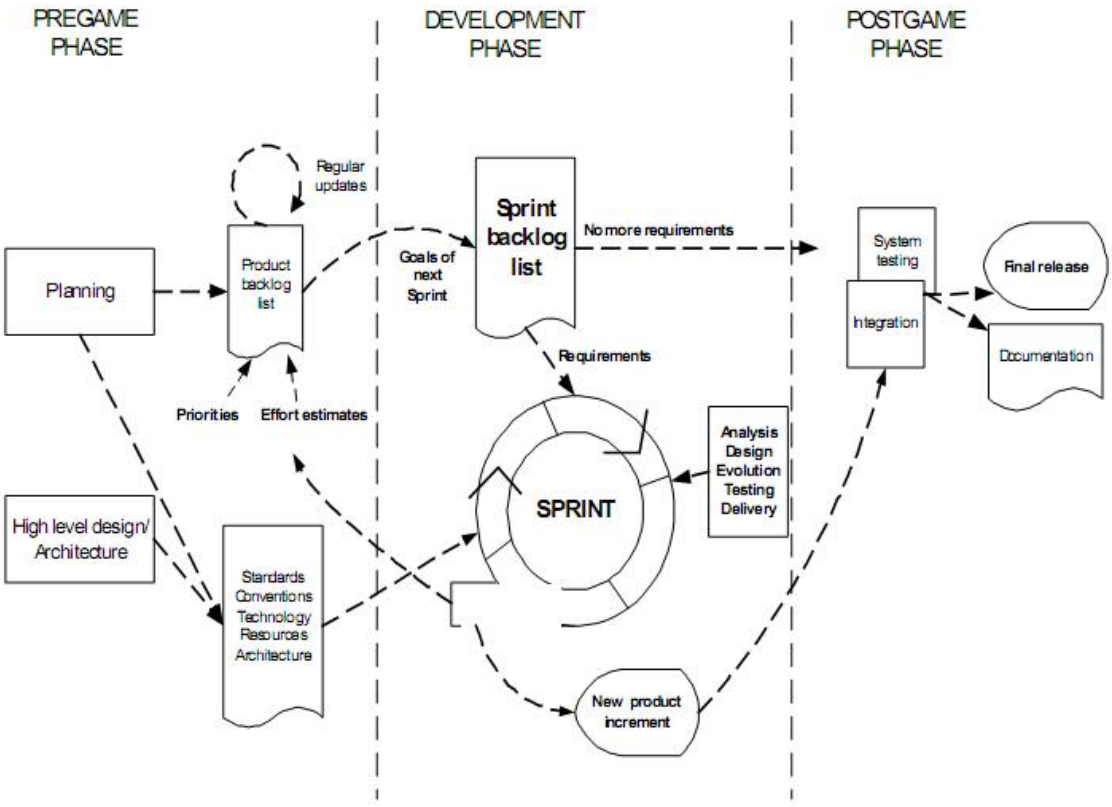
4.2 Prosessi

Scrumin prosessi sisältää kolme vaihetta: alustus (*pre-game*), kehitys (*development*) ja viimeistely (*post-game*). Vaiheet on esitelty Scrumin prosessia esittävässä kuviossa 7 ja ne käydään seuraavaksi yksitellen läpi omissa aliluvuissa 4.2.1–4.2.3.

4.2.1 Alustus

Alustusvaihe sisältää kaksi alivaihetta: suunnittelun sekä arkkitehtuurin ja korkean tason mallien laatimisen. Ensimmäisen suunnittelualivaiheen, tuloksena saadaan määritelmä uudesta, projektin myötä toteutettavasta järjestelmästä tai tuotteesta. Vaiheen aikana luodaan tekninen ja toiminnallinen tuotteen työlista (*product backlog*) (ks. luku 4.4), joka sisältää kaikki vaatimukset, jotka ovat tiedossa tässä vaiheessa projektia. Vaatimukset voivat olla peräisin niin asiakkaalta, myyntihenkilöstöltä, asiakastuesta kuin ohjelmistokehittäjiltäkin. Vaatimukset priorisoidaan ja niiden toteutukseen tarvittava työmäärä arvioidaan. Tuotteen työlistaa päivitetään jatkuvasti priorisoimalla, uusilla ja yksityiskohtaisemmilla asioilla ja niihin liittyvillä tarkemmilla arvioilla. Suunnitteluvaihe sisältää myös kuvaukset projektitiimistä, tarvittavista työkaluista ja muista resursseista, riskienhallinnasta ja hallinnallisista asioista, perehtymistarpeista sekä hyväksymiskäytännöistä. (Abrahamsson ym. 2002, 29.; ks. myös Schwaber

1995.) Kniberg (2006, 9) pitää tuotteen työlistaa Scrumin keskipisteenä. Tuotteen työlistan sisältö pyritään kuvaamaan asiakkaan terminologiaa käyttäen, millä pyritään varmistamaan se, että asiakas sisäistää kaikki projektin kehitettävät toiminnallisuudet ja muutokset (Kniberg 2006, 9).



KUVIO 7. Scrumin prosessi (Abrahamsson ym. 2002, 28).

Arkkitehtuuri- ja korkean tason mallin laatimisen myötä saadaan selville tarkemmalla tasolla, kuinka tuotteen ominaisuuslistassa olevat asiat tullaan toteuttamaan. Vaiheen aikana pyritään myös tunnistamaan välttämättömät muutokset, jotka on tehtävä tuotteen työlistassa oleville toteutettaville osakokonaisuuksille. (Abrahamsson ym. 2002, 29.) Lisäksi pidetään syntyneen korkean tason mallin katselmointipalaveri, jossa eri osapuolet esittelevät kuinka lähteä toteuttamaan tai tarvittaessa vielä ensin muuttamaan työlistan osakokonaisuuksia (Schwaber 1995).

4.2.2 Kehitysvaihe

Kehitysvaihe (*development phase*), joka tunnetaan myös pelivaiheena (*game phase*), on Scrumin varsinainen ketterä osuus. Kehitysvaiheessa järjestelmää kehitetään pyrähdyksissä. Ympäristöön ja teknologiaan liittyviä tunnistettuja muuttuvia tekijöitä, jotka saattavat muuttua projektin myötä, on paljon. Siksi muuttuvia tekijöitä pyritään havaitsemaan ja hallitsemaan pyrähdyksen aikana lukuisten erilaisten Scrum-käytäntöjen avulla. (Abrahamsson ym. 2002, 29.)

Scrum tähtää epävarmuustekijöiden ja mahdollisten muutosten läpi projektin kestävään kontrollointiin. Tällöin tarvittaviin muutoksiin kyetään vastaamaan paremmin ja joustavammin. (Abrahamsson ym. 2002, 29.)

4.2.3 Viimeistely

Viimeistelyvaiheessa tapahtuu julkaisun sulkeminen. Vaiheeseen siirrytään kun on päästy yhteisymmärrykseen siitä, että pyrähdyksen muuttujat, kuten vaatimukset ovat valmiit. Tässä vaiheessa toteutukseen liittyviä uusia toiminnallisuuksia tai toteutuksia ei tulisi nousta enää esille. (Abrahamsson ym. 2002, 30.)

Kehitetty järjestelmä tai ohjelmisto on valmis julkaistavaksi, ja viimeistelyvaiheessa suoritetaankin julkaisemisen vaatimat valmistelutyöt, kuten integraatio, järjestelmätestaus ja dokumentointi. (Abrahamsson ym. 2002, 30; Schwaber 1995.)

4.3 Roolit

Scrum määrittää kolme erilaista roolia, joita ovat tuotteen omistaja, Scrum-mestari ja tiimi. Jokaisella roolilla on erilaisia tehtäviä ja tarkoituksia prosessissa ja sen käytännöissä. Cohn (2009, 134) esittää, että vaikka Scrumin tuotteen omistajan ja Scrum-mestarin roolit ovat uusia, niiden sisältämät vastuut eivät. Lisäksi korkeatasoiset ja hyvää tulosta tekevät tiimit ovat aina tienneet tehokkaim-

mat toimintatapansa ja ymmärtäneet itseohjautuvuuden merkityksen (Cohn 2009, 134). Aliluvuissa 4.3.1–4.3.3 perehdytään Scrumin kolmeen viralliseen rooliin, ja aliluvussa 4.3.4 esitellään asiakas-rooli, joka ei kuulu Scrumin määrittelemiin rooleihin, mutta se on monesti olennainen osa projekteja.

4.3.1 Tuotteen omistaja

Tuotteen omistaja on vastuussa tuotteen työlistan hallinnasta ja ylläpidosta. Valinnasta projektin tuotteen omistajaksi päättävät Scrum-mestari, asiakas ja johto yhdessä. Tuotteen omistaja hallinnoi projektia ja vastaa siitä, että tiimit toteuttavat liiketoiminnan kannalta tärkeimpiä asioita. (Schwaber & Beedle 2002, 34–35.)

Sopivaa tuotteen omistajaa etsittäessä henkilöistä kannattaa kenties valita se, joka on päätöksentekotaitoinen ja jonka päätöksiin luotetaan. Tiimi on riippuvainen tuotteen omistajasta, joten kyseiseen rooliin valittavalla henkilöllä on oltava aikaa olla läsnä tiiminsä kanssa, tai ainakin hyvin tavoitettavissa. (Cohn 2009, 130–131.)

Yhtenä järkevänä vaihtoehtona kehitystiimin ja asiakkaan välisen viestinnän hoitamiseen on sopia viestinnän olevan tuotteen omistajan vastuulla. Crispinin ja Gregoryn (2009, 141) mielestä useiden ihmisten erilaisten näkökulmien eteenpäin vieminen yhden henkilön kautta sisältää tiedon osittaisen häviämisen tai muuttumisen riskin.. Crispin ja Gregory (2009, 373) pitävät kuitenkin tuotteen omistaja -roolia kokonaisuutena tarkasteltuna hyvänä asiana asiakkaan kannalta.

4.3.2 Scrum-mestari

Scrum-mestari on vastuussa siitä, että projekti etenee suunniteltujen toimenpiteiden ja aikataulujen mukaisesti. Projektissa on toimittava Scrum-mestarin määrittelemien käytäntöjen, arvojen ja sääntöjen mukaisesti. Scrum-mestari on

vuorovaikutuksessa läpi projektin niin Scrum-tiimin, asiakkaan kuin johdonkin kanssa. Mikäli Scrum-tiimin työskentelyä haittaavia ja tuottavuutta heikentäviä esteitä syntyy, on niistä eroon pääsemisestä vastuussa Scrum-mestari. (Schwaber & Beedle 2002, 31–32.) Scrum-mestari voi edesauttaa projektin tehokasta etenemistä myös huolehtimalla, että prosessia ja käytäntöjä kehitetään projektin vaatimusten mukaisesti.

Scrum-mestarin roolia pidetään erittäin vaativana, eikä siihen täten sovi kuka tahansa henkilö. Cohnin (2009, 119–120) mukaan tärkeimpiä Scrum-mestareilta löytyviä luonteenpiirteitä ja ominaisuuksia ovat muun muassa vastuullisuus ja uhrautuvaisuus. Lisäksi Scrum-mestarin on oltava yhteistyökykyinen ja sitoutunut työtehtäviinsä. (Cohn 2009, 119–120.)

4.3.3 Scrum-tiimi

Scrum-tiimi on itseohjautuva projektitiimi, jolla on valta organisoida itsensä ja päättää tarvittavista toimenpiteistä saavuttaakseen jokaisen pyrähdyksen tavoitteet. Scrum-tiimi osallistuu varsinaisen toteutustyön lisäksi moniin Scrumin käytäntöihin. Näitä ovat muun muassa toistuva arviointi (*effort estimation*), pyrähdyksen työlistan luonti ja tuotteen työlistan katselmoinnit. Lisäksi tiimin työskentelyä haittaavista esteistä ilmoittaminen on luonnollisesti tiimin itsensä vastuulla ja sen parhaaksi. (Abrahamsson ym. 2002, 31.)

Cohnin (2009, 204–205) mukaan on yleinen harhaluulo, että kaikkien Scrum-tiimin jäsenten tulisi olla keskenään tasavertaisia muun muassa teknologiaosaamisen suhteen sen sijaan, että kukin tiimin jäsen olisi jonkin tietyn asian tai alan erikoisosaaja. Vaikka Scrum-tiimin jäsenten olisi hyvä osata tehdä monia asioita, on heistä jokaisella kuitenkin käytännössä jokin ensisijainen työtehtävä, rooli tai osa-alue projektissa. Mikäli tiimi koostuu pelkästään henkilöistä, joilla on kohtalainen perustietämys useimmista asioista, ei pyrähdyksen yksittäisiä haasteita saada todennäköisesti ratkaistua riittävän nopeasti. (Cohn 2009, 205.)

4.3.4 Asiakas

Asiakas osallistuu tuotteen työlistaan liittyviin tehtäviin, kuten pyrähdyksissä toteutettavien toiminnallisuuksien valintaprosessiin ja priorisointiin (Abrahamsson ym. 2002, 31). Tuotteen omistajaksi projektiin voidaan nimetä myös henkilö asiakkaan puolelta. Käytännössä tämä ei ole aina mahdollista, minkä takia tuotteen omistajaksi valitaankin tällöin sopiva henkilö toimittajaorganisaatiosta.

Monesti projekteissa asiakas haluaa suorittaa tilaamalleen järjestelmälle tai tuotteelle hyväksymistestauksen. Hyväksymistestauksen tarkoituksena on varmistua tuotetun ohjelmiston vaatimusten mukaisesta ja virheettömästä toiminnasta. Hyväksymistestauksen riittävä onnistuminen on monesti ehtona sille, että asiakas suostuu maksamaan tehdystä kehitystyöstä. Asiakkaan niin halutessa voidaan hyväksymistestausta tehdä ketterissä menetelmissä jonkin verran jo projektin aktiivisen kehitysvaiheen aikana, sillä – edellä mainituin tavoin – toimivia toiminnallisuuksia valmistuu koko ajan.

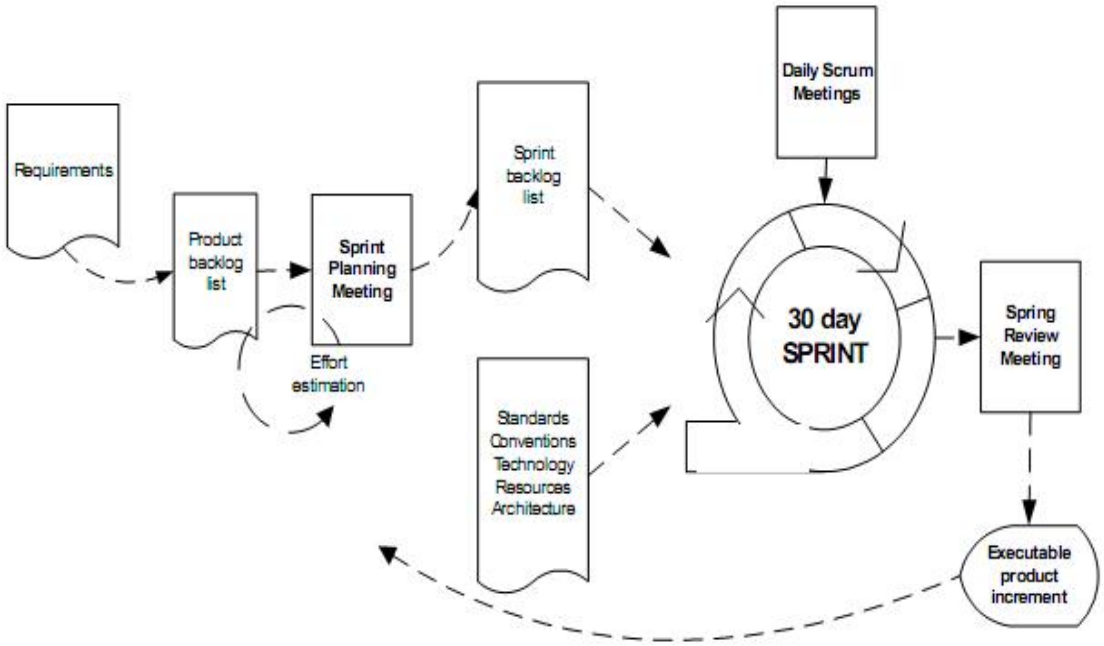
4.4 Käytännöt

Scrum ei vaadi tai edes tarjoa käytettäväksi mitään tarkkoja ohjelmistokehityskäytäntöjä. Sen sijaan se vaatii tiettyjä hallinnoimiskäytäntöjä ja työkaluja eri vaiheissaan. Näiden käytön tarkoituksena on pyrkiä välttämään ennustamattomuuden ja kompleksisuuden tuomaa kaaosta projektiin. (Schwaber 1995.) Seuraavaksi esitellään muutamia Scrumin käytännöistä yleisimpiä.

Tuotteen työlista. Tuotteen työlista on muuttuva, priorisoitu jono ja varasto liiketoiminnallisia ja teknisiä toiminnallisuuksia, jotka halutaan toteuttaa projektissa tuotettavaan ohjelmistoon. Kaikki työ, joka vaaditaan tehdyksi kehitettävää ohjelmistoa varten, sisällytetään tuotteen työlistaan ja poimitaan sieltä ajallaan pyrähdykseen toteutettavaksi. (Schwaber & Beedle 2002, 32.)

Toistuva arviointi. Toistuva arviointi on iteratiivinen prosessi, jossa fokusoidutaan syvällisemmin työlistan sisällön arvioihin. Toistuvaa arviointia tehdään silloin, kun tuotteen työlistan sisältämistä asioista on tarjolla tarkempaa tietoa. Toimenpiteen suorittavat tuotteen omistaja ja Scrum-tiimi yhdessä. (Abrahamsson ym. 2002, 32.)

Pyrähdys. Pyrähdyksellä tarkoitetaan yhtä kehitysjaksoa, jonka myötä syntyvä tuotos on ainakin periaatteessa julkaisuvalmis eli asennettavissa tuotantoympäristöön. Tyypillisesti pyrähdyksen kesto on neljä viikkoa eli noin kolmekymmentä päivää, mutta pituus voi vaihdella viikosta kahteen kuukauteen. Pyrähdyksen käytännöt, niiden keskinäiset riippuvuudet ja syötteet on esitetty kuviossa 8. (Schwaber 2004, 1–14.)



KUVIO 8. Scrumin pyrähdys käytäntöineen ja syötteineen (Abrahamsson ym. 2002, 33).

Pyrähdyksen työlista. Pyrähdyksen työlista (*sprint backlog*) on lähtökohta jokaiselle pyrähdykselle. Se kuvaa pyrähdyksen aikana toteutettavat tehtävät tarkalla tasolla. Pyrähdyksen työlista sisältää ne asiat, jotka pyrähdyksessä halutaan toteuttaa ja jotka edellytetään toteuttamisen mahdollistamiseksi. (Schwaber &

Beedle 2002, 71.) Työlistan sisällön valinta suoritetaan priorisoitujen asioiden ja pyrähdykseen asetettujen tavoitteiden mukaan pyrähdyksen suunnittelupalaverissa, johon osallistuvat Scrum-tiimi, Scrum-mestari sekä tuotteen omistaja. Toisin kuin tuotteen työlista, pyrähdyksen työlista on vakaa pyrähdyksen alkamisesta sen päättymiseen asti. (Abrahamsson ym. 2002, 33.)

Etenemisvauhti. Pyrähdyksen aikana valmiiksi saatujen käyttäjätarinoiden tai toiminnallisuuksien lukumäärä on projektin etenemisvauhti (*velocity)*. Kun projektia suunnitellaan, voidaan apuna käyttää odotettavissa olevaa, keskimääräistä etenemisvauhtia, joka ennustetaan esimerkiksi projektin aiemmista pyrähdyksistä tai aiemman, samantyyppisen, projektin perusteella. Etenemisvauhti voidaan mitata ja ilmoittaa pisteinä, päivinä tai tunteina. (Crispin & Gregory 2009, 499.)

Etenemisvauhdin laskemisessa mukaan kannattaa ottaa vain tiimin täysin valmiiksi saamat käyttäjätarinat ja toiminnallisuudet. Melkein valmiiksi tulleet jätetään pois, koska niiden tarkkaa valmiusastetta on vaikeaa määrittää ja keskeneräiset toiminnallisuudet eivät yleensä tarjoa loppukäyttäjille tai asiakkaille mitään konkreettista hyötyä.

4.5 Tapaamiset

Scrum sisältää useita erityyppisiä projektiryhmän tapaamisia, joista jokaisella on oma tarkoituksensa. Nämä tapaamiset esitellään seuraavaksi.

Pyrähdyksen suunnittelupalaveri. Pyrähdyksen suunnittelupalaveri (*sprint planning meeting*) on kaksivaiheinen palaveri, jonka organisoi Scrum-mestari. Ensimmäiseen vaiheeseen, jossa päätetään seuraavan pyrähdyksen tavoitteet ja toiminnallisuudet, osallistuvat asiakas, loppukäyttäjät, johto, tuotteen omistaja ja Srum-tiimi. Palaverin toinen vaihe pidetään Scrum-mestarin ja Scrum-tiimin toimesta. Tällöin keskitytään tarkemmin kehitettäväksi valittuihin toiminnalli-

suuksiin, eli siihen, kuinka asiat tullaan käytännössä toteuttamaan. (Abrahamsson ym. 2002, 33.)

Päivittäinen tilannekatsaus. Päivittäinen tilannekatsaus (*daily Scrum meeting*) pidetään joka päivä. Tapahtumaan osallistuu Scrum-tiimi ja tapahtuman keston tulisi olla ainoastaan 5–15 minuuttia. Päivittäisen tilannekatsauksen tarkoituksena on paitsi helpottaa kehityksen seurantaa, myös toimia suunnittelupalaverien tukena: mitä on tehty edellisen päivittäisen tilannekatsauksen jälkeen ja mitä tullaan tekemään ennen seuraavaa päivittäistä tilannekatsausta. Tilannekatsauksissa nostetaan lisäksi esille mahdolliset ongelmat, jotka haittaavat työntekoa tai toiminnallisuuksien valmistumista. (Schwaber & Beedle 2002, 40.)

Pyrähdyksen katselmointi. Pyrähdyksen viimeisenä päivänä pidettävässä katselmoinnissa (s*print review meeting*) Scrum-tiimi ja Scrum-mestari esittelevät pyrähdyksen tulokset johdolle, asiakkaalle, loppukäyttäjille ja tuotteen omistajille. Tilaisuus on luonteeltaan epävirallinen ja vapaamuotoinen. Osallistujat arvioivat tuotteen kasvua (*increment*) ja tekevät sen pohjalta päätöksen jatkotoimenpiteistä. Pyrähdyksen katselmoinnin myötä on mahdollista, että tuotteen työlista päivittyy joko uusilla vaatimuksilla tai olemassa oleviin toiminnallisuuksiin kohdistuvilla muutoksilla. (Abrahamsson ym. 2002, 34.) Ainoastaan ne pyrähdysten tuotokset, jotka ovat aidosti valmiita (*done*), osoittavat projektin etenemistä. Melkein valmiita ja keskeneräisiä toiminnallisuuksia ei oteta huomioon, eikä niistä täten olla kiinnostuneita.

Jälkiarviointi. Jälkiarvioinnilla (*retrospective*) tarkoitetaan pyrähdyksen jälkeen pidettävää tapaamista. Jälkiarvioinnin tarkoitus on katselmoida sitä, kuinka menneessä pyrähdyksessä toimittiin ja onnistuttiin. Scrum-tiimi analysoi ja kehittää omaa toimintaansa, jotta siitä tulisi ajan myötä mahdollisimman tarkoituksenmukainen ja tehokas. Jälkiarvioinnin tulisi kestää maksimissaan kolme tuntia ja sen vetäjänä toimii Scrum-mestari. (Schwaber 2004, 141.; Davies & Sedley 2009, 183–184.) Kniberg (2006, 57) pitää jälkiarviointeja Scrumin toiseksi tärkeimpänä tapahtumana asettaen jälkiarviointien edelle pyrähdyksen suunnitte-

lupalaverin. Molemmat tapahtumat ovat Knibergin (2006, 57) mielestä tärkeitä, koska niiden avulla on mahdollista muuttaa asioita parempaan suuntaan. Tarkkaa, yhtä ainoaa oikeaoppista kuvausta jälkiarviointien toteutukseen ei ole esitetty – tärkeintä on että jälkiarvioinneista pidetään kiinni (Kniberg 2006, 57). Crispinin ja Gregoryn (2009, 61) mukaan jälkiarvioinnit ovat oiva mahdollisuus arvioida, mitä testaajien on tarkoitus tarjota tiimille ja mitä taitoja on vielä kehitettävä: testaajat saattavat esimerkiksi tarvita tulevissa pyrähdyksissä kehittäjien tukea tietyntyyppisessä testauksessa.

5 KETTERÄN MENETELMÄN PILOTOINTI OHJELMISTO-ORGANISAATIOSSA

Tässä luvussa käsitellään organisaatiossa tapahtuvaa ketterän ohjelmistokehitysmenetelmän pilotointia ja käyttöönottoa. Luvussa luodaan katsaus kirjallisuudesta ja artikkeleista löytyvien ketterien menetelmien pilottiprojektien raportteihin ja organisaatioiden kokemuksiin. Lisäksi esitellään syitä ja tekijöitä onnistumisiin ja epäonnistumisiin, joita ketteriä menetelmiä käyttöönottavat ohjelmistoalan organisaatiot ovat kohdanneet.

5.1 Yleistä

Monet organisaatiot ovat kiinnostuneita ketterien menetelmien käyttöönotosta saadakseen etua lukuisista ketterien menetelmien tarjoamista eduista. Etuina pidetään muun muassa nopeampaa sijoitetun pääoman tuottoa, parempaa ohjelmiston laatua ja mahdollisuutta korkeampaan asiakastyytyväisyyteen. Käyttöönottoon ei kuitenkaan ole olemassa selkeää prosessia auttamaan organisaatioita läpi muutosprosessin. (Sidky, Arthur & Bohner 2007.)

Kirjallisuudesta löytyvät ketterien menetelmien käyttöönoton ja pilotoinnin kokemuksista kertovat organisaatiot ovat luonteeltaan erilaisia. Mukaan mahtuu niin pieniä kuin suuriakin organisaatioita ja tuoteliiketoimintaa harjoittavista organisaatioista räätälöityjen järjestelmien toteuttamisiin keskittyviä organisaatiota. Tutkielmassa ei rajata raportteja kokemuksista, muutoksen aiheuttamista ongelmista ja onnistumistarinoista pelkästään toimeksiantajaorganisaation kaltaisiin, projektiliiketoimintaa harjoittaviin organisaatioihin. Syynä tähän on tarkoitus kerätä ja analysoida yleisen tason kokemuksia ja esille nousseita sudenkuoppia, joista on hyvä tietää organisaation rakenteesta tai tarkasta toimialasta riippumatta.

Se, mitä käyttöönoton onnistumisella tarkoitetaan, ei välttämättä ole yksiselitteistä. Elssamadisy (2008, 14) kritisoi organisaatioiden ketterän menetelmän käyttöönoton onnistumista kuvaavia mittareita ja määrityksiä. Mikäli onnistunut käyttöönotto tarkoittaa sitä, että organisaatio harjoittaa yhtä tai useampaa ketterän menetelmän tunnuspiirteistä käytäntöä, on seurauksena käyttöönoton onnistuminen suurimmaksi osaksi. Edellä esitetty määritelmä ei ole kuitenkaan kovin tyydyttävä tai hyödyllinen. Organisaation tulisi myös katsoa taaksepäin, alkuperäisiin tavoitteisiin, jotka haluttiin toteuttaa uuden ohjelmistokehitysmenetelmän käyttöönoton myötä. (Elssamadisy 2008, 14.)

Paremmiksi vaihtoehdoiksi määrittämään ketterän menetelmän käyttöönoton onnistumista Elssamadisy (2008, 14) esittää ohjelmistovirheiden määrän laskemisen, ajan joka kuluu kun tuote tai ohjelmiston versio saadaan markkinoille pienenemisen, tuottavuuden paranemisen ja organisaation lisäarvon saamisen tuottavuuden ja paremman ohjelmiston kehittämisen kautta. (Elssamadisy 2008, 14.)

5.2 Muutosvastarinta

Muutos suunnitelmakeskeisestä kulttuurista ketteriin kehitystapoihin ei ole helppo. Muutoksen toteuttaminen vaatii muutoksia moniin vakiintuneisiin käytäntöihin ja se voi ulottua jopa sidosryhmien perusarvoihin. Vaikutusalueen piiriin kuuluvat niin vaatimukset, muutoksenhallinta, loppukäyttäjän mukanaolo, osapuolien halu ottaa vastuuta, sopimusten hallinta kuin myös kyvykkyys elää epävarmuuksien kanssa. (Hirsch 2005.)

Motivaatio muutokseen löytyy usein ihmisistä, jotka oikeasti haluavat muutosta. Hyvät ohjelmistokehittäjät ovat luonteeltaan hyviä oppimaan ja ratkomaan ongelmia. Ohjelmistoalalla uusia teknologioita tulee jatkuvasti markkinoille, ja näiden oppiminen voi olla aikaa vievää. Lisäksi ohjelmiston toteuttaminen on prosessi, jossa ratkaistaan ongelmia jatkuvasti. Täten voisi helposti kuvitella,

että ohjelmistokehityksessä mukana olevat ihmiset ovat tottuneet muutoksiin myös varsinaisen koneella tehtävän ohjelmistokehitystyön ulkopuolella ja että ollaan halukkaita kokeilemaan uusia ideoita. Näin ei kuitenkaan ole, vaan muutosvastarintaa esiintyy runsaasti. (Kelly 2008, 157–158.)

Ihmiset sitoutuvat muutoksiin omien, ei toisten syiden takia. Kun ihmisiä kysytään sitoutumaan muutokseen, heidän ensimmäinen huolenaiheensa on se, mitä saavutettavaa tai menetettävää on luvassa. Onnistunut ohjelmistoprosessin parannus on enimmäkseen ihmistenhallintaharjoitus eikä niinkään insinööritaitojen hallintaharjoitus. Suurin osa ihmisistä tekee asioita omalla tavallaan, sillä valittu tapa koetaan miellyttävimmäksi. Ei siis pitäisi olla yllätys, että ohjelmistoalalla ihmisten saaminen tekemään asioita jonkun toisen kuin heidän itsensä tavalla voi olla haaste täynnä yllätyksiä. (Craig & Jaskiel 2002, 388.)

Pelko on ihmisen voimakkaimpia tuntemuksia, ja se voi pahimmillaan vaarantaa siirtymän ketteriin menetelmiin. Jos yksilöt kokevat uuden ketterän menetelmän uhkaavan heidän työtään ja työtapojaan, he tulevat vastustamaan muutosta. (Crispin & Gregory 2009, 38.) Ketteriä menetelmiä sanotaan yleisesti ottaen menetelmiksi, jotka ottavat ihmiset ja ihmisarvot huomioon. Tämän perusteella esimerkiksi muutoksen suunnitelmakeskeisistä ohjelmistokehitysmenetelmistä ketteriin menetelmiin voisi helposti luulla olevan mieluinen organisaation työntekijöille.

Muutosvastarintaa esiintyy kuitenkin niin yksilö- kuin tiimitasollakin. Davies ja Sedley (2009, 31) toteavat, että ei ole helppoa saada tiimiä tekemään muutoksia toimintatapoihinsa: Ihmisten on ymmärrettävä, mikä ajaa muutosta ja mitä hyötyjä siitä on, ennen kuin he heittäytyvät mukaan muutokseen ja kenties hyväksyvät sen ajan myötä. Hyväksi vaihtoehdoksi liikkeelle lähtöön Davies ja Sedley (2009, 31) esittävät ketterien menetelmien aatteen ajamista ja puolestapuhumista, jonka myötä saadaan nopeasti esille ihmisten vastalauseita. On ymmärrettävää ja luonnollista, että muutoksen ja uusien työtapojen tuomista riskeistä ol-

laan huolissaan, vaikka muutokseen koettaisiin olevan hyviä perusteluja ja tarvetta. (Davies & Sedley 2009, 31.)

