Itsenäisen ohjelmistokehittäjän ketterät menetelmät

Mika Kähkönen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Maaliskuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Mika Kähkönen: Itsenäisen ohjelmistokehittäjän ketterät menetelmät
Pro gradu -tutkielma, 42 sivua, 4 liitesivua
Maaliskuu 2014

Ohjelmistotuotannon ketterät menetelmät pyrkivät perinteisiä malleja nopeampiin tuloksiin pienentämällä kynnystä reagoida muutoksiin. Useimmiten nämä menetelmät on tarkoitettu ryhmässä sovellettaviksi, mutta joitakin menetelmiä on sovitettu myös itsenäisen kehittäjän tarpeisiin.

Tutkielma on systemaattinen kirjallisuuskatsaus, jossa etsitään tarkoilla hakulausekkeilla tietojenkäsittelyn tietokannoista ja verkosta tutkimusartikkeleita, valitaan niistä itsenäisen ohjelmistokehittäjän malleja esittelevät tutkimukset, analysoidaan valitut ja lopuksi kootaan niistä synteesi.

Soolomalleja on kolmenlaisia: 1) henkilökohtaisen ohjelmistoprosessin ja ketterän menetelmän yhdistävät, 2) yhtä ketterää menetelmää käyttävät tai useampaa yhdistävät, mutta niistä ryhmäkäytännöt poistavat ja 3) näistä erillinen menetelmä Solo Iterative Process. Jokaisesta ryhmästä esitellään yksi menetelmä laajemmin.

Menetelmissä toistuvat itsenäiselle kehittäjälle hyväksi havaittuina käytäntöinä muun muassa iteratiivisuus, vaatimussuunnittelu asiakkaan kanssa, visualisointi, tuotteen kehitysjono, töiden kohtuullistaminen, yksikkötestaukset ja versionhallinta. Ohjelmistokehittäjä saa malleista selkärankaa omalle työlleen.

Avainsanat ja -sanonnat: ohjelmistotuotanto, ketterä kehitys, agile, XP, Scrum, Kanban, henkilökohtainen ohjelmistoprosessi, PSP, PXP, Agile Solo, SIP.

Sisällysluettelo

1. Johdanto....1
2. Ohjelmistotuotanto ja ketterä kehitys....3
    2.1. Ketteriä menetelmiä....4
        2.1.1. XP....7
        2.1.2. Scrum....8
        2.1.3. Getting Real ja AUP....9
    2.2. Kanban....10
    2.3. Personal Software Process....11
3. Tutkimusmenetelmä....15
    3.1. Tutkimuskysymys....15
    3.2. Kevyt järjestelmällinen kirjallisuuskatsaus....15
    3.3. Haun määrittely ja rajaus....17
    3.4. Laadun arviointi....19
    3.5. Analyysi ja synteesi....19
4. Tulokset....21
    4.1. Haku....21
    4.2. Katsaus....24
    4.3. PSP-menetelmät....30
5. Kolme itsenäisen kehittäjän menetelmää....31
    5.1. PXP....31
    5.2. Agile Solo....34
    5.3. Solo Iterative Process....35
6. Pohdinta....38
Viiteluettelo....39
Liite 1. Tiedonkeruulomake 1....43
Liite 2. Tiedonkeruulomake 2....44
Liite 3. Laadunarviointilomake....45
Liite 4. Verkkosivuja itsenäisen kehittäjän menetelmistä....46

1. Johdanto

Vuonna 2010 ohjelmistoalalla työskenteli Suomessa 4184 henkilöä 4087:ssä alle neljän hengen yrityksessä, joten suurin osa näistä oli yhden hengen yrityksiä. Nämä yksityisyritykset muodostivat noin 10 % Suomen ohjelmistoyrityksistä. Myös ohjelmistoalan liikevaihdosta noin 10 % koostui yhden hengen yrityksistä. [Rönkkö and Peltonen, 2012] Ketterän kehityksen menetelmät (engl. *agile methods*), joiden tarkoitus on nopeuttaa ja tehostaa ohjelmistokehitystä, on usein kehitetty ryhmän toiminnan tehostamiseen. Ne eivät siis sellaisenaan hyödytä sovelluskehittäjiä, jotka työskentelevät itsenäisesti.

Tämän tutkielman tarkoitus on löytää itsenäisen kehittäjän ohjelmistotuotantoon sovellettuja ketteriä menetelmiä. Tutkielman tulosten perusteella yksityisyrittäjänä toimivat sovelluskehittäjät voivat tutustua erilaisiin malleihin ja etsiä lisätietoa niistä.

Itsenäinen ohjelmistokehittäjä ei ole yksiselitteinen käsite. Siksi on ensin tarpeen rajata ja määritellä tutkielman tarkoittama kohde. Itsenäiseksi kehittäjäksi voidaan ajatella esimerkiksi:

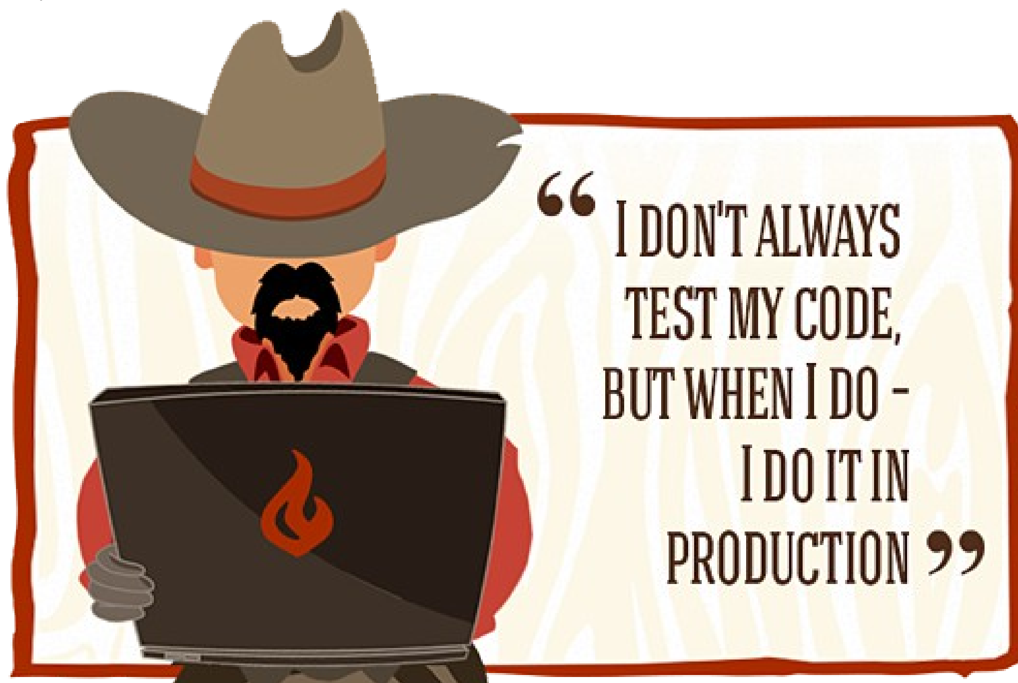
1. *Ryhmän jäsen*, jonka tehtäviin kuuluu paljon itsenäistä työtä, mutta koodi, jota tuotetaan, on yhteinen.
2. *Ohjelmistoalan yksityisyrittäjä*, joka vastaa sovelluksen kehittämisestä itse. Yhteistyökumppanina on asiakas. Tämä tyyppi voi olla myös toiselle yritykselle itsenäistä projektia tekevä alihankkija.
3. *Ohjelmistoprojekteja itsenäisesti tekevä ohjelmoija*, joka kehittää tuotetta ilman projektia tilannutta asiakasta joko itselle tai markkinoille myyntiin tai avoimena lähdekoodina ilmaiseksi. Tämä vastaa edellistä tyyppiä, mutta asiakkaan puuttuessa ulkoisia aikarajoja ei ole.

Akpata ja Riha [2004] käyttävät samantyyppistä jaottelua. Tässä tutkielmassa itsenäisellä kehittäjällä tarkoitetaan tyyppejä 2 ja 3. Heillä voi olla alihankkijoita joissakin ohjelmistokehityksen osa-alueissa kuten graafisessa suunnittelussa ja varsinkin pelialalla esimerkiksi äänitehosteissa, mutta ohjelmointityön luonne on itsenäinen. Tutkielma käsittelee ensisijaisesti tyypin 2 kehittäjiä.

Aiheesta on tehty jonkin verran tutkimusta. Henkilökohtaiseen kehittymiseen tarkoitetusta henkilökohtaisesta ohjelmistoprosessista (engl. *Personal Software Process, PSP*) tutkimuksia on paljonkin. PSP:tä ketteriin menetelmiin yhdistävistä tutkimuksista tuorein on Shenin ja muiden [2013] järjestelmällinen kirjallisuuskatsaus. Muita itsenäiselle kehittäjälle sovellettuja ketteriä menetelmiä on kehitetty ja tutkittu melko

vähän. Osa tutkimuksista on ilmestynyt vertaisarvioiduissa lehdissä ja osa on opinnäytetöitä. Muita soolomenetelmiä kuin ketteriä ei juuri ole tutkittu.

Ohjelmistotuotanto perustuu useimmiten ryhmätyöhön. Itsenäisiä kehittäjiä saatetaan jopa pitää haitallisina ”cowboy-koodareina”, jotka eivät piittaa hyväksi havaituista menetelmistä ja ampuvat lonkalta eli tuottavat suunnittelematonta, heikkolaatuista koodia. [Noel, 2004; Moweble, 2013] Kuva 1 korostaa asennetta, jolla cowboy-koodareihin suhtaudutaan.



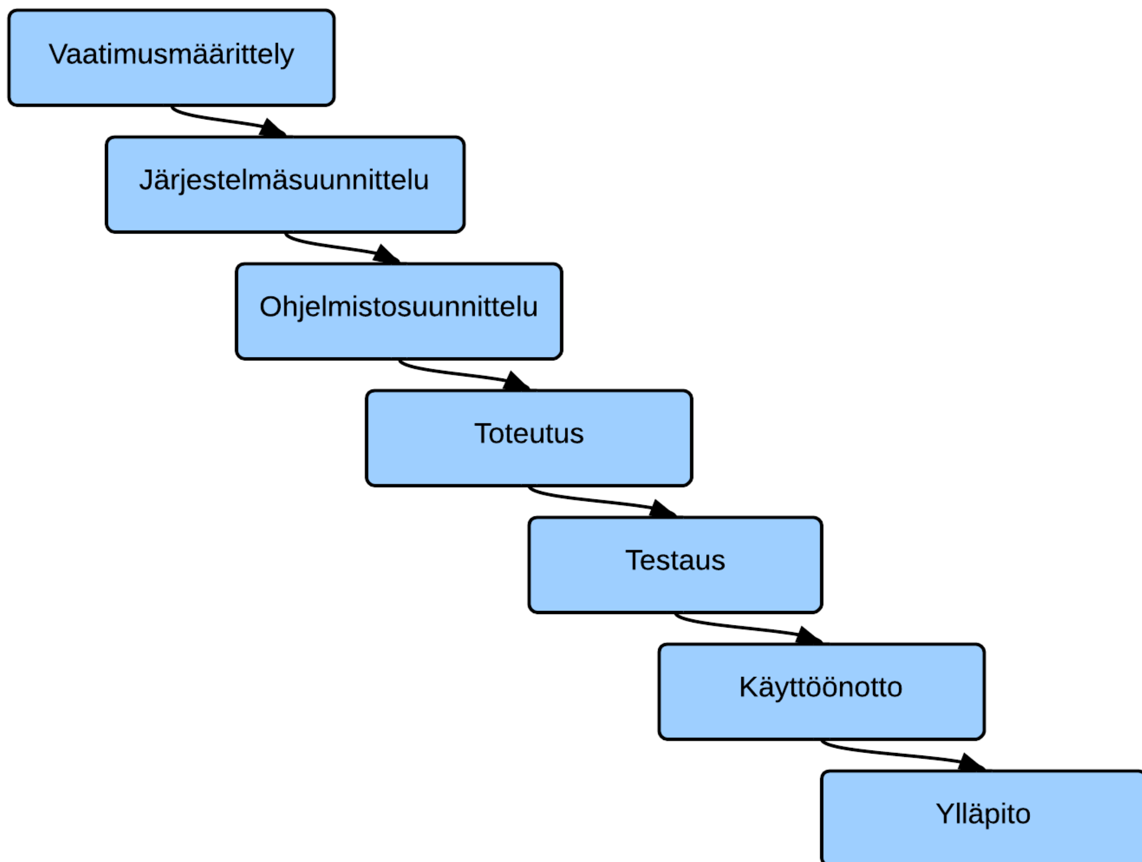
Kuva 1. Cowboy-koodareita pidetään hyvistä käytännöistä piittaamattomina [Peteva, 2013].

Toisaalta Curtisin [2001] mukaan cowboy on itsenäinen asiantuntija, joka johtajista välittämättä tekee niin kuin kokemuksensa mukaan on parasta. Tällöin muiden on sopeuduttava. Myönteisestä näkökulmasta katsoen itsenäisinä cowboy-koodariheeroksia olivat muun muassa Linus Torvalds *Linuxin* ensimmäisiä versioita kehittäessään ja *IRC*:n keksijä Jarkko Oikarinen. Nämä kaksi näkemystä, konna ja sankari, saavat olla pontimena, kun nyt etsitään itsenäisen kehittäjän ketteriä menetelmiä.

Tutkielman aluksi esitellään luvussa 2 ohjelmistotuotannon ja ketterän ohjelmistokehityksen peruskäsitteet ja yleisimmät mallit. Luvussa 3 selostetaan järjestelmällisen kirjallisuuskatsauksen tutkimusmenetelmä: millaisia ovat katsaukseen valikoituneiden tutkimusten haku- ja valintakriteerit sekä millaisella menetelmällä tutkimukset analysoidaan [Kitchenham et al., 2007]. Kirjallisuuskatsauksen tulokset käydään läpi luvussa 4. Luvussa 5 esitellään kolme soolomenetelmää ja luvussa 6 arvioidaan tämän tutkielman luotettavuus ja tehdään johtopäätökset.

2. Ohjelmistotuotanto ja ketterä kehitys

Ohjelmistotuotanto on prosessi, joka koostuu eri vaiheista (puhutaan myös ohjelmiston elinkaaresta). Yksi tapa jaotella vaiheet on *vaatimusmäärittely* (mitä ohjelmiston pitää pystyä tekemään), *järjestelmäsuunnittelu* (miten laitteet ja ohjelmistot liittyvät toisiinsa), *ohjelmistosuunnittelu* (toiminnallisessa määrittelyssä mitä ohjelmistolla voi tehdä ja arkkitehtuurisuunnitelmassa minkälainen rakenne ohjelman toteutuksella on), *toteutus* (ohjelmointi), *testaus* (virheiden etsintä ja sen tarkistaminen, tekeekö ohjelma mitä siltä vaaditaan) ja *käyttöönotto* (asennus ja käyttökoulutus). Käyttöönoton jälkeen on *ylläpitovaihe*, jossa voidaan joutua korjaamaan virheitä ja päivittämään uusia versioita. Ohjelmistokehitystä, jossa nämä vaiheet suoritetaan peräkkäin aina edellinen vaihe valmiiksi saattaen, kutsutaan *vesiputousmalliksi*. [Sommerville, 2008]



Kuva 2. Vesiputousmallin mukainen ohjelmistokehitys [Sommerville, 2008].

