Mikko Merisalo

KARTOITUS MITTAREISTA OSAKSI SCRUM-TIIMIN OHJELMISTOKEHITYSTÄ

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Merisalo, Mikko
Kartoitus mittareista osaksi Scrum-tiimin ohjelmistokehitystä
Jyväskylä: Jyväskylän yliopisto, 2020, 72 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaajat: Seppänen, Ville; Palola, Timo

Ohjelmistokehityksessä, kuten yleisesti liiketoiminnassa, on hyödyllistä käyttää mittareita toiminnan tukena. Ohjelmistokehityksessä käytetään nykyisin varsin usein niin sanottuja ketteriä ohjelmistokehitysmenetelmiä, joista yksi suosituimmista on Scrum. Ketterien menetelmien tueksi on listattu useita erilaisia ja paljon käytettyjä mittareita. Suosituimmat mittarit eivät välttämättä sovellu suoraan kaikkiin tilanteisiin, vaan käyttöön otettavat mittarit olisi hyvä valita kohdealueen ja tarpeiden mukaan.

Tämän tutkimuksen tarkoituksena oli selvittää tutkimuksen kohteena olevan Scrum-tiimin näkökulmasta tiimin ohjelmistokehityksen tueksi käyttöönotettavia mittareita. Mittareiden selvittämisen pohjustukseksi toinen tutkimuksen tarkoitus oli kuvata kohdetiimin ohjelmistokehitysprosessia, jossa Scrum-viitekehyksen ohella hyödynnettiin esimerkiksi DevOps-käytänteitä. Tutkimusmenetelmänä käytettiin iteratiivista design science -otetta, missä tuloksia täydennettiin vaiheittain tiimin kanssa keskustellen. Tutkimuksen tuloksena syntyi vuokaavioita tiimin ohjelmistokehitysprosessista ja mittarilistauksia.

Tiimin ohjelmistokehitysprosesseissa kuvattiin tiimin kehitysjonon tehtävien etenemisen prosessi ja miten sen osana näkyy esimerkiksi koodien katselmointi pull request –pohjaisesti ja koodin automaattinen julkaiseminen. Automaatio on yksi DevOpsin ulottuvuuksista ja hyvä kohde mittareille esimerkiksi automaattisen ja ajantasaisen datan ansiosta. Kehitysjonotehtävien prosesseista kuvattiin featuren, user storyn, taskin ja bugin vaiheet sekä keskustelussa tärkeäksi nostettu release-suunnittelu yleisesti. Mittareita kerättiin useaa eri tiimin toimintaa koskettavan osa-alueen kirjallisuusviitteistä 226 mittarin listaukseen, josta iteratiivisesti rajattiin lopullinen 13 mittarin joukko, missä yhdeksän mittaria oli kirjallisuuden pohjalta kerätystä listasta ja neljä keskustelun pohjalta listaan lisättyä mittaria.

Ohjelmistokehitysprosessien kuvaaminen ei vain pohjustanut mittarikeskustelua, mutta myös mahdollisti tiimille keskustelun oman ohjelmistokehitysprosessinsa tilasta ja sen tulevaisuudesta, sekä loi havainnollistavan kuvauksen kehitysprosessista tiimille. Iteratiivinen ja kirjallisuuteen pohjaava mittareiden kartoittaminen auttoi tiimiä rajaamaan usealta eri osa-alueelta joukon eniten tiimiä itseään kiinnostavia mittareita, joka ei vain ole listaus suosituimmista muualla käytössä olevista mittareista.

Asiasanat: ohjelmistomittarit, Scrum, DevOps, ketterä kehittäminen

ABSTRACT

Merisalo, Mikko
Metrics determination as part of a Scrum team's software development process
Jyväskylä: University of Jyväskylä, 2020, 72 pp.
Information Systems, Master's Thesis
Supervisors: Seppänen, Ville; Palola, Timo

In software business, as in any business, metrics can be a useful tool to support managerial business decisions. Nowadays many of the methods used to build software products are part of agile development method family, out of which one of the most popular is Scrum. Several studies present metrics to support agile software development methods. However, these are often the most popular and most used metrics, which might not suit for every situation. Thus, metrics should be selected based on the target domain and observed needs.

The purpose of this study is to determine metrics from the target Scrum team's perspective to support the team in their software development process. In order to help mapping the metrics, another purpose for this study is to model the team's software development process, which along with Scrum includes for example DevOps practices. Design science type method used builds the results iteratively based on discussions done with the team. The results of this study are software development process models and metrics lists.

The process models describe the handling of the team's backlog items and how are, for example, pull request based code review and continuous delivery included in the process. Automation is one of the main features of DevOps. Based on its automatic and timely data creation, it is a very suitable target for various metrics. Backlog item process models include models for feature, user story, task and bug iterations and a general process for release planning, which was brought up in the team discussions as an important process area. An original list of 226 metrics from articles categorized as coming from various sectors within the team's area of operation narrowed down to a list of 13 metrics during the discussion. Nine of the final metrics are from the original article list and four new were suggested in the team discussions.

Software process models, in addition to giving basis to the metrics discussion, also facilitated a discussion on how the team want to develop and run their software development now and in the future. It also forms a visualization for the team of their development process. Defining metrics iteratively from a list of metrics used in various agile software development sectors enabled the team to form a list of metrics that are suitable to the specific situation and to the needs of the team themselves. Not just a list of the most used metrics in the industry.

Keywords: software metrics, Scrum, DevOps, agile software development

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Usein akateemisessa kirjallisuudessa korostetaan mittaamisen ja mittaroinnin tärkeyttä liiketoiminnan tukena. Näin on myös ohjelmistokehityksen osalta. Tom DeMarc on sanonut, että ”Sitä ei voi hallita, mitä ei voi mitata” (”You cannot control what you cannot measure”) (Tahir ym., 2016). Monissa tutkimuksissa todetaan, että ohjelmistokehityksessä esimerkiksi tuotteen laadun, ohjelmistokehityksen etenemisen ja työpanoksen mittaaminen on tärkeää ja hyödyllistä (Ordonez & Haddad, 2008; Padmini ym., 2015; Tahir ym., 2016). Ohjelmistokehityksen mittaroinnilla ja sen tutkimuksella onkin ohjelmistoalan ikään suhteutettuna jo varsin pitkä historia. Ensimmäisiä mittareita on esitetty käytetyn jo 1960-luvulla, ensimmäinen kirja aiheesta on julkaistu 1970-luvulla ja tutkimusta aiheesta on tehty myös jo useamman vuosikymmenen ajan (Fenton & Neil, 2000). Usein kuitenkin tutkijat huomauttavat, että mittareiden käyttöä ei liiketoiminnassa ole niin paljoa vuosien aikana hyödynnetty, vaikka tutkimukset niiden tärkeyttä ovatkin korostaneet (Fenton & Neil, 2000; Ordonez & Haddad, 2008).

Ohjelmistomittareiden liiketoimintakäytön ja tutkimuksen alkuaikojen ohjelmistokehitysmenetelmiä kuvataan usein nykyisin ns. perinteisiksi ohjelmistokehitysmenetelmiksi. Nykypäivänä perinteiset menetelmät ovat suurelta osin korvaantuneet ns. ketterillä ohjelmistomenetelmillä, joista yksi suosituimmista on Scrum, ja myös DevOps on viime aikoina noussut suosituksi (CollabNet VersionOne, 2019). Ketterät ohjelmistokehitysmenetelmät eivät kahlitse käyttäjiään tiettyihin muotteihin vaan menetelmien sisällä on usein paljon varaa soveltaa. Scrum ei esimerkiksi ota kantaa varsinaisiin ohjelmistokoodin tuottamisen menetelmiin, vaan se on viitekehys ja projektinhallintamalli mahdollistamassa ohjelmistotuotteiden kehittämistä (Schwaber & Sutherland, 2017).

Perinteisien ohjelmistomenetelmien mittareita on kehitetty ja käytetty useita, mutta kaikki eivät sellaisenaan sovellu käytettäväksi ketterissä ohjelmistomenetelmissä, vaikkakin ketterissä prosesseissa on myös samoja metriikoita käytössä kuin perinteisissä menetelmissä (Padmini ym., 2015). Ketterien ohjelmistokehitysmenetelmien tueksi on myös kehitetty useita mittareita (Kupiainen ym., 2015; Padmini ym., 2015). Ketteristä menetelmistä Scrumissa prosessin

seurantaan esitetään usein työkaluksi ns. Burndown-kaaviota, mutta laajemmin menetelmäkuvauksissa ei mittarointiin kovin tarkasti mennä: esimerkiksi Scrum-oppaassa (Schwaber & Sutherland, 2017) mainitaan tuotosten tarkastelu ja edistymisen seuranta työmäärän osalta, mutta mittareista ei tarkemmin mainita mitään. DevOpsin kuvauksissa sen sijaan mittarointi on huomioitu paremmin, koska yhdeksi DevOpsin ulottuvuuksista on nostettu juuri mittarointi (Lwakatare ym., 2015).

Ketterien ohjelmistokehitysmenetelmien mittaroinnissa on otettava huomioon tarve soveltaa mittareita parhaiten tilanteeseen sopivaksi. Jos valitaan vain mittareita, jotka ovat useimmiten käytössä, ei ne välttämättä sovellu omaan tarkoitukseen ja kohdealueeseen (Misra & Omorodion, 2011). Esimerkiksi Scrum-menetelmän tapauksessa, kuten mainittu, menetelmä antaa paljon vapauksia itse toteutuksen osalta, joka voi vaikuttaa myös parhaiten eri tilanteisiin soveltuviin mittareihin. Eri tilanteissa käytettävät metriikat pitäisikin siten valita tiimin ja kohdealueen tarpeiden mukaan, ja valmiita mittareita voi soveltaa omiin tarpeisiinsa sopiviksi ja käyttää niitä niin, että ne mittaavat juuri jokaisessa tilanteessa toiminnan kannalta tärkeimpiä ja tarpeellisimpia suureita (Misra & Omorodion, 2011). Ketterän kehittämisen tiimit ovatkin käyttäneet omia mittareitaan tekemisen tukena (Kupiainen ym., 2015). Tutkimuksen näkökulmasta olisikin mielenkiintoista dokumentoida ohjelmistokehitystiimin mittareiden käyttöönottoprosessia nojaten olemassa oleviin mittareihin ja niiden tiimin kesken tapahtuvaan valintaan ja soveltamiseen omaan kohdealueeseen sopivaksi. Useissa ohjelmistomittareita listaavissa tutkimuksissa kerätään haastatteluilla jo käytössä olevia mittareita. Tämä tutkimus painottaakin mittareiden valitsemisen prosessia mittarilistauksien ohella.

Tämä tutkimus tarkastelee kohteena olevan Scrum-tiimin kehitysprosesseja ja niihin liittyviä mittareita. Tiimillä oli ollut Scrum ohjelmistokehitysmallina käytössä noin kaksi vuotta ja tutkimus oli osana tiimin tavoitetta paremmin seurata omaa toimintaansa ja sitä kautta mahdollistaa toiminnan jatkuva parantaminen. Tämän tutkimuksen tavoitteena oli yhteistyössä kohteena olevan Scrum-tiimin kanssa kuvata tiimin kehitysprosessia ja listata prosessia tukevia mittareita. Koska Scrum-mallin sisällä voi käyttää erilaisia varsinaisia ohjelmistokoodin kehittämisen tapoja ja prosesseja, vaikuttaa se siten myös siihen, mitä mittareita kannattaa prosessin tueksi ottaa. Näin ollen tutkimuksen osaksi on myös otettu tiimin kehitysmallin kuvaaminen, mikä samalla toimi johdatteluna ja pohjustuksena mittareiden keräämiselle, analysoinnille ja priorisoinnille.

Tutkimuksen tavoite voidaan jakaa kahteen tarkempaan kysymykseen tiimin toimintaan liittyen:

1. Mikä on tiimin käytössä oleva Scrum-prosessin mukainen kehitysprosessi?
2. Mitä mittareita tiimin näkökulmasta tulisi ottaa osaksi kehitysprosessin seurantaa?

Tutkimuksen tekijä on osana kehitystiimiä kehittäjän ja tiimin varalla olevan Scrum Masterin roolissa.

Tutkimuksen seuraavissa kolmessa luvussa esitetään tutkimuksen tieteellistä taustaa ja aihepiirin termejä. Luvussa 5 kuvataan tarkemmin tutkimuksen kohdealue ja menetelmät. Luvussa 6 esitetään tutkimuksen tulokset, luvussa 7 tuloksien tulkinta ja lopuksi luvussa 8 esitetään tutkimuksen johtopäätökset.

2 KETTERÄ KEHITTÄMINEN

Tässä luvussa käsitellään tieteellistä taustaa ketterästä kehittämisestä, sen historiasta ja periaatteista, hyödyistä, haitoista ja vaikutuksista. Erikseen esitellään ketterä ohjelmistokehitysmenetelmä Scrum ja varsinkin viime vuosina suosituksi noussut DevOps.

2.1 Mikä on ketterä kehittäminen?

Ketterällä kehittämisellä (engl. Agile development) viitataan ohjelmistokehityksen metodologioihin, jotka nousivat vaihtoehtoisiksi menetelmiksi ns. perinteisten menetelmien rinnalle. Perinteiset menetelmät kuten esimerkiksi vesiputousmalli (engl. Waterfall model) painottavat etukäteissuunnittelua ja järjestyksessä etenevää vaiheittaista järjestelmän kehittämistä. Järjestelmän vaatimukset kerätään alussa ja itse järjestelmä kehitetään tietyssä järjestyksessä, eikä prosessissa palata takaisin päin (Szalvay, 2004; Stoica ym., 2013). Nykyajan nopeasti muuttuvat liiketoimintaympäristö ja vaatimukset eivät suosi perinteisiä menetelmiä, mihin taas ketterät menetelmät pystyvät paremmin vastaamaan iteratiivisella, tehokkuutta ja kommunikaatiota painottavalla otteellaan (Szalvay, 2004; Nerur ym., 2005). Gillin ja Henderson-Sellersin (2006) määritelmä ketteryydestä ehdottaa, että se on kykyä sopeutua odotettuihin tai odottamattomiin muutoksiin, toimia lyhyellä aikajänteellä, hyödyntää laadukkaita, mutta yksinkertaisia työkaluja ja hyödyntää ajantasaista aikaisempaa tietämystä oppiakseen ympäristöstään.

Ketterän kehittämisen liikehdintä sai alkunsa vuonna 2001 julkaistulla Ketterän ohjelmistokehityksen julistuksella (engl. Manifesto for Agile Software Development) (Beck ym., 2001). Julistuksellaan sen kirjoittajat tavoittelivat vaihtoehtoa raskaalle ja dokumentaatiopainotteiselle tavalle kehittää ohjelmistoja (Beck ym., 2001). Julkaisussa he esittelivät ohjelmistokehitysnäkemyksensä neljä arvoa ja 12 periaatetta. Arvot suosivat ohjelmistokehittäjien välistä kommunikaatiota ja ylipäätään ihmisen roolia yli byrokraattisten prosessien ja työkalujen,

toimivan, yksinkertaisen mutta teknologialtaan kehittyneen ja testatun ohjelmiston säännöllistä julkaisua yli raskaan ohjelmiston dokumentoinnin, toimivaa ja vapaampaa asiakkaiden ja ohjelmistokehittäjien välistä kommunikaatiota ja yhteistyötä yli raskaiden ja tarkkojen sopimuspohjaisten keskusteluiden sekä ohjelmistokehityksen elinkaaren aikana syntyviin muutostarpeisiin vastaamista yli tarkan suunnitelman noudattamisen (Abrahamsson ym., 2002). Periaatteissa painotetaan muun muassa, että toimiva ohjelmisto on edistyksen tärkein mittari, tärkein prioriteetti on asiakastyytyväisyys, kommunikaatiossa painotetaan kasvokkain käytävää keskustelua ja ohjelmistokehittäjien ja liiketoimintaosaajien pitää tehdä päivittäistä yhteistyötä, parhaat tuotokset syntyvät itseohjautuvien tiimien kautta sekä suositaan yksinkertaisuutta (Beck ym., 2001). Yleisesti ottaen siis julistuksen mukainen ketterä kehittäminen ei keskity liikaa etukäteiseen, tiettyyn prosessiin kahlitsevaan suunnitteluun tai erilaisten dokumentaatioiden tekemiseen, vaan suunnittelu jaetaan osaksi iteratiivista kehitysprosessia. Näin ollen on mahdollista kehittää järjestelmiä iteratiivisesti suosien heti alusta vahvaa asiakkaisen osallistamista ja kasvokkain käytävää yhteistyötä ja keskustelua, jolloin on mahdollista reagoida hyvin muuttuviin vaatimuksiin (Serrador & Pinto, 2015).

Ketterä kehittäminen käsitellään usein erilaisten kehitysmenetelmien kautta. Abrahamsson ym. (2002) määrittelevät ohjelmistokehitysmenetelmän olevan ketterä, kun se on inkrementaalinen, yhteistyökykyinen, yksinkertainen ja joustava. Ketterä kehittäminen ja niiden menetelmät ovatkin saaneet aikaan suuren muutoksen niin ohjelmistokehityksen alalla kuin sitä tutkivassa tieteellisessä kirjallisuudessa (Dybå & Dingsøyr, 2008; Dingsøyr ym., 2012). Osa näistä menetelmistä oli kuitenkin kehitetty jo ennen Ketterän ohjelmistokehityksen julistuksen ilmestymistä (esim. Schwaber, 1997; Beck, 1999), ja monien eri menetelmien kehittäjiä olikin julistuksen tekijöinä (Beck ym., 2001). Julistuksen jälkeen menetelmiä on syntynyt lisää ja niitä on kehitetty edelleen. Esimerkiksi Abrahamsson ym. (2002) esittelevät erilaisia ketterän kehittämisen menetelmiä, joita ovat esimerkiksi Extreme programming (XP), Scrum, The rational Unified Process ja Open Source Software development. Myöhemmin kehitettyjä ketteriä menetelmiä on esimerkiksi suositut Lean software development, Kanban ja paljon yhteistä ketterien menetelmien kanssa omaava DevOps.

Ketterän kehittämisen hyötyjä ja haittoja on esitetty useita. Ketterään kehittämiseen siirryttäessä laajamittaisen ja ison organisaation muutos on osoittautunut hankalammaksi kuin pienten organisaatioiden ja ryhmien, mikä tukee yleistä näkemystä, että ketterä kehittäminen sopii paremmin pienille tiimeille (Dybå & Dingsøyr, 2008; Dikert ym., 2016). Ketterän kehittäminen voi kuitenkin toimia hyvin erilaisissa ympäristöissä ja organisaatioissa. Lisäksi asiakasyhteistyö mahdollistaa parempaa kommunikaatiota, mutta samalla se voi olla raskasta asiakkaana toimiville osapuolille (Dybå & Dingsøyr, 2008). Tiimeissä sen jäsenet saavat toimia vapaammin ja heillä on korkeampi työtyytyväisyys, mutta tiimien jäseniä voi olla hankala siirtää tiimistä toiseen, ja kulttuurin ja menetelmän omaksumisessa voi olla hankaluuksia. Työn tehokkuuden ja laadun on myös kerrottu nousevan (Dybå & Dingsøyr, 2008; Vijayasarathy & Turk, 2008).

Yksittäisten hyötyjen ja haittojen ohella on kuitenkin tärkeää myös selvittää, että onko ketterän kehittämisen menetelmillä vaikutuksia niitä käyttävien projektien onnistumiseen. Usean projektin kattavassa tutkimuksessa Serrador ja Pinto (2015) havaitsivat ketterien menetelmien vaikuttavan positiivisesti projektien onnistumiseen tarkasteltaessa projektien tehokkuutta ja asiakastyytyväisyyttä. Lisäksi he havaitsivat, että projektin visiolla ja tavoitteilla on vaikutusta ketterien menetelmien ja projektin onnistumisen väliseen suhteeseen, mutta toisaalta tiimin laaja kokemuspohja tai projektin monimutkaisuus ei näyttäisi olevan vaatimuksena projektien onnistumiselle (Serrador & Pinto, 2015). Käsiteltäessä projekteja, joissa siirrytään käyttämään ketterän kehittämisen menetelmiä, kriittisiä käyttöönoton onnistumisen tekijöitä on havaittu olevan esimerkiksi oikea ketterän menetelmän käyttöönoton prosessi ja sen kustomointi tarpeenmukaiseksi, tietyn menetelmän mukaisten prosessien oikea käyttö, tiimin osaaminen ja johdon tuki muutokselle (Chow & Cao, 2008; Dikert ym., 2016). Haasteita muutosprojektissa voi aiheuttaa menetelmän käyttöönoton hankaluus, kehityksen ulkopuolisten toimintojen liittäminen menetelmään ja osallistuvien muutosvastaisuus tai ohjeistavan kirjallisuuden puute (Dikert ym. 2016).

Ketterän kehittämisen menetelmät ovat nousseet ohjelmistokehitystä harjoittavien toimijoiden joukossa erittäin suosituiksi kehitysmenetelmiksi. Esimerkiksi vuosittain julkaistavassa annual State of Agile -raportissa (CollabNet VersionOne, 2019) loppuvuonna 2018 yli 90 prosenttia kyselyyn vastanneiden organisaatioista käyttävänsä ketteriä ohjelmistokehitysmenetelmiä. Raportista käy myös ilmi, että suosituimpia syitä ketteriin ohjelmistokehitysmenetelmiin siirtymisessä olivat julkaisutahdin nopeuttaminen, muuttuviin vaatimuksiin vastaaminen ja tuottavuuden parantaminen. Havaittuja toteutuneita hyötyjä siirtymisestä ketteriin menetelmiin raportoitiin olevan edellä mainittujen ohella tiimihengen parantuminen, läpinäkyvyys sekä liiketoiminnan ja IT:n linjaaminen (CollabNet VersionOne, 2019). Lähes samaan aikaan annual State of Agile -kyselyn kanssa suoritettiin kysely ketterän kehittämisen ja ketteryyden tilasta suomalaisissa yrityksissä. Tulokset ovat samansuuntaisia: 83 prosentilla vastanneista on joko suunnitteilla tai jo tehtynä siirtymä ketteriin menetelmiin, ja syitä siirtymille olivat esimerkiksi tuottavuuden parantaminen, kyky paremmin vastata muutoksiin sekä työtyytyväisyys (Kettunen ym., 2019).

2.2 Scrum

Ketteryysraporttien kokonaiskuvan ohella esitetään usein myös yksittäisen ketterien menetelmien käyttöä haasteltavien organisaatioissa. Menetelmistä suosituimmaksi on noussut Scrum: annual State of Agile -raportin mukaan 72 prosenttia käyttää Scrumia tai Scrum-johdannaista (CollabNet VersionOne, 2019). Suomalaisiin yrityksiin kohdistuneessa selvityksessä Rodriguez ym. (2012) raportoivat, että 83 prosentilla vastanneista oli organisaatiossaan käytössä Scrum. Näin ollen Scrum on noussut erittäin käytetyksi tavaksi toteuttaa ketterää oh-

jelmistokehittämistä, mutta Scrum saattaa soveltua myös käytettäväksi muilla aloilla (esim. Streule ym., 2016; Hernández ym., 2019).

Terminä Scrum viittaa rugbyssä käytettävään aloitusmuodostelmaan ja tieteellisessä kirjallisuudessa sen ensimmäisen kerran esittelivät Takeuchi ja Nonaka (1986). He käyttivät termiä kuvaamaan uutta tuotekehityksen tapaa, jossa vanhantyyppisen peräkkäisten vaiheiden prosessimallin sijaan tuotekehitystä voisi tehdä rinnakkaisten ja yhteistyötä tekevien prosessien ja tiimien kesken. Prosessissa on ominaista muun muassa itseohjautuvuus, limittäiset kehitysvaiheet ja vähäinen valvonta (Takeuchi & Nonaka, 1986). Scrum ketterän ohjelmistokehityksen käsitteenä sisältää samoja ajatuksia mitä jo Takeuchi ja Nonaka (1986) esittelivät. Ohjelmistokehityksen kontekstissa Scrumin ensimmäistä kertaa esittelivät Schwaber ja Sutherland vuonna 1995 (Schwaber & Sutherland, 2017).

Schwaber ja Sutherland (2017) määrittelevät Scrumin olevan viitekehys, jonka avulla kehittää, toimittaa ja ylläpitää monimutkaisia tuotteita. Ohjelmistokehitysprosessi on monimutkainen, monivaiheinen ja siihen vaikuttavat monet muuttuvat ympäristötekijät. Näin ollen kehitysprosessin pitää olla joustava (Schwaber, 1997). Scrum on empiirinen, iteratiivinen ja inkrementaalinen ohjelmistokehityksen viitekehys, jossa ei suoraan määritellä itse kehityksen menetelmiä, vaan sen sisällä voi käyttää useita menetelmiä ja prosesseja (Schwaber, 1997; Schwaber & Sutherland, 2017). Joustavuus on Scumin ytimessä: muutostarpeita ei käsitellä vain kehitysprosessin alussa, vaan muutoksia voi tehdä missä vain kehityksen vaiheessa (Schwaber, 1997). Scrum sopii siten varsin hyvin nykypäivän nopeasti muuttuvaan toimintaympäristöön.

