Kimmo Peltoniemi

XP VAIHTOEHTONA PERINTEISILLE OHJELMISTOPRO-SESSIMALLEILLE

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

26.2.2009

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Kimmo Peltoniemi

Yhteystiedot: kimmo.m.peltoniemi@jyu.fi

Työn nimi: XP vaihtoehtona perinteisille ohjelmistoprosessimalleille

Title in English: XP as an alternative process model to traditional process models

Työ: Pro gradu -tutkielma

Sivumäärä: 134

Linja: Ohjelmistotekniikka

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: Prosessimalli, ohjelmistoprosessi, XP, Extreme Programming, ketterät menetelmät, perinteiset menetelmät, vesiputousmalli

Keywords: Process model, software process, XP, Extreme Programming, agile methods, traditional methods, waterfall model

Tiivistelmä: Tutkielmassa tarkastellaan ohjelmistoteollisuuden tilaa, XP-prosessimallin vaikutusta ohjelmistotuotantoon ja perehdytään pintapuolisesti kahteen erilaiseen ohjelmistotuotannon prosessimallien perheeseen, perinteisiin ja joustaviin prosessimalleihin. Tutkielmassa verrataan yleisimpiä käytössä olevia perinteisiä ja ketteriä prosessimalleja toisiinsa, niiden eroavaisuuksia ja soveltuvuuksia erilaisiin projekteihin. Tutkielmassa käsitellään myös XP:n ja pariohjelmoinnin soveltamisesta saatuja tutkimustuloksia.

Abstract: This thesis examines state of the software industry, what kind of impact XP-process model has in the software production. Thesis also examines two different kinds of process model families, traditional and agile methods, their differences and suitability to different kind of software projects. In this thesis we also examine study results from XP and pair programming.

Esipuhe

Kiitokset niille tahoille, jotka ovat mahdollistaneet opiskelun suorittamisen Chydenius-instituutissa viimeisten vuosien aikana. Ja niin, että se tapahtui kenenkään kiinnittämättä siihen koskaan mitään huomiota.

Vilpittömät pahoitteluni niille luennoitsijoille ja kanssaopiskelijoille, joita allekirjoittaneen luennoilla kuorsaaminen ja torkkuminen on saattanut häiritä. Mutta kuten tunnettu sanonta kuuluu, ”avaruudessa kukaan ei kuule huutoasi, mutta Chydiksen luentosalissa kaikki kuulevat kuorsauksesi”.

Tämän tutkielman teossa pyrittiin käyttämään perinteisen vesiputousmallin kaltaista toimintatapaa. Mallin periaatteiden mukaisesti seuraavaan vaiheeseen eli tutkielman lukuun ei siirrytty ennen kuin edellinen oli saatu täysin valmiiksi. Ajan mittaan tämä osoittautui perin huonoksi ja paljon aikaa vieneeksi valinnaksi, kaikkien huonojen valintojen äidiksi. Lopulta tutkielman valmistumiseen kuluikin aikaa yli puolitoista vuotta.

Mallin käyttäminen tässä yhteydessä aiheutti sen, että tutkielman sivumäärä ja käytettyjen lähteiden lukumäärä kasvoi jopa yli kaikkien karmeimpien ennakkosuunnitelmien. Niinpä ensimmäiset valmiiksi saadut versiot muistuttivatkin enemmän kierrätykseen heitettyä Hesburgerin sillisalaattia kuin edes rimaa hipoen hyväksyttävän tason saavuttavaa tutkielmaa. Professori Ismo Hakalan ja FM Veli-Matti Tornikosken suosiollisella, sitkeällä ja kärsivällisellä ohjauksella se loppujen lopuksi saavutti julkaistavaksi kelpaavan tason.

Kokkolassa 25.02.2009

Kimmo Peltoniemi

Termiluettelo

ACM	Association for Computing Machinery.
Agile Manifesto	Ketterän ohjelmistokehityksen manifesti
ASD	Adaptive Software Development
DSDM	Dynamic Systems Development Method
FDD	Feature-Driven Development
IEEE	Institute of Electrical and Electronics Engineers
Inkrementaalinen	Kumuloituva, lisääntyvä
IRA	Individual Retirement Account, USA: n veroviranomainen
Iteratiivinen	Vaiheittainen, jaksoittainen
Java	Sun Microsystemsin kehittämä ohjelmistoalusta
JUnit	Javalle suunniteltu testauskehys
RUP	Rational Unified Process
XML	eXtensible Markup Language, merkintäkieli
XP	Extreme Programming
VTT	Valtion teknillinen tutkimuskeskus

Sisältö

1 Johdanto

Ohjelmistoista on tullut tekniikan ja tutkimuksen kehitystä eteenpäin ajava voima, vaikuttaen samalla tärkeiden päätösten tekoon. Ohjelmistot erottavat nykyaikaiset tuotteet ja palvelut vanhanaikaisista. Niitä käytetään kaikkialla ja tulevaisuudessa yhä enemmän, ollen siten oleellinen osa ihmisten jokapäiväistä elämää. Ohjelmistotuotteiden jatkuvasti suureneva koko ja lisääntyvä monimutkaisuus aiheuttavat sen, että niitä tekevien ihmisten lukumäärä kasvaa. Kokemuksesta kuitenkin tiedetään, että ohjelmistoprojektin henkilömäärän kasvaessa projektiryhmän tuottavuus heikkenee. Tutkimusten mukaan ohjelmistoprojektien onnistumistodennäköisyys ei ole kovin hyvä, suuren osan myöhästyessä, ylittäen aikataulunsa ja budjettinsa ja pahimmassa tapauksessa ne epäonnistuvat kokonaan.

Ohjelmistoja suunnittelevissa ja toteuttavissa yrityksissä on yleensä käytössä jokin prosessimalli ohjelmistojen tuottamiseksi. Se toimii kehyksenä niille toimille, joita vaaditaan ohjelmiston tuottamiseksi ja määrittelee lähestymistavan, jota ohjelmiston tuottamisessa käytetään. Yrityksissä on havaittu hyvin määritellyn, dokumentoidun ja toistettavissa olevan prosessin noudattamisen tuottavan järjestelmällisesti parempia tuloksia pienemmillä kustannuksilla. Siten prosessimallin käyttö ohjelmistotuotannossa on minkä tahansa ohjelmistotalon menestykselle elintärkeää, sillä ohjelmistokehitykselle asetetaan kovia paineita kustannusten ja aikataulujen osalta.

Erään määritelmän mukaan ohjelmistotuotantoprosessi määrittelee kuka tekee mitä, koska ja kuinka. Erilaisia prosessimalleja on runsaasti erilaisiin tarpeisiin, erikokoisille työryhmille ja erilaisiin projekteihin. Kaikki mallit eivät kuitenkaan sovellu kaikenlaisille organisaatioille tai kaikkiin projekteihin ja sen vuoksi on tärkeää kyetä valitsemaan oikein mitoitettu prosessimalli. Tiukkaan suunnitelmallisuuteen perustuvat perinteiset prosessimallit on havaittu monissa tapauksissa liian jäykiksi ja kykenemättömiksi vastaamaan asiakkaan nopeasti muuttuviin vaatimuksiin. Näin on syntynyt tarve uusille ja entistä joustavammille prosessimalleille, jotka pyrkivät eliminoimaan ohjelmistokehityksen riskejä.

Ratkaisuksi tällaisiin ongelmiin on lähdetty kehittämään perinteisiä prosessimalleja ketterämpiä malleja, jotka pyrkivät tuomaan kaivattua joustavuutta ohjelmistokehitykseen. Ketterissä prosessimalleissa nähdään toimivan ja laadukkaan ohjelmiston olevan tärkeämpää kuin kattavan dokumentaation ja niiden tavoitteena on lyhyin väliajoin tuottaa asiakkaalle toimivia versioita kehitettävästä ohjelmistosta, pyritään hallitsemaan muuttuvia vaatimuksia myös myöhäisessä projektin vaiheessa, sekä painotetaan yhteistyötä asiakkaan kanssa.

Internetiin liittyvä ohjelmistokehitys poikkeaa perinteisestä ohjelmistokehityksestä. Sille on ominaista, että sovellusten toteuttamiseen käytettävissä oleva aika on lyhyempi kuin koskaan aikaisemmin, sovelluksille asetettavat vaatimukset muuttuvat nopeassa tahdissa ja tuotteen saaminen ajoissa markkinoille on kriittistä. Samalla tavoin voidaan myös ajatella mobiililaitteisiin liittyvän ohjelmistokehityksen poikkeavan perinteisestä ohjelmistokehityksestä. Mobiililaitteiden ja niihin sisältyvien sovellusten elinkaari on nopeasti lyhentynyt, eikä sovelluskehitystä voida enää toteuttaa kovin pitkällä aikajänteellä.

Tutkielma tarkastelee XP-prosessimallin vaikutusta ohjelmistotuotantoon ja perehtyy pintapuolisesti kahteen erilaiseen ohjelmistotuotannon prosessimallien menetelmäperheeseen, perinteisiin ja ketteriin menetelmiin. Tutkielmassa käydään läpi lyhyesti yleisimpiä käytössä olevia perinteisiä ja ketteriä prosessimalleja, sekä tarkastelee niiden eroavaisuuksia ja soveltuvuuksia erilaisiin projekteihin. Tarkemmin esitellään perinteinen ja edelleen laajasti käytössä oleva vesiputousmalli sekä uudempi XP-prosessimalli. Nämä mallit ovat toimineet osaltaan esikuvina muille perinteisille ja ketterille prosessimalleille.

Ohjelmistoprosessimalleja voidaan arvioida sen perusteella, kuinka joustavasti muutostenhallinta voidaan saada sujumaan, kuinka usein ohjelmistoa testataan ja kuinka usein tai nopeasti ohjelmiston toimivia osia voidaan toimittaa asiakkaalle. XP eroaa merkittävästi vesiputousmallista, koska näiden mallien perusoletukset ohjelmistoprojektin toimintaympäristöstä poikkeavat toisistaan suuresti. Vesiputousmallissa ohjelmiston määrittelyjen ja teknologian oletetaan pysyvän samana koko ajan, kun vastaavasti XP olettaa toimintaympäristön olevan epästabiili ja siksi sen muutosprosessi on tehty kevyeksi.

Tutkielman luvun 2 tarkoituksena on toimia pohjana myöhemmin käsiteltäville aiheille. Luvun alussa esitellään lyhyesti ohjelmistotuotannon historiaa, yleisiä piirteitä sekä ohjel-

mistotuotannon termistöä. Sen jälkeen käydään läpi ohjelmistotuotannon vaihejakoa, ohjelmiston elinkaarta, sekä prosessimallien merkitystä yleensä. Luvun lopuksi esitellään lyhyesti ohjelmistotuotannon ja perinteisten kehitysprosessien tyypillisiä ongelmia sekä tutustutaan yhdysvaltalaisen tutkimuslaitoksen tutkimustuloksiin ohjelmistoprojektien tilasta viimeisen kymmenen vuoden aikana. Luku 3 esittelee lyhyesti yleisimpiä perinteisiä prosessimalleja, poikkeuksena tarkemmin käsiteltävä vesiputousmalli. Perinteisen prosessimallin testausmallina esitellään edelleenkin paljon käytetty V-malli.

Luvussa 4 tutustutaan ketterien ohjelmistoprosessien perusideoihin ja esitellään lyhyesti yleisimpiä ketteriä prosessimalleja. Niissä kehitystyö tapahtuu pienissä ryhmissä, asiakkaan ollessa yleensä aktiivisesti mukana. Luku 5 käsittelee XP:tä ohjelmoinnin prosessimallina ja sen keskeisen aseman vuoksi malli pyritään käymään läpi muita tässä yhteydessä esitettyjä perusteellisemmin. Luvussa käydään läpi XP prosessimallina, XP:tä käyttävän projektin kulku ja henkilöstön roolit. Lisäksi tutustutaan mallin ydinarvoihin ja -käytäntöihin. Luvun lopuksi käydään läpi XP:n laadunhallintaan ja testaukseen liittyviä seikkoja.

Luvun 6 aluksi perehdytään kirjallisuudesta esille tulleisiin tapoihin vertailla prosessimalleja ja vertaillaan yleisellä tasolla ketteriä ja perinteisiä prosessimalleja toisiinsa sekä tarkastellaan ketterien menetelmien soveltuvuutta yleisesti. Luvussa verrataan XP:tä yksinään joihinkin perinteisiin ja ketteriin prosessimalleihin. Lisäksi perehdytään XP:n hyviin ja huonoihin puoliin, etuihin ja haittoihin sekä pohditaan mallin soveltuvuutta erilaisiin ohjelmistoprojekteihin. Päähuomio luvussa kohdistetaan valmiin tutkimusmateriaalin tarkasteluun koskien XP:tä ja erästä sen keskeistä käytäntöä, pariohjelmointia.

Tutkimukset ovat laajuudeltaan erilaisia, suoritettu erilaisissa ympäristöissä ja eri puolilla maailmaa. XP:tä koskevien tutkimusten perusteella on tarkoitus saada kokonaiskuva mallia käyttävien projektien nykytilasta, ongelmista, tuloksista, sekä minkälaisia vaikutuksia XP:n käyttöönotolla ja sen eri käytäntöjen käyttöönotolla on ollut. Lisäksi tarkastellaan XP:n käyttöönoton vaikutuksia ohjelmoijan ja ohjelmistoprosessin tuottavuuteen, sekä muita käytöstä saatuja kokemuksia. Tarkoituksena on myös selventää pariohjelmoinnin roolia eräänä XP:n keskeisenä työmenetelmänä.

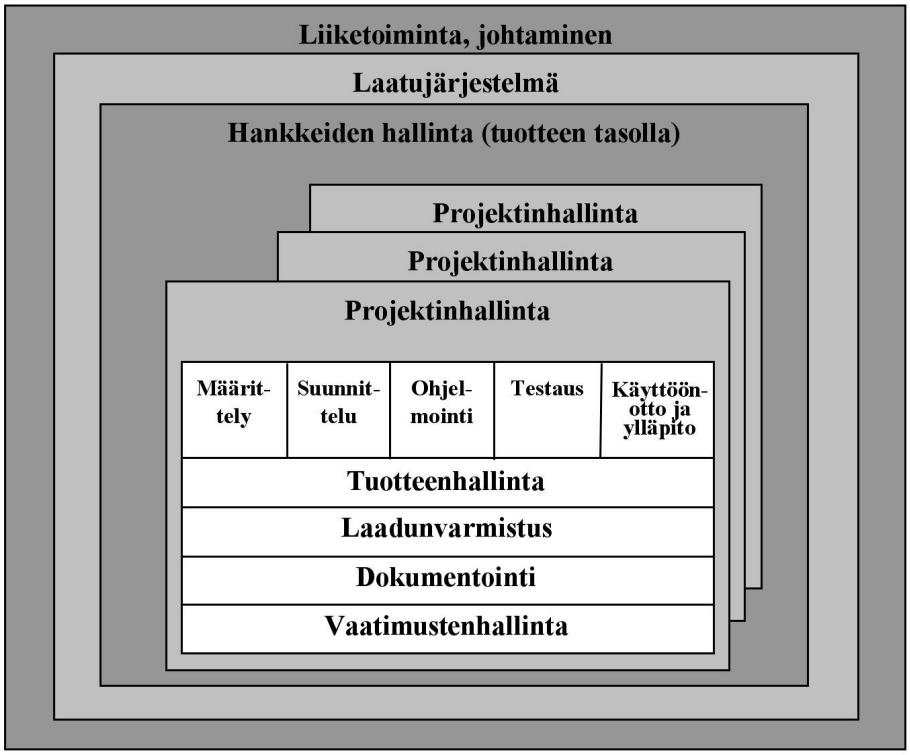
2 OHJELMISTOTUOTANNOSTA YLEENSÄ

Tämän luvun tarkoituksena on toimia pohjana tutkielmassa myöhemmin käsiteltäville aiheille. Luvun alussa esitellään lyhyesti ohjelmistotuotannon historiaa, yleisiä piirteitä sekä ohjelmistotuotannon termistöä. Sen jälkeen käydään läpi ohjelmistotuotannon vaihejakoa, ohjelmiston elinkaarta, ohjelmistokehitysmenetelmien koulukuntia sekä prosessimallien merkitystä yleensä. Luvun lopuksi tutkaillaan lyhyesti ohjelmistotuotannolle ja perinteisille kehitysprosesseille tyypillisiä ongelmia sekä tutustutaan tutkimustuloksiin ohjelmistoprojektien tilasta.

2.1 Ohjelmistotuotannon historiasta, kehityksestä ja käsitteistä

Ohjelmistoista on tullut monilla eri talouden ja teollisuuden aloilla kehitystä eteenpäin ajava voima [Pre05]. Ne vaikuttavat tärkeiden päätösten tekoon toimien samalla perustyökaluina tekniikan ja tutkimuksen ongelmaratkaisulle. Ohjelmistot erottavat nykyaikaiset tuotteet ja palvelut vanhanaikaisista. Niitä käytetään kaikkialla ja tulevaisuudessa yhä enemmän, ollen siten olennainen osa jokapäiväistä elämää. Ohjelmistoihin liittyy myös paljon ongelmia, joista suurimpia ovat erityyppiset ohjelmistovirheet. Yrityksissä suurimmat ohjelmistoihin ja niiden kehittämiseen liittyvät ongelmat ovat ohjelmistokehityksestä koituvat liian korkeat kustannukset, venyneet aikataulut ja usein myös epäonnistuneet projektit [Hai04].

Ohjelmistotuotannolla terminä tarkoitetaan ohjelmistotyötä, jonka tuloksena syntyvät järjestelmät täyttävät kohtuullisesti käyttäjien toiveet ja odotukset valmistuen laadittujen aikataulujen ja kustannusarvioiden puitteissa. Ohjelmistotuotanto voidaan jakaa eri alueisiin kuvassa 1 esitetyllä tavalla, joita ovat kuvan esimerkin mukaisesti laatujärjestelmä, projektinhallinta, tuotteenhallinta, laadunvarmistus, dokumentointi, määrittely, suunnittelu, testaus, käyttöönotto ja ylläpito. [Hai04]



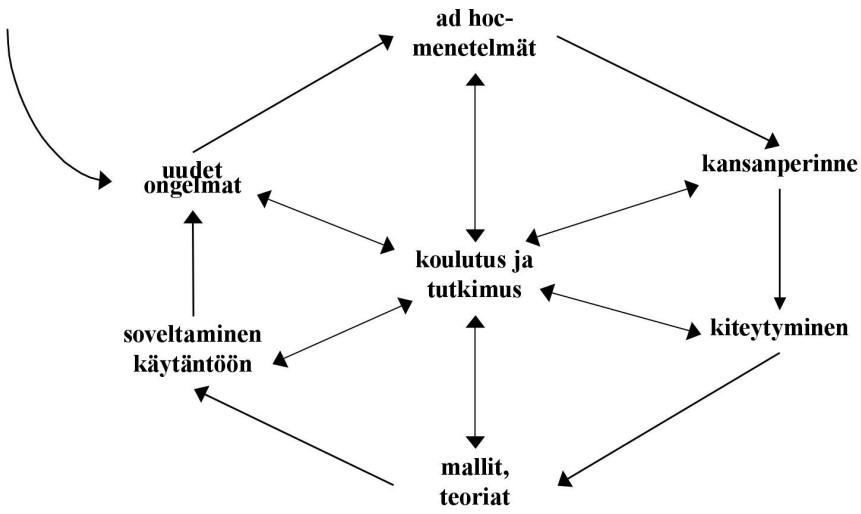
Kuva 1. Ohjelmistotuotannon osa-alueet. [Hai04]

1950–60-luvuilla, jolloin elettiin vielä tietokoneistumisen alkuaikoja, suurin osa kustannuksista sijoitettiin laitteistoihin ja ohjelmistot nähtiin vain sivutuotteina, joilla laitteistot saatiin toimimaan. Ohjelmistomaailma toimi ilman mitään sääntöjä, määriteltyjä metodeja oli vain vähän ja harva käytti niitä. [Pre05] [Somm00]. 1960-luvun lopun ohjelmistokriisin seurauksena huomattiin tarve kehittyneemmille menetelmille, joilla valmistaa ohjelmistotuotteita. Vastaukseksi ongelmaan kehitettiin menetelmiä, joita kutsuttiin myöhemmin yhteisellä nimellä ohjelmistotuotannoksi. [Ghe91]

Ohjelmistotuotanto esiteltiin ensimmäisen kerran omana terminä 1960-luvun lopun ohjelmistokriisin aikaan. Silloisen ohjelmistokriisin taustalla oli, etteivät entiset ohjelmistokehitysmenetelmät enää sopineet uusille ja entistä tehokkaammille tietokoneille. Ohjelmistokehitysprojektit myöhästyivät pahasti, ylittäen samalla budjettinsa ja tuloksena saatujen ohjelmistojen ylläpito oli hankalaa. Tilanne on yhä edelleenkin samankaltainen erilaisten ohjelmistojen tarpeen ja kysynnän kasvaessa kehitysmenetelmien tehokkuuden jäädessä

kasvavan kysynnän tahdista. Tämä johtaa vanhojen virheiden, kuten esimerkiksi aikataulujen venymisen ja budjetin ylittämisen, toistumiseen yhä uudelleen [Somm00].

Tultaessa 1970-luvulle sattumanvarainen ohjelmistotuotanto ei enää menestynyt, sillä tietokonelaitteistojen suorituskyvyn kasvaminen teki mahdolliseksi suunnitella ja käyttää entistä suurempia ja monipuolisempia ohjelmistoja. Ohjelmistojen koon kasvaminen ja samoihin aikoihin alkanut vanhojen ohjelmistojen uusiminen vähensivät ad hoc-tuotantoa. Laajan ohjelmistotuotteen tekeminen oli välttämätöntä organisoida koordinoiduksi prosessiksi, sillä suuren työn hallitseminen ja tehokas työskentely vaativat järjestelmällistä ja suunnitelmallista toimintatapaa. Ensimmäiset ohjelmistokehitysmenetelmät syntyivät muista insinööritieteistä lainattujen mallien pohjalta. Kuvassa 2 esitetään kehämäinen malli ohjelmistotuotannon syntymiselle, jossa lähtökohtana ovat käytännön ongelmat ja niihin etsittävät ratkaisut [Hai04].



Kuva 2. Ohjelmistotuotannon kehittyminen. [Hai04]

Kuvan mallissa kelpaa aluksi mikä tahansa ongelman ratkaiseva tapa ongelman ratkaisuksi. Tällöin sovelletaan niin sanottuja ad hoc-menetelmiä. Hiljalleen löydetään ratkaisuja, jotka toimivat useammassakin kuin yhdessä tapauksessa. Näistä ratkaisuista syntyy alan kansanperinnettä. Kansanperinteenä välittyvän osaamisen kehittyessä järjestelmällisem-

mäksi, se kiteytyy heuristiikoiksi ja työmenetelmiksi. Jatkossa menetelmät kehittyvät tarpeeksi järjestelmällisiksi muodostaakseen malleja ja teorioita. Sovellettaessa malleja käytäntöön, löydetään uusia ongelmia ja sovellusalueita, joille entiset ratkaisumallit eivät enää sovi. Tällöin palataan jälleen takaisin alkuun ja ad hoc-menetelmiin. [Hai04]

Mallit, teoriat sekä käytäntö yhdessä muodostavat ohjelmistokehityksessä käytettävän ohjelmistokehitysmenetelmän. Menetelmä määritellään terminä kokoelmaksi erilaisia teknisiä työvaiheita sekä ohjeita siitä, missä järjestyksessä ja millä tavalla nämä työvaiheet suoritetaan tietyn tavoitteen saavuttamiseksi. Ohjelmistokehitysmenetelmien kehittymiseen ja erilaistumiseen ovat vaikuttaneet asiakkaiden tarpeet. Ohjelmistotuotannon johtavaksi periaatteeksi on muodostunut asiakkaan tarpeiden mukaisen ohjelmistotuotteen toteuttaminen ja lähtökohtana onkin usein asiakkaan toimintaan liittyvä ongelma, jota ohjelmistotuote pyrkii ratkaisemaan. Tässä työssä tarkoitetaan asiakkaalla ohjelmistoprojektien rahoittajia, ohjelmistojen loppukäyttäjiä sekä räätälöityjä ohjelmistoja toteuttavien ohjelmistoyritysten asiakasyrityksiä.

Ohjelmistokehitysprosessi tai ohjelmistoprosessi on kehys toimille, joita vaaditaan ohjelmiston tuottamiseksi. Prosessi määrittelee lähestymistavan, jota ohjelmiston tuottamisessa käytetään. Vastaavasti itse ohjelmisto määritellään niin, että se on ensiksikin joukko käskyjä, jotka suoritettuna tuottavat toivotun toiminnallisuuden ja suorituskyvyn. Toiseksi ohjelmisto on joukko tietorakenteita, jotka mahdollistavat riittävän informaation käsittelyn ja kolmanneksi, ohjelmisto on joukko dokumentteja, jotka kuvaavat ohjelmiston toiminnan ja käytön. [Pre05]

Ohjelmistotuotteita tuotetaan hyvin erilaisiin käyttötarkoituksiin ja erilaisille käyttäjäryhmille. Ohjelmistotuotteet voidaan karkeasti jakaa kahteen eri ryhmään, joista ensimmäisessä ovat yleiset ohjelmisto- eli massatuotteet, jotka tuotetaan ohjelmistoyrityksessä tai avoimessa ohjelmistonkehityksessä. Tällöin ohjelmistotuotteet jaetaan tai myydään kenelle tahansa ohjelmistoa tarvitsevalle. Toisessa ryhmässä tulevat räätälöidyt, asiakkaan tilauksesta ja tarpeiden mukaisesti toteutettavat ohjelmistotuotteet. [Somm00]

2.2 Ohjelmistotuotannon prosessimalleista ja kehitysmenetelmistä

Ohjelmistokehitystä on tehty lähes aina vaihejakoisesti ja tämä ajatus on peräisin jo 1950-luvulta [Ben87]. Winston Roycen toimesta vuonna 1970 esitelty vesiputousmalli on perinteisin vaihejakomalli [Roy70]. Ohjelmiston tekemiseen käytetty vaihejako- eli prosessimalli määrittelee kehitysstrategian, jota käytetään ohjelmiston tuottamiseen. Erilaisia malleja on useita ja mallin valinta tehdään sekä projektin että tuotteen ominaisuuksien perusteella [Pre05].

Prosessimallin ohella myös ohjelmiston elinkaari on yleisesti käytetty käsite. Jokaisella ohjelmistotuotteella on elinkaari, millä tarkoitetaan aikaa ohjelmiston kehittämisen aloittamisesta aina siihen saakka, kunnes ohjelmisto poistetaan käytöstä. Ohjelmiston elinkaari jaetaan prosessimallin avulla erilaisiin vaiheisiin. Prosessimallit eli vaihejakomallit määrittelevät tavan, jolla ohjelmiston elinkaari jaetaan sykleihin tai vaiheisiin. Ne eroavat toisistaan eri vaiheiden painotuksessa ja suoritusjärjestyksessä. [Hai04]

Sommervillen [Somm00] mukaan jokaisesta prosessimallista voidaan löytää neljä perustavan laatuista aktiviteettia. Ensimmäiseksi löytyvä aktiviteetti on ohjelmiston suunnittelu, jossa ohjelman toiminnallisuus ja sen rajoitteet määritellään. Seuraavaksi löytyvä aktiviteetti on ohjelmiston kehitys, jossa ohjelmisto kehitetään suunnitelmien mukaisesti. Kolmas aktiviteetti on ohjelmiston validointi, jossa ohjelmisto testataan toteuttaako se asiakkaan vaatimukset. Neljäs on ohjelmiston evoluutio, jolloin ohjelmisto muuntautuu muuttuvia asiakasvaatimuksia vastaavaksi.

Yleisesti prosessimallit vastaavat kysymykseen mitä ohjelmistoprojektissa tulisi seuraavaksi tehdä ja kauanko sen tekemiseen kuluu aikaa [Boe00]. Prosessimallien avulla ohjelmistoprosessia pyritään ymmärtämään ja kontrolloimaan luoden näin pohjan, jolle ohjelmistokehitys perustuu [Not88]. Ohjelmistoprosessin määritys on samaan aikaan sekä tekninen kehys että johtamisen malli ohjelmistonkehittäjien työskentelylle ja menetelmien käytölle [Hum89]. Prosessimuotoisen työtavan tavoitteita ovat esimerkiksi kulujen vähentäminen, ajan säästäminen, tuotteen laadun parantaminen ja työn mielekkyyden lisääminen hallitsemalla informaation prosessointia [Dav90].

Ohjelmistoprosessin avulla ohjelmistonkehittäjät kykenevät tekemään oikeita valintoja ja esimerkiksi ongelmakohdat kyetään paikantamaan ja eristämään entistä nopeammin ennen kuin ne aiheuttavat liikaa kustannuksia. Hyvin määritetyn ohjelmistoprosessin avulla ongelmien ennustettavuus ja hallinta paranee ja ohjelmistokehityksestä tulee vakaampaa. Ohjelmistoprosessi on aina yrityskohtainen ja siitä voi olla samaan aikaan käytössä useita eri muunnelmia. Ohjelmistoprosessien yhteiset piirteet mahdollistavat niiden hallinnan ja luokittelun. Prosessien hallinnan mahdollistamiseksi täytyy niiden sisältää joitakin määreitä, sääntöjä ja käytäntöjä vakauden saavuttamiseksi. [Hum89]

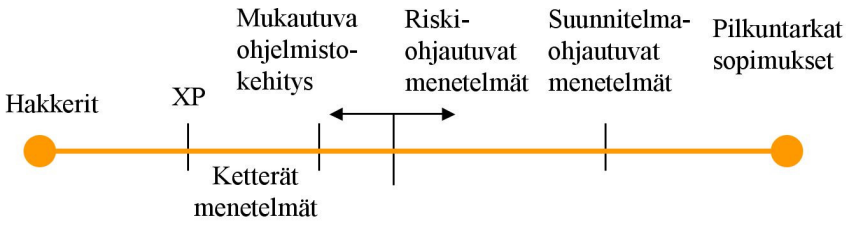
Sommervillen [Somm00] mukaan ohjelmistoprosessit voidaan erottaa toisistaan määreiden ilmenemistavan mukaan, nimeten määreiksi prosessin määrittelyn ymmärrettävyyden, etenemisen näkyvyyden ja välitulosten selvyyden. Muita määreitä ovat esimerkiksi mahdollisten virheiden aikainen havaittavuus, prosessin kestävyys ja ylläpidettävyys suhteessa ympäröivään organisaatioon ja ohjelmiston valmistumisaika. Edellä mainitut ohjelmistoprosessin määreet pätevät kaikkiin ohjelmistoprosesseihin, vaikka prosessin sisältämät tehtävät saattavatkin vaihdella.

Ohjelmistoprosessien yhteydessä termit inkrementaalinen ja iteratiivinen ohjelmistonkehitys sekoitetaan usein keskenään tai ne määritellään samaksi asiaksi. Iteratiivinen ohjelmistonkehitys tarkoittaa prosessimallia, jossa sama asia tehdään kerta toisensa jälkeen tarkentaen, laajentaen ja korjaten edellistä tuotosta. Iteratiivisuuteen kuuluu myös se, että suunnitelmat täydentyvät koko ajan prosessin aikana. Iteraation ideana on sopivin väliajoin tarkastaa, että projekti etenee oikeaan suuntaan ja tehdä suunnitelmat seuraavaa iteraatiota varten.

Inkrementaalinen ohjelmistonkehitys on iteraation tulos, mutta inkrementteja voidaan tehdä myös ilman iteraatioita. Tällöin inkrementit on suunniteltu etukäteen, mutta ne toteutetaan useammassa osassa. Inkrementaalinen ohjelmistonkehitys sisältää kaikki ne prosessimallit, joissa lopputuotos saavutetaan väliversioiden, inkrementtien, kautta. Nämä väliversiot ovat testattuja ja toimivia ohjelmia, joista puuttuu projektin myöhemmissä vaiheissa tehtäviä tiettyjä toiminnallisuuksia.

Erilaisten ohjelmistokehitysmenetelmien kannattajat voidaan jakaa karkeasti kahteen koulukuntaan niiden tuottamien dokumentaation määrän perusteella. Ensimmäinen näistä kahdesta koulukunnasta kannattaa suunnitelmaohjautuvia, rakenteellisia, malleja joissa esimerkiksi erilaiset tehtävät, merkkipaalut ja määrittelyt suunnitellaan ja dokumentoidaan mahdollisimman tarkasti. Ohjelmiston kehitys nähdään elinkaarena, joka alkaa tarpeiden määrittelystä ja päättyy ohjelmiston ylläpitoon.

Toinen koulukunta kannattaa menetelmiä, joissa suunnittelu perustuu ihmisten välisellä keskustelulla tapahtuvaan tiedonhankintaan. Tähän menetelmään kuuluvia menetelmiä kutsutaan tässä yhteydessä ketteriksi ohjelmistokehitysmenetelmiksi. Kuvassa 3 esitetään graafisesti eri ohjelmistokehitysmenetelmien suunnitelmallisuusasteita, joiden perusteella voidaan lajitella menetelmiä eri ryhmiin. Suunnitelmallisuusaste kuvaa sen kirjallisen työn määrää, jota ohjelmiston toteuttamiseksi tehdään [Boe02].



Kuva 3. Eri ohjelmistokehitysmenetelmien suunnitelmallisuus. [Boe02]

Kuvassa vasemmalla ovat hakkerit, jotka ryhtyvät ohjelmoimaan ilman kirjallisia suunnitelmia. Keskiviivan oikealla puolella ovat suunnitelmiin pohjautuvat menetelmät, päätyen oikealla olevaan pilkuntarkkaan sopimuksessa ohjelmiston ominaisuudet määrittelevään suunnitelmaohjautuvaan ohjelmistokehitysmenetelmään.

Tultaessa 1990-luvun loppupuolelle alkoivat useat uudet ohjelmistokehitysmenetelmät saada huomiota ohjelmistotekniikan alalla. Nämä menetelmät yhdistelivät vanhoja ja uusia ideoita sekä muuntelivat vanhoja ja hyväksi koettuja. Yhteistä näille menetelmille oli muun muassa läheinen yhteistyö eri sidosryhmien kesken, kasvokkain tapahtuva kommunikointi

asiakkaan kanssa sekä ohjelmointikäytännöt ja muuttuviin vaatimuksiin vastaamaan kykenevät työryhmät.

Vuonna 2001 Yhdysvalloissa uusien ohjelmistokehitysmenetelmien kehittäjät kokoontuivat miettimään, mitä yhteistä näissä menetelmissä voisi olla. Tässä yhteydessä menetelmiä alettiin kutsua termillä ”ketterä”, jonka katsottiin kuvaavan hyvin menetelmille yhteisiä piirteitä. Kokoontumisessa muotoiltiin ja julkaistiin manifesti, joka määritteli ketterälle sovelluskehitykselle yleiset arvot ja periaatteet. [Abr02] [Bec01]

Ketterät menetelmät ovat viime vuosina saaneet etenkin Yhdysvalloissa lähes kansanliikkeen piirteitä ja monet ohjelmistonkehittäjät ovat kokeneet ne tervetulleeksi vastavoimaksi niille kankeille työmenetelmille, joita esimerkiksi Yhdysvaltojen julkishallinnon projekteissa suositaan [Hai04]. Tähän mennessä on erilaisia ketteriä ohjelmistokehitysmenetelmiä määritelty ja esitelty yhteensä kymmenkunta [Abr02]. Niiden tärkeimpiä yhteisiä piirteitä ovat iteratiivinen ohjelmistokehitys sekä asiakkaan tiivis mukanaolo koko ohjelmistoprojektin aikana [Abr02].

Tunnetuimpana ketterien menetelmien ohjelmistokehitysmenetelmänä pidetään yleisesti Kent Beckin vuonna 1999 määrittelemää XP-mallia (Extreme Programming) [Bec00]. Sitä pidetään ketterien ohjelmistokehitysmenetelmien uranuurtajana ja kehityksen alkuna ja XP:n keskeisen aseman vuoksi tässä työssä keskitytään XP-mallin laajempaan käsittelyyn.

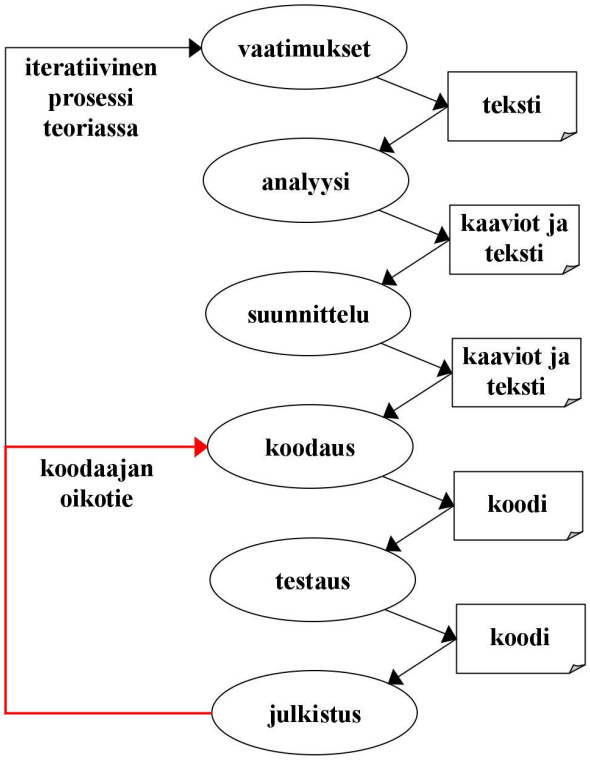
2.3 Ohjelmistotuotannon ongelmista

Ohjelmistotuotannon ensimmäiset menetelmät eivät kyenneet poistamaan 1960-luvulla syntynyttä ohjelmistokriisiä [Somm00]. Ohjelmistokriisin alussa ohjelmistotuotannon pahimpana ongelmana pidettiin ohjelmistotyön heikkoa tuottavuutta. Nykyään ongelmat käsitetään laajemmin kokonaisvaltaiseksi ohjelmistotuotannon laatuongelmaksi, sillä ohjelmistot ovat huonolaatuisia ja kalliita, valmistuvat myöhässä aikataulustaan ollen samalla tarkoitukseensa epäsopivia ja kaiken lisäksi usein selvästi viallisia [Wik07c]. Ohjelmistotuotannon ongelmakohdiksi mainitaan muun muassa käyttäjätarpeiden ymmärtäminen, projektin laajuuden määritteleminen sekä muutosten hallinta [Ree99]. Sommerville listaa

tärkeimmiksi ongelmiksi toimintojen integroinnin, projektin hallinnan ja kommunikaation [Somm00].

Brooks [Bro87] määrittelee No Silver Bullet-artikkelissaan syitä, mikä tekee ohjelmistotuotannosta niin vaikeaa. Brooksin artikkelin mukaan tietokoneohjelmistot ovat kokoonsa nähden mahdollisesti monimutkaisimpia yksittäisiä rakennelmia mitä ihminen on koskaan tehnyt ja sen vuoksi myös niiden tekemiseen käytettävät menetelmät ovat monimutkaisia. Tietokone-ohjelmistojen olisi kyettävä mukautumaan mitä erilaisimpiin käyttötilanteisiin ja -tarkoituksiin, sen lisäksi niihin kohdistuu jatkuvasti muutospaineita ja muutokset tulisi olla helposti tehtävissä. Lisäksi ohjelmistot ovat luonteeltaan abstrakteja, näkymättömiä ja vailla fyysisiä ulottuvuuksia.

Kleppe [Kle03] mainitsee ohjelmistotuotannon päällimmäisiksi ongelmiksi tuottavuusongelman, siirrettävyysongelman, yhteentoimivuusongelman sekä dokumentaatio- ja ylläpitoongelman. Perinteisen ohjelmistotuotantoprosessin mukaisesti suurin osa dokumenteista ja kaavioista syntyy ohjelmointia edeltävissä vaiheissa. Usein käy kuitenkin niin, että ohjelmoinnin alkaessa dokumenttien sisältämät kaaviot saattavat menettää merkityksensä, johtuen esimerkiksi niiden epätarkkuudesta. Väli ohjelmakoodin ja kaavioiden välillä kasvaa entisestään, mikäli sovellukseen tehdään muutoksia. Koska korkea tuottavuus on nykyään ensisijaisen tärkeää, ajan puute muodostuu ongelmaksi eikä dokumenttien muokkaaminen edistä varsinaisen sovelluksen tekemistä. Nämä seikat aiheuttavat tuottavuusongelman. Kuvassa 4 esitetään perinteisessä ohjelmistoprosessissa käytetty ohjelmistokehittäjän oikotie dokumenttien päivittämisen osalta.



Kuva 4. Perinteisen ohjelmistotuotantoprosessin oikopolku. [Kle03]

Nopeasti kehittyvät teknologiat pakottavat yritykset sopeutumaan uusien asioiden mukanaan tuomiin muutoksiin. Yritykset ovat pakotettuja ottamaan käyttöön uusia teknologioita, koska asiakkaat saattavat vaatia omiin järjestelmiinsä muutoksia [Kle03]. Uudella teknologialla voidaan myös ratkaista kriittisiä ongelmia, joten muutos on väistämätön. On myös huomioitava, etteivät erilaisten työkaluohjelmistojen kehittäjät tue vanhoja ohjelmistoversioitaan loputtomiin teknologioiden muuttuessa.

Siirrettävyysongelman vuoksi joudutaan miettimään muunnetaanko sovellus uudelle teknologialle vai uudelle versiolle jo olemassa olevasta teknologiasta. Kummassakin tapauksessa sovelluksen pitää kyetä kommunikoimaan ongelmitta uudella teknologialla tehtyjen sovellusten kanssa. Yhteentoimivuusongelma koetaan silloin, kun sovellusten pitäisi kyetä kommunikoimaan monilla eri teknologioilla rakennettujen järjestelmien ja komponenttien kanssa [Kle03].

Ohjelmistotuotannon heikkona lenkkinä pidetään dokumentaation tuottamista, joka vaikuttaa koko prosessin laatuun. Dokumentaatio tuotetaan tavallisesti ohjelmoinnin jälkeen kiireellä ja vain koska sen on oltava olemassa myöhempiä ohjelmistokehitysprosessin vaiheita varten. Dokumenttien huono laatu johtuu pakon tunteesta, jota sen tekeminen aiheuttaa sovelluskehittäjien keskuudessa. Tämän syyn seurauksena dokumenttien huolellinen tarkastaminen jää usein tekemättä. [Kle03]

Internetiin liittyvä ohjelmistokehitys poikkeaa perinteisestä ohjelmistokehityksestä. Baskerville [Bas03] kuvaa Internetiin liittyvää ohjelmistokehitystä siten, että sovellusten toteuttamiseen käytössä oleva aika on lyhyempi kuin koskaan aikaisemmin, sovelluksille asetettavat vaatimukset muuttuvat nopeassa tahdissa, sovelluksen laadun kriteerit poikkeavat perinteisestä ohjelmistokehityksestä ja tuotteen saaminen ajoissa markkinoille on kriittistä. Baskervillen [Bas03] mukaan sovellusten elinkaaren lyhentyminen johtaa siihen, että ylläpidettävyyteen tähtäävät toimenpiteet unohdetaan osittain tai kokonaan. Internetnopeus edellyttää uudenlaisia toimintatapoja, kun sovelluskehitystä halutaan tehdä hallitusti vaaditusta nopeudesta huolimatta. Varteenotettavan vaihtoehdon tähän tarjoavat ketterän ohjelmistokehityksen menetelmät.

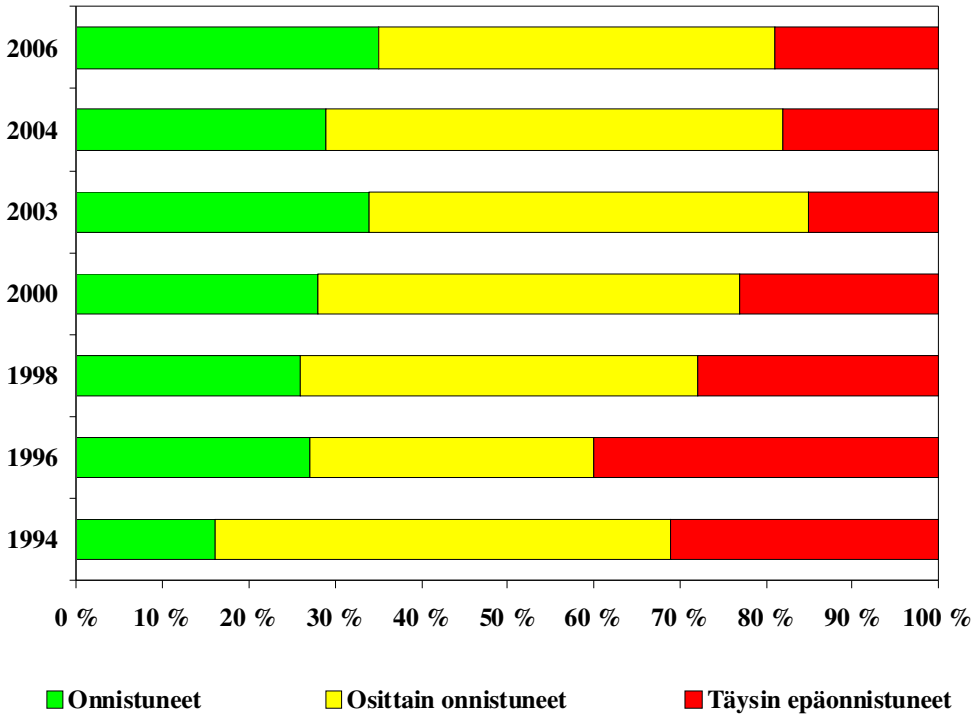
Samalla tavoin mobiililaitteisiin liittyvän ohjelmistokehityksen voidaan ajatella eroavan perinteisestä ohjelmistokehityksestä. Internetiin ja mobiililaitteisiin liittyvälle ohjelmistokehitykselle on yhteistä ohjelmistotuotteiden lyhyt elinkaari ja kriittinen vaatimus saada ohjelmistotuote ajoissa markkinoille sekä vaatimus tehdä sovelluskehitystä nopeasti, mutta hallitusti. Näin on syntynyt tarve uusille ja entistä joustavammille prosessimalleille, jotka pyrkivät eliminoimaan ohjelmistokehityksen suurimpia riskejä.

Perinteiset prosessimallit on havaittu monissa tapauksissa liian jäykiksi ja kykenemättömiksi vastaamaan asiakkaan muuttuviin vaatimuksiin. Ratkaisuksi tällaisiin ongelmiin on lähdetty kehittämään joustavampia prosessimalleja. Ketterät prosessimallit pyrkivät tuomaan kaivattua joustavuutta ohjelmistokehitykseen. Mallien tavoitteena on tuottaa lyhyin väliajoin asiakkaalle toimivia versioita kehitettävästä ohjelmistosta, pyrkiä hallitsemaan muuttuvia vaatimuksia myös myöhäisessä projektin vaiheessa, sekä olla entistä enemmän

yhteistyössä asiakkaan kanssa. Ketterissä prosessimalleissa nähdään toimivan ja laadukkaan ohjelmiston olevan tärkeämpää kuin kattavan dokumentaation.

