AGILE-MENETELMÄT

Antti Kainulainen

Opinnäytetyö

YLEMPI AMK-TUTKINTO

Toukokuu 2008

JYVÄSKYLÄN
AMMATTIKORKEAKOULU

Teknologiaosaamisen johtamisen koulutusohjelma

Tekijä(t)

KAINULAINEN, Antti

Julkaisun laji
Opinnäytetyö. Ylempi ammattikorkeakoulututkinto.

Sivumäärä	Julkaisun kieli
112	suomi

Luottamuksellisuus
-

Työn nimi

AGILE-MENETELMÄT

Koulutusohjelma
Teknologiaosaamisen johtamisen koulutusohjelma. Ylempi ammattikorkeakoulututkinto.

Työn ohjaaja(t)
MATILAINEN, Jorma, yliopettaja
FRANSSILA, Tommi, lehtori

Toimeksiantaja(t), yhteyshenkilö
TietoEnator Oyj Telecom & Media
MANNINEN, Jani, Section Manager

Tiivistelmä
Ohjelmistoprojektit on perinteisesti toteutettu niin sanottuun vesiputousmalliin perustuvaa jaksottaista prosessia noudattaen. Nopeasti kehittyvällä ohjelmistoalalla on kuitenkin viime aikoina arvosteltu jaksottaisten prosessimallien sovellettavuutta. Alkuperäisiä suunnitelmia joudutaan lähes poikkeuksetta muuttamaan projektin kestäessä ja tällöin jaksottaisen mallin mukaisesti työskenneltäessä on palattava aiempiin vaiheisiin, vaikka niiden läpivientiin on projektin alussa käytetty runsaasti aikaa. Vesiputousmallinen prosessi on monissa yrityksissä korvattu asiakkaan ohjaamalle iteratiiviselle ja inkrementaaliselle työskentelytavalle perustuvalla agile- eli ketteräksi menetelmäksi kutsutulla prosessimallilla.

Tässä tutkimuksessa tarkastellaan kirjallisuuskatsauksen sekä empiirisen tapaustutkimuksen avulla agile-menetelmien mahdollisia hyötyjä tietotekniikan palvelualan näkökulmasta. Tavoitteena oli selvittää, mikä agile-menetelmistä olisi viisainta ottaa käyttöön. Tutkimuksessa käsitellään kaikkia alkuperäisiä agile-menetelmiä (ASD, XP, Scrum, Crystal-menetelmät, FDD ja DSDM). Tutkimuksen perusteella sopivaa agile-menetelmää käyttäen voidaan tehostaa ohjelmistoprosessia. Omalle projektille soveltuvan menetelmän valitsemiseen tulee kuitenkin kiinnittää huomiota, sillä mitään agile-menetelmää ei tutkimuksen perusteella ole järkevää soveltaa, ellei jokaista valitun menetelmän osa-aluetta ole mahdollista noudattaa. Agile-menetelmä tulee ottaa käyttöön joko muuttamalla nykyinen ohjelmistoprosessi täysin valitun menetelmän mukaiseksi, kehittämällä oma agile-prosessi tai upottamalla agile-menetelmien tärkeimpiä osia nykyiseen ohjelmistoprosessiin mahdollisuuksien puitteissa. Perinteisiin menetelmiin tottuneen yrityksen on viisainta aloittaa agile-menetelmien kokeileminen viimeksi mainittua sovellustapaa noudattaen.

Avainsanat (asiasanat)
Agile, agile-menetelmät, ketterät menetelmät, ohjelmistokehitys, ohjelmistoprosessi

Toimeksiantajan myöntämä raportin julkaisulupa

Paikka	Aika	Allekirjoitus
		Nimenselvennös

Author(s)	Type of Publication	
KAINULAINEN, Antti	Master's Thesis	
	Pages 112	Language Finnish
	Confidential ☐ until	

Title
AGILE SOFTWARE DEVELOPMENT

Degree Programme
Professional Master’s Degree Programme in Technological Competence Management.

Supervisor(s)
MATILAINEN, Jorma, Senior Lecturer
FRANSSILA, Tommi, Lecturer

Commissioner(s), contact person
TietoEnator Oyj Telecom & Media
MANNINEN, Jani, Section Manager

Abstract
Software projects have traditionally followed a sequential process model called the waterfall model. However, sequental processes are criticized for responding too slowly to changes, which are more and more common in the rapidly evolving software business. More often than not, initial project plans need to be updated during the project, even though the planning phase of the project has required a signficant effort in the beginning of the project. Many companies have replaced their waterfall-based sequential processes with customer-driven iterative and incremental process models called agile methodologies.

The aim of this thesis was to describe the agile methodologies, evaluate their advantages and determine which agile-methodology is the most useful from software services business point of view. All original agile methodologies – which include ASD, XP, Scrum, Crystal-family, FDD and DSDM – are concidered.

Based on the results, it is possible to improve the software process by implementing an agile methodology. However, every specified factor in the selected agile methodology must be implemented in order for the methodology to be effective. If all factors of the methodology cannot be implemented, a methodology more suitable for your project environment should be selected. Due to the extent of the changes required when moving from a sequential process model to an agile model, the safest way to go is to embed as much of the most important factors of agile methodologies as possible to the exisiting software process first. After this initial step toward agility, the possible change to a defined agile methodology later on will be conciderably easier.

Keywords
Agile, Software Development, Software Process

Commissioner’s permission to publish this report

Place	Date	Signature
		Clarification

1. JOHDANTO 6

2. TAUSTATIETOA 9

2.1 Ohjelmistopalvelualan piirteitä 9
2.2 Vesiputousmalli 9
2.3 Lean-ohjelmistokehitys 13
2.3.1 Eroon jätteistä 14
2.3.2 Prosessi 15
2.3.3 Kanban-ajattelu 18
2.4 Yleistä agile-menetelmistä 19

3. AGILE-MENETELMÄT 22

3.1 Scrum 22
3.1.1 Scrumin vaihetuotteet 22
3.1.2 Scrum-roolit 25
3.1.3 Scrum-prosessi 26
3.1.4 Sprintien tuotokset 28
3.1.5 Skaalaus 29
3.1.6 Testaaminen 29
3.1.7 Scrumin käyttökokemuksia 30
3.1.8 Yhteenveto ja arviointi 31
3.2 Extreme Programming (XP) 33
3.2.1 XP:n periaatteet 33
3.2.2 XP:n tiimit ja roolit 39
3.2.3 XP-prosessi 41
3.2.4 Yhteenveto ja arviointi 46
3.3 Adaptive Software Development (ASD) 49
3.3.1 ASD-prosessi 50
3.3.2 Yhteenveto ja arviointi 54
3.4 Crystal-menetelmät 55
3.4.1 Yleistä Crystal-perheestä 55
3.4.2 Menetelmän valitseminen 59
3.4.3 Crystal Clear 60
3.4.4 Crystal Orange 61
3.4.5 Crystal Orange Web 62
3.4.6 Yhteenveto ja arviointi 64
3.5 Feature-Driven Development (FDD) 65
3.5.1 FDD-prosessi 66
3.5.2 Skaalautuvuus 68
3.5.3 Yhteenveto ja arviointi 68
3.6 Dynamic Systems Development Method (DSDM) 70
3.6.1 DSDM:n periaatteet 70
3.6.2 DSDM-prosessi 73
3.6.3 Yhteenveto ja arviointi 75

4. TULOKSET 77

4.1 Lähdekritiikki 77
4.2 Agile-menetelmien vertailua 78
4.3 Agile-menetelmien hyödyt 78
4.3.1 Oikea tuote 79
4.3.2 Riittävä tuote aikataulussa 81
4.3.3 Tukitoimia 81
4.3.4 Suunnittelu ja dokumentointi 83

5. TULOSTEN TARKASTELU 84

5.1 Kokeiluprojektit 84
5.2 Sovellettavuus 86
5.3 Suositeltava ratkaisu 87
5.3.1 Vesiputousmalli, vaatimusmäärittely asiakkaalta, asiakas passiivinen 88
5.3.2 Vesiputousmalli, vaatimusmäärittely yhteistyössä, asiakas passiivinen 90
5.3.3 Vesiputousmalli, priorisoitu vaatimusmäärittely yhteistyössä, asiakas passiivinen 91
5.3.4 Vesiputousmallin vaihetuotteet, priorisoitu vaatimusmäärittely yhteistyössä, asiakas passiivinen 92
5.3.5 Vesiputousmallin vaihetuotteet, priorisoitu vaatimusmäärittely yhteistyössä, asiakas aktiivinen 93
5.3.6 Ei ennalta määriteltyä prosessia, aktiivinen asiakas 93
5.3.7 Tukitoimien lisääminen 94
5.3.8 Yhteenveto 94

LIITE 1 Agile Manifesti
LIITE 2 CASE: Sovellettu Scrum
LIITE 3 CASE: Oma ketterä prosessi

1. JOHDANTO

Ohjelmistokehityksessä on 1970-luvulta saakka luotettu yleisesti niin kutsutun v*esiputousmallin* mukaisiin prosesseihin. Vesiputousmallisessa prosessissa vaiheet seuraavat toinen toistaan ja edellisessä vaiheessa valmistunut vaihetuote on seuraavan vaiheen sisääntulokriteeri. Kaikkiin vesiputousmallin vaiheisiin kuuluu laadunvarmistusta, kuten testausta tai katselmointeja. (Haikala&Märijärvi 1998, 27).

Teoriassa tämä malli kuulostaa hyvinkin toimivalta, mutta käytännössä se on osoittautunut kankeaksi. Vesiputousmalliin perustuvia prosesseja arvostellaan erityisesti siitä, että onnistuakseen on projektin suunnitteluvaiheisiin (esitutkimus, määrittely, suunnittelu) käytettävä paljon aikaa. Näin kestää kauan ennen kuin päästään itse ohjelmointityön tekemiseen - puhumattakaan tilanteesta, jossa käytettävissä olisi jotain valmista toiminnallisuutta sisältävä tuote. Prosessin noudatettavuutta on myös arvosteltu, sillä käytännössä ohjelmistoprojektin alussa kaikkien lopullisten vaatimusten luetteleminen on hyvin vaikeaa - joissakin tapauksissa jopa mahdotonta (Kroll&Kruchten 2004, 6-9; Pressman 2005, 79-80).

Vesiputousmallisen ohjelmistokehityksen korvaajaksi on viime aikoina uskottu nousevan teollisuudessa suositun Lean-menetelmän ohjelmistokehitykseen soveltuvan version, Lean-ohjelmistokehityksen, pohjalta kehitetyt agile- eli ketterät menetelmät. (Pressman 2005, 103-105).

Agile-menetelmissä ohjelmistoprojekti on iteratiivinen ja inkrementaalinen. Tämä tarkoittaa sitä, että työ on jaettu tietyn mittaisiin iteraatioihin eli jaksoihin (yleensä 2-4 viikkoa), joissa toteutetaan aina ohjelmistoinkrementti eli toimiva ja ajettavissa oleva osa lopullisesta ohjelmistosta. (Cockburn 2007, 373-374).

Agile-menetelmien tutkimisesta tekee haasteellista se, että puolueettomia julkaisuja aiheesta ei käytännössä ole. Suurin osa agile-kirjallisuudesta on menetelmien perustajien kirjoittamia ja siten usein jopa mainospuheen kaltaisia.

Agile-menetelmiä kritisoivia lähteitä on saatavilla valitettavan vähän. Yleisin kritiikki agile-menetelmiä kohtaan kohdistuu menetelmän sovellettavuuteen laajempiin projekteihin (Kalermo&Rissanen 2002, 85), mikä onkin tutkimusten vähyyden vuoksi aiheellista kritiikkiä. Valitettavasti valtaosa agile-kritiikistä on kuitenkin kirjoitettu puutteellisten tietojen ja puutteellisen kokemuksen perusteella. Yleisintä on agile-menetelmien kritisoiminen yksittäisen, useimmiten jonkin agile-menetelmän virheellisestä käyttöönotosta johtuneen epäonnistumisen perusteella (esimerkiksi Yegge 2006) tai ilman käytännön kokemusta ainoastaan ennakkoasenteiden sävyttämän teoriatiedon perusteella (esimerkiksi Longstreet 2008).

Aikaisemmissa tutkimuksissa agile-menetelmiä on käsitelty joko hyvin yleisellä tasolla tai on valittu vain 1-2 yleisintä menetelmää (yleensä *XP* ja *Scrum*) ja niiden on yleistetty koskevan koko agile-menetelmien kirjoa (Nelli 2006). Tässä tutkimuksessa käsitellään kaikkia alkuperäisiä agile-menetelmiä, eli niitä kevytmenetelmiä, joilta oli edustajia mukana Agile Alliancen perustamiseen johtaneessa kokouksessa. Nämä menetelmät ovat ASD (Adaptive Systems Development), XP (Extreme Programming), Scrum, Crystal-menetelmät, FDD (Feature-Driven Development) sekä DSDM (Dynamic Systems Development Method). (Cockburn 2007, 369).

Agile-menetelmät on alun perin suunniteltu tuotebisneksen käyttöön, joten menetelmien soveltuvuutta palvelualan työhön ei ole erityisesti tutkittu. Tässä tutkimuksessa tarkastellaan agile-menetelmien sovellettavuutta nimenomaan palvelualan näkökulmasta.

Tutkimuksessa pyritään vastaamaan seuraavaan kysymykseen:
Voidaanko ohjelmistokehityksen tehokkuutta tietotekniikan palvelualan yrityksessä parantaa agile-ajattelun mukaisella ohjelmistokehitysmallilla?

Lisäksi, mikäli tutkimuksen perusteella vastaus kysymykseen on myönteinen, pyritään hakemaan vastaus myös jatkokysymykseen:

Mikä agile-menetelmistä olisi käyttökelpoisin tietotekniikan palvelualan yrityksessä?

Tavoitteena on myös kuvailla eri agile-menetelmiä riittävän yksityiskohtaisesti, jotta agile-menetelmistä kiinnostuneet voisivat käyttää tätä tutkimusta tarvittaessa hakuteoksena.

Tutkimusmenetelminä on käytetty kirjallisuustutkimusta sekä empiiristä tapaustutkimusta. Tapauskuvaukset (Liitteet 2 ja 3) on kuitenkin salassapitosyistä jätetty pois verkossa julkaistusta versiosta.

2. TAUSTATIETOA

2.1 Ohjelmistopalvelualan piirteitä

Ohjelmistopalvelualalla tuotekehitysprosessin muokkaaminen voi olla haasteellisempaa kuin esimerkiksi tuotebisneksessä. Usein asiakas vaatii tiettyjen vaihetuotteiden toteuttamista, ja joissain tapauksissa jopa määrää käytettävän ohjelmistoprosessin tai sen raamit. Palvelualalla toteutettavat ohjelmistoprojektit ovat myös usein vain osa suurempaa ohjelmistoprojektia tai –kokonaisuutta. Tällöin esimerkiksi rajapintojen tarkka määritteleminen on useiden saman ohjelmistokokonaisuuden eri komponenttien yhtäaikaisen toteuttamisen kannalta välttämätöntä.

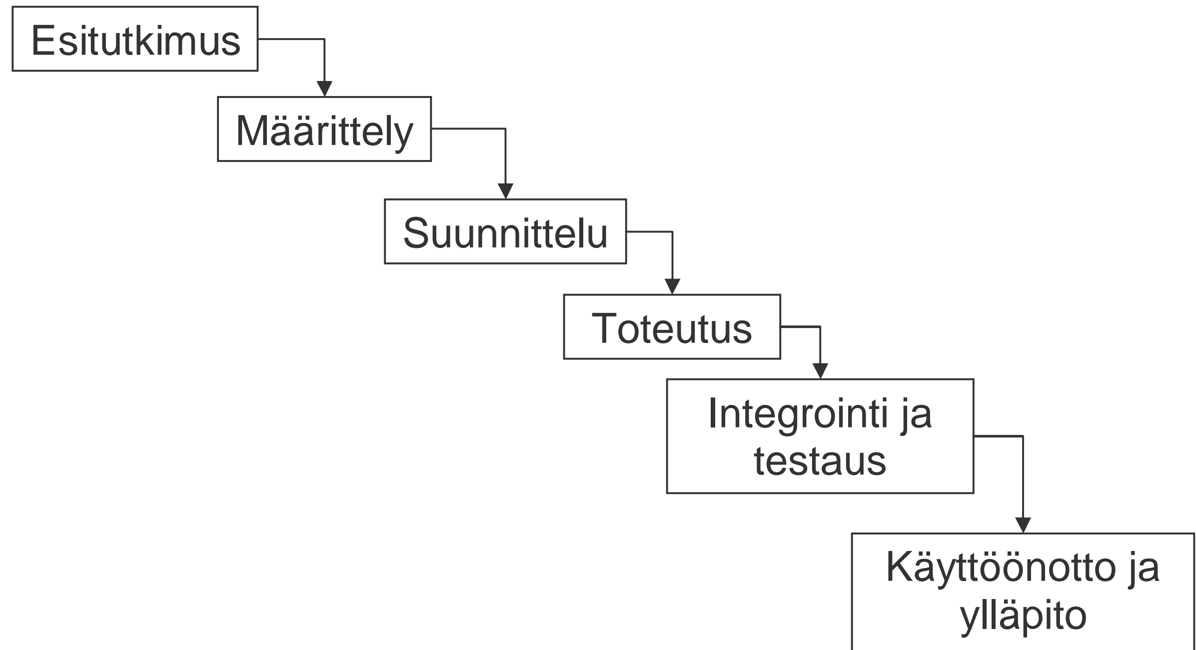
Projektitiimeissä on palvelualalla usein työntekijöitä, jotka ovat mukana kyseisessä projektissa vain osalla työpanoksestaan. Tällaisia tilanteita pyritään tietenkin välttämään, mutta koska koko henkilöstölle on saatava joka päivä mahdollisimman suuri käyttöaste ja koska projektit ovat harvoin täsmälleen saman kestoisia, ovat tällaiset erikoisjärjestelyt joskus tarpeen. Tämä tilanne on tärkeää huomioida etenkin palavereja järjestettäessä. Palaveriin käytetty aika suhteessa tehokkaaseen työaikaan on 50 % työpanoksella mukana olevan henkilön tapauksessa kaksinkertainen 100 % työpanoksella osallistuvaan verrattuna.

Palvelualalla projekteihin myös osallistuu usein työntekijöitä useilta paikkakunnilta. Tyypillisesti projektiryhmään kuuluu myös useampien eri yritysten työntekijöitä.

2.2 Vesiputousmalli

Vesiputousmallista esiintyy useita muunnelmia, mutta yleisimmin ne sisältävät kuvion 1 mukaiset vaiheet: esitutkimus, määrittely, suunnittelu, toteutus sekä

integrointi ja testaus. Usein lopussa on vielä erikseen käyttöönotto- ja ylläpitovaihe (Haikala&Märijärvi 1998, 25).



KUVIO 1 *Vesiputousmalli* (Haikala&Märijärvi 1998, 25 muk. Scach 1990)

Esitutkimusvaihe alkaa yleisten, korkeamman tason vaatimusten selvittämisellä. Käytännössä tässä vaiheessa siis selvitetään miksi ohjelmisto tai järjestelmä tulee tehdä. Tämä vaihe on hyvin haastava, sillä asiakkaan todelliset tarpeet on saatava selville ja ne on ymmärrettävä perusteellisesti. Mikäli esitutkimusvaiheessa määritetyt asiakasvaatimukset ovat virheellisiä tai väärin ymmärrettyjä, ei hyvään lopputulokseen ole mahdollista päästä. (Haikala & Märijärvi, 26)

Määrittelyvaiheessa laaditaan ensin esitutkimusvaiheen tulosten perusteella vaatimusmäärittelydokumentti, johon kirjataan kaikki asiakkaan tarvitsemat ominaisuudet. Tämäkin työvaihe vaatii äärimmäistä tarkkuutta, sillä juuri tätä dokumenttia vasten tarkastellaan projektin lopuksi sisältääkö lopputuote kaikki tilatut ominaisuudet. Määrittelyvaiheessa työstetään vielä vaatimusmäärittelyn pohjalta ohjelmiston toteutusmäärittelydokumentti, jossa kuvataan ohjelmiston toiminnot sekä rajoitukset. Toteutusmäärittelydokumenttiin voi kirjata myös

sellaisia ohjelmistoa koskevia vaatimuksia, mitkä eivät ole toiminnallisia vaatimuksia, esimerkiksi suoritustehoon liittyviä vaatimuksia. (Haikala & Märijärvi, 27)

Suunnitteluvaiheessa suunnitellaan ohjelmiston toteutus toteutusmäärittelyn perusteella. Usein suunnitteluvaiheessa laaditaan ensin korkean tason suunnitelma (arkkitehtuurisuunnitelma) ja sen jälkeen suunnitellaan toteutus tarkemmalla tasolla. (Haikala & Märijärvi, 28-29)

Toteutusvaiheessa kirjoitetaan itse ohjelmakoodi suunnitteludokumentin mukaisesti. Ohjelmointivaihe katsotaan päättyneeksi, kun kaikki vaatimusmäärittelyssä mainitut ominaisuudet on toteutettu ja lopullisesta ohjelmistosta saadaan virheetön käännös. (Haikala & Märijärvi, 29)

Testausvaiheessa ohjelmistosta yritetään löytää virheitä. Yleensä testausvaihe sisältää ainakin moduulitestauksen, toimintotestausta sekä integrointitestausta. (Haikala & Märijärvi, 29)

Ohjelmiston käyttöönoton jälkeen ohjelmisto siirtyy yleensä ylläpidon piiriin. Ylläpito on asiakkaan ongelmien ratkomista, esimerkiksi virheiden korjaamista, neuvontaa sekä parannusehdotuksien käsittelemistä. (Haikala & Märijärvi, 29)

Paperilla vesiputousmalli näyttää hyvinkin toimivalta, mutta käytännössä se on osoittautunut vaikeaksi noudattaa. Vesiputousmallin mukaisen ohjelmistoprosessin onnistuminen on täysin riippuvainen määrittely- ja suunnitteluvaiheiden täsmällisyydestä. Käytännössä on kuitenkin huomattu, että kaikkien lopullisten vaatimusten luetteleminen ja määritteleminen projektin aluksi on usein hyvin hankalaa, joskus esimerkiksi muuttuvista olosuhteista johtuen jopa mahdotonta (Pressman 2005, 80). Määrittely- ja suunnitteluvaiheiden kriittisyyden vuoksi niihin käytetään yleensä myös runsaasti aikaa ja näin kestää kauan ennen kuin päästään itse ohjelmointityön tekemiseen, puhumattakaan tilanteesta, jossa käytettävissä olisi jotain valmista toiminnallisuutta sisältävä tuote asiakkaalle näytettäväksi. (Cockburn 2001, 218).

RUP (Rational Unified Process)-menetelmän kannattajat kritisoivat vesiputousmalliin perustuvien menetelmien tapaa pyrkiä tuottamaan lopullisia vaatimus- ja suunnitteludokumentteja suurella vaivalla projektin aluksi, vaikka noita dokumentteja joudutaankin käytännössä aina päivittämään projektin lopuksi. RUP-prosessin perusrakenne muistuttaa läheisesti vesiputousmallia, mutta suurin ero on siinä, että vaatimus- ja suunnitteludokumenttien todennäköinen päivittämisen tarve tiedostetaan ja tästä syystä vaihetuotteita ei alkuvaiheessa edes yritetä saattaa lopulliseen muotoonsa, vaan ne tehdään aina vain seuraavaan vaiheeseen siirtymisen mahdollistavalle tasolle. (Kroll&Kruchten 2004, 11, 91).

Myös muuttuneisiin vaatimuksiin suhtautumista vesiputousmalliin perustuvissa ohjelmistoprojekteissa on usein kritisoitu (Kalermo&Rissanen 2002, 40 mukaan Boehm, 2002). Lisäksi vesiputousmalliin perustuvassa projektissa syntyy vaihetuotteina paljon dokumentteja. Tämä on joidenkin kritisoijien mukaan johtanut jopa siihen, että dokumenttien tekemiseen keskitytään enemmän kuin itse ohjelmointityöhön (DeMarco&Lister 1987, 116-118). Suuri määrä vaihetuotteita aiheuttaa myös suuren määrän päivitystyötä jokaisen ohjelmistoon tehtävän muutoksen yhteydessä (Kroll&Kruchten 2004, 60).

Vesiputousmalliin kohdistuneeseen kritiikkiin tulee kuitenkin suhtautua varauksella, sillä vaikka osa kritiikistä osuukin täysin kohdalleen, on suuri osa myös vahvasti karrikoitua ja osa suorastaan aiheetonta. Karrikointi johtuu yleensä tarkoituksellisesta mustamaalaamisesta, jonka tavoitteena on omien vaihtoehtoisten menetelmien etujen suurenteleminen. Aiheeton kritiikki taas on usein seurausta osallistumisesta huonosti johdettuihin tai suunniteltuihin vesiputousmallin mukaisiin projekteihin tai käsitysten perustamiseen vain kirjatiedon varaan käytännön kokemuksen puutteesta johtuen.

Vesiputousmallin kiistattomia etuja ovat kuitenkin projektin keston ennustettavuus sekä huolellisten suunnitteluvaiheiden tuoma järjestelmän vakaus. Myös esimerkiksi ulkoisten rajapintojen lyöminen lukkoon sekä dokumentoiminen jo

projektin alkuvaiheessa mahdollistavat sen, että sidosryhmät (esimerkiksi testaajat tai kyseistä ohjelmistoa hyödyntävien muiden ohjelmistojen tekijät) voivat aloittaa oman työskentelynsä yhtäaikaisesti projektiryhmän kanssa. (Kroll&Kruchten 2004, 60).

