Ketterien ohjelmistokehitys menetelmien vertailu: Scrum ja Extreme Programming

Ville Hotakainen

Haaga-Helia ammattikorkeakoulu

Amk-opinnäytetyö

2021

Tietojenkäsittelyn koulutusohjelma

Tiivistelmä

Tekijä(t)
Ville Hotakainen

Tutkinto
Tradenomi

Opinnäytetyön nimi
Ketterien ohjelmistokehitys menetelmien vertailu: Scrum ja Extreme Programming

Sivu- ja liitesivumäärä
26

Tässä opinnäytetyössä tutkitaan kahta ketterää menetelmää: Scrum ja Extreme Programming (XP). Kummastakin menetelmästä selvitettiin eroavaisuuksia sekä niiden projektisoveltuvuutta ohjelmistotuotannossa käytettäviin projekteihin.

Ketterien menetelmien havaittiin syntyneen perinteisten menetelmien puutteista vastata nopeasti muuttuviin vaatimuksiin ohjelmistokehityksessä. Menetelmät pyrkivät tuomaan asiakkaan lähemmäksi kehitysprosessia, joka vähentää turhien suunniteluiden toteuttamista. Menetelmät pyrkivät tuomaan asiakkaan lähemmäksi kehitysprosessia, joka vähentää turhien suunniteluiden toteuttamista.

Tutkimuksen perusteella voidaan huomata, että vaikka menetelmät pohjautuvat ketterien menetelmien ideologiasta, ne eroavat huomattavasti toisistaan. Scrum pyrkii hallinnoimaan yrityksen projektinhallintaa ja sitä kautta parantamaan tuotettavuutta, kun taas XP keskittyy enemmänkin teknillisten käytäntöjen kautta toteuttamaan konkreettisia tuotoksia asiakkaalle mahdollisimman nopeasti.

Ohjelmistoprojekteja on hyvin erilaisia, joten valitun menetelmän täytyy soveltua projektin käyttöön. Scrum sekä Extreme Programming perustuvat ketteristä menetelmistä, mutta niiden käyttötarkoitukset ovat erilaisia. Scrum on menetelmistä yleiskäyttöisempi. Menetelmä keskittyy projektinhallintaan, kun taas Extreme Programming on suunniteltu auttamaan kehittäjiä, jotka työskentelevät tuotekehityksen parissa.

Asiasanat

Ohjelmistokehitys menetelmät, Perinteiset menetelmät, Ketterät menetelmät, Scrum, Extreme Programming.

Sisällys

1 Johdanto

Tässä opinnäytetyössä tutkitaan kahta ohjelmistokehitys menetelmää Scrum sekä Extreme Programming. Tutkimuksessa tavoite on selvittää valittujen ohjelmistokehitys menetelmien erovaisuuksia, jonka avulla pystytään osoittamaan millaisiin ohjelmistoprojekteihin kyseiset menetelmät soveltuvat. Aihe on erityisesti ajankohtainen aloittelevalle ohjelmistokehittäjälle, joka tulee kohtamaan molempien menetelmien käytäntöjä sekä arvoja työympäristössä. Tunnistamalle valittujen menetelmien prosesseja yksityiskohtaisemmin voidaan mahdollistaa lukijalle ymmärrys, kuinka molemmissa menetelmissä toimitaan. Tutkimukseen valittujen ketterien menetelmien valinta perustui tekijän omien kiinnostuksen kautta lisätä ymmärrystä valituista menetelmistä.

Tutkimuksessa ensimmäisenä tutustutaan ohjelmistotuotannon määrittelyyn, joka pohjustaa ohjelmistotuotanto prosessin sekä ohjelmistokehitys menetelmät. Tämän jälkeen perehdytään perinteisiin menetelmiin. Menetelmistä perehdytään erityisesti vesiputous, inkrementaalinen sekä evoluutiomalleihin, sillä ne edustavat perinteisten menetelmien erilaisia lähtökohtia ja ne ovat oleellisia ketterien menetelmien syntyyn. Viimeiseksi perehdytään ketteriin menetelmiin sekä valittujen ketterien ohjelmistokehitysmenetelmien arvoihin, käytäntöihin sekä elinkaareen.

Tutkimus toteutetaan laadullisena tutkimuksena. Tutkimuksen tietoperusta koostuu virallisista ja luotettavista lähteistä, joilla vastataan seuraaviin tutkimusongelmiin:

- Mistä ja miksi ketterät menetelmät ovat syntyneet?
- Miten Scrum ja Extreme Programming eroavat toisistaan?

1.1 Opinnäytetyön rakenne

Tämän työn ensimmäinen luku koostuu johdannosta, joka sisältää opinnäytetyön tavoitteet, rajaukset, rakenteen sekä sanaston. Opinnäytteen luvussa 2 käydään läpi ohjelmistotuotannon määritelmää sen elinkaari sekä pohjustetaan ohjelmistotuotannossa käytettäviä käsitteitä. Esitellään perinteisiä ohjelmistotuotannon menetelmiä, syvennytään tarkemmin eri menetelmien hyviin sekä huonoihin puoliin sekä niiden toiminnallisuuksiin. Luvussa 3 käydään läpi ketteriä menetelmiä ja pohjustetaan opinnäytteeseen valitut menetelmät Scrum sekä Extreme Programming. Luvussa 4 vertaillaan kahta opinnäytetyöhön valittua ketterää ohjelmistokehitys menetelmää ja tutkitaan saatuja tuloksia. Lopuksi luku 5 sisältää pohdinnan ja johtopäätökset.

1.2 Sanasto

Bugi	Ohjelmistovirhe, joka aiheuttaa virheellisen tuotoksen ohjelmalle. (IEEE 1990, 31).
Inkrementaalinen kehitys	Ohjelmistokehitys tekniikka, jossa vaatimusten määrittely, design, toteutus ja testaus tapahtuvat päällekkäin toteutettavina iteraatioina. (IEEE 1990, 39).
Iteraatio	Toistuvasti suoritettava vaiheiden sarja (IEEE 1990, 43).
Lähdekoodi	Tekstilista komennoista sopivassa muodossa tietokoneohjelman suoritettavaksi. (IEEE 1990, 68).
Ohjelmistokehitys prosessi	Prosessi, jolla käyttäjän tarve käännetään ohjelmistotuotteeksi (IEEE 1990, 67).
Ohjelmiston elinkaari	Ajanjakso, joka alkaa ohjelmistotuotteen suunnittelusta ja päättyy, kun ohjelmisto ei ole enää käytettävissä (IEEE 1990, 68).
Ohjelmisto-ominaisuus	Eroteltava ominaisuus ohjelmistokohteesta (IEEE 1990, 67).
Ohjelmistotuote	Tietokoneohjelmien, menettelyjen ja niihin liittyvien asiakirjojen joukko, joka toimitetaan käyttäjälle (IEEE 1990, 68).
Prosessi	Sekvenssi suoritetuista vaiheista tiettyyn tarkoitukseen (IEEE 1990, 57).
Refaktorointi	tarkoittaa prosessia, jossa nykyistä lähdekoodia muokataan niin, että sen toiminnallisuus pysyy samana, mutta rakenne muuttuu. Yleisesti tarkoitetaan koodin siistimistä ja ohjelmointivirheiden korjaamista. (Fowler 2018).

Testi	Toiminta, jossa järjestelmä tai komponentti suoritetaan tietyissä olosuhteissa, tulokset havaitaan ja kirjataan (IEEE 1990, 74).

2 Ohjelmistotuotannosta perinteisiin menetelmiin

Ohjelmistotuotanto käsitteenä ilmestyi ensimmäisen kerran vuonna 1968 Naton järjestämässä konferenssissa, jossa keskusteltiin sen ajan hankaluuksista kehittää suuria ja monimutkaisia tietojärjestelmiä. Ehdotettiin, että insinöörimäisen lähestymistavan omaksuminen ohjelmistokehitykseen vähentäisi kustannuksia ja johtaisi luotettavampiin ohjelmistoihin. (Sommerville 2007, 5.)

Nykypäivänä termistä löytyy paljon erilaisia määritelmiä eri lähteistä, mutta esimerkiksi IEEE (Institute of Electrical and Electronics Engineers), joka on yksi suurimmista teknologia-alan vaikuttajista, määrittelee termin seuraavasti:

> “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software”. (IEEE 1990, 67).

Määritelmä suomennettuna tarkoittaa, että ohjelmistotuotanto on systemaattista, kurinalaista, ja mitattavissa oleva tapa ohjelmistojen kehittämiseen, operointiin sekä ylläpitoon. Vuonna 2004 valmistunut ja 2014 päivitetty SWEBOK eli Software Engineering Body of Knowledge (Bourque & Fairley 2014) julkaisu, joka on yleisesti hyväksytty lähde ohjelmistotuotannon määrittelyyn, listaa ohjelmistotuotannon käytäntöjen jakautuvan seuraaviin osa-alueisiin:

1. Ohjelmistovaatimukset ilmaisevat ohjelmistotuotteelle asetettuja vaatimuksia sekä tarpeita, joita lopputuotteen tulisi noudattaa.

2. Ohjelmiston muotoilu tarkoittaa ohjelmistovaatimusten analysoinnissa kuvattujen ohjelmistojen sisäisten rakenteiden suunnittelua.

3. Ohjelmiston rakennus viittaa ohjelmistojen yksityiskohtaiseen luomiseen yhdistämällä yksityiskohtaista suunnittelua, koodausta, yksikkötestausta, integraatiotestausta, virheenkorjausta ja todentamista.

4. Ohjelmistotestaus on toiminto, jolla varmistutaan tuotteen oikeanlaisesta toimivuudesta sekä vähennetään mahdollisia virhetiloja.

5. Ohjelmiston ylläpito tarkoittaa ohjelmiston mukautumista uusiin toimintoihin, nykyisten ominaisuuksien parantamista sekä refaktorointia.

Kuva 1. Ohjelmistotuotannon osa-alueet (mukaillen SFIA 2014)

6. Ohjelmiston kokoonpanohallintaprosessi huolehtii ohjelmistojen rakentamisessa käytettyjen kirjastojen, laitteistojen ja laiteohjelmistojen ohjelmistoon tekemien muutosten syystemaattisesta hallinnasta.

7. Ohjelmistotuotannon hallinta tarkoittaa suunnittelua, koordinointia, raportointia ja projektinhallintaa, jota ohjelmistojen tekemiseen tarvitaan.

8. Ohjelmistotuotantoprosessi pitää sisällään toimintoja, kuten suunnittelu, vaatimus-määrittely, rakentaminen sekä testaus, jotka edistävät ohjelmiston toteutumista.

