Tampereen yliopisto

Joonas Salmi

VAATIMUSMÄÄRITTELY KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

Kandidaatintyö
Tieto- ja sähkötekniikan tiedekunta
Outi Sievi-Korte
Toukokuu 2020

TIIVISTELMÄ

Joonas Salmi: Vaatimusmäärittely ketterässä ohjelmistokehityksessä
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan kandidaatin tutkinto-ohjelma
Toukokuu 2020

Tämä kandidaatintyö on kirjallisuuskatsaus ketterän ohjelmistokehityksen vaatimusmäärittelyn eri metodeista. Työllä pyritään selvittämään ketterän vaatimusmäärittelyn heikkoudet ja mitkä seikat vaikuttavat ketterän vaatimusmäärittelyn onnistumiseen. Kandidaatintyössä lähteinä on käytetty pääasiassa kirjallisuuskatsauksia sekä tieteellisiä julkaisuja.

Ohjelmistokehitysmallit jaetaan perinteiseen ja ketterään ohjelmistokehitysmalliin. Perinteisessä ohjelmistokehityksessä pääpaino on kattavassa suunnittelussa ja vaatimusmäärittelyssä ennen varsinaisen ohjelmistokehityksen aloittamista [1]. Perinteisistä malleista työssä esitellään vesiputousmalli, v-malli ja spiraalimalli. Ketterissä malleissa pyritään saamaan tuotettua asiakkaille mahdollisimman nopeasti tärkeimmät vaatimukset täyttävä toimiva järjestelmä [2]. Ketterillä malleilla kattavalla dokumentaatiolla ei ole samanlaista painoarvoa, kuin perinteisillä malleilla [3]. Ketteristä malleista työssä esitellään scrum. Vaatimusmäärittely käydään aluksi läpi perinteisen ohjelmistokehityksen näkökulmasta.

Lopuksi työssä tarkastellaan ketterän vaatimusmäärittelyn metodeja, joita ovat muun muassa vaatimusten priorisointi, kommunikointi, käyttötapaukset, käyttäjätarinat, prototyypit. Vaatimusten priorisoinnissa poimitaan tärkeimmät vaatimukset kehitysjonon kärkeen [4]. Kommunikaatio on avainasemassa ketterissä menetelmissä, koska se muovaa asiakkaan vaatimuksia projektin edetessä ja näin vaatimuksetkin voivat kehittyä [3]. Käyttötapauksilla voidaan kuvata ohjelmiston eri ominaisuuksia eri tarkkuustasoilla [5]. Käyttäjätarinat kuvaavat ominaisuuksia, jotka antavat arvoa asiakkaalle [4]. Käyttötapauksilla ja käyttäjätarinoilla voidaan kuvata ohjelmiston eri vaatimukset [4, 5]. Prototyypit ovat yksi tehokkaimmista ja nopeimmista tavoista auttaa asiakasta visualisoimaan vaatimukset [4].

Ketterän ohjelmistokehityksen vaatimusmäärittelyyn liittyy monia haasteita, jotka perustuvat pääosin mallin nopeatempoiseen muuntuvuuteen. Ketterään vaatimusmäärittelyyn liittyy vähäinen dokumentointi ja tämä voi tietyissä tilanteissa aiheuttaa merkittäviä tiedonkulun aukkoja [3]. Dokumentaation heikkoudet ovat havaittavissa erityisesti suurissa ja keskisuurissa projekteissa [6]. Liiketoiminnan näkökulmasta rajoittaviksi tekijöiksi muodostuvat aika, kasvavat kulut ja asiakasedustajaan kohdistuva työmäärä [3]. Myös budjetin ja aikataulun arvioiminen voi olla haasteellista [4]. Koska vaatimusmäärittely elää koko projektin ajan, vaatimukset voivat projektin edetessä kasvaa poikkeuksellisen suuriksi [3]. On myös mahdollista, että ketterien mallien ominaisuuksista hyödynnetään vain niitä, jotka nopeuttavat ohjelmistokehitystä, mutta samaan aikaan jätetään huomioimatta ne menetelmät, jotka parantavat ohjelmiston laatua [7].

Ketterän ohjelmistokehitysmallin hyödyt ovat havaittu etenkin pienissä projekteissa. Keskisuurissa ja suurissa projekteissa dokumentaation heikkous luo omat haasteensa ja näin ollen hyöty voi jäädä saavuttamatta. [6] Ketterää ohjelmistokehitystä on kuitenkin käytetty onnistuneesti myös suurissakin projekteissa [8].

Avainsanat: vaatimusmäärittely, agile, scrum

ALKUSANAT

Tämä Kandidaatintyö on tehty Tampereen yliopistolle (entinen Tampereen teknillinen yliopisto) Informaatioteknologian ja viestinnän tiedekunnalle, joka on ollut valtavana tukena kandidaatintyön tekemisessä. Idea kandidaatintyöhön lähti ketterien menetelmien yleisesti vähäisestä dokumentaatiosta ja perinteisistä menetelmistä eroavista vaatimusmäärittelyn prosesseista. Kandidaatintyön kirjoittaminen on ollut pitkä prosessi. Onneksi vaimo on kannustanut jatkamaan työn parissa ja sen ansiosta tämä projekti on nyt pystytty saattamaan päätökseen.

Hämeenlinnassa, 4.5.2020

Joonas Salmi

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. TEOREETTINEN TAUSTA 2
    2.1 Perinteisen ohjelmistokehityksen mallit 2
    2.2 Ketterä ohjelmistokehitys 5
    2.3 Vaatimusmäärittely 7
3. KETTERÄ VAATIMUSMÄÄRITTELY 11
    3.1 Vaatimusmäärittelyn menetelmät ketterässä ohjelmistokehityksessä 11
    3.2 Vaatimusmäärittelyn haasteet ketterässä ohjelmistokehityksessä 13
4. YHTEENVETO 15
LÄHTEET 16

LYHENTEET JA MERKINNÄT

JAD	Joint Application Development – menetelmä, jossa yrityksen tuotekehitystiimi, johto sekä käyttäjät työskentelevät yhdessä järjestelmän suunnittelemiseksi.
Wiki	Verkkosivu, joka mahdollistaa käyttäjille sen sisällön ja rakenteen muokkaamisen yhteistyössä.

1. JOHDANTO

Vaatimusmäärittely on jatkuva osa ohjelmistokehitystä. Ketterässä ohjelmistokehityksessä ei ole tarkoitus tehdä heti lopullista vaatimusmäärittelyä, vaan vaatimuksia tarkennetaan lisää projektin edetessä. Kandidaatintyössä pyrin selvittämään, mitkä asiat vaikuttavat siihen ja miten. Miten esimerkiksi käyttäjämäärän kasvu tai projektin jäsenten määrä vaikuttavat vaatimusmäärittelyyn. Onko olemassa jo jokin valmis malli, joka tällä hetkellä toimii tarpeeksi hyvin ja onko siinä jotain, joka kannattaisi tehdä eri tavalla projektin tietyssä vaiheessa?

