Tatu Ruuska

VAATIMUSMÄÄRITTELY KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2012

TIIVISTELMÄ

Ruuska, Tatu
Vaatimusmäärittely ketterässä ohjelmistokehityksessä
Jyväskylä: Jyväskylän yliopisto, 2012, 78 s.
Tietojärjestelmätiede, pro gradu -tutkielma.
Ohjaaja(t): Leppänen Mauri

Tämän pro gradu -tutkielman tavoitteena on selvittää, miten vaatimusmäärittely toteutetaan ketterässä ohjelmistokehityksessä ja millaisia käytänteitä sekä tekniikoita vaatimusmäärittelyssä voidaan käyttää. Aihetta käsitellään vertailemalla eroja perinteisen ja ketterän lähestymistavan välillä erityisesti vaatimusmäärittelyn osalta. Ketterän vaatimusmäärittelyn käytänteitä ja tekniikoita tarkastellaan yksityiskohtaisesti. Tutkielma perustuu aihetta käsittelevään kirjallisuuteen.

Ohjelmistokehityksen osalta lähestymistapojen erot näkyvät mm. projektien johtamisessa ja hallinnassa, yleisessä toimintatavassa, kehittäjien ja asiakkaiden rooleissa, suunnittelussa, arkkitehtuurissa ja toteutuksessa. Ketterä vaatimusmäärittely sisältää samat vaiheet kuin perinteisessä lähestymistavassa, joskin erilailla ajoitettuina ja painotettuina. Vaatimusmäärittely on kuitenkin jatkuvaa, ja vaatimukset kehittyvät projektin edetessä. Perinteisen vaatimusmäärittelyn vaiheet suoritetaan suurelta osin peräkkäisinä, kun taas ketterässä vaatimusmäärittelyssä niitä suoritetaan voimakkaasti iteroiden. Tutkielmassa esitellään myös laajasti ketterään vaatimusmäärittelyyn tarkoitettuja käytäntöjä ja tekniikoita, jotka liittyvät mm. vaatimusten esittämiseen, jakamiseen, priorisointiin, dokumentointiin ja jäljitykseen. Lopuksi työssä esitellään myös ketterän vaatimusmäärittelyn haasteita, joihin on tärkeää varautua ohjelmistokehityksessä.

Tutkimuksen tuloksia voidaan hyödyntää yleiskuvan saamiseksi perinteisen ja ketterän vaatimusmäärittelyn eroista. Tutkimuksessa esitettyjä ketterän vaatimusmäärittelyn tekniikoita voidaan harkita käytettäväksi käytännön projekteissa

Asiasanat: vaatimusmäärittely, perinteinen lähestymistapa, ketterä lähestymistapa, XP, Scrum

ABSTRACT

Ruuska, Tatu
Requirements Engineering in Agile Software Development
Jyväskylä: University of Jyväskylä, 2012, 78 p.
Information Systems, Master's Thesis
Supervisor(s): Leppänen, Mauri

This Master's thesis aims to find out how requirements engineering is implemented in agile software development and what kinds of practices and techniques have been suggested for usage in agile requirements engineering. The research is conducted by first comparing the traditional and agile development approaches, especially focusing on differences in requirements engineering. Second, a large set of practices of agile requirements engineering are described in detail. This thesis is based on the literature about requirements engineering and agile software development.

The study shows that the traditional and agile approaches differ in project management and leadership, developers' and customers' roles, planning, architecture and implementation. The phases of the traditional requirements engineering can also be found in agile requirements engineering, although with different timing and emphasis. Requirements engineering process in agile development is continuous and requirements evolve throughout the project. In traditional requirements engineering, the phases are executed in a consecutive manner, whereas in agile requirements engineering the phases are executed in an iterative manner. The study also describes a large array of practices of agile requirements engineering to be used, among others, in presenting, decomposing, prioritizing, documenting and tracing requirements. Agile requirements engineering includes numerous challenges, which are outlined.

The results of this study can be utilized to have an insight into differences between traditional and agile requirements engineering. The descriptions of the practices of agile requirements engineering help planning the way of how an agile software engineering project is executed.

Keywords: requirements engineering, traditional approach, agile approach, XP, Scrum

KUVIOT

SISÄLLYS

1 JOHDANTO

Vaatimusmäärittely on yksi keskeisimmistä ja samalla haastavimmista osista ohjelmistojen kehittämistä (Orr, 2004). Useimmat ohjelmistossa tai järjestelmässä esiintyvistä virheistä ja puutteista johtuvat huonosti tehdystä vaatimusmäärittelystä. Tämän seurauksena on mahdollista, että toimitukset viivästyvät, kehitys- ja ylläpitokustannukset kasvavat ja asiakkaiden tyytymättömyys lisääntyy. Lopputuloksena on huonosti toimiva järjestelmä, jota ei pahimmassa tapauksessa haluta edes käyttää (Kotonya & Sommerville, 2002).

Vaatimusmäärittelyllä tarkoitetaan prosessia, joka kattaa toiminnan vaatimusten esille saannista aina niiden validointiin saakka. Se jaetaan perinteisesti kahteen osaan, vaatimusten kehittämiseen (engl. requirements development) ja vaatimusten hallintaan (engl. requirements management). Kehittämisosa jakaantuu edelleen neljään vaiheeseen: vaatimusten esillesaanti (engl. elicitation), neuvottelu ja analysointi, dokumentointi ja validointi. Vaatimustenhallintaan kuuluvat muutosten hallinta, version hallinta, tilan seuranta, tunnistus sekä jäljityslinkistö. (Kotonya & Sommerville, 2002)

Perinteisen lähestymistavan mukaan toimittaessa vaatimusmäärittelyn vaiheet ajoittuvat pääosin peräkkäin (Kotonya & Sommerville (2002) ja ne edeltävät muita ohjelmistokehityksen vaiheita (Royce 1970). Viimeisen vuosikymmenen aikana on perinteisen lähestymistavan rinnalle tullut ketterä lähestymistapa (engl. agile approach), joka perustuu Agile-manifestiin (engl. Agile manifesto, Agile Alliance, 2001). Agile–manifesti esittää ketterän kehittämisen arvot ja periaatteet. Uusi lähestymistapa syntyi vastareaktiona perinteiselle ohjelmistokehittämiselle, jota on kritisoitu kankeaksi ja prosessiltaan ennalta määritellyksi (Bruegge, Reiss & Schiller, 2009). Ketterään kehittämiseen on esitetty useampia menetelmiä, esimerkiksi Scrum (Sutherland & Schwaber, 2010), XP (Beck, 1999), Kanban (Anderson, 2010) ja DSDM (Stapleton, 1997). Niille on yhteistä pyrkimys asiakastyytyväisyyden lisäämiseen, muuttuviin vaatimuksiin reagoimiseen, useisiin toimivan ohjelmiston julkaisuihin sekä asiakkaan ja kehittäjien yhteistyön tiivistäminen.

Ketterässä vaatimusmäärittelyssä voidaan käyttää soveltuvin osin perinteisen vaatimusmäärittelyn periaatteita ja käytänteitä (Paetsch, Eberlain & Maurer, 2003). Mutta ketterä lähestymistapa tuo myös muutoksia. Esimerkiksi ketterässä kehittämisessä vaatimuksia priorisoidaan voimakkaammin ja kussakin iteraatiossa keskitytään asiakkaan näkökulmasta kaikkein tärkeimpien vaatimusten toteuttamiseen. Vaatimusmäärittelyä tehdään koko projektin ajan, minkä johdosta asiakas on tiiviisti mukana projektin aloituksesta sen päättämiseen asti (Sillitti & Succi, 2005).

Kirjallisuudessa on esitetty myös paremmin ketterään vaatimusmäärittelyyn sopivia käytäntöjä. Rameshin, Caon ja Baskervillen (2007) mukaan käytänteet ovat kasvokkain käyty viestintä, iteratiivinen vaatimusmäärittely, priorisoidut vaatimukset, jatkuva suunnittelu vaatimusten hallinnassa, prototyypit, arviointipalaverit ja hyväksyntätestaus. Paetschin ym. (2003) mukaan ketterissä menetelmissä vaatimusmäärittelyn käytänteitä ovat asiakkaan läsnäolo, haastattelut, mallintaminen, dokumentointi, validointi ja vaatimusten jäljitys.

Tämän pro gradu -tutkielman kohteena on ketterä vaatimusmäärittely. Tutkimusongelmana on:

Millainen on vaatimusmäärittely ketterässä ohjelmistokehityksessä?

Tutkimusongelma voidaan jakaa seuraaviin tutkimuskysymyksiin:

- Mitä ketterällä kehittämisellä tarkoitetaan?
- Miten ketterä vaatimusmäärittely eroaa perinteisestä vaatimusmäärittelystä?
- Millaisia käytänteitä ja tekniikoita on esitetty kirjallisuudessa käytettäväksi ketterässä vaatimusmäärittelyssä?
- Millaisia haasteita liittyy ketterään vaatimusmäärittelyyn?

Aihetta tarkastellaan vaatimusmäärittelyä ja ketterää kehittämistä käsittelevän kirjallisuuden pohjalta käsitteellisteoreettisen tutkimusotteen mukaisesti.

Tutkielma on jäsennetty seitsemään lukuun. Luvussa 2 esitellään yleisesti vaatimusmäärittelyä sellaisena kuin se perinteisessä kehittämisessä ymmärretään. Ensiksi esitetään ohjelmistonkehitysprosessi ja vaatimusmäärittelyn asema siinä. Toisena annetaan vaatimuksen määritelmä, ja kolmantena kuvataan vaatimusmäärittelyn vaiheet. Luvussa 3 kuvaillaan ketterien menetelmien käytänteitä ja periaatteita Agile-manifestin perusteella. Ketteristä menetelmistä käsitellään tarkemmin XP:n ja Scrumin sisältämiä käytänteitä ja rooleja. Luvussa 4 verrataan perinteistä ja ketterää lähestymistapaa sekä ketterää ja perinteisestä vaatimusmäärittelyä toisiinsa. Luvussa 5 esitellään ketterässä vaatimusmäärittelyssä käytettyjä periaatteita ja tekniikoita yksityiskohtaisesti. Luvussa 6 esitetään ketterän vaatimusmäärittelyn haasteita. Tutkielma päättyy lukuun 7, jossa esitetään yhteenveto, tutkimuksen tulokset ja rajoitukset arvioidaan niiden hyödynnettävyyttä ja esitetään jatkotutkimusaiheita.

2 VAATIMUSMÄÄRITTELY

Tämän luvun tarkoituksena on esitellä yleisesti vaatimusmäärittelyä siten kuin se perinteisesti ymmärretään. Perinteistä käsitystä edustavat muiden muassa Pressman (2001), Kotonya ja Sommerville (2002), van Lamsveerde (2009) ja Pohl (1994). Aluksi esitetään ohjelmistokehitysprosessi ja osoitetaan, mikä on vaatimusmäärittelyn asema siinä. Toiseksi määritellään vaatimuksen käsite ja esitetään erilaisia luokituksia vaatimuksille. Kolmanneksi kuvataan, minkälaisista vaiheista vaatimusmäärittely koostuu. Tämän jälkeen kutakin vaihetta ja niissä käytettäviä tekniikoita kuvataan tarkemmin omissa alaluvuissaan. Viimeisenä tässä luvussa on vaatimushallinnan esittely ja yhteenveto

2.1 Vaatimusmäärittely osana ohjelmistokehitystä

Ohjelmistokehityksellä (engl. software development, software engineering) tarkoitetaan ohjelmiston järjestelmällistä kehittämistä, käyttöä ja ylläpitoa (IEEE, 1990). Pressman (2001) mainitsee, että ohjelmistokehitys sisältää prosessin, hallinnan, tekniset menetelmät sekä työkalut. Ohjelmistokehityksen hallittavuuden, laadun ja tuottavuuden parantamiseksi on kehitetty lukuisia prosessimalleja. *Prosessimalli* osoittaa, miten ohjelmistot ja järjestelmät suunnitellaan ja toteutetaan. Tunnetuimpia perinteisistä prosessimalleista ovat vesiputousmalli (engl. waterfall model, linear sequential model) (Royce, 1970), prototyyppimalli (Floyd, 1984), RAD-malli (Martin, 1991), inkrementaalinen malli (Pressman, 1997) ja spiraalimalli (Boehm, 1988). Mallit koostuvat pääosin samanlaisista tehtävistä, mutta niiden suoritusjärjestyksissä on eroja. Seuraavassa tarkastellaan ohjelmistokehitysprosessin tehtäviä yksinkertaisimman mallin eli vesiputousmallin vaiheiden mukaisesti.

Vesiputousmallin mukaan ohjelmistokehitys koostuu analyysistä, suunnittelusta, toteutuksesta, testauksesta ja ylläpidosta (Kotonya & Sommerville, 2002). Analysoinnilla pyritään selvittämään mitä ohjelmiston täytyy tehdä (Pressman 2001). Suunnittelu on monivaiheinen prosessi. Tässä vaiheessa huomion kohteena ovat ohjelmiston tietorakenne, ohjelmistoarkkitehtuuri, rajapintakuvaus ja toimintalogiikan kuvaus algoritmitasolla (engl. algorithmic detail). Suunnitteluvaihe muuntaa ohjelmistolle asetetut vaatimukset sellaiseen muotoon, että niiden laatu voidaan arvioida. Tämä tehdään ennen ohjelmoinnin aloittamista. Suunnitteluvaihe dokumentoidaan ohjelmiston konfigurointia varten. (Pressman, 2001)

Kolmas vaihe on toteutus. Pressmanin (2001) mukaan edellisessä vaiheessa tehty suunnittelu käännetään koneluettavaan muotoon koodauksella. Kääntäminen voidaan suorittaa joskus mekaanisesti, riippuen siitä, kuinka tarkasti suunnittelu on tehty.

Neljäntenä vaiheena vesiputousmallissa on testaus. Pressmanin (2001) kuvaamana vaiheen tarkoituksena on testata ohjelmiston sisäiset loogisuudet ja ulkoiset toiminnot. Ensin mainitun tarkoituksena on käskytason (engl. statements) testaus, viimeksi mainitun tarkoituksena ovat virheiden löytäminen ja sen varmistaminen, että ohjelmiston syötteistä saadaan vaatimusten mukaiset tulokset.

Viides ja viimeinen vaihe on ylläpito. Muutokset ohjelmistossa ovat välttämättömiä ohjelmiston julkistamisen jälkeen. Vaiheen tarkoituksena on mahdollisten virheiden korjaaminen, ulkoisen ympäristön muutosten (esim. uusi käyttöjärjestelmä) huomioiminen ohjelmistossa sekä asiakkaiden vaatimien lisäysten ja muutosten tekeminen ohjelmistoon esimerkiksi suorituskyvyn parantamiseksi tai uusien toiminnallisuuksien aikaansaamiseksi.

Vaatimusmäärittely sisältyy ohjelmistokehityksen ensimmäiseen vaiheeseen. Tässä tutkielmassa *vaatimusmäärittelyllä* tarkoitetaan kaikkea sitä toimintaa, jolla vaatimukset saadaan esille, dokumentoitua, validoitua ja ylläpidettyä. Tässä työssä tarkastellaan tätä osaa ohjelmistokehityksestä.

2.2 Vaatimuksen käsite ja vaatimusluokkia

Vaatimukselle on esitetty kirjallisuudessa monenlaisia määritelmiä. IEEE:n standardin (1990) mukaan vaatimuksella tarkoitetaan tilaa tai pystyvyyttä, jonka järjestelmä tai järjestelmän komponentti täyttää sopimuksen, standardin tai formaalin dokumentoinnin mukaisesti. Kotonya ja Sommerville (2002, 4) määrittelevät *vaatimuksen* määrityksenä, joka kuvaa palvelut, joita järjestelmän tulisi tarjota, sekä rajoitteet järjestelmän toiminnalle. Vaatimukset kerätään erilaisilla tekniikoilla sidosryhmiltä heidän esittämiensä tarpeiden ja toiveiden täyttämiseksi. Tässä työssä käytetään Kotonyan ja Sommervillen (2002) määritelmää, koska se kuvaa selkeästi, mitä vaatimuksella tarkoitetaan ja mikä on sen tehtävä vaatimusmäärittelyssä.

Vaatimukset voidaan jakaa toiminnallisiin ja ei-toiminnallisiin vaatimuksiin. Toiminnalliset vaatimukset liittyvät järjestelmän toimintoihin, kun taas ei-toiminnalliset vaatimukset liittyvät järjestelmän rajoituksiin. Kuten vaatimukselle, myös toiminnallisille ja ei-toiminnallisille vaatimuksille on esitetty erilaisia määritelmiä.

van Lamsweerden (2009) mukaan *toiminnalliset vaatimukset* määrittävät, millaisia toiminnallisia ominaisuuksia ohjelmistolla tulisi olla. Toiminnalliset vaatimukset voivat viitata ympäristön olosuhteisiin, missä vaatimuksen esittämä toiminto tapahtuu. IEEE standardi (1990) kuvaa toiminnallista vaatimusta vaatimuksena, joka määrittää suoritettavan toiminnan järjestelmälle tai järjestelmän komponentille. Pohl (1994) kuvaa toiminnalliset vaatimukset järjestelmän pakollisiksi tehtäviksi.

Kotonyan ja Sommervillen (2002) mukaan *ei-toiminnalliset vaatimukset* määrittelevät järjestelmän yleiset ominaisuudet ja piirteet. Ne asettavat ehdot, miten käyttäjän toiminnalliset vaatimukset toteutetaan. Usein suurempi järjestelmä määrittelee siihen sulautettavan ohjelmiston ei-toiminnolliset vaatimukset. Kassab, Daneva ja Ormandjieva (2007) mainitsevat, että ei-toiminnalliset vaatimukset ovat avaintekijä erottumisessa muista kilpailevista ohjelmistoista. Mikäli ei-toiminnalliset vaatimukset toteutetaan huolimattomasti, ne aiheuttavat viivästyksiä, kustannusten huomattavaa lisääntymistä ja projektien epäonnistumisia. Pohlin (1994) mukaan ei-toiminnalliset vaatimukset kuvaavat järjestelmän palveluiden rajoituksia.

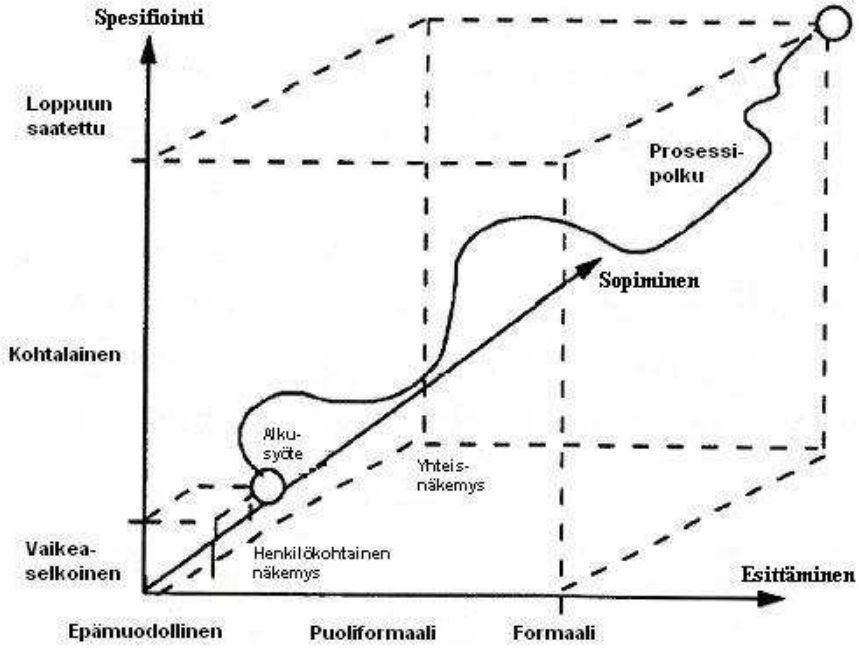
Ei-toiminnalliset vaatimukset voidaan jakaa edelleen alaluokkiin. van Lamsweerde (2009, 24) jakaa ei-toiminnalliset vaatimukset päätasolla neljään luokkaan: palvelun laatuvaatimukset, yhdenmukaisuusvaatimukset, arkkitehtuurivaatimukset sekä kehittämistä koskevat vaatimukset (Kuvio 1). *Laatuvaatimukset* ilmaisevat millaisia laatuun liittyviä ominaisuuksia suojauksen, turvallisuuden, luotettavuuden, suorituskyvyn ja rajapinnan osalta ohjelmistolla tulisi olla. *Yhdenmukaisuusvaatimukset* (engl. compliance requirements) kuvaavat lakien ja säännösten, sosiaalisten normien sekä kulttuuristen ja poliittisten seikkojen asettamia rajoitteita ohjelmistolle. *Arkkitehtuurivaatimukset* määräävät ohjelmiston rakenteelliset rajoitukset. *Kehitysvaatimukset* eivät määritä sitä, miten ohjelmisto täyttää asetetut toiminnalliset vaatimukset, vaan kuinka ohjelmisto tulisi toteuttaa. Tällaisia ovat vaatimukset toimitusaikatauluista, ominaisuuksien vaihtelevuudesta, ylläpidosta ja uudelleen käytöstä.

Toiminnallisten ja ei-toiminnallisten vaatimusten ero ei ole aina selkeä, vaan ne voivat olla osittain päällekkäisiä (van Lamsweerde, 2009). Asiakkaan esittämä, usein abstrakti ei-toiminnallinen vaatimus voi muuttua yksityiskohtaiseksi toiminnalliseksi vaatimukseksi, kun vaatimusta tarkennetaan riittävästi (Kotonya & Sommerville, 2002).

Pohl (1994) on kuvannut vaatimusmäärittelyprosessia kolmen ulottuvuuden avulla (Kuvio 2). Nämä ulottuvuudet ovat spesifiointi, esittäminen ja sopiminen. Näillä kaikilla ulottuvuuksilla on omat tavoitteensa vaatimusmäärittelyssä. *Spesifiointiulottuvuus* (engl. specification) kuvaa sitä kehitystä, joka tapahtuu siirryttäessä ohjelmistoa koskevista hyvin epämääräisistä käsityksistä kohti

KUVIO 1 Ei-toiminnallisten vaatimusten luokittelu (van Lamsweerde, 2009, 24)

tarkasti määriteltyjä (spesifioituja) vaatimuksia. Toisena ulottuvuutena on *esittämisulottuvuus* (engl. representation), joka kuvaa vaatimusten erilaisia esitysmuotoja epämuodollisista kuvauksista puoliformaaleihin ja formaaleihin. Epämuodolliset kuvaukset koostuvat normaalista puhekielestä, äänistä, animaatioista ja esimerkeistä. Tämän tyyppiset kuvaukset ovat hyvin ilmaisuvoimaisia. Puoliformaalit esittämistavat antavat selkeän yleiskatsauksen järjestelmästä, usein kaaviokuvina. Formaalit esitykset perustuvat kieleen, jonka syntaksi ja semantiikka ovat hyvin määriteltyjä.



KUVIO 2 Vaatimusmäärittelyprosessi kolmessa ulottuvuudessa (Pohl, 1994, 249)

Kolmantena ulottuvuutena on *sopiminen* (engl. agreement), joka kuvaa vaatimuksista vallitsevien käsitysten yhdenmukaisuusastetta eri osapuolten välillä. Vaatimusmäärittelyprosessia kuviossa kuvaa viiva, joka etenee alkutilannetta vastaavasta pisteestä kohti päätepistettä. (Pohl, 1994). Viivan mutkaisuus pyrkii osoittamaan sitä, että prosessi ei aina ole kovin suoraviivainen.

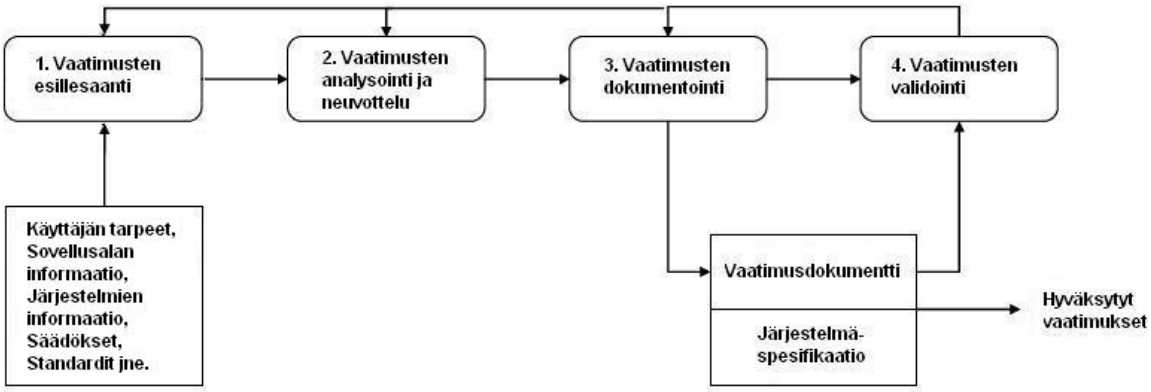
2.3 Vaatimusmäärittelyprosessi

Vaatimusmäärittelyä voidaan jäsentää monella tavalla, kuten karkealla (engl. coarse-grain) tai yksityiskohtaisella (engl. fine-grain) toimintamallilla, roolitoimintamallilla (engl. role-action) tai entiteetti-relaatiomallilla (engl. entity-relation) (Kotonya & Sommerville, 2002). Ensin mainittu malli kuvaa vaatimusmäärittelyprosessin periaatteet ja vaiheet ja niiden suhteet toisiinsa. Yksityiskohtainen toimintamalli kuvaa tiettyä prosessia tarkemmin. Mallia voidaan käyttää olemassa olevien prosessien ymmärtämiseksi ja parantamiseksi. Roolitoimintamalli kuvaa prosessissa mukana olevat ihmiset ja heidän tehtävänsä. Mallia voidaan myös hyödyntää prosessin ymmärtämisessä sekä automatisoinnissa. Viimeisenä mainittu entiteetti-relaatiomalli kuvaa prosessin syötteitä, tulosteita ja välituloksia. Tätä mallia voidaan käyttää esimerkiksi laadunhallinnassa (Kotonya & Sommerville, 2002).

Seuraavassa käytetään Kotonyan ja Sommervillen (2002) esittämää prosessimallia (Kuvio 3), joka on karkeantason toimintamalli. Sen mukaan vaatimusmäärittely koostuu neljästä vaiheesta, jotka ovat vaatimusten esillesaanti, vaatimusten analysointi ja neuvottelu, vaatimusten dokumentointi ja vaatimusten validointi. Jokainen vaihe käydään läpi useita kertoja prosessin aikana. Näiden vaiheiden lisäksi koko vaatimusmäärittelyprosessin aikana toimii vaatimusten hallinta, jonka avulla vaatimuksiin tehtyjä muutoksia voidaan kontrolloidusti hallita. Jokaista edellä mainittua vaihetta käsitellään tarkemmin seuraavissa alaluvuissa.

Vaatimusmäärittelyprosessi sisältää Kotonyan ja Sommervillen (2002, 36) mukaan erilaisia toimijoita, jotka vastaavat prosessin etenemisestä. Kullakin on prosessissa oma roolinsa, kuten projektijohtaja, järjestelmänkehittäjä jne. Prosessissa mukana olevien mielenkiinto järjestelmää kohtaan voi ulottua tietyn ongelman ratkaisemisesta tietyn toiminnon tukemiseen tai järjestelmän kehittämisen teknisen ongelman ratkaisuun.

Kotonya ja Sommerville (2002, 37) mainitsevat, että erilaiset sidosryhmät vaikuttavat vaatimusmäärittelyprosesseihin monella tavalla. Näihin sidosryhmiin kuuluvat esimerkiksi ohjelmistoalan ammattilaiset, järjestelmän käyttäjät, ulkoiset valvojat ja sovellusalueen asiantuntijat, jotka kaikki vaikuttavat prosessiin omalla panoksellaan. Tämä johtuu siitä, että erilaisen taustan omaavilla sidosryhmillä on erilainen tekninen tietämys järjestelmästä sekä erilaiset henkilökohtaiset tai organisatoriset tavoitteet. Osa sidosryhmistä on esimerkiksi organisaation eri osastoilta, ja he ajavat omia tavoitteitaan ottamatta huomioon muiden tavoitteita tai vaatimusmäärittelyprosessin kokonaisuutta.



KUVIO 3 Vaatimusmäärittelyprosessin toimintamalli (Kotonya & Sommerville, 2002, 32)

Poliittisella vaikuttamisella on siten oma merkityksensä vaatimusmäärittelyprosessissa. Organisaatio-, osasto- ja yksilötasolla voidaan vaikuttaa siihen, mitkä vaatimukset toteutetaan ja mitkä hylätään (vrt. sopiminen-ulottuvuus Kuviossa 3). Eri henkilöt yrittävät vaikuttaa järjestelmän vaatimuksiin oman poliittisen vaikutusvaltansa kasvattamiseksi tai ylläpitämiseksi (Kotonya & Sommerville, 2002, 38).

2.4 Esillesaanti

Oikeiden vaatimusten esillesaanti sidosryhmiltä on tärkeää projektin onnistumisen kannalta. *Esillesaantivaiheen* tavoitteena on asianomaisten todellisten tarpeiden ja vaatimusten selvittäminen. Esillesaannin helpottamiseksi on kehitetty erilaisia tekniikoita, joita voidaan hyödyntää vaatimusten selvittämiseksi. Tekniikoiden tarkoituksena on vähentää vaatimusten monitulkintaisuutta ja lisätä niiden selkeyttä (Chua, Bernardo, & Verner, 2010). Esillesaantivaiheen tekniikoita voidaan luokitella monella tavalla. Takats ja Brewer (2005) ovat luokitelleet tekniikat sen mukaan, mihin projektin vaiheisiin ne soveltuvat. Nuseibehin ja Eastbrookin (2000) tekemän jaotuksen perusteella tekniikat voidaan luokitella kuuteen ryhmään perustuen projektin tarpeiden mukaan (vrt. Taulukko 1). Nämä ryhmät ovat: 1. perinteiset tekniikat, 2. prototyypit, 3. ryhmätekniikka, 4. kontekstuaaliset tekniikat, 5. kognitiiviset tekniikat ja 6. mallilähtöiset tekniikat.