9. Ohjelmistotuotannon mallien ja metodien tavoitteina on tehdä ohjelmistojen raken-tamisen toiminnasta järjestelmällistä ja toistettavaa. Tarjotaan lähestymistapoja ongelmanratkaisuun sekä menetelmiä ohjelmiston rakentamiseen ja analysoimi-seen.

10. Ohjelmiston laatua voidaan pitää laajana käsitteenä, jolla tarkoitetaan ohjelmisto-tuotteen kykyä vastata asiakkaan tai loppukäyttäjän tarpeita, toiveita ja odotuksia.

11. Ohjelmistotuotannon ammattitaitoiset opit viittaavat ammattitaitoon ja asenteeseen, joka ohjelmoijalla on omattava toimiakseen ammattimaisesti, vastuullisesti ja eettisesti työssään.

12. Ohjelmistotuotannon ekonomia viittaa ohjelmistojen teknillisten ominaisuuksien sopivuuden liiketoiminnan tavoitteisiin.

Ohjelmistotuotantoon kuuluu paljon erilaisia vaiheita ja prosesseja. SWEBOK (2014) on jakanut osa-alueisiin vielä yksityiskohtaisemmin vaatimuksia, joita sisäisten sidosryhmien tulisi toteuttaa eri ohjelmistotuotannon vaiheissa. Nämä vaatimukset riippuvat yksilöllisesti organisaation luonteesta, tiimirakenteista sekä ohjelmoijien osaamisesta.

2.1 Ohjelmistokehityksen vaiheet ja elinkaari

Ohjelmistokehityksen voi jaotella erilaisiin vaiheisiin, jotka yhdessä muodostavat ohjelmistokehityksen elinkaaren. Ohjelmistokehityksen elinkaari tarkoittaa ohjelmistojen systemaattista kehittämistä, jolla parannetaan todennäköisyyttä, että ohjelmistotuote vastaa haluttua laatua sekä valmistumista määräajan puitteissa. Lyhyesti voidaan määritellä, että ohjelmistokehityksen elinkaaren tavoitteena on vähentää riskejä ja kustannuksia suoraviivaistamalla kehitysprosessia. Tyypillisesti elinkaari muodostuu seuraavista vaiheista: Vaatimusmäärittely, Ohjelmiston muotoilu, Ohjelmointi, Testaus, Julkaisu ja ylläpitö. (Loubser 2021, luku 9.)

Elinkaaren vaiheita toteutetaan eri tavoin, tarpeiden mukaan, mutta kaikki ohjelmat käyvät mainitut vaiheet läpi. Jokainen erilainen lähestymistapa vaiheiden toteuttamiseen tunnetaan ohjelmistokehityksen elinkaarimallina. Mallit kuvaavat, kuinka ohjelmistoja kehitetään tai tulisi kehittää. Ohjelmistokehityksen elinkaarimallit jaetaan kahteen lähestymistapaan; perinteisiin sekä ketteriin menetelmiin. Suurin ero malleissa on niiden lähestyminen elinkaaren vaiheisiin. (Dooley 2011, luku 2.)

Perinteisissä menetelmissä kehitysprosessi on yleensä tiukempi vaiheiden ja julkaisujen suhteen. Tyypillisesti mallissa vaiheet suunnitellaan ja dokumentoidaan tarkasti etukäteen. Laadittua suunnitelmaa noudatetaan systemaattisesti ja vaiheittain. Jokainen vaihe suoritetaan yksi kerrallaan loppuun ennen kuin seuraavaan vaiheeseen siirtyminen sallitaan. (Dooley 2011, luku 2). Kun taas Ketterät menetelmät toimivat inkrementaalisesti eli, vaiheita toteutetaan pienissä yhteen sulautetuissa osissa ja usein tapahtuvissa julkaisuissa, joita kutsutaan iteraatioiksi. Ketterien menetelmien tarkoituksena on mahdollistaa nopea reagointi muuttuviin vaatimuksiin. (Marshall & Bruno 2009, luku 1.)

2.2 Perinteiset menetelmät

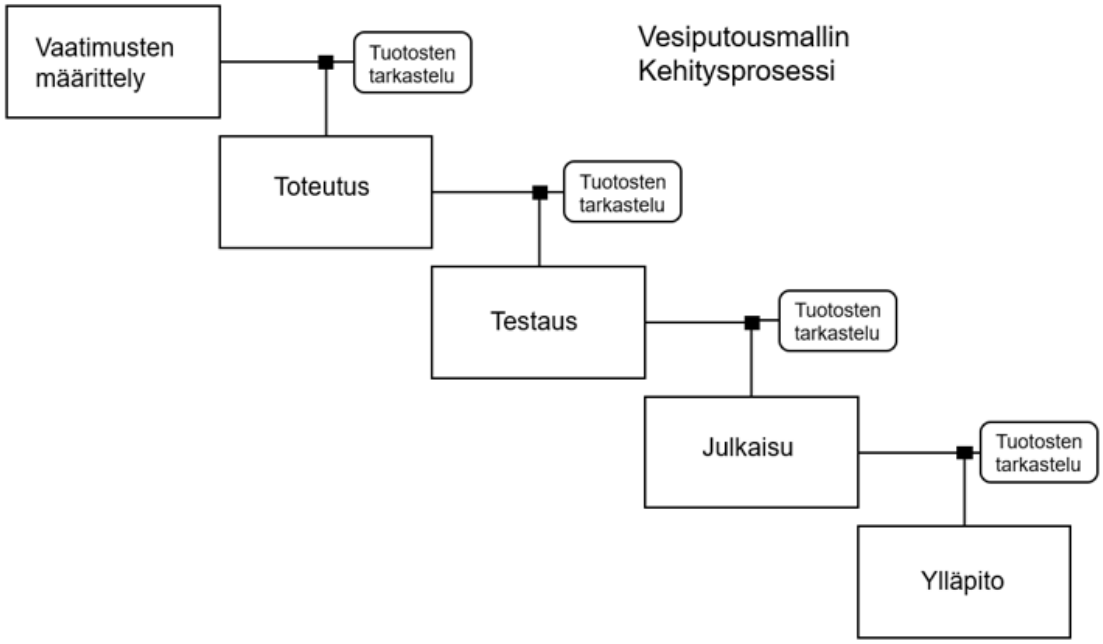
Perinteisiin menetelmiin luokitellaan menetelmiä tai malleja, jotka ovat ns. suunnitelmavetoisia prosesseja. Suunnitelmavetoiset prosessit ovat prosesseja, joissa prosessitoiminnot suunnitellaan etukäteen ja edistymistä mitataan tähän suunnitelmaan nähden (Sommerville 2011, 29.) Menetelmien elinkaaren vaiheita kuvataan vaihemallisina toimintajaksoina, kuten vaatimusten, suunnittelun ja toteutuksen määrittelynä, joissa ei oteta kantaa tai paneuduta tarkemmin vaiheiden sisällöllisiin käytäntöihin. Perinteisissä ohjelmistokehitysprosessimalleissa kehitys etenee toistamalla peräkkäisiä toimintoja, joissa vaatimukset edeltävät suunnittelua, toteutus ennen testausta ja niin edelleen. (Scacchi 2001, 5; Widrig & Leffingwell 2003, 3.)

Yleensä, kun puhutaan perinteisistä ohjelmistokehityksen menetelmistä, tarkoitetaan itseasiassa vesiputousmallia. Vesiputousmallin kehittäjänä pidetään Winston Roycea, joka pohdiskeli artikkelissaan (1970, 329–335) isojen ohjelmistoprojektien kehittämiseen liittyviä ongelmia. Artikkelissa Royce esittelee prosessimallin, jonka ideana on suorittaa elinkaaren vaiheita lineaarisesti.

Vesiputousmallin ohjelmistokehitys prosessi alkaa, kun asiakas on määritellyt ohjelmistotuotteelle vaatimukset halutuista toiminnallisuuksista, jotka kehittäjätiimi vastaanottaa. Vaatimukset dokumentoidaan vaatimusasiakirjaan. Vaatimusten perusteella on kehittäjätiimin tarkoituksena suunnitella ohjelmistokehys, joka muodostaa teknisen rungon rakennettavalle ohjelmalle. Ennen varsinaisen koodaamisen aloittamista suunnitelman tarkastavat eri hallinnalliset osat, jotka varmistavat vaatimusten vastaavan kehittäjien laatimaa suunnitelmaa.

Vesiputousmallille on tyypillistä suorittaa tarkastuksia aina ennen kuin siirrytään prosessissa seuraavaan vaiheeseen. Jokaisesta vaiheesta syntyy paljon dokumentaatiota, joten vesiputousmallia saatetaan kutsua myös dokumentaatioon perustuvaksi kehitykseksi. Tarkastuksien tarkoituksena on huolehtia laadun säilyvyydestä, vertailemalla saatuja tuloksia vaadittuihin tuloksiin. (Tsui 2013, 60; Petersen, Wohlin & Baca 2009, 3.)

Kehitysvaiheessa vaatimukset pannaan täytäntöön koodin muodossa. Kehitysvaiheen jälkeen siirrytään testaukseen. Testauksen suorittaa yleensä erillinen testausryhmä, joka tarkastaa alkuperäisten vaatimusten täsmäävän sen hetkistä järjestelmää. Tavoitteena on varmistua, että kaikki vaatimukset huomioidaan ja, että järjestelmä toimii tavoitellulla tavalla. Testauksesta järjestelmä julkaistaan käyttäjille, josta järjestelmä siirtyy prosessin viimeiseen vaiheeseen, joka on ylläpito. Ylläpito huolehtii bugien korjaamisesta sekä mahdollisten pienten toiminallisuuksien lisäämistä järjestelmään, mikäli asiakas niin vaatii. (Crookshank 2015, luku 4; Sommerville 2011, 31.)



Kuva 2. Tyypillinen vesiputousmallin kehitysprosessi. (mukaillen Petersen, Wohlin & Baca. 2009)

Vesiputousmalliin on kohdistunut paljon kritiikkiä varsinkin sen jäykkyyden takia. Ohjelmistoprojektit ovat alttiita muutoksille, joihin vesiputousmallin on vaikea vastata. Vesiputousmallia noudattavien ohjelmistoprojektien ongelmaksi on muodostunut alustavien vaatimusten suunnittelu niin huolellisesti, ettei muutoksia projektin edetessä tarvitsisi tehdä. Asiakkaiden tarpeet muuttuvat, joten oletettavasti aikaisemmat määritetyt vaatimukset eivät vastaa uusia. Vesiputousmalli ei anna kehittäjille mahdollisuutta palata projektissa aikaisempiin vaiheisiin, joissa muutoksia tarvitsisi tehdä. Joten uusien ominaisuuksien lisääminen tulee erittäin vaikeaksi. (Lynch 2019; Ingeno 2018.)