2.3 Lean-ohjelmistokehitys

Lean-ohjelmistokehitys perustuu seitsemälle periaatteelle: hankkiutuminen ”jätteistä” eroon, päätöksenteon venyttäminen, nopea toimitus, palautteesta oppiminen, riittävien valtuuksien antaminen tiimeille, järjestelmän yhtenäisyyden sisäänrakentaminen sekä kokonaisuuden hahmottaminen. (Poppendieck 2003, XXV-XXVII)

Kaikki Leanin periaatteet liittyvät toisiinsa, mutta kaiken perustana on tarpeettoman työn – jätteiden (*waste)*, kuten sitä Leanissä kutsutaan – hylkääminen. ”Jätteillä” tarkoitetaan kaikkea projektissa tehtävää työtä, mikä ei tuo asiakkaalle lisäarvoa. ”Jätettä” voi siis olla vaikkapa jokin dokumentti, joka vain itsepintaisesti roikkuu prosessissa mukana, vaikkei sitä itse asiassa ole kukaan koskaan todella tarvinnut. (Poppendieck & Poppendieck 2003, XXV, 2-4)

Venyttämällä päätöksentekoa vältetään hätiköityjä päätöksiä. Nopeasti tehty päätös perustuu usein arvioille työskentelykentästä, kun taas pidempään harkittu päätös ehtii saamaan tuekseen faktoja. Hätiköidyt päätöksen johtavat usein ylimääräiseen työhön – eli ”jätteisiin”. Päätöksentekoa ei kuitenkaan ole mahdollista venyttää, ellei kyetä toimittamaan asiakkaalle lisäarvoa hyvin nopeasti projektin käynnistyttyä. Nopea toimitus mahdollistaa myös palautteen saamisen ja palaute mahdollistaa oppimisen. Nopea toimitus taas on mahdollista vain mikäli tiimit voivat työskennellä tehokkaasti, ja sen edellytyksenä on, että tiimeillä on riittävästi valtuuksia työnsä mahdollisimman itsenäiseen tekemiseen. Koko järjestelmä kuitenkin kaatuu omaan mahdottomuuteensa, ellei kokonaisuutta ole hahmotettu heti projektin aluksi ja kokonaisuuden hahmottaminen on lähes mahdotonta, ellei järjestelmä ei ole yhtenäinen, selkeä kokonaisuus. (Poppendieck & Poppendieck 2003, XXV-XXVII)

2.3.1 Eroon jätteistä

”If a component is sitting on a shelf gathering dust, that is waste. If a manufacturing plant makes more stuff than is immediately needed, that is waste. If developers code more features than are immediately needed, that is waste.”
- Mary & Tom Poppendieck (Poppendieck & Poppendieck 2003, XXV)

Kaikkein tärkein Lean-periaatteista on jätteiden tunnistaminen ja hankkiutuminen niistä eroon. Lean-menetelmää otettaessa käyttöön onkin aluksi opeteltava tunnistamaan jätteet, eli työ ja vaihetuotteet, jotka eivät tuo lisäarvoa asiakkaalle. Tässä toimii apuna karrikoitu kysymys: voiko annetun tehtävän tehdä ilman X:ää? Jos voi, X on jätettä. (Poppendieck & Poppendieck 2003, 2-4)

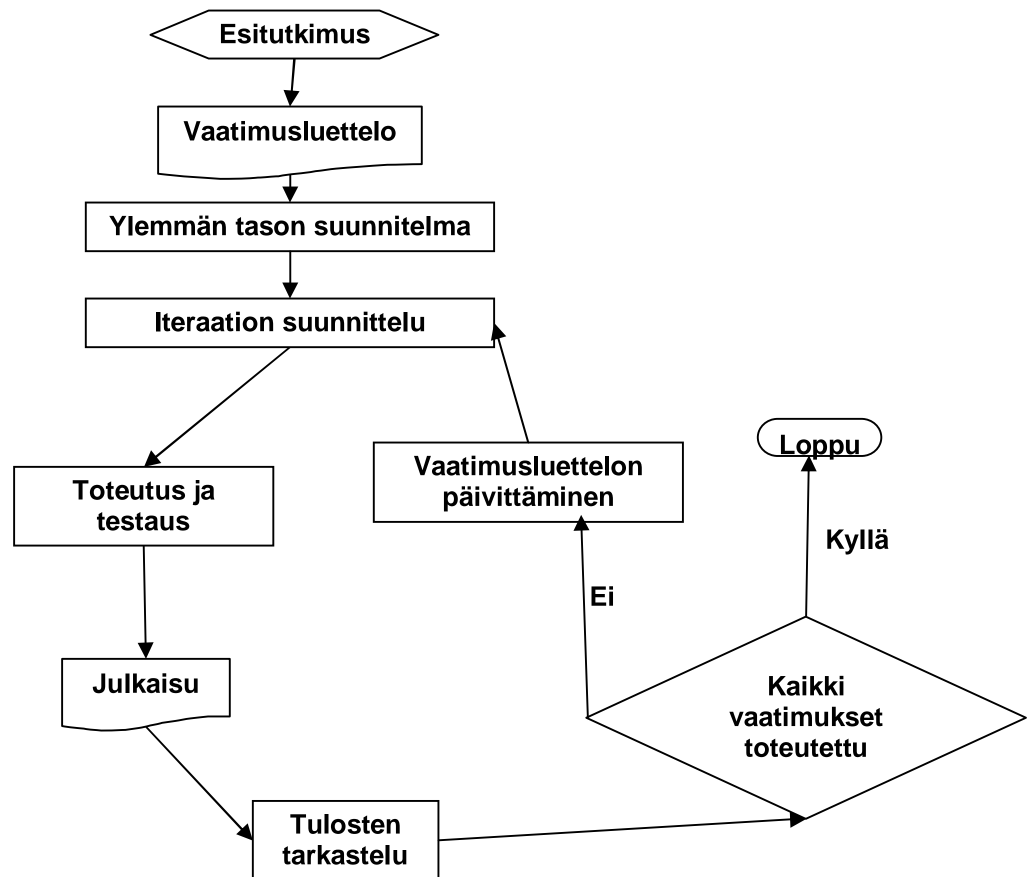
Kun jätteet on opittu tunnistamaan, etsitään työmäärältään suurin jätteeksi luokiteltava asia – ja pudotetaan se pois ohjelmistoprojektista. Tämän jälkeen etsitään pahin jäljelle jäävä jätteiden lähde, ja pudotetaan se pois. Kun tätä jatketaan niin kauan kun jätettä vain löytyy, tullaan lopulta huomaamaan, että jopa osa ennakolta ehdottoman tärkeäksi oletetuista prosessin osista tai vaihetuotteista on todellisuudessa mukana vain ”kaiken varalta”, eikä tuo todellista lisäarvoa asiakkaalle. (Poppendieck & Poppendieck 2003, 2-4)

Ohjelmistotyössä on perinteisiä menetelmiä käytettäessä useita jätteiksi luokiteltavia osa-alueita. Esimerkiksi sellaiset dokumentit, jotka tehdään aina, mutta joita kukaan ei käytännössä lue, ovat jätettä. Hyvä tapa testata dokumentin tarpeellisuus, on kysyä: odottaako joku tämän dokumentin valmistumista? Jos vastaus on ei, dokumentti on jätettä. Myös ylimääräiset ominaisuudet, joita ”saatetaan tarvita”, ovat selkeästi jätettä. sillä mikäli ne toteutetaan, on ne myös integroitava. Kun ominaisuudet on integroitu, on ne myös testattava aina kun ohjelmistoon lisätään uutta toiminnallisuutta. Nyrkkisääntönä ominaisuuksien osalta voidaan pitää sitä, että jos ominaisuutta ei tarvita juuri nyt, se on jätettä. (Poppendieck & Poppendieck 2003, 4-8, 50)

Myös viat aiheuttavat paljon ylimääräistä työtä ja ovat siten jätettä. Tästä syystä Lean-ajattelun mukaan koodia tulisi testata heti kun se on valmista, koska pieni vika, joka korjataan mahdollisimman aikaisessa vaiheessa, aiheuttaa merkittävästi vähemmän päänvaivaa kuin suuren vian jäljittäminen ja korjaaminen myöhemmin. (Poppendieck & Poppendieck 2003, 8, 26)

2.3.2 Prosessi

Lean-ohjelmistokehityksessä työskennellään inkrementaalisesti ja iteratiivisesti. Jokaisen iteraation tuloksena syntyy toimiva sovellus, jossa on mukana osa lopullisen ohjelmiston ominaisuuksista. Tämä työskentelytapa mahdollistaa sen, että asiakas voi ohjata projektiryhmän työskentelyä, sillä asiakkaalle voidaan toimittaa toimiva ”demoversio” tuotteesta. Demoversion toiminnasta annetun palautteen perusteella voidaan tehdä tarvittavia muutoksia tuotteeseen. Näin varmistetaan, että lopullinen tuote on juuri sellainen, minkä asiakas halusi. Iteraatioiden pituudet voivat olla projektista riippuen kahdesta jopa kymmeneen viikkoon. (Poppendieck & Poppendieck 2003, 28).



KUVIO 2 *Yksinkertaistettu Lean-menetelmän mukainen prosessi*

Lean-projekti (kuvio 2) alkaa asiakasvaatimusten kartoittamisella. Asiakkaan kanssa listataan asiakkaan haluamat ominaisuudet ja tehdään ensimmäinen versio vaatimusluettelosta. Tuota luetteloa voidaan päivittää tarvittaessa jokaisen iteraation jälkeen.

Seuraavaksi suunnitellaan toteutuksen pääperiaatteet (high-level design), käytettävät työkalut sekä muut projektityön aloittamiselle välttämättömät seikat (esimerkiksi ohjelmointikielen valitseminen, linjanvetoja kolmansien osapuolien ohjelmistojen hyödyntämisestä, ja niin edelleen). Lean-ajattelussa haastetaan perinteisten menetelmien tapa suunnitella kaikki alusta pitäen hyvin yksityiskohtaisesti, vaikkei kaikkea tarvittavaa tietoa ole saatavilla. Poppendieckien mukaan näin toimittaessa ei itse asiassa suunnitella, vaan pikemminkin yritetään ennustaa. Leanissa tähdennetään kyllä suunnittelemisen tärkeyttä, mutta aikaa ei tulisi haaskata tarkkojen suunnitelmien tekemiseen

pelkkien ennakkoarvioiden perusteella. Lean- tai agile-lähestymistavassa ei siis ole tarkoitus pyrkiä dokumentittomuuteen, mutta yksityiskohtiin ei tule mennä liian aikaisessa vaiheessa. (Poppendieck & Poppendieck 2003, 50-56)

Tämän jälkeen aloitetaan ensimmäinen iteraatio. Iteraatio alkaa aina suunnittelupalaverilla, jossa asiakas ensin valitsee vaatimusluettelosta muutamia tärkeimpinä pitämiään ominaisuuksia. Seuraavaksi projektitiimi arvioi noiden ominaisuuksien toteuttamiseen menevän ajan. Näiden tietojen perusteella päätetään seuraavan iteraation sisältö. Tällä tavalla työskenneltäessä asiakkaan näkökulmasta tärkeimmät ominaisuudet tulevat toteutetuksi ensin. (Poppendieck & Poppendieck 2003, 29)

Iteraation aikana suunnitellaan, toteutetaan sekä testataan kyseisen iteraation sisältämät ominaisuudet. Lean-projektissa ei koskaan toteuteta muuta kuin ne ominaisuudet, mitä iteraation sisällössä on mainittu. Mitään ylimääräistä, mitä ”todennäköisesti” tullaan tarvitsemaan jatkossa, ei toteuteta ”kaiken varalta”, vaan keskitytään ainoastaan vaadittuihin ominaisuuksiin. Lean ottaa kantaa myös ohjelmointiteknisiin seikkoihin. Työskennellessä tulisi jatkuvasti kiinnittää huomiota testattavuuteen, ja yksikkötestejä (unit tests) tulisi kirjoittaa paljon, jotta kaikki vanha toiminnallisuus olisi helposti testattavissa aina kun uutta lisätään. Lisäksi eri ohjelmakoodin osia ei pitäisi suunnitella siten, että ne on aina ajettava samassa järjestyksessä (sequential programming). Modulaarista ohjelmointitapaa tulisi suosia, jotta jonkin ohjelmiston osan poistaminen tai korvaaminen helpottuisi. Ylläpidon helpottamiseksi samaa asiaa ei saa esiintyä koodissa useammassa kuin yhdessä paikassa ja rajapinnat tulee suunnitella huolellisesti, jotta niitä ei tarvitse muuttaa projektin kestäessä. (Poppendieck & Poppendieck 2003, 29, 58-60)

Mikäli projektitiimin työmääräarviot eivät jostain syystä osu kohdalleen, eikä kaikkea iteraatiossa toteutettaviksi suunniteltuja ominaisuuksia saadakaan valmiiksi, tulee projektitiimin toteuttaa mieluummin muutama toivottu ominaisuus kokonaan, kuin jättää kaikki ominaisuudet puolitiehen. (Poppendieck & Poppendieck 2003, 29)

Kun iteraatio on valmis, sen tulokset esitellään asiakkaalle. Asiakas antaa palautetta, jonka perusteella päätetään, onko muutoksille aihetta. Jos muutoksille on aihetta tai asiakkaalle on tullut mieleen uusia vaatimuksia lopulliselle ohjelmistolle, päivitetään vaatimusluettelo. (Poppendieck & Poppendieck 2003, 29)

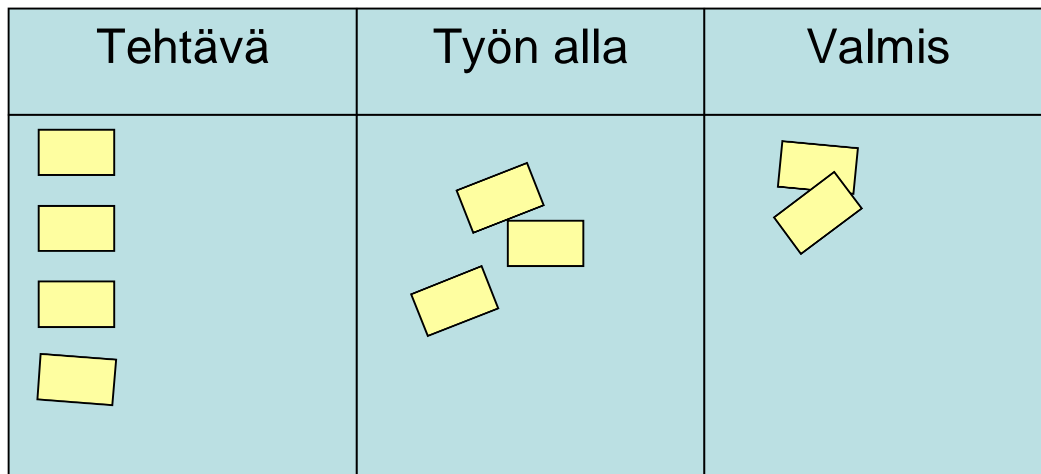
Seuraavan iteraation aluksi päätetään jälleen asiakkaan kanssa, mitä ominaisuuksia tullaan toteuttamaan seuraavassa iteraatiossa. Iteraatiosykliä toistetaan kunnes tuote on asiakkaan hyväksymällä tasolla.

2.3.3 Kanban-ajattelu

Leanin ajattelumalli perustuu kanban-menetelmään, joka on Japanista lähtöisin oleva asiakasvetoinen tuotantomenetelmä. Yksinkertainen esimerkki kanban-menetelmästä on tehdas-toimittaja-kauppa-ketju:

1) Asiakas ostaa tavaran, myyjä ottaa tuotteen hyllystä ja laittaa tilalle kanban-kortin.
2) Kanban-kortit kerätään hyllyistä ja välitetään tavaratoimittajalle
3) Tavaratoimittaja ottaa varastostaan kanban-korteissa mainittujen tuotteiden kokoamiseen tarvittavat osat, ja laittaa tilalle kanban-kortteja.
4) Kanban-kortit kerätään ja välitetään tehtaalle
5) Tehtaalla valmistetaan pyydetyt kanban-korteissa mainitut osat

Leanissa kanban-ajattelua sovelletaan ohjelmistokehitykseen. Iteraation alussa asiakas valitsee toteutettavat ominaisuudet – eli äskeisen esimerkin mukaisesti asiakas ottaa tuotteen hyllystä, kanban-kortti asetetaan tilalle. Toteutettavat ominaisuudet kirjataan lapuille, jotka asetetaan erilliselle tehtävätaululle (kuvio 3) sopiviin lokeroihin, eli toimitaan aivan kuin kanban-kortteja kerätessä. Tehtävätaulussa on sarakkeet ”tehtävä”, ”työn alla” sekä ”valmis”.



KUVIO 3 *Tehtävätaulu*

Taululle kiinnitetään iteraation alussa lappu jokaista iteraation aikana toteutettavaa ominaisuutta kohti. Kun joku projektitiimistä ottaa ominaisuuden tehtäväkseen, hän siirtää lapun kohtaan ”työn alla”, ja vastaavasti kohtaan ”valmis”, kun on saanut ominaisuuden toteutettua. Tehtävätaulu on hyvä keino tehtävien jakamiseksi sekä selkeä tapa nähdä yhdellä silmäyksellä iteraation tilanne. (Poppendieck & Poppendieck 2003, 29, 72-75)

2.4 Yleistä agile-menetelmistä

17 Lean-ajatteluun perustuvien, silloin kevytmenetelmiksi (light-weight methods) kutsuttujen ohjelmistokehitysmallien edustajaa kokoontui vuonna 2001 Utahin Snowbirdissä Yhdysvalloissa tutkiakseen löytyisikö eri kevytmenetelmistä jotain yhteistä. Mukana oli edustajia Extreme Programmingista(XP), Scrumista, Adaptive Software Developmentista (ASD), Crystal-menetelmistä, Feature-Driven Developmentista (FDD) sekä Dynamic Systems Development Methodista (DSDM). Osallistujat pääsivät joissakin yleisen tason asioissa yhteisymmärrykseen ja perustivat *Agile Alliance*-nimisen kaupallista hyötyä tavoittelemattoman (nk. non-profit) järjestön. (Cockburn 2007, 270, 369).

Agile-menetelmien perustaksi Agile Alliance kirjasi 12 periaatetta sisältävän normiston, *Agile Manifestin*, sekä neljä toimivan ohjelmistoprosessin ydinarvoa (engl. core values). Näiden periaatteiden mukaisesti toimivia menetelmiä päätettiin kutsua *agile-menetelmiksi* (suom. *ketterät menetelmät*). (Cockburn 2007, 370)

Agile-menetelmien ydinarvot ovat ”Yksilöt ja kanssakäyminen on tärkeämpää kuin prosessit ja työkalut”, ”Toimiva ohjelmakoodi on tärkeämpää kuin kattava dokumentaatio”, ”Yhteistyö asiakkaan kanssa on tärkeämpää kuin sopimusneuvottelut” sekä ”Muutoksiin reagoiminen on tärkeämpää kuin suunnitelman mukaan toimiminen”. (Cockburn 2007, 370-372).

Agile Manifestissa (LIITE 1) luetelluista periaatteista neljä ensimmäistä teroittavat toimivan ohjelmakoodin nopean toimittamisen tärkeyttä. Nopean muuttuviin vaatimuksiin reagoimisen mahdollistamiseksi suositellaan iteratiivista ja inkrementaalista toimintatapaa, jossa seuraavan iteraation sisältö suunnitellaan edellisen päätyttyä. Lisäksi, koska jokaisen iteraation tuloksena on synnyttävä toimivaa ja ajettavissa olevaa ohjelmistoa, on asiakkaan mahdollista huomata jo aikaisessa vaiheessa, onko ohjelmistosta tulossa sellainen, mitä oli suunniteltu. (Cockburn 2007 373-374)

Seuraavat neljä periaatetta sekä kymmenes periaate keskittyvät ihmislähtöisyyden sekä yhteistyön merkityksen korostamiseen. Projektin menestymisen muistutetaan perustuvan ennen kaikkea pätevien ja motivoituneiden ihmisten työpanokseen. Itseohjautuvat tiimit, joilla on riittävästi valtuuksia oman työnsä tekemiseen tehokkaasti, ovat edellytys onnistuneelle ohjelmistoprojektille. (Cockburn 2007, 375-376).

Jäljellä olevista kolmesta periaatteista kaksi muistuttavat, että tiimien tulisi arvioida omaa työskentelyään säännöllisin väliajoin tehostaakseen toimintaansa ja kolmas varoittaa ylimääräisen työn vaaroista. (Cockburn 2007, 376-377).

Kuten näistä periaatteista on huomattavissa, agile-menetelmät ovat kokoelma kokeneiden ohjelmistosuunnittelijoiden käyttämiä ja hyväksi toteamia menetelmiä (engl. *best practises*). Jokaisen agile-menetelmän jokainen osa on jäljitettävissä aina 1960-luvulle saakka – jotkut jopa 1950-luvulle – mutta menetelmien käyttäjät eivät vain aiemmin ole kirjanneet työskentelytapojaan mihinkään. Menetelmät ovat siis jollain tasolla olleet olemassa jo kauan, mutta niille ei ole ollut todellista markkinatarvetta ennen kuin 1990-luvulla, jolloin IT-ala ja teknologia alkoivat kehittyä niin huimaa vauhtia, ettei perinteisillä menetelmillä enää ehditty ajoissa markkinoille (Kroll&Kruchten 2004, 52; Cockburn 2007, 260-261).

Agile-menetelmät ovat kuitenkin vielä hyvin tuoreita, eikä niiden tehokkuutta siten ole vielä ole tutkittu riittävästi luotettavien johtopäätöksien tekemiseksi niiden tehokkuuden suhteen. Menetelmiä on sovellettu menestyksekkäästi, mutta toisaalta niiden edellyttämän radikaalin ajattelu- ja työskentelytapamuutoksen vuoksi useilla yrityksillä on ollut vaikeuksia menetelmien käyttöönottamisessa. (Kroll&Kruchten 2004, 52)

3. AGILE-MENETELMÄT

3.1 Scrum

Extreme Programmingin ohella yleisin agile-menetelmistä on jonkin verran myös projektinhallinnallisia piirteitä määrittelevä Scrum. Scrum-prosessi seuraa hyvin tarkasti Lean-prosessia sekä Leanin periaatteita: ei ylimääräisiä vaihetuotteita, aluksi korkean tason suunnitelma tehtävästä ohjelmistosta, sen asiakkaan kanssa työstetty priorisoitu ominaisuusluettelo ja viimein itse ominaisuudet kuukauden kerrallaan kestävissä iteraatioissa tai *Sprinteissä*, kuten niitä Scrumissa kutsutaan. Scrumissa käytetään usein myös kanban-tyyppistä tehtävätaulua. (Schwaber 2004, 6, 133-139)

3.1.1 Scrumin vaihetuotteet

Scrumissa määritellyt vaihetuotteet ovat Product Backlog, Sprint Backlog, Burndown Chart sekä asiakkaalle lisäarvoa tuova inkrementti lopullisesta tuotteesta (Schwaber, 2004, 9-14)

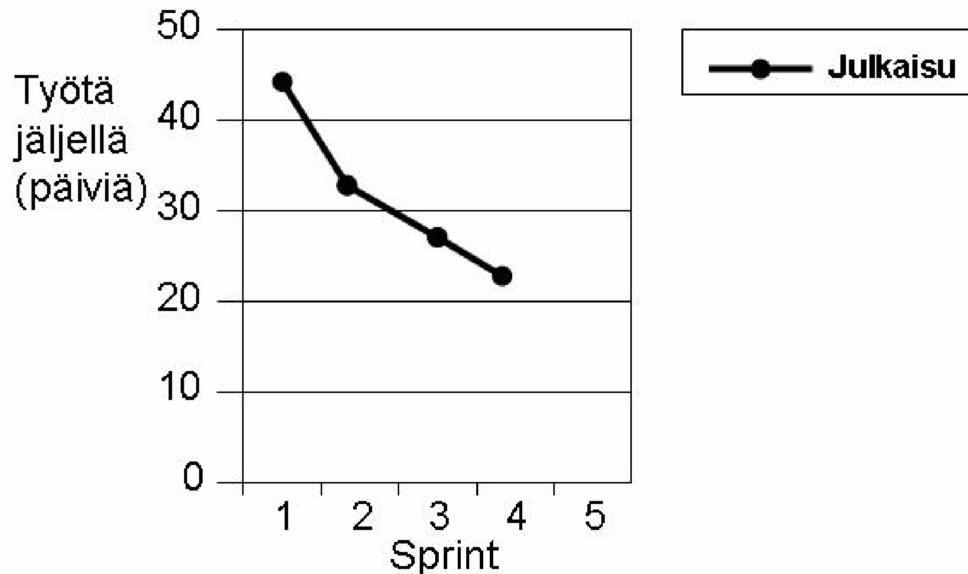
Product Backlog on luettelo kaikista lopullisen tuotteen vaatimuksista. Tärkeä lähtökohta Scrumissa on, että asiakkaalle pyritään tuottamaan lisäarvoa mahdollisimman aikaisin, ja tähän tavoitteeseen pyritään pääsemään toteuttamalla asiakkaan mielestä tärkeimmät ominaisuudet ensin. Tästä syystä vaatimusten on ehdottomasti oltava Product Backlogissa tärkeysjärjestyksessä. (Schwaber 2004, 10-11)

Product Backlogista (kuvio 4) ilmenee jokaisen vaatimuksen alkuperäinen kestoarvio (päivinä) sekä sprinteittäin jaoteltuna jäljellä oleva kesto. (Schwaber 2004, 11)

Tuotteen nimi	
Vaatimus	**kesto (d)**
Alkutoimet	3
Ensimmäinen ominaisuus	9
Toinen ominaisuus	5
Kolmas ominaisuus	3
SPRINT 1	**20**
Neljäs ominaisuus	12
Viides ominaisuus	6
Kuudes ominaisuus	3
SPRINT 2	**21**
Seitsemäs ominaisuus	8
Kahdeksas ominaisuus	8
Yhdeksäs ominaisuus	4
SPRINT 3	**20**

KUVIO 4 *Product Backlog.*

Jäljellä olevien kestojen summan sekä sprintien suunnitellun lukumäärän perusteella pidetään yllä *Burndown Chartia* (kuvio 5), eli kuvaajaa, josta nähdään jäljellä olevan työn suhde jäljellä olevaan aikaan. (Schwaber 2004, 11)



KUVIO 5 *Burndown Chart* (Schwaber 2004, 12)

Burndown Chartista nähdään helposti yhdellä silmäyksellä kuinka nopeasti jäljellä oleva työmäärä vähenee sprintien edetessä. Kaavion perusteella voi esimerkiksi tehdä johtopäätöksiä siitä, saadaanko kaikki jäljellä oleva työ valmiiksi tiettyyn määräaikaan mennessä. Jos arvioidaan, että ei saada, on joko valittava jokin

muu tavoitepäivämäärä tai poistettava Product Ownerin johdolla joitakin vaatimuksia Product Backlogilta, luonnollisesti tärkeysjärjestyksen loppupäästä. Scrumissa on lähdetty ajatuksesta, että projektin valmistuminen ajallaan on tärkeämpää kuin kaikkien esille tulleiden vaatimusten toteuttaminen. Tämä saattaa aluksi kuulostaa jopa sopimusrikkomukselta, mutta näin ei kuitenkaan ole, sillä vastuu Product Backlogista on Product Ownerilla, joka siis on asiakkaan edustaja. Jos Product Owner suostuu pudottamaan vaatimuksia Product Backlogilta, voidaan turvallisesti olettaa, etteivät ne ole erityisen kriittisiä ominaisuuksia – ja ovat näin ollen Lean-ajattelun mukaisesti jätettä. (Schwaber 2004, 11)

Sprint Backlog (kuvio 6) on luettelo tehtävistä, jotka tullaan tekemään seuraavan Sprintin aikana. Sprint Backlogin sisältö määritellään siten, että Product Backlogilta valitaan niin monta tärkeintä vaatimusta, kuin yhden sprintin aikana arvioidaan saatavan tehdyksi. Product Backlogin vaatimukset pilkotaan 4-16 työtunnin mittaisiksi tehtäviksi ennen niiden lisäämistä Sprint Backlogille. (Schwaber 2004, 12)

			Työtä jäljellä (tuntia)						
Task	**Vastuu**	**Tila**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Tehtävä1	Seppo	Valmis	16	9	0				
Tehtävä2	Seppo	Käynnissä	10	4	1	1	1	1	1
Tehtävä3	Timo	Valmis	6	0					
Tehtävä4	Timo	Ei aloitettu	12	12	12	12	12	12	12
Tehtävä5	Pekka	Valmis	5	0					
Tehtävä6	Timo	Ei aloitettu	6	6	6	6	6	6	6
Tehtävä7	Pekka	Käynnissä	9	4	2	2	2	2	2
Tehtävä8	Jussi	Valmis	7	0					
Tehtävä9	Jussi	Käynnissä	7	7	5	5	5	5	5
Tehtävä10	Pekka	Ei aloitettu	4	4	4	4	4	4	4
Tehtävä11	Timo	Käynnissä	7	7	4	4	4	4	4
Tehtävä12	Seppo	Ei aloitettu	3	3	3	3	3	3	3

KUVIO 6 *Sprint Backlog* (Schwaber 2004, 13)

Sprint Backlogista ilmenee jokaisen tehtävän vastuuhenkilö, tehtävän tila (ei aloitettu, kesken, valmis) sekä arvioitu jäljellä oleva kesto, jota päivitetään joka päivä. (Schwaber 2004, 13-14).

Scrumissa jokaisen Sprintin täytyy tuottaa ajettava ja toimiva sovellus, joka pitää sisällään lopullisen tuotteen toiminnallisuutta. Tämän sovelluksen on lisäksi oltava myytävissä oleva tuote, joka tuo asiakkaalle lisäarvoa ja sisältää osan lopullisen ohjelmiston vaatimuksista. (Schwaber 2004, 12)

3.1.2 Scrum-roolit

Tärkeimmät roolit Scrum-projektissa ovat Product Owner, Scrum Master sekä projektitiimi. (Schwaber 2004, 6-7)

Product Owner on asiakkaan edustaja, joka tuottaa ensimmäisen version Product Backlogista. Scrumissa vaatimuksia voidaan lisätä, poistaa tai muuttaa projektin kestäessä jokaisen iteraatiojakson jälkeen. Product Owner vastaa siitä, että lopullisen tuotteen vaatimukset sekä niiden tärkeysjärjestys ovat koko ajan projektiryhmän tiedossa. (Schwaber 2004, 6-7)

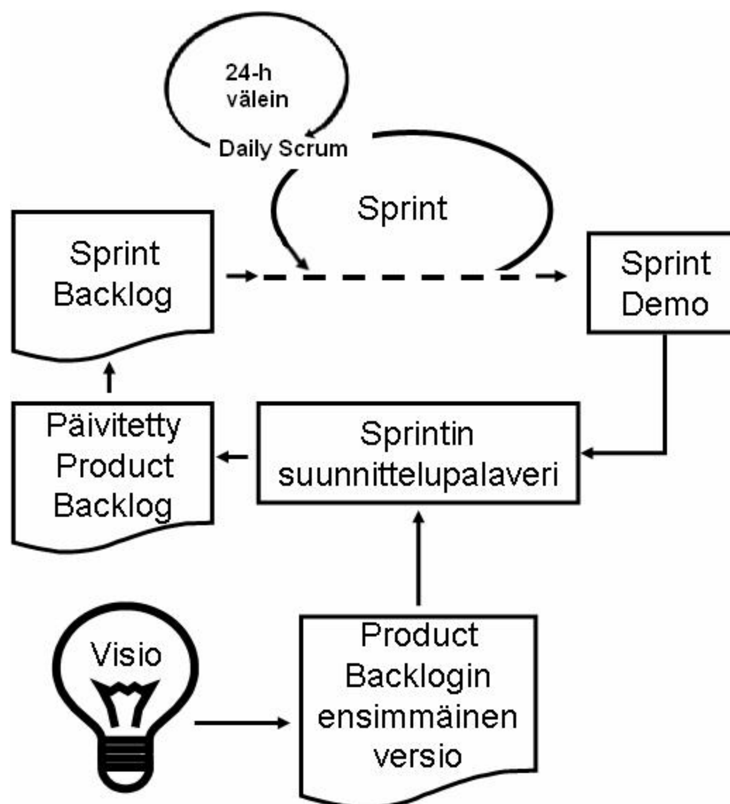
Projektitiimin tehtävänä on tuottaa Product Backlogissa mainituista ominaisuuksista toimiva kokonaisuus. Scrum-projektissa ei ole projektipäällikköä, vaan tarkoituksena on, että tiimit ovat itseohjautuvia, ja niillä on itsenäisen työn edellyttämät valtuudet päättää omista asioistaan. Muuntautuminen johdetusta työryhmästä itseohjautuvaksi tiimiksi on haasteellista, mutta Scrumin onnistumisen kannalta myös välttämätöntä. Projektitiimissä täytyy vallita kollektiivinen vastuu kaikesta tekemisestä. Jokaisen tiiminjäsenen on Scrumissa oltava moniosaaja, ”*henkilön ei tarvitse olla testaaja testatakseen eikä suunnittelija suunnitellakseen*”. (Schwaber 2004, 6-7, 101-102, 104)

Scrum Masterin tehtävänä on valvoa, että Scrum-menetelmää noudatetaan oikein sekä neuvoa tarvittaessa projektitiimiä Scrumiin liittyvissä seikoissa. Pelkkä prosessipoliisi Scrum Master ei kuitenkaan ole, vaan hänellä on myös tärkeä tehtävä projektitiimin työrauhan varmistajana. Scrum Master pyrkii aktiivisesti poistamaan esteitä projektitiimin tieltä sekä tehostamaan tiimin toimintaa. Scrum Master ei kuitenkaan missään tapauksessa ole

projektipäällikkö, rooli on lähempänä toiminnan kehittäjän tai laatupäällikön tehtävää. Kokeneiden projektipäälliköiden onkin usein vaikea muuntautua Scrum Mastereiksi, koska he ovat tottuneet kantamaan vastuun projekteista, jakamaan tehtäviä sekä ottamaan suuren roolin projektin läpiviennissä, ja siksi ajatus itseohjautuvasta tiimistä on vaikea sisäistää. Päinvastoin kuin projektipäälliköllä, Scrum Masterilla ei ole minkäänlaista käskyvaltaa projektitiimin suhteen. (Schwaber 2004, 28, 36)

3.1.3 Scrum-prosessi

Scrum-prosessi (kuvio 7) alkaa yleisen tason visiosta, jonka Product Owner koostaa kyseisen vision toteuttamiseen tähtääväksi suunnitelmaksi. Tuo suunnitelma pitää sisällään myös ensimmäisen version Product Backlogista.