Ensimmäiseen ryhmään kuuluvat perinteiset tekniikat, joita ovat kyselyt, haastattelut ja dokumenttien analysointi. Toiseen ryhmään kuuluvat kertakäyttöiset (engl. throw-away) ja kehittyvät (engl. evolutionary) prototyypit, joita rakennetaan järjestelmän havainnollistamiseksi ja vaatimusten kuvailemisen helpottamiseksi. Kolmas ryhmä sisältää sidosryhmien ryhmätapaamisia, joissa vaatimuksia etsitään useilla erilaisilla menetelmillä. Tekniikan tarkoituksena on edistää esillesaatujen vaatimusten hyväksymistä ryhmädynamiikan avulla. Tällaisia menetelmiä ovat esimerkiksi aivoriihet, fokusryhmät (engl. focus groups),

TAULUKKO 1 Esillesaantitekniikoita (vrt. Tuunanen, 2005, 17)

Luokittelu	**Esimerkkejä tekniikoista**
Perinteinen tekniikka	Kysely, haastattelu, dokumenttianalyysi
Prototyyppitekniikka	Kertakäyttöinen ja kehittyvä prototyyppi
Ryhmätyöskentelytekniikka	Aivoriihi, fokusryhmä, RAD- ja JAD-ryhmät
Kontekstuaalinen tekniikka	Etnografia, keskusteluanalyysi, kontekstuaalinen kysely
Kognitiivinen tekniikka	Protokolla-analyysi, porrastaminen, korttienlajittelu
Mallilähtöinen tekniikka	Tavoitepohjainen ja skenaariopohjainen tekniikka, KAOS

RAD (engl. Rapid Application Development) ja JAD (engl. Joint Application Development) -työryhmät. (Tuunanen, 2005, 18)

Neljänteen ryhmään kuuluvat kontekstuaaliset tekniikat, joissa hyödynnetään etnografisia tekniikoita, etnologiaa ja keskusteluanalyysiä. Näiden tekniikoiden avulla etsitään vuorovaikutus- ja keskustelumalleja. Viidenteen ryhmään kuuluvat kognitiiviset tekniikat, jotka on alun alkaen kehitetty tietämyksen hankintaan. Tähän ryhmään kuuluvia tekniikoita ovat esimerkiksi protokolla-analyysi, missä asiantuntija selittää tarkkailijalle tehtävää sen suorituksen edetessä. Porrastamista (engl. laddering) käytetään sidosryhmien tietämyksen rakenteen ja sisällön esillesaamiseksi. Korttienlajittelu (engl. card sorting) -tekniikassa sidosryhmät lajittelevat kortteja sopiviin ryhmiin kohdealueittain.

Kuudes ryhmä sisältää mallilähtöisiä (engl. model-driven) tekniikoita, jotka käyttävät malleja vaatimusten esillesaamiseen. Tähän ryhmään kuuluvia tekniikoita on tavoitepohjainen (engl. goal-based methods), skenaariopohjainen ja KAOS -lähestymistapa (Tuunanen, 2005, 18).

2.5 Neuvottelu ja analyysi

Vaatimusten analyysi- ja neuvotteluvaiheessa pyritään löytämään mahdollisia ongelmakohtia järjestelmävaatimuksista sekä saavuttamaan yhteisymmärrys sidosryhmien välillä. Tämä vaihe on osittain päällekkäinen esillesaanti- ja validointivaiheiden kanssa. Neuvottelu ja analyysi ovat kalliita ja aikaa vieviä prosesseja, koska dokumenttien lukeminen vaatii huolellista pohdintaa sen suhteen, miten esitetyt vaatimukset ovat toteutettavissa. Neuvotteluissa täytyy huomioida organisatoriset ja poliittiset vaikutukset vaatimusten hyväksyttävyyteen (Kotonya & Sommerville, 2002, 77).

Neuvottelu- ja analyysivaiheeseen on kehitetty erilaisia tekniikoita, joista osa on listattu Taulukossa 2. Osa näistä tekniikoista soveltuu myös muihin vaatimusmäärittelyvaiheisiin, kuten esillesaanti-, dokumentointi- ja validointivaiheisiin. Kertakäyttöisen prototyypin tarkoituksena on selventää asiakkaalle kaikkein hankalimpia vaatimuksia ja niiden toimintaa. Toinen taulukossa esiintyvä prototyyppi on kehittyvä.

TAULUKKO 2 Neuvottelu- ja analyysitekniikoita (Jiang, Eberlain & Far, 2006, 120–121)

Tekniikka	**Soveltuvuusalue**
Kertakäyttöinen prototyyppi (engl. throw-away prototype)	Neuvottelu ja analyysi, esillesaanti, validointi
Kehittyvä prototyyppi (engl. evolutionary prototype)	Neuvottelu ja analyysi, esillesaanti, validointi
Näkökulmapohjainen lähestymistapa (engl. viewpoint-based approach)	Neuvottelu ja analyysi
Skenaariopohjainen lähestymistapa (engl. scenario-based approach)	Neuvottelu ja analyysi, esillesaanti, validointi, dokumentointi
Tavoitelähtöinen analyysi (engl. goal-oriented analysis)	Neuvottelu ja analyysi, esillesaanti, dokumentointi
Päätöstaulut (engl. decision tables)	Neuvottelu ja analyysi, dokumentointi, validointi
Tilakone (engl. state machine)	Neuvottelu ja analyysi, validointi, dokumentointi
Petri-verkot (engl. Petrinets)	Neuvottelu ja analyysi, validointi, dokumentointi
Analyyttinen hierarkiaprosessi (engl. analytic hierarchy process)	Neuvottelu ja analyysi
Korttienlajittelu (engl. card sorting)	Neuvottelu ja analyysi

Tämän tekniikan tarkoituksena on tarjota asiakkaalle toimiva järjestelmä vaatimusten toiminnan ymmärtämisen lisäämäksi (Kotonya & Sommerville, 2002, 74).

Näkökulmapohjaisessa lähestymistavassa vaatimuksia tarkastellaan useammasta näkökulmasta. Näkökulmina voidaan käyttää sidosryhmän näkökulmaa sekä organisaation ja sovellusalueen (engl. domain knowledge) näkökulmia (Kotonya & Sommerville, 2002, 171).

Skenaariot mahdollistavat kehitettävän järjestelmän kuvaamisen asiakkaalle erilaisten esimerkkien avulla. Esimerkit osoittavat, miten järjestelmä toimii oletetuissa tilanteissa. Skenaarioiden avulla järjestelmän toiminnasta voidaan neuvotella asiakkaiden kesken ja tehdä päätöksiä sopivista ominaisuuksista sekä toiminnoista. Skenaariot sopivat myös esillesaantivaiheeseen. (van Lamsweerde, 2009)

Tavoitelähtöisessä analyysissä tavoitteet tunnistetaan erilaisista informaatiolähteistä ja muutetaan vaatimuksiksi, jotka järjestelmän tulee täyttää. Analyysin aikana tavoitteita tarkennetaan ja niistä etsitään mahdollisia esteitä, jotka vaikeuttavat tavoitteisiin pääsemistä. Tavoitteet voivat liittyä toteuttamiseen tai ylläpitoon. (Lapouchnian, 2005)

Päätöstauluja käytetään monitulkintaisten ja monimutkaisten vaatimusten ”if-then”-ehtojen jäsentämiseksi. Lisäksi päätöstaulujen avulla voidaan tarkistaa ehtojen kattavuus ja toisteisuus laskemalla sarakkeet ennen taulujen sieventämistä. Päätöstauluja voidaan käyttää myös hyväksyntätestauksessa. (van Lamsweerde, 2009, 122–123)

Järjestelmälle esitettyjä vaatimuksia voidaan analysoida tilakoneen avulla. Kehitettävä järjestelmä jaetaan määriteltyihin tiloihin, jotka perustuvat esille

saatuihin vaatimuksiin. Tilakaaviolla kuvataan tilat ja niiden väliset siirtymät. Tilakaavion avulla asiakas ja kehittäjät saavat jäsentyneemmän kuvan järjestelmän toiminnasta. (Braude & Bernstein, 2011)

Petri-verkkoja voidaan käyttää vaatimusten analysointiin muiden muassa käyttötapausten puutteiden, monitulkintaisuuksien ja epäjohdonmukaisuuksien paljastamiseksi. Petri-verkkojen käyttöä auttaa niiden visuaalisen esitystapa. Lisäksi analyysin avuksi on olemassa työkaluja, jotka tukevat Petri-verkkoja. (Lee, Cha & Kwon, 1998)

Analyyttinen hierarkiaprosessi on joustava monikriteerinen analyysimenetelmä, jonka avulla voidaan tehdä päätöksiä. Vaatimukset asetetaan erilaisille tasoille hierarkiaksi. Korkeimmalla tasolla on päätösongelman kokonaistavoite, seuraavilla tasoilla on kokonaistavoitteeseen vaikuttavat tekijät ja alimmalla tasolla ovat ratkaisuvaihtoehdot. Hierarkia mahdollistaa kokonaiskuvan saamisen päätöksen tekemisestä ja siihen vaikuttavista tekijöistä. Tasoille asetettuja vaatimuksia vertaillaan pareittain, jonka jälkeen päätetään kumpi parista on sopivampi vaihtoehto päätösongelman ratkaisemiseksi. (Mohammad, Mustafa & Al-Bahar, 1991)

Korttienlajittelussa sidosryhmille annetaan kortteja, jotka kuvaavat tiettyä kohdealueen osaa, ominaisuutta tai toimintoa. Jokainen sidosryhmä lajittelee kortit mieleiseensä järjestykseen. Lajittelun jälkeen sidosryhmiä haastatellaan, millä perusteella kortit ovat juuri tässä järjestyksessä. Korttien järjestyksestä neuvotellaan yhteisymmärryksen löytämiseksi. (van Lamsweerde, 2009, 66)

2.6 Dokumentointi

Dokumentointivaiheen tarkoituksena on kirjata muistiin tavalla tai toisella esille saadut vaatimukset sidosryhmien (esim. asiakkaat, kehittäjät, johtajat) käyttöön. Dokumentoinnin tulosta kutsutaan vaatimusdokumentiksi (engl. requirements document), toiminnalliseksi spesifikaatioksi tai järjestelmän vaatimusspesifikaatioksi (Kotonya & Sommerville, 2002,15). Hull, Jackson ja Dick (2007) esittävät kaksi vaatimusta vaatimusdokumentille: dokumentin luettavuus ja vaatimusten prosessoitavuus (engl. processability). Ensin mainittu liittyy dokumentin organisointiin, jälkimmäinen vaatimusten ilmaisuun selkeästi ja täsmällisesti siten, että vaatimukset ovat jäljitettävissä.

Kotonyan ja Sommervillen (2002,15) mukaan vaatimusdokumentin tulisi sisältää ainakin seuraavat kohdat:

1. järjestelmältä vaaditut palvelut ja toiminnot
2. järjestelmän rajoitukset
3. yleiset ominaisuudet
4. muihin järjestelmiin kohdistuvat integrointimääritykset
5. järjestelmän sovellusalue
6. järjestelmän kehittämisprosessin rajoitukset.

Huolellisesti laadittu vaatimusdokumentti sisältää johdanto-osan, joka antaa yleiskatsauksen järjestelmästä. Sen tarkoituksena on kuvata myös järjestelmän tuki liiketoiminnalle. Kolmantena hyödyllisenä asiana on sanasto, joka määrittää dokumentissa käytetyn teknisen termistön. Sanasto on erityisen tärkeä sidosryhmille, koska sen avulla eri taustan omaavat henkilöt ymmärtävät käytetyt termit samalla tavalla, eikä väärinymmärryksiä pääse syntymään (Kotonya & Sommerville, 2002, 15). IEEE:n (1993) esittämän standardin mukaan vaatimusdokumentin tulisi sisältää edellä mainittujen kohtien lisäksi vaatimusten yksityiskohtaiset määritykset. Kotonyan ja Sommervillen (2002) mukaan vaatimusdokumenttistandardia voidaan soveltaa tilannekohtaisesti. Heidän mukaan joitakin dokumentin kohtia voi jättää pois ja uusia kohtia lisätä tarpeen mukaisesti.

Taulukossa 3 on esitetty hyvyyskriteereitä vaatimuksille (van Lamsweerde, 2009, 35). Näitä kriteereitä käytetään vaatimusdokumenttien arvioinnissa, ja ne määrittävät vaatimusmäärittelyprosessin tavoitteet. *Kattavuus* (engl. completeness) tarkoittaa, että vaatimukset kattavat uuden järjestelmän kaikki tunnistetut tarpeet ja tavoitteet. *Asianmukaisuus* (engl. pertinence) tarkoittaa, että kaikkien kirjattujen vaatimusten tulee liittyä selkeästi niihin tavoitteisiin, joita uuden järjestelmän (engl. system-to-be) halutaan täyttävän. *Asiaankuuluvuus* (engl. adequancy) korostaa, että vaatimusten tulee koskea todellisia tarpeita, mieluiten realistisia. *Yksiselitteisyys* (engl. unambiguity) on tärkeä kriteeri, jottei vaatimuksia voida tulkita monella tapaa. Vaatimusten *mitattavuus* (engl. measureability) on hyödyllinen analyysivaiheessa, jolloin analysoija voi vertailla erilaisia vaihtoehtoja keskenään. Vaatimusten täytyy olla myös *johdonmukaisia* (engl. consistency) ja kokonaisuutena yhteensopivia. *Soveltuvuus* (engl. feasibility) on tarpeellinen kriteeri budjetti-, aikataulu- ja teknologiarajoitusten suhteen. *Hyvä rakenne* (engl. good structuring) on vaatimusdokumentin ominaisuus, joka edellyttää, että dokumentointi on jäsennelty selkeästi. Dokumentaation tulisi olla *muokattavissa* (engl. modifiability) päivitysten ja laajennusten muodossa. Vaatimusten *jäljitettävyyttä* (engl. traceability) tarvitaan sen selvittämiseksi, milloin vaatimuksia on lisätty tai muokattu dokumentaatioon ja mitä seurausvaikutuksia niillä on (van Lamsweerde, 2009, 35).

Taulukossa 3 esitetään myös tyypillisiä heikkouksia tai virheitä, joita vaatimusten dokumentoinnin yhteydessä voi esiintyä. Vaikeimpia havaita ovat *vaatimusten pois jääminen* (engl. omission). Tällainen virhe voi tapahtua missä tahansa vaatimusten määrittelyvaiheessa. *Ristiriita* (engl. contradiction) vaatimusdokumentissa tarkoittaa, että jokin vaatimus on yhteensopimaton toisen vaatimuksen kanssa. *Asiaankuulumattomuus* (engl. inadequancy) esiintyy silloin, kun vaatimus ei kuvaa ominaisuutta riittävästi. *Monitulkintaisuus* (engl. ambiguity) aiheuttaa vaatimusten ymmärtämisen eri tavoin. *Kohinalla* (engl. noise) tarkoitetaan epäolennaisuuksia vaatimusdokumentissa. Vaatimus on silloin *hyödytön* (engl. unfeasibility), kun sitä ei voida budjetin, aikataulun tai kehitysalustan mukaisesti toteuttaa. Dokumentin rakenne on *heikko* (engl. poor structuring), mikäli vaatimukset on esitetty ilman mitään selkeää rakennetta. *Heikko muokattavuus* (engl. poor modifiability) ilmenee silloin, kun vaatimukseen

TAULUKKO 3 Vaatimusten hyvyyskriteereitä ja heikkouksia (van Lamsweerde, 2009, 35–37)

Hyvyyskriteereitä	**Tyypillisiä heikkouksia**
Kattavuus	Poisjääminen
Asianmukaisuus	Ristiriita
Asiaankuuluvuus	Asiaankuulumattomuus
Yksiselitteisyys	Monitulkintaisuus
Mitattavuus	Mittaamattomuus
Johdonmukainen	Kohina
Soveltuvuus	Hyödytön
Hyvä rakenne	Heikko rakenne
Muokattavuus	Heikko muokattavuus
Ymmärrettävyys	Vaikeaselkoisuus
Jäljitettävyys	Ylimääritelty

tehdyt muutokset edellyttävät laajamittaisia muutoksia myös muualla. *Vaikeaselkoisuus* (engl. opacity) esiintyy silloin, kun vaatimusten perustelut, riippuvuudet ja esittäjät jäävät epäselviksi. *Ylimääritelty* (engl. overspecification) vaatimus on kysymyksessä silloin, kun se ei liity ongelmaan vaan sen tekniseen ratkaisuun. Kaikkein vaarallisimpia virheitä ovat vaatimusten poisjääminen, ristiriidat, asiaankuulumattomuus, monitulkintaiset vaatimukset ja niiden mittaamattomuus. (van Lamsweerde, 2009, 37)

2.7 Validointi

Vaatimusten validoinnin tarkoituksena on paljastaa mahdolliset virheet ja epäjohdonmukaisuudet esillesaaduista vaatimuksista (Kotonya & Sommerville, 2002). Validointiprosessiin osallistuvat sidosryhmät, järjestelmäsuunnittelijat sekä vaatimusmäärittelyssä mukana olevat henkilöt. Itse prosessi on pitkäkestoinen, kestäen viikoista jopa kuukausiin riippuen järjestelmän laajuudesta ja monimutkaisuudesta. Prosessiin on kuitenkin syytä käyttää tarpeeksi aikaa mahdollisien virheiden poistamiseksi. Esiintyviä virheitä voi olla laatustandardien noudattamatta jättäminen, huonosti ilmaistut ja moniselitteiset vaatimukset sekä ristiriidat vaatimuksissa. Huolellisesti toteutettu validointi säästää huomattavasti kustannuksia, erityisesti siksi että myöhemmiltä korjauksilta vältytään (Kotonya & Sommerville, 2002).

Validointiprosessia varten on kehitetty useita tekniikoita, kuten vaatimusten katselmointi (engl. requirements review), prototyypit, mallivalidointi (engl. model validation), vaatimusten testaus, animointi, tarkastaminen (engl. inspection) ja omin sanoin kertominen. Yleisin validointitekniikka on vaatimusten katselmointi, jossa asianomaiset tarkistavat vaatimusdokumentin, keskustelevat löydetyistä ongelmista ja sopivat ongelmien ratkaisemisesta (Kotonya & Sommerville, 2002). Kertakäyttöistä prototyyppiä käytetään ristiriitaisten käsitysten ratkaisemiseksi kehittäjien ja asianomaisten välillä. Kehittyvää prototyyppiä

sovelletaan toimivan prototyypin kehittämiseksi ja parantamiseksi saadun palautteen perusteella (Raja, 2009).

Mallivalidoinnin tarkoituksena on demonstroida mallin ristiriidattomuus, osoittaa useamman järjestelmämallin sisäinen ja ulkoinen yhtenäisyys sekä osoittaa, että mallit kuvaavat asianomaisten todellisia vaatimuksia (Kotonya & Sommerville, 2002, 103). Vaatimustestauksessa tutkitaan, toimivatko toteutetut vaatimukset halutulla tavalla. Vaatimuksille tulisi olla mahdollista laatia testitapauksia kohtuullisen helposti. Jos näin ei ole, on se merkki ongelmallisista vaatimuksista (Kotonya & Sommerville, 2002, 106). Animoinnissa sidosryhmät voivat läpi käydä peräkkäisiä tapahtumia järjestelmälle annetun käyttäytymismallin mukaisesti. Järjestelmäntilan käyttäytymistä voidaan animoida vision mukaan ja läpi käydä vaihtoehtoiset tapahtumat (Uchitel, Chatley, Kramer & Magee, 2004).

Validointi voi olla ongelmallista, koska sidosryhmien näkemykset vaatimuksista voivat olla ristiriidassa toistensa kanssa. Ristiriitoja pyritään ratkaisemaan neuvotteluvaiheessa siten, että kunkin tavoitteet voitaisiin kuitenkin säilyttää (Nuseibeh & Eastbrook, 2000).

2.8 Vaatimusten hallinta

Vaatimusten hallinnan tarkoituksena on hallita ja seurata tehtyjä muutoksia järjestelmän vaatimuksiin, luoda kommunikaatioyhteys sidosryhmien ja kehittäjien välillä ja varmistaa oikeat vaatimukset järjestelmälle. Muutoksia vaatimuksiin aiheuttavat järjestelmän ympäristön vaihtelut, ulkoiset tekijät, huonosti analysoidut vaatimukset, asiakkaan ymmärryksen lisääntyminen, organisatoriset muutokset sekä virheelliset ja epäjohdonmukaiset vaatimukset (Kotonya & Sommerville, 2002; Wang & Lai, 2001).

Vaatimusten hallintaan kuuluvat oleellisina osina vaatimusten tunnistaminen, varastointi, muutoshallinta ja vaatimusten jäljityslinkistö. Jokaisella vaatimuksella täytyy olla tunnus, jolla vaatimuksen voi yksilöidä. Ilman tunnistamismahdollisuutta vaatimusten hallinta on mahdotonta. Tunnistamiseen on kehitetty tekniikoita, kuten dynaaminen uudelleen numerointi, tietokantatallennetunnistaminen ja symbolinen tunnistaminen. Vaatimusten varastointiin käytetään tietokantaa. Muutoshallinta varmistaa samanlaisten tietojen keräämisen jokaisesta ehdotetusta muutoksesta sekä muutoksen aiheuttamista hyödyistä ja kustannuksista. (Kotonya & Sommerville, 2002). Jäljityslinkistö on tärkeässä roolissa vaatimusten hallinnassa, koska se vaikuttaa siihen, kuinka helposti dokumentteja voidaan lukea, selata, kysellä ja muokata. Nuseibeh ja Eastbrook (2000) mainitsevat artikkelissaan, että jäljityslinkistön tulee mahdollistaa vaatimusten seuranta elinkaarta pitkin eteen- ja taaksepäin.

Vaatimusten vakaus (engl. requirements stability) riippuu siitä, mihin järjestelmän osaan vaatimus liittyy. Sovellusalueeseen ja järjestelmän perusolemukseen liittyvät vaatimukset ovat vakaampia ja vaihtelevat harvemmin. Vaa-

timusten vaihtelevuus on tyypillistä silloin, kun järjestelmän toteutus on sidoksissa tiettyyn ympäristöön ja tietylle asiakkaalle (Kotonya & Sommerville, 2002).

2.9 Yhteenveto

Tässä luvussa kuvailtiin perinteistä vaatimusmäärittelyä, sen sijoittumista ohjelmistokehitysprosessiin, sen yhteydessä esiintyviä vaatimuksia sekä vaatimusmäärittelyyn kuuluvia vaiheita.

Ohjelmistonkehitysprosessia tarkasteltiin yksinkertaisen vesiputousmallin avulla, joka jakaa prosessin viiteen vaiheeseen: analyysi, suunnittelu, toteutus, testaus ja ylläpito. Vaatimusmäärittely sisältyy ensimmäiseen vaiheeseen. Vaatimukset jaetaan toiminnallisiin ja ei-toiminnallisiin vaatimuksiin. Toiminnalliset vaatimukset kuvaavat ohjelmistolle tai järjestelmälle haluttuja toimintoja sekä ominaisuuksia, kun taas ei-toiminnalliset vaatimukset kuvaavat järjestelmän rajoituksia. Ei-toiminnallisille vaatimuksille esitettiin tarkempi luokittelu.

Vaatimusmäärittely koostuu neljästä vaiheesta (esillesaanti, analyysi ja neuvottelu, dokumentointi ja validointi) ja vaatimusten hallinnasta. Jokaista vaihetta ja sen tekniikoita kuvailtiin lyhyesti. Useimmat esitetyistä tekniikoista soveltuvat useampaan vaatimusmäärittelyn vaiheeseen. Vaatimusten hallintaan kuuluvat vaatimusten tunnistaminen, varastointi, muutoshallinta ja vaatimusten jäljityslinkistö.

3 KETTERÄ LÄHESTYMISTAPA

Tämän luvun tarkoituksena on muodostaa yleiskäsitys ketterästä lähestymistavasta. Luvussa kerrotaan ensin ketterästä lähestymistavasta pääasiassa perustuen Agile-manifestiin. Sen jälkeen kuvataan kahta yleisimmin käytettyä ketterää menetelmää, XP:tä ja Scrumia. Kummastakin tuodaan esille prosessimalli, käytänteet ja roolit.

3.1 Ketteryys ja Agile-manifesti

Ketteryys (engl. agility) on vaikeasti määriteltävissä oleva käsite (Qumer & Henderson-Sellers, 2006). Qumerin ja Henderson-Sellersin (2006) mukaan Dove (1997) mainitsee, että jokaisella on omanlaisensa näkemys siitä mitä ketteryys on. Yleisesti ottaen ketteryys on nopean (engl. quick) ja vikkelän (engl. nimble) liikkumisen ominaisuus tai kyky (Lyytinen & Rose, 2006, 183). Qumerin ja Henderson-Sellersin (2006) mukaan Wongin ja Whitmanin (1999) mielestä ketteryys mahdollistaa nopean reagoinnin odottamattomiin muutoksiin joustavasti, sopeutuvasti ja monipuolisesti. Qumer ja Henderson-Sellers (2006) määrittelevät ketteryyden kestäväksi käyttäytymiseksi, joka mahdollistaa odotettujen tai odottamattomien muutosten toteuttamisen nopeasti ja joustavasti. Muutokset tehdään yksinkertaisilla, taloudellisilla ja laadukkailla välineillä joustavassa ympäristössä. Ketteryys soveltaa edellisistä projekteista opittua tietämystä. Conboy (2009) on tutkinut syvällisesti ketteryyden käsitteellistä taustaa. Hän on päätynyt seuraavaan määritelmään: ”[Ketteryys tarkoittaa] kehittämismenetelmän jatkuvaa valmiutta nopeasti tai luontaisesti luoda muutosta, ennakoivasti tai reagoivasti tukea muutosta sekä oppia muutoksesta samalla, kun se lisää asiakkaan saamaa arvoa (tuottavuus, laatu ja yksinkertaisuus) yhteisten [menetelmä-]komponenttien ja ympäristösuhteiden avulla. ” (Conboy, 2009, 340)

Useimmiten viitattu käsitys ketteryydestä pohjautuu Agile-allianssin (2001) laatimaan manifestiin, joka sisältää neljä arvoa ja 12 periaatetta. Arvojen mukaisesti ketterässä kehittämisessä arvostetaan:

yksilöitä ja vuorovaikutusta enemmän kuin prosesseja ja työkaluja,

toimivaa ohjelmistoa enemmän kuin kattavaa dokumentointia,

yhteistyötä asiakkaan kanssa enemmän kuin sopimusneuvottelua, ja

muutoksiin reagoimista enemmän kuin suunnitelman seuraamista.

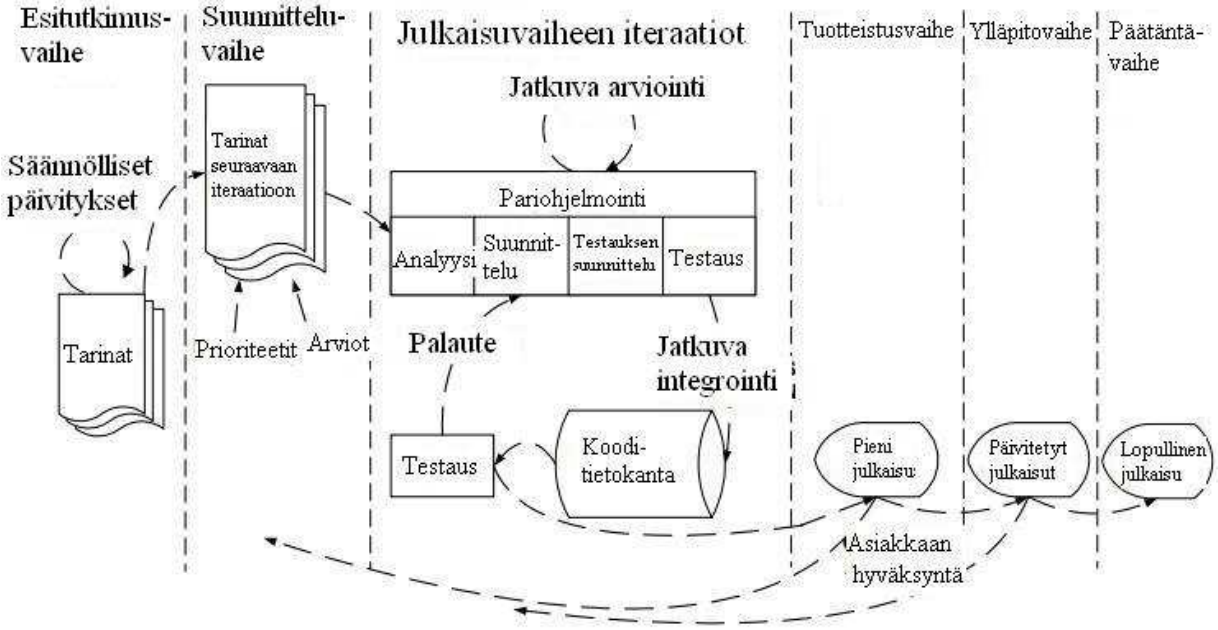
Agile-manifestissa (Agile Alliance, 2001) arvot on esitetty siten, että vasemmalla puolella esitetyt korostuvat. Ketterässä kehittämisessä arvostetaan yksilöitä ja vuorovaikutusta. Erityisesti tämä ilmenee tiimityöskentelyn muodossa. Keskeisenä tehtävänä ketterissä kehittämisessä on toimivan ohjelmiston varmistaminen jatkuvalla testauksella ja useilla pienillä julkaisuilla. Koodi pyritään pitämään mahdollisimman yksinkertaisena ja korkeatasoisena. Näin dokumentaation kirjoittamista voidaan huomattavasti vähentää koodin toimiessa dokumentaationa. Asiakkaan ja kehittäjien välillä tapahtuvaa kommunikointia suositaan paljon ketterässä kehittämisessä. Neuvotteluvaihe onkin hyvien suhteiden luomista asiakkaisiin ja niiden ylläpitämistä. Ketterässä kehittämisessä reagoidaan muutoksiin nopeasti. Kehittäjät ja asiakkaat on valtuutettu tekemään muutoksia kehitysprosessiin tarpeen niin vaatiessa. (Abrahamsson, Salo, Ronkainen & Warsta, 2002)

Kirjallisuudessa on esitelty lukuisia ketteriä menetelmiä. Näitä ovat muiden muassa XP (Beck, 1999), Scrum (Schwaber & Beedle, 2002), Crystal (Cockburn, 2002), Dynamic Systems Development Method (DSDM) (Stapleton, 1997), Feature Driven Development (FDD) (Palmer & Felsing, 2002), Lean software development (Poppendieck & Poppendieck, 2003), Mobile-DTM (Abrahamsson ym., 2004), Kanban (Kniberg & Skarin, 2009; Anderson 2010), Agile Modelling (Ambler, 2002), Agile Software Process Model (Ayoama, 1998), Internet-Speed Development (ISD) (Cusumano & Yoffie, 1999) ja Adaptive Software Development (ASD) (Highsmith, 2000). Seuraavaksi edellä mainituista menetelmistä tarkemmin kuvataan XP:tä ja Scrumia, koska ne ovat kaikkein tunnetuimmat ja käytetyimmät menetelmät. Lisäksi kirjallisuudessa on paljon käsitelty XP:n ja Scrumin käytänteitä.

3.2 XP

XP (engl. eXtreme Programming) (Beck, 1999; Beck & Andres, 2004) on eräs tunnetuimmista ketteristä menetelmistä, joka on tarkoitettu ohjelmistokehitykseen pienille ja keskisuurille tiimeille. XP soveltuu sellaisiin projekteihin, joissa vaatimukset ovat epämääräisiä ja nopeasti muuttuvia. (Abrahamsson ym., 2002)

XP-prosessi koostuu viidestä vaiheesta (Kuvio 4): esitutkimus (engl. exploration), suunnittelu (engl. planning), iteraatiot julkaisua varten (engl. iterations to release), tuotteistaminen (engl. productionizing), ylläpito (engl. maintenance) ja viimeisenä päätäntä (engl. death). (Abrahamsson ym., 2002)



KUVIO 4 XP:n elinkaari (Abrahamsson ym., 2002, 19).