Testauksen suorittaminen valmiille tuotteelle tuottaa myös ongelmia, kun teknilliset ongelmat havaitaan liian myöhään. Lopullisen tuotteen muokkaaminen pakottaa usein vaiheiden läpikäyntiä uudelleen, joka pitkittää projektin valmistumista. Vikojen korjaaminen saattaa tulla hyvinkin kalliiksi, sillä ongelmat mahdollisesti pakottavat muuttamaan tuotoksen teknillistä rakennetta perusteellisesti. On myös mahdollista, että alkuperäiset vaatimukset tulkitaan lähtökohtaisesti väärin tai asiakas ei ole täysin ymmärtänyt omia vaatimuksia, joten projekti on alun pitäen viallinen. (Fowler 2005; Ingeno 2018.)

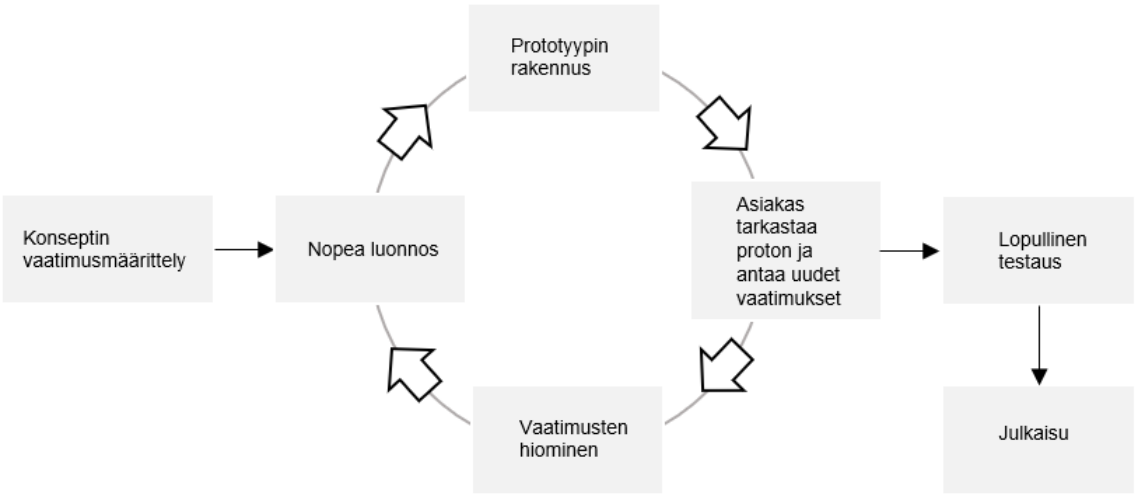
Nykypäivänä ohjelmistotyö on usein nopeatahtista, joka vaatii muutoksiin nopeaa reagoimista. Muutoksiin vaikuttaa yleensä bisnesympäristön kasvava kilpailu. Vesiputousmallin on vaikea reagoida nopeasti muuttuviin vaatimuksiin, joten joustavampia elinkaarimalleja tarvitaan. (Sommerville 2007, 392; Ingeno 2018.)

Inkrementaalinen tai vaiheittainen toimitusmalli jakaa vesiputousmallille tyypillisen lineaarisen ohjelmistokehityksen prosessin pienempiin yksilöllisiin vaiheisiin, jossa järjestelmää rakennetaan pieni pala kerrallaan. Järjestelmää kehitetään yksi kokonaisuus kerrallaan, jossa samaistuvat lineaariselle ohjelmistoprosessille ominaiset kehitysvaiheet, kuten vaatimusten analysointi, suunnittelu, ohjelmointi, testaus ja julkaisu. Vaiheen julkaisun jälkeen jatketaan aina uudella vaiheella, joka pohjautuu aikaisemmasta julkaisusta. Tulevat vaiheet lisäävät aina uusia toiminnallisuuksia aikaisempaan rakennettavaan tuotteeseen, kunnes toiminnallisuudet vastaavat asiakkaan vaatimuksia. (McConnel 1996, luku 7; Tsui 2013, 61; Dutt & Subramanian 2015, luku 1.)

Etu perinteiseen vesiputousmalliin verrattuna on se, että inkrementaalinen malli mahdollistaa toimivien kokonaisuuksien viennin asiakkaalle huomattavasti nopeammin. Inkrementaalinen malli ei kuitenkaan lyhennä ohjelmistotuotteen rakentamiseen tarvittavaa aikaa, mutta se mahdollistaa asiakkaalle läpinäkyvämmän kuvan projektin edistymisestä. Asiakkaan ei tarvitse odottaa projektin päättymistä ennen kuin ohjelmistoa voidaan käyttää. Jakamalle prosessit pienempiin osiin, pystytään myös paremmin puuttumaan teknillisiin ongelmiin lisääntyvän testaamisen ansiosta. Julkaisujen tai toimitusten ajankohtien arviointi myös helpottuu, kun niitä tehdään useimmin kuin vain kerran projektin loppupuolella. (McConnel 1996, luku 7 & 36; Tsui 2013, 61.)

Malli on silti melko jäykkä ja raskas, kuten vesiputousmalli. Useat pienet julkaisut vaativat silti yhtä paljon suunnittelua ja sovellusalueen ymmärtämistä, kuin vesiputousmallikin. Vaikka julkaisut mahdollistavat tärkeän palautteen saamista asiakkailta, ei niihin välttämättä pystytä vaikuttamaan, sillä menetelmä ei mahdollista aikaisempien vaiheiden toteutuksien muuttamista. Inkrementaalinen malli ei siis sovellu projekteihin, jossa ei varmuudella tiedetä, että millaisia ominaisuuksia rakennettavalla tuotteella tulisi olla. Näiden ongelmien ratkaisemiseksi alkoi syntyä uusia menetelmiä. (McConnel 1996, luku 36; Tsui 2013, 62; Dooley 2011.)

Aikaisempien menetelmien ongelmaksi muodostui epäselvien ja muutosherkkien vaatimusten ratkaiseminen tuntemattomissa sovellusalueissa. Menetelmien käyttäjät tämän huomasivat, mutta käytössä olevat menetelmät eivät riittäneet kyseisten ongelmien ratkaisemiseen. Näiden ongelmien ratkaisemiseksi soveltui evoluutioprototyyppi malli, jonka tavoitteena oli rakentaa järjestelmiä vain tiedossa olevien vaatimusten perusteella sekä parannella aikaisempien tuotosten toiminnallisuuksia asiakas- sekä käyttäjäpalautteen avustuksella (Martinez 2020.)



Kuva 4. Evoluutioprototyyppimallin elinkaari. (Mukaillen Dooley 2011)

Prototyyppi on tuotteen tai järjestelmän kehittämiseksi tarkoitettu prosessi, joka jäljittelee tai simuloi todellista tuotetta tai järjestelmää. Periaatteessa prototyyppejä käytetään asiakaspalautteen saamiseksi, jolla voidaan paremmin varmistua lopullisen tuotteen suunnittelun täsmäävän asiakkaan tarpeita ennen lopullisen tuotteen julkaisua. Prototyyppien valmistus alkaa asiakkaalta saatujen vaatimusten toteuttamista toimiviksi ominaisuuksiksi. Nämä ominaisuudet esitetään asiakkaalle, josta pyritään saamaan palautetta seuraavien versioiden parantamista varten. Tätä prosessia toistetaan niin kauan, että asiakas sekä prototyyppien kehittäjät ovat tyytyväisiä lopputuloksesta (Martinez 2020.)

Evoluutioprototyyppimalli soveltuu erinomaisesti tilanteisiin, jossa kokeillaan uutta tuotetta tai tekniikkaa, jota ei tällä hetkellä tunneta selvästi. Tällaisessa tilanteessa vaatimukset kirkastuvat vasta kun, asiakkaat pääsevät konkreettisesti käyttämään kehitteillä olevaa järjestelmää. Käytön jälkeen ymmärrys järjestelmästä paranee, joka mahdollistaa järjestelmän oikeanlaisen muokattavuuden. Haittapuolena prototyyppimallille kuitenkin on se, että projektin alussa on mahdotonta tietää, kuinka kauan kestää, että tuote on asiakkaan määrittelyjen mukaan valmis. Joten malli on taipuvainen epärealistisiin aikatauluihin sekä budjetin ylityksiin. Evoluutioprototyyppi malli oli lopulta se menetelmä, joka kehittyi nykyaikaiseksi ketteräksi kehitysprosessiksi. (Martinez 2020; Dooley 2011, luku 2.)

3 Ketterät menetelmät

Perinteisten menetelmien uskottiin vielä 90-luvun alussa olevan tehokkain ja varmin tapa kehittää ohjelmistoja. Näin olivat päätelleet ohjelmistokehittäjät, jotka työskentelivät suurten ja pitkäkestoisten ohjelmistoprojektien kanssa. Oli yleistä, että projektissa työskenteli monta yritystä ja tiimiä eri puolilta maailmaa. Kun perinteisiä menetelmiä alettiin käyttämään pienemmissä projekteissa, havaittiin ongelmia. Huomattiin, että suunnitelmavetoisten eli perinteisten ohjelmistokehitys menetelmien lähestymistavat käyttävät suuria määriä aikaa sekä resursseja suunnitteluun ja dokumentointiin. Oli tavallista, että ohjelmistokehitysprosessi koostuu enimmäkseen tavoista selvittää kuinka ohjelmistoa tulisi kehittää, kuin itse kehittämisestä. Prosessimallit tukivat erityisesti laajojen, pitkäikäisten ohjelmistojen kehitystyötä, mutta pieniin ja keskisuuriin ohjelmistoprojekteihin ne osoittautuivat usein turhan jäykiksi. Perinteisten menetelmien vastapainoksi alkoi syntyä uudenlaisia ohjelmistokehitys menetelmiä, joiden edellytykset muuttuvien vaatimusten ratkaisuun sekä nopeampaan järjestelmien julkaisu tahtiin olivat parempia aikaisempiin suunnitelmavetoisiin menetelmiin verrattuna. (Sommerville 2011, 58–59; Dooley 2011, luku 2.)