Aluksi työssä perehdytään perinteisiin ja ketteriin ohjelmistokehityksen malleihin, mikä luo pohjan vaatimusmäärittelyn yksityiskohtaisempaan tarkasteluun. Perinteisen ohjelmistokehityksen malleista olen keskittynyt vesiputousmallin esittämiseen. Lisäksi perinteisistä malleista on esitelty lyhyesti v-malli ja spiraalimalli. Ketterässä ohjelmistokehityksen malleista olen valinnut scrumin, jota kandidaatintyössä käsittelen. Ketterän ohjelmistokehityksen malleja on monia muitakin, mutta scrumin toimintamallin iteratiivisuus tekee siitä hyvän esimerkin.

Ohjelmistokehitysmallien esittelyn jälkeen paneudun vaatimusmäärittelyn yleisiin periaatteisiin ja tavoitteisiin päätyen lopulta selvittämään ketterän vaatimusmäärittelyn ominaisuuksia. Lopussa käsittelen ketterän ohjelmistokehityksen vaatimusmäärittelyn haasteita. Kandidaatintyöni toimii yleisenä katsauksena vaatimusmäärittelyn toteutukseen ja ongelmiin.

Kandidaatintyö on toteutettu kirjallisuuskatsauksena, joskin lähdemateriaali jää hieman suppeaksi. Lähteet ovat kuitenkin luotettavia alan julkaisuja. Osa lähteistä on 2000-luvun alussa kirjoitettuja, mutta näiden käyttö on perusteltua, koska niiden sisältämä tieto esimerkiksi vesiputousmallista ei ole olennaisesti muuttunut.

2. TEOREETTINEN TAUSTA

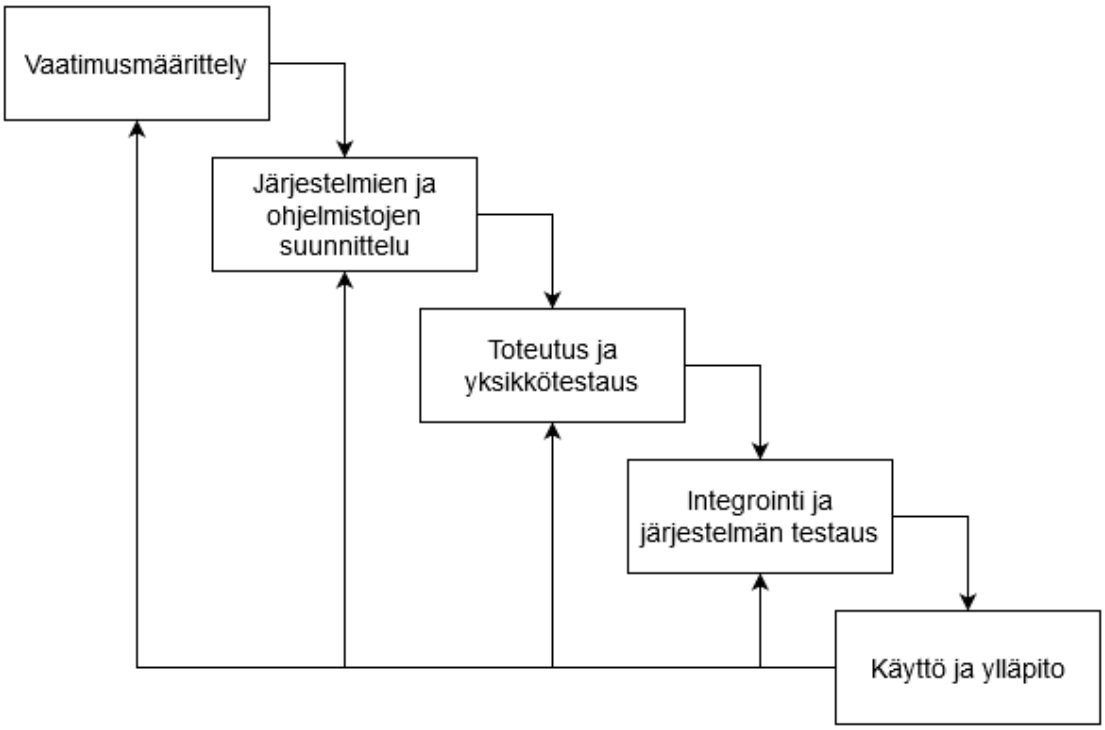
Ohjelmistokehityksen elinkaarimallit voi jakaa karkeasti kahteen eri tyyliin; perinteiseen ja ketterään ohjelmistokehitykseen. Perinteisen ohjelmistokehityksen malleja ovat esimerkiksi vesiputousmalli, spiraalimalli ja v-malli. Perinteisen ohjelmistokehityksen malleissa eri vaiheet suoritetaan järjestyksessä ja loppuun asti ennen seuraavan vaiheen aloittamista. Vesiputousmalli on vanhin ja parhaiten tunnettu ohjelmistokehitysmalli [9]. Perinteisissä keinoissa ohjelmiston dokumentaatio on tärkeässä osassa ja se pyritään saamaan valmiiksi ennen ohjelmakoodin kirjoittamista.

Ketterän ohjelmistokehityksen metodeilla dokumentaatio ei ole niin tärkeässä roolissa. Dokumentaation virkaa hoitaa lähdekoodi ja käyttäjätarinat. Ketterissä metodeissa pyritään aloittamaan ohjelmiston lähdekoodin kirjoittaminen mahdollisimman aikaisin. Ketterillä metodeilla ylläpidetään valmiutta vastata nopeasti asiakkaan muuttuviin toiveisiin ja vaatimuksiin. [3] Vaikka ketterien ohjelmistokehitysmetodien suosio on kasvanut viimevuosien aikana, perinteiset metodit ovat yhä suosittuja. Perinteisten ja ketterien metodien rinnalle on muodostunut niin kutsuttu hybridimalli, joka soveltaa eri metodeja. [10]

2.1 Perinteisen ohjelmistokehityksen mallit

Perinteinen ohjelmistokehitys vaatii aina projektin alussa kattavan dokumentaation ohjelmiston vaatimuksista. Perinteinen ohjelmistokehitys antaa suuren painoarvon vaatimusmäärittelylle ja niiden dokumentoinnille. Perinteisessä ohjelmistokehityksessä uskotaan olevan mahdollista kerätä täysin kattavat vaatimukset jo projektin alussa. [1] Erityisesti vesiputousmallissa on tärkeää, että vaatimukset ovat kerätty kattavasti ja että niistä ollaan yhtä mieltä ennen seuraavaan suunnitteluvaiheeseen siirtymistä, koska jokainen muutos vaatimuksiin tai uusi vaatimus tarkoittaa palaamista takaisin vaatimusmäärittelyvaiheeseen [11].

Vesiputousmalli on perinteinen ohjelmistokehityksen lähestymistapa. Vesiputousmallin vaiheet ovat järjestyksessä: vaatimusmäärittely, järjestelmien ja ohjelmistojen suunnittelu, toteutus ja yksikkötestaus, integrointi ja järjestelmän testaus sekä käyttö ja ylläpito (Kuva 1) [2]. Ohjelmistokehitysprojekti alkaa aina vaatimusmäärittelystä, jolla pyritään saamaan selville käyttäjän tarpeet ohjelmistoa koskien [11].