Esitutkimusvaiheessa asiakkaat määrittävät vaatimukset ensimmäistä julkaisua varten. Vaatimukset kirjoitetaan XP:n käytänteiden mukaisesti korteille asiakkaan toimesta. Tiimin tehtävänä on tässä vaiheessa tutustua projektissa käytettäviin työkaluihin, käytänteisiin ja teknologiaan. Vaiheen kesto on parista viikosta muutamaan kuukauteen. (Abrahamsson ym., 2002)

Suunnitteluvaiheen tarkoituksena on käyttäjätarinoiden priorisointi ja ensimmäisen julkaisun sisällöstä päättäminen. Lisäksi ohjelmoijat arvioivat käyttäjätarinoiden vaatiman toteutusajan. Ohjelmoijien tehtävänä on myös aikataulun suunnittelu ja sen hyväksyminen. Suunnitteluvaiheen kesto on pari päivää. (Abrahamsson ym., 2002)

Edellisessä vaiheessa priorisoidut ja aikataulutetut tehtävät jaetaan *iteraatioihin julkaisua varten*. Ensimmäisessä iteraatiossa toteutetaan järjestelmän arkkitehtuuri, joka tehdään käyttäjätarinoiden pohjalta. Iteraatioissa toteutetaan asiakkaan valitsemat käyttäjätarinat. Jokaisen iteraation kesto on yhdestä neljään viikkoa. (Abrahamsson ym., 2002)

Tuotteistamisvaiheessa järjestelmä testataan ja tarkistetaan perusteellisesti ennen järjestelmän luovuttamista asiakkaalle. Viime hetkellä voi ilmetä vielä uusia vaatimuksia, jotka olisi tarpeellista olla mukana järjestelmässä. Näiden osalta päätetään, toteutetaanko vaatimukset nyt vai jätetäänkö ne seuraavaan julkaisuun. (Abrahamsson ym., 2002)

Ylläpitovaiheessa asiakkaalle luovutettua järjestelmää ylläpidetään ja tarjotaan asiakastukea. Lopuksi on *päätäntävaihe*, jolloin kaikki asiakkaan kirjoittamat käyttäjätarinat ovat toteutettu ja järjestelmä on asiakkaan toiveiden mukainen. Tässä vaiheessa kirjoitetaan tarpeellinen dokumentaatio, koska muutoksia

arkkitehtuuriin, suunnitteluun ja koodiin ei enää tehdä. Päätäntävaihe tulee eteen myös silloin, kun järjestelmä ei ole toiveiden mukainen tai järjestelmän jatkokehittäminen on liian kallista. (Abrahamsson ym., 2002)

XP:hen on sisällytetty useita erilaisia käytänteitä (Abrahamsson ym., 2002), joita ovat:

- *Suunnittelupeli* (engl. planning game), jossa ohjelmoijat arvioivat työmäärän asiakkaan valitsemien käyttäjätarinoiden toteuttamista varten. Asiakas päättää myös julkaisuajankohdan.
- *Pienet julkaisut* (engl. small releases). Järjestelmästä julkaistaan toimivia osia jopa päivittäin.
- *Metaforia* (engl. metaphor) käyttäen kuvataan, miten järjestelmä toimii asiakkaan ja tiimin mielestä.
- *Suunnittelu* pidetään mahdollisimman *yksinkertaisena* (engl. simple planning) toteutettavan järjestelmän osille.
- XP:ssä ohjelmistokehitys on *testauslähtöistä* (engl. test driven), joten testitapaukset tehdään ennen koodin kirjoittamista. Lisäksi testaus on jatkuvaa koko projektin ajan.
- *Refaktoroinnin* (engl. refactoring) avulla toisteiset järjestelmän osat poistetaan ja lisätään järjestelmän joustavuutta ja selkeyttä.
- *Pariohjelmoinnissa* (engl. pair programming) kaksi ohjelmoijaa kirjoittaa koodia yhdellä tietokoneella.
- *Yhteisomistajuus* (engl. collective ownership) tarkoittaa, että jokaisella tiimin jäsenellä on mahdollisuus muuttaa koodia.
- Uutta koodia lisätään järjestelmään päivittäin ja se pitää integroida. Tästä syystä *integrointi on jatkuvaa* (engl. continuous integration).
- *Asiakas on läsnä* (engl. on-site customer) kehittämässä järjestelmää projektin alusta loppuun asti.
- *Koodausstandardit* (engl. coding standards) ovat sääntöjä, joiden mukaan ohjelmointi toteutetaan yhdenmukaisesti.
- *Avoin työtila* (engl. open workspace) mahdollistaa tehokkaan kommunikoinnin kehittäjien välillä.
- *Vain säännöt* (engl. just rules). Tiimi muodostaa itse omat sääntönsä, joiden mukaan he toimivat. Sääntöjä voidaan kuitenkin muuttaa milloin vain.

Edellä mainituista käytänteistä tärkeimpiä ovat Abrahamssonin ym. (2002) mukaan lyhyet iteraatiot, useat julkaisut, jatkuva integrointi ja testaus, yhteisomistajuus, vähäinen dokumentaatio ja pariohjelmointi.

XP-menetelmä määrittää myös joukon rooleja. Näitä ovat: ohjelmoija, testaaja, asiakas, jäljittäjä (engl. tracker), valmentaja (engl. coach), konsultti ja johtaja (engl. manager). Jäljittäjän tehtävänä on antaa palautetta tiimille siitä, kuinka hyvin se on arvioinut iteraatioihin tarvittavat työpanoksen ja resurssit. Jäljittäjä seuraa myös iteraatioiden etenemistä ja arvioi asetetun päämäärän toteutumista. Valmentaja vastaa projektin kokonaisuudesta ja ohjaa tiimiä XP:n käy-

tänteiden oikein noudattamisessa. Konsultti on tiimin ulkopuolinen jäsen, joka auttaa tiimiä mahdollisten ongelmien ratkaisemisessa. (Abrahamsson ym., 2002)

3.3 Scrum

Toinen tunnetuista ketteristä menetelmistä on Scrum (Schwaber & Beedle 2002; Schwaber & Sutherland 2010), joka on iteratiivinen ja inkrementaalinen kehitysmenetelmä monimutkaisille projekteille, tuotteille ja sovelluksille. Scrum-menetelmä mahdollistaa tuotteiden kehittämisen ja monimutkaisten ongelmien ratkaisemisen luovasti ja lisäarvoa tuottavasti.

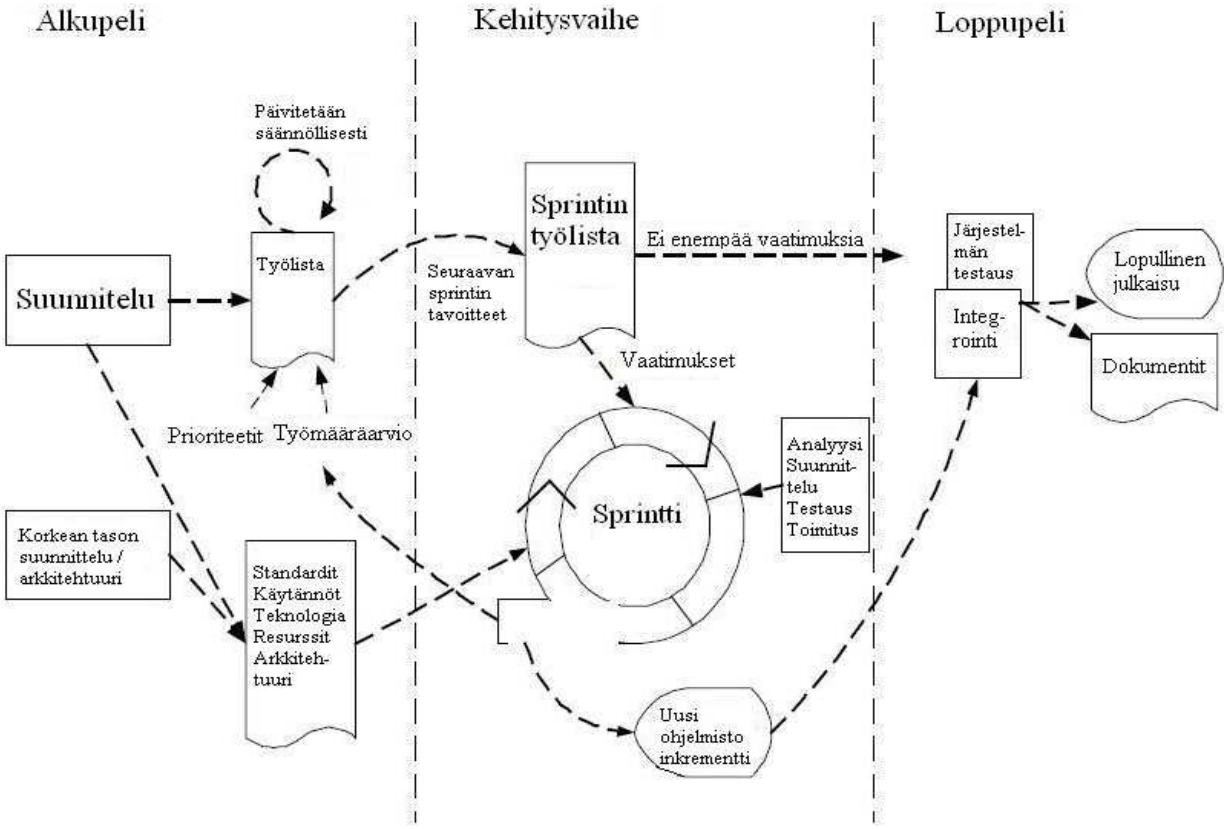
Scrum-menetelmässä prosessi etenee kolmessa vaiheessa (Kuvio 5), jotka ovat esipeli (engl. pregame), kehitysvaihe (engl. development) ja loppupeli (engl. postgame). *Esipelin* tarkoituksena on tehdä suunnitelmat projektille sekä arkkitehtuurille. Suunnitelmat kattavat kehitystiimin ja työkalujen määritykset, tarvittavien resurssien arvioinnin ja vahvistuksen johdolta projektin toteuttamista varten. Arkkitehtuuri ja järjestelmä suunnitellaan korkealla tasolla työlistan sisällön perusteella. Vaatimukset työlistaan kerätään mm. asiakkailta, kehittäjiltä, myynti- ja markkinointiosaston henkilöiltä. (Abrahamsson ym., 2002)

Kehitysvaiheessa järjestelmä kehitetään sprinteissä. Tälle vaiheelle on tyypillistä, että muutoksia jopa odotetaan, jotta niihin voidaan reagoida nopeasti. Scrumin käytänteiden avulla odotettuja muutoksia seurataan ja valvotaan hallitusti sprintin ajan. Muutoksia aiheuttavat erilaiset teknologiset ja ympäristölliset tekijät, kuten aikataulut, vaatimukset, laatu, työkalut jne. (Abrahamsson ym., 2002)

Loppupelivaiheessa työlistassa olleet vaatimukset on toteutettu ja järjestelmä tai sen osa on valmis julkaistavaksi. Muita tehtäviä loppupelivaiheessa ovat integrointi, järjestelmätestaus ja dokumentaation kirjoittaminen. (Abrahamsson ym., 2002)

Myös Scrum sisältää erilaisia käytänteitä, joiden tarkoituksena on taata säännönmukaisuus. Scrumin käytänteistä keskeisin on *sprintti* (engl. sprint), joka on lyhyt, noin 2–-4 viikkoa kestävä kehitysiteraatio. Jokaisen sprintin aikana tuotetaan käytettävä ja julkaisukelpoinen versio tuotteesta. Koska sprintin pituus on enintään kuukausi, mahdolliset riskit rajoittuvat vain yhden sprintin ajaksi, eikä koko projektin ajalle. Jokaisen kehityskierroksen alussa tiimi valitsee priorisoidusta *tuotteen työlistasta* (engl. product backlog) tehtävät, jotka toteutetaan sprintin aikana. Työlista sisältää kaikki tuotteelle halutut ominaisuudet, toiminnot, vaatimukset, korjaukset ja parannukset. Tehtävät (engl. task) ovat asiakkaalta esillesaatuja vaatimuksia. Ennen sprintin aloittamista *pidetään sprintin suunnittelutapaaminen* (engl. sprint planning meeting), jossa tuoteomistaja esittelee priorisoidun työlistan tiimille. Tiimi ja tuoteomistaja päättävät yhdessä, kuinka paljon työlistan tehtäviä yhteen sprinttiin sisällytetään.

Suunnittelupalaverin kesto kuukauden pituiselle sprintille on enintään kahdeksan tuntia. Tämän jälkeen tiimi tekee suunnitelmat, miten tehtävät toteutetaan. *Sprintin työlistassa* (engl. sprint backlog) ovat ne tehtävät, jotka on



KUVIO 5 Scrum -prosessin kolme vaihetta (Abrahamsson ym., 2002, 28).

tarkoitus tehdä sprintin aikana. Scrumiin kuuluu tärkeänä osana *päivittäiset tapaamiset* (engl. daily meetings), joissa keskustellaan ja seurataan projektin etenemistä noin 15 minuutin ajan. Tapaamisessa myös suunnitellaan mitä on tarkoitus tehdä seuraavan 24 tunnin aikana. Tehtävien valmistumista sprintin aikana seurataan *edistymiskäyrän avulla* (engl. sprint burndown chart). Tiimi päivittää käyrän edistymistä päivittäisten tapaamisten yhteydessä. Työlistasta ilmenee tehtävien arvioitu toteutusaika ja vastuuhenkilöt. Kun sprintti on päättynyt, tiimi ja sidosryhmät pitävät enintään neljä tuntia kestävän *katselmointipalaverin* (engl. sprint review meeting), jossa arvioidaan sprintin tuloksia. Palaverin aikana sidosryhmät antavat palautetta, jota tiimi voi hyödyntää seuraavassa sprintissä. Lisäksi katselmointipalaverissa tarkistetaan työlista seuraavalle sprintille. Ennen uuden sprintin aloittamista Scrumissa on tapana pitää kolmen tunnin pituinen sprintin *retrospektiivi* (engl. sprint retrospective meeting), jossa Scrum-mestarin johdolla keskustellaan siitä, mitä parannuksia voidaan tehdä, jotta seuraava sprintti sujuisi paremmin tai tuottavammin. Tärkeää Scrumissa on se, että jokaisen sprintin jälkeen tiimillä on esittää toimiva osa ohjelmistoa, joka on testattu, integroitu sekä mahdollisesti valmis markkinoille. (Schwaber & Sutherland, 2010; Sutherland & Schwaber, 2011)

Toimiakseen parhaalla mahdollisella tavalla Scrum vaatii erilaisia rooleja. Näitä rooleja ovat: tuoteomistaja (engl. product owner), tiimi (engl. team) sekä

Scrum-mestari (engl. Scrum Master). *Tuoteomistajan* vastuulla on saada kehitettävälle tuotteelle mahdollisimman korkea investoinnin tuottoprosentti (ROI) valitsemalla vaatimukset, joilla on korkea liiketoiminta-arvo ja pienet kustannukset. Tuoteomistaja priorisoi työlistan jokaiselle sprintille aina uudestaan. Lisäksi tuoteomistaja on vastuussa työn laadusta. Usein ohjelmistoprojekteissa asiakas ja tuoteomistaja ovat sama henkilö. (Sutherland & Schwaber, 2011)

Tiimin tehtävänä on toteuttaa ohjelmisto tuoteomistajan toiveiden mukaisesti. Tiimi voi myös antaa ehdotuksia ja ideoita tuoteomistajalle kehitettävään ohjelmistoon. Tiimi on itseorganisoituva, itseohjautuva ja monitaitoinen. Monitaitoinen tiimi on tarpeellinen, koska sen vastuualueisiin kuuluu ohjelmoinnin lisäksi myös muita tehtäviä, kuten testaus ja liiketoiminta-analyysit. Lisäksi tiimillä on itsehallinto ja täysi vastuu ohjelmistosta. Jäsenten määrä tiimissä on vähintään kolme ja enintään yhdeksän. Liian pieni tiimi vähentää vuorovaikutusta ja voi olla osaamiseltaan riittämätön. Liian suuri tiimi taas vaatii paljon koordinointia ja aiheuttaa kehitysprosessin hajanaisuutta. (Sutherland & Schwaber, 2011; Schwaber & Sutherland, 2010)

Scrum-mestarin tehtävänä on auttaa tiimiä, tuoteomistajaa ja organisaatiota parhaalla mahdollisella tavalla, jotta projekti onnistuisi. Scrum-mestari vastaa siitä, että tiimi ja tuoteomistaja pystyvät tekemään työnsä ilman häiriötä. Tiimiä Scrum-mestari ohjaa moniosaamiseen ja itseohjautuvuuteen. Scrum-mestari vastaa Scrumin opettamisesta tiimille, jos sen osaaminen on puutteellista tai Scrumin käytänteet väärin ymmärretty. Lisäksi Scrum-mestari suunnittelee, ohjaa ja kouluttaa Scrumin oikeaan käyttämiseen organisaatiossa. Scrum-mestari myös varmistaa, että jokainen organisaation työntekijä on perillä kaikista Scrumin käytänteistä ja siitä, että käytänteitä myös noudatetaan. Scrum-mestari keskustelee myös muiden Scrum-mestareiden kanssa, miten parantaa Scrumin käyttöä organisaatiossa. Huomion arvoisena asiana on, ettei Scrum-mestari ole tiimin päällikkö eikä projektipäällikkö. (Sutherland & Schwaber, 2011; Schwaber & Sutherland, 2010)

4 VERTAILEVA YLEISKUVAUS KETTERÄSTÄ KEHITTÄMISESTÄ JA VAATIMUSMÄÄRITTELYSTÄ

Luvussa 2 kuvailtiin järjestelmäkehitystä ja vaatimusmäärittelyä siten kuin ne perinteisen lähestymistavan mukaisessa kehittämisessä nähdään. Tässä luvussa on tarkoitus kuvata ketterää kehittämistä ja ketterää vaatimusmäärittelyä suhteessa perinteiseen lähestymistapaan. Ensimmäisessä alaluvussa vertaillaan perinteistä ja ketterää lähestymistapaa yleisellä tasolla. Toisessa alaluvussa tarkastellaan lähestymistapojen välisiä eroja erityisesti vaatimusmäärittelyn osalta. Vertailun tarkoituksena on muodostaa yleiskuva ketterästä vaatimusmäärittelystä, jota seuraavassa luvussa tarkennetaan.

4.1 Yleisesti perinteisen ja ketterän lähestymistavan eroista

Perinteisen ja ketterän kehittämisen välillä on paljon samanlaisuuksia mutta myös oleellisia eroavaisuuksia. Tässä alaluvussa esitetään lyhyt yhteenveto lähestymistapojen eroista perustuen lähinnä Boehmin (2002), Leffingwellin (2007), Overhagen, Schlaudererin ja Birkmeierin (2011) ja Conboyn, Coylen, Xiaofengin ja Pikkaraisen (2011) tutkimuksiin. Eroja tarkastellaan seuraavan jäsennyksen mukaisesti: projektin johtaminen, yleinen toimintatapa, kehittäjät, asiakkaat, suunnittelu, arkkitehtuuri, toteuttaminen, testaus ja integrointi (Taulukko 4).

Nerurin, Mahapatran ja Mangalarajin (2005) mukaan projektin johtaminen ja hallinta ketterässä lähestymistavassa ovat yhteistyö- ja ihmispainotteisia. Projektin johtamisesta vastaavan henkilön (Scrum-menetelmän mukaan Scrum-mestarin) tehtävänä onkin helpottaa projektin etenemistä mahdollisimman paljon. Scrum-menetelmässä projektin kontrolloimiseksi käytetään edistymiskäyrää (engl. burndown charts), josta selviävät jäljellä olevat tehtävämäärät (Overhage ym., 2011).

TAULUKKO 4 Ketterän ja perinteisen lähestymistavan vertailu (vrt. Boehm, 2002, 68).

Tarkastelun kohde	**Ketterä lähestymistapa**	**Perinteinen lähestymistapa**
Projektin johtaminen	Yhteistyö- ja ihmispainotteinen.	Prosessipainotteinen.
Yleinen toimintatapa	Inkrementaalinen ja iteratiivinen.	Vaiheittainen.
Kehittäjät	Ketteriä, osaavia, lähekkäin työskenteleviä ja yhteistyökykyisiä.	Suunnittelupainotteinen, riittävät taidot, pääsy ulkoiseen tietämykseen.
Asiakkaat	Omistautuneita, asiantuntevia, lähekkäin työskenteleviä, yhteistyökykyisiä, edustavia, valtuutettuja, kriittisiä ja jatkuvasti läsnä olevia.	Pääsy tietämykseen, yhteistyökykyisiä, edustavia, valtuutettuja, yleensä mukana vain vaatimusmäärittelyssä.
Projektin suunnittelu	Jatkuvaa.	Etukäteen suoritettavaa.
Arkkitehtuuri	Suunniteltu nykyisille vaatimuksille.	Suunniteltu tiedossa oleville sekä ennakoitaville vaatimuksille.
Toteuttaminen	Tärkeät ominaisuudet ensin.	Kaikki ominaisuudet ja toiminnot yhtä tärkeitä.
Testaus ja integrointi	Jatkuvaa.	Yleensä projektin lopussa.

Perinteisessä lähestymistavassa johtamistapa perustuu pitkälle optimoitujen sekä toistettavien prosessien tunnistamiseen. Ohjelmistojen kehittämisen suunnittelu ja johtaminen toteutetaan käskyihin ja kontrollointiin (engl. control and command) perustuvalla johtamistyylillä. (Overhage ym., 2011)

Projektin hallitsemiseksi tiimijäsenet raportoivat yleensä tilaraporteilla tai välietappien yhteydessä, kuinka paljon järjestelmää on toteutettu (Overhage ym., 2011).

Ketterälle kehittämiselle on tyypillistä, että ohjelmistoa rakennetaan inkrementaalisesti ja iteratiivisesti. Tämä tarkoittaa sitä, että suunnitelmia, ainakin tarkemmalla tasolla, tehdään osa kerrallaan ja suunnitelmaa seuraa aina kyseisen osan toteutus. Vaiheita iteroidaan runsaasti, jolloin ollaan siis hyvin kaukana perinteisen lähestymistavan vaiheittaisesta toimintatavasta, jossa äärimmäisessä tapauksessa vaiheet toteutetaan peräkkäin (Leffingwell, 2007).

Ketterässä lähestymistavassa kehittäjien asiantuntemus ja taidot ovat tärkeässä roolissa. Conboyn ym. (2011) mukaan kehittäjillä tulisi olla useita taitoja, koska henkilöt toimivat useammassa roolissa kehitystiimissä. Kehitystiimi on itseohjautuva, mikä myös kannustaa roolien vaihtamiseen tiimissä. Tiimin jäsenten tulisi olla yhtä aikaa ohjelmoijia, testaajia, arkkitehteja, asiakkaita, laadun varmistajia ja melkeinpä kaikkea mahdollista, mitä ohjelmistojen tekemiseen voi liittyä. Ketterä lähestymistapa vaatii sopivien henkilöiden löytämisen tiimiin, jotta käytetty menetelmä (esim. Scrum) toimisi mahdollisimman tehokkaasti. Boehm (2002) mainitsee, että ilman osaavia kehittäjiä esimerkiksi suunnitteluvaiheesta voi tulla enemmän tai vähemmän sekava. Toiminnan ketteryys riippuu pikemminkin tiimin jäsenten hiljaisesta tiedosta ja tietämyksen laajuudesta kuin tiedon tarkasta ja kattavasta dokumentoinnista. Nerurin ym. (2005) mukaan hiljaisen tiedon jakaminen onnistuu tiimin jäsenten roolivaihtelun ja

vuorovaikutuksen kautta. Tällä tavoin tieto siirtyy tiimissä eteenpäin, eikä dokumentointi ole välttämätöntä.

Perinteisessä lähestymistavassa kehittäjät voivat olla ammattitaidoiltaan ja osaamiseltaan eritasoisia. Roolit ovat enemmän yksilökeskeisiä, ja kehittäjä voi parantaa osaamistaan omassa tietyssä tehtävässään. Kommunikointi ja tiedon välittäminen tapahtuu kattavan ja formaalin dokumentoinnin kautta (Boehm & Turner, 2003; Nerur ym., 2005).

Asiakkaan rooli ketterässä kehittämisessä on erittäin tärkeä (Conboy ym., 2011). Boehm (2002) toteaa, että ketterän kehittämisen onnistumisen kannalta asiakkaan tulisi olla motivoitunut ja omistautunut työskentelemään kehitystiimin kanssa. Esimerkiksi Scrum-menetelmässä tehdään tiivistä yhteistyötä asiakkaan kanssa koko projektin ajan (Overhagen ym., 2011). Asiakas päättää kehitystiimin kanssa, mitä ominaisuuksia ja vaatimuksia missäkin iteraatiossa toteutetaan. Usein asiakkaan osallistuminen kehittämisprojektiin on käytännössä liian vähäistä, muiden työtehtävien ja kiireiden vuoksi. Perinteisessä lähestymistavassa asiakasta tarvitaan lähinnä vaatimusmäärittelyn yhteydessä. Molemmissa tapauksissa asiakkaat ovat valtuutettuja tekemään päätöksiä. He ovat omistautuneita ja yhteistyökykyisiä. Lisäksi asiakkailla tulisi olla riittävä tietämys, että he pystyvät vastaamaan kehittäjien esittämiin kysymyksiin.

Yksityiskohtainen suunnittelu ketterässä kehittämisessä viimeistellään ennen ohjelmoinnin aloittamista, juuri ajoissa (engl. just-in-time). Näin suunnitelmaa ei tarvitse muokata useita kertoja vaatimusten muuttuessa. Suunnittelua tehdään kahdella tasolla, yleisellä ja yksityiskohtaisella. Yleisen tason suunnittelu keskittyy ohjelmiston julkaisuihin. Yksityiskohtaisella tasolla keskitytään iteraatioiden suunnitteluun. Suunnittelua tehdään aina jokaista iteraatiota ennen ja ohjelmiston julkaisun jälkeen (Leffingwell, 2007). Overhagen ym. (2011) mainitsevat, että esimerkiksi Scrum-menetelmässä suunnittelu toteutetaan ja sitä hallitaan iteraatiokohtaisesti erilaisia strategiatasoja käyttäen.

Perinteisessä lähestymistavassa suunnittelu tehdään Leffingwellin (2007) mukaan yksityiskohtaisesti ja tarkasti aikaisessa vaiheessa projektia. Overhagen ym. (2011) mukaan suunnitteluprosessi on etukäteen hallinnoitu ja suunniteltu, ja se toteutetaan jakamalla tehtävät osiin sekä käyttämällä virstanpylväitä. Perinteisessä lähestymistavassa pyritään tekemään ensin suunnitelmat melkein valmiiksi siten, että niitä voivat arvioida ulkopuoliset arvioijat. Boehm (2002) mainitsee, että perinteisen lähestymistavan suunnitteluun liittyy riskejä, mikäli kehitettävän sovellukseen joudutaan tekemään nopeita muutoksia. Tämän vuoksi suunnitelmat voivat vanhentua, jolloin niitä ei voi enää käyttää tai muutoksien aiheuttamat ylläpitokustannukset voivat kasvaa suuriksi.

Lähestymistapojen ero näkyy myös arkkitehtuurin suunnittelussa. Ketterässä lähestymistavassa arkkitehtuurin suunnittelu kehittyy projektin edetessä, kun taas perinteisessä lähestymistavassa arkkitehtuuri suunnitellaan etukäteen ja dokumentoidaan tarkasti projektin alkuvaiheissa (Leffingwell, 2007).

Ohjelmoinnin osalta ketterässä kehittämisessä koko tiimi keskittyy ensin tärkeimpien ominaisuuksien ja toimintojen toteuttamiseen. Kun järjestelmälle tärkeimmät ominaisuudet ovat valmiit, voidaan vaatimusten mukaisesti lisätä

vähemmän tärkeitä ominaisuuksia ja toimintoja. Perinteisessä lähestymistavassa tämän tapaista ominaisuuksien ja toimintojen luokittelua ei ole (Leffingwell, 2007).

Koodin testaus ja integrointi ovat ketterässä kehittämisessä jatkuvaa, minkä johdosta vähitellen valmistuva koodi on toimivaa. Yksikkö- ja hyväksyntätestaus suoritetaan kehittäjien toimesta ensimmäiseksi tai samanaikaisesti koodin kirjoittamisen yhteydessä. Testaus on mahdollisuuksien mukaan pitkälle automatisoitu. Asiakkaiden tehtävänä on kirjoittaa testitapauksia. Kehittäjät auttavat testitapausten kirjoittamisessa, mikä nopeuttaa testausta ja parantaa koodin laatua. Perinteisessä lähestymistavassa testauksen suorittavat erilliset testaustiimit pääasiassa projektin lopussa. Koodi on yleensä testaamatonta ja suurissa palasissa. Testausautomaatio on käytettävissä, ja integrointi tehdään vasta myöhemmässä vaiheessa (Leffingwell, 2007, 79).

Yhteenvetona vertailusta voidaan Nerurin ym. (2005) mukaisesti todeta, että ketterässä lähestymistavassa on perusajatuksena luottamus ihmisiin ja heidän luovuuteensa, ei niinkään prosesseihin, kuten perinteisessä lähestymistavassa on asianlaita. Ketterässä lähestymistavassa projekti pilkotaan pienempiin osiin (iteraatioihin XP.ssä, sprintteihin Scrumissa), jotka sisältävät suunnittelua, kehittämistä, integrointia, testausta ja tuotteen jakelua. Perinteisessä lähestymistavassa projekti määritellään tarkasti ja toteutetaan täsmällisen ja kattavan suunnittelun avulla. Perinteinen lähestymistapa on prosessikeskeinen, jonka keskeisenä ajatuksena on, että mahdolliset häiriötekijät voidaan tunnistaa ja poistaa jatkuvalla prosessien mittaamisella ja parantamisella.

4.2 Perinteisen ja ketterän vaatimusmäärittelyn vertailu

Seuraavaksi verrataan tarkemmin perinteistä ja ketterää lähestymistapaa erityisesti vaatimusmäärittelyn osalta. Vertailu tehdään ensin vaihejaon mukaan ja sen jälkeen vaatimusten kehittymisen suhteen.

4.2.1 Vaihejakoon perustuva vertailu

Perinteisen ja ketterän vaatimusmäärittelyn eroja voidaan tarkastella vaatimusmäärittelyn vaiheiden mukaisesti. Luvussa 2 esitettiin vaiheiksi vaatimusten esillesaanti, neuvottelu ja analyysi, dokumentointi ja validointi. Taulukossa 5 on esitetty yhteenveto vaihekohtaisesta vertailusta Rameshin ym. (2007) mukaisesti.

Perinteisen vaatimusmäärittelyn vaiheet esillesaannista validointiin suoritetaan suuressa määrin peräkkäisinä prosesseina. Ketterässä vaatimusmäärittelyssä vaiheet sulautuvat taas toisiinsa, eikä niillä ole selkeitä rajoja kuten perinteisessä vaatimusmäärittelyssä. Kaikki vaiheet toistuvat yhä uudelleen jokaisen iteraation yhteydessä projektin päättämiseen saakka. (Ramesh ym., 2007)

TAULUKKO 5 Perinteisen ja ketterän vaatimusmäärittelyn vaiheiden eroja (Ramesh ym., 2007, 469).

