Ketterä kehitys, äärimmäiset vaatimukset?

Reeta Karjalainen

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Toukokuu 2010

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Reeta Karjalainen: Ketterä kehitys, äärimmäiset vaatimukset?
Pro gradu -tutkielma, 81 sivua
Toukokuu 2010

Vaatimusten määrittely muodostaa ohjelmistoprojektin kulmakiven, jonka varaan projektin kaikki myöhempi menestys rakentuu. Vaatimusten muuttuminen kesken projektin on varsin tavallista mutta varmasti ongelmallista. Ketterien ohjelmistokehitysmenetelmien väitetään paitsi sietävän epävarmuutta myös suorastaan syleilevän muutoksia, mutta miten se käytännössä tapahtuu? Ja pitääkö ketterässä ohjelmistokehityksessä vaatimuksistakin tulla jotenkin ketteriä tai äärimmäisiä? Tässä tutkielmassa vertaillaan, miten vaatimusten määrittely perinteisissä ohjelmistokehitysmalleissa eroaa ketterien menetelmien käytännöistä, ja millaisia ilmiöitä ns. ketteryys voi aiheuttaa. Tutkielmassa selvitetään, mitä ketterä vaatimusten käsittely on, ja mihin ketterät menetelmät vaatimusten määrittelyn ja hallinnan näkökulmasta kelpaavat.

Avainsanat ja -sanonnat: ohjelmistotuotanto, ohjelmistoprojekti, ketterät menetelmät, Scrum, XP, vaatimukset, vaatimusten määrittely, vaatimusten hallinta.

Sisällys

1. Johdanto

Ohjelmistoprojektit ovat alttiita epäonnistumiselle, ja aivan liian usein järjestelmät eivät valmistu suunnitellun aikataulun tai budjetin puitteissa. Joskus systeemit eivät ehkä vastaa käyttäjiensä todellisia tarpeita, ja toisaalta, joskus osa vaivalla kehitetyistä ominaisuuksista voi osoittautua käytännössä aivan turhiksi. Jotkut systeemit saattavat olla toivottoman epäluotettavia käyttää, ja havaittujen virheiden korjaaminen tai uusien ominaisuuksien lisääminen voi olla kannattamattoman kallista. Ohjelmistoprojektin epäonnistumisen ei useinkaan voida katsoa johtuvan suoraan siitä, että projektiväki olisi epäpätevää tai käytetyt ohjelmistokehitysmenetelmät- tai työkalut olisivat puutteellisia, vaan ongelmat saattavat juontaa juurensa epäonnistumisiin projektin *vaatimusten määrittelyssä* (requirements engineering).

Ohjelmistoa tai tietojärjestelmää rakennettaessa on keskeistä aluksi kartoittaa, millaisia toimintoja ja ominaisuuksia systeemiltä oikeastaan vaaditaan; tätä työvaihetta ohjelmistokehityksessä kutsutaan vaatimusten määrittelyksi. Leffingwell ja Widrig [2000] määrittelevät, että *vaatimus* (requirement) on kyky, joka ohjelmistolla täytyy olla, jotta se voisi ratkaista käyttäjänsä ongelman tai auttaa tätä saavuttamaan tavoitteensa. Kotonya ja Sommerville [1997] taas määrittelevät, että vaatimus on esitys systeemiltä edellytettävästä palvelusta tai rajoituksesta, ja että *vaatimusmäärittely* (requirements specification) on formaali dokumentti, joka kokoaa systeemin vaatimukset yhtenäiseksi esitykseksi. Vaatimusten määrittely muodostaa ohjelmistoprojektin kulmakiven, jonka varaan projektin kaikki myöhempi menestys rakentuu. Vaatimusten pohjalta lasketaan projektin työmäärä- ja kustannusarviot, laaditaan suunnitelmat ja resursoidaan ja aikataulutetaan hanke. Lopullisen ohjelmiston onnistuneisuutta arvioidaan vertaamalla sitä annettua vaatimusmäärittelyä vasten.

Globalisaation myötä perinteiset ohjelmistoalan kehitystehtävät ovat yhä enenevässä määrin siirtymässä halvemman kustannustason maihin. Ohjelmistoprojekteissa asiakasrajapinnan täytyy kuitenkin olla siellä, missä asiakkaankin, joten suomalaisiakin ohjelmistoala ammattilaisia tarvitaan edelleen. Tehtävä työ on kuitenkin entistä useammin ohjelmoinnin ja testaamisen sijaan ns. *etulinjan* (front-end) haasteita: tuotekonseptien suunnittelua, projektisuunnitelmien laatimista, soveltuvuus-, käyttökelpoisuus- ja käytettävyysanalyysejä, arkkitehtuuristen päätöksien ja teknologisten valintojen tekemistä, sopimusneuvotteluja jne. Ohjelmistosuunnittelijoiden olisi siis kehitettävä osaamistaan mm. vaatimusten käsittelyn osalta. Vaatimusten määrittelyyn ja hallinnointiin soveltuvia tekniikoita ja menetelmiä on tutkittu paljon tietojenkäsittelytieteen piirissä, ja niitä kohtaan riittää kiinnostusta, koska ne tarjoavat mahdollisuuden ennustaa ja hallita ohjelmistoprojektin kulkua. Ongelmana on kuitenkin se, että projektit kohtaavat usein odottamattomia teknisiä haasteita ja joutuvat siksi poikkeamaan tehdyistä

suunnitelmista. Myös vaatimuksilla on taipumus muuttua projektin aikana, jolloin ennusteet tehtävästä työstä menevät pieleen.

Ketterät ohjelmistokehitysmenetelmät ovat viime vuosina saavuttaneet yhä lisääntyvässä määrin suosiota. Niiden keskeisenä ajatuksena on, että koska muutokset ovat väistämättömiä ja koska kaaoksen kontrollointi on mahdotonta, ohjelmistoprojekteissa tulisi siirtyä etukäteissuunnitteluun perustuvasta toiminnasta kohti muutoksiin aktiivisesti reagoivaa ja mukautuvaa toimintamallia. Esimerkiksi vaatimuksia ei pitäisi analysoida tai määritellä kovin tarkasti ennen työhön ryhtymistä, koska ne kuitenkin vielä muuttuvat, vaan pitäisi luottaa siihen, että projektin edetessä ymmärtämys lisääntyy ja tilanne selkiytyy niin tarvittavien ominaisuuksien kuin teknisten yksityiskohtienkin suhteen. Ketterien ohjelmistomenetelmien väitetään paitsi sietävän epävarmuutta, suorastaan syleilevän muutoksia, mutta kysymys herää, miten se käytännössä tapahtuu? Pitääkö ketterässä ohjelmistokehityksessä vaatimuksistakin tulla jotenkin äärimmäisiä? Ja miten kaaokseen valtaan alistuminen voisi tuottaa paremman tuloksen kuin edes osittain oikeaan osunut projektin suunnittelu?

Ketteriä kehitysmenetelmiä voisi pitää asiakkaan näkökulmasta vaarallisena ja hankalana, koska – vaatimusmäärittelyn puuttuessa – toimittaja ei lupaa toimittaa mitään tiettyä toiminnallisuutta, vaan projektitiimi vain sitoutuu kehittämään pyydettyjä toiminnallisuuksia tietyn budjetin puitteissa. Asiakkaan pitää siis sitoutua ohjelmistoprojektiin jo ennen kuin hänellä on varmuutta siitä, pystyykö ohjelmiston toimittaja toimittamaan mitään hänen tarvitsemastaan toiminnallisuudesta. On pakko ihmetellä, miten projektista sopiminen tai lopputuotteen oikeellisuuden validointi voi ketterissä projekteissa toimia asiakasta tyydyttävällä tavalla. Käykö asiakkaan niin kuin kissan, joka vanhassa kansansadussa pyysi hiirtä räätälöimään itselleen takin hienosta kankaasta? Sadussa hiiri ei onnistunut tekemään kissalle takkia, mutta lupasi toimittaa seuraavalla viikolla housut, ja kun hiiri ei saanut housuja aikaiseksi, se lupasi ommella liivit, ja sitten kintaat, myssyn ja lopulta rusetin. Lopulta koko kaunis kangas oli silputtu pilalle, mutta kissa ei saanut edes rusettia – ja söi hiiren.

Perinteisessä ohjelmistokehityksessä asiakas antaa vaatimuksensa valitsemalleen ohjelmiston toimittajalle, joka sitten ryhtyy toteuttamaan niitä. Sen sijaan ketterissä menetelmissä asiakkaan edustaja ja kehitystiimi päättävät keskenään projektin aikana, mitä yritetään saada aikaiseksi, mikä on päätöksenteon dynamiikan kannalta hyvin kiinnostavaa. Janis [1982] identifioi tutkimuksessaan ns. *ryhmäajattelun* (groupthink) ilmiön, joka tarkoittaa sitä, että yksittäisen ryhmän jäsenen on hyvin vaikea kääntyä ryhmässä jo saavutettua konsensusta vastaan. Ryhmäajattelun vallitessa vaihtoehtoisten ratkaisujen etsintä ja arviointi jää pintapuoleiseksi, ja ryhmä on puolueellinen arvioidessaan suosimaansa vaihtoehtoon liittyviä riskejä ja hyötyjä. Tämä herättää kysymyksen siitä, miten asiakas voi pitää puolensa vaatimuksia analysoitaessa ja määriteltäessä.

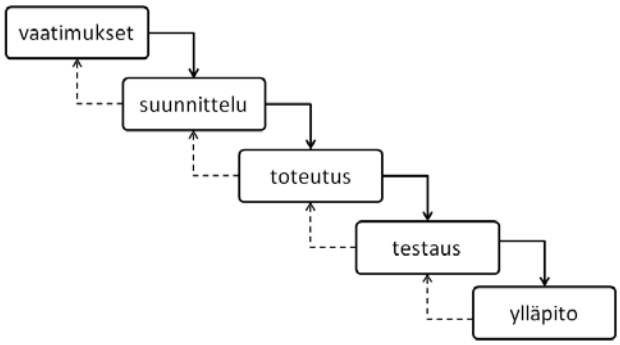
Miten ketterissä projekteissa varmistutaan siitä, että asiakas tulee kuulluksi ja ymmärretyksi? Ja miten käy asiakastyytyväisyyden?

Tässä tutkielmassa vertaillaan, miten vaatimusten määrittely perinteisissä ohjelmistokehitysmalleissa eroaa ketterien menetelmien käytännöistä, ja millaisia ilmiöitä ns. ketteryys vaatimusten määrittelyn ja hallinnan näkökulmasta aiheuttaa. Tarkoituksena on vaatimusten määrittelyä ja ketteriä menetelmiä koskevan kirjallisuuden pohjalta konstruoida kuva siitä, millaista on ketterä vaatimusten käsittely ja mihin se kelpaa. Tästä lähtökohdasta käsin tutkielmassa pyritään edelleen selventämään, miten käyttökelpoisia ketterät menetelmät itsessään ovat ja millaisia reunaehtoja ja rajoituksia niihin liittyy.

Tutkielman luvussa 2 tarkastellaan vaatimusten määrittelyn roolia ja merkitystä ohjelmistokehityksen kannalta. Luvussa 3 perehdytään siihen, mistä vaiheista vaatimusten määrittelyn prosessi koostuu ja millaisia tavoitteita ja tehtäviä vaatimusten käsittelyyn perinteisesti liitetään. Luku 4 perehdyttää lukijan ketterien menetelmien periaatteisiin, ja esittelee kaksi suosituinta ketterää ohjelmistokehitysmallia: Extreme Programmingin ja Scrumin. Luvussa 5 peilataan ketterien menetelmien vaatimuksien käsittelyyn liittyviä käytäntöjä perinteisen vaatimusten määrittelyn prosessia vasten ja etsitään niistä vastaavuuksia, kun taas luku 6 esittelee ketterän vaatimusten määrittelyn erikoispiirteitä ja ongelmia. Luvussa 7 arvioidaan ketteriin menetelmiin sisältyviä mahdollisuuksia ja toisaalta niitä rajoittavia reunaehtoja. Luku 8 tutkiskelee ketterien menetelmien ja vaatimusten suhdetta yleensä, ja luvussa 9 pohditaan, mitä ketterien menetelmien käyttöönotto oikeastaan tarkoittaa ja edellyttää.

2. Vaatimusten määrittelystä yleisesti

Perinteisissä ohjelmistokehitysmalleissa vaatimusten määrittely kuuluu projektin ensimmäisiin vaiheisiin. Royce [1970] esitteli ensimmäisenä lineaarisesti etenevän, vaiheistetun ohjelmistokehitysprosessin, joka on myöhemmin tullut tunnetuksi nk. ”vesiputousmallina”. Kuten kuvasta 1 nähdään, vesiputousmallissa vaatimusten analysointia seuraa ohjelmiston suunnitteluvaihe, ja se jälkeen ohjelmisto toteutetaan, integroidaan tarkoitettuun ympäristöönsä ja testataan. Viimeisenä vaiheena kehitysprosessissa on ohjelmiston ylläpito. Yhdestä ohjelmistokehitysprosessin vaiheesta voi joskus olla tarpeen palata yksi tai useampiakin askeleita taaksepäin; esim. testauksessa löytynyt virhe edellyttää korjausta ainakin itse toteutettuun ohjelmistoon tai jopa ohjelmiston suunniteltuihin käyttötapoihin.



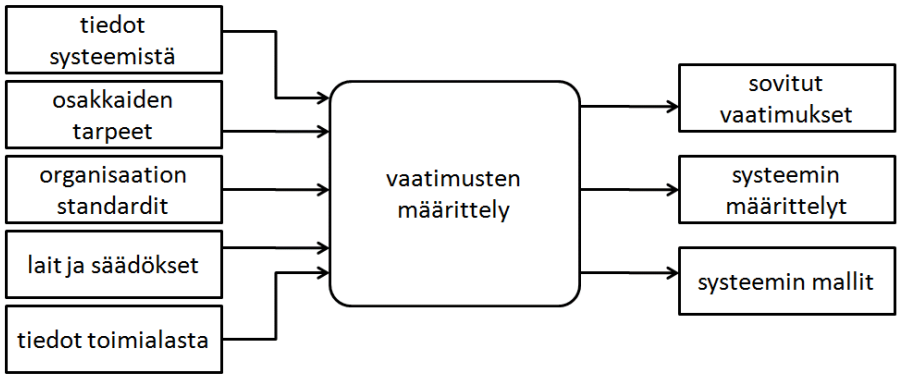
Kuva 1. Roycen [1970] ehdotus ohjelmistokehitysmalliksi.

Vesiputousmallia pidetään nykyään vanhentuneena ja liioitellun yksinkertaisena, ja sitä harvoin enää seurataan sellaisenaan. Käytännössä ko. kehitysmalli toteutetaan usein iteratiivisesti, ja itse Roycekin [1970] suositteli aikoinaan esittelemänsä mallin toistuvaa suorittamista. On mahdollista jakaa tarvittava ohjelmisto pienempiin osajulkaisuihin, ja seurata vesiputousmallia kunkin yksittäinen osajulkaisun toteuttamiseksi. Iteratiivisuus voidaan toteuttaa siten, että jokaisen yksittäisen toiminnallisuuden kohdalla käydään läpi kaikki mallin esittämät vaiheet.

Ennen vaatimusten keräämisen aloittamista on tärkeää tunnistaa projektiin vaikuttavat *osakkaat* (stakeholders) ja osakasryhmät ja saada heidän edustajansa osalliseksi vaatimusten kartoittamisesta. Hofmann ja Lehner [2001] määrittelevät, että projektin osakkailla tarkoitetaan kaikkia niitä ihmisiä ja ryhmiä, joilla on jokin intressi kehitettävän systeemin suhteen, tai joiden intressejä kehitysprojekti koskee. Tällaisia voivat olla esim. ohjelmiston tulevat käyttäjät, käyttäjien kouluttajat, ohjelmiston teettävä asiakas, ohjelmiston kehittäjät ja testaajat, markkinointiosasto, projektin johto, ohjelmiston asentajat ja ylläpitohenkilökunta. Mitä enemmän projektissa on osakkaita, sitä haastavampaa on kaikkien osakkaiden tarpeiden yhteensovittaminen onnistuneesti. Esimerkiksi ohjelmiston ostaja saattaa haluta rahoilleen mahdollisimman paljon vastinetta,

kuten uusia ominaisuuksia, kun taas ohjelman tulevat käyttäjät saattavat toivoa, että ohjelmisto muuttaisi heidän tottumuksiaan mahdollisimman vähän.

Uudella ohjelmistolla pyritään usein ratkaisemaan jokin sen käyttäjien ongelma tai tyydyttämään jokin tarve. Projektin osakkaiden tarpeet muodostavat pohjan vaatimuksille, mutta ne ovat vain osa kaikkia vaatimusten määrittelyssä tarvittavia tietoja. Kotonya ja Sommerville [1997] tähdentävät, että lisäksi täytyy mahdollisesti kerätä informaatiota olemassa olevasta tai vastaavista systeemeistä ja koko toimialueesta ja ympäristöstä sinänsä. Kuten kuvassa 2 on esitetty, myös lait ja säädökset sekä ostaja- ja tekijäorganisaation standardit ja käytännöt on otettava huomioon systeemin kokonaiskuvaa määriteltäessä.



Kuva 2. Vaatimusten määrittelyn syötteet Kotonyan ja Sommervillen mukaan [1997].

Leffingwell ja Widrig [2000] jakavat vaatimukset kolmeen kategoriaan: toiminnallisiin ja ei-toiminnallisiin vaatimuksiin sekä rajoituksiin, jotka pitää ottaa huomioon systeemiä toteutettaessa. Esimerkki toiminnallisesta vaatimuksesta voisi olla se, että soitto-ohjelman edellytetään toistavan käyttäjän valitsema MP3-kappale tämän painettua ”play”-nappulaa. Ei-toiminnallinen vaatimus voisi olla se, äänen laadun pitää olla riittävän hyvä, esim. se ei saa pätkiä eikä säröillä normaalissa kuormituksessa. Toteutusrajoitus voisi olla esimerkiksi sellainen, että ohjelmiston pitää toimia Linux-pohjaisissa laitteissa.

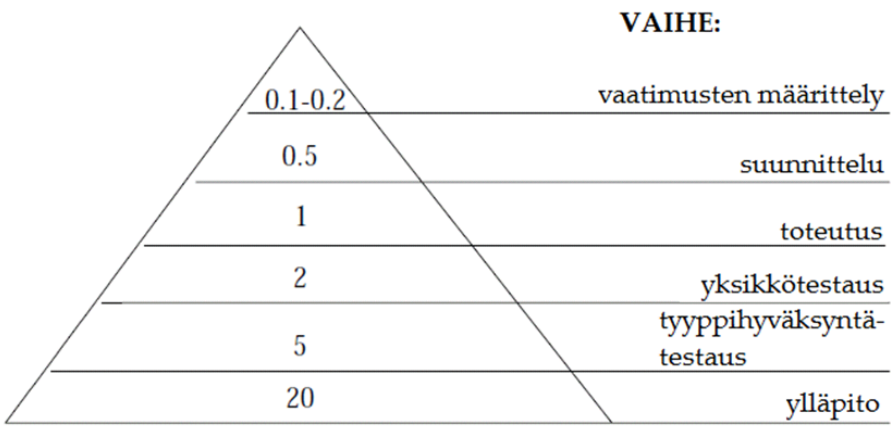
Vaatimusten määrittelyn tuotoksena syntyy vaatimusmäärittelydokumentaatio. Varsinaisen, sovitun vaatimuslistauksen lisäksi se sisältää myös sellaista systeemin toimialueeseen liittyvää tietoa, jota ei voida suoraan ilmaista systeemiltä edellytettävien toimintojen tai rajoitusten kautta, kuten esim. kuvauksia muiden systeemiin liittyvien järjestelmien rajapinnoista tai toiminnoista, tai standardeja, joihin mukautumista systeemiltä edellytetään. Vaatimusmäärittely on sekoitus ongelmakentän, systeemin toiminnan ja ominaisuuksien sekä suunnittelun ja toteutuksen rajoituksen kuvauksia. Vaatimusten määrittelyn erottaminen systeemin suunnittelusta ei siksi ole mielekästä eikä edes mahdollista, vaan prosessit ovat toisistaan riippuvaisia. Leffingwell ja Widrig [2000] pelkistävätkin, että voidakseen vastata kysymykseen ”Mitä ohjelmiston oikeas-

taan pitäisi tehdä?” kehittäjien on osattava 1) analysoida ongelma, 2) ymmärtää käyttäjien tarpeet sekä 3) määritellä tarvittava systeemi.

Vaatimusmäärittelydokumentaatiolla on usein monenlaisia lukijoita: projektin asiakkaat haluavat varmistua siitä, että sovittu järjestelmä vastaa heidän tarpeitaan, ja projektin johto taas perustaa esitettyihin vaatimuksiin omat arvionsa projektin kustannuksista ja resurssitarpeista. Järjestelmän toteuttajat tarvitsevat vaatimusmäärittelyä ymmärtääkseen millainen systeemi heidän tulisi kehittää, testaajat suunnittelevat testitapauksensa vaatimusmäärittelyn pohjalta, ja ylläpitohenkilökuntakin käyttää dokumentaatiota apunaan osatakseen korjata systeemiä ja tehdäkseen siihen muutoksia.

Vaatimusten määrittely on projektin kulmakivi ja samalla myös ensimmäinen mahdollisuus tehdä pahoja virheitä. Leffingwell ja Widrig [2000] arvioivat, että vaatimusten määrittelyssä tapahtuvat virheet ovat myös kaikkein todennäköisimpiä virheitä. Standish Groupin [1995] vuonna 1994 teettämä laaja kyselytutkimus paljasti, että nimenomaisesti vaatimuksiin liittyvät ongelmat aiheuttivat yli kolmanneksen projektien kohtaamista hankaluuksista. Tutkituista projektiorganisaatioista 12% oli törmännyt keskeneräisiin vaatimusmäärittelyihin, toiset 12% muuttuviin vaatimuksiin ja 13% koki, ettei saanut riittävästi tarvittavaa ohjausta ja palautetta asiakkaalta ja käyttäjiltä. Kuvaavaa on, että projekteja haittasivat huomattavasti vähemmän esim. epärealistiset toteutusaikataulut (4%), riittämätön resursointi (6%) tai puutteet henkilökunnan osaamisessa (7 %).

Vaatimusten määrittely vaikuttaa väistämättä kaikkiin ohjelmistoprojektin seuraaviin työvaiheisiin, kuten toteutukseen ja testaukseen, ja myöhempien muutosten tekeminen vaatimusmäärittelyyn on usein hyvin vaivalloista. Leffingwell ja Widrig [2000] esittävät, että virheiden korjaamisen suhteellista kalleutta voidaan kuvata kuvan 3 tapaisella pyramidilla.



Kuva 3. Virheiden suhteellinen kalleus Leffingwellin ja Widrigin [2000] mukaan.

Leffingwell ja Widrig [2000] arvioivat, että vaatimusmäärittelystä juontuvat virheet ovat n. 10 kertaa kalliimpia korjata kuin tavalliset, toteutusvaiheessa tehdyt ohjelmoin-

tivirheet. Jos vaatimuksissa lymyävät virheet saadaan kiinni vasta projektin ylläpitovaiheessa, niiden korjaaminen siinä vaiheessa on jopa 100-200 kertaa kalliimpaa kuin jos virhe olisi havaittu jo alun perin vaatimuksia määriteltäessä.

Vaatimusmäärittelyn virheet ovat hankalia siksi, että niiden todellisen luonteen tunnistaminen voi olla hankalaa. Kun virhe ensimmäisen kerran havaitaan, kehittäjät luonnollisesti alkavat ensin etsiä virhettä viimeisimmistä työvaiheistaan, esim. joko testitapausten määrittelystä, toteutuksesta tai systeemin suunnitelmista. Kun virheen selvittely etenee vaihe vaiheelta työn tuloksia tarkastellen taaksepäin, lopulta jossain vaiheessa huomataan, että ongelmana onkin lähtökohtaisesti virheellisesti käsitetty tai kirjattu vaatimus. Systeemin suunnitelmissa voidaan virheen havaitsemisen jälkeen tehdä tietysti vaikka täysi U-käännös, mutta ajallisesti paluu projektin lähtöpisteeseen ei onnistu. Boehm ja Papaccio [1988] arvioivatkin, että vaatimusmäärittelystä kumpuavien virheiden korjaamiseen voi upota jopa 25-40% koko projektin budjetista.

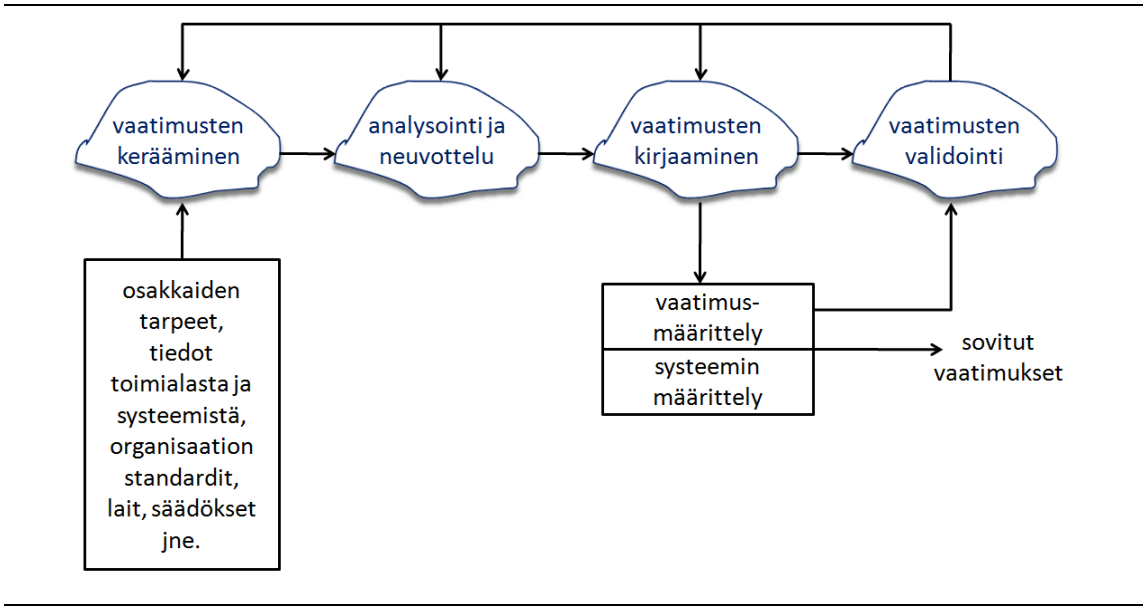
Vaatimusten määrittely on virhealtista, koska se on *vaikeaa*. Bechtold [2005] huomauttaa, että vaikka käyttäjät usein ovat hyvin selvillä ongelmasta, jonka he haluaisivat uudella ohjelmistolla ratkaista, vaatimusten johtaminen ongelmasta voi olla vaikeaa. On haasteellista kuvitella tarvittavaa ratkaisua, koska se muuttaa juuri niitä totuttuja toimintatapoja, joiden yhteydessä ongelma ilmenee. Lisäksi ohjelmiston kehittäjät ja käyttäjät voivat erilaisten taustojensa takia käyttää hyvin erilaista kieltä kuvatessaan ongelmaa, ja he katsovat sitä eri näkökulmista: käyttäjät ajattelivat ohjelmistoa liiketoimintojensa toteuttamisen työkaluna, kun taas kehittäjät pohtivat, miten haluttu ohjelmisto voitaisiin rakentaa. Hofmann ja Lehner [2001] kuvaavat, että jotkin projektin osakkaat, kuten yrityksen markkinointiosasto, saattavat esittää täysin epärealistisia toiveita kehitettävän ohjelmiston suhteen, koska heillä ei ole käytössään samaa teknistä tietämystä kuin ohjelmiston kehittäjillä.

Vaatimusten määrittelyn onnistunut läpivienti edellyttää erilaista osaamista, kuten haastattelu- ja neuvottelutaitoja, kuin mitä ohjelmistoinsinööreiltä on ehkä perinteisesti totuttu odottamaan. Haasteellista on myös se, että tiukan toteutusaikataulun aiheuttaman paineen allakin kehittäjien pitäisi osata hillitä intoaan ”käydä käsiksi” työhön, ja keskittyä ensin kunnolla asiakkaan tarpeisiin. Projektin osakkailta voi toki kysellä heidän odotuksistaan ja toiveistaan, mutta Leffingwell ja Widrig [2000] muistuttavat, että ilmaistut toiveet eivät välttämättä vastaa todellisia tarpeita. Soveltuvat ratkaisut pitää intuitiivisesti johtaa kaikesta saatavilla olevasta informaatiosta, jolloin vaatimusmäärittely voi olla luonteeltaan pikemminkin taidetta kuin tiedettä. Tutkiessaan vaatimusten määrittelyn käytännön toteutusta 15 eri ohjelmistokehitysprojektissa, joissa oli yhteensä 76 osakasta, Hofmann ja Lehner [2001] totesivat, että osakkaiden ottaminen mukaan ohjelmiston vaatimusten määrittelyyn mahdollisimman varhaisessa vaiheessa lisää kaikkien ymmärrystä sekä kehitettävän tuotteen mahdollisuuksista että kehitysprosessista.

Vaatimusmäärittelyt kirjoitetaan usein helposti ymmärrettäviksi toiminnallisuuden kuvauksiksi, ja Firesmith [2003] epäilee, että se saattaa johdattaa kuvittelemaan, että vaatimusten määrittely itsessään olisi jotenkin triviaali tehtävä, josta aloittelijakin voi suoriutua. Hän arvioi, että niin kriittinen tehtävä kuin vaatimusmäärittely onkin, ohjelmistoalan insinööreiltä puuttuu usein koulutusta sekä vaatimusten määrittelyn prosessista että niistä tekniikoista, joka tukevat vaatimusten määrittelyn eri tehtäviä. Lisäksi Leffingwell ja Widrig [2000] korostavat, että kyse ei ole yksilösuorituksesta, vaan tuloksellinen vaatimusten määrittely ja hallinta aina edellyttävät osaavaa ja tehokasta ohjelmistokehitystiimiä.

3. Vaatimusten määrittely prosessina

Leffingwell ja Widrig [2000] määrittelevät, että *vaatimusten hallinnassa* (requirements management) on kyse sekä systemaattisesta tavasta kerätä, organisoida ja kirjata vaatimuksia että prosessista, jolla muodostetaan asiakkaan ja projektitiimin välille yhteisymmärrys systeemin vaatimuksista. Hofmann ja Lehner [2001] tunnistavat vaatimusten määrittelyssä seuraavat vaiheet: *kerääminen* (requirements elicitation), *mallintaminen* (requirements modeling) ja *varmentaminen* (requirements validation and verification). Keräämisvaiheessa vaatimukset ensin kootaan ja niiden vaikutuksia analysoidaan, mallintamisvaiheessa ne pyritään kuvaamaan ja varmentamisessa tarkistamaan vaatimusten kuvauksista, että tarpeet on ymmärretty oikein, sekä verifioimaan, että ohjelmisto on vaatimuksissa kuvatun kaltainen.



Kuva 4. Vaatimusten määrittelyn vaiheet Kotonyan ja Sommervillen [1997] mukaan.

Kotonya ja Sommerville [1997] määrittelevät, että vaatimusten määrittelyssä on kyse systemaattisesta prosessista, jossa pyritään ymmärtämään tulevien käyttäjien tarpeet ja muodostamaan sen pohjalta systeemille eritelty, analyyttinen vaatimusdokumentaatio, eli *vaatimusmäärittely* (requirements specification). He esittävät, että vaatimusten määrittelyn prosessiin kuuluvat ainakin seuraavat vaiheet: vaatimusten *kerääminen* (elicitation), vaatimusten *analysointi ja neuvottelu* (analysis and negotiation), vaatimusten *kirjaaminen* (documentation) ja vaatimusten *validointi* (validation). Nämä vaatimusten määrittelyn vaiheet on esitetty kuvassa 4. Vaatimusmäärittelydokumentaatiota lisäksi *hallinnoidaan* (management) koko ohjelmistoprojektin ajan, ja sen kuvaamien vaatimusten muuttumista ja kehittymistä seurataan.

Käytännön kautta toimiviksi hioutuneet toimintatavat ovat projekteille kriittinen menestystekijä, eivätkä räätälöidyt toimintatavat useinkaan ole sellaisenaan siirrettävissä organisaatiosta toiseen. Kotonya ja Sommerville [1997] toteavat, että koska erilaisil-

le järjestelmille esitetään usein luonteeltaan hyvinkin erilaisia vaatimuksia, ei voi edellyttää, että olisi olemassa yksi ainoa ja oikeaoppinen tapa määritellä niitä. Vaatimusten määrittelyn käytännön toteutus riippuu siis tulevan systeemin toimialueesta ja sitä kehittävän organisaation yleisistä käytännöistä sekä siitä, kuka vaatimukset kerää ja dokumentoi ja keiden luettavaksi ja ymmärrettäväksi itse vaatimusmäärittely on tarkoitettu.

Vuori [2009] muistuttaa lisäksi, että vaatimusmäärittelyn pitää perustua sen ymmärtämiseen, mikä on tärkeää liiketoiminnan, prosessien ja käyttäjien näkökulmasta. Hän kritisoi, että monien organisaatioiden vaatimusten määrittelyn prosesseja leimaavat edelleenkin hyvin perinteiset peruspatologiat, kuten että ratkaistava ongelmakenttä rajataan liian laajaksi ja tarkastelunäkökulma liian kapeaksi. Vaatimusten analyysi ja kehittely voivat jäädä puolitiehen tai käyttäjätutkimuksia tai riskianalyysi ei tehdä lainkaan. Vuori [2009] korostaa, että vain tunnistamalla nykyisten prosessien puutteet organisaatio voi kehittää toimintaansa tuloksellisempaan suuntaan. Onnistunut vaatimusten määrittely tuottaa sivutuotteenaan yhteistä ymmärrystä, oppimista ja sitoutumista.

Jos vaatimusten määrittelyssä on oltu taitamattomia tai piittaamattomia, ei vaatimusmäärittelyn laadun voi olettaa olevan kovin korkea. Vaatimusmäärittelyssä ilmeneviä ongelmia ovat tyypillisesti mm. se, etteivät listatut vaatimukset kuvaa käyttäjien todellisia tarpeita tai että ne ovat puutteellisia tai keskenään epäjohdonmukaisia tai suorastaan ristiriitaisia. Vaatimukset voivat olla myös monitulkintaisia siten, että asiakas, vaatimusmäärittelyn tekijät sekä tulevan systeemin kehittäjät ovat kukin tahollaan saattaneet ymmärtää vaatimukset eri tavalla – tällaisessa tapauksessa on ilmeistä, että asiakas ei todennäköisesti tule olemaan tyytyväinen toimitettuun systeemiin. Jos riskianalyysiä ei ole tehty, vaatimusten laadullisia kriteerejä ei ehkä ole tunnistettu, ja järjestelmän kriittiset toiminnot ja piirteet saattavat jäädä liian vähälle huomiolle.

3.1. Vaatimusten kerääminen

Vaatimusten kerääminen on ensimmäinen vaihe vaatimusten määrittelyssä. Vaatimuksen kerätään konsultoimalla osakkaita sekä tutustumalla systeemistä ja sen toimialueesta saatavilla olevaan tietoon. Kotonya ja Sommerville [1997] listaavat, että kehittäjien täytyy hankkia ymmärrystä 1) toimialueesta, 2) varsinaisesta ratkaistavasta ongelmasta, 3) asiakkaan liiketoimintamallista ja organisaatiosta ja 4) järjestelmän tulevien osakkaiden tarpeista. Jos kehitettävänä on esimerkiksi uusi kokoelman luettelointiohjelma kirjastolle, on tarpeellista ymmärtää laajemminkin, miten kirjastot yleensä toimivat, ja toisaalta, miten juuri tämä kyseinen kirjasto haluaa kokoelmansa luetteloida, miten kokoelmat näkyvät kirjaston hallinnoinnissa ja työn organisoinnissa, ja millaisia toiveita luettelointijärjestelmää käyttävillä ja ylläpitävillä ihmisillä on.

Osakkaiden haastattelu on yksi keskeisimpiä tekniikoita vaatimuksia kerättäessä. Haastattelu voi olla joko suljettu, jolloin se etenee etukäteen päätetyn kysymysrungon mukaan, tai avoin, jolloin keskustelu ei ole sidottu tiettyyn formaattiin, tai jotain näiden

ääripäiden väliltä. Avoimen haastattelun vahvuutena on sisällön rikkaus: haastateltu voi tuoda paremmin esiin mielestään keskeisiä asioita, ja toisaalta haastattelija voi ohjata keskustelua siihen suuntaan, josta nousee esiin mielenkiintoisimpia teemoja. Toisaalta avoimien haastattelujen tuottamat vastaukset eivät ole yhteismitallisia eivätkä siksi helposti vertailtavissa tai suhteutettavissa. Haasteena voi olla myös yhteisen ”kielen” löytyminen, jos haastateltava ja haastattelija ovat taustoiltaan kovin erilaisia ja tottuneet erilaiseen terminologiaan.