Muutoksen kohteena olevaa tiimiä kohtaan on näytettävä luottamusta ja uskoa. Ketterien menetelmien ja muutoksen esittelijän usko on tärkeää, ja se voi antaa ihmisille rohkeutta ottaa ensi askeleet muutokseen. Lisäksi on huolehdittava siitä, ettei tiimiä pakoteta ottamaan liian suuria harppauksia liian nopeasti. Ihmisille on annettava riittävästi aikaa uusien ideoiden pohtimiseen ja käsittelyyn. Asioista ja muutoksista on myös hyvä puhua ennen kuin ne otetaan käyttöön. (Davies & Sedley 2009, 31.)

Ketterien menetelmien mukaiseen toimintaan sopeutuminen on haastavinta testaajille (Agile 2009; Crispin ja Gregory 2009, 35). Crispin ja Gregory (2009, 44) esittävät syitä testaajien muutosvastarinnalle, joka syntyy kun organisaatio tai tiimi siirtyy ketteriin menetelmiin. Näitä syitä esitellään seuraavaksi.

Identiteetin menettäminen liittyy pelkoon seurauksena erillisen ja itsenäisen laadunvarmistus- ja testaustiimin hajottamisen myötä. Lisäksi pelätään, että taidot eivät riitä ketterässä tiimissä työskentelyyn, mikä johtaa työpaikan menettämiseen. Crispin ja Gregory (2009, 45) kertovat kuulleensa laadunvarmistusesimiesten (*quality assurance managers*) kysyvän heiltä roolinsa soveltuvuutta, kun organisaatio on siirtymässä ketterän menetelmän käyttöön. Tätä pidetään selvänä merkkinä identiteetin menettämisen pelosta. (Crispin & Gregory 2009, 45.)

Uudet roolit ja työtehtävät voivat nousta keskustelunaiheeksi ketterien menetelmien käyttöönotossa. Tähän keskusteluun kannattaa käyttää aikaa ja analysoida tuotteiden ja projektien tarvitsemat henkilöt paremman onnistumisen takaamiseksi. Lisäksi on tärkeää, että kaikki tietävät ja ymmärtävät roolinsa ja vastuunsa uudessa, ketterässä tiimissä. Tämä vaatii luonnollisesti aikaa ja harjoittelua. (Crispin & Gregory 2009, 45.)

Testaajien *perehtymisen ja harjoittelun puute* uudessa ketterässä tiimissä voi aiheuttaa tilanteita, joissa testaaja ei tiedä kuinka hänen tulisi toimia yhdessä tiiminsä kanssa hyvän lopputuloksen takaamiseksi. Crispinin ja Gregoryn (2009, 46) mielestä on tärkeää, että *tiimin jäsenet sisäistävät ketterien menetelmien konseptin samalla tavalla*. Mikäli osa tiimin jäsenistä ei esimerkiksi suostu kokeilemaan testaajien esittämää jatkuvan integroinnin käyttöönottoa, on luvassa ongelmia (Crispin & Gregory 2009, 46).

Vanhat kokemukset ja asenteet esimerkiksi aiemmin epäonnistuneiden työmenetelmien muutoksesta voivat kyteä ihmisten mielissä pitkään ja vaikuttaa myöhemmin tehtäviin uudistuksiin ja kokeiluihin. Tätä voidaan estää jatkossa antamalla kaikille mahdollisuus olla mukana vaikuttamassa muutoksessa ja korostamalla tiimin itseohjautuvuutta. (Crispin & Gregory 2009, 48.)

Kulttuurierot roolien välillä ovat selviä: kehittäjät tyypillisesti haluavat tuotoksen julkaistavaksi mahdollisimman pian, asiakkaat eivät ole välttämättä tottuneet kommunikoimaan suoraan kehittäjien kanssa ja testaajat ovat kenties tottuneet astumaan mukaan projektiin kunnolla vasta projektin loppupuolella. Ketterien menetelmien käyttöönoton myötä edellä mainitut tavat sekoittuvat, ja tätä varten on selvitettävä ihmisten tarvitsema tuki ja huolehdittava siitä, että jokainen osapuoli tuntee olonsa turvalliseksi ja voi vaikuttaa lopputulokseen omalla työpanoksellaan. (Crispin & Gregory 2009, 48–49.)

Muutosten esittely. Crispin ja Gregory (2009, 49) toteavat, että mitä tahansa muutosta toteutettaessa kannattaa varautua haittavaikutuksiin ja haasteisiin. Tiimi ei ole varma uusista prosesseista ja käytännöistä, jotkin ryhmät tai yksilöt ovat lojaaleja vanhoille tavoille tehdä asioita ja jotkut taas ovat epävarmoja tai hidastavat muutoksen positiivista kehitystä. (Crispin & Gregory 2009, 49.)

5.3 Ketterän menetelmän käyttöönotto

Ketteriin menetelmiin siirtyminen ei ole helppoa ja niiden käyttöönotto ja omaksuminen ovat jatkuvan oppimisen ja kehittämisen prosessi. Muutos vaatii paljon työtä, äärimmäistä keskittymistä ja kovaa kuria. (Schatz & Abdelshafi 2005.) Onnistuminen ketterien menetelmien kulttuurin tuomisessa organisaatioon riippuu organisaation kyvystä sopeutua, mutta samalla tarvitaan myös yleisten tavoitteiden ja periaatteiden luomista organisaatioon tai sen yksikköön, jota muutos koskee (Cho 2009). Monet tapaustutkimukset, artikkelit ja kokemuksiin perustuvat raportit (ks. esim. Sutherland 2001) sisältävät menestyksekkäitä tarinoita Scrumin käyttöönotoista yrityksissä ja organisaatioissa. Kuitenkin, monien edellä mainitun tyyppisten raporttien heikkoutena, tai jopa ongelmana voitaneen pitää ainakin niiden kvantitatiivisten tulosten vähäisyyttä pelkän havainnoinnin tueksi.

Mikäli organisaatiossa on tiimi, joka on siirtynyt ketterien menetelmien käyttöön aiemmin, on tästä luonnollisesti paljon apua ja etua toiselle tiimille, joka myös suunnittelee ketterien menetelmien käyttöönottoa. Liiallinen toisen tiimin työtapojen seuraaminen saattaa kuitenkin olla petollista, mikäli tiimit ovat erilaisia esimerkiksi projektien luonteen, asiakkaiden ja teknologioiden suhteen.

Cohnin (2009, 5) mukaan muutos juuri Scrumin käyttöön sisältyy muutamia asioita, jotka tekevät käyttöönotosta haastavan. Näistä yhtenä Cohn (2009, 9) pitää organisaatiomuutosten myötä löytyviä ”parhaita käytäntöjä”. Ne ovat vaarallisia, sillä monesti ne eivät ole todellisuudessa parhaita mahdollisia tapoja tehdä jokin asia. Parhaina käytäntöinä niitä pidetään pelkästään siksi, koska joku henkilö on todennut ne toimiviksi, ja koska parempaakaan ei ole vielä ehditty miettiä tai kokeilla. On vaarallista, mikäli nämä käytännöt otetaan organisaatiossa laajasti käyttöön ilman, että niiden mahdollisesti sisältämiä kompastuskiviä selvitetään. (Cohn 2009, 9.)

Sidky ym. (2007) esittävät yhdeksi vaihtoehdoksi hallittuun ketteriin menetelmiin siirtymiseen kehittelemänsä viitekehyksen, jonka tarkoitus on sekä tuottaa rakenteellinen ja toistettavissa oleva lähestymistapa että opastaa ja avustaa organisaatiota ketterän menetelmän käyttöönottoon. Viitekehys muodostuu ketteryyden mittaamisindeksistä ja nelitasoisesta prosessista (Sidky ym. (2007).

Mittaamisindeksi, josta Sidky ym. (2007) käyttävät nimeä Sidky Agile Measurement Index (SAMI), käsittää viisi ketteryyden tasoa, joita käytetään tunnistamaan ketteryyden potentiaalisuus projekteissa ja organisaatioissa. Toinen viitekehyksen osa, nelitasoinen prosessi puolestaan auttaa päättelemään missä määrin organisaatiot ovat valmiita ketterän menetelmän käyttöönottoon. Lisäksi prosessin avulla saadaan esille ne ketterien menetelmien osakokonaisuudet, joiden käyttöönottoa kannattaisi harkita ensin ja jotka parhaiten sopivat organisaation kontekstiin. (Sidky ym. 2007.)

Sidky ym. (2007) toteavat viitekehyksen olevan menetelmäriippumaton, joten se ei aseta rajoitteita XP:n, Scrumin tai minkään muunkaan ketterän menetelmän käyttöön. Sidky ym. (2007) pitävät kehittämäänsä menetelmää pioneerin asemassa olevana viitekehyksenä, jonka avulla pystytään vastaamaan ketterien menetelmien käyttöönoton ongelmiin.

On myös esitetty näkemyksiä, joiden perusteella ei ole välttämätöntä luopua vanhoista, esimerkiksi suunnitelmakeskeisten menetelmien toimivista ja käytössä olleista käytännöistä. Boehmin ja Turnerin (2003b) mielestä niin suunnitelmakeskeisillä kuin myös ketterillä menetelmillä on tilanteesta riippuen erinäisiä puutteita, jotka voivat johtaa projektin epäonnistumiseen, mikäli puutteisiin ei puututa asianmukaisella tavalla. Boehm ja Turner (2003b) esittelevät projektien organisointiin riskeihin perustuvan lähestymistavan sekoittamalla sekä ketterien että suunnitelmakeskeisten menetelmien piirteitä ja käytäntöjä projekteihin tapauskohtaisesti.

Maples (2009) esittää onnistuneen Scrumiin siirtymisen edellyttävän uskallusta poiketa säännöistä, sillä jokainen organisaatio on erilainen. Ketterien menetelmien pääperiaatteet kannattaa kylläkin pitää mielessä, ja antaa jonkin ketterien menetelmien asiantuntijan olla mukana varmistamassa, etteivät periaatteet jää mukautumisen myötä liian taka-alalle. Maples (2009) jatkaa, että ketteristä menetelmistä saatavat hyödyt ovat ilmeisiä, mutta on olemassa kriittisiä tekijöitä jotka voivat vaikuttaa onnistumisen todennäköisyyteen. Tärkeimmäksi näistä Maples (2009) pitää kulttuuria, jonka täytyy edesauttaa luottamuksen kasvamista ja mahdollistaa tunnistamaan tekijöitä, jotka voisivat parantaa kokonaisuutta entisestään.

Cohnin ja Fordin (2003) mukaan uuden, ketterän menetelmän menestykseen vaikuttaa se, kuinka menetelmä esitellään organisaatiossa. Mikä tahansa uusi prosessi todennäköisesti kiehtoo osaa kehittäjistä, jotka haluavat siten olla ensimmäisenä kokeilemassa tätä uutta asiaa. Toisaalta samalla ne kehittäjät, jotka vastustavat muutosta, pitävät tätä uutta asiaa pelkästään hidastavana tekijänä ja kompastuskivenä. (Cohn & Ford 2003.)

Ohjelmistokehityksen ketteryys ymmärretään monesti virheellisesti ketterinä käytäntöinä arvojen asemesta. Tämän tuloksena ketterien menetelmien käyttöönotossa keskitytäänkin väärään asiaan, konkreettiseen käytäntöön kuten Scrumin joka-aamuiseen palaveriin. Käytännön toimien ollessa tilanneriippuvaisia joudutaan helposti ongelmiin. Ketterän menetelmän käyttöönoton tulisi siis olla tiettyjen käytäntöjen sijaan arvojen ja periaatteiden muutosprosessi. Organisaation johdon ehdotus tai käsky ottaa ketterä menetelmä käyttöön on huono peruste ketterien menetelmien käyttöönottoon. Muutosidean ja -halukkuuden tulisi lähteä nimenomaan itseohjautuvasta tiimistä mahdollistaen tiimin itsensä päättää, mitä muutetaan ja miten. (Larmanin esipuhe teoksessa Elssamadisy 2007, xi.)

Milunsky (2009) esittää artikkelissaan näkemyksiä projektin eri osapuolien kohtaamista muutoksista, kun organisaatiossa siirrytään käyttämään ketteriä mene-

telmiä ohjelmistokehityksessä. Alla esitellään muutamia tärkeimmistä osapuolista Milunskyn (2009) mukaan.

Asiakas. Vesiputousmallia käytettäessä asiakas pidetään helposti tietyllä etäisyydellä lukuun ottamatta projektin aloitusta ja lopetusta. Lisäksi asiakkaalle tehdään selväksi, että kaikki projektin alkuvaiheen jälkeen tulevat muutospyynnöt kehitettävään ohjelmistoon tietää muutoksenhallintaprosessia ja lisäkustannuksia. Ketterissä menetelmissä asiakas on läheisessä yhteistyössä ja kontaktissa läpi projektin, ja muutoksia tiedetään tulevan ja ne hyväksytään. (Milunsky 2009.)

Kehittäjät. Mikäli käyttöön otetaan XP, on pariohjelmoinnin opettelussa monelle kehittäjälle luvassa Milunskyn (2009) mukaan haasteita. Milunsky (2009) täsmentää pariohjelmoinnin olevan monelle kehittäjälle kova paikka henkisesti, sillä kehittäjän tuottama koodi on katselmoitavana koko ajan, ja pariohjelmoinnin luonteeseen kuuluu jatkuva asioiden tarkastelu ja jopa kritisointi. Milunskyn (2009) mielestä tämä on kuitenkin pelkästään positiivinen asia, sillä hän näkee pariohjelmoinnin parantavan kehittäjän taitoja ja nostavan siten ohjelmiston laatua.

Toisena kehittäjien kohtaamana haasteena Milunsky (2009) pitää ketterien menetelmien edellytystä, jossa laatu rakennetaan tuotteeseen heti alusta lähtien. Tämän mahdollistavat yksikkötestit ja TDD, jotka molemmat voivat myös olla kehittäjälle vaikeita asioita hyväksyä ja omaksua pariohjelmoinnin ohella. Positiivisina muutoksina kehittäjien kannalta Milunsky (2009) puolestaan näkee ketterien menetelmien tiimityön ja yhteenkuuluvuuden tunteen: Joko koko projektitiimi onnistuu yhdessä, tai sitten kaikki epäonnistuvat. Tiimin jäsenet pyrkivät kaikin keinoin auttamaan toisiaan, kun he tarvitsevat apua (Milunsky 2009).

Testaajat. Testaajat tekevät ketteriä menetelmiä soveltavissa projekteissa paljon muutakin kuin varsinaista testaustyötä. Milunsky (2009) pitää hyvänä asiana testaajien osallistumista projektiin aktiivisesti heti ensimmäisestä pyrähdykses-

tä lähtien. Tämän myötä testaajien on mahdollista vaikuttaa projektin onnistumiseen tarkentamalla esimerkiksi käyttäjätarinoiden yksityiskohtia ja hyväksymistestauksen läpäisykriteerejä. Nämä auttavat kehittäjiä ymmärtämään mitä on tehtävä, että testit saadaan hyväksytysti läpi. Tuloksena on koodin laadun ja tarkkuuden paraneminen, joista jälkimmäisellä tarkoitetaan sitä, kuinka hyvin toteutus vastaa käyttäjävaatimuksia. (Milunsky 2009.)

Testaajien vastuulla on Milunskyn (2009) mielestä automatisoida mahdollisimman suuri osuus toimintotestauksesta. Kattavasti automatisoitu toimintotestaus myötävaikuttaa tiimin kokonaistehokkuuteen ja tuottavuuteen. Lisäksi ketterien menetelmien ajattelutapa pyrkii kaikkien tiimin jäsenten tasavertaistamiseen. Tämä ilmenee muun muassa siten, että Scrumissa puhutaan vain kehittäjistä. Tästä voidaan johtaa testaajan olevan kehittäjä, joka tekee muita asioita kuin tuottaa ohjelmistokoodia. (Milunsky 2009.)

5.4 Käyttöönoton sudenkuopat

Keith ja Cohn (2008) esittelevät asioita, joita tekemällä ketterien menetelmien käyttöönotossa voidaan epäonnistua täysin. Keith ja Cohn (2008) ovat jakaneet nämä riskejä sisältävät, käyttöönoton epäonnistumiseen johtavat toimenpiteet neljään kategoriaan, joita ovat johtoon, tiimiin, tuotteen omistajaan ja prosessiin liittyvät seikat. Seuraavaksi esitellään Keithiä ja Cohnia (2008) mukaillen niitä tiimiin ja prosessiin liittyviä asioita, joita tulisi välttää. Johtoon ja tuotteen omistajaan liittyvät asiat on jätetty pois tässä yhteydessä.

Sudenkuopat, jota tiimin tulisi välttää. Keithin ja Cohnin (2008) mukaan jatkuva epäonnistuminen pyrähdyksessä tehtävän työmäärän ja toiminnallisuuksien toimittamisessa johtaa ennen pitkään ongelmiin: kehitettäviä toiminnallisuuksia jää joka pyrähdyksessä yli, joten ne kasaantuvat pikku hiljaa. Riskialttiiksi mainitaan myös tiimien muodostaminen ilman, että tiimiin kuuluu sekä kehittäjiä, testaajia että arkkitehtejä. Ei kannata unohtaa myöskään tiimien koon merkitys-

tä: tuottavuus on vaarassa laskea kommunikaation haasteiden myötä, jotka syntyvät suurissa tiimeissä. (Keith & Cohn 2008.)

Prosessiin liittyvät kompastuskivet. Keithin ja Cohnin (2008) mielestä ketterän menetelmän prosessi tulisi ottaa käyttöön niin kuin kirjoissa sanotaan, eikä siten, että prosessia alettaisiin muokata heti aluksi. Keith ja Cohn (2008) jatkavat, että tärkeitä ketterien menetelmien käytäntöjä ei tulisi muokata tai jättää kokonaan pois ennen kuin ne ymmärretään täysin. Ketterien menetelmien käyttöönotossa ei myöskään tulisi keskittyä yleisesti käytettyihin käytäntöihin ilman, että niiden perustana olevat periaatteet ja arvot ymmärretään (Keith & Cohn 2008). Näiden lisäksi Keith ja Cohn (2008) painottavat myös jatkuvan prosessin parantamisen tärkeyttä, teknisten käytäntöjen muuttamisen muistamista sekä työtehtävien priorisointia, jonka toimiessa pystytään toteuttamaan aina tärkeimmät asiat ensin.

Samansuuntaisiin tuloksiin siitä, mitkä asiat voivat helposti viedä ketterien menetelmien käyttöönoton sivuraiteille on tullut myös Larman (2004, 127–128). Seuraavaksi kuvataan kuusi yleisimmistä Scrumin käyttöön liittyvistä virheistä ratkaisuineen Larmanin (2004, 127–128) mukaan:

- Tiimi ei ole itseohjautuva: esimiehet tai Scrum-mestari ohjailee tai organisoi tiimiä.

Esimiehen tai Scrum-mestarin kokema houkutus kertoa tai ehdottaa tiimin jäsenille miten työskennellä tai ratkoa ongelmia voi kasvaa suureksi pyrähdyksen aikana. Monilla esimiehillä on luontaisesti tapana ohjata ja suunnitella tiimin varsinaista toteutustyötä, mikä on Scrumin vastaista. Asioita, joihin esimerkiksi Scrum-mestarin voidaan katsoa puuttuvan, ovat muun muassa tiimin työtä haittaavia esteiden poistaminen nopeasti ja työrauhan takaaminen tiimille.

- Uuden työn lisääminen pyrähdykseen tai Scrum-tiimin jäsenelle.

Scrumissa pyrähdyksessä olevan työn tekemiseen halutaan tarjota työrauha. Asioiden sekoittamisen välttämiseksi halutaan olla muuttamatta pyrähdyksen vaatimuksia sen ollessa käynnissä.

- Tuotteen omistaja ei ole jämäkkä tai ei päätä asioista.

Scrum on asiakaslähtöinen: Tuotteen omistajan on tehtävä päätöksiä tuotteen työlistan prioriteeteista ja valittava toteutettavat toiminnallisuudet seuraavaan pyrähdykseen.

- Pyrähdyksen katselmointipalaveria ei pidetä.

Palautteen antaminen ja jatkuva mukautuminen ohjailevat Scrum-tiimiä. Asiakkaan informoimiseksi tarvitaan demo ja katselmointi, jotta he voisivat vaikuttaa paremmin seuraaviin pyrähdyksiin.

- Liikaa hallinnoijia.

Scrumissa tuotteen työlistan vaatimuksiin, prioriteetteihin ja seuraavan pyrähdyksen työmäärän organisointiin ottaa kantaa vain yksi henkilö: tuotteen omistaja.

- Dokumentointi on huonolla tasolla.

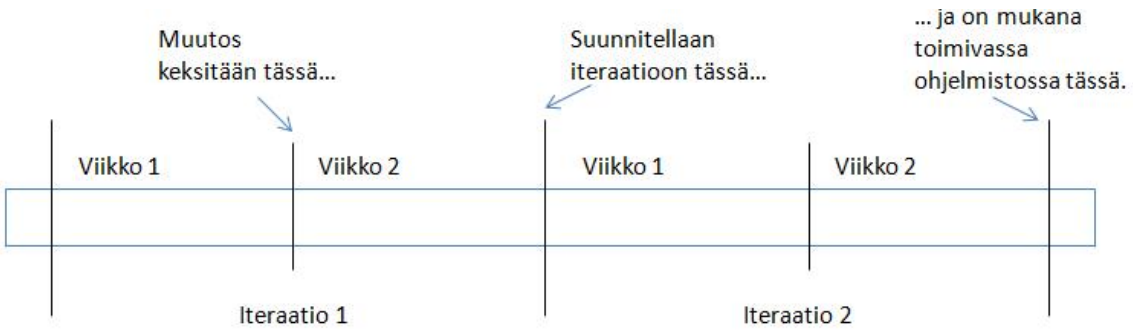
Scrum ei ole dokumentoinnin vastainen ohjelmistokehitystapa: Keskustelu projektin dokumentaation määrästä on vain jätetty pois Scrumin määrittelystä. Dokumentaatiota tuotetaan, mikäli sille katsotaan olevan aitoa tarvetta.

- Scrum-palaveri on liian pitkä tai siinä ei keskitytä olennaisiin asioihin.

Palaveri pidetään alle 20 minuutin mittaisena, ja siinä keskustellaan vain Scrumin määrittämistä kysymyksistä. (Larman 2004, 127–128.)

Cohn (2006a) pitää pyrähdyksen sopivan pituuden valintaa yhtenä tärkeimmistä asioista iteratiivista ohjelmistokehitysprosessia käyttöönotettaessa. Yleisimmät suositukset vaihtelevat viikosta neljään viikkoon, mutta paras pyrähdyksen pituus on yksilöllistä tiimien ja projektien välillä. Pyrähdyksen pituus vaikuttaa moniin asioihin kuten siihen, kuinka usein kehitettävän järjestelmän ominaisuuksia voidaan esitellä käyttäjille ja asiakkaalle, kuinka usein edistymistä voidaan mitata tarkemmalla tasolla ja myös siihen, kuinka usein voidaan säätää projektin tavoitteita ja muuttaa vaatimuksia. Mitä enemmän projektissa on paljon epävarmuutta aiheuttavia tekijöitä, sitä lyhyemmän pyrähdyksen valinta on järkevää. (Cohn 2006a.)

Ketterien menetelmien ideologiaan kuuluu, ettei tiimin työskentelyä häiritä kesken pyrähdyksen. Täten on selvitettävä, onko asioiden ja tehtävien priorisoinnin toistumissyklillä ja ajalla, joka kestää uusien toiminnallisuuksien keksimisestä niiden saamiseksi mukaan toimivaan ohjelmistoon merkitystä. Cohnin (2006b) mukaan, aika joka menee uuden toiminnallisuuden saamiseksi toimivaan ohjelmistoon, on keskimäärin 1½-kertainen käytettävän pyrähdyksen pituuteen nähden. Tämä on havainnollistettu alla olevassa kuviossa 9.



KUVIO 9. Uuden toiminnallisuuden tuotantoon saamiseksi kuluva aika (Cohnia 2006b mukaillen).

Viimeisimpänä pyrähdyksen pituuden valintaan vaikuttavana tekijänä Cohn (2006b) mainitsee sen ajan määrittämisen ja selvittämisen, jonka puitteissa jokainen saadaan työskentelemään yhdenmukaisella työtahdilla läpi pyrähdyksen. Cohn (2006b) käyttää tästä esimerkkinä vesiputousmallia ohjelmistokehitysmenetelmänä käyttäneeseen projektin, jossa tiimin jäsenet ottivat työnteon

ensimmäiset kuukaudet rennosti. Viimeiset projektin kuukaudet he vastaavasti työskentelivät yötä myöten, projektin myöhästymisestä johtuen tehty lisäkuukausi mukaan lukien. (Cohn 2006b.)

5.5 Pilotointiin vaikuttavia tekijöitä

Kun organisaatio ryhtyy kokeilemaan Scrumia, on panoksena enemmän kuin kyseisen, yksittäisen projektin onnistuminen. Kyseessä on mahdollisuus havainnollistaa osapuolille Scrumin tuomat lisäarvot eli laadun paraneminen, kustannusten vähentäminen ja toimivan ohjelmiston tuotantoon viemiseen tarvittavan ajan väheneminen. Toisaalta huonosti toteutetun pilotin seurauksena organisaatiossa vallitsee ilmapiiri, jonka mukaan Scrum ei toimi. (Mar & Szalvay 2008.)

Milloin ja miten pilotointiin sitten kannattaa ryhtyä? Onko pilottiprojektin koolla, asiakkaalla tai projektiin valittavilla jäsenillä merkitystä? Entä voiko käytettävä teknologia asettaa rajoituksia esimerkiksi testauksen automatisoinnin toteutukseen? Seuraavaksi käydään läpi ketterän menetelmän pilotointiin olennaisesti vaikuttavia tekijöitä ja etsitään vastauksia edellä esitettyihin kysymyksiin.

5.5.1 Miten muututaan?

Cohn (2008) sekä Crispin ja Gregory (2009, 61) ovat esittäneet näkemyksiään ketteriin menetelmiin siirtymisen vaihtoehdoista. Näitä esitellään seuraavaksi Cohnin (2008) sekä Crispinin ja Gregoryn (2009, 61) mukaan.

Muutoksen laajuus. Etuina siirryttäessä ketteriin menetelmiin vaiheittain, esimerkiksi tiimeittäin Cohn (2008) näkee tehdyistä virheistä aiheutuvien kustannusten minimoinnin. Virheen tekeminen yhdessä tiimissä tai tiimin projektissa on turvallisempaa ja halvempaa kuin saman virheen tekeminen laajemmassa mittakaavassa. Lisäksi siirtymällä ketteriin menetelmiin vaiheittain voidaan en-

simmäiseksi muuttujaksi valita sellainen tiimi tai yksittäinen projekti, jonka uskotaan olevan vahvoilla pilotissa onnistumisessa. Haittapuolina vaiheittain siirtymisessä Cohn (2008) pitää pidempää ajanjaksoa, joka kuluu siihen kunnes koko organisaatio tai tiimi käyttää ketteriä menetelmiä. Tämän lisäksi ei pidä luottaa liikaa tuloksiin ja johtopäätöksiin, jotka saadaan kun ainoastaan organisaation yksi tiimi tai projekti on kokeillut ketteriä menetelmiä. Menestys pienelle projektille, jonka henkilöt on valittu huolitellusti, ei välttämättä takaa samaa lopputulosta toisessa, esimerkiksi suuremmassa projektissa. (Cohn 2008.)

Crispin ja Gregory (2009, 61) esittävät, että vaikka koko organisaatio ottaisi käyttöön ketterän menetelmän yhdellä kertaa, jotkin tiimit onnistuvat käyttöönotossa paremmin kuin toiset. Mikäli koko organisaatio tai tiimi siirtyy ketteriin menetelmiin kerralla, on etuna muun muassa kahden päällekkäisen ohjelmistokehitysprosessin käytön välttäminen. Tämän lisäksi vastarinta on usein vähäisempää, sillä uutta menetelmää vastustavat henkilöt eivät pysty elättelemään toivoa vanhoihin menetelmiin ja käytäntöihin palaamisesta niin helposti, kuin mitä tilanteissa, joissa valtaosa henkilöistä ja projekteista käyttää tuttua ja turvallista ohjelmistokehitysmenetelmää. Kerralla ketteriin menetelmiin siirtymisen suurimpana riskinä on tehtyjen virheiden kustannusten kertautuminen läpi muutosvaiheen. Lisäksi kertamuutos aiheuttaa todennäköisesti tiimien uudelleenorganisointia, ja aiheuttaa myös muilla tavoin stressiä organisaatioon tai tiimiin. (Cohn 2008.)

Mitä muutetaan ensin? Kun organisaatio ja tiimit haluavat tulla ketteräksi, monesti tärkeimpänä pidetään käytäntöjen, kuten TDD:n, pariohjelmoinnin, jatkuvan integroinnin ja automatisoidun testauksen käyttöönottoa. Vastakohtana toimintatapana edellä mainitulle on ajatus muuttua ketteräksi uskoen, että muutos ketteriin menetelmiin käy helpoiten iteratiivisuuden omaksumisen kautta. Kun tärkeämpänä pidetään teknisiä käytäntöjä, ne kaikki kannattaa ottaa käyttöön nopeasti. Tällöin voidaan helpommin välttyä väittelyltä ja erimielisyyksiltä siitä, mitkä käytännöistä otettaisiin ensin työn alle. Tämän seurauksе-

na nopeat parannukset työkäytännöissä ovat mahdollisia. Haittapuolena pidetään liiallista keskittymistä teknisiin käytäntöihin, minkä seurauksena helposti unohdetaan käyttäjäkeskeinen ajattelu, joka olisi tärkeä osa ketteräksi tulemista. (Cohn 2008.)

Cohn (2008) pitää iteratiivisen työskentelyn valitsemista ensimmäiseksi kehityskohteeksi asiana, josta on helppoa lähteä liikkeelle. Tässä tosin piilee se vaara, että iteratiivisuuden sisäistänyt ja siihen oppinut tiimi jäsenineen voi pitää iteratiivisuutta riittävänä: Muita käytänteitä ei katsota enää tarvittavan, mikä ei pidä Cohnin (2008) mukaan paikkaansa – muutos on vasta alussa.

Hiljaa vai avoimin ovin? Kolmantena tärkeänä asiana ketteriin menetelmiin siirtymisessä Cohn (2008) pitää alkuvaiheen kokemuksista ja tuloksista puhumista. Oletetaan, että organisaatiossa päätetään toimia ketterän menetelmän käyttöönoton suhteen kaikessa hiljaisuudessa pois lukien se tiimi, jossa uusia asioita opetellaan. Tällöin saavutetaan mahdollisuus lykätä pilotoinnin tuloksista kertomista sinne asti, kunnes asiat voidaan nähdä positiivisessa valossa. Vastakohtaisesti tätä taktiikkaa käyttäessä ketteriä menetelmiä harjoitteleva tiimi ei saa organisaatiolta tukea toimintaansa, kuten esimerkiksi mahdollisiin muutoksiin, joita on tehtävä. (Cohn 2008.)