Scrumin viitekehyksessä määritellään kehityksen säännöt tapahtumina, tuotoksina ja erityisen Scrum-tiimin rooleina. Scrumin tapahtumat voidaan jakaa kolmeen osaryhmään: ennen toteutusta tehtävä suunnittelu (pregame) kuten työjonon koontia ja ylemmän tason arkkitehtuurisuunnittelua, itse toteutus eli sprintti (game) ja toteutuksen jälkeiset toimet kuten julkaisudokumenttien tekeminen ja järjestelmän integrointi kohteeseen (postgame) (Schwaber, 1997). Scrumin tapahtumista on keskiössä sprintti, joka on iteratiivinen ja aikarajattu kehitysjakso, jonka puitteissa tuotetaan potentiaalisesti julkaisukelpoinen inkrementti (Schwaber & Sutherland, 2017). Sprintti koostaa yhteen Scrumin toteutusvaiheen tapahtumat, joita ovat sprintin suunnittelu, päivittäispalaveri (daily), sprintin katselmointi ja sprintin retrospektiivi (Schwaber & Sutherland, 2017). Tapahtumat siten vievät eteenpäin itse kehitystyötä ja sen hallinnoimista. Sprintin suunnittelupalaverissa suunnitellaan Scrum-tiimin kesken tulevassa sprintissä tehtävä työ. Sprintin aikana päivittäin pidetään päivittäispalaveri, jossa käydään läpi edellisen päivän töiden eteneminen ja suunnitellaan tulevain päivän työt, kaikki enintään 15 minuutin aikana. Sprintin katselmoinnissa käydään läpi sprintin aikana tehty työ, inkrementti, ja sen mukaan tarpeen vaatiessa sopeutetaan työjonoa tulevalle sprintille. Sprintin lopuksi pidetään vielä retrospektiivi, jossa Scrum-tiimi voi tarkastella sprintin toimintaansa ja siten yrittää parantaa omaa kehitysprosessiaan (Schwaber & Sutherland, 2017).

Scrumin tuotoksia ovat jo mainitun inkrementin ohella tuotteen ja sprintin kehitysjonot (Schwaber & Sutherland, 2017). Tuotteen kehitysjono on aina kyseisellä hetkellä oleva tietämys siitä, mitä tuotteesta tullaan tulevaisuudessa toteuttamaan. Tuotteen kehitysjonoa muokataan jatkuvasti tuotteen ja tietämyksen kehityksen myötä ja sitä pitää siten myös jalostaa jatkuvasti. Olennainen osa tuotejonoa ja jalostamista on yksittäisten tehtävien priorisoiminen. Sprintin kehitysjono koostuu tietylle sprintille tuotteen kehitysjonosta otetuista yksittäisistä tehtävistä, jotka ovat ikään kuin ennuste siitä, mitä sprintin aikana tullaan toteuttamaan. Niin sprintin kuin tuotteen kehitysjonojen kehitystyön määrää ja edistymistä on hyvä seurata esimerkiksi ns. burndown-kaavioiden avulla, mikä kertoo jäljellä olevan työn määrää suhteessa jäljellä olevaan aikaan. Sprinttien myötä tuotteen inkrementti kasvaa sprintissä kuvattujen kehitysjonotehtävien verran ja lähestyy iteratiivisesti sille tuotejonossa määritettyä tuotteen kehitysjonon mukaista tavoitetta.

Scrum-tiimin jäsenet on kuvattu Scrumin erilaisina rooleina, joita ovat tuoteomistaja (Product Owner, PO), Scrum Master (SM) ja kehitystiimi (Schwaber ja Sutherland, 2017). Tiimi kuvataan usein itseohjautuvana ja monitaitoisena: tiimi ei ole riippuvainen ulkopuolisista toimijoista osaamisen tai käytettävien menetelmien suhteen, vaan päätäntävalta, miten työnsä tehdä, on tiimillä. Tuoteomistaja on henkilö, joka kommunikoi asiakkaiden ja loppukäyttäjien kanssa oman tuotteensa vaatimuksista, kehityksestä ja tuotoksista. Tuoteomistaja luo, ylläpitää ja on vastuussa asiakkaiden toiveiden pohjalta luodusta tuotteensa kehitysjonosta. Kehitystiimi on vastuussa kehitysjonon mukaisten tehtävien toteuttamisesta ja inkrementin kehittämisestä. Scrum Master on vastuussa Scrumin mukaisen prosessin toteutumisesta ja siten ohjeistaa ja kouluttaa tiimiläisiä ja sidosryhmien edustajia toimimaan Scrumin periaatteiden mukaisesti.

Scrum on siis noussut yhdeksi eniten käytetyimmistä ketterän kehittämisen menetelmistä. Mutta miten Scrum sopii ketterän kehittämisen yleisiin ajatuksiin esimerkiksi Ketterän ohjelmistokehityksen julistuksen osalta? Qumer ja Henderson-Sellers (2008) analysoivat Scrumin ketteryyttä kehittämällään viitekehyksellä. He käsittelivät Scrumin ketteryyttä neljästä eri näkökulmasta. Scrumin skaalan osalta Scrum soveltuu minkäkokoisille projekteille vaan ja tiimejä voi olla useita, mutta yhden tiimin koko on alle 10 henkilöä. Scrumin ketteryyden asteen osalta postgame-vaihe ei saa hyviä arvosanoja, eikä Lean-ajattelu näy mukana Scrumissa, muutoin ketteryyden aste on hyvä. Scrum käytänteet ovat varsin ketteriä, mutta nekään eivät ole yhteydessä Lean-ajatteluun. Lisäksi käytänteet liittyvät kehityksen ja projektin hallinnan prosesseihin, mutta eivät niinkään konfiguraation tai prosessin hallintaan (Qumer & Henderson-Sellers, 2008). Tuloksissa näkyy esimerkiksi Scrumin tarkoitus jättää itse kehityksen prosessit kuvaamatta ja toisaalta Scrum ei suoraan ota kantaa kustannustehokkuuteen viitekehyksen kuvauksen osissa. Toinen mielenkiintoinen kysymys on, että miksi Scrum toimii, koska eihän se muuten olisi niin suosittu? Pries-Heje ja Pries-Heje (2011) esittävät havaitsemiaan syitä kysymykseen miksi Scrum on toimiva ketterän projektihallinnan menetelmä. Heidän mukaansa Scrum toimii esimerkiksi, koska Scrum mahdollistaa verkostoitumista ja kasvat-

taa luottamusta, Scrum luo yhteisen kielen ja tavoitteen tiimille sekä rakenteen tapaamisille, Scrum mahdollistaa työn ja laadun hallinnan ja seurannan.

Schwaber ja Beedle (2002) jakavat Scrumin käyttöönottoprojektit kahteen tyyppiin: Scrumin käyttöönotto uuteen ja jo olemassa olevaan projektiin (Schwaber & Beedle, 2002, s. 57-59). Uuden projektin tapauksessa alussa luodaan aloituskehitysjono, jonka pohjalta luodaan alustava järjestelmäkehys. Ensimmäisen sprintin tarkoitus on pystyttää kehys ja tehdä ensimmäinen toimiva kokonaisuus järjestelmään, jonka voi esitellä asiakkaalle. Samaan aikaan kehityksen kanssa tuoteomistaja ja asiakas laajentavat kehitysjonoa. Näiden pohjalta syntyy tuotteen kehitysjono tuleville sprinteille. (Schwaber & Beedle, 2002). Jo olemassa olevan projektin muuttaminen Scrumia käyttäväksi alkaa usein tilanteella, jossa kehitysympäristö ja teknologia ovat jo käytössä, mutta tiimillä on ongelmia muuttuvien vaatimusten ja vaikea teknologian kanssa (Schwaber & Beedle, 2002). Scrumin käyttöönotto voi kyseisessä tapauksessa alkaa ottamalla käyttöön päivittäispalaverit, jotka voivat kertoa ongelmista. Toimintaa voi siten suunnata asiakkaan tärkeimpien toiveiden mukaiseksi. Ensimmäisen sprintin tavoite on esittää toimiva toiminnallisuus järjestelmästä, ja sprintin lopuksi päätetään mitä tehdään seuraavaksi (Schwaber & Beedle, 2002).

Scrum on siis suosituin käytetyistä ketteristä menetelmistä ja sitä käyttävien tiimien määrä todennäköisesti vain kasvaa uusien käyttöönottoprojektien myötä: State of Agile -raportissa (CollabNet VersionOne, 2019) mainitaan, että 97 prosentissa kyselyyn vastanneista organisaatioista oli ketterän kehittämisen menetelmä käytössä, mutta samalla 78 prosentilla vastanneista vain osa tiimeistä on ketteriä. Scrumin käyttöönotto ja itse Scrumin mukainen kehittäminen ei aina kuitenkaan onnistu ilman ongelmia, ja projektien onnistumisen kannalta on hyvä kiinnittää huomiota havaittuihin Scrumin ongelmiin, jotta niitä voitaisiin välttää tai niistä toipua parhaan mukaan. Akif ja Majeed (2012) esittävät useita havaitsemiaan Scrumin ongelmia, ja miten niitä korjata. Esimerkiksi järjestelmän ja koodin laatu voi kärsiä, koska lyhyiden sprinttien jälkeen on aina jotain saatava valmiiksi. Tätä voi kirjoittajien mukaan korjata keskittymällä ja panostamalla enemmän myös laadullisiin asioihin kehityksen aikana (Akif & Majeed, 2012). Samasta syystä ongelmia voi myös ilmetä uusien osien integroimisessa ja julkaisujen hallinnassa, johon myös on kiinnitettävä huomiota kehityksen aikana. Muita esitettyjä ongelmia olivat esimerkiksi sprinttien joko liian lyhyt tai pitkä kesto, puutteet Scrum-prosessin tuntemuksessa, jättämällä metriikat, kuten burndown-kaaviot, hyödyntämättä ja kehitysjonon hallinnan puutteet esimerkiksi rakenteen ja dokumentaation osalta (Akif & Majeed, 2012). Akif ja Majeed (2012) mainitsevat lisäksi ongelmaksi monitiimitilanteen, jossa usea tiimi työskentelee samassa projektissa. Marchenko ja Abrahamsson (2008) tutkivat usean Scrum tiimin ja usean kehitysprojektin tilannetta, ja löysivät useita ongelmakohtia. Näitä olivat esimerkiksi liian tarkka tai väljä Scrum-prosessin noudattaminen, bugikorjausten ja ylläpitotöiden suuri määrä, liika erikoistuminen tiettyihin tehtäviin, liian täydet sprintit ja ongelmat työn edistymisen seurannassa.

2.3 DevOps

Ketterät ohjelmistokehitysmenetelmät, kuten Scrum, keskittyvät yhdessä kehittäjien ja asiakkaiden kanssa julkaisemaan lyhyessä ajassa vähittäisiä muutoksia kehitettävään ohjelmistoon. Painotus on siten ohjelmistokehityksessä eikä niinkään sen julkaisemisessa tai ylläpidossa: jokaisen vaiheen jälkeen voidaan julkaista uusi ohjelmistoversio, joka esitellään asiakkaalle, mutta prosessissa ei oteta kantaa mitä asiakas tekee tuotoksella. Varsinainen ohjelmiston arvo asiakkaalle, eli tuotantoympäristöön loppukäyttäjille asti viety toimiva ohjelmisto, jää kyseisenlaisessa tilanteessa vähemmälle huomiolle. Lisäksi usein ketterien menetelmien lyhyt ja tarkasti vaiheittaisesti määrätty julkaisutahti on liian nopea ja rajattu ylläpidolle, eikä ketterän kehityksen nopean tahdin hyödyt siten suoraan näy loppukäyttäjille (Hüttermann, 2012 s. 23).

DevOps tulee englannin kielen ohjelmistokehitystä (development) ja ylläpitoa (operations) kuvaavista sanoista, joka tuli käyttöön terminä vuoden 2009 aikoihin (Mezak, 2018). Ohjelmistokehityksellä viitataan tässä tapauksessa prosesseihin, joilla määritellään, suunnitellaan, toteutetaan, testataan ja integroidaan ohjelmistoja tai niiden komponentteja. Ylläpidolla taas tarkoitetaan mm. ohjelmistojen asennusta, päivittämistä, hallintaa, valvontaa ja käyttäjien tukemista tuotantoympäristössä (Lwakatare, 2017). Ohjelmistokehitys ja ylläpito ovat siis usein eri tiimien vastuulla, ja tiimit eivät välttämättä ole edes samassa yrityksessä: ylläpito voi olla ulkoistettu toisen yrityksen vastuulle ja toisaalta ohjelmisto voi olla ostettu ulkopuolelta, jolloin ohjelmistokehitys on ulkoistettua ja lopullinen tuota tulee ostavan yrityksen ylläpidettäväksi. Eri tiimit tarkoittavat myös eri tavoitteita: ohjelmistokehityksessä painotus on muutoksessa ja ylläpidon osalta taas pysyvyydessä. Eri tavoitteet voivat taas johtaa tiimien välisiin konflikteihin (Hüttermann, 2012, s. 15-31). Kommunikaatio ja yhteistyö ohjelmistokehitys- ja ylläpitotiimien välillä voi olla ongelmallista ja huonosti hallittua (Tessem & Iden, 2008). DevOpsin tavoitteena onkin yhdistää näiden kahden tiimin jäseniä ja parantaa kommunikaatiota ja yhteistyötä esimerkiksi linjaamalla tiimien tavoitteita yhtenevämmiksi (Hüttermann 2012, s. 27).

DevOpsilla on monia määritelmiä, mutta ei yhteistä näkemystä, että mitä termillä tarkoitetaan tai mitä se sisältää (Lwakatare, 2017). Lwakatare (2017) jakaa määritelmät tavoitepainotteisiin tai keinopainotteisiin määritelmiin. Esimerkiksi Bass ym. (2015) määrittelevät DevOpsin tavoitepainotteisesti olevan kokoelma käytäntöjä, joilla tavoitellaan lyhyempää ohjelmistomuutosväliä laadusta tinkimättä. Esimerkki keinopainotteisesta määritelmästä on Jabbari ym. (2016) määritelmä, jossa DevOps kuvataan olevan kehitysmetodologia ohjelmistokehityksen ja ylläpidon etäisyyden lyhentämiseksi painottaen kommunikaatiota, yhteistyötä, jatkuvaa integraatiota, laadunvarmistusta, automaatiota ja kehitysmenetelmien käyttöä. Toisaalta DevOps voidaan määritellä eräänlaisena työnkuvana tai roolina organisaatiossa, jossa työnkuvaan kuuluu toimia niin ohjelmistokehityksen kuin ylläpidon alueelta. Tälle vastamääritelmä taas ei näe DevOpsia roolina, vaan eräänlaisina vaatimuksina, jotka pitää ottaa huomioon

ohjelmistokehityksen laajassa kuvassa kuten itse kehityksessä ja myös testauksessa, julkaisemisessa, tuessa ja mittaamisessa (Roche, 2013). Kokonaisuutena eri määritelmissä painottuu kuitenkin juuri ohjelmistokehityksen ja ylläpidon välisen yhteistyön parantaminen eri tavoilla, jolla haetaan tehostusta ohjelmistomuutosten saamisessa asiakkaiden käyttöön.

DevOpsin määritelmissä usein mainitaan erilaiset käytännöt tai tavat, joilla yritetään parantaa yhteistyötä ohjelmistokehityksen ja ylläpidon välillä. Jabbari ym. (2016) kuvaavat kirjallisuudesta kootusti erilaisia DevOpsin käytänteitä ohjelmistokehityksen eri vaiheista. Käytänteitä ovat esimerkiksi jatkuva suunnittelu, jatkuva integrointi, jatkuva monitorointi, jatkuva ja automaattinen julkaiseminen, palautteen antaminen kehittäjien ja ylläpitäjien välillä, yhtenäinen muutosten ja konfiguraation hallinta, vaatimusten määrittely ja sidosryhmien osallistuminen (Jabbari ym., 2016).

DevOpsia voidaan myös kuvata neljän eri ulottuvuuden kautta: yhteistyö, automaatio, mittaaminen ja valvonta (Lwakatare ym., 2015). Yhteistyö viittaa juuri ohjelmistokehittäjien väliseen yhteistyöhön ja sellaisen kulttuurin rakentamiseen, joka suosii tällaista yhteistyötä. Automaatio viittaa ohjelmistokoodin automaattiseen koostamiseen, testaamiseen ja julkaisemiseen halutulle alustalle ja paikkaan. Mittaamisella viitataan ohjelmistokehitysprosessin kannalta oleellisten tekijöiden mittaamista. Monitorointi taas koskee ohjelmistokehityksen eri vaiheiden monitorointia erilaisten työkalujen avulla.

DevOps ei ole varsinaisesti ohjelmistokehitysmenetelmä tai suoranaisesti ketterä menetelmä. DevOpsin voi nähdä kuitenkin laajentavan ja tukevan ketteriä ohjelmistomenetelmiä, kuten esimerkiksi tuomalla mukanaan ohjelmistokehityksen automatiikkaa ja monitorointia sekä painotusta yhteistyöhön ja kommunikaatioon (Jabbari ym., 2016). Jabbari ym. (2016) listaavat DevOpsin ja ketterien menetelmien käytänteitä, missä näkyy esimerkiksi jatkuvan integraation ja julkaisemisen esiintyminen molemmilla puolilla.

DevOps on nousemassa varsin suosituksi ohjelmistoyritysten keskuudessa, mikä on huomioitu myös annual State of Agile -raportissa (CollabNet VersionOne, 2019): 73 prosentilla vastanneiden organisaatioista on meneillään DevOps-aloite ja 90 prosenttia näki DevOps-muutoksen organisaatiossaan tärkeänä asiana. Tärkeimmiksi mittareiksi DevOpsin onnistumiselle olivat julkaisutahdin nopeutuminen ja parantunut laatu (CollabNet VersionOne, 2019). DevOps on siten nousemassa varsin tärkeäksi osaksi ketterää ohjelmistokehitystä Scrumin ohella. Aika näyttää kuinka DevOps-käyttöönotot onnistuvat ja kuinka jäädäkseen DevOps on tullut osana ketterää ohjelmistokehitystä.

3 JATKUVA OHJELMISTOKEHITYS JA JULKAISUT

DevOpsin yhtenä ulottuvuutena on automaatio, joka nostaa keskiöön uusien koodi-inkrementtien automaattista käsittelyä esimerkiksi testaamisen ja integroinnin osalta. Automaatioon kyseisessä kontekstissa liittyvät termit ja käsitteet eivät välttämättä ole kovinkaan uusia, mutta DevOpsin myötä niihin on alettu kiinnittää enemmän huomiota. Kuten jo edellä mainittu, Scrum ei ota kantaa inkrementtien integroimiseen olemassa olevaan, mikä taas on tärkeä osa DevOpsia.

Automaatioon ja siinä käytettävien kokonaisuuksien käsittelyyn liittyy monia eri konsepteja, jotka usein niputetaan jatkuvan toiminnan alle: jatkuvasti tuotetaan uutta ohjelmiston osaa, jota jatkuvasti testataan ja asennetaan sekä joka vaatii jatkuvaa hallintaa (Fitzgerald & Stol, 2014). Jatkuvien toimintojen myötä voidaan saada esimerkiksi nopeammin palautetta kehityksen kulusta ja järjestelmän toimivuudesta, järjestelmän laatu voi parantua ja asiakastyytyväisyys nousta sekä kehitys- ja ylläpitotiimien yhteistyö voi parantua (Shahin ym., 2017). Alla tarkemmin esiteltynä termistöä muutamista jatkuvan toiminnon keskeisimmistä konsepteista.

3.1 Jatkuva integraatio

Ohjelmistokehitysprojektissa on usein tapana, että usea henkilö työstää saman ohjelman eri osia samaan aikaan työlle tehtyjen tarkennettujen vaatimusten mukaisesti. Jotta ohjelmasta saadaan toimiva sisältäen kaikki eri osat, on ne luonnollisesti yhdistettävä toisiinsa eli integroitava toimivaksi kokonaisuudeksi. Integroinnin voi kärjistetysti jakaa kahteen erilaiseen toteutustapaan: koodin toteutus ensin ja integroinnit omana projektinaan kehityksen loppuvaiheessa tai jatkuva koodin integrointi osana päivittäistä kooditoteutusta (Humble & Farley, 2011). Vasta loppuvaiheessa tehtävän integroinnin näkökulmasta ongelmallista on, että ohjelmakoodi kokonaisuutena on suurimman osan ajasta toimimattomassa tilassa, ja vasta lopussa, kun eri osia integroidaan kokonaisuudeksi, voi

ilmetä ongelmia, jotka olisi ollut helpompi ja nopeampi muuttaa kesken toteutuksen. Jatkuvan integroinnin tapauksessa järjestelmä on suurimman osan ajasta toimivassa tilassa, koska toteutuksen integrointi tehdään jopa useita kertoja päivässä, jolloin usein automaattisesti testataan lisätyn kooditoteutuksen toimivuus suhteessa muihin osiin järjestelmää. Tämä tosin vaatii ongelmien ilmetessä resursseja integrointiin läpi koko toteutuksen. Erot kahden näkökannan välissä voidaan tiivistää niin, että loppuvaiheessa tapahtuvassa integroinnissa ohjelmisto on toimimattomassa tilassa, ellei joku todista, että se toimii, ja taas jatkuvan integraation tapauksessa ohjelmisto on todistetusti toimivassa tilassa, ellei automatiikka ilmoita virheistä, jotka tulevat heti ilmi integroinnin yhteydessä (Humble & Farley, 2011).

Jatkuva integraatio (engl. continuous integration) oli jo osana ketterän ohjelmistokehitysmenetelmän Extreme programming (XP) käytänteitä vuosituhannen vaihteessa. XP:n käytänteissä jatkuva integraatio määriteltiin siten, että se on uuden koodin integrointia osaksi muuta järjestelmää muutaman tunnin välein niin, että lisäyksen yhteydessä tehtävien testien on mentävä läpi, tai muuten integrointi hylätään (Beck, 1999).

Jatkuva integrointia voidaan kuvata tarkemmin erilaisten käytänteiden ja integrointiprosessin osien kautta. Fowler (2006) esittää kymmenen käytäntöä kuvaamaan tehokasta jatkuvan integroinnin prosessia. Tärkeänä osana jatkuvaa integrointia on keskitetty versionhallinta, mistä kehittäjät noutavat koodin muutoksia varten, ja minne uusi koodi integroidaan. Koodi tulisi pystyä kääntämään toimivaksi ohjelmaksi automaattisesti, jolloin se voi toimia automaattisena testinä itsessään, että järjestelmä on kokonaisuutena toimiva uusien integrointienkin jälkeen. Lisäksi osaksi koodia ja automaattista buildiä tulisi luoda automatisoidut testit, jotka varmistavat koodin sisäistä toimintalogiikkaa toimivan buildin ohella. Kaikkien kehittäjien tulisi integroida koodi versionhallintaan joka päivä, jotta se voidaan testata, ja testaus pitäisi tapahtua erillisellä niin sanotulla buildipalvelimella, joka varmistaa ohjelmiston toimivuuden eri ympäristössä, kuin missä se on kehitetty. Buildit tulisi myös koittaa pitää nopeina, jotta palaute mahdollisista virheistä tulisi myös nopeasti, eikä esimerkiksi vasta seuraavana päivänä. Kaikkien kehittäjien tulisi helposti pystyä hakemaan uusimmat muutokset ja olla tietoisia uusista integrointiajoista esimerkiksi erillisen buildien tilan näyttävän monitorin tai nettisivun kautta. Toimivan koodi tulisi myös automaattisesti asentaa eri kohdeympäristöihin, jotka ainakin testiympäristön osalta muistuttavat mahdollisimman paljon tuotantoympäristöä. Fowler (2006). Näin pystyy testaamaan koko ohjelmiston toimivuuden osana jatkuvaa kehitystä ja tulevaa lopullista toimitusympäristöä.

Jatkuvan integroinnin hyötyinä on ohjelmistokehityksen prosessi, jonka myötä järjestelmän tila on paremmin selvillä, ongelmiin päästään kiinni nopeasti ja järjestelmän omistajille ja kehityksen johtajille tulee ajankohtaista tietoa kehityksestä. Prosessi myös pakottaa ottamaan huomioon testauksen osana kehitystä. Lisäksi se helpottaa bugien löytymistä ja mahdollistaa osaltaan järjestelmän jatkuvan toimituksen eri ympäristöihin. (Fowler, 2006; Humble & Farley, 2011.)

3.2 Jatkuva toimitus ja jatkuva käyttöönotto

Ketterien ohjelmistokehitysmenetelmien myötä asiakkaat ovat säännöllisesti mukana osana kehitystä ja pystyvät paremmin vaikuttamaan tuleviin integroitaviin järjestelmän osiin. Asiakkaat eivät kuitenkaan välttämättä tyydy katselmointeihin tehdyistä muutoksista vaan antaakseen paremmin palautetta kehityksestä, heidän pitäisi päästä itse käyttämään päivitettyä järjestelmää nopeasti päivitysten jälkeen (Virmani, 2015). Järjestelmä pitäisi siten saada asennettua asiakkaan ympäristöön aina uusien päivitysten jälkeen, jotta asiakkaat voisivat sitä testata ja antaa palautetta. Järjestelmän nopea toimitus asiakkaan ympäristöihin voidaan nähdä jatkumona jatkuvalle integraatiolle, jolloin jatkuvan integraatioprosessin myötä järjestelmä on ajan tasalla ja toimivana versionhallinnassa, josta se voitaisiin siirtää varsin nopealla aikataululla haluttuun ympäristöön ja asentaa sinne.