Usein ihmisten on vaikea sanallisesti kuvata tehtäviä, joita he hoitavat päivittäisellä rutiinilla. Tällaisissa tapauksissa tulevien käyttäjien tarkkailu on tehokas keino saada tietoa järjestelmän todellisista käyttötavoista. Erilaisten käyttötilanteiden eli skenaarioiden kehitteleminen ja analysointi sekä roolileikit yhdessä osakkaiden kanssa ovat myös tehokas tapa valottaa vaatimuksia.

Leffingwell ja Widrig [2000] kuvaavat, kuinka ”Kyllä, mutta...” -syndrooma on toistuva ongelma ohjelmistoja kehitettäessä. Luonteenomaista on, että asiakas voi sinänsä olla hyvin vaikuttunut projektin tähänastisesta edistyksestä, mutta tarkempi tarkastelu paljastaakin koko joukon pieniä asioita, jotka asiakas sittenkin toivoisi toteutettavan toisella tavalla: ”vau, sehän näyttää hienolta, hmm, mutta mitäs jos...?”. Leffingwell ja Widrig [2000] arvelevat, että koska ohjelmiston suunnittelu on abstraktia toimintaa, asiakkaiden on vaikea hahmottaa suunnitelmia ennen kuin he saavat jotain konkreettista kokeiltavakseen. Prototyyppien kehittäminen ja tutkiminen auttaa asiakkaita ymmärtämään, kuinka järjestelmä voisi toimia, ja kuinka he haluaisivat sen toimivan. Prototyyppejä voi myös käyttää pohjana yhteiselle ideoinnille. Kotonya ja Sommerville [1997] muistuttavat kuitenkin, että prototyyppien kehittäminen ja käyttäjien totuttaminen siihen voi olla kallista ja niin hidasta, että projektin koko aikataulu lykkäytyy eteenpäin. Ainoastaan paperiset prototyypit, kuten piirrokset ja kuvat käyttöliittymästä, ovat suhteellisen nopeita toteuttaa.

Leffingwell ja Widrig [2000] huomauttavat, että eräs haastavimpia piirteitä vaatimuksia kerättäessä on tietää, koska riittävät vaatimukset on saatu kasaan. Ongelmana on, että mitä enemmän vaatimuksia kerää, sitä enemmän niitä tuntuu aina vain löytyvän lisää; keräystyö pitää vain jossain vaiheessa osata lopettaa.

3.2. Vaatimusten analysointi ja neuvottelu

Kun vaatimukset on saatu kerättyä, niitä täytyy tutkia niiden todellisen merkityksen ymmärtämiseksi. Eri osakkaiden kesken on päätettävä, mitkä vaatimuksista voidaan yhteisesti hyväksyä. Jos hankkeessa on monenlaisia osakkaita, on melkein väistämätöntä, että näkökulma- ja tarve-erojen takia osa vaatimuksista on ristiriitaisia. Vaatimukset voivat myös olla liian ylimalkaisia tai puutteellisia tai suorastaan mahdottomia toteuttaa. Lopullinen, kaikkien hyväksymä vaatimuslista syntyy yhteistyössä osakkaiden kanssa neuvottelemalla.

Kotonya ja Sommerville [1997] esittävät, että kaikki vaatimukset arvioidaan ensin ehdottoman tarpeellisuuden näkökulmasta, ja että kaikki ne vaatimukset, jotka eivät liiketoiminnallisten tavoitteiden tai ongelmien ratkaisemisen kannalta ole merkityksellisiä, rajataan pois. Seuraavaksi setvitään ristiriitaiset vaatimukset; selvästikin vain osa sellaisista vaatimuksista voi tulla hyväksytyksi, jos sopivaa kompromissia ei löydy. Muodostetun vaatimuslistan tulee nyt olla täydellinen siinä mielessä, että yhtään olennaista palvelua tai rajoitetta ei saa puuttua siitä. Viimeisimmäksi vaatimuslistaa karsitaan niin, että jäljelle jäävät vain vaatimukset, jotka ovat toteutuskelpoisia sekä teknisesti että projektin budjetin ja käytettävissä olevan ajan puitteissa.

Vaatimusten systemaattinen analysointi on pikkutarkkaa puuhaa, ja siinä käytetään usein apuna erilaisia tarkistuslistoja ja matriiseja. Jokaista vaatimusta tarkastellaan erikseen, esim. onko vaatimus tarpeellinen, edellyttääkö se jotain tiettyä tekniikka tai teknologiaa, onko vaatimus yksiselitteinen tai miten vaatimuksen toteutuminen voidaan testata? Vaatimuksia pitää myös verrata toisiinsa, pareittain, ja arvioida, ovatko vaatimukset ristiriitaisia tai kenties päällekkäisiä. Ristiriitojen ratkaisemiseksi projektin osakkaiden olisi osattava arvioida eri vaatimusten liiketoiminnallisia seuraamuksia ja priorisoida vaatimukset sen mukaan, mitkä toiminnot ovat tärkeimpiä koko ohjelmiston menestymisen kannalta.

Vaatimusten priorisointi on tärkeää, koska tyypillisesti vaatimuksia kertyy niin paljon, etteivät resurssit riitä kaikkien niiden toteuttamiseen – ainakaan ensimmäiseen tuoteversioon. Mikäli vaatimuksia ei ole laitettu tärkeysjärjestykseen, voi käydä niin, että kehittäjät käyttävät kohtuuttomasti aikaa jonkin pienen ja veikeän mutta suhteellisen käyttämättömäksi jäävän toiminnon toteuttamiseen, samalla kun jotain kriittistä jää toteuttamatta. Paetch ja kumppanit [2003] muistuttavat, että myös kehittäjien on asiantuntijoina osallistuttava priorisointiin ja annettava arvionsa yksittäisten vaatimusten toteutushinnasta sekä niihin mahdollisesti liittyvien muiden vaatimusten ja vaikeuksien vaikutuksesta sekä riskien todennäköisyyksistä. Myös vaatimusten väliset, toteutustekniset riippuvuuden on otettava huomioon. Tällaisessa arvioinnissa etusijalle nousevat luonnollisestikin ne vaatimukset, jotka tuottavat maksimaalisen hyödyn suhteessa kustannuksiin ja riskeihin.

3.3. Vaatimusten dokumentointi

Perinteisesti vaatimusten määrittelyssä laitetaan paljon painoarvoa vaatimusmäärittelydokumentaatiolle: paitsi että se on kuvaus toteuttavasta järjestelmästä, se on myös sopimus, johon sekä asiakas että kehittäjät voivat tarpeen tullen nojata. Vaatimukset voidaan kirjata joko luonnollisella kielellä, jollakin kuvauskielellä tai kaavioina (kuten UML) tai kuvina, tai vaikka käyttäen jotain pseudo-ohjelmointikieltä (kuten PDL). Leffingwell ja Widrig [2000] kärjistävät, että vaatimuslista voi tilanteesta riippuen olla vaikkapa vain lyhyt listaus liitutaululla. Hofmann ja Lehner [2001] huomauttavat, että

sanallisten määritelmien ohessa on tavallista kehittää erilaisia systeemimalleja, joissa toisissa pyritään erittelemään suunniteltua toiminnallisuutta ja toisissa taas ohjelmiston sisäisiä asioita, kuten tarvittavia tietorakenteita. Lisäksi myös prototyyppejä, joita osakkaat voivat kokeilla ja testata, voidaan käyttää apuna vaatimuksia kirjattaessa.

Vaatimusmäärittelyssä tärkeintä on se, että dokumentaatio olisi kaikkien osapuolten ymmärrettävissä, sillä myöhemmässä vaiheessa toteutuneen ohjelmiston toimintoja ja ominaisuuksia arvostellaan sitä vasten. Kotonya ja Sommerville [1997] arvioivat, että vaikka luonnollinen kieli on usein helppotajuisin ja suoraviivainen tapa määritellä vaatimuksia, vaatimusten tulkinta voi muodostua ongelmalliseksi etenkin silloin, kun toiminnallisuuteen kuuluu monimutkaista ehdollisuutta; *jos A niin B, paitsi silloin jos C, kuitenkin aina kun D*, jne. Myös terminologian huolimaton käyttö tai epäonnistuneet sanavalinnat voivat tehdä vaatimuksista monitulkintaisia.

Leffingwell ja Widrig [2000] suosittelevat vaatimusten dokumentointia *käyttötapauksina* (use case), jotka kuvaavat, kuka ko. tilanteessa tekee mitä, ja miten systeemi siihen reagoi. Tyypillisesti käyttötapauksessa kuvataan päätoimijan päämäärä, tilanteen laukaisseet tekijät, tilanteen alkuehdot, muut osallistujat ja systeemin reagointi, miten tilanne päättyy onnistuneesti ja milloin sen voi katsoa epäonnistuneen. Kuvassa 5 on esimerkki verkkokaupassa tapahtuvasta käyttötapauksesta, jossa käyttäjä muuttaa ostoskorissa olevien tuotteiden määrää, ja systeemi tarkistaa, että käyttäjän syöttämä lukumäärä on järkevä. Jokainen tilanteen vaihe on kirjattu käyttötapaukseen, kuten myös vaihtoehtoiset tapahtumapolut.

KÄYTTÖTAPAUS NRO 5	Muuta määrää.	
Tavoite	Muuta ostoskorissa olevien tavaroiden lukumäärää.	
Alkuehdot	Käyttäjä on painanut "Muuta kappalemäärää" -nappia.	
Onnistunut lopputulos	Käyttäjä on onnistuneesti muuttanut ostoskorissaan olevien tavaroiden määrää.	
Epäonnistunut lopputulos	Käyttäjä ei onnistunut muuttamaan tavaroiden määrää ostokorissaan.	
Toimijat	Ostaja	
Laukaisija	"Muuta kappalemäärää" -nappia on painettu.	
KÄYTTÖTAPAUKSEN KUVAUS	**askel**	**toiminta**
	1	Systeemi näyttää taulun, jossa on seuraavat sarakkeet: Tuotekoodi, Kuvaus, Määrä ja Hinta. Määrä-sarakkeen pitäisi olla muokattavissa.
	2	Käyttäjä kirjoittaa soluun uuden määrän, ja painaa "Päivitä ostoskori" -nappulaa.
	3	Systeemi tarkistaa, että käyttäjä antoi luvun, joka on suurempi kuin nolla.
	4	Siirry KÄYTTÖTAPAUKSEEN NRO 8: Näytä kori.
VAIHTOEHDOT		**Suorituspolun haara:**
	3	Jos käyttäjä antoi määräksi luvun nolla tai jotain muuta kuin kokonaisluvun, näytä virheviesti "Lukumäärän pitäisi olla nollaa suurempi kokonaisluku."

Kuva 5. Käyttötapaus kuvattuna taulukkoon.

Usein käyttötapaukseen kirjataan myös käyttöliittymään tms. liittyviä huomioita, kuten näppäinten tai toimintojen nimet ja mahdollisesti näytetyt virheilmoitukset. Käyttötapauksissa on vahvuutena se, että ne tuovat esille systeemin eri toimijoiden roolit sekä

sen, miten eri toiminnot ja toimijat liittyvät toisiinsa. Yksityiskohtaiset mutta käytännönläheiset käyttötapaukset ovat helposti projektin eri osakkaiden ymmärrettävissä, ja ne muodostavat eräänlaisen paperiprototyypin systeemistä. Leffingwell ja Widrig [2000] huomauttavat, että koska ainoa tapa verifioida kokonaisen systeemin oikeellisuus on testata sitä vaatimuksia vasten, vaatimusten määrittely selkeiksi toiminnallisuuden kuvauksiksi on etu siinäkin suhteessa.

Leffingwell ja Widrig [2000] korostavat, että yksittäinen vaatimus ei ole yhtä kuin systeemin *toiminto* (feature), vaan systeemin toiminnot tyypillisesti koostuvat useista vaatimuksista. Vaatimusmäärittely tulisi pitää pelkistettynä ja selkeänä, ja sen tulisi keskittyä kuvamaan vain sitä, mitä järjestelmän odotetaan tekevän. Esimerkiksi projektin aikataulu ja resurssiasiat eivät kuulu vaatimusmäärittelyyn eivätkä myöskään vaatimusten testaukseen tai itse toteutukseen liittyvät tekniset asiat. Niiden paikka on projektin muissa, vaatimusmäärittelystä johdetuissa suunnitteludokumenteissa.

Kotonya ja Sommerville [1997] sen sijaan katsovat, että vaatimusta ei voi koskaan pelkistää vain kuvaukseksi siitä, *mitä* systeemin edellytetään tekevän, vaan vaatimuksen pitäisi sisältää myös kuvaus siitä, *miten* systeemin edellytetään suoriutuvan ko. tehtävästä; ts. millaisia toimintatapoja ja lopputuloksia järjestelmältä odotetaan. Vaikka vaatimusten yksinkertaistaminen listaksi olisikin houkuttelevaa, on muistettava, että käytännössä vaatimusmäärittelyä lukevat usein systeemiä toteuttavat insinöörit, joiden on helpompi hahmottaa käytännöllisiä toteutukseen liittyviä ohjeita kuin pelkistettyjä, abstrakteja toiminnallisuuden kuvauksia.

Kotonya ja Sommerville [1997] huomauttavat myös, että käytännön toteutustapojen rajoittaminen jo vaatimusmäärittelyn tasolla esim. standardeihin vetoamalla on järkevää myös siksi, että sitä kautta voidaan parhaiten varmistaa järjestelmän tuleva yhteensopivuus muiden ympäristönsä järjestelmien kanssa. Silloin kun järjestelmän käytännön toteuttajat ovat eri ihmisiä kuin vaatimusmäärittelyn tekijät, vaatimusdokumentaatio on myös hyvä keino siirtää eteenpäin toimialueeseen liittyvää yleistä tietoa ja osaamista.

3.4. Vaatimusten validointi

Kirjatut vaatimukset tulee tarkistaa sen varmistamiseksi, että ne on määritelty johdonmukaisesti ja täsmällisesti, ja että ne muodostavat yhtenäisen, riittävän informatiivisen kokonaisuuden. Kotonya ja Sommerville [1997] korostavat, että siinä missä aiemmassa vaatimusten analysointivaiheessa (ks. kohta 3.2) keskityttiin tunnistamaan systeemin kannalta oikeat vaatimukset, tässä vaiheessa tutkitaan, onko kerätyt vaatimukset ymmärretty oikein. Vaatimusten validointi on tärkeää, sillä kuten Denger ja Olsson [2005] huomauttavat, arviolta 40% kaikista ohjelmistokehitysprojektissa havaituista ongelmista johtuu suoraan tai epäsuorasti huonolaatuisista vaatimuksista. Haasteena vaatimusten validoinnissa on se, että toisin kuin arvioitaessa esim. toimitettua ohjelmistoa, arvioijil-

la ei vaatimusmäärittelyä evaluoidessaan useinkaan ole käytössään mitään konkreettista, jota vasten he voisivat analysoida kirjattuja vaatimuksia.

Vaatimusmäärittelyä voidaan validoida lukemalla sitä ja vertaamalla esitystä muuhun vaatimusten määrittelyn aikana kerättyyn aineistoon. Hofmann ja Lehner [2001] toteavat, että erilaisten dokumenttikatselmointien ja -katselmusten järjestämisen ohella vaatimusten validointiin on tarjolla hämmästyttävän vähän menetelmiä. Apuna dokumentaation validoinnissa voidaan käyttää erilaisia tarkastuslistoja, mutta Denger ja Olsson [2005] huomauttavat, ettei hyviä, ajan tasalla olevia tarkistuslistoja useinkaan ole saatavilla. Tyypillisesti tarkastuslistat ovat kysymyksissään niin yleisellä tasolla, että erillinen käyttöohje niiden tulkitsemiseksi olisi tarpeen. Ennemmin Denger ja Olsson [2005] suosittelevat vaatimusten validointia käyttötapausten pohjalta, sillä skenaariot ohjaavat lukijan suoraan keskittymään olennaisimpiin toimintoihin ja lisäävät yleistä ymmärrystä systeemin suhteen. Hofmann ja Lehner [2001] suosittavat lisäksi, että kaikki katselmuksissa esitetyt kommentit ja erityisesti päätökset ja niiden syyt kirjataan huolellisesti.

Vaatimusten pohjalta tapahtuva systeemin testaaminen on myös eräänlaista vaatimusten validointia, ja myös luotua prototyyppiä voidaan käyttää apuna vaatimusten testaamisessa. Denger ja Olsson [2005] korostavat, että vaikka systeemi- ja hyväksyntätestit usein ajetaankin vasta sitten, kun koko ohjelmisto on saatu valmiiksi, testitapausten kehittäminen on käytännössä vaatimusten validointia. Siksi testitapaukset tulisi suunnitella mahdollisimman aikaisessa vaiheessa projektia – vaikka vaatimusmäärittelyn tarkastuksen ohessa.

Vaatimuksia validoitaessa voidaan huomata, että aiemmista analysoinneista huolimatta vaatimusmäärittelyssä on yhä ristiriitaisuuksia tai epätarkasti kuvattuja vaatimuksia. Systeemin malleja tutkimalla saatetaan myös huomata ongelmia, joita ei ole aikaisemmin tiedostettu. Tyypillistä on myös, että vaatimusten määrittelyistä puuttuvat kokonaan laadulliset kriteerit. Jokaiseen toiminnalliseen vaatimukseen sisältyy aina myös implisiittisiä, laadullisia vaatimuksia – esimerkiksi käynnistettäessä musiikkisoitinta käyttäjällä on jokin käsitys siitä, kuinka kauan soiton alkamista pitäisi korkeintaan joutua odottamaan.

IEEE:n standardi 830 [1998] määrittelee vaatimusspesifikaation ominaisuuksia. Se esittää, että vaatimusten tulisi olla 1) oikeita, 2) yksiselitteisiä, 3) täydellisiä, 4) ristiriidattomia, 5) arvioitu tärkeytensä ja/tai pysyvyytensä suhteen, 6) todennettavissa, 7) muutettavissa ja 8) jäljitettävissä. Vaatimusten oikeellisuus tarkoittaa sitä, ne todella kuvaavat haluttua järjestelmän käyttäytymistä. Yksiselitteisyys tarkoittaa sitä, että vaatimuksen voi tulkita vain yhdellä ainoalla tavalla, ja täydellisyys sitä, että vaatimuksen yhteydessä on kuvattu myös kaikki vaatimukseen liittyvät rajoitukset ja laadulliset kriteerit. Vaatimusten pysyvyys tarkoittaa sitä, miten muuttumaton vaatimuksen uskotaan

olevan, kun taas todennettavuus sitä, että vaatimus on jollakin testillä mahdollista todistaa toteutetuksi.

Vaatimuksen muutettavuus tarkoittaa sitä, että vaatimusmäärittelyn rakenne on sellainen, että jotain vaatimusta muutettaessa on helppo havaita muutoksen seuraukset muuhun vaatimusmäärittelyyn. Asiaan liittyy olennaisesti jäljitettävyys: mikäli vaatimusten väliset riippuvuudet on tuotu selvästi esiin, vaatimusten vaikutusta toisiinsa on myös helppo arvioida ja seurata. Jäljitettävyys tarkoittaa myös sitä, että vaatimusten alkuperä (esim. jonkun tietyn käyttäjän tietty ongelma) on löydettävissä vaatimuksen pohjaksi. Denger ja Olsson [2005] lisäävät listaan, että vaatimusten tulisi olla myös 9) ymmärrettäviä, 10) toteutettavissa ja 11) riittävän yksityiskohtaisesti kuvattuja.

Leffingwell ja Widrig [2000] pelkistävät, että pohjimmiltaan vaatimus on jotain, jota vasten systeemiä tullaan arvioimaan. Olipa kyse sitten toiminnallisista tai ei-toiminnallisista vaatimuksista tai vain toteutusrajoitteista, kaikkien näiden toteutuminen tulee pystyä verifioimaan. Myös luonteeltaan ei-laadullisia vaatimuksia, kuten systeemin turvallisuuteen, suorituskykyyn ja toimintavarmuuteen liittyviä tarpeita, voidaan mitata ja arvioida; se on vain hieman työläämpää kuin toiminnallisten vaatimuksien kohdalla.

3.5. Vaatimusten hallinnointi

Mitä pitempi projektin on ajallisesti, sitä todennäköisempää on, että osa sen vaatimuksista muuttuu. Koska projektin edessä usein myös asiakkaan tai käyttäjien käsitys omista tarpeista ja toiveista selkeytyy, on varsin tavallista, että he esittävät muutostoivomuksia projektin myöhemmissä vaiheissa. On myös mahdollista, että kilpailutilanne markkinoilla edellyttää uusien ominaisuuksien lisäämistä suunniteltuun tuotteeseen, tai vaihtoehtoisesti toimintojen karsimista, jotta tuote saataisiin nopeammin valmiiksi. Muutokset osakkaiden toimintatavoissa tai ympäröivissä IT-systeemeissä voivat heijastua vaatimuksiin, ja myös uudet lait tai asetukset voivat edellyttää lisätoimintoja ohjelmistoon. Toteutuksen valmistuessa voi myös ilmetä, että jotkut alkuperäisistä vaatimuksista ovat teknisesti liian kalliita tai vaikeita toteuttaa.

Reed ja kumppanit [2004] esittävät, että vaatimusten muutokset voidaan jakaa kahteen kategoriaan. Ensinnäkin on olemassa ns. ”todellisia” muutoksia, joiden ilmaantumista edes parhaat toimialan asiantuntijat eivät olisi kyenneet ennakoimaan. Toisaalta osa vaatimuksiin kohdistuvista muutostarpeista juontuu vain puutteellisesti järjestetystä vaatimusten keräämisestä ja analysoinnista, ts. inhimillisistä virheistä. Virheitä voi tapahtua esimerkiksi sen takia, että vaatimusten kerääjiltä puuttuu tarpeellista ymmärrystä toimialasta, eikä asiantuntijaa ole mahdollista saada vaatimusten määrittelyn tueksi. Voi myös olla, että systeemin tulevat käyttäjät eivät ole määrittelyprosessin alkaessa olleet täysin tietoisia siitä, mitä he yrittävät työrutiineillaan pohjimmiltaan saavuttaa, ja mitä ongelmia uuden ohjelmiston pitäisi ratkaista. Reed ja kumppanit [2004] toteavat,

että vaikka kaikkiin muutostarpeisiin pitää reagoida, on toimintatapojen parantamiseksi hyvä selvittää, mistä muutokset vaatimuksissa johtuvat.

Vaatimusten hallinnoinnissa on kyse vaatimusmäärittelyn ja sen kehityksen seurannasta ja hallinnassa. Koska vaatimusmäärittely on eräänlainen sopimus osakkaiden ja systeemin kehittäjien välillä, sen muutoksiin pitää suhtautua kuin sopimusmuutoksiin. Kotonya ja Sommerville [1997] kuvaavat, että vaatimusten hallinta koskettaa niin vaatimuksien sisältöä kuin vaatimusten välisiä riippuvuuksiakin. Lisäksi vaatimusmäärittely on riippuvainen alkuperäisistä vaatimusmäärittelyssä käytetyistä lähdemateriaaleista, ja toisaalta, koko joukko projektin suunnitteludokumentaatiota perustuu vaatimusmäärittelyyn. Jos jotain vaatimusta halutaan muuttaa, pitää voida selvittää, mistä muutostarve johtuu, ja tarkastella sitä kautta muutoksen tarpeellisuutta. Myös vaatimuksen muuttamisen seurannaisvaikutuksia pitää voida kriittisesti arvioida, ja tarpeen tullen toteuttaa tarvittavat muutokset myös muihin vaatimuksiin ja dokumentaatioon.

Vaatimusmäärittelyn versionhallinta mahdollistaa dokumentin muutoshistorian tutkimisen. Muuttunut vaatimusmäärittely voidaan historiatietojensa pohjalta uudelleen validoida selvittämällä muutosaloitteiden tekijät, muutosten syyt ja muutosten seuraukset. Myös projektin hallinta liittyy olennaisesti vaatimusten hallintaan, sillä vaatimusten toteutumista seuraamalla voidaan arvioida projektityön etenemistä.

4. Ketterät menetelmät

Ohjelmiston tilaajan näkökulmasta ohjelmiston kehittäminen kestää aina liian kauan. Bechtold [2005] kuvaa, kuinka ennen ohjelmistoprojektin aloittamista sen asiakkaat ovat ehkä käyttäneet jo useita kuukausia valmisteluihin pohtiessaan projektin kannattavuutta ja tuotteen mahdollisuuksia, järjestellessään tarjouskilpailua, tutkiessaan esitettyjä tarjouksia ja valitessaan ohjelmistolleen toimittajaa. Kun ohjelmistoprojektin toteuttamisesta on tehty päätös, asiakas haluaisi asioiden etenevän vauhdilla: ”Joko lopultakin voidaan aloittaa?”. Ohjelmistokehittäjillä on siis kova paine aloittaa työt heti, vaikka tuotteen vaatimuksetkaan eivät olisi vielä selvillä. Toisaalta tiedetään hyvin se, kuinka paljon hankaluuksia epätarkasti määritellyt, muuttuvat vaatimukset aiheuttavat kehitystyössä ja kuinka ne riskeeraavat koko projektin onnistumisen; liikkuvaan maaliin on tunnetusti vaikea osua. Ratkaisuna tähän ristiaallokkoon tarjotaan ketteriä ohjelmistokehitysmenetelmiä.

Ketterissä ohjelmistokehitysmenetelmissä keskeistä on läheinen yhteistyö asiakkaan kanssa ja ohjelmiston toimittaminen pienissä, toimiviksi todetuissa erissä, pala palalta. Lisäksi Beck ja kumppanit [2001] korostavat ketterien menetelmien manifestissaan, että ohjelmistokehityksessä pitäisi arvostaa yksilöitä ja heidän välistä kanssakäymistään, kuten keskusteluja, neuvotteluja ja yhdessä ideointia, enemmän kuin prosesseja tai työkaluja, ja että muuttuneisiin tarpeisiin reagointi on tärkeämpää kuin minkään etukäteen laaditun suunnitelman seuraaminen. Asiakkaan tarpeiden ja näkemysten kuuleminen on sopimusneuvotteluja tärkeämpää, ja painopisteen tulisi olla toimivan ohjelmiston kehittämisessä dokumentaation tuottamisen sijaan, sillä hyvin kirjoitettu ohjelmakoodi tarjoaa jo itsessään riittävän dokumentaation systeemistä.

Ketterillä menetelmillä pyritään asiakastyytyväisyyden parantamiseen sekä tehokkaampaan ohjelmistotuotantoon. Oletus siitä, että yhteistyö asiakkaan kanssa lisää asiakastyytyväisyyttä, perustuu mm. McKeen ja kumppanien [1994] tekemään tutkimukseen. He totesivat, että sekä asiakkaan mahdollisuudella päästä aktiivisesti vaikuttamaan kehitettävään ohjelmistoon että asiakkaan ja kehittäjien välisen kommunikoinnin määrällä ja laadulla oli suora vaikutus siihen, miten tyytyväinen asiakas oli lopputuotteeseen. Sillitti ja muut [2005] lisäävät, että jatkuva yhteistyö, avoin kommunikointi sekä pala palalta rakentuva ohjelmisto, jonka edistyminen on helppo havainta, myös lisäävät asiakkaan luottamusta ja tyytyväisyyttä ohjelmiston kehittäjiin sekä itse tuotantoprosessiin.

Ohjelmistokehityksen väitetään tulevan nopeammaksi ketterillä menetelmillä mm. sen takia, että kun tehtävä työ on hajotettu pieniin, helposti hallittaviin ja hahmotettaviin inkrementteihin, aikaa ei tuhlata mihinkään sellaisiin tehtäviin, jotka eivät ole juuri sillä hetkellä välttämättömiä hoitaa. Näin vältytään esim. suotta kehittelemästä myöhemmin turhiksi osoittautuvia ominaisuuksia tai valmiuksia tuotteeseen. Koska

jokaisen iteraation lopputuloksena on itsenäinen ja sellaisenaan valmis ja testattu tuote, asiakas voi aina iteraation lopuksi arvioida, onko tuote on jo riittävän hyvä julkaistavaksi, vaikkei jotain alun perin asiakkaan haaveilemaa toiminnallisuutta vielä olisikaan toteutettu. Sillitti ja Succi [2005] kuvaavat, että ketterässä ohjelmistokehityksessä on minimalistinen tavoite tuottaa vain juuri sen verran toiminnallisuutta kuin mitä on ehdottoman välttämätöntä asiakkaan tarpeiden tyydyttämiseksi. Toisin kuin perinteisissä ohjelmistokehitysmalleissa, laajennettavan arkkitehtuurin tai uudelleenkäytettävien komponenttien suunnittelua pidetään ajanhukkana – jopa virheenä – ellei asiakas ole pyytänyt juuri sellaisia ominaisuuksia.

Sillitti ja muut [2005] huomauttavat, että iteratiivisen kehityksen myötä vaatimuksiin liittyvää epävarmuutta voidaan vähentää, kun kaikkea tarvittua tietoa ei ole pakko kerätä ja ymmärtää kerralla oikein. Iteratiivisuus tuo myös joustavuutta ja valinnanvapautta kehitystyöhön, sillä jokainen tehty päätös rajoittaa mahdollisuuksia tehdä jatkossa muita päätöksiä. Mitä pidemmälle sitovien teknisten päätösten tekeminen voidaan ajallisesti lykätä, sitä helpommin ohjelmisto mukautuu erilaisiin tarkoituksiin vielä myöhemmässäkin vaiheessa.

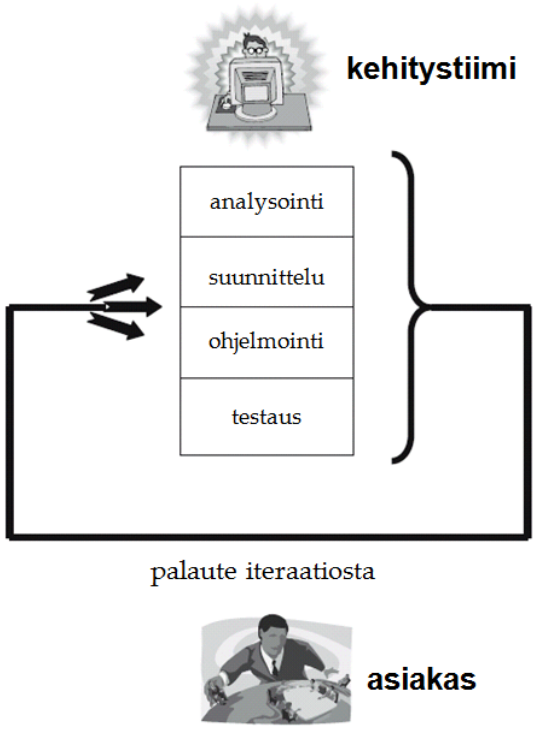
Yksi syy ketterien ohjelmistokehitysmenetelmien esiinmarssiin on alalla tapahtunut huima tuotannollinen kehitys, joka mahdollistaa aiempaa joustavamman työskentelyn. Leffingwell [2009] muistuttaa, että perinteinen vesiputousmalli, jossa jokainen työvaihe suoritetaan itsenäisesti ennen seuraavaan siirtymistä (ks. kuva 1) toimi erinomaisen hyvin aikana, jolloin ohjelmistokehitys oli työlästä ja virhealtista ja siksi myös huomauttavan kallista, ja ohjelmoijilla oli käytössään hyvin rajoitettu valikoima työkaluja ja systeemitason resursseja (kuten muistia). Tuolloin ohjelmistot yksinkertaisesti piti saada kerralla toimimaan oikein ja sopimaan juuri tiettyyn laitteistoon. Iteratiiviset ja inkrementaaliset kehitysmallit taas saivat alkunsa samoihin aikoihin kuin oliokeskeinen ohjelmistotuotanto alkoi lyödä itseään läpi, eli teknologinen edistys heijastui heti ohjelmistokehitysprosesseihin. Moderneilla ohjelmistokehitystyökaluilla työskentely on nopeaa, joten ohjelmistokehittäjillä on nyt lupa kokeilla erilaisia mahdollisia ratkaisumalleja, ja muutamiin erehdyksiinkin on varaa.

Ketteriä menetelmiä on kritisoitu siitä, että ne ovat ainakin joltain osin vain kokoelman vanhoja menetelmiä uudelleen paketoituna. Iteratiivisen Rational Unified Process -ohjelmistokehitysmallin (RUP) menetelmän isänä pidetty Jacobson [2008] huomauttaa happamasti, että useimmat ketterien menetelmien leimalliset ominaisuudet ovat olleet olemassa jo niitä edeltäneissäkin ohjelmistokehitysmenetelmissä. Hän muistuttaa, että jo Boehm esitteli iteratiivisuuden perusidean noin 25 vuotta sitten spiraalimaisessa ohjelmistokehitysmallissaan. Myös RUP, vaikkei olekaan ns. ketterä menetelmä, perustuu iteratiivisen kehityksen periaatteille – pitkälti juuri niistä samoista syistä, kuten nopeampi ja joustavampi kehitystyö, muutoksien ja riskien parempi hallinta, joilla ketteriä menetelmiä nyt kaupitellaan. Ketterien menetelmien tekniset ja sosiaali-

set aspektit, kuten toiminto- tai testauslähtöinen kehittäminen, käyttäjätarinat, päivittäinen kommunikointi ja tiimityö, eivät myöskään ole mitään uutta. Jacobson [2008] katsookin, että ketterien menetelmien suurin vahvuus on prosessien keveydessä ja muunneltavuudessa verrattuna perinteisiin malleihin, sekä sen esiin nostamisessa, että onnistunut ohjelmistokehitys ensisijaisesti edellyttää motivoituneita ja päteviä kehittäjiä.

Abrahamsson ja muut [2003] listaavat, että tunnettuja ketteriä menetelmiä ovat mm. Highsmithin [2000] esittelemä *Adaptive Software Development* (ASD), Amblerin [2002] kehittelemä *Agile Modeling* (AM), *Crystal Methods* -menetelmät Cockburnin [2002] mukaan, Stapletonin [1997] kuvaama *Dynamic System Design Model* (DSDM), Beckin [1999] ideoima *Extreme Programming* (XP), *Feature Driven Development* (FDD) Palmerin ja Felsingin [2002] mukaan sekä Schwaberin ja Beedlen [2002] kehittämä *Scrum*. Käytännössä kaikki nämä menetelmät ovat ainakin joltain osin iteratiivisia, ja iteraatiot ovat suhteellisen lyhyitä (noin kahdesta viikosta kahteen kuukautta), koska tarkoituksena on keskittyä ratkaisemaan vain muutama selkeä, hyvin määritelty ongelma kerrallaan. Jos tarvittavat toiminnallisuudet ovat laajoja tai monimutkaisia, ne pilkotaan pienempiin osiin ja toteutetaan inkrementaalisesti useamman iteraation aikana.

Iteraation aluksi kaikki tunnetut vaatimukset laitetaan aina sen hetkisen ymmärryksen mukaiseen tärkeysjärjestykseen, ja listan alusta valitaan sitten sopiva määrä tehtävää seuraavaan iteraatioon.



Kuva 6. Iteratiivisessa ohjelmistokehityksessä asiakaspalaute ohjaa prosessia.

Kuva 6 esittelee, miten iteraation aikana käydään peräkkäin läpi kaikki ohjelmistokehityksen perusvaiheet, kuten vaatimusten analysointi sekä ohjelman suunnittelu, toteutus ja testaus. Käytännössä iteraation eri vaiheiden sisältö riippuu käytettävästä menetelmästä; joissakin menetelmissä esim. ajetaan iteraation lopussa myös tyyppihyväksyntätestit. Ohjelmisto toteutetaan tiimityönä, usein periaatteella ”kaikki tekevät kaikkea”. Abrahamsson ja muut [2002] huomauttavat, että ketterissä menetelmissä kehittäjät istuvat toimistoissa usein fyysisesti lähellä toisinaan, mieluiten työskennellen samassa tilassa, ja että yhteistyötä ja tiimihenkeä pyritään kaikin keinoin lisäämään. Kehittäjiä kannustetaan myös pitämään ohjelmakoodi helppolukuisena ja siten vähentämään tarvetta kirjoittaa ohjelmistolle dokumentaatio. Tarkoituksena on, että jokaisen iteraation lopuksi tuotettu ohjelmakoodi on valmista julkaistavaksi ja asiakas saa sen arvioitavakseen. Jatkuvan palautteen avulla kehitystoimintaa ohjataan oikeaan suuntaan ja mahdolliset erehdykset voidaan nopeasti havaita ja korjata.

