Akseli Lajunen

EXTREME PROGRAMMING

Tietojärjestelmätieteen
kandidaatintutkielma
21.04.2007

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Lajunen, Akseli Herman
Tietojärjestelmätieteen kandidaatintutkielma
Jyväskylä: Jyväskylän yliopisto, 2007.
45 s.

Tässä tutkielmassa esitellään Extreme Programming prosessinhallinta-menetelmä ja kuinka se vastaa ohjelmistotuotannossa esiintyviin ongelmiin. Ohjelmistotuotannossa yleisesti esiintyviä ongelmia ovat aikataulujen ja resurssien ylittyminen, liiallinen virheiden määrä ja ohjelmiston vaatimuksien muuttuminen kesken projektin.

Extreme Programming on pienille tai keskisuurille projekteille sopiva ketterä prosessinhallintamenetelmä, joka sallii epämääräisesti määritellyt vaatimukset sekä niiden muuttumisen ohjelmiston elinkaaren aikana. Se panostaa runsaan dokumentoinnin sijaan kommunikaatioon ja tiiviiseen yhteistyöhön asiakkaan kanssa.

AVAINSANAT: Extreme Programming, ketterä prosessinhallintamenetelmä.

SISÄLLYSLUETTELO

1 JOHDANTO

Ohjelmistoala on kohdannut elinaikanaan monia ongelmia. Projekteille asetetut aikataulut ja resurssit ylittyvät. Tuotettu ohjelmisto ei aina vastaa asiakkaan vaatimuksia. Ohjelmisto saadaan toteutettua, mutta se sisältää liiallisesti virheitä, jotta se voitaisiin ottaa tuotantokäyttöön. Ongelmista johtuen ohjelmoijat joutuvat tekemään ylitöitä ja väsyvät jatkuvaan kiireeseen.

Erilaisilla prosessinhallintamenetelmillä on pyritty ratkaisemaan ohjelmistoalalla esiintyviä ongelmia. Perinteisiin prosessinhallintamenetelmiin liittyy usein paljon raskaita suunnittelu- ja dokumenttipainotteisia vaiheita, joissa pyritään hallitsemaan riskiä suunnittelemalla projekti etukäteen mahdollisimman pitkälle. Tällaisissa menetelmissä ohjelmistoprosessi kuvataan rationaaliseksi prosessiksi, jossa stabiileista asiakasvaatimuksista päädytään vaatimukset täyttävään ohjelmistoon. Todellisuudessa vain muutos on pysyvää: Käytännössä asiakasvaatimukset elävät koko ohjelmiston elinkaaren ajan ja jo tuotantoprosessin aikana (Haikala & Märijärvi 2001).

Kun perinteiset menetelmät ovat kankeita muutokselle ja muutenkin hitaita liikkeissään, ketterät menetelmät tarjoavat niille vaihtoehdon. Ketterät menetelmät sallivat epämääräisesti määritellyt vaatimukset sekä niiden muuttumisen ohjelmiston elinkaaren aikana, kestävät painetta asiakkaan ja tuotantotiimin välillä sekä mahdollistavat tuotteen aikaisen toimituksen. (Cohn & Ford 2003)

Vuonna Kent Beck (1999) esitteli kirjassaan Extreme Programming Explained prosessinhallintamenetelmän nimeltä Extreme Programming, jonka jälkeen se on noussut hyvin suosituksi ja keskustelua herättäväksi menetelmäksi. Se on saanut paljon puolestapuhujia, mutta myös menetelmän vastustajia. XP

panostaa massiivisen dokumentaation sijasta kommunikaatioon ja dynaamiseen luonteeseensa. Menetelmä pyritään pitämään keveänä ja yritetään päästä tuottamaan asiakkaalle järjestelmän ydinarvoa mahdollisimman nopeasti sekä kehittämään tuotetta asteittain eteenpäin. Toimiakseen menetelmä vaatii kommunikaation lisäksi tiivistä yhteistyötä asiakkaan kanssa. XP koostuu neljästä ydinarvosta ja kahdestatoista käytänteestä, jotka esitellään luvussa 3. Luvun lopussa käydään läpi myös vähemmän tunnettua uutta XP:tä, jonka Beck esitteli vuonna 2004.

Tavoitteena on selvittää millainen menetelmä Extreme Programming on ja millä tavoin se pyrkii vastaamaan ohjelmistokehityksessä esiintyviin ongelmiin.

2 KÄSITTEISTÖÄ

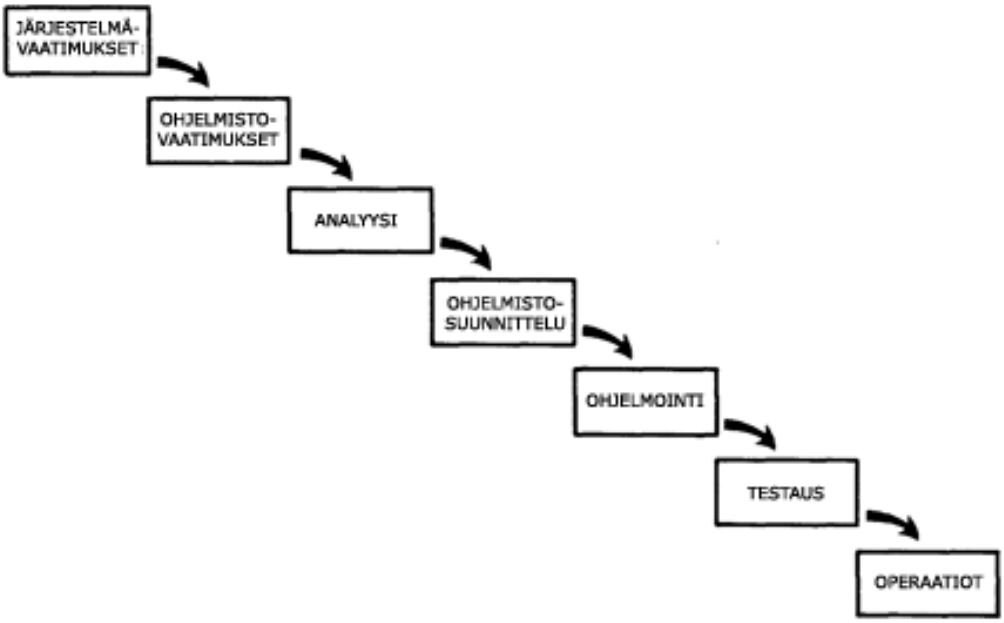
Tässä luvussa käydään läpi tutkielman keskeistä käsitteistöä sekä hieman ohjelmistoalan kehittymistä alkuajoista nykypäivään.

2.1 Elinkaarimallit

Elinkaarimalli määrittää sen kuinka ohjelmiston kehitys etenee ensimmäisestä ideasta viimeiseen pensselin vetoon. Elinkaarimallin päätehtävä on määritellä järjestys, jossa projekti määrittää, suunnittelee, toteuttaa, katselmoi, testaa ja toteuttaa sen toiminnot. Se asettaa kriteerit, joiden perusteella päätetään, voidaanko sen hetkisestä tehtävästä siirtyä toiseen. Koska elinkaari määrittelee projektin kokonaissuunnitelman, on sen valinnalla suuri vaikutus projektin onnistumiselle. Valitusta elinkaarimallista riippuen voidaan sillä minimoida turhaa työtä ja riskialttiutta sekä parantaa asiakassuhteita. Sopimaton elinkaarimalli voi taas olla hitaan ja tarpeettoman työn sekä ohjelmoijien turhautumisen lähde. (McConnell 2002)

2.2 Vesiputousmalli

Perinteisin elinkaarimalli on vesiputousmalli, jossa vaiheet seuraavat lineaarisesti toisiaan. Vesiputousmalli historia pohjaa Royce Winstonin vuonna 1970 kirjoittamaan artikkeliin ”Managing the Development of Large Software Systems”. Hän ei kuitenkaan koskaan kutsunut sitä vesiputousmalliksi eikä esitellyt sitä sellaisena, millaiseksi yleinen käsitys siitä muotoutui. (Weisert 2003). Hän ennemminkin kritisoi kuvion 1 mukaista mallia, jossa jokaisen vaiheen tuotokset ovat stabiileja, eikä niitä muutostarpeista huolimatta lähdetä muokkaamaan. (Winston 1970)



Kuvio 1

Vesiputousmallin kaltaisissa menetelmissä ei siirrytä seuraavaan vaiheeseen ennen kuin edellinen on saatu valmiiksi. Vaiheen hyväksymisen jälkeen sitä ei ole myöskään tarkoitus palata jälkikäteen korjailemaan. Vesiputousmalli pohjautuu vahvasti dokumentaatioon, jolloin jokaisen vaiheen tuotoksena saadaan tietty dokumentaatio. Vesiputousmallin on perinteisesti ajateltu poistavan virheet projektin halvassa vaiheessa suunnittelemalla projekti mahdollisimman pitkälle etukäteen (McConnell 2002).

Beckin mukaan vesiputousmallin ja muiden vastaavien menetelmien yksi suurimmista ongelmista on kuitenkin seuraavanlainen: Kehittäjät ovat vuosikymmeniä valittaneet: ”Asiakkaat eivät osaa kertoa meille mitä he haluavat. Kun annamme heille juuri sitä mitä he ovat määritelleet, he eivät pidä lopputuloksesta.” (Beck 1999). Asiakkaiden on siis vaikea tietää millaisen järjestelmän he haluavat ilman, että heillä on mitään konkreettista tukenaan.

Tilanteessa, jossa vaatimukset eivät ole tarkasti tiedossa, paremmin soveltuvan lähestymistavan tarjoavat iteratiiviset mallit, joita myös myöhemmin esiteltävät ketterät prosessinhallintamenetelmät käyttävät.

