Extreme Programming

Harri Lindberg

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Joulukuu 2003.

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Harri Lindberg: Extreme Programming
Pro gradu -tutkielma, 67 sivua
Joulukuu 2003

Tutkimusten mukaan ohjelmistoprojektien onnistumistodennäköisyys ei ole hyvä. Projektit myöhästelevät ja ylittävät aikataulunsa. Ohjelmistoprojektien hallintaan on käytetty perinteisiä menetelmiä kuten vesiputousmallia, mutta niiden noudattaminen hyvinkään dokumentoidun laatujärjestelmän puitteissa ei useinkaan auta projektipäälliköitä.

Muutaman viime vuoden aikana on keskusteltu uusista projektinhallinta-metodologioista, joilla näitä ongelmia voitaisiin välttää tai vähentää. Näistä eniten esillä on ollut Extreme Programming, josta käytetään myös yleisesti lyhennystä XP. Tämän tutkimuksen tarkoituksena on tutustua kyseiseen metodologiaan sekä vertailtu sitä perinteisiin metodologioihin. Tarkastelu tapahtuu etupäässä projektipäällikön näkökulmasta, mutta esiin on tuotu myös ohjelmoijien, asiakkaan ja johdon näkökulmia. Lopussa on esitetty projektimalleja, joiden toteuttamisessa XP:stä saattaisi olla apua. Toisaalta tarkoituksena on ollut myös suhtautua XP:hen terveen kriittisesti ja etsiä siitä mahdollisimman luotettavaa tietoa, jotta tiedettäisiin, milloin sen käyttö ei ole järkevää. Myös tällaisia projektimalleja ja reunaehtoja on esitetty tutkimuksen lopussa.

Avainsanat ja -sanonnat: Extreme Programming, projektinhallinta, ohjelmistokehitys.
CR-luokat: K.6.3, K.6.1, D.2.9.

Sisällysluettelo

1. Johdanto 1
1.1. Projektimuotoinen ohjelmistojen rakentaminen 1
1.2. Perinteisiä ohjelmiston elinkaarimalleja 3
1.2.1. Johdanto elinkaarimalleihin 3
1.2.2. Vesiputousmalli 3
1.2.3. Inkrementaalinen kehittäminen 5
1.2.4. Iteratiivinen malli 7
1.2.5. Evoluutiomalli 8
1.2.6. Prototyyppien tekeminen 9
1.2.7. Rational Unified Process 10
1.3. Ohjelmistoprojektien ongelmia 10
1.3.1. Ohjelmistoprojektien onnistumisesta 10
1.3.2. Epäonnistumisten yleisimpiä syitä 11
1.4. Tutkimusongelma 12
2. Extreme Programming -metodologia 13
2.1. Syntyhistoria 13
2.2. Perustana lean manufacturing 14
2.3. Metodologian arvot 16
2.4. Ohjelmiston rakentamisprosessi XP-metodologian mukaisesti 17
2.5. Henkilöstön roolit Extreme Programming -projektissa 21
3. Muita joustavia projektinhallintamenetelmiä 23
3.1. Menetelmille yhteisiä piirteitä 23
3.2. Adaptive Software Development 24
3.3. Crystal-metodologiaperhe 26
3.4. Dynamic Systems Development Method 28
3.5. Feature-Driven Development 30
3.6. Scrum 32
4. Vertailua muihin projektinhallinnan menetelmiin 35
4.1. Menetelmien vertailuperusteita 35
4.2. Vertailua perinteisiin menetelmiin 35
4.3. Vertailua muihin joustaviin menetelmiin 38
5. Extreme Programming -menetelmät tutkimusten ja kokemusten valossa 40
5.1. Iteratiivinen elinkaarimalli 40
5.2. Suunnittelupeli 42
5.3. Yhteisen metaforan käyttö 43
5.4. Yksinkertainen suunnittelu 43
5.5. Automatisoitu testaus 44

5.6. Refactoring 45
5.7. Pariohjelmointi 46
5.8. Koodin yhteisomistus 48
5.9. Jatkuva koodin integrointi 49
5.10. Koodauskonvention käyttö 50
5.11. Läheinen yhteistyö asiakkaan kanssa 51
5.12. Ihmisläheiset piirteet 53
6. Metodologian arviointia kokonaisuutena 55
6.1. Case-tutkimuksia XP-projekteista 55
6.2. Tutkimus useista XP-projekteista 57
6.3. XP:tä kohtaan esitettyä kritiikkiä 58
6.4. Extreme Programming ja ohjelmistoprojektien sopimusmallit 59
7. Yhteenveto 61
7.1. Projektimalleja, joihin Extreme Programming sopii 61
7.2. Projektimalleja, joihin Extreme Programming ei sovi 61
7.3. Teknisiä reunaehtoja 63
7.4. Loppupäätelmät 64

Viiteluettelo 65

1. Johdanto

1.1. Projektimuotoinen ohjelmistojen rakentaminen

Ymmärtääksemme ohjelmistoprojekteja on syytä ensin syventyä yleiseen projektin käsitteeseen. Suomen Standardoimisliiton projektitoimintasanasto [SFS 1981] määrittelee projektin seuraavasti:

> "Projekti on varta vasten muodostetun organisaation määrätarkoitusta varten toteuttama ainutkertainen hanke, jonka laajuus- ja laatutavoitteet sekä aika- ja kustannuspanokset on ennalta määritetty."

Ohjelmistoprojekteissa tämä määritelmä täyttyy hyvin. Sillä on tarkoitusta varten muodostettu projektiryhmä, jossa eri asiantuntijoilla on erilaisia rooleja. Ohjelmistoprojekti on ainutkertainen hanke, sillä vaikka monilla projekteilla onkin tavoitteena tuottaa lähes samanlainen ohjelmisto, ovat projektien toimintaympäristöt ja niistä aiheutuvat muuttujat joka kerta erilaiset. Projektilla on myös määritellyt tavoitteet sekä niiden tavoittamiseen tarvittavat aika- ja kustannustavoitteet. Kaikki tavoitteet on kirjattu projektin alussa syntyviin dokumentteihin.

Määritelmä vastaa myös siihen, miksi ohjelmistojen rakentaminen on perinteisesti ollut nimenomaan projektimuotoista toimintaa. Ohjelmiston rakentamisprosessilla nähdään olevan selkeä tavoite, selkeät alku- ja loppupisteet, ne tarvitsevat aina juuri tietynlaista asiantuntemusta onnistuakseen ja projektien laajuus- ja laatutavoitteet ovat projektia valmisteltaessa tiedossa. Ohjelmistojen monistettavuus kopioimalla takaa ainutkertaisuuden, samoin niiden monimutkaisuus, jolloin ei todennäköisesti synny kahta täysin samanlaista ohjelmistoa.

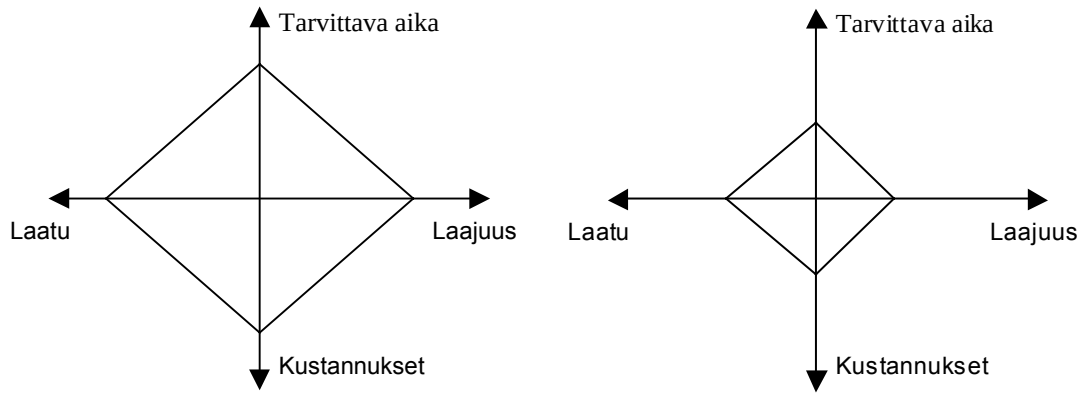
Käytetystä määritelmästä saadaan erotettua myös neljä projektiin vaikuttavaa tekijää, jotka ovat laajuus, laatu, tarvittava aika ja kustannus:

- Ohjelmiston laajuudella tarkoitetaan sitä, mitä asiakkaan toivomia ominaisuuksia ohjelmisto sisältää. Optimaalisen laajuuden käsitteellä tarkoitetaan, että ohjelmistossa on kaikki asiakkaan siihen toivomat ominaisuudet, mutta ei mitään ylimääräistä. Ylimääräiset ominaisuudet eivät enää olisi käyttökelpoisia, mutta niiden rakentaminen veisi aikaa ja rahaa.
- Ohjelmiston laadun ominaisuuksia määrittelemään tehty ISO 9126-standardi [Pressman, 2000] antaa laadukkaalle ohjelmistolle kuusi perusominaisuutta: toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys. Nämä käsitteet ovat kuitenkin varsin

moniselitteisiä. Lisäksi viime aikoina paljon huomiota saanut tietoturvallisuus on mainittu vain osana toiminnallisuutta ilman omaa ryhmäänsä.

- Tarvittavalla ajalla tarkoitetaan ohjelmiston rakentamiseen käytettyä aikaa esitutkimusprojektin aloittamisesta käyttöönoton tai mahdollisen koulutuksen päättymiseen.
- Kustannuksella tarkoitetaan ohjelmiston kokonaishintaa. Toimittajan projektityön lisäksi kustannuksiin tulee sisällyttää myös lisenssimaksut, asiakkaan kustannukset määrittely- ja käyttöönottotyöstä sekä mahdollisesta koulutukseen osallistumisesta.

Erilaiset organisaatiot asettavat projektiensa tavoitteita sen mukaan, kuinka paljon heillä on käytettävissään rahaa ja aikaa projektin läpivientiin. Ohjelmiston täydellinen laajuus ja huippulaatu ovat aluksi jokaisen organisaation tavoitteita, mutta usein joudutaan huomaamaan, että niihin tarvittavat aika- ja kustannusresurssit voivat olla organisaatiolle kestämättömiä. Esimerkiksi sellaiset virheettömyystavoitteet, jotka asetetaan sädehoitolaitteille, ovat erittäin kalliita ja aikaa vieviä niiden vaatiman huolellisen testauksen takia. Ohjelmiston laajuuden ja laadun kasvaessa kasvavat väistämättä myös sen vaatimat kehityskustannukset ja kehitykseen käytettävä aika. Näin ollen projektin tavoitteisiin liittyy valinta toisaalta tarvittavan ajan ja kustannusten, toisaalta ohjelmiston laadun ja laajuuden välillä. Jokainen organisaatio joutuu toisin sanoen valitsemaan ohjelmiston optimaalisen laajuuden ja laadun käytettävissään olevien aika- ja kustannusresurssien puitteissa. Kyseistä valintaa voidaan havainnollistaa kuvan 1 nelikentällä.



Kuva 1: Erilaisia projektitavoitteiden nelikenttämalleja.

Kuvassa 1 vasemmalla puolella kuvattu projekti tavoittelee korkeaa laatua ja toimintojen laajuutta. Projektista ollaan myös valmiita maksamaan enemmän ja hyväksytään se, että ohjelmiston rakentaminen kestää kauemmin. Tällainen

malli on tyypillinen silloin, kun ohjelmistoa käytetään kriittisissä toiminnoissa kuten ydinvoimaloissa tai henkeä ylläpitävissä sairaalalaitteissa. Malli on käytössä myös, kun ohjelmiston päivittäminen on kallista, vaikeaa tai jopa mahdotonta. Tällaisia tapauksia ovat esimerkiksi integroidut ohjelmistot matkapuhelimiin tai avaruusluotaimiin.

Kuvassa 1 oikealla puolella kuvattu projekti tavoittelee aikataulu- ja kustannussäästöjä tinkimällä ohjelmiston laajuudesta ja osin myös laadusta. Projektimalli on yleinen pienissä organisaatioissa, joilla ei ole riittävän suurta budjettia rakentaa ohjelmistoa kerralla kuntoon. Projektimallia käytetään myös, jos ohjelmiston valmistumisella on kiire ja sen kriittiset osat halutaan saada käyttöön mahdollisimman nopeasti. Tällaisissa tapauksissa puuttuvat ominaisuudet voidaan lisätä ohjelmistoon jatkoprojekteissa.

1.2. Perinteisiä ohjelmiston elinkaarimalleja

1.2.1. Johdanto elinkaarimalleihin

Ohjelmistojen kehitysprosessin tueksi on kehitetty erilaisia ohjelmistojen elinkaarimalleja. Elinkaarimallien tarkoitus on määritellä ne vaiheet, jotka tarvitaan ohjelmistoprojektissa asiakkaan toiveista aina valmiiseen ohjelmistoon saakka. Erilaisia elinkaarimalleja käytetään jatkuvasti kehitettäessä mitä erilaisimpia tuotteita projektiluonteisesti erilaisilla toimialoilla, eniten kuitenkin insinöörimaailmassa.

Ohjelmiston elinkaarimalleista on johdettu erilaisia projektinhallintamenetelmiä ja –metodologioita. Nämä tuovat elinkaarimalliin lisäarvona eri henkilöiden roolit ja tarkentavat usein niitä menetelmiä, joilla vaiheiden toteuttaminen suoritetaan ja niiden toteutuminen varmistetaan. Tässä tutkielmassa Extreme Programming -menetelmä esitetään projektinhallintamenetelmän tasolla. Sen vertailukohtana olevat menetelmät esitetään yksinkertaisuuden vuoksi elinkaarimallin tasolla, vaikka niistä monet voidaankin lukea piirteidensä vuoksi projektinhallintamenetelmiksi.

Tämä luku noudattelee kirjan Ohjelmistotuotanto [Haikala ja Marijärvi, 1998] esittämistapaa elinkaarimallien ja niiden sisällön suhteen. Elinkaarimallien nimeäminen poikkeaa hieman alkuperäisestä lähteestä, sillä samoista malleista käytetään yleisesti eri nimityksiä.

1.2.2. Vesiputousmalli

Vesiputousmalli on perinteinen ohjelmistoprojektin elinkaarimalli, jota on käytetty jo pitkään erilaisissa insinööriprojekteissa rakennettaessa siltoja, rakennuksia ja oikeastaan mitä tahansa tekniikkaa. Tätä kautta vesiputousmalli on tullut myös ohjelmistojen rakentamisen välineeksi. Vesiputousmalli on

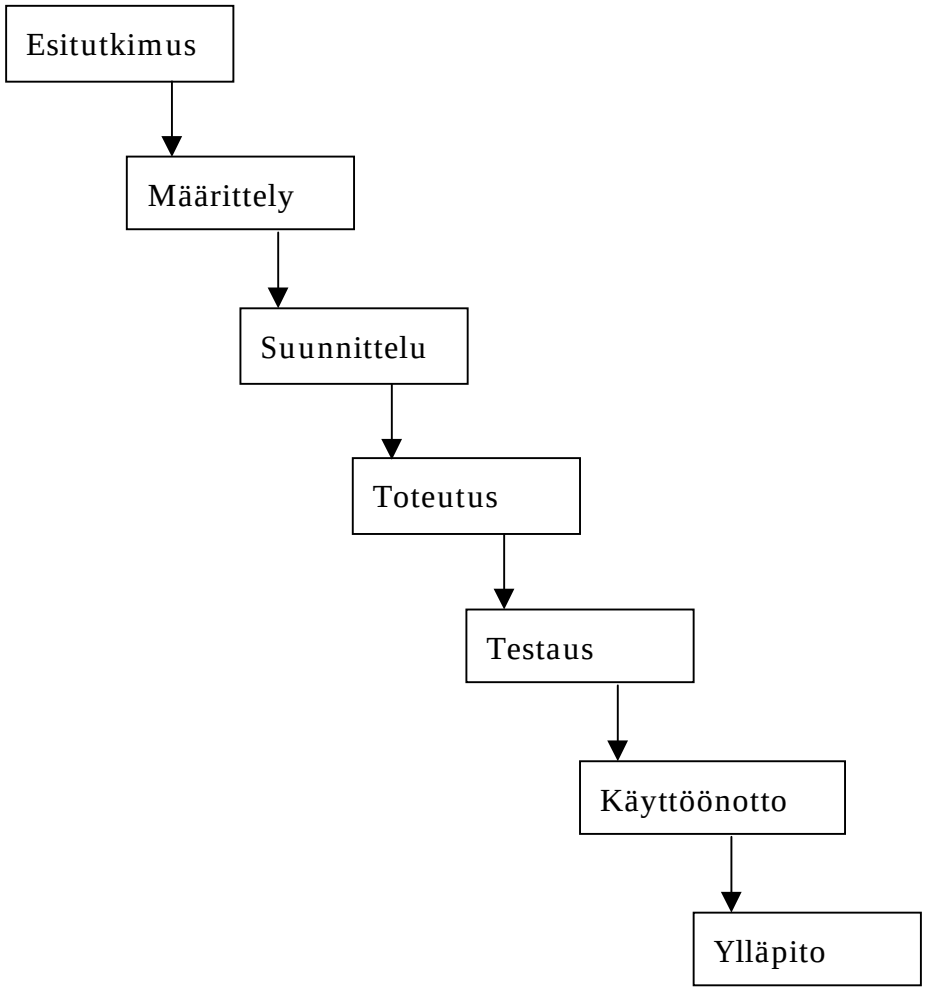
ensimmäinen ohjelmistokehityksessä käytetty selkeä malli ja se on tuonut järjestelmällisyytensä ansiosta tehokkuutta aiempiin, varsin kaoottisiin tapoihin kehittää ohjelmistoja. Nämä tavat saattoivat myös poiketa radikaalisti toimittajasta riippuen.

Vesiputousmallissa projekti jakautuu selvästi erillisiin vaiheisiin, jotka suoritetaan peräkkäin järjestyksessä ja kukin vaihe vain kerran. Eri vaiheet ovat:

- Esitutkimus, jonka tarkoituksena on tehdä kustannus-hyöty -analyysi projektista eli tutkia, onko ohjelmistoprojekti oikea ratkaisu esillä olevaan ongelmaan. Tarkoituksena on myös saada arvio projektiin tarvittavasta ajasta, henkilöstöstä ja kustannuksesta. Esitutkimusvaiheesta syntyy yleensä raportti ja usein myös alustava projektisuunnitelma.
- Määrittelyvaihe, jossa pyritään etsimään vastaus kysymykseen "Mitä ohjelmiston tulisi tehdä?". Määrittelyvaiheessa kerätään ohjelmistolle asetetut toiminnalliset, tekniset ja muut vaatimukset dokumenttiin nimeltä vaatimusmäärittely. Määrittelyvaiheessa tarkennetaan myös esitutkimusvaiheen projektisuunnitelma saadun uuden tiedon perusteella.
- Suunnitteluvaiheessa pyritään löytämään vastaus kysymykseen "Miten ohjelmisto tulisi rakentaa?". Tässä vaiheessa suunnitellaan ohjelma teknisesti, mutta ei varsinaisesti ohjelmoida vielä mitään. Vaiheen lopputuloksena on yksi tai useampia teknisiä dokumentteja, joissa määritellään ohjelmiston tekninen arkkitehtuuri, pysyvät datat ja niiden talletusratkaisut, käyttöliittymät, rinnakkaisuuden periaatteet, viestinvälitysmekanismit ja niin edelleen.
- Toteutusvaiheessa ohjelmoidaan varsinainen ohjelmakoodi. Vaiheen tuloksena ovat valmiit ohjelmarakenteet sekä niiden koodikommentit.
- Testausvaiheessa toteutusvaiheessa syntynyt koodi testataan. Erilaisia testauskriteereitä on useita: laajuus, toimintojen toimiminen oikeilla syötteillä, toimintojen toimiminen väärillä syötteillä, käytettävyys, tietoturva ja niin edelleen. Testausta voidaan tehdä useita kierroksia; näin tapahtuu kun löydetyt virheet korjataan ja korjatut toiminnot testataan uudelleen. Ennen varsinaista testausta projektissa tuotetaan testaussuunnitelma, jonka mukaan testaus suoritetaan. Testausvaiheessa syntyy puolestaan testausraportti, joka kertoo testien tuloksista. Testausta varten voidaan suuremmissa projekteissa laatia myös oma projektisuunnitelma.
- Käyttöönottovaiheessa ohjelmisto siirretään asiakkaan käyttöön ja annetaan mahdollisesti käyttäjille tarvittava koulutus ohjelmiston käyttöön.

- Ylläpitovaiheessa ohjelmisto on siirtynyt organisaation käyttöön ja sitä jatkokehitetään sekä käytössä huomattuja uusia toiveita toteutetaan. Myös käytössä havaittuja virheitä tai puutteita korjataan.

  Vesiputousmallin vaiheet on koottu kuvaan 2.



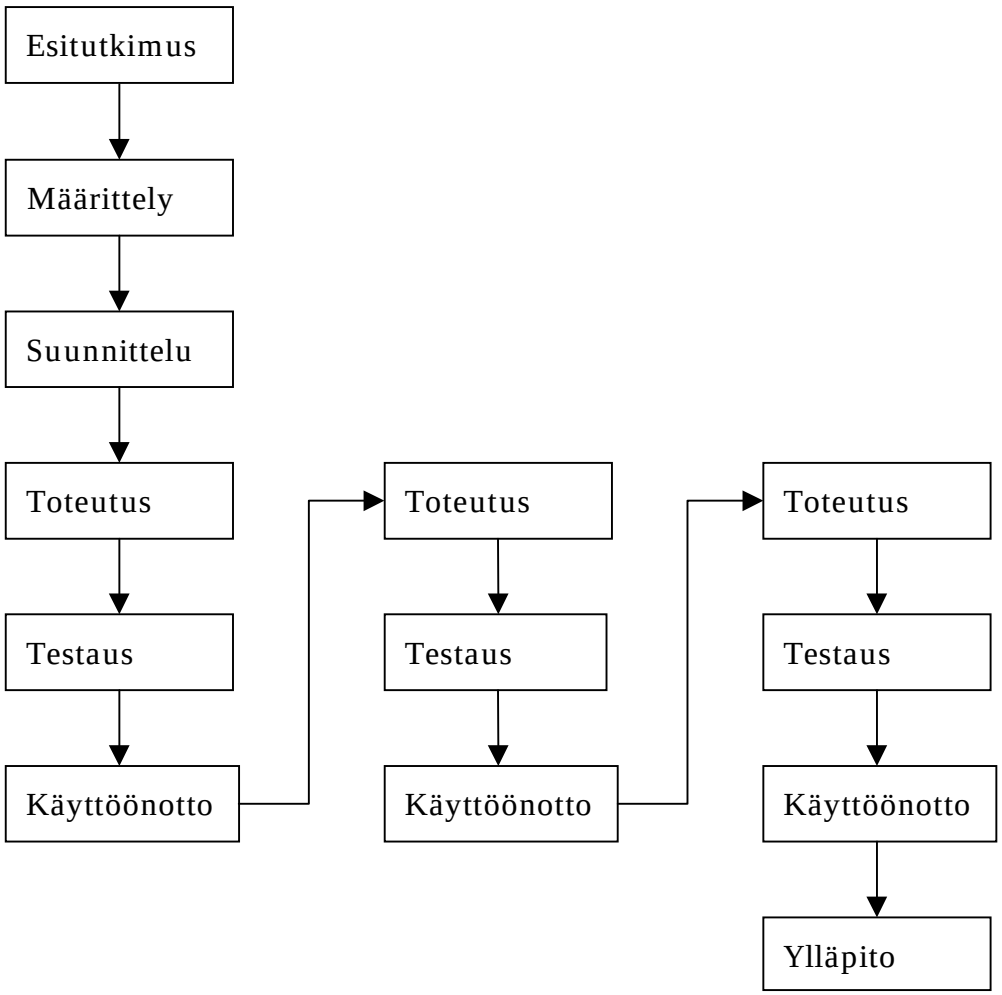
Kuva 2: Vesiputousmallin vaiheet.

1.2.3. Inkrementaalinen kehittäminen

Inkrementaalisen kehittämisen malli on muuten samanlainen kuin vesiputousmalli, mutta suunnittelun jälkeiset vaiheet suoritetaan useampaan kertaan. Ohjelmasta toimitetaan asiakkaalle useita versioita, joista jokainen sisältää edellisten versioiden toiminnallisuuden lisäksi uusia piirteitä ja ominaisuuksia. Kaikki toimitettavat versiot pohjautuvat samoihin vaatimuksiin, määrittelyyn ja suunnitteluun. Joissain yhteyksissä inkrementaalisen kehittämisen mallista käytetään myös nimitystä asteittaisen kehittämisen malli.

Tämä malli sopii erityisesti melko laajoihin projekteihin sekä tuotekehitykseen. Vesiputousmalliin verrattuna hyötyjä ovat esimerkiksi aikaisempi palaute asiakkaalta sekä pienempi yhtäaikainen resurssien tarve. Asiakas näkee osia ohjelmistosta valmiina ja voi puuttua mahdollisiin ongelmakohtiin aikaisemmin. Saadessaan ohjelmiston toimintoja tuotantokäyttöön jo kehitysaikana, asiakkaan kehityskustannusten takaisinmaksuaika lyhenee vesiputousmalliin verrattuna.

Huonona puolena inkrementaalisessa kehittämisessä on, että myös aikaisemmin kehitetyt osat on testattava uudelleen jokaisen uuden version yhteydessä, jotta voidaan varmistua kokonaisuuden toimivuudesta. Tämä kasvattaa ohjelmistokehityksen kokonaiskustannuksia, erityisesti mikäli testausta ei ole mahdollista automatisoida. Inkrementaalisen kehittämisen malli on esitetty kuvassa 3.



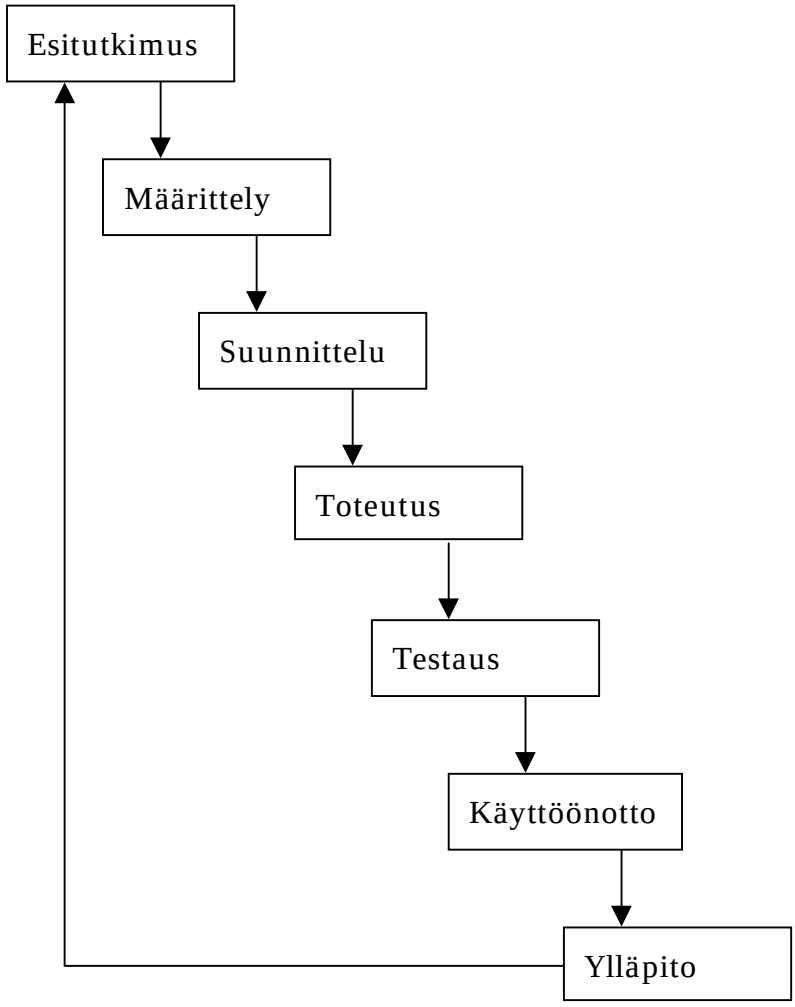
Kuva 3: Inkrementaalisen kehittämisen vaiheet.