4.1. Extreme Programming

Extreme Programmingin (XP) perustana on joukko hyväksi koeteltuja ohjelmistokehityskäytäntöjä, joista Beck [1999] loi yhtenäisen ohjelmistokehitysprosessin järjestämällä ja yhdistämällä käytännöt. Beck [2007] kertoo, että menetelmän nimi kuvaa sen tarkoitusta mennä äärimmäisyyksiin niissä käytännöissä, joiden katsotaan tukevan menestyksekästä ohjelmistokehitystä. XP perustuu yksinkertaisuuden, kommunikoinnin, palautteen ja rohkeuden periaatteille, ja se on suunnattu pienille ja keskikoksille tiimeille, joissa on maksimissaan 3-20 jäsentä. Abrahamsson ja muut [2003] kuvaavat, että XP:n keskeisiä piirteitä ovat hyvin lyhyet iteraatiot, jatkuva kommunikointi, koordinointi ja uudelleenjärjestely, ohjelmiston jatkuva integrointi ja testaus, yhteisvastuullinen ohjelmakoodin omistus sekä pariohjelmointi. Vaatimukset kuvataan lyhyinä *käyttäjätarinoina* (user story), jotka ovat muotoa ”<tässä roolissa> minä voin <tehdä jotain> saadakseni <jotain>”. Tarinoiden tarkoituksena on kuvata tilannetta käyttäjän näkökulmasta sekä hänen motiivejaan.

Beckin [1999] mukaan XP-prosessin vaiheet ovat *tutkiskelu* (Exploration), *suunnittelu* (Planning), *iterointi* (Iterations to Release), *tuotteistus* (Productionizing), *ylläpito* (Maintenance) ja *lopetus* (Death). Abrahamsson ja muut [2002] kuvaavat kuvassa 7 esitetyn prosessin vaiheita näin:

1. **Tutkiskeluvaiheessa** asiakas kirjoittaa pienille korteille vaatimuksensa ohjelmiston ensimmäisen julkaisuversion toiminnallisuuksien suhteen. Vaatimukset ilmaistaan käyttäjätarinoiden muodossa, ja jokainen kortti edustaa yhtä erillistä toimintoa. Samaan aikaan kehitystiimi tutustuu projektin työkaluihin ja ympäristöön ja alkaa rakentaa karkeaa prototyyppiä tuotteesta voidakseen paremmin testata työkaluja ja tarvittavaa arkkitehtuuria. Tämä vaihe kestää muutaman viikon.

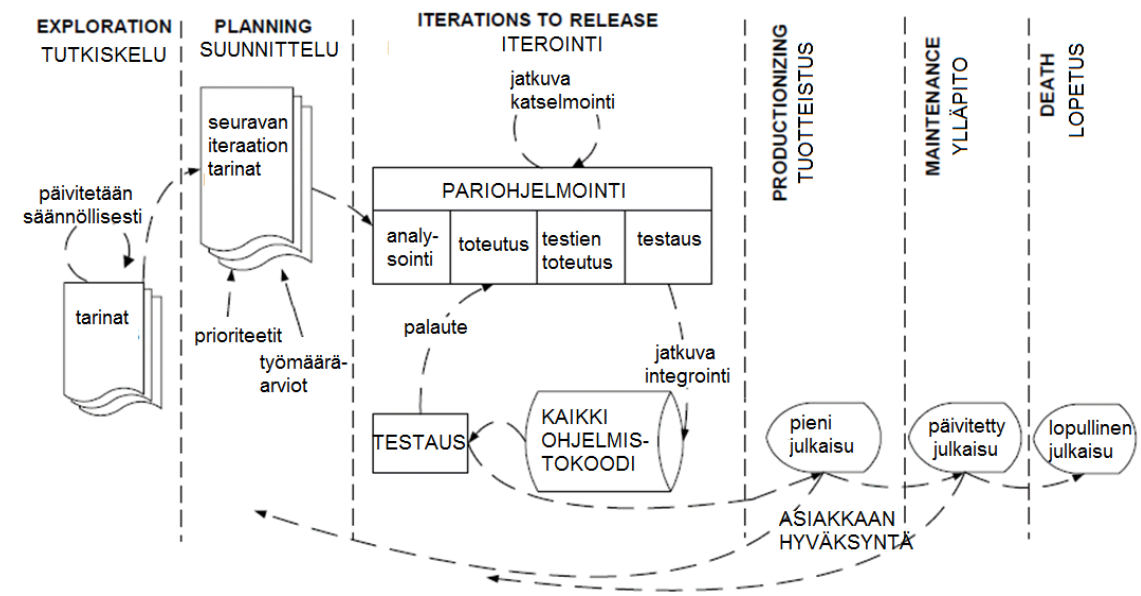
2. **Suunnitteluvaiheessa** kehittäjät antavat työmääräarviot kustakin tarinasta, ja tarinat laitetaan prioriteettijärjestykseen. Ensimmäisen julkaisuversion sisältö päätetään ja aikataulu koko projektille laaditaan; projektin aikataulu ei yleensä ole muutamaa kuukautta pitempi. Tämä vaihe kestää ehkä pari päivää.

3. Sovittu aikataulu jaetaan iteraatioihin, joista jokaisen toteutus kestää yhdestä neljään viikkoa. Ensimmäiseksi valitaan toteutettavaksi jokin sellainen tarina, jonka myötä koko tarvittavan arkkitehtuurin kaikki osat saadaan paikoilleen. **Iterointivaiheessa** käydään läpi useita iteraatiokierroksia. Asiakas valitsee toteutettavat tarinat jokaiseen iteraation. Viimeisen iteraation lopuksi tuote on valmis ensimmäiseen pieneen julkaisuun sekä tuotteistukseen.

4. **Tuotteistusvaiheessa** ohjelmisto alistetaan suorituskykyä mittaavalle ja muulle testaukselle. Jos virheitä löytyy, ne korjataan, ja muutoksia on yhä mahdollista tehdä ohjelmistoon. Iteraatioiden pituus lyhenee nyt yhteen viikkoon.

5. **Ylläpitovaiheessa** ohjelmisto luovutetaan asiakkaalle. Kehitystiimi vastaa sekä ohjelmiston ylläpidosta että sen jatkokehittämisestä.

6. **Lopetusvaiheessa** kaikki tarpeellisiksi katsotut tarinat on toteutettu, ja ohjelmisto on valmis. Tarvittava dokumentaatio kirjoitetaan nyt, kun systeemi ei enää muutu.



Kuva 7. Extreme Programming -prosessi kaaviokuvana Beckin [1999] mukaan.

Abrahamsson ja muut [2002] kuvaavat, että XP:ssä asiakkaan edustajalla on keskeinen rooli tuotekehityksessä, ja hänen pitää olla päivittäin paikalla keskustelemassa projektin ohjelmistokehittäjien kanssa. Asiakkaan edellytetään määrittelevän vaatimukset, eli käyttäjätarinat, sekä suunnittelevan niihin sopivat testitapaukset toteutuksen verifiointia varten. Asiakas myös validoi, milloin kukin tarina on saatu valmiiksi, ja laittaa tarinat tärkeysjärjestykseen.

Kussakin käyttäjätarinassa kuvataan aina yksi asia, jonka käyttäjä haluaa tehdä tuotteella, esim. ”Asiakkaana haluan etsiä kirjan kirjaston valikoimista kirjailijan nimen perusteella”. Wake [2003] opastaa, että hyvä käyttäjätarina on pieni, testattavissa oleva itsenäinen toiminnallisuus, jolla on sellaisenaan arvoa käyttäjälle. Wake kehottaa ajattelemaan koko tuotteen täytekakuksi, jossa erilaiset tekniset yksityiskohdat – palvelimet, tietokannat, ohjelmistot – muodostavat kukin oman kerroksensa, ja jossa on kuorrutteena käyttäjälle näkyvä käyttöliittymä. Nyt jos haluamme tarjota asiakkaalle maistipalan täytekakusta, emme tietenkään leikkaa kakkua vaakasuuntaisiin viipaleisiin, koska silloin asiakas maistaisi vain yhtä kerrosta kakusta, eikä saisi siitä kokonaiskuvaa. Sen sijaan kakku pitää leikata lohkoihin pystysuunnassa. Samaan tapaan itse tuotetta ei myöskään kehitetä kerros kerrokselta, vaan kakkupala kakkupalalta. Käyttäjätarina on siten vain maistiainen koko toiminnallisuudesta, mutta se silti koostuu kaikista niistä teknisistä kerroksista, joita valmiissa tuotteessa tullaan tarvitsemaan.

Asiakas päättää eri julkaisuiden sisällöstä ja aikataulusta, ja iteraatiota suunniteltaessa kehittäjät haastattelevat asiakasta saadakseen paremman käsityksen vaatimuksista. Kehittäjät antavat sitten tarinoista työmääräarvionsa, joiden perusteella asiakas voi aikatauluttaa ja arvioida tarinoita. Sillitti ja Succi [20005] kuvaavat, että XP-projektin edistymistä seurataan tarinataulun avulla: iteraation kuuluvat tarinat, jotka asiakas on kirjoittanut pienille paperilapuille, kiinnitetään isolle taululle tai seinälle, kaikki nähtäville. Tarinataulu on jaettu pystysuunnassa kolmeen sektoriin: aloittamattomien, työn alla olevien ja toteutettujen vaatimusten alueisiin. Kun tarinan toteutus etenee, sen parissa työskentelevä kehittäjä siirtää lappua taululla eteenpäin.

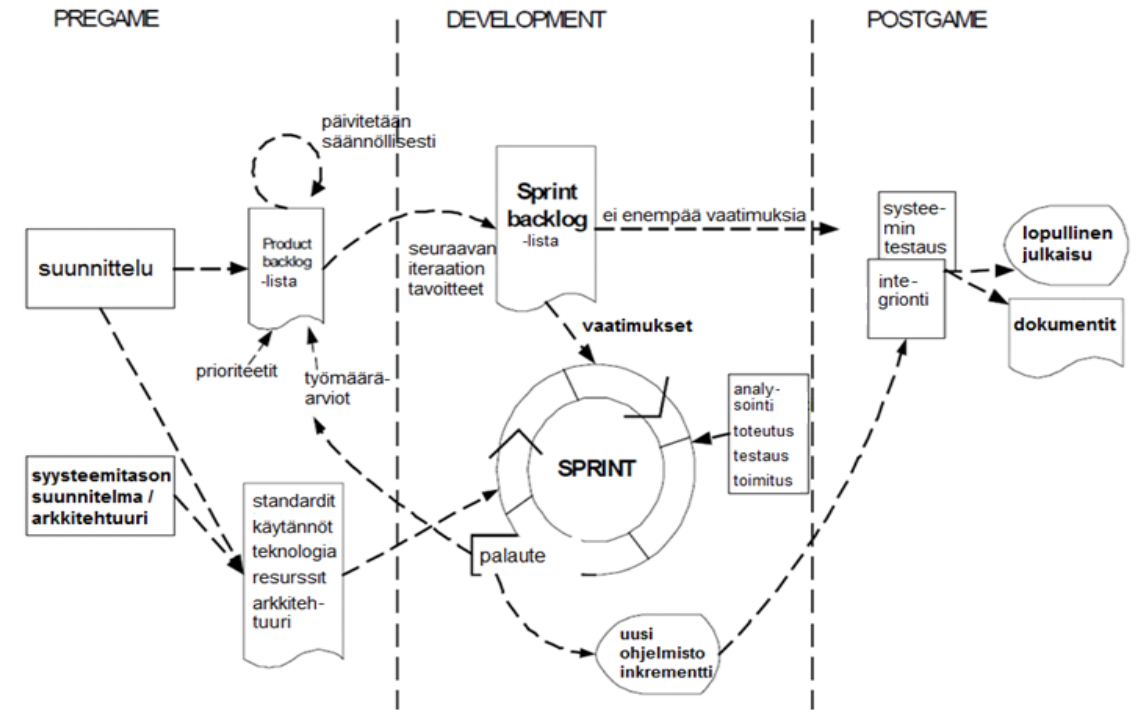
XP:ssä kehitystyö on testauslähtöistä, eli kehittäjien oletetaan toteuttavan testitapauksen ennen ohjelmakoodin kirjoittamista. Ohjelmakoodi toteutetaan pariohjelmointina, ja päivän aikana tuotettu ohjelmakoodi integroidaan ohjelmistoon aina päivän päätteeksi. Ensimmäinen ohjelmistojulkaisu tulee ulos tyypillisesti n. 2-3 kk kuluttua projektin aloituksesta, ja sen jälkeen julkaisuja voidaan tuottaa vaikka joka päivä. Ylitöitä ei sallita.

4.2. Scrum

Schwaber ja Beedle [2002] kuvaavat oivaltaneensa, miten vähän ohjelmistojen toteuttaminen käytännössä muistuttaa rakentamista tai tehdastuotantoa. Ohjelmisto on aina ns. ”uusi” tuote, ja sen kehittäminen on siksi empiiristä työtä, jonka ennustettavuus on heikkoa. Monet tekniset rajoitteet ja mahdollisuudet tulevat ilmeisiksi vasta työn ede-

tessä ja aiheuttavat osaltaan muutoksia vaatimuksiin. Ohjelmiston suunnittelua ja toteuttamista ei siis ole mahdollista erottaa tiukasti toisistaan. Perinteisesti on ajateltu, että huolellisesti suunnitellut, yksityiskohtaisesti määritellyt ja tarkasti noudatetut prosessit, organisaatiomallit ja roolitukset ovat paras tapa varmistaa kehitystyön odotusten mukainen laatu ja eteneminen. Koska ohjelmistoprojektien haasteet kuitenkin ovat aina erilaisia, optimaalista kehitysprosessia ei käytännössä ole mahdollista määritellä. Sen sijaan oikeanlaisella ohjelmistokehitysmallilla prosessia voidaan jossain määrin hallita ja ohjata oikeaan suuntaan.

Scrum on ohjelmistoprojektin organisointi- ja hallintamalli, ja se kuvaa, kuinka tiimin jäsenten tulisi toimia tuottaakseen joustavasti ja tehokkaasti ohjelmistoja jatkuvasti muuttuvassa ympäristössä. Tavoitteena Scrumissa on parantaa jo olemassa olevia käytäntöjä ongelmakohtien tunnistamisen kautta. Scrum ei esittele eikä suosittele mitään tiettyjä toteutustekniikoita (kuten pariohjelmointi). Schwaberin ja Beedlen [2002] mukaan Scrum-ohjelmistokehitysprojektissa on kolme vaihetta, jotka on esitetty kuvassa 8:



Kuva 8. Scrum -prosessi kaaviokuvana Schwaberin ja Beedlen [2002] mukaan.

1. **Pregame**-vaiheessa kehitetään projektisuunnitelma sekä tarvittavat systeemitason ja arkkitehtuuriset suunnitelmat. Projektille varataan työntekijät (ihanteellinen tiimin koko on n. 5-9 henkilöä), työkalut sekä muut resurssit. Kuten kuvasta 8 käy ilmi, projektille luodaan ns. ”Product backlog” -lista, joka sisältää tuotteeseen liittyvät vaatimukset sekä laaditaan ohjelmiston julkaisuaikataulu.

2. **Development**-vaiheessa tuote kehitetään iteraatioiden eli ”sprinttien” kautta. Iteraation alussa pidetään aina suunnittelupalaveri, jossa Product backlog -listalta valitaan alkavassa iteraatiossa toteutettavat vaatimukset ”Sprint backlog” -listalle; työlistaa ei voi muuttaa iteraation aikana. Joka päivä järjestetään noin 15 minuutin mittainen pikapalaveri, jossa tiimin jäsenet kertovat, mitä ovat tehneet sitten viime palaverin ja mitä aikovat seuraavaksi tehdä. Sprintin lopuksi järjestetään katselmus, jossa arvioidaan iteraation onnistuneisuutta sekä esitellään aikaansaatu toiminnallisuus. Yksi iteraatiokierros voi kestää viikosta kuukauteen, ja ensimmäinen julkaisu tehdään 3-8 inkrementin jälkeen.

3. **Postgame**-vaiheessa kaikki tarvittavat vaatimukset on saatu valmiiksi. Systeemi integroidaan, testataan ja dokumentoidaan.

Abrahamsson ja muut [2002] tarkentavat, että Product backlog -listalla oleva vaatimukset voivat olla yhtä hyvin peräisin niin asiakkaalta kuin myynti- tai markkinointiosastolta, asiakastuesta tai projektitiimiläisiltä itseltäänkin. Projektin edetessä työlistalle ilmestyy vaatimusten lisäksi myös muita tärkeitä tehtäviä, kuten virheenkorjauksia tai päivitystehtäviä.

Asiakasta edustaa projektissa ns. Product Owner, joka toimii linkkinä asiakkaan ja projektitiimin välillä ja jonka vastuulla Product backlog -lista on. Projektitiimiläiset antavat työmääräarviot kaikille työlistan tehtäville ja vaatimuksille, ja Product Ownerin tehtävänä on pitää lista koko ajan prioriteettijärjestyksessä, jotta uusien tehtävien valitseminen sieltä olisi mahdollisimman suoraviivaista.

Scrum-tiimin on tarkoitus toimia itseohjautuvasti, ja tiimiläiset saavat itse päättää, miten kunkin iteraation tavoitteisiin pyritään. Scrum-projektissa projektipäällikkö ei ole enää perinteinen työnjohtaja vaan valmentaja, joka poistaa esteitä toteutustiimin tieltä ja neuvottelee yrityksen ylemmän johdon kanssa. Lisäksi Scrum-tiimiin kuuluun ns. Scrum Master, joka on eräänlainen tiiminvetäjä. Hänen tehtäviinsä kuuluu valvoa ja opastaa oikeaoppisen prosessin ylläpidossa ja toimia välittäjänä projektijohdon ja tiimin välissä. Käytännössä projektin vetäjä voi itse toimia Scrum Masterina, mutta projektipäällikkö voi myös keskittyä hallinnollisiin tehtäviin, tai toimia Product Ownerina, jolloin hänellä saattaa olla useampikin projekti valvottavanaan ja huollettavanaan yhtä aikaa. Tällöin on tärkeää, että projektilla on osaava ja riittävät toimintavaltuudet omaava Scrum Master ohjaamassa päivittäistä työskentelyä ja päätöksentekoa.

5. Vaatimusten käsittely ketterissä menetelmissä

Perinteinen vaatimusten määrittely perustuu olettamukselle, että mitä myöhemmin ohjelmistovirheet havaitaan, sitä kalliimpia ne ovat korjata, jolloin on järkevää uhrata aikaa ja vaivaa vaatimusten perinpohjaiselle selvittelylle ennen toteuttamiseen ryhtymistä. Implisiittisesti tällöin myös oletetaan, että on mahdollista määritellä pysyvä ja yksityiskohtainen kokoelma vaatimuksia heti projektityön alkuvaiheessa. Ketterät menetelmät taas väittävät, että kumpikaan em. olettamuksista ei pidä paikkaansa, ainakaan useimmiten, ja että tulevaisuuden vaatimusten ennustaminen ja niiden yksityiskohtien arvailu on epäonnistumiseen tuomittu yritys.

Ketterät menetelmät olettavat, että vaatimusten muuttuminen on väistämätöntä etenkin siksi, että sekä asiakkaan että systeemin kehittäjien ymmärrys omista tarpeistaan sekä kehitettävästä systeemistä paranee ja syvenee sitä mukaa kun projekti etenee. Beck [1999] analysoi, että koko ”vaatimus”-sana on terminä lähtökohtaisesti väärä, sillä se antaa ymmärtää, että kyse olisi jostakin tarkkaan määritellystä, pakollisesta tarpeesta, vaikka kyse on vain lähtökohdasta, josta käsin tuotetta aletaan muovata. Myös ”en tiedä mitä haluan, mutta tiedän kun näen sen” -ilmiö näkyy asiakkaan käyttäytymisessä; vasta tutkimalla konkreettista systeemin toteutusta asiakas alkaa hahmottaa, millaisen ohjelmiston hän oikeastaan haluaisi ja tarvitsisi. Oletusta tukee tutkimus, jossa Sillitti ja muut [2005] vertailivat kahdeksan ”ketterän” ja kahdeksan ”perinteisen” yrityksen kokemuksia. Tutkimuksen mukaan 75% ketterissä yrityksissä haastatelluista toimijoista oli sitä mieltä, että vaatimukset muuttuvat “usein” tai “aina”, ja myös perinteisistä yrityksissä 63 % koki samoin.

Koska siis vaatimukset joka tapauksessa muuttuvat moneen otteeseen, on järkevää pyrkiä sopeutumaan jatkuvaan muutokseen. Sillitti ja muut [2005] korostavat, että ketterissä menetelmissä jokaista muutosta pidetään positiivisena asiana, koska se mahdollistaa systeemin parantamisen asiakkaan toivomaan suuntaan. Ketterän filosofian mukaan vaatimuksen muutos on aina paitsi todiste oppimisesta, myös mahtava tilaisuus parantaa tuotetta asiakkaan toiveiden mukaiseksi. Kuvaavaa on, että ym. tutkimuksessa 88% perinteisissä yrityksissä toimijoista koki vaatimusten muuttumisen vakavaksi ongelmaksi, kun taas ketteristä yrityksissä vain 13% haastatelluista oli sitä mieltä.

Ketterien menetelmien pohjalla on näkemys siitä, että kehittyneiden ohjelmistotuotantomenetelmien ja -työkalujen takia muutosten toteuttaminen ohjelmistoon ei ole projektin myöhemmässä vaiheessakaan olennaisesti sen kalliimpaa kuin projektin alkupuolella, etenkin kun muutokset ovat kuitenkin välttämätön kuluerä. Pelkkä iteratiivinen kehitysprosessi ei kuitenkaan sinänsä riitä ketterän kehityksen mittariksi, vaan koko ohjelmistokehitysmallin pitää tukea jatkuvaa asiakkaan kanssa neuvottelua, yhteistyötä, itseohjautuvaa ja vastuullista työskentelyä sekä nopeaa muutosten havainnointia ja muutoksiin reagointia. Cao ja Ramesh [2008] luonnehtivat, että ketteryys on tarpeen

myös vaatimusten käsittelyssä, jotta voidaan vastata haasteisiin, joita tuovat nopeat muutokset kilpailutilanteessa, käytettävissä olevan teknologian kehitys sekä tarve olla entistä nopeammin markkinoilla. Yksityiskohtaisen vaatimusmäärittelyn sijaan projektille laaditaan vaatimusten pohjalta vain kevyt ”tiekartta”, joka näyttää etenemissuunnan, ja joka tarkentuu vaatimusten tasolle työn edetessä.

Ketterät menetelmät korostavat, että tuotteen kaikki ei-välttämättömät toiminnallisuudet ovat tuhlausta, joka voi koitua koko projektin kohtaloksi. Jos vaatimuslistalla on asioita, joita ei välttämättä tarvita, niiden toteuttamiseen ja ylläpitoon tärvätään aivan suotta aikaa ja vaivaa, samalla kun ohjelmakoodin monimutkaisuus lisääntyy. Turhat toiminnallisuudet tuhlaavat myös systeemin käytettävissä olevaa muistia, virtaa ja prosessointitehoa, ja ne tekevät tuotteesta vain monimutkaisemman käyttää. Tarpeettomien toimintojen toteuttaminen voi myös viivästyttää koko ohjelmiston valmistumisaikataulua, ja näin vähentävät ohjelmiston tuottavuutta asiakkaan silmissä. Sillitti ja Succi [2005] toteavat, että pyrkimys eliminoida kaikki turha johtaa siihen, että vaatimusten määrittelyn onnistuminen ja erityisesti vaatimusten priorisointi ovat kaikkein tärkeimmät menestystekijät ketterässä projektissa.

Perinteinen vaatimusten määrittely, jota pidetään dokumenttikeskeisenä ja raskaana prosessina, ei sellaisenaan tietenkään istu ketteriin kehitysmenetelmiin. Paetch ja muut [2003] kuitenkin huomauttavat, että monet vaatimusten määrittelyn prosessin vaiheet, kuten vaatimusten kerääminen, analysointi ja validointi, ovat silti käytännössä jollain tavoin edustettuina ketterissä projekteissa. Abrahamsson ja muut [2003] analysoivat, että useimmat ketterät menetelmät opastavat myös vaatimusten käsittelyssä. Tosin siinä missä Scrum tarjoaa eniten konkreettisia ohjeita vaatimusten käsittelyyn, XP, FDD ja DSDM kuvaavat enemmänkin abstrakteja vaatimusten määrittelyn ja hallinnan periaatteita. Vaatimuksien käsittelyyn käytetään ketterissä menetelmissä kuitenkin paljon vähemmän aikaa kuin perinteisemmissä kehitysmalleissa. Koska varsinainen toteutustyö tehdään iteratiivisesti, pienissä erissä, voidaan vaatimuksienkin kohdalla keskittyä tarkastelemaan ainoastaan niitä toimintoja, jotka ovat olennaisia juuri ko. toteutusvaiheen kannalta.

Ketteryyttä korostavat ohjelmistontuotantoprojektit pyrkivät noudattamaan ketteriä periaatteita myös käsitellessään vaatimuksia. Cao ja Ramesh [2008] totesivat haastateltuaan 16 eri ketterää organisaatiota ja niiden vaatimusten määrittelyn käytäntöjä, että kaikki organisaatiot suosivat kasvotusten tapahtuvaa kommunikointia tärkeimpänä tapana välittää tietoa ja neuvottelivat aina asiakkaan kanssa jokaisen vaatimuksen yksityiskohdista. Melkein kaikki hyödynsivät iteratiivisuutta vaatimusten kehittelyssä, ja vaatimusten priorisointia tarkasteltiin joka syklissä. Useimmat projektit hyödynsivät myös edelleen kehiteltäviä prototyyppejä selvittäessään asiakkaan tarpeita, ja useimmat myös kehittivät testejä, joilla vaatimukset voitiin validoida iteraation tuotoksia katselmoitaessa.

5.1. Vaatimusten ketterä kerääminen

Kaikki ohjelmistoprojektit alkavat jonkinlaisella vaatimusten keräämisellä. Koska ketterissä projekteissa toteutustyö on tyypillisesti iteratiivista, myös vaatimusten kerääminen sekä niistä neuvottelu usein toistetaan jokaisen iteraation yhteydessä. Projektin edetessä, osakkailta saadun palautteen avulla, vaatimukset alkavat ilmestyä näkyviin. Paetch ja muut [2003] kuvaavat, kuinka XP:n “Planning Game” -neuvottelu yhdistää vaatimusten keräämisen, analysoinnin ja neuvottelun. Neuvotteluissa asiakas kirjoittaa haluamiensa toimintojen kuvaukset pienille tarinakorteille, joista kehittäjät ja asiakas sitten keskustelevat, yksi kerrallaan. Saatuaan riittävän käsityksen ehdotetusta tarinasta kehittäjät arvioivat, miten työläs kuvattu toiminto olisi toteuttaa, ja näiden tietojen pohjalta asiakas ja kehittäjän yhdessä päättävät, mitkä tarinoista otetaan toteutettavaksi seuraavaan ohjelmiston julkaisuversioon.

Sillitti ja Succi [2005] kuvaavat, että tavat kerätä vaatimuksia eivät eroa paljoakaan perinteisissä ja ketterissä projekteissa. Esimerkiksi ketterissä menetelmissä vaatimuksia kerätään koko projektitiimin voimin, verrattuna perinteisten menetelmien suosimiin pienempiin vaatimusmäärittelyryhmiin. Scrumissa asiakas ja kehittäjät luovat yhdessä tuotteelle vaatimuslistan, jota he analysoivat yhdessä. FDD:ssä kehitetään ensin korkean abstraktiotason malli koko systeemistä, ja malliin kehityt toiminnot vastaavat systeemin vaatimuksia. DSDM:ssä toiminnot eli vaatimukset johdetaan projektin edetessä projektin suunnitteludokumentaatiosta, kuten liiketoimintasuunnitelmasta. Paetch ja muut [2003] huomauttavat, että siinä missä DSDM:ssä ja Scrumissa vaatimusten keräys on ensisijaisesti ryhmätyötä, muut ketterät metodit korostavat asiakkaan vastuuta prosessista.

Beckin ja muiden [2001] julistamien ketterien periaatteiden mukaan kasvokkain tapahtuva keskustelu on tehokkain ja suositeltavin tapa paljastaa ja välittää tietoa. Useimmat ketterät metodit suosittelevatkin pääasialliseksi tavaksi kerätä vaatimuksia esim. asiakkaan haastattelemisen sekä ajatuksilla ja ideoilla leikkimiseksi asiakkaan kanssa. Korkala ja kumppanit [2006] havaitsivat, että XP-projektissa tehtyjen virheiden lukumäärä nousi jopa viisinkertaiseksi silloin, kun asiakkaan edustaja ei ollut paikalla. Fyysisesti tavoitettavissa olevan asiakkaan vaikutus oli siis näin merkittävä, vaikka asiakas käytti työpaikalla vain noin viidenneksen työajastaan varsinaiseen projektityöskentelyyn ja kysymyksiin vastaamiseen ja hoiti muun ajan toisenlaisia työtehtäviä.

Ketteriä projekteja haastatellessaan Cao ja Ramesh [2008] saivat selville, että useimmat organisaatiot pitivät kasvokkain tapahtuvan kommunikoinnin etuna sitä, että asiakkailla oli näin parempi mahdollisuus ohjata projektia muuttuneiden vaatimuksiensa suuntaan. Lisäksi koettiin, että epämuodollinen kommunikointi vähensi tarvetta aikaa vievälle dokumentaatiolle tai hitaille, virallisille hyväksyntäprosesseille. Haasteena kommunikoinnissa oli se, että avoin keskustelu edellyttää luottamusta ja yhteishenkeä, jota etenkään projektin alkuvaiheessa ei vielä ole syntynyt.

5.2. Vaatimusten ketterä analysointi ja neuvottelu

Ketterissä projekteissa vaatimuksia analysoidaan ja niistä neuvotellaan seuraavan iteraation aluksi. Koska useimmat ketterät menetelmät kehottavat säännöllisiin, pieniin osajulkaisuihin, voidaan viimeisintä ohjelmistoversiota käyttää myös eräänlaisena prototyyppiä uusia toimintoja kehitettäessä tai toiminnallisuutta parannettaessa. Verrattuna perinteiseen vaatimusten määrittelyyn ketterä vaatimuksista neuvotteleminen tuo aikataululliset ja sisällölliset kompromissimahdollisuudet paremmin esille. Kun toteutettavat ominaisuudet on jaettu pieniin, modulaarisiin osiin, asiakkaan on helpompi arvioida, mitä ominaisuuksia hän haluaa aikataulun puitteissa kehittää, ja toisaalta tutkia, miten julkaisuaikataulun venyttäminen tai lyhentäminen on mahdollista vaatimuslistaa muokkaamalla. Caon ja Rameshin [2008] tutkimuksen mukaan myös projektin kehittäjät kokivat, että vaatimukset olivat helpompia ymmärtää, kun he saattoivat käydä niitä läpi pienissä erissä, keskustellen asiakkaan kanssa joka iteraatiosta erikseen.

Paetch ja muut [2003] kuvaavat, kuinka Scrum-menetelmän ”Sprint Planning” -kokouksessa, aina uuden sprintin alussa, asiakas valitsee Product backlogilta toteutettavan sisällön alkavaan sprinttiin, ns. Sprint backlog -listalle. Kuten Scrumissa, myös useimmissa muissa ketterissä menetelmissä asiakkaan tehtävänä on järjestää vaatimukset tärkeysjärjestykseen, eikä hänen tarvitse osata ottaa huomioon esim. vaatimusten välisiä teknisiä riippuvuuksia tai hankaluuksia, jos hän ei katso sitä tarpeelliseksi. Tästä on se hyöty, että vaatimukset ovat näin aina kaikkein tarkoituksenmukaisimmassa järjestyksessä, ilman raskaita analysointiprosesseja. Cao ja Ramesh [2008] huomauttavat, että silloin kun projektissa on useita osakkaita, jotka priorisoivat eri omaisuuksia, konsensuksen löytäminen lyhyen iteraatiokierroksen aikana voi olla haastavaa, mutta kasvokkain tapahtuva neuvottelu kaikkien osakkaiden kesken voi auttaa asiaa.

Toisin kuin perinteisessä priorisoinnissa, jossa tyydytään organisoimaan vaatimukset kertaalleen tärkeysjärjestykseen, ketterissä projekteissa vaatimusten painoarvo harkitaan jokaista iteraatiokertaa varten erikseen. Cao ja Ramesh [2008] huomauttavat, että priorisointi ei kohdistu vain osakkaiden esittämiin vaatimuksiin, vaan vaatimusten rinnalle nostetaan tarkasteltaviksi myös muita työtehtäviä, kuten virheenkorjauksia, ohjelmakoodin siivoustehtäviä tai muuta sellaista. Myös kehittäjät voivat esittää vaatimuksia silloin, kun jokin tehtävä – esim. arkkitehtuurinen muutos johonkin rajapintaan – on välttämätön muiden vaatimusten suorittamisen kannalta.

Vaatimusten priorisoinnin kannalta on olennaista, että kehitystiimi osaa antaa kullekin työtehtävälle uskottavan työmääräarvion, jotta tiimi voi päätellä, mitkä asiakkaan tilaamista vaatimuksista on mahdollista toteuttaa iteraation aikana. Ketterien menetelmien vahvuutena pidetäänkin sitä, että vaikka työmäärän arvioiminen on vaikeaa, sen jatkuva harjoittelu iteraatiosta toiseen parantaa kehittäjien arviointikykyä ja ennusteiden luotettavuutta. Ketterien menetelmien hyödyntäminen voi tätä kautta parantaa koko organisaation kykyä asettaa ja saavuttaa tavoitteita.

5.3. Vaatimusten dokumentointi?

Beck ja muut [2001] julistavat, että ketterässä ohjelmistokehityksessä toimivaa ohjelmistoa pitäisi aina arvostaa enemmän kuin kattavaa dokumentaatiota. Tämä on käytännössä johtanut siihen, että ketterissä projekteissa harrastetaan dokumentointia hyvin laiskasti ja pintapuolisesti. Systeemeistä ei yleensä vaivauduta tekemään kattavia kuvauksia, ja tuotetun dokumentaation päätarkoituksena on tukea keskeneräistä toteutustyötä, jossa tärkeimpinä tiedonlähteinä ovat kuitenkin keskustelut asiakkaan kanssa.

Dokumentaation luomisessa haasteena on yrittää kuvitella kaikkia niitä kysymyksiä, joita systeemin myöhemmillä käyttäjillä tai ylläpitäjillä voi herätä, ja yrittää vastata näihin kysymyksiin mahdollisimman selkeästi ja kattavasti. Laajan dokumentaation ylläpito on myös työlästä. Vaatimusmäärittelyn kirjoittaminen edellyttää syvällistä ymmärrystä vaatimuksista ja systeemistä, ja Paetch ja muut [2003] huomauttavatkin, että varsinaisen vaatimusmäärittelyn kirjoittamatta jättäminen voi olla hyvin kustannustehokas tapa toimia. Kun vaatimuksia ei tarvitse yrittääkään ymmärtää ennen niiden toteuttamista, kaikki tarvittava systeemitieto onkin jo saattanut karttua siihen mennessä kuin itsestään.

Ketterissä menetelmissä vaatimukset kirjataan määrittelydokumentaation sijaan esim. tarinakorteille tai -listoille. Paetch ja muut [2003] analysoivat, että Scrumissa ylläpidettävä Product backlog -lista, jonne on kirjattu kuvaukset kaikista tiedetyistä toiminnoista, tarvittavista parannuksista, havaituista virheistä ja tarpeellisiksi tai hyödyllisiksi katsotuista työtehtävistä, on oikeastaan projektin vaatimusmäärittely – joskin epätäydellinen ja muutosaltis sellainen. Sprint backlog -lista taas on käytännössä yksittäisen iteraation vaatimuslista. FDD:ssä vaatimuksista pidetään yksinkertaista listaa, kun taas DSDM:ssä vaatimuksista kerätään projektin edetessä kehittyvä dokumentaatio. XP:ssä vaatimusmäärittely koostuu yksittäisistä, kertakäyttöisistä tarinalapuista sekä tarinataulusta, jolle lappuja asetellaan. Sharp ja muut [2009] toteavat, että vaikka tarinalappujen tekstuaalinen sisältö on vähäinen, niiden notaatio ja visuaalinen konteksti sisältävät runsaasti projektin työläisille aukeavaa informaatiota. Tähdellä tarinalapun nurkassa voidaan esimerkiksi kuvata tehtävän suhteellista tärkeyttä, lapun käsialasta voidaan päätellä kirjoittaja, jolta voi kysyä lisätietoja, ja lappujen sommittelu taululla taas kertoo tarinoiden suhteista toisiinsa.

