Tampereen yliopisto

Niklas Nurminen

# PERINTEISEN JA KETTERÄN OHJELMISTOKEHITYKSEN VÄLISET EROT OHJELMISTOTUOTANNOSSA

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Marko Helenius
Joulukuu 2020

# TIIVISTELMÄ

Niklas Nurminen: Perinteisen ja ketterän ohjelmistokehityksen väliset erot ohjelmistotuotannossa
Kandidaatintyö
Tampereen yliopisto
Ohjelmistotekniikka
Joulukuu 2020

---

Ohjelmistoja käytetään lähes jokaisessa tietoteknisessä laitteessa tänä päivänä. Tämä on johtanut erinäisiin ajatus- ja toimintamalleihin ohjelmistotuotannossa. Ohjelmistotuotanto pitää sisällään ohjelmistokehityksen, joka voidaan jakaa kahteen eri ryhmään: perinteiseen ja ketterään. Päätutkimuskysymys on selvittää näiden kahden välisiä eroja tutkimalla niitä eri näkökulmista. Alatutkimuskysymyksessä selvitetään, miten organisaatiot voivat suorittaa migraation perinteisestä ketterään ohjelmistokehitykseen.

Tutkimus suoritettiin systemaattisena kirjallisuuskatsauksena. Tutkimuskysymyksiin lähteet valikoituvat systemaattisesti neljästä eri tietokannasta. Lähteiden tuli käsitellä perinteisen ja ketterän ohjelmistokehityksen eroavaisuuksia tai migraatiota. Muut lähteet etsittiin normaalisti eri tietokannoista avainsanojen avulla. Lähes kaikki lähteet olivat tieteellisiä artikkeleita, kirjoja tai muita kirjallisuuslähteitä.

Tutkimuksen tulokseksi saatiin, että perinteiset ja ketterät ohjelmistokehitysmenetelmät eroavat toisistaan niin prosessi- kuin organisaatiotasolla. Perinteinen ohjelmistokehitys omaa jäykän rakenteen, jossa muutoksille ei ole tilaa. Ne etenevät suoraviivaisesti eli lineaarisesti alusta loppuun ennalta määrätyllä tavalla. Tärkeintä on kattava vaatimusmäärittely ja jatkuva dokumentointi. Ketterä ohjelmistokehitys on puolestaan joustava rakenne, mikä on tärkeää nykypäivän dynaamisessa toimintaympäristössä. Se pohjautuu enemmän yhteistyöhön ja ihmisiin kuin prosesseihin ja menetelmiin. Ketterät menetelmät perustuvat iteraatioihin, joissa uusia versioita tuotetaan lyhyellä aikavälillä useasti. Perinteiset menetelmät taas useasti perustuvat yhteen pitkään iteraatioon.

Migraatio perinteisestä ketterään ohjelmistokehitykseen muuttaa organisaatiorakennetta. Organisaation tulee oppia luottamaan uusiin työkaluihin ja menetelmiin. Sen lisäksi työntekijöiden jokapäiväinen toiminta muuttuu enemmän yhteistyölähtöiseksi ja vastuullisemmaksi, koska päätöksiä ei enää laadi vain projektinjohtaja. Toisin sanoen migraatio vaatii organisaatiolta avoimuutta ottaa vastaan näitä muutoksia. Kaikkia ketterien menetelmien työkaluja ja menetelmiä ei ole tarkoitus implementoida heti käyttöön vaan pikemminkin kokeilla, mitkä sopivat organisaatiolle, ja tätä kautta mahdollisesti ottaa käyttöön.

Avainsanat: ketterä ohjelmistokehitys, perinteinen ohjelmistokehitys, migraatio

# ALKUSANAT

Kandidaatintutkielman aiheen valinta oli ehkä työn haastavin osuus, koska mielessäni oli useita eri vaihtoehtoja. Lopulta kumminkin ohjelmistokehitysmenetelmät veivät voiton. Uskon, että tulevaisuudessa tulen käyttämään menetelmiä useasti. Tämän lisäksi menetelmät eivät vain rajoitu ohjelmistoalalle vaan ne soveltuvat myös muualle.

Haluaisin kiittää kandiryhmääni ja etenkin opponoijia.

Tampereella, 8.12.2020

Niklas Nurminen

# SISÄLLYSLUETTELO

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| ACM | engl. Association for Computing Machinery, joka on maailmanlaajuinen tieteellinen yhteisö. |
| IEEE | engl. Institute of Electrical and Electronics Engineers, joka on kansainvälinen tekniikan alan järjestö. |

.

# 1. JOHDANTO

Digitalisaatio kasvaa ja teknologia kehittyy jatkuvasti eteenpäin, mikä on vaikuttanut olennaisesti myös ohjelmistotuotannon kehittymiseen. Sen seurauksena uusia ajatus- ja toimintamalleja on kehitetty ja kehitetään edelleen.

Ohjelmistotuotanto on paljon muutakin kuin pätkä koodia. IEEE:n mukaan se tarkoittaa systemaattista, kurinalaista ja mitattavissa olevaa tapaa ohjelmistojen kehittämiseen, operointiin ja ylläpitoon [1]. Ohjelmistotuotanto on siis laaja käsite, jonka osa-alueista muodostuu ohjelmiston elinkaari. Se pitää sisällään vaatimusten määrittelyn, suunnittelun, toteutuksen, testauksen ja ylläpidon.

Ohjelmistokehitysmenetelmät voidaan jakaa kahteen eri osaan: perinteisiin ja ketteriin. Perinteisen ohjelmistokehityksen menetelmät (engl. traditional software development methologies) ovat lähes joustamattomia prosesseja, joissa tavoitteet ja järjestys on määritelty projektin alussa. Ketterät ohjelmistokehityksen menetelmät (engl. agile software development methologies) taas ovat tehokkaita, asiakaslähtöisiä ja joustavia, mikä on tärkeä elementti jatkuvasti muuttuvien markkinoiden keskellä. [2] Ohjelmistonkehitysmenetelmien avulla voidaan säätää ohjelmiston elinkaaren jäsentelyä.

Tämän kandidaatintyön aiheena on selvittää perinteisen ja ketterän ohjelmistokehityksen välisit erot ohjelmistotuotannossa. Eroja käsitellään monelta eri osa-alueelta, mutta keskeisimpänä asiana tutkitaan prosessien eroavaisuuksia. Lisäksi tutkitaan, miten migraatio perinteisestä ketterään ohjelmistokehitykseen onnistuu. Tutkimus toteutettiin systemaattisena kirjallisuuskatsauksena.

Työ alkaa tutkimusmenetelmän ja -aineiston esittelyllä luvussa kaksi. Varsinaiseen aiheeseen siirrytään luvussa 3, jossa esitellään ohjelmistotuotannon historiaa. Luvussa neljä esitetään perinteisiä ohjelmistokehitysmenetelmiä. Tarkasteltavia menetelmiä ovat vesiputousmalli (luku 4.1), spiraalimalli (luku 4.2) ja V-malli (luku 4.3). Luku viisi puolestaan käsittelee kahta ketterää ohjelmistokehitysmenetelmää: Extreme Programming (luku 5.1) ja Scrum (luku 5.2). Luvussa kuusi käsitellään perinteisen ja ketterän ohjelmistokehityksen eroavaisuuksia ja migraatiota. Se on jaettu neljään alalukuun seuraavasti: eroavaisuudet (luku 6.1), suosio (luku 6.2), migraatio (luku 6.3) ja

synteesi (luku 6.4). Seitsemännessä eli viimeisessä luvussa esitetään tutkielman yhteenveto.

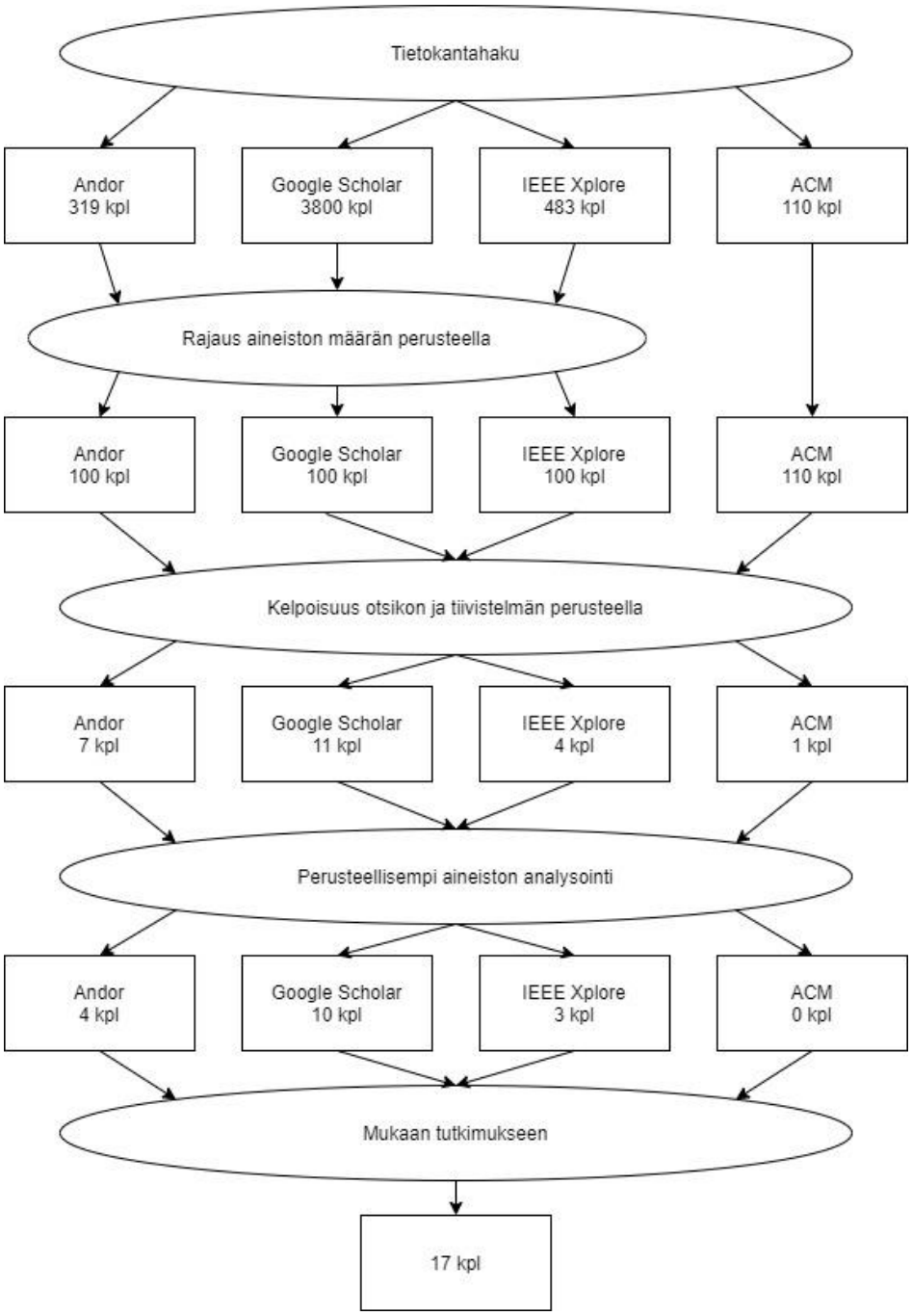
# 2. TUTKIMUSMENETELMÄ JA -AINEISTO

Tutkielma suoritettiin systemaattisena kirjallisuuskatsauksena. Tutkimuskysymyksiin etsittiin aineistoa systemaattisesti neljästä eri tietokannasta (Andor, Google Scholar, IEEE ja ACM). Tiedonhaku suoritettiin syyskuussa 2020. Tietokantahakua rajattiin käyttämällä kahta eri avainsanaa, jotka olivat ”*traditional software development*” ja ”*agile software development”*. Avainsanat oli yhdistetty ja-operaatiolla. Aineiston tuli käsitellä perinteisen ja ketterän ohjelmistokehityksen eroja tai niiden migraatiota. Lähteitä löytyi aiheesta runsaasti varsinkin ketterästä ohjelmistokehityksestä. Valtaosa aineistosta on vertaisarvioituja tieteellisiä artikkeleita, julkaisuja tai muita kirjallisuuslähteitä. Vallitsevan koronapandemian takia kaikkien lähteiden tuli olla sähköisessä muodossa.

