Vaatimusmäärittelyn merkitys palveluna ostetun tietojärjestelmän kehittämisessä

Sanna Harju

Opinnäytetyö
Tietojärjestelmäosaamisen koulutusohjelma
2021

Tiivistelmä

Haaga-Helia
ammattikorkeakoulu Oy

Tekijä(t)
Sanna Harju

Koulutusohjelma
Tietojärjestelmäosaamisen koulutusohjelma, YAMK

Raportin/Opinnäytetyön nimi
Vaatimusmäärittelyn merkitys palveluna ostetun tietojärjestelmän kehittämisessä

Sivu- ja liitesivumäärä
81+1

Tämän opinnäytetyön tavoitteena oli tutkia vaatimusmäärittelyn merkitystä palveluna ostetun tietojärjestelmän kehittämisessä yhdessä Suomen suurimmista finanssialan yrityksistä. Opinnäytetyön idea lähti omakohtaisista kokemuksista ja tarpeista kehittää vaatimusmäärittelyä ketterässä ohjelmistokehittämisessä. Työskentelen vaatimusmäärittelijänä isossa uudistamishankkeessa, jossa on tarkoitus muutaman vuoden aikana uusia yrityksen varallisuudenhoidon alueen järjestelmät ja palvelumalli. Koronavuosi 2020-2021 toi kuitenkin opinnäytetyön kirjoittamiseen ja valmistumiseen omat haasteensa, kuten myös ketterään ohjelmistokehitykseen ja oli jopa vaarana, että tämä työ ei koskaan valmistu.

Vaatimusmäärittelyä joskus vähätellään ja sen merkitystä aliarvioidaan, joka näkyy usein projektien hidastumisena tai suoranaisina epäonnistumisina. Tämä motivoi minua itseäni tutustumaan paremmin tähän alueeseen. Opinnäytetyön teoriaosuudessa käynkin ensin läpi ohjelmistokehitystä ja sen elinkaarimallia ja eri kehittämismenetelmiä vesiputousmallista erilaisiin ketteriin menetelmiin. Tämän jälkeen on vuorossa vaatimusmäärittelyn merkitys ohjelmistotuotannossa. Olen tutustunut vaatimuksen määritelmään ja vaatimusten luokitteluun ja erilaisin vaatimustyyppeihin. Käyn myös läpi vaatimusmäärittelyn osa-alueet ja erityispiirteet ketterässä ohjelmistokehityksessä.

Itse tutkimus noudattaa laadullisen tapaustutkimuksen periaatteita ja muodostui lomakehaastattelusta, jossa muutamalta hankkeen jäseneltä kysyttiin vaatimusmäärittelyn merkityksestä ja ketterästä ohjelmistokehityksestä. Kyselyn tulokset tukevat olettamusta, että vaatimusmärittelyn merkitys on edelleen merkittävä myös ketterässä ohjelmistokehityksessä.

Asiasanat
vaatimusmäärittely, vaatimus, ohjelmistokehittäminen, ketterä, tietojärjestelmän kehittäminen

Sisällys

1 Johdanto

Opinnäytetyössäni aion tutkia, millainen merkitys vaatimusmäärittelyllä on palveluna ostetun tietojärjestelmän kehittämisessä vai tarvitaanko sitä lainkaan.

Idea tähän työhön syntyi käytännön kokemuksen kautta. Olen parin vuoden ajan työskennellyt mittavassa tietojärjestelmän uudistamishankkeessa eräässä suomalaisessa finanssitavaratalossa, ja kokenut lähes päivittäin tuskaa vaatimismäärittelyn onnistumisesta. Olen alkanut miettiä, miten voisimme jo alkumetreillä taklata epäonnistumisen pelon ja huolen sitä, olemmeko jättäneet jotain huomioimatta vaatimusmäärittelyn saralla. Miten suhtautua ajatukseen, että jotain itsestään selvää on jäänyt pois vaatimuslistalta ja miten voisimme muuttaa juurtuneita toimintatapoja ja -malleja ketterämpään suuntaan. Mainittakoon tässä, että hankkeessa käytetään tietojärjestelmän hankkijan eli ostajan osalta ketteriä toimintatapoja, mutta toimittajan osalta ollaan enemmän perinteisessä vesiputousmallissa.

Tietojärjestelmä on tiedoista, tietoja käsittelevistä ihmisistä, tietojenkäsittelylaitteista, tiedonsiirtolaitteista, ohjelmistoista ja tietojen käsittelysäännöistä koostuva järjestelmä, jonka tarkoituksena on tietojen käsittelyn avulla tehostaa tai helpottaa jotain toimintaa tai tehdä se ylipäätään mahdolliseksi. (Wikipedia f 2021)

1.1 Taustat

Opinnäytetyön kohteena olevassa finanssialan yrityksessä (myöh. yritys) on jo muutama vuosi sitten alettu miettiä varallisuudenhoidon alueen kokonaisuudistusta aina myynnistä tietojärjestelmiin ja liiketoimintaprosesseihin. Ensimmäiset askeleet otettiin vuonna 2016 rahoitusvälineiden markkinoita sääntelevän MiFID II -direktiivin ja MiFIR-asetuksen vaatimien muutosten myötä. Uudistamishanketta varten perustettiin työryhmä, jonka yksi tavoite oli selvittää myynnin uudistamisen kokonaisratkaisua. Aika nopeasti päädyttiin kuitenkin siihen, että koko operatiivinen toimintamalli vaatii uudistamista. Yksi syy tähän oli se, että oli haastavaa sovittaa yhteen lyhyen aikavälin regulaatiovaateet sekä keskipitkän ja pitkän aikavälin kilpailukyvyn parantaminen. Keskeiseksi kysymykseksi nousi ratkaisun vaiheistaminen niin, että varmistetaan paras lopputulos pidemmällä aikavälillä ja samalla vältetään moninkertaisten investointien tekeminen vanhaan operatiiviseen malliin.

Uudistamistyöryhmässä tekemistä selvityksistä havaittiin myös, että varallisuudenhoidon järjestelmäkokonaisuus on heikossa kunnossa, eikä kykene nykyisellään vastaamaan digitalisaation vaatimuksiin. Nykyinen monimutkainen ja hajanainen järjestelmäarkkitehtuuri

rakentuu eri tuoteyhtiöiden vuosikymmenien aikana rakennetuista prosesseista ja järjestelmistä. Usein ratkaisuja ei ole suunniteltu siihen käyttöön missä ne ovat ja järjestelmäketjut ja integraatiot ovat monimutkaisia. Lyhykäisyydessään tietojärjestelmien puutelistan voisi tiivistää näihin kolmeen kohtaan:

- Toimintavarmuus on heikkoa ja usein manuaalisten ratkaisujen varassa.
- Vanhentuneet teknologiat ja osaamisen niukkuus tekevät kehittämisestä haastavaa ja riskialtista
- Regulaatiovaatimuksiin vastaaminen ja liiketoimintapalveluiden kehittäminen on hidasta, kallista ja heikosti tuottavaa

Varallisuudenhoito kävi siis läpi intensiivisen harjoituksen analysoidessaan nykyisen toimintamallin, palveluprosessit ja tietojärjestelmät. Tähän analyysiin perustuen yritys muodosti tulevaisuuden vision ja päätti aloittaa varallisuudenhoidon alueen liiketoiminnan muutosohjelman tulevaisuuden kasvua ja digitaalista liiketoimintaa tukevan mallin luomiseksi. Tavoitteena on uudistaa liiketoimintamalli, mahdollistaa nopea ja tehokas palvelukehitys, minimoida sääntelykustannukset, vähentää toimintakuluja ja uudistaa ja yksinkertaistaa IT-järjestelmä.

Uudistamistyöryhmässä mietittiin paljon erilaisia skenaarioita ja työn pohjalta tunnistettiin neljä skenaariota operatiivisen mallin uudistamiselle sekä MiFID II vaatimuksiin vastaamiselle. Skenaariot olivat:

1. Ei kokonaisvaltaista operatiivisen mallin uudistamista
2. Täsmäratkaisuja usealta toimittajalta
3. Kokonaisratkaisu yhdeltä toimijalta ilman BPO:ta
4. Standardoitu ja automatisoitu BPO-kokonaisratkaisu

Uudistamistyöryhmä päätyi esittämään kokonaisvaltaista ratkaisua, koska se tukisi asiakasnäkökulmasta tärkeimpiä prosesseja (asiakkaan haltuunotto, toimeksiannot ja raportointi) yksinkertaistamalla sirpaleista järjestelmäkokonaisuutta ja parantaisi asiakaskokemusta ohjaamalla yritystä toimimaan tehokkaammilla, standardoiduilla prosesseilla. Lisäksi toimittajan ehdottama muutos korvaisi suoraan noin 10 sovellusta, jotka aiheuttavat suurimman osan nykyisistä käyttökustannuksista.

Uudistamisen kärkenä ovat uudet ja asiakaslähtöiset palvelumallit, jotka on helppo kytkeä osaksi kaikkia asiakaskohtaamisia.

- Uudet digitaaliset palvelumallit asiakkaille.
- Uudistettu asiakaslähtöinen ja digitaalinen liiketoimintamalli, toimintatavat ja prosessit.
- Järjestelmäalusta, joka mahdollistaa jatkuvat palvelu-uudistukset.

Uudistamishankkeessa oli tarkoitus yhdellä tietojärjestelmällä, alustalla tai palvelulla korvata tukku vanhoja sovelluksia.

1.2 Työn tarkoitus ja tavoitteet

Ilman toimivia tietojärjestelmiä yrityksillä ei olisi nyky-yhteiskunnassa minkäänlaisia toimintaedellytyksiä ja onkin havaittavissa kuinka haavoittuvaisia yritykset ovat, jos tietojärjestelmät eivät toimi kunnolla. Samoin myös tietojärjestelmäuudistukset ovat viime vuosina olleet tapetilla juuri siitä syystä, että toimitukset ovat viivästyneet puutteellisten tai muuttuneiden vaatimusten vuoksi. Yritykselle, sekä toimittajalle että tietojärjestelmän hankkijalle, saattaa tällaisesta uutisoinnista kertyä mittavia taloudellisia tappioita ja mainehaittaa.

Tässä opinnäytetyössä käsittelen tietojärjestelmän uudistamishanketta vaatimusmäärittelyn näkökulmasta. Teoriaosuudessa pyrin selvittämään ohjelmistotuotantoa ja ohjelmistotuotannon menetelmiä, sekä vaatimusmäärittelyn teoriaa ja mikä on vaatimusmäärittelyn merkitys ohjelmistokehityksessä.

1.3 Tutkimuskysymykset

Tämän opinnäytetyön tutkimusongelma voidaan muotoilla seuraavanlaisesti: ”Mikä on vaatimusmäärittelyn merkitys palveluna ostetun tietojärjestelmän kehittämisessä?” Tutkittava aihe voidaan jakaa kahteen tutkimuskysymykseen: ”Tarvitaanko vaatimusmäärittelyä palveluna ostetun ohjelmiston kehittämisessä” ja ”Miten vaatimusmäärittely toteutuu palveluna ostetun tietojärjestelmän kehittämisessä”.

Tutkimusta varten olen teoriaosuudessa perehtynyt ohjelmistokehitysmenetelmiin ja vaatimusmäärittelyyn.

1.4 Rajaukset

Tässä opinnäytetyössä kuvataan vain vaatimusmäärittelyä ja sen merkitystä tietojärjestelmän kehittämisessä. Itse tietojärjestelmän hankinta, kehittäminen, testaaminen ja käyttöönottoon liittyvät vaiheet rajataan opinnäytetyön ulkopuolelle. Ohjelmistotuotantoa ja kehittämismenetelmiä on tutkittu vain teoriassa ja lähinnä vaatimusmäärittelyn näkökulmasta.

2 Ohjelmistotuotanto

Suurimpia muutoksia viimeisen viidenkymmenen vuoden aikana on ohjelmistotekniikan hyödyntäminen eri muodoissa. Ensimmäiset ohjelmoitavat tietokoneet kehitettiin 1940-luvun lopulla. Tällöin käynnistettiin myös ensimmäiset tietojärjestelmä- ja ohjelmistokehityshankkeet. Automaattinen tietojenkäsittely tuli osaksi jokapäiväistä byrokratiaa 1960-70 -luvulla, tietokonepelit yleistyivät 1980-luvulla ja nykyään ohjelmistojen käyttö osana erilaisia laitteita on usein välttämätöntä. Monet meistä käyttävät ohjelmistoja huomaamattaan, esimerkiksi vaihtaessaan televisiokanavaa kaukosäätimellä tai tankatessaan autoa. Ohjelmistotekniikkaa tarvitaan myös ohjaamaan kokonaisia järjestelmiä, kuten esimerkiksi omakotitalon sähkömittareita, pullonpalautusautomaatteja, metsäkoneita ja ydinvoimaloita. Siksi ohjelmistotekniikka on tärkeä menestystekijä myös muilla teollisuuden aloilla kuin ohjelmistoteollisuudessa. (Haikala & Mikkonen 2011,11.)

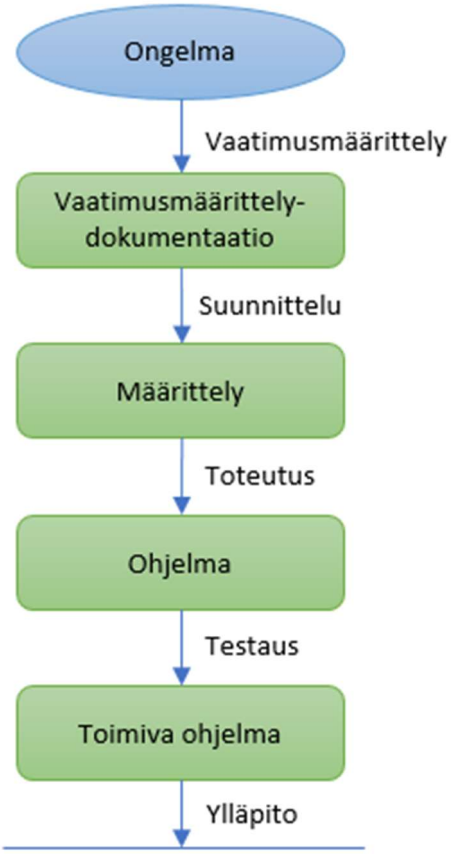
Ohjelmistot ovat myös muuttaneet tapaa, jolla suhtaudumme itseemme. Shakkipeliä on aina pidetty ihmisen älykkyyden huippuna, mutta tämä käsitys muuttui rajusti, kun tietokone (ja ohjelmisto) kukisti hallitsevan shakin maailmanmestarin. Nykyään tavallisella tietokoneella toimiva ohjelmisto on tehokkaampi kuin mikään inhimillinen shakinpelaaja. Ohjelmistot ja Internet ovat muun muassa muuttaneet tapaamme kommunikoida, olla yhteydessä toisiimme ja tehdä ostoksia.

Ohjelmistoteollisuudesta on tullut yksi maailman tärkeimmistä tekniikan aloista. Nykyään tuottamistamme ohjelmistoista on nopeasti tulossa suurimmassa osassa maailmaa henkisen omaisuuden ruumiillistuma. Yksinkertaisesti sanottuna nykymaailmamme riippuu ohjelmistoista. Viimeisten 40-50 vuoden aikana ohjelmistotuotannon saralla on kehitetty erilaisia ohjelmistokehitysmenetelmiä - prosessikehyksiä - joita käytetään työn jäsentämiseen, ohjaamiseen ja hallintaan. Ensimmäisessä vaiheessa menetelmänä oli niin sanottu "cut-and-try" -menetelmä eli ohjelmistoja kehitettiin yrityksen ja erehdyksen kautta. Tästä menetelmästä on käytetty myös nimitystä ”Code and fix” eli koodattiin ohjelma ja katsottiin, toimiiko se. Suurelta osin se toimikin, kunnes ohjelmistoista tuli niin suuria ja monimutkaisia että piti kehittää paremmin jäsenneltyjä ja hallitumpia menetelmiä ja malleja. (Leffingwell. 2011.) Lähdekirjallisuudessa näistä menetelmistä tai malleista käytetään erilaisia nimityksiä; puhutaan ohjelmiston elinkaarimallista, ohjelmistokehityksen vaihejakomallista, projekti- tai prosessimalleista tai yksinkertaisesti ohjelmiston toteutusmenetelmistä. Wikipedian (2021) mukaan ohjelmiston elinkaarella tarkoitetaan aikaa, joka kuluu ohjelmiston kehittämisen aloittamisesta sen poistamiseen käytöstä. Vaihejakomallilla tarkoitetaan tapaa, jolla kehitystyö tai koko elinkaari jaetaan vaiheisiin. Seuraavissa kappaleissa tutustumme tarkemmin ohjelmistotuotantoon ja ohjelmistokehityksen menetelmiin.

2.1 Ohjelmistotuotannon elinkaari

Taloa rakennettaessa rakentaja ei aloita rakentamista tiilien kasaamisella. Pikemminkin ensin analysoidaan asiakkaan vaatimukset ja mahdollisuudet ottaen huomioon sellaiset tekijät kuin perheen rakenne, harrastukset, talous ja niin edelleen, jonka jälkeen arkkitehti huomioi nämä tekijät taloa suunnitellessaan. Varsinainen rakentaminen aloitetaan vasta, kun suunnittelusta on sovittu. On tarkoituksenmukaista toimia samalla tavalla ohjelmistoja rakennettaessa. Ensinnäkin analysoidaan ratkaistava ongelma ja kuvataan vaatimukset hyvin tarkasti. Sitten näiden vaatimusten perusteella tehdään suunnittelu. Lopuksi aloitetaan rakennusprosessi, ts. ratkaisun varsinainen ohjelmointi. Ohjelmistojen elinkaaren vaiheet on esitetty kuvassa 1. (van Vliet 2008.)

van Vlietin (2008) kuvassa 1 esitetty prosessimalli on melko yksinkertainen. Todellisuudessa asiat ovat yleensä monimutkaisempia. Esimerkiksi suunnitteluvaihe jaetaan usein arkkitehtoniseksi suunnitteluvaiheeksi ja yksityiskohtaiseksi suunnitteluvaiheeksi, ja usein erotetaan myös eri testausvaiheet. Peruselementit pysyvät kuitenkin kuvan 1 mukaisina. Nämä vaiheet on läpikäytävä jokaisessa projektissa, mutta projektityypistä ja työympäristöstä riippuen saatetaan tarvita yksityiskohtaisempi suunnitelma.



Kuva 1, Ohjelmistojen elinkaari van Vlietin (2008) mukaan

Kuvassa 1 vaiheet on kuvattu peräkkäin. Van Vlietin (2008) mukaan joissakin ohjelmistoprojekteissa näitä toimintoja ei välttämättä eroteta toisistaan niin tarkasti kuin tässä on kuvattu. Toiminnot voivat mennä ja yleensä menevätkin päällekkäin. Esimerkiksi on täysin mahdollista aloittaa järjestelmän yhden osan toteutus, vaikka joitain muita osia ei ole vielä täysin suunniteltu. Havaitut virheet tai muuttuneet vaatimukset mahdollistavat paluun aikaisempiin vaiheisiin. Onkin parempi ajatella näitä vaiheita sarjana työnkulkuja. Ensin suurin osa resursseista käytetään vaatimusten suunnitteluun ja myöhemmin työ siirtyy toteutuksen ja testauksen työnkulkuihin.

Subramanianin (2015) mukaan elinkaaren vaiheet ovat:

1. Vaatimusten analyysivaihe
2. Suunnitteluvaihe
3. Koodaus- ja testausvaihe
4. Toteutusvaihe
5. Ylläpitovaihe

SWEBOK:in (2014) mukaan ohjelmistotuotanto jakautuu seuraaviin osa-alueisiin:

- Software requirements, vaatimusten määrittely
- Software design, suunnittelu
- Software construction, ohjelmiston toteutus
- Software testing, testaus
- Software maintenance, ylläpito

2.1.1 Vaatimusmäärittely

Wikipedian (2021) mukaan tämän vaiheen tarkoituksena on selvittää ne ohjelmistotuotteelle asetetut tavoitteet, mitkä valmiin järjestelmän tulisi täyttää. Vaatimusanalyysin tarkoituksena ei ole ottaa millään tavoin kantaa siihen, *miten* nämä tavoitteet saavutetaan. Näin toimitaan siksi, että on edullisinta siirtää toteutusmenetelmiin liittyvät päätökset mahdollisimman myöhäiseen vaiheeseen. Tyypillisesti ohjelmiston tekninen toiminta muuttuu sen elinkaaren myötä monistakin syistä, ja muutosten tekeminen on sitä kalliimpaa taloudellisesti, mitä aikaisempaan vaiheeseen ne joudutaan kohdistamaan. Toisinaan vaatimuksiin voi toki kuulua *rajoituksia*, jotka esimerkiksi sitovat toteutuksen tiettyyn ohjelmointikieleen esimerkiksi asiakkaan erityistarpeiden vuoksi.

SWEBOK määrittelee vaatimusmäärittelyn seuraavasti: *Software requirements* tarkoittaa ohjelmistolle asetettuja vaatimuksia, eli sitä miten rakennettavan ohjelmiston tulisi loppukäyttäjän tai tilaajan mielestä toimia.

Subramanianin. (2015) mukaan vaatimusanalyysi on ohjelmistokehityksen ensimmäinen vaihe, jossa asiakas ja liiketoiminta-analyytikot ovat vuorovaikutuksessa keskenään ja dokumentoivat ohjelmiston lopputuloksen vaatimukset. Asiakkaat kertovat vaatimuksensa siitä, miten lopputuotteen tulisi toimia. Vaatimukset muodostavat minkä tahansa projektin tai tuotteen lähtökohdan. Vaatimuksiin sisältyvät ulkoasu (käyttöliittymävaatimukset), toiminnalliset vaatimukset ja ei-toiminnalliset vaatimukset. Van Vlietin (2008) mukaan ympäristön asettamat vaatimukset voivat sisältää laitteiston ja tukiohjelmiston tai kehitettävän järjestelmän mahdollisten käyttäjien määrän. Vaihtoehtoisesti vaatimusten analyysi voi johtaa tiettyihin rajoituksiin, jotka kohdistuvat hankittavaan laitteistoon tai organisaatioon, jossa järjestelmän on tarkoitus toimia.

Van Vliet (2008) mukaan yksi osa vaatimusten suunnittelusta on toteutettavuustutkimus. Toteutettavuustutkimuksen tarkoituksena on arvioida, onko ongelmaan olemassa ratkaisu, joka on sekä taloudellisesti että teknisesti toteutettavissa.

Mitä huolellisimpia olemme vaatimusten suunnitteluvaiheessa, sitä suurempi on mahdollisuus, että lopullinen järjestelmä vastaa odotuksia. Tätä varten asiaan osallistuvien henkilöiden (muun muassa asiakas, mahdolliset käyttäjät, suunnittelijat ja ohjelmoijat) on tehtävä tiivistä yhteistyötä. Näillä henkilöillä on usein hyvin erilainen tausta, mikä ei helpota viestintää. Tämän toiminnon lopputuloksena syntyy vaatimusmäärittely (eng. *requirement specification*). Vaatimusmäärittelyyn syvennyn tarkemmin kappaleessa 3.

2.1.2 Suunnittelu

Van Vlietin (2008) mukaan suunnitteluvaiheessa (eng. *design*) kehitetään koko järjestelmän malli, joka koodattuna jollekin ohjelmointikielelle ratkaisee käyttäjän ongelman. Tätä varten ongelma hajotetaan hallittaviksi paloiksi, joita kutsutaan komponenteiksi; näiden komponenttien toiminnot ja niiden väliset rajapinnat määritetään hyvin tarkasti. Vaatimusmäärittelyä ja suunnittelua pidetään joskus ärsyttävänä johdantona ohjelmointiin, joka nähdään usein todellisena työnä. Tämä asenne voi vaikuttaa erittäin negatiivisesti ohjelmiston laatuun.

SWEBOKin mukaan *Software design* taas tarkoittaa halutun kaltaisen toiminnallisuuden omaavan ohjelmiston sisäisen rakenteen suunnittelua.

Suunnittelu voidaan jakaa järjestelmäsuunnitteluun ja ohjelmistosuunnitteluun. Järjestelmäsuunnittelussa tarkastellaan järjestelmien välistä työnjakoa ja integrointia sekä laitteiston ja ohjelmiston välistä työnjakoa. Järjestelmäsuunnittelussa voidaan päättää hajauttaa

eri ohjelmistoja eri laitteille tai asentaa samaa ohjelmistoa usealle laitteelle. Järjestelmäsuunnittelu on tavallinen vaihe räätälöityjen ohjelmistojen tuotannossa. Valmisohjelmien tuotannossa sitä ei yleensä tarvita. (Wikipedia 2021a.)

Ohjelmiston suunnittelu riippuu tuotteen monimutkaisuudesta, tarvittavista graafisista käyttöliittymistä ja loppukäyttäjien vaatimasta helppokäyttöisyydestä. Suunnittelun tulisi sisältää kaikki asiakkaan kanssa sovitut vaatimukset, jolloin suunnitteluvaihe toimii siltana vaatimusten analyysivaiheen ja koodausvaiheen välillä. Suunnitteluvaiheessa liiketoiminnan vaatimukset muutetaan toiminnallisiksi ja teknisiksi määrittelyiksi (eng. *specification*). (Subramanian 2015).

Varhaisilla suunnittelupäätöksillä on merkittävä vaikutus lopullisen järjestelmän laatuun. Nämä varhaiset suunnittelupäätökset voidaan sisällyttää järjestelmän kokonaiskuvaan eli arkkitehtuuriin. Arkkitehtuuri voidaan seuraavaksi arvioida, ja se voi toimia mallina samanlaisten järjestelmien kehittämisessä tai sitä voidaan käyttää uudelleenkäytettävien komponenttien kehittämisen runkona. Sellaisenaan järjestelmän arkkitehtuurikuvaus on tärkeä merkkipaaluasiakirja nykypäivän ohjelmistokehitysprojekteissa.

Suunnitteluvaiheessa yritämme erottaa, mitä tehdään sen sijaan että miettisimme, miten se tehdään. Meidän pitäisi keskittyä itse ongelmaan emmekä saisi antaa toteutukseen liittyviä huolenaiheiden häiritä suunnittelua. Suunnitteluvaiheen tulos, tekninen määrittely, on lähtökohta toteutusvaiheelle.

2.1.3 Koodaus ja testaus

Subramanianin (2015) mukaan koodaus määritellään vaiheeksi, jossa dokumentoitu suunnittelu muutetaan koodiriviksi ohjelmointikielellä. Tyypillisessä vesiputousmallissa koodaus tehdään vaatimusmäärittelyn ja suunnittelun jälkeen.

SWEBOK määrittelee koodauksen seuraavasti: *Software construction* viittaa aktiviteetteihin, joiden avulla suunniteltu ohjelmisto saadaan toimivaksi tuotteeksi eli käytännössä ohjelmointia ja debuggausta.

Van Vlietin (2008) mukaan toteutusvaiheessa keskitytään yksittäisiin komponentteihin, jolloin lähtökohta on komponentin määrittely. Usein on tarpeen ottaa käyttöön ylimääräinen "suunnitteluvaihe", koska askel komponenttien määrittelystä suoritettavaan koodiin on usein liian pitkä. Tällaisissa tapauksissa voidaan hyödyntää joitain korkean tason ohjelmointikielen kaltaisia merkintöjä, kuten näennäiskoodia eli pseudokoodia. (Pseudokoodi

on eräänlainen ohjelmointikieli. Sen syntaksit ja semantiikka ovat yleensä vähemmän tiukkoja, joten algoritmit voidaan muotoilla korkeammalla, abstraktimmalla tasolla.)

On tärkeää huomata, että ohjelmoijan ensimmäisen tavoitteen tulisi olla hyvin dokumentoidun, luotettavan, helposti luettavan, joustavan ja oikean ohjelman kehittäminen. Tavoitteen ei pitäisi olla tuottaa erittäin tehokasta ohjelmaa täynnä temppuja. Toteutusvaiheen tulos on suoritettava ohjelma.

Koodausvaiheen päätyttyä koodi on testattava alustalla sen oikean toimivuuden varmistamiseksi. Testauksen suorittaa erillinen asiantuntijaryhmä eli ohjelmistotestaajat, jotka ovat erikoistuneet testitapausten luomiseen ja tuotteen testaamiseen erilaisilla testeillä. Vasta sen jälkeen, kun testit ovat osoittaneet tuotteen luotettavuuden, se siirtyy toteutusvaiheeseen (implementointi). (Subramanian 2015.)

Itse asiassa on väärin sanoa, että testaus on erillinen vaihe toteutuksen jälkeen. Tämä viittaa siihen, että meidän ei tarvitse vaivautua testaamaan ennen kuin toteutus on valmis. Tämä ei ole totta. Itse asiassa tämä on yksi suurimmista virheistä, joita voidaan tehdä. Ohjelmoija suorittaa yksikkötestejä jo koodausvaiheessa, ennen kuin ohjelmistotestaajat saavat koodin testattavakseen. Testaukseen on kiinnitettävä huomiota myös vaatimusten suunnitteluvaiheessa. Seuraavien vaiheiden aikana testausta jatketaan ja parannetaan. Mitä aikaisemmin virheitä havaitaan, sitä halvempi on niiden korjaaminen.

SWEBOKin mukaan *software testing* tarkoittaa niitä menetelmiä, joilla varmistutaan siitä, että ohjelmisto toimii kuten halutaan ja että se on riittävän bugiton käytettäväksi.

Van Vlietin (2008) mallin mukaisten vaiheiden aikana tehdään kahdenlaista testausta. On testattava, että siirtyminen seuraavien vaiheiden välillä on oikea, tätä vaihetta kutsutaan todentamiseksi (verifiointi). On myös tarkistettava, että ollaan edelleen oikealla tiellä käyttäjien vaatimusten täyttämisessä (validointi). Verifiointi- ja validointitoimintojen lisääminen kuvan 1.2 lineaariseen malliin tuottaa niin kutsutun ohjelmistokehityksen vesiputousmallin.

2.1.4 Ylläpito

Subramanian (2015) mukaan ohjelmistojen ylläpito on ohjelmistotuotannon ydin. Järjestelmän kehittämisen jälkeen järjestelmä otetaan käyttöön tuotantoympäristössä. Tuotantojärjestelmän muokkaamista toimituksen jälkeen vikojen korjaamiseksi, suorituskyvyn parantamiseksi ja sopeuttamiseksi muuttuvaan ympäristöön kutsutaan ohjelmistojen ylläpidoksi.

Ohjelmiston toimituksen jälkeen siitä voi löytyä virheitä, jotka ovat jääneet huomaamatta. Nämä virheet on luonnollisesti korjattava. Lisäksi järjestelmän todellinen käyttö voi johtaa muutospyyntöihin. Kaikille näille muutostyypeille on annettu melko valitettava termi, nimittäin ”ylläpito”. Ylläpito koskee kaikkia toimintoja, joita tarvitaan järjestelmän pitämiseksi toimintakunnossa sen jälkeen, kun se on toimitettu käyttäjälle. (van Vliet, 2008.)

SWEBOKin mukaan suurin osa ohjelmistoista ei valmistu lopullisesti koskaan. Kun ensimmäinen versio otetaan käyttöön, alkaa ylläpito eli *software maintenance* eli virheitä korjataan ja ohjelmistoa laajennetaan uusilla toiminnoilla.

Kehitystiimin vastuu ei pääty tuotteen käyttöönottoon, vaan se jatkuu huomattavan kauan tämän jälkeen. Asiakkaille ja käyttäjille on tarjottava koulutusta tuotteen käyttämisestä, sillä ilman asianmukaista koulutusta ja tietoa tuotteen toiminnasta on mahdollisuus virheisiin ja epäonnistumisiin. Kaikki käyttäjät eivät ole niin taitavia ohjelmiston teknisissä ominaisuuksissa kuin kehittäjät itse. Ohjelmistoa ylläpidetään ja päivitetään jatkuvasti asiakkaiden ja loppukäyttäjien kasvavien vaatimusten mukaisesti. Kaikki ylläpitovaiheen aikana tehdyt muutokset dokumentoidaan selkeästi myöhempää käyttöä varten. Kun on kehitettävä uusi tuote, jolla on samanlaiset ominaisuudet, nämä dokumentit tarkistetaan uudelleenkäytettävien komponenttien analysoimiseksi. Uudelleenkäytettävien komponenttien myötä kehitystiimin käyttämä aika lyhenee ja työ helpottuu, koska suunnittelu- ja koodausprosessi kyseisen koodin osalta voidaan ohittaa.

2.1.5 Projektinhallinta

Projektinhallinta on toimintaa, joka kattaa kaikki eri vaiheet. Kuten muissakin hankkeissa, ohjelmistokehityshankkeita on hoidettava asianmukaisesti, jotta voidaan varmistaa, että tuote toimitetaan ajallaan ja budjetin rajoissa. Ohjelmistokehityksen näkyvyys- ja jatkuvuusominaisuudet sekä se, että monet ohjelmistokehitysprojektit toteutetaan ilman riittävää aikaisempaa kokemusta, vaikeuttavat huomattavasti projektin hallintaa. Monet esimerkit ohjelmistokehitysprojekteista, jotka eivät ole pysyneet aikataulussa, antavat runsaasti todisteita siitä, että projektinhallinnassa riittää vielä käsiteltävää.