Dokumentaation vähäisyys voi olla ketterän projektin vahvuus, sillä Sillitti ja muut [2005] ovat todenneet, että perinteisiä ohjelmistokehitysmenetelmiä käyttävistä projekteista jopa puolella oli ongelmia vaatimusmäärittelyn kanssa: vaatimukset oli joko kirjattu väärin tai kirjattu kuvaus oli ymmärretty väärin. Mitä keskeisempi rooli dokumentaatiolla on informaation välittämisen kannalta projektissa, sitä todennäköisemmin kuvauksiin pääsee pujahtamaan virheitä, ja sitä suurempi haittavaikutus kirjallisilla virheillä on. Ketterät menetelmät, joissa kehittäjät toimivat elävänä dokumentaa-

tiona ja joissa kommunikointi tapahtuu ”livenä”, eivät ehkä ole niin alttiita väärinymmärryksille.

5.4. Vaatimusten ketterä validointi

Cao ja Ramesh [2008] kuvaavat, kuinka käytännössä kaikissa ketterissä projekteissa harjoitetaan vaatimusten validointia. Jokaisen iteraatiosyklin lopuksi projektin osakkaat järjestävät palaverin, jossa käydään läpi iteraation saavutukset ja jossa tutustutaan aikaan saatuun ohjelmistoon. Vaatimuksia ei siis validoida vaatimusmäärittelydokumentaation kautta, kuten perinteisesti, vaan arvioimalla vaatimusten toteutusta. Katselmuksessa asiakkaat voivat kokeilla systeemiä ja todeta itse, ovatko heidän pyytämänsä toiminnot toteutettu siten kuin he halusivat ja antaa toiminnallisuudesta palautetta. Toteutuksesta keskustellaan, ja asiakas voi kysellä systeemin yksityiskohdista ja pyytää toteutukseen muutoksia. Sekä XP:ssä että Scrumissa säännölliset tuotekatselmoinnit muodostavat pohjan niin vaatimusten validoinnille kuin tuotteen verifioinnillekin. FDD:ssä järjestään joka viikko palaveri, jossa tarkastellaan projektin edistymistä, ja DSDM:n toteutettua ohjelmistoa testataan joka iteraatiokierroksen päätteeksi.

Vaatimusten validoinnissa toteutuksen kautta on se etu, että katselmuksessa projektin osakkaat saavat ensikäden tietoa ohjelmiston vahvuuksista sekä huomaavat sen mahdolliset puutteet ja epäjohdonmukaisuudet. Kokemuksiensa pohjalta osakkaat voivat paremmin arvioida, mihin suuntaan ohjelmistoa tulisi seuraavaksi kehittää. Riippuen käytetystä kehitysmallista ohjelmisto voidaan myös julkaista ja ottaa (koe)käyttöön oikeilla käyttäjillä heti validoinnin jälkeen. Näin ohjelmisto saadaan nopeasti tuottavaan käyttöön, ja toisaalta kehittäjät saavat siitä entistä yksityiskohtaisempaa palautetta kehitystyönsä tueksi.

Koska vaatimusmäärittelyä, jota vasten ohjelmistoa voisi arvioida, ei ehkä ole olemassakaan, monissa ketterissä menetelmissä suositaan testitapausten määrittelyä vaatimusten määrittelyn yhteydessä. Paetch ja muut [2003] kuvaavat, kuinka XP-menetelmässä asiakas kehittää jokaiselle seuraavassa ohjelmistoversiossa toteutettavaksi hyväksytylle tarinalle myös testitapauksen, jolla tarina voidaan myöhemmin verifioida toteutetuksi. Vaatimuksia validoidessaan asiakkaat voivat esimerkiksi ajaa iteraation suunnittelun yhteydessä määritellyt testit ja päätellä siitä, onko iteraation tavoitteissa onnistuttu. Cao ja Ramesh [2008] huomauttavat, että ketterissä projekteissa testit ovat usein ainoa tapa linkittää abstraktit vaatimukset ja konkreettinen toteutus toisiinsa ja tuoda siten jäljitettävyyttä systeemiin. Validointitestejä ajamalla voidaan myös nopeasti kokeilla, miten erilaiset muutokset systeemiin vaikuttaisivat.

5.5. Vaatimusten ketterä hallinnointi

Koska ketterät menetelmät rohkaisevat muutoksien hyväksyntään, ne eivät yleensä tarjoa minkäänlaisia keinoja vaatimusten muutosten hallintaan tai tarkkailuun. Tämä johtaa siihen, että vaatimukset voivat myös muuttua nopeammin ja useammin kuin perin-

teisissä projekteissa, joissa vaatimusten muuttaminen saattaa edellyttää raskaan vaikutusanalyysin tekemistä sekä muutoksen hyväksyttämistä korkealla yrityksen hierarkiassa. Koska ketterissä projekteissa ei tyypillisesti katsota tarpeellisiksi ylläpitää versiohistoriatietoja vaatimuksista, vaatimusten evoluutiota ei myöskään edes voi tarkastella. Paetch ja muut [2003] huomauttavat, että tämä voi olla viisasta, koska edes perinteisissä projekteissa ei ole pystytty toistaiseksi todistamaan, että seurannasta olisi jotain konkreettista hyötyä. Muutoksien seuranta olisi myös hyvin työlästä ketterissä projekteissa, koska useimmat vaatimukset ovat aluksi hyvin epämääräisiä, ja tarkentuvat vasta projektin edetessä.

Paetch ja muut [2003] huomauttavat, että Scrumissa vaatimusten muutoksien hallintaan voidaan ainakin jossain määrin toteuttaa seuraamalla projektin Product backlog -listan kehitystä; projektin edetessä tehtävälistan tulisi lyhentyä. Myös tehtävien työmääräarvioiden ja kuvausten muuttumista voidaan seurata. Myös FDD:ssä viikoittaisissa edistyspalavereissa tuotetut raportit ovat käytännössä vaatimusten hallintaa ja seurantaa. Mikäli vaatimuslistoista tallennettaisiin uusi versio joka iteraatioon, tulisi myös vaatimusten muuttuminen dokumentoiduksi, ja myöhemmin voitaisiin tarkastella, mikä vaatimuksissa on muuttunut ja miten.

Koska ketterissä menetelmissä projektin hallinta pyörii vaatimuksien ympärillä, sekä osakkaita että kehittäjiä kiinnostaa usein ensisijaisesti vaatimusten toteutuksen edistyminen. Cao ja Ramesh [2008] kuvaavat, että vaikka iteraatioiden katselmointipalaverien tarkoituksena onkin validoida olemassa oleva systeemi ja saada palautetta sen toiminnasta, osakkaiden näkökulmasta palaverit ovat oiva keino seurata projektin etenemistä. Useimmissa ketterissä menetelmissä työn edistymistä seurataan myös päivittäin projektiin loppuun saakka, esim. XP:ssä tarkastellaan päivittäin tarinakorttien ja Scrumissa Sprint backlogin etenemistä.

Sharp ja muut [2009] kuvaavat, kuinka XP:ssä kehittäjät merkitsevät tarinaseinällä oleviin tarinalappuihin joka päivä kunkin tarinan statuksen, kuinka kauan tarinan toteuttamiseen on käytetty aikaa tähän mennessä sekä arvionsa siitä, kuinka paljon aikaa tarinan toteuttamiseen vielä menee. Esimerkiksi punaisella tähdellä voidaan merkitä kokonaan aloittamatonta tarinaa, keltaisella testausta odottavaa ja vihreällä asiakkaan hyväksymää. Iteraation edetessä kehittäjät siirtävät omia tehtävälappujaan seinällä eteenpäin, vasemmalta oikealle, sen mukaan, miten työt edistyvät, ja valmiit laput poistetaan seinältä kokonaan. Iteraation päätyttyä taulu siivotaan, ja uuden alkaessa sille kiinnitetään uudet, suoritettavat tarinalaput. Myös Scrumissa voidaan käyttää vastaavalla tavalla toimivaa tehtävätaulua työn organisoinnin ja seurannan apuna. Tarinaseinän tarkoituksena on tarjota yksiselitteinen näkymä projektin etenemiseen, jota kuka hyvänsä asiasta kiinnostunut voi käydä katsomassa.

6. Ketteryyden sivuvaikutuksista

Ketterien menetelmien ongelmana on pidetty sitä, että ne antavat hyvän tekosyyn heittää kaikki perinteiset käytännöt ja opit yli laidan myös liittyen vaatimusten määrittelyyn ja hallintaan. Ketteryyden tarkoituksena ei kuitenkaan ole aiheuttaa kaaosta, vaan päinvastoin tunnistaa muutosten väistämätön läsnäolo ja pyrkiä reagoimaan siihen tietoisesti. On kuitenkin epävarmaa, kuinka hyvin ketterä vaatimusten käsittely vastaa niihin haasteisiin, joita jatkuva muutos aiheuttaa. Ovatko korkean tason toiminnallisuuden kuvaukset, kuten käyttäjätarinat, riittävä lähtökohta vaatimusten keräämiselle? Saavatko kehittäjät riittävän yksityiskohtaista tietoa vaatimuksista juttelemalla asiakkaan kanssa? Tai riittääkö kommunikointiin kannustaminen takaamaan sen, että kaikki yksittäisten kehittäjien tietoonsa onkima vaatimuksiin liittyvä tietämys jakautuu ja jalkautuu kehitystiimin sisällä?

Turk ja kumppanit [2005] huomauttavat, että ketterät menetelmät ovat todennäköisesti vain osittain oikeassa olettaessaan, että korjaustöihin liittyvät kustannukset pysyisivät vakiona koko projektin ajan. He huomauttavat, että kaikkiin muutoksiin ei tulisi suhtautua yhtä huolettomasti, sillä muutoksen hinta riippuu ensisijaisesti muutoksen vaikutusten laajuudesta. Siinä missä yksittäisen komponentin muuttaminen voi tosiaan olla yhtä edullista projektin loppupuolella kuin sen alussakin, laajat arkkitehtuuriset muutokset voivat kaataa koko projektin, ellei muutostarpeita ole ajoissa huomattu. Sillitti ja Succi [2005] huomauttavat myös, että vaatimusten pilkkominen voi olla huomattava haaste ketterissä menetelmissä. Paitsi että vaatimusten tulisi olla riittävän pieniä yksittäisessä iteraatiossa toteutettavaksi, niiden tulisi ihanteellisesti olla myös aidosti toisistaan irrallisia, jotta ne voidaan toteuttaa missä järjestyksessä tahansa.

Ketteriä menetelmiä on syytetty siitä, että progressiivinen vaatimusten käsittely tekee kehitystyöstä tehotonta: jos esimerkiksi tiettyyn ohjelmakoodin pätkään vaikuttavat vaatimukset muuttuvat moneen kertaan, myös toteutustyö pitää tehdä toistamiseen. Jatkuva samojen asioiden pyörittely iteraatiosta toiseen voi myös turhauttaa kehittäjiä, kun taas analysoimalla vaatimus kerralla kunnolla toiminto olisi ehkä voitu saada toteutettua ”oikein” jo yhdessä iteraatiossa. On kuitenkin huomattava, että samassa tilanteessa oltaisiin myös perinteisillä kehitysmenetelmillä, jos ongelmana on se, että asiakas jatkuvasti muuttaa mieltään vaatimustensa suhteen.

Stephens ja Rosenberg [2003] kritisoivat kärkkäästi XP-menetelmää ja ihmettelevät, onko koko ketterä kehitysmalli vain keksitty päättämättömien asiakkaiden paapomiseksi. He muistuttavat, että koska vaatimukset vaikuttavat projektissa kaikkeen suunnittelusta alkaen, joistakin asioista pitäisi vain osata tehdä päätös ajoissa, eikä lykätä asioita hamaan tulevaisuuteen. Esimerkiksi päätös siitä, voiko verkkokaupan asiakkaalla olla asiakastiedoissa käytössään yksi, kaksi vai kolme osoitetta, tai voiko

osoitteen jättää kenties kokonaan pois, vaikuttaa perustavaa laatua olevalla tavalla niin rakennettavaan tietokantaan, ohjelmakoodiin kuin käyttöliittymäänkin.

Ketterissä menetelmissä on Stephensin ja Rosenbergin [2003] mukaan kehäpäätelmän makua. Menetelmät olettavat, että mitään vaatimusmäärittelyä ei tarvita, koska jatkuva asiakkaan kanssa keskusteleminen on luotettavampi tapa kerätä ja analysoida vaatimuksia. Koska vaatimusmäärittelyä ei ole, ja myös systeemin suunnitteluun käytetään hyvin vähän aikaa ennen koodaukseen ryhtymistä, myös ohjelmistosta puuttuu selkeä rakenne. Tämä johtaa siihen, että toteutusta on vaikea ymmärtää, ja sen oikeellisuutta on vaikea todentaa sitä tutkimalla. Tämän takia systeemiä pitää koko ajan testata, mutta testien suunnittelu edellyttää tietoa vaaditusta toiminnallisuudesta. Koska vaatimusmäärittelyä ei ole, ratkaisuna tähän ongelmaan tarjotaan jälleen kerran asiakasta kanssa neuvottelemista! XP:n prosessien on sanottu tukevan toisiaan ja sopivan yhteen kuin palapelin palaset. Stephensin ja Rosenbergin mukaan rakennelma muistuttaa pikemminkin korttitaloa, jossa jokainen menetelmän osa on kriittinen, koska toiset osat rakentuvat sen varaan, ja he ihmettelevät, miten näin virheherkkää menetelmää voidaan nimittää adaptiiviseksi.

Pinheiro [2003] kritisoi sitä, että ketterät menetelmät nostavat ohjelmiston nopean toimituksen kaikkein kriittisimmäksi tekijäksi ohjelmistokehityksessä, ohi ohjelmiston laadun tai muiden menestystekijöiden. Hän toteaa myös, että vaikka ohjelmiston varhaisella, inkrementaalisella toimittamisella on tunnetusti paljon positiivisia vaikutuksia, pitäisi tunnustaa se, ettei iteratiivisuus sinänsä välttämättä takaa lyhyempää kehitysaikaa. Nopeampaan ohjelmistokehitykseen voidaan päästä vain leikkaamalla varsinaista työmäärää, ja Pinheiro väittääkin, että ainoa asia, mikä esim. XP:ssä on todella viety äärimmäisyyksiin, on vaatimusten käsittely. Siihen on varattu äärimmäisen vähän aikaa iteraatioissa, jotka jo sinänsä ovat äärimmäisen lyhyitä. Pinheiro kritisoi vielä, että kiire tekee vaatimusten käsittelystä ilmeisen riskialtista. Oikea tapa yrittää korjata tilannetta ei voi olla se, että vaatimuksia käsitellään aina vain pintapuolisemmin ja yksinkertaisemmin, kuten vain jutustelemalla asiakkaan kanssa.

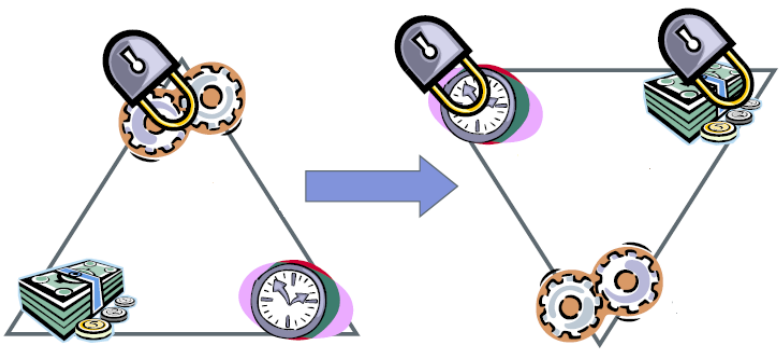
Pinheiro [2003] kaipaa lisää rehellisyyttä peliin ja vaatii, että meidän pitäisi tunnustaa, että mitään ”äärimmäisiä vaatimuksia” ei ole olemassakaan; on vain hyvin tai huonosti hoidettuja sellaisia. Pyrkimys päästä markkinoille ensimmäisenä johtaa väistämättä kompromisseihin tuotteen laadussa. Ympäristö ja käyttäjät vaikuttavat luonnollisestikin siihen, millaista laatua tuotteelta vaaditaan, ja pienet viat tai toimintaongelmat voivat olla täysin hyväksyttävä tilanne, jos nopea julkaisu on edellytys markkinoilla selviytymiselle. Tällöin puhutaan ohjelmiston ns. ”suhteellisesta laadusta”. Pinheiro kuitenkin pitää huolestuttavana, että nyt on kehitetty kokonainen joukko ns. ketteriä menetelmiä piilottamaan ja oikeuttamaan tämä vaihtokauppa. Hän korostaa, ettei hän sinänsä halua haudata ketteriä menetelmiä tai palata vanhaan, mutta hän toivoo sen tiedostamista, että ”nopeus” on vain yksi mahdollinen vaatimus muiden joukossa. Pyr-

kimyksissä kohti ketterämpiä kehitystapoja tulisi pysähtyä miettimään, miten ketteryys vaikuttaa vaatimusten käsittelyyn, mitkä ovat ketteryyden rajat ja mitä hyötyä ja haittoja aina vain nopeutetumpi tuotekehitys tuo mukaan.

6.1. Sopimisen hankaluudesta

Boehm [2002] muistelee, kuinka vanhoina hyvinä aikoina perusteellisesti tehty vaatimusten määrittely oli ensiarvoisen tärkeä ohjelmistosopimuksien laatimisen kannalta. Tiukkojen, yksityiskohtaisten vaatimusmäärittelyjen tuottaminen asetti tarjouskilpailussa kilpailijat samalla viivalle, kun jokaisen täytyi laatia tarjous juuri tietynlaisen ohjelmiston toteuttamiseksi. Toisaalta vaatimusmäärittelyt varmistivat sen, että asiakkaalle todella oli luvattu tietty toiminnallisuus. Projektin edetessä vaatimukset toki saattoivat hiukan muuttua, mutta muutosten hallinta ja toteuttaminen oli yleensä melko helppoa ja suoraviivaista, kun pohjatyö vaatimusmäärittelylle oli tehty hyvin.

Ketterällä aikakaudella sopimuksista neuvottelun täytyy muuttua, kuten kuva 9 esittää. Perinteisessä ohjelmistokehitysmallissa lyödään ensin lukkoon ne vaatimukset, jotka tuotteelta halutaan. Sen jälkeen asiakas ja toimittaja neuvottelevat vaatimusten pohjalta tuotteelle aikataulun ja budjetin. Ketterät menetelmät taas edellyttävät päinvastaista lähestymistapaa. Paetch ja muut [2003] kuvaavat, että vaatimusmäärittelyn puuttuessa ketterissä projekteissa päädytään usein kustannuspohjaiseen kauppaan, jossa sovitaan, kuinka kauan aikaa tai kuinka suurella summalla ohjelmistoa kehitetään, ja tuotteen vaatimuslista elää projektin aikana.



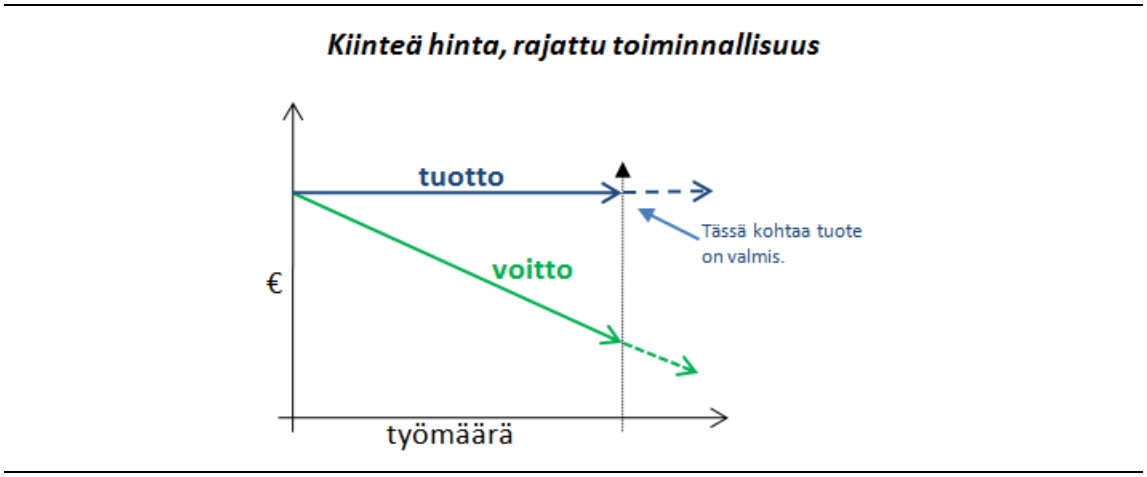
Kuva 9. Sopimuskäytäntöjen muuttuminen

Sillitti ja muut [2005] havaitsivat haastattelututkimuksessaan, että n. 75% perinteisiä ohjelmistokehitysmenetelmiä hyödyntävistä yrityksistä teki asiakkaansa kanssa kiinteähintaisen, tiettyihin toimitettaviin toiminnallisuuksiin perustuvan sopimuksen. Mikäli vaatimukset muuttuivat merkittävästi projektin aikana, hinnasta ja aikataulusta neuvoteltiin uudestaan. Sen sijaan 80 % ketteristä yrityksistä laati asiakkaansa kanssa sopimuksen, jossa toimittavan ohjelmiston vaatimuksia ei oltu etukäteen lyöty kiinni, vaan asiakas sai oikeuden määritellä projektin jokaisen iteraation yhteydessä, mitä toiminnallisuutta hän seuraavaksi halusi kehitettävän.

Mitä tarkemmin toimitussopimukset on laadittu, sitä luottavaisemmin mielin asiakas voi projektin edistystä seurata. Bechtold [2005] huomauttaa, että vaikka kehittäjien näkökulmasta olisikin järkevää analysoida vaatimuksia “lennosta”, koska ne kuitenkin muuttuvat, asiakkaat katsovat tilannetta toisesta näkökulmasta. Voidakseen tehdä hankintapäätöksen asiakkaan pitää usein saada tarkka ja luotettava arvio projektin hinnasta ja aikataulusta jo ennen projektin aloittamista. Etenkin teollisuutta ja julkistaloutta edustavat tahot ovat lisäksi tottuneet hoitamaan ohjelmistohankintansa tarjouskilpailun kautta.

Bechtold [2005] kritisoi, että ketterien menetelmien markkinamiehet ovat jättäneet täysin huomiotta ne arkiset hankaluudet, joihin useimmat yritykset joutuvat, kun nämä näkökulmat törmäävät. Paetch ja muut [2003] esittävät, että yhteistyö asiakkaan kanssa sekä projektin näkyvä eteneminen saavat asiakkaan kyllä luottamaan sopimuskumppaniinsa, muuta Bechtoldista tämä ei riitä. Hän väittää, että käytännössä varsin harvoilla asiakkailla on mielenkiintoa tilata ohjelmistojaan alun alkaen ”mustina laatikkoina”, vaikka pääsisivätkin sitten läheltä seuraamaan ja ohjaamaan kehitystyötä.

Stevens [2009] arvioi erilaisia sopimuskäytäntöjä ja niiden soveltuvuutta ketteriin hankkeisiin. *Kiinteä hinta, rajattu toiminnallisuus* (Fixed Price / Fixed Scope) -tyyppisessä urakkasopimuksessa projektin hinta on sovittu kiinteästi, ja projekti on valmis, kun toimittaja on toimittanut asiakkaalle tämän tilaaman ohjelmiston.



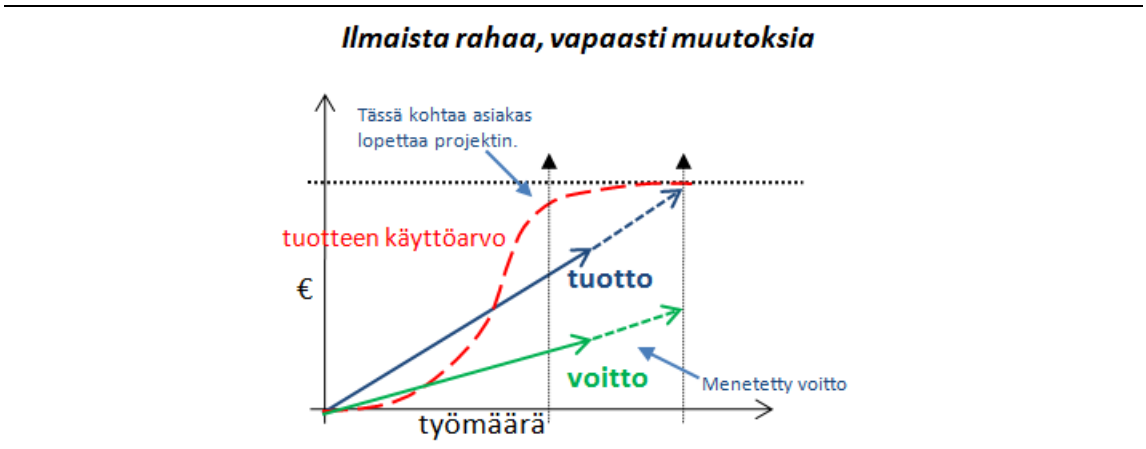
Kuva 10. Kiinteästi sovittu projekti toimittajan näkökulmasta Stevensin [2009] mukaan.

Kuten kuvasta 10 näkyy, mitä pidempää projekti kestää, sitä vähemmän voittoa toimittaja voi hankkeesta saavuttaa, koska tuotto (l. projektille sovittu hinta) pysyy koko ajan samana, joten käytännössä ohjelmiston toimittaja kantaa kaikki riskit. Toimittajan intresseissä on tällöin rajoittaa asiakkaan pyytämiä muutoksia kaikin keinoin, vaikkapa tiukan ja moniportaisen muutosvaatimusprosessin avulla. Ketterään kehitykseen tällainen malli ei sovellu, koska se käy helposti toimittajalle liian kalliiksi, ja koska luonteva yhteistyö toimittajan ja asiakkaan välillä on kilpailevista tavoitteista johtuen mahdotonta.

Laskutyö (Time and Materials) on sopimusmalli, jossa toimittaja toimittaa mitä vain asiakas keksii pyytää, ja laskuttaa asiakasta sen mukaan, soveltuu hyvin ketterään kehitykseen, mutta on asiakkaan puolelta riskaabeli. Käytännössä asiakas siis sitoutuu maksamaan kaikki kulut, mutta toimittaja ei lupaa varmasti toimittaa mitään. Toimittajalla ei ole mitään intressiä hillitä kulujaan, joten asiakas joutuu näkemään paljon vaivaa varmistuakseen siitä, että resursseja käytetään mielekkäästi. Stevens [2009] luonnehtiikin, että tämän tyyppinen projekti päättyy vasta, kun asiakas niin päättää – tai tältä loppuvat rahat.

Joskus laskutyö-sopimukseen lisätään hintakatto, jonka saavuttamisen jälkeen projektin muuttuu käytännössä kiinteästi hinnoitellun projektin kaltaiseksi. Jos taas projekti valmistuu hintakattoa nopeammin, se maksaa vähemmän. Stevens analysoi, että asiakkaan näkökulmasta malli voi olla houkutteleva, mutta käytännössä tällaisilla projekteilla on taipumusta käyttäytyä kuin kyse olisi kiinteästi hinnoitellusta sopimuksesta, koska toimittaja pyrkii osumaan tarkasti määriteltyyn maksimihintaan. Sopimuksissa määriteltävät bonukset ja sanktiot aikataulu- ja kustannusrajoitusten ylittämisestä taas ovat ongelmallisia ketterässä projektissa siksi, että ne saavat toimittajan välttämään kaikkia muutoksia, jotka voivat aiheuttaa aikataulusta lipsumista. Sanktiot eivät myöskään motivoi asiakasta pyrkimään tehokkaaseen projektin läpivientiin, sillä mitä kauemmin projekti jatkuisi, sitä halvemmaksi se tulisi asiakkaalle.

Beaumont [2008] ja Stevens [2009] esittävät ketterille projekteille *ilmaista rahaa ja vapaasti muutoksia* (Money for Nothing, Changes for Free) -sopimusmallia, jossa on säädetty ns. *kiinteä piste* (fix point), johon asti haluttu toiminnallisuus on kutakuinkin tiedossa ja työmäärät ja kustannukset on arvioitu sen mukaan. Tämän päälle lisätään n. 20-30% lisää aikaa ja kustannuksia, jolloin päästään projektin kokonaisbudjettiin. Budjetoidulla ”lisäajalla” on tarkoitus kattaa vaatimusten muutoksista ja virhearvioinneista aiheutuvia kustannuksia.



Kuva 11. Yhteistyöhön kannustava sopimusmalli Stevensin [2009] mukaan.

Budjetti ei kuitenkaan ole projektin sovittu hinta, vaan se vain määrittelee sopimukselle rajat. Työtä lähdetään tekemään kiinteän pisteen määrityksien perusteella, mutta asiak-

kaalla on oikeus jättää toimintoja pois, esittää uusia toiveita sekä vaihtaa toimintojen järjestystä keskenään. Asiakas voi myös koska hyvänsä keskeyttää projektin, maksamalla toimittajalle siihenastiset kulut sekä 20% jäljellä olevasta budjetista.

Kuva 11 esittää, miten ”ilmaista rahaa, vapaasti muutoksia” -tyyppinen ketterä projekti, jossa tuotteen arvo asiakkaan silmissä lisääntyy nopeasti iteratiivisen kehityksen myötä, käyttäytyy tuotteen toimittajan näkökulmasta. Toimittajalla on positiivinen asenne mahdollisiin aikataulu- ja toiminnallisuusmuutoksiin, koska sekä tuotto että voitto kasvavat sitä mukaa, kun projektiin tehdään työtä. Toisaalta aikaiseen projektin valmistumiseen kannustaa ns. ”ilmainen raha”, eli bonus, jonka asiakas maksaa, jos päättää projektin ennen budjetin täyttymistä. Myös asiakas on innokas päättämään projektin ajoissa, koska hänkin säästää siinä kulujaan. Beaumont [2008] ja Stevens [2009] uskovat, että tällaisella sopimuksella projektiin liittyviä riskejä voidaan jakaa tasapuolisesti asiakkaan ja toimittajan kesken.

Jos varsinainen projektisopimus uhkaa jäädä epämääräiseksi, iteraatiokohtaiset sopimukset voitaisiin ottaa avuksi täsmentämään sitä. Stevens [2009] esittää, että iteraatioiden sisällöstä voitaisiin sopia kirjallisesti, vaikka kyseessä ei olisikaan virallinen sopimus. Tärkeintä sopimuksessa olisi se, että kehittäjät ja asiakas yhdessä määrittävät ja kirjaavat, mitä iteraatiossa on tarkoitus saada aikaiseksi. Jokainen iteraatio voidaan tällöin käsittää minimuotoiseksi, kiinteästi rajatuksi sopimukseksi, jossa käyttävä aika (l. iteraation pituus), toiminnallisuus (l. iteraation backlog-lista), laatuvaatimukset (l. hyväksyntäkriteerit) ja hinta (l. kehitystiimin koko * iteraation työtunnit) on lyöty lukkoon. Asiakkaalle tällainen sopimusmalli voi tuoda tunteen projektin paremmasta kontrollista, ja vaikka kiinteästi hinnoiteltuihin projekteihin aina liittyykin jonkinlainen riski toimittajan näkökulmasta, iteraation lyhyys on omiaan parantamaan työmäärien ennustettavuutta.

Cao ja Ramesh [2008] kuvaavat, että ketterät kustannusarviot tehdään usein vain tiedossa olevien käyttäjätarinoiden pohjalta. Koska tunnetut käyttäjätarinat ovat vain osa koko toteutusta, ja voivat nekin muuttua tai jäädä toteuttamatta, tämä lähestymistapa tekee arvioista jokseenkin epäluotettavia. Palaaminen tiukkaan vaatimusten analysointiin ja vaatimusmäärittelyn tuottamiseen ei kuitenkaan auta moderneissa tarjouskilpailuissa. Asiakkaan vaatimukset voivat olla kovin epämääräisiä, ja toisaalta vaatimusten perusteellinen analysointi vie aikaa, jota ei ehkä ole annettu kovin paljon. Hidastelun seurauksena joku kilpailija saattaa onnistua nappaamaan sopimuksen itselleen. Kuitenkin liian alhaiseksi annettu hinta-arvio tarjouksessa voi tehdä projektista sen toimittajalle kannattamattoman, kun taas liian kallis arvio voi saada asiakkaan peräänty mään hankkeesta. Lisäksi liian suuret lupaukset toiminnallisuuden suhteen voivat johtaa asiakkaan pettymykseen. Bechtold [2005] ihmetteleekin, mihin tietoihin ohjelmistotarjousten pitäisi pohjautua, koska edellisistä projekteista saatuja kokemuksia ei voi hyödyntää uutta ja erilaista ohjelmistoa suunniteltaessa.

Conolly ja muut [2008] kuvaavat ketterää menetelmää käyttäjätarinoiden kehittämiseksi silloin, kun asiakasta ei ole mahdollista haastatella lisätietojen saamiseksi, esimerkiksi juuri tarjouskilpailun kohdalla. Heidän tutkimusesimerkissään koehenkilöiden tehtävänä oli laatia vaatimusmäärittely jääkiekko-otteluita esittelevän nettipalvelun pohjaksi käyttäen apunaan Conolly ja muiden laatimaa apuohjelmaa. Conolly ja muiden ajatuksena on, että vaikka ohjelmiston vaatimuksia ei tiedetä, niitä voidaan kuitenkin päätellä reaalimaailman vihjeistä, sillä harva ohjelmisto on aivan uusi lajissaan.

Tutkimuksessa koehenkilöt saivat vapaasti tutkia netistä löytyviä jääkiekkoon liittyvä sivustoja ja palveluita. Keksiessään jonkun mahdollisen käyttäjätarinan sivustojen pohjalta, esim. että ”käyttäjänä haluan tarkistaa sivustolta, onko tänään yhtään ottelua, jota voisin mennä seuraamaan”, koehenkilöt saattoivat apuohjelman avulla asettaa löytämilleen sivuille muistilappuja, joihin he merkitsivät sen tarinan, johon ko. sivut tai toiminnot liittyivät. Apuohjelman avulla käyttäjät saattoivat pitää kirjaa kehitetyistä tarinoista, ja tarinoiden linkkejä seuraamalla jälleen tarkastella tarinaan liittyviä nettisivustoja. Conollyn ja muiden [2008] mukaan käyttäjätarinat tulevat näin kuvatuksi nopeasti ja vähällä vaivalla.

6.2. Vaatimuksien esittämisestä

Useat ketterät menetelmät suosivat käyttäjätarinoiden käyttämistä vaatimuksien määrittelyssä. Käyttäjätarinoiden vahvuutena on niissä käytetty luonnollinen kieli sekä yksinkertaisuus, jotka tekevät tarinoista selkeitä ja helposti omaksuttavia. Tarina on yleensä hyvin lyhyt, jotta se mahtuisi pienelle paperilapulle, ja kirjoitettu muotoon ”<tässä roolissa> minä voin <tehdä jotain> saadakseni <jotain>”, esim. verkkokaupan yhteydessä tarina voisi kuulua ”Ostajana minä voin päivittää ostokorini tietoja poistaakseni sieltä jonkin ostoksen.” Tarina- tai tehtävälaput tulostetaan ja kiinnitetään seinään esille kaikkien nähtäville. Leffingwell [2009] korostaa fyysisen tehtävälapun tärkeyttä myös Scrum-menetelmässä. Hän kehottaa pitämään laput pieninä ja käyttämään niihin kirjoittamisessa sen verran paksua tussia, ettei lappuun mahdu paljon tekstiä. Näin tehtävälappu takaa sen, että tehtävän yksityiskohdista todella keskustellaan ennen toteuttamista. Leffingwell ja Behrens [2009] muistuttavat, että käyttäjätarina on koko ohjelmistokehitysprosessin tärkein yksikkö, sillä se kantaa mukaan liiketoiminnallisen arvon ja hyödyn, jota ketterällä kehitystyöllä on tarkoitus hankkia asiakkaalle.

Siinä missä käyttäjätarina on yksinkertainen kuvaus käyttäjän haluista ja tarpeista, kuvattuna käyttäjän näkökulmasta, käyttötapaukset ovat pitkiä, yksityiskohtaisia kuvauksia tietystä käyttötilanteesta, sen etenemisestä ja etenemisvaihtoehdoista sekä siitä, miten eri osapuolet, kuten verkkokaupan asiakas ja tilausjärjestelmä, tilanteessa toimivat. Gallardo-Valencia ja muut [2007] tutkivat, miten käyttäjätarinoiden ja käyttötapausten käyttö eroavat toisistaan, ja miten vaatimusten muoto vaikuttaa vaatimusten käsittelyyn ketterissä projekteissa. Tutkimuksessa selvisi, että käyttötapaukset nopeuttivat ja

paransivat vaatimusten hahmottamista. Ne työryhmät, joilla oli käytössään käyttötapauksina määritellyt vaatimukset, käyttivät vähiten aikaa vaatimusten analysointiin, mutta onnistuivat silti toteuttamaan vaatimuksen kaikkein parhaiten. Käyttötapaus-ryhmät myös kyselivät paikalla olleilta asiakkailta kaikkein relevanteimpia tarkentavia kysymyksiä.