Näitä uusia menetelmiä alettiin kutsua ketteriksi menetelmiksi. Menetelmien tyypillisiä ominaisuuksia ovat lyhyet elinkaaret, itseorganisoituvat tiimit, kasvoittain tapahtuva kommunikaatio sekä asiakkaan liittäminen kehitysprosessiin. (Marshall ym. 2009, luku 1). Ketterät menetelmät noudattavat inkrementaalista ja iteratiivista lähestymistapaa ohjelmistojen määrittämiseen, kehitykseen sekä tuotteen toimittamiseen. Ne soveltuvat parhaiten tilanteisiin, jossa järjestelmän vaatimukset muuttuvat nopeasti kehitysprosessin aikana. Menetelmien tarkoituksena on toimittaa toimivia ohjelmistoja nopeasti asiakkaille, jotka voivat tarvittaessa ehdottaa uusien tai muuttuneiden vaatimuksien uudistamista järjestelmän myöhemmissä vaiheissa. Menetelmät pyrkivät vähentämään prosessin byrokratiaa välttämällä työtä, joka ei määrittelyn hetkellä nähdä tarpeelliseksi kehitettävissä olevien ominaisuuksien valmistamiseen, ja poistamalla turhaa dokumentaatiota sekä ominaisuuksia, joita ei todennäköisesti koskaan tulla käyttämään. Ketterien menetelmien perustan luo Agile Manifesto, joka määrittelee neljä arvoa sekä kaksitoista käytäntöä, jota kaikki menetelmät noudattavat. (Sommerville 2011, 58–59). Nämä neljä arvoa ovat (Agile Manifesto 2001):

- **Yksilöitä ja vuorovaikutusta** enemmän kuin prosesseja ja työkaluja.
- **Toimivaa sovellusta** enemmän kuin kokonaisvaltaista dokumentaatiota.
- **Asiakasyhteistyötä** enemmän kuin sopimusneuvottelua.
- **Muutoksiin reagoimista** enemmän kuin suunnitelman noudattamista.

Agile Manifeston (2001) määrittelemät periaatteita, joita ketterien menetelmien tulisi noudattaa:

1. Korkein prioriteetti on asiakastyytyväisyyden takaaminen aikaisilla ja jatkuvilla ohjelmistotoimituksilla.
2. Muuttuvat vaatimukset hyväksytään, jopa kehityksen lopussa. Ketterien prosessien mukautuvuudella voi asiakas saavuttaa kilpailuedun.
3. Toimivaa ohjelmistoa toimitetaan säännöllisesti, kahden viikon tai kahden kuukauden välein.
4. Liiketoimintahenkilöstön täytyy työskennellä ohjelmistokehittäjien kanssa päivittäin projektin aikana.
5. Ketterissä menetelmissä rakennetaan projektin motivoituneiden yksilöiden ympärille, joille annetaan tarvittava työympäristö ja tuki.
6. Tehokkain tapa välittää tietoa on kasvokkain toteutettava kommunikaatio.
7. Toimiva ohjelmisto on tärkein edistyksen mitta.
8. Ketterät prosessit edistävät kestävää kehitystä. Sponsorit, kehittäjät ja käyttäjien pitää pystyä ylläpitämään tasaista työtahtia.
9. Jatkuva huomio teknilliseen erinomaisuuteen ja hyvään suunnitteluun parantaa ketteryyttä.
10. Yksinkertaisuus on välttämätöntä.
11. Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoiduilla tiimeillä.
12. Säännöllisin väliajoin tiimi arvioi, miten se pystyy toimimaan tehokkaammin. Arvioinnin perusteella toimintaa muutetaan asianmukaisesti.

Ketterien menetelmien tyypilliset ominaisuudet eivät ole kuitenkaan täysin uudenlaisia vaan samantapaisia käytäntöjä on jo käytetty perinteisissä menetelmissä. Menetelmien ominaisuudet perustuvatkin aikaisempien ohjelmistokehitys menetelmien käytöstä saatuihin kokemuksiin, jotka todettiin toimiviksi. (Tsui 2013, 84; Larman 2003, luku 2.) Ketteriä menetelmiä on monenlaisia, jotka eroavat toisistaan, mutta kaikkien lähtökohtaisena tarkoituksena on pystyä mukautumaan nopeasti ohjelmistokehityksen muuttuviin vaatimuksiin. (Ingeno 2018).

3.1 Scrum

Scrumin kehittivät Ken Schwaber ja Jeff Sutherland 90-luvun alkupuolella helpottamaan monimutkaisten ohjelmistokehitysprojektien hallintaa. Scrum perustuu arvoihin, periaatteisiin ja käytäntöihin, jotka luovat menetelmän perustan. Scrum on tarkoituksella jätetty kes-

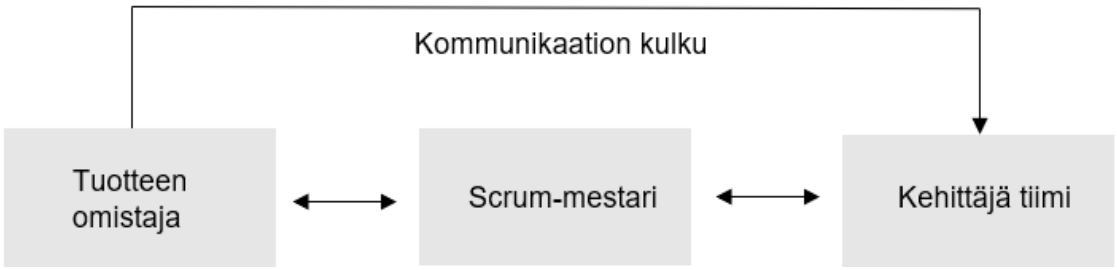
keneräiseksi, joka mahdollistaa projektien yksilöllisiin tarpeisiin tarvittavia teknillisiä lähestymistapojen valitsemisen. Menetelmä ei siis ota kantaa teknologioista, joita tulisi käyttää vaan sen ideana on antaa tarvittavat työkalut sopeutumaan muuttuviin olosuhteisiin, joilla pyritään varmistumaan, että valmistettava tuote valmistuu ajallaan sekä budjetin rajoissa asiakkaalle. (Rubin 2012, 2; Schwaber & Sutherland, 2020.)

Scrum perustuu empiiriseen ja Lean-ajattelutapaan eli ideaan, jossa tieto hankitaan kokemusten sekä havaintojen perusteella tehdyistä päätöksistä. Menetelmä pyrkii vähentämään ylimääräisiä kustannuksia keskittymällä vain tarpeellisiin asioihin. Scrum rakentuu kolmeen empiirisen tukipilariin, jotka ovat läpinäkyvyys, tarkastelu ja sopeutuminen. Tämä tarkoittaa sitä, että Scrumin prosessien täytyy olla näkyvissä kaikille tekijöille sekä lopputuotoksen vastaanottajille. Tehdyille tuotoksille on suorettava tarkastuksia usein ja huolellisesti, jotta mahdollisia ongelmia voidaan havaita. Lisäksi tuotteen sekä sovellettujen prosessien täytyy säilyttää mukautuvaisuus tavoitteiden saavuttamista varten. (Schwaber & Sutherland, 2020.)

Scrumin idea on, että pieni tiimi keskittyy yhden tavoitteen ympärille ja kokoaa kehityksen sprintteihin. Sprintit ovat ennalta määritettyjä ajanjaksoja; maksimissaan kuukauden pituisia, joissa tapahtuu kaikki tarvittava työ kyseisen Sprintin tavoitteiden saavuttamiseksi. Sprintti sisältää erilaisia tapahtumia, kuten Sprintin suunnittelu, päivittäinen Scrum, Sprintin tarkastelu ja Sprintin retrospektiivi, joilla pyritään auttamaan projektin hallinnointia ja ennustettavuutta. (Schwaber & Sutherland, 2020.)

Scrum tiimi koostu pienestä joukosta ihmisiä, tyypillisesti kymmenen tai vähemmän, joka on itseorganisoituva ja monitaitoinen. Tiimissä ei ole alaryhmiä tai hierarkioita. Se toimii yhtenäisenä yksikkönä, joka keskittyy yhdessä päätettyjen tavoitteiden toteuttamisesta. Scrum tiimiin kuuluu kolme henkilöroolia; Tuotteen omistaja, kehittäjätiimi sekä Scrummestari. (Schwaber & Sutherland, 2020.)

Tuotteen omistaja on yksi henkilö, joka on vastuussa kehittäjätiimin tulosten maksimoimisesta. Tuotteen omistaja voi olla vastuussa yhdestä tai useammasta tiimistä samanaikaisesti. Henkilön tavoitteina on selvittää asiakkailta saatujen vaatimusten eli toivottujen ominaisuuksien priorisointi siten, että asiakkaalle taataan maksimaalinen arvo. Tämä tapahtuu tuotteen kehitysjonon avulla. Tuotteen kehitysjono on järjestetty luettelo tai lista, jossa säilytetään kaikki kehitettävissä olevan tuotteen valmistukseen tarvittavat tehtävät. Tästä listasta jaetaan tehtäviä kehittäjätiimin työstettäväksi. Tuotteen omistaja on vastuussa tästä listasta ja kehityksen suunnasta, eli siitä mitä ollaan tekemässä ja missä järjestyksessä. (Schwaber & Sutherland, 2020.)



Kuva 5. Scrum tiimi havainnollistettu. (Mukaillen Rubin 2012, luku 2)

Kehittäjätiimi on vastuussa toiminnallisuuden kehittämisestä. Tiimi on itsehallinnoiva, itseorganisoituvia ja monitaitoinen. Tiimin vastuulla on selvittää, kuinka kehitysjonon tehtävistä voidaan koota toiminnallisuuksien joukko, joita työstetään Sprinteissä. Tiimin jäsenet ovat yhdessä vastuussa iteraatioiden onnistumisesta. (Schwaber & Sutherland, 2020.)

Scrum-mestari vastaa Scrumin prosesseista, opettamalla Scrumin sääntöjä ja käytäntöjä kaikille projektissa mukana oleville sidosryhmille ja varmistamalla, että tiimin tehokkuus säilyy niitä noudattamalla. Scrum-mestari on myös vastuussa mahdollisten esteiden poistamisesta kehitystiimien sekä tuotteen omistajan ja asiakkaiden välillä. (Schwaber & Sutherland, 2020.)

Seuraavaksi kuvataan Scrumin elinkaarta sekä vaiheita Rubinin (2012, luku 2) mukaan. Scrumissa työ tapahtuu iteraatioissa, jotka ovat yleensä kuukauden mittaisia. Näitä iteraatioita kutsutaan sprinteiksi. Sprintit ovat ajanjaksoja, joissa on alkamisaika sekä päättymisaika. Sprintin tavoitteena on valmistaa, jotain hyödyllistä arvoa asiakkaalle tai käyttäjälle. Sprintit seuraavat aina toistaan eli kun aikaisempi loppuu, uusi alkaa.