Jos ketterien menetelmien käyttö on uutta ja se on kaikkien tiedossa, pilotin jäsenet todennäköisesti haluavat tehdä parastaan ja sitoutuvat projektiin. Lisäksi asioiden peittelemättömyys antaa organisaatiosta tai tiimistä kuvan, jonka perusteella muutosta ketteriin menetelmiin ei epäröidä ja siten ainoastaan suunnitella ja kokeilla, vaan tahtotilana on myös onnistua aidosti. Toisaalta uusista kokeiluista julkisesti puhuminen ennen kuin mitään on käytännössä tapahtunut, on omiaan tuomaan esiin yleisesti muutosta vastustavat työntekijät ja heidän vastalauseensa ketteriin menetelmiin siirtymistä kohtaan. (Cohn 2008.)

5.5.2 Pilottiprojektin luonne

Pilotti on ensimmäinen kerta, kun valitun ketterän menetelmän uusia toimintatapoja ja käytäntöjä kokeillaan oikeassa projektissa. Kyseessä on eräänlainen markkinointitapahtuma uudelle prosessille. Mikäli pilottiin valitaan vääräntyyppinen projekti, voi projektin menestys olla uhattuna. Tämä puolestaan saattaa kokeillun ohjelmistokehitysmenetelmän huonoon valoon niin asiakkaiden kuin organisaation silmissä. (Smith & Sidky 2009, 107.)

Marin ja Szalvayn (2008) mukaan Scrum ei ole mikään hopealuoti, joka automaattisesti korjaisi epäkohdat kuntoon. Jos projektilla ei mene hyvin syystä tai toisesta, ei Scrumista ole todennäköisesti pelastamaan sitä. Päinvastoin, Scrum-pilotti vaatii niin paljon liikkumavaraa kuin organisaatiolla on mahdollista sitä tarjota. Vain ne projektit, joilla on vähän haittatekijöitä ja kompastuskiviä, voivat todella näyttää Scrumin mahdollisuudet parantaa yhteistyötä, tuottavuutta ja liiketoimintanäkökulmasta ajatellen kustannusten ennustamista. (Mar & Szalvay 2008.)

Mar ja Szalvay (2008) ovat tulleet siihen tulokseen, että Scrum-pilottiprojektin menestykseen vaikuttaa projektin tyyppi, eli se, onko kyseessä uusi kehitysprojekti, vai jo käynnissä oleva projekti. Mar ja Szalvay (2008) jatkavat suosittelemalla pilottiprojektiksi alkamatonta projektia, jolloin projektin henkilöt saavat aloittaa asioiden tekemisen uudella tavalla puhtaalta pöydältä. Muutettaessa käynnissä olevan projektin ohjelmistokehitystapa Scrumiin, voi sekä olemassa olevan koodin käsittely että Scrumin käytännöt yhdessä aiheuttaa kehittäjille ylimääräistä taakkaa ja sekaannusta (Mar & Szalvay 2008).

Valitusta projektista ei kuitenkaan olla yhtä mieltä. Schwaberin ja Beedlen (2002, 57–59) mielestä Scrum voidaan ottaa käyttöön niin uuteen, kuin jo käynnissä olevaan projektiin. Tyypillisenä käynnissä olevana Scrumin käyttöönottoprojektina Schwaber ja Beedle (2002, 58–59) pitävät tilannetta, jossa kehitysympäristö ja käytettävä teknologia ovat tiedossa ja olemassa, mutta projektitiimi

kamppailee muuttuvien vaatimusten ja kompleksin teknologian kanssa. Tällöin Scrumin esittely käynnistetään päivittäisillä Scrum-palavereilla Scrum-mestarin toimesta, ja ensimmäisen pyrähdyksen tavoitteeksi tulisi asettaa ”kyky demonstroida mitä tahansa käyttäjätoiminnallisuutta valitulla teknologialla”. Tällöin tiimi huomaa pääsevänsä tavoitteeseensa ja uskoo itseensä. Lisäksi asiakas uskoo tiimiin. (Schwaber & Beedle 2002, 59.)

Mikäli Scrum otetaan käyttöön uuden projektin myötä, on Schwaberin ja Beedlen (2002, 57–59) mielestä hyvä lähteä liikkeelle työskentelemällä Scrum-tiimin ja asiakkaan kanssa useiden päivien ajan ensimmäisen tuotteen työlistan aikaansaamiseksi. Tässä vaiheessa tuotteen työlista voi koostua liiketoiminnallisista toiminnallisuuksista ja teknologiaan liittyvistä vaatimuksista. Ensimmäisen pyrähdyksen tavoite voisi siten olla esimerkiksi mahdollisuus esitellä käyttäjätoiminnallisuuden tärkeä pala valitun teknologian avulla. Edellä mainittu tavoite luonnollisesti edellyttää järjestelmän viitekehyksen suunnittelua ja toteutusta: Tuotettavaan järjestelmään tarvitaan toimiva rakenne, jotta uusia ominaisuuksia voidaan myöhemmin helposti lisätä. (Schwaber & Beedle 2002, 57–58.)

Smithin ja Sidkyn (2009, 107) mukaan helpoin tapa vaikeuttaa Scrumin käyttöönottoa on kokeilla sen toimivuutta suuressa projektissa. Suuri projekti vaatii suuren ihmismäärän koulutusta, jonka seurauksena kyky kerätä kokemuksia ja säätää prosessia niiden mukaan vaikeutuu huomattavasti. Liian pieni projekti voi sekin johtaa ongelmiin: työtä ei ole riittävästi jolloin kokemuksiakaan ei saada kerättyä. (Smith & Sidky 2009, 107.)

Cohnin (2009, 82–83) mukaan ideaalisen pilottiprojektin valintaan vaikuttavat ainakin projektin kesto, koko ja tärkeys. Projektin pituuden tulisi olla keskipitkä, sillä liian lyhyet pilottiprojektit saavat organisaatiossa olevat skeptikot väittämään, että Scrum toimii vain lyhyissä pienissä projekteissa. Liian pitkäkestoinen projekti puolestaan ei mahdollista pilotoinnin onnistumista kuin vasta pitkän ajan kuluttua, projektin päätyttyä. (Cohn 2009, 83.)

Pilottiprojektin ihanteellisena kokona ja luonteena Cohn (2009, 83) pitää sellaista projektia, jonka toteutuksen voi aloittaa jopa yksi tiimi, jonka jäsenet työskentelevät fyysisesti samassa paikassa. Projektin tarvitsemien tiimien määrän ei tulisi nousta yli viiteen, jos se suinkin on mahdollista. (Cohn 2009, 83.)

Pilottiprojektiksi voi olla houkuttelevaa valita vähän riskejä sisältävä, vähäpätöinen projekti. Cohn (2009, 83) suosittelee juuri päinvastoin, eli valitsemaan tärkeän projektin perustellen tärkeän projektin saavan enemmän huomiota pilotoinnin suhteen. Lisäksi valitsemalla pilottiin tärkeä projekti, saadaan projektitiimin jäsenet sitoutumaan paremmin Scrumin käytäntöihin, jotka voivat olla haasteellisiakin. (Cohn 2009, 83.)

5.5.3 Projektin koko ja tiimin jäsenet

Organisaatio haluaa ymmärrettävästi onnistua ketterien menetelmin kokeilemisessa tai vaiheittaisessa käyttöönotossa, jolloin yksi tärkeimmistä kysymyksistä ja päätöksistä on oikeiden henkilöiden valinta pilottiprojekteihin. Agile Manifeston (2001) pääperiaatteet antavat jotakin, joskin epäsuoraa osviittaa siitä, kuinka henkilöiden valinta pilottiprojekteihin kannattaisi tehdä. Nerurin ym. (2005) mukaan ketterät menetelmät luottavat pitkälti ammattitaitoisten ja motivoituneiden ihmisten saatavuuteen. Nerur ym. (2005) täsmentävät, että on olemassa vähän todisteita siitä, että ketterät menetelmät toimisivat ilman osaavia ja keskiverto-osaamistason yläpuolella olevia työntekijöitä.

Ideaalipilottiprojektilla on yksi tuotteen omistaja (*Product owner*). Tätä tekijää pidetään yhtenä tärkeimmistä pilotin onnistumisen takeista. Pilottiprojektia tekevän tiimin tulisi olla fyysisesti lähekkäin toisiaan, jolloin kommunikaatio ja yhteistyö ovat tehokkainta. Toisekseen, tiimin koostumuksen tulisi olla sellainen, että jäsenillä on erilaista osaamista ja tietämystä. Tiimi, jolta puuttuu omistautunut johtajuus ja selvät suunnat, tuhlaa todennäköisesti aikaa ja voimiaan yrittäessään tulkita huonosti määritettyjä pyrähdyksen tavoitteita. (Mar & Szalvay 2008.)

Suurin yksittäinen avain onnistuneeseen Scrum-pilottiin lienee yksinkertaisesti siihen valittavien ihmisten käytännön kokemus. Kokeneen Scrum-valmentajan on helppo valita parhaat projektit pilotointiin, sillä hän tietää monien Scrum-projektien kokemusten myötä kuinka erilaiset tekijät, niin positiiviset kuin negatiivisetkin vaikuttavat lopputulokseen. Henkilö, jolla on hyvä tietämys ja käytännön kokemusta Scrumin käytöstä, voi havainnollistaa Scrumin periaatteita ja prosesseja käytännön esimerkkien kautta.

Seuraavaksi tarkastellaan muutamia Agile Manifeston (2001) periaatteista ja otetaan kantaa niistä johdettavissa ja löydettävissä oleviin vaatimuksiin tai suosituksiin ketteriä menetelmiä käyttävän projektin henkilöistä.

- Jatkuva tekniseen erinomaisuuteen ja hyvään suunnitteluun panostaminen parantavat ketteryyttä.

Yllä olevan periaatteen mukaan tyypillinen ketterien menetelmien käyttäjä kykenee toistuvasti valitsemaan oikeat tekniset ratkaisut, mikä edellyttää luonnollisesti jatkuvaa oppimista teknologioiden suhteen. Lisäksi tuloksena on laadukasta jälkeä suunnittelutyössä.

- Parhaat arkkitehtuurit, vaatimukset ja suunnittelumallit kumpuavat itseohjautuvista tiimeistä.

Yllä mainittu periaate antaa ymmärtää tiimin itsessään olevan poikkeuksetta kykenevä luonnostelemaan parhaat arkkitehtuurit, vaatimukset ja suunnittelumallit.

- Säännöllisin väliajoin tiimi pysähtyy miettimään, kuinka voitaisiin tulla vielä tehokkaammaksi ja säätää toimintatapojaan sen mukaisesti.

Yllä esitetyn periaatteen mukaan tiimi kykenee aina vain parempaan suoritukseen sisäisten keskustelujen ja arviointien myötä.

Esitettyjen periaatteiden ja niiden analysointien perusteella voidaan tulla siihen tulokseen, että ketterän kehittäjän tulee olla motivoitunut, kokenut, taidokas ja itsenäiseen työhön kykenevä, mutta kuitenkin yhteisen hyvän tärkeimmäksi asettava henkilö, jolla on hyvät kommunikointitaidot ja joka kykenee oman panoksensa kautta vaikuttamaan tiimin suorituksen laatuun positiivisesti. Vaikuttaisi siltä, että ketteriä menetelmiä käyttävältä tiimiltä vaaditaan paljon.

Mikäli tiimin jäsenet ovat toisilleen entuudestaan tuntemattomia, kestää aikansa että ihmiset tutustuvat kunnolla ja saavat tätä kautta luottamusta toisiinsa. Ketterien menetelmien sisältämät moninaiset tapaamiset, kuten päivittäiset tilannepalaverit ovat omiaan ihmisten keskinäisten suhteiden luomiseen. Tiimiä muodostettaessa projektia varten voi vastarintaa muodostua silloin, kun idea testaajan ja kehittäjän vierekkäisistä työpisteistä kokee päivänvalon. Ajatus on kuitenkin hyvä, sillä ketterän tiimin luonteenpiireisiin ei kuulu tiimin jäsenten erillään oleminen. (Davies & Sedley 2009, 51.)

Testaajien integroiminen osaksi ketterää tiimiä on järkevä toimenpide, mutta siirtymä ei välttämättä ole niin yksinkertainen, kuin miltä se kuulostaa. Testaajat kokevat monesti paikkansa ja tehtävänsä tiimissä epäselväksi. Lisäksi verrattuna kehittäjien pariohjelmoinnin (*pair programming)*, TDD:n ja muiden ketterien käytäntöjen koulutuksiin, testaajat eivät monesti saa minkäänlaista koulutusta tarvitsemiinsa työtehtäviin, kuten paritestaukseen, työskentelyyn keskeneräisten ja muuttuvien vaatimusten kanssa, automatisointiin ja kaikkiin muihin uusiin taitoihin joita vaaditaan. Toisaalta testaajat kokevat kykenevänsä parempiin tuloksiin työssään ollessaan lähellä kehittäjiä, ja yhteiset lounastauot puolestaan kasvattavat tiimihenkeä positiiviseen suuntaan. (Crispin & Gregory 2009, 61.)

Tiimin koostumuksen sisältäessä myös testaajia, on etuna se, että projektille on yksi yhteinen budjetti ja aikataulu. Tämä varmistaa sen, että testauksesta ei voida tinkiä tapauksissa, joissa jonkin toiminnallisuuden toteutus venyy. Jos jotakin ominaisuutta ei ehditä testata, se tarkoittaa, että kyseistä ominaisuutta ei siis

ehditä tehdä ollenkaan. (Crispin & Gregory 2009, 61.) Itkonen ym. (2005) puolestaan pitävät kiinteitä aikarajoja huonoina testauksen kannalta: testausjaksoa ei ole mahdollista pidentää, vaikka ohjelmistovirheitä löytyisi enemmän mitä oli arvioitu.

5.5.4 Teknologia

Teknologiaan liittyvät asiat muodostavat yhden osakokonaisuuden onnistuneelle pilotille. Jotta tiimi voisi hoitaa työnsä, tulee sillä olla oikeanlaiset työkalut. (Mar & Szalvay 2008.) Nerur ym. (2005) nostavat esille myös ketterissä menetelmissä tunnistamiaan teknologiaan liittyviä ongelmia ja heikkouksia. Heidän mielestään ketterien menetelmien käyttö ei välttämättä ole yhtä jouhevaa keskuskone-teknologiaan perustuvien järjestelmien (*mainframe technology*) kuin oliopohjaisten järjestelmien (*object oriented systems*) kehittämiseen. Ketterien menetelmien käyttöä suunnittelevien organisaatioiden on investoitava työkaluihin, jotka tukevat nopeaa ja iteratiivista ohjelmistokehitystä, versionhallintaa, refaktorointia ja muita ketterien menetelmien käytäntöjä (Nerur ym. 2005).

Koodin jatkuvan refaktoroinnin merkityksestä ja mahdollisista haittavaikutuksista ei ole päästy selkeään yhteisymmärrykseen. Turk ym. (2002) näkevät refaktoroinnin vaarana suunnittelulle ja arkkitehtuurille, sillä ohjelmiston muuttaminen tällä keinolla voi tehdä ohjelmistosta virhealttiin. Lindvall ym. (2002) puolestaan esittävät, että kun testaus tehdään hyvin ja oikealla tavalla, ei refaktorointia ja arkkitehtuurillisia muutoksia pidetä riskiä lisäävinä tekijöinä. Marin ja Szalvayn (2008) mukaan teknologian ohella projektin onnistumisen todennäköisyys kasvaa myös kehityskäytäntöjen myötä, sillä ne täydentävät Scrumia. Näitä ovat muun muassa jatkuva integrointi, TDD ja pariohjelmointi (Mar & Szalvay 2008).

5.6 Onnistumisen mittaaminen ja seuranta

Ketterän menetelmän pilotoinnin tai käyttöönoton yksi tärkeimmistä osa-alueista on määrittää ja mitata sitä, kuinka hyvin on onnistuttu. Cohnin (2009, 429) mukaan Scrumin käyttöönotto on kompleksi prosessi, ja vastausta siihen, kuinka käyttöönotossa on onnistuttu, ei ole helppo antaa.

Seuraavaksi esitellään yleisimpiä kysymyksiä Scrumin käyttöönotosta Cohnin (2009, 429–430) mukaan:

- Onko Scrumiin panostaminen ollut hyödyllistä ja kannattavaa?
- Mitä meidän tulisi parantaa seuraavaksi?
- Tulisiko meidän jatkaa Scrumin käyttöä?
- Olemmeko parempia ohjelmistokehityksessä nyt, kuin olimme vuosi sitten?
- Teemmekö parempia tuotteita?
- Sisältävätkö tuotteemme vähemmän ohjelmistovirheitä?
- Olemmeko nopeampia kuin olimme aiemmin?

Käyttöönoton onnistumista kuvaavien, yksinkertaistenkin metriikoiden kerääminen on työlästä ja työtä, jota ohjelmistoalalla ei ole totuttu tekemään. Metriikan avulla saadaan monenlaisia graafeja ja tilastoja jotka havainnollistavat muutoksen onnistumista. Cohnin (2009, 443) mukaan suurimmat metriikan keräämisestä saatavat hyödyt ovat läpinäkyvyyden lisääntyminen, tärkeimpien jatkokehityskohteiden selville saaminen ja parantunut taistelukyky vanhoihin tapoihin ajautumista vastaan.

5.7 Esimerkkiprojekteja

Seuraavaksi käydään läpi muutamia ketterien menetelmien käyttöönottoon liittyviä esimerkkejä. Niiden avulla pyritään havainnollistamaan ohjelmistokehitysmenetelmän vaihdossa huomioon otettavia tärkeitä seikkoja, joihin organisaatioiden olisi erityisen hyvä kiinnittää huomiota.

Sumrell (2007) kuvaa artikkelissaan lääkealan ohjelmistotalon muutosta vesiputousmallisesta ohjelmistokehityksestä Scrumiin. Käytettäessä vesiputousmallia organisaation tuotepäälliköt kirjoittivat laajoja vaatimusdokumentteja, joiden perusteella kehittäjät toteuttivat sovelluksia, ja laadunvarmistustiimi testasi tuotokset ahkerasti ennen asiakkaalle toimitusta (Sumrell 2007).

Syitä muutoshalukkuuteen oli useita. Ajoittain ilmeni epäselvyyksiä siitä, mitä asiakas oikeasti haluaa ja aikaa kului liikaa niin sovellusten testaamiseen kuin uusien, turhien ominaisuuksien kehittämiseen. Testaamisen myötä löydettiin ongelmia, joiden korjaaminen ja uudelleentestaus veivät liikaa aikaa. Organisaatiossa tiedostettiin tarve organisoida ohjelmistokehitysprosessi uudelleen siten, että uusia sovelluksia ja ominaisuuksia saataisiin jatkossa nopeammin markkinoille, ja siten myös asiakkaalta nopeammin palautetta. Testaus haluttiin aloittaa kehitysprosessin alussa, että vältettäisiin laajamittainen uudelleen tehtävä työ ennen käyttöönottoa. Nämä asiat toteuttamalla uskottiin, että asiakas saisi vain sitä mitä haluaa – nopeammin kuin normaalisti. (Sumrell 2007.)

Kehitysorganisaatiossa ymmärrettiin tarve siirtyä vesiputousmallista ketterimpiin käytäntöihin. Useiden ohjelmistokehitysmenetelmien vertailun ja harjoitteluiden tuloksena organisaatiossa katsottiin Scrumin olevan sopivin heidän tarpeisiinsa. Koska kenelläkään ei ollut käytännön kokemusta ketterästä ohjelmistokehityksestä, osallistui kaksi työntekijää pilottitiimistä Scrum Master Certification -ohjelmaan samalla kun muut opiskelivat kirjallisuuden avulla ketteristä menetelmistä. (Sumrell 2007.)

Pyrähdykset eivät kuitenkaan menneet odotetulla tavalla: Regressiotestaus ei mahtunut pyrähdyksen aikatauluun ja testauksen automatisointi ei onnistunut puutteellisen käyttöliittymän takia. Tiimin jäsenet turhautuivat ketterän testauksen etenemättömyyden takia. Sumrell (2007) ei halunnut uskoa organisaation toimintaympäristön olevan niin kompleksinen, että se estäisi testauksen ottamisen mukaan pyrähdyksiin. He päättivät palkata kaksi ketteriin menetelmiin erikoistunutta konsulttia paikan päälle valmentamaan ihmisiä. Yksi pääparannuskohteista oli saada vastaus siihen, kuinka laadunvarmistuskäytännöt saataisiin toimimaan ketterässä projektissa. Konsulttien avulla ymmärrettiin, että tähän tarvittaisiin työtapojen muutoksia koko tiimin tasolla. (Sumrell 2007.)

Artikkelin lopussa Sumrell (2007) listaa kuusi tärkeintä opetusta heidän ketterien menetelmien pilotista, jossa erityisesti laadunvarmistus ja testaus osoittautui vaikeaksi asiaksi muuttaa. Ensimmäiseksi hän kehottaa olemaan realisti ja turvautumaan ketterien menetelmien huippuasiantuntijoiden ja konsulttien apuun, mikäli heille koetaan olevan vähänkään tarvetta. Toisekseen Sumrellin (2007) mielestä on syytä unohtaa vanhat roolit ja vastuut – ne eivät todennäköisesti tule säilymään entisellään ketterien menetelmien käytäntöjen myötä. Kolmantena korostetaan jokaisen tiimin jäsenen vastuuta ohjelmiston laadusta: se ei ole pelkästään laatuihmisten ja testaajien asia. Neljäntenä asiana ohjeistetaan aloittamaan jonkin kevyehkön testauksen automatisointityökalun käyttö niin pian kuin mahdollista. Viidenneksi on korostettu muutoksenhallinnan tärkeyttä ja joustavuutta, ja kuudes, viimeinen ohje kehottaa olemaan realisti: Ei kannata odottaa muutosprosessin vesiputousmallista ketteriin menetelmiin sujuvan ongelmitta ja yhdessä yössä. (Sumrell 2007.)

Marchi (2009) toteaa työskentelemässään organisaatiossa Scrumin käyttöönoton yhteydessä tehtyjen virheiden opettaneen paljon. Marhcin (2009) mielestä mikä tahansa kehitettävä toiminnallisuus voidaan lähes poikkeuksetta pilkkoa pienempiin osiin, minkä seurauksena havaitaan pienempiä alitoiminnallisuuksia, joista osa on kriittisiä alkuperäiselle toiminnallisuudelle ja osa ei. Marchi (2009)

kehottaakin jättämään pois ne osat, joiden todetaan olevan turhia. Tästä huolimatta voidaan edelleen toteuttaa toiminnallisuus, jossa ei enää ole mitään ”turhaa” (Marchi 2009).

Salesforce.com puolestaan käytti vesiputousmallia hyvin tuloksin organisaation ollessa pieni. Kasvun myötä muutospaineita ohjelmistokehitystapaan aiheuttivat muun muassa projektin alussa tehdyt epätarkat arvioinnit, jotka johtivat päivämäärien siirtymisiin, läpinäkyvyyden puutteeseen tuotejulkaisun eri tasoilla, liian myöhään saatuihin palautteisiin toiminnallisuuksille ja tuottavuuden vähenemiseen tiimin kasvun myötä. Tuloksena oli yhdenaikainen, koko organisaation laajuinen ketteriin menetelmiin siirtyminen, joka onnistui yli odotusten vain kolmessa kuukaudessa. (Fry & Greene 2007.)

Hyvin menneestä ketterien menetelmien käyttöön siirtymisestä huolimatta Fry ja Greene (2007) nostavat esille asioita, joita Salesforce.com:ssa tehtäisiin toisin, mikäli muutos ketteriin menetelmiin voitaisiin tehdä uudelleen. Tuotteen omistajat koulutettaisiin aiemmin ja suuremmalla panostuksella. Ulkopuolinen kouluttaja hankittaisiin aiemmin. Ketterien menetelmien vaatimat tekniset asiat, kuten infrastruktuuri testauksen automatisointiin hoidettaisiin kuntoon heti alussa. Ketterien menetelmien ”säännöt” otettaisiin vakavammin, sillä esimerkiksi itseohjautuvuus voidaan käsittää monella eri tavalla. (Fry & Greene 2007.)

Kniberg (2006) esittää järkeväksi ja tehokkaaksi tavaksi ajoittaa tiimien pyrähdykset samaksi, jolloin organisaatioon saataisiin yhtenäinen ”sydämensyke”. Idea on hyvä, mutta siihen ei ole helppoa päästä, sillä organisaatioiden projektien aikataulut voivat olla vahvasti riippuvaisia asiakkaista ja testausresurssit on otettava huomioon. Projektien kulkiessa samalla syklillä tulisi testaajille huomattavia työpiikkejä säännöllisin väliajoin.

Jochems ja Rodgers (2007) kuvaavat artikkelissaan projektin onnistumista, kun ohjelmistokehitysmenetelmäksi valittiin vesiputousmallin sijaan ketterät menetelmät. Yksi esille nostetuista, opituista asioista liittyy testaukseen: Ketterissä

menetelmissä tapahtuvaan testauksen käytäntöihin ja työkaluihin kannattaa hankkia riittävästi oikeanlaista koulutusta. Lisäksi huomattiin, että viestintäkanavat on pidettävä auki jatkuvasti, ja esille nousseisiin ongelmiin on puututtava välittömästi. Unohtaa ei sovi myöskään sitä, että on annettava projektitiimin itsensä kehittää ja omistaa käytössä oleva prosessi. (Jochems & Rodgers 2007.)

6 EMPIIRISEN TUTKIMUKSEN KOHDE, MENETELMÄT JA TOTEUTTAMINEN

Tässä luvussa esitellään tutkimuksen käytännön toteutukseen liittyviä olennaisia tekijöitä. Luvussa esitellään tutkimuksen tavoite ja tutkimusmenetelmät sekä kuvataan toimeksiantajaorganisaation tiimiä, jossa pilotoinnit tehdään.

6.1 Tutkimuksen kohde ja tavoitteet

Tutkimuksessa oli mukana kaksi Scrumia pilotoinutta projektia. Projektien valintaperusteina olivat pääasiassa niiden suotuisa ajankohta ja erilaisuus: kesto, koko, asiakas, käytettävä käyttöympäristö eli Projekti A:n Notes-client ja Projekti B:n www-selain. Lisäksi alkuhaastatteluihin valittiin henkilöitä kaikista tiimeistä ja tasoista, kehittäjistä tiiminvetäjiin ja johtoportaaseen. Tutkimuksen tavoitteena on selvittää mitä haasteita ja muutostoimenpiteitä ohjelmistotestaukseen on odotettavissa, kun projekteissa tapahtuvaa ohjelmistokehitystä aletaan tehdä perinteisten ohjelmistokehitysmenetelmien sijaan Scrumia käyttäen.

6.2 Tutkimusmenetelmä

Tutkielman tutkimusmenetelmä on tapaustutkimus, jonka tiedonhankintatekniikoita ovat kirjallisuuskatsauksen lisäksi toimeksiantajaorganisaatiossa toteutettu puoliavoin haastattelu, pilottiprojektien jäsenille tehty kysely sekä osallistuva havainnointi. Seuraavaksi esitellään tarkemmin tutkielmassa käytettyjä tutkimusmenetelmiä.

Kyselyn ja haastattelun avulla saadaan selville, mitä henkilöt ajattelevat, tuntevat ja uskovat. Toisaalta niiden avulla ei kuitenkaan selviä se, mitä todellisuudessa tapahtuu. (Hirsjärvi, Remes & Sajavaara 2008, 207.) Haastatteluista ei

myöskään voida varmistua siitä, miten vakavasti vastaajat ovat tutkimukseen suhtautuneet.

Hirsjärven ym. (2008, 130) mukaan tapaustutkimuksella tarkoitetaan yksityiskohtaista, intensiivistä tietoa yksittäisestä tapauksesta tai pienestä joukosta toisiinsa suhteessa olevia tapauksia. He täsmentävät tapaustutkimuksen luonnetta kuvaamalla sen tyypillisiä piirteitä: Valitaan yksittäinen tapaus, tilanne tai joukko tapauksia, joissa kohteena on yksilö, ryhmä tai yhteisö; kiinnostuksen kohteena ovat usein prosessit; aineistoa kerätään useita metodeja käyttämällä, muun muassa havainnoin, haastatteluin ja dokumentteja tutkien (Hirsjärvi ym. 2008, 130; ks. myös Yin 2003, 12–14).

Tapaustutkimusta, monipuolisuudessaan, voidaan käyttää mistä tahansa filosofisesta näkökulmasta, olkoon se sitten positivisti, tulkitseva tai kriittinen. Tyypillisesti tapaustutkimus yhdistääkin useita kvalitatiivisia tiedonkeruutapoja kuten haastattelut, dokumentointi ja havainnointi, mutta siihen voi sisältyä myös kvantitatiivista tiedonkeruuta kyselyjen ja aikajaksojen muodossa. (Dubé & Paré 2003.)

Yin (2003, 89–90) pitää haastatteluja yhtenä tärkeimmistä tapaustutkimuksen tietolähteistä. Haastattelut ovat luonteeltaan monesti enemmänkin ohjattuja keskusteluja rakenteellisten kyselyjen sijaan, eli vaikka pyrkimyksenä olisi toteuttaa tiukka ja raamitettu kysely, voi käytännön toteuma olla aihepiirin ympärillä aaltoileva. Lisäksi tapaustutkimuksen haastattelut tarjoavat mahdollisuuden esittää avainasemassa oleville haastateltaville niin faktoja antavia kysymyksiä kuin myös heidän mielipiteitään avaavia kysymyksiä. Tietyissä tilanteissa voidaan jopa tiedustella haastateltavalta ehdotuksia jatkotutkimusaiheille tai muista potentiaalisista ja hyödyllisistä haastateltavista. (Yin 2003, 89–90.)

Patton (2002, 4) määrittelee havainnoinnin kvalitatiivisen aineiston keruumenetelmäksi, jossa havainnoista saatu data koostuu ihmisten aktiviteettien, käyttäytymisen, toimenpiteiden, ihmisten välisten vuorovaikutusten ja organisatoristen

prosessien yksityiskohtaisista kuvauksista. Patton (2002, 5) jatkaa toteamalla kvalitatiivisen aineiston (*data*) laadun riippuvan suurissa määrin metodologisista taidoista, havainnointikyvystä ja tutkijan rehellisyydestä. Järjestelmällinen ja täsmällinen havainnointi ei ole pelkästään läsnä olemista ja ympärille katsomista. Toisaalta, haastattelutkin vaativat paljon muuta kuin pelkän kysymysten esittämisen. Hyödyllisten ja uskottavien havaintojen ja tulosten saaminen havainnoinnin ja haastattelujen myötä vaatii kuria, tietämystä, harjoittelua, luovuutta ja kovaa työtä. (Patton 2002, 5.)

Havainnointimetodina tutkimuksessa käytettiin sekä osallistuvaa havainnointia (Yin 2003, 93–94) että ääneenajattelu-lähestymistapaa (*think-aloud)* (Patton 2002, 385). Osallistuva havainnointi on tutkimuksen aineiston keruuta, jossa tutkija ei ole puhtaasti passiivinen havainnoija. Suurimpana etuna osallistuvaa havainnointia käytettäessä Yin (2007, 94) näkee mahdollisuuden päästä mukaan tapahtumiin tai ryhmään, jotka muuten jäisivät saavuttamattomiin. Toisekseen tutkimusta tekevä osallistuva havainnoija saa mahdollisuuden havaita todellisuus tutkittavan ryhmän jäsenen näkökulmasta ulkopuolisen tarkkailijan sijaan (Yin 2007, 94).