Käyttäjätarinoita käyttänyt kehitysryhmä analysoi vaatimuksia kaikkein pisimpään, koska vaatimukset oli määritelty korkealla tasolla ja vaativat paljon tarkentamista. Käyttäjätarinaryhmä käyttikin paljon aikaa haastatellessaan asiakasta ja yrittäessään ymmärtää vaatimuksia. Tämä sinänsä ei välttämättä osoita käyttäjätarinoita huonoksi vaihtoehdoksi vaatimusten määrittelyssä, sillä myös käyttötapausten määrittely vie aika, ja toisaalta useissa tutkimuksissa on todettu, että usein ihmiset mieluummin keskustelevat asioista kuin lukevat dokumentaatiota. Gallardo-Valencian ja muiden [2007] koe todistaa kuitenkin sen, että ohjelmiston kehittäminen käyttötapauksien pohjalta on nopeaa ja että asiakasta tarvitaan tällöin vähemmän ohjaamaan kehitystyötä. Leffingwell ja Behrens [2009] huomauttavat, että vaikka tarkoituksena on pitää käyttäjätarinat yksinkertaisina, niihin voidaan tarpeen mukaan liittää mitä tahansa lisämateriaalia – taulukkoja, kuvia, koodinpätkiä jne. – kunhan kaikki lisätiedot on hankittu neuvotteluprosessin aikana, työstämällä käyttäjätarinaa asiakkaan ja sidosryhmien kanssa.

Rodríguez ja muut [2009] pitävät käyttäjätarinoiden käyttämisen ongelmana sitä, että käyttäjätarinoiden keskinäisiä vaikutussuhteita on vaikea tuoda esille. Asiakkaan voi olla vaikea ymmärtää, miksi jokin hänen näkökulmastaan yksinkertainen tarina yhtäkkiä edellyttää myös toisten tarinoiden samanaikaista toteuttamista. Joissakin tarinoissa saattaa myös olla yhteisenä tekijänä jokin risteävä systeemitason vaatimus, joka pitäisi osata ottaa huomioon jokaista tarinaa pohdittaessa, tai joka saattaa edellyttää että ko. tarinoita käsitellään jossain määrin ryppäänä. Rodríguez ja muut yrittivät ensin laajentaa käyttäjätarinoita sellaisiksi, että niihin olisi voinut lisätä yksityiskohtaista tietoa toiminnallisista tai systeemitason riippuvuuksista, mutta lisäykset vaikeuttivat vaatimusten hallintaa ja asiakkaan ymmärtämystä niistä. Ratkaisuna Rodríguez ja muut päätyivät käyttämään käyttäjätarinoiden ohessa *systeemitarinoita* (system stories). He kuitenkin pohtivat ratkaisun oikeellisuutta, koska perinteisesti on ajateltu, että ei-toiminnallisten vaatimusten käsittely ja hallinnointi eroaa toiminnallisista vaatimuksista.

Käyttäjätarina- ja tehtävälappujen kanssa käytetty seurantataulu on ketterissä menetelmissä yhtä olennainen tekijä kuin laput itse. Tarkoituksena on, että seinää vilkaisemalla kuka hyvänsä voi olla helposti selvillä projektin etenemisestä, mutta Sharp ja muut [2009] totesivat analyysissään, että taulun sisältö avautuu ensisijaisesti projektin kehittäjäryhmälle, joka osaa tulkita sen pieniä yksityiskohtia, kuten lappujen sijaintia taululla ja suhteessa toisiinsa. Esimerkiksi lapun sijoittaminen aivan taulun alareunaan saattaa merkitä, että lappu on yllättävien ongelmien takia toistaiseksi laitettu syr-

jään, kun taas lapun puuttuminen ja pelkkä ”haamumerkintä” taululla voi merkitä sitä, että se on jollakulla työn alla.

Paperiset tarinalaput ovat alttiita hukkumiselle tai tuhoutumiselle, eikä niistä voi helposti etsiä tietoja. Lappuja ei ole myöskään helppo jakaa sellaisten tiimin jäsenten kanssa, jotka eivät ole fyysisesti paikalla, joten kiinnostus lappujen muuttamiseksi sähköiseen muotoon on suurta. Koska sekä tarinalapuissa että seurantataulussa on kuitenkin paljon visuaalista, tulkittava informaatiota, josta osa avautuu vain taulun kokonaisuutta katsomalla, niiden siirtäminen sähköisiksi versioiksi on ongelmallista. Kääriäinen ja muut [2004] epäonnistuvat yrityksessään kehittää ketterälle tiimille seurantataulua ja lappuja simuloiva ohjelmisto, Storymanager. Tiimi hylkäsi työkalun muutaman iteraation kokeilun jälkeen todeten, että lappujen siirtely taululla ja niihin piirustelu oli paljon helpompaa kuin Storymanagerin käyttäminen. Lisäksi projektin kokonaistilanteen tarkistaminen työkalusta oli vaikeaa, kun taas taululta katsottuna tilanne hahmottui selvästi. Kääriäinen ja muut toteavat, että epäonnistuminen olisi oikeastaan pitänyt arvata, sillä heidän ehdottamansa sähköinen työkalu sotii ketteryyden periaatteita vastaan vähentäen yksinkertaisuutta sekä avointa kommunikointia. Paitsi seurannan väline, tarinataulu on koko tiimiä puhutteleva keskustelunavaaja.

Sillitti ja Succi [2005] kuvaavat, kuinka sekä perinteisessä että ketterässä vaatimusten määrittelyssä on haasteena ”yhteisen kielen” löytäminen, koska asiakas ja kehittäjät ovat tottuneet käyttämään erilaisia termejä, ja toisaalta, kehittäjillä ei ole asiakkaan asiantuntemusta ko. toimialueesta. Voidakseen keskustella asiakkaan tai käyttäjien kanssa näiden tarpeista kehittäjien tulisi kuitenkin jotenkin päästä sisälle käyttäjien kulttuuriin ja omaksua näiden käyttämä kieli ja sanasto. Surenda [2008] huomauttaa, että vaikka ketterät menetelmät ovat vahvistaneet ohjelmistokehityksen ”pehmeiden” osa-alueiden, kuten kommunikoinnin, yhteistyön ja palautteen merkitystä, menetelmät kommunikaatiokuilujen ylittämiseksi ovat vielä lapsenkengissään.

Surenda [2008] huomasi tutkimusprojektissaan, että monimutkaisessa ympäristössä etnografisten menetelmien systemaattinen käyttö sekä käyttötilanteiden mallintaminen auttoi kehittäjiä pääsemään selville käyttäjien tarpeista. Tutkimuksessa kehitettiin osakkeiden kursseja ja osto- ja myyntitilannetta esittelevää ohjelmistoa, joka voisi avustaa pörssianalyytikoita tekemään osto- tai myyntisuosituksia osakkeista. Analyytikoilla oli jo omat spesifiset menetelmänsä päätöksen tekemisen tueksi, ja he käyttävät apunaan huomauttavan monenlaisia lähteitä, joita he arvostivat eri tavalla tilanteesta riippuen. Surenda toteaa, että näiden käytäntöjen selvittäminen vain haastattelemalla analyytikkoja ei todennäköisesti olisi onnistunut.

Vaatimusten analysoinnin apuna voidaan käyttää erilaisten mallien lisäksi myös prototyyppejä. Cao ja Ramesh [2008] kuvaavat, kuinka joissakin organisaatioissa on havahduttu siihen, että ohjelmiston tuottaminen evolutionäärisen prototyypin kautta voi aiheuttaa merkittäviä ongelmia ohjelmiston skaalautuvuuden, turvallisuuden ja vakau-

den suhteen. Ketterä kehitysmalli yhdistettynä prototyyppien käyttöön voi myös johtaa asiakkaan kuvittelemaan, että prototyyppi on valmiin tuotteen aihio silloinkin, kun se on vain karkea esitys lopputuotteesta. Asiakkaan voi olla vaikea hyväksyä sitä, että projektin lähetessä loppuaan prototyypin kypsyttäminen tuotantoon sopivaksi, skaalautuvaksi ja vakaaksi ohjelmistoksi edellyttää entistä pidempiä iteraatioita.

6.3. Asiakkaan roolista

Ketterät menetelmät nojaavat jatkuvaan yhteistyöhön asiakkaan kanssa, ja jotkut niistä, kuten XP, olettavat, että asiakas on koko ajan saatavilla vastaamaan kehittäjien kysymyksiin. Paetch ja muut [2003] analysoivat, että riippuvuus asiakkaasta saattaa olla ketterien menetelmien heikoin kohta: ne luottavat liiaksi siihen, että asiakas on halukas osallistumaan kehitystyöhön, pätevä vastaamaan kaikkiin kysymyksiin ja että asiakkaalla on myös valta tehdä kaikki tarvittavat, kriittiset päätökset. Grisham ja Perry [2005] toteavat, että useat tutkimukset ovat osoittaneet, ettei tällaisen osaavan asiakkaan löytäminen ole kovinkaan helppoa tai tavallista. Joissakin ketterissä menetelmissä asiakkaan pitäisi myös kyetä toimimaan useissa eri rooleissa kehittäjien tukena, esim. testitapauksia suunniteltaessa. Grisham ja Perry huomauttavat, että kuvatun kaltainen työntekijä on varmasti arvostettu ja tuiki tarpeellinen omassa organisaatiossaankin, joten ei ole ehkä kovinkaan todennäköistä, että asiakasyritys pitäisi kannattava ko. ihmisen luovuttamista ”toisten käyttöön” vain yhden ohjelmiston kehittämiseksi.

Cao ja Ramesh [2008] toteavat, että projektit eivät useinkaan ole tyytyväisiä siihen, millaista ohjausta ja opastusta projekti asiakkaalta saa. Eräs haastatelluista projektipäälliköistä oli sitä mieltä, että hänen projektinsa asiakkaat olivat suorastaan ”huonoja” asiakkaita, koska nämä eivät kyenneet tarjoamaan relevanttia tietoa kehittäjille. On varsin tavallista, että asiakkaan edustajat eivät osaa tai halua ottaa vastuuta ”oikeiden” ja harkittujen vaatimusten antamisesta, vaan muuttavat mieltään jopa kesken iteraation. Kuitenkin ketterä projekti voi menestyä vain silloin, kun koossa on kerrallaan vähintään sen verran tunnettuja vaatimuksia kuin mitä yhdessä iteraatiossa voidaan toteuttaa.

Ketteriä menetelmiä on syytetty siitä, että raskas vastuu vaatimuksien määrittelystä, joka perinteisesti on kuulunut ohjelmistokehitysryhmälle tai erityisille vaatimusanalyytikoille, on ulkoistettu asiakkaan ongelmaksi. Koska vaatimukset ovat projektissa ”kuuma peruna”, se on ketterissä menetelmissä ovelasti lykätty asiakkaan hoidettavaksi, ja asiakas voi vain syyttää itseään, jos valmis tuote ei olekaan sellainen kuin hän toivoi. Periaate, jonka mukaan ohjelmistosuunnittelijoiden tulisi keskittyä tekemään ohjelmistoon liittyviä päätöksiä ja liikemiesten liiketoiminnallisia päätöksiä, kukin keskittyen siis omaan erityisosaamiseensa, ei ole kuitenkaan reilu, koska vaatimusten määrittelyyn tarvitaan muutakin osaamista kuin pelkkää toimialatietoutta. Vaatimusten määrittelyn tulisikin olla ryhmätyötä, vaikka asiakkaalla olisikin siinä johtava rooli.

Pinheiro [2003] kritisoi XP-menetelmää siitä, että vaatimuksien määrittelyyn osallistuvien asiakkaiden määrä on siinä viety äärimmäisyyksiin: yhden ainoa ihmisen pitäisi kyetä kaikkien mahdollisten vaatimusten lähteenä. Perinteisessäkin vaatimusten keräämisessä ongelmana on huomata kaikki mahdolliset osakkaat, joilla voi olla vaikutusta vaatimuksiin, ja saada heidän kaikkien tarpeensa ja näkökulmasta otetuksi huomioon. Silloinkin kun vaatimuksia käsitellään koko projektiryhmän voimin (kuten Scrumissa tai DSDM:ssä), ei voida mitenkään taata sitä, että kaikki tarvittava taustatieto olisi saatu kerättyä. Miten ihmeessä vaatimukset voitaisiin siis menestyksekkäästi puristaa ulos yhdestä osakkaasta, kuten XP:ssä oletetaan? Pinheiro huomauttaa, että ainoastaan yhden asiakkaan kuuleminen saattaa helpottaa vaatimusten käsittelyä, mutta se on kovin virhealtista. Mitä jos juuri tuolla asiakkaalla on virheellinen käsitys siitä, millaista ohjelmaa loppukäyttäjät tarvitsevat?

Pinheiro [2003] huomauttaa, että asiakkaan kanssa keskustellessa voi olla vaikea erottaa tämän tarpeita siitä, mitä tämä sanoo haluavansa. Ja onko sopivaa, että samat kehittäjät, jotka rakentavat ohjelman, myös selvittelevät vaatimuksia – eikö siinä piile aivan ilmeinen riski kapeille tulkinnoille? Cao ja Ramesh [2008] huomauttavat, että koska perinteinen vaatimusten määrittely pyrkii kokoamaan vaatimuksia mahdollisimman laajalti, keskustelujen lisäksi käytetään monenlaisia analysointi-työkaluja sekä katselmointeja. Näin voidaan paremmin löytää ristiriitaisuudet ja ongelmakohdat sekä varmistua siitä, että kaikki tarvittava tieto on saatu kerättyä. Ketterällä vaatimusten määrittelyllä on tässä suhteessa kovin vähän tarjottavaa. Martin ja muut [2004] taas kauhistelevat sitä työn määrää, joka ketterissä menetelmissä ladotaan asiakkaan edustajan harteille. Martinin ja muiden seuraamissa ohjelmistoprojekteissa kävi ilmi, että tuhansien erilaisten loppukäyttäjien edustaminen on asiakkaalle kovin haastava tehtävä. Eräs asiakas kommentoi, että hän olisi mielellään käyttänyt koko työaikansa projektiryhmän avustamiseen, mutta että hän ei millään voinut, koska hänen piti työskennellä myös loppukäyttäjien ja liiketoimintayksikön väen kanssa saadakseen paremman käsityksen näiden tarpeista sekä välittääkseen näille tietoa projektin etenemistä.

Ei ole kovin yllättävää, että kuten Caon ja Rameshin [2008] tutkimuksessa kävi ilmi, useilla organisaatioilla on vaikeuksia saada asiakkaan edustaja osallistumaan täysipainoisesti kehitystyöhön. Lopulta useimmat projektit järjestivät niin, että projektipäällikkö tai joku muu vastaavassa asemassa oleva alkoi toimia eräänlaisena asiakkaan sijaisena. Hankaluutena tässä oli ilmeisen intressiristiriidan ja lähteen uskottavuuden lisäksi se, että koska ketterien projektien oletettiin toimivan itseohjautuvasti, projektipäälliköt olivat käytännössä osa-aikaisia työntekijöitä, toimien samaan aikaan useissa eri projekteissa. Huomattavan suuren osan ajasta projektin kehittäjät olivat siis täysin omillaan vaatimustensa kanssa.

Martin ja muut [2004] havaitsivat, että projektitiimit usein päätyvät itse kirjoittamaan itselleen vaatimuksia. Eräässäkin tapauksessa projektipäällikkö laati kaikki vaa-

timukset, sillä hän ei syystä tai toisesta uskonut, että asiakas osaisi tai ehtisi hoitaa asiaa. Hämmästyttävintä tässä oli kuitenkin ehkä se, että kukaan projektin osallistujista ei pitänyt tätä käytäntöä minään ongelmana tai todisteena ketterien menetelmien heikkouksista. Fowler [2000] toteaa, että ilman yhteistyötä tekevää asiakasta on mahdotonta realisoida niitä hyötyjä, jotka adaptiivinen kehitysprosessi tarjoaisi. Hän kuitenkin lohduttaa, että hänen kokemuksensa mukaan vastahankaisetkin asiakkaat usein päättävät ryhtyä yhteistyöhön nähtyään ensin projektin toiminnassa ja vakuututtuaan siitä, että prosessilla voidaan saavuttaa heille itselleen merkityksellisiä tavoitteita; kyse on lopultakin ymmärtämisestä ja luottamuksen voittamisesta.

Rachaneva ja muut [2008] huomauttavat, että ketterissä menetelmissä projektipäällikön rooli muuttuu ratkaisevasti, koska vaatimukset muodostavat pohjan sekä projektin että tuotteen hallinnalle. Vaatimusten määrittely ja hallinta ei siis olekaan enää ohjelmiston kehittämiseen liittyvä tekninen tehtävä, vaan osa projektin hallintaa. Scrumissa ei edellytetä ”oikean” asiakkaan saamista mukaan projektiin, mutta asiakkaalla pitää olla silti projektissa aktiivinen edustajansa, ”Product Owner”. Product Ownerin roolissa on usein joku henkilö projektin omasta organisaatiosta, jolloin hänen pitää olla eräänlainen supersankari: hänen täytyy pitää yhteyttä asiakkaaseen ja toimia tämän tulkkina ja äänitorvena, mutta myös kyetä kuvaamaan vaatimuksia kehittäjien kielellä, kertomaan kehittäjille lisätietoja vaatimuksista sekä järjestämään vaatimukset markkinatilanteen mukaiseen tärkeysjärjestykseen. Product Ownerin pitää myös seurata projektin edistymistä ja ohjata sitä oikean asiakkaan tapaan.

Sillitti ja muut [2005] totesivat, että vaatimuksiin liittyvän tiedon esiin saattaminen on ongelma niin perinteisissä kuin ketterissäkin projekteissa. Tutkimuksessa 75% ketteristä ja 100% perinteisistä projekteista arvioi, että asiakkaiden kyky kuvata vaatimuksia monipuolisesti, tarkasti ja oikein ei ollut edes tyydyttävällä tasolla. Suurimpia ongelmia olivat epäselvyys liiketoiminnallisissa tavoitteissa, kommunikointi asiakkaan kanssa sekä ylipäätään tietämyksen ja osaamisen vähäisyys. Sillitti ja muut toteavat kuitenkin, että ketterät menetelmät yrittävät tarjota ongelmiin ratkaisuja: liiketoiminnallisten tavoitteiden oletetaan kirkastuvan projektin edetessä, kun asiakas voi testata ja kokeilla kehittyvää tuotetta, ja kasvokkain keskustelu taas on omiaan vähentämään väärinkäsityksiä.

Mielenkiintoinen havainto Sillittin ja muiden tutkimuksessa oli se, että toisin kuin ketterissä projekteissa, osa (22%) perinteisistä projekteista raportoi, että heillä oli henkilökemioiden takia ongelmia kommunikoida menestyksekkäästi asiakkaan kanssa. Silitti ja muut tulkitsevat tämän niin, että koska ketterissä menetelmissä työskennellään paljon yhdessä ja kommunikointi perustuu henkilökohtaiseen kontaktiin, se lisää ryhmän yhtenäisyyttä ja vähentää henkilöiden välistä kitkaa, eikä asiakaskaan ole samalla tavalla ryhmän ”ulkopuolinen” jäsen kuin perinteisissä projekteissa.

6.4. Priorisointi – kuka ja miten?

Jos asiakkaalta kysyy, mitkä ohjelmistoon suunniteluista toiminnallisuuksista ovat kaikkein tärkeimpiä, asiakas saattaa listata valtaosan toiminnoista ”tärkeiksi”. Asiaa ei juuri paranna se, jos vaatimuksen kategorisoi ”tärkeiden” ja ”vähemmän tärkeiden” sijaan ”välttämättömiin”, ”suositeltaviin” ja ”mahdollisiin”; asiakashan saattaa pitää kaikkia vaatimuksiaan välttämättöminä tuotteen menestyksen kannalta. Racheva ja kumppanit [2008] tuovat esiin, että ketterissä menetelmissä tilanne on väistämättä toinen, koska koko iteratiivinen työskentely lähtee siitä, että vain osa vaatimuksista voidaan toteuttaa seuraavaksi. Tällöin osa vaatimuksista saattaa jäädä kokonaan toteutumatta. Reed ja muut [2004] huomauttavat, että koska ketterissä menetelmissä vaatimuksia kerätään epäformaalisti ja pienissä erissä, vaatimusten toteuttaminen löytöjärjestyksessä saattaisi johtaa siihen, että jotkin tärkeät toiminnot unohtuvat kokonaan, tai löytyvät liian myöhään projektin onnistumisen kannalta. Tämän vuoksi vaatimusten jatkuva priorisointi on tärkeämpää ketterissä kuin perinteisissä ohjelmistokehitysmalleissa.

Ketterän priorisoinnin tavoitteena on valita, mitä vaatimuksia seuraavaan iteraatioon valitaan mukaan. Haasteena priorisoinnissa on se, miten voi tietää, mitkä vaatimukset ovat kaikkein tärkeimpiä, sekä se, että kerran tehty priorisointi ei riitä, vaan koska tilanne voi muuttua tai vaatimukset tarkentua, priorisointi on tehtävä uudestaan joka iteraatiota varten. Toisaalta liian suuret muutokset vaatimusten tärkeysjärjestyksessä voivat johtaa kehitystyön epävakauteen. Priorisointipäätöksissä pitää myös olla suhteellisen nopea, koska seuraavan iteraation tavoitteet on voitava lyödä lukkoon tunnissa tai parissa.

Kaikissa ketterissä menetelmissä asiakas, tai asiakkaan edustaja, on vastuussa vaatimuksien priorisoinnista, koska hänellä oletetaan olevan paras näkemys projektin liiketaloudellisista tavoitteista. Grisham ja Perry [2005] huomauttavat kuitenkin, että on vaikea tietää, kokevatko asiakkaat vaatimusten jatkuvan järjestelemisen mielekkääksi ja hyödylliseksi. He varoittavat, että epäpätevä tai kärsimätön asiakas voi turhautua jatkuvaan velvoitteeseensa arvioida vaatimuksia. Asiakasta voi myös ärsyttää se, miten mikään kehitystyö projektissa ei etene ilman määriteltyä järjestystä, vaikka hänen mielestään välttämättömiä tehtäviä olisi vielä vaikka kuinka paljon jäljellä; jos ne kaikki pitää kuitenkin toteuttaa, miksi vaivautua järjestelemään? Martin ja muut [2004] havaitsivat, että asiakkaat kokivat vastuunsa priorisoinnista raskaaksi. Eräs heistä totesi vihaavansa koko arviointityötä ja sanoi, ettei enää kestänyt tehtävänsä paineita; miten hän voisi olla varma siitä, mikä on tärkeää?

Jos vaatimusten priorisointi on vastenmielistä, asiakas saattaa vältellä sen tekemistä, kunnes on liian myöhäistä. Martinin ja muiden [2004] tutkimuksessa eräs asiakas tunnusti, ettei hän osannut ottaa järjestelyvastuutaan tosissaan, ennen kuin vasta projekti edetessä kävi ilmeiseksi, ettei kaikkia toimintoja millään ehdittäisi tehdä tuotteeseen.

Cao ja Ramesh [2008] taas havaitsivat, että vaatimusten priorisointi pelkän liiketaloudellisen arvon mukaan johti joissakin projekteissa suuriin ongelmiin. Koska asiakas ei osaa arvioida projektin teknisiä tarpeita, erään projektin vaatimukset toteutettiin sellaisessa järjestyksessä, että laajennettavan arkkitehtuurin luominen ohjelmiston pohjalle oli mahdotonta. Toisessa projektissa taas tietoturvallisuuteen ja tehokkuuteen liittyvät vaatimukset jäivät käsittelemättä, koska asiakas ei osannut pitää niitä kehitysvaiheessa tärkeinä, vaikka ne käyttöönotossa osoittautuivatkin kriittisiksi puutteiksi. Tietoturvan ujuttaminen ohjelmistoon jälkikäteen taas on varsin hankalaa.

Joissakin ketterissä kehitysmalleissa vaatimusten priorisoinnista vastaa asiakkaan edustaja, kun taas esim. Scrumissa Product Ownerin roolissa saattaa joku olla projektin omasta henkilöstöstä. Tällaisen ”asiakkaan” hyvänä puolena saattaa olla tekninen pätevyys, mutta kuten Ktata ja Levésque [2009] huomauttavat, vastaavasti asiakkaan taidot arvioida vaatimuksia taloudellisista lähtökohdista voivat olla heikot. Mitä laajempi projekti on kyseessä ja mitä useampia toiveiltaan ristiriitaisia osakkaita siihen on sotkeutunut, sitä mahdottomampaa Product Ownerin on keksiä kaikki osakkaita miellyttävä ja teknisesti toteuttamiskelpoinen järjestys vaatimuksille. Ktata ja Levésque epäilevät, että jos Product Owner ei uskalla tipauttaa mitään vaatimuksia listalta, tilanteessa voi käydä niin, että projekti jatkuu ja jatkuu, kunnes kaikki mahdolliset toiveet on toteutettu. Tällöin kaikki ketteryyden lupaamat edut on haaskattu, ja lopputuloksena voi sen sijaan olla vain epäkoherentisti toteutettu ohjelmisto. Ktata ja Levésque ehdottavatkin, ettei vaatimusten priorisoinnin tulisi koskaan jäädä yksittäisen ihmisen harteille, vaan asiakkaalla tai tämän edustajalla tulisi olla tukenaan projektin kehitystyötä ohjaava komitea.

Vaatimusten priorisoinnissa otetaan huomioon vaatimuksen taloudellisen tärkeyden lisäksi työmääräarvio, jonka kehittäjätiimi on vaatimukseen liittänyt, sekä joissakin tapauksissa myös kehittäjien vaatimukselle antama riskiarvio tai arkkitehtuuriset tai toiminnalliset riippuvuudet. Racheva ja muut [2008] huomauttavat, että vaikka jotkut käyttäjätarinat eivät itsessään olisi kovin mielenkiintoisia, ne saattavat olla välttämättömiä joidenkin myöhempien tarinoiden toteuttamisen kannalta. Racheva ja muut esittävät, että vaatimusten esittäminen puumaisena rakenteena helpottaisi vaatimusten riippuvuuksien ja yhteneväisyyksien tunnistamista. Näin priorisoijan olisi helpompi arvioida vaatimuksen kokonaisvaikutusta projektiin ja tehdä päätöksensä sen perusteella.

Logue ja McDaid [2008] huomauttavat, että vaatimusten työmäärä- tai kannattavuusarvioihin liittyy ketterässä ympäristössä vähintäänkin yhtä paljon epävarmuutta kuin perinteisissäkin projekteissa. Koska osa vaatimuksista alkaa selkeytyä ehkä vasta projektin edettyä, vaatimusten priorisointi etenkin projektin alkuvaiheessa on epävarmaa. Kuitenkin juuri ensimmäiset toteuttavat vaatimukset saattavat sanella paljonkin, millaiseksi ohjelmisto jatkossa rakentuu. Logue ja McDaid ehdottavat, että projektiryhmäläisten tulisikin antaa jokaisesta arvioitavasta suureesta niin optimistinen, pessimistinen kuin todennäköisinkin arvio. Projektiryhmäläiset siis arvioisivat kunkin iteraa-

tioon harkitun vaatimuksen työmäärää ja taloudellisesta arvoa sekä sitä, kuinka paljon työtä projektissa ehditään seuraavan iteraation aikana suorittaa.

Loguen ja McDaidin [2008] mukaan PERT-menetelmästä tuttua jakaumaa hyödyntämällä voidaan hallita työmääräarvioihin liittyvää epävarmuutta. PERT-menetelmä perustuu siihen, että tehtävien kestoaikojen vaihteluvälistä voidaan tehdä tilastollisia analyysejä. Hughes ja Cotterell [2006] kuvaavat, että arvioiden keskihajonnasta voidaan päätellä, kuinka suuria epävarmuustekijöitä työtehtäviin liittyy; intuitiivisestikin on selvää, että mitä kauempana optimistinen ja pessimistinen arvio ovat toisistaan, sitä epäluotettavampi annettu arvio on. Projektin tai sen jonkun iteraation työmääräarvioiden keskihajonnoista voidaan edelleen laskea PERT-tunnuslukuja, joiden avulla voidaan normaalijakaumaa vasten arvioida, mikä on todennäköisyys saada suunnitellut työt valmiiksi haluttuun päivämäärään mennessä. Toisaalta, kun työmääräarvioiden jakaumat ovat selvillä, voidaan seuraavan iteraation kohdalla hyödyntää myös Monte Carlo -simulaatiomallia ja tarkastella, mitä vaatimuksia valitsemalla päästäisiin todennäköisimmin optimaaliseen lopputulokseen. Helminen [2008] kuitenkin muistuttaa, että normaalijakaumaan perustuvat tilastolliset analyysit edellyttävät, että aineistoa – eli työtehtäviä ja niistä annettuja työmääräarvioita – on riittävästi ja että tapaukset ovat toisistaan tilastollisesti riippumattomia. Pienissä projekteissa tai iteraatioissa tämä edellytys ei välttämättä toteudu, jolloin analyysien tuottamat tulokset eivät ole luotettavia.

Grisham ja Perry [2005] ovat todenneet, että asiakkaat lykkäävät priorisointipäätösten tekemistä mahdollisimman myöhäiseen vaiheeseen joskus silloinkin, kun hankaliin vaatimuksiin tarttuminen antaisi mahdollisuuden tutkia ja ymmärtää niitä paremmin. Grisham ja Perry pohtivat, johtuuko viivyttely halusta välttää riskejä vai vaivannäköä, vai onko kyse siitä, että asiakas kokee, että hänen olisi vaikea kommunikoida tai perustella preferenssejään tiimille. Selvästi myös tilanteet, joissa asiakkaalle on tarjolla liian vähän tai liian paljon vaihtoehtoja, ovat hankalia. Ikävin tilanne on silloin, jos asiakas kokee, ettei hänellä ole valittavanaan yhtäkään houkuttelevaa vaihtoehtoa. Tällöin asiakas voi luopua koko priorisoinnista, eikä iteraation lopputulos voi millään vastata asiakkaan toiveita.

6.5. Dokumentaation puutteesta

Ketterät menetelmät eivät pidä dokumentaatiota varsinaisesti turhana, vaan ainoastaan vähemmän tärkeänä kuin toimivaa tuotetta. Ketterissä menetelmissä katsotaan, ettei systeemin laatu riipu mitenkään dokumentaatiosta, vaan se kehittyy iteratiivisesti jatkuvan integroinnin, systemaattisen testauksen ja korjailun tuloksena. Kajko-Mattson [2008] huomauttaa, että dokumentaatiota tulisi kuitenkin olla olemassa sen verran kuin on välttämätöntä kommunikoinnin tukemiseksi. Paetch ja muut [2003] kuvaavat, että tyypillisesti ketterissä projekteissa tuotetaan dokumentaatiota lähinnä asiakkaan toiveesta. Tällainen dokumentaatio on usein suhteellisen lyhyt ja keskittyy kuvaamaan systeemin toiminnallisuutta, jolloin sen ajan tasalla pitämisestä ei ole kohtuuttomasti

vaivaa. Koska ymmärrys ja tieto systeemistä muuttuu ja lisääntyy koko ajan, projektin työläiset ja osakkaat toimivat systeemin ”elävänä dokumentaationa”.

Osa ketterien projektien nopeudesta tulee siis siitä, että perinteinen dokumentointityö jätetään tekemättä. Paetch ja muut [2003] väittävät, että koska ketterät projekti tuottavat siistiä, kompaktia ja luettavaa koodia, dokumentaation puutteesta ei ehkä ole mitään mainittavaa haittaa. Sillitti ja Succi [2005] huomauttavat, että silloin kuin projektiryhmä on pieni, tällainen ketterä toiminta on mahdollista, koska kaikki tarvittava tieto voidaan välittää keskustelemalla. Kun projektin koko kasvaa tai sillä on useita osakkaita, kasvotusten tapahtuvasta kommunikoinnista tulee tehottomampaa ja epäluotettavampaa. Esimerkiksi asiakkaan kimppuun ei voi päästää suurta laumaa kyselijöitä kerrallaan, joten haasteeksi jää sen varmistaminen, että yksilön hankkimat tiedot välittyvät koko ryhmälle. Tällöin tiedonvälitys alkaa edellyttää tuekseen apuvälineitä, kuten esim. kirjallisia tuotoksia.

Turk ja muut [2005] katsovat, että ketterät menetelmät soveltuvat huonosti laajojen projektien toteuttamiseen, osin juuri kehittymättömien dokumentointikäytäntöjen takia. Mitä monimutkaisemmasta systeemistä on kyse, sitä mahdottomampaa sen ymmärtäminen on vain ohjelmakoodia tutkimalla. Suuria järjestelmiä jatkokehitettäessä suurin osa projektin ajasta kuluu siihen, että kehittäjät yrittävät ymmärtää, miten heidän suunnittelemansa muutokset vaikuttaisivat systeemin. Turk ja muut esittävät, että jonkinlaisen dokumentaation tuottaminen ja ylläpitäminen – vaikkapa mallien muodossa – olisi välttämätöntä myös ketterissä projekteissa.

Dokumentaation sivuuttaminen ketterissä projekteissa voi olla tuotannollisesta näkökulmasta katsoen tehokasta, mutta pidemmän päälle se johtaa varmasti hankaluuksiin. Cao ja Ramesh [2008] muistuttavat, että pienikin katkos tai virhe kommunikaatiossa voi aiheuttaa ongelmia silloin, kun dokumentaatiota ei voida käyttää apuna tietojen varmistamiseksi. Tällaisia informaatiokatkoksia voivat aiheuttaa esim. projektin avainhenkilöiden lähteminen tai sairastuminen, äkkinäiset muutokset vaatimuksissa, asiakkaan edustajan tavoittamattomuus tai ohjelmiston nopea muuttuminen tai kasvaminen. Cao ja Ramesh totesivat, että heidän tutkimissaan projekteissa dokumentaation puute vaikeutti merkittävästi mm. ohjelmiston skaalautuvuuden toteuttamista, ohjelmiston jatkokehittämistä sekä uusien työntekijöiden perehdyttämistä projektiin. Vaatimusmäärittelyn puute altisti projektit myös sille, että projektit alkoivat valmistumisen sijaan huomaamatta kasvaa koko ajan laajemmaksi.

Kajko-Mattson [2008] haastatteli 18 eri ketterää organisaatiota ja sai selville, että monissa nuorissakin organisaatioissa on jo ehditty huomata, millaisia ongelmia ja riskejä minimalistinen dokumentaatio aiheuttaa. Puolet organisaatioista oli esimerkiksi havainnut, että ohjelmistokehittäjillä oli vaikeuksia pysyä kärryillä kehittämästään hyvin kevyesti dokumentoidusta systeemistä, mikä johti arkkitehtuurin rapautumiseen sekä kompleksiseen ja virhealttiiseen ohjelmakoodiin. Kolmellatoista organisaatiolla oli

ollut vaikeuksia toteuttaa muutoksia olemassa oleviin järjestelmiin, ja kaksitoista organisaatiota tunnusti, että keskeisten työntekijöiden yllättävä menettäminen olisi organisaatiolle vakava ongelma. Puolet organisaatioista valitti, että riittämätön dokumentaatio oli jo johtanut siihen, että kommunikoinnista oli tulossa rasite, ja jatkuvat keskustelut ja palaverit varastivat aikaa varsinaiselta työnteolta.

Kajko-Mattsonin [2008] tutkimuksessa kävi ilmi, että ainoastaan suullisesti välitetty tieto oli ongelmallista myös siksi, että kehittäjät unohtivat keskustelut ajan myötä, ja monia havaittuja virheitä toistettiin yhä uudestaan. Bechtold [2005] on myös havainnut tämän ongelman, ja kuvaa, kuinka mielikuvitus – tai sen puute! – yhdistettynä sekä asiakkaan että kehittäjien yhtä huonoon muistiin voi aiheuttaa pahoja ristiriitoja projektissa ja vaarantaa menestyksekkään yhteistyön. Bechtold ehdottaa, että ainakin iteraatioiden suunnitelmat pitäisi dokumentoida ja tarkistuttaa asiakkaalla ennen työhön ryhtymistä. Hän huomauttaa, että vaikka ohjelmistokehitysprojektissa on monenlaisia ongelmia, joiden kanssa täytyy vain tulla jotenkin toimeen, ei ole mitään syytä antaa huonomuistisuuden rampauttaa projektia. Tietysti siinä tapauksessa, että asiakkaan toiveet todella muuttuvat joka viikko, muutosten dokumentoinnista voi olla varsin vähän apua.

Suulliseen kommunikointiin liittyvien hankaluuksien välttämiseksi joissakin organisaatioissa on Kajko-Mattsonin [2008] mukaan otettu käyttöön ns. Communication Ownerin rooli. Communication Ownerin tehtävänä on levittää niin sanallista kuin kirjallistakin tietoa systeemistä. Kajko-Mattsonin huomauttaa, että tällaisen roolin olemassa olo ei kuitenkaan ole mikään tae siitä, että asiat tulisivat riittävästi dokumentoiduksi jatkossakaan.