Jatkuva järjestelmän potentiaalinen siirtäminen asiakkaan ympäristöihin voidaan jakaa kahteen eri termiin: jatkuva toimitus (continuous delivery) tavoittelee kehitettävälle järjestelmälle tilaa, jossa siitä voitaisiin ottaa koska vain asennuspaketti ja asentaa se tuotantoympäristöön, kun taas jatkuva käyttöönotto (continuous deployment) vie toimituksen idean vielä hieman pidemmälle, jolloin järjestelmään tehdyt päivitysintegraatio viedään aina automaattisesti asiakkaan tuotantoympäristöön ilman erillistä tuotantoonvientipäätöstä, mikä taas vaaditaan tuotantoasennuksissa jatkuvan toimituksen osalta (Shahin ym., 2017). Aina ei ole mahdollista tehdä automaattista asennusta asiakkaan ympäristöön esimerkiksi pääsyrajoitusten, tietoturvan tai järjestelmätyypin takia, jolloin jatkuva käyttöönotto ei ole välttämättä mahdollista. Toisaalta asiakkaan vaatimukset voivat myös määritellä sen, että haluavatko he jatkuvasti päivitystoimituksia järjestelmäänsä vai vain erikseen sovittuina ajanhetkinä. Mikäli halutaan jatkuva käyttöönotto, mahdollistaa se todella nopeat uusien toiminnallisuuksien tai virhekorjausten päivityksen asiakkaan ympäristöön, mutta samalla se vaatii enemmän työtä, jotta voidaan varmistaa toimiva julkaisuprosessi ja integraatiot alustojen ja järjestelmien välillä.

Jatkuvan toimituksen ja käyttöönoton myötä mahdollistetaan nopeat toimitusajat uusille järjestelmäpäivityksille, mutta siirtyminen käyttämään näitä prosesseja ei välttämättä ole helppoa. Jatkuva toimitus ja käyttöönotto voidaan siis nähdä jatkuvan integraation jatkeena. Tässä siirtymässä voi tulla eteen useita mahdollisia ongelmia, varsinkin kun on tavoitteena automaattinen käyttöönotto. Esimerkiksi järjestelmän ja kohdeympäristön verkkojen konfiguraatioiden toimivuus voi tai kehitysprosessin läpinäkyvyys eri projektin jäsenille voi tuottaa ongelmia (Olsson, 2012). Muita ongelmia kyseisessä siirtymisessä voi olla jatkuvan käyttöönoton rasitteet palvelimille, koska toimituksia voi tulle useita kertoja päivässä (Claps ym., 2015). Ongelmia yleisesti jatkuvan käyttöönoton hyödyntämisessä voi olla tarve saada organisaatioon sisällytettyä Lean-ajattelua, joka helpottaa siirtymistä, ja toisaalta siirtymällä pitää olla niin kehitystiimien kuin johdon tuki onnistuakseen. Myös mahdolliset vastuumuutokset voivat

vaikuttaa, jos esimerkiksi kehitystiimin jäsenille tulee lisävastuita käyttöönoton hallinnassa (Claps ym., 2015). Jatkuvan toimituksen näkökulmasta ongelmia voi aiheuttaa esimerkiksi eri sidosryhmien vastustus yrityskulttuurin muutoksessa tai teknologisten muutosten tarve mahdollistamaan jatkuva toimitus (Chen, 2015), sekä järjestelmän näkökulmasta arkkitehtuuriin, jatkuvaan integrointiin sekä testaukseen liittyvät ongelmat voivat olla hidastamassa tai esteenä jatkuvalle toimitukselle (Laukkanen ym., 2017). Havaittuihin ongelmiin jatkuvan toimituksen ja käyttöönoton yhteydessä on myös etsitty ratkaisuja, joita esimerkiksi yrityskulttuurin muutoksen osalta voi olla jatkuvan toimituksen jatkuva myyminen eri sidosryhmille tai järjestelmän näkökulmasta esimerkiksi järjestelmän modularisointi ja mahdollisuus helposti palata takaisin edelliseen toimivaan versioon (Chen, 2017; Laukkanen ym., 2017).

Jatkuvan toimituksen ja käyttöönoton mainituissa hyödyissä on samoja, mitä aikaisemmin mainittu jatkuvien toimintojen yhteydessä. Hyötyjä on jatkuvan toimituksen osalta esimerkiksi nopeutettu ohjelmiston toimitus ja kehitysjonotehtävien suoritus, parempi palaute järjestelmän toiminnasta, parantunut työteho ja tuottavuus, parantunut järjestelmän ja julkaisujen laatu sekä kasvanut asiakastyytyväisyys (Chen, 2015). Jatkuvan käyttöönoton yhteydessä hyödyiksi mainitaan nopeutunut palaute järjestelmän toiminnasta, tihentyneet julkaisuvälit, koodin parantunut laatu, kasvanut asiakastyytyväisyys, parempi tuottavuus sekä tiiviimpi yhteistyö kehittäjien ja ylläpitäjien välillä (Leppänen ym., 2015).

3.3 Jatkuva testaus

Jatkuvan integraation olennainen osa on testaus, joka usein tapahtuu automaattisesti esimerkiksi versionhallintatyökalun avulla. Testaus voidaan siten nähdä myös jatkuvana toimintona, jatkuva testaus (continuous testing) (Saff & Ernst, 2003). Perinteinen, vesiputousmallipohjainen näkemys ohjelmistotestaukseen on, että se suoritetaan omana vaiheenaan, kun järjestelmä on muutoin ohjelmistokoodin osalta valmis. Ketterissä ohjelmistokehitysmenetelmissä testausta suoritetaan iteratiivisesti kehityksen osana. Jatkuva testaus on ohjelmistokoodiin kehityksen aikana lisättyjen testien ajamista esimerkiksi versionhallintaohjelmiston kautta aina kun muutoksia integroidaan olemassa olevaan, jolloin kehittäjän ei itse tarvitse testejä varsinaisesti ajaa ja testausta voidaan tehdä ennalta määrättyjen konfiguraatioiden mukaisesti (Saff & Ernst, 2003). Jatkuvan testauksena avulla pyritään siirtämään testaus mahdollisimman lähelle kehitystyötä pois vain loppuvaiheen erillisestä prosessivaiheesta (Fitzgerald & Stol, 2014). Jatkuva testaus on myös osana laajempaa ohjelmiston laadunhallintaa ja testausta, jossa automaattisten ohjelmistokooditestien, kuten yksikkö- ja integraatiotestit, ohella varmistetaan ohjelman toimivuus esimerkiksi tuottajaorganisaation testausryhmän toimesta ja asiakkaan hyväksymistestien kautta.

Jatkuvan, automaattinen testauksen käyttöönotto organisaation ohjelmistoprojekteissa vaatii muutosta kehityksen luonteessa, jolloin testaus otetaan osaksi muuta ohjelmistokoodin kehitystä ja järjestelmien suunnittelussa ja arkkitehtuurissa otetaan huomioon koodin testattavuus. Jatkuvaan testaukseen siirtyminen esimerkiksi jatkuvaa käyttöönottoa ajatellen voi siten vaati paljon resursseja (Rodríguez ym., 2017). Jatkuvan testauksen tavoitteina on esimerkiksi mahdollistaa virheiden aikaisempi löytyminen ja niiden poistaminen järjestelmästä (Fitzgerald & Stol, 2014). Automaattinen testaus voi myös parantaa ohjelman laatua, helpottaa virheiden korjausta, kun tehdyt muutokset ovat vielä kehittäjän muistissa ja voi ylipäätään nopeuttaa järjestelmän kehitysaikaa (Saff & Ernst, 2003; Fitzgerald & Stol, 2014; Rodríguez ym., 2017).

3.4 Julkaisut ja julkaisun hallinta

Ketterän ohjelmistokehityksen kulmakiviä ovat kehityksen iteratiivisuus ja inkrementaalisuus: kehitettävä ohjelmistotyö pilkotaan osiin ja jaksoissa kehitetään ohjelmiston osia, joita liitetään osaksi yhteistä ja yhtenäistä kokonaisuutta. Yksittäiset osat koostuvat toiminnallisuuksista (engl. feature), jotka integroituna kokonaisuuteen muodostavat (ohjelmisto)julkaisun (engl. release) (Ruhe, 2005). Julkaisut vaativat pohjaksi työn, jossa ohjelmiston sidosryhmien toiveet järjestelmälle selvitetään ja kuvataan, ja kuvausten pohjalta luodaan osiin jaettu työlista tehtävistä, joilla halutun lainen ohjelmisto saadaan tehtyä. Jo vuosituhannen vaihteessa Beck (1999) kuvasi Extreme programming -kehitysmenetelmässä ohjelmistokuvaukset käyttäjätarinoina (engl. user story), joista koostetaan julkaisut. Myös Scrumissa työn pohjana on priorisoitu tuotteen työjono, josta valitaan sprintille otettavat tehtävät osaksi sprintin työjonoa.

Osiin pilkottu kehitys luo tilanteen, jossa yksittäisiin julkaisuihin pitää valita halutut toiminnallisuudet, jotka siten toteutetaan halutussa järjestyksessä. Julkaisun hallinta on prosessi, jossa valitaan eri julkaisuihin tulevat toiminnallisuudet ottaen huomioon esimerkiksi tekniset vaatimukset, resurssit, budjetointi, mahdolliset riskit, sidosryhmien vaatimukset, aikarajoitteet tai vaatimusten riippuvuudet (Penny, 2002; Ruhe & Saliu, 2005). Suunnitellut julkaisun sisällöt voidaan myös nähdä eräänlaisena sopimuksena suunnitelmista vastaavien osapuolien ja suunnitelman toteuttavien eli kehittäjien kesken siitä, mitä seuraavaksi tullaan ohjelmistoon tekemään (Penny, 2002).

4 OHJELMISTOMITTARIT

Ohjelmistomittareihin liittyy olennaisen osana mittaaminen. Mittaamiseen liittyy paljon erilaisia määritelmiä, kuvauksia ja vaiheita. Mittaaminen (engl. measurement) yleisesti voidaan määritellä esimerkiksi prosessina, jossa tosielämän entiteettien ominaisuuksia kuvaamaan asetetaan niille numeroarvoja tai muita symboleja tarkasti määrättyjen sääntöjen mukaisesti (Fenton & Bieman, 2014). Mitta tai metriikka (engl. measure tai metric) on siten luku tai symboli, joka kuvaa entiteetin tiettyä ominaisuutta. Tosielämän entiteetti voi olla esimerkiksi ihminen, ohjelmiston järjestelmäkuvaus tai ohjelmistokehityksen tietty vaihe kuten testaus. Entiteetin ominaisuus on sen erityispiirre, joka voi olla esimerkiksi ihmisen tapauksessa pituus tai paino, järjestelmäkuvauksen tapauksessa sen tyyppi tai pituus ja ohjelmakehitysvaiheen tapauksessa esimerkiksi kyseisen vaiheen kesto (Fenton, 1994).

Tietyn ominaisuuden mittaaminen voi olla suoraa tai epäsuoraa eli johdettua. Suora mittaaminen ei riipu muista ominaisuuksista, esimerkiksi ohjelmiston testausvaiheen kesto saadaan suoraan sen aloitus- ja lopetusaikamäärien erotuksena. Tietyn ominaisuuden epäsuora mittaaminen riippuu toisten ominaisuuksien mittaustulosten arvoista, esimerkiksi tietyn ohjelmiston arvoa tai ohjelmiston hyvyyttä ei välttämättä pysty suoraan yhden ominaisuuden avulla määrittämään vaan se pitää johtaa useammasta ominaisuudesta (Fenton, 1994; Fenton & Bieman, 2014). Toisaalta ominaisuus voi olla sisäinen tai ulkoinen: sisäinen ominaisuus voidaan mitata suoraan kohteena olevan elementin puitteissa itsessään, kuten ohjelmiston kehitysprosessivaiheen kesto, kun taas ulkoinen ominaisuus riippuu myös muista entiteetin toimintaympäristön entiteeteistä, kuten esimerkiksi ohjelmiston luotettavuus ei riipu vain ohjelmistosta itsestään vaan myös esimerkiksi alustasta ja käyttäjästä (Fenton, 1994).

Mitattava data voi olla myös eri tyyppistä ja se voidaan jakaa ainakin kuuteen eri tyyppiin. Nominaalisessa datassa ei mittausarvoilla ole järjestystä vaan eri arvoille voidaan laittaa tiettyjä symboleja niitä kuvaamaan. Ordinaalidatassa dataa voidaan järjestää, mutta arvojen välistä arvomuutosta ei voi verrata, mitä taas voi intervallidatassa tehdä. Suhdelukudatalla voidaan arvioida arvojen välisiä suhteita. Lisäksi data voi olla muodossa, jossa se ennustaa ohjelmiston ke-

hittämiseen vaadittavaa työtä tai data voi olla absoluuttisia lukuja (Mills, 1988; Misra & Omorodion, 2011).

Lisäksi tärkeä on määrittää mittauksessa käytetty malli, jolla sovitaan tarkasti eräänlainen näkökulma, miten tiettyä ominaisuutta mitataan, jotta mittaaminen eri entiteettien saman ominaisuuden välillä on yhteneväistä. Esimeriksi jos mitataan ohjelmistokoodin pituutta, on tarkasti määriteltävä mikä on uniikki ohjelmistokoodirivi (Fenton, 1994). Mittaamisella voidaan myös tavoitella arviointia tai tulevaisuuden ennustamista (Fenton, 1994), jolloin esimerkiksi nykytilaa arvioidaan mittaamalla tiettyjä ominaisuuksia ja käyttäen hyväksi nykytilan arvioita yritetään mallintamalla arvioida tulevaisuuden kehittymistä kyseisten tai niistä johdettujen ominaisuuksien osalta.

Mittarilla voidaan tarkoittaa joko mittaamiseen käytettävää fyysistä mittalaitetta tai ohjelmistomittareiden tapauksessa eräänlaista indikaattoria, jota voidaan käyttää kuvaamaan halutun kohteen ominaisuuksia. Indikaattorimittarissa siten yhdistyvät mitta, mittaaminen ja malli mittaamisen näkökulmasta. Ohjelmistomittarit, tai ohjelmistometriikat, (engl. software metrics) on Gaffney (1981) määritellyt olevan objektiivinen ja matemaattinen mitta kuvaamaan ohjelmiston ominaisuuksia kvantitatiivisesti. Ohjelmistomittareiden voidaan myös määritellä olevan laajemmin kuvaus toiminnoista, jotka liittyvät mittaamiseen ohjelmistotekniikan alueella kuten esimerkiksi perinteisiä ohjelmistomittareita ohjelmistokoodin luonteesta, ennustemalleja ohjelmistosta tai laadunvarmistuksen toimintoja kuten testausvaiheessa havaittujen virheiden laskentaa (Fenton & Neil, 1999).

Ohjelmistoihin liittyviä mittareita voidaan hyödyntää hyvin laajasti eri osa-alueilla. Ohjelmistojen mittaamisen näkökulmasta mittarit voidaan jakaa esimerkiksi kolmeen eri alueeseen riippuen mitattavasta entiteetistä: prosessit, kuten ohjelmiston suunnittelu, toteutus ja testaus, mitkä tapahtuvat ajan mittaan; tuotteet, jotka syntyvät ohjelmistoprosessien tuloksena, kuten esimerkiksi ohjelmistosuunnitelmat ja ohjelmistokoodi; resurssit, jotka ovat prosessien osatekijöitä, kuten ihmiset ja kehitys- ja versionhallintaohjelmistot (Fenton, 1994; Fenton & Neil, 1999). Misra ja Omorodion (2011) lisäävät vielä neljänneksi kategoriaksi projektit, joilla voidaan viitata perinteisempiin projektin hallinnan mittareihin kuten projektin kustannukset, aika, tuloksen laatu tai liiketoimintahyöty. Toisaalta Ordonez ja Haddad (2008) esittävät ohjelmistomittareiden määrittämisen tueksi kymmenen eri ohjelmistoprojektien osa-aluetta, jonne niitä kuvaavia mittareita voi liittää: kustannus ja työaika-arviot mahdollistamaan tarkemmat ohjelmistoprojektien suunnitelmat ja toteutukset, tuottavuuslaskelmat arvioimaan työntekijöiden panosta projekteihin, tiedonkeräys luomaan mahdollisuudet saada oikeaa dataa mittareista, laadulliset arvioinnit esimerkiksi ohjelmiston käytettävyyden ja tehokkuuden arviointiin, luotettavuusmallit havaitsemaan virheitä, prosessien seuranta mittaroimaan ohjelmistokehitystä ja ylläpitoa, projektien seuranta mahdollistamaan paremmat arviot projektin sisällöistä ja etenemisestä sekä itse ohjelmistotuotteen eri osa-alueiden kuten määrittelyjen, ohjelmistokoodin kompleksisuuden tai testikattavuuden arvioinnit (Ordonez & Haddad, 2008).

Erilaisia ohjelmistomittareita voidaan eri osa-alueille kuvata ja käyttää varsin paljon. Mittareita ei kuitenkaan pidä ottaa käyttöön vain mittaamisen vuoksi, vaan jokaisen mittarin taustalla pitää olla joku tavoite mistä syystä kyseistä mittaria käytetään, mikä myös kertoo, miten mittarilla kerättyä dataa käytetään (Fenton & Bieman, 2014). Lisäksi on hyvä määrittää, ketä varten dataa tietyllä mittarilla kerätään. Fenton ja Bieman (2014) esittävät ohjelmistoprojektin johdon ja kehittäjän näkökulmasta eri tietoja, joita projektista olisi tiedettävä mittareiden avulla. Johto voi haluta tietää mitä eri ohjelmistoprojektin prosessit kuten vaatimusmäärittely, suunnittelu tai kehitys maksavat, kuinka projektin työntekijät suoriutuvat työstään eri prosessivaiheissa, kuinka laadukasta koodi on esimerkiksi virheiden ja bugien määrän kautta tai kuinka tyytyväisiä asiakkaat ovat tuotteeseen. Kehittäjien näkökulmasta mielenkiintoisia tietoja ovat esimerkiksi ohjelmiston vaatimusten laatu niin, että niitä vastaan voidaan toteuttaa ja testata järjestelmän kehitystä ja toimintaa, ohjelmiston virheiden määrät niin, että voidaan virheet paikallistaa ajoissa ja ne korjata tai täyttyvätkö prosessien eri vaiheissa kehitykselle määritetyt tavoitteet työn myötä (Fenton & Bieman, 2014).

Miksi ohjelmistoprojekteissa kannattaa käyttää mittareita? Mittareiden avulla saadaan prosesseista tietoa, jota voidaan käyttää niiden hallinnassa ja palveluiden tuottamisessa asiakkaille, ja samalla mittarit antavat palautetta havaituista hyödyistä, joita prosesseista ja palveluista saadaan niiden tuottajille. Metriikoiden avulla saadaan tietoa ongelmakohdista ja onnistumisista, ja niiden syistä. (McWhirter & Gaughan, 2012). Metriikoiden avulla voidaan siten tarkastella ohjelmistoprojekteja ja parantaa niiden toimintaa datan pohjalta, jotta on mahdollista tehdä parempia päätöksiä tekemisen parantamiseksi ja tueksi (Kupiainen ym., 2015). Mittareiden käyttämisen hyötyjä ovat esimerkiksi projektien etenemisen seuranta, tuotteen laaduntarkkailu ja parantunut projektin ennustamiskyky ja johtaminen (Padmini ym., 2015). Fenton ja Neil (1999) näkevät mittareiden hyödyntämisen tärkeimmän hyödyn olevan tiedon tuottamisen ohjelmistokehityksen johdon päätöksien tueksi. Mills (1988) esitti jo vuonna 1988 ohjelmistometriikan tärkeimmäksi tavoitteeksi ohjelmiston kehitykseen eniten vaikuttavien muuttujien määrittämisen ja mittaamisen.

Kupiainen ym. (2015) selvittivät mitä syitä ja vaikutuksia mittareiden hyödyntämisellä ketterän kehittämisen ohjelmistoprojekteissa on. He jakoivat tulokset viiteen eri osa-alueeseen. Sprintin ja projektin suunnittelussa mittareita on hyödynnetty priorisoimaan kehityksenalaisia toiminnallisuuksia ja selvittämään iteraatioihin otettavan työn määrää, sekä selvittämään työn vaativuutta ja kuormaa. Sprintin tai projektin seurannassa on käytetty mittareita seuraamaan työn edistymistä ja pystytäänkö saavuttamaan asetettuja tavoitteita. Lisäksi mittareita on hyödynnetty tuomaan läpinäkyvyyttä tekemiseen ja työn edistymiseen, sekä tasapainottamaan työtä työntekijöiden kesken. Tuotteen laadunvarmistuksessa mittareita on käytetty parantamaan tuotteen laatua ja varmistamaan testauksen taso. Mittareita on lisäksi käytetty löytämään ongelmakohtia ohjelmistokehitysprosessissa ja tuomaan esiin mahdollisia parannuskohteita.

Mittareita on myös hyödynnetty työntekijöiden motivaation ja käyttäytymisen muutoksessa, jotta virheet havaittaisiin aikaisemmin ja esimerkiksi teknistä velkaa yritettäisiin vähentää osana kehitystä (Kupiainen ym., 2015).

4.1 Mittareita ketterässä ohjelmistokehityksessä

Ketteriä ohjelmistokehitysmenetelmiä on useita ja eri menetelmien sisäisiä vaiheita ja prosesseja monia. Näin ollen kehittämistä voi mittaroida useassa eri vaiheessa ja useaan eri tarkoitukseen, kuten Kupiainen ym. (2015) edellä kuvatusti ovat esittäneet. Kirjallisuudesta löytyy useita artikkeleita, joissa esitetään tai on selvitetty käytettyjä mittareita ketterän kehittämisen eri osa-alueille ja eri menetelmiä hyödyntäville toimijoille.

Kupiainen ym. (2015) selvittivät yleisesti ketterän kehittämisen ja Leanin kontekstissa käytettyjä mittareita systemaattisen kirjallisuuskatsauksen avulla. He esittivät listaa 30 artikkelista löydetyistä mittareista, joissa eniten esiintyneitä mittareita oli esimerkiksi tiimin kehitysnopeus (velocity), arvioitu työaika (effort estimate), asiakastyytyväisyys, virheiden lukumäärä (defect count), teknologinen velka, buildien tila (build status), läpimenoaika (lead time), aktiivisten työtehtävien määrä ja suljettujen tehtävien suhteellinen määrä kaikkiin tehtäviin verrattuna. Kupiainen ym. (2015) mukaan mittareita käytettiin mm. sprintin suunnittelussa ja etenemisen seurannassa, laadunhallinnassa, kehitysprosessin ongelmienselvityksessä ja ihmisten motivoinnissa. Padmini ym. (2015) myös listaavat eri ketterän kehittämisen mittareita eri yrityksille tekemiensä kyselyiden ja haastatteluiden pohjalta. Heidän mukaansa eniten käytettyjä mittareita olivat esimerkiksi ajallaan toimitus (delivery on time), työkyvyn määrä (work capacity), testikattavuus, otetun työn suhteellinen määrä (percentage of adopted work), virhekorjauksien läpimenoaika, sprintin burndown-kaavio ja kehitysnopeus. Edellä mainituista mittareista haastateltavat olivat sanoneet esimerkiksi ajallaan toimituksen osalta, että se kuvaa hyvin onko kehityksen laajuus hallinnassa ja, että sitä voi käyttää kuvaamaan tulevaisuuden kehitystä, ja testikattavuudesta mainittiin, että se vaikuttaa tuotteen laatuun ja vähentää uudelleentestauksen käytettävää aikaa (Padmini ym., 2015).

Useat tutkimukset keskittyvät tarkempiin ketterän kehittämisen osa-alueisiin tai menetelmiin ja listaavat niihin liittyviä havaitsemiaan mittareita. Ketterän kehittämisen menetelmistä Scrum on tämän hetken yksi suosituimmista. Scrumiin liittyviä mittareita on listannut esimerkiksi Downey ja Sutherland (2013), jotka esittelevät kymmenen mittaria parantamaan toimintaa Scrumin kontekstissa: kehitysnopeus, työkyvyn määrä, vauhtikerroin (focus factor), otetun työn suhteellinen määrä, ”löydetyn” työn suhteellinen määrä (percentage of found work), arvioiden ja ennusteiden osuvuus, arvon nousu (targeted value increase), eri tasoisten työtehtävien onnistumisprosentti (success at scale) ja sprintin onnistuminen (win/loss record). Agarwal ja Majumbar (2012) esittävät seitsemän mittaria Scrum-projektien seuraamiseen: kehitysnopeus, poikkeamat koodin kehityskäytänteistä, liiketoiminta-arvo, virheiden määrä, kehitystehtä-

vien määrä, automaattitestien määrä ja testien kokonaismäärä. Medeiros ym. (2015) listaavat Scrumissa käytettäviksi esimerkkimittareiksi muun muassa tehtyjen työjonotehtävien määrän, keskimääräisen ajan työjonotehtävän tekemiseen, katselmoinnissa hyväksyttyjen tai hylättyjen tehtävien määrän, saatiinko kaikki sprintin työjonon tehtävät tehtyä sprintin aikana, kuinka tuoteomistaja osallistui Scrumin eri tapaamisiin ja kuinka kehittäjät osallistuivat päivittäistapaamisiin.