2.3 Iteratiivinen malli

Spiraalimalli on perinteisin esimerkki iteratiivisesta mallista. Alkuperäisen spiraalimallin esitteli Barry Boehm vuonna 1985. Spiraalimallissa yksi iteraatio sisältää useita vaiheita esitutkimuksesta ylläpitoon, ja iteraatiot toistuvat useamman kerran järjestelmän elinkaaren aikana. Kunkin iteraation jälkeen arvioidaan projektin riskit, ja katsotaan kannattaako projektia jatkaa. Mikäli projektia päätetään jatkaa, on päästy lähemmäksi lopputuotetta. Iteratiivinen malli mahdollistaa suunnitelmien tarkentamisen projektin edetessä. (Barry Boehm 1985)

2.4 Inkrementaalinen malli

Monia prosessinhallintamenetelmiä voidaan sanoa myös inkrementaalisiksi. Inkrementaalisessa mallissa lopputuotetta kehitetään lisäämällä siihen asteittain pieniä toiminnallisuuspalasia. (Haikala & Märijärvi 2001) Tällöin valitaan tietty toiminnallisuuskokonaisuus, joka toteutetaan ja lisätään järjestelmään. Kun inkrementti on valmis, se annetaan asiakkaalle, joka tutkii ja testaa sen, ja siitä tehtyjä havaintoja käytetään hyväksi seuraavia palasia määritellessä.

2.5 Prosessinhallintamenetelmä

Elinkaarimalleista on kehitelty erilaisia prosessinhallintamenetelmiä. Prosessinhallintamenetelmät sisältävät projektin elinkaarimallin lisäksi runsaasti muuta ohjeistusta projektin työskentelytavoista ja käytänteistä. Perinteiset prosessinhallintamenetelmät noudattavat vesiputousmallin kaltaista

elinkaarimallia. Iteraativinen sekä inkrementaalinen malli toteutuvat esimerkiksi tässä tutkielmassa käsitellyssä XP -menetelmässä.

2.6 Ketterät menetelmät

Ohjelmistotuotannon alkuaikoina ohjelmoijat ohjelmoivat ilman sääntöjä oman mielensä mukaan, ja ohjelmistoista tuli usein sellaisia, että vain tekijä tiesi miten ne toimivat. Kun ohjelmistojen koot kasvoivat, kävi ohjelmien ylläpito melko mahdottomaksi. Alkuajoista viisastuneina 80–luvulla ruvettiin laatimaan säännöstöjä siitä, kuinka ohjelma pitäisi toteuttaa ja dokumentoida. Näistä menetelmistä tuli todella raskaita, ja dokumentoinnin määrä ei pysynyt enää käsissä. Dokumentaation määrästä johtuen sitä oli myös vaikea pitää ajan tasalla. Raskailla menetelmillä projektia pystyi kyllä kontrolloimaan, mutta ohjelmoijat joutuivat tekemään paljon ylimääräistä työtä toimiessaan laadittujen säännöstöjen mukaisesti, eikä lopputuote aina ollut asiakasvaatimusten mukainen. (Wells 2001)

Vuosituhannen vaihteessa ongelmaan herättiin, ja ohjelmistoalalle saapui kevyempiä menetelmiä. Vuonna 2001 kevyempien menetelmien kehittäjät kokoontuivat saman katon alle ja kehittivät ketterien menetelmien ohjelmajulkaisun. Siihen on koottu neljä keskeistä arvoa, jotka ohjaavat työskentelyä. Ohjelmajulkaisun mukaan yksilöt ja vuorovaikutukset ovat tärkeämpiä kuin prosessit ja työkalut. On oleellisempaa keskittyä tekemään toimiva ohjelmisto kuin tuottamaan kattava dokumentaatio. Lisäksi se korostaa yhteistyön asiakkaan kanssa olevan tärkeämpää kuin sopimusneuvottelut. Myös muutoksiin vastaaminen ajaa ohi suunnitelman noudattamisen. (Highsmith 2001).

Ketterät menetelmät lähtevät periaatteesta, että vain muutama yksinkertainen säännöstö riittää. Vaikeasti noudatettavia sääntöjä yksinkertaistetaan, ja dokumentaatiosta karsitaan tarpeettomat pois. Ketterissä menetelmissä valitaan vain sellaiset käytänteet, jotka auttavat viemään ohjelmiston laatua eteenpäin, ja joita on helppo noudattaa. (Wells 2001)

Ketterät menetelmät ovat iteratiivisia ja kehitys etenee usein pienissä inkrementeissä.

3 EXTREME PROGRAMMING

Beck (1999) pohti ohjelmistoprosessin ongelmia, ja tuli siihen tulokseen, että jos joillakin ohjelmistotuotannon käytännöillä saadaan edesautettua ohjelmistoprosessin onnistumista, kannattaa silloin panostaa erityisesti näihin kohtiin. Esimerkiksi jos katselmointi parantaa koodin laatua, koodia katselmoidaan koko ajan kirjoittamalla kaikki koodi pariohjelmoinnilla. Jos testauksella pystytään vähentämään virheitä, suoritetaan testausta jatkuvasti automatisoimalla testitapaukset. Mikäli yksinkertaisuus parantaa ymmärrettävyyttä, jätetään ohjelma niin yksinkertaiseksi kuin sen hetkisellä toiminnallisuudella on mahdollista. Jos integrointitestaus on tärkeää, järjestelmä integroidaan ja testataan useamman kerran päivässä. (Beck 1999)

Beck (2004) vertaa XP:tä auton ohjaamiseen. Autolla ajaminen ei ole pelkästään auton suunnassa pitämistä, vaan sitä, että tehdään alati pieniä korjauksia tilanteen vaatimalla tavalla. Beckin mukaan kyseinen metafora toteutuu XP:ssä kahdella tasolla. Asiakas ohjailee järjestelmän sisältöä ja sitä, millaisia ongelmia järjestelmän kuuluisi ratkoa. Asiakas ei kuitenkaan tiedä valmiiksi reittiä, jota järjestelmän kehityksen tulisi noudatella, vaan hän keskittyy pitämään mielessään, mihin suuntaan horisontissa haluaa mennä. Viikko viikolta reitti tarkentuu tien kaarteiden mukaan. Aivan kuten asiakas ohjailee järjestelmän sisältöä, koko tiimi ohjailee kehitysprosessia alkaen käytössä olevista käytänteistä. Jokainen projekti on erilainen riippuen ajasta, paikasta ja projektiin liittyvistä henkilöistä. Tällöin yksi ja sama työskentelytapa ei sovi joka projektille. Projektin edetessä tiimi tulee tietoiseksi siitä, mitkä käytänteet nopeuttavat, ja mitkä hidastavat heitä saavuttamasta maalia. Kukin käytänne on testi, jolla pyritään parantamaan tehokkuutta, kommunikaatiota,

luottamusta sekä tuottavuutta. Projektia ohjailemalla koko tiimi löytää juuri heille sopivan työskentelytavan.

Tässä luvussa esitellään XP:n arvot ja käytänteet. Jäljempänä kerrotaan kuinka Beckin viisi vuotta myöhemmin esittelemän uuden XP:n käytänteet eroavat alkuperäisestä.

3.1 XP:n ydinarvot

XP:llä on viisi ydinarvoa: kommunikaatio, yksinkertaisuus, rohkeus, palaute ja kunnioitus, joista viimeinen tuli mukaan vasta Beckin (2004) esittelemässä uudessa XP:ssä. Extreme Programming sivusto esittelee arvot seuraavasti:

XP ohjelmoijat kommunikoivat tiiviisti asiakkaiden ja työkavereidensa kanssa, jolloin turha dokumentaatio voidaan korvata viestimällä kasvokkain. XP - ohjelmoijat pitävät järjestelmän suunnitelman yksinkertaisena. He lähtevät rohkeasti toteuttamaan järjestelmää vaikkei kaikkia vaatimuksia olisikaan vielä tarkennettu. He saavat palautetta testaamalla järjestelmää jo ensimmäisestä päivästä lähtien. He toimittavat järjestelmän asiakkaalle niin varhaisessa vaiheessa kuin mahdollista, ja tekevät siihen asiakkaan pyytämät muutokset. Näin he voivat näin vastata muuttuviin asiakasvaatimuksiin ja teknologiaan. (Wells 2001)

3.1.1 Kommunikaatio

Kommunikaatio on XP:n käytänteiden perusedellytys. Useimmat ongelmat ja virheet ohjelmistoprojekteissa aiheutuvat kommunikaation puutteesta. Toisinaan asiakkaalta ei osata kysyä oikeita kysymyksiä tai kehitystiimi ei ymmärrä asiakkaan kertomuksia oikein. Joskus ohjelmoijat eivät muista kertoa kriittisistä järjestelmänmuutoksista toisilleen, jolloin niitä ei osata ottaa

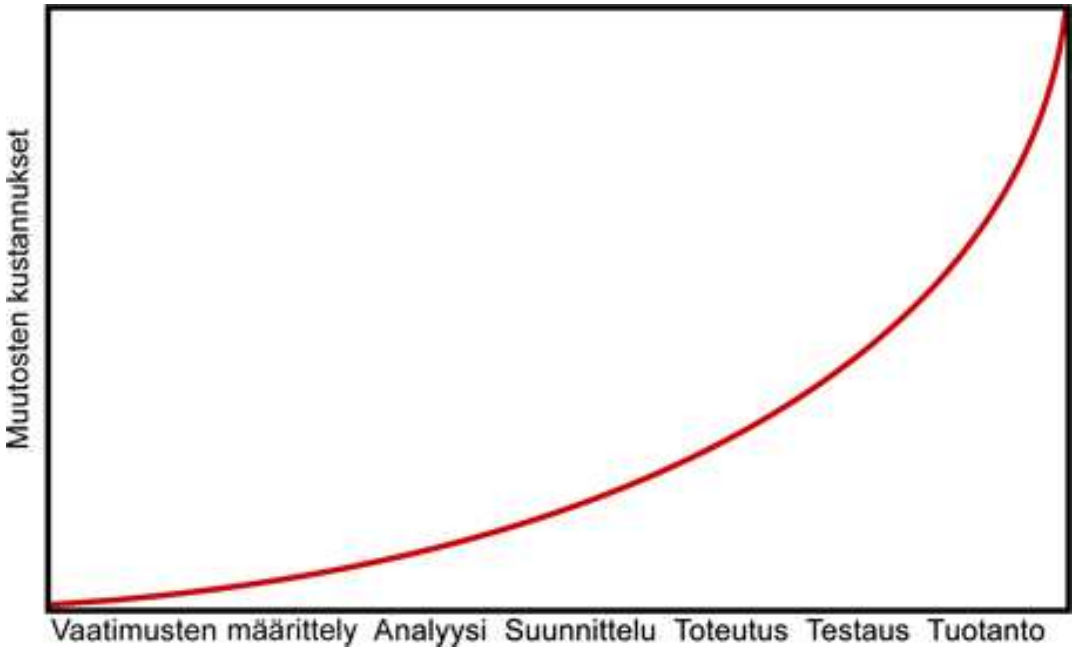
huomioon. Myös yritysjohdon ja kehitystiimin välisessä kommunikaatiossa voi olla puutteita, jolloin projektin etenemisestä voi muodostua väärä kuva. (Beck 1999)