Kuvassa 2 nähdään vesiputousmallin mukainen ohjelmistotuotannon prosessi. *Ketterä ohjelmistokehitys* (engl. *agile software development*) nousi vastareaktioksi vesiputousmallin koetulle kankeudelle. Ketterä kehitys on yhteisnimitys erilaisille ohjelmistotuotantomalleille, joille yhteistä on nopea reagointi muutoksiin, suora viestintä ja pyrkimys saada aikaan toimiva ohjelmisto nopeasti. Toteutus on syklinen ja iteratiivinen: Syklin aikana ohjelmistotuotannon koko prosessi käydään läpi ja lopputuloksena on valmis tuote. Tärkeimmät ominaisuudet kehitetään ensin, seuraavalla iteraatiokierroksella lisätään ominaisuuksia tai korjataan edellisiä. Näin ohjelman keskeisimmät osiot saavat eniten huomiota. [Sommerville, 2008]

Iteraatio on itsessään pieni projekti: iteraatiolle tehdään suunnitelma ja ohjelmalle vaatimusmäärittely, suunnittelu, ohjelmointi, testaus ja dokumentointi. Iteraation tuloksena on toimiva versio ohjelmasta, joka on valmiimpi ja monipuolisempi kuin edellisen iteraation versio. [Sommerville, 2008]

2.1. Ketteriä menetelmiä

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.

Kuva 3. Ketterän ohjelmistokehityksen julistus [Beck et al., 2001a].

Ketterien mallien historia alkaa 1990-luvulta, jolloin tyytymättömyys raskasta etukäteissuunnittelua korostavaa vesiputousmallia kohtaan kasvoi. Kehitettiin useita menetelmiä, kuten Crystal-menetelmät [Cockburn, 2004], Dynamic Software Development Method (DSDM) [Stapleton, 2003], Feature-Driven Development [Palmer and Felsing, 2002], Lean Software Development [Poppendieck and Poppendieck, 2003],

Scrum [Schwaber and Beedle, 2001] ja Extreme Programming (XP) [Beck, 1999]. Kaikki nämä perustuvat iteratiivisiin kehityssykleihin, mutta ne toteuttavat idean eri tavoin. [Dybå and Dingsøyr, 2008; Sommerville, 2008]

Sana ”agile” tuli käyttöön varsinaisesti vasta ketterän julistuksen (engl. *Agile Manifesto*) myötä vuonna 2001. Joukko ohjelmistokehittäjiä kirjoitti julistuksen, jossa he määrittelivät ketterien menetelmien neljä perusarvoa (kuva 3).

Ketterät menetelmät ovat saaneet myös kritiikkiä. Pääasiassa kritiikki on keskittynyt viiteen asiaan: 1) ketteryys ei ole uutta, vaan vastaavia menetelmiä on ollut käytössä jo pitkään, 2) arkkitehtuurisuunnittelun jättäminen vähälle huomiolle ei tuota optimaalisia ratkaisuja ohjelmasuunnitteluun, 3) monilla ketterien menetelmien väitteillä on vain vähän tieteellistä pohjaa, 4) XP:n käytäntöjä toteutetaan harvoin kirjaimellisesti ja 5) ketterät menetelmät sopivat pienille ryhmille, kun taas suuriin projekteihin sopivat muunlaiset mallit. [Dybå and Dingsøyr, 2008] Myös Sommervillen [2008] mielestä ketterät menetelmät sopivat pieniin tai keskisuuriin projekteihin.

Dybå ja Dingsøyr [2008] toteuttivat järjestelmällisen kirjallisuuskatsauksen ketterien menetelmien käytöstä ja niiden eduista. Katsauksen päätulos oli, että etuja kyllä on, mutta tulosten tieteellinen näyttö oli heikko. Tästä syystä neuvoja ohjelmisto-kehittäjille on vaikea antaa. Lisäksi tutkimukset käsittelivät lähes pelkästään XP-mallia ja erityisesti projektinhallintamenetelmiä kuten Scrumia pitäisi tutkia enemmän. Myös uudempi katsaus osoitti, että ketteristä menetelmistä on etua, mutta tutkimusten validiteetissa esiintyneiden ongelmien vuoksi (esimerkiksi tutkija saattoi olla itse kehittäjänä tutkimassaan ohjelmistoprojektissa) lisätutkimusta tarvittaisiin ennen kuin tuloksia voisi yleistää [Sletholt et al., 2011].

Yleisimmät ketterät menetelmät ovat Crystal-menetelmät, DSDM, Feature-Driven Development, Lean Software Development, Scrum ja XP [Dybå and Dingsøyr, 2008]. Esittelen tarkemmin useimmissa soolomenetelmissä käytetyt Scrumin ja XP:n sekä Kanban-menetelmän ja henkilökohtaisen ohjelmistoprosessin. Lisäksi esittelen yhdessä soolomenetelmässä hyödynnettävät Getting Realin [37signals, 2006] ja AUP:n [Ambler, 2014].

Esittelyn pohjaksi luettelen ketterän kehityksen 12 perusperiaatetta [Beck et al., 2001b]:

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 lyhyempää 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.*

Sommerville [2008] tekee muutaman huomautuksen ketteristä periaatteista. Vaikka asiakkaan osallistaminen (periaate 4) on hieno ajatus, ei asiakkaalla välttämättä ole aikaa tai halua osallistua kehitystyöhön päivittäin. Samaan ongelmaan voi törmätä myös asiakkaalle työskentelevä itsenäinen kehittäjä. Toiseksi ryhmän jäsenten välinen kemia ei välttämättä toimi. Tämä ei ole itsenäiselle kehittäjälle mikään ongelma.

Kolmanneksi muutosten asettaminen tärkeysjärjestykseen voi olla hyvin hankalaa erityisesti usean yhteistyökumppanin kanssa, sillä jokaisella järjestys voi olla erilainen. Itsenäinen kehittäjä työskentelee pakostikin suppeampien projektien parissa, joten yhteistyökumppaneita ei välttämättä ole monta.

Neljänneksi yksinkertaisuuden säilyttäminen voi tuoda lisätyötä. Viidenneksi sopimusten kirjoittaminen hankaloituu. Syklisessä ja iteratiivisessa projektissa asiakkaalta tulee uusia vaatimuksia projektin edetessä. Kehittäjän tulee tarkoin miettiä, onko palkkaus työtuntien mukaan vai urakkapalkkana. Jälkimmäisessä lisävaatimukset tuovat haasteita: mitkä vaatimukset kuuluvat alkuperäiseen palkkaan ja mistä pitää maksaa lisää palkkaa.

2.1.1. XP

Extreme Programming (XP) on kenties yleisimmin käytetty ketterä menetelmä. Nimi tulee siitä, että se ”vie äärimmilleen” (engl. *extreme*) menetelmien periaatteet kuten iteratiivisen kehityksen ja asiakkaan osallistumisen. [Sommerville, 2008]

XP:ssä vaatimukset kootaan käyttäjätarinoina, joista toimitetaan tehtävälistat (engl. *series of tasks*). Jokaista tehtävää varten tehdään testi, ja koodin pitää läpäistä nämä testit ennen kuin se voidaan lisätä valmiiseen järjestelmään. Kehityksen aikana sovelluksesta julkaistaan versioita lyhyin väliajoin. [Sommerville, 2008]

XP perustuu seuraaville 12 käytännölle [Sommerville 2008; Agarwal and Umphress, 2008]:

1. Suunnittelupeli
2. Läsnä oleva asiakas
3. Pienet julkaisut
4. Yksinkertainen suunnittelu
5. Koodin refaktorointi
6. Yhteinen omistajuus
7. Yhteiset ohjelmointistandardit
8. Testivetoinen kehitys
9. Pariohjelmointi
10. Jatkuva integraatio
11. Sopiva työteho
12. Vertauskuva.

Suunnittelupelissä (engl. *The Planning Game*) ryhmä suunnittelee tuotetta yhdessä asiakkaan kanssa. Iteraatiokierroksen asiakas osallistuu kehitykseen siten, että hän keskustelee käyttäjätarinoista ryhmän kanssa ja kirjoittaa kustakin tarinasta kortin. Ryhmä jaottelee nämä tehtäviksi, arvioi työmäärän ja toteutuksen vaatimat resurssit. Sen jälkeen asiakas asettaa toteutettavat tarinat tärkeysjärjestykseen. Jos ja kun vaatimukset muuttuvat, tehdään uusi tarinakortti, ja asiakas arvioi sen tärkeyden suhteessa aiempiin. Vaatimuksia voidaan myös tarvittaessa poistaa. [Sommerville, 2008]

Asiakkaan edustajan pitää olla jatkuvasti XP-ryhmän käytettävissä. *Läsnä oleva asiakas* on osa kehitysryhmää ja on vastuussa vaatimusten esittämisestä.

Pienet julkaisut tarkoittaa, että iteraatiot ovat lyhyitä: ensin toteutetaan pienin mahdollinen määrä tärkeimpiä ominaisuuksia, sitten iteratiivisesti lisätään ominaisuuksia. *Yksinkertaisella suunnittelulla* korostetaan sitä, että suunnittelua tehdään vain kulloinkin toteutettavia ominaisuuksia varten. Jos mieleen tulee useampia vaihtoehtoja, valitaan yksinkertaisin. Laajeneva järjestelmä pidetään koossa muuttamalla koodia ymmärrettävämmäksi (*refaktorointi*).

Jokaisen odotetaan refaktoroivan koodia tarpeen mukaan. Kaikki koodi on *yhteisesti omistettua*. Siksi myöskin jokaisen pitää toimia *yhteisten ohjelmointistandardien* mukaan. Mistään koodin osasta ei pidä tunnistaa, kenen tekemää se on.

Kehitys toteutetaan *testivetoisesti*. Ennen ohjelmointia kirjoitetaan yksikkötestit, jotka ohjelman tulee läpäistä. Yli-Rohdaisella [2012] on hyvä yleisesitys yksikkötesteistä. Itse ohjelmoinnin tekee aina kaksi ohjelmoijaa yhdessä *pariohjelmointina*. Toinen kirjoittaa, toinen seuraa vierestä, antaa tukea ja arvioi. Välillä vaihdetaan vuoroa. Kun tehtävä on valmis, yksikkötestit suoritetaan ja koodin pitää läpäistä ne ennen kuin se voidaan *integroida* järjestelmään. *Sopiva työteho* tarkoittaa, että ylitöitä ei suvaita, sillä niiden tuloksena on heikkoa koodia.

Helpottaakseen viestintää projekti pitää yllä listaa nimityksistä ja niiden selityksistä. Tätä kutsutaan *vertauskuvaksi* (engl. *metafora*).

2.1.2. Scrum

Scrum on projektinhallinnan ketterä malli. Sen keskeisiin piirteisiin kuuluu tuotteen kehitysjono (engl. *product backlog*), neljän viikon kehityssyklit eli *sprintit*, päivittäiset palaverit ja ryhmän jäsenille määritellyt roolit. Kehitysjonoon lisätään tärkeysjärjestykseen ne vaatimukset, joita kehitysryhmän pitää tehdä. Vaatimuksissa kuvataan, mitä pitäisi toteuttaa sen sijaan, että kerrottaisiin, kuinka pitäisi toteuttaa. Jokaisen sprintin jälkeen on tarkoitus esitellä toimiva versio yhteistyökumppaneille. [Scrum, 2014; James, 2014]

Scrumin *kehitysryhmä* on vastuussa tuotteesta. Ryhmä koostuu 3–9 työntekijästä, joilla on erilaisia osaamis- tai vastuualueita (esimerkiksi analysointi, suunnittelu, kehitys, testaus ja dokumentointi).

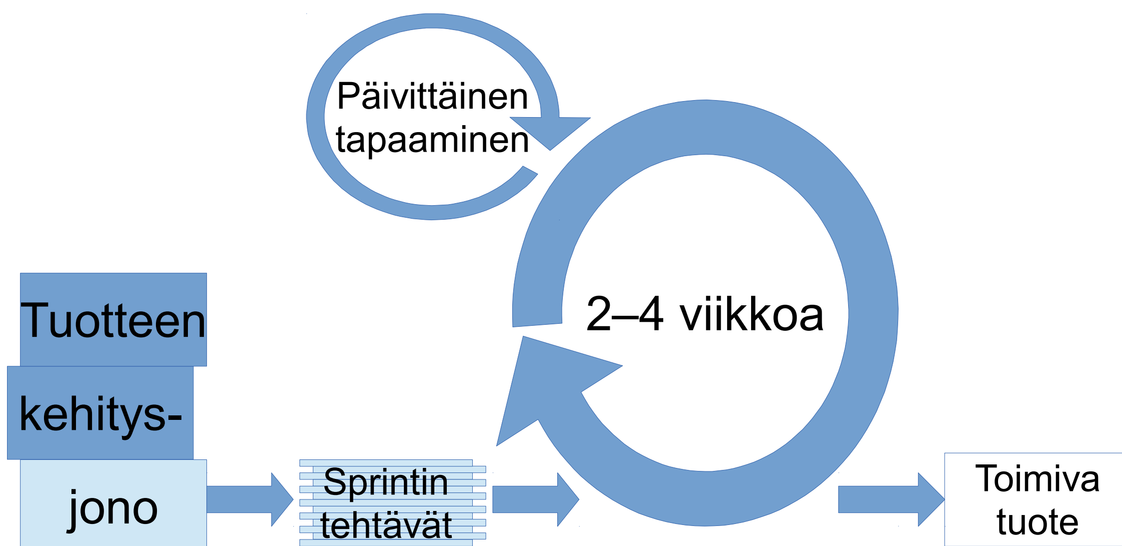
Scrum-ryhmässä on yksi *tuoteomistaja* (engl. *Product Owner*). Hän edustaa asiakasta ja varmistaa, että tuotteesta on asiakkaalle etua. Tuoteomistaja kirjoittaa käyttäjätarinoita (lyhyitä virkkeitä siitä, mitä erityyppiset käyttäjät tuotteella haluavat tehdä), priorisoi ne ja lisää ne tuotteen kehitysjonoon. Priorisointi perustuu esimerkiksi riskiin, taloudelliseen arvoon, riippuvuuksiin ja aikarajoihin.

Yksi ryhmäläisistä on *Scrum Master*. Hänen tehtävänään on poistaa esteitä, jotta ryhmä saavuttaisi tavoitteensa. Scrum Master huolehtii, että ryhmä toteuttaa Scrumia esimerkiksi toimimalla puheenjohtajana tapaamisissa. Hän ei ole projektinjohtaja, vaan johtamiseen liittyvät tehtävät on jaettu kehitysryhmän kanssa tai tuoteomistaja huolehtii niistä.

Kehitys tapahtuu iteratiivisesti ja iteraatiokierrosta eli kehityssykliä nimitetään sprintiksi, joka kestää yhdestä neljään viikkoa. Sprintti alkaa suunnittelukokouksella, joka päivä pidetään tapaaminen ja lopuksi pidetään sprinttikatselmus. Sprintin jälkeen tuotteen on tarkoitus olla toimiva, testattu ja dokumentoitu (kuva 4).

Suunnittelukokouksessa ryhmä valitsee, mitkä kehitysjonon vaatimuksista toteutetaan sprintin aikana. Vaatimukset voivat olla esimerkiksi ominaisuuksia, bugikorjauksia tai esimerkiksi dokumentointia. Vaatimuksiin kuluva aika arvioidaan ja vaatimus lisätään sprintin omaan kehitysjonoon (engl. *sprint backlog*). Vaatimus jaetaan tehtäviin (engl. *task*), jotka vastaavat siihen, kuinka vaatimus täytetään. Suunnittelukokous kestää yhden päivän.