Tärkeä toiminto, jota edellä ei ole erikseen yksilöity, on dokumentointi. Järjestelmädokumentaation keskeisiä komponentteja ovat projektisuunnitelma, laatusuunnitelma, vaatimusmäärittely, arkkitehtuurikuvaus, suunnitteludokumentit ja testaussuunnitelma. Suurempien hankkeiden kohdalla projektin dokumentointiin on käytettävä paljon aikaa ja dokumentointi on aloitettava projektin alkuvaiheessa. Käytännössä dokumentaatiota pide-

tään usein tasapainottavana eränä. Koska monet projektit viivästyvät, dokumentaatio kärsii siitä eniten. Ohjelmistojen ylläpitäjät ja kehittäjät tietävät tämän ja mukauttavat toimintatapansa sen mukaan. Nyrkkisääntönä (Leffingwell 2003) todetaan, että mitä lähemmäksi koodia tullaan, sitä tarkempaa ohjelmistosuunnittelijoiden käytettäväksi tarkoitetun dokumentaation pitää olla. Vanhentuneet vaatimusdokumentit ja muut korkean tason asiakirjat voivat silti antaa arvokkaita vihjeitä. Ne ovat hyödyllisiä ihmisille, joiden täytyy oppia uudesta järjestelmästä tai kehittää esimerkiksi testitapauksia. Vanhentunut matalan tason dokumentaatio on arvoton ja ohjelmoijat tutkivat koodia dokumentaation sijaan. Koska järjestelmään tehdään muutoksia toimituksen jälkeen huomaamatta jääneiden virheiden tai muuttuneiden käyttäjävaatimusten vuoksi, asianmukainen ja ajan tasalla oleva dokumentaatio on ratkaisevan tärkeää ylläpidon aikana.

SWEBOKin mukaan ohjelmistotuotanto sisältää edellä mainittujen vaiheiden lisäksi myös muita osa-alueita. Näitä ovat:

- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering models and methods
- Software quality

Software configuration management viittaa ohjelmiston käyttöön saattamiseen liittyviin kirjastojen, laitteistojen ja käännösprosessin konfigurointiin sekä ohjelmiston versiointiin. Ohjelmistojen tekemiseen liittyy paljon suunnittelua, koordinointia, hallinnointia ja raportointia eli “managementtiä”, tähän viittaa *software engineering management*.

Software engineering process eli ohjelmistotuotantoprosessi kuvaa tapoja tai menetelmiä, joiden avulla ohjelmistoa kehittävien henkilöiden tulisi hoitaa ja ajoittaa eri aktiviteettejä (vaatimukset, suunnittelu, koodaus, testaus), joita ohjelmiston kehittäminen edellyttää. *Software engineering models and methods* kuvaa yksityiskohtaisempia menetelmiä, joita ohjelmistokehityksessä käytetään, kuten mallinnusta ja erilaisia suunnittelumenetelmiä. Ohjelmistojen laatu eli *software quality* on vielä testaustakin avarampi näkemys siihen, miten laadukas ohjelmisto on. Virheettömyyttä ja oikein toimimista keskeisempi kysymys onkin se, sopiiko ohjelmisto ylipäätään käyttäjien tarpeeseen, eli onko se sopiva käyttötarkoitukseensa.

Kuva 2 antaa karkean kuvan suhteellisen isosta työmäärästä, joka yleensä käytetään ohjelmistotuotannon eri vaiheisiin määrittelystä järjestelmän toimittamiseen saakka. Näiden tietojen perusteella syntyy hyvin selkeä trendi, ns. 40-20-40-sääntö: vain 20% työmäärästä käytetään järjestelmän todelliseen ohjelmointiin (koodaamiseen), kun taas edelliset

vaiheet (vaatimusmäärittely ja suunnittelu) ja testaus kukin kuluttaa noin 40% kokonaistyömäärästä. (van Vliet 2008).

Kuva 2. Ohjelmistotuotannon eri vaiheisiin käytetty työmäärä van Vlietin (2008) mukaan

Riippuen erityisistä reunaehdoista, rakennettavan järjestelmän ominaisuuksista ja niin edelleen, tämän säännön muunnelmia löytyy. Toistuvissa kehityshankkeissa ero esimerkiksi vaatimusmäärittelyn, suunnittelun, toteutuksen ja yksikkötestauksen välillä hämärtyy. Suurimmalle osalle projekteja tämä nyrkkisääntö on kuitenkin varsin toimiva. Tämä ei tarkoita, että 40-20-40 -sääntöön on pyrittävä. Vaatimusmäärittelyn aikana tehdyt virheet ovat kalliimpia korjata. On paljon parempi laittaa enemmän energiaa vaatimusten suunnitteluvaiheeseen kuin yrittää poistaa virheet aikaa vievässä testausvaiheessa tai, mikä vielä pahempaa, ylläpidon aikana. Boehmin (1988) mukaan onnistuneet projektit noudattavat 60-15-25-mallia: 60% vaatimusmäärittelyä ja suunnittelua, 15% toteutusta ja 25% testausta. Viesti on selkeä: mitä kauemmin lykkäät koodausta, sitä aikaisemmin olet valmis.

Kuva 2 ei näytä ylläpitotoimenpiteiden työmäärän laajuutta. Kun tarkastellaan ohjelmistojärjestelmän kokonaiskustannuksia sen käyttöiän aikana, käy ilmi, että keskimäärin yksistään ylläpito kuluttaa 50–75% näistä kustannuksista; katso myös kuva 1. Pelkästään ylläpito kuluttaa enemmän kuin eri kehitysvaiheet yhdessä. (van Vliet, 2008).

2.1.6 Elinkaaren loppu

Wikipedian mukaan ohjelma ei ”kuole” koskaan. Ohjelma voi tulla tarpeettomaksi kun

- käyttöympäristö muuttuu siten, että ohjelmaa ei voida enää käyttää
- tulee uusi parempi menetelmä suorittaa ohjelman tekemät asiat
- ohjelman tekemien toimenpiteiden tarve loppuu

- ohjelman toiminnot yhdistetään uuteen ohjelmaan
- ohjelman käyttö ja kehittely loppuu.

Rajlichin (2016, 21) mukaan ohjelmisto lakkaa kehittymästä ja siirtyy elinkaarensa viimeisiin vaiheisiin useista eri syistä. Yksi on se, että ohjelmisto saavuttaa stabiliteetin, jolloin suuria evoluutiomuutoksia ei enää vaadita, vaikka harvoja pieniä korjauksia saattaa silti tapahtua. Stabiliteetti tulee esiin alueilla, joilla ei ole vaihtuvuutta, esimerkiksi sulautettu ohjelmisto, joka ohjaa mekaanista laitetta. Muut alueet eivät koskaan saavuta tällaista stabiliteettia, ja ohjelmiston kehitys voi jatkua loputtomiin. Liiketoiminnallisista syistä johtajat päättävät toisinaan kuitenkin pysäyttää kalliin kehityksen, siirtää ohjelmiston kunnossapitoon ja rajoittaa muutokset minimiin.

Toinen syy Rajlichin (2016, 21) mukaan on se, että ohjelmistojen kehitys voi myös päättyä tahattomasti, kun koodin monimutkaisuus lähtee käsistä tai kun ohjelmistotiimi menettää kehittämiseen tarvittavat taidot. Tässä tilanteessa ohjelmoijat käyttävät usein nopeita ja pinnallisia muutoksia, jotka hämmentävät ja monimutkaistavat ohjelmistorakennetta entisestään. Tuloksena olevaa sekavaa ja monimutkaista ohjelmistokoodia kutsutaan pilaantuneeksi koodiksi, ja se tekee kehittämisen yhä vaikeammaksi ja lopulta mahdottomaksi.

Ohjelmiston kunnossapidon aikana ohjelmoijat eivät enää tee suuria muutoksia ohjelmistoon, mutta tekevät kuitenkin pieniä korjauksia, jotka pitävät ohjelmiston käyttökelpoisena. Kunnossapitovaiheessa olevia ohjelmistoja on kutsuttu legacy-ohjelmistoiksi, vanhentuneiksi ohjelmistoiksi tai ylläpidossa oleviksi ohjelmistoiksi. Kun ohjelmisto on kunnossapitovaiheessa, on erittäin vaikeaa, kallista ja riskialtista palauttaa se kehitysvaiheeseen. Apua on edelleen annettava niille käyttäjille, jotka käyttävät järjestelmää, mutta muutospyyntöjä ei enää hyväksytä, ja käyttäjien on keksittävä tapoja, joilla kiertää mahdolliset puutteet tai viat. Kun korvaava järjestelmä on otettu käyttöön, voidaan järjestelmä sulkea kokonaan pois tuotannosta. Tiedot osoittavat, että suurten ja menestyksekkäitten ohjelmistojen keskimääräinen käyttöikä on noin 10–20 vuotta. (Rajlich 2016, 21.)

2.1.7 Poikkeukset elinkaarimallin käytössä

Kuva 1 sisältää ohjelmiston elinkaaren vaiheittaisen perusmallin. Jotkin ohjelmistoprojektit eivät kuitenkaan läpäise kaikkia vaiheita, vaan ohittavat osan niistä. Esimerkiksi epäonnistuneet projektit päättyvät ennenaikaisesti; jotkut lopetetaan jopa vaatimusmäärittelyn aikana, eivätkä ne koskaan pääse suunnitteluun ja seuraaviin vaiheisiin. Jotkin projektit päättyvät useiden kehityskierrosten jälkeen, mutta ne eivät koskaan siirry ylläpitovaiheeseen. Puhtaat kehitysprojektit alkavat olemassa olevan ohjelmiston laajennuksena ja kehittävät sen uudeksi ja erilaiseksi; nämä projektit ohittavat alkuvaiheen ja alkavat suoraan

suunnittelusta. Pienet ja lyhytaikaiset projektit saattavat ohittaa ohjelmistokehityksen ja siirtyä suoraan ylläpitoon ja sitä seuraaviin vaiheisiin, koska uutta toiminnallisuutta ei tarvitse lisätä kehittämisen kautta. Versioprojektit taas koskevat suurta asiakaskuntaa. Jotkut käyttäjät saavat uusia versioita heti, kun ne tulevat saataville, mutta toiset käyttäjät haluavat säilyttää vanhemmat versiot. Vanhempia versioita ei enää kehitetä, koska useiden versioiden samanaikainen kehittäminen olisi hyvin monimutkaista ja tuhlaavaa. (Rajlich 2016, 22.)

2.2 Ohjelmistokehityksen menetelmät

Tietokoneiden alkuaikoina laitteet olivat kalliita ja ohjelmistot mitättömiä laitteistoihin nähden. Alussa ohjelmointi tapahtui kaapeleita yhdistämällä, kunnes kehitettiin ensimmäiset konekieliset ohjelmat. Yleensä sovellusten käyttäjät koodasivat itse tarvitsemansa ohjelmat. Näitä käytettiin toisen maailmansodan aikaan muun muassa salakirjoitusten murtamiseen tai ohjusten lentoradan laskemiseen. Vähitellen ohjelmistot alkoivat kasvaa ja käyttöalue laajeni muuallekin kuin sotateollisuuteen. Ohjelmistoja alettiin tehdä loppukäyttäjille, jotka eivät enää olleet ohjelmistoalan ammattilaisia, ja kuilu ohjelmiston tekijöiden ja loppukäyttäjien välillä alkoi kasvaa. Tästä seurauksena oli se, että ohjelmistoala alkoi joutua vaikeuksiin ja ongelmiksi Wikipedian (2021b) mukaan muodostuivat muun muassa seuraavat seikat:

- budjetit ylittyivät
- projektit myöhästyivät aikataulusta
- ohjelmistot olivat tehottomia, niiden laatu oli huono ja ne eivät vastanneet käyttäjien tarpeita
- koodin ylläpito ja laajentaminen oli vaikeaa
- ohjelmistoja ei aina saatu ollenkaan toimitettua

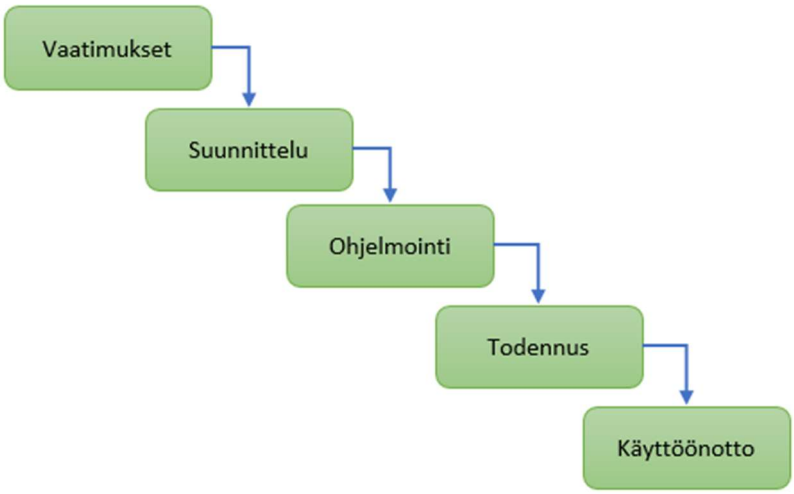
Suurin syy tähän oli se, että tietokoneista oli tullut entistä tehokkaampia ja laskentatehon nopea lisääntyminen oli ohittanut ohjelmoijien kyvyn käyttää näitä ominaisuuksia tehokkaasti. Syntyi idea, että ohjelmistojen kehittämisen tulisi olla kuin mikä tahansa insinöörityö, eli kuten esimerkiksi talon rakentamisessa tulee rakennettava talo ensin tarkasti määritellä ja suunnitella, jonka jälkeen itse rakentaminen on aika suoraviivainen projekti. Viime vuosikymmenien aikana on kehitetty erilaisia prosesseja ja menetelmiä ohjelmistojen laadunhallinnan parantamiseksi. Suuret, monimutkaiset, huonosti määritellyt ja tuntemattomia näkökulmia sisältävät ohjelmistoprojektit ovat kuitenkin alttiita isoille ja odottamattomille ongelmille.

Ohjelmistokehityksen prosessimallit ovat systemaattisia menetelmiä ohjelmistotuotteen kehityksen ohjaamiseksi ja koordinoimiseksi, jotta kaikki asetetut tavoitteet voidaan saa-

vuttaa. Metodologisia prosesseja seurataan vaatimusten analysoinnista ylläpitovaiheeseen saakka. Systemaattiset prosessit tarjoavat ohjeita ja estävät kehitystiimiä poikkeamasta suunnitellulta polulta, mikä johtaisi epäonnistumiseen. Lukuisia ohjelmistokehityksen projektimalleja ja -menetelmiä on aikojen saatossa kehitetty; esittelen seuraavissa kappaleissa niistä lyhyesti vesiputousmallin ja ketterät kehitysmallit.

2.2.1 Vesiputousmalli

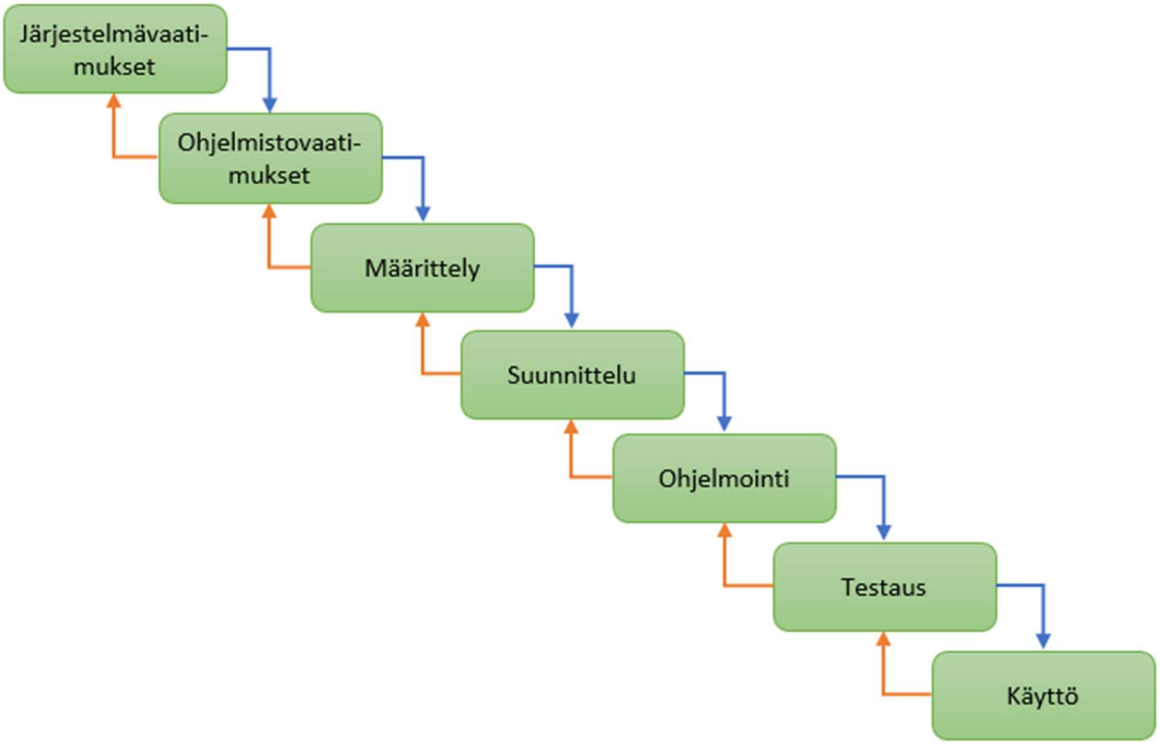
Ensimmäisiä suurehkoja tietokoneohjelmia laadittaessa huomattiin nopeasti tarve systemaattisille työmenetelmille. Ensimmäisiä selkeästi vaiheitettuja lähestymistapoja ohjelmistojen kehittämiseen, mukaan lukien iterointi ja palaute, löytyy jo 1960-luvun alun julkaisuista. Vuonna 1970 Winston Royce julkaisi yhden alamme eniten viitatuista ja samalla myös eniten väärinymmärretyistä artikkeleista: Managing the Development of Large Software Systems. Sen keskeinen sisältö on vesiputousmalli, jonka yksinkertaistettu versio on esitetty kuvassa 3. Tässä mallissa ohjelmistokehitys tapahtui vesiputousmaisesti eli lineaarisesti toisiaan seuraavien vaiheiden sarjana. Vaatimuksista sovitaan, suunnitelma luodaan ja näiden vaiheiden jälkeen seuraa ohjelman koodaaminen. Lopuksi ohjelmisto testataan sen vaatimusten ja suunnittelun mukaisuuden varmistamiseksi. Vesiputousmallin perusajatus on, että jokainen vaihe tehdään valmiiksi ennen seuravan vaiheen aloittamista eikä edelliseen vaiheeseen enää palata. Tarkoituksena on, että jokainen vaihe tuottaa dokumentin tai joukon dokumentteja, jotka toimivat syötteenä seuraavalle vaiheelle. Jälkikäteen näitä dokumentteja on mahdotonta muuttaa.



Kuva 3. Yksinkertainen vesiputousmalli Leffingwelliä mukaillen (2011)

Ironista kyllä, Royce kuvasi sitä itse asiassa mallina, jota ei voitu suositella laajamittaiseen ohjelmistokehitykseen. Uraauurtavassa artikkelissaan (Royce 1970) hän kirjoitti näin:

Kokemukseni mukaan yksinkertaisempi malli ei ole koskaan toiminut suuressa ohjelmistokehitystyössä. Tämän jälkeen hän kuvaili parannettua mallia, johon kuului ensin prototyypin rakentaminen ja sitten prototyypin käyttö sekä vaiheiden välinen palaute lopullisen ohjelmiston rakentamiseksi. Royce pitää iterointia taaksepäin mallin keskeisenä ominaisuutena, kuten kuvassa 4 on esitetty, ja hän myös toteaa, että mikäli samanlaista järjestelmää ei ole aikaisemmin tehty, järjestelmä kannattaa toteuttaa kahdesti. Royce toteaa lisäksi, että jos ongelmat paljastuvat vasta testausvaiheessa, kustannusarvio voi helposti kaksinkertaistua. (Haikala & Mikkonen 2010, 37).



Kuva 4. Vesiputousmalli, jossa otettu mukaan iterointi, vapaasti suomennettu lähteestä (Royce, 1970)

Royce jatkoi mallin kehittelyä ja päätyi lopulta malliin, jossa sovelluksesta tulee ensin tehdä prototyyppi ja vasta siitä saatujen kokemusten valossa kannattaa suunnitella ja toteuttaa lopullinen ohjelmisto. Royce esitteleekin artikkelinsa loppupuolella mallin, missä ohjelmisto tehdään kahdessa iteraatiossa (Royce, 1970). Sallimalla iterointi ja vaiheiden käynnistäminen jo ennen edellisen vaiheen loppua saadaan moneen tilanteeseen sopiva toimintamalli (Haikala & Mikkonen 2010, 37). Vesiputousmallia noudattavasta ohjelmistoprosessista käytetään usein nimitystä plan based process eli suunnitelmavetoinen prosessi, koska prosessi on tarkkaan etukäteen suunniteltu, resursoitu ja aikataulutettu.

Uudempien ohjelmistokehitystekniikoiden sisältä löytää yleensä vesiputousmallin alkuperäisen idean jollakin tavoin toteutettuna, vaikka mallilla on paljon heikkouksia. Vesiputousmalli sopiikin pieniin projekteihin, joissa vaatimukset ovat selkeitä.

Vesiputousmallin ongelmat Tainan, Salmenkiven & Lemströmin (2011) mukaan ovat:

- Vaatimukset eivät saa muuttua.
- Asiakas näkee valmista vasta projektin päätyttyä.
- Aikataulun lipsuessa testaus kärsii.
- Projektityöntekijät turhautuvat jatkuvaan dokumentointiin ja tarkistuksiin.
- Malli ei suosi luovuutta.
- Väärille urille joutuneen projektin saattaminen oikeaan suuntaan on vaikeaa.
- Myöhässä olevaa projektia on vaikea saada takaisin aikatauluun.

Vesiputousmallin hyötyjä Taina ym. (2011) mukaan ovat:

- Malli on selkeä ja helposti omaksuttava.
- Mallille on kehitetty laaja teoria- ja työkalutuki.
- Malli sopii hierarkkisiin organisaatioihin.
- Malli ei rajoita tai vaadi käytettäviä tekniikoita, joten se on helposti räätälöitävissä.
- Malli toimii myös sellaisten tekijöiden kanssa, jotka eivät halua jatkuvaa ryhmätyötä ja isoja projektipalavereja. Malli ei myöskään vaadi tekijöiltä erityistaitoja.
- Mallille on kehitetty toimiva laadunvarmistus ja toimiva prosessin parannus.
- Asiakas saa selkeän ja ymmärrettävän kustannusarvion jo projektin alussa.
    - Mutta kustannusarvio on usein väärä. Projektin alussa tehty arvio voi heittää nelinkertaisesti verrattuna todellisuuteen.
- Kiireisen asiakkaan ei tarvitse osallistua projektityöhön.

2.2.2 Spiraalimalli

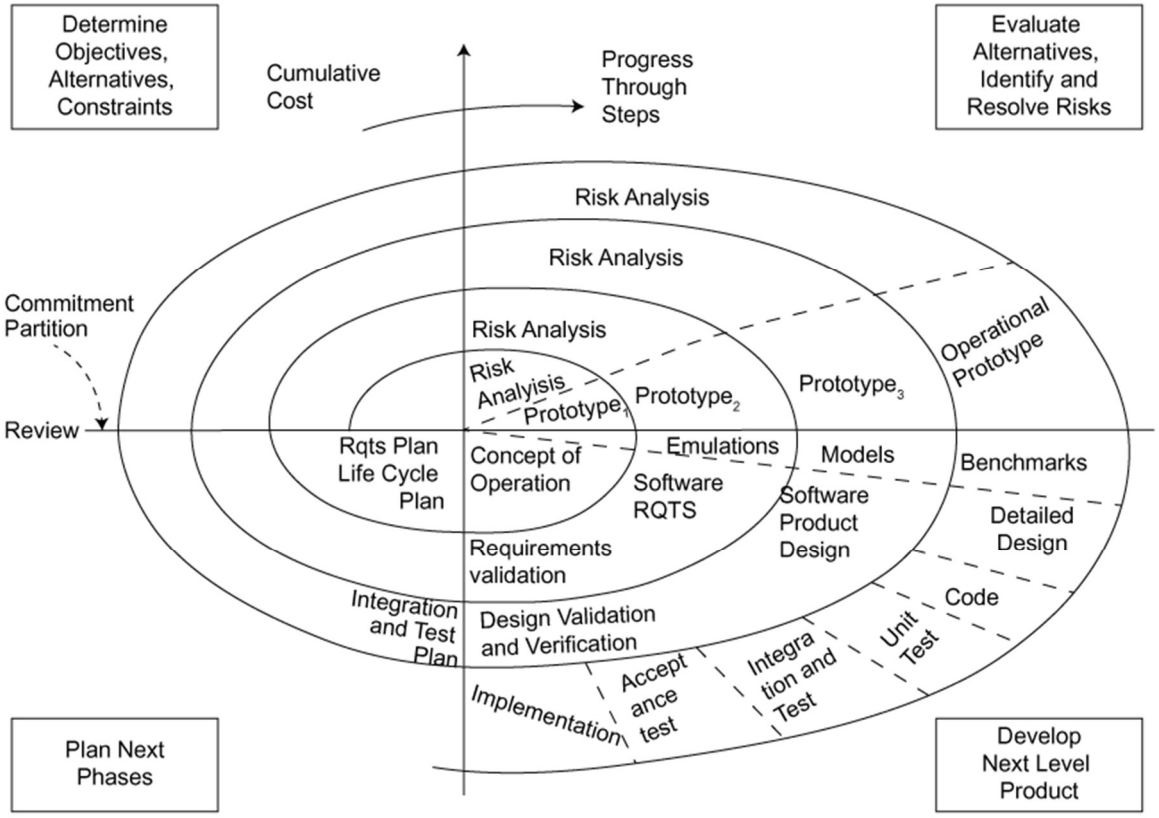
Boehmin keskeinen tutkimus (1988) suositteli erilaista menetelmää ohjelmistokehitysprosessin ohjaamiseksi. Hänen lanseeraama "spiraalimalli" toimii ohjelmistokehityksen mallina niille, jotka uskovat menestyksen noudattavan enemmän riskilähtöistä ja inkrementaalista kehityspolkua.

Leffingwell & Widrig (2003) mukaan spiraalimallissa kehitystä ohjaavat aluksi joukko riskilähtöisiä prototyyppejä; tämän jälkeen strukturoitua vesiputouksen kaltaista prosessia käytetään lopullisen järjestelmän tuottamiseen. Tietenkin väärinkäytettynä spiraalimallilla voi olla yhtä monta ongelmaa kuin väärin käytetyllä vesiputousmallilla. Aikaisemmin esitetyt menetelmät kyllä ottivat riskit huomioon, mutta vasta projektin loppupuolella, mikä lisäsi niiden uudelleen tekemisen taakkaa. Jos riskin todettiin olevan liian suuri, tuote oli suunniteltava uudestaan. Ottaessaan huomioon riskienhallintaprosessin ja testausprosessin merkityksen spiraalimalli määritteli uuden strategian. (Subramanian 2016).

Spiraalimallissa vaatimusmäärittelyllä on edelleen vahva ja aikainen paikkansa. Ensimmäinen kierros spiraalin ympäri on tarkoitettu ensisijaisesti vaatimusten ymmärtämiseen ja

vaatimusten tiettyyn validointiin ennen vakavamman kehityksen aloittamista. Sen jälkeen mallissa oletetaan olevan toinen, suurempi "spiraali", jonka tarkoituksena on kehittää ratkaisu suurimmaksi osaksi vaiheistettuna niin, että suunnittelu, koodaus, integrointi ja testaus seuraavat toisiaan peräkkäisinä vaiheina. Sellaisenaan tämä oli ensimmäinen julkaistu malli, joka perustui enemmän havaintoihin perustuvaan vaatimusmäärittelyyn, tosin sitä seurasi melko perinteinen peräkkäinen, vesiputouksen kaltainen prosessi, mutta joka sisälsi jatkuvan palautteen. Jotkut antavat kiitosta tästä ajattelusta lähtökohtana iteratiiviselle ja inkrementaaliselle (ja nyt yhä ketterämmälle) ohjelmistokehitysmenetelmien liikkeelle.

Spiraalimalli sisältää iteratiiviset vaiheet spiraaliesityksenä, kuten kuvassa 5 on esitetty. Jokainen mallin ympyrä tarkoittaa edellisen tilan seuraavaa tasoa.



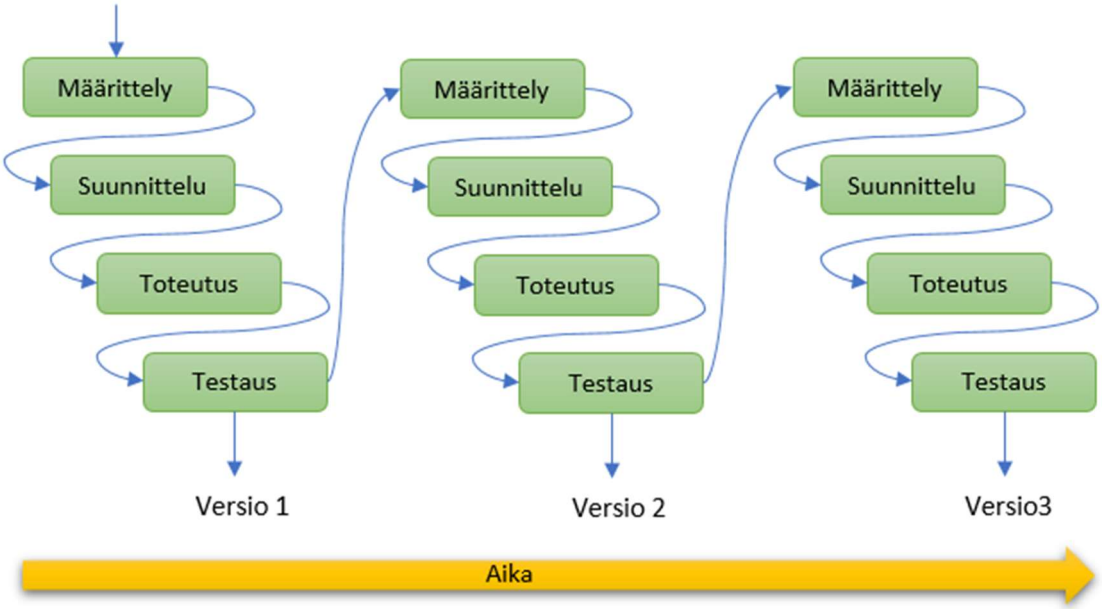
Kuva 5. Spiraalimalli Subramanianin (2015) mukaan

Spiraalimalli alkaa erityisesti siis vaatimusten suunnittelusta ja konseptin validoinnista, jota seuraa yksi tai useampi prototyyppi, jotta voidaan ajoissa vahvistaa, että järjestelmän vaatimukset on ymmärretty oikein. Tämän prosessin tärkein etu on saada mahdollisimman aikaisin mahdollisimman paljon ”Kyllä, mutta” -palautetta käyttäjiltä ja asiakkailta. Tämän perusteellisen lähestymistavan vastustajat huomauttavat, että nykypäivän ympäristössä

meillä ei yleensä ole aikaa täydelliseen konseptin validointiin ja kahteen tai kolmeen prototyyppiin, jota seuraa täsmällinen vesiputousmenetelmä. (Leffingwell & Widrig, 2003).

2.2.3 Iteratiiviset menetelmät

Vuosituhannen alussa monet tiimit olivat omaksuneet uuden lähestymistavan, joka yhdistää parhaat vesiputous- ja spiraalimallit ja on näiden kahden yhdistelmä. Iteratiivinen eli toistuva kehitysmalli on joiltakin osin tarkennus vaiheistetulle prototyyppimallille. Tässä lähestymistavassa koko elinkaari koostuu useista iteraatioista. Jokainen iteraatio on itsessään miniprojekti, joka koostuu elinkaaren eri vaiheista (tutkimus ja analyysi, suunnittelu, toteutus ja testaus). Toistot voivat olla sarjana tai rinnakkain, mutta ne integroidaan lopulta projektin julkaisuun. Yhden toiston eli iteraation pituus voi olla esimerkiksi yksi kuukausi, jolloin toteutettavat vaatimukset voidaan jakaa iteraatioihin ja jokaisen iteraation lopussa voidaan demonstroida siihen mennessä toteutettuja vaatimuksia. Lopullinen iterointi johtaa koko ohjelmistotuotteen julkaisemiseen. (Foster, 2014.). Käytännössä iteratiivisen kehitysmallin voisi ajatella olevan sarja perättäisiä vesiputouksia. Iteratiivisessa lähestymistavassa elinkaarivaiheet erotetaan kuitenkin kussakin vaiheessa tapahtuvista loogisista ohjelmistotoiminnoista, mikä antaa mahdollisuuden palata erilaisiin toimintoihin, kuten vaatimuksiin, suunnitteluun ja toteutukseen, projektin eri iteraatioiden aikana. Lisäksi, kuten spiraalimallissa, jokainen iterointi on suunniteltu vähentämään kehitystoiminnan tässä vaiheessa esiintyviä riskejä. (Leffingwell & Widrig, 2003).

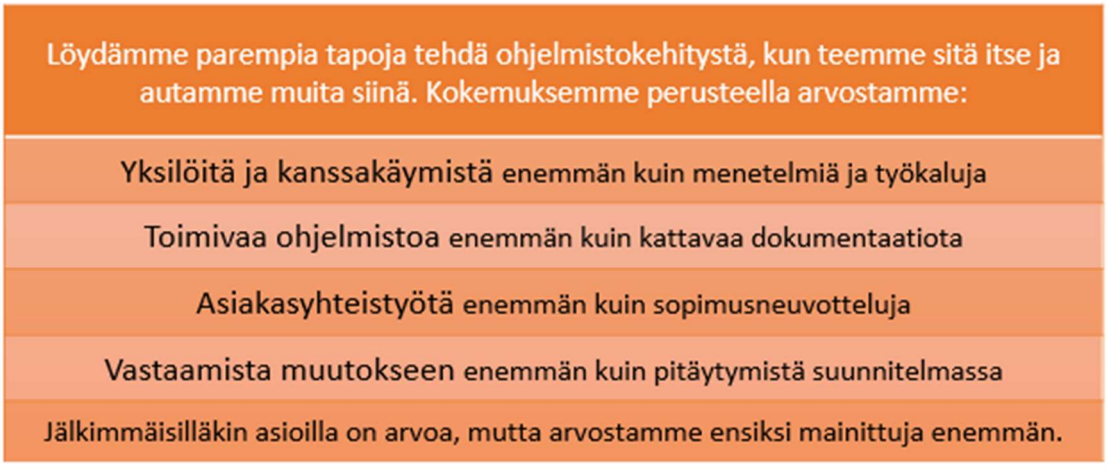


Kuva 6. Iteratiivinen lähestymistapa peräkkäisinä vesiputouksina Haikalaa & Mikkosta mukaillen (2011)

Jokaisen iteraation tuloksena saadaan julkaisu eli lopullisen järjestelmän toimiva osajoukko. Lisäävässä ohjelmistokehityksessä (eng. *incremental development*) tuotteeseen lisätään sykleittäin uusia ominaisuuksia. Näin syntyy iteratiivinen ja inkrementaalinen ohjelmistokehitys eli IDD (eng. *iterative and incremental development*). Kaikki nykyiset ketterät prosessimallit ovat IID-variantteja. (Tapio ym. 2011.)

