Tampereen yliopisto

Joonas Koski

MÄÄRITTELYMENETELMÄT KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2020

TIIVISTELMÄ

Joonas Koski: Määrittelymenetelmät ketterässä ohjelmistokehityksessä
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2020

Nykypäivänä yleisenä trendinä on pyrkiä tehostamaan kaikkea toimintaa äärimmilleen ja tavoitteena on saada näkyviä tuloksia aikaan heti sekä reagoida muuttuviin tilanteisiin nopeasti ja joustavasti. Vastaavanlainen trendi on vallannut myös ohjelmistokehityksen ja projekteja pyritään ohjaamaan lähes poikkeuksetta ketterien menetelmien avulla. Sen sijaan, että ohjelmistoa kehitettäisiin pitkiä aikoja ja julkaistaisiin suurempi kokonaisuus kerralla, on siirrytty ajatusmaailmaan, jossa ohjelmistoa toimitetaan pienemmissä erissä laajentaen saatavilla olevaa toiminnallisuutta hiljalleen. Ketterissä menetelmissä asiakas pidetään hyvin lähellä prosessia ja jatkuvasti mukana ohjelmiston kehityksen eri vaiheissa, jolloin ohjelmiston laatu paranee sekä asiakkaan sille asettamien vaatimusten täyttyminen on todennäköisempää. Jatkuvan asiakkaan ja sidosryhmien kanssa kommunikoinnin myötä vaatimusmäärittelyn merkitys korostuu ja vaatimuksiin tehdään muutoksia, korjauksia ja lisäyksiä pitkin projektia.

Tämän diplomityön aihe keskittyy juuri vaatimusmäärittelyyn sekä siihen kuuluvien määrittelyiden tekemisessä käytettäviin menetelmiin. Painopiste työssä on matemaattisten, toiminnallisuuden todentamiseen ja tarkistamiseen käytettävien formaalien menetelmien tutkimisessa. Formaalien menetelmien pääasiallinen tarkoitus on auttaa havaitsemaan ja korjaamaan mahdolliset virheet mahdollisimman aikaisessa vaiheessa projektia ja näin pyrkiä säästämään projektin kustannuksia ja nopeuttamaan aikataulua sekä parantaa lopullisen ohjelmiston laatua. Diplomityö tehtiin osana työn toimeksiantajana toimineen Konecranesin ohjelmistoprojektia, jonka aikana vaatimusmäärittelyn käytänteistä ketterien menetelmien yhteydessä voitiin tehdä käytännön havaintoja. Valtaosa tutkimuksesta toteutettiin kuitenkin alan kirjallisuuteen ja viime aikaisiin aiheesta tehtyihin tutkimuksiin tutustumalla. Tavoitteena oli selvittää, minkälaisia käytäntöjä formaalien menetelmien käyttöön liittyy ketterässä ohjelmistoprojektissa ja onko formaalien menetelmien käytölle nykypäivänä edellytyksiä.

Vaikka formaaleja menetelmiä on kehitetty ja käytetty jo useiden vuosikymmenien ajan, ei niiden käyttö ole koskaan yleistynyt valtavirran työkaluksi ohjelmistoja kehitettäessä. Päällimmäisenä syynä tähän on varmasti formaalien menetelmien korkea oppimiskynnys. Matematiikkaa ja logiikkaa käyttävien menetelmien hallitseminen ja täyden hyödyn irti saaminen vaatii käytettävään menetelmään paneutumista sekä lahjakkuutta matematiikan ja loogisen päättelyn saralla. Ongelma korostuu entisestään, kun kyseistä osaamista pitäisi löytyä sekä määrittelyiden tekijältä että varsinaisesta toteutuksesta vastaavalta kehittäjältä.

Formaalien menetelmien ongelmiin on pyritty löytämään ratkaisuja niiden koko olemassaolon ajan. Yhtenä ratkaisuna on ehdotettu kevennettyä mallia, jossa koko kehitysprosessin formalisoimisen sijaan keskityttäisiin käyttämään formaaleja menetelmiä vain turvallisuudeltaan kriittisten osa-alueiden toteutukseen ja muu kehitys toteutettaisiin edelleen perinteisemmillä menetelmillä. Formaaleja menetelmiä voitaisiin käyttää myös vanhaa järjestelmää modernisoitaessa, jolloin vanhasta ohjelmistosta voitaisiin muodostaa formaali kuvaus. Kuvauksen avulla vanhasta järjestelmästä saataisiin kattava käsitys ja toiminnallisuuden toteuttaminen uuteen järjestelmään voitaisiin pohjustaa valmiiksi todistettuun ja tarkistettuun tapaukseen.

Formaalien kielten käytöstä määrittelyn työkaluna on todistettavasti runsaasti hyötyjä, mutta niiden yleistyminen ei näytä kuitenkaan kovin todennäköiseltä lähitulevaisuudessa. Jotta formaalien menetelmien käyttö voisi toden teolla lisääntyä, vaatisi se kokonaan uuden formaalin kielen kehittämistä. Uuden kielen pitäisi olla helpommin lähestyttävä ja sisäistettävissä pienemmällä oppimiskynnyksellä. Myös laadukkaiden ohjelmistojen ja käyttöä tukevien työkalujen saatavuus olisi ehdoton vaatimus.

Avainsanat: vaatimusmäärittely, formaalit menetelmät, ketterä ohjelmistokehitys

ABSTRACT

Joonas Koski: Specification methods in agile software development
Master of Science Thesis
Tampere University
Information Technology
May 2020

Nowadays it's a common trend to try and improve all processes to extreme and try to achieve visible results immediately. Reacting to changing situations fast and flexibly is becoming a new norm. This kind of trend has also taken over in software development as projects are being managed more and more with agile methods. Instead of developing software for longer periods at a time, there's been a shift into a model, where software is delivered in smaller pieces, expanding available functionality gradually. The key element in agile development methods is to keep the customer as close to the process as possible and constantly involved in the development of the software. This improves the quality of the system and ensures that the software is more likely to fulfil the needs of the customer. Constant communication with the customer and other stakeholders means also that the requirements process becomes even more important part of the project as requirements are being changed, fixed and added throughout the project.

This master's thesis focuses on requirements engineering in agile software development, especially on the methods that are used to specify the requirements. The work is heavily focused towards studying mathematical formal methods, which are used for verification and checking correctness of requirements. Formal methods are typically used for improving software quality by trying to locate and fix errors in system functionality as early as possible. In addition to improving quality, use of formal methods may help reduce project costs and staying in schedule. This master's thesis was assigned by Konecranes and done as a part of their software development project. The idea was to study requirements engineering practices in a real-world agile software development project and be able to draw conclusions from the used methods. Most of the study towards specification methods was done by getting familiarized with literature and other studies done in the field. The goal was to find out what kind of practices are involved in using formal methods with agile development methods and is the usage of formal methods possible in modern software development.

Even though formal methods have been developed and used for decades, they still haven't become a mainstream tool when developing software. Most apparent reason for this is the steep learning curve of formal methods. To gain full benefits of formal methods, one should be able to master both mathematical and logical aspects of the language. The problem is heightened when this kind of expertise is required not only from the person creating the specifications, but also from the developer reading those specifications.

Throughout the existence of formal methods, there has been various attempts in trying to solve these problems. One of the proposed solutions has been an approach to use a lightweight model of using formal methods. In this model, instead of formalizing the whole development process, only the parts that are critical for security are exposed to formal methods. This will allow the use of traditional methods for rest of the development tasks and still improve the quality in critical areas. Another possible use for formal methods could be in modernization projects where a new system is being developed to replace an older version. In this kind of projects, a formal description of the old system could be created to help give a better understanding of the old system as well as provide foundation for verified system requirements in developing the new system.

The use of formal methods as a tool is proven to be beneficial, but their widespread use is still very unlikely in the future. If usage of formal methods were to increase, a completely new language would have to be developed. This new language would have to be easier to approach and provide a more gradual learning curve. Also, high quality supporting software and tools would be an absolute requirement.

Keywords: Requirements engineering, formal methods, agile software development

ALKUSANAT

Diplomi-insinööriksi opiskeleminen ei ole ollut minulle tavoitteena lähellekään aina. Ammattikoulun jälkeen heikon työllisyystilanteen seurauksena lähdin opiskelemaan tietotekniikkaa ammattikorkeakouluun ja valmistumisen jälkeen ajattelin ensin opintojen olleen siinä. Hyvin pian mieleen kuitenkin hiipi ajatus, että olisiko kuitenkin mahdollista opiskella vielä hieman lisää ja valmistua diplomi-insinööriksi? Viimeinen niitti haaveelle oli, kun tyttöystäväni keskusteli ystävänsä kanssa ja kysyttäessä, olinko siis diplomi-insinööri, hän viittasi minuun ”pelkkänä” insinöörinä.

Muutaman työelämässä vietetyn vuoden jälkeen päätin vihdoin tehdä haaveelle jotain ja hain Tampereen teknilliseen yliopistoon suoraan maisterivaiheeseen. Voinkin olla kiitollinen Suomen tarjoaman koulutusjärjestelmän mahdollistaessa kyseisenlaisen, ehkä hieman epätavallisenkin opintopolun.

Haluan kiittää Konecranesia mahdollisuudesta tämän diplomityön tekemiseen. Kiitos Matti Lehdolle arvokkaasta ohjauksesta ja avusta työn aiheen muodostamisessa sekä tsemppaamisesta työn aikana. Kiitos myös professori Kari Syställe kirjoittamisen haastamisesta ja työn muodostumisessa ohjanneista kommenteista.

Tampereella, 3.5.2020

Joonas Koski

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

Alloy	Avoimen lähdekoodin mallinnuskieli
ASCII	American standard code for information interchange
BRS	Business requirements specification
ConOps	Concept of operations
CZT	Community Z tools
FME	Formal methods Europe
IBM	International business machines corporation
IEEE	Institute of electrical and electronics engineers
ISO/IEC	Kansainvälinen standardointijärjestö
Kanban	Työn näkyvyyttä korostava ketterä menetelmä
LaTeX	Ladontajärjestelmä tieteelliseen kirjoittamiseen
MVC	Model view controller
OMG	Object management group
OMT	Object modeling technique
OOSE	Object oriented software engineering
PL/I	Programming language/One
rCos	Refinement of component and object systems
Scrum	Ketterä projektinhallintamenetelmä
SDLC	Software development life cycle
SRS	Software requirements specification
SyRS	System requirements specification
StRS	Stakeholder requirements specification
UML	Unified modeling language
VDM	Vienna development method
TDD	Test driven development
XP	Extreme programming

1. JOHDANTO

Vaatimusmäärittelyllä on tärkeä rooli ohjelmistoprojektin onnistumisen kannalta. Huolella tehty vaatimusmäärittely säästää projektin kustannuksia ja auttaa pysymään aikataulussa. Samalla tuotetun ohjelmiston laatu paranee ja sen sisältämät ominaisuudet täyttävät paremmin asiakkaan tarpeet. Huonosti toteutettu vaatimusmäärittely saattaa johtaa väärinymmärrettyjen määrittelyiden myötä virheisiin ohjelmistossa, joiden korjaaminen voi pahimmillaan vaatia valtavasti rahaa ja muita resursseja sekä viivästyttää ohjelmiston saattamista valmiiksi.

Tässä työssä on tarkoitus tutustua vaatimusmäärittelyn käytäntöihin ja perehtyä määrittelyiden tekemiseen käytettäviin menetelmiin. Määrittelymenetelmistä erityisesti formaaleilla menetelmillä on työn kannalta merkittävä rooli. Formaaleilla menetelmillä tarkoitetaan matemaattisia ja loogista päättelyä hyödyntäviä menetelmiä, joiden tavoitteena on auttaa havaitsemaan määrittelyihin liittyviä virheitä aikaisessa vaiheessa ja näin edesauttaa laadukkaamman ohjelmiston tuottamisessa.

Formaaleista menetelmistä tutustutaan niiden piirteisiin ja ominaisuuksiin, käyttöön nykypäivänä sekä tulevaisuuden näkymiin. Työn puitteissa on tarkoitus tutustua tarkemmin kahteen formaaliin menetelmään; Z-notaatioon ja Vienna development methodiin. Nykypäivän ketterä ohjelmistokehitys tuo omat vaatimuksensa myös määrittelytyölle, joten tällä on vaikutus luonnollisesti myös sille, miten formaaleja menetelmiä voidaan käyttää ketterien menetelmien yhteydessä. Työssä on tarkoitus selvittää, onko formaalien menetelmien käyttö ketterien menetelmien yhteydessä yleistä ja minkälaisia ratkaisuja tähän on kehitetty sekä minkälaisen ohjelmiston määrittelyssä formaaleista menetelmistä olisi erityisesti hyötyä.

Diplomityö tehtiin osana työn toimeksiantajana toimineen Konecranesin ohjelmistoprojektia, jonka yhteydessä tutkimus oli tarkoitus suorittaa. Projektista oli tarkoitus ammentaa käytännön huomioita ketterässä ohjelmistokehityksessä tehtävästä vaatimusmäärittelystä sekä formaalien menetelmien käyttöön liittyvistä vaatimuksista. Muuten tutkimus oli tarkoitus toteuttaa perehtymällä alan kirjallisuuteen sekä aiempiin tutkimuksiin.

Työ rakentuu kuudesta luvusta. Toisen luvun tarkoitus on pohjustaa lukijalle ohjelmistokehityksen käytänteitä ja auttaa muodostamaan käsitys vaatimusmäärittelyn roolista oh-

jelmiston elinkaaren osana. Kolmannessa luvussa perehdytään tarkemmin vaatimusmäärittelyn piirteisiin sekä vaatimusmäärittelyn jakautumiseen viiteen vaiheeseen; esillesaantiin, analyysiin, dokumentointiin, validointiin ja vaatimustenhallintaan. Neljännessä luvussa esitellään määrittelyiden tekemiseen käytettäviä menetelmiä aina formaaleista menetelmistä puoliformaaleihin menetelmiin ja lopulta informaaleihin menetelmiin. Viidennessä luvussa syvennytään vielä tarkemmin formaalien menetelmien käyttöön ketterässä ohjelmistokehityksessä ja siihen, miten formaaleja menetelmiä käytetään nykypäivänä. Kuudennessa luvussa analysoidaan saatuja tuloksia ja pohditaan formaalien menetelmien käyttömahdollisuuksia nykypäivänä. Työn viimeisessä luvussa muodostetaan yhteenveto ja kerrataan saadut tulokset.

2. KETTERÄ OHJELMISTOKEHITYS

Muuttuvan maailman ja sitä myöden jatkuvien teknologian, liiketoiminnan, sosiaalisten rakenteiden sekä ihmisten asenteiden muutosten vuoksi myös perinteinen ohjelmistojen kehitys on ollut muutospaineen alla. Vuosien saatossa ohjelmistokehitystyötä helpottamaan, tehostamaan ja nopeuttamaan on luotu erilaisia toimintamalleja, näitä toimintamalleja kutsutaan usein ketteriksi (engl. agile) ohjelmistokehitysmetodeiksi (Holcombe, 2008).

Tämän luvun tarkoitus on esitellä ohjelmistokehitystä kokonaisuutena ja auttaa lukijaa muodostamaan käsitys perustasolla sen käytänteistä, prosesseista ja vaiheista sekä ketteryyden tuomista erityispiirteistä. Tämän työn aiheena oleva vaatimusmäärittely on vain yksi osa ohjelmistokehityksen eri vaiheista.

2.1 Ohjelmistokehitysprosessin vaiheet ja ohjelmiston elinkaari

Ohjelmistokehitysprosessi voidaan mieltää sarjana toisiinsa liittyviä tapahtumia, jotka johtavat ohjelmiston syntyyn. Johtuen siitä, että eri tyyppisiä ohjelmistoja on monia, myös käytettäviä prosesseja on erilaisia, eikä tarjolla ole yhtä toimintamallia, joka toimisi kaiken tyyppisille ohjelmistoille. Käytettävä prosessi määräytyy usein kehitettävän ohjelmiston, järjestelmän tilaajan tai asiakkaan antamien vaatimusten ja ohjelmistoa kehittävien ihmisten taitojen perusteella (Sommerville, 2016, s. 44). Näiden lisäksi myös ohjelmiston kehitystyöhön suunniteltu aikataulu sekä kehityksen lähtötilanne ja valmiina oleva aineisto ja tietämys voivat vaikuttaa menetelmän valintaan.

Sommervillen (2016, s. 44) mukaan eri ohjelmistokehitysprosesseihin kuuluu, niiden eroista huolimatta, ainakin alla olevat neljä perusvaihetta.

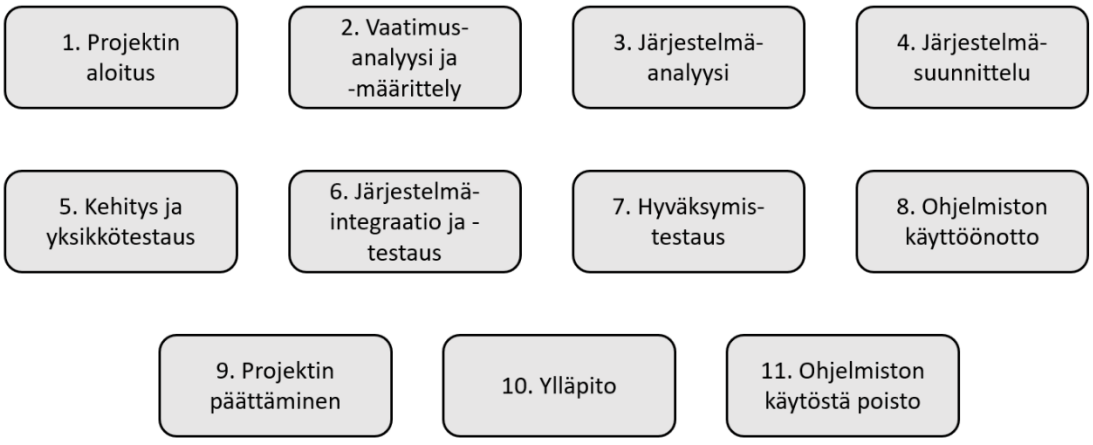
1. Ohjelmiston määrittely
2. Ohjelmiston kehitys
3. Ohjelmiston validointi
4. Ohjelmiston evoluutio

Ohjelmiston määrittelyvaiheessa selvitetään asiakkaan vaatimukset sekä selvitetään mitä toiminnallisuutta ohjelmiston tulisi tarjota, kuten myös mahdolliset ohjelmistoa tai sen toimintaa koskevat rajoitteet. Ohjelmiston kehitysvaiheessa aiemmin kerättyjä vaa-

timuksia vastaava ohjelmisto toteutetaan. Validointivaiheessa taas varmistetaan, että toteutettu ohjelmisto vastaa asiakkaan asettamia vaatimuksia. Viimeisellä, evoluutiovaiheella, tarkoitetaan ohjelmiston valmiiksi saattamisen jälkeistä aikaa, jolloin ohjelmiston tulee kehittyä vastaamaan asiakkaan muuttuvia tarpeita (Sommerville, 2016, s. 44).

Ohjelmistoja kehitettäessä puhutaan usein myös elinkaaresta (engl. life cycle), joka kattaa ohjelmiston koko eliniän eri vaiheineen ensimmäisestä ideasta tai tarpeesta, kehityksen kautta ylläpitoon ja ohjelman käytöstä poistoon. Tällaisesta elinkaaresta käytetään usein termiä SDLC (engl. software development life cycle).

Leon (2015, s. 18) jakaa SDLC:n yhteentoista eri vaiheeseen (Kuva 1); projektin aloitus (engl. project start-up), vaatimusanalyysi ja -määrittely (requirements analysis and requirements specification), järjestelmäanalyysi (engl. system analysis), järjestelmäsuunnittelu (engl. system design), ohjelmiston kehitys ja yksikkötestaus (engl. development and unit testing), järjestelmäintegraatio ja -testaus (engl. system integration testing), hyväksymistestaus (engl. acceptance testing), ohjelmiston käyttöönotto (engl. implementation), projektin päättäminen (engl. project windup), ylläpito (engl. maintenance) sekä ohjelmiston käytöstä poisto (engl. retirement). Kuhunkin vaiheeseen sisältyy joukko erinäisiä tapahtumia tai toimia, joita ei suinkaan kaikkia suoriteta jokaisessa projektissa. Projektin läpivientiin sisältyvät toimet ja vaiheet vaihtelevat projektin koon, luonteen ja monimutkaisuuden mukaan ja yllämainitut vaiheet kokonaisuudessaan sisältyvät usein vain suurimpiin ohjelmistoprojekteihin.



Kuva 1. *Ohjelmiston elinkaaren eri vaiheet.*

Projektin aloitusvaiheessa muodostetaan tarvittava projektitiimi ja valitaan projektille vetäjä. Tähän vaiheeseen kuuluu myös toimia, jotka ovat käytännössä yhteisiä mille tahansa projektille, ei vain ohjelmistoprojekteille, kuten budjetin ja projektisuunnitelman

määrittäminen, ylemmän tason vaatimuksiin tutustuminen sekä projektin tavoitteiden asettaminen. Aloitusvaiheen yhteydessä yleensä myös hankitaan tarvittava laitteisto ja ohjelmistot sekä varmistetaan tarvittavien kehitys- ja testausympäristöjen toiminta (Leon, 2015, s. 18-19).

Vaatimusanalyysin ja -määrittelyn aikana ohjelmalle asetetut vaatimukset pyritään selvittämään ja dokumentoimaan riittävän selkeällä ja yksityiskohtaisella tasolla, jotta ohjelman toteuttaminen niiden avulla on mahdollista. Leon (2015, s. 20) mainitsee myös, että tässä vaiheessa suunnitellaan projektin resursointi ja arvioidaan tehtäviin kuluvaa aikaa sekä jaetaan tehtävät tiimin jäsenten kesken. Tämän vaiheen keskiössä on kuitenkin vaatimusten ymmärtäminen, johon kuuluu nykyisen järjestelmän ongelmakohtien, puutteiden, rajoitteiden ja mahdollisuuksien selvittäminen. Tämä tapahtuu usein haastattelemalla käyttäjiä ja tutkimalla saatavilla olevaa dokumentaatiota. Täysin uutta järjestelmää kehitettäessä, tämän vaiheen aikana pyritään ymmärtämään funktiot ja toiminnot, jotka järjestelmän tulisi toteuttaa. Näiden tietojen avulla muodostetaan ja dokumentoidaan käyttäjävaatimukset (engl. user requirements), joiden avulla voidaan siirtyä seuraavaan vaiheeseen.

Järjestelmäanalyysissä aiemman vaiheen käyttäjävaatimukset käydään läpi ja pohditaan eri vaihtoehtoja näiden toteuttamiseen. Vaihtoehtoja sitten arvioidaan ja sopivista ratkaisuista muodostetaan järjestelmäkuvaus, jota voidaan ehdottaa asiakkaalle. Jos käyttäjävaatimuksissa on epäselvyyksiä ja niiden selkeä ymmärtäminen on riskinä projektin kannalta, voidaan tässä vaiheessa tehdä prototypointi, jonka avulla asiakkaalle voidaan demonstroida mahdollista ratkaisua (Leon, 2015, s. 21-22).

Järjestelmäsuunnittelun aikana voidaan tehdä sekä korkean että matalan tason suunnittelua. Korkean tason suunnitteluun kuuluu järjestelmään tulevien moduulien, ohjelmien, funktioiden ja rutiinien tunnistamista ja määrittämistä. Myös käyttöliittymään kuuluvat navigointi, valikot, virheviestit ja raportit sekä ohjelman syötteet ja tulosteet suunnitellaan tässä vaiheessa. Muita suunniteltavia asioita on ohjelman järjestelmäarkkitehtuuri, käytettävät teknologiat, moduulien välinen kommunikointi sekä ohjelman mahdolliset riippuvuudet ja tietokantarakenteet. Korkean tason suunnittelun jälkeen päätettyjä ratkaisuja voidaan tarkentaa matalan tason suunnittelulla, jolloin päätetään mitä kirjastoja, yleisiä rutiineja tai ohjelmarunkoja käytetään. Myös mahdolliset uudelleen käytettävät ohjelman osat voidaan tunnistaa ja suunnitella tässä vaiheessa (Leon, 2015, s. 23-24).

Seuraavana vuorossa on varsinainen toteutusjakso, eli ohjelmointi- ja yksikkötestausvaihe, jossa suunnitellut ohjelmat, kirjastot, moduulit, funktiot ja muut ohjelman osat ohjelmoidaan ja testataan. Vaiheen tuotoksena on yksikkötestattu ohjelma testausraportteineen ja lokeineen (Leon, 2015, s. 25)

Järjestelmäintegraatiolla varmistetaan, että ohjelman osat toimivat keskenään ja ohjelmalle tarkoitettu ympäristö toimii sekä mahdolliset yhteydet tietokantaan ja verkkoon toimivat. Tähän vaiheeseen kuuluu myös järjestelmätestaus, jossa testataan kokonaisuuden toimintaa ja kerätään tarvittavaa palautetta testaajilta. Monesti myös puhutaan alfa ja beeta testaamisesta, joissa rajattu joukko käyttäjiä testaa valmista ohjelmistoa käytettävyyden ja mahdollisten virheiden varalta. Eroina näillä on se, että alfa testauksessa testataan ohjelmiston toiminnallisuutta hyvin rajatulla joukolla testaajia ja beeta testauksessa testaajien määrää on kasvatettu, mutta samalla keskitytään vain ohjelmallisten virheiden löytämiseen (Leon, 2015, s. 26).

Kun ohjelmisto on todettu valmiiksi, on hyväksymistestauksen vuoro. Tyypillisesti vaiheella on suurempi rooli, kun tehdään erikseen tietylle asiakkaalle räätälöityä ohjelmistoratkaisua ja toteutetut ominaisuudet perustuvat asiakkaan esittämiin vaatimuksiin. Näissä tapauksissa testauksen suorittaa yleensä asiakas itse, tai jokin asiakkaan valtuuttama taho. Hyväksymistestauksen tarkoitus on verrata ohjelmiston toimivuutta hyväksymiskriteereihin ja auttaa asiakasta päättämään täyttääkö ohjelmisto sille alussa asetetut vaatimukset. Kehitystiimin tehtävänä tässä vaiheessa on auttaa testaamisessa sekä korjata mahdolliset virheet ja puutteet, jotka testin aikana löydetään (Leon, 2015, s. 27).

Integraatiotestauksen, asiakkaan hyväksymistestauksen ja näissä vaiheissa löydettyjen virheiden tai puutteiden korjauksen jälkeen ohjelma on lopullisesti valmis ja valmiina luovutettavaksi asiakkaalle. Seuraavana vaiheena ohjelmiston elinkaaressa on siis ohjelmiston käyttöönotto, jossa ohjelmisto asennetaan asiakkaan ympäristöön ja asiakkaan henkilökunnalle annetaan tarvittaessa koulutusta ohjelmiston käytöstä (Leon, 2015, s. 27).

Käyttöönoton jälkeen tehdään projektin päättäminen, jossa projektille varatut resurssit vapautetaan, mahdolliset asiakkaalta lainassa olleet tuotteet palautetaan ja suoritetaan kaikki muut projektin päättämiseen liittyvät toimet (Leon, 2015, s. 28).

Ohjelmiston kehityksen ja käyttöönoton jälkeen ohjelmisto siirtyy ylläpitovaiheeseen, jossa ohjelmiston käyttäjille annetaan tukea mahdollisissa ongelmatilanteissa sekä mahdolliset ohjelmistovirheet, jotka ovat jääneet huomaamatta aiemmissa testausvaiheissa, korjataan (Leon, 2015, s. 28). Monessa tapauksessa ohjelmiston kehitystä saatetaan

myös jatkaa ylläpitovaiheen aikana. Asiakkaalle saattaa tulla uusia vaatimuksia ja vanhan ohjelmiston muokkaaminen on halvempaa ja nopeampaa, kuin kokonaan uuden ohjelmiston kehittäminen.

Ohjelmiston elinkaaren lopussa, kun ohjelmisto on ollut käytössä vuosia, tulee vastaan vaihe, jolloin ohjelmiston ylläpito ei ole enää kustannustehokasta. Tämä voi johtua esimerkiksi siitä, että uuden ominaisuuden lisääminen vanhaan ohjelmistoon muuttaisi sen perustaa niin radikaalisti, että jopa kokonaan uuden ohjelmiston tekeminen voisi osoittautua kustannustehokkaammaksi. Muita syitä voivat olla tarve vaihtaa ohjelmiston laitteisto uuteen, joka ei tue enää vanhaa ohjelmistoa, tai jos järjestelmä on ajautunut tilaan, jossa pienikin muutos yhteen moduuliin saattaa vaarantaa koko järjestelmän toimivuuden (Leon, 2015, s. 28).

Monet näistä vaiheista saattavat toimia lomittain tai osittain päällekkäin. Esimerkiksi nykyisissä ohjelmistoprojekteissa kehitystä ja integraatiota saatetaan tehdä saman aikaisesti, kun valmiiksi testatut ohjelman osat integroidaan heti osaksi järjestelmää.

2.2 Ketteryyden määritelmä ja Agile manifesto

Kuten aiemmin jo todettiin, ketteriä menetelmiä on aloitettu kehittämään vastaamaan alati muuttuvan maailman tarpeisiin. Bisneskulttuuri on muuttunut siihen, että tuotteita on saatava nopeammin käytettäväksi, vaikka se tarkoittaisi rajatumpaa määrää ominaisuuksia.

Perinteisesti ohjelmiston laatua ja onnistumista on voitu mitata neljällä eri mittarilla; *tuottavuus* (engl. productivity), joko yksittäisten toimijoiden tai kokonaisten tiimien, jolloin mittarina voi toimia esimerkiksi tuotettujen koodirivien määrä. *Markkinoille saattamiseen kuluva aika* (engl. time-to-market), jolloin voidaan mitata keskimääräisiä aikoja projektilla valmistua, tai sille, että saadaan toimitettua jotain merkityksellistä ja uutta lisäarvoa tuotteelle. *Käytetyn prosessin kypsyyttä* (engl. process maturity), jossa seurataan projektin käytänteiden johdonmukaisuutta, yhtenäisyyttä ja standardointia. Sekä t*oimitetun koodin laatua* (engl. quality), jolla tarkoitetaan koodissa olevien virheiden määrää ja vakavuutta. Tätä mitataan usein virheiden esiintymistiheyden ja tietyn aikaikkunan puitteissa korjattujen virheiden määrän yhdistelmillä. Nykypäivänä, kun vaatimuksena on vastata yhä nopeammin asiakkaan muuttuviin vaatimuksiin, näiden mainittujen kriteerien täyttäminen on yhä vaikeampaa. Tämän vuoksi organisaatiot ovat siirtymässä nopeasti käyttämään ketteriä menetelmiä ohjelmistojensa kehityksessä (Ali Babar & Brown & Mistrik, 2013, s. xxxi).

Ketterillä menetelmillä yritetään yleensä tavoitella nopeampaa kehitysprosessia, jossa keskiössä on muutoksiin reagointi ja asiakkaan tuominen mukaan mahdollisimman lähelle prosessia. Projektin kannalta juuri asiakkaalla on laajin tietämys järjestelmälle asetettavista vaatimuksista, joten jatkuva keskustelu projektin kuluessa auttaa projektissa onnistumisessa.

Cooke (2012, s. 29-30) määrittelee ketteryyden kollektiivisena terminä metodologioille, joiden tarkoituksena on lisätä ohjelmistoratkaisujen relevanttiutta, laatua, joustavuutta ja liiketoiminnallista arvoa. Häneen mukaansa ketterillä menetelmillä on yhteisenä ainakin seitsemän eri peruspiirrettä; etukäteen tehtävän suunnittelun korvaaminen inkrementaalisella suunnittelulla, laadun rakentaminen etukäteen merkittävänä osana prosessia, teknisten riskien huomioiminen mahdollisimman aikaisessa vaiheessa, muutosten mahdollistaminen alkuperäisiin vaatimuksiin samalla minimoiden niiden mahdolliset haittavaikutukset, jatkuvan bisnesarvon toimittaminen priorisoimalla korkeimman arvon tehtävät ensimmäiseksi, henkilökunnan kannustaminen ja tukeminen näiden korkean arvon tehtävien suorittamisessa sekä jatkuvaan kommunikaatioon kannustaminen projektitiimin ja liiketoiminnan edustajien välillä. Näiden lisäksi Sommerville (2016, s. 74) mainitsee kehitystyötä tukevien työkalujen käytön, esimerkiksi erinäiset automatisoinnit liittyen testaukseen tai käyttöliittymän luomiseen sekä muutosten hallintaa tukevat työkalut.

Yhtenä ketterien menetelmien määrittelevänä tekijänä on myös pidetty 2001 kirjoitettua Agile manifestoa, joka määrittelee ketteryyden neljän eri piirteen ympärille, joilla pyritään vastaamaan perinteisten menetelmien ongelmakohtiin.

Näitä piirteitä ovat yksilöiden ja vuorovaikutuksen merkityksellisyys prosessien ja työkalujen sijaan, keskittyminen toimittamaan toimivaa ohjelmistoa laajan dokumentaation sijasta, vuorovaikutus asiakkaan kanssa projektin aikana pelkän sopimusneuvottelun varassa olemisen sijaan ja projektin aikana tapahtuviin muutoksiin reagoiminen tarkan suunnitelman noudattamisen sijasta (Agile alliance, 2001).

Ketteryyttä tarkasteltaessa pääpiirteiksi nousevat siis nopeus, muutoksiin reagointi sekä tiivis yhteistyö asiakkaan kanssa. Asiakkaan kanssa kommunikoimalla sekä palautetta keräämällä voidaan tehdä muutoksia järjestelmälle asetettaviin vaatimuksiin ja näihin pitää pystyä reagoimaan projektin vaiheesta riippumatta riittävän tehokkaasti ja nopeasti. Ylimääräinen byrokratia ja dokumentointi pyritään jättämään pois ja liiallisen etukäteen tehdyn suunnittelun sijasta suunnittelua tehdään iteroiden pitkin projektia. Tästä syystä ketterät menetelmät ovatkin kasvattaneet suosiotaan ohjelmistoja kehitettäessä, sillä niiden avulla voidaan toimittaa tarkemmin asiakkaan vaatimuksiin räätälöityjä ratkaisuja ja

konkreettista liiketoiminnallista hyötyä tarjoavia ominaisuuksia, aiempaa nopeammin ja kuitenkin tinkimättä lopullisen tuotteen laadusta.

2.3 Ohjelmistokehitysmenetelmät

Ohjelmistokehitysprosessia ohjaamaan on kehitetty erilaisia kehitysmalleja ja -menetelmiä, joiden tehtävänä on auttaa kehitystyön etenemisessä ja hallinnassa sekä tarjota työkaluja kehitysprosessin eri vaiheisiin. Despan (2014) mukaan ohjelmistokehitysmenetelmät ovat joukko sääntöjä ja ohjeita, joita käytetään ohjelmiston määrittelyssä, suunnittelussa, kehityksessä, testauksessa, asentamisessa ja ylläpidossa. Menetelmä voi tarjota myös joukon ydinarvoja, joita projektitiimin tulisi noudattaa.

Despa (2014) listaa tutkimuksessaan joukon erilaisia menetelmiä, joista niin sanottuja perinteisiä menetelmiä edustavat esimerkiksi vesiputous (engl. waterfall), prototypointi (engl. ptorotyping), spiraali (engl. spiral) sekä V-malli (engl. V-model). Ketteristä menetelmistä mukana ovat iteratiivinen ja inkrementaalinen malli (engl. iterative and incremental), XP (engl. extreme programming) sekä Scrum.