Vaihe	Perinteinen vaatimusmäärittely	Ketterä vaatimusmäärittely
Esillesaanti	Tehdään projektin alussa, ennen kehittämisen aloittamista. Vaatimukset ovat pysyviä eikä muutoksia tehdä.	Iteratiivinen esillesaantiprosessi. Kasvokkain tapahtuva keskustelu asiakkaan ja kehittäjien välillä. Vaatimukset muuttuvat projektin edetessä.
Neuvottelu ja analyysi	Keskittyy usein ristiriitojen ratkaisemiseen ja eriävien mielipiteiden sovitteluun.	Korkean tason vaatimusten analysointi projektin alussa. Analysointi ja suunnittelu ovat jatkuvaa. Kasvokkain käytävä neuvottelu. Priorisointi huomion kohteena.
Dokumentointi	Formaalia dokumentointia on runsaasti. Tieto siirtyy dokumenttien välityksellä.	Vaatimusten dokumentointia vähän tai ei ollenkaan, epäformaalit käyttäjätarinat tai lista ominaisuuksista yleisin dokumentointitapa. Tieto siirtyy kokouksissa.
Validointi	Pitkäkestoinen prosessi. Validoidaan vaatimusdokumentaation johdonmukaisuus ja kattavuus.	Useita arviointipalavereita sidosryhmien kanssa. Validointi tapahtuu kommunikoiden asiakkaan ja kehittäjien välillä.

Tärkeässä roolissa ketterässä vaatimusmäärittelyssä ovat kommunikointi asiakkaan ja kehittäjien välillä sekä iteratiivisuus. Perinteisessä vaatimusmäärittelyssä esillesaantivaiheessa pyritään löytämään kaikki vaatimukset ennen kehitystyön aloittamista. (Ramesh ym., 2007).

Neuvottelu- ja analyysivaiheessa keskitytään sidosryhmien eriävien mielipiteiden ja vaatimusten välisten ristiriitojen ratkaisemiseen. Dokumentointi on formaalia ja sitä tuotetaan runsaasti. Tämän vuoksi tieto siirtyy kehitystiimin sisällä dokumentoinnin kautta. Validointi on usein pitkäkestoista, sen aikana pyritään varmistamaan vaatimusdokumentaation johdonmukaisuus ja kattavuus (Ramesh ym., 2007).

Ketterässä vaatimusmäärittelyssä vaatimukset muuttuvat projektin aikana, minkä vuoksi esillesaantivaiheeseen palataan toistuvasti (Overhage ym., 2011). Vaatimukset ketterässä lähestymistavassa ovat priorisoituja, vapaamuotoisia käyttäjätarinoita (engl. user stories) ja testitapauksia. Ketteristä menetelmistä esimerkiksi Scrum-menetelmässä tiimi työskentelee yhdessä asiakkaan kanssa vaatimusten tunnistamisessa ja vaatimusten priorisoinnissa. Tunnistetut vaatimukset kerätään priorisoituun listaan (tuotteen työlista). Tähän listaan voidaan lisätä uusia vaatimuksia tai muuttaa niiden järjestystä (Overhage ym., 2011). Uusia vaatimuksia voidaan tunnistaa jokaisen ohjelmiston osan julkistamisen jälkeen. Kehitystiimi ja asiakas keskustelevat ohjelmistosta ja arvioivat sen ominaisuuksia ja toimintoja. Keskustelun aikana asiakas antaa palautetta ohjelmistosta ja esittää uusia vaatimuksia. Esille saadut vaatimukset toteutetaan seuraavassa iteraatiossa (Overhage ym., 2011).

Perinteisessä vaatimusmäärittelyssä vaatimukset ovat tarkasti dokumentoituja, formaaleja ja ennalta määriteltyjä. Jos uusia vaatimuksia joudutaan lisäämään tai muokkaamaan, muutokset vaatimuksiin täytyy tehdä muutoshallinnan kautta (Boehm & Turner, 2003; Overhage ym., 2011). Perinteinen lähestymistapa toimii parhainten silloin, kun vaatimukset pysyvät vakaina ja niiden vaihtuvuus on pientä. Mitä suurempi on vaatimusten vaihtuvuus, sitä enemmän vaikeuksia voi esiintyä. Vaatimusten vaihtuvuus vaikuttaa myös arkkitehtuuriin, koska raskastekoinen arkkitehtuuri- ja suunnitteludokumentointi reagoivat hitaasti nopeasti muuttuviin vaatimuksiin. (Boehm, 2002)

Ketterissä menetelmissä neuvottelu- ja analysointivaihe keskittyy vaatimusten muutoksiin, hienosäätöön ja priorisointiin. Korkeantason vaatimukset analysoidaan projektin alussa. Tässä vaiheessa tunnistetaan järjestelmän kannalta tärkeimmät vaatimukset, jotka toimivat aloituskohtana tarkemmalle suunnittelulle. Dokumentointi ketterässä vaatimusmäärittelyssä suoritetaan kirjoittamalla vapaamuotoisia käyttäjätarinoita tai listaamalla mitä ominaisuuksia järjestelmä sisältää. Dokumentointi onkin pääosin korvattu asiakkaan ja kehitystiimin välisillä keskusteluilla ja viestinnällä. Vaatimusten validointi tehdään arviointipalavereissa jokaisen iteraation jälkeen. Palaverin tarkoituksena on esitellä toteutettuja ominaisuuksia ja antaa palautetta niistä. Lisäksi arviointipalaveri varmistaa projektin etenemisen aikataulussa, lisää luottamusta sidosryhmissä ja parantaa ongelmien havaitsemista. Usein validointi tehdään jo iteraation suunnittelutapaamisissa. Myös hyväksyntätestejä voidaan käyttää validoinnin apuna. Validointivaiheessa on oleellista se, miten vaatimukset vastaavat asiakkaan tarpeita ja toiveita, eikä niinkään vaatimusdokumentaation kattavuus ja johdonmukaisuus. (Ramesh ym., 2007)

4.2.2 Vaatimusten kehittymiseen perustuva vertailu

Perinteisen ja ketterän vaatimusmäärittelyn eroja voidaan tarkastella myös Pohlin (1994) esittämän kolmeen ulottuvuuteen perustuvan kuutiomallin (Kuvio 2) avulla. Samat kolme ulottuvuutta, spesifiointi, esittäminen ja sopiminen ovat mukana myös ketterässä vaatimusmäärittelyssä, kuitenkin erilaisella painotuksella. Zhang ym. (2010) ovat kuvanneet Pohlin (1994) esittämiä ulottuvuuksia ketterän vaatimusmäärittelyn näkökulmasta seuraavasti.

Ensimmäisenä ulottuvuutena on spesifiointi. Zhang ym. (2010) mainitsevat, että vaatimuksia ei tarvitse spesifioida ennalta projektin alkuvaiheissa. Kaikkia näkökulmia tai tarpeita ohjelmistolle ei tällöin vielä tiedetä riittävällä tarkkuustasolla, joten vaatimusten spesifiointi on turhaa ennen ohjelmiston tai prototyypin kehitystyön aloittamista. Zhang ym. (2010) toteavat, että vaatimukset voidaan määritellä tarkasti vasta silloin, kun ne valitaan toteutettavaksi. Asiakkaiden ymmärrys ja tarpeet nimittäin muuttuvat kehitystyön edetessä. Perinteisessä vaatimusmäärittelyssä vaatimukset pyritään spesifioimaan tarkasti jo alkuvaiheessa.

Toisena ulottuvuutena on esittäminen. Ketterässä vaatimusmäärittelyssä voidaan käyttää prototyyppejä ja toimivaa ohjelman osaa selkiyttämään ja li-

säämään asianomaisten ymmärrystä kehitysprojektin lopputuloksesta. Prototyyppien avulla voidaan esittää todellisen ohjelmiston toimintoja, käyttäytymistä ja validoida vaatimuksia. Perinteisessä vaatimusmäärittelyssä vaatimuksia esitetään epämuodollisina (esim. animaatio), puoliformaaleina (esim. kaaviot) ja lopuksi formaaleina (esim. dokumentaatio). Zhang ym. (2010) mainitsevat, että on myös mahdollista lisätä vaatimuksia toimivalle ohjelmistolle, kuten ketterässä kehittämisessä yleisesti tehdään. Tässä tapauksessa ohjelmisto rakennetaan olemassa olevan ohjelman päälle. Toimiva ohjelmisto on tehokas esittämistapa, koska se esittää esille saadut vaatimukset käytännössä, niiden todellisessa käyttöympäristössä. Ohjelmistoa, joka toimii oikein, voidaan Zhangin ym. (2010) mukaan käyttää perustana uusien vaatimuksien esillesaamiseksi sekä vaatimusten paranteluun. Tällainen paranneltu esitystapa mahdollistaa helpon siirtymän korkean tason vaatimusten kuvauksesta yksityiskohtaisempiin kuvauksiin sekä lisää vaatimusten selkeyttä ja ymmärrettävyyttä. Zhangin ym. (2010) mukaan vaatimusten kuvaus ei ole sidottu formaaliin esitystapaan, toisin kuin perinteisessä vaatimusmäärittelyssä, mikä mahdollistaa joustavan ja kevyemmän prosessin vaatimusten esittämiselle. Tämän vuoksi vaatimusten esittämisprosessi on joustava ja kevytrakenteinen.

Kolmantena ulottuvuutena on sopiminen, jossa asianomaisten henkilökohtaiset näkemykset vähitellen lähentyvät muodostaen lopuksi yhteisen näkemyksen vaatimuksista. Zhang ym. (2010) mainitsevat, että inkrementaalinen kehittäminen hyödyntää prototyyppejä ja toimivia ohjelmanosia. Tämä mahdollistaa sen, että osa vaatimuksista voidaan lisätä edellisissä inkrementeissä kehitettyyn ohjelmaan. Uusi versio edellisestä prototyypistä tai toimivasta ohjelmasta lisää asianomaisten käsitystä lopullisesta ohjelmistosta ja mahdollistaa henkilökohtaisten mielipiteiden ja näkemysten esittämisen ohjelmasta. Zhangin ym. (2010) mukaan useat ohjelman julkaisut antavat asianomaisille mahdollisuuden arvioida ohjelmaa, muuttaa käsityksiä lopullisesta ohjelmistosta ja antaa palautetta tulevissa iteraatioissa toteutettaville vaatimuksille. Joustavuus auttaa henkilökohtaisten näkemysten muovautumista yhteiseksi näkemykseksi ohjelmistosta. Perinteisessä vaatimusmäärittelyssä sopiminen asianomaisten välillä helpottuu sitä mukaan, kun vaatimusten spesifiointi ja esittäminen tarkentuvat. Sopimisen osalta perinteinen ja ketterä vaatimusmäärittely etenevät samalla tavalla, joskin ketterässä vaatimusmäärittelyssä se on joustavampaa.

4.3 Yhteenveto

Tässä luvussa vertailtiin perinteistä ja ketterää kehittämistä ja vaatimusmäärittelyä yleisellä tasolla. Ohjelmistokehityksen osalta erot näkyvät mm. projektien johtamisessa ja hallinnassa, yleisessä toimintatavassa, kehittäjien ja asiakkaiden rooleissa, suunnittelussa, arkkitehtuurissa ja toteutuksessa. Ketterä lähestymistapa keskittyy enemmän ihmisiin, kun taas perinteinen lähestymistapa on enemmän prosessipainotteinen.

Vaatimusmäärittely on jatkuvaa ketterässä kehittämisessä ja vaatimukset voivat muuttua projektin edetessä, kun taas perinteisessä lähestymistavassa vaatimusmäärittely tehdään pääosin projektin alussa, eikä muutoksia tarkasti dokumentoituihin vaatimuksiin haluta tehtävän. Myös vaatimusmäärittelyvaiheiden ajoituksessa esiintyy eroja. Perinteisen vaatimusmäärittelyn vaiheet suoritetaan suurelta osin peräkkäisinä, kun taas ketterässä vaatimusmäärittelyssä niitä suoritetaan voimakkaasti iteroiden.

5 KETTERÄN VAATIMUSMÄÄRITTELYN KÄYTÄNTEITÄ

Tässä luvussa kuvataan ketterää vaatimusmäärittelyä yksityiskohtaisemmin käytänteiden ja tekniikoiden tasolla. Aluksi tarkastellaan vaatimusten esittämistä ja niiden jakamista osiin, jos ne ovat liian laajoja ymmärtämisen ja työmäärän arvioimisen kannalta. Tämän jälkeen kerrotaan, miten ei-toiminnallisia vaatimuksia on esitetty määritettävän ja käsiteltävän. Neljäntenä esitellään erilaisia vaatimusten priorisointitekniikoita. Viidenneksi kuvataan, millaisina vaatimusten laatukriteerit nähdään ketterässä kehittämisessä. Kuudentena tarkastellaan yleisemmin dokumentaatiota, ja seitsemäntenä esitetään, miten mallintamista voidaan käyttää vaatimusmäärittelyn tukena. Tämän jälkeen kerrotaan, millä tavalla ketterässä kehittämisessä pyritään varmistamaan vaatimusten jäljitettävyys. Luku päättyy vaatimusmäärittelyn roolien tarkasteluun ja yhteenvetoon.

5.1 Vaatimusten esittäminen

Ketterässä vaatimusmäärittelyssä vaatimukset esitetään käyttäjätarinoiden muodossa. Leffingwellin (2011, 59) mukaan ketterissä menetelmissä (esim. XP) käyttäjätarinat kirjoitetaan usein käsin sopiville korteille. Kortteja käytetään suunnitteluun, arviointiin, priorisointiin ja esittämiseen päivittäisissä palavereissa. XP:ssä asiakas kirjoittaa käyttäjätarinan, kun taas Scrum-menetelmässä tuoteomistaja on vastuussa tarinoiden kirjoittamisesta. Myös kehittäjät voivat osallistua käyttäjätarinoiden laadintaan. Tuoteomistaja hyväksyy ja määrittelee tarinoiden priorisoinnin. Koska XP:ssä asiakas on kirjoittajan roolissa, hänen osallistumisensa projektiin linkittyy käyttäjätarinoiden kautta tiiviisti. (Leffingwell, 2011, 100). Tarinat voivat olla aluksi laajoja kuvauksia, ja niiden sisältämien vaatimusten abstraktiotaso on yleensä korkea. Tarinoita kuitenkin tarkennetaan toteutusvaiheessa riittävän yksityiskohtaisiksi tehtäviksi (Zhang ym., 2010).

Käyttäjätarinat ovat alun perin lähtöisin XP:stä, josta ne ovat siirtyneet muihin ketteriin menetelmiin. Leffingwell (2011,100) määrittelee käyttäjätarinan seuraavasti: *käyttäjätarina* esittää lyhyesti ja selkeästi toiminnallisen vaatimuksen, joita järjestelmän kuuluu toteuttaa. Tarinat toimivat välineinä, jotka määrittelevät järjestelmän käyttäytymistä ymmärrettävästi käyttäjille ja kehittäjille. Huomion kohde tarinoissa on käyttäjän saama hyöty ja arvo. Käyttäjätarinat mahdollistavat myös tehokkaan vaatimusten hallinnan, koska tarinat ovat kooltaan pieniä ja kevyitä.

Useimmiten tarinat ovat kirjoitettu lapuille, jotka kuvailevat lyhyesti jonkin järjestelmälle halutun toiminnon. Yksinkertaisimmillaan ne muodostavat listan toiminnoista tai ominaisuuksista, joita järjestelmän tulee suorittaa. Koska käyttäjätarinat ovat lyhyitä, ei niissä kuvailla järjestelmän käyttäytymistä yksityiskohtaisesti. (Leffingwell, 2011, 101) Liian yksityiskohtaisen kuvauksen tekemisellä vältetään viivästykset kehittämisessä, ylimääritellyt rajoitukset toteutettavaan ohjelmistoon, vaatimusten inventaariot ja liian aikaisin tehdyt spesifioinnit. Käyttäjätarinat tarvitsevat vähän ylläpitoa, joten niitä ei tarvitse säilyttää toteutuksen jälkeen. (Leffingwell, 2011, 102) Tarkempi käyttäytymisen suunnittelu ja hyväksyntäkriteerit tehdään tiimin ja tuoteomistajan välisissä keskusteluissa myöhemmissä vaiheissa.

Leffingwellin (2011, 101) mukaan käyttäjätarinat eivät ole vaatimuksia perinteisen lähestymistavan mukaisesti käsitettynä. Erot ilmenevät pieninä yksityiskohtina, joita Leffingwellin (2011, 101) mukaan ovat: käyttäjätarinat eivät ole yksityiskohtaisesti määriteltyjä vaatimuksia (engl. system shall do), vaan ne ovat neuvoteltavissa olevia ilmaisuja siitä, mitä järjestelmän tarvitsee tehdä (engl. system needs to do). Tarinat esittävät arvoa tuottavia toimintoja, jotka voidaan toteuttaa muutamassa päivässä tai viikossa. Lisäksi tarinoiden toteuttamiseen tarvittava työmäärä on helppo ja nopea arvioida. Listoihin on myös nopea lisätä uusia tarinoita ja poistaa vanhoja (vertaa perinteisen vaatimusmäärittelyn yksityiskohtaisiin dokumentteihin). Käyttäjätarina sekä koodi muodostavat dokumentaation, jota päivitetään projektin kuluessa. Leffingwell (2011, 101)

Käyttäjätarinoilla on Leffingwellin (2011, 103) mukaan yleinen muoto, joka koostuu roolista, toiminnosta ja liiketoiminta-arvosta. Yleinen rakenne on:

> *Toimiessani <roolissa> minä pystyn tekemään <toiminnon> ja sillä saavuttamaan <liiketoiminta-arvoa>.*

Esimerkki käyttäjätarinarakenteen käytöstä (Leffingwell, 2011,103):

> *Kuluttajana <rooli> haluan nähdä päivittäisen sähkönkulutukseni <toiminto>, jotta voin vähentää sähkönkulutusta <liiketoiminta-arvo>.*

Rooli merkitsee henkilöä, joka suorittaa toiminnan tai saa hyötyä toiminnasta. *Toiminto* tarkoittaa suoritusta, jonka järjestelmä tekee. *Liiketoiminta-arvo* esittää sitä arvoa, mikä toiminnosta saadaan. Leffingwell (2011) käyttää nimitystä käyttäjänääni (engl. user voice), koska se laajentaa ongelma- ja ratkaisualuetta. Tär-

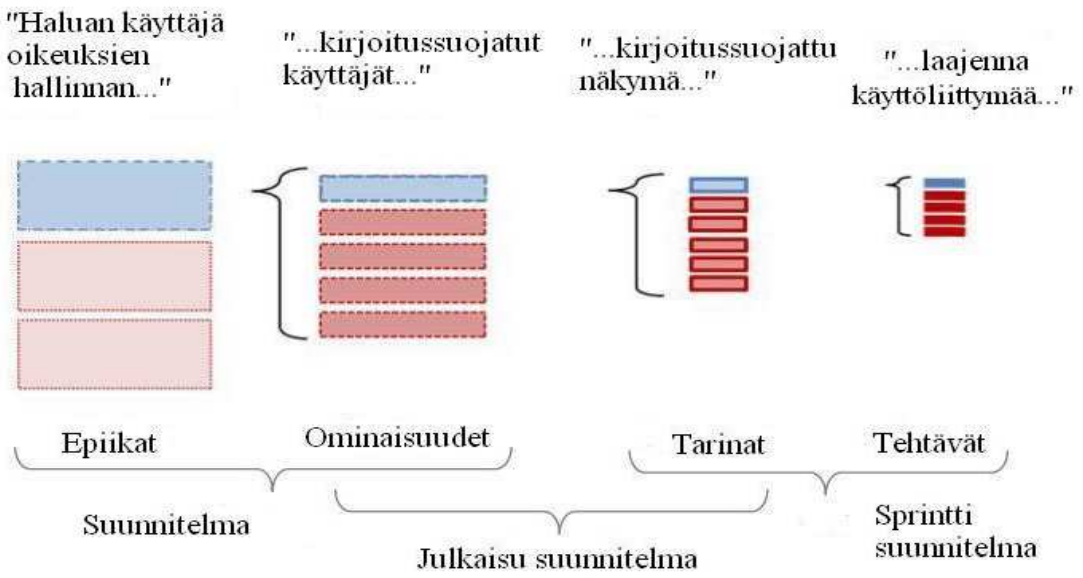
keänä huomiona Leffingwell (2011, 104) näkee sen, että tiimi keskittyy kahteen oleelliseen asiaan: liiketoiminta-arvon lisäämiseen sekä todellisten ongelmien selvittämiseen. Näkökulma käyttäjätarinassa on käyttäjän huomioiminen, mikä suuntaa tiimin huomion edellä mainittuihin seikkoihin.

5.2 Vaatimusten jakaminen

Usein asiakkaan käyttäjätarinat ovat liian suuria toteutettavaksi yhdessä iteraatiossa. Tämänlaisissa tapauksissa käyttäjätarinat jaetaan pienemmiksi osiksi. Leffingwell (2011) ja Cohn (2005) kutsuvat suuria käyttäjätarinoita epiikeiksi (engl. epics) ja ominaisuuksiksi (engl. features). *Epiikit* ovat suuria ja epämääräisiä hahmotelmia käyttäjän haluamista toiminnoista ja ominaisuuksista. Suurten käyttäjätarinoiden toteuttamisen kestoa on vaikea arvioida, joten niitä on senkin takia hyödyllistä jakaa pienempiin osiin. (Cohn, 2005, 121; Leffingwell, 2011, 111).

Vähäniitty (2012, 28) on kuvannut vaatimusten jakamista suurista *epiikeistä*, ominaisuuksien kautta tarinoihin ja tehtäviin Kuvion 6 tavoin. Ensimmäisenä Kuviossa 6 on käyttäjän esittämä epämääräinen vaatimus käyttäjäoikeuksien hallinnasta. Sitä tarkentamalla saadaan *ominaisuus* järjestelmälle, joka rajaa hallinnan kirjoitussuojatuille käyttäjille. Ominaisuutta jälleen tarkentamalla siitä saadaan *käyttäjätarina*, jossa vaatimuksena on toteuttaa kirjoitussuojattu näkymä järjestelmään. Lopulta käyttäjätarina tarkentuu *tehtäväksi*, jossa tiettyä käyttöliittymän osaa täytyy laajentaa, jotta käyttäjäoikeuksien hallinta onnistuu. Punaiset palkit esittävät vaatimuksia ja niiden toteuttamiseen arvioitua aikaa. Kuten Kuviosta 6 käy ilmi, vaatimukset tarkentuvat sitä enemmän mitä pienempiin osiin ne jaetaan ja samalla toteuttamiseen kuluva aika vähenee (palkin väri tummenee ja koko pienenee). Useasti epiikeissä esitetty epämääräinen vaatimus näyttää paljon aikaa vievältä toiminnolta tai ominaisuudelta. Kuitenkin riittävällä tarkentamisella ja jakamisella alkuperäinen vaatimus paljastuu usein varsin helpoksi ja nopeaksi toteuttaa. Vaatimuksia tarkennetaan siten, että niitä mahtuu 3–4 kappaletta iteraatioon. (Vähäniitty, 2012, 29)

Vlaanderen, Brinkkemper ja Jansen (2010) tarkastelevat vaatimusten jakamista käyttäen vähän erilaista terminologiaa. He nimeävät seuraavat tarkkuustasot vaatimuksille: näkemys (engl. vision), teema (engl. theme), luonnos (engl. concept) ja vaatimusten määritys (engl. requirements definition). Ensimmäisenä tasona on *näkemys* eli visio siitä, millainen järjestelmän tulisi olla. Asiakas kuvailee yleisesti, mitä ominaisuuksia ja toimintoja järjestelmä sisältää. Kun näkemys on muodostettu, siirrytään toiselle tasolle, joka on teema. *Teemat* tarkentavat näkemystä yksityiskohtaisemmaksi. Ne kuvaavat liiketoiminnallisen ongelman ja sen pääkohdat. Scrum-menetelmässä tuotepäällikkö määrittelee toiminnallisuuden tarkoituksen ja sen tuoman liiketoiminta-arvon. Ennen kuin teemat hyväksytään, tiimi arvioi niiden toteutuskelpoisuuden.



KUVIO 6 Vaatimusten progressiivinen kehittyminen eniten arvoa lisääviksi (Vähäniitty, 2012, 28).

Kolmantena tasona on *luonnos*, jolla teema jaetaan pienempiin osiin. Se on teeman korkeantason ydinajatus, joka koostuu ratkaisujen kuvauksista (engl. solution stories). Näistä kuvauksista voidaan muodostaa yksityiskohtaisia vaatimuksia kehitettävälle järjestelmälle tai ohjelmistolle. Luonnoksen kuvaus sisältää sen tarkoituksen lyhyesti ja riittävän tarkasti esitettynä, jotta kuvauksesta voidaan määritellä yksityiskohtaisia *vaatimuksia*. Neljäs taso sisältää vaatimusten määrittelyn, jossa luonnoksista johdetaan lista vaatimuksia.

Aluksi vaatimukset kuvataan ilman yksityiskohtaisia määrityksiä. Tämän jälkeen kehitystiimi arvioi vaatimusten yhteensopivuuden ja toteutuskelpoisuuden. Vasta näiden vaiheiden jälkeen tiimi määrittelee vaatimukset yksityiskohtaisesti ja niiden sisältämät toiminnallisuudet kahdessa vaiheessa. Scrum-menetelmässä Vlaanderen ym. (2010) mukaan tiimi käsittelee määritykset yhden sprintin aikana ja varmistaa vaatimusten toteutuskelpoisuuden, johdonmukaisuuden ja ymmärrettävyyden. Toinen tarkastelu vaatimuksille tehdään niiden selkeyden varmistamiseksi siten, että jokainen tiimin jäsen ymmärtää jokaisen vaatimuksen. Lopulta vaatimukset ovat riittävän tarkasti määriteltyjä ja valmiita toteutettavaksi. Vaatimuksen kuvaukset voivat sisältää kaavioita, teknistä spesifikaatiota ja muuta oleellista tietoa. (Vlaanderen ym., 2010)

Cohn (2005, 122) on esittänyt vaihtoehtoisia tapoja käyttäjätarinoiden jakamiselle. Ensimmäinen tapa on jakaa suuri käyttäjätarina datan rajojen mukaisesti. Tiimi voi jakaa tarinan datan tyypin perusteella. Esimerkiksi data voi olla käyttäjän antamia syötteitä, jotka ovat negatiivisia lukuja tai liittyvät tiettyyn asianyhteyteen, kuten kotimaan tai ulkomaan puhelinnumerot. (Cohn 2005, 123) Toinen tapa on poistaa käyttäjätarinasta viittaukset poikkeustilanteiden käsittelyihin. Ilman poikkeustilanteen kuvausta käyttäjätarina muodostaa oman ko-

konaisuuden ja poikkeus- tai virhetilanteen sisältävä käyttäjätarina oman kokonaisuuden. Näin toimimalla yhdestä käyttäjätarinasta tulee kaksi pienempää kokonaisuutta (Cohn, 2005, 123).

Jakamisen voi toteuttaa myös käyttäjätarinan sisältämien toimintojen perusteella. Mikäli käyttäjätarinassa on paljon toiminnallisuuksia (esim. hakutoiminto ja päivitystoiminto), jotka ovat liian hankalia toteuttaa yhdessä iteraatiossa, toiminnot jaetaan siten, että ne voidaan toteuttaa yhdessä iteraatiossa kokonaan tai osittain. Vaikka sovellukseen haluttu toiminto ei toteutuisi valmiiksi asti yhden iteraation aikana, voidaan sen toimintaperiaate esittää sidosryhmille. Tässä yhteydessä priorisointi on avuksi, koska monimutkaisten toimintojen toteuttamisen vaatimaa aikaa on vaikeaa arvioida. Toteuttamalla helpommat toiminnot ensiksi kehitystiimi kartuttaa ymmärrystään kohdealueesta ja pystyy arvioimaan vaikeampien toimintojen keston tarkemmin. (Cohn, 2005, 124).

Edellä esitetyn toimintojen jakamisen voi Cohnin (2005) mukaan tehdä käyttämällä CRUD-menetelmää (engl. Create, Read, Update, Delete). CRUD-menetelmän avulla esimerkiksi käyttäjätarina, jossa vaatimuksena on lisätä käyttäjiä, poistaa käyttäjiä ja muokata käyttäjien tietoja, jaetaan kolmeen osaan. Jokainen edellä mainittu toiminto toteutetaan iteraatiossa tai iteraatioissa omana kokonaisuutenaan. (Cohn, 2005, 125).

Sovelluksessa on ominaisuuksia, jotka ovat ortogonaalisia tai yhteydessä toisensa kanssa. Muihin ominaisuuksiin vaikuttavia ominaisuuksia ovat esimerkiksi turvallisuus, tiedontallennus ja virheiden käsittely (engl. errorhandling). Ominaisuuksia voidaan eristää toisistaan, jolloin käyttäjätarina voidaan jakaa iteraatioihin sopiviin osiin. Esimerkiksi sovellus, jonka on tarkoitus esittää käyttäjäoikeuksien mukaisesti erilaisia näkymiä, voidaan Cohnin (2005) mukaan jakaa kahteen iteraatioon. Ensimmäiseksi toteutetaan näkymät ja toiseksi käyttäjäkohtaiset oikeudet (Cohn, 2005, 126).

Käyttäjätarinoissa esiintyvien toiminnallisten ja ei-toiminnallisten vaatimuksien toteuttamiseen kehitystiimi tarvitsee erisuuruisen ajan. Erityisesti ei-toiminnalliset vaatimukset vaativat pidemmän ajan. Ratkaisuna tähän ongelmaan Cohn (2005) ehdottaa, että toiminnalliset ja ei-toiminnalliset vaatimukset toteutetaan omissa iteraatioissaan.

Käyttäjätarinoita voidaan jakaa myös asettamalla erilaisia prioriteetteja vaatimuksille. Usein suuressa käyttäjätarinassa on monta pienempää käyttäjätarinaa, jotka ovat prioriteetiltaan vähäisempiä. (Cohn, 2005).

Joissakin tapauksissa jokin ominaisuus käyttäjätarinassa voi olla vaikea jakaa. Yleisesti tämän kaltainen ongelma ratkaistaan osittamalla ominaisuus rakenteellisiin tehtäviin (engl. constituent task). Cohnin (2005) mukaan käyttäjätarinaa ei tulisi kuitenkaan jakaa rakenteellisiin tehtäviin. Sen sijaan tulisi toteuttaa osa jokaisesta ominaisuuden kerroksesta. Esimerkiksi toteutetaan vain osa käyttöliittymästä, osa sovelluslogiikasta ja osa tietokannasta, eikä yhtä osa-aluetta kokonaan (Cohn, 2005, 128).

Myös erästä XP:ssä esiintyvää tekniikkaa voidaan käyttää käyttäjätarinoiden jakamiseen. Piikkien (engl. spikes) avulla kehitystiimi voi analysoida käyt-

täjätarinan sisältämien toimintojen tai ominaisuuksien käyttäytymistä ja arvioida käyttäjätarinan jakamista pienempiin osiin. (Leffingwell, 2011, 114)

5.3 Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset ovat haasteellisia ketterässä vaatimusmäärittelyssä, koska niihin ei useinkaan kiinnitetä samaan tapaan huomiota kuin toiminnallisiin vaatimuksiin. Ne ovat kuitenkin yhtä tärkeitä kuin toiminnalliset vaatimukset, koska ne vaikuttavat ohjelmiston laatuun ratkaisevasti. Seuraavaksi tarkastellaan ensin ei-toiminnallisten vaatimusten määrittelyä yleisesti ja sen jälkeen erikseen ei-toiminnallisia vaatimuksia, jotka koskevat käytettävyyttä, luotettavuutta, suorituskykyä ja turvallisuutta.