XP pyrkii maksimoimaan kommunikaation tiimin keskuudessa sekä asiakkaan ja tiimin välillä. Tiimin keskuudessa kommunikaatio korostuu pariohjelmoinnissa, jolloin yhdellä tietokoneella istuu kaksi henkilöä: kun toinen kirjoittaa koodia, toinen miettii samalla koodin rakennetta, testitapauksia ja antaa partnerille kehitysideoita. Kommunikaatio asiakkaan ja tiimin välillä toteutetaan tuomalla asiakkaan edustaja kehitystiimin jäseneksi. Tällöin asiakas päättää mitkä ominaisuudet ovat tärkeimpiä, ja on lisäksi aina paikalla vastaamassa kehitystiimin kysymyksiin. (Burke 2003)

3.1.2 Yksinkertaisuus

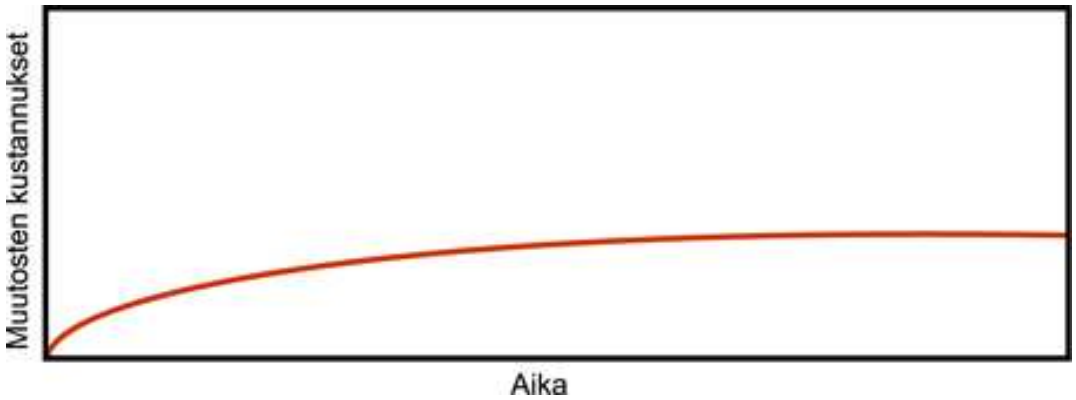
Yksinkertaisuus on tärkeimpiä XP:n ydinajatuksista. Yksinkertaisuuden omaksuminen vaikuttaa syvällisesti kykyyn omaksua koko XP -menetelmä. Pitämällä suunnitelmat yksinkertaisina vähenee riski siitä, että suunniteltaisiin monimutkaisia ohjelmistorunkoja, joita asiakas ei lopulta edes tarvitse. Yksinkertaisuus mahdollistaa koodin muokkaamisen muuttuvien asiakasvaatimuksien mukaan. Tärkeintä on löytää yksinkertainen ratkaisu sen hetkiseen ongelmaan sen sijaan, että pyrittäisiin toteuttamaan kaikki mahdolliset tulevaisuuden tarpeet. (Burke 2003)

Ohjelmistotuotannon yleisen väittämän mukaan muutoksien hinta nousee eksponentiaalisesti projektin edetessä. Kuviossa 2 on kuvattu kyseinen skenaario. Tilanne johtuu usein siitä, että koodimäärän kasvaessa muutos yhteen paikkaan vaikuttaa myös lukuisiin muihin ohjelmiston osiin.



Kuvio 2

Beckin (1999) mukaan XP -projekteissa tiedostetaan ajan vaikutus muutoksien hintaan, mutta kustannukset eivät välttämättä nouse räjähdysmäisesti. Kuvion 3 mukaiseen tilanteeseen päästään yksinkertaisella suunnittelulla, automaattisilla testeillä sekä koodin jatkuvalla refaktoroinnilla. Refaktorointi käydään läpi käytänteet osioissa, mutta lyhyesti se tarkoittaa koodin siistimistä, selkeyttämistä sekä turhien rakenteiden poistamista.



Kuvio 3.

3.1.3 Palaute

Myöhemmin esiteltävät käytänteet toimivat sillä edellytyksellä, että kehitettävästä järjestelmästä saadaan palautetta paljon ja usein. Käytänteet kuten pienet julkaisut, jatkuva integrointi ja testaus antavat hyvin selkeää palautetta kehitettävästä järjestelmästä. Palaute mahdollistaa projektin etenemisen varmoin askelin. (Newkirk 2001)

Konkreettinen palaute on korvaamatonta, sillä turha optimismi on ohjelmistoalalla turmiollista. Palautejärjestelmä toimii monella eri tasolla. Ohjelmoijat kirjoittavat yksikkötestit kaikille toiminnallisuuksille, jotka mahdollisesti voisivat toimia virheellisesti. Testejä kirjoittamalla ja niitä ajamalla ohjelmoijan on keskityttävä minuutti minuutilta oman koodinsa tasoon. Kun asiakas kirjoittaa käyttäjätarinoita ohjelmiston ominaisuuksista, ohjelmoijat arvioivat ne heti, jotta asiakas saisi konkreettista palautetta tarinoidensa laadusta. Palaute toimii viikko- ja kuukausitasolla, kun järjestelmästä annetaan pieniä julkaisuja asiakkaalle testattavaksi. Julkaisut antavat konkreettistaa palautetta kehitettävän järjestelmän tilasta, ja kertovat asiakkaalle, onko järjestelmän kehitys menossa oikeaan suuntaan. Järjestelmä laitetaan tuotantoon heti kun se on mahdollista, ja näin saadaan aikaista palautetta järjestelmän toiminnasta käytännössä. (Beck 1999)

3.1.4 Rohkeus

Monet perinteiset prosessinhallintamenetelmät perustuvat pitkälle laadittuihin etukäteissuunnitelmiin, joilla yritetään varmistaa projektin onnistuminen. Newkirkin (2001) mukaan rohkeus tarkoittaa sitä, että uskalletaan lopettaa suunnittelutyö ajoissa, ja aloittaa toteuttaminen, jonka kautta järjestelmän vaatimukset tarkentuvat.

Beckin (1999) mukaan vaatii myös rohkeutta tehdä suuria arkkitehtuurisia muutoksia suhteellisen valmiiseen ohjelmistoon tai heittää monimutkaisesti toteutettu koodi pois, vaikka se toimisi, ja koodata se uudestaan – tällä kertaa yksinkertaisesti.

Kunnioitus

Uudessa versiossa XP:stä mukaan tuli uusi ydinarvo: kunnioitus. Jos tiimin jäsenet eivät välitä toisistaan tai toistensa työstä, prosessimenetelmä ei voi toimia. Jos tiimin jäsenet eivät välitä projektista, mikään ei voi sitä pelastaa. Jokaisella tiimin jäsenellä on yhtä suuri arvo. Jokaista tiimin jäsentä ja projektin viiteryhmiä, joita ohjelmiston kehittäminen koskettaa, täytyy kunnioittaa. (Beck 2004)

3.2 XP -ohjelmoinnin käytänteet

XP:n arvot konkretisoituvat käytänteiden muodossa. Käytänteet eivät ole ohjelmistoalalla uusia, vaan XP pyrkii yhdistelemään hyväksi havaittuja menetelmiä toimivaksi kokonaisuudeksi, jossa eri osa-alueet tukevat toisiaan. Toisien käytänteiden vahvuuden paikkaavat toisien heikkouksia. (Beck 1999)

3.2.1 Suunnittelupeli

Jokainen ohjelmistoprojekti – oli se iso tai pieni – täytyy olla jollakin tavoin suunniteltu. Suunnittelu tarjoaa osapuolille yleisen käsityksen siitä, mitä ohjelmisto tulee sisältämään, kuinka paljon resursseja sen toteuttaminen vaatii sekä mitä riskejä kehityksen aikana mahdollisesti tulee vastaan.

Suunnittelu XP:ssä pyritään pitämään mahdollisimman yksinkertaisena ja yritetään päästä toteuttamaan järjestelmää nopeasti. Kun järjestelmää päästään

toteuttamaan nopeasti, asiakkaat näkevät prototyyppimäisen lähestymistavan ansiosta ovatko suunnitelmat juuri sitä, mitä he oikeasti haluavat. (Astels 2002)

Järjestelmän toiminnallisuuden määritteleminen varsinkin kokemattomalle asiakkaalle on monesti haastava tehtävä. XP:n suunnittelupeli tarjoaa yksinkertaisen tavan kuvata toiminnallisuus käyttäjätarinoiden muodossa. (Astels 2002) Käyttäjätarinat ovat yksinkertaisia korteille kirjoitettuja kuvauksia järjestelmän toiminnoista, ja ne on tarkoitus heittää pois käytön jälkeen (Beck 1999). Käyttäjätarinoiden lisäksi suuren osan suunnitteluun antavat keskustelut, joita korteissa olevien asioiden tiimoilta käydään. Kortit, joihin tarinat ovat kirjoitettu, voidaan ripustaa seinälle, josta työntekijät ne havaitsevat. Sanallisen tai graafisen kuvauksen lisäksi tarinoille annetaan lyhyet nimet. Jos tarinoista halutaan tiedottaa organisaation muille osille, voidaan ne ajoittain kääntää formaattisempaan muotoon. Yksi XP -tyylisen suunnittelun ominaisuuksista on se, että tarinat analysoidaan hyvin aikaisessa vaiheessa. Aikainen analysointi pistää kaikki osapuolet pohtimaan, kuinka tarinoista saataisiin eniten vastinetta pienimmällä mahdollisella panostuksella. Yksittäisen tarinan analysointi tapahtuu arvioimalla kuinka paljon työtä sen toteuttaminen vaatii, mikä on sen liiketoiminta-arvo sekä mikä on sen prioriteetti. Jos tarina havaitaan liian laajaksi, se voidaan puolittaa kahdeksi tai jos tarina havaitaan liiketoiminta-arvoltaan tärkeäksi, sitä voidaan laajentaa. (Beck 2004)

Beckin (1999) mukaan suunnittelupelissä on kolme vaihetta: tutkimus- (exploration), sitoutumis- (commitment) ja ohjaamisvaihe (steering). Suunnittelupelissä asiakas ja kehitystiimi käyvät eräänlaista dialogia, jossa asiakas tekee päätökset liittyen järjestelmän liiketoiminnallisuuteen, kun taas tuotantotiimi vastaa teknisistä päätöksistä.