Aineiston karsinta suoritettiin kolmessa eri vaiheessa kuvan 1 mukaisesti. Ensimmäisessä vaiheessa rajattiin tarkasteltavien lähteiden määrää ilman perehtymistä ainestoon, koska aineisto löytyi niin runsaasti. Lähteitä löytyi seuraavasti:

- Andor 319 kpl
- Google Scholar 3800 kpl
- IEEE Xplore 483 kpl
- ACM 110 kpl

ACM tietokannasta huomioitiin kaikki lähteet, koska niitä löytyi vain 110 kappaletta. Lopuista tietokannoista huomioitiin vain 100 ensimmäistä. Toisessa vaiheessa kaikki lähteet katsottiin läpi niin, että vain otsikko ja tiivistelmä luettiin. Tämä vaihe karsi suurimman osan lähteistä. Viimeisessä eli kolmannessa vaiheessa tutustuttiin perusteellisemmin aineistoon. Tällä tarkoitetaan sitä, että lähteet selattiin läpi ja varmistettiin, että haluttu tieto löytyy sieltä. Tämän perusteella lähde otettiin mukaan tutkimukseen tai hylättiin. Karsinnan lopputuloksena saatiin 17 lähdettä, jotka otettiin mukaan tutkimukseen.




***Kuva 1. Aineiston karsinnan toteutus***

Pääaineiston lisäksi kerättiin yksityiskohtaisempaa tietoa esimerkiksi ohjelmistotuotannon historiasta ja ohjelmistokehityksen eri menetelmistä. Lähteitä etsittiin muun muassa seuraavilla hakusanoilla: ”*software development history*”, ”*scrum*”, ”*extreme programming*” ja ”*waterfall*”.

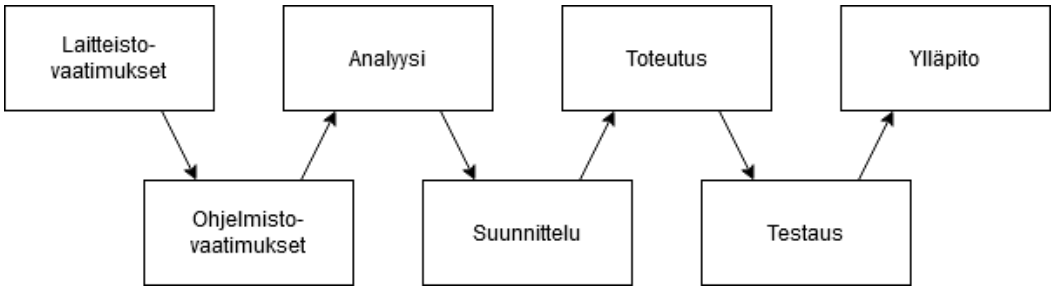
# 3. OHJELMISTOTUOTANNON HISTORIA

Ensimmäiset digitaaliset tietotokoneet otettiin käyttöön 1940-luvulla, minkä jälkeen tietotekniikka ja ohjelmointi alkoivat yleistymään. Ensimmäisiä ohjelmistokehityksen mentaliteetteja oli ”code and fix”, jossa ohjelmistoa kehitettiin ilman kunnollista suunnitelmaa sekä päätöksiä tehtiin ilman kunnollista pohdintaa. Alkeellinen mentaliteetti kuitenkin toimi sen ajan järjestelmissä, mutta niiden kehittyessä monimutkaisemmaksi törmättiin ongelmiin. Uusien ominaisuuksien käyttöönotto ja ongelmien korjaus ohjelmallisesti alkoivat olemaan erittäin vaikeita. [3, s. 8] Vuonna 1968 NATO järjesti konferenssin, jossa julkistettiin ohjelmistokriisi. Siihen oli monia syitä, mutta keskeisimmät olivat seuraavat:

- Projektit myöhästyivät aikataulusta ja budjetit ylittyivät.
- Ohjelmien dokumentaatiot olivat huonoja ja niiden käyttö vaikeaa.
- Ohjelmistot olivat tehottomia ja ylläpito vaikeaa.
- Ohjelmiston ominaisuudet eivät vastanneet vaatimuksia.

Konferenssissa esiteltiin ensimmäistä kertaa termi ”ohjelmistotuotanto”, jota kuvailtiin kurinalaiseksi insinöörintyöksi. Sen tarkoitus oli tarjota laadukkaita ja tehokkaita ohjelmistoja taloudellisesti. Se sisälsi myös työkalut ja tehtävänjaon työn tekemiseksi. [4, s. 8]

Winston Royce julkaisi vuonna 1970 artikkelin, jossa hän esittelee kuvan 2 mukaisen lineaarisen prosessimallin. Se kattaa ohjelmiston elinkaaren eri vaiheet. Mallia alettiin myöhemmin kutsua vesiputousmalliksi, joka on yksi perinteisen ohjelmistokehityksen menetelmistä. [5, s. 239] Lineaarisissa malleissa siis aluksi määritellään vaatimukset, minkä jälkeen edetään aina vaihe kerrallaan eteenpäin, kunnes lopuksi testataan ja jäädään ylläpitämään ohjelmistoa.




***Kuva 2. Winston Roycen kehittämä prosessimalli [5, s. 330]***

Lineaarisilla malleilla on lukuisia ongelmia, eikä edes Royce suositellut niitä käytettävän ainakaan monimutkaisissa ohjelmistoprojekteissa. Hän pikemminkin kannatti iteratiivisia ja inkrementaalisia ohjelmistokehityksen malleja, jotka alkoivatkin kasvattamaan suosiotaan 1990-luvun alussa. [6]

Iteratiivinen ja inkrementaalinen tapa perustuu siihen, että ohjelmistonkehitys jaetaan iteraatioihin eli aikaväleihin. Jokaisessa iteraatiossa suunnitellaan, määritetään, toteutetaan ja testataan ohjelmistoa, mikä mahdollistaa sen, että kehittäjät pystyvät oppimaan aiemmasta iteraatiosta ja kehittämään ohjelmistoa askel kerrallaan. Ohjelmisto siis valmistuu inkrementaalisesti. Mallista entistä paremman tekee se, että jokaisen iteraation vaiheessa keskustellaan asiakaspalautetta antavan asiakkaan kanssa. Asiakaspalautteen perusteella voidaan muuttaa ohjelmistoa seuraavaan iteraatioon. [6]

Perinteisen ohjelmistokehityksen metodologia ei kuitenkaan pysynyt mukana teknologian kehityksessä, sillä ohjelmistojen kehittäminen oli liian hidasta, ja se epäonnistui useasti. Tämän vuoksi muutamat itsenäiset asiantuntijat alkoivat kehitellä uusia ketterän ohjelmistokehityksen menetelmiä (engl. agile software development methods), kuten Extreme Programming ja Scrum. [7, s. 1–2]

Helmikuussa 2001 pidettiin kokous, johon osallistui 17 ketterien menetelmien asiantuntijaa. Kokouksen tavoite oli määritellä yleiset periaatteet ketterälle ohjelmistokehitykselle. Tulokseksi saatiin määritelmä termille ”ketterä” (engl. agile). Termi koostuu neljästä osasta: ”Agile”, ”Agile Manifesto”, 12 periaatetta ketterille menetelmille ja ”Agile Alliance”. [8, luku 1]

### *Agile*

Termillä tarkoitetaan ohjelmistoteollisuuden kevyitä projektinhallintaprosesseja, jotka kannustavat iteratiiviseen lähestymistapaan [8, luku 1].

### *Agile Manifesto*

Agile Manifesto luo pohjan ketterille menetelmille. Manifesto koostuu neljästä vastakkainasetetusta kohdasta, jotka ovat seuraavat [8, luku 1]:

- Yksilöt ja interaktio ovat tärkeämpiä kuin prosessit ja työkalut.
- Toimiva ohjelmisto on tärkeämpi kuin kattava dokumentaatio.
- Asiakasyhteistyö on tärkeämpi kuin sopimusneuvottelut.
- Muutoksiin reagointi on tärkeämpää kuin suunnittelun noudattaminen.

### *12 periaatetta ketterille menetelmille*

1. Tärkein tavoite on tyydyttää asiakas, mikä onnistuu toimittamalla aikaisin ja säännöllisesti versioita ohjelmistosta.
2. Ota vastaan muutokset, vaikka ne tulisivat eteen projektin loppupuolella. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailuedun saavuttamiseksi.
3. Toimita parin viikon tai kuukauden välein toimiva versio ohjelmistosta.
4. Toimiva ohjelmisto on tärkein mittari edistymiselle.
5. Rakenna projekteja motivoituneiden yksilöiden kanssa. Anna heille tuki ja puitteet, joita he tarvitsevat ja luota siihen, että he saavat työn tehtyä.
6. Parhaat suunnitelmat, vaatimukset ja arkkitehtuurit syntyvät itseorganisoituvissa tiimeissä.
7. Tehokkain ja toimivan tapa informaation välittämiseksi on keskustella kasvotusten kehittäjätiimin kanssa.
8. Liikemiehien ja kehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.
9. Yksinkertaisuus, ei tärkeiden tavoitteiden eliminointi, on välttämätöntä.
10. Jatkuva huomio tekniseen osaamiseen ja suunnitteluun parantaa ketteryyttä.
11. Tietyin aikavälein tiimi reflektoi, miten se olla vieläkin tehokkaampi, ja tämän avulla tiimi voi säätää käyttäytymistään oikein.
12. Ketterät menetelmät kannustavat kestävää kehittymistä. Sponsoreiden, kehittäjien ja käyttäjien tulisi pystyä ylläpitämään tasaista tahtia loputtomiin. [8, luku 1]

### *Agile Alliance*

Agile Alliance on voittoa tavoittelematon globaali organisaatio, jonka tavoite on edistää ketterää ohjelmistokehitystä. Organisaatiossa uskotaan, että ketterät menetelmät luovat nopeasti arvoa, mikä tekee ohjelmistoteollisuudesta tuottavamman, ihmiskeskeisemmän ja kestävämmän. [8, luku 1]

# 4. PERINTEISEN OHJELMISTOKEHITYKSEN MENETELMIÄ

## 4.1 Vesiputousmalli

Vesiputousmalli (engl. waterfall model) on yksi tunnetuimmista ja vanhimmista ohjelmiston elinkaarimalleista. Malli on niin suosittu, että se on edelleenkin aktiivisesti käytössä eri ohjelmistotuotanto projekteissa. [9, s. 106] Vesiputousmalli perustuu viiteen eri vaiheeseen kuvan 3 mukaisesti.