DevOps on Scrumin ohella nykypäivänä suosittu toimintamalli ohjelmistokehityksen alalla. Tutkimuksia DevOpsin mittareista kokonaisuutena ei ole tieteellisessä kirjallisuudessa niin paljoa, kuten on esimerkiksi Scrumiin liittyen. Lwakatare ym. (2016) esittävätkin, että DevOpsiin liittyen voisi käyttää liiketoimintaan nojaavia yleisiä mittareita kehityksen ja ylläpidon aloilta, joilla voisi arvioida molempien DevOpsin osa-aluekokonaisuuksien toimintaan. DevOpsin mukaisessa kehittämisessä voisi siten käyttää mittareita esimerkiksi yleisesti ohjelmistokehitysmenetelmän piiristä, kuten esimerkiksi Scrumin mittareita. DevOpsin ulottuvuuksista yksi on automaatio, johon liittyy esimerkiksi aiemmin kuvattuja jatkuvan ohjelmistokehityksen konsepteja ja toimintoja. Automaattinen koodin integrointi, testaus ja asennus antavat hyvät mahdollisuudet käyttää myös automaattisia mittareita ohjelmistokehitysprosessin tukena. Jatkuvan toimituksen näkökulmasta Lehtonen ym. (2015) esittävät mittareita kehittämisen ja kehitysputken (pipeline level) tueksi. Sopivia kehityksen seurannan mittareita heidän mukaansa ovat kehitysaika, toimitusaika, toiminnallisuuden (feature) käyttöönoton aktivointiaika ja vanhin toimittamaton toiminnallisuus (oldest done feature, ODF), ja toisaalta kehitysputken hallinnan osalta toiminnallisuuksien määrä kuukaudessa, julkaisujen määrä kuukaudessa ja toiminnallisuuden nopein läpimenoaika (Lehtonen ym., 2015).

Ketterien ohjelmistomenetelmien, kuten myös esimerkiksi perinteisten menetelmien, keskiössä on itse ohjelmistotuotteen tekeminen. Ketterien ohjelmistomenetelmien prosesseissa ohjelmistokoodin tuottamiseen liittyvät menetelmät tai prosessit saatetaan jättää jopa kokonaan kuvaamatta, kuten Scrumin tapauksessa, jolloin jokainen menetelmää käyttävä voi soveltaa omia tapoja ja prosessejaan laajemman menetelmän tai kehyksen sisällä. Ohjelmistokoodiin liittyviä mittareita on käytetty ja tutkittu jo usean vuosikymmenen ajan ja uusia mittareita luodaan edelleen. Lähdekoodimittarit ovatkin tärkeä osa ohjelmistojen mittaamisen alalla (Nunez-Varela ym., 2017). Esimerkiksi Nunez-Varela ym. (2017) listaavat eri tutkimuksien pohjalta käytetyimpiä mittareita. Heidän tuloksissaan oliosuuntautuneessa ohjelmistokehityksessä suosituimpia mittareita olivat esimerkiksi Weighted Methods per Class (WMC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), Depth of Inheritance Tree (DIT), Lines of Code (LOC) ja Number of Children (NOC).

Kokonaisuudessaan eri ohjelmistokehityksen osa-alojen seuraamisen ja hallinnan mittareita on useita, osalla niistä on pitkäkin käyttöhistoria ja toisaalta uusia mittareita kehitetään jatkuvasti. Mittarit voivat olla eri menetelmien tai osa-alueiden välillä osittain samoja, vaihdella samanlaisten kohteiden välillä tai esimerkiksi vastata hieman eri tarkoitukseen. Näin ollen ei ole välttämättä ole-

massa aina sopivia mittareita tiettyyn samaan tilanteeseen. Lisäksi esimerkiksi ketterissä ohjelmistomenetelmissä jätetään tilaa soveltaa tekemistä esimerkiksi tiimi- tai kohdealuekohtaisesti, ja täten parhaiten soveltuvat mittarit voivat vaihdella jo sovellettujen menetelmien eroavaisuuksien myötä. Näin ollen jokaisen tiimin tai kehitysprosessin hallitsijoiden pitää soveltaa mittareita tarpeiden, vaatimusten ja mahdollisuuksien mukaan miten kyseisissä tilanteissa nähdään parhaaksi ja toimintaa edistäviksi.

5 KOHDEALUE JA MENETELMÄT

Tutkimus toteutettiin osana tutkimuksen kohteena olevan Scrum-tiimin toimintaa ja kehitystyötä oman ohjelmistokehitysprosessin parantamiseksi. Tutkimuksessa kerättiin tarvittavaa aineistoa vaiheittain iteratiivisesti sitä parantaen ja tiimin kanssa keskustellen. Seuraavaksi kuvataan tarkemmin kohdealuetta ja tiimin tilannetta ennen tutkimusta, sekä itse tutkimuksessa käytettyä tutkimusotetta ja tutkimuksen vaiheita.

5.1 Kohdealuekuvaus

Tutkimuksen kehitystiiminä toimi Scrum-tiimi, jossa oli neljä kehittäjää, tutkimuksen loppuvaiheessa rekrytoinnin myötä tiimi kasvoi yhdellä vakinaisella kehittäjällä, Scrum Master, joka toimi myös osittain kehittäjänä, ja tiimin päätuoteomistaja. Tiimillä oli tutkimuksen aloituksen aikaan ollut Scrum-viitekehys käytössä vuoden ajan. Tiimin jäsenillä oli sitä ennen ollut Scrum-kokemusta vaihtelevasti: kahdella kehittäjällä oli useamman vuoden kokemus Scrum:sta ja Scrum Master oli toiminut kyseisessä roolissa jo aikaisemmassa työhistoriassaan, lopuilla kehittäjillä ei ollut aikaisempaa Scrum-kokemusta ennen tiimiin liittymistään, kuten ei tuoteomistajallakaan. Tiimi toimii isossa suomalaisessa organisaatiossa vastuullaan useita vaihtelevan kokoisia järjestelmiä. Kaikkien järjestelmien kehityksellinen ylläpitovastuu oli myös tiimillä. Tiimi toimi kehityksen osalta itsenäisesti, mutta esimerkiksi määrittely, testaus ja alustojen ylläpito olivat pääosin muiden tiimien vastuulla. Tiimi ei siten ollut itse kehittämisen osalta riippuvainen muista tiimeistä tai tehnyt muiden kanssa samaa toteutusta.

Tiimin versionhallinta ja työnohjaustyökaluna oli Team Foundation Server 2018 update 3 (TFS). TFS:ssä tiimillä oli käytössä koodienhallintaan Git-pohjainen versionhallinta. Tehtävienhallinnassa tiimi käytti neljän erilaisen tehtävän ja tehtävän tason kuvaustapaa: Feature koostaa yhteen loogisen, maksimissaan noin parin kuukauden kestoisen tehtäväkokonaisuuden, joka koostuu

useimmiten useasta user story -tehtävästä, missä on tarkemmin kuvattu yksi looginen tehtäväkokonaisuus sisältäen muun muassa kuvauksen, laajuusarvoin ja valmistumisen hyväksymiskriteerit. User storyn alla oli vielä Task-tason kokonaisuus, jossa User storyn toteutus oli jaettu selviin pienempiin toteutettaviin tehtäviin sisältäen arvoidut, jäljellä olevat ja toteutuneet tunnit. Näiden lisäksi tiimillä oli käytössä myös Bugi, jossa kuvataan järjestelmien virhetilanne.

Tutkimuksen alkuvaiheessa tiimi oli ottamassa käyttöön omaa suljettua kehitysverkkoa, joka tutkimuksen aikana otettiin tuotantokäyttöön. Juuri oma kehitysverkko antoi tiimille mahdollisuuden paremmin alkaa seurata omaa kehitystoimintaansa, mikä oli myös alkusysäys tälle tutkimukselle. Kaikki esitettävät tulokset tulevat siten koskemaan tiimin kehitysverkon sisäistä toimintaa ja tiimin mahdollisuuksia esimerkiksi DevOps-maiseen kehittämiseen ja kehityksen mittarointiin.

Vuoden mittaisen Scrum-pohjaisen kehityksen kautta tiimille oli kehittynyt osittain vakiintuneet käytänteet kehittämisprosessien osalta, mutta tiimillä oli uuden kehitysverkon käyttöönottamisen myötä halu kehittää myös omia toimintaprosessejaan paremmiksi. Prossien tarkastelu toimi myös hyvänä pohjana keskustelulle tarvittavista ja käyttöönotettavista kehityksen seurannan mittareista. Tiimillä ei ennen tutkimusta ollut kuvattuna heidän kehityksen vaiheita prosessikaaviotyyppisissä kaavioissa, vain joitain tekstipohjaisia kuvauksia ja erinäisiä aiheen ympärillä käytyjä keskusteluja. Lisäksi tiimillä ei ollut ennen tutkimusta omia mittareita käytössä.

Tutkimuksessa käytettiin seminaarimaista keskustelevaa tutkimustapaa, jota pohjustivat tutkimuksen tekijän valmistelevat työt. Tiimin jäsenet osallistuivat kaikkiin keskusteluihin ja lisäksi osassa keskusteluista oli myös mukana jotain yksittäisiä yhteistyötiimien tai tiimin esimiestason henkilöitä. Keskustelut nojasivat aina edellisten keskusteluiden pohjalta tutkimuksen tekijän tekemiin prosessikuvausten tai mittarilistauksien päivityksiin. Tutkimus oli siten iteratiivista ja aina edellistä tulosta kehittävää. Tutkimuksen taustalla toimi tieteellisenä viitekehyksenä design science -tyyppinen tutkimusote.

Seuraavissa kappaleissa esitellään lyhyesti design science -tutkimusmenetelmää ja tutkimuksen vaiheita jaoteltuna tutkimuskysymysten kautta kahteen alakappaleeseen kehitysprosessin kuvauksesta ja mittareiden kartoittamisesta.

5.2 Design science

Design science on tapa tehdä tieteellistä tutkimusta, missä yhdistyy erilaisia tieteenfilosofisia näkemyksiä ja analyyttisiä ja synteettisiä tekniikoita (Vaishnavi ym., 2004). Hevner ja Chatterjee (2010) määrittelevät design science -tutkimuksen olevan tutkimusparadigma, missä tutkimuksen ja tutkimustiedon lisääntymisen ohella luodaan innovatiivisia ja hyödyllisiä artefakteja vastaamaan ihmisten tarpeista nouseviin kysymyksiin. Artefaktien ohella design science -tutkimus voi keskittyä laajentamaan olemassa olevaa tutkimustietoa

design science -tutkimuksen piirissä. Näin ollen design science -tutkimus on mielenkiintoinen paradigma, koska se yhdistää käytännöllisten, ihmisten käyttöön suunniteltavien ratkaisujen luomisen artefaktien kautta tieteelliseen tutkimukseen ja tieteellisen tietämyksen laajentamiseen luotaessa design science -tutkimusteorioita ja tutkimustietoa. Tutkimustuloksena voi siis syntyä suunnitteluartefakteja (design artifacts), suunnitteluteorioita (design theories) tai molempia (Baskerville ym., 2018). Artefaktien luominen voidaan kuvata kahden aktiviteetin kautta: tietämyksen lisääminen uusien ja innovatiivisten artefaktien luomisen myötä ja toisaalta artefaktien käytön ja toiminnan arvioiminen. Teorioiden tapauksessa tutkimuksen tarkoituksena on laajentaa tietämystä luomalla teorian pohjalta ohjeistuksia ja määritelmiä järjestelmäkehityksen tuotoksista (Vaishnavi ym., 2004).

Peffers ym. (2007) esittävät design science -tutkimuksen prosessimallin, jossa on kuusi vaihetta. Ensimmäinen vaihe on ongelman esittäminen ja motivointi. Ongelman pohjalta lähdetään tavoittelemaan artefaktia, joka toisi ratkaisun ongelmaan. Toisessa vaiheessa määritellään toteutettavalle ratkaisulle tavoitteet, jotka voivat olla kvantitatiivisia tai kvalitatiivisia. Kolmannessa vaiheessa määritetään tavoiteltu artefakti tarkemmin ja luodaan se. Artefakti voi olla mikä vain suunniteltu ja tieteellisen työpanoksen sisältävä tuotos. Neljäs vaihe on artefaktin demonstrointi, missä esitetään, miten artefakti ratkaisee alussa esitetyn ongelman. Viidennessä vaiheessa arvioidaan, kuinka hyvin artefaktin tuottama tulos ratkaisee alkuperäisen ongelman. Viimeisessä kuudennessa vaiheessa esitetään ongelma, sen motivointi ja tärkeys, luomisen menetelmät, artefakti ja sen toimivuus ratkaisemaan ongelma kohdeyleisölle, joka voi tilanteesta riippuen koostua tutkijoista tai kohdealueen ammatinharjoittajista. Tutkimuksen prosessimallissa edetään vaiheittain ja iteratiivisesti, mutta Peffers ym. (2007) esittävät, että tilanteesta riippuen prosessin voi aloittaa neljästä eri kohdasta. Ongelmakeskeisessä tapauksessa aloitetaan ensimmäisestä vaiheesta, jolloin esimerkiksi havaitusta ongelmasta saadaan idea tutkimuksen tarpeelle. Tavoitekeskeinen tutkimus alkaa vaiheesta kaksi, missä on esitetty tarve tutkimukselle, johon artefakti toisi ratkaisun. Suunnittelu- ja toteutuskeskeinen lähestymistapa alkaa vaiheesta kolme, jolloin on olemassa jo artefakti, mutta sitä ei ole kyseiseen ongelmaan tai kohdealueeseen vielä sovellettu. Viimeinen asiakas- tai asiayhteyskeskeinen tapa on aloittaa tutkimus neljännestä vaiheesta, jolloin on jo olemassa toimiva ratkaisu ongelmaan, jolloin voidaan takautuvasti parantaa kyseessä olevan artefaktin design science -prosessimallin mukaista luontia.

Tässä tutkimuksessa vastataan tutkimuskysymyksissä esitettyihin ongelmiin luomalla kaksi artefaktikokonaisuutta tiimin ohjelmistokehitysprosessista ja prosessin seurantaa ja kehitystä tukevista mittareista. Tutkimuksen lähtökohta on Peffers ym. (2007) prosessimallin tavoitekeskeisen ratkaisun mukaisesti vaiheesta kaksi: tiimillä oli ongelma saada käyttöön mittaristo toimintansa tueksi ja sen luomista tukemaan prosessikuvaus tiimin ohjelmistokehityksen kulusta.

5.3 Tiimin kehitysprosessin kuvaus

Tutkimuksen ensimmäinen vaihe oli kuvata tiimin ohjelmistokehityksen prosessit siten, miten tiimi silloisten prosessien pohjalta haluaisi kehityksen tulevaisuudessa menevän uuden kehitysverkon käyttöönoton myötä. Prosessikuvauksissa huomioitiin ensin silloinen tila ja keskustellen iteratiivisesti suunniteltiin prosessia, joka vastaisi tiimin tulevaisuuden kehitysmallia. Prosessit kuvattiin yleisluontoisina vuokaavioina käyttäen ohjelmaa yEd (v. 3.19). Prosesseihin rajattiin mukaan vain TFS:n sisällä tapahtuva työjonotehtävien luominen ja käsittely sekä niihin liittyvä ohjelmistokoodin käsittely soveltuvin osin lähinnä koskien automaattista koodinkäsittelyä kuten buildausta ja testausta. Näin prosesseista rajautui ulos vaiheita, joissa tiimi ei välttämättä suoraan ole mukana, kuten esimerkiksi ylemmän tason toteutettavien ohjelmien suunnittelu ja määrittely tai eri järjestelmien tuoteomistajien välinen suunnittelutyö. Lisäksi suurin osa mittaroinnissa käytettävästä datasta oli saatavilla vain TFS:n kautta, joten oli luonnollista rajata toimet koskemaan vain ko. alustalla tapahtuvia toimia. Tutkimuksen tekijä muokkasi itsenäisesti kaavioita, joita sitten videopalavereissa (Skype-tapaamisia, joissa tutkimuksen tekijä jakoi ruudun muille osallistujille) käsiteltiin, ja tutkimuksen tekijä joko suoraan muokkasin keskustelun aikana kaavioita tai kirjasi ylös muistiinpanoja keskustelusta tulevia kaavioiden iteraatioversioita varten.

Ensimmäisen raakaversion tiimin prosesseista tutkimuksen tekijä tuotti oman kokemuksensa pohjalta tiimin toiminnasta. Kyseistä versiota käytettiin pohjana ensimmäisessä videokeskustelussa, jossa paikalla oli koko Scrum-tiimi mukaan lukien siis tuoteomistaja ja Scrum Master. Keskustelun pohjalta tutkimuksen tekijä laajensi kaavioita, jota käsiteltiin seuraavassa videotapaamisessa ja niin edelleen. Noin puolissa tapaamisista paikalla oli vain kehitystiimin jäseniä ja lopuissa myös tuoteomistaja. Videotapaamisissa käsiteltiin usein vain tiettyä pienempää osaa prosessista, joka saattoi laajentua useampaan osaan keskustelun pohjalta tai joku prosessi ei koettu tarpeelliseksi kuvata siinä vaiheessa. Prosessi käsiteltiin läpi laajemmin tiimin seminaaripäivässä, jossa ensimmäisen kerran käytiin kerralla koko siihen asti koostettu prosessikuvaus. Silloin paikalla oli koko Scrum-tiimi, laajemmasta eräänlaisesta tiimin DevOps-kokoonpanosta alustatiimin jäsen sekä tiimin esimies ja palvelupäällikkö. Noin kolmen tunnin seminaarikeskustelun aika käytiin koko tavoiteltu prosessi läpi tutkimuksen tekijän johtamana. Keskustelun tuloksena muokatut kaaviot käytiin vielä Scrum-tiimin kesken läpi.

5.4 Kehitysprosessia tukevien mittareiden kartoittaminen

Mittareiden kerääminen aloitettiin kartoittamalla kirjallisuudesta mittareita. Tutkimuksen tekijä jakoi kohdealueen eri osa-alueisiin, mistä tarkemmin kartoittaa mittareita. Osa-alueet määritettiin siten, että ne koskivat suoraan tii-

min ohjelmistokehitysmenetelmiä, kuten Scrum tai DevOps, tai muuten liittyivät tiimin ohjelmistokehitykseen, kuten automaattinen koodin julkaiseminen. Jokaisen osa-alueen kohdalla listattiin kirjallisuudesta mittareita, joko käytössä havaittuja mittareita tai sitten kyseiseen kohdealueeseen ehdotettuja mittareita. Lähteitä etsittiin Google Scholar -palvelusta osa-alueiden hakusanoilla ja tarvittaessa käytettiin myös ei-tieteellisiä internetsivuja, jos esitettyjä mittareita ei tieteellistä viitteistä tarpeeksi löytynyt. Mittarit koostettiin Excel-taulukkoon, jossa oli mittarin nimi ja viite.

Kirjallisuuden pohjalta koostetusta mittarilistauksesta luotiin vielä tarkemmin eri osa-alueisiin jaoteltu taulukko. Alkuperäisen mittarilistan mittarit jaoteltiin mittarin kohteen tai luonteen perusteella kategorioihin, joita tunnistettiin Scrum-tiimin ohjelmistokehitysprosessin avulla tai muuten olivat loogisia ohjelmistokehityksen osakokonaisuuksia itse mittareiden pohjalta luotuna. Tiimin prosessin mukaisiksi kategorioiksi määritettiin esimerkiksi sprinttiin, työjonotehtäviin tai testaukseen liittyvät mittarit ja yleisesti ohjelmistokoodiin liittyvät mittarit. Mittareista poistettiin mahdollisia havaittuja tuplia ja mittarit sijoitettiin sarakkeittain eri kategorioihin vain nimensä perusteella. Tässä vaiheessa vielä kaikki mittarit olivat englanninkielisiä.

Tiimin seminaaripäivässä käsiteltiin jo edellä mainitulla laajennetulla tiimikokoonpanolla kirjallisuuden pohjalta koostettuja mittareita. Kerätyt mittarit oli toimitettu kaikille osallistujille etukäteen ja osa-alueisiin jaetun mittarilistauksen ohella toisella Excel-taulukon välilehdellä oli myös jokaisesta mittarista viite ja mahdollisesti lisätietoa. Myös noin kolmen tunnin mittaisessa keskustelussa tutkimuksen tekijän johdolla tiimi keskusteli kerätyistä mittareita ja mitkä niistä kategoria kerrallaan olisi tiimiä kiinnostavia ja mahdollisesti voisi siten ottaa osaksi tiimin toiminnan hallintaa ja seurantaa. Jokaisen kategorian mittareista valittiin yksi tai useampi mittari tulevaisuuden jatkotarkasteluun värjäämällä mittarin sarake valitsemattomista sarakkeista eroavalla värillä. Samalla osallistujilla oli mahdollisuus nostaa esitettyjen mittareiden ulkopuolelta tai esitetyistä mittareista johdettuja mittareita mukaan jatkotarkasteluihin jokaisen kategorian kohdalle. Tilaisuudessa esitetyt uudet mittarit kirjattiin suomeksi samaan kategoriaan kirjallisuudesta kerättyjen mittareiden kanssa.

Jatkotarkasteluun valitut mittarit koostettiin omaan Excel-taulukkoon, jonne jokaisesta mittarista kerättiin lisätietoja: mittarin suomenkielinen käännös, tuliko se kirjallisuudesta vai keskustelun pohjalta, mittarin kategoria, mittarin datan muoto (tuleeko data automaattisesti osana esimerkiksi TFS-itemien käsittelyä vai pitääkö mittarin data kerätä erikseen, kuten esimerkiksi, että kehittäjän pitää muistaa syöttää tekemänsä tunnit oikeille työtehtäville) ja mittarin seurantajakso. Jatkojalostettua taulukkoa käytettiin jälleen Scrum-tiimin kesken (tuoteomistaja, Scrum Master ja kaksi kehittäjää) käydyssä keskustelussa mittareiden priorisointiin. Tiimi keskusteli mittareista ja äänesti niiden tärkeydestä: jokainen osallistuja sai merkitä listasta kymmenen mittaria, jotka näkivät prioriteetiltaan tärkeimmiksi ja ns. ensimmäisessä vaiheessa käyttöönotettaviksi. Äänestyksen tuloksista keskusteltiin ja todettiin, että mittarit, joilla oli enemmän kuin yksi ääni otettiin mukaan lopulliseen mittarilistaukseen. Näiden lisäksi

otettiin listaukseen mukaan muita yhden äänen saaneita mittareita, jotka keskustellen todettiin tarpeelliseksi ottaa mukaan listaukseen.

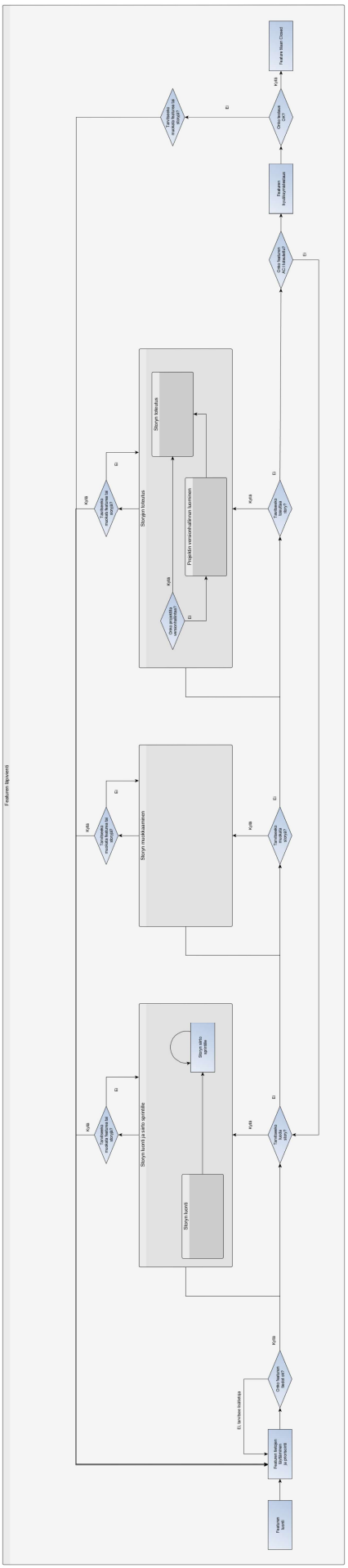
6 TULOKSET

6.1 Tiimin kehitysprosessi

Kehitysprosessin kuvaukset jaettiin kehitysjonolla olevien tehtävätasojen mukaisiin osiin, joissa vielä mahdollisesti tarkempia alaosia. Lisäksi mukaan otettiin release-suunnittelu.