Ohjelmistoprojektien onnistumisesta on tehty useita tutkimuksia eri puolilla maailmaa. Yhdysvaltalainen Standish Group-tutkimuslaitos on viimeisen kymmenen vuoden aikana tutkinut yli 50 000:ta ohjelmistoprojektia ja tutkimuslaitoksen tekemän CHAOS-raportin tulosten perusteella ilmenee, että onnistuneiden ohjelmistoprojektien osuus on kasvanut sinä aikana. Tärkeimpänä syynä siihen pidetään projektien koon huomattavaa pienenemistä ja sitä, että ohjelmistoprojekteissa on prosessimalleiksi yhä enenevässä määrin valittu perinteisen vesiputousmallin sijaan iteratiiviset mallit. Lisäksi todetaan, että yritykset ja niiden takana olevat ihmiset osaavat käyttää hyväkseen erilaisia projektihallinnon keinoja paljon paremmin kuin aiemmin. [Has07] [Kol07] [SoD07] [SoM04] [Sta95] [Sta98]

Standish Groupin tutkimuksissa projektit luokitellaan kolmeen eri luokkaan niiden onnistumisen perusteella. Ensiksi onnistuneisiin projekteihin, jotka valmistuvat kaikkine sovittuine ominaisuuksineen pysyen aikataulussaan ja budjetissaan. Toiseksi osittain onnistuneisiin projekteihin, jotka valmistuvat karsituin ominaisuuksin, myöhässä aikataulusta ja/tai suunniteltua suuremmin kustannuksin eli ylittäen budjetin ja kolmanneksi projekteihin, jotka lopetettiin ennen niiden valmistumista ja luokiteltiin täysin epäonnistuneiksi. Huolimatta onnistuneiden ohjelmistoprojektien osuuden kasvusta, vuonna 2006 vain 35 % projekteista onnistui täydellisesti pysyen sekä aikataulussaan että budjetissaan. 46 % ohjelmistotuotantoprojekteista onnistui vain osittain ja 19 % projekteista lopetettiin kokonaan ennen valmistumista. Tutkimusten tulosjakauma, alkaen vuodesta 1994 ja päätyen vuoteen 2006, on esitetty kuvassa 5. [Has07] [Kol07] [SoD07] [SoM04] [Sta95] [Sta98]



Kuva 5. Standish Groupin tutkimustulokset vuosilta 1994-2006. [Has07] [Kol07] [SoD07] [SoM04] [Sta95] [Sta98]

Puhuttaessa rahasummista, jotka menetetään epäonnistuneisiin ohjelmistoprojekteihin, voidaan todeta niiden olevan merkittäviä. Standish Group arvioi, että pelkästään Yhdysvalloissa menetetään vuosittain 80–145 miljardia dollaria epäonnistuneisiin ja peruttuihin ohjelmistoprojekteihin. Lisäksi artikkelissa todetaan, että jopa 25–40%, kaikesta ohjelmistoprojekteihin käytetystä rahamäärästä kuluu vikojen etsimiseen ja korjaamiseen. Ongelmat eivät rajoitu pelkästään ohjelmistoalan yrityksiin, sillä huonosti toteutetut ja toimivat ohjelmistot aiheuttavat vuosittain Yhdysvalloissa muille kuin IT-alan yrityksille 30 miljardin dollarin tappiot. Artikkelin mukaan 60–80% ohjelmistoprojektien epäonnistumisista voidaan johtaa vaatimusten huonoon määrittelyyn ja analysointiin, asiakkaiden osallistumisen puuttumiseen ja projektin huonoon johtoon. Niinpä kaksi kolmesta ohjelmistoprojektista ajautuu ongelmiin tai päättyy epäonnistumiseen. [Has07]

3 PERINTEISET PROSESSIMALLIT

Tässä luvussa tarkastellaan lyhyesti yleisimpiä ohjelmistokehityksessä käytettäviä perinteisiä prosessimalleja ja niiden perusperiaatteita. Vesiputousmalli on tunnetuin ja perinteisin ohjelmistotuotannossa käytetty elinkaarimalli ollen edelleenkin hyvin yleisesti käytössä. Lisäksi perinteiset prosessimallit pohjautuvat osin vesiputousmalliin, minkä vuoksi se käydään läpi ensiksi muita yksityiskohtaisemmin. Muut perinteiset prosessimallit esitellään vesiputousmallin jälkeen omassa kokonaisuudessaan.

3.1 Yleistä perinteisistä prosessimalleista

Perinteistä ohjelmistokehitystä kutsutaan kirjallisuudessa myös suunnitelmalähtöiseksi ohjelmistokehitykseksi. Niissä suoritettavat tehtävät ja määrittelyt pyritään suunnittelemaan sekä dokumentoimaan mahdollisimman tarkasti. Ohjelmiston kehitys nähdään elinkaarena, joka alkaa toteuttavan ohjelmiston tarpeiden määrittelystä päätyen lopulta valmiin ohjelmiston ylläpitoon.

Perinteiset prosessimallit perustuvat ajatukseen ohjelmiston toteuttamisesta kokoelmana lineaarisesti eteneviä vaiheita, vaiheiden ollessa perusajatukseltaan yleensä samoja kaikissa malleissa. Erona on yleensä, että vaiheiden esiintymistiheys ja toteutustapa voivat vaihdella. Vaiheet ovat yleensä määrittely, suunnittelu, toteutus, testaus sekä käyttöönotto ja ylläpito [Hai04].

Vaiheita kutsutaan yhteisesti ohjelmiston elinkaaren vaihejaoksi ja ne etenevät järjestyksessä määrittelystä ylläpitoon. Lopputuloksena on asiakkaan tarpeita ja määrittelyjä vastaava ohjelmisto. Vaiheisiin liittyy yleensä myös dokumentaatio, joka valmistuu aina vaiheen päättyessä. Kustakin vaiheesta syntyvä dokumentaatio ohjaa omalta osaltaan ohjelmistokehitystä eteenpäin seuraavaan vaiheeseen [Vli00]. Ohjelman elinkaarella tarkoitetaan aikaa, joka kuluu ohjelmiston kehittämisen aloittamisesta sen käytöstä poistamiseen [Hai04].

Yleisesti käytössä olevia perinteisiä prosessimalleja ovat vesiputous-, prototyyppi-, evoluutio- ja spiraalimalli. Tiedot eri prosessimallien käytön laajuudesta ovat vaihtelevia ja riip-

puvat kulloinkin käsillä olevasta tietolähteestä. Sommervillen [Somm00] mukaan evoluutiomallin mukaiset menetelmät ovat tällä hetkellä laajimmin käytettyjä. Sommerville lisää, että iteratiiviset menetelmät ovat tulevaisuudessa laajasti käytettäviä menetelmiä.

Robinson [Rob07] toteaa käytetyimmän lähestymistavan ohjelmistotuotannossa olevan edelleenkin perinteinen koodaa ja korjaa-malli. Kroll [Kro07] esittää artikkelissaan, että useimmat ohjelmistokehitysryhmät käyttävät edelleenkin perinteistä vesiputousmallia, mutta Krollkin on sitä mieltä, että ohjelmistoteollisuus on siirtymässä kohti iteratiivisia, tarkentavia ja laajentavia menetelmiä. Tässä tutkielmassa kuitenkin lähdetään siitä oletuksesta, että perinteinen vesiputousmalli, sekä muut perinteiset mallit ovat edelleenkin käytössä useimmissa tapauksissa ohjelmistoprosessimallina.

3.2 Vesiputousmalli

3.2.1 Yleistä vesiputousmallista

Ohjelmistotuotantoprosessi on perinteisesti perustunut vesiputousmallille. Winston Royce esitteli mallin vuonna 1970 tavoitteenaan saada ohjelmistotuotannosta systemaattisempaa. Royce kuvasi mallissaan oman aikansa ohjelmistotuotantoon vakiintuneita käytäntöjä pyrkien kokoamaan niiden parhaat ominaisuudet. [Roy70]

Vesiputousmallia sovellettiin yleisesti vielä 1990-luvun alkupuolen ohjelmistotuotannossa, koska ohjelmistot toteutettiin alusta loppuun asti kokonaisuuksina. Vesiputousmalli on ollut pohjana useille muille, hieman kehittyneimmille elinkaarimalleille, joilla on korjailtu perinteisen vesiputousmallin sisältämiä ongelmia. Joskus vesiputousmallista puhutaan myös klassisena elinkaarimallina tai lineaarisena peräkkäismallina [Pre05].

Vesiputousmalli periytyy muista teollisuusprosesseista parantaen prosessin näkyvyyttä. Vesiputousmallia voidaan kuvata dokumenttivetoiseksi prosessiksi, jossa jokainen vaihe tuottaa dokumentaatiota, kuten esimerkiksi suunnitteludokumentteja ja testisuunnitelmia. Niiden perusteella tunnistetaan eri vaiheiden päättymiset ja ennen siirtymistä seuraavaan vaiheeseen vaaditaan aina dokumentti. [Hai04] [McC02] [Somm00]

Vaatimusten määrittely tehdään aina prosessin alussa ja uusia vaatimuksia ei hyväksytä enää jälkeenpäin. Vaatimusten määrittelyvaihe ja määrittelyvaiheesta syntyvä dokumentaatio toimivat sopimuksena asiakkaan ja toimittajan välillä. Jokaisessa prosessivaiheessa hyväksytty tuotos pysyy muuttumattomana ohjelmiston tuotannon loppuun saakka. Seuraava vaihe aloitetaan, kun asiakas tai projektin johtoryhmä on antanut hyväksyntänsä edellisen vaiheen valmistumiselle, vaiheen tuotoksen toimiessa seuraavan vaiheen lähtökohtana.

Vesiputousmallista on olemassa erilaisia muunnelmia, mutta perusajatus on niissä kaikissa lähes samanlainen. Mallin avulla suunnitteluprosessin rakennetta voidaan tarkastella yleisesti, sillä kaikki tunnetut prosessimallit sisältävät siinä kuvaillut prosessin vaiheet. Mallien erot eivät ole työvaiheiden sisällössä, vaan siinä miten työt jaksotetaan ja ryhmitetään prosessin aikana.

3.2.2 Vesiputousmallin vaiheet

Vesiputousmalli koostuu eri vaiheista, jotka suoritetaan loppuun saakka ja joista saatavat tulokset toimivat seuraavan vaiheen syötteenä hyväksymisen jälkeen. Vesiputousmallissa projekti jakautuu selvästi erillisiin vaiheisiin eli perustehtäviin, jotka suoritetaan peräkkäin järjestyksessä ja kukin vaihe vain ja ainoastaan kerran. Vesiputousmallin vaiheita ovat vaatimusten analysointi- ja esitutkimusvaihe, määrittelyvaihe, järjestelmän- ja ohjelmiston suunnitteluvaihe, toteutusvaihe, integrointi- ja testausvaihe, sekä käyttöönotto- ja ylläpitovaihe. [Hai04]

3.2.2.1 Vaatimusten analysointi- ja esitutkimusvaihe

Vaatimusten analysointi- ja esitutkimusvaiheen tarkoituksena on tutkia, onko ohjelmistoprojekti oikea ratkaisu esillä olevaan ongelmaan. Tarkoituksena on myös saada arvio projektiin tarvittavasta ajasta, henkilöstöstä ja kustannuksista. Esitutkimusvaiheesta syntyy yleensä raportti ja useimmiten myös alustava projektisuunnitelma. Vaatimusten analysointi- ja esitutkimusvaihe suoritetaan tiiviissä yhteistyössä asiakkaan kanssa, yrittäen selvittää minkälainen ratkaistava ongelma on. Kun ongelma on yleisesti ottaen selvillä, mietitään onko se mahdollista ratkaista olemassa olevilla resursseilla. Asiakkaalle pyritään arvioi-

maan mahdollisia kustannuksia, projektin kestoa, sekä muita käytännön asioita. Vaiheen yhtenä tärkeänä päämääränä on selvittää kannattaako projektia aloittaa. [Somm00]

3.2.2.2 Määrittelyvaihe

Määrittelyvaiheessa pyritään etsimään vastaus kysymykseen, mitä ohjelmiston tulisi tehdä. Määrittelyvaiheessa kerätään ohjelmistolle asetetut toiminnalliset, tekniset ja muut vaatimukset, sekä tarkennetaan myös esitutkimusvaiheen projektisuunnitelma saadun uuden tiedon perusteella. Määrittelyvaiheessa mietitään, minkälainen järjestelmä pitäisi laatia, jotta vaatimusten analysointivaiheessa määritelty ongelma ratkeaisi. Asiakkaan vaatimuksista johdetaan sovelluksen vaatimukset. Tässä vaiheessa ei vielä mietitä käytännön toteutusta tai tekniikkaa, vaan pohditaan järjestelmän erilaisia käyttötapauksia ja toiminnallisia vaatimuksia. Määrittelyvaiheen tuotoksena valmistuu vaatimusmäärittely, jossa kuvataan järjestelmälle asetetut vaatimukset. [Somm00]

3.2.2.3 Suunnitteluvaihe

Suunnitteluvaiheessa pyritään löytämään vastaus kysymykseen, miten ohjelmisto tulisi rakentaa. Tässä vaiheessa ohjelma suunnitellaan teknisesti, mutta mitään ei vielä ohjelmoida. Suunnitteluvaiheen lopputuloksena syntyy yksi tai useampia teknisiä dokumentteja, joissa määritellään ohjelmiston arkkitehtuuri, pysyvät datat ja niiden talletusratkaisut, käyttöliittymät, viestinvälitysmekanismit ja niin edelleen. Suunnitteluvaiheessa ei enää olla paljoa tekemisissä ohjelmistotuotteen tilanneen asiakkaan kanssa, vaan siirrytään enemmän jo ohjelmistotuotteen toteutuksen puolelle. Tässä vaiheessa mietitään kuinka määrittelyvaiheessa määritellyt vaatimukset täyttävä järjestelmä saadaan toteutettua. Vaiheen tuotoksena on järjestelmän ja ohjelmiston tekninen määrittely. [Somm00]

3.2.2.4 Toteutusvaihe

Toteutusvaiheessa päästään itse ohjelmointiin ja tekniseen toteutukseen. Pohjana ovat suunnitteluvaiheessa tehdyt tekniset määrittelyt ja päämääränä on toteuttaa järjestelmän eri kokonaisuudet. Vaiheen tuloksena ovat valmiit ohjelmakoodirakenteet sekä niiden koodikommentit. Järjestelmän eri osat testataan erikseen niiden toteutuksen jälkeen. [Somm00]

Vesiputousmallin ideologiaan kuuluu, että varsinaiseen ohjelmointiin halutaan käyttää niin vähän energiaa kuin mahdollista. Tähän pyritään mahdollisimman perusteellisella suunnittelulla. Asiakkaalta pyritään saamaan selville järjestelmän määrittelyt täydellisinä ennen toteutustyön aloittamista. Todellisuudessa tähän tilanteeseen pääseminen on kuitenkin hyvin vaikeaa tai jopa mahdotonta. [Vli00]

3.2.2.5 Integrointi- ja testausvaihe

Integrointi- ja testausvaiheessa eri osat yhdistetään toimivaksi kokonaisuudeksi ja niiden toiminta testataan. Tässä vaiheessa testataan esimerkiksi rajapintojen toimivuus, koko järjestelmän verifioinnissa selvitetään että järjestelmä toimii, ja validoinnilla selvitetään että järjestelmä tekee varmasti sen mitä järjestelmän halutaan tekevän. Erilaisia testauskriteereitä voivat olla esimerkiksi ohjelmiston toimintojen toimiminen oikeilla ja väärillä syötteillä, käytettävyys, tietoturva ja niin edelleen. [Somm00]

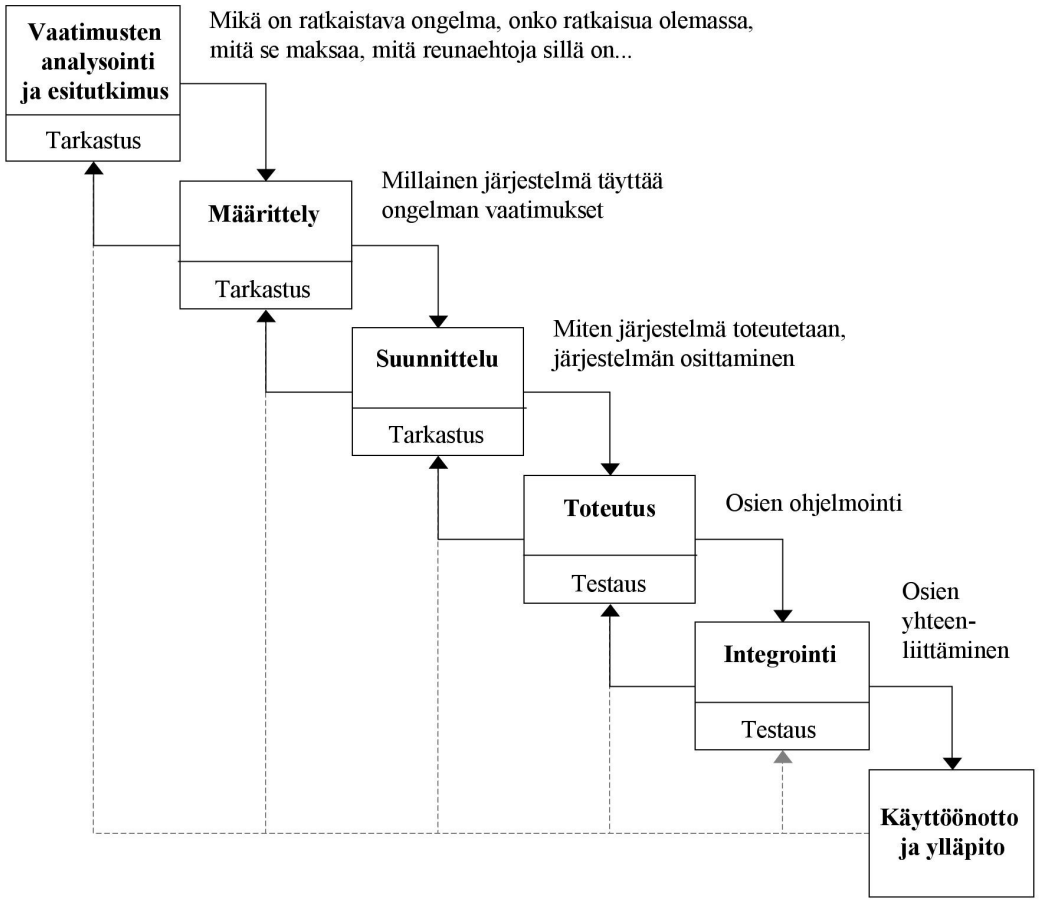
Ennen varsinaista testausta projektissa tuotetaan testaussuunnitelma, jonka mukaan testaus suoritetaan. Integrointi- ja testausvaiheessa syntyy puolestaan testausraportti, joka kertoo testien tuloksista. Testausta varten voidaan suuremmissa projekteissa laatia myös oma projektisuunnitelma. Testausta voidaan suorittaa useita kierroksia, mikä yleensä suoritetaan kun löydetyt virheet korjataan ja korjatut toiminnot testataan uudelleen.

Vesiputousmallissa laadunvarmistus ja testaus ovat sisäänrakennettuja ominaisuuksia. Jokaisen vaiheen suorituksen jälkeen tarkistetaan, ollaanko tekemässä oikeanlaista ohjelmistoa, sekä ollaanko tekemässä ohjelmistoa oikein. Mikäli näissä testeissä huomataan, että työn tuloksena on syntymässä vääränlainen ohjelmisto tai ohjelmistoa ei ole toteutettu oikein, voidaan palata takaisin tähän työvaiheeseen ja korjata havaitut puutteet. [Pre05]

3.2.2.6 Käyttöönotto- ja ylläpitovaihe

Käyttöönotto- ja ylläpitovaiheessa järjestelmän tulisi olla toimiva ja tyydyttää asiakkaan sille asettamia tavoitteita [Somm00]. Tässä vaiheessa ohjelmisto siirretään organisaation käyttöön, annetaan mahdollisesti käyttäjille tarvittava koulutus ohjelmiston käyttöön ja sitä jatkokehitetään sekä käytössä huomattuja uusia toiveita toteutetaan. Myös käytössä havait-

tuja virheitä tai puutteita korjataan. Haikalan [Hai04] mukaan esitetyn vesiputousmallin eri vaiheet on koottu kuvaan 6.



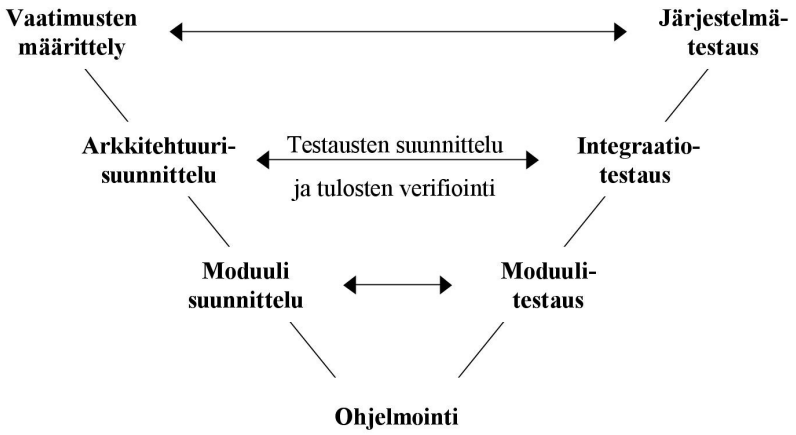
Kuva 6. Vesiputousmalli. [Hai04]

3.2.3 Vesiputousmallin testaus (V-malli)

Vesiputousmallista on kehitetty toinen versio, jota kutsutaan yleisesti V-malliksi. Se on syntynyt vertailemalla ja kehittämällä ohjelmistoprojekteissa käytettyjä testausmenetelmiä. V-mallissa määritellään, että yhden tason testauksen tulee olla suoritettuna, ennen kuin voidaan siirtyä testaamaan seuraavaa tasoa. V-mallissa on vesiputousmallin joka vaiheeseen yhdistetty vastaava testaus, esimerkiksi vaatimusmäärittelystä hyväksymistestaus, ja näin on aikajanalle saatu syntymään niin kutsuttu V-malli, jonka vasen sakara kuvastaa

ohjelmistoprosessin eri vaiheita ja oikea kunkin vaiheen testausta. Mallissa korostetaan tarkastuksen ja testauksen merkitystä jokaiseen vaiheeseen liittyen [Gil93].

Ohjelmistotestauksen tarkoitus on löytää ohjelmistossa olevia virheitä. V-mallin ydinajatus on, että testit suunnitellaan testaustasoa vastaavalla suunnittelutasolla ja testauksen tuloksia vertaillaan aina saman tason toteutuksen määrittelyssä tehtyihin dokumentteihin [Hai04]. Kuvan 7 mukaisessa V-mallissa testaus on jaettu kolmeen eri tasoon [Jor94].



Kuva 7. V-malli. [Jor94]

V-mallin mukaisissa testauksen vaiheissa testaus nähdään eri näkökulmista ja ne paljastavat erilaisia virheitä testauksen kohteesta. Testauksen eri vaiheissa paljastuvat virheet voidaan luokitella vaihejakomallin mukaisesti niin, että yksikkötestauksen paljastamat virheet ovat ohjelmointivirheitä. Vastaavasti integrointitestauksessa esiin tulevat virheet johtuvat suunnitteluvirheistä. Lopulta järjestelmätestauksessa paljastuvat virheet voivat pahimmassa tapauksessa johtaa koko prosessin alkuun eli virheellisesti suoritettuun määrittelyvaiheeseen.

Testausvaiheiden menetelmien tulee olla mahdollisimman hyvin kunkin testausvaiheen tyypillisiä virheitä paljastavia, sillä mitä ylemmälle tasolle virheet testausmallissa kulkevat mukana ohjelmassa, sitä kalliimmaksi ja monimutkaisemmaksi niiden korjaaminen tulee [Hai04]. V-malli ei välttämättä lyhennä prosessin loppuvaiheessa havaittavan virheen ja

vaatimusmäärittelyn välistä etäisyyttä, koska tuotteen järjestelmätestaus tapahtuu vasta prosessin lopussa.

3.2.4 Yleistä arviointia vesiputousmallista

Vesiputousmallin rakenne on selkeä, helposti kontrolloitavissa ja jokainen vaihe tuottaa selkeät dokumentaatiot. Toisaalta prosessimalli on kallis, hidas ja joustamaton. Prosessimalli sopii projekteille, joissa vaatimukset eivät muutu projektin aikana ja joissa työ voidaan saattaa loppuun suoraviivaisen prosessin mukaisesti. Vesiputousmallissa vaiheet seuraavat toisiaan alkaen esitutkimuksesta ja päättyen käyttöönottoon ja ylläpitoon, mutta käytännössä edellä mainittu toteutuu käytännössä vain harvoin. Todellisissa kehityshankkeissa vaiheet eivät etene niin suoraviivaisesti eteenpäin. [Pre05]

Peräkkäiset vaiheet ovat yleensä toisistaan riippuvia ja tietyn vaiheen suoritus saattaa paljastaa edellisten vaiheiden virheitä, jolloin joudutaan mahdollisesti palaamaan takaisin edellisiin vaiheisiin ja ne joudutaan tekemään uudelleen virheiden takia. Huonona puolena on myös se, että lopullinen tuote on nähtävissä vasta prosessin lopussa ja muutosten toteuttaminen on silloin hankalaa. Lisäksi lopullisen tuotteen ohjelmointi aloitetaan vasta pitkän määrittely- ja suunnitteluvaiheen jälkeen. Vesiputousmallissa työnkulku ei ole aina paras mahdollinen, sillä vaiheiden peräkkäisyys saattaa aiheuttaa sen, että ohjelmistokehittäjät joutuvat odottamaan edellisen vaiheen valmistumista ennen siirtymistä seuraavaan vaiheeseen. Tämä osaltaan aiheuttaa sen, että vesiputousmallin avulla on hidasta tuottaa ensimmäistäkään toimivaa versiota asiakkaalle. [Pre05]

Ohjelmistoprosessin johtamisen kannalta vesiputousmalli on selkeä ja yksinkertainen. Käytännössä sitoutuminen vesiputousmalliin on kuitenkin mahdotonta ja haittaa tuotteen tekemistä, sillä tuotantoprosessin aikana havaitut virheet on pystyttävä korjaamaan riippumatta virheen tai virheiden tekohetkestä. Myös prosessin aikana kehittyvät hyvät ideat jäävät huomioimatta, koska vesiputousmalli ei ole menetelmänä joustava [Hai04].

Vesiputousmallissa etäisyys vaatimusten määrittelystä testausvaiheeseen on suuri sekä ajallisesti että budjetin kannalta. Myös myöhään aloitettuun testaukseen liittyy paljon riskejä. Tuotannon loppuvaiheessa löytyvät määrittelyvirheet aiheuttavat helposti tuotteen

viivästymisen ja budjetin ylityksen. Suuressa ohjelmistoprojektissa suunnitteluvaihe venyy helposti niin pitkäksi, että tuote ei ehdi suunniteltuun markkinatilanteeseen. Mikäli edellinen pääsee tapahtumaan, tuote vanhenee pitkän suunnittelun tuloksena tai markkinatilanne voi muuttua oleellisesti.

Vaikka vesiputousmallissa onkin ongelmia, on se joka tapauksessa aina parempi tapa toteuttaa ohjelmistokehitystä kuin epämääräinen lähestymistapa, jossa prosessia ei ole määritelty lainkaan [Pre05]. Vesiputousmalli soveltuukin hyvin lähinnä projekteihin, joissa käytetään hyvin tunnettuja tekniikoita ja tuotteen määrittely on vähän muuttuva [McC02]. Vesiputousmallin mukainen toiminta onkin eduksi etenkin monimutkaisissa projekteissa, joissa laatuvaatimukset ovat tärkeämpiä kuin aikataulu tai budjetti.

3.3 Muut perinteiset prosessimallit

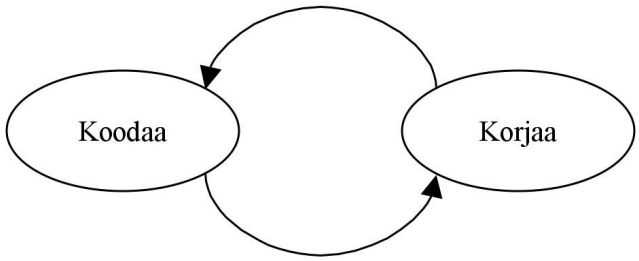
Muita yleisesti käytössä olevia perinteisiä prosessimalleja ovat prototyyppi-, evoluutio- ja spiraalimallit. Lisäksi esitellään lyhyesti RUP-menetelmä, joka on Rational Softwaren kehittämä ohjelmistoprojektin hallintamalli. Muita perinteisiä prosessimalleja edeltää kuitenkin lyhyt kuvaus tilanteesta, jossa varsinaista prosessimallia ei oikeasti ole käytössä lainkaan. Tätä mallia kutsutaan koodaa ja korjaa-malliksi.

3.3.1 Koodaa ja korjaa -malli

Toteuttaessaan ensimmäisiä ohjelmia koodaajat pyrkivät ilman etukäteissuunnittelua vain ohjelman koodaamiseen ja koodauksessa syntyneiden virheiden korjaamiseen. Näin syntynyt koodaa ja korjaa-malli on tuttu kaikille, jotka ovat toteuttaneet tietokoneohjelmia. Mallia ei voida pitää varsinaisena vaihejakomallina. Mikäli projektissa ei ole tehty erityistä valintaa jonkin vaihejakomallin käyttämiseksi, käytetään siinä todennäköisesti koodaa ja korjaa-mallia [Kro07].

Koodaa ja korjaa-mallia käytettäessä ohjelmoijalla voi olla, tai voi olla olematta, jonkinlainen suunnitelma tai määritelmä siitä, mitä ohjelman tulisi tehdä. Tämän jälkeen käytetään satunnaisia suunnittelu-, toteutus- ja testauskäytäntöjä, kunnes julkaisuvalmis ohjelmistotuote on saatu valmiiksi. Malli on harvoin hyödyllinen, mutta siitä huolimatta se on edel-

leen yleisesti käytössä. Kuvassa 8 on havainnoitu mallin pelkistettyä kulkua koodauksen ja korjauksen välillä. [Kro07]



Kuva 8. Koodaa ja korjaa - mallin kulku

Koodaa ja korjaa-mallissa on kuitenkin olemassa omat etunsa. Mallissa ei tehdä ylimääräistä työtä, sillä aikaa ei kulu juuri lainkaan muissa malleissa tapahtuvaan suunnitteluun, dokumentointiin tai esimerkiksi laadunhallintaan. Koodaa ja korjaa-malli vaatii vain vähän ammattitaitoa, sillä jokainen ohjelmointia osaava voi toteuttaa ohjelmia tällä menetelmällä. [Kro07]

Haittapuolina mallissa on, että sitä käytettäessä joudutaan hyvin nopeasti vaikeuksiin, mikäli pyritään toteuttamaan vähänkään suurempaa ohjelmistokokonaisuutta. Koodaa ja korjaa-malli ei määritä prosessia testaamiseen, version- tai tuotteenhallintaan. Ohjelmoijien työnjako on epäselvää, eivätkä toteuttajat todennäköisesti ymmärrä käyttäjän tarpeita. [Rob07]

Mikäli käyttäjiltä tulee muutostarpeita ohjelmakoodiin kesken projektin, ohjelmiston tehty toiminnallisuus ei välttämättä enää vastaakaan käyttäjien tarpeita, minkä seurauksena tehtyä työtä joudutaan heittämään pois tai sitä joudutaan ainakin muuttamaan oleellisesti. Huonosti jäsennetyn koodin muuttaminen tulee aina kalliiksi ja muutaman korjauskierroksen tuloksena syntyvää huonosti jäsentynyttä koodia on vaikeaa ja aikaa vievää muuttaa.

Koodaa ja korjaa-malli on kevyt, eikä sen hallintaan mene aikaa, mutta samalla se on myös epävarma ja vaikea hallita. Ohjelmistoprojektin kulloistakin vaihetta on vaikeaa tietää, minkä seurauksena tuotteen valmistumisajankohtaa on vaikeaa arvioida. Koska järjestelmällistä testausta ei suoriteta välttämättä lainkaan, jää lopputuotteeseen helposti virheitä.

Myös jatkokehittäjät saavat eteensä suuren haasteen, mikäli dokumentointi on jäänyt hoitamatta.

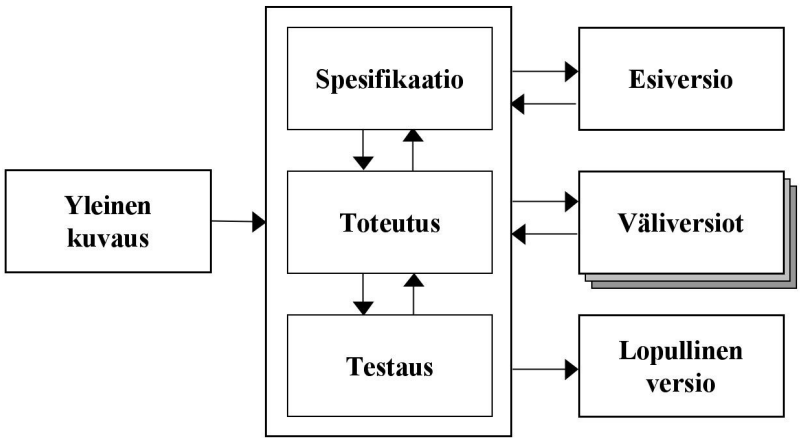
Koodaa ja korjaa-malli on kuitenkin käyttökelpoinen pienissä ohjelmistoprojekteissa, joissa tuotos heitetään projektin jälkeen pois eikä sitä käytetä sen jälkeen. Tällaisia ohjelmia voivat olla esimerkiksi pienet demonstraatio-ohjelmat, prototyypit tai jonkun tietyn tekniikan toteuttamiskelpoisuutta testaavat ohjelmat. [Kro07]

3.3.2 Evoluutiomalli

Evoluutiomalli perustuu prosessiin, jossa suunnitellaan ja kehitetään esiversio asiakkaalle arvioitavaksi toteutettavasta ohjelmasta. Saadun palautteen perusteella ohjelmistoa kehitetään eteenpäin lisäämällä siihen tarpeellisia toimintoja ja parantamalla niitä toimintoja, jotka toimivat käyttäjien mielestä huonosti. Tätä jatketaan kunnes hyväksyttävä järjestelmä on valmis. [Somm00]

Evoluutiomallin tavoitteena on rakentaa tuotteesta ensiksi pelkkä ydin, jota sitten kehitetään edelleen seuraavissa projekteissa [Hai04]. Evoluutiomallin ideana on, että ohjelmistotuotteet ja niiden vaatimukset kehittyvät ajan mittaan joka tapauksessa ja prosessin olisi kyettävä mukautumaan muuttuviin vaatimuksiin ja uusiin tavoitteisiin. Evoluutiomalli koostuu peräkkäisistä sarjoista, joissa ohjelmistotuotteeseen lisätään vähitellen yhä uusia ominaisuuksia. [Gil88]

Evoluutiomallin kehittyminen johtui osaltaan vesiputousmallin toiminnallisesta kankeudesta suunniteltaessa ja toteutettaessa suuria ja vaikeasti määriteltäviä tuotteita. Mallin avulla pyrittiin saamaan vaatimusten määrittely, tuotteen tekeminen ja käyttö ajallisesti lähemmäksi toisiaan, ollen luonteeltaan samaan aikaan sekä iteratiivinen että inkrementaalinen prosessimalli. Evoluutiomallissa suunnittelu, toteutus ja testausvaiheet eivät ole erillisiä perättäisiä vaiheita vaan niitä toteutetaan rinnakkaisesti ja osittain samanaikaisesti ja näin tuotteen saaminen käyttöön nopeutuu. Kuvassa 9 on havainnollistettu evoluutiomallin kulkua. [Somm00]



Kuva 9. Evoluutiomalli. [Somm00]

Jokaisen iterointikierroksen jälkeen tuote on tarpeeksi toiminnallinen, jotta se voidaan julkaista seuraavana versiona. Malli on käyttökelpoinen silloin, kun kaikkia tarvittavia resursseja ei ole kerralla saatavilla. Kerrostamalla tuotteen kehittämistä myös tarvittavia resursseja voidaan paremmin aikatauluttaa. [Pre05]

Evoluutiomallin tutkivassa versiossa analysoidaan asiakkaan vaatimukset ja todelliset tarpeet työskentelemällä aktiivisesti asiakkaan kanssa. Ohjelmiston ensimmäisen version toteutus aloitetaan asiakkaalle keskeisestä kokonaisuudesta, joka on saatu määriteltyä. Toisen version määrittelyä aloitetaan samalla, kun ensimmäistä vielä tehdään. Ohjelmistotuotannon vaiheet lomittuvat versioiden kesken. Tuote rakennetaan osissa ja ensimmäiset versiot ovat toiminnoiltaan puutteellisia. [Somm00]

Asiakkaan annetaan vaikuttaa osien väliseen suunnitteluun ja niiden valmistuksen ajoitukseen. Tuote saatetaan luovuttaa asiakkaalle testausta varten, mutta se ei ole tarkoitettu tuotantokäyttöön. Tuotteen toiminnallisuutta lisätään seuraavissa versioissa, kunnes se lopulta on asiakkaan vaatimusten mukainen. Vaatimusten määrittelyä tarkennetaan ja kehitetään koko prosessin ajan, ja testaus toteutetaan jokaista versiota tehtäessä. [Somm00]

Evoluutiomallin toisessa versiossa rakennetaan koeohjelmistoja, joiden avulla opetellaan ymmärtämään ongelma-aluetta ja sen erityispiirteitä. Testaamalla koeohjelmistoa yhdessä asiakkaan kanssa tuotetaan mahdollisimman kattava toiminnallinen määrittelydokumentti.

Koeohjelmisto ei ole samanlainen kuin lopullinen tuote, vaan se sisältää ainoastaan vaatimusten määrittelyprosessin kannalta tärkeät ja oleelliset osat. Kun toiminnallinen määrittelydokumentti on tuotettu, voidaan aloittaa itse tuotteen tekeminen. [Somm00]

Evoluutiomallin huonona puolena pidetään sitä, että projektin alussa ja sen aikana, on mahdotonta täysin tietää projektin lopputulosta ja kauanko kestää ennen kuin saadaan aikaan hyväksyttävä tuote. Projektin aikana ei myöskään tiedetä, kuinka monta iteraatiokierrosta tarvitaan hyväksyttävän lopputuotteen toteuttamiseen. Toisaalta evoluutiomallissa asiakas näkee lopputuotteen jatkuvan edistymisen ja voi varmistua tilatun tuotteen valmistumisesta. Huonona puolena evoluutiomallin mukaisessa kehittämisessä on myös se, että aiemmin kehitetyt osat on testattava aina uudelleen jokaisen uuden version yhteydessä, jotta voidaan varmistua kokonaisuuden toimivuudesta.

Pienten inkrementtien käytöstä johtuen pitkän tähtäimen arkkitehtuurisuunnittelu puuttuu. Tämä saattaa johtaa helposti siihen, että koodista tulee jäsentymätöntä ja toteutettavan ohjelmiston kokonaisarkkitehtuuri rapautuu. On myös mahdollista, että tilapäisiksi tarkoitetuista alkuvaiheiden ratkaisuista ei päästäkään eroon. Myös versionhallinnan kannalta evoluutiomalli on haasteellinen. Lisäksi dokumentointi on työlästä, koska jokaisen version pienetkin muutokset tulee korjata kaikkiin dokumentaatioihin. [Hai04]

Evoluutiomallia pidetään sopivana laajoihin projekteihin sekä tuotekehitykseen. Asiakas näkee osia ohjelmistosta valmiina ja voi puuttua mahdollisiin ongelmakohtiin aikaisemmin. Mallia suositellaan käytettäviksi vain pienissä tai keskisuurissa ohjelmistotuotteissa, joiden käytön elinkaari on suhteellisen lyhyt [Somm00].

Evoluutiomallia pidetään käytännöllisenä tilanteissa, joissa vaatimukset muuttuvat nopeasti tai niitä ei ymmärretä prosessin alussa; tilanteissa, joissa asiakas ei halua sitoutua johonkin ennalta määriteltyyn vaatimuskokonaisuuteen, ja tilanteissa joissa asiakkaat eivätkä kehittäjät tunne ongelmakenttää tarpeeksi hyvin. [McC02]

3.3.3 Prototyyppimalli

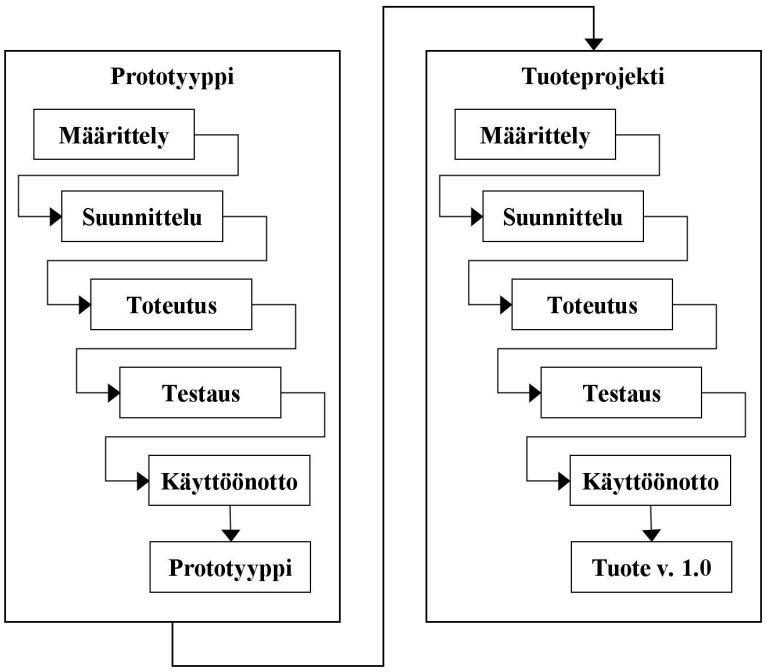
Prototyyppimallilla tarkoitetaan yleisesti työskentelymallia, jossa jotakin ohjelmistotuotteen ominaisuutta tai piirrettä kokeillaan luonnoksella ennen varsinaisen tuotteen rakentamista [Hai04]. Prototyyppien tarve liittyy useimmiten vaikeuteen määritellä asiakkaan vaatimuksia ohjelmistotuotteen suhteen, nykyiseen tilanteeseen ei olla tyytyväisiä ja uuden ohjelmistotuotteen avulla halutaan päästä entistä parempaan tilanteeseen. Useasti kuitenkin asiakkaalla on vaikeuksia määritellä, millainen parempi tilanne tarkalleen olisi. Prototyyppien avulla tapahtuva esimerkkien antaminen voi auttaa tilanteen hahmottamisessa. [Vli00]

Prototyyppi on hyvin rajoitetulla toiminnallisuudella varustettu versio kehitettävästä ohjelmistosta. Yleinen esimerkki prototyypistä on, että ohjelmistolle on tehty valmis käyttöliittymä, mutta liittymän takana oleva sovelluslogiikka ja tietojen haku- ja tallennustoiminnot on vielä toteuttamatta. Kehitettävien ohjelmistojen prototyypit pyritään toteuttamaan mahdollisimman edullisesti ja nopeasti. [Vli00]

Toteutettavien toimintojen osalta eroavaisuudet todelliseen järjestelmään voivat koskea esimerkiksi tietojen hakua ja tallennusta. Valmiissa prototyypissä ei välttämättä tarvitse olla mitään sovelluslogiikkaa tietojen hakuun ja tallennukseen, mikäli mallinnuksen kohteena ovat esimerkiksi käyttöliittymän erikoispiirteet. Mikäli sovelluslogiikkaa kuitenkin toteutetaan jo prototyyppiin, voi prototyyppi erota varsinaisesta järjestelmästä siten, että jotkin järjestelmän tehokkuusmääreet kuten nopeus ja vikasietoisuus eivät ole samanlaisia kuin itse varsinaisessa järjestelmässä. [Vli00]

Prototyypin toteuttamisen jälkeen on tarjolla kaksi pääsääntöistä vaihtoehtoa, joiden mukaan ohjelmistokehitysprojekti voi edetä. Ensinnäkin prototyypin valmistuttua prototyyppiä käytetään varsinaisen järjestelmän määrittelyyn. Varsinainen järjestelmä toteutetaan alusta alkaen uudelleen ja prototyyppi hylätään. Toisessa tapauksessa prototyyppi kehitetään valmiiksi järjestelmäksi. [Hai04]

Kuvassa 10 esitetään esimerkki ensin mainitusta vaihtoehdosta. Tässä tapauksessa sekä prototyypin että varsinaisen järjestelmän suunnittelu ja toteutus etenee tapaan, jossa varsinaisen järjestelmän määrittelyssä on käytettävissä prototyyppiä varten tehdyt määrittelyt.



Kuva 10. Prototyyppimallin eteneminen. [Hai04]

Usein vaihtoehto, jossa prototyyppi toimii varsinaisen järjestelmän alustana, on houkuttelevampi. Tällöin valmis prototyyppi voidaan käyttää hyväksi kehitettäessä valmista järjestelmää. Molemmilla vaihtoehdoissa on kuitenkin omat hyvät ja huonot puolensa. Ohjelmiston ylläpitokustannukset voivat kohota huomattavasti, mikäli nopealla aikataululla koottua ja huonolla ohjelmointitavalla toteutettua prototyyppiä käytetään varsinaisen järjestelmän runkona [Vli00]. Prototyyppimalli sopii erityisesti sellaisten projektien prosessimalliksi, joissa asiakkaalle on saatava nopeasti jokin konkreettinen tulos. Mallissa edetään iteraatiokierroksittain, joihin jokaiseen kuuluu määrittely-, suunnittelu-, toteutus- ja testausvaiheet. [Hai04]

Ensimmäisellä iteraatiokierroksella tehdään määritellyt yleiset ominaisuudet, jonka jälkeen prototyyppi testataan, evaluoidaan ja määritellään puutteet ja valmiit ominaisuudet. Määrittelyvaiheessa havaitut puutteet suunnitellaan ja toteutetaan projektista toinen prototyyppi, joka taas testataan ja evaluoidaan. Tätä jatketaan siihen saakka, kunnes prototyyppi on saavuttanut tavoitteensa.

Mallin yhtenä etuna pidetään sitä, että vuorovaikutus asiakkaan kanssa on tiivistä, jolloin asiakkaan vaatimukset on helppo kerätä ja pitää ne ajan tasalla prototyypin edetessä. Ongelmana on pidetty muun muassa sitä, että asiakas saattaa luulla keskeneräistä tuotetta valmiiksi [Hai04]. Valmista prototyyppiä halutaan usein yhä parantaa ja parantaa, jolloin iteraatiokierroksien määrä yhä lisääntyy ja prosessin päätökseen saattaminen venyy [Pre05]. Prototyyppimallin hyviksi käyttötarkoituksiksi on esitetty tilanteita, joissa asiakkaan antamat määrittelyt ovat epäselviä sekä järjestelmät, joissa käyttöliittymillä on keskeinen rooli. Asiakkaan ja järjestelmän toteuttajien on kuitenkin syytä olla tietoisia prototyyppimallin käytön mahdollisuuksista ja riskeistä [Vli00].