KUVIO 7 *Scrum-prosessi (Schwaber 2004, 9)*

Product Backlogissa mainitut ominaisuudet priorisoidaan siten, että eniten lisäarvoa asiakkaalle tuovat ominaisuudet ovat listan kärkipäässä. Tämän jälkeen

ominaisuudet jaetaan alustavasti julkaisuihin. Tässä vaiheessa syntynyt Product Backlog tai jako julkaisuihin ei ole lopullinen, vaan niitä joudutaan usein muuttamaan projektin kestäessä. Julkaisuun vaaditut ominaisuudet toteutetaan yhdessä tai useammassa 30 kalenteripäivää kestävässä iteraatiojaksossa, eli Sprintissä. (Schwaber 2004, 7-8).

Sprint alkaa suunnittelupalaverilla. Suunnittelupalaverissa Product Owner sekä projektitiimi päättävät yhdessä, mitä ominaisuuksia tullaan toteuttamaan seuraavan Sprintin aikana. Käytännössä Product Owner valitsee muutamia ominaisuuksia Product Backlogin kärkipäästä ja projektitiimi kertoo, mitkä niistä voidaan toteuttaa Sprintin aikana. Nämä ominaisuudet siirretään Sprintin tehtävälistalle, eli Sprint Backlogille. Suunnittelupalaveri kestää (enintään) kahdeksan tuntia. (Schwaber 2004, 8)

Kun suunnittelupalaveri on pidetty, projektitiimi aloittaa välittömästi ominaisuuksien toteuttamisen Sprint Backlogin mukaisesti. Joka päivä projektitiimi sekä Scrum Master pitävät enintään 15 minuuttia kestävän Daily Scrum-palaverin, jossa kukin projektitiimin jäsen kertoo, mitä on parhaillaan tekemässä, mitä aikoo tehdä seuraavaksi sekä mitä ongelmia on mahdollisesti tullut vastaan. Kyseessä ei kuitenkaan ole kehumiskilpailu, missä kukin kertoo kaiken mitä on koko uransa aikana tehnyt, vaan tavoitteena on tiedon tasaaminen. Scrum Master ei ole paikalla perinteisen projektipäällikön ominaisuudessa, eikä Daily Scrumin tarkoitus missään tapauksessa ole raportoida Scrum Masterilla tiiminjäsenten tekemisistä. Scrum Master on vain sivustaseuraaja, ja on paikalla ainoastaan siltä varalta, että projektitiimi on sattumalta törmännyt johonkin työskentelyä haittaavaan esteeseen, jonka poistamisessa hän voisi olla avuksi. Palaverin tarkoituksena on siis saattaa kaikki mahdolliset ongelmat kaikkien tietoon sekä toisaalta ehkäistä tilanteita, joissa useampi projektiryhmäläinen tekee samaa asiaa toisistaan tietämättä. (Schwaber 2004, 8, 27-28, 99, 103-104, 135)

Jokaisen Sprintin on tuotettava toimivaa, integroitavissa olevaa ohjelmakoodia. Sprintin lopuksi pidetään aina Sprint Review-palaveri (joissain lähteissä tästä

käytetään nimitystä Sprint Demo), jossa projektitiimi esittelee aikaansaannoksensa Product Ownerille sekä asiasta kiinnostuneille sidosryhmille. Ainoastaan täysin valmiit ominaisuudet integroidaan ja esitellään, kaikki keskeneräinen työ siirretään seuraavan Sprintin Backlogille. Palaverin tarkoituksena on varmistaa, että projektitiimi on ymmärtänyt annetun tehtävän oikein ja että tuote vastaa Product Ownerin näkemystä. Sprint Review-palaveri kestää neljä tuntia, ja projektitiimi käyttää noin tunnin sen valmistelemiseen. (Schwaber 2004, 9, 137-138).

Sprint Review-palaverin jälkeen, ennen seuraavan Sprintin suunnittelupalaveria Scrum Master pitää projektitiimin kanssa Sprintin arviointipalaverin, jossa tiimi arvioi käytettyjä menetelmiä sekä prosessia. Mikäli menetelmissä tai prosessissa on ilmennyt jotain, mikä ei palvele tarkoitustaan, sitä pyritään muuttamaan tai se poistetaan. Palaverin kesto on kolme tuntia, ja sen tavoitteena on kehittää tiimin toimintaa seuraavia Sprintejä ajatellen. (Schwaber 2004, 9, 138-139).

Tämän jälkeen sykli aloitetaan alusta pitämällä uusi Sprintin suunnittelupalaveri. Sykliä jatketaan, kunnes Product Owner katsoo, että riittävästi ominaisuuksia on toteutettu tai kaikki Product Backlogissa listatut ominaisuudet on toteutettu.

3.1.4 Sprintien tuotokset

Scrumissa jokaisen Sprintin tuloksena syntyy aina toimivaa ohjelmistoa ja koska ominaisuudet toteutetaan Lean-ajattelun mukaisesti tärkeysjärjestyksessä, on tuotteella todellista arvoa asiakkaalle jo aikaisessa vaiheessa. Scrumin puolestapuhujat lupaavat jopa myytävissä olevaa ohjelmistoa jokaisen Sprintin tuloksena (Schwaber 2004, 12). Kuitenkaan missään löytämässäni case-kuvauksessa tähän tavoitteiseen ei ole vielä ensimmäisten Sprintien aikana päästy (esimerkiksi Schwaber 2004, 19-20, 42, 57, 63, 64-65). Schwaber (2004, 69) on asian suhteen ristiriitainen: toisaalta hän teroittaa, että jokaisen Sprintin tuloksena on synnyttävä myytävissä oleva tuote, mutta vaatii toisaalta, että projektitiimin, Product Ownerin sekä sidosryhmien ”on kyettävä näkemään

projekti sarjana Sprintejä, jotka johtavat julkaisuun”. Jos julkaisuun päästään vasta sarjan Sprintejä jälkeen, ei myytävissä olevaa tuotetta ole syntynyt *jokaisen* Sprintin jälkeen, vaan ainoastaan sarjan viimeisen Sprintin jälkeen. Useimmissa muissa agile-menetelmissä luvataan vain tuottaa toimiva, integroitavissa oleva ohjelmiston osa jokaisen iteraation tuloksena (esimerkiksi Astels et al. 2002, 77, 181), mikä on mielestäni uskottavampi tavoite.

3.1.5 Skaalaus

Mikäli projektiin osallistuu henkilöitä eri paikkakunnilta tai projektissa on suuri määrä työntekijöitä, vaikeutuu kommunikaatio sekä tiedon jakaminen tiiminjäsenten kesken merkittävästi. Tällaista projektia ei kannata yrittää toteuttaa yhtenä Scrum-projektina, vaan viisainta on muodostaa useita Scrum-tiimejä, jotka pyörittävät kukin omaa Scrum-osaprojektiaan. Osaprojekteja hallitaan yhdellä suuremmalla Scrum-projektilla, jonka Scrum-palavereihin osallistuu vain yksi edustaja jokaisesta Scrum-tiimistä, ja asiat käsitellään noissa palavereissa tiimitason Scrumeja korkeammalla tasolla. (Schwaber 2004, 123-124)

Suuremman Scrum-projektin pystyttäminen vaatii usein paljon käytännön järjestelyjä. Tällöin Product Backlogille voidaan ottaa tehtäviä, mitkä eivät ole mitään lopullisen ohjelmiston osia, vaan Scrum-prosessin hallintaan liittyviä käytännön tehtäviä (Schwaber 2004, 120).

Scrum on alun perin suunniteltu henkilömäärältään pienehköille projekteille, mutta soveltuu siis myös laajempiin projekteihin, joskin vaatii tuolloin toimiakseen erityisjärjestelyjä.

3.1.6 Testaaminen

Vaikka Scrum pyrkii olemaan kattava prosessimalli, jättää se esimerkiksi toimintotestauksen täysin huomiotta. Jokaisessa Sprintissä otetaan mukaan uutta toiminnallisuutta, jonka on oltava ”huolellisesti testattua”, mutta yksityiskohtaisemmin testauspuolta ei kuvata, eikä vanhojen ominaisuuksien

regressiotestauksesta puhuta mitään (Schwaber 2004, 12-13). Ilmeisesti oletetaan, että kaikki toteutusvaiheessa tehdyt testit suunnitellaan kattaviksi ja ne tehdään automaattisiksi, jolloin voidaan ajaa koko järjestelmän kattavia unit testejä ajastettuna, tarvittaessa vaikka joka yö. Kattavien testien suunnitteleminen vie kuitenkin aikaa ja vaatii erityistä testausosaamista. Schwaber (2004, 104) kirjoittaa, ettei henkilön ”tarvitse olla testaaja testatakseen”. Tästä olen osittain samaa mieltä: kuka tahansa tiiminjäsen voi varmasti ajaa tai jopa kirjoittaa testejä, mutta henkilön on oltava testaaja tai hänen on ainakin omattava paljon kokemusta testaamisesta *suunnitellakseen* todella kattavan testauksen. Mikäli erillistä toimintotestausta ei tehdä, on projektitiimissä siis oltava mukana ainakin yksi testauksen todellinen asiantuntija, tai lopullisen tuotteen laadusta ei ole takeita.

3.1.7 Scrumin käyttökokemuksia

Etsiessäni teoriatietoa tämän tutkimuksen perustaksi, havaitsin, että Scrumista on huomattavasti helpompaa löytää tietoa kuin muista agile-menetelmistä. Tämän uskon olevan yksi syy Scrumin suureen suosioon. Menetelmän suosiota kasvattaa varmasti myös se, että se tarjoaa johdolle hyvän näkyvyyden projektin etenemisestä Burndown-kaavioiden sekä Backlogien ”työtä jäljellä”-kentän muodossa. Sprint Backlogin ”Työtä jäljellä”-kentän käytännön hyödyllisyys kuitenkin epäilyttää. Koska työntekijät varmastikin joka tapauksessa raportoivat kaikki tehdyt työtunnit johonkin työaikaraportointijärjestelmään, on vaikkapa taulukkolaskentaohjelmaan helppo tehdä laskukaava, jolla lasketaan jäljellä oleva työmäärä vähentämällä tehdyt työtunnit alkuperäisestä arviosta. Näin ollen, tuodakseen jotain lisäarvoa, pitäisi Backlogin joka päivä päivitettävä ”työtä jäljellä”-arvion olla kyseisen laskukaavan tulosta tarkempi. Tarkan arvion tekeminen päivittäin veisi kuitenkin suhteettoman paljon aikaa, joten epäilen, että projektitiimi käytännössä ”arvioi” jäljellä olevan keston joko nimenomaan laskemalla sen edellä mainitun laskukaavan mukaisesti tai antamalla nopeasti jonkin mihinkään perustumattoman vakiovastauksen. Näissä tapauksissa arvio ei tuo lisäarvoa, vaan on pelkkää hukkaan heitettyä työaikaa, eli mainio esimerkki

Lean-ajattelun määrittelemästä jätteestä. Tästä syystä Sprint Backlogin käyttämiseen jäljellä olevan keston arvioimiseen ei kannata panostaa, vaan Backlog kannattaa yksinkertaistaa pelkäksi projektitiimin tehtävälistaksi, josta tiiminjäsenet näkevät, mitä ominaisuuksia Sprintin aikana tulee tehdä, niiden ajankohtaiset tilat sekä vastuuhenkilöt.

Käytännön kokemukset Scrumista näyttävät jakautuvat kahteen: onnistujiin, sekä niihin, jotka eivät käyttäneet Scrumia kaikilta osin oikein ja ovat siksi epäonnistuneet. Merkille pantavaa on se, että en löytänyt kuvauksia epäonnistuneista projekteista, joissa on todella käytetty Scrumia, mutta löysin useita esimerkkejä epäonnistuneista projekteista, joissa oli käytetty ”sovellettua Scrumia” (Yegge 2006; Liite 2). Kaikissa näissä tapauksissa projektiryhmä oli jättänyt joitakin Scrumin osia pois korvaamatta kyseisen prosessin osan jättämää puutetta mitenkään. Lähestymistapa on siis ollut Scrumin muokkaaminen siten, että vanhoja toimintatapoja ei tarvitsisi muuttaa. Tämä on tietenkin nurinkurinen lähestymistapa. Tarkoitushan on nimenomaan tehdä asiat uudella tavalla ja saavuttaa tuloksia sitä kautta. Toimintatapoja pitäisi siis muuttaa Scrumin mukaisiksi, eikä toisin päin. Mikäli toimintatapoja ei ole mahdollista muuttaa Scrumin mukaisiksi, esimerkiksi koska prosessia tai sen vaihetuotteita ei saa ulkoisista syistä muuttaa, ei Scrumia näihin esimerkkeihin vedoten kannata edes yrittää soveltaa.

3.1.8 Yhteenveto ja arviointi

Scrumin suurimmat hyödyt ovat itsenäisten tiimien toteuttaman ja asiakkaan aktiivisen osallistumisen ohjaaman iteratiivisen ja inkrementaalisen työskentelytavan tuomia etuja. Scrum vaikuttaakin rakennetun näiden prosessin osien luomalle perustalle, ja muut määritellyt vaiheet, vaihetuotteet sekä toiminnot on määritelty lähinnä tukemaan tai tehostamaan näitä prosessin osia. Esimerkiksi Daily Scrumit sekä tiimin jäsenten työpisteiden sijoittaminen lähelle toisiaan tehostavat tiimin toimintaa ja mahdollistavat itseohjautuvuuden, Scrum Masterin rooli on määritelty, jotteivät perinteiseen työskentelytapaan tottuneet

projektipäälliköt puuttuisi liiaksi tiimin toimintaan, Product Backlog mahdollistaa tärkeimpien asioiden toteuttamisen ensin ja niin edelleen.

Scrum vaikuttaa soveltuvan hyvin projekteihin, joissa toimitaan pienissä tiimeissä, jotka työskentelevät samoissa tiloissa. Mikäli osa projektiin osallistuvista on eri paikkakunnalla tai jopa eri aikavyöhykkeellä, vaikeutuu tiedonjako sekä kommunikointi niin paljon, että Scrumin käyttämisen asianmukaisuutta on syytä harkita tarkoin. Scrumia on kuitenkin käytetty myös suuremmissa projekteissa luvussa 3.1.5 kuvattua skaalausmenetelmää hyödyntäen. (Schwaber 120-121, 124-126)

Scrumin käyttöönoton edellytys on, että ohjelmistoprosessi on mahdollista muokata täysin Scrumin mukaiseksi. Mikäli tämä ei ole mahdollista, ei Scrumia ole järkevää ottaa käyttöön.

Menetelmän epäilyttävimpänä puolena pidän palaverien määrää suhteessa tehokkaaseen työaikaan. Jokaisella 30 päivän jaksolla – josta keskimäärin 21 on työpäiviä – pidetään aina suunnittelupalaveri (8 tuntia), Daily Scrumit (joka päivä 15 min, eli 21 päivässä yli 5 tuntia), Review-palaveri (5 tuntia valmisteluineen) sekä arviointipalaveri (3 tuntia) (Schwaber, 2004, 8-9). Jokaisessa Sprintissä palavereja on siis 21 tuntia. Jos mukana on työntekijä, joka on täydellä työkuormalla mukana kyseisessä projektissa, tekee hän 21 työpäivässä n. 157,5 tuntia töitä projektille. Hänen työajastaan palavereihin kuluu 13 %. Vielä epäilyttävämmältä suhdeluku kuulostaa, kun ajatellaan palvelualalla yleisiä tilanteita, joissa joku henkilö saattaa olla mukana projektissa vaikkapa vain 50 % kuormalla. Tällainen henkilö käyttäisi siis palavereissa istumiseen 21 tuntia kussakin Sprintissä kyseiselle projektille tekemästään n. 79 tunnista, eli reilusti yli neljänneksen työpanoksestaan.

Scrumin yleistasoinen vaatimusluettelo, jota päivitetään projektin kestäessä sekä yleisen tason visio tulevasta järjestelmästä tarkkojen teknisten dokumenttien sijaan säästävät varmasti paljon aikaa projektin alussa ja itse ohjelmointityövaiheeseen päästään perinteistä prosessimallia noudattavaa

projektia nopeammin. Toki teknisten dokumenttien tekemiseen kuluva aika riippuu myös projektista: jos esimerkiksi sidosryhmien työskentelyn mahdollistamiseksi rajapintamäärittelyjä on tehtävä tarkalla tasolla jo projektin alussa, on nuo määrittelyt tehtävä prosessimallista riippumatta. Täsmällisten teknisten dokumenttien puuttumisen hyötyjä ovat säästetty aika sekä se, että dokumenttien puuttuminen ohjaa projektitiimiä tekemään luettavampaa ohjelmakoodia. Aikaa säästyy projektin alussa sekä samojen dokumenttien mittavalta uudelleen kirjoittamiselta välttymisen vuoksi projektin lopussa. Suurin ajansäästö syntyy kuitenkin siitä, että kun ohjelmistoon myöhemmin tehdään jotain muutoksia, on päivitettävää vähemmän. Vain ohjelmakoodi sekä ehkä asiakasdokumentti on tarpeen päivittää, eikä niiden lisäksi useita erilaisia teknisiä dokumentteja. Dokumenttien puuttuminen aiheuttaa kuitenkin ongelmia erityisen mittavissa järjestelmissä, joissa kokonaiskuvaa järjestelmästä on vaikea hahmottaa pelkän ohjelmakoodin perusteella. (Cockburn 219-220)

3.2 Extreme Programming (XP)

Nimensä mukaisesti Extreme Programming (XP) pyrkii kohdentamaan kaiken tekemisen itse ohjelmointityöhön. Ohjelmistoprosessista on Lean-ajattelun mukaisesti pyritty jättämään kaikki ylimääräiset vaiheet sekä vaihetuotteet pois. (Astels, et al. 2002, xvii, xxvi)

3.2.1 XP:n periaatteet

XP perustuu 14 periaatteelle:

1) Asiakkaat mukana projekteissa

XP:ssä asiakkaan edustajan – eli henkilön, joka on yksi tulevan tuotteen käyttäjistä – on työskenneltävä suoraan projektitiimin kanssa. Asiakkaan edustajan on voitava olla sellaisessa asemessa, että hän voi aidosti edustaa kaikkia loppukäyttäjiä. (Astels et al. 2002, 4)

Asiakkaan edustajan tehtävä on vastata projektitiimin esittämiin kysymyksiin, tehdä päätöksiä ominaisuuksien prioriteettien suhteen sekä päättää kunkin ohjelmistojulkaisun sisällöstä. Edustaja myös osallistuu suunnitteluun kirjoittamalla sekä priorisoimalla käyttäjätarinoita. Edustajalta vaaditaan myös jonkin verran teknistä osaamista, sillä XP:ssä asiakkaan edustaja kirjoittaa sekä ajaa hyväksyntätestit (engl. *Acceptance test*) (Astels et al. 2002, 4).

2) Käytä metaforia vaikeiden kokonaisuuksien kuvailemiseen

XP-projekteissa on omintakeinen tapa käyttää kokonaisuuksien kuvailemiseen metaforia. Taustalla tässä käytännössä on ajatus, että jos on vaikea määritellä *mikä* haluttu tuote on, kenties on helpompaa määritellä *millainen* se on. (Astels et al. 2002, 4, 34)

Esimerkkeinä on käytetty mm. seuraavia lauseita:
Ohjelmisto, jota tarvitsen, on kuin arkistointijärjestelmä Thomas Edisonin papereita varten. (Astels et al. 2002, 34)
Käyttäjärajapinta on kuin laskentataulukko. (Astels et al. 2002, 35)

Kumpikaan esimerkeistähän ei tosiasiassa ole metafora, vaan molemmat ovat *vertauksia* (Collins Cobuild 1995, 1044-1045,1554; Nurmi et al. 2001, 273). XP-prosessin vaiheiden kuvausten perusteella vertaus olisikin tässä yhteydessä oikeampi termi, mutta koska metafora on kuitenkin valittu vaihetuotteen viralliseksi nimeksi (Astels et al. 2002, 34; Fowler 2004, 9-10), käytän sekaannusten välttämiseksi kyseistä termiä myös tässä tutkimuksessa.

3) Suunnittelu

Kaikki ohjelmistoprojektit on suunniteltava. Ei ole kuitenkaan tarkoituksen mukaista käyttää kuukausia tai vuosia yksityiskohtaisen suunnitelman tekemiseen, sillä suunnitelmasta tulee joka tapauksessa puutteellinen, ja sitä on projektin kestäessä päivitettävä. (Astels et al. 2002, 5)

XP:ssä suunnitelmia tehdessä lopullinen päätäntävalta on niillä, keillä on paras tietämys kyseisestä aiheesta. Näin ollen liiketaloudellisesta puolesta perillä olevat henkilöt päättävät projektin kokonaistavoitteen rajauksesta, toiminnallisuuksien prioriteeteista ja julkaisujen sisällöistä sekä ajankohdista. Tekniset asiantuntijat taas tekevät työmääräarviot, pohtivat teknisistä päätöksistä koituvia seurauksia, päättävät tiimin organisoitumisesta sekä tekevät yksityiskohtaisemman, tehtävätasoisen, aikataulutuksen. (Astels et al. 2002, 5-6)

4) Lyhyet palaverit

Kokouksissa on usein liikaa käsiteltäviä aiheita, ja samassa kokouksessa halutaan usein sekavasti käsitellä sekä yleisen tason tiedotusasioita, että tärkeitä teknisiä seikkoja (Malik 2002, 241). Tällainen vakiintunut käytäntö onkin johtanut siihen, että ohjelmoijat eivät yleensä pidä kokouksista: ne mielletään usein pitkiksi ja tylsiksi tilaisuuksiksi, mitkä vievät aikaa ”oikeilta töiltä” (Astels et al. 2002, 6).

Kokoukset ovat kuitenkin korvaamaton kommunikointikanava, mutta ainoastaan silloin, kun ne ovat lyhyitä ja keskittyvät rajattuun aihealueeseen. XP:ssä kokouksien pituuteen pyritään vaikuttamaan määräämällä kaikki osallistujat seisomaan. Ajatuksena on, että jos kukaan ei saa istua, eivät kokoukset kestä kauaa ja ihmiset keskittyvät helpommin itse asiaan. (Astels et al. 2002, 6).

Paras ajankohta seisontakokoukselle on joka aamu, alkaen sellaiseen aikaan, että kaikki ovat juuri ehtineet työpaikalle. Kokouksessa kukin osallistuja kertoo mitä teki eilen, mitä aikoo tehdä tänään sekä tuo esille mahdollisia ongelmia, joihin on törmännyt. Kokous vastaa Scrumin Daily Scrumia (Schwaber 2004, 8, 99), eli se ei ole kehumiskilpailu, eikä sen tarkoitus ole tekemisten raportoiminen jollekulle. Tavoitteena on ainoastaan tiedon tasaaminen projektitiimin kesken. (Astels et al. 2002, 6-7)

5) *Testit ennen koodia (Test-first design)*

XP:ssä ominaisuuden toteuttaminen aloitetaan aina kirjoittamalla yksikkötesti (unit test) ja vasta testin kirjoittamisen jälkeen kirjoitetaan ohjelmakoodi, jolla testi saadaan läpäistyä. Yksikkötesti on yksinkertainen testi, joka testaa tulevan ominaisuuden toiminnan. Testien suunnittelemiseen sovelletaan kahta hyvin yksinkertaista sääntöä:

1) *Testaa kaikkea, mikä voi mennä rikki*
2) *Älä testaa mitään, mikä ei voi mennä rikki*

(Astels et al. 2002, 7-8)

Kun testit kirjoitetaan ensin, varmistutaan siitä, että ominaisuus on toteutettu oikein ja ominaisuus tulee varmasti testattua. Lisäksi vältytään pitkäveteiseltä kattavan moduulitestisetinkirjoittamisvaiheelta koko projektin tai inkrementin lopussa. (Astels et al. 2002, 7)

Asiaa tarkemmin ajattelematta voisi olettaa tällaisen ohjelmointitavan lisäävän työmäärää. Näin ei kuitenkaan todellisuudessa ole. Ensinnäkin aloittaessaan ominaisuuden ohjelmoimista, joutuu ohjelmoija joka tapauksessa ainakin ajatuksen tasolla suunnittelemaan, miten uuden toiminnallisuuden toimivuus todennetaan. Ei siis ole suuri vaiva kirjoittaa ajatuksia saman tien ylös käyttäen sitä ohjelmointikieltä, jolla ominaisuuskin tullaan toteuttamaan. Lisäksi pidemmällä aikavälillä aikaa säästyy, kun erillistä moduulitestien suunnitteluvaihetta ei tarvita. (Astels et al. 2002, 7)

Yksikkötestit on myös yleensä helppo automatisoida, jolloin kaikki yksikkötestit voi ajaa vaikkapa jokaisen ohjelmakoodin jäädytyksen jälkeen. Koska yksikkötestejä on tarkoitus ajaa usein, tulee niiden olla lyhyitä ja yksinkertaisia. (Astels et al. 2002, 8)

6) Yksinkertaisin mahdollinen rakenne

Ohjelmakoodin rakenteeksi tulee aina valita Lean-ajattelun mukaisesti yksinkertaisin tämän hetken tarpeisiin soveltuva ratkaisu. Joustavampi rakenne auttaisi kyllä tulevien vaatimusten ottamisessa mukaan, mutta XP:ssä ei rakennetta suunnitella tulevan varalle, vaan rakennetta päivitetään jatkuvasti. (Astels et al. 2002, 8)

7) Pariohjelmointi

Vanha viisaus sanoo, että kaksi päätä on parempi kuin yksi, mutta XP:n mukaan kaksi päätä yhdessä on enemmän kuin kaksi päätä erikseen (Astels et al. 2002, 9).

Kaikki ohjelmointityö tehdään XP:ssä parityönä. Toinen ohjelmoijista – jota kutsutaan *driveriksi* eli ajajaksi – ohjelmoi, ja toinen – *partner* – seuraa toisen työskentelyä sivusta. Astels, Granville ja Novak (2002, 10) tiivistävät pariohjelmoinnin perusajatuksen osuvasti:

The partner can see the forest while the driver sees the trees.