6.6. Ei-toiminnallisista vaatimuksista

Hofmann ja Lehner [2001] kuvaavat, miten eräissä perinteisillä menetelmillä toteutetuissa projekteissa asiakkaita pyydettiin arvioimaan vaatimusten määrittelyn tuloksia niin vaatimuksien kattavuuden kuin vaatimusmäärittelyn laadunkin näkökulmasta. Annettujen arvosanojen keskiarvo asteikolla 1-10 oli varsin alhainen: 6,6. Asiakkaat kritisoivat, että vaatimusten kattavuus oli heikko, etenkin mitä tuli ei-toiminnallisiin vaatimuksiin, kuten suorituskykyyn tai toimintavarmuuteen. Vaikutti siltä, että projektin johto oli täysin tyytyväinen siinä vaiheessa kun systeemin tietorakenteet oli saatu määriteltyä. Muu ohjelmistoväki taas oli kiinnostunut vain toiminnallisista vaatimuksista ja siitä, miten ne voitaisiin toteuttaa. Asiakkaat valittivat, että kun tietotyypit ja funktiot varastivat kaiken huomion, systeemi kokonaisuutena jäi hahmottamatta, ja se johti puutteellisesti määriteltyihin ei-toiminnallisiin vaatimuksiin.

Ketterät menetelmät keskittyvät toiminnallisuuksien, tarinoiden, toteuttamiseen, joten ei-toiminnallisten vaatimusten tilanne ei ole olennaisesti parantunut vuosituhannen alusta. Sillitti ja Succi [2005] huomauttavat, että ketterät menetelmät kuvaavat vaatimusten keräämistä ja analysointia sekä mahdollisia tekniikoita ylipäätäänkin hyvin niukasti, ja se, miten ei-toiminnallisten vaatimuksien uskotaan löytyvän, on menetelmi-

en kuvauksissa täysin sivuutettu. Paetch ja kumppanit [2003] huomauttavat, että ei-toiminnalliset vaatimukset tulisi saada selville ensi tilassa, sillä ne vaikuttavat suuresti kehitystyöhön, esim. tietokannan, ohjelmointikielen tai käyttöjärjestelmän valintaan. Tavallista kuitenkin on, että vasta ohjelmiston kypsyttyä monimutkaisempien käyttäjätarinoiden tasolle huomataan, että esim. ohjelmiston eri komponenttien rajapintoja ja viestiliikennettä täytyy muuttaa, jotta systeemi toimisi paremmin suunnitellussa ympäristössään.

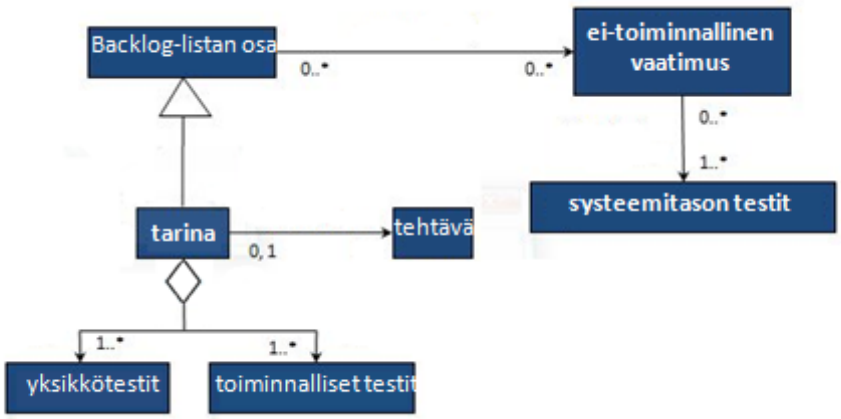
Uutta ketterissä menetelmissä on se, että vastuu vaatimusten määrittelystä on nyt siirretty kehitystiimiltä asiakkaalle. Cao ja Ramesh [2008] harmittelevat, että asiakkaat usein keskittyvät vain miettimään, mitä systeemin tulisi tehdä, ja jättävät täysin huomiotta, miten sellaisten tekijöiden kuin laajennettavuus, ylläpidettävyys, siirrettävyys, turvallisuus tai suorituskyky tulisi ilmetä tuotteessa. Käytettävyysasioista sen sijaan saattaa tulla puhe, koska ohjelmiston koekäyttö yleensä paljastaa puutteita sen käyttöliittymässä. Rodríguez ja muut [2009] huomauttavat, että asiakkaalta puuttuu usein teknistä näkemystä ei-toiminnallisten vaatimusten arvioimiseksi. Ongelmana ei ole niinkään se, etteivätkö he osaisi muotoilla vaatimuksia, vaan se, että asiakkaat eivät hahmota, mitä seuraamuksia jokin ei-toiminnallinen vaatimus voi kehitystyön myöhemmässä vaiheessa aiheuttaa, ellei sitä osata ottaa huomioon heti alkumetreillä.

Rodríguez ja muut [2009] kuvaavat, että vaatimuksia kerättäessä asiakkaan näkemys, joka pohjautuu hänen toivomiinsa toiminnallisuuksiin, eroaa usein ohjelmiston kehittäjien, jotka yrittävät johtaa toiminnoista toteuttamiseen liittyä huomioita, näkemyksistä. Tämän ei pitäisi olla mikään ongelma vaan mahdollisuus katsoa asiaa monesta näkökulmasta, ja tunnistaa kaikki olennaiset vaatimukset. Vaatimusten määrittelyn tulisi olla tiimityötä. Rodríguez ja muut esittävät, että myös kaikki ei-toiminnalliset vaatimukset voisi yrittää kuvata käyttäjätarinoiden avulla. Erottelu eri kategorioihin voi olla kuitenkin tarpeen mm. testausta ajatellen. Tällöin esimerkiksi turvallisuuteen liittyvät huolenaiheet voisi esittää vaikka *väärinkäyttäjätarinoiden* (abuser story) kautta. Racheva ja kumppanit [2008] ehdottavat, että ei-toiminnallisten vaatimusten hahmottamista ja käsittelyä helpottamaan niihin voisi liittää myös arvion siitä, miten todennäköinen ko. vaatimus on systeemille. Jos asiakas tietää esimerkiksi varmasti, että systeemillä tulee olemaan tuhansia käyttäjiä, skaalautuvuuden vaatimus pitää ottaa huomioon heti alusta saakka.

Ei-toiminnallisten vaatimusten kirjaaminen omiksi käyttäjätarinoikseen helpottaa vaatimuksien huomioon ottamista, mutta niiden käsittely toiminnallisten vaatimusten rinnalla voi osoittautua hankalaksi. Jos ei-toiminnallisen vaatimuksen arvo on kovin pieni, sitä ei ehkä koskaan ryhdytä toteuttamaan. Vaatimuksen suuri arvo taas vääristää kokonaiskuvaa iteraation vaatimusten tärkeydestä. Jos vaatimuksen työmääräarviot ovat kovin suuret, mutta vaatimus toteutetaan käytännössä osana jotain toiminnallista vaatimusta, projektissa kerätty tilastollinen aineisto iteraatiossa tehdystä työstä vääristyy –

kun taas liian pieni työmääräarvio saattaa saada tilanteen näyttämään siltä, ettei työ etene lainkaan. Koko projektin ajan kestävät ei-toiminnalliset käyttäjätarinat taas alentavat työmoraalia, koska niiden eteen joutuu koko ajan tekemään työtä näkemättä ”tulosta”. Ei-toiminnalliset käyttäjätarinat voivat kuitenkin antaa hyvän pohjan funktionaalisten testitapausten suunnittelulle silloin, kun ne liittyvät suoraan joihinkin toiminnallisiin vaatimuksiin tai tulevat esiin niiden yhteydessä.

Sillitti ja Succi [2005] esittävät, että ketterissä menetelmissä ei-toiminnallisten vaatimuksien tunnistaminen on onneksi helpompaa kuin perinteisissä projekteissa, eikä niiden määrittely ole siksi niin tärkeää. Koska jokaisen iteraation tuloksena on valmis ohjelmisto, asiakas voi heti testata tuotetta ja havaita mahdolliset ongelmat nopeasti. Leffingwell [2009] on samoilla linjoilla ja huomauttaa, että useimmat ei-toiminnalliset vaatimukset liittyvät pikemminkin koko systeemin laatuun kuin yksittäisiin käyttäjätarinoihin. Siksi niiden tunnistaminen, validointi ja arviointi pitäisikin tehdä systeemiä tarkastelemalla eikä suoraan käyttäjätarinoiden määrittelyn tai testaamisen kautta.



Kuva 12. Ketterän vaatimusten määrittelyn artefaktit Leffingwellin [2009] mukaan.

Leffingwellin [2009] mallissa projektin työlista koostuu käyttäjätarinoista (ja muista tarpeellisista toimista, kuten virheenkorjauksista), jotka toteutetaan useamman osatehtävän avulla. Kuten kuvassa 12 on esitetty, jokaista tarinaa kohti on määritelty sekä yksikkötestit että toiminnalliset testit. Jokaista työlistan kohtaa vasten taas on määritelty joukko ei-toiminnallisia vaatimuksia, ns. hyväksyntäkriteerit, joista systeemitason testit johdetaan. Esimerkiksi käyttäjätarina voi olla seuraavanlainen: ”Talon asukkaana haluan tietää päivittäisen energiankulutukseni, jotta voisin ymmärtää, miten voisin jatkossa vähentää energiakulujani”. Tähän käyttäjätarinaan liittyvät hyväksyntäkriteerit voisivat olla vaikka seuraavanlaiset: ”Ohjelma lukee mittarin 10 sekunnin välein.”, ”Mittarin paneeli näyttää jokaisen luetun lukeman.”, ”Ohjelma piirtää ruudulle päivitetyn energiankulutusgraafin 15 minuutin välein.”, ja ”Ohjelman ei tarvitse näyttää edellisten päivien graafeja (toinen tarina).” Kun jokin työlistan kohta valmistuu, se validoidaan ajamalla systeemitason testit tuotetta vasten.

6.7. Asiakastyytyväisyydestä

Grisham ja Perry [2005] analysoivat, että yritykset mitata tai määritellä asiakastyytyväisyyttä ovat usein kovin ohjelmistokeskeisiä. Projektin onnistumista mitataan sillä, onnistuttiinko asiakkaan vaatimukset toteuttamaan, valmistuiko projekti suunnitellun aikataulun mukaisesti ja budjetin puitteissa, ja kuinka laadukasta – esim. selkeää ja virheetöntä – tuotettu ohjelmakoodi on. Historia tuntee kuitenkin monia sellaisia projekteja, jotka ovat valmistuneet sinänsä onnistuneesti, mutta silti asiakas on ollut lopputulokseen tyytymätön; selvästikään siis projektin tavoitteiden saavuttaminen ei takaa asiakkaan onnellisuutta. Grisham ja Perry [2005] ehdottavat, että monimutkaisten analyysien sijaan asiakkaalta voisi ehkä vain kysyä, ryhtyisikö hän vastaavaan hankkeeseen uudestaan, ja jos ei, miksi ei?

Ketterien menetelmien vahvuutena on se, että jatkuva asiakkaan kanssa toimiminen auttaa keskittymään juuri niihin asioihin, joita asiakas pitää tärkeänä. Pitää kuitenkin huomata, että vaikka kommunikointia olisikin määrällisesti paljon, myös keskustelujen laatu on olennaista sen kannalta, että tiedot välittyvät oikein. Grisham ja Perry [2005] pitävät XP:n suurena voittona pidettyä C3-projektia mielenkiintoisena esimerkkinä asiakkaan ja ohjelmiston kehittäjien odotusten ja havaintojen välisestä kuilusta. C3-projektissa toteutettiin Daimler-Chryslerille uusi palkanlaskentajärjestelmä, ja projektin osallistujat maalasivat projektista jälkikäteen auvoisan kuvan ja julistivat toimittaneensa asiakkaalle laadukkaan ohjelmiston aikataulun ja budjetin puitteissa. XP-kriitikot Stephens ja Rosenbert [2003] ovat kuitenkin keskustelupalstoja seuraamalla tulleet siihen tulokseen, ettei projekti ehkä ollutkaan mikään loistava onnistuminen, ainakaan kaikkien mielestä. Vain noin kolmasosa vaatimuksista saatiin valmiiksi, ja hanke pilasi XP:n maineen kokonaan asiakkaan silmissä.

Beck [2007] pitää ketterien menetelmien suosion yhtenä suurimpana syynä sitä, että lyhyet iteraatiot lisäävät ohjelmistokehitysprosessin läpinäkyvyyttä ja mahdollistavat sellaisen vastuullisuuden ja rehellisyyden, johon ei ole enää pitkään aikaan päästy perinteisillä menetelmillä. Grisham ja Perry [20005] huomauttavat, että juuri läpinäkyvyys on kuitenkin joidenkin projektipäälliköiden suurimpia huolenaiheita ketterissä menetelmissä: he pelkäävät, että asiakas joutuu todistamaan sitä, kuinka projektihenkilökunta sählää ja sekoilee. Silloin kun kaikki sujuu projektissa hyvin, sen seuraaminen läheltä todennäköisesti lisää asiakkaan luottamusta ja tyytyväisyyttä projektiin. Toisaalta, jos projekti lipsuu aikataulusta tai kohtaa teknisiä vaikeuksia, hankaluuksia on mahdoton piilottaa asiakkaalta. Tällaisten epäonnistumisten todistaminen tekee asiakkaasta levottoman ja aiheuttaa tyytymättömyyttä ja epäluottamusta projektia kohtaan, ja pahimmassa tapauksessa asiakas voi jopa alkaa yrittää itse ratkaista projektin organisointiin ja henkilöstöön liittyviä asioita.

Grisham ja Perry [2005] huomauttavat, että myös muut ketterille menetelmille tyypilliset piirteet kuin läpinäkyvyys voivat aiheuttaa kitkaa asiakkaan ja projektiryh-

män välille. Useissa ketterissä menetelmissä asiakkaan edellytetään osallistuvan aktiivisesti kehitystyön ohjaamiseen, mutta päätellen siitä, miten usein projekteilla on ongelmia saada asiakkaan edustajia avukseen, asiakkaat eivät pidä tätä käytäntöä itselleen merkityksellisenä saati hyödyllisenä. Projektiryhmän esittämät vaatimukset siitä, kuinka asiakkaan pitäisi itse panostaa enemmän jo ostamansa ohjelmiston kehittämiseen, voivat ärsyttää asiakasta. Grisham ja Perry [2005] huomauttavat myös, etteivät kaikki asiakkaat halua tai osaa sopeutua ketterään maailmaan, ja että asiakkaalla voi olla omia syitä vaatia esimerkiksi tietynlaista dokumentaatiota tai virallisten koodikatselmointien järjestämistä. Näitä toivomuksia tulisi kunnioittaa, sillä yritykset pakottaa asiakas ketteriin käytäntöihin voivat vakavasti vahingoittaa asiakassuhdetta.

Sillitti ja muut [2005] selvittivät tutkimuksessaan sitä, kuinka tyytyväiset ohjelmistokehittäjät olivat suhteeseensa asiakkaaseen vaatimusten määrittelyn aikana. Tutkituista ketteristä projekteista 75% oli sitä mieltä, että vaatimusten kerääminen oli ollut ”hyvin tyydyttävä” prosessi, kun taas perinteisiä menetelmiä käyttäneistä projekteista vain 50% piti prosessin onnistuneisuutta ”tyydyttävänä”, ja jopa 37% katsoi, ettei vaatimusten määrittely ollut sujunut juurikaan toivotulla tavalla. Selvästikin siis ketterien menetelmien käyttö lisää ainakin ohjelmistokehittäjien tyytyväisyyttä asiakkaaseen.

Ferraira ja Cohen [2008] saivat selville 59 ketterää projektia kattaneessa haastattelututkimuksessaan, että mitä enemmän projektit noudattivat ketteriä periaatteita, kuten työn jakamista iteraatioihin, jatkuvaa integrointia, nopeaa palautteen antoja ja siihen reagointia, testauslähtöistä suunnittelua ja yhteisvastuullisuutta, sitä tyytyväisempiä asiakkaat olivat sekä projektiin että sen tuotoksiin. Mielenkiintoista oli, että kaiken kaikkiaan asiakkaat olivat hivenen tyytymättömämpiä itse kehitysprosessiin kuin lopputuloksena tuotettuun ohjelmistoon. Tämä saattaa johtua siitä, että ketterissä menetelmissä asiakkaan edustajalla on raskas rooli, mikä saattaa pitemmän päälle johtaa väsymiseen ja turhautumiseen.

Ferraira ja Cohen [2008] havaitsivat myös, että mitä tyytymättömämpi asiakas oli kehitysprojektiin, sitä huonommaksi hän arvioi myös tuotetun ohjelmiston. Ferraira ja Cohen ehdottavatkin, että asiakastyytyväisyyden varmistamiseksi ohjelmistokehittäjien tulisi tarkkailla asiakkaan mielialaa projektin aikana; mitä tyytyväisempi asiakas on yhteistyöhön ja projektin etenemiseen, sitä tyytyväisemmin hän todennäköisesti ottaa tuotteenkin vastaan.

6.8. Validoinnista

Ketterät menetelmät ovat tyypillisesti jokseenkin piittaamattomia, mitä tulee tuotetun ohjelmiston laadun arviointiin; jos tuote kelpaa asiakkaalle, sen on riittävän hyvä. Perinteisesti on kuitenkin ajateltu, että laatu on jotain, joka voidaan määritellä mitattavilla suureilla, ja jota vertailla eri tuotteiden välillä. Kun tätä aspektia ei ole, nousee vaatimusten validointi entistä tärkeämpään rooliin.

Käytännössä vaatimukset validoidaan ketterissä menetelmissä testaamalla aikaansaatua tuotetta. Leffingwell [2009] korostaa, että toimintojen jatkuva testaaminen on keskeinen osa toimivaa ketterää kehitystä ja että yhtäkään tehtävää ei pitäisi ottaa työlistalle, ellei ole selvää, miten tehtävän onnistuneisuus voidaan validoida. Jokaista käyttäjätarinaa kohden pitäisi siis olla suunniteltuna joukko yksikkö- ja toiminnallisuustason testitapauksia. Käyttäjätarinat itsessään ovat vain tilapäisiä muistutuksia tehtävästä työtä, kun taas testit ovat pysyvä esitys järjestelmän toiminnallisuudesta. Leffingwell pitää myös tärkeänä, että Product Owner tarkastaa ja hyväksyy jokaiseen toteutuslistalle otettuun käyttäjätarinaan liittyvät testitapaukset, jotka myös osaltaan ohjaavat toteutustyötä.

Leffingwell [2009] myöntää, että testauksen järjestäminen erilaisissa ympäristöissä voi olla haastavaa, mutta testaamatta jättäminen on varma tapa ampua itseään jalkaan. Esimerkiksi verkkokaupan tarjoamien tilauspalvelujen yksikkötestaamisen suunnittelu vaatii mielikuvitusta, mutta Leffingwell muistuttaa, että ohjelmistokehitystyössä on lopultakin kyse juuri erilaisten ongelmien ratkaisemisesta. Jos itse kehitystyö on helpottunut esimerkiksi erilaisten valmiiden kirjastojen kautta, kenties testaaminen voitaisiin keskittää noihin kirjastoihin? Tilanteissa, joissa ohjelmistossa ei ole lainkaan varsinaista käyttöliittymää, validointi voidaan yrittää tehdä esimerkiksi simuloimalla niitä olosuhteita ja tilanteita, joihin ohjelmisto on tarkoitettu, ja tarkkailemalla ohjelmiston käyttäytymistä siellä.

Cao ja Ramesh [2008] totesivat tutkimuksessaan, että juuri jatkuva testaaminen on se suositeltu käytäntö, josta ketterissä menetelmissä useimmiten luistetaan. Testien kirjoittaminen etenkin ennen varsinaista ohjelmakoodia on haastavaa, eivätkä kehittäjät ole tottuneet siihen. Testitapausten suunnittelu myös edellyttää yhteistyötä asiakkaan kanssa, jotta testattava vaatimus tulisi paremmin ymmärretyksi. Cao ja Ramesh toteavat, että koska useilla organisaatioilla on jo vaikeuksia saada asiakkaan edustaja osallistumaan vaatimusten määrittelyyn, ei ole yllättävää, että testitapauksien suunnittelussa kehittäjät saattavat jäädä täysin oman onnensa nojaan. Pinheiro [2003] kritisoi XP:n käytäntöjä, joissa asiakas itse on suoraan vastuussa vaatimuksiin liittyvien testitapauksien kehittelemisestä. Hän kuvaa, että asiakkailta puuttuu usein tarvittavaa kokemusta testien määrittelystä, ja että XP ei anna mitään ohjeita siitä, miten käyttäjätarina tulisi muuntaa testitapauksiksi.

On epävarmaa, ovatko pelkät käyttäjätarinat todella riittävän monipuolisia, jotta yksikkö- ja hyväksyntätestaus tulisi niitä tutkimalla katetuksi edes kohtuullisella laajuudella. Pinheiron mukaan ketterästä testauksesta tekee ongelmallisen sekin, että se ei ole toteutuksesta erillinen toiminto, jolloin on suuri riski, että suunnitellut testitapaukset ovat mukautuneet suunniteltuun toteutukseen. Tällaiset testitapaukset soveltuvat siis vain validointikäyttöön – niiden toimivuus todistaa sen, että vaatimukset on toteutettu – mutta mitään oikeita ongelmia sellaisilla testeillä on vaikea löytää. Testitapausten luon-

ti ennen ohjelmakoodin kirjoittamista ei Pinheiron mukaan ratkaise mitään, koska silloin ohjelmisto saatetaan laatia sellaiseksi, että se varmasti selviytyy testeistä.

Turk ja kumppanit [2005] esittävät, että oikeiden käyttäjien käyttäminen ohjelmiston toiminnallisuuden validoinnissa voisi olla hyvä idea, koska projektissa mukana olevat asiakkaan edustajat saattavat olla jo liian ”koulutettuja” arvioimaan tuotetta. Voi kuitenkin olla vaikea saada käyttäjät innostumaan kokeiluasteella olevan ohjelmiston kanssa leikkimisestä. Lacey [2008] kuvaa, kuinka erään hänen kokemansa ketterän projektin isoimpia ongelmia oli se, etteivät asiakasorganisaatioon kuuluneet tuotteen tulevat loppukäyttäjät vaivautuneet testaamaan julkaistua tuotetta käytännössä, vaikka asiakas väittikin niin tapahtuneen jokaisen iteraation päätteeksi. Kun tuote sitten lopulta luovutettiin asiakkaalle, loppukäyttäjät löysivät siitä monenlaisia vikoja, joiden korjaaminen viivästytti pahasti tuotteen suunniteltua käyttöönottoa ja aiheutti huomattavan budjettiylityksen.

6.9. Ketteryyden skaalautuvuudesta

Ketteryyden yhtenä ongelmana on pidetty sitä, etteivät käytännöt sovellu laajojen projektien toteuttamiseen. Erityisesti Scrum ja XP on alun perin kehitetty pienten itsenäisten kehitystiimien ohjenuoriksi, eivätkä ne ota kantaa useammista tiimeistä koostuvien projektien tai kokonaisten organisaatioiden, jotka voivat koostua erillisistä toteutus-, testaus-, käyttöliittymäsuunnittelu- ja markkinointiosastoista, toiminnan ohjaamiseen. Erilaisia ryhmätyötyyppisiä laajennuksia Scrumiin ja XP:hen on kyllä esitetty laajastikin, mutta on vaikea arvioida, missä määrin nämä ad hoc -tyyppiset lisäykset todella soveltuvat ketterään kehitykseen.

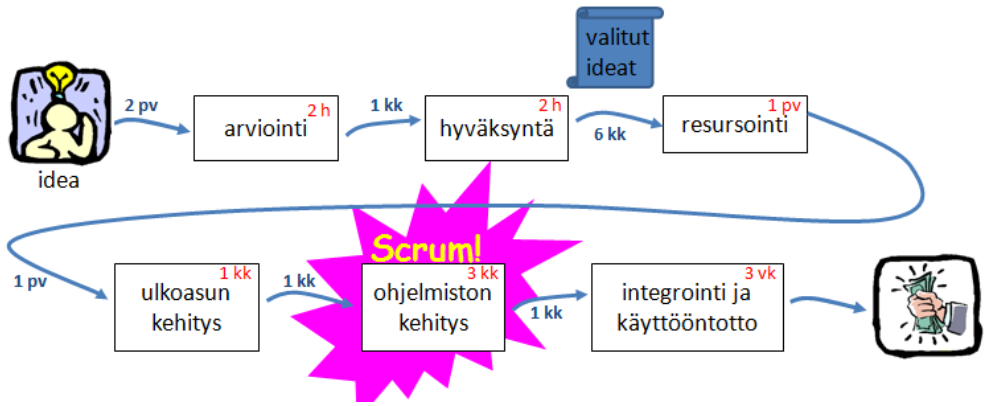
Ktata ja Levésque [2009] ovat pohtineet, miten tuotepäällikkö voisi laajoissa projekteissa parhaiten muuntaa asiakkaiden liiketoiminnalliset tarpeet lukuisiksi tarinoiksi ja jakaa ne eri kehitystiimeille toteutettavaksi. Ktata ja Levésque [2009] ehdottavat, että tarinoiden sijaan asiakkaiden tarpeet voitaisiin määritellä systeemin *päämäärien* (goal) kuvaamisen kautta. Systeemin päämäärät kuvaavat niitä tavoitteita, jotka systeemin tulisi saavuttaa eri toimijoiden yhteistoiminnan kautta. Ktata ja Levésque arvelevat, että asiakkaiden olisi huomauttavasti helpompaa laittaa päämäärät tärkeysjärjestykseen kuin kaikki mahdolliset tarinat, sillä päämääriä on vähemmän ja ne ovat samalla abstraktiotasolla. Tämän jälkeen yksittäiset vaatimukset, kuten käyttäjätarinat, pitäisi linkittää päämääriin. Näin tarinat saataisiin hierarkiseen järjestykseen yksiulotteisen tärkeyslistan sijaan.

Schaber [2008] esittää, että skaalautuvuus voidaan saavuttaa hierarkisilla Scrum-tiimeillä, joita pyöritetään synkronoidusti. Tässä mallissa tiimeillä on paitsi omat päivittäiset palaverinsa, myös yhteistyökokouksia, jossa eri tiimien edustajat tapaavat viikoittain. Käytännössä jokainen tiimi lähettää oman edustajansa seuraavan hierarkiatason palaveriin, ja sieltä edelleen valitaan edustaja seuraavan hierarkiatason viikkopalaveriin tasolle niin, että lopulta kaikki organisaation ryhmät ovat kuin osa toimivaa rat-

taistoa, joka on koko ajan tietoinen toiminnastaan. Schwaber myöntää, että tällainen ”Scrum-of-Scrums” on haastava kokonaisuus saada toimimaan, mutta väittää, että se on parempi vaihtoehto kuin moni muu organisaatiomalli.

Shalloway [2008] kuitenkin pitää epärealistisena olettaa, että jokin keskikokoinen ohjelmistotalo, jolla on useita erilaisia ohjelmistotuotteita ja -komponentteja kehitettävänään, voisi organisoida toimintansa pelkän Scrumin avulla. Shallowayn esittämää tilannetta mukaillen voidaan esittää seuraavanlainen esimerkki, jossa yksi kehitystiimi työskentelee tuotteen A parissa, ja kehittää sitä varten komponenttia K. Myöhemmin käynnistyvä tuotetiimi B tarvitsisi muiden osien ohella myös komponenttia K tuotteeseensa, joskin pienillä muutoksilla. Scrum-sääntöjen mukaan tiimi A ei voi alkaa muokata K-osaa B-tiimin tarpeisiin, vaikka kyse olisi pienistä muutoksista, koska A-tiimin asiakas ei hyödy työstä mitenkään. Todennäköisesti A- ja B-tiimien yhteistä tarvetta ei edes huomata, jos kukaan ei seuraa mitä tuotekehityksessä kokonaisuudessaan puuhataan. Kuitenkin kokonaisuuden kannalta olisi järkevintä, jos A-tiimi voisi vähän avustaa B-tiimiä – mutta Scrumissa ei pyritä optimoimaan toimintaa *kokonaisuuden* suhteen.

Taipale ja Tanninen [2009] taas esittävät esimerkin, jossa yrityksessä otettiin käyttöön Scrum tuotekehitystä tehostamaan, mutta jostain kumman syystä yrityksessä ei silti kyetty tuottamaan nopeammin uusia tuotteita markkinoille.

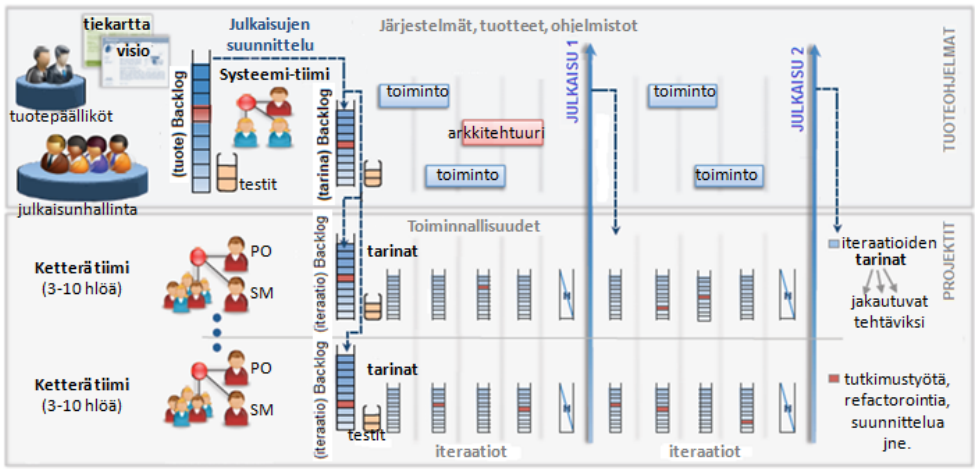


Kuva 13. Tuotteen kehityspolku ja Scrumin vaikutusalue

Tuotekehitysketjun analyysi kuvassa 13 paljastaa, että ketterät menetelmät ovat käytössä vain ohjelmiston kehitysvaiheessa, ja ennen siihen vaiheeseen pääsyä aikaa on ehditty tuhlata jo yli seitsemän kuukautta. Scrumin vaikutusalueen ulkopuolelle jäävät tuotteen ideointi, tuotevision ja liiketoimintamallin kehittäminen, tuotteistaminen, aikataulutus ja resursointi, käytettävyyssuunnittelu sekä myös integrointi ja käyttöönotto. Taipale ja Tanninen toteavat, että Scrum ei yksinään riitä yrityksen ohjenuoraksi, sillä se ei kerro, miten koko liiketoiminta voitaisiin organisoida ketterällä tavalla.

Myös Leffingwell [2009] toteaa, että siinä missä Scrum toimii hyvin tiimitasolla, menetelmän skaalaaminen kattamaan koko yritysorganisaatio vaatii lisäponnisteluja. Hän huomauttaa lisäksi, että käyttäjätarinat ovat aivan liian pieniä kokonaisuuksia ku-

vaamaan tavoitteita yrityksen tuotekehitystasolla, ja lisäksi niitä on aivan liikaa yhden tuotepäällikön seurattavaksi. Jos esimerkiksi yksi kahdeksan henkilön tiimi kykenee saamaan aikaiseksi 15 käyttäjätarinaa yhden sprintin (joka kestää kaksi viikkoa) aikana, ja vastaavia kehitystiimejä on valvottavana vaikkapa 25 kappaletta, ehtivät tiimit tuottaa puolessa vuodessa tuottaa 15 x 25 x 13 = 4875 tarinaa; tuollaisen kokonaisuuden hallinta on hyvin haastavaa.



Kuva 14. Notkeaa organisaatiota pyöritetään hierarkisten työlistojen avulla.

Leffingwell [2009] esittää, että ketterät tiimit tarvitsevat taustakseen kuvassa 14 esitetyn mallin mukaisen notkean yritysorganisaation. Yrityksen johdon tasolla pitäisi keskittyä seuraamaan tuotteiden tai järjestelmien ominaisuuksien kehitystä, ja ylläpitää niistä systeemitason backlog-listaa. Yksittäinen toiminto voidaan sitten edelleen palastella tarinoiksi tai tehtäviksi, joita tiimit hallinnoivat omien backlog-listojensa kautta. Luonnollisesti backlog-listoilta edellytetään tällöin tietynlaista kypsyyttä; Leffingwell neuvookin tiimejä varaamaan joka iteraatiosta jonkin verran aikaa myös täysin uusien, systeemitason toimintojen tutkimiseen ja arvioimiseen. Näin systeemitasolla toimiva, julkaisuja suunnitteleva julkaisunhallintatiimi saa tarvitsemaan palautetta, kuten toimintojen työmääräarvioita, tuotteiden julkaisuaikataulun suunnitteluun, ja systeemi- ja tiimitason backlog-listat voidaan pitää synkronissa. Systeemitiimiä taas tarvitaan testaamaan tuotteita toimintojen tasolla ja pitämään yllä hyväksyntätestejä.

Leffingwellin [2009] mukaan malli on notkea, koska tiimitasolla käsitellään edelleenkin vain käyttäjätarinoita, vaikka yrityksen johto perustaakin pitemmän tähtäimen suunnittelun tarinoista koostuviin toimintoihin. Tiimi voidaan toki järjestää toimintojen sijaan esimerkiksi tiettyjen tuotteiden, järjestelmien tai kohdealustojen mukaan. Työn organisointi on yksinkertaista – vain työlistoja ja lappuja – ja kaikki toiminnot ja käyttäjätarinat analysoidaan vasta siinä vaiheessa, kun niitä aletaan implementoida, jolloin vältytään vaatimusmäärittelyjen varastoinnilta.

On kuitenkin kysyttävä, missä määrin Leffingwellin [2009] esittämä kompleksinen organisaatiorakenne enää eroaa perinteisestä ohjelmistotalon organisoinnista? On

varmaan jokseenkin tavallista, että yleisen tason vaatimuksia tutkii ensin joukko ylemmän tason kokeneita insinöörejä ja arkkitehtejä, jotka jakavat ne edelleen pienempiin toimintoihin, ja että tarkemmat, komponentti- tai toimintokohtaiset analyysit tehdään tiimeissä, joten siinä mielessä rakenne ei tunnu mitenkään mullistavalta. Onko ainoa saavutus tässä mallissa se, että nk. keskijohto on jätetty pois tai nimetty uudelleen Product Ownereiksi ja Scrum Mastereiksi? Sinänsä mallissa on arvokasta se, miten keskijohto pidetään kiinteästi kontaktissa varsinaiseen toteutustyöhön, ja toisaalta se, miten työmääräarviot ja tekniset päätökset annetaan organisaation ammattilaisten – eli käytännön kehitystyötä tekevien insinöörien – ratkottavaksi. Fowler [2000] huomauttaa, että toisin kuin useimmilla muilla tekniikan aloilla, ohjelmistotuotannossa teknologia kehittyy hyvin nopeasti, ja jo muutaman keskijohdossa vietetyn vuoden jälkeen insinöörin tekninen kompetenssi voi olla täysin vanhentunutta. Tässä tilanteessa on tärkeää, että kaikki teknisiä arvioita tekevät työntekijät pysyvät kosketuksissa arkiseen kehitystyöhön.

Outoa Leffingwellin [2009] notkeassa organisaatiomallissa ovat ns. julkaisujen suunnittelupalaverit, joissa suunnitellaan useiden tulevien iteraatioiden yhteisiä sisältöjä. Koko konsepti eivät tunnu kovinkaan ketterältä, sillä eikö tarkoituksena ollut keskittää kaikki voimavarat siihen, että juuri nyt akuutein toiminnallisuus saataisiin toimintakuntoon, eikä suunnitella koko seuraavan puolen vuoden ohjelmaa? Erikoista on myös se, että tuotteiden toteuttamisen parissa puurtavien kehittäjien pitäisi alkaa käyttää aikaansa myös sellaisten tulevaisuuden systeemitason toimintojen tutkimiseen, joita ei juuri nyt tarvita. Tuntuu optimistiselta kuvitella, ettei tällainen malli olisi omiaan luomaan häiriöitä tiimien omalle kehitystyölle ja aiheuttaisi juuri sitä ”melua”, jota ketterien menetelmien tulisi vähentää. Millä ajalla tiimit edes analysoivat ja arvioivat tulevia työtehtäviään, ilman että se haittaa heidän keskittymistään tämänhetkisiin projekteihin? Laajojen analyysien tekemiseenhän voi mennä päiviä tuntien sijaan. Jos taas arkkitehtuuri ja teknologiat on valittu etukäteen, ja kyse on vain vaatimuksien yksityiskohdista, voi kysyä, mitä tapahtui sille periaatteelle, jonka mukaan ketterä tiimi on itse paras asiantuntija ratkaisemaan, miten asiakkaan toivoma toiminnallisuus toteutetaan?