Kuva 1. *Vesiputousmalli [2]*

Vesiputousmalli on kaikkein tehokkain tapa toteuttaa ohjelmistokehitysprojekti, jos ohjelmiston on suunniteltu täydellisen vaatimusmäärittelyn tuloksena. Ohjelmisto toteutetaan valmiin ja virheettömän suunnittelun perusteella. Suunnitelma täytyy toteuttaa juuri niin kuin on suunniteltu, koska muutokset tulevat sitä kalliimmaksi mitä myöhemmin ne toteutetaan. Vaatimuksiin saattaa kuitenkin tulla muutoksia suunnitteluvirheen, tai suunnitteluvaiheessa huomaamatta jääneiden asioiden takia. Jokainen muutos vaatimuksiin kasvattaa ohjelmistokehityksen kuluja. [11]

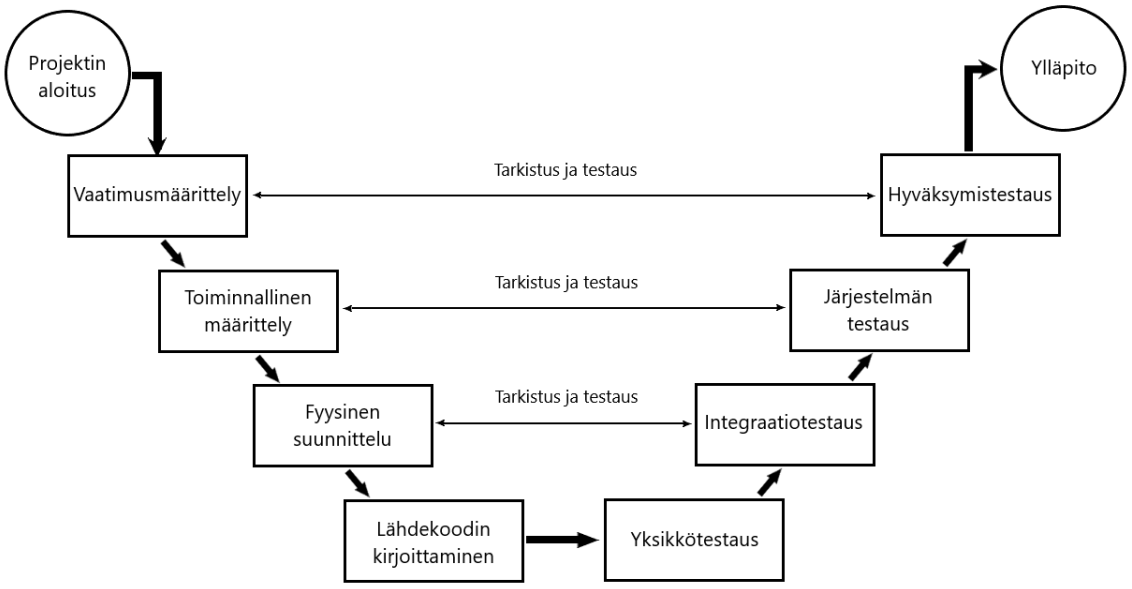
Kun vesiputousmallia noudattaa perusteellisesti, tuloksena on kattava dokumentaatio ohjelmistosta. Erityisesti vaatimusmäärittely- ja testausvaiheessa muodostuu kattava kuvaus tuotteen vaatimuksista, teknisistä tiedoista ja ohjelmiston suunnitelmista. Kattava dokumentaatio ja määrittely auttaa erityisesti testausvaiheessa. Testaajilla, jotka eivät vesiputousmallissa kuulu ohjelmiston kehittäjiin, on kattava kuvaus ohjelmiston toiminnallisuudesta käytössään. Ohjelmiston testaajat vahvistavat, että ohjelmisto vastaa annettua dokumentaatiota. Näin ollen hyvin eritelty ohjelmiston määritelmä testausvaiheen alussa mahdollistaa testaajien tehokkaan työskentelyn. [11]

Suurten innovoivien projektien toteuttaminen ilman suunnitteluvaiheen jälkeisiä muutostarpeita on kuitenkin lähes mahdotonta. Asiakkaan idea ohjelmistosta ja ohjelmistovaatimukset saattavat konkretisoitua vasta ensimmäisen prototyypin, tai toimivan version jälkeen. Vesiputousmallissa uudet muutokset tarkoittavat sitä, että on palattava takaisin

vaatimusmäärittely- ja suunnitteluvaiheeseen, lisättävä uudet vaatimukset ohjelmiston dokumentaatioon ja suunnitelmiin ja päivittää ohjelmiston lähdekoodi vastaamaan muuttuneita vaatimuksia. [11]

Joihinkin projekteihin vesiputousmalli on edelleen paras toteutustapa. Erityisesti pienet selkeät ohjelmistoprojektit, joissa voi suurimmaksi osaksi hyödyntää jo olemassa olevia resursseja. Jos ohjelmistoprojekti etenee suunnitelmien mukaan, vesiputousmalli on toimiva ratkaisu. Suunnitelmiin tulee kuitenkin usein muutoksia kesken ohjelmistoprojektin ja usein suurin osa ongelmista paljastuu vasta ohjelmistoprojektin loppuvaiheissa. [11] Ketterissä malleissa pyritään vastaamaan näihin heikkouksiin. Ketterien mallien luonteeseen kuuluu varautuminen siihen, että muutoksia tulee väistämättä projektin eri vaiheissa [11].

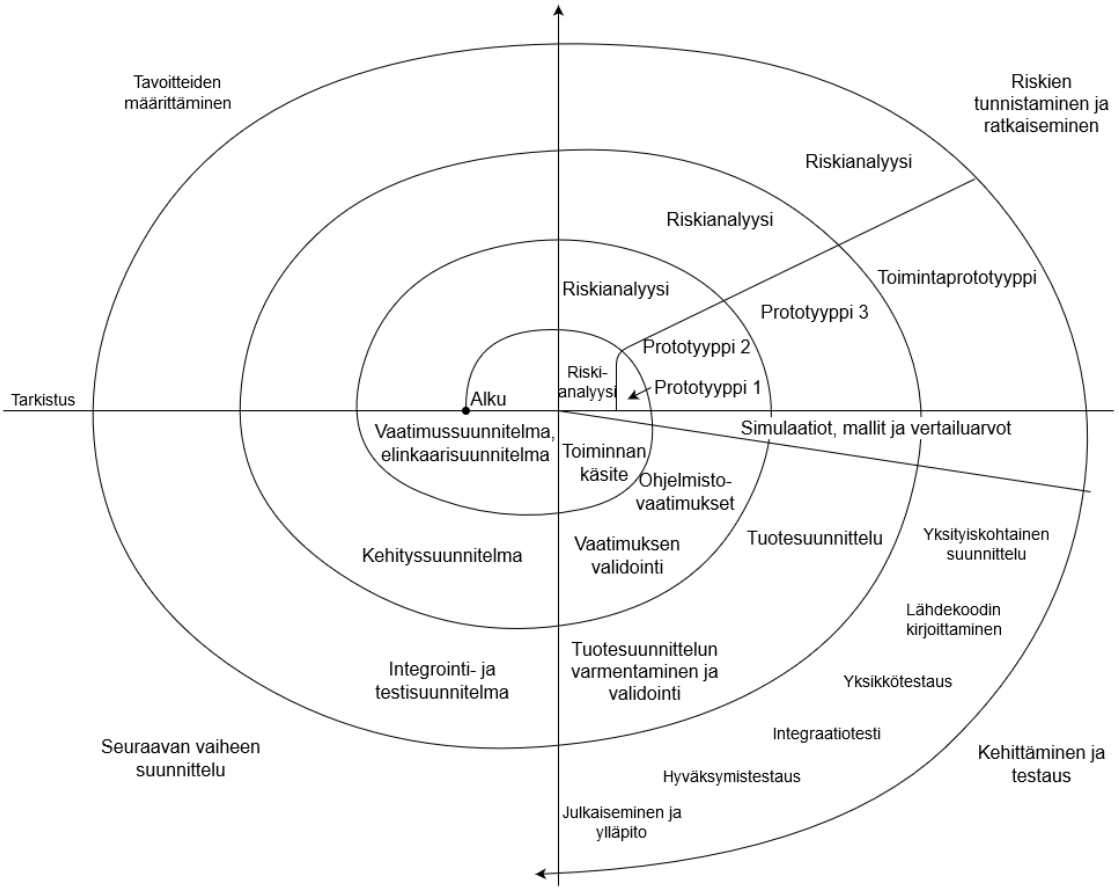
Varmennus- ja validointimallia, joka tunnetaan paremmin nimellä v-malli, pidetään vesiputousmallin jatkeena. V-mallin tärkeä näkökulma on testaustoimien, kuten testaussuunnitelman luominen ja testien suunnittelu tapahtuu hyvissä ajoin ennen lähdekoodin kirjoittamista. Tämä säästää runsaasti aikaa ja testaustiimi saa hyvän ymmärryksen projektista jo alussa, koska testaustiimi on osallisena projektissa heti alusta lähtien. [12] V-malli havainnollistaa, että testisuunnitelmat pitäisi johtaa ohjelmiston määrittelyn ja suunnittelun perusteella (Kuva 2) [2].