2.2.4 Ketterä ohjelmistokehitys

"Ketterä" on yhteinen termi menetelmille ja käytännöille, jotka ovat tulleet esiin kahden viime vuosikymmenen aikana ohjelmistoratkaisujen merkityksen, laadun, joustavuuden ja liiketoiminnallisen arvon lisäämiseksi. Nämä mukautuvat hallintamenetelmät on tarkoitettu erityisesti sellaisten ongelmien ratkaisemiseen, jotka ovat historiallisesti vaivanneet ohjelmistokehitystä ja palvelujen toimittamista IT-alalla - mukaan lukien budjetin ylitykset, myöhästyneet määräajat, huonolaatuiset tuotokset ja tyytymättömät käyttäjät. Kevyitä iteratiivisia ohjelmistonkehitysmenetelmiä oli vuoteen 2000 mennessä julkaistu jo kymmenkunta, joista eniten huomiota on vuosien varrella saanut XP eli eXtreme Programming. Vuonna 2001 joukko kevyiden kehitysmenetelmien kehittäjiä, kannattajia ja konsultteja kokoontui keskustelemaan kehittämisen menetelmistä. Erityisesti Yhdysvalloissa suosittiin suunnitelmavetoisia, raskaita ohjelmistokehitysprosesseja. Niiden vastapainoksi kokouksen osallistujat alkoivat kutsua kevyitä kehittämismenetelmiä ketteriksi (agile) menetelmiksi. Kokouksen aikana sovittiin ketteriä menetelmiä yhdistävistä periaatteista ja perustettiin ketterien menetelmien edistämistä ajava Agile Alliance-järjestö, jonka peruskirjana julkaistiin Agile Manifesto. (Haikala & Mikkonen 2011, 44.) Agile Alliance täytti juuri 40 vuotta ja voi edelleen hyvin. Manifestin ydin on sen arvoissa, jotka on esitetty seuraavassa kuvassa.



Kuva 7. Agile Manifesto (Agile Alliance 2001)

Manifestin arvot on liian usein ja liian helposti tulkittu niin, että prosessi, työkalut, dokumentaatio ja suunnitelmat ovat turhia. Tarkkaan ottaen manifesti kuitenkin tunnustaa ne

arvokkaiksi ohjelmistonkehitystyön kannalta, mutta toteaa, että kaikkein tärkeintä on kuitenkin tyytyväinen asiakas ja toimiva ohjelmisto. Arvot konkretisoituvat manifestin 12 perusarvoon, jotka on kirjattu alla olevaan taulukkoon.

Taulukko 1. Ketterän kehityksen 12 periaatetta (Agile Alliance 2001):

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

Monet ketterän perusarvot on helppo hyväksyä tietyin rajauksin. Arvojen lähtökohtana tuntuu monissa kohdissa olevan ajatus pienehköstä asiakasprojektista. Tuotekehityshankkeisiin, joissa ohjelmisto on usein laajemman kokonaisuuden osa, edellä mainittuja periaatteita voi olla vaikea tai jopa mahdoton soveltaa. Esimerkiksi asiakkaan jatkuva läsnäolo ei tuotekehityshankkeissa tule yleensä kysymykseen ja kasvokkain kommunikoinnin järjestäminen suurissa maantieteellisesti hajautetuissa projekteissa on haasteellista.

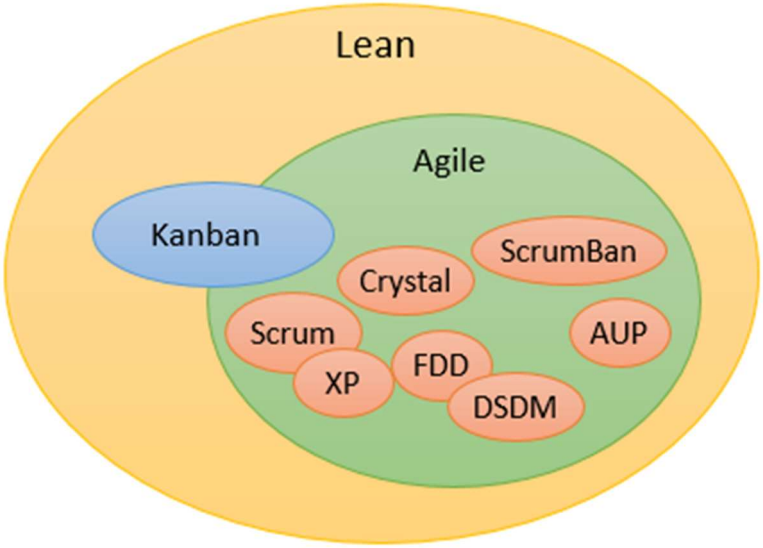
On syytä huomata, että vaikka termi "ketterä" yleistyi manifestin jälkeen, projektiryhmien nykyiset lähestymistavat ja tekniikat olivat käytössä jo ennen ketterää manifestia monien vuosien ja joissakin tapauksissa vuosikymmenien ajan. Ketterät lähestymistavat ja ketterät menetelmät ovat sateenvarjotermejä, jotka kattavat erilaisia kehyksiä ja menetelmiä. Kuva 8 sijoittaa ketterän kontekstiin ja visualisoi sen yleisenä terminä, viitaten mihin tahansa lähestymistapaan, tekniikkaan, kehykseen, menetelmään tai käytäntöön, joka täyttää ketterän manifestin arvot ja periaatteet. Kuvassa 8 esitetään myös ketterä ja Kanban-menetelmä lean-alaryhminä. Tämä johtuu siitä, että niillä on yhteisiä lean-käsitteitä, kuten: "keskittyminen arvoon", "pienet eräkoot" ja "hukan poistaminen". (Project Management Institute 2017.)

Yksi tapa ajatella lean-ajattelun, ketterän ja Kanban-menetelmän välistä suhdetta on pitää ketterää ja Kanban-menetelmää lean-ajattelun jälkeläisinä. Toisin sanoen, lean-ajattelu on eräänlainen superjoukko, jolla on samoja ominaisuuksia ketterän ja Kanbanin kanssa. Toyotan autotehtaalta tunnetuksi tullut lean-ajattelutapa on siis tavallaan ketterien menetelmien kantaäiti, josta esimerkiksi Scrumin perusteet tulevat. Nämä yhteiset ominaisuudet ovat hyvin samanlaisia ja keskittyvät arvon tuottamiseen, ihmisten kunnioittamiseen, hukan minimointiin, avoimuuteen, muutoksiin sopeutumiseen ja jatkuvaan parantamiseen. Projektiryhmien mielestä on joskus hyödyllistä sekoittaa erilaisia menetelmiä – ja pitäisikin tehdä niin kuin tiimille tai organisaatiolle parhaiten sopii, riippumatta menetelmän alkuperästä. Tavoitteena on paras tulos käytetystä lähestymistavasta huolimatta.

Lean-ajattelu on johtamisfilosofia, joka keskittyy seitsemän erilaisen turhuuden eli hukan (tuottamattoman toiminnon) poistamiseen. Sen avulla pyritään parantamaan asiakastyytyväisyyttä ja laatua, pienentämään kustannuksia ja lyhentämään tuotannon läpimenoaikoja. Lean pyrkii siihen, että oikea määrä oikeanlaatuisia oikeita asioita saadaan oikeaan aikaan oikeaan paikkaan oikean laatuisena. Samaan aikaan vähennetään kaikkea turhaa ja ollaan joustavia sekä avoimia muutoksille. Leanissa on keskeistä tunnistaa ja eliminoida hukka nopeasti ja tehokkaasti, pienentää kustannuksia sekä parantaa laatua. Hukalla tarkoitetaan ylimääräisiä, tuottamattomia toimintoja, jotka hidastavat prosessia tai tuottavat tarpeettomia kustannuksia. Hukka on seurausta prosesseissa tapahtuvista vioista ja virheistä, jotka vaihtelu aiheuttaa. Jos poistetaan vain hukkaa, hukka tulee aina uudestaan koska hukan syytä ei ole poistettu, vain "oire". (Wikipedia 2021c.)

Kanban-menetelmä on saanut inspiraationsa alkuperäisestä lean-ajattelun mukaisesta valmistusjärjestelmästä ja sitä käytetään erityisesti tietotyöhön. Se syntyi 2000-luvun puolivälissä vaihtoehtona ketterille menetelmille. Kanban-menetelmä on vähemmän mää-

räävä kuin jotkut ketterät lähestymistavat ja vähemmän häiritsevä, koska se on alkuperäinen "aloita siitä missä olet” -lähestymistapa. Projektiryhmät voivat aloittaa Kanban-menetelmän soveltamisen suhteellisen helposti ja edetä kohti muita ketteriä lähestymistapoja, jos he pitävät sitä tarpeellisena tai tarkoituksenmukaisena. (Project Management Institute 2017.)



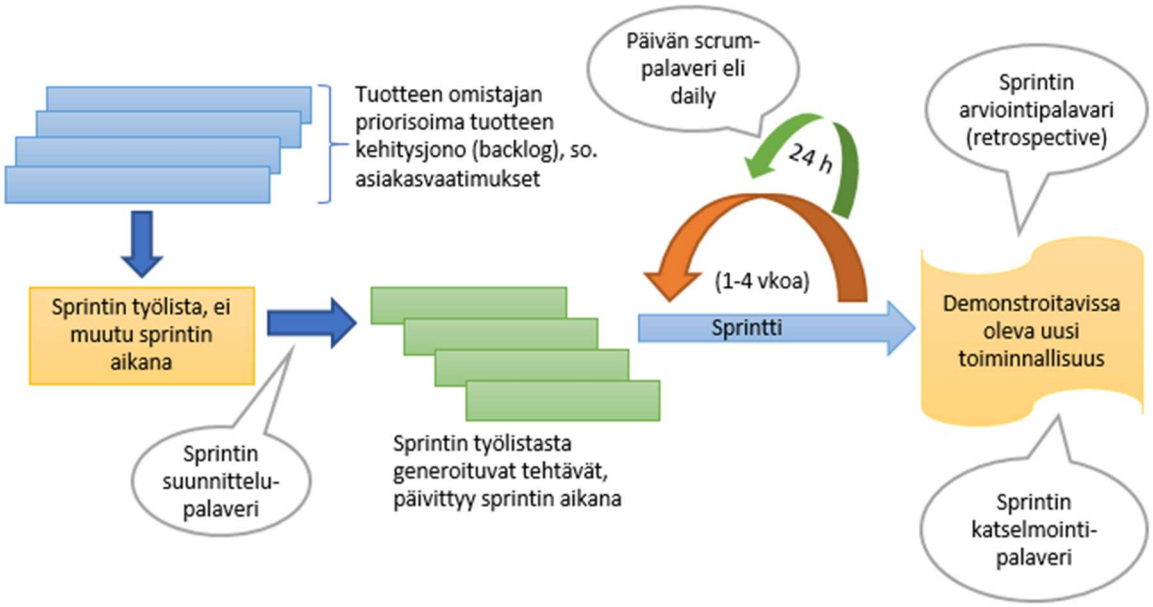
Kuva 8. Agile on kattotermi usealla ketterälle menetelmälle tai lähestymistavalle Project Management Instituten mukaan (2017)

Cooken (2014) mukaan Kanban on ketterä työmäärän ja muutoksenhallintamenetelmä, jota voidaan käyttää yhdessä muiden ketterien menetelmien kanssa tai niistä riippumatta. Tällä hetkellä sitä käytetään laajimmin IT-tuki- ja ylläpitotoimien hallinnassa, jossa ensisijainen työ voi muuttua viikoittain, päivittäin tai jopa tunneittain.

Haikalan & Mikkosen (2010, 47) Scrum on yksi tunnetuimmista ja käytetyimmistä ketteristä menetelmistä, jopa niin, että sanasta Scrum on tullut lähes sanan ketterä synonyymi. Usein projektinhallintamenetelmien lanseeraus ei yleensä ole innostanut ohjelmistosuunnittelijoita mutta Scrum on tästä säännöstä selkeä poikkeus; se on saanut pääsääntöisesti ohjelmistosuunnittelijoilta lämpimän vastanoton. Vaikka Scrumia yleensä ketteränä menetelmänä pidetäänkin, todellisuudessa Scrum ei välttämättä ole kovinkaan ketterä, vaan sitä voidaan pitää jopa jonkinlaisena ketteryyden jäykistäjänäkin.

Scrumin etuihin kuuluu yksinkertaisuus; sen perusperiaatteet on helppo selittää nopeasti ja ensi näkemältä Scrum tuntuisi tarjoaman ratkaisuja hyvinkin moniin ohjelmistotuotannon ongelmiin. Yksinkertaisuus on jossain määrin kutienkin harhaanjohtavaa, sillä Scrum ottaa kantaa vain joihinkin ohjelmiston elinkaareen liittyvistä tehtävistä. Esimerkiksi vaati-

musmäärittely on ulkoistettu projektin omistajan tehtäväksi. Scrum ei myöskään ole projektinhallintamenetelmä, vaan lähinnä projektin toteutusvaiheeseen tarkoitettu tapa hallita ja organisoida projektin iteraatioita eikä se ota kantaa käytettyihin kehitysmenetelmiin tai työkaluihin. Toisin sanoen Scrum yksistään ei riitä vaan se on yhdistettävä jollain tavalla kaikkeen muuhun projektinhallinnassa käytettävään välineistöön. (Haikala & Mikkonen, 2010, 47.)



Kuva 9. Scrum-prosessi Haikalan & Mikkosen (2011) mukaan

Scrum on saanut nimensä rugbystä, jossa scrum on keino aloittaa pelaaminen uudelleen sääntöjen rikkomisen jälkeen. Ketterän Scrum-metodologian idea on, että pieni joukkue yhdistetään yhden tavoitteen ympärille ja se kokoontuu kehityksen sprintteihin, jotka vievät heitä kohti tätä tavoitetta. (Dooley 2011.)

Scrumissa työskennellään toistavasti ja lisäävästi (eng. *iterative-incremental*) ennustettavuuden optimoimiseksi ja riskien kontrolloimiseksi. Tavoitteena oleva tuote kehittyy pikkuhiljaa täydellisemmäksi ja valmiimmaksi useiden kehitysjaksojen aikana. Kehitysjaksoa kutsutaan sprintiksi (eng. *Sprint*). Sprintti on 1-4 viikon mittainen aikaraja, jonka sisällä tuotetaan “valmiin” määritelmän täyttävä, käyttökelpoinen ja potentiaalisesti julkaisukelpoinen tuoteversio. Jokaisen sprintin sisältö sovitaan sprintin suunnittelupalaverissa ennen sprintin aloitusta, ja toteutettaviksi valitaan sellaisia tuotteen kehitysjonon (eng. *backlog*) kohtia, joilla on sillä hetkellä suurin merkitys projektin onnistumiselle. Sprintin lopuksi järjestetään sprinttikatselmus, jossa kehitystiimi esittelee sprintin konkreettiset saavutukset (esim. ohjelmiston uusimman version) tuoteomistajalle sekä mahdollisille sidosryhmien edustajille palautteen saamiseksi ja ymmärryksen lisäämiseksi kehityksen tilasta. Ennen seuraavan

sprintin aloittamista pidetään vielä sprintin retrospektiivi, jossa tarkastellaan prosessin näkökulmasta mikä sprintin aikana sujui hyvin ja mitä voitaisiin parantaa seuraavassa sprintissä. (Wikipedia 2021d.)

Tässä kappaleessa en mene sen syvällisemmin Scrumiin tai muihin ketteriin menetelmiin, koska niistä löytyy hyvin paljon aineistoa eikä ole tarkoituksenmukaista kertoa niistä tässä kohtaa.

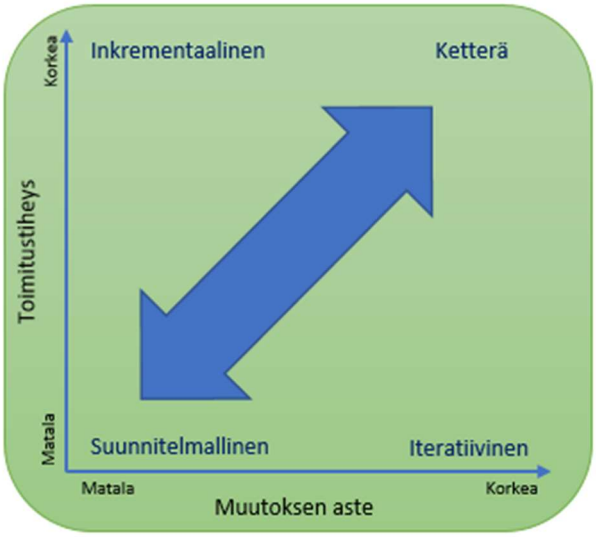
2.2.5 Elinkaarimallin valinta

Kuvassa 10 esitetään yhteenveto neljän elinkaarimallin ominaisuuksista. On tärkeää huomata, että kaikilla projekteilla on nämä ominaisuudet – mikään projekti ei ole täysin vailla vaatimuksia, toimitusta, muutosta ja tavoitteita koskevia näkökohtia. Projektin ominaispiirteet määrittävät, mikä elinkaarimalli sopii parhaiten projektiin.

Menetelmä	Vaatimukset	Toiminnot	Toimitus	Tavoite
Suunnitelmallinen	Kiinnitetty	Suoritetaan kerran koko projektin aikana	Yksi toimitus	Kulujen hallinta
Iteratiivinen	Dynaaminen	Toistetaan kunnes oikein	Useita toimituksia	Ratkaisun oikeellisuus
Inkrementaalinen	Dynaaminen	Toistetaan kerran inkrementin aikana	Säännöllisiä pieniä toimituksia	Nopeus
Ketterä	Dynaaminen	Toistetaan kunnes oikein	Säännöllisiä pieniä toimituksia	Arvon tuottaminen asiakkaalle säännöllisten toimitusten ja palautteen kautta

Kuva 10. Yhteenveto neljän elinkaarimallin ominaisuuksista (Project Management Institute 2017)

Toinen tapa ymmärtää, miten projektin elinkaaret vaihtelevat, on käyttää jatkumoa, joka vaihtelee toisen pään suunnitelmallisista jaksoista toisessa päässä oleviin ketteriin sykleihin, ja jossa keskellä on enemmän toistuvia eli iteratiivisia tai lisääviä eli inkrementaalisia jaksoja. (Project Management Institute 2017.)



Kuva 11. Elinkaarimallien jatkumo Project Management Instituten mukaan (2017)

Hankkeita on monen muotoisia ja kokoisia, ja niihin voi ryhtyä monin eri tavoin. Projektitiimit tarvitsevat tietoa käytettävissä olevista ominaisuuksista ja vaihtoehdoista valitakseen sen lähestymistavan, joka todennäköisimmin onnistuu juuri siinä tilanteessa. Mikään elinkaarimalli ei ole täydellinen tai sopiva kaikkiin projekteihin. Sen sijaan jokainen projekti löytää elinkaarimallin jatkumosta kohdan, joka tarjoaa optimaalisen ominaisuuksien tasapainon kontekstilleen. Erityisesti

- Suunnitelmallisessa tai ennustavassa elinkaarimallissa hyödynnetään asioita, jotka ovat tunnettuja ja todistettuja. Tämä vähentynyt epävarmuus ja monimutkaisuus mahdollistavat sen, että tiimit voivat segmentoida työn ennakoitavissa olevien ryhmittelyjen sarjaan.
- Iteratiivisissa malleissa annetaan osittain valmiista tai keskeneräisestä työstä saadun palautteen parantaa ja muokata työtä.
- Inkrementaalisessa mallissa toimitetaan valmis tuote, jota asiakas voi käyttää välittömästi.
- Ketterissä malleissa hyödynnetään sekä iteratiivisia että inkrementaalisia ominaisuuksia. Kun tiimit käyttävät ketteriä lähestymistapoja, he iteroivat luodakseen valmiita julkaisuja. Tiimi saa varhaista palautetta ja tarjoaa asiakkaalle tuotteen suhteen näkyvyyttä, luottamusta ja hallintaa. Koska tiimi voi julkaista tuotteen aikaisemmin, projekti mahdollistaa sijoitukselle aikaisemman tuoton, koska tiimi tuottaa ensin sen työn, jolla on korkein arvo.

Kokonaisessa hankkeessa ei tarvitse käyttää yhtä ainoaa lähestymistapaa. Hankkeissa yhdistyvät usein eri elinkaarielementit tiettyjen tavoitteiden saavuttamiseksi. Suunnitelmallisen, iteratiivisen, inkrementaalisen ja/tai ketterän lähestymistavan yhdistelmä on hybridilähestymistapa.

Hybridimallin ensimmäisissä kehitysvaiheissa hyödynnetään ketterää kehityskaarta, jota seuraa suunnitelmallinen käyttöönottovaihe. Tätä lähestymistapaa voidaan käyttää, kun

hankkeen kehitysvaiheessa esiintyy epävarmuutta, monimutkaisuutta ja riskejä, jotka hyötyisivät ketterästä lähestymistavasta. Tätä seuraisi määritelty, toistettava käyttöönottovaihe, joka on tarkoituksenmukaista toteuttaa suunnitelmallisella tavalla, ehkä jopa eri tiimin voimalla. Esimerkki tästä lähestymistavasta on uuden korkean teknologian tuotteen kehittäminen, jota seuraa tuhansien käyttäjien käyttöönotto ja koulutus. (Project Management Institute 2017.)

3 Vaatimukset osana ohjelmistotuotantoa

Tässä luvussa käydään tarkemmin läpi vaatimusmäärittelyn merkitystä ohjelmistokehityksessä sekä sen eri vaiheita, metodeja ja käytänteitä sekä perinteisen että ketterän ohjelmistokehityksen näkökulmasta. Vaatimusmäärittely on yksi haastavimmista ohjelmistokehityksen osa-alueista. Se on myös kiistatta tärkein näkökulma, koska se luo perustan kaikille myöhemmille ohjelmistokehityksen projektitöille.

Ohjelmistoprojektien menestys riippuu suuresti vaatimusmäärittelyprosessin laadusta. Huolimatta uusista ohjelmistotuotannon tekniikoista ohjelmistojärjestelmien kehittämishankkeet ovat edelleen alttiita epäonnistumisille. Aivan liian usein monimutkaiset ohjelmistoprojektit ovat myöhässä, ylittävät budjetin eivätkä täytä järjestelmän loppukäyttäjien tai järjestelmästä maksavan organisaation todellisia tarpeita. Tällaiseen johtopäätökseen ovat päätyneet useat asiantuntijat ja monet tutkimukset osoittavatkin**,** että suurin syy ohjelmistoprojektien epäonnistumiselle on huono vaatimusmäärittely. Hakalan & Mikkosen (2011, 61) mukaan hyvin suoritettu vaatimusmäärittely on onnistuneen ohjelmistoprojektin perusedellytyksiä ja se toimii yhdistävänä tekijänä muiden ohjelmistotuotantoon liittyvien toimintojen välillä. Vaatimusmäärittely on siten yksi tärkeimmistä ohjelmistoprojektin vaiheista. Se toimii ohjelmiston perustana ja sen tarkoitus on kertoa, mitä eri sidosryhmiin kuuluvat toimijat ohjelmistolta haluavat. Chemuturin (2011, 10) mukaan vaatimusmäärittely on prosessi, jonka aikana käyttäjien vaatimukset muokataan sellaiseen muotoon, että ne mahdollistavat käyttäjien tarpeet täyttävän ohjelmiston kehittämisen. Vaatimusmäärittely sisältää vaatimusten keräämisen, analysoinnin, kirjaamisen ja seurannan läpi ohjelmistokehityksen. Laplanten (2011, 2) mukaan vaatimukset ovat tärkeitä, koska ne muodostavat perustan kaikelle seuraavalle kehitystyölle. Kun vaatimukset on asetettu, kehittäjät aloittavat muun teknisen työn: järjestelmän suunnittelun, kehittämisen, testauksen, toteutuksen ja käytön.

Laplanten (2011, 2) mukaan niin sanottu "todellinen työ" (ohjelmiston kehittäminen tai ohjelmointi) halutaan aloittaa liian nopeasti liian usein. Monet asiakkaat ja projektipäälliköt näyttävät uskovan, että varsinainen ohjelmointityö ("koodaus") osoittaa edistymistä. Alan kokemuksen mukaan järjestelmän kehittämiseen liittyviin vaatimuksiin ja toimintoihin ei käytetä riittävästi aikaa ja vaivaa. Alan kokemus myös vahvistaa, että parempi tapa on sijoittaa enemmän aikaa vaatimusten keräämiseen, analysointiin ja johtamiseen. Syynä on se, että koodaustyö aloitetaan tyypillisesti paljon aikaisemmin kuin pitäisi, sillä todellisten vaatimusten tunnistamiseen ja vaatimuksiin liittyvien toimintojen suunnitteluun tarvitaan lisää aikaa.

Vaatimusmäärittely ei kuitenkaan ole pääasiassa vain vaatimusdokumentin kirjoittamista. Sen sijaan vaatimusmäärittelyssä pitäisi keskittyä ymmärtämään liike-elämän ongelmia ja tarjoamaan niihin ratkaisun. Ohjelmistot ovat olemassa jonkinlaisten ongelmien ratkaisemiseksi, samoin kuin laitteistot ja palvelut. Vaatimusmäärittelyssä taitoa vaativin osuus on todellisen ongelman löytäminen. Kun se löydetään, saadaan perusta vaihtoehtoisten ratkaisujen tunnistamiseen ja valitsemiseen. Pohjimmiltaan vaatimukset eivät siis koske kirjallisia vaatimuksia sinänsä, vaan pikemminkin ratkaistavan ongelman selville saamista. Paakin (2010) mukaan ennen kuin voimme toteuttaa jonkin ongelman ratkaisevan ohjelmiston, pitää ymmärtää ja määritellä, mikä on se ratkaistava ongelma. Tätä varten pitää keksiä, ymmärtää, muotoilla ja päättää kolme asiaa:

- Mikä on ratkaistava ongelma?
- Miksi ongelma pitää ratkaista?
- Kenen vastuulla on ongelman ratkaisu?

Vaatimusmäärittelyssä vastataan siis kolmeen kysymykseen: Mitä halutaan? Miksi halutaan? Kuka ottaa vastuun?

Vaikka tässä työssä viitataan vaatimusten ”asiakirjoihin” tai ”dokumentteihin”, niiden ei tarvitse olla perinteisiä paperisia tai sähköisiä asiakirjoja. Sen sijaan niitä voisi ajatella yksinkertaisesti säilöinä, joihin voi tallentaa vaatimuksia koskevan tiedon. Tällainen säilö voi olla perinteinen asiakirja, tai se voi olla laskentataulukko, kaavio, tietokanta, vaatimustenhallintatyökalu tai jokin näiden yhdistelmä.

3.1 Vaatimuksen määritelmä

Vaatimukset ovat avain teknisten projektien onnistumiseen tai epäonnistumiseen. Ne ovat kaiken jatkotyön perusta ja olennainen osa ohjelmiston suunnittelua. Kun ryhmä ihmisiä alkaa keskustella vaatimuksista, he törmäävät usein ensimmäiseksi terminologiaongelmaan. Eri projektiryhmän jäsenet, sidosryhmien edustajat, kehittäjät tai käyttäjät saattavat kuvata yhtä ominaisuutta käyttäjän vaatimukseksi, ohjelmistovaatimukseksi, liiketoiminnan vaatimukseksi, toiminnalliseksi vaatimukseksi, järjestelmävaatimukseksi, tuotevaatimukseksi, projektivaatimukseksi, käyttäjätarinaksi, ominaisuudeksi tai rajoitukseksi. Asiakkaan määrittelemät vaatimukset saattavat kuulostaa kehittäjästä korkean tason tuotekonseptilta ja kehittäjän käsite vaatimuksista saattaa taas kuulostaa käyttäjästä yksityiskohtaiselta käyttöliittymäsuunnittelulta. Tämä ymmärtämisen monimuotoisuus johtaa usein sekaannukseen ja turhautumiseen.

Vaatimukselle löytyy useita, sanamuodoiltaan hieman toisistaan poikkeavia määrittelyitä. Hakalan & Mikkosen (2011, 61) mukaan vaatimus on jotain, mitä tuotteella pystyy tekemään, tai ominaisuus, joka tuotteella tulee olla. Wiegersin & Beattyn (2013) suosikkimääritelmä on lainattu Leffingwelliltä (2011) ja se kuuluu karkeasti käännettynä näin: ”Vaatimukset ovat erittely siitä, mitä tulisi toteuttaa. Ne kuvaavat järjestelmän käyttäytymistä tai järjestelmän ominaisuutta tai määritettä. Ne voivat rajoittaa järjestelmän kehittämisprosessia.” Tässä määritelmässä tunnustetaan erityyppiset tiedot, joita yhdessä kutsutaan vaatimuksiksi. Vaatimukset kattavat sekä käyttäjän näkemyksen ulkoisesta järjestelmän toiminnasta että kehittäjän näkemyksen joistakin sisäisistä ominaisuuksista. Ne sisältävät sekä järjestelmän käyttäytymisen tietyissä olosuhteissa että ne ominaisuudet, jotka tekevät järjestelmästä sopivan niille käyttäjille, joille se on tarkoitettu. Ohjelmistoyhteisöt eivät käytä sanaa ”vaatimus” samassa merkityksessä kuin sanan määritelmä sanakirjassa: jotain vaadittua tai pakollista, tarvetta tai välttämättömyyttä. Ihmiset kysyvät joskus, onko heidän edes priorisoitava vaatimuksia, koska ehkä matalan prioriteetin vaatimuksia ei koskaan toteuteta. Jos sitä ei todella tarvita, se ei ole vaatimus, he väittävät. Ehkä, mutta miten tuohon väitteeseen sitten pitäisi reagoida. Jos vaatimus siirretään tämän päivän projektista määrittelemättömään tulevaan julkaisuun, pidetäänkö sitä edelleen vaatimuksena? Vastaus on kyllä.

Robertsonin & Robertsonin (2013) mukaan vaatimukset ovat jotain, mitä ohjelmistotuotteen, laitteistotuotteen tai palvelun, tai mitä tahansa aiotaankin rakentaa, on tarkoitus tehdä ja olla. Vaatimuksia on riippumatta siitä, löydetäänkö ne vai ei, kirjataanko ne ylös vai ei. Tuote ei tietenkään ole koskaan oikea, ellei se ole vaatimusten mukainen, joten tällä tavalla voidaan ajatella vaatimuksia jonkinlaisena luonnollisena lakina, ja ne on löydettävä.

Chemuturin (2011, 3) mukaan ohjelmistokehitysteollisuus noudattaa yleensä näitä yllä mainittuja määritelmiä, mutta koska olosuhteet muuttuvat jatkuvasti erityisesti ohjelmistojen ja ohjelmistokehityksen alalla, tarvitaan kattavampi määritelmä ja ikään kuin yhteenvetona yllä olevista Chemuturi määritteli vaatimuksen ohjelmistokehityshankkeiden yhteydessä seuraavasti: ”Vaatimus on minkä tahansa sidosryhmän tarve, odotus, rajoitus tai (käyttö)liittymä, joka ehdotetun ohjelmistotuotteen on täytettävä sen kehittämisen aikana”. Tämä määritelmä pitää sisällään seuraavat avaintermit:

1. **Tarve** - jotain perustavaa laatua olevaa, jota ilman olemassaolo muuttuu kestämättömäksi. Se on ehdoton vähimmäistarve, jotta järjestelmä olisi hyödyllinen. Jos tarvetta ei täytetä, järjestelmästä tulee käyttökelvoton tai vähemmän käyttökelpoinen.
2. **Odotus** - ilmoittamaton tarve. Kun käyttäjät uskovat ohjelmistojen kehittämisen tiimille, odotetaan, että kehitystiimi asiantuntemuksellaan ymmärtää myös käyttäjän odotukset.

3. **Rajoitus** - este, jonka kanssa käyttäjän on elettävä. Se voi olla ohjelmistosuunnittelun tai -kehityksen rajoitus tai reunaehto.
4. **Käyttöliittymä** - se on perusta vuorovaikutukselle käyttäjän, asiakkaiden ja toimittajien välillä.
5. **Sidosryhmä** – sidosryhmä on joku, johon inhimillisen toiminnan tulos vaikuttaa. Ohjelmistokehityshankkeessa on useita sidosryhmiä, kuten loppukäyttäjät, projektitiimiläiset, markkinointitiimin jäsenet ja johtajat.
6. **Joka on täytettävä**– Tarve pitää täyttää. Jos sitä ei voida täyttää johtuen joko teknologisista tai taloudellista rajoituksista, siitä tulee tulevaisuuden vaatimus. Jos tarvetta ei voida täyttää nykyisellä pyrkimyksellä, pyrkimys itsessään muuttuu tarpeettomaksi.
7. **Ehdotettu ohjelmistotuote** - se kohta, jossa tarpeen odotetaan täyttyvän, tämän työn lopputulos.
8. **Kehittämisen aikana** - Tämä määrittää aikajanan, jolloin tarve on täytettävä. Jos tarve ei toteudu nykyisen kehittämisen aikana, tarve jää täyttämättä tai on tulevaisuuden tarve.