Ketterien menetelmien uskotaan johtavan vastuullisiin ja itseohjautuviin tiimeihin, jotka kykenevät luovasti ratkomaan asiakkaansa tarpeita. Esitetyssä mallissa tarpeet ja prioriteetit eivät kuitenkaan tule enää asiakkaalta. Sen sijaan julkaisuja suunnitteleva johtoryhmä koordinoi, mitä toiminnallisuutta eri tiimien tulisi saada aikaiseksi. Luovuus ja innovatiivisuus on Leffinwellin [2009] mallissa rajattu organisaation ylemmän hierarkiatason etuoikeudeksi, ja tiimien tehtäväksi jää vain antaa töistä aikatauluarvauksensa ja ryhtyä toteuttamaan haluttua toiminnallisuutta. Esitetty toimintamalli on varmaan toimiva rakenne ison ohjelmistotalon toiminnan organisoimiseksi, mutta ketterästä ”ihmiset ennen prosesseja” -hengestä siinä ei enää näy kovinkaan paljon merkkejä.

7. Ketteryyden käyttökelpoisuudesta

Ketterien menetelmien on helppo uskoa toimivan hyvin tilanteissa, joissa tuotteen nopea markkinoille pääsy on keskeinen vaatimus, ja joissa muut vaatimukset eivät ole kiveen hakattuja, vaan projektin jäsenillä on vapaus kokeilla yrityksen ja erehdyksen kautta, millainen tuote parhaiten soveltuu asiakkaan tarpeisiin. Mutta onko ketterillä menetelmillä mitään annettavaa laajoihin ja vaatimuksiltaan ja arkkitehtuuriltaan kompleksisiin ohjelmistoprojekteihin? Ainakin julkishallinnon puolella tilausta muutokselle olisi, sillä monet epäonnistuneet massiiviset ohjelmistoprojektit paitsi syövät verorahoja myös jopa vaarantavat kansalaisten terveyden ja hyvinvoinnin.

7.1. Laajoja hankkeita, monimuotoisia ongelmia

Suomessa käytetään noin kymmentä erilaista, enemmän tai vähemmän moitittua potilastietojärjestelmää, joiden kehittämiseen ja harmonisoimiseen kunnat ovat käyttäneet yli 15 miljoonaa euroa vuosina 2003–2007. Sosiaali- ja terveysministeriön erityisasiantuntija Itkonen [2007] harmittelee sitä, etteivät suomalaisten potilastietojärjestelmien kehittämishankkeet ole moniltakaan osin lunastaneet niihin kohdistuneita odotuksia. Järjestelmiä tutkinut Nykänen [2007] arvioi, että Suomessa käytettävät potilastietojärjestelmät ovat ajastaan jäljessä, eivätkä ne vastaa nykykäyttäjien tarpeita. Potilastietojärjestelmien arkkitehtuuri ja organisointi ovat monimutkaisia, ja järjestelmät perustuvat osittain vanhentuneisiin ratkaisuihin. Vasteajat ovat hitaat, kun ohjelmistoja käyttää suuri joukko ihmisiä samaan aikaan. Ohjelmistojen käytettävyys heikko, jolloin tietojen syöttämisessä on helppo tehdä virheitä. Nykänen ymmärtää hyvin lääkäreiden ja sairaanhoitajien turhautumisen, sillä terveydenhuollon työntekijät joutuvat käyttämään ison osan ajastaan syöttämällä tietoja järjestelmiin yhä uudestaan yrityksen ja erehdyksen kautta, mutta järjestelmät antavat ulospäin hyvin vähän hyötyä käyttäjilleen.

Itkonen [2007] toteaa, että monet terveydenhuollon ohjelmistohankkeet ovat pahasti myöhässä aikataulustaan ja että erilaisten järjestelmien harmonisointi tulee jatkumaan vielä vuosia. Hän kritisoi, että ohjelmistotoimittajien toimintaa on ohjannut lähinnä tavoite rahastaa kaikki tekemänsä toimenpiteet moneen kertaan. Nykäsen [2007] mukaan osasyy ongelmiin on järjestelmien tilaajissa, sillä kuntien ja sairaanhoitopiirien tulisi vaatia ohjelmistotoimittajilta nykyistä enemmän. Nykänen suosittaakin tarkkoja ja yksityiskohtaisesti määriteltyjä sopimuksia, joihin kirjataan mitä tehdään, työn selvät määräajat sekä sanktiot niiden ylittämisestä ja siitä, jos ohjelmistotoimittaja sooloilee liikaa omiaan.

Ilmeisesti Nykäsen neuvot tulivat liian myöhään, sillä vuonna 2007 aloitettu Sosiaali- ja terveysministeriön ja Kelan vetämä suuri kansallinen potilastietojärjestelmähanke on jo ajautunut alalle perin tyypillisiin vaikeuksiin. Hankkeessa oli tarkoituksena yhdistää kymmenet erilaiset kuntien käytössä olevat potilas-, resepti- ja hoitotietojärjes-

telmät yhdeksi kokonaisuudeksi vuoteen 2011 mennessä. Kallio [2009] kuitenkin kertoo, että jo nyt on selvää, ettei hanke tule pysymään suunnitellussa aikataulussa, eikä kukaan osaa arvata todellista valmistumispäivämäärää. Mitään valmista projektissa ei ole vielä ehditty saada valmiiksi, mutta projektin budjetti on paisunut reippaasti. Alun perin hankkeeseen oli varattu 20 miljoonaa euroa, mutta nykyinen arvio on, että 23 miljoona ei riitä kattamaan kustannuksia kuin vuoden 2010 loppuu saakka, ja sen päälle tulevat vielä jatkokehitys- ja käyttöönottokulut kaikissa eri kunnissa. Luvut tuntuvat kovin suurilta, kun ottaa huomioon, että Kela itse arvioi, että järjestelmän yksi osa, eResepti, voisi 10 vuoden kuluessa säästää kuluja ehkä vain noin 10 miljoonan euron verran. Nähtäväksi jää, pystyykö yhdistämishanke mitenkään pureutumaan eri järjestelmien erilaisiin ongelmiin, vai onko tuloksena vain huonosti toimivien, epäkäytettävien ohjelmistojen jo toimitettaessa vanhentunut verkosto.

Valtiontalouden tarkastusviraston johtava toiminnantarkastaja Voutilainen [2008] on huolissaan siitä, miten tuottavuusohjelmien ohjaamat valtio ja kunnat päätyvät täysin riippuvaisiksi ohjelmistojensa toimittajista vähentäessään IT-ammattilaisten määrää palkkalistoiltaan. Voutilainen kuvaa, että kun ohjelmistohankkeeseen ryhdytään, ei julkisessa hallinnossa välttämättä tiedetä, mitä ollaan lähdössä tekemään eikä osata vaatia toimittajalta riittävästi. Tällöin tarjouspyyntöjä varten tarvittavia vaatimusmäärittelyjä tekemään täytyy ehkä palkata jokin ulkopuolinen konsultti tai IT-firma. Samainen määrittelyn tekijä voi silloin räätälöidä vaatimukset sellaisiksi, että hänen edustamansa yritys on edullisessa asemassa muihin kilpailijoihin nähden. Lisäksi, jos vaatimusmäärittelyssä on puutteita, lisätyöt tehdään tuntiperusteista laskutusta käyttäen, mikä johtaa paisuviin budjetteihin ja venyviin aikatauluihin. Avoimia ohjelmistorajapintoja ei myöskään osata vaatia tai valvoa, mikä johtaa irrallisiin, keskenään toimimattomiin osaratkaisuihin, joiden yhdistämisestä järjestelmät toimittaneet ohjelmistotalot voivat jälleen lisätöinä laskuttaa.

Vaikka suurimmat suomalaiset ohjelmistotalot edelleenkin hyödyntävät pääosin perinteisiä ohjelmistokehitysmenetelmiä, joihin kuuluu keskeisenä osana perusteellisten vaatimusmäärittelyjen laatiminen ennen toteutukseen ryhtymistä, näyttää keskeneräisten ja toiminnoiltaan puutteellisten ja epätyydyttävien ohjelmistojen toimittaminen olevan vähintäänkin maan tapa. Annetut aikataulut venyvät ilman, että mitään valmista toiminnallisuutta olisi saatu aikaiseksi, jolloin asiakkaat joutuvat yhä uudelleen sitoutumaan maksumiehiksi saadakseen edes jotain vastinetta jo uhraamilleen rahoille. Kun ohjelmistot sitten otetaan käyttöön, niistä löytyneitä virheitä ei enää ole varaa tai halua korjata, jolloin ohjelmiston pilottikäyttövaihe jää täysin hyödyttömäksi.

Ongelmallista on sekin, että vastaavissa hankkeissa ohjelmistojen toimintakonseptien suunnitteleminen jää puolitiehen. Tietojärjestelmät on suunniteltu usein lähinnä korvaamaan paperista arkistoa, mikä näkyy siinä, ettei sähköisen muodon tuomia kaikkia mahdollisuuksia osata tulla ajatelleeksi ennen kuin järjestelmä on jo valmis ja käy-

tössä – esim. potilastietojärjestelmistä olisi kätevää voida noutaa nopeasti kaikki ne potilaat, joilla on tiettyjä oireita ja tietty lääkitys, yhdeksi tutkimusryhmäksi, mikä ei paperisten arkistojen kohdalla olisi ollut mitenkään mahdollista. Projektien kannalta tuotekehitysideat tulevat siis mieleen aivan liian myöhään. Näyttää lisäksi siltä, että niissä tilanteissa, joissa asiakkaalta puuttuu teknistä tai liiketoiminnallista kompetenssia määritellä vaatimuksiaan, konsultointiyrityksetkään eivät aina kykene tarjoamaan puolueetonta ja luotettavaa apua.

7.2. Toiveita ja mahdollisuuksia

Julkishankkeiden ongelmia ei ehkä kokonaan voi sälyttää osaamattomien asiakkaiden tai ahneiden ohjelmistotalojen syyksi, vaan kenties kyse on siitä, että kattavien, yksityiskohtaisia vaatimuksia sisältävien sopimusten laatiminen on laajojen projektien kohdalla vain yksinkertaisesti liian vaikeaa? Siinä tapauksessa julkishallinnon ohjelmistohankkeissa havaittuja ongelmia voitaisiin ehkä lievittää ketteriä kehitysmalleja seuraamalla. Huolet siitä, että ketterät menetelmät erityisesti altistaisivat asiakkaan keskeneräisille toteutuksille (vert. ”Hiiri kissan räätälinä” -satu) tai toimittajan harrastamalle manipuloinnille (vert. ryhmäajattelu) tuntuvat jokseenkin irrelevanteilta edellä esitettyä ongelmataustaa vasten.

Jos ohjelmistoja kehitettäisiin yhteistyössä asiakkaan kanssa, vaatimusmäärittelyn mahdolliset virheet tai puutteet voitaisiin huomata varhaisemmassa vaiheessa ja asiakkaalla olisi parempi kontrolli ohjata työskentelyä oikeaan suuntaan. Jos järjestelmät julkaistaisiin erillisinä, pieninä toiminnallisina osajulkaisuina, tulisi paremmin taatuksi se, että asiakas saa ainakin jotain vastinetta rahoilleen, ja loppukäyttäjien kokemukset voitaisiin paremmin ottaa huomioon. Asiakkaan näkökulmasta voi olla parempi, että toimittaja toimittaa sovitun aikataulun puitteissa jotain kuin ei mitään, ja että yksittäisen toiminnallisuuden jääminen uupumaan on pienempi haitta kuin se, että hanke ei etene lainkaan ja alkaa siten mahdollisesti estää kaikkien muiden osa-alueiden kehitystä. Toiminnalliset osajulkaisut myös mahdollistaisivat eri osien toimittamisen kilpailuttamisen erikseen sekä epätyydyttävän toimittajan vaihtamisen kesken hankkeen.

Ketterissä menetelmissä asiakkaalla on päävastuu vaatimusten määrittelystä, ja tätä voi pitää jonkinasteisena haasteena hankkeissa, joissa vaatimuksia on paljon ja joissa asiakkaalta puuttuu teknistä osaamista. Tilanne ei kuitenkaan ole juuri huonompi kuin perinteisiä menetelmiä käyttävissä projekteissa, joissa asiakas kertoo, mitä haluaa, ja joissa yksi ryhmä insinöörejä kehittää sen pohjalta formaalin vaatimusdokumentaation, jota taas toinen ryhmä insinöörejä tulkitsee kehittäessään määriteltyä systeemiä. Vastuu siitä, että tietää, mitä haluaa ja tarvitsee, on aina ollut projekteissa asiakkaalla, mutta ketterässä projektissa asiakkaalla on aitiopaikka havaita, miten hänen vaatimuksensa ymmärretään, ja hänellä on mahdollisuus puuttua aktiivisesti näihin tulkintoihin. Tekniseksi tuekseen asiakas voi edelleenkin palkata konsultin, ja projektia voi olla ohjaamassa kokonainen *johtoryhmä* (steering group) yksittäisen ihmisen sijaan.

Kun yksityiskohtaista vaatimusmäärittelyä ei yritetäkään tehdä etukäteen tarjouskisaa varten vaan vaatimukset rakentuvat osana iteratiivista kehitysprosessia, on asiakkaan mahdollisesti helpompi myös varmistua siitä, että hänen tekniseksi avukseen palkkaamansa ohjelmistokonsultti edustaa eri intressejä kuin tuotteen toteuttava ohjelmistotalo ja on siten puolueeton.

Vaatimusten tarkentumisen ohella vaatimusten muuttumiseen liittyy ketterissä menetelmissä mielenkiintoisia ilmiöitä. Caon ja Rameshin [2008] tekemässä haastatteluihin perustuvassa tutkimuksessa projektit kuvasivat, että tyypillisesti vaatimukset muuttuivat iteraatioiden välillä siten, että joitakin listattuja toimintoja päätettiin jättää toteuttamatta, ja joitakin uusia vaatimuksia lisättiin listalle. Joskus asiakas saattoi pyytää jo toteutettuihin toimintoihin pieniä muutoksia. Mitään suurta vaatimusten muuttuvuutta tai vaihtelevuutta ei kuitenkaan havaittu yhdessäkään projektissa. Vaikuttaakin siltä kuin ketterät kehitysmenetelmät, jotka yrittävät valmistaa projektia kestämään muutosten iskut, olisivatkin ketterien käytäntöjensä, kuten asiakasyhteistyön ja jatkuvan kommunikoinnin, takia omiaan *ehkäisemään* vaatimusten muuttumista.

Ketterässä projektissa vaatimuksia käsitellään aina tärkeysjärjestyksessä, ja työn alle otettuja tarinoita työstetään kunnes asiakas on tyytyväinen tuloksiin. Voi olla, että vaatimusten muutosten aiheuttamat negatiiviset vaikutukset jäävät tämän takia mahdollisimman pieneksi, ja että juuri siksi ketterien menetelmien voi väittää ”syleilevän muutosta”. Jos työstettävä vaatimus on listalla kaikkein ensimmäisenä, on selvää, ettei projektissa ole tulossa mitään sen tärkeämpää työn alle, ja silloin ko. vaatimuksen muutoksien vaikutusta tuleviin työtehtäviin ei tarvitse surra. Projektin myöhemmässä vaiheessa toteutettavan vaatimuksen muuttuminen suuntaan tai toiseen taas ei ole niin merkityksellistä, koska palapelin tärkeimmät palaset on jo saatu siinä vaiheessa paikalleen. Mitä pitemmällä projektissa ollaan, sitä rennommin jäljellä oleviin vaatimuksiin voidaan myös suhtautua, ja toisaalta myös muutoksien todennäköisyys vähenee projektin loppu kohden luonnostaan.

Yksi keskeisiä ketterien menetelmien suosion syistä liittyy epäilemättä siihen, miten eettisesti miellyttävät puitteet kehys tarjoaa. Beck [2007] uskoo, että maailmassa on suuri tilaus raa’an rehelliselle ohjelmistotuotannolle, jossa asiakas voi itse nähdä, miten kehittäjät ponnistelevat toiminnallisuuksien toteuttamiseksi, ja jossa kehittäjät voivat kerrankin luvata asiakkaalle vain juuri sen verran toiminnallisuutta kuin minkä tietävät pystyvänsä toteuttamaan. Ketterät menetelmät rakentuvat luottamukselle ja uskolle siihen, että insinöörit haluavat tehdä parhaansa, ovat motivoituneita kuulemaan asiakasta ja haluavat itse organisoida toimintansa niin, että asiakas saa mitä on tilannut. Koska kehitystyö on läpinäkyvää, oman edun tavoittelu tulee selvästi näkyväksi, mikä on omiaan rajoittamaan ohjelmistokehittäjiin kohdistuvaa painetta toimia vain oman firmansa eduksi. Tällaisessa ympäristössä ohjelmistokehittäjien on helpompi toimia alansa eettisten periaatteiden mukaisesti, eli kuten ACM [2010] on ne esittänyt: pyrkiä varmista-

maan niin yleisön kuin oman asiakkaansa ja työnantajansa etu ja turvallisuus, varmistua siitä, että toimitetut ohjelmistot ovat parasta mahdollista laatua, pitää kiinni ammatillisesta lahjomattomuudestaan ja riippumattomuudestaan, nostaa oman alansa ammatillista arvostusta ja edistää eettistä toimintaa.

7.3. Reunaehtoja ja rajoituksia

Jalot eettiset pyrkimykset eivät yksin riitä takaamaan ketterien projektien molemminpuolista onnistumista, vaan keskeiseen asemaan nousee myös laaditun sopimuksen tyyppi, sillä se väistämättä ohjaa osapuolia joko välinpitämättömään, puolustautuvaan tai yhteistyötä lisäävään toimintaan. Kirjallinen, tavoitteet ja odotetut tuotokset määrittelevä sopimus on ilman muuta ehdottoman tärkeä, jotta projekti laajuus saadaan rajattua, ja jotta sen valmistumista ja onnistuneisuutta voitaisiin myöhemmin arvioida. Sopimukseen tulisi myös kirjata, voidaanko tavoitteita muuttaa projektin aikana, ja jos voidaan, niin millä ehdoin ja kuinka paljon. Joissakin projekteissa sopimukseen vetoaminen voi olla ainoa keino saada työ päätökseen, jos asiakas on halukas aina vain kehittämään tuotetta eteenpäin.

Fowler [2000] korostaa, että kiinteähintainen sopimus ei voi mitenkään soveltua ketterään kehitykseen, ja huomauttaa, että asiakkaan pitäisi ymmärtää, että kiinteästi hinnoitellun projektin epäonnistuessa molemmat osapuolet kärsivät yhtä lailla. Fowler järkeilee, että oikeastaan asiakkaan menetys on vielä suurempi, vaikkei hän edes maksaisi tuotteesta, sillä eihän asiakas olisi ryhtynyt koko hankkeeseen, ellei hänellä olisi tarvetta tuotteelle ja ellei hän olisi arvioinut, että tuotteen liiketaloudellinen arvo on suurempi kuin sitä varten budjetoidut kehityskulut. Fowler muistuttaa, että perinteiset suunnitelmapohjaiset projektit katsotaan onnistuneiksi silloin, kun tuotettu ohjelmisto vastaa suunnitelmaa, mutta ketterillä menetelmillä on mahdollista saavuttaa jotain paljon alkuperäisiä kaavailuja parempaa. Toisaalta laskutyö-tyyppinenkään sopimus – joita julkishallinto käytännössä näyttää suosivan – ei ole reilu, koska siinä kaikki riskit jäävät asiakkaan kannettavaksi. Beaumontin [2008] esittelemä ”ilmaista rahaa ja vapaasti muutoksia -tyyppinen jaetun riskin mahdollistava sopimus vaikuttaisi sopivalta kompromissilta, ja tilannetta voidaan vielä parantaa käsittelemällä jokaisen iteraation toimituslistaa omana sopimuksenaan.

Harmillista kyllä, globaali taloudellinen laskusuhdanne tekee asiakkaista entistä varovaisempia, jolloin kiinteähintaiset sopimukset saattavat asiakkaasta vaikuttaa kätevältä tavalta kontrolloida investointeja myös ketterien projektien kohdalla. Tällöin voi käydä niin, että vaikka projektin vaatimukset olisivat riittävällä tasolla ketterän projektin aloittamista ajatellen, tarvittavan tarkkoja tietoja varsinaisen sopimuksen laatimiseksi ei ole. Jos ohjelmiston toimittaja ei ole halukas yksin kantamaan epämääräisen projektinmäärittelyn tuottamia riskejä – kustannusarviota kun on vaikea antaa ilman tarkkaa tietoa siitä, mitä täytyy saada aikaiseksi – hankkeen aloitus lykkääntyy. Usein toimittajat eivät myöskään ole innokkaita tekemään sopimuksen laatimiseksi tarvittavaa

taustatutkimusta omalla kustannuksellaan, koska he katsovat, että ko. selvitystyö olisi olennainen osa koko empiiristä ohjelmistokehitysprojektia muutenkin, ja siten osa laskutettavia töitä. Ketterien menetelmien tulevaisuus on pitkälti kiinni sopimuskäytäntöjen kehittymisestä, sillä ilman todellista kumppanuutta yhteistyö ei voi toimia toivotulla tavalla.

Siinä missä ketterät menetelmät tuntuvat ”luonnolliselta” tavalta järjestää työt yksittäisen ryhmän kesken, ongelmana on näiden periaatteiden hyödyntäminen koko organisaation tasolla, eli ns. skaalautuvuus. Skaalautuvuuden ongelma on kuitenkin ratkaistava, jotta ketteriä menetelmiä voitaisiin menestyksekkäästi hyödyntää laajoissa projekteissa. Kohdassa 6.9 on esitelty eri asiantuntijoiden ehdotuksia skaalautuvuuden parantamiseksi, mutta näissä ratkaisuvaihtoehdoissa on se periaatteellinen ongelma, että niillä yritetään sovittaa yhteen ristiriitaisia lähtökohtia, eivätkä ne siksi voi johtaa optimaalisiin ratkaisuihin. Ketterissä menetelmissä on keskeistä nimenomaan vaatimusten ja niiden käsittelyn ”oikea-aikaisuus”, ja tästä on luonnollisena seurauksena se, että näkyvyys tulevaisuuteen – tuleviin tuoteperheisiin, epiikoihin, roadmappeihin tms. – jää väkisinkin heikoksi. Tämän epävarmuuden sietäminen olisi juuri sitä paljonpuhuttua rohkeutta, jonka varaan kaikki empiirinen toiminta rakentuu.

Toisin kuin Leffingwell [2009] ehdottaa, työskenteleviä projektitiimejä ei pitäisi häiritä sillä, että ne laitetaan laatimaan analyysejä toiminnoista, jotka eivät ole vielä tulossa tiimin seuraavan iteraation työlistalle. Tulevien vaatimusten arvioinnin ja aikatauluttamisen ei pitäisi olla merkityksellistä, sillä jos tiimit jo työskentelevät niin tehokkaasti kuin ne voivat tärkeimpien asioiden eteen, niin miten aikatauluttamisella voitaisiin enää parantaa tilannetta mitenkään? Organisaatiot, joissa työtä tehdään ns. etukenossa tulevaisuutta varten vaatimuspuita rakennellen tai tulevien julkaisujen sisältöjä luonnostellen – olivatpa ne määrittelyt sitten dokumenteissa, seinällä liimalappusina tai ihmisten korvien välissä – eivät perimmäiseltä olemukseltaan ole ketteriä eivätkä adaptiivisia, vaan aivan perinteisiä tuotekehitysorganisaatioita. Sinänsä perinteisyydessä ei ole mitään pahaa, vaan kyse on siitä, miten tärkeää nimenomaisesti joustavuus ja nopea reagointikyky ovat koko organisaatiolle. Luonnollisestikin omaa tuoteperhettään kehittävä iso ohjelmistotalo voi olla toiminnassaan paljon strukturoidumpi ja kankeampi kuin pieni alihankintatoimisto, jonka pitää olla heti valmis loikkaamaan kiinni asiakkaalta saatuun tehtävään, ja jonka pitää pystyä hyödyntämään kulloinkin vapaana olevia resurssejaan tehokkaasti voidakseen taata asiakkaalleen nopean ja onnistuneen projektin läpiviennin.

Schwaber [2008b] vaikuttaisi olevan oikeilla jäljillä uskoessaan, että Scrum-tiimien synkronointi on täysin riittävä tapa saada ketterät menetelmät skaalautumaan, sillä vain näin voidaan varmistua siitä, että kaikki tehty työ on tarpeellista kaikkein tärkeimpien tämänhetkisten tavoitteiden kannalta. Samalla myös päätöksenteko siitä, mitä ja miten toteutetaan, säilyy parhaimmilla asiantuntijoilla, eli ketterillä tiimeillä itsel-

lään. Ketterien menetelmien skaalautumisen mahdollistamisessa on olennaisempaa organisaation kyky joustaa ja sallia sen yksiköiden dynaamisesti järjestäytyä erilaisten tavoitteiden ympärille kuin mikään organisatorisen kaikenkattavan tuotekehitysprosessin määrittely. Ketterissä menetelmissä pyritään parantamaan ja ylläpitämään ennustettavuutta pilkkomalla tavoitteet aina vain pienempiin osiin, eikä tarkoituksena tällöin suinkaan ole, että kokonaisuutta hallittaisiin osiensa kautta, vaan että kokonaisuus rakentuisi osistaan. Vaatimukset eivät siis muodosta puuta, jota voisi analysoida lehti lehdeltä, vaan pikemminkin pienistä puroista kasvaa iso virta. Ehkä ketterien menetelmien skaalautuvuudella on näin ollen ns. luonnolliset rajansa, joiden ylittäminen ei ole mahdollista infrastruktuurisin keinoin menettämättä samalla jotain olennaista ketteryyden tuomista eduista.

8. Ketteryydestä, vaatimuksista ja niiden suhteesta

Tämän tutkielman tarkoituksena oli selvittää, miten ketterien ohjelmistokehitysmenetelmien seuraaminen vaikuttaa vaatimuksien määrittelyyn, mutta analyysiin edetessä on tullut ilmeiseksi, että kyse on itse asiassa päinvastaisesta vaikutussuhteesta, eli siitä, miten määritellyt vaatimukset vaikuttavat tuotekehitysprosessiin. Vaikuttaa selvältä, että tilanteissa, joissa projektin vaatimusmäärittelyt ovat luonteeltaan epämääräisiä ja kehitystyö pikemminkin empiiristä tutkimustyötä kuin koneiston kasaamista, ketterät menetelmät tarjoavat oivat puitteet työstää vaatimusmäärittelyjä. Leffingwell ja Behrens [2009] lisäksi korostavat, että vain neuvoteltavat, joustavat vaatimukset, kuten käyttäjätarinat, tuovat suunnitelmiin ennustettavuutta. Jos vaatimukset ovat kovin rajoittavia ja yksityiskohtaisia, tiimi ei voi neuvotella eri lähestymistapojen hyödyistä ja haitoista, kun taas avoimemmiksi jätetyt vaatimukset antavat tiimille mahdollisuuden ratkoa asiakkaan liiketoiminnallisia haasteita yhdessä.

Ketterien menetelmien suhde vaatimuksiin on hyvin käytännöllinen: kehittäjien tulisi pelata niillä korteilla joita heillä on – ts. käydä käsiksi työhön hyödyntäen niitä tietoja ja resursseja joita heillä on saatavilla – eikä tuhlata aikaa istumassa ja toivomassa parempaa huomista. Usein laaja-alaisin ja syvällisinkään analysointi ei voi paljastaa vaatimuksen todellista merkitystä ja ominaisluonnetta siinä määrin kuin vaatimuksen käytännön toteutuksen arviointi. Mielenkiintoista kyllä, ketterät menetelmät soveltuvat näin ollen kahden hyvin erityyppisen kehityshankkeen läpivientiin: toisaalta projekteihin, joissa on tavoitteena tuottaa jonkinlainen tuote asiakkaan tarpeisiin *nopeasti*, ja toisaalta projekteihin, jotka ovat tutkimuksellisia eli aidosti *empiirisiä*.

Esimerkki edellä mainitusta hanketyypistä voisi olla projekti, jossa asiakkaalle pitää kehittää nopeasti tuote, jossa kopioidaan tai yhdistellään jo markkinoilla olevien kilpailijoiden tuotteiden ominaisuuksia. Tällaisessa projektissa on keskeistä vaatimusten nopea ”päättely”, esim. vaikka kilpailijoiden tuotteita vertailemalla, ja lyhyet iteraatiot, joissa aina ensimmäinen riittävän toimiva toteutus heti hyväksytään. Jokaisen iteraation lopuksi tuotteen markkinakelpoisuus sellaisenaan arvioidaan, ja heti tyydyttävän tason saavutettuaan tuote lähtee kauppoihin. Esimerkki jälkimmäisestä projektityypistä taas voisi olla jokin toteuttamiskelpoisuustutkimus, jossa demonstroidaan, voitaisiinko jokin uusi teknologia tai standardi toteuttaa tai integroida tietyssä ympäristössä, ja pyritään samalla selvittämään, mitä haittoja ja etuja eri lähestymistapoihin sisältyy. Tällaisessa projektissa keskeistä on erilaisten toteutusvaihtoehtojen kokeileminen, toki toiminnallisuuteen pyrkien, ja työn tuloksena on tarkentunut vaatimusmäärittely ja mahdollisesti myös jonkinasteisesti toimiva prototyyppi.

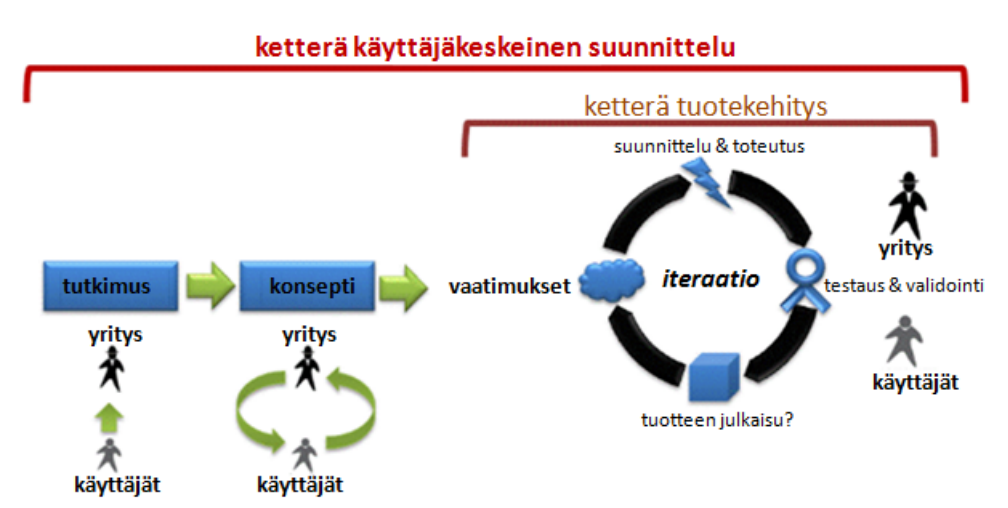
Ketterät menetelmät perustuvat siihen, että kehittäjillä on jonkinasteinen vapaus leikkiä ja kokeilla etsiessään toimivaa ratkaisua, mutta toisaalta ketterät menetelmät lupaavat myös nopeasti käyttövalmista toiminnallisuutta. On syytä huomata, että esitet-

ty lähtökohta on jossain määrin ristiriidassa odotetun lopputuloksen kanssa, ja että riippuu ensisijaisesti projektin tavoitteista – ts. vaatimuksista – miten projektin lopputulos voi onnistuessaankaan vastata asiakkaan odotuksia. Ensimmäinen toimiva toteutus ei tarkoita optimaalista laatua tai innovatiivisinta ratkaisua, ja toisaalta, omaperäisin tai toimivin toteutusvaihtoehto ei välttämättä löydy nopeasti. Ketterä projekti voi olla vain niin hyvä kuin sille asetetut vaatimukset sallivat, ja realistisesti ajatellen, projekti voi kenties toivoa saavuttaa vain jomman kumman tavoitteen: innovatiivisen tai nopean ratkaisun.

Edellä mainittu rajoitus olisi hyvä muistaa, sillä keskeneräisen tuotekonseptin tarjoileminen loppukäyttäjille voi nopeiden voittojen tuomisen sijaan tahrata yrityksen maineen ja uskottavuuden käyttäjien silmissä. Colfelt [2010] varoittaa, että koska vaatimuksien keräämisen prosessia ei ole määritelty ketterissä menetelmissä, voi käydä niin, että asiakas luulee, että hän voi ja että hänen täytyy osata edustaa kaikkien tulevien käyttäjien tarpeita. Confelt muistelee projektia, jossa yletön tehokkuuden ja dynaamisuuden korostaminen ja ketterien periaatteiden uskonnollinen seuraaminen johtivat siihen, ettei minkäänlaisille käyttäjätutkimuksille annettu aikaa, vaan kaikki vaatimukset kumpusivat tuotepäällikön omasta mielikuvituksesta. Edelleen suunnittelu- ja toteutustyöt vietiin hosuen läpi, koska kaikki piti saada valmiiksi aina yhden iteraation aikana. Seuraavassa iteraatiossa sitten jouduttiin käyttämään aikaa edellisen iteraation aikana kehkeytyneiden ongelmien korjaamiseen, samalla kun häärättiin kasaan vielä lisää toiminnallisuutta. Voisi luulla olevan selvää, ettei tällaisella prosessilla voida kehittää uutta voittajatuotetta, mutta ilmeisesti ketteriin menetelmiin liitetyt odotukset ovat niin suuria, että ne haittaavat kriittistä ajattelua.

Loppukäyttäjän näkökulma voi helposti unohtua ketteriä menetelmiä käytettäessä, koska loppukäyttäjille ei ole määritelty prosesseissa selvää roolia, toisin kuin esim. asiakkaalle. Ketterät menetelmät eivät kuitenkaan kiellä käyttäjäkeskeistä suunnittelua, niin kuin ne eivät kiellä dokumentoimastakaan – ne vain opastavat ajattelemaan että toimiva tuote on suunnitelmia ja dokumentaatiota tärkeämpi. Colfelt [2010] muistuttaa, ettei ketterässäkään maailmassa voida olettaa, että vaatimukset vain tipahtavat taivaasta asiakkaan tai kehittäjien syliin. Käyttäjäkeskeinen suunnittelu on paljon enemmän kuin vain käyttöliittymän arviointia, ja iteraatioissa tapahtuva vaatimusten työstäminen – niiden tarkentaminen ja tulkitseminen – on jatkumoa sille strategiselle konseptitason tutkimukselle ja suunnittelulle, jonka pitäisi tapahtua *jo ennen* varsinaisen projektin aloittamista. Colfelt huomauttaa, että esimerkiksi useimmat markkinoilla olevat MP3-soittimet täyttävät suunnilleen samat toiminnalliset vaatimukset ja ovat kohtuullisen käytettäviäkin, mutta vain Applen iPod onnistui saavuttamaan suuren yleisön suosion. Osasyynä tähän on Colfeltin mukaan se, että iPodin taustavoimana on kokonainen iTunes-musiikkipalvelukonsepti.

Colfelt [2010] ehdottaa, että Scrumia käytettäessä konseptisuunnittelu voisi tapahtua esimerkiksi nk. iteraatio nro. 0:n aikana, kuten kuvassa 15 on hahmoteltu. Suunnitteluvaiheen aikana ei ole tarkoitus tuottaa koodia, vaan sen sijaan testata ideoita ja laatia yleisen tason suunnitelma siitä, millaista tuotteen ja sen käyttäjän vuorovaikutuksen halutaan olevan. Tällaisen tiekartan pohjalta varsinaista käyttöliittymää ja tuotteen toimintoja voidaan kehittää ja toteuttaa varsinaisten iteraatiokierrosten aikana.



Kuva 15. Käyttäjäkeskeinen suunnitteluvaihe edeltää toteutusvaihetta.

Ketterän vaatimusten käsittelyn ehdottomana etuna voi pitää sitä, miten tiimi voi iteraatioissa tapahtuvan prototypoinnin kautta käytännössä varastaa hieman lisää aikaa vaatimusten analysointiin ja saa samalla asiakkaankin kiinnostumaan vaatimusten määrittelystä. Sen sijaan se, miten huonosti menetelmät tukevat vaatimusten keräämistä ja projektin osakkaiden tunnistamista, on omiaan johtamaan hyvin kapeaan katsontakantaan. Valjastamalla käyttäjäkeskeinen suunnittelu osaksi ketterää ohjelmistokehitysprosessia voidaan päästä kiinni myös loppukäyttäjien tarpeisiin ja mieltymyksiin.