1.2.4. Iteratiivinen malli

Iteratiivisessa mallissa toistetaan suunnitelmallisesti koko vesiputousmallia useita kertoja, kunnes ohjelmisto on kokonaan toteutettu. Jokaisella iteraatiokierroksella otetaan mukaan uusia vaatimuksia ja näin ohjelmistosta syntyy uusi versio joka kerta, kun kaikki vaiheet on saatu suoritettua. Iteratiivista mallia kutsutaan myös nimellä spiraalimalli.

Iteratiivinen malli sopii projekteihin, joissa vaatimukset on heikosti ymmärretty tai asiakas on vielä itsekin epävarma siitä, mitkä ovat ohjelmiston lopulliset vaatimukset. Tällaisia ovat esimerkiksi vaatimukset, jotka perustuvat uudistuvaan lainsäädäntöön, joka ei kuitenkaan ole vielä valmis. Malli on myös vesiputousmallia joustavampi, kun osa vaatimuksista on selvästi muita vaatimuksia vaikeampia toteuttaa tai teknologia ei ole riittävän kypsää koko järjestelmän toteuttamiseksi kerralla. Näissä tapauksissa ainakin osa järjestelmästä saadaan käyttöön ja siten osa kustannuksista katettua nopeasti, jolloin asiakas saa uusia kehityspanoksia tulevia versioita varten.

Iteratiivisen mallin vaiheet on esitetty kuvassa 4.



Kuva 4: Iteratiivisen mallin vaiheet.

Inkrementaalisen kehityksen mallilla ja iteratiivisella mallilla on paljon yhteistä. Molemmissa malleissa toistetaan tiettyjä vaiheita (iteratiivisuus) ja toistojen seurauksena ohjelmiin syntyy jokaisen iteraation jälkeen lisää ominaisuuksia (inkrementaalisuus). Mallien välillä on kuitenkin kaksi eroa. Ensinnäkin iteratiivisessa mallissa toistetaan kaikkia ohjelmiston rakentamisprosessin vaiheita, kun taas inkrementaalisessa mallissa toistetaan vain viimeisiä. Toiseksi iteratiivisen mallin kehityskierroksilla voidaan ohjelmistoon lisätä tai siitä jopa poistaa toimintoja, kun inkrementaalisessa mallissa rakennetaan ennalta päätetyt toiminnot suunnitelman mukaisesti. Näin ollen inkrementaalinen malli voidaan tulkita iteratiivisen mallin erikoistapauksena.

1.2.5. Evoluutiomalli

Evoluutiomalli muistuttaa jossain määrin luonnossa tapahtuvaa evoluutiota. Tässä mallissa ohjelmistoa rakennetaan pikku hiljaa ja annetaan se käyttäjille koekäyttöön. Saadun palautteen perusteella ohjelmistoa kehitetään eteenpäin lisäämällä siihen tarpeellisia toimintoja ja parantamalla toimintoja, jotka toimivat käyttäjien mielestä huonosti. Ero edellisiin malleihin verrattuna on siinä, että evoluutio ei ole tarkasti vaiheistettua vaan sitä tehdään ohjelman toimintojen mukaan. Evoluutiokierroksella ohjelmoidaan muutama uusi toiminto ja korjataan vanhoja, toimitetaan ohjelmisto asiakkaalle ja tehdään seuraava kierros palautteen perusteella. Suunnittelua ja testausta ei useinkaan tehdä samassa laajuudessa kuin edellisissä malleissa, mikä on tuottanut osittain aiheellistakin kritiikkiä mallia kohtaan. Malli on kuitenkin katsottu toimivaksi tilanteissa, joissa asiakkaan vaatimukset ohjelmistolle ovat erittäin epäselvät, mutta toisaalta yhteistyö asiakkaan kanssa sujuu erinomaisesti. Malli vaatii monissa tapauksissa asiakkaalta keskimääräistä enemmän teknistä ymmärrystä, sillä osa virheiden havaitsemisesta siirtyy toimittajan tekemästä testauksesta asiakkaan vastuulle.

Suurten ohjelmistokokonaisuuksien voidaan myös katsoa kehittyvän tietyllä tavalla evoluution mukaan. Esimerkiksi suosituista käyttöjärjestelmistä ja toimisto-ohjelmistoista ilmestyy parin vuoden välein uusia versioita, joihin on lisätty asiakkaiden tarvitsemiksi oletettuja toimintoja. Etenkin käyttöjärjestelmien uusissa versioissa tulee tukea uusimmille tekniikoille ja standardeille. Nämä ohjelmistot kehittyvät pitkän, yli kymmenen vuoden aikavälin aikana jatkuvasti suuremmiksi. Uusien standardien tuen lisääminen käyttöjärjestelmiin on perusteltua, mutta toimisto-ohjelmien evoluutiota on kritisoitu asiakkaan rahastamiseksi jatkuvan päivitystarpeen takia.

Joissain yhteyksissä näkee iteratiivisia ohjelmistokehityksen malleja kutsuttavan myös evoluutiomalleiksi, mutta tässä on iteratiiviset mallit erotettu

varsinaisesta evoluutiomallista siinä, että niissä toistetaan nimenomaan ohjelmistoprojektien vaiheita, kun taas evoluutiomalli on luonteeltaan hieman kaoottisempi. Nimeämiseen liittyvästä ristiriitaisuudesta voidaan päätellä, etteivät eri mallien väliset erot ole kovin selkeästi määriteltyjä.

1.2.6. Prototyyppien tekeminen

Prototyyppi on hyvin rajoitetulla toiminnallisuudella varustettu versio ohjelmistosta. Yleinen esimerkki prototyypistä on, että ohjelmiston käyttöliittymä on valmis, mutta sen takana olevaa sovelluslogiikkaa ja tietojen haku- ja tallennustoimintoja ei ole toteutettu, vaan niiden rajapinnan metodeista palautuu aina sama tulos.

Tällaisten prototyyppien avulla on hyvä selventää erityisesti käyttöliittymiin kohdistuvia vaatimuksia tilanteissa, joissa vaatimukset ovat epäselviä tilaajallekin. Prototyypin rakentaminen ei nykyaikaisilla Rapid Application Development (RAD) -työkaluilla kestä kovinkaan kauan, joten prototyyppejä voidaan iteratiivisesti parannella asiakkaan toiveiden mukaan kustannustehokkaasti useita kertoja ja näin parantaa prototyypin laatua, kunnes se on asiakkaan mielestä riittävän hyvä. Tällöin prototyyppi on tehnyt tehtävänsä ja se voidaan hylätä. Varsinainen ohjelmisto tulee rakentaa käyttöliittymältään juuri prototyypin kaltaiseksi, mutta siinä ei saisi olla mukana yhtään varsinaisen prototyypin koodia. Prototyyppien muuttaminen toimivaksi ohjelmistoksi on ongelmallista, sillä prototyypit eivät pysty ottamaan huomioon ohjelmiston arkkitehtuuria kokonaisuudessaan, mikä saattaa aiheuttaa ongelmia suorituskyvyn, sovelluslogiikan tai ylläpidettävyyden osalta. Esimerkiksi RAD-työkalujen tuottama käyttöliittymäkoodi on sinänsä virheetöntä, mutta se on rakennettu siten, että kaikki koodi suoritetaan tapahtumankäsittelijöissä. Näin ollen muutokset sovelluslogiikassa tai tietovarastoissa aiheuttaisivat muutostarpeen jokaiseen tapahtumankäsittelijään. Ohjelmiston koon kasvaessa satoihin tapahtumankäsittelijöihin tästä lähestymistavasta tulee nopeasti käyttökelvoton.

Prototyyppien käyttö sopii kuitenkin erinomaisesti projekteihin, joissa ohjelmiston käyttöliittymän määritykset ovat hyvin epäselviä, mutta asiakkaalla on halu ja mahdollisuus osallistua tiiviisti ohjelmiston alkuvaiheen prototyyppien kokeilemiseen. Prototyyppien rakentaminen ei valitettavasti sovi kovin hyvin muiden kuin käyttöliittymävaatimusten selvittämiseen. Prototyyppien eräänä uhkana on nähty myös se, että aikataulupaineessa olevissa projekteissa varsinainen ohjelmisto saatetaan kuitenkin rakentaa prototyypin koodin varaan, jolloin ohjelmakoodin taso heikkenee merkittävästi.

1.2.7. Rational Unified Process

Rational Unified Process eli RUP on Rational Softwaren kehittämä ohjelmistoprojektin hallintamalli. RUP perustuu kolmeen asiaan:

1. “Best practices” eli parhaiksi koetut työskentelytavat. Nämä ovat ohjelmistoprojektin iteratiivisuus, vaatimusten huolellinen hallinta, komponenttiarkkitehtuurien käyttö, visuaalinen ohjelmiston mallintaminen, jatkuva laadunvalvonta ja muutosten huolellinen hallinta.
2. Yhtenäisten, projektinhallintaan soveltuvien, Rationalin valmistamien työkalujen käyttö.
3. Rationalin konsultointipalveluiden käyttö.

Kuten edellisestä voidaan huomata, on prosessin kehittäneellä yhtiöllä metodologian suhteen selkeä kaupallinen intressi. RUP:tä voi kuitenkin soveltaa myös pelkästään parhaiden työskentelytapojen pohjalta ja niiden pohjalta on myös rakennettu RUP:tä soveltavia malleja. Näistä työtavoista vain iteratiivisuus on selkeästi vertailtavissa muihin projektimalleihin, sillä monissa malleissa ei oteta kantaa esimerkiksi siihen, mallinnetaanko ohjelmistoa visuaalisesti esimerkiksi UML-kaavioilla vai ei. Toisaalta kaikissa malleissa painotetaan vaatimusten ja muutosten hallinnan tärkeyttä projektin onnistumisen kannalta. Näin voidaan pitää perusteltuna myös näkemystä, jonka mukaan RUP ei ole varsinainen projektinhallintametodologia, vaan paremminkin tietynlainen työkalupakki, jota voidaan soveltaa tarpeen mukaan useissa iteratiivisissa malleissa.

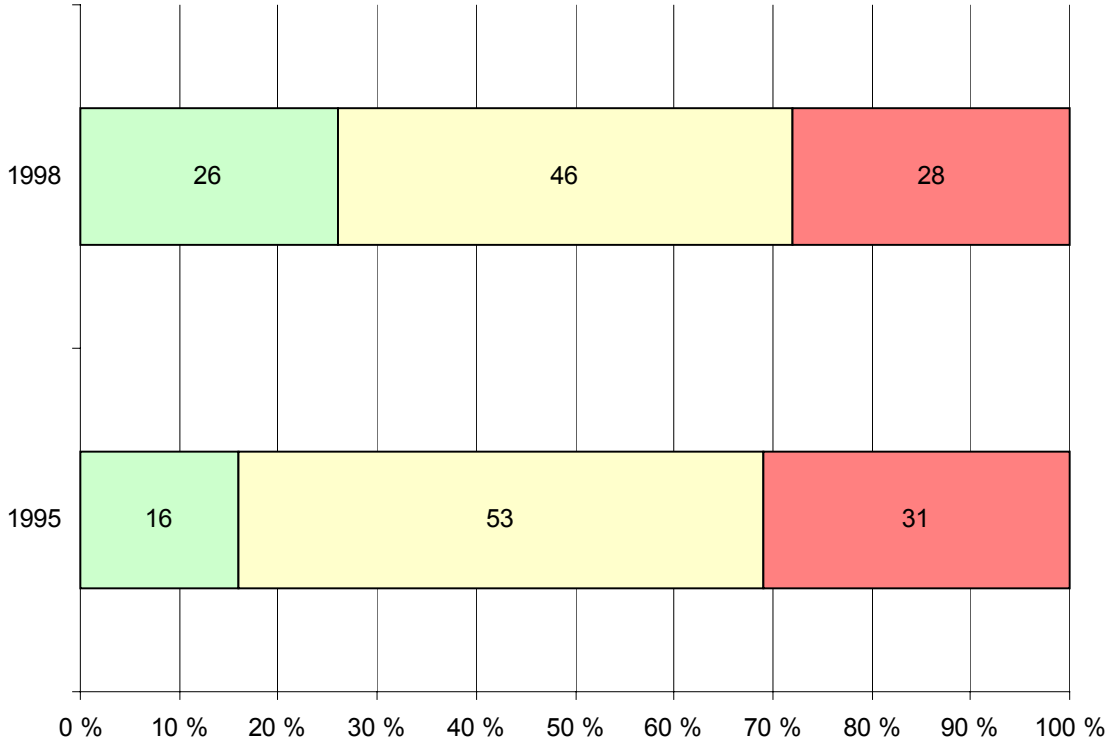
1.3. Ohjelmistoprojektien ongelmia

1.3.1. Ohjelmistoprojektien onnistumisesta

Ohjelmistoprojektien onnistumisesta on tehty useita tutkimuksia. Niiden tulokset ovat melko yhteneväisiä. Esimerkiksi tutkimusyhtiö Standish Groupin [Standish Group, 1995] mukaan ohjelmistojen rakentaminen projekteissa ei suju kovinkaan hyvin. Yhtiön raportissa jaettiin projektit kolmeen eri luokkaan niiden onnistumisen perusteella:

1. Projektit, jotka valmistuivat ajallaan sekä budjetin puitteissa.
2. Projektit, jotka valmistuivat, mutta ylittivät aikataulunsa, budjettinsa tai molemmat.
3. Projektit, jotka lopetettiin ennen niiden valmistumista ja katsottiin epäonnistuneiksi.

Raportin mukaan vain 16 % projekteista oli onnistuneita eli kuului ryhmään yksi. Ryhmään kaksi, eli aikataulunsa tai budjettinsa ylittäneisiin, kuului 53 %. Kuitenkin peräti 31 % projekteista kuului ryhmään kolme, epäonnistuneet projektit. Luvut olivat keskimääräistä suurempia suuremmissa projekteissa ja keskimääräistä pienempiä pienemmissä. Jatkotutkimuksessa [Standish Group, 1998] huomattiin, että vuonna 1998 tulokset olivat hieman parantuneet, mutta epäonnistumisia oli edelleen runsaasti. Tällöin ryhmään yksi kuului 26 %, ryhmään kaksi 46 % ja ryhmään kolme 28 % ohjelmistoprojekteista. Tutkimusten tulosjakauma on havainnollistettu kuvassa 5.



Kuva 5: Standish Groupin tutkimuksen tulosjakauma.

Ohjelmistoprojektien onnistumista mittaavissa tutkimuksissa on kuitenkin puutteensa. Standish Groupin tutkimus mittaa vain kahta luvussa yksi mainituista ohjelmistoprojektin neljästä muuttujasta, eli aika- ja kustannusarvioiden toteutumista. Sen sijaan ohjelmiston laatua tai laajuutta ei ole otettu huomioon. Tämä on luonnollista, sillä aika ja raha ovat ohjelmiston kvantitatiivisia mittareita ja siten helpompia mitata. Sen sijaan laatu ja osin myös laajuus ovat kvalitatiivisia, joita varsinkaan asiakkaan johto ei osaa mitata yhtä hyvin, koska ei usein itse käytä projekteissa syntyneitä ohjelmistoja.

Ohjelmistoprojekteissa joudutaan tilanteisiin, jolloin aikataulun pitäminen on ensiarvoisen tärkeää. Tällöin voidaan tinkiä ohjelmiston laadusta esimerkiksi testausta vähentämällä tai laajuudesta jättämällä joitain ohjelmistoon alun perin ajateltuja ominaisuuksia siitä pois. Kuitenkin tällaiset projektit näkyvät tutkimuksissa onnistuneina, joten todellinen tilanne voi olla jopa yllä esitettyäkin huonompi.

1.3.2. Epäonnistumisten yleisimpiä syitä

Samassa tutkimuksessa käytiin IT-johdolle suunnatun kyselytutkimuksen avulla läpi myös epäonnistumisten yleisimpiä syitä. Syitä on luonnollisesti useita, mutta kolme yleisintä vastausta erottuvat yleisyydellään muista:

1. Käyttäjien osallistumisen puute (12,8 %).
2. Epätäydelliset määrittelyt projektin alkuvaiheessa (12,3 %).
3. Määrittelyjen muuttuminen projektin edetessä (11,8 %).

Kohdat kaksi ja kolme liittyvät läheisesti toisiinsa, ja myös ensimmäinen kohta vaikuttaa määrittelyjen tasoon. Boehm [1981] esittää, että hyvinkin sujuneissa projekteissa noin 25 % vaatimuksista muuttuu määrittelyvaiheen jälkeen. Huonosti sujuneissa luvun voi odottaa olevan suurempi.

Ohjelmistojen vaatimukset siis muuttuvat hyvin usein. Tämä aiheuttaa väistämättä paluuta edellisiin vaiheisiin ja vaiheiden suorittamista useita kertoja. Perinteinen vesiputousmalli kuitenkin perustuu jokaisen vaiheen, myös määrittelyn, tekemiseen vain kerran. Näin ollen kaikki sitä seuraavatkin vaiheet tehdään ideaalitapauksessa vain kerran. Käytännössä tämä ei kuitenkaan tunnu toimivan, nimenomaan määrittelyjen epätäydellisyyden ja muuttumisen vuoksi. Esitetyissä iteratiivisissa malleissa tämä on otettu paremmin huomioon, mutta näissäkään ei oteta huomioon käyttäjien osallistumiseen vaikuttamista. Näin on syntynyt tarve uusille ohjelmistokehitysmalleille, jotka pyrkivät eliminoimaan ohjelmistokehityksen suurimpia riskejä.

1.4. Tutkimusongelma

Tässä työssä esitellään ohjelmistoprojektien hallintaan tarkoitettu Extreme Programming (XP) –metodologia ja muita sen kaltaisia joustavia menetelmiä. Työssä on tarkoitus tutkia sitä, onko XP:stä hyötyä tässä luvussa mainittujen ongelmien ratkaisemisessa ja millä tavoin se nämä ongelmat mahdollisesti ratkaisee. Tarkastelu tapahtuu etupäässä projektipäällikön näkökulmasta, mutta esiin on tuotu myös ohjelmoijien, asiakkaan ja johdon näkökulmia. Tarkoitus on löytää myös projektimalleja, joissa XP:tä voisi soveltaa. Samoin on tarkoitus myös löytää projektimalleja, joihin XP:n käyttö ei sovi.

2. Extreme Programming -metodologia

2.1. Syntyhistoria

Kuten vesiputousmallilla, myös Extreme Programmingilla on syntyhistoriansa teollisessa tuotannossa. Ensimmäiset XP:hen liittyvät ajatukset voidaan ajoittaa jo 1700-luvun loppuun, jolloin Eli Whitney keksi rajapintojen ja niihin sopivien, keskenään vaihdettavien osien teorian. Näihin ajatuksiin pohjaa suurin osa nykyaikaisesta teollisesta suunnittelusta ja tuotannosta. Teorian sovelluksia tietojenkäsittelytieteessä ovat esimerkiksi PC-arkkitehtuuri ja olio-ohjelmointikielten rajapinnat ja abstraktit luokat.

Teollistumisen alussa 1800-luvun lopulla Frederick Taylor loi idean tieteellisestä johtamisesta (scientific management). Tieteellisessä johtamisessa tutkittiin työntekijöiden toimia tarkasti kellon ja työssä toistuvien liikesarjojen analysoinnin avulla. Analyysien perusteella pyrittiin optimoimaan materiaalien virtoja ja työntekijöiden tehtävien suoritustapaa ja -järjestystä. Tämä toi suuren lisäyksen tuottavuuteen organisoimattomaan toimintaan verrattuna, mutta tieteellinen johtaminen näki työntekijät ainoastaan toistoon kykenevinä mekaanisina koneina eikä ottanut huomioon työn henkistä puolta.

1900-luvun alussa tapahtui eräs teollistumisen suurimmista läpimurroista, kun Henry Ford kykeni tuottamaan autoja ennennäkemättömän tehokkaasti. Hän yhdisti tuotannontekijät saumattomaksi ketjuksi ja pystyi näin hyödyntämään liukuhihnoja aivan uudella tavalla.

Maailman muuttuessa Fordin mallissa alkoi kuitenkin ilmetä puutteita. Se soveltui ainoastaan täysin samanlaisten tuotteiden valmistukseen, mutta tuotekehitys tuotti joka vuosi hieman parannettuja automalleja, joiden vaatimiin muutoksiin Fordin tuotantojärjestelmät sopeutuivat huonosti. Myös työläisten tietoisuus oikeuksistaan työnsä sisällön suhteen kasvoi ja ammattiliitot ajoivat tehtaiden työoloihin parannuksia, joihin Fordin malli ei voinut vastata. Näin ollen General Motors ohittikin Fordin tuottavuudessa jo 1930-luvulla alkuperäistä Fordin mallia hieman parantamalla.

Japanilaiset kiinnittivät huomionsa Yhdysvaltain teolliseen tuotantokykyyn toisen maailmansodan aikana ja alkoivat sen jälkeen uudistaa omaa teollista järjestelmäänsä. He huomasivat Fordin menetelmän edut, mutta samalla tunnistivat myös sen puutteet. Erityisenä puutteena nähtiin, että työntekijöiden huono kohtelu ei soveltunut japanilaiseen arvomaailmaan eikä heidän ammattitaidostaan tai kokemuksestaan saatu koneellisessa prosessissa juuri mitään hyötyä irti. Näin Japanissa aloitettiin Fordin menetelmän kehittäminen.

2.2. Perustana lean manufacturing

Toyota Motor Companyssa Taichii Ohno ja Shigeo Shingo löysivät vastaukset Fordin mallin ihmisten kohtelua ja tuotteiden erilaisuutta koskeviin ongelmiin. Havaintojensa pohjalta he kehittivät järjestelmän nimeltään Toyota Production System (TPS), joka otettiin menestyksekkäästi käyttöön Toyotalla samoin kuin monissa muissakin japanilaisissa yrityksissä. Siitä alettiin käyttää nimeä Kaizen, joka tarkoittaa jatkuvaa kehitystä. Järjestelmää pidetään erittäin tärkeänä tekijänä niin sanotun Japanin talousihmeen synnyssä toisen maailmansodan jälkeen.

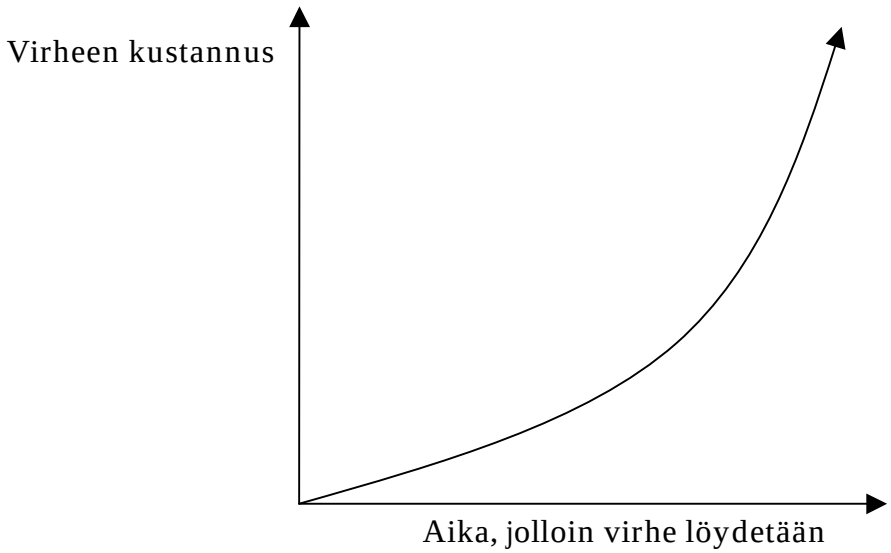
Autoteollisuuden syntyhistoriasta kertovassa kirjassaan Womack [1990] esittelee termin lean manufacturing, vapaasti suomennettuna kevyt valmistaminen. Tällä termillä pyrittiin yleistämään Toyota Production Systemin periaatteet niin, että niitä voidaan soveltaa useilla eri teollisuuden aloilla. Extreme Programming perustuu lean manufacturing -ajatteluun, samoin kuin kaikki muutkin uudet ja joustavat ohjelmistoprojektien mallit.

Lean manufacturingin perustavana ideana on jätteen (waste) vähentäminen. Jäte on tässä laajempi käsite kuin vain yli jääneet materiaalit, epäonnistuneet tuotteet, ympäristöön joutuneet saasteet ja hukkaenergia. Jätteellä ymmärretään myös turhaa työtä sekä turhaan odottamiseen menevää aikaa niin ihmisiltä, koneilta kuin itse tuotteiltakin. Jätettä vähennetään seuraavan viiden eri lean manufacturing -periaatteen avulla.

1. Työn tekeminen modulaarisesti työpisteissä (workcell). Työpisteen ideana on, että yhdessä pisteessä voidaan tehdä yksittäinen työ mahdollisimman täydellisesti tai ainakin minimoida prosessissa tarvittavien työpisteiden määrä. Näin tiedon ja materiaalien kulku prosessin läpi on mahdollisimman mutkatonta. Työpisteajattelun periaatteena on siis asioiden pitämiseen mahdollisimman yksinkertaisina ja samaan työhön osallistuvien henkilöiden työskentely samassa paikassa.
2. Kanban-toiminnanohjaus. Kanbanin perusperiaate on sama kuin nykyaikaisella supermarketilla ja se perustuu välittömän palautteen mahdollistamiin nopeisiin sykleihin. Vähittäiskaupan logistiikassa tuotteet jaetaan keskusvarastolta marketteihin korkeintaan muutaman päivän sykleillä, usein päivittäin. Lyhyessä ajassa tuotteita ei ole ehditty myydä suuria määriä, joten nopea toimitussykli antaa mahdollisuuden kuljettaa kerralla useampia erilaisia tuotteita. Kun asiakas ostaa tuotteen, kassajärjestelmästä lähtee automaattisesti signaali keskusvarastolle ja tehtaalle, joille on päivän päätteeksi kerääntynyt täydellinen tieto myydyistä tuotteista. Välittömän palautteen ansiosta

seuraavan päivän toimituksella voidaan täydentää supermarketin varasto ja suunnitella tarkemmin tulevaa tuotannon tarvetta. Näin toimittaessa varastot eivät lopu kesken mutta eivät myöskään kasva liian suuriksi. Sama idea on suoraan sovellettavissa tehdasautomaation tuotantolinjoihin, joissa työpisteillä on tarvittavia syötteitä ja ne tuottavat tulosteita syötteiksi seuraaville työpisteille.