Ennen projektin aloittamista asiakkaalla on visio rakennettavasta tuotteesta. Tuotteen omistaja, Scrum-tiimi ja sidosryhmät jaottelevat rakennettavan tuotteen vaatimukset ominaisuuksiin, jotka kirjataan listaan nimeltä tuotteen kehitysjono. Tämä lista sisältää alustavasti kaikki ominaisuudet, jotka tarvitaan tuotteen kehittämiseen. Projektin edetessä siihen voidaan myös lisätä uusia ominaisuuksia tai muuttaa jo aikaisempia ominaisuuksia.

Tuotteen omistaja määrittelee sidosryhmien kanssa kehitysjonon priorisoinnin. Tarkoituksena on selvittää projektin etenemisen kannalta tärkeimmät ominaisuudet ensimmäiseksi toteutettavaksi. Tuotteen kehitysjono edustaa useiden viikkojen tai kuukausien työtä, joka on työmäärältään suurempi, mitä yhden sprintin aika voidaan toteuttaa. Joten tuotteen omistaja, kehittäjätiimi sekä Scrum-mestari toteuttavat sprintin suunnittelua, jossa selvitetään sprintille valittavat ominaisuudet.

Sprintti alkaa sen suunnittelulla, jossa tuotteen omistaja sekä kehittäjätiimi sopivat yhdessä sprintin tavoitteista. Kehittäjätiimi katselmoi tuotteen kehitysjonoa ja selvittää, mitä

tehtäviä valitaan tulevalle sprintille, jotta kyseisen sprintin tavoitteet toteutuvat. Scrum-tiimin jäsenet luovat sprintin suunnittelun aikana toisen tehtävien kehitysjonon, jota kutsutaan sprintin kehitysjonoksi. Tämä kehitysjono kuvaa yksityiskohtaisemmin, kuinka tiimi aikoo toteuttaa valittuja tehtäviä kyseisen sprintin aikana.

Kun Scrum-tiimi on saanut sprintin suunnittelun päätökseen ja sopinut seuraavan sprintin sisällöstä, aloittaa kehittäjätiimi toteuttamaan valittuja tehtäviä Scrum-mestarin opastuksella. Kukaan ei kerro kehitystiimille missä järjestyksessä tehtäviä pitäisi toteutetaan. Sen sijaan tiimin jäsenet määrittelevät työjärjestyksen heidän mielestään parhaalla tavalla. Joka päivä sprintin suorittamisen aikana tiimin jäsenet pyrkivät hallinnoimaan työnkulkua päivittäisellä scrumilla.

Päivittäisessä scrumissa tiimin jäsenet vastailevat vuorotellen kysymyksiin, kuten: “Mitä olen saavuttanut edellisen palaverin jälkeen?”, ”Mitä aion tehdä seuraavaan palaveriin mennessä?” ja ”Mitä esteitä tai haittoja olen kohdannut omassa työskentelyssäni”. Vastaamalla näihin kysymyksiin kaikki ymmärtävät paremmin projektin kulun kokonaiskuvaa ja siitä, miten projekti etenee sprintin tavoitteisiin verraten.

Sprintin lopussa suoritetaan kaksi tarkastusvaihetta, ennen sen päättymistä. Näistä ensimmäinen on Sprintin katselmointi. Tämän toiminnan tavoitteena on tarkastaa juuri valmistuneiden ominaisuuksien korrelaatio koko tuotettavan ohjelmiston ominaisuuksiin. Tämä tapahtuu sidosryhmien välisen kommunikoinnin avulla. Onnistunut katselmointi helpottaa tiimin jäseniä tarkastelemaan tuotoksiaan ja muuttamaan niitä mahdollisesti tulevissa sprinteissä, sillä Scrumissa ei sallita ominaisuuksien muokkaamista, jos ne eivät vastaa kyseisen sprintin tavoitteita.

Toinen tarkastusvaihe on Sprintin retrospektiivi, jossa tarkastetaan ja muokataan käytettäviä prosesseja. Retrospektiivin aikana kehitystiimi, Scrum-mestari sekä tuotteen omistaja kokoontuvat keskustelemaan siitä, mikä toimii ja ei toimi tiimin toiminnassa. Retrospektiivin lopussa tunnistetaan ja sitoudutaan prosessimuutoksiin, joita toteutetaan seuraavassa sprintissä. Kun sprintin retrospektiivi on suoritettu, koko kehityssykli toistetaan uudelleen; alkaen sprintin suunnittelusta, jossa määritetään uudelleen sprintille valittavat tehtävät kehitysjonosta. Sprinttejä toistetaan niin kauan, että rakennettava ohjelmisto on valmis julkaistavaksi asiakkaalle.

Scrumin kehittäjät toteavat, että Scrum on yksinkertaisesti ymmärrettävissä, mutta vaikea hallita. Osa syyksi muodostuvat monet prosessit, käytännöt ja tekniikat, jotka ovat Scrumin onnistumisen kannalta välttämättömiä. Bob Martin, joka tunnetaan yhtenä ketterien menetelmien pioneerina, nostaa esiin muutamia oleellisia ongelmia Scrumiin liittyen:

Koska Scrum ei ota kantaa ohjelmistokehityksen teknillisiin lähestymistapoihin, on riskinä, että kehitettävissä olevan tuotteen tai sovelluksen laatuun ei kiinnitetä tarpeeksi huomiota, joka saattaa vaikuttaa kehityksen ketteryyteen. Scrum ei ota myöskään kantaa testaukseen, jolla pystyttäisiin vaikuttamaan laadun säilyvyyteen. Martinin mielestä myös Scrummestarin rooli voi olla ongelmallinen. Hänen mielestään rooli voi altistaa henkilön toimimaan liian kontrolloivana tekijänä projekteissa. Martinin mielestä on myös ongelmallista, että Scrum olettaa tiimien olevan täysin itseorganisoituvia. Itseorganisoituminen voi toimia joissakin konteksteissa tai joillakin tiimeillä, mutta jos tuotetta on tekemässä useita tiimejä, voidaan ajautua ongelmiin, jos luotetaan pelkästään itseorganisoitumiseen. (Martin 2010.)

3.2 Extreme Programming (XP)

Extreme Programming (XP) on ketterä ohjelmistokehitys menetelmä, jonka Kent Beck kehitti ratkaisemaan ohjelmistokehityksen rajoitteita hallita epämääräisiä tai nopeasti muuttuvia vaatimuksia. Se perustuu arvokokoelmaan, toisiaan täydentävien periaatteiden joukkoon sekä käytäntöjen kokonaisuuteen. XP erottuu muista ketteristä menetelmistä sen lyhyiden kehitysjaksojen, tiiviin yhteistyön, jatkuvan palautteen sekä varhaisen testauksen takia (Beck & Andres 2004, luku 1). Beckin (1999) mukaan, XP:n nimi pohjautuu ideasta, jossa tavalliset ohjelmistokehityksessä käytettävät käytännöt ja käsitteet, kuten testaus, pariohjelmointi ja ohjelmiston elinkaarisykli viedään äärimmäisyyksiin.

XP pohjautuu viiteen arvoon, joiden kautta käytäntöjen ymmärtäminen on helpompaa. Näitä arvoja ovat kommunikointi, yksinkertaisuus, palaute, kunnioitus ja rohkeus. (Beck & Andres 2004, luku 1.4). Arvot peilaat ketterien menetelmien arvoja hyvin paljon, mutta niitä on hyvä katselmoida XP:n näkökulmasta.

XP kannustaa kasvotusten tehtävää kommunikaatiota kehittäjien sekä kehittäjien ja asiakkaan välillä, sillä se on tärkeää tiimityön ja joustavuuden luomisessa. Tämä antaa tiimille mahdollisuuden vastata muuttuviin vaatimuksiin, olla selvillä tavoitteista, projektin tilasta sekä prioriteeteista. Se tukee myös ketteryyttä levittämällä tietoa kehitystiimissä, jolloin vältetään tarve ylläpitää kirjallista dokumentaatiota. (Loftus & Ratcliffe. 2005, 312; Wells 2009; Warden 2003, luku 3.)

XP kannustaa kehittäjiä etsimään yksinkertaisinta tapaa tunnetun ongelman ratkaisemiseen, asiakkaan sijoituksen arvon maksimoimiseksi. Pienten yksinkertaisten askeleiden avulla lievennetään epäonnistumisia niiden tapahtuessa. Se tarkoittaa vain valittujen ongelmien ratkaisemista. XP:ssä uskotaan, että kehittäjien ei kannata lisätä suunnitteluominaisuuksia, jotka saattavat ratkaista jonkin ennakoidun tulevaisuuden ongelmia, joka ei välttämättä toteudu. (Loftus & Ratcliffe. 2005, 311; Wells 2009; Warden 2003, luku 3.)

Kehittäjät esittelevät ohjelmistoaan aikaisin, joiden perusteella tehdään tarvittavia muutoksia. Tiimit pyrkivät tuottamaan niin paljon palautetta kuin pystyvät ja käsittelemään ne mahdollisimman nopeasti. Tiimit yrittävät lyhentämään palautesykliä minuutteihin tai tunteihin viikkojen tai kuukausien sijaan. Uskomus on, että mitä nopeammin tiedät, sitä nopeammin voit sopeutua. (Beck & Andres 2004, luku 1.)

Jokainen tiimin jäsen ansaitsee muiden kunnioituksen, sillä jokainen tuo arvoa. Jokaisen jäsenen työtä pitää arvostaa. Kehittäjät kunnioittavat sidosryhmien asiantuntemusta ja päinvastoin. (Beck & Andres 2004, luku 1.4). Rohkeudella tarkoitetaan hankalien päätösten tekemistä sekä vastuunottamista ongelmien ratkaisuun. Mikäli koodissa havaitaan puutteita, täytyy niihin puuttua. Asiakkaalle täytyy pystyä olemaan avoin vaikeissakin tilanteissa. (Warden 2003.)

XP:n käytännöt ovat menetelmiä, joita tiimit käyttävät päivittäisessä työskentelyssään. Käytännöt tukevat sekä tukeutuvat toisiinsa. Käytäntöjen tarkoituksena on tehdä työskentelystä helpompaa ja tehokkaampaa. Menetelmät ovat kuitenkin tilannekohtaisia, joten kaikkia ei kannata käyttää, jos ne eivät tilanteeseen sovi (Warden 2003, luku 2.) Beck (1999) toi esille kaksitoista käytäntöä, jotka Warden (2003) luokittelee koodiin, kehittäjiin ja liiketoimintaan liittyviin käytäntöihin. Nämä yhdessä rakentavat menetelmän pohjan. Seuraavaksi kuvataan tärkeimpiä Wardenin (2003, luku 2) määrittämiä ydinkäytäntöä yksityiskohtaisemmin.