Ääneenajattelu-lähestymistavassa ideana on havainnoitavien osapuolien ääneen ajattelu samalla, kun he suorittavat asioita (Patton 2002, 385). Tässä tutkielmassa seurattiin siis ensisijaisesti Scrumin projektiin tuomien uusien käytäntöjen ja työtehtävien suorittamista ja niistä keskustelua, sisältäen testaukseen kohdistuvat haasteet. Lisäksi tarkasteltiin myös käytännön toteutumia, kuten esimerkiksi kehittäjien testitapausten syntymistä ja laatua. Havainnointia tehtiin viikoittain johtuen Scrumin aamupalavereista ja havainnoijan työpisteen sijainnista projektiryhmien välittömässä läheisyydessä. Molemmissa projekteissa havainnointiyksikkönä oli koko projektiryhmä, eli henkilötasolle asti menevää seurantaa ei käytetty.

Tutkielmassa on siis käytetty kolmea eri tiedonkeruumenetelmää: puoliavointa haastattelua, osallistuvaa havainnointia ja sähköpostin välityksellä suoritettua

kyselyä pilottiin osallistuneille henkilöille. Seuraavaksi esitellään näiden tiedonkeruumenetelmien organisointi ja toteutus.

6.2.1 Alkuhaastattelut

Loppukesällä 2008 toimeksiantajaorganisaatiossa suoritettiin haastatteluja, joiden aiheina olivat ohjelmistotestauksen silloinen tila ja tärkeimmät kehityskohteet. Haastateltavia oli yhteensä 19. Haastateltaviksi valittiin organisaation koko johto (neljä henkilöä), kaikki tiiminvetäjät (viisi henkilöä) ja tiimin jäseniä (yhdeksän henkilöä). Haastattelujen tuloksena saatiin siis lista testauksen kehittämiskohteista. Haastatteluiden tulokset esitellään myöhemmin luvussa 7.1. Saadut tulokset johtivat osaltaan Scrumin pilotointiin Notes-tiimissä.

Puoliavoimessa haastattelussa kaikista tiimistä haastateltiin tiiminvetäjä ja kahdesta kolmeen hänen valitsemaansa, oman tiiminsä ohjelmistokehittäjää. Haastattelukysymykset olivat hieman toisistaan poikkeavia riippuen siitä, oliko haastateltava johtoryhmästä, tiiminvetäjä vai kehittäjä. Haastattelurungot esitellään liitteessä 2. Tiiminvetäjiä ohjeistettiin valitsemaan tiimeistään haastateltavaksi sellaisia henkilöitä, joiden haastattelusta olisi mahdollisimman paljon hyötyä testauskäytäntöjen epäkohtien selvittämiseksi ja parantamiseksi. Patton (2002, 242–243) kutsuu edellä mainittua otantavalintamenetelmää tarkoituksenmukaiseksi näytteenotoksi (*purposeful sampling)*. Siinä valitut kohteet ovat määrätietoisesti valittuja tietopitoisia otoksia (*information-rich case)*. Lisäksi haastateltiin yrityksen johtoryhmä, johon kuuluvat neljä henkilöä ovat myös yrityksen perustajajäseniä.

Haastattelussa haastattelijoita oli kaksi ja haastateltavana kerrallaan yksi henkilö. Haastattelurunko kysymyksineen oli käytössä, mutta sitä ei noudatettu täsmällisesti, vaan haastattelu oli luonteeltaan keskustelu, joka ohjasi itseään. Puoliavoimen haastattelumuodon käyttöön päädyttiin, sillä sen nähtiin tarjoavan mahdollisuuden selvittää syvällistä tietoa tutkittavasta aiheesta.

6.2.2 Havainnoinnin ja loppukyselyjen organisointi ja toteutus

Tutkielmassa suoritettua osallistuvaa havainnointia käytettiin molemmissa pilottiprojekteissa. Projektitiimien jäsenille ei erikseen ilmoitettu havainnointia suoritettavan, sillä se olisi voinut vaikuttaa ihmisten käyttäytymiseen ja avoimeen keskusteluun.

Loppukysely Scrumin pilotoinnin ja testauksen onnistumiseen liittyen järjestettiin molempien pilottiprojektien jäsenille. Kyselyyn otettiin mukaan pilottiprojektien tiimeistä kaikki henkilöt työtehtäviin tai projekteissa tehtyihin työmääriin katsomatta. Kysymyksiä oli 13, ja ne lähetettiin vastaajille sähköpostitse. Kysymykset ovat liitteessä 3.

6.3 Pilotointi

Tapaa, jota tiimin keskisuurissa (300–700 tuntia) projekteissa käytettiin, on vaikeaa kategorisoida yhteen ohjelmistokehitysmenetelmään. Lähimpänä yleisesti kirjallisuudessa esitetyistä menetelmistä lienee inkrementaalinen ohjelmistokehitystapa. Käyttäjien todelliset tarpeet eivät toisinaan olleet tiedossa vielä siinäkään vaiheessa, kun muutospyyntöjä tuli kehittäjille. Täten, kun muutoksia tehtiin, tuotos annettiin asiakkaan kokeiltavaksi ja kommentoitavaksi. Stabiilit, kiireelliset ja/tai asiakkaan priorisoimat komponentit ja osakokonaisuudet toteutettiin ensin. Kokeiluversioiden valmistus ja tarjoaminen asiakkaalle ja niistä saatu palaute nähtiin osana kehitystyötä. Luonnollisesti epäselviä toteutuspyyntöjä ei lähdetty tekemään, vaan ne vaativat selvittämistä ennen toteutusta.

Perinteisen ohjelmistokehitysmenetelmien elinkaari on monissa tapauksissa kankea ohjelmistotuotteen laadun takaamiseen, mikäli aikaa tuotteen saamiseksi ohjelmistokehityksestä markkinoille halutaan samalla vähentää. Monissa tapauksissa laadunvarmistus ja ohjelmistotestaus ovat ne asiat, joita lykätään ja vähennetään sovittujen takarajojen ja määräaikojen mennessä umpeen. Tällöin

yrityksillä voi olla tarve miettiä uutta prosessimallia, joka ottaa huomioon ja arvostaa laatunäkökulmaa jokaisella ohjelmistokehityksen tasolla ilman, että tuotteen toimitukseen sovitut aikataulut kärsivät.

6.3.1 Pilotoinnin motiivit

Tiimin toimintatavoista, ainakin joidenkin projektien kohdalta käytetyistä toimintatavoista, voidaan tunnistaa äärimmäistä ketteryyttä jo vuosien takaa. Tuskin ketterämpää ja mutkattomampaa ohjelmistokehitystä on olemassakaan, kuin seuraavassa esimerkissä: Asiakas ottaa yhteyttä ohjelmistokehittäjään ja pyytää tekemään pienen muutoksen tai korjauksen asiakkaalla jo käytössä olevaan järjestelmään. Kehittäjä toteuttaa muutoksen välittömästi ja päivittää asiakkaan testiympäristön. Asiakas testaa haluamansa muutoksen, toteaa sen toimivaksi ja ottaa muutokset käyttöön tuotantoympäristössä. Missään vaiheessa ei ole välttämättä tarvittu palavereja, dokumentaatiota tai muita yleisiä projektinhallintaan liittyviä toimenpiteitä.

Perusteita Scrumin pilotoinnille toimeksiantajaorganisaation tiimissä on useita. Näitä ovat esimerkiksi asiakkaiden lisääntyvät viime hetken muutokset ja vaateet kyetä reagoimaan muutoksiin nopeammin ja suopeammin. Lisäksi organisaation toisessa tiimissä on saatu hyviä tuloksia Scrumin käytöstä vuosien mittaan. Tiimi haluaa luonnollisesti kehittyä ja kokeilla uusia toimintatapoja. On järkevää kokeilla Scrumin käyttöä proaktiivisesti, ennen kuin jonakin päivänä asiakas kysyy, miksi Scrumia ei käytetä tai ole edes harkittu kokeiltavan. Asiakkailta on jo vuosien ajan tullut muutospyyntöjä kaikissa projektin vaiheissa. Haasteena on ollut kuitenkin muutosten priorisoimattomuus ja projektin loppuvaihe: muutosten huomataan alkavan vaikuttaa aikatauluun ja kustannuksiin. Tämän seurauksena asiakkaan kanssa aloitetaan neuvottelut siitä, että kaikkea alun perin suunniteltua ei voidakaan välttämättä toteuttaa jäljellä olevan ajan ja budjetin takia. Scrumin uskotaan lisäksi palvelevan asiakasta paremmin kuin menetelmät, joissa muutoksia ei voitaisi aidosti ottaa vastaan toteutuksen ollessa täydessä vauhdissa.

Pilotoinnin myötä ajatusmallia muutettiin: vaatimusten jatkuvaa muuttumista moneen kertaan ja suuntaan ei pidetty enää ongelmana, vaan vallitsevana tilana, johon oli mukauduttava ja johon haluttiin löytää toimintamallit. Projektien ja niissä toimivien henkilöiden valintaan vaikuttivat sekä projektien sopiva ajoitus ja koko, että niissä jo aiemmin toimineiden henkilöiden tunnetut taidot ja kokemus. Toisaalta projekteihin haluttiin myös vähemmän aikaa talossa olleita kehittäjiä.

Luvun alussa esitetty esimerkki tiimin toimintatavoista on edelleen todellisuutta tiimin pienissä projekteissa, eikä esimerkin kaltaisiin projekteihin ole tarvetta tai edes järkevää lähteä soveltamaan Scrumia. Isompiin sen sijaan on ja niissä tähän asti käytetyt itsetekoiset, niin sanotusti ketterät toimintatavat halutaan saada systemaattisemmiksi, ettei asioita tehdä reaktiivisesti. Haluttuja tuloksia ovat parempi projektinhallinta, ennustettavuus, seurattavuus sekä asiakkaan aktivoiminen projektin kaikkiin vaiheisiin. Lisäksi halutaan saada parempi hyöty irti käytössä olevista ohjelmistokehitystyökaluista ja pysyä ohjelmistokehityksen kärjessä. Tiimi haluaa siis olla kokonaisuutena tarkastellen ketterä: pienissä projekteissa ollaan jo, isommat projektit uskotaan saatavan ketteriksi Scrumin avulla.

Projekteissa, joissa on luotu testauksen ohessa testaussuunnitelmia ja -raportteja, koettiin tällaisen dokumentaation tekemiseen menneen melko paljon aikaa niistä saatuihin hyötyihin nähden. Testitapauksista ja -suunnitelmista on kylläkin monesti asiakkaalle hyötyä ja joskus asiakas vaatiikin testausdokumentaatiota, jolloin ne luonnollisesti tehdään ja toimitetaan. Testausdokumentaation ajan tasalla pitäminen koettiin haasteelliseksi: testausdokumentaation testitapaukset saattoivat kohdistua sellaisiin ominaisuuksiin, joita ei vastoin aiempia suunnitelmia ollut päätetty toteuttaa ollenkaan. Haasteeksi koettiin myös edellytys tuntea hyvin liiketoimintaspesifisten projektien kehitettävät järjestelmät: liiketoimintakriittisten ja kompleksisten ohjelmistojen tunteminen ja testaaminen vie aikaa. Nämä syyt johtivat toisinaan siihen, että projektille vara-

tuista testaustunneista kului enemmän aikaa testitapausten suunnitteluun ja kirjoittamiseen kuin testitapausten suorittamiseen ja tulosten raportointiin, mikä ei ole kovin tehokasta ohjelmiston toimivuuden selvittämistä ajatellen.

Scrumin myötä haluttiin saada läpinäkyvyyttä projekteihin. Monelle toimeksiantajaorganisaation Notes-tiimin projektille on ollut tyypillistä, että kehittäjät ja projektipäälliköt saavat asiakkaalta suoraan sähköpostitse muutospyyntöjä ja ohjelmistovirhekuvauksia kehitettävään tai jo käytössä olevaan järjestelmään. Toimintatapa on hyvä siinä mielessä, että kommunikointia asiakkaan suuntaan ei tehdä jokaisen toimittajapuolen toimesta, vaan keskitetysti. Kuitenkin, projektiryhmän sisäiset viestintäkäytännöt nousevat tällöin tärkeään asemaan projektin läpinäkyvyyden ja tiedon kulkemisen takaamiseksi – muuttuneiden vaatimusten ja teknisten ratkaisujen on oltava kaikkien, jopa testaajan tiedossa. Lisäksi kehittäjien on hyvä saada mahdollisimman nopeaa palautetta tekemistään muutoksista, ja testauksen tuomat havainnot antavat esimerkiksi viikkoraporttia paremmin koko projektitiimille reaaliaikaisen ja tarkemman kuvan siitä, miten projekti todellisuudessa etenee.

Projekteissa, joissa testaus oli projektin lopussa, testaajat olivat monesti ylikuormitettuja projektin loppuvaiheilla. Lisäksi testauksen myötä löydettyjä ohjelmistovirheitä jouduttiin punnitsemaan tarkkaan: ehdittäisiinkö niitä korjata enää ennen edessä olevaa ensimmäistä julkaisua vai ei. Virheiden löytyminen ja korjaaminen lopussa johti siis helposti aikataulujen pettämiseen.

6.3.2 Pilotoinnin valmistelut

Notes-tiimin sisäisiä Scrum-koulutuksia pidettiin muutama ja niissä saatiin hyvää keskustelua aikaan. Scrumin periaatteita kehuttiin hyviksi teoriatasolla, mutta niiden käytännön toimivuutta epäiltiin. Tiimissä on kehittäjiä, jotka tekevät montaa projektia yhtä aikaa ilman varsinaista hallinnollista projektipäällikköä. Tämä tarkoittaa käytännössä sitä, että kehittäjä sekä tekee toteutuksen että hoitaa samalla projektipäällikön tehtävät ja asiakasyhteydenpidon. Pienimpien

projektien hyvänä puolena pidettiin läpinäkyvyyttä: asianomainen kehittäjä tietää projektin tilanteen, joten dokumentointia ei pidetty tärkeänä.

Keskustelujen myötä nousi esille, että asiakkaiden muutospyyntöjen dokumentointi hoidettiin joissakin tapauksissa epäsystemaattisesti esimerkiksi siirtämättä sähköpostissa tai puhelimitse saatuja muutospyyntöjä vaatimustenhallintatyökaluun. Lisäksi asiakkailta tulevien kehitys- ja muutospyyntöjen sanottiin olevan toisinaan liian teknisiä, jolloin ei olla varmoja, onko muutospyyntö järkevä. Ratkaisu ongelmaan olisi saada asiakkaalta muutospyynnöt jatkossa enemmän business-tasoisina, jolloin toteutustapa jäisi kehittäjän vastuulle.

6.3.3 Lotus Notes ja Domino -teknologia

Työryhmäohjelmistolla, joka Lotus Noteskin on, tarkoitetaan ohjelmistoperhettä, jonka avulla työryhmän yksilöt voivat jakaa tietoja verkotetussa ympäristössä ajasta ja paikasta riippumatta. Työryhmällä tarkoitetaan tässä yhteydessä yhteisen päämäärän hyväksi työskentelevää joukkoa. Työryhmäohjelmistojen juuret ulottuvat vuoteen 1989, jolloin Lotus Notesin versio 1.0 ilmestyi markkinoille. (Virtala 2003, 2.)

Työryhmäohjelmiston avulla tiedonjako nopeutuu ja helpottuu: tieto on välitettävissä kaikille työryhmän jäsenille välittömästi ja samanaikaisesti. Tiedon jakaminen ja oikean tiedon helppo saatavuus ovat työryhmän työskentelyssä avainasemassa. Suurin osa työryhmän käyttämistä tietokoneelle tallennetuista tiedoista on jäsentymättömässä muodossa, kuten sähköposteissa ja erilaisissa asiakirjoissa. Lotus Notesilla pyritään keräämään ja hallitsemaan juuri tällaista tietoa. Yhtä tärkeää olisi myös saada kerättyä hiljaista tietoa, joka kulkee dokumentoimattomana ihmisten mukana. (Virtala 2003, 3.)

Perinteisin ja keskeisin työryhmäohjelmiston ominaisuus on sähköposti. Sähköposti ja kalenteritoiminnot ovat perustoimintoja, joita työryhmäohjelmistolta poikkeuksetta vaaditaan. Oman, henkilökohtaisen kalenterin lisäksi on hyvä

pystyä tarvittaessa näkemään muiden työntekijöiden menoja jaetun kalenterinäkymän kautta ryhmän yhteisen ajankäytön tehostamiseksi. Muita tärkeitä työryhmäohjelmiston toimintoja ovat muun muassa keskustelufoorumit, tietosisällön tuottaminen ja hallinta, tehtävien hallinta sekä hakutoiminnot. (Virtala 2003, 3–4.) Aiherajauksen voiksi toimintoja ei käsitellä tässä yhteydessä tarkemmin, vaan seuraavaksi esitellään varsinaista Notes-ohjelmistokehityksen puolta.

Virtala (2003, 8) määrittelee Lotus Notesin työryhmä- ja viestintäsovellusten kehittämiseen ja käyttämiseen tarkoitetuksi asiakas-palvelin -ohjelmistoksi. Asiakas-palvelin-ratkaisu tarkoittaa sitä, että ympäristössä on palvelintietokone, joka käsittelee asiakastietokoneiden eli käyttäjien työasemien lähettämiä pyyntöjä ja jakaa tietoa käyttäjille (Virtala 2003, 8).

Notes-käyttäjien palvelinohjelmistona toimii Lotus Domino, joka pystyy palvelemaan Notes-käyttäjien ohella myös internet-selainkäyttäjiä. Tämän mahdollistaa Dominon sisältämä WWW-palvelin. (Tulisalo ym. 2002, 4.) Domino-palvelimella on lukuisia tehtäviä. Näistä keskeisimpiä ovat Notes-asiakirjoja ja sovellusrakenteita sisältävien tietokantojen hallinta, tietoturvan ja käyttäjäoikeuksien hallinta, hajautettujen tietokantojen synkronointi eli toisintaminen sekä sovellus- ja sähköpostipalvelujen tarjonta työasemille. Työasemille Lotus Notes tarjoaa puolestaan kolme erilaista ohjelmistoa: Lotus Notes -client, joka on järjestelmän käyttäjien ohjelmisto, Domino Designer sovelluskehittäjille ja Domino Administrator järjestelmien ylläpitäjille. (Virtala 2003, 8).

Buchanin (2006) mielestä Notes-sovellusten testaamiseen tulisi käyttää eri rooleissa toimivia henkilöitä ja eri ympäristöjä: Kehittäjillä tulisi olla kehitysympäristö toteutusta ja yksikkötestausta varten. Testaajan suorittamaa hyväksyntätestausta varten on oma testausympäristö. Järjestelmien ylläpitäjät puolestaan vastaavat tietokantojen päivityksistä ja siirtämisistä eri ympäristöjen välillä. (Buchan 2006.)

Domino tukee neljää sisään rakennettua kieltä, joita sovelluskehityksessä on mahdollista käyttää tilanteen mukaan. Näitä kieliä ovat makro, LotusScript, javaScript ja Java. Yleisesti ottaen voidaan sanoa, että paras valinta toteutuskieleksi kuhunkin tilanteeseen on se, jonka nähdään suoriutuvan tehtävästä vähimmällä ohjelmointityöllä. (Tulisalo ym. 2002, 558.) Lisäksi huomattavaa on, että sovelluksia on mahdollista tehdä joko Notes-clientille, web-selaimelle tai molemmissa yhtä aikaa toimivia.

Makrokieli on paras valinta esimerkiksi tilanteissa, jossa toissa halutaan toteuttaa operaatioita nykyiselle, auki olevalle dokumentille. Makrokieltä kannattaa käyttää myös silloin, kun tarvitaan toimenpide-nappeja (*action buttons*) peruskomentoihin kuten tiedoston tallennukseen. LotusScript on yleensä paras valinta silloin, kun haluttua toteutusta ei voida tehdä helposti makrokielellä. LotusScriptin avulla päästään helposti käsiksi toisiin Notes-tietokantoihin tai muihin kuin Notes-pohjaisiin tietolähteisiin. (Tulisalo ym. 2002, 558.)

JavaScript on hyvä työkalu web-lomakkeiden toteutukseen, erityisesti lomakkeiden kenttien validointiin. Selaimessa suoritettavan koodin avulla voidaan välttyä esimerkiksi lomakkeen lähetyksestä palvelimelle, joka tekee laskutoimituksia ja lähettää muokatun sivun takaisin käyttäjän selaimeen. Java puolestaan on hyvä valinta käytettäväksi esimerkiksi silloin, kun Notes-lomakkeisiin halutaan lisää toiminnallisuuksia Java-sovelmien avulla. (Tulisalo ym. 2002, 558.)

Agentit mahdollistavat automatisoitujen toimintojen suorittamisen erilaisten herätteiden perusteella Lotus Notes -tietokannassa. Agentit ovat yksittäisiä ohjelmia, joiden avulla voidaan suorittaa tietokannassa tiettyjä tehtäviä käyttäjälle, kuten täyttää tai tuhota dokumentteja tai lähettää sähköpostia. Agenttien kautta voidaan myös kommunikoida muiden, ulkoisten sovellusten kanssa. (Tulisalo ym. 2002, 248.)

Javaa ja LotusScriptiä verrattaessa voidaan yleisesti ottaen todeta molempien kykenevän ja soveltuvan perusasioiden toteuttamiseen. LotusScript saattaa olla hiukan nopeampi, ja käytännössä vasta Designer 8.5.1 -versio sisältää kunnollisen Java-editorin, jolla sovelluskehittäjän on mielekästä koodata Java-kieltä. Java-kielellä voidaan toteuttaa muun muassa ftp- ja http-socket-yhteyksiä, mikä on yksi Javan tarjoamista eduista. Lisäksi Javaan on olemassa suuri määrä valmiita kirjastoja yleisiin operaatioihin kuten pdf- ja Excel-tiedostojen generointiin.

7 TUTKIMUSTULOKSET

Tässä luvussa esitellään ennen pilotointia suoritettujen alkuhaastatteluiden tulokset. Tulokset sisältävät pilotoinnin aikaiset havainnot ja pilottiprojektien päättymisen jälkeiset, pilotteihin osallistuneille suoritetut kyselyt.

7.1 Alkuhaastatteluiden tulokset

Haastattelujen tuloksena tunnistettiin runsaasti testauksen kehittämiskohteita. Haastattelutulokset ovat kyseisen ryhmän antama yleisotos mielipiteistä, eli haastatteluista on poimittu ne asiat ja vastaukset, jotka ovat toistuneet haastatteluissa useimmin.

Kehittäjien vastaukset ovat peräisin kolmelta kehittäjältä tiimistä, jossa Scrum-pilotit päätettiin myöhemmin toteuttaa. Tällä halutaan peilata kyseisen tiimin silloista testaustilannetta ja testauksen haasteita ennen Scrumin kokeilua. Toisekseen muiden tiimien haastatteluhetkellä vallinnut testauksen tila ei suoranaisesti liity tähän tutkielmaan, sillä kyseisten haasteiden ja kehittämiskohteiden parantumista ja etenemistä ei ole mitattu ja dokumentoitu.

7.1.1 Kehittäjät

Notes-tiimin kehittäjät totesivat manuaalista yksikkötestausta tehtävän jonkin verran. Puutteeksi tunnistettiin testauksen kohdistuminen ainoastaan yksittäisiin, kehitettyihin toiminnallisuuksiin, minkä myötä kokonaisvaltaista järjestelmätestausta ei ole. Testauksen suunnittelua dokumentaation perusteella pidettiin hyvänä asiana, mutta sen onnistumisen nähtiin riippuvan projekteista: dokumentaatio vaihtelee projektien välillä paljon. Syynä dokumentaation määrän ja laadun vaihteluun nähtiin olevan epäselvyydet asiakkaiden haluamista toiminnallisuuksista.

Testaukselle koettiin olevan tarvetta enemmän, kuin haastatteluhetkellä vallinnut yleinen tilanne antoi myöten. Testaajista toivottiin jarruttamisen sijaan olevan apua sovelluksen kehitykseen. Haasteeksi kehittäjät kokivat oman toteutuksen testaamisessa ilmenevän sokeuden oman koodin testaukseen. Osa asiakkaista testaa heille toimitettavia ohjelmistoja aktiivisesti, mutta asiakkaiden testaus ei yleensä ota kantaa niinkään kooditasolle, vaan fokuksena ovat sovellusten varsinaiset toiminnallisuudet. Pääsääntöisesti asiakas olettaa tuotoksen olevan jo testattu ja toimiva. Joissakin projekteissa asiakas hoitaa järjestelmällisemmän testauksen kuitenkin itse.

Notes-tiimissä suoritettavaa laajempaa kokonaisuuksien testausta todettiin tehtävän lähinnä silloin, kun siihen jää aikaa. Tällöin testaus on mielivaltaista, eli sitä ei ole ennalta suunniteltu tai dokumentoitu millään tavoin. Testauksen automatisoinnille koettiin tarvetta, mutta sen toteuttamisen vaikeusastetta pidettiin teknologiasta johtuen melko korkeana.

Kehittäjät uskoivat testauksen edellytysten olevan pääsääntöisesti kunnossa. Tuki testaukseen esimiesten suunnalta oli olemassa, sillä tiiminvetäjä kannusti testaukseen ja sen kehittämiseen. Lisäksi testausta nähtiin pystyttävän suorittamaan käytössä olevien ohjelmistokehityksen työkalujen ja järjestelmien avulla. Eniten kaivattiin erilaisten testiraporttien generointimahdollisuutta ja testidokumentaation keskittämistä pelkästään yhteen järjestelmään.

Kehittäjien mielestä testauksen kehittäminen ja jalkauttaminen tiimissä edellyttää testaajien ottamista projekteihin heti aloituspalavereista lähtien ja testaukselle olisi varattava aikaa projektissa. Testausta projektin aikana tietyin aikavälein pidettiin myös tärkeänä. Kenenkään kehittäjän ei ajateltu koodaavan niin hyvin, että yksikkötestausta olisi vara jättää suorittamatta.

Testaus on kehittäjien mielestä ollut epämääräistä, melko vähäistä ja sattumanvaraista. Uusien toiminnallisuuksien kuvauksissa nähtiin kehittämisen varaa, ja asiaan panostamisella eli kuvauksien tarkentamisella uskottiin olevan positiivi-

nen vaikutus myös testauksen helpottumiseen ja parantumiseen. Nykyjärjestelmiin kaipailtiin parannuksia monilta osin: Niiden ei nähty soveltuvan testauksen avuksi. Toisaalta kehityshistorian näkyvyyteen kaivattiin selkeyttä, jotta olisi helppoa nähdä kuka on toteuttanut ja testannut mitäkin. Lisäksi esitettiin dokumentaation sisältyvyyttä samaan yhteyteen.

Yhteisiä pelisääntöjä kaivattiin, mutta todettiin tiimin projektien olevan niin erilaisia keskenään, että käytäntöjen tulisi olla sovellettavissa. Automatisoidun testauksen kartoitusta suositeltiin myös tehtävän. Suunnitelmiin ja tuntiarvioihin suositeltiin kiinnitettävän huomiota, sillä testaukseen ja sen myötä löytyvien virheiden korjaukseen menee aikaa yllättävän paljon.

7.1.2 Tiiminvetäjät

Jokainen tiiminvetäjä piti yhtenä suurimmista puutteista yhteisten, vakiintuneiden testauskäytäntöjen puuttumista. Tämän asian kuntoon saamiseksi ehdotettiin muun muassa käytännön työn kautta liikkeelle lähtemistä, nykyisten työkalujen soveltuvuuden arvioimista ja testausasenteen ja yhteistyöhalukkuuden parantamista.

Testausta kehittäjätasolla todettiin tehdyn jossain määrin, lähinnä silloin kun siihen on ollut aikaa ja asiakas on suostunut maksamaan siitä. Joidenkuiden kokeneempien kehittäjien koettiin kuitenkin käyttävän omia tapojaan tehdä yksikkötestaus vähintään kohtuullisen hyvin, mikä näkyi tasaisen hyvänä työn laatuna. Käyttöliittymätestausta on toisinaan tehty ristiin eri tiimien välillä. Testausosaamisen lisäämisen ja testauksen kartoittamisen arvioitiin lisäävän organisaation mainetta ja parantavan asiakkaan luottamusta. Aiemmin testausta ei ole välttämättä pidetty kovinkaan tuottavana asiakastyönä. Kiinnostusta testauksen kaupallistamiseen ja markkinointiin esiintyi laajasti.

Tiiminvetäjien mielestä testauksen perusasiat, kuten käytännöt ja asenne, on saatava kuntoon ennen mahdollisia testaustyökalujen hankintoja. Testauskäy-

täntöjen uskottiin vaihtelevan laajasti tiimien välillä. Projektien testausmäärissä nähtiin olevan suuria eroja. Haasteena pidettiin asiakkaan haluttomuutta maksaa testauksesta, puhumattakaan sen dokumentoinnista.

Yleisesti ottaen testaukseen panostamista pidettiin äärimmäisen hyvänä asiana, ja nyt katsottiin olevan hyvä tilaisuus olla etuajassa testauksessa ja luoda sen kautta lisää liiketoimintaa. Asiakkaiden koettiin tietävän ja ymmärtävän, että järjestelmissä on aina ohjelmistovirheitä ja kun ne on listattu ja korjattu ja testaus on läpinäkyvämpää, arvioitiin luottamuksen yritystä kohtaan kasvavan.

7.1.3 Johto

Johdon mielestä projekteissa on paljon niin hyviä kuin huonojakin esimerkkejä testauksen hoitamisesta. Osaan kriittisistä sovelluksista on tarkoituksellisesti varattu ja käytetty enemmän aikaa testaukseen kuin varsinaiseen toteutukseen. Yhdenmukaista prosessia testaukseen ei ole ollut. Kunnolla suoritetun testauksen uskottiin vähentävän merkittävästi vikojen määrää. Harmittavana tosiasiana pidettiin sitä, että testausta tehdään monesti kehittäjävetoisesti, jolloin on vaarana sokeutua omille virheilleen.

Testauksen dokumentointia on tehty jossain määrin, mutta mitään olemassa olevaa, yhtenäistä käytäntöä dokumentointiin ei ole ollut. Testausta ei ole dokumentoitu, mikäli asiakas ei ole sitä erikseen pyytänyt. Ja jos ja kun testaukseen liittyvien dokumenttien tarve tulee esiin, niin se pitää hinnoitella erikseen.

Johto näkee testauksessa olevan isoja eroja tiimeittäin ja projekteittain jo lähtökohdista alkaen. Järjestelmiä testataan hyvin eri tavoin ja myös asenne testaukseen vaihtelee tosi paljon. Koko organisaation laajuiset yhtenäiset käytännöt olisi saatava riittävälle tasolle ja niitä pitäisi pystyä tarvittaessa soveltamaan. Testaukseen tarvitaan erikseen paitsi testaajat myös käytännöt resurssienvaraukseen ja testaukseen. Käytäntöjä on tärkeää pystyä myös ylläpitämään ja kehittämään jatkuvasti.

Suurimpina haasteina pidettiin testauksen ja yhtenäisten käytäntöjen saamista mukaan projekteihin ja tiimeihin. Ei esimerkiksi tiedetä mitä kuuluisi tehdä, kuka tekee testitapaukset, kuka ylläpitää niitä, minkä verran testaukseen ja sen suunnitteluun varataan aikaa jne. Lisäksi asiakkaan on tiedettävä, mitä heiltä odotetaan testauksessa. Vahvuutena pidettiin näkemystä siitä, että jokainen yksilö varmasti kykenee tehokkaaseen testaukseen, kunhan testauksen merkitys osataan perustella oikein ja siihen varataan riittävästi aikaa.