***Vaatimusten määrittely***

Ensimmäisessä vaiheessa kerätään kaikki tarvittava tieto ohjelmiston- ja järjestelmänvaatimuksista. Vaatimusten tulee olla niin hyvin laadittuja, että ohjelmisto pystytään näiden avulla toteuttamaan alusta loppuun. Tässä vaiheessa ei keskitytä tarkastelemaan, miten asiat tehdään vaan pikemminkin, mitä tehdään. Vaatimukset dokumentoidaan käyttämällä luonnollista kieltä, ei ohjelmointikieltä. Dokumentointivaihetta kutsutaan mallin vaatimusdokumentoinniksi (engl. software requirements specification). Tämä vaihe on usein kaikkein virhealttein sekä eniten aikaa ja resursseja vievä. [9, s. 106; 10, s. 2680]

***Suunnittelu***

Suunnitteluvaihe perustuu ensimmäisen vaiheen vaatimusdokumentaatioon, jota kehittäjät opiskelevat ja tutkivat. Tämän avulla voidaan kartoittaa ohjelman rakenne, ohjelmointikieli, tietokannat ja käyttöliittymä. Tällöin saadaan määriteltyä systeemin arkkitehtuuri, jotka kehittäjät noudattavat. Tämäkin vaihe dokumentoidaan tarkasti, ja sitä kutsutaan ohjelmistonsuunnittelu päätökseksi (engl. software design description). [10, s. 2681]

***Toteutus***

Tässä vaiheessa alkaa itse ohjelmointityö, joka perustuu täysin suunnitteluvaiheen dokumentaatioon. Ohjelmistoa kehitetään ja testataan yksikkötesteillä moduuli kerrallaan. Vasta onnistuneen testauksen jälkeen voidaan integroida moduuleja toisiinsa. Useasti tässä vaiheessa tulee ongelmia, jotka vaativat moduulin korjaamista. Ongelmat voivat johtua erinäisistä odottamattomista ongelmista, kuten huonosta

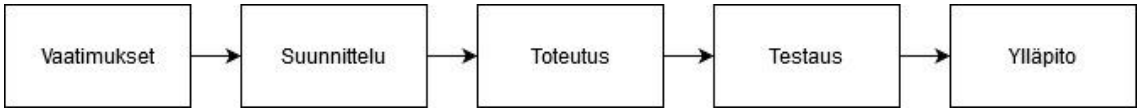
kommunikaatiosta tai muista piiloutuneista suunnitteluvaiheen ongelmista. [9, s. 107; 10, s. 2681]

***Testaus***

Edeltävän vaiheen yksikkötestit eivät testaa koko ohjelman toimintaa, koska moduulit testattiin erikseen eikä niiden yhteensopivuutta testattu toisiinsa. Tässä vaiheessa testataan ohjelmisto yhtenä kokonaisuutena, jotta varmistetaan tuotteen laatu. Kattava ohjelmistontestaus alentaa ylläpitokuluja ja tekee tuotteesta luotettavamman. Tätä kautta myös asiakkaat pysyvät tyytyväisempänä. Testausvaihe maksaa puolet koko projektin kustannuksista. [9, s. 107; 10, s. 2681]

***Ylläpito***

Ylläpitovaihe tulee vasta sen jälkeen, kun ohjelmisto on julkaistu. Vaihe pitää sisällään mahdolliset ongelman korjaukset, muutokset tai kokonaan uusien ominaisuuksien toteuttamisen. [9, s. 107; 10, s. 2681]




***Kuva 3. Vesiputousmalli [10, s. 2682]***

Vesiputousmallin hyviä puolia ovat seuraavat:

- Se on yksinkertainen ja helposti ymmärrettävä.
- Vaiheet etenevät lineaarisesti eteenpäin yksi kerrallaan.
- Malli toimii hyvin, kun projekti on pieni ja vaatimukset voidaan määritellä hyvin.
- Virheet nähdään helposti jo projektin alkutekijöissä.
- Vaiheet ovat määritelty tarkasti, mikä vähentää väärinymmärryksiä ja ongelmatilanteita.
- Ei vaadi paljon resursseja.
- Dokumentaatio on hyvin laadittu, mikä helpottaa myös tulevaisuudessa ongelmien ratkaisua ja luettavuutta. [11, s. 114; 12, s. 6]

Puolestaan huonoja puolia ovat seuraavat:

- Pientenkin muutosten takia projekti pitää aloittaa alusta tai pahimmassa tapauksessa hylätä.
- Prototyyppejä ei tule, vaan toimiva ohjelma saadaan tuotettua vasta projektin lopussa.

- Muutoksia ei hyväksytä, joten malli ei ole joustava.
- Vaiheiden kestoa on vaikea arvioida, joten tuote saattaa myöhästyä.
- Vaiheissa ei voi mennä taaksepäin. [11, s. 114; 12, s. 6]

## 4.2 Spiraalimalli

Spiraalimalli on yksi tärkeimpiä ohjelmiston elinkaarimalleja, jossa korostetaan riskien analysointia ja hallintaa. Spiraali on jaettu kuvan 4 mukaisesti neljään eri vaiheeseen. Vaiheet suoritetaan vesiputousmallin tavoin lineaarisessa järjestyksessä niin, että ohjelmisto rakentuu vaiheittain eli inkrementaalisesti eteenpäin. Spiraali koostuu silmukoista, joiden määrä vaihtelee projektin riskin mukaan. Jokainen silmukka käy läpi kaikki neljä eri vaihetta. [13]

***Määritä tavoitteet***

Ensimmäisessä vaiheessa kerätään aina asiakkaalta vaatimukset, joita analysoidaan ja joiden pohjalta määritellään tavoitteet. Tässä vaiheessa esitetään myös kaikki vaihtoehtoiset ratkaisut koko silmukalle. [13]

***Tunnista ja ratkaise riskit***

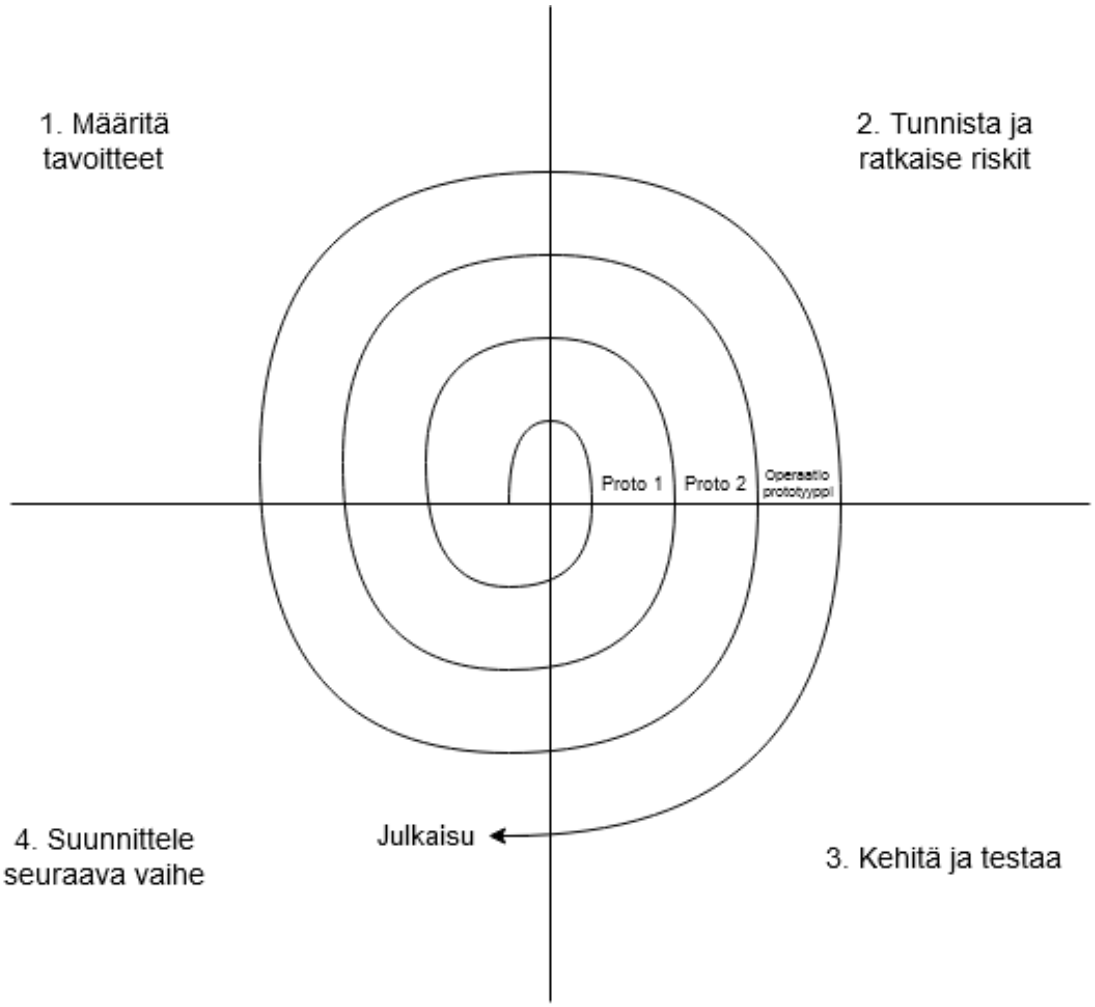
Toisessa vaiheessa tutkitaan projektiin liittyviä riskejä. Projektin sisältäessä riskejä kerätään kaikki mahdolliset ratkaisut, joilla voidaan ratkaista riski. Tämän jälkeen pohditaan, mikä ratkaisusta olisi kaikista parhain. Parhaimmat ratkaisun löydyttyä voidaan toteuttaa strategia. Tämän vaiheen lopussa tuotetaan prototyyppi ohjelmistosta. [9, s. 107; 13]