Tutkimusvaiheen tarkoituksena on antaa molemmille osapuolille ymmärrys järjestelmän toiminnallisuudesta. Asiakas kuvaa käyttäjätarinoin mitä järjestelmän tulisi tehdä. Kehitystiimi arvioi kuinka kauan käyttäjätarinan toteuttamiseen menee aikaa. Jos kehitystiimi ei voi käyttäjätarinan perusteella arvioida tuottamiseen kuluvaa aikaa, asiakasta pyydetään jakamaan käyttäjätarina kahteen tai useampaan osaan. Asiakas voi jakaa käyttäjätarinan kahtia myös oma-aloitteisesti, mikäli huomaa jonkin osan tarinasta olevan tärkeämpi kuin toinen (Beck 1999).

Sitoutumisvaiheessa asiakas määrittelee projektin laajuuden eli kuinka laaja järjestelmän tulee olla, jotta se toisi liiketoiminta-arvoa. Asiakas kertoo myös milloin järjestelmän viimeistään tulisi olla valmis. Asiakas lajittelee käyttötapaukset kolmeen pinoon: (1) Toiminnot, joita ilman järjestelmä ei toimi. (2) Toiminnot, jotka eivät ole niin tärkeitä, mutta tuottavat huomattavaa liiketoiminta-arvoa. (3) Toiminnot, jotka olisi mukava saada. Kehitystiimi lajittelee käyttäjätarinat niiden riskin mukaan kolmeen pinoon. (1) Tarinat, jotka voidaan arvioida sellaisenaan. (2) Tarinat, jotka voidaan arvioida melko hyvin. (3) Tarinat joita ei voi arvioida lainkaan. Kehitystiimi arvioi kuinka nopeasti kukin vaatimus voidaan toteuttaa, jonka jälkeen asiakas voi valita, mitkä vaatimukset he haluavat mukaan järjestelmään huomioiden itse asettamansa julkaisupäivämäärän. (Beck 1999).

Ohjausvaiheessa suunnitellaan mitä käyttäjätarinoita halutaan kuhunkin iteraatioon. Xp:ssä iteraation pituus on yhdestä kolmeen viikkoon. Ensimmäiseen iteraatioon valittavien asioiden täytyy olla sellaisia, että järjestelmästä saadaan järkevä alkioasteella oleva kokonaisuus. Ohjausvaiheessa asiakas voi kirjoittaa uusia käyttäjätarinoita, mikäli huomaa uusia tarpeita. Tuotantotiimi voi tarkentaa käyttäjätarinoille asetettua työmäärää, mikäli huomaa aikaisempien arvioiden heittävän. Tällöin

asiakkaalta täytyy kysyä mitä ominaisuuksia voidaan karsia, jotta aikataulussa pysyttäisiin. (Beck 1999).

3.2.2 Yksinkertainen suunnitelma

Suunnittelun avainsanana XP:ssä on yksinkertaisuus. Suunnitellaan siis vain se, mikä juuri sillä hetkellä on välttämätöntä. Silloinkin tehdään yksinkertaisin asia mikä vain voi toimia. Tapaa, jossa suunnitelmia lykätään niin pitkälle, kuin mahdollista, kutsutaan Astelsin mukaan oikea-aikaiseksi suunnitteluksi. XP lähtee oletuksesta, että suunnitelma on aina epätarkka ja mitä pidemmälle tulevaisuuteen suunnitellaan, sitä epätarkemmaksi suunnitelma käy. Oikea-aikaisella suunnittelulla vältytään turhalta työltä, kun jo etukäteen suunniteltuja ominaisuuksia ei jää toteuttamatta. Väittämä, että XP:ssä ei tehdä suunnittelua, ei ole totta, koska XP:ssä suunnittelu on jatkuvaa. (Astels 2002)

Vaikka XP -ohjelmoinnissa pyritään välttämään liikaa suunnittelua, se ei sulje pois mahdollisuutta käyttää esimerkiksi UML -kieltä (Unified Modelling Language) mallintamiseen. Tällöin täytyy vain muistaa, että kaavioiden piirtely palvelee tarkoitusta. Kaavioilla pyritään ensisijaisesti parantamaan kommunikaatiota muiden kehittäjien kanssa. Tehokas kommunikaatio mallintamisen osalta tarkoittaa sitä, että valitaan mallinnettavaksi vain tärkeimmät asiat. Ei piirretä diagrammeja jokaisesta luokasta – vain tärkeimmistä. Ei piirretä jokaisesta metodista sekvenssikaavioita – vain tärkeimmistä jne. Tällöin diagrammit ovat myös helppo pitää ajan tasalla. (Fowler 2001)

Beck (1999) antaa neljä kriteeriä yksinkertaisuuteen:

- Järjestelmä suoriutuu kaikista testeistä
- Järjestelmä ilmaisee selkeästi kaikki ideat, jotka ovat tarpeellisia

- Järjestelmä ei sisällä toistoa
- Järjestelmä sisältää mahdollisimman vähän luokkia ja metodeja

Käytännössä yllä olevat tarkoittavat (Jeffries 2000) mukaan seuraavaa:

- Tärkeintä on, että koodi tekee sen, mitä sen on suunniteltu tekevän. XP:ssä tästä varmistutaan vain yhdellä tavalla – testitapauksilla.
- Koodia lukiessa on helppo löytää ideat koodin taustalta, kun se ilmaisee selkeästi kaikki tarpeelliset toiminnot.
- Kun koodissa ei ole toistoa, muutoksia ei tarvitse tehdä kuin yhteen paikkaan.
- Karsimalla koodista aika ajoin turhat luokat ja metodit pois, siihen ei jää turhaa ylimäärää.

Martin Fowler neuvoo artikkelissaan ”Is Design Dead”, ettei kannata turhaan uhrata kaikkea kapasiteettiaan pohtimalla onko suunnitelmat tarpeeksi yksinkertaista. Sen sijaan halu refaktoroida suunnitelmia yksinkertaisemmaksi on paljon tärkeämpää silloin, kun koodissa ilmenee turhaa kompleksisuutta.

3.2.3 Refaktorointi

Refaktorointi on järjestelmän rakenteen parantamista ilman, että sen toiminnallisuus muuttuu. Refaktorointia tulee suorittaa aina, kun sen havaitaan olevan tarpeellista. (Astels 2002) Yksikkötestejä laadittaessa huomaa usein, että oman koodin käyttäminen on monimutkaista, ja asian voisi suorittaa yksinkertaisemminkin. Tällöin koodin rakennetta tulee selkeyttää. Refaktorointia tulee myös suorittaa, mikäli koodissa havaitaan duplikaattisia rakenteita. Refaktorointia suoritetaan koko projektin elinkaaren ajan. Yleisin syy siihen, miksi ohjelmoijat pelkäävät toimivan koodin muuntelua on se, että he pelkäävät rikkovansa sen (Jeffries 2000). Mikäli refaktoroinnin seurauksena jokin muu järjestelmän osa vioittuu, kertovat testit siitä heti (Beck 1999). Jos

huomataan, että refaktoroinnin tulos ei miellytä, voi refaktorointiaskeleen myös peruuttaa versionhallinnan avulla. (Jeffries 2000)

3.2.4 Pienet julkisukset

Jos asiakas tietäisi projektin alussa tarkalleen mitä hän ohjelmistolta haluaa, ja ohjelmaa toteutettaessa haluaisi vielä samaa asiaa, se olisi ehkä ensimmäinen kerta ohjelmistoalan historiassa. Yleensä projektin edetessä projektia täytyy ohjailla asiakkaan muuttuvien vaatimuksien myötä. Yksi tärkeimmistä asioista mitä XP -projektissa voi tehdä, on julkaista ohjelmistosta versioita aikaisessa vaiheessa - paljon ja usein. Pienet julkistukset antavat mahdollisuuden saada tietää, mitä asiakas oikeasti haluaa. (Jeffries 2000)

Keskeneräisestä demoversiosta, jossa mikään ei toimi, julkistusta on turha tehdä. Sen sijaan jokin kokonaisuus tehdään loppuun asti siten, että se on järkevää julkaista. Suunnittelu julkaisu kerrallaan on helpompaa, kuin suunnitella koko projekti kerralla. (Astels 2002) Julkaisut suunnitellaan siten, että tärkeimmät toiminnot toteutetaan ensimmäiseksi, jolloin aikataulun lipsuessa julkaisusta jäävät vain vähemmän tärkeimmät ominaisuudet pois. Voi olla vaikea kuvitella, että tuote annettaisiin asiakkaalle ennen kuin se on täysin valmis. Asiakas on kuitenkin vain tyytyväinen, että saa konkreettisia todisteita projektin etenemisestä. Lisäksi projektiryhmä saa asiakaspalautteen kautta tietää onko projekti kulkemassa oikeaan suuntaan. (Beck 1999)

3.2.5 Pariohjelmointi

Jokainen koodirivi XP -projektissa kirjoitetaan pareittain. XP:ssä oletetaan, että kun kaksi ohjelmoijaa työskentelee yhdessä, he tuottavat enemmän, virheettömämpää ja parempilaatuista koodia, kuin että he työskentelisivät

erillään (Jeffries 2000). Pariohjelmoinnissa yhdellä työpisteellä työskentelee kaksi henkilöä. Toinen pitää hallussaan näppäimistöä ja hiirtä ja kirjoittaa koodia. Partneri seuraa vierestä, tekee aktiivisesti huomautuksia syntaksivirheistä ja antaa ohjelmoijalle ideoita ja ehdotuksia. (Astels 2002) Pariohjelmointi on siis kahden henkilön välistä yhtäjaksoista dialogia, jonka aikana koodia analysoidaan, suunnitellaan sekä testataan (Beck 2004).