Rooleja vaihdetaan tietyin väliajoin, mutta niitä voi vaihtaa myös silloin, jos partnerilla on mielessään jokin toteutusidea käsillä olevaan ongelmaan. XP:ssä rooleihin suhtaudutaan niin vakavasti, että rooli vaihdetaan usein leikkisästi pyytämällä tai tarjoamalla *ajovuoroa* toiselle (*”Here, you drive!”* tai *”Can I drive?”*). (Astels et al. 2002, 9-10)

8) Sopikaa ohjelmointistandardeista

Tiimit määrittelevät ohjelmointistandardit, joita kaikkien tulee noudattaa. Kun kaikki ohjelmoivat samojen periaatteiden mukaan, helpottuu koodin lukeminen, pariohjelmointi sekä refaktorointi. (Astels et al. 2002, 10-11)

9) Ohjelmakoodin jaettu omistajuus

Joissakin ohjelmistomenetelmissä suositaan esimerkiksi luokkaomistajuuksien määrittelemistä. XP ei tätä menetelmää suosi, sillä jos omistajuus on määritelty, täytyy jokaisen, joka tekee kyseiselle alueelle muutoksia, hyväksyttää muutokset koodin omistajalla. Tämä hidastaa ohjelmistokehitystä tarpeettomasti. Luokkaomistajuus johtaa myös tilanteeseen, jossa jokainen tiiminjäsen tuntee jonkin osan ohjelmistosta todella hyvin, eikä muita lainkaan – eikä kukaan tunne koko järjestelmää. Jaetun ohjelmakoodin omistajuuden edellytys on periaatteen 8 mukainen ohjelmointistandardien määritteleminen ja noudattaminen. (Astels et al. 2002, 10-12)

10) Integroikaa jatkuvasti

Tiimin jäsenten tulee integroida tekemänsä ohjelmakoodi kokonaisjärjestelmään mahdollisimman usein, käytännössä vähintään kerran päivässä sekä jokaisen valmiiksi tulleen ohjelmiston osan jälkeen. Jatkuva integroiminen vähentää ongelmallisia tilanteita, joissa useampi ohjelmoija on sattumalta muokannut samaa ohjelmiston osaa. (Astels et al. 2002, 13)

11) Refaktoroikaa

R*efaktorointi* tarkoittaa ohjelmiston sisäisen rakenteen parantamista siten, että ohjelmiston ulospäin näkyvä toiminta ei muutu. Ohjelmistoa tulee refaktoroida jatkuvasti, jotta arkkitehtuuri säilyisi selkeänä ja yksinkertaisena. (Astels et al. 2002, 13)

12) Julkaisut pienissä inkrementeissä

Pienet julkaisut tuovat osia lopullisesta ohjelmistosta loppukäyttäjän kokeiltavaksi usein ja aikaisessa vaiheessa. Julkaisuvälien tulisi olla yhdestä muutamaan kuukauteen. (Astels et al. 2002, 14)

13) Varokaa loppuun palamista

Kukaan ei työskentele parhaalla mahdollisella teholla stressaantuneena. Loppuun palamisia on helppo estää: lähettäkää työntekijät kotiin, kun kahdeksan tunnin työpäivä on täynnä. (Astels et al. 2002, 14)

14) Hyväksykää muutokset

Projektin alussa kootut vaatimukset vaativat varmasti päivitystä ennen projektin päättymistä. Jos projektiryhmä hyväksyy tämän tosiasian ja varautuu muutoksiin, muutokset eivät enää ole taakka, vaan ne muuttuvat kilpailueduksi. (Astels et al. 2002, 15)

3.2.2 XP:n tiimit ja roolit

XP:ssä työntekijät on jaettu kahteen keskenään yhteistyössä toimivaan tiimiin: asiakastiimiin sekä tuotekehitystiimiin. Tuotekehitystiimejä voi olla useampia. Molempiin tiimeihin on määritelty muutamia rooleja erilaisten hoidettavien tehtävien kuvaamiseksi. (Astels et al. 2002, 18, 26)

Asiakastiimissä on tuotteen loppukäyttäjiä, mutta myös muita rooleja: tarinankertojia, hyväksyjiä, rahoittajat (XP:ssä heitä kutsutaan *Gold Ownereiksi*), suunnittelijoita sekä päällikkö (XP:ssä *Big Boss*). (Astels et al. 2002, 19)

Tarinankertojat ohjaavat tuotekehitystiimin toimintaa. He tuntevat liiketoiminta-alueen ja kirjoittavat projektin vaatimukset eli *käyttäjätarinat*. *Hyväksyjät* kirjoittavat hyväksyntätestit ja varmistavat, että jokainen julkaisu vastaa tarinankertojien tarpeita. Hyväksyjät ovat usein myös tarinankertojia, mutta eivät kuitenkaan aina. (Astels et al. 2002, 19-20)

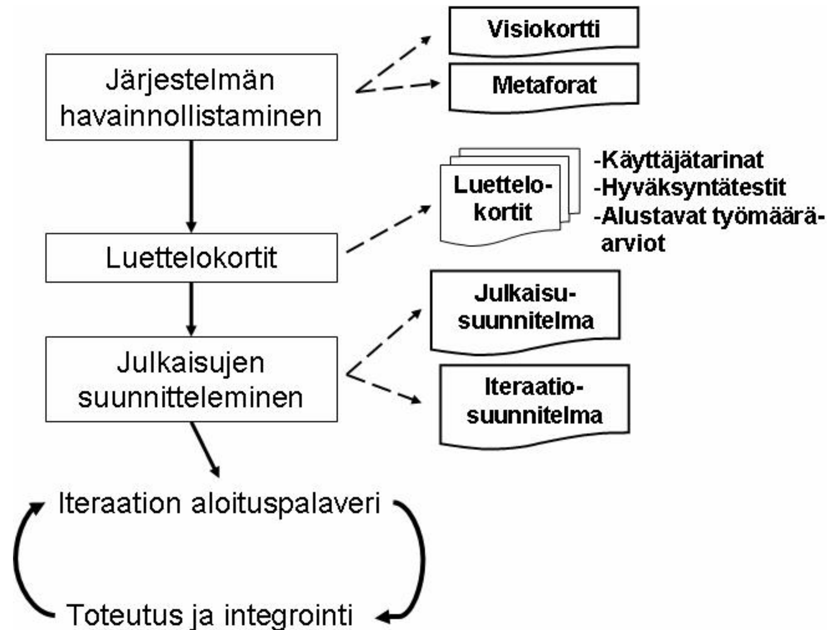
Vaikka jokaisen inkrementin tuloksena syntyy toimivaa ohjelmakoodia, XP:ssä ei jokaisen inkrementin tuotosta julkaista. *Suunnittelijat* aikatauluttavat julkaistavat ohjelmistoversiot asiakkaan tarpeiden mukaisesti. (Astels et al. 2002, 20, 181)

Rahoittajat varmistavat, että projektilla on tarpeellinen rahoitus, tarpeellinen määrä henkilöstöä sekä tarvittavat työkalut. *Päällikkö* on koko projektin johtaja, joka poistaa byrokraattisia esteitä projektitiimin tieltä, ja varmistaa, että työt tulevat tehdyksi. Päällikkö siis käytännössä työskentelee molemmille tiimeille, vaikka organisaatiokaavion mukaan he kaikki työskentelevät hänelle. Päällikkö on yleensä *rahoittaja* tai rahoittajan esimies. (Astels et al. 2002, 20-21)

Tuotekehitystiimissä on ohjelmoijia, joista joillekin on määrätty ohjelmointityön lisäksi tietyt roolit: valmentaja, jäljittäjä, fasilitoija sekä arkkitehti. *Valmentaja* on kokenut työntekijä, jonka päätehtävä on varmistaa, että prosessi on kunnossa ja projekti etenee. Valmentajat myös pyrkivät tuomaan esille sellaisia seikkoja, jotka ovat ohjelmiston kannalta tärkeitä, mutta joita asiakastiimi ei ole osannut vaatia. *Jäljittäjän* päätehtävä on tiimin suorituksen parantaminen. Hän tutkii työmääräarvioiden paikkansa pitävyyttä ja sekä projektiryhmän työskentelynopeutta ja antaa palautetta projektitiimille. *Fasilitoija* on kommunikointikykyinen henkilö, jonka päätehtävänä on vastata tiimien välisestä kanssakäymisestä ja suhteista. *Arkkitehdin* rooli poikkeaa XP:ssä hieman perinteisestä. XP:ssä arkkitehti ei suunnittele ohjelmistoarkkitehtuuria etukäteen, vaan kehittä arkkitehtuuria sitä mukaa, kun sitä tarvitaan. Luonnollisesti tällainen lähestymistapa vaatii refaktorointia, joka on tietenkin myös arkkitehdin tehtävä. Arkkitehti myös kirjoittaa ja ajaa arkkitehtuuria testaavia testitapauksia. (Astels et al. 2002, 22-25)

3.2.3 XP-prosessi

XP-prosessi voidaan jakaa viiteen vaiheeseen kuvion 8 mukaisesti.



KUVIO 8 *XP-prosessi*

Järjestelmän havainnollistaminen

Prosessi alkaa järjestelmän havainnollistamisella, jossa pyritään muodostamaan selkeä käsitys siitä, millaista järjestelmää ollaan tekemässä. Joskus asiakkaalla voi olla hyvinkin selkeä näkemys tarvittavasta järjestelmästä, mutta useimmiten lähtökohtana on vain hatara, yleisen tason käsitys halutusta tuotteesta. Järjestelmän havainnollistamiseen käytetään XP:ssä *visiokorttia (Vision Card)* sekä metaforia. (Astels et al. 2002, 32-34, 35)

Visiokortti on alle 25 sanan kuvaus toivotusta järjestelmästä ja sen kirjoittamisesta vastaa asiakastiimi. Tavoitteen tiivistäminen 25 sanaan pakottaa asiakkaan todella miettimään projektin tavoitteita. Tuotekehitystiimin tulee välittömästi kortin valmistuttua pohtia, mitä kyseisen tavoitteen täyttäminen tulee käytännössä vaatimaan, mitä riskejä projektilla on sekä mitä teknologiaa tarvitaan. Visiokortin tekemisen jälkeen asiakastiimi pyrkii XP:n toisen periaatteen

mukaisesti kuvailemaan millainen haluttu tuote on vertauksia tai metaforia käyttämällä. (Astels et al. 2002, 32-34, 35)

Luettelokortit

Prosessin seuraavassa vaiheessa kirjoitetaan *luettelokortit* (*Index Card)*, joita käytetään XP:ssä vaatimusmäärittelydokumentin sijaan. Jokaista toteutettavan järjestelmän asiakkaalle näkyvää selkeää osaa kohden kirjoitetaan yksi luettelokortti. (Astels et al. 2002, 39, 41)

Kortin kirjoittamisen aloittavat asiakastiimin *tarinankertojat.* He kirjoittavat kullekin kortille yhden *käyttäjätarinan* (User Story), joka on käytännössä yksinkertaisin mahdollinen tapa kuvailla järjestelmän osaa. Luettelokortin yläosaan kirjoitetaan toiminnon otsikko (2-3 sanaa), esimerkiksi ”tilauksen lähettäminen” ja keskiosaan tarkempi kuvaus (1-3 lausetta) toiminnosta. (Astels et al. 2002, 40-41)

Luettelokortit kootaan niiden toteuttamien suurempien toimintakokonaisuuksien perusteella *pinoksi.* Pinon jokaisen kortin vasempaan yläreunaan kirjoitetaan juokseva numero osoittamaan kortin sijaintia pinossa. Seuraavan pinon numerointi aloitetaan edellisen pinon viimeisestä numerosta. Kun kaikki pinot ovat valmiina, ne kootaan *pakaksi (Deck)*. Pakka edustaa näin ollen asiakkaan sen hetkistä näkemystä tulevasta tuotteesta. Mikäli tuotteen vaatimukset muuttuvat, luettelokortteja voidaan lisätä tai poistaa pakasta. (Astels et al. 2002, 42)

Kun pakat ovat valmiina, asiakastiimi kirjoittaa yhden *hyväksyntätestin* jokaista käyttäjätarinaa kohden. Hyväksyntätestien kirjoittaminen varmistaa, että kaikki käyttäjätarinat ovat testattavissa. Testit kirjoitetaan ennen kuin kyseiset käyttäjätarinat toteutetaan, jotta iteraatioiden tulokset voidaan testata välittömästi iteraation päätyttyä. Vikojen löytäminen on XP:ssä yksikkötestien tehtävä, joten hyväksyntätestien tarkoituksena on vain käyttäjätarinassa kuvatun toiminnallisuuden varmistaminen. Tästä syystä hyväksyntätesteistä ei ole tarpeen tehdä täydellisen kattavia. Testit kannattaa myös mahdollisuuksien mukaan automatisoida testaamisen helpottamiseksi. (Astels et al. 2002, 49-50, 55)

Tuotekehitystiimi voi aloittaa työmääräarvioiden tekemisen samanaikaisesti kun asiakastiimi kirjoittaa hyväksyntätestejä. Aluksi tuotekehitystiimi arvioi kunkin käyttäjätarinan vaativan työmäärän miestyöviikkoina (XP:ssä miestyöviikosta käytetään termiä *Story Point*) ja kirjoittaa arvioidun keston luettelokortin oikeaan yläkulmaan. Käyttäjätarinoiden riippuvuuksiin ei tässä vaiheessa kiinnitetä huomiota, vaan työmääräarviot tehdään sellaisesta lähtökohdasta, että kaikki riippuvuudet on jo toteutettu. (Astels et al. 2002, 57, 70,73)

Kestoltaan yli 3 miestyöviikon mittaista käyttäjätarinaa kutsutaan *eepokseksi.* Eepokset on jaettava osiin, jotta ne on helpompi toteuttaa lyhyissä iteraatioissa. Jos osiin jakaminen nähdään tarpeelliseksi, luettelokortin oikeaan yläkulmaan kirjoitetaan ”split”. Lopullinen päätäntävalta jakamisesta on asiakastiimillä, mutta tuotekehitystiimi voi tietenkin tehdä ehdotuksia jakamistavasta. Jakaminen on usein tarpeen, ja se tietenkin lisää käyttäjätarinoiden kokonaismäärää, mikä on hyvä huomioida projektin kestoa arvioidessa. (Astels et al. 2002, 72-73)

Työmääräarvioissa voidaan käyttää apuna *spike*-tuotoksia tai *yksinkertaista ratkaisumallia*. Nämä eivät ole varsinaisia prosessin vaihetuotteita, mutta niistä voi olla hyötyä hankalampien käyttäjätarinoiden työmääriä arvioitaessa. (Astels et al. 2002, 58-59)

Spike-tuotos tarkoittaa ohjelmakoodin kirjoittamista työmääräarvioinnin helpottamiseksi. Ohjelmakoodia kirjoitetaan vain niin pitkälle, että tehtävä selkiytyy ja luotettava työmääräarvio on mahdollista antaa. Spike ei ole lopullista koodia, eikä sitä saa liittää lopulliseen tuotteeseen sellaisenaan. (Astels et al. 2002, 59, 74; Pressman 2005, 112)

Yksinkertainen ratkaisumalli (One Simple Solution) on yksi mahdollinen ratkaisu, jolla päästään haluttuun lopputulokseen. Samoin kuin spike-tuotoksen, myös yksinkertaisen ratkaisumallin tarkoituksena on selventää tehtävää ja helpottaa siten työmääräarvion tekemistä. Mikäli tuotekehitystiimin jäsenet keksivät useampia ratkaisumalleja, niistä yksinkertaisin valitaan tässä vaiheessa. Myös

yksinkertainen ratkaisumalli tuhotaan, kun työmääräarviot on tehty, koska sen ei haluta ohjaavan työntekoa. (Astels et al. 2002, 58-59)

Julkaisujen suunnitteleminen

Kun työmääräarviot on tehty, siirrytään *julkaisujen suunnitelemisvaiheeseen.* Aluksi asiakastiimin suunnittelija tekee julkaisusuunnitelman, mikä sisältää arvion projektin kokonaiskestosta sekä projektiaikaisten julkaisujen ajankohdat. Jälkimmäinen on tarpeen, koska XP:ssä ei yleensä julkaista jokaisen inkrementin tuotosta loppukäyttäjälle saakka, vaan joskus tuotos annetaan vain asiakastiimin käyttöön. Kokonaiskeston arvioinnissa suunnittelija käyttää annettuja työmääräarvioita. Toisesta iteraatiosta lähtien suunnitelmaa voidaan täsmentää, kun projektitiimin työskentelynopeus alkaa hahmottua. Työskentelynopeuden laskee jäljittäjä toteumien perusteella.
(Astels et al. 2002, 77, 181)

Kun julkaisusuunnitelma on valmis, tehdään iteraatiosuunnitelma, eli jaetaan käyttäjätarinat alustavasti iteraatioihin. Koko projektin valmistumisajankohta päätetään usein byrokratiasyistä etukäteen. XP-projektissa aikataulu ei aikarajoitettujen iteraatioiden vuoksi voi venyä, joten mikäli aika ei jostain syystä riitä kaikkien ominaisuuksien toteuttamiseen, on projektiryhmän tingittävä toteutettavista ominaisuuksista. Tällöin on tärkeää, että kaikki tärkeimmät ominaisuudet toteutetaan, ja toteutuksesta pudotetaan vain muutamia vähiten tärkeitä ominaisuuksia. Tästä syystä käyttäjätarinapinot toteutetaan tärkeysjärjestyksessä ja siksi käyttäjätarinapinot on priorisoitava. Priorisoinnin hoitaa asiakastiimi, pääosin *suunnittelija*. (Astels et al. 2002, 79, 84-87)

Iteraatiosuunnitelma pitää sisällään myös iteraatioiden aloitus- ja lopetuspäivämäärät sekä toteutukseen osallistuvat tiimit. Tässä vaiheessa käyttäjätarinoiden riippuvuudet toisistaan tarkastetaan ja kirjataan iteraatiosuunnitelmaan. Käyttäjätarina voidaan ottaa iteraatioon mukaan vain, jos sen riippuvuudet on jo toteutettu tai ne toteutetaan samassa iteraatiossa. (Astels et al. 2002, 79, 84-87)

Iteraatiot

Kun iteraatiosuunnitelma on valmis, aloitetaan toteutustyö iteraatioissa. Jokaisen iteraation ensimmäinen vaihe on *iteraation aloituspalaveri*, jossa valitaan seuraavassa iteraatiossa toteutettavat käyttäjätarinat käyttäen pohjatietona iteraatiosuunnitelmaan kirjattua alustavaa jakoa. Asiakastiimi esivalitsee tärkeimmät vielä toteuttamatta olevista käyttäjätarinoista, ja tarinankertojat esittelevät ne aloituspalaverissa. Seuraavaksi tuotekehitystiimi jakaa jokaisen käyttäjätarinan lyhyisiin, testattavissa oleviin tehtäväkokonaisuuksiin, eli *taskeihin*. XP:n viidennen periaatteen mukaisesti ohjelmoijien on aina ennen itse taskin ohjelmakoodin toteuttamista kirjoitettava yksikkötesti, jolla toteutettavan taskin toiminnallisuus voidaan varmentaa. (Astels et al. 2002, 86, 90-91)

Kun taskijako on tehty, jokaiselle taskille määrätään vastuullinen – tai XP:n mukaan ”*kullekin taskille ilmoittautuu vapaaehtoiseksi tekijäksi joku tuotekehitystiimistä”* – ja sen jälkeen vastuulliset arvioivat kuinka monta työpäivää heillä menee kunkin taskin suorittamiseen. Tässä yhteydessä on huomattava, että työmääräarviot annetaan todellisina työpäivinä, ei esimerkiksi teoreettisina 8 tunnin työpäivinä. Kunkin työntekijän on siis huomioitava mahdolliset muut työtehtävänsä sekä yleisiin asioihin kuluva aika arviota tehdessään. (Astels et al. 2002, 92)

Kun työmääräarviot on tehty, lasketaan kaikkien taskien arvioidut kestot yhteen. Mikäli taskien toteuttamiseen arvioitu aika ylittää iteraation pituuden, asiakastiimi päättää mitkä käyttäjätarinoista voidaan jättää pois tästä iteraatiosta ja siirtää seuraavaan iteraatioon. (Astels et al. 2002, 92)

Kun iteraation sisältö on päätetty, tuotekehitystiimi toteuttaa kyseiseen iteraatioon valitut käyttäjätarinat pariohjelmointina, tehden aina yksikkötestit ensin ja sen jälkeen toteutuksen, jolla yksikkötesti läpäistään. (Pressman 2005, 112)

Toteutusvaiheen aikana tuotekehitystiimi pitää XP:n viidennen periaatteen mukaisesti päivittäin lyhyen kokouksen, jossa käydään läpi kunkin tiiminjäsenen kohdalta kolme kysymystä, jotka ovat ”mitä teit eilen?”, ”mitä aiot tehdä tänään?”

sekä ”onko ongelmia?”. Muistutettakoon vielä kerran, että kokouksen tavoitteena on ainoastaan tiedon jakaminen, ei missään tapauksessa työn johtaminen tai toisten tiiminjäsenten työn etenemisen valvonta. (Astels et al. 2002, 93-94)

Iteraatiot ovat aikataulullisesti rajattuja (time-boxed), eli iteraation tuotoksen, ohjelmistoinkrementin, myöhästyminen ei ole mahdollista. Mikäli kaikkea, mitä iteraatioon oli suunniteltu, ei saada valmiiksi, siirretään kesken jääneet tehtävät seuraavaan iteraatioon. Tällä tavalla varmistetaan, että jotain käyttökelpoista ohjelmistoa julkaistaan varmasti ennalta määrättyinä ajankohtina. Iteraation lopuksi asiakastiimi ajaa hyväksyntätestin uudelle ohjemistoinkrementille. Jokaisen inkrementin on läpäistävä hyväksyntätestit, vaikkei jokaista inkrementtiä yleensä viedäkään loppukäyttäjälle saakka. (Astels et al. 2002, 87, 49-50, 181)

Seuraavaksi pidetään uusi iteraation aloituspalaveri, jossa päätetään seuraavassa iteraatiossa toteutettavat käyttäjätarinat, jotka sen jälkeen jaetaan taskeihin ja toteutetaan. Näitä vaiheita toistetaan projektin päättymispäivään saakka. Mikäli projektin kokonaiskestolle ei ole asetettu aikaraamia, projekti päättyy, kun kaikki käyttäjätarinat on toteutettu. Mikäli seuraavan iteraation tuotos on tarkoitus julkaista loppukäyttäjälle, lisätään iteraatioon yleensä laajamittaisempaa testausta. (Astels et al. 2002, 86, 78, 185)

3.2.4 Yhteenveto ja arviointi

XP:n tehokkuus perustuu ohjelmointityöhön keskittymiseen sekä Lean-henkiseen mahdollisimman vähäisen työpanoksen käyttämiseen toissijaisiin toimiin. Toiminnan perustana ovat itsenäisten tiimien toteuttaman ja asiakkaan aktiivisen osallistumisen ohjaaman iteratiivisen ja inkrementaalisen tuomat edut. XP:ssä määritellään lisäksi näitä prosessin osia tukevia toimia sekä itse ohjelmointityötä tehostavia toimia.

XP:ssä perinteisestä projektimallista on poikettu niin radikaalisti, että mikäli menetelmästä halutaan todellista hyötyä, on kaikkia prosessin osia noudatettava

(Astels et al. 2002, 193, 195, xvii). Perinteisillä menetelmillä työskennelleiltä tiimeiltä tai yrityksiltä vaaditaan näin ollen ohjelmistoprosessin täydellistä uudistamista sekä kaiken tutun ja turvallisen hylkäämistä.

XP:ssä on määritelty mielenkiintoisia menetelmiä, kuten testauslähtöinen ohjelmistokehitys, mutta myös omituisuuksia, kuten pariohjelmointi ja seisontapalaverit. Kaikki toiminnot on määritelty pakollisiksi prosessin osiksi. Tämä ratkaisu kummastuttaa, sillä agile-menetelmät on kehitetty siksi, että liian tarkkaan määritellystä ja raskaasta prosessista johtuen perinteisissä menetelmissä kului liikaa aikaa ja voimavaroja prosessin seuraamiseen vain prosessin seuraamisen vuoksi. Nyt on siis kehitetty tilalle uusi prosessimalli – jossa määritellään kaikki toimet vielä tarkemmin kuin perinteisissä menetelmissä. Tämä ajattelutapa ei mielestäni ole täysin johdonmukainen.

XP:ssä vaadittuun pariohjelmointiin siirtyminen on varmasti erittäin vaikea perustella projektitiimiläisille. Pelkkä ajatus jatkuvasta valvovan silmän alla työskentelemisestä on varmasti monelle epämieluisa, eikä proksemiikankaan (Kielijelppi 2008, BodyLanguageExpert 2008) vaikutusta sovi väheksyä. Astels, Granville ja Novak perustelevat pariohjelmoinnin pakollisuutta mainitsemalla, että ”tutkimukset osoittavat - - että ohjelmoiminen pareissa on yhtä tuottavaa, kuin se, että molemmat työskentelisivät yksin” (Astels et al. 2002, 79). Tällaiset tutkimukset olisivat hyvin kiinnostavia, mutta valitettavasti kyseiset herrat jättävät mainitsematta mikä tutkimus tarkkaan ottaen on kyseessä ja missä tulokset ovat nähtävillä – mikä on perin outoa, sillä muutoin lähteet on esitelty heidän kirjoituksissaan kautta linjan moitteettomasti.

Myös XP:n seisontapalaverit kummastuttavat. Ovatko ohjelmoijat todella niin kurittomia, että heitä ei saa keskittymään käsillä olevaan aiheeseen, jos heidän sallitaan istua? Oma kokemukseni agile-prosessien soveltamisesta ei tue tätä käsitystä (Liite 3), mutta yksittäisen projektin perusteella ei toki voi tehdä merkittäviä yleistyksiä. Päivittäinen tiedontasauspalaveri ei sinänsä kuulosta huonolta ajatukselta, mutta tuolittomuuden määritteleminen pakolliseksi prosessin osaksi ei vaikuta aivan loppuun asti ajatellulta päätökseltä. Perusteluna

seisontapalavereille mainittiin se, että perinteiset palaverit ovat ohjelmoijien mielestä ”pitkiä ja tylsiä” ja pois oikeiden töiden tekemisestä (Astels et al. 2002, 6). Seisoskelukäytäntö voi hyvinkin auttaa palaverien pituuteen, mutta vaikka menettelytapa lieneekin tarkoitettu perin vitsikkääksi kevennykseksi normaaleihin rutiineihin, en usko sen poistavan palaverien tylsyyttä. Mikäli tiimiläisillä on todella sellainen asenne, että palavereihin kuluva aika on pois oikeiden töiden tekemisestä, on vaikea uskoa heidän kokevan jokapäiväistä seisoskeluleikkiä erityisen motivoivana asiana.

XP:n ehdoton vaatimus asiakkaan kokopäiväisestä läsnäolosta (Astels et al. 2002, 191) on myös haastellinen. Hyödyt ovat toki kiistattomia, mutta ongelmallista on se, kuinka asiakas saadaan tällaiseen järjestelyyn mukaan.

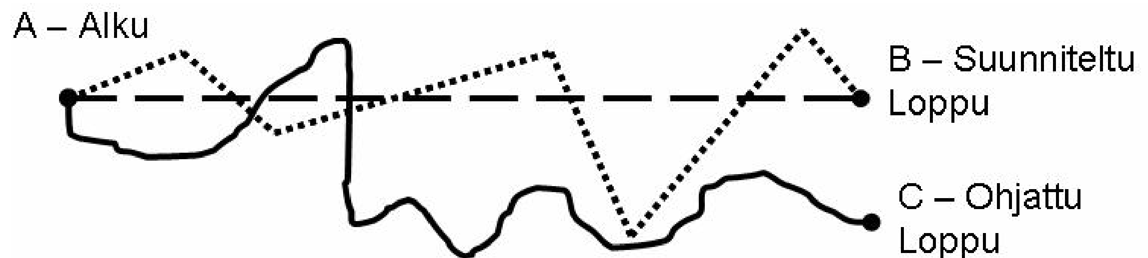
Astels, Granville ja Novak eivät juuri käsittele dokumentointia, joskaan dokumenttien kirjoittamista ei varsinaisesti kielletäkään. Alistair Cockburn (2007, 42-43, 219-220) kuitenkin huomauttaa, että on yleinen väärinkäsitys, että agile-menetelmissä olisi tarkoitus pyrkiä dokumentittomuuteen. Esimerkissään hän käyttää nimenomaan XP:tä *väärin* tulkinnutta projektiryhmää, joka ei tuottanut mitään teknisiä dokumentteja. XP:täkin käytettäessä tulisi siis tuottaa dokumentteja, mutta vain aidosti tarpeellinen määrä (Cockburn 2007, 219).