Rodriquezin, Yagilen, Alarconin ja Garbajosan (2009) mukaan ajatuksena ketterässä vaatimusmäärittelyssä on, että käyttäjätarinat ovat riittäviä toiminnallisten ja ei-toiminnallisten vaatimusten esittämiseksi. Tästä johtuen selkeää luokittelua toiminnallisten ja ei-toiminnallisten vaatimusten välillä ei tarvitse tehdä. Monissa tapauksissa ei-toiminnalliset vaatimukset koskevat useampaa käyttäjätarinaa, jolloin niistä tulee läpileikkaavia vaatimuksia (engl. cross-cutting requirements). Läpileikkaavat vaatimukset vaikuttavat suunnitteluun, arviointiin ja testaukseen, mikä tekee niiden hallinnan vaikeaksi. (Rodriquez ym., 2009)

Paetsch ym. (2003) huomauttavat, että ketterään vaatimusmäärittelyyn tulisi lisätä ei-toiminnallisten vaatimusten painotusta enemmän. Niiden huomiotta jättäminen lisää riskejä, koska niiden hallintaan ei ole määriteltyä tekniikkaa. Myös Ramesh ja Cao (2008) huomioivat ei-toiminnallisten vaatimusten laiminlyönnin, josta voi aiheutua merkittäviä ongelmia tulevaisuudessa. Rameshin ja Caon (2008) mukaan poikkeuksen kuitenkin tekee vaatimus helposta käytettävyydestä, joka asiakkaan aktiivisen läsnäolon myötä on huomioitu kunnolla.

Leffingwellin (2011, 341) mukaan ketterissä menetelmissä ei osoiteta tarkkaan, missä kohtaa ei-toiminnalliset vaatimukset tulisi ottaa huomioon. Hän esittää itse kokonaiskuvan (engl. big picture) (Leffingwell 2011, 32), jossa on tuotu erikseen esille ei-toiminnalliset vaatimukset. Niitä kutsutaan työlistarajoituksiksi (engl. backlog constraints), koska ei-toiminnalliset vaatimukset rajoittavat työlistassa olevien toiminnallisten vaatimusten toimintaa. Yksi ei-toiminnallinen vaatimus voi rajoittaa yhtä tai useampaa toiminnallista vaatimusta. Ei-toiminnalliset vaatimukset voivat myös koskea (engl. apply) useampaan työlistan tehtävää. Esimerkiksi ei-toiminnallinen vaatimus ”järjestelmä tukee 100 samanaikaista käyttäjää” voi vaikuttaa useaan tehtävään työlistassa. (Leffingwellin, 2011, 341) Ei-toiminnallisia vaatimuksia voidaan kirjoittaa käyttäjätarinoiden muotoon, kuten ketterissä menetelmissä on tapana tehdä. Käyttäjätarinoina esitetyt ei-toiminnalliset vaatimukset selventävät, mistä vaatimus tulee, lisäävät liiketoiminnallista arvoa käyttäjälle ja lisäävät vaatimuksen ymmärrettävyyttä. (Leffingwell, 2011, 342) Leffingwell (2011, 342) on esittänyt esi-

merkkejä ei-toiminnallisista vaatimuksista käyttäjätarinoina. Vertailun vuoksi vaatimukset ovat esitetty ensiksi perinteisen vaatimusmäärittelyn mukaisesti:

1. Kaikki viestit täytyy näyttää vähintään minuutin kuluessa saapumisesta.
2. Kaikkien avoimen lähdekoodin ohjelmistojen täytyy olla talouspäällikön hyväksymiä.
3. Päivitä kännykkäsovellus uudella logolla.

Samat kolme ei-toiminnallista vaatimusta käyttäjätarinoina (Leffingwell 2011, 342):

1. Asiakkaana haluan, että minulle ilmoitetaan kaikki viestit vähintään minuutin kuluessa niiden saapumisesta, jotta voin reagoida ja toimia nopeasti. (lisää paljon arvoa).
2. Talouspäällikkönä haluan varmistua, että emme käytä avoimen lähdekoodin ohjelmistoja, joita en ole hyväksynyt, jotta emme riko lisenssejä. (lisää arvoa jonkin verran).
3. Tuotepäällikkönä haluan varmistaa, että markkinointiosasto hyväksyy päivitetyn logon. (ei lisää kovin paljoa arvoa).

Seuraavassa tarkastellaan lähemmin esityksiä, jotka koskevat tietyn tyyppisten ei-toiminnallisten vaatimusten määrittämistä ketterässä kehittämisessä. Tällaisia ovat käytettävyyttä, luotettavuutta, suorituskykyä ja turvallisuutta koskevat vaatimukset.

Käytettävyys on tärkeä ominaisuus ohjelmistoissa. Leffingwell (2011, 343) mainitsee, että ohjelmiston käytettävyys on usein käyttäjäkohtaista eikä siihen ole yksiselitteistä spesifiointitapaa. Käytettävyyden määrittelyssä huomio kiinnitetään, esimerkiksi miettimällä, kuinka paljon käyttäjä tarvitsee harjoittelua perus- ja tehokäyttäjän tehtävistä suoriutumiseksi. Käytettävyyttä voidaan parantaa vertailemalla käyttäjien kokemusta samanlaisista järjestelmistä, joita he ovat käyttäneet aikaisemmin. Myös erilaiset manuaalit, vinkit ja velhot (engl. wizards) auttavat käytettävyyden lisäämisessä. (Leffingwell, 2011, 343).

Sohaibin ja Khanin (2010) mukaan käytettävyyden määrittely ja suunnittelu on sivuutettu ketterissä menetelmissä liian helposti. Ketterät menetelmät toteuttavat toimivan ohjelmiston, mutta se ei välttämättä ole hyvä käytettävyydeltään. Sohaib ja Khan (2010) toteavat, että erityisesti käytettävyysvaatimukset ovat puutteellisesti huomioitu. Ketteriin menetelmiin tulisikin lisätä käytettävyyden suunnittelua, jotta ohjelmiston käytettävyys parantuisi.

Silvan, Martinin, Maurerin ja Silveiran (2011) mukaan käyttäjäkeskeisen suunnittelun (engl. user centered design) ja ketterien menetelmien yhteensovittamisessa ongelmaksi muodostuvat menetelmien erilaiset toimintatavat. Käyttäjälähtöinen suunnittelu vaatii paljon analysointia ja tutkimusta ennen ominaisuuksien toteuttamista, kun taas ketterät menetelmät tuottavat mahdollisimman nopeasti toimivia osia ohjelmistosta. Käytettävyyden ja ketterien menetelmien integroimisessa voidaan hyödyntää erilaisia käytänteitä, kuten etukäteis-

suunnittelu, prototyypit, käyttäjätarinat, käyttäjätestaus (user testing), käytettävyyden arviointi ja yhden sprintin edelläkäynti (engl. one sprint ahead) (Silva ym., 2011).

Silvan ym. (2011) mukaan lyhyt käytettävyyden suunnitteluvaihe voidaan lisätä ketteriin menetelmiin ennen projektin aloittamista. Esimerkiksi, XP:ssä ennen suunnitteluvaihetta (engl. planning game) asiakkaille voidaan tehdä kysely käytettävyydestä, jolloin suunnitteluvaiheen aikana voidaan keskustella käytettävyyteen liittyvistä vaatimuksista. Scrumin mukaisessa kehittämisessä voi olla kaksi tuoteomistajaa. Toinen tuoteomistaja keskittyy pelkästään käytettävyyteen ja toinen muihin tuoteomistajan tehtäviin. Käyttäjätarinoista osan tulisi olla pelkästään käytettävyyteen liittyviä. Myös prototyyppejä voidaan hyödyntää käytettävyyden parantamiseksi. Prototyypit mahdollistavat käyttöliittymälle ja käytettävyydelle esitettyjen vaatimusten kokeilemisen ja neuvottelemisen. Lisäksi prototyyppejä voidaan käyttää pohjana käytettävyyden kehittämiselle ja käyttää käytettävyyden arvioimisessa. Käytettävyyttä on hyvä testata käyttäjillä jokaisen iteraation jälkeen. Testauksessa voidaan esimerkiksi käyttää skenaarioita, paperiprototyyppejä, ääneen ajattelua ja käyttäjien palautetta. Testauksen tavoitteena, tekniikasta riippumatta, on käytettävyyden kehittäminen. Silvan ym. (2011) mukaan käyttäjätarinat voidaan laatia esimerkiksi käytettävyys skenaarioista. Käytettävyyden arviointi voidaan Silvan (2011) mukaan tehdä prototyyppien yhteydessä tai sprintin arvioinnin aikana. Näin kehittäjät osallistuvat käytettävyyden arviointiin yhdessä käytettävyysasiatuntijoiden kanssa. Silvan ym. (2011) mukaan käytettävyyden suunnittelutiimien tulisi edetä aina yhden sprintin edellä kehitystiimiä. Näin käytettävyyden suunnittelu on aina valmiina kehittäjien toteutettavaksi sekä käytettävyyttä voidaan parantaa siitä saadun palautteen perusteella seuraavissa sprinteissä. Muita tekniikoita ovat käyttäjien tehtäväanalyysin liittäminen käyttäjätarinoihin ja käyttäjätarinoiden kirjoittaminen käytettävyystestauksen perusteella. Käyttäjätarinoissa olisi kuitenkin hyvä huomioida myös ohjelmiston käytettävyys niitä arvioitaessa. (Silva ym., 2011)

Beyer, Holtzblatt ja Baker (2004) ovat kehittäneet nopean kontekstuaalisen suunnittelumenetelmän, jossa yhdistetään ketterä kehittäminen kontekstuaaliseen suunnitteluun. Tässä menetelmässä käytettävyystiimi toimii yhden sprintin edellä kehitystiimiä ja toteuttaa tarvittavat, sidosryhmillä testatut suunnitelmat käyttöliittymälle sekä käytettävyydelle kehitystiimin toteutettavaksi. Syn ja Millerin (2008) kehittämää Autodesk-menetelmää voidaan käyttää myös käytettävyyden suunnitteluun. Myös tässä menetelmässä käytettävyystiimi aloittaa suunnittelun ennen ensimmäistä iteraatiota ja pysyy suunnittelussa muutaman iteraation edellä kehitystiimiä. Näin käytettävyyttä voidaan testata ja asiakas voi hyväksyä tai hylätä suunnitelman. Singhin (2008) U-Scrum menetelmä on kehitetty Scrumin ja käytettävyyssuunnittelun yhdistämiseen. Tässä menetelmässä Scrumiin otetaan mukaan toinen tuoteomistaja, joka keskittyy vain käytettävyyteen ja siihen liittyvien tehtävien hoitamiseen. Menetelmässä lisätään Scrumiin suunnittelustudio, jossa kehitystiimi, käytettävyystiimin ja sidosryh-

mät tekevät yhteistyötä käytettävyysvaatimuksia vastaavan suunnitelman löytämiseksi. Erilaisista vaihtoehdoista keskustellaan palautetta antaen.

Luotettavuus on oleellinen ominaisuus, joka mahdollistaa järjestelmän oikean toiminnan. Luotettavuus sisältää useita osatekijöitä, kuten saatavuus (engl. availability), tarkkuus (engl. accuracy), vikojen välinen ilmenemisaika (engl. time between failures), vikojen korjausaika (engl. repair time), viat (engl. defects) ja turvallisuus (engl. security) (Leffingwell 2011, 344). Tärkeimpinä edellä mainituista ominaisuuksista Leffingwellin (2011) mukaan ovat saatavuus ja turvallisuus. Vaatimuksena voi olla, että järjestelmä on saatavilla ja toiminnassa tiettyinä ajankohtina tai koko ajan. Turvallisuus on erityisen tärkeää sovelluksissa. Etenkin liiketoiminnan kannalta järjestelmän turvallisuus tulee suunnitella tarkasti. Turvallisuutta voidaan määritellä myös toiminnallisten vaatimusten, suunnittelun, koodauksen ja standardien kautta (Leffingwell, 2011, 344). Luotettavuuden testaukseen on ohjelmointikielispesifejä työkaluja, joiden avulla kehittäjät voivat testata mahdollisia luotettavuusvikojen aiheuttajia, kuten muistivuotoja (engl. memory leaks). (Leffingwell, 2011, 351)

Far (2007) esittää luotettavuuden parantamiseksi käytänteitä ohjelmistojen luotettavuustekniikasta (engl. software reliability engineering, SRE) yhdistettäväksi ketterään kehittämiseen. Eräs tällainen käytänne on tuotteen määrittely, jossa määritellään asiakas ja käyttäjät ohjelmistolle. Toiminnallisen profiilin kehittämisessä määritetään, miten ohjelmistoa käytetään. Toiminnallisten profiilien avulla voidaan ohjata myös testausta. Luotettavuuden testaus voidaan lisätä hyväksyntä- tai suorituskykytestaukseen. Testauksen tuloksia voidaan käyttää apuna ohjelmiston julkaisuajankohdan päättämiseksi. Testituloksien avulla voidaan arvioida, onko ohjelmiston luotettavuus riittävän korkealla tasolla, jotta se voidaan antaa asiakkaan käyttöön. (Far, 2007)

Suorituskyky ei-toiminnallisena vaatimuksena määrittelee, miten tehokkaasti järjestelmä vastaa käyttäjien ja toisten järjestelmien palvelupyyntöihin. Suorituskyky vaikuttaa siihen, miten järjestelmän käyttäytyminen muuttuu kuormituksen alaisena. Leffingwellin (2011) mukaan suorituskyvyn osatekijöitä ovat esimerkiksi vasteaika (engl. response time), kapasiteetti (engl. capacity), skaalautuvuus (engl. scalability) ja resurssien hyödyntäminen (engl. resource utilization). Leffingwellin (2011, 352) mukaan suorituskykyä voidaan testata siihen tarkoitetuilla työkaluilla, kuten erilaisilla käyttäjä- ja järjestelmänkuormitussimulaattoreilla. Avoimen lähdekoodin tai kaupalliset työkalut mahdollistavat järjestelmän palvelimien, verkoston, komponenttien ja muiden tärkeiden osien suuren kuormituksen testauksen. Tuloksia analysoidaan kokonaiskuvan saamiseksi järjestelmän toimivuudesta kuormituksen aikana.

Hon, Johnsonin, Williamsin ja Maximilien (2006) mukaan huonosti tehty vaatimusten spesifiointi vaikeuttaa suorituskyvylle asetettujen tavoitteiden ymmärtämistä. Heidän mielestään erityisesti ketterissä menetelmissä riski suorituskyvyn väärin ymmärtämiseksi on korkea johtuen puutteellisesta dokumentoimisesta.

Ho ym. (2006) esittävät PREM (engl. performance requirements evolution model) -mallin suorituskyvyn parantamiseksi. PREM-malli mahdollistaa suori-

tuskykyvaatimusten esillesaamisen riittävällä tarkkuudella niiden validoimiseksi ja toteuttamiseksi. PREM-mallissa suorituskyvyn spesifiointia tehdään neljällä tasolla. Jokaisella tasolla määritystä tarkennetaan, kunnes se on projektin ja vaatimusten kanssa tasapainossa. 0-tasolla suorituskykyvaatimukset ovat epämuodollisia ja kuvailtu asiakkaan näkökulmasta. 0-tasolla tunnistetaan järjestelmän ne osat, joilla on suorituskykyvaatimuksia. Tällä tasolla vaatimukset validoidaan laadullisella arvioinnilla. 1-tasolla vaatimukset ovat mittavassa muodossa, jolloin niitä voidaan testata automaattisesti. Tämän tason vaatimukset kuvaavat yhden käyttäjän ajonaikaista (engl. run-time) järjestelmää. Asiakas, tiimi, kohde-alueen asiantuntija tai tehty tutkimus voi asettaa 1-tason vaatimuksia. Validointi 1-tasolla suoritetaan yksinkertaisella testauksella. 2-tason vaatimukset ovat laadullisia ja spesifioivat järjestelmän prosessien käyttöasteen. Tällä tasolla havaitaan komponentin ja järjestelmän vuorovaikutus. Vaatimukset kuvaavat, millaisiin tavoitteisiin järjestelmän suorituskyvyn tulisi yltää usean käyttäjän aiheuttaman kuormituksen alaisena. 2-tason vaatimusten validointi tehdään yksinkertaisten mallien avulla. Vaatimukset tasolla 3 kuvaavat ”pahin mahdollinen tilanne”-tyyppistä tilannetta. 3-taso esittää, miten komponentit toimivat yhdessä kuormituksen alla. Erilaisille toiminnoille voidaan spesifioida tietty aikaraja, minkä rajoissa toiminto suoritetaan. 3-tason vaatimukset on suunnattu reaaliaikaisille ja suorituskykykriittisille järjestelmille. Validoiminen suoritetaan realistisilla käyttökuormilla. (Ho ym., 2006)

Peeters (2005) ehdottaa, että järjestelmän *turvallisuus* (engl. security) voidaan huomioida kirjoittamalla käyttäjätarinoiden vastapariksi väärinkäyttötarinoita (engl. abuser stories). Väärinkäyttötarinat kuvaavat, miten järjestelmää voidaan väärinkäyttää ja mitä seurauksia väärinkäytöllä on. Kirjoittamalla väärinkäyttötarinoita järjestelmän turvallisuusvaatimukset saadaan selville. Väärinkäyttötarinoita voidaan Peetersin (2005) mukaan luokitella sen mukaisesti, kuinka paljon ne uhkaavat järjestelmän liiketoiminnallista arvoa ja kuinka todennäköinen väärinkäyttö tai hyökkäys on. Peetersin (2005) mukaan väärinkäyttötarinoita testataan yrittämällä torjua (engl. refutation) ne. Torjuminen kuvaa, miten väärinkäyttötarinat estetään ilman niiden aiheuttamaa vahinkoa. Kuitenkin riski väärinkäytöksille on aina olemassa, eikä järjestelmä voi olla täysin suojattu. Väärinkäyttötarinat mahdollistavat liiketoiminta-arvon seuraamisen ja auttavat järkevän turvallisuussuunnitelman tekemisessä. Lisäksi väärinkäyttötarinoiden avulla voidaan arvioida tarvittava työmäärä turvallisuuden toteuttamiseksi. (Peeters, 2005)

5.4 Vaatimusten priorisointi

Vaatimusten *priorisoinnin* tarkoituksena on valita kehitettävään ohjelmistoon kaikkein tärkeimmät ja oleellisimmat vaatimukset sekä päättää niiden toteutusjärjestyksestä. Projektin budjetti ja aikataulu ovat rajalliset, joten on tärkeää toteuttaa hyödylliset toiminnot ja ominaisuudet ensin (Paetsch ym., 2003). Aikaisin tehty priorisointi auttaa tiimiä päättämään, mitkä vaatimukset voidaan aika-

taulun mukaisesti toteuttaa ja mitkä jättää pois. Asiakas ja tiimi keskustelevat, mitkä vaatimukset tuottavat eniten asiakastyytyväisyyttä ja arvoa. Tiimin tehtävänä on kertoa asiakkaalle mahdollisista riskeistä, kustannuksista sekä muista huomion arvoisista asioista liittyen asiakkaan priorisoimiin vaatimuksiin. Ketterissä menetelmissä vaatimukset priorisoidaan tavallisesti siten, että eniten liiketoiminta-arvoa lisäävät vaatimukset toteutetaan ensiksi (Paetsch ym., 2003).

Priorisointia tehdään useammassa yhteydessä ketterää kehittämistä. Ensimmäisessä vaiheessa priorisoinnin kohteena ovat epiikit ja ominaisuudet. Scrumin yhteydessä priorisoinnin kohteena ovat tuotteen työlistan (engl. product backlog) käyttäjätarinat. Priorisointia tehdään myös siinä vaiheessa, kun päätetään, mitkä tuotteen työlistalla olevista käyttäjätarinoista valitaan seuraavan iteraation työlistalle (Scrumissa engl. sprint backlog). Priorisointia tehdään myös sitä mukaa, kuin uusia vaatimuksia saadaan selville, esimerkiksi jokaisen iteraation jälkeen. Iteraation aikana kehittäjät voivat tavallisesti itse päättää, missä järjestyksessä käyttäjätarinat ja niitä vastaavat tehtävät toteutetaan. Tässä yhteydessä tehtävää priorisointia ei tässä käsitellä.

Tuotepäällikkö on päävastuussa ominaisuuksien ja käyttäjätarinoiden valinnasta tuotetyölistaan. Leffingwellin (2011, 208) mukaan tuoteomistaja puolestaan vastaa tuotetyölistassa olevien osien priorisoinnista. Ketterien menetelmien mukaan hänellä on lopullinen ratkaisuvalta sen suhteen, mitkä valitaan iteraation työlistaan. Näihin yleisperiaatteisiin on kuitenkin poikkeuksia. Rachevan, Danevan, Sikkelin ja Wieringan (2010) mukaan pienissä projekteissa kehittäjät tekevät useimmiten päätökset vaatimusten priorisoinnista, koska asiakkaalla ei ole välttämättä riittävää ymmärrystä kohdealueesta, eikä asiakasorganisaatiolla ole varaa pitää asiakasta projektissa mukana kokopäiväisesti. Kehittäjillä on usein paljon kokemusta ja tietämystä kohdealueesta sekä kehittämisestä, joten asiakas voi jättää priorisoinnin heidän vastuulle. Näissä tapauksissa asiakkaan ja kehittäjien välillä tulee vallita hyvä luottamus.

Kirjallisuudessa on esitetty erilaisia tapoja suorittaa priorisointia ja priorisointiperusteita (Racheva ym. 2010; Harris & Cohn 2006; Leffingwell 2011). Seuraavassa käsitellään niitä lähemmin.

Rachevan ym. (2010) mukaan joissakin projekteissa käytäntönä on, että vaatimuksia ei priorisoida niiden tuottaman arvon perusteella, vaan sen perusteella, kuinka paljon arvoa tuottava vaatimus vähentäisi ohjelmiston arvoa, jos sitä ei toteuteta. Näitä Racheva ym. (2010) kutsuvat negatiiviseksi arvoksi (engl. negative value). Tärkeän ominaisuuden puuttuminen voi aiheuttaa ongelmia liiketoiminnalle ja liiketoimintaa tukeville prosesseille. Pienissä projekteissa asiakas haluaa rahoilleen mahdollisimman paljon vastinetta. Projekteissa, joissa resurssit ovat rajalliset, vaatimukset priorisoidaan siten, että vain korkeimman prioriteetin saaneet vaatimukset toteutetaan ohjelmistoon. Kun taas suurissa ja keskisuurissa projekteissa korkeimman prioriteetin vaatimukset valitaan seuraavaan iteraatioon toteutettavaksi. (Racheva ym., 2010)

Harrisin ja Cohnin (2006) ehdottama tapa priorisointiin keskittyy ominaisuuksien aiheuttamaan muutoskustannuksiin. Harrisin ja Cohnin (2006) mukaan asettamalla jokaiselle ominaisuudelle arvioitu muutoskustannus (engl.

expected cost of change, ECC), voidaan arvioida, kuinka todennäköisiä muutokset ovat ja mitkä ovat ominaisuuden kustannukset. Ominaisuudet luokitellaan ECC:n perusteella matalaksi tai korkeaksi. Osa järjestelmälle vaadituista ominaisuuksista voi olla muuttumattomia, jolloin niiden muutoskustannukset ovat pienet ja ECC-arvo matala. Osa ominaisuuksista taas on usein alttiita muutoksille ja niiden muutoskustannukset ovat korkeat ja ECC-arvot suuria. (Harris & Cohn, 2006)

Harrisin ja Cohnin (2006) mukaan ominaisuudet priorisoidaan toteuttamalla ensin pienen ECC-arvon ominaisuudet ja etenemällä askel kerrallaan korkean ECC-arvon ominaisuuksiin. Hyötynä on projektin pienentyneet kehityskustannukset. Huomioitavana asiana on Harrisin ja Cohnin (2006) mukaan, kuinka paljon tietoa tiimi saa ominaisuuksien toteuttamisen kautta. Tämäkin seikka vaikuttaa siihen, miten vaatimukset priorisoidaan. ECC-arvot voivat muuttua niissä ominaisuuksissa, jotka on priorisoitu toteutettavaksi myöhemmin. Harris ja Cohn (2006) ehdottavat, että edellisissä iteraatioissa saavutettu tietämys käytetään seuraavan iteraation suunnitteluun ja siinä toteutettavien vaatimusten priorisointiin. Suunnittelua ja priorisointia ei tarvitse tehdä pitkälle tulevaisuuteen, vaan riittää, kun tiedetään mitä seuraavassa iteraatiossa tehdään. (Harris & Cohn, 2006)

Leffingwellin (2011, 209) mukaan työlistan priorisointiin yleisellä tasolla vaikuttavat useat erilaiset tekijät, kuten vaatimukseen liittyvät käyttäjän hyödyt, toteuttamisriskit, toteuttamiskustannukset ja taloudelliset saatavat. Ominaisuuksien priorisointi ohjelmistotasolla perustuu taloudelliseen hyötyyn. Kuitenkin priorisointi seuraavaan iteraatioon on helpompaa kuin edellä kuvatuissa tapauksissa. Priorisointia helpottavat seuraavan iteraation tavoitteet, edellisen iteraation tulokset, samankokoiset tehtävät ja priorisoinnin rajaus vain seuraavaan iteraatioon. (Leffingwell, 2011, 209)

Vaikeissa ja monimutkaisemmissa tapauksissa vaatimusten priorisointi ei ole yhtä helppoa. Leffingwell (2011, 210) ehdottaa kolmea tapaa arvioida työlistan vaatimuksia (Taulukko 6): yksittäisen käyttäjän saama riippumaton (engl. independent) hyöty, iteraation hyöty ja riskin minimointi. Ensimmäisessä tavassa käyttäjätarinoita vertaillaan käyttäjän niistä saaman hyödyn näkökulmasta. Toiseksi voidaan tarkastella, kuinka hyvin tiimi saavuttaa asetetut tavoitteet iteraation aikana. Kolmas tekijä huomioi informaation, jonka avulla voidaan pienentää riskiä tulevissa kehitystehtävissä. Arvioimalla mainittuja tekijöitä voidaan priorisointipäätösten tekemistä helpottaa. Esimerkiksi taulukossa 6 ensimmäisenä oleva Tarina US53 on arvioitu olevan riippumaton arvolla viisi (asteikko 1-9), mikä tarkoittaa, että käyttäjän saama hyöty tarinasta on melko suuri. Toiseksi on arvioitu tiimille asetettujen tavoitteiden toteutumista arvolla viisi, mikä tarkoittaa, että tiimillä on hyvä mahdollisuus päästä asetettuun tavoitteiseen. Kolmanneksi on arvioitu tarinasta löydetyn informaation pienentävän riskiä kuuden pisteen arvoisesti tulevissa kehitystehtävissä. Kun jokainen arvo lasketaan yhteen, saadaan summa, joka kertoo tarinan US53 arvon olevan 16. (Leffingwell, 2011, 210)

TAULUKKO 6 Arvioidut ja lajitellut työlistan kohteet. (Leffingwell, 2011, 210)

Lajiteltu	Riippumaton	Iteraatio	Riskin minimointi	Painotus
Tarina US53	5	5	6	16
Piikki S43	2	5	7	14
Tarina US17	6	4	2	12
Tarina US32	8	2	1	11
Vika DE311	6	2	1	9

Vaatimusten priorisoinnissa voidaan käyttää myös MoSCoW-tekniikkaa (Hatton, 2008). Tämän tekniikan avulla vaatimuksia voidaan arvioida niiden tärkeyden perusteella. MoSCoW nimitys tulee englanninkielen sanoista Must, Should, Could ja Wont.