Prototyyppien ongelmana on se, että monesti toimivalta näyttävä prototyyppi on koottu kasaan palasista, jotka yhdessä eivät muodosta toimivaa kokonaisuutta. On hyvin tavallista, että prototyypin tekeminen lisää paineita saada prototyyppi valmiiksi, joten toteutuksessa tehdään usein kompromisseja, jotta prototyyppi saataisiin toteutettua nopeasti. Myös aikataulun laatiminen voi osoittautua mahdottomaksi tehtäväksi, sillä kaikki halutut ominaisuudet eivät ole tiedossa projektia aloitettaessa. Prototyyppimalli voi kuitenkin olla tehokas, mikäli yhteiset käytännöt saadaan sovittua kaikkien osapuolien kesken eikä projektin kokonaiskeston arviointi ole erityisen tärkeätä. [Pre05]

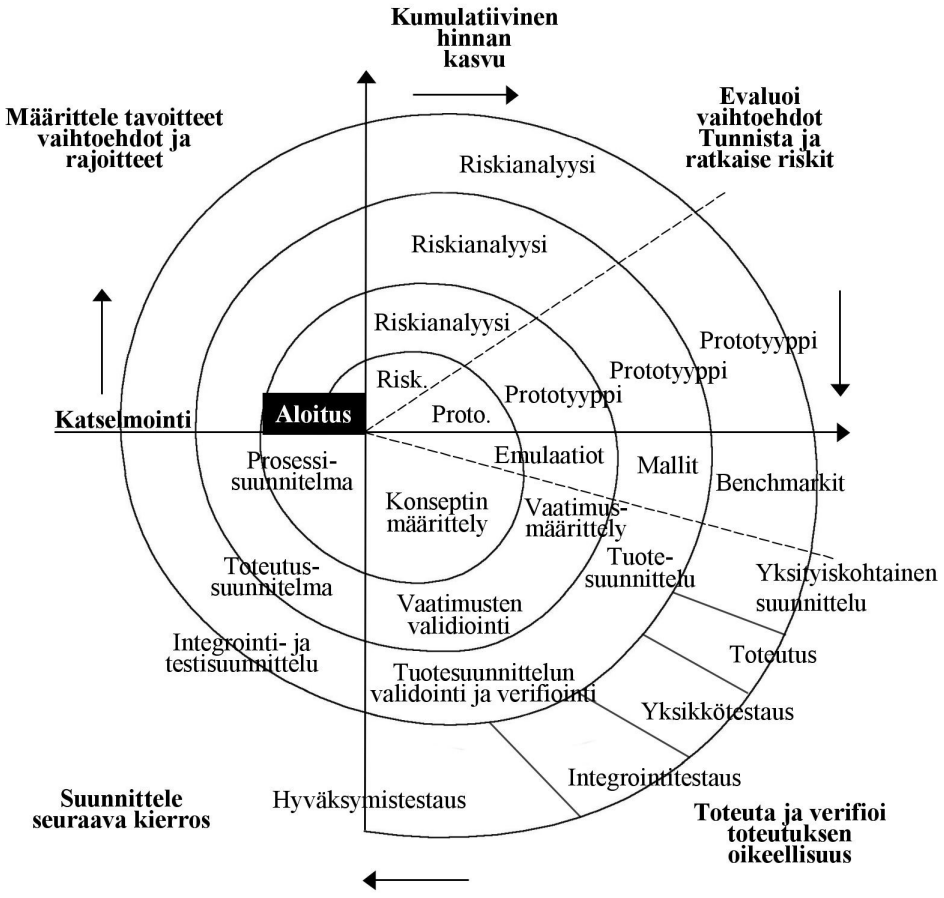
3.3.4 Spiraalimalli

Spiraalimalli on Barry Boehmin vuonna 1988 esittelemä vaihejakomalli, joka kehitettiin jo olemassa olevien prosessimallien parannukseksi yhdistäen evoluutiomallin iteratiivisen luonteen ja vesiputousmallin kontrolloidun ja systemaattisen lähestymistavan [Pre05]. Sen tarkoituksena oli myös yhdistää vesiputous- ja prototyyppimallin parhaat piirteet sekä korjata niissä ilmenneitä puutteita. Spiraalimalli on riskien ohjaama prosessimalli ja lähtökohtana mallin kehittämiselle on ollut riskien tunnistaminen ja riskien toteutumisen estäminen. [Somm00]

Spiraalimallin prosessille on tunnusomaista syklinen prosessi, joka johtaa ohjelmiston inkrementaaliseen kasvuun [Somm00]. Kussakin syklissä edellisissä sykleissä toteutettua ohjelmistotuotettua laajennetaan ja/tai tarkennetaan. Riskianalyysi ja riskien hallinta ovat

olennainen ja tärkeä osa spiraalimallia ja ne ovat oma vaiheensa joka kierroksella. Spiraalimallissa jokainen spiraalin kierros sisältää neljä toimenpidettä, ensimmäiseksi tavoitteiden määrittelyn, vaihtoehdot ja rajoitteet, toiseksi vaihtoehtojen arvioinnin, riskianalyysin ja riskien hallinnan, kolmanneksi iteraatioon kuuluvan tuotoksen toteuttamisen ja varmistamisen, että se on tehty oikein ja neljänneksi seuraavan iteraatiokierroksen suunnittelun. [Somm00]

Jokaisessa syklissä vaiheet suoritetaan samassa järjestyksessä, eikä syklien määrää ole rajoitettu. Kuvan 11 mukaisesti spiraalimallissa lähdetään spiraalin keskeltä ja päädytään ulommas. Etäisyys keskipisteestä kuvaa projektin kumulatiivisia kustannuksia, vastaavasti mitä kauempana keskipisteestä ollaan, sitä valmiimpi on tuotteen malli [McC02].



Kuva 11. Spiraalimalli. [Somm00]

Spiraalimalli ymmärretään usein väärin, sillä mallin luullaan olevan vain sarja peräkkäisiä vesiputousmalleja, ohjelmistoprojekti seuraa yksittäistä spiraalikierrosta ja kaikki edellä esitetyssä kuvassa esitetyt toimenpiteet on aina tehtävä, eikä mallissa voida palata taaksepäin korjaamaan tai muuttamaan aikaisemmin tehtyjä päätöksiä. [Boe00]

Spiraalimallin esittelemisen jälkeen mallia on käytetty useissa ohjelmistoprojekteissa ja saatujen kokemusten perusteella prosessimallia on kehitetty edelleen. Alkuperäisen mallin vaikeutena nähtiin vaikeus asettaa tavoitteita ja rajoitteita. Spiraalimallia parannettiin vastaamaan näihin tarpeisiin. Parannettua spiraalimallia kutsutaan WinWin-spiraalimalliksi ja se laajentaa alkuperäistä mallia kolmella toiminnolla. Ensimmäiseksi pyritään tunnistamaan seuraavan iteraatiokierroksen kohteena oleva asiakas. Toiseksi pyritään tunnistamaan asiakkaan tavoitteet ja kolmanneksi hyväksytään seuraavan iteraatiokierroksen tavoitteet. [Boe94]

Spiraalimallia käytetään yleensä laajojen järjestelmien kehittämiseen, sillä jatkuva riskien analysointi auttaa sekä järjestelmän kehittäjää että asiakasta hallitsemaan ohjelmistoprosessia. Siitä huolimatta mallin ongelmaksi on nähty uskottavuuden luonti asiakkaan suuntaan. Samalla mallin käyttö edellyttää käyttäjältään kykyä ja halua panostaa riskienhallintaan. Eräitä spiraalimallin huonoja puolia ovat sen monimutkaisuus, sopeutuminen talousnäkökohtiin, kehityksen eteneminen sykleissä saattaa aiheuttaa ongelmia sopimusneuvotteluissa ja sopimuksien määrittelyissä. [Pre05]

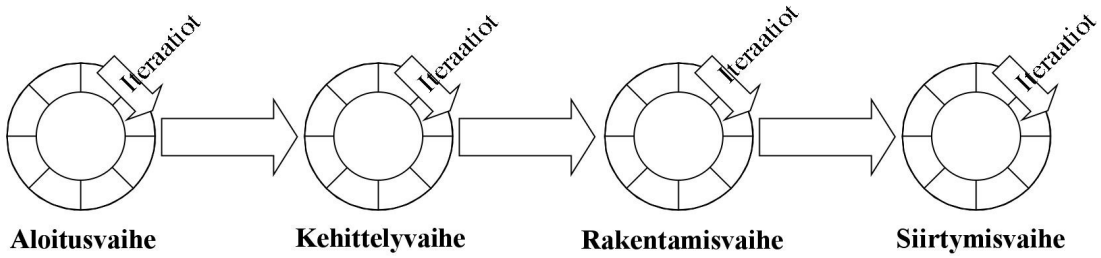
Eräs spiraalimallin tärkeimmistä eduista on se, että kustannusten noustessa riskit vähenevät. Mitä enemmän aikaa ja rahaa käytetään, sitä pienemmäksi riski muodostuu. Riskeihin painottuvana mallina se antaa varoittaa ylitsepääsemättömistä riskeistä, eikä mahdottomaksi osoittautuva projekti tule liian kalliiksi. [McC02]

3.3.5 RUP / Rational Unified Process

Rationalin Unified-prosessi (Rational Unified Process, RUP) on Rational Softwaren kehittämä ohjelmistoprojektin hallintamalli ollen samalla muodollisesti ja tarkasti määritelty prosessi. RUP perustuu ensiksikin parhaiksi koettuihin työskentelytapoihin, toiseksi yhtenäisten, Rationalin valmistamien projektinhallintatyökalujen käyttöön ja kolmanneksi Ra-

tionalin konsultointipalveluiden käyttöön. Prosessin kehittäneellä yhtiöllä onkin selvä kaupallinen intressi koskien RUP-mallia. RUP:ia voidaan soveltaa myös käyttäen pelkästään mallin parhaita työskentelytapoja ja niiden pohjalta on myös rakennettu RUP:ia soveltavia malleja. RUP-prosessi on myös prosessikehys mikä tarkoittaa, että sitä voidaan muokata erilaisiin organisaatioihin ja tarpeisiin sopivaksi [Kru01].

RUP-prosessia voidaan pitää evoluutio- ja spiraalimallien sovelluksena, joka perustuu peräkkäisiin iteraatioihin, joista jokainen muodostaa oman pienen vesiputouksensa [Kre05]. RUP:n mukainen prosessi koostuu neljästä tarkkaan suunnitellusta vaiheesta, jotka seuraavat lineaarisesti toisiaan. Kuva 12 havainnollistaa RUP:n vaiheiden sijoittumista toisiinsa nähden prosessissa, jossa ensimmäisenä on aloitusvaihe. Sitä seuraavat kehittely-, rakentamis- ja lopuksi siirtymävaihe. Jokaisen vaiheen lopettaa erityinen virstanpylväs. [Kru01]



Kuva 12. RUP:n vaiheet ja iteraatioiden sijoittuminen. [Kru01]

Vaiheet ovat RUP:ssa keskeisessä asemassa, kun mallissa kuvataan ohjelman elinkaarta. Rational määrittelee vaiheen siten, että se on kahden virstanpylvään välinen aika, jolloin määritelty osa tavoitteista on saatu toteutettua ja päätös seuraavaan vaiheeseen siirtymisestä saadaan tehtyä [Kre05]. Vaiheiden aikana suoritetaan tiettyjä tehtäväkokonaisuuksia, joita ovat esimerkiksi vaatimusten määrittäminen, analyysi ja suunnittelu, toteutus, testaus, sekä käyttöönotto. Toiminnot vastaavat pääosin vesiputousmallin vaiheita. [Kru01]

Aloitusvaiheen aikana keskitytään määrittämään projektin laajuus, kuvaamaan projektin kannalta pahimpia riskejä, kuvailemaan alustavia liiketoimintamalleja ja miettimään liiketoiminnan kannalta, kannattaako koko projektia edes lainkaan aloittaa. Vaiheen tuloksena ovat projektia yleisellä tasolla kuvaava dokumentti, projektisuunnitelma, alustava riskien

arviointi sekä alustava käyttötapausmalli. Aloitusvaiheen päättävästä virstanpylväästä käytetään nimitystä elinkaaritavoite. Seuraavana vuorossa olevan kehittelyvaiheen tehtäviin kuuluu määritellä esimerkiksi projektin perusarkkitehtuuri, laatukriteerit ja toiminnallisuus. Suurin osa vaatimuksista on tässä vaiheessa jo selvillä. Rakennettava järjestelmä analysoidaan ja suunnitellaan ja toteutettavasta järjestelmästä voidaan tehdä useita prototyyppejä. Tämän vaiheen päättävää virstanpylvästä kutsutaan elinkaariarkkitehtuuriksi. [Kru01]

Kehittelyvaiheen jälkeen on vuorossa rakentamisvaihe, jonka aikana varsinaisesti toteutetaan ohjelmiston ominaisuudet esimerkiksi uudelleenkäytettävien komponenttien muodossa. Vaiheen aikana tuotetaan yksi tai useampia ohjelmistoversioita, ennen kuin siirrytään seuraavaan vaiheeseen [Abr02]. Rakentamisvaiheessa kaikki vaatimukset toteutetaan iteraatioiden avulla lopulliseksi järjestelmäksi. Kukin iteraatio tuottaa entistä täydellisemmän ja valmiimman ohjelmistotuotteen, sisältäen myös tarvittavan testauksen ja dokumentoinnin. Lopullinen tulos rakentamisvaiheesta on valmis ohjelmistokokonaisuus. Tämän vaiheen päättävää virstanpylvästä kutsutaan lähtötoimintakyvyksi. [Kru01]

Viimeisessä eli siirtymävaiheessa järjestelmä laitetaan toimituskuntoon, asennetaan ja otetaan käyttöön tuotantoympäristössä. Olennaisia vaiheen aikana suoritettavia toimintoja ovat esimerkiksi käyttäjien ja ohjelmiston ylläpitäjien koulutus, mahdollisten virheiden ja ongelmien korjaaminen, sekä ohjelmiston laadun testaaminen suhteessa odotuksiin. Vaiheen aikana tuotetaan ohjelmistosta uusia versioita, mikäli se koetaan tarpeelliseksi. Siirtymävaiheen päättävää virstanpylvästä kutsutaan tuotejulkistukseksi. [Kru01]

RUP-mallin soveltamista sellaisenaan pidetään liian raskaana moniin tapauksiin, niinpä menetelmää hyödynnetään myös kevyempänä versiona, jolloin kehitysprosessissa on alettu soveltamaan yhä enemmän ketterien menetelmien käytäntöjä. On myös mietitty, mitä RUP:sta tulisi tällöin ottaa kehitysprosessiin mukaan ja mitä olisi syytä jättää prosessin ja sen toteutuksen ulkopuolelle. RUP:n puutteena on pidetty sitä, että vaikka metodologian kerrotaan olevan vapaasti sovellettavissa, niin kunnon ohjeistusta erityyppisiin sovellustapoihin ei ole [Abr02].

4 KETTERÄT MENETELMÄT

Tässä luvussa esitellään lyhyesti ketterien menetelmien pääperiaatteita ja tunnetuimpia ketteriä menetelmiä. Luvussa esiteltävät ketterät menetelmät ovat Scrum, Crystal-metodologiaperhe, ASD eli Adaptive Software Development, DSDM eli Dynamic Systems Development Method ja FDD eli Feature-Driven Development. Menetelmät on pyritty valitsemaan siten, että niille on jo muodostunut vakiintunut asema ketterien menetelmien keskuudessa. Tunnetuimpana näistä uusista menetelmistä pidetään Kent Beckin vuonna 1999 esittelemää Extreme Programming-mallia, eli lyhyesti XP:tä [Bec00]. Sitä pidetään ketterien ohjelmistokehitysmenetelmien edelläkävijänä ja kehityksen alkuna ja se esitellään laajemmin omassa luvussaan.

4.1 Yleistä ketteristä menetelmistä

90-luvun loppupuolella useat uudet ohjelmistokehitysmetodologiat saivat julkisuudessa yhä enemmän huomiota. Ne olivat yhdistelmiä vanhoista ja uusista, sekä muunnelluista vanhoista ideoista. Näille metodologioille oli yhteistä kehitysryhmän ja asiakkaan kesken tapahtuva yhteistyö, kasvokkain tapahtuva kommunikointi, säännöllinen uuden vastineen tuottaminen asiakkaan investoinneille, itseorganisoituvat ja tiiviit kehitysryhmät, sekä ohjelmointikäytännöt ja kehitysryhmät, jotka kykenevät vastaamaan nopeasti muuttuviin vaatimuksiin [Bec01].

Vuonna 2001 Yhdysvalloissa pidettiin työpaja, jossa monet näiden uusien menetelmien kehittäjistä ja käyttäjistä kokoontuivat miettimään, mitä yhteistä näissä menetelmissä oli. Tässä yhteydessä otettiin käyttöön termi ”ketterä”, joka kuvaa näille menetelmille yhteisiä piirteitä ja kykyä mukautua vallitsevaan tilanteeseen ja tuottaa käyttökelpoisia ohjelmistoja yksinkertaisesti ja nopeasti. Samalla myös muotoiltiin ja julkaistiin ketterän ohjelmistokehityksen manifesti, Agile Software Development Manifesto, joka määritteli ketterälle ohjelmistokehitykselle arvot ja periaatteet. [Abr02] [Bec01]

Ohjelmistokehitykselle asetetaan kovia vaatimuksia kustannusten ja aikataulujen osalta. Lisäksi järjestelmien määrittelyt muuttuvat jatkuvasti. Nämä ilmiöt koskevat erityisesti Internetin selainpohjaisia ohjelmistoja. Tämän vuoksi kiinnostus nopeampia ja joustavia ohjelmistokehitysmenetelmiä kohtaan on kasvanut jatkuvasti. Ketterien ohjelmistokehitysmenetelmien avulla kehittäjät voivat kasvattaa tuottavuutta säilyttäen samalla ohjelmistojen laadun ja muokattavuuden korkeana [Mau02]. Ne pyrkivät vastaamaan ohjelmistotuotannon haasteisiin tarjoamalla uuden, perinteisiä menetelmiä kevyemmän, joustavamman ja tehokkaamman lähestymistavan, jolloin ohjelmistoa kehitetään iteratiivisesti eli vähän kerrallaan.

Ketterissä ohjelmistoprosesseissa käytetään paljon käytäntöjä, joiden on todettu vaikuttavan positiivisesti ohjelmiston laatuun. Amblerin [Amb05] mukaan ketterät ohjelmistoprosessit tuottavat laadultaan parempia ohjelmistoja kuin perinteiset prosessimallit. Huo [Huo04] löytää tälle havainnolle kolme syytä. Ensinnäkin laadunvarmistus on sisällytetty osaksi kehityskäytäntöjä. Toiseksi laaduntarkistuksia tehdään paljon useammin kuin perinteisissä ohjelmistoprosesseissa ja kolmanneksi laadunvarmistus aloitetaan jo projektin alkuvaiheessa. Tiivis yhteistyö asiakkaan kanssa ja vaatimuksiin reagoiminen vähentää riskiä, että tuote ei vastaa asiakkaan odotuksia. Kehitystyö tapahtuu iteratiivisesti ja jokaiseen iteraatioon liittyy testausta. Lisäksi laadunvalvonta alkaa jo ensimmäisessä iteraatiosta.

Tähän mennessä ketteriä ohjelmistokehitysmenetelmiä on määritelty ja esitelty kymmenkunta erilaista. Keskeisimpiä yhteisiä piirteitä niissä ovat iteratiivinen ohjelmistokehitys sekä asiakkaan tiivis mukanaolo projektissa. Eroja ovat esimerkiksi keskittyminen ohjelmiston elinkaaren eri osiin tai tarkkuus, jolla mallin käytännön soveltamista on ohjeistettu. Osa malleista on luonteeltaan hyvin abstrakteja, osan koostuessa konkreettisista ohjelmointityön käytännöistä. [Abr02]

4.1.1 Ketterien menetelmien yleiset periaatteet ja perusarvot

Ketterän ohjelmistokehityksen yleiset periaatteet ja perusarvot, joita kaikki yksittäiset menetelmät noudattavat, on esitetty ketterän ohjelmistokehityksen manifestissa [Bec01]. Tämän manifestin tärkein osa on julkilausuma, jossa määritellään yhteiset ketterän ohjelmis-

tokehityksen arvot. Agile Manifestossa ketterien menetelmien arvomaailman määrittävät neljä teesiä, jotka ovat vapaasti suomennettuina seuraavanlaisia: [Amb05] [Bec01]

- Yksilöitä ja kommunikaatiota arvostetaan enemmän kuin jäykkiä prosesseja ja työkaluja.
- Toimivaa sovellusta arvostetaan enemmän kuin perusteellista dokumentaatiota.
- Joustavaa yhteistyötä asiakkaan kanssa arvostetaan enemmän kuin sopimusneuvotteluita.
- Muutoksen hyväksymistä ja hallintaa arvostetaan enemmän kuin tiukkaa suunnitelman noudattamista.

Agile Alliance [Bec01] esittää verkkosivuillaan, että projektin joustavuus tulisi nostaa viidenneksi projektiin vaikuttavaksi muuttujaksi edellä esitettyjen arvojen lisäksi. Mahdollisuus muuttaa ohjelmiston määrittelyjä ja toimintojen priorisointia ilman suuria kustannusvaikutuksia koetaan olevan asiakkaalle merkittävä kilpailuetu. Agile Alliance korostaa yksilön ja vuorovaikutuksen merkitystä. Yksilöitä rohkaistaan toimimaan tiiviissä yhteistyössä keskenään, erityisesti asiakkaan suuntaan. Ketterät menetelmät korostavat kasvokkain tapahtuvaa ja muuta välitöntä viestintää, kirjallisen viestinnän sijaan. Vastaavasti tarkasteltaessa projektiryhmän tehokkuutta ja tuloksia, nähdään toteutetun ohjelmiston olevan ainoa luotettava mittari.

Agile Manifestossa korostetaan yhteistyötä asiakkaan kanssa. Ketterissä menetelmissä tiiviillä yhteistyöllä asiakkaan kanssa kyetään vähentämään tehtyjen sopimusten merkitystä vaatimusten asettajina ja samalla pitämään asiakas tyytyväisenä [Bec01]. Uuden ohjelmiston vaatimukset eivät voi olla täysin selvillä ennen kuin käyttäjä on käyttänyt järjestelmää. Lisäksi vaatimukset muuttuvat jatkuvasti. Niinpä tehtyjä suunnitelmia pitäisi voida muuttaa aina tarpeen tullen ja saada ne heijastamaan ohjelmiston nykytilannetta [Coc02]. Abrahamsson toteaa, että kehitystyöhön osallistuvien tulisi olla varautuneita tekemään muutoksia ja että olemassa olevassa sopimuksessa määriteltyjen välineiden tulisi mahdollistaa ja tukea muutosten suorittamista [Abr02].

Perusarvojen lisäksi Agile Alliance on määritellyt myös joukon periaatteita, jotka tukevat Agile Manifeston arvoja ja auttavat konkretisoimaan Agile Manifeston ajattelutapaa. Nämä periaatteet ovat vapaasti suomennettuina seuraavanlaisia: [Bec01]

1. Tärkeintä on asiakastyytyväisyys, joka saavutetaan julkaisemalla jatkuvasti ja aikaisin uusia hyödyllisiä versioita ohjelmistosta.
2. Muuttuvat vaatimukset hyväksytään ja otetaan vastaan tervetulleina, myös kehityksen loppuvaiheessa. Ketterät menetelmät valjastavat muutoksen asiakkaan kilpailueduksi.
3. Toimiva ohjelmisto toimitetaan säännöllisesti, muutaman viikon tai kuukauden välein, mieluummin usein kuin harvoin.
4. Liiketalouden ammattilaisten ja ohjelmistokehittäjien on työskenneltävä tiiviisti yhdessä päivittäin koko projektin ajan.
5. Projektit rakennetaan motivoituneiden yksilöiden ympärille, joille annetaan ympäristö ja tarvittava tuki, sekä luotetaan siihen, että he saavat työn tehtyä.
6. Kaikkein tehokkain tapa välittää tietoa kehitysryhmälle ja kehitysryhmän sisällä on kasvokkain tapahtuva keskustelu.
7. Toimiva ohjelmisto on projektin etenemisen ensisijainen mittari.
8. Ketterät menetelmät suosivat kestävää kehitystä. Asiakkaiden, kehittäjien ja käyttäjien tulisi kyetä pitämään jatkuvasti yllä tasainen työtahti.
9. Jatkuva panostaminen teknisen toteutuksen laatuun ja hyvään sovelluksen arkkitehtuurisuunnitteluun lisäävät ketteryyttä ja mukautumiskykyä.
10. Yksinkertaisuus, taito maksimoida tekemättömän työn määrä, on oleellista. Pyritään aina yksinkertaisimpaan mahdolliseen ratkaisuun, joka täyttää tarpeen.
11. Parhaat sovellusarkkitehtuurit, määrittelyt ja toteutukset saavat alkunsa itseorganisoituvista kehitysryhmistä.
12. Tasaisin väliajoin kehitysryhmä miettii miten voisi tulla entistä tuottavammaksi, sopeuttaen ja muokaten toimintaansa sen mukaisesti.

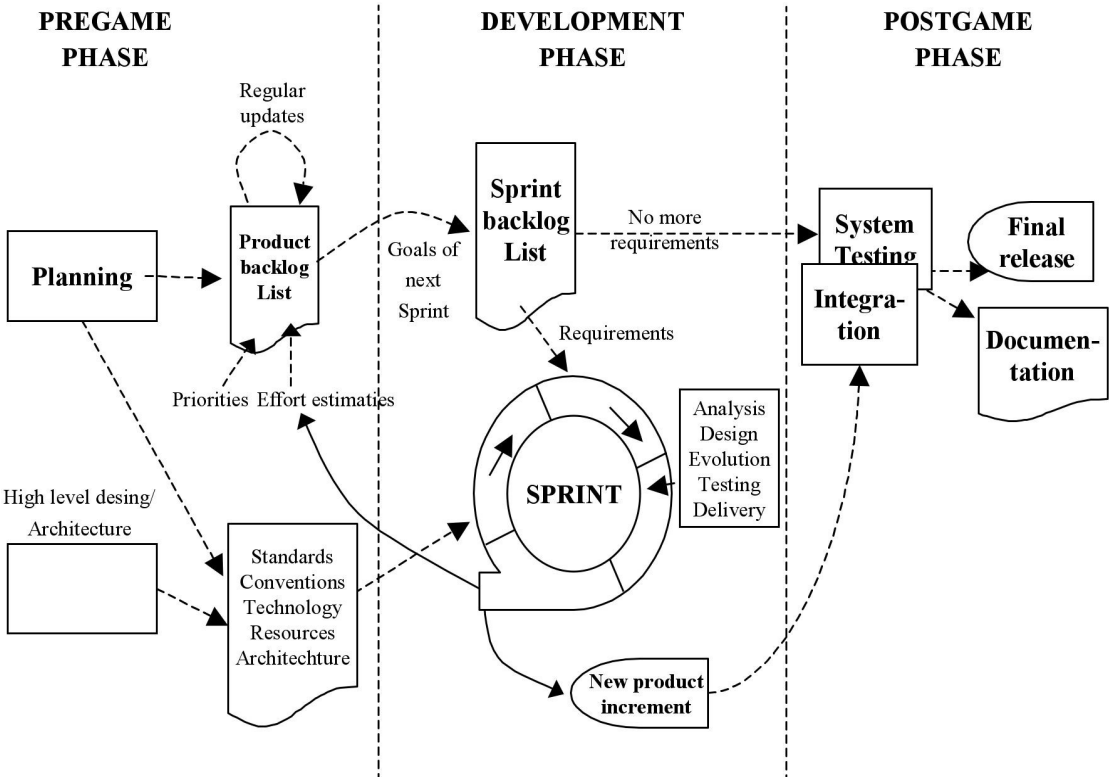
Ketterän kehityksen menetelmien erottamiseksi perinteisistä Abrahamsson [Abr02] koostaa edellä kuvatut ketterän kehityksen arvot ja periaatteet seuraavanlaisiksi tunnusmerkeik-

si. Ketterä ohjelmistokehitys on inkrementaalista, jossa tapahtuu pieniä versiojulkaisuja tiheään tahtiin. Kehitystyö tapahtuu ohjelmistokehittäjien ja asiakkaan toimiessa koko ajan yhteistyössä, se on suoraviivaista ja sopeutuu mahdollisiin muutoksiin nopeasti.

4.2 Scrum

Jeff Sutherland kehitti Scrum-menetelmän [Schw06] 1990-luvun alkupuolella ja sitä ovat myöhemmin kehittäneet Beedle ja Schwaber. Scrum on alun perin rugby-termi, missä se tarkoittaa pallon tuomista takaisin peliin. Scrum ei ole tarkka prosessimalli, vaan kehys jolla voidaan kontrolloida ohjelmistojen toteutukseen liittyvää epävarmuutta [Scr07]. Menetelmän perusperiaatteita ovat muutoksiin sopeutuminen, kommunikoinnin maksimointi, toteutustyön inkrementaalisuus ja runsas ohjelmistojen testaaminen. Scrum on luonteeltaan iteratiivinen ja inkrementaalinen menetelmä, joka on tarkoitettu käytettäväksi hallitsemattomassa ympäristössä [Ris00] [Scr07].

Scrum ei tarkasti määrittele miten vaatimusten keruun, suunnittelun tai ohjelmiston toteuksen tulee tapahtua, vaan se antaa ohjeet projektien organisoinnista yleisellä tasolla Scrum-prosessia noudattamalla. Se jakautuu kolmeen vaiheeseen, ensinnäkin alustavaan vaiheeseen (Pregame phase), toiseksi tuotantovaiheeseen (Development phase) ja lopuksi jälkivaiheeseen (Postgame phase). Alustava vaihe jakautuu suunnittelu- ja arkkitehtuurivaiheisiin. Suunnitteluvaiheessa tehdään kaikkien projektin osapuolten yhteistyönä tekemättömien töiden lista (Product backlog list), joka on eräänlainen yhdistelmä vaatimusmäärittelyä ja projektisuunnitelmaa. Tähän listaan on koottu kaikki tarvittavat työt ohjelmistotuotteen onnistuneen lopputilan saavuttamiseksi. [Mou07] Scrum-menetelmän etenemistä havainnollistetaan kuvassa 13.



Kuva 13. Scrum-menetelmän eteneminen. [Abr02]

Scrum-menetelmän tuotantovaiheessa järjestelmää kehitetään urheilutermiä käyttäen sprinteissä, jotka ovat iteratiivisia kierroksia. Sprinttien avulla lähdetään purkamaan tekemättömien töiden listaa. Jokainen uusi iteraatiokierros perustuu edellisille iteraatiokierroksille. Jokaisen sprintin alussa pidetään sprintin suunnittelupalaveri, jossa asiakas tai muu tuotteen omistaja määrittelee tärkeysjärjestyksen työlistassa luetelluille töille. Tämän jälkeen Scrum-kehitysryhmä valitsee tulevan sprintin aikana tehtävissä olevat työt. Työt siirretään ohjelmistotuotteen työlistasta sprintin työlistaan. Yhden sprintin kesto vaihtelee yhdestä viikosta yhteen kuukauteen. [Abr02] [Ris00]

Iteraation aikana toistuvat perinteiset ohjelmistoprojektin vaiheet kuten vaatimusten keruu, jota kutsutaan tässä tapauksessa vaatimusten tarkennukseksi, tekninen suunnittelu, ohjelmiston toteutus, testaus ja valmiin ohjelmistotuotteen toimittaminen asiakkaalle. 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 list). Tämän listan tehtäviä ei enää muuteta vaikka vaatimukset muuttuisivatkin. Muuttuneet asiat otetaan huomioon vasta seuraavissa iteraatiossa. [Abr02] [Ris00]

Iteraation aikana aikataulu ei jousta, sillä sille on asetettu kiinteä määräaika. Mikäli iteraation aikataulu uhkaa pettää, tingitään sillä kertaa toteutettavien työtehtävien määrästä. Tuotantovaiheessa tulee esiin Scrum-kehyksen ominainen piirre, eli päivittäiset Sprint-palaverit kehitysryhmän kesken. Niiden tehtävänä on varmistaa, ettei kukaan projektiryhmästä jää vaille jotakin tärkeää tietoa tai juutu kehitystyössä vastaan tuleviin ongelmiin. Koska palavereja pidetään niin usein, on niiden pituudeksi määrätty enimmillään 15–30 minuuttia [Ris00].

Palaverit ovat määrämuotoisia Scrum Masterin eli projektipäällikön johtamia tilaisuuksia, joissa ei ole tarkoitus ratkaista teknisiä ongelmia, vaan ainoastaan seurata työn etenemistä. Palaverissa jokainen kehitysryhmän jäsen vastaa seuraaviin kysymyksiin: [Ris00]

- Mitä toimintoja iteraation tekemättömien töiden listasta olet saanut valmiiksi edellisen palaverin jälkeen?
- Mitä esteitä sinulla on työsi edistämisessä?
- Mitä toimintoja iteraation tekemättömien töiden listasta aiot saada valmiiksi seuraavaan palaveriin mennessä?

Jokaisen sprintin päätteeksi Scrum-kehitysryhmä esittelee sprintin loppupalaverissa päättyneen sprintin aikana toteutetut toiminnot asiakkaalle. Tällöin asiakkaat arvioivat toteutettuja toimintoja ja antavat toteuttajille palautetta. Loppupalaverissa tarkastetaan iteraation tulokset ja päivitetään tekemättömien töiden lista. Tässä vaiheessa on jälleen mahdollista lisätä, poistaa, muuttaa ja uudelleen priorisoida listan tehtäviä. Loppupalaverissa voidaan päättää myös uuden sprintin aloittamisesta käymällä läpi koko tuotteen työlistaa ja palaamalla näin uuden sprintin suunnitteluvaiheeseen. [Mou07] [Pre05] [Ris00]

Projektin loppupuolella päätetään, onko ohjelmisto jo valmis vai vieläkö tarvitaan lisää iteraatiokierroksia. Mikäli uusien toimintojen tekemistä ei enää nähdä tarpeelliseksi tai

kustannustehokkaaksi, siirrytään Scrum-kehyksen jälkivaiheeseen. Tässä vaiheessa suoritetaan ohjelmiston eri osien integrointi, systeemitestaus, dokumentointi sekä ohjelmistotuotteen toimitus asiakkaalle. Scrumissa voidaan jokaisen iteraation jälkeen joko toimittaa ohjelmiston nykyinen versio asiakkaalle tai vain esitellä sen toiminnallisuutta ilman varsinaista toimitusta. Scrum-kehystä suositellaan käytettäväksi pienissä kehitysryhmissä, joiden koko on alle 10 henkilöä. Mikäli kehitysryhmän koko on tätä suurempi, suositellaan ryhmän jakamista alle 10 hengen ryhmiksi. [Abr02] [Pre05]

4.3 Crystal-menetelmäperhe ja Crystal Clear

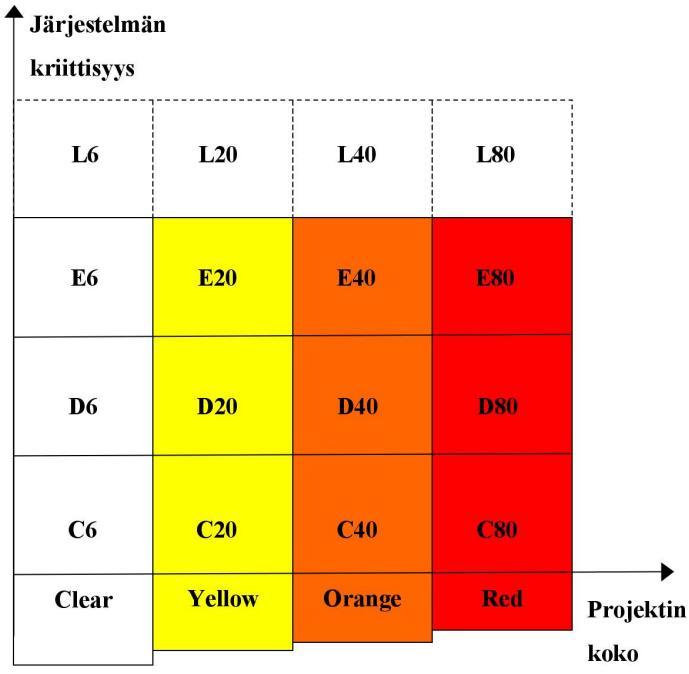
Crystal-metodologiat ovat Alistair Cockburnin kehittämä ketterien kehitysmenetelmien perhe. Ne ovat kokoelma erilaisia menetelmiä, joista voidaan aina projektin tarpeiden mukaan valita tarkoitukseen sopivin vaihtoehto. Crystal-menetelmien ydinelementtejä ovat ihmis- ja kommunikaatiokeskeisyys, inkrementaalinen kehitys ja lyhyt kehityssykli, kehitysmenetelmän sovitus ja muokattavuus tarpeen mukaan sekä arviointipalaverit ennen ja jälkeen kehityssykliä. Valitessa käytettävää menetelmää projekteja arvioidaan projektiryhmän koon ja projektin kriittisyyden mukaan. [Coc02]

Projektiryhmän koko vaikuttaa kommunikointitarpeisiin ja -menetelmiin, vastaavasti projektin kriittisyys voidaan ryhmitellä eri tavoilla. Crystal-menetelmät keskittyvät erityisesti yksilöiden rooleihin ja ryhmän kommunikaation määrittelemiseen, minkä vuoksi projektin henkilömäärä on ensisijainen kriteerimenetelmän valinnalle. Menetelmät eivät kuitenkaan tarjoa tarkempia ohjeita siitä, kuinka ja miten varsinainen kehitystyö tulisi konkreettisesti tehdä. Crystal-menetelmissä on ohjelmistoprojekteille käytössä erilaisia kriittisyysasteita nimettynä sen mukaan, miten menetykset kohdistuisivat mikäli toteutettava ohjelmisto menettäisi toimintakykynsä ohjelmistovirheen takia. Kriittisyysasteita ovat: [Coc02]

- Mukavuus (Comfort, lyhenteenä kirjain C). Ohjelmistovirheen seurauksena tehtävät joudutaan tekemään tietokoneen sijasta käsin, mikä vie enemmän aikaa.
- Kohtuullinen raha (Discretionary money, lyhenteenä kirjain D). Ohjelmistovirheen seurauksena tapahtuva rahallinen menetys voidaan myöhemmin korvata.

- Kriittinen (tai olennainen) raha (Essential money, lyhenteenä kirjain E). Ohjelmistovirheen seurauksena tapahtuvat taloudelliset menetykset ovat yksilön tai organisaation kannalta merkittäviä.
- Elämä (Life, lyhenteenä kirjain L). Ohjelmistovirheen seurauksena tapahtuvat tilanteet saattavat vaarantaa ihmishenkiä.

Sopivan kehitysmenetelmän valinta Crystal-menetelmäperheen sisällä tapahtuu yhdistämällä ohjelmistoprojektin kehitysryhmän koko toteutettavan järjestelmän kriittisyyskoodiin. Näin ollen esimerkiksi D20 tarkoittaa tilannetta, jossa ohjelmistoa on kehittämässä kaksikymmentä henkilöä ja kehitettävän ohjelmiston virheellinen toiminta voi pahimmassa tapauksessa johtaa rahalliseen menetykseen. Crystal-menetelmät erotetaan toisistaan värien avulla. Vähiten henkilöresursseja vaativa menetelmä on Crystal Clear (kirkas), seuraavaksi tulevat Yellow (keltainen), Orange (oranssi) ja Red (punainen). Näistä Crystal Clear on suunniteltu 1–6:n, Yellow alle 20:n, Orange 20–40:n ja Red 40–80:n henkilön kokoisille ohjelmistoprojekteille. Crystal-menetelmäperhe havannoillistetaan kuvassa 14. [Coc02]



Kuva 14. Crystal-menetelmäperhe graafisesti havainnollistettuna. [Coc02]

Crystal-menetelmien perhe ei ole vielä täysin valmis ja sen ohjeet ovat valmiita vain Crystal Clear- ja Orange -tasoisille projekteille. Näissä kahdessa ohjelmiston tuotantokierros on luonteeltaan iteratiivinen. Iteraation ensimmäistä osaa kutsutaan järjestämiseksi, jossa valitaan seuraavaan iteraatioon tulevat toiminnot ja samalla arvioidaan toimintojen tekemiseen kuluva aika [Abr02]. Jokaista ohjelmiston inkrementtiä kohden tehdään yksi tai useita iteraatioita ja jokaisen iteraation päätteeksi käyttäjä katselmoi tuloksia ja antaa niistä palautetta [Coc01]. Ohjelmistoa mitataan edistymisen ja vakauden suhteen. Edistymistä mitataan niin sanotuilla merkkipaaluilla, joita voivat olla esimerkiksi yksittäisen käyttötapauksen koodin ja sen testauksen valmistuminen. Vakausaste kertoo kuinka luotettavasti ohjelmisto toimii. Kun ohjelmiston vakausaste on saatu riittävän vakaaksi katselmointia varten, voidaan se esitellä asiakkaalle [Abr02].

Crystal Clear on Crystal-nimiseen menetelmäperheeseen kuuluva, erittäin kevyt ja epämuodollinen menetelmä, joka on suunniteltu pienille ohjelmistoprojekteille [Coc02]. Sen periaatteisiin kuuluu intensiivinen kommunikaatio, mahdollisimman kevyet projektin tuotokset, tiheä julkaisusykli sekä byrokratian vähentäminen, millä tarkoitetaan pakollisten toimenpiteiden sekä tuotoksien määrän minimointia [Coc07]. Alla luetellaan ne käytännöt ja ohjelmistoprojektin tuotokset, joiden toteuttamista ja/tai toimittamista Crystal Clear edellyttää: [Abr02]

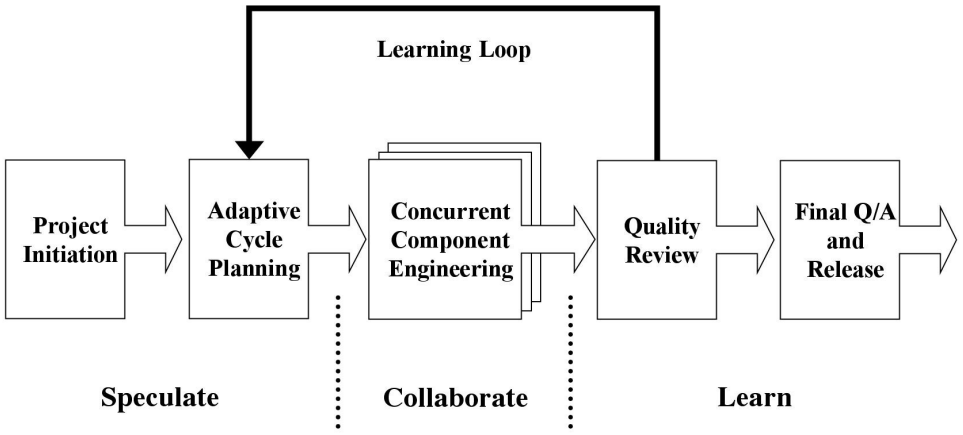
- Toimiva ohjelmisto toimitetaan inkrementaalisesti säännöllisin väliajoin.
- Projektin etenemisen mittarina käytetään toimivaan ohjelmistoon ja sen ominaisuuksiin sidottuja etappeja mieluummin kuin kirjoitettuja dokumentteja.
- Asiakkaan edustajan on oltava läsnä koko projektin ajan.
- Ohjelmiston kriittisimmät osat testataan automaattisesti.
- Projektin aikana koodia katselmoidaan jatkuvasti.
- Projektin seuranta- ja kehityspalaveri pidetään inkrementin puolivälissä ja sen jälkeen.

Crystal Clear-menetelmässä projektin kulku jaksottuu inkrementteihin, mikä ajallisesti tarkoittaa 1–3 kuukauden jaksoa, jonka lopuksi asiakkaalle toimitetaan versio ohjelmistosta. Projektisuunnittelun tehtävänä on jakaa kehitettävän ohjelmiston ominaisuudet niin, että

1–3 kuukauden aikarajaa ohjelmiston toimittamisessa ei ylitetä. Projektissa on tarpeen mukaan yksi tai useampia inkrementtejä peräkkäin, kunnes projektin lopuksi toimitetaan täysin valmis ohjelmisto kaikkine oheistuotoksineen. Yksi inkrementti voi koostua kahdesta tai useammasta peräkkäisestä iteraatiosta, mikä tarkoittaa, että kehitysryhmä varaa aikaa kehittämiensä ohjelmakoodin ja/tai dokumenttien tarkasteluun, arvioimiseen ja parantamiseen. [Abr02]

4.4 Adaptive Software Development / ASD

ASD eli Adaptive Software Development-malli esiteltiin Jim Highsmithin toimesta vuonna 2000 [Abr02]. ASD-mallin taustalta löytyy näkemys, jonka mukaan ohjelmistoprojektien toimintaympäristö on arvaamaton ja jatkuvassa muutoksessa. ASD:n on esitelty soveltuvan erityisesti monimutkaisten ohjelmistojen ja järjestelmien toteutukseen. Sen keskeisiä toiminta-ajatuksia ovat ihmisten välinen yhteistyö ja itseorganisoituvat kehitysryhmät. ASD-malli pyrkii hylkäämään ennustettavuuteen pyrkivän vesiputousmallin työtavat ja ohjelmisto tuotetaan evoluutiomallia mukaillen lyhyissä toimitussykleissä. Mallille on määritelty kolmivaiheinen sykli, jossa perinteinen määrittely-, suunnittelu- ja toteutusmalli on korvattu kierrolla, jossa spekulointi-, yhteistyö- ja oppimisvaiheet vuorottelevat [4-4-1]. ASD:n vaiheet esitetään kuvassa 15.