Beckin (2004) mukaan pariohjelmoijat:

- Pitävät toisensa kiinni käsillä olevassa tehtävässä.
- Käyvät toistensa kanssa ideapalavereja järjestelmän parantamisesta
- Selventävät ideoita.
- Pitävät toisensa velvollisina noudattamaan tiimin käytänteitä.
- Auttavat, kun partneri jää jumiin.

Viimeinen kohta vähentää merkittävästi turhautumista tilanteissa, joissa ohjelmoija on tullut sokeaksi omille virheilleen.

Parit eivät ole kiinteitä vaan parien kokoonpanot muuttuvat. Henkilö siirtyy siis tietyin väliajoin parista toiseen. Tämä parantaa tiedon leviämistä projektiryhmän kesken. (Beck 1999)

Pariohjelmointi ei kiellä yksin ajattelua. Mikäli haluaa hioa ideaa yksin, voi sen tehdä. Prototyypin voi muotoilla pseudokoodina ja tuoda sen sitten parin nähtäväksi. Parin kanssa se on helppo ja nopea toteuttaa. (Beck 2004)

3.2.6 Asiakas tiimin jäsenenä

XP -tiimi pyrkii edistymään projektissa niin nopeasti kuin mahdollista. XP:n ydinarvo yksinkertaisuus tukee tätä tavoitetta. Määrittelyjen yksityiskohtainen kirjoittaminen vie pitkän ajan, ja dokumentaatiot eivät kommunikoi kovinkaan hyvin. On paljon tehokkaampaa ottaa asiakas tiimin jäseneksi, ja käydä keskusteluja suoraan asiakkaan kanssa siitä, mitä projekti tarvitsee. Asiakkaan edustaja työskentelee siis samoissa tiloissa kuin ohjelmoijat. Tällöin järjestelmää lähdetään toteuttamaan yksinkertaisien käyttäjätarinoiden pohjalta, joita tarkennetaan projektin edetessä. Mikäli asiakas ei olisi ohjelmoijien kanssa samoissa tiloissa, käyttäjätarinoiden tarkennuksia olisi huomattavasti hitaampaa selvittää. Ohjelmoijat joutuisivat tekemään päätöksiä käsillä olevasta tehtävästä arvailujen varassa, ja pystyisivät varmistamaan asian vasta myöhemmin. Turhan työn sekä odottelun määrä kasvaa tällöin huomattavasti. Asiakas tiimin jäsenenä luo myös siteen kehitystiimin ja asiakkaan välillä, ja lähentää usein organisaatioiden suhteita. (Jeffries 2000)

3.2.7 Testaus

Testaus kuuluu hyvin olennaisena osana XP -menetelmään. Testeihin panostamalla pyritään vähentämään järjestelmän virhemäärää, ja nostamaan kehittäjien luottamusta omaan koodiin. Testaus koostuu testien kirjoittamisesta ennen koodia, automaatiotestauksesta ja asiakastesteistä

Testauslähtöinen kehitys

Yksikkötestit kirjoitetaan ennen varsinaista koodia. Yksikkötestit varmistavat, että jokin tietty toiminnallisuus toimii, kuten sen on oletettu toimivan. (Astels 2002) XP:ssä toimintoa ei voi pitää valmiina ennen kuin sille tehdyt testitapaukset menevät läpi (Beck 1999). Koodaus XP –menetelmän mukaisesti

etenee seuraavasti: kirjoita testi, kirjoita sitä vastaava ohjelma. Tätä prosessia jatketaan, kunnes koko toiminto on valmis.

Testien kirjoittamisella ennen koodia saavutetaan monenlaisia etuja. Ohjelmoija pystyy määrittelemään mielessään kuinka toiminnon tulisi toimia, ennen kuin hän rupeaa sitä työstämään. Työstettävä toiminto on joka tapauksessa määriteltävä, joko ohjelmoijan päässä tai paperilla, joten miksi määrittelyä ei voisi hoitaa samalla syntaksilla, kun itse koodi kirjoitetaan. Näin testaus saadaan osaksi suunnittelutoimintaa, kun perinteisellä menetelmällä se olisi ylimääräistä työtä vasta toteutuksen jälkeen. Kun testit menevät läpi, ohjelmoija voi luottaa, että hänen koodinsa toimii ja siirtyä seuraavaan tehtävään. (Astels 2002) Epävarmuuden vähentymisellä on suora vaikutus työssä viihtymiseen. Usein kiireisessä projektissa kädet ovat täynnä töitä, ja ennen kuin yhtä tehtävää on saanut valmiiksi, toista ollaan jo työntämässä sisään. Tällöin tuotantoon pääsee usein liian paljon testaamatonta koodia, jonka virheet ajatellaan löytyvän virheraporttien muodossa. Tämä on kuitenkin kallis tapa ja vaikuttaa myös negatiivisesti tuotantoyhtiön maineeseen. Tältäkin skenaariolta vältyttäisiin kirjoittamalla testit ensin.

Automaatiotestaus

Jotta testejä olisi helppo ajaa helposti milloin tahansa, testit on automatisoitava. Java maailmassa testien automatisointi onnistuu esimerkiksi JUnit ohjelmistolla. Koodin kattavat yksikkötestit, jotka ovat automatisoitu, mahdollistavat rohkean koodin refaktoroinnin: jos jotain menee rikki, testitapaukset kertovat siitä heti. Myös jatkuva integrointi helpottuu huomattavasti, kun testit kertovat, ovatko ohjelman toisessa osassa tehdyt muutokset aiheuttaneet virheitä ohjelman muissa osissa.

Asiakastestit

Asiakas testaa kunkin iteraation päätteeksi tehdyt julkistukset asiakastestein. Näin varmistutaan siitä, että järjestelmä vastaa käyttäjätarinoissa kuvattua toiminnallisuutta. Testauksen yhteydessä asiakas voi todeta, oliko määritelty toiminto käytännössä juuri sitä, mitä he halusivat.

3.2.8 Jatkuva integrointi

Järjestelmän integrointi tulisi suorittaa vähintään kerran päivässä, mutta mielellään useammin. Perussääntönä voi pitää, että integrointi suoritetaan aina, kun on saanut valmiiksi yhden tehtävän. Automaattiset testit on syytä ajaa aina integroinnin päätteeksi. Jatkuva integrointi pitää myös muun kehittäjätiimin ajan tasalla järjestelmän tilasta. (Newkirk 2001)

3.2.9 Koodin yhteisomistus

Perinteisessä ohjelmistokehityksessä vain yksi henkilö tekee muokkauksia kuhunkin koodilohkoon, ja vain hänellä on erikoistietämys kyseiseen toiminnallisuuteen. Tällöin projekti on vaikeuksissa, jos projektiryhmän jäsen sairastuu tai jää esimerkiksi isyyslomalle. XP pyrkii siihen, että jokainen ohjelmoija on jollakin tasolla perillä jokaisesta ohjelmiston osasta. Henkilö, joka näkee kehitystarpeen tietyssä koodiosiossa, tekee muutoksen itse sen sijaan, että lähettää parannusehdotuksen koodin alkuperäiselle tekijälle. Koodin yhteisomistus ei toimi ilman hyvää kommunikaatiota, kunnollista versionhallintaohjelmaa ja jatkuvaa integrointia. Järjestelmä on siis integroitava vähintään päivittäin, jotteivät tehdyt muutokset aiheuta ristiriitoja muissa ohjelmiston osissa. Koodin yhteisomistuksen vuoksi koodikatselmointia tulee suoritettua väkisinkin, kun joutuu tutkimaan työkaverin tekemää koodia.

3.2.10 Kestävä työtahti

XP –ohjelmoinnissa ei ole kiellettyä tehdä ylitöitä, mutta niitä ei tulisi tehdä useana viikkona peräkkäin (Astels 2002). Jatkuvaa ylityötä tulisi välttää, jotta työntekijät voisivat käyttää aikaa myös yksityiselämäänsä. Itsensä ylirasittaminen tänään pilaa usein seuraavat työpäivät. Väsyneenä tulee helpommin tehtyä virheitä ja suunnitteluratkaisuja, jotka kostautuvat projektin edetessä moninkertaisesti. Ohjelmistokehityksessä tärkeintä on oivaltamiskyky, ja parhaat oivallukset tulevat hyvin levänneiltä ja rentoutuneilta aivoilta. (Beck 2004)

Sairaana on parempi jäädä kotiin parantamaan itsensä, ja palata terveenä takaisin energiseen työhön. Sairaana töihin tuleminen on myös tartuntavaaran takia iso riski projektille. (Beck 2004)

Organisoimalla omaa työntekoaan etukäteen on mahdollista parantaa oman työnsä tuottavuutta ilman tarvetta ylitöille. Päivästä voi esimerkiksi varata muutaman tunnin pelkästään ohjelmoinnille, jonka ajaksi sulkee sähköpostin puhelimen sekä pikaviestimet. (Beck 2004)

3.2.11 Ohjelmointistandardit

Yhteistyöntukemiseksi tulee ottaa käyttöön tiimin laajuiset ohjelmointikäytänteet, joita kaikki noudattavat (Astels 2002). Koodin luku helpottuu huomattavasti silloin, kun ohjelmoijat kirjoittavat koodin saman standardin mukaisesti. Standardi kannattaa valita käytettävän ohjelmointikielen mukaisesti. Javan tapauksessa voi käyttää esimerkiksi Sunin laatimia suosituksia. Koodin kommentoinnista, sisennyskäytänteistä sekä koodin lukemista selkeyttävien rivinvaihtojen määrästä kannattaa sopia tiimin kesken.