6.1.1 Feature

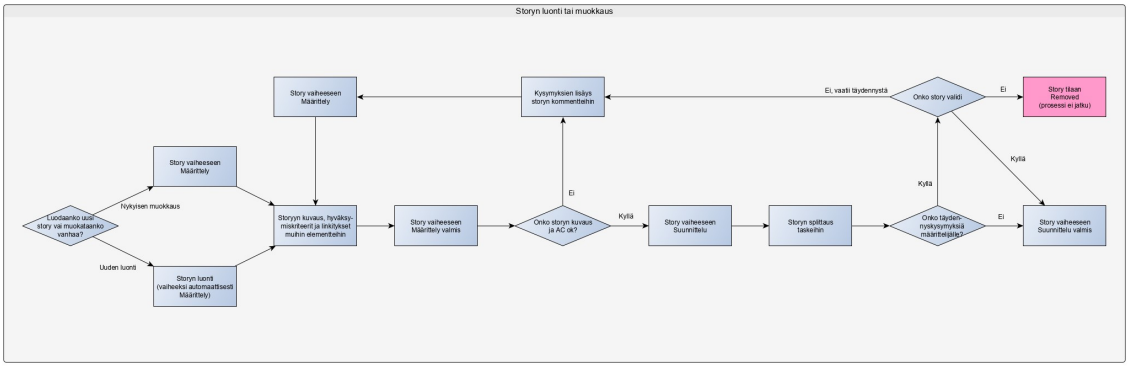
Feature oli tutkimushetkellä ylin TFS:n työjonolla oleva tehtävätaso, jota tiimi käytti. Se koostaa yhteen alemman tason user story -tehtäviä. Featuren läpiviennistä mallinnettiin kaavio, joka esitetty kuviossa 1. Featuren läpivienti koostuu featuren luonnista ja sen tietojen lisäämisestä, siihen liittyvien storyjen luonnista ja muokkauksesta, storyn toteutuksesta ja featuren hyväksymisestä. Tarvittaessa featuren ja siihen liittyen storyjen tietoja tulee olla mahdollista muokata luonnin jälkeen ja toteutuksen aikana.



KUVIO 1 Featuren läpivienti

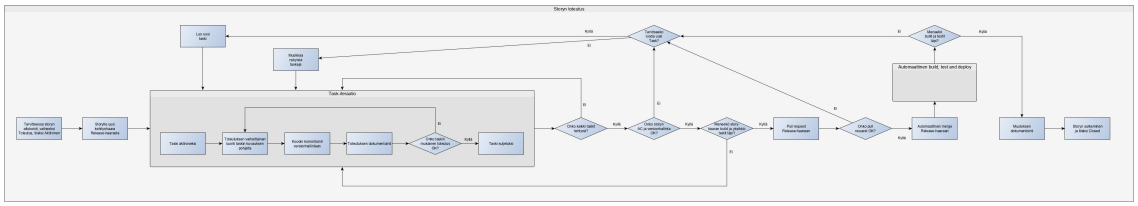
6.1.2 User story ja task

Varsinaiset työnjonon tehtävät, jotka lisätään sprinteille, ovat user story -tason tehtäviä. Jokaisella user storylla on oltava ylätasollaan feature, ja storyn tehtävät on jaettava alatason taskeihin. Storyn luonti- ja muokkausprosessissa (kuvio 2) olennaisia kohtia ovat lisättyjen tietojen tarkistus kehitystiimin toimesta ja toisaalta storyn tietojen jakaminen (“splittaus”) taskeihin. Storyä ei hyväksytä toteutettavaksi, ellei siinä ole kaikkia tietoja oikein lisättynä, vaan se palautetaan takaisin muokattavaksi lisääjälleen. Erityisesti storyn hyväksymiskriteerit pitää olla hyvin kirjattuna ja oikeassa muodossa, jolloin kriteerit toimivat eräänlaisena tarkistuslistana storyn valmistumisen toteamiselle. Toteutukseen hyväksytyt storyt pilkotaan taskeihin tiimin toimesta, jolloin arvioidaan jokaisen taskin työmäärä ja siten storyn ennakoitu työmäärä.



KUVIO 2 User storyn luonti ja muokkaus

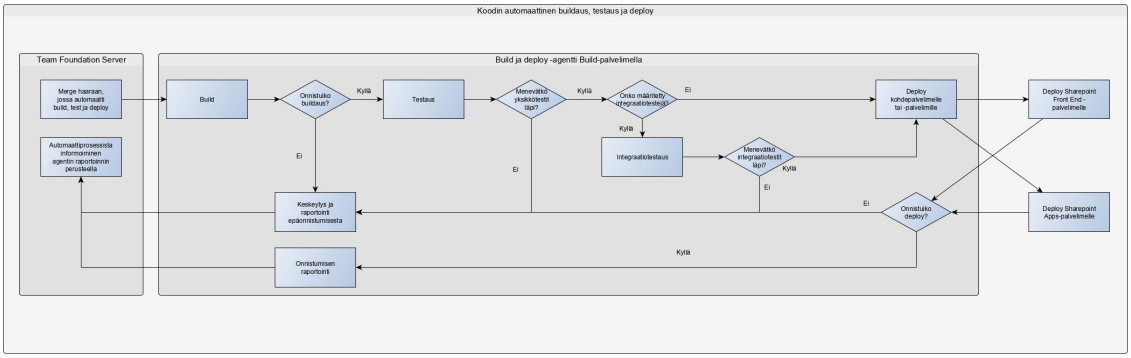
Storyjen luonnin ja splittauksen jälkeen story voidaan sprintillä toteuttaa (kuvio 3). Storyn toteutuksessa sille luodaan versionhallintaan oma kehityshaara, joka otetaan release-tason haarasta. Story laitetaan aktiiviseksi ja varsinainen toteutus tehdään vaiheittain taskien mukaisesti. Samalla dokumentoidaan tehtävää työtä ja kirjataan ylös taskeihin käytetyt tunnit. Tarvittaessa muokataan taskeja, jos havaitaan, että taskeissa olevat tehtävät eivät täytä storyssä olevia vaatimuksia. Olennaiset osat storyn toteutuksessa ovat lisäksi erityisesti tehdyn työn katselmointi pull request -tyyppisesti ja automaattiset koodin buildaus, testaus ja tarvittaessa myös asennus kehitysympäristöön. Pull requestissa kehittäjä tekee kehityshaarasta release-tason haaraan eräänlaisen merge-pyynnön, joka pitää muiden kehittäjien käydä hyväksymässä ennen sen liittämistä osaksi release-haaran koodia. Tarvittaessa toteutusta pitää tässä vaiheessa vielä muuttaa.



KUVIO 3 User storyn toteutus

6.1.3 Koodin automaattinen julkaiseminen

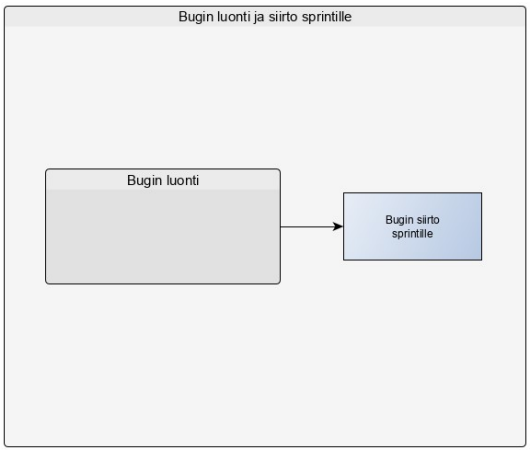
Hyväksytyn pull requestin jälkeen TFS:n versionhallinnassa tehdään automaattiset koodin buildaus- ja testaustoimet sekä tarvittaessa toteutuksen asennus (deploy) kehityspalvelimelle (kuvio 4). Automaattisia buildejä ja testejä on mahdollista tarvittaessa lisätä myös jokaiseen versionhallintaan tehtävään koodin "push"-lisäykseen.



KUVIO 4 Koodin automaattinen buildaus, testaus ja deploy

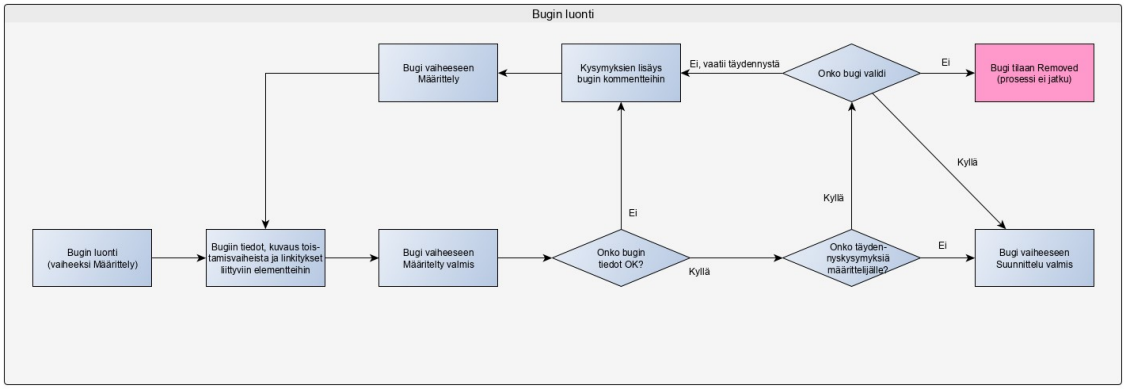
6.1.4 Bugi

Feature-tason ja sen alatasoilla olevien tehtävien ohella tiimin kehitysjonolla on myös bugeja, toteutuksen ohjelmointivirheitä, joilla ei ole ylätason työjonotehtäväelementtiä, eikä niitä myös jaeta alatason taskeihin. Bugeja havaitaan ja niitä luodaan nopeammalla aikataululla kuin feature ja storyjä. Niiden luonnin jälkeen pitääkin heti päättää mille sprintille ne otetaan niiden kriittisyyden ja vaikuttavuuden perusteella (kuvio 5)



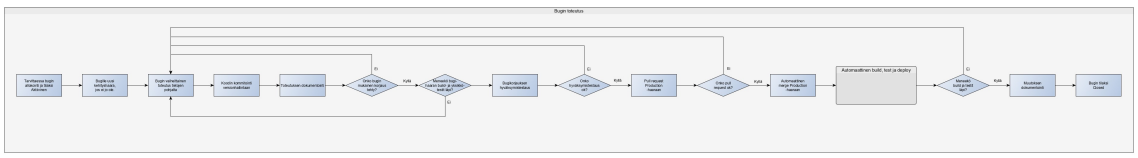
KUVIO 5 Bugin luonti ja siirto sprintille

Bugin luonnin yhteydessä kehitystiimin arvioi, onko kaikki tiedot täytetty oikein, jotta bugia voi lähteä selvittämään (kuvio 6). Puutteelliset bugin palautetaan muokattavaksi. Joskus tiimille tulevat bugit eivät ole valideja bugeja vaan esimerkiksi ohjelmassa olevia toiminnallisuuksia, joista pitää käyttäjiä ohjeistaa. Tällöin bugit poistetaan. Joskus myös toinen bugikorjaus tai toiminnallisuuden lisäys on korjannut mahdollisen bugissa mainitun ongelman, jolloin myös bugi ei ole enää validi ja sen voi poistaa.



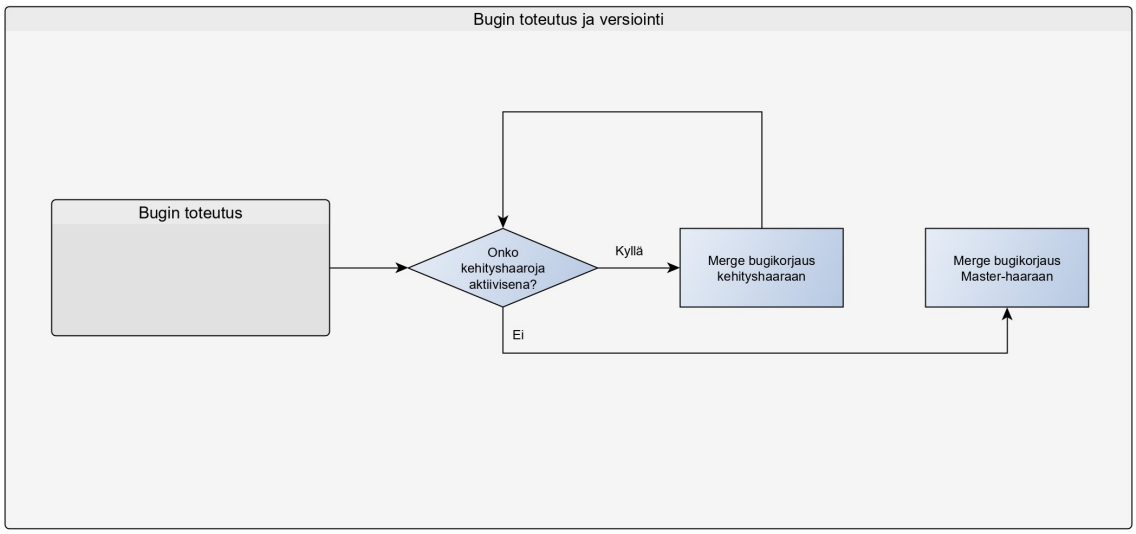
KUVIO 6 Bugin luonti

Bugin toteutus vastaa paljolti storyn toteutusta, mutta kaikki tekeminen on kuvattuna itse bugin tiedoissa, ja bugin valmistumista arvioidaan bugissa mainittuihin hyväksymiskriteereihin verrattuna. Myös bugin toteutuksessa on mukana koodin katselmointi pull request -tyyppisesti ja koodin automaattinen build, testaus ja tarvittaessa deploy (kuvio 7).



KUVIO 7 Bugin toteutus

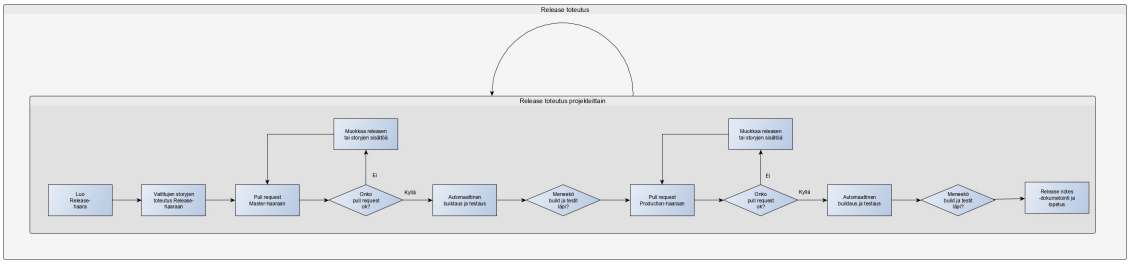
Bugin versioinnin yhteydessä on oltava tarkkana, että toteutus siirretään ylemmän tason release- ja master-haaran ohella myös muihin mahdollisesti aktiivisiin kehityshaaroihin, jotta niiden yhdistäminen ylemmän tason haaroihin ei poista jo lisättyä korjausta tulevista toteutuksen asennettavista versioista (kuvio 8).



KUVIO 8 Bugin toteutus ja versiointi

6.1.5 Release-suunnittelu

Työjonolla olevien tehtävien ohella keskustelussa nostettiin esille releasesuunnittelun tärkeys myös mittariston ja ohjelmistokehityksen toteutuksen suunnittelussa ja seurannassa. Näin ollen työjonotehtävien ohella mallinnettiin projekteittain tapahtuva versionhallinnan release-haaran käsittely. Releasen toteutuksessa valitaan projekteittain mukaan tiettyyn releaseen kuuluvia toteutuksia ja ne yhdistetään master- ja production-tason haaraan haluttuna ajankohtana (kuvio 9). Mukana haarojen yhdistämisissä on automaattiset build, testaus ja deploy, jotka epäonnistuessaan saattavat johtaa releasessa mukana olevien storyjen sisältöjen ja toteutuksien muokkaukseen.



KUVIO 9 Release-toteutus

6.2 Tiimin mittarit

Alustavan kirjallisuuskatsauksen ja tiimin kohdealueen pohjalta mittareita päätettiin tarkemmin kartoittaa viideltä eri osa-alueelta: Scrum, DevOps, ketterä kehittäminen (Agile) yleisesti, automaatio sekä ohjelmistot ja ohjelmistokoodi. Kyseisiä termejä (englanniksi) käytettiin hakemaan Google Scholar -hakukoneella tieteellisiä artikkeleita mittaristoihin liittyvästä kirjallisuudesta. DevOps-kirjallisuudesta ei löytynyt hakuhetkellä montaa artikkelia mittareista, joten ko. osa-alueelta haettiin mittareita myös ei-tieteellisistä lähteistä Google-internethaulla. Mittareita listattiin yhteensä eri osa-alueilta 226 kappaletta 21 eri lähteestä, joista 18 oli tieteellisiä artikkeleita ja kolme internetsivua. Osa-alueittain mittareita listattiin niin, että Scrum-mittareita löytyi 72, DevOpsista 8 tieteellistä ja 40 internetlähteistä, Agile-mittareita listattiin 62, automaatiosta 7, ohjelmistokoodista tai ohjelmistoista 37. (LIITE 1 Taulukko 1 on kuvattu eri osa-alueilta löydetyt mittarit ja niiden viitteet.)

Listattujen mittareiden osa-aluejaotelmaa muutettiin mittareiden pohjalta seuraavaan vaiheeseen tarkemmaksi tiimin kohdealue huomioiden. Samalla poistettiin suorat mittariduplikaatit alkuperäisten osa-alueiden väliltä. Mittarit jaettiin 11 suomenkieliseen kategoriaan (suluissa kappalemäärät): sprintti (9), feature (4), backlogitemit (16), virheet/bugit (23), testaus (5), kaaviot (5), ajanhallinta (8), tiimi (24), koodianalytiikka (41), automatiikka (12) ja muut (35). Mittareita näissä kategorioissa on listattuna 182. Mittarit olivat vielä kaikki englannin kielellä. (LIITE 1 Taulukossa 2 on listattu mittarit 11 kategoriassa.)

Seminaaripäivässä käydyn keskustelun pohjalta valittiin jokaisesta kategoriasta mittareita, jotka olivat tiimille kiinnostavia ja potentiaalisia mittareita ottaa osaksi tiimin ohjelmistokehitysprosessin seurantaa. Valmiista mittarilistauksesta valittiin 55 mittaria. Tämän lisäksi keskustelussa nostettiin valmiin listauksen ulkopuolelta 17 mittaria. Keskustelun aikana taulukkoon lisättiin myös uusi sarake suunnittelujakso, jonka alle lisättiin yleinen mittari jakson onnistuminen sekä suunniteltujen featureiden/releasien onnistuminen, mutta niitä ei lopulta otettu osaksi valittuja mittareita. (LIITE 1 Taulukko 3 on esitetty keskustelussa valitut ja keskustelun pohjalta lisätyt mittarit.)

Jatkokäsittelyyn valitut mittarit koostettiin omaan taulukkoon (taulukko 1). Samalla jokaisen mittarin tietoja tarkennettiin ja laajennettiin alustavasti. Käsittelyn aikana poistettiin neljä mittaria, joilla havaittiin olevan duplikaatti mittarilistauksessa. Tässä vaiheessa mittarilistauksessa oli siten 68 mittaria 10 eri osa-aluekategoriassa. Mittareista 30 datan voisi alustavan arvion mukaan kerätä automaattisesti pääosin TFS-itemien käsittelyjen ja versionhallinnassa olevan koodin ja kooditapahtumien pohjalta, kun taas 38 mittarin datat pitäisi joko osittain tai kokonaan koostaa käsin tai esimerkiksi erilaisten skriptien avulla TFS-datan perusteella.

TAULUKKO 1 Seminaarikeskustelussa jatkoon valitut mittarit

N:o	Mittari (suom.)	Mittari (alkup.)	Data	Seuranta
1	Itemien kokonaismäärä	Total number of PBIs in the release/Sprint	Autom.	Sprintti
2	Tehtyjen itemien määrä projekteittain	The number of PBIs completed in the release/Sprint	Manuaal.	Sprintti
3	Tehtyjen itemien määrä suhteessa kaikkiin projekteittain	Sprint Backlog Completion (SBC)	Manuaal.	Sprintti
4	Onko uusi sprintti aloitettu 24h edellisen sprintin päätöksestä?	Sprint Latency (SL)	Manuaal.	Vuosi
5	Sprintistä toiseen siirtyvien itemien määrä projekteittain	Siirtyvien itemien määrä sprintistä toiseen	Autom.	Suunnittelujakso
6	Vanhin valmistunut asentamaton feature	Oldest done feature (ODF)	Manuaal.	Vuosi
7	Valmistuneiden featureiden lukumäärä	Features per month/muu ajanjakso (FPM) (valmiit vs uudet)	Manuaal.	Suunnittelujakso
8	Julkaisujen lukumäärä	Releases Per Month (RPM)/muu ajanjakso	Autom.	Vuosi
9	Featuren käyttö	Feature usage	Manuaal.	Vuosi
10	Featuren liiketoimintahyöty	Featuren liiketoimintahyöty (tarkka määritys, että miten mitataan)	Manuaal.	Vuosi
11	Storyjen lukumäärä	Number of stories	Autom.	Suunnittelujakso
12	Storyn taskien valmistuneet tuntimäärät suhteessa kokonaismäärään	Story complete percentage	Manuaal.	Viikko
13	Taskien toteutuneet tunnit	Effort expended on tasks	Manuaal.	Viikko
14	Taskien tehdyt tunnit suhteessa jäljellä oleviin	Remaining task effort	Manuaal.	Viikko
15	Tehtyjen taskien lukumäärä	Tasks done	Manuaal.	Viikko
16	Keskimääräiset taskien ennustetut ja tehdyt tuntimäärät	Average time to complete a task (estimated and actual)	Manuaal.	Vuosi
17	Hylättyjen itemien määrä katselmoinnissa	Number of items approved or disapproved in the Review meeting	Manuaal.	Vuosi
18	Aktiivisten itemien määrä	Work in progress	Manuaal.	Sprintti
19	Storyn tila- ja asennusvaiheiden päivämääräsiirtymä	Storyn vaiheiden etenemisen aikamääreet	Manuaal.	Suunnittelujakso
20	Lukumäärä itemien tilasiirtymistä Removed-tilaan	Kuinka paljon siirretään storyjä Removed-tilaan	Autom.	Vuosi
21	Lukumäärä storyjen siirtymistä Määrittely-vaiheeseen	Kuinka paljon siirretään storyjä Määrittely-tilaan	Manuaal.	Vuosi
22	Testausvaiheessa havaittujen bugien määrä	Number of defects discovered during testing process	Manuaal.	Vuosi
23	Uusien tai aktiivisten bugien määrä työjonossa	Open defects	Autom.	Suunnittelujakso

TAULUKKO 1 jatkuu

N:o	Mittari (suom.)	Mittari (alkup.)	Data	Seuranta
24	Bugin luontipäivämäärä suhteessa nykyhetkeen	Defect age	Manuaal.	Vuosi
25	Bugien määrä	Defects per iteration	Autom.	Suunnittelujakso
26	Havaittujen bugien määrä katselmoinnissa	The number of errors found during the Sprint review meeting (for each PBI)	Manuaal.	Vuosi
27	Bugin korjausaika (tilasiirtymä välillä Uusi-Suljettu)	Bug correction time from new-to-closed state	Manuaal.	Vuosi
28	Bugien kokonaismäärät havaittuna kehityksen/testauksen aikana	Virheet kehityksessä/testauksessa vs. vuoden käytön aikana havaitut	Autom.	Vuosi
29	Bugien kokonaismäärä backlogilla	Bugien kokonaismäärä backlogilla	Autom.	Vuosi
30	Testitapausten määrä testaussuunnitelmassa	Number of test cases	Manuaal.	Vuosi
31	Yksikkö- ja integraatiotestien lukumäärä	Number of tests	Autom.	Vuosi
32	Hyväksyttyjen testitapausten määrä testaussuunnitelmassa	Number of passed test cases	Manuaal.	Vuosi
33	Julkaisutestauksen kesto	Releasen hyväksymistestauksen kesto	Manuaal.	Vuosi
34	Projektin burndown chart	Product (project) burndown chart	Manuaal.	Suunnittelujakso
35	Työnosituskaavio	Work breakdown structure chart	Autom.	Suunnittelujakso
36	Sprintin burndown chart	Sprint burndown chart	Manuaal.	Sprintti
37	Henkilöresurssikaavio	Henkilöresurssikaavio (tunteja per henkilö esim. per sprintti)	Manuaal.	Sprintti
38	Epäonnistuneen buildin korjausaika	Fix time of failed build	Autom.	Vuosi
39	Julkaisun läpimenoaika	Releasen lead time	Manuaal.	Vuosi
40	Suunniteltu kokonaistyömäärä	Planned effort	Manuaal.	Sprintti
41	Toteutunut kokonaistyömäärä	Actual effort	Manuaal.	Sprintti
42	Työteho	Work capacity	Manuaal.	Vuosi
43	Suunnittelemattoman työn määrää	Unplanned work	Manuaal.	Suunnittelujakso
44	Työn kustannus	Cost of development hour (feature)	Manuaal.	Suunnittelujakso
45	Kommenttirivien määrä	Lines of Comments (LCOMM)	Autom.	Vuosi
46	Koodirivien määrä	Lines of Source Code (SLOC)	Autom.	Vuosi