Käyttäjäkeskeiset tekniikat, kuten käyttäjien observointi, toimintapolku-, rooli- ja tavoitekeskeiset analyysit, etnografiset tutkimukset jne. ovat tuttuja menetelmiä jo perinteisen vaatimusten määrittelyn puolelta. Tekniikoiden ansiot vaatimusten keräämisessä ja tutkimisessa on laajalti tunnustettu, ja niiden hyödyntäminen ketterissä menetelmissä tuo ketterää vaatimusten käsittelyä lähemmäs perinteisen vaatimusmäärittelyn puolella vakiintunutta analyyttistä tarkkuutta ja tasoa. Käyttäjäkeskeisellä suunnittelulla voidaan myös tarvittaessa osaltaan mitätöidä niitä ongelmia, joita osallistumiskyvytön tai -haluton asiakas, jolta ei saada riittävästi palautetta ja ohjeistusta vaatimusten tunnistamiseksi ja analysoimiseksi, voi ketterälle projektille aiheuttaa. Silloin kun projektilta toivotaan erityisesti kustannustehokkuutta, vaatimusten priorisointi on ensiarvoisen tärkeää, jotta osa vähemmän tärkeistä vaatimuksista voidaan jättää toteuttamatta. Käyttötapaus- ja käytettävyystutkimuksista saadun tiedon pohjalta asiakas ja kehitystiimi

voivat valistuneemmin priorisoida vaatimuslistaa, mikä myös osaltaan vähentää asiakkaaseen ja kehittäjiin kohdistuvia päätöksentekopaineita.

Päätöksenteko ketterässä kehitystiimissä on kollektiivista, mikä tuo oman dynamiikkansa koko ohjelmistokehitysprosessiin. Janisin [1982] mukaan ryhmän yhtenäisyys lisää ryhmän jäsenten luottamusta ryhmän päätöksentekokykyyn, ja ryhmä on siksi taipuvainen tekemään rohkeampia päätöksiä kuin mihin sen yksittäiset jäsenet olisivat ryhtyneet. Ketterissä tiimeissä mahdollisesti ilmenevää ryhmäajattelua ja sen vaikutuksia vaatimusmäärittelyn prosessiin ei kuitenkaan ole juurikaan tutkittu. Sinänsä vaikuttaa sopivalta, että ketterissä menetelmissä painotetaan ryhmätyöskentelyn suotuisuutta, sillä onhan ”rohkeus” – rohkeus kokeilla uusia lähestymistapoja, rohkeus edetä toteutustyössä epäselvistä vaatimuksista ja tuntemattomista rajoituksista huolimatta, rohkeus jättää huomisen ongelmat huomisen ratkaistavaksi, jne. – keskeinen arvo esim. XP-menetelmässä. Kenties juuri ryhmäajattelusta kumpuava yltiöpäinen optimismi ja vahva sitoutumishalu ovat välttämätön edellytys ketterien menetelmien onnistuneelle toteuttamiselle.

Ryhmäpäätöksentekoa arvioineessa tutkimuksessaan Moon ja muut [2003] tulivat siihen tulokseen, että tiimit, jotka tekevät päätökset yhdessä, tekevät paitsi rohkeampia ja tehokkaampia päätöksiä, myös – toisin kuin perinteisesti on uskottu Janisin [1982] analyysin pohjalta – puolueettomampia ja realistisempia arvioita tilanteesta. Tutkimuksessa verrokkiryhmät, joissa tiimien jäsenet saivat ensin itse muodostaa henkilökohtaisen mielipiteensä ennen ryhmän tapaamista, olivat puolueellisempia ja taipuvaisempia liioittelemaan erityisesti tilanteiden negatiivisia aspekteja ja päätyivät vain hyvin varovaisiin ja käytännössä tehottomiin toimiin. Viisaat päät kannattanee tosiaan lyödä yhteen ja panostaa aitoon ryhmäpäätöksentekoon, kuten ketterissä menetelmissä opastetaan.

9. Päin ketteryyttä!

Standish Groupin [2009] julkistamassa raportissa tiedotettiin, että vain 32% kaikista tutkituista projekteista päättyi onnistuneesti. Lähes puolet (44%) projekteista oli vaikeuksissa, mikä tarkoittaa, että ne joko valmistuivat myöhemmin kuin oli suunniteltu, ylittivät budjettinsa tai tärkeitä vaatimuksia jäi toteuttamatta. Neljännes (24%) projekteista epäonnistui, millä tarkoitetaan sitä, että projektit joko keskeytettiin ennen valmistumistaan tai niiden tuotoksia ei koskaan otettu käyttöön. Näitä lukuja on luonnehdittu huonoimmiksi vuosikausiin, sillä muutaman viime vuoden ajan onnistumisluvuissa on näkynyt pientä parantumistrendiä. Miten tässä näin kävi? Eikö ketteryydellä päästä parempiin tuloksiin, vai eikö sitä vain käytetä riittävästi ja oikein?

Beck [2007] torjuu ajatuksen siitä, että ketterät menetelmät olisivat jokin ihmelääke epäonnistumisien torjumiseksi. Hän toteaa, että väistämättä osa ohjelmistoprojekteista tulee aina epäonnistumaan, jos ei muusta syystä niin siksi, että on olemassa paljon lupaavia tuoteideoita, jotka eivät vielä vain toimi käytännössä. Beck analysoi, että ketteryydellä voidaan saavuttaa se etu, että ne projektit, jotka joka tapauksessa olisivat tuomittuja epäonnistumaan, päättyvät nopeammin. Näin vapautuu aikaa ja resursseja hyödyllisempiin hankkeisiin.

Schwaber [2008] arvioi, että jopa kolme neljästä Scrum-menetelmää kokeilevasta organisaatiosta ei käytännössä onnistu saavuttumaan toivomiaan hyötyjä. Schwaberin mukaan useat yritysjohtajat tuntuvat yhä kuvittelevan, että Scrum olisi jonkinlainen iteratiivinen vesiputousmalli, mutta vain nopeampi ja keveämpi toteuttaa! Ketterät menetelmät kuitenkin vaativat johtajiltaan entistä tiiviimpää projektin seurantaa ja riskien arviointia, koska mitään perinteisten, suunnitelmallisuuteen nojaavien ohjelmistokehitysmallien mukanaan tuomia turvaverkkoja ei enää ole olemassa. Ongelmana on usein myös se, että organisaatiot eivät yritä korjata niitä puutteita, joita Scrumin esittely organisaatiossa väistämättä paljastaa, vaan mieluummin mukauttavat ketteriä toimintatapojaan saadakseen ne paremmin sopimaan yhteen ongelmallisten käytäntöjensä kanssa; lopputuloksena on rampautettu Scrum-prosessi.

Scrum on toki vain organisointimalli, ja kehyksen sisällä erilaiset organisaatiot voivat hyödyntää niitä ketteriä menetelmiä, kuten vaikka pariohjelmointia, jotka on koettu toimiviksi. Samaan tapaan muutkin ketterät menetelmät ovat vain viitekehyksiä, joista toteuttajan tulee valita omiin tarpeisiinsa sopivat käytännöt. Tämän lisäksi kehittäjien tulee tarpeen mukaan soveltaa kehyksen puitteissa muita hyväksi havaitsemiaan tekniikoita ja työkaluja. Esimerkiksi Schwaber ja Beedle [2002] korostavat, että Scrum-kehitystiimin jäsenillä on oikeus kokeilla ja käyttää hyväkseen mitä hyvänsä menetelmiä ja konsteja saadakseen käyttäjätarinat toteutettua. Koska ketterät periaatteet kuitenkin rohkaisevat pitämään asiat yksinkertaisina, välttämään raskasta etukäteissuunnittelua ja käymään käsiksi tehtävään työhön, ongelmaksi voi muodostua se, että kokemat-

tomat ja erilaisista tekniikoista ja menetelmistä tietämättömät kehittäjät kuvittelevat, että heidän on pakko vain ryhtyä suoraan toteuttamaan ohjelmistoa aivan liian puutteellisten lähtötietojen varassa. Kehittäjät saattavat myös uskoa, että kaikki suunnitteleminen tai dokumentointi sinänsä olisi pahasta ja kiellettyä.

Ketterissä menetelmissä kehittäjien tietämättömyys ja kokemattomuus oman alansa tekniikoiden suhteen voi heijastua erityisen haitallisesti vaatimusten määrittelyyn, koska ketterissä menetelmissä sitä osa-aluetta on yleensä kuvattu hyvin niukasti. Ketterät kehittäjät joutuvat itse miettimään, miten he voisivat mitata prototyyppituotteensa käytettävyyttä, tai miten asiakas voisi arvioida erilaisten liiketoimintatavoitteidensa arvoa ja käyttäjätarinoiden tärkeyttä työlistajärjestyksiä luotaessa. Vaihtoehdoista tietämätön kehitystiimi saattaa tällöin luulla, että asiakkaan kanssa keskusteleminen ja yhdessä arvaileminen ovat ainoat sopivat ja sallitut menetelmät vaatimuksien keräämiseksi. Näin kaikki ne hyödyt, joita vaatimusten määrittelyssä voitaisiin saavuttaa jatkuvalla yhteistoiminnalla asiakkaan kanssa, ulosmitataan tiimin kokemattomuudella ja osaamattomuudella. Prosesseja ohjaavan byrokratian vähäisyys ketterissä projekteissa edellyttää kehittäjiltä kurinalaista ja ammattimaista työskentelyotetta. Kuten kaikkea muutakin ammatillista taitoa, ketterässä maailmassa selviytymiseksi tarvitaan entistä enemmän myös vaatimus-määrittelyosaamista.

Ketterien projektien epäonnistumisia selitellään joskus sillä, että ketteriä periaatteita on toteutettu jotenkin ”väärin” eli epäortodoksisesti, koska projekti epäonnistui; ajatuksena tuntuu olevan, ettei oikeilla menetelmillä *voi* epäonnistua. Kyse voi kuitenkin olla myös siitä, että projekti on valinnut seurattavan ketterän kehyksen väärin tai toteuttanut sitä liiankin kirjaimellisesti ja ”oikeaoppisesti”. Chow ja Cao [2008] suorittivat 109 eri ketterää organisaatiota kattaneen kyselyn, jolla selvitettiin, mitkä asiat ovat keskeisiä ketterien projektien menestymisen kannalta. Tutkijat totesivat, että niin kauan kuin projektilla on käytössään osaava tiimi, joka seuraa ketteriä periaatteita ja tuotettu ohjelmisto julkaistaan iteratiivisesti, projektilla on erinomaiset mahdollisuudet olla menestyksekäs. Tutkimuksessa havaittiin, että varsinaisilla ketterillä tekniikoilla, joita eri menetelmissä joskus pikkutarkasti kuvataan (kuten tarinaseinä tai pariohjelmointi) ei näyttänyt olevan mitään vaikutusta projektin onnistumiseen. Projektit voivat siis huoletta räätälöidä omiin tottumuksiinsa soveltuvat toimintatavat, kunhan seuraavat toiminnassaan ketteriä periaatteita.

Boehm ja Turner [2004] ovat todenneet, että muuttuvien vaatimusten ohella myös henkilöstön kyvykkyys, projektin koko ja työpaikan kulttuuri sekä toimitettavan ohjelmiston kriittisyys vaikuttavat siihen, miten onnistuneeksi valittu ohjelmistokehitysmalli osoittautuu. Reed ja muut [2004] huomauttavat, että myös arkkitehtuurisilla valinnoilla voidaan saavuttaa samoja etuja kuin oikean kehitysprosessin valinnalla. He esittävät, että esimerkiksi *evolutionääriseen* (evolvable) arkkitehtuuriin perustuvat systeemit ovat

vaatimusten muutosten suhteen ihan yhtä joustavia kuin mihin ketterillä menetelmillä voidaan päästä.

Puheet ”oikeasta” tavasta toteuttaa ketteriä menetelmiä ovat harhaanjohtavia ja vahingollisia, koska ne voivat aiheuttaa sen, että siinä vaiheessa kun organisaatio päättää ryhtyä kokeilemaan ketteriä menetelmiä, terve järki lentää ikkunasta ulos. Valittua ketterää menetelmää saatetaan seurata joustamattomasti ja oppikirjamaisesti, ja kaikki kohdatut hankaluudet selitetään vain alkuvaikeuksiksi. Kannustavathan monet ketterät oppaat ensin sitkeästi kokeilemaan ketteriä metodeja ennen niiden tuomitsemista. Kuitenkin mukautuvuus on ketterien kehitysmallien keskeinen piirre eikä yhden koon ohjelmistokehitysmalleja ole olemassakaan. Siirtyessään ketterään kehitykseen organisaation tulisi miettiä, mitä nimenomaisia ongelmia ketteryydellä yritetään ratkaista, ja keskittyä parantamaan toimintaansa juuri niissä suhteissa.

10. Yhteenveto

Vaatimuksien *olemus* perinteisissä ohjelmistokehitysmalleissa on hieman erilainen kuin ketterissä menetelmissä, mutta vaatimusten tärkeys ei ole vähentynyt. Tässä tutkielmassa on tarkasteltu, millaisena ketterät menetelmät näyttäytyvät vaatimusten määrittelyn näkökulmasta. Tavoitteena oli selvittää, mitä mahdollisuuksia ja haasteita ketterä vaatimusten käsittely tuo mukanaan ja millaisin varauksin ketteriä vaatimusten käsittelyn käytäntöjä voi suositella. Tutkitun valossa vaatimusten määrittelyn käytännöistä ketterissä projekteissa on vaikea sanoa kovinkaan paljon mitään ehdottoman varmaa, koska menetelmät antavat vain yleisiä ohjeita ja periaatteita ketterän ohjelmistokehityshankkeen organisoimiseksi. Käytännön toteutukset vaihtelevat, ja lähtötietojen laatu sekä kehitystiimin ammattitaito ja kokemus vaikuttavat valittuihin vaatimuksien käsittelyn tekniikoihin.

Ainoat perinteisen vaatimusten määrittelyn vaiheet, jotka voi suoraan tunnistaa ketteristä menetelmistä, ovat vaatimusten analysointi ja niistä neuvottelu (ks. kohdat 3.2 ja 5.2) sekä vaatimusten validointi (ks. kohdat 3.4 ja 5.4). Kuitenkin myös ketterissä hankkeissa vaatimusten käsittely edellyttää jonkinlaista vaatimusten keräämistä, dokumentointia ja hallinnointia. Siksi ketterissä prosesseissa on useita yhtymäkohtia perinteiseen vaatimusten määrittelyyn, vain hieman eri tekniikoilla toteutettuna. Vaatimukset voidaan esimerkiksi kirjata ylös ensin yksinkertaisina käyttäjätarinoina, ja toteuttamisensa jälkeen vaatimukset jatkavat olemassaoloaan funktionaalisuutta testaavissa testitapauksissa. Tietoisemmalla vaatimusten käsittelyllä ketterien käytäntöjen osumatarkkuutta voidaan entisestään parantaa, ja monet käyttäjäkeskeisestä suunnittelusta tutut tekniikat ovat sulautettavissa osaksi ketteriä menetelmiä.

10.1. Rohkaisun sanoja

Ketterät menetelmät tarjoavat ohjelmistosuunnittelijalle mahdollisuuden tehdä työnsä aiempaa itsenäisemmin, mielekkäämmin ja eettisemmin. Yhteistyö poikkiteknisten haasteiden parissa oman tiimin ja asiakkaan kanssa kehittää sekä halua että kykyä kantaa vastuunsa projektista, ja haasteiden selättäminen laajentaa ammatillista osaamista ja lisää ammattiylpeyttä. Erityisesti asiakkaille ketterillä menetelmillä on paljon annettavaa, sillä läpinäkyvän kehitysprosessin ansiosta asiakkaan on helpompi nähdä ja kontrolloida, miten hänen liiketoiminnalliset tarpeensa tulkitaan ja muunnetaan tuotteeksi. Jatkuva priorisointi taas parantaa mahdollisuuksia saavuttaa ainakin kaikkein tärkeimmät tavoitteet.

Vaatimusten määrittelyn näkökulmasta ketteriä menetelmiä voi pitää jopa uraa uurtavina siltä osin kuin projekteissa ei vain pyritä toteuttamaan vaatimuksia, vaan luomaan niillä *lisäarvoa* asiakkaalle. Kehitysprosessin aikana vaatimuksia työstetään iteratiivisesti ja niiden analysointi, priorisointi ja määrittely tehdään ryhmätyönä hyö-

dyntäen kehittyvää tuotetta sekä vaatimusten validoinnissa että prototyypin tapaan uusien vaatimuksien hahmottelussa. Tässä prosessissa aluksi epämääräisillä vaatimuksilla on mahdollisuus kasvaa työstämisen myötä paremmaksi ja toimivammaksi tuotteeksi kuin mitä asiakas alun perin osasi toivoakaan. Vaatimusten muuttuminen ketterän projektin aikana aiheuttaa vähemmän ongelmia kuin perinteisesti, osittain siitä syystä, että vaatimusten toteuttaminen prioriteettijärjestyksessä ja iteratiivisesti rajaavat muutosten vaikutusaluetta. Ketterien menetelmien on todettu jopa torjuvan muutoksia, eli tasainen työskentely vaatimusten parissa iteraatiosta toiseen tuo prosessiin uudenlaista vakautta.

Ketterässä maailmassa vaatimusten ei tarvitse muuttua notkeiksi tai äärimmäisiksi, mutta onnistunut vaatimusten määrittely edellyttää, että perinteisen vaatimusmäärittelyprosessin tärkeimmät osatehtävät toteutetaan ketterän kehyksen sisällä. Tämä on haasteellista ja edellyttää kokenutta ja osaavaa kehitystiimiä, mutta se ei suinkaan ole mahdotonta. Vaatimusten keräämisessä ja priorisoinnissa voidaan hyödyntää apuna käyttäjäkeskeisen suunnittelun tekniikoita. Ei-toiminnalliset vaatimukset voidaan tunnistaa toiminnallisten ominaisuuksien hyväksyntäkriteerien kautta ja validoida systeemitason testauksella. Niin ikään vaatimusten jäljitettävyyttä ja keskinäisiä vaikutuksia voidaan seurata testitapausten avulla.

Ketterien käytäntöjen skaalaaminen tiimitasolta koko yrityksen tasolle on iso organisatorinen haaste, sillä kömpelöt hierarkiat voivat tehdä ketteryyden osaltaan tyhjäksi; vaatimuksien näkökulmasta skaalautuvuus tai sen puute ei kuitenkaan ole kovin keskeinen ongelma. Suurin haaste ketterissä menetelmissä lienee siinä, miten ne edellyttävät asiakkaalta aktiivista osallistumista kehitysprosessiin, ja miten oivaltavaa, joustavaa ja samalla kurinalaista työskentelyotetta ohjelmistokehittäjiltä vaaditaan. Se on kuitenkin pieni hinta maksettavaksi epärealististen aikataulujen ja virheellisten työmääräarvioiden kurimuksesta pääsemiseksi.

10.2. Avoimia kysymyksiä

Vaatimukset kaikuvat läpi koko ohjelmistoprojektin elinkaaren: niiden pohjalta lasketaan työmäärä- ja kustannusarviot, laaditaan suunnitelmat ja aikataulut, päätetään projektin resursoinnista ja ylläpidosta jne. Näin on asian laita myös ketterissä projekteissa, joskin verhotummin, koska vaatimukset ovat siellä enemmänkin abstraktio, ja ne ovat fyysisesti näkyvissä vain lyhyen aikaa, tynkinä, erilaisille tarinalapuille kuvattuna. Osa-alueita, jonne vaatimuksien ja niiden käsittelyn johdannaisvaikutukset ulottuvat, on enemmän kuin tässä tutkielmassa on mahdollista kattaa, ja tutkielma on näiltä osin vain pintaa raapaiseva.

Kuten tutkielmassa todettiin, sopimuskäytännöt vaikuttavat ketterissäkin projekteissa siihen, miten vaatimukset käytännössä halutaan tulkita. Tutkimusaineistoa siitä, millaiset päätöspuut johtavat eri sopimusmallien valintaan, ja millä perusteella ohjelmistoprojektin kustannusarvio lasketaan, ei ole kovin paljon saatavilla. Ketterien projektien kohdalla myös sopimukset ohjelmakoodin omistusoikeudesta voivat osoittautua

keskeisiksi, jos asiakas haluaa kilpailuttaa ja teettää tarvitsemansa ohjelmiston vaiheittain, niin että eri ohjelmistotaloilla on mahdollisuus jatkaa työtä edellisen ohjelmistojulkaisun päälle.

Eräs mielenkiintoisimpia tutkielmassa raportoituja havaintoja on se, että ketterät menetelmät ehkäisisivät vaatimusten muuttumista projektin aikana. Koska vaatimukset kuitenkin ovat alun alkaen hyvin epämääräisiä ja hahmottomia, niiden muuttumista on ehkä vaikea havaita, mutta ne saattavat silti elää – vaatimustenhan jopa odotetaan kehittyvän! Ketterissä projekteissa päädytään usein refraktoroimaan ohjelmakoodia, mikä saattaa olla viite käyttäjätarinoissa tai niiden prioriteeteissa tai hyväksyntäkriteereissä tapahtuneista muutoksista.

Jossain määrin epäselväksi jää myös se, mikä tekisi ketteriä menetelmiä seuraavasta projektista erityisen adaptiivisen ja muutoksille otollisen. Muutokset vaatimuksissa aiheuttavat joka tapauksia muutoksia toteutettuun toiminnallisuuteen ja voivat edelleen vaikuttaa myös muihin vaatimuksiin. Jos ohjelmiston toimintaa tai rakennetta pitää muuttaa, sitä muutetaan. Tässä ei liene eroa ketterien ja perinteisten projektien välillä. Perinteisissä projekteissa muutostyöt ehkä tehdään lisälaskutuksella, mutta ei ketterissäkään projekteissa harjoiteta hyväntekeväisyyttä, vaan jos projektilla ei ole ”piikki auki” asiakkaan laskuun, jotain toiminnallisuutta saattaa muutosten toteuttamisen takia jäädä pois ohjelmistosta. On vaikea nähdä, mitä mullistavaa ketterät menetelmät voivat tarjota vaatimusten muutoksien suhteen, sillä muutoksiin *reagoiminen* on varsin arkista toimintaa – onhan luontevaa esim. ottaa sateenvarjo mukaansa lähtiessään ulos sateeseen, vaikka aiemmin päivällä olisikin paistanut.

Ketterän toiminnan skaalaaminen on ketterien menetelmien yleistymisen myötä entistä tarpeellisempaa, ja aiheeseen tuovat omat sivujuonteensa erilaiset perinteisen ohjelmistokehityksen puolelta tutut ja tunnettuutta saavuttaneet viitekehykset, kuten CCMI, joita kehittäjät ovat kiinnostuneita yhdistämään ketteriin toimintoihinsa. Leffinwellin [2009] esittelemässä notkeassa organisaatiossa näkyy jo siirtyminen ketteryyden paradigmasta kohti *notkeita* (lean) periaatteita. Ketterät ja notkeat menetelmät kumpuavat samoista lähtökohdista, mutta niiden linjaukset poikkeavat jossain määrin toisistaan. Molemmat menetelmät pyrkivät turhan työn minimointiin, mutta notkeassa ajattelussa yksi keskeisiä periaatteita on ”kokonaisuuden optimointi”, eli pyrkimys ratkaisuihin, jotka ovat tehokkaimpia koko organisaation kannalta. Sellaisenaan se soveltuu hyvin tuotantolinjojen ja miksei myös ohjelmistotalojen toiminnan järjestämiseen. Ketterässä kehityksessä taas on ideana keskittää kaikki toiminta yhden kriittisen tehtävän ympärille – optimointia yhden mission suhteen siis – ja eteenpäin siirrytään vasta kun se saatu hoidetuksi. Nähtäväksi jää, miten ketteryyttä ja notkeutta sekoittavat ohjelmistokehitysmallit onnistuvat yhdistämään nämä ristiriitaiset pyrkimykset.

Viiteluettelo

[Abrahamsson et al., 2002] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen and Juhani Warsta, *Agile Software Development Methods – Review and Analysis*. VTT Publications, 2002.

[Abrahamsson et al., 2003] Pekka Abrahamsson, Juhani Warsta, Mikko Sipponen and Jussi Ronkainen, New Directions on Agile Methods: A Comparative Analysis. In: *Proc. of the 25th International Conference on Software Engineering* (2003), 244-254.

[ACM, 2010] Association for Computing Machinery and IEEE-CS, Software Engineering Code of Ethics and Professional Practice. Available in http://www.acm.org/about/se-code.

[Ambler, 2002] Scott Ambler, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

[Beaumont, 2008] Serge Beaumont, Agile & Contacts. *Presentation in Scrum Gathering Stockholm 2008.* Available in www.scrumalliance.org/resource_download/442.

[Bechtold, 2005] Dirk Bechtold, Requirement Analysis and Agility. In: *Workshop on the Interplay of Requirements Engineering and Project Management in Software Projects* (2005). Available in http://www-swe.informatik.uni-heidelberg.de/home/events/REProMan/Bechtold_re&agility.pdf.

[Beck, 1999] Kent Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Beck, 2007] Kent Beck, interviewed by Paul Krill (November 12th, 2007). Available in http://www.computerworld.com/s/article/9046399/Extreme_Programming_inventor_talks_about_agile_development.

[Beck et al., 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler et al. (February 13th 2001), Manifesto for agile software development. Available in http://agilemanifesto.org/.

[Boehm and Papaccio, 1988] Barry Boehm and Philip Papaccio, Undertanding and Controlling Software costs. *IEEE Transactions on Software Engineering*, **14**, 10 (1988), 1462-1473.

[Boehm and Turner, 2004] Barry Boehm and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

[Cao and Ramesh, 2008] Lan Cao and Balasubramaniam Ramesh, Agile requirements engineering practices: an empirical study. *IEEE Software*, **25**, 1 (2008), 60-67.

[Chow and Cao, 2008] Tsun Chow and Dac-Buu Cao, A survey study of critical success factors in agile software projects. *Journal of Systems and Software*, **81**, 6 (2008), 961-971.

[Cockburn, 2002] Alistair Cockburn, *Agile Software Development*. Addison-Wesley, 2002.

[Conolly et al., 2008] David Connolly, Frank Keenan and Brendan Ryder, Tag Oriented Agile Requirements Identification. In: *Proc. of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems* (2008), 497-498.

[Colfelt, 2010] Anthony Colfelt, Bringing User Centered Design to the Agile Environment. Available in http://www.boxesandarrows.com/view/bringing-user.

[Denger and Olsson, 2005] Christian Denger and Thomas Olsson, Quality Assurance in Requirements Engineering. In: Aybuke Aurun and Claes Wohlin (eds.), *Engineering and Managing Software Requirements*, Springer, 2005, 163-182.

[Ferreira and Cohen, 2008] Carlos Ferraira and Jason Cohen, Agile systems development and stakeholder satisfaction: a South African empirical study. In: *Proc. of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology* (2008), 48-55.

[Firesmith, 2003] Donald Firesmith (September 12th, 2003), The Business Case for Requirements Engineering". *Presentation in Requirements Engineering conference*. Available in http://www.sei.cmu.edu/library/assets/case.pdf.

[Fowler, 2000] Martin Fowler, The New Methodology (2000). Available in http://martinfowler.com/articles/newMethodology.html#TheUnpredictabilityOfRequirements.

[Gallardo-Valencia et al., 2007] Rosalva Gallardo-Valencia, Vivian Olivera and Susan Sim, Are Use Cases beneficial for developers using Agile Requirements? In: *Proc. of the 5th International Workshop on Comparative Evaluation in Requirements Engineering* (2007), 11-22.

[Grisham and Perry, 2005] Paul Grisham and Dewayne Perry, Customer relationships and Extreme Programming. In: *Proc. of the 2005 Workshop on Human and Social Factors of Software* (2005), 1-6.

[Helminen, 2008] Heli Helminen, *Työmääräarviointi ja aikataulusuunnittelu IT-projekteissa*. Pro Gradu -tutkielma, Tampereen Yliopisto, 2008. Available in http://www.cs.uta.fi/research/thesis/masters/Helminen_Heli.pdf.

[Highsmith, 2000] James Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, 2000.

[Hofmann and Lehner, 2001] Hubert F. Hofmann and Franz Lehner, Requirements Engineering as a Success Factor in Software Projects. *IEEE Software*, 18, 4 (2001), 58-66.

[Hughes and Cotterell, 2006] Bob Hughes and Mike Cotterell, *Software Project Management (4th edition)*. McGraw-Hill, 2006.

[IEEE, 1998] IEEE Standard Association, IEEE Specification Std-830, IEEE recommended practice for software requirements specifications. Available in http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html

[Itkonen, 2007] ks. Nykänen [2007].

[Jacobson, 2008] Ivar Jacobson, Everyone wants to be agile. *Dr. Dobb's Digest*, Digital edition, July 24th 2008. Available in http://www.ddj.com/architect/209600574.

[Janis, 1982] Irving Janis, *Groupthink: Psychological Studies of Policy Decisions and Fiascoes*. Wadsworth Publishing, 1982.

[Kajko-Mattsson, 2008] Mira Kajko-Mattsson, Problems in agile trenches. In: *Proc. of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (2008), 111-119.

[Kallio, 2009] Anne Kallio Ville Mäkisen haastattelemana, Kansallinen terveysarkisto myöhästyy. *Tietoviikko* (30.10.2009). Saatavilla osoitteessa http://www.tietoviikko.fi/kaikki_uutiset/article342550.ece.

[Korkala et al., 2006] Mikko Korkala, Pekka Abrahamsson and Pekka Kyllönen, A case study on the impact of customer communication on defects in agile software development. In: *Proc of the Conference on AGILE 2006* (2006), 76-88.

[Kotonya and Sommerville, 1997] Gerald Kotonya and Ian Somerville, *Requirements Engineering*. John Wiley & Sons, 1997.

[Ktata and Levésque, 2009] Qualid Ktata and Ghislain Levésque, Agile development: issues and avenues requiring a substantial enhancement of the business perspective in large projects. In: *Proc. of the 2009 C3S2E Conference* (2009), 59-66.

[Kääriäinen et al., 2004] Jukka Kääriäinen, Juha Koskela, Pekka Abrahamsson and Juha Takalo, Improving requirements management in extreme programming with tool support – an improvement that failed. In: *Proc. of the 30th EUROMICRO Conference* (2004), 342-351.

[Lacey, 2008] Mitch Lacey, When Working Software Is Not Enough: A Story of Project Failure. *Presentation in Agile 2008 Conference*. Available in http://www.infoq.com/presentations/A-Story-of-Project-Failure-Mitch-Lacey.

[Leffingwell, 2009] Dean Leffingwill, A Lean and Scalable Requirements Information Model for the Agile Enterprises. *Presentation in the Object-Oriented Day Conference in Tampere University of Technology, 2009.* Available in http://scalingsoftwareagility.files.wordpress.com/2007/03/the-big-picture-of-enteprise-agilitywhitepaper.pdf.

[Leffingwell and Behrens, 2009] Dean Leffingwell and Pete Behrens, A User Story Primer. Whitepaper for *Agile Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Available in http://scalingsoftwareagility.files.wordpress.com/2009/11/user-story-primer_1.pdf.

[Leffingwell and Widrig, 2000] Dean Leffingwill and Dean Widrig, *Managing Software Requirements – a Unified Approarch*. Addison-Wesley, 2000.

[Logue and McDaid, 2008] Kevin Logue and Kevin McDaid, Handling uncertainty in agile requirement and scheduling using statistical simulation. In: *Proc. of the Agile 2008 Conference* (2008), 72-83.

[Martin et al., 2004] Angela Martin, Robert Biddle and James Noble, The XP customer role in practice: three studies. In: *Agile Development Conference* (2004), 42-54.

[McKeen et al., 1994] James McKeen, Tor Guimaraes and James Wetherbe, The Relationship Between User Participation and User Satisfaction: An Investigation of Four Contingency Factors. *Management Information Systems Quarterly*, **18**, 4 (1994), 427-451.

[Nykänen, 2007] Pirkko Nykänen ja Pentti Itkonen Janne Oraan haastattelemana, Lääkärin näkökulma unohtui potilastietojärjestelmistä. *Lääkärilehti* (8.11.2007). Saatavilla osoitteessa http://www.laakarilehti.fi/uutinen.html?type=1/news_id=5289/L%E4%E4k%E4rin+n%E4k%F6kulma+unohtui+potilastietoj%E4rjestelmist%E4+.

[Paetsch et al., 2003] Frauke Paetsch, Armin Eberlein and Frank Mauer, Requirements engineering and agile software development. In: *Proc. of 12th IEEE International Workshops on Enabling Technologies* (2003), 308-313.

[Palmer and Felsing, 2002] Stephen Palmer and John Felsing, *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.

[Pinheiro, 2003] Francisco Pinheiro, Requirements honesty. *Requirements Engineering*, **8**, 3 (2003), 183-192.

[Racheva et al., 2008] Zornitza Racheva, Maya Daneva and Luigi Buglione, Supporting the Dynamic Reprioritization of Requirements in Agile Development of Software Products. In: *Proc. of the 2nd International Workshop on Software Product Management* (2008), 49-58.

[Reed et al., 2004] Karl Reed, Ernesto Damiani, Gabriele Gianini and Alberto Colombo, Agile management of uncertain requirements via generalizations: a case study. In: *Proc. of the 2004 Workshop on Quantitative Techniques for Software Agile Process* (2004), 40-45.

[Rodríguez et al., 2009] Pilar Rodríguez, Agistín Yagüe, Pedro Alarcón and Juan Garbajosa, Some findings concerning requirements in agile methodologies. In: Frank Bomaries, Markku Oivo, Päivi Jaring and Pekka Abrahamsson (eds.), *Product-Focused Software Process Improvement – 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009 Proceedings*, Springer, 2009, 171-184.

[Royce, 1970] Winston Royce, Managing the Development of Large Software Systems. In: *Proc. of the IEEE WESCON Conference* (1970), 1-9.

[Schwaber, 2008a] Ken Schwaber interviewed by Agile Collab (February 19th 2008), Interview with Ken Schwaber. Available in http://www.agilecollab.com/interview-with-ken-schwaber.

[Schwaber, 2008b], ks. Shalloway [2008].

[Schwaber and Beedle, 2002] Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum*. Prentice-Hall, 2002.

[Shalloway, 2008] Ken Schwaber and Alan Shalloway (16th November 2008). *Discussion in Scrum Development forum.* Available in http://groups.yahoo.com/group/scrumdevelopment/message/34566.

[Sharp et al., 2009] Helen Sharp, Hugh Robinson and Marian Petre, The role of Physical Artefacts in Agile Software Development: Two Complementary Perspectives. *Interacting with Computers*, **21**, 1-2 (2009), 108-116.

[Sillitti and Succi, 2005] Alberto Sillitti and Giancarlo Succi, Requirements Engineering for Agile Methods. In: Aybüke Aurum and Claes Wohlin (eds.), *Engineering and Managing Software Requirements*, Springer, 2005, 209-326.

[Sillitti et al., 2005] Alberto Sillitti, Martina Ceschi, Barbara Russo and Giancarlo Succi, Managing uncertainty in requirements: a survey in plan-based and agile companies. In: *Proc. of 11th IEEE International Software Metrics Symposium* (2005), 17-26.

[Standish, 1995] The Standish Group International, CHAOS research study report. Available in http://net.educause.edu/ir/library/pdf/NCP08083B.pdf.

[Standish, 2009] The Standish Group International, CHAOS Summary 2009: New Standish group report shows more projects failing and less successful projects. Available in http://www.standishgroup.com/newsroom/chaos_2009.php.

[Stapleton, 1997] Jennifer Stapleton, *Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley, 1997.

[Stephens and Rosenberg, 2003] Matt Stephens and Doug Rosenberg, *User Programming Refactored: The Case Against XP*. A! Press, 2003.

[Stevens, 2009] Peter Stevens, 10 Contracts for your next Agile Software Project (April 29th, 2009). Available in http://agilesoftwaredevelopment.com/blog/peterstev/10-agile-contracts.

[Surendra, 2008] Nanda Surendra, Using on ethnographic process to conduct requirements analysis for agile systems development. *Information Technology and Management*, **9**, 1 (2008), 55-69.

[Turk et al., 2005] Daniel Turk, Robert France and Bernhard Rumpe, Assumptions underlying agile software development processes. *Journal of Database Management*, **16**, 4 (2005), 62-87.

[Voutilainen, 2008] Tomi Voutilainen Mikko Pulliaisen haastattelemana, Kunnat ja valtio ovat ostaneet holtittomasti it-sikaa säkissä. *Aamulehti* (27.12.2008).

[Vuori, 2009] Matti Vuori, Vaatimusmäärittelyn huonoimmat käytännöt. *Systeemityö*, 2 (2009), 20-21.

[Wake, 2003] William Wake, INVEST in Stories / SMART Tasks (August 2003). Available in http://xp123.com/xplor/xp0308/.