3.2.12 Vertauskuva

Vertauskuvan eli metaforan on tarkoitus tarjota kokonaiskuva kehitettävästä järjestelmästä ja siitä, mitä sen on tarkoitus tehdä. Vertauskuvan avulla mahdollisesti monimutkainen kohdealue selitetään yleisellä kielellä siten, että kaikki osapuolet ymmärtävät mistä on kyse.

4 UUSI XP

Viisi vuotta ensimmäisen kirjansa julkaisemisen jälkeen Kent Beck julkaisi yhdessä Cynthia Andresin kanssa uuden version XP:stä kirjassaan Extreme Programming Explained Second Edition (2004). Painos ei ole vain ensimmäisen kirjan uudelleenlämmittely, vaan se uudistaa XP:n lähes täysin. Ensimmäinen kirja perustui neljälle ydinarvolle ja neljälletoista käytänteelle, eikä ohjelmistokehitys ollut XP:n mukaista, jos ei omaksunut niitä kaikkia. Uudessa XP:ssä käytänteet on jaoteltu ensi- ja toissijaisiin käytänteisiin, joista jälkimmäisiä ei tule ottaa käyttöön ennen kuin ensisijaiset on omaksuttu. Uusi XP ei suosittele kaikkien käytänteiden käyttöönottoa heti. Sen sijaan ensisijaisista käytänteistä valitaan organisaatiolle sopivimmat, ja ne voidaan implementoida vaiheittaisesti esimerkiksi ohjelmoimalla vain osan työajasta pareittain. Ensisijaisten käytänteiden omaksumisen jälkeen voidaan harkita, mitä toissijaisista käytänteistä otettaisiin käyttöön.

Kaksi vanhoista käytänteistä - ohjelmointistandardit ja vertauskuva ovat kokonaan hylätty. Lisäksi arvoihin on tullut yksi lisää: kunnioitus. Tässä luvussa esittelen XP:n uudet käytänteet. Luku pohjautuu täysin Beckin uuteen kirjaan ellei toisin mainita.

4.1 Ensisijaiset käytänteet

Kun XP -menetelmää ollaan ottamassa organisaatioon käyttöön, on turvallista aloittaa implementoimalla ensisijaiset käytänteet ennen toissijaisia käytänteitä. Esittelen käytänteistä vain ne, jotka eroavat huomattavasti vanhasta.

Viikoittainen sykli

Ohjelmistokehitystä suoritetaan viikon sykleissä. Jokaisen viikon alussa pidetään kokous, jossa asiakas päättää, mitkä käyttäjätarinat kyseisenä viikkona toteutetaan. Kehitystiimi pilkkoo tarpeen vaatiessa valitut tarinat tehtäviksi, joille voidaan valita yksittäiset vastuuhenkilöt. Viikko aloitetaan kirjoittamalla tehtäville automatisoidut testit, jotka ajetaan, kun tarinat ovat toteutettu.

Vuosineljänneksen sykli

Projektit täytyy suunnitella myös pidemmissä ajanjaksoissa. Vuosineljänneksen suunnittelu antaa tiimille, projektille ja sen kehitykselle suuntaviivat ja pidemmän tähtäimen tavoitteet. Vuosineljänneksen suunnittelukokouksessa käydään läpi seuraavia asioita:

- Tunnistetaan kehityksen pullonkaulat – erityisesti sellaiset, jotka ovat tiimistä johtumattomia.
- Vuosineljännekselle valitaan teema, joka ohjaa kehitystyötä. Teemat auttavat pitämään päivittäisessä työssä mielessä, mitä järjestelmällä pyritään saamaan aikaan.
- Pohditaan miten projekti istuu organisaatioon.

Verkkaisuus

Tiimin suorituskykyä ei tule yliarvioida. Lupauksia, joita ei voida täyttää, tulee välttää. Jokaisessa ohjelmistoprojektissa ilmenee odottamattomia ongelmia, jotka venyttävät aikataulua. Näin ollen työmääräarvioihin tulee jättää turvamarginaali. Mikäli asiat saadaan tehtyä nopeammin kuin arvioitiin, voidaan aina toimittaa enemmän kuin luvattiin.

Inkrementaalinen suunnittelu

XP vastustaa ajatusta koko järjestelmän suunnittelusta kerralla. Sen sijaan XP pyrkii saamaan nopeasti aikaan jotain konkreettista, jonka avulla suunnitelmia voidaan tarkentaa. Inkrementaalisen suunnittelun tulisi tapahtua koodauksen ohessa.

Kokonainen tiimi

Tiimin tulee koostua ihmisistä, joilla on kaikki tietotaito, joka vaaditaan projektin onnistumiselle. Tiimillä tulisi olla vahva yhteenkuuluvuudentunne sekä halu auttaa toisiansa. Tiimin kokoonpano ei ole staattinen. Jos tietyt tiedot, taidot tai asenteet muodostuvat projektin kannalta tärkeiksi, tiimiin kannattaa ottaa taidot omaava jäsen. Jos taas jonkun jäsenen erityistietämystä ei enää tarvita, hänet voidaan siirtää muihin tehtäviin. Projektissa voi esimerkiksi olla jaksoja, joissa kaivataan erityistä tietokantaosaamista. Tällöin kaivataan väliaikaisesti asiaan erikoistuneen henkilön tietämystä.

Kahdentoista henkilön kokoiset tiimit voivat vielä kommunikoida toistensa kanssa tehokkaasti. Sadan henkilön kokoluokassa kaikkien henkilöiden tunteminen on jo vaikeaa, eikä tällöin saavuteta riittävää luottamustasoa. Jotkut organisaatiot yrittävät pitää tiimejä, joissa ihmiset työskentelevät useassa projektissa samaan aikaan. Tällöin suuri osa ohjelmoijien ajasta kuluu

siirtymisestä tehtävästä toiseen. Parempi vaihtoehto on erottaa tiimi kahdeksi, joista kukin keskittyy vain yhden asiakkaan asioiden hoitoon. Tällöin työntekijät vapautuvat pirstoutuneesta ajattelutavasta ja asiakkaat voivat luottaa siihen, että heidän asioitaan hoidetaan alituisesti. Kun tiimi työskentelee vain yhden asiakkaan kanssa, ja sen kokoonpano ei vaihtele, tiimille syntyy tunne, että teemme tätä yhdessä.

Yhteinen työtila

Kehitystiimin tulisi työskennellä tilassa, johon mahtuu koko tiimi, ja joka mahdollistaa maksimaalisen kommunikoinnin. Kun ohjelmointi tapahtuu yleisellä paikalla, ihmiset voivat kaivata myös hieman yksityisyyttä. Tätä voi tarjota erottamalla työskentelytilat sermeillä tai rajoittamalla työskentelytunteja, jolloin myös yksityiselämälle jää aikaa.

Informatiivinen työskentely-ympäristö

Työskentely-ympäristön tulisi tarjota informaatiota projektin edistymisestä, tekemättömistä töistä sekä korjaamattomista virheistä. Kiinnostuneen havainnoijan tulisi saada nopeasti kuva siitä, miten projekti etenee. Katsomalla lähemmin hänen tulisi saada lisäinformaatiota mahdollisista ongelmakohdista.

Informatiivisen työskentely-ympäristön voi toteuttaa kiinnittämällä käyttäjätarinat seinälle. Tarinat voi luokitella, jotta halutun informaation löytäminen olisi helppoa. Luokittelu voi tapahtua esimerkiksi järjestelemällä tarinat niiden toteuttamisasteen mukaan: toteutetut tarinat, tämän viikon tarinat, seuraavan julkaisun tarinat, arvioitavat tarinat.

Työskentely-ympäristön tulisi tarjota myös apuja muille inhimillisille tarpeille. Virvokkeet ja pikkupurtava rohkaisevat sosiaaliseen kanssakäymiseen. Puhtaat tilat antavat ajatuksille tilaa keskittyä käsillä olevaan ongelmaan. Jos

työskentelytilassa on kaavioita tai tauluja, joita ei päivitetä, ne kannattaa kaataa pois ja käyttää tila ajantasaisen informaation tarjoamiseen.

Järjestelmän nopea koostaminen

Jotta kääntäminen voitaisiin tehdä usein ja palautetta järjestelmän tilasta saataisiin paljon, järjestelmän kääntäminen ja testien ajaminen ei saisi viedä yli kymmentä minuuttia. Järjestelmän pystyttämisen tulee tapahtua automaattisesti ilman tarvetta manuaaliseen puuttumiseen. Manuaalinen järjestelmän pystyttäminen johtaa vähentyneeseen testaamiseen ja useampiin koodivirheisiin: ”Teimmekö virheen? Käännetään ja katsotaan.”

4.2 Toissijaiset käytänteet

Toissijaisia käytänteitä on vaikea tai jopa vaarallista ottaa käyttöön, ennen kuin niitä vastaavat ensisijaiset käytänteet ovat alustavasti omaksuttu. Esimerkiksi päivittäisen julkaisemisen käyttöönotto ilman, että ohjelmiston virhemäärä on saatu vähäiseksi muiden käytänteiden avulla (pariohjelmointi, jatkuvan integrointi, testaus ennen ohjelmointia), voi olla tuhoisaa.

Asteittainen päivitys

Korvattaessa vanhaa järjestelmää uudella kannattaa aloittaa korvaamalla osa ominaisuuksista heti, ja jatkaa asteittain kunnes koko järjestelmä on uusittu. ”Kaikki tai ei mitään” -asennetta kannattaa välttää.

Päivittäinen julkaiseminen

Joka yö ohjelmisto laitetaan tuotantoon. On riski, jos versiot tuotannossa ja omalla koneella eroavat liiaksi.

Sopimuksen laajuus

Liian pitkäaikaisia sopimuksia tulisi välttää. Riskiä, jossa olosuhteet muuttuvat, mutta sama sopimus on edelleen voimassa, voidaan vähentää tekemällä laajan sopimuksen sijasta kolme pienempää.