***Kehitä ja testaa***

Kolmannessa vaiheessa ohjelmistoa kehitetään ja testaillaan. Vaiheen lopuksi on tuotettu uusi versio ohjelmistosta. [9, s. 107; 13]

***Suunnittele seuraava kierros***

Neljännessä eli viimeisessä vaiheessa asiakas arvioi tuotettua ohjelmistoa. Arvioinnin jälkeen voidaan siirtyä seuraavaan silmukkaan, jos silmukoita on jäljellä. [9, s. 107; 13]



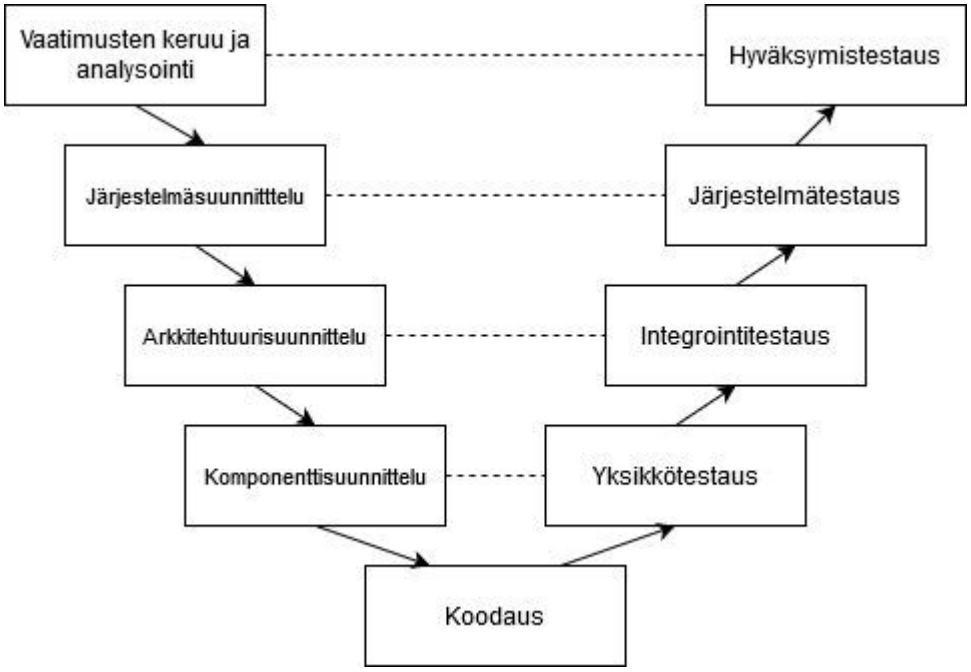

***Kuva 4. Spiraalimalli [13]***

Spiraalimalli soveltuu parhaiten projekteihin, joissa riskien ja kustannusten arviointi on ensiarvoisen tärkeää. Sen lisäksi se soveltuu isoihin projekteihin, joissa projektin vaatimukset eivät ole täysin selviä tai monimutkaisia. Malli tuottaa iteraatioiden välissä prototyyppejä, joiden avulla asiakas pystyy näkemään ohjelmiston nykytilanteen, ja tätä kautta hän pystyy vaikuttamaan seuraaviin iteraatioihin. [9, s. 108; 13]

Malli ei kuitenkaan ole virheetön. Vertailessa sitä muihin ohjelmistokehitysmenetelmiin huomataan, että se vaatii huomattavan määrän suunnittelua ja dokumentaatiota. Myöskään se ei ole kustannustehokas, sillä mitä pidemmälle spiraalia kierretään, sitä kalliimmaksi se tulee. Näiden lisäksi malli vie runsaasti aikaa, koska riskien tunnistaminen ja analysointi täytyy tehdä tarkasti. Ongelmien takia mallia ei ole tehokasta käyttää pienissä projekteissa. [9, s. 108; 13]

## 4.3 V-malli

V-malli on yksi jatkokehitelty malli vesiputousmallista. Mallin on kehittänyt Paul Rook myöhään 1980-luvulla, mutta sitä käytetään edelleen tänä päivänä. [14 s.30] V-mallin tarkoituksena on parantaa muutamia vesiputousmenetelmän riskejä, kuten verifiointi ja validointi. Verifioinnilla tarkoitetaan staattista analyysiä, jossa tutkitaan, onko ohjelmiston vaatimukset saavutettu ilman, että ohjelmistoa ajetaan. Validoinnilla puolestaan tarkoitetaan staattista analyysiä, jolla tutkitaan, saavuttaako ohjelmisto vaatimukset ja oletukset, kun ohjelmistoa ajetaan. [15] Malli etenee lineaarisesti, kuten vesiputousmallikin, mutta tärkeimpänä ominaisuutena jokaisessa suunnittelu- ja toteutusvaiheessa sillä on myös testausvaihe. Tällöin jokaiselle vaiheelle määritellään vastaavia testejä testausvaiheeseen kuvan 5 mukaan. Tämä mahdollistaa tehokkaan ja riskittömämmän ohjelmistokehityksen, sillä riskit voidaan havaita jo prosessin alkuvaiheessa. [16 s.10]




***Kuva 5. V-malli [14, s. 31]***

V-malli voidaan jakaa kahteen eri osaan: suunnitteluun ja toteutukseen sekä testaukseen. Suunnitteluun ja toteutukseen kuuluvat seuraavat vaiheet [15]:

- *Vaatimusten keruun ja analysoinnin* vaiheessa kommunikoidaan asiakkaan kanssa ja kartoitetaan, mitkä ovat vaatimukset ja odotukset ohjelmistolle.
- *Järjestelmänsuunnittelussa* suunnitellaan järjestelmää, joka pitää sisällään laitteisto- ja kommunikointityökalut.

- *Arkkitehtuurisuunnittelussa* aiempaa vaihdetta pilkotaan pienempiin komponentteihin, jotka sisältävät funktioita. Näin saadaan käsitys siitä, miten ohjelmiston kommunikointi ja tiedonsiirto toimivat moduulien kesken.
- *Komponenttisuunnittelussa* moduulin toiminnallisuus suunnitellaan loppuun. Tätä vaihetta kutsutaan myös nimellä ”Low-Level Design” eli LLD.

Testaus-osioon kuuluvat seuraavat vaiheet [15]:

- *Yksikkötestaus* suunnitellaan komponenttisuunnitteluvaiheessa. Vaiheen tarkoitus on eliminoida mahdollisia bugeja koodi- tai yksikkötasolla.
- *Integraatiotestaus* suoritetaan yksikkötestauksen jälkeen. Vaiheen tarkoitus on testata moduulien yhteensopivuutta muiden moduulien kanssa. Tämä suunnitellaan arkkitehtuurisuunnittelu vaiheessa.
- *Järjestelmätestaus* testaa koko ohjelmistoa, kuten funktioita ja kommunikaatiota joka suuntaan. Tämä suunnitellaan järjestelmäsuunnitteluvaiheessa.
- *Hyväksymistestaus* tunnetaan myös englanniksi nimellä ”User Acceptance Testing” eli UAT. Vaiheen tarkoitus on tutkia, ovatko kaikki asiakkaan vaatimukset huomioitu ja onko ohjelmisto julkaisukelpoinen.

V-malli sopii parhaiten projekteihin, joissa vaatimukset ovat alussa selvät eivätkä ne muutu prosessin aikana. V-mallia käytetään suurimmaksi osaksi sen takia, että se on yksinkertainen hallita, ja virheet löydetään jo heti projektin alkutekijöissä. [15] Lisäksi muita hyviä puolia ovat seuraavat:

- Testausvaiheet on määritelty ennen koodausta, mikä säästää paljon aikaa. [12, s. 7]
- Malli toimii pienille projekteille. [12, s. 7]
- Onnistumistodennäköisyys on suurempi, koska menetelmä keskittyy verifiointiin ja validointiin koko elinkaaren ajan. [15]

Huonoja puolia puolestaan ovat seuraavat:

- Mallin rakenne on jäykkä, sillä vaatimukset pitää olla määritelty aluksi eikä niitä voi kesken prosessin muuttaa. [12, s. 7]
- Menetelmä ei tue yhdenaikaisuutta vaiheiden tekemisen kanssa. [12, s. 7; 16]
- V-malli ei perustu iteraatioihin. [15]
- Malli ei sovellu olio-ohjelmointiprojekteihin. [16]

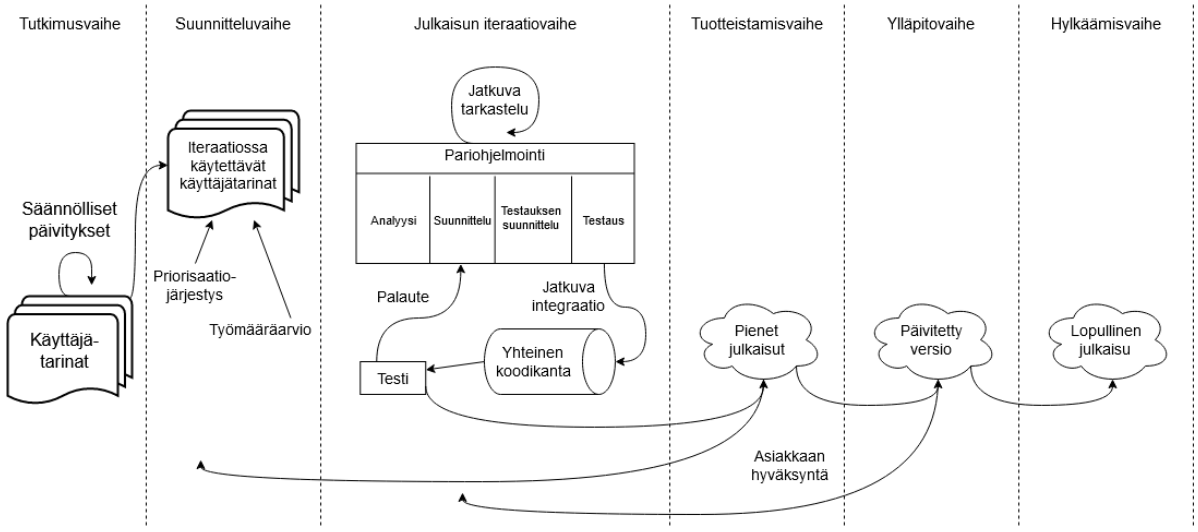
- Malli ei sisällä tiettyä reittiä, jolla voidaan tutkia ongelmia testausvaiheessa. [12, s. 7]