Must-arvon saaneiden vaatimusten on ehdottomasti oltava mukana järjestelmässä eikä niistä neuvotella. Mikäli Must -vaatimuksia ei ole toteutettu järjestelmässä, katsotaan projekti epäonnistuneeksi. Siksi onkin tärkeää, että löydetään yhteisymmärrys siitä, mitkä ovat Must-luokituksen arvoisia vaatimuksia. Näiden vaatimusten tulisi muodostaa yhtenäinen ryhmä, jossa ovat huolellisesti valitut vaatimukset. Should-vaatimukset ovat tärkeitä ja niiden olisi hyvä olla mukana järjestelmässä. Ilman Should-arvon mukaisia vaatimuksia järjestelmä toimii tiettyyn pisteeseen asti ongelmitta. Could-vaatimukset lisäävät liiketoiminta-arvoa, mutta eivät ole välttämättömiä. Wont-vaatimukset eivät ole ensimmäisenä toteutettavien listalla, joten ne voivat odottaa vuoroaan tulla toteutetuiksi tulevissa julkaisuissa. Wont-vaatimusten määrittäminen on kuitenkin yhtälailla tärkeitä. Hyvin tehty lista Wont-vaatimuksista auttaa yhteisymmärryksen luomisessa sidosryhmien välillä, koska jokaisen ryhmän vaatimuksia voidaan lisätä Wont-listalle. Tiimi tietää, millaisia vaatimuksia järjestelmään voidaan lisätä myöhemmässä vaiheessa ja kuinka ne täytyy huomioida nykyhetkellä. Myöhemmässä vaiheessa toteutettavien vaatimusten osalta voidaan tehdä alustavaa suunnittelua. (Hatton, 2008; Coley consulting, 2012)

Kano-mallia (Lacey, 2012) käyttäen priorisointi tehdään asiakastyytyväisyyden mukaisesti. Tässä mallissa keskeisenä asiana on vaatimusten priorisointi, joka pyrkii erottelemaan tärkeät ominaisuudet vähemmän tärkeistä asiakastyytyväisyyden näkökulmasta. Kano-mallissa kaikilta sidosryhmiltä kysytään toiminnallisten ja ei-toiminnallisten vaatimusten osalta kysymyksiä, joihin vastaus valitaan listasta. Vastaus voi olla: pitää (engl. like), odottaa (engl. expect), puolueeton (engl. neutral), hyväksyy (engl. live with) ja ei pidä (engl. dislike). Vastaukset kootaan taulukkoon, josta nähdään, kuinka tärkeäksi kukin vaatimus osoittautuu. Vaatimukset saavat merkinnän, joka kuvaa vaatimusten priorisointitärkeyttä. Merkinnät Kano-mallissa ovat: Exciters (innostavat), Baseline (lähtökohta), Linear (suoraviivaiset), Indifferent (keskinkertaiset), Questionable (kyseenalaiset) ja Reverse (käänteiset). (Lacey, 2012)

Exciter-vaatimukset ovat odottamattomia vaatimuksia, jotka nousevat esille projektin edetessä asiakkailta saadun palautteen kautta. Tämän luokituksen saavat vaatimukset ovat sellaisia, jotka eroavat kilpailijoiden tuotteiden ominaisuuksista. Baseline-vaatimukset vastaavat MoSCoW-tekniikan Must-vaatimuksia, jotka ovat kaikkein tärkeimmät vaatimukset ja sijoittuvat prio-

risointilistan kärkipäähän. Linear-vaatimukset ovat toiseksi tärkeimmät vaatimukset ja ne vaikuttavat suuresti asiakastyytyväisyyteen. Näitä vaatimuksia kutsutaan myös suorituskykyvaatimuksiksi. Indifferent-vaatimukset ovat vähiten kiinnostavia asiakkaan näkökulmasta ja ne tuottavat vähiten liiketoiminta-arvoa. Mikäli vaatimuksille tulee arvoksi Questionable tai Reverse, on kysymyksissä, vastauksissa tai laskennassa esiintynyt virhe. (Lacey, 2012)

Luke Hohman (2006) on kehittänyt ”Osta piirre” (engl. Buy a Feature) -tekniikan, missä erilaisista vaatimuksista tehdään lista ja jokaiselle vaatimukselle asetetaan hinta. Hinnan määrää vaatimuksen kehityskustannukset, asiakastyytyväisyys tai jokin muu oleellinen asia. Sidosryhmät ostavat vaatimuksia, jotka he haluavat toteutettavaksi seuraavaksi. Muutamat vaatimukset on tarkoituksella hinnoiteltu niin korkeaksi, etteivät yhdenkään sidosryhmän edustajan rahat riitä yksinään sitä ostamaan. Tämän tarkoituksena saada sidosryhmät yhdistämään rahansa, mikäli he haluavat vaatimuksen toteutettavaksi. Rahojen yhdistämisen ajatuksena on, että se kannustaa sidosryhmiä tekemään yhteistyötä ja neuvottelemaan, mitkä vaatimukset ovat kaikkein tärkeimpiä. Neuvottelu onkin tämän menetelmän ydin, koska mitä enemmän sidosryhmät neuvottelevat vaatimuksista, sitä tarkemmin tiedetään, mitkä ovat kunkin sidosryhmän osalta tärkeimmät vaatimukset. (Lacey, 2012)

Luke Hohman (2006) on kehittänyt myös ”Karsi tuotepuuta” (engl. Prune the Product Tree) -nimisen priorisointitekniikan. Tässä tekniikassa seinälle piirretään iso puu, jonka oksille asiakas laittaa lehtiä, jotka esittävät järjestelmälle haluttuja ominaisuuksia, jotka toteutetaan seuraavissa iteraatioissa. Paksut oksat kuvaavat päätoimintoja järjestelmässä ja puun latva esittää toimintoja, jotka ovat mukana tämänhetkisessä järjestelmässä. Esimerkiksi asiakas voi laittaa ominaisuuksia kuvaavia lehtiä ydintoimintojen kohdalle enemmän kuin muualle. Tavoitteena on saada tasapainoinen puu, jossa vaatimukset on priorisoitu, kuitenkaan unohtamatta ominaisuuksia, jotka täydentävät tärkeimpiä vaatimuksia. Tavoitteena on antaa asiakkaalle mahdollisuus osallistua päätöksen tekoon tarkastelemalla vaatimuksia kokonaisvaltaisesti. (Lacey, 2012)

Seuraava priorisointitekniikka on Karl Weigersin (1999) kehittämä suhteellinen painotus (engl. relative weighting). Tekniikka perustuu sidosryhmien antamaan palautteeseen sekä tiimin tekemään arvioon vaatimusten priorisoinnista. Kahden edellä mainitun priorisointitekniikan mukaisesti myös suhteellinen painotus edellyttää tuoteomistajan aktiivista osallistumista priorisointiin. Tekniikassa vaatimuksille annetaan erilaisia painotuksia sen perusteella, kuinka paljon hyötyä käyttäjälle vaatimuksesta on valmiissa järjestelmässä. Tekniikassa myös arvioidaan sitä, millainen haitta käyttäjälle aiheutuu, jos vaatimusta ei toteuteta järjestelmässä. Lisäksi arvioidaan kustannukset ja riskit prosentteina. Kun asiakas on arvioinut hyödyt, haitat, kustannukset ja riskit, on tiimin vuoro tehdä oma arvio arvoasteikolla 1-9. Tiimi arvioi kustannukset ja riskit vaatimusten toteuttamiselle. Tiimi arvioi myös vaatimusten toteuttamisen vaikeustason. Arvioiden tekemisen jälkeen lasketaan jokaiselle vaatimukselle tulos, joka määrittää sen prioriteetin. Mitä suurempi tulos, sitä tärkeämpi vaatimus on. Arviointi tehdään aina uudestaan, kun työlistaan lisätään uusia vaatimuksia tai kun

käyttäjätarinaa tarkennetaan. Työlistan priorisointi on jatkuva prosessi koko projektin ajan. (Lacey, 2012)

Joskus projektin aikataulu voi osoittautua liian tiukaksi, kaikesta arvioinnista ja suunnittelusta huolimatta. Kun projekti on jäljessä aikataulusta, joutuu projektipäällikkö pohtimaan, miten järjestelmä saadaan valmiiksi ajoissa. Näissä tapauksissa hyvä ratkaisu on Cohnin (2010, 295) mukaan pienentää projektin laajuutta (engl. scope). Osa esille saaduista ominaisuuksista jätetään pois, jotta projekti olisi valmis sille asetettuun päivämäärään mennessä. Mikäli työlista on priorisoitu oikein, tärkeimmät ominaisuudet on jo toteutettu. Näin viimeiseksi priorisoidut tehtävät voidaan jättää tekemättä ja järjestelmä valmistuu ajoissa. (Cohn, 2010, 295)

5.5 Vaatimusten laatukriteerit

Kuten perinteisessä vaatimusmäärittelyssä (vrt. Taulukko 3), myös ketterässä vaatimusmäärittelyssä vaatimuksille on esitetty laatukriteereitä. Seuraavassa kuvataan ensin Leffingwellin (2011) esittämä INVEST-malli käyttäjätarinoiden arvioimiseksi ja sen jälkeen Duncanin (2001) laatukriteerejä XP:n yhteydessä määritellyille vaatimuksille.

Leffingwellin (2011) INVEST-mallin nimi on lyhenne sanoista Independent, Negotiable, Valuable, Estimable, Small sekä Testable. Ensimmäinen laatukriteeri on siis *riippumattomuus* (engl. independent). Tämä tarkoittaa, että käyttäjätarina muodostaa muista tarinoista riippumattoman kokonaisuuden, jota voidaan kehittää, testata ja julkaista (engl. delivered). Käyttäjätarinat muodostavat yleensä suuremman toiminnallisen kokonaisuuden, jossa erilaiset toiminnot ovat riippuvaisia toisistaan. Yksittäiset toiminnot toimivat itsenäisesti ja lisäävät ohjelmiston arvoa. (Leffingwellin, 2011, 106) Käyttäjätarinat voivat sisältää myös hyödytöntä riippuvuutta, joko toiminnallista tai teknistä. Hyödytön riippuvuus ilmenee tapauksissa, joissa osa käyttäjätarinasta ei muodosta itsenäistä kokonaisuutta ilman siihen liittyvää osaa. Ratkaisuna ongelmaan poistetaan riippuvuus jakamalla käyttäjätarina niin, että jaetun tarinan osat muodostavat kaksi itsenäistä kokonaisuutta (Leffingwellin, 2011). Riippumattomuus on selkeä etu, koska ohjelmiston kehittämistä voidaan jatkaa keskeyttämättä.

Toisena laatukriteerinä on vaatimusten *neuvoteltavuus* (engl. negotiable). Leffingwellin (2011) mukaan käyttäjätarinoiden tulee mahdollistaa neuvottelut vaatimuksista, jotta vaatimuksia voidaan testata, kehittää ja hyväksyä. Tarkkoja sopimuksia tarkasti määritellyistä vaatimuksista ei ole, vaan niistä voidaan neuvotella yhteistyötä tehden ja palautetta antaen. Neuvotteluprosessi käydään liiketoiminnasta vastaavien ja tiimin välillä. Neuvottelussa huomioidaan liiketoiminnan tärkeys ja laillisuus. Yhteistyön ja tiimityön merkitys on suuri ongelmanratkaisussa ja sen rooli korostuu neuvottelun yhteydessä. Neuvottelu käyttäjätarinoista mahdollistaa paremman arvioinnin, koska vaatimuksia ei ole tarkasti määritelty. Myös tiimi ja liiketoiminnan edustajat voivat tehdä komp-

romisseja toimintojen ja julkaisupäivien osalta. Tiimi saa enemmän vapauksia toteuttaa vaatimukset, koska käyttäjätarinat ovat joustavia. (Leffingwell, 2011, 107)

Leffingwellin (2011) mukaan kolmas laatukriteeri, *arvo* (engl. value), on tärkein, koska tiimin tehtävänä on luoda mahdollisimman paljon arvoa tietyssä ajassa ja tietyillä resursseilla. Jokaisen käyttäjätarinan tulee antaa hyötyä ja arvoa sidosryhmille. Työlista priorisoidaan sen perusteella, kuinka paljon tiimi pystyy antamaan arvoa ja hyötyä listassa olevilla tehtävillä. Haasteena onkin, kuinka tiimi onnistuu kirjoittamaan käyttäjätarinoista mahdollisimman paljon arvoa tuovia. Tarinoiden tulisi olla lyhyitä ja ulottua arkkitehtuuriin asti. Näin arvon esittäminen käyttäjille käy selkeämmäksi. Käyttäjän vuorovaikutus järjestelmän kanssa on yleisin huomion kohde arvon suhteen. Leffingwellin (2011) huomauttaa, että joskus arvo kannattaa suunnata asiakkaan edustajaan tai tärkeimpään sidosryhmään. Teknisen ratkaisun arvoa voi olla vaikea selittää sidosryhmille. Ratkaisuna tähän ongelmaan Leffingwellin (2011) esittää, että tekninen ratkaisu esitetään käyttäjätarinan muodossa, jolloin sen tuottama arvo käy asiakkaalle ymmärrettävämmäksi.

Neljäntenä laatukriteerinä INVEST-mallissa on *arvioitavuus* (engl. estimable). Oikein kirjoitetusta käyttäjätarinasta tiimi pystyy arvioimaan sen vaatiman toteutusajan ja työmäärän helposti. Vähimmäisarvio, minkä tiimin tulisi tehdä, on pystyykö käyttäjätarinan toteuttamaan yhden iteraation aikana. Liian suuret ja epäselvät tarinat ovat vaikeampia arvioida. Tästä syystä tällaiset käyttäjätarinat tulee jakaa pienempiin osiin. Epäselviin tarinoihin voidaan käyttää piikkejä (engl. spikes) selventämään ongelmallisia kohtia. Käyttämällä edellä mainittuja ratkaisuita voidaan tarinoiden arvioitavuutta parantaa. Leffingwellin (2011) mukaan arviointi lisää tiimin yhteistä ymmärrystä siitä, mistä käyttäjätarinassa on kyse. Tämän lisäksi arviointi tuo esille puuttuvia hyväksyntä (engl. acceptance) -kriteereitä ja olettamuksia (Leffingwell, 2011, 109).

Edelliseen liittyy oleellisena laatukriteerinä käyttäjätarinan koko, jonka tulisi olla riittävän *pieni* (engl. small). Käyttäjätarina tulisi olla toteutettavissa yhdessä iteraatiossa. Lyhyet käyttäjätarinat mahdollistavat ketteryyden sekä tuottavuuden, koska prosessin nopeutuessa vaatimukset voidaan toteuttaa nopeammin. Tässä yhteydessä Leffingwell (2011) nostaa esille kaksi asiaa: lisääntynyt suoritusteho (engl. throughput) ja vähentynyt monimutkaisuus. Lisääntynyt suoritusteho tarkoittaa nopeammin toteutettavia käyttäjätarinoita niiden pienen koon ansiosta. Kun tarinat ovat tietyn kokoisia, pysyy niiden toteuttamiseksi käytetty aika samana, eli noin yhtenä iteraationa. Tämän ansiosta arviointi voidaan tehdä tarkasti. Vähentynyt monimutkaisuus myös lisää ketteryyttä, koska pienet käyttäjätarinat on helpompi ymmärtää ja sitä kautta nopeampia toteuttaa. Erityisesti tämä ilmenee testauksessa, jossa monimutkaisuuden vähentyessä toiminnoissa testaukseen käytettävä aika pienenee. (Leffingwell, 2011, 110)

Viimeisenä laatukriteerinä INVEST-mallissa on *testattavuus* (engl. testable). Käyttäjätarinoiden pitää olla helppoja testata. Käyttäjätarinat, joita ei voi testata, ovat huonosti laadittuja, liian monimutkaisia tai riippuvat muista tarinoista.

Tiimit voivat varmistaa käyttäjätarinoiden testattavuuden kirjoittamalla testin ensimmäiseksi, kuten XP:ssä, jossa automatisoidut yksikkötestit kirjoitetaan ennen koodin kirjoittamista. (Leffingwell, 2011, 111)

Duncan (2001) on analysoinut XP:n vaatimusmäärittelyprosessia vaatimusten spesifiointidokumentille asetettujen 24 laatukriteerin perusteella. Duncan (2001) on arvioinut XP:ssä tuotettavien vaatimuksien laatua teoreettisesta näkökulmasta. Osan kriteereistä XP täyttää hyvin, mutta osan huonosti (Taulukko 7). Taulukossa 7 olevat ”+” -merkit tarkoittavat, että XP parantaa vaatimuksia, kun taas ”-” -merkki tarkoittaa heikennystä. Laatukriteerit, jotka on merkitty ”+/-” -merkillä, tarkoittavat, että XP osittain vahvistaa sekä osittain heikentää vaatimusten laatua. Kysymysmerkki ”?” osoittaa, että kyseistä kriteeriä XP ei huomioi. (Duncan, 2001) Seuraavaksi laatukriteereitä käsitellään tarkemmin Duncanin (2001) esityksen mukaisesti.

TAULUKKO 7 24 laatukriteeriä vaatimusten spesifiointidokumentille (Duncan, 2001, 20)

Laatukriteeri	**Merkki**	**Laatukriteeri**	**Merkki**
Yksiselitteinen	+	Sähköisesti tallennettu	+/-
Kattava	-	Suoritettava/selittävä	+/-
Virheetön	+	Selitetty suhteellisen tärkeäksi	+
Ymmärrettävä	+	Selitetty suhteellisen vakaaksi	?
Vahvistettavissa	+	Selitetty versio	+
Sisäisesti johdonmukainen	+/-	Yksittäinen	-
Ulkoisesti johdonmukainen	+/-	Riittävän yksityiskohtainen	?
Toteutuskelpoinen	+	Täsmällinen	?
Tiivis	+	Uudelleenkäytettävä	?
Suunnittelusta riippumaton	+/-	Jäljitetty	?
Jäljitettävissä	?	Organisoitu	?
Muokattavissa	+	Läpileikkaava	?

Duncanin (2001) mukaan XP:ssä vaatimukset ovat *yksiselitteisiä, virheettömiä* ja *ymmärrettäviä*. Tämä johtuu asiakkaan mukanaolosta. Asiakas yleensä kirjoittaa käyttäjätarinat liiketoimintanäkökulmasta, joten vaatimukset ovat virheettömiä. Vaatimukset ovat virheettömiä silloin, kun ne esittävät todellisia vaatimuksia järjestelmälle. Vaatimusten *muokattavuus* on XP:ssä mahdollista inkrementaalisen kehittämisen ansiosta. Asiakas voi *muuttaa* vaatimuksia melkein missä projektin vaiheessa tahansa, mikäli tarve sellaiseen on. Vaatimukset on myös *priorisoitu (selitetty tärkeäksi)* asiakkaan toimesta. Näin kehittäjät tietävät, mitkä vaatimukset tulevassa iteraatiossa toteutetaan. XP:ssä ohjelmistosta julkaistaan toimivia osia, jotka lisäävät liiketoiminta-arvoa. Julkaistut osat ovat *toteutuskelpoisia*. Osaa ohjelmistosta / järjestelmästä ei voida ottaa käyttöön, koska sitä ei ole vielä toteutettu. Duncanin (2001) mukaan *suunnittelusta riippumattomat* (engl. design independent) vaatimukset ovat epäkäytännöllisiä johtuen oliopohjaisista kehitysmenetelmistä. Käyttäjätarinasta osa voi olla suunnittelusta riippumaton, mutta vaatimuksen mukana oleva yksikkötesti voi olla järjestelmästä riippuvainen. Vaatimuksia ei XP:ssä *tallenneta sähköiseen muotoon*, vaan ne kirjoitetaan käsin korteille. Duncanin (2001) mukaan asiakkaan on hel-

pompi tehdä muutoksia järjestelmään, kun vaatimukset on kirjoitettu kortteille eikä sähköiseen muotoon. Koska XP:ssä koodin tuottaminen on etusijalla, vaatimusdokumentit ovat *tiiviitä* ja *kattavat* vain kaikkein tärkeimmät asiat. Duncanin (2001) mukaan on mahdollista, että vähäisen analyysin myötä järjestelmään voi jäädä puutteita. Havaitut puutteet on mahdollista korjata myöhemmin projektin aikana.

Myös XP:ssä käytettäville piikeille (engl. spikes) on laadullisia kriteereitä, kuten arvioitavuus, demonstroitavuus ja hyväksyttävyys. Piikit arvioidaan ja mitoitetaan käyttäjätarinoiden tapaan sopiviksi yhteen iteraatioon kerrallaan. Piikkien tuottaman informaation avulla tiimi voi vähentää epävarmuutta ja tunnistaa uusia tarinoita. Piikkien antamat tulokset ovat demonstroitavissa tiimille ja muille sidosryhmille. Koska piikkien antamat tulokset voidaan esitellä sidosryhmille, he voivat osallistua päätöksien tekemiseen sekä jakaa päätösten vastuuta tasapuolisesti. Tuoteomistaja hyväksyy piikit, mikäli piikki täyttää yllä esitetyt hyväksymiskriteerit. Piikit ovat enemmän poikkeus kuin sääntö, ja niihin sisältyy riskejä. Leffingwell (2011) huomauttaa, että piikkejä tulisi käyttää harkiten silloin, kun käyttäjätarinoiden jakaminen ei riitä. Piikit ovat perusteltuja silloin, kun kyseessä on suurempi ja kriittinen epävarmuus. (Leffingwell, 2011, 115–116)

5.6 Dokumentointi

Yleisesti ketterissä menetelmissä on sellainen käsitys, ettei kattavaa dokumentointia ole järkevää tehdä, koska sen kirjoittaminen ei ole kustannustehokasta (Paetsch ym., 2003). Kuitenkin esimerkiksi Crystal (Cockburn, 2002), Scrum (Schwaber & Sutherland, 2010) ja DSDM (Stapleton, 1997) tukevat dokumentoinnin kirjoittamista, joskaan ei yksityiskohtaisesti. Se, kuinka tarkasti projektin dokumentointi toteutetaan, jää kehitystiimin ratkaistavaksi. (Paetsch, ym., 2003). Cohnin (2010, 238) mukaan ketterässä kehittämisessä on tärkeää löytää sopiva tasapaino dokumentaation ja keskusteluiden välillä. Rubin ja Rubin (2011) huomauttavat, että analyysivaiheen dokumentaatio olisi tarpeellista ketterissä menetelmissä. Rubinin ja Rubinin (2011) mukaan analyysivaiheen käsitemallit (engl. conceptual models), joita yleensä ei tehdä ketterissä menetelmissä, ovat hyödyllisiä kehittäjien ja sidosryhmien välisessä kommunikaatiossa. Ketterässä vaatimusmäärittelyssä yleisesti käytettyinä vaatimusten dokumentointitapoina ovat käyttäjätarinat, testitapaukset, tehtävät ja työlista. Lisäksi vaatimusdokumenttia käytetään laki- tai sopimussyistä selventämään vaatimuksia (Cohn, 2010, 238). Leffingwellin (2011) mukaan erityisesti käyttäjätarinat ovat tärkein dokumentointimuoto ketterässä vaatimusmäärittelyssä.

Dokumentoinnin tarpeellisuutta voidaan perustella tiimin koolla. Isossa kehitystiimissä hyvästä dokumentoinnista on enemmän hyötyä kuin pienessä tiimissä. Tämä johtuu siitä, että ilman dokumentointia aikaa kuluu asioiden selittämiseen jokaiselle tiimin jäsenelle erikseen. Pienemmässä tiimissä asia on päinvastoin. Ketterissä menetelmissä dokumentaatio on ajantasaista johtuen

sen pienestä määrästä, ja se keskittyy järjestelmän tärkeimpien osa-alueiden kuvaamiseen. (Paetsch, ym., 2003).

Hajautetuissa projekteissa dokumentaation tarve lisääntyy merkittävästi, koska tiimin jäsenet eivät työskentele yhdessä. Hajautetuissa projekteissa käytetään formaaleita suunnitteludokumentteja. Hajautetun tiimin tekemät suunnitteludokumentit validoidaan pääorganisaatiossa ennen niiden toteuttamisen aloittamista. Muita dokumentteja, joita hajautetuissa projekteissa käytetään, ovat: kyselyjäljitys (engl. query tracker), päätösjäljitys (engl. decision tracker) ja Scrum-tapaamisten pöytäkirjat (engl. minutes of scrum meeting). (Sureshchandra & Shrinivasavadhi, 2008)

Rameshin ym. (2008) mukaan vaatimusten dokumentointi jää usein erilaisten toimintojen toteuttamisen varjoon. Tiimi keskittyy dokumentoinnin sijasta esillesaatujen vaatimusten toteuttamiseen, eikä niinkään niiden dokumentointiin. Yhtenä syynä Rameshin ym. (2008) mukaan on se, että tiimi työskentelee nopeassa tahdissa, jolloin aikaa vaatimusten dokumentointiin ole.

Nawrocki, Jasinski, Walter ja Wojciechowski (2002) ehdottavat, että XP:tä on mahdollista laajentaa lisäämällä vaatimusten dokumentointi yhdeksi osa-alueeksi. Varsinaista dokumentaatiota XP:ssä ei tehdä, vaan kirjoitettu koodi ja testitapaukset ovat ainoat luodut dokumentit. Nawrockin ym. (2002) mukaan vaatimusten dokumentointi voidaan siirtää testaajan vastuulle, koska hyväksyntätestitapaukset ovat samalla abstraktiotasolla kuin vaatimukset, joten testaaja voi analysoida sekä dokumentoida vaatimukset. Nawrocki ym. (2002) perustelevat ehdotustaan sillä, että testaaja hallinnoi testitapauksia, joten hän voi myös samalla huolehtia vaatimusten hallinnasta. Testitapaukset voidaan linkittää vaatimuksiin, jolloin vaatimusten muutokset voidaan helposti yhdistää niitä vastaaviin testitapauksiin. Vaatimusten dokumentointi ei vaikuta haitallisesti muihin XP:n käytäntöihin vaan vahvistaa XP:tä. (Nawrocki ym., 2002)

Ketterässä vaatimusmäärittelyssä tiimin käyttämät dokumentit ovat yksinkertaisia. Näiden dokumenttien avulla kehitystiimin on helppo ja nopea toteuttaa tarvittavat mallit, jotka kattavat jokaisen sidosryhmän vaatimukset. Yksinkertaiset vaatimusdokumentit sopivat myös ketterään ajattelutapaan. (Leffingwell, 2011, 55).

Vaatimusten dokumentaatio on jatkuva prosessi, jossa vaatimusten yksityiskohdat tarkennetaan juuri ajoissa (engl. just-in-time), kun niiden toteuttaminen alkaa. Etenkin ketterässä vaatimusmäärittelyssä vaatimuksia on vaikea tietää ja dokumentoida etukäteen, koska vaatimukset muuttuvat usein sidosryhmien ymmärryksen lisääntyessä ja kohdealueen muutoksien myötä. (Zhang ym., 2010)

5.7 Mallintaminen vaatimusmäärittelyn tukena

Ketterän lähestymistavan on katsottu haluavan tehdä selvän eron perinteiseen lähestymistapaan, jolle on tyypillistä monenlaisten mallien tekeminen. Keskeisimpänä mallintamiskielenä on viime vuosina ollut UML (engl. Unified Mode-

ling Language) (Booch, Rumbaugh & Jacobson, 1999). Suhtautuminen mallintamiseen on ollut kuitenkin mielipiteitä jakava aihe ketterän kehittämisen kannattajien keskuudessa.

Kinnusen (2007) tekemän selvityksen mukaan Fowler ja Kendall (2004, 8, 47, 68) ja Astels, Miller ja Noval (2002, 142) suosittelevat UML:n käyttöä XP:n yhteydessä. Astels ym. (2002, 144) pitävät luokka- ja sekvenssikaavioita hyödyllisinä. Rumpen ja Schröderin (2002) mukaan 35 % XP-projekteista käytettiin UML:ää (Kinnunen 2007, 63). Saman selvityksen mukaan FDD:ssä (Palmer & Felsing, 2002) kohdealueen yleismallin tekemisessä voidaan käyttää UML-luokkakaaviota. Luokkakaavioiden lisäksi voidaan käyttää tarvittaessa myös sekvenssikaavioita. DSDM:ssä (Stapleton, 1997) ohjeistetaan käyttämään kontekstikaaviota kohdealueen ymmärtämiseen, ER-kaaviota käsitteellisen tietomallin luomiseen sekä tietovirtakaaviota ja käyttötapauskaaviota liiketoimintaprosessien esittämiseen (Kinnunen, 2007, 79).

Leffingwellin (2011, 357) mukaan aktiviteettikaavio on hyvä keino vaatimusten analysoimiseksi kaavion tunnettavuuden kautta. UML-notaatio on helppo ymmärtää kuvallisen esitystavan ansiosta. Toinen hyöty, minkä aktiviteettikaavio tarjoaa, on sen yksiselitteisyys. Pseudokoodi yhdistää kirjoitetun kielen informaation ja ohjelmointikielen syntaksin sekä kontrollirakenteen. Tekstin asettelu ja formaatti lisäävät yksiselitteisyyttä. Näiden asioiden yhdistelmän myötä monitulkintaisuuden mahdollisuus vähenee. Pseudokoodin etuna on, että ohjelmointia ymmärtämättömät sidosryhmät ymmärtävät käyttäjätarinan toimintojen logiikan. (Leffingwell, 2011, 359)

Päätöstaulut ja päätöspuut (engl. decision tables, decision trees) ovat avuksi silloin, kun käyttäjätarinassa on yhdistelmiä syötteille. Päätöstaulut auttavat yhdistelmien ymmärtämistä ja varmistavat jokaisen yhdistelmän mahdolliset järjestykset (Leffingwell, 2011, 360).

Leffingwellin (2011, 366) mukaan käyttötapauksia (engl. use cases) voidaan käyttää ketterässä vaatimusmäärittelyssä analysointiin ja spesifiointiin. Käyttötapaukset ovat enemmän perinteisen vaatimusmäärittelyn analysointitekniikka. Erityisesti käyttötapaukset ovat hyödyllisiä silloin, kun kehitettävä järjestelmä koostuu pienemmistä järjestelmistä.

Leffingwell (2011, 364) ehdottaa sekvenssikaaviota käytettäväksi prosessien mallintamiseksi. Sekvenssikaaviolla voidaan selvittää, millaisia viestejä erilaiset prosessit lähettävät toisilleen sekä missä järjestyksessä viestit lähetetään. Myös entiteettikaaviot (engl. entity-relationship diagrams) ovat hänen mielestään käyttökelpoisia silloin, kun käyttäjätarinat sisältävät järjestelmän sisältämän datan rakenteellisia kuvauksia. Entiteettikaaviot esittävät korkeantason arkkitehtuurisen kuvauksen esimerkiksi asiakkaiden tiedoista. (Leffingwell, 2011, 365)

5.8 Vaatimusten jäljitys

Vaatimusten jäljityksen tarkoituksena on vaatimusten elinkaaren seuraaminen ja esittäminen. Jäljityksen ansiosta vaatimuksen elinkaarta voidaan edetä sekä eteenpäin että taaksepäin. Ketterässä kehittämisessä jäljityksen onnistuminen edellyttää kehittäjien ja sidosryhmien välisten tärkeiden keskusteluiden tallentamista (Lee & Guadagno, 2003)

Zhangin ym. (2010) esittämän jäljityslinkistön avulla voidaan seurata, millaisia käyttäjätarinoita ja tehtäviä komponentit sisältävät tai millaisia testitapauksia mihinkin osaan järjestelmästä liittyy. Kehittäjät voivat seurata jäljityslinkistön avulla, mitkä sidosryhmien kanssa käydyt keskustelut, käyttäjätarinat ja vaatimukset liittyvät kirjoitettuun koodiin. Jäljityslinkistön avulla asiakkaat voivat seurata kehitysprosessin etenemistä tarkastelemalla, mitkä käyttäjätarinat ovat toteutettu järjestelmään. Vaatimusten dokumentoinnin yhteydessä jäljityslinkistöt sijoittuvat kahteen kategoriaan: linkistöt työkaluissa ja linkistöt työkalujen välillä. Projektin aikana käytetään erilaisia työkaluja kehittämisen tukena, jotka mahdollistavat jäljittämisen työkalulla tehtyihin dokumentteihin. Kehityksessä käytettyjen työkalujen välillä on tarpeellista olla yhteys, jotta erilaisten työkalujen sisältämät tiedot voidaan linkittää oikeisiin vaatimuksiin kokonaisuuden hallitsemiseksi. (Zhang ym., 2010)

Kommunikointi ja vuorovaikutus ovat ketterissä menetelmissä ydinasia, erityisesti käyttäjätarinoista käydyt keskustelut. Projektin koko määrittelee, kuinka kommunikaatio sidosryhmien välillä toteutetaan. Pienissä projekteissa kommunikointi vaatimuksista on vaivatonta, koska tiimin jäsenet ovat lähellä toisiaan, eikä erityisiä tekniikoita tarvita ongelmien ratkaisemiseksi. Suurissa projekteissa kommunikaatio on monimutkaisempaa. Tästä johtuen väärinymmärryksen ja virheiden mahdollisuus kasvaa. Erittäin tärkeän järjestelmän (esim. talous- ja sairaalajärjestelmät) vaatimuksissa ei saa olla virheitä eikä monitulkintaisuutta. Lisäksi vaatimuksien täytyy olla jäljitettävissä (Leffingwell, 2011, 356). Palavereissa käytyjä keskusteluita harvoin tallennetaan mihinkään, vaan ne säilyvät tiimin yhteisenä tietona. Tästä aiheutuu ongelmia suurissa ja hajautetuissa projekteissa, koska tehokkaan ja ajantasaisen kommunikaation ylläpitäminen käy hankalaksi sekä virhealttiiksi. (Lee ym., 2003) Kun dokumentoitu informaatio voidaan jäljittää kehitysprojektin aikana, se Leen ym. (2003) mukaan parantaa ohjelmiston laadunvarmistusta. Vaatimusten jäljitys on tärkeää ketterässä kehittämisessä, koska se mahdollistaa järjestelmän korkean laadun ja järjestelmien toteuttamisen ajoissa, lisää vastuuta, helpottaa muutoshallintaa (Espinoza & Garbajoa, 2011; Ghazarian, 2008). Lisäksi vaatimusten jäljitys helpottaa kommunikointia, tukee muutosten integrointia, varmistaa laadun ja vähentää väärinymmärryksiä (Espinoza & Garbajoa, 2011). XP:ssä jäljitys on tarpeellista suunnittelupelin (engl. planning game) aikana, jolloin tiimi ja asiakas neuvottelevat käyttäjätarinoista. Tarinoista kirjoitetaan testitapauksia, jotka toimivat vaatimuksina. Suunnittelun aikana neuvotellaan, miten vaatimukset toteutetaan. Näiden vaiheiden välillä tulisi olla yhteys, mikä mahdollis-

taisi jäljityksen. Mikä tärkeintä, yhteyksien semantiikan tulisi olla täsmällistä, jotta yhteydet viittaavat oikeisiin kohteisiin. (Espinoza & Garbajoa, 2011)

Vaatimusten jäljitykseen on useita erilaisia tekniikoita, mutta ongelmana niissä on Espinozan ja Garbajoan (2011) mukaan se, että ne perustuvat oletukseen valmiista vaatimusten spesifiointidokumentista, jota ketterässä kehittämisessä ei tehdä. Spesifiointidokumentin sijaan vaatimukset on dokumentoitu käyttäjätarinoiden tai käyttötapausten muodossa (Zhang ym., 2010). Lisäksi ketterissä menetelmissä vaatimusten jäljitys tehdään manuaalisesti. Koska vaatimukset, ominaisuudet ja käyttäjätarinat on yleensä kirjoitettu käsin erilaisille korteille, esimerkiksi XP:ssä käytetään CRC -kortteja, vaikeutuu vaatimusten jäljittäminen projektin koon kasvaessa. Erityisesti hajautetuissa tiimeissä vaatimusten jäljitys on kuitenkin tärkeää, koska kehittäjät joutuvat käyttämään samoja dokumentteja samaan aikaan, jolloin ongelmaksi muodostuu dokumenttien ylläpito. Suurissa tai monimutkaisissa projekteissa jäljityslinkistön hallinta vaikeutuu ja linkistön tarkkuus riippuu päivitysten määrästä. Huonosti ylläpidetty jäljityslinkistö ei toimi oikein ja yhteydet vaatimusdokumenttien välillä ovat virheelliset. Jäljitys viallisien yhteyksien vuoksi ei onnistu vaatimusten elinkaaren mukaisesti, vaan jäljittäminen voi ohjautua vääriin dokumentteihin. (Lee ym., 2003)

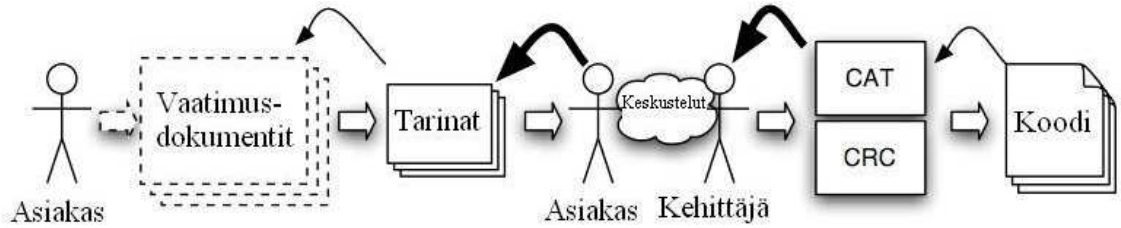
Edellisen lisäksi Zhang ym. (2010) toteavat, että kehittäjät ovat usein vastahakoisia osallistumaan jäljitysdokumentaation kirjoittamiseen, koska se vaatii manuaalista työtä. Pilkkoutuneen informaation yhdistäminen käsin tai automaattisesti on kuitenkin tärkeää. Etenkin suurissa projekteissa vaatimusten jäljityksestä on hyötyä, mutta dokumenttien käsin kirjoittaminen ja niiden ylläpito vaatii paljon työtä (Lee ym., 2003).