3. Total Quality Management (TQM) ja sen perustana oleva malli laskea virheistä syntyviä kustannuksia. Mallissa virheistä syntyviin kustannuksiin lasketaan kaikki työntekijöiden virheen korjaamiseksi eri vaiheissa tarvittavat aika- ja muut resurssit, jolloin päädytään lähes poikkeuksetta suureen rahasummaan. Virheen ehkäisemiseksi kannattaa siis panostaa resursseja kunnes rajakustannus eli virheen kokonaishinta tulee vastaan. Sama idea on taustalla myös Boehmin kuuluisassa kuvaajassa virheen korjaamishinnasta ohjelmistoprojektin eri vaiheissa [Boehm, 1981]. Tässä kuvaajassahan virheen hinta nousee varsin nopeasti, kun projektissa edetään pitemmälle.



Kuva 6: Virheen korjauskustannukset kuluneen ajan funktiona.

4. Nopeat ja helpot asennukset. Asennukset eli tuotteesta toiseen siirtyminen tuotantolinjalla on vaikeaa ja riskialtista. Tämän seurauksena samaa tuotetta on tehty suuria määriä ja pidetty asennusten määrä vähäisenä. Tämä kuitenkin johtaa suuriin varastoihin, eli tuote viettää varastossa pitkän ajan ja sitoo jatkuvasti pääomaa. Varastossa myymättä makaava tuote syö resursseja ja tulee siis ymmärtää jätteeksi,

jota tulee vähentää tekemällä asennuksista helpompia. Tällöin voidaan laajentaa tuotevalikoimaa ja lyhentää varastointiaikoja.

5. Tiimien kehittäminen. Henkilöstön osaamisesta huolehtiminen on keskeistä kaikessa toiminnassa, erityisesti nopeasti muuttuvissa ympäristöissä kuten ohjelmistoteknologiassa. Osaamisesta huolehtiminen synnyttää kompetenssia, joka puolestaan johtaa ylpeyteen omasta työstä. Kaikki tämä yhdessä tuottaa onnistumisen elämyksiä henkilöstölle ja onnistunutta työtä yritykselle.

2.3. Metodologian arvot

Kirjassaan Extreme Programming Explained – Embrace Change [Beck, 1999] Kent Beck esittelee aluksi Extreme Programming -metodologian perustana olevat neljä arvoa, jotta sen käytäntöjen ymmärtäminen olisi helpompaa. Tässä noudatetaan samaa periaatetta. Metodologian neljä arvoa ovat kommunikaatio, yksinkertaisuus, palaute ja rohkeus.

Sanonnan mukaan 80 % maailman ongelmista johtuu huonosta tai riittämättömästä kommunikaatiosta. XP:ssä noudatetaan työtapoja, joita on mahdotonta tehdä ilman kunnollista kommunikaatiota ihmisten välillä. Silti kommunikaation puute on vaarana myös XP-projekteissa, koska dokumentointia harrastetaan vähemmän kuin muissa menetelmissä. XP luottaakin enemmän keskusteluun kuin dokumenttien kirjoittamiseen. Kommunikaatiota toki arvostetaan sekä perinteisemmissä ohjelmistonkehitysmalleissa että XP:ssä.

Yksinkertaisuudella viitataan siihen, että järjestelmä toteutetaan yksinkertaisimmalla mahdollisella toimivalla tavalla, aivan kuten lean developmentissa. Toisin sanoen XP-suunnittelussa ei saisi koskaan varautua sellaisiin vaatimuksiin, joita ei suunnitteluhetkellä ole tiedossa, eli järjestelmää ei tietoisesti suunnitella laajennettavaksi eri suuntiin. Tämä on eräänlainen vedonlyönti sen puolesta, että muutosten aiheuttama lisätyö myöhemmin on pienempi kuin laajennettavuuden lisääminen heti suunnitteluvaiheessa. Tätä vedonlyöntisuhdetta parantaa XP:n käyttämä refactoring-menetelmä eli ohjelmiston arkkitehtuurin muokkaaminen tarkkoja sääntöjä käyttäen. Tämä arvo on tietoisessa ristiriidassa perinteisten ohjelmistonkehityksen mallien kanssa, jotka kannustavat rakentamaan helposti laajennettavia järjestelmiä.

Kolmantena arvona on palaute, jota XP:ssä kerätään usein ja heti tehdyn työn jälkeen, aivan kuten Kanban-toiminnanohjauksessa. Palautetta kerätään toimittajalta, asiakkaalta sekä itse rakennettavalta ohjelmistolta. Rakennettavan järjestelmän tilasta saadaan palautetta päivittäisten integrointien ja testausten kautta. Myös toimittaja ja asiakas saavat nopeaa palautetta toisiltaan.

Rohkeus on neljäs XP-metodologian arvo. Koska XP on monilta periaatteiltaan varsin radikaali, jo pelkkä menetelmän käyttäminen vaatii rohkeutta. Rohkeutta XP vaatii myös asiakkaalta, joka ei voi XP:n luonteen vuoksi saada heti projektin aluksi selkeää lupausta koko ohjelmiston kustannuksista ja työmääristä, vaan on itse mukana vaikuttamassa niihin projektin aikana.

2.4. Ohjelmiston rakentamisprosessi XP-metodologian mukaisesti

XP-metodologian mukainen rakentamisprosessi muistuttaa iteratiivisen kehityksen ja evoluutiomallin vastaavia. Prosessi ei ota kantaa siihen, miten esitutkimusvaihe tehdään, vaan kustannus-hyöty -analyysi voidaan tehdä perinteisin menetelmin. XP:n erot muihin malleihin alkavat määrittelyvaiheesta.

Määrittelyvaihe XP:ssä on samantapainen kuin evoluutiomallin määrittelyvaihe. Koko ohjelmistoa ei määritellä kerralla, vaan projektin alussa määritellään vain ensimmäiseen versioon tarvittavat toiminnot. Koska uusia versioita annetaan asiakkaalle hyvin usein, kyse on ensimmäisen version kohdalla korkeintaan puolen vuoden projektista. Seuraavat versiot pyritään saamaan asiakkaalle aikavälillä yhdestä kolmeen kuukautta.

Määrittelyvaihetta kutsutaan nimellä suunnittelupeli (planning game). Tässä vaiheessa asiakas esittää tarvittavat toiminnot ja priorisoi ne. Tämän jälkeen ohjelmiston toimittaja antaa jokaisesta toiminnosta työmäärä-, aikataulu- ja kustannusarvion sekä kertoo asiakkaalle niiden muut tekniset vaikutukset, kuten esimerkiksi tarvittavat lisenssimaksut. Aikatauluarvion tekee toteuttajaryhmä yhdessä. Näiden tietojen avulla asiakas päättää sen, mitä toimintoja seuraavaan versioon tulee. Asiakkaalle annetaan aikatauluarvioiden perusteella mahdollisuus myös päättää siitä, koska tuleva versio on käytettävissä. Ne toiminnot, jotka eivät tule vielä seuraavaan versioon, kirjataan ylös ja säästetään seuraavaa suunnittelupeliä varten. Näin jaetaan vastuu toiminnoista ja käytetyistä teknologioista selkeästi: asiakas määrittelee toiminnot ja niiden toteuttamisjärjestyksen, toimittaja määrittelee tarvittavat teknologiat. Aikataulu määritellään yhteisesti: toteuttajat määrittelevät, kuinka paljon aikaa toimintojen tekeminen vie. Asiakas puolestaan kokoaa näistä toiminnoista haluamansa kokonaisuudet ja pääsee näin päättämään, koska kokonaisuuden on oltava valmis.

Yhteisen kielen löytämiseksi projektiryhmä ja asiakas sopivat kielellisestä metaforasta, jonka avulla mallinnetaan järjestelmän toimintaa. Metaforalla tarkoitetaan tunnetun käsitteen käyttöä auttamaan vähemmän tunnetun käsitteen ymmärtämistä niiden yhteisten piirteiden perusteella. Metafora on suunniteltu abstrahoimaan järjestelmän tekniset ominaisuudet ja pitämään kommunikaation mahdollisimman yksinkertaisena ja sujuvana. Ehkä paras

esimerkki löytyy nykyisistä verkkokaupoista, joissa metaforana käytetään valintamyymälää. Kaikista verkkokaupoistahan löytyy valintamyymälöistä tuttua terminologiaa: ostoskori, osasto, kassa ja niin edelleen.

Määrittelyssä käytetään käyttötapaustekniikkaa (use cases). Käyttötapaus on määritelty sarjaksi järjestelmän toimintoja, joiden tehtävänä on tuottaa mitattavaa hyötyä järjestelmän käyttäjälle [Jacobson *et al*., 1995]. Tarkemmin sanottuna käyttötapauksessa kirjataan selväkielisenä, asiakkaan ymmärtämänä ei-teknisenä tekstinä ne asiat, jotka yhden tehtäväkokonaisuuden aikana käyttäjän näkökulmasta halutaan tehdä ja millaiset tulokset hän haluaa tehtävästään nähdä.

XP:ssä käyttötapauksia kutsutaan nimellä tarina (story). Loppukäyttäjä kirjoittaa käyttötapaukset, joskin ohjelmoijat voivat myös ehdottaa niitä. Käyttötapauksiin ei tällöin kirjata ylös teknisiä yksityiskohtia kuten erilaisia poikkeuksia tai virhetilanteita. Kun asiakas on kirjoittanut käyttötapaukset, hän antaa ne toimittajalle, joka arvioi ne ja antaa asiakkaalle niistä palautetta, joka on yksi XP:n arvoista. Tällä tavalla käyttötapauksia voidaan iteroida ja niiden laatua parantaa. Kun käyttötapaukset ovat riittävän yksityiskohtaisia toteutusvaihetta varten, voidaan ohjelmointi aloittaa. Teknisen suunnittelun vaihetta ei XP:ssä ole, vaan se tehdään erikseen jokaisen käyttötapauksen pohjalta. Koko järjestelmää ei suunnitella kokonaisuutena lainkaan.

Toteutusvaiheessa tulevat esiin XP:n radikaaleimmat erot muihin metodologioihin verrattuna. Ohjelmointivaiheen aluksi, ennen kuin riviäkään lopullista ohjelmakoodia on kirjoitettu, ohjelmoija kirjoittaa käyttötapaukselle vastaavan moduulitestitapauksen (test case). Vasta tämän jälkeen ohjelmoidaan varsinainen ohjelmakoodi.

Ohjelmakoodi kirjoitetaan yksinkertaisuuden periaatetta noudattaen tekemällä ohjelmointiongelmaan yksinkertaisin mahdollinen toimiva ratkaisu. Ohjelmoinnissa ei siis esimerkiksi koskaan käytetä olio-ohjelmoinnille tyypillistä uudelleenkäytön periaatetta, mikäli uudelleenkäytön kohdetta ei ole jo valmiiksi tiedossa. Arkkitehtuuri suunnitellaan vain niiden toimintojen perusteella, jotka ovat tulossa seuraavaan ohjelmiston versioon. Yksinkertaisuuden arvo toteutuu selvimmin juuri tässä. Lisäksi ohjelmointi vain tätä hetkeä varten, varautumatta myöhemmin tarvittaviin arkkitehtuuri-muutoksiin, vaatii rohkeutta.

Kun käyttötapaukseen tarvittava koodi on valmistunut, se integroidaan välittömästi muuhun ohjelmakoodiin. Kirjoitettu testitapaus puolestaan integroidaan automatisoituun testausjärjestelmään, jollaisia ovat esimerkiksi Java-ohjelmille suunnitellut JTest [JTest] ja JUnit [JUnit]. Kun testitapaukset on integroitu, ajetaan välittömästi kaikki testausjärjestelmän sisältämät testit.

Mikäli kaikki testit onnistuvat, toteutus on onnistunut ja ohjelmointivaihetta voidaan jatkaa seuraavalla käyttötapauksella. Mikäli jokin testeistä palautti virheen, virhe korjataan välittömästi ja testaamista jatketaan kunnes virheitä ei enää tule. Tämä tukeekin hyvin edellä esitettyä palautteen arvoa.

Lean developmentin tapaan nopeissa sykleissä tapahtuvasta jopa useita kertoja päivässä tapahtuvasta integroinnista ja testaamisesta saavutetaan etuina koodin jatkuva virheettömyys ja jatkuva tieto järjestelmän tilasta. Näin eri ohjelmoijien koodinosat ovat aina synkronisia keskenään ja mahdolliset arkkitehtuurivirheet päästään korjaamaan nopeasti. Tämä tietenkin sillä edellytyksellä, että laaditut automaattiset testit saavuttavat riittävän testikattavuuden.

Ohjelmoidessa kaikki lopulliseen ohjelmaan menevä ohjelmakoodi tuotetaan pariohjelmointina, eli kahden ohjelmoijan työskennellessä yhdessä, yhdellä tietokoneella, saman ohjelmakoodin parissa. Kun toinen parista ohjelmoi, toisen tehtävänä on samalla lukea kaikki toisen kirjoittama koodi ja korjata havaitsemiaan virheitä saman tien. Ohjelmoijat myös keskustelevat keskenään parhaista toteutusvaihtoehdoista.

Pariohjelmointi johtaa osaltaan myös siihen, että yksittäisillä ohjelmiston osilla ei ole varsinaista sitä omistavaa ohjelmoijaa, vaan kaikki koodi on kaikkien ohjelmoijien yhteisomistuksessa. Tällöin myös kuka tahansa projektiryhmän jäsen voi muokata toisen jäsenen tekemään koodia paremmaksi. Koodin yhteisomistus, samoin kuin pariohjelmointi, vaatii yhteisen ohjelmointityylin standardoinnin määrittelemään, miltä koodin tulisi näyttää.

Ohjelmoija ei saa kuitenkaan muokata koodia mielensä mukaan vaan ainoastaan refactoring-menetelmän mukaisesti. Refactoring on Martin Fowlerin kehittämä [Fowler, 1999] ohjelmakoodin paremmaksi muokkaamiseen tarkoitettu menetelmä. Fowler itse määrittelee sen prosessiksi, jossa muokataan ohjelman rakennetta paremmaksi muuttamatta sen toiminnallisuutta. Refactoring sisältää useita kymmeniä sääntöjä tähän tarkoitukseen. Tyypillinen esimerkki säännöstä on ohje siitä, miten yksi oliokielen luokka jaetaan pääluokkaan ja sen perivään luokkaan.

XP kannustaa voimakkaasti käyttämään ohjelmoinnissa ohjelmistokehityksen rutiineja automatisoiva apuvälineitä. Automaattiset testaustyökalut mainittiin aiemmin ja versionhallintatyökalujen käyttö lienee jo perusedellytys kaikissa projekteissa. Lisäksi usein tapahtuvan integroinnin takia myös testattavan versioehdokkaan, ns. “buildin”, rakentamiseen suositellaan käytettäväksi sopivaa automaattityökalua, esimerkiksi Ant-skriptiä

[Ant]. Automaattisten työkalujen avulla voidaan helpottaa asennuksia lean developmentin tapaan, mikä mahdollistaa nopeamman asennussyklin.

XP suosittelee koodin optimoinnin tapahtuvan vasta, kun kaikki toiminnallisuus on toteutettu ja toimivaksi testattu. Näin optimointiin käytetty työ ei mene hukkaan, mikäli vaatimukset muuttuvat projektin aikana. Optimointi saattaa joissain tapauksissa tehdä ohjelmakoodista vaikeammin ymmärrettävää, mikä vaikeuttaa jatkokehitystä.

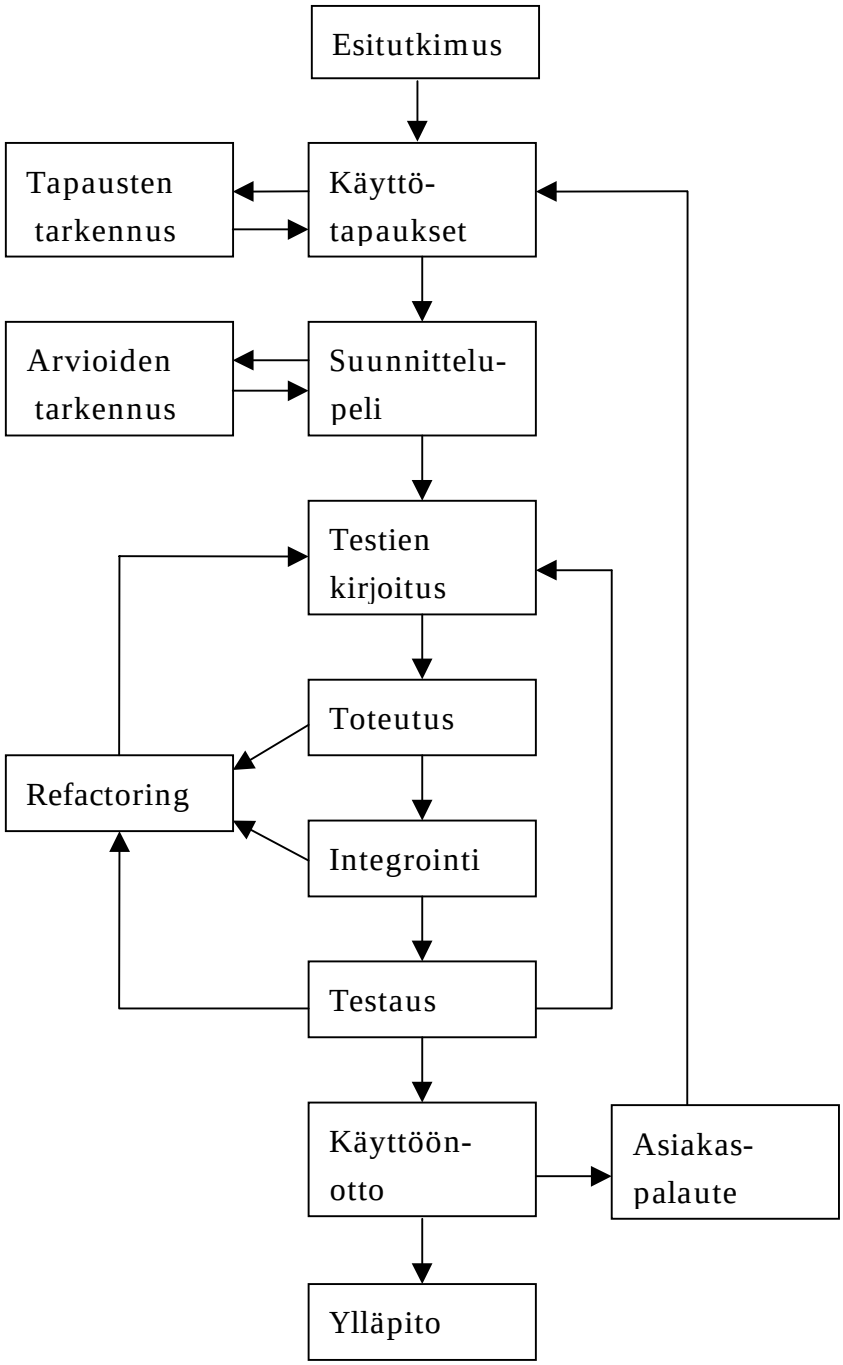
Kun koodi on moduulitestattu ja optimoitu, sille tehdään vielä toiminnallinen testaus (functional testing). Asiakkaan tehtävänä on ollut käyttötapauksen valmistumisen jälkeen kirjoittaa käyttötapausta vastaava toiminnallinen testi, joka perinteisesti tehdään käyttöliittymälle syötteitä antaen ja tiettyjä tuloksia odottaen. Testin suorittaa asiakas projektiryhmän avustamana. Projektiryhmä korjaa poikkeamat odotetuista tuloksista saman tien ja asiakas testaa ne uudelleen, kunnes poikkeamia ei enää esiinny.

Ohjelmointi- ja testausvaihe päättyy, kun kaikki versioon aiotut käyttötapaukset on ohjelmoitu, optimoitu, niiden testit kirjoitettu ja sekä moduuli- että toiminnalliset testit ovat menneet läpi. Tällöin ohjelmisto voidaan asentaa asiakkaalle ja aloittaa seuraava kehityskierros uudella suunnittelupelillä.

Edellisissä kappaleissa ei tarkoituksella kerrottu lainkaan erilaisista projektin tuottamista dokumenteista. XP noudattaa lean developmentin jätteen vähentämisen periaatetta dokumenttien suhteen siten, että se katsoo jätteeksi kaikki dokumentit, joista ei ole hyötyä projektin valmistuttua. Jotkut dokumentit ovat hyödyllisiä ohjelmiston ylläpidon kannalta, eikä XP varsinaisesti kiellä dokumentointia. Se ei kuitenkaan anna ohjeita siitä, mitä dokumentteja projektissa tulisi kirjoittaa käyttötapausten kirjallista dokumentointia lukuun ottamatta. Kaikki muu projektin kommunikaatio tulisi tapahtua henkilökohtaisesti ihmisten kesken, ei dokumenttien välityksellä.

On huomattava, että XP ei rajoitu ainoastaan ohjelmistojen rakentamiseen, vaan lean developmentin tavoin kiinnittää huomiota myös projekteissa työskentelevien ihmisten hyvinvointiin. Eräs XP:n periaatteista onkin 40 tunnin työviikko, joka tarkoittaa että projekteissa ei saa tehdä ylitöitä, ei ainakaan kahta viikkoa peräkkäin. Beck onkin kuvannut XP:tä ilmauksella ”humanistinen tapa tehdä ohjelmia” [Beck, 1998].

Extreme Programming -metodologian vaihejakomalli on esitetty kuvassa 7.



Kuva 7: Extreme Programming -metodologian vaiheet.

2.5. Henkilöstön roolit Extreme Programming -projektissa

Beck [1999] kuvailee myös henkilöstön rooleja XP-projektissa. Rooleja nimetessä on syytä muistaa, että yhdellä henkilöllä voi olla projektissa useita rooleja.

- Ohjelmoija on projektiryhmän peruspilari. Hän suunnittelee ja kirjoittaa koodin muiden ohjelmoijien kanssa. Hänen vastuullaan on myös

ohjelmakoodin moduulitestaus. Hän antaa arviot toimintojen toteuttamiseen tarvittavasta ajasta. XP-projektissa kaikki projektiryhmän jäsenet ohjelmoijat mukaan luettuna kommunikoivat suoraan asiakkaan kanssa, kun perinteisesti yhteyshenkilönä on toiminut projektipäällikkö.

- Asiakkaan vastuulla on antaa käyttötapaukset projektiryhmälle ja suunnitella toiminnallisia testejä. Hänen vastuullaan on päättää lisäysten sisällöstä, priorisoida käyttötapauksia ja päättää, milloin jokin käyttötapaus on hyväksytysti toimitettu.
- Testaajat auttavat asiakasta toiminnallisten testien suunnittelussa ja voivat myös suorittaa varsinaisen toiminnallisen testauksen. He tiedottavat muille ryhmän jäsenille löydetyistä virheistä.
- Mittaaja (tracker) seuraa projektin edistymistä ja annettujen aikatauluarvioiden toteutumista. Mikäli toteutuneet aikataulut poikkeavat arvioista, hän ohjaa ryhmää tarkastamaan arvioitaan oikeaan suuntaan. Mittaajan rooli on perinteisissä projektiorganisaatiossa usein kuulunut projektipäällikölle.
- Valmentaja (coach) on Extreme Programming -metodologian asiantuntija ja opastaa ryhmää XP:n soveltamisessa käytäntöön. Hän myös valvoo, että XP:n arvoja ja toimintatapoja noudatetaan. Myös tämä rooli on perinteisessä linjaorganisaatiossa usein kuulunut projektipäällikölle.
- Konsultti on liiketoiminnallisen tai teknisen alueen erikoisasiantuntija, joka auttaa ryhmää alueeseen liittyvissä erityiskysymyksissä. Konsultteja voi olla useita ja heidän roolinsa projektissa on lähinnä vieraileva.
- Johtajan (big boss) vastuulla on päätöksenteko. Hän seuraa projektia ja poistaa edistymisen tiellä olevia esteitä.

Edellä olevan luettelon perustella voidaan havaita, että XP ei sinänsä vaadi organisaatiomuutoksia verrattuna perinteiseen ohjelmistotuotannon linjaorganisaatioon. Vanhat ohjelmoijan, testaajan, projektipäällikön ja johtajan toimenkuvat pysyvät melko muuttumattomina, tosin vastuuta työmäärien arvioinnista ja asiakkaan kanssa kommunikaatiosta annetaan projektipäällikön lisäksi myös ohjelmoijille. Projektipäällikkö ottaa yleensä sekä mittaajan että valmentajan roolit. Projektipäälliköllä tulisi olla hyvä ymmärrys XP:stä, jotta projekti voisi onnistua.

3. Muita joustavia projektinhallintamenetelmiä

3.1. Menetelmille yhteisiä piirteitä

Extreme Programmingin rinnalle on syntynyt useita sen kaltaisia projektinhallintamenetelmiä, jotka ovat saaneet ehkä hieman vähemmän julkisuutta. Kaikki nämä menetelmät perustuvat lean developmentin periaatteisiin. Niiden fokus on projektissa toimivissa ihmisissä, projektin tuloksen mittaamisessa toimivan ohjelmiston kautta, mahdollisimman pienessä metodologisessa byrokratiassa ja mahdollisimman hyvässä kommunikaatiossa [Highsmith, 2000]. Menetelmien käytännön työn yhteisiä piirteitä ovat kehitysprosessin iteratiivisuus, ohjelmiston inkrementtien toimitusvälin pitäminen lyhyenä ja sopeutuminen erilaisiin ohjelmistoprojektissa tapahtuviin muutoksiin [Abrahamsson *et al*, 2002].

Menetelmät eivät kilpaile keskenään, vaan ne toimivat läheisessä yhteistyössä. Yhteistyö näkyy parhaiten Agile Alliance -nimisen yhdistyksen [Agile Alliance] muodossa, joka perustettiin marraskuussa 2001. Menetelmiä kutsuttiin aluksi kevyiksi (lightweight) menetelmiksi niiden vähäisen byrokratian vuoksi, mutta nykyään yhdistyksen aktiivit puhuvat mieluummin joustavista (agile) kuin kevyistä menetelmistä. Joustavuudella viitataan kykyyn sopeutua projektien nopeasti muuttuviin vaatimuksiin. Yhdistyksen verkkosivuilla on laaja ja hyvin jäsennelty kokoelma joustavia menetelmiä käsitteleviä artikkeleita. Yhdistys pitää myös vuotuisia konferensseja aiheesta.

Agile Alliance on julkaissut näkemyksensä ohjelmistokehityksen periaatteista manifestissa, jota se kutsuu nimellä Agile Manifesto [Agile Manifesto]. Tässä manifestissa he kertovat etsivänsä parempia tapoja tehdä ohjelmistoja niitä tekemällä ja muita siinä auttamalla. Manifestissa kootaan myös metodologioiden yhteisiä arvoja:

- Yksilöitä ja yhteistyötä arvostetaan enemmän kuin prosesseja ja työkaluja.
- Toimivaa ohjelmistoa arvostetaan enemmän kuin kattavaa dokumentaatiota.
- Yhteistyötä asiakkaan kanssa arvostetaan enemmän kuin yksityiskohtaisia sopimusneuvotteluja.
- Muutoskykyä arvostetaan enemmän kuin suunnitelman noudattamista.

Agile Alliance esittää verkkosivuillaan, että projektin joustavuus tulisi nostaa viidenneksi projektiin vaikuttavaksi muuttujaksi kohdassa 1.1 esitettyjen neljän muuttujan lisäksi. Heidän mielestään joustavuus, eli

mahdollisuus muuttaa ohjelmiston määrittelyjä ja toimintojen priorisointia ilman suuria kustannusvaikutuksia, on asiakkaalle strateginen kilpailuetu.