Kuva 2. *V-mallin kuvaaja [12]*

Spiraalimallissa jokainen kierros kuvastaa ohjelmistokehitysprosessin eri vaihetta. Sisin kierros voidaan ajatella kuvastavan järjestelmän toteutettavuutta, seuraava vaatimus-

määrittelyä, seuraava ohjelmiston suunnittelua ja seuraava toteutusta (Kuva 3). Jokainen spiraalin kierros voidaan jakaa neljään lohkoon: tavoitteiden määrittely, riskien arviointi, kehittäminen ja vahvistaminen sekä suunnittelu. [2]



Kuva 3. *Spiraalimalli [2]*

Spiraalimallissa suunnittelu ja prototyypit tehdään vaiheissa. Spiraalimallissa keskeistä on riskien arviointi ja projektin riskien minimointi. Tämä voidaan saavuttaa jakamalla projekti pienempiin osiin, joka pienentää muutostarpeiden vaikutuksia ohjelmistokehitysprojektiin. Se auttaa myös riskien arvioinnissa ja päätöksessä jatketaanko ohjelmistokehitysprojektia. [9]

2.2 Ketterä ohjelmistokehitys

Ketterällä ohjelmistokehityksellä tarkoitetaan metodeja, jotka perustuvat vaiheittaiseen ja toistuvaan tuotantoon [4]. Ketterän ohjelmistokehityksen julistuksessa [13] arvostetaan yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja, toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota, asiakasyhteistyötä enemmän kuin sopimusneuvotteluja ja vastaamista muutokseen enemmän kuin pitäytymistä suunnitel-

massa. Useat ketterän ohjelmistokehityksen metodit jakavat samat perusperiaatteet: lyhyet iteroinnit, useasti tapahtuvat ohjelmistojulkaisut, toimintojen priorisointi asiakkaan toiminnosta saaman arvon perusteella, yksinkertainen suunnittelu ja vertaisarviointi. Eri metodeilla on omat käytäntönsä, miten nämä perusperiaatteet määritellään. [4] Ottamalla käyttöön sopivan ketterän ohjelmistokehityksen metodin, ohjelmistokehittäjä pystyy vastaamaan joustavammin ja herkemmin muuttuviin tilanteisiin ja vaatimuksiin [6].

Ketterien menetelmien vahvuuksiin lukeutuu aikaiset versiojulkistukset. Näin asiakas saa arvoa ohjelmistosta jo sen kehityksen alkuvaiheessa. Ensimmäisissä versioissa ohjelmiston vaaditusta toiminnallisuudesta on toteutettuna vain kaikkein suurimman prioriteetin toiminnallisuus. Seuraavissa versioissa toiminnallisuutta korjataan, tai lisätään asiakkaan vaatimusten perusteella. [2]

Ketterissä menetelmissä asiakkaan osallistuminen projektiin on tärkeää, mutta usein haastavaa. Idea ohjelmistokehitykseen täysipäiväisesti osallistuvasta asiakkaasta, joka voi edustaa asiakasta (sidosryhmää), on houkutteleva. Usein tämä ei kuitenkaan ole mahdollista, koska asiakkaaseen kohdistuu samaan aikaan asiakkaan oman organisaation tehtäviä ja vastuita, eikä asiakas voi siksi olla osallisena projektissa kokoaikaisesti. [2] On myös huomattu, että yksi asiakasta edustava henkilö ei aina esitä asiakkaan vaatimuksia tarkasti ja kehittäjät joutuvat usein tulkitsemaan vaatimuksia sekä ottamaan vastuuta niiden priorisoinnista [14]. Haasteena on usein myös vaatimusten/muutosten priorisointi. Varsinkin järjestelmissä, joissa sidosryhmän koko on suuri. Tyypillisesti jokainen sidosryhmän jäsen antaa eri prioriteetin eri muutoksille/vaatimuksille. [2]

Sommerville esittää, että ketterät menetelmät eivät sovellu kaikkeen ohjelmistokehitykseen. Ne soveltuvat parhaiten piennen ja keskisuuren liiketoiminnan omaaviin järjestelmiin ja henkilökohtaisten tietokoneiden ohjelmistoihin. Ketterät menetelmät eivät sovellu hyvin laajojen järjestelmien kehitykseen, joissa kehitystiimit sijaitsevat eri paikoissa ja joissa saattaa olla monimutkaista vuorovaikutusta eri laitteisto- ja ohjelmistojärjestelmien välillä. Ketteriä menetelmiä ei pitäisi myöskään käyttää kriittisten järjestelmien ohjelmistokehityksessä. Näissä järjestelmävaatimusten yksityiskohtainen analysointi on välttämätöntä, jotta pystytään selvittämään niiden vaikutukset järjestelmän turvallisuuteen ja tietoturvaan. [2]

Eri metodeja ovat muun muassa Scrum, Extreme Programming, Kanban, Feature-driven development. Tässä työssä esitellään näistä metodeista vain Scrum esimerkkinä ketterästä ohjelmistokehitysmetodista.