Zhangin ym. (2010) mukaan työkaluja vaatimusten dokumentointiin ja analysointiin on runsaasti. Zhang ym. (2010) luokittelevat vaatimustenhallintatyökalut neljään ryhmään: yleiskäyttöiset, yhteistyö-, vaatimustenhallinta- ja prototyyppityökalut. Näistä työkaluista prototyyppityökalut ovat Zhangin ym. (2010) mukaan sopivin ketterään vaatimusten dokumentointiin. Eräs prototyyppityökalu on Internet-sovelluksien toteuttamiseen kehitetty Vixtory (Zhang ym., 2010), joka mahdollistaa vaatimusten dokumentoinnin suoraan kohdesovellukseen. Myös sidosryhmät voivat osallistua kehittämiseen ja antaa palautetta projektin aikana Vixtoryn kautta. Erillistä vaatimusdokumenttia ei tarvitse tehdä, vaan vaatimukset ovat osa sovellusta. Vaatimusten jäljitys tapahtuu URL -osoitteiden välityksellä. Vixtory:ssa vaatimukset on kuvattu tekstimuotoon.

Ghazarian (2008) esittää vaatimustenjäljitykseen menetelmän, joka perustuu lähdekoodin rakenteeseen. Menetelmässä keskeisenä on vaatimuskomponentin jäljitysmalli (engl. design pattern). Tässä menetelmässä kattavaa vaatimusdokumenttia ei tarvita, mikä Ghazarian (2008) mukaan sopii hyvin ketteriin menetelmiin. Ainoa dokumentti on kehittäjien kirjoittama koodi. Rakenteen yhdenmukaisuus koodissa mahdollistaa jäljitysmallin käyttämisen vaatimusten jäljittämiseksi. (Ghazarian, 2008)

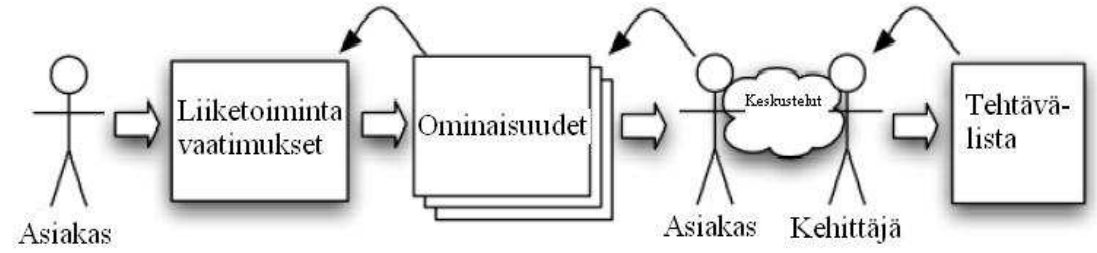
Leen ym. (2003) mukaan vaatimusten jäljitys onnistuu hyvin silloin, kun asiakkaan kanssa käyty keskustelu ongelmasta tarkennetaan vaatimusten spesi-

fioinniksi. Lee ym. (2003) esittävät yhdeksi tavaksi vaatimusten jäljittämiseksi FLUID: Echo -työkalun, jonka avulla dokumentteja voidaan luoda vapaasti. Koska ketterässä kehittämisessä on tapana käydä paljon keskusteluita, mahdollistaa FLUID: Echo näiden keskusteluiden informaation tallentamisen käyttäjätarinoiksi. Näistä dokumenteista voidaan tehdä uusia dokumentteja, joten niiden välillä täytyy olla jäljitettävä yhteys. FLUID: Echo mahdollistaa käyttäjätarinoiden tarkentamisen muita projektissa käytettäviä dokumentteja vastaaviksi. FLUID: Echo -työkalun avulla kehittäjät voivat tarkentaa asiakkaiden tarpeet toteutettaviksi vaatimuksiksi projektin edetessä. Kuviossa 7 on kuvattu, miten XP:ssä vaatimuksia voidaan jäljittää FLUID: Echo -työkalun avulla. Jäljitys onnistuu koodista lähtien askel kerrallaan aina alkuperäisiin vaatimusdokumentteihin asti.



KUVIO 7 Kokonainen jäljitysmalli koodista vaatimuksiin XP:ssä (Lee, ym., 2003, 58)

Kuviossa 8 on vaatimusten jäljitys kuvattu Scrum -menetelmän osalta ensin työlistasta asiakkaan ja kehittäjän välisiin keskusteluihin ja sieltä ominaisuuksiin ja lopuksi asiakkaan esittämiin liiketoimintavaatimuksiin asti. Leen ym. (2003) mukaan FLUID: Echo -työkalu mahdollistaan projektin hallinnan, ominaisuuksien esillesaannin liiketoimintavaatimuksista (engl. business condition requirements), keskustelut ominaisuuksien priorisoinnista ja tehtävälistan hallinnan. (Lee ym., 2003)



KUVIO 8 Vaatimusten jäljitys Scrum -menetelmässä (Lee, ym., 2003, 59)

5.9 Ketterän vaatimusmäärittelyn roolit

Ketterä vaatimusmäärittely koskee monia keskeisiä rooleja. Asiakas, projektipäällikkö ja kehittäjät ovat pääosan esittäjiä, ja heistä riippuu kehitysprojektin eteneminen ja onnistuminen. Ennen kehitysprojektin aloittamista on tarpeellista

kartoittaa mahdolliset sidosryhmät. Sidosryhmät voidaan jakaa Leffingwellin (2011, 120) mukaisesti kahteen luokkaan: järjestelmäsidosryhmään ja projektisidosryhmiin. *Järjestelmäsidosryhmään* kuuluvat järjestelmän pääkäyttäjät, pääkäyttäjien kanssa työskentelevät sekä järjestelmästä hyötyvät henkilöt. *Projektisidosryhmään* kuuluvat ovat budjetista ja aikataulusta vastaavat, järjestelmän kehittämisen tuoman ymmärryksen hyödyntäjät, järjestelmän markkinointiin, myyntiin, asennukseen ja ylläpitoon liittyvät henkilöt. Kehitystiimin tulee muodostaa molempien sidosryhmien esittämistä vaatimuksista yhtenäinen kokonaisuus (Leffingwell, 2011, 120). Seuraavaksi tarkastellaan lähemmin keskeisten roolien toimintaa ketterässä vaatimusmäärittelyssä.

Sillitti ja Succi (2005) toteavat, että *asiakkaan* läsnäolo kehitysprosessin aikana on hyvin merkityksellinen, ja sillä on huomattava vaikutus kehitysprojektin onnistumiseksi. Koska dokumentointi on vähäistä, korvaa asiakas tämän puutteen. Kehittäjät keskustelevat vaatimuksista asiakkaan kanssa ja kysyvät tarpeen mukaan selventäviä kysymyksiä vaatimuksista, jotta vältyttäisiin väärinymmärryksiltä. Asiakkaan antama palaute on tärkeää kehitystiimille, koska asiakas voi tunnistaa ja kertoa mahdollisista ongelmista tiimille. Tällä voidaan osaltaan varmistaa, että projekti pysyy aikataulussa. Mikäli asiakas ei ole riittävästi läsnä kehitysprojektissa, heikentyy ketterän kehittämisen tehokkuus (Sillitti & Succi 2005). Leffingwellin (2011, 122) mukaan sidosryhmien täytyy tehdä pysyviä ja oikea-aikaisia päätöksiä projektin laajuudesta ja vaatimusten priorisoinnista. Joskus asiakkaita on enemmän kuin yksi. Tällöin jokainen asiakas ei voi olla koko ajan läsnä projektin aikana. Nawrocki ym. (2002) ehdottavat, että testaaja / analysoija vastaa tällöin asiakkaan roolista. Koska testaaja / analysoija on päivittäin asiakkaiden kanssa tekemisissä, hän voi vastata kehittäjien esittämiin kysymyksiin asiakkaiden puolesta.

Nawrockin ym. (2002) mukaan XP:ssä asiakas edustaa loppukäyttäjää (engl. end user) osallistuen tiimin kanssa järjestelmän kehittämiseen toimien tietolähteenä tiimin esittämiin kysymyksiin sekä tekemällä liiketoimintaa koskevia päätöksiä. XP:ssä käytävä suunnittelupeli on asiakkaan tehtävistä tärkein. Suunnittelupelin aikana asiakas päättää tulevan julkaisun laajuuden ja mitkä tehtävät seuraavassa iteraatiossa toteutetaan. Oletuksena XP:ssä on, että asiakkaita on vain yksi projektia kohden. Näin varmasti onkin pienemmän kokoluokan projekteissa, mutta suuremmissa projekteissa asiakkaan edustajia voi olla useampi. Tällöin XP olettaa, että edustajilla on yhteneväinen ymmärrys projektiin liittyvissä kysymyksissä. Useampi asiakasedustaja voi aiheuttaa ongelmia, ja niihin olisi hyvä varautua ennakolta. Nawrocki ym. (2002) mainitsevat Sommervillen ja Sawyerin (1997) kuvauksen mukaisesti, että useat asiakkaan edustajat voidaan sisällyttää projektiin tunnistamalla sidosryhmät, keräämällä vaatimuksia erilaisista näkökulmista, huomioimalla eri osastojen edustajat ja varautumalla konfliktitilanteisiin (Nawrocki ym., 2002).

Johdon roolina on luoda toimivat olosuhteet kehitystiimin ja asiakkaan sujuvalle ja tehokkaalle kommunikoinnille. Muita tehtäviä johtajalle ovat sopimuksista neuvottelu, sopivien ihmisten valinta kehitystiimiin ja yhteistyön parantaminen (Sillitti & Succi, 2005). *Tuotepäällikön* (engl. product manager) vas-

tuulla on tuotteen menestys markkinoilla. Hänen roolinsa painottuu ohjelmistojen julkaisemiseen, määrittelyyn ja markkinointiin. (Leffingwell, 2011, 204). *Tuoteomistajalla* (engl. product owner) on tärkeä rooli kehitysprojektissa, sillä hän toimii edustajana kaikille sidosryhmille. Tärkeäksi roolin tekee se, että tuoteomistajan tehtävänä on saattaa vaatimukset kaikille sidosryhmille hyväksyttävään muotoon sekä ohjata sidosryhmien odotukset samaan päämäärään (Leffingwell, 2011, 121). Tuoteomistaja voi Leffingwellin (2011, 121) mukaan yhdistää sidosryhmien mielipiteet yhdeksi priorisoiduksi työlistaksi kahdella tavalla: auttaen (engl. facilitating) tai johtamalla (engl. leading) sidosryhmiä. Ensimmäisessä tavassa tuoteomistaja auttaa sidosryhmiä löytämään yhteisen ehdotuksen työlistaksi, jonka sidosryhmät voivat hyväksyä. Toisessa tavassa tuoteomistaja tekee päätökset sidosryhmien puolesta ottaen huomioon sidosryhmien asiantuntemuksen ja kokemuksen alalta, jolle järjestelmä on tarkoitettu. Markkinat, kilpailu tai muut trendit ohjaavat tuoteomistajan tekemiä päätöksiä vaatimusten suhteen (Leffingwell, 2011, 121). Eräs oleellinen asia on sidosryhmien tuntema luottamus tuoteomistajaa sekä kehitystiimiä kohtaan. Luottamus parantaa projektin etenemistä ja sidosryhmien välistä toimintaa (Leffingwell, 2011).

Kehitystiimin rooli ketterässä vaatimusmäärittelyssä on moninainen. Ensimmäiseksi jokainen kehitystiimin jäsen osallistuu vaatimusten keräämiseen ja neuvotteluun. Toiseksi kehittäjien täytyy olla aktiivisessa vuorovaikutuksessa asiakkaan kanssa saadakseen riittävästi palautetta. Kehittäjien ammattitaidon täytyy olla tarpeeksi korkea, ja heidän täytyy osata työskennellä ryhmässä ja kyetä kommunikoimaan asiakkaan ”omalla kielellä”. Kolmanneksi kehitystiimi kouluttaa asiakasta, jotta asiakas ymmärtäisi vaatimukset ja pystyisi antamaan palautetta ohjelmistosta. (Sillitti & Succi, 2005) Kehitystiimin täytyy myös työskennellä toisten organisaatioiden tiimien kanssa, jotta uuden järjestelmän integrointi olemassa oleviin järjestelmiin onnistuisi sujuvasti. Tiimi, joka järjestelmän kehittää, ei välttämättä ole se, joka ylläpitää järjestelmää. Mikäli ylläpitotiimi on ulkopuolinen, tulisi se myös kytkeä jollakin tavalla kehitysprojektiin. Näin ylläpitotiimi saa tietoa siitä, miten järjestelmä toimii ja mitä asioita tulee huomioida sen ylläpidossa. (Leffingwell, 2011, 122)

5.10 Yhteenveto

Tässä luvussa käsiteltiin ketterän vaatimusmäärittelyn erilaisia käytänteitä ja tekniikoita. Vaatimukset esitetään pääasiassa käyttäjätarinoina, joiden tehtävänä on esittää toiminnot ja ominaisuudet järjestelmälle. Usein käyttäjätarinat ovat suuria, joten niitä täytyy jakaa pienempiin osiin, jotta ne voidaan toteuttaa yhden iteraation aikana ja niiden suunnittelu ja arviointi helpottuvat. Vaatimusten jakamiseen on tarjolla erilaisia tekniikoita, kuten vaatimusten tarkkuustason perusteella jakaminen, käyttäjätarinan sisältämien toimintojen perusteella jakaminen, CRUD -menetelmä ja toiminnallisen ja ei-toiminnallisten vaatimusten toteuttaminen erikseen. Ei-toiminnallisten vaatimusten määrittäminen todettiin ketterässä kehittämisessä haasteelliseksi ja esiteltiin ratkaisuja käytettävyyttä,

luotettavuutta, suorituskykyä ja turvallisuutta koskevien vaatimusten määrittämiseen ja käsittelyyn. Vaatimusten priorisointi on tärkeää ketterässä vaatimusmäärittelyssä, ja sitä tehdään useassa vaiheessa projektia. Luvussa esiteltiin useita priorisointitekniikoita, kuten muutoskustannuksen perusteella priorisointi, käyttäjän saama hyöty, iteraation saama hyöty, riskin minimointi, Kano-malli, ”Osta piirre”, ”Karsi tuotepuuta” ja suhteellinen painotus -tekniikat. Vaatimusten laatua voidaan tarkastella ketterässä kehittämisessä esimerkiksi INVEST-mallin mukaisesti. Lisäksi XP:ssä käytettäville piikeille on omat laatukriteerit. Dokumentointi ketterässä vaatimusmäärittelyssä on vähäistä. Pääasiallisena dokumentaationa ovat käyttäjätarinat, testitapaukset ja koodi. Tiimi päättää, kuinka paljon dokumentaatiota tehdään. Vaatimusmäärittelyn tukena voidaan käyttää tarpeen mukaan mallintamista. Kirjallisuudessa suositellut mallit ovat pääasiassa UML-malleja, kuten sekvenssikaaviot, luokkakaaviot ja aktiviteettikaaviot. Vaatimusten jäljityksen tukena voidaan käyttää erilaisia työkaluja, kuten FLUID: Echo ja Vixtory. Jäljityksen perustana ovat käyttäjätarinat, keskustelut ja koodi, joiden sisältö tulisi olla jäljitettävissä projektin aikana. Ketterässä vaatimusmäärittelyssä on erilaisia rooleja, joilla jokaisella on omat tehtävänsä projektin aikana. Tärkeimmät roolit lankeavat asiakkaalle, projektinjohdolle ja kehittäjille. Asiakkaan tehtävänä on päättää järjestelmää koskevat ehdotukset ja toimia kehittäjien apuna epäselvyyksien esiintyessä. Projektinjohdon tehtävänä on varmistaa toimivat olosuhteet tiimin ja asiakkaan tehokkaalle kommunikoinnille. Kehittäjien rooli on toteuttaa järjestelmä asiakkaan esittämien vaatimusten perusteella, vaatimusten esillesaanti ja neuvottelu, vuorovaikutus asiakkaan kanssa riittävän palautteen saamiseksi sekä asiakkaan koulutus.

6 KETTERÄN VAATIMUSMÄÄRITTELYN HAASTEITA

Edellisissä luvuissa on käsitelty laajasti ja yksityiskohtaisesti ketterän vaatimusmäärittelyn eroja perinteiseen vaatimusmäärittelyyn sekä kirjallisuudessa esitettyjä käytänteitä ja tekniikoita ketterään vaatimusmäärittelyyn. Tämän luvun tarkoituksena on tuoda esille ketterän vaatimusmäärittelyn haasteita. Haasteiden huomioiminen ja niihin reagoiminen ajoissa parantavat onnistumisen mahdollisuuksia kehitysprojekteissa. Rameshin ym. (2007) mukaan kriittisiä haasteita ketterässä vaatimusmäärittelyssä ovat (a) ongelmat projektin kokonaiskustannusten ja keston arvioinnissa, (b) riittämätön tai sopimaton arkkitehtuuri, (c) ei-toiminnallisten ominaisuuksien laiminlyönti, (d) asiakkaan riittämätön mukanaolo, (e) yksiulotteinen vaatimusten priorisointi, (f) vaatimusten riittämätön validointi ja (g) vähäinen dokumentaatio.

Projektin kokonaiskustannusten ja keston arviointi on vaikeampaa ketterässä vaatimusmäärittelyssä kuin perinteisessä lähestymistavassa. Ramesh ym. (2007) toteavat, että vaikeuksia kustannusten ja keston arvioimisessa projektin alkuvaiheessa aiheuttavat muuttuvat vaatimukset, epävakaa ongelma-alue ja dynaaminen suunnitteluvaihe. Lisäksi ketterästä vaatimusmäärittelystä puuttuu formaali vaatimusten analysointivaihe, mistä seuraa, että alustava arvio projektin koosta perustuu käyttäjätarinoiden määrään. Arvio on perusta, jota päivitetään useasti kehitysprojektin aikana. Tiedossa olevat ominaisuudet ja toiminnallisuudet jaetaan pienempiin osiin. Jokainen osa arvioidaan erikseen. Kehitysprosessin aikana lisäyksiä ja hylkäyksiä ominaisuuksiin ja toiminnallisuuksiin tulee kuitenkin paljon. Tomayko (2002) ehdottaa kustannusarvion tekemistä iteraatiokohtaisesti, jolloin jokaiselle iteraatiolle tehdään oma arvio mahdollisista kustannuksista. Myös Rameshin ym. (2007) mukaan arviointi voidaan tehdä iteraatiokohtaisesti, mikä helpottaisi arviointia.

Toisena haasteena Ramesh ym. (2007) mainitsevat *riittämättömän tai sopimattoman arkkitehtuurin*. Aikaisessa kehitysvaiheessa valittu arkkitehtuuri ei välttämättä sovellu tai on riittämätön kehittämisen pohjaksi myöhemmässä vaiheessa, kun uusia vaatimuksia saadaan selville (Tomayko, 2002; Ramesh ym., 2007). Jos tämän tyyppinen ongelma esiintyy, täytyy arkkitehtuuri suunnitella

uudestaan, mikä taas lisää kustannuksia ja pidentää kehitysprojektin kestoa. Ramesh ym. (2007) esittävät ratkaisuna koodin muokkausta eli refaktorointia. Esimerkiksi XP:ssä koodia muokataan aktiivisesti järjestelmän arkkitehtuurin parantamiseksi. Tutkijoiden mukaan refaktoroinnin toteuttaminen riippuu paljon kehittäjien ammattitaidosta ja osaamisesta sekä aikatauluista. Jatkuva refaktorointi ei välttämättä ratkaise Rameshin ym. (2007) mukaan kokonaan vaillinaisen arkkitehtuurin ongelmaa. Jatkuva koodin muokkaus on kallista, ja joissakin tapauksissa ainoa vaihtoehto on kirjoittaa koodi uudestaan.

Kolmas haaste ketterässä vaatimusmäärittelyssä on *ei-toiminnallisten vaatimusten laiminlyönti*. Rameshin ym. (2007) mukaan ei-toiminnalliset vaatimukset on usein huonosti määritelty tai unohdettu iteraatioiden aikana. Vaatimusten esillesaantivaiheessa asiakkailla on taipumus keskittyä liikaa ohjelmiston päätoimintojen kuvailemiseen ja ei-toiminnalliset vaatimukset, kuten turvallisuus, käytettävyys, varmuus, ja skaalautuvuus jäävät vähemmälle huomiolle. Rameshin ym. (2007) mukaan organisaatiot, jotka ovat laiminlyöneet ei-toiminnalliset vaatimukset kehittämisen alussa, kohtaavat ongelmia myöhemmin, kun järjestelmän koko kasvaa. Myös Paetsch ym. (2003) painottavat, että ei-toiminnallisten vaatimusten käsittely on huonosti toteutettu ketterässä vaatimusmäärittelyssä. Sillittin ja Succin (2005) mukaan mitään vakiintunutta tekniikkaa ei-toiminnallisten vaatimusten esillesaamikseksi ei ketterässä vaatimusmäärittelyssä ole, vaan niitä kerätään esillesaantivaiheen yhteydessä. Sillittin ja Succin (2005) mukaan ei-toiminnallisten vaatimusten spesifiointi ei ole ketterässä vaatimusmäärittelyssä niin tärkeässä roolissa kuin esimerkiksi perinteinen vaatimusmäärittelyn yhteydessä. Spesifiointi on korvattu aktiivisella vuorovaikutuksella asiakkaan kanssa. Sidosryhmät eivät aina huomioi ei-toiminnallisia vaatimuksia tarpeeksi, vaan he keskittyvät enemmän toiminnallisiin vaatimuksiin (Rodriquez ym., 2009; Sillitti & Succi, 2005). Rodriquezin ym. (2009) mukaan tämä johtuu asiakkaan tietämättömyydestä järjestelmän teknisistä osa-alueista ja niihin liittyvistä ei-toiminnallisista vaatimuksista. Sillitti ja Succi (2005) toteavat, että tiimin tulisi auttaa asiakasta tunnistamaan ei-toiminnallisia vaatimuksia.

Neljäntenä haasteena Ramesh ym. (2007) mainitsevat *asiakkaan riittämättömän osallistumisen kehitysprosessiin*. Ketterä lähestymistapa vaatii paljon asiakkaan ja kehitystiimin väliseltä viestinnältä. Viestinnän tehokkuus riippuu asiakkaan mahdollisuudesta olla tarvittaessa kehitystiimin saatavilla, asiakkaiden välisestä yhteisymmärryksestä ja heidän luotettavuudestaan. Viimeksi mainittu seikka on tärkeä erityisesti projektin alkuvaiheissa. Ramesh ym. (2007) huomauttavat, että heidän tutkimuksen mukaan asiakas on harvoin läsnä kokopäiväisesti kehitystiimin käytettävissä. Mikäli asiakasryhmiä on useampia, on yhteisymmärryksen löytäminen lyhyissä iteraatioissa vaikeaa. Haastavaksi tilanteen tekee se, että jokainen asiakasryhmä katsoo järjestelmää omasta näkökulmasta. Oman haasteen asettaa myös se, että asiakas ei välttämättä ole täysin ymmärtänyt esittämiään vaatimuksia (Ramesh ym., 2007). Tällainen vaillinainen näkemys järjestelmästä aiheuttaa ongelmia, koska jokaisella asiakkaalla on oma käsityksensä siitä, mitkä ominaisuudet ovat tärkeitä ja mitkä eivät. Ramesh

ym. (2007) esittävät ratkaisuksi riittävän neuvottelun asiakkaan ja kehitystiimin välillä kaikkien asiakasryhmien vaatimusten integroimiseksi osaksi järjestelmää. Luottamus molemmin puolin on tärkeässä roolissa ketterässä kehittämisessä yleisestikin. Vaatimusmäärittelyn yhteydessä luottamus on oleellista erityisesti silloin, kun asiakkaalla ei ole kokemusta ketterästä vaatimusmäärittelystä. Yksityiskohtaisten vaatimusten määrittelyn ja suunnittelun puuttuminen voi aiheuttaa epäluottamusta.

Viides haasteellinen asia ketterässä vaatimusmäärittelyssä Rameshin ym. (2007) mukaan on *yksiulotteinen priorisointi*. Vaatimusten priorisoinnissa käytetään yleisesti ainoana tai hallitsevana kriteerinä liiketoiminta-arvoa. Tästä voi aiheutua Rameshin ym. (2007) mukaan ongelmia arkkitehtuurin skaalautumisen suhteen tai järjestelmään ei voida lisätä muita vaatimuksia (esim. turvallisuus ja tehokkuus), jotka ovat kriittisiä järjestelmän käyttämisen suhteen.

Kuudentena haasteita aiheuttavana asiana on Paetschin ym. (2003) ja Rameshin ym. (2007) esittämä *vaatimusten riittämättön validointi*. Ketterässä vaatimusmäärittelyssä validoinnin merkitys korostuu perinteistä lähestymistapaa enemmän, koska ketterissä kehittämisessä yksityiskohtaista mallintamista ei tehdä riittävästi. Lisäksi tärkeiden vaatimusten puutteellinen spesifiointi voi vaikeuttaa kehittämistä. Rameshin ym. (2007) mukaan mallintaminen voi jäädä tekemättä kiireen vuoksi. Vaatimusten johdonmukaisuuden tarkistaminen suoritetaan harvoin projektin aikana, mikä lisää validoinnin merkitystä. Validointi keskittyy varmistamaan, että vaatimukset vastaavat asiakkaan tämän hetkisiä toiveita. Tämän takia yksityiskohtaista validointia tulisi lisätä prosessin aikana.

Viimeisenä haasteena Rameshin ym. (2007) ja Espinozan ja Garbajoan (2011) mukaan on *vähäinen dokumentaatio*. Ketterässä lähestymistavassa paino on toteuttamisessa eikä yksityiskohtaisten vaatimus- ja suunnitteludokumentaation kirjoittamisessa. Kehitystyötä tehdään vauhdilla kireiden aikataulujen mukaan, eikä aikaa ole välttämättä dokumentaation tekemiseksi. Dokumentoinnin puuttuminen korvataan runsaalla kommunikoinnilla, mutta vaikeuksia voi ilmetä esimerkiksi silloin, kun asiakkaaseen ei saada yhteyttä, vaatimukset muuttuvat nopeasti, uusia jäseniä tulee kehitystiimiin ja sovelluksen monimutkaisuus kasvaa. Näissä tapauksissa dokumentaatiosta olisi paljon hyötyä ja ongelmilta vältyttäisiin (Ramesh ym., 2007). Dokumentaation puute voi aiheuttaa viivästyksiä, koska tiimi joutuu selittämään suullisesti asiat. Erityisesti dokumentaation puute ilmenee ongelmina järjestelmien ylläpidossa (Paetch ym., 2003). Tosin kirjoitettu dokumentaatio voi johtaa väärinymmärryksiin, varsinkin silloin jos dokumentaatio on kirjoitettu formaaliin muotoon eikä sen sisältämiä epäloogisuuksia välttämättä kyseenalaisteta. Muitakin ongelmia kirjoitetussa dokumentaatiossa voi olla (Cohn, 2010, 237): vaatimusten tarkoitus ei käy riittävän selvästi ilmi ja kirjoitettu dokumentaatio vähentää koko tiimin vastuuta, koska osa tiimistä voi olettaa, että toiset tekevät sen mitä dokumentaatiossa on kirjoitettu. (Cohn, 2010, 237)

7 YHTEENVETO

Tämän tutkielman tarkoituksena oli selvittää, millainen on vaatimusmäärittely ketterässä ohjelmistokehityksessä. Tutkimuskysymyksinä olivat:

- Miten ketterä vaatimusmäärittely eroaa perinteisestä vaatimusmäärittelystä?
- Mitä ketterällä kehittämisellä tarkoitetaan?
- Millaisia käytänteitä ja tekniikoita on esitetty kirjallisuudessa käytettäväksi ketterässä vaatimusmäärittelyssä?
- Millaisia haasteita liittyy ketterään vaatimusmäärittelyyn?