# 5. KETTERÄN OHJELMISTOKEHITYKSEN MENETELMIÄ

## 5.1 Extreme Programming (XP)

Extreme Programming on tunnetuimpia ketterän ohjelmistokehityksen menetelmiä. Sen kuvaillaan soveltuvan parhaiten pieniin ja keskisuuriin kehittäjätiimeihin, jotka toimivat dynaamisessa ympäristössä [17]. XP noudattaa viittä eri pääperiaatetta, jotka ovat kommunikointi, yksinkertaisuus, palaute, rohkaiseminen ja kunnioitus. Menetelmä on suosittu sen asiakastyytyväisyyden vuoksi, sillä XP:n avulla priorisoidaan ominaisuuksia, joita asiakas tarvitsee sillä hetkellä. Tämän lisäksi XP:n säännöt ovat yksinkertaiset, mikä tekee siitä intuitiivisen käyttää. Voidaan puhua, että XP koostuu pienistä palasista, jotka eivät yksikseen käy järkeen, mutta kun ne yhdistetään, saadaan toimiva ja tehokas menetelmä. [18]

XP:n elinkaari muodostuu kuudesta vaiheesta kuvan 6 mukaisesti: tutkimus, suunnittelu, julkaisun iteraatio, tuotteistaminen, ylläpito ja hylkääminen. [19, s. 19]




***Kuva 6. XP:n elinkaaren vaiheet [19]***

*Tutkimusvaiheessa* (engl. exploration phase) käyttäjät kirjoittavat käyttäjätarinoita (engl. user story), joita he haluavat ensimmäiseen julkaisuun. Käyttäjätarinat vastaavat ominaisuutta, joka halutaan lisätä ohjelmistoon. Tämän lisäksi projektin työntekijät tutustuvat teknologiaan, käytäntöihin ja työkaluihin, joita projektissa tullaan käyttämään. Vaihe kestää yleensä muutamasta viikosta muutamaan kuukauteen. [19, s. 20]

*Suunnitteluvaiheessa* (engl. planning phase) käyttäjät asettavat käyttäjätarinat priorisaatiojärjestykseen ja päättävät, mitkä niistä tulevat ensimmäiseen julkaisuun mukaan. Tämän jälkeen kehittäjät arvioivat työmäärän kullekin käyttäjätarinalle. Tätä kautta saadaan yhdessä käyttäjien kanssa arvioitua ensimmäisen julkaisun päivämäärä. Tämä vaihe kestää pari päivää. [19, s. 20]

*Julkaisun iteraatiovaiheessa* (engl. iterations to release phase) aikataulu hajotetaan pienempiin iteraatioihin, joista jokaisen kesto on neljä viikkoa. Järjestelmän arkkitehtuuri luodaan ensimmäisessä iteraatioissa. Jokaisen iteraation lopussa suoritetaan toiminalliset testit. Lopuksi, kun viimeinenkin iteraatio on suoritettu, on järjestelmä valmis tuotantoon. [19, s. 20]

*Tuotteistamisvaiheessa* (engl. productionizing phase) järjestelmän suorituskykyä testataan ennen kuin se voidaan julkaista asiakkaalle. Tässä vaiheessa muutoksia voi vielä tulla, mikä voi johtaa uusiin iteraatioihin. Tällaisissa tapauksissa iteraatioiden tahtia kuitenkin pitää nopeuttaa esimerkiksi kolmesta viikosta viikkoon. [19, s. 20]

*Ylläpitovaiheessa* (engl. maintenance phase) ylläpidetään aiemmin julkaistua versiota järjestelmässä samaan aikaan, kun uusia iteraatioita suoritetaan. [19, s. 20]

*Hylkäämisvaihe* (engl. death phase) konkretisoituu siinä vaiheessa, kun käyttäjällä ei ole vaatimuksia uusille ominaisuuksille eli käyttäjätarinoille. Tässä vaiheessa voidaan todeta, että ohjelmisto on valmis, ja sen kehittäminen voidaan lopettaa. Lopuksi kuitenkin kirjoitetaan dokumentaatio järjestelmästä. [19, s. 21]

XP perustuu 12:een seuraavaan eri käytäntöön, joita voidaan käyttää elinkaaren eri vaiheissa [20, s. 99]:

1. Suunnittelupelissä määritetään haluttavat ominaisuudet sekä seuraavan julkaisun ajankohta.
2. Pienet julkaisut mahdollistavat nopeammin uuden version julkaisun, koska tehdään vähän kerrallaan.
3. Metafora auttaa ymmärtämään järjestelmän toimintaa, kun kaikilla on sama käsitys, mitä asiat tarkoittavat.
4. Yksinkertainen suunnittelu auttaa keskittymään nykyhetkeen, koska siinä minimoidaan ylimääräinen suunnittelu.
5. Refaktoroinnissa muokataan koodia paremmaksi ilman, että sen toiminnallisuus muuttuu.
6. Testaus varmistaa, että ominaisuudet toimivat halutulla tavalla. Testien läpipääsy on myös edellytys projektin jatkamiselle.
7. Pariohjelmoinnissa koodia kirjoitetaan parin kanssa yhdessä yhdellä tietokoneella.
8. Yhteisomistajuus mahdollistaa, että kehittäjistä kuka tahansa voi muokata koodia riippumatta järjestelmän osasta.

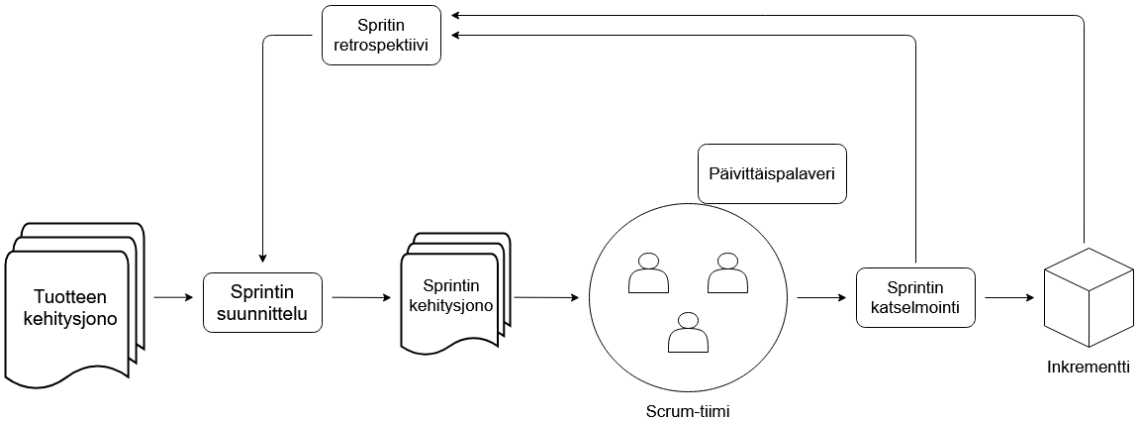
9. Jatkuva integraatio tarkoittaa, että uutta koodia integroidaan useasti päivässä järjestelmään.
10. 40-tuntinen viikko takaa, että kukaan ei työskentele enempää kuin 40 tuntia viikossa.
11. Ohjelmointistandardit ovat yhteisiä standardeja koodissa, mikä parantaa kehittäjien kommunikointia, sillä periaatteet ovat samat.
12. Saavutettavissa oleva asiakas mahdollistaa jatkuvan yhteydenpidon asiakkaaseen.

XP:n avulla ohjelmistokehitys on joustavaa ja helposti havaittavaa, sillä uusia versioita tuotetaan nopealla tahdilla. Se ottaa asiakkaan huomioon, ja virheet löytyvät nopeasti jatkuvan testauksen vuoksi. Järjestelmän koodi on parempaa, sillä se toteutetaan pariohjelmoinnin avulla. Ohjelmointia tekee siis aina samanaikaisesti kaksi silmäparia, mikä mahdollistaa heidän tietämysten yhdistämisen. XP sopii isoillekin projekteille, sillä kaikilla on yhteiset standardit ja jokainen saa toimia eri osa-alueilla. [12, s. 10]

XP ei kuitenkaan ole virheetön vaan siinä on myös tiettyjä rajoitteita. Kehittäjien eri tietämyksen tasot voi vaikeuttaa ohjelmointia, koska koodaus tehdään pääosin pariohjelmointina. Tällöin toinen joutuu tekemään käytännössä kaiken. Tämän lisäksi pariohjelmointi voi aiheuttaa joillekin henkilöille suorituspaineita, sillä he ovat jatkuvasti tarkkailevan silmän alla. XP ei myöskään takaa, että julkaisut ovat aina ajan tasalla, koska ongelmia saattaa tulla varsinkin silloin, kuin projektia johtaa epäpätevä johtaja. [12, s. 10–11]

## 5.2 Scrum

Scrum on malliltaan iteratiivinen ja inkrementaalinen, kuten monet muutkin ketterän ohjelmistokehityksen menetelmät. Scrum on projektinhallintaan tarkoitettu viitekehys, johon voidaan ottaa erilaisia tekniikoita ja prosesseja mukaan. Scrum on yksinkertainen ymmärtää, mutta erittäin vaikea taitaa. Viitekehys koostuu tapahtumista, tuotoksista, säännöistä ja Scrum-tiimistä kuvan 7 mukaan. [21, s. 4]




***Kuva 7. Scrum-viitekehys [22]***

Scrum-tiimi rakentuu kehitystiimistä, Scrum Masterista ja tuoteomistajasta. Kyseinen rakenne optimoi luovuuden, joustavuuden ja tuottavuuden. Onnistuminen kumminkin riippuu siitä, miten hyvin jäsenet pystyvät omaksumaan viitekehyksen arvoja, joita ovat sitoutuminen, avoimuus, kunnioitus, rohkeus ja keskittyminen. Tiimin sisällä täytyy olla myös avoin ilmapiiri, jotta voidaan saavuttaa haluttu päämäärä. [21, s. 6] Tiimiin kuuluvat seuraavat jäsenet [21, s 7–8]:

- *Tuoteomistaja* vastaa tuotteen arvon maksimoinnista ohjaamalla kehitystiimiä. Toisin sanoen tuoteomistaja tekee päätökset, joita kehitystiimin tulee kunnioittaa. Loppupeleissä hän on vastuussa tuotteesta.
- *Kehitystiimi* koostuu alan ammattilaisista, jotka kehittävät tuotetta kehitysjonon sisällön mukaisesti. Henkilöt ovat itseohjautuvia, ja yhdessä he soveltavat osaamistaan saavuttaakseen päämäärän. Tiimin koon tulisi olla 3–9 henkilöä, jotta se pysyy ketteränä eikä koordinointiin mene liikaa aikaa.
- *Scrummaster* vastaa itse viitekehyksen tukemisesta ja edistämisestä. Hän auttaa niin sisäisiä jäseniä kuin ulkopuolisia ymmärtämään käytäntöjä, sääntöjä ja teoriaa.