Scrum on ketterän ohjelmistokehityksen metodi, joka keskittyy vahvasti projektinhallintaan sekä iteratiivisiin ja ketteriin tuotantoprosesseihin, avoimuuteen, näkyvyyteen ja yhteistyöhön tuotantotiimin ja asiakkaan välillä. Scrumissa tuotantotiimiä kutsutaan scrumtiimiksi, joka muodostuu analyytikoista, ohjelmistokehittäjistä ja -testaajista. Tällaisella tiiminmuodostuksella pyritään parantamaan tiimin yhteistyötä ja saamaan tiimin jäsenille yhtenäisempi näkemys yhdessä suoritettavista tehtävistä. Scrumtiimiä johtaa ja motivoi scrummaster scrumin arvojen, käytäntöjen ja tuotantoprosessien mukaan. [15]

Scrumin tuotantoprosessi järjestetään tuotteen kehitysjonon perusteella. Tuotteen kehitysjono, josta vastaa tuoteomistaja, on tuotteelta vaadittu toiminnallisuus tärkeysjärjestyksessä. Kehitysjono saattaa muuttua asiakkaan muuttuvien tarpeiden mukaan. Scrumissa ojelmistokehitys suoritetaan sprinteissä. Sprintit koostuvat suoritettavista tehtävistä ja prosesseista, jotka scrumtiimi suorittaa saavuttaakseen sprintin tavoitteet. Sprintin pituus on ennalta määritetty ja se kestää tyypillisesti 5-30 päivää. Scrummaster ja scrumtiimi päättävät sprintin tavoitteista ja sprintin kehitysjonosta jokaisen sprintin alussa pidettävässä suunnittelukokouksessa. Suunnittelukokouksessa määritetään sprintin tavoite ja tuotettavan ohjelmiston toiminnallisuus, joka sprintin aikana tehdään valmiiksi sprintin kehitysjonon perusteella. Koko projektin ajan on näkyvillä edistymiskäyrä, jossa näkyy jäljellä olevan työn määrä suhteessa jäljellä olevaan aikaan. Lyhyissä päivittäisissä scrumkokouksissa projektin jäsenet esittävät lyhyesti mitä he ovat tehneet edeltävän päivän aikana, mitä tehtäviä he tekevät tänään ja mitä haasteita ja esteitä tehtävien suorittamisella saattaa olla. [15]

Sprintin lopussa on sprintin katselmointikokous, jossa scrumtiimi, tuoteomistaja, muu hallinto sekä yksi tai useampi asiakkaan edustaja arvioivat tiimin tuotantoprosessin ja edistymisen sprintin tavoitteeseen verraten. Lopuksi scrumtiimi, scrummaster ja mahdollisesti tuoteomistaja järjestävät tapaamisen, jota kutsutaan sprintin retrospektiiviksi. Retrospektiivissä jokainen osallistuja keskustelee ja arvioi prosessin ja tuotteen onnistumista. [15]

2.3 Vaatimusmäärittely

Vaatimusmäärittelyllä tarkoitetaan prosessia, jolla selvitetään ohjelmiston tarkoitus, eli mitä systeemi tekee ja miten se sen tekee. Perinteisessä ohjelmistokehityksessä vaatimusmäärittely tehdään ohjelmistokehityksen alkuvaiheessa, ennen varsinaisen tuotannon alkua. Tästä syystä vaatimusten muuttaminen myöhemmässä vaiheessa on vaikeaa ja usein kallista, jos se on mahdollista ollenkaan. [4] Ketterässä ohjelmistokehityksessä pyritään välttämään nämä ongelmat.

Elshandidy & Mazen esittävät, että perinteisessä vaatimusmäärittelyssä oletetaan seuraavaa: (1) asiakas tietää tarkasti alusta lähtien, mitä he ohjelmistolta vaativat; (2) ohjelmiston tuottajat ymmärtävät asiakkaan vaatimukset oikein ja selvästi; (3) vain yksi tai useampi sidosryhmä on vastuussa vaatimusten täsmentämisestä; ja (4) eri toimintojen välillä on oltava tarkka rajaus ja keskitytään vain vähän poikkitoiminnallisiin tiimeihin [4].

Vaatimusmäärittelyn voi suorittaa vaiheittain. Ensimmäisenä vaiheena voidaan pitää vaatimusten keräämistä. Tässä vaiheessa määritellään kehitettävän järjestelmän rajoitukset käyttäen eri tekniikoita apuna, kuten prototyyppejä, aivoriihtä (Brainstorming), haastatteluita ja käyttötapauskaaviota. Seuraavaksi neuvotteluilla ja vaatimusten analysoinnilla tarkistetaan, että vaatimukset ovat yhteneväisiä ja toteuttamiskelpoisia, sekä saadaan parempi kokonaiskuva. Tässä vaiheessa voidaan myös muokata vaatimuksia selvemmiksi ohjelmiston tuottajille. Vaatimuksia on myös mahdollista priorisoida ajan tai resurssien mukaan. Dokumentoinnilla vaatimukset määritetään lähtökohdaksi funktionaalisille ja ei funktionaalisille vaatimuksille. Lopuksi varmistetaan, että vaatimukset täyttävät asiakkaan tarpeet. [16]

Vaatimusten keräämisen ja niiden muutosten hallinnan lisäksi täytyy kiinnittää huomiota myös muihin, vaatimuksiin liittyviin tehtäviin:

- Sidosryhmien tunnistaminen: Sidosryhmään kuuluu jokainen, joka on kiinnostunut järjestelmästä tai sen erityistarpeet täyttävästä ominaisuudesta.
- Vaatimusten kerääminen: Ymmärryksen saaminen asiakkaiden ja käyttäjien tarpeista suunniteltuun järjestelmään ja heidän odotuksistaan siihen.
- Vaatimusten tunnistaminen: Tähän vaiheeseen kuuluu vaatimusten ilmaiseminen yksinkertaisilla virkkeillä ja niiden yhdistäminen kokonaisuudeksi. Liiketoiminnan tarpeet tai vaatimukset ovat yrityksen olennaista toimintaa ja ne johdetaan liiketoiminnan tavoitteista. Liiketoiminnan skenaarioita voi käyttää välineenä liiketoiminnan vaatimuksien ymmärtämiseksi. Tärkein asia järjestelmän menestyksessä on missä määrin se tukee liiketoiminnan vaatimuksia ja helpottaa organisaatiota niiden saavuttamisessa.
- Vaatimusten selventäminen ja uudelleen ilmaisu: Tällä varmistetaan, että vaatimukset kuvaavat asiakkaan oikeat tarpeet ja ovat sellaisessa muodossa, että ohjelmistokehittäjät ymmärtävät ne ja voivat käyttää niitä.
- Vaatimusten analysointi: Tällä varmistetaan, että ne ovat hyvin määritelty ja noudattavat hyvien vaatimusten kriteerejä.

- Vaatimusten uudelleenmäärittely tavalla, joka tarkoittaa samaa asiaa jokaiselle sidosryhmän jäsenelle: Sidosryhmän eri jäsenillä saattaa olla keskenään hyvin erilaiset näkökulmat systeemistä ja sen vaatimuksista. Tämä saattaa joskus edellyttää erityisen sanaston tai projektisanakirjan opetteluun ja vaatii usein huomattavan määrän aikaa ja vaivaa yhteisen ymmärryksen saavuttamiseen.
- Vaatimusten määrittely: Tämä vaatii sisällyttämään jokaiseen vaatimukseen niiden kaikki tarkat yksityiskohdat, jotta ne voidaan sisällyttää tekniseen dokumenttiin tai muuhun dokumenttiin projektin koon mukaan.
- Vaatimusten priorisointi: Kaikki vaatimukset eivät ole yhtä tärkeitä suunnitellun systeemin käyttäjille ja asiakkaille. Jotkut ovat kriittisiä, jotkut korkean prioriteetin, toiset keskitason prioriteetin ja osa jopa alhaisen prioriteetin vaatimuksia. Vaatimusten priorisointi on tärkeää, koska kaikkien täyttämiseen ei ole aikaa ja rahaa. Vaatimusten priorisoinnilla on mahdollista kohdistaa resurssit korkeimman prioriteetin vaatimuksiin ja mahdollisesti myöhemmin julkaista versio, joka täyttää myös alemman prioriteetin vaatimukset.