Kuva 15. ASD-mallin vaiheet. [Abr02]

ASD-mallissa spekulointi korvaa perinteisten menetelmien määrittelyvaiheen. Vaihetta kutsutaan spekuloinniksi, koska ohjelmistoprojektit ovat alttiita muutoksille eikä voida etukäteen tietää lopullisen tuotteen tarkkoja yksityiskohtia. Tällöin spekuloinnilla pyritään pääsemään yksimielisyyteen ohjelmistoprojektin suurista suuntaviivoista. Ohjelmiston toteutusvaiheen korvaa yhteistyövaihe. ASD-mallissa ihmisten välinen yhteistyö nähdään kriittisenä menestystekijänä koko projektille. Vastaavasti perinteisissä ohjelmistoprojekteissa tapahtuva laaduntarkkailu, katselmoinnit ja loppupalaverit on korvattu ASD-mallissa oppimisvaiheella. [Pre05]

ASD-projektin aloittaa spekulointivaihe. Tässä vaiheessa hyödynnetään asiakkaan asettamia tavoitteita, aikataulua, ja perusvaatimuksia ja niiden perusteella määritellään tarvittava määrä ohjelmiston kehitysversioiden julkaisuja, joille määrätään julkaisuajankohdat. Samalla päätetään myös koko ohjelmistoprojektin valmistumisajankohta. Kehitysversioiden ja koko ohjelmistoprojektin julkaisuajankohdat ovat kiinteitä, eikä niistä voida tinkiä. Jos jokin asetetuista tavoitepäivämääristä on vaarassa ylittyä, asiaa ei pyritä korjaamaan tekemällä ylitöillä tai vaarantamalla ohjelmiston laatua, vaan supistamalla kyseiseen kehitysversioon kuuluvia ominaisuuksia. [Pre05]

Yhteistyövaiheella tarkoitetaan varsinaista ohjelmiston toteutustyötä. ASD-malli antaa paljon vapauksia toteutuksen suhteen ja kehitysryhmät voivat toteutuksessa käyttää muita menetelmiä. Kulloisenkin iteraatiokierroksen määrittely annetaan toteuttajille tavoitteiden muodossa, jonka jälkeen kehitysryhmä toimii itseohjautuvasti näiden tavoitteiden saavuttamiseksi. Yhteistyövaiheessa voi olla useita rinnakkaisia ohjelmistokehitysryhmiä, jotka toteuttavat järjestelmän eri osakokonaisuuksia yhtäaikaisesti. Näin ollen ASD sopii myös suurempien ja hajallaan olevien kehitysryhmien käyttöön, mutta tällöin tarvitaan erityisen kurinalaisia tiedotus- ja yhteistyökäytäntöjä. Yhteistyövaiheessa pidetään tärkeänä kehitysryhmän sisäistä ja kehitysryhmien keskinäistä viestintää ja tiedon jakamista. Kun kehitysryhmä katsoo saavuttaneensa tavoitteet, voidaan siirtyä oppimisvaiheeseen. [Hig00]

Oppimisvaiheessa tarkastellaan valmistuneen iteraatiokierroksen tai koko ohjelmistoprojektin tuloksia. Arvioinnin tarkoituksena on huolehtia siitä, että kehitysryhmän keskittymi-

sen kohteina ovat alussa määritellyt päämäärät ja tavoitteet. Kehitysryhmät arvioivat työn laatua seuraavista näkökulmista: [Hig00]

- Asiakasnäkökulma eli projektin tulosten laatu asiakkaan näkökulmasta. Toiko tämä iteraatio asiakkaalle jotain uutta ja hyödyllistä. Tässä tapauksessa tarvittava tieto saadaan asiakkaalta.
- Tekninen näkökulma eli projektin tulosten laatu. Tässä tapauksessa tarvittava tieto hankitaan perinteisin ohjelmistojen laaduntarkkailun menetelmin, eli käyttäen katselmointia ja testausta.
- Projektiryhmän sisäisen toiminnan taso eli kehitysryhmän toiminta ja sen käytössä olevat menetelmät. Voiko toimintatapoja tai työmenetelmiä parantaa jollain tavalla. Mikä toimii ja mikä ei. Mitä on tehtävä enemmän tai vähemmän. Tarvittava tieto tulee projektiryhmältä itseltään.
- Projektin aikataulujen pitävyys ja projektin tilanne eli onko pysytty suunnitelluissa aikatauluista. Tarvittava tieto saadaan kaikkien osallistujien yhteistyönä ja se saattaa johtaa muutoksiin seuraavien iteraatioiden toteutuksessa.

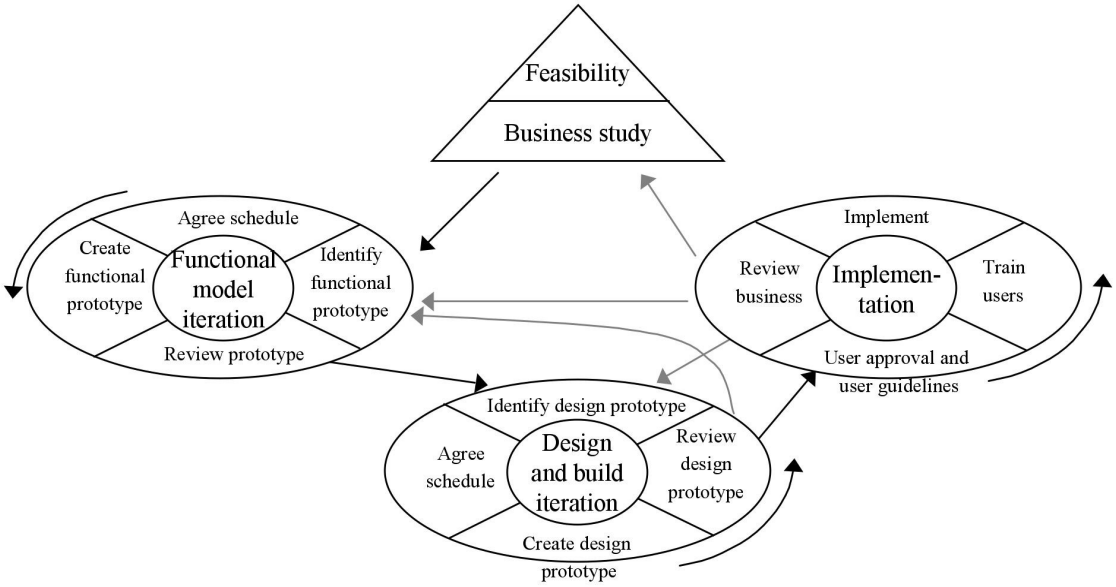
Oppimisvaiheen jälkeen aloitetaan seuraava iteraatiokierros. Edellä mainitut ASD-mallin vaiheet toistuvat iteraatiokierroksissa niin pitkään, kunnes projektille määritelty valmistumispäivä saavutetaan. Keskeistä ASD-mallissa on toimittaa lopputuote asiakkaalle ennalta määritettynä valmistumispäivänä, vaikka se olisi ominaisuuksiltaan puutteellinenkin. [Pre05]

4.5 Dynamic Systems Development Method / DSDM

DSDM eli Dynamic Systems Development Method on ensi kerran vuonna 1994 esitelty ketterä prosessimalli. Sen ideana on määrätä ensin ohjelmiston kehittämiseen käytettävä aika ja resurssit ja sopeuttaa sen jälkeen ohjelmiston toiminnallisuus niiden mukaan eli ohjelmiston laajuudesta voidaan joustaa, mutta sovittuja resursseja ei voida muuttaa. Iteraatioiden aikataulut ovat samalla tavoin tiukasti kiinnitetyt. Mikäli osan iteraatioon tarkoitettujen tehtävien suorittaminen osoittautuu mahdottomaksi määräaikaan mennessä, DSDM pitää aikataulussa pysymistä tärkeämpänä kuin toimintojen määrää, jolloin DSDM neuvoo

supistamaan toimintojen määrää. Menetelmää ylläpitää DSDM Consortium, joka on avoin ja voittoa tavoittelematon yhteisö. [Abr02]

DSDM sisältää viisi vaihetta, joista kaksi ensimmäistä eli esitutkimus (Feasibility study) ja vaatimusten määrittely (Business study) ovat peräkkäisiä ja suoritetaan vain kerran. Muut kolme vaihetta, eli toimintojen määrittelyvaihe (Functional model iteration), tekninen suunnittelu- ja toteutusvaihe (Desing and build iteration) sekä käyttöönottovaihe (Implementation) ovat luonteeltaan iteratiivisia ja inkrementaalisia. DSDM-mallin vaiheet on havainnollistettu kuvassa 16. [Abr02]



Kuva 16. DSDM-mallin vaiheet. [Stap02]

DSDM-mallin esitutkimus vastaa perinteisten menetelmien esitutkimusvaihetta ja tällöin valmistuvat esitutkimusraportti ja karkean tason projektisuunnitelma. Tässä vaiheessa myös tarkistetaan DSDM:n olevan soveltuvin menetelmä meneillään olevan ohjelmistoprojektin hallintaan. Vaatimusten määrittely vastaa perinteisten menetelmien vaatimusmäärittelyä sillä erotuksella, että DSDM:n vaatimusten määrittelyssä syntyvät myös karkean

tason arkkitehtuurisuunnitelma ja prototyyppien kehittämissuunnitelma myöhemmin seuraavia vaiheita varten. [Abr02]

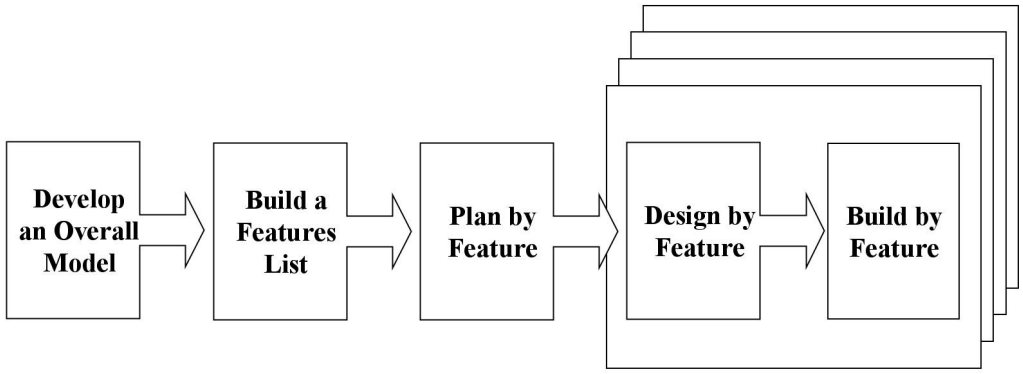
Varsinainen toimintojen määrittely suoritetaan toimintojen määrittelyvaiheessa ja samalla tehdään myös prototyyppejä määriteltyjen toimintojen pohjalta. Tässä vaiheessa määritellään myös ei-toiminnalliset vaatimukset ja näille vaatimuksille tehdään oma dokumentaatio. Määrittelyvaiheessa jokainen iteraatiokierros tuottaa uuden asiakkaalle esiteltävän prototyypin, josta myös kirjoitetaan katselmointiraportti seuraavia iteraatioita varten. Jokainen iteraatio tuottaa lisäksi myös toiminnallisen määrittelyn kaavioineen sekä riskianalyysin kehittämistyöstä. DSDM-mallissa testaus alkaa toimintojen määrittelyvaiheessa ja se tehdään jokaiselle prototyypille. Prototyyppien on tarkoitus olla testauksen jälkeen tarpeeksi hyviä toimiakseen myöhemmin pohjana varsinaiselle järjestelmälle. [Abr02]

Teknisen suunnittelun ja toteutuksen vaiheessa valmistuva lopullinen järjestelmä rakennetaan prototyyppien perusteella ja jokaisen iteraation lopuksi se esitellään asiakkaalle tiedonsaantia ja kommentteja varten. Käyttöönottovaiheessa valmis järjestelmä luovutetaan asiakkaalle käyttöä varten, kirjoitetaan tarpeellinen dokumentaatio sekä annetaan asiakkaalle järjestelmän käyttöön tarvittava koulutus. Tällöin myös kirjoitetaan projektin loppuraportti, jossa tarkastellaan projektin kulun lisäksi myös sitä, onko projektin aikana jouduttu tinkimään järjestelmän ominaisuuksista tai onko niitä ilmaantunut lisää. [Abr02]

DSDM:n voidaan katsoa sopivan projekteihin, joissa on tiukka budjetti ja aikataulu. DSDM:n painopiste on ohjauksessa ja ryhmätyössä, eikä se anna tarkkoja ohjeita varsinaisen ohjelmointityön tekemisestä. DSDM-mallille on kuitenkin määritelty yhdeksän eri käytäntöä, jotka pääosin noudattavat ketterien menetelmien periaatteita. Mallille ominaisia piirteitä ovat muita ketteriä menetelmiä tarkempi ja kattavampi dokumentointi ja jo varhaisessa vaiheessa tapahtuva valmistettavan ohjelmiston perusvaatimusten vakiinnuttaminen. Nämä seikat tekevät DSDM:stä muita ketteriä menetelmiä vähemmän joustavan. [Abr02] [DSD07]

4.6 Feature-Driven Development / FDD

FDD eli Feature-Driven Development on vuonna 2000 ensimmäisen kerran esitelty ketterä menetelmä [Coa00] ja se on suunniteltu integroitumaan osaksi toista ohjelmistokehitysmallia [Pal02]. FDD keskittyy ohjelmiston teknisen suunnittelun, toteutuksen ja testauksen suorittamiseen, eikä se suoranaisesti tarjoa kehitysmallia kaikkien ohjelmistoprojektin vaiheiden hallintaan [Abr02]. Menetelmä koostuu viidestä eri vaiheesta, jotka ovat karkean mallin kehittäminen (Develop an overall model), toimintolistan kerääminen (Build a features list), kehityksen suunnittelu (Plan by feature), toimintojen suunnittelu (Design by feature) ja toteutus (Build by feature). Näistä kaksi viimeistä vaihetta ovat luonteeltaan iteratiivisia. [Abr02]. FDD-menetelmän vaiheet on esitetty kuvassa 17.



Kuva 17. FDD-menetelmän vaiheet. [Pal02]

Ensimmäisen vaiheen eli karkean mallin kehittämisen alussa ohjelmiston vaatimukset on jo selvitetty ennakolta valmiiksi jonkin muun kattavamman ohjelmistoprojektimallin avulla [Pal02]. Vaiheen alussa ohjelmistokehittäjille esitellään vaatimukset yleisellä tasolla. Sen jälkeen ohjelmistokehittäjät jakautuvat pienempiin ryhmiin, jolloin jokaiselle ryhmälle esitellään ryhmäkohtaiset tarkennetut vaatimukset. Näiden vaatimusten perusteella kohdealuetta ja sen toimintoja mallinnetaan karkean tason luokka- ja sekvenssikaavioilla. [Abr02]

Toisessa vaiheessa kerätään toimintolista, joka sisältää sovellukseen halutut toiminnot. Toimintolistaa katselmoidaan asiakkaan kanssa, jolloin voidaan varmistua, että lista kattaa kaikki asiakkaan tarpeet. Toimintolistaa voidaan päivittää projektin aikana ja sitä voidaan myös muuttaa tarpeen mukaan, mikä lisää menetelmän joustavuutta. Kolmannessa eli kehityksen suunnitteluvaiheessa karkean tason luokkakaavion luokat jaetaan ohjelmistokehitysryhmille kehitettäviksi ja ylläpidettäviksi sekä suunnitellaan kehitysryhmille toteutusaikataulut. [Abr02]

Neljännen eli toimintojen suunnitteluvaiheen aluksi valitaan toiminnoista ne, jotka tässä iteraatiossa toteutetaan ja muodostetaan ohjelmistokehittäjistä toimintojen toteutukseen sopivat ryhmät. Suunnittelu on perinteistä ohjelmiston teknistä suunnittelua, josta siirrytään myöhemmin iteraation sisällä ohjelmointi- eli toimintojen toteutusvaiheeseen. Samaan aikaan voi toimia useita rinnakkaisia, eri toimintoja kehittäviä, ohjelmistokehitysryhmiä. [Abr02]

Ohjelmointiin kuuluu ohjelmakoodin kirjoittamisen lisäksi myös koodikatselmoinnit, moduulitestaus sekä iteraation lopuksi valmistuneen koodin integrointi aiemmissa iteraatioissa syntyneeseen koodiin. Tämän jälkeen valitaan seuraavan iteraation toiminnot ja niistä tehdään seuraava inkrementti. Toimintojen suunnittelu- ja toteutusvaiheita toistetaan iteratiivisesti, kunnes kaikki toimintolistan toiminnot on saatu valmiiksi jolloin koko prosessi päättyy. [Abr02]

FDD-mallille on määritelty kahdeksan eri käytäntöä, jotka noudattelevat ketterien menetelmien periaatteita. FDD:n kehittäjät pitävät mallia sopivana uusille ohjelmistoprojekteille, sekä ohjelmistojen korjaus- ja päivitysprojekteille. FDD-malli suositellaan otettavaksi käyttöön vähitellen projektin etenemisen mukaan. Mallin kehittäjät pitävät sitä sopivaksi vaihtoehdoksi käytettäväksi myös kriittisiin järjestelmiin, mikä seikka erottaa sen muista ketteristä menetelmistä. [Pal02]

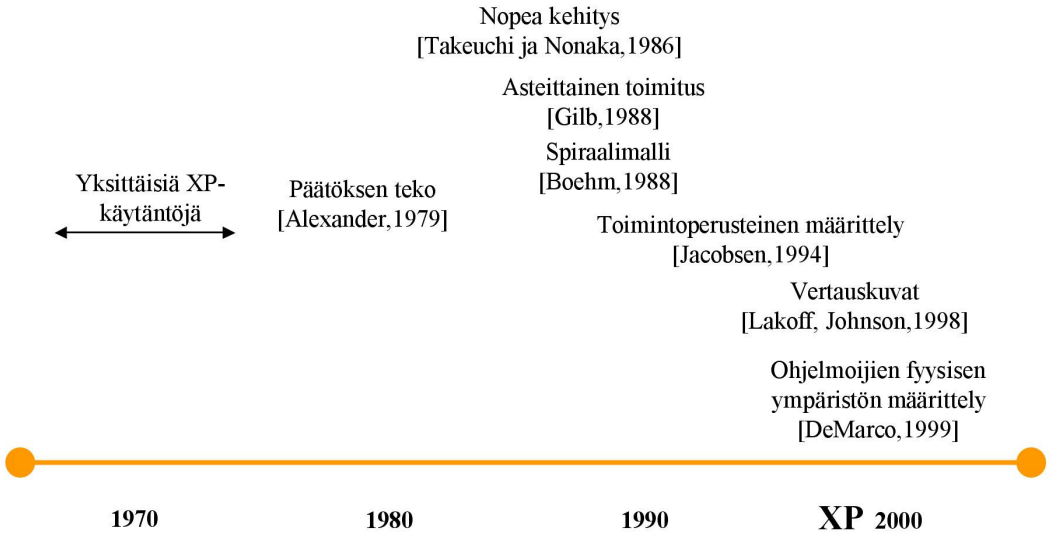
5 EXTREME PROGRAMMING (XP)

Luvussa 5 käsitellään XP:tä ohjelmoinnin prosessimallina ja tässä yhteydessä käydään läpi myös muut mallin pääpiirteet eli XP:n ydinarvot ja käytännöt. Luvun lopussa tutustutaan XP:n laadunhallintaan ja testaukseen liittyviin seikkoihin. Kent Beckin vuonna 1999 määrittelemää XP-mallia (Extreme Programming) pidetään yleisesti tunnetuimpana ketterien menetelmien ohjelmistokehitysmenetelmänä ja sitä pidetään myös ketterien menetelmien uranuurtajana ja kehityksen alkuna [Bec00].

5.1 Yleistä XP-menetelmästä

1990-luvun lopussa Kent Beck analysoi miten ohjelmistoprojektien laatua voisi parantaa. Beck oli havainnut joitakin yhtenäisiä seikkoja, jotka tuottivat eniten ongelmia ohjelmistoprojekteissa. Kommunikaatiota ei ollut riittävästi, toteutetut ratkaisut olivat monimutkaisia, työn laadusta ei saatu palautetta ajoissa ja kehittäjiltä puuttui tahtoa ratkaista eteen tulevat ongelmat puutteellisenkin tiedon varassa. Näistä seikoista johdettavista arvoista eli kommunikaatiosta, yksinkertaisuudesta, palautteesta ja rohkeudesta Beck kehitti Extreme Programming–mallin. [Aue03] [Bec00] Myöhemmin Beck lisäsi XP:n arvoihin myös kehitysryhmän jäsenten välisen kunnioituksen [Jar07] [Wik07b].

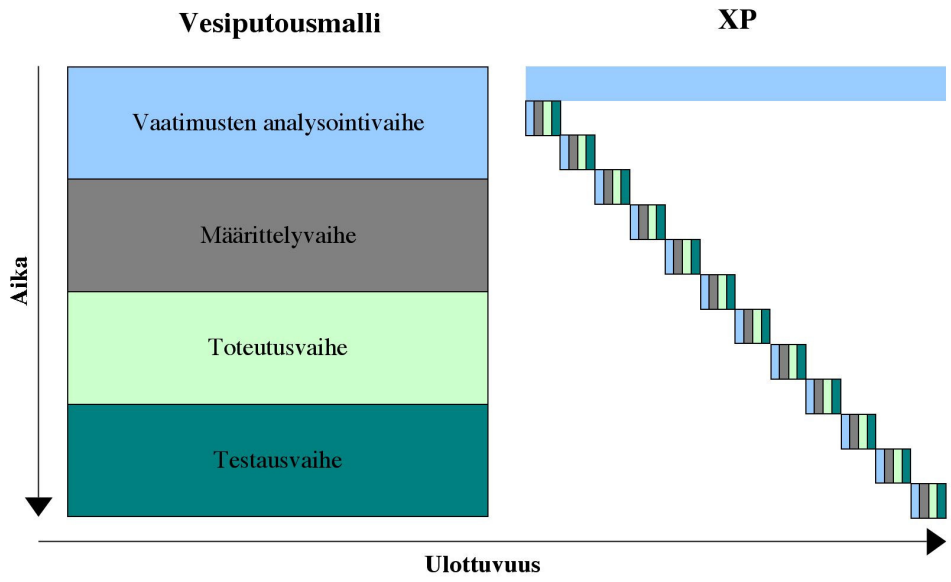
Beckin ideoima ja kehittämä XP on yhdistelmä ideoita ja käytäntöjä aikaisemmista, hyviksi havaituista menetelmistä [Abr02]. Ensimmäiset XP:n taustalla olevat ideat ovat peräisin 1970-luvun lopulta, mutta pääosiltaan Beck kehitti XP:n 1990-luvun puolessa välissä. XP:n kehityskaari on esitelty kuvassa 18. Beckin mukaan XP-menetelmälle on luonteenomaista jatkuvan muutoksen hyväksyminen, tehokas työskentely, laadukkaan ohjelmiston tuottaminen sekä ohjelmistokehitysryhmän jäsenten viihtyminen projektissa [Bec00]. XP:n mukaisessa prosessissa ohjelmistosta tuotetaan lyhyissä sykleissä uusia versioita ja jatkuvan iteraation avulla pyritään löytämään ohjelmistolle sen tarvittava toiminnallisuus.



Kuva 18. Extreme Programming-ideologian juuret. [Abr02]

Aiemmin esiteltyjen perinteisten menetelmien ja XP:n välillä ilmenevät eroavaisuudet tulevat esille varautumisessa mahdollisiin muutoksiin. Perinteisesti järjestelmien arkkitehtuuri suunnitellaan siten, että uutta toiminnallisuutta lisätään pienin muutoksin. XP:ssä ei pyritä ennustamaan tulevaisuutta, koska tulevia tarpeita ei välttämättä tiedetä tai vaatimukset saattavat muuttua ajan kuluessa. Järjestelmästä tehdään jokaisessa iteraatiossa vain niin monimutkainen kuin sen hetkiset vaatimukset edellyttävät. XP:ssä tulevien vaatimusten mukainen muuntautumiskyky pyritään takaamaan ohjelmiston yksinkertaisella rakenteella ja helpolla muunneltavuudella.

Kuvassa 19 havainnollistetaan miten vesiputousmallin ja XP:n kehityssyklit eroavat toisistaan. Vesiputousmallissa jokainen vaihe suoritetaan loppuun saakka ennen kuin seuraavaan vaiheeseen siirrytään. XP:ssä kunkin iteraation aikajänne on erittäin lyhyt ja iteraatioiden määrä suuri, mutta lopulta saavutettava toiminnallisuus on ihannetilanteessa sama kuin vesiputousmallilla.



Kuva 19. Vesiputousmallin ja XP:n kehityssyklien eroavaisuudet. [Bec99]

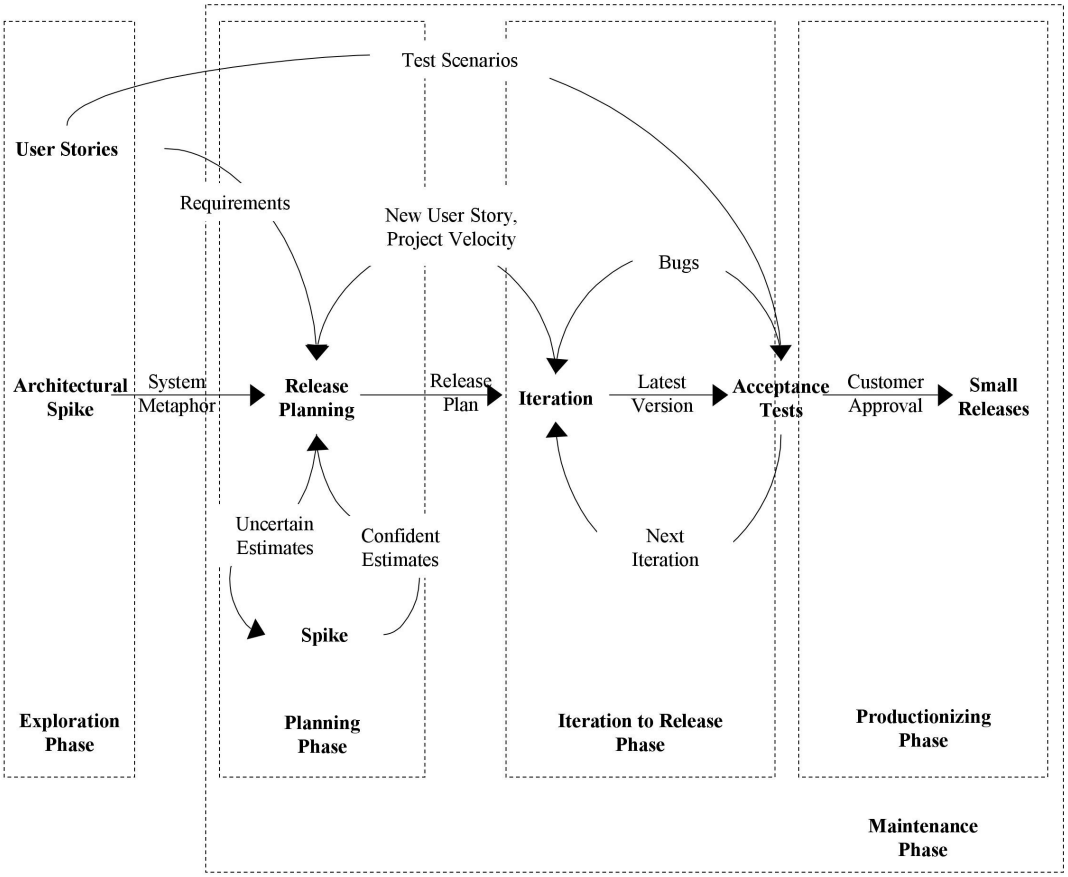
5.2 XP:n prosessimalli

XP:n prosessimallin tavoitteena on tuottaa yksinkertaisin mahdollinen ratkaisu, joka täyttää aina sen hetkiset vaatimukset [Bec99]. XP:n prosessi on luonteeltaan iteratiivinen ja inkrementaalinen [Schn03]. Yhden vaiheen tuotoksena on inkrementti, joka koostuu viimeisimmän ja sitä edeltävien vaiheiden tuotoksista. XP ei ole kuitenkaan puhtaasti inkrementaalinen, sillä myös aiempaa ohjelmakoodia muokataan jatkuvasti paremmaksi.

XP ei ole vaiheistettu prosessimalli, kuten esimerkiksi vesiputousmalli on, mutta sen elinkaari voidaan esittää vaiheistettuna. XP:ssä ei vaiheilla ole samanlaista merkitystä kuin esimerkiksi vesiputousmallissa, sillä vaiheiden välillä voidaan vuorotella ja ne voivat esiintyä jopa niin, että yhden vaiheen aikana tehdään samanaikaisesti suunnittelua, toteutusta, testausta ja integrointia [Amb02]. Vaiheiden avulla voidaan havainnollistaa XP:tä käyttävän ohjelmistoprojektin elinkaarta helpommin ymmärrettäväksi. [Bec00]

XP:ssä ohjelmistoprojektin vaatimukset määritellään käyttäen käyttäjäkertomuksia ja niiden kirjoittaminen aloitetaan yleensä koko järjestelmän päämäärästä [Bec00]. Käyttäjäkertomukset ovat pienimpiä mahdollisia vaatimuksia, jotka voidaan kuvata jonkin toimin-

non suorittamiseksi. Kertomusten tarkoituksena on antaa kehittäjälle riittävästi tietoja, jotta kehittäjä kykenee niiden pohjalta arvioimaan toteuttamiseen tarvittavan ajan [Bec00]. XP:ssä ohjelmistosta tehdään useita, tuotantokäyttöön valmiita julkaisuja, jotka taas sisältävät joukon pienempiä iteraatioita [Mue02]. Jokaisen iteraation aluksi asiakas ja kehittäjät sopivat, mitkä käyttäjäkertomukset toteutetaan toiminnallisuuksina iteraatiossa [Bec99]. Kuvassa 20 on havainnollistettu XP:n prosessimallin kulkua.



Kuva 20. XP-projektin vaiheet [Wel01].

5.2.1 Tutkimusvaihe (Exploration Phase)

Ensimmäisenä XP-projektin vaiheista on tutkimusvaihe, jossa luodaan projektin perusta ja alkuvaiheen vaatimukset ja sen kesto vaihtelee muutamasta viikosta muutamaan kuukau-

teen. Tutkimusvaihe käsittää alustavan arkkitehtuurin tutkimisen sekä alkuvaiheen käyttäjäkertomusten kirjoittamisen. XP:ssä asiakas kuvaa käyttäjäkertomusten avulla miten järjestelmän tulisi toimia. Tämä vastaa perinteisten ohjelmistokehitysmallien vaatimusmäärittelyä. Tutkimusvaihe käsittää alustavan arkkitehtuurin tutkimisen sekä alkuvaiheiden käyttäjäkertomusten kirjoittamisen. [Bec00]

XP:ssä ei ole määritelty tarkkaan millaisia käyttäjäkertomusten tulisi olla ja miten ne olisi dokumentoitava. XP:ssä ainoastaan suositellaan, että käyttäjäkertomuksen tulee olla lyhyt kuvaus ohjelmiston toiminnallisuudesta ja että sen pohjalta voidaan laatia hyväksymistestaus. Beckin mukaan alkuvaiheessa kerättyjen käyttäjäkertomusten tulisi sisältää mahdollisimman kattava kuvaus järjestelmästä, sillä ne määrittävät projektin laajuuden [Bec00]. Käyttäjäkertomus määritellään sarjaksi järjestelmän toimintoja, joiden tehtävänä on tuottaa mitattavissa olevaa hyötyä järjestelmän käyttäjälle [Jac95]. Käyttäjäkertomuksiin ei kirjata teknisiä yksityiskohtia kuten esimerkiksi poikkeuksia tai virhetilanteita.

5.2.2 Suunnitteluvaihe (Planning Phase)

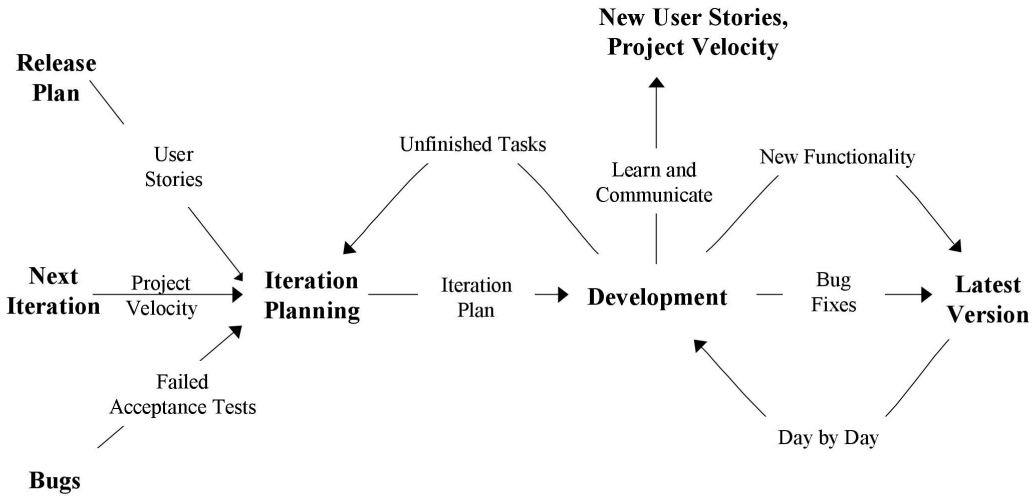
Tutkimusvaihetta seuraa suunnitteluvaihe. Sen tarkoituksena on päättää ensimmäisen julkistuksen päivämäärä, sen sisältö ja tuottaa julkistussuunnitelma [Bec00]. Julkistuksen tulisi sisältää vain se joukko kertomuksia, joiden avulla seuraavasta ohjelmaversiosta voidaan saada lisäarvoa eli asiakkaan kannalta arvokkain joukko käyttäjäkertomuksista [Amb02]. Julkistussuunnitelman tarkoitus on tarjota asiakkaalle kuva projektin kokonaiskestosta ja sitä käytetään myöhemmin iteraatiovaiheessa tapahtuvassa iteraatioiden suunnittelussa.

Jotta projektin kokonaiskestoa ja -kustannuksia voitaisiin arvioida, on oltava jokin tapa arvioida käyttäjäkertomusten työmääriä. Ohjelmistokehittäjät arvioivat käyttäjäkertomusten pohjalta paljonko aikaa tarvitaan kunkin kertomuksen toteuttamiseen ja mitä riskejä niihin liittyy. Mikäli arviointi on kertomuksen monimutkaisuuden vuoksi mahdotonta, jaetaan kertomus pienempiin osiin, jotka sitten arvioidaan kukin osa erikseen. Julkistuksia suunniteltaessa projekti saatetaan hylätä, ellei saavutettavan hyödyn suhde projektin hintaan tunnu järkevältä. On esitetty, että suunnitteluvaihe on otollinen vaihe hylätä projekti,

ennen kuin siihen on tehty merkittäviä panostuksia [Ast02]. Siirtyminen eteenpäin iteraatiovaiheeseen on asiakkaan puolelta osoitus sitoutumisesta projektiin.

5.2.3 Iteraatiovaihe (Iteration to Release Phase)

Beckin mukaan iteraatiovaihe on eniten työtä vaativa XP:n vaihemallin mukaisista vaiheista ja tässä yhteydessä tapahtuu pääosa itse kehitystyöhön liittyvistä seikoista eli ohjelmoinnista, testauksesta ja integroinnista [Bec00]. Iteraatiossa järjestelmään lisätään jokin rajattu toiminnallisuus sekä korjataan käyttäjiltä saadun palautteen perusteella havaitut viat. Iteraatioita suunnitellessa keskitytään nykyisen iteraation käyttäjäkertomuksiin ja kehitysryhmä arvioi välttämättömät tehtävät käyttäjäkertomuksen toteuttamiseksi [Bec00]. Iteraatiossa suoritettavien yksittäisten tehtävien tulee olla lyhytkestoisia ja testattavia. Yhden iteraation kesto on 1-4 viikkoa ja kunkin iteraation lopputuloksena on uusi julkaistavissa oleva toimiva ohjelmaversio [Bec00]. Kuvassa 21 havainnollistetaan yksittäisen iteraation elinkaarta.



Kuva 21. XP-mallin iteraatio [Wel01].

Iteraatiosuunnitelma on tietty määrä arvioituja tehtäviä eli käyttäjäkertomusten osia, jotka ositetaan tehtäviksi niihin käytettävien työmäärien arvioinnin helpottamiseksi. Yksittäisten tehtävien tulee olla lyhytkestoisia ja testattavia, eikä niiden kesto saisi ylittää kahdeksaa tuntia. Tätä pidemmät tehtävät tulisi pyrkiä jakamaan osiin [Ast02]. Tietyn iteraation paris-

sa työskenneltäessä saatetaan löytää uusia käyttäjäkertomuksia, joita ei ole vielä arvioitu. Niiden osalta etsitään arviointia varten käyttäjäkertomuksiin sisältyvät tehtävät. Uusien käyttäjäkertomusten arvioinnin jälkeen saatetaan joitakin käyttäjäkertomuksia joutua lykkäämään, mikäli ne eivät mahdu iteraatioon [Amb02].

Koko ohjelmiston ensimmäiseen iteraatioon valitaan sellaiset kertomukset, joiden avulla voidaan luoda koko järjestelmän arkkitehtuuri. Valittujen kertomusten pitäisi kuitenkin olla mahdollisimman yksinkertaisia ja nopeita toteuttaa, jotta järjestelmä saadaan ohjelmoitua ja testattua nopeasti ja sitä kautta saadaan palaute sen toimivuudesta.

5.2.4 Tuotteistamisvaihe (Productionizing Phase)

Tuotteistamisvaiheessa varmistetaan, että kehitettävä järjestelmä on valmis tuote. Vaihe sisältää esimerkiksi järjestelmä-, kuormitus- ja asennustestausta [Amb02]. Tässä vaiheessa julkistaminen tapahtuu jo oikeaan ympäristöön kehitysympäristön sijasta. Tuotteistamisvaiheessa syntyy myös tarpeellinen dokumentaatio eri sidosryhmille [Amb02]. Kun uusi toiminnallisuus on saatu toimimaan, sille suoritetaan hyväksymistestit ja ohjelmisto annetaan testattavaksi ja kokeiltavaksi asiakkaalle. Saadun palautteen perusteella tehdään tarvittavat muutokset. Siinä vaiheessa, kun asiakas on hyväksynyt ohjelmiston toiminnallisuuden, suoritetaan varsinainen ohjelmistojulkistus. Tämän jälkeen XP:n ohjelmistoprosessi alkaa taas alusta, jolloin asiakas ja kehitysryhmä alkavat yhdessä kartoittamaan seuraavan ohjelmajulkistuksen laajuutta [Bec00]. Tuotteistamisvaihe saattaa sisältää useita iteraatiokierroksia [Abr02].

5.2.5 Ylläpitovaihe (Maintenance Phase)

Vaihemallin viimeinen vaihe on ylläpitovaihe [Amb02]. Vaihe on XP-projektin normaalitila ja se on eräänlainen paketti, joka sisältää suunnittelu-, iteraatio- ja tuotteistamisvaiheet. Yleensäkin kaikki toiminnot, mitkä perinteisessä prosessissa kuuluvat ylläpidon piiriin, kuuluvat siihen myös XP-mallissa [Bec00]. Ylläpitovaiheessa käyttäjät määrittelevät uusia ominaisuuksia sisältäviä kertomuksia, joita ohjelmistokehittäjät toteuttavat. Myös käyttäjätukitoiminnot sisältyvät ylläpitovaiheeseen.

5.3 XP:n ydinarvot

XP:n käytäntöjä ohjaavat seuraavaksi esiteltävät viisi ydinarvoa. Ne ovat perustana kaikelle XP-projektissa tapahtuvalle toiminnalle ja ydinarvojen avulla XP:n myöhemmin esiteltävien käytäntöjen ymmärtäminen on helpompaa [Bec00]. XP:n neljä alkuperäistä arvoa ovat kommunikaatio, yksinkertaisuus, palaute ja rohkeus. Lisäksi viidenneksi arvoksi ollaan ottamassa kehitysryhmän jäsenten välistä kunnioitusta [Wik07b]. XP:n arvot ovat yhteneväisiä aiemmin esitellyn ketterien menetelmien manifestin kanssa ja juuri nämä yhteiset arvot tekevät XP:stä ketterän menetelmän [Aue03][Bec01].

5.3.1 Kommunikaatio

Epäonnistuneiden ohjelmistoprojektien syyt voidaan hyvin usein johtaa puutteelliseen kommunikaatioon ja erään tunnetun sanonnan mukaan neljä viidesosaa maailman ongelmista johtuu nimenomaan juuri huonosta tai riittämättömästä kommunikaatiosta. Tästä syystä XP:ssä kommunikaatio on nostettu keskeiseen rooliin. Kommunikaation halutaan tapahtuvan paitsi kehitysryhmän sisällä, myös kehitysryhmän ja asiakkaan kesken [Bec00] [New02]. XP:ssä noudatetaan työtapoja, joita on mahdotonta tehdä ilman ihmisten välistä kommunikaatiota. Lisäksi suurimman osan kommunikaatiosta tulisi olla suoraan henkilökohtaisesti tapahtuvaa keskustelua. Kuitenkin myös XP-projekteissa kommunikaation puute on vaarana, sillä dokumentointia valmistetaan vähemmän kuin muissa menetelmissä.

Perinteisissä ohjelmistotuotantoprosesseissa suunnitelmien ja päätösten dokumentoimiseen menevä aika käytetään XP-mallissa ryhmän sisällä tapahtuvaan suoraan ja epämuodolliseen kommunikaatioon. Maurerin [Mau02] mielestä tällaisesta lähestymistavasta on hyötyä silloin, kun ohjelmistokehitysryhmä on suhteellisen pieni ja voidaan olettaa, että suoralla kommunikaatiolla saavutetaan koko ohjelmistokehitysryhmä. Suora kommunikaatio on myös nopeampaa kuin tehtyjen päätösten dokumentointi. Dokumentaatiota tehtäessä aikaa kuluu tulevaisuuden ennustamiseen ja siihen, mitä dokumentaation käyttäjä haluaisi mahdollisesti tietää. Mikäli nämä ennustukset eivät osu kohdalleen, dokumentointiin kulutettu aika mennyt on hukkaan. Samalla tavalla käy, mikäli kehitettävän ohjelmiston suunnitelmat muuttuvat merkittävästi.

5.3.2 Yksinkertaisuus

Yksinkertaisuuden arvon mukaisesti kaikki suunnitelmat ja kaikki koodi on pidettävä aina mahdollisimman yksinkertaisina millä tarkoitetaan sitä, että kehittämisen kohteena oleva ohjelmisto toteutetaan yksinkertaisimmalla mahdollisella toimivalla tavalla [Bec00]. Ohjelmakoodi kirjoitetaan nykyhetken vaatimusten mukaan sen sijaan, että yritettäisiin ennustaa tulevaisuuden vaatimuksia [Bec00]. Tulevaisuuden vaatimusten ennustaminen on aina vaikeaa ja epävarmaa, joten siihen käytetty aika katsotaan XP:ssä hukkaan heitetyksi [New02].

XP:ssä ei saisi koskaan varautua sellaisiin vaatimuksiin, jotka eivät ole suunnitteluhetkellä vielä tiedossa, eikä ohjelmistoa tietoisesti suunnitella laajennettavaksi eri suuntiin [Bec00]. Tällä on ajateltu sitä, että muutoksista aiheutuva lisätyö myöhemmin on pienempi kuin mahdollisesti tulevaisuudessa tapahtuvan laajennettavuuden lisääminen heti suunnitteluvaiheessa. XP painottaa yksinkertaisen rakenteen tärkeyttä. Beckin mukaan yksinkertaisina pidettävien ohjelmistojen tulee noudattaa seuraavia kriteerejä, ensinnäkin ohjelmistokoodi läpäisee kaikki testit, toiseksi koodi kommunikoi ohjelmoijalle kaiken tarvittavan, kolmanneksi seikaksi Beck mainitsee sen, että kopioitua koodia ei ole ja neljänneksi, että ohjelmisto sisältää ainoastaan minimimäärän luokkia ja metodeja. [Bec00]

5.3.3 Palaute

Kolmantena arvona on palaute, jota XP:ssä kerätään usein ja välittömästi tehdyn työn jälkeen. Kehittäjien ja asiakkaiden tulisi saada palautetta järjestelmän tilasta niin usein kuin mahdollista [Kee04]. Sitä kerätään toimittajalta, asiakkaalta sekä itse rakennettavasta ohjelmistosta [Bec00]. XP:n käytännöt toimivat siten, että palautetta kehitettävästä järjestelmästä saadaan ja annetaan kehityksen alusta lähtien useasti ja paljon. XP:n käytännöt, kuten pienet julkistukset, jatkuva integrointi ja testauslähtöinen kehitys, mahdollistavat aikaisen ja jatkuvan palautteen [Bec00] [New02].

5.3.4 Rohkeus

Rohkeus on neljäs XP-metodologian arvo. XP on monilta periaatteiltaan varsin radikaali, joten jo pelkkä XP:n käyttö vaatii rohkeutta. Monet ohjelmistoprosessimallit pyrkivät toi-

mimaan varman päälle, jolloin etukäteen tapahtuvassa suunnittelussa pyritään varautumaan kaikkiin mahdollisiin tulevaisuuden skenaarioihin. XP:ssä rohkeudella tarkoitetaan sitä, että suunnittelutyö uskalletaan lopettaa ajoissa ja aloittaa järjestelmän toteuttaminen, vaikka kaikkia mahdollisia tulevaisuuteen liittyviä asioita ei olisikaan vielä saatu ratkaistua. [Bec00] [New02]

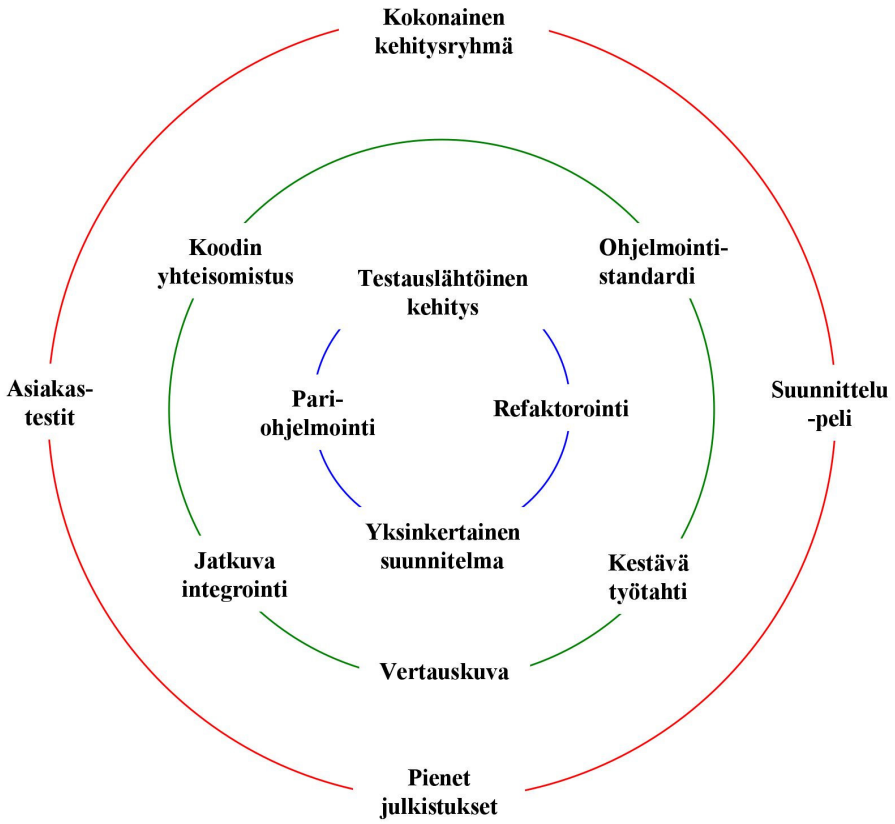
Tämä XP-metodologian arvo perustuu sille ajatukselle, että ohjelmistokehittäjien tulee kyetä näkemään, milloin kehitysprosessi on ajautunut väärään suuntaan ja korjaukset ovat välttämättömiä. Ongelmien korjaaminen saattaa tarkoittaa monen päivän töiden heittämistä pois ja koodin kirjoittamista uudelleen, vaikka se olisikin aiemmin läpäissyt testit. [Kee04]

5.3.5 Kunnioitus

XP:ssä kehitysryhmän jäsenet pyrkivät kunnioittamaan toisiaan ja toistensa aikaansaannoksia. XP:n ydinarvojen mukaan ohjelmoijan ei pitäisi tehdä koskaan muutoksia, jotka aiheuttavat sekaannusta, viivästyksiä tai vaikeuttavat ohjelmiston testausta. Kehitysryhmän jäsenet kunnioittavat työtänsä pyrkien aina korkeaan laatuun ja etsivät parasta mahdollista ratkaisua käyttäen XP:n refraktorointi-käytäntöä. [Wik07b]

5.4 XP:n käytännöt

XP-ohjelmoinnissa on 13 käytäntöä, joita ohjaavat aiemmin esitellyt ydinarvot. XP:n ydinarvoja toteutetaan ja sovelletaan käytäntöön noudattamalla XP:n käytäntöjä [Kee04]. XP:ssä käytettävät työskentelymenetelmät eivät ole täysin uusia, sillä niissä on yhdistetty aiemmin hyväksi koettuja työskentelytapoja, jotka erikseen käytettynä omaavat joitakin puutteita. XP:ssä on pyritty siihen, että eri käytännöt tukevat toinen toisiaan ja tekevät menetelmästä elinvoimaisemman. Kaikki kehitystyö XP-projektissa tapahtuu käyttäen tässä yhteydessä esiteltäviä käytäntöjä. [Bec00][Jef07] [Mit03] [Wel01]



Kuva 22. Extreme Programming-prosessimallin käytännöt. [Jar07] [Wel01]

Kuvassa 22 on esitetty XP:n käytäntöjen nivoutuminen kerroksittain. Käytäntöjen voidaan katsoa kuuluviksi kolmeen eri kategoriaan niin, että ne voidaan jakaa yksilön, kehitysryhmän ja koko organisaation suorittamiin käytäntöihin [Ast02]. Kuvan ulkokehällä ovat koko organisaation, keskimmäisellä kehitysryhmän ja sisimmällä yksilön toteuttamat käytännöt. XP:n käytännöt ovat toisistaan riippuvaisia ja toisiaan tukevia [Van05].