Testauksessa on paljon eroja eri asiakkaiden ja projektien välillä. Jotkut asiakkaat ovat ymmärtäneet testauksen tärkeyden, jolloin aikatauluja ja työmääriä suunnitellessa voidaan varata hyvin aikaa testaukseen. Osa asiakkaista ei taas sisäistä testauksen tärkeyttä, vaan kyseenalaistavat kehittäjien taidot, mikäli testauksesta pitäisi maksaa. Tähän pitäisi kiinnittää huomiota jo sopimustasolla ottaen samalla ottaa kantaa siihen, miten asiakas haluaa projektin aikana testauksen tehtäväksi. Testauksessa on siis korjattavaa ja parannettavaa myös asiakastasolla.

Johto halusi testausprosessin olevan dokumentoitu ja toteutettu myös käytännössä. Tätä ei pidetty suoranaisena kilpailuetuna, mutta jos voidaan osoittaa testauksen olevan kunnossa, on se varmasti yksi luotettavuustekijä asiakkaan suuntaan.

7.2 Havainnoinnin tulokset

Seuraavaksi esitellään molemmista pilottiprojekteista osallistuvan havainnoinnin myötä saatuja kokemuksia ja tuloksia. Alakohdat 7.2.1 ja 7.2.2 sisältävät myös tiimien yhdessä toteamia, projektiin liittyviä onnistumisia ja kehityskohteita, jotka nousivat esiin keskustelemalla tiimin kesken projektin aikana niin aamupalavereissa kuin pyrähdyksen jälkiarvioinneissakin.

7.2.1 Pilottiprojekti A

Ensimmäinen Scrum-pilottiprojekti oli Notes-clientilla käytettävä tietokanta, jolla hallitaan materiaalitilausprosessia. Pyrähdyksien pituudeksi päätettiin keskustelun päätteeksi kaksi viikkoa eli kymmenen työpäivää. Pyrähdyksien pituus herätti keskustelua projektin suunnitteluvaiheessa, sillä koko projektin kesto oli vain kymmenen viikkoa.

Ensimmäisen viikon aikana kolmen hengen projektiryhmä sopi käytännöistä, joista osa oli uusia, ja niitä sovellettiin läpi projektin. Toinen projektin kehittäjistä ei ehtinyt käyttää pilottiprojektiin niin paljon tunteja kuin alun perin oli suunniteltu. Tämän arviointivirheen olisi mahdollisesti voinut välttää käyttämällä Knibergin (2006) mainitsemaa keskittymiskerrointa (*focus factor*), joka määrittää tiimin jäsenten kyvyn keskittyä kyseessä olevaan pyrähdykseen. Alhainen keskittymiskerroin tarkoittaa tiimin odottavan kohtaavansa monia työtä häiritseviä keskeytyksiä (Kniberg 2006). Ensimmäinen pyrähdys ei sisältänyt testausta, joskin pyrähdyksen tavoitteena saada ainoastaan testausympäristö pystyyn. Ensimmäisen pyrähdyksen myötä opittiin ja päätettiin, että jatkossa vaikeimpia toiminnallisuuksia ei jätetä toteutettavaksi pyrähdyksen loppupuolelle, vaan niiden toteutus aloitetaan heti, kun prioriteetti sen sallii. Lisäksi päätettiin jatkossa kysyä asiakkaalta epäselvistä toiminnallisuuksista lisätietoa mahdollisimman pian, sillä vastausten saaminen ensimmäisessä pyrähdyksessä kesti toisinaan kauan.

Päivittäiset palaverit antoivat testaajalle ensisijaisen tärkeää tietoa kehittäjien kokemista ongelmista ja heidän antamistaan vinkeistä, mihin testauksessa erityisesti kannattaisi kiinnittää huomiota. Myös kehittäjät kokivat aamupalaverit tehokkaaksi ja hyväksi käytännöksi.

Haasteeksi projektissa koettiin järjestelmässä käytetty tilausten elinkaarikaavio, jonka lopullista sisältöä asiakas ei osannut täydellisesti kuvata. Lisäksi yksi Scrumin tunnusmerkeistä, asiakkaan suorittama pyrähdyksiin otettavien toteu-

tuksien valinta ei toiminut, sillä asiakas ei halunnut ottaa toteutusjärjestykseen juurikaan kantaa. Tätä kylläkin sovellettiin onnistuneesti siten, että asiakas hyväksyi projektiryhmän tekemän listauksen asioista, jotka toteutukseen kannattaisi sisällyttää.

Pyrähdyksen aikana tehtävät tuotokset testattiin vasta pyrähdyksen päätyttyä, eli käytännössä seuraavan viikon alussa. Tätä voidaan pitää huonona siinä mielessä, että mahdollisten ongelmien löytyessä niiden korjauksia ei ehditty ottaa mukaan menossa olleeseen pyrähdykseen. Testaus suoritettiin pääsääntöisesti pyrähdysten lopussa, sillä kehittäjät kokivat pyrähdyksen keskellä suoritetun testauksen toteutusta häiritseväksi tekijäksi. Syynä tähän oli se, että kehitettävän järjestelmän ollessa testaajalle täysin uusi, oli testaajan jatkuvasti kysyttävä kehittäjiltä neuvoja. Puutteeksi projektissa voidaan laskea myös testaajan poisjääminen testaukseen kuluvien tuntien arvioinnista.

Projektin kannalta oli äärimmäisen tehokasta sijoittaa projektin jäsenet istumaan fyysisesti toistensa lähelle, sillä tällöin projektiin liittyvä informaatio siirtyi paremmin tiimin sisällä. Uudesta testityökalusta löydettiin puutteita ja parannusehdotuksia etenkin seurattavuuden suhteen, sillä sitä ei ollut suunniteltu ketteriä menetelmiä varten. Kehittäjät kokivat joutuvansa kuluttamaan liikaa aikaa testitapausten kirjoittamiseen.

Asiakkaan testauksessa löytyneiden virheiden hallinnassa oli parannettavaa, nyt siihen koettiin menevän liikaa aikaa. Testaukseen kulunut aika oli melko pitkä suhteessa projektin kokonaiskestoon. Syyn tähän uskottiin olevan lyhyissä pyrähdyksissä, joiden päätteeksi tuotos oli kokonaisuudessaan testattava toimivaksi. Regressiotestauksen automatisoinnilla olisi säästetty paljon aikaa.

7.2.2 Pilottiprojekti B

Kuusi henkilöä työllistäneen projekti B:n tavoitteena oli tehdä asiakkaalle asianhallintajärjestelmä. Lähtökohdat projektille olivat erilaiset projekti A:han

verrattuna. Asiakas tiedosti ja suostui Scrumin pilotointiin projektissa ja lähti itsekin innolla mukaan kokeilemaan Scrumia ja sen soveltuvuutta asiakkaan roolissa. Projektin tarkka työmäärä ei ollut sen alkuvaiheessa tiedossa, mutta arvio liikkui noin tuhannessa tunnissa.

Haasteeksi projektissa havaittiin sen aikaisessa vaiheessa regressiotestauksen suuri tarve, minkä ratkaisuksi kirjallisuus esittää testauksen automatisointia. Testauksen automatisoinnin organisointi ja käyttöönotto tähän, jo käynnissä olevaan projektiin koettiin kuitenkin mahdottomaksi toteuttaa. Automatisoidut testit olisivat auttaneet sekä testauksen nopeuden että toistettavuuden parantamisessa.

Kuten Crispin ja Gregory (2009, 32–33) ovat esittäneet, ketteriä menetelmiä käytettäessä testaajalle ei välttämättä ole tarjolla varsinaista testaustyötä jatkuvasti pyrähdysten aikana. Tästä syystä projektikohtaisesti on mietittävä, millä muilla tavoin testaajan resursseja voidaan käyttää järkevästi paremman lopputuloksen saamiseksi (Crispin & Gregory 2009, 32–33). Projekti B:ssä tämä asia huomattiin todeksi, ja täten sovittiinkin, että testaajan roolissa oleva henkilö vastaa varsinaisen testauksen lisäksi muun muassa testiympäristön ylläpidosta, IBM:n Team Studio Analyzer -työkalun käytöstä sekä tietokantojen toimittamisesta asiakkaalle pyrähdyksen päätyttyä. Tämän lisäksi asiakkaan toivoman listauksen pyrähdyksien uusista ominaisuuksista teki yleensä testaaja.

Koska projekti B oli projekti A:ta suurempi, muutosherkempi ja kompleksisempi, kohdattiin siinä myös enemmän haasteita. Kiireeltä ei vältytty esimerkiksi silloin, kun testaajalle näytettiin vihreää valoa testauksen aloittamiseen vasta pyrähdyksen viimeisen eli neljännen viikon puolivälissä. Kiire oli itsestäänselvyys, sillä tekemättä tässä vaiheessa oli siis vielä testaus, löydettyjen ohjelmistovirheiden korjaaminen, uudelleentestaus ja tietokantojen toimittaminen asiakkaalle.

Kehitettävien toiminnallisuuksien tulkinta oli hankalaa testaajan näkökulmasta, sillä monessa tapauksessa toiminnallisuuskuvausta ei ollut kirjoitettu yksiselkoisesti: mitään tarkempaa lisätietoa ei ollut saatavilla ilman, että toiminnallisuudesta kysyttiin lisätietoa kehittäjältä. Lisäksi moni toiminnallisuuskuvaus oli toiminnallisuuden testauksen aloittamiseen mennessä ehtinyt vanhentua. Vanhentuminen saattoi johtua esimerkiksi toiminnallisuuskuvauksen kirjoittamisen jälkeen asiakkaan kanssa pidetystä palaverista, jonne testaajaa ei aina kutsuttu mukaan.

Kehittäjien testaamisesta ei syntynyt projektin alkuvaiheilla mitään konkreettista dokumentaatiota, mihin luvattiin kehittäjien toimesta olevan parannusta luvassa jatkossa. Tuloksena oli sopimus siitä, että toiminnallisuuksien listauksien ja kuvauksien ylläpitoon tarkoitetun järjestelmän jokaisen toiminnallisuuden kommenttikenttään laitetaan testausohje. Ohjeessa kerrotaan, miten kehittäjä on varmistanut toteuttamansa toiminnallisuuden toimivuuden. Tätä kuvausta voidaan pitää luonteeltaan samantyylisenä kuin testaajan tekemää ohjelmistovirheraportin kuvausta – kuitenkin sillä erotuksella, että ensin mainitussa kerrotaan miten toiminnallisuus todetaan, kun taas jälkimmäinen kertoo, miten toiminnallisuuden tai komponentin sisältämä ohjelmistovirhe saadaan ilmenemään. Testauksen tehostamiseksi Scrum-valmentaja puuttui testaukseen neljännen pyrähdyksen päätteeksi muistuttamalla tiimiä, että kehittäjillä on edelleen vastuu toteutuksensa testaamisesta, ja jos testaajalle luvataan jokin päivämäärä jolloin testattavaa on, siitä on ehdottomasti pidettävä kiinni. Lisäksi testaaja koki olevansa liian vähän käytettävissä useassa projekti B:n pyrähdyksessä muiden projektien takia. Ratkaisuksi tähän päätettiin yrittää pystyisikö testaaja pyhittämään jatkossa projekti B:n kunkin pyrähdyksen viimeisen viikon kalenterista pelkästään projekti B:lle. Tämä oli perusteltua, sillä projektin pyrähdysten viimeiset viikot sisälsivät muun muassa testausta, dokumentointia ja tietokantojen toimittamista asiakkaalle.

Aamupalaverit koettiin tehokkaaksi käytännöksi, mutta niiden ajankohta vaihteli päivittäin. Tähän haluttiin parannusta, joten tiimin kesken sovittiin aamupalaverit pidettäviksi jatkossa tasan klo 9.30, jos ei toisin sovita. Syynä täsmällisiin aamupalavereihin oli aiemmat epätietoisuudet kellonajasta, ja monesti aamupalaverit jopa unohtuivat tai jäivät muuten vain pitämättä.

Joka pyrähdyksessä jäi toteuttamatta toiminnallisuuksia, jotka Scrumin käytännön mukaisesti siirrettiin tehtäväksi seuraavassa pyrähdyksessä. Suurimpaan osaan näistä oli selittävänä tekijänä epäonnistunut arviointi: toiminnallisuuksiin liittyi enemmän työtä ja selvitettäviä asioita, kuin alun perin oli osattu ottaa huomioon. Projektin edetessä alettiin huomata dokumentoinnin tärkeys, mutta sitä ehdittiin tehdä vasta projektin loppuvaiheilla.

Kun projektin jäseniltä kysyttiin mielipiteitä Scrumista pitkin projektia, pidettiin Scrumia järkevänä valintana pilottiprojekti B:n kaltaiseen isompaan projektiin, jossa on monta tekijää ja toteutettavat asiat muuttuvat pitkin projektia. Aamupalavereita pidettiin tehokkaana ja hyvänä käytäntönä. Testauksen kannalta pyrähdysten tuoma säännöllinen rytmi tekemiseen oli hyvä: ilman pyrähdyksiä testaaja saisi valmiita, testattavia tuotoksia todennäköisesti epäsäännöllisesti ja vähemmän. Projekti B:ssä testaus tehtiin pääasiallisesti pyrähdysten lopussa, sillä se koettiin järkevimmäksi tavaksi mahdollistaa pyrähdyksen ehjä toteutusjakso. Suurimpana yksittäisenä syynä tähän oli pyrähdysten aikana toteutettavien toiminnallisuuksien keskinäiset riippuvuudet, joiden takia toiminnallisuuksia ei välttämättä ollut testattavaksi.

Projekti B:n testaajasta tuntui useiden pyrähdysten aikana siltä, että kehittäjät unohtivat tai heitä ei ainakaan kiinnostanut miettiä osakokonaisuuksien toteutusjärjestystä siten, että testaus olisi voitu aloittaa mahdollisimman aikaisin. Pitkin pyrähdystä suoritettua testausta puolustaisi se tosiasia, että on sitä parempi, mitä aiemmin ohjelmistovirheet havaitaan ja korjataan. Toisaalta kaikki osakokonaisuudet on joka tapauksessa testattava myöhemmin. Asioista keskusteltiin tiimin kesken ja yhteisenä päätöksenä päätettiin antaa kehittäjille toteu-

tukseen kolmen viikon työrauha, josta mainitaan myös Scrumin periaatteissa. Testaus pysyisi siis edelleen pyrähdyksen lopussa suoritettavana työvaiheena.

Haasteeksi projekti B:ssä koettiin ainakin testaajan näkökulmasta kehittäjien ajoittainen laiskuus testata tuotoksiaan asiakkaan tuotantoympäristössä käytössä olevalla selaimella. Asiassa tapahtui kehitystä projektin myötä, kun siitä muistutettiin Scrum-valmentajan kanssa sinnikkäästi. Koska kehitettävässä ohjelmistossa tietyt komponentit ja osakokonaisuudet vaikuttivat olevan toistuvasti rikki, koki testaaja haastavaksi tehdä päätöksen siitä, milloin ja mistä löydöksestä tulisi tehdä ohjelmistovirheraportti kehittäjälle ja mistä ei. Ongelmaan saatiin selkeä parannus korostamalla riittävästi toteutusjärjestystä: löytyneet ohjelmistovirheet korjataan ja mahdollisesti edellisessä pyrähdyksessä keskenjääneet toiminnallisuudet toteutetaan aivan ensimmäiseksi. Eräässä aamupalaverissa keskusteltiin asiakkaan suorittaman testauksen tuloksista ja niiden saattamisesta kaikkien näkyville läpinäkyvyyden ja informaation kulun parantamiseksi. Asiaan tulikin parannus: projektin jäsenet saivat sähköpostitse tietoa asiakkaan löytämistä ohjelmistovirheistä, joita olisi kenties ollut hyvä käydä läpi koko projektiryhmän kesken.

Projekti B sisälsi paljon toiminnallisuuksia, joiden valmiiksi saaminen edellytti monen kehittäjän työpanosta: yksi tekee agentin, toinen toteuttaa tietokantaosuutta ja kolmas viimeistelee toiminnallisuuden graafista käyttöliittymää varten. Toiminnallisuudet sisältävä järjestelmä vaatii kuitenkin yhden toiminnallisuuden olevan vain yhden henkilön nimissä. Tällaisten monelle kehittäjälle kuuluvien toiminnallisuuksien havaittiin olevan keskimääräistä huonommin testattuja ja pitkään keskeneräisiä. Tiimi sopi käytännöksi vaihdella toiminnallisuuden vastuuhenkilöä järjestelmässä sitä mukaa, kun sen toteutusvuoro siirtyy henkilöltä toiselle. Vastuussa tällaisen toiminnallisuuden toimivuudesta, testauksesta ja testausohjeista on henkilö, joka on tehnyt toiminnallisuudelle viimeisenä toimenpiteitä.

Tiimi koki tarvitsevansa selkeitä pelisääntöjä toiminnallisuuksien vastuiden lisäksi myös määritelmiin valmiista toiminnallisuuksista. Täten sovittiin, että ohjelmistovirheen löytyessä valmiiksi saadusta toiminnallisuudesta avataan virheeseen liittyvä toiminnallisuuskuvaus sen sijaan, että kirjoitettaisiin uusi ohjelmistovirheraportti. Samoin toiminnallisuuskuvausten duplikaattien toivottiin sisältävän tietoa siitä, miten ja miltä osin kuvaukset ovat yhteneviä. Kehittäjiltä edellytettyjä testausohjeita näkyi toiminnallisuuksien kuvauskentässä edelleen melko vähän. Tämän johdosta projekti B:n Scrum-valmentaja esitti pyrähdyksen arviointipalaverin yhteydessä seuraavaa: jatkossa pyrähdysten suljettutilaiset toiminnallisuuskuvaukset avattaisiin testausohjeen puuttuessa uudelleen täydennettäviksi, vaikka varsinaisessa toiminnallisuudessa ei olisikaan puutteita tai virheitä.

Erään pyrähdyksen suunnittelupalaverissa todettiin asiakkaan priorisoimien toiminnallisuuksien toteutusjärjestyksen olevan haastava, sillä asiat ja toiminnallisuudet näyttivät olevan riippuvaisia toisistaan. Yhdeksi ratkaisuksi jatkoa ajatellen pohdittiin mahdollisuutta pilkkoa toiminnallisuudet vielä pienempiin osiin. Lisäksi todettiin pyrähdysten aikana projektille tehtyjen tuntien määrän olevan suurempi kuin toiminnallisuuksiin arvioidut ja varatut tunnit, mikä ei saa antaa väärää kuvaa etenemisnopeudesta. Kokonaistuntimäärää vääristi osaltaan se, että testausta varten pyrähdyksiin ei ollut liitetty työtehtävää, jonka kuvaus olisi siis ollut esimerkiksi ”manuaalinen testaus”.

Testauksen saamisessa suoritettavaksi tasaisesti pitkin pyrähdystä ei onnistuttu kovin hyvin. Jatkuva palautteen antaminen kehittäjille olisi kuitenkin ollut erittäin tärkeää. Tämän johdosta testaaja mietti mahdollisuutta kokeilla testitapausten kirjoittamista heti pyrähdyksen alussa tekemällä testaushahmotelmia sen hetkisten toiminnallisuuskuvausten perusteella. Liian yksityiskohtaisia testitapauksia ei kannata tehdä, koska toiminnallisuudet todennäköisesti muuttuvat jonkin verran: muutoksien myötä taustalogiikkaan tulee edelleen muutoksia, jotka vaikuttavat ainakin testaukseen. Testausta, ainakaan näkyvää sellaista, ei

vieläkään testaajan mielestä tehty koko projektin voimin tarpeeksi, minkä myötä testaus voisi helposti alkaa laahata kehityksestä jäljessä.

Projektin toteutustavat ja siihen kuuluvat osakokonaisuudet johtivat siihen, että monet asiat olivat keskenään riippuvuussuhteessa vaikuttaen toistensa toiminnallisuuksiin. Tästä johtuen ainakin testausnäkökulmasta katsottuna olisi kenties ollut havainnollistavaa luoda taulukko tai kaavio kuvaamaan asioiden riippuvuussuhteita. Testaajan kovasti odottamaa tilannetta, jossa pyrähdykseen valituilla osakokonaisuuksilla ei olisi ollut riippuvuussuhteita, ei toteutunut täysin missään vaiheessa projektia.

Testaukseen varatut tuntimäärät pyrähdyksittäin olivat tuotteen omistajan tekemiä karkeita arvioita. Testaajan parasta arviota testaukseen menevään aikaan ei yleensä kysytty, vaikka toisaalta sitä ei olisi voitukaan tietää kovin tarkasti. Asiaa olisi voitu kokeilla korjata tutustuttamalla testaaja pyrähdykseen valittuihin toteutettaviin toiminnallisuuksiin, vaikka niiden kuvausten perusteella olisi todennäköisesti ollut vaikeaa antaa tuntiarviota. Lisäksi niiden perusteella testausta ei olisi kannattanut suunnitella: testitapaukset olisivat todennäköisesti olleet niiden suoritushetkellä käyttökelvottomia, sillä toiminnallisuudet saattoivat muuttua vielä pyrähdyksen aikana. Suuntaa antavien testitapausten kirjoittaminen pyrähdyksen tavoitteita vasten osoittautui kuitenkin hyväksi ja toimivaksi käytännöksi.

Uutena käytäntönä oli siis lupa merkitä toiminnallisuus valmiiksi vasta sitten, kun sille oli kirjoitettu lyhyt testausohje. Tämä johti siihen, että erään pyrähdyksen viimeisen viikon alussa kaikista pyrähdyksen toiminnallisuuksista oli auki olevia toiminnallisuuksia yli puolet. Testaaja koki olonsa lievästi ulkopuoliseksi pyrähdyksen aikana: kun kehittäjiltä kyseltiin, olisiko testattavaa ollut, eivät kehittäjät käyttäneet tätä tilaisuutta hyväkseen.

Pyrähdyksien sisältöä suunniteltaessa tiimi uskoi projektissa ehdittävän toteuttaa enemmän asioita, kuin käytännössä ehdittiin lopulta tehdä. Tähän olisi pe-

riaatteessa pitänyt etsiä ja löytää ratkaisukeinoja. Asia tiedostettiin, mutta sitä ei kuitenkaan haluttu muuttaa, sillä tämän kaltaisen toiminnan, jossa toteutumattomat asiat puskuroitiin seuraavaan pyrähdykseen, nähtiin toimivan hyvin. Osittain kyseinen menettelytapa valittiin myös pakon sanelemana, sillä asiakaspalavereja pidettiin kiireestä johtuen melko harvoin. Mikäli tekeminen olisi pyrähdyksen loppuvaiheilla loppunut kesken, olisi tuotteen työlistan asioita ollut vaikeaa alkaa toteuttaa ilman asiakaspalaveria, jossa toteutuksesta keskusteltaisiin tarkemmalla tasolla: mitä tuotteen työlistassa olevat toiminnallisuudet todella tarkoittivat ja mitä niillä haluttiin saavuttaa.

7.3 Loppukyselyiden tulokset

Projekti A:n päätyttyä ja projekti B:n ollessa käynnissä, molempien pilottiprojektien jäsenille toteutettiin sähköpostitse kysely, jossa selvitettiin Scrumin käytön kokemuksia ja vaikutuksia projektien onnistumiseen, testaukseen ja lopputuloksiin. Kyselyt järjestettiin eri aikaan ja niiden toteuttamishetkellä projekti A oli päättynyt, kun taas projekti B:tä oli jäljellä reilut neljä viikkoa. Kyselyn kysymykset ovat liitteessä 3.

7.3.1 Projekti A

Projektin jäsenten mielestä projektin luonne ja kokonaiskesto olivat sopivia pilotointia varten. Muutoksia alkuperäiseen järjestelmän spesifikaatioon ei toteutuksen aikana tullut kovinkaan paljon, mikä koettiin Scrumin pilotointivaiheessa hyväksi asiaksi. Koska projektille oli riittävästi resursseja ja työmäärä sopivan iso, nähtiin järkeväksi jakaa projekti pyrähdyksiin. Pyrähdyksen pituudeksi valittu kaksi viikkoa puolestaan koettiin liian lyhyeksi: pyrähdysten alkuun ja loppuun liittyvät suunnittelut ja Scrumin käytännöt veivät tiimin jäsenten mielestä aikaa varsinaiselta kehitystyöltä, mikä korostui varsinkin lyhyissä pyrähdyksissä. Tiimin kehittäjät kokivat, että aikaa varsinaiselle toteutustyölle jäi liian vähän.

Asiakkaan koettiin suhtautuvan uusiin toimintatapoihin positiivisesti ja asiakas myös jousti tilanteissa, joissa Scrumin ennustettavuuden myötä osattiin kysellä mahdollisuutta jättää osa toiminnallisuuksista pois, mikäli kaikkea ei ehdittäisi tehdä. Pyrähdyksiin valittavien toteutettavien toiminnallisuuksien priorisointia asiakkaan toimesta jäätiin kaipaamaan useista pyynnöistä huolimatta. Selvimmin asiakkaalle Scrumin pilotointi näkyi viikkoraporteissa, työmäärien toteutumien näkemisessä aikaisemmassa vaiheessa sekä mahdollisuutena testata toteutettuja toiminnallisuuksia jo projektin aikana eikä vasta sen loppuvaiheessa. Edellä lueteltujen asioiden uskottiin vaikuttavan asiakkaan tyytyväisyyteen, sillä asiakkaan ei tarvinnut odotella kuukausia ilman minkäänlaisia konkreettisia tuloksia.

Projektin jäsenet kokivat noudattaneensa Scrum-käytäntöjä pääasiassa tunnollisesti. Kahden henkilön pitämät päivittäiset tilannepalaverit koettiin turhiksi, sillä kaikkien projektitiimin jäsenten työpisteet ovat vierekkäiset: tietoa pystyttiin jakamaan riittävästi joka tapauksessa päivän mittaan. Pyrähdysten päätöspalavereiden koettiin toistavan liikaa itseään ja niihin kaivattiin parempaa sisältöä. Kehitettävien toiminnallisuuksien käsittely ja läpivienti ei mennyt aina sovittujen pelisääntöjen mukaan, sillä tarkkaan sovittujen käytäntöjen noudattaminen koettiin joissain tilanteissa turhan jäykäksi. Yhtä selkeintä käytäntöä, jonka toimivuus tai tuomat hyödyt olisi koettu erityisen hyväksi, ei osattu nostaa esille.

Ennen projektia pidettiin sisäisiä Scrum-koulutuksia ja -perehdytyksiä, mutta pilottiprojekti A:n jäsenet eivät ehtineet osallistua kaikkiin koulutuksiin. Scrumia lähdettiin siis kokeilemaan ja opettelemaan melko vähillä opeilla ja kokemuksilla, mutta ensimmäisen pyrähdyksen jälkeen asioiden koettiin menevän selkeästi parempaan suuntaan. Organisaatiosta löytyvien kokeneiden Scrum-osaajien läsnäoloa olisi kuitenkin kaivattu esimerkiksi pyrähdysten suunnitteluja arviointipalavereissa.

Testauksen työmäärän arviointi koettiin vaikeammaksi Scrumia käytettäessä kuin vesiputousmallisissa projekteissa, joissa testaus sijoittuu projektin loppuun ja sille annetaan yleensä jokin kiinteä kokonaisarvio. Testaustyömäärien nähtiin kasvaneen, sillä testausta oli otettava mukaan aiempaa enemmän. Projektin sisäisen viestinnän nähtiin parantuneen selvästi. Työmäärien ja projektin tilanteen seuraaminen oli tiiviimpää ja seuraaminen toi hyviä tuloksia. Toisaalta päivittäisten tilannepalaverien tuomien hyötyjen uskottiin lisääntyvän, mikäli projektin tiimi olisi isompi. Muutosta kehittäjät kokivat myös työnsuunnittelussa: Scrumin myötä oli suunniteltava työtehtävänsä yhdestä kahteen päivää eteenpäin aiempaa tarkemmin.

Testausta pystyttiin kehittäjien mielestä delegoimaan kehittäjien ja testaajan välillä, koska kaikkien aikataulut ja tekemiset päivätasolla oli hyvin kaikkien tiedossa. Kaikkien tiimiin kuuluvien mielestä pyrähdyksen pituus oli liian lyhyt myös testauksen kannalta: mikäli se olisi ollut pidempi, olisi toteutuksen, testauksen ja sen myötä löydettävien ohjelmistovirheiden korjaus ollut hallitumpaa. Pohdittavaksi esitettiin myös sitä, mitä kaikkea kannattaa testata etenkin lyhyissä, kahden viikon pyrähdyksissä ja mitä ei: kehittäjien mielestä yksikkötestejä kannatti tehdä, mutta pyrähdysten välillä tapahtuva testaus aiheutti heidän mielestään osittain hämmennystä ja epätietoisuutta.

Scrumin käytön nähtiin parantaneen sovelluskehittäjien työtehtävien onnistunutta läpivientiä. Teknisten ongelmien esiintuonti päivittäisissä tilannepalavereissa koettiin tehokkaaksi menetelmäksi ratkoa ongelmia. Lisäksi aikataulujen nähtiin pitäneen paremmin. Vastuu palaverien valmisteluista ja pyrähdysten suunnitteluista oli tuotteen omistajalla, joka oli samalla myös yksi kehittäjistä., Ymmärrettävästi hän koki näihin menneen ylimääräistä aikaa, joka oli pois toteutustyöstä. Yhteydenpitoa asiakkaaseen oli enemmän, kuin mitä vesiputousmallia käytettäessä olisi todennäköisesti ollut.

Käytettyihin työkaluihin oltiin pääasiassa tyytyväisiä, mutta etenkin henkilöresursseja kuvaavan taulukon saamista automatisoiduksi kaivattiin. Scrumin

erilaisten kaavioiden uskottiin olevan hyviä, mutta niiden käytössä ja soveltamisessa nähtiin olevan parantamisen varaa ja opeteltavaa vielä projektin päätyttyäkin. Erityisen paljon parannusta toivottiin asiakkaalle toimitettavan tuntiraportin luomisen helpottamiseksi. Virheenhallintajärjestelmän käyttö koettiin melko vaivattomaksi ja hyödylliseksi, kunhan tarvittavien ominaisuuksien käyttö opittiin.

Kun projektin jäseniä pyydettiin listaamaan perusteluineen asioita, joita heidän mielestään olisi kannattanut tehdä toisin, pidettiin valittua pyrähdyksen pituutta eniten häirinneenä tekijänä: liian lyhyiden pyrähdysten koettiin rasittaneen projektia. Projektin suunnitteluun ja aloitukseen olisi myös voitu panostaa enemmän, jolloin alku olisi saatu hallitummaksi.

Pilottiprojektin jäsenet ilmaisivat olevansa valmiita osallistumaan Scrumia käyttäviin projekteihin myös jatkossa, mutta heidän mielestään päätös ohjelmistokehitysmenetelmän valinnasta kannattaa tehdä harkiten. Ratkaisevina tekijöinä valintaa tehdessä pidettiin asiakkaan yhteistyöhalukkuutta ja aktiivisuutta, projektin kokoa sekä kehittäjien luonteenpiirteitä ja kokemuksia. Kokonaisuutena ajatellen Scrumia luonnehdittiin hyväksi ja toimivaksi menetelmäksi.

7.3.2 Projekti B

Tiimin jäsenet pitivät Scrumia soveltuvana projekti B:n kaltaisiin isompiin projekteihin. Projektia pidettiin sopivana myös pilotoinnille sen keston takia. Scrumin uskottiin myös pystyneen osoittamaan hyötynsä paremmin isossa projektissa. Pyrähdyksen pituudestakin oltiin yksimielisiä: Neljän viikon pyrähdystä sopivana pitäneet perustelivat pyrähdyksessä jäävän aikaa muuhunkin testaamisen ja toimittamisen ohella. Täten niiden ei arvioitu korostuneen liikaa pyrähdyksen pituuden ollessa riittävä. Myös kehittäjät kokivat saaneen tarpeeksi aikaa kehitystyölle pyrähdyksissä.