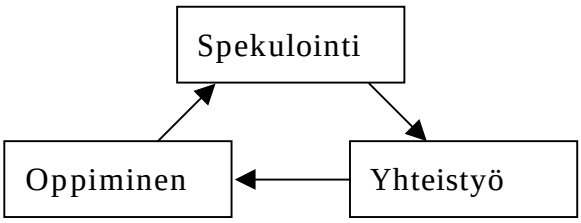
Seuraavaksi esitellään tärkeimmät joustavat menetelmät Adaptive Software Development eli ASD, Crystal-metodologiaperhe, Dynamic Systems Development Method eli DSDM, Feature-Driven Development eli FDD ja Scrum. Luku perustuu osin VTT:llä julkaistuun tutkimukseen Agile [Abrahamsson *et al*, 2002], jossa on esitelty ja vertailtu joustavia metodologioita.

Myös Open Source -kehitysmalli on monissa julkaisuissa rinnastettu joustaviin menetelmiin, mutta sillä ei lähemmin tarkasteltuna ole paljoakaan yhteistä varsinaisten joustavien menetelmien kanssa. Myös sen syntyhistoria on erilainen. Rinnastus johtuu enemmänkin siitä, että se tuli uutena ja radikaalina menetelmänä yleiseen tietoisuuteen samanaikaisesti joustavien menetelmien kanssa. Sama tilanne on aiemmin esitellyn Rational Unified Processin kanssa. Menetelmä on iteratiivinen, mutta se ei muuten pohjaudu joustavien menetelmien yleisiin periaatteisiin. Alla olevat menetelmät onkin valittu tähän tutkimukseen sillä perusteella, että niillä on selkeä asema Agile Alliance -yhteisössä.

3.2. Adaptive Software Development

Jim Highsmithin kehittämä Adaptive Software Development (ASD) perustuu Brian Arthurin [Arthur, 1996] esittämään näkemykseen siitä, että ohjelmistoprojektien toimintaympäristö on hyvin ennalta arvaamaton ja jatkuvasti muuttuva. Niinpä ASD:ssä hylätään ennustettavuuteen pyrkivä vesiputousmalli ja ohjelmisto rakennetaan evoluutiomallin tapaisella tavalla lyhyissä toimitussykleissä.

Tässä syklissä perinteinen määrittely-suunnittelu-toteutus –malli on korvattu syklillä, jossa vuorottelevat vaiheet spekulointi, yhteistyö ja oppiminen (speculation-colloboration-learning). Spekulointi on nimetty korvaamaan määrittely, koska muutosalttiissa projekteissa ei voida etukäteen tietää lopullisen tuotteen tarkkoja yksityiskohtia, vaan spekuloimalla pyritään pääsemään yksimielisyyteen suurista suuntaviivoista. Ohjelmiston rakentaminen on korvattu sanalla yhteistyö koska ihmisten välinen yhteistyö nähdään kriittisenä menestystekijänä. Projektin laaduntarkkailu, sisältäen katselmoinnit ja loppupalaverit, on korvattu sanalla oppiminen. ASD:n vaiheet on esitetty kuvassa 8.



Kuva 8: Adaptive Software Developmentin malli.

Spekulointivaiheessa määritellään ensin yleisellä tasolla mitä ollaan tekemässä ja asetetaan sen mukainen projektin valmistumispäivä. Tämän jälkeen harkitaan, kuinka monta iteraatiokierrosta projektissa tarvitaan ja asetetaan jokaiselle iteraatiolle valmistumispäivät. Valmistumispäivät ovat stabiilit, mutta niitä ei saa käyttää väärin ylitöiden tai heikentyneen laadun muodossa. Jos projekti on vaarassa jäädä jälkeen aikataulustaan, on tehtävä päätös siitä, mitä ominaisuuksia kyseisestä iteraatiosta jätetään pois. Iteraatioiden valinnan jälkeen määritellään, mitkä tehtävät tehdään missäkin iteraatiossa tehtävien tärkeysjärjestyksessä. Tämä tehdään sillä periaatteella, että jokainen iteraatio tuo asiakkaalle uusia ja hyödyllisiä ominaisuuksia. Näin saadaan projektin tehtävälista valmiiksi, jolloin spekulointivaihe on suoritettu.

Yhteistyövaiheessa ASD eroaa Extreme Programmingista antamalla paljon enemmän vapauksia varsinaisen toteutusvaiheen suorittamiseen. Käytännössä projektiryhmät ovat itseohjautuvia tiimejä. Projekteissa, joissa tiimit ovat pieniä ja työskentelevät samassa paikassa, suositellaan käytettäväksi XP:n menetelmiä. Adaptive Software Development sopii myös suurempien ja maantieteellisesti hajallaan olevien tiimien käyttöön, tällöin tosin tarvitaan kurinalaisempia tiedotus- ja yhteistyökäytäntöjä. Varsinainen työ annetaan tiimille kuitenkin tavoitteiden muodossa, jonka jälkeen tiimi toimii itseohjautuvasti niiden saavuttamiseksi.

Kun tiimi katsoo saavuttaneensa tavoitteet, seuraa oppimisvaihe. Tässä kiinnitetään huomiota neljään seikkaan:

- Projektin tulosten laatu asiakkaan näkökulmasta. Saiko asiakas todella jotain uutta ja hyödyllistä tällä iteraatiokierroksella? Tämä tieto saadaan asiakkaalta.
- Projektin tulosten laatu teknisestä näkökulmasta. Onko tekniikka paras mahdollinen? Tämä tieto saadaan perinteisin ohjelmistojen laaduntarkkailun menetelmin, eli katselmoimalla ja testaamalla.

- Projektiryhmän sisäisen toiminnan taso. Voisiko toimintatapoja tai työmenetelmiä eli itse prosessin laatua parantaa jollain tavalla? Vastaus tähän saadaan projektiryhmältä itseltään.
- Projektin aikataulujen pitävyys. Ollaanko aikatauluista edellä tai jäljessä? Tieto saadaan kaikkien osallistujien yhteistyönä ja se saattaa johtaa muutoksiin seuraavien iteraatioiden toteutuksessa.

Oppimisvaiheen jälkeen alkaa seuraava iteraatiokierros, kunnes saavutetaan projektin valmistumispäivä. Projektin valmistumisesta valmistumispäivänä pidetään tiukemmin kiinni kuin ohjelmiston täydellisestä laajuudesta, ellei asiakas nimenomaan päätä toisin.

3.3. Crystal-metodologiaperhe

Crystal koostuu useista eri metodologioista, joista valitaan tarkoituksenmukaisin kuhunkin projektiin. Menetelmillä on kuitenkin yhteinen perusta ja niihin lisätään kontrollia sitä mukaa, mitä suurempi ja haastavampi projekti on kyseessä. Menetelmien kehittäjinä ovat toimineet Alistair Cockburn ja Jim Highsmith.

Projekteja arvioidaan kahden muuttujan mukaan. Ensimmäinen niistä on projektiryhmän koko, joka määrittelee kommunikaatiotarpeet. Toinen muuttuja on projektin kriittisyys. Eri kriittisyysasteita on neljä kappaletta nimettynä sen mukaan, mihin menetykset kohdistuisivat jos toteutettava ohjelmisto menettäisi toimintakykynsä [Cockburn 2002a]:

- Mukavuus (comfort), josta käytetään lyhennettä C. Tähän kuuluvat tilanteet, joissa ohjelmistovirheen seurauksena tehtävät joudutaan tekemään käsin tietokoneen sijasta, joka vie enemmän aikaa.
- Kohtuullinen raha (discretionary money), josta käytetään lyhennettä D. Tähän kuuluvat tilanteet, joissa rahallinen menetys on myöhemmin korvattavissa, esimerkiksi palkanmaksuohjelmisto laskee työntekijälle liian pienen palkan.
- Kriittinen raha (essential money), josta käytetään lyhennettä E. Tähän kuuluvat tilanteet, joissa taloudelliset menetykset ovat yksilön tai koko organisaation kannalta merkittäviä ja saattavat johtaa vararikkoon tai konkurssiin.
- Elämä (life), josta käytetään lyhennettä L. Tähän kuuluvat tilanteet, joissa ohjelmistovirhe saattaa vaarantaa useiden ihmisten hengen, esimerkiksi ydinvoimaloiden ohjelmistoissa.

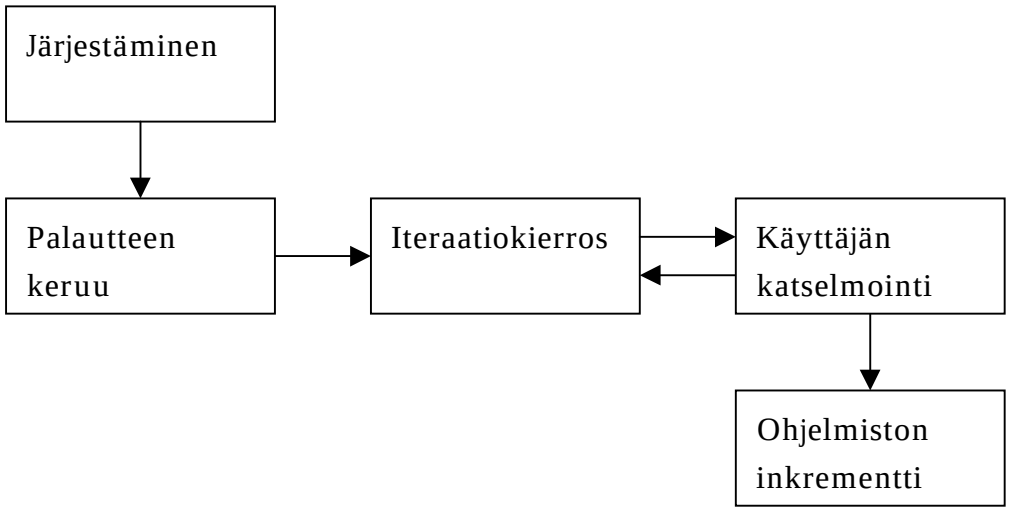
Yhdistämällä projektiryhmän koko ja sen kriittisyys saadaan projektin haastavuutta kuvaava tunnus. Esimerkiksi D6 tarkoittaa projektia, jossa

työskentelee kuusi henkilöä ja sen tuottaman ohjelmiston virhe voi merkitä pahimmillaan kohtuullista rahan menetystä. Eri metodologiat nimetään eri väreillä siten, että kaikkein vähiten henkilöitä sisältävä metodologia on kirkas (clear), seuraavaksi keltainen (yellow), oranssi (orange) ja kaikkein suurimpiin projekteihin punainen (red). Näitä tarkennetaan edelleen projektin kriittisyyden mukaan. Crystal-metodologiaperhe ei kuitenkaan vielä tällä hetkellä ole täysin valmis, sillä ohjeet ovat valmiit vain kirkas- ja oranssi-tasoisiin projekteihin. Seuraavassa taulukossa on koottu eri metodologiat siten, että pystysarakkeilla on merkitty projektiryhmän koko ja vaakariveillä projektin kriittisyys. Viimeisellä rivillä on kyseisen sarakkeen kuvaamiin projektimalleihin soveltuva metodologia. Valmiit metodologiat on kuvattu harmaalla pohjalla, keskeneräiset valkoisella.

Taulukko 1: Crystal-metodologiaperhe

	Enintään 6 henkilöä	Enintään 20 henkilöä	Enintään 40 henkilöä	Enintään 80 henkilöä
Mukavuus	C6	C20	C40	C80
Kohtuullinen raha	D6	D20	D40	D80
Kriittinen raha	E6	E20	E40	E80
Elämä	L6	L20	L40	L80
Käytettävä menetelmä	Kirkas	Keltainen	Oranssi	Punainen

Varsinainen tuotantosykli on sekä kirkkaassa että oranssissa vaihtoehdossa iteratiivinen. Kirkkaassa iteraatioiden väli on yhdestä kolmeen kuukautta, mutta oranssissa sitä voidaan venyttää neljään. Iteraation ensimmäistä osaa kutsutaan nimellä järjestäminen (staging), jossa XP:n suunnittelupelin tapaan valitaan seuraavaan iteraatioon tulevat toiminnot ja arvioidaan niiden suorittamiseen kuluva aika. Jokaista ohjelmiston inkrementtiä kohden tehdään kirkkaassa vaihtoehdossa yksi iteraatio, kun taas oranssissa tehdään useita iteraatioita, mikä erottaa sen muista iteratiivisista menetelmistä. Jokaisen iteraation päätteeksi käyttäjä katselmoi tuloksia ja antaa niistä palautetta. Inkrementtien välissä pidetään palautteen keruutilaisuus, jossa arvioidaan nykyistä prosessia ja mietitään miten sitä voisi entisestään kehittää.



Kuva 9: Crystal-metodologiaperheen kehitysmalli.

Ohjelmistoa mitataan edistymisen ja vakauden suhteen. Edistymistä mitataan niin sanotuilla merkkipaaluilla (milestones), joita voivat olla esimerkiksi käyttötapauksen koodin valmistuminen ja sen testauksen valmistuminen. Vakausaste kertoo, kuinka luotettavasti ohjelmisto toimii. Kun vakausaste ”riittävän vakaa katselmoitavaksi” on saavutettu, voidaan ohjelmisto esitellä asiakkaalle.

Koska oranssi vaihtoehto tukee jopa 40 hengen projekteja, ne täytyy jakaa aliprojekteihin. Aliprojektien ryhmien muodostamiseen tarjotaan holistista strategiaa (holistic diversity strategy). Sen perusideana on jakaa henkilöstö aliprojekteihin siten, että jokaiseen aliprojektiin tulee useiden eri erikoisalueiden osaajia. Holistinen strategia tarjoaa myös ohjeita aliprojektien välisen kommunikaation ja koordinaation hoitamiseen.

Toisin kuten Extreme Programming, Crystal-metodologiat eivät tarjoa ohjeita siitä, miten itse sovelluskehitystyö pitäisi tehdä, vaan jättää nämä asiat projektiryhmän harkintaan. Tällaisia asioita nimitetään paikallisiksi (local matters), joilla tarkoitetaan asioita, joihin metodologia ei ota kantaa, mutta jotka on määriteltävä ennen projektin aloittamista.

3.4. Dynamic Systems Development Method

Dynamic Systems Development Method eli DSDM syntyi 1990-luvun alkupuolella. Vuonna 1994 16 organisaatiota perusti avoimen ja voittoa tavoittelemattoman DSDM Consortiumin, joka ylläpitää metodologiaa ja siihen liittyvää verkkosivustoa. Metodologian perusideana on, että ohjelmistoprojektin neljästä muuttujasta ainoastaan ohjelmiston laajuudesta joustetaan, kun taas sovittuihin taloudellisiin ja aikaresursseihin ei saa tulla

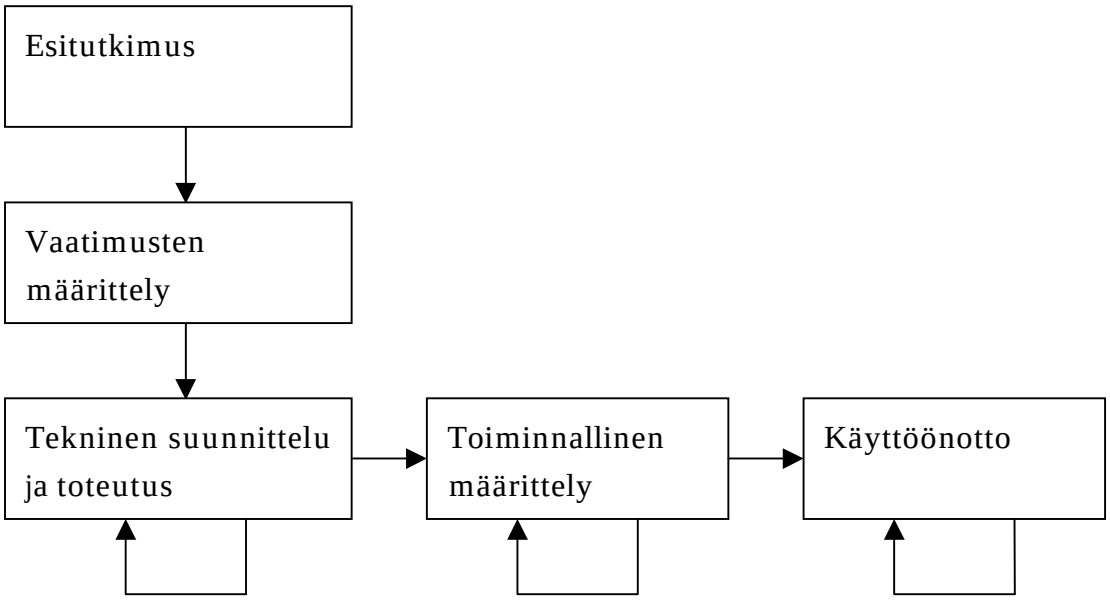
muutoksia. Iteraatioiden aikataulut ovat Adaptive Software Developmentin tavoin tiukasti kiinnitetyt (time-boxed). Mikäli kaikkien iteraatioon tarkoitettujen tehtävien tekeminen määräaikaan mennessä osoittautuu mahdottomaksi, DSDM neuvoo tinkimään toimintojen määrästä, ei määräajasta.

DSDM-prosessissa on viisi vaihetta. Näistä vaiheista kaksi ensimmäistä, esitutkimus (feasibility study) ja vaatimusten määrittely (business study), ovat peräkkäisiä ja tapahtuvat vain kerran. Kolme seuraavaa, toimintojen määrittelyvaihe, tekninen suunnittelu- ja toteutusvaihe sekä käyttöönottovaihe, ovat iteratiivisia. Esitutkimusvaihe on perinteisten menetelmien esitutkimusvaiheen mukainen; siinä syntyvät dokumentit esitutkimusraportti ja karkean tason projektisuunnitelma. Vaiheessa on tarkoitus myös tarkistaa, että DSDM on soveltuvin menetelmä projektin hallintaan. Myös vaatimusten määrittelyvaihe noudattelee perinteisiä menetelmiä sillä poikkeuksella, että vaatimusmäärittelyvaiheessa syntyvät myös karkean tason arkkitehtuurisuunnitelma ja prototyyppien kehittämissuunnitelma seuraavia vaiheita varten.

Toimintojen määrittelyvaiheessa tehdään varsinaisen toimintojen määrittelyn lisäksi prototyyppejä määriteltyjen toimintojen pohjalta. Tässä vaiheessa määritellään myös ei-toiminnalliset vaatimukset, jotka kirjataan omaan dokumenttiinsa. Vaiheen jokainen iteraatiokierros tuottaa uuden prototyypin, joka esitellään asiakkaalle ja josta kirjoitetaan katselmointiraportti seuraavia iteraatioita varten. Muut iteraation tuotteet ovat perinteinen toiminnallinen määrittely kaavioineen sekä kehittämistyön riskianalyysi. Testaus alkaa DSDM:ssä jo tässä vaiheessa ja tehdään jokaiselle prototyypille. Perinteisestä prototyyppien kehittämisestä poiketen prototyyppejä ei hylätä, vaan niiden tarkoitus on olla testattuina tarpeeksi hyviä toimiakseen pohjana varsinaiselle järjestelmälle.

Teknisen suunnittelun ja toteutuksen vaiheessa suunnitellaan varsinaiset ohjelmistokomponentit ja testataan ne. Vaiheessa rakennetaan prototyyppien perusteella lopullista järjestelmää ja jokaisen iteraation lopussa järjestelmä jälleen esitellään asiakkaalle kommentteja varten. Viimeisessä eli käyttöönottovaiheessa valmis järjestelmä asennetaan asiakkaan käytettäväksi. Siinä myös kirjoitetaan tarvittava dokumentaatio sekä annetaan käyttäjille tarvittava koulutus. Käyttöönottovaiheessa kirjoitetaan käyttöohjeen lisäksi myös projektin loppuraportti, jossa tarkastellaan projektin kulkua ja sitä, onko joitain ominaisuuksia jouduttu jättämään pois tai ilmaantunut projektin aikana lisää. Näiden toteuttamiseksi DSDM tarjoaa eri vaihtoehtoja riippuen siitä,

kuinka kriittisiä ja minkä tyyppisiä ominaisuudet ovat. Dynamic Systems Development Modelin vaiheet on esitetty kuvassa 10.



Kuva 10: Dynamic Systems Development Modelin vaiheet.

DSDM ei anna tarkkoja ohjeita varsinaisen ohjelmointityön tekemisestä, vaan sen painopiste on ohjauksessa ja tiimityössä. DSDM määrittelee yhdeksän työtapaa, joista suurin osa on yhteisiä joustavien menetelmien yleisten periaatteiden kanssa. Omina erityispiirteinä nousevat muita vastaavia menetelmiä suurempi dokumentointi ja perusvaatimusten jäädyttäminen korkealla tasolla. Nämä tekevätkin DSDM:stä hieman muita vastaavia menetelmiä vähemmän joustavan.

3.5. Feature-Driven Development

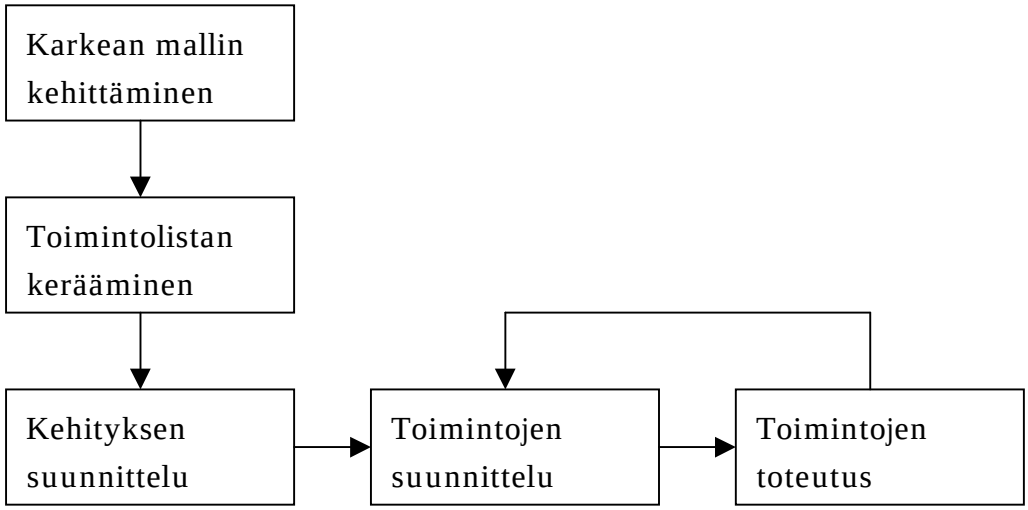
Feature-Driven Development (FDD) ei varsinaisesti tarjoa metodologiaa kaikkien ohjelmistoprojektin vaiheiden hallintaan, vaan se keskittyy ohjelmiston teknisen suunnittelun, toteutuksen ja testauksen suorittamiseen. Se on kuitenkin tunnustettu osa Agile Alliance -yhteisöä, joten sen esitteleminen tässä on perusteltua. FDD on suunniteltu integroitumaan osaksi toista ohjelmistokehitysmetodologiaa [Palmer ja Felsing, 2002]. FDD on melko uusi menetelmä jopa muihin joustaviin menetelmiin verrattuna, sillä se esiteltiin vasta vuonna 2000 [Coad *et al*, 2000].

Feature-Driven Development jakautuu viiteen eri vaiheeseen, joista kaksi viimeistä ovat iteratiivisia. Vaiheet ovat karkean mallin kehittäminen, toimintolistan kerääminen, kehityksen suunnittelu, toimintojen suunnittelu ja toimintojen toteutus. Karkean mallin kehittämisen alussa ohjelmiston vaatimukset on jo selvitetty jonkin laajemman projektinhallinnan menetelmän

avulla. Vaiheen aluksi suunnittelijoille esitellään vaatimukset yleisellä tasolla. Tämän jälkeen suunnittelijat jakautuvat pienempiin ryhmiin ja heille esitellään ryhmäkohtaisesti heitä koskevat vaatimukset tarkemmalla tasolla. Tämän perusteella toimintoja ja kohdealuetta mallinnetaan sekä karkean tason luokkakaaviolla että sekvenssikaavioilla.

Toisessa vaiheessa kerätään toimintolista, johon ryhmitellään sovellukseen halutut toiminnot. Toimintolista katselmoidaan asiakkaan kanssa. Näin voidaan varmistua, että lista kattaa kaikki asiakkaan tarpeet, mutta turhia toimintoja ei ole mukana. Toimintolistaa voidaan päivittää koko projektin ajan lisäten, muuttaen ja poistaen toimintoja, joka tuo menetelmään joustavuutta. Kolmannessa eli kehityksen suunnitteluvaiheessa vaiheessa karkean luokkakaavion luokat jaetaan toteuttajille kehitettäviksi ja ylläpidettäviksi sekä suunnitellaan toimintoryhmille toteutusaikataulut.

Toimintojen suunnittelu- ja toteutusvaiheita toistetaan iteratiivisesti. Suunnitteluvaiheen aluksi valitaan toiminnoista ne, jotka tässä iteraatiossa toteutetaan ja muodostetaan suunnittelijoista niiden toteutukseen sopivat ryhmät. Samaan aikaan voi toimia useita rinnakkaisia ryhmiä, jotka kehittävät eri toimintoja. Suunnittelu on perinteistä ohjelmiston teknistä suunnittelua, josta siirrytään iteraation sisällä ohjelmointivaiheeseen. Ohjelmointiin kuuluu paitsi itse ohjelmakoodin kirjoittaminen, myös moduulitestaus, koodikatselmoinnit sekä iteraation lopuksi valmistuneen koodin integrointi aikaisemmissa iteraatioissa syntyneeseen koodiin. Tämän jälkeen valitaan seuraavan iteraation toiminnot ja tehdään niistä seuraava inkrementti. Koko prosessi päättyy, kun kaikki toimintolistan toiminnot ovat valmistuneet. Feature-Driven Developmentin vaiheet on esitetty kuvassa 11.



Kuva 11: Feature-Driven Developmentin vaiheet.

Alkuperäisen kirjan jälkeen menetelmää on kehitetty edelleen. Feature-Driven Developmentista on kaksi versiota, tässä on esitelty uudempi versio. Eri versiot eivät poikkea merkittävästi toisistaan. Muutoksia on tullut vaiheiden yksityiskohtiin, esimerkiksi toimintojen priorisointi on jäänyt pois, mutta pääsääntönä oleva viiden vaiheen jaottelu on pysynyt ennallaan.

3.6. Scrum

Urheilutermi “scrum” tulee alunperin rugbystä, jossa se tarkoittaa pallon tuomista yhteistyöllä takaisin peliin [Schwaber ja Beedle, 2002]. Se kehittyi kahden yrityksen, Advanced Development Methodsin ja VMARK Softwaren yhteistyönä [Schwaber, 1996]. Yritykset eivät olleet tyytyväisiä kehitysprosesseihinsa ja halusivat uudistaa niitä. Samalla ne halusivat tuoda kehitystyönsä tulokset koko ohjelmistoalan kehittymisen pohjaksi. Scrum näkee ohjelmistoprojektien toimintaympäristön samanlaisena kuin Adaptive Software Development, eli siinä on useita kontrolloimattomia muuttujia, joihin projektin tulee sopeutua, jotta asiakkaalle saadaan todellista hyötyä. Scrum ei verkkosivustonsa [Scrum] mukaan ole metodologia, vaan kehys tai runko (framework) ohjelmistojen toteutustyöhön liittyvän epävarmuuden kontrollointiin. Näin ollen tässäkin yhteydessä puhutaan kehyksestä. Scrum ei määrittele tarkasti sitä, miten vaatimusten keruu, suunnittelu tai ohjelmointi tulisi tapahtua, vaan antaa ohjeet vain projektien organisoinnista yleisellä tasolla Scrum-prosessia noudattamalla. Tässä suhteessa se eroaa merkittävästi XP:stä.