Käyttöperustainen laskutus

Yleensä ohjelmistosta saadaan kassavirtaa jokaisesta julkaisusta. Tällöin ohjelmiston toimittaja haluaa julkaista mahdollisimman monta versiota, lisäten vain sen verran ominaisuuksia, että tuotteelle voidaan antaa uusi versionumero. Varsinaiset käyttäjät taas haluaisivat kerralla paljon ominaisuuksia ja harvemmin päivityksiä, johtuen päivittämisen työläydestä. Molempia osapuolia miellyttävä laskutus saadaan aikaan, kun laskutetaan joka kerta, kun järjestelmää käytetään. Käytön mukaan määräytyvä rahavirta toimii loistavana palautteena järjestelmän tasosta, ja pitää yllä toimittajan intressiä kehittää palvelua.

Tiimien säilyttäminen

Toimivat kehitystiimit kuuluisi säilyttää samoina useammissa projekteissa. Suhteet, jotka he jakavat projektissa, ovat arvokkaita, eikä niitä kuuluisi hajottaa.

Tiimien pienennys

Parantamalla ohjelmistokehityksen laatua saadaan tiimistä vapautettua resursseja, joista voi muodostaa uusia tiimejä. Huomioitava on, että henkilöstön työtaakka pysyy samana, vaikka tiimin koko vähenisi.

Syy-seuraussuhde analyysi

Joka kerta, kun löydetään virhe, poistetaan virheen lisäksi sen aiheuttaja. Tällöin ei korjata pelkästään virhettä, vaan estetään samanlaisien virheiden syntyminen uudestaan.

Koodi ja testaus

Koodaus ja testaus ovat ainoita pysyviä tuotoksia, ja tämä käytänne täytyy säilyttää. Muut dokumentoinnit voidaan generoida niiden pohjalta.

Yksittäinen koodikanta

Järjestelmästä pidetään vain yhtä virallista versiota. Väliaikaisia haaroja järjestelmästä voi tehdä, mutta niiden elinkaari saa olla enintään muutaman tunnin mittainen.

4.3 Vanha vs. uusi

Selkeyttääkseni vanhojen ja uusien käytänteiden välistä suhdetta olen listannut ne taulukkoihin 1 ja 2. Taulukoiden vasemmassa sarakkeessa on alkuperäisen XP:n käytänteet ja oikeassa uudet. Kullakin taulukon rivillä on toisiinsa jollakin tavoin linkittyvät käytänteet. Lihavoidut käytänteet ovat ensisijaisia.

Taulukossa 1 kuvataan mitkä XP:n uusista käytänteistä linkittyvät suoraan vanhaan.

Vanha	**Uusi**
Pariohjelmointi (Pair Programming)	**Pariohjelmointi (Pair Programming)**
Asiakas tiimin jäsenenä (On-site Customer)	Asiakas tiimin jäsenenä (Real Customer Involvement)
Koodin yhteisomistus (Collective Code	Koodin yhteisomistus (Shared Code)

OwnerShip)	
Testaus (Testing)	**Testauslähtöinen kehitys (Test-First Programming)**
Kestävä työtahti (Sustainable Pace)	Energinen työ (Energized Work)
Jatkuva integrointi (Continuous Integration)	**Jatkuva integrointi (Continuous Integration)**

Taulukko 1. Suoraan vanhasta uuteen linkittyvät käytänteet

Taulukossa 2 on ryhmitelty riveittäin kirjoittajan mielestä toisiaan sivuavat käytänteet.

Vanha	**Uusi**
Suunnittelupeli (Planning Game)	**Tarinat (Stories)** **Viikottainen sykli (Weekly Cycle)** **Vuosineljänneksen sykli (Quarterly Cycle)** **Verkkaisuus (Slack)**
Pienet julkistukset (Small Releases)	**Viikottainen sykli (Weekly Cycle)** Asteittainen päivittäminen (Incremental Deployment) Päivittäinen julkaiseminen (Daily Deployment)
Yksinkertainen suunnitelma (Smple Design) Refaktorointi (Refactoring)	**Inkrementaalinen suunnittelu (Incremental Design)**
Jatkuva integrointi (Continuous Integration)	**Jatkuva integrointi Continuous Integration** **Järjestelmän nopea koostaminen (Ten-Minute Build)** Päivittäinen julkaiseminen (Daily deployment)

Taulukko 2. Toisiaan sivuavat käytänteet

Alla olevassa listassa on listattu XP:n uudet käytänteet, jotka eivät linkity vanhojen kanssa.

- Kokonainen tiimi (Whole team)

- Yksi koodikanta (Single Code Base)
- Asteittainen päivitys (Incremental Deployment)
- Sopimuksen laajuus (Negotiated Scope Contract)
- Käyttöperusteinen laskutus (Pay-Per Use)
- Yhteinen työtila (Sit together)
- Informatiivinen työskentely-ympäristö (Informative Worspace)
- Tiimin jatkuvuus (Team Continuity)
- Koodi ja testaus (Code and Tests)
- Tiimien piennennys (Shrinking Teams)

4.4 Yhteenveto uudesta XP:stä

Beckin (2004) esittelemä uusi XP eroaa huomattavasti alkuperäisestä, mutta alkuperäisistä periaatteista on kuitenkin pidetty kiinni. Uudesta kirjasta huomaa, että viiden vuoden aikana käytännön oppia on kertynyt. Monia uusia ideoita on esitelty, ja vastaavasti joistakin on luovuttu. Kun alkuperäisessä kirjassa oli 12 käytännettä, uudessa niitä on 24. Lisääntynyt käytänteiden määrä voi hankaloittaa kokonaiskuvan saamista XP – menetelmästä, mutta toisaalta ne myös täsmentävät esimerkiksi suunnitteluvaiheen toteuttamista. Lisääntynyt käytänteiden määrä on kirjoittajan mielestä hieman ketterien menetelmien periaatteiden vastainen, jossa lähdetään ajatuksesta, että vain muutama yksinkertainen säännöstö riittää. Jotkin toissijaisista käytänteistä ovat myös hieman hämmentäviä. Esimerkiksi tiimien säilyttäminen ja tiimien pienentäminen tuntuvat olevan keskenään hieman ristiriidassa. Täsmällistä kaiken ratkaisevaa ohjenuoraa ohjelmistokehityksestä uusi XP ei pyri antamaan, vaan kokoelman hyviä ideoita ja käytänteitä, joista jokainen organisaatio voi valita heille osuvimmat.

5 XP:N RAJOITTEITA

Tässä luvussa käsittelen XP:n rajoitteita. Pyrin myös arvioimaan millaisiin projekteihin XP sopii, ja millaisiin ei. Pohdin myös, kuinka XP:n periaatteita voitaisiin menestyksellisesti noudattaa ongelmakohdista huolimatta.

5.1 Arkkitehtuuri

Erillisen arkkitehtuurivaiheen puuttuminen on herättänyt kritiikkiä XP:tä kohtaan. Ohjelmiston suuntaviivat antava metafora oli tarkoitettu ohjaamaan arkkitehtuuria, mutta uudessa XP:n versiosta siitä on luovuttu. Järjestelmän arkkitehtuuria ei kiinnitetä, koska järjestelmä halutaan pitää juostavana muutoksille. XP:n arvo yksinkertaisuus vaatii pitämään arkkitehtuurin mahdollisimman yksinkertaisena, ja lisäämään järjestelmän monimutkaisuutta vain tarpeen vaatiessa (West 2002). XP – projekteissa järjestelmän arkkitehtuurisuunnittelu pyritään korvaamaan jatkuvan refaktoroinnin ja tiiviin asiakasyhteistyön avulla. Kannattaa kuitenkin harkita, onko järjestelmän arkkitehtuuri parempi suunnitella alussa hyvin, vai tehdä siihen kehityksen yhteydessä useita muutoksia.

5.2 Työmäärän arvioiminen

Aikataulujen arvioiminen XP – menetelmällä voi olla haasteellista, koska ohjelmistokokonaisuus hahmottuu vielä osittain kehityksen aikana. Näin ollen projektit, joiden aika- ja työmääräarviot on oltava tarkasti selvillä ennen projektin aloitusta, eivät sovi XP projekteille. Sopimusmenettelyt, jotka joustavat resursseissa ja aikatauluista, sopivat XP -projekteille paljon paremmin. Beck (2004) neuvookin uudessa kirjassaan jättämään työmääräarvioihin turvamarginaalin, jota voidaan käyttää silloin, kun kohdataan odottamattomia

ongelmia. Toisaalta XP:n avulla päästään järjestelmää toteuttamaan hallitusti huomattavasti pienemmällä dokumentaatiomäärällä kuin perinteisissä menetelmissä, jolloin aika voidaan käyttää määrittelyjen sijaan tuottavaan työhön. Näin ollen kehityskustannukset jäävät myös asiakkaan osalta edullisemmiksi. Hyppy tuntemattomaan vaatii vain hieman rohkeutta.