Päivittäisissä Scrum-tapaamisissa (15 minuutin tapaaminen pidetään aina samaan aikaan samassa paikassa) jokainen kertoo, mitä on tehnyt edellisen tapaamisen jälkeen, mitä aikoo tehdä seuraavaksi ja onko jotain etenemisesteitä. Scrum Master dokumentoi esteet ja pyrkii tapaamisen jälkeen ratkaisemaan niitä.



Kuva 4. Scrum-kehitysmalli [James, 2014].

Sprinttikatselmuksessa (kestää 4 tuntia) ryhmä esittelee, mitä on saatu tehdyksi ja mitä suunniteltua jäi tekemättä. Tuotteen toimiva versio esitetään yhteistyökumppaneille ja kerätään palaute. *Sprintin arviointikokouksessa* jokainen ryhmäläinen arvioi, miten kulunut sprintti meni: mikä meni hyvin ja mitä pitäisi parantaa seuraavaan sprinttiin.

Ryhmä voi myös ylläpitää työmääräkuvaajaa (engl. *burn down chart*), josta näkee, miten paljon työmäärää on sprintissä tai vaihtoehtoisesti koko projektissa jäljellä.

2.1.3. Getting Real ja AUP

Getting Real on Basecamp-yhtiön (aiemmin 37signals, kehittänyt muun muassa Ruby on Rails -ohjelmistokehyksen) kehittämä ketterän kehityksen malli. Sen pääideana on keskittyä kaikkein olennaisimpaan ja tehdä paremmin tekemällä vähemmän. Getting Real kehottaa kehittäjiä vähentämään ominaisuuksia, antamaan asiakkaalle vähemmän

valintoja tekemällä päätöksiä heidän puolestaan, kieltäytymään lisäominaisuuksista ja tekemään nopeasti versio, jossa tärkeimmät ominaisuudet toimivat. Tuotetta testataan todellisilla (engl. *real*) käyttäjillä, jotta saadaan palautetta. [37signals, 2006]

Agile Unified Process (AUP) on Amblerin [2014] kehittämä yksinkertaistettu versio IBM:n Rational Unified Processista (RUP). Elinkaarimallinsa tueksi RUP tiivistää ohjelmistokehityksen kuuteen käytäntöön, joita ovat 1) iteratiivinen kehitys, 2) vaatimusten hallinta, 3) komponentteihin perustuva kehitys, 4) visuaalinen mallinnus, 5) jatkuva laadunvarmistus ja 6) muutosten kontrollointi. AUP yksinkertaistaa tätä muun muassa siten, että erilaisista tuotoksista (engl. *artifact*; esimerkiksi mallinnukset, dokumentaatio) tehdään vain riittävän hyviä, sillä niillä on tapana muuttua projektin edetessä.

2.2. Kanban

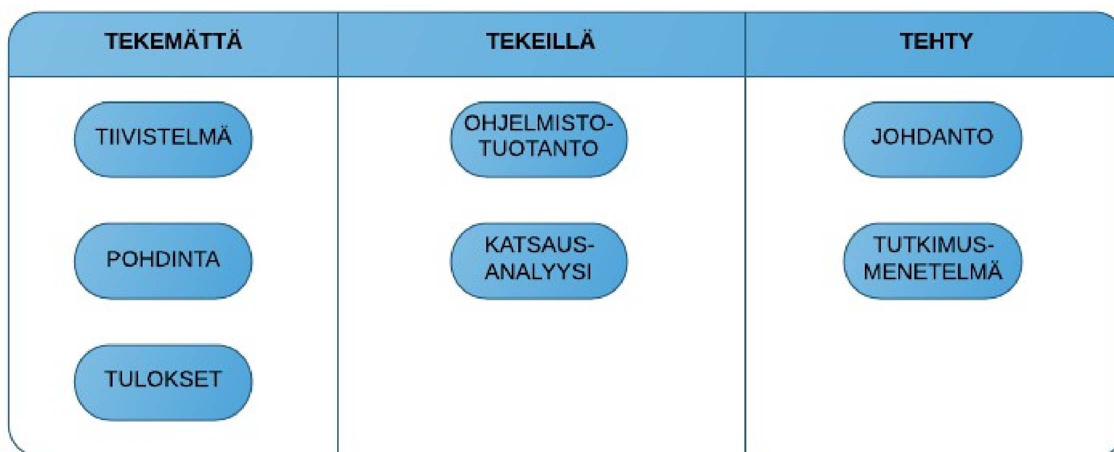
Kanban ei ole vain ohjelmistokehityksen menetelmä eikä varsinaisesti edes ketterä menetelmä ketterän julistuksen tarkoittamassa mielessä, vaan ajatusmalli, jolla organisaatiot voivat kehittää itseään. Kanbanin neljä periaatetta ovat [Kanban, 2014]:

1. Aloita sillä, mitä teet nyt. Kanban ei määrittele rooleja tai prosesseja, se pyrkii muutokseen.
2. Tee muutos vähitellen ja evolutiivisesti. Jatkuva, hiljalleen kehittyvä muutos on tapa, jolla saadaan aikaan pysyviä muutoksia.
3. Kunnioita nykyisiä prosesseja, rooleja, vastuita ja nimityksiä. Näin vältetään muutosvastarintaa.
4. Johtajuutta on kaikilla tasoilla. Sitä pitää rohkaista.

Periaatteista nähdään, että Kanban on todellakin filosofia, jolla pyritään muuttamaan organisaation toimintakulttuuria. Itsenäinen kehittäjä pystynee toteuttamaan muutoksen nopeammin, sillä uuteen opetettavia henkilöitä on vain yksi.

Periaatteiden lisäksi menestyvien Kanban-toteutusten on havaittu sisältävän kuusi ydinkäytäntöä [Kanban, 2014], joista ”visualisoi työn eteneminen” ja ”rajoita tekeillä olevia töitä” esiintyvät tämän tutkielman menetelmissä. Kolmas, ”hallitse töiden etenemistä”, pitää sisällään esimerkiksi tehtävien tekemiseen kuluvan ajan mittaamisen. Näin voi saada selville ja kehittää työvaiheita, joihin joutuu panostamaan enemmän. Kolme muuta käytäntöä liittyvät enemmänkin organisaation toimintaan ja ovat: ”tee käytännöistä näkyviä”, ”hanki palautetta työvaiheista” ja ”kehity yhteisesti, kehity kokemuksesta”.

Töiden rajoittaminen ja sujuvuuden ylläpitäminen hoidetaan Kanban-korteilla, joissa kuvataan työtehtävä. Kortit, jotka voivat olla vaikkapa tarralappuja, kiinnitetään Kanban-tauluun, joka on jaettu sarakkeisiin työn vaiheen mukaan: esimerkiksi ”odottaa toteutusta”, ”kehityksessä” ja ”valmis” (kuva 5). Vaiheita voi olla useampikin. Tekeillä olevien töiden määrälle asetetaan enimmäisraja. Näin visualisoidaan prosessin tila ja pystytään keskittymään sopivaan määrään tehtäviä. [Nyström, 2011]



Kuva 5. Esimerkki Kanban-taulusta graduntekoprosessiin sovellettuna.

2.3. Personal Software Process

Humphreyn [2000] kehittämä henkilökohtainen ohjelmistokehitysprosessi (engl. *Personal Software Process* eli PSP) on menetelmä sovelluskehittäjän ammattitaidon itsearviointiin ja parantamiseen. PSP:n avulla kehittäjä suunnittelee, mittaa ja hallinnoi työskentelyään toimintaohjeiden ja lomakkeiden avulla.

PSP:n periaatteiden mukaan kehittäjä [Humphrey, 2000]:

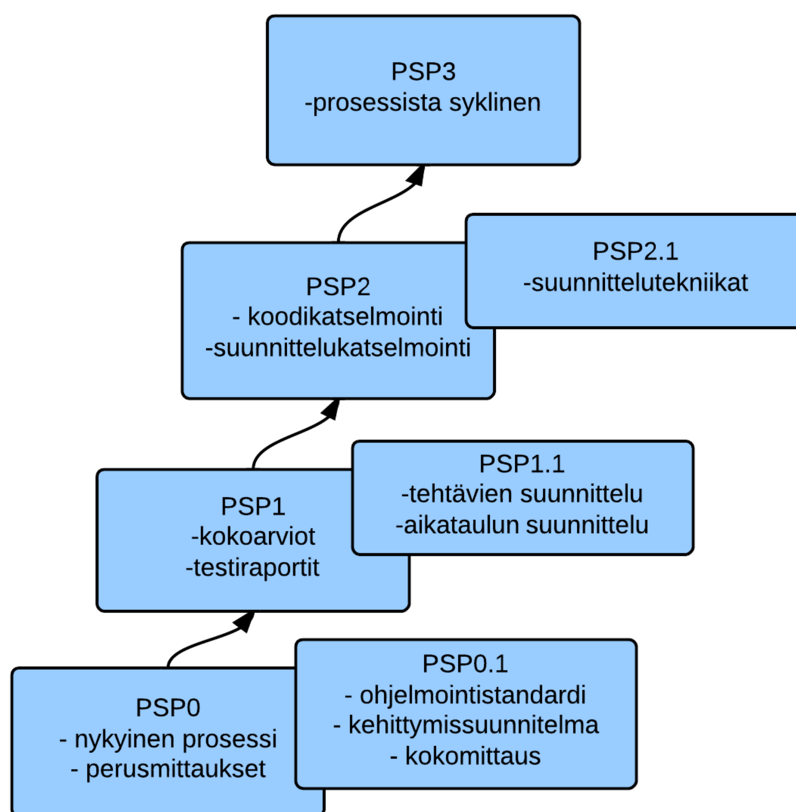
- toteuttaa määriteltyjä toimintaohjeita,
- on työhönsä sitoutunut,
- korjaa puutteet ajoissa tai mieluummin välttää ne ja
- perustaa työnsä itsestä mitattuun dataan, sillä jokainen kehittäjä on erilainen.

Humphreyn [2000] mukaan PSP on tarkoituksella kehitetty niin, että sitä voi käyttää muiden menetelmien kanssa yhdessä. Myös ketteriin menetelmiin sitä on sovellettu ja tutkimuksia aiheesta on muutamia. Shen ja muut [2013] ovat julkaisseet systemaattisen kirjallisuuskatsauksen PSP:n hyödyntämisestä ketterien menetelmien kanssa.

PSP:n mukaan pitää työt ensin suunnitella toimintaohjeiden mukaisesti. Jokaiseen vaiheeseen kulunut aika mitataan, huomatut puutteet ohjelmassa, niiden korjaustapa ja

valmistuneiden tuotteiden koko kirjataan ylös. Jotta prosessi tuottaisi laadukasta työtä, tuotteen laatua pitää suunnitella, mitata ja seurata. Laatuun pitää panostaa alusta asti. Lopuksi kehittäjän pitää analysoida tulokset ja käyttää löydöksiä itsensä kehittämiseen. [Humphrey, 2000]

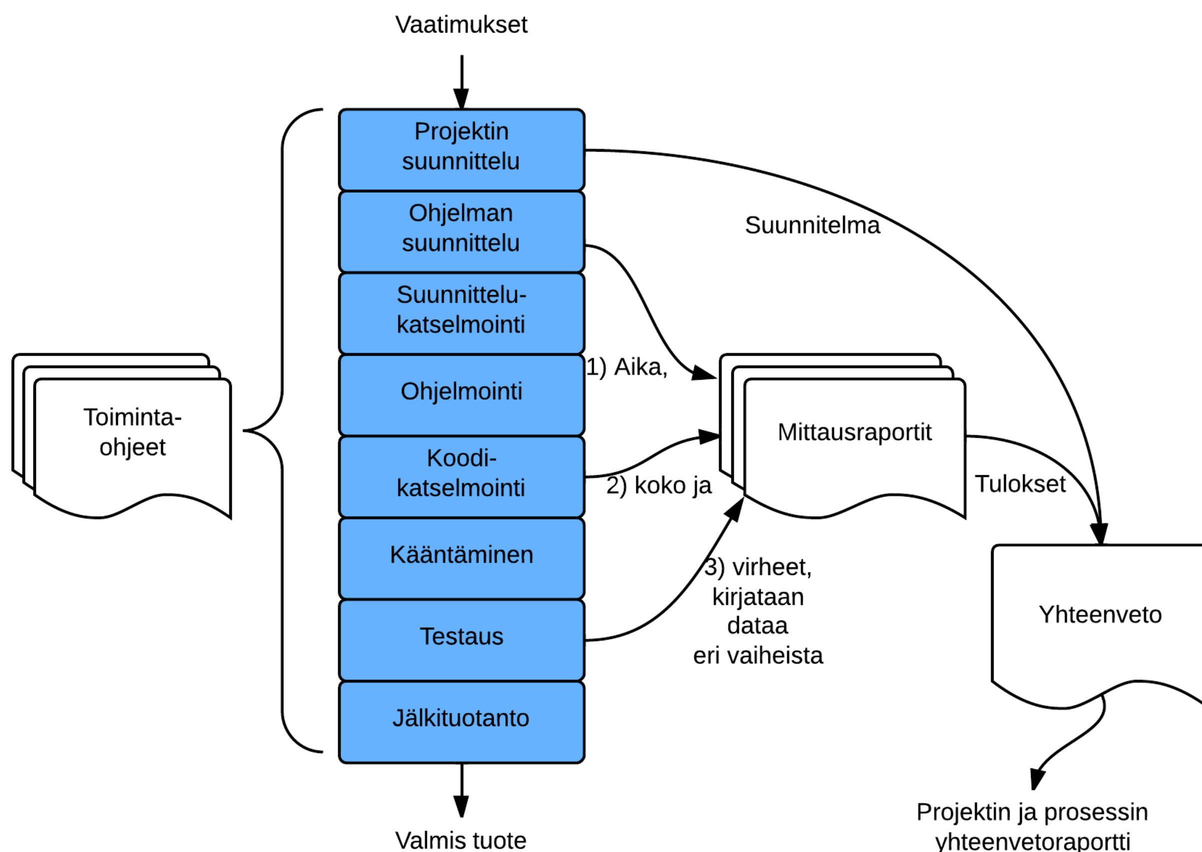
PSP:n käyttöönotto sisältää neljä eri tasoa, joista viimeisessä eli PSP3-tasossa prosessista tehdään syklinen. Tällainen vaiheistus on käytössä menetelmän kehittäneen Carnegie Mellon University -yliopiston Software Engineering Institute -tutkimus-keskuksen kurssilla. Opiskelijat tekevät useita projekteja ja ottavat aina seuraavassa projektissa käyttöön uuden PSP-tason. Kuvassa 6 esitetään tasot kaaviona. [Humphrey, 2000]



Kuva 6. PSP:n vaiheittainen opettelu [Humphrey, 2000].

Tasolla PSP0 opitaan prosessi ja mittaaminen. PSP-prosessissa on kolme vaihetta: työn suunnittelu, kehitys (suunnittelu, ohjelmointi, testaus) ja jälkituotantovaihe (engl. *post mortem*). Mittaukset tehdään tällä tasolla nykyiseen työtapaan kuluvasta ajasta ja havaituista ja korjatuista virheistä. PSP0.1 lisää ohjelmointistandardit, ohjelman koon mittaamisen ja henkilökohtaisen kehittymissuunnitelman. Jälkituotantovaiheessa mitataan ohjelman koko ja kerätään yhteen aiemmin mitattu data projektin yhteenvetoraporttiin. [Humphrey, 2000]

PSP1 tuo mukaan koon arvioimisen ja testiraportin. Arvion pohjana käytetään nollatason mittauksia. PSP1.1-tasolla tehtävät ja aikataulu suunnitellaan. Tasolla PSP2 oleva katselmoi koodinsa ja suunnitelmansa. Virheitä pyritään estämään ennalta tai poistamaan. Tasolla 2.1 otetaan mukaan suunnittelu- ja analysointitekniikoita. [Humphrey, 2000] Lopulta on opittu PSP-menetelmä, jota selventävä kaavio on kuvassa 7.