TAULUKKO 1 jatkuu

N:o	Mittari (suom.)	Mittari (alkup.)	Data	Seuranta
47	Muokattujen luokkien/koodirivien määrä	Number of Revisions (NR)	Autom.	Vuosi
48	Testikoodikattavuus	Test code coverage	Autom.	Suunnittelujakso
49	Koodin kompleksisuustestaus	McCabe cyclomatic complexity	Autom.	Vuosi
50	Deprekoituneen koodin määrää	Kuinka paljon deprekoitunutta koodia	Autom.	Vuosi
51	Toistuvan koodin määrä	Kuinka paljon toistoa koodissa (kts. Yo menetelmät)	Autom.	Suunnittelujakso
52	Automaattideploymenttien onnistumisprosentti	Deployment success rate	Autom.	Vuosi
53	Automaattitestien läpäisyprosentti	Automated test pass percentage	Autom.	Vuosi
54	Julkaisujen kehitykseen käytetty aika	Development time (releasen tasolla koko ajan kehitysaika)	Manuaal.	Vuosi
55	Automaattideploymenttiin mennyt aika	Deployment time	Autom.	Vuosi
56	Epäonnistuneiden automaattitestien suhde kokonaistestimäärään	Test failure rate	Autom.	Vuosi
57	Onnistuneiden automaattitestien suhde kokonaistestimäärään	Test success rate	Autom.	Vuosi
58	Build-tilojen aikamäärien (onnistunut/epäonnistunut) suhde	Build status (build ok vs build not ok aikasuhde)	Autom.	Vuosi
59	Buildien määrä per feature	Build määrä featuressa	Autom.	Suunnittelujakso
60	Buildin kesto	Build time	Autom.	Vuosi
61	Featuren tai storyn ennakoidut tuntitarpeet	Effort estimate	Manuaal.	Sprintti
62	Committien päivittäiset lukumäärät	Check-ins per day	Autom.	Suunnittelujakso
63	Ylläpitotehtäviin käytettävien työtuntien määrä kokonaismäärästä	Maintenance effort	Manuaal.	Suunnittelujakso
64	Deploymenttien lukumäärät julkaisuissa, niiden keskimääräiset aikavälit	Deployment frequency	Autom.	Vuosi
65	Tikettien määrä	Customer tickets	Autom.	Vuosi
66	Käytössä olevien resurssien suhde toteutuneiden tuntien määrään	Resource utilization	Manuaal.	Suunnittelujakso
67	Taskien lopullisten tehtyjen tuntien määrä ennakoituun	Accuracy of Estimation	Manuaal.	Suunnittelujakso
68	Suljettujen storyjen ja taskien määrä suhtessa ennakoituun	Business value delivered	Manuaal.	Sprintti

Täydennetyn taulukko 1 mukaisen mittarilistauksen pohjalta käydyssä keskustelussa mittareita priorisoitiin ja niistä valittiin 13 mittaria tärkeimmiksi ja ensimmäisenä käyttöönotettavien mittarien joukkoon. Valitut mittarit olivat sprintistä toiseen siirtyvien itemien määrä projekteittain, featuren liiketoimintahyöty, storyn taskien valmistuneet tuntimäärät suhteessa kokonaismäärään, keskimääräiset taskien ennustetut ja tehdyt tuntimäärät, storyn tila- ja asennusvaiheiden päivämääräsiirtymä, testausvaiheessa havaittujen bugien määrä, sprintin burndown chart, julkaisun läpimenoaika, suunnittelemattoman työn määrä, testikoodikattavuus, ylläpitotehtäviin käytettävien tuntien määrä kokonaismäärästä, käytössä olevien resurssien suhde toteutuneiden tuntien määrään ja suljettujen storyjen ja taskien määrä suhteessa ennakoituun. Yksikään mittari ei saanut neljää ääntä eli ääntä jokaiselta osallistujalta, mutta neljä mittaria sai kolmen osallistujan äänet. Lopuilla valituista mittareista oli joko yksi tai kaksi ääntä. Valittujen lisäksi 15 mittaria sai yhden äänen, mutta ei tullut valittua lopulliseen listaukseen. Valituissa mittareissa oli mukana mittareita jokaisesta mittarikategoriasta paitsi automatiikka, ja yhdeksän valittua mittaria pohjautui kirjallisuuteen ja neljä oli keskustelun kautta listauksen lisättyjä. Mittareista vain kahden data tulee automaattisesti TFS-itemien käsittelyn pohjalta, muihin vaaditaan joko täysin omaa dataa kuten featuren liiketoimintahyöty tai muuten esimerkiksi kehittäjien lisäämiä tietoja kuten erilaiset työjonotehtäviin käytettävät tuntimäärät.

13 valitusta mittarista yhdeksän oli siis kirjallisuuden pohjalta kerättynä. Näistä yhdeksästä mittarista Agile-kirjallisuudesta tuli 3 mittaria (storyn taskien valmistuneet tuntimäärät suhteessa kokonaismäärään, testausvaiheessa havaittujen bugien määrä ja ylläpitotehtäviin käytettävien tuntien määrä kokonaismäärästä), kuten myös Scrum-lähteistä (keskimääräiset taskien ennustetut ja tehdyt tuntimäärät, sprintin burndown chart ja testikoodikattavuus) ja DevOps-kirjallisuudesta 2 mittaria (suunnittelemattoman työn määrä ja käytössä olevien resurssien suhde toteutuneiden tuntien määrään). Yksi mittari, eli liiketoimintahyöty esiintyi kaikissa kolmessa edellä mainitussa kategoriassa alkuperäisessä mittarilistassa.

7 TULOSTEN TULKINTA

Keskustelevalla ja iteratiivisella otteella luotiin vuokaaviotyyppiset diagrammit havainnollistamaan tiimin ohjelmistokehityksen prosesseja. Tiimillä oli käytössä TFS-ohjelmisto työtehtävien ja versionhallinnan hoitamiseen. Kaavioissa siten mallinnettiin tiimin käytössä olevat TFS-työjonojen tehtäväluokkien käsittelyt ja niihin liittyvät versionhallinnan toimet. Scrumin eri roolien vastuut ja tehtävät työjonotehtävien siirtelystä jätettiin mallinnuksesta pois keskittyen vain itse tehtävien prosessiin tiimin toiminnassa. Ylimmällä tasolla tehtäväluokista oli feature, joka koostuu user story -tason tehtävistä, minkä alla taas on taskeja. Niistä erillisenä tehtäväluokkanaan oli bugi.

Featuren läpiviennissä luodaan ensin feature tietoineen ja sitten lisätään sen alaiset storyt ja myöhemmin niille taskit. Koska storyt luodaan aina featuren alaisiksi, pitää featuren tiedot olla oikein täytettynä ennen kuin niille voidaan lisätä storyjä. Kaavio ei ota tarkemmin kantaa miten tietojen hyväksymisprosessi etenee, toisin kuin storyn yhteydessä, mutta mahdollistaakseen hyvin kuvatut user storyt, pitää featuren tiedot olla riittävän tarkat, kattavat ja oikealla tavalla kirjatut. Scrum-oppaan mukaan tuoteomistaja on vastuussa kehitysjonon hallinnasta ja jonon kohtien selkeästä ilmaisusta (Schwaber & Sutherland, 2017). Tuoteomistajan ja tiimin pitääkin olla hyvin sopinut keskenään, miten featuren tiedot täytetään, jotta niiden pohjalta voidaan user storyjä tehdä.

Featuren valmistumista edeltää storyjen valmistumien jälkeen featuren hyväksymistestaus. Näin ollen prosessissa otetaan tarkasti kantaa siihen, että missä vaiheessa tiimin työtä testataan ja se hyväksytetään, eli nyt featuren tasolla. Tiimillä on siten vapaammin mahdollisuuksia tehdä featuren alaisia user storyn -tason tehtäviä ja hyväksyttää ne yhtenä kokonaisuutena featuren tasolla. Tämä myös mahdollistaa featuren hyväksyjille mahdollisuuden tarkastella laajempia kokonaisuuksia, eikä yksittäisiä user storyjä. Featuret pitääkin olla suunniteltu kerralla testattaviksi kokonaisuuksiksi. Kaaviosta käy myös ilmi, että mikäli hyväksymistestaus ei mene läpi, pitää tarkastella featuren ja storyjen sisältöä ja mahdollisesti muokata niitä. Hyväksymiseen mahdollisesti vaadittavat muutokset tulisi luonnollisesti kirjata ylös työjonolle ja siten featuren alaisia tehtäviä pitää sen mukaan muokata. Jos työt jäävät kirjaamatta, eivät työjonon

tehtävien ja niiden sisältöjen käsittelyä mahdollisesti seuraavat mittarit välttämättä saa kiinni kaikkea työhän käytettävää aikaa, jolloin mittareiden antamat tiedot voivat olla vääriä ja johtaa vääriin johtopäätöksiin.

Storyn luontiin tai muokkaukseen ja toteutukseen luotiin omat kaaviot. Storyn luonnissa/muokkauksessa tulee tarkemmin kuvattua storyjen sisältöjen hyväksyminen ennen sen jakamista taskeihin ja toteuttamista. Storyn kuvaus ja hyväksymiskriteerit pitää olla riittävän tarkasti kuvattuna, että se voidaan hyväksyä ja jakaa varsinaisiksi tarkemmiksi työtehtäviksi. Tiedot kertovat kehittäjälle, miten työ pitäisi toteuttaa, ja mikäli tieto ei ole relevanttia tai se on huonosti kuvattua, ei työn tulos ole välttämättä sitä, mitä työn määrittelijä on halunnut. Prosessissa on siten huomioitu tietojen oikein täyttämisen tärkeys storyn tilasiirtymiä hyödyntämällä. Siirtymistä voi esimerkiksi lähettää automaattiviestejä eri osapuolille, jotta ennen seuraavaa tilasiirrosta tiedot ovat laajemmalla joukolla hyväksytetyt.

Storyn toteutuksessa jokaisen storyn tekeminen aloitetaan ottamalla viimeisin toteutuksen tila release-haarasta omaksi story-haarakseen. Release-haara pitää näin ollen olla aina ajan tasalla. Storyn toteutuksen loppuvaiheessa tuotos viedään taas takaisin release-haaraan, jolloin pitää varmistaa, että kehityksen aikaiset mahdolliset muutokset on sisällytetty mukaan haarojen yhdistämiseen, ettei jo tehtyä työtä häviä tai haarojen yhdistäminen aiheuta konfliktitilanteita haarojen välille. Useamman kehittäjän tehdessä töitä samaan projektiin, pitää kehityksen haaroituspolitiikat olla selvästi kuvattuja ja kaikkien tiedossa, ja lisäksi kaikkien osallistujien on oltava ajan tasalla varsinkin aktiivisen haaran tilanteesta ja sinne tehtävästä työtä. Storyn toteutuksen kaaviossa ensimmäisessä vaiheessa kuvataankin, kuinka aluksi story merkitään tietylle kehittäjälle ja sen tila vaihdetaan aktiiviseksi, jolloin kaikki tietävät, että milloin mitäkin tehtävää edistetään ja kenen toimesta. Tilasiirtymien kautta kaikki sidosryhmät voivat seurata sprinttien tilannetta story-tasolla, ja myös storyn alaisten taskien tasolla. Story on ennen toteutusta jaettu taskien perustuen storyssä oleviin tietoihin. Task-iteraatiossa storyn taskeja tehdään tekemistä koko ajan versioiden ja dokumentoiden. Toteutuksen lopussa on tärkeää verrata taskien mukaisesti tehtyä kokonaisuutta storyn tietoihin, jotta mahdolliset puutteet saadaan heti kirjattua joko olemassa oleviin taskeihin tai uusiin.

Bugin luonti vastaa paljon storyn luonnin prosessia. Oleellisina eroina ovat, että bugeja ei jaeta taskeihin ja jo bugin luonnin yhteydessä tehtävä siirto sprintille. Bugit tulevat usein kesken sprintin ja ovat kriittisyydeltään vaihtelevan tasoisia. Kun bugin on luotu ja hyväksytty validiksi bugiksi, pitää heti päättää missä vaiheessa se toteutetaan. Kriittisyydeltään tärkeät bugit pitää ottaa heti käsittelyyn tai toteutettava kuluvan sprintin aikana, vähemmän kriittisemmät bugit myöhemmissä sprinteissä. Näin ollen tiimillä pitää olla yhdessä sovitut tavat arvottaa bugien kriittisyys ja sitä myöten niiden sijoittaminen sprintille. Sprintillä pitää olla myös tilaa sijoittaa mahdollisia bugeja kesken toteutuksen. Scrum-oppaassa (Schwaber & Sutherland, 2017) esitetään, että sprintin suunnittelupalaverin lähtökohtana ovat kehitysjono ja viimeisin inkrementti, sekä kehitystiimin kapasiteetti ja aiempi suorituskyky. Tiimin tulee siten pystyä

aikaisemman tekemisen kautta arvioimaan, kuinka paljon eri tehtäviä se pystyy sprintissä toteuttamaan. Sprintille pitää suunnittelussa jättää tilaa esimerkiksi juuri bugeille tai muille odottamattomille tapahtumille, sekä toisaalta muille tiedetyille tapahtumille, kuten Scrumin tapahtumatapaamiset. Toinen tärkeä tieto myös kehitettävien järjestelmiensä ylläpitovastuussa olevalle kehitystiimille on pystyä arvioimaan, kuinka paljon kehitettävistä sovelluksista keskimäärin tiettynä aikajaksona tulee bugi-ilmoituksia tiimin käsiteltäväksi, ja kuinka paljon niistä päätyy sprintille käsiteltäviksi bugeiksi. Kaikki käyttäjiltä tiimille tulevat bugi-ilmoitukset eivät välttämättä ole bugeja tai, että ilmoitusten pohjalta lopulliseen toteutukseen tehtäisiin korjaus, vaan bugit voivat olla myös ohjelman piirteitä tai johtaa laajempaan ohjelman muutokseen esim. feature-tasolla, mikä voikin aiheuttaa ongelmia esimerkiksi analysoitaessa bugien ilmaantumisia tai arvioitaessa niiden syntymistä (Herzig ym., 2013).

Bugin toteutuksessa toteutetaan vain itse bugin mukainen työ ilman jakamista alatehtäviin, jolloin bugin etenemistä voi seurata bugin omien tilasiirtymien ja lisätietojen avulla. Lisäksi bugilla ei ole yläluokitusta, joten sen itsessään pitää läpäistä hyväksymistestaus ja mahdolliset muutokset työhön tehdään vain itse bugin tietoihin. Muutoin bugin toteutus vastaa luonnin ohella paljon storyn toteutusta.

Keskustelujen myötä päätettiin mallintaa myös TFS:n työtehtävien ulkopuolelle jäävä release-taso sen haaroituskäytänteiden kautta. Julkaisujen suunnittelussa ja hallinnassa tiimi sopii sidosryhmien kanssa missä vaiheessa mikäkin toteutus lisätään osaksi jo toimivaa ohjelmistokokonaisuutta. Jo aikaisemmin storyjen yhteydessä mainittu release-haara vastaa juuri tietylle julkaisulle varattua haaraa, mihin kehitys sovitun julkaisusuunnitelman mukaan lisätään. Julkaisujen kautta tiimin versionhallinnassa käyttöön tulevat myös ns. ylempien tasojen haarat, joita ovat master ja production -haarat. Jälleen tärkeäksi nousee suunnitella tiimin haaroituspolitiikka, nyt myös ylemmillä tasoilla, eli milloin on lupa yhdistää release-haara tuotantosiirtoa odottavaan master-haaraan ja missä vaiheessa taas tuotantoa vastaavaan production-haaraan. Nyt tiimi on päättänyt käyttää varsin usein käytetyn master-haaran ohella myös production haaraa. Usein master kuvastaa tuotannon tilannetta, mutta nyt se on eräänlainen odotustila varsinaista tuotantosiirtoa varten, ja tuotannon tilanne ja aina tuotantoon siirrettävissä oleva koodi on production-haarassa. Haaroituksilla ei niin suurta väliä Git-pohjaisessa versionhallintajärjestelmässä ole itse järjestelmän kannalta, mutta kaikkien on silti tiedettävä yhteinen tapa, miten sitä toteuttaa, jotta kaikki haarat toimivat ja ovat ajantasaisia omissa vaiheissaan.

Tiimin käytössä oleva TFS-järjestelmä mahdollistaa Git-pohjaisen hajautetun versionhallinnan käytön, ja tiimillä onkin Git käytössä. Gitin, tai yleisemmin hajautetun versionhallinnan, hyötyinä ovat esimerkiksi keskitetyt tiedon ja prosessiautomaation käsittely, testaus, arviointi, keskustelu ja takaisinhaaroituskäytäntö ns. pull request (Yu ym., 2015). Pull requestit ovat tiimillä käytössä storyjen, bugien ja releasen yhteydessä koodin katselmoinnin työkaluna, mikä näkyy ko. työjonotehtävien ja release-toteutuksen prosessikaavioissa. Yleisesti ohjelmistokehityksen pull-mallissa versionhallinta ei ole suoraan jaettu osallis-

tujien kesken vaan se on hajautettu niin, että jokainen toteuttaja ottaa versionhallinnasta itselleen kopion haluamastaan haarasta ja sen jälkeen tekee kyseiseen haaraan muutoksia toisista riippumatta (Gousios ym., 2015). Muutoksia tehdään ja tallennetaan Git-pohjaisessa hajautetussa mallissa paikallisesti haluttuun haaraan, mutta haaran tiedot voidaan tallentaa, ja usein tallennetaankin, push-viestien kautta myös varsinaiseen versionhallintaan, jotta muutokset eivät ole vain yhden kehittäjän koneella. Muutokset ovat kuitenkin tässä vaiheessa vielä aina omassa haarassaan. Kun muutokset ovat valmiit, liitetään ne osaksi myös muiden kehittäjien käyttämää haaraa, joka on usein juuri sama haara, mistä kehittäjän oma haara on otettu. Tässä vaiheessa varsin usein käytetään juuri pull requesteja, missä kehittäjä tekee pyynnön yhdistää oma haara ns. ylemmän tason yleisessä käytössä olevaan haaraan (Gousios ym., 2015). Pyyntö menee muille kehityksessä mukana oleville ja heistä sovittu määrä osallistujia tarkistaa pyynnön sisällön, ennen kuin se hyväksytään osaksi toista haaraa (Gousios ym., 2015).

Viimeinen mallinnettu kaavio oli koodin automaattiseen integraatioon ja jatkuvaan toimitukseen liittyvä kaavio. Kuten pull requestit, niin myös jatkuvan integraation ja toimituksen prosessit ovat mukana kaikilla tiimin versionhallintaa päivittävien työjonotehtävien toteutuksessa. Automaatio auttaa nopeuttamaan ohjelmistojen asennuksia kohdepalvelimille ja toisaalta tekee siitä yhtenevää eri kertojen välille. Ketterissä menetelmissä ei kovin usein oteta kantaa esimerkiksi jatkuvan toimituksen prosesseihin, mutta se on vahvasti mukana osana DevOps-tyyppistä kehitystä. Ottaen jatkuvan integraation ja toimituksen osaksi kaikkia versionhallinnan haarojen yhdistämisiä, tavoitteleekin tiimi selkeästi DevOps-maista kehittämistä ja DevOpsin kuvaamia hyötyjä. Tällainen automatiikka osana ohjelmistokehitystä on myös mittareiden kannalta varsin mielenkiintoinen lisä kehitysprosessiin. Jokainen automaattisen jatkuvan integraation ja toimituksen prosessin vaiheista on luotava etukäteen ja määritettävä siihen apuna käytettävään alustaan tai ohjelmistoihin niiden toiminnan osapalasiksi. Näin ollen jokainen vaihe tunnetaan ja ne etenevät automaattisesti esimerkiksi versionhallinnan muutosten osana. Siksi prosessiin liitetyt mittarit voisivat myös toimia automaattisesti osana kokonaisuutta ja tuottaa varsin reaaliaikaisesti mittaridataa prosessin seuraajille. Mittarit ovat myös automatiikan myötä hyvin verrattavissa aikaisempiin tuloksiin siten parantaen niiden luotettavuutta arvioitaessa vaihteluja mittarin tulosten välillä. Esimerkiksi Lehtonen ym. (2015) selvitti jatkuvaan toimitukseen liittyviä mittareita, joita he listasivat olevan esimerkiksi featuren kehitys- ja tuotantoonsiirtoaika. He huomauttivat, että jatkuvassa toimituksessa mahdollista mittaridataa voidaan saada paljon ja useasta eri mittarista, usein varsin pienellä lisätyöllä (Lehtonen ym., 2015). Onkin tärkeää valita seurattavat mittarit tarkkaan, ettei vain ota kaikkea mahdollista saatavilla olevaa dataa, jolloin ei välttämättä pysty olennaisimpia asioita tarkasti seuraamaan. Lehtonen ym. (2015) myös huomauttavat, että heidän tapauksessaan automaattisesti tulevat mittarit jatkuvan toimituksen prosesseista koskettavat vain kehittäjiä, ja mikäli halutaan käyttäjiä koskevaa tai toisaalta tuotteen hallintaan liittyvää dataa, pitää prosessiin tehdä muutoksia ne huomi-

oiden. Yleisesti ottaen mittareiden kannalta pitääkin varoa vauhtisokeutta, vaan valita sopiva määrä mittareita, jotka vastaavat niitä toiveita ja kohderyhmiä, mistä dataa halutaan kerätä. Lisäksi kaikki haluttu data ei tule välttämättä automaattisesti, vaan voi vaatia omien lisäosien rakentamista osaksi valmiita julkaisemisen prosesseja.

Schwaber (1997) esitti Scrum-prosessikaavion, jossa kuvataan Scrumin eri vaiheita ja tapahtumia. Usein Scrum-prosessikuvauksissa näkeekin juuri Scrumin tapahtumia, tuotoksia ja rooleja, siis asioita, joilla Scrumia määritellään. Scrumin prosessista on myös esitetty erilaisia jatkokehitysmalleja ja muunnoksia, kuten esimerkiksi Safe Scrum (Stålhane ym., 2012), IScrum (Ashraf & Aftab, 2017) ja APLE Scrum (Dıaz ym., 2011). Mallit vastaavat paljon alkuperäistä, eikä niissä mennä kovinkaan tarkalle tasolle itse kehityksen osalta. Ne vain hieman laajentavat tai muokkaavat alkuperäistä mallia. Edellisiä tarkemmalla tasolla Scrum-johdannaisprosessia kuvaa Maier ym. (2017) esittämä Secure Scrum – prosessi, jossa integroidaan turvallisuustoimia Scrum-prosessiin. Kyseisessä prosessimallissa turvallisuustoimien ohella kuvataan esimerkiksi työjonotehtävien määrittelyn ja toteutuksen valmiusasteen tarkistus.

Kun vertaa Maier ym. (2017) mallia tässä tutkimuksessa esitettyihin prosessikaavioihin, on tämän tutkimuksen mallit yleisen Scrum-prosessin kaavioita ja sisältävät tarkemmalla tasolla esimerkiksi työn laadun varmistusta pull request –tyyppisesti ja koodin automaattista julkistamista. Lisäksi tämän tutkimuksen malli on luotu keskustelussa Scrum-tiimin kanssa, kun taas Maier ym. (2017) loivat mallin ensin ja sen jälkeen tekivät kyselyn Scrum-tiimille sen arvioimiseksi. Kaiken kaikkiaan tässä tutkimuksessa esitetään Scrumin kehitysprosessia paljon tarkemmalla tasolla, kuin muissa tutkimuksissa, esimerkiksi huomioiden työjonotehtävien ohella jatkuvan kehityksen ja julkaisemisen toimia. Prosessimallit antavat siten mielenkiintoisen perspektiivin Scrumin soveltamiseen kehitystyössä tasolla, jota ei muissa tutkimuksissa ole esitetty, mikä varmasti kiinnostaa niin akateemisia lukijoita, kuin Scrumia työssään käyttäviä tahoja. On kuitenkin huomioitava, että malli on yhden Scrum-tiimin näkemys omasta kehitysprosessistaan, ja siten ei suoraan sovellettavissa muihin tiimeihin.

Mittareista käytävää keskustelua pohjustamaan kerättiin mittareita eri osa-alueilta. Osa-alueet valittiin tiimin kohdealueen perusteella: ohjelmistoja kehittävällä ja ylläpitävällä tiimillä oli käytössä ketteriin ohjelmistomenetelmiin kuuluva Scrum-viitekehys, johon tiimillä oli haluja yhdistää DevOpsin käytänteitä, missä taas yhtenä ulottuvuutena on ohjelmistokehityksen automaatio. Osa-aleilta kerätyistä materiaaleista listattiin 226 mittaria. Eniten mittareita listattiin osa-alueilta Agile, Scrum ja ohjelmistot tai ohjelmistokoodi, kun taas tieteellisiä DevOps-mittareita löytyi vain 8 ja automaatiosta 7 kappaletta. Mittarimäärien löytymisen jakaumaa selittää esimerkiksi ohjelmistokoodimittareiden osalta niiden pitkä historia. Ohjelmistokoodimittareita on käytetty jo 1960-luvulla kuten esimerkiksi koodirivien määrä (lines of code, LOC) (Fenton & Neil, 2000). Mittareita on siten myös ollut mahdollista tutkia usean vuosikymmenen ajan ja niistä löytyvää kirjallisuutta löytyy hyvin. Ketterä kehittäminen ja sen yksi suosituimmista menetelmistä Scrum ovat myös olleet suosittuja tut-

kimuksenkohteita ja iältäänkin ne ovat jo lähes kaksi vuosikymmentä vanhoja. Ketterän kehittämisen mittareista on tehty myös useampia koosteartikkeleita, joista esimerkkinä Kupiainen ym. (2015). Mittaroinnin kirjallisuutta on edellä mainituista osa-alueista siten selvästi paremmin saatavilla, kuin uudempien osa-alueiden osalta.