Asiakkaan roolissa ja suhtautumisessa ketterään ohjelmistokehitysmenetelmään ja sen kokeiluun ei ilmennyt mitään odottamatonta tai negatiivista. Tähän arvioitiin vaikuttaneen projekti B:tä edeltäneen, samalle asiakkaalle toteutetun järjestelmän kehitysprojektin: toiminnallisuuksia kasattiin pikkuhiljaa asiakkaan kokeiltavaksi ja palavereita pidettiin sovituin väliajoin. Projekti B:ssä asiakas tuki Scrumin käyttöä, eikä tiimin mielestä Scrumia kohtaan ollut havaittavissa minkään näköistä muutosvastarintaa. Sen sijaan pyrähdysten pituudesta asiakkaan uskottiin olleen lyhyempien pyrähdysten kannalla, sillä asiakas olisi ajoittain halunnut nähdä ja kokeilla tuloksia useammin kuin neljän viikon välein. Asiakkaan kiinnostusta pyrähdysten mukanaan tuomiin lukuisiin palavereihin pidettiin puolestaan vähäisenä.

Tiimin jäsenten mielestä käytäntöjä noudatettiin riittävän hyvin, ja osasta käytännöistä luovuttiin sovitusti. Hyödyllisimpänä pidettiin päivittäistä tilannekatsausta: niiden avulla oli helppo nostaa esille pieniä asioita, joiden toteutukset olivat saattaneet jäädä muuten mahdollisesti roikkumaan. Lisäksi tiimin jäsenet saivat projektin toteutuksen tilanteen tehokkaasti selville aamupalaverien kautta. Etätöitä tekevän tiimin jäsenen mukaan ottamista päivittäiseen tilannekatsaukseen tosin kaivattiin. Myös iteratiivisuuden tuomaa toimitusvarmuutta projektiin pidettiin hyvänä asiana. Yleisesti ottaen käytäntöjä pidettiin hyvinä, mutta niille uskottiin olevan projektikohtaista soveltamistarvetta. Projekti B:ssä soveltamista ja jatkuvaa käytäntöjen parantamista nähtiin tehdyn jälkiarvioinneissa jossakin määrin.

Scrumin käyttöön liittyvien perehdytysten ja koulutusten määrät ja niihin liittyvät mielipiteet vaihtelivat projektitiimin jäsenten välillä jonkin verran. Osa tiimin jäsenistä piti muutamaa sisäistä koulutusta ja toisesta tiimistä saatua konsultointiapua riittävänä, kun taas osa ihmisistä olisi kaivannut pilottiprojektissa enemmän tukea Scrumista kokemusta omaavalta kollegalta.

Pääsääntöisesti Scrumin ei nähty tuovan muutosta työmäärän arviointiin. Kyselyyn vastanneista löytyi kuitenkin henkilö, joka totesi työmäärien ja tehtävien

priorisoinnin olevan helpompaa: Scrumia käytettäessä koettiin nähtävän selkeästi, mitä tehtäviä kullekin projektiryhmän jäsenelle on annettu. Myöskään priorisointia ei kaikkien mielestä nähty tehdyn täysin Scrumin mukaisesti pisteyttämällä kaikki toiminnallisuudet ja asiat, vaan katsomalla laajemmista kokonaisuuksista ne asiat, jotka on saatava valmiiksi seuraavaksi. Mahdollisuuden siirtää toiminnallisuuksia seuraavaan pyrähdykseen osan mielestä helpotti priorisointia. Osasyynä priorisoinnin paranemiseen pidettiin myös uusia, parantuneita työkaluja.

Viestinnän arvioitiin sekä tehostuneen että lisääntyneen Scrumin palaverikäytäntöjen myötä. Muutaman vastaajan mielestä projektin tilanteen seurantaa auttoivat työkalujen erilaiset kaaviot, jotka perustuivat muun muassa pyrähdyksiin valittujen toiminnallisuuksien valmistumisen kuvaamiseen. Osan mielestä kaaviot ja palaverit eivät kerro projektin todellisesta tilanteesta mitään: status saadaan sen sijaan selville ottamalla huomioon aikataulu, jäljellä oleva budjetti ja se, kuinka lähellä ollaan projektin tavoitetta, eli valmistuuko sovelluksesta ajoissa sellainen versio, joka voidaan siirtää tuotantoympäristöön.

Scrumin nähtiin tuovan testausta paremmin esiin ja kiinnittävän ohjelmistokehittäjän huomion paremmin toteutuksen toimivuuteen jo toteutusvaiheessa. Osa tiimin jäsenistä olisi kaivannut projektiin varsinaisia testausviikkoja, jolloin kehitettävän järjestelmän toimivuutta katselmoitaisiin kunnolla eikä uusien toiminnallisuuksien toteutusta tehtäisi ollenkaan. Projekti B:ssä sovitut testauskäytänteet ja määritelmät valmiista toiminnallisuudesta pakottivat kehittäjät tekemään toiminnallisuudet kerralla valmiiksi. Osan mielestä projekteissa ei voi koskaan olla liikaa testausta ja asiakkaan saamien väliversioiden laadun uskottiinkin parantuneen lisääntyneen testauksen myötä.

Kysyttäessä tiimiltä Scrumin vaikutuksista sovelluskehittäjien työtehtävien onnistuneeseen läpivientiin, todettiin seuraavaa: pyrähdysten aikataulutus ja ohjelmiston kelpoisuusvaatimus tuotantoon vietäväksi aiheutti kehittäjille ainakin jossakin määrin paineita mutta myös selkiytti työn tekemistä. Monella tiimin

jäsenellä ei ollut mielipidettä käytetyistä työkaluista, joten he eivät olleet todennäköisesti käyttäneet tai hyödyntäneet taulukoita ja pyrähdysten suunnitteluun tarkoitettuja ohjelmia. Pyrähdyksen etenemistä kuvaavia taulukoita pidettiin enemmänkin tuotteen omistajan työkaluna.

Tiimin mielestä projekti B:n Scrum-pilotoinnissa olisi voitu tehdä muutamia asioita toisin. Kehittäjät esittivät, että he olisivat saaneet enemmän irti Scrumista, mikäli heitä olisi koulutettu enemmän sen käyttöön. Asiakaspalavereihin olisi pitänyt osallistua koko tiimi, että tietoa olisi saatu jaettua enemmän ja tehokkaammin. Toiminnallisuuksien tarkempaa suunnittelua ennen niiden toteutusta kaivattiin monen ihmisen taholta, sillä asioita koettiin välillä toteutettavan vähän miten sattuu, eikä parhaiden käytänteiden mukaisesti. Kun asioita pysähdyttäisiin miettimään, voitaisiin tulokseksi saada yleispäteviä ratkaisuja, joiden sovellettavuus ja käyttö eivät jäisi pelkästään kuluvaan pyrähdykseen. Esille nostettiin uudestaan myös toive pitää kiinni siitä, ettei pyrähdyksen viimeisellä, eli testausviikolla kehitettäisi enää uusia toiminnallisuuksia, vaan keskityttäisiin täysipainoisesti testaukseen.

Kaikki kyselyyn vastanneet ilmoittivat olevansa valmiita käyttämään Scrumia myös jatkossa, joskaan sen käyttöä ei nähty järkeväksi pienissä projekteissa. Yksi vastaajista ei pitänyt Scrumin ideasta, jossa tiimi vastaa tuotoksesta: vastaaja oli sitä mieltä, että projekteissa kuuluisi edelleen olla eri osa-alueille omat vastuuhenkilönsä. Vastauksissa uskottiin myös asiakkaan saaneen aidosti positiivisen kuvan Scrumista.

7.4 Haastattelut vs. havainnot vs. loppukyselyt

Kun haastatteluja, loppukyselyjä ja tutkielman tekijän suorittamaa havainnointia verrataan keskenään, tunnistetaan sekä monia yhteneväisyyksiä että eriäväisyyksiä niin projektien kuin käytetyn tutkimusmenetelmänkin tulosten suhteen. Seuraavaksi perehdytään näihin havaittuihin ristiriitoihin, keskittyen erityisesti

mahdollisiin pilotoinnin myötä toteutuneisiin parannuksiin projektin läpiviemisessä tai testauksessa.

Alkuhaastattelujen perusteella testaus kaipasi kehittämistä niin toimeksiantajaorganisaation johdon, tiiminvetäjien kuin tiiminkin mielestä. Testaukseen kaivattiin ennen kaikkea yhtenäisiä käytäntöjä ja pelisääntöjä. Tarve testaukseen tunnistettiin, mutta kehittäjien asenteessa sitä kohtaan oli parantamisen varaa. Vesiputousmallia ohjelmistokehitysmenetelmänä käyttäneissä projekteissa testaus oli yhtenä viimeisimmistä suoritettavista kokonaisuuksista. Jos projektissa tuli sen loppuvaiheilla kiire, oli testaus se vaihe, josta kurottiin aikaa pois. Kehittäjät esittivät myös testauksen automatisoinnin selvittämistä. Kaikki kolme vastaajaryhmää pitivät testauksen haasteena myös dokumentoinnin laadun vaihtelevuutta ja sitä kautta muodostunutta hiljaista tietoa.

Projekti A:ssa ja projekti B:ssä suoritettu havainnointi vahvisti sen, että kehittäjien asennoituminen testauksen tärkeyteen ja sen varsinaiseen suorittamiseen vaihtelee. Scrumin käytön myötä testaukseen alettiin kuitenkin panostaa muun muassa kirjoittamalla testitapauksia sekä varaamalla sen tekemiseen aikaa. Etenkin projekti B:ssä testauksen automatisoinnin tärkeys sen puuttuessa kokonaan huomattiin projektin edetessä, kun testattavia toiminnallisuuksia tuli jatkuvasti huomattavan paljon lisää. Projekti A olisi luultavasti saatu hoidettua melko hyvin tuloksin myös vanhan, vesiputousmallisen ohjelmistokehitysmenetelmän mukaisesti. Sen sijaan projekti B:tä olisi ollut mahdotonta suunnitella etukäteen niin hyvin, että lopullinen toteutus näiden suunnitelmien ja asiakkaan projektin alussa antamien vaatimusten perusteella olisi ollut juuri sitä, mitä asiakas oikeasti halusi.

Havainnoinnin perusteella niin testaaja kuin kehittäjätkin kokivat molemmissa projekteissa päivittäistä tilannepalaveria erinomaisena käytäntönä jakaa projektiin liittyvää tietoa nopeasti ja tehokkaasti tiimille. Tämän ketterän käytännön yhdessä pilotteja varten toteutetun työpisteiden vaihdon kanssa voidaan todeta ratkaisevan tai paikkaavan ainakin osaltaan alkuhaastatteluissa mainittua do-

kumentaation puutteellisuutta ja hiljaisen tiedon olemassaoloa. Myös alkuhaastatteluissa mainittu testauksen vähäinen määrä nähtiin havainnoinnin perusteella saatavan kuriin pilottiprojekteissa, sillä Scrumin myötä sitä ei voitu jättää tekemättä. Pilottiprojektien myötä opittiin myös antamaan testaajalle muitakin projektiin liittyviä työtehtäviä varsinaisen testaustyön lisäksi.

Tiimin itseohjautuvuutta ja ketterien käytäntöjen soveltamista oli havaittavissa enemmän projekti B:ssä. Tähän johtivat todennäköisesti projekti B:n suurempi tiimin koko ja suurempi määrä haasteita, joihin oli tärkeää löytää ratkaisut. Lisäksi projekti B:n loppuvaiheilla myös kehittäjät alkoivat ymmärtää testauksen merkitystä.

Loppukyselyistä ilmenee etenkin projekti B:n tiimin positiivinen asennemuutos Scrumia kohtaan. Havainnoinnin alkuvaiheilla huomattu skeptisyys muun muassa päivittäisiin tilannepalavereihin oli kadonnut täysin – kyseistä käytäntöä pidettiin jopa yhtenä parhaista käytänteistä. Loppukyselyistä saadut vastaukset sekä pyrähdyksen pituudesta että järkevästä projektin koosta ovat yhteneviä havainnoinnin tulosten kanssa: Kahden viikon pyrähdyksissä varsinaista kehittäjien toteutustyötä ei ehditty tekemään kehittäjien mielestä tarpeeksi. Lisäksi Scrumin soveltuvuutta pienehköihin, tässä tapauksessa projekti A:n kaltaisiin projekteihin kannattaa jatkossa miettiä tarkkaan.

Molempien pilottiprojektien henkilövalinnoissa onnistuttiin hyvin. Tiimit olivat heterogeenisiä sisältäen sekä kokeneita että uransa alkuvaiheilla olevia kehittäjiä. Tällainen tiimin kokoonpano Scrumin pilotoinnissa lienee parempi valinta kuin tiimi, jossa on pelkkiä seniorikehittäjiä tai juuri valmistuneita, uraansa aloittavia kehittäjiä. Toisaalta Scrumissa tärkeänä pidettyyn asiaan, tiimin itseohjautuvuuteen, on kehittäjien kokemuksella tuskin kuitenkaan kovin paljon merkitystä, vaan tärkeämpiä ovat asenne ja sosiaaliset taidot. Loppukyselyistä ilmeni myös monen kehittäjän kokeneen tarvetta saada enemmän koulutusta Scrumin käyttöön ja soveltamiseen ennen pilotointia. Tämä heijastui myös havainnointituloksissa, sillä ilmiselvästi kaikki pilotointiin osallistuneet eivät

ymmärtäneet tai edes tiedostaneet kaikkien käytäntöjen käyttöönoton perimmäisiä syitä, joita ovat ketterien menetelmien arvot ja periaatteet.

7.5 Vastaus tutkimusongelmaan

Tutkimuksen tavoitteena oli selvittää mitä haasteita ja muutostoimenpiteitä ohjelmistotestaukseen on odotettavissa, kun projekteissa tapahtuvaa ohjelmistokehitystä aletaan tehdä perinteisten ohjelmistokehitysmenetelmien sijaan Scrumia käyttäen. Tutkimusongelma oli seuraava:

- Miten ohjelmistotestausta tehdään Scrumissa?

Jotta edellä esitettyyn kysymykseen voidaan vastata, on hyvä tarkastella myös varsinaista muutosta Scrumin käyttöön. Ketterien menetelmien käyttöönotto ja soveltamisen oppiminen on pitkä prosessi. Muutos ketteriin menetelmiin ei pääty onnistuneeseen esimerkiksi vesiputousmallista Scrumiin siirtymiseen, vaan muuttuu saatujen kokemusten avulla jatkuvaksi parantamiseksi ja käytäntöjen muokkaamiseksi. Muutokseen vaikuttavat muutosvastarinta, projektin luonne, ketterän menetelmän valinta, ajankohta, tiimin jäsenet sekä käytettävä teknologia. Näitä kaikkia on siis pohdittava ja suunniteltava tarkkaan ennen käyttöönottoa tai pilotointia. Ketteryys edellyttää pitkällä tähtäimellä ammattitaitoisia, motivoituneita tiimejä ja työntekijöitä sekä tehokasta ja jatkuvaa kommunikointia tiimien sisällä, niiden välillä ja asiakkaiden kanssa. Hyvin hoidettu ketterän menetelmän käyttöönotto parantaa myös testauksen laatua, sillä tiimi ymmärtää testauksen merkityksen ja siitä huolehtimisen kuuluvan kaikille.

Käyttöönotossa ovat ratkaisevissa asemissa organisaatiokulttuuri ja asenne. Mikäli työntekijät eivät koe saavansa uusista käytännöistä hyötyä ja etuja, tai käytäntöjen koetaan uhkaavan työntekijää jollakin tavoin, tulee ketterän menetelmän täysimittaisessa soveltamisessa olemaan paljon haasteita. Näiden lisäksi avainasemassa on myös asiakassuhteiden, -kommunikoinnin ja asiakkaan toi-

minta- ja ajatusmallien saattaminen toimeksiantajan käyttämää ketterää menetelmää vastaavaksi. Tässä piilee toisaalta vaarana se, että asiakas saa väärän kuvan toimintamalleista ja kuvittelee voivansa muuttaa mieltymyksiään kehitettävästä järjestelmästä mielin määrin.

Ketterien menetelmien käytäntöjen perimmäinen tarkoitus kannattaa selvittää Agile Manifeston periaatteista ja arvoista. Tästä esimerkkinä voidaan mainita päivittäiset tilannepalaverit: niiden avulla voidaan toteuttaa useita ketterien menetelmien periaatteita ja arvoja, sillä kasvotusten tapahtuva, lyhytkestoinen keskustelu nykytilanteesta edesauttaa niin toimivan ohjelmiston tuottamista, kykyä vastata muutoksiin kuin tiimin itseohjautuvuuttakin. Tiimin tulee ottaa käytännöt käyttöön sen takia, että niiden merkitys ja perimmäinen syy, ketterien menetelmien arvojen toteutuminen, saadaan toteutettua käytäntöjen avulla. On vaarallista alkaa käyttää erilaisia käytäntöjä pelkästään sen takia, että niitä käyttävät muutkin. Tällöin ei välttämättä tiedetä, miksi asioita tehdään tietyllä tavalla.

Ketterän menetelmän käyttöönotossa on oltava huolellinen siinä, mitä muutetaan: sopeutujana on tiimi toimintatapoineen eikä esimerkiksi pyrähdys, jonka pituutta vaihdellaan tilannekohtaisesti projektin aikana. Vastaavasti testauksen on pääsääntöisesti mukauduttava ja sulauduttava projekteihin sen sijaan, että projektit eläisivät tai muuttuisivat testauksen takia. Asia ei ole kuitenkaan täysi yksiselitteinen, sillä testauksen saaminen tehokkaaksi voi edellyttää esimerkiksi toteutusaikataulujen hienosäätöä.

Projekti B:n useassa pyrähdyksessä kävi juuri niin, kuin Scrumin käyttöönoton alkuvaiheilla on yleistä: tiimi uskoi kykenevänsä tekemään asioita enemmän kuin käytännössä ehti. Pyrähdyksen suunnittelussa on oltava realisti ja varattava muun muassa löytyneiden ohjelmistovirheiden korjauksille aikaa reilusti.

Pilottiprojekteista saatujen kokemusten perusteella yksi merkittävimmistä Scrumin tuomista positiivisista muutoksista testaukseen oli testauksen pakot-

taminen mukaan projektiin. Vaikka testausta ei onnistuttu täysimittaisesti hajauttamaan suoritettavaksi pitkin pyrähdystä, auttoi pyrähdysten päätteeksi tehty järjestelmätestaus saamaan säännöllisin väliajoin tietoa kehitettävän järjestelmän toimivuudesta. Lisäksi pilottiprojekteista saatiin lukuisia arvokkaita havaintoja: testauksen automatisoinnin tärkeys, järkevän toteutusjärjestyksen mahdollistaminen tehokkaan suunnittelun avulla ja kommunikoinnin merkitys toiminnallisuuksien syvimmän olemuksen selvittämiseksi.

Scrum ei itse määrittele, kuinka ohjelmistoprojektin toteutus tai testaus tulisi suorittaa. Olisi kuitenkin virheellistä sanoa, että Scrumin viesti testauksen suhteen olisi ”testaa miten haluat, pääasia että testaat” tai ettei Scrumin käyttö vaikuttaisi testaukseen. Viimeiseksi mainittu voidaan perustella muun muassa seuraavilla asioilla: testauksen on mukauduttava projektin pyrähdysten pituuteen, testausta on pystyttävä automatisoimaan ja testaajalta vaaditaan aiempaa enemmän jatkuvaa viestintää muiden projektin jäsenten kanssa. Lisäksi projektissa käytettävällä teknologialla on vaikutusta esimerkiksi edellä mainittujen asioiden toteuttamiseksi: erityisesti Notes- ja Domino-ympäristössä yksikkötestauksen automatisointi on haastavaa. Toimeksiantajaorganisaatiossa tehtyjen selvitysten mukaan Notes-sovellusten yksikkötestauksen automatisointiin ei ole kuitenkaan tarjolla järkevää työkalua tai menetelmää.

Scrumissa tapahtuvalla testauksella ja sitä tekevällä testaajalla vaikuttaisi olevan perinteisiä menetelmiin verrattuna enemmän vaikutusvaltaa niin ohjelmistokehitysprosessiin kuin kehitettävään lopputuotteeseen. Perinteisten ohjelmistokehitysmenetelmien testauksen tarkoituksena on löytää virheitä käyttämällä ohjelmaa keskittyneesti ja määrätietoisesti. Scrumissa testaus on luonteeltaan rakentavaa, eli laatu rakennetaan tuotteen sisään jatkuvalla testauksella ja integroinnilla ja niiden yhdessä synnyttämällä palautteella kehitettävästä tuotteesta. Scrumin testaus edellyttää koko tiimin osallistumista testaukseen ja testauksen automatisointia monella eri tasolla. Verrattuna vesiputousmallia ohjelmistomenetelmänä käyttävään projektiin, Scrumissa testaajan on tehtävä

enemmän töitä pitääkseen yllä tietämyksensä kehitettävästä järjestelmästä ja siihen kohdistuvista muutoksista jatkuvalla tiedonkeruulla, yhteydenpidolla ja tutkivalla testauksella. Testausta dokumentoidaan ja mitataan tarpeen mukaan ja tilannekohtaisesti. Testausta on automatisoitava siinä määrin, kuin teknologia sitä tukee, ja niiltä osin joita tiedetään tarvittavan testata vielä myöhemmin. Lisäksi pyrähdyksen testauksen kokonaissuunnittelu kannattaa aloittaa heti sen alussa.

Vesiputousmallia käytettäessä testisuunnitelmien, –tapausten ja –raporttien kirjoittamista pidetään monesti itsestäänselvyytenä. Scrumissa tällaiseen ei aina ole välttämättä aikaa tai mielekkyyttä, sillä edellä mainitun dokumentaation arvo voi käydä mitättömäksi, mikäli sitä ei kyetä pitämään ajan tasalla. Tästä huolimatta voi olla järkevää dokumentoida suoritusohjeineen sellaiset testitapaukset, joita tiedetään suoritettavan useasti ja mahdollisesti monien eri henkilöiden toimesta.

Pilotoinnin myötä voidaan todeta, että asiakas ei luonnollisestikaan odota maksavansa ohjelmistovirheistä, vaan olettaa saavansa valmiin ja täysin toimivan ohjelmiston. Tässä tutkielmassa ei etsitty vastausta siihen, onko tähän syynä puhtaasti Scrum ja sen läpinäkyvyys. Niin asiakas kuin toimittaja haluavat käyttää ohjelmistokehitysmenetelmänä Scrumia, johon kuuluu muun muassa muutosten vastaanottaminen pitkin kehitysvaihetta ja asiakkaan toimesta tapahtuva kehitettävien asioiden toteutusjärjestyksen priorisointi. Niin yleisesti ohjelmistokehitysprojekteissa kuin etenkin tällaisen toimintatavan vallitessa asiakkaan voitaneen olettaa ymmärtävän ja hyväksyvän, että jatkuvan muutoksen myötä ohjelmistovirheitä ja väärinymmärryksiä voi esiintyä. On jokseenkin ristiriitaista, että tästä huolimatta ohjelmistovirheitä ei uskalleta katsoa silmiin ja niiden olemassaoloa ei haluta hyväksyä.

Kirjallisuuden perusteella tutkiva testaus ja testauksen automatisointi ovat tärkeimpiä testauskäytänteitä Scrumissa. Tätä tukevat myös tutkielmassa analysoitujen pilottiprojektien, etenkin projekti B:n havainnot ja kokemukset. Tutki-

van testauksen myötä testaajan aikaa säästyy juuri siihen mihin pitääkin, eli testattavan ohjelmiston läpikäyntiin ja tutustumiseen. Tutkiva testaus ei ole helppoa: testaajan on sekä ymmärrettävä, mitkä riskit ovat tärkeitä tiimille, että osattava fokusoida testauksen kattavuus ja päätellä mitä milloinkin on testattavissa. Helpotusta näihin haasteisiin tuovat osaltaan niin Scrumin kuin myös muiden ketterien menetelmien arvot ja periaatteet, muun muassa jatkuva kommunikointi ihmisten kesken.

Kuten aiemmin luvussa 2.2 todettiin, Beizerin (1990, 4) mukaan ohjelmistokehitysmenetelmissä ohjelmistovirhe osoitetaan odotetun lopputuloksen poikkeamana: poikkeaman havainnointiin tarvitaan olemassa oleva dokumentaatio, josta voidaan määritellä odotetut testien suoritusvaiheet odotettavissa olevine seuraamuksineen. Tämä ei toteudu ketterissä menetelmissä, sillä testaajalla ei ole aina mahdollisuutta saada ajan tasalla olevaa dokumentaatiota testauksen tueksi.

Toimiva yhteistyö eli kommunikointi tiimin sisällä on yksi tärkeimmistä lähtökohdista tukemaan ketterää testausta. Tätä pilottiprojekteissa on noudatettukin kiitettävästi: kehittäjät ovat kommunikoineet toistensa kanssa tiiviisti ja testaaja on mennyt testaamaan kehittäjien keskuuteen, tavoitteenaan keskustelun vaivattomuus. Lisäksi kommunikointia on helpottanut organisaatiossa käytössä oleva pikaviestinohjelma Sametime.

Manuaalinen testaus on hidasta, ja siksi palaute ohjelmistovirheestä saadaan toisinaan liian pitkän ajan päästä siitä hetkestä, jona ongelmien syntymiseen johtaneet koodimuutokset tehtiin. Tämän takia voi olla vaikeampaa paikallistaa ohjelmistovirheen perimmäistä syytä, joten korjaaminenkin voi kestää kauan. Tämä kaikki yhdessä sotii ketterien menetelmien ideologiaa vastaan, jossa saadaan jatkuvaa palautetta ja tietoa ongelmista ja kehitettävän järjestelmän sen hetkisestä tilanteesta.

Tutkivaa testausta suorittavan henkilön on keskityttävä parantamaan kykyä olla sinut ja keskustella testauksen kohteena olevan ohjelmiston kanssa. testaajan on tunnettava testattava ohjelmisto – ilman ymmärrystä ohjelmiston käyttö- ja liiketoimintatarkoituksesta sekä teknologiasta, jolla ohjelmisto on rakennettu, on testaus harvoin tehokasta ja tuloksekasta. Tutkiva testaus on helposti hallitsematonta ja sen läpinäkyvyys on huono, mutta ratkaisuna tähän voidaan pitää Bachin (2000) sessiopohjaista testauksen hallintaa.

Niin kuin kaikki muukin testausmenetelmät, myös tutkiva testaus sisältää heikkouksia. Samalla kun se tarjoaa hyvän mahdollisuuden täydentää ennalta suunniteltujen testitapausten jättämiä aukkoja, on kattavuus myös yksi tutkivan testauksen ongelmista. Kun testauksen kohteena oleva järjestelmä täydentyy uusilla tai muuttuvilla ominaisuuksilla jopa päivittäin, tulee järjestelmästä testauksen liikkuva kohde, johon on vaikeaa osua ja josta vaikeaa tarrata kiinni. Haasteena ketterissä menetelmissä testaajan kannalta ovat tilanteet, jossa loppukäyttäjälle tuleva ohjelmiston tietty toiminnallisuus ei ole täysin testaajan tiedossa, jolloin voi olla erittäin vaikeaa suunnitella tehokkaita ja kompleksisia testausskenaarioita. Tämä voi johtaa pelkkään suunnittelemattomaan, tutkivaan testaukseen jonka tueksi on kylläkin mahdollista käyttää ainakin aiemmin mainittua Bachin esittelemää sessiopohjaista testauksenhallintaa.

Kirjallisuus ei anna yksiselitteistä vastausta siihen, kuinka testaus tulisi hoitaa Scrum-projektissa. Yksi vastaus voisi yksinkertaisesti olla se, että ketterä testaus on testausta, joka noudattaa Agile Manifestoa, mukautuu valittuun ketterään menetelmään ja joka palvelee projektin ohjelmistokehitystä pitäen sitä testauksen asiakkaana ja tuottaen jatkuvaa informaatiota ohjelmiston laadusta. Yksi pilotoinnin myötä esiinnoussut havainto on, että testaaja sitoutuu enemmän projektiin ja saa täten uusia rooleja tai ainakin työtehtäviä varsinaisen testaustyön lisäksi, halusipa tai ei.

Onnistunut laadunvarmistuksen ja sen sisältämän testauksen menestyksekäs mukaan saaminen ketteriin menetelmiin vaatii myös laadunvarmistuksen roo-

lin ja merkityksen ymmärtämistä. Tuotteen laatua ei tulisi arvioida ja mitata yksinomaan löydettyjen ohjelmistovirheiden tai kirjoitettujen testitapausten lukumäärän perusteella. Pikemminkin laadunvarmistuksen tarkoituksena on ymmärtää ja mitata riskejä, ja lisäksi laatu rakennetaan kehitettävän tuotteen tai ohjelmiston sisään jo toteutusvaiheessa jatkuvan testauksen ja palautteenantamisen avulla.

Pilottiprojekti B:n myötä tehtiin myös seuraava havainto: jos kuluvan pyrähdyksen puolivälissä osakokonaisuudet on toteutettu sekä jo kertaalleen testattuja, on kaikki uudet toiminnallisuudet silti testattava uudelleen aivan pyrähdyksen päätteeksi. Testaajalla on siis enemmän työtä verrattuna vesiputousmalliin, jossa kaikki testaus sijoittuu projektin loppuun, jolloin helpommin vältytään kahteen kertaan tehdyltä testaukselta. Etuna vaiheittaisessa testaamisessa on nopeampi ohjelmistovirheiden kiinnisaaminen ja korjaaminen.

8 POHDINTA JA JOHTOPÄÄTÖKSET

Toimeksiantajaorganisaation Notes-tiimissä Scrumin käyttöönotossa on onnistuttu tähän mennessä keskinkertaisesti. Pilottiprojektien ei voida sanoa käyttäneen Scrum-viitekehystä kovin tiukasti kaikkine käytäntöineen, mikä oli ensimmäistä kertaa Scrumia sovellettaessa odotettavissakin. Pilottiprojektien perusteella Scrum osoittautui toimivaksi ja muokattavissa olevaksi menetelmäksi. Scrumin käyttöönoton kynnys koettiin lopulta pieneksi alussa ilmenneen muutosvastarinnan laantumisen jälkeen. Syynä tähän ovat todennäköisesti Scrumista puuttuvat tarkat insinöörikäytännöt, jotka sanelisivat liian tarkasti sen, kuinka toteutuksessa tulee menetellä. Kääntöpuolena vapaudessa toimia halutulla tavalla voidaan nähdä mahdollisuus maskeerata Scrum vanhan prosessin päälle ilman, että entistä toimintamallia todella muutetaan juurikaan. Tämä voi pahimmillaan johtaa Scrumin hylkäämiseen epätoimivana menetelmänä, vaikka todellisuudessa sitä ei ole sovellettu kunnolla vielä ollenkaan. Tällaista, vanhan mallin ehostamista Scrumilla nähtiin tapahtuvan ainakin pyrähdysten luonteessa, sillä etenkin projekti B:n pyrähdykset olivat ajoittain luonteeltaan vesiputousmallin mukaisia, itsenäisiä kehityssyklejä.

Notes-tiimin tahtotila on ollut systematisoida projektien toimintamalleja. Yhtenä ratkaisuvaihtoehtona tähän pidettiin Scrumia ja se vaikuttaisikin projekti B:stä saatujen kokemusten nojalla toimivalta ratkaisulta tiimin suuriin, yli tuhannen tunnin projekteihin. Vastaavasti pieniin projekteihin tai ylläpitoprojekteihin ei Scrum ole välttämättä oikea valinta ainakaan Notes-tiimissä.