5.4.1 Yksilön toteuttamat käytännöt

Yksinkertainen suunnitelma

XP-projektin suunnitelma on pidettävä niin yksinkertaisena kuin järjestelmän toiminnallinen taso sen sallii [Ste04]. Ylimääräistä monimutkaisuutta ei tarvita tai sallita ja se pyritään löytämään ja poistamaan mahdollisimman nopeasti [Abr02] [Kee04]. Sen vuoksi suunni-

telman on käsitettävä vain seuraavan iteraation aikana lisättävät toiminnot. Mikäli koodin havaitaan muuttuvan epäselväksi, sitä muutetaan ja parannetaan, kunnes se on jälleen mahdollisimman yksinkertaista [Ste04].

XP:ssä pyritään suunnittelemaan kerrallaan vain niin pitkälle kuin on välttämätöntä. Tämä perustuu käsitykseen, ettei suunniteltuja toimintoja todennäköisesti tarvita ja vielä suurempi on todennäköisyys sille, että tarvitaan jotakin täysin ennustamatonta. Yksinkertaisinta ratkaisua haettaessa etsitään ensin jokin mahdollinen ratkaisu asiakkaan pulmaan, sillä ilman minkäänlaista ratkaisumallia ei voida myöskään arvioida käyttäjäkertomusten toteuttamiskustannuksia. Beck [Bec00] esittää yksinkertaisen suunnitelman olevan kelvollinen, jos se suoriutuu kaikista testeistä, on selkeä ja ymmärrettävä, sisältää mahdollisimman pienen määrän luokkia ja metodeja eikä sisällä toistoa.

Refaktorointi

Refaktorointi on prosessi, jossa ohjelmistoa muutetaan pienin askelin niin, että ohjelmiston ulkoista toimintaa ei muuteta, mutta sen sisäistä rakennetta parannetaan. Refaktoroinnilla pyritään parantamaan järjestelmän rakennetta poistamalla eri luokissa esiintyvää päällekkäistä toiminnallisuutta ja luomalla koodista uudelleenkäytettävämpää, yksinkertaisempaa ja ylläpidettävämpää [Bec00] [Smi01]. XP:ssä kehitys aloitetaan hyvällä ja yksinkertaisella rakenteella ja projektin edetessä refaktoroinnilla varmistetaan, että ohjelmakoodin rakenne pysyy koko ajan yksinkertaisena ja soveltuvana sen hetkiseen tilanteeseen [Hed03].

Koodin refaktorointi on systemaattista, kurinalaista ja ankaraa koodin sisäisen esityksen asteittaista parannusta [Ast02]. Koodin refaktorointia suoritetaan jatkuvasti koko projektin ajan. XP:ssä jokainen ohjelmoija on vastuussa rakenteen parantelusta ja sitä suoritetaan aina, kun siihen on tarvetta. Yksikkötestien avulla voidaan varmistua, että toiminnallisuus on säilynyt rakenteen tai ohjelmakoodin muutoksesta huolimatta ennallaan. Ilman kattavia yksikkötestejä refaktorointi ei olisi mahdollista sillä muuten muutoksiin liittyvät riskit voisivat kasvaa liian suuriksi. [Hed03]

Pariohjelmointi

Pariohjelmointi on XP:n ydinkäytäntö. Jokaisen XP-projektin tuottaman koodirivin kirjoittavat pareittain toimivat ohjelmoijat. Toinen ohjelmoijista kirjoittaa koodia, toisen toimiessa taustalla huomauttaen virheistä ja tehden ehdotuksia koodin suhteen. Tämä käytäntö varmistaa, että kaikki tuotantokoodi on aina vähintään yhden muun ohjelmoijan tarkastamaa, mikä taas johtaa parempaan rakenteeseen, testaukseen ja ohjelmakoodin laatuun [Bec00]. Koodauksen lisäksi samanaikaisesti tapahtuu muita toimintoja, kuten suunnittelua, analysointia, refaktorointia, testausta ja koodin tarkastusta [Ast02]. Parit eivät välttämättä ole kiinteitä, vaan ne voivat vaihdella päivittäin ja/tai suoritettavien tehtävien mukaan. [Bec00] [Ste04].

Pariohjelmoinnista tehdyt tutkimukset osoittavat, että pari tuottaa laadukkaampaa ohjelmakoodia lähes samassa ajassa, kuin jos ohjelmoijat työskentelisivät yksittäin [Hed03] [Coc00]. Tutkimusten mukaan valtaosa pariohjelmoinnin oppineista ohjelmoijista pitävät tätä käytäntöä parempana kuin yksintyöskentelyä [Hed03]. Pariohjelmointi ei ainoastaan auta tuottamaan laadukkaampaa ohjelmakoodia, vaan se edesauttaa jakamaan tietoa ja osaamista eteenpäin. Ohjelmoijat oppivat toisiltaan, taidot kehittyvät ja ohjelmoijista tulee entistä hyödyllisempiä sekä ohjelmistokehitysryhmälle että yritykselle [Hed03].

Testauslähtöinen kehitys

Testauslähtöinen kehitystoimintatapa tarkoittaa sekä kehittäjien määrittelemien yksikkötestien että asiakkaan määrittelemien hyväksymistestien tekemistä ennen toiminnon varsinaista toteutusta. Koko XP:n työskentelymenetelmä perustuu testaukseen ja ilman kattavaa testausta XP:n eräät muut työskentelytavat, kuten refaktorointi ja jatkuva integrointi, eivät olisi mahdollisia XP:n edellyttämässä nopeudessa ja laajuudessa. XP:n ohjelmistotestaus koostuu yksikkö- ja hyväksymistesteistä. Yksikkötestit eroavat hyväksymistesteistä siten, että hyväksymistestit testaavat kokonaisia käyttäjäkertomuksia, kun taas yksikkötestit testaavat käyttäjäkertomusten osia. Tyypillisiä yksikkötestauksen kohteita ovat esimerkiksi luokat ja metodit [Ast02].

Vaadittavia testejä ajetaan jatkuvasti uuden koodin kirjoittamisen tai vanhan koodin muuttamisen jälkeen [Kee04]. Tehtävä on suoritettu vasta silloin, kun ohjelmakoodi läpäisee täydellisesti kaikki yksikkötestit ja se on integroitu muuhun koodiin. Koska XP:ssä ei ole erillistä testausvaihetta testauksen tapahtuessa aina kehitystyön yhteydessä, on yksikkötestien mentävä aina täysin läpi [Hed03]. Etuina jatkuvasti tapahtuvasta testauksesta on se, että se tarjoaa runsaasti palautetta kehitettävästä järjestelmästä [Ste04]. Testaus ei saisi kuitenkaan hidastaa itse kehitystyötä, joten on tärkeätä automatisoida tapahtuva testaus. [Hed03].

5.4.2 Kehitysryhmän toteuttamat käytännöt

Vertauskuva

Vertauskuvan eli metaforan tarkoitus on esittää tuntemattomaan kohdealueeseen sisältyvä vaikea asia ymmärrettävällä tavalla. Jokaisessa XP-projektissa voidaan käyttää yhtä tai useampaa vertauskuvaa ohjaamaan projektia sekä tarjoamaan kehitysryhmille yhteinen käsitteellinen malli niin, että kaikki kehitysryhmän jäsenet käyttävät samanlaista kieltä ja terminologiaa puhuessaan kehitettävästä järjestelmästä [Ast02]. Vertauskuva voidaan esittää esimerkiksi yhdessä käyttäjäkertomuksessa, jossa kuvataan koko järjestelmän idea [Mau02]. Vertauskuva kertoo hyvin ytimekkäästi sen, minkälainen järjestelmä on tarkoitus kehittää ja sen voidaan katsoa olevan korkean tason kuvaus järjestelmän arkkitehtuurista [Mau02].

Alkuvaiheessa on tärkeintä, että asiakas onnistuu luomaan kehittäjälle jonkinlaisen mielikuvan siitä, mitä järjestelmän tulisi olla. Vertauskuvia voidaan käyttää koko järjestelmän kehittämisen ajan. Jossakin vaiheessa niitä saatetaan poistaa, mikäli huomataan, että ne joko on jo toteutettu tai ovat tulleet tarpeettomiksi. Vertauskuva voidaan myös todeta vääräksi, jolloin sitä ei enää tarvita [Ast02]. Vaikkei kattavaa vertauskuvaa löydettäisikään, XP-kehitysryhmät käyttävät joka tapauksessa yhteisiä nimityksiä järjestelmästä ja sen osista. Näin varmistutaan, että kaikki ymmärtävät järjestelmän toiminnan, mistä jotain haluttua toiminnallisuutta pitää etsiä tai mikä olisi oikea paikka lisätä uutta toiminnallisuutta [Hed03].

Jatkuva integrointi

XP:ssä järjestelmän on käännyttävä, olla ajettavissa ja selviydyttävä kaikista testeistä riippumatta järjestelmän hetkellisestä toiminnallisesta tasosta. Tästä seuraa, että kaiken järjestelmään lisättävän koodin tulee kääntyä, olla ajettavaa ja selviytyä testeistä [Ste04]. Jatkuvan integroinnin seurauksena järjestelmästä voidaan joutua ottamaan lukuisia uusia koosteita päivässä [Ste04]. Koosteella tarkoitetaan versiota koko järjestelmästä jollakin tietyllä hetkellä. Integroinnin suositellaan tapahtuvan vähintään kerran päivässä, mieluummin useammin [Ast02]. Mikäli ohjelmakoodi integroidaan vain harvoin, integroinnista tulee helposti hankala ja aikaa vievä operaatio [Ast02].

Jatkuvalla integroinnilla varmistetaan ohjelmiston eri osien yhteensopivuus ja aiemmin luotujen testien läpäisemisen varmistaminen missä tahansa ohjelmiston kehitysvaiheessa [Abr02]. Kaikki ohjelmistokehittäjät työskentelevät samassa kehityshaarassa, mikä tarkoittaa sitä, että kaikki työskentelevät saman ohjelmakoodin parissa [Kee04]. Integrointi tuo esille helposti virheitä ja ongelmia, joita ei ole ilmennyt ennen integrointia suoritetuissa testeissä. Lisäksi voi käydä niin, että integrointi annetaan sellaiselle henkilölle tehtäväksi, joka ei tunne kunnolla koko järjestelmää. Tämä voi johtaa virheelliseen ohjelmakoodiin ja tuhlaa aikaa, mikäli ohjelmoijat joutuvat odottamaan ennen kuin voivat toteuttaa joitakin uusia ominaisuuksia [Hed03].

Koodin yhteisomistus

XP:ssä koko kehitysryhmä omistaa kaiken järjestelmän koodin ja sen dokumentaation [Ste04]. Jokainen kehitysryhmän jäsen saa tehdä muutoksia mihin tahansa koodiin tai dokumenttiin nähdessään siihen olevan tarvetta [Bec00]. Koska useat parit voivat muokata samanaikaisesti samaa ohjelmakoodin osaa, tulee versionhallinnan käyttäminen välttämättömäksi. Pelkkä versionhallinnan käyttäminen ei kuitenkaan riitä, vaan kaikkien kehitysryhmän jäsenten on päästävä käsiksi mihin tahansa koodin osaan milloin tahansa [Ast02]. Jotta koodin yhteisomistus toimisi käytännössä, on välttämätöntä suorittaa integrointia lyhyin väliajoin, jotta mahdolliset ristiriitaiset muutokset voidaan huomata nopeasti ja pitää ne mahdollisimman pieninä.

Koska ohjelmakoodi on samanaikaisesti monen ihmisen tarkkailun kohteena, koodin laatu paranee ja virheiden mahdollisuus vähenee [Hed03]. Mikäli ohjelmakoodi on vain yksittäisten henkilöiden omistuksessa, vaaditut ominaisuudet sijoitetaan helposti väärään paikkaan. Tästä voi olla seurauksena vaikeasti ylläpidettävää ja rumaa ohjelmakoodia, joka on täynnä turhaa toistoa [Hed03]. Eräs koodin yhteisomistuksen tarkoitus on kannustaa kaikkia kehitysryhmän jäseniä tuomaan esille omia ideoitaan ja parempia vaihtoehtoja toteuttaa järjestelmä [Wel01].

Ohjelmointistandardi

Kehitysryhmän kaikkien jäsenten tulee käyttää samaa ohjelmointistandardia. Sillä, mikä tämä standardi on, ei ole niin suurta merkitystä kuin sillä, että sitä todella käytetään [Ast02] [Jef07]. Myöskään käytettävän standardin yksityiskohdat eivät ole tärkeitä. Tärkeintä on se, että kaikki ohjelmakoodi näyttää jokaiselle kehittäjälle tutulta ja näin ollen tukee ohjelmakoodin yhteisomistajuutta. Ohjelmointistandardin eräänä tarkoituksena on, että kaikki ohjelmakoodi näyttää kuin sen olisi kirjoittanut yksi asiantunteva ja ammattitaitoinen henkilö.

Ohjelmointistandardi varmistaa keskittymisen itse koodin sisältöön, koska kaikki käyttävät samaa ohjelmointistandardia. Siten esimerkiksi koodin ulkoasu, kommentointikäytäntö ja muuttujien nimeäminen ovat kaikille tuttuja, joten niihin ei enää tarvitse kiinnittää paljoa huomiota. Käytäntö toteuttaa osaltaan kommunikaation ydinarvoa, sillä yhteiset ohjelmointistandardit auttavat koodin pitämisessä yhteneväisenä luokasta toiseen koodin ollessa helppolukuista ja ymmärrettävää. [Bec00] [New02]

Kestävä työtahti

Kestävä työtahti tarkoittaa sitä, että jokaiseen iteraatioon pitäisi kulua kehitysryhmältä suunnilleen sama työpanos ja työaika. Tästä käytännöstä puhutaan usein myös termeillä 40-tuntinen viikko tai säilytettävissä oleva työtahti [Hed03]. XP-prosessimalli ei salli ylitöitä, sillä ne voivat johtaa stressiin, lisäylitöihin ja väsymyksestä johtuviin virheisiin, joten pitkällä tähtäimellä säilyttämällä kestävä työtahti voidaan tuottaa lyhyemmässä ajassa pa-

rempia ohjelmia [Ste04]. Käytäntö ei ole siinä mielessä sitova, ettei XP-projektissa koskaan tehtäisi yli 40-tuntista työviikkoa, mutta ylityöt eivät saa olla pysyvä ilmiö.

5.4.3 Organisaation toteuttamat käytännöt

Pienet julkistukset

XP-projektissa julkaistaan testattua, toimivaa koodia, jonka toiminnallisuus on asiakkaan määrittämää. Julkistuksia tapahtuu usein, mikä tarkoittaa sitä, että ne ovat suppeita [Ste04]. Vaikka julkistukset ovat pieniä ja julkistuksia tapahtuu usein, julkistuksen tulee olla toimiva, kokonainen ja järkevä. Vain osittaisia tai keskeneräisiä toimintoja toteuttavia julkistuksia ei julkaista [Ast02]. Pienten julkaisujen käytäntöä noudatetaan kahdella tavalla, ensinnäkin kehitysryhmä julkaisee ajettavan ja testatun ohjelmiston jokaisen iteraation lopussa antaen sen asiakkaan käyttöön ja toiseksi XP-kehitysryhmä julkaisee uusia versioita säännöllisesti myös varsinaisille sovelluksen loppukäyttäjille, mikäli kyseessä eivät ole aiemmin mainitut asiakkaita [Hed03]. Kunnollisten toimivien versioiden julkaiseminen usein on mahdollista jatkuvan integroinnin, asiakastestien ja testauslähtöisen kehityksen ansiosta [Hed03].

Kun ohjelmisto annetaan jokaisen iteraation jälkeen asiakkaalle, kaikki projektissa tapahtuva kehitys on läpinäkyvää ja asiakas saa jatkuvasti vastinetta projektiin sijoittamilleen varoille [Hed03]. Asiakas arvioi saamansa toimituksen ja päättää sen perusteella, mitä ominaisuuksia hän tahtoo mukaan seuraavaan julkistukseen, joka tapahtuu seuraavan iteraation päätteeksi. Tämä auttaa tarkempien suunnitelmien tekemisessä, sillä palautteesta voidaan jatkuvasti ottaa opiksi ja ennustaa paremmin mitä seuraavan iteraation aikana on mahdollista järjestelmään lisätä.

Asiakastestit

Asiakkaan tehtävänä on hyväksymistestien suunnittelu ja niiden vapaamuotoinen kuvaaminen, kun taas kehitysryhmä toteuttaa ja automatisoi hyväksymistestit ajaen niitä säännöllisesti voidakseen todentaa, vastaako toimitettu järjestelmä käyttäjäkertomuksissa kuvattuja tavoitteita [Ste04]. Hyväksymistesti on samalla konkreettinen tilanne, jossa järjestelmää

käyttäessä joudutaan tilanteeseen, jossa kyseinen piirre ilmenee. Hyväksymistestien automatisointi voi nopeuttaa uusien julkistusten hyväksymistä huomattavasti ja on tärkeää, että käytetään jotain automatisoitua testaustapaa.

Hyväksymistestien tavoitteena on saavuttaa konkreettinen, yksinkertainen ratkaisu, jonka avulla voidaan osoittaa järjestelmän tietyn piirteen toimivan. Hyväksymistestejä kirjoitettaessa voidaan myös todeta ylimääräistä monimutkaisuutta käyttäjäkertomuksissa, jolloin niitä voidaan jakaa uusiksi käyttäjäkertomuksiksi. Hyväksymistesti voidaan jakaa kolmeen osaan. Ensinnäkin esiehdot kertovat minkälaisten olosuhteiden vallitessa testi voidaan suorittaa. Toinen osa on itse testi ja kolmanneksi arvioidaan testin jälkiehdot, jotka osoittavat testin onnistumisesta. [Ast02]

Kokonainen kehitysryhmä

Kokonainen kehitysryhmä luo yhteyden asiakkaan ja kehitysryhmän välille. XP:n kehitysryhmään kuuluu oleellisena osana aina paikan päällä oleva asiakas. Asiakas on mukana kehitysryhmässä oman alansa asiantuntijana auttaen kehittäjiä järjestelmän toteuttamisessa [Bec00]. Asiakkaalla on suuri vastuu ja valta tehdä päätöksiä projektin kehitystyön suunnasta [Ste04]. Kehitysryhmän kanssa toimivan asiakkaan edustajan tulisi olla yksi niistä, jotka käyttävät järjestelmää sen valmistuttua. Asiakkaalla pitäisi olla myös kyky tuoda julki se, mitä hänen työtoverinsa järjestelmältä haluavat [Ast02]. Asiakas antaa vaatimukset kehitettävälle järjestelmälle, asettaa prioriteetit halutuille uusille ominaisuuksille ja ohjaa projektin kulkua [Hed03].

Asiakkaan läsnäolo voi nopeuttaa ongelmanratkaisua, sillä kehitysryhmä voi jatkuvasti tukeutua asiakkaan mielipiteisiin ongelmien ilmaantuessa. Asiakas on sijoitettuna samaan paikkaan kehittäjien kanssa, jolloin kehittäjät voivat saada häneltä nopeasti palautetta ja vastauksia esiin tulleisiin kysymyksiin [Kee04]. Asiakkaan ei siis tarvitse välttämättä olla vahva tietoteknisessä osaamisessa, vaan tärkeintä on tietää mitä sovelluksen loppukäyttäjä sovellukselta vaatii ja kuinka sovellus voisi tätä parhaiten palvella.

Suunnittelupeli

XP:ssä tarkoituksena on saada aikaan hyvin minimaalinen suunnitelma projektin päämäärästä ja keinoista sinne pääsemiseksi ja tiedostaa mahdolliset riskit. Suunnitelma on aina epätarkka ja muuttuu projektin edetessä ja se käsittää aina seuraavan iteraation aikana järjestelmään lisättävät ominaisuudet. XP suunnittelu keskittyy kahteen avainkysymykseen ohjelmiston kehityksessä. Ensinnäkin sen ennakoimiseen, mitä saadaan valmiiksi määräpäivään mennessä ja toiseksi sen päättämiseen, mitä tehdään seuraavaksi. Pääpaino on projektin ohjaamisessa ennemmin kuin tarkassa ennustamisessa, mitä tullaan tarvitsemaan ja kuinka kauan siihen menee aikaa [Hed03].

XP:ssä on kaksi suunnitteluvaihetta, jotka pyrkivät antamaan vastauksen edellä mainittuihin avainkysymyksiin. Julkaisun suunnittelu on käytäntö, missä asiakas esittää ohjelmoijille halutut toteutettavat ominaisuudet ja ohjelmoijat arvioivat niiden vaikeusasteet. Julkaisusuunnitelma tarkentuu ajan kuluessa, mutta jo ensimmäinenkin suunnitelma on tarpeeksi tarkka, että voidaan tehdä päätöksiä projektin suhteen. Iteraation suunnittelu on käytäntö, jonka mukaan kehitysryhmän etenemissuunta tarkistetaan joka toinen viikko. Kehitysryhmä kehittää ohjelmistoa muutaman viikon iteraatioissa, joiden lopputuloksena on ajettava ja käyttökelpoinen sovellus. Mikäli edellisessä iteraatiossa on joidenkin ominaisuuksien toteutus jäänyt kesken, otetaan niiden vaatima työmäärä huomioon uusia tehtäviä valittaessa. [Hed03]

5.5 Henkilöstön roolit XP-mallissa

XP:ssä on joitakin projektinhallintaan liittyviä rooleja, vaikka ryhmä pyrkiikin tekemään päätökset yhdessä ja kommunikoimaan paljon. XP-projektin on todettu onnistuvan parhaiten, jos siihen osallistuville on annettu selkeät roolit. Yksi rooli voi olla jaettu useamman henkilön kesken ja toisaalta yksi henkilö voi omaksua monta roolia. Näin ollen sama henkilö voi olla sekä ohjelmoija että testaaja. XP-prosessimallin roolit on tässä yhteydessä esitetty Beckin mukaan. [Bec00]

Ohjelmoija

Ohjelmoija tekee ohjelmistotuotteen asiakkaan laatimien käyttäjäkertomusten pohjalta pitäen samalla ohjelmistokoodin mahdollisimman yksinkertaisena ja antaa arviot toimintojen toteuttamiseen tarvittavasta ajasta. Ohjelmoija on ohjelmistoprojektiryhmän tukipilari, joka suunnittelee ja kirjoittaa koodin yhdessä muiden ryhmän ohjelmoijien kanssa. Ohjelmoijan vastuulla on myös ohjelmistokoodin yksikkötestaus. XP-projektissa kaikki projektiryhmän jäsenet, mukaan luettuna ohjelmoijat, kommunikoivat suoraan asiakkaan kanssa, kun perinteisesti yhteyshenkilönä on toiminut projektipäällikkö.

Asiakas

Asiakas määrittelee käyttäjäkertomusten avulla ohjelmiston toiminnallisuuden sekä hyväksymistestit. Asiakkaan vastuulla on antaa käyttäjäkertomukset ohjelmistoprojektiryhmälle ja suunnitella toiminnallisia testejä. Asiakkaan vastuulla on päättää lisäysten sisällöstä, priorisoida käyttötapauksia ja päättää, milloin jokin käyttötapaus on hyväksytysti toimitettu.

Testaaja

XP:n mukaisesti jokainen ohjelmoija on omalta osaltaan vastuussa testauksesta, mutta testaaja tarkastelee myös muiden ohjelmoijien suorittamaa testausta ja tarvittaessa auttaa ja koordinoi integrointi- ja hyväksymistestejä. Testaajan tehtävänä on varmistaa, että haluttu toiminnallisuus on saatu toteutettua. Testaaja suorittaa tarvittavat testaustehtävät säännöllisesti raportoiden saaduista tuloksista ja löydetyistä virheistä muille kehitysryhmän jäsenille. Testaajat auttavat asiakasta toiminnallisten testien suunnittelussa ja voivat myös suorittaa varsinaisen toiminnallisen testauksen.

Mittaaja

Mittaaja seuraa projektin edistymistä ja annettujen aikatauluarvioiden toteutumista. Mikäli toteutuneet aikataulut poikkeavat annetuista arvioista, mittaaja ohjaa ryhmää tarkastamaan

arvioitaan oikeaan suuntaan. Perinteisissä prosessimalleissa mittaajan rooli on usein kuulunut projektipäällikölle.

Valmentaja

Valmentaja on henkilö, jolla on aiempaa kokemusta XP-prosessista sekä hyvät ohjelmointi-, testaus- ja refaktorointitaidot. Valmentajan roolina on huolehtia, että kehitysryhmä soveltaa XP:n prosessia, arvoja ja toimintatapoja käytännössä myös vaikeuksien sattuessa ja tarvittaessa muuttaa sitä, jotta se sopii paremmin projektin toteutusympäristöön. Myös tämä rooli on perinteisissä prosessimalleissa kuulunut usein projektipäällikölle.

Konsultti

Konsultti on ulkopuolinen henkilö, joka on liiketoiminnallisen tai teknisen alueen erikoisasiantuntija ja joka voi tietämyksellään auttaa projektiryhmää alueeseen liittyvissä erityiskysymyksissä. Konsultteja voi olla useita ja heidän roolinsa on projektissa lähinnä vieraileva.

Johtaja / projektipäällikkö

Johtajan/projektipäällikön vastuulla on päätöksenteko. Johtajan tehtävänä on huolehtia kehitysryhmän tarpeista, seurata projektin etenemistä ja poistaa edistymisen tiellä olevia esteitä. Projektipäällikön tehtäviin kuuluu myös laatia budjetti, hankkia tarvittavat resurssit, työvälineet integrointitestausta ja hyväksymistestausta varten sekä oikeanlaiset tilat kehitystyötä varten. Projektipäällikkö myös varmistaa, että kehitysryhmän jäsenet voivat aina tarpeen tulleen keskustella oikean ihmisen kanssa käyttäjäkertomuksen toteutuksesta. Vastuuta työmäärien arvioinnista ja asiakkaan kanssa tapahtuvasta kommunikaatiosta hajautetaan projektipäällikön lisäksi myös kehitysryhmän jäsenille.

5.6 Dokumentaatio XP:ssä

XP:ssä ei suositella erillisen dokumentaation tuottamista. XP-mallissa ei määritellä projektissa tuotettavaa dokumentaatiota muuten kuin käyttötapausten ja käyttäjäkertomusten kirjaamisen osalta. Kaikki muu projektin kommunikaatio tulisi tapahtua henkilökohtaisesti

ihmisten kesken, eikä dokumenttien välityksellä. XP:ssä dokumentaatiota voi ja pitää laatia, mikäli sen nähdään olevan tarpeellista järjestelmän monimutkaisuuden vuoksi. Dokumentaation määrä olisi pidettävä minimissään, sillä sen ylläpito saattaisi helposti hidastaa kehitystyötä. Dokumentaationa toimivat esimerkiksi käyttäjäkertomukset, ohjelmakoodi, yksikkö- ja hyväksymistestit. XP katsoo turhaksi kaikki dokumentit, joista ei ole hyötyä projektin valmistumisen jälkeen.

XP:ssä luotetaan, että käyttäjäkertomukset, keskustelut, kehitystyössä mukana oleva asiakas ja pariohjelmointi synnyttävät ja ylläpitävät keskustelua, jolloin ajatusten vaihto olisi helppoa ja väärinymmärrykset voitaisiin havaita nopeasti. XP:ssä tieto siirtyy suullisesti ryhmän sisällä ja kaikkea ylimääräistä dokumentointia vältetään. Theunissenin [The03] mukaan kehitysryhmässä tulisi olla dokumentaatiosta vastaava henkilö, mikäli projekti vaatii esimerkiksi tietyn standardin mukaisia dokumentteja. Dokumentoija voi kartuttaa tietojaan keskustelemalla ohjelmistokehittäjien kanssa ja käyttää automaattisia työkaluja, joilla lähdekoodista voidaan tuottaa tietoa ohjelmiston rakenteesta.

5.7 Laadunhallinta ja testaus XP:ssä

Beckin [Bec00] mukaan ohjelmistoprojektit ovat yleensä tasapainoilua kustannusten, ajan, laadun ja ohjelmiston laajuuden välillä. Ohjelmistoprojekteissa kiinnitetään yleensä kustannukset, aika ja laajuus, jolloin laatu saattaa jäädä heikoksi. Tätä ongelmaa voidaan hallita sopeuttamalla toteuttavan ohjelmiston laajuutta, jolloin laatukin voidaan pitää halutulla tasolla. XP:n käytäntöjen soveltamisella ja yhteisten arvojen jakamisella on havaittu olevan positiivinen vaikutus kehitettävän ohjelmiston laatuun. XP:n käytännöistä asiakkaan läsnäolo tekee laadun ja laajuuden suhteesta asiakkaan kannalta konkreettista [Mül02].

Eräs XP:n eroavaisuuksista verrattuna perinteiseen vesiputousmalliin on erillisen testivaiheen puuttuminen. XP:ssä testaaminen otetaan alusta asti osaksi kehitysprosessia ja sen koko työskentelymenetelmä perustuu testaukseen. XP:ssä testaamisen tehtävänä on varmistaa, etteivät ominaisuudet katoa tai tuhoudu refaktoroinnissa. XP:n kannattajat väittävät, että testaamisen ohjaaman kehityksen seurauksena ohjelmistoja voidaan muuttaa helpommin, kehitys tapahtuu nopeammin, kehittäjien luottamus ohjelmakoodiin lisääntyy ja

virheiden määrä laskee niin paljon, että lyhyestä syklistä ja siihen liittyvästä jatkuvasta testauksesta koituva lisärasite on siedettävä [Mül02].

5.7.1 Laadunhallinta XP:ssä

Beck jakaa ohjelmiston laadun ulkoiseen ja sisäiseen laatuun [Bec00]. Ulkoinen laatu kertoo, millaisena asiakas kokee tuotteen laadun. Sisäinen laatu on ohjelmistokehittäjän laatunäkökulma tuotteeseen, joka ilmenee esimerkiksi koodin ymmärrettävyytenä [Bec00]. Vaikka sisäisestä laadusta usein tingitään kiireessä, tulee se todennäköisesti näkymään myöhemmin myös ulkoisessa laadussa ohjelmiston elinkaaren aikana esimerkiksi vaikeana ylläpidettävyytenä.

XP:ssä ulkoinen laatu varmistetaan parhaiten sillä, että asiakas on mukana kehitysryhmässä. Tämä mahdollistaa asiakkaalle nopean palautteen antamisen kehittäjille ohjelmistoon liittyvissä kysymyksissä. XP panostaa ohjelmiston sisäisen laadun hallintaan ja erityistä huomiota kiinnitetään kehittäjien motivaatioon, koska motivaation on huomattu vaikuttavan lopulta myös ohjelmiston laatuun [Zus05]. Motivaatiota ja kehitysryhmän yhteishenkeä pyritään ylläpitämään yhteisillä ja yhdenmukaisilla työskentelykäytännöillä sekä jatkuvalla suullisella kommunikoinnilla. Beckin määrittelemään XP:hen ei kuulu erillistä laadunvarmistushenkilöä vaan sen tehtävät kuuluvat kaikille kehitysryhmän jäsenille [Bec00].

Perinteisissä ohjelmistoprojekteissa ohjelmiston laatu mitataan usein vasta projektin loppuvaiheessa, jolloin projektiryhmän ulkopuolinen laadunvarmistaja etsii ohjelmistosta vikoja. XP:ssä laatuvastuu hajautetaan koko kehitysryhmälle, sillä sekä asiakas että ohjelmistokehittäjät suorittavat ohjelmistolle jatkuvaa testausta tehden samalla ulkopuolisen laadunvarmistuksen tarpeettomaksi [Bec00]. XP-projekteissa ohjelmiston ja prosessin laatua tarkkaillaan myös mittaamalla. Mittaajan tehtävänä on seurata projektin etenemistä vertaamalla työmääräarvioita toteutusaikoihin sekä keräämällä ohjelmistoa koskevaa tietoa kuten esimerkiksi testitapausten määrää [Bec00].

5.7.2 Testaus XP:ssä

XP:n ohjelmistotestaus koostuu yksikkö- ja hyväksymistesteistä [Bec00]. Yksikkötestaus koostuu yksittäisten luokkien ja moduulien testauksesta, hyväksymistestauksen kuvatessa koko järjestelmän toiminnallisuutta. Myös integrointitestit luetaan XP:ssä yleisellä tasolla yksikkötesteihin. Testauslähtöisen kehityksen, pariohjelmoinnin ja jatkuvan integraation ansiosta tehdyt virheet ja niiden vaikutusalue huomataan mahdollisimman aikaisin, jolloin virheet voidaan korjata nopeasti. Jotta kattava ja nopeatahtinen testaaminen ja virheenkorjaaminen olisivat mahdollista, vaaditaan testiympäristöltä laajaa automatisointia. Testiympäristön automatisointi onkin olennainen osa XP-mallia [Bur03].

Käyttäjäkertomuksiin perustuvilla hyväksymistesteillä asiakas varmistuu toimitettavan ohjelmiston laadusta [Bec00]. Asiakkaan tehtävänä on määritellä, minkälaisen tehtävän ohjelman tulee pystyä suorittamaan, jotta se toteuttaa käyttäjäkertomuksen käyttötapauksen oikein. Ohjelmistokehittäjät tekevät tyypillisesti hyväksymistestauksen ja siihen liittyvät testitapaukset, koska asiakkaalla saattaa puuttua tarvittavaa ammattitaitoa tehdä kattavia ja hyviä testejä. Asiakas on kuitenkin mukana hyväksymistestauksessa, koska ohjelmisto hyväksytään tai hylätään pitkälti niiden perusteella. Lisäksi asiakkaan on ymmärrettävä hyväksymistestit [Jef99]. Hyväksymistestit kuvaavat koko järjestelmän toiminnallisuutta ja ne toimivat XP:ssä kehityksen etenemisen mittarina.

XP:n testauskäytännöt keskittyvät yksikkötestaukseen. Mikäli testauksen tai ohjelmoinnin aikana ohjelmasta löydetään virhe, siitä kirjoitetaan testitapaus. Näin varmistutaan, ettei sama virhe voi olla myöhemmin ohjelmassa. Jokainen iteraatiokierros aloitetaan tekemällä yksikkötestit tarvittavia uusia toiminnallisuuksia varten. Yksikkötestit suunnitellaan niin, että ne kuvaavat mahdollisimman pienen toiminnallisuuden lisäyksen ohjelmistoon ja ne laaditaan ennen varsinaisen koodin kirjoitusta. Tämän jälkeen ohjelmistoa korjataan kunnes se läpäisee tehdyt yksikkötestit [Bur03]. XP:ssä yksikkötestien on mentävä jatkuvasti täysin läpi, koska ohjelmistokehitys perustuu lyhyihin sykleihin eikä XP:ssä ole projektin lopussa erillistä testausvaihetta [Jef99].

Ohjelmistokoodin kehittäjä kirjoittaa yksikkötestit. Yksikkötestaus ja ohjelmointi tehdään samanaikaisesti, mikä on perinteistä ohjelmointia nopeampaa ja antaa kehittäjälle välitöntä

palautetta kehitettävästä järjestelmästä [Bec00]. Yksikkötestien on mentävä täysin läpi, ennen kuin testausta jatketaan integraatiovaiheeseen [Bur03]. Integraatiotestaus on koko ajan jatkuvaa, sillä ohjelmiston osa liitetään muuhun ohjelmistoon saman tien, kun ohjelmiston osalle tehtävät yksikkötestit ovat onnistuneet [AnD03]. XP:n yksikkötestausvaiheessa testitapaukset tehdään pääasiassa mustalaatikkomenetelmällä.

Ohjelmistokoodin luokista ja moduuleista ei tehdä yleensä erillistä dokumentaatiota ja sen vuoksi testit toimivat myös kommunikointivälineenä kehitysryhmän jäsenten kesken. Niistä käy ilmi miten luokkia käytetään ja millaisia erikoistapauksia niihin liittyy. Beckin mukaan testitapaukset on hyvä kirjoittaa, mikäli metodin toimintaan liittyy erityispiirteitä tai poikkeustapauksia, metodin toiminnallisuus tai käyttötarkoitus on epäselvä tai monimutkainen tai ohjelman toiminnassa on virhe, jota aiemmat testit eivät ole todenneet [Bec00].

6 ARVIOINTIA JA TUTKIMUSTULOKSIA XP–MALLISTA JA KETTERISTÄ MENETELMISTÄ

Luvun 6 aluksi perehdytään kirjallisuudesta esille tulleisiin tapoihin vertailla prosessimalleja ja vertaillaan yleisellä tasolla ketteriä ja perinteisiä prosessimalleja toisiinsa. Luvussa verrataan XP:tä yksinään joihinkin perinteisiin ja ketteriin prosessimalleihin. Lisäksi luvussa pohditaan XP:n soveltuvuutta erilaisiin ohjelmistoprojekteihin. Luvussa esitellään myös muutamia esille tulleita laajoja case-tapauksia XP:n käytöstä. Päähuomio luvussa kohdistetaan valmiin tutkimusmateriaalin tarkasteluun koskien XP:tä ja erästä sen keskeistä käytäntöä, pariohjelmointia. Tutkimukset ovat laajuudeltaan erilaisia, suoritettu erilaisissa ympäristöissä ja eri puolilla maailmaa.

6.1 Prosessimallien vertailuperusteista

Prosessimallien vertailu on hankalaa ja tehdyt johtopäätökset saattavat riippua tutkijasta johtuvista subjektiivisista seikoista tai ne saatetaan tehdä pelkän tunteen perusteella [Ost91]. Edellä mainittujen epämuodollisten menetelmien rinnalle on kehitetty kvasimuodollisia menetelmiä, joissa vertailun suorittamista varten on tehty peruslähtökohtia [Ost92]. Tällaisia menetelmiä ovat esimerkiksi [Sto03]:

1. Ideaalisen prosessimallin kuvaus ja muiden mallien vertaileminen siihen.
2. Prosessimalleille tärkeiden ominaisuuksien kokoaminen yhteen ja mallien vertailu tähän ominaisuuksien kokoelmaan.
3. Priori-hypoteesin muodostaminen prosessimallin toiminnasta ja sen vertailu eri prosessimalleista saatuihin empiirisiin tutkimustuloksiin.
4. Metakielen muodostaminen vertailun yhteiseksi pohjaksi, johon eri prosessimalleja verrataan.
5. Jonkin tietyn ongelman esittely ja eri prosessimallien tarjoamien ratkaisumahdollisuuksien vertailu.

Tässä yhteydessä sovelletaan ensin mainittua menetelmää ottamalla Extreme Programming ideaaliseksi prosessimalliksi ja arvioimalla muita menetelmiä sitä vasten. XP:tä ei kuitenkaan pidetä tässä yhteydessä täydellisenä menetelmänä. Lisäksi myös ketteriä ja perinteisiä menetelmiä verrataan toisiinsa yleisesti.

6.2 Ketterien ja perinteisten prosessimallien vertailua yleisesti

Tutkielman alussa ohjelmistokehitysmenetelmät jaettiin dokumentaation määrän perusteella kahteen ryhmään. Tämän lisäksi menetelmille voidaan löytää myös muita niitä kuvaavia piirteitä. Boehm on määritellyt seuraavassa esiteltävässä taulukossa kuvatut keskeiset osa-alueet sekä ketterille että perinteisille ohjelmistokehitysmenetelmille [Boe02].

Osa-alue	Ketterät menetelmät	Perinteiset menetelmät
Kehittäjät	Ketteriä, asiantuntevia, rinnakkaiseen työhön kykeneviä, samassa paikassa, yhteistyökykyisiä	Suunnitelmalähtöisiä, riittävät taidot, pääsy ulkoisiin tietolähteisiin
Asiakkaat	Omistautuneita, asiantuntevia, rinnakkaiseen työhön kykeneviä, yhteistyöhaluisia, riittävät valtuudet, samassa paikassa	Saavat tietonsa asiantuntevilta henkilöiltä, yhteistyöhaluisia asiakkaita
Vaatimukset	Ilmenevät vähitellen, muuttuvat nopeasti	Tiedetään varhaisessa vaiheessa, ovat pitkälti muuttumattomia
Arkkitehtuuri	Suunnitellaan tämän hetken tarpeisiin	Suunnitellaan tämän hetken ja tulevaisuuden tarpeisiin
Refaktorointi	Edullista	Kallista

Koko	Pienemmät kehitysryhmät ja projektit	Suuremmat kehitysryhmät ja projektit
Ensisijainen tavoite	Nopea hyödyn ja arvon tuottaminen	Korkea luotettavuus, toimintavarmuus ja virheettömyys

Taulukko 1. Ketterien ja perinteisten menetelmien tyypillisiä piirteitä. [Boe02]

Ohjelmistokehittäjille asetettavat vaatimukset poikkeavat toisistaan ketterissä ja perinteisissä menetelmissä. Osaavan ja asiantuntevan henkilöstön hankkiminen on keskeinen osa ketteriä menetelmiä, sillä useimmat niistä edellyttävät ohjelmistokehittäjiltä laajaa asiantuntemusta ja osaamista, hyvää kommunikaatiokykyä ja omistautumista työlleen [Boe02]. Eräänä ketterien menetelmien ongelmana onkin pidetty hyvien ohjelmistokehittäjien löytämistä. Perinteiset menetelmät eivät nojaa niin paljoa yksittäisten ohjelmistokehittäjien kykyihin kuin ketterät menetelmät. Yksilöiden tietojen ja taitojen puutteita voidaan täydentää laajemmilla suunnitelmilla, suuremmalla dokumentaatiomäärällä ja tiedon hakemisella muista lähteistä [Boe02].

Puhuttaessa toteutettavan järjestelmän arkkitehtuurista, tarkoitetaan sillä järjestelmän teknisiä ratkaisuja. Ketterissä menetelmissä kannustetaan käyttämään tekniikkaa, joka kykenee täyttämään minimissään juuri sen hetkisen tarpeen ja etenkin silloin, kun tulevaisuutta ei voida ennustaa. Tämä seikka voi kuitenkin aiheuttaa tehokkuuden laskua, mikäli tulevaisuuden tarpeet tiedetään. Perinteisissä menetelmissä ennustettavissa oleviin muutoksiin varaudutaan alusta alkaen etukäteismäärittelyillä ja arkkitehtuurisuunnitelmilla. Tällöin ohjelmistokehittäjille tarjoutuu mahdollisuus tehdä järjestelmään liityntöjä ja ominaisuuksia, jotka auttavat toteuttamaan uusia ominaisuuksia nopeasti. [Boe04]

Ketterissä menetelmissä tapahtuvan refaktoroinnin edullisuus perustuu siihen, että sitä tehdään jatkuvasti. Sen ideana on, että jatkuvasti tapahtuvat pienet parannukset ehkäisevät isojen ja kalliiden muutostöiden tarvetta. Boehmin mukaan tämä olettamus ei kuitenkaan aina pidä paikkaansa. Jatkuvan parantamisen menetelmä toimii silloin, kun kyseessä ovat melko pienet järjestelmät ja ohjelmistokehittäjät ovat taidoiltaan asiantuntijoita. Jatkuvasti tapahtuvaan rakenteen kehitystyöhön ei voida luottaa, mikäli järjestelmien koot kasvavat.

Tällöin hyväksi koetut yksinkertaiset arkkitehtuuriratkaisut eivät välttämättä mukaudu suurempiin järjestelmiin. [Boe04]

Ketterien menetelmien käyttö edellyttää yleensä läsnä olevaa ja ohjelmiston kehitystyöhön omistautunutta asiakasta, joka tuntee myös järjestelmän tarpeet. Tämä aiheuttaa ketterille menetelmille sen riskin, ettei asiakkaalta saada aina välttämättä käyttöön riittävän pätevää ja tarvittaessa käytettävissä olevaa asiantuntevaa edustajaa. Yritykset saattavat tarjota ohjelmistokehittäjien avuksi henkilöä, jolla on vähiten vastuuta ja vain vähän asiantuntemusta. Vastaavasti perinteiset menetelmät edellyttävät asiakasta, jonka kanssa voidaan etukäteen laatia tarkasti sopimukset, suunnitelmat ja määrittelyt [Boe04].

Merkittävin asiakasta kohtaava riski perinteisten menetelmien kohdalla koituu liian tarkasta sopimuksien laatimisesta. Niiden liian tarkka seuraaminen saattaa johtaa keskittymisen harhautumiseen, jolloin varsinaisen ohjelmistokehitystyön sijasta keskitytään liiaksi byrokratiaan [Boe04]. Ketterissä menetelmissä määrittelyt kuvataan epämuodollisiksi ja joustaviksi tarinoiksi. Tämän tekee mahdolliseksi nopeassa syklissä tapahtuva uusien versioiden tuottaminen ja luottamus siihen, että tarvittavat muutokset kyetään myös toteuttamaan. Edellisen version valmistuttua asiakkaan kanssa voidaan neuvotella seuraavaan versioon tarvittavista uusista ominaisuuksista [Boe04].