Scrum-prosessi jakautuu kolmeen vaiheeseen: alustavaan vaiheeseen (pre-game), tuotantovaiheeseen (development phase tai game phase) ja jälkivaiheeseen (post-game). Alustava vaihe jakautuu kahteen alivaiheeseen, suunnittelu- ja arkkitehtuurivaiheisiin. Suunnitteluvaiheessa tehdään kaikkien projektin osapuolten yhteistyönä tekemättömien töiden lista (product backlog list), joka on eräänlainen vaatimusmäärittelyn ja projektisuunnitelman yhdistelmä. Siihen kerätään ja priorisoidaan vaatimusmäärittelyn tapaan kaikki suunnitteluvaiheessa tiedossa olevat vaatimukset. Projektisuunnitelman tapaan vaatimuksille annetaan aikatauluarviot. Lisäksi listaan kirjataan projektihenkilöstö, käytettävät työkalut, projektin riskit ja niiden hallinta sekä muita projektisuunnitelmalle tyypillisiä asioita. Listaa päivitetään jatkuvasti heti, kun uutta tai tarkennettua tietoa vaatimuksista, aikatauluarvioista tai muista asioista on saatavilla. Arkkitehtuurivaiheessa suunnitellaan nimensä mukaisesti tuotettavan järjestelmän arkkitehtuuri tiedossa olevien vaatimusten mukaan. Vaiheeseen liittyy myös arkkitehtuurin katselmointi (review meeting) sekä vaatimusten alustava jakaminen eri iteraatiokierrosten tuloksiin. Tyypillinen iteraatiokierrosten määrä on kolmesta kahdeksaan.

Tuotantovaihetta kutsutaan Scrumissa jälleen urheilutermein kiriksi (sprint). Tuotantovaiheessa tapahtuvat kaikki Scrumin iteraatiokierrokset. Iteraatiossa toistuvat perinteiset ohjelmistoprojektin vaiheet: vaatimusten keruu (Scrumin tapauksessa vaatimusten tarkennus), tekninen suunnittelu, toteutus, testaus ja ohjelmiston toimitus. Iteraation alussa tekemättömien töiden lista päivitetään uusien toimintojen ja priorisointien suhteen ja valitaan siitä iteraatiossa tehtävät työt, jotka muodostavat iteraation oman tekemättömien töiden listan (sprint backlog). Tämän listan tehtäviä ei saa enää muuttaa vaikka vaatimukset muuttuisivatkin, vaan muuttuneet asiat otetaan huomioon vasta seuraavissa iteraatiossa [Rising ja Janoff, 2000]. ASD:n ja DSDM:n tapaan myöskään iteraation aikataulussa ei jousteta, vaan sille on asetettu kiinteä määräaika, jonka ylittymisen uhatessa tulee tinkiä toimintojen määrästä.

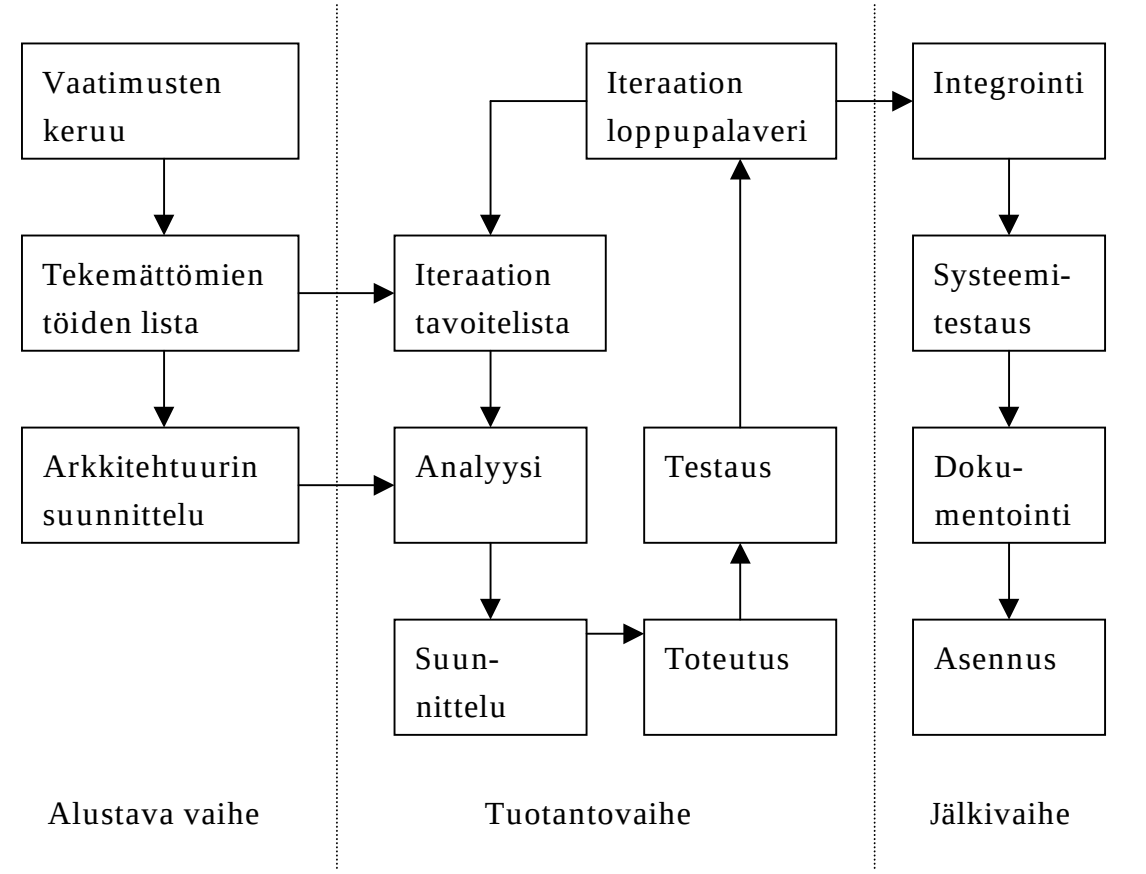
Tuotantovaiheessa esiintyy Scrum-kehykselle ominainen piirre: päivittäiset palaverit projektiryhmän kesken. Palaverien tiheydellä pyritään varmistamaan, ettei kukaan projektiryhmästä jää pidemmäksi aikaa ilman jotain tärkeää tietoa tai juutu johonkin kehitystyössä vastaan tulevan ongelmaan. Koska palavereja on päivittäin, niiden pituudeksi on määrätty maksimissaan 15 minuuttia. Palaverit ovat hyvin määrämuotoisia projektipäällikön johtamia tilaisuuksia, joissa ei ole tarkoitus ratkaista teknisiä ongelmia, vaan ainoastaan seurata työn etenemistä. Palaverissa jokaiselta ohjelmoijalta kysytään vuorotellen seuraavat kolme kysymystä:

1. Mitä toimintoja iteraation tekemättömien töiden listasta olet saanut valmiiksi edellisen palaverin jälkeen?
2. Onko työsi etenemisen esteenä mitään ongelmia?
3. Mitä toimintoja iteraation tekemättömien töiden listasta aiot saada valmiiksi seuraavaan palaveriin mennessä?

Näiden kysymysten avulla pyritään varmistamaan, että ryhmän fokus pysyy iteraation tavoitteissa, etenemisen esteet huomataan nopeasti ja kaikki projektin sidosryhmät ovat jatkuvasti tietoisia projektin tilasta.

Kun tuotantovaiheen iteraatio on päättynyt, pidetään kaikkien projektin sidosryhmien kesken iteraation päätöspalaveri. Tässä palaverissa tarkastetaan iteraation tulokset ja päivitetään tekemättömien töiden lista. Tässä vaiheessa on jälleen mahdollista lisätä, poistaa, muuttaa ja uudelleen priorisoida listan tehtäviä. Projektin loppupuolella päätetään, onko ohjelmisto jo valmis vai vieläkö uusia iteraatiokierroksia tarvitaan. Kun uusien toimintojen tekemistä ei enää nähdä tarpeelliseksi tai kustannustehokkaaksi, siirrytään Scrum-kehyksen jälkivaiheeseen. Jälkivaiheessa tehdään ohjelmiston eri osien integrointi,

systeemitestaus, asennus asiakkaalle ja dokumentointi. Scrumissa voidaan jokaisen iteraation jälkeen joko toimittaa ohjelmiston nykyinen versio asiakkaalle tai vain esitellä toiminnallisuutta ilman varsinaista toimitusta. Tässä suhteessa se eroaa muista joustavista menetelmistä, jotka perustuvat uuden ohjelmiston version toimittamiseen jokaisen iteraation jälkeen. Scrum-kehyksen vaiheet on esitetty kuvassa 12.



Kuva 12: Scrum-kehyksen vaiheet.

4. Vertailua muihin projektinhallinnan menetelmiin

4.1. Menetelmien vertailuperusteita

Erilaisten metodologioiden vertailu on vaikea tehtävä ja johtaa usein johtopäätösten tekoon vertailijasta riippuvien subjektiivisten seikkojen ja intuition perusteella [Song ja Osterweil, 1991]. Tällaisten epäformaalien menetelmien rinnalle on kehitetty kvasiformaaleja menetelmiä [Song ja Osterweil, 1992], joissa on otettu peruslähtökohtia vertailun suorittamiseen. Tällaisia kvasiformaaleja menetelmiä ovat [Sol, 1983]:

1. Ideaalimetodologian kuvaus ja muiden metodologioiden vertailu sitä vasten.
2. Metodologioiden tärkeiden ominaisuuksien kokoaminen yhteen ja metodologioiden vertailu tätä ominaisuuksien joukkoa vasten.
3. A priori -hypoteesin muodostaminen metodologian toiminnasta ja sen vertailu eri metodologioista saatuihin empiirisiin tutkimustuloksiin.
4. Metakielen muodostaminen vertailun yhteiseksi pohjaksi ja referenssiksi, jota vasten metodologioita verrataan.
5. Jonkin tietyn ongelman esittely ja eri metodologioiden tarjoamien ratkaisumahdollisuuksien vertailu.

Koska tarkoituksena tässä tutkimuksessa on nimenomaan vertailla muita ohjelmistotuotannon menetelmiä Extreme Programmingiin, sovelletaan tässä menetelmää 1 ottamalla Extreme Programming ideaalimetodologiaksi ja arvioimalla muiden menetelmien käytäntöjä sitä vasten. Tämä ei kuitenkaan tarkoita, että Extreme Programming nähtäisiin sinänsä ideaalisena ohjelmistotuotannon menetelmänä.

4.2. Vertailua perinteisiin menetelmiin

Edellisten lukujen perusteella voidaan jo arvata, että XP:n erot perinteisiin menetelmiin verrattuna ovat melko suuria. Seuraavassa on tuotu esiin olennaisimmat eroavaisuudet.

- Vaihejakomalli poikkeaa eniten XP:n ja vesiputousmallin välillä. Vesiputousmallissa tehdään jokainen projektin vaihe vain kerran, sen sijaan XP:ssä pyritään mahdollisimman pieniin iteraatioihin eli useisiin vaiheiden toistokertoihin. Asteittaisen kehityksen malli on osittain iteratiivinen, mutta iteraatiokierroksia on huomattavasti vähemmän. Vähemmän niitä on myös täysin iteratiivisessa spiraalimallissa. Evolutionaarinen kehitys toistaa myös iteraatioita, mutta niiden

lukumäärä ei ole etukäteen selvillä. Kuitenkin kaikki mallit ovat pitkällä aikajaksolla tarkasteltuna lineaarisia: ne alkavat esitutkimuksella ja päättyvät ohjelmiston viimeiseen käyttöönottoon.

- Asiakkaalle tärkeä kustannusten ennustettavuus on vesiputousmallissa paras: kustannukset voidaan laskea varsin tarkasti jos oletetaan, ettei kustannuksia kasvattavia muutoksia tule kovin runsaasti. Asteittaisen kehityksen malli tarjoaa myös lähes yhtä hyvän ennustettavuuden. Myös iteratiivinen malli tarjoaa hyvän pohjan kustannusten laskentaan, sillä siinä voidaan laskea tarkasti ainakin jokaisen iteraatiokierroksen kustannus. Sen sijaan projektit, joiden hallintaan valitaan evolutionaarinen kehitys tai XP ovat luonteeltaan ennalta arvaamattomia, mistä myös seuraa, ettei niiden kustannusten ennustettavuus ole yhtä hyvä.
- Ohjelmiston arkkitehtuuri rakennetaan vesiputousmallissa jo sen suunnitteluvaiheessa. Samoin asteittaisen kehittämisen mallissa arkkitehtuuri suunnitellaan projektin aluksi. Myös spiraalimallissa ja RUP:ssä tämä on pyrkimyksenä, vaikka tulevat iteraatiokierrokset saattavatkin tuoda siihen muutoksia. Sen sijaan evolutionaarisessa kehityksessä arkkitehtuuri muotoutuu ohjelmiston evoluution myötä. XP:ssä arkkitehtuurisuunnittelua ei tehdä omana vaiheenaan lainkaan, vaan se on korvattu refactoring-menetelmän käytöllä.
- Muutostenhallinnan käytännöt ovat byrokraattisimmat vesiputousmallissa. Siinä muutokset täytyy hyväksyttää projektin ohjausryhmällä, jonka jälkeen ne vasta voidaan toteuttaa. Muutosten toteuttaminen vaatii usein paluuta aikaisempiin vaiheisiin, erityisesti suunnitteluvaiheeseen. Asteittaisen kehityksen mallissa muutostenhallinta on suurilta osin vesiputousmallin kaltainen, mutta jos muutos voidaan tehdä ilman, että projektissa on palattava itse suunnitteluvaiheeseen, voidaan se tehdä seuraavassa iteraatiossa. XP:ssä, spiraalimallissa ja evolutionaarisessa kehityksessä muutoksia voidaan tehdä jokaisen käyttöönoton jälkeen tapahtuvassa seuraavan version ominaisuuksia kartoittavassa palaverissa, joten niissä muutostenhallinta on helpointa.
- Asiakkaan rooli poikkeaa XP:ssä kaikista perinteisistä menetelmistä. Perinteisesti asiakkaan roolina on antaa määrittelyt ja osallistua projektin ohjausryhmään, iteratiivisissa malleissa myös priorisoida toimintoja iteraatioiden välillä. Tosin vesiputousmallissa priorisointikin on mahdotonta, sillä kaikki toiminnallisuus toimitetaan kerralla. XP:ssä

sen sijaan asiakas on osa projektiryhmää ja aina paikalla tarvittaessa antamassa lisätietoja ja katselmoimassa tuloksia.

- Itse ohjelmointityön suorittamisesta ei moni perinteinen menetelmä anna tarkkoja ohjeita. RUP antaa tarkat ohjeet esimerkiksi siitä, miten ohjelman luokkarakenteita mallinnetaan UML:llä. Samoin Extreme Programming antaa hyvin tarkat ohjeet ohjelmointityön tekemisestä. Tosin RUP:n ja XP:n menetelmiä voidaan käyttää myös muissa malleissa, muut mallit eivät vain määrää niiden käyttöä pakolliseksi.
- Ohjelmiston integrointi tapahtuu vesiputousmallissa vain kerran. Sen sijaan muissa perinteisissä malleissa ohjelmisto integroidaan jokaisen iteraatiokierroksen lopuksi. XP:ssä integrointia harrastetaan eniten ja se on päivittäinen tapahtuma.
- Ohjelmiston testauksessa esiintyvät erot ovat pitkälti integroinnissa esiintyvien erojen tyyppisiä. Vesiputousmallissa ohjelmisto testataan vain kerran, muissa perinteisissä malleissa ohjelmisto testataan jokaisen iteraation lopuksi. XP:ssä ohjelmiston moduulitestaus on päivittäistä ja moduulitestit kirjoitetaan ennen ohjelmakoodia. Lisäksi siinä on jokaisen iteraation lopuksi toiminnallisuuden testaus.
- Vesiputousmallissa koko ohjelmisto toimitetaan kerralla ja aika projektin aloittamisesta ensimmäiseen toimitukseen on näistä malleista pisin. Muissa iteratiivisissa malleissa ohjelmisto toimitetaan useissa osissa. XP:ssä ohjelmisto pyritään toimittamaan mahdollisimman pienissä osissa, joten siinä myös ensimmäinen toimitus asiakkaalle tapahtuu nopeimmin.

Yllä esitetyn perusteella huomataan, että projektimallit voidaan asettaa järjestysasteikolle sen mukaan, kuinka joustavasti muutostenhallinta sujuu, kuinka usein ohjelmistoa testataan ja kuinka usein ohjelmiston inkrementtejä toimitetaan asiakkaalle. Tämä järjestys kulkee vesiputousmallista asteittaisen kehityksen, spiraalimallin ja evolutionaarisen kehityksen kautta XP:hen. Hintana joustavuudesta tulee käytettyjen resurssipanostusten ennustettavuuden heikkeneminen pitkällä tähtäyksellä.

Extreme Programming eroaa merkittävästi erityisesti vesiputousmallista, koska näiden kahden mallin perusoletus ohjelmistoprojektin toimintaympäristöstä poikkeaa eniten. Vesiputousmallissa oletetaan määrittelyjen ja teknologian pysyvän stabiilina, joten muutosprosessi on raskas. Jos tämä oletus pitää paikkansa, vesiputousmalli tuntuu sopivalta. Toisessa ääripäässä Extreme Programming olettaa erittäin epästabiilin toimintaympäristön ja siksi sen muutosprosessi on kevyt. Kevyeen dokumentaatioon ja juuri ennen ohjelmointia tapahtuvaan suunnitteluun ei ole

sitoutunut vesiputousmallin verran pääomaa, joten suuretkaan suunnan vaihdokset projektissa eivät tule yhtä kalliiksi.

Muut perinteiset menetelmät olettavat toimintaympäristön vesiputousmallia labiilimmaksi, mutta XP:tä stabiilimmaksi. Ne hyödyntävät iteraatioiden tuomaa muutoksenhallinnan tehokkuutta, mutta eivät ole muuten yhtä rohkeita menetelmiä kuin Extreme Programming.

Kuitenkin XP:n ohjelmiston rakentamiseen käyttämää 12 menetelmää voidaan hyvin soveltaa myös kaikissa perinteisissä menetelmissä. Soveltamismahdollisuuksia on selvitetty joissain tutkimuksissa, esimerkiksi Cockburnin artikkelisarjassa Learning from Agile Software Development [Cockburn 2002b, 2002c].

4.3. Vertailua muihin joustaviin menetelmiin

Joustavissa menetelmissä toistuu useita yhteisiä piirteitä. Näitä piirteitä on esitetty kohdassa 3.1, joten tässä keskitytään tarkastelemaan menetelmien keskeisiä eroja. Nämä erot voidaan huomata erityisesti seuraavissa piirteissä:

- Aikataulun tärkeyden korostaminen poikkeaa toisistaan eri menetelmissä. ASD, Scrum ja DSDM ovat menetelmiä, joissa iteraation pituus on tiukasti ennalta määritelty. XP ja muut joustavat menetelmät antavat tiukassa tilanteessa asiakkaalle mahdollisuuden päättää, joustetaanko iteraation laajuudesta vai kehitysajasta. Laadusta joustamista sen sijaan mikään menetelmä ei salli.
- Projektiryhmän sallituissa koostumuksissa on eroja. XP sallii vain yhden, hyvin kiinteässä yhteistyössä toimivan ryhmän. Crystal-metodologiaperhe ja Scrum tukevat myös maantieteellisesti hajaantuneita projektiryhmiä. XP myös rajaa projektiryhmän ja sitä kautta toteutettavan projektin koon, eikä se sovellu kovin suuriin projekteihin. Crystal-metodologiaperhe ja DSDM sallivat suuremmat projektiryhmät, tosin DSDM:ssä osaprojektin ryhmäkoko on käytännössä enintään kuusi henkilöä.
- Prototyyppejä ei yleensä joustavissa menetelmissä käytetä. Poikkeuksena kuitenkin DSDM, jossa niitä tehdään, vieläpä siten että prototyypin koodia käytetään lopullisessa järjestelmässä.
- Dokumentoinnin määrä on erilainen eri menetelmissä. XP ei määrittele projektissa tuotettavaa dokumentaatiota muuten kuin käyttötapausten kirjaamisen osalta. Sen luonteeseen kuuluu, että liikaa dokumentaatiota vältetään, mutta toisaalta yleisenä ohjeena kuitenkin on, että sitä pitää tehdä harkinnan mukaan riittävästi. Joustavissa menetelmissä käytetyn dokumentaation määrä vaihtelee menetelmien välillä. Eniten dokumentaatiota tuotetaan DSDM:ssä, joka määrittelee vaiheiden

lopputulokset tarvittavien dokumenttien avulla ja jossa dokumentaatiota on huomattavasti XP:tä enemmän. Sen dokumentaation määrä muistuttaa jo selvästi perinteisten menetelmien määrää.

- Ohjelmiston arkkitehtuurisuunnitteluun suhtaudutaan XP:ssä muista poikkeavasti. Siinä ei ole selkeää ohjelmiston arkkitehtuurin teknistä suunnitteluvaihetta, mutta useimmissa muissa joustavissa menetelmissä sellainen on. XP:ssä tämä on korvattu refactoring-menetelmällä, jossa ohjelmiston arkkitehtuuria parannetaan jatkuvasti ohjelmointityön aikana.

Verrattaessa muita joustavia menetelmiä XP:hen on helppo huomata, että XP:tä vastaavia ohjeita esimerkiksi pariohjelmoinnista, testauksen automatisoinnista tai koodauskonventioista ei löydy muista menetelmistä. Extreme Programming antaakin huomattavasti tarkempia ohjeita jokapäiväiseen ohjelmointityöhön, kun muut metodologiat ovat luonteeltaan enemmän hallinnollisia.

Tästä erosta on noussut esiin myös mielenkiintoinen ajatus yhdistellä XP:n ohjelmointikäytäntöjä muiden joustavien menetelmien hallinnollisten menetelmien kanssa, luoden näin ”uusia” joustavia menetelmiä. Toisaalta tästä voidaan tehdä se johtopäätös, että joustavien menetelmien menetelmäkehitys on vielä kesken. Raportteja eri joustavia menetelmiä yhdistelevistä kokeiluista on jo tarjolla ja tällaisten yhdistelmien merkitys saattaa kasvaa tulevaisuudessa.

5. Extreme Programming -menetelmät tutkimusten ja kokemusten valossa

Metodologian nimellä ”Extreme Programming” on Beckin [1999] mukaan nimen molemmista sanoista muodostuva merkitys. Sanalla extreme (äärimmäisyys) hän tarkoittaa, että XP:hen kootut hyväksi havaitut käytännöt viedään äärimmäisyyksiin. Esimerkiksi pariohjelmointi on katselmoinnin viemistä äärimmäisyyksiin, jatkuva testaus on testauksen viemistä äärimmäisyyksiin ja nopea evoluutiosykli on palautteen viemistä äärimmäisyyksiin. Sanalla programming (ohjelmointi) hän tarkoittaa, että XP on nimenomaan ohjelmointipainotteinen tapa toteuttaa projekteja. Siinä ei tuoteta paksuja teknisiä suunnittelu- tai testausdokumentteja, vaan pyritään tuottamaan mahdollisimman yksinkertaisia ohjelmia vähemmällä suunnittelulla ja dokumentoinnilla, jolloin aikaa jää enemmän itse ohjelmointiin. Mikäli suunnittelun puute tuottaa huonoja arkkitehtuuriratkaisuja, ne korjataan refactoring-menetelmää hyväksikäyttäen eli ohjelmoimalla.

Extreme Programming sisältää 12 menetelmää. Nämä menetelmät ovat kaikki jo entuudestaan ohjelmistotuotannossa käytettyjä; ne on vain koottu XP:ssä yhteen innovatiivisella tavalla. Eri menetelmillä on vahvuutensa lisäksi kuitenkin myös heikkouksia. XP:n menetelmät on valittu siten, että menetelmien vahvuudet tukevat toisten menetelmien heikkouksia [Beck, 1999]. Näin kokonaisuus pysyy vahvana yksittäisten menetelmien heikkouksista huolimatta.

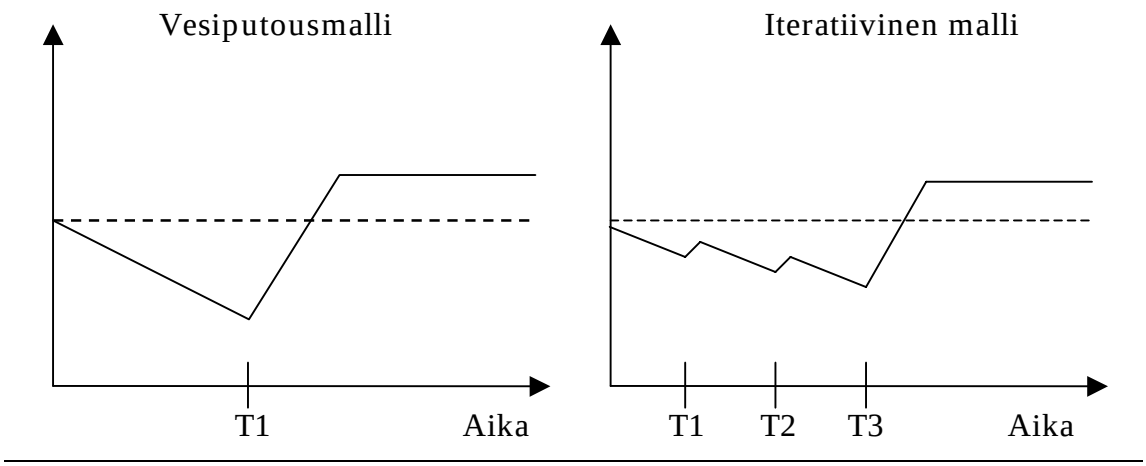
Koska menetelmät ovat melko vanhoja, niitä on myös tutkittu vuosien varrella runsaasti. Kaikkien menetelmistä tehtyjen tutkimusten esittäminen tässä tutkimuksessa on mahdotonta, joten tähän lukuun on koottu eräitä keskeisiä tutkimustuloksia. Luvussa on esitetty myös havaintoja käytännön työelämästä.

5.1. Iteratiivinen elinkaarimalli

Ohjelmistot ovat taloustieteellisesti ajateltuina investointeja, joilla on tietty takaisinmaksuaika. Takaisinmaksuaikana ohjelmiston tuomat hyödyt kattavat sen kustannukset ja sen jälkeen ohjelmisto tuottaa investoinnille tuottoa. Iteratiivista elinkaarimallia käyttävässä ohjelmistoprojektissa on mahdollisuus ottaa ohjelmisto käyttöön osissa, jolloin hyödyt alkavat konkretisoitua jo ensimmäisestä tuotantoonotosta lähtien. Näin ollen ohjelmiston takaisinmaksuaika lyhenee ja se sitoo vähemmän pääomaa. Huonona puolena iteratiivisessa mallissa taas on se, että koko ohjelmisto on testattava jokaisen

iteraatiokierroksen lopuksi, mikä johtaa samojen toimintojen testaamiseen useaan kertaan. Tämä puolestaan kasvattaa ohjelmiston rakentamiseen tarvittavaa aikaa.