Yhteistä molemmille pilottiprojekteille oli tuotteen omistajan roolin sattuminen henkilölle, joka toimi projektissa samalla myös aktiivisesti sovelluskehittäjän roolissa. Jatkoa ajatellen tuotteen omistajan olisi kenties hyvä pystyä keskittymään enemmän rooliinsa, sillä se sisältää paljon työtehtäviä, joiden hoitamista toiminnallisuuksien kehitystehtävät häiritsevät. Ongelmaan on kuitenkaan vai-

keaa löytää toimivaa ratkaisua muun muassa projektien luonteesta ja koosta johtuen.

Ketterien menetelmien pilotointi lähti Notes-tiimissä liikkeelle johdon esittämänä. Tämä saattoi osaltaan vaikuttaa ihmisten reaktioihin ja muutosvastarintaan: tähän mennessä toteutuneita, kahta pilottiprojektia edeltäneet sisäiset koulutukset saivat usean tiimin jäsenen suhtautumaan Scrumiin miettien ”Tätäkö tämä onkin? Lisää työtä, byrokratiaa ja uusia työkaluja?”. Muutoksen liikkeellepanijat olisivat ideaalitilanteessa sekä johto että alaiset eli projektien jäsenet yhdessä: kun olemassa olevia yhteisiä haasteita ja kehittämiskohteita nostetaan esille yhdessä ja peilataan niitä esimerkiksi Scrumin käytäntöihin, joiden uskotaan voivan auttaa haasteissa, on ihmisten sitoutuminen työtapojen ja viitekehysten muutokseen todennäköisesti suopeampaa ja parempaa. Pilottiprojekteja ajatellen olisi kaikkien projektien jäsenten ollut hyvä saada enemmän perehdytystä ja koulutusta Scrumin käyttöön.

Scrumin kokonaisvaltainen käyttö ei onnistu, mikäli myös asiakas ei sitoudu siihen, joten sen käyttö on hyväksytettävä asiakkaalla. Tämä edellyttänee monesti perusteltua puolustuspuhetta Scrumin puolesta ja sen käytöstä saatavien etujen esille nostamista. Pilottiprojekteissa Scrumin käytön perustelussa onnistuttiin hyvin. Yhtenä vahvimpana asiakkaan luottamusta kasvattaneena tekijänä lienee ollut heille esitetty fakta ketterien menetelmien asiakkaalle tarjoamasta edusta: Scrumin priorisoidun tuotteen työlistan avulla asiakas saa etenkin kiinteähintaisissa tai määritetyn ajan puitteissa toimivissa projekteissa maksimaalisen hyödyn. Tällöin asiakas maksaa juuri sitä, mitä se saa – ja eniten haluaa.

Ketterän menetelmän onnistunut käyttöönotto pelkän kirjallisuuteen pohjautuvan itseopiskelun avulla kuulostaa sinisilmäisyydeltä ja hyväuskoisuudelta, mutta voi toki joissakin tapauksissa olla mahdollistakin. Edes sisäiset koulutukset eivät välttämättä tarjoa oikotietä onneen, mikäli kouluttajina toimivat organisaation Scrum-osaajat työskentelevät eri tiimissä kuin Scrumia pilotoivat pro-

jektitiimit. Jo pelkästään eroavaisuudet käytetyssä teknologiassa, asiakkaissa sekä tiimin toimintatavoissa ja kulttuurissa riittävät hämmentämään toiminta-alueita sen verran, että toisessa kontekstissa toimineet menetelmät eivät olisi välttämättä yhtä menestyksekkäitä toisen kontekstin pilotissa.

Luvussa viisi kirjallisuuden ja tieteellisten artikkelien kautta käsitelty ketterien menetelmien pilotointi ja käyttöönotto ohjelmisto-organisaatiossa osoittautui hyödylliseksi taustatutkimukseksi pilottiprojekteja ajatellen: saatujen käyttökokemusten ja vinkkien perusteella osattiin välttää muutamia yleisiä sudenkuoppia, joihin ketterää menetelmää pilotoivat organisaatiot tai tiimit yleensä törmäävät. Kuten Knibergkin (2007) on esittänyt, myös etenkin projekti B:ssä testaus ja laadunvarmistus koettiin yhdeksi haastavimmista asioista toteuttaa tehokkaasti.

Projekti B:n tiimin haasteisiin vastaaminen itseohjautuvasti alkoi toimia yhä paremmin projektin edetessä. Tästä yhtenä parhaimmista aikaansaannoksista voitaneen pitää muutamien kehittäjien aitoa kiinnostusta saada testaus tehokkaammaksi. Lisäksi projektissa on päätetty kokeilla Tuomikosken ja Tervosen (2009) mainitsemaa koko tiimin yhteistä testaussessiota. Molempien pilottiprojektien tiimien itseohjautuvuutta olisi ollut mahdollista hyödyntää vielä toteutunutta enemmänkin. Ongelmatilanteissa tiimi on monesti itse paras tietolähde: se pystyy todennäköisesti ratkaisemaan omat ongelmansa paremmin kuin Scrum-mestari. Tiimi on tietoinen miksi ongelma on syntynyt, mitä sille on jo kenties yritetty tehdä ja mitä vaihtoehtoja olisi vielä jäljellä.

Pilottiprojekteissa suoritetun havainnoinnin perusteella vaikuttaisi ainakin projekti B:n mukaan siltä, että kun projektiin saadaan mukaan testaaja, pyritään testausta siirtämään pelkästään hänen vastuulleen. Selkeimmin asia ilmeni tilanteissa, joissa testaaja oli raportoinut ohjelmistovirheestä ja kehittäjä oli sen sittemmin korjannut. Kun testaaja varmisti regressiotestauksessa ohjelmistovirheen häviämisen paikasta, jossa se edellisen kerran todettiin, oli sattumanvaraista, oliko virhe korjaantunut asiakkaan käyttämälle selainversiolle vai ei. Ai-

emmin esitetyn mukaisesti ketteristä menetelmistä kertova kirjallisuus toteaa niissä suoritettavan testauksen kuuluvan koko tiimille, mikä ei täten toteutunut projekti B:ssä odotetulla tavalla. Tästä haasteesta keskusteltiin tiimin, tuotteen omistajan ja Scrum-mestarin kesken, minkä myötä kehittäjien suorittama ohjelmistovirheiden korjausten testaus parani projekti B:n edetessä.

Pilotoinnin myötä havaittiin, että testauksen aito ja tehokas integroiminen ketteriä menetelmiä käyttävään projektiin on haastavaa. Scrumin odotettiin edesauttavan testauksen pakottamista mukaan projekteihin, sillä Scrumissa testausta tulisi suorittaa säännöllisesti jokaisessa pyrähdyksessä, missä onnistuttiinkin melko hyvin. Jatkoa ajatellen toimeksiantajaorganisaation Notes-tiimin Scrumia käyttävissä projekteissa lienee viisainta lähteä liikkeelle varmistamalla kaikkien eri osapuolten sisäistäneen testauksen merkityksen ja tärkeyden. Järkevinä perusteluina testauksen vähälle jättämiseen ei voida pitää kehityksen hidastumista tai hyvin suoritetun yksikkötestauksen riittävyyttä.

Scrumin pilotointiin toimeksiantajaorganisaation Notes-tiimissä lähdettiin, koska Scrumin odotettiin tuovan systemaattisuutta ja työkaluja koko projektin aikaiseen muutoksen hallintaan ja jatkuvaan kommunikointiin asiakkaan kanssa. Näiden avulla pidettiin mahdollisena välttää samanaikaisesti sekä projektien loppuvaiheiden vaatimustulvaa että tarve kieltäytyä projektin lisätöistä. Kahden tähän mennessä tehdyn pilottiprojektin perusteella on liian aikaista tehdä varmoja johtopäätöksiä, mutta näyttäisi siltä, että Scrumin käytön myötä ollaan oikeilla jäljillä.

Tämän tutkielman sisältämistä pilottiprojekteista tutkivaa testausta käytettiin enemmän projektissa B, jossa aikaa testitapausten kirjoittamiseen ei juurikaan ollut. Tutkivan testauksen käyttö osoitti kuitenkin vahvuutensa: suunnittelemattoman testauksen myötä käyttöliittymästä paljastui käyttöskenaarioita, joita ei ollut otettu huomioon suunnitteluvaiheessa. Suoritetut toiminnot johtivat luonnollisesti ohjelmistovirheiden ja käytettävyyspuutteiden löytymiseen. Lisäksi huomattiin, että kynnys testaukseen tutkivan testauksen myötä on mata-

lampi kuin testitapauspohjaisessa testauksessa. Haasteena projekti B:ssä oli toiminnallisuuksien erittäin suuri määrä, jota pyrittiin pitämään kurissa listaamalla mahdollisimman lyhyesti mutta kattavasti testattavia toiminnallisuuksia ja osakokonaisuuksia.

Pilottiprojektien havainnoinnin tulokset täsmäävät kirjallisuuden esittämään ketterien menetelmien testauksen sijoittumiseen projektissa: testauksen fokusointia ja suorittamisen ajankohtaa on pyrittävä muuttamaan viimeisestä laaduntarkastusasemasta siten, että testaus tuottaa projektille mahdollisimman paljon informaatiota, palautetta ja näkyvyyttä. Käytännössä tämä tarkoittaa pyrähdyksessä suoritettavaa jatkuvaa testausta – niin manuaalista kuin automatisoituakin. Testausta, niin kuin kehitystäkään, ei voida ajatella enää omina vaiheinaan, sillä testausta olisi hyvä kyetä suorittamaan jatkuvasti. Näin voidaan varmistua jatkuvasta edistymisestä sekä siitä, että pyrähdyksen aikana kehitettävät uudet ominaisuudet ja toiminnallisuudet ovat aidosti valmiita. Pitkät aikavälit toteutuksen ja sille suoritetun testauksen välillä lisäävät riskejä ja hukkaa, jolla tarkoitetaan ketterissä menetelmissä kaikkea ylimääräistä.

Jatkuva testauksen suorittaminen voi tuntua uusille Scrumia käyttäville tiimeille samaan aikaan liian työläältä ja aikaa vievältä, minkä myötä houkutuksena voi olla muuttaa testausta suoritettavan esimerkiksi joka kolmannessa pyrähdyksessä. Tässä astuttaisiin kuitenkin harhaan, sillä testauksen tulisi olla yksi keskeisimpiä käytäntöjä Scrumissa ja sen tulisi ohjata ohjelmistokehitystä jatkuvasti. Mikäli testausta halutaan hajauttaa suoritettavaksi harvemmin kuin joka pyrähdyksessä, on taustalla todennäköisesti testauksen työläys. Yksi ratkaisu tähän haasteeseen ja kustannuskysymykseen on testauksen automatisointi. Toisaalta automatisointikin on haastavaa, sillä mikäli siitä halutaan saada maksimaalinen hyöty, on testejä kyettävä automatisoimaan jatkuvasti muuttuville toiminnallisuuksille, joita kehittäjät toteuttavat samaan aikaan. Cohnin (2009, 316) mielestä ainakin toiminnallisuuksien toteutus ja automatisoitujen

testien suunnittelu voidaan aloittaa rinnakkaisesti esimerkiksi hyväksymistestivetoisen kehityksen avulla.

Kun testauksen automatisointia tehdään jo ennen toteutuksen valmistumista, voidaan kyseessä nähdä olevan TDD:n soveltaminen. Tällöin testityökalun on oltava jokin muu kuin tallenna ja toista -tyyppinen, jonka avulla testiä ei voida saada valmiiksi eli tallennettua ennen koodin valmistumista. Käytännössä automatisointityökalun olisi täten oltava sellainen, jossa ajettava testi saataisiin valmiiksi koodaamalla ja vaatimuksia noudattaen.

Ketterää menetelmää käyttävässä projektissa kehitettyjä toiminnallisuuksia kuvaavan termin ”valmis” määritelmä joudutaan monesti sopimaan projekti- tai organisaatiokohtaisesti. Yleisesti ottaen toiminnallisuuden ollessa valmis, voidaan sen olettaa olevan vähintään toteutettu ja testattu. Tämän perusteella voidaan esittää, että mikäli pyrähdyksen aikana on tehtävä paljon testaustyötä, voi toiminnallisuuksien valmis-tilaan saattaminen johtaa osan toteutettaviksi suunniteltujen toiminnallisuuksien poisjättämiseen pyrähdyksestä. Molemmissa pilottiprojekteissa oli tarvetta keskustella ja sopia tiimin kesken määritelmät sille, milloin asioiden voitiin sanoa olevan valmiita. Projekti B:ssä tätä määritelmää päivitettiin projektin etenemisen myötä.

Riippumatta käytettävästä ohjelmistokehitysmenetelmästä dokumentaation tärkeyttä ei voi väheksyä. Valitettavasti ketterät menetelmät eivät tuo mitään uutta ratkaisua sen ajan tasalla pitämiseen. Dokumentoinnin puutteellisuuden myötä syntyneisiin haasteisiin törmättiin pilotoinnin muutos- ja korjauspyynnöissä: asiakkaat ilmoittivat muutospyynnöistä ja korjausehdotuksista kehittäjille puhelimitse tai sähköpostitse. Tällöin kehittäjä toteuttivat muutokset toisinaan ilman, että muutoksesta syntyi mitään muistiinpanoja tai kommentteja, joista olisi ollut hyötyä myöhemmin. Vaikka yksi ketterien menetelmien arvoista toteaa toimivan ohjelmiston olevan dokumentaatiota tärkeämpää, ei tätä pidä ymmärtää siten, ettei dokumentaatiota tehtäisi – varsinkaan tilanteissa, joissa dokumentaatiolle on todella tarvetta. Projekti B:ssä osittain puutteellisen do-

kumentaation parantaminen olisi todennäköisesti ajoittain edesauttanut myös toimivamman ohjelmiston aikaansaamista.

Kun dokumentaatiota ei ole saatavilla, voi suunnitelmakeskeisiä ohjelmistokehitysmenetelmiä käyttäneissä projekteissa toiminut testaaja kohdata ketterien menetelmien myötä haasteita. Osa löydetyistä ohjelmistovirheistä aiheuttaa erimielisyyttä eri osapuolten kesken varsinkin tilanteissa, joissa testaajalla ei ole tarkkaa dokumentaatiota testauksen tueksi. Kehittäjän mielestä löydetty epäloogisuus tai toiminnallisuuden puuttuminen ei ole virhe, vaan laajennustyö mahdolliseen järjestelmän jatkokehitykseen.

Ketteriä menetelmiä käyttävään projektiin menevän testaajan on todennäköisesti muutettava asennoitumistaan ja testauksessa käytettäviä toimintatapoja tuodakseen lisäarvoa projektille. Testaajan on kyettävä tekemään nopeita päätöksiä, oltava luova, ajateltava asioita asiakkaan näkökulmasta ja ennen kaikkea kyettävä hankkimaan ymmärrys testattavan ohjelmiston toiminnallisuuksista ja riskitekijöistä jopa ilman kehittäjien apua tai dokumentaatiota.

Yleispätevää määritelmää sille, kuinka testaus tulisi Scrumissa hoitaa, on vaikeaa antaa. Tutkielman tuloksissa luvussa seitsemän, esitettiin yleisiä hyväksi todettuja ja osittain välttämättömiäkin käytäntöjä, kuten tutkiva testaus ja testauksen automatisointi. Näistä etenkin testauksen automatisoinnin avulla olisi pilottiprojekteissa todennäköisesti ylletty korkeampilaatuisiin tuloksiin. Tutkivaa testausta käytettiin puolestaan etenkin projekti B:ssä ja siitä saadut kokemukset olivat positiivisia. Scrumin soveltaminen on kontekstiriippuvaista, joten miksei testauksenkin osalta voitaisi toimeksiantajaorganisaation Notes-tiimissä kysyä itseltä, mitä Scrum ja testaus siinä tarkoittaa juuri meille ja meidän ympäristössä?

Testauksen automatisointi Notes- ja Domino-ympäristössä, jossa myös molemmat pilottiprojekteissa kehitetyt järjestelmät toimivat, vaikuttaisi olevan melko haasteellista, muttei kuitenkaan mahdotonta. Yksikkötestaus eli esimerkiksi au-

tomatisoitu LotusScript-agentti on jäänyt ainakin tähän asti toteuttamatta. Automatisoidun toimintotestauksen käytännön esimerkkinä voidaan esittää automatisoitua kehitettävän järjestelmän Notes-asetustietokannan tietojen muuttamista ja sen aiheuttamien muutosten oikeellisuuden varmistamista käyttöliittymässä. Kun ympäristönä on Notes-client, ei tämänkään toteuttaminen ole yksinkertaista. Käyttöliittymätestauksen automatisointi on sen sijaan helpommin lähestyttävissä: graafisen Notes-sovelluksen käyttöliittymän tai Notes-pohjaisen www-käyttöliittymän toiminnon suorittaminen (esimerkiksi tilauksen tekeminen järjestelmässä) automatisoidusti voidaan tehdä tallenna ja toista -työkalulla.

Molemmissa pilottiprojekteissa koettiin haastavaksi suorittaa testausta aktiivisen kehitysvaiheen aikana: valmiiden osakokonaisuuksien testaamista haittasi osin niiden riippuvuussuhteet muihin kesken oleviin komponentteihin tai toiminnallisuuksiin. Yksi vaihtoehto vastata tähän haasteeseen olisi voinut olla kokeilla menettelyä, jossa kehittäjät olisivat päivittäin tiettyyn kellonaikaan kertoneet tiimille, mitä he ovat saaneet valmiiksi ja mitkä asiat valmiina testattavaksi. Näin menettelemällä testaajalle olisi todennäköisesti ollut helpompaa alkaa testata pyrähdyksen viimeisellä viikolla sen aikana syntyneitä tuotoksia, joista hän ei välttämättä ollut aikaisemmin kuullut juuri mitään. Edellä mainittua tilannetta pyrittiin kuitenkin paikkaamaan siten, että kehittäjät jakoivat testaajalle vinkkejä siitä, mihin kussakin pyrähdyksessä kannattaa panostaa: mitä uutta on tehty ja mitkä ovat ne asiat, joihin testausta eritoten kannattaa kohdistaa.

Kirjallisuuden perusteella automatisoidulla yksikkötestauksella voidaan saada karsittua ohjelmistovirheitä tehokkaasti siten, etteivät ne enää aiheuta ongelmia myöhemmissä testausvaiheissa. Automatisoidun yksikkötestauksen tärkeyteen on kuitenkin vaikea ottaa kantaa pilottiprojektien perusteella, sillä kuten jo aiemmin todettiin, ei projekteissa käytetty Notes/Domino-teknologia mahdollista helppoa yksikkötestauksen automatisointia.

Pilottiprojekti B:n testauksen pyrähdykseen sijoittaminen oli asia, jota pohdittiin paljon ja kokeiltiin toteuttaakin eri tavoin, yhtä selkeästi parasta vaihtoehtoa kuitenkaan löytämättä. Jos Scrumin periaatteita olisi tunnettu paremmin, olisi testaajan kohtaamista haasteita ollut mahdollista tehdä työlistaan tehtäviä, joiden aihe olisi käsitellyt toteutuksen ja testauksen huonoa yhteen sulautuvuutta. Tällainen työlistan tehtävä olisi voitu merkitä valmiiksi esimerkiksi vasta sitten, kun toteutusjärjestystä olisi muutettu siten että se parantaisi toteutuksen ja testauksen synkronointia. Koska Notes-tiimille on kertynyt kokemuksia Scrumista toistaiseksi melko vähän, ei testauksen sijoituskohdasta pyrähdyksessä olla vielä tässä vaiheessa yksimielisiä, ja eri vaihtoehtoja testauksen sijoittamiseksi tullaankin varmasti kokeilemaan jatkossa. Lähtökohtaisesti uskomus on kuitenkin se, että on oltava mahdollista testata valmiita osakokonaisuuksia jo pitkin pyrähdystä. Tämä vaatii koko tiimiltä työn organisointia ja tehokasta kommunikointia.

Projekti A:ssa suurimmaksi haasteeksi osoittautui juuri testaus. Tähän vaikutti muun muassa toiminnallisuuksien jääminen kesken toteutusvaiheen päätyttyä, minkä myötä keskeneräiset toiminnallisuudet hankaloittivat testiympäristön valmiiden toiminnallisuuksien testaamista. Kahden viikon pyrähdykset koettiin liian lyhyiksi ainakin nyt, kun Scrum oli uusi lähestymistapa projektin läpiviemiseen. Scrumin ja lyhyiden pyrähdysten selkeäksi eduksi voidaan todeta niiden tarjoama mahdollisuus tarkkailla projektin etenemistä jatkuvasti. Tämä edesauttaa mahdollisten aikatauluviivästymisten ja työmääräylitysten havaitsemista hyvissä ajoin, jolloin edellä mainittuihin asioihin on vielä mahdollista vaikuttaa. Projekti A:sta saatujen kokemusten perusteella kehittäjien kannattaa jatkaa testitapausten tekemistä, sillä tällä varmistetaan kehittäjien testaustyön tekeminen. Tällöin myös virheiden määrä pienenee, sillä kehittäjät löysivät samalla toteutuksesta epäloogisuuksia testitapauksia kirjoittaessaan.

Pyrähdysten sisällönsuunnittelupalavereissa olisi hyvä käydä valitut toteutettavat toiminnallisuudet läpi ainakin silloin, jos projektiryhmä ei tunne kehitet-

tävää tai laajennettavaa sovellusta kovin hyvin tai jos toiminnallisuuksia ei ole kuvattu riittävän tarkalla tasolla. Etenkin projekti B:n kohdalla kävi niin, ettei kaikkien tiimin jäsenten palavereihin osallistumista ei nähty tarpeelliseksi. Tämä lisäsi todennäköisesti jonkin verran hiljaisen tiedon määrää – tehdyistä palaverimuistioista huolimatta.

Mikäli asiakas testaa kehitettävää ohjelmistoa aktiivisesti pyrähdyksen aikana, on testauksen tuloksena realistista odottaa ohjelmistovirheraportteja. Löydettyjen virheiden korjaaminen menee oletusarvoisesti uusien toiminnallisuuksien kehittämisen edelle, joten jokaiseen pyrähdykseen on syytä varata aikaa virheiden selvittämistyölle ja korjaamiselle. Koko projektin keskimääräisen etenemisvauhdin selvittäminen ja seuraaminen on hyödyllistä: niiden avulla on helpompaa arvioida, mitä kaikkea pyrähdyksessä ehditään todennäköisesti tehdä ja mitä ei.

Toimeksiantajaorganisaation Notes-tiimin tavoitteena on saada automatisoitua käyttöliittymän testausta jatkossa siten, että esimerkiksi kenttätarkistukset, lomakkeet ja virheilmoitusten esiintymiset testattaisiin ainakin osittain automaattisesti. Tätä lienee helpointa lähteä toteuttamaan selainpohjaisten järjestelmien kehitysprojekteissa. Manuaalisesta testauksesta ei tulla koskaan pääsemään eroon – eikä siitä eroon haluttaisikaan, sillä automatisointiin ei pidä luottaa liikaa ja muun muassa järjestelmien ulkoasun, teemojen ja käytettävyyden testaukseen on järkevintä käyttää tutkivaa testausta. Organisaation toisessa tiimissä arviolta jopa viisikymmentä prosenttia kaikista ohjelmistovirheistä on käyttöliittymään liittyviä näkyviä vikoja, joten jos samansuuntaista virheen esiintymisjakaumaa voidaan olettaa esiintyvän Notes-tiimissä, on myös sen kontekstissa varmasti aidosti tarvetta niin manuaaliselle kuin automatisoidullekin testaukselle.

Tarkastaminen (*checking*) ja tutkiminen (*exploring*) antavat testaajille erilaista tietoa. Tarkastamisen taustalla ovat konkreettiset odotukset toiminnallisuuksista, kun taas tutkimiseen ei tarvita mitään esitietoja testattavasta järjestelmästä tai

ohjelmistosta. Tutkiminen paljastaa helpommin asioita, joita ei ole välttämättä otettu huomioon tarkastamisessa. Automatisointia mietittäessä on tarkastaminen tutkimista helpompaa ja mielekkäämpää automatisoida, sillä tarkastaminen on ennalta tiedettyä toimintaa tutkimisen perustuessa enemmän intuition varassa toimimiseen.

Ketterien menetelmien käyttöönotossa on oltava maltillinen: täysimittaista uuden menetelmän käyttöä ei tule aloittaa ennen sen tehokasta käyttöä edellyttävien asioiden, kuten kunnollisen perehtymisen ja tarvittavien työkalujen selvittämistä. Toisaalta on vaikea vetää raja esimerkiksi harjoittelun ja käytännön pilotoinnin välille: onko ”oikea pilotointi” sitä, että pilotointia tehdään oikeassa asiakasprojektissa? Saadaanko sisäisistä projekteista, joista puuttuu ulkopuolinen asiakas, mitään oikeita käytännön kokemuksia vai ainoastaan vinkkejä siihen, voiko jokin tietty ketterä menetelmä toimia asiakasprojekteissa?

Projekti B:ssä havaittuun toteutuksen venymiseen ja sen aiheuttamaan testauksen myöhässä alkamiseen on jatkoa ajatellen pystyttävä keksimään ratkaisu. Tuotteen työlistan kärkipäähän priorisoitujen toiminnallisuuksien sisällön arvioiminen jo ennen pyrähdykseen ottamista voisi olla yksi kokeilemisen arvoinen asia. Tämä voisi vähentää sitä, että osakokonaisuuksien toteuttamiseen tarvittava todellinen työpanos selviääkin vasta asioiden ollessa jo toteutuksen alla ja on kenties jo tehty asioita tavoilla, joiden järkevyydestä ei olla täysin varmoja.

Verrattaessa alkuhaastatteluissa nousseita testauksen haasteita ja kehityskohteita piloteista saatuihin havaintoihin ja loppukyselyjen vastauksiin, voidaan todeta testauksen olevan menossa oikeaan suuntaan: yhteisiä testauskäytäntöjä haetaan ja kehitetään aktiivisesti yhdessä, kehittäjien kiinnostus laadun parantamiseen testauksen kautta on herännyt, testausta on saatu tehokkaammin projekteihin ja testauksen automatisoinnin tärkeyden tunnistaminen on johtanut tahtotilaan saada sitä mukaan projekteihin osaksi laadunvarmistusta.

9 YHTEENVETO

Tässä tutkielmassa tarkasteltiin kirjallisuuskatsauksen ja tapaustutkimuksen muodossa ketterän menetelmän pilotointia ohjelmisto-organisaatiossa ja ohjelmistotestaukseen kohdistuvia muutostarpeita organisaation siirtyessä perinteisistä ohjelmistokehitysmenetelmistä Scrumin käyttöön. Tutkimuksen tavoitteena oli selvittää mitä haasteita ja muutostoimenpiteitä ohjelmistotestaukseen on odotettavissa, kun projekteissa tapahtuvaa ohjelmistokehitystä aletaan tehdä perinteisten ohjelmistokehitysmenetelmien sijaan Scrumia käyttäen.

Aluksi, luvussa kaksi esiteltiin perinteiset ohjelmistokehitysmenetelmät ja määriteltiin sekä kuvattiin niissä tapahtuva ohjelmistotestaus. Tämän jälkeen luvussa kolme perehdyttiin ketteriin menetelmiin ja vastaavasti tarkasteltiin niissä suoritettavaa testausta. Lisäksi luotiin katsaus testaajaan rooliin. Luvussa neljä keskityttiin toimeksiantajaorganisaation valitsemaan pilotoitavaan ketterään menetelmään, Scrumiin. Luku viisi käsitteli ketterän menetelmän pilotointia ohjelmisto-organisaatiossa haasteineen ja kirjallisuudesta löytyvine case-esimerkkeineen. Luvussa kuusi kuvattiin tutkielman empiirisen osuuden kohde, tutkimuksen menetelmät ja sen toteuttaminen. Tutkimustulokset, jotka käsittävät alkuhaastatteluiden, havainnoinnin ja loppukyselyiden tulokset ja tutkimusongelmaan esitetyn vastauksen, esiteltiin luvussa seitsemän, jonka jälkeinen luku kahdeksan sisälsi pohdinnan ja johtopäätökset.

Molemmat pilottiprojektit sisälsivät haasteita Scrumin käytön myötä. Muutosvastarintaa kohdistui niin Scrumin käytänteisiin kuin myös testauksen muuttuneisiin toimintatapoihin, joista jälkimmäistä pidetään kirjallisuudessa ketterien menetelmien yhtenä vaikeimmista osa-alueista. Pilottiprojekteissa testausta saatiin jaettua jonkin verran koko tiimin kesken, mutta tekemistä testauksen tehostamiseksi riittää vielä paljon jatkossa. Testauksen automatisoinnin tärkeys ja ajanpuute yksityiskohtaisten testitapausten kirjoittamiseen, jotka molemmat

mainitaan myös kirjallisuudessa, todistettiin paikkansapitäviksi myös pilotti-projekteissa.

Ketterien menetelmien pilotointia ja käyttöönottoa koskevissa lukuisissa tutkimuksissa on esitetty ketterän menetelmän käyttöönoton olevan haastava ja pitkä prosessi. Käyttöönotto ei pääty periaatteessa koskaan, vaan on saatujen kokemusten avulla tapahtuvaa jatkuvaa käytäntöjen parantamista ja muokkaamista. Käyttöönoton onnistumiseen vaikuttavat niin huolellinen taustatyö ja suunnittelu, muutosvastarinta, organisaatiokulttuuri, projektin luonne, ketterän menetelmän valinta, ajankohta, tiimin jäsenet kuin käytettävä teknologiakin. Näitä kaikkia on siis pohdittava ja suunniteltava tarkkaan ennen käyttöönottoa tai pilotointia. Ketteryys edellyttää pitkällä tähtäimellä ammattitaitoisia, motivoituneita tiimejä ja työntekijöitä sekä tehokasta ja jatkuvaa kommunikointia tiimien sisällä, niiden välillä ja asiakkaiden kanssa.

Mikäli työntekijät eivät koe saavansa uusista käytännöistä hyötyä ja etuja, tai käytäntöjen koetaan uhkaavan työntekijää jollakin tavoin, tulee ketterän menetelmän täysimittaisessa soveltamisessa olemaan paljon haasteita. Näiden lisäksi avainasemassa on myös asiakassuhteiden, -kommunikoinnin ja asiakkaan toiminta- ja ajatusmallien saattaminen toimeksiantajan käyttämää ketterää menetelmää vastaavaksi. Tässä piilee toisaalta vaarana se, että asiakas saa väärän kuvan toimintamalleista ja kuvittelee voivansa muuttaa mieltymyksiään kehitettävästä järjestelmästä mielin määrin.

Ketterän menetelmän käyttöönotossa on oltava huolellinen siinä, mitä muutetaan: sopeutujana on tiimi toimintatapoineen eikä esimerkiksi pyrähdys, jonka pituutta vaihdellaan tilannekohtaisesti projektin aikana. Vastaavasti testauksen on pääsääntöisesti mukauduttava ja sulauduttava projekteihin sen sijaan, että projektit eläisivät tai muuttuisivat testauksen takia. Asia ei ole kuitenkaan täysi yksiselitteinen, sillä testauksen saamiseksi tehokkaaksi voi edellyttää esimerkiksi toteutusaikataulujen hienosäätöä.

Ketterien menetelmien käytäntöjen perimmäinen tarkoitus kannattaa selvittää Agile Manifeston periaatteista ja arvoista. Tästä esimerkkinä voidaan mainita päivittäiset tilannepalaverit: niiden avulla voidaan toteuttaa useita ketterien menetelmien periaatteita ja arvoja, sillä kasvotusten tapahtuva, lyhytkestoinen keskustelu nykytilanteesta edesauttaa niin toimivan ohjelmiston tuottamista, kykyä vastata muutoksiin kuin tiimin itseohjautuvuuttakin. Tiimin tulee ottaa käytännöt käyttöön siksi, että niiden merkitys ja perimmäinen syy, ketterien menetelmien arvojen toteutuminen, saadaan toteutettua käytäntöjen avulla. On vaarallista alkaa käyttää erilaisia käytäntöjä pelkästään siksi, että niitä käyttävät muutkin. Tällöin ei välttämättä tiedetä, miksi asioita tehdään tietyllä tavalla.