5.3 Asiakas ryhmän jäsenenä

Asiakas ei ole usein halukas antamaan yhtä työntekijöistään kokonaan projektin käyttöön, eikä se maantieteellisistä rajoituksista johtuen ole aina mahdollistakaan. Mikäli asiakas ei voi olla alati paikalla, kannattaa Jeffriesin (2000) mukaan panostaa seuraaviin seikkoihin. Asiakas olisi hyvä saada ainakin suunnittelukokouksiin, joissa keskustellaan prioriteeteista ja järjestelmän vaatimuksista. Jos asiakas ei pääse paikanpäälle, ohjelmoijat voidaan viedä asiakkaan luokse keskustelemaan ohjelmistosta, tekemään kysymyksiä ja rakentamaan luottamuksellisia asiakassuhteita, joita ei kasvottomalla sähköpostiviestinnällä koskaan saavuteta. Sähköpostiviestinnässä kannattaa varautua väärinymmärryksiin, joiden takia viestien sisältö on hyvä käydä läpi myös rikkaamman viestintäkanavan välityksellä. Mikäli asiakas ei voi olla paikanpäällä seuraamassa järjestelmän kehitystä, helpottavat usein tehtävät pienet julkistukset viestimistä huomattavasti. (Jeffries 2000)

5.4 Pariohjelmointi

Yritysjohdolle ja asiakkaalle on mahdollisesti vaikeaa perustella, miksi tulisi tuhlata kaksi työntekijää siihen, jonka yksikin voi tehdä. Pariohjelmointi on kuitenkin osoittautunut Nosakin (1998) mukaan tehokkaaksi tavaksi ohjelmoida. Hän järjesti testin, jossa toinen luokka opiskelijoita ohjelmoi pareittain, ja kontrolliryhmä yksinään. Pariohjelmoineet saivat tehtävän

valmiiksi keskimäärin 30,2 minuutissa, kun kontrolliryhmällä siihen meni 42,6 minuuttia. Voidaan kuitenkin kyseenalaistaa pariohjelmoinnin tehokkuus, koska vasta kaksi kertaa nopeammin suoriutumalla pariohjelmoijat olisivat käyttäneet saman verran työaikaa kuin yksin ohjelmoineet. Pariohjelmoineet saivat Nosakin testissä arvioinnista, jossa arvioitiin koodin luettavuutta (max 2p) ja toiminnallisuutta (max 6p), yhteispisteiksi 7,6, kun kontrolliryhmän arvosanaksi jäi 5,6. Näin ollen Nosak nostaa esille kaksi syytä, miksi käyttää pariohjelmointia. Ensinnä 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. (Nosak 1998)

Williams (2000) teki vastaavan testin pariohjelmoinnista yliopistoympäristössä, ja sai vastaavia tuloksia kuin Nosak omassa testissään. Pariohjelmoijat suoriutuivat tehtävistä 40-50% nopeammin kuin yksin ohjelmoineet. Pariohjelmoijien tuottama koodi oli huomattavasti laadukkaampaa, ja se läpäisi useammat testitapaukset kuin kontrolliryhmä. 90% pariohjelmoijista kertoivat ohjelmoivansa mieluummin pareittain kuin yksin, ja olevansa luottavaisempia koodinsa toimintaan. Eräs opiskelija kuvaili pariohjelmointia Williamsin artikkelissa seuraavanlaisesti:

> Kun itse olin ohjelmointivuorossa, keskityin täysillä työhöni. Halusin näyttää oman osaamiseni ja laadukkaan työskentelyni parilleni. Kun ohjelmoin, tunsin itseni itsevarmaksi: Jos teen virheen voin luottaa, että parini huomaa sen. Kun olin tarkkailija, oikoluin jokaista parini kirjoittamaa koodiriviä. Tunsin olevani vastuussa, jos koodiin tulisi virheitä, enkä huomaisi niitä. (Williams 2000)

Williamsin ja Kosakin tutkimuksissa testien osallistujina olivat opiskelijat, joilla ei välttämättä ollut vankkaa ohjelmointiosaamista. Yksinkertaisissa tehtävissä, joita suorittavat kokeneet ammattilaiset, ei pariohjelmoinnista välttämättä saada parasta mahdollista hyötyä. Jos toisen osapuolen varmuus omaan osaamiseen ei ole samaa luokkaa kuin parin, voi kyseisen parin osallistuminen olla vähäistä, ja pariohjelmoinnin tuoma lisäarvo saattaa jäädä vähäiseksi (Williams 2002). Kyseisessä tilanteessa pariohjelmointi kuitenkin kaventaa osaamiseroja nopeasti, ja taitamattomampi pari oppii vähitellen antamaan vinkkejä ja ehdotuksia.

Pariohjelmointia voi rajoittaa ohjelmoijien erilaiset vuorokausirytmit ja joustavat työajat. Parien käymä keskustelu voi häiritä yksin ohjelmoijien keskittymistä työhönsä. Lisäksi jotkut työntekijät haluavat mieluummin työskennellä yksin kuin pareittain. Pariohjelmointi on kuitenkin kokeilemisen arvoista, ja sen voi ottaa käyttöön osittain työskentelemällä esimerkiksi pari kertaa viikossa pareittain. Pariohjelmoinnin etuja - ilman puhdasta pariohjelmointia - voi hakea myös työskentelemällä samassa yhteisessä työtilassa ja kysymällä työkaverin apua aina, kun jää jumiin.

5.5 Tiimin koko

XP ei sovi isoille projekteille. Kun tiimin koko on useita kymmeniä tai satoja henkilöitä, ei XP ole enää paras mahdollinen menetelmä. (Beck 1999). XP:n vastustajat antavat usein kritiikkiä olettaen, että XP:tä käytetään suurien projektien läpiviemiseksi, vaikka sellaiseen sitä ei ole edes tarkoitettu. Raskaammissa ohjelmistokehitysmenetelmissä dokumentaatiolla on suuri rooli. XP:ssä suullisella viestinnällä pyritään korvaamaan dokumentaatiota, jolloin tiimin kasvaessa asioista tiedottaminen kaikille vaikeutuu. XP:ssä koodin yhteisomistuksen vuoksi ohjelmoijien pitäisi olla jollain asteella tietoisia

järjestelmän kaikista toiminnoista. Miljoonia koodirivejä sisältävässä ohjelmistossa tämä lienee kohtuuton vaatimus.

5.6 Riskialttiit projektit

Metodologia ei Beckin (1999) mukaan sovi projekteille, joissa virheille ei ole varaa. XP:ssä kannustetaan rohkeuteen, ja kuka tahansa saa tehdä muutoksia mihin tahansa koodin osaan, jos näkee sen olevan tarpeellista. Virheitä voi näin ollen syntyä. Esimerkiksi taisteluohjuksen ohjaussovelluksessa tällaiseen ei kuitenkaan ole varaa.

6 YHTEENVETO

Tutkielmassa käsiteltiin Extreme Programming menetelmää, joka lukeutuu ketteriin menetelmiin. Menetelmä soveltuu erityisen hyvin pienille projektiryhmille, joissa kehitettävän järjestelmän vaatimukset voivat muuttua projektin aikana. Tutkimuskysymyksenä oli, millä tavoin XP vastaa ohjelmistoalalla ilmeneviin ongelmiin. Johdannossa ongelmiksi mainittiin kustannuksien ylitykset, aikataulujen venymiset, liiallinen virheiden määrä sekä tilanteet, joissa lopputuote ei vastaa asiakkaan toiveita. XP soveltuu parhaiten viimeisenä mainitun ongelman ratkaisemiseen. Tiiviillä kommunikaatiolla, pienillä julkaisuilla sekä ottamalla asiakas tiimin jäseneksi saadaan tuotettua juuri sellainen sovellus kuin asiakas haluaa. Tuotteen laatu saadaan pidettyä korkealla testauslähtöisen kehityksen, automaattisten testitapausten sekä asiakastestien avulla. Lisäksi pariohjelmointi eliminoi suuren osan virheistä jo ennen niiden syntymistä. Sen sijaan kysymyksiä herätti se, kuinka menetelmällä saadaan tuotettua tarkkoja aikataulu- ja kustannusarvioita, mikäli vaatimusmäärittelyä ei tehdä mahdollisimman tarkalla tasolla ennen toteutuksen aloittamista.

XP:tä voi olla hankala ottaa kerralla yrityksen käyttöön, koska Beck kuvaa sen käytänteet hyvin lyhyesti ja suuntaa-antavasti. Menetelmää tulkitsevaa kirjallisuutta, jota adoptiovaiheessa voi hyödyntää, löytyy kuitenkin runsaasti. Sen käyttöönotto vaatii kuitenkin metodologian työstämistä kullekin organisaatiolle sopivaksi. Menetelmä sopii mainiosti ohjelmistotaloille, joissa prosessinhallintamenetelmän ei haluta olevan turhan kontrolloiva. XP – tai mikään muukaan menetelmä - ei tule ratkaisemaan kaikkia ohjelmistoalalla esiintyviä ongelmia, mutta se tarjoaa hyvän työkalun, jolla ohjelmistoprojektien onnistumismahdollisuuksia pystytään parantamaan.

LÄHDELUETTELO

Astels, D., Miller, G., & Novak, M. 2002. A practical guide to extreme programming. Prentice Hall PTR.

Beck, K. 1999. Extreme programming explained: embrace change. Addison-Wesley Longman

Beck, K., & Anders, C. 2004. Extreme programming explained: embrace change – second edition. Addison-Wesley Longman, Boston

Boehm, B., 1985. A spiral model of software development and enhancement.

Burke, E. M., & Coyner B. M. 2003. Extreme programming cookbook. O'Reilly media

Fowler, M. 2001. Is Design Dead? [viitattu 14.3.2007]. Saatavilla www -muodossa <http://www.martinfowler.com/articles/designDead.html>

Haikala, I., & Märijärvi J., 2001. Ohjelmistotuotanto Talentum Media Oy, Pieksamäki

HighSmith, Jim. 2001. History: the agile manifesto. [viitattu 14.3.2007] Saatavilla www -muodossa < http://www.agilemanifesto.org/history.html>

Jeffries, R., Anderson, A., & Hendrickson, C., 2000. Extreme Programming Installed. Addison-Wesley

McConnell, S., 2002, Ohjelmistotuotannon hallinta. Edita Prima Oy, Helsinki

Newkirk, J., & Martin R.C. 2001. Extreme Programming in Practice

Nosek, J.T. 1998. The case for collaborative programming, Comm. ACM, Vol. 41, No. 3, 105–108.

Weisert, C., 2003. There's no such thing as the waterfall approach! (and there never was). Conrad, Information Disciplines, Inc., Chicago.[viitattu 14.3.2007]. Saatavilla WWW-muodossa <http://www.idinews.com/waterfall.html>

Wells, D. 2001 Extreme Programming: A gentle introduction.[viitattu 14.4.2007]. Saatavilla www -muodossa <http://www.extremeprogramming.org/>

Winston, R. 1970, Managing the Development of Large Software Systems, Proceedings of IEEE WESCON, vol. 26, no. August, p. 1-9

West, D. 2002. Metaphor, architecture and XP. 3rd International Conference on Extreme Programming

Williams, L., Kessler, R.R., Cunningham, W., & Jeffries, R. 2000. Strengthening the case for pair programming, IEEE Software, vol. 17, No. 4, 19-25.