Kuva 7. PSP-prosessi [Humhprey, 2000].

Prosessin toimintaohjeet (engl. *scripts*) sisältävät tarkemmat ohjeet jokaisesta prosessin vaiheesta. Toimintaohjeissa voisi esimerkiksi lukea: ”käytä PROBE-menetelmää arvioimaan muutettavien koodirivien määrä”, ”hanki vaatimukset”, ”täytä koon arvioimisen lomake” ja ”suorita yksikkötestaus”. [Humphrey, 2000] Taulukossa 1 on Humphreytä [2000] mukaillen tehty esimerkki toimintaohjeista.

PROBE-menetelmällä (edustajiin perustuva arviointi, engl. *Proxy Based Estimating*) arvioidaan LOC (koodirivit, engl. *Lines of Code*). Aluksi arvioidaan tarvittavat ohjelmaluokat (proxyt, edustajat), sitten niihin tehtävien funktioiden määrä ja

tyyppi. Käyttämällä aiempaa tietoa samantyyppisistä funktioista saadaan arvioitua rivien määrä. [Humprey, 2000]

Taulukko 1. Esimerkki PSP-prosessin toimintaohjeista [Humphrey, 2000].

Tarkoitus	• Esimerkki.
Alkuehto	• Ongelman kuvaus. • Aiempi mittausdata. • Täyttämätön yhteenvetoraportti. • Koonarviointilomake.
Suunnittelu	• Hanki vaatimukset. • Käytä PROBE-menetelmää arvioimaan uusien ja muutettavien koodirivien määrä. • Täytä koonarviointilomake. • Arvioi kehitykseen kuluva aika. • Kirjaa kulunut aika.
Kehitys	• Suunnittele ohjelma. • Ohjelmoi ohjelma. • Testaa ohjelmaa ja korjaa (ja kirjaa lomakkeelle) virheet. • Kirjaa kulunut aika.
Jälkituotanto	• Täytä yhteenvetoraportti mitatulla datalla.
Lopetusehto	• Testattu ohjelma. • Yhteenvetoraportti täytetty arviointi- ja mittausdatalla.

3. Tutkimusmenetelmä

3.1. Tutkimuskysymys

Tutkielman tavoitteena on kartoittaa tutkimustietoa ketterien menetelmien hyödyistä ja haitoista yksittäisen kehittäjän tarpeisiin. Tarkoitus on ensinnäkin selvittää, millaisia ketterän kehityksen malleja on kehitetty yhden kehittäjän ohjelmistotuotantoon. Tähän sovelletaan järjestelmällisen kirjallisuuskatsauksen menetelmää, jolla löydetään joukko tutkimuksia tarkasteltaviksi.

Valituista ketteristä menetelmistä selvitetään, 1) mihin menetelmään ne perustuvat, 2) mitä keskeisiä piirteitä niissä on ja 3) onko niitä sovellettu käytännössä. Katsauksen perusteella karsitaan tarkempaan esittelyyn kolme eri malleihin perustuvaa menetelmää. Tässä esittelyssä menetelmistä kerrotaan, mihin ketterään malliin ne perustuvat, miten iteratiivisuus toteutetaan, millaisia muita ketteriä käytäntöjä niissä on ja miten asiakas huomioidaan. Tulosten pohjalta annetaan suosituksia itsenäiselle kehittäjälle.

3.2. Kevyt järjestelmällinen kirjallisuuskatsaus

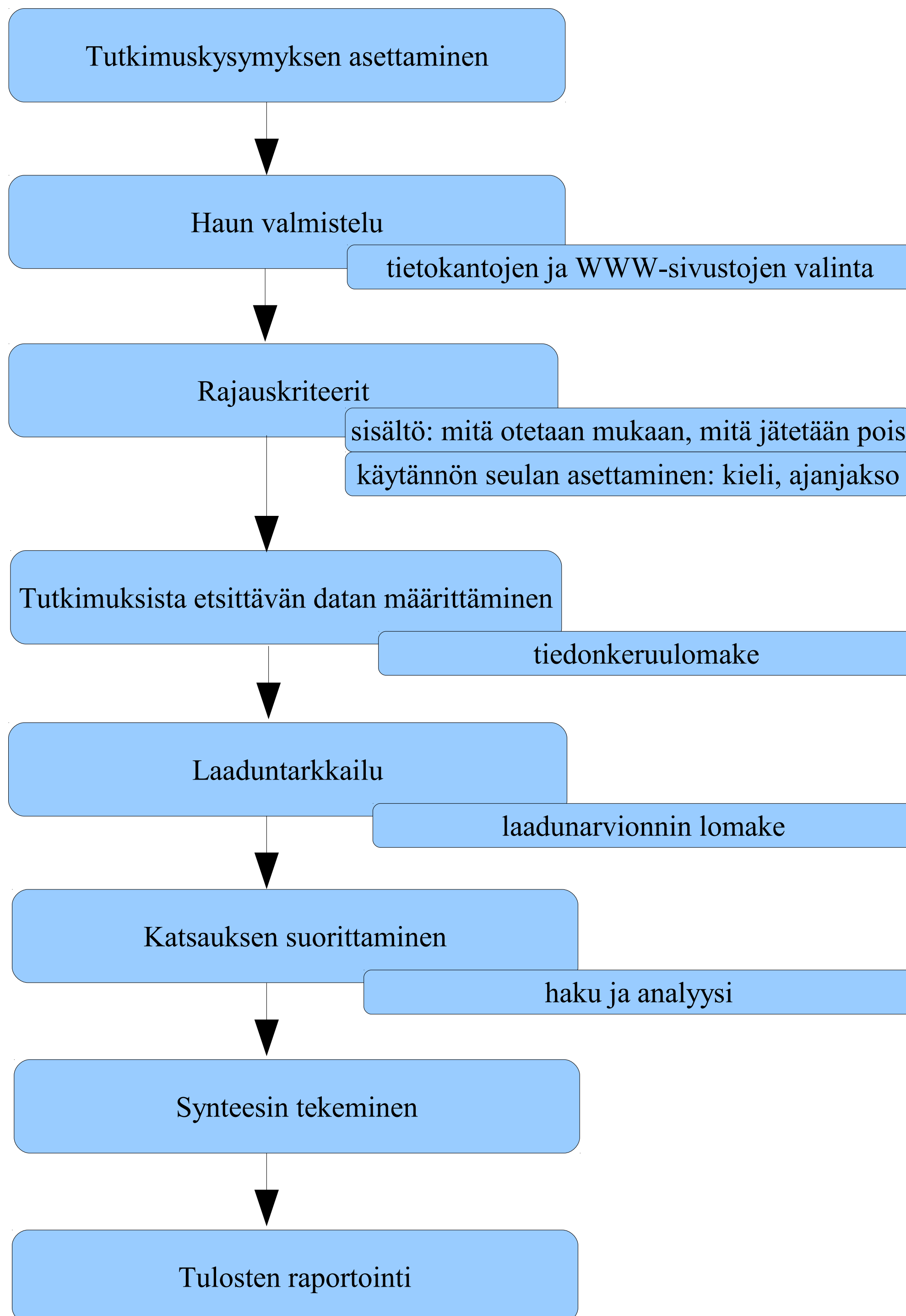
Järjestelmällisen kirjallisuuskatsauksen tarkoituksena on tunnistaa, arvioida ja tulkita kaikki saatavilla oleva alkuperäistutkimus aiheesta. Tieteellistä hyötyä katsauksesta on, jos se on kattava ja perusteellinen. Hyvin tehty katsaus on toistettavissa ainakin haun ja tutkimusten rajauksen osalta. [Kitchenham et al., 2007]

Kirjallisuuskatsaus alkaa menetelmien määrittämisellä: millä hakusanoilla tutkimuksia haetaan ja mistä, miten hakutulos rajataan ja millä arvioidaan tutkimuksen laatu, miten varsinainen katsaus analysoidaan ja miten tiedoista kootaan synteesi. Määritelmät elävät jonkin verran tutkimuksen edetessä. Esimerkiksi hakulausekkeiden tekeminen on iteratiivinen prosessi. [Kitchenham et al., 2007]

Jos ennen analysointia havaitaan, että aineistoa on vähän tai aihe on hyvin laaja, voi *järjestelmällinen kartoitustutkimus* (engl. *systematic mapping study* tai *scoping study*) olla parempi vaihtoehto. Kartoitus käyttää järjestelmällisen katsauksen menetelmiä, mutta siinä hakulausekkeet eivät ole niin tarkkoja ja antavat enemmän hakutuloksia, eikä analyysia tehdä niin syvällisesti. [Kitchenham et al., 2007]

Kartoituksessa tutkimukset luokitellaan ja tyypitellään tiivistelmien perusteella. Tarkoitus on saada laaja yleiskuva tutkimusalueesta: millaisia tutkimuksia on tehty ja toisaalta, onko jokin osa-alue puutteellisesti tutkittu. [Petersen et al., 2008] Tähän

tutkielmaan järjestelmällinen kartoitus ei sovellu, koska siinä tehtävä analyysi on liian pintapuolinen: vain tiivistelmät luetaan.



Kuva 8. Järjestelmällinen kirjallisuuskatsaus Salmista [2011], Finkiä [2005] ja Kitchenhamia ja muita [2007] mukaillen.

Järjestelmällinen katsaus toteutetaan yleensä tutkimusryhmässä ja Kitchenham ja muut [2007] ehdottavatkin yksittäiselle tutkijalle kevyempää versiota. Sen tärkeimmät vaiheet ovat tutkimusmetodin kehittäminen, tutkimuskysymyksen määrittäminen, haku, tutkimuksista haettavan datan määrittäminen, listojen ylläpitäminen rajatuista tutkimuksista, datan synteesi ja raportointi.

Kuvassa 8 on Finkin [2005] järjestelmällisen kirjallisuuskatsauksen mallia [Salminen, 2011] mukaillen tehty yhteenveto kevyemmästä kirjallisuuskatsauksesta, jollainen tässä tutkielmassa toteutetaan. Esimerkiksi katsauksen suorittamisen kohdalla olisi täydemmässä versiossa prosessin testaus ja katsauksen tekijöiden kouluttaminen.

3.3. Haun määrittely ja rajaus

Haku tehdään Googlen ja Nelliportaalin aiheeseen liittyvistä tietokannoista. Aiheen ulkopuoliset tietokannat *Communication & Mass Media Complete (Ebsco)* ja *LISA* ovat mukana haussa siksi, että ne kuuluvat Nelliportaalissa samaan pikahakuryhmään. Tietokannat ovat seuraavat:

1. Nelliportaalin aineistohaku tietokantaryhmästä ”Viestintä, tietojenkäsittely, informaatiotieteet” (http://www.nelliportaali.fi)

- *EBSCOhost Academic Search Premier*

  Tietokannassa on kokotekstinä noin 4 000 vertaisarvioitua lehteä lähes kaikilta tieteenaloilta.

- *Communication & Mass Media Complete (Ebsco)*

  Kokoelma sisältää viitteitä ja kokotekstejä viestinnän alan teoksiin.

- *Emerald*

  Yli 190 vertaisarvioidun lehden tietokanta Emerald on yleistieteellinen. Osa lehdistä on teknologian alalta.

- *LISA: Library and Information Science Abstracts (ProQuest)*

  LISA on informaatiotieteiden lehtiviitteitä sisältävä tietokanta.

- *ACM Digital Library*

  On tietojenkäsittelyn lehtien ja konferenssijulkaisujen kokotekstien tietokanta.

- *IEEE/IET Electronic Library* (IEL)

  Tietotekniikan, elektroniikan ja sähkötekniikan alojen tietokanta, joka sisältää verkkolehtiä, konferenssijulkaisuja ja standardeja.

2. *Google Scholar* -haku (http://scholar.google.com)

   Googlen tieteellisten teosten hakupalvelu. Kattaa tieteellisten artikkelien lisäksi muun muassa seminaariesitelmiä.

3. *Google Search* -haku (http://www.google.fi)

   Google-haku on Internetin käytetyin hakupalvelu. Googlen haku pisteyttää ja järjestää sivut PageRank-algoritmin mukaan. Mitä enemmän sivulla on viittauksia, sitä korkeammalle se nousee tuloksissa.

Itsenäisen kehittäjän ketterille menetelmille ei ole vakiintunutta englanninkielistä käsitettä. Käsitettä ”solo agile” esiintyy jonkin verran. Toisinaan sanaa ”lone” käytetään tarkoittamaan yksittäistä kehittäjää. Ohjelmistokehittäjä on ”developer”. Toisaalta käytetään myös sanaa ”programmer”.

Haun haastavuutta lisää kuitenkin se, että sanaa ”solo” käytetään parikoodauksen (engl. *pair programming*) vastakohtana: ”solo” ei useimmiten viittaa yksittäisen kehittäjän ”ryhmään” vaan siihen, että ryhmään kuuluvat jäsenet ohjelmoivat itsenäisesti. Sanan ”pair” yleisyyden vuoksi hausta ei voi hylätä tuloksia, joissa sana esiintyy (-”pair programming”). Relevanttejakin hakutuloksia jää silloin piiloon.

Yhdistämällä yllä olevat saadaan Googlen hakulauseeksi ”*lone OR solo programmer OR developer agile*”. Googlen haku on itsestään AND-haku, mutta vaihtoehtoiset sanat pitää ilmoittaa OR-operaattorilla [Google, 2014]. Tulokset ovat samat hakulauseelle ”(*lone OR solo) (programmer OR developer) agile*”, mutta sulkumerkkejä ei tarvita.

Toisena hakusanana käytetään ”*"personal software process" agile*”, jolla saadaan etsittyä täsmällisesti sellaiset ”Personal Software Process” -hakulausetta käsittävät sivut, joissa esiintyy myös sana ”agile”. Kolmantena korvataan ensimmäisestä hakulauseesta ”*agile*” sanalla ”*iterative*”, sillä hakutulokset antoivat aihetta kokeilla tätäkin hakusanaa.

Nelliportaalissa hakulogiikka on erilainen ja pitkän hakusanan käyttäminen tuo vähemmän tuloksia. Siellä käytetään hakusanaa ”*lone developer agile*” kaikista hakukentistä. Lisäksi tehdään haut, joissa ”*lone*” ja ”*developer*” korvataan sanoilla ”*solo*” ja ”*programmer*”.

Hakutuloksista raportoidaan tietokannoittain ja hakusanoittain tulosten määrä, haun päivämäärä ja sivu, mihin asti hakutuloksia luettiin. Haku tehdään yllä mainituista tietokannoista.

Hakutuloksia voi olla suuri määrä (satojatuhansia), joten kaikkia ei voi käydä läpi. Kun tulokset järjestetään relevanssin mukaan, on todennäköistä, että viimeisten tulosten joukossa ei enää löydy sopivia. Siksi tulosten lukeminen loppuun asti ei ole järkevää.

Läpikäynti lopetetaan, jos sopivaa hakutulosta ei löydy viideltä peräkkäiseltä sivulta (20 hakutulosta sivua kohden).

Hakutuloksista tarkistetaan tieteelliset teokset (tutkimusartikkelit, seminaari-esitelmät, kirjat ja opinnäytteet), joissa otsikon tai tiivistelmän perusteella esitellään *jokin yksittäiselle kehittäjälle tarkoitettu ketterä menetelmä*.