XP:n soveltuvuutta suuriin projekteihin on yleisesti arvosteltu. Astels, Granville ja Novak vakuuttavat, että XP:n on kyllä sovellettavissa suuriin projekteihin, mutta esitetty skaalausratkaisu on hyvin keinotekoinen: suuri projektiryhmä, jossa on osallisia monella eri paikkakunnalla, tulee jakaa useaksi, XP:tä *itsenäisesti* käyttäväksi tiimiksi. Mitään ohjeita tiimien välisen kommunikoinnin järjestämisestä ei anneta. Mielestäni tällaisessa ratkaisussa XP:tä ei skaalata, vaan pyöritetään skaalamaatonta XP-prosessia useassa tiimissä, joten ratkaisu ei ole osoitus XP:n skaalautuvuudesta, vaan päinvastoin sen *skaalautumattomuudesta*. Esimerkiksi Scrumissa skaalaamisen perusajatus on sama, mutta siinä on myös tiimien välinen kommunikointi huomioitu. (Astels et al. 2002, 197-199; (Schwaber 2004, 123-124))

3.3 Adaptive Software Development (ASD)

Adaptive Software Development (ASD) tähtää muita agile-menetelmiä selkeämmin projektin hallintaan. Mitään tiettyjä ohjelmistokehitystekniikoita ei määritellä eikä velvoiteta käyttämään, niiden osalta projektiryhmälle jätetään vapaat kädet. (Highsmith 2000, 70)

ASD:ssä muutosta pidetään ohjelmistoprojekteissa luonnollisena. Perinteisissä menetelmissä suunnitelmasta poikkeaminen mielletään virheeksi, joka on korjattava pikimmiten. ASD:ssä poikkeamat mielletään oikeaan suuntaan ohjaaviksi, positiivisiksi asioiksi. (Highsmith 2000, 43)



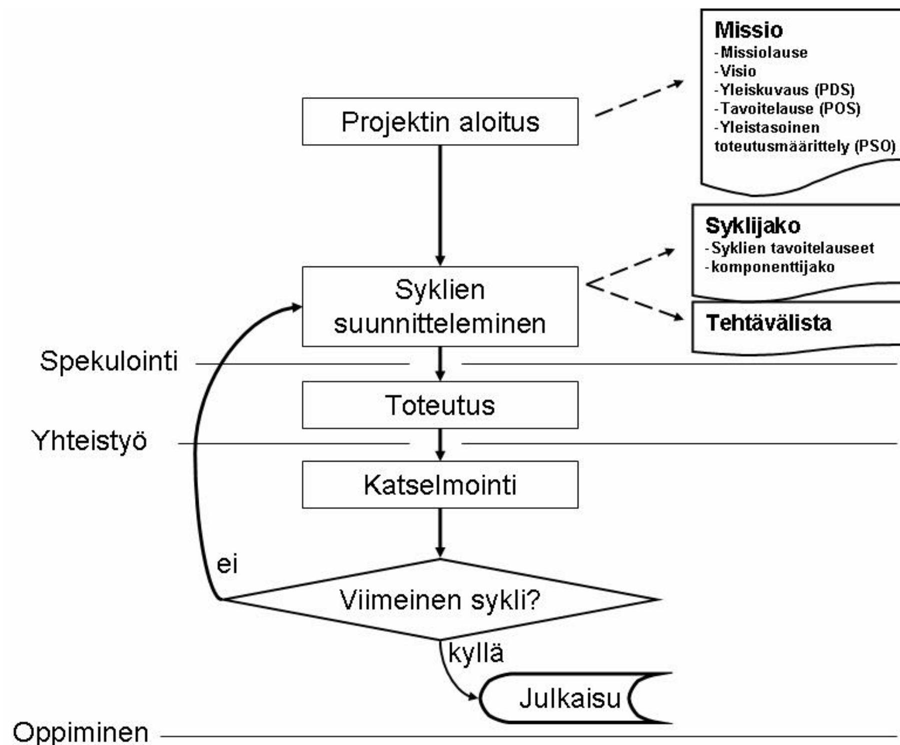
KUVIO 9 *Ohjattu lopputulos. (Highsmith 2000, 43)*

Kuviossa 9 on esitetty kuvitteellinen projekti, jonka on suunniteltu etenevän katkoviivaa pitkin lähtökohdasta A suoraan suunniteltuun lopputulokseen B. Tiheämpi katkoviiva kuvaa perinteistä projektimallia, jossa jokainen alkuperäisestä suunnasta poikkeaminen korjataan mahdollisimman pian, jotta päädyttäisiin alussa suunniteltuun kohteeseen. Yhtenäinen, mutkitteleva viiva taas kuvaa ASD-projektia. ASD-projekti lähtee samoista lähtökohdista, mutta sen määränpään oletetaan poikkeavan alkuperäisestä (piste C). Perusajatus on, että monimutkaista järjestelmää toteutettaessa aikaisessa vaiheessa tehtyä suunnitelmaa noudattamalla saadaan kyllä aikaan sellainen järjestelmä, mikä suunniteltiin. Ainoa ongelma on, että järjestelmä tuskin on sellainen mitä todella tarvittiin. (Highsmith 2000, 42-43)

3.3.1 ASD-prosessi

ASD-prosessin vaiheet sekä niiden tavoitteet muistuttavat sisällöltään ja tavoitteiltaan perinteistä vesiputousmallista prosessia. ASD eroaa kuitenkin perinteisistä menetelmistä siten, että siinä vaiheiden ei vaadita olevan jaksottaisia, vaan ne saavat olla osittain samanaikaisia. Lisäksi ASD:ssä varsinainen toteutusvaihe suoritetaan sarjana iteraatioita. (Highsmith 2000, 41)

Prosessin päävaiheet ovat spekulointi (speculate), yhteistyö (collaborate) ja oppiminen (learn). Jokainen päävaihe sisältää täsmällisempiä alivaiheita kuvion 10 mukaisesti. (Highsmith 2000, 84; Pressman 2005, 115).



KUVIO 10 *ASD-Prosessi* (Pressman 2005, 115; Highsmith 2000, 84)

Projektin aloitus

Prosessin aluksi työstetään projektin *missio*. Missio pitää sisällään kaiken, mikä auttaa tiimiä projektin tavoitteiden määrittelemisessä, eli vastaa sisällöltään perinteisen prosessimallin esitutkimusvaihetta. (Highsmith 2000, 44)

Mission työstäminen alkaa missiokuvauksen määrittelemisestä. Missiokuvaus on lyhyt, yleisen tason kuvaus projektin tavoitteesta. Kuvaus ohjaa projektia, mutta ei kuitenkaan yksin riitä, vaan tiettyjä tekijöitä on määriteltävä myös tarkemmalla tasolla (Highsmith 2000, 53, 55)

Missiolauseen lisäksi projektin missio sisältää kolme vaihetuotetta: vision, yleiskuvauksen sekä yleistasoisen toteutusmäärittelyn. (Highsmith 2000, 61)

Vision tavoitteena on kuvailla projektin päätavoitteet liiketaloudellisesta näkökulmasta. Se on siis perinteinen esitutkimusraportti, joka sisältää muun muassa selvitystä projektin taustasta sekä riippuvuuksista, arvioita tarvittavista resursseista, projektin liiketaloudelliset tavoitteet sekä suorituskykyvaatimukset. Lisäksi visio sisältää alle 25-50-sanaisen *visiolauseen*, jossa on mainittava tuleva tuote, sen asiakas, sekä miten tuote eroaa merkittävimmistä kilpailevista tuotteista. (Highsmith 2000, 62-64)

Projektin yleiskuvauksessa (Project Data Sheet, PDS) visio on tiivistetty yhdelle sivulle, josta kuka tahansa asiasta kiinnostunut voi saada nopealla vilkaisulla yleiskäsityksen projektista. Yleiskuvauksen tekemisen tarkoituksena on myös pakottaa projektitiimi pohtimaan tavoitteita ja ymmärtämään ne, sillä muutoin tiivistäminen vain yhteen sivuun ei ole mahdollista. Yleiskuvaus sisältää t*avoitelauseen* (Project Objective Statement, POS), mikä on alle 25-sanainen kuvaus projektin tavoitteista, sisältäen yleistasoista tietoa aikataulusta, resursseista sekä aiheen rajaamisesta. (Highsmith 2000, 65-66)

Yleistasoinen toteutusmäärittelyn (Product Specification Outline, PSO) päätarkoitus on tietenkin toiminnallisuuden kuvaaminen yleisellä tasolla, mutta se sisältää myös kuvauksen projektin tavoitteiden rajaamisesta sekä määrittelee

tarvittavat ominaisuuskokonaisuudet eli *komponentit*. ASD:ssä on kolmenlaisia komponentteja: asiakkaalle lisäarvoa tuovat pääkomponentteja, järjestelmän vaatimia *teknologiakomponentit* (esimerkiksi verkot, tietokannat, käyttöjärjestelmät) sekä *tukikomponentteja* (esimerkiksi koulutusmateeriali). (Highsmith 2000, 69-70)

Syklien suunnitteleminen

Kun missio on määritelty, aloitetaan iteraatioiden – joita ASD:ssä kutsutaan *sykleiksi* – suunnitteleminen. Suunnitteluvaiheessa arvioidaan projektin kokonaiskesto, jaetaan projekti alustavasti sykleihin, määritellään ja päätetään asiakaskatselmointipäivät. ASD:ssä syklien pituudet ovat 4-8 viikkoa alle yhdeksän kuukautta kestävissä projekteissa ja 6-10 viikkoa sitä pidemmissä. ASD:ssä kaikki projektin syklit eivät myöskään välttämättä ole samanpituisia. Jokaiselle syklille määritellään myös *tavoitelause*, samaan tapaan kuin koko projektille missiovaiheessa määritelty PSO. (Highsmith 2000, 94-95)

Kun raamit on asetettu, kohdennetaan pääkomponentit tiettyihin sykleihin. Alkupään sykleissä kannattaa keskittyä tärkeimpien toiminnallisuuksien toteuttamiseen riittävällä tasolla ja vasta myöhemmissä näiden toiminnallisuuksien viimeistelemiseen sekä toissijaisten toiminnallisuuksien toteuttamiseen. Myös riskialttiimmat komponentit kannattaa kohdentaa alkupään sykleihin. Kaikista muista agile-menetelmistä poiketen ASD:ssä monimutkaisia komponentteja ei jaeta pienempiin osiin, vaan ne toteutetaan useammissa sykleissä. (Highsmith 2000, 97-98)

Seuraavaksi kohdennetaan tekniset komponentit sekä tukikomponentit sykleihin. Myös dokumentit kohdennetaan sykleihin, vaikka tiedostetaankin se tosiasia, että dokumentteja joudutaan päivittämään projektin kestäessä. (Highsmith 2000, 98-99)

Kun komponentit on kohdennettu, tehdään tehtävälista. Lista jätetään usein virallisen prosessin ulkopuolelle, jolloin projektitiimi voi toteuttaa listan haluamallaan tavalla ja käyttää sitä työn jakamiseen. Mikäli lista kuitenkin jostain

syystä halutaan prosessin viralliseksi vaihetuotteeksi, työstetään se lisäämällä komponenttiluetteloon jokaisen komponentin kohdalle yksityiskohtaisempia vaiheita. (Highsmith 2000, 99).

Toteutus

Kun kokonaissuunnitelma projektin läpiviennistä on saatu valmiiksi, aloitetaan ensimmäinen tuotekehityssykli. Kuten muissakin agile-menetelmissä, sykli on aikarajoitettu, eli mikäli aika jostain syystä loppuu kesken, ei aikataulu veny, vaan osa komponenteista siirretään seuraavan syklin tavoitteisiin. (Highsmith 2000, 83)

Katselmointi

Syklin jälkeen pidetään asiakaskatselmointi. Katselmoinnissa tutkitaan vikojen määrää sekä tavoitteiden saavuttamista ja arvioidaan tiimin työskentelytapojen tehokkuutta sekä aiemmin tuotettujen vaihetuotteiden ajantasaisuutta. Tavoitteena on löytää vastaukset seuraaviin kysymyksiin:

- Eteneekö projekti mission mukaisesti?
- Ovatko vaihetuotteet yhä ajan tasalla?
- Vastaako tuotoksien laatu asiakkaan teknisiä vaatimuksia ja odotuksia?
- Työskenteleekö projektitiimi tehokkaasti?

(Highsmith 2000, 101)

Katselmoinnin jälkeen tehdään katselmoinnissa saadun palautteen perusteella tarpeelliset muutokset vaihetuotteisiin, mukaan lukien syklien toteutussuunnitelmaan. Käytännössä siis palataan syklien suunnitteluvaiheeseen. Päivitysten jälkeen aloitetaan seuraava sykli. Syklejä toistetaan, kunnes kaikki tarvittavat ominaisuudet on toteutettu. (Highsmith 2000, 84; Pressman 2005, 115)

Vaikka syklien tuotokset ovatkin toimivaa ja ajettavaa ohjelmistoa, ja niiden toimintaa arvioidaan asiakaskatselmoinneissa, ei ASD:ssä jokaisen syklin tuotosta julkaista myyntiin saakka. Myyntiin menevä tuotos, eli uusi *versio*, on

usean syklin tuotos, ja sen valmistuminen kestää yleensä useita kuukausia. (Highsmith 2000, 91)

3.3.2 Yhteenveto ja arviointi

Ennen tämän tutkimuksen aloittamista kummastelin suuresti sitä, miten vaikeaa ASD-prosessista oli löytää yksityiskohtaista kuvausta. Internet-hauilla löytyi vain hyvin yleistasoisia kirjoituksia ja ohjelmistoprosesseja yleisesti kuvailevissa teoksissakin (esimerkiksi Pressman 2005) ASD-prosessikuvaus oli rankasti yleistetty. Asiaa enemmän tutkittuani minulle selvisi, että syy on hyvin yksinkertainen: tarkka kuvaus on erittäin vaikea tehdä saatavilla olevan julkaistun materiaalin (käytännössä vain Highsmith, 2000) perusteella. Highsmithin kielteisyys määriteltyjä prosesseja kohtaan (Highsmith 2000, 56-57) on valitettavasti johtanut prosessin vaiheiden, vaihetuotteiden sekä roolien selkeiden ja johdonmukaisten kuvausten puuttumiseen. Highsmith ei toki kutsu ASD:tä prosessiksi, vaan haluaa käyttää sanaa *pattern* (malli), mutta siihen voi vain todeta Shakespearen sanoin *”That, which we call a rose, by any other name would smell as sweet*” (Shakespeare, 1992, 700).

Kun prosessi nyt kuitenkin on kovan työn tuloksena saatu jollain tavalla jäsennettyä ja prosessia voidaan alkaa arvioimaan tarkemmin, löytyy siitä mielenkiintoisia piirteitä. Varsinaiset hyödyt ovat iteratiivisen ja inkrementaalisen ohjelmistokehityksen tuomia etuja. ASD:n toimien perustana vaikuttaisi olevan perinteisten ohjelmistokehitystapojen ja iteratiivisen työskentelytavan yhdistäminen.

Erityisesti pidän ASD:ssä siitä, että mitään ohjelmointitekniikoita ei määritellä prosessin pakolliseksi osaksi (kuten XP:ssä), mutta prosessin sisältöä ei kuitenkaan jätetä projektitiimin kehitettäväksi (kuten Crystal-menetelmissä).

Selkein ero muihin agile-menetelmiin on vesiputousmallista suoraan lainattu mittava esitutkimusvaihe. Ero vesiputousmalliin on muutenkin hiuksenhieno. käytännössä kyse on samasta prosessista sillä erotuksella, että ASD sallii

palaamisen aikaisempiin vaiheisiin. Vesiputousmallissa aikaisempiin vaiheisiin voidaan toki myös palata, mutta ainoastaan pakon edessä ja silloinkin periaatteessa prosessin vastaisesti (Highsmith 2000, 86). Myös dokumentaation tärkeys on ASD:ssä tiedostettu, joskin sen määrää pyritään rajoittamaan (Highsmith 2000, 99). ASD:ssä ei myöskään keskitytä vesiputousmallisen tuotekehityksen moittimiseen, vaan päinvastoin jopa suositellaan sitä tietyn tyyppisiin projekteihin. (Highsmith 2000, 95)

Kyseenalaisena pidän ASD:ssä sitä, että prosessi sallii iteraatioiden tavoitelistalle otettavan tavoitteita, joita ei ole tarkoituskaan saada kyseisen iteraation aikana valmiiksi (Highsmith 2000, 97, 99). Yleensä agile-menetelmissä iteraatioon määritelty tavoite on joko tehty (100 %) tai sitä ei ole tehty (0 %), mitään ”melkein valmis”-määreitä ei sallita, koska tiedostetaan, että tällöin työmäärän arvioiminen ja samalla iteraatioiden suunnitteleminen hankaloituu.

Hieman karrikoituna ASD on perinteisen ohjelmistokehitysmallin mukaista tuotekehitystä sillä erotuksella, että toteutusvaihe suoritetaan iteraatioissa ja aikaisempiin vaiheisiin palaamisen odotetaan olevan tarpeen.

3.4 Crystal-menetelmät

3.4.1 Yleistä Crystal-perheestä

Crystal-menetelmien perusajatus on tarjota malleja ohjelmistoprosessin suunnittelemiseksi. Ajatus ei siis ole yrittääkään tarjota valmista prosessia, kuten kaikissa muissa agile-menetelmissä, vaan ainoastaan joukko ohjeita ja sääntöjä, joiden perusteella jokainen projektiryhmä voi räätälöidä omiin tarkoituksiinsa sopivan prosessin. (Cockburn 2007, 335-337)

Kaikki Crystal-menetelmät noudattavat määriteltyjä sääntöjä sekä Cockburnin määrittelemiä yleisiä prosessinsuunnitteluperiaatteita. Säännöissä määrätään, että ohjelmistokehityksen on oltava inkrementaalista, ja että

prosessinarviointipalavereja on pidettävä ainakin jokaisen inkrementin lopussa ja alussa, mieluiten myös keskivaiheilla. (Cockburn 2007,339)

Cockburnin (2007, 182) määrittelemät seitsemän prosessinsuunnitteluperiaatetta ovat:

1) *Kommunikointi kasvotusten on halvin ja nopein tiedonvälityskeino*

Tiimiläisten tulisi työsennellä lähellä toisiaan, mieluiten samassa tilassa, jotta kommunikointi olisi helppoa ja nopeaa. (Cockburn 2007, 183)

2) *Ylimääräiset vaiheet ja vaihetuotteet tulevat kalliiksi*

Vaihetuotteiden tarve vaihtelee projektin koosta ja esimerkiksi turvallisuusvaatimuksista riippuen, mutta prosessia uudistettaessa tulisi jokaisen vaiheen ja vaihetuotteen kohdalta pohtia onko sen tuoma todellinen hyöty suhteessa sen vaatimaan työmäärään. (Cockburn 2007, 183-184)

3) *Henkilömäärältään suuremmat tiimit vaativat raskaamman ja kontrolloidumman prosessin*

Kun projektiryhmän henkilömäärä nousee useisiin kymmeniin, eivät kaikki 4-6 hengen tiimeissä hyvin toimivat menetelmät enää toimikaan, vaan joudutaan käyttämään raskaampia menetelmiä. Esimerkiksi kommunikoinnissa ei enää voida luottaa pelkkään ensimmäisessä periaatteessa mainostettuun kasvotusten keskustelemiseen, vaan kommunikointikanavia on määriteltävä tarkoin, jotta ryhmäläiset pysyvät ajan tasalla toistensa tekemisistä. (Cockburn 2007, 185)

4) *Kriittisemmät projektit vaativat tiukemman ja tarkemmin määritellyn prosessin*

Cockburn (2007, 186-187) jakaa projektien kriittisyyden neljään kategoriaan viasta koituvien mahdollisten seurausten mukaan.

Lievin kategoria on mukavuuden menettäminen. Tähän kategoriaan menisi esimerkiksi kahviautomaatti, joka vian vuoksi jättääkin maitokahvin tilanneelta maidon pois. (Cockburn 2007, 186)

Seuraavat kategoriat ovat taloudellinen tappio ja merkittävä taloudellinen tappio. Jälkimmäiseen kategoriaan menevät vain sellaiset projektit, joissa vika voi aiheuttaa pahimmassa tapauksessa jopa konkursseja. Tällaiset projektit ovat hyvin harvinaisia. Esimerkiksi virhe laskutuksessa aiheuttaisi varmasti kuluja, mutta kulut eivät tulisi olemaan niin mittavia, että koko yrityksen tulevaisuus olisi vaakalaudalla. (Cockburn 2007, 186)

Äärimmäisin kriittisyyden kategoria on vika, jonka seurauksena voi olla kuolema. Tähän kategoriaan voisi mennä vaikkapa hävittäjän ohjausjärjestelmä (Cockburn 2007, 186)

Koska vioista aiheutuvat seuraukset kasvavat projektin kriittisyyden mukaan, on vikojen määrää saatava vähennettyä samassa suhteessa. Vikojen vähentäminen taas vaatii valvontaa ja tiukemman prosessin. (Cockburn 2007, 185-186)

5) *Palautteen ja kommunikoinnin lisääminen vähentää vaihetuotteiden tarvetta*

Jos projektitiimi toimittaa toimivan inkrementin lopullisesta ohjelmistosta määräajoin, säilyy asiakkaalla selkeä käsitys projektin suunnasta. Lisäksi Inkrementeistä annetun palautteen perusteella projektitiimi voi entistä paremmin ymmärtää asiakkaan tarpeita. Nämä seikat vähentävät esimerkiksi projektiaikaisten katselmointien tarvetta. (Cockburn 2007, 187-188)

Osa vaihetuotteista on tarkoitettu vain projektitiimin keskinäiseen tiedonvälitykseen. Mikäli projektitiimit ovat riittävän pieniä ja työskentelevät lähellä toisiaan, on keskinäinen kommunikointi helppoa, eikä keskinäiseen tiedonjakoon tarvita vaihetuotteita. (Cockburn 2007, 188)

6) *Omaksutut työskentelytavat, taidot ja ymmärtäminen mieluummin kuin prosessit, kaavamaisuus ja dokumentit*

Tällä periaatteella Cockburn haluaa muistuttaa, ettei prosessilta pidä odottaa mahdottomia. Prosessien ja kaavojen mukaan toimimisella on oma tarkoituksena, mutta ne eivät koskaan korvaa työntekijöiden taidon tarvetta. (Cockburn 2007, 188-189).

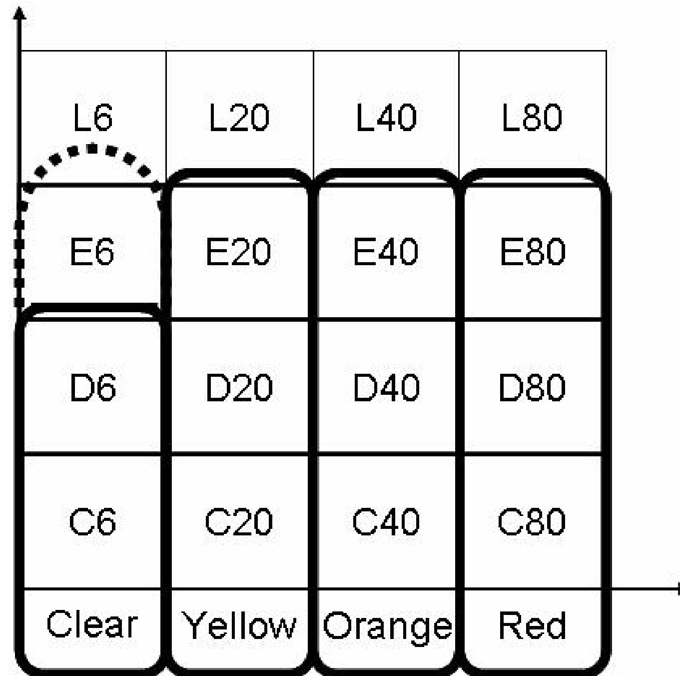
Dokumentit ovat yksi keino tiedon välittämiseksi, mutta niin kutsuttua piilevää tietoa – eli tietoa, jota ihminen ei osaa pukea sanoiksi – ei niillä voi välittää, vaan siihen tarvitaan sosiaalista vuorovaikutusta. (Cockburn 2007, 189; Burlton 2001, 74; Ruohotie 1998, 10-11, 20).

7) *Tehokkuudesta voi tinkiä toimissa, jotka eivät voi muodostua pullonkaulaksi*

Tehokkuudesta tinkimisellä ei tässä yhteydessä tarkoiteta puoliteholla työskentelemistä, vaan sellaista tilannetta, jossa työ on mahdollista tehdä ensin nopeasti ja korjailla sitä myöhemmien tarpeen vaatiessa. Tiimien tulee keskittyä sellaisten toimien tehokkaaseen suorittamiseen, joista voi kehittyä projektin pullonkaula. Muissa toimissa voidaan käyttää aikaa saman työn tekemiseen joiltain osin uudelleen, eikä tämä Cockburnin mukaan hidasta projektin kokonaisnopeutta. (Cockburn 2007, 191-192)

3.4.2 Menetelmän valitseminen

Erilaisia projekteja varten tulisi valita kooltaan ja turvallisuusvaativuuksiltaan nimenomaan omaa projektia vastaava Crystal-menetelmä (Cockburn 2007, 337-338). Teoriatasolla hyvin monenlaisia projekteja on tuettu (kuvio 11).



KUVIO 11 *Crystal-menetelmien nimet ja kohdeprojektit* (Cockburn 2007, 338)

Kun projektin henkilömäärä kasvaa, liikutaan taulukossa vasemmalta oikealle. Numerot osoittavat projektitiimin henkilömäärää. Jos siis projektitiimissä on jäseniä kuusi tai vähemmän, kuten agile-menetelmissä usein suositellaan, kannatta valita Crystal Clear, jos jäsenmäärä on 20, kannattaa turvautua Crystal Yellow:hun, ja niin edelleen. (Cockburn 2007, 338)

Vastaavasti kun projektin kriittisyys kasvaa, liikutaan taulukossa alhaalta ylöspäin. Kriittisyyden suhteen käytetään perusperiaate 4:n yhteydessä kuvattua vikojen mahdollisesti aiheuttamien seurausten mukaan tehtyä luokittelua: C=Comfort (mukavuuden menettäminen), D=Discretionary monies (taloudelliset

tappiot), E=Essential monies (Merkittävä taloudellinen tappio) ja L=Life-critical (vika voi johtaa jopa kuolemaan). (Cockburn 2007, 338)

Ainoa ongelma tässä hienossa kaaviossa on se, että todellisuudessa Crystal-menetelmiä on suunniteltu ja julkaistu vain kolme: Crystal Clear, Crystal Orange sekä Crystal Orange Web. Näistäkin ainoastaan Clearista on julkaistu kirja. Eli mikäli sattuu tarvitsemaan prosessimallia yli 40-henkiselle projektiryhmälle, Crystal-menetelmät eivät lupauksista huolimatta voi ainakaan tällä hetkellä auttaa. (Cockburn 2007, 335, 358)

3.4.3 Crystal Clear

Crystal Clear on Crystal-menetelmistä pisimmälle kehitetty. Se on tarkoitettu 6-10 hengen tiimeille, jotka työskentelevät projekteissa, joissa viat voivat aiheuttaa taloudellisia tappioita, eli Crystal-kategorioita käyttäen D6-projekteille. Menetelmää voidaan kuitenkin soveltaa myös 8-10-hengen tiimeille sekä E-kategoriaan (vakavat taloudelliset tappiot) lisäämällä prosessiin kommunikointiin liittyviä vaatimuksia sekä testausta. (Cockburn 2007, 340)

Crystal Clear-tiimissä on oltava ainakin budjettivastaava, vanhempi ohjelmistosuunnittelija, ohjelmistosuunnittelija sekä käyttäjäryhmän edustaja (edustaja saa olla osa-aikainen). Tarvitaan myös liiketaloudellisen alueen asiantuntija sekä vaatimusten kirjoittajia, mutta ne voivat olla samoja henkilöitä kuin edellä mainitut. (Cockburn 2007, 340)

Varsinaista prosessia ei ole määritelty, sen voi jokainen projekti määritellä itse annettujen standardien puitteissa:

- Ohjelmistoa on toimitettava säännöllisesti, 2-3 kk välein
- Edistymistä mitataan virstanpylväiden avulla, joita voivat olla esimerkiksi ohjelmistojulkaisut. Virstanpylväs ei kuitenkaan voi olla mikään dokumentti.
- Automaattista testausta on oltava jonkin verran

- Käyttäjän edustajan on oltava mukana projektissa. Käyttäjäkatselmointeja järjestetään kaksi jokaista julkaisua kohden
- Tuote- ja menetelmäkehitystyöpajoja pidetään jokaisen inkrementin alussa, lopussa sekä puolivälissä

(Cockburn 2007, 340-341)

Jokainen määritelty standardi on siinä mielessä pakollinen, että mitään niistä ei saa jättä pois prosessista. Standardien tilalle voidaan kuitenkin vaihtaa jokin muu menetelmä, mikäli kyseinen standardi on täysin korvattavissa vaihtoehtoisella menetelmällä.
(Cockburn 2007, 341)

Prosessin vaihetuotteita ovat julkaisukaavio ja -aikataulu, katselmointiaikataulu, tarvittavat suunnitteludokumentit, rajapintojen kuvaukset, testitapaukset, käyttöohje (tarvittaessa) sekä tietenkin toimiva ohjelmakoodi. Projektiaikaista dokumentaatiota on siis oltava, mutta Crystal Clear ei määrittele dokumenttien sisältöä eikä toteutustapaa, ne asiat voi projektitiimi päättää itse (Cockburn 2007, 341).