Kuvassa 13 on verrattu perinteisen vesiputousmallin ja iteratiivisen mallin takaisinmaksuaikoja ja projekteihin sitoutunutta pääomaa. Vesiputousmallissa kulut kasvavat tasaisesti käyttöönottohetkeen T1 asti, jolloin ne alkavat vähentyä ja nousevat lopulta katkoviivalla kuvatun nollatason yläpuolelle, alkaen maksaa investointia takaisin. Iteratiivisessa mallissa saadaan ohjelmiston hyötyjä käyttöön jokaisen käyttöönoton yhteydessä, jolloin projektin tarvitsema kokonaispääoma pysyy lähempänä nollatasoa. Vaikka iteratiivisen mallin viimeinen käyttöönotto on myöhemmin kuin vesiputousmallin, siihen sitoutuu vähemmän pääomaa, mikä vähentää projektin pääomakustannuksia. Kuvaajassa on oletettu kaikkien kolmen iteraation olevan yhtä arvokkaita, mutta käytännössä on viisainta sijoittaa ensimmäiseen iteraatioon kaikkein arvokkaimmat ohjelmiston ominaisuudet. Mikäli tämä on teknisesti mahdollista, se lisää iteratiivisesta elinkaarimallista saatavia etuja entisestään.



Kuva 13: Vesiputousmallin ja iteratiivisen mallin takaisinmaksuajat.

Iteratiivinen projekti voidaan tehdä myös siten, että iteraation lopuksi asiakas ei ota ohjelmistoa käyttöön, vaan ainoastaan tarkistaa siihen asti tuotetut osat. Tällä tavalla toimittaessa ohjelmistosta saadaan aikaisemmin palautetta sekä mahdolliset virheet ja väärinymmärrykset voidaan korjata nopeammin. Virheiden varhaisen havaitsemisen vuoksi niiden korjauskustannukset pysyvät kohtuullisina, sillä korjauskustannukset ovat sitä alhaisemmat, mitä aikaisemmin virheet havaitaan. Valinta siitä, otetaanko ohjelmisto osissa käyttöön, tulee tehdä asiakkaan toimesta tapauskohtaisesti vertaillen näiden tekijöiden kustannusvaikutuksia.

Iteratiivinen ohjelmistoprojektin malli on myös todettu tehokkaaksi tavaksi kohdata puutteellisiin tai muuttuviin määrittelyihin liittyviä ongelmia [Brownsword ja McUmber, 1991]. Uudet määrittelyt ja niiden priorisointi muihin määrittelyihin nähden voidaan neuvotella jokaisen iteraation lopuksi. Brownsword ja McUmber kuitenkin toteavat, että siirtyminen vesiputousmallista iteratiivisen mallin käyttöön vaatii organisaatiossa huolellista suunnittelua.

Kokemukset käytännön työelämästä tukevat tutkimuksen tuloksia. Iteratiivisissa projekteissa muutosten hallinta on ollut helpompaa. Osa asiakkaista on jopa itse ehdottanut iteratiivisten menetelmien käyttöä havaittuaan, että ohjelmiston vaatimukset eivät ole kokonaisuudessaan selkeitä. Laajalle kuluttajajoukolle avoimien verkkopalveluiden käyttöliittymien kehittämisessä iteraatio on oikeastaan ainoa mahdollinen lähestymistapa, sillä nämä käyttöliittymät täytyy hioa erittäin hyviksi yhteistyössä käytettävyysasiantuntijoiden ja graafikkojen kanssa.

Mikäli ohjelmiston käyttöönotto on erityisen vaikeaa tai siihen liittyy riski esimerkiksi tuotannon katkeamisesta ja sitä kautta taloudellisista menetyksistä, kannattaa käyttöönottojen lukumäärä minimoida. Tällöin iteratiivinen malli useiden käyttöönottojen kanssa ei ole hyvä vaihtoehto. Samoin jos ohjelmisto on pahvilaatikossa myytävä levylle monistettu tuote, asiakas odottaa saavansa uuden version mukana riittävästi vastinetta rahoilleen. Tällöin pienten lisäysten tekeminen ohjelmaan on mahdollista vain ilmaisten päivitysten kautta. Uusiin versioihin on sen sijaan koottava merkittävästi uutta toiminnallisuutta.

5.2. Suunnittelupeli

Määrittelyn muotona suunnittelupeli antaa asiakkaalle mahdollisuuden priorisoida ohjelmiston toimintoja: asiakas voi päättää, mikä on eri toimintojen välinen prioriteetti, ja ohjelmisto rakennetaan ja otetaan käyttöön tässä järjestyksessä. Suunnittelupeli antaa asiakkaalle myös mahdollisuuden muuttaa mieltään projektin aikana ja vaihtaa eri inkrementtien sisältöä tulevien suunnittelupelisessioiden aikana. Se antaa asiakkaalle myös vallan päättää, koska eri inkrementit ovat valmiita ja mitä ne sisältävät. Tämä on merkittävä ero etenkin perinteisiin menetelmiin verrattuna.

Ohjelmiston määritteleminen saattaa olla vaikeaa, varsinkin sellaisille asiakkaille, joilla ei ole kokemusta ohjelmistoprojekteista. Näihin tilanteisiin suunnittelupeli sopii hyvin, sillä määrittely on varsin yksinkertaista sovelletun käyttötapaustekniikan ansiosta. Määrittelyistä asiakas saa myös heti palautetta ja voi näin kehittää määrittelyjään nopeasti.

Perinteisesti vaikeaan työmäärien arviointiin suunnittelupeli tuo lisämahdollisuuksia. Koska arvioita tehdään usein ja pienille asioille kerrallaan, voidaan arvioista odottaa tulevan tarkempia. Arvioita voidaan mitata jokaisen iteraation jälkeen ja tarkentaa niitä mittaustulosten perusteella. Fowler [2000] piti XP-projektin eräänä parhaista onnistumisista nimenomaan aikatauluarvioiden pitävyyttä.

XP:n suunnittelupeli kokoaa hyvin ohjelmiston toiminnalliset ominaisuudet, mutta toisaalta ei-toiminnallisten ominaisuuksien keruuseen ja dokumentointiin ei ole tarjolla formaalia tapaa [Emery, 2002], [Keefer, 2002]. Tällaiset ominaisuudet, kuten suorituskyky tai tietoturva, voivat kuitenkin olla kriittisiä sen kannalta, onko ohjelmistosta lopulta hyötyä asiakkaalle.

5.3. Yhteisen metaforan käyttö

Metafora toteuttaa osaltaan XP:n kommunikaation arvoa tarjoamalla ihmisille yhteisen, helpomman tavan puhua järjestelmästä abstrahoiden tarpeettoman teknisen monimutkaisuuden. Metaforien väitetään vähentävän turhaa asioiden selittämistarvetta ja väärinymmärryksen riskiä. Näin yhteisen kielen löytämisen asiakkaan kanssa tulisi helpottua.

McConnell [1993] esittää, että metafora toimii ongelmanratkaisussa kuin heuristiikka. Se ei kerro suoraan, miten vastaus löydetään, vaan antaa ainoastaan vihjeen siitä, mistä vastaus löytyy. Näin se pitäisi erottaa algoritmista, joka johtaa suoraan vastaukseen. Tämän perusteella voidaan ajatella, että metaforaan kuitenkin liittyy väärinymmärtämisen mahdollisuus. Väärinymmärretty metafora toimii luonnollisesti kommunikaatiota heikentävästi.

Case-tutkimukset Extreme Programming -projekteista osoittavat, että metaforan käyttö oli kaikista 12:sta XP:n menetelmistä vähiten käytetty. Sen hyödyllisyydestä saatiin myös kaikkein vähiten positiivisia kokemuksia, joskaan sen ei todettu vaikeuttavankaan projektien kulkua. Case-tutkimuksista on kerrottu tarkemmin kohdassa 6.1.

5.4. Yksinkertainen suunnittelu

Yksinkertainen suunnittelu on eräänlainen vedonlyönti sen puolesta, että on edullisempaa rakentaa ohjelmiston arkkitehtuuri aloitushetkellä tiedossa olevien toimintojen perusteella ja muokata arkkitehtuuria tarpeen mukaan myöhemmin, kuin että rakennetaan ohjelmistoon kaikkiin muutoksiin varautuva arkkitehtuuri. Tämä on eräs XP:n radikaaleimmista ajatuksista, joka on myös herättänyt paljon keskustelua. Yksinkertainen suunnittelu ei toimisi XP:ssä kuitenkaan yksinään, vaan tarvitsee tuekseen refactoring-menetelmän ja tiiviin yhteistyön asiakkaan kanssa.

Aiemmin esitetyn muutoksen hintaa koskevan kuvaajan perustella on selvää, että myöhemmin ohjelmistoon tullut muutos on kalliimpi kuin aiemmin tullut. Toisaalta myöhemmin rakennettaessa on mahdollista tietää varmasti, mitä ohjelmistoon todella tarvitaan, joten tehtävien muutosten määrä on pienempi. XP:ssä tämä on nimetty ”YAGNI-periaatteeksi” (You Ain’t Gonna Need It). Tämän periaatteen mukaan kehittäjillä on tapana yliarvioida se, mitä järjestelmältä todella tulevaisuudessa vaaditaan ja siten kirjoittaa tarpeetonta koodia.

Eräs perustelu yksinkertaiselle suunnittelulle on lean developmentista tuttu ”just in time” -periaate, jonka mukaan varastot on hyvä pitää mahdollisimman pieninä varastoitaviin tuotteisiin sitoutuvan pääoman takia. Jos ohjelmassa varaudutaan toimintoihin, joita ei lopulliseen ohjelmaan välttämättä tule, sitoutuu projektiin varattua pääomaa tähän koodiin samoin kuin varastossa olevaan fyysiseen tuotteeseen. Tämä kasvattaa projektin kustannuksia ja hidastaa kehitystyötä.

Optimoinnin suhteen XP:n tapa rakentaa mahdollisimman yksinkertainen ohjelmisto ja optimoida se vasta lopuksi on Stevensin [1981] ja McConnellin [1993] mukaan osoittautunut paremmaksi vaihtoehdoksi kuin optimoida ohjelmistoa jatkuvasti. Yksinkertaisuuden arvon noudattaminen tuottaa luonnollisesti tehokkaampia ohjelmia. Ohjelmiston optimointi vasta projektin lopuksi on mahdollista kaikissa projektinhallintametodologioissa, mutta vain XP määrittelee sen eksplisiittisesti.

Tulevaisuudessa kokemukset XP-metodologian käytöstä osoittavat, onko aiemmin esitetty vedonlyönti kannattava. Edellisen perusteella voidaan todeta, että jos järjestelmän perusvaatimuksena on rakentaa helposti laajennettava järjestelmä tai kehityksen kohteena on yleiskäyttöinen luokkakirjasto, XP ei ole oikea projektinhallintamenetelmä.

5.5. Automatisoitu testaus

Testauksen automatisointi XP:ssä ei eroa juurikaan perinteisestä moduulitestauksesta. Tosin Extreme Programming poikkeaa perinteistä siinä, että se suosittelee moduulitestien kirjoituksen tapahtuvan ennen koodin kirjoittamista. Lisäksi moduulitestejä suositellaan kirjoitettavaksi aina kun niitä tarvitaan, eli myös ohjelmoinnin aikana, sen jälkeen ja aina virheen löytyessä. Moduulitestaus myös tapahtuu jokaisen integroinnin jälkeen, eikä vasta projektin testausvaiheessa. Näin saadaan aiemmin palautetta ohjelmiston tilasta. Kuitenkin voidaan olettaa, että XP:n moduulitestaukseen pätevät samat edut ja ongelmat ovat samansuuntaisia kuin perinteisessä moduulitestauksessa.

Moduulitestaus on paljon käytetty keino ohjelmistojen laadunvarmistuksessa. Se vähentää lopputuotteessa olevaa virheiden määrää ja

siten niistä johtuvia kustannuksia. Joissain tapauksissa moduulitestaukseen panostettu aika voitaisiin käyttää tehokkaamminkin esimerkiksi ihmisen tekemän testauksen avulla. Testien suunnittelussa voi olla myös ongelmia. Usein on esimerkiksi vaikeaa suunnitella testiä, joka testaisi vain tietyn osan koodista eikä olisi riippuvainen suuresta joukosta siihen liittyvää koodia. Moduulitestausta ei myöskään välttämättä voi tehdä erilaisille pienlaitteille, joissa ohjelmakoodi on integroitu ROM- tai flash-muisteihin. Myös itse testiin voi tulla virhe, joka puolestaan saattaa peittää virheen varsinaisessa ohjelmassa.

Kirjoittajalla on hieman kokemusta testauksen automatisoinnista työelämässä. Automatisoinnin aloitus on ollut hankalaa, sillä moduulitestausta ei juuri opeteta alan oppilaitoksissa, vaan se täytyy opetella työn ohessa. Moduulitestauksen oppiminen ei kuitenkaan ole ohjelmointitaitoiselle vaikeaa, moduulitestit syntyvät melko nopeasti ja niiden avulla on myös havaittu virheitä ennen ohjelmiston osien integrointia. Testien automaattinen ajaminen jokaisen buildin yhteydessä tuo turvallisuuden tunnetta siitä, etteivät ohjelmistoon tehdyt muutokset ole hajottaneet aiemmin kirjoitettua koodia. Näin ollen näen automatisoidun moduulitestauksen olevan varsin hyödyllistä. Siihen käytetty aika tulee todennäköisesti maksamaan itsensä takaisin erityisesti silloin, kun ohjelmiston laatutavoitteet on asetettu korkealle.

5.6. Refactoring

Refactoring, josta on esitetty myös suomennosta refaktorointi, tarkoittaa ohjelmakoodin arkkitehtuurin muuttamista ilman sen toiminnallisuuden muuttamista. Virheiden korjausta tai nimeämiskäytännön muutoksia ei siis voida pitää refaktorointina. Tavoitteena refaktoroinnilla on parantaa ohjelman rakennetta, jotta sitä olisi helppo ylläpitää tulevaisuudessa. Helppo ylläpidettävyys on XP-kehitysprojekteissa tärkeää, sillä toimintaympäristöstä on odotettavissa jatkuvia muutostarpeita. Näin muutosten hintaa voidaan vähentää. Refaktoroinnin etuna on myös mainittu helpompi ohjelmavirheiden löytyminen: kun ohjelmakoodi on selkeää, virheetkin on helpompi nähdä. Usein selkeä arkkitehtuuri helpottaa myös moduulitestien kirjoittamista.

Extreme Programming suosittelee, että projektiryhmässä kuka tahansa voi refaktoroida kenen tahansa ryhmäläisen ohjelmakoodia. Onnistuakseen refactoring tarvitseekin toimivat moduulitestit ja ryhmässä sovitut koodauskonventiot. Refactoring voi tuoda ohjelmaan virheitä, joko refaktoroituun koodiin tai siihen yhteydessä oleviin moduleihin. Testien avulla tällaiset virheet voidaan havaita heti. Ilman koodauskonventioita refaktorointi voi sekoittaa eri ihmisten konventioita keskenään, jolloin koodin luettavuus kärsii.

Refactoring on erityisen tarpeellinen XP-projekteissa, sillä niissä ei erikseen suunnitella ohjelmiston arkkitehtuuria etukäteen. Refactoring tuntuukin olevan nimenomaan arkkitehtuurisuunnittelun korvaaja. Tämä on eräs XP:n eniten kritisoituja ominaisuuksia, sillä yleinen mielipide ohjelmistokehittäjien keskuudessa on, että on parempi tapa suunnitella hyvä arkkitehtuuri projektin alussa kuin muuttaa sitä useasti projektin aikana. Argumenttia erillisestä arkkitehtuurisuunnittelusta tukee se seikka, että useimmissa muissa joustavissa menetelmissä on erillinen arkkitehtuurisuunnittelun vaihe.

5.7. Pariohjelmointi

Kahden tai useamman hengen ryhmissä työskentely on kauan tunnettu käytäntö kaikissa teknistä suunnittelua vaativissa ammateissa, myös ohjelmoinnissa. Monissa perinteisissä ohjelmointiprojekteissa järjestetään ryhmissä suunnittelupalavereja ja koodikatselmointeja. XP:n suosiman pariohjelmoinnin eduiksi on esitetty nopeammin valmistuvat projektit, parempi koodin laatu ja kasvanut työtyytyväisyys.

Pariohjelmointi on osoittautunut [McDowell *et al*. 2002], [Williams ja Upchurch, 2001] tehokkaaksi yliopistojen kursseilla järjestetyissä kokeissa. Ohjelmien rakentamiseen tarvittu aika lyhentyi, niissä oli valmistuttuaan vähemmän virheitä, pariohjelmointiin osallistuneet opiskelijat olivat tyytyväisiä työtapaan ja he halusivat jatkaa pariohjelmointia myös tulevaisuudessa. Mielenkiintoinen tulos oli myös se, että pariohjelmointi tuotti 10-30 prosenttia vähemmän koodirivejä, vaikka kirjoitetut ohjelmat toimivat samalla tavalla. Näiden tutkimusten harjoitustehtävät olivat kuitenkin pieniä, muutaman sadan rivin ohjelmia. Näin ollen pariohjelmoinnin soveltuvuudesta huomattavasti suurempien reaalimaailman sovellusten kehittämiseen ei voida näiden tutkimusten perusteella esittää kuin suuntaa antavia kommentteja.

Reaalimaailman ohjelmistokehityksessä pariohjelmointia käyttäneet projektit ovat valmistuneet 40-50 prosenttia nopeammin yksin ohjelmoitaviin projekteihin verrattuna [Williams *et al*. 2000]. Samansuuntaisia tuloksia on saatu myös lähteessä [Cockburn *et al*. 2000], jossa pariohjelmointiprojektin suorissa palkkakustannuksissa arvioitiin 15 prosentin kasvu yksittäisiin ohjelmoijiin verrattuna. Molemmissa tutkimuksissa kuitenkin korostettiin, että ohjelmiston laatu on lisääntynyt merkittävästi, ihmisten työtyytyväisyys on kasvanut ja ohjelmointitieto levinnyt. Pariohjelmointina tehdyt ohjelmat läpäisivät suuremman osuuden testitapauksista kuin yksin ohjelmoidut. Pariohjelmoinnin kustannuksia laskettaessa lähteessä [Cockburn *et al*, 2000] kehotetaan laskemaan ohjelmistoprojektin kokonaiskustannukset, jotta pariohjelmoinnin kokonaistaloudellisuus saataisiin selville. Vähentyneen virheiden määrän myötä testaamisen ja tuotetuen kustannukset vähentyvät,

joten pariohjelmointi nähdään tällä perusteella myös kustannustehokkaammaksi kuin yksin ohjelmointi. Kaikkein radikaaleimmat tutkimustulokset [Jensen, 2003] kertovat 10 hengen ja 50 000 koodirivin projektissa tehokkuuden kasvaneen jopa 127 prosenttia virheiden vähentyessä samalla yhteen tuhannesosaan, kun organisaatiossa toteutettua pariohjelmointiprojektia verrattiin aikaisempien projektien mittaustuloksiin.

Ohjelmoijien tuottavuuserot ovat tunnetusti melko suuria. Parhaat ohjelmoijat tuottavat ohjelmistoja jopa kymmenkertaisella nopeudella heikoimpiin verrattuna, samoin erot virheiden määrässä ovat kymmenkertaiset [McConnell, 1993]. Pariohjelmointi lisää tiedon liikkuvuutta ohjelmoijien välillä, joten sen voisi odottaa myös kaventavan ohjelmoijien välisiä tuottavuuseroja. Tutkimustuloksia asiasta ei kuitenkaan vielä ole. Mikäli tätä jossain tutkimuksessa tarkasteltaisiin, olisi myös mielenkiintoista tietää, onko erojen kaventuminen tapahtunut parhaiden ohjelmoijien tuottavuuden kustannuksella.

Kirjoittajalla on myös alustavia kokemuksia pariohjelmoinnista työelämästä. Ne tukevat edellä mainituissa tutkimuksissa esiin tulleita seikkoja. Pariohjelmointi on nopeuttanut uuden teknologian opettelua ja monimutkaisten konfigurointiongelmien ratkaisua, mutta toisaalta helpohkoissa ohjelmointitehtävissä siitä ei ole vastaavaa hyötyä. Myös ryhmähenkeen ja sitä kautta työtyytyväisyyteen sillä on alustavan arvion mukaan ollut positiivinen vaikutus. Pariohjelmointia voi kuitenkin olla vaikea järjestää, jos organisaatiossa on joustavat työajat ja ryhmän jäsenten vuorokausirytmit poikkeavat toisistaan. Samoin pariohjelmoinnissa syntyy luonnollisesti keskustelua, joka saattaa häiritä samassa huoneessa yksin työskenteleviä. Jotkut ihmiset myös työskentelevät luonnostaan mielummin yksin.

Extreme Programming määrittelee [Beck, 1999], että kaikki lopulliseen ohjelmistotuotteeseen päätyvä ohjelmakoodi tulisi tuottaa pariohjelmointina. Kuitenkaan yhdessäkään pariohjelmointia tai XP:tä käsittelevässä case-tapauksessa tätä sääntöä ei noudatettu. Pariohjelmointia käytettiin projektin alkuvaiheessa, sillä monet asiat olivat vielä silloin epäselviä. Projektin edetessä pariohjelmoinnista luovuttiin, kun ohjelmoijat saavuttivat varmuuden siitä, mitä oltiin tekemässä. Uskonkin, että XP:n tiukka pariohjelmointisääntö on lievenemässä kohti pariohjelmoinnin käyttöä silloin, kun se on tarkoituksenmukaista. Jatkuva integrointi, testaus ja koodin yhteisomistus tukevat kuitenkin riittävästi projektin laatutavoitteita.

Pariohjelmoinnin käyttöä ohjelmistokehityksessä saattaa olla vaikea perustella organisaation johdolle ja asiakkaalle. Näiden ryhmien tyypillinen

ajattelutapa on, että pariohjelmointi kaksinkertaistaa kustannukset, eikä tuottavuuden kasvuun ja parantuneeseen laatuun kiinnitetä huomiota. Pariohjelmointia onkin paras esittää johdolle johdon omalla kielellä, jossa kerrotaan selkeitä lukuja tuottavuudesta ja investointien kannattavuudesta (return of investment, ROI) [Cockburn *et al*. 2000].

5.8. Koodin yhteisomistus

Koodin yhteisomistus on toinen XP:n kiistellyimmistä menetelmistä. Siitä huolimatta itse yhteisomistuksesta ei ole tutkimustuloksia tarjolla, joten yhteisomistuksen vaikutusta projekteihin voidaan vain spekuloida.

Koodin yhteisomistus vaatii luonnollisesti yhteisen koodauskonvention noudattamista. Se asettaa käytännössä myös suuret vaatimukset koodin kommentoinnille, jotta toinen ohjelmoija voisi ymmärtää, miksi koodi on kirjoitettu tietyllä tavalla. Toimiessaan tämä on hyvä asia, mutta usein koodin kommentointia täydennetään vasta projektin lopussa. Näin ollen siitä ei ole hyötyä ohjelmaa rakennettaessa vaan vasta jatkokehitysvaiheessa.

Yhteisomistus vaatii käytännössä versionhallintaohjelmiston käyttöä, mikä toisaalta on varsin itsestään selvä asia usean ohjelmoijan projekteissa. Koska jokainen ohjelmoija voi editoida jokaista tiedostoa, voidaan versionhallinnan konfliktien odottaa lisääntyvän. Näiden selvittely saattaa hidastaa projektin etenemistä.

Nopea palaute automaattisista testeistä ja asiakkaalta on yhteisomistuksessa entistä tärkeämpää, jotta mahdolliset virheet huomattaisiin mahdollisimman aikaisin.

Projektissa on usein eri ohjelmoinnin osa-alueiden asiantuntijoita, kuten käyttöliittymiin tai tietokantoihin erikoistuneita henkilöitä. Heidän tekemänsä koodin yhteisomistus ei ehkä tuo toivottua tulosta, sillä ryhmän muut jäsenet eivät ehkä uskalla editoida heidän koodiaan, vaikka huomaisivatkin siinä ongelmia. Toisaalta joillain pitkälle erikoistuneilla asiantuntijoilla osaaminen on voinut tulla niin kapea-alaiseksi, ettei heidän tekemästään koodikatselmoinnista ohjelman muihin osiin saada vastaavaa hyötyä.

Mielenkiintoinen kysymys on myös, muuttaako yhteisomistus ohjelmoijien suhtautumista koodiinsa. Kun kaikki työtoverit lukevat koodia, kasvaako ohjelmoijan vastuuntunto oman koodinsa suhteen? Vai voiko ohjelmoija ajatella, että hän voi jättää koodiin huonoja ratkaisuja, koska joku korjaa ne kuitenkin myöhemmin? Koodin yhteisomistus luo potentiaalisen riskin, jos ryhmän yhteishenki heikkenee huomattavasti. Ryhmän jäsenet voivat alkaa syytellä virheistä toisiaan ja yhteisvastuu voi muuttua vastuuttomuudeksi. Työelämän koodikatselmoinneista saadun kokemuksen mukaan ohjelmoijat haluavat usein hioa koodiaan ennen katselmointeja, joten näin voidaan olettaa

yhteisomistuksen mielummin lisäävän kuin vähentävän ohjelmoijien vastuun tunnetta omasta työstään.

5.9. Jatkuva koodin integrointi

Jatkuva koodin integrointi on sekin pitkään ohjelmistokehityksessä mukana ollut käytäntö. Monet sekä Microsoftin että Open Source -yhteisön projektit integroivat ohjelmistonsa säännöllisesti joka yö. Näissä integroinneissa kääntyy kymmeniä miljoonia koodirivejä.

Integroinnissa löytyy yleensä erityisesti eri ohjelmoijien tekemien komponenttien välisen vuorovaikutuksen ongelmia. Koska perinteisessä vesiputousmallissa integrointi tapahtuu vain kerran, näitä ongelmia voi tulla kerralla paljon ja integrointi voi tuntua vaativalta. Näin ollen jatkuva integrointi voi tuntua psykologisesti epämiellyttävältä vesiputousmallin ongelmiin tottuneiden ohjelmistokehittäjien mielestä. Usein tapahtuva integrointi pitää kuitenkin integroitavat inkrementit pieninä, jolloin ongelmat ovat helpommin löydettävissä ja integrointiin tarvittava aika vähenee. Näin saadaan myös aiemmin palautetta ohjelmiston tilasta, eikä arkkitehtuurisia katastrofeja pääse syntymään kovin helposti.

Jatkuva koodin integrointi asettaa vaatimuksia ohjelmistokehitysympäristölle. Käytännössä se vaatii yhden koodilähteen käyttöä (single source point), johon tarpeeseen versionhallintaohjelmistot vastaavat hyvin. Samoin integrointi on pystyttävä mahdollisimman pitkälle automatisoimaan samankaltaisena toistuvan työn välttämiseksi. Tähän tarkoitukseen voidaan käyttää vaikkapa Ant-skriptejä. Lopuksi jatkuva integrointi vaatii toimivan, automatisoidun moduulitestauksen. Näin saadaan jälleen palautetta ohjelmiston tilasta jokaisen integraation jälkeen.