Ketterät menetelmät olettavat ettei pitkälle menevä ja tarkka tulevaisuuden suunnittelu ole kannattavaa. Suunnitelmat eivät kuitenkaan pidä ja muutoksia tulee aina. Perinteiset menetelmät suosivat yksityiskohtaisia, muodollisia ja tarkasti dokumentoituja määrittelyjä. Niiden heikkoutena on sopeutumattomuus muutoksiin ja vaikeus asettaa ominaisuuksia tärkeysjärjestykseen. Perinteisten menetelmien menettelytavoissa on kuitenkin omat vahvuutensa. Luotettavuus, suorituskyky ja skaalautuvuus tulevat paremmin otetuksi huomioon perinteisissä menetelmissä. Nämä seikat ovat tärkeitä etenkin laajoissa, kriittisiä tehtäviä hoitavissa järjestelmissä [Boe04].

Arvioitaessa ketterien menetelmien soveltuvuutta erilaisiin ohjelmistoprojekteihin, on otettava huomioon muutamia niiden käyttöä rajoittavia seikkoja [Tur02].

1. Hajautetut kehitysympäristöt

2. Ohjelmistojen alihankinta
3. Uudelleenkäytettävien ohjelmistokomponenttien kehittäminen
4. Suuret ohjelmistokehitysryhmät
5. Turvallisuuskriittiset ohjelmistot
6. Laajat ja monimutkaiset ohjelmistot

Työskenneltäessä hajautetussa kehitysympäristössä, on ketterissä menetelmissä korostettu kasvokkain tapahtuva kommunikointi vaikeaa, vaikka nykyaikaisilla viestintävälineillä tätä tilannetta voidaankin parantaa. Sovittavan osakokonaisuuden rajaaminen on ketterien menetelmien työtavoilla haastavaa, mikäli käytetään alihankintaa. Keinoksi tähän on ehdotettu alihankintasopimusten toteuttamista kaksiosaisena. Tällöin sopimuksen ensimmäinen osa kuvaa keskeisen perusrakenteen. Toinen osa on muuttuva, jolloin se voi sisältää vaihtelevan määrän ominaisuuksia ja työtä. [Tur02]

Ketterillä menetelmillä kehitettyjen ohjelmistokomponenttien uudelleenkäytettävyys on usein kyseenalaistettu ketterien menetelmien perustavoitteen takia. Tällä tarkoitetaan juuri tällä hetkellä käsillä olevan ongelman ratkaisevan ohjelmiston tuottamista. Suurien ohjelmistokehitysryhmien kohdalla projektiorganisaation hallinnointi edellyttää useita kommunikaatioväyliä, jotta kokonaisuutta voidaan hallita. Tähän ketterissä menetelmissä tapahtuva kommunikointitapa, joka perustuu kasvokkain tapahtuvaan keskusteluun, ei välttämättä skaalaudu. [Tur02]

Turvallisuuskriittiset ohjelmistot ovat laadunvarmistuksen kannalta haastavia. Turk [Tur02] esittääkin epäilyksen, etteivät ketterät menetelmät ole riittäviä tämän tyyppisille ohjelmistoille. Perinteisten menetelmien tarkasti määritellyt testitapaukset ja testaussuunnitelmat tarjoavat tähän paremman, mutta samalla myös kalliimman ja hitaamman tavan. Laajojen järjestelmien kohdalla todetaan, ettei koodin refaktorointi poista suunnittelun tarvetta laajoissa järjestelmissä. Kun ohjelmisto ylittää tietyn koon, ei ohjelmistokomponenttien uudelleenohjelmointi ole enää järkevää.

6.3 XP:n vertailua perinteisiin prosessimalleihin

Vaihejakomalli poikkeaa eniten XP:n ja vesiputousmallin välillä. Vesiputousmallissa koko ohjelmisto toimitetaan kerralla ja aika projektin aloittamisesta ensimmäiseen toimitukseen kestää nyt esille tulleista prosessimalleista kauimmin [Hai04]. Vastaavasti XP:ssä ohjelmisto pyritään toimittamaan mahdollisimman pienissä osissa, joten ensimmäinen toimitus asiakkaalle tapahtuu vesiputousmallia nopeammin. Vesiputousmallissa ohjelmiston arkkitehtuuri rakennetaan jo suunnitteluvaiheessa [Hai04]. Myös spiraalimallissa ja RUP:ssa on pyrkimyksenä suunnitella ohjelmiston arkkitehtuuri projektin alussa, vaikka tulevat iteraatiokierrokset tuovatkin siihen muutoksia. Sen sijaan evoluutiomallissa arkkitehtuuri muotoutuu ohjelmiston evoluution myötä. XP:ssä arkkitehtuurisuunnittelua ei tehdä omana vaiheenaan lainkaan, vaan se on korvattu refaktoroinnin käytöllä [Bec00].

XP:ssä ohjelmakoodi on tärkein tuotoksista. Muita tuotoksia ei määritellä ja yksinkertaisuus toimii ainoana ohjenuorana. Erilaisten dokumenttien laatimista tai mallien käyttöä ei ole XP-mallissa ohjeistettu. Mikäli dokumentin laatiminen katsotaan tarpeelliseksi tai jotakin mallia tarvitaan, niitä voidaan laatia tai käyttää. Suurin painoarvo on itse ohjelmakoodissa, jonka voidaan katsoa olevan XP:tä käyttävän projektin ainoa merkityksellinen tuotos [Bec00]. RUP:ssa tuotoksilla on suuri merkitys. Niinpä siinä onkin tarkasti määritelty, kenen tehtävänä on laatia dokumentti jonkin toiminnon seurauksena [Kru01].

Ohjelmiston integrointi tapahtuu vesiputousmallissa kerran ja vain kerran [Hai04]. Sen sijaan muissa perinteisissä malleissa ohjelmisto integroidaan jokaisen iteraatiokierroksen päätteeksi. XP:ssä integrointia harrastetaan eniten ja se tehdään päivittäin [Bec00]. Spiraalimallissa iteraatiokierroksia on huomattavasti vähemmän. Evoluutiomalli toistaa myös iteraatioita, mutta iteraatiokierrosten lukumäärä ei ole etukäteen selvillä.

Ohjelmiston testauksessa esiintyvät erot ovat pitkälti integroinnissa esiintyvien erojen tyyppisiä. Vesiputousmallissa ohjelmisto testataan vain kerran, muissa perinteisissä malleissa ohjelmisto testataan jokaisen iteraation lopuksi [Hai04]. XP on testauslähtöinen prosessi ja sitä suoritetaan jatkuvasti [Bec00]. Jatkuva testaaminen tarjoaa runsaasti palautetta järjestelmästä sekä paljastaa mahdolliset virheet välittömästi niiden ilmestyessä. XP-mallissa testi kirjoitetaan ennen koodia, jolloin se toimii samalla myös määrityksenä tule-

valle koodille. Ennen koodausta tapahtuva testaus erottaa XP-mallin merkittävästi vesiputousmallista ja RUP:sta.

XP:ssä kaikki testit automatisoidaan ja lisätään kasvavaan testisarjaan, jota ajetaan jatkuvasti [Bec00]. Testausta suoritetaan jokaisen integroinnin yhteydessä ja testauksen automatisointi on hyvin tärkeää integroinnin tehokkaan onnistumisen kannalta. Kunkin iteraation päätteeksi tuotettava julkistus testataan vielä erikseen hyväksymistestein, joiden sisällön asiakas on suunnitellut [Bec00]. Myöskään RUP:ssa tapahtuva testaus ei ole erillinen vaihe tai toiminto, vaan muun toiminnan kanssa tapahtuvaa jatkuvaa toimintaa, jonka tarkoituksena on paljastaa virheet aikaisessa vaiheessa ja tarjota palautetta kehityksen alla olevasta järjestelmästä [Kru01].

Asiakkaan rooli XP:ssä poikkeaa perinteisistä menetelmistä. Asiakkaan roolina on perinteisesti ollut antaa määrittelyt ja osallistua projektin ohjausryhmään. Sen sijaan XP:ssä asiakas on osa projektiryhmää ja aina läsnä antamassa tarvittaessa lisätietoja ja suorittamassa katselmointia [Bec00]. Merkittävin ero XP-mallin ja perinteisten menetelmien välillä on XP-mallissa toteutettavat pariohjelmoinnin, refaktoroinnin ja koodin yhteisomistuksen käytännöt [Bec00]. Näistä pariohjelmointi on XP-mallin merkittävä ominaispiirre, joka käsittää kaikki ohjelmistotuotantoprosessin sisäiset aktiviteetit määrittelystä testaukseen ja koodin tarkastukseen, ohjelmoinnin ollessa vain yksi meneillään olevista aktiviteeteista [Bec00].

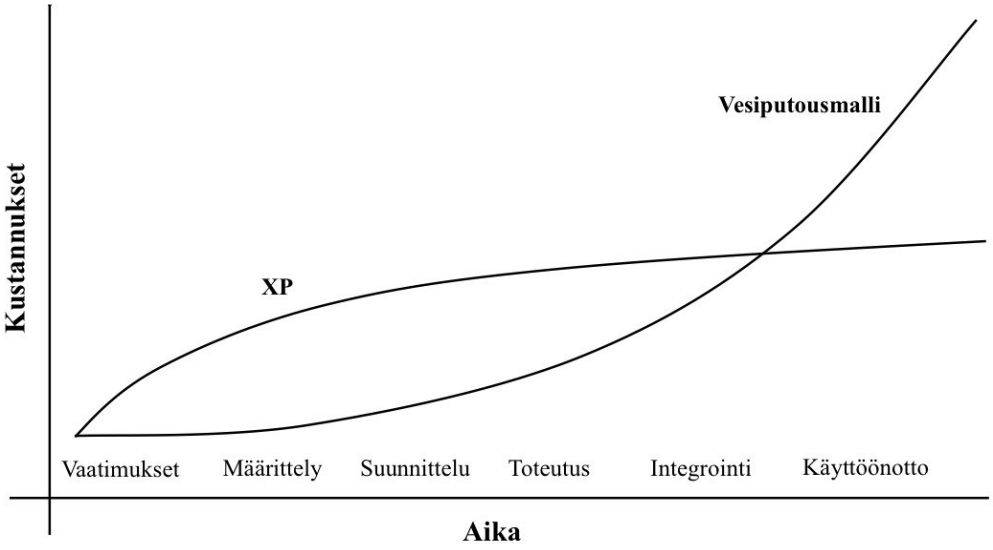
Pariohjelmointi tarjoaa myös kumulatiivisen vaikutuksen muihin ohjelmistotuotannon osa-alueisiin, sillä ohjelmistoprojektissa on joka tapauksessa tehtävä suunnittelupäätöksiä, tuottaa, arvioida ja testata ohjelmakoodia, refaktoroida sitä ja lisäksi koko kehitysryhmän on kommunikoitava keskenään. Astels toteaa, että pariohjelmoinnissa ongelmia saattaa aiheuttaa yhteinen kehitysympäristö. Tällä tarkoitetaan sitä, että jotkut jäsenet ovat niin kiintyneitä johonkin työvälineeseen, että heidän on pakko totuttaa koko kehitysryhmä kyseisen työvälineen käyttöön. Myös jatkuva integrointi saattaa aiheuttaa ongelmia, sillä ohjelmointiparit saattavat tehdä ristiriitaisia muutoksia samaan koodiin. Tätä ei kuitenkaan tapahdu kovin usein. [Ast02]

Ohjelmistoprosessimalleja voidaan arvioida myös sen perusteella, kuinka joustavasti muutostenhallinta voidaan saada sujumaan, kuinka usein ohjelmistoa testataan ja kuinka usein tai nopeasti ohjelmiston toimivia osia voidaan toimittaa asiakkaalle. Tämä järjestys kulkee vesiputousmallista spiraalimallin ja evoluutiomallin kautta XP:hen. Muutostenhallinnan käytännöt ovat byrokraattisimmat vesiputousmallissa, jossa muutokset täytyy hyväksyttää projektin ohjausryhmällä, jonka jälkeen ne vasta voidaan toteuttaa. Muutosten toteuttaminen vaatii usein paluuta aikaisempiin vaiheisiin, erityisesti suunnitteluvaiheeseen [Hai04]. XP:ssä, spiraalimallissa ja evoluutiomallissa voidaan tehdä muutoksia jokaisen käyttöönoton jälkeen tapahtuvassa palaverissa, jossa kartoitetaan seuraavan version ominaisuuksia, joten muutostenhallinta on niissä helpointa.

XP eroaa merkittävästi vesiputousmallista, koska näiden mallien perusoletus ohjelmistoprojektin toimintaympäristöstä ja sen stabiilisuudesta poikkeavat toisistaan suuresti. Vesiputousmallissa ohjelmiston määrittelyjen ja teknologian oletetaan pysyvän samana koko ajan. Vastaavasti XP olettaa toimintaympäristön olevan epästabiilin ja siksi sen muutosprosessi on tehty kevyeksi [Bec00].

Kevyeen dokumentaatioon ja juuri ennen ohjelmointia tapahtuvaan suunnitteluun ei ole sitoutunut XP:ssä niin paljoa pääomaa kuin vesiputousmallissa, joten projektissa tapahtuvat suuretkaan muutokset eivät tule niin kalliiksi kuin vesiputousmallissa. Muut perinteiset menetelmät olettavat toimintaympäristön vesiputousmallia labiilimmaksi, mutta stabiilimmaksi kuin XP:ssä. Ne hyödyntävät iteraatioiden tuomaa muutoksenhallinnan tehokkuutta, mutta eivät mene yhtä pitkälle kuin XP.

Kuvassa 23 havainnollistetaan, miten muutoksen aiheuttamat kustannukset muuttuvat perinteisen vesiputous- ja XP-mallin mukaisen ohjelmistotuotantoprosessin eri vaiheissa. XP-mallin kustannusten muutosta kuvaavaa käyrää voidaan soveltaa myös muihin iteratiivisiin prosesseihin.



Kuva 23. Kustannusten muutokset vesiputous- ja XP-mallissa. [Ast02] [Bec00]

Kuvasta 23 voidaan todeta, miten vaatimusten muuttumisen kustannukset kasvavat vesiputousmallissa sitä suuremmiksi, mitä myöhemmässä vaiheessa järjestelmän vaatimuksia muutetaan. XP auttaa pitämään kustannusten kasvun pienenä, mikä tarkoittaa sitä, että vaatimusten muuttuminen missä tahansa projektin vaiheessa maksaa suunnilleen saman verran.

Asiakkaalle tärkeä kustannusten ennustettavuus näyttäisi olevan vesiputousmallissa paras, sillä siinä kustannukset voidaan laskea varsin tarkasti etukäteen, jos voidaan olettaa, ettei kustannuksia kasvattavia muutoksia tule kovin runsaasti. Sen sijaan projektit, joiden hallintaan valitaan XP, voivat olla luonteeltaan ennalta arvaamattomia, mistä myös seuraa, ettei niiden kustannusten ennustettavuus ole yhtä hyvä. Lisääntyvän joustavuuden mukana tulee vaikeus ennustaa projektista koituvia kustannuksia pitkällä tähtäimellä. Näin ollen ohjelmistoprojektille on vaikea hahmottaa lopullista hintaa.

6.4 XP:n vertailua muihin ketteriin menetelmiin

Tarkasteltaessa XP:n ja muiden ketterien menetelmien välillä olevia eroavaisuuksia, voidaan huomata, ettei XP:tä vastaavia ohjeita pariohjelmoinnista, testauksen automatisoinnis-

ta tai koodin yhteisomistuksesta löydy muista ketteristä menetelmistä. XP antaa huomattavasti tarkempia ohjeita joka päivä tapahtuvaan ohjelmointityöhön kuin muut ketterät menetelmät. Vastaavasti muut ketterät menetelmät ovat luonteeltaan enemmän hallinnollisia. Tästä erosta on noussut esiin ajatus yhdistellä XP:n käytäntöjä muiden ketterien menetelmien kanssa ja luoda näin uusia menetelmiä. Eri ketteriä menetelmiä yhdistelevistä kokeiluista on jo tarjolla tuloksia ja tämän kaltaisten yhdistelmien merkitys saattaa kasvaa tulevaisuudessa.

Toisin kuin useimmissa muissa ketterissä menetelmissä, XP:ssä ei ole ohjelmiston arkkitehtuurin teknistä suunnitteluvaihetta. XP ei myöskään ota kantaa mihinkään projektinhallinnalliseen toimintaan. Kaikki projektinhallinta on siten tavallaan asiakkaan vastuulla, vaikkei tätä vastuuta asiakkaalle varsinaisesti määritelläkään. Scrum puolestaan määrittelee vain tarpeelliset roolit [Abr02]. Scrum-menetelmässä työnjaon kuvaaminen ja roolien tiedottaminen on projektin ulkopuolellekin siten helppoa. Prosessimallin pitäisi myös kyetä parantamaan sitä luottamusta, joka näyttää yleensä puuttuvan ohjelmistoprojekteissa. Tämän seikan korjaamiseksi XP ei tarjoa mitään merkittävää uutuutta. Sen sijaan Scrumin on raportoitu korjanneen luottamusta, jopa silloinkin kun käyttäjät eivät ole tienneet käyttävänsä sitä [Schw06].

Ketterille menetelmille sallituissa projektiryhmän koostumuksissa on menetelmien välillä eroja. XP sallii vain yhden, kiinteässä yhteistyössä toimivan projektiryhmän ja tämä seikka omalta osaltaan rajaa toteutettavan projektin kokoa ja XP:n soveltuvuutta suuriin ohjelmistoprojekteihin. Vastaavasti Crystal-menetelmäperhe ja DSDM sallivat suuremmat projektiryhmät [Abr02]. Ketterissä menetelmissä ei yleensä käytetä prototyyppejä. Tästä on poikkeuksena kuitenkin DSDM, jossa prototyyppejä tehdään siten, että prototyypin koodia käytetään myös lopullisessa järjestelmässä [Abr02]. Crystal-menetelmäperhe ja Scrum tukevat myös maantieteellisesti hajaantuneita projektiryhmiä [Coc02] [Schw06].

Ketterissä menetelmissä valmistettavan dokumentoinnin määrä vaihtelee eri menetelmissä. XP määrittelee projektissa tuotettavan dokumentaation vain käyttötapausten kirjaamisen osalta [Bec00]. XP:n luonteeseen kuuluu liian dokumentaation välttäminen, mutta yleisenä ohjeena on, että sitä tehdään harkinnan ja tarpeen mukaan riittävästi [Bec00]. DSDM:ssä

tuotetaan eniten dokumentaatiota, sillä siinä vaiheiden lopputulokset määritellään tarvittavien dokumenttien avulla ja dokumentaatiota on XP:tä huomattavasti enemmän [Abr02].

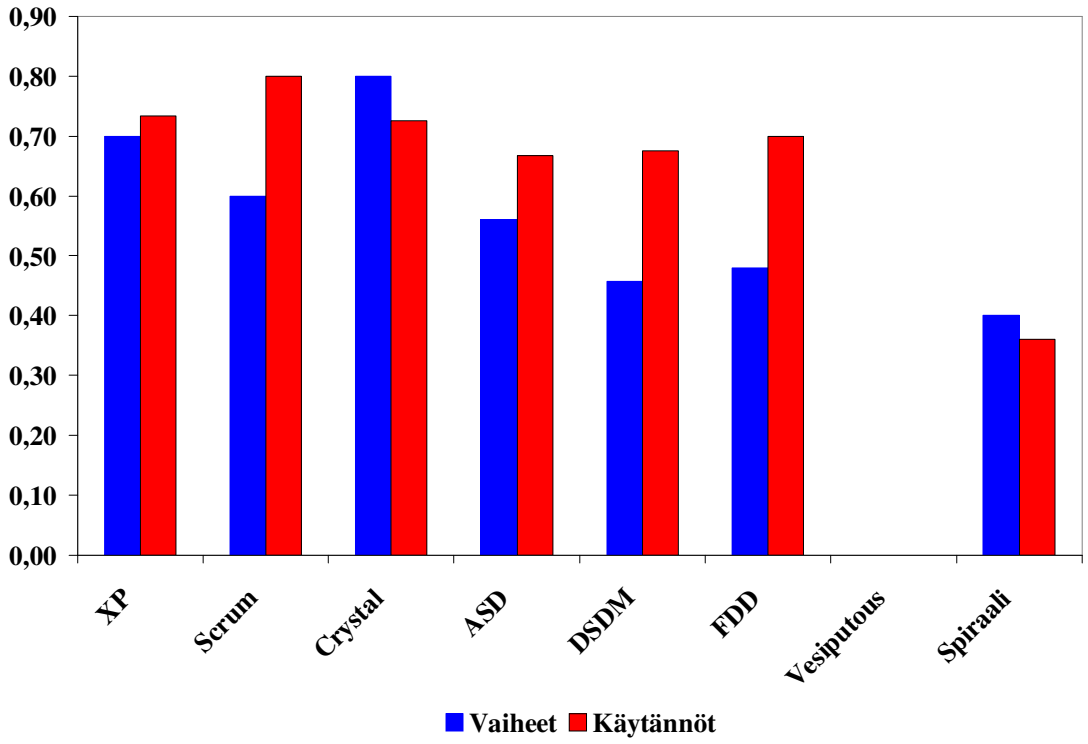
Aikataulun tärkeyttä korostetaan eri tavalla menetelmien välillä. Scrum-, ASD- ja DSDM-menetelmissä iteraation pituus on ennalta tarkkaan määritelty [Abr02]. XP ja muut ketterät menetelmät antavat asiakkaalle mahdollisuuden päättää, joustetaanko iteraation laajuudesta, ominaisuuksista vai kehitysajasta. Sen sijaan missään menetelmässä ei sallita joustamista laadun suhteen. XP ja Scrum arvioivat kevyemmin ajan, jossa vaatimuksia saadaan toteutettua. Tämän jälkeen koko kehitysryhmä osallistuu aikataulun laatimiseen ja asiakas vastaa mielekkään kokonaisuuden toteuttamisesta.

XP:ssä projektin edistymistä seurataan käyttäjäkertomuksista johdettujen tehtävien tiloilla [Bec00]. XP ei kuitenkaan kerro, miten tämä tieto saadaan raportoitua ja esitettyä eteenpäin tuotekehityksen ulkopuolelle. Ongelmien raportointia ei käsitellä XP-mallissa. Scrum puolestaan kertoo konkreettisesti, miten raportointi tehdään. Scrum myös käsittelee ongelmien raportoimisen ja myös kertoo monissa tilanteissa, kuinka niihin on reagoitava [Schw06]. Asiakkaalta tulevien vaatimusten saaminen tuotekehitykselle selkeinä ja ymmärrettävinä on vaikeaa. Ainoastaan XP käsittelee hyvin sitä, kuinka vaatimuksista saadaan helposti omaksuttavia. Vastaavasti Scrum ei tarjoa kovin hyviä vastauksia kohdeyrityksen vaatimuksia koskeviin ongelmiin.

Qumer ja Henderson-Sellers [Hen07] ovat tutkineet kvantitatiivisesti ketterien menetelmien ketteryyden astetta. Tutkittujen menetelmien vaiheet ja käytänteet oli pisteytetty sen mukaan, tukivatko ne ketterien menetelmien piirteitä vai eivät. Tässä tutkimuksessa tarkasteltuja menetelmiä olivat XP, Scrum, Crystal-menetelmäperhe, ASD, DSDM ja FDD. Vertailun vuoksi mukana oli myös kaksi perinteistä ja yhä yleisesti käytettyä prosessimallia, eli vesiputous- ja spiraalimallit.

Tutkimuksesta saaduista tuloksista voidaan nähdä, että ketterissä menetelmissä on eroja menetelmien vaiheiden ja käytäntöjen ketteryyden suhteen. Voidaan myös huomata, että kaikki ketterät menetelmät voidaan luokitella selkeästi ketterämmäksi kuin nyt vertailussa mukana olleet perinteiset menetelmät. Tuloksia voitaneen kuitenkin pitää korkeintaan

suuntaa-antavina. Kuvassa 24 esitetään saadut tulokset. Mitä suuremman arvon menetelmä on saanut, sitä ketterämpiä ovat sen vaiheet ja käytännöt. [Hen07]

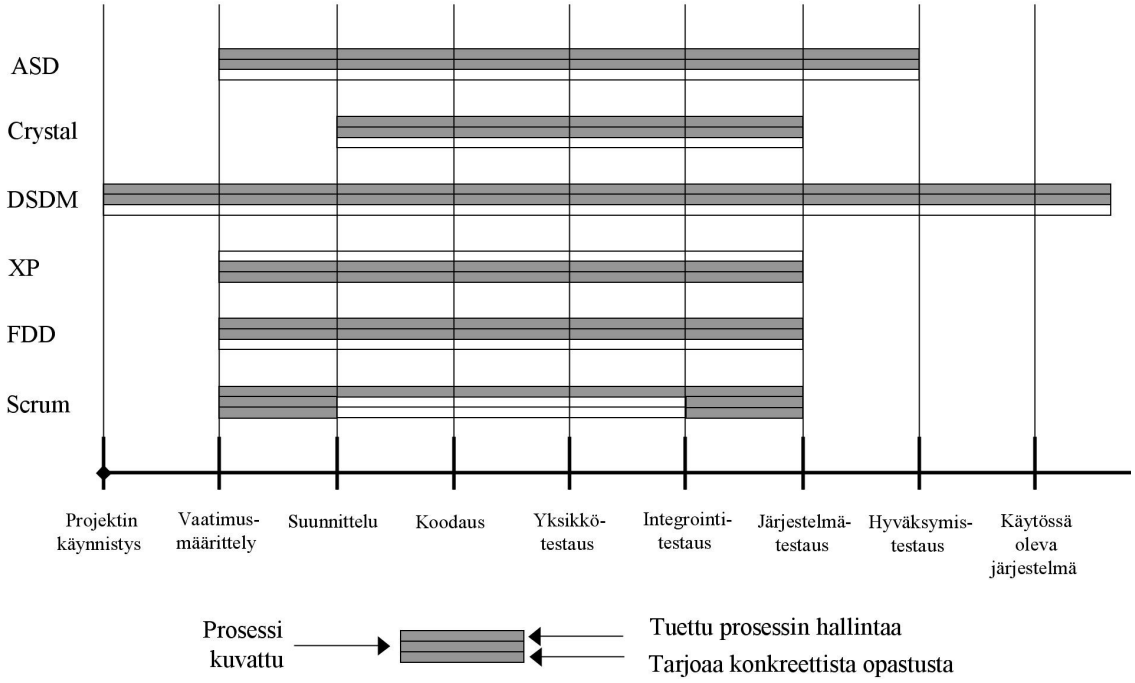


Kuva 24. Ketterien menetelmien ketteryyden taso. [Hen07]

Abrahamsson [Abr02] on puolestaan vertaillut ketterien menetelmien tarjoamaa tukea projektin-hallinnan, prosessin ja konkreettisen ohjeistuksen osalta suhteessa ohjelmistotuotannon elinkaaren vaiheisiin. Vertailun mukaan eri ketterät menetelmät keskittyvät eri osiin ohjelmiston elinkaaressa sekä myös eri tasoille. Tässä yhteydessä tasoina mainitaan prosessi, projektinhallinta ja konkreettiset työskentelykäytännöt. Abrahamssonin mukaan XP keskittyy erityisesti konkreettisiin työskentelykäytäntöihin ja prosessiin, mutta ei lainkaan projektinhallintaan. Vastaavasti Scrum keskittyy suureksi osaksi vain projektinhallintaan ja siinä ilmeneviin ongelmiin.

Abrahamsson esittää näkemyksensä ketterien menetelmien erilaisuudesta kuvassa 25 [Abr02]. Vaiheet esitetään kuviossa siten, että kukin vaihe alkaa nimen osoittamasta koh-

dasta päättyen sen oikeanpuoleiseen päätepisteeseen. Ylin palkki kuvaa miten hyvin kukin menetelmä tukee projektinhallintaa. Keskimmäinen palkki kertoo mitä ohjelmistotuotannon elinkaaren vaiheita menetelmä kattaa. Alin palkki kuvaa millä tarkkuudella menetelmä tarjoaa konkreettista ohjeistusta. Harmaa väri palkissa osoittaa, että kukin menetelmä kattaa tarkasteltavan kriteerin kyseisessä elinkaaren vaiheessa, kun valkoinen väri puolestaan osoittaa tuen puuttumisen.



Kuva 25. Ketterien menetelmien projektinhallinnan, prosessin ja konkreettisen ohjeistuksen kattavuuden vertailu [Abr02]

Kuvasta havaitaan, ettei nyt tehdyn tarkastelun kriteereillä ole tällä hetkellä olemassa täysin kattavaa ketterää menetelmää, vaan jokainen menetelmä on keskittynyt tarjoamaan tietynlaista tukea joihinkin ohjelmistokehityksen elinkaaren vaiheisiin. Jotkut menetelmät, kuten DSDM, pyrkivät kattamaan ohjelmistotuotannon elinkaaren kokonaan. DSDM kattaa projektinhallinnan ja tarjoaa kuvatun prosessin, muttei tarjoa kuitenkaan konkreettista opastusta. Toiset ketterät menetelmät puolestaan keskittyvät enemmän konkreettisen opas-

tuksen tarjoamiseen. Kuvan 25 perusteella voidaan myös havaita, miten XP ja Scrum täydentävät hyvin toistensa puutteita menetelmien kattavuudessa.

6.5 XP:n soveltuvuus käytännössä

Tässä yhteydessä pyritään arvioimaan millaisiin projekteihin XP soveltuu ja millaisiin eivät sovellu. Tämän jälkeen tarkastellaan XP:n käytöstä saatuja käytännön kokemuksia ja sitä kohtaan esitettyä kritiikkiä, sekä menetelmän käytön rajoitteita ja reunaehtoja.

6.5.1 Projektimalleja, joihin XP soveltuu

XP:n voidaan katsoa olevan parhaimmillaan ohjelmistoprojekteissa, joissa vaatimukset eivät ole selvillä tai niiden uskotaan muuttuvan ohjelmistoprojektin aikana. Tällöin XP:n kyky reagoida nopeasti muutoksiin auttaa saavuttamaan hyötyä. Hyötyä koituu myös siitä seikasta ettei suunnitteluun ja dokumentointiin käytetä ylimääräistä aikaa, kun ennustaminen olisi vaikeaa joka tapauksessa. [Bec00]

Maurer [Mau02] esittää XP:n toimivan parhaiten 5-15 henkeä käsittävissä ohjelmistokehitysryhmissä. XP:n sopiminen parhaiten pieniin ohjelmistoprojekteihin johtuu osaltaan XP:ssä noudatettavasta koodin yhteisomistuksesta. Koska projektin kaikkien jäsenten on tunnettava kaikki projektissa toteutettu koodi tästä seuraa, että projektin on oltava hahmotettavissa olevan kokoinen. Astels [Ast02] esittää kuitenkin, että myös XP:tä on mahdollista skaalata. Suuret ohjelmistoprojektit voidaan jakaa aliprojekteihin, eikä ole mahdotonta viedä aliprojekti lävitse käyttäen XP:n menetelmiä. [Tab00].

Hyvä kohde XP:n käytölle ovat ohjelmistoprojektit, joissa sovellusalue on uusi tai vaikea ja jota projektihenkilöstö ei kokonaisuudessaan täysin ymmärrä. Asiakkaan kanssa tapahtuva tiivis yhteistyö lisää ymmärrystä alueesta ja iteraatioiden avulla virheet huomataan nopeasti. Myös uudella ja vielä kehittyvällä teknologialla tehtävät ohjelmistoprojektit ovat hyvä kohde XP:n käytölle. Uusi teknologia saattaa tuottaa helposti yllätyksiä, jolloin kaikkia vaatimuksia ei voida toteuttaa ja projektissa joudutaan muuttamaan suunnitelmia. Tällöin XP:ssä suoritettava vähäisempi dokumentaatio projektin alussa säästää kustannuksia. [Bec00]

XP:n käyttöä voitaneen myös harkita projekteissa, joissa perustoiminnallisuus on saatava hyvin nopeasti käyttöön ja tätä XP:n lyhyt iteraatiosykli tukee. Projektit, jotka tuottavat teknisesti helposti päivitettäviä ohjelmistoja voivat olla erinomaisia XP:n käytön kohteita. Tällaisia ohjelmistotuotteita ovat esimerkiksi palvelinpohjaiset, selainkäyttöliittymällä toimivat ohjelmistot.

6.5.2 Projektimalleja, joihin XP ei sovellu

Beck [Bec00] suosittaa, että yli kahdenkymmenen hengen ohjelmistoprojektit tehdään muilla menetelmillä kuin XP:llä. Beckin [Bec00] mukaan XP ei myöskään sovi ohjelmistoprojekteille, joissa ei ole varaa tehdä virheitä. XP kannustaa rohkeuteen ja kenellä tahansa on mahdollisuus tehdä tarpeen mukaan muutoksia mihin tahansa koodin osaan ja tästä voi seurata virheitä. Sellaisten ohjelmistojen rakentaminen, joita on vaikeaa tai mahdotonta päivittää käyttöönoton jälkeen, ovat huonoja käyttökohteita iteraatioita käyttävälle XP:lle. Myös ohjelmistoprojektit, joissa asiakkaan on tarkasti tiedettävä jo ennakkoon projektiin kuluvat aika- ja kustannusresurssit eivät sovi XP:n käyttökohteeksi, sillä XP:ssä lopullinen resurssien tarve tiedetään vasta ohjelmistoprojektin päättymisen jälkeen.

Turkin [Tur02] mukaan erittäin korkeaa luotettavuutta vaativat ohjelmistoprojektit eivät myöskään sovi toteutettavaksi XP:llä, sillä siinä ei tarjota esimerkiksi muodollisia katselmointeja tai tapoja mitata testikattavuutta, jotka ovat usein vaatimuksina kaikkein kriittisimmissä järjestelmissä. XP:ssä ei ole myöskään hyviä keinoja ei-toiminnallisten ominaisuuksien määrittelyyn. Myös eri organisaatioiden tapa kilpailuttaa määrittely- ja toteutusprojekteja erikseen estää XP:n käytön. XP toimiikin parhaiten käytettäessä joustavaa sopimusmenettelyä.

Beckin [Bec00] mukaan ohjelmistoprojektit, joissa henkilöstö on maantieteellisesti jakautunut, on parempi toteuttaa käyttäen jotain muuta menetelmää. Suuret maantieteelliset etäisyydet henkilöstön välillä vaikeuttaa XP:lle tärkeitä työtapoja, kuten henkilökohtaista kommunikointia, pariohjelmointia ja koodin yhteisomistusta. Tämä seikka samalla usein sulkee pois myös ohjelmistoprojektit, joissa osa ohjelmointityöstä suoritetaan alihankintana. Toisaalta myös pienet ohjelmistoprojektit, joissa on alle neljä henkilöä, ovat huonoja

kohteita XP:n käytölle. Niin pienissä ohjelmistoprojekteissa ei ole mahdollista käyttää kunnolla hyväksi pariohjelmointia.

XP:n käyttöä kannattaa välttää sellaisissa projekteissa, joita aloitettaessa voidaan epäillä, ettei asiakkaalla ole riittävästi aikaa tai halua osallistua projektiin. Suurimpana esteenä XP:n käytölle Beck mainitsee organisaatiot, joilta puuttuu tarve tai rohkeus kokeilla uusia menetelmiä. XP:hen tulee siirtyä, jos prosessien laatu ei tyydytä organisaatiossa. Toisaalta vaikka prosesseissa olisikin parannettavaa, saattavat organisaatiokulttuurit olla kykenemättömiä ja joustamattomia XP:n vaatimiin muutoksiin. [Bec00]

Henkilöstön huono yhteishenki ja ryhmähengen puuttuminen ovat vakavia uhkia jokaiselle ohjelmistoprojektille. XP:tä käyttävissä projekteissa uhka korostuu entisestään, koska ohjelmakoodi tuotetaan pariohjelmointina ja tietoa välitetään henkilökohtaisesti. Henkilöstön vaihtuessa tieto saattaa kadota ja sen uudelleen saaminen voi olla vaikeaa. [Bec00].

6.5.3 Case-tutkimuksia XP:n käytöstä

XP:n kohtaan tunnettu mielenkiinto on tuottanut useita case-tutkimuksia, joista tässä yhteydessä lyhyesti tarkastellaan muutamaa yleisesti tunnettua. Chryslerin C3-projekti oli historian ensimmäinen XP:tä käyttänyt ohjelmistoprojekti ja sitä esitellään usein XP:n käytön malliesimerkkinä. Toinen tapaus kertoo XP:n käyttöönotosta yksittäisessä ohjelmistoprojektissa eräässä suuressa yrityksessä, jossa aiemmin oli käytetty vain perinteisiä ohjelmistotuotannon menetelmiä.

C3 eli Chrysler Comprehensive Compensation System oli Chrysler -konsernin lähes sadantuhannen työntekijän palkan maksuun tarkoitettu ohjelmisto. Projekti alkoi tammikuussa 1995 ja maaliskuussa 1996 se oli ajautunut vaikeuksiin noudatettuaan siihen asti perinteisiä ohjelmistotuotannon menetelmiä. Tässä vaiheessa XP:n kehittänyt Kent Beck tuli mukaan projektiin ja menetelmäksi vaihdettiin XP. Tämän jälkeen projekti alkoikin saada nopeasti tuloksia aikaan, muttei koskaan onnistunut täysin saavuttamaan tavoitteitaan [Ker02] [Wik07a].

Parhaimmillaan ohjelmisto pystyi maksamaan palkkaa noin 90 prosentille työntekijöistä. Saksalainen Daimler-Benz osti Chrysler-yhtiön vuonna 1998, minkä jälkeen yhtiö on tunnettu nimellä DaimlerChrysler. Uusi yhtiö lopetti C3-projektin keskeneräisenä helmikuussa 2000 ja myöhemmin XP:n käytön projektinhallinnassa. Jonkin aikaa myöhemmin DaimlerChrysler kuitenkin otti uudelleen XP:n käyttöönsä. [Ans08][Wik07a]

Asiakkaalla oli selvä näkemys C3-projektissa siitä, millaisen valmiin ohjelman tulisi olla ja kuvaili sen 145 helposti ymmärrettävän käyttäjäkertomuksen muodossa. Ohjelmistokehitysryhmän keskuudessa oli jatkuvasti asiakasta edustava henkilö ja ohjelmoijat saattoivat milloin tahansa kysyä ohjelmiston toiminnallisuuteen liittyvistä asioista. Projektin puolivälissä asiakasta edustava henkilö kuitenkin vaihtui ja uuden henkilön laatimat vaatimukset ja käyttäjäkertomukset eivät olleet enää yhtä helposti ymmärrettäviä [Bec00]. Jeffriesin [Deu01] mukaan asiakasta edustavan henkilön vaihtuminen johti lopulta koko projektin epäonnistumiseen. Tämän perusteella Jeffries suositteleekin asiakkaan opastamista XP:n käytössä ja käyttäjäkertomusten kirjoittamisessa.

Grenning [Gre01] kertoo artikkelissaan XP:n käyttöönotosta yksittäisessä ohjelmistoprojektissa eräässä suuressa yrityksessä, jossa aiemmin oli käytetty vain perinteisiä ohjelmistotuotannon menetelmiä. Hänen mukaansa XP:n käyttöönotto suuryrityksessä aiheutti ohjelmistonkehittäjissä suurta muutosvastarintaa. Yritys tuotti turvallisuuden kannalta kriittisiä järjestelmiä luottaen vesiputousmalliin ja tiukkoihin käytäntöihin. Yrityksessä ei kuitenkaan oltu tyytyväisiä saatuihin tuloksiin, sillä ohjelmistoissa esiintyi katselmoinneista huolimatta virheitä ja projektien aikataulut ylittyivät. Näistä syistä johtuen yritys päätti kokeilla olio-ohjelmointia, käyttötapauksia ja iteratiivisia menetelmiä, palkaten Grenningin viemään läpi muutosta organisaatiossa. Grenning halusi mennä pidemmälle ehdottaen, että yritys kokeilisi XP:n käytäntöjä.

Grenning [Gre01] tiesi, että XP:n käyttöönotto kokonaisuudessaan yhdellä kerralla herättäisi henkilöstössä suurta vastustusta yrityksen organisaatiokulttuurin vuoksi. Yrityksen johto ja henkilöstö oli saatava ymmärtämään ja tukemaan XP:n tapaa hallita ohjelmistoprojektia. Grenning päätti ottaa käyttöön XP:n soveltaen sen käytäntöjä. Vaatimusten määrittelyssä vaatimukset kirjattiin yrityksen entisen prosessin mukaisina käyttötapauksina. Yri-

tyksessä oli perinteisesti kirjoitettu runsaasti dokumentointia ja ehdotus ohjelmistosta täysin ilman dokumentointia olisi ollut mahdotonta perustella. Grenning teki jälleen kompromissin ja ehdotti suppeamman dokumentoinnin tuottamista ylläpitoa varten.

Perusteluissaan Grenning [Gre01] käytti hyväkseen XP:n käytäntöjen toisiaan tukevia ominaisuuksia, kuten esimerkiksi katselmointien puuttumista pariohjelmoinnilla, pariohjelmointia koodin yhteisomistuksella ja niin edelleen. Lopulta Grenning onnistuikin saamaan projektista vastaavan johtajan XP:n taakse ja kehitystyö saattoi alkaa. Ohjelmistossa oli 125 käyttötapausta. Aivan työn alussa kehitysryhmä ei uskaltanut tehdä ohjelmistoa täysin ilman suunnittelua, vaan se teki joitakin luokka- ja sekvenssikaavioita.

Koska kehitysryhmän jäsenet tiesivät, että suunnitelmat tulisivat todennäköisesti muuttumaan, kaavioita ei dokumentoitu sähköisesti. Ohjelmoitaessa käytettiin pariohjelmointia ja tässä vaiheessa voitiin huomata, ettei koodin yhteisomistajuutta tarvittu, koska kehitysryhmän jäsenten ohjelmointityyli oli niin samankaltainen. Kehitysryhmällä oli ohjelmiston suhteen myös suorituskykyvaatimuksia ja suunnittelumallien avulla onnistuttiin tuottamaan skaalautuva ohjelmisto. Lopputuloksena oli, että projekti onnistui ja saatuun tulokseen oltiin tyytyväisiä. Projektin alku kuitenkin osoitti, miten vaikeaa on XP:n tuominen sellaiseen organisaatioon, jossa on pitkään käytetty perinteisiä menetelmiä. [Gre01]

6.6 Tutkimustuloksia XP:stä ja pariohjelmoinnista

Tutkielman tässä kohdassa käydään läpi eri tieteellisistä kirjastoista koottua valmista tutkimusmateriaalia. Materiaali käsittelee XP:n ja pariohjelmoinnin soveltamisesta saatuja kokemuksia, tutkimustuloksia ja käytöstä tehtyjä havaintoja. Eri tutkimuksia kerättiin yhteensä 40 kappaletta ja kaikkiaan materiaalia kertyi yhteensä noin 400 sivua. Koottu tutkimusmateriaali käytiin kokonaan lävitse keväällä 2008. Osa materiaalista jätettiin käyttämättä tässä yhteydessä tutkimusten päällekkäisyyksien takia ja/tai koska ne olivat aihealueiltaan tällä kertaa ja tähän tarkoitukseen käyttökelvottomia.

Nyt esiteltävät tutkimukset vaihtelevat laajuudeltaan ja ne on suoritettu eri puolilla maailmaa erilaisissa ympäristöissä. Tutkimuksista käydään lyhyesti läpi niiden käsittelemät aiheet ja niistä saadut keskeisimmät tutkimustulokset. XP:tä ja pariohjelmointia käsittelevä

materiaali on pyritty käymään lävitse aihealueittain ja se on kerätty yliopistoympäristöstä ja yritysmaailmasta saaduista kokemuksista ja tutkimustuloksista. XP:n käytännöistä ainoastaan pariohjelmointia on tutkittu jossakin laajuudessa ja tässä yhteydessä käsiteltävät pariohjelmointia käsittelevät tutkimukset on tehty pääosin yliopistoympäristössä.

6.6.1 Tutkimustuloksia XP:n käytöstä saaduista kokemuksista

Bunsen [Bun04] tutkimuksessa käsitellään Kaiserslauternin yliopistossa XP:n käyttöönotosta saatuja kokemuksia. Tutkimuksen tavoitteena oli opettaa oppilaat kehittämään ohjelmistoja ketterillä menetelmillä ja erityisesti XP:tä käyttäen. Toisena tavoitteena oli tutkia voidaanko XP:tä tuntemattomille opettaa käyttämään sitä ja mitä vaikutuksia sillä on näin syntyvään järjestelmään. Lisäksi oppilaiden oli kyettävä vertaamaan XP:llä saatavia tuloksia niihin tuloksiin, joita saadaan perinteisimmillä prosessimalleilla.

Bunsen [Bun04] tutkimuksen mukaan ketterien menetelmien eräinä ongelmina ovat ohjauksen puute ja se seikka, että ketterien menetelmien onnistuminen riippuu siitä, kuinka kurinalaisia ja kokeneita kehittäjät ovat. Vaikka yliopistoympäristössä opiskelevat oppisivatkin nopeasti XP:n käyttämiä työtapoja ja käytäntöjä, on niiden soveltaminen todellisessa ympäristössä vaikeampaa. Tutkimuksessa ilmeni, että opiskelijat pitivät XP:n käyttämisestä, miten sitä käyttäen voitiin kehitettävä järjestelmä toteuttaa, sekä XP:n tuomasta vapaudesta ja vastuusta kehitystyön aikana. Kaikki projektiin osallistuneet olivat kuitenkin sitä mieltä, että tulevissa projekteissa pitäisi kiinnittää enemmän huomiota asiakkaan vaatimiin ominaisuuksiin, kuten käytettävyyteen ja toiminnallisuuksiin. Toinen huomio oli, että ilman teknologia-asiantuntijaa projekti saattaa epäonnistua. Tutkimuksen loppupäätelmänä esitettiin, että vaikka ketterät menetelmät ovatkin helppoja oppia ja tuottavat nopeasti tuloksia, ne eivät ole paras vaihtoehto aloittaa ohjelmistokehitys mikäli tavoitteena on korkealuokkainen ohjelmistotuote.

Myös McKinneyn [McK04] tutkimuksessa selvitetään mitä etuja XP:n käytäntöjen käyttöönotolla saadaan ohjelmointikurssilla. Tutkimus toi esille etuja, mitä XP:n käytäntöjä käyttämällä saadaan. Näitä olivat esimerkiksi pariohjelmointia ja muita XP:n käytäntöjä käyttämällä saatu parempi itseluottamus, pariohjelmoinnin mukanaan tuoma positiivinen

paine, yhteenkuuluvaisuuden tunne, työetiikka, parempi itseoppivaisuus ja yhteistyöstä oppiminen. Muita kvantitiivisen analyysin esille tuomia seikkoja olivat, että XP:n käytännöt kannustivat kehittämään muita ammatillisia taitoja, kuten kommunikaatiota, sitoutumista, yhteistyökykyä ja mukautuvaisuutta uusiin tilanteisiin.

Lopulta myös Sherrellin ja Robertsonin [She06] tutkimuksen tarkoituksena oli tutkia ja kerätä aineistoa, mitä vaikutuksia XP:n käyttöönotolla oli yliopistoympäristössä tapahtuvassa ohjelmoinnin opiskelussa ja opetuksessa. Sherrellin ja Robertsonin tutkimuksessa molemmat tutkijat tulivat siihen johtopäätökseen, että XP:tä ja sen käytäntöjä ja periaatteita soveltamalla päästiin parempiin tuloksiin kuin perinteisillä ohjelmistotuotannon menetelmillä. Lisäksi työryhmän moraali oli korkeammalla tasolla, mikä johtui tutkimuksen mukaan siitä, että toimivaa ohjelmistotuotetta päästiin tekemään jo projektin alusta alkaen.