Näiden tehtävien lisäksi alemman tärkeysasteen tehtäviin kuuluvat:

- Uusien vaatimusten johtaminen: Jotkut vaatimukset muodostuvat systeemin suunnittelun vuoksi, vaikka eivät hyödyttäisikään loppukäyttäjää. Esimerkiksi vaatimuksesta säilöä paljon dataa saattaa olla tuloksena vaatimus tallennuskapasiteetista.
- Vaatimusten kategorisointi: Esimerkiksi vaatimusten kategorisointi laitteiston, ohjelmiston, koulutuksen ja dokumentaation mukaan. Tämä prosessi osoittautuu usein odotettua monimutkaisemmaksi, kun useampi kuin yksi kategoria täyttää joidenkin vaatimusten tarpeet.
- Vaatimusten jakaminen: Vaatimukset jaetaan systeemin eri komponenteille ja osajärjestelmille. Jakoa ei voida välttämättä aina tyydyttää vain yhdellä osajärjestelmällä tai komponentilla.
- Vaatimusten seuraaminen: Tarvitaan valmiudet seurata tai jäljittää missä systeemin osassa eri vaatimukset täytetään, että voidaan varmistaa jokaisen vaatimuksen huomiointi. Tämä voidaan suorittaa usein automatisoidulla työkalulla.
- Vaatimusten hallinnointi: Täytyy pystyä lisäämään, poistamaan ja muokkaamaan vaatimuksia systeemin suunnittelun, kehittämisen, integroinnin, testauksen, julkaisun ja käytön aikana.

- Vaatimusten testaus ja todentaminen: Vaatimusten, suunnitelmien, lähdekoodin, testisuunnitelmien ja tuotteiden tarkistaminen, jolla varmistetaan, että vaatimukset on täytetty.
- Vaatimusten validointi: Varmistetaan, että oikeat vaatimukset ovat toteutettu toimitetussa järjestelmässä. Vaatimusten validoinnin järjestys tulisi olla priorisoitu rajallisen rahoituksen vuoksi. [17]

Vaatimusmäärittelyprosessiin käytetään projektin resursseja usein aivan liian vähän. Vähäinen panostaminen vaatimusmäärittelyyn onkin pohjimmainen syy, miksi monet projektit epäonnistuvat [17].

3. KETTERÄ VAATIMUSMÄÄRITTELY

Vaatimusmäärittely on yleisesti hyväksytty olevan kaikkein tärkein, kriittisin ja monimutkaisin prosessi ohjelmistokehityksessä. On tärkeää kehittää laadukkaita ohjelmistoja, jotka tyydyttävät käyttäjän tarpeet ilman virheitä. [18] Toisin kuin perinteisessä ohjelmistokehityksessä, ketterän vaatimusmäärittelyn menetelmät eivät ole peräkkäisiä, vaan ne ovat iteratiivisia ja ne suoritetaan jokaisen lyhyen kehityssyklin (sprintin) aikana [16]. Tämä luo uusia haasteita ketterien menetelmien vaatimusmäärittelylle. Vaatimusmäärittelyn menetelmien rooli ketterissä menetelmissä on edelleen tuntematon ohjelmistokehitysyhteisölle [3]. Perinteinen vaatimusmäärittely perustuu vesiputousmalliin [16]. Siksi perinteisen vaatimusmäärittelyn menetelmiä ei voi sellaisenaan soveltaa ketteriin ohjelmistokehityksen menetelmiin.

Koska ketteriä ohjelmistokehityksen malleja käyttävät yritykset toimivat muuttuvassa ympäristössä, täydellisten ohjelmistovaatimusten määrittäminen on usein käytännössä mahdotonta. Alussa ehdotetut vaatimukset muuttuvat väistämättä, koska asiakkaan on mahdotonta ennustaa, miten järjestelmä vaikuttaa käytössä oleviin käytäntöihin, miten se vaikuttaa muihin järjestelmiin ja mitä käyttäjien toiminnoista pitäisi automatisoida. Oikeat vaatimukset saattavat selventyä vasta ohjelmiston käyttöönoton jälkeen, kun käyttäjille on kertynyt kokemusta siitä. [2]

3.1 Vaatimusmäärittelyn menetelmät ketterässä ohjelmistokehityksessä

Ketterän ohjelmistokehityksen julistuksen yksi takana olevista periaatteista on: ”Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa.” [13] Ketterässä kehityksessä siis oletetaan, että vaatimusmäärittely jatkuu systeemin koko eliniän ajan [16].

Koska ketterä ohjelmistokehitys tapahtuu lyhyissä iteraatioissa, joiden tuloksina saadaan useita julkaisuja (inkrementtejä), ohjelmistokehitysprosessi on dynaaminen. Vaatimukset määritellään aluksi asiakkaan kanssa ja ne listataan asiakkaan toivelistan muotoon. Asiakkaan vaatimuksista keskustellaan muutaman viikon välein. Keskusteluissa vaatimusten ymmärtäminen parantuu, sekä niitä voidaan priorisoida seuraavaa iteraatiota varten [4]. Tämä toimintatapa on herättänyt kysymyksiä, miten vaatimusmäärittelyyn tulisi suhtautua näin joustavassa ja dynaamisessa työskentelytavassa. Vaatimusmäärittelyn muuttunut rooli aiheuttaa haasteita sen toteuttamiselle. [16]

Varhaiset inkrementit voivat tarjota korkean prioriteetin toiminnallisuutta, joten asiakas saa arvoa ohjelmistosta jo kehityksen aikaisessa vaiheessa. Asiakkaat voivat nähdä vaatimuksensa käytännössä ja he voivat sen perusteella määrätä niihin muutoksia myöhempiin versiojulkaisuihin. [2]

Ketterien ohjelmistokehitysprosessien kokonaisvaltainen luonne ja keskittyminen kasvokkaiseen viestintään tekevät vaikeaksi erottaa vaatimusmäärittelyyn kuuluvat ja vaatimusmäärittelyyn kuulumattomat ketterät käytännöt toisistaan. Yhteistyössä suunnitellut lyhyet iteraatiot, tai päivittäisten tiimipalavereiden järjestäminen, joita ei perinteisesti pidetä osana vaatimusmäärittelyä, mahdollistavat kevyet vaatimusmäärittelyyn liittyvät käytännöt, kuten käyttäjätarinoiden kirjoittamisen. [19]

Käyttötapaukset ovat joustava tapa kuvata funktionaalisia vaatimuksia. Ne voidaan esittää eri abstraktiotasoilla suppeista kaavioista, käyttötapauskaavioihin, tärkeimpiin skenaarioihin, poikkeuksiin ja vaihtoehtoisiin vaiheisiin. Osa käyttötapauksista voidaan esittää vain käyttötapauskaavioina ilman lisäkuvauksia ja muissa voi olla määritelty lisäksi poikkeuksia ja vaihtoehtoisia vaiheita. [5] Käyttötapaukset ovat epävirallisia ja helppoja käyttää, mikä auttaa sidosryhmiä ymmärtämään ja vahvistamaan vaatimukset [6].