Kupiainen ym. (2015) toteavat, että heidän tuloksien pohjalta ketterän kehittämisen mittauskohteet ovat prosesseihin ja tuotteeseen kohdistuvia, eikä niinkään ihmisten toimia mittaavia. Tässä tutkimuksessa listattujen mittareiden osalta mukana on kuitenkin myös mittareita, jotka mittaavat ihmisten toimia, kuten esimerkiksi eri roolien eri tapahtumiin osallistumisen määriä, kehittäjien työn tehokkuutta tai tiimin motivaatiota tehdä työtä. Siten voikin olla, että yleisten ketterän kehittämisen menetelmien ja kokonaisuuden ulkopuolelta listataan suhteessa enemmän mittareita, kuin mitä niiden sisällä on. Toisaalta alkuperäisessä listauksessa Agile-kohdan alta löytyy myös esimerkiksi juuri kehittäjän työn tehokkuuteen liittyviä mittareita, joten myös ketterän kehittämisen alueella käytetään ihmisten toimia mittaavia mittareita.

Seminaaripäivän keskustelun myötä ennalta kerätyistä mittareista kiinnostaviksi mittareiksi nostettiin mittareita jokaiselta osa-alueelta tasaisesti, pois lukien keskustelun aikana luotu uusi kategoria suunnittelujakso. Laajasta 181:n mittarin listauksesta valituksi tuli noin 30 prosenttia mittareista ja niin, että määrällisesti pienemmistä osa-alueista tuli valituksi suhteellisesti selvästi enemmän kuin isoimmista osa-alueista. Lisäksi keskustelun aikana lähes jokaiselta osa-alueelta nostettiin kiinnostavaksi mittariksi yksi tai useampi uusi osallistujia kiinnostava mittari. Mittareiden kartoittaminen ja jakaminen eri kategorioihin siten onnistui herättämään kiinnostusta erilaisiin ja eri kohteita mittaaviin mittareihin. Lisäksi keskustelu mahdollisti uusien ja osallistujien omien näkökantojen esiin tuonnin eri kategorioiden yhteydessä, jolloin mittarilistaus laajeni lähes jokaisen kategorian yhteydessä.

Suuren alkumäärän ja eri kategoriasta monen kiinnostavan mittarin valitsemisen myötä mittareita oli tässä vaiheessa käsiteltävänä vielä varsin suuri määrä, 68. Mittareita rajattiin priorisointikeskustelun myötä pienemmäksi mittarilistauksesta, jossa jokaisen mittarin tietoja oli täydennetty mahdollistamaan parempi keskustelu ja arviointi mittareiden välillä. Lopulliseen listaukseen valittiin 13 mittaria, joista yhdeksän oli alkuperäisestä mittarilistauksesta valittuja ja neljä seminaarikeskustelussa esiin nostettuja. Priorisointi ei siten suosinut erityisesti omia tai ennalta kerättyjä mittareita. Toisaalta 68 mittarin datasta alle puolen datan pystyisi keräämään pääosin TFS-järjestelmän avulla automaattisesti ja loppujen osalta se vaatii osittain tai kokonaan manuaalityötä. Valituista 13 mittarista vain neljän datan saisi automaattisesti ja loppujen osalta pitäisi prosesseihin lisätä juuri mittaridataa keräämiä osia. Toisaalta mittareita valittiin lopulliseen listaukseen tasaisesti eri kategorioista. Näin ollen mittarin valintaan vaikutti enemmän mittarin sisällön tärkeys, kuin sen keräykseen erikseen vaadittava työn määrä tai kategoria. Tosin osallistujat saattoivat äänestää mittareita tarkoituksella eri kategorioista, jotta ne kaikki tulisivat edustetuksi lopun mittareissa ja työn seurannassa.

Kupiainen ym. (2015) listasivat tutkimuksessaan mittareita heidän määrittelemän vaikuttavuuden mukaan. Näin ollen on mielenkiintoista verrata heidän kirjallisuuden pohjalta priorisoimaa mittarilistausta tässä tutkimuksessa listattuihin 13 korkeimmalle priorisoituun mittarilistaan. Tämän tutkimuksen lopullisesta mittarilistasta kolme oli mukana Kupiainen ym. (2015) kokoaman listan 13 parhaan mittarin joukossa. Tosin kaikkien sisällöt eroavat hieman tämän ja Kupiaisen tutkimuksen välillä. Kupiainen ym. (2015) listauksesta löytyivät julkaisun läpimenoaika yleisesti ja virheiden lukumäärä, jotka tässä tutkimuksessa vastaavat mittareita julkaisun läpimenoaika ja testausvaiheessa havaittujen bugien määrä. Lisäksi tässä tutkimuksessa valittiin suljettujen storyjen ja taskien määrä suhteessa ennakoituun. Kupiainen ym. (2015) listalla oli mukana mittari story complete percentage, joka sisällöllisesti on sama kuin edellä mainittu tämän tutkimuksen mittari. Tämän tutkimuksen mittareista muutama oli lähellä niitä, mitä Kupiainen ym. (2015) listaavat, mutta suurimmalta osin listaukset eroavat. Tuloksissa siten näkyy yksittäisen tiimin heidän tarpeisiinsa valitsema lista, jossa keskusteluun osallistujat arvottavat mittareita heidän näkökulmiensa kautta. Tuloksissa siten korostuu enemmän halu saada itselle mielenkiintoisista kohteista mittaridataa kuin, että ketterän kehittämisen alueella olisi useita kaikkialla päteviä mittareita. Kupiainen ym. (2015) mainitsevat myös, että vaikuttavuuden arviointi heidän listan tapauksessa on varsin subjektiivista ja tapauskohtaista.

Yksittäisistä valituista mittareista mielenkiintoinen Scrum-tiimissä tuoteomistajille voisi olla alkuperäisestä mittarilistauksesta jatkoon valitsematta jätetty läpimenoaika, joka kuitenkin keskustelussa nostettiin esille erityisesti juuri featuren yhteydessä mielenkiintoiseksi, eli mittaustaso haluttiin määrittää juuri featuren tasolle. Feature oli ylin mallintamisessa ja myös mittarikeskustelussa käytetty työjonotehtävien taso ja se sisälsi myös tiimin kehitysprosessissa hyväksymistestauksen. Featuren voikin nähdä olevan tuoteomistajien tärkeimpiä työkaluja työjonolla ja sen läpimenon seuranta siten eniten kohdennettu juuri tuoteomistajille. Toinen myös todennäköisimmin erityisesti tuoteomistajia ja sidosryhmiäkin kiinnostava mittari oli featuren liiketoimintahyöty, jonka tarkempaa kuvausta tosin ei keskustelussa määritetty. Scrum Masterille todennäköisimmin mielenkiintoisia mittareita voisi olla esimerkiksi sprintin burndown chart, storyjen valmistuneiden tuntien määrä suhteessa kokonaismäärään tai sprintistä toiseen siirtyvien itemien määrä projekteittain. Edellä mainitut mittarit kertovat Scrumin mukaisen ohjelmistokehityksen edistymisestä ja toisaalta esimerkiksi sprinttien suunnittelun onnistumisesta. Jos sprintin suunnittelussa ei osata tarpeeksi hyvin ottaa huomioon tiimin kykyä tehdä työjonolla olevia tehtäviä tai toisaalta tunneta tehtävien järjestelmien ominaisuuksia tehtävän työn tai esimerkiksi ilmaantuvien virheiden määrän osalta, niin työt eivät etene suunnitellusti ja jäädään burndown-kaaviossa jälkeen ennakoidusta ja tehtäviä siirtyy sprintiltä toiseen. Kehitystiimiä voisi taas valituista mittareista erityisesti kiinnostaa esimerkiksi testikoodikattavuus ja testivaiheessa havaittujen bugien määrät. Suuri testikoodikattavuus helpottaa esimerkiksi koodin muokkausten ja refaktorointien yhteydessä, koska aikaisempaa toiminnallisuutta muuttavat

muutokset voivat aiheuttaa testien epäonnistumisprosentteja, jolloin samalla pitää tulee tarve katsoa tehtyjä muutoksia laajemmin ja selvittää miksi testi ei mene läpi. Bugien määrä taas kertoo myös koodin ja toteutuksen laadusta, ja mikäli bugeja paljon ilmenee, voi tulla kyseeseen lisätä työjonolle tehtäviä esimerkiksi koodin refaktorointiin tai teknisen velan poistoon liittyen.

8 JOHTOPÄÄTÖKSET

Tämän tutkimuksen tarkoituksena oli auttaa tiimiä löytämään heidän toiminnalleen sopivia mittareita Scrum- ja DevOps-tyyppisen ohjelmistokehityksen tueksi. Tutkimuksessa luotiin lähtökohta tiimin mittarivalinnoille määrittämällä heidän ohjelmistokehityksen prosessit käsiteltävien työjonotehtävien osalta vuokaaviotyyppisiin prosessikuvauksiin. Prosessien kuvauksen jälkeen käytyjen keskustelujen aikana valittiin tiimille heitä eniten kiinnostavat mittarit. Tutkimusote oli iteratiivinen, edellisiä tuotoksia laajentava ja tarkentava, jotta suuremmista ja määrittelemättömistä kokonaisuuksista saatiin kasaan tiimiä hyödyttäviä kaavioita ja mittarilistauksia.

Ensimmäinen tutkimuskysymys oli, että mikä on tiimin käytössä oleva Scrum-prosessin mukainen kehitysprosessi. Tutkimuksessa kuvattiin tiimin prosesseja usealla eri työjonotehtävän tasolla, joita olivat feature, story, task ja bugi. Lisäksi mallintamisen yhteydessä lisättiin kaaviot julkaisujen hallinnalle ja automaattiselle koodin julkaisemiselle, jotka molemmat havaittiin tärkeiksi osiksi tiimin ohjelmistokehityksen prosesseja. Prosessien kuvaaminen antoi hyvää pohjaa tiimille tarkastella omaa kehitysprosessiaan, koska kaavioita luotaessa tiimin piti keskustella eri vaiheista yksityiskohtaisesti ja sopia yhteisiä linjoja tekemiselle. Tutkimuksessa esitettiinkin useita kohtia, missä tiimin on tarkkaan keskusteltava ja sovittava, että miten kehitystä tehdään, ja kirjata se ylös kaikkien nähtäville. Lisäksi kaaviot havainnollistavat sen hetkisen näkemyksen toteutuksen mallista tiimille, josta voi tarkistaa miten on yhteisesti sovittu tehtäväksi, ja erityisesti se mahdollistaa pohjan tiimin oman prosessi kehittämiselle. Myös sidosryhmille kuvatut prosessit auttavat havainnollistamaan tiimin ohjelmistokehityksen vaiheita. Kun on olemassa kaavio työn vaiheista, voi hyvin huomata myös kohtia, joissa tiimin prosesseissa olisi kehittämisen aihetta tai, että joku vaihe ei ole kaaviossa kuvattu tai se on siellä eri tavalla ilmaistu, kuin se todellisuudessa toteutuu. Kaaviot ja niiden luominen ovat myös hyvä fasilitaattori tiimin sisäisen prosessin kuvaukselle ja jatkokehitykselle. Se on myös suunnitelma, miten kehitystä toteuttaa nyt ja tulevaisuudessa. Koska jos ei ole suunnitelmaa, ei ole myös suunnitelmaa, jota muuttaa.

Tutkimuksessa esitettyjen kaavioiden ulkopuolelle rajautui esimerkiksi eri roolien toiminta ohjelmistokehitysprosessissa. Lisäksi mallit koskevat vain ohjelmistokehityksen työtehtäviä ja siten esimerkiksi niiden suunnitteluun tai sidosryhmäyhteistyöhön toteutuksen suunnittelussa ja seurannassa ei otettu kantaa. Tiimillä on siten paljon asioita kaavioiden ulkopuolella, mitä pitää ottaa huomioon. Näin ollen, vaikka noudatellaankin hyvin dokumentoituja ja paljon käytettyjä ohjelmistokehityksen menetelmiä, toimintamalleja ja prosesseja, kuten esimerkiksi tiimin käytössä oleva Scrum-viitekehys, niin niiden ulkopuolelle jää paljon tehtäviä, toimintoja ja prosesseja, jotka ovat tärkeänä osana tiimin toimintaa, ja siten myös tärkeä tunnistaa ja kuvata osaksi tiimin toimintaa. Tutkimuksessa määritetyt kaaviot ovat hyvä pohja, jonka päälle rakentaa myös muiden tiimin toiminnan prosessien kuvauksia.

Toinen tutkimuksen tutkimuskysymys oli selvittää, että mitä mittareita tiimi haluaa osaksi kehitysprosessin seurantaa. Tavoitteena oli siten luoda tiimille kehitystyön tueksi pohja heitä kiinnostavista mittareista, joita tiimi voi tutkimuksen jälkeen lähteä ottamaan osaksi ohjelmistokehityksen prosessejaan. Mittareista kaikki eivät ole suoraan osana tai integroitavissa osaksi ensimmäisen tutkimuskysymyksen mukaista prosessikuvauskokonaisuutta. Tämä ei ollut tutkimuksen tavoitteenakaan, vaan ennemminkin prosessien kautta fasilitoida tiimille tilaisuus keskustella prosesseistaan ja kuvata heidän tulevaisuuden tavoitetilaa nykyprosessien pohjalta, minkä mukaista kehitystä tiimi tulevaisuudessa haluaa tehdä, ja siten luoda tausta myös mittareiden valitsemiselle ja siitä käytävälle keskustelulle. Tutkimuksessa ei haluttu rajoittaa mittareita vain juuri tiettyihin kuvattuihin kehitysprosessien kaavioiden vaiheisiin, vaan mahdollistaa tutkimuksissakin kuvattu tiimin omaan kohdealueeseen keskittyvien mittareiden valitseminen laajasta, tiimin toiminta-alueen mukaisesta mittarijoukosta.

Mittarikeskustelun pohjaksi löytyi laaja joukko kirjallisuuden pohjalta kerättyjä mittareita. Suurin osa mittareista oli tieteellisistä artikkeleista kerättyjä, mutta DevOps-osa-alueen osalta piti mittareita ottaa myös internet-lähteistä tieteellisten lähteiden vähyyden johdosta. Eri kategorioihin jaettu mittarilistaus mahdollisti eri alueisiin vaiheittain keskittyvän keskustelun, mikä mahdollisti myös laajemman alkuvaiheen mittarimäärän käsittelyn. Keskustelu onnistui hyvin auttamaan tiimiä ensin rajaamaan mittarijakaumaa pienemmäksi kategorioiden sisällä ja tämän jälkeen priorisoimaan alkuperäisestä yli 200 mittarin joukosta eniten tiimiä kiinnostavimmat reilu tusina mittaria, joissa mukana oli myös neljä keskustelussa esiin nostettua ja mittarilistan ulkopuolista mittaria. Siten myös mahdollistui vapaampi keskustelu tiimiä kiinnostavista mittareista, eikä tarvinnut pitäytyä vain pohjaksi listatuissa mittareissa.

Iteratiivinen ja aina edellistä tulosta jalostava tutkimusote oli hyvä tilanteessa, jossa prosessikuvauksia tai mittarilistauksia ei aikaisemmin ollut. Keskustellen tiimi pystyi tutkimuksen tekijän luoman pohjan kautta aina parantamaan, tarkentamaan ja toisaalta rajaamaan työssä läpikäytyjä aineistoja. Aihealueina kehitysprosessin kuvaus ja mittariston luominen ovat niin laajoja, että ne vaativat paljon aikaa ja valmistautumista. Normaalin työn ohessa tutkimuk-

sessa esitetyllä tavalla pystyttiin luomaan hyvä pohja niin prosessien kuvaukselle kuin tiimin toimintaa seuraavalla mittaristollekin.

Seuraava luonnollinen vaihe tutkimuksen jälkeen olisi ottaa tiimin valitsemia mittareita osaksi tiimin toimintaa ja ohjelmistokehityksen prosesseja. Osan valittujen mittareiden datasta saa suoraan esimerkiksi tiimin käytössä olevan TFS-järjestelmän kautta, mutta tiimin pitää myös allokoida aikaa mittareiden käyttöönottoon niiden prosessiin liittämisen ja toiminnan mukauttamisen osalta. Mittariston osalta mielenkiintoinen jatkotutkimuksen aihe olisi esimerkiksi noin vuosi mittareiden käyttöönoton jälkeen tehdä uudestaan selvitys tiimiä kiinnostavista mittareista. Ennen tutkimusta tiimillä ei ollut omia mittareita käytössä, joten mittariston käytön jälkeen tiimin tietämys mittareiden toiminnasta varmasti kasvaa ja mahdolliset näkemykset juuri tiimille sopivista mittareista saattaisi sitä myöten muuttua. Myös ohjelmistokehitysprosessien osalta olisi mielenkiintoista käydä tiimin kanssa uudestaan keskusteluja, että ovatko tutkimuksessa kuvatut prosessit vielä valideja esimerkiksi vuosi tutkimuksen jälkeen. Voisi myös selvittää onko tiimille ollut apua tutkimuksessa määritetyistä kaavioista ja onko niitä käytetty arvioimaan tai kehittämään tiimin omaa toimintaa edelleen.

LÄHTEET

Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002). Agile software development methods: Review and analysis, *VTT publication*, 478, Espoo, Finland, 107p.

Agarwal, M. & Majumdar, R. (2012). Tracking scrum projects tools, metrics and myths about agile. *International Journal of Emerging Technology and Advanced Engineering*, 2, 97-104.

Akif, R. & Majeed, H. (2012). Issues and challenges in Scrum implementation. *International Journal of Scientific & Engineering Research*, 3(8), 1-4.

Ashraf, S. & Aftab, S. (2017). IScrum: An improved scrum process model. *International Journal of Modern Education and Computer Science*, 9(8), 16.

Baskerville, R., Baiyere, A., Gregor, S., Hevner, A. & Rossi, M. (2018). Design science research contributions: Finding a balance between artifact and theory. *Journal of the Association for Information Systems*, 19(5), 3.

Bass, L., Weber, I. & Zhu, L. (2015). DevOps: A software architect's perspective. *Addison-Wesley Professional*.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, (10), 70-77.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., ym. (2001). Manifesto for agile software development.

Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2), 50-54.

Chen, L. (2017). Continuous delivery: overcoming adoption challenges. *Journal of Systems and Software*, 128, 72-86.

Chow, T. & Cao, D.B. (2008). A survey study of critical success factors in agile software projects. *Journal of systems and software*, 81(6), 961-971.

Claps, G.G., Svensson, R.B. & Aurum, A. (2015). On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, 57, 21-31.

CollabNet VersionOne. (2019). 13th annual State of Agile report. https://www.stateofagile.com/#ufh-i-521251909-13th-annual-state-of-agile-report/473508 Haettu 4.10.2019 klo 15:00.

Dıaz, J., Pérez, J., Yagüe, A., & Garbajosa, J. (2011). Tailoring the scrum development process to address agile product line engineering. *Proceedings of the XV Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2011),* 91.

Dikert, K., Paasivaara, M. & Lassenius, C. (2016). Challenges and success factors for large-scale agile transformations: A systematic literature review. *Journal of Systems and Software*, 119, 87-108.

Dingsøyr, T., Nerur, S., Balijepally, V. & Moe, N.B. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, 85(6), 1213– 1221.

Downey, S. & Sutherland, J. (2013). Scrum metrics for hyperproductive teams: how they fly like fighter aircraft. *46th Hawaii International Conference on System Sciences*, 4870-4878. IEEE.

Dybå, T. & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9-10), 833-859.

Fenton, N. (1994). Software measurement: A necessary scientific basis. IEEE *Transactions on software engineering*, 20(3), 199-206.

Fenton, N. & Bieman, J. (2014). Software metrics: a rigorous and practical approach. *CRC press*.

Fenton, N. E. & Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3), 149-157.

Fitzgerald, B. & Stol, K.J. (2014). Continuous software engineering and beyond: trends and challenges. *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pp. 1-9. ACM.

Fowler, M. (2006). Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html. Haettu 19.10.2019 klo 14:00.

Gaffney Jr, J. E. (1981). Metrics in software quality assurance. *Proceedings of the ACM'81 conference,* s. 126-130. ACM.

Gill, A.Q. & Henderson-Sellers, B. (2006). Measuring agility and adaptibility of agile methods: A 4 dimensional analytical tool. *The IADIS international conference on applied computing*, 2006. IADIS Press.

Gousios, G., Zaidman, A., Storey, M. A. & Van Deursen, A. (2015). Work practices and challenges in pull-based development: the integrator's perspective. *In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 358-368. IEEE.

Hernández, T.R., Kreye, M. & Eppinger, S. (2019). Applicability of Agile and Scrum to Product-Service Systems. *EurOMA Conference*, 1-10.

Herzig, K., Just, S., & Zeller, A. (2013). It's not a bug, it's a feature: how misclassification impacts bug prediction. *In 2013 35th International Conference on Software Engineering (ICSE)*, 392-401. IEEE.

Hevner, A., & Chatterjee, S. (2010). Design science research in information systems. Theory and Practice. *Integrated Series in Information Systems*, 22, Springer US.

Humble, J. & Farley, D. (2011). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. *Pearson Education*.

Hüttermann, M. (2012). DevOps for developers. *Apress*.

Jabbari, R., bin Ali, N., Petersen, K. & Tanveer, B. (2016). What is DevOps?: A systematic mapping study on definitions and practices. *Proceedings of the Scientific Workshop Proceedings of XP2016*, p. 12. ACM.

Kettunen, P., Laanti, M., Fagerholm, F., Mikkonen, T. & Männistö, T. (2019). Finnish enterprise agile transformations: a survey study. *International Conference on Agile Software Development*, 97-104. Springer, Cham.

Kupiainen, E., Mäntylä, M. V. & Itkonen, J. (2015). Using metrics in Agile and Lean Software Development – A systematic literature review of industrial studies. *Information and Software Technology*, 62, 143-163.

Laukkanen, E., Itkonen, J. & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82, 55-79.

Lehtonen, T., Suonsyrjä, S., Kilamo, T. & Mikkonen, T. (2015). Defining metrics for continuous delivery and deployment pipeline. *SPLST*, 16-30.

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V. & Männistö, T. (2015). The highways and country roads to continuous deployment. *IEEE software*, 32(2), 64-72.

Lwakatare, L.E. (2017). DevOps adoption and implementation in software development practice: concept, practices, benefits and challenges. *Acta Universitatis Ouluensis, A scientiae Rerum Naturalium* 702.

Lwakatare, L.E., Kuvaja, P. & Oivo, M. (2015). Dimensions of devops. *International conference on agile software development*, 212-217. Springer, Cham.

Lwakatare, L.E., Kuvaja, P. & Oivo, M. (2016). An exploratory study of devops extending the dimensions of devops with practices. *ICSEA 2016*, 104.

Maier, P., Ma, Z., & Bloem, R. (2017). Towards a secure scrum process for agile web application development. *In Proceedings of the 12th International Conference on Availability, Reliability and Security*, 1-8.

Marchenko, A. & Abrahamsson, P. (2008). Scrum in a multiproject environment: An ethnographically-inspired case study on the adoption challenges. *In Agile 2008 Conference*, pp. 15-26. IEEE.

McWhirter, K. & Gaughan, T. (2012). The Definitive Guide to IT Service Metrics (Vol. 1). *IT Governance Publishing*.

Medeiros, D.B., Neto, P.D.A.D.S., Passos, E.B. & De Souza Araújo, W. (2015). Working and playing with Scrum. *International Journal of Software Engineering and Knowledge Engineering*, 25(06), 993-1015.

Mezak, S. (2018). The Origins of DevOps: What's in a Name? https://devops.com/the-origins-of-devops-whats-in-a-name/. Haettu 5.4.2019 klo 16:00.

Mills, E. E. (1988). Software Metrics. SEI Curriculum Module SEICM-12-1.1 Seattle University. *Software Engineering Institute, Carnegie Mellon University*.

Misra, S. & Omorodion, M. (2011). Survey on agile metrics and their inter-relationship with other traditional development metrics. *ACM SIGSOFT Software Engineering Notes*, 36(6), 1-3.

Nerur, S., Mahapatra, R. & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5), 72-78.

Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E. & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128, 164-197.

Olsson, H.H., Alahyari, H. & Bosch, J. (2012). Climbing the "Stairway to Heaven"--A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. *2012 38th euromicro conference on software engineering and advanced applications*, pp. 392-399. IEEE.

Ordonez, M. J. & Haddad, H. M. (2008). The state of metrics in software industry. *Fifth International Conference on Information Technology: New Generations*, s. 453-458. IEEE.

Padmini, K.J., Bandara, H.D. & Perera, I. (2015). Use of software metrics in agile software development process. *Moratuwa Engineering Research Conference (MERCon)*, 312-317. IEEE.

Peffers, K., Tuunanen, T., Rothenberger, M.A. & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, 24(3), 45-77.

Pries-Heje, L. & Pries-Heje, J. (2011). Why Scrum works: A case study from an agile distributed project in Denmark and India. *In 2011 Agile Conference*, pp. 20-28. IEEE.