Crystal Clear määrittelee siis vain sääntöjä, joiden mukaan kukin projekti voi itse kehittää sopivan ohjelmistoprosessin tapauskohtaisesti. Mitään työskentelytapoja ei vaadita käytettävän eikä mitään prosessin vaiheita määritellä. Kaikki on räätälöitävissä. (Cockburn 2007, 340-342, 358-359)

3.4.4 Crystal Orange

Crystal Orange on suunniteltu D40-projekteille, eli alle 40-henkiselle projektiryhmälle, jonka jäsenet työskentelevät samoissa tiloissa, ja jossa järjestelmäviat voivat aiheuttaa taloudellisia tappioita. (Cockburn 2007, 342)

Menetelmämalli on kevyt, jos huomioidaan, että se toimii 40-henkisissäkin projekteissa, mutta se on silti sen verran raskas, ettei sitä kannata soveltaa alle 10-henkisiin projekteihin. (Cockburn 2007, 344)

Crystal Orange määrittelee 7 tiimiä, joista yksi vastaa projektin valvonnasta ja yksi keskittyy infrastruktuuriin. Teknisestä suunnittelusta vastaavat järjestelmäsuunnittelu-, arkkitehtuuri- sekä teknologiatiimit. Lopullisesta toteutuksesta vastaavat toimintotiimi sekä testaustiimi. (Cockburn 2007, 343)

Rooleja on määritelty 14. Päävastuu on luonnollisesti projektipäälliköllä. Budjetista ja liiketaloudellisesta puolesta huolehtivat budjettivastaava sekä liiketaloudellisen alueen asiantuntija ja analysoija. Ohjelmiston rakenteesta vastaamaan on määritelty 4 roolia arkkitehdistä teknisen alueen asiantuntijaan. Käytettävyysasioista vastaavat pääosin käytettävyysasiantuntija sekä käyttöliittymäsuunnittelija. Toteutuksesta vastaavat johtava ohjelmistosuunnittelija sekä muut ohjelmistosuunnittelijat ja testaaja. Kirjalliset vaihetuotteet ovat dokumentoijan vastuulla. (Cockburn 2007, 343)

Vaihetuotteita on määritelty 11, joistain tärkein on tietenkin toimiva ohjelmakoodi. Muita vaihetuotteita ovat tilanneraportit, aikataulutukset sekä erilaiset tekniset dokumentit, kuten vaatimus- ja käyttöliittymämäärittelyt sekä teknistä toteutusta kuvaava mallinnukset. (Cockburn 2007, 343)

Pakolliset standardit ovat samat kuin Crystal Clearissa sillä erotuksella, että iteraatiojaksot saavat olla jopa 3-4 kk pitkiä. Myös Crystal Orangessa korvaavia menetelmiä saa ottaa käyttöön, mutta mitään aihealuetta ei saa jättää pois. (Cockburn 2007, 344)

Mitään työskentelytapoja tai prosessin vaiheita ei jälleen määritellä, vaan luetellaan vain sääntöjä ja periaatteita, joiden perusteella projekti voi itse kehittää sopivan ohjelmistoprosessin. (Cockburn 2007, 343-344)

3.4.5 Crystal Orange Web

Crystal Orange Web on alun perin kehitetty yritykselle, joka toimitti jatkuvana prosessina ohjelmistoa internetiin. Crystal Orange Web eroaa siis Crystal

Orangesta siten, että se on suunniteltu jatkuvaan ohjelmistokehitykseen, eikä yksittäisen projektin toteuttamiseen. (Cockburn 2007, 344)

Mentelmä pohjautuu neljälle periaatteelle:

1) *Jatkuva, nopea toimittaminen*

Tässä prosessimallissa iteraatiot ovat todella lyhyitä, vain kahden viikon mittaisia. Jokaisen iteraatiojakson jälkeen pidetään itsearviointityöpaja toiminnan kehittämiseksi. (Cockburn 2007, 345-346)

2) *Keskittyminen projektin etenemiseen ja häiriöiden poistaminen*

Tavoitteena on varmistaa, että jokainen projektitiimiläinen tekee sitä, mikä on yrityksen näkökulmasta hyödyllisintä. Työtehtävät on jaettu niin lyhyisiin kokonaisuuksiin, että ne voidaan toteuttaa kahden viikon iteraatiojaksoissa. (Cockburn 2007, 347)

3) *Vikojen määrä mahdollisimman vähäiseksi*

Koska iteraatiojaksot ovat lyhyitä, vie vikojen korjaaminen merkittävästi aikaa ja resursseja jatkokehitykseltä. Tästä syystä laatuun panostetaan jatkuvasti, jotta vikoja jäisi julkaisuun mahdollisimman vähän. Apuna vikojen löytämisessä toteutusvaiheen aikana käytetään täysin automatisoitua regressiotestausta. (Cockburn 2007, 347)

4) *Yhteinen päämäärä*

Kaikille osallisille tehdään selväksi projektin ja yrityksen päämäärä. Tähän tavoitteeseen pyritään pääsemään jakamalla tiimeihin osaajia eri osa-alueilta. Kun tiimissä on mukana niin ohjelmoijia ja testaajia kuin liiketaloudellisten näkökulmien tuntijoitakin, saadaan tietoa välitettyä puolin ja toisin nopeasti. (Cockburn 2007, 348)

Prosessimallissa on tarkoituksella jätetty prosessin ja vaihetuotteiden rooli pieneksi ja keskitytty projektin etenemiseen (Cockburn 2007, 348). Tämä lähestymistapa on omituinen, sillä tarkoituksena on esitellä malli, jonka pohjalta prosessi voidaan räätälöidä omiin tarpeisiin. Ilman prosessin tai vaihetuotteiden määrittelyä Crystal Orange Web on siis käytännössä vain neljä periaatetta – kaikki muu on jokaisen projektin itse päätettävissä.

3.4.6 Yhteenveto ja arviointi

Crystal-menetelmät tarjoavat vain muutamia periaatteita, nykymuodossaan kolmenlaisille projekteille. Todella paljon jätetään projektitiimin suunnittelukyvyn varaan. Crystal Clearin yhteydessä luetellaan muutamia työskentelytekniikoitakin, joista joitakin projektitiimi voisi kenties nähdä aiheelliseksi kokeilla, mutta niistäkään ei tarjota kuin yleisen tason kuvaus. Niidenkin osalta projektitiimin on kaiketi tarkoitus etsiä lisätietoja oma-aloitteisesti, jos kiinnostus heräsi (Cockburn 2007, 357).

Crystal-menelmien lähtökohta, eli räätälöitävien prosessimallien tarjoaminen prosessisuunnittelun avuksi, kuulostaa ajatuksen tasolla toimivalta, mutta käytännössä Crystal-menetelmistä jää kovin keskeneräinen vaikutus. Ohjeistus tuntuu olevan karrikoituna verrattavissa kuvitteelliseen kakunteko-ohjeeseen, olkoon se nimeltään vaikkapa ”Crystal-cake”-menetelmä:
Perusosat:
1) käytä sokeria
2) käytä munia.
Kukin leipuri voi lisätä ainesosia halutessaan, mutta mitään ei saa poistaa.
Ainesosan saa myös korvata, mikäli täysin vastaava ainesosa on olemassa.
Varsinainen kakuntekoprosessi jätetään kunkin leipurin omaan harkintaan.
Näillä ohjeilla varmasti syntyy hyviä kakkuja – mikäli menetelmää soveltava leipuri sattuu oleman sellaisten tekemisessä erityisen hyvä.

Vastaavasti pätevä prosessisuunnittelija onnistuu varmasti kehittämään onnistuneen ohjelmistoprosessin Crystalin määrittelemiä periaatteita noudattaen – mutta hän pystyisi todennäköisesti samaan myös sen tutkimatta kyseisiä periaatteita. Toisaalta kokematon prosessisuunnittelija tuskin onnistuisi noin suurpiirteisten ohjeiden perusteella kehittämään toimivaa prosessia omaa projektiaan varten, vaan tarvitsisi valtavan määrän lisätietoa prosessisuunnittelusta.

Samaan tarkoitukseen kuin Crystal on myös suunniteltu menetelmä nimeltä *Rational Unified Process* (RUP, Kroll&Kruchten 2004). Koska RUP ei kuulu alkuperäisiin agile-menetelmiin, on se rajattu tämän tutkimuksen ulkopuolelle, mutta pikaisen tutustumisen perusteella se on kattavampi ja paremmin määritelty menetelmä kuin Crystal, joten tiettyjen määräysten perusteella oman prosessin luomisesta kiinnostuneen kannattaa paneutua myös siihen.

3.5 Feature-Driven Development (FDD)

Feature-Driven Development (FDD) on alun perin tehty olio-ohjelmointiprojektien prosessimalliksi, mutta ajan myötä se on kehittynyt kokonaisvaltaiseksi agile-menetelmäksi (Pressman 2005, 120).

FDD:lle ominaista ovat lyhyet, kahden viikon iteraatiojaksot, joissa kaikissa toteutetaan asiakkaalle lisäarvoa tuovaa toiminnallisuutta. Kun toimintaa ohjaavat nimenomaan ominaisuudet, voidaan projektin eteneminen ilmaista luotettavasti prosenttilukuina, mikä tietenkin miellyttää sekä johtoporrasta että asiakasta. Kun esimerkiksi 45 % ominaisuuksista on toteutettu, FDD-projekti on *todella* 45 %:sti valmis. Vastaavanlainen etenemisen *laskeminen* ei perinteisissä projektimalleissa ole mahdollinen, vaan etenemistä voidaan yleensä vain arvioida. (Coad 1999, 183-184; Pressman 2005, 120)

Nimensä mukaisesti FDD:n kantava ajatus on, että projektia ohjaa lista toteuttavista ominaisuuksista (engl. *feature*). Ominaisuus on ”asiakkaan näkökulmasta hyödyllinen tuotos, jonka toteuttamiseen kuluu enintään kaksi

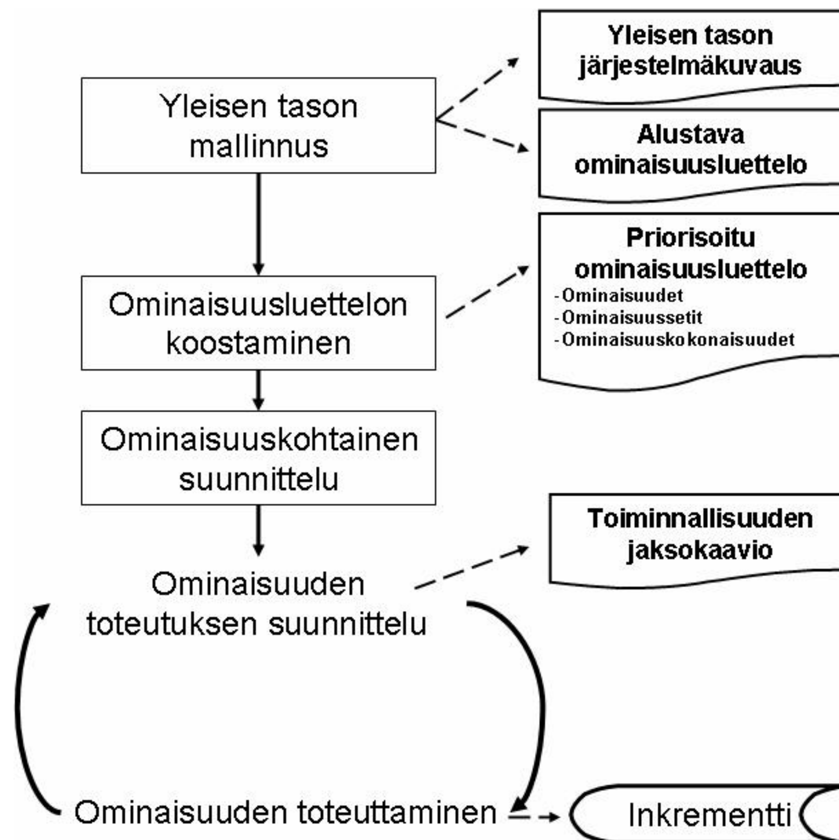
viikkoa”. Ominaisuus voidaan kuvata kaavalla: *<suoritettava toiminto> <kohde: kuka/mikä>n <kohteen täsmennys>*, esimerkiksi: "Laske myynnin loppusumma" (Coad 1999, 184-185)

Ominaisuuksista kootaan suuremman kokonaisuuden toteuttavia *ominaisuussettejä* (Feature Set), joiden kaava on *<kohde>n <tekeminen>*, esimerkiksi ”Laskun lähettäminen”. (Coad 1999, 185)

Ominaisuusseteistä voidaan koota edelleen *ominaisuuskokonaisuuksia* (Major Feature Set), joiden kaava on *<kohde>n hallinta*, esimerkiksi ”Laskutuksen hallinta”. (Coad 1999, 185)

3.5.1 FDD-prosessi

FDD-prosessi (kuvio 12) on jaettu viiteen aliprosessiin, joista kolme ensimmäistä toteutetaan jaksoittain ja kahta viimeistä toistetaan sykleissä. (Coad 1999, 190)



KUVIO 12 *FDD-Prosessi* (Coad 1999, 190)

Aliprosessi 1: Yleisen tason mallinnus

Ensimmäinen aliprosessi on *yleisen tason mallinnus* (Overall Model Shape), jossa kuvaillaan järjestelmä yleisellä tasolla sekä koostetaan alustava ominaisuusluettelo. Aliprosessiin osallistuvat asiakasryhmän edustajia sekä projektitiimi. Olio-ohjelmointiprojekteissa tämän aliprosessin aikana työstetään myös luokkakaavioita. (Coad 1999, 191)

Aliprosessi 2: Ominaisuusluettelon koostaminen

Seuraavan aliprosessin, *ominaisuusluettelon koostamisen*, sisäänmenokriteerit ovat edellisen aliprosessin vaihetuotteet, eli yleisen tason järjestelmäkuvaus sekä alustava ominaisuusluettelo. Tässä aliprosessissa projektitiimi määrittelee tarkemmin alustavassa ominaisuusluettelossa listatut ominaisuudet ja muodostaa niistä ominaisuussettejä sekä – kokonaisuuksia. Monimutkaiset ominaisuudet jaetaan sopivan kokoisiksi, työmäärältään enintään kahden viikon kestoisiksi kokonaisuuksiksi. Tämän jälkeen ominaisuudet, ominaisuussetit sekä ominaisuuskokonaisuudet priorisoidaan, jotta asiakkaan kannalta tärkein toiminnallisuus olisi mahdollista toteuttaa ensin. (Coad 1999, 192)

Aliprosessi 3: Ominaisuuskohtainen suunnittelu

Kolmas aliprosessi on *ominaisuuskohtainen suunnittelu* (Plan By Feature), jossa projektipäällikkö sekä joukko vanhempia ohjelmistosuunnittelijoita projektitiimistä määrittelevät ominaisuuksien sekä ominaisuussettien ja – kokonaisuuksien toteuttamisjärjestyksen ja arvioivat niiden alustavat valmistumisajankohdat. Tässä vaiheessa määritellään olio-ohjelmointiprojekteissa myös luokkaomistajuudet ja muunlaisissa projekteissa ominaisuusvastuulliset. Ominaisuuskokonaisuuksien vastuullisiksi määrätään vanhempia ohjelmistosuunnittelijoita. (Coad 1999, 193)

Aliprosessi 4: Ominaisuuden toteutuksen suunnittelu

Seuraavaksi prioriteettilistan kärjestä valitaan toteutettava ominaisuus ja sen toteutus suunnitellaan *ominaisuuden toteutuksen suunnittelu*-aliprosessissa (Design By Feature, DBF). Vanhempi ohjelmistosuunnittelija, jonka vastuulle määrättyyn ominaisuuskokonaisuuteen valittu ominaisuus kuuluu, sekä tietenkin

ominaisuuden toteutuksesta vastaavat tiimiläiset muodostavat jaksokaavion ominaisuuden toiminnasta sekä sen yhteyksistä muuhun järjestelmään. (Coad 1999, 194)

Aliprosessi 5: Ominaisuuden toteuttaminen

Viimeinen aliprosessi on *ominaisuuden toteuttaminen* (Build By Feature, BBF), jossa ominaisuus toteutetaan, integroidaan lopulliseen järjestelmään ja testataan. Prosessin ulosmenokriteeri on toimiva ohjelmisto, jossa on mukana uusi, asiakkaalle lisäarvoa tuova ominaisuus. (Coad 1999, 195, 184)

Kahta viimeistä aliprosessia, ominaisuuden toteutuksen suunnittelua ja toteuttamista, toistetaan kunnes kaikki ominaisuusluettelon ominaisuudet on toteutettu. (Coad 1999, 184)

3.5.2 Skaalautuvuus

FDD on suunniteltu suuremmille projekteille, joten projektissa oletetaankin olevan useita vanhempien ohjelmistosuunnittelijoiden (chief programmer) vetämiä tiimejä. Jokainen tiimi voi toteuttaa toiminnallisuutta ominaisuus kerrallaan samanaikaisesti muiden tiimien kanssa. Samansuuntaista skaalaamismenetelmää on ehdotettu muidenkin agile-menetelmien yhteydessä, mutta FDD:ssä useamman tiimin käyttämiselle on todelliset edellytykset kattavan suunnitteluvaiheen (aliprosessit 1-3) ansiosta. (Coad 1999, 196, 191-193; Pearson 2005, 120)

3.5.3 Yhteenveto ja arviointi

FDD keskittyy ASD:n tavoin toteutusvaiheen yksityiskohtien sijaan projektin johdollisiin seikkoihin. Mitään ohjelmointitekniikoita ei määritellä tai vaadita käyttämään, vaan sellaiset seikat jätetään projektitiimin päätettäväksi, mutta itse prosessi määritellään silti riittävällä tarkkuudella.

FDD:n hyödyt ovat itsenäisten tiimien toteuttaman iteratiivisen ja inkrementaalisen työskentelytavan tuomia etuja.

Luokka- tai ominaisuusomistajuus katsotaan FDD:ssä tarpeelliseksi. Tätä toimintamallia perustellaan muun muassa sillä, että se tuo loogista yhtenäisyyttä ohjelmakoodiin (Coad 1999, 196). Tässä menetelmässä on kuitenkin myös merkittävä varjopuolensa. Astels, Granville ja Novak (2002, 11-12) muistuttavat, että luokkaomistajuus johtaa kapeaan asiantuntijuuteen, eli tilanteeseen, jossa tiimiläinen tuntee hyvin pienen osan ohjelmistosta hyvin, muttei toisia osia lainkaan, eikä kenellekään muodostu asiantuntijuutta kokonaisuudesta.

FDD:ssä prosessien kestoja ei määritellä lukuun ottamatta ominaisuuden toteuttamisvaihetta (aliprosessi 5). Tämä seikka vie uskottavuutta FDD:n ennustettavuusmainostukselta (Coad 1999, 183-184). Vaikka projektin valmiusaste onkin helposti laskettavissa prosentteina, projektin jäljellä olevaa kestoa on kuitenkin vaikea arvioida, koska esimerkiksi aliprosessi 4:n kestoa – tai edes sitä, onko se aikarajoitettu (engl. time-boxed) – ei kerrota. Iteratiivinen ohjelmistokehitys ei toimi, mikäli aikataulun venyminen on mahdollista, joten sopii olettaa, että iteraatiot ovat aikarajoitettuja, mutta sitä ei erikseen mainita.

FDD:n skaalautuvuus kuulostaa toimivalta, kuten tietysti sopii odottaakin nimenomaan suurempiin järjestelmiin suunnitellulta prosessilta.

Asiakkaan roolia ei korosteta, mikä on mielestäni tuotteen lopputuloksen oikeellisuuden kannalta vakava puute. FDD:n prosessikuvauksesta jää epäselväksi näkevätkö asiakkaat iteraatioiden tuloksia ennen koko projektin päättymistä. Iteratiivinen ja inkrementaalinen ohjelmistokehitys helpottavat tehtävän hahmottamista ja nopeuttavat tuotekehitystyötä vaikkei tuotoksia esiteltäisikään asiakkaalle, mutta parhaan hyödyn menetelmästä saa vasta silloin, kun asiakas saadaan osallistumaan iteraatioiden tulosten julkistamistilaisuuksiin ja voi näin ohjata projektia oikeaan suuntaan.

3.6 Dynamic Systems Development Method (DSDM)

DSDM on alun perin RAD:in (Rapid Application Development) (Martin 1991) pohjalta kehitetty prosessi, jonka on julkaissut DSDM Consortium. Consortiumin perusti vuonna 1994 17 erikokoista yritystä, mukana pienempiä IT-alan yrityksiä sekä merkittäviä, kuten IBM ja Oracle. Nykyään jäseniä on yli 1000 ja DSDM:stä on tullut ainakin Iso-Britanniassa de facto- standardi RAD-tyyppisissä projekteissa (Stapleton 1997, xi, xv)

Perusajatukset DSDM:ssä ovat samansuuntaisia kuin muissakin agile-menetelmissä. Itseohjautuvien, riittävät valtuudet työnsä tekemiseen omaavien tiimien tärkeyttä korostetaan, samoin liiketaloudellisen näkökulman huomioimista sekä asiakkaan osallistumista tuotekehitysprojektiin. Iteratiivista ja inkrementaalista tuotekehitystä luonnollisesti suositaan ja voimavarojen kohdentamista asiakkaan näkökulmasta tärkeimpien tehtävien tekemiseen korostetaan. (Stapleton 1997, xii-xiv)

DSDM:ssä toimintaa ohjaa myös ajattelutapa, jonka mukaan vaihetuotteiden ei tarvitse olla täydellisiä, vaan ainoastaan *riittävän* valmiita. Vaihetuotteita työstetään näin ollen aina vain siihen saakka, että siirtyminen prosessin seuraavaan vaiheeseen on mahdollista (Pressman 2005, 116; Stapleton 1997, 5).

3.6.1 DSDM:n periaatteet

DSDM-prosessia ohjaa yhdeksän periaatetta. Jotta DSDM toimisi, on kaikkia periaatteita noudatettava. (Stapleton 1997, xvi, 11).

1) Loppukäyttäjä aktiivisesti mukana projektissa

Perinteisissä menetelmissä loppukäyttäjä on mukana vain projektin alussa tekemässä vaatimusmäärittelyjä sekä lopussa hyväksyntätesteissä tai

loppukatselmoinnissa. DSDM-projektissa muutaman loppukäyttäjäryhmän edustajan tulisi olla mahdollisimman helposti tavoitettavissa koko projektin ajan. Näillä henkilöillä tulee olla myös jonkin verran teknistä osaamista. Ihannetilanteessa loppukäyttäjä on täysipainoinen projektitiimin jäsen, mutta sellainen järjestely ei aina ole käytännössä mahdollista. (Stapleton 1997, 11-12, 49)

2) *Tiimeillä tulee olla riittävästi valtaa toimintaansa liittyvien päätösten tekemiseen*

Jokaisen pienenkin päätöksen hyväksyttäminen useammalla organisaatiotasolla on raskasta ja hidasta. Projektin nopea eteneminen on mahdollista vain siten, että projektitiimi voi tehdä nopeita päätöksiä tekemisensä suhteen. (Stapleton 1997,13)

3) *Toimita näkyviä tuloksia usein*

DSDM:ssä toimitetaan jotain näkyvää ja käyttökelpoista – oli se sitten ohjelmistoa tai vaikkapa vuokaavioita – ennalta määritellyin väliajoin. Toimitusten tulee tapahtua mahdollisimman usein, jopa viikoittain. Tällä pyritään palautteen saamiseen ja sen mukaisten korjaustoimenpiteiden toteuttamiseen mahdollisimman nopeasti ja usein. (Stapleton 1997, 14)

4) *Vaihetuotteiden merkittävyys liiketaloudellisesta näkökulmasta tärkeintä*

Tällä periaatteella tarkoitetaan sitä, että on tärkeämpää toimittaa asiakkaan näkökulmasta tärkeitä ominaisuuksia, kuin keskittyä toimittamaan aivan kaikki alkuperäisessä vaatimusmäärittelyssä mieleen juolahtaneet ominaisuudet. (Stapleton 1997, 15)

5) Iteratiivinen ja Inkrementaalinen ohjelmistokehitys

Iteratiivista ohjelmistokehitystä käytettäessä on mahdollista toimittaa loppukäyttäjälle toimivia lopullisen ohjelmiston osia usein ja nopeasti. Näin loppukäyttäjä voi ohjata projektin suuntaa helpommin. (Stapleton 1997, 15)

6) Muutosten on oltava peruttavissa

Mikäli DSDM:n periaatteita on noudatettu, projektin pitäisi pysyä oikeassa suunnassa. Mikäli jossain vaiheessa kuitenkin ilmenee, että projekti on harhautunut päätavoitteistaan, on oltava mahdollista palata taaksepäin sopivaan tilanteeseen. Tämän periaatteen toteuttaminen vaatii ajantasaista dokumentaatiota sekä tietenkin hyvän versionhallintajärjestelmän. (Stapleton 1997, 16)

7) Vaatimukset kirjataan yleisellä tasolla

Tämä periaatteen tavoitteena on vaatimusten kirjaaminen jo aikaisessa vaiheessa, jotta vaatimukset voivat ohjata projektia. Alkuvaiheessa ei kuitenkaan ole mahdollista tietää kaikkia vaatimusten yksityiskohtia, joten ne on kirjattava yleisellä tasolla. (Stapleton 1997, 16)

8) Testaus jatkuvaa koko projektin ajan

DSDM-prosessissa ei erikseen määritellä testausvaihetta, koska testausta on tarkoitus tehdä jatkuvasti koko projektin keston ajan, eikä vain projektin tai inkrementin lopuksi. (Stapleton 1997, 8,16-17)

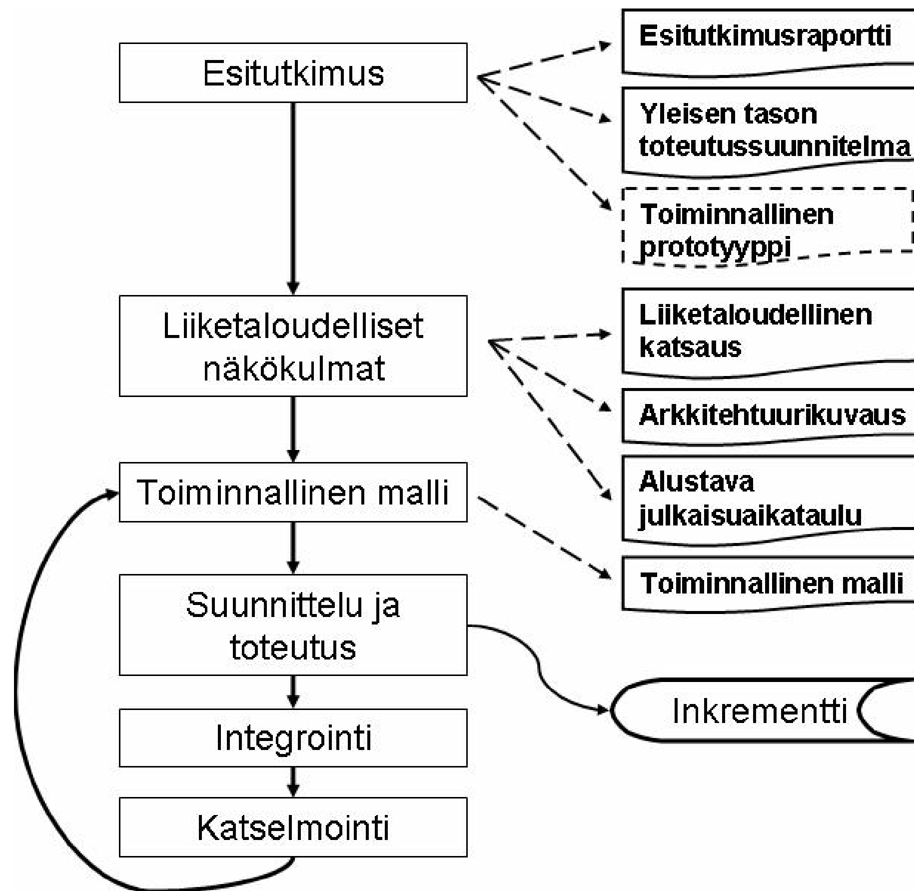
9) Yhteistyö osapuolien kesken

Vastakkainasettelu eri osapuolien kesken hidastaa projektia ja vaikeuttaa jokaisen osallisen työntekoa. Tällä periaatteella pyritään välttämään tuota vastakkainasettelua, koskien niin asiakas-toimittaja- suhdetta kuin myös

esimerkiksi toimittaja-alihankkijasuhteita. Molemminpuolinen luottamus sekä yhteistyöhalukkuus ovat ratkaisevia tekijöitä projektin menestymistä ajatellen. (Stapleton 1997, 17-18)

3.6.2 DSDM-prosessi

DSDM-prosessissa (kuvio 13) on viisi vaihetta, joista kaksi ensimmäistä ovat jaksottaisia, ja muiden on tarkoitus olla ainakin osin päällekkäisiä, joskin DSDM sallii myös niiden suorittamisen vaiheittain tarpeen vaatiessa. (Stapleton 1997, 4)



KUVIO 13 *DSDM-prosessi* (Stapleton 1997, 3)

Prosessissa määritellyt vaihetuotteet ovat pakollisia siinä mielessä, että ne on kaikki tuotettava, mutta niiden sisällöt sekä toteutustapa ovat vapaasti päätettävissä. Mikäli jokin projektiryhmän mielestä tarpeellinen vaihetuote

puuttuu DSDM:stä, on sen lisääminen prosessiin sallittua. Vaihetuotteiden sisällön suhteen noudatetaan DSDM:n ohjausajatusta: tuotoksien on oltava *riittävällä* tasolla, eli tasolla, joka mahdollistaa prosessin seuraavan vaiheen aloittamisen. (Stapleton 1997, 4)

Esitutkimus

DSDM-projekti alkaa esitutkimusvaiheella. Vaihe on sisällöltään pääosin samanlainen kuin perinteisissä menetelmissä, mutta kestoltaan vain muutamia viikkoja. Tästä syystä tarkastelu on yleistasoisempaa kuin perinteisissä menetelmissä. Esitutkimuksessa määritellään haluttu järjestelmä yleisellä tasolla, pohditaan onko ratkaisu teknisesti mahdollinen ja onko se liiketaloudellisessa mielessä järkevä sekä tietenkin arvioidaan soveltuuko DSDM kyseiseen projektiin. (Stapleton 1997, 4; Pressman 2005, 116)

Esitutkimusvaiheen tuotokset ovat esitutkimusraportti, yleisen tason toteutussuunnitelma (Outline Plan of Development) sekä tarvittaessa pikaisesti toteutettu toimiva prototyyppi. Prototyyppiä ei yleensä tarvita esitutkimusvaiheen läpiviemiseen, mutta joissakin monimutkaisissa projekteissa se voi olla tarpeen. (Stapleton 1997, 5)

Liiketaloudellisten näkökulmien tarkastelu

Seuraava prosessin vaihe on liiketaloudellisten näkökulmien tarkastelu (Business Study). Vaihe toteutetaan sarjana työpaja-tilaisuuksia, joihin osallistuu loppukäyttäjäryhmästä sekä projektitiimistä nopeaan päätöksentekoon kykeneviä henkilöitä. Vaiheen tärkein tuotos on liiketaloudellinen katsaus, jossa määritellään ja priorisoidaan korkean tason toiminnot, jotta niistä tärkeimmät voidaan toteuttaa ensin. Muita tuotoksia ovat arkkitehtuurikuvaus sekä alustava suunnitelma prototyyppien julkaisuaikataulusta. (Stapleton 1997, 5-6)

Toiminnallinen malli

Näiden vaiheiden jälkeen aloitetaan *toiminnallisen mallin* (Functional Model) työstäminen. Toiminnallinen malli on mahdollisimman tarkka kuvaus projektin liiketaloudellisista näkökulmista. Se pitää sisällään täsmennetyn version

esitutkimusvaiheessa tehdystä toteutussuunnitelmasta sekä priorisoidun listan toteutettavista ominaisuuksista. Sellaisista vaatimuksista, jotka eivät ole toiminnallisia, tehdään erillinen lista. Toiminnalliseen malliin sisältyy myös päivitetty riskianalyysi. Toisesta inkrementistä lähtien tässä prosessin vaiheessa myös dokumentoidaan ja talletetaan aiempien inkrementtien katselmointitilaisuuksien kommentit. (Stapleton 1997, 7-8)

Suunnittelu ja toteutus
Varsinainen ohjelmointityö suoritetaan suunnittelu- ja toteutusiteraatiossa. Iteraatio on yleensä osin samanaikainen toiminnallisen mallin työstämisvaiheen kanssa. Tämän vaiheen tärkein tuotos on luonnollisesti toimiva ja testattu inkrementti lopullisesta ohjelmistosta. Vaihetuotteina syntyy tarvittava dokumentaatio inkrementteihin liittyen. Ohjelmistokehitystekniikoita ei DSDM:ssä määritellä, vaan projektiryhmä saa itse päättää menetelmät, millä iteraation työt käytännössä tehdään. (Stapleton 1997, 8-9).