Toteuta koodi ja muotoilu yksinkertaisesti. Ratkaise asiakkaan nykyinen tarve. Älä arvaile tulevia tarpeita. Tee yksinkertaisin asia, joka voidaan toteuttaa. Yksinkertaisia malleja on helpompi ymmärtää ja selittää. Yksinkertaista koodia on helpompi testata, ylläpitää ja muuttaa.

Refaktoroi koodisi säännöllisesti. Kun olet suorittanut testauksen, muokkaa toteutettua koodia uudelleen. Jaa pitkät menetelmät ja toiminnot pienemmiksi. Hyödynnä mahdollisuutta yksinkertaistaa koodia ja sen suunnittelua. Jätä koodi helpommin ymmärrettäväksi ja muokattavaksi.

Kehitä koodausstandardeja, joiden avulla kehittäjät voivat kommunikoida. Koodi on ensisijainen viestinnän muoto projektissa. Parhaat koodausstandardit ovat ohjeita eivätkä käskyjä. Ne edustavat projektisi yhteisiä arvoja.

Kehitä yhteinen sanasto. Sidosryhmien teknillinen ymmärrys kehitettävästä ohjelmistosta ei välttämättä ole samalla tasolla kuin sen kehittäjillä. Joten on tärkeää määritellä yhdessä sovittu sanasto, joka mahdollistaa kaikkien sidosryhmien ymmärtävän rakennettavan tuotoksen sisällöstä.

Omaksu testilähtöinen kehitys. Testivetoinen ohjelmointi on XP:lle ominainen ohjelmistokehityksen lähestymistapa, jossa ennen toiminnallisuuden kehittämistä sille rakennetaan testit, jotka varmistavat, että kyseinen toiminnallisuus vastaa oikeanlaisia vaatimuksia. Tällöin vältytään turhien ominaisuuksien kehittämiseltä ja bugien syntymiseltä.

Pariohjelmointi. Kaikki ohjelmointi XP:ssä toteutetaan pariohjelmointina. Tämä tarkoittaa käytännössä sitä, että kaksi kehittäjää työskentelee yhdellä tietokoneella, jossa toinen kirjoittaa koodia ja toinen huolehtii mahdollisten virheiden ilmoittamisesta. XP:n mukaan pari ohjelmointi vähentää testien, refaktoroinnin ja muiden tehtävien laiminlyöntiä. Lisäksi kahden henkilön on helpompaa huomata virheitä, joka vähentää niiden syntyä sekä lisää kommunikaatiota, joka johtaa laadukkaamman koodiin toteuttamiseen.

Hyväksy kollektiivinen koodin omistus, jossa koko kooditieto kuuluu koko XP-tiimille. Jokainen kehittäjä voi muuttaa mitä tahansa koodin osaa lisätäkseen toiminnallisuuksia tai korjatakseen vikoja.

Jatkuva integraatio on käytäntö, jossa valmistettu ominaisuus testataan välittömästi. Testien tarkoituksena on varmistaa, että kyseinen ominaisuus toimii ja on valmis lisättäväksi muiden ominaisuuksien kokonaisuuteen. Jatkuva integraatio säilyttää projektin kokonaiskuvan näkymän kaikille projektin jäsenille.

Lisää asiakas tiimiin. Asiakas tarjoaa liiketoiminnan näkökulman ohjelmiston todellisesta käyttäjästä. Säännöllinen, luotettava ja nopea viestintä sidosryhmien välillä lisää luottamusta, vähentää väärinkäsityksiä ja tuottaa halutut tulokset nopeammin. Asiakkaan tulee tehdä tiivistä yhteistyötä kehittäjien kanssa. Ihannetapauksessa asiakas työskentelee kehittäjien rinnalla. Mitä lähempänä asiakas on muuta tiimiä, sitä parempi.

Hyödynnä suunnittelupeliä. XP käyttää vaatimusten priorisointiin suunnittelu peliä, jossa pyritään selvittämään rakennettavien ominaisuuksien toteutusjärjestys. Suunnittelupelin tavoitteena on maksimoida tuotettujen ominaisuuksien arvo. XP jakaa suunnitteluvastuun kehittäjien sekä asiakkaan välillä. Asiakas tekee liiketoimintapäätöksiä päättäessään resurssien jakamisesta ja asettamalla ominaisuuksien prioriteetit. Asiakas pohdiskelee toteutettavaan järjestelmään verraten seuraavanlaisia kysymyksiä:

- Mitä pitäisi tehdä?
- Mikä on kunkin ominaisuuden arvo ja riski?
- Mitkä ominaisuudet ovat tärkeämpiä kuin muut?

Kehittäjät tekevät teknisiä päätöksiä, valitsevat tekniikoita ja toteutuksen yksityiskohtia. Kehittäjät pohdiskelevat seuraavanlaisia kysymyksiä:

- Kuinka ominaisuus tulisi toteuttaa?
- Millaisen teknisen riskin jokaisella ominaisuudella on?
- Kuinka kauan ominaisuuden käyttöönotto kestää?

Työskentele kestävällä nopeudella. Tarkoittaa työskentelyä silloin kun olet siihen motivoitunut henkisesti ja fyysisesti valmis. Käytännöllä huomioidaan, että vaikka vaikeissa olosuhteissa työskentelyä joutuu tekemään, ei siihen tilanteeseen kannata pyrkiä. Sillä energisenä ja motivoituneena pystytään tekemään laadukkaampaa jälkeä, kuin stressaantuneena ja väsyneenä. Tiimin jäsenten tulisi siis huomioida omaa toimintaansa ja suunnitella sitä niin, että kyseisiin tilanteisiin ei jouduttaisi.

Beckin (2004) julkaisemassa toisessa painoksessa tuodaan lisäksi esille XP:n kehitysprosessin aikana toteutettavia käytäntöjä:

Yhdessä istuminen. Tarkoituksena on pystyä olemaan tilassa, jossa koko tiimi pystyy ja mahtuu työskentelemään. Tällä tavoin pystytään maksimoimaan tiimin jäsenten välinen kommunikaatio, eikä aikaa mene turhiin siirtymisiin. Käytäntö olettaa, että mitä enemmän kasvokkaista vuorovaikutusta tiimin jäsenillä on, sitä humaanimpaa ja tehokkaampaa työskentely tulee olemaan.

Tarinat ovat lyhyitä kuvauksia ominaisuuksista, joita ohjelman tulisi tehdä. Ne perustuvat asiakkaan vaatimiin tuotteen toiminnallisuuksiin. Tyypillisesti tarinat kirjoitetaan fyysiselle paperilapulle, joista koostuu kokoelma toteutettavia tehtäviä kehittäjille.

Viikkosykli on ajanjakso, jossa kehitys tapahtuu. XP:lle on ominaista toteuttaa viikon mittaisia iteraatioita, jossa tiimi katselmoi kehityksen etenemistä ja kommunikoi asiakkaan kanssa, mitä tarinoita toteutetaan. Tavoitteena on saada toteutettua ja esitettyä asiakkaalle kokoelma toimivia ominaisuuksia, jotka voidaan liittää aikaisemmilla sykleillä tehtyihin tuotoksiin.

Neljännessyklin tarkoituksena on katselmoida koko projektin etenemistä ja mahdollisien kehitysprosessiin liittyvien esteiden poistamisesta, joita viikkosykleissä on huomattu. Neljännessykleissä tiimi reflektoi myös omaa tekemistään ja miten se kohtaa projektin tavoitteita. Asiakkaan tarkoituksena on pitää huoli selkeiden tavoitteiden määrittämisestä, jotta kehittäjät voivat keskittyä vain olennaisten ominaisuuksien rakentamiseen.

XP:ssä henkilöroolit eivät ole ennalta määritettyjä, vaan roolitus tapahtuu osaamisen kannalta, joka tarkoittaa sitä, että ohjelmoijasta voi tulla esimerkiksi tuotteen omistaja tai toimia monessa roolissa samanaikaisesti, jos tarve niin vaatii. On kuitenkin todettu, että selkeät roolit auttavat tiimejä aluksi toimimaan ryhmässä paremmin. (Beck 2004, luku 10.) Seuraavaksi on kuvattu Wardenin (2013) mukaan tärkeimpiä henkilörooleja:

Ohjelmoijat ovat olleet XP:n alusta asti tärkeässä roolissa, sillä he ovat niitä, jotka rakentavat projektille olennaisia toiminnallisuuksia. Tämä kyseinen rooli on jo itsessään melko yksiselitteinen. XP tiimissä ohjelmoijat arvioivat tarinoita ja jakavat niitä pienempiin tehtäviin, kirjoittavat koodia ominaisuuksien toteuttamiseksi, automatisoivat kehitysprosesseja ja kirjoittavat testejä. Ohjelmoijat työskentelevät tiiviisti toistensa kanssa pariohjelmoiden.

Valmentaja ohjaa ja mentoroi tiimiä. Hän ajaa XP:n arvojen ja käytäntöjen oikeaoppista käyttöä. Valmentaja on yleensä henkilö, jolla on kokemusta XP-projekteista. Hän toimii myös tiimin ja muiden sidosryhmien välikätenä.

Mittaaja on vastuussa projektin aikataulusta. Hän pyrkii säilyttämään tiimin nopeuden ja torjumaan mahdollisia haittoja, jotka tähän vaikuttavat. Mittaaja huolehtii ajallisten muutosten, ylitöiden ja testeistä kerätyn datan hallinnoinnista.

Asiakas vie projektia enteenpäin. Hän määrittelee projektin tavoitteet ja yrityslähtöiset päätökset. Asiakas tekee tiivistä yhteistyötä kehittäjien kanssa, tekemällä käyttäjätarinoita ja päättämällä aikataulutuksesta. Asiakas toimii myös loppukäyttäjän asemassa määrittelemässä tarvittavia ominaisuuksia, jotka projektin tuotosten täytyy täyttää.

XP:n elinkaari muodostuu kuudesta vaiheesta. Vaiheet ovat tutkimusvaihe, suunnitteluvaihe, iteraation julkaisuvaihe, tuotteistamisvaihe, ylläpitovaihe ja lopetusvaihe. Seuraavaksi kuvataan vaiheita tarkemmin Beckin (1999) mukaan:

Tutkimusvaiheessa asiakas selvittää asioita, jotka hän haluaa ensimmäiseen julkaisuun vietäväksi. Vaatimukset kirjataan tarinakorteille, jotka kehittäjät vastaanottavat. Kehittäjätiimi tutustuu kaikkiin teknologioihin sekä käytäntöihin, joita mahdollisesti halutaan tai tar-

vitaan projektin aikana käytettäväksi. Mikäli kaikki on jo entuudestaan tuttua kehittäjätiimille voi tutkimusvaihe kestää muutaman viikon, mutta on mahdollista, että muutama kuukausi tarvitaan uusien teknologioiden ja käytäntöjen opetteluun.