Qumer, A. & Henderson-Sellers, B. (2008). An evaluation of the degree of agility in six agile methods and its applicability for method engineering. *Information and software technology*, 50(4), 280-295.

Roche, J. (2013). Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11), 38-43.

Rodríguez, P., Markkula, J., Oivo, M. & Turula, K. (2012). Survey on agile and lean usage in finnish software industry. *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 139-148. IEEE.

Saff, D. & Ernst, M.D. (2003). Reducing wasted development time via continuous testing. *14th International Symposium on Software Reliability Engineering*, pp. 281-292. IEEE.

Schwaber, Ken. (1997). Scrum development process. Business object design and implementation. *Springer, London*, 117-134.

Schwaber, K. & Beedle, M. (2002). Agile software development with Scrum (Vol. 1). *Upper Saddle River: Prentice Hall*.

Schwaber, K. & Sutherland, J. (2017). Scrum-opas. Scrumin määritelmä ja pelisäännöt. *Scrum Org*.

Serrador, P. & Pinto, J.K. (2015). Does Agile work? A quantitative analysis of agile project success. *International Journal of Project Management*, 33(5), 1040-1051.

Shahin, M., Babar, M.A. & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5, 3909-3943.

Stoica, M., Mircea, M. & Ghilic-Micu, B. (2013). Software Development: Agile vs. Traditional. *Informatica Economica*, 17(4).

Streule, T., Miserini, N., Bartlomé, O., Klippel, M. & De Soto, B.G. (2016). Implementation of scrum in the construction industry. *Procedia engineering*, 164, 269-276.

Stålhane, T., Myklebust, T. & Hanssen, G. K. (2012). The application of Safe Scrum to IEC 61508 certifiable software. *In 11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference*, 6052-6061.

Szalvay, V. (2004). An introduction to agile software development. *Danube technologies*, 3.

Tahir, T., Rasool, G. & Gencel, C. (2016). A systematic literature review on software measurement programs. *Information and Software Technology*, 73, 101-121.

Takeuchi, H. & Nonaka, I. (1986). The new new product development game. *Harvard business review*, 64(1), 137-146.

Tessem, B. & Iden, J. (2008). Cooperation between developers and operations in software engineering projects. *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, 105-108. ACM.

Vaishnavi, V., Kuechler, W. and Petter, S. (2004). Design Science Research in Information Systems. Luotu 2004 and muokattu 2015 asti Vaishnavi, V. and Kuechler, W. toimesta. Muokattu viimeksi Vaishnavi, V. and Petter, S. 2019. URL: http://www.desrist.org/design-research-in-information-systems/.

Vijayasarathy, L.E.O.R. & Turk, D. (2008). Agile software development: A survey of early adopters. *Journal of Information Technology Management*, 19(2), 1-8.

Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pp. 78-82. IEEE.

Yu, Y., Wang, H., Filkov, V., Devanbu, P. & Vasilescu, B. (2015). Wait for it: Determinants of pull request evaluation latency on github. *In 2015 IEEE/ACM 12th working conference on mining software repositories*, 367-371. IEEE.

Tutkimuksessa kerättyjen mittareiden lähteet

Agarwal, M. & Majumdar, R. (2012). Tracking scrum projects tools, metrics and myths about agile. *International Journal of Emerging Technology and Advanced Engineering*, 2(3), 97-104.

Barton, B., Schwaber, K. & Rawsthorne, D. (2007). Reporting Scrum project progress to executive management through metrics. *The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process*, 101-108.

Dobran, B. (2019). 15 DevOps Metrics & KPIs That Enterprises Should Be Tracking. https://phoenixnap.com/blog/devops-metrics-kpis Haettu 26.4.2019 klo 16:20.

Downey, S., & Sutherland, J. (2013). Scrum metrics for hyperproductive teams: how they fly like fighter aircraft. *In 2013 46th hawaii international conference on system sciences*, 4870-4878. IEEE.

Elliot, S. (2014). DevOps and the cost of downtime: Fortune 1000 best practice metrics quantified. *International Data Corporation* (IDC).

Jureczko, M. & Madeyski, L. (2011) Software product metrics used to build defect prediction models.

Kupiainen, E., Mäntylä, M. V. & Itkonen, J. (2015). Using metrics in Agile and Lean Software Development – A systematic literature review of industrial studies. *Information and Software Technology*, 62, 143-163.

Lehtonen, T., Suonsyrjä, S., Kilamo, T. & Mikkonen, T. (2015). Defining metrics for continuous delivery and deployment pipeline. *SPLST*, 16-30.

Madeyski, L. & Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3), 393-422.

Mahnic, V. & Vrana, I. (2007). Using stakeholder driven process performance measurement for monitoring the performance of a Scrum based software development process. *Electrotechnical Review*, 74(5), 241-247.

Mahnic, V. & Zabkar, N. (2008). Using COBIT indicators for measuring scrum-based software development. *Wseas transactions on computers*, 7(10), 1605-1617.

Medeiros, D. B., Neto, P.D.A.D.S., Passos, E. B. & De Souza Araújo, W. (2015). Working and playing with Scrum. *International Journal of Software Engineering and Knowledge Engineering*, 25(06), 993-1015.

Misra, S. & Omorodion, M. (2011). Survey on agile metrics and their inter-relationship with other traditional development metrics. *ACM SIGSOFT Software Engineering Notes*, 36(6), 1-3.

Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128, 164-197.

Olague, H. M., Etzkorn, L. H., Messimer, S. L. & Delugach, H. S. (2008). An empirical validation of object - oriented class complexity metrics and their ability to predict error - prone classes in highly iterative, or agile, software: a case study. Journal of software maintenance and evolution: *Research and practice*, 20(3), 171-197.

Padmanabhan, A. & Rawat, L. DevOps Metrics. https://devopedia.org/devops-metrics Haettu 26.4.2019 klo 16:00.

Padmini, K.J., Bandara, H.D. & Perera, I. (2015). Use of software metrics in agile software development process. *Moratuwa Engineering Research Conference (MERCon),* 312-317. IEEE.

Perkusich, M., Gorgônio, K. C., Almeida, H. & Perkusich, A. (2017). Assisting the continuous improvement of Scrum projects using metrics and Bayesian networks. *Journal of Software: Evolution and Process*, 29(6), e1835.

Shatnawi, R. (2015). Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process*, 27(2), 95-113.

Waller, J., Ehmke, N. C. & Hasselbring, W. (2015). Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes*, 40(2), 1-4.

Watson, M. (2017). 15 Metrics for DevOps Success. https://stackify.com/15-metrics-for-devops-success/ Haettu 26.4.2019 klo 16:00.

LIITE 1 MITTARITAULUKKOJEN ITERAATIOVERSIOITA

TAULUKKO 1 Kirjallisuudesta listatut mittarit osa-alueittain

Osa-alue	Mittarit	Lähde
Scrum	1) Test code coverage, 2) Team motivation	Perkusich ym., 2017
	3) Product burndown chart, 4) Work breakdown structure chart, 5) Parking lot chart	Barton ym., 2007
	6) Velocity, 7) Work Capacity, 8) Focus Factor, 9) Percentage of Adopted Work, 10) Percentage of Found Work, 11) Accuracy of Estimation, 12) Accuracy of Forecast, 13) Targeted Value Increase (TVI+), 14) Success at Scale, 15) Win/Loss Record	Downey & Sutherland, 2013
	16) Burn down charts, 17) Progress chart, 18) Velocity, 19) Standard violation, 20) Business value delivered, 21) Defects per iteration, 22) Number of stories, 23) Level of automation, 24) Number of tests	Agarwal & Majumdar, 2012
	25) Product backlog, 26) Product burndown chart, 27) Sprint backlog, 28) Sprint burndown chart	Mahnic & Zabkar, 2008
	29) Work spent on day i for each task j in the Sprint Backlog, 30) Work remaining on day i for each task j in the Sprint Backlog, 31) Work remaining on day i for each task j in the Sprint Backlog, 32) Work spent on day i for each task j in the Sprint Backlog, 33) The length of the Sprint (number of working days in the Sprint), 34) The number of days elapsed from the beginning of the Sprint, 35) Work remaining on day i for each task j in the Sprint Backlog, 36) Work spent on day i for each task j in the Sprint Backlog, 37) Cost of Team member's engineering hour (for each task j in the Sprint Backlog), 38) The length of the Sprint (number of working days in the Sprint), 39) The number of days elapsed from the beginning of the Sprint, 40) The number of errors found during the Sprint review meeting (for each PBI), 41) The number of errors reported by the user in a fixed period after release (for each PBI), 42) The size of the code (for each PBI separately), 43) Work spent on day i for each task j in the Sprint Backlog referring to rework, 44) Cost of Team member's engineering hour, 45) Total number of PBIs in the release/Sprint, 46) The number of PBIs completed in the release/Sprint, 47) Total number of tasks in the Sprint, 48) The number of tasks completed during the Sprint, 49) Work remaining on day i for each task j in the Sprint Backlog, 50) Work spent on day i for each task j in the Sprint Backlog, 51) The length of the Sprint (number of working days in the Sprint), 52) Percentage of Team member's engagement in the project, 53) The number of administrative days, 54) The number of Team members (the size of Team t), 55) The number of Team members (the size of Team t), 56) The total number of developers, 57) Surveys: evaluate working conditions or	Mahnic & Vrana, 2007

TAULUKKO 1 jatkuu

Osa-alue	Mittarit	Lähde
	customer satisfaction per sprint/release	
	58) Number of tasks completed, 59) Average time to complete a task (estimated and actual), 60) Duration of each sprint (planned and actual), 61) Participation in Daily Scrum, 62) Number of items approved or disapproved in the Review meeting, 63) Super Scrum Master (SSM), 64) PO Presence (POPr), 65) PO Partner (POPa), 66) Sprint Backlog Completion (SBC), 67) Sprint Review Acceptance (SRA), 68) Clockwork Developer (CD), 69) Clockwork Team (CT), 70) Daily Scrum Developer Presence (DSDP), 71) Daily Scrum Team Realization (DSTR), 72) Sprint Latency (SL)	Medeiros ym., 2015
DevOps	73) Git branch build, 74) Log analytics, 75) Average time to repair infrastucture failure, 76) Average time to repair application failure, 77) Average time to restore production failure, 78) Timeline for code change impact on customers (from commit to production), 79) Average time spent per application on unplanned work	Elliot, 2014
	80) Performance benchmarking continuous integration code	Waller ym., 2015
	81) Deployment frequency, 82) Change volume, 83) Deployment time¸ 84) Lead time, 85) Customer tickets, 86) Automated test pass %, 87) Defect escape rate, 88) Availability, 89) Service level agreements, 90) Failed deployments¸ 91) Error rates, 92) Application usage and traffic, 93) Application performance, 94) Mean time to detection (MTTD), 95) Mean time to recovery (MTTR)	Watson, 2017
	96) Lead time, 97) change complexity¸ 98) deployment frequency, 99) MTTR, 100) deployment success rate, 101) application error rate, 102) escaped defects¸ 103) number of support tickets, 104) automated test pass percentage¸ 105) availability, 106) scalability, 107) latency, 108) resource utilization¸ 109) usability, 110) defect age, 111) subscription renewals¸ 112) feature usage¸ 113) business impact, 114) application usage and traffic, 115) Mean Time To Detect (MTTD), 116) Mean Time To Failure (MTTF), 117) Mean Time Between Failures (MTBF), 118) Mean Time To Repair (MTTR)	Padmanabhan & Rawat, 2019
	119) Unplanned work, 120) Cycle time	Dobran, 2019
Agile	121) Duration of testing, 122) Number of defects discovered during testing process, 123) Time a programmer spends on a project, 124) Programmer productivity, 125) Requirements stability, 126) Effort expended on tasks, 127) Extra-project activities, 128) Elapsed time, 129) Computer resources, 130) Planned effort¸ 131) Actual effort	Misra & Omorodion, 2011
	132) Delivery on time¸ 133) Work capacity, 134) Unit test coverage for the developed code¸ 135) Percentage of adopted work, 136) Bug correction time from new-to-closed state¸ 137) Sprint-level effort burndown¸ 138) Velocity, 139) Percentage of found work¸ 140) Open defect	Padmini ym., 2015

TAULUKKO 1 jatkuu

Osa-alue	Mittarit	Lähde
	severity index, 141) Focus factor, 142) Cost of quality, 143) Defect severity index, 144) Technical debt¸ 145) Defect slippage rate, 146) Customer satisfaction survey, 147) Accuracy of estimation, 148) Accuracy of forecast, 149) Requirements clarity index, 150) Defect density, 151) Defect removal efficiency, 152) Targeted value increase	
	153) Business value delivered, 154) Number of test cases, 155) Velocity, 156) Work in progress, 157) Critical defects sent by customer, 158) Open defects¸ 159) Test failure rate¸ 160) Test success rate, 161) Remaining task effort¸ 162) Technical debt board, 163) Technical debt in effort, 164) Build status, 165) Burndown, 166) Check-ins per day¸ 167) Number of automated passing test steps, 168) Faults per iteration, 169) Story estimates, 170) Task's expected end date, 171) Effort estimate, 172) Tasks done, 173) Fix time of failed build, 174) Percentage of stories prepared for sprint, 175) Cycle time, 176) Lead time, 177) Processing time, 178) Queue time, 179) Change requests per requirement, 180) Implemented vs wasted requirements, 181) Maintenance effort, 182) Story complete percentage	Kupiainen ym., 2015
Automation	183) Development time, 184) Deployment time, 185) Activation time, 186) Oldes done feature (ODF), 187) Features per month (FPM), 188) Releases Per Month (RPM), 189) Fastest Possible Feature Lead Time	Lehtonen ym., 2015
Code/ software	190) McCabe cyclomatic complexity, 191) Weighted methods per class (WMC), 192) Weighted methods per class McCabe (WMC McCabe), 193) Standard deviation method complexity (SDMC), 194) Average method complexity (AMC), 195) Maximum cyclomatic complexity of a single method of a class (CC Max), 196) Number of instance methods (NIM), 197) Number of trivial methods (NTM), 198) Average lines of code, 199) Lines of code per class	Olague ym., 2008
	200) Coupling between objects (CBO), 201) Depth of Inheritance Hierarchy (DIT), 202) Number of Child Classes (NOC), 203) Lack of Cohesion of Methods (LCOM), 204) Response for Class (RFC), 205) Weighted Methods Complexity (WMC)	Shatnawi, 2015
	206) Number of Attributes (NOA), 207) Number of Public Methods (NOPM), 208) Lines of Comments (LCOMM), 209) Lines of Source Code (SLOC)	Nuñez-Varela ym., 2017
	210) Data Access Metric (DAM), 211) Measure of Aggregation (MOA), 212) Measure of Functional Abstraction (MFA), 213) Cohesion Among Methods of Class (CAM), 214) Inheritance Coupling (IC)¸ 215) Coupling Between Methods (CBM), 216) Average Method Complexity (AMC), 217) Afferent couplings (Ca)¸ 218) Efferent couplings (Ce)	Jureczko & Madeyski, 2011
	219) Size of final program, 220) Development time, 221) Type of methodology used, 222) Length of source code	Misra & Omorodion, 2011
	223) Number of Revisions (NR), 224) Number of Distinct	Madeyski & Jureczko,

2015

TAULUKKO 1 jatkuu

Osa-alue	Mittarit	Lähde
	Committers (NDC), 225) Number of Modified Lines (NML), 226) Number of Defects in Previous Version (NDPV)	

TAULUKKO 2 Kirjallisuudesta listatut mittarit kategorisoidusti

Kategoria	**Mittari**
Sprintti	1) Percentage of stories prepared for sprint, 2) Sprint backlog, 3) The number of days elapsed from the beginning of the Sprint, 4) Total number of PBIs in the release/Sprint, 5) The number of PBIs completed in the release/Sprint, 6) Duration of each sprint (planned and actual), 7) Sprint Backlog Completion (SBC), 8) Sprint Review Acceptance (SRA), 9) Sprint Latency (SL)
Feature	10) Oldest done feature (ODF), 11) Features per month/other time period, 12) Releases Per Month (RPM), 13) Feature usage
Backlogitemit	14) Product backlog, 15) Number of stories, 16) Story estimates, 17) Story complete percentage, 18) Effort expended on tasks, 19) Remaining task effort, 20) Task's expected end date, 21) Tasks done, 22) Work spent on day i for each task j in the Sprint Backlog, 23) Work remaining on day i for each task j in the Sprint Backlog, 24) Work spent on day i for each task j in the Sprint Backlog, 25) Cost of Team member's engineering hour (for each task j in the Sprint Backlog), 26) Work spent on day i for each task j in the Sprint Backlog referring to rework, 27) Average time to complete a task (estimated and actual), 28) Number of items approved or disapproved in the Review meeting, 29) Work in progress
Virheet/bugit	30) Defect severity index, 31) Defect slippage rate, 32) Defect density, 33) Defect removal efficiency, 34) Number of defects discovered during testing process, 35) Open defects, 36) Defect escape rate, 37) Escaped defects, 38) Defect age, 39) Defects per iteration, 40) Open defect severity index, 41) The number of errors found during the Sprint review meeting (for each PBI), 42) The number of errors reported by the user in a fixed period after release (for each PBI), 43) Average time to repair infrastucture failure, 44) Average time to repair application failure, 45) Average time to restore production failure, 46) Bug correction time from new-to-closed state, 47) Faults per iteration, 48) Critical defects sent by customer, 49) Mean Time To Detect (MTTD), 50) Mean Time To Failure (MTTF), 51) Mean Time Between Failures (MTBF), 52) Mean Time To Repair/Recovery (MTTR)
Testaus	53) Duration of testing, 54) Number of test cases, 55) Number of tests, 56) Number of passed test cases, 57) Number of passed tests
Kaaviot	58) Product (project) burndown chart, 59) Work breakdown structure chart, 60) Parking lot chart, 61) Progress chart, 62) Sprint burndown chart
Ajanhallinta	63) Delivery on time, 64) Cycle time, 65) Lead time, 66) Processing time, 67) Queue time, 68) Elapsed time, 69) Fix time of failed build, 70) Timeline for code change impact on customers (from commit to production)
Tiimi	71) Cost of Team member's engineering hour, 72) Percentage of Team member's engagement in the project, 73) The number of administrative days, 74) The number of Team members (the size of Team t), 75) The total number of developers, 76) Time a programmer spends on a project, 77) Programmer productivity, 78) Extra-project activities, 79) Planned effort, 80) Actual effort, 81) Work capacity, 82) Unplanned work, 83) Team motivation, 84) Percentage of Adopted Work, 85) Percentage of Found Work, 86) Super Scrum Master (SSM), 87) PO Presence (POPr), 88) PO Partner (POPa), 89) Clockwork Developer (CD),

TAULUKKO 2 jatkuu

Kategoria	Mittari
	90) Clockwork Team (CT), 91) Daily Scrum Developer Presence (DSDP), 92) Daily Scrum Team Realization (DSTR), 93) Participation in Daily Scrum, 94) Cost of development hour (feature)
Koodianalytiikka	95) Size of final program, 96) The size of the code (for each PBI separately), 97) Length of source code, 98) Average lines of code, 99) Lines of code per class, 100) Lines of Comments (LCOMM), 101) Lines of Source Code (SLOC), 102) Number of Revisions (NR), 103) Number of Distinct Committers (NDC), 104) Number of Modified Lines (NML), 105) Number of Defects in Previous Version (NDPV), 106) Number of Attributes (NOA), 107) Number of Public Methods (NOPM), 108) Type of methodology used, 109) Git branch build, 110) Failed deployments, 111) Application error rate, 112) Test code coverage, 113) McCabe cyclomatic complexity, 114) Weighted methods per class (WMC), 115) Weighted methods per class McCabe (WMC McCabe), 116) Standard deviation method complexity (SDMC), 117) Average method complexity (AMC), 118) Maximum cyclomatic complexity of a single method of a class (CC Max), 119) Number of instance methods (NIM), 120) Number of trivial methods (NTM), 121) Coupling between objects (CBO), 122) Depth of Inheritance Hierarchy (DIT), 123) Number of Child Classes (NOC), 124) Lack of Cohesion of Methods (LCOM), 125) Response for Class (RFC), 126) Weighted Methods Complexity (WMC), 127) Data Access Metric (DAM), 128) Measure of Aggregation (MOA), 129) Measure of Functional Abstraction (MFA), 130) Cohesion Among Methods of Class (CAM), 131) Inheritance Coupling (IC), 132) Coupling Between Methods (CBM), 133) Average Method Complexity (AMC), 134) Afferent couplings (Ca), 135) Efferent couplings (Ce)
Automatiikka	136) Performance benchmarking continuous integration code, 137) Deployment success rate, 138) Automated test pass percentage, 139) Development time, 140) Deployment time, 141) Activation time (first user use of feature), 142) Fastest Possible Feature build and test phase Lead Time, 143) Level of automation, 144) Test failure rate, 145) Test success rate, 146) Number of automated passing test steps, 147) Build status
Muut	148) Computer resources, 149) Technical debt, 150) Velocity, 151) Customer satisfaction survey, 152) Requirements stability, 153) Requirements clarity index, 154) Focus factor, 155) Cost of quality, 156) Effort estimate, 157) Check-ins per day, 158) Change requests per requirement, 159) Implemented vs wasted requirements, 160) Maintenance effort, 161) Deployment frequency, 162) Change volume, 163) Customer tickets, 164) Availability, 165) Service level agreements, 166) Application usage and traffic, 167) Application performance, 168) Change complexity, 169) Deployment frequency, 170) Scalability, 171) Latency, 172) Resource utilization, 173) Usability, 174) Application usage and traffic, 175) Accuracy of Estimation, 176) Accuracy of Forecast, 177) Targeted Value Increase (TVI+), 178) Success at Scale, 179) Win/Loss Record, 180) Evaluate working conditions, 181) Standard violation, 182) Business value delivered

TAULUKKO 3 Seminaarikeskustelussa jatkoon valitut mittarit kategorioittain ja tyypeittäin

Kategoria	Tyyppi	Mittari
Sprintti	Kirjallisuus	1) Total number of PBIs in the release/Sprint, 2) The number of PBIs completed in the release/Sprint, 3) Sprint Backlog Completion (SBC), 4) Sprint Review Acceptance (SRA)
	Keskustelu	5) Siirtyvien itemien määrä sprintistä toiseen
Feature	Kirjallisuus	6) Oldest done feature (ODF), 7) Features per month/muu ajanjakso (FPM), 8) Releases Per Month (RPM), 9) Feature usage
	Keskustelu	10) Featuren liiketoimintahyöty
Backlogitemit	Kirjallisuus	11) Number of stories, 12) Story complete percentage, 13) Effort expended on tasks, 14) Remaining task effort, 15) Tasks done, 16) Average time to complete a task (estimated and actual), 17) Number of items approved or disapproved in the Review meeting, 18) Work in progress
	Keskustelu	19) Storyn vaiheiden etenemisen aikamääreet, 20) Kuinka paljon siirretään storyjä Removed-tilaan, 21) Storyjen määrä, 22) Kuinka paljon siirretään storyjä Määrittely-tilaan
Virheet/bugit	Kirjallisuus	23) Number of defects discovered during testing process, 24) Open defects, 25) Defect age, 26) Defects per iteration, 27) The number of errors found during the Sprint review meeting (for each PBI), 28) Bug correction time from new-to-closed state
	Keskustelu	29) Virheet kehityksessä/testauksessa vs. vuoden käytön aikana havaitut, 30) Bugien kokonaismäärä backlogilla
Testaus	Kirjallisuus	31) Number of test cases, 32) Number of tests, 33) Number of passed test cases, 34) Number of passed tests
	Keskustelu	35) Releasen hyväksymistestauksen kesto
Kaaviot	Kirjallisuus	36) Product (project) burndown chart, 37) Work breakdown structure chart, 38) Progress chart, 39) Sprint burndown chart
	Keskustelu	40) Henkilöresurssikaavio (tunteja per henkilö esim. per sprintti)
Ajanhallinta	Kirjallisuus	41) Fix time of failed build
	Keskustelu	42) Releasen lead time
Tiimi	Kirjallisuus	43) Planned effort, 44) Actual effort, 45) Work capacity, 46) Unplanned work, 47) Cost of development hour (feature)
Koodianalytiikka	Kirjallisuus	48) Lines of Comments (LCOMM), 49) Lines of Source Code (SLOC), 50) Number of Revisions (NR), 51) Test code coverage, 52) McCabe cyclomatic complexity
	Keskustelu	53) Kuinka paljon deprekoitunutta koodia, 54) Kuinka paljon toistoa koodissa
Automatiikka	Kirjallisuus	55) Deployment success rate, 56) Automated test pass percentage, 57) Development time, 58) Deployment time, 59) Test failure rate, 60) Test success rate, 61) Build status

TAULUKKO 3 jatkuu

Kategoria	Tyyppi	Mittari
	Keskustelu	62) Uuden (featuren) ominaisuuden käyttö(aste), 63) Build määrä featuressa, 64) Build time
Muut	Kirjallisuus	65) Effort estimate, 66) Check-ins per day, 67) Maintenance effort, 68) Deployment frequency, 69) Customer tickets, 70) Resource utilization, 71) Accuracy of Estimation
	Keskustelu	72) Business value delivered