Integrointi
DSDM-prosessin – tai oikeammin yhden DSDM-syklin – viimeinen vaihe on integrointi. Siinä tuotokset siirretään mahdollisesti käytetystä testiympäristöstä lopulliseen ympäristöön, ja annetaan tarvittava käyttökoulutus loppukäyttäjille. Vaiheen tärkein tuotos on uudella toiminnallisuudella päivitetty järjestelmä. Vaihetuotteina syntyy käyttäjädokumentaatiota. (Stapleton 1997, 9)

Katselmointi
Loppukatselmoinnissa sekä tutkitaan inkrementin tuloksia, että päätetään projektin jatkosta: onko riittävästi ominaisuuksia toteutettu, vai onko tarvetta uusille inkrementeille? (Stapleton 1997, 10)

3.6.3 Yhteenveto ja arviointi

DSDM:ssä on selvästikin keskitytty itse prosessiin ja sen läpiviemiseen, eikä se siksi pakota käyttämään mitään tiettyjä ohjelmistokehitystekniikoita. DSDM:n

hyödyt ovat itsenäisten tiimien toteuttaman ja loppukäyttäjän aktiivisen osallistumisen ohjaaman iteratiivisen ja inkrementaalisen työskentelytavan tuomia etuja. DSDM on piristävä poikkeus agile-menetelmien kirjossa siksi, että siitä ei kuvastu pakonomainen tarve erottua perinteisistä menetelmistä eikä tiettyjen perinteisen prosessimallin vaiheiden tarpeellisuutta pelätä tunnustaa. Tästä syystä prosessiin mukaan otettujen perinteisten menetelmien osat on kuvailtu sellaisenaan, keksimättä keinotekoisia nyanssieroja oman prosessin osien ja niitä vastaavien perinteisten menetelmien osien välille.

Prosessin läpiviemiseen keskittymisen ansiosta projektin esitutkimukseen kiinnitetään DSDM:ssä useimpia agile-menetelmiä enemmän huomiota. Vaiheet (esitutkimus sekä liiketaloudellisten näkökulmien tarkastelu) ovat kuitenkin riittävän keveitä, eikä niissä yritetäkään saada aikaan täydellisiä, vaan ainoastaan yleistasoisia suunnitelmia. Hyvänä piirteenä DSDM:ssä pidän myös testauksen huomioimista ja sen selkeää yhdistämistä prosessiin.

Toiminnallisen mallin tekemisen hyödyt sen sijaan jäävät hieman epäselviksi. Prosessivaiheen tavoitteena on esitutkimusvaiheen sekä liiketaloudellisen tarkastelun aikana syntyneiden vaihetuotteiden päivittäminen. Kuulostaa mielestäni monimutkaiselta tehdä ensin yleistasoiset suunnitelmat ja sen jälkeen käynnistää uusi prosessin vaihe, jossa suunnitelmista tehdään hieman täsmällisempiä, muttei silti vieläkään erityisen täsmällisiä. Voisi olettaa, että samaan lopputulokseen päästäisiin tekemällä esitutkimusvaiheen ja liiketaloudellisten näkökulmien tarkastelun aikana syntyneistä vaihetuotteista saman tien riittävän yksityiskohtaisia sekä päivittämällä ja täsmentämällä niitä tarpeen vaatiessa.

DSDM vaatii loppukäyttäjältä merkittävää roolia, mikä voi osoittautua käytännössä hankalaksi toteuttaa – mutta tämä ongelma koskee toki myös kaikkia muita agile-menetelmiä.

4. TULOKSET

4.1 Lähdekritiikki

Agile-menetelmien kuvauksista on selvästi havaittavissa menetelmien kehittäjien tuskastuminen vesiputousmallisiin projekteihin. Valitettavasti osa heistä ei ole tästä syystä kyennyt tämän arvioimaan vesiputousmallista ohjelmistoprosessia objektiivisesti, vaan on valinnut myyntipropagandatyylisen ”kilpailevan menetelmän” epäkohtia ja oman menetelmän etuja vahvasti liioittelevan kirjoitustyylin. Yksi kirjoittajista sortui jopa räikeään ”Keisarin uudet vaatteet” -tyyliseen myyntipuheeseen, jossa ilmoitetaan, että mikäli lukija ei sisäistä kirjoittajan hienoa prosessimallia, on lukijan ammattitaitoa syytä epäillä: *”Aloittelijatiimit eivät löydä selkeää noudatettavaa kaavaa tutkiessaan Crystal-menetelmiä, ja siirtyvät käyttämään jotain muita menetelmiä, kuten Scrumia tai XP:tä. Kokeneet tiimit - - pitävät Crystalista, koska se tarjoaa juuri riittävän määrän sääntöjä ”* (Cockburn 2007, 354).

Myyntipuhetyylisistä kirjoituksista on joskus vaikea löytää esitellyn menetelmän todellisia hyötyjä tai edes menetelmän vaiheiden yksityiskohtaista kuvausta.

4.2 Agile-menetelmien vertailua

Kuviossa 14 on esitelty eri agile-menetelmien tärkeimpiä piirteitä.

	Scrum	XP	ASD	Crystal	FDD	DSDM
Suunnitteluvai-heen laajuus	Osin määritelty, lyhytkestoinen, mutta riittävä	Osin määritelty, mahdollisimman lyhytkestoinen, mutta riittävä	Tarkoin määritelty, kattava ja pitkäkestoinen	Ei määritelty	Tarkoin määritelty, melko kattava, melko pitkäkestoinen	Tarkoin määritelty, kattava ja melko pitkäkestoinen
Iteratiivisuus	Kevyt suunnittelu, toteutusvaihe iteratiivinen	Kevyt suunnittelu, toteutusvaihe iteratiivinen	Jaksottainen suunnittelu-vaihe, toteutusvaihe iteratiivinen	Heikosti määritelty, ainakin toteutus-vaihe iteratiivinen	Jaksottainen suunnittelu-vaihe, toteutusvaihe iteratiivinen	Jaksottainen suunnittelu-vaihe, toteutusvaihe iteratiivinen
Iteraation pituus	30 päivää	2-4 viikkoa	4-10 viikkoa	2-4 kuukautta	2 viikkoa	ei määritelty
Asiakkaan osallistuminen	Suunnittelu, Iteraatioiden suunnittelu, tulokset	Päivittäinen, tiimin aktiivinen jäsen	Suunnittelu, iteraatioiden tulosten katselmointi	Ei tarkoin määritelty, mutta aktiivinen	Suunnittelu, iteraatioiden osalta ei määritelty	Suunnittelu, iteraatioiden tulosten katselmointi
Asiakkaalta vaadittu tekninen osaaminen	Ei erityistä vaatimusta	Korkea	Ei erityistä vaatimusta	Ei määritelty	Ei määritelty	Ei erityistä vaatimusta
Prosessin määrittely	Vaiheet tarkasti määritelty, työskentelytavat vapaat	Vaiheet ylimalkaisesti määritelty, vaihetuotteet ja työskentelytavat määritelty yksityiskohtaisesti	Vaiheet määritelty, vapaat työskentely-tavat	Ei määritelty	Vaiheet ja vaihetuotteet yksityiskohtai-sesti määritelty, työskentelytavat vapaat	Vaiheet ja vaihetuotteet yksityiskohta-isesti määritelty, työskentelytav at vapaat
Vaihetuotteiden määrä	4	6	5	Ei määritelty	5	7
Soveltuva henkilömäärä	Ei rajoituksia, skaalautuvuus hoidettu	Pienet, alle 10 hengen projektit	Ei rajoituksia, skaalautuu hyvin	Eri prosessit erikokoisille projekteille	Ei rajoituksia, suunniteltu suuremmille projekteille	Ei rajoituksia, käytetään yleensä suuremmissa projekteissa

KUVIO 14 *Agile-menetelmien ominaisuuksia*

4.3 Agile-menetelmien hyödyt

Merkittävimmät hyödyt onnistuneesti toteutetusta agile-projektista verrattuna perinteisellä menetelmällä toteutettuun projektiin ovat oikeampi tuote ja toimitus ajallaan. Molemmat näistä ovat asiakkaan ohjaaman iteratiivisen ja

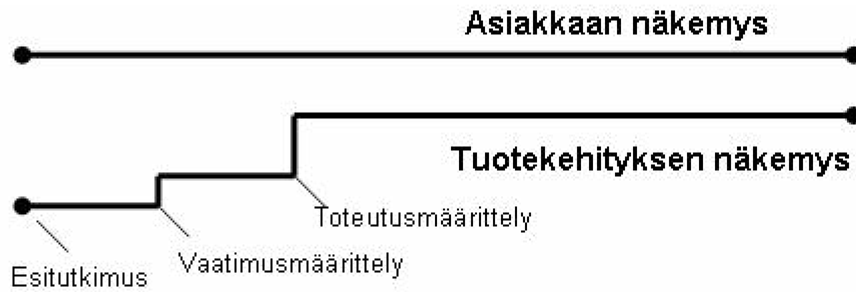
inkrementaalisen työskentelytavan tuomia etuja. Kaikki agile-menetelmät perustuvatkin näille tekijöille. Niiden lisäksi menetelmissä määritellään joukko tukitoimia, joilla luodaan edellytykset näille seikoille tai tehostetaan niiden vaikutusta. Koska Agile-menetelmät perustuvat Lean-ajattelulle, saa menetelmien perusasioista selkeimmän käsityksen tutkimalla Lean-prosessia (luku 2.3.2).

4.3.1 Oikea tuote

Oikeaan tuotteeseen pääseminen voidaan varmistaa vain siten, että asiakas osallistuu projektiin koko sen keston ajan. Projektin alussa tehdyt suunnitelmat ovat erittäin harvoin täydellisiä. Tämä ei pidä paikkaansa ainoastaan muutosherkillä tai tuntemattomilla alueilla työskenneltäessä, vaan myös niin kutsutuissa vakaissa ympäristöissä. Vakaissa ympäristöissä projektisuunnitelmien puutteet voivat johtua esimerkiksi heikosta projektitiimin ja asiakkaan yhteistyöstä, jolloin projektitiimi ei toisaalta ymmärrä kaikkia vaatimuksia oikein eikä asiakas toisaalta hyödynnä projektitiimin teknistä osaamista vaatimusten määrittelemisessä ja tulee näin ollen kirjanneeksi epäolennaisia, puutteellisia tai jopa teknisesti mahdottomia vaatimuksia.

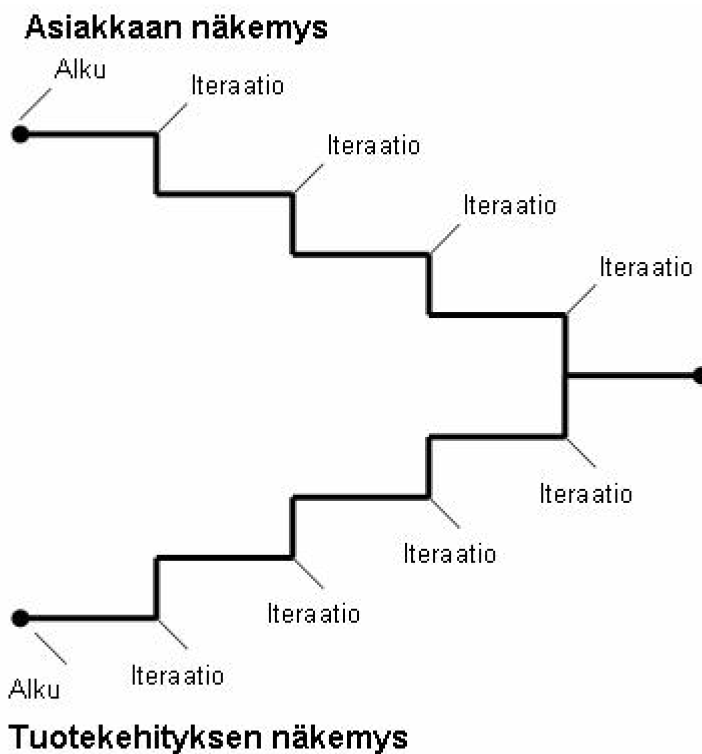
Asiakkaan tulee osallistua projektiin myös sen kestäessä. Tämä on mahdollista vain iteratiivisella ja inkrementaalisella tavalla työskenneltäessä, sillä siten asiakas näkee jo aikaisessa vaiheessa millainen lopullisesta tuotteesta on tulossa, ja voi vaikuttaa projektin lopputulokseen.

Mitä enemmän asiakkaan on mahdollista osallistua projektin ohjaamiseen, sen lähemmäksi asiakkaan todella tarvitsemaa tuotetta päästään. Perinteisellä tavalla työskenneltäessä asiakas osallistuu projektiin vain alussa sekä vaihetuotteiden katselmoinneissa. Tällöin asiakkaan sekä projektitiimin näkemykset lopullisesta tuotteesta etenisivät kuvion 15 mukaisesti.



KUVIO 15 (Highsmith 2000, 186)

Nyt on kuitenkin huomattava, että kuten luvussa 3.3 esitetyssä kaaviossa (kuvio 9) osoitettiin, projektin lopullisen määränpään ei useinkaan kuuluisi olla alkuperäisen suunnitelman mukaisessa määränpäässä (Highsmith 2000, 42-43). Asiakkaan osallistuessa iteraatioiden julkistustilaisuuksiin ohjaamaan projektia oikeaan suuntaan, päästään kuvion 16 osoittamaan tilanteeseen, jossa asiakkaan ja projektitiimin näkemykset lähenevät toisiaan projektin edetessä.



KUVIO 16 *Asiakkaan ja tuotekehityksen näkemys agile-menetelmissä* (Highsmith 2000, 186)

4.3.2 Riittävä tuote aikataulussa

Työskenteleminen aikarajoitetuissa iteraatioissa priorisoidun ominaisuusluettelon perusteella varmistaa sen, että tärkein osa ohjelmiston toiminnallisuudesta on toteutettu aikataulussa.

Iteraatioiden on oltava aikarajoitteisia, jotta jokaisen julkaisun ajankohta olisi ennakoitavissa. Jokaisen on myös tuotettava toimiva ja ajettava lopullisen ohjelmiston osa.

Iteratiivinen työskentelytapa edellyttää selkeää, jokaisen iteraation alussa päivitettävää tehtävälistaa. Tehtävälista taas ei palvele tarkoitustaan ilman selkeää ominaisuusluetteloa. Mikäli ominaisuusluettelo on asiakkaan kanssa yhteistyössä priorisoitu, varmistetaan se, että jos projektin määräaikaan mennessä ei kaikkia ominaisuuksia saada toteutettua, puuttuu lopullisesta tuotteesta vain vähiten tärkeitä ominaisuuksia. Tuote tällöin on kuitenkin vähintäänkin käyttökelpoinen, yleensä jopa aivan riittävän valmis. Jäljelle jääneet ominaisuudet ovat usein tulevaisuuden varalle suunniteltuja lisäominaisuuksia, jotka ovat Lean-ajattelun mukaisesti jätettä, ja ne tulisi näin ollen karsia pois toteutuksesta (Poppendieck & Poppendieck 2003, 6).

4.3.3 Tukitoimia

Kaikki agile-menetelmät perustuvat asiakkaan ohjaamalle iteratiiviselle ja inkrementaaliselle työskentelytavalle. Muut menetelmissä kuvaillut toiminteet tai vaiheet ovat yleensä näitä kahta peruselementtiä tukevia seikkoja.

Agile-menetelmissä määriteltyjä tukitoimia ovat muun muassa tiimien kokojen pitäminen pieninä, tiimien itseohjautuvuus sekä tiimin jäsenten työpisteiden sijaitseminen samoissa tiloissa.

2-4 viikon iteraatiojaksoissa ei yksinkertaisesti ole aikaa toimille, jotka eivät suoraan edistä projektia. Tiimien on oltava pieniä ja tiimiläisten työpisteiden on sijaittava lähellä toisiaan, jotta tiedon välittäminen tiimin kesken olisi mahdollisimman nopeaa ja yksinkertaista. Itseohjautuvuus sekä se, että tiimeillä on riittävästi valtaa ja oikeuksia tehdä jokapäiväiseen työhönsä liittyviä päätöksiä estää projektin etenemisen kannalta tärkeiden päätösten juuttumisen monitasoisiin byrokratiakoukeroihin.

Agile-menetelmien periaatteissa mainitaan myös tasainen työskentelytahti (Cockburn 2007, 376-377), minkä tarkoituksena on estää työntekijöiden loppuun palamisia. Iteratiivisen työskentelytavan merkittävin etu perinteiseen vesiputousmalliseen tapaan verrattuna on projektitiimin kannalta se, että tiimiläisillä säilyy selkeä käsitys siitä, mitä minäkin päivänä on tehtävä, jotta aikataulussa pysytään. Jos ajatellaan esimerkiksi vuoden kestävää projektia, tietäisi perinteisellä prosessimallilla työskentelevä projektitiimi vain sen, että nuo kymmenet tai sadat ominaisuudet on oltava toteutettuna vuoden kuluttua. Kuinka tiimiläiset nyt voivat arvioida, mitä on tehtävä juuri tänään, jotta aikataulussa pysytään? Käytäntö on osoittanut, että eivät mitenkään, ja lähes poikkeuksetta käykin niin, että projektin alussa työtahti on liian hidas, ja se kiihtyy lähes sietämättömäksi projektin määräajan lähestyessä. Iteratiivisessa mallissa projektitiimi tietää, mitkä ominaisuudet on toteuttava seuraavan määritellyn ajanjakson, esimerkiksi yhden kuukauden, aikana. Tällöin tavoitteiden asettaminen viikko- tai jopa päivätasolle on mahdollista, mikä vastaavasti mahdollistaa tasaisen työskentelytahdin saavuttamisen. Tasaisella, ennakoitavissa olevalla työtahdilla, sekä selkeillä tavoitteilla on myös työntekijöiden motivaatiota kasvattavia vaikutuksia (Malik 2002, 156,158; Kettunen 2003, 37-38; Ruohotie 1998, 57).

Muita tukitoimia ovat työskentelyn tehostamiseen kussakin iteraatiossa tarkoitetut työskentelytekniikoihin liittyvät menetelmät, esimerkiksi XP:n pariohjelmointi.

4.3.4 Suunnittelu ja dokumentointi

Jokainen ohjelmistoprojekti vaatii suunnittelemista. Agile-menetelmien kannattajat kuitenkin kritisoivat perinteisten ohjelmistokehitysprosessien tapaa pyrkiä täydellisiin projektisuunnitelmiin projektin alussa. Tämä tapa onkin kyseenalainen, sillä suunnitteludokumentit muuttuvat takuulla projektin kestäessä, toisissa projekteissa merkittävästi ja toisissa selvitään vain pienillä päivityksillä. Usein ei myöskään ole edes mahdollista määritellä projektin toimintaa puutteellisten tietojen vuoksi vielä projektin alussa sillä tarkkuudella, mitä perinteiset prosessimallit vaativat. (Kroll&Kruchten 2004, 11, 91; Pressman 2005, 80; Cockburn 2007, 219)

Agile-menetelmissä suunnitelmat tehdään *riittävälle* tasolle, eli vain niin valmiiksi, että siirtyminen prosessin seuraavaan vaiheeseen on mahdollista. ”Riittävä” taso on tietenkin projektikohtaisesti muuttuva käsite. Joissakin projekteissa täytyy tiettyjä lopullisen toteutuksen osia, kuten vaikkapa ulkoisia rajapintoja tai tietoturvaan liittyviä seikkoja, määritellä tarkalla tasolla jo projektin alussa. Toisissa projekteissa taas ei tarvita vielä alkuvaiheessa kuin yleisen tason suunnitelma toteutettavasta järjestelmästä. Jälkimmäisessä tapauksessa projekti käynnistettäisiin agile-menetelmiä käytettäessä yleisen suunnitelman perusteella ja tarkemmat toteutussuunnitelmat tehtäisiin iteraatiokohtaisesti. (Cockburn 2007, 247-249)

Dokumenttien määrän sekä sisällön suhteen noudetaan pääosin seuraavaa ohjetta: dokumentteja tuotetaan *riittävä* määrä, ja ne ovat sisällöltään *riittävän* kattavia. Kaikki ylimääräinen on karsittu Lean-ajattelun mukaisesti pois. (Cockburn 2007, 219-220, 42)

5. TULOSTEN TARKASTELU

5.1 Kokeiluprojektit

Liitteissä 2 ja 3 kuvaillaan kahta agile-projektia, joissa on käytetty agile-menetelmiä. Toisessa prosessimallina oli Scrum-sovellus ja toisessa käytettiin itse määriteltyä agile-prosessia.

Scrum-sovellus

Scrum-sovelluksessa (Liite 2) projektin tulokset eivät olleet millään tavalla merkittäviä: tärkeimmät ominaisuudet saatiin toteutettua, kokonaisuuden testaaminen oli osin myöhässä ja toissijaisia tavoitteita jäi toteuttamatta tai valmistumatta. Samaan lopputulokseen olisi varmasti päästy myös perinteistä projektimallia noudattamalla. Syynä heikkoon lopputulokseen ei kuitenkaan ollut Scrum, vaan perinteisestä projektimallista poikkeavien – mutta menetelmän menestymisen kannalta tärkeiden – muutosten jättäminen pois prosessista. Käytetystä prosessista puuttuivat pakollisista Scrum-elementeistä Product Owner, Product Backlog, Daily Scrum sekä Burndown Chart. Lisäksi asiakkaan läsnäolo oli puutteellista, projektitiimi ei työskennellyt yhdellä paikkakunnalla, Sprint Backlog oli virheellinen, Scrum Masterin nimikkeellä toiminut henkilö toimi vain tavallisena projektipäällikönä ja koko projektia oli käskyttämässä Scrum Masterin esimies. Päätavoitteiden toteutuminen projektissa oli paljolti projektitiimin päätavoitteista vastaavien henkilöiden oma-aloitteisen itseorganisoitumisen ansiota. Koska prosessissa ei siis käytännössä ollut Scrumin osista mukana kuin iteratiivinen ohjelmistokehitys, eikä siitäkään saatu Product Backlogin puuttumisen sekä Sprint Backlogien virheellisyyden vuoksi kaikkea mahdollista hyötyä irti, ei prosessia ole aiheellista edes kutsua Scrumiksi. Tiimien ei myöskään sallittu olevan itseohjautuvia, vaan projektia päätettiin johtaa perinteisesti. Tällä menettelyllä oli odotetut seuraukset, eli ylimääräinen byrokratiaporras hidasti lyhyissä sykleissä toimivan projektiryhmän työskentelyä merkittävästi. (Liite 2)

Vaikkei kyseistä projektia voidakaan hyödyntää Scrumin tehokkuuden arvioimisessa, toi se hyvin esiin Scrumin käyttöönottamista edellyttävän ajattelutavan muutoksen sekä vanhoista tavoista irrottautumisen vaikeuden. Vanhoista tavoista irrottautuminen on sitä vaikeampaa, mitä paremmin vanha rutiini on joskus toiminut. Tutkimuksen (Ruohotie 1998, 32-33 muk. Kevätsalo 1992) mukaan työnantajat pitäytyvätkin usein rutiineissa jopa taloudellisen edun kustannuksella. Kuvailtu projekti myös osoittaa, ettei Scrumia voida ottaa käyttöön vanhojen toimintatapojen ehdoilla, vaan toimintatapoja on uskallettava muokata Scrumin mukaisiksi.

Oma agile-prosessi

Toisessa kuvaillussa projektissa lähestymistapa oli erilainen. Valmiin prosessimallin soveltamisen sijaan agile-menetelmistä kokemusta omaava ohjelmistoarkkitehti kehitti kyseisen projektin tarpeisiin soveltuvan agile-ajattelun mukaisen prosessin. Kuten Liitteessä 3 kuvaillaan, prosessi poikkesi agile-periaatteista vain asiakkaan aktiiviseen osallistumiseen liittyviltä osin. Asiakkaan aktiivinen mukana oleminen olisi varmasti ollut etu, mutta nykyiselläänkin kyseisessä projektissa toimitettiin myyntiin kelpaava ensimmäinen versio tuotteesta yhdessä miestyökuukaudessa. Perinteisellä projektimallilla yhdessä kuukaudessa ei olisi vielä päästy edes määrittelyvaiheen loppuun, puhumattakaan toimivan sovelluksen toimittamisesta.

Kyseisessä projektissa käytettyä prosessia arvioidessa on kuitenkin huomattava, että kyseessä oli agile-menetelmien hyödyntämiseen erityisen sovelias projekti: pienen tiimin alusta loppuun toteuttama tuote, jonka tuotekehitysprosessi oli mahdollista määritellä kokonaan itse nimenomaan kyseisen projektin tarpeisiin. Projektin erittäin nopean aikataulun vuoksi useimmat määritellyistä agile-menetelmistä eivät olisi sellaisenaan soveltuneet tähän projektiin, vaan mitä tahansa valittua menetelmää olisi joka tapauksessa jouduttu joiltain osin soveltamaan. Kehitetty prosessi ei varmastikaan sellaisenaan toimisi missä tahansa projekteissa, mutta osoittaa silti sen, että parhaaseen tulokseen päästään joissain tilanteissa vain suunnittelemalla juuri omalle projektille soveltuva prosessi olemassa olevien mallien hyödyntämisen sijaan.