Robertsonin & Robertsonin (2013) mukaan vaatimus on jotain, joka tuotteen on tehtävä tukeakseen omistajansa liiketoimintaa, tai laatu, joka tekee siitä omistajan silmissä hyväksyttävän ja houkuttelevan. Vaatimus on olemassa joko siksi, että tuotetyyppi vaatii tiettyjä toimintoja ja ominaisuuksia, tai siksi, että asiakas pyytää perustellusti kyseisen vaatimuksen olevan osa toimitettua tuotetta.

Kotonyan ja Sommervillen (2004, 7) mukaan vaatimuksen periaatteessa tulisi ilmaista, mitä ohjelmiston tai palvelun pitäisi tehdä mieluummin kuin että miten se sen tekisi. Käytännössä asia ei kuitenkaan ole niin yksiselitteinen, koska vaatimusmäärittelyä lukevat yleensä käytännönläheiset kehittäjät, jotka ymmärtävät paremmin toteutuskuvauksia kuin abstrakteja ongelman selityksiä. Lisäksi vaatimusmäärittelyn kohteena oleva ohjelmisto tai järjestelmä on yleensä osa suurempaa kokonaisuutta, joten ollakseen yhteensopiva muiden ohjelmistojen kanssa sekä ollakseen standardien mukainen, saatetaan joutua määrittelemään myös sellaiset toteutuskäytännöt, jotka rajoittavat järjestelmän suunnittelua. Siksi vaatimukset sisältävät aina sekoituksen ongelmanratkaisua, kuvauksia järjestelmän käyttäytymisestä ja ominaisuuksista sekä suunnittelu- ja valmistusrajoituksia. Tämä voi aiheuttaa ongelmia, koska suunnittelu- ja valmistusrajoitukset voivat olla ristiriidassa muiden vaatimusten kanssa. Siitä huolimatta se on vaatimusmäärittelyn todellisuutta, joten vaatimusmäärittelyprosessin pitää sisältää toimia löydettyjen ongelmien ratkaisemiseksi.

3.2 Vaatimusten luokittelu ja vaatimustyypit

Vaatimuksia voidaan luokitella usealla eri tavalla. Haikala & Mikkonen (2011, 61.) luokittelevat vaatimukset kolmeen luokkaan:

- Toiminnallinen vaatimus
- Ei-toiminnallinen vaatimus
- Reunaehdot

Robertson & Robertson (2013) luokittelee vaatimukset samalla tavalla kuin Haikala ja Mikkonen eli toiminnallisiin, ei-toiminnallisiin ja rajoituksiin eli reunaehtoihin. Toiminnallinen vaatimus kuvaa toimintoa, joka tuotteen on toteutettava, jotta siitä olisi hyötyä sen käyttäjälle. Lähes mikä tahansa toiminto (laske, tarkasta, julkaise tai useimmat muut aktiiviset verbit) voi olla toiminnallinen vaatimus. Ei-toiminnalliset vaatimukset ovat ominaisuuksia, jotka tuotteella on oltava, jotta sen omistaja ja käyttäjä voivat hyväksyä sen. Joissakin tapauksissa ei-toiminnalliset vaatimukset – jotka määrittävät sellaiset ominaisuudet kuin suorituskyky, ilme ja tuntuma, käytettävyys, turvallisuus ja oikeudelliset ominaisuudet – ovat ratkaisevan tärkeitä tuotteen onnistumisen kannalta. Rajoitukset tai reunaehdot ovat maailmanlaajuisia vaatimuksia. Ne voivat olla rajoituksia itse hankkeelle tai rajoituksia tuotteen mahdolliselle suunnittelulle.

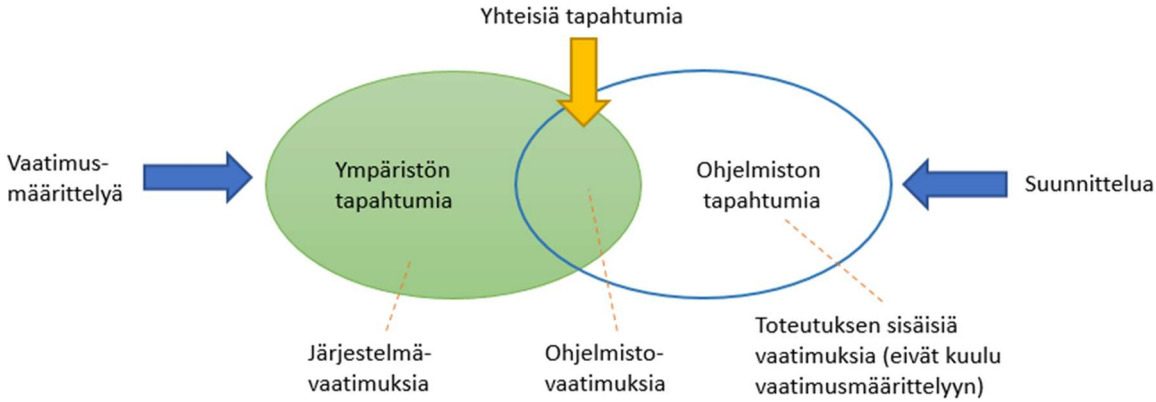
Chemuturi (2011, 13) taas luokittelee vaatimukset ydintoimintoihin (core functionality requirements) ja lisä- tai tukitoimintoihin (ancillary functionality requirements). Ydintoiminnot ovat niitä toiminnallisuuksia, joita ilman tuote ei ole hyödyllinen käyttäjälleen. Nämä toiminnallisuudet pitää toteuttaa, eikä niihin voida tehdä poikkeuksia tai niitä ei voida uudelleen järjestää tuotteen kehittämisen aikana. Ydintoiminnot koskevat liiketoimintaprosessien tehokkuutta ja ohjelmistokehityksen päätarkoitus on toteuttaa nämä ydintoiminnot. Lisätoiminnot taas tukevat ydintoimintoja. Vaikka lisätoiminnot eivät täyty, tuote on silti hyödyllinen, mutta se voi aiheuttaa tuottavuuden menetyksiä tai tuote ei välttämättä ole turvallinen käyttää. Yksi huomionarvoinen asia on se, että asiakas ei useinkaan voi määritellä lisätoimintoja ja jopa odottaa kehitystiimin huolehtivan lisätoiminnallisuudesta. Lisä- tai tukitoimintoja ovat Chemuturin (2013, 15-17) mukaan:

- lakisääteiset toiminnallisuudet, kuten lakiin tai asetuksiin perustuvat vaatimukset tai erilaiset standardit
- turvallisuustoiminnallisuudet, joihin luetaan kuuluvaksi muun muassa logiikkavirheiltä tai tiedon menettämiseltä suojautuminen
- suojaustoiminnallisuudet, jotka suojaavat ulkopuolisilta hyökkäyksiltä
- käytettävyys - tekee ohjelmiston käytöstä intuitiivista eikä käyttäjän tarvitse lukea ohjekirjoja
- tietojen eheyden suojaustoiminnot - estetään käyttäjän mahdollisuudet syöttää virheellistä tietoa, esimerkiksi numerokenttään ei voi syöttää tekstiä
- vasteajat - verkkopohjaisissa sovelluksissa vasteajat ovat tärkeitä, koska käyttäjä ja palvelin eivät välttämättä ole samassa paikassa, ja jos sovellus ei vastaa tarpeeksi nopeasti, käyttäjä voi keskeyttää sovelluksen tai tehdä jotain muuta
- muistirajoitukset - tämä oli tärkeä toiminnallisuus varsinkin aikaisemmin mutta kun ohjelmistot ohjaavat jokaista laitetta tänään, rajoitus säilyy edelleen
- ohjelmiston jalanjälkirajoitus (footprint) - tarkoittaa sirulle asennettavan ohjelmiston kokorajoitusta
- vikasietoisuus - tämä takaa sen, että ohjelmisto ei kaadu tai keskeydy, vaikka käyttäjä tekisikin virheen

- luotettavuus - ohjelmisto pitää toteuttaa niin että erilaiset päivitykset eivät estä ohjelmiston käyttöä tai anna ohjelmistovirhettä, myöskään päivitykset eivät anna ohjelmistovirhettä vaan käyttäjä ymmärtää, että ohjelmisto pitää päivittää
- feel-good toiminnallisuus - käyttöliittymän tulee tuottaa käyttäjälle mielihyvää, käyttöliittymän tulee olla sähäkkä ja seksikäs, käytetään kauniita kuvia ja värejä
- arvostus – saa käyttäjän tuntemaan ylpeyttä; ohjelmistoissa tällainen toiminnallisuus voidaan toteuttaa parempina ilmoituksina ja virhetilanteiden käsittelynä sekä lisäämällä toimintoja, joita mikään muu ohjelmisto ei tarjoa
- kilpailuetu - voi olla ydintoiminnallisuutta tai lisätoiminnallisuus, ohjelmisto omaa enemmän toiminnallisuuksia kuin muut vastaavat ohjelmat.

Paakki (2010) jakaa vaatimukset järjestelmä- ja ohjelmistovaatimuksiin ja luokittelee ohjelmistovaatimukset edelleen toiminnallisiin ja ei-toiminnallisiin vaatimuksiin sekä reunaehtoihin. Vaatimusmäärittelyssä selvitetään, millä ehdoilla tuleva ohjelmisto toimii (toimintaympäristö) ja miten ohjelmisto kommunikoi ulkomaailman kanssa (ohjelmiston ja toimintaympäristön rajapinta).

Toimintaympäristön vaatimukset ovat järjestelmävaatimuksia ja rajapinnan vaatimukset ovat ohjelmistovaatimuksia. Tämä auttaa ymmärtämään myös vaatimusmäärittelyn ja suunnittelun eron koska usein nämä kaksi asiaa saattavat mennä sekaisin tai aiheuttaa väärinymmärryksiä. Järjestelmävaatimus tarkoittaa sellaista toimintaympäristön vaatimusta, jonka tuleva ohjelmisto toteuttaa yhdessä muiden järjestelmän komponenttien kanssa. Ohjelmistovaatimus puolestaan on sellainen toimintaympäristön vaatimus, jonka ohjelmisto toteuttaa yksin. Nämä vaikuttavat välillisesti ongelmakenttään rajapinnan kautta. Ohjelmistovaatimukset ovat myös järjestelmävaatimuksia, koska ne ovat osa ongelmakenttää. Järjestelmävaatimukset eivät usein ole ohjelmistovaatimuksia. Ohjelmistovaatimukset tulevat ohjelmistokehittäjien käyttöön, joten ne on kuvattava heidän ymmärtämällään tavalla. (Paakki 2010.)



Kuva 12. Ohjelmiston toimintaympäristön kaavakuva Paakin (2010) mukaan

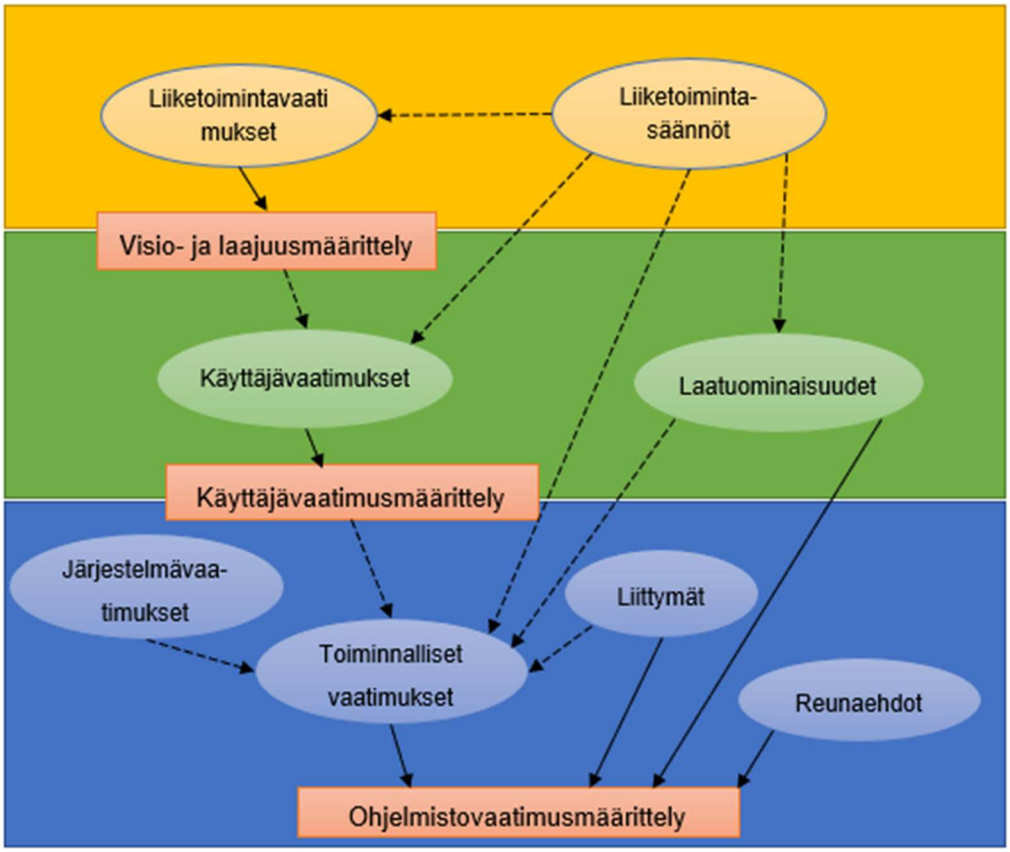
Wiegers & Beatty (2013) luokittelevat vaatimukset tarkemmalla tasolla, koska vaatimustietoja on monenlaisia ja termi ”vaatimus” on heidän mielestään ylikuormitettu (taulukko 2).

Taulukko 2. Vaatimustietotyypit Wiegersin & Beattyn mukaan.

Termit	**Määritelmä**
Liiketoimintavaatimus (Business requirement)	Tuotteen rakentavan organisaation korkean tason liiketoiminnan tavoite; kertoo, miksi järjestelmä toteutetaan.
Liiketoimintasääntö (Business rule)	Käytäntö, suuntaviiva, standardi tai sääntely, joka määrittelee tai rajoittaa jotakin liiketoiminnan osa-aluetta. Ei itsessään ohjelmistovaatimus, vaan monenlaisten ohjelmistovaatimusten alkuperä.
Reunaehto tai rajoitus (Constraint)	Tuotteen suunnittelussa ja toteuttamisessa kehittäjän käytettävissä oleville valinnoille asetettavat rajoitukset tai reunaehdot.
Ulkoinen liittymävaatimus (External interface requirement)	Kuvaus ohjelmistojärjestelmän ja käyttäjän, toisen ohjelmistojärjestelmän tai laitteiston välisestä yhteydestä.
Ominaisuus (Feature)	Yksi tai useampi loogisesti toisiinsa liittyvä järjestelmäominaisuus, joka tuottaa arvoa käyttäjälle ja jota kuvaavat joukko toiminnallisia vaatimuksia.
Toiminnallinen vaatimus (Functional requirement)	Kuvaus käyttäytymisestä, jota järjestelmä osoittaa tietyissä olosuhteissa.
Ei-toiminnallinen vaatimus (Nonfunctional requirement)	Kuvaus ominaisuudesta, joka järjestelmällä pitää olla, tai rajoitus, jota sen on kunnioitettava.
Laatuominaisuus (Quality attribute)	Eräänlainen ei-toiminnallinen vaatimus, joka kuvaa tuotteen palvelua tai suorituskykyä.
Järjestelmävaatimus (System requirement)	Ylätason vaatimus ohjelmistolle, joka sisältää useita alijärjestelmiä; nämä voivat olla joko pelkkiä ohjelmistoja tai sekä ohjelmistoja että laitteistoja.
Käyttäjävaatimus (User requirement)	Tavoite tai tehtävä, jonka tiettyjen käyttäjäluokkien on kyettävä suorittamaan järjestelmällä.

Wiegersin ja Beattyn (2013) mukaan ohjelmistovaatimuksiin kuuluu kolme erillistä tasoa: liiketoiminnan vaatimukset, käyttäjien vaatimukset ja toiminnalliset vaatimukset. Lisäksi jokaisessa järjestelmässä on valikoima ei-toiminnallisia vaatimuksia. Kuva 13 kuvaa tapaa ajatella näitä erityyppisiä vaatimuksia. Kuten tilastotieteilijä George E. P. Box tunnetusti sanoi: "Pohjimmiltaan kaikki mallit ovat vääriä, mutta jotkut ovat hyödyllisiä". Tämä pätee varmasti kuvaan 13. Tämä malli ei ole kattava, mutta se tarjoaa hyödyllisen perustan tunnistaa ja järjestää vaatimuksia.

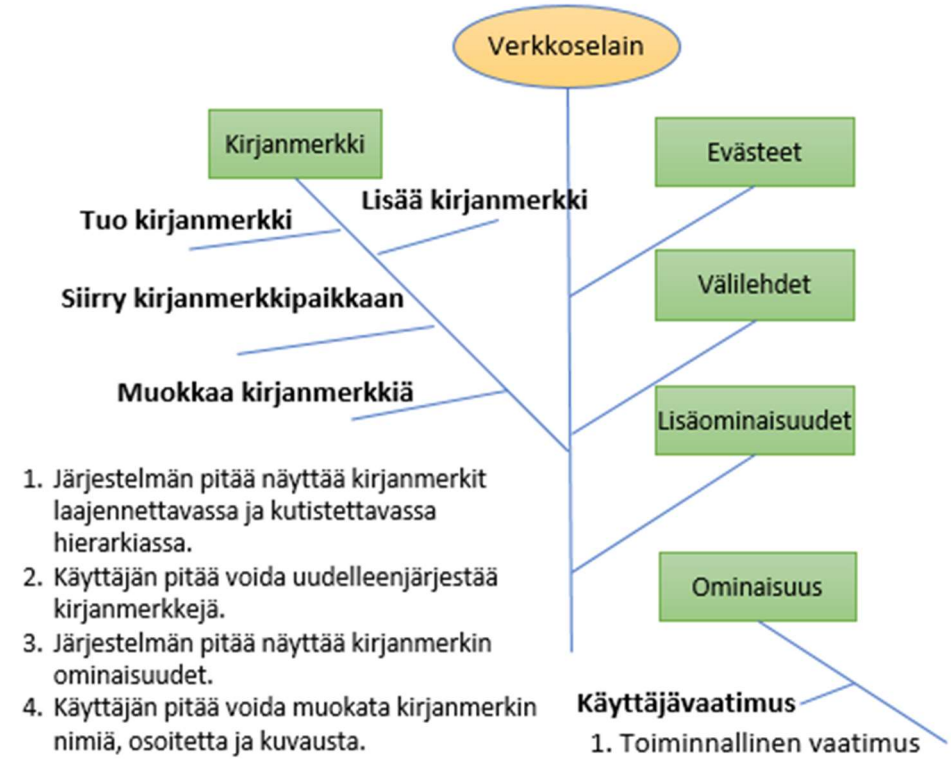
Kuvan 13 ellipsit edustavat vaatimustietotyyppejä, ja suorakulmiot osoittavat asiakirjat tai dokumentit, joihin nämä tiedot tallennetaan. Kiinteät nuolet osoittavat, että tietyntyyppinen tieto tallennetaan tyypillisesti ilmoitettuun määrittelyyn tai asiakirjaan. (Liiketoimintasäännöt ja järjestelmävaatimukset tallennetaan erillään ohjelmistovaatimuksista, kuten vastaavasti liiketoimintasääntöjen luettelossa tai järjestelmävaatimusten määrittelyssä.) Pisteviivat osoittavat nuolet osoittavat, että tietyntyyppiset tiedot ovat toisen tyyppisten vaatimusten alkuperä tai vaikuttavat niihin. Tietovaatimuksia ei esitetä nimenomaisesti tässä kaaviossa. Toiminnot manipuloivat tietoja, joten tietovaatimukset voivat näkyä kaikilla kolmella tasolla.



Kuva 13. Suhteet erilaisten vaatimustyyppien välillä. Kiinteät nuolet tarkoittavat "on tallennettu"; pisteviivat tarkoittavat "alkuperä" tai "vaikutus" (Wiegers ja Beatty 2013)

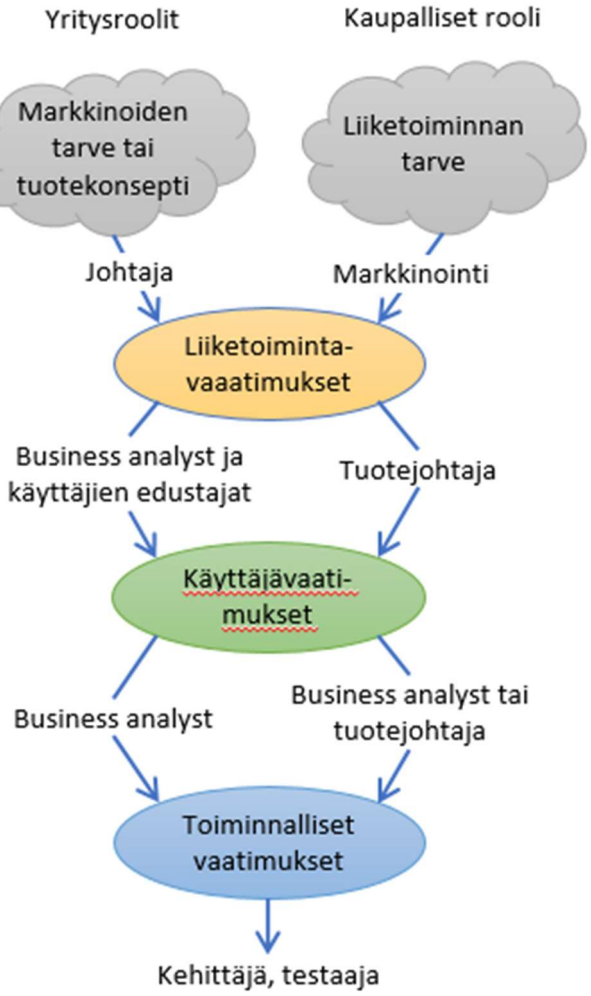
Taulukossa 1 on kuvattu erilaiset vaatimustietotyypit ja niiden selitykset. Haluan tässä kuvata tarkemmin ko. taulukossa esiintyvää vaatimustietotyyppiä, nimittäin ominaisuutta. Ominaisuus koostuu yhdestä tai useammasta loogisesti toisiinsa liittyvästä järjestelmäominaisuudesta, jotka tuottavat arvoa käyttäjälle ja joita kuvaavat joukko toiminnallisia vaatimuksia. Asiakkaan lista toivotuista tuoteominaisuuksista ei välttämättä vastaa käyttäjän tehtäviin liittyvien tarpeiden kuvausta, joten on tarve kuvata nämä ominaisuudet tarkemmalla tasolla. Verkkoselaimen kirjanmerkit, oikeinkirjoituksen tarkistajat, kyky määrittää mukautettu harjoitusohjelma kuntolaitteelle ja automaattinen viruksen allekirjoituksen päivitys haittaohjelmien torjuntatuotteessa ovat esimerkkejä ominaisuuksista. Ominaisuus voi kattaa useita käyttäjän vaatimuksia, joista kukin tarkoittaa, että tietyt toiminnalliset vaatimukset on toteutettava, jotta käyttäjä voi suorittaa kunkin käyttäjän vaatimuksen kuvaaman tehtävän.

Kuva 14 kuvaa ominaisuuspuuta, analyysimallia, joka osoittaa, kuinka ominaisuus voidaan hierarkkisesti hajottaa joukoksi pienempiä ominaisuuksia, jotka liittyvät tiettyihin käyttäjien vaatimuksiin ja johtavat toiminnallisten vaatimusten määrittelyyn (Wiegers & Beatty 2013.)



Kuva 14. Ominaisuuksien, käyttäjien vaatimusten ja toiminnallisten vaatimusten väliset suhteet (Wiegers & Beatty 2013).

Kuva 15 kuvaa, kuinka eri sidosryhmät voivat osallistua kolmen eri tason eli liiketoiminnan vaatimusten, käyttäjien vaatimusten ja toiminnallisten vaatimusten kartoittamiseen. Eri organisaatiot käyttävät eri nimiä näihin toimintoihin liittyville rooleille; kannattaa miettiä, kuka suorittaa nämä toiminnot omassa organisaatiossa. Roolien nimet vaihtelevat usein riippuen siitä, kehittääkö organisaatio ohjelmistoa omaan käyttöön vai kaupalliseen käyttöön (Wiegers & Beatty 2013). En ole tässä työssä kääntänyt Business Analystin merkitystä suomeksi, koska Business Analyst (BA) on yleisesti käytössä oleva nimike myös Suomessa.



Kuva 15. Esimerkki siitä, miten eri sidosryhmät osallistuvat vaatimusten kehittämiseen (Wiegers & Beatty 2013)

Johtajat tai markkinointi määrittelevät tunnistettujen liiketoimintatarpeiden, markkinoiden tarpeiden tai jännittävän uuden tuotekonseptin perusteella ohjelmistoille liiketoiminnan vaatimukset, jotka auttavat yritystä toimimaan tehokkaammin (tietojärjestelmien osalta) tai kilpailemaan menestyksekkäästi markkinoilla (kaupallisista tuotteista). Yritysympäristössä BA työskentelee tyypillisesti käyttäjien edustajien kanssa tunnistaakseen käyttäjien vaati-

mukset. Kaupallisia tuotteita kehittävissä yrityksissä usein tuotepäällikkö on se, joka määrittelee, mitkä ominaisuudet sisällytetään uuteen tuotteeseen. Jokaisen käyttäjän vaatimuksen ja ominaisuuden on oltava yhdenmukainen liiketoiminnan vaatimusten kanssa. Käyttäjätarpeista BA tai tuotepäällikkö johtaa toiminnot, joiden avulla käyttäjät voivat saavuttaa tavoitteensa. Kehittäjät käyttävät toiminnallisia ja ei-toiminnallisia vaatimuksia suunnitellessaan ratkaisuja, jotka toteuttavat tarvittavan toiminnallisuuden rajoitusten asettamissa puitteissa. Testaajat päättävät, miten voidaan varmistaa, että vaatimukset on pantu täytäntöön oikein. (Wiegers & Beatty 2013.)

Taulukko 3. Aurum ja Wohlin (2013) luokittelevat ja ryhmittelevät vaatimuksia seuraavasti:

Toiminnalliset vaatimukset - mitä järjestelmä tekee Ei-toiminnalliset vaatimukset - rajoitukset tai reunaehdot sellaisille ratkaisuille, jotka täyttävät toiminnalliset vaatimukset, esim. tarkkuus, suorituskyky, turvallisuus ja muunneltavuus
Tavoitetason vaatimukset - liittyvät liiketoiminnan tavoitteisiin Toimialuetason vaatimukset - liittyvät ongelma-alueeseen Tuotetason vaatimukset - liittyvät tuotteeseen Suunnittelutason vaatimukset - mitä rakennetaan
Ensisijaiset vaatimukset – saadaan sidosryhmiltä Johdetut vaatimukset - johdetaan ensisijaisista vaatimuksista
Muut luokittelut, esimerkiksi Liiketoimintavaatimukset vs. tekniset vaatimukset Tuotevaatimukset vs. prosessivaatimukset, esimerkiksi liiketoiminnan tarpeet vs. miten käyttäjät käyttävät järjestelmää Rooleihin perustuvat vaatimukset, esimerkiksi asiakasvaatimukset, käyttäjävaatimukset, IT-vaatimukset, järjestelmävaatimukset ja turvallisuusvaatimukset

Vaikka kirjallisuudessa tehdään ero erityyppisten vaatimusten välillä, käytännössä tällaisten erojen tunnistaminen ei ole niin helppoa. Esimerkiksi tietoturvaan liittyvä käyttäjävaatimus voidaan luokitella ei-toiminnalliseksi vaatimukseksi. Toteutuksen aikana siihen voi kuitenkin liittyä muita vaatimuksia, jotka ovat selvästi toiminnallisia vaatimuksia, kuten käyttäjän valtuutus. Kotonya & Sommerville (2004, 4-5) luokittelevat systeemi- tai järjestelmävaatimukset seuraavasti:

- yleiset vaatimukset, jossa määritellään laajasti, mitä järjestelmien tulisi tehdä
- toiminnalliset vaatimukset, jotka määrittelevät osan järjestelmän toiminnallisuudesta

- toteuttamisvaatimukset, jotka ilmaisevat miten ohjelmisto tai järjestelmä tulee toteuttaa
- suorituskykyvaatimukset, jotka määrittelevät esimerkiksi järjestelmän suurimman hyväksyttävän suorituskyvyn
- käytettävyysvaatimukset, joissa esimerkiksi määritetään suurin sallittu aika järjestelmän käytön havainnollistamiseksi

Vaatimukset ovat yleensä aina sekoitus ongelmatietoja, ilmaisuja järjestelmän käyttäytymisestä ja ominaisuuksista sekä suunnittelu- ja valmistusrajoituksia. Tämä voi aiheuttaa vaikeuksia, koska suunnittelu- ja valmistusrajoitukset voivat olla ristiriidassa muiden vaatimusten kanssa. Myös Kotonya & Sommerville (2004) erottavat ei-toiminnalliset vaatimukset omaksi luokakseen, lisäksi he yhdistävät ei-toiminnallisiin vaatimuksiin myös rajoitukset ja reunaehdot.

Kuten voidaan huomata, joidenkin mielestä toiminnalliset vaatimukset kattavat koko joukon sellaisia vaatimuksia, jotka joku toinen asiantuntija määrittelee omiksi vaatimustyypeikseen. Vaatimuksia voidaan luokitella ja tyypitellä hyvinkin eri tavoilla, mutta oleellisinta on mielestä tunnistaa, mitä vaatimuksia ylipäätään on olemassa ja miten liiketoiminnan ongelma pystyttäisiin täten parhaiten ratkaisemaan.

3.2.1 Ei-toiminnalliset vaatimukset

Wiegers ja Beatty (2013) tuovat esiin sen, että monien vuosien ajan ohjelmistotuotetta koskevat vaatimukset on luokiteltu laajasti joko toiminnallisiksi tai ei-toiminnallisiksi. Toiminnalliset vaatimukset ovat ilmeisiä: ne kuvaavat järjestelmän havaittavaa käyttäytymistä eri olosuhteissa. Kuitenkin monet ihmiset eivät pidä termistä "ei-toiminnallinen". Termi kertoo, mitä vaatimukset eivät ole, mutta se ei kerro mitä ne ovat. Voidaankin puhua muista kuin toiminnallisista vaatimuksista, mutta pysytään jatkossa kuitenkin termissä ei-toiminnalliset vaatimukset.

Muut kuin toiminnalliset vaatimukset eivät välttämättä tarkoita, *mitä* järjestelmä tekee, vaan *kuinka hyvin* se tekee nuo asiat. Ne kuvaavat järjestelmän tärkeitä ominaisuuksia tai erityispiirteitä, kuten järjestelmän saatavuus, käytettävyys, suojaus, suorituskyky ja monet muut ominaisuudet. Jotkut pitävät ei-toiminnallisia vaatimuksia synonyyminä laatuominaisuuksille, mutta se on liian rajoittavaa. Esimerkiksi suunnittelu- ja toteutusrajoitukset ovat myös ei-toiminnallisia vaatimuksia, samoin kuin ulkoisen rajapinnan vaatimukset.

Edelleen ei-toiminnalliset vaatimukset koskevat ympäristöä, jossa järjestelmä toimii, kuten alustaa, siirrettävyyttä, yhteensopivuutta ja rajoituksia. Moniin tuotteisiin vaikuttavat myös

määräystenmukaisuus-, sääntely- ja sertifiointivaatimukset. Tuotteille voi olla lokalisointivaatimuksia, joissa on otettava huomioon käyttäjien kulttuurit, kielet, lait, valuutat, terminologia, oikeinkirjoitus ja muut ominaisuudet. Vaikka tällaiset vaatimukset on määritelty ei-toiminnallisilla termeillä, BA:n pitää johtaa näistä tyypillisesti lukuisia toiminnallisuuksia varmistaakseen, että järjestelmällä on kaikki halutut käyttäytymismuodot ja ominaisuudet.

Kotonya & Sommerville ovat omistaneet kokonaisen luvun ei-toiminnallisille vaatimuksille kirjassa Requirements Engineering. Processes and techniques (2004). Heidän mukaansa ei-toiminnalliset vaatimukset määrittelevät tuloksena olevan järjestelmän yleiset ominaisuudet tai piirteet ja ne asettavat rajoituksia käyttäjän vaatimusten täyttämiseksi. Käyttäjä voi asettaa ohjelmistoille rajoituksia, jotka liittyvät rajapintoihin, laatuun, resursseihin ja aikatauluihin. Ei-toiminnalliset vaatimukset ovat usein kriittisen tärkeitä, koska ne ovat järjestelmäpalveluiden rajoituksia tai reunaehtoja, ja toiminnalliset vaatimukset voidaan joutua uhraamaan näiden ei-toiminnallisten rajoitusten täyttämiseksi.

Tietyt ei-toiminnalliset vaatimukset voivat rajoittaa itse kehitysprosessia enemmän kuin itse tuotetta. Ne sisältävät vaatimuksia käytettävistä kehitysstandardeista, metodeista ja ohjelmointikielistä. Asiakkaat yleensä asettavat nämä rajoitteet hyvän laadun saavuttamiseksi sekä ylläpitääkseen niiden metodien yhteensopivuutta, joita käytetään järjestelmän suunnittelussa ja toteuttamisessa. Muut ei-toiminnalliset vaatimukset eivät välttämättä liity suoraan tuotteeseen tai kehitysprosessiin, vaan ne johdetaan ulkoisista rajoituksista yrityksen ulkopuolelta. Näihin kuuluvat oikeudelliset, taloudelliset ja yhteentoimivuusrajoitukset.