Rajauksen ulkopuolelle jäävät *verkkosivut*. Tällaisiin verkkosivuihin kuuluu useita keskustelusivuja ja blogikirjoituksia, joissa käsitellään yksittäisen kehittäjän mahdollisuuksista käyttää ketteriä menetelmiä. Vaikka verkkosivuja ei katsaukseen otetakaan, niiden selaamisesta oli hyötyä tiedon etsinnässä ja hakusanojen määrittelyssä.

Käytännön sanelema karsintakriteeri on *lähteen saatavuus*. Kolmas rajoite on *kieli*: englanti kelpuutetaan. Tosin jo hakusanat rajaavat itsestään kielen englantiin.

3.4. Laadun arviointi

Laaduntarkkailu kuuluu järjestelmälliseen kirjallisuuskatsaukseen. Laatu voi olla yksi karsintakriteereistä tai sen avulla voi arvioida yksittäisten tutkimusten painoarvoa. Laadun osatekijöinä voi pitää tutkimuksen reliabiliteettia (luotettavuutta), validiteettia (tarkkuutta) ja yleistettävyyttä. Tutkimusten luotettavuusarvio toteutetaan lomakkeella, johon kerätään tietoa mukaan otetuista tutkimuksista. [Kitchenham et al., 2007] Lomake on liitteessä 3. Se koostuu kysymyksistä, joilla arvioidaan tutkimuksen taustan käsittelyn, menetelmän ja tulosten laatu. Kysymykset on koostettu Kitchenhamin ja muiden [2007] ohjeistuksen esimerkeistä.

Opinnäytetöitä ja verkkosivuja voidaan pitää heikkoina lähteinä. Vertaisarvioiduissa lehdissä julkaistujen tutkimusartikkeleiden laatua voisi luokitella lehden IF-arvon (Journal Impact Factor), AI-indeksin (Aggregate Immediacy Index) tai muiden tunnuslukujen avulla [Eskelinen ja Karsikas, 2012]. Hakutulokset antoivat sekä opinnäytetöitä että tutkimusartikkeleita. Löydettyjen tutkimusten vähyyden vuoksi opinnäytteetkin kuitenkin hyväksytään mukaan eikä laatukarsintaa tehdä. Siksi laadun arviointi toteutetaan muun analyysin yhteydessä.

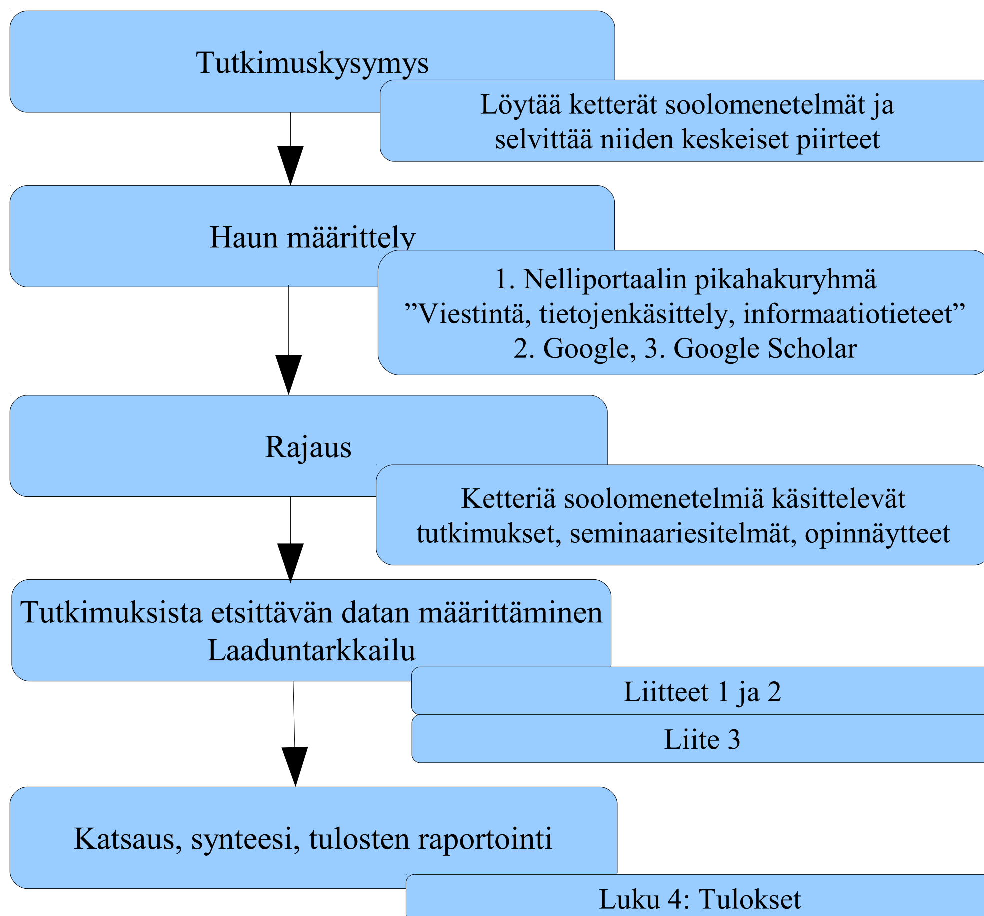
3.5. Analyysi ja synteesi

Viimeisenä vaiheena suoritetaan tiedon erottelu eli *analyysi* ja tietojen yhdistäminen eli *synteesi*. Analyysi tehdään tiedonkeruulomakkeella. Perustietojen kuten viitetietojen lisäksi lomakkeelle kerätään tutkimuskysymyksestä nousevia kysymyksiä. Tiedonkeruu toteutetaan tutkimusryhmissä yhtä tutkimusta kohden ainakin kahden henkilön voimin erikseen. Näin saavutetaan luotettavampi tulos ja eroavaisuuksista voidaan keskustella

ja toistaa keruu uudelle lomakkeelle virheiden korjaamista varten. Yksittäisen tutkijan tapauksessa voidaan käyttää luotettavuuden varmistamiseksi esimerkiksi uudelleen keruuta: tutkija analysoi tiedot uudestaan satunnaisesti valitusta otoksesta tutkimuksia. [Kitchenham et al., 2007]

Tässä tutkielmassa tiedonkeruu toteutetaan vaiheittain. Ensimmäisessä vaiheessa kerätään perustietojen lisäksi tutkielman tyyppi (opinnäyte, artikkeli) ja se, mihin ketteriin malleihin menetelmät perustuvat. Toisessa vaiheessa menetelmiä selvitetään tarkemmin: millainen on tutkimusmenetelmä, miten ketteriä käytäntöjä toteutetaan ja millaisia tuloksia tutkimuksesta on saatu. Laatuarvio toteutetaan tässä yhteydessä. Näin kukin tutkimus tulee arvioiduksi kolmeen kertaan. Tiedonkeruulomakkeet ovat liitteissä 1 ja 2.

Tiedon synteesin tarkoituksena on kerätä analysoitu tieto yhteen esimerkiksi erilaisiin taulukoihin. Synteesi voi olla kuvaileva, tai siihen voi yhdistää kvantitatiivisen yhteenvedon [Kitchenham et al., 2007]. Tiedon esittämiseen käytetään kuvailevaa synteesiä ja taulukointia. Kuva 9 tiivistää katsauksen keskeiset vaiheet.



Kuva 9. Tiivistelmä katsauksen suorittamisesta.

4. Tulokset

4.1. Haku

Taulukossa 2 on lueteltu suoritetut haut. Olen merkinnyt hauille lyhenteen (Lyh.), jolla voin viitata tiettyyn hakuun. Hauista on kerrottu tietokanta, käytetty hakusana, hakutulosten määrä, monennellako sivulla haku on keskeytetty ja milloin haku on tehty. Nelliportaalin tietokannasta tekemäni neljä hakua (N1, N2, molemmat ensin sanan ”developer” kanssa, sitten sanan ”programmer” kanssa) tuottivat siedettävän määrän hakutuloksia ja saatoin käydä läpi ne kaikki. Googlen hakutulokset G1, G2, G3, GS1, GS2 keskeytin, kun viiteen sivuun ei tuloksia enää löytynyt. Toteutin Google-haut G1, GS1 ja G3 ensin 11.1.2014. Toistin haut 4.3.2014 OR-operaattorin kanssa.

Taulukko 2. Haut.

Tietokanta	Lyh.	Käytetty hakusana	Hakutuloksia	Haku keskeytetty	Haun päivämäärä
Google	G1	solo OR lone programmer OR developer agile	845 000	sivulla 9	4.3.2014
Google	G2	"personal software process" agile	1 960 000	sivulla 8	11.1.2014
Google Scholar	GS1	solo OR lone programmer OR developer agile	2680	sivulla 7	4.3.2014
Google Scholar	GS2	"personal software process" agile	789	sivulla 5	11.1.2014
Nelliportaali	N1	lone developer/program mer agile	56 ja 48	loppuun asti	11.1.2014
Nelliportaali	N2	solo developer/program mer agile	64 ja 62	loppuun asti	11.1.2014
Google	G3	solo OR lone programmer OR developer iterative	460 000	sivulla 9	4.3.2014

Hakutulokset ovat taulukossa 3. Haku GS2 ei tuottanut yhtään sopivaa hakutulosta. Haut ”Solo Scrum”, ”solo lean”, ”solo kanban”, ja niin edelleen, eivät tuota enempää tieteellisiä teoksia. Myöskään haku sanan ”single” kanssa ei tuonut lisää tuloksia. Akpata ja Riha [2004] esittävät, että *XP41* (*XP for one,* XP yhdelle) olisi käytössä oleva nimitys yhden kehittäjän XP:lle. Hakusanalla ei kuitenkaan löydä muita tutkimuksia.

Taulukko 3. Löydetyt tutkimukset sekä löytymisjärjestyksessä haku, jolla ne löytyivät.

#	Tutkimuksen otsikko	Tekijä, vuosi	Haku
1	*Can Extreme Programming be used by a Lone Programmer?*	[Akpata and Riha, 2004]	G1 G3 GS1
2	*Agile Solo - Defining and Evaluating an Agile Software Development Process for a Single Software Developer*	[Nyström, 2011]	G1 G2 G3 GS1
3	*Cowboy: An Agile Programming Methodology for a Solo Programmer*	[Hollar, 2006]	G1 G3 GS1
4	*From Scrum to Solo: How Small is Too Small a Team to Still Call it Software Engineering?*	[Dent et al., 2008]	G1 G3 GS1
5	*Integrating PSP with agile process: a systematic review*	[Shen et al., 2013]	G2
6	*Personal Extreme Programming – An Agile Process for Autonomous Developers*	[Dzhurov et al., 2009]	G2
7	*SCRUM-PSP: Embracing Process Agility and Discipline*	[Rong et al., 2010]	G2
8	*Extreme Programming for a Single Person Team*	[Agarwal and Umphress, 2008]	N1 GS1
9	*Software Change in the Solo Iterative Process*	[Dorman and Rajlich, 2012]	N2 GS1 G3
10	*eXtreme Solo - A Case Study in Single Developer eXtreme Programming*	[Chronin, 2001]	G3

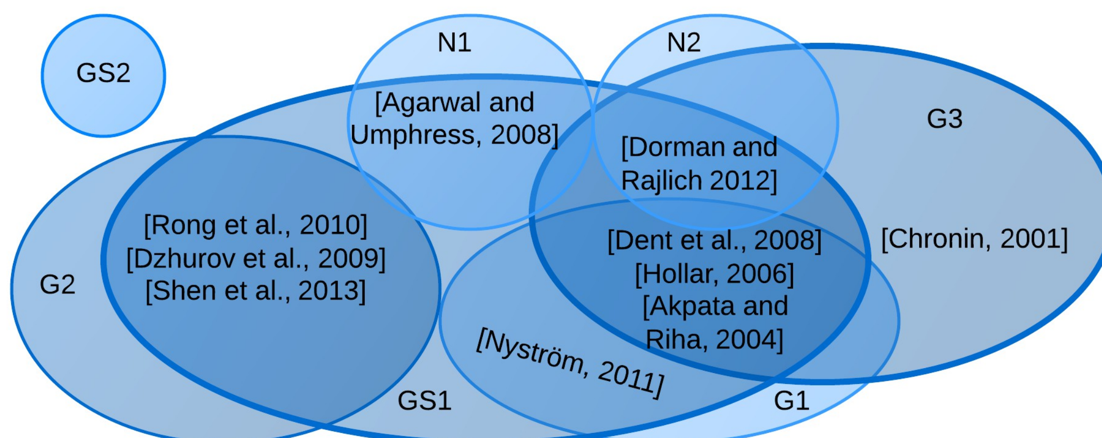
Haulla G3 löytyi lupaava otsikko *Extreme Programming for Solo Projects*, mutta tiedosto ei ollut enää saatavilla. Verkko-osoitteen perusteella se olisi kuitenkin ollut diaesitys ja olisi karsiutunut muutenkin pois. Monet karsiutuneet tutkimukset vertailivat pari- ja soolo-ohjelmoinnin eroja.

Keskustelusivujen lisäksi haun tuloksena löytyi kuitenkin sivustoja, joissa käsitellään itsenäisen kehittäjän ketteriä menetelmiä. Vaikka ne karsiutuvatkin katsauksesta, on niissä mielenkiintoisia ohjeita ja suosituksia. Esittelen ne liitteessä 4.

Monet hakutulokset löytyivät myös toisten hakujen avulla, kuten taulukosta 4 huomataan. Kuvasta 10 nähdään, että samaan hakutulokseen olisi päästy tekemällä pelkästään haut GS1 ja G3. Eri hakupalveluiden ja hakulausekkeiden käyttäminen oli siis perusteltua.

Taulukko 4. Tutkimukset hauittain.

Haku	Tutkimukset
G1	[Akpata and Riha, 2004], [Nyström, 2011], [Hollar, 2006], [Dent et al., 2008]
G2	[Nyström, 2011], [Shen et al., 2013], [Dzhurov et al., 2009], [Rong et al., 2010]
G3	[Akpata and Riha, 2004], [Nyström, 2011], [Hollar, 2006], [Dent et al., 2008], [Dorman and Rajlich, 2012], [Chronin, 2001]
GS1	[Akpata and Riha, 2004], [Nyström, 2011], [Hollar, 2006], [Dent et al., 2008], [Agarwal and Umphress, 2008], [Dorman and Rajlich, 2012]
GS2	–
N1	[Agarwal and Umphress, 2008]
N2	[Dorman and Rajlich, 2012]



Kuva 10. Tutkimukset hauittain Venn-diagrammina.

4.2. Katsaus

Tiedonkeruun ensimmäisen vaiheen toteutin 18.1.2014. Tulokset tästä ovat taulukossa 5 (sivu 26). Tiedonkeruun tulosten perusteella saatoin jakaa soolomenetelmät kolmeen ryhmään sen mukaan, mikä niiden kehittelyn pohjana on ollut:

1. Menetelmät, jotka perustuvat PSP:n ja jonkin ketterän menetelmän yhdistämiseen
2. Ketterät menetelmät, joista on karsittu ryhmien toimintaan liittyvät käytännöt
3. Solo Iterative Process, joka on erillinen kahdesta ensimmäisestä

Tutkimuksissa on mukana kirjallisuuskatsaus [Shen et al., 2013], joka käsittelee ryhmän 1 menetelmiä. Taulukon 5 tutkimukset 2–4 on käsitelty tuossa katsauksessa. Katsauksen tutkimuskysymykset ovat lähes samat kuin omani, joten referoin tutkimuksen tulokset sen sijaan, että täyttäisin niistä toisen tiedonkeruulomakkeen. SCRUM-PSP-menetelmä ei lopulta ollutkaan itsenäiselle kehittäjälle tarkoitettu, vaan yksittäiselle kehittäjälle osana ryhmää. Toiseen tiedonkeruuseen jäävät näin ollen tutkimukset 5–10. Käytän jatkossa lyhyempää, alleviivattua osaa nimistä.