Scrum-viitekehys koostuu viidestä eri tapahtumasta: sprintin suunnittelu, sprintti, päivittäispalaveri, sprintin katselmointi ja sprintin retrospektiivi. Tapahtumat ovat ennalta sovittuja ja aikarajattuja. Maksimipituudet voidaan päättää ennen itse tapahtumaa, mutta kuitenkin niin, että asiat saadaan käytyä läpi huolella. Sprintti pitää itsessään kaikki tapahtumat sisällään. [21, s. 9]

- *Sprintin suunnittelussa* laaditaan, mitä sprintissä tullaan tekemään. Siihen osallistuu kaikki tiimin jäsenet. Suunnittelussa pyritään samaan vastaus kahteen kysymykseen. Ensimmäiseksi pohditaan, mitä toiminnallisuuksia voidaan ottaa

mukaan alkavaan sprinttiin eli määritetään realistiset tavoitteet. Tämän jälkeen pohditaan, miten työ toteutetaan valmiiksi inkrementiksi. [21, s. 10]

- *Sprintti* on tavallaan koko viitekehyksen sydän, joka kestää enintään kuukauden kerrallaan. Tämän aikana tuotetaan potentiaalisesti julkaisukelpoinen inkrementti. Sprintin aikana kuitenkin voidaan tehdä useampikin julkaisu, joista inkrementti koostuu. Sprintti on jatkuvasti käynnissä. Kun aiempi loppuu uusi alkaa välittömästi. Sprintti siis sisältää viitekehyksen kaikki tapahtumat. Se voidaan keskeyttää vain ja ainoastaan tuoteomistajan luvalla, mutta hänen päätöksiinsä voivat vaikuttaa muut sidosryhmät tai tiimin sisäiset henkilöt. [21, s. 9]
- *Päivittäispalaveri* on tapahtuma, jossa suunnitellaan työt aina seuraaviksi 24 tunniksi. Se pidetään sprintin jokaisena päivänä. Sen enimmäiskesto tulisi olla 15 minuuttia. Palaverin tavoite on tarkastella edistymistä, mutta myös samalla parantaa tiimin yhteishenkeä. Tämän lisäksi päivittäispalaveri auttaa kehitystiimiä saavuttamaan sprintin tavoitteet. [21, s. 12]
- *Sprintin katselmointi* tapahtuu sprintin lopussa. Sen tarkoitus on tarkastella kehitettyä inkrementtiä yhdessä Scrum-tiimin ja muiden sidosryhmien kanssa. Siinä käydään läpi toiminnallisuuksia, joita saatiin valmiiksi ja jotka jäivät vielä kesken. Katselmointi on tavallaan epävirallinen palaveri, jonka tavoitteena on saada palautetta ja edistää vuoropuhelua. [21, s. 13]
- *Spritin retrospektiivin* tarkoitus on kehittää Scrum-tiimin työskentelyä seuraavaan sprinttiin. Tapahtuma on ennen seuraavan sprintin suunnittelupalaveria, mutta kuitenkin sprintin katselmoinnin jälkeen. Retrospektiivin tavoitteena on tarkastella, miten edellinen sprintti sujui. Tämän avulla voidaan tunnistaa ja määrittää tärkeimmät parannukset ja luoda Scrum-tiimin omaa suunnitelmaa työskentelyn edistämiseksi. [21, s. 14]

Viitekehys pitää sisällään myös kolme eri tuotosta. Näiden tarkoitus on kuvata työmäärää tai lisätä arvoa [21, s. 14]. Tuotokset ovat seuraavat:

- *Tuotteen kehitysjono* on lista, jossa pidetään kaikki tiedot tuotteesta. Toisin sanoen se pitää sisällään valmiit ja tulossa olevat toiminnallisuudet sekä vaatimukset ja muutokset. Listan kohteet pitävät sisällään kuvauksen, arvon ja työmääräarvion. Itse lista ei tule koskaan valmiiksi vaan se on olemassa niin kauan, kun tuote on olemassa. [21, s. 15]

- *Sprintin kehitysjono* puolestaan pitää sisällään sprintin tavoitteet. Toisin sanoen asiat, jotka tulevat kyseisen sprintin aikana inkrementtiin. Kehitysjonosta nähdään reaaliajassa sprintin edistymistä, sillä sitä mukaan, kun asioita tehdään, ne merkataan valmiiksi. [21, s. 16]
- *Inkrementti* on sprintin ja sitä edeltävien sprinttien aikaansaannos. Tuotteen tulee olla käyttövalmis. Inkrementtien avulla tuotetta rakennetaan pala kerrallaan eteenpäin. [21, s. 17]

Scrum viitekehyksessä sprintit ovat lyhyitä ja palaute jatkuvaa, joten se sopeutuu hyvin dynaamiseen ympäristöön. Scrum-tiimillä on aina tietty tavoite, ja heitä motivoidaan jatkuvasti niin yksilöinä kuin tiiminä, mikä näkyy positiivisesti projektin lopputuloksessa. Projektin kehittymistä on helppo tarkkailla, koska kehitysjonosta näkyy tulevat ja tehdyt toiminnallisuudet. Näiden lisäksi Scrum toimii kaikille teknologioille ja ohjelmointikielille. [12, s. 9]

Scrum viitekehys perustuu pitkälti Scrum-tiimin yhteishenkeen ja luottamukseen. Yksilötasolla taas motivaatioon ja taitoihin. Näiden uupuessa voidaan joutua tilanteeseen, jossa tehtävät eivät ole hyvin määriteltyjä ja projekti ei etene suunnitelman mukaisesti. Tämä johtaa projektin epäonnistumiseen. [12, s. 10]

# 6. PERINTEISEN JA KETTERÄN OHJELMISTOKEHITYKSEN VERTAILU

## 6.1 Eroavaisuudet

Ketterät ja perinteiset menetelmät eroavat toisistaan monin eri tavoin niin prosesseissa kuin projektinhallinnassa. Eroavaisuuksien ymmärtämisellä ja niiden soveltamisella on tärkeä vaikutus menetelmän valintaprosessiin. Ei ole järkevää valita menetelmää, jota ei ymmärrä tai hallitse tai joka ei sovi projektin luonteeseen. Kuitenkin kaikkien menetelmien päätarkoitus on parantaa ohjelmistokehitystä ja tuottaa tuote tietyssä aikavälissä tietyllä hinnalla. [23, s. 17]

Ketterät menetelmät ovat luonteeltaan mukautuvia eli joustavia. Perinteiset menetelmät taas noudattavat enemmän ennalta sovittua polkua, jossa edetään aina ennalta arvattavasti. Perinteisissä menetelmissä useimmiten tavoitteet määritellään heti projektin alussa. Tällöin voidaan sanoa, että ne käytännössä nojautuvat täysin huolelliseen vaatimusmäärittelyyn ja suunnitteluun eikä muutoksia voi tulla kesken prosessin. Ketterillä menetelmillä puolestaan ei ole loppuun tehtyä suunnitelmaa. Kaikkia ominaisuuksia ei siis ole määritelty alussa. Ne onkin kehitetty niin, että ne mahdollistavat joustavuuden dynaamisen ympäristön keskellä. [24, s. 70] Projektin vaatimukset voivat siis muuttua useasti projektin aikana, mutta sen ei tulisi aiheuttaa ongelmia prosessin loppuun viemisessä.

Ketterässä ohjelmistokehityksessä yhteistyö ja ihmiset ovat yksi tärkeimmistä palasista koko prosessissa, kun taas perinteinen tukeutuu enemmän prosesseihin ja työkaluihin. Kuitenkaan ei voida sanoa, että perinteisissä menetelmissä yhteistyö ja ihmiset eivät ole tärkeitä vaan pikemminkin, että nimenomaan ketterässä ne korostuvat vielä enemmän. Yhteistyöllä viitataan niin asiakkaisiin kuin tiimin jäseniin. Ketterissä menetelmissä asiakkaat ovat hyvin isossa roolissa, sillä he ovat aktiivisesti tuotteen kehittämisprosessissa mukana alusta loppuun saakka. Perinteisissä menetelmissä asiakkaiden rooli on huomattavasti pienempi, koska tuotetta voidaan käytännössä kehittää alusta loppuun vaatimusten perusteella. Sen lisäksi yleensä inkrementtejä ei tule yhtä enempää, joten asiakasta ei tavallaan edes tarvita alkumäärittelyiden jälkeen. [25, s. 4757] Perinteiset menetelmät sopivat isoille tiimeille ja projekteille, kun taas ketterät pienille. [3, s. 38]

Perinteisissä menetelmissä kehittäjät ovat orientoituneita suunnitelmalle eikä heiltä vaadita muuta kuin perusosaamista. Yleisesti perinteisissä menetelmissä kehittäjät

sidotaan maantieteellisesti johonkin tiettyyn paikkaan, kun taas ketterissä menetelmissä kehittäjät voivat ongelmitta työskennellä hajautetusti ympäri maailman. Tämän lisäksi ketterät kehittäjät ovat yleensä ovat omistautuneita työlleen, ja heidän taitonsa ovat korkeaa luokkaa. [24, s. 72]

Inkrementaali on tuttu käsite varsinkin ketterille menetelmille. Sillä tarkoitetaan uutta versiota tuotteesta. Niitä syntyy menetelmissä, joissa tehdään jatkuvasti iteraatioita, kuten esimerkiksi Scrum-viitekehyksessä. Jokainen iteraatio on tavallaan yksi pieni projekti, jonka lopputuloksena saadaan uusi inkrementti tuotteesta. Perinteiset menetelmät, kuten tavalliset vesiputousmallit koostuvat periaatteessa yhdestä iteraatiosta. Tällöin iteraatio kestää ajallisesti yleensä huomattavasti pidemmän ajan kuin ketterissä menetelmissä. Siten ketterät menetelmät tuottavat asiakkaalle jotain konkreettista huomattavasti nopeammin kuin perinteiset menetelmät. [25, s. 4757] Iteraatioihin liittyy myös testaaminen, joka suoritetaan yleensä perinteisissä menetelmissä koodin valmistuttua. Ketterissä tehdään lähes samalla tavalla, mutta iteraatioita on useampi, joten testit tehdään joka iteraatiolla. [24, s. 72]

Perinteiset menetelmät pitää sisällään paljon sääntöjä ja laajan dokumentaation, joka tehdään käytännössä vaatimusten perusteella. Nämä pikkuiset asiat kumminkin tekevät perinteisistä menetelmistä paremman vaihtoehdon varsinkin turvallisuuskriittisille tuotteille. [25, s. 570] Ketterissä menetelmissä puolestaan dokumentaatio yleensä laaditaan tuotteen kehityksen loputtua eli viimeisessä vaiheessa.