Merkittävinä XP:n käytön etuina mainittiin esimerkiksi merkittävä tiedonjakamisen lisäys kehitysryhmän jäsenten välillä, vähentynyt dokumentaation ja syntyneiden virheiden määrä, ohjelmiston parantunut rakenne ja se seikka, että henkilöstö tuntui nauttivan itse ohjelmoinnista huomattavasti aiempaa enemmän. XP:ssä tehtävä vähäinen dokumentaatio aiheutti sen, että opiskelijat tuottivat enemmän ohjelmakoodirivejä kuin muita prosessimalleja käyttämällä. Lisäksi pidettiin hyödyllisenä XP:n testaukselle asetettuja vaatimuksia, koska tällöin opiskelijat joutuivat suunnittelemaan ja toteuttamaan testitapauksia koko kehitystyön aikana. Tässä yhteydessä saadut tutkimustulokset näyttävän tukevan sitä ajatusta, että ketterät menetelmät auttavat opiskelijoita siten, että he luottavat enemmän omiin taitoihinsa ohjelmistoprojektien aikana ja että lopuksi valmistunut ohjelmistotuote on aiempaa parempi. [She06]

Misicin [Mis06] tutkimuksessa haluttiin tietää käyttäjien havaintoja ja mielipiteitä XP:stä ja sen käytöstä, periaatteista ja käytännöistä. Tutkimus tehtiin Kanadassa Manitoban yliopistossa ja siinä käsiteltiin XP:llä toteutettua pilottihanketta ja tarkasteltiin projektista saatuja kokemuksia ja tuloksia. Tutkimukseen osallistui yhteensä 85 henkilöä, jotka olivat toimineet ohjelmistoteollisuudessa valtaosin yli 5 vuotta. Teknisestä näkökohdasta katsoen XP:tä ja sen käytäntöjä, sääntöjä ja ohjeita oli tarkasteltava niin yksityiskohtaisesti kuin on mahdollista. Sosiaalisesta ja johdon kannalta oli tarpeellista tutkia XP:n ja muiden mene-

telmien vaikutuksia, jotta voitaisiin paremmin tuntea niiden vaikutuksia käytännössä. XP:n käyttö ja hylkääminen riippuvat pitkälti siitä, minkälaisia mielipiteitä sen käyttö aiheuttaa sitä käyttävissä ohjelmistokehittäjissä ja -organisaatioissa.

Tutkimukseen osallistuneet olivat sitä mieltä, että XP antaa tuloksia eikä sen käyttäminen edellytä käyttäjiltä mitään erityistä. Osallistujat olivat myös sitä mieltä, että XP pitäisi ottaa käyttöön paikallisesti soveltaen ja etteivät kaikki sen käytännöt ole yhtä hyödyllisiä. Tutkimus toteaa, että ohjelmistonkehittäjät näyttävät pitävän XP:stä, koska se on perusteiltaan käytännön läheinen. Vastauksista kävi myös ilmi ettei XP:hen siirtymisen koettu olevan kiinni asiakkaan tai johdon hyväksymisestä. Kun tutkimuksessa kysyttiin eteen tulleista ongelmista koskien XP:n pääperiaatteiden käyttöä käytännössä, palautteen antaminen ja kommunikaatio koettiin muita helpommaksi toteuttaa. Yllättäen minkään käytännön ei koettu olevan liian vaikea käyttää. Yksinkertainen rakenne koettiin vaikeimmaksi ja sitä lähimmäksi todettiin pariohjelmointi, asiakastestit ja koodin yhteisomistus. [Mis06]

Melnikin ja Maurerin [Mel05] tutkimuksessa perehdyttiin ketteriin työmenetelmiin opiskelijoiden näkökulmasta, kuinka opiskelijat suhtautuivat ketteriin menetelmiin yleisesti, sekä yksittäisiin joustaviin työmenetelmiin. Tutkimuksessa käytettävät työmenetelmät oli otettu XP:stä. Kolmivuotisen tutkimuksen tulosten perusteella opiskelijat suhtautuvat hyvin myönteisesti ketterien menetelmien käyttämiin työtapoihin eikä suhtautumisessa ole eroja eri koulutusohjelmissa ja -tasoissa. Lähes kaikki hyväksyivät ja pitivät ketterien menetelmien tuomista mahdollisuuksista. Tämä siitäkin huolimatta, että XP:n ja sen menetelmien käyttöönotto katsottiin vaikeammaksi oppilaitosympäristössä kuin ohjelmistoteollisuudessa. Tutkimus esittää, että oppilaiden lähes poikkeuksetta positiivinen asennoituminen ketteriin menetelmiin ennakoi niiden merkittävästi nykyistä laajempaa käyttöä myöhemmin tulevaisuudessa ohjelmistoteollisuudessa ja että ketterillä tulee olemaan merkittävä vaikutus ohjelmistokehitykseen tulevaisuudessa.

LeJeunen [LeJ05] tutkimuksessa kokeiltiin XP:tä käytännön ohjelmistoprojektissa, jossa kahdeksantoista opiskelijaa oli jaettuna kolmeen kuuden opiskelijan ryhmään tehden samaa ohjelmistoprojektia kevätlukukauden ajan vuonna 2004. Tutkimuksen tavoitteena oli, että opiskelijat tunnistavat XP-mallin vahvuudet ja heikkoudet ja lopulta toteuttavat ohjel-

mistokehitysprojektin käyttäen määriteltyjä prosesseja ja käytäntöjä. Kaiken kaikkiaan opiskelijat onnistuivat projekteissaan XP:n käytössä. Opiskelijat pitivät pariohjelmoinnista, automaattisesta testauksesta ja testauslähtöisestä ohjelmistokehityksestä ja uskoivat niiden edesauttaneen onnistumisessa. XP:n käytännöistä yksinkertaista rakennetta ja refaktorointia ei saatu toimimaan tyydyttävällä tavalla.

XP:n tarjoamista käytännöistä osa osoittautui hyödyllisiksi, osa ei tarjonnut merkittävää etua. Ryhmän pieni koko XP:ssä koettiin hyödyksi muutostenhallinnan yhteydessä. Automatisoitu testaus koettiin eduksi, sillä se vähensi pelkoa iteraatioiden välillä tapahtuvaan muutokseen. Pariohjelmointi koettiin menestykseksi ja opiskelijat pitivät siitä ja erona aikaisempiin vastaaviin kursseihin heikommin ohjelmoivat eivät karttaneet osallistumista itse ohjelmointityöhön. Jatkuva testaus katsottiin myös menestykseksi ja opiskelijat omaksuivat jatkuvan testauksen vaatiman määrän ja laadun nopeasti. XP:n käytännöistä yksinkertainen rakenne osoittautui epäonnistuneeksi, sillä oppilaat eivät kyenneet näkemään työnsä ja valmiin ohjelmistotuotteen kokonaiskuvaa. [LeJ05]

Ryhmät eivät toimittaneet täydellisesti toteutettua ohjelmistotuotetta haluttuine ominaisuuksineen. Kuitenkin se koodi, mitä luovutettiin, oli korkealaatuista ja hyvin testattua ohjelmakoodia. XP:n katsottiin soveltuvan hyvin tilanteisiin, jossa järjestelmää ja sen ominaisuuksia on muutettava, eivätkä muutokset aiheuta häiriöitä tehtävään kehitystyöhön. Tutkimus ei pidä XP:tä ihanteellisena mallina käytettäväksi aloittelevien opiskelijoiden ohjelmointikurssilla. Tutkimuksessa todetaan, että XP:n käytännöt ottavat hyvin huomioon vaatimusten, rakenteen ja koodin muuttamisen, mutta ne eivät hallinnoi hyvin ohjelmistotuotteen päämääriä. Lopputuloksena tutkimuksessa todetaan, että XP on saattanut luoda hyvin tehokkaan lähestymistavan hyvin ohjelmoijakeskeisiin käytäntöihin. [LeJ05]

Dalcherin [Dal05] tekemän tutkimuksen tarkoituksena oli tutkia erilaisia lähestymistapoja ohjelmistojen tuotantoon ja niiden keskinäistä tehokkuutta. Tutkimus suoritettiin yliopistoympäristössä käsittäen yhteensä 55 kehittäjää. 15 ryhmää työskenteli samanlaisen sovelluksen parissa käyttäen vesiputousmallin tyyppistä V-mallia, inkrementaalista, evoluutiomallia sekä XP-mallia. Tutkimustulokset, jotka koostuivat menetelmien suhteellisista ansioista, kuten esimerkiksi attribuuteista, tuotteen ominaisuuksista, käytetystä ajasta ja ohjel-

mointituloksista, auttavat ymmärtämään miten tietyn menetelmän valitseminen vaikuttaa lopputuotteeseen eli ohjelmistoprojektista syntyvään ohjelmistoon.

Kullekin työryhmälle annettiin kehitettäväksi interaktiivinen ratkaisu, joka käytti hyväkseen tietokantaa ja oli tarkoitettu käytettäväksi www-käyttöliittymän avulla. Jokainen tuote oli muihin verrattuna hieman erilainen ja näin taattiin kaikille ryhmille samat lähtökohdat. Ohjelmistokehittäjät sijoitettiin täysin satunnaisesti eri ryhmiin, joista jokainen käsitti 3-4 kehittäjää. Jokainen ryhmä käytti samoin satunnaisesti valittua ohjelmistokehitystapaa ja samoin työn aiheet annettiin satunnaisesti. Tietoja projektien edistymisestä koottiin jatkuvasti ja saadut tulokset tarkistettiin ja vahvistettiin. [Dal05]

Saaduista tutkimustuloksista voidaan todeta, että Java-koodin osalta XP-ryhmät tuottivat keskimäärin 4836 koodiriviä ja muiden ryhmien tulosten ollessa välillä 1032–1803 koodiriviä. Samankaltaisia saatiin, kun verrattiin ryhmien saamia tuloksia XML-koodiriveillä. Tällöinkin XP-menetelmää käyttävät ryhmät saivat aikaan merkittävästi enemmän koodirivejä kuin muita menetelmiä käyttäneet. Tulokset antavat ymmärtää, että XP:llä on muita prosessimalleja korkeampi tuottavuus miten saadaan aikaan valmis ohjelmistotuote. Vastaavasti lähinnä vesiputousmallia vastaavassa V-mallissa kulutettiin paljon aikaa vaatimusten määrittelyyn ja suunnitteluaktiviteetteihin. [Dal05]

Tässä tutkimuksessa XP:ssä vaatimusten määrittely vei käytetystä ajasta 2,7 %, suunnittelu 4,4 %, ohjelmointi 31 %, integrointi ja testaus 9 %, tarkastelu 7 %, korjaus 10 % ja muut aktiviteetit 26 %. Huomattavaa on se, että V-mallia käyttäneet ryhmät käyttivät edellä mainittuihin muihin aktiviteetteihin keskimäärin kaksinkertaisen määrän aikaa. Tulokset näyttävät vahvistavan sen seikan, että XP vaatii vähemmän panostuksia projektin alkuvaiheissa, erityisesti vaatimusten määrittelyssä. Korjausten vaatimissa resursseissa XP kulutti enemmän kuin inkrementaalinen ja evoluutiomalli. [Dal05]

Vaatimusten ja suunnitelman osalta V- ja inkrementaalinen malli tuottivat huomattavan määrän sivuja, sanoja ja spesifikaatiorivejä. Evoluutiomalli oli tässä suhteessa lähes yhtä tuottelias. XP tuotti vähemmän kuin neljäsosan edellisten tuottamasta sivumäärästä, kuudesosan verran sanoista ja vain kahdeksasosan edellisten tuottamasta rivimäärästä koskien spesifikaatioita. Toisin sanoen XP tuottaa vähemmän sivuja koskien vaatimuksia ja vä-

hemmän sanoja, jotka kuvaavat näitä vaatimuksia. Tuotteen koon ja tuottavuuden osalta XP tuotti 3,5 kertaa enemmän riveinä koodia kuin V-malli, 2,7 kertaa enemmän kuin inkrementaalinen malli ja 2,2 kertaa enemmän kuin evoluutiomalli. [Dal05]

Kun mittayksikkönä oli koodirivejä kuukaudessa, XP:tä käyttäneet ryhmät saivat aikaan 4,8-kertaisesti sen määrän koodia kuin V-mallia käyttäneet ryhmät, 2,8-kertaa enemmän kuin inkrementaalista mallia käyttäneet ryhmät ja 2,3-kertaa enemmän kuin evoluutiomallia käyttäneet ryhmät. Kun tuloksia tarkasteltiin pelkästään tuotettujen koodirivien näkökulmasta, XP:tä käyttäen saatiin aikaiseksi 2262 koodiriviä kuukaudessa keskimääräisen aikaansaannoksen ollessa 1077. Ohjelmistoteollisuudessa keskimääräisen tuottavuuden koodiriveissä arvioidaan olevan 200–500 koodiriviä kuukaudessa. V-mallia lukuun ottamatta kaikki prosessimallit ylittivät tämän keskimääräisen arvon. [Dal05]

Tutkimuksen lopputulokseksi saatiin, että XP:tä prosessimallinaan käyttäneet olivat tehokkaimpia tuottaen rivimäärällä mitattuna eniten ohjelmakoodia ja suurimman määrän käyttöliittyminä toimineita näyttöjä. Puhuttaessa itse prosessista XP:tä käyttäneet käyttivät vähiten aikaa määrittelyihin tuottaen samalla sivumäärällisesti pienimmän määrän tuotoksia määrittelyistä. Samoin kävi erilaisten diagrammien määrässä. XP pyrkii maksimoimaan itse ohjelmointiin käytettävän ajan esimerkiksi minimoimalla tehtävän dokumentaation määrää. Tällä tavalla pyritään siihen, että saadaan aikaiseksi enemmän ja nopeammin käytännön tuloksia, kuten ohjelmakoodirivejä ja käyttöliittymänä toimivia näyttöjä. [Dal05]

Laymanin ja kumppaneiden [Lay05] tutkimuksessa etsittiin vastausta siihen, lisääkö XP:n käyttöönotto tuottavuutta ja vähenevätkö virheet ohjelmakoodissa. Tutkimus keskittyi Sabre Airline Solutionin ketterän ohjelmistokehitysryhmän työskentelyyn ja tuloksiin. Ryhmä koostui 15:sta ohjelmistokehittäjästä, yhdestä testaajasta ja useasta vain tiettyyn osa-alueeseen keskittyneestä henkilöstä. Tutkimus tehtiin yhteistyössä Pohjois-Carolinan yliopiston ja Sabre Airline Solutionin kesken.

Ennen julkaisua ilmenevä virhetiheys ja virheiden poistamisen tehokkuus ovat molemmat tärkeitä indikaattoreita ilmaistaessa ohjelmiston laatua ennen sen julkaisemista. Molemmissa tapauksissa XP:n katsottiin parantavan ohjelmiston laatua. Ohjelmiston julkaisun jälkeistä aikaa ja ohjelmiston toimintaa sinä aikana seurattiin puolen vuoden ajan ja tänä

aikana ohjelmistossa ilmeni keskimääräistä vähemmän virheitä. Tutkimuksessa verrattiin liiketoimintaan vaikuttavia tuloksia, kun ohjelmistotuote kehitettiin käyttäen perinteisiä menetelmiä ja myöhemmin, kun jatkokehityksessä käytettiin XP-mallia ja sen työmenetelmiä. Tuloksena oli, että käytettäessä XP:tä saavutettiin 65 %:n vähennys ennen julkaisua löydetyissä virheissä ja 35 %:n vähennys julkaisun jälkeen havaituissa ohjelmistovirheissä. Lisäksi voitiin todeta 50 %:n parannus tuottavuudessa XP-julkaisujen osalta. [Lay05]

IBM [Wil04] toteutti tutkimuksen, jossa XP:n vaikutuksia tutkittiin pienessä 7–11 hengen ohjelmistokehitysryhmässä. Tässä yhteydessä suoritettiin kaksi ohjelmiston julkaisua, joiden tarkoituksena oli auttaa ryhmää siirtymisessä XP:n käytäntöihin ja stabilisoimaan sen käyttö. Tutkimuksen tuloksina saatiin, että ennen julkaisua löydetyt virheet vähenivät 50 % ja julkaisun jälkeen löydettyjen virheiden määrä väheni 40 %. Lisäksi voitiin yleisesti todeta tuottavuuden parantuneen. Vastaavasti VTT:n [Abr03] teettämässä tutkimuksessa selvittiin, mitä vaikutuksia XP:n prosessimallin käytöllä on ohjelmistotuotteen laatuun. Se suoritettiin kontrolloituna case-tutkimuksena neljän hengen ohjelmistokehitysryhmällä käyttämällä XP:tä. Projekti kesti kahdeksan viikkoa resurssien ja aikataulun ollessa kiinteitä. Vertailtaessa ensimmäistä ja toista ohjelmistonjulkaisua, voitiin todeta että suunnittelutarkkuus parani 26 %, tuottavuus parani 12 koodirivin verran tunnissa, mutta virheiden määrä pysyi vakiona.

Rumpe ja Schröder [Rum02] käyttivät kyselytutkimusta kootakseen tilastotietoa useista XP-projekteista. Sen tarkoituksena oli selventää XP:n käyttöä todellisessa ohjelmistoteollisuuden ympäristössä. Kesällä 2001 XP:tä käyttäviltä projektipäälliköiltä ja kehittäjiltä pyydettiin vastauksia kyselyyn useiden eri kanavien kautta. Tutkimus oli maailmanlaajuinen ja se toteutettiin lähinnä Internetin välityksellä. Tutkimuksen kysymykset jakautuivat kolmeen osioon, joista ensimmäisen osion kysymykset käsittelivät XP:tä käyttävää organisaatiota ja itse ohjelmistoprojektia. 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 yhteensä 45 kappaletta, lähes kaikki projektipäälliköiltä ja ohjelmistokehittäjiltä.

Noin puolet kohteena olevista projekteista oli kyselyn aikana vielä kesken. Tutkimuksen yllättävin tulos oli, että 45 projektista 44 oli onnistuneita, yksi osittain onnistunut, eikä epäonnistuneita ei ollut lainkaan. Verrattaessa tätä tulosta aiemmin esitettyihin Standish Groupin tutkimustuloksiin, ovat erot todella suuria. Tähän saattaa olla olemassa kolme eri syytä: [Rum02]

1. XP on avain menestykseen.

2. Kehittäjillä on tapana arvostaa työnsä tulokset korkeammalle kuin asiakkailla, eikä tässä yhteydessä asiaa ja/tai mielipiteitä tiedusteltu lainkaan asiakkailta.

3. Voidaan epäillä, että kyselyyn vastattiin herkemmin, mikäli XP-projektissa oltiin onnistuttu. Tutkimuksen mukaan uuden teknologian aikaiset käyttäjät ovat aina korkeasti motivoituneita ja tämä seikka voi vääristää tutkimustuloksia. XP:n käytön eräänä motivaationa oli henkilökohtainen kiinnostuneisuus XP:tä kohtaan, minkä takia tämä seikka osaltaan selittää tässä tutkimuksessa saatuja tuloksia.

Noin 70 % tutkimuksessa mukana olleista projekteista oli toteutettu Java-kielellä. Tätä seikkaa voidaan tulkita siten, että uusien teknologioiden käyttäjät kokeilevat mielellään myös uusia menetelmiä. Noin 60 % kyselyyn vastanneista oli Euroopasta, vaikka osuus maailman ohjelmistoliiketoiminnasta ei ole vastaavaa luokkaa. Mielenkiintoista oli myös se, että henkilökohtaisesti kyselyyn pyydetyistä XP:n käyttäjistä peräti 78 % ei osallistunut kyselyyn, sillä he eivät olisi saaneet virallisesti käyttää XP:tä projekteissaan, eivätkä halunneet XP:n käytön tulevan ilmi. Tästä päätellen ohjelmistoja tekevissä organisaatioissa on voimakasta muutosvastarintaa. Kyselytutkimuksen tuloksia voidaan pitää kuitenkin vain suuntaa antavina, sillä sen otos oli pieni ja osa projekteista oli kesken, joten niiden osalta lopullisen onnistumisen arviointi oli vielä liian aikaista. [Rum02]

Tutkimuksen mukaan kaikki kyselyyn vastanneet olisivat valmiita sopivan tilaisuuden tullen käyttämään XP:tä myöhemmissäkin ohjelmistoprojekteissa. XP:n käyttö ja käyttöönotto kohtaavat ongelmia johdon suhteen, sillä se on usein skeptinen tai yhtiön filosofia ei esimerkiksi salli läsnä olevaa asiakasta. Hyödyllisimmäksi XP:n käytännöistä koettiin koo-

din yhteisomistus, testauslähtöinen kehitys sekä jatkuva integrointi. XP:n tärkeimpinä menestystekijöinä mainittiin testaus, pariohjelmointi ja se, että XP:ssä keskitytään olennaisiin päämääriin. Tutkimuksen tuloksena oli, että XP tarjoaa mielenkiintoisen vaihtoehdon ohjelmistokehitykseen. XP:n menetelmiä ja käytäntöjä voidaan muokata perinteisiin menetelmiin sopiviksi ja ottaa käyttöön uusissa yhteyksissä kuten esimerkiksi suurissa ja hyvin rakenteellisissa telekommunikaatiojärjestelmissä ja sulautetuissa tietokonejärjestelmissä, kuten myös laajoissa ohjelmistoprojekteissa. [Rum02]

Hutchesonin [Hut03] tutkimuksessa käytiin läpi, kuinka XP:tä sovellettiin eräässä miljoonien dollarien WWW-ohjelmistoprojektissa. Tässä yhteydessä tutkitaan XP:n käytöstä saatuja kokemuksia Fiduciary Trust-yrityksen erään laajan Internet-sovelluksen luonnissa ja tapauksesta saatuja kokemuksia. Ohjelmistoprojekti tarjosi hyvän tilaisuuden tutkia tapausta, jossa ei suoriteta erillistä testausta ja tästä aiheutuneita tapahtumia. XP:ssä erillisiä testaajia ei pidetä tarpeellisina eikä niiden katsota tuovan ohjelmistotuotteelle juuri lainkaan lisäarvoa.

Ohjelmistoprojekti toteutti 401(k)-nimisen WWW-projektin, joka hallinnoi mikroyritysten eläkerahastosuunnitelmia. Tätä ennen vastaavanlaisia palveluita oli ollut tarjolla vain keskisuurille ja tätä suuremmille yrityksille. Tämä seikka vaikeutti merkittävästi eläketietojen siirtymistä eteenpäin, mikäli henkilö siirtyi suuryrityksen palveluksesta mikroyritykseen. Koska mikroyrityksille ei ollut olemassa tällaista palvelua, eläkerahojen takaisinlainaus oli mahdotonta ja se oli suoritettava IRA:n eli Yhdysvaltain veroviranomaisen kautta. Tämä uusi ohjelmistotuote mahdollisti mikroyrityksille hallinnoida omia eläkerahastojaan ja oli siten valmistuttuaan erityisen hyödyllinen ja tarpeellinen. [Hut03]

Tutkimuksen tekijä piti onnistuneen projektin heikkoina puolina sitä, että se antoi johdolle väärän viestin, että ilman kunnollista testausta onnistunut projekti voi onnistua keneltä tahansa. Lisäksi johto kuvitteli, että testausta suoritettiin vain osalle sovellusta, vaikka XP on nimenomaan testauslähtöinen malli. Ilman jatkuvaa testausta yritys olisi huomannut virheet ennemmin tai myöhemmin ja niiden täydessä laajuudessa. Tutkimuksessa esitetään myös muita syitä miksi XP:tä käyttäviä ohjelmistoprojekteja voidaan luonnehtia menestyksiksi. Eräs syy oli, että Internet-ympäristössä juuri kukaan ei mittaa kuinka paljon liiketoimintaa

menetetään, jos ohjelmisto on osittain tai kokonaan epäonnistunut. Näin mikä tahansa onnistuminen lisää tulosta ja vain muutama epäonnistuminen on otettu mukaan. [Hut03]

Hutchesonin [Hut03] tutkimuksen mukaan eräs yhteinen tekijä onnistuneille XP-projekteille on se, että projektien valmistuessa suuri osa kehitysryhmästä on valmiina korjaamaan havaittuja virheitä, ennen kuin liian moni asiakas on nähnyt valmiin lopputuotteen. Vastaavasti perinteisissä suunnitelmaohjautuvissa malleissa koodin kirjoittaneet kehittäjät yleensä ohjataan muihin projekteihin heti, kun kehittäjien tekemä koodi on saatu valmiiksi. Tällöin ei jää jäljelle ketään sellaista henkilöä korjaamaan ohjelmassa ilmeneviä virheitä, jolle valmistunut koodi on tuttua. Tämä seikka aiheuttaa ongelmia ja kustannuksia suurissa integrointiprojekteissa, joissa ongelmia ei löydetä ennen koodin integroimista.

Vaikka osa ohjelmistokehittäjistä pitääkin erillisiä testaajia turhina XP:ssä, osa on kuitenkin varovaisempia. 401(k)-projektin johtavan kehittäjän mielestä Internet-ympäristöön tehtävä sovellus tarvitsee testaushenkilöstöä, koska tähän ympäristöön oli jo vuosia asennettu testaamattomia ohjelmistoja ja tämä oli ongelmallista. Tässä yhteydessä kävi kuitenkin selväksi se, että ammattimainen testaushenkilöstö on tarpeellista myös XP-mallissa. Suunnitteluympäristön, jonka määrittelevät asiakkaan odotukset ja kehittäjän virtuositeetti, täytyy olla ohjattavissa niin, ettei itse päämäärä pääse karkaamaan. [Hut03]

Harrisonin [Har03] tutkimuksessa käydään läpi amerikkalaisen Avaya-nimisen suuren ohjelmistoyrityksen saamia kokemuksia XP:n soveltamisesta käytännössä kuudessa eri ohjelmistoprojektissa. Näistä saadut kokemukset toimivat suuntaa-antavina, kun halutaan miettiä XP:n ja muiden ketterien menetelmien soveltuvuutta suuriin ohjelmistoyrityksiin ja -projekteihin. Vaikka XP ei arvostakaan ohjelmiston arkkitehtuurisuunnittelua, nyt tarkasteltavissa projekteissa keskityttiin erityisesti ohjelmiston arkkitehtuuriin. Projekteissa saavutettiin automatisoidun testauksen osalta jonkin verran menestystä ja useimmissa käytettiin pariohjelmointia. Molemmista XP:n käytännöistä oli hyötyä erikokoisissa ohjelmistoprojekteissa.

Avayassa yksittäiset kehittäjät ja pienet ryhmät alkoivat tutkia itsenäisesti ja toisistaan tietämättä XP:n ominaisuuksia ja käytäntöjä ja soveltamaan niitä sitten käytäntöön ohjelmistoprojekteissa. Projektit olivat aluksia pieniä tai suurten projektien osia. Projekteihin osal-

listui parhaimmillaan 40 henkilöä, joissa kehittäjien määrä vaihteli kahdesta kahteenkymmeneen. Tässä yhteydessä projekteihin osallistuneista kehittäjistä haastateltiin viittätoista henkilöä. Tutkimuksen päättyessä projektit olivat keskeneräisiä ja yksi oli keskeytetty liiketoiminnallisista syistä. [Har03]

Koska valtaosa tutkituista projekteista oli suurempien projektien osia, ei tämän takia kaikkia XP:n käytäntöjä voitu ottaa käyttöön. Mikään ohjelmistoprojekti ei noudattanut käytäntöjä orjallisesti, vaan niitä noudatettiin tilanteen mukaan. Tutkimuksen mukaan ohjelmistoprojektien oleminen osana suurempia kokonaisuuksia vaikutti niistä saatujen tietojen hankintaan ja laatuun. Tutkimuksen mukaan projekteista saadut kokemukset antavat suuntaa siten, että jotkin XP:n käytännöistä voivat olla hyvinkin hyödyllisiä suurissa ohjelmistoalan yrityksissä, kunhan niitä sovelletaan yrityksen prosesseihin ja organisaatioon sopiviksi. [Har03]

Tutkimuksessa päädytään seuraavassa esitettäviin toimenpiteisiin kuinka XP:tä voidaan käyttää suurissa ohjelmistoalan yrityksissä. Ohjelmistoprojekteissa pitäisi aktiivisesti priorisoida vaatimuksia XP:n käyttämän suunnittelupelin tavoin. Tämä kuitenkin vaatii ottamaan huomioon asiakkaan ja markkinoinnin mielipiteitä. Tutkimus esittää, että parasta olisi luoda jonkinlainen toimiva ohjelmistoratkaisu, johon lisätään toiminnallisuutta niin paljon ja usein kuin on mahdollista. Tällöin kehitysryhmän aikataulu voidaan lyödä kuukausiksi eteenpäin, mutta samalla aikatauluun saadaan jonkin verran joustavuutta muutoksien varalta. [Har03]

Ohjelmiston arkkitehtuurisuunnittelua suositellaan jatkettavaksi, etenkin suurissa järjestelmissä. Edellä mainituissa ohjelmiston arkkitehtuuria pidetään tärkeinä, sillä se auttaa muodostamaan yhteisen jaetun näkemyksen järjestelmästä ja sen osista, auttaa suunnittelemaan siihen lisättäviä osia ja tarkistaa kykeneekö järjestelmän rakenne vastaamaan sille asetettuja vaatimuksia etenkin ei-toiminnallisten ominaisuuksien kuten luotettavuuden ja suorituskyvyn osalta. Lyhyet iteraatiot tulevat olemaan tärkeä osa laajoja ketterillä menetelmillä toteutettavia projekteja. Lisäksi niitä voidaan käyttää tilanteissa, joissa vain osa projektista soveltaa ketteriä työmenetelmiä. Suurissa ohjelmistoprojekteissa on usein mo-

nia sisäisiä riippuvaisuuksia ja tämä vaikuttaa eri iteraatioiden sisältöön, mutta ei niiden kestoon. [Har03]

XP:ssä käytössä oleva laaja yksikkötestaus on käyttökelpoinen myös suurissa ohjelmistoprojekteissa ja erityisesti automatisoitu testaus koetaan hyvin hyödylliseksi. Suurissa projekteissa päivittäinen regressiotestausmäärä saattaa kuitenkin kasvaa niin suureksi, että sitä on mahdotonta toteuttaa. Tässä tapauksessa on tarkoituksenmukaista testata vain oleelliset osat järjestelmästä. XP:ssä ei mainita mitään järjestelmätason testauksesta, mutta tätä tarvitaan kaikissa suurissa ohjelmistoprojekteissa. Lyhyet iteraatiot tarjoavat kuitenkin järjestelmätestaajille mahdollisuuden testata toimivaa ohjelmistoversiota jo aikaisessa vaiheessa ja tämä voi olla merkittävä etu testauksessa. [Har03]

Suuret ohjelmistoprojektit hyötyvät enemmän osittaisesta koodin yhteisomistuksesta kuin XP:ssä käytössä olevasta täydellistä koodin yhteisomistuksesta. Järjestelmän koon kasvaessa tulee mahdottomaksi, että jokainen kehittäjä hallitsisi kaiken järjestelmässä olevan ohjelmistokoodin. Yleisesti on mahdotonta, että suurissa ohjelmistoprojekteissa asiakas on koko ajan läsnä tai kun on kyseessä projekti, jolla on monta asiakasta. Tässä yhteydessä pidetään kuitenkin hyvänä, että joku edustaa asiakkaan etuja tai on yhteydessä asiakkaisiin. Käyttötapausten tekeminen voisi olla eduksi suurissa ketteriä menetelmiä käyttävissä projekteissa. Puhuttaessa jatkuvasta integroinnista, suurissa järjestelmissä sitä ei voida tehdä jatkuvasti, vaan pikemminkin tietyin ja tasaisin väliajoin. Muita XP:n käytäntöjä voidaan ottaa käyttöön, mikäli ne sopivat kunkin ohjelmistoprojektin toimintatapoihin ja -kulttuuriin. [Har03]

Avayassa tultiin siihen johtopäätökseen, että XP:stä oli paljon hyödyllistä. Mikään projekti ei noudattanut XP:tä orjallisesti ja tähän löydettiin kaksi pääsyytä. Ensinnäkin kaikki XP:n käytännöt eivät soveltuneet käytettäväksi suuriin projekteihin. Toiseksi jotkin käytännöt eivät toimineet yhdessä muiden käytäntöjen kanssa jo käynnissä olevissa projekteissa. [Har03]

Bahlin [Bah03] tutkimuksessa käsiteltiin sitä kuinka siirtyminen vesiputousmallista XP-malliin tapahtui käytännössä. Tutkimuksen kohteena oli kanadalainen ohjelmistoalan yritys, joka tuotti tietopalveluita terveydenhuollon piirissä. Yritys oli päättänyt muuttaa kehi-

tysmenetelmänsä perinteisestä vesiputousmallista XP:hen. Yritystä kutsutaan tässä yhteydessä kuvitteellisella ABC-nimellä. ABC kehitti kahta ohjelmistoprojektia, joista TrackMed toteutettiin käyttäen perinteistä vesiputousmallia ja InsightMed XP-menetelmää. Kummastakin projektista tutkittiin erityisesti neljää eri osa-aluetta eli tiedon luomista, käytettävyyttä, helppokäyttöisyyttä ja miten ohjelmistotuotetta aiottiin käyttää tulevaisuudessa. Osa-alueita tutkittiin erilaisin havainnoin ja haastatteluin 14 kehitysryhmän jäsenen kanssa kolmen vuoden ajan vuosien 1999 ja 2002 välisenä aikana.

TrackMed käynnistettiin vuonna 1999 ja sen tarkoituksena oli käsitellä 60 000 kanadalaisen lääkärin tarvitsemia lääketietoja. TrackMed toteutettiin käyttämällä perinteistä vesiputousmallia. Alkuvaiheessa kehitystyöhön arvioitiin menevän vuoden verran aikaa. Kun projektin aloittamisesta oli kulunut aikaa kaksi vuotta, oli selvää, ettei tuotteen julkaiseminen tullut kysymykseen. Kului vielä vuoden verran aikaa ennen kuin ensimmäiselle asiakkaalle voitiin antaa TrackMed-raportti. Sovellus itsessään oli laajentunut yhdestä neljäksi erilliseksi sovellukseksi. Lopulta TrackMed valmistui lähes kaksi vuotta myöhässä eli 60% yli suunnitellun ajan ylittäen samalla lähes 1,5 miljoonalla dollarilla eli lähes 50%:lla alkuperäisen budjettinsa. [Bah03]

Vesiputousmallin käytettävyyttä ja helppokäyttöisyyttä arvioitiin aluksi positiivisesti koko kehitysryhmän osalta. Kehitysryhmä oli tottunut käyttämään perinteistä vesiputousmallia pitäen sitä selkeänä ja suoraviivaisena mallina siten, että se antaa rakenteen kehitysprosessille mutta toisaalta myös estää vaatimusten muutoksia. Kehittäjät totesivat tekevänsä sen mitä liiketoiminta haluaa ja toivoo, mutta tällöin ongelmana on, ettei mikään tai mitään koskaan valmistu. Lopulta kehitysryhmän jäsenillä oli negatiivinen käsitys vesiputousmallin käytöstä eivätkä halunneet käyttää sitä enää myöhemmin. Vaikka vesiputousmallilla onkin pitkä historia, sitä on helppo ymmärtää ja käyttää niin tämä esimerkkitapaus osaltaan osoittaa, ettei sitä pidä käyttää kaiken tyyppisissä ohjelmistoprojekteissa. [Bah03]

InsightMed:n kehitystyö aloitettiin vuoden 2000 alussa, jolloin koko kehitysryhmä siirrettiin tähän projektiin TrackMed:n tekemisen sijasta. Tässä yhteydessä yrityksen johto päätti kokeilla XP:n käyttöä. InsightMed oli www-pohjainen sovellus, joka mahdollisti ABC:n asiakkaiden käyttää liiketoimintatietoa käsittäviä ohjelmistomoduuleita. Tietovaraston

käyttöliittymä oli uusi vaatien uuden moduulin ulkopuolisen tiedon latausta, muuttamista ja selausta varten. Osa tiedosta oli peräisin aiemmin mainitusta TrackMed-sovelluksesta. [Bah03]

XP:tä käyttämällä rakennusvaihe käsitti kolme iteraatiota sisältäen melko suuria arkkitehtuurimuutoksia toisen ja kolmannen iteraation välissä. Niiden aikana ohjelmasta tehtiin vähintään kerran viikossa ajettava versio ja kolmannen iteraation loppuvaiheessa ajettava versio ohjelmasta olikin jo melko vakaa. Vaikka ohjelmiston laatu olikin melko korkea, ei kaikkia vaadittuja ominaisuuksia saatu valmiiksi ennen lopullisen iteraation loppua. Kaiken kaikkiaan tuote viivästyi vuoden verran ylittäen näin alkuperäisen aikataulun 50 %:lla prosentilla ja budjettinsa 25 %:lla. InsightMed:n ensijulkaisu tapahtui vuoden 2001 puolivälissä. [Bah03]

Kun työ InsightMed-sovelluksen parissa alkoi, oli XP:n käyttämisestä olemassa tässä yhteydessä yhteisymmärrys. Vaikka ohjelmistoprojekti toimikin hyvin vaatimusten määrittelyn osalta, siinä ei noudatettu orjallisesti XP:n käytäntöjä. Projektissa kulutettiin muutamia kuukausia siihen, että päästiin asiakkaan kanssa yhteisymmärrykseen siitä mitä asiakas halusi ja mitä kehitysryhmä kykenisi toteuttamaan. Projektin alussa projektiryhmä tarvitsi viikon verran XP:n käytäntöjen harjoittelemista ja ulkopuolista apua ennen kuin InsightMed voitiin aloittaa, sillä XP oli kehitysryhmälle ennestään vieras tapa kehittää ohjelmistoja. XP:n työtavat koettiin kuitenkin hyödyllisiksi ja helpoiksi oppia. [Bah03]

Tämän tutkimuksen mukaan XP:llä on omat heikkoutensa. Tarve suunnitella paljon saattaa aiheuttaa ongelmia projektipäällikölle. Jokainen iteraatio loppuu tutkimukseen mitä seuraava iteraatio pitää sisällään. Projektin elinkaaren aikana joka hetki pidetään silmällä kolmea suunnitelmaa, eli ensinnäkin nykyistä iteraatiota, toiseksi seuraavaa iteraatiota ja kolmanneksi koko projektin kokonaissuunnitelmaa. Toinen riski tulee siitä, että on vaikeata raportoida johdolle projektin edistymisestä, sillä iteraatiot ovat luonteeltaan toistuvia. [Bah03]

Shukla ja Williams [Shu02] käyttivät Vuonna 2001 Pohjois-Carolinan valtion yliopistossa ohjelmoinnin opetuksessa sekä perinteisiä menetelmiä että XP:tä. XP-mallia käyttäneet oppilaat pitivät sitä hyödyllisenä ja kertoivat oppineensa paljon sen avulla ketterien mene-

telmien työtavoista ja periaatteista. Loppupäätelmänä kuitenkin oli, että tutkimuksen tekijät suosittelivat hybridiprosessimallia, joka sisältää sekä ketteriä että perinteisiä työmenetelmiä ja -käytäntöjä.

Lopulta tutkimuksessa esitetään, että ohjelmistokehitys siirtyy tarkasti määritellyistä suurista järjestelmistä kohti järjestelmiä, joiden kehityssyklit mitataan kuukausissa ja joiden ympäristö muuttuu koko ajan. Tämä ohjelmistokehityksen muuttuva ympäristö pakottaa ohjelmistoteollisuuden etsimään uusia tapoja kehittää ohjelmistoja nopeasti, toistettavasti, ennustettavasti ja tehdä tämä kaikki niin, että ohjelmistotuotteen laatu pysyy korkeana. Tutkimuksen mukaan nämä ovat edesauttaneet ketterien menetelmien käyttöä ja käytön lisäystä ohjelmistoteollisuudessa. [Shu02]

6.6.2 Tutkimustuloksia pariohjelmoinnin käytöstä saaduista kokemuksista

Kahden tai useamman hengen ryhmissä työskentely on kauan tunnettu käytäntö lähes kaikissa teknistä suunnittelua vaativissa ammateissa. Ohjelmistoteollisuudessa kehittäjät tekevät yhteistyötä toistensa kanssa suurimman osan päivästä. Erään tiedon mukaan ohjelmistokehittäjät käyttävät 30 % ajasta työskennellen yksin, 50 % työskennellen jonkun toisen kanssa ja 20 % kahden tai useamman henkilön kanssa. Pariohjelmointi on eräs XP:n tärkeimmistä käytännöistä ja se otettiin osaksi XP:tä, koska sen väitettiin kasvattavan käyttäjiensä tuottavuutta ja työskentelytyytyväisyyttä parantaen samalla kommunikointia ja ohjelmiston laatua. XP:n käytännöistä pariohjelmointia on tutkittu muita käytäntöjä laajemmin.

Swinburnen [Kee04] teknillisen korkeakoulun XP-projektiryhmä sovelsi pariohjelmoinnin käytäntöä menestyksekkäästi. Ohjelmakoodin laatu oli parempaa, ohjelmointi oli tehokkaampaa, kommunikointi opiskelijoiden välillä lisääntyi, ongelmat kyettiin ratkaisemaan helpommin ja ohjelmointi oli miellyttävämpää. Kurssin viimeisen kuukauden aikana yksi opiskelija ei voinut osallistua enää työskentelyyn, jolloin pariohjelmoinnista jouduttiin luopumaan. Tällöin ero pariohjelmoinnin ja yksin tehtävän ohjelmoinnin välillä tuli selkeästi näkyviin. Lähes kaikki pariohjelmoinnista seuranneet hyvät asiat onnistuivat heikommin yksin ohjelmoidessa.

Williamsin [Wil00] tutkimuksen tuloksena oli se, että pariohjelmointi tuotti 10–30 prosenttia vähemmän koodirivejä, vaikka kirjoitetut ohjelmat toimivatkin samalla tavalla. Näiden tutkimusten harjoitustehtävät olivat kuitenkin pieniä, vain muutaman sadan koodirivin 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 kuitenkin valmistuneet 40–50 prosenttia nopeammin yksin ohjelmoitaviin projekteihin verrattuna. Kaikkein radikaaleimmat tutkimustulokset 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 [Jen03].

Pariohjelmointi on osoittautunut myös Nosekin [Nos98] mukaan tehokkaaksi tavaksi ohjelmoida. Nosek järjesti testin, jossa toinen luokka opiskelijoita ohjelmoi pareittain, ja kontrolliryhmä yksinään. Pariohjelmoineet saivat tehtävän valmiiksi keskimäärin 30 minuutissa, kun kontrolliryhmällä siihen meni 43 minuuttia. Nosek [Nos98] nostaa esille kaksi syytä, miksi käyttää pariohjelmointia. Ensinnäkin huonosti kirjoitettu koodi aiheuttaa enemmän virheitä järjestelmän elinkaaren aikana kuin hieman useamman työtunnin käyttäminen laadukkaan koodin kirjoittamiseen pareittain. Järjestelmän kehitystä voidaan siis nopeuttaa pariohjelmoinnilla. Toiseksi ohjelmiston laatu paranee, kun algoritmi viestitään parille ennen sen kääntämistä ohjelmakoodiksi, jonka jälkeen pari validoi kirjoitettavan ohjelmakoodin.

McDowellin [McD02] tutkimuksessa tarkasteltiin pariohjelmoinnin vaikutuksia opiskelijoiden saamiin tuloksiin ohjelmointikurssilla. Tutkimus suoritettiin Kalifornian yliopistossa ja sen käyttämät tiedot kerättiin 600 opiskelijan joukosta, jotka suorittivat kurssille asetetut ohjelmointitehtävät joko pareissa tai työskennellen yksinään. Pariohjelmointia käyttäneet opiskelijat tuottivat parempia ohjelmia, selvisivät loppukokeesta vähintäänkin yhtä hyvin kuin yksin työskennelleet opiskelijat ja täten suorittivat kurssin paremmin arvosanoin. Tässäkin tutkimuksessa esitetään loppupäätelmänä, että pariohjelmointi tuottaa parempia ohjelmia samalla kun se parantaa käyttäjiensä opiskelusuorituksia.

Myös Cockburnin ja Williamsin [Coc00] tutkimus käsitteli sitä, onko pariohjelmointi yksintyöskentelyä tehokkaampaa, pariohjelmoinnin kustannustehokkuutta, tyytyväisyyttä siihen ja sen käytön seurauksena syntyneen tuotteen laatua. Tutkimus suoritettiin vuonna 1999 Salt Lake Cityn yliopistossa. Erääksi tutkimustulokseksi saatiin, että pariohjelmointiprojektin suorissa palkkakustannuksissa arvioitiin 15 %:n kasvu yksittäisiin ohjelmoijiin verrattuna. Tutkimuksessa kuitenkin korostettiin, että ohjelmiston laatu on lisääntynyt merkittävästi, ihmisten työtyytyväisyys on kasvanut ja ohjelmointitieto levinnyt. Pariohjelmointina tehdyt ohjelmat myös läpäisivät suuremman osuuden testitapauksista kuin yksin ohjelmoidut.

Pariohjelmoinnin kustannuksia laskettaessa tutkimuksessa kehotetaan laskemaan ohjelmistoprojektin kokonaiskustannukset, jotta pariohjelmoinnin kokonaistaloudellisuus saataisiin selville. Vähentyneen virheiden määrän myötä testaamisen ja tuotteen annetun tuen kustannukset vähentyvät, joten pariohjelmointi voidaan nähdä myös tällä perusteella yksinohjelmointia kustannustehokkaammaksi. Tutkimuksessa kävi myös ilmi, että työparit työskentelivät yli kaksinkertaisella nopeudella, työn tuloksena syntynyt ohjelmistokoodi oli suunnittelultaan parempaa, yksinkertaisempaa ja helpompaa laajentaa myöhemmin. [Coc00]

Cockburnin ja Williamsin [Coc00] tekemä tutkimus esittää loppupäätelminään pariohjelmoinnin eduiksi, että ohjelmakoodissa olevia virheitä kyetään löytämään nopeammin, koska koodia tarkastelee yhden sijaan koko ajan kaksi ja ohjelman valmistuttua siitä löydetään vähemmän virheitä jatkuvan tarkastelun vuoksi. Koska pariohjelmoinnissa kommunikointi on jatkuvaa, tuottaa se lyhyempää ja paremman rakenteen omaavaa ohjelmistokoodia, lisäksi työparit ratkaisevat ongelmat nopeammin kuin yksittäiset ohjelmoijat. Pariohjelmointia käyttävät henkilöt oppivat enemmän itse kehitettävästä järjestelmästä ja ohjelmistoprojektin valmistuttua useampi henkilö on oppinut perin juurin kehitetyn järjestelmän ominaisuudet.