Tärkein päämäärä vaatimusmäärittelyssä ketterässä ympäristössä on saada asiakas artikuloimaan ja kommunikoimaan omat tarpeensa. Tämä tapahtuu kasvokkain käytävällä kommunikaatiolla. Haastattelut ovat perinteisin ja eniten käytetty metodi. Haastattelu voi olla jäsennelty valmiilla kysymyksillä, tai se voi olla jäsentelemätön, jossa painoarvo on enemmän avoimella keskustelulla. Menetelmänä voi lisäksi käyttää Joint Application Design (JAD), jossa seurataan hyvin jäsenneltyä ryhmäistunnon kaavaa tarkoilla ennalta määritetyillä vaiheilla ja toiminnoilla. Lisäksi jokaiselle osallistujalle on omat ennalta määritellyt roolit. [4] On todettu, että merkittävin tekijä projektin onnistumiselle on asiakkaan aktiivinen osallistuminen [3].

Käyttäjätarinoilla kirjataan ketterissä menetelmissä järjestelmän vaatimukset. Käyttäjätarinat ovat yleisin väline ketterässä ohjelmistokehityksessä [20]. Käyttäjätarina kuvaa ominaisuuden, joka antaa arvoa asiakkaalle, ja se on yleensä kirjoitettu paperilapulle. Jokaisen paperilapun etupuolelle kirjataan käyttäjätarinan kuvaus ja takapuolelle kirjataan sen hyväksymistestit. Niitä käytetään myös vaatimusten suunnittelussa ja dokumentoinnissa. Asiakkailla on yleensä vastuu käyttäjätarinoiden kirjoittamisesta. Näin varmistutaan siitä, että ne ovat kirjoitettu sellaisella liikekielellä, jonka he ymmärtävät. Ennen käyttäjätarinoiden kirjoittamista, voidaan olettaa, että asiakkaiden täytyy ajatella mitä he odottavat järjestelmältä. Tämän jälkeen he miettivät erityistä toiminnallisuutta, joka

toteuttaa järjestelmälle asetetut odotukset. Tätä mietintää vastaavana tapahtumana voidaan pitää perinteisen ohjelmistokehityksen aivoriihtä. [4]

Prototyypit ovat yksi nopeimmista ja tehokkaimmista tavoista auttaa asiakasta visualisoimaan vaatimukset [4]. Prototyyppejä käytetään vaatimusmäärittelyn tukena vaatimusten esiin tuomisessa ja niiden validoinnissa. Ketterissä menetelmissä prototyypit eivät ole osa kehitettävää ohjelmistoa, joten niitä käsitellään poisheitettävinä prototyyppeinä [2, 4]. Asiakkaat saattavat saada uusia ideoita vaatimuksiin ja huomata ohjelmiston heikkoudet ja vahvuudet helpommin prototyyppien avulla. He voivat sen jälkeen ehdottaa uusia vaatimuksia ohjelmistolle. Prototyyppejä käytetään myös konseptien demonstroinnissa, eri suunnitteluvaihtoehtojen kokeiluissa ja yleisesti ongelmien selventämisessä ja ongelmanratkaisuissa. Prototyyppejä tehdään usein niistä järjestelmän osista, jotka ymmärretään heikoiten. Vastaavasti ohjelmistokehitys aloitetaan niistä osista, jotka ovat selkeimpiä ja prioriteettina korkeimmalla. [2] Prototyyppejä on kolmea eri päätyyppiä: matalan tarkkuuden prototyypit, korkean tarkkuuden prototyypit ja wizard-of-oz prototyypit. Matalan tarkkuuden prototyyppeihin lukeutuu nopeasti käsin luonnostellut, tai tietokoneohjelmalla luonnostellut kuvaukset käyttöliittymästä. Korkean tason prototyypit ovat nopeasti valmistettavia realistisia luonnoksia käyttöliittymästä joko ohjelmana, tai verkkosivuna. Wizard-of-oz prototyypissä ohjaava henkilö simuloi järjestelmän vastauksia käyttäjän syötteeseen. [4]

Vaatimuksia priorisoidaan kaiken aikaa ja suurimman prioriteetin vaatimukset toteutetaan ensin. Asiakas näkee inkrementeissä vaatimusten toteutumisen kehitettävässä ohjelmistossa [4]. Scrumissa priorisointi toteutuu tuoteomistajan ylläpitämässä kehitysjonossa koko projektin ajan. Vaatimusmäärittely ei juurikaan muutu ohjelmistokehitysprojektin aikana ketterissä menetelmissä. Niitä kerätään ja tarkennetaan koko ajan projektin edetessä.

3.2 Vaatimusmäärittelyn haasteet ketterässä ohjelmistokehityksessä

Ketterän ohjelmistokehityksen vaatimusmäärittelyn haasteet perustuvat pääosin mallin nopeatempoiseen muuntuvuuteen. Vähäinen dokumentointi aiheuttaa haasteita kehitystiimeille, kun ketterissä menetelmissä korvataan perinteinen vaatimusten dokumentointi käyttäjätarinoilla, joka voi luoda tiedonkulkuun aukkoja. Jos nämä aukot syntyvät äkillisten vaatimusten muutosten, asiakasedustajan tavoitettavuuden, projektin monimutkaisuuden ja dokumentaation puutteen vuoksi, ongelmat kasaantuvat. [3] Dokumentaation heikkoudet ovat havaittavissa erityisesti suurissa ja keskisuurissa projekteissa, koska kommunikaatio kansainvälisten tiimien kesken voi olla hankalaa [6].

Asiakkaan oletetaan olevan tavoitettavissa ketterissä malleissa. Tämä ei kuitenkaan ole aina mahdollista. [4] Asiakkaan tavoitettavuus on todettu olevan yleinen haaste. Liiketoiminnan näkökulmasta rajoittaviksi tekijöiksi muodostuvat aika, kasvavat kulut ja asiakasedustajaan kohdistuva työmäärä. [3]

Budjetin ja aikataulun arviointi voi olla haasteellista ketteriä menetelmiä käyttäville organisaatioille [4]. Lisäksi projektin alussa viimeistelty arkkitehtuuri voi muodostua soveltumattomaksi vaatimusten muuttuessa muodostaen turhaa lähdekoodia. [3]

Prototyyppejä pidetään yksinkertaisena tapana tarkistaa vaatimusten määrittely asiakkaiden kanssa. Tämä kasvattaa asiakkaan odotuksia tuotettavalta järjestelmältä, joka saattaa aiheuttaa haasteita, koska asiakkaan asettamien vaatimuksien määrä kasvaa poikkeuksellisesti. Lisäksi monen korkean tarkkuuden prototyypin tekeminen voi kasvattaa kuluja huomattavasti. [3]