Suunnitteluvaiheessa asiakas sekä kehittäjätiimi sopivat yhdessä ensimmäiseen iteraation sisällöstä. Asiakkaan määrittämät tarinat priorisoidaan ja niille luodaan aikataulu, jolloin niiden toteuttaminen tulisi olla valmis. Sidosryhmät myös suunnittelevat ensimmäisen julkaisun ajankohtaa, joka pitäisi tapahtua 2–6 kuukauden kuluttua. Itsessään suunnitteluvaihe pitäisi kestää vain muutaman päivän, mikäli tutkimusvaihe on suoritettu huolellisesti.

Iteraation julkaisuvaihe. Tässä vaiheessa suunnitteluiden aikana valittujen toiminnallisuuksien toteuttaminen jaetaan yhdestä neljään viikkoon kestäviin iteraatioihin, jossa projektin kehittämisprosessi toteutuu. Ensimmäisellä iteraatiolla luodaan koko järjestelmä arkkitehtuuri. Seuraaville iteraatioille asiakas valitsee toiminnallisuuksia sen perusteellä, mikä on järkevintä ja parasta rakentaa seuraavaksi. Jokaisen iteraation lopussa täytyy kehittäjien rakentamien toiminnallisuuksien läpäistä niille tarkoitetut testit, jonka jälkeen ne voidaan luovuttaa asiakkaalle. Iteraatioiden lopuksi on siis tarkoituksena luovuttaa asiakkaalle aina pala toimivaa kokonaisuutta.

Tuotteistamisvaiheessa toteutetaan lisää testejä ja parannetaan tarvittaessa järjestelmän suorituskykyä. Vaiheessa tarkastetaan, että toiminnallisuudet ovat valmiita julkaistavaksi. Tuotteistamisvaiheessa voidaan huomata myös mahdollisia lisäominaisuuksia järjestelmään, joita kyseessä olevaan julkaisuun ei pystytä toteuttamaan. Nämä dokumentoidaan ja toteutetaan mahdollisesti tulevissa iteraatioissa tai ylläpitovaiheen jälkeen.

Ylläpitovaiheessa asiakkaalla on käytössä kehitettyjä toiminnallisuuksia, joiden toimintaa valvotaan. Ylläpitovaihe tarvitsee huolellisuutta, sillä käytössä olevien järjestelmien täytyy toimia sekä huomioida, että tulevien julkaisujen toiminnallisuuden tulevat toimivaan jo julkaistujen toiminnallisuuksien kanssa. Kehittämistahti myös hidastuu ylläpitovaiheessa, koska kehitettäviä ominaisuuksia ei tässä vaiheessa lisätä. Vaiheeseen täytyy silti säilyttää työntekijöitä, jotta asiakkaalta saadut palautteet ja ongelmat pystytään hoitamaan.

Lopetusvaihe tapahtuu silloin kun, asiakkaalla ei enää ole ominaisuuksia lisättäväksi järjestelmään. Lopetus tapahtuu yhteisenä päätöksenä, mutta loppukädessä asiakkaan puolesta. Mikäli järjestelmään ei olla tyytyväisiä tai siinä havaitaan puutteita, ei projektia voida lopettaa. Onnistuneen projektin jälkeen on tyypillistä, että tiimi reflektoi omaa tekemistään ja miettii, miten seuraavalla kerralla samankaltaista projektia kannattaisi toteuttaa.

XP projekteja on vaikea toteuttaa, jos asiakas ei pysty toimimaan tiiviisti menetelmän käyttäjien kanssa. XP:tä on vaikea soveltaa projekteihin, jos kasvokkain tapahtuvaa vuorovaikutusta ei voida toteuttaa. XP keskittyy vahvasti koodin tuottamiseen, joka saattaa vaikuttaa ohjelmistotuotteen muotoilun laiminlyöntiin. Rakennettavan tuotteen ulkonäkö lopulta myy sovelluksen, joten asiakas voi olla tyytymätön lopputuotteeseen, ellei muotoilu ole riittävän hyvä. XP:ssä iteraatioiden pituudet ovat vain viikon pituisia, joten on mahdollista, että tiukat aikataulut lisäät ohjelmistokehittäjien stressiä. Mikäli kehittäjillä on korkea stressitaso tehtäviä suorittaessa, on todennäköistä, että virheitä syntyy koodauksen aikana. (Panayotova 2018.)

4 Ketterien menetelmien vertailu

Vaikka Scrum ja XP pohjautuvat ketteristä menetelmistä, on niillä silti eroavaisuuksia. Taulukko 1 listaa tärkeimmät eroavaisuudet, joiden avulla pystytään selkeämmin erottelemaan menetelmät toisistaan. Taulukko perustuu teoriaosuudessa kerätyistä havainnoista.

Taulukko 1. Scrumin ja XP:n vertailu.

Vertailun kohde	Scrum	Extreme Programming
Kehitystyyli	Inkrementaalinen ja iteratiivinen	Inkrementaalinen ja iteratiivinen
Kehitysprosessin vaiheet	Sprintin suunnittelu, Sprintin toteutus, Sprintin tarkastelu ja Sprintin retrospektiivi	Tutkimusvaihe, Suunnitteluvaihe, Iteraation julkaisuvaihe, Tuotteistamisvaihe, Ylläpitovaihe ja Lopetusvaihe
Iteraation pituus	4 viikkoa	1–4 viikkoa
Roolit	Kehittäjätiimi, Tuotteen omistaja, Scrum-mestari	Ohjelmoija, asiakas, valmentaja, mittaaja
Tiimien lukumäärä	1–4 tai enemmän	1 tiimi projektia kohden
Yhden tiimin koko	6–10 henkilö	3–16 henkilöä
Fyysinen yhteistyö	Ei määritetty	Kyllä
Fokus	Projektinhallinta ja tuottavuus	Tuotekehitys
Teknillinen lähestymistapa	Ei määritetty	Testaa ensin lähestymistapa, jossa apuna käytetään pariohjelmointia, yksinkertaista muotoilua sekä koodin refaktorointia.

Skaalautuvuus	Hyvä	Huono
Dokumentaatio	Vähäistä	Vähäistä
Työn priorisointi	Scrum-mestari kysyy Tuotteen omistajalta työjärjestyksestä, mutta tiimi valitsee itse	Asiakas päättää järjestyksen
Muutosten salliminen iteraatioissa	Ei sallittu	Kyllä
Testaus	Ei määritetty	Kyllä
Laadunvalvonta	Ei määritetty	Jatkuvilla testauksilla
Projektin koko	Kaikki	Pieni

Tarkastelun perusteella voidaan huomata, että vaikka menetelmät pohjautuvat ketterien menetelmien ideologiasta, ne eroavat huomattavasti toisistaan. Scrum pyrkii hallinnoimaan yrityksen projektinhallintaa ja sitä kautta parantamaan tuotettavuutta, kun taas XP keskittyy enemmänkin teknillisten käytäntöjen kautta toteuttamaan konkreettisia tuotoksia asiakkaalle mahdollisimman nopeasti. XP käyttää testivetoista lähestymistapaa ohjelmointiin, kun taas Scrumissa teknologioihin liittyvien käytäntöjen valitseminen on menetelmän käyttäjien vastuulla. Scrum toteuttaa iteraatioita sprinteissä ja XP:ssä iteraatiota kutsutaan sykleiksi. Molempien menetelmien iteraatioiden tarkoituksena on pyrkiä lisäämään projektin kokonaisuuteen lisää toiminnallisuuksia. Iteraatioiden pituudet puolestaan eroavat. Scrumia voidaan toteuttaa vaihtelevissa pituuksissa, mutta yleistä on toimia kuukauden mittaisissa sykleissä ja XP:ssä toteutetaan yhden tai kahden viikon syklejä. Scrum on hyvin tiukka muutosten tekemisen sallimisesta näissä iteraatioissa, kun taas XP:ssä tämä on sallittua. Tiimit ovat pitkälti samankokoisia, mutta Scrum mahdollistaa useamman tiimin

työskentelyn yhdessä projektissa. Näin ei toimita XP:ssä, jossa vain yksi tiimi toteuttaa yhtä projektia.

Kasvotusten tehtävä yhteistyö on XP:lle ominaista ja ilman tätä mahdollisuutta projektien toteuttaminen voi olla hankalaa, sillä jatkuva ja rikas kommunikaatio on yksi menetelmän perusolettamuksista. Se vaatii hyvin paljon sidosryhmien jatkuvaa sitoutumista kehitysprosessiin. Scrum ei painota fyysisen kanssakäymisen tärkeyttä. Roolitus tapahtuu menetelmissä myös hieman eri tavalla. Scrumissa henkilöroolit ovat ennalta määrättyjä, kun taas XP:ssä henkilöroolien rajat ovat häilyvämpiä. XP:lle on yleistä, että roolit valitaan osaamisen kautta sekä yhden henkilön toimiminen useammassa roolissa samanaikaisesti sallitaan. XP on hyvin tarkka projektin koosta, jossa menetelmää käytetään. Menetelmä ei sovellu ison ryhmän käytettäväksi, sillä XP vaatii tiukkaa yhteistyötä sekä rikasta ja nopeaa kommunikointia tiimin sisällä. Suuret tiimit eivät mahdollista XP:tä toimimaan tarkoitetulla tavalla. Scrumia taas voidaan toteuttaa isoissa ryhmissä, koska se lähtökohtaisesti keskittyy projektinhallintaan.

Scrum on vertailussa olevista menetelmistä yleiskäyttöisempi. Sitä voidaan käyttää niin pienissä kuin suurissa projekteissa. Menetelmän skaalautuvuus mahdollistaa soveltuvuuden lukuisiin erityyppisiin projekteihin. Se soveltuu liiketoiminnan kannalta kriittisiin projekteihin, joissa on tiukka aikataulu ja monimutkaiset muuttuvat vaatimukset. Menetelmä onkin hyvin suosittu nykypäivänä, sillä vuosittaisessa “Annual State of Agile Report“ (2020) kyselyssä, joka on pitkäkestoisin ketteriin menetelmiin kohdentuva kysely raportoi, että yli puolet vastanneista yrityksistä kertoi käyttävänsä Scrumia, kun taas XP:tä käyttivät vain 1 % vastanneista. XP puolestaan soveltuu pieniin tuotekehitykseen tarkoitettuihin projekteihin, joissa asiakkaalla ei välttämättä ole tietoa järjestelmän lopullisesta käyttötarkoituksesta. Menetelmän käytäntöjen tarkoituksena on pyrkiä tiimin ja asiakkaan välisen vuorovaikutuksen kautta löytämään asiakkaan todellinen tarve. XP:n käyttö ohjelmistokehityksessä on jäänyt vähäisemmäksi. Suurimmiksi syiksi voidaan nostaa yksilöllisempi käyttötarkoitus, riippuvaisuus sidosryhmien panoksesta kehityksen osana olemisessa sekä samassa sijainnissa tapahtuvan kehittämisen pakko.