Abrahamssonin ja Hulkon [Abr05] tutkimuksessa haluttiin koota pariohjelmointia koskevien tutkimusten tuloksia, kuinka sitä käytetään, kuinka se käytettäessä vaikuttaa ohjelmiston laatuun ja kuinka tieto pariohjelmoinnista on edennyt. Eräs tutkituimpia pariohjelmoinnin

ominaisuuksia on ollut tuottavuuden ja kulujen lisäyksen suhde, koska voidaan olettaa että kehittäjien määrän kasvattaminen pariohjelmointia varten kasvattaa samassa suhteessa myös kustannuksia. Tälläkin kertaa tutkimustulokset viittaavat siihen, että pariohjelmointi on yksin työskentelyä tehokkaampaa. Pariohjelmointi vaatii kuitenkin enemmän panostuksia kuin yksintyöskentely, mutta kulujen kasvua ei tapahdu samassa suhteessa. Siirtymävaiheessa kulujen kasvu on nopeampaa, mutta tuottavuuden kasvaessa pariohjelmoinnista tulee yksin työskentelyä tehokkaampaa.

Abrahamssonin ja Hulkon [Abr05] tutkimuksessa tehdään myös yhteenvetoa pariohjelmoinnista suoritetuista empiirisistä tutkimuksista. Seuraavassa esitettävä taulukko kokoaa yhteen pariohjelmoinnista esille tulleet väitteet. Vasemmanpuoleinen sarake esittää pariohjelmoinnista tehdyn väittämän ja oikeanpuoleinen toteaa väittämän todenmukaisuuden.

Pariohjelmoinnista tehty väittämä	**Todenmukaisuus**
Pariohjelmointi lyhentää kehitystyön aikaa	ei
Pariohjelmointi vaatii enemmän resursseja	ei
Pariohjelmointi tuottaa korkeatasoisempaa koodia kuin yksintyöskentely	kyllä
Pariohjelmointi on tuottavampaa kuin yksintyöskentely	kyllä
Pariohjelmointia käyttämällä tehdään vähemmän virheitä	kyllä
Kehittäjät pitävät pariohjelmoinnista enemmän	kyllä
Pariohjelmointi on hyödyllisempää monimutkaisissa tehtävissä	kyllä
Pariohjelmoinnista on hyötyä, kun sopeutetaan uutta henkilöä työtehtäviin	kyllä
Pariohjelmoinnin kustannushyöty (laatu vs. kustannukset) ei ole tarkasti tiedossa	kyllä

Taulukko 2: Yhteenveto pariohjelmoinnista tehdyistä empiirisistä tutkimuksista. [Abr05]

Abrahamssonin ja Hulkon [Abr05] tutkimus esittää, että pariohjelmoinnin tuottavuus verrattuna yksintyöskentelyyn riippuu suuresti kulloisestakin ohjelmistoprojektista. Tutkimus ei löydä merkkejä siitä, että jompikumpi työskentelytapa olisi ylivertaista toiseen verrattuna, mutta huomauttaa yksintyöskentelyä käyttäneiden tulosten vaihtelevan suuresti. Vastaavasti pariohjelmointia käyttäneet näyttävät saaneen samankaltaisia tuloksia tehokkuuden osalta tutkimuksesta toiseen. Myös tässä tutkimuksessa esitetään loppuyhteenvetona, että pariohjelmointi lisää itseluottamusta ja on tehokas keino uusien taitojen oppimiseen ja harjoittelemiseen. Eräänä pariohjelmoinnin ongelmana tutkimus tuo esille sen seikan, että kehittäjien voi olla vaikeata löytää aikaa työskennellä parina.

Padbergin ja Mullerin [Pad03] tutkimus tehtiin Karlsruhen yliopistossa ja sen tavoitteena oli tutkia, onko pariohjelmointia käyttämällä saatavissa potentiaalista etua ohjelmistoprojektin eri vaiheissa. Tutkimuksessa käytettiin projektimallia, jonka tarkoituksena oli osoittaa kuinka vahvasti riippuvaisia pariohjelmointia käyttävät projektit ovat työparien nopeudesta ja kyvystä löytää ohjelmakoodista virheitä. Tutkimuksessa painotettiin, että käytettäessä pariohjelmointia eri markkinatekijät aiheuttavat pariohjelmoinnin onnistumista kohtaan kovia paineita. Padbergin ja Mullerin [Pad03] tutkimus antoi selkeän vastauksen, koska pariohjelmointia voidaan käyttää. Mikäli ohjelmistoprojektin onnistumisen kannalta lyhyt aika on oleellisen tärkeä, kehittäjien määrän lisääminen pariohjelmoinnin toteuttamiseksi kannattaa toteuttaa huolimatta kasvavista kustannuksista. Edellä mainittu tulee kannattavaksi, koska ohjelmointipareilta voidaan odottaa korkeampaa tuottavuutta ja parempilaatuista ohjelmistokoodia kuin yksinään työskenteleviltä ohjelmoijilta.

Ciolkowskin ja Schlemmerin [Cio03] tutkimuksen tarkoituksena oli varmistaa muiden tutkimusten tutkimustuloksia työskentelemällä suuremmalla ryhmällä ja tehden suurempaa ohjelmointityötä. Tutkimus tehtiin Kaiserslauternin yliopistossa ja tarkkailtavana oli pariohjelmoinnin tuloksena syntyneen ohjelmakoodin laatu, ohjelmointityön täytäntöönpanoon vaadittava työ sekä opiskelijoiden subjektiivinen käsitys pariohjelmoinnista. Kaikki pariohjelmointia koskeva materiaali jaettiin kaikille ja työparit valittiin satunnaisesti. Pariohjelmointi oli kaikille ennestään vierasta ja tämä aiheutti lopullisissa tuloksissa heikennystä tehokkuuden suhteen. Toteutettavaa järjestelmää käytettiin graafisen käyttöliit-

tymän kautta, eikä sille voitu tehdä automaattisia testitapauksia, joilla tarkastella ohjelmistokoodin laatua.

Edellä esitettiin pariohjelmointia koskevien empiiristen tutkimusten väittämä, jonka mukaan pariohjelmointia käyttämällä saadaan korkeampilaatuista ohjelmistokoodia kuin tavallisesti. Tätä väittämää tukevia selkeitä todisteita ei saatu tästä tutkimuksesta. Myöskään sitä seikkaa, että pariohjelmointi vähentää koodirivien määrää 20 %:la ja että tämä indikoi parempaa koodin laatua, ei saatu tässä tutkimuksessa vahvistettua. Eräs syy siihen ettei näitä väittämiä voitu vahvistaa saattoi olla se, että ryhmät modifioivat jo olemassa olevaa järjestelmää, jolloin järjestelmä itsessään rajoitti ryhmien toimintamahdollisuuksia. [Cio03]

Eräs pariohjelmointia koskevista väittämistä on että pariohjelmointia käyttävät ryhmät tarvitsevat vain vähän enemmän panostuksia. Tälle väittämälle tutkimus löysi jonkin asteista tukea. Tässä yhteydessä pariohjelmointia käyttäneet tarvitsivat noin 10 % enemmän panostuksia kuin tavallisesti. Pariohjelmoinnin kannattajat väittävät myös sen auttavan käyttäjiä oppimaan uusia tekniikoita nopeammin, mutta tälle väitteelle ei löydetty todisteita. Syitä tähän voi olla esimerkiksi se, että pariohjelmointi ei auta uuden tekniikan omaksumisessa tai ettei sitä käytetty uuden tekniikan opettelemiseen. Loppupäätelminään tutkimus kertoo, että pariohjelmointia käyttäneet opiskelijat käyttäisivät mahdollisuuden tullen sitä uudelleen ja että pariohjelmointia käyttäen saatiin aikaan hyvin rakentava ilmapiiri. [Cio03]

6.6.3 Yhteenveto tutkimustuloksista

Yliopistoympäristöstä saadut tutkimustulokset osoittavat, että opiskelijat pitävät XP:n käyttämisestä, sen tuomasta vapaudesta ja vastuusta kehitystyön aikana ja miten sitä käyttäen kehitettävä järjestelmä voidaan toteuttaa. Tutkimukset esittävät, että oppilaiden positiivinen asennoituminen XP:hen ennakoi merkittävästi nykyistä laajempaa käyttöä tulevaisuudessa ohjelmistoteollisuudessa. Ketteriä menetelmiä ei kuitenkaan pidetä parhaana vaihtoehtona aloittaa ohjelmistokehitystyö, mikäli tavoitteena on korkealuokkainen ohjelmistotuote. Kokemattomien kehittäjien katsotaan tarvitsevan ohjausta saavuttaakseen ketterien menetelmien käytössä tarvittavaa kokemusta ja kurinalaisuutta. Tutkimustulokset

näyttävät tukevan myös sitä, että ketterät menetelmät auttavat opiskelijoita luottamaan enemmän omiin taitoihinsa.

Ohjelmistonkehittäjät pitivät XP:stä, koska se on perusteiltaan käytännön läheinen, antaa tuloksia eikä sen käyttäminen edellytä käyttäjiltä mitään erityistä. Kun kysyttiin eteen tulleista ongelmista koskien XP:n pääperiaatteiden käyttöä, palautteen antaminen ja kommunikaatio koettiin helpoimmiksi toteuttaa. Vastaavasti yksinkertainen rakenne, pariohjelmointi, asiakastestit ja koodin yhteisomistus koettiin vaikeimmiksi. Hyödyllisimmäksi XP:n käytännöistä koettiin koodin yhteisomistus, yksinkertainen rakenne, testauslähtöinen kehitys sekä jatkuva integrointi. XP:n tärkeimpinä menestystekijöinä mainittiin testaus, pariohjelmointi sekä se, että XP:ssä keskitytään olennaisiin päämääriin.

Jonkinlainen käsitys XP:tä käyttävien projektien tuloksista ja nykytilasta saatiin esille Rumpe ja Schröderin [Rum02] kyselytutkimuksen kautta. Tutkimuksen yllättävin tulos oli, että 45 projektista 44 onnistui. Tämän tutkimustuloksen luotettavuutta kuitenkin haittaa se, että kyselytutkimus lähetettiin noin 200:lle vain 22%:n vastatessa. Kuitenkin kun tätä verrataan tutkielmassa aiemmin esitettyihin Standish Groupin tutkimustuloksiin, voidaan huomata erojen olevan todella suuria. Tuloksen saattaa aiheuttaa se, että XP todellakin on avain menestykseen. Toisaalta tuloksiin saattaa vaikuttaa myös se, että kehittäjillä on tapana arvostaa työnsä tulokset korkeammalle kuin asiakkailla. XP-projektien korkea onnistumisaste kuitenkin kertoo sen, että XP:tä voidaan käyttää onnistuneesti ohjelmistoprojekteissa.

Eräänä yhteisenä tekijänä onnistuneille XP-projekteille pidetään sitä, että niiden valmistuessa suuri osa kehitysryhmästä on edelleen koossa ja valmiina korjaamaan havaittuja virheitä. Niinpä kehittäjät korjaavat ohjelmakoodia, jonka ovat itse tehneet ja tuntevat hyvin. XP ei kuitenkaan sisällä mitään tasapainottavia aktiviteetteja, joilla taataan toimiva ohjelmistotuote. XP ei myöskään sisällä mitään perustavanlaatuisia metriikoita.

Tutkimustulokset antavat ymmärtää, että XP:llä on muita korkeampi tuottavuus koskien sitä miten prosessimalli saa aikaan valmiin ohjelmistotuotteen. Tulokset vahvistavat myös sen seikan, että XP vaatii vähemmän panostuksia projektin alkuvaiheissa, erityisesti vaatimusten määrittelyssä. XP:n kyky ottaa asiakas mukaan ja säännölliset ohjelmistojulkaisut

näyttävät lisäävän ohjelmiston käyttötoiminnallisuutta ja tuottavan ohjelmistotuotteelle paremman hyväksyttävyyden ja nämä seikat tuovat lisäarvoa valmistuvalle ohjelmistotuotteelle. Verrattaessa liiketoimintaan vaikuttavia tuloksia, voitiin huomata, että käytettäessä XP:tä saavutettiin merkittäviä vähennyksiä ennen julkaisua löydetyissä virheissä ja julkaisun jälkeen havaituissa ohjelmistovirheissä.

XP:tä koskevien tutkimusten yhteenvedoksi voidaan vetää, että XP on saattanut luoda tehokkaan lähestymistavan hyvin ohjelmoijakeskeisiin käytäntöihin. Ohjelmistoprojekteista saadut kokemukset antavat suuntaa siten, että jotkin XP:n käytännöistä voivat olla hyvinkin hyödyllisiä suurissa ohjelmistoalan yrityksissä, kunhan niitä sovelletaan yrityksen organisaatioon ja prosesseihin sopiviksi. Tutkimuksissa esitetään, että XP:llä ja muilla ketterillä menetelmillä tulee olemaan merkittävä vaikutus ohjelmistokehitykseen tulevaisuudessa. XP:n menetelmiä ja käytäntöjä voidaan ottaa käyttöön uusissa yhteyksissä kuten esimerkiksi suurissa ja hyvin rakenteellisissa telekommunikaatiojärjestelmissä ja sulautetuissa tietokonejärjestelmissä, kuten myös laajoissa ohjelmistoprojekteissa.

Projektipäälliköt ja ohjelmistokehittäjät pitävät XP:n merkittävimpänä menestystekijänä pariohjelmointia sen ollessa samalla XP:n käytännöistä ainoa, jota on tutkittu laajemmin. Tutkimusten perusteella pariohjelmoinnilla on positiivinen vaikutus ryhmähenkeen ja sitä kautta myös työtyytyväisyyteen. Abrahamssonin ja Hulkon [Abr05] tutkimuksessa esitetään, että pariohjelmointi lisää myös itseluottamusta ja se on tehokas keino uusien taitojen oppimiseen ja harjoittelemiseen. Pariohjelmointia voi kuitenkin olla vaikea järjestää, mikäli organisaatiossa on joustavat työajat ja ryhmän jäsenten vuorokausirytmit poikkeavat toisistaan.

Eräs tutkituimpia pariohjelmoinnin ominaisuuksia on ollut tuottavuuden ja kulujen lisäyksen suhde. Saadut tutkimustulokset viittaavat siihen, että pariohjelmointi on yksin työskentelyä tehokkaampaa. Pariohjelmointi vaatii enemmän panostuksia kuin yksintyöskentely, mutta kulujen kasvua ei tapahdu samassa suhteessa. Siirtymävaiheessa kulujen kasvu on nopeampaa, mutta tuottavuuden kasvaessa pariohjelmoinnista tulee yksin työskentelyä tehokkaampaa. Tässä yhteydessä saatiin selkeä vastaus, koska pariohjelmointia voidaan käyttää ja koska ei. Mikäli lyhyt aika on oleellisen tärkeä ohjelmistoprojektin onnistumisen

kannalta, kannattaa kehittäjien määrän lisääminen pariohjelmoinnin toteuttamiseksi toteuttaa huolimatta kasvavista kustannuksista.

Pariohjelmoinnin soveltuvuudesta suurten reaalimaailman sovellusten kehittämiseen ei voida nyt käsiteltyjen tutkimusten perusteella esittää kuin suuntaa antavia kommentteja. Reaalimaailman ohjelmistokehityksessä pariohjelmointia käyttäneet projektit ovat kuitenkin valmistuneet 40–50 prosenttia nopeammin yksin ohjelmoitaviin projekteihin verrattuna. Pariohjelmoinnilla säästetään kehitystyön kokonaiskustannuksissa vaikka henkilöstöä tarvitaankin enemmän. Nosek [Nos98] nostaa esiin kaksi syytä, miksi käyttää pariohjelmointia. Ensinnäkin huonosti kirjoitettu koodi aiheuttaa enemmän virheitä järjestelmän elinkaaren aikana kuin hieman useamman työtunnin käyttäminen laadukkaan koodin kirjoittamiseen pareittain. Toiseksi ohjelmiston laatu paranee, kun algoritmi viestitetään työparille ennen sen kääntämistä ohjelmakoodiksi.

Lopulta Abrahamssonin ja Hulkon [Abr05] tutkimuksessa tehdään yhteenvetoa pariohjelmoinnista suoritetuista empiirisistä tutkimuksista. Sen mukaan pariohjelmointi on tuottavampaa kuin yksin työskentely, pariohjelmointia käyttämällä tehdään vähemmän virheitä ja se on hyödyllisempää monimutkaisissa tehtävissä. Pariohjelmoinnin lopullinen kustannushyöty ei ole kuitenkaan tarkasti tiedossa. Tutkimus ei löydä merkkejä siitä, että jompikumpi työskentelytapa olisi ylivertaista toiseen verrattuna, mutta huomauttaa yksintyöskentelyä käyttäneiden tulosten vaihtelevan suuresti. Vastaavasti pariohjelmointia käyttäneet näyttävät saaneen samankaltaisia tuloksia tehokkuuden osalta tutkimuksesta toiseen.

7 YHTEENVETO

Ohjelmistoprojektit ovat tasapainoilua kustannusten, ajan, laadun ja ohjelmiston laajuuden välillä. Niinpä projekteilla on suuri todennäköisyys epäonnistua. Syitä tähän voidaan hakea esimerkiksi vaatimusten huonosta määrittelystä ja analysoinnista tai asiakkaan osallistumisen puuttumisesta. Erääksi ratkaisuksi ohjelmistokehityksen ongelmiin on luotu ketterämpiä prosessimalleja, joiden pyrkimyksenä on tuoda joustavuutta ohjelmistojen kehitykseen. XP:tä pidetään yleisesti tunnetuimpana ketteränä menetelmänä, mutta siihen ja muihin ketteriin menetelmiin suhtaudutaan teollisuudessa ristiriitaisesti. Monet XP:n käytännöistä näyttävät kuitenkin tutkimusten perusteella toimivan hyvin. Mikään yksittäinen menetelmä ei kuitenkaan voi ratkaista kaikkia ohjelmistoprojektien ongelmia.

Tutkielman tarkoituksena oli käsitellä, millä tavoin XP vastaa ohjelmistoalalla ilmeneviin ongelmiin. Tutkielmassa tarkasteltiin XP:n vaikutuksia ohjelmistotuotantoon ja se esitteli lyhyesti yleisimpiä käytössä olevia perinteisiä ja ketteriä prosessimalleja, sekä tutki niiden eroavaisuuksia ja soveltuvuuksia erilaisiin ohjelmistoprojekteihin. Tarkimmin esiteltiin perinteinen ja edelleenkin yleisesti käytössä oleva vesiputousmalli sekä uudempi XP-malli, sillä nämä mallit ovat toimineet osaltaan esikuvina muille perinteisille ja ketterille prosessimalleille. Tutkimusten perusteella oli tarkoitus saada muodostettua kokonaiskuva minkälaisia vaikutuksia XP:n käyttöönotolla ja sen eri käytäntöjen käyttöönotolla on ollut. Lisäksi tarkasteltiin XP:n vaikutuksia ohjelmistoprosessin tuottavuuteen, sekä muita sen käytöstä saatuja kokemuksia. Tarkoituksena oli myös selventää pariohjelmoinnin roolia eräänä XP:n keskeisenä työmenetelmänä.

Tutkielmassa esitettiin, että ohjelmistokehitys on siirtymässä tarkasti määritellyistä suurista järjestelmistä kohti järjestelmiä, joiden kehityssyklit mitataan kuukausissa ja toimintaympäristö muuttuu jatkuvasti. Nämä seikat pakottavat etsimään uusia tapoja kehittää ohjelmistoja nopeasti ja siten, että ohjelmiston laatu pysyy korkeana. Tutkimukset esittävät, että kehittäjien positiivinen asennoituminen XP:hen ennakoi sen ja muiden ketterien menetelmien merkittävästi nykyistä laajempaa käyttöä tulevaisuudessa. Tutkimustulokset antavat ymmärtää, että XP:llä on muita prosessimalleja korkeampi tuottavuus. Ne vahvistavat

myös sen seikan, että XP vaatii vähemmän panostuksia projektin alkuvaiheissa ja erityisesti vaatimusten määrittelyssä.

XP eroaa merkittävästi perinteisesti käytetyistä menetelmistä ja sen voidaan katsoa olevan parhaimmillaan projekteissa, joissa järjestelmän vaatimukset eivät ole selvillä tai niiden uskotaan muuttuvan projektin aikana. XP tarjoaa myös hyviä käytäntöjä, joita voidaan käyttää ohjelmistoprojekteissa, vaikkei XP:tä käytettäisikään kokonaisuudessaan. Hyödyllisimmiksi käytännöistä koettiin tutkimusten perusteella pariohjelmointi, koodin yhteisomistus, yksinkertainen rakenne, testauslähtöinen kehitys sekä jatkuva integrointi. Tutkielmassa esitellyt tutkimukset pitävät XP:n tärkeimpinä menestystekijöinä testausta, pariohjelmointia sekä sitä, että XP:ssä keskitytään lopputuloksen kannalta olennaisiin päämääriin.

XP:tä kokonaisuudessaan koskevien tutkimusten yhteenvedoksi voidaan vetää, että se on luonut tehokkaan lähestymistavan eri ohjelmoijakeskeisiin käytäntöihin. Ohjelmistoprojekteista saadut kokemukset antavat suuntaa siten, että jotkin XP:n käytännöistä voivat olla hyvinkin hyödyllisiä suurissa ohjelmistoalan yrityksissä, kunhan niitä sovelletaan niin, että ne sopivat yrityksen organisaatioon ja prosesseihin. Tutkimukset esittävät että, XP:n menetelmiä ja käytäntöjä voidaan ottaa käyttöön uusissa yhteyksissä kuten esimerkiksi suurissa telekommunikaatiojärjestelmissä ja laajoissa ohjelmistoprojekteissa. Tutkimukset myös suosittelevat erilaisten hybridiprosessimallien kehittämistä, jolloin mallit sisältävät sekä ketteriä että perinteisiä työmenetelmiä ja -käytäntöjä

Eräs esimerkki hybridiprosessimallin kehittämisestä on VTT:n Abrahamssonin työryhmän kehittämä Mobile-D, jota käytetään mobiililaitteiden ohjelmistosovelluksiin. Mobile-D perustuu XP:n kehityskäytäntöihin, Crystal-menetelmien mahdollisuuteen valita käytettävä menetelmä projektiryhmän koon ja projektin kriittisyyden mukaan sekä RUP:n elinkaaren kattavuuteen. Mobile-D on optimoitu alle kymmenen hengen kehitysryhmille ja sen tavoitteena on luoda täysin toimiva mobiilisovellus alle kymmenessä viikossa. Sen käytäntöihin kuuluvat esimerkiksi jatkuva integrointi ja pariohjelmointi, jotka ovat XP:stä tuttuja. Abrahamsson sai vuonna 2007 Nokia Foundationin palkinnon ohjelmistokehitystyöstään ja eräänä perusteena mainitaan juuri Mobile-D. [Abr04] [Web07]

Suomeen ketterät ohjelmistonkehityksen menetelmät rantautuivat vuonna 2002. Samaan aikaan käynnistettiin myös VTT:ssä strateginen tutkimus ketterien menetelmien soveltamisen mahdollisuuksista ohjelmistoalan yrityksissä. Suomessa ketterien menetelmien tutkimus näyttää keskittyneen VTT:n Oulun tutkimuskeskukseen ja Oulun yliopistoon. VTT:n julkaisemat tutkimukset antavat ymmärtää, että taustalla saattaisi olla myös matkapuhelinvalmistaja Nokian kiinnostus koskien ketterien menetelmien soveltamisesta saatavia mahdollisuuksia. Tähän on saatu vahvistusta tarkastelemalla kyseisen yrityksen työpaikkailmoituksia pidemmältä ajanjaksolta. Ketterien menetelmien laaja käyttöönotto Nokian taholta aiheuttaa sen, että yrityksen ohjelmistoalihankkijoiden on omaksuttava samoja menetelmiä. Nämä yhdessä olivat syinä valittaessa tutkielmalle aihetta.

XP:n käytännöistä on pariohjelmointia tutkittu eniten ja tutkimustulokset viittaavat siihen, että se on tehokkaampaa kuin työskennellä yksin. Pariohjelmointi vaatii yksintyöskentelyä enemmän panostuksia, mutta kulujen kasvua ei tapahdu samassa suhteessa. Mikäli lyhyt aika on oleellisen tärkeä ohjelmistoprojektin onnistumisen kannalta, kannattaa kehittäjien määrän lisääminen pariohjelmoinnin toteuttamiseksi toteuttaa huolimatta kasvavista kustannuksista. Pariohjelmoinnin soveltuvuudesta suurten reaalimaailman sovellusten kehittämiseen ei voida nyt käsiteltyjen tutkimusten perusteella esittää perusteltuja arvioita, mutta ohjelmistokehityksessä pariohjelmointia käyttäneet projektit ovat kuitenkin valmistuneet 40–50 %:a nopeammin yksin ohjelmoitaviin projekteihin verrattuna.

Tulevaisuus näyttää, mikä XP:n ja muiden ketterien menetelmien rooli tulee olemaan ohjelmistoteollisuudessa, mutta voidaan olettaa niiden käytön yleistyvän etenkin Internetin ja mobiililaitteiden ohjelmistokehityksessä. Tutkimuksissa suositellaan erilaisten hybridiprosessimallien suunnittelemista ja käyttöä. Tällaisten menetelmien luominen, käyttöönotto ja saatujen tulosten tutkiminen voisikin olla eräänä jatkotutkimuksen aiheena. Koska ohjelmistoprojektit ovat aina tasapainoilua kustannusten, ajan, laadun ja ohjelmiston laajuuden välillä, olisi mielenkiintoista tutkia kuinka lyhentää ohjelmistotuotteesta olevan idean ja valmiin ohjelmistotuotteen välillä olevaa ajallista välimatkaa, kuinka se voitaisiin tehdä kustannustehokkaasti, laadusta tinkimättä ja millä eri malleilla siihen voitaisiin päästä.

8 LÄHDELUETTELO

[Abr02] Abrahamsson Pekka, Salo Outi, Ronkainen Jussi ja Warsta Juhani, “Agile software development methods. Review and analysis”, VTT Publications, saatavilla WWW-muodossa <URL: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf>. 2002.

[Abr03] Abrahamsson Pekka, EXtreme Programming: First Results from a Controlled Case Study, “Proceedings of the 29th EUROMICRO Conference “New Waves in System Architecture (EUROMICRO’03)”, IEEE Computer Society, s. 1–8, 2003.

[Abr04] Abrahamsson Pekka, Hanhineva Antti, Hulkko Hanna ja Ihme Tuomas, Mobile-D: An Agile Approach for Mobile Application Development, VTT Publications, Communications of the ACM, s. 174–175, 2004.

[Abr05] Abrahamsson Pekka ja Hulkko Hanna, A Multiple Case Study on the Impact of Pair Programming on Product Quality, Communications of the ACM, s. 495–504, 2005.

[Amb02] Ambler Steven, ”Agile Modeling”, John Wiley & Sons, Inc., New York. 2002.

[Amb05] Ambler Steven, Quality in an agile world, Software Quality Professional, IEEE Computer Society, 2005, s. 34–40.

[AnD03] Anderson David J. ja Schragenheim Eli, ”Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results”, Prentice Hall PTR, 2003.

[Ans08] Answers.com, “Chrysler Comprehensive Compensation System”, WWW-sivusto, saatavilla WWW-muodossa <URL:http://www.answers.com/Chrysler%20Comprehensive%20Compensation%20System, viitattu 29.01.2008.

[Ast02] Astels David, Miller Granville ja Novak Miroslav, ”A practical guide to eXtreme Programming”, Prentice Hall PTR, Upper Saddle River, NJ 07458, 2002.

[Aue03] Auer Ken, Meade Erik ja Reeves Gareth, ”The Rules of XP”, saatavilla WWW-muodossa <URL:http://www.rolemodelsoftware.com/moreAboutUs/publications/rulesOfXp.php/>, 2003, viitattu 16.8.2007.

[Bah03] Bahli Bouchaib ja Abou Zeid El Sayed, The role of knowledge creation in adopting Extreme Programming: An empirical study, IEEE Computer Society, s. 75–87, 2003.

[Bas03] Baskerville Richard, Ramesh Balasubramaniam, Levine Linda, Pries-Heje Jan ja Slaughter Sandra, Is Internet-speed software development different?, IEEE Computer Society, s.70–77, 2003.

[Bec99] Beck Kent, Embracing Change with Extreme Programming, IEEE Computer Society, s. 70–77, 1999.

[Bec00] Beck Kent, ”Extreme Programming Explained”, Addison-Wesley Inc., 2000.

[Bec01] Beck Kent, Beedle Mike, van Bennekum Arie ja Cockburn Alistair,”Manifesto for Agile Software Development”, saatavilla WWW-muodossa <URL: http://www.agile-manifesto.org/principles.html>, 2001, viitattu 4.7.2007

[Ben87] Benington Herbert D., Production of large Computer Programs, “Proceedings of ONS. Symp. Advanced Programming Methods for Digital Computer, 1956, myös: Proceeding of the Ninth International Conference on Software Engineering”, IEEE Computer Society, s.15-27, 1987.

[Boe94] Boehm Barry ja Bose Prasanta, A Collaborative Spiral Software Process Model Based on Theory W, USC Center for Software Engineering, IEEE Computer Society, s. 59-68, 1994.

[Boe00] Boehm Barry, Wilfred J. Hansenin editoima, Spiral Development: Experience, Principles and Refinements, Spiral Development Workshop, Communications of the ACM, s. 1-37, 2000.

[Boe02] Boehm Barry, Get ready For the Agile Methods, With Care, IEEE Computer Society, s. 64- 69, 2002.

[Boe04] Boehm Barry ja Turner Richard, ”Balancing Agility and Discipline: A Guide for the Perplexed”, Pearson Education Inc., 2004.

[Bro87] Brooks Frederick P jr.,"No Silver Bullet: Essence and Accidents of Software Engineering", saatavilla WWW-muodossa <URL:http://www.lips.utexas.edu/ee382c-15005/Readings/Readings1/05-Broo87.pdf>, 1987.

[Bun04] Bunse Christian, Feldmann Raimund ja Dörr Jörg, Agile Methods in Software Engineering Education, Springer-Verlag Berlin Heidelberg, IEEE Computer Society, s. 284–293, 2004.

[Bur03] Burke Eric M. ja Coyner Brian M., ”Java Extreme Programming Cookbook”, O'Reilly, 2003.

[Cio03] Ciolkowski Marcus ja Schlemmer Michael, Experiences with a Case Study on Pair Programming, Communications of the ACM, s. 1–7, 2003.

[Coa00] Coad Peter, Lefebvre Eric ja De Luca Jeff, ”Java Modeling In Color with UML: Enterprise Components and Process”, Prentice Hall, 2000.

[Coc00] Cockburn Alistair ja Williams Laurie, The Costs and Benefits of Pair Programming, University of Utah Computer Science, Communications of the ACM. S. 1-11, 2000.

[Coc01] Cockburn Alistair, ”Surviving Object-Oriented projects”, Addison-Wesley, Boston, 2001.

[Coc02] Cockburn Alistair, ”Agile Software Development”, Addison-Wesley, 2002.

[Coc07] Cockburn Alistair, “Draft for Crystal Clear”, saatavilla WWW-muodossa <URL:http://web.archive.org/web/20040405010325/http://members.aol.com/humansandt/crystal/clear/>, viitattu 10.7.2007.

[Dal05] Dalcher Darren, Benediktsson Oddur ja Thorbergsson Helgi, Development Life Cycle Management: A Multiproject Experiment, IEEE Computer Society, s. 1–8, 2005.

[Dav90] Davenport Thomas H. ja Short James E., The New Industrial Engineering: Information Technology and Business Process Redesign, Sloan Management Review, IEEE Engineering Management Review, s. 46-60, 1990.

[Deu01] van Deursen Arie, Customer Involvement in Extreme Programming, XP2001 Workshop Report, Communications of the ACM, s.1-4, 2001.

[DSD07] DSDM Consortium, "Dynamic Systems Development Method", WWW-sivusto <URL: http:// WWW.dsdm.org> >, 2007.

[Ghe91] Ghezzi Carlo, Jazayer Mehdi ja Mandrioli Dino, "Fundamentals of software engineering", Prentice-Hall, 1991.

[Gil88] Gilb Tom, "Principles of software engineering management", Addison-Wesley, 1988.

[Gil93] Gilb Tom ja Graham Dorothy, "Software Inspection", Addison-Wesley, 1993.

[Gre01] Grenning James, Extreme Programming at a Process-Intensive Company. XP Universe Launching ,IEEE, s. 27-33, 2001.

[Hai04] Haikala Ilkka ja Märijärvi Jukka, "Ohjelmistotuotanto", Suomen ATK-kustannus Oy, Helsinki, 2004.

[Har03] Harrison Neil B., A Study of Extreme Programming in a Large Company, IEEE Computer Society, s. 1–7, 2003.

[Has07] Hass Kathleen B.,"The Blending of Traditional and Agile Project Management", PM World Today - May 2007 (Vol. IX, Issue V)", saatavilla WWW-muodossa <URL:http://www.pmforum.org/library/tips/2007/PDFs/Hass-5-07.pdf>, viitattu 11.6.2007

[Hed03] Hedin Görel, Bendix Lars ja Magnusson Boris, Introducing Software Engineering by means of Extreme Programming, Department of Computer Science Lund Institute of Technology, IEEE Computer Society, s. 586-593, 2003.

[Hen07] Henderson-Sellers Brian ja Qumer Asif, “An evaluation of the degree of agility in six agile methods and its applicability for method engineering, Information and Software Technology”, saatavilla WWW-osoitteessa <URL: http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V0B-4N1JRNN3&_user=1234512&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000052082&_version=1&_urlVersion=0&_userid=1234512&md5=b2507f8b27583909b500dbb87cd09bce>, 2007, Viitattu 20.01.2008

[Hig00] Highsmith Jim,”Retiring lifecycle dinosaurs, Software Testing and Quality Engineering, July/August”, saatavilla WWW-muodossa <URL:http://www.jimhighsmith.com/articles/Dinosaurs.pdf>, 2000.

[Hum89] Humphrey Watts S. ja Kellner Marc I, Software Process Modeling: Principles of Entity Process Models, Communications of the ACM, s. 331–342, 1989.

[Huo04] Huo Ming, Verner June, Zhu Liming ja Babar Muhammad Ali, Software quality and agile methods, “COMPSAC 04: Proceedings of the 28th Annual International Computer Software and Applications Conference”, IEEE Computer Society, s. 520–525, 2004.

[Hut03] Hutcheson Marnie L., Testing in the eXtreme Programming (XP) Paradigm: A Case Study, IEEE Computer Society, s. 1–12, 2003.

[Jac95] Jacobson Ivar, Ericsson Maria ja Jacobson Agneta, ”The Object Advantage: Business Process Reengineering With Object Technology”, Addison-Wesley, 1995.

[Jar07] Jarvis Bob ja Gristock Stephen P, ”Extreme Programming (XP) Six Sigma CMMI JPOrgan Chase case study”, saatavilla WWW-muodossa <URL: http://www.sei.cmu.edu/cmmi/presentations/sepg05.presentations/jarvis-gristock.pdf>, viitattu 27.08.2007.

[Jef99] Jeffries Ronald,”Extreme testing. Software testing & quality engineering March/April 1999”, saatavilla WWW-muodossa <URL: http://www.xprogramming.com/SP99%20Extreme%20for%20Web.pdf>, viitattu 28.08.2007.

[Jef07] Jeffries Ronald, “XProgramming.com, an Agile Software Development Resource”, WWW-sivusto <URL: http://www.xprogramming.com/xpmag/whatisxp.htm>, 2007.

[Jen03] Jensen Randall W., “A pair programming experience”, saatavilla WWW-muodossa <URL:http://http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html>, 2003, Viitattu 22.02.2008

[Jor94] Jorgensen Paul C. ja Erickson Carl, Object-Oriented Integration Testing, Vol. 37 No. 9, Communications of the ACM, s.29-38, 1994.

[Kee04] Keefe Karen, Dick Martin, “Using Extreme Programming in a Capstone Project”. CRPIT ’30: Proceedings of the sixth conference on Australian computing education, Communications of the ACM, s. 151– 160, 2004.

[Ker02] Keefer Gerold,”Extreme Programming Considered Harmful for Reliable Software Development”, StickyMinds.com, AVOCA GmbH, saatavilla WWW-muodossa <URL: http://www.agilealliance.com/system/article/file/945/file.pdf>, s. 1-18>, 2002.

[Kle03] Kleppe Anneke, Warmer Jos ja Bast Wim, ”MDA Explained: The Model Driven Architecture: Practice and Promise”, Pearson Education Inc., Boston, 2003.

[Kol07] Kollanus Sami, ”Projektinhallinta, TJTA330 Ohjelmistotuotanto”, Saatavilla WWW-muodossa URL: http://users.jyu.fi/~kolli/OHTU2007/materiaali/Projektinhallinta.pdf, 17.1.2007.

[Kre05] Krebs Joe, ”RUP in the dialogue with Scrum”, IBM Rational”, saatavilla WWW-muodossa <URL:http://www-128.ibm.com/developerworks/rational/library/feb05/krebs/>, viitattu 24.05.2007.

[Kro07] Kroll Per,”Transitioning from waterfall to iterative development”, saatavilla WWW-muodossa URL:http://www-128.ibm.com/developerworks/rational/library/4243.html>, viitattu 16.05.2007.

[Kru01] Kruchten Philippe, ”The Rational Unified Process: An Introduction”, Boston: Addison-Wesley Longman Publishing Co, 2001.

[Lay05] Layman Lucas, Williams Laurie ja Cunningham Lynn, Motivations and Measurements in an Agile Case Study, Communications of the ACM, s. 14–24, 2005.

[LeJ05] LeJeune Noel F.,Teaching software engineering practices with Extreme programming, Consortium for Computing Sciences in Colleges (CCSC), s. 107–117, 2005.

[Mau02] Maurer Frank ja Martel Sebastien, Extreme programming, Rapid development for Web-based applications, IEEE Internet Computing, s. 86-90, 2002.

[McC02] McConnell Steve, “Ohjelmistotuotannon hallinta”, IT Press, Helsinki, 2002.

[McD02] McDowell Charlie, Werner Linda, Bullock Heather ja Fernald Julian, The effects of pair-programming on performance in an introductory programming course, SIGCSE'02, Communications of the ACM, s. 38-42, 2002.

[McK04] McKinney Dawn, Froeseth Julie, Robertson Jason, Denton Leo F. ja Ensminger David, Agile CS1 Labs:eXtreme Programming Practices in an Introductory Programming Course, XP/Agile Universe 2004, IEEE Computer Society, s. 164–174, 2004.

[Mel05] Melnik Grigori ja Maurer Frank, A Cross-Program Investigation of Students Perceptions of Agile Methods, ICSE’05, Communications of the ACM, s. 481–488, 2005.

[Mis06] Misic Vojislav B., Perceptions of Extreme Programming: An Exploratory Study, ACM SIGSOFT Software Engineering, s. 1–8, 2006.

[Mit03] Mitchell Ian,”Extreme Programming, The New Zealand Way”, WWW-sivusto <URL:http://www.xp.co.nz>, 2003.

[Mou07] Mountain Goat Software, ”Agile Development Training and Consulting”, WWW-sivusto URL: http://www.mountaingoatsoftware.com/scrum, viitattu 07.07.2007.

[Mue02] Mueller Gary ja Borzuchowski Janet, Extreme Embedded – A Report from the Front Line, IEEE Computer Society, s. 1-8, 2002.

[Mül02] Müller Matthias ja Hagner Oliver, Experiment about test-first programming, IEEE Proceedings Software 149, s. 131-136, 2002.

[New02] Newkirk James, Introduction to agile processes and extreme programming, “Proceedings of the 25th international conference on Software engineering”, Communications of the ACM, s. 695-696, 2002.

[Nos98] Nosek John T., The case for collaborative programming, Communications of the ACM, March 1998/Vol. 41,

[Not88] Notkin David, The Relationship between Software Development Environments and the Software Process, ACM, s. 107-109, 1988. through process modeling, IEEE Computer Society, s. 29-44, 1991.

[Ost91] Osterweil Leon J. ja Song Xiping, Comparing design methodologies through process modeling. 1 st International Conference on Software Process. IEEE CS Press, s. 105–108, 1991.

[Ost92] Osterweil Leon J. ja Song Xiping, Towards objective, systematic design-method comparisons, IEEE Software, s. 43-54, 1992.

[Pad03] Padberg Frank ja Muller Matthias M., Analyzing the Cost and Benefit of Pair Programming, “Proceedings of the Ninth International Software Metrics Symposium (METRICS’03)”, IEEE Computer Society, s. 1–12, 2003.

[Pal02] Palmer Stephen ja Felsing John, ”A Practical Guide to Feature-Driven Development”, Prentice Hall, 2002.

[Pre05] Pressman Roger S.,”Software Engineering: A Practitioner’s Approach”, McGraw-Hill, Singapore, 2005.

[Ree99] Reel, John S, Critical Success Factors in Software Projects, IEEE Software, May/June 99, s. 19-23, 1999.

[Ris00] Rising Linda ja Janoff Norman S., The Scrum Software Development Process for Small Teams, IEEE Software, July/August 00, s.2-8, 2000.

[Rob07] Robinson Phil,"The Importance of Being Agile", saatavilla WWW-muodossa <URL:http://www.lonsdalesystems.com/dokuwiki/lib/exe/fetch.php?id=documents%3Awhitepapers&cache=cache&media=documents:agile_2.pdf>, viitattu 16.05.2007.

[Roy70] Royce Winston W.,Managing the Development of Large Software Systems, Proceedings IEEE WESCON, August 70, s. 1-9, 1970.

[Rum02] Rumpe Bernhard ja Schröder Astrid, Quantitative Survey on Extreme Programming Projects, ACM, s. 1–6, 2002.

[Schn03] Schneider Jean-Guy ja Johnston Lorraine, eXtreme Programming at universities: an Educational Perspective, IEEE, s. 594–599, 2003.

[Schw06] Schwaber Ken ja Beedle Mike A., "Agile Software Development with Scrum", Pearson Education, 2006.

[Scr07] Scrum-metodologian verkkosivusto, WWW-sivusto <URL: http://www.control-chaos .com/resources/>, 7.7.2007.

[She06] Sherrell Linda B. ja Robertson Jeff J, Pair programming and agile software development: Experiences in a gollege setting, Consortium for Computing Sciences in Colleges CCSC, s. 145–153, 2006.

[Shu02] Shukla Anuja ja Williams Laurie, Extreme Programming For A Core Software Engineering Course, "Proceedings of the 15th Conference on Software Engineering Education and Training (CSEETí02)", IEEE, s. 1–8, 2002.

[Smi01] Smith Suzanne ja Stoecklin Sara, What We Can Learn from Extreme Programming, Consortium for Computing in Small Colleges CCSC, s. 144–151, 2001.

[SoD07] Software Development Times-verkkojulkaisu,"Standish Group Report: There's Less Development Chaos Today", saatavilla WWW-muodossa <URL: http://www.sdtimes.com/content/article.aspx?ArticleID=30247>, viitattu 11.6.2007.

[SoM04] SoftwareMag.com-verkkojulkaisu,”Standish: Project Success Rates Improved Over 10 Years”, Saatavilla WWW-muodossa <URL: http://www.softwaremag.com/ L.cfm?Doc =newsletter/2004-01-15/Standish>, viitattu 29.5.2007.

[Somm00] Sommerville Ian, ”Software Engineering 5th ed”, Addison-Wesley Inc., 2000.

[Sta95] Standish Group-verkkojulkaisu,”Chaos Report. Standish Group”, 1995, saatavilla WWW-muodossa <URL:http://www.standishgroup.com/sample_research/index.php>, viitattu 29.5.2007.

[Sta98] Standish Group-verkkojulkaisu,”Chaos: A Recipe for Success. Standish Group”, 1998, saatavilla WWW-muodossa <URL:http://www.ii.metu.edu.tr/~is526/course_material /papers/ChaosReport1998.pdf>, viitattu 29.5.2007

[Stap02] Stapleton Jim, ”Dynamic systems development method - The method in practice”, Addison-Wesley, 2002.

[Ste04] Steinberg Daniel ja Palmer Daniel, ”Extreme software engineering: a hands-on approach”, Pearson Education Inc., Upper Saddle River, New Jersey, 2004.

[Sto03] Stojanovic Zoran, Dahanayake Ajantha ja Sol Henk, Modeling and Architectural Design in Agile Development Methodologies, “International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, EMMSAD03, Velden, Austria, 2003”, saatavilla WWW-muodossa <URL:http://www.emmsad.org/2003/Final%20 Copy/27.pdf>, viitattu 31.12.2007.

[Tab00] Taber Cara ja Fowler Martin, An iteration in the life of an XP project, Cutter IT journal, November 2000, s. 13-21, saatavilla WWW-muodossa <URL:http://www.scribd. com/doc/196726/An-iteration-in-the-life-of-XP-project>, viitattu 31.12.2007.

[The03] Theunissen, W. H. Morkel, Kourie, Derrick . G. ja Watson, Bruce. W., Standards and agile software development. SAICSIT ’03: Proceedings of the 2003 annual research conference. South African Institute for Computer Scientists and Information Technologists, sivut 178–188, 2003.

[Tur02] Turk Dan, France Robert ja Rumpe Bernhard, Limitations of Agile Software Processes, “Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002 Alghero Italy”, IEEE, s. 1-4, 2002.

[Van05] Vanderburg Glenn, Extreme Programming Annealed, “OOPSLA’05, October 16–20, 2005, San Diego, California, USA”, ACM, s. 539–545, 2005.

[Vli00] Van Vliet Hans.”Software Engineering, Principles and Practice, Second edition”. John Wiley & Sons, Ltd. England, 2000.

[Web07] Webwire-verkkojulkaisu, “Nokia Foundation award to Pekka Abrahamsson”, saatavilla WWW-muodossa <URL:http://www.webwire.com/ViewPressRel.asp?aId= 53925, viitattu 01.07.2008.

[Wel01] Wells Donovan J.,”Extreme Programming: A Gentle Instruction”, WWW-sivusto <URL: http://www.extremeprogramming.org/extremeprogrammingorg.zip>, 17.02.2006.

[Wik07a] Wikipedia, The Free Encyclopedia, “Chrysler Comprehensive Compensation System”, WWW-sivusto <URL:http://en.wikipedia.org/wiki/Chrysler_Comprehensive_ Compensation_System, 29.01.2008.

[Wik07b] Wikipedia, The Free Encyclopedia, “Extreme Programming”, WWW-sivusto <URL:http://en.wikipedia.org/wiki/Extreme_Programming>, 29.04.2007.

[Wik07c] Wikipedia, The Free Encyclopedia,”Software crisis”, WWW-sivusto, <URL:http: //en.wikipedia.org/wiki/Software_crisis>, 24.04.2007.

[Wil00] Williams Laurie, Kessler Robert, Cunningham Ward ja Jeffries Ron, Strengthening the Case for Pair-Programming, July/August 2000 IEEE Software, s. 19–25, 2000.

[Wil04] Williams Laurie ja Layman Lucas, Toward a framework for evaluating extreme

programming. "8th Internation Conference on Empirical Assessment in Software Engineering (EASE 2004)" Workshop - 26th International Conference on Software Engineering , s. 11–20, 2004.

[Zus05] Zuser Wolgang, Heil Stefan ja Grechenig Thomas, Software quality development and assurance in RUP, MSF and XP: a comparative study. Proceedings of the third workshop on Software quality, ACM, s. 1–6, 2005.