## 6.2 Suosio

Verkkosivulla ”Stack Overflow” yli 57 000 kehittäjää on vastannut kyselyyn siitä, mitä menetelmää he suosivat. Tilastojen mukaan vuonna 2018 lähes 86 % heistä suosi ketteriä ohjelmistokehitysmenetelmiä ja niistä suosituin oli Scrum 63 prosentilla. Perinteisiä ohjelmistokehityksen menetelmiä käytti vain noin 15 %. [27] Tilastot kuitenkin vaihtelevat hieman, sillä Project Management Institute suoritti 2017 tutkimuksen, johon osallistui lähes 4000 ohjelmistoalan ammattilaista. Tutkimuksen täytyi antaa arvio, kuinka monta prosenttia viimeisen 12 kuukauden aikana suoritetuista projekteista oli suoritettu seuraavilla menetelmillä: ketterät, vesiputousmalli, hybridi tai muu. Tulokseksi saatiin: 21 % ketteriä, 37 % vesiputousmalli, 20 % hybridi ja 23 % muita menetelmiä. [28, s. 19]

## 6.3 Migraatio

Ketterien menetelmien käyttöön ottaminen ei ole vain uusien työkalujen ja nimien muuttamista ketteräksi vaan vaatii organisaatiolta töitä. Organisaation tavat saattavat muuttua paljon, ja tätä kautta ne vaikuttavat yksilöiden jokapäiväisiin toimintoihin. Kaikkia ei kuitenkaan tarvitse osata heti vaan matkan varrella taidot kehittyvät, ja näin voidaan vastaanottaa uusia muutoksia. Ketterien menetelmien ymmärtäminen vaatii organisaatiolta paljon sitoutumista, jotta niistä voidaan hyötyä. [29] Aiemmin esitettyjen aineistojen pohjalta tiedetään, että perinteiset ja ketterät menetelmät eroavat prosesseissa ja käytänteissä paljon toisistaa. Uudet toimintatavat haastavat koko organisaatiota ajattelemaan uudelleen. [30, s. 75]

Organisaatio kohtaa migraatiossa monia eri haasteita, kuten johtamiseen ja organisaatioon liittyviä haasteita. Organisaation sisäinen kulttuuri muuttuu radikaalisti, sillä yksilöiden käyttäytyminen ja toiminta muuttuu erilaisten arvojen myötä. Arvot muuttuvat komento ja kontrolli tyylistä yhteistyö ja johtamisen malliin. Tämä vaatii jokaiselta yksilöltä erilaisia ajatusmalleja, miten asiakkaiden ja tiimin kanssa toimitaan, sillä kommunikoinnin määrä kasvaa huomattavasti verrattuna perinteisiin menetelmiin. Näiden lisäksi johtajan roolin on yhä tärkeämmässä asemassa, koska hänen täytyy koordinoida haastavia prosesseja, jotka vaativat kattavaa ymmärrystä aiheesta. [30, s. 76]

Ihmisiin luottaminen on toinen haaste migraatiossa. Sillä kuten aiemmin osoitettiin, on yhteistyö on yksi tärkeimmistä asioista ketterissä menetelmissä. Yhteistyö taas noujautuu luottamiseen. Ketterissä menetelmissä pariohjelmointi, yhteinen oppiminen, työpajat ja yhdessä päätösten tekeminen saattaa olla liikaa joillekin. Tämä näkyy varsinkin silloin, kun yksilöt ovat tottuneet tekemään täysin itsenäisesti töitä. Perinteisissä menetelmissä projektinjohtaja yleensä päättää kaikki asiat, kun taas ketterissä menetelmissä kehittäjät ja asiakas tekevät suurimman osan päätöksistä. Tämä johtaa siihen, että jokaisen pitää luottaa projektin jäseniin ja asiakkaaseen. [30, s. 76]

Prosessiin liittyvät ongelmat ovat kolmas asia, joka tulee organisaatioiden huomioida. Ketterät menetelmät nojautuvat paljon oletuksiin, suunnitteluun sekä sen ymmärtämiseen, että asiat saattavat olla epävarmoja ja muuttua vielä. Yksi isoin haaste migraatiolle on ymmärtää, että ketterät menetelmät rakentuvat useiden iteraatioiden kautta. Kyseinen muutos aiheuttaa suuria muutoksia tekniikoihin, työkaluihin, työmenetelmiin, kommunikointiin, ongelmanratkaisuun ja ihmisten rooleihin. [30, s. 77]

Viimeinen ongelma ovat teknologiset ongelmat, sillä organisaatiot ovat oppineet luottamaan heidän nykyisiin menetelmiinsä ja työkaluihinsa. Tällöin uusien menetelmien ja työkalujen käyttöönottaminen saattaa aiheuttaa ongelmia organisaatiossa. Ketteriin menetelmiin siirtyessä organisaatioiden tulee pystyä investoimaan uusiin työkaluihin, jotka tukevat nopeaa iteratiivista kehitystä. Nopea iteratiivinen ohjelmistokehitys pitää sisällään eri menetelmiä, yksikkötestausta, versiointi-/kokoonpanohallintaa ja muita menetelmiä. Uudet työkalut eivät kuitenkaan itsessään tuo vielä arvoa organisaatiolle, vaan ihmisten pitää opetella käyttämään niitä oikein arvon luomiseksi. [30, s. 77]

## 6.4 Synteesi

Aiemmin esitettyjen tietojen perusteella voidaan todeta, että menetelmän valintaan vaikuttavat monet eri tekijät. Menetelmän valintaprosessissa tulisi huomioida projektin luonne ja organisaatiorakenne. Näin saadaan valittua mahdollisimman tehokas ja toimiva menetelmä, jotta projektin tavoitteet täyttyvät mahdollisimman tehokkaasti ja riskittömästi.

Tietoja yhdistelemällä havaitaan, että ketterät ohjelmistokehitysmenetelmät voivat toimia isoissakin projekteissa. Tämä ei kuitenkaan tapahdu kasvattamalla tiimikokoa, sillä suurten tiimien koordinointi on vaikeaa. Tämä puolestaan aiheuttaa ongelmia kommunikoinnissa ja motivaation ylläpidossa. Sen sijaan muodostetaan useita ketteriä tiimejä, jotka toimivat itsenäisinä soluina projektissa. Tällöin vaalitaan ketterien menetelmien sääntöjä, mutta mahdollistetaan yhtä suurien projektien toteuttaminen kuin perinteisillä menetelmillä.

Projektin luonne on tärkeä havaita heti aluksi tarkastelemalle sen käyttötarkoitusta. Esimerkiksi yleensä turvallisuuskriittisille tuotteille vaatimukset ovat selvillä. Tällöin on järkevämpää käyttää lineaarisia ohjelmistokehityksen menetelmiä, koska tiedetään, etteivät vaatimukset muutu projektin aikana. Jos taas havaitaan, että projektin vaatimukset eivät ole täysin selvillä tai että ne voivat muuttua projektin aikana, on huomattavasti tehokkaampaa käyttää ketteriä menetelmiä, jotka sopivat dynaamiseen ympäristöön.

Organisaation tietämys eri menetelmistä on iso haaste, sillä menetelmien sujuva käyttäminen vaatii paljon resursseja ja omistautumista. Aineiston perusteella voidaan sanoa, että organisaatioiden kannattaa opetella ketterien menetelmien käyttöä, koska tänä päivänä kehitystyö on dynaamista ja enemmän asiakaslähtöistä. Sen lisäksi asiakas saa nopeammin ensimmäisen toimivan version heidän käyttöönsä, ja näin he pystyvät edelleen parantamaan tuotetta näköisekseen. Vahva suhde asiakkaan ja

kehittäjien välillä luo avoimen ilmapiirin, jossa mahdolliset ongelmatilanteet tulevat hyvin esille ja jossa tavoitteisiin päästään tehokkaasti.

Yhdistämällä aineistojen tietoja [6–30] toisiinsa ja tekemällä omia johtopäätöksiä voidaan muodostaa taulukkojen 1 ja 2 mukaiset yhteenvedot. Omat johtopäätökset perustuvat tutkielmassa esitettyyn aineistoon ja niiden tulkintaan. Taulukossa esitetyt tiedot eivät ole siis absoluuttinen totuus aiheesta vaan nimenomaan perustuvat omiin tulkintoihin aineiston pohjalta.

***Taulukko 1. Menetelmien vertailu***

| | **Vesiputousmalli** | **Spiraalimalli** | **V-malli** | **Extreme Programming** | **Scrum** |
|---|---|---|---|---|---|
| **Rakenne** | Lineaarinen | Iteratiivinen | Lineaarinen | Iteratiivinen/ Inkrementaalinen | Iteratiivinen/ Inkrementaalinen |
| **Luonne** | Jäykkä | Joustava | Jäykkä | Joustava | Joustava |
| **Kustannukset** | Matala | Korkea | Korkea | Korkea | Korkea |
| **Kustannusten arviointi** | Helppo | Vaikea | Helppo | Vaikea | Vaikea |
| **Onnistumisaste** | Matala | Korkea | Korkea | Korkea | Korkea |
| **Kehitysaika** | Pitkä | Lyhyt | Pitkä | Lyhyt | Lyhyt |
| **Testaus** | Lopussa | Jokaisessa iteraatiossa | Lopussa | Jokaisessa iteraatiossa | Jokaisessa iteraatiossa |
| **Vaikeusaste** | Helppo | Keskitaso | Keskitaso | Vaikea | Vaikea |
| **Asiakkaan osallistuminen** | Vain alussa | Korkea | Vain alussa | Korkea | Korkea |

***Taulukko 2. Perinteisen ja ketterän ohjelmistokehityksen vertailu***

| | **Ketterä ohjelmistokehitys** | **Perinteinen ohjelmistokehitys** |
|---|---|---|
| **Johtamistapa** | Johtajuus ja yhteistyö | Komento ja kontrolli |
| **Joustava** | Kyllä | Ei |
| **Projekti koko** | Pienet ja keskisuuret | Suuret |
| **Kehittäjät** | Joustavia ja omistautuneita | Orientoituneita suunnitelmalle |
| **Vaatimukset** | Ei tiedossa | Tiedossa projektin alussa |
| **Tiimikoko** | Pieni | Suuri |
| **Arkkitehtuuri** | Suunnitellaan muuttuvien vaatimusten mukaan | Suunnitellaan ennalta määrätyn mukaiseksi |
| **Testaus** | Jokaisessa iteraatiossa | Lopussa |
| **Asiakaan osallistuminen** | Korkea | Matala |
| **Uudelleen aloittamisen kustannukset** | Matala | Korkea |

Taulukosta 1 löytyy tutkielmassa käytettyjen menetelmien eroavaisuuksia toisiinsa. Lähes kaikki taulukon tiedot ovat esitetty ohjelmistokehitysmenetelmien luvuissa, mutta osittain kustannusten sekä onnistumis- ja vaikeusasteen käsittelyä ei löydy. Kustannusten osalta vesiputous- ja spiraalimallin kustannukset ovat esitetty aiemmin. Vesiputousmallin kustannukset olivat matalat sen jäykän rakenteen ja yksinkertaisen prosessin takia. Spiraalimallin kustannukset olivat puolestaan korkeat, koska spiraalia voidaan joutua kiertämään useita kertoja, ja mitä pidemmälle edettiin sitä kalliimmaksi se kävi. Muiden menetelmien osalta kustannukset perustuvat omiin johtopäätöksiin. V-

malli on vesiputousmallin kanssa lineaarinen, mutta monimutkaisempi sekä testaukseen panostetaan enemmän. Näin voidaan todeta, että sen kustannukset ovat korkeammat kuin vesiputousmallissa, koska työtä on enemmän. Extreme Programming ja Scrum ovat spiraalimallin kanssa iteratiivisia. Menetelmät ovat myös monimutkaisempia käyttää kuin spiraalimalli, sääntöjen ja käytänteiden takia. Näin ollen voidaan sanoa, että menetelmät ovat vähintäänkin yhtä kalliita käyttää kuin spiraalimalli.