Startup yritykset hyödyntävät ketterien mallien niitä ominaisuuksia, jotka nopeuttavat ohjelmistokehitystä, mutta samaan aikaan jättävät huomioimatta ketterät menetelmät, jotka parantavat ohjelmiston laatua. Ohjelmistokehitystä nopeuttaviin ominaisuuksiin lukeutuu muun muassa lyhyet iteraatiot, iteraatioiden suunnittelu, julkaisujen suunnittelu. Laatua parantaviin ominaisuuksiin lukeutuu muun muassa yksikkötestaus, lähdekoodin refaktorointi, testiohjattu kehitys. Laadulla on alhainen prioriteetti ja startup yrityksissä teknistä velkaa kertyy varsinkin yritysten varhaisissa vaiheissa. Alhainen käyttöaste laatua parantavilla menetelmillä ei ole ainoastaan startup yrityksille tyypillistä, vaan sitä on havaittavissa yleisesti ohjelmistoalan yrityksissä. [7]

Globalisaation ajaessa organisaatioita entistä hajautetummiksi, eri alueilla tapahtuvasta ohjelmistokehityksestä on tulossa normaalia. Siksi onkin tarpeellista valmistella ohjelmistokehitysprojektit hallitsemaan työskentely hajautetuissa ympäristöissä. Erityisesti suurissa ja hajautetuissa ohjelmistokehitysprojekteissa korostuu säännöllisten kasvokkaisten tapaamisten tärkeys. Tällaisissa suurissa projekteissa vaatimukset on usein määritelty verkkosivulla, joka mahdollistaa käyttäjille sen sisällön ja rakenteen muokkaamisen yhteistyössä (wiki). [1]

4. YHTEENVETO

Ketterän ohjelmistokehityksen julistuksen yksi oleellisimmista periaatteista on: ”Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa.” [13] Ketterä ohjelmistokehitys on sopeutunut tämän päivän nopeatempoiseen liikemaailmaan, jossa ohjelmistokehityksen täytyy pystyä mukautumaan muuttuvan ympäristön luomiin haasteisiin.

Ketteristä ohjelmistokehityksen menetelmistä on tullut yleisiä niin pienissä kuin suurissakin ohjelmistoalan yrityksissä. Koska startup yrityksiin liittyy erilaisia epävarmuustekijöitä ja koska niiltä vaaditaan kykyä reagoida muutoksiin nopeasti, ketterät ohjelmistokehitysmallit ovat luonnollinen valinta myös niiden käyttöön. [7] Globalisaatio ja ketterän vaatimusmäärittelyn nopea muuntuvuus luovat ketterille ohjelmistokehityksen menetelmille haasteita, mutta sen on kuitenkin todettu johtavan projekteissa hyviin lopputuloksiin.

Selkeää vaatimusmäärittelyn prosessia ohjelmistokehityksen ketterissä malleissa on vaikea kuvata, sillä vaatimusmärittely elää projektin mukana. Kirjallisuudessa on selvitetty useita eri työkaluja ja tekniikoita, joilla vaatimusmäärittelyä tehdään, mutta lopulliset tekniikat valikoituvat kuitenkin projektin tyylin ja sen tavoitteiden mukaan.

Ketterä ohjelmistokehitys ei ole myöskään täysin irtautunut perinteisestä ohjelmistokehityksestä. Eri tekniikoita käytetään niin perinteisissä kuin ketterissä menetelmissä. Molemmissa on kuitenkin konsensus siinä, että muuntautumiskyky muuttuviin vaatimuksiin on tärkeämpää kuin pitäytyminen tiukasti suunnitelmassa. [6] Mielestäni onkin oleellista tietää ja ymmärtää molempien kehitysmallien periaatteet ja toimintakäytännöt, jotta pystyy mukauttamaan myös oman osaamisensa tämän päivän työympäristöön.

LÄHTEET

[1] A. Safwat, M.B. Senousy, Addressing Challenges of Ultra Large Scale System on Requirements Engineering, Procedia Computer Science, Vol. 65, 2015, pp. 442-449.

[2] I. Sommerville, Software engineering, 8th ed. Addison-Wesley, Harlow, 2006, .

[3] I. Inayat, S.S. Salim, S. Marczak, M. Daneva, S. Shamshirband, A systematic literature review on agile requirements engineering practices and challenges, Computers in Human Behavior, Vol. 51, Iss. Part B, 2015, pp. 915-929.

[4] H. Elshandidy, S. Mazen, Agile and traditional requirements engineering: A survey, International Journal of Scientific & Engineering Research, Vol. 4, Iss. 9, 2013, pp. 473-482.

[5] Agile requirements engineering: A research perspective, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2014, pp. 40-51.

[6] L. Zamudio, J.A. Aguilar, C. Tripp, S. Misra, A requirements engineering techniques review in agile software development methods, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 683-698.

[7] Are software startups applying agile practices? The state of the practice from a large survey, in: Lecture Notes in Business Information Processing, Springer Verlag, 2017, pp. 167-183.

[8] T. Dingsøyr, N.B. Moe, T.E. Fægri, E.A. Seim, Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation, Empirical Software Engineering, Vol. 23, Iss. 1, 2018, pp. 490-520.

[9] A. Alshamrani, A. Bahattab, A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model, International Journal of Computer Science Issues (IJCSI), Vol. 12, Iss. 1, 2015, pp. 106-111. Available (accessed ID: proquest1660801422): .

[10] L.R. Vijayasarathy, C.W. Butler, Choice of Software Development Methodologies: Do Organizational, Project, and Team Characteristics Matter? IEEE Software, Vol. 33, Iss. 5, 2016, pp. 86-94.

[11] T. Stober, U. Hansmann, Traditional Software Development, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, 15-33 p.

[12] G. Regulwar, P. Deshmukh, R. Tugnayat, P. Jawandhiya, V. Gulhane, Variations in V Model for Software Development, International Journal of Advanced Research in Computer Science, Vol. 1, Iss. 2, 2010, Available (accessed ID: proquest1443699838): .

[13] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland & D. Thomas, Manifesto for Agile Software Development, http://agilemanifesto.org/.

[14] V.T. Heikkila, D. Damian, C. Lassenius, M. Paasivaara, A Mapping Study on Requirements Engineering in Agile Software Development, 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, IEEE, pp. 199-207.

[15] K. Kautz, T.H. Johanson, A. Uldahl, The Perceived Impact of the Agile Development and Project Management Method Scrum on Information Systems and Software Development Productivity, Australasian Journal of Information Systems, Vol. 18, Iss. 3, 2014, .

[16] K. Curcio, T. Navarro, A. Malucelli, S. Reinehr, Requirements engineering: A systematic mapping study in agile software development, The Journal of Systems & Software, Vol. 139, 2018, pp. 32-50.

[17] R.R. Young, I. Books24x7, The requirements engineering handbook, illustrat ed. Artech House, Boston, 2003, .

[18] Requirements: Towards an understanding on why software projects fail, in: AIP Conference Proceedings, American Institute of Physics, Melville, 2016, .

[19] M. Ochodek, S. Kopczyńska, Perceived importance of agile requirements engineering practices – A survey, The Journal of Systems & Software, Vol. 143, 2018, pp. 29-43.

[20] E. Schön, J. Thomaschewski, M.J. Escalona, Agile Requirements Engineering: A systematic literature review, Computer Standards & Interfaces, Vol. 49, 2017, pp. 79-91. Available (accessed ID: elsevier_sdoi_10_1016_j_csi_2016_08_011): .