5.2 Sovellettavuus

Jokainen agile-menetelmistä vaikuttaa suunnitellun tietyn tyyppisille projekteille. Scrum soveltuu parhaiten omaa tuotetta tiedossa olevalle maksavalle asiakkaalle tekeville, XP erityisen muutosherkkään ympäristöön, ASD suuremmille projekteille, FDD suuremmille olio-ohjelmointiprojekteille ja DSDM projekteille, joissa on tarkkoja ulkoisia vaatimuksia, esimerkiksi turvallisuusvaatimuksia. Crystal pyrkii huomioimaan erityyppiset projektit, ja määrittelee siksi vain muutamia yleistasoisia periaatteita jättäen prosessin suunnittelemisen sekä menetelmien valitsemisen projektiryhmän huoleksi.

Vaikka menetelmät vaikuttavatkin soveltuvan parhaiten kuvatunlaisiin projekteihin, on niitä toki mahdollista soveltaa myös muunlaisissa projekteissa. Tällöin on kuitenkin huomattava, että jokainen prosessin osa kussakin prosessimallissa on suunniteltu tukemaan toisiaan, joten menetelmiä tulisi soveltaa vain alkuperäisessä muodossaan. Vaiheiden jättäminen pois aiheuttaa varmasti ongelmia, ja vaiheiden korvaaminen toisilla edellyttää hyvin kattavaa ja huolellista muutoksen aiheuttamien seuraamusten analysointia. Esimerkiksi XP:ssä katselmoinnit ja täsmälliset suunnitteludokumentit on korvattu asiakkaan aktiivisella osallistumisella, joten XP:n soveltaminen projektissa, johon asiakkaan edustajan ei ole mahdollista osallistua täyspainoisesti ei ole järkevää. (Cockburn 2007, 344).

Tietotekniikan palvelualalla voidaan käyttää monenlaisia sopimusmalleja. Jotain määriteltyä agile-menetelmää voidaan kokeilla, kunhan sopimus ei vaadi minkään tietyn prosessin noudattamista tai tiettyjen vaihetuotteiden tuottamista, ja lisäksi asiakas sitoutuu osallistumaan projektin toteutukseen läpi koko projektin keston. Mikäli jokin näistä ehdoista ei toteudu, ei mitään agile-menetelmää voida soveltaa sellaisenaan.

Jos olosuhteet ovat jonkin määritellyn agile-menetelmän kokeilemisen kannalta suotuisat, tulee menetelmän valitsemiseen vielä kiinnittää erityistä huomiota. Esimerkiksi Scrumia ei tule soveltaa suuren palaverimäärän vuoksi tiimeissä, joissa on henkilöitä, jotka eivät voi antaa kyseiselle projektille täyttä työpanosta ja tarkempaa suunnitteluvaihetta vaativiin projekteihin tulee soveltaa ASD:tä, FDD:tä tai DSDM:ää, ei missään tapauksessa XP:tä eikä mielellään Scrumia.

5.3 Suositeltava ratkaisu

Kaikilla ohjelmistoprosesseilla on vahvuuksia ja heikkouksia. Koska jokainen projekti on erilainen, ei mikään prosessi sovellu kaikille. Näin ollen projektitiimien on syytä käyttää aikaa oikean prosessin valitsemiseen. Mikäli sopivaa prosessia ei löydy, on se suunniteltava itse agile- perusperiaatteita pohjana käyttäen. (Astels et al. 2002, xxix; Cockburn 2007, 228).

Siirtyminen perinteisestä prosessimallista suoraan johonkin tarkasti määriteltyyn agile-menetelmään on erittäin vaikeaa. Muutoksen mittavuuden vuoksi epäonnistuminen on pelottavan todennäköistä, ja epäonnistuneet yritykset luovat negatiivisen käsityksen agile-menetelmistä kaikille osallisille. Tästä syystä agile-menetelmistä kiinnostuneiden olisi turvallisempaa ainakin aluksi kokeilla vain tärkeimpien perusajatusten upottamista käytössä olevaan prosessiin projektiympäristön sallimassa määrin. Tämä tapa ei tuo yhtä hyviä tuloksia kuin käytetyn prosessimallin korvaaminen jollain määritellyllä agile-menetelmällä, mutta se tuottaa silti parempia tuloksia kuin perinteinen työskentelytapa. Näin saadut positiivisten kokemukset helpottavat mahdollista siirtymistä jonkin määritellyn agile-menetelmän käyttämiseen jatkossa.

Kaikkien agile-menetelmien perustana on asiakkaan ohjaama iteratiivinen ja inkrementaalinen ohjelmistokehitys. Luvuissa 4.3.1-4.3.3 on kuvailtu miten tästä menetelmästä saadaan paras hyöty irti, mutta iteratiivisen ohjelmistokehityksen soveltaminen on mahdollista myös tarkemmin rajatuissa ympäristöissä (Cockburn 2007, 222, 292).

Seuraavissa luvuissa kuvaillaan iteratiivisen ohjelmistokehityksen soveltamista erilaisissa ympäristöissä. Esitellyt menetelmät eivät siis ole esimerkkejä itsekehitetyistä agile-prosesseista, vaan ainoastaan esimerkkejä agile-menetelmien tärkeimpien elementtien upottamisesta käytettyyn prosessiin.

5.3.1 Vesiputousmalli, vaatimusmäärittely asiakkaalta, asiakas passiivinen

Tässä esimerkkitapauksessa asiakas kirjoittaa itsenäisesti vaatimusmäärittelyn, jonka perusteella projekti on tarkoitus toteuttaa. Jatkossa asiakas osallistuu ainoastaan vaihetuotteiden katselmointeihin, eikä näin ollen voinut todella ohjata projektia. Projektitiimi on myös määrätty käyttämään vesiputousmallista prosessia. Tällainen järjestely on agile-menetelmien soveltamisen kannalta erittäin epäedullinen. Implementointivaiheen suorittaminen iteratiivisena ja inkrementaalisena on kuitenkin mahdollista, ja pelkästään sillä on mahdollista saavuttaa etuja perinteiseen työskentelytapaan verrattuna.

Iteratiivinen työskentely alkaa projektin ominaisuusluettelon koostamisella. Tässä esimerkkitapauksessa luettelo perustuisi asiakkaan tekemälle vaatimusmäärittelylle. Koska asiakkaalla ei yleensä ole samantasoista osaamista projektin tekniseltä alueelta, ovat asiakkaan yksin tekemien vaatimusmäärittelyjen vaatimukset joskus liian yleistasoisia testattavuuden kannalta, usein osittain päällekkäisiä, ja joskus jopa teknisesti mahdottomia. Tästä syystä ominaisuusluettelon koostaminen on tässä esimerkkitapauksessa haasteellista ja aikaa vievää, mutta kuitenkin ehdottoman tarpeellista.

Kun ominaisuusluettelo on saatu valmiiksi, jaetaan prosessin implementointivaiheeseen sekä moduulitestaukseen varattu aika iteraatiojaksoihin ja jaetaan ominaisuudet alustavasti iteraatioihin.

Tämän jälkeen toiminnallisuus toteutetaan aikarajoitetuissa iteraatioissa, joista jokaisen tuloksena on synnyttävä testattu, toimiva ja ajettava lopullisen ohjelmiston osa. Testaus on tässä vaiheessa yksikkötestausta, ja esimerkiksi

”testit ennen koodia” (test-first design)-menetelmää voi kokeilla yksikkötestien kirjoittamisen varmistamiseksi. Yksikkötestejä tulisi pyrkiä automatisoimaan mahdollisuuksien puitteissa. Iteratiivisella tavalla työskenneltäessä on projektitiimin talletettava toteuttamaansa ohjelmakoodia mahdollisimman usein. Näin ollen helppokäyttöinen versionhallintatyökalu on ehdoton edellytys iteratiivisen työskentelyn onnistumiselle.

Inkrementaalinen tuotekehitys edellyttää jatkuvaa ohjelmiston rakenteen huomioimista. Ohjelmistoa tulee refaktoroida aina tarpeen vaatiessa, jotta rakenne pysyy selkeänä ja mahdollisimman yksinkertaisena.

Tiimin olisi hyvä myös arvioida omaa toimintaansa esimerkiksi erillisillä toiminnanarviointipalavereilla, joissa käydään läpi kunkin tiimin jäsenen hyväksi tai huonoksi kokemia käytetyn työskentelytavan piirteitä. Näitä palavereja tulee pitää säännöllisin väliajoin, esimerkiksi muutaman iteraation välein.

Iteratiivisella ja inkrementaalisella tavalla työskenneltäessä projektitiimillä säilyy koko ajan riittävä käsitys siitä, mitä minäkin päivänä on saatava valmiiksi, jotta aikataulussa pysytään. Tällöin riski sille, että implementointivaiheen alussa työskenneltäisiin liian hitaasti ja lopussa tulisi lähes hallitsematon kiire, pienenee huomattavasti. Selkeät, lyhyen tähtäimen tavoitteet myös motivoivat ja auttavat suoriutumaan tehtävistä aikataulussa.

Kiusallisen usein projektien implementointivaiheen kesto on arvioitu virheellisesti, ja tällöin perinteisellä tavalla työskenneltäessä projektin valmistumispäivämäärä siirtyy. Pahimmassa tapauksessa alkuperäisen suunnitelman mukaisena valmistumispäivänä ei välttämättä ole valmiina mitään sellaista, mitä toiminto- tai järjestelmätestaajien olisi järkevää testata. Tällöin testaajat joutuvat vain odottelemaan projektin valmistumista, mikä on tietenkin tarpeetonta tyhjäkäyntiä. Iteratiivisesti ja inkrementaalisesti työskenneltäessä projektin alkuperäisenä määräpäivänä projekti on todennäköisesti valmiimpi kuin perinteisellä tavalla työskenneltäessä ja lisäksi tuotos on varmasti yksikkötestattu ja ajettava. Koska ominaisuusluetteloa ei kuitenkaan asiakkaan passiivisen roolin vuoksi ollut

mahdollista priorisoida, joudutaan jäljelle jääneetkin ominaisuudet varmasti toteuttamaan. Testaajat voivat kuitenkin aloittaa toimintotestausta toteutettujen ominaisuuksien osalta samalla kun projektitiimi toteuttaa puuttuvia ominaisuuksia. Kun testaajat ovat saaneet alkuperäisessä aikataulussa valmistuneet ominaisuudet testattua, on puuttuvatkin ominaisuudet jo todennäköisesti toteutettu.

Perinteisellä menetelmällä implementointivaiheen keston aliarvioiminen johtaisi koko projektin myöhästymiseen, mutta mikäli implementointivaihe toteutettaisiin aikarajoitetuissa iteraatioissa, on projektin mahdollista valmistua alkuperäisen aikataulun puitteissa.

5.3.2 Vesiputousmalli, vaatimusmäärittely yhteistyössä, asiakas passiivinen

Mikäli vaatimusmäärittely työstetään yhdessä asiakkaan kanssa, vältytään edellä mainituilta vaatimusten laatuongelmilta. Paremman vaatimusmäärittelyn perusteella ominaisuusluettelo on helpompi ja nopeampi toteuttaa. Tällöin myös iteraatiojaksojen tavoitteet ovat selkeämmät ja projekti tulee tarpeettomien vaatimusten puuttumisen vuoksi nopeammin valmiiksi. Vaatimusten pudottaminen jo määrittelyvaiheessa lyhentäisi kuitenkin todennäköisesti myös projektin arvioitua kestoa, joten muutoksesta ei olisi vaikutusta projektin kokonaisaikataulussa pysymiseen.

Projekti tulee toteuttaa samoin kuin edellisessä esimerkissä, ja tässäkin tapauksessa implementoitivaiheen keston aliarvioiminen saataisiin todennäköisesti kirittyä kiinni testausvaiheessa, joten projekti valmistuisi jälleen aikataulussa. Koska vaatimusmäärittely on kuitenkin tällä kertaa tehty yhteistyössä ja vaatimukset ovat siten selkeämpiä eikä päällekkäisyyksiä ole, olisi projekti perinteisiä menetelmiä sekä edellistä prosessiesimerkkiä noudattaviin projekteihin verrattuna aikaisemmin valmis. Vaatimusmäärittelyvaiheen yhteistyön ansiosta vältytään myös vaatimusten väärinymmärryksiltä, joten

projektin lopputulos on varmasti hyvin lähellä asiakkaan alkuperäistä käsitystä tarvittavasta tuotteesta.

5.3.3 Vesiputousmalli, priorisoitu vaatimusmäärittely yhteistyössä, asiakas passiivinen

Yhteistyössä toteutettu vaatimusmäärittely, jossa vaatimukset on priorisoitu, ohjaa iteratiivisesti työskentelevää projektitiimiä oikeaan suuntaan. Priorisointi voi olla hyvinkin yksinkertainen, esimerkiksi Astelsin, Granvillen ja Novakin (2002) esittelemä kolmivaiheinen priorisointimenetelmä riittää mainiosti:

A: ”Koko järjestelmä on turha ilman tätä”

B: ”Todella tarpeellinen, mutta ominaisuutta ilman pärjätään tilapäisesti”

C: ”Joskus ehkä tarpeellinen ominaisuus”

(Astels et al. 2002, 213)

Itse toteutusvaihe on sama kuin edellisissä esimerkeissä, mutta tässä tapauksessa ominaisuusluettelo on priorisoitu vaatimusluettelon perusteella, ja siksi projektitiimin on mahdollista toteuttaa tärkeimmät ominaisuudet ensin. Mikäli implementointivaiheen kesto olisi tässäkin projektissa aliarvioitu, ja ominaisuuksia jäisi alkuperäisessä määräajan koittaessa toteuttamatta, olisivat toteuttamatta jääneet ominaisuudet *vähiten tärkeitä*. Ensiarvoisen tärkeistä ominaisuuksista olisi todennäköisesti kaikki toteutettu alkuperäisessä aikataulussa. Tällöin olisi mahdollista neuvotella asiakkaan kanssa jäljellejääneiden toissijaisten ominaisuuksien kohtalosta: onko ne todella tarpeen toteuttaa tässä yhteydessä, toteutetaanko ne vasta tarpeen vaatiessa vai toteutetaanko lainkaan. Pareton periaatteen mukaan 80 % minkä tahansa seurauksista aiheutuu 20 % seurausten syitä (BetterExplained 2008). Periaatetta on sovellettu ohjelmistoalalla muodossa ”80 % minkä tahansa ohjelmiston käyttäjistä käyttää 20 % kyseisen ohjelmiston ominaisuuksista” (Schwaber 2004, 149), joten on täysin mahdollista, että tuote on riittävän valmis, vaikka muutamia vähiten tärkeitä ominaisuuksia puuttuisikin.

Projekti valmistuisi siis implementointivaiheen aikatauluarvion virheestä huolimatta hyvin suurella todennäköisyydellä alkuperäisessä aikataulussaan ja lisäksi jäljelle jäävien ominaisuuksien kohtalosta riippuen myös implementointivaiheen olisi mahdollista valmistua ajallaan. Edellisen esimerkin tavoin vaatimusten väärinymmärryksiltä vältytään yhteistyössä tehdyn vaatimusmäärittelyn ansiosta, ja projektin lopputulos tulee jälleen olemaan erittäin lähellä alkuperäistä suunnitelmaa.

5.3.4 Vesiputousmallin vaihetuotteet, priorisoitu vaatimusmäärittely yhteistyössä, asiakas passiivinen

Tässä esimerkkitapauksessa asiakas vaatii edellisten esimerkin tavoin tiettyjen vaihetuotteiden toteuttamista, mutta ei tällä kertaa määrittele missä vaiheessa projektia ne on toteutettava. Muilta osin toimintaympäristö on samanlainen kuin edellisessä esimerkissä.

Kun täsmällisiä suunnitteludokumentteja (vaatimusmäärittelyä lukuun ottamatta) ei tarvitse toteuttaa ja katselmoida ennen implementointivaiheen aloittamista, voidaan iteraatiosykleihin liittää dokumenttien päivittäminen soveltuvilta osin. Tämä on huomattavasti nopeampi menettelytapa kuin suunnitteludokumenttien kirjoittaminen etukäteen ja niiden päivittäminen todellista toteutusta vastaaviksi projektin jälkeen. Iteraation lopputuloksena on tässä tapauksessa toimiva ja testattu lopullisen ohjelmiston osa, jonka tekniset dokumentit vastaavat todellista toteutusta.

Tämä toimintatapa tuottaisi yhteistyössä toteutetun vaatimusmäärittelyn ansiosta asiakkaan alkuperäistä näkemystä vastaavan tuotteen, joka olisi dokumentoinnissa säästetyn ajan vuoksi valmis nopeammin kuin perinteisillä menetelmillä tai aiempien esimerkkien mukaisilla menetelmillä työskenneltäessä.

5.3.5 Vesiputousmallin vaihetuotteet, priorisoitu vaatimusmäärittely yhteistyössä, asiakas aktiivinen

Tässä esimerkkitapauksessa lähtökohdat ovat muutoin samanlaiset kuin edellisessä esimerkissä, mutta tällä kertaa asiakas sitoutuu ottamaan aktiivisemman roolin projektin ohjaamisessa.

Asiakkaan näkemys halutusta tuotteesta ei ennen projektin aloittamista useinkaan vastaa asiakkaan todellisia tarpeita. Tästä syystä projektin alussa tehdyn suunnitelman seuraaminen ei välttämättä tuota sellaista tuotetta, minkä asiakas todella tarvitsee. (Highsmith 2000, 42-43)

Iteratiivisesti ja inkrementaalisesti työskenneltäessä voidaan asiakkaalle esitellä tasaisin aikavälein tuotteen toiminnallisuutta ja käytettävyyttä. Tässä esimerkkiprojektissa asiakas on sitoutunut osallistumaan projektin läpivientiin aktiivisesti, joten asiakkaan edustaja on aina läsnä iteraatioiden tulosten julkistamistilaisuudessa. Kun asiakas näkee tuotteen kehittymisen, muodostuu myös asiakkaalle oikeampi käsitys tavoitellusta tuotteesta. Asiakas voi ohjata projektia antamalla palautetta jokaisen iteraation jälkeen. Alkuperäistä vaatimusluetteloa voidaan myös muuttaa viimeisimmän käsityksen mukaiseksi.

Tällä menetelmällä työskenneltäessä on mahdollista toteuttaa perinteisiä menetelmiä nopeammin tuote, joka ei välttämättä ole alkuperäisen suunnitelman mukainen, mutta vastaa asiakkaan todellisia tarpeita.

5.3.6 Ei ennalta määriteltyä prosessia, aktiivinen asiakas

Kaikkein paras ympäristö agile-menetelmien soveltamisen suhteen on sellainen, jossa ei ole ennalta määrättyjä pakollisia vaiheita tai vaihetuotteita ja jossa asiakas sitoutuu osallistumaan projektin läpivientiin aktiivisesti. Tällöin tulee toimia pääosin edellisen esimerkin mukaisesti, mutta lisäksi projektitiimin on mahdollista poistaa prosessista kaikki ylimääräiset, asiakkaalle lisäarvoa

tuottamattomat piirteet Lean-periaatteiden mukaisesti (Poppendieck & Poppendieck 2003, XXV, 2-4).

Kun prosessissa keskitytään vain lisäarvoa tuottaviin toimiin, lyhenee projektin kesto merkittävästi. Tällä menetelmällä työskenneltäessä on siksi mahdollista toteuttaa hyvinkin nopeasti tuote, joka vastaa asiakkaan todellisia tarpeita.

Menetelmä hyödyntää agile-menetelmien tehokkaimpia tulosta parantavia piirteitä ja noudattaa agile-menetelmien ydinarvoja. Mikäli tiimien itseohjautuvuus sallitaan, noudattaa menetelmä myös kaikkia agile-periaatteita, jolloin kyseisen menetelmän voidaan todeta olevan täysipainoinen agile-menetelmä.

5.3.7 Tukitoimien lisääminen

Edellä mainittujen toimintamallien tehokkuutta voi parantaa liittämällä prosessiin sen tärkeimpiä elementtejä tukevia toimia mahdollisuuksien puitteissa. Tukitoimia ovat esimerkiksi XP:n tuotekehityksen tehokkuutta parantamaan tarkoitettu pariohjelmointi (Astels et al. 2002, 9-10), Scrumin tiimin jäsenten keskinäistä tiedonjakoa edistävät lyhyet päivittäiset palaverit (Schwaber 2004, 8, 99) ja XP:n palaverien tehokkuuden parantamiseen tarkoitetut seisontapalaverit (Astels et al. 2002, 6). Cockburn (2007, 357-358) määrittelee useita tukitoimia, joihin kannattaa perehtyä siinä vaiheessa, kun prosessin peruselementit ovat kohdallaan.

Tukitoimia on järkevää kokeilla projekteissa. Kokemusten perusteella on jatkossa hylättävä toimet, jotka eivät toimineet toivotulla tavalla ja vastaavasti vakinaistaa hyödylliseksi koetut toimet ohjelmistoprosessin osaksi. (Cockburn 2007, 356).

5.3.8 Yhteenveto

Agile-periaatteita ja ydinarvoja noudattamalla voidaan tuottaa asiakkaalle projektin alussa tehtävän suunnitelman mukaan etenemiseen verrattuna

ajankohtaisempi ja oikeampi tuote, ja se voidaan julkaista perinteisiä menetelmiä merkittävästi nopeammin. Agile-menetelmien tehokkuus perustuu asiakkaan aktiiviselle roolille, iteratiiviselle ja inkrementaaliselle työskentelytavalle sekä näitä elementtejä tehostaville tukitoimille. Vaikka menetelmät ovat tehokkaimmillaan pienissä tiimeissä toteutetuissa projekteissa, voidaan menetelmien käyttöönotolla tehostaa ohjelmistokehitystä muissakin ympäristöissä. Näin ollen agile-menetelmiä on mahdollista ja järkevää soveltaa myös tietotekniikan palvelualalla.

Koska kaikki projektit sekä niissä toimivat henkilöt ovat kuitenkin erilaisia, ei yhden oikean agile-prosessimallin kehittäminen ole mahdollista. Olemassa olevat agile-menetelmät ovat tehokkaita niissä ympäristöissä, joihin ne on kehitetty, mutta menettävät tehoaan merkittävästi silloin, kun projektiympäristö ei ole sellainen, millaiseksi menetelmän kehittäjä on sen määritellyt. Mitä tahansa menetelmistä kannattaa siis käyttää, mikäli menetelmän kaikkia pakollisia osia on mahdollista noudattaa, mutta mitään menetelmistä ei kannata yrittää ottaa käyttöön sovellettuna. Mikäli jotain menetelmää aiotaan käyttää, on siis joko muokattava projektiympäristö kyseiselle menetelmälle suotuisaksi, valittava jokin soveltuvampi menetelmä tai kehitettävä kokonaan uusi menetelmä oman projektin tarpeisiin. Tilanteissa, joissa omaa projektiympäristöä ei ole mahdollista muokata agile-menetelmille täysin soveltuvaksi, on silti järkevää pyrkiä sulauttamaan agilen tärkeimpiä ominaisuuksia mukaan omaan projektiympäristöön soveltuvin osin. (Cockburn 2007, 222, 244, 292; Liite 2; Liite 3; Luvut 5.3.1-5.3.6)

LÄHTEET

Agile Alliance 2002, Agile Manifesto. Luettu 13.11.2006. http://www.agilealliance.com/.

Astels, D., Granville, M., Novak, M. 2002, A Practical Guide to Extreme Programming. Prentice Hall, NJ USA.

BetterExplained 2008, Understanding the Pareto Principle (The 80/20 Rule). Luettu 3.5.2008. http://betterexplained.com/articles/understanding-the-pareto-principle-the-8020-rule/

BodyLanguageExpert 2008, Body Language and Proxemics. Luettu 3.5.2008 http://www.bodylanguageexpert.co.uk/BodyLanguageAndProxemics.html

Burlton, R. 2001, Business Process Management: Profiting from Process. Sams Publishing, Indiana USA.

Coad, P. 1999, Java Modeling in Color with UML. Prentice-Hall, NJ, USA.

Cockburn, A. 2001, Agile Software Development. Addison-Wesley, Boston, USA.

Cockburn, A. 2007, Agile Software Development 2nd edition – The Cooperative Game. Addison-Wesley, Boston, USA.

Collins Cobuild English Dictionary, 1995. Useita kirjoittajia. HarperCollins Publisher, Ltd., Lontoo, Iso-Britannia.

DeMarco, T., Lister, T. 1987, Peopleware: Productive Projects and Teams. Dorset House, New York NY USA.

Haikala I., Märijärvi J. 1998, Ohjelmistotuotanto. Gummerus Kirjapaino, Jyväskylä.

Highsmith, J.A. 2000, Adaptive Software Development – A Collaborative Approach to Managing Complex Systems. Dorset House, NY USA.

Fowler, M. 2004, Is Design Dead? ThoughtWorks, www.martinfowler.com/articles/designDead.html. Luettu 7.11.2006.

Kalermo, J., Rissanen J. 2002, Agile software development in theory and practice – Master's thesis in Information systems science. University of Jyväskylä, Department of Computer Science and Information Systems.

Kettunen, S. 2003, Onnistu Projektissa. WS Bookwell Oy, Juva.

Kielijelppi 2008, Proksemiikka eli tilankäyttö. Kielijelpin www-sivusto. http://www.kielijelppi.fi/puheviestintalisatieto/proksemiikka-eli-tilankaytto . Luettu 23.4.2008

Kroll, P., Kruchten, P. 2004, The Rational Unified Process Made Easy – a Practitioner's Guide to the RUP. Addison-Wesley, Boston USA.

Longstreet, D. 2008, The Agile Method and Other Fairy Tales. http://www.softwaremetrics.com/Agile/ .Luettu 1.11.2007

Malik, F. 2002, Toimiva johtaminen käytännössä. Multiprint Oy, Helsinki.

Martin, J. 1991, Rapid Application Development. Macmillan, NY USA

Nelli 2006, Yliopistojen, ammattikorkeakoulujen sekä yleisten kirjastojen hakuportaali. http://www.nelliportaali.fi/ Hakuja tehty 14.-16.11.2006.

Nurmi, T., Rekiaro, I., Rekiaro, P., Sorjanen, T. 2001, Gummeruksen suuri sivistyssanakirja. Gummerus Kirjapaino Oy, Jyväskylä.

Poppendieck, M., Poppendieck, T. 2003, Lean SW Development, Addison-Wesley, USA.

Pressman, R.S. 2005, Software Engineering – A Practitioner's Approach, 6th edition. McGraw-Hill, New York NY USA.

Ruohotie, P. 1998, Motivaatio, tahto ja oppiminen. Edita, Helsinki.

Schwaber, K. 2004, Agile Project Management with Scrum, Redmond WA USA, Microsoft Press.

Shakespeare, W. 1992, Shakespeare – The Complete Works. The Bath Press, Bath, Iso-Britannia.

Stapleton, J. 1997, Dynamic Systems Development Method – The Method in Practice. Addison-Wesley, Iso-Britannia.

Yegge, S. 2006, Good Agile, Bad Agile. http://steve-yegge.blogspot.com/2006/09/good-agile-bad-agile_27.html. Luettu 15.3.2007.

Liite 1: Agile Manifesti

Agile ydinarvot:

- Yksilöt ja kanssakäyminen on tärkeämpää kuin prosessit ja työkalut
- Toimiva ohjelmakoodi on tärkeämpää kuin kattava dokumentaatio
- Yhteistyö asiakkaan kanssa on tärkeämpää kuin sopimusneuvottelut
- Muutoksiin reagoiminen on tärkeämpää kuin suunnitelman mukaan toimiminen

Agile periaatteet:

1. Meille on tärkeintä asiakkaan tarpeiden tyydyttäminen toimittamalla asiakkaalle lisäarvoa tuovaa ohjelmistoa aikaisin ja usein
2. Toimitamme toimivaa ohjelmistoa usein, mieluiten vähintään muutaman viikon, kuitenkin enintään muutaman kuukauden välein
3. Toimiva ohjelmisto on tärkein projektin etenemisen mittari
4. Muutokset, myös myöhäisessä vaiheessa tulleet, ovat tervetulleita. Agile-prosessit kääntävät muutokset kilpailueduksi asiakkaalle
5. Liiketaloudellisen sekä teknisen puolen henkilöt työskentelevät päivittäin yhdessä koko projektin ajan
6. Rakenna projektisi motivoituneen henkilöstön ympärille. Anna heille tarvittava tuki ja luota heihin
7. Kaikkein tehokkain tiedonvälittämiskeino on keskustelu kasvotusten
8. Itseorganisoituneet tiimit tuottavat parhaat ohjelmistojen rakenteet sekä vaatimukset
9. Jatkuva ohjelmiston rakenteen sekä teknisen laadun huomioiminen parantaa projektin ketteryyttä
10. Agile-prosessit suosivat ohjelmistokehityksen jatkuvuutta. Kaikkien projektiin osallistuvien on kyettävä säilyttämään tasainen työskentelytahti päivästä toiseen.
11. Yksinkertaisuus – eli tarpeettoman työn tekemisen välttäminen – on ensiarvoisen tärkeää.
12. Tiimi arvioi oman työskentelynsä tehokkuutta säännöllisin aikavälein tehostaakseen toimintaansa

(Cockburn 2007, 373-377)