Kustannusten arviointi perustuu menetelmien luonteeseen. Lineaariset menetelmät ovat jäykkiä rakenteita, missä vaatimukset pitää olla selvillä projektin alussa eikä muutoksia oteta vastaan. Joustavissa menetelmissä puolestaan voi tulla muutoksia pitkin projektia, mikä tekee kustannusten arvioinnista erittäin haastava. Tällöin voidaan sanoa, että kustannukset ovat jäykissä rakenteissa helppo arvioida, kun taas joustavassa vaikeaa. Onnistumisasteet perustuvat iteratiivisten mallien osalta niiden joustavuuteen, koska niitä voidaan parannella niin kauan kunnes asiakas on tyytyväinen. V-mallin korkea onnistumisaste voidaan perustella sen kattavien testauksien vuoksi. Kuitenkin tässä oletetaan, että V-malli sopii kyseisen projektin luonteeseen.

Menetelmien vaikeusasteet tavallaan jo perusteltiin aikaisemmin. Extreme Programming ja Scrum ovat vaikeusasteiltaan vaikeita, koska niiden prosessit sisältävät paljon sääntöjä ja käytänteitä. Spiraalimalli ja V-malli ovat keskivaikeita, koska ne ovat vesiputousmallista monimutkaisempia. Tällöin vertailun helpoin malli on vesiputousmalli, sen yksinkertaisuuden takia.

Taulukko 2 on rakennettu osittain taulukon 1 avulla. Siinä yleistetään joitakin asioita kuten se, että perinteinen ohjelmistokehitys ei ole joustava vaikkakin spiraalimalli on sitä. Tämä perustuu määräenemmistöön, sillä tutkielmassa käsiteltiin kolmea eri perinteistä menetelmää, ja kaksi kolmesta eivät olleet joustavia. Kaikki muut tiedot paitsi uudelleen aloittamisen kustannukset ovat esitetty aiemmin ”eroavaisuus” luvussa (luku 6.1). Uudelleen aloittamisen kustannukset voidaan perustella ketterän ohjelmistokehityksen osalta niiden joustavuuteen. Oletetaan, että projektiin tulee erittäin suuri muutos, mikä vaatii projektin uudelleen aloittamisen. Ketterien menetelmien tapauksessa tämä ei kuitenkaan vaadi puhtaalta pöydältä aloittamista, koska seuraavassa iteraatiossa voidaan tehdä vaadittavat muutostyöt. Tällaisia tilanteita varten ketteriä menetelmiä käytetään. Perinteiset ohjelmistokehitysmenetelmät puolestaan joudutaan suurella todennäköisyydellä aloittamaan täysin alusta, mikä johtaa vaatimusten ja arkkitehtuurin uudelleen kartoittamiseen. Näin ollen perinteisten menetelmien uudelleen aloittaminen on kalliimpaa kuin ketterien.

# 7. YHTEENVETO

Työn päätutkimuskysymys oli selvittää perinteisen ja ketterän ohjelmistokehityksen väliset erot ohjelmistotuotannossa. Toisena tutkimuskysymyksenä oli tutkia, miten organisaatio voi siirtyä perinteisestä ohjelmistokehityksestä ketterään.

Tulokseksi saatiin, että perinteiset ja ketterät menetelmät eroavat toisistaan paljon niin prosessi- kuin organisaatiotasolla. Prosessin suurimmat muutokset olivat, että ketterät menetelmät ovat iteratiivisia ja inkrementaalisia, kun taas perinteiset olivat lähinnä lineaarisia, joissa tuote tuotetaan yhden iteraation kautta. Toisin sanoen perinteiset menetelmät ovat jäykkiä rakenteita, jossa muutoksille ei ole tilaa, kun taas ketterät menetelmät ovat joustavia, muutoksia tukevia rakenteita. Ketterät menetelmät ovat näin yleensä parempi lähestymistapa tänä päivänä, koska elämme hyvin dynaamisessa ympäristössä. Toisena keskeisenä asiana pidetään ilmapiirin muutosta. Ketterät menetelmät ovat johtamis- ja yhteistyölähtöisiä menetelmiä, joissa jatkuva yhteydenpito asiakkaan kanssa on elintärkeää projektin onnistumiseksi. Perinteisissä menetelmissä asiakkaat ovat myös kehityksessä mukana, mutta vain alussa määrittelemässä tavoitteita ja lopussa tarkastamassa.

Migraatio perinteisestä ketterään ohjelmistokehitykseen vaati organisaatiolta paljon muutoksia. Perinteiset ja ketterät menetelmät eroavat toisistaan radikaalisti, mikä vaikuttaa koko organisaation toimintaan. Muutosprosessi vaati organisaatiolta omistautumista ja resursseja, jotta ketterät menetelmät voidaan käyttöönottaa tehokkaasti. Aluksi ei kuitenkaan ole tärkeintä osata kaikkia menetelmiä tai työkaluja vaan lähinnä kerryttää osaamista ajan myötä. Suurimpana ongelmana tutkijat pitivät vakiintunutta organisaatiotyyliä, jossa luotetaan vanhoihin menetelmiin ja työkaluihin ja jossa ei uskalleta ottaa muutosta vastaan.

Tilastojen mukaan ketterät ohjelmistokehitysmenetelmät ovat kasvaneet ja korvanneet perinteisen ohjelmistokehityksen menetelmiä. Näin voidaan todeta, että ketterien menetelmien ymmärtäminen ja osaaminen on tärkeää nykypäivän ohjelmistoyrityksille. Kuitenkin tärkeintä on huomioida projektin luonne ja organisaation osaamisentaso, jotta tuote saadaan kehitettyä mahdollisimman riskittömästä ja tehokkaasti asiakkaalle.

# LÄHTEET


[1] A. J. Wiebe, & C. W. Chan. (2012). Ontology driven software engineering. Paper presented at the 2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), 1-4. doi:10.1109/CCECE.2012.6334938

[2] Arikpo, I., & Osofisan, A. (2010). Introducing agile software development into an organization: the role of the customer. Computing and Information Systems Journal, 14(2), 28–.

[3] Awad, M. A. (2005). A comparison between agile and traditional software development methodologies. University of Western Australia, 30.

[4] Naur, P., Randell, B., Bauer, F. L., & NATO Science Committee. (1969). Software engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Brussels: Scientific Affairs Division, NATO.

[5] Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. Proc. IEEE WESTCON, Los Angeles, 1--9.

[6] C. Larman, & V. R. Basili. (2003). Iterative and incremental developments. a brief history doi:10.1109/MC.2003.1204375

[7] Cohen, D., Lindvall, M., & Costa, P. (2003). Agile software development. DACS SOAR Report, 11, 2003.

[8] Cimolini, P., & Cannell, K. (2012). Agile Oracle Application Express (1st ed. 2012.). Apress. https://doi.org/10.1007/978-1-4302-3760-0

[9] Alshamrani, A., & Bahattab, A. (2015). A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model. International Journal of Computer Science Issues (IJCSI), 12(1), 106.

[10] Kannan, V., Jhajharia, S., & Verma, S. (2014). Agile vs waterfall: A Comparative Analysis.

[11] Ragunath, P. K., Velmourougan, S., Davachelvan, P., Kayalvizhi, S., & Ravimohan, R. (2010). Evolving a new model (SDLC Model-2010) for software development life cycle (SDLC). International Journal of Computer Science and Network Security, 10(1), 112-119.

[12] Amlani, R. D. (2012). Advantages and limitations of different SDLC models. International Journal of Computer Applications & Information Technology, 1(3), 6-11.

[13] Kumar Pal, S. (2018). Software Engineering | Spiral Model. Saatavissa (viitattu 20.10.2020): https://www.geeksforgeeks.org/software-engineering-spiral-model/?ref=lbp

[14] Mathur, S., & Malik, S. (2010). Advancements in the V-Model. International Journal of Computer Applications, 1(12), 29-34.

[15] Kumar, D. (2019). Software Engineering | Spiral Model. Saatavissa (viitattu: 21.10.2020): https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/

[16] Hijazi, H., Khdour, T., & Alarabeyyat, A. (2012). A review of risk management in different software development methodologies. International Journal of Computer Applications, 45(7), 8–12.

[17] Andres, C., & Beck, K. (2004). Extreme programming explained: embrace change. In Extreme programming explained: embrace change. Addison-Wesley Professional.

[18] Wells, D. (2009). Extreme Programming. Saatavissa (viitattu: 21.10.2020): http://www.extremeprogramming.org/

[19] Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile software development methods: Review and analysis. VTT Technical Research Centre of Finland. VTT Publications, No. 478 http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf

[20] R. Juric. (2000). Extreme programming and its development practices. Paper presented at the - ITI 2000. Proceedings of the 22nd International Conference on Information Technology Interfaces (Cat. no.00EX411), 97-104.

[21] Schwaber, K., & Sutherland, J. (2017). Scrum-opas. Saatavissa (viitattu: 26.10.2020): https://scrumwell.files.wordpress.com/2018/03/2017-scrum-guide-fi-v1-02.pdf

[22] Scrum. (n.d). What is Scrum? Saatavissa (viitattu: 3.12.2020): https://www.scrum.org/resources/what-is-scrum

[23] Rehman, I., Ullah, S., Rauf, A., & Shahid, A. (2010). Scope management in agile versus traditional software development methods. 1–6. https://doi.org/10.1145/1890810.1890820

[24] Stoica, M., Mircea, M., & Ghilic-Micu, B. (2013). Software development: Agile vs. traditional. Informatica Economica, 17, 64-76. doi:10.12948/issn14531305/17.4.2013.06

[25] Aitken, A., & Ilango, V. (2013, January). A comparative analysis of traditional software engineering and agile software development. In 2013 46th Hawaii International Conference on System Sciences (pp. 4751-4760). IEEE.

[26] Keshta, N., & Morgan, Y. (2017, October). Comparison between traditional plan-based and agile software processes according to team size & project domain (A systematic literature review). In 2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON) (pp. 567-575). IEEE.

[27] Stack Overflow. (2018). Developer Survey Results. Saatavissa (viitattu: 27.10.2020): https://insights.stackoverflow.com/survey/2018#development-practices

[28] Project Management Institute. (2017). Success Rates Rise. Saatavissa (viitattu 27.10.2020): https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2017.pdf

[29] Stober, T., & Hansmann, U. (2010). Agile Software Development Best Practices for Large Software Development Projects (1st ed. 2010.). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-70832-2

[30] Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, *48*(5), 72-78.