Artturi Korvenranta

KETTERÄT MENETELMÄT ITSENÄISELLE OHJELMISTOKEHITTÄJÄLLE

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2017

TIIVISTELMÄ

Korvenranta, Artturi
Ketterät menetelmät itsenäiselle ohjelmistokehittäjälle
Jyväskylä: Jyväskylän yliopisto, 2017, 27 s.
Tietojärjestelmätiede, kandidaatin tutkielma
Ohjaaja: Halttunen, Veikko

Ketterät menetelmät kasvattavat suosiotaan ohjelmistoyritysten keskuudessa, mutta suurin osa menetelmistä jättää kokonaan huomiotta yksin työskentelevien kehittäjien joukon. Tässä tutkielmassa pyritään kirjallisuuskatsauksen keinoin selvittämään, millaisia ketteriä menetelmiä itsenäiselle ohjelmistokehittäjälle on olemassa ja miten ne eroavat toisistaan. Tutkielmassa tarkastellaan lähemmin neljää itsenäisen kehittäjän ketterää menetelmää ja vertaillaan niitä toisiinsa henkilökohtaisen ohjelmistoprosessin työtapojen pohjalta. Itsenäisen kehittäjän menetelmien havaitaan eroavan toisistaan varsin laajasti, mikä antaa kehittäjälle paremmat mahdollisuudet valita itselleen sopiva menetelmä henkilökohtaisten mieltymysten tai projektin luonteen mukaan.

Asiasanat: ketterä kehitys, ketterä menetelmä, ohjelmistokehitys, itsenäinen kehittäjä, henkilökohtainen ohjelmistoprosessi

ABSTRACT

Korvenranta, Artturi
Agile methods for autonomous software developer
Jyväskylä: University of Jyväskylä, 2017, 27 p.
Information Systems, Bachelor's Thesis
Supervisor: Halttunen, Veikko

Agile development methods are growing in popularity among software companies. However, most of these methods overlook developers working by themselves. This thesis aims to discover what sorts of methods exist for autonomous developers and how they differ from one another. The thesis is conducted as a literature review. In this thesis we take a closer look at four agile methods designed for autonomous developer and compare them with each other based on the practices of the Personal Software Process. We discover that the methods for autonomous developers differ substantially, which allows the developer to base his/her choice of method more on personal preference and the nature of the project.

Keywords: agile development, agile method, software development, autonomous developer, Personal Software Process

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Ketterät menetelmät ohjelmistokehityksessä eivät ole ainoastaan kasvattaneet suosiotaan viime vuosien aikana, vaan niistä on tullut ohjelmistokehityksen valtavirtaa (West, Grant, Gerush, & D'silva, 2010). Suurin osa ketteristä menetelmistä ja niiden esittelemistä käytännöistä on suunniteltu kuitenkin kehitystiimin käyttöön jättämällä yhden hengen "tiimit" huomioimatta. Vuonna 2010 Suomessa toimi pieniä, korkeintaan neljän hengen ohjelmistoyrityksiä, 4087 kappaletta, jotka vastasivat 9,8 %:sta alan liikevaihdosta. Yhteensä näissä yrityksissä työskenteli 4184 henkilöä, eli valtaosassa yrityksistä työskenteli vain yksi henkilö. (Rönkkö & Peltonen, 2012.)

Yhden hengen yritykset eivät kuitenkaan ole ainoita, joissa itsenäisen kehittäjän ketterät menetelmät saattavat olla avuksi. Isommissakin yrityksissä on mahdollista olla pieniä projekteja, joissa kehitystyöstä vastaa yksi ainoa henkilö. Toisaalta itsenäinen kehittäjä voi kehittää ohjelmistoa myös itselleen tavoittelematta taloudellista hyötyä, jolloin tämä on itse oma asiakkaansa (Akpata & Riha, 2004).

Tässä tutkielmassa itsenäisellä kehittäjällä tarkoitetaan Akpatan ja Rihan (2004) määrittelemää mukaillen sellaista ohjelmistokehittäjää, joka työskentelee projektissa, jossa hän on ainoa ohjelmoija. Vastaavasti yksittäisellä kehittäjällä tarkoitetaan tässä tutkielmassa yhtä ohjelmistokehittäjää ottamatta kantaa siihen, työskenteleekö tämä ryhmässä vai ei.

Tässä tutkielmassa pyritään löytämään vastaukset seuraaviin tutkimusongelmiin:

- Millaisia ketteriä menetelmiä itsenäiselle ohjelmistokehittäjälle on olemassa?
- Miten itsenäiselle kehittäjälle tarkoitetut menetelmät eroavat toisistaan?

Luvussa 2 esitellään ketterät ohjelmistokehityksen menetelmät yleisesti ja tutustutaan kahteen menetelmään tarkemmin, sekä luodaan yleiskatsaus yksittäiselle kehittäjälle tarkoitettuun ohjelmistokehityksen toimintamalliin. Luvussa 3 tu-

tustutaan neljään itsenäiselle kehittäjälle tarkoitettuun menetelmään, ja luvussa 4 näitä menetelmiä vertaillaan toisiinsa.

Tutkimusmenetelmä tässä tutkielmassa on kirjallisuuskatsaus, joka toteutettiin etsimällä aiheeseen liittyviä tieteellisiä julkaisuja internetissä tarjolla olevien hakutyökalujen avulla. Tutkielma antaa yleiskuvan siitä, millaisia itsenäisen kehittäjän toimintamalleja on olemassa, ja miten ne eroavat toisistaan. Lisäksi tutkielman tulosten avulla itsenäinen ohjelmistoyrittäjä voi tehdä valistuneemman päätöksen siitä, millaisia ketteriä menetelmiä hänen kannattaa omaksua hyödyttääkseen liiketoimintaansa.

2 OHJELMISTOKEHITYKSEN TOIMINTAMALLEJA

Tässä luvussa tarkastellaan millaisia toimintamalleja ohjelmistokehitykseen on tarjolla. Luvussa 2.1 tarkastellaan ketterää ohjelmistokehitystä yleisesti, jonka jälkeen luodaan lähempi katsaus kahteen ketterään kehitysmenetelmään: Scrum ja Extreme Programming. Lopulta tutustutaan yksittäisen kehittäjän avuksi luotuun prosessimalliin: Personal Software Process.

2.1 Ketterä ohjelmistokehitys

Ketterä ohjelmistokehitys sai alkunsa 1990-luvun lopulla, kun useat uusia ja vanhoja ajatuksia sisältävät toimintamallit alkoivat saada julkisuutta. Nämä toimintamallit korostivat läheistä kehittäjä-asiakassuhdetta, liikearvon nopeata välittämistä, ja tiiviitä, itseohjautuvia tiimejä. Vuonna 2001 tätä toimintamallien joukkoa alettiin kutsua ketteräksi ohjelmistokehitykseksi. (Agile Alliance, 2015.) Ketterien menetelmien yhteiset arvot ja periaatteet kuvataan ketterän ohjelmistokehityksen julistuksessa sekä 12 periaatteen yhteenvedossa. Ketterän ohjelmistokehityksen julistus sen arvoineen kuuluu näin:

> *Löydämme parempia tapoja tehdä ohjelmistokehitystä, kun teemme sitä itse ja autamme muita siinä. Kokemuksemme perusteella arvostamme:*
>
> ***Yksilöitä ja kanssakäymistä*** *enemmän kuin menetelmiä ja työkaluja*
> ***Toimivaa ohjelmistoa*** *enemmän kuin kattavaa dokumentaatiota*
> ***Asiakasyhteistyötä*** *enemmän kuin sopimusneuvotteluja*
> ***Vastaamista muutokseen*** *enemmän kuin pitäytymistä suunnitelmassa*
>
> *Jälkimmäisilläkin asioilla on arvoa, mutta arvostamme ensiksi mainittuja enemmän. (Beck ym., 2001.)*

Ketterän ohjelmistokehityksen 12 periaatetta ovat:

1. *Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.*
2. *Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.*
3. *Toimitamme versioita toimivasta ohjelmistosta säännöllisesti, parin viikon tai kuukauden välein, ja suosimme lyhempää aikaväliä.*
4. *Liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.*
5. *Rakennamme projektit motivoituneiden yksilöiden ympärille. Annamme heille puitteet ja tuen, jonka he tarvitsevat ja luotamme siihen, että he saavat työn tehtyä.*
6. *Tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu.*
7. *Toimiva ohjelmisto on edistymisen ensisijainen mittari.*
8. *Ketterät menetelmät kannustavat kestävään toimintatapaan. Hankkeen omistajien, kehittäjien ja ohjelmiston käyttäjien tulisi pystyä ylläpitämään työtahtinsa hamaan tulevaisuuteen.*
9. *Teknisen laadun ja ohjelmiston hyvän rakenteen jatkuva huomiointi edesauttaa ketteryyttä.*
10. *Yksinkertaisuus - tekemättä jätettävän työn maksimointi - on oleellista.*
11. *Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoituvissa tiimeissä.*
12. *Tiimi tarkastelee säännöllisesti, kuinka parantaa tehokkuuttaan, ja mukauttaa toimintaansa sen mukaisesti. (Beck ym., 2001.)*

Andrew Pham ja Phuong-Van Pham (2012) korostavat neljää erityistä syytä, miksi ketterät menetelmät tuottavat tavallisesti parempia tuloksia projektinhallinnassa ja ohjelmistokehityksessä: riskienhallinta, kehityksen lyhyempi elinkaari, mukautuvampi projektinhallinta ja ihmisläheisemmät prosessit. Kolme ensinnä mainittua johtuvat ketterien prosessien iteratiivisuudesta, joka mahdollistaa nopean reagoinnin toimintaympäristön yllättäviin muutoksiin sekä vähentää eri vaiheiden arviointiin ja hyväksymiseen kuluvaa aikaa. Ihmisläheisyydellä tarkoitetaan sitä, että siinä missä aiemmin johtajat kertoivat tiiminjäsenille mitä näiden tulee tehdä, nyt tiimit päättävät omillaan siitä, miten tehtävistä suoriudutaan. (Pham & Pham, 2012.)

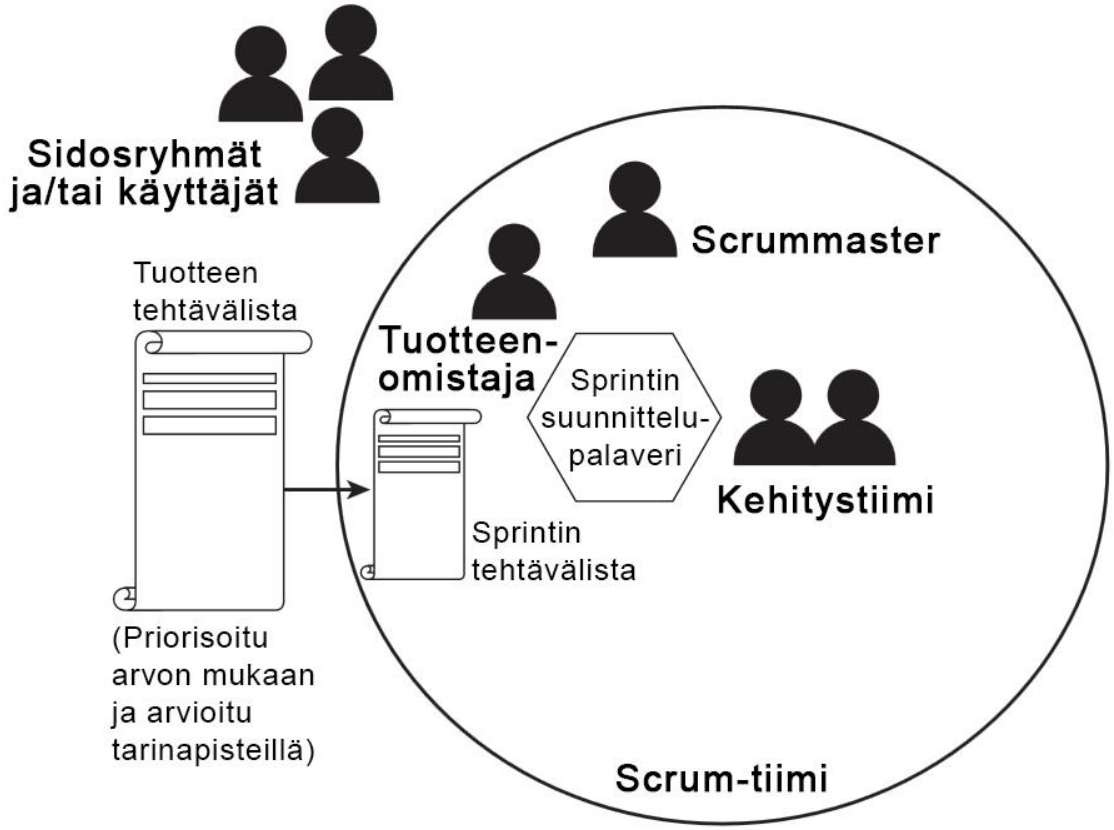
Seuraavaksi tarkastellaan lähemmin kahta ketterän kehityksen periaatteita mukailevaa viitekehystä: Scrum ja Extreme Programming.

2.1.1 Scrum

Scrum on ohjelmistokehityksen malli, joka pyrkii vastaamaan toimintaympäristön muutokseen perinteisiä malleja, kuten vesiputous- tai spiraalimalleja, paremmin. Toisin kuin perinteisissä malleissa, Scrumissa oletetaan, että ohjelmiston suunnittelu- ja kehitysprosessit ovat arvaamattomia. (Schwaber, 1997.)

Scrum koostuu kolmesta vaiheesta: suunnittelu-, kehitys- ja päättämisvaihe. Suunnittelu- ja päättämisvaiheet ovat ennalta määriteltyjä prosesseja, joiden eteneminen on lineaarista. Kehitysvaihe koostuu toistuvista sprinteistä, jotka ovat yleensä 1-4 viikon mittaisia kehitysjaksoja, joiden aikana määrätään kyseessä olevan sprintin tavoitteet, pyritään saavuttamaan tavoitteet, ja arvioidaan sprintin tulos. (Schwaber, 1997.) Scrum ei ehdota mitään varsinaisia ohjelmistokehityksen tekniikoita toteutukseen, vaan keskittyy ennemmin siihen, miten tiimin jäsenten tulee toimia muuttuvassa ympäristössä (Abrahamsson, Salo, Ronkainen, & Warsta, 2002).

Scrum-tiimissä on kuusi eri roolia, joilla jokaisella on omat tehtävänsä: Scrummaster, Product Owner (suom. *tuotteenomistaja*), kehitystiimi, asiakas, käyttäjä ja johto (Abrahamsson ym., 2002). Seuraavaksi malli scrum-projektin lähtötilanteesta rooleineen (Kuvio 1). Kuviossa asiakasta, käyttäjää ja johtoa edustaa yksi entiteetti, ”sidosryhmät ja/tai käyttäjät”.



Kuvio 1: Scrum-projektin lähtötilanne (Pham & Pham, 2012)

Scrum-projektin alussa sidosryhmät tai näitä edustamat käyttäjät määrittelevät tuotteelle asetetut vaatimukset, joista tuotteenomistaja luo tuotteelle tehtävälistan. Tuotteenomistaja jakaa tehtävät sprintteihin yhdessä muiden scrum-tiimin jäsenten kanssa, jonka jälkeen varsinainen kehitystyö voi alkaa. (Pham & Pham, 2012.)

Schwaberin (1997) mukaan Scrumin avulla kehitetty tuote on joustava. Se määräytyy toimintaympäristön muuttujien mukaan, joihin lukeutuu esimerkiksi aika, kilpailu, hinta ja käytännöllisyys. Toimitettavana olevaan tuotteeseen tehdään koko projektin ajan muutoksia niin, että se vastaa toimintaympäristön vaatimuksia paremmin.

2.1.2 Extreme Programming

Extreme Programming (XP) on Scrumin tavoin kehitetty vastaamaan perinteisiä malleja paremmin muuttuviin ohjelmistokehityksen olosuhteisiin (Beck, 1999). XP sisältää 12 käytäntöä, joihin kehitystiimi sitoutuu. Beck (1999) muistuttaa kuitenkin, että sääntöjä voi muuttaa, jos tiimi päättää niin ja osaa arvioida muutoksen vaikutukset. XP:n 12 käytäntöä ovat (Agarwal & Umphress, 2008; Beck, 1999; Cronin, 2003):

1. Suunnittelupeli
2. Pienet julkaisut
3. Metafora
4. Yksinkertainen suunnittelu
5. Testit
6. Refaktorointi
7. Pariohjelmointi
8. Jatkuva integraatio
9. Yhteisomistajuus
10. Paikalla oleva asiakas
11. 40-tuntinen viikko
12. Ohjelmointistandardit

Suunnittelupelissä asiakas luo käyttäjätarinoita, joiden implementoinnin kestolle ohjelmoijat antavat arvion. Toimitetun arvion perusteella asiakas päättää julkaisujen laajuuden ja keston. Ohjelmoijat sisällyttävät järjestelmään vain ne toiminnot, joita asiakkaan kyseiseen iteraatioon valitsemat käyttäjätarinat vaativat. *Pienillä julkaisuilla* tarkoitetaan mahdollisimman lyhyitä kehityssyklejä, ja tavoitteena on saada toimiva versio tuotantoon niin pian kuin mahdollista.

Metaforat ovat kutsumanimiä, jotka helpottavat kehitysprosessin ohjaamista ja asiakkaan kanssa kommunikointia. *Yksinkertainen suunnittelu* XP:ssä tarkoittaa kaiken ylimääräisen työn minimointia. Tällä tarkoitetaan esimerkiksi liian pitkälle tulevaisuuteen suunnittelemisen välttämistä sekä mahdollisimman yksinkertaisen ohjelmointitavan valintaa.

Testit tehdään XP:ssä ennen varsinaista koodaamista. Ennen kuin kirjoitetun koodin saa implementoida järjestelmään, täytyy sen läpäistä sille kirjoitetut testit. Asiakas kirjoittaa jokaiselle käyttäjätarinalle käyttötestit, jotka ohjelman täytyy myös läpäistä. *Refaktorointi* tarkoittaa koodin muuttamista niin, että uusien ominaisuuksien lisääminen siihen on mahdollisimman suoraviivaista.

Pariohjelmoinnissa kaksi henkilöä tuottaa koodia yhtä työasemaa käyttäen. Pareista toinen kirjoittaa koodia ja toinen seuraa vierestä ja arvioi sitä. Vuoroja vaihdetaan säännöllisesti. *Jatkuvalla integraatiolla* tarkoitetaan, että uusi koodi integroidaan olemassa olevaan järjestelmään korkeintaan parin tunnin viiveellä sen kirjoittamisesta.

Yhteisomistajuus tarkoittaa, että kuka tahansa ohjelmoija voi parantaa koodia missä tahansa järjestelmän osassa huolimatta siitä, kuka sen on alun perin

kirjoittanut. *Paikalla oleva asiakas* -käytäntö takaa sen, että kehitystiimillä on asiakkaan edustaja jatkuvasti käytettävissään.

40-tuntinen viikko on korkein sallittu työmäärä, sillä ylityöt johtavat huonoon koodin laatuun. *Ohjelmointistandardit* takaavat yhtenäisen koodin läpi koko ohjelmiston, mikä edesauttaa jatkuvuutta esimerkiksi henkilöstön vaihtuessa.

Extreme Programming on osoittautunut toimivaksi ratkaisuksi myös käytännössä. Rumpen ja Schröderin (2014) toteuttama 45 vastaajan kyselytutkimus osoittaa, että 93,3 % XP:tä käyttäneistä yrityksistä ja organisaatioista käyttäisi mallia uudestaankin. Loput 6,7 % puolestaan haluaisivat parannetun version XP:stä, mutta jokainen kyselyyn vastannut kertoo kannattavansa Extreme Programmingin hyödyntämistä jatkossakin.

2.2 Personal Software Process

Personal Software Process (PSP, suom. *henkilökohtainen ohjelmistoprosessi*) on yksittäisen ohjelmistokehittäjän tueksi laadittu viitekehys. PSP pohjautuu seuraaviin suunnittelun ja laadun periaatteisiin (Humphrey, 2000):

- Jokainen kehittäjä on erilainen; ollakseen tehokkaita, kehittäjien täytyy suunnitella työnsä ja perustella suunnitelmansa henkilökohtaisella tiedollaan.
- Parantaakseen suoritustaan yhtäjaksoisesti, kehittäjien täytyy käyttää itsenäisesti hyvin määriteltyjä ja mitattuja prosesseja.
- Tuottaakseen laadukkaita tuotteita, kehittäjien täytyy tuntea vastuuta tuotteensa laadusta. Ensiluokkaiset tuotteet eivät synny vahingossa, vaan kehittäjien täytyy pyrkiä laadukkaaseen työhön.
- Vikojen etsiminen ja korjaaminen on halvinta tehdä mahdollisimman aikaisessa vaiheessa prosessia.
- On tehokkaampaa ennaltaehkäistä vikoja kuin etsiä ja korjata niitä.
- Nopein ja halvin tapa tehdä työ on aina oikea tapa.

Humphreyn (2000) mukaan PSP:ssä on neljä toimintatapaa, joiden mukaan kehittäjän tulee toimia: määritellyn prosessin noudattaminen, työn suunnittelu, tiedon kerääminen, ja prosessin analysointi ja kehittäminen kerätyn tiedon perusteella. Näiden toimintatapojen seuraamiseen PSP tarjoaa laajan määrän ohjeita ja mallipohjia, joita kehittäjän tulee työssään hyödyntää. Ohjeet liittyvät esimerkiksi työajan, työn koon ja sen laadun seurantaan, ja työn suunnitteluun. (Humphrey, 2000). Edellä mainittuja toimintatapoja tarkastellaan tarkemmin tämän tutkielman luvussa 4, jossa itsenäiselle kehittäjälle suunniteltuja ketteriä menetelmiä vertaillaan toimintatapojen pohjalta.

PSP:n käyttöönotto tapahtuu vaiheittain seitsemän version kautta (PSP0, PSP0.1, …, PSP2.1, PSP3), joissa jokaisessa on samankaltaiset kokoelmat lokeja, lomakkeita, ohjeita ja standardeja. Ohjeet tarjoavat kehittäjälle tarkan kuvauksen kustakin prosessin vaiheesta sekä mallipohjat datan keräykseen ja hallin-

taan. Humphreyn (2000) mukaan PSP:n opetus alkaa yliopistoissa niin, että opiskelijat kirjoittavat yhden tai kaksi ohjelmaa versiolla PSP0 ja siirtyvät sitten seuraavaan versioon. Kuhunkin ohjelmaan sovelletaan niitä PSP:n käytäntöjä, joita versio tuo tullessaan sekä edellisissä versioissa esiteltyjä käytäntöjä. PSP:n vaiheistus on kuvattu seuraavassa taulukossa (Taulukko 1):

Taulukko 1: PSP:n vaiheet (Humphrey, 2000)

Versio	**PSP:n käytännöt**
PSP0	• Nykyinen prosessi • Perusmittaukset
PSP0.1	• Ohjelmointistandardi • Ehdotus prosessin parantamiseksi • Koon mittaukset
PSP1	• Koon arviointi • Testiraportti
PSP1.1	• Tehtävien suunnittelu • Aikataulun suunnittelu
PSP2	• Koodin arviointi • Suunnittelun arviointi
PSP2.1	Suunnittelun mallipohjat
PSP3	Syklinen kehitys

Vaikka PSP:n on tutkittu parantavan työn laatua (Hayes & Over, 1997; Stark & Crocker, 2003), on se saanut osakseen myös kritiikkiä sen vaatimasta laajasta dokumentaatiosta ja kurinalaisesta noudattamisesta (Dzhurov, Krasteva, & Ilieva, 2009). Jotta PSP soveltuisi paremmin ketterään kehitysympäristöön, on siitä johdettu monia malleja, jotka yhdistävät PSP:n ja jonkin olemassa olevan ketterän menetelmän (Shen, Rong, & Shao, 2013).

Shen ym. (2013) toteavat kirjallisuuskatsauksessaan, että PSP:n yhdistämisellä johonkin ketterään menetelmään on kuusi keskeistä etua:

1. PSP tarjoaa mittareita ja mekanismeja datan seurantaan.
2. PSP auttaa perustellun arvion tekemisessä.
3. PSP auttaa optimoimaan suunnitelmia.
4. PSP tukee laadullisten tavoitteiden saavuttamista.
5. PSP tarjoaa lisädokumentaatiota ketterille menetelmille.
6. PSP auttaa yksittäistä kehittäjää parantamaan henkilökohtaista työpanostaan asteittain.

Seuraavassa luvussa tarkastellaan yksittäiselle kehittäjälle suunnattuja ketteriä menetelmiä ja luvussa 4 näitä menetelmiä vertaillaan keskenään PSP:n toimintatapojen pohjalta.

3 KETTERÄT MENETELMÄT ITSENÄISELLE KEHITTÄJÄLLE

Tässä luvussa esitellään neljä itsenäiselle ohjelmistokehittäjälle tarkoitettua ketterää menetelmää: Personal Extreme Programming, Agile Solo, Cowboy ja Solo Iterative Process. Luvussa 4 näitä menetelmiä vertaillaan toisiinsa käyttäen apuna Humphreyn (2000) määrittämiä toimintatapoja PSP:lle.

3.1 Personal Extreme Programming

Agarwalin ja Umphressin (2008) kehittämä Personal Extreme Programming (PXP) hyödyntää sekä PSP:n että XP:n tärkeimpiä periaatteita. PXP pyrkii olemaan ratkaisu raskaimpien ja keveimpien menetelmien välissä, olemalla riittävän perusteellinen rasittamatta kehittäjää kuitenkaan liikaa ylimääräisellä byrokratialla. PXP keventää PSP:n vaatimaa dokumentaatiota ja hyödyntää XP:stä niitä ominaisuuksia, jotka sopivat itsenäiselle kehittäjälle. Tuloksena on iteratiivinen prosessimalli, joka antaa kehittäjälle työkalut reagoida ja toimia joustavammin muutostilanteissa. (Agarwal & Umphress, 2008; Dzhurov ym., 2009.)

Agarwal ja Umphress (2008) toteavat, että suurinta osaa XP:n 12 käytännöstä (ks. 2.1.2) voi käyttää suoraan PXP:ssä, ja loputkin voi ottaa käyttöön pienin muutoksin. Pariohjelmoinnin tuomia hyötyjä on kuitenkin yksin työskennellessä mahdotonta saavuttaa, mihin Agarwal ja Umphress (2008) ehdottavat korvaavaksi toimintatavaksi esimerkiksi muistutuksia kollegalta.

PXP sisältää PSP:lle ominaiset toimintaohjeet. Ohjeet antavat kattavan rakenteen koko projektin läpivientiin IF- ja FOR-silmukoiden muodossa. PXP:n toimintaohje kehitysvaihetta lukuun ottamatta on kuvattu seuraavassa taulukossa (Taulukko 2).

Taulukko 2: PXP-toimintaohjeet (Agarwal & Umphress, 2008; suomentanut Kähkönen, 2014)

Alkuehto	A1. Ohjelmointistandardi käytössä.
Suunnittelu	S1. Tee tai hanki vaatimukset: A. Kirjoita vertauskuva. B. Kirjoita käyttäjätarinat. S2. FOR kaikki käyttäjätarinat: A. Jaa käyttäjätarina ominaisuuksiin. END S3. Tee liiketoimintavetoinen suunnittelu (käytä toimialueesi sanastoa suunnittelussa). S4. Ryhmittele ominaisuudet järkevästi (OminaisuusJoukko). S5. Kirjoita hyväksymistestit OminaisuusJoukoille. S5. Arvioi OminaisuusJoukon koko ja kehitykseen kuluva aika päivinä. S7. Järjestä OminaisuusJoukot tärkeyden
Kehitys	PXP:n kehitysvaihe koostuu sisäkkäisistä IF- ja FOR-silmukoista, jotka antavat kattavan kehyksen ohjelmiston kehitykselle (ks. Taulukko 3).
Jälkituotanto	J1. Toteuta lopullinen hyväksymistestaus tuotannon koodille.
Lopetusehto	L1. Valmis, testattu ohjelma.

PXP:n kehitysvaihe on laajuudeltaan toimintaohjeen vaiheista kattavin. Luettavuuden parantamiseksi kehitystä kuvaava osuus on tässä työssä esitetty omassa taulukossaan, mutta kronologisesti vaihe on suunnittelu- ja jälkituotantovaiheiden välissä. Kehitysvaiheen toimintaohje on kuvattu seuraavassa taulukossa (Taulukko 3).

Taulukko 3: PXP-toimintaohjeiden kehitysvaihe (Agarwal & Umphress, 2008; suomentanut Kähkönen, 2014)

Kehitys	K1. FOR kaikki OminaisuusJoukot TärkeysJonossa 1. Ota ensimmäinen OminaisuusJoukko TärkeysJonosta. 2. IF (muutos tehty OminaisuusJoukkoon == YES) THEN a. Päivitä OminaisuusJoukko. b. Järjestä TärkeysJono uudelleen. 3. ELSE a. Päivitä suunnittelu ja tee iteraatiosuunnitelma nykyiselle OminaisuusJoukolle. b. FOR kaikille Ominaisuuksille OminaisuusJoukossa 1. Ota Ominaisuus ja jaa se tehtäviin. 2. Järjestä tehtävät TehtäväJonoksi tärkeyden mukaan. 3. FOR kaikille tehtäville TehtäväJonossa a. Ota tärkein tehtävä. b. Kirjoita tehtävän yksikkötesti. c. Kirjoita/muokkaa koodi tehtävän mukaisesti. d. Käy koodi läpi (engl. code walkthrough). e. Päivitä koodi versionhallinnan kehityshaaraan. f. Käännä ja tee yksikkötesti. g. IF (Yksikkötesti == Läpi) THEN 1. Tee koodille hyväksymistesti.

	2. IF (Hyväksymistesti == Läpi) THEN a. Integroi koodi refaktorointihaaraan 3. ELSE a. Luo tehtävä "korjaa koodi" tärkeydellä 1. h. ELSE 1. Luo tehtävä "korjaa koodi" tärkeydellä 1. 4. END c. END 4. Tee integraatiotesti refaktorointihaarassa. 5. IF (Integraatiotesti == Läpi) THEN a. Refaktoroi koodi. b. Integroi tuotantohaaraan. 6. ELSE a. Luo uusi tehtävä: "korjaa refaktorointihaara" tärkeydellä 1. 7. Tee hyväksymistesti tuotantohaaran koodille. 8. IF (Hyväksymistesti == Läpi) THEN a. Julkaise iteraation aikana tuotettu ohjelma. b. IF (Uusi OminaisuusJoukko annetaan == Yes) THEN 1. Päivitä TärkeysJono 9. ELSE a. Luo tehtävä "korjaa tuotantohaara" tärkeydellä 1. END

PXP eroaa kaikista muista tässä tutkielmassa esitellyistä itsenäisen kehittäjän menetelmistä etenkin siten, että se tarjoaa hyvin tarkan polun, jota seurata ohjelmointityön edetessä. PXP ottaa myös kaikkein eniten kantaa kehityksen teknisiin yksityiskohtiin, kuten yksikkötestaamiseen tai versionhallintaan. Sen sijaan kehittäjä-asiakas-yhteistyö saa toimintaohjeissa vähemmän huomiota, eikä sen käytännön toimintatapoihin oteta kantaa lainkaan.

3.2 Agile Solo

Agile Solo on Anna Nyströmin (2011) kehittämä ohjelmistokehityksen prosessimalli, joka on tarkoitettu yhden kehittäjän projekteille. Se on kehitetty valmiita ketteriä menetelmiä, kuten Scrumia ja XP:tä, analysoiden ja niitä itsenäiselle kehittäjälle sopivaksi soveltaen (Nyström, 2011).

Agile Solossa on viikoittaiset ja kuukausittaiset iteraatiot. Viikkotasolla ohjelmisto esitellään silloisessa tilassaan vastaavalle esimiehelle tai suoraan asiakkaalle, jonka jälkeen projektitehtävien prioriteetti päivitetään ja työtä jatketaan tärkeimmästä tehtävästä alkaen. Toimiva ohjelmisto toimitetaan asiakkaalle kerran kuukaudessa ja tälle annetaan mahdollisuus kommentoida työtä. Jokaisen iteraation tavoitteet tulee päivittää viikoittaisissa palavereissa asiakaspalautteen perusteella. (Nyström, 2011.)

Agile Solo jättää kehittäjän ja asiakkaan väliseksi päätökseksi sen, miten iteraatioiden tehtäviä seurataan. On kuitenkin tärkeätä, että valittu tapa toimia

on mahdollisimman yksinkertainen. Sekä kehittäjän että asiakkaan tulisi helposti päästä näkemään yleiskatsaus projektin etenemisestä varmistaakseen, että siinä tehdään oikeita asioita. Jokainen iteraatio päätetään arviointiin, jossa verrataan suunniteltuja ja toteutuneita työaikoja, ja keskustellaan yleisesti miten toimintaa parannetaan seuraavaan iteraatioon. (Nyström, 2011.)

Testivetoinen kehitys on Agile Solon tärkein ominaisuus. Kun testit kirjoitetaan etukäteen, varmistetaan, että tehdään vain niitä asioita, joita on pakko tehdä. Se auttaa kehittäjää myös hahmottamaan lopputuloksen paremmin. (Nyström, 2011.)

Kompensoidakseen niitä hyötyjä, joita pariohjelmoinnilla saataisiin, Agile Solo esittää vaihtoehtoisia tapoja ohjelmakoodin arvioimiseen. Esimiehen tai toisen ohjelmoijan tulee arvioida kirjoitettu koodi vähintään kaksi kertaa viikossa. Virheiden välttämisen lisäksi työtapa auttaa kehittäjää tuottamaan yhtenäisempää ja paremmin kommentoitua koodia. Kehittäjän tulee myös itse tarkastella kirjoittamaansa koodia päivittäin. (Nyström, 2011.) Nyström (2011) toteaa, että Agile Solon yksittäisiä käytäntöjä tulee räätälöidä toimintaympäristön mukaan, ja kehitysprosessia tulee arvioida ja muuttaa koko projektin ajan.

3.3 Cowboy

Agile Solon tapaan myös Ashby Brooks Hollarin (2006) kehittämä Cowboy on saanut vahvasti vaikutteita muista ketteristä menetelmistä. Cowboyn lähtökohtainen ajatus on olla helppo noudattaa, mutta riittävän hyödyllinen, jotta sen käyttöön ottaminen olisi kannattavaa. Cowboyn metodologia on jaettu neljään osa-alueeseen: yleiset käytännöt, asiakassuhde, iteraatioiden hallinta, ja käyttöönotto. (Hollar, 2006.)

Hollar (2006) laati Cowboysta seuraavan yleiskuvauksen, joka toimii pohjana kehitykselle, ja jota on tarkoitus muuttaa kunkin projektin tarpeiden mukaan:

I. Asiakas-kehittäjäsuhde
   a. Jos asiakas ei pääse tapaamisiin, määrittäkää asiakkaalle edustaja
   b. Käytä kyselyä varmistaaksesi, että asiakas on tyytyväinen projektin etenemiseen ja palaverikäytäntöihin
   c. Käytä työkaluja kuten sähköposti, pikaviestimet ja puhelut kysyäksesi asiakkaalta kysymyksiä kehityksen aikana
II. Palaverit
   a. Järjestäkää suunnittelupalaveri, jossa määritellään alustavat ominaisuudet
      i. Pitäkää palaveri viihtyisässä huoneessa, jossa on riittävästi pöytä- ja kirjoitustaulutilaa
      ii. Määrittäkää yleistavoitteet eri käyttäjäryhmille
      iii. Määrittäkää kehittäjälle alustava tehtävälista, joka on tärkeysjärjestyksessä

iv. Kehittäkää asiasanasto mahdollisille monitulkintaisille termeille
v. Suunnitelkaa alustava käyttöliittymä

b. Versionarviointipalaverit
   i. Aloittakaa palaveri tarjoamalla asiakkaalle viimeisimmän julkaisun testausohje
   ii. Tunnistaakseen uudelleenkirjoitustehtävät (engl. *refactoring tasks*), kehittäjän tulee kirjata ylös asiakkaan suulliset kommentit ja havainnot tämän kohtaamista ongelmista
   iii. Asiakas ja kehittäjä päivittävät tavoitelistan
   iv. Asiakas ja kehittäjä päivittävät ja uudelleenpriorisoivat tehtävälistan
   v. Asiakas ja kehittäjä muokkaavat asiasanastoa tarpeen mukaan
   vi. Asiakas ja kehittäjä keskustelevat muutoksista käyttöliittymään
   vii. Asiakas vastaa kyselyyn

III. Osatulokset

a. Tavoitelista
   i. Tavoitteet tulee pitää niin yleisellä tasolla kuin mahdollista, tunnistaen ne loppukäyttäjän tarpeet, jotka järjestelmä tyydyttää
   ii. Jokaisesta tavoitteesta johdetaan yksi, tai mieluummin useampi, tehtävä
   iii. Kategorisoi tavoitteet loppukäyttäjäryhmien mukaan
   iv. Jokainen yksittäinen tavoite tulee olla kirjoitettu kokonaisena lauseena ja siinä tulee käyttää verbejä kuten ”sallita, tarjota, mahdollistaa”

b. Tehtävälista
   i. Tehtävät ovat sellaisia toimintoja, joita järjestelmä antaa loppukäyttäjän suorittaa
   ii. Jokainen tehtävä alkaa lauseella ”(käyttäjäryhmän) täytyy pystyä…”
   iii. Ryhmitelkää tehtävät kategorioihin sen mukaan mihin tavoitteeseen tai pääominaisuuteen ne kuuluvat

c. Asiasanasto
   i. Määritelkää asiasanastossa tärkeimmät termit ja sellaiset termit, jotka voidaan tulkita monella eri tavalla
   ii. Varmistakaa, että asiakas ja kehittäjä ovat samaa mieltä määritelmästä
   iii. Pitäkää asiasanasto käsillä jokaisessa palaverissa

d. Koodi
   i. Käytä integroitua kehitysympäristöä (IDE), jossa on mielellään tuki koodin täydentämiselle
   ii. Seuraa ohjelmointikäytäntöjä tehdäksesi yhdenmukaista koodia
   iii. Valitse merkitykselliset nimet luokille, metodeille ja muuttujille parantaaksesi luettavuutta
   iv. Kommentoi luokat ja metodit asianmukaisesti
   v. Käytä yksikkötestausta varmistaaksesi jokaisen luokan toiminta

vi. Ylläpidä tehtävälistaa mukailevaa muistilistaa, joka sisältää uudelleenkirjoitustehtävät ja muutokset käyttöliittymään

Cowboyn yleiskuvaus muistuttaa hieman aiemmin esiteltyä PXP:n toimintaohjetta, joskin Cowboy ottaa enemmän kantaa koko projektiin pelkän ohjelmoinnin sijaan. Koodiin liittyvät asiat ovat myös mukana, mutta vain yhtenä alaosana, siinä missä projektin tavoitteiden selvittäminen ja niihin pääseminen on keskiössä. Metodologisesti Cowboyn voisi ajatella PXP:n ja Agile Solon välimaastoon, sen antaen tarkkoja ehdotuksia toimintamalleista koko projektin elinkaaren kattaen, jättäen kuitenkin tietyt osat, kuten tiedon keräämisen, kokonaan menetelmän käyttäjän huolehdittavaksi.

3.4 Solo Iterative Process

Solo Iterative Process (SIP) on itsenäisen ohjelmistokehityksen prosessimalli, joka ottaa muita tässä luvussa esiteltyjä malleja enemmän kantaa ohjelmointityöhön ja vähemmän projektin hallintaan. Dormanin ja Rajlichin (2012) mukaan SIP:n iteraatiot ovat tavanomaisia ketteriä prosesseja yksinkertaisempia. SIP koostuu neljästä osasta (Dorman, 2011):

1. Tuotteen tehtävälista (engl. *product backlog)*
2. Software Change
3. Iteraatio ja julkaisu
4. SIP:n mittaaminen

Tuotteen tehtävälista on ainoa SIP:n vaihe, johon kuuluu joku muukin projektin sidosryhmän edustaja kuin itse kehittäjä. Tehtävälista koostuu käyttäjätarinoista, eli tarkemmista kuvauksista niistä muutoksista, jotka sovellukseen tulee jossakin vaiheessa implementoida. Kuka tahansa projektin sidosryhmien edustajista voi lisätä uusia käyttäjätarinoita tehtävälistalle, esimerkiksi loppukäyttäjät tai ohjelmoija itse. (Dorman, 2011.)

Software Change (SC, suom. *ohjelmistomuutos*) on SIP:n keskeisin osa, ja tarkoittaa sitä vaihetta iteraatiossa, kun ohjelmiston lähdekoodia muutetaan (Dorman, 2011). Jokainen SC koostuu viidestä vaiheesta: konseptin sijainti (engl. *concept location*), vaikutusanalyysi (*impact analysis*), toteutus (*actualization*), refaktorointi (*refactoring*) ja tarkistus (*verification*) (Dorman & Rajlich, 2012).

Konseptin sijainti -vaiheessa selvitetään, mitä nimenomaista osaa koodissa tullaan muuttamaan kyseessä olevassa iteraatiossa. Kun sijainti on selvillä, tehdään vaikutusanalyysi, jonka tuloksena on arvioitu vaikutusjoukko, eli ne ohjelman luokat, joihin kehittäjä olettaa muutoksen vaikuttavan. (Dorman & Rajlich, 2012.)

Toteutusvaiheessa muutokset sisällytetään koodiin muuttamalla olemassa olevia luokkia tai luomalla uusia. Muutettuja luokkia nimetään muutosjoukoksi,

ja sen tulisi vastata aiemmin syntynyttä vaikutusjoukkoa. (Dorman & Rajlich, 2012.)

Refaktorointi toteutetaan muokkaamalla sovelluksen rakennetta ilman, että sen toiminnallisuus muuttuu. Refaktoroinnilla joko valmistaudutaan toteutusvaiheeseen (prefaktorointi) tai se tehdään toteutuksen jälkeen poistamalla eitoivotut sivuvaikutukset (postfaktorointi). Tarkistusvaiheessa varmistetaan ohjelman toiminta uusien päivitysten jäljiltä. (Dorman & Rajlich, 2012.)

Iteraatio ja julkaisu on SIP:n vaihe, jossa kehittäjä päättää käynnissä olevan iteraation. Jokaisen iteraation päätteeksi lähdekoodin tulee olla valmista ja hyvälaatuista, mutta kehittäjä päättää silti julkaistaanko ohjelma loppukäyttäjille vai tehdäänkö siihen lisää iteraatioita. (Dorman, 2011.)

SIP:n mittaaminen tapahtuu kolmella määrätyllä tavalla: ajan seuraaminen, virheiden seuraaminen ja iteraation tehtävälistan taulukko. Näistä tärkein on ajan seuraaminen, jossa kehittäjä seuraa kuinka paljon aikaa yhden tehtävän suorittamiseen menee, ja jos tehtävä vaatii ohjelmointia, seurataan myös koodirivin kirjoittamiseen kulunutta aikaa. Kerätty data auttaa kehittäjää tulevaisuuden tehtävien suunnittelussa ja aikataulun arvioinnissa. (Dorman, 2011.)

Virheitä seurattaessa kirjataan ylös virheen löytymisen päivämäärä; tehtävä, jota tehtiin kun virhe löydettiin; virheen sijainti; virheen alkuperä; ja aika, jolloin virhe korjattiin. Virheistä kirjan pitäminen auttaa kehittäjää seuraamaan niiden korjaamiseen kuluvaa aikaa sekä selvittämään, mitkä tehtävät johtavat useimmiten uusiin virheisiin. (Dorman, 2011.)

Kun kehittäjä valitsee seuraavaan iteraatioon toteutettavat ominaisuudet, hän luo iteraation tehtävälistalle taulukon. Taulukkoon arvioidaan aika, joka kuluu kunkin tehtävän suorittamiseen perustuen aiemmin kerättyyn dataan aikaseurannasta. Sitä mukaa kun tehtävät valmistuvat, niihin oikeasti käytetty aika merkataan taulukkoon arvion rinnalle. (Dorman, 2011.)

4 Menetelmien vertailu

Hayes ja Over toteuttivat vuonna 1997 empiirisen tutkimuksen, jossa tutkittiin PSP:n vaikutusta lähes 300 opiskelijan ohjelmointityöhön. Tutkimuksessa havaittiin, että PSP:n seuraaminen parantaa työn koon ja määrän arvioinnin tarkkuutta, ja tuotteen ja prosessin laatua. Tutkimuksessa todettiin myös, että hyödyt saavutetaan tuottavuuden pysyessä ennallaan. (Hayes & Over, 1997). Myös Stark ja Crocker (2003) toteavat, että PSP:n käyttäminen auttaa kehittäjiä parantamaan sekä omaa toimintaansa että tuotteen laatua.

Koska PSP on todistetusti tehokas yksittäisen ohjelmistokehittäjän menetelmä, on luonnollista vertailla muita itsenäisen kehittäjän menetelmiä PSP:n perustaviin toimintatapoihin. Kuten luvussa 2.2 mainittiin, PSP:n toimintatavat ovat määritellyn prosessin noudattaminen, työn suunnittelu, tiedon kerääminen, ja prosessin analysointi ja kehittäminen kerätyn tiedon perusteella. Näiden toimintatapojen lisäksi vertailemme itsenäisen kehittäjän menetelmiä myös asiakassuhteen perusteella, joka on tärkeä osa ketterää kehitystä, mutta jää PSP:ssä vähälle huomiolle.

PSP:n asiayhteydessä määritellyn prosessin noudattaminen tarkoittaa PSP:n omien toimintaohjeiden mukaan toimimista. Tämän luvun vertailussa vertailemme sitä, kuinka kattavat ja täsmälliset ohjeet kyseessä oleva vertailtava menetelmä tarjoaa.

Työn suunnittelua vertailtaessa tarkastellaan miten mikäkin vertailumenetelmä ohjeistaa suunnittelemaan sovelluskehitystyötä. Vastaavasti tiedon kerääminen ja kerätyn tiedon käyttäminen -osiossa tarkastellaan kunkin menetelmän tiedon keräämisen käytäntöjä.

Viimeisenä vertailemme itsenäisen kehittäjän menetelmiä niiden asiakassuhteen perusteella. Osiossa tarkastellaan esimerkiksi millaisia keinoja vertailtava menetelmä ehdottaa kehittäjä-asiakas-viestintään tai asiakkaan roolin määrittelyyn. Vertailun kohteena on luvussa 3 esitellyt neljä itsenäisen kehittäjän menetelmää: Personal Extreme Programming, Agile Solo, Cowboy ja Solo Iterative Process.

4.1 Määritellyn prosessin noudattaminen

PXP:ssä tarjotaan prosessin noudattamiseen lähes yhtä kattavat ohjeet kuin PSP:ssä. PXP:n tarjoamissa toimintaohjeissa käydään läpi kaikki iteraation vaiheet aina suunnittelusta kehityksen lopettamiseen kohta kohdalta. Kehittäjän tulee noudattaa laadittuja toimintaohjeita. Kirjallisten toimintaohjeiden lisäksi PXP tarjoaa 12 käytäntöä, jotka on muokattu XP:n käytännöistä itsenäiselle kehittäjälle soveltuvaksi.

Agile Solo ja Cowboy eivät tarjoa valmista listaa tehtävistä, jotka pitää suorittaa. Sen sijaan molemmat menetelmät kokoavat hyväksi havaittuja käytäntöjä muista ketteristä menetelmistä, ja niitä sovelletaan itsenäisen kehittäjän tarpeisiin. Sekä Agile Solo että Cowboy suosittelevat käytännöistä poikkeamista, jos toimintaympäristö niin vaatii.

Jokaisessa SIP:n iteraatiossa käytetään toistuvasti SC-prosessia, jonka vaiheet on ennalta määritelty ja joita kehittäjän tulee noudattaa. Myös jokaisen SC:n ympärillä tapahtuvat toimenpiteet on määritelty SIP:ssä.

4.2 Työn suunnittelu

PXP perii suunnittelupeli-käytännön suoraan XP:ltä. Jos kehittäjä työskentelee itselleen tai tietää tarpeeksi asiakkaan näkökulmasta, hänen tarvitsee vain vaihtaa itsensä asiakkaan rooliin. PXP:ssä hyvinä suunnittelukäytäntöinä pidetään käyttäjätarinoiden kirjoittamista, arvioimista ja priorisoimista.

Agile Solossa viikoittaisten ja kuukausittaisten iteraatioiden tavoitteet päivitetään viikoittaisissa palavereissa. Palavereissa priorisoidaan tehtävät asiakkaan ja käyttäjien kommenttien perusteella, ja arvioidaan iteraation aikataulu.

Cowboyta käytettäessä luodaan aluksi listat tavoitteille ja tehtäville yhdessä asiakkaan kanssa. Listat päivitetään kehittäjän ja asiakkaan välisissä versionarviointipalavereissa.

SIP:ssä luodaan kehitettävälle tuotteelle tehtävälista, josta jokaiseen SC-prosessiin poimitaan toteutettavat ominaisuudet. Tehtävälista koostuu projektin sidosryhmien jäsenten luomista käyttäjätarinoista, jotka ovat periaatteeltaan hyvin samanlaisia PXP:n käyttäjätarinoiden kanssa.

4.3 Tiedon kerääminen ja kerätyn tiedon käyttäminen

PXP ja Cowboy eivät määrittele työkaluja tiedon keräämiselle. Agile Solo suosittelee, että ennen jokaista iteraatiota sen sisältämien tehtävien ajankäytöstä tehdään arvio, joka kirjataan ylös. Etukäteen laadittuja arvioita verrataan toteumaan ja sen perusteella pyritään parantamaan tulevaisuudessa tehtäviä arvioita.

SIP:ssä kehittäjä pitää kirjaa sekä tehtäviin käytetystä ajasta että ohjelmistoon syntyneistä virheistä, joiden perusteella tulevaisuuden tehtäviä arvioidaan. SIP:ssä on käytössä myös iteraation tehtävälistan taulukko, joka toimii samalla periaatteella kuin edellä mainittu Agile Solon aikaseuranta.

4.4 Asiakassuhde

PXP:ssä perii XP:ltä paikalla oleva asiakas -käytännön, jota kuitenkin on muokattu sellaiseen muotoon, että sähköposti- tai puhelinyhteys asiakkaaseen on riittävä, kunhan asiakas on avoin viestintään. Tarvittaessa asiakas osallistuu suunnittelupeli-vaiheeseen.

Jo mainittujen viikkopalaverien lisäksi Agile Solossa toimiva ohjelmisto toimitetaan asiakkaalle testattavaksi kerran kuukaudessa. Asiakas voi antaa ohjelmistosta kommentteja, joita käytetään seuraavien iteraatioiden suunnittelussa.

Cowboyssa asiakas tai asiakkaalle nimetty edustaja osallistuu kaikkiin palavereihin. Asiakasta varten laaditaan kysely prosessin etenemisestä, johon asiakas vastaa jokaisessa versionarviointipalaverissa. SIP:ssä asiakas on mukana ainoastaan käyttäjätarinoiden laatimisessa.

Tässä luvussa itsenäisen kehittäjän ketteriä menetelmiä vertailtiin neljän osa-alueen perusteella toisiinsa. Seuraavassa taulukossa (Taulukko 4) on esitetty tiivistettynä tässä luvussa tehdyt havainnot kustakin menetelmästä ja osa-alueesta.

Taulukko 4: Itsenäisen kehittäjän menetelmien vertailu PSP:n toimintatapojen perusteella

	PXP	**Agile Solo**	**Cowboy**	**SIP**
Määritellyn prosessin noudattaminen	- Tarkat toimintaohjeet - XP:n käytännöt	- Käytännöt muista ketteristä menetelmistä	- Käytännöt muista ketteristä menetelmistä	- SC-prosessi
Työn suunnittelu	- Suunnittelupeli - Käyttäjätarinat	- Viikoittaiset palaverit	- Tavoite- ja tehtävälistat - Palaverit	- Tuotteen tehtävälista - Käyttäjätarinat
Tiedon kerääminen ja kerätyn tiedon käyttäminen	- Ei määritelty	- Ajankäytön seuranta	- Ei määritelty	- Ajan seuranta - Virheiden seuranta - Iteraation tehtävälistan taulukko
Asiakassuhde	- Sähköposti- tai puhelinyhteys riittävä - Suunnittelupeli	- Viikkopalaverit - Kuukausittaiset julkaisut	- Palaverit - Kysely	- Käyttäjätarinoiden laatiminen

5 YHTEENVETO

Tässä tutkielmassa tutustuttiin itsenäiselle ohjelmistokehittäjälle tarkoitettuihin ketteriin menetelmiin. Luvussa 2 tarkasteltiin ketterää kehitystä yleisesti sekä luotiin katsaus kahteen ketterän kehityksen menetelmään, Scrumiin ja Extreme Programmingiin. Niiden lisäksi esiteltiin yksittäisen kehittäjän tarpeisiin suunniteltu Personal Software Process.

Luvussa 3 pyrittiin löytämään vastaus tutkimuskysymykseen ”millaisia ketteriä menetelmiä itsenäiselle ohjelmistokehittäjälle on olemassa?”, ja esiteltiin neljä menetelmää: Personal Extreme Programming, Agile Solo, Cowboy ja Solo Iterative Process. Luvussa 4 etsittiin vastausta tutkimuskysymykseen ”miten itsenäiselle kehittäjälle tarkoitetut menetelmät eroavat toisistaan?”, ja vertailtiin aiemmin esiteltyjä menetelmiä keskenään PSP:n toimintatapojen avulla.

Itsenäisen kehittäjän menetelmien vertailussa havaittiin, että menetelmät poikkeavat toisistaan paikoitellen suuresti. Agile Solo ja Cowboy ovat vertailluista menetelmistä keskenään samankaltaisimmat, sillä molemmat on rakennettu olemassa olevia ketteriä menetelmiä silmällä pitäen. Suurin ero Agile Solon ja Cowboyn välillä on se, että Cowboy ei tarjoa mitään työkaluja tiedon keräämiseen ja analysointiin.

PXP ja SIP sisältävät niin ikään runsaasti yhtäläisyyksiä. Molemmat menetelmät sisältävät tarkan kuvauksen prosessin etenemisestä, jota kehittäjän tulee noudattaa, ja molemmat käyttävät iteraatioiden suunnittelussa hyödyksi käyttäjätarinoita. Poikkeuksellisesti PXP ei esittele minkäänlaisia keinoja tiedon keräämiseen, siinä missä SIP:n tiedon keräys- ja analysointitavat ovat vertailujoukon kattavimmat. Vertailluista menetelmistä PXP ja SIP ottavat vähiten kantaa asiakas-kehittäjä-yhteistyöhön, kun taas Agile Solossa ja Cowboyssa asiakkaan osallistumista korostetaan useasti.

Käsitellyn aineiston perusteella itsenäisen kehittäjän kannattaa omaksua jokin menetelmä kehityksensä tueksi. Menetelmän valintaan vaikuttaa toimintaympäristö sekä kehittäjän omat mieltymykset. Kurinalaisen ja tarkkoja ohjeita suosivan kehittäjän kannattaa kokeilla PXP:n tai SIP:n seuraamista kehitystyössään, kun taas ketterämpiä arvoja suosiva kehittäjä voi tuntea olonsa kotoisammaksi Agile Solon tai Cowboyn kaltaisen menetelmän parissa. Mikään

menetelmä ei ole sellaisenaan täydellinen ja on erittäin todennäköistä, että alati muuttuva ohjelmistokehityksen toimintaympäristö pakottaa soveltamaan jotakin määrättyä käytäntöä hieman poikkeuksellisesti.

Tulevaisuudessa aiheesta voisi tutkia esimerkiksi esiteltyjen menetelmien käytöstä saatuja kokemuksia, tai toteuttaa tapaustutkimuksen jollakin menetelmistä. Aihetta voisi myös laajentaa koskemaan yksittäisen kehittäjän roolia projektitiimin sisällä, ja tämän mahdollisia määriteltyjä työtapoja.

LÄHTEET

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. *VTT Publications 478.*

Agarwal, R., & Umphress, D. (2008). Extreme programming for a single person team. *Proceedings of the 46th Annual Southeast Regional Conference on XX,* 82-87.

Agile Alliance. (2015). What is agile software development? Haettu osoitteesta https://www.agilealliance.org/agile101/what-is-agile/.

Akpata, E., & Riha, K. (2004). Can extreme programming be used by a lone programmer? *Systems Integration,* 167.

Beck, K. (1999). Embracing change with extreme programming. *Computer, 32*(10), 70-77.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Jeffries, R. (2001). *The Agile Manifesto.*

Cronin, G. (2003). eXtreme solo: A case study in single developer eXtreme programming. *University of Auckland.*

Dorman, C. (2011). An experience report of the solo iterative process. *Wayne State University.*

Dorman, C., & Rajlich, V. (2012). Software change in the solo iterative process: An experience report. *Agile Conference (AGILE), 2012,* 21-30.

Dzhurov, Y., Krasteva, I., & Ilieva, S. (2009). Personal extreme Programming–An agile process for autonomous developers. *Sofia University.*

Hayes, W., & Over, J. W. (1997). The Personal Software Process (PSPSM): An Empirical Study of the Impact of PSP on Individual Engineers. *Technical Report, Carnegie Mellon University.*

Hollar, A. B. (2006). Cowboy: An agile programming methodology for a solo programmer. *Virginia Commonwealth University.*

Humphrey, W. S. (2000). The personal software process (PSP). *Technical Report, Carnegie Mellon University.*

Kähkönen, M. (2014). Itsenäisen ohjelmistokehittäjän ketterät menetelmät. *Pro gradu -tutkielma, Tampereen Yliopisto.*

Nyström, A. (2011). Agile solo-defining and evaluating an agile software development process for a single software developer. *Master of Science Thesis in Software Engineering and Technology, Chalmers University of Technology.*

Pham, A., & Pham, P. (2012). Scrum in action: Agile software project management and development. *Boston, MA: Course Technology.*

Rönkkö, M., & Peltonen, J. (2012). Software industry survey 2012. *Aalto Univ. School of Science.*

Rumpe, B., & Schröder, A. (2014). Quantitative survey on extreme programming projects. *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Italy,* 95-100, 2002.

Schwaber, K. (1997). Scrum development process. *Business object design and implementation*, 117-134, Springer.

Shen, M. J., Rong, G. P., & Shao, D. (2013). Integrating PSP with agile process: A systematic review. *Advanced Materials Research, 765* 1697-1703.

Stark, J. A., & Crocker, R. (2003). Trends in software process: The PSP and agile methods. *Software, IEEE, 20*(3), 89-91.

West, D., Grant, T., Gerush, M., & D'silva, D. (2010). Agile development: Mainstream adoption has changed agility. *Forrester Research, 2*, 41.