5 Pohdinta

Tämän opinnäytetyön tarkoituksena oli tutkia kahta ketterää ohjelmistokehitys menetelmää sekä verrata niitä keskenään. Valitut menetelmät olivat Scrum sekä Extreme Programming (XP). Opinnäytetyön päätavoitteena oli saada selkeyttä menetelmien toimintatavoista sekä käytännöistä, jotta pystytään selvittämään menetelmien käyttösoveltuvuus ohjelmistoprojekteissa. Tutkimuksen aikana kerättyjen tietojen avulla saatiin selvitettyä vastaukset tutkimuskysymyksiin.

- Mistä ja miksi ketterät menetelmät ovat syntyneet?
- Miten Scrum ja Extreme Programming (XP) eroavat toisistaan?

Ensimmäiseen tutkimuskysymykseen lähdettiin hakemaan vastausta perehtymällä perinteisiin menetelmiin. Lähdeaineiston perusteella huomattiin, että ketterät menetelmät perustuvat perinteisten menetelmien heikkouteen vastata pienten sekä keskisuurten ohjelmistoprojektien nopeasti muuttuviin vaatimuksiin. Ketterät menetelmät lähestyvät ohjelmistokehitystä ihmislähtöisemmästä näkökulmasta. Ne pyrkivät ratkaisemaan perinteisten menetelmien heikkouksia yksinkertaistamalla sekä poistamalla turhia prosesseja ja pilkkomalla kehitystyötä pienempiin ajanjaksoihin, joissa asiakas tuodaan osaksi kehitysprosessia. Tämä tiivis yhteistyö mahdollistaa ongelmien ratkaisuun niiden sattuessa sekä ohjelmiston toimittamisen asiakkaalle paljon nopeammin perinteisiin menetelmiin verrattuna.

Toiseen tutkimuskysymykseen vastaaminen aloitettiin perehtymällä molempiin ketteriin menetelmiin. Perehdyttiin molempien menetelmien elinkaareen sekä käytäntöihin, joiden avulla pystyttiin havaitsemaan eroavaisuuksia. Nämä eroavaisuudet ovat havainnollistettu Taulukossa 1. Huomattiin, että Scrum on vertailussa olevista menetelmistä yleiskäyttöisempi, sillä se perustaa toimintansa projektinhallintaan. Se jättää tarkoituksella määrittelemättömiksi toimintamalleja, joka mahdollistaa menetelmän käytön monissa erityyppisissä projekteissa. Extreme Programming (XP) puolestaan keskittyy hyvin tarkasti helpottamaan tuotekehitykseen liittyviä projekteja. Sen käyttö on täten suppeampi Scrumiin verrattuna. Sen käyttöönotto vaatii tiivistä kasvokkain tehtävää yhteistyötä sekä kokemusta testilähtöisestä ohjelmistokehityksestä.

Ketterien menetelmien valinta projekteihin perustuu siihen, millaisia ominaisuuksia projektilla on. Lisäksi on tärkeää ymmärtää, että jokainen projekti on erilainen, eikä saman menetelmän käyttö välttämättä sovellu seuraavaan. Tämän työn vertailun avustuksella pystytään havainnollistamaan paremmin molempien menetelmien soveltuvuutta erilaisiin ohjelmistokehitysprojekteihin.

Lähteet

Beck, K, Beedle, M, Bennekum, A, Cockburn, A, Cunningham, W, Fowler, M, Grenning, J, Highsmith, J, Hunt, A, Jeffries, R, Kern, J, Maric, B, Martin, R, Mellor, S, Schwaber, K, Sutherland, J, Thomas, D. 2001a. Manifesto for Agile Software Development. Luettavissa: https://agilemanifesto.org. Luettu: 1.6.2021.

Beck, K, Beedle, M, Bennekum, A, Cockburn, A, Cunningham, W, Fowler, M, Grenning, J, Highsmith, J, Hunt, A, Jeffries, R, Kern, J, Maric, B, Martin, R, Mellor, S, Schwaber, K, Sutherland, J, Thomas, D. 2001b. Principles behind the Agile Manifesto. Luettavissa: https://agilemanifesto.org/principles.html. Luettu: 1.6.2021.

Beck, K. 1999. Extreme Programming Explained. Luettavissa: https://learning.oreilly.com/library/view/extreme-programming-explained/0201616416/pr03.html. Luettu. 12.7.2021.

Beck, K., Andres, C.2004. Extreme Programming Explained: Embrace Change, Second Edition. Addison-Wesley Professional. Luettavissa: https://learning.oreilly.com/library/view/extreme-programming-explained/0321278658/. Luettu: 20.3.2021.

Bourque, P, Fairley, R. 2014. Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society. Luettavissa: https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf. Luettu: 20.7.2021.

Crookshanks, E. 2015. Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software. Apress. Luettavissa: https://learning.oreilly.com/library/view/practical-software-development/9781484206201/9781484206218_Ch04.xhtml#Sec1. Luettu: 3.5.2021.

Digital.ai. 2020. 14th Annual State of Agile Report. Luettavissa: https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494. Luettu: 18.7.2021.

Dooley, J. 2011. Software Development and Professional Practice. Apress. Luettavissa: https://learning.oreilly.com/library/view/software-development-and/9781430238010/Chapter02.html#ch2. Luettu.17.7.2021.

Dutt, S, Subramanian, C. 2015. Software Engineering. Pearson Education India. Luettavissa: https://learning.oreilly.com/library/view/software-engineering/9789332558298/xhtml/Chapter001.xhtml#h4-026. Luettu: 3.5.2021.

Fowler, M. 2005. The New Methodology. Luettavissa: https://martinfowler.com/articles/newMethodology.html. Luettu: 6.5.2021.

Fowler, M. 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. Luettavissa: https://learning.oreilly.com/library/view/refactoring-improving-the/9780134757681/preface.xhtml#ch00lev1sec1. Luettu. 20.7.2021.

IEEE. 1990. IEEE Standard Glossary of Software Engineering Terminology. Luettavissa: http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf. Luettu. 2.5.2021.

Ingeno, J. 2018. Software Architect's Handbook. Pact Publishing. Luettavissa: https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/29a600c0-227d-4a4f-b3a0-7c5b759aecf3.xhtml. Luettu. 6.5.2021.

Larman, C. 2003. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional. Luettavissa: https://learning.oreilly.com/library/view/agile-and-iterative/0131111558/. Luettu.6.5.2021.

Loubser, N. 2021.Software Engineering for Absolute Beginners. Apress. Luettavissa: https://learning.oreilly.com/library/view/Software+Engineering+for+Absolute+Beginners:+Your+Guide+to+Creating+Software+Products/9781484266229/html/501322_1_En_9_Chapter.xhtml#Sec2. Luettu: 22.3.2021.

Lynch, W. 2019. What is the Problems of Waterfall Model. Luettavissa: https://warren2lynch.medium.com/what-is-the-problems-of-waterfall-model-38de858f1058. Luettu: 5.5.2021.

Martin, B. 2010. Scrum/Agile Failings or the Theses of Uncle Bob Martin. Luettavissa: https://www.infoq.com/news/2010/02/scrum-failings/. Luettu: 9.7.2021.

Martinez, P. 2020. What is Evolutionary Prototype. Luettavissa: https://mockitt.wondershare.com/prototyping/evolutionary-prototyping.html. Luettu: 17.7.2021.

McConnel, S. 1996. Rapid Development: Taming Wild Software Schedules. Microsoft Press. Luettavissa: https://learning.oreilly.com/library/view/rapid-development-taming/9780735634725/. Luettu: 5.5.2021.

Panayotova, E. 2018. What are the Pros and Cons of Extreme Programming (XP). Luettavissa: https://simpleprogrammer.com/pros-cons-extreme-programming-xp/. Luettu: 16.7.2021.

Petersen, K, Wohlin, C, Baca, D. 2009. The Waterfall Model in Large-Scale Development. Luettavissa: https://www.researchgate.net/publication/30498645_The_Waterfall_Model_in_Large-Scale_Development. 3.5. 2021. Luettu: 3.5.2021.

Royce, W. 1970. Managing The Development of Large Software Systems. TRW. Luettavissa: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf. Luettu: 28.4.2021.

Rubin, K. 2012. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley Professional. Luettavissa: https://learning.oreilly.com/library/view/essential-scrum-a/9780321700407/ch02.html. Luettu. 6.7.2021.

Scacchi, W. 2001. Process Models in Software Engineering. Luettavissa: https://www.researchgate.net/publication/229504479_Process_Models_in_Software_Engineering. Luettu: 20.5.2021.

Schwaber, K, Sutherland, J. 2020. Scrum Guide 2020. Luettavissa: https://scrumguides.org/scrum-guide.html#scrum-theory. Luettu: 6.7.2020.

SFIA 2014. SWEBOK – The Guide to the Software Engineering Body of Knowledge. Luettavissa: https://sfia-online.org/staging/en/tools-and-resources/bodies-of-knowledge/swebok-the-guide-to-the-software-engineering-body-of-knowledge. Luettu: 20.7.2021.

Sommerville, I. 2007. Software Engineering 8th edition. Luettavissa: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxwaXBlcnBhc2hhfGd4OjRmODdjNzY3NDU4Njc4ZWE. Luettu: 3.5.2021

Sommerville, I. 2011. Software Engineering 9th edition. Luettavissa: https://engineering.futureuniversity.com/BOOKS%20FOR%20IT/Software-Engineering-9th-Edition-by-Ian-Sommerville.pdf. Luettu. 6.4.2021.

Tsui. 2013. Essentials of Software Engineering, 3rd Edition. Jones & Bartlett Learning. Luettavissa: https://learning.oreilly.com/library/view/essentials-of-software/9781449691998/. Luettu: 5.5.2021.

Warden, S. 2003. Extreme Programming Pocket Guide. O'Reilly Media, Inc. Luettavissa: https://learning.oreilly.com/library/view/extreme-programming-pocket/9781449399849/pt02.html. Luettu: 21.3.2021.

Widrig, D, Leffingwell, D. 2003. Managing Software Requirements: A Use Casse Approach, Second Edition. Addison-Wesley Professional. Luettavissa: https://learning.oreilly.com/library/view/managing-software-requirements/032112247X/ch03.html. Luettu: 20.5.2021.