Taulukoissa 6 ja 7 (sivut 27 ja 28) on lueteltu tiedonkeruu 2 -lomakkeella (liite 2; toteutettu 25.1.–5.2.2014) kerätyt menetelmien keskeiset käytännöt ja tulokset. Menetelmien koostamiseen tekijät eivät ole käyttäneet minkäänlaisia malleja tai mittareita, eivät myöskään niiden arvioimiseen. Koostamisen perusteluina on käytetty omaa kokemusta tai tuntumaa siitä, mikä voisi olla hyvä yksittäiselle kehittäjälle.

Kirjassaan ketteristä käytännöistä Schuh [2005] mainitsee lyhyesti, että itsenäinen kehittäjä voi toteuttaa seuraavia ketteriä käytäntöjä: yksinkertainen suunnittelu, refaktorointi, testivetoinen kehitys ja automatisoidut kääntäminen, asennus ja testaus. Samoja suosituksia esiintyy myös katsauksen tutkimuksissa.

Jokaisessa keskeisenä käytäntönä oli syklinen kehitys ja lyhyet, kahdesta neljään viikkoon pituiset iteraatiot. Neljä kuudesta käytti ainakin yhtenä pohjana XP-menetelmää. Dent ja muut [2008] eivät esitelleet täsmällistä menetelmää, mutta heilläkin oli mukana XP-käytännöistä käyttäjätarinat ja testivetoinen kehitys. Scrum oli pohjana kahdessa, AUP ja Getting Real yhdessä. Täysin erillinen menetelmä oli Solo Iterative Process.

Scrumia käyttävät menetelmät sisälsivät projektin kehitysjonon, samoin Solo Iterative Process. XP-pohjaisissa oli kaikissa jonkinlainen suunnittelupeli ja tehtäväkorttien tekeminen. Muissa menetelmissä parikoodaus jätettiin pois, mutta kahdessa sitä kompensoitiin koodin itsearviolla. Toinen näistä menetelmistä käytti

siihen Pomodoro-menetelmää, jonka selitän kohdassa 5.2 Agile Solon tarkemman esittelyn yhteydessä.

Akpata ja Riha [2004] jakavat tutkimuksessaan XP:n käytännöt kolmeen ryhmään. Muutettuna soveltuvat suunnittelupeli, työajan rajoittaminen, yksinkertainen suunnittelu, jatkuva integraatio ja läsnä oleva asiakas. Muut soveltuvat sellaisenaan, mutta parikoodaus ja yhteinen omistus eivät lainkaan. Myös muut XP-pohjaiset menetelmät ymmärrettävästi hylkäävät parikoodauksen ja yhteisen omistajuuden.

Jokaisen tutkimuksen tulos oli, että menetelmä oli onnistunut. Kanbania soveltanut piti mallinnusta ja visualisointia tärkeänä kommunikoinnin välineenä asiakkaan kanssa. Useimmissa menetelmissä suositeltiin yksinkertaista suunnittelua ja versionhallinnan käyttämistä. Yksinkertaistamista tehtiin myös käytäntöjen kanssa: tietokoneistetusta kehitysjonosta tuli paperinen työlista.

Kahdessa tapaustutkimuksessa yksikkötestit jätettiin tekemättä tai vähemmälle huomiolle, koska tekijä koki ne hankaliksi. Etukäteen valmistautumista korosti kaksi menetelmää: Agile Solossa valmiiksi mietitty agenda tehosti arviointikokouksia ja SIP:ssa koodin vaikutusanalyysi ennustaa refaktorointia.

Taulukossa 8 (sivu 29) on tiedonkeruun tuloksia asiakasyhteistyöstä. Tapaustutkimusten toteuttamisesta kerättiin tiedot siitä, keitä tutkimuksessa oli mukana ja vastasiko tutkimus tutkimuskysymyksiin. Lisäksi taulukossa on keskeiset osat laatuarviosta (liite 3).

Asiakas on mukana neljässä tapaustutkimuksessa. Viidennessä asiakasta ei ole huomioitu. Asiakas osallistuu kehitykseen suunnittelun aikana: kolmessa suunnittelupelin avulla, yhdessä Getting Real -tyyppisesti rajoittamalla asiakkaan vaihtoehtoja. Scrumia soveltavissa menetelmissä asiakas osallistuu arviointikokoukseen.

Tulosten luotettavuutta ja yleistettävyyttä oli arvoitu kolmessa viidestä (Dentillä ja muilla [2008] ei tapaustutkimusta) kiitettävästi, yhdessä jonkin verran ja yhdessä ei lainkaan. Tuloksia ei pidetty yleistettävinä. Jokainen tapaustutkimus lukuun ottamatta Akpatan ja Rihan [2004] kuvausta oli tutkijan itsensä toteuttama.

Taulukko 5. Tiedonkeruu 1 -lomakkeen tulokset.

#	Tutkimuksen otsikko	Tekijä, vuosi	Pohja-menetelmä	Tyyppi	Sivut
1	*Integrating PSP with agile process: a systematic review*	[Shen et al., 2013]	XP, PSP	Tutkimus-artikkeli	7
2	*Personal Extreme Programming – An Agile Process for Autonomous Developers*	[Dzhurov et al., 2009]	XP, PSP	Tutkimus-artikkeli	8
3	*Extreme Programming for a Single Person Team*	[Agarwal and Umphress, 2008]	XP, PSP	Tutkimus-artikkeli	6
4	*SCRUM-PSP: Embracing Process Agility and Discipline*	[Rong et al., 2010]	Scrum, PSP	Tutkimus-artikkeli	10
5	*eXtreme Solo - A Case Study in Single Developer eXtreme Programming*	[Chronin, 2001]	XP	Opinnäyte	68
6	*Can Extreme Programming be used by a Lone Programmer?*	[Akpata and Riha, 2004]	XP	Tutkimus-artikkeli	9
7	*Agile Solo - Defining and Evaluating an Agile Software Development Process for a Single Software Developer*	[Nyström, 2011]	Scrum, XP, Kanban	Opinnäyte	45
8	*Cowboy: An Agile Programming Methodology for a Solo Programmer*	[Hollar, 2006]	Scrum, XP, AUP, Getting Real	Opinnäyte	80
9	*From Scrum to Solo: How Small is Too Small a Team to Still Call it Software Engineering?*	[Dent et al., 2008]	–	Seminaarin kokemus-raportti	8
10	*Software Change in the Solo Iterative Process*	[Dorman and Rajlich, 2012]	(SC, SIP)	Tutkimus-artikkeli	10

Taulukko 6. Tiedonkeruu 2, keskeiset käytännöt.

#	Tutkimus Pohjamenetelmät	Keskeiset käytännöt
5	[Chronin, 2001]: *eXtreme Solo* XP	XP:n 12 käytäntöä poislukien parikoodaus ja yhteisomistus.
6	[Akpata and Riha, 2004]: *Extreme Programming by a Lone Programmer* XP	XP:n käytännöt jaettuna kolmeen ryhmään: käyvät sellaisenaan, käyvät muutettuna, eivät sovellu lainkaan.
7	[Nyström, 2011]: *Agile Solo* Scrum, XP, Kanban	*Scrumista*: 4 viikon sprintit, 1 viikolla prioriteettien asettaminen ja suunnittelu, arviointi sprintin jälkeen *Testivetoinen kehitys*. *Mallinnus*. *Itsearvio koodista* (Pomodoro); *Kanban-visualisointi*; *projektinhallintaan* XP:n käyttäjätarinat tai Scrumin kehitysjono, miten tekijä itse haluaa.
8	[Hollar, 2006]: *Cowboy* Scrum, XP, AUP, Getting Real	*Yleiset*: kehitysjono (todo-lista), yksinkertainen suunnittelu; AUP-tuotokset A*siakas*: ks. taulukko 7 P*rojektinhallinta*: maks. 2 viikon iteraatiot, kommentointi; K*ehitys*: yksikkötestaus, versionhallinta.
9	[Dent et al., 2008]: *From Scrum to Solo* –	Versionhallinta, käyttäjätarinat, testivetoinen kehitys, päiväkirjavetoinen kehitys.
10	[Dorman and Rajlich, 2012]: *Software Change in the Solo Iterative Process* (SC, SIP)	Ohjelmamuutos (engl. *Software Change*), kehitysjono, versionhallinta ja itsearvio. Käsitellään tarkemmin kohdassa 5.3.

Taulukko 7. Tiedonkeruu 2, keskeiset tulokset.

#	Tutkimus Pohjamenetelmät	Keskeiset tulokset
5	[Chronin, 2001]: *eXtreme Solo* XP	Suunnittelupeli ja testivetoinen kehitys edistävät itsenäistä kehitystä.
6	[Akpata and Riha, 2004]: *Extreme Programming by a Lone Programmer* XP	Testien suunnittelu oli ajoittain hankalaa, ylityö väheni; esittää kysymyksen: ”onko yksin kehittäminen varsinaisesti XP:tä?”
7	[Nyström, 2011]: *Agile Solo* Scrum, XP, Kanban	Pomodoro-menetelmä ei liian lyhyenä kelpaa luovaan työhön; *mallinnus* varmisti, että asiakaskin tiesi mitä oltiin tekemässä; *Kanban*-visualisointi varmisti, että työtä ei ollut liikaa; *testivetoinen kehitys* varmisti, että tiesi mitä teki.
8	[Hollar, 2006]: *Cowboy* Scrum, XP, AUP, Getting Real	Onnistunut projekti, valmiiksi mietitty agenda tapaamisiin piti aikataulun kunnossa, yksikkötestaus jäi tekemättä koska tekijä ei osannut tehdä niitä, projekti oli osa-aikainen joten tulokset eivät yleistettävissä.
9	[Dent et al., 2008]: *From Scrum to Solo* –	Ei tapaustutkimusta, ei tuloksia; menetelmän hyötyjä perustellaan kirjoittajan omalla kokemuksella.
10	[Dorman and Rajlich, 2012]: *Software Change in the Solo Iterative Process* (SC, SIP)	Projekti oli onnistunut; vaikutusanalyysi oli tärkeä ja sillä pystyi ennustamaan esifaktorointia – mutta ei jälkifaktorointia; myös muutoksen kompleksisuus pitäisi arvioida; käytetyt ohjelmat edistivät projektin onnistumista.

Taulukko 8. Tiedonkeruu 2, asiakasyhteistyö ja tapaustutkimuksen laatu.

#	Tutkimus	Asiakasyhteistyön toteuttaminen	Tapaustutkimuksen laatu
5	[Chronin, 2001]: *eXtreme Solo*	Suunnittelupeli	Omasta projektista, ei määrittele tutkimuskysymyksiä; luotettavuutta ja yleistettävyyttä arvioitu jonkin verran.
6	[Akpata and Riha, 2004]: *Extreme Programming by a Lone Programmer*	Saatavilla oleva asiakas, suunnittelupeli	Kuvailee keskeiset huomiot eräästä XP for One -projektista, luotettavuutta ja yleistettävyyttä ei arvioitu.
7	[Nyström, 2011]: *Agile Solo*	Asiakas mukana suunnittelussa; kokoukset viikoittain ja isommat arviointikokoukset kuukausittain	Omasta projektista, tutkimuskysymykset selvät, luotettavuutta ja yleistettävyyttä arvioitu kiitettävästi.
8	[Hollar, 2006]: *Cowboy*	Getting Real -tyyppinen asiakas: rajoita aikaa ja resursseja, kieltäydy liiasta; yhteinen sanasto; arviokokouksissa asiakas testaa ohjelman tehtäväkortin avulla; kyselylomakkeet tapaamisissa	Omasta projektista, tulokset esitelty selkeästi, tulkintoja niukasti, yleistettävyyttä arvioitu kiitettävästi.
9	[Dent et al., 2008]: *From Scrum to Solo*	Ei huomioitu	Ei tapaustutkimusta, ei selkeää menetelmää, ei tuloksia.
10	[Dorman and Rajlich, 2012]: *Software Change in the Solo Iterative Process*	Ei huomioitu	Omasta projektista, tutkimuskysymykset selvät, luotettavuutta ja yleistettävyyttä arvioitu kiitettävästi.

4.3. PSP-menetelmät

Shenin ja muiden [2013] järjestelmällinen kirjallisuuskatsaus PSP:tä hyödyntävistä ketteristä menetelmistä sisältää myös minun löytämäni Dzhurovin ja muiden [2009], Agarwalin ja Umphressin [2008] ja Rongin ja muiden [2010] tutkimukset. Osa katsauksessa tutkituista menetelmistä ei kuitenkaan käsittele itsenäisen kehittäjän menetelmiä, sillä PSP:tä voi käyttää myös ryhmässä.

Tällainen on esimerkiksi Scrumin ja PSP:n yhdistänyt Rongin ja muiden [2010] menetelmä, jossa jokainen Scrum-ryhmän jäsen toteuttaa PSP-prosessia – kehittäjä ei siis ole itse koko Scrum-ryhmä eri rooleissa. Sinänsä on mielenkiintoista yhdistää etukäteissuunnittelua korostava PSP ketteriin menetelmiin, joiden yhtenä tavoitteena on sietää odottamattomia muutoksia.

Shenin ja muiden [2013] katsauksessa oli tarkoituksena etsiä PSP:tä ja ketteriä menetelmiä yhdistävät tutkimukset ja niiden päätulokset. Lisäksi arvioitiin tulosten tieteellistä todistusvoimaa ja niistä saatavia suosituksia tutkimukseen ja käytäntöön. Shen ryhmineen löysi neljä menetelmää, joita on yhdistetty PSP:hen: Scrum, RUP, DSDM ja XP. Esittelen näistä tarkemmin XP:tä käyttävän menetelmän kohdassa 5.1. XP-tutkimuksia oli tehty kahdeksan, Scrumia kolme, RUP:ia ja DSDM:ää kumpaakin vain yksi.

PSP:n yhdistäminen ketteriin menetelmiin tuo katsaukseen valikoituneiden tutkimusten mukaan seuraavia etuja:

1. PSP tarjoaa mittareita ja menetelmiä tiedon keräämiseen.
2. PSP:n avulla voi tehdä järkeviä arvioita.
3. PSP auttaa tehostamaan suunnitelmia.
4. PSP tukee laatutavoitteisiin pääsyä.
5. PSP tuo lisädokumentaatiota ketteriin menetelmiin.
6. PSP auttaa yksittäistä kehittäjää parantamaan henkilökohtaista tasoaan.

Yhteenvetona katsaus esittää, että PSP:tä pitää soveltaa projektin omien tavoitteiden mukaisesti. Menetelmiä oli testattu vain vähän todellisissa projekteissa.

5. Kolme itsenäisen kehittäjän menetelmää

Esiteltäväksi valitsen kolme tutkimusta seuraavien kriteerien mukaan:

1. Projektinhallinta ja asiakasyhteistyö tulisi huomioida.
2. Esiteltävät menetelmät pitäisi olla eri kategorioista (PSP, ketterä ilman ryhmäkäytäntöjä ja SIP).
3. Laadun tulisi olla riittävä (eli tapaustutkimusta, tulosten luotettavuutta ja menetelmän yleistettävyyttä arvioitu).

Koska projektinhallinta on SIP-tutkimuksessa mukana ja tutkimus on riittävän laadukas, kelpuutan sen mukaan.