Testaus Scrumissa edellyttää koko tiimin sitoutumista, pyrähdysten suunnittelua, jatkuvaa vuorovaikutusta, automatisointia ja kykyä tunnistaa riskejä ja toiminnallisuuksia ilman dokumentaatiota. Scrumissa ohjelmiston laatu ja siihen kuuluva testaus ei ole ainoastaan testaajien, vaan koko tiimin vastuulla. Testaus ei ole viimeinen projektissa suoritettu vaihe, joka toimii eräänlaisena turvaverkkona ja viimeisenä tarkastuspisteenä ennen ohjelmiston siirtämistä tuotantoympäristöön, vaan testausta suoritetaan koko projektin ajan. Tämän ja jatkuvan integroinnin avulla laatu rakennetaan tuotteen sisään jo sen toteutusvaiheessa.

Testauksen automatisointi on kirjallisuuden perusteella äärimmäisen tärkeä ketterän testauksen osa-alue, jota tulisi tehdä monella eri tasolla. Automatisoidun testauksen tärkeyttä tukevat myös tämän tutkielman pilottiprojekteista saadut tulokset: automatisoidun testauksen puuttuessa etenkin regressiotestaus jäi puutteelliseksi. Testauksen automatisointi on kallista, sillä ylläpitoon kuluu paljon aikaa muun muassa ympäristöjen pystytyksiin ja testitapausten päivityksiin. Lisäksi tehokas testauksen automatisointi edellyttää kompetenssia ja hyviä työkaluja.

Tutkiva testaus on yhdistettyä testauksen suunnittelua, suorittamista ja oppimista, joten se on hyvinkin pitkälti vastakohta vesiputousmallissa tyypilliselle,

perinteiselle dokumentaatiokeskeiselle testaukselle. Tutkivan testauksen tehokkuus perustuu pitkälti testaajan taitoihin ja kykyyn analysoida järjestelmää ja sen käyttäytymistä. Tutkivan testauksen etuja ovat sen tehokkuus, muokattavuus ja jämäkkyys. Testitapaukset ja -skriptit alkavat menettää tehoaan ajan myötä, kun samoja tapauksia ja skenaarioita on suoritettu useita kertoja. Mutta koska tutkiva testaus on suorittamista, jossa muokataan toimintaa tilanteen mukaan ja yritetään tuoda koko ajan jotakin uutta esille, löydetään uusia ohjelmistovirheitä todennäköisemmin sen avulla kuin vanhoja, jo useita kertoja suoritettuja testitapauksia suorittamalla.

Tutkimuksen rajoitteena voidaan pitää ketterien menetelmien pilotoinneista ja käyttöönotoista kertovien tieteellisten julkaisujen ja raporttien luonnetta: ne ovat edelleen pääsääntöisesti sisällöltään positiivisia eli kertovat onnistumiseen johtaneista käyttöönotoista ja toimintatapojen muutoksista. Lisäksi voidaan kyseenalaistaa ketterien menetelmien käyttöönotosta ja sen käytöstä tehtyjen tilastojen luotettavuus: voi olla, että onnistuneisiin käyttöönottoihin verrattuna kynnys epäonnistuneiden tapausten julki tuomiseen ei ole helppoa.

Tässä tutkielman ulkopuolelle rajattiin se, kuinka paljon testauksen tehokkuuteen vaikuttavat ketterien menetelmien käytännöt ja tekniikat. Jatkotutkimusaiheena olisi mielenkiintoista selvittää laajassa mittakaavassa se, kuinka hyvin ketterää testausta voi suorittaa, mikäli sitä tukevien käytäntöjen, kuten muun muassa jatkuvan integroinnin ja refaktoroinnin toteuttaminen ei ole mahdollista.

Jatkotutkimuksen osalta toiseksi mielenkiintoiseksi ja tämän tutkielman puitteissa tehdyn kirjallisuuskatsauksen perusteella tutkimattomaksi aihealueeksi voidaan nostaa myös ketterän testauksen kannattavuuden ja kustannusten selvittäminen: missä suhteessa on kustannustehokasta suorittaa testausta projektin laskuun verrattuna loppukäyttäjien tuotantovaiheessa suorittamaan testaukseen. Monissa tilanteissa testauksen määrää ei voida päättää pelkästään kustannusten perusteella, sillä vähintään yhtä tärkeänä tekijänä on kehitettävän jär-

jestelmän liiketoimintakriittisyys esimerkiksi pankkisovelluksien tai sairaaloiden järjestelmien kanssa. Myös se, kenelle järjestelmä tehdään, ratkaissee niin toteutukseen kuin testaukseenkin panostamista: sisäisesti, organisaatiolle itselleen kehitettyjen järjestelmien mahdolliset ohjelmistovirheet eivät todennäköisesti saa aikaan niin suurta fyysistä tai imagollista vahinkoa kuin ulkoiselle asiakkaalle toimitetut, virheellisesti toimivat ohjelmistot. Testauksen kustannuksia laskettaessa on selvitettävä myös se, milloin automatisoitua testausta kannattaa todella lähteä toteuttamaan. Mielenkiintoista olisi pohtia myös tutkivan testauksen kustannusten laskemisessa käytettävää mittaria: kiinnostaako projektin osapuolia tutkivaan testaukseen käytetyt tunnit, sen avulla löydettyjen ohjelmistovirheiden määrä ja vakavuus vai löydettyjen ohjelmistovirheiden ajankohta eli se, miten nopeasti virheet löydetään?

Tässä tutkielmassa käsiteltiin ketterien menetelmien testausta ensisijaisesti testaajan näkökulmasta. Kirjallisuuden perusteella testaus ja laadunvarmistus kuuluvat kuitenkin koko tiimin vastuulle. Täten jatkotutkimustarvetta olisi myös sille, onko ketterissä menetelmissä perinteisiä ohjelmistokehitysmenetelmiä helpompaa saada testausta ja laadunvarmistusta yhteisesti koko tiimin vastuulle. Tässä yhteydessä voitaisiin myös selvittää, mitä ovat hyvän kehittäjän testausominaisuudet, eli millainen kehittäjä on hyvä ja tehokas testaaja. Kohdistuuko kehittäjää kohtaan muita vaatimuksia sen lisäksi, että hänen on osattava ajatella eri tavalla, kriittisesti omaa tuotostaan kohtaan?

Neljänneksi jatkotutkimusaiheeksi sopisi erikoisempien testausten, kuten käytettävyystestauksen, tietoturvatestauksen ja kuormitustestauksen suorittaminen Scrumissa: voidaanko edellä mainitut testaukset hoitaa perinteisten ohjelmistokehitysmenetelmien kuvaamien käytäntöjen mukaisesti?

LÄHTEET

Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta J. (2002). *Agile Software Development Methods. Review and Analysis*. Espoo: VTT Publications.

Abrahamsson, P., Warsta, J., Siponen, M. T. & Ronkainen, J. (2003). New Directions on Agile Methods: A Comparative Analysis. Teoksessa L. Clarke, L. Dillon & W. Tichy (toim.) *Proceedings of the 25th International Conference on Software Engineering*, Oregon, Portland, May 3–10. Los Alamitos: IEEE Computer Society, 244–254.

Adrion, W. R., Branstad, M. A. & Cherniavsky, J. C. (1982). Validation, Verification and Testing of Computer Software. *ACM Computing Surveys*, 14(2), 159–192.

Agile (2009). *Agile testing: The Changing Role of Testers* [online]. Saatavilla www-osoitteessa <http://www.agile-software-development.com/2008/02/agile-testing-changing-role-of-testers.html> [viitattu 27.9.2009].

Agile Manifesto (2001). *Manifesto for Agile Software Development* [online]. Saatavilla www-osoitteesta <http://www.agilemanifesto.org> [viitattu 2.4.2009].

Agraval, M. (2008). *SCRUM Development Process* [online]. Saatavilla www-osoitteessa <http://geekswithblogs.net/emanish/archive/2008/10/24/126087.aspx> [viitattu 27.10.2009].

Asproni, G. (2004). Motivation, Teamwork and Agile Development. *Agile times*. 4(1), 8–15.

Bach, J. (2000). *Session-Based Test Management* [online]. Saatavilla www-osoitteesta <http://www.satisfice.com/articles/sbtm.pdf> [viitattu 16.9.2009].

Bach, J. (2001). *What is Explorary Testing?* [online]. Saatavilla www-osoitteesta <http://www.stickyminds.com/s.asp?F=S2255_COL_2> [viitattu 1.10.2009].

Bach, J. (2002). *Explorary Testing Explained* [online]. Saatavilla www-osoitteesta <http://www.satisfice.com/articles/et-article.pdf> [viitattu 6.4.2009].

Beavers, P. A. (2007). Managing a Large "Agile" Software Engineering Organization. Teoksessa J. Eckstein, F. Maurer, R. Davies, G. Melnik & G. Pollice (toim.) *Proceedings of the Agile 2007*, Washington DC, August 13–17. Los Alamitos: IEEE Computer Society, 296–303.

Beizer, B. (1990). *Software Testing Techniques (2nd edition)*. New York, NY: International Thomson Computer Press.

BetterExplained. (2008). *Understanding the Pareto Principle (The 80/20 rule* [online]*)*. Saatavilla www-osoitteesta <http://betterexplained.com/articles/understanding-the-pareto-principle-the-8020-rule/> [viitattu 29.9.2009].

Black, R. (2007). *Pragmatic Software Testing: Becoming and Effective and Efficient Test Professional*. Indianapolis, IN: Wiley.

Boehm, B. W. (1979). Guidelines for Verifying and Validating Software Requirements and Design Specifications. Teoksessa P. A. Samet (toim.) *Proceedings of Euro IFIP'79*, London, UK, September 25–28. Amsterdam: Elsevier, 711–719.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *Computer, 21*(5), 61–72.

Boehm, B. W. & Turner, R. (2003a). Observations on Balancing Discipline and Agility. Teoksessa *Proceedings of the 2003 Agile Development Conference*, Salt Lake City, UT, June 25–28. Washington: IEEE Computer Society, 32–39.

Boehm, B. W. & Turner, R. (2003b). Using risk to balance agile and plan-driven methods. *Computer, 36*(6), 57–66.

Buchan, B. (2006). *LotusScript Coding – Best Practices* [online]. Saatavilla www-osoitteesta <http://www.hadsl.com/hadsl.nsf/Vienna_Buchan_Lotusscript Coding Best Practices.pdf> [viitattu 26.10.2009].

Cao, L. & Ramesh, B. (2007). Agile Software Development: Ad hoc Practices or Sound Principles? *IT Professional, 9*(2), 41–47.

Cho, L. (2009). Adopting an Agile Culture. Teoksessa Y. Dubinsky, T. Dybå, S. Adoplh & A. Sidky (toim.) *Proceedings of the Agile 2009 Conference*, Chicago, IL, August 24–28. Los Alamitos: IEEE Computer Society, 400–403.

Cohn, M. (2006a). *Selecting the Right Iteration Length for Your Software Development Process* [online]. Saatavissa www-osoitteesta <http://www.mountaingoatsoftware.com/articles/30-selecting-the-right-iteration-length-for-your-software-development-process> [viitattu 12.10.2009].

Cohn, M. (2006b). *Agile Estimating and Planning*. Upper Saddle River, NJ: Prentice Hall PTR.

Cohn, M. (2008). *Patterns of Agile Adoption* [online]. Saatavilla www-osoitteesta <http://www.agilejournal.com/articles/columns/column-articles/734-patterns-of-agile-adoption> [viitattu 3.10.2009].

Cohn, M. (2009). *Succeeding with Agile. Software Development Using Scrum*. Upper Saddle River, NJ: Addison-Wesley.

Cohn, M. & Ford, D. (2003). Introducing an Agile Process to an Organization. *Computer,* 36(6), 74–78.

Craig, R. D. & Jaskiel S. P. (2002). *Systematic Software Testing*. Norwood, MA: Artech House Publishers.

Crispin, L. & Gregory, J. (2009). *Agile Testing. A Practical Guide for Testers and Agile Teams*. Upper Saddle River, NJ: Pearson Addison-Wesley.

Crispin, L. & House, T. (2002). *Testing Extreme Programming.* Boston, MA: Pearson Education.

Davies, R. & Sedley, L. (2009). *Agile Coaching*. Raleigh, NC: The Pragmatic Bookshelf.

Dijkstra, E. W. (1972). *Structured Programming – Chapter I: Notes on Structured Programming.* London: Academic Press, 1–82.

Dubé, L. & Paré, G. (2003). Rigor in Information Systems Positivist Case Research: Current Practices, Trends, and Recommendations. *MIS Quarterly*, 27(4), 597–636.

Elssamadisy, A. (2008). *Agile Adoption Patterns: A Roadmap to Organizational Success.* Boston, MA: Addison-Wesley Professional.

Fry, C. & Greene, S. (2007). Large Scale Agile Transformation in an On-demand World. Teoksessa J. Eckstein, F. Maurer, R. Davies, G. Melnik & G. Pollice

(toim.) *Proceedings of the Agile 2007*, Washington DC, August 13–17. Los Alamitos: IEEE Computer Society, 136–142.

Gat, I. (2006). How BMC is Scaling Agile Development. Teoksessa J. Chao, M. Cohn, F. Maurer, H. Sharp & J. Shore (toim.) *Proceedings of the conference on AGILE 2006*, Minnesota, Minneapolis, July 23–28. Washington: IEEE Computer Society, 315–320.

Glass, R. L. (2003). *Facts and Fallacies of Software Engineering*. Boston, MA: Addison-Wesley.

Haikala, I. & Märijärvi, J. (1997). *Ohjelmistotuotanto*. 4. uudistettu painos. Jyväskylä: Gummerus Kirjapaino Oy.

Haikala, I. & Märijärvi, J. (2006). *Ohjelmistotuotanto*. 11. uudistettu painos. Jyväskylä: Gummerus Kirjapaino Oy.

Hennell, M. A., Hedley, D. & Riddell, I. J. (1984). Assessing a Class of Software Tools. Teoksessa T. Straerer, W. E. Howded & J.-C. Rault (toim.) *Proceedings of the 7th International Conference on Software Engineering*, Orlando, Florida, March 26–29. Piscataway: IEEE Press, 266–277.

Hetzel, B. (1988). *The Complete Guide to Software Testing*. (2nd ed.). Wellesley, MA: QED Information Sciences.

Hirsch, M. (2005). Moving From a Plan Driven Culture to Agile Development. Teoksessa *Proceedings of the 27th International Conference on Software Engineering*, St. louis, MO, May 15–21. New York: ACM, 38.

Hirsjärvi, S., Remes, P. & Sajavaara, P. (2008). *Tutki ja Kirjoita*. 13. uusittu painos. Helsinki: Tammi.

Hochmüller, E. & Mittermeir, R. T. (2008). Agile Process Myths. Teoksessa P. Kruchten & S. Adolph (toim.) *Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices Or Shoot-Out at the Agile Corral*, Leipzig, Germany, May 10–18. New York: ACM, 5–8.

Holler, R. (2006). Debunking Myths of Agile Development. *Better Software*, 8(5), 22–27.

Itkonen, J., Mäntylä, M. V. & Lassenius, C. (2007). Defect Detection Efficiency: Test Case Based vs. Exploratory Testing. Teoksessa *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*,

Madrid, Spain, September 20–21. Washington: IEEE Computer Society, 61–70.

Itkonen, J., Rautiainen, K. & Lassenius, C. (2005). Towards Understanding Quality Assurance in Agile Software Development. Teoksessa H.E. Andersin, R. Niemi & V. Hirvonen (toim.) *Proceedings of the International Conference on Agility*, Helsinki, Finland, July 27–28. Helsinki University of Technology, 201–207.

Jochems, R. & Rodgers, S. (2007). The Rollercoaster of Required Agile Transition. Teoksessa G. Melnik & P. Kruchten (toim.) *Proceedings of the AGILE 2007*, Washington DC, August 13–17. Los Alamitos: IEEE Computer Society, 229–233.

Juristo, N., Moreno, A. M. & Vegas, S. (2004). Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering,* 9(1–2), 7–44.

Kaner, C., Bach, J. & Pettichord, B. (2002). *Lessons Learned in Software Testing*. New York, NY: Wiley.

Karhu, K., Repo, T., Taipale, O. & Smolander, K. (2009). Empirical Observations on Software Testing Automation. Teoksessa *Proceedings of the International Conference on Software Testing Verification and Validation*, Denver, Colorado, April 1–4. Washington, IEEE Computer Society: 201–209.

Keith, C. & Cohn, M. (2008). How to Fail with Agile? 20 Tips to Help You Avoid Success. *Better Software*, 10(6), 24–28.

Kelly, A. (2008). *Changing Software Development: Learning to Become Agile.* Hoboken, NJ: Wiley.

Kemerer, C. F. (1987). An Empirical Validation of Software Cost Estimation Models. *Communications of the ACM, 30*(5), 416–429.

Kinoshita, F. (2008). Practices of an agile team. Teoksessa G. Melnik, P. Kruchten & M. Poppendieck (toim.) *Proceedings of the Agile 2008*, Toronto, Canada, August 4–8. Washington, IEEE Computer Society: 373–377.

Kniberg, H. (2007). *Scrum and XP from the Trenches. How We Do Scrum* [online]. C4Media. Saatavilla www-osoitteesta <http://www.infoq.com/minibooks/scrum-xp-from-the-trenches> [viitattu 10.9.2009].

Kohl, J. (2005). *Exploratory Testing on Agile Teams* [online]. Saatavilla www-osoitteesta <http://www.informit.com/articles/printerfriendly.aspx?p=405514> [viitattu 21.11.2009].

Larman, C. (2004). *Agile and Iterative Development: A Manager's Guide*. Boston, MA: Addison-Wesley.

Larman, C. & Basili, V. R. (2003). Iterative and Incremental Developments. A Brief History. *Computer,* 36(6), 47–56.

Lindvall, M., Basili, V. R., Boehm, B. W., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. A. & Zelkowitz, M.V. (2002). Empirical findings in agile methods. Teoksessa D. Wells & L. Williams (toim.) *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, Chigago, IL, August 4–7. London: Springer-Verlag, 197–207.

Loveland, S., Miller, G., Prewitt, R. & Shannon, M. (2005). *Software Testing Techniques: Finding the Defects that Matter.* Hingham, MA: Charles River Media.

Manna, Z. & Waldinger, R. (1978). The Logic of Computer Programming. *Software Engineering, IEEE Transactions on Software Engineering,* 4(3), 199–229.

Maples, C. (2009). Enterprise Agile Transformation: The Two-year Wall. Teoksessa Y. Dubinsky, T. Dybå, S. Adolph & A. Sidky (toim.) *Proceedings of the Agile 2009 Conference*, Chicago, IL, August 24–28. Washington: IEEE Computer Society, 90–95.

Mar, K. & Szalvay, L. (2008). Where to Begin?: Criteria for Choosing a Successful Scrum Pilot. Julkaisussa *The Agile Journal* [online]. Saatavilla www-osoitteesta <http://www.agilejournal.com/articles/columns/articles/1231-where-to-begin-criteria-for-choosing-a-successful-scrum-pilot> [Viitattu 24.4.2009].

Marchi, M. (2009). Weaponized Scrum. Teoksessa Y. Dubinsky, T. Dybå, S. Adolph & A. Sidky (toim.) *Proceedings of the Agile 2009 Conference*, Chicago, IL, August 24–28. Washington: IEEE Computer Society, 107–112.

Miller, A. & Carter, E. (2007). Agility and the Inconceivably Large. Teoksessa J. Eckstein, F. Maurer, R. Davies, G. Melnik & G. Pollice (toim.) *Proceedings*

of the Agile 2007, Washington DC, August 13–17. Los Alamitos: IEEE Computer Society, 304–308.

Milunsky, J. (2009). *Coping with Change on Scrum Projects* [online]. Saatavilla www-osoitteesta <http://www.infoq.com/articles/coping-with-change> [Viitattu 1.10.2009].

Myers, G. J. (2004). *The Art of Software Testing* (2nd edition). Hoboken, NJ: John Wiley & Sons.

Nerur, S., Mahapatra, R. & Mangalaraj, G. (2005). Challenges of Migrating to Agile Methodologies. *Communications of the ACM*, 48(5), 72–78.

Patton, M. Q. (2002). *Qualitative Research and Evaluation Methods*. Thousands Oaks, CA: Sage Publications.

Patton, R. (2006). *Software Testing.* (2nd edition). Indianapolis: Sams publishing.

Pettichord, B. (2000). Testers and Developers Think Differently. *Software Testing & Quality Engineering*, 2(1), 42–45.

Pezzé, M. & Young, M. (2008). *Software Testing and Analysis: Process, Principles and Techniques*. Hoboken, NJ**:** Wiley.

Royce, W. (1970). Managing the Development of Large Software Systems: Concepts and Techniques. Teoksessa *Proceedings of the IEEE Wescon*, Los Angeles, CA, August 25–28. 1–9.

Schatz, B. & Abdelshafi, I. (2005). Primavera Gets Agile: A Successful Transition to Agile Development. *IEEE Software,* 22(3), 36–42.

Schatz, B. & Abdelshafi, I. (2006). The Agile Marathon. Teoksessa J. Chao, M. Cohn, F. Maurer, H. Sharp & J. Shore (toim.) *Proceedings of the conference on AGILE 2006*, Minnesota, Minneapolis, July 23–28. Washington: IEEE Computer Society, 8–146.

Schwaber, K. (1995). *SCRUM Development Process* [online]. Saatavilla www-osoitteesta <http://www.jeffsutherland.com/oopsla/schwapub.pdf> [viitattu 13.9.2009].

Schwaber, K. (2004). *Agile Project Management with Scrum*. Redmond, WA: Microsoft Press.

Sidky, A., Arthur, J. & Bohner, S. (2007). A Disciplined Approach to Adopting Agile Practices: The Agile Adoption Framework. *Innovations in Systems and Software Engineering,* 3(3), 203–216.

Smith, G. & Sidky, A. (2009). *Becoming Agile: In An Imperfect World*. Greenwich, CT: Manning Publications ltd.

Sumrell, M. (2007). From Waterfall to Agile - How Does a QA Team Transition? Teoksessa J. Eckstein, F. Maurer, R. Davies, G. Melnik & G. Pollice (toim.) *Proceedings of the Agile 2007*, Washington DC, August 13–17. Los Alamitos: IEEE Computer Society, 291–295.

Sutherland, J. (2001). Agile *Can* Scale: Inventing and Reinventing SCRUM in Five Companies. *Cutter IT Journal*, 14(12), 5–11.

Takeuchi, H. & Nonaka, I. (1986). The New New Product Development Game. *Hardvard Business Review,* (January-February), 137–146.

Tichy, W. F. (2004). Agile Development: Evaluation and Experience. Teoksessa *Proceedings of the 26th International Conference on Software Engineering 2004*, Edinburgh, Scotland, May 23–28. Los Alamitos: IEEE Computer Society, 692.

Tulisalo, T., Carlsen, R., Guirard, A., Hartikainen, P., McCarthy, G. & Pecly, G. (2002). Domino Designer 6: A Developer's Handbook [online]. Saatavilla www-osoitteessa http://www.redbooks.ibm.com/redbooks/pdfs/sg246854.pdf [viitattu 11.12.2009].

Tuomikoski, J. & Tervonen, I. (2009). Absorbing Software Testing into the Scrum Method. Teoksessa F. Bomarius, M. Oivo, P. Jaring & P. Abrahamsson (toim.) *Proceedings of the 10th International PROFES Conference*, Oulu, Finland, June 15–17. Berlin: Springer, 199–215.

Turk, D., France, R. & Rumpe, B. (2002). Limitations of Agile Software Processes. Teoksessa *Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering*, Alghero, Italy, May 26–29. 43–46.

Virtala, A. (2003). *LOTUS NOTES -työryhmäohjelmiston peruskirja*. Jyväskylä: Docendo.

Waters, K. (2008). *eXtreme Programming Versus Scrum* [online]. Saatavilla www-osoitteesta <http://www.agile-software-

development.com/2008/04/extreme-programming-versus-scrum.html> [viitattu 17.10.2009].

Wellens, K. (2008). The Record & Playback Fairy Tale. *Testing Experience,* 1(4), 22–23.

Williams, L. & Cockburn, A. (2003). Agile Software Development: It's About Feedback and Change. *Computer*, 36(6), 39–43.

Yin, R. K. (2003). *Case Study Research: Design and methods.* (3rd edition). Thousand Oaks, CA: Sage.

LIITE 1. KETTERIEN MENETELMIEN PERIAATTEET

Alla esitellään Agile Manifeston (2001) mukaiset ketterien menetelmien yleiset periaatteet.

- Tärkeimpänä tehtävämme on pitää asiakas tyytyväisenä toimittamalla nopeasti ja jatkuvasti arvokas sovellus.
- Toivotamme tervetulleeksi muuttuvat vaatimukset, myös projektin loppupuolella. Ketterät prosessit hyväksyvät muutokset asiakkaan kilpailueduksi.
- Toimita toimiva sovellus usein, parin viikon - parin kuukauden välein, pyrkien mahdollisimman nopeaan sykliin.
- Liiketoiminnan asiantuntijoiden ja kehittäjien on toimittava yhdessä päivittäin koko projektin ajan.
- Rakenna projektit motivoituneiden yksilöiden ympärille. Anna heille ympäristö ja heidän tarvitsemansa tuki ja luota että he tekevät työnsä.
- Tehokkain kommunikointikeino ja tapa tiedonilmaisuun on kasvokkain käyty keskustelu.
- Ensisijainen edistymisen mittari on toimiva sovellus.
- Ketterät prosessit suosivat kestävää kehitystä. Sponsoreiden, kehittäjien ja käyttäjien on pystyttävä ylläpitämään tasaista työtahtia jatkuvasti.
- Jatkuva tekniseen erinomaisuuteen ja hyvään suunnitteluun panostaminen parantavat ketteryyttä.
- Yksinkertaisuus – tekemättä jätetyn työn määrän maksimointi – on olennaista.
- Parhaat arkkitehtuurit, vaatimukset ja suunnittelumallit kumpuavat itseohjautuvista tiimeistä.

- Säännöllisin väliajoin tiimi pysähtyy miettimään, kuinka voitaisiin tulla vielä tehokkaammaksi ja säätää toimintatapojaan sen mukaisesti.

LIITE 2. HAASTATTELUKYSYMYSRUNKO

Alla olevaa kysymysrunkoa käytettiin pohjana haastatteluille, joiden tulokset esiteltiin luvussa 7.1.

- Miten testaus näkyy työssäsi?
    - tulokset
    - testaustavat
    - dokumentointi, sekä sen ylläpito ja ymmärrettävyys
- Millaisena näet yrityksen / tiimisi testauksen nykytilanteen ja tahtotilan testaukseen?
    - käytetyt resurssit ja tarve vs. todellisuus
    - asenne testaukseen
    - osaaminen
    - heikkoudet ja vahvuudet
    - johdon tuki
    - asenne uusiin työtapoihin ja menetelmiin
- Millaiset ovat erot testauksessa eri projektien/asiakkaiden välillä
- Testauksen priorisointi esim. versioiden välillä (ohjelmistopäivitykset ja mitä testataan)
- Millainen on nykyjärjestelmien tuki testaukseen
- Prosentuaalinen työjakauma (noin arvio) työtehtävien välillä
- Testauksen tuotteistaminen/kaupallistaminen, onko järkevää ja milloin/miten?
- Toiminta virheen / bugin löytymisen jälkeen eli miten prosessi menee nykyisin?
- Jos voisit itsenäisesti päättää, miten haluaisit toteuttaa testauksen ja testausprosessin tiimissäsi / omassa työssäsi?
    - miten prosessin pitäisi edetä?
    - millä työkaluilla?
    - kuka tekee mitäkin? (esim. testitapausten laatiminen)

    - testaajien varaaminen
- Muut avoimet kehitysideat?

LIITE 3. LOPPUKYSELY PILOTTIIN OSALLISTUNEILLE

Alla esitellään Scrum-pilottiprojektien jäsenille sähköpostitse lähetetyn loppukyselyn kysymykset saateteksteineen.

Moi,

kuten aiemmin olen jo asiasta maininnut, niin pyytäisin teitä vastaamaan seuraaviin kysymyksiin. Kyselyn tarkoitus on olla osa graduani, eli tässä haetaan nyt niitä kokemuksia ja fiiliksiä Scrum-pilotista xxx-projektissa. Lisäksi kokemuksilla on varmasti merkitystä Notes-tiimin Scrumin käytössä jatkossa.

Vastatkaa jokaiseen kysymykseen, ja rehellisyys on valttia. Gradussa en luonnollisesti mainitse kenenkään haastateltavan tai asiakkaan nimeä. Käytännössä täytynee XXX:n kanssa sitten katsoa, mitä voi kertoa ja mitä ei. Toivoisin, että perustelet tai heität jonkin käytännön esimerkin jokaiseen kysymykseen jos ja kun on vain mahdollista. Tästä on muistuttamassa sanat "miksi" tai "perustele" kysymysten lopussa.

Kiitos tuhannesti jo etukäteen! Niin ja pahoittelut nopeasta aikataulusta ja hoputtamisesta.

Timo

Loppukyselyn kysymykset:

1. Oliko projektin luonne ja kokonaiskesto pilotointia varten sopiva? Perustele.

2. Oliko sprintin pituudeksi valittu 4 viikkoa mielestäsi oikea? Perustele.

3. Kuvaile asiakkaan roolia ja suhtautumista uuteen, ketterään ohjelmistokehitysmenetelmään?

4. Miten hyvin Scrum-käytäntöjä mielestäsi noudatettiin? Jos jostakin luistettiin/sovellettiin, miksi? Jos mahdollista, nosta Scrumin käytännöistä hyödyllisin esille ja perustele valintasi. (product backlog, daily scrum meeting, sprint planning, retrospective eli jälkiarvioinnit sprintistä jne.)

5. Oliko Scrum-perehdytyksestä / sisäisistä koulutuksista hyötyä projektin onnistuneeseen läpiviemiseen? Olisiko Scrumia jo käyttäneiden kollegoiden tullut olla enemmän konsultoimassa jossakin asiassa?

6. Onko vaatimusten työmäärien arviointi/priorisointi helpottunut vai vaikeutunut? Miksi?

7. Mihin suuntaan projektin sisäinen viestintä ja projektin statuksen seuranta meni Scrumin käytön myötä? Perustele.

8. Aiheuttiko Scrum-mallin mukainen testaus (turhaa) lisätyötä projektin jäsenille? Jos kyllä, mitä? Lisäksi ehdota, miten voitaisiin mahdollisesti toimia jatkossa paremmin.

9. Oliko Scrumin käytöllä vaikutusta projektipäällikön tehtävien määrään? Jos kyllä, mitä?

10. Oliko Scrumin käytöllä vaikutusta sovelluskehittäjän työtehtävien onnistuneeseen läpivientiin? Jos kyllä, mitä?

11. Arvioi mahdollisesti käytettyjen Scrum-työkalujen tehokkuutta ja toimivuutta. Esim. Greenhopper, burndown-chartit, raportointityökalut tms.

12. Mitä olisi mielestäsi kannattanut tehdä toisin?

13. Mitkä ovat tällä hetkellä ajatukset ja tuntemukset Scrumista? Oletko valmis käyttämään scrumia jatkossa pilotista saatujen kokemusten perusteella? miksi?

14. Muuta?