Jatkuva integrointi vaatii myös, että jokainen ohjelmoija vie toimivaa koodia versionhallintaan päivittäin, jotta palaute ohjelmiston tilasta olisi mahdollisimman oikeaa. Tällä on sikäli hyvä vaikutus, että pitkällinen koodin pitäminen itsellä ei ole hyvä käytäntö. Erityisesti tämä korostuu XP-projekteissa, joissa koodia voidaan jatkuvasti refaktoroida. Mutta asian kääntöpuoli on se, että useinkaan ei ole mahdollista saada joka päivä toimivaa koodia aikaan. Ihmiset voivat olla työmatkoilla, lomalla tai sairastua. Samoin jotkut ohjelmointiongelmat vain vievät enemmän aikaa.

Kirjoittajalla on kokemusta jatkuvista integroinneista projektissa, jossa integroitiin säännöllisesti joka viikko. Toisaalta ohjelmoijat eivät viettäneet kaikkea aikaansa projektin parissa, vaan yhdestä kahteen työpäivää viikossa. Ensimmäiset integrointikerrat olivat vaikeampia, toivat esiin virheitä ja veivät aikaa, mutta myöhempinä viikkoina integroinnit sujuivat nopeasti. Ohjelmisto valmistui ajallaan ja oli hyvälaatuinen.

5.10. Koodauskonvention käyttö

Koodauskonventio koostuu kahdesta asiasta: nimeämistyylistä ja ohjelmointityylistä. Nimeämistyylillä tarkoitetaan sitä, miten esimerkiksi ohjelmiston muuttujat, metodit ja luokat nimetään. Ohjelmointityylillä puolestaan tarkoitetaan sitä, miltä ohjelmakoodi näyttää visuaalisesti eli miten se on rivitetty, sisennetty ja miten eri ohjelman osat on erotettu toisistaan.

Muuttujien nimeämisessä tärkeintä on se, että annettu nimi kuvaa täsmällisesti ja kokonaisuudessaan sitä asiaa, jota muuttuja ohjelmakoodissa edustaa [McConnell, 1993]. Nimen pitäisi kuvata mahdollisuuksien mukaan ohjelman mallintamaa reaalimaailmaa eikä niinkään tietoteknisiä asioita. Sopivasta nimien pituudesta ja niin sanotun unkarilaisen notaation käytöstä keskustellaan edelleen. Java-ohjelmoijien keskuudessa käytetään konventiosta johtuen pitkiä ja kuvaavia nimiä, Windows-ohjelmoijat taas ovat perinteisesti käyttäneet unkarilaista notaatiota.

Ohjelmointityylin perimmäinen tarkoitus on visualisoida ohjelman rakennetta. Hyvään ohjelmointityyliin kuuluu lauseiden ryhmittely yhteenkuuluviin joukkoihin, yleensä ehto- ja toistorakenteiden mukaan, pitäen perussääntönä yhtä lausetta yhdellä kooditiedoston rivillä. Ryhmittely saadaan aikaan käyttämällä koodissa tyhjiä rivejä sekä koodin sisentämistä tabulaattorin avulla. Myös yhtenäinen käytäntö koodin kommentoinnissa kuuluu hyvään ohjelmointityyliin.

Organisaatioilla on yleisesti käytössään erilaisia dokumenttipohjia, jotta kaikissa niiden tuottamissa dokumenteissa olisi yhtenäinen ulkoasu ja rakenne. Ohjelmakoodi on ohjelmistoyrityksen tärkein dokumentti, joten on luonnollista, että koodi kirjoitetaan valmiille pohjalle. Nämä valmiit pohjat yhtenäistävät koodin ulkoasua jo merkittävästi, samalla kun tuovat esiin koodin metatiedot eli kirjoittajan, version, kirjoitusajan, tekijänoikeudet ja niin edelleen. Hyvän ohjelmointityylin saavuttaminen tapahtuu helpoimmin, mikäli projektiryhmällä on käytössään yhtenevä ohjelmointiympäristö. Mikäli ryhmän käyttämät ohjelmointiympäristöt poikkeavat toisistaan, niiden oletusasetukset saattavat tuottaa erityylisiä lähdekooditiedostoja.

Jokainen toisen ihmisen kirjoittamaa koodia lukenut ihminen ymmärtää, kuinka tärkeää yhteisen koodauskonvention käyttö on. Huonosti nimetyt muuttujat ja outo tyyli hidastavat merkittävästi koodin ymmärtämistä ja johtavat helpommin väärinymmärryksiin sekä sitä kautta ohjelmointivirheisiin. Ilman konventiota kirjoitetulla koodilla on myös huono vaikutus ryhmähenkeen siinä vaiheessa, kun toisen kirjoittamaa koodia pitää ryhtyä ylläpitämään. Koodauskonventioihin liittyvät ongelmat tuntuvat kuitenkin

olevan työelämästä väistymässä, sillä alan koulutuksessa asiaan on kiinnitetty huomiota jo ensimmäisistä ohjelmointikursseista lähtien.

Koodauskonventiosta voidaan sopia projektiryhmäkohtaisesti, organisaatiokohtaisesti tai jopa globaalisti. Viimeisestä esimerkkinä ovat Java-ohjelmointikielen koodauskonventiot, jotka määrittelevät varsin tarkasti sekä nimeämis- että ohjelmointityylin. Kyseinen konventio on erittäin suosittu ja on muodostunut jo de facto -standardiksi Java-ohjelmoijien keskuudessa. Konvention käyttöä helpottamaan on myös kehitetty erilaisia tarkistusohjelmia, jotka raportoivat automaattisesti poikkeamista. Tärkein koodauskonventioiden seurantatapa ovat kuitenkin katselmoinnit.

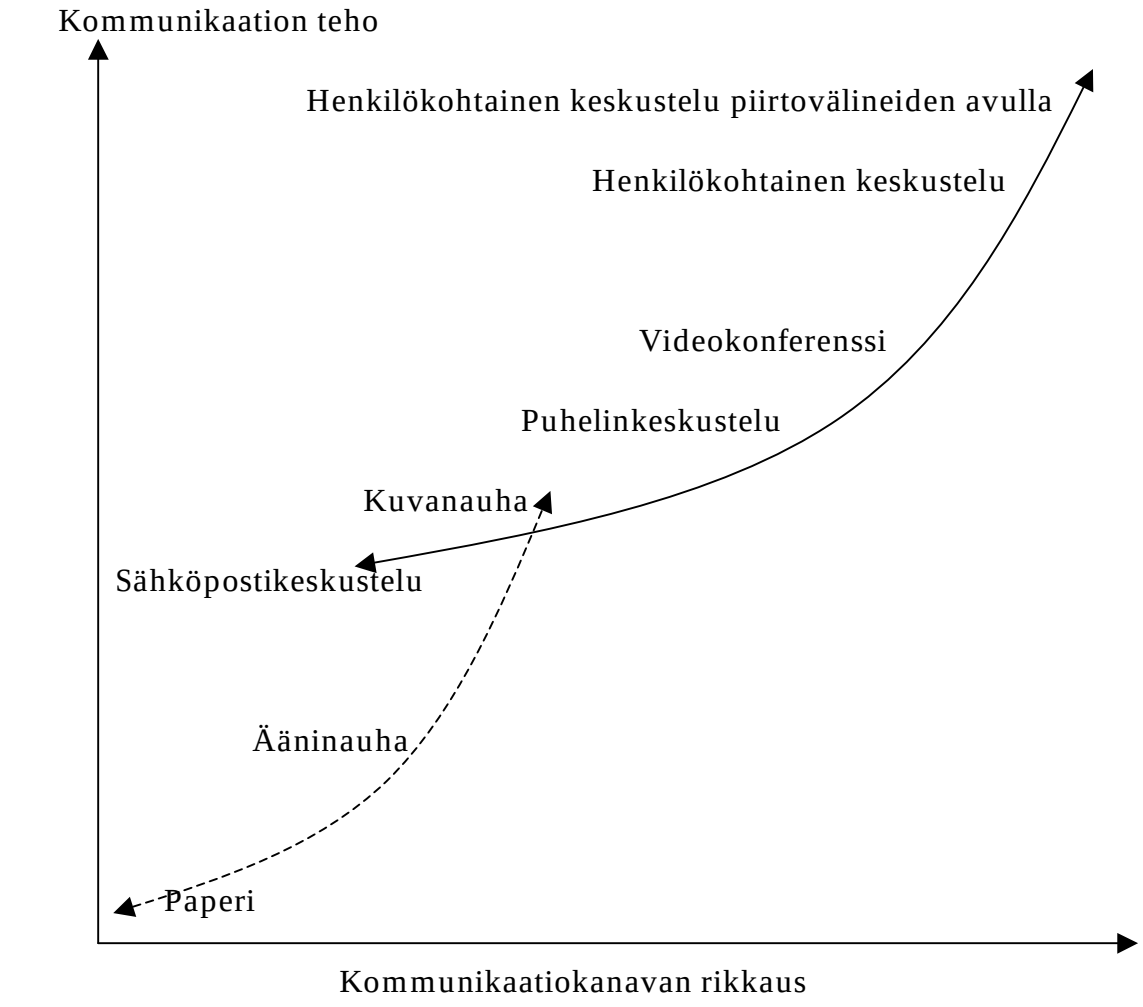
Sunin standardiksi muodostunut Java-konventio osaltaan osoittaa jo koodauskonvention käytön tarpeellisuuden. Koodauskonventioiden käytöstä ei ole raportoitu mitään negatiivisia vaikutuksia, joskin jotkut ohjelmoijat saattavat argumentoida, että heidän luovuuttaan rajoitetaan niiden avulla. Todellisuudessa heidän luovuutensa kuitenkin vain ohjataan tuottavaan työhön.

5.11. Läheinen yhteistyö asiakkaan kanssa

Asiakkaan ja toimittajan välisen kommunikaation puute todettiin aiemmin suurimmaksi yksittäiseksi syyksi ohjelmistoprojektien epäonnistumiseen. Näin ollen XP:ssä asiakkaan tulee olla jatkuvasti fyysisesti lähellä projektiryhmää. Lähellä ollessaan asiakkaalla on mahdollisuus vastata välittömästi ohjelmoijien kysymyksiin, tarkentaa määrityksiä ja antaa palautetta näkemästään ohjelmasta. Tämä toimintamalli perustuu XP:n kommunikaation arvoon.

Cockburn [2002a] esittää kuvan 14 mukaisen mallin siitä, kuinka tehokasta kommunikaatio on eri välineitä käytettäessä. Kuvaajassa esitetään katkoviivalla erilaisia dokumentaatiovälineitä ja yhtenäisellä viivalla erilaisia välineitä ohjelmiston mallintamiseen. Paperidokumentaatio nähdään heikoimmaksi dokumentaatiovälineeksi, vaikka sitä ohjelmistoprojekteissa on perinteisesti tehty runsaasti. Paperidokumentti kuitenkin rajoittaa ilmaisun kuviin ja sanoihin, kun taas liikkuva kuva ja ääni tuovat dokumentaatioon uusia mahdollisuuksia. Samoin mallintamisessa heikoimmaksi nähdään sähköpostikeskustelu, koska siinä on rajoitetuin ilmaisuvoima ja se perustuu kysymys-vastaus -malliin. Mallintamisen teho nousee sitä suuremmaksi, mitä suuremmaksi ilmaisuvoima kasvaa. Henkilökohtaisessa kommunikaatiossa toiselle osapuolelle voi esittää spontaaneja kysymyksiä eikä aikaa kulu keskustelun tallentamiseen esimerkiksi kirjoittamalla. Se mahdollistaa myös sanattoman kommunikaation. Kaikkein tehokkaimmaksi nähdään henkilökohtainen keskustelu piirtovälineiden avulla, esimerkiksi fläppitaulun

ääressä. Tällöin on käytössä parhaat mahdollisuudet oman asian selkeään esittämiseen.



Kuva 14: Erilaisten kommunikaatiomallien tehokkuus.

Kommunikaatioteoriasta tiedämme, että viestinnän tehokkuuteen vaikuttavat muutkin seikat kuin käytettävä väline. Vaikuttavia seikkoja ovat lisäksi viestin lähettäjä, sen vastaanottaja ja itse viestin sisältö. Näin ollen eri kanavien tehokkuus saattaa vaihdella riippuen sitä käyttävistä ihmisistä ja viestin sisällöstä. Yksittäisissä tapauksissa kuvaajassa esitetty järjestys ei ehkä pidäkään paikkaansa.

Asiakkaan lähellä oleminen olisi ideaalitilanne jokaisessa ohjelmistoprojektissa. Oma kokemus ja lähteet [Keefer, 2002] kuitenkin osoittavat, että asiakkailla on myös useimmiten kiire omien töidensä kanssa, eikä heillä ole realistisia mahdollisuuksia antaa kyvykästä henkilöä projektiryhmän jatkuvaan käyttöön. Se merkitsisi myös asiakkaan kustannusten selvää kasvua. Jos asiakas voi irrottaa henkilön omista tehtävistään, hän voi olla kokematon eikä siten pysty antamaan riittävää

panosta projektiryhmän käyttöön. XP tarjoaa siis projektiryhmän kannalta ihanteellisen ratkaisun ongelmaan, mutta asiakkaan kannalta ratkaisu ei ole paras mahdollinen.

5.12. Ihmisläheiset piirteet

Työssä viihtymisen merkityksestä tuottavuuden lisääjänä on viime aikoina keskusteltu paljon. Asialla on kuitenkin jo pitkä historia, olihan Toyota Production Systemin eräs parannus Henry Fordin menetelmiin nimenomaan henkilöstön hyvinvointiin ja osallistumiseen keskittyvä ajattelutapa. Ensimmäinen ohjelmistoalan julkaisu ohjelmistoalan ”pehmeistä arvoista” oli Edsger Dijkstran ”Programming considered as human activity” [Dijkstra, 1965]. Uusinta näkemystä asiasta edustaa ehkä jo klassikon mainetta nauttiva kirja Peopleware [DeMarco *et al*, 1999].

XP:ssä ihmisläheisiä piirteitä kuvaa käytäntö, jonka alkuperäinen nimi on 40 tunnin työviikko. Beck [1999] määritteli tämän kirjassaan siten, ettei projektiryhmä saa tehdä ylitöitä, ei ainakaan kahtena peräkkäisenä viikkona. Joustavissa menetelmissä tämä käytäntö on kuitenkin laajennettu kaikkia niiden ihmisläheisiä piirteitä koskevaksi, sillä tämä on perimmäinen idea niin käytännön kuin Agile Manifestonkin takana.

Moni organisaatio luottaa täsmällisesti määriteltyihin prosesseihin projektien onnistumisten edellytyksenä. Määritellyissä prosesseissa perusideana on, että kaikki työntekijät saavuttavat prosessia noudattamalla avulla hyväksyttävän laatutason. Hyvät yksilöt saavuttaisivat tämän prosessista riippumatta, mutta heikommat tarvitsevat siihen prosessin apua. Tunnettu prosessien määrittelymalli on Carnegie-Mellon -yliopiston kehittämä Capability Maturity Model eli CMM, jossa organisaation prosessien määrittelytaso jaetaan viiteen luokkaan.

Joustavat menetelmät omaavat luonnostaan toisenlaisen lähestymistavan. Ne luottavat kevyeen prosessiin ja keskittyvät ihmisiin. Niissä ei yritetä saada kaikkia ihmisiä toimimaan saman prosessin mukaan, vaan pyritään löytämään ihmisten henkilökohtaisia vahvuuksia ja panostetaan niihin. Syynä tähän on se, että ohjelmistokehityksessä esiin tulevat ongelmat ovat kompleksisia ja epälineaarisia, eikä niitä ole mahdollista ratkaista yksinkertaistetuilla lineaarisilla prosesseilla [Cockburn, 2001].

Myös XP:n prosessi on kevyt, mutta se on kuitenkin erittäin tarkasti määritelty. Prosessia täytyy myös noudattaa, sillä menetelmät tukevat toisiaan ja prosessista lipsuminen tuo esiin menetelmien heikkoudet. Tässä suhteessa XP eroaa merkittävästi muista joustavista menetelmistä. XP:n lähestymistavassa voidaan nähdä siis ristiriitaisuutta Agile Manifeston periaatteiden kanssa.

Käytännön ohjelmistokehitystyössä ihmisläheisten arvojen merkitys tulee esiin. On selvää, että pariohjelmoinnin toimimiseksi projektiryhmän tulee osata toimia ja kommunikoida keskenään, eikä ryhmän sisäisiä ristiriitoja saa olla. Ryhmähengen tai ryhmän ihmissuhteiden heikkeneminen voi tuottaa XP:ssä suurempia ongelmia kuin perinteisissä prosesseissa, koska sillä on suora vaikutus pariohjelmoinnin ja suullisen viestinnän onnistumiseen.. Samoin henkilöstön vaihtuvuus on suurempi riski XP-projekteissa, joissa tieto ei ole dokumenteissa, vaan ihmisten hiljaisena tietona. Tosin Cockburn *et al.* [2000] kertovat tapauksesta, jossa pariohjelmointi on itse asiassa auttanut hajanaista ryhmää saavuttamaan hyvän ryhmähengen ja lisäämään ryhmän sisäistä kommunikaatiota.

Vaikka XP:n tuomat edut ja riskit henkilöstön tyytyväisyyteen on tuotu tutkimuksissa esiin, voidaan kuitenkin olettaa, että henkilöjohtamisella ja työntekijöiden hyvinvoinnista huolehtimisella on suurempi vaikutus henkilöstön tyytyväisyyteen kuin millään yksittäisellä ohjelmistotuotannon prosessilla.

6. Metodologian arviointia kokonaisuutena

6.1. Case-tutkimuksia XP-projekteista

Extreme Programmingin herättämä mielenkiinto on tuottanut siitä useita yksittäisiä case-tutkimuksia. Runsauden vuoksi niistä tässä tutkimuksessa tarkastellaan kolmea. Ensin esitellään C3-projekti, joka on historian ensimmäinen XP-projekti ja joka esitellään malliesimerkkinä Beckin XP:tä esittelevässä perusteoksessa [Beck, 1999]. Lisäksi tarkastellaan kahta muuta tutkimusta. Näistä ensimmäinen esittelee XP:n käyttökokemuksia uudehkossa keskisuuressa Secure Trading -yrityksessä, joka on sitoutunut käyttämään projektinhallinnassaan vain XP:tä. Jälkimmäinen kertoo XP:n käyttöönotosta yksittäisessä projektissa eräässä suuressa yrityksessä, jossa on aiemmin käytetty vain perinteisiä projektinhallintamenetelmiä.

Chrysler Comprehensive Compensation System eli lyhennettynä C3 oli DaimlerChrysler -konsernin 87 000 työntekijän palkanmaksuun tarkoitettu ohjelmisto. Projekti alkoi tammikuussa 1995 ja maaliskuussa 1996 se oli vaikeuksissa noudatettuaan siihen asti perinteisiä menetelmiä suurella projektiryhmällä. Tällöin XP:n kehittänyt Kent Beck tuli mukaan projektiin ja projektissa otettiin projektinhallintamenetelmäksi XP. Projekti alkoikin tämän jälkeen saada nopeasti tuloksia aikaan.

Projekti ei kuitenkaan onnistunut saavuttamaan tavoitteitaan [Keefer, 2002]. Ohjelmisto pystyi parhaimmillaan maksamaan palkkaa noin 90 prosentille työntekijöistä. Helmikuussa 2000 projekti lopetettiin keskeneräisenä ja DaimlerChrysler lopetti XP:n käytön projektinhallinnassa. Tämän perusteella voidaan kyseenalaistaa C3-projektin maine XP-projektien malliesimerkkinä.

Secure Trading [Gittins *et al*, 2001] hyödyntää ohjetta, jonka mukaan XP kannattaa ottaa käyttöön osissa [Beck ja Fowler, 2000]. Projektiryhmä otti ensimmäisenä menetelmänä käyttöön pariohjelmoinnin, joka ei ollut niin suuri menestys kuin olisi voitu olettaa. 28 % ohjelmoijista olisi mielummin työskennellyt yksin ja 57 % oli sitä mieltä, ettei pariohjelmointi voisi toimia kaikkien projektiryhmän jäsenten kanssa. Lisäksi erilaiset projektien ylläpitotehtävät hankaloittivat pariohjelmointia.

Seuraavaksi Secure Trading otti käyttöön suunnittelupelin, jossa se onnistui paljon paremmin. Hankalatkin käyttötapaukset saatiin pilkottua useiksi helpoiksi tapauksiksi. Sen sijaan asiakkaan jatkuva läsnäolo ei ollut mahdollista, joten asiakkaan virkaa toimitti asiakkaan kanssa läheisessä yhteistyössä toiminut henkilö. Kommunikaation onnistumiseen Secure Trading panosti paljon ja otti XP:hen mukaan Scrum-kehyksen mukaiset lyhyet

päivittäiset palaverit, joista he saivat hyviä kokemuksia. Sen sijaan kommunikaatioon läheisesti liittyvästä metaforan käytöstä ei nähty hyötyä ja siitä luovuttiin ainakin aluksi.

Itse ohjelmoinnin osalta yksinkertaisesta suunnittelusta oli luovuttava, koska Secure Trading ei katsonut olevansa tarpeeksi kypsä kantamaan sen tuomaa riskiä. Automatisoitu testaus nähtiin hyödylliseksi, mutta se herätti ohjelmoijissa myös vastustusta, kun projektin aikataulu kävi tiukaksi. Lisäksi kyselyyn vastanneista peräti 71 % oli sitä mieltä, että yrityksen moduulitestausmenetelmät eivät olleet lainkaan riittävällä tasolla. Myös refactoring ja koodin yhteisomistus herättivät ohjelmoijissa huolestuneisuutta. Vaikka he myönsivätkin menetelmien hyvät puolet, he kokivat olevansa vastuussa omasta koodistaan. Heidän mielestään toisten ohjelmoijien tekemät muutokset koodiin eivät aina tuoneet parannuksia, vaan saattoivat tehdä koodista jopa heikkolaatuisempaa.

XP:n käyttöönotto suuryrityksessä toi esiin suurta muutosvastarintaa [Grenning, 2001]. Yritys tuotti turvallisuuden kannalta kriittisiä järjestelmiä perinteisin menetelmin, luottaen vesiputousmalliin ja tiukkoihin katselmointikäytäntöihin. Se ei kuitenkaan ollut tyytyväinen tuloksiinsa, sillä ohjelmistoissa esiintyi katselmoinneista huolimatta virheitä, projektien aikataulut venyivät ja prosessi oli hyvin raskas. Niinpä yritys päätti kokeilla olio-ohjelmointia, käyttötapauksia ja iteratiivisia menetelmiä, palkaten Grenningin viemään muutoksen läpi organisaatiossa. Grenning päätti kuitenkin mennä askeleen pitemmälle ja esitti, että yritys kokeilisi joitain XP:n menetelmiä.

Hän kuitenkin tiesi, että XP:n käyttöönotto sellaisenaan olisi herättänyt suurta vastustusta yrityksen organisaatiokulttuurin vuoksi. Hänen oli saatava ohjelmoijat ja varsinkin johto ymmärtämään ja tukemaan XP:n tapaa hallita projektia. Niinpä hän päätti soveltaa XP:n käytäntöjä. Vaatimusten määrittelyssä hän tuli hieman vastaan ja vaatimukset kirjattiin yrityksen prosessin mukaisina käyttötapauksina. Dokumentaation määrä oli keskeistä, olihan yrityksessä kirjoitettu perinteisesti runsaasti dokumentaatiota ja ehdotus ohjelmistosta täysin ilman paperidokumentaatiota olisi ollut mahdoton perustella. Niinpä hän teki kompromissin ja ehdotti suppeamman dokumentaatiopaketin tuottamista ylläpitoa varten. Hän myös käytti perusteluissa hyväkseen XP:n menetelmien toisiaan tukevia ominaisuuksia, tukien katselmointien puuttumista pariohjelmoinnilla, pariohjelmointia koodauskonventioilla ja niin edelleen. Hän onnistuikin saamaan vastaavan johtajan menetelmän taakse ja kehitystyö alkoi.

Ohjelmistossa oli 125 käyttötapausta. Aivan kehitystyön aluksi ryhmä ei uskaltanut tehdä ohjelmistoa ilman suunnittelua, vaan se teki joitain luokka- ja sekvenssikaavioita. Koska he tiesivät, että suunnitelmat todennäköisesti tulisivat muuttumaan, he eivät kuitenkaan dokumentoineet kaavioita sähköisesti. Ohjelmoitaessa he käyttivät pariohjelmointia ja huomasivat, etteivät oikeastaan tarvinneet koodauskonventioita, koska heidän ohjelmointityylinsä oli samanlainen. Heillä oli myös suorituskykyvaatimuksia ja he onnistuivat suunnittelumallien avulla tuottamaan skaalautuvan ohjelmiston. Projekti onnistui ja johto oli tyytyväinen. Sen alku kuitenkin osoitti, kuinka vaikeaa XP:n tuominen on organisaatioon, joka on pitkään käyttänyt perinteisiä menetelmiä. Tätä tukevat myös muut tutkimukset sekä Internetin foorumeilla käyty keskustelu.

6.2. Tutkimus useista XP-projekteista

Vaikka XP:stä on julkaistu useita yhtä projektia käsitteleviä case-tutkimuksia, sen käytöstä ei ole juurikaan olemassa yhtä projektia laajempia tutkimustuloksia. Kuitenkin Rumpe ja Schröder ovat kyselytutkimuksen avulla koonneet tilastotietoa useista XP-projekteista [Rumpe ja Schröder, 2002].

Kesällä 2001 pyydettiin ohjelmoijilta ja projektipäälliköiltä vastauksia kyselyyn useiden eri kanavien kautta. Tutkimus oli maailmanlaajuinen ja se toimi lähinnä Internetin välityksellä. Siinä oli 33 kysymystä, jotka jakautuivat kolmeen osioon. Ensimmäisen osion kysymykset käsittelivät XP:tä käyttävää organisaatiota ja itse projektia. Toisessa osassa kysyttiin XP:n soveltamistapoja projektissa ja projektin onnistumista. Kolmas osa keräsi käyttäjien mielipiteitä ja kartoitti kiinnostusta käyttää XP:tä tulevaisuudessa. Vastauksia kyselyyn saatiin 45 kappaletta, lähes kaikki ohjelmoijilta ja projektipäälliköiltä. Noin puolet kohteena olevista projekteista oli kyselyn aikana vielä kesken.

Tutkimuksen hätkähdyttävin tulos oli, että 45:stä projektista 44 oli onnistuneita, yksi osittain onnistunut. Epäonnistuneita ei siis ollut lainkaan. Verrattaessa tätä kohdassa 1.3.1 esitettyihin Standish Groupin tutkimustuloksiin kaikista ohjelmistoprojekteista, erot ovat todella suuret. Tutkimuksen mukaan myös täydet 100 % vastaajista käyttäisi XP:tä uudelleen, jos sopiva tilaisuus tulisi.

Noin 70 % tutkimuksessa mukana olleista projekteista oli toteutettu Java-kielellä. Tämä voidaan tulkita siten, että uusien teknologioiden käyttäjät kokeilevat mielellään myös uusia projektinhallintamenetelmiä. Merkille pantavaa oli, että noin 60 % kyselyyn vastanneista oli Euroopasta, vaikkei Euroopan osuus maailman ohjelmistoliiketoiminnasta ole vastaavaa luokkaa. Mielenkiintoinen yksityiskohta oli myös se, että henkilökohtaisesti kyselyyn pyydetyistä XP-käyttäjistä peräti 78 % ei osallistunut kyselyyn, sillä he eivät

olisi saaneet virallisesti käyttää XP:tä projekteissaan, eivätkä halunneet XP:n käytön tulevan ilmi. Tästä voidaan päätellä, että organisaatioiden muutosvastarinta on edelleen voimakasta.