Ensimmäiseen tutkimuskysymykseen vastaamiseksi työssä esiteltiin ensin vaatimusmäärittely sellaisena kuin se perinteisessä lähestymistavassa ymmärretään. Tämän mukaan vaatimusmäärittely koostuu neljästä vaiheesta: esillesaannista, analyysista ja neuvottelusta, validoinnista ja dokumentoinnista. Toiseksi työssä kuvattiin lyhyesti, mitä ketterällä lähestymistavalla ja sen mukaisilla Scrum- ja XP-menetelmillä tarkoitetaan. Näiden esitysten pohjalta työssä verrattiin lähestymistapoja ja niiden mukaisia vaatimusmäärittelyjä toisiinsa. Kolmanneksi esitettiin ketterän vaatimusmäärittelyn käytänteitä ja tekniikoita. Viimeisenä käsiteltiin mahdollisia haasteita, joita ketterässä vaatimusmäärittelyssä esiintyy.

Perinteisen ja ketterän lähestymistapojen erot näkyvät projektin johtamisessa, yleisessä toimintatavassa, kehittäjien ja asiakkaan rooleissa, projektin suunnittelussa, arkkitehtuurissa, ohjelmoinnissa, testauksessa ja integroinnissa. Perinteinen lähestymistavan johtaminen on prosessipainotteinen ja yleinen toimintatapa on tiukasti vaiheittain. Kehittäjien rooli on yksilökeskeinen ja osaaminen on tehtäväkohtaista. Asiakkaan rooli painottuu vaatimusmäärittelyyn, ja projektin suunnittelu tehdään ennen projektin aloittamista mahdollisimman kattavasti. Arkkitehtuuri on suunniteltu tiedossa oleville ja ennakoitaville vaatimuksille. Ohjelmoinnissa kaikki ominaisuudet ja toiminnot ovat yhtä tärkeitä ja kaikki sopimukseen kirjoitetut vaatimukset myös toteutetaan. Testaustiimit suorittavat testauksen ja integroinnin projektin lopussa.

Ketterässä lähestymistavassa projektin johtaminen painottuu yhteistyöhön ja ihmisiin. Yleinen toimintatapa on inkrementaalinen sekä iteratiivinen. Kehittäjät ovat moniosaajia, yhteistyökykyisiä ja lähekkäin työskenteleviä. Samoin asiakkaat ovat lähekkäin työskenteleviä ja he osallistuvat koko projektin ajan kehittämistehtäviin. Projektin suunnittelu on jatkuvaa koko projektin ajan. Arkkitehtuurin suunnittelu tehdään tiedossa oleville vaatimuksille. Tärkeimmiksi priorisoidut ominaisuudet ja toiminnot toteutetaan ensiksi, jotta ne varmasti ehditään toteuttaa projektin aikana. Testaus ja integrointi ovat jatkuvaa, joten jokainen valmiiksi saatu osa järjestelmästä testataan ja integroidaan heti järjestelmään.

Kaikki perinteisen vaatimusmäärittelyn vaiheet esiintyvät myös ketterässä vaatimusmäärittelyssä. Merkittävin ero perinteisen ja ketterän vaatimusmäärittelyn välillä on vaiheiden toistuminen jokaisen iteraation aikana. Tästä syystä kaikkia vaatimuksia ei tarvitse tietää etukäteen, vaan riittää että vaatimukset tiedetään seuraavan iteraation osalta. Näin vaatimusten tarkempi analyysi ja suunnittelu tehdään juuri ennen iteraation aloittamista, eikä etukäteen kuten perinteisessä vaatimusmäärittelyssä on tapana tehdä. Validointi perinteisessä vaatimusmäärittelyssä on pitkäkestoinen prosessi, jossa validoidaan vaatimusdokumentaation johdonmukaisuus ja kattavuus. Ketterässä vaatimusmäärittelyssä validointi toteutetaan useissa arviointipalavereissa sidosryhmien kanssa, jossa vaatimukset validoidaan asiakkaan ja kehittäjien välisissä keskusteluissa. Dokumentaatio on perinteisessä vaatimusmäärittelyssä formaalia ja sitä tuotetaan runsaasti. Tieto siirtyy pääasiassa dokumenttien välityksellä, kun taas ketterässä vaatimusmäärittelyssä dokumentaatiota on vähän tai ei ollenkaan, epäformaalit käyttäjätarinat, koodi tai lista ominaisuuksista on yleisin dokumentointitapa. Tieto siirtyy useissa kokouksissa projektin aikana runsaan vuorovaikutuksen kautta.

Työssä esiteltiin runsaasti käytänteitä ja tekniikoita, joiden sanotaan kirjallisuudessa soveltuvan ketterään vaatimusmäärittelyyn. Nämä liittyvät vaatimusten esittämiseen ja jakamiseen, ei-toiminnallisten vaatimusten määrittämiseen, priorisointiin, laatuun, dokumentointiin, mallintamiseen, jäljitettävyyteen ja vaatimusmäärittelyn rooleihin. Ketterässä vaatimusmäärittelyssä vaatimukset esitetään pääosin käyttäjätarinoina, jotka kuvaavat tietyn toiminnon tai ominaisuuden, jonka järjestelmä toteuttaa. Ne ovat usein suuria, joten niitä joudutaan jakamaan pienempiin osiin erilaisin tavoin. Usein ei-toiminnalliset vaatimukset jäävät toiminnallisten vaatimusten varjoon, minkä vuoksi tulee käyttää tilanteeseen sopivia tekniikoita niiden esiin nostamiseksi, esittämiseksi ja käsittelemiseksi. Työssä esiteltiin erityisesti käytettävyyttä, luotettavuutta, suorituskykyä ja turvallisuutta koskevien vaatimuksien määrittämistekniikoita. Ketterässä vaatimusmäärittelyssä vaatimuksia priorisoidaan koko projektin ajan. Käytettävissä on monenlaisia tekniikoita ja kriteereitä (esim. liiketoiminta-arvoon perustuva, ECC -arvoon perustuva, riskin minimointiin perustuva, asiakastyytyväisyyteen perustuva ja MoSCoW-tekniikka). Dokumentointi ketterässä vaatimusmäärittelyssä toteutuu käyttäjätarinoina, valmiina koodina ja testitapauksina. Kattavaa dokumentaatiota ei tehdä projektin alussa, vaan do-

kumentaatiota tehdään koko projektin ajan. Vaatimusmäärittelyn tukena voidaan käyttää mallintamista, joka usein perustuu UML-kieleen. Mallintamisen tarpeellisuutta tulee kuitenkin pohtia tapauskohtaisesti ja käyttää sitä silloin, kun siitä on todellista hyötyä. Vaatimusten jäljitykseen ketterässä vaatimusmäärittelyssä on tässä tutkielmassa esitetty vaihtoehtoina jäljityslinkistöä sekä Vixtory- sekä FLUID: Echo -työkaluja. Useimmat vaatimusten jäljitykseen kehitetyt tekniikat perustuvat vaatimusten spesifiointidokumentteihin, jota ketterässä vaatimusmäärittelyssä ei tehdä

Ketterän vaatimusmäärittelyn roolit ovat merkityksellisiä, koska heistä riippuu projektin onnistuminen. Asiakkaan rooli on erittäin tärkeä, koska hänestä riippuu, millainen on projektin lopputulos. Myös projektinjohdon on oltava ammattitaitoinen riittävien resurssien tarjoamiseksi kehittäjille, luottamuksen luomisessa asiakkaaseen ja hyvän ilmapiirin luomisessa tiimin keskuudessa. Kehittäjien tulee olla erittäin osaavia monella eri tavoin, koska kehittäjien tehtävät ovat laajat.

Lopuksi työssä esiteltiin ketterässä vaatimusmäärittelyssä esiintyviä haasteita. Näitä ovat ongelmat projektin kokonaiskustannusten ja keston arvioinnissa, riittämätön tai sopimaton arkkitehtuuri, ei-toiminnallisten ominaisuuksien laiminlyönti, asiakkaan riittämätön mukanaolo, yksiulotteinen vaatimusten priorisointi, vaatimusten riittämätön verifiointi ja vähäinen dokumentaatio. Se, realisoituvatko haasteisiin liittyvät ongelmat, riippuu siitä missä määrin ne huomioidaan jo projektin alusta alkaen. Projektin johdon vastuulla olisikin kartoittaa edellisten projektien perusteella, missä vaiheessa haasteita ja vaikeuksia esiintyi.

Yksityiskohtainen vaatimusmäärittelykäytänteiden ja -tekniikoiden tarkastelu (vrt. Luku 5) vahvisti perinteisen ja ketterän vaatimusmäärittelyn eroista kirjallisuudessa esitetyt yleiset käsitykset (vrt. Luku 4). Samoin esityksestä kävi ilmi, että ketterän vaatimusmäärittelyn käytänteet (vrt. Luku 5) eroavat perinteisestä vaatimusmäärittelyn käytänteistä (vrt. Luku 2). Kommunikointia ja vuorovaikutusta on paljon enemmän, koska se on tarkoitettu dokumentaation kirjoittamisen korvaajaksi. Vaatimusten esittäminen eroaa siinä, että vaatimukset kerätään korkean tason käyttäjätarinoina, joita tarkennetaan sekä jaetaan iteraatioihin sopiviksi. Perinteisessä vaatimusmäärittelyssä vaatimukset esitetään yksityiskohtaisina ja tarkkoina määritelminä. Dokumentointi on huomattavasti pienemmässä roolissa ketterässä vaatimusmäärittelyssä. Vaatimuksissa esiintyy runsaasti muutoksia, eikä niihin ole järkevää käyttää aikaa jatkuvaan dokumentaation muokkaamiseen. Priorisointia ketterässä vaatimusmäärittelyssä tehdään useissa kohdissa projektin edessä. Validointi sekä neuvottelu ja analyysi ovat samanlaisia molemmissa lähestymistavoissa, joskin vaatimusten validointi suoritetaan keskustelemalla sidosryhmien kanssa arviointipalavereissa, dokumenttien tarkistamisen sijaan. Vaatimusten hallintaa ketterissä menetelmissä ei varsinaisesti ole. Useat erilaiset palaverit ja tapaamiset mahdollistavat vaatimusten toteutusten seuraaminen, mutta lukuisien muutosten seuraaminen ja hallinta olisi ketterässä vaatimusmäärittelyssä melkeinpä mahdotonta. Vaatimusten jäljitys perinteisessä vaatimusmäärittelyssä on helpompi

toteuttaa kuin ketterässä vaatimusmäärittelyssä, koska jäljitys vaatii kattavaa dokumentaatiota vaatimusten kehittymisestä. Lisäksi vaatimuksella tulisi olla yksilöllinen tunnus, jotta sitä voitaisiin seurata. Koska vaatimukset ketterässä vaatimusmäärittelyssä ovat alussa epämääräisiä ja laajoja, olisi vaatimusten ylläpitäminen työlästä.

Tutkimuksen tuloksia voidaan hyödyntää yleiskuvan saamiseksi perinteisen ja ketterän lähestymistavan eroista sekä vaatimusmäärittelyjen eroista. Tutkimuksessa esitettyjä ketterän vaatimusmäärittelyn tekniikoita vaatimusten priorisointiin, jakamiseen, jäljitykseen ja ei-toiminnallisten vaatimusten huomiointiin voidaan harkita käytettäväksi käytännön projekteissa. Luvussa 6 esitetyt haasteet ovat hyödyllisiä projektinjohdon analysoidessa toteutuneiden projektien kokemuksia ja suunnitellessa, miten niihin voitaisiin paremmin varautua tulevissa projekteissa

Vaatimusmäärittely on tutkimusalueena laaja. Tässä tutkimuksessa keskityttiin tarkastelemaan aluetta pääasiassa ketterän lähestymistavan näkökulmasta. Vaikka työ perustuu varsin laajaan lähdeaineistoon, sen ulkopuolelle on kuitenkin jäänyt vielä yleisesti ketterää vaatimusmäärittelyä koskevia sekä yksittäisiä käytänteitä koskevia esityksiä. Rajoituksena on myös se, että vaatimusmäärittelyä käsiteltiin pääasiassa XP:n ja Scrumin yhteydessä. Muissa ketterissä menetelmissä voi olla niille ominaisia käytänteitä ja tekniikoita, jotka voivat olla erilaisia kuin edellä mainituissa menetelmissä.

Jatkotutkimusaiheena voidaan tutkia, miten muissa ketterissä menetelmissä vaatimusmäärittely toteutetaan ja liittyykö joihinkin menetelmiin tiettyjä käytäntöjä. Toisena jatkotutkimusaiheena voidaan tutkia, kuinka organisaatiot hyödyntävät käytännössä tässä työssä esitettyjä tekniikoita ja millaisia vahvuuksia sekä heikkouksia niissä esiintyy.

LÄHTEET

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile Software Development Methods*. VTT Publication 478.

Agile Alliance. (2001). *Agile Manifesto*. Haettu 31.8.2012 osoitteesta http://agilemanifesto.org/

Ambler, S. (2002). *Agile Modelling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York

Anderson, D. (2010). *Kanban: Succesful Evolutionary Change fo Your Technology Business*. USA: Blue Hole Press.

Aoyama, M. (1998). Web-based agile software development. *IEEE Software* 15(6), 56-65.

Astels, D., Miller, G. & Noval, M. (2002). *A practical guide to eXtreme programming.* Upper Saddle River (NJ): Prentice-Hall.

Beck, K. (1999). *Extreme Programming Explained*. Boston: Addison-Wesley.

Beck, K. & Andres, C. (2004). *Extreme programming explained: embrace change*, 2nd edition. Addison-Wesley.

Beyer, H., Holtzblatt, K. & Baker, L. (2004). An Agile Customer-Centered Method: Rapid Contextual Design. *Lecture Notes in Computer Science,* 3134(2004), Extreme Programming and Agile Methods - XP/Agile Universe 2004, 527-554.

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer* 21(5), 61–72 .

Boehm, B. (2002). Get Ready for Agile Methods, with Care. *IEEE Computer*, 35(1), 64–69 .

Boehm, B. & Turner, R. (2003). Observations on Balancing Discipline and Agility. Teoksessa F. Titsworth (toim.), *Proceedings of the Agile Development Conference (ACD-2003), Salt Lake City, Utah, USA, 25-28 June (*s. 32–40 *).* Los Alamitos: IEEE Computer Society.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The unified modeling language user guide*. Reading, Massachusetts: Addison-Wesley Longman, Inc.

Braude, E. & Bernstein, M. (2011). *Software Engineering – Modern Approaches,* 2nd edition. USA: John Wiley & Sons, Inc.

Bruegge, B., Reiss, M. & Schiller, J. (2009). Agile Principles in Academic Education: A Case Study. Teoksesssa S. Lafiti (toim.), *Proceedings of the 6th International Conference on Information Technology: New Generations*. *Las Vegas, Nevada, USA, 27–29 April*. (s. 1684-1686). Los Alamitos: IEEE Computer Society.

Chua, B.B, Bernardo, D.V. & Verner, J. (2010). Understanding the Use of Elicitation Approaches for Effective Requirements Gathering. Teoksessa J. Hall., H. Kandl, L. Lavazza, G. Buchgeher & O. Takaki (toim.), *Proceedings of the 5th International Conference on Software Engineering Advances (ICSEA). Nice, France, 22–27 August.* (s. 325–330). Los Alamitos: IEEE Computer Society.

Cockburn, A. (2002). *Agile software development*. Addison-Wesley.

Cohn, M. (2010). *Succeeding With Agile – Software Development Using Scrum.* Upper Saddle River, NJ: Addison-Wesley.

Cohn, M. (2005). *Agile Estimating and Planning*. Pearsson Education, Inc.

Coley consulting. (2012). MoSCoW Prioritisation. Haettu 31.8.2012 osoittteesta http://www.coleyconsulting.co.uk/moscow.htm

Conboy K. 2009. Agility from first principles: reconstructing the concept of agility in information systems development. *Information Systems Research*, 20(3), 329–354. Informs.

Conboy, K., Coyle, S., Xiaofeng, W. & Pikkarainen, M. (2011). People over Process: Key Challenges in Agile Development. *Software IEEE*, 28(4), 48–57. Los Alamitos: IEEE Computer Society.

Cusumano, M. & Yoffie, D. (1999). Software development on Internet time, *IEEE Computer,* 32(10), 60–69.

Dove, R. (1997). The Meaning of Life and The Meaning of Agility. *Paradigm Shift International.* Haettu 31.8.2012 osoitteesta http://www.parshift.com/Essays/essay001.htm

Duncan, R. (2001) The quality of requirements in Extreme Programming. *CrossTalk, The Journal of Defence Software Engineering*. June, 19-22.

Espinoza, A. & Garganoza, J. (2011). A study to support agile methods more effectively through traceability. *Innovations of Systems and Software Engineering*, 7(1), 53–69.

Far, B. (2007). Software Reliability Engineering for Agile Software Development. Teoksessa *Proceedings of the Canadian Conference on Electrical and Computer Engineering, Vancouver, Canada, April 22-26* (s. 694–697).

Floyd, C. (1984). A systematic look at prototyping. Teoksessa R. Budde, K. Kuhlennkam, L. Matiassen & H. Zullighoven (toim.). *Approaches to Prototyping*. Springer-Verlag, Berlin, (s. 1–18).

Fowler, M. & Kendall, S. (2004). *UML*. Jyväskylä: Docendo.

Ghazarian, A. (2008). Traceability patterns: and approach to requirements traceability in agile software development. Teoksessa S. C. Misra, R. Revetria, L. M. Sztandera, M. Iliescu, A. Zaharim & H. Parsiani (toim.), *Proceedings of the 8th WSEAS International Conference on Applied Computer Science.* (s. 236–241). WSEAS, Stevens Point, Wisconsin, USA.

Harris, R., S. & Cohn, M. (2006). Incorporating Learning and Expected Cost of Change in Prioritizing Features on Agile Projects. Teoksessa P. Abrahamsson, M. Marchesi & G. Succi (toim.) *XP'06 Proceedings of the 7th international conference on Extreme Programming and Agile Processes in Software Engineering* (s. 175–180)

Hatton, S. (2008). Choosing the Right Priotization Method. Teoksessa F. K. Hussain & E. Chang (toim.) *Proceedings of the 19th Australian Software Engineering Conference (ASWEC).Perth, 23-28 March.* (s. 517–526). Los Alamitos: IEEE Computer Society.

Highsmith, J. (2000). *Adaptive software development: a collaborative approach to managing complex systems*. Dorset Hourse Publ.

Ho, C.-W., Johnson, M., Williams, L., & Maximilien, E. (2006) On agile performance requirements specification and testing. Teoksessa J. Chao, M. Cohn, F. Maurer, H. Sharp & J. Shore (toim.) *Proceedings of the Agile 2006. Minneapolis, Minnesota, USA, 23-28 July*. (s. 6–12). Los Alamitos: IEEE Computer Society.

Hohman, L. (2006). *Innovation Games Creating Breakthrough Products Through Collaborative Play*. Addison-Wesley. Upper Saddle River, NJ.

Hull, K., Jackson, K. & Dick, J. (2005). *Requirements Engineering*. Second Edition. Springer Science+Business Media. United States of America.

IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990.

IEEE. (1993). *IEEE Recommended Practice for Software Requirements Specifications*.

Jiang, L., Eberlain, A. & Far, B. H. (2006). A Case Study validation of a knowledge-based approach for the selection of requirements engineering techniques. *Requirements Engineering Journal*, 13(2), 117–146.

Kassab, M., Daneva, M. & Ormandjieva, O. (2007). Scope Management of Non-Functional Requirements. Teoksessa P. Müller, P. Liggesmeyer & E. Maehle (toim.), *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Application*. Lübeck, Germany, 27–31 August. (s. 409–417). Los Alamitos: IEEE Computer Society.

Kinnunen, M. (2007). *Ketterien menetelmien vertailu ja analyysi UML/UP-viitekehyksen avulla.* Pro gradu –tutkielma. Jyväskylän Yliopisto.

Kniberg, H., Skarin, M. (2009). *Kanban and Scrum – making the most of both, Enterprise.* Software Development Series, InfoQ.

Kotonya, G. & Sommerville, I. (2002). *Requirements Engineering Processes and Techniques.* John Wiley & Sons. Great Britain.

Lacey, M. (2012). *Prioritization*. Haettu 20.3.2012. osoitteesta http://msdn.microsoft.com/en-us/library/hh765981(v=vs.110).aspx

Lamsweerde van, A. (2009). *Requirements Engineering From System Goals to UML Models to Software Specifications*. Wiley, NJ, USA.

Lapouchnian, A. (2005). *Goal-oriented requirements engineering: an overview of the current research*, Depth Report, University of Toronto.

Lee, W., J., Cha, S., D. & Kwon, Y., R. (1998). Integration and analysis of use cases using modular Petri nets in requirements engineering. *Transactions on Engineering Management, IEEE*, 24(12), 1115–1130.

Lee, C. & Guadagno, L. (2003). FLUID: Echo – Agile Requirements Authoring and Traceability. Teoksessa *Proceedings of the 2003 Midwest Software Engineering Conference,* Chicago, June. (s. 50–61).

Leffingwell, D. (2007). *Scaling Software Agility – Best Practises for Large Enterprises.* Addison-Wesley.

Leffingwell, D. (2011). *Agile Software Requirements - Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley.

Lyytinen, K. & Rose, G. (2006). Information systems development agility as organizational learning. *European Journal of Information Systems 15(2)*, 183–199.

Martin, J. (1991). *RAD, Rapid Application Development*. MacMillan Publishing Co. New York.

Mohammad, A., Mustafa, M.,A. & Al-Bahar, J.,F. (1991). Project risk assessment using the analytic hierarchy process. *Transactions on Engineering Management, IEEE*, 38(1) 46–52.

Nawrocki, J., Jasinski, M., Walter, B. & Wojciechowski, A. (2002). Extreme Programming Modified: Embrace Requirements Engineering Practices. Teoksessa *Proceedings of the IEEE Joint International Conference on Requirements Engineering,* September 9–13. (s. 303–310).

Nerur, S., Mahapatra, R. & Mangalaraj, G. (2005). Challenges of migrating to agile methologies. *Communication of the ACM,* 48(5), 73–78.

Nuseibeh, B. & Eastbrook, S. (2000). Requirements Engineering: A Roadmap. Teoksessa *Proceedings of the Conference on the Future of Software Engineering,* Limerick, Ireland, June 4–11. (s. 35–46). New York: ACM.

Orr, K. (2004). Agile Requirements: Opportunity or Oxymoron?. *IEEE Software*, 21(3), 71–73.

Overhage, S., Schlauderer, S. & Birkmeier, D. (2011). What Makes IT Personel Adopt Scrum? A Framework of Drivers and Inhibitors to Developer Acceptance. Teoksessa R., H. Sprague, Jr. (toim.), *Proceedings of the 44th Annual Hawaii International Conference on System Sciences.* Kolua, Kauai, Hawaii, 4–7 January. (s. 1–10). Los Alamitos: IEEE Computer Society.

Paetsch, F., Eberlein, A. & Maurer, F. (2003). Requirements Engineering and Agile Software Development. Teoksessa S. Kawada (toim.), *Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprices, Wet Ice. Linz, Austria, 9–11 June.* (s. 308–313). USA: Printing House.

Palmer, S. & Felsing, J. (2002). *A practical guide to feature-driven development*. Upper Saddle River, NJ:Prentice Hall.

Peeters, J. (2005). Agile security requirements engineering. Haettu 15.4.2012 osoitteesta http://secappdev.org/handouts/2008/abuser%20stories.pdf

Pohl, K. (1994). The Tree Dimensions of Requirements: A Framework and its Applications, *Information Systems,* 19(3), 243–258.

Poppendieck, M. & Poppendieck, T. (2003). *Lean software development – An Agile toolkit*. Addison & Wesley.

Pressman, R. (1997). *Software Engineering: A practitioner's approach*. Fourth Edition NewYork: McGraw-Hill Inc.

Pressman, R., S. (2001). *Software Engineering A practioner's approach*. Fifth Edition. (Mc.Graw-Hill series in computer science), New York, NY, 10020

Qumer, A. & Henderson-Sellers, B. (2006). Measuring agility and adoptability of agile methods: a 4-dimensional analytical tool. Teoksessa N. Guimares, P. Isaias A. Goikoetxea (toim.), *Proceedings of the IADIS International Conference on Applied Computing.* (s. 503-507). IADIS Press.

Racheva, Z., Daneva, M., Sikkel, K. & Wieringa, R. (2010). Do We Know Enough about Requirements Prioritization in Agile Projects: Insight From a Case Study. Teoksessa B. Warner (toim.) *Proceedings of the 18th IEEE International Requirements Engineering. Sydney, South Wales, Australia, 27 september – 1 October.* (s. 147–156). Conference Publishing Services (CPS).

Raja, U., A. (2009). Empirical studies of requirements validation techniques. Teoksessa *Proceedings of the 2nd International Conference on Computer, Control and Communication*. Karachi, Pakistan, 17–18 February. (s. 1–9). Los Alamitos: IEEE Computer Society.

Ramesh, B. & Cao, L. (2008). Agile requirements engineering practices: an empirical study. *Software IEEE*, 25(1), 60–67. IEEE Computer Society.

Ramesh, B., Cao, L. & Baskerville, R. (2007). Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5), 449–480. Blackwell Publishing ltd.

Rodriquez, P., Yagile, A., Alarcon, P. & Garbajosa, J. (2009). Some Findings Concerning Requirements in Agile Methologies. Teoksessa F. Bomaries, M. Oivo, P. Jaring & P. Abrahamsson (toim.) *Proceedings of the Product Focused Software Process Improvement – PROFES. Oulu, Finland, June 15–17*. Springer, (s. 171–184).

Royce, W. (1970). Managing the development of large software system: Concepts and techniques. Teoksessa *Proceedings of the 9th International Conference on Software Engineering*. (s. 1–9). IEEE Computer Society.

Rubin, E. & Rubin, H. (2011). Supporting agile software development through active documentation. *Requirements Engineering* 16(2), 117–132.

Rumpe, B. & Schröder, A. (2002). Quantitative Survey on Extreme Programming Projects. Teoksessa M. Marchesi & G. Succi (toim.) *Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Alghero, Italy, May 26–30*. University of Cagliari (s. 95–100).

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

Schwaber, K. & Sutherland, J. (2010) *Scrum guide*. Haettu 5.6.2012 osoitteesta http://www.agilistapm.com/scrum-guide/

Sillitti, A. & Succi, G. (2005). Requirements engineering for agile methods. Teoksessa A. Aurum & C. Wohlin (toim.), *Engineering and Managing Software Requirements*. Springer. Germany.

Silva, S., Martin, T., Maurer, A. & Silveira, M. (2011). User-Centered Design and Agile Methods: A Systematic Review. Teoksessa L. O'Connor (toim.) *Proceedings of the Agile Conference. Salt Lake City, Utah. August 8–12.* Conference Publishing Services (CPS).

Singh, M. (2008). U-Scrum an agile methology for promoting usability. Teoksessa G. Melnik, P. Kruchten & M. Poppendieck (toim.) *Proceedings of the Agile 2008 Conference. Toronto, Canada. August 4-8*. (s. 555–560). Los Alamitos: IEEE Computer Society.

Sohaib, O. & Khan, K. (2010). Integrating Usability Engineering and Agile Software Development: A Literature Review. Teoksessa J. Wang & K. Wang (toim.) *International Conference On Computer Design And Applications (ICCDA). Qinhaungdao, Hebei, China. June 25–27.* (s. 32–38). Los Alamitos: IEEE Computer Society.

Sommerville, I. & Sawyer, P. (1997). *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons.

Stapleton, J. (1997). DSDM, dynamic system developmet: the method in practice. Harlow: Addison-Wesley.

Sureshchandra, K. & Shrinivasavadhani, J. (2008). Adopting Agile in Distributed Development. Teoksessa S. Ceballos (toim.) *Proceedings of the 3rd International Conference On Global Software Engineering.Bangalore, India, August 17–20.* (s. 217–221). Los Alamitos: IEEE Computer Society.

Sutherland, J. & Schwaber, K. (2011). The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework. Draft 29 Jan 2011, Paris. Haettu 10.10.2011 osoittteesta http://jeffsutherland.com/ScrumPapers.pdf

Sy, D. & Miller, L. (2008). Optimizing agile user-centered design. Teoksessa *CHI EA '08: CHI '08 extended abstracts on Human factors in computing systems*. April. (s. 3897-3900). New York: ACM.

Takats, A. & Brewer, N. (2005). Improving communtication between customers and developers. Teoksessa S. Kawada (toim.) *Proceedings of the Agile Conference,* Denver, Colorado, USA 24-29 July. (s. 243–252). Los Alamitos: IEEE Computer Society.

Tomayko, J., E. (2002). Engineering Unstable Requirements Using Agile Methods. Teoksessa *International Conference on Time-Constrained Requirements Engineering*, Essen, Germany, 9 September. Haettu 28.5.2012 osoitteesta http://cf.agilealliance.org/articles/system/article/file/1162/file.pdf

Tuunanen, T. (2005). *Requirements for wide-audience end-users*. Dissertation Thesis, Helsinki University of Economics. Finland.

Uchitel, S., Chatley, R., Kramer, J. & Magee, J. (2004) Fluent-Based Animation: Exploiting the Relation between Goals and Scenarios for Requirements Validation. Teoksessa *Proceedings of the 12th International Requirements Engineering Conference,* Kyoto, Japan 6–10 September. (s. 208–217). Los Alamitos: IEEE Computer Society.

Ungar, J., M. & White, J. (2008) Agile User Centered Design: Enter The Design Studio – A Case Study. Teoksessa *CHI ´08 extended abstracts on Human factors in computing systems* (s. 2167–2178). New York: ACM 2008

Vlaanderen, K., Brinkkemper, S., Jansen, S. & Jaspers, E. (2009). The agile requirements refinery: applying Scrum principles to software production management. Teoksessa *International Workshop on Software Product Management.* (s. 1–10).

Vähäniitty, J. (2012). *Towards agile product and portolio management*. Dissertation Thesis. Aalto University. Finland.

Wang, Q. & Lai, Z. (2001). Requirements Management for the Incremental Development Model. Teoksessa D. C. Young (toim.) *Proceedings of the Second Asia-Pacific Coference on Quality Software,* Hong Kong, 10–11 December. (s. 295–301). Los Alamitos: IEEE Computer Society.

Wong, S., P. & Whitman, L. (1999). Attaining Agility At The Enterprise Level. Teoksessa *Proceedings of the 4th Annual International Conference on Industrial Engineering Theory, Applications and Practice*. San Antonio, Texas, USA, 17-20 November. (s. 1-5).

Weigers, K. (1999). *Software Requirements*. Microsoft Press. Redmond, Washington.

Zhang, Z., Arvela, M., Berki, E., Muhonen, M., Nummenmaa, J. & Poranen, T. (2010). Towards Lightweight Requirements Documentation. *Journal of Software Engineering & Applications*, 3(9), 882–889.