Agile Solo ja Cowboy nousevat edelle muita projektinhallinnan ja asiakasyhteistyön kuvailun ansiosta. Agile Solo -tutkielmassa menetelmän arviointi oli kattavampi, joten se olkoon toinen esiteltävä malli.

PSP-malleista valitsen esiteltäväksi Agarwalin ja Umphressin [2008] PXP-mallin, sillä hän esittää menetelmässään yksityiskohtaisen PSP-toimintaohjeen.

5.1. PXP

Agarwal ja Umphress [2008] esittelevät kehittelemänsä PXP-menetelmän (Personal Extreme Programming). PXP yhdistää PSP:n toimintaohjelistauksen XP:n käytäntöihin.

Myös PXP joutuu poistamaan tai muuttamaan ryhmien toimintaan liittyviä XP-käytäntöjä. Ensinnäkin PXP:stä jätetään pois pariohjelmointi, koska yksin sitä on mahdoton tehdä. Koodin yhteisomistajuus on PXP:ssä linjassa XP:n kanssa, koska jokainen (yksi) ohjelmoija pääsee muokkaamaan kaikkien (sen yhden ja saman) koodia. Vaikka on vain yksi tekijä, koodistandardejakin tarvitaan, jotta koodin yhtenäisyys säilyy. Samoin itsenäinen kehittäjä tekee vertauskuvat.

Asiakas ei aina kykene olemaan läsnä yksittäisen kehittäjän projekteissa. PXP ehdottaa muita viestintävälineitä (puhelut, viestit) yhteydenpitoon asiakkaan kanssa. Suunnittelupelistä PXP jopa ehdottaa, että jos asiakasta ei ole, tekijä itse vaihtaa rooleja suunnittelupelin aikana.

Muut käytännöt sovelletaan sellaisenaan: Julkaisuja tuotetaan sopivin, lyhyin väliajoin. Yksinkertainen suunnittelu, yksikkötestit, jatkuva integraatio ja refaktorointi tehdään. Ylitunteja vältetään ja 40-tuntinen viikko on tavoite. Lisäksi kehittäjä voi soveltaa käytäntöjä hyväksi näkemällään tavalla niin kuin muutenkin XP:ssä.

Lisänä XP:lle PXP tuo PSP-mallisen toimintaohjeen. Toimintaohje on taulukoissa 9 ja 10. Ohjeistus annetaan paikoin FOR-silmukoina. Suunnittelu sisältää vaatimusten ja ominaisuuksien keräämisen. Tässä PXP kehottaa käyttämään toimialueen sanastoa, jotta kommunikointi asiakkaan kanssa tehostuu. Ominaisuudet järjestetään joukoiksi ja joukkojen sisällä järjestetään tärkeyden mukaan listaksi.

Kehitysiteraatiossa poimitaan listasta yksi ominaisuus kerrallaan tärkeimmästä aloittaen, jaetaan se tehtäviksi ja toteutetaan tehtävät. Ohjelmaa kehitetään kolmessa haarassa: kehitys, refaktorointi, tuotanto. Kehitys tehdään kehityshaarassa ja iteraation lopuksi koodi viedään refaktorointihaaraan. Refaktoroinnin jälkeen koodi julkaistaan tuotantohaarassa.

PXP:n toimintaohjeiden kohta J1 ei vastaa PSP-mallia. Jälkituotantovaiheessa olisi tarkoitus analysoida mitattu data, sillä datan mittaamisen tarkoituksena on käyttää sitä jatkossa hyödyksi. PXP ei huomioi tätä lainkaan. Menetelmästä puuttuvat PSP2-tason suunnittelukatselmointi ja suunnittelupohjat. Koodikatselmointi on mukana. PSP1-tasolta puuttuu testausraportointi, vaikka aikataulu- ja tehtäväsuunnittelu ovatkin mukana. Virheiden kirjaamista ei mainita ja kehittymissuunnitelmaa ei pyydetä tekemään. Itsenäinen kehittäjä, joka haluaa kehittyä henkilökohtaisen ohjelmisto-prosessin mukaisesti, ei voi ilman näitä lisäyksiä ottaa PXP:tä käyttöönsä.

Taulukko 9. PXP-toimintaohjeet [Agarwal ja Umphress, 2008].

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 mukaan TärkeysJonoksi.
	S8.	Tee iteraatioille aikataulu.
Kehitys	Katso taulukko 10.	
Jälkituotanto	J1.	Toteuta lopullinen hyväksymistestaus tuotannon koodille.
Lopetusehto	L1.	Valmis, testattu ohjelma.

Taulukko 10. PXP-toimintaohjeiden kehitysvaihe [Agarwal ja Umphress, 2008].

Kehitys	D1. 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

5.2. Agile Solo

Nyströmin [2011] *Agile Solo* on Chalmersin yliopistolle tehty M.Sc.-tasoinen opinnäytetyö. Nyströmin tavoitteena on ollut tehdä ketterään julistukseen ja muihin ketteriin malleihin pohjautuva yksittäisen kehittäjän menetelmä. Ensiksi hän esittelee menetelmänsä, sitten tekee projektin sitä käyttäen ja lopuksi arvioi menetelmän käytännöt. Kuvaan 11 olen koostanut kaavion Agile Solo -mallista, jota selostan alla.

Scrumista Nyström ottaa iteraatiokierrokseksi *neljän viikon sprintit*: Joka viikko pidetään asiakkaan kanssa esittely, jotta varmistetaan jatkuva palaute. Esittelyssä päivitetään tehtävien tärkeysjärjestystä. Neljännen viikon lopuksi tehdään julkaisu ja asiakas testaa tuotteen.

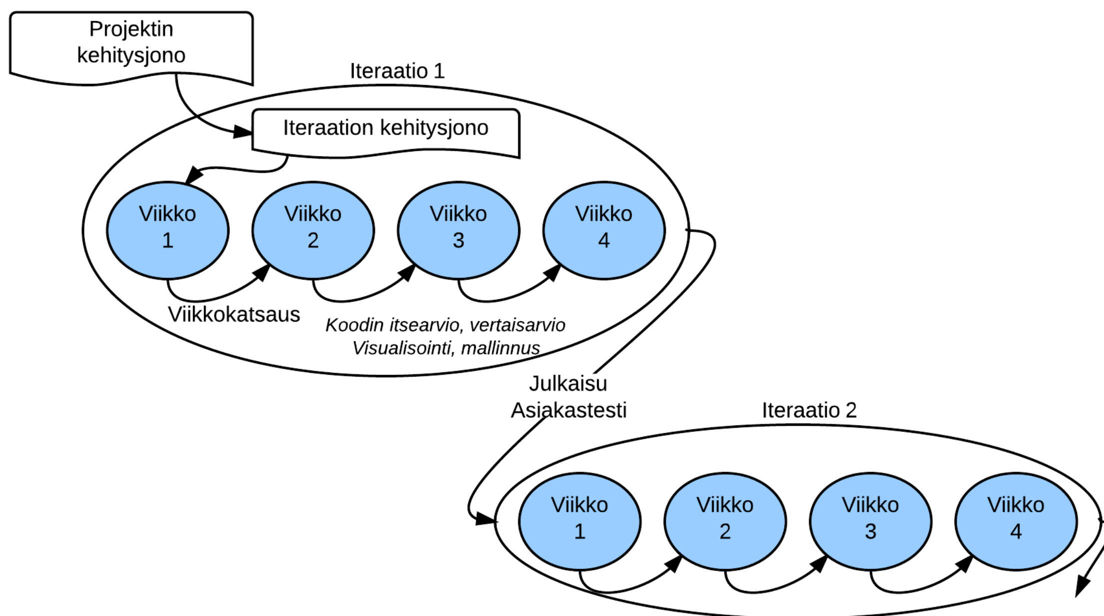
Iteraation suunnittelussa priorisoinnin lisäksi arvioidaan aika ja lopuksi verrataan arvioitua todellisuudessa kuluneeseen aikaan: näin jatkossa saadaan tarkempia arvioita. *Projektinhallintaan* kehittäjä voi itse valita, ottaako XP:n käyttäjätarinat tai Scrum-tyyppisen kehitysjonon. Hallintamenetelmä kannattaa kuitenkin tehdä yksinkertainen ja sen käytössä olla joustava. Nyström käytti itse valitsi kehitysjonon, joka lopulta yksinkertaistui paperiseksi työlistaksi.

Agile Solossa lisäksi kehotetaan *visualisoimaan työtä* itse valitsemallaan tavalla, esimerkiksi Kanban-taululla. Näin pysyy itse selvillä työn vaiheista ja voi kätevästi näyttää tilanteen asiakkaallekin. *Mallinnusta* tulee tehdä sopivasti, jotta viestintä asiakkaan kanssa helpottuisi. Mallinnuksen pitää olla yksinkertaisia, koska ohjelmalla on ketterässä kehityksessä tapa muuttua voimakkaasti. Mallinnukseen ei kannata käyttää aikaa liiaksi. Nyström koki mallinnuksen aikaa vieväksi ja piti parempana kielellistä keskustelua, koska mallien ymmärtämiseen saattaa tarvita taustatietoja ohjelmistokehityksestä.

Itse kehitysvaiheessa Agile Solo korostaa XP:n mukaista testivetoista kehitystä. Pariohjelmoinnin sijaan käytetään vertaisarviointia ja itsearviota. Nyströmillä on ollut mielessä opiskelijaprojekti tai muu tilanne, jossa ohjelmoijalla on ohjaaja tai kollega. Nyström ehdottaa, että koodi näytettäisiin kaksi kertaa viikossa toiselle henkilölle, joka arvioi koodin. [Nyström, 2011] Aina tällaista mahdollisuutta ei itsenäisellä kehittäjällä kuitenkaan ole.

Koodin itsearviointiin Agile Solossa aluksi ehdotetaan Pomodoro-menetelmää. Pomodoro on Francesco Cirillon [2006] kehittämä ajanhallintamenetelmä, jonka tarkoitus on lisätä työtehoa. Menetelmässä tehdään 25 minuuttia yhtäjaksoista työtä, jonka jälkeen pidetään 5 minuutin tauko. Neljän jakson jälkeen pidetään pidempi, 15–30 minuutin tauko. Agile Solossa tauon jälkeen kehittäjä arvioi tekemänsä koodin itse.

Nyström joutui kuitenkin hylkäämään Pomodoron, sillä hän koki, että 25 minuuttia oli liian vähän luovaan työhön: ”päästyäni syvään keskittymisen tilaan ja ollessani tehokkaimmillani, minun tuli keskeyttää työni”. Hylkäämällä tiukat aikarajoitukset hän tunsi työnsä hyvin palkitsevaksi aina, kun sai jotain valmiiksi. Vähemmän kiinnostavaan tekemiseen, kuten dokumentointiin tai uuden tutkimiseen Pomodoron lyhyet työrupeamat olivat hänestä hyviä.



Kuva 11. Agile Solo -malli.

5.3. Solo Iterative Process

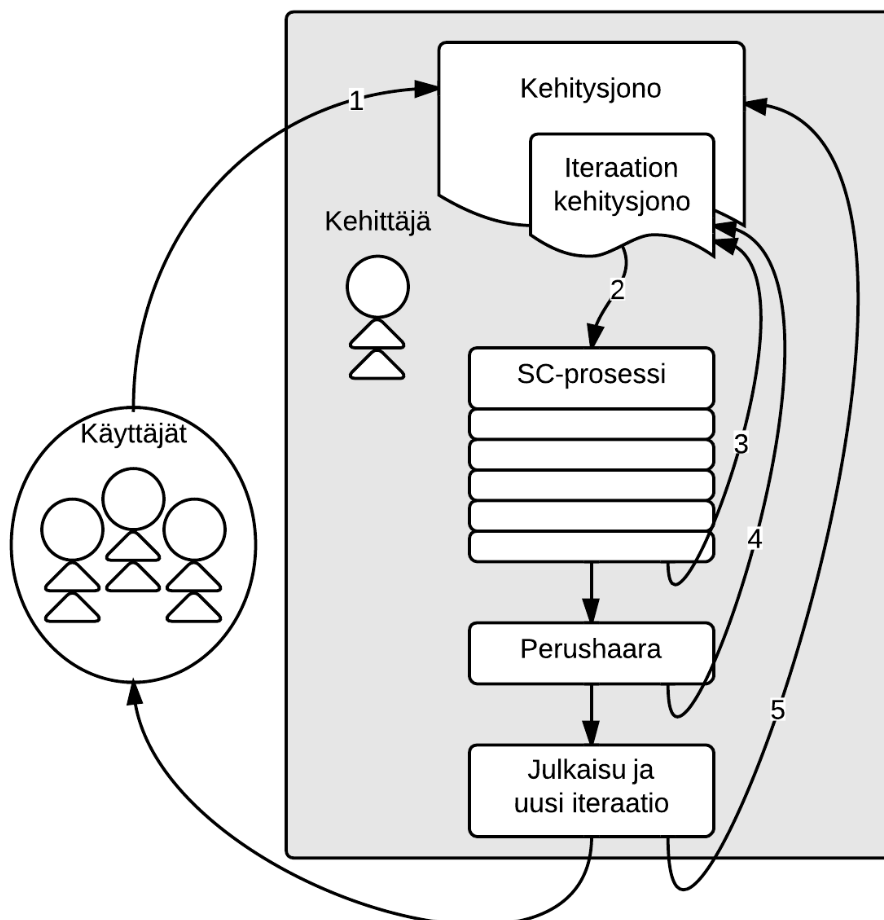
Rajlichin [2011] kehittämä malli Solo Iterative Process, SIP, perustuu hänen kehittämäänsä ohjelmistokehityksen malliin *ohjelmamuutoksesta* (engl. *Software Change, SC*). SC koostuu eri vaiheista, joita läpikäymällä saadaan aikaan muutos ohjelmaan. SIP:ssa toistetaan SC:tä, eli SIP on iteratiivinen malli. Rajlichilla on myös ryhmien toimintaan tarkoitettu malli Team Iterative Process, TIP. [Dorman and Rajlich, 2012]

SIP-iteraatiossa on viisi vaihetta:

1. Hankitaan, analysoidaan ja priorisoidaan vaatimukset tuotteen kehitysjonoksi.
2. Valitaan toteutettavat vaatimukset ja tehdään niistä iteraation kehitysjono (engl. *iteration backlog*).
3. Toteutetaan muutokset ohjelmamuutosprosessina vaatimus kerrallaan (SC).

4. Tallennetaan koodi välillä perushaaraan (vertaa PXP-mallin kehitys- ja refaktorointihaarat).
5. Testataan ohjelma ja tehdään julkaisu ennen seuraavaa iteraatiota.

Vaiheet on numeroitu myös kuvaan 12. Harmaa suorakulmio kuvaa sitä, että kehittäjä on vastuussa suorakulmion sisäpuolella olevista toiminnoista. Vaiheessa 1 saadaan käyttäjiltä (asiakkaat, yhteistyökumppanit) vaatimuksia, joista koostetaan tuotteen kehitysjono. Sykliä 2–3 toistetaan, kunnes iteraation vaatimukset on toteutettu. Vaatimus merkitään tehdyksi tai poistetaan kehitysjonosta. Sopivin väliajoin koodi tallennetaan perushaaraan (sykli 4). Kun iteraation kehitysjono on tehty, julkaistaan versio ja palataan tuotteen kehitysjonoon. Tätä kuvaa sykli 5. Julkaistu versio viedään käyttäjille testattavaksi. Käyttäjiltä saadun palautteen perusteella tuotteen kehitysjonoa päivitetään. [Rajlich, 2011]



Kuva 12. SIP-prosessi [Rajlich, 2011].