Kyselyn tuloksia voidaan kuitenkin pitää vain suuntaa antavina. Otos on pieni eikä kovin edustava. Lisäksi voidaan epäillä, että kyselyyn on vastattu herkemmin, jos XP-projektissa on onnistuttu. Puolet projekteista oli myös kesken, joten niiden onnistumisen arviointi on ennenaikaista. Merkittävin kysymys kuitenkin on projektin onnistumisen mittaaminen. Standish Groupin tutkimuksessa kyselyyn vastasivat asiakkaan ja johdon edustajat, joten olisi mielenkiintoista tietää, olisivatko näiden XP-projektien asiakkaat ja johtajat samaa mieltä projektien onnistumisesta kuin niiden tekijät.

6.3. XP:tä kohtaan esitettyä kritiikkiä

Extreme Programming on herättänyt paljon keskustelua alan lehdistössä ja Internetin foorumeilla. Suhtautuminen XP:hen ja muihin joustaviin menetelmiin tuntuu olevan kahtiajakoista: osa pitää XP:tä erinomaisena menetelmänä, osa ei näe siinä juuri mitään hyvää.

XP:stä on kirjoitettu myös joitain siihen kriittisesti suhtautuvia artikkeleita. Niiden laatu vaihtelee, sillä joistain huomaa etteivät niiden kirjoittajat ole huomioineet XP:n rajoituksia. Tällaisissa artikkeleissa XP:tä kritisoidaan sen huonosta soveltumisesta esimerkiksi suuriin ja maantieteellisesti hajautettuihin projekteihin, vaikka sellaisiin sitä ei ole tarkoitettukaan. Kriittisissä artikkeleissa on esitetty myös hyviä argumentteja, joita esitetään seuraavassa.

Ilman selkeää aikataulua toiminnan suunnittelu ja johtaminen voivat vaikeutua. Tämä koskee sekä asiakasta että ohjelmiston toimittajaa. Ilman selkeää projektin hintaa myös budjetointi on vaikeaa. Sama ongelma tosin tulee vastaan kaikissa projekteissa, jotka ylittävät aikataulunsa ja budjettinsa. Projektin lopullinen kustannus tarkentuu usein vasta projektin aikana.

XP luottaa vahvasti palautteeseen toiminnan kehittämisen muotona. Se ei kuitenkaan millään tavoin määrittele, että projektissa pitäisi mitata ja seurata esimerkiksi työmääräarvioiden toteutumista. Tosin mittaamista ei mitenkään kielletä, mutta sitä ei vain ole otettu metodivalikoimaan mukaan eikä siitä anneta mitään ohjeita. Kohdassa 4.5 esitetyssä XP:n roolijaossa mittaaja on kuitenkin mukana.

Kommunikaatio on XP:n perusarvoja. Projektiryhmän kommunikaatio sisäisesti ja asiakkaan suuntaan on toteutettu hyvin, mutta kommunikaatiota johdon suuntaan ei ole toteutettu millään tavalla. Johto kuitenkin viime kädessä hyväksyy käytettävät työtavat ja budjetit. Johdon vastuulle roolijaossa on annettu projektin esteiden poistaminen. Usein johto haluaa kuitenkin osallistua projektiin tarkemmin ja saada siitä enemmän tietoa.

Ohjelmoijat ja projektipäälliköt tuntuvat suhtautuvan myönteisimmin XP-menetelmään. Kriittisintä suhtautuminen on asiakkaiden ja johdon keskuudessa. Suhtautumista selittää se, että XP:n työtavat tuovatkin projektiryhmälle muihin menetelmiin verrattuna paljon lisäarvoa, mutta vaatii asiakkaalta ja johdolta enemmän rohkeutta ja resursseja. Asiakkaat ja johto kuitenkin odottavat saavansa tämän lisäarvon kaikkien muidenkin menetelmien käytön kautta, vaikkei niin käytännössä kävisikään.

6.4. Extreme Programming ja ohjelmistoprojektien sopimusmallit

Koska asiakkaalle annetaan Extreme Programming -projektissa valtaa vaikuttaa vaatimuksiin ja ohjata projektia, tarvitsevat joustavat projektit myös erilaisen sopimusmallin asiakkaan ja toimittajan välillä kuin perinteiset projektimallit. Perinteinen sopimusmalli on yleensä sisältänyt kiinnitettynä kolme luvussa 1 mainituista neljästä ohjelmistoprojektin muuttujasta eli ohjelmiston laajuuden, hinnan ja toimitusajan. Laatutavoitteista niissä harvoin puhutaan, tilaaja odottaa usein saavansa parasta mahdollista laatua. Käytännössä kuitenkin tapahtuu liian usein niin, että saavuttaakseen kolme sopimuksessa mainittua muuttujaa ohjelmiston toimittaja joutuu tinkimään laadusta. Tämä näyttää hyvältä paperilla, mutta ei pidemmän päälle ole kestävä ratkaisu.

Joustavien projektien sopimuksissa kirjataan sopimukseen muuttujista yleensä kaksi: projektin pituus ja hinta. Hinta saadaan projektiin osallistuvien henkilöiden määrästä, joiden työstä laskutetaan tiettyä tunti-, päivä- tai kuukausilaskutushintaa. Sopimuksen pituus ei välttämättä ole projektin loppumispäivä, vaan useimmiten se ajankohta, jolloin sopimusta tarkistetaan seuraavan kerran. Sen sijaan projektin laajuutta ei määritellä sopimuksessa, vaan asiakas saa antaa ryhmälle haluamansa käyttötapaukset sopimuksessa mainittuna aikana ja muuttaa niiden priorisointia vapaasti. Joustavat sopimukset ovatkin lähempänä henkilöstövuokrauksen toimintamallia, kun taas perinteiset sopimukset pitäytyvät tiukasti asiakas-toimittaja -mallissa. Niitä on myös kritisoitu siitä, että ne siirtävät riskin projektin onnistumisesta kokonaan ostajalle.

Toisaalta ohjelmiston toimittajalla on myös mahdollisuus sitoutua kiinteähintaiseen sopimukseen ja toteuttaa silti projekti XP-menetelmän mukaisesti. Tämä kuitenkin tuo riskin toimittajalle, mutta mikäli riski on hallittu ja toimittaja onnistuu projekteissaan pitkällä tähtäimellä, se voi saada lisäarvoa XP:n menetelmien käytöstä. Toisaalta XP on niin nuori menetelmä, ettei yhdelläkään toimittajalla voi olla siitä pitkää kokemusta, mikä luonnollisesti vähentää riskinottohalukkuutta.

Nykyisten ohjelmistotoimitusten sopimusmalleina usein toimivat IT2000 [IT2000, 2000] ja Valtion tietotekniikkahankintojen yleiset sopimusehdot [VYSE

1998, 1998] eivät sovellu kovin hyvin joustavien sopimusten tekoon. Erityisesti valtion sopimusehdot ovat hyvin tarkasti määritellyt ja pyrkivät varmistamaan tilaajan edut mahdollisimman hyvin. IT2000 on kehitetty puolueettomaksi sopimusmalliksi yrityksille ja tarjoaa hieman enemmän mahdollisuuksia joustavuuteen. Esimerkiksi ohjelmiston toimitus osissa on mahdollista IT2000:ssa.

Julkisen alan ohjelmistoprojektit tulee lain mukaan kilpailuttaa ja julkisten organisaatioiden tulee valita kokonaistaloudellisesti paras vaihtoehto. Tämä on osaltaan johtanut siihen, että varsin usein julkiset projektit ovat kaksivaiheisia. Ensimmäisessä vaiheessa kilpailutetaan määrittelyprojekti, jossa rakennetaan tarkka määrittely. Toisessa vaiheessa käydään tarjouskilpailu varsinaisen ohjelmiston rakentamisesta määrittelyjen perusteella. Vaikka melko usein määrittelijänä toiminut organisaatio voittaa kilpailun myös varsinaisesta järjestelmän rakentamisesta, on mallista helppo huomata, ettei se sovellu joustavien menetelmien käyttöön.

7. Yhteenveto

Edellä mainittujen seikkojen ja lähteiden perusteella on lopuksi koottu yhteen projektimalleja, joihin Extreme Programming sopii. Tämän jälkeen on listattu malleja, joihin Extreme Programming sopii huonosti tai ei ollenkaan. Listat eivät suinkaan ole täydellisiä, ainoastaan suuntaa antavia. Lopuksi on esitetty vielä teknisiä reunaehtoja sekä muita tutkimuksen perusteella saatuja johtopäätöksiä.

7.1. Projektimalleja, joihin Extreme Programming sopii

Erityisen hyvin Extreme Programming sopii projekteihin, joissa vaatimukset ovat epäselviä tai niiden uskotaan muuttuvan projektin aikana. XP tuntuu keskittyvän nimenomaan määrittelyissä tapahtuvien muutosten negatiivisten vaikutusten eliminoimiseen. Kohdassa 1.3.2. esitettiin, että suurimmat ongelmat projekteissa liittyvät nimenomaan joko määrittelyjen puutteellisuuteen tai niiden muuttumiseen. Beck totesi kirjassaan [Beck, 1999]: ”(Projektien) ongelmana eivät ole vaatimusten muuttuminen vaan se, etteivät käytetyt projektimallit varaudu vaatimusten muutoksiin”. Useat iteraatiot, vastuunjako toiminnallisten ja teknisten vaatimusten välillä sekä asiakkaan vaikutus projektin ohjaukseen tukevat tätä varautumista.

Toinen hyvä käyttökohde ovat projektit, jotka tehdään uudelle tai vaikealle sovellusalueelle, jota projektihenkilöstö ei täysin ymmärrä. Tiivis yhteistyö asiakkaan kanssa lisää ymmärrystä alueesta ja virheet huomataan iteraatioilla nopeasti.

Myös projektit, jotka tehdään uudella ja vielä kehittyvällä teknologialla, ovat hyvä kohde XP-menetelmän käytölle. Nämä muutokset on hyvä hallita joustavasti ja pariohjelmointi, tiivis kommunikaatio ja koodin yhteisomistus levittävät uutta tietoa projektiryhmän sisällä tehokkaasti. Uusi teknologia tuottaa myös helposti yllätyksiä, joiden vuoksi kaikkia vaatimuksia ei voida toteuttaa ja projektissa joudutaan vaihtamaan suuntaa. Tällöin vähäisempi dokumentaatio projektin alussa säästää kustannuksia.

Edellisten lisäksi XP:n käyttöä on hyvä harkita projekteissa, joissa perustoiminnallisuus tulee saada käyttöön hyvin nopeasti. Nopea iteraatioiden sykli tukee tätä erinomaisesti. Tämä tosin edellyttää, että ohjelmiston toiminnallisuus on jaettavissa erikseen toimitettavissa oleviin osiin.

7.2. Projektimalleja, joihin Extreme Programming ei sovi

Sellaisten ohjelmistojen rakentaminen, joiden päivittäminen käyttöönoton jälkeen on vaikeaa tai mahdotonta, ovat luonnollisesti huono käyttökohde iteraatioita käyttävälle XP:lle. Toisaalta teknisesti helposti päivitettäviä

ohjelmistoja tuottavat projektit, kuten esimerkiksi sovelluspalvelinpohjaiset selainkäyttöliittymällä toimivat ohjelmistot, voivat olla erinomaisia XP:n käytön kohteita.

Projektit, joissa asiakkaan on tarkasti tiedettävä siihen kuluvat aika- ja kustannusresurssit esimerkiksi budjetoinnin takia, eivät sovi XP-projekteiksi. XP:ssä lopullinen resurssitarve tiedetään vasta ohjelmiston valmistuttua.

XP:n käyttöä kannattaa välttää myös sellaisissa projekteissa, joita aloitettaessa on syytä epäillä, ettei asiakkaalla ole riittävästi aikaa tai halua osallistua projektiin. Tämä tietenkin vaikeuttaa yhteistyötä kaikissa projektimalleissa. Periaatteessa mitään ohjelmistoprojektia kannata aloittaa ennen kuin on varmistuttu, että asiakkaalta saadaan projektin onnistumisen kannalta riittävät resurssit.

Erityisen korkeaa luotettavuutta vaativat projektit eivät artikkelin [Turk, 2002] sovi toteutettavaksi XP:llä. Extreme Programming ei tarjoa esimerkiksi muodollisia koodikatselmointeja tai tapoja mitata testikattavuutta, jotka ovat kaikkein kriittisimmissä järjestelmissä usein vaatimuksina. Myöskään ei-toiminnallisten ominaisuuksien määrittelyyn ei ole XP:ssä hyviä keinoja.

Jotkut sopimusmallit, esimerkiksi valtion tietotekniikkahankintojen yleiset sopimusehdot, eivät mahdollista XP:n käyttöä. Samoin asiakasorganisaatioiden tapa kilpailuttaa määrittely- ja toteutusprojektit erikseen estää XP:n käytön. XP toimii parhaiten joustavan sopimusmenettelyn kanssa, kuten muutkin joustavat projektinhallintamenetelmät.

Beckin [1999] mukaan myös projektit, joissa henkilöstö on jakautunut maantieteellisesti, on parempi viedä läpi jotain toista projektinhallintamenetelmää käyttäen. Tällöin henkilökohtainen kommunikointi, pariohjelmointi ja koodin yhteisomistus vaikeutuvat. Tämä usein sulkee pois myös projektit, joissa käytetään alihankintaa suorittamaan osaa ohjelmointityöstä.

Suurimpana esteenä XP:n käytölle Beck kuitenkin mainitsee organisaatiot, joilta puuttuu tarve tai rohkeus kokeilla uusia menetelmiä. Sananlaskun mukaan ehjää ei ole tarpeen korjata, joten XP:hen tulee siirtyä vain, jos prosessien laatu ei organisaatiossa tyydytä. Toisaalta vaikka parannettavaa prosesseissa olisikin, organisaatiokulttuurit saattavat olla kykenemättömiä XP:n vaatimiin muutoksiin.

Suuret, yli kahdenkymmenen hengen projektit suositellaan Beckin kirjassa tehtäväksi muilla menetelmillä kuin XP:llä. Beckin mukaan työn tuottavuus laskee liiaksi XP:tä käytettäessä projektia varten tarvittavan kommunikaation takia. Kommunikaation tuottama lisätyön tarve koskee kaikkia projektimalleja, mutta XP:ssä on sen lisäksi saatava päivittäiseen integraatioon mukaan kaikki

tuotettu koodi, mikä saattaa muodostaa prosessiin pullonkaulan. Tämän kokoluokan projektit jaetaan usein aliprojekteihin, joten aliprojektin läpivienti XP:n menetelmillä ei ole poissuljettua. XP:n periaatteita on myös sovellettu onnistuneesti suurissakin projekteissa, vaikka varsinaista XP-prosessia ei olekaan noudatettu [Taber, 2000].

Toisaalta myös pienet projektit, joissa on alle neljä henkilöä ovat huonoja XP:n käyttökohteita. Yhden hengen projekteissa pariohjelmointi ei ole mahdollista, kahden hengen projekteissa siitä tuskin saadaan vastaavaa hyötyä ja kolmen hengen projekteissa voidaan muodostaa vain yksi pari kerrallaan.

Henkilöstön huono yhteishenki ja varsinaisen ryhmähengen puuttuminen on vakava uhka jokaiselle projektille. XP-projekteissa tämä uhka korostuu entisestään, koska ohjelmakoodi tuotetaan pariohjelmointina ja tieto välitetään henkilökohtaisesti, ei dokumenttien muodossa. Näin ollen henkilöstön vaihtuessa tietoa saattaa hävitä ja sen uudelleen saaminen voi olla vaikeaa.

Edellä mainituista seikoista voidaan huomata, että XP:n käyttöä rajoittavia seikkoja on varsin runsaasti ja ne rajaavat pois luultavasti suurimman osan tämän päivän ohjelmistoprojekteista. Monissa projekteissa, joihin XP sopisi kohdan 7.1 perusteella, voi olla seikkoja, joiden perusteella ne eivät kuitenkaan sovi XP-projekteiksi. Siitä huolimatta mahdollisia XP–projekteja on varmasti olemassa huomattava määrä.

7.3. Teknisiä reunaehtoja

Edellä ei ole juurikaan otettu kantaa siihen, asettavatko erilaiset teknologiat rajoituksia XP:n käytölle. Tutkimuksia asiasta ei ole vielä juuri käytettävissä. Robert Martin pohti artikkelissaan [Martin, 2000] XP:n käyttöä C++-projekteissa. Hän lähti oletuksesta, että C++:n vahva staattinen tyypitys ja #include-lauseet tekevät siitä huomattavasti vaikeampaa refaktoroida kuin Javasta. Hän kuitenkin päätyy tulokseen, jossa XP sopii myös C++-projekteihin, kunhan koodin joustavuuteen kiinnitetään jatkuvaa huomiota.

Sen sijaan sellaiset projektit, jotka toteutetaan jollain muulla ohjelmointikielellä kuin oliokielellä, eivät sovi XP-projekteiksi. Refactoring-menetelmä on erittäin oliokeskeinen, eikä sitä käytännössä voi soveltaa muihin ohjelmointiparadigmoihin.

Internetin XP-foorumeilla on keskusteltu ahkerasti XP:n soveltamista ohjelmistoihin, joissa tietokannalla on merkittävä osuus. Yleisen ajattelutavan mukaisesti tietokanta tulee voida määritellä ensin valmiiksi, jotta sen “päälle” on helppo rakentaa varsinaista ohjelmistoa. Toisaalta muutos tietokannassa vaikuttaa aina ainakin perinteisen kolmikerrosarkkitehtuurin tietovarastoja käsittelevään koodiin. Näin ollen XP:n jatkuva muutos ja refaktorointi tuovat tietokantojen käyttöön ristiriidan. Viime aikoina joustavien menetelmien

yhteisössä on kuitenkin kirjoitettu alustavia suosituksia siitä, miten tämä ongelma voidaan ratkaista. Näidenkään suositusten tehosta ei kuitenkaan vielä ole tuloksia.

7.4. Loppupäätelmät

Luettuani Beckin [1999] kirjan XP:stä, näin sen erittäin hyvänä ratkaisuna moniin ongelmiin, joita olin kohdannut työssäni ohjelmistoyrityksen projektipäällikkönä. Kirja oli mielenkiintoisesti kirjoitettu ja lähestymistavaltaan jopa empaattinen, sillä se alkoi esittämällä projektityön suurimpia ongelmia ja XP:n esittämän yksityiskohtaisen ratkaisun. Tämä oli lähtökohtana myös tämän tutkimuksen kirjoittamiselle.

Aiheeseen tutustuttuani asenteeni on osin muuttunut. XP:ssä on tutkitusti hyviä käytäntöjä, mutta tutkimukset ovat myös osoittaneet, että XP kokonaisuutena voi toimia vain osassa projekteista. Niissä projekteissa joissa XP:tä on mahdollisuus käyttää, ovat onnistumisen edellytykset muutenkin keskimääräistä korkeammalla hyvän johdon tuen ja asiakkaan asiantuntemuksen ansiosta. Kokonaisuutena mielikuva XP:stä on kuitenkin edelleen positiivinen, sillä se tarjoaa hyviä käytäntöjä, joita voi hyväksikäyttää kaikissa ohjelmistoprojekteissa, vaikka ne eivät XP:tä kokonaisuudessaan käyttäisikään.

Extreme Programming on saanut aikaan paljon keskustelua. Tämä osaltaan antaa viitteen siitä, että alan prosessit eivät ole valmiita ja uusista toimintatavoista ollaan kiinnostuneita. Monet XP-metodologian menetelmistä näyttävät toimivilta myös tutkimusten valossa. Toisaalta XP:n käyttöä kokonaisuutena suurissa ohjelmistoprojekteissa on tutkittu vielä varsin vähän, eikä riittävän laajoja näyttöjä sen toimivuudesta ole vielä saatu. Toisaalta mikään ei myöskään viittaa siihen, että XP olisi huono menetelmä, vaan tähän asti esitettyjen tulosten voidaan sanoa olevan rohkaisevia.

Tietojenkäsittelytieteessä kuitenkin tiedetään, ettei mikään yksittäinen teknologia tai menetelmä tule ratkaisemaan kaikkia ohjelmistoprojektien ongelmia, vaikka niiden tullessa laajempaan tietoisuuteen niihin kohdistuu suurta huomiota. Näin asia on myös XP:n kanssa: siitä ei tule hopealuotia kaikkien ohjelmistoprojektien ongelmien ratkaisuun, mutta se saattaa tarjota hyvän työkalun projektipäälliköille niihin projekteihin, joihin se parhaiten sopii.

Viiteluettelo

[Abrahamsson *et al*, 2002] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen and Juhani Warsta, Agile development methods. Review and analysis. VTT Publications, 2002.

[Agile Alliance] Agile Alliance -organisaatio. http://www.agilealliance.org/.

[Agile Manifesto] Agile Manifeston periaatteet. http://agilemanifesto.org/.

[Ant] Ant 1.5.1. Apache Jakarta -projekti. http://jakarta.apache.org/ant/.

[Beck, 1998] Kent Beck, Extreme Programming: A Humanistic Discipline of Software Development. *Lecture Notes in Computer Science.* 1382, 1-16.

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

[Beck ja Fowler, 2000] Kent Beck and Martin Fowler, *Planning Extreme Programming.* Addison-Wesley, 2000.

[Boehm, 1981] Barry W. Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

[Brownsword ja McUmber, 1991] Lisa Brownsword and Rick McUmber, *Proceedings of the Conference on TRI-Ada '91: Today's Accomplishments; Tomorrow's Expectation*. 378 – 386.

[Cockburn, 2001] Alistair Cockburn, Agile software development, the people factor. *IEEE Computer, November 2001 131-133.*

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

[Cockburn 2002b] Alistair Cockburn, Learning from agile software development, part 1. *Crosstalk, October 2002.*

[Cockburn 2002c] Alistair Cockburn, Learning from agile software development, part 2. *Crosstalk, November 2002.*

[Cockburn *et al*, 2000] Alistair Cockburn, Laurie Williams. The costs and benefits of pair programming. *XP 2000.*

[Coad *et al*, 2000] Peter Coad, Eric Lefebvre and Jeff De Luca, *Java Modeling In Color With UML: Enterprise Components and Process,* Prentice Hall 1999.

[Emery, 2002] Patrick Emery, The dangers of Extreme Programming, 2002. http://members.cox.net/cobbler/XPDangers.htm.

[Fowler, 1999] Martin Fowler, *Refactoring.* Addison-Wesley, 1999.

[Gittins *et al*, 2001] Robert Gittins, Sian Hope and Ifor Williams, Qualitative Studies of XP in Medium Sized Business. *XP 2001.*

[Grenning, 2001] James Grenning, Using XP in a Big Process Company. *XP Universe 2001.*

[Haikala ja Marijärvi, 1998] Ilkka Haikala ja Jukka Marijärvi. *Ohjelmistotuotanto.* Suomen ATK-kustannus Oy, 1998.

[Highsmith, 1997] Jim Highsmith, Messy, exciting and anxiety-ridden – adaptive software development. *American Programmer, Volume X, No. 1; January 1997.*

[Highsmith, 2000] Jim Highsmith, Retiring lifecycle dinosaurs. *Software Testing and Quality Engineering, July/August 2000*. 22-28.

[IT2000, 2000] Keskuskauppakamari, Logistiikkayhdistys ry, Tietotekniikan liitto ry, TIPAL ry, IT2000 Tietotekniikka-alan sopimusehdot, Talentum 2000.

[Jacobson *et al*., 1995]. I. Jacobson, M. Ericsson and A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technology,* Addison-Wesley, 1995.

[Jensen, 2003] Randall W. Jensen. A pair programming experience. Crosstalk, March 2003.

[JTest] JTest-testausohjelmisto. http://www.parasoft.com/jsp/products/ home.jsp?product=Jtest

[JUnit] JUnit-testausohjelmisto. http://www.junit.org

[Keefer, 2002] Gerold Keefer, Extreme Programming Considered Harmful for Reliable Software Development. *StickyMinds.com*, 2002.

[Martin, 2000] Robert C. Martin, Can XP be used with C++? *Objectmentor.com*, 2000.

[McConnell, 1993] Steve McConnell, *Code Complete*. Microsoft Press, 1993.

[McDowell *et al*., 2002] Charlie McDowell, Linda Werner, Heather Bullock and Julian Fernald, The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin 34,* 1 (2002), 38-42.

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

[Pressman, 2000] R.S. Pressman, *Software Engineering: A Practitioner' s Approach.* McGraw-Hill, 2000.

[Rising ja Janoff, 2000] Linda Rising and Norman S. Janoff, *The Scrum Software Development Process for Small Teams*. IEEE Software, 2000.

[Rumpe ja Schröder, 2002] Bernhard Rumbe ja Astrid Schröder, Quantative Survey on Extreme Programming Projects. *XP 2002.*

[SFS, 1981] Suomen Standardoimisliitto, *Projektitoiminta, toimintaverkkosanasto ja piirrosmerkit*. Suomen Standardoimisliitto, 1981.

[Schwaber, 1996] Ken Schwaber, The origins of SCRUM, *OOPSLA 1996.*

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

[Scrum] Scrum-metodologian verkkosivusto www.controlchaos.com

[Sol, 1983] H. G. Sol, A Feature Analysis of Information Systems Design Methodologies: Methodological Considerations. *Information systems design methodologies: a feature analysis*. Elseview 1983.

[Song ja Osterweil, 1991] Leon J. Osterweil ja Xiping Song, Comparing design methodologies through process modeling. *1st International Conference on Software Process*. IEEE CS Press, 1991.

[Song ja Osterweil, 1992] Leon J. Osterweil ja Xiping Song, Towards objective, systematic design-method comparisons. *IEEE Software*, (1992).

[Standish Group, 1995] Standish Group Ltd, *Chaos Report*. Standish Group, 1995. Also available http://www.standishgroup.com/sample_research/index.php.

[Standish Group, 1998] Standish Group Ltd, *Chaos: A Recipe for Success*. Standish Group, 1998. Also available http://www.standishgroup.com/sample_research/chaos1998.pdf.

[Stevens, 1981] Wayne Stevens, *Using Structured Design*. John Wiley, 1981.

[Taber, 2000] Cara Taber, Martin Fowler, An iteration in the life of an XP project. Cutter IT journal, November 2000.

[Turk, 2002] Dan Turk, Limitations of Agile Software Processes, *XP 2002*.

[VYSE 1998, 1998] Valtioneuvosto, Valtion tietotekniikkahankintojen yleiset sopimusehdot, 1998.

[Walton, 2002] Bill Walton, XP, that dog don't hunt. *StickyMinds.com*, 2002.

[Williams *et al*., 2000] Laurie Williams, Robert Kessler, Ward Cunningham and Ron Jeffries, Strengthening the Case for Pair-Programming. *IEEE Software*, (2000).

[Williams ja Upchurch, 2001] Laurie Williams and Richard L. Upchurch, *In support of student pair-programming.* ACM SIGCSE Bulletin 33, 1 (2001), 327 - 331.

[Womack, 1990] James P. Womack, *The machine that changed the world.* Rawson, 1990.