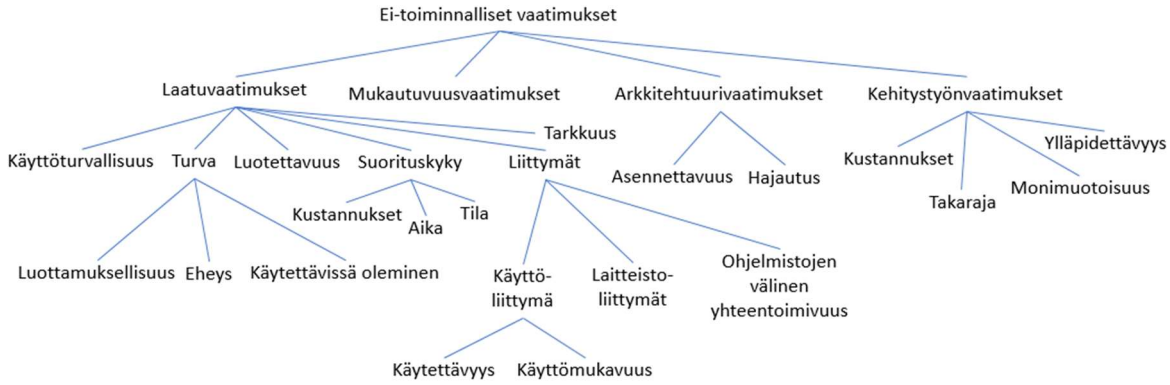
Vaikka onkin usein hyödyllistä erottaa vaatimukset, jotka määrittelevät järjestelmän toiminnallisuuden, ei-toiminnallisista vaatimuksista, on tärkeää ymmärtää, että toiminnallisten ja ei-toiminnallisten vaatimusten välillä ei ole selkeää eroa. Vaatimus ilmaistaan toiminnallisena tai ei-toiminnallisena vaatimuksena riippuen vaatimusten yksityiskohtien tasosta tai järjestelmän asiakkaan ja järjestelmän kehittäjän välisen luottamuksen tasosta. Tämän havainnollistamiseksi tarkastelkaamme vaatimusta järjestelmän turvallisuudesta: järjestelmän on varmistettava, että tiedot suojataan luvattomalta käytöltä. Tavanomaisesti tätä pidetään ei-toiminnallisena vaatimuksena, koska siinä ei määritellä spesifisiä järjestelmän toimintoja, jotka pitää ilmaista. Vaatimus voidaan kuitenkin täsmentää hieman yksityiskohtaisemmin seuraavasti: järjestelmään on sisällyttävä käyttäjän valtuutusmenettely, jossa käyttäjien on tunnistauduttava kirjautumisnimellä ja salasanalla. Vain tällä tavalla valtuutetut käyttäjät voivat käyttää järjestelmätietoja. Tässä muodossa vaatimus näyttää pikemminkin toiminnalliselta vaatimukselta, koska se määrittää toiminnon (käyttäjän sisäänkir-

jautumisen), joka on sisällytettävä järjestelmään. Itse asiassa tämä kuvaa hyvin yleistä tilannetta, jossa abstrakti ei-toiminnallinen vaatimus hajotetaan yksityiskohtaisemmiksi toiminnallisiksi alijärjestelmävaatimuksiksi.

Paakki (2010) jakaa ei-toiminnalliset vaatimukset neljään kategoriaan:

- Laatuvaatimukset (quality (of service), jotka kuvaavat ohjelmiston laatuun liittyviä ominaisuuksia vastaten kysymykseen ”Miten hyvin” (How well). Näillä on yhteys ohjelmiston laatutekijöihin (quality factors, quality attributes).
- Mukautuvuusvaatimukset (compliance), jotka kuvaavat ohjelmiston suhdetta kansallisiin ja kansainvälisiin lakeihin, sosiaalisiin sääntöihin, poliittisiin tai kulttuurisiin rajoituksiin, standardeihin jne.
- Arkkitehtuurivaatimukset (architectural constraint), jotka kuvaavat, miten tuleva ohjelmisto liittyy toimintaympäristöönsä.
- Kehitystyön vaatimukset (development constraint), jotka kuvaavat, mitä ohjelmistotekniikan prosesseja ja menetelmiä on käytettävä, mitkä ovat kehitystyön sallitut kustannukset jne. Nämä eivät liity ohjelmiston käyttöön.

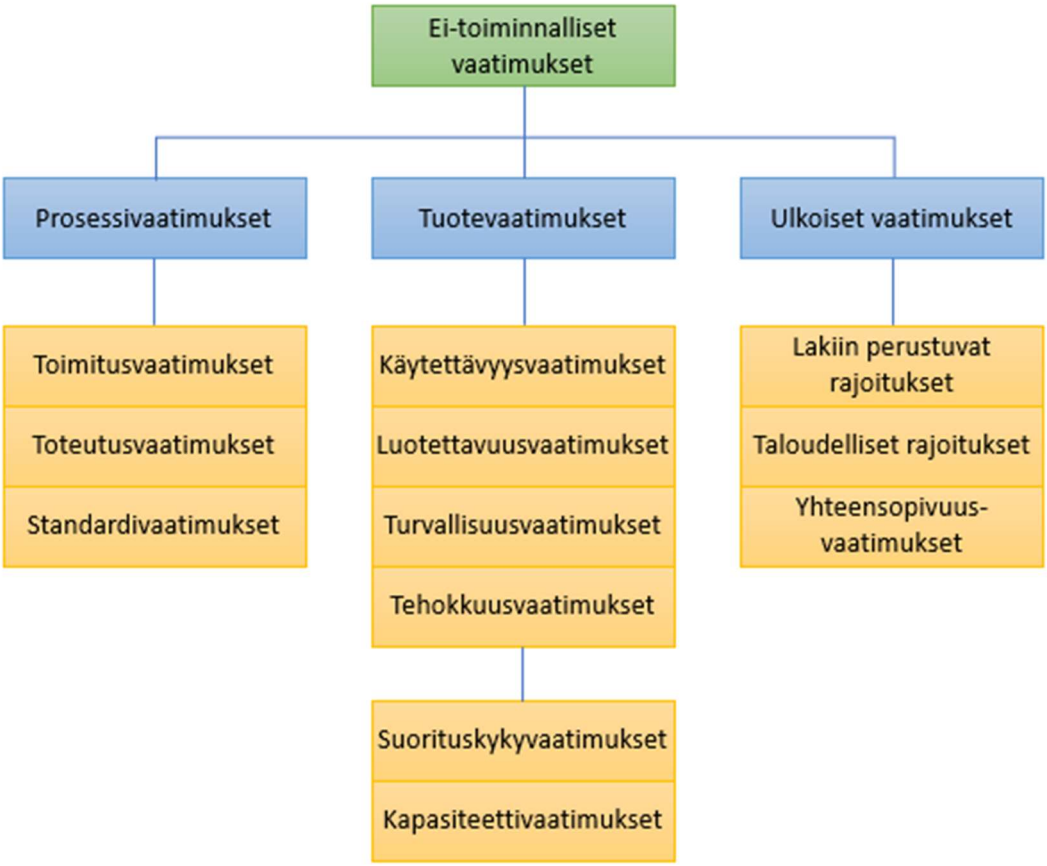
Nämä neljä kategoriaa jaetaan vielä alaluokkiin kuvan 16 mukaisesti.



Kuva 16. Paakin (2010) luokittelu, joka perustuu Axel van Lamsweerden (2009) ei-toiminnallisten vaatimusten luokitteluun.

Kotonya & Sommerville (2004) viittaavat IEEE-Std-830 – 1993 -standardiin, joka listaa 13 erilaista ei-toiminnallista vaatimusta. Näitä ovat toiminnalliset vaatimukset, suorituskykyvaatimukset, liittymävaatimukset, operatiiviset vaatimukset, resurssivaatimukset, todennusvaatimukset, hyväksymisvaatimukset, dokumentointivaatimukset, suojausvaatimukset, siirrettävyysvaatimukset, laatuvaatimukset, luotettavuusvaatimukset, ylläpitovaatimukset ja turvallisuusvaatimukset.

Sommerville itse luokittelee ei-toiminnalliset vaatimukset prosessi-, projekti- ja ulkoisiin vaatimuksiin. Kuvassa 17 näytetään rakenne, joka perustuu näihin luokitteluihin. Käytännössä näiden luokitteluiden välillä ei ole selvää eroa ja monet vaatimukset voidaan helposti luokitella useamman kuin yhden otsikon alle.



Kuva 17. Ei-toiminnallisten vaatimuksien luokittelu Sommervillen mukaan

Vaatimustyypittelystä huomaamme, että siinä missä esimerkiksi Chemuturi luokittelee turvallisuus- ja suojaustoiminnallisuudet omiksi vaatimustyypeiksi, Kotonya ja Sommerville kuten myös Aurum ja Wohlin taas luokittelevat ne ei-toiminnallisiksi vaatimuksiksi. Kirjallisuuden perusteella ei-toiminnallisten vaatimuksien luokittelussa ei ole havaittavissa suuria eroja.

3.2.2 Projektivaatimukset

Tähän mennessä olemme keskustelleet vaatimuksista, jotka kuvaavat rakennettavan ohjelmistojärjestelmän ominaisuuksia. Nämä ovat niin sanottuja tuotevaatimuksia. Projekteilla on kuitenkin varmasti myös muita odotuksia tai toimintoja, jotka eivät ole osa tiimin toteuttamaa ohjelmistoa, mutta jotka ovat välttämättömiä koko projektin onnistuneelle loppuun saattamiselle. Nämä ovat projektivaatimuksia, mutta eivät tuotevaatimuksia. Vaatimusmäärittely sisältää tuotevaatimukset, mutta sen ei tulisi sisältää suunnittelu- tai toteutustietoja (muita kuin tunnettuja rajoituksia), projektisuunnitelmia, testisuunnitelmia tai vastaavia tietoja. Tällaiset kohteet tulisi erottaa niin, että vaatimusten kehittämistoimissa voidaan keskittyä sen ymmärtämiseen, mitä tiimi aikoo rakentaa tai toteuttaa. Wiegersin ja Beattyn (2013) mukaan projektin vaatimukset sisältävät seuraavat asiat:

- Kehitystiimin tarvitsemat fyysiset resurssit, kuten työasemat, erikoislaitteistot, testauslaboratoriot, testaustyökalut ja -välineet, ryhmätilat ja videoneuvottelulaitteet.
- Henkilöstön koulutustarpeet.
- Käyttäjädokumentaatio, mukaan lukien koulutusmateriaalit, oppaat, viitekäsikirjat ja julkaisutiedot.
- Tukidokumentaatio, kuten tukipalvelun resurssit, kenttähuolto- ja huoltotiedot laitteille.
- Toimintaympäristössä tarvittavat infrastruktuurin muutokset.
- Vaatimukset ja menettelytavat tuotteen julkaisemiselle, asentamiselle käyttöympäristöön, konfiguroinnille ja asennuksen testaamiselle.
- Vaatimukset ja menettelyt siirtymisestä vanhasta järjestelmästä uuteen, kuten tietojen siirto- ja muuntamisvaatimukset, tietoturva-asetukset, tuotannon keskeyttäminen ja koulutus osaamisen puutteiden korjaamiseksi; näitä kutsutaan joskus siirtymävaatimuksiksi.
- Tuotesertifiointi ja määräystenmukaisuusvaatimukset (compliance requirements).
- Päivitetyt käytännöt, prosessit, organisaatiorakenteet ja vastaavat asiakirjat.
- Kolmansien osapuolien ohjelmistojen ja laitteistokomponenttien hankinta, hankinta ja lisensointi.
- Beetatestaus-, valmistus-, pakkaus-, markkinointi- ja jakeluvaatimukset.
- Asiakaspalvelutasosopimukset.
- Vaatimukset ohjelmistoon liittyvän henkisen omaisuuden oikeudellisen suojan (patentit, tavaramerkit tai tekijänoikeudet) saamiseksi.

Tässä työssä ei enää käsitellä näitä projektivaatimuksia. Tämä ei tarkoita, että ne eivät ole tärkeitä, vaan että ne eivät kuulu ohjelmistotuotevaatimusten kehittämiseen ja hallintaan. Projektivaatimukset tulevat usein esille tuotevaatimusten yhteydessä ja ne kuuluu tallentaa projektinhallintasuunnitelmaan, jossa on eriteltävä kaikki odotetut projektitoiminnot ja tuotokset.

Erityisesti liike-elämän sovelluksissa ihmiset viittaavat toisinaan "ratkaisuun", joka kattaa sekä tuotevaatimukset (jotka ovat pääasiassa Business Analystien vastuulla) että projektivaatimukset (jotka ovat pääasiassa projektipäällikön vastuulla). He saattavat käyttää termiä "ratkaisun laajuus" viitaten "kaikkeen, mikä on tehtävä projektin loppuun saattamiseksi onnistuneesti".

3.3 Vaatimusmäärittely

Olitpa sitten rakentamassa räätälöityjä järjestelmiä, kokoamassa järjestelmiä komponenteista, käyttämässä kaupallisia valmiita ohjelmistoja, hyödyntämässä avoimen lähdekoodin ohjelmistoja, ulkoistamassa kehitystäsi tai tekemässä muutoksia olemassa oleviin ohjelmistoihin, sinun on silti kartoitettava, löydettävä, ymmärrettävä ja ilmaistava vaatimuksia.

Paakin (2010) mukaan vaatimusmäärittelyllä on prosessi, joka sisältää vaatimusmäärittelyn työvaiheet, toimijat ja tuotokset. Vaatimusmäärittelyn työvaiheet sisältävät tuotoksen tekemiseen tarvittavat tehtävät. Vaatimusmäärittelyn toimijat ovat tulevan järjestelmän tai

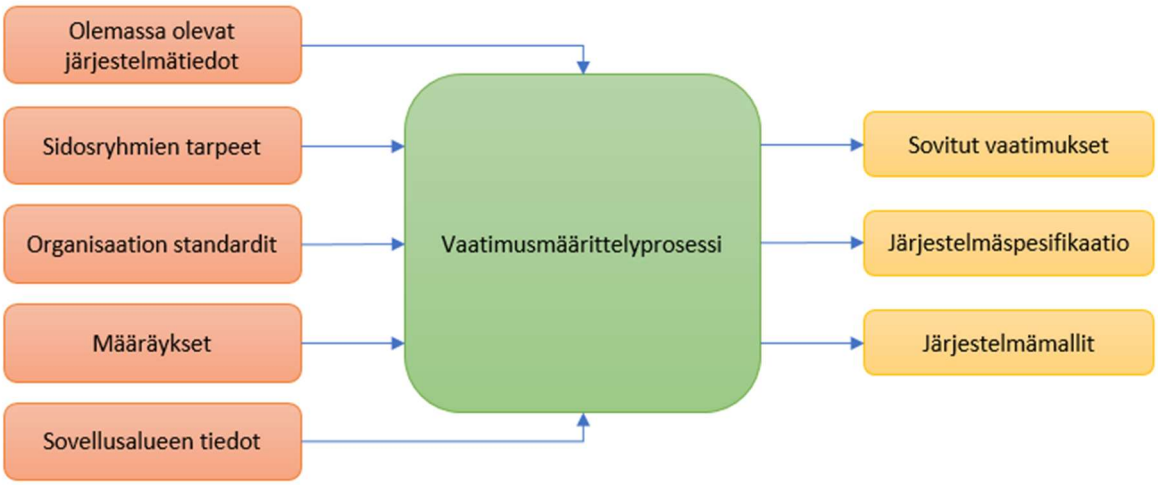
ohjelmiston sidosryhmiä (eng. *stakeholders*). Sidosryhmä tarkoittaa henkilöä tai ryhmää, jonka toimintaan tuleva järjestelmä vaikuttaa, joka on vastuussa tulevan järjestelmän vaatimuksista tai joka osallistuu valmiinjärjestelmän hyväksymiseen. Vaatimusmäärittelyn tuotos on vaatimusdokumentti (eng. *requirements document*). Se kuvaa sekä löydetyt vaatimukset että tulevan ohjelmiston ja toimintaympäristön välisen rajapinnan.

Kotonya & Sommerville (2004) kuvaavat vaatimusmäärittelyprosessia kuvassa 16. Heidän mukaansa vaatimusmäärittelynprosessin syötteitä ovat

- tiedot korvattavan järjestelmän toiminnallisuudesta tai määriteltävän järjestelmän kanssa vuorovaikutuksessa olevista muista järjestelmistä
- kuvaukset järjestelmän sidosryhmien tarpeista, joita he tarvitsevat työnsä tukemiseksi
- standardit, jotka koskevat muun muassa järjestelmän kehittämiskäytäntöjä ja laadunhallintaa
- ulkoiset määräykset, kuten terveys- ja turvallisuusmääräykset, joita sovelletaan järjestelmässä
- yleiset tiedot järjestelmän sovellusalueesta

Vaatimusmäärittelyprosessin tuotoksia puolestaan Kotonyan & Sommervillen (2004) mukaan ovat:

- Sovitut vaatimukset: sidosryhmien ymmärrettävä ja osapuolten hyväksymä kuvaus järjestelmävaatimuksesta
- Järjestelmäspesifikaatio: tämä on tarkempi erittely järjestelmän toiminnallisuudesta, joka voidaan tuottaa joissakin tapauksissa
- Järjestelmämallit: joukko malleja, kuten tietovirtamalli, oliomalli, prosessimalli ja niin edelleen, jotka kuvaavat järjestelmää eri näkökulmista

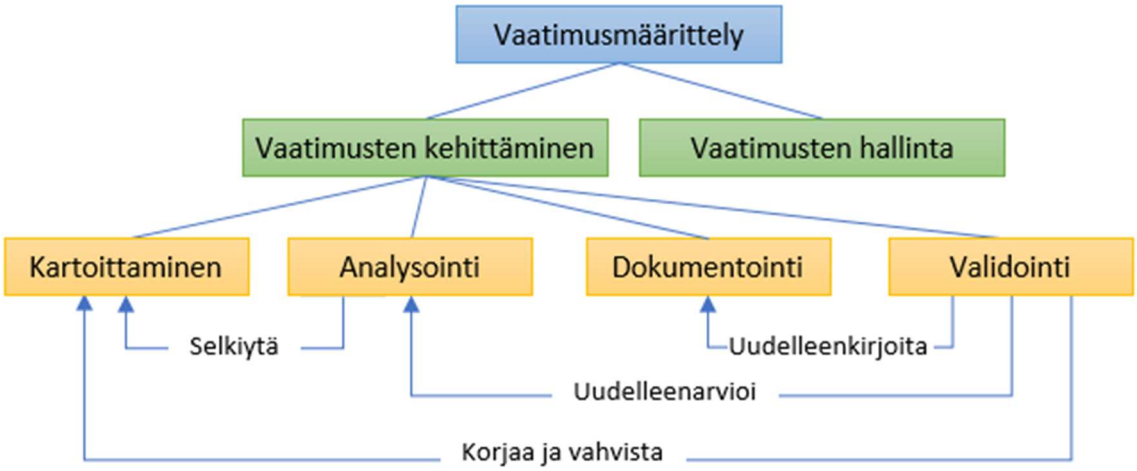


Kuva 18. Vaatimusmäärittelyprosessin syötteet ja tuotokset Kotonyan & Sommervillen (2004) mukaan

Wiegersin (2010) mukaan on hyödyllistä jakaa vaatimusmäärittely (eng. *requirements engineering*) vaatimusten kehittämiseen (eng. *requirements development*) ja vaatimusten hallintaan (eng. *requirements management*) kuten kuvassa 19 on esitetty. Vaatimusten

kehittämisen tavoitteena on tunnistaa, sopia ja kirjata joukko toiminnallisia vaatimuksia ja tuotteen ominaisuuksia, joilla saavutetaan asetetut liiketoiminnan tavoitteet. Wiegers (2010) jakaa vaatimusten kehittämisen kartoittamiseen, analysointiin, dokumentointiin ja validointiin. Vaatimusten hallinnan keskeinen tarkoitus taas on hallita muutoksia sovittuihin vaatimuksiin, jotka liittyvät tiettyyn tuotejulkistukseen. Vaatimusten hallinta sisältää myös yksittäisten vaatimusten tilan seuraamisen ja vaatimusten jäljittämisen sekä taaksepäin niiden alkuperään että eteenpäin suunnitteluelementteihin, koodimoduuleihin ja testeihin.

Wiegers ei tarkoita tässä sitä, että hankkeiden tulisi noudattaa klassista vesiputousta tai peräkkäistä elinkaarilähestymistapaa. Kaikkia tuotteen vaatimuksia ei tarvitse kehittää ennen kuin tuotteen suunnittelun ja rakentaminen aloitetaan. Itse asiassa valmiiden vaatimusten hiominen ja parantaminen saattaa johtaa tavoitteen laajentumisen ja analyysin halvaantumiseen. Todellisuudessa tarpeet, käsitteet ja ymmärrys muuttuvat ajan myötä ja tuotteen kehittyessä. Riippumatta siitä, mitä toteutusmenetelmää projekti noudattaa - olipa se sitten puhdas vesiputous, vaiheittainen, iteratiivinen, inkrementaalinen, ketterä tai jokin hybridi - nämä on tehtävä riippumatta valitusta toteutusmenetelmästä tai tuotteen elinkaaresta. Tiimin on ymmärrettävä kyseisen osan vaatimukset ennen kuin se voi rakentaa sen huolimatta siitä, onko lopullisesta tuotteesta määritelty tietylle julkaisusyklille 1 tai 100 prosenttia.

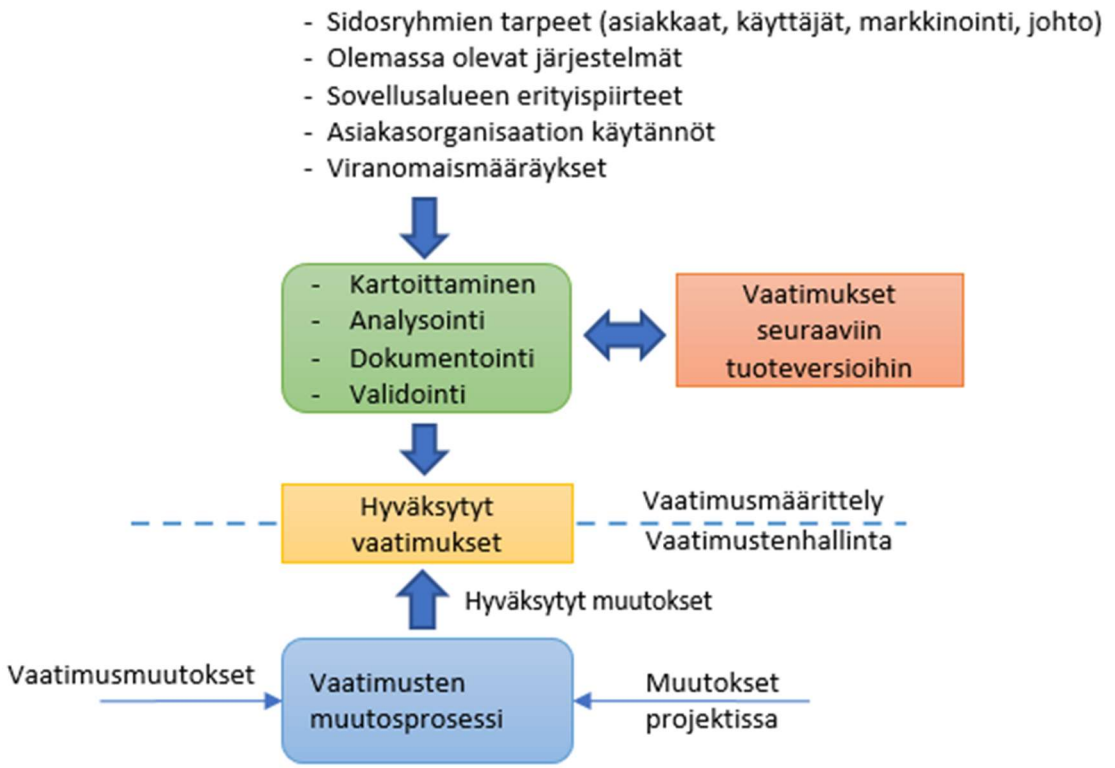


Kuva 19. Vaatimusmäärittely jaettuna eri osa-alueisiin (Wiegers, 2010)

Haikala & Mikkonen (2011, 66) taas käyttävät termiä ”vaatimustenkäsittely” (eng. *requirements engineering*) kattamaan koko prosessin, ja jakavat vaatimustenkäsittelyn vaatimusmäärittelyyn (eng. *requirements definition*) ja vaatimustenhallintaan (eng. *requirements management*). Itse olen tässä työssä käyttänyt vaatimusmäärittely- ja vaatimustenhallintatermejä. Tämä termien monimuotoisuus saattaa hämätä valistunuttakin lukijaa ja siksi

olen tässä työssä pyrkinyt käyttämään sanaa vaatimusmäärittely kuvaamaan koko prosessia. Kyseessä lienee kuitenkin eroavaisuudet termien kääntämisessä ja lähteinä käytetyssä kirjallisuudessa. Käytännössä asiasisältö on kuitenkin sama riippumatta siitä mitä termejä on käytetty.

Haikala & Mikkonen (2011, 66-67) jakavat vaatimusmäärittelyn neljään osaan (kuva 20). Vaatimusten kartoittamisessa tavallisia tapoja ovat muun muassa käyttäjien ja sidosryhmien haastattelut, aivoriihet ja työpajat. Vaatimusten analyysissä tarkennetaan vaatimuksia sekä selvitetään niiden keskinäisiä suhteita ja prioriteettia. Vaatimukset dokumentoidaan sovitulla tavalla esimerkiksi Excel-taulukkoon. Vaatimusten kirjaamisessa on viime vuosina yleistyneet niin sanotut issue tracking-järjestelmät, johon voidaan määritellä dokumenttipohja ja työprosessi, jonka mukaan vaatimus etenee. Vaatimusten validointi tehdään yleensä katselmoimalla vaatimusmäärittelyt yhdessä asiakkaan kanssa.

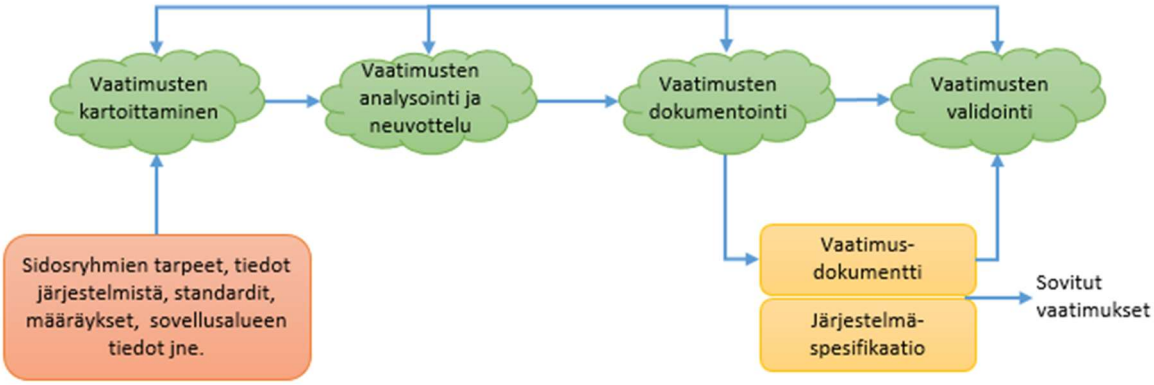


Kuva 20. Vaatimustenkäsittelyprosessi Haikalan & Mikkosen (2011, 66) mukaan

3.4 Vaatimusmäärittelyn osa-alueet

Seuraavaksi perehdymme tarkemmin vaatimusmäärittelyprosessin eri vaiheisiin. Kuvassa 21 kuvataan Kotonyan ja Sommervillen (2004) korkean tason kuvaus vaatimusmäärittelyprosessista. Tässä mallissa toiminnot on kuvattu pilvimuodossa, jotta voidaan osoittaa, että näiden toimintojen välillä ei ole selkeitä rajoja. Käytännössä nämä toiminnot lomittuvat ja niiden välillä tehdään paljon iteraatioita ja annetaan palautetta. Prosessimalli on aina

yksinkertaistettu kuvaus jostain prosessista. Se on yleensä tuotettu jostain tietystä näkökulmasta, joten samasta prosessista voi olla useita erilaisia malleja. Tämä yksi malli ei anna täydellistä ymmärrystä prosessista, mutta se kuvaa vaatimusmäärittelyprosessin perustoiminnot ja niiden suhteellisen järjestyksen. Tämäntyyppinen kuva ei neuvo miten vaatimusmäärittelyprosessi pitää suorittaa tai panna täytäntöön mutta se antaa meille hyvän yleiskuvan.



Kuva 21. Karkean tason vaatimusmäärittelyn prosessimalli Kotonyan ja Sommerville mukaan

Perinteisessä ohjelmistokehityksessä vaatimusmäärittely on usein oma projektinsa, ja siinä pyritään selvittämään asiakkaan vaatimukset mahdollisimman perusteellisesti. Seuraavassa käydään läpi vaatimusmäärittelyprosessia tarkemmin.

3.4.1 Vaatimusten kartoittaminen

Kartoittamisessa (eng. *elicication*) hankitaan tietoa projektin toimialueesta ja vaatimuksista. Toimialuetietojen eri lähteitä ovat asiakkaat, yrityskäsikirjat, olemassa olevat samantyyppiset ohjelmistot, standardit ja projektin muut sidosryhmät. Vaatimusten kartoittamiseen käytettyjä tekniikoita ovat haastattelut, aivoriihi, tehtävien analysointi, Delphi-tekniikka, prototyyppien muodostaminen ja niin edelleen. Kartoittaminen ei tuota muodollisia malleja ymmärretyistä vaatimuksista vaan se laajentaa analyytikon toimialueymmärrystä ja auttaa siten antamaan panosta seuraavaan vaiheeseen.

Wiegersin & Beattyn mukaan (2013) kartoittaminen kattaa kaikki toimet, jotka liittyvät vaatimusten löytämiseen, kuten haastattelut, työpajat, dokumenttien analysointi, prototyyppien luominen ja muut. Avaintoiminnot ovat:

- Tuotteen odotettavissa olevien käyttäjäluokkien ja muiden sidosryhmien tunnistaminen.
- Liiketoiminnan tavoitteiden ja niihin kohdistuvien käyttäjätehtävien ymmärtäminen.
- Uuden tuotteen käyttöympäristön oppiminen.
- Työskentely kutakin käyttäjäluokkaa edustavien henkilöiden kanssa heidän toiminnallisuustarpeidensa ja laatuvaatimustensa ymmärtämiseksi.

Paakin (2010) mukaan tässä työvaiheessa syvennetään tietämystä ongelmakentästä, hankitaan raakatietoa tulevasta järjestelmästä ja sen käyttöympäristöstä sekä etsitään potentiaalisia vaatimuksia:

- Miten uudella teknologialla ja/tai muuttuneilla liiketoimintamalleilla voidaan korjata nykyisen järjestelmän heikkouksia menettämättä sen vahvuuksia.
- Mitä nykyistä järjestelmää parantavia tavoitteita tulevalla järjestelmällä on ja mitä erilaisia vaihtoehtoja tavoitteiden täyttämiseen on.
- Ympäristön ja tekniikan tulevalle järjestelmälle asettamat rajoitteet.
- Mahdolliset vastuunjaot (Kuka-taso).
- Tyypillisiä tulevan ohjelmiston ja sen toimintaympäristön välistä vuorovaikutusta kuvaavia skenaarioita.
- Ongelmakentän ja toimintaympäristön ominaisuudet ja oletukset.
- Edellisten kohtien kanssa yhteensopivat tulevan ohjelmiston vaatimukset.

Kotonyan ja Sommerville (2004) mukaan vaatimukset ovat löydettävissä konsultoimalla sidosryhmiä, tutustumalla olemassa olevaan dokumentaatioon, toimialueosaamisen kautta ja hyödyntämällä markkinatutkimuksista.

3.4.2 Vaatimusten analysointi ja neuvottelu

Tärkeässä analysointivaiheessa korkeamman tason vaatimuksista johdetaan yksityiskohtaisemmat ja tarkemmat vaatimukset. Analysointiin (eng. *analysis*) kuuluu myös useiden erilaisten näkymien tai mallien luominen, kuten prototyypit, graafiset analyysimallit ja testit. Vaatimusanalyysin muita näkökohtia ovat prioriteettineuvottelut, puuttuvien vaatimusten etsiminen ja teknisen toteutettavuuden, riskin ja vikatilojen arviointi. Analysointivaiheessa voidaan tarkentaa ymmärrystä, joka kehittyi vaatimusten kartoittamisen aikana. (Wiegers & Beatty, 2013.)

Kotonyan & Sommerville (2004) mukaan vaatimukset analysoidaan yksityiskohtaisesti ja eri sidosryhmät neuvottelevat keskenään päättääkseen siitä, mitkä vaatimukset hyväksytään. Tämä vaihe on tärkeä, koska eri lähteistä kerätyt vaatimukset saattavat olla ristiriidassa keskenään, saatu tieto voi olla epätäydellistä tai vaatimukset saattavat olla liian kalliita toteuttaa annetussa budjetissa. Yllensä vaatimuksissa on neuvotteluvaraa ja neuvottelu onkin tarpeellista, jotta vaatimukset saadaan hyväksyttyä.

Paakin (2010) mukaan tässä vaiheessa ratkotaan kartoittamisen aikana esille nousseet kysymykset ja ongelmat ja vaatimuksiin liittyvät ristiriidat ratkotaan. Eri näkökulmat ja sidosryhmät voivat nähdä ongelmakentän ja tulevan ohjelmiston eri tavoin, mikä voi johtaa ristiriitaisiin vaatimuksiin tai erilaisiin tulkintoihin. Tulevan järjestelmän riskit kartoitetaan ja

järjestelmälle tehdään riskianalyysi. Kartoitusvaiheessa esille tulleet vaihtoehtoiset ratkaisut arvioidaan ja vaihtoehdoista valitaan paras. Vaatimukset myös priorisoidaan; se helpottaa sekä vaatimusten evaluointia että toteutusaikataulun laadintaa.

3.4.3 Dokumentointi

Dokumentointiin (eng. *specification*) sisältyy erityyppisten vaatimustietojen tallentaminen lomakkeisiin tai asiakirjoihin, jotka helpottavat viestintää projektin sidosryhmien välillä. Perinteisesti nämä dokumentit sisältävät luonnollista kieltä. Myös muut esitystekniikat ovat arvokkaita, kuten graafiset analyysimallit, taulukot ja matemaattiset lausekkeet. "Eritelmä" voi koostua perinteisten asiakirjojen sijasta tietokantaan, kuten kaupallisten vaatimusten hallintatyökaluun, tallennetuista vaatimuksista. (Wiegers 2010.) Hyväksytyt vaatimukset dokumentoidaan asiaankuuluvan yksityiskohtaisesti. Yleisesti ottaen dokumentoinnin pitää olla sen tasoista, että kaikki sidosryhmät sen ymmärtävät. Siksi dokumentoinnissa tulee noudattaa luonnollista kieltä ja ymmärrettäviä graafisia esityksiä. (Kotonya & Sommerville 2004.)

Viime vuosina ketterien menetelmien myötä on yleistynyt niin sanottujen issue tracking – järjestelmien käyttö vaatimusten kirjaamisessa. Vaatimusten hallintajärjestelmään voidaan määritellä vaatimukselle dokumenttipohja sekä työprosessi, jonka mukaan vaatimus etenee (analysointi, toteutus, testaus, valmis ja niin edelleen). (Haikala & Mikkonen 2011, 67.)

Vaatimusmäärittelyn tuloksena syntyy yleensä dokumentti, jota kutsutaan nimellä ”toiminnallinen määrittely”. Se sisältää tavallisesti sekä asiakas- että ohjelmistovaatimukset. Usein määrittelydokumentti voi olla myös ”kaupanteon kohde”, jos halutaan kilpailuttaa toteutusprojekti omana projektinaan. Julkisrahoitteisissa hankkeissa tämä on pakollista EU:n kilpailuttamissääntöjen takia. Muita määrittelyn yhteydessä mahdollisesti syntyviä dokumentteja ovat alustava käyttöohje ja testaussuunnitelma. (Haikala & Mikkonen 2011, 68.)

Tässä vaiheessa tarkennetaan, strukturoidaan ja dokumentoidaan arviointivaiheessa hyväksyttyjä tulevan järjestelmän ominaisuuksia. Tuloksena saadaan vaatimusdokumentti, joka sisältää tulevan järjestelmän tavoitteet, määritelmät, ongelmakentän ominaisuudet, vastuut, järjestelmävaatimukset, ohjelmistovaatimukset ja käyttöympäristöoletukset. Dokumentti voi sisältää myös perusteluja tehdyille valinnoille, hyväksymistestitapauksia ja kustannusarvioita. (Paakki, 2010.)

3.4.4 Validointi

Validoinnilla (eng. *validation*) varmistetaan, että nämä vaatimukset ovat oikeita, täyttävät asiakkaiden tarpeet ja että niillä on kaikki korkealaatuisten vaatimusten ominaisuudet. Validointi voi johtaa siihen, että jotkin vaatimukset kirjoitetaan uudestaan, alkuperäinen analyysi arvioidaan uudelleen tai dokumentoitujen vaatimusten joukkoa korjataan ja tarkennetaan. Validointi tehdään yleensä katselmoimalla vaatimusmäärittelydokumentti yhdessä asiakkaan kanssa. (Wiegers 2010.)

Paakin (2010) mukaan validointivaiheessa vaatimusdokumentista varmennetaan, että tuleva järjestelmä täyttää sidosryhmien tarpeet. Samalla varmistetaan vaatimusdokumentin laatu: spesifikaatiot on kuvattu selkeästi ja yhtenevästi, ne ovat ristiriidattomat ja kattavat ongelmakentän. Vahvistettu vaatimusdokumentti toimii syötteenä kehitystyölle. Vaatimusmäärittelyprosessi on iteratiivinen. Kaikkea ei selvitetä kerralla, vaan ongelmakentän ymmärrys ja vaatimukset tarkentuvat vaiheittain.

3.5 Vaatimusmuutosten hallinta

Kuuluisa, alkuaan kreikkalaisen filosofin Heraclituksen lanseeraama, lentävä lause ”Mikään ei ole niin pysyvää kuin muutos” sopii hyvin myös vaatimusmäärittelyyn. Yksi vaatimusmäärittelyn aliarvostetuimmista näkökulmista on vaatimustenhallinta. Se on prosessi, joka sisältää ajan saatossa muuttuvien vaatimusten hallinnan. Vaatimukset muuttuvat, koska järjestelmäympäristöt kehittyvät ja asiakkaat ymmärtävät paremmin todelliset tarpeensa. Joitakin vaatimuksia on saatettu unohtaa, lainsääntö muuttuu, tulee uusia direktiivejä ja asetuksia tai yritys uudelleenorganisoituu. Uusia vaatimuksia syntyy ja myös olemassa olevat vaatimukset muuttuvat koko ohjelmiston kehittämisen ajan. Usein jopa 50% kaikista vaatimuksista muuttuu ennen kuin ohjelma on saatu tuotantoon. Tämä saattaa aiheuttaa vakavia ongelmia ohjelmistokehittäjille. Jotta nämä vaikeudet voidaan minimoida, tarvitaan vaatimustenhallintaa muutoksien dokumentoimiseen ja kontrolloimiseen (Kotonya & Sommerville 2004, 113.) (Chemuturi, 117.)

Keskeiset teemat vaatimustenhallinnassa ovat Kotonyan ja Sommervillen (2004, 114) mukaan:

- sovittujen vaatimusten muutosten hallinta
- vaatimusten välisten suhteiden hallinta
- vaatimusten välisten riippuvuuksien hallinta.

Vaatimuksia ei voida hallita tehokkaasti ilman vaatimusten jäljitettävyyttä. Vaatimus on jäljitettävissä, jos tiedetään sen tekijä (esittäjä), olemassaolon syy, muut siihen liittyvät vaatimukset ja miten vaatimus liittyy muuhun informaation, kuten suunnitteluun, toteutukseen

ja käyttäjädokumentaation. Jäljitettävyystietoja käytetään, jotta voidaan löytää muut mahdolliset vaatimukset, joihin ehdotetut muutokset voivat vaikuttaa. Hyvällä vaatimustenhallinnalla, kuten riippuvuuksien ylläpitämisellä, on pitkävaikutteisia etuja, kuten suurempi asiakastyytyväisyys ja alhaisemmat ohjelmistokehityskulut.

Haikala & Mikkonen esittävät (2011, 67), että vaatimusmuutosten hallinta on keskeisin prosessi jo siitäkin syystä, että asiakasprojekteissa se on erikseen laskutettavaa työtä ja josta yleensä aina kiistellään, koska vaatimuksia ei ole kuvattu riittävän tarkasti. Vaatimusten muutosprosessin vaiheistukseen kuuluu tyypillisesti muutospyynnön tekeminen, analysointi, testaus ja hyväksyminen. Muutosprosessissa vaatimusta käsitellään samaan tapaan kuin vaatimusmäärittelyssäkin. Muutospyynnöt hyväksytään esimerkiksi projektin johtoryhmässä. Kotonyan ja Sommervillen (2004, 135) mukaan muutoksenhallint politiikoissa tulisi määritellä muutosten hallintaan käytettävät prosessit ja tiedot, jotka tulisi liittää jokaiseen muutospyyntöön. Toimintatavoissa tulisi myös määritellä kuka on vastuussa muutoksenhallintaprosessissa.

Ketterissä menetelmissä vaatimusmuutosten käsittelyä ei oikeastaan ole, koska muutokset käsitellään uusina vaatimuksina tuotteen kehitysjonossa tai viimeistään sprintin suunnittelukokouksessa.

Muuttuneiden vaatimusten kommunikointi projektitiimille on erittäin tärkeää. Muutokset voidaan kommunikoida monella tapaa: puhelimitse, sähköpostilla, käytetyn työkalun kautta (esimerkiksi aikaisemmin mainittu issue tracking-järjestelmä kuten Jira) tai henkilökohtaisesti kasvokkain. Ketterissä menetelmissä suositaan yleensä kasvokkain tapahtuvaa kommunikointia eikä kirjallista dokumenttia muutoksesta välttämättä tehdä. Kuitenkin kirjallisen esityksen hyödyt ovat kiistattomat: kirjallisesti esitetty muutospyyntö auttaa pitämään kirjaa muutoksista ja muutospyyntö on jäljitettävissä eivätkä muutospyynnöt pääse unohtumaan.

3.6 Vaatimusmäärittelyn erityispiirteitä ketterässä ohjelmistokehityksessä

Vaatimismäärittely on jo lähtökohtaisesti erilaista ketterissä menetelmien ohjelmistokehityksessä kuin perinteisissä suunnitelmavetoisissa menetelmissä. Riippumatta mistään erityisestä menetelmästä, ketterä vaatimusten kohtelu on jo pohjimmiltaan erilaista. Huomaamme sen heti perusperiaatteista:
Manifestiperiaate # 1 - Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.

Manifestiperiaate 2 - Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.

Kaiken kaikkiaan ketterien menetelmien vaikutus ohjelmistoteollisuuteen on ollut dramaattinen ja konkreettinen. Sen sijaan, että investoisimme kuukausia kestäviin yksityiskohtaisten ohjelmistovaatimusten määrittelyihin, tiimit keskittyvät toimittamaan varhaisia, lisäarvoa tuottavia tarinoita integroiduksi perustasoksi. Varhainen julkaisu palvelee vaatimusten ja arkkitehtonisten oletusten testaamista ja pienentää riskiä, kun oletukset ominaisuuksien ja komponenttien integroinnista joko hylätään tai hyväksytään.

Ohjausryhmä ja käyttäjäyhteisö eivät enää odota kuukausikaupalla henkeään pidätellen toivoen, että tiimi rakentaa oikeaa asiaa. Pahimmillaankin seuraava tarkistuspiste on vain noin viikon päässä, ja käyttäjät saattavat pystyä ottamaan käyttöön varhaisimmatkin iteraatiot omassa työympäristössään. Ketterissä malleissa voidaan vaatimusmäärittelyä lähestyä paljon joustavammin: prosessi on paljon käytännönläheisempi, vuorovaikutteisempi ja tapahtuu juuri oikea-aikaisesti eikä vaatimuksia kerätä etukäteen odottamaan mahdollista toteutusta. Poissa ovat perinteiset ohjelmistovaatimukset, suunnittelumääritykset ja vastaavat, ja niiden myötä epäsuora sitoutuminen toimittaa "kaikki nämä asiat" kiinteässä aikataulussa ja kiinteiden resurssien perusteella.

Huomionarvoista on, kuinka ketterät periaatteet tunnustavat käsityksen siitä, että vaatimukset muuttuvat koko prosessin ajan. Ketterät periaatteet korostavat myös usein tapahtuvaa henkilökohtaista viestintää (tämä ominaisuus on hyödyllinen myös muiden kuin ohjelmistojärjestelmien suunnittelussa). Ketterien prosessimallien vaatimusmäärittelyn erityispiirteet poikkeavat "perinteisestä" vesiputouksesta ja nykyaikaisemmista malleista, kuten iteratiivisesta tai spiraalikehitysmallista. Nämä muut mallit suosivat paljon ennakkosuunnittelua vaativaa vaatimusmäärittelyprosessia ja usein mittavien teknisten asiakirjojen tuottamista.

Ketterät ohjelmistokehitysmenetelmät ovat osa iteratiivisia menetelmiä, jotka painottavat muutoksen hyväksymistä ja alleviivaavat yhteistyön merkitystä ja laadukkaiden tuotteiden varhaista julkaisua. Työkoodia pidetään kehitysprosessin todellisena artefaktina. Mallit, suunnitelmat ja dokumentaatio ovat tärkeitä ja niillä on arvoa, mutta vain toimivien ohjelmistojen kehittämisen tukemiseksi, toisin kuin muissa jo esiin tulleissa lähestymistavoissa. Tämä ei kuitenkaan tarkoita sitä, että ketterä kehitystapa olisi kaikkien vapaasti sovellettavissa tai käytettävissä. On olemassa hyvin selkeitä käytäntöjä ja periaatteita, jotka ketterien metodologien on omaksuttava.

Ketterät menetelmät ovat pikemminkin mukautuvia kuin ennustavia. Tämä lähestymistapa eroaa merkittävästi aiemmin käsitellyistä malleista, jotka korostavat ohjelmiston yksityiskohtaista ja aikaa vievää suunnittelua ja joiden ohjelmistovaatimusten merkittävät muutokset voivat olla ongelmallisia. Ketterät menetelmät ovat pikemminkin "ihmislähtöisiä" kuin prosessikeskeisiä. Tämä tarkoittaa sitä, että nimenomaisesti halutaan tehdä kehityksestä "hauskaa". Oletettavasti tämä johtuu siitä, että ohjelmistovaatimusten määrittelyjen ja ohjelmistosuunnittelun kuvausten kirjoittaminen on hankalaa ja siten minimoitavaa. (Laplante, 2011.)

Ketteristä menetelmistä on aivan oikein tullut tavanomainen tapa luoda ja toimittaa ohjelmistojärjestelmiä. Useimmissa ketterissä menetelmissä ei määrätä mitään tapaa kerätä vaatimuksia ja muuttaa niitä määrittelyiksi. Tämän aukon täyttämiseksi useimmat analyytikot ja kehittäjät turvautuvat käyttämään käyttäjätarinoita (eng. *user story*) analyysi- ja kehitysprosessin lähtökohtana. Käyttäjätarinat ovat yleensä asiayhteydeltään liian laajoja ollakseen hyödyllisiä ilman valtavaa määrää asiayhteyden suodatusta ja purkamista. Tämä johtaa niin sanottuun käyttäjätarinoiden helvettiin, jossa kehitysjono (eng. *backlog*) koostuu kymmenistä eri tarinoista, jotka kuvaavat kaikkea ei-aktiivisten sidosryhmien toiveista teknisiin rajoituksiin. Tällaisia kehitysjonoja on erittäin vaikea hallita tai priorisoida. (Heath, 2020.)

Paakin (2010) mukaan ketterä ohjelmistokehitys ja vaatimusmäärittely eivät ole ristiriidassa keskenään. Myös vaatimusmäärittelyn spiraali on iteratiivinen prosessi. Ketterissä ohjelmistoprojekteissa jokainen sprintti tai iteraatio aloitetaan suunnittelukokouksella, jossa asiakas ja kehitystiimi sopivat kyseisessä iteraatiossa toteutettavista vaatimuksista. Suuri osa vaatimuksista saadaan aiemmin tuotetusta kehitysjonosta, pieni osa asiakkaalta lisäyksinä kehitysjonoon. Vaatimusmäärittelyn näkökulmasta jokainen spiraalin iteraatio lisättynä lyhyellä toteutusvaiheella vastaa ketterän ohjelmistokehityksen sykliä. Chemuturin (2014, 220) mukaan toinen tunnusmerkki ketterälle vaatimusmäärittelylle Agile Manifestin lisäksi on se, että vaatimuksia tulee koko ajan lisää ja että ne eivät ole koskaan valmiita. Ohjelmistokehityksen päämäärä on täyttää nykyiset vaatimukset, ei jotain oletettuja tulevia vaatimuksia.

Kuten olemme aikaisemmin oppineet ketterin menetelmien arvoista ja periaatteista, ketterä toimii täysin erilaisella ja innovatiivisella tavalla kuin vesiputousmalli. Sen sijaan, että dokumentoitaisiin monimutkaisen järjestelmän kaikki yksityiskohdat, ketterä edellyttää yk-

sinkertaisia ohjeita yksinkertaiselle toiminnolle, ja vaatimusten yksityiskohdat lisätään keskustelujen aikana. Vaatimukset keskittyvät käyttäjän ja asiakkaan tarpeiden tyydyttämiseen, konkreettisen liiketoiminta-arvon tuottamiseen ja yhteistyön rohkaisemiseen. Scrumissa vaatimuksia kutsutaan käyttäjätarinoiksi. Käyttäjätarinan on määritelmänsä mukaan oltava riittävän pieni, jotta se voidaan suunnitella, koodata ja testata sprintissä tai iteraatiossa. Koska sprintin tai iteraation pituus on lyhyt - 2–4 viikkoa -, se pakottaa käyttäjäkertomuksemme yksinkertaisiksi ja ytimekkääksi. (Ashmore & Runyan 2014.)

Mikä on käyttäjätarina?
Mitä: Käyttäjätarina on lyhyt ja yksinkertainen kuvaus pyydetystä ominaisuudesta (tai ominaisuuden osasta).
Kuka: Käyttäjätarina sisältää sen henkilön näkökulman, joka käyttää pyydettyä ominaisuutta tai hyötyy siitä.
Miksi: Käyttäjätarina sisältää ominaisuuden arvon, jotta tiimi voi ymmärtää, mikä ajaa tätä pyyntöä.
Milloin: Käyttäjätarina ei tosiasiallisesti määritä aikakehystä, mutta käyttäjäkertomuksen priorisointiin perustuu aikakäsitys: Ne, jotka ovat aikasidonnaisempia tai jotka lisäävät liiketoiminnan arvoa, asetetaan listalla etusijalle.

Käyttäjätarinan muoto on hyvin tarkka ja sen käyttäjäystävällisen lauserakenteen avulla voidaan kerätä arvokasta tietoa. Tässä on perusmuoto: <Käyttäjänä> haluan, että <jokin tavoite>, jotta <jokin syy>. Esimerkiksi ”Maksajana haluan, että voin tallentaa saajan tiedot, jotta voin käyttää niin helposti uudelleen”. Nämä datapisteet ovat kaikki merkittäviä ja hyödyllisiä; kukin niistä sallii tuotteen omistajan ilmaista vaatimuksen osan, ja se kehottaa vuoropuheluun ja yhteistyöhön kehittäjien kanssa. (Ashmore ym. 2014.)

Kuka, mitä ja miksi luovat tehokkaan yhdistelmän, joten tiimillä on paljon enemmän mistä jatkaa tekemistä kuin perinteisestä vaatimusten dokumentointitavasta. Toinen arvokas näkökohta käyttäjätarinoissa on se, että se rohkaisee keskusteluun tai neuvotteluihin. Ei ole enää tilannetta, jossa tuotteen omistajan on tiedettävä kaikki vaatimukset etukäteen ja toimitettava kumoamattomat asiakirjat halutusta ominaisuudesta. Nyt tiimi ja tuotteen omistaja pyrkivät yhdessä löytämään parhaan tavan vastata tarinassa ilmaistuihin tarpeisiin. Käyttäjätarina ei ole määrittely, vaan viestintä- ja yhteistyötyökalu. Tarinoita ei pidä luovuttaa kehitystiimille, vaan niitä on täydennettävä keskustelulla.

Mitä tulee hyvien käyttäjätarinoiden kirjoittamiseen, niin Bill Wake (Ashmore 2014) mainitsi ensimmäisen kerran lyhenteen INVEST vuonna 2003, ja sitä käytetään edelleen käyttäjätarinoiden tehokkuutta mitattaessa. Tuotteiden omistajien on investoitava hyvään käyttäjäkertomukseen eli hyvä käyttäjätarina täyttää seuraavat kriteerit:

- Itsenäinen (Independent) - Käyttäjätarinan on oltava itsenäinen. Sen on oltava ominaisuus tai ominaisuuden osa, joka voidaan testata ja toteuttaa ainutkertaisena elementtinä. Käyttäjätarinoiden ei tulisi mahdollisuuksien mukaan olla riippuvaisia muista toiminnoista. Ihannetapauksessa ne on kirjoitettu niin, että ne voidaan toimittaa missä tahansa järjestyksessä.
- Neuvoteltavissa (Negotiable) - Kuten mainittiin, käyttäjätarinan tulisi rohkaista yhteistyöhön ja keskustelemaan parhaasta tavasta ratkaista esitetty ongelma. Tiimin, agile masterin ja tuoteomistajan on oltava avoimia keskustelemaan käytettävissä olevista vaihtoehdoista.
- Arvokas (Valuable) - Syy siihen, miksi tehdään mitään ketterästi, on lisätä liiketoiminnan arvoa, ja mitä enemmän liiketoiminnan arvoa tuotetaan, sitä korkeampi tarinan prioriteetti. Jos tarina ei tuota liiketoiminnalle lisäarvoa, tiimin ei pitäisi työskennellä sen parissa.
- Arvioitavissa (Estimatable) - Onko tarina liian iso vai liian epämääräinen? Sen on oltava riittävän selkeä, jotta kehittäjät ja testaajat voivat kohtuudella arvioida toimituksen monimutkaisuuden ja keston.
- Pieni (Small) - Tarinan on oltava riittävän pieni, jotta se voidaan suorittaa yhden sprintin aikana.
- Testattava (Testable) – Tarina on ominaisuuksiltaan riittävä, ja se on kirjoitettu siten, että se voidaan testata, jotta voidaan varmistua, että se toimii odotetulla tavalla.

Useimmat käyttäjätarinat eivät ala käyttäjäkertomuksina; ne alkavat epiceinä eli aihioina. Aihio (epiciä harvoin suomennetaan, mutta käytän tässä työssä nimeä aihio) on yksinkertaisesti käyttäjätarina, joka on liian suuri suunniteltavaksi, koodattavaksi ja testattavaksi yhdessä sprintissä. Suurin osa vaatimuksista alkaa aihioina, koska meillä on idea, jota alamme kuvata ensimmäisessä käyttäjätarinassa. Kun teemme yhteistyötä tiimin ja käyttäjien kanssa, huomaamme, että ominaisuudella on monia eri puolia. Kuten sanotaan, "piru asuu yksityiskohdissa", ja kun näistä yksityiskohdista keskustellaan, niin aihio jakautuu lukuisiin lapsitarinoihin. Tämä on tyypillinen vanhempi – lapsi -suhde, ja yhdestä aihiosta syntyy monia lapsitarinoita. Tämä ei ole huono asia: on aloitettava suuresta tai jopa pienestä ideasta, ennen kuin voidaan löytää kaikki yksityiskohtaiset ratkaisut, jotka tarvitaan kyseisen idean toteuttamiseksi. (Ashmore ym. 2014.)

4 Tutkimusmenetelmät

Tässä luvussa kerrotaan tarkemmin tämän tutkimuksen tutkimusstrategiasta sekä tutkimus- ja tiedonkeruumenetelmistä sekä analyysimenetelmistä. Tutkimuksen tulokset esitetään luvussa 5.

Tutkimuksen tavoitteena on selvittää, miten teoriaosuudessa kuvatut erilaiset vaatimustyypit ja vaatimusmäärittelyn osa-alueet käytännössä toimivat palveluna hankitun tietojärjestelmän hybridimallisessa ohjelmistokehittämisessä, jossa tilaaja-toimittajaorganisaatio on hajautettu ympäri maailmaa. Tutkimuksen avulla pyritään myös selvittämään, millaisia hyötyjä ja haasteita kohdataan päivittäisessä työssä projektissa, jonka kunnianhimoisena tavoitteena on uudistaa sekä liiketoimintaprosessit että korvata osa vanhoista legacy-järjestelmistä uudella tietojärjestelmällä.

4.1 Tutkimuksen lähestymistapa

Tutkimuksen lähestymistavaksi valitsin laadullisen eli kvalitatiivisen tutkimuksen. Laadullisessa tutkimuksessa pyritään tyypillisesti ymmärtämään tutkimuksessa tarkasteltavaa ilmiötä tutkimuksen kohteina olevien henkilöiden näkökulmasta. Tämä tarkoittaa sitä, että ollaan kiinnostuneita tutkimuksen kohteena olevien henkilöiden kokemuksista, ajatuksista, tunteista ja niistä merkityksistä, joita ihmiset tutkimuksen kohteena olevalle asialle antavat. Koska on vaikea päästä toisen henkilön kokemusmaailmaan sisälle ja kokea asiat sellaisena kuin hän ne kokee, on kehitetty erilaisia menetelmiä, joiden avulla pyritään helpottamaan tutkimuksen tekemistä. Menetelmillä tarkoitetaan niitä keinoja, joiden avulla tutkimuksen aihetta lähestytään ja pyritään vastaamaan tutkimuskysymyksiin. (Puusa, Juuti & Aaltio, 2020.)

Puusa ym. (2020) mukaan aloittelevaa tutkijaa saattaa tieteen kieli ja käsitteiden kirjo joskus hämmentää. Menetelmän lisäksi käytetään sellaisia termejä kuin tutkimuksen lähestymistapa, tutkimusote, tutkimusstrategia ja metodologia. Menetelmällä yleensä viitataan erilaisiin konkreettisiin tapoihin kerätä aineistoa tutkimusta varten. Menetelmät liittyvät niin aineiston hankintaan kuin erilaisiin aineiston analyysimenetelmiinkin. Tutkimuksen lähestymistapa viittaa usein siihen, onko tutkimus laadullinen, määrällinen, empiirinen tai ei-empiirinen. Tutkimusote taas kertoo täsmällisemmin, millainen tutkimus on. Tutkimus voi olla esimerkiksi laadullinen, narratiivinen tutkimus tai laadullinen etnografinen tutkimus. Tutkimusotteen sijaan näkee samasta asiasta käytettävän myös nimitystä tutkimusstrategia. Metodologia taas pitää sisällään tutkijan oletukset tutkimuskohteen luonteesta ja olemuksesta sekä käsityksen siitä, kuinka ilmiöstä on mahdollista saada tietoa.

Laadullisessa tutkimuksessa käytetään aineiston hankinnassa tavanomaisesti yksilöhaastatteluja, ryhmähaastatteluja, dokumenttiaineistoja tai havainnointia eri muodoissaan. On myös tavanomaista yhdistellä eri aineistonkeruumenetelmiä toisiinsa. (Puusa ym. 2020.)

Usein laadullista tutkimusta lähdetään jäsentämään vertaamalla sitä määrälliseen tutkimukseen, vaikka ne eivät välttämättä ole toistensa vastakohtia. Tärkein ero verrattaessa laadullista tutkimusta määrälliseen on se, että määrällinen tutkimus pitää sisällään oletuksen siitä, että kohde teoriasta ja tutkijasta riippumaton. Tutkimuksen lähestymistapa, laadullinen ja määrällinen, ovat vain välineitä, eivät arvoja tai tavoitteita sinänsä. (Puusa ym.2020.) Metsämuurosen (2006, 87) mukaan myös kvantitatiivisessa eli määrällisessä tutkimuksessa voidaan käyttää menetelmiä, jotka perinteisesti luetaan kuuluviksi kvalitatiiviseen eli laadulliseen tutkimukseen.

Käytettävien menetelmien tarkastelu ei aina auta tunnistamaan kummasta tutkimusotteesta on kysymys. Sen sijaan sekä tutkimusten tavoite että tutkimusotteiden aineistot ovat keskenään laadullisesti erilaisia. Määrällisen tutkimuksen tulokset ovat numeerisessa muodossa, kun taas laadullisen tutkimuksen aineistot ovat pääosin erilaisia tekstejä. Määrällisen tutkimuksen tavoitteet ilmaistaan usein testattavien hypoteesien muodossa, kun taas laadullisen tutkimuksen tavoitteet ovat usein kuvailevia. (Puusa ym. 2020.)

Puusa ym. (2020) mukaan laadullinen lähestymistapa korostaa todellisuuden ja siitä saatavan tiedon subjektiivista luonnetta. Tämä on yksi laadullisen tutkimuksen keskeinen tunnuspiirre. Laadullinen tutkimus keskittyy tarkastelemaan yksittäisiä tapauksia ja olennaista on osallistuvien henkilöiden näkökulma ja tutkijan vuorovaikutus yksittäisen havainnon kanssa.

Taulukkoon 4 on tiivistetty vertailu laadullisen ja määrällisen metodologian eroista laadullisen tutkimustradition oleellisten tutkimusmetodien kannalta

Taulukko 4. Tutkimusmetodit eri metodologioissa (Metsämuuronen 2006).

Metodi	**Metodologia**	
	Kvantitatiivinen tutkimus	Kvalitatiivinen tutkimus
Havainnoiminen	Alustavaa työtä esimerkiksi varsinaista lomaketta varten	Perustava menetelmä toisen kulttuurin ymmärtämisessä
Tekstianalyysi	Kvantitatiivinen sisällön analyysi; tutkijan asettaminen kategorioiden laskeminen	Kulttuurin jäsenten käyttämien kategorioiden ymmärtäminen

Haastattelu	”Survey-tutkimus” Strukturoitujen valintakysymysten esittäminen satunnaisesti valitulle otokselle.	”Avoimien kysymysten” esittäminen valituille yksilöille tai ryhmille.
Litterointi	Harvoin käytössä; esimerkiksi tarkistettaessa haastatteluäänityksen paikkansapitävyys	Käytetään sen ymmärtämiseen, kuinka tutkittavat organisoivat puheensa

4.2 Tutkimusstrategia

Tutkimusstrategiaksi valitsin tapaustutkimuksen, jonka aineistonkeruumenetelmänä toimii sekä laadullinen haastattelututkimus että omakohtainen havainnointi.

Tapaustutkimus on yksi laadullisen tutkimuksen tiedonhankinnan strategia. Tapaustutkimus eli case study voidaan määritellä empiiriseksi tutkimukseksi, joka monipuolisia ja monilla tavoilla hankittuja tietoja käyttäen tutkii nykyistä tapahtumaa tai toimivaa ihmistä tietyssä ympäristössä (Metsämuuronen 2006, 91). Toisaalta tapaustutkimus on määritelty myös yksinkertaisesti toiminnassa olevan tapahtuman tutkimukseksi. Metsämuurosen (2006, 91) mukaan tapaustutkimuksen luonteeseen kuuluu se, että tutkittavasta tapauksesta pyritään kokoamaan monipuolisesti ja monella tapaa tietoja. Pyrkimyksenä on ymmärtää ilmiötä yhä syvällisemmin. Puusa ym. (2020) määrittelevät tapaustutkimuksen olevan tutkimusstrategia, joka mahdollistaa ilmiön tarkastelun sen omassa luonnollisessa kontekstissa käyttäen useita tietolähteitä. Tapaustutkimuksen tavoitteena on tuoda teoria kosketuksiin empiirisen maailman kanssa.

Metsämuurosen (2006) mukaan Cohen & Manion ovat siteeranneet Adelmania ja hänen kollegoitaan (1980) ja kirjanneet ylös, millaisia mahdollisia etuja tapaustutkimuksesta on:

- Tapaustutkimuksen aineisto on paradoksaalisesti ”voimakkaasti totta”, mutta vaikeasti organisoitavissa. Tämä johtuu siitä, että tapaustutkimus on ”jalat maassa” -tutkimusta, joka perustuu tutkittavan omiin kokemuksiin. Näin ollen tapaustutkimis tarjoaa luonnollisen pohjan yleistämiselle.
- Tapaustutkimus sallii yleistämiset.
- Tapaustutkimuksella huomataan sosiaalisten totuuksien monimutkaisuus ja sisäkkäisyys. Parhaat tapaustutkimukset pystyvät tarjoamaan tukea vaihtoehtoisille tulkinnoille.
- Tapaustutkimukset tuotoksina muodostavat kuvailevan materiaalin arkiston, josta voidaan tehdä erilaisia tulkintoja.
- Tapaustutkimukset ovat usein ”askel toimintaan”. Niiden lähtökohta on usein toiminnallinen ja niiden tuloksia myös sovelletaan käytännössä.
- Tapaustutkimuksen raportointi on mahdollista tehdä kansantajuiseksi ja siinä on mahdollista välttää tavanomaiselle tutkimukselle tyypillistä sisäänpäin lämpiävää

tiedeslangia. Tapaustutkimus voi siis palvella monenlaista lukijakuntaa. Tapaustutkimusraportti sallii lukijan tehdä omia johtopäätöksiä tutkimuksen tuloksista.

-

Tapaustutkimukseen liittyvä keskeinen kysymys onkin: mitä voimme oppia yhdestä tapauksesta? Tapaustutkimus voivaan ymmärtää keskeiseksi kvalitatiivisen metodologian tiedonhankinnan strategiaksi, sillä lähes kaikki strategiat käyttävät lähestymistapanaan tapaustutkimusta. Tosin sanoen lähes kaikki kvalitatiivinen tutkimus on tapaustutkimusta. Muita yleisempiä laadullisen tiedonhankinnan strategioita ovat etnografia, fenomenografia, Grounded Theory ja toimintatutkimus. (Metsämuuronen 2006, 92.)

4.3 Aineiston hankintamenetelmät

Kussakin kvalitatiivisen tutkimuksen strategioissa voidaan käyttää yhtä tai useampaa metodia, näitä ovat muun muassa haastattelu, tarkkailu sekä kirjallisen materiaalin käyttö. Kullakin menetelmällä on omat ominaispiirteensä, vahvuutensa ja myös rajoitteensa. (Metsämuuronen 2016, 111.)

Haastattelu voidaan tehdä monella tavalla: yksilöhaastattelu kasvoista kasvoihin, ryhmähaastattelu kasvoista kasvoihin, postitettu tai paikan päällä kerätty lomakehaastattelu tai puhelimitse tehty haastattelu. Jotkut eivät pidä kyselylomaketutkimusta haastatteluna lainkaan. Haastattelu voi olla puolistrukturoitu, strukturoitu tai avoin. Haastattelu voi kestää viidestä minuutista useisiin päiviin. Haastattelua voidaan pitää tarkkailun ohella eräänlaisena perusmenetelmänä, joka soveltuu moneen tilanteeseen. Aina kun haastattelu on mielekäs tapa hankkia tietoa, sitä kannattaa käyttää, vaikka se onkin melko työläs ja jatkoanalyysin kannalta vaatelias. (Metsämuuronen 2006, 111-114.)

Haastattelun metodinen etu on, että haastateltavaksi voidaan valita henkilöitä, joilla etukäteen tiedetään olevan kokemusta tutkittavasta ilmiöstä tai tietoa aiheesta. Silloin puhutaan tarkoituksenmukaisesta, harkinnanvaraisesta näytteestä. Haastattelussa on keskeistä pyrkiä saamaan mahdollisimman paljon tietoa halutusta asiasta ja monipuolinen kuva kiinnostuksen kohteena olevasta ilmiöstä. (Puusa ym., 2020.)

Havainnoinnissa tai tarkkailussa on kyse siitä, että tutkija tarkkailee enemmän tai vähemmän objektiivisesti tutkimuksen kohdetta ja tekee havainnoinnin aikana muistiinpanoja tai kenttäraporttia. Havainnointi voidaan jakaa perinteisesti neljään eriasteiseen osallistumiseen: havainnointi ilman varsinaista osallistumista, havainnointi osallistujana, osallistuja havainnoijana ja täydellinen osallistuja (Metsämuuronen 2006, 116). Havainnoinnin etuna on erityisesti autenttisuus eli se, että havainnoimalla on mahdollista päästä seuraamaan todellisia tilanteita reaaliaikaisesti, tässä ja nyt. Havainnoinnin avulla on lisäksi mahdollista

tarkastella ilmiöitä prosessina ja pidemmän ajan kuluessa. Havainnoin etuna on myös se kokonaisvaltaisuus, koska saatu tieto on suoraan kytkettävissä asiayhteyteensä. Havainnoinnin avulla tutkija voi myös todentaa, miten haastattelussa kerrottu tai dokumenttien kautta ilmaistu asia toteutuu käytännössä. (Puusa ym. 2020.)

Tutkimuksen empiireissä osan eli tapaustutkimuksen aineistonkeruumenetelmänä olivat haastattelut ja havainnointi projektiin osallistujana. Haastattelu toteutettiin avoimena ja anonyyminä lomakehaastatteluna. Lisäksi tämän opinnäytetyön tekijänä havainnoin vaatimusmäärittelyprosessia ja sen käytäntöjä puolen vuoden ajan vuoden 2020 marraskuusta alkaen.

4.4 Aineisto

Tutkimuksessa tein hakuja Haaga-Helian kirjastoon ja tietokantoihin sekä indeksointipalveluihin, kuten Google Scholariin. Lukujen 2 ja 3 kirjallisuuslähteenä käytin sekä ohjelmistotuotantoon (Software Development, Software Engineering) että vaatimusmäärittelyyn (Requirements Enginering) keskittyviä teoksia. Osa teoksista on noin kymmenen vuoden takaa, osa jopa vanhempia, mutta Haaga-Helian kirjastossa ei ollut paljonkaan tätä uudempia teoksia e-kirjoina saatavilla. Paperikirjat olivat pitkälti lainassa ja korona-ajan takia laina-aikoja pidennettiin niin, että kirjoja oli vaikea saada. Korona-aika vaikeutti selvästi myös tutkimustapoihin liittyvän kirjallisuuden lainaamista, koska niissäkin laina-ajat olivat pitkiä ja e-kirjojen laina-ajat jatkuvasti täynnä. Luultavasti kevät oli muutenkin suosittua aikaa opinnäytetyön tekijöiden keskuudessa ja se osaltaan vaikeutti lähdekirjallisuuden hankintamahdollisuuksia. Suurin osa kirjastoista oli keväällä 2021 kiinni tai ainakin lyhennetyin aukioloajoin, joka osaltaan lisäsi e-kirjojen suosiota ja pidensi paperikirjojen laina-aikoja.

Tapaustutkimus koostuu ohjelmistokehitysprojektista, jossa ensimmäinen julkaisu palveluna ostetusta tietojärjestelmästä oli tarkoitus ottaa käyttöön ketterin periaattein toteutetun pienimutoisen kehitystyön jälkeen. Ensimmäinen julkaisu oli niin sanottu MVP eli Minimun Viable Product (pienin toimiva tuote). Wikipedian (2021e) määritelmän mukaan pienin toimiva tuote on uusi, markkinoille vasta yrittävä tuote tai palvelu, jossa on vain sellaiset ominaisuudet, jotka tyydyttävät varhaisten asiakkaiden tarpeet. Pienimmän toimivan tuotteen tarkoitus on kerätä palautetta tuotekehitykseen. Joidenkin asiantuntijoiden mukaan olennaista on luoda tuote, jota voidaan jo myydä, vaikka se ei täyttäisi läheskään suunnittelijoiden visiota lopullisesta tuotteesta.

Empiirisessä osassa on tarkoitus testata ja peilata käytäntöön tutkimuskirjallisuudessa esitettyjä teorioita, miten vaatimusmäärittelyä tehdään ketterässä ohjelmistokehityksessä, kun ohjelmisto on ostettu palveluna ja onko vaatimusmäärittely yleensä ollenkaan tarpeen, kun on hankittu valmis tuote tai palvelu.

5 Tutkimustulokset

Tässä luvussa käydään läpi tutkimustulokset. Kuvaan ensin itse projektia, projektiryhmää, projektin työskentelytapoja ja projektin vaatimusmäärittelyprosessia, jotta lukija saa käsityksen tilanteesta, jota tutkimuksen käsittelee. Itse tutkimustuloksissa etsin vastausta tutkimuskysymyksiini eli siihen, miten vaatimusmäärittely toteutuu projektissa, mitkä ovat sen haasteet ja mahdollisuudet, ja mikä yleensä on vaatimusmäärittelyn merkitys tässä kyseisessä kehitysprojektissa, kun tietojärjestelmä on ostettu palveluna.

5.1 Projektin kuvaus

Projekti oli alkanut vuonna 2017 kun yritys alkoi kartoittaa, minkälaisia mahdollisuuksia on uusia tai uudistaa varallisuudenhoidon alueen järjestelmiä. Kerättiin liiketoimintavaatimuksia ja mietittiin, rakennetaanko itse vai löytyykö markkinoilta valmis tuote. Silloinen projektiryhmä tutustui markkinoilla oleviin valmisohjelmistoihin, SaaS- ja PBaas-ratkaisuihin ja näiden toimien tuloksena päätettiin hankkia valmis ratkaisu, kuten jo tämän työn ensimmäisessä luvussa on kerrottu. Yrityksen näkökulmasta valmis ratkaisu tarkoitti seuraavia asioita:

- Ostetaan valmis (80%) commodity-tuote, jonka konfiguroidaan (20%) omiin tarpeisiin:
    - ei tarvitse määritellä kaikkea alusta alkaen itse koska tuote on jo olemassa
    - toimittaja tuntee hyvin valmiin tuotteensa toiminnan ja siinä olevat prosessit
    - toimittaja tuntee varallisuudenhoidon liiketoimintaa ja sen optimiprosesseja
    - toimittajalla on koeteltu käyttöönoton prosessi, johon tukeudutaan.
- Mukaudutaan ensisijaisesti tuotteen toiminnallisuuksiin ja siinä oleviin prosesseihin, jotka ovat jo käytössä useilla asiakkailla;
    - jokainen muutos tuotteen nykyprosesseihin on mietittävä tarkkaan
    - muutoksen ja asetuksen ero ymmärrettävä
    - asetus ei riko tuotetta, mutta muutos voi.

Toimittaja eli palveluntarjoaja puolestaan pyrkii vakioimaan oman tuotteensa ja minimoimaan käyttöönottoon kuluvan ajan,

- jotta sen tarjoaminen palveluna (SaaS, BPaaS) pysyy kustannustehokkaana
- koska käyttöönotto on kiinteähintainen

Käytännössä toimittajan tarjoama tuote käsittää kokonaisen tietojärjestelmän, joka koostuu useasta eri sovelluksesta. Toimittaja ottaa merkittävän vastuun (myös sopimuksellisesti) liiketoimintaa toteuttavan alustan pyörittämisestä. Yhteistyö on todellista kumppanuutta, jossa riskit jaetaan ja onnistumisesta palkitaan. Varsinainen kehityshanke alkoi vuoden 2019 alussa ja itse tulin projektiin keväällä 2019, kun alustavat liiketoimintavaatimukset ensimmäisen julkaistavan tuotteen osalta oli jo kerätty ja osittain analysoitu. Oli myös päätetty, että ensimmäinen julkaisu on niin sanottu MVP eli pienin toimiva tuote.

5.1.1 Projektiryhmän rakenne

Yrityksessä oltiin siirtymässä SAFe-mallisesta kehittämisestä heimomalliin ja projektiryhmä muodostikin oman heimon, joka jakautui aluksi kolmeen joukkueeseen. Työskentelytapoja pystyttiin paljolti sopimaan heimon sisällä, koska heimojen on tarkoitus olla autonomisia ja itseohjautuvia. Tietyt perusperiaatteet olivat tietenkin olemassa. Heimomallia käyttää muun muassa Spotify mutta yleensä heimomallia ei kopioida suoraan vaan jokainen yritys muokkaa siitä omannäköisensä mallin. Voisi sanoa, että tässä tapauksessa ketterä malli on eräänlainen hybridi, jossa mukaan on otettu hyviksi havaittuja käytäntöjä muista ketteristä malleista ja vesiputousmallista.

Heimossamme oli siis kolme joukkuetta tai itse puhumme niistä tiimeinä; yksi tiimi vastaa asiakaskäyttöliittymästä, toisen tiimin vastuulla on itse alustaan liittyvät toiminnot ja liittymät muihin järjestelmiin ja kolmas tiimi suunnittelee tulevaa migraatiota. Itse kuulun alustatiimiin. Myöhemmin tiimejä on perustettu lisää, muun muassa Systeemi-tiimi, joka vastaa käyttöönotoista ja yleensäkin siitä, että julkaisun jälkeen homma toimii lukuisten eri toimijoiden kesken. Varsinaisissa kehitystiimeissä on tuoteomistaja (product owner eli PO), määrittelijöitä eli solution analysteja (lyh. SA), kehittäjiä sekä testaajia. Heimossa on lisäksi tech lead, joka vastaa arkkitehtuurista ja teknisestä suunnittelusta yli tiimirajojen. Heimolla oli aluksi oma agile coach, mutta siitä luovuttiin ja heimoon otettiin kaksi agile masteria, joilla kummallakin on kaksi tiimiä. Heimoon kuuluu myös business lead, joka vastaa liiketoiminnallisista näkökulmista, ja heimoa johtaa heimopäällikkö eli tribe lead. Heimoon kuuluu paljon myös muita rooleja, mutta ne eivät ole relevantteja tämän tutkimuksen kannalta. Toimittajalla on lisäksi omat määrittelijänsä, kehittäjänsä ja testaajansa. Toimittajan puolella on projektipäällikkö, testauspäällikkö ja eräänlainen scrum master, joka vetää toimittajan omia tiimejä. Varsinaisesti yrityksen ja toimittajan tiimejä ei ole sekoitettu keskenään, vaikka tietyt toimittajan määrittelijät tekivät yhteistyötä tiettyjen tiimien SA:iden kanssa.

Tiimit tekevät joustavasti työtä yhdessä eikä niiden väliset rajat ole kiveen hakattuja. Alustatiimi esimerkiksi saattaa tehdä määrittelyitä, jotka toteuttaa käyttöliittymätiimi. Käytännössä yrityksen ja toimittajan työnjako menee niin, että yritys tekee vaatimusmäärittelyä, joista osan toteuttaa yrityksen omat kehittäjät ja osan toimittaja. Koska kyseessä on valmiit ohjelmistot, niin vaatimusmäärittely kohdistuu lähinnä liittymiin tai sellaiseen toiminnallisuuteen, jota tuotteessa ei ole, mutta joka tarvitaan. Esimerkki liittymistä on tuotteen integroiminen yrityksen omiin järjestelmiin, kuten asiakastietoihin, tilitietoihin ja kirjanpitoon. Omat haasteensa tuo se, että toimittaja on uusiseelantilainen ja sen päätoimipaikka Euroopassa on Iso-Britanniassa. Brexit toi tähän kuvioon vielä omat lisämausteensa. Myös

se, että yritys ei halunnut uutta palvelua asennettavan mihin tahansa pilveen vaan dedikoituun Suomessa sijaitsevaan konesaliin, aiheuttaa projektin kuluessa omat haasteensa.

Kun projekti alkoi, toimittajalta Suomessa oli vaihtelevasti noin 10-15 henkilöä, lähinnä ohjelmoija ja määrittelijöitä sekä projektipäällikkö ja asiakaspäällikkö. Satunnaisesti paikalla oli myös testauspäällikkö ja scrum master. Osa toimittajan työntekijöistä muutti Suomeen, mutta osa tekee töitä kotimaastaan käsin ja käy Suomessa vain tarvittaessa. Työntekijät on hyvin kansainvälistä: brittejä, skotteja, uusiseelantilaisia, yhdysvaltalainen, slovenialaisia, pakistanilainen, kiinalaisia ja nykyään myös suomalaisia. Toimittajan puolelta kaikki projektiin osallistujat eivät näy yritykselle mitenkään, joten lopullista lukumäärää on vaikea arvioida, mutta tällä hetkellä yritykselle näkyvää henkilöstöä on kolmisenkymmentä. Yrityksen omissa tiimeissä on vastaavasti projektin alussa ollut parikymmentä henkilöä, joista suurin osa yrityksen omia mutta mukana on myös vuokratyövoimaa. Tiimien lisääntyessä projektin ja myös heimon henkilömäärä on kasvanut ja tällä hetkellä yrityksestä on heimossa mukana noin viitisenkymmentä henkilöä ja muista heimoista projektiin osallistuu parikymmentä henkilöä, osa kokoaikaisesti ja osa omien töiden ohella.

5.1.2 Projektiryhmän työskentelytavat

Kuvaan tässä lyhyesti heimon työskentelytapojen periaatteet, käytännöt saattavat poiketa periaatteista, mutta palaan niihin tutkimustuloksissa.

Projektin alkaessa heimolla oli oma tila avokonttorissa. Tämä on myös ketterien toimintaperiaatteiden mukaista, koska Agile Manifestin mukaan liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan ja tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu. Käytännössä tämä toimii erinomaisesti, tieto kulkee eri tiimien välillä ja kun kuulee puhuttavan jostain itselle tärkeästä asiasta, keskusteluun voi hypätä jouhevasti mukaan. Näin toimittiin kevääseen 2020 saakka, jolloin maailmalla alkio levitä virus, jonka sittemmin opimme tuntemaan nimellä covid-19, tuttavallisemmin korona. Maaliskuun alussa koko yritys siirtyi etätöihin, ja suurin osa ulkomaalaisista projektin jäsenistä palasi kotimaihinsa. Tämä on jossain määrin voinut vaikuttaa tiimien toimintaan ja tiimien väliseen tiedonkulkuun. Tätä työtä kirjoitettaessa olemme olleet etätöissä vuoden ja kaksi kuukautta, ja paluu normaaliin on odotettavissa kesälomien jälkeen – tosin voidaanko puhua paluusta normaaliin, koska näyttää siltä, että ainakin osittainen etätyön tekeminen on uusi normaali. Koska kasvokkain käydyt keskustelut ovat minimissä, on Teamsista tullut tärkein työväline. Jokapäiväisten kokousten lisäksi chattailu, ad hoc-palaverit ja erilaiset

työpajat ovat jokapäiväistä toimintaa. Usein koko päivää menee erilaisissa palavereissa, kun ennen koronaa asiat saatettiin keskustella avokonttorissa ilman erillisiä kokouksia.

Ketterien menetelmien mukaisesti koko yrityksessä sprintin pituudeksi on määritelty kaksi viikkoa. Jokainen tiimi pyrkii pitämään säännöllisesti ketterään toimintatapaan kuuluvia seremonioita, joita ovat päivittäiset kokoukset eli dailyt, sekä joka toinen viikko olevat sprinttisuunnittelu ja retrospektiivi. Lisäksi joka toinen viikko on heimon yhteinen demo, jossa kaikki tiimit esittelevät aikaansaannoksensa. Demo voi olla toimiva ohjelma, osa ohjelmaa, testi, käyttöliittymävideo tai esitys esimerkiksi liittymän toiminnasta, jota on vaikea demota ilman näkyvää toiminnallisuutta. Lisäksi on paljon myös muita kokouksia, kuten projektin ohjausryhmän kokoukset ja tuoteomistajien synkronointipalaverit.

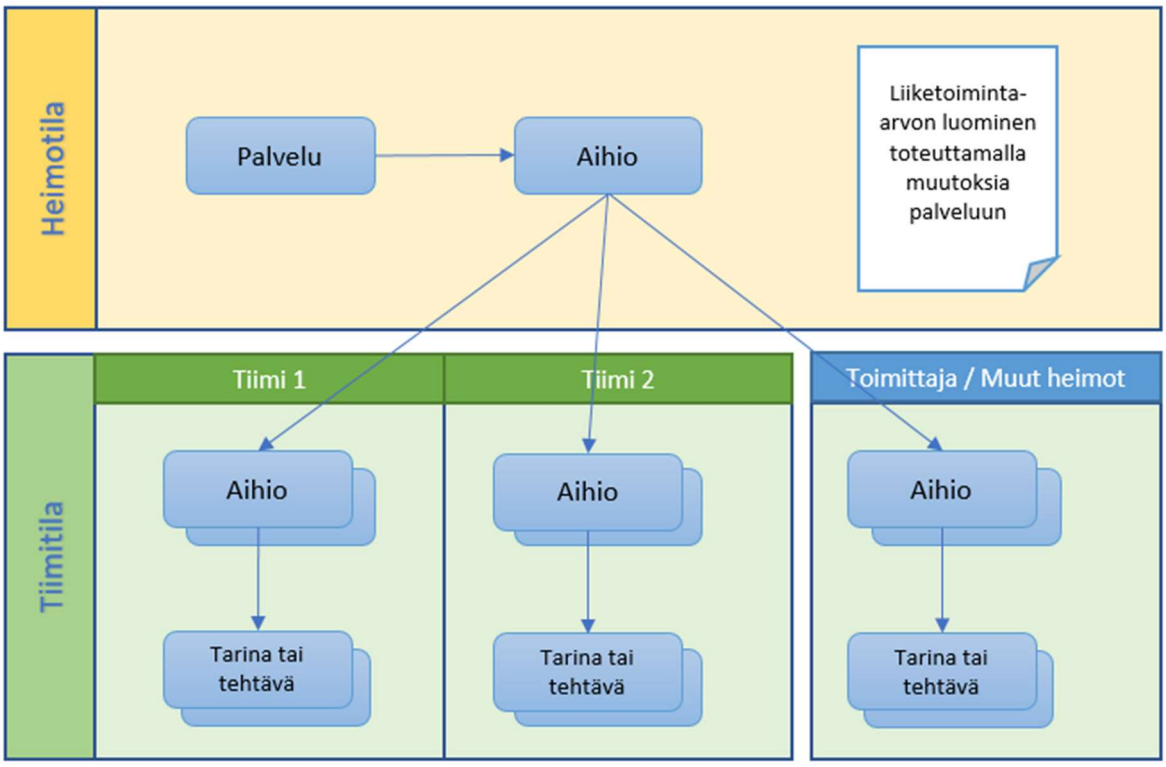
Heimolla on käytössään useita eri sovelluksia, mutta selvästi hyödyllisimmät työn tekemisen kannalta ovat Microsoftin viestintä- ja yhteistyöalusta Teams sekä Atlassianin organisaatiowikiohjelmisto Confluence ja tehtävienhallintaohjelmisto Jira. Jira on monipuolinen työnhallintatyökalu, joka mahdollistaa työn priorisoinnin, ajoituksen ja seurannan. Confluence on dokumentointityökalu ja sitä voidaan käyttää työn yksityiskohtaisempaan kuvaamiseen Jiran käytön tukemiseksi ja myös muun versionhallintaa vaativan dokumentaation tallentamiseen. Kullakin heimolla on oma työtila Confluencessa ja tiimeillä on yleensä oma sivustonsa heimon työtilassa. Jiraa puolestaan käytetään kehitysjonon ja työtehtävien hallintaan sekä heimotasolla tavoitteiden hallintaan. Jokaisella heimolla on yksi oma Jira-projekti tavoitteidensa hallintaan ja lisäksi tiimeillä on omat Jira-projektinsa kehitysjonojensa hallintaan. Käytännössä Jiraa käytetään tehtävienhallintaan ja varsinaiset vaatimukset ja määrittelyt ovat Confluencessa, mutta näissä on paljon heimokohtaisia ja jopa tiimikohtaisia eroja. Periaate kuitenkin on, että määrittelyitä ei kirjoiteta Jira-tehtäviin, koska ne ovat projektinaikaisia tehtäviä eikä niitä tallenneta myöhempää käyttöä varten.

5.1.3 Vaatimusmäärittelyprosessi

Projektin alussa on käyty toimittajan kanssa läpi uuden tietojärjestelmän ja sen yrityksen käyttöön tulevien sovellusten ominaisuudet ja kyvykkyydet, eli mitä tietojärjestelmällä voidaan tehdä ja mihin sitä voidaan käyttää. Tämän lisäksi on etsitty toiminallisuudesta niitä aukkoja, jotka pitää täyttää, jotta toimittajan sovellukset toimisivat yhteen yrityksen omien järjestelmien kanssa ja mitä pitää kehittää, jotta liiketoiminnan vaatimukset tulisi täytettyä. Tämän perusteella aloitettiin liiketoiminnan vaatimuksien kartoittaminen ja tehtiin päätös ensimmäisestä lanseerattavasta tuotteesta. Ensimmäiseksi julkaisuksi valittiin loppuasiakkaiden käyttöön tuleva mobiilisovellus, jonka avulla asiakas voi aloittaa tavoitteellisen

säästämisen. Tämän jälkeen alkoi varsinainen vaatimusten kartoittaminen, analysointi sekä suunnittelu.

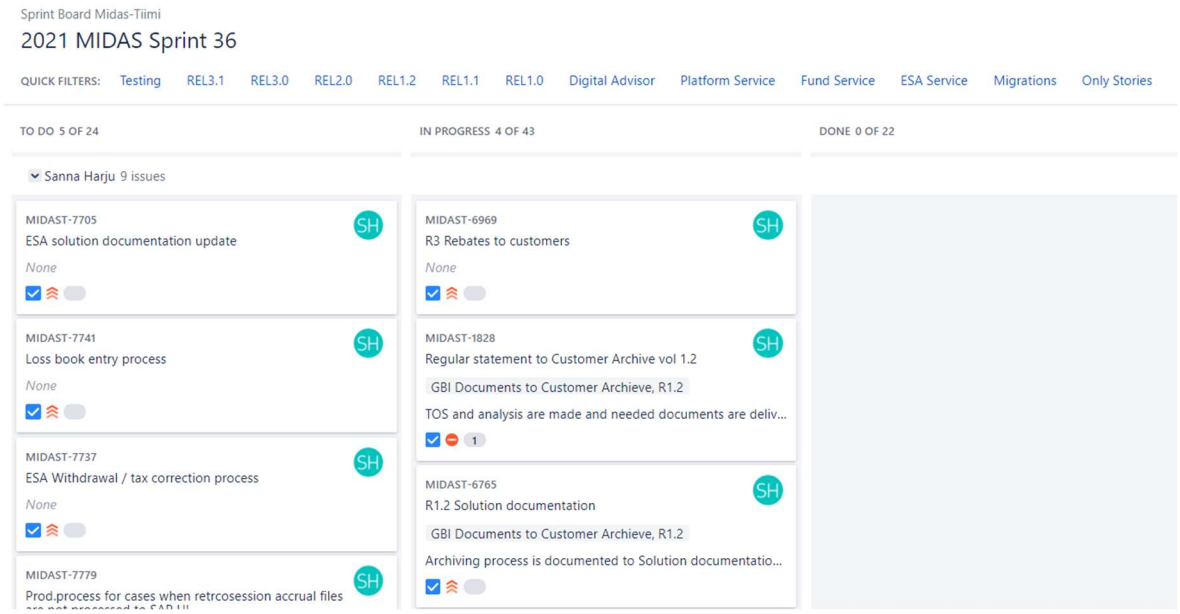
Käytännössä vaatimukset koostuvat aihioista (epic), tarinoista (story) ja tehtävistä (task). Liiketoiminta-aihio (business-epic) kuvaa merkittävää tavoitetta tai kohdetta, jolla on suoraa vaikutusta liiketoimintaan. Mahdollistaja-aihio (enabler epic) kuvaa tavoitetta tai kohdetta, jolla on epäsuoria liiketoiminnan vaikutuksia, kuten yhteiset palvelut, tutkimus, sääntelytehtävät tai arkkitehtuurin kehittäminen. Käytännössä aihio toteutetaan aina yhden heimon toimesta. Tarinat puolestaan ovat konkreettisia lisäarvoa tuottavia vaatimuksia, joilla on suoraa liiketoimintavaikutusta ja jotka tiimi toimittaa kahden viikon pituisen sprintin aikana. Aihio tai tarina voidaan vielä pilkkoa pienemmiksi tehtäviksi. Sprinttisuunnittelussa nostetaan tiimin kehitysjonosta sprintin työlistalle ne tarinat ja tehtävät, jotka tiimin aikoo toteuttaa seuraavan sprintin aikana. Aihioita puolestaan seurataan kanban-taululla, mutta niillä ei ole määriteltyä aikaikkunaa, jossa ne pitäisi toteuttaa.



Kuva 22. Aihiot, tarinat ja tehtävät heimon ja tiimin Jirassa.

Tiimissä vastuut jakautuvat siten, että käytännössä tuoteomistaja kirjoittaa aihiot Confluenceen ja Jiraan tiimin kanbanille sekä linkittää nämä keskenään, jotta myös Confluencessa näkyy aihion tila. Aihion tila muuttuu kanbanilla sitä mukaa, kun työtä edistetään. Kanban eroaa työlistasta siten, että kun työlista on sidottu sprinttiin niin kanbanilla olevilla aihiolle ei välttämättä ole määräaikaa. SA:t puolestaan pilkkovat aihiota pienemmäksi eli

tarinoiksi tai tehtäviksi, jos on tarpeen. Agile master pitää yllä sprintin työlistaa ja valvoo, että kaikki tarinat ja tehtävät ovat asianmukaisesti ajan tasalla ja priorisoitu. Agile master siis organisoi ja seuraa tiimin töitä. SA:t tekevät prosessikuvauksia, liittymäkuvauksia ja muuta määrittelyä tarpeen mukaan. Periaatteena on, että turhaa dokumentaatiota pyritään välttämään. Testaajat kirjoittavat testitapaukset määrittelyiden pohjalta. Käytännössä sprintin työlistalla kaikki tarinat ja tehtävät siirtyvät tilasta toiseen ja henkilöltä toiselle sitä mukaa, kun tarina tai tehtävä valmistuu ja siirtyy testattavaksi.



Kuva 23. Esimerkki Jiran työlistasta.

5.2 Tutkimuksen tulokset

Tutkimuksen haastatteluosuus tehtiin avoimena ja anonyyminä lomakekyselynä, osittain ajanpuutteen ja osittain tutkimuksen aikana jyllänneen koronaepidemian takia. Lisäksi tutkijana koin, että kyselyyn osallistujat vastaavat kysymyksiin suoremmin, kun he saavat vasta anonyymisti. Lomake lähetettiin kahdeksalle henkilölle, joista kuusi vastasi. Haastateltujen joukossa oli sekä SA:ita, tuoteomistajia, kehittäjiä ja testaajia. Koska kysely oli anonyymi, minulla ei ole tietoa siitä, ketkä kaikki vastasivat. Lomakekyselyn kysymykset on esitetty liitteessä 1. Vaikka kysymykset olivat avoimia, nousi kaikista vastauksista esiin muutama yhtenäinen teema. Näitä olivat:

- yhteistyö ja kanssakäyminen
- dokumentaatio ja muutoshallinta
- vaatimusmäärittelyn rooli
- kehittämismenetelmien käyttö

Teemat ovat myös aika suoraan verrannollisia Agile Manifestin periaatteisiin, joihin tutustuimme aikaisemmin kappaleessa 2.2.4. Osittain vastaukset menevät päällekkäin ja koh-

distuvat useaan teemaan; tämä saattaa johtua lomakehaastattelun muodosta, kun haastattelija ei ole ohjaamassa keskustelua vaan vastaukset ikään kuin rönsyilevät. Alla olevat lainaukset on kopioitu suoraan vastauksista.

5.2.1 Yhteistyö ja kanssakäyminen

Yhteistyö on yksi ketterän perusarvoista ja se korostui monessa vastauksessa monelta eri kantilta. Vastauksissa nousee esiin erityisesti SA:n, kehittäjän ja testaajan välinen kanssakäyminen ja varsinkin sen puute. Vastaajat toivoivat enemmän SA:iden ja kehittäjien yhteistyötä ja yhdessä tekemistä ja myös kaikkien kolmen roolin eli SA:n, kehittäjän ja testaajan yhdessä tekemistä ja kanssakäymistä korostettiin. Useassa vastauksessa tähdennettiin yhteisen työajan tärkeyttä ja myös testaajien mukanaolo määrittelyissä oli toivottavaa. Osa vastaajista toivoi myös yhteisiä katselmointitilaisuuksia sekä palvelumuotoilun osallistamista. Toisaalta yhteistyön koettiin myös parantuneen siitä, mitä se oli aikaisemmin.

> ”Solution analystin ja developerien parempaa yhteystyötä; yhdessä tekemistä - suunnitellaan tarvittavat user story yhdessä ja taskit niiden tekemiseen. Puuttuu ns. yhteinen planning. Samassa yhteydessä voidaan sopia tarvittava työtä tukeva dokumentaatio ja loppudokumentaatio.”

> ”Olisi mahtavaa, jos määrittelyssä olisi mukana edes jossain sopivassa määrin myös testaajat sekä aina tarpeen tullen kehittäjä, ja toimittajan puolelta relevantti edustus, kun tarpeen. Riittävä yhteinen työaika ja ymmärryksen saaminen kaikille relevanteille osapuolille vaatimuksista olisi hyvä olla olemassa.”

> ”Erityisesti asiakaskokemukseen vaikuttavissa osissa olisi hienoa, jos kokonaisuutta voisi esitellä palvelu- tai käyttäjäkokemussuunnittelijalle (UX designer). Myös loogisten kokonaisuuksien määrittelyn yhteen katsominen olisi myös varsin hyvä tehdä ennen kuin koodausta aloitetaan, jotta saataisiin ymmärrys, että miten yksi kokonaisuus toimii yhteen.”

> ”Yhteistyö on parantunut ainakin heimon sisällä, erityisesti tiimeissä. Tekemisessä tuntuu olevan yhdessä tekemisen meininkiä ja halua parantaa ja kehittää työtapoja ja tekemistä.”

Vastauksissa kävi ilmi myös, se että esimerkiksi määrittelyiden ja testitapauksien kirjoittamisen välissä on liian pitkä aika, jolloin ikään kuin myös määrittelijä joutuu palauttamaan mieliinsä, mitä jollakin asialla on tarkoittanut käydessään määrittelyitä läpi testaajan kanssa. Tämä saattaa johtaa siihen, että paljon tietoa jää välittymättä, varsinkin jos tes-

taaja kirjoittaa testitapaukset itsenäisesti eikä käy määrittelyitä läpi SA:n kanssa. Toteuttajien katsottiin olevan liian kaukana määrittelystä eli määrittelyn ja toteutuksen välissä kuluu liian kauan aikaa ja usein määrittelyä joudutaan muuttamaan toteutusvaiheessa, koska esimerkiksi vaatimus muuttuu tai reunaehdot eivät ole olleet määrittelijän tiedossa. Myös testauksen ja toteutuksen väli on liian suuri, jolloin toteuttajalla ole automaattitestauksen hyötyjä käytettävissä.

> ”Vaatimusmäärittelyn ja testitapausten rakentamisen välillä on liian pitkä aika, testaajan soisi olevan mukana määrittelykeskusteluissa. Silloin testauksessa pääsisi helpommin kiinni erilaisiin poikkeustapauksiin tai muuten erityisesti huomioitaviin asioihin; nyt ajan kuluessa ja kun testitapaukset tehdään dokumentointiin perustuen, paljon tietoa jää välittymättä.”

Yhteistyö toimittajan kanssa koettiin myös haasteellisena ja sen toivottiin parantuvan tai ainakin lisääntyvän. Mutta toisaalta joku toinen koki yhteistyön toimittajan kanssa olevan toimivaa ja hyvää. Toimittajan ja tiimien välisessä yhteistyössä haasteellisena pidettiin läpinäkyvyyttä; aika usein on tilanne, että tiimissä ei välttämättä ole tietoa siitä, mitä toimittaja tekee. Tieto saattaa olla kuitenkin jollain ylemmällä taholla, kuten heimopäälliköllä tai tech leadilla. Koronan aikaan yhteistyötä on heikentänyt myös se, että olemme siirtyneet etätöihin eikä enää ole mahdollisuutta spontaaniin keskusteluun.

> ”Toimittajan kanssa voisimme käydä heidän johdolla heidän tunnistamat tekemiset.”

> ”Yhteistyö toimittajan kanssa on usein aika kankeaa ja hidasta eikä puhuta aina samoista asioista samoilla termeillä. Myös näkyvyys muihin tiimeihin ja heidän tekemisiin, saati muiden heimojen meitä koskeviin tekemisiin on usein melko heikko, ja se tekee haastavaksi joskus nähdä ja tietää mitä toisaalla tapahtuu niiden asioiden osalta, jotka vaikuttavat heimon tekemisiin.”

> ”Huonosti on mennyt vaatimusmäärittelyt toimittajan kanssa, jossa ratkaisut eivät ole läpinäkymiä. Haastetta lisää se, että toimittaja olettaa itse asioita eikä tarkista niitä OP:lta. Täyttä ymmärrystä vaatimuksista ei pystytä todentamaan, vasta toteutuksen jälkeisessä testauksessa.”

Ajankäyttö nousi myös keskeiseksi teemaksi; toivottiin lisää aikaa keskustella vaatimuksista yhdessä toimittajan kanssa. Aikataulut koettiin liian tiukoiksi. Joku vastaajista nosti esiin analysointivaiheen ja sen toimimattomuuden, eli tässäkin toivottiin enemmän yhteistä aikaa ja keskinäistä kanssakäymistä toimittajan kanssa. Projektin roolitusta pidettiin epäselvänä ja myös henkilöiden muuttuvat vastuut yrityksen ja toimittajan välillä koettiin hämmentävinä.

Usean vastaajan mielestä käyttäjätarinoiden taso vaihteli liikaa, toiset ovat liian yksityiskohtaisia ja toiset taas liian ympäripyöreitä tai laajoja. Myös tapa tarkastella käyttäjätarinoita vaihteli eli toisilta meni läpi karkeampi taso, kun toinen vaati enemmän yksityiskohtia. Sisäisiä katselmointeja toivottiin lisää. Ongelmalliseksi koettiin myös yhteisen ymmärryksen puute siitä, milloin käyttäjätarina on valmis hyväksyttäväksi ja mitä voidaan muuttaa myöhemmin.

Vastauksissa toivottiin lisää avoimempaa keskustelua yrityksen ja toimittajan välillä sekä lisää yhteisiä kartoitus- ja määrittelysessioita yrityksen ja toimittajan SA:iden välillä samoin kuin yhteisiä työskentelytapoja. Tuoteomistajan eli PO:n roolia pidettiin liian määräävänä ja toivottiin lisää tiimihenkeä ja yhdessä tekemisen meininkiä. Ongelmalliseksi koettiin rajojen puute, roolit ja vastuut eivät ole selvillä, jolloin helposti tulee tehtyä sellaista mitä ei koeta kuuluvaksi omaan rooliin tai sitten joku tehtävä saattaa jäädä hoitamatta, koska ei tiedetä, kenelle sen tekeminen kuuluu.

> ”Toivoisin enemmän vuoropuhelua tiimissä eikä PO:n itsevaltaisuutta. Toivoisin myös enemmän tiimihenkeä ja yhdessä tekemisen meininkiä sekä selkeitä rajoja meidän tekemiselle tai tuelle muiden tiimien tekemiseen.”

5.2.2 Dokumentaatio ja muutoshallinta

Dokumentaation tarpeellisuutta korostettiin kaikissa vastauksissa. Osa vastaajista on sitä mieltä, että kuvalliset dokumentit ovat parhaita; näitä ovat esimerkiksi prosessikuvaukset. Samoin kokonaisuuden kuvaus, liiketoimintasäännöt ja sanasto katsottiin tärkeiksi dokumenteiksi. Varsinkin kokonaisuuden kuvaus pitäisi ottaa osaksi vaatimusmäärittelyä, sillä se auttaa tunnistamaan ja kiinnittämään kehityksen kohteet ja rajaamaan pois alueet, joita ei tarvita. Samoin kokonaiskuvan kautta pystytään johtamaan reunaehdot kuhunkin pienempään osa-alueeseen.

> ”Kaikenlaiset kuvalliset esitysmuodot olisivat parasta dokumentaatiota. Leipätekstin lukeminen ja erityisesti kaiken tärkeän informaation sisäistäminen siitä on paljon haastavampaa. Tärkeät muistettavat asiat olisi hyvä tuoda esimerkiksi "täppäämällä" kuvissa, silloin kun tämä on luontevaa. Samaten listamuotoinen luettelointi (numeroitu tai bullet) edesauttaa sitä että kaikki erilliset asiat noteerataan.”

Erityisen tärkeänä pidettiin sitä, että ratkaisut olisi myös perusteltu, koska tämä lisäisi ymmärrystä siitä, miksi juuri kyseiseen ratkaisuun on päädytty. Käyttötapauksia toivottiin, koska niistä pääsee käsiksi konkretiaan ja käyttötarkoitukseen. Useassa vastauksessa

tuotiin esiin myös dokumenttien välisen linkityksen puute ja se että dokumentteja oli usein vaikea löytää, koska ne ovat hajallaan eikä niiden välillä ole linkitystä. Määrittelydokumentaatio saattaa myös koostua niin pienistä osista, että kokonaisuus hukkuu. Kun samaan kokonaisuuteen tehdään työtä kahdessa tai useammassa tiimissä, on hankala nähdä kokonaisuutta ja varsinkaan sitä työtä, mitä on tehty oman tiimin ulkopuolella, oli yhden vastaajan mielipide. Myös Confluencen rakenne tuottaa päänvaivaa; jos dokumentti esimerkiksi siirretään toiseen paikkaan, on sen löytäminen vaikeaa. Tähän toivottaisiin jonkinlaista kehitysideaa, koska Confluencen vakionavigointi on melko kankea ja hidas käyttää.

> ”Selkokielistä, myös heimon ulkopuolisille ymmärrettävää dokumentaatiota, jossa olisi kuvattu analyyseista nousseet päätökset ja linjaukset, jos sellaisia tehty. Eli ymmärrys muillekin, että miksi johonkin ratkaisuun oli päädytty. Myös prosessikaaviot, liiketoimintaprosessien kuvaukset sanallisesti (ylätasolla) sekä teknisten ratkaisujen kuvaukset ovat olleet tärkeitä jälkikäteenkin. Toiminnallisuuksien riippuvuudet ja ehkä sellainen tehdyn (esim. palvelun) kokonaisuuden kokonaiskuva olisi myös äärimmäisen tärkeä. Use caset olisivat usein myös varsin hyödyllisiä ja niistä saisi kiinni käyttötarkoitukset ja konkretian”

> ”Solution analystin pitäisi tuottaa sekä liiketoiminnan kannalta tärkeitä dokumentteja, kuten prosesseja, sääntöjä ja sanastoa. Näiden avulla liiketoiminta pystyy todentamaan, että toiminnallisuus vastaa vaatimuksia. Prosesseilla kuvataan automatisoituja osia ja manuaalisia vaiheita, sekä saadaan ymmärrys myös työn määrästä tarvittavien prosessin input/outputtien kautta. Sanastolla tuodaan myös yhtenäinen käsitteistö asian ympärille. Säännöillä pystytään johtamaan tarkempaa ymmärrystä asiasta ja käyttötapaukset. Kehittäjille työtä tukevaa dokumentaatiota, esim. liittymätietojen kuvausta sekä palvelujen kuvausta.”

Muutoshallinta koettiin ongelmallisena. Toimittajan katsottiin tekevän muutosmäärittelyä mutta jättävän oman dokumentaationsa täydentämättä, jolloin kokonaisuus kärsii. Samoin tekninen suunnittelu tehdään vasta sen jälkeen, kun vaatimukset eli käyttäjätarinat on jo hyväksytty (sign off); tällöin suunnittelua tehdessä saatetaan huomata, että ratkaisua joudutaan muuttamaan, jolloin se poikkeaa määritellystä. Tätä ei toimittaja aina edes välttämättä kerro ja määrittelyn ja toteutuksen välinen ristiriita paljastuu vasta testauksessa. Toteuttajat saattavat myös muuttaa määriteltyä kokonaisuutta. Erään vastaajan mielestä toimittajan kanssa pitäisi tarkemmin miettiä, mitä määrittelyitä tuotetaan ennen niiden hyväksymistä. Toimittajan tekemä dokumentaatio koettiin monikäsitteisenä tai vaikeaselkoisena; vastaajan mielestä se näyttää ”lakitekstiltä” ja sitä luetaan kuin lakitekstiä. Toimittajan

tuottamaa määrittelyä kritisoitiin myös siitä, että jokaisesta muutoksesta tehdään uusi dokumentti ja alkuperäinen määrittely jää olemaan sellaisenaan. Vastaaja olisi pitänyt mielekkäämpänä, jos muutokset merkittäisiin selkeästi alkuperäiseen määrittelyyn.

5.2.3 Vaatimusmäärittelyn merkitys

Vaatimusmäärittely on melko tiukasti sidoksissa tuotettuun dokumentaatioon. Tässä tilanteessa tämä piirre tuntuu vielä korostuvan, koska keskinäinen kanssakäyminen on jäänyt kokonaan pois eikä asioista enää keskustella kasvotusten. Kaiken kaikkiaan vaatimusmäärittelyä pidetään erittäin tärkeänä ja kriittisenä toimintona myös ketterässä ohjelmistokehityksessä. Vaatimusmäärittelyn kautta löytyy tuoteomistajan tahtotila sekä aukot toimittajan tarjoaman alustan ja tahtotilan välillä. Vaatimusmäärittelyn kautta löytyy myös testattava ja toteutettava kokonaisuus sekä dokumentaatio ylläpitoa varten.

Vaatimusmäärittely on myös erittäin iso osa kokonaisuuden hallintaa ja se edesauttaa ymmärtämään aiempien ratkaisujen muutokset seuraaviin vaiheisiin ja jotta uutta kehitystä voidaan tehdä. Vaatimusmäärittelyn roolia ei saisi vähätellä vaan päinvastoin korostaa, varsinkin valmisohjelmistojen ja palvelukeskusten integraatioissa. Meille pitää olla selvää, mitä halutaan ja millaisia toiminnallisuuksia tarvitsemme. Vaatimukset tulee olla asiallisesti kirjattu tai toimittaja voi yrittää hyötyä tilanteista, joissa vaatimukset ovat puutteellisia tai epäselviä.

> ”Merkitystä ei saisi vähätellä, vaan paremminkin korostaa - varsinkin valmisohjelmistojen/palvelukeskusten integraatioissa. Asiakkaana meillä on oltava selvää, mitä haluamme saavuttaa ja millaisia toiminnallisuuksia tarvitsemme. Jollei näitä ole asiallisesti kirjattu, toimittaja vie meitä kuin pässiä narussa.”

Vaatimusmäärittelyn katsotaan olevan merkittävässä roolissa myös silloin, kun tulkitaan ja tarkennetaan tuoteomistajalta saatuja ylätason vaatimuksia yhdessä kehittäjien ja toimittajan resurssien kanssa. Vaatimusmäärittely sai ylipääteen paljon positiivista palautetta. Ratkaisut käydään läpi yhdessä ja ne on dokumentoitu hyvin. Määrittelijöillä on hyvä käsitys oman vastuualueen liiketoiminta- ja sovelluslogiikasta.

5.2.4 Kehittämismenetelmien käyttö

Yksi kysymyksistä koski ketterien menetelmien käyttöä. Tämän kappaleen vastaukset liittyvät myös osittain jo ensimmäisessä kappaleessa esille tulleeseen ketterän ensimmäiseen perusperiaatteeseen eli kanssakäymiseen. Erityisesti tässäkin korostuu yhteistyön ja yhtäaikaisen tekemisen puute.

Ketterät ohjelmistokehitysmenetelmät on otettu joidenkin vastaajien mielestä hyvin käyttöön ja toisten mielestä niitä ei käytetä lainkaan, vaan kehitysmalli on jostain iteroivan vesiputouksen ja ketterän välimaastosta. Toisaalta on myös nähty, että heimossa on pystytty jopa kehittämään ketteriä menetelmiä ja toimintatapoja edelleen heimon ja tiimin omaan tarpeeseen. Vastauksista kävi ilmi myös, että toimittajan resurssit eivät ole yrityksen omissa ketterissä tiimeissä vaan toimittajan omassa projektissa. Osa vastaajista oli sitä mieltä, että toteutus on suurimmassa määrin vesiputousmallin mukaista, mutta se on kääritty ketterään käärepaperiin. Joissain tiimeissä kehitys on tehty puhtaasti määrittely-toteutus-testaus -mallilla.

Erään vastaajan mielestä tiimit ovat siiloutuneet ja niitä johdetaan eri tavalla. Toimittajan vesiputousmainen tapa tehdä töitä kävi ilmi kahdesta vastauksesta. Agile Coachin puuttumista pidettiin ongelmallisena, koska kukaan ei tällöin vie ketterää toiminta mallia heimossa eteenpäin ja agile masterit taas ovat liian ylityöllistettyjä ehtiäkseen edistää ketteriä menetelmiä. Myös hankkeen johdon ei katsottu olevan kartalla ketterin toimintatapojen suhteen. Vastaajien joukossa oli myös heitä, joiden mielestä ketterät kehitysmenetelmän on otettu töiden seurantaan ja hallinnointiin.

> ”Ketteriä menetelmiä ei juurikaan käytetä. Ei edes käyttöliittymätoteutuksissa niin että määrittely, toteutus ja testaus tekisi yhteistyötä yhtäaikaa, jolloin automaattitestit olisivat jo toteutuksen käytettävissä ja määrittely olisi toteutuksen mukainen kun yhdessä suunniteltaisiin mikä toteutukselle on mahdollista.”

> ”Heimo on ottanut hyvin käyttöön ketterät kehitysmenetelmät töiden seurantaan ja hallinnointiin. Lisäksi heimossa on myös kehitetty niitä edelleen tiimin omaan tarpeeseen.”

Osa vastaajista oli sitä mieltä, että emme toimi vielä tarpeeksi ketterästi. Tästä seurauksena on se, että tiimillä on liikaa tehtäviä samanaikaisesti ja yritetään haukata liian isoa palaa kerralla. Työskentely on tehotonta ja voimavaroja kuluttavaa.

> ”Siirtyä enemmän ketterään malliin, jossa tehdään pienempiä palasia valmiiksi alusta loppuun ja sitten vasta siirrytään tekemään seuraavaa. Nyt on turhan monta päällekkäistä toiminnallisuutta yhtä aikaa pöydällä. Se tekee työstä tehotonta ja aivoja kuluttavaa”

Vastauksissa tuotiin esille Kanbanissa käytettävän WIP-limiitin käyttöönotto tai jokin muu tapa rajoittaa työmäärää. WIP (Work In Progress) -limiitti tarkoittaa sitä, että työlistalla voi

yhdessä työvaiheessa olla vain asetetun limiitin verran tehtäviä kerrallaan. Edelliset tehtävät täytyy suorittaa seuraavaan vaiheeseen asti ennen kuin uusita tehtäviä voidaan siirtää eteenpäin.

> ”Omassa tiimissä voisimme päästä tehokkaampaan tekemiseen, jos seuraisimme tarkemmin WIP limiittiä sekä myös töiden edistymistä sprintissä ja todeta mahdolliset blokkerit töiden viivästymiselle ja voisi tehdä tähän tarvittavat korjausliikkeet. Monesti työt siirtyvät "vain" sprintistä toiseen”

Ketterät seremoniat, kuten dailyt, retrot ja demot, nähtiin positiivisena asiana ja niiden katsottiin tukevan tekemistä omassa tiimissä. Ketteriä seremonioita vastaajien mielestä noudatetaan hyvin. Päivittäiset kokoukset myös parantavat tiedonkulkua ja lisäävät tiimihenkeä. Toisaalta seremonioita on myös moitittu siitä, että ne vievät aikaa itse tekemiseltä. Vastauksissa toivottiin enemmän jatkuvan parantamisen mallin käyttöönottoa.

Haasteena koettiin se, että sekä toimittajalla että heimolla on omat Confluence- ja Jira -työtilat ja jotkut tehtävät ja vaatimukset kuvataan kahteen kertaan. Lisäksi kaikki yrityksen työntekijät eivät pääse toimittajan työtiloihin, jonka takia dokumentteja esimerkiksi kopioidaan työtilasta toiseen. Tämä teettää lisää työtä ja voi aiheuttaa sekaannusta, jos tällaisia dokumentteja ei pidetä ajan tasalla. Linkityskään ei tässä tapauksessa auta, jos käyttövaltuudet toiseen sovellukseen puuttuvat.

Vastauksissa tuli ilmi myös se, että vaatimusten kartoittamisen ja analysoinnin menetelmät ovat epäselviä ja pitäisi löytää yhteinen ymmärrys heimon ja toimittajan välillä, miten vaatimuksia kerätään ja milloin vaatimus on valmis hyväksyttäväksi. Osittain toimittajan työskentelytavan takia muutoksien tekeminen hyväksyttyihin vaatimuksiin on vaikeaa ja hidasta.

6 Johtopäätökset ja pohdinta

Tutkimuksen tulokset ovat mielestäni aika yksiselitteiset. Projektissa käytetyt kehittämismenetelmät eivät ole selviä; paperilla ollaan ketteriä ja yritetään tehdä ohjelmistokehitystä ketterillä menetelmillä, mutta käytännössä kehittäminen on hyvin pitkälti vesiputousmallisesta kehittämistä. Projektissa ei toteudu kunnolla edes iteratiiviset tai inkrementaaliset toteutustavat, koska iteraation lopuksi ei ole esittää mitään valmista tuotosta eikä kehitystä tehdä vaiheittain. Julkaisuja on todella harvoin, vain noin kerran tai kaksi vuodessa, ja ne ovat isoja. Käytännössä ohjelmistoa tehdään niin, että ensin kartoitetaan ja analysoidaan vaatimuksia, ja sitten vaatimukset esitetään kehittäjällä, että tässä, toteuta ja lopulta muutamia viikkoja tai jopa kuukausia kestäneen toteutusvaiheen jälkeen testaajat saavat testattavakseen toimivan ohjelman tai osan siitä. Tässä vaiheessa määrittelijä on jo unohtanut, mitä on määritellyt ja testaajan on vaikea tulkita omia testitapauksiaan. E2E-testaus ovat hankalaa, koska yhteensovitettavia palasia on kerralla niin paljon, että on vaikea tietää, missä kohtaa ohjelma menee pieleen, jos sovellus ei toimikaan. Joko palaset tai kehittämiskohteet ovat liian suuria, jotta ne voidaan toteuttaa yhdessä sprintissä niin kuin olisi ketterien periaatteiden mukaista. Käytännössä yksi aihio on melko iso kokonaisuus ja se pitäisi osata pilkkoa yhden sprintin mittaisiksi pienemmiksi tehtäviksi ja tarinoiksi. Tässä meillä on vielä opittavaa, koska pilkkomisen jälkeenkin tarinoiden ja tehtävien valmistuminen saattaa viedä useiden sprinttien ajan.

Yhteistyö ja yhdessä tekeminen puuttuu, toisin sanoa vaatimuksia pitäisi kerätä ja kirjoittaa yhdessä kehittäjien ja testaajien kanssa, mutta tällä hetkellä vaatimusten analysointi on aika yksinäistä työtä ja se jää liian usein vain SA:n harteille. Kun vaatimus on analysoitu, niin vasta silloin otetaan testaaja ja kehittäjä mukaan ja heille esitellään tavallaan valmis vaatimus. Tässä vaiheessa varsinkin testaajilta voi tulla kullanarvoisia vinkkejä ja näkökulmia, samoin kehittäjältä ja silloin vaatimusta täydennetään ja korjataan. Aikaa menee taas hukkaan, kun SA kirjoittaa vaatimuksen uusiksi, jonka jälkeen taas katselmoidaan sitä yhdessä. Ajan säästö olisi luultavasti mahdollista, jos aihio otettaisiin heti käsittelyyn, pilkottaisiin se sopivan kokoisiin tehtäviin ja analysoitaisiin yhdessä. Prosessi menee muutenkin nyt niin, että ensin tuoteomistaja kirjoittaa aihiot, jotka sitten sprinttisuunnittelussa pilkotaan sopiviksi tehtäviksi lähinnä SA:n ja agile masterin toimesta otsikkotasolla. Analysointi jää kuitenkin aika pitkälle SA:n vastuulle, kun siihen sen sijaan pitäisi valjastaa koko tiimi tai ainakin se kehittäjä ja testaaja, joille tehtävä myöhemmin siirtyy toteutettavaksi ja testattavaksi. Tästä voin vetää sen johtopäätöksen, että kanssakäymistä ja yhdessä tekemistä tiimin kesken pitää ehdottomasti lisätä. Yhteistä aikaa tekemiselle pitää löytyä ja vesiputousmallisesta kehittämisestä pitää pikkuhiljaa päästä siirtymään kohti oikeasti ketterämpää suuntaa.

Ajasta tuntuu koko ajan olevan pula, määräajat kaatuvat päälle, työskentely on tehotonta, kun yritetään saada mahdollisimman paljon aikaa mahdollisimman nopeasti. Samanaikaisia tehtäviä on niin paljon, että mihinkään ei voida keskittyä kunnolla vaan tekeminen on ”rääpimistä”. Tehdään ensin vähän tuota ja sitten taas vähän tätä toista. Tehtäviä ei saada valmiiksi vaan ne vaan siirtyvät sprintistä toiseen, kunnes huomataan, että tämän pitää olla valmis nyt, mieluummin eilen. Tähänkin auttaisi aihioiden pilkkominen pienemmiksi ja realistisempi suhtautuminen toteutettaviin tehtäviin ja niiden viemään aikaan.

Vaatimusmäärittelyn merkitystä ei voi liikaa korostaa. Vaikka on ostettu periaatteessa valmis tuote tai oikeastaan palvelu, niin kuitenkin on paljon asioita, mitä pitää määritellä, analysoida, toteuttaa ja testata ennen kuin palvelu on valmis käyttöön otettavaksi. Tietenkin paljon riippuu sovelluksesta tai ohjelmistosta itsessään, tekstinkäsittelyohjelma on huomattavasti helpompi ottaa käyttöön ilman mittavia konfigurointeja kuin asiakkaille kohdennettu uusi palvelu, jossa tarvitaan paljon liityntäpintaa master dataan. Jos aluksi yritetään mennä mahdollisimman vähillä konfiguroinneilla ja liittymillä olemassa oleviin järjestelmiin, saattaa se kostautua myöhemmin entistä vaikeampana ja monimutkaisempana kehittämisenä, ja tehtyjä ratkaisuja voidaan jouduta jopa purkamaan ja rakentamaan uudestaan. Ketterät projektit vaativat periaatteessa samantyyppisiä vaatimuksia kuin perinteiset kehityshankkeet. Jonkun tarvitsee edelleen kartoittaa vaatimukset, analysoida ne, dokumentoida erilaiset vaatimukset sopivalla tarkkuudella ja vahvistaa, että vaatimukset täyttävät liiketoiminnan tavoitteet. Yksityiskohtaisia vaatimuksia ei kuitenkaan dokumentoida kerralla ketterän projektin alussa. Sen sijaan korkean tason vaatimukset pitää saada luotua kehitysjonoon suunnittelun alkuvaiheessa.

Yksi merkittävä haaste on se, että mitä valmiilla sovelluksella voidaan tehdä ja mitä pitää rakentaa lisää, jotta se olisi oikeasti käytettävä ja toisi lisäarvoa asiakkaille. Tässäkin vaatimusmäärittelijän rooli on elintärkeä, juuri hän selvittää aukkoja valmisohjelman ja muiden järjestelmien välillä, piirtää prosessikuvia toimintatavoista ja -malleista ja sovellusten toiminnallisuudesta ja selvittää, mitä rajapintoja tai liittymiä tarvitaan. On myös haastavaa kehittää uusi tuote tai palvelu, jota asiakas haluaa käyttää, ilman että tehdään mittavaa asiakas- tai käyttäjätutkimusta rakentamalla prototyyppejä. Vuoden kestäneen kehitysprojektin jälkeen voidaan huomata, että tämä ei ollutkaan sitä, mitä asiakkaat halusivat. Ketteriä menetelmiä hyödyntäen olisi mahdollista kehittää ihan oikea MVP, jolla voitaisiin koeponnistaa, onko tämä nyt sitä, mitä asiakas haluaa. Tiukasti säännellyssä maailmassa tämä on kuitenkin haavekuva, joka on mahdoton toteuttaa. Kaikki regulaation, lain ja ase-

tusten asettamat vaatimukset ja reunaehdot tulee täyttää ilman mitään poikkeuksia ja tällainen kehittäminen vaatii paljon aikaa ja resursseja. Tällöin väistämättä koko kehittämisen prosessi ei voi olla kovin ketterää.

Ketterän ohjelmakehityksen ikuinen dilemma on dokumentaatio; millaista dokumentaatiota tarvitaan ja miten sitä tehdään ja minne sitä tehdään. Yleensä tämä riippuu tosi paljon projektista ja yrityksen käytännöistä. Tutkimuksessa paljastui, että tiimien käytännöt saattavat poiketa aika tavalla toisistaan; toisissa tiimeissä saatetaan tuottaa paljon erilaista dokumentaatiota ja toisissa vähemmän. Myös aihioiden, tehtävien ja tarinoiden koot eivät olleet yhteismitallisia keskenään tiimien välillä. Tämä ei kuitenkaan ole mitenkään haitallista, jos tiimi toimii muuten hyvin ja vaatimusten perusteella toteutus ja testaus onnistuu. Tiimissä saattaa olla enemmän keskustelua kuin kirjallista dokumentaatiota. Ihmiset ajattelevat joskus, että ketterien projektiryhmien ei pitäisi kirjoittaa vaatimuksia, mutta se ei pidä paikkaansa. Sen sijaan ketterät menetelmät kannustavat luomaan vähimmäismäärän dokumentaatiota, joka tarvitaan kehittäjien ja testaajien ohjaamiseksi ja heidän työnsä tukemiseksi. Kaikki muu dokumentaatio paitsi se, jota kehitys- ja testiryhmät tarvitsevat (tai jota vaaditaan sääntöjen tai standardien täyttämiseksi), on hukkaan menevää työtä.

Muutospyyntöjen ja muuttuneiden vaatimusten kirjaaminen ja hallinnoiminen tuntuu olevan tiimeissä joskus epäselvää. Ketterät menetelmät kuitenkin toivottavat muutokset tervetulleiksi ja se on kaiken lisäksi yksi perusperiaatteista. Muutoshallintaa ei saisi tehdä liian raskaaksi ja muutoksista pitäisi pystyä kommunikoimaan avoimesti. Tärkeää on, että muutokset kuvataan uusina vaatimuksina ja priorisoidaan kehitysjonossa kuin mikä tahansa vaatimus. Joskus muutospyyntö voi aiheuttaa jopa sen, että jokin aikaisemmin esitetty vaatimus poistetaan kehitysjonosta tarpeettomana. Muutokset pitää kuitenkin aina kirjata ylös, ei niin, että kehittäjä muuttaa koodia jonkun pyynnöstä kertomatta siitä muille; tällöin testauksessa voi paljastua, että koodi ei vastaakaan vaatimuksia. Muutoshallinnan käytännöistä on hyvä sopia etukäteen toimittajan kansaa, ettei tule yllätyksiä. Toimittajan kehitysjono voi olla niin täynnä, että uudet tai muuttuneet vaatimukset menevät paljon myöhäisemmäksi kuin mitä asiakas on olettanut. Joskus muutospyyntö voi vaarantaa koko julkaisun. Näin varsinkin projekteissa, joissa toimittaja toimii enemmän vesiputousmaisesti. Organisaatiot kuitenkin tietävät, että projekteissa tapahtuu muutoksia. Jopa liiketoiminnan tavoitteet voivat muuttua. Kun muutospyyntöjä syntyy, suurin muutos, jonka SA:n pitää osata tehdä, on sanoa ”Okei, puhutaan vaan muutoksesta”, sen sijaan että sanoisi: ”Meillä on tällainen muutostenhallintaprosessi, jota pitää noudattaa” tai ”Ei, tämä toiminnallisuus on jätetty ulkopuolelle”. Tämä kannustaa tiimiä luomaan tai muuttamaan käyttäjätarinoita ja priorisoimaan niitä vasten kaikkia jo kehitysjonossa olevia vaatimuksia.

Kuten kaikissa hankkeissa, myös ketterien projektiryhmien on hallittava muutoksia harkitusti vähentääkseen niiden kielteistä vaikutusta, ja ne pitävät muutoksia todennäköisinä ja jopa kannustavat niihin.

Kaiken kaikkiaan tutkimuksen tulokset olivat pitkälti sen suuntaiset kuin oli odotettavissa havaintojeni perusteella. Vaatimusmäärittelyä tarvitaan edelleen ohjelmistokehittämisessä, vaikka kysymyksessä on ohjelmistotuote, johon ei toistaiseksi tehdä varsinaista räätälöintiä, mutta johon toteutetaan kuitenkin paljon liittymiä yrityksen omiin järjestelmiin sekä uusia käyttöliittymiä asiakastarpeisiin. Ja kun projekti etenee, myös liiketoiminnan tarpeet ja vaatimukset lisääntyvät, jolloin myös vaatimusmäärittelyn merkitys edelleen kasvaa. Dokumentointi on projektissa jo hyvällä mallilla, mutta jatkossa pitää kehittää muutoshallintaa ja lisätä yhteistyötä ja yhdessä tekemistä vaatimusmäärittelyvaiheessa. Pitää muistaa, että vaatimusmäärittelyprosessi toimii sellaisenaan myös ketterässä ohjelmistokehityksessä, prosessi vain tapahtuu lyhyemmissä sykleissä ketterän kehittämisen iteraatioiden aikataulussa.

Lähteet

Agile Alliance. 2001. Agile Manifesto. Luettavissa https://agilemanifesto.org/iso/fi/manifesto.html Luettu 31.3.2021

Ashmore, S. & Runyan, K. 2014. Introduction to Agile Methods. Addison-Wesley Professional.

Aurum, A. & Wohlin, C. 2005. Engineering and Managing Software Requirements. Springer. Berlin.

Boehm, B.W.1988. A spiral model of software development and enhancement. IEEE Computer, Vol. 21, No. 5. s. 61-72.

Chemuturi, M. 2011. Requirements Engineering and Management for Software Development Projects. Springer. New York.

Cooke, J.L. 2012. Everything you want to know about Agile. IT Governance Publishing.

Foster, Elvis C. 2014. Software Engineering: A Methodical Approach. Apress.

Dooley, J. 2011. Software Development and Professional Practice. Apress.

Hakala, I. & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Talentum. Helsinki.

Heath, F. 2020. Managing Software Requirements the Agile Way. Packt Publishig.

Kotonya, G. & Sommerville, I. 2004. Requirements Engineering processes and techniques. John Wiley & Sons. Chichester.

Laplante, P. A. 2011. Requirements Engineering for Software and Systems. CRC Press. Boca Raton.

Leffingwell, D. 2011. Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Addison-Wesley Professional. New Jersey.

Leffingwell, D. & Widrig, D. 2003. Managing Software Requirements: A Use Case Approach, Second Edition. Addison-Wesley Professional. New Jersey.

Metsämuuronen, J. 2006. Laadullisen tutkimuksen käsikirja. Gummerus Kirjapaino Oy. Jyväskylä.

Mohapatra, P.K.J. 2011. Software Engineering (A Lifecycle Approach). New Age International (P) Limited, Publishers. New Delhi.

Project Management Institute. 2017. Agile Practice Guide. Newtown Square. Pennsylvania.

Project Management Institute. 2016. Requirements management: a practice guide. Newtown Square. Pennsylvania.

Puusa, A., Juuti, P. & Aaltio, I. 2020. Laadullisen tutkimuksen näkökulmat ja menetelmät. Gaudeamus

Rajlich, V. 2016. Software Engineering. CRC Press. Boca Raton.

Robertson, J. & Robertson, S. 2013. Mastering the Requirements Process, Getting Requirements Right, Third Edition. Addison-Wesley. New Jersey.

Subramanian, C. 2015. Software Engineering. Pearson Education.

SWEBOK Guide V3. 2014. Guide to the Software Engineering Body of Knowledge. A Project of the IEEE Computer Society.

Taina J., Salmenkivi M. & Lemström K. 2011. Ohjelmistotuotannon kurssi. Luettavissa https://www.cs.helsinki.fi/u/klemstro/ohtu_k2011/luento11.pdf Luettu 12.4.2021.

van Vliet, H. 2008. Software Engineering: Principles and Practice. John Wiley & Sons. Chichester.

Wiegers, K. 2010. More about software requirements. Thorny issues and practical advice. Microsoft Press.

Wiegers, K. & Beatty, J. 2013. Software Requirements. Microsoft Press.

Wikipedia a. 2021. Ohjelmistotuotanto. Luettavissa: https://fi.wikipedia.org/wiki/Ohjelmistotuotanto Luettu 23.3.2021

Wikipedia b. 2021. Software crises. Luettavissa: https://en.wikipedia.org/wiki/Software_crisis. Luettu 31.3.2021

Wikipedia c. 2021. Lean. Luettavissa: https://fi.wikipedia.org/wiki/Lean Luettu 16.4.2021.

Wikipedia d. 2021. Scrum. Luettavissa: Scrum – Wikipedia Luettu 16.4.2021

Wikipedia e. 2021. MVP. Luettavissa: https://fi.wikipedia.org/wiki/Pienin_toimiva_tuote Luettu 9.5.2021

Wikipedia f. 2021. Tietojärjestelmä. Luettavissa https://fi.wikipedia.org/wiki/Tietoj%C3%A4rjestelm%C3%A4 Luettu 10.5.2021.

Young, R.R. 2004. Requirements Engineering Handbook. Artech House. Boston.

Liite 1

Lomaketutkimuksessa esitetyt kysymykset

Millainen rooli tai merkitys vaatimusmäärittelyllä mielestäsi on nykyisen mallisessa ohjelmistokehittämisessä?

Mikä on mennyt mielestäsi hyvin vaatimusmäärittely- / analysointiprosessissa omasta näkökulmastasi?

Entä mikä on mennyt mielestäsi huonosti vaatimusmäärittely-/ analysointiprosessissa omasta näkökulmastasi?

Millaista dokumentaatiota toivoisit solution analystien tuottavan? Tai millaista dokumentaatiota näkisit tarpeelliseksi tuottaa?

Miten haluaisit parantaa tai kehittää vaatimusmäärittelyä tiimeissä? Vaatimusmäärittelyllä tarkoitan tässä laaja-alaisesti sekä omille kehittäjille että toimittajalle annettuja taskeja ja storyjä sekä dokumentaatiota.

Miten ketterien kehitysmenetelmien käyttö mielestäsi toteutuu?

Miten haluaisit kehittää oman tiimisi työskentelytapoja?