Ohjelmamuutosprosessia (SC) ennen vaatimukselle tehdään muutosanalyysi. Sen jälkeen toimitaan seuraavasti [Dorman and Rajlich, 2012]:

1. Paikannus (engl. *concept location*)
    - Etsitään koodista luokka, joka täytyy muuttaa.
2. Vaikutusanalyysi (engl. *impact analysis*)
    - Tutkitaan, mitkä muut luokat mahdollisesti tulevat muuttumaan tai poistumaan.
3. Aktualisointi (engl. *actualization*)
    - Tehdään muutokset koodiin.
4. Refaktorointi (engl. *refactoring*)
    - Muutetaan ohjelman luokkakaaviota.
    - Tehdään aktualisointia ennen (engl. *prefactoring*) tai sen jälkeen (engl. *postfactoring*).
5. Todennus (engl. *verification*)
    - Testataan, onko muutos tehty toimivasti.

6. Pohdinta

Hausta on voinut jäädä pois joitakin sopivia tutkimuksia. Esimerkiksi PSP:tä hyödyntäviä menetelmiä Shenin ja muiden [2013] kirjallisuuskatsauksessa oli 12. Tähän niistä löytyi itse katsauksen lisäksi 3. Tutkielman otsikkoa ja hakutuloksen kuvausta tulkitseva karsinta ei siten välttämättä riitä mukaanlukukriteeriksi. Shenin ja muiden katsauksen lähdeluettelosta toisaalta huomaa, että osa tutkimuksista ei viittaa lainkaan tämän tutkielman tarkoittamiin itsenäisen kehittäjän ketteriin menetelmiin (löydetty SCRUM-PSP [Rong, 2010] ei tarkemman lukemisen perusteella olekaan sellainen).

Shen ryhmineen oli käyttänyt katsauksessaan useampia tietokantoja kuin minä. Näitä olivat ISI Web of Science, ProQuest, ScienceDirect, SpringerLink, Kluwer Online, CiteseerX Library ja Wiley InterScience. Testihaku ”*lone agile programming*” SpringerLinkiin, ISI:in ja ProQuestiin, joista katsauksessa löytyi eniten tuloksia, ei kuitenkaan tuonut lisää tämän tutkielman tarkoitukseen liittyviä tutkimuksia.

Toteutin katsauksen yksin. Järjestelmällinen kirjallisuuskatsaus tehdään yleensä ryhmätyönä, jotta karsintaa, analyysia ja tulkintoja tekisi useampi henkilö. Tässä tutkielmassa pyrin vähentämään vinoumaa tekemällä kolme eri analyysiä (tiedonkeruut 1 ja 2, laatuarvio) ja eri aikaan, jotta tulkintojen välille tulisi ajallista etäisyyttä.

Yhdessäkään tutkimuksessa ei käytetty minkäänlaisia mittareita projektien onnistumisen arvioimiseen. Tutkielmien vähyyden ja tulosten pääasiallinen perustuminen tutkijan omaan kokemukseen aiheuttavat sen, että seuraavia johtopäätöksiä ei voi laajasti yleistää.

Tutkimuksissa toistuvat tietyt suositukset. Itsenäisen kehittäjän kannattaa käyttää versionhallintaa: lähetä usein, lähetä ajoissa. Ajan ja työmäärän hallintaan suositellaan Kanban-taulua ja ylitöiden välttämistä. Rahoitus kannattaa miettiä: projekti- vai tuntipalkka? Kummassakin tapauksessa on sovittava, mitkä asiat kuuluvat projektin palkkaukseen. Oletusvastaus lisävaatimuksiin on: ”ei”.

Iteratiivisuus lyhyine sykleineen on suositeltavaa, varsinkin jos asiakas ei täysin tiedä, mitä haluaa. Asiakkaan osallistaminen vaatimussuunnittelusta alkaen on tärkeää, samoin jatkuva palautteen pyytäminen. Älä tee kaikkea valmiiksi, vaan hanki palautetta julkaisemalla toimiva sovellus tärkeimmillä ominaisuuksilla. Asiakkaan edustajan pitää olla sellainen, jolla on valta päättää sovelluksen toiminnasta. Ilman asiakasta tekevän kannattaa julkaista kokeiluversioita.

Itsenäisenkin ohjelmoijan kannattaa ylläpitää luetteloa käsitteistä ja nimityksistä, joita projektissa käytetään ja noudattaa yleisiä ohjelmointistandardeja.

Viiteluettelo

[37signals, 2006] 37signals, *Getting Real. The smarter, faster, easier way to build a successful web application*. 37signals, 2006.

[Agarwal and Umphress, 2008] Ravikant Agarwal and David Umphress, Extreme programming for a single person team. In: *Proc. of the 46th Annual Southeast Regional Conference on XX* (2008), 82–87.

[Akpata and Riha, 2004] Edward Akpata and Karel Riha, Can extreme programming be used by a lone programmer? In: *Proc. of the 12th International Conference on Systems Integration* (2004), Prague, Czech Republic, 167–175.

[Ambler, 2014] Scott W. Ambler, *The Agile Unified Process*. Cited 8.2.2014 from: http://www.ambysoft.com/unifiedprocess/agileUP.html

[Beck, 1999] Kent Beck, *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Beck et al., 2001a] Kent Beck et al., *Ketterän ohjelmistokehityksen julistus*. Cited 28.1.2014 from: http://agilemanifesto.org/iso/fi/manifesto.html

[Beck et al., 2001b] Kent Beck et al., *Ketterän ohjelmistokehityksen 12 periaatetta*. Cited 28.1.2014 from: http://www.agilemanifesto.org/iso/fi/principles.html

[Cirillo, 2006] Francesco Cirillo, *The Pomodoro Technique.* Francesco Cirillo, 2006.

[Cockburn, 2004] Alistair Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley, 2004.

[Cronin, 2001] Gareth Cronin, *eXtreme Solo - A Case Study in Single Developer eXtreme Programming.* Master's thesis, University of Auckland, 2001.

[Curtis, 2001] Tex Curtis, So you wanna be a cowboy? *IEEE Software*, **18**, 2 (Mar./Apr. 2001), 110–111.

[Dent et al., 2008] Andrew Dent, Ashley Aitken and Sonya Rosbotham, From scrum to solo: how small is too small a team to still call it software engineering? In: *Proc. of the 19th Australian Software Engineering Conference: ASWEC 2008; Experience Report* (2008), Engineers Australia, 152–159.

[Dorman and Rajlich, 2012] Christopher Dorman and Václav Rajlich, Software change in the solo iterative process: an experience report. In: *Proc. of Agile Conference* (2012), 21–30.

[Dybå and Dingsøyr, 2008] Tore Dybå and Torgeir Dingsøyr, Empirical studies of agile software development: a systematic review. *Information and Software Technology* **50**, 9-10 (Aug. 2008), 833–859.

[Dzhurov et al., 2009] Yani Dzhurov, Iva Krasteva and Sylvia Ilieva, Personal extreme programming – an agile process for autonomous developers. In: *Proc. of International Conference on SOFTWARE, SERVICES & SEMANTIC TECHNOLOGIES* (Oct. 2009), Sofia, Bulgaria, 252–259.

[Eskelinen ja Karsikas, 2012] Harri Eskelinen ja Sami Karsikas, *Tutkimusmetodiikan perusteet - tekniikan alan oppikirja*. Lappeenrannan teknillisen yliopiston koulutus- ja kehittämiskeskus, julkaisu 12, 2012.

[Fink, 2005] Arlene Fink, *Conducting Research Literature Reviews: From the Internet to the Paper*. Sage Publications, Inc., 2005.

[Google, 2014] Google, *Search operators*. Google 2014. Cited 4.3.2014 from: https://support.google.com/websearch/answer/136861?hl=en&ref_topic=3180167

[Hollar, 2006] Ashby Brooks Hollar, *Cowboy: An Agile Programming Methodology for a Solo Programmer*. Master's thesis, Virginia Commonwealth University, 2006.

[Humphrey, 2000] Watts S. Humphrey: *The Personal Software Process (PSP)*. Technical report, Carnegie Mellon University, 2000.

[James, 2014] Michael James, *Scrum Reference Card*. Cited 8.2.2014 from: http://scrumreferencecard.com/scrum-reference-card/

[Kitchenham et al., 2007] Barbara Kitchenham, Stuart Charters, David Budgen, Pearl Brereton, Mark Turner, Steve Linkman, Magne Jørgensen, Emilia Mendes and Giuseppe Visaggio, *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical report, EBSE, 2007.

[Moweble, 2013] *The Pros And Cons Of Cowboy Coding*. MoWeble 11.6.2013. Cited 18.1.2014 from: http://www.moweble.com/the-pros-and-cons-of-cowboy-coding.html

[Noel, 2004] Noel: *Cowboy Coders and the Hero Programmer Culture*. Games from Within -blog 13.3.2004. Cited 18.1.2014 from: http://gamesfromwithin.com/cowboy-coders-and-the-hero-programmer-culture

[Nyström, 2011] Anna Nyström, *Defining and Evaluating an Agile Software Development Process for a Single Software Developer*. Master's thesis, Chalmers University of Technology, 2011.

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

[Petersen et al., 2008] Kai Petersen, Robert Feldt, Shahid Mujtaba and Michael Mattsson, Systematic mapping studies in software engineering. In: *Proc. of the 12th International Conference on Evaluation and Assessment in Software Engineering* **17** (2008), British Computer Society Swinton, 68-77.

[Peteva, 2013] Dima Peteva, *Siteground Staging - Don't be a Cowboy Coder!* 28.3.2013. Cited 18.1.2014 from: http://blog.siteground.com/siteground-staging/

[Poppendieck and Poppendieck, 2003] Mary Poppendieck and Tom Poppendieck, *Lean Software Development – An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

[Rajlich, 2011] Václav Rajlich, *Software Engineering : The Current Practice.* Chapman and Hall, 2011.

[Rong et al., 2010] Guoping Rong, Dong Shao and He Zhang, SCRUM-PSP: embracing process agility and discipline. In: *Proc. of the 17th Asia Pasific Software Engineering Conference* (2010), 316–325.

[Rönkkö and Peltonen, 2012] Mikko Rönkkö and Juhana Peltonen, *Software Industry Survey 2012*. Aalto University, 2012.

[Salminen, 2011] Ari Salminen, *Mikä kirjallisuuskatsaus? Johdatus kirjallisuuskatsauksen tyyppeihin ja hallintotieteellisiin sovelluksiin*. Vaasan yliopiston julkaisuja, 2011.

[Schuh, 2005] Peter Schuh, *Integrating Agile Development in the Real World*. Charles River Media, 2005.

[Schwaber and Beedle, 2001] Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum.* Upper Saddle River: Prentice Hall 2001.

[Shen et al., 2013] Mengjiao Shen, Guoping Rong and Dong Shao, Integrating PSP with agile process: a systematic review. *Advanced Materials Research* **765-767** (2013), 1697–1703.

[Sletholt et al., 2011] Magnus Sletholt, Jo Hannay, Dietmar Pfahl, Hans Benestad and Hans Langtangen, A literature review of agile practices and their effects in scientific software development, In: *Proc. of the 4th International Workshop on Software Engineering for Computational Science and Engineering (SECSE '11)* (May 2011), ACM, 1–9.

[Sommerville, 2008] Ian Sommerville, *Software Engineering*. 8. ed. Addison-Wesley, 2008.

[Stapleton, 2003] Jennifer Stapleton, *DSDM: Business Focused Development.* Pearson Education, 2003.

[Yli-Rohdainen, 2012] Kimmo Yli-Rohdainen, *Testivetoisen kehityksen hyödyntäminen sulautetun tiedonkeruuohjelmiston toteuttamisessa.* Diplomityö, Tampereen teknillinen yliopisto, 2012.

Liite 1. Tiedonkeruulomake 1

Asia	Tieto	Lisähuomiot
Analysointipäivä		
Tutkimuksen nimi		
Tekijä		
Vuosi		
Tyyppi		
Pohjamenetelmä(t)		
Sivumäärä		

Liite 2. Tiedonkeruulomake 2

Aihe	Tieto	Lisähuomiot
Analysointipäivä		
Tutkimuksen nimi ja tekijä, vuosi		
Mistä pohjamenetelmästä koostettu? Miten?		
Koostamisen perustelut		
Menetelmän keskeiset ketterät käytännöt		
Keskeiset tulokset		
Asiakkaan huomioiminen		
Tapaustutkimus: Mitä tutkittiin? Vastaako tutkimuskysymyksiin?		

Liite 3. Laadunarviointilomake

Laatu	Tieto	Lisähuomiot
Analysointipäivä		
Tutkimuksen nimi ja tekijä, vuosi		
Tutkimuksen luokka (esim. gradu, tutkimusartikkeli)		
Tausta Tutkittava ilmiö määritelty?		
Kirjallisuuskatsaus tehty?		
Tavoitteet määritelty?		
Menetelmät Tutkimusmenetelmä kuvattu selkeästi?		
Tulokset Tulokset esitetty selkeästi?		
Tietojen ja tulkintojen yhteys nähtävillä?		
Muuta Luotettavuutta arvioitu?		
Yleistettävyyttä arvioitu?		
Lähteiden tyypit?	lehtiartikkelit, kirjat, opinnäytteet, verkkosivut, viestintä	

Liite 4. Verkkosivuja itsenäisen kehittäjän menetelmistä

StackExchange-keskustelusivusto

Sivu on esimerkki keskustelusivusta. Kysyjä kertoo olevansa itsenäinen kehittäjä ja kysyy, onko mitään ketteriä menetelmiä häntä varten. Eräs vastaaja ehdottaa erilaisia käytäntöjä: säännölliset asiakastapaamiset, tuotteen kehitysjonon ylläpitäminen, yksikkötestaukset ja itsearviointi. Toinen linkittää Personal XP -artikkeliin ja kolmas kertoo Flying Donut -nimisestä Scrum-projektinhallinnan ohjelmasta.

http://programmers.stackexchange.com/questions/141818/how-can-a-single-developer-make-use-of-agile-methods

An Agile Case Study - Using Agile for FileMaker Development Projects

Jason Mundock esittelee Scrum-pohjaisen soolomenetelmänsä tässä ei-tieteellisenä pois rajautuneessa tapaustutkimuksessaan. Kehittäjällä on asiakas, mutta muuten kehitystyö tapahtuu itsenäisesti. Varsinkin vaatimusten kerääminen ja aikataulutus onnistuivat.

http://www.jasonmundok.com/papers/GEN025_AnAgileCaseStudy.pdf

Personal Kanban

Sivusto esittelee Kanban-menetelmän, jota voi soveltaa henkilökohtaisesti. Se perustuu kahteen sääntöön: visualisoi työsi ja rajoita tekeillä olevia tehtäviäsi.

http://www.personalkanban.com

AgileZen

AgileZen myös itsenäiselle kehittäjälle sopivat projektinhallintasovellus.

http://www.agilezen.com

Best Practices for the Solo Developer

Esitelmässä kerrotaan itsenäiselle kehittäjälle sopivista hyvistä käytänteistä: käytä ulkoista versionhallintaa, pidä kirjaa töistäsi ja ajankäytöstäsi (esimerkiksi AgileZen-ohjelmalla).

http://www.slideshare.net/mjeaton/best-practices-for-the-solo-developer

A Force of One - Agile and the Solo Developer

Kirjoittaja esittelee hyviä käytäntöjä: käyttäjätarinat, käytä tuotteen kehitysjonoa, julkaise usein, tee yhteistyötä asiakkaan kanssa, yksinkertainen suunnittelu, Kanban: visualisoi tehtävät ja vältä ylitöitä.

http://www.slideshare.net/clintedmonson/a-force-of-one-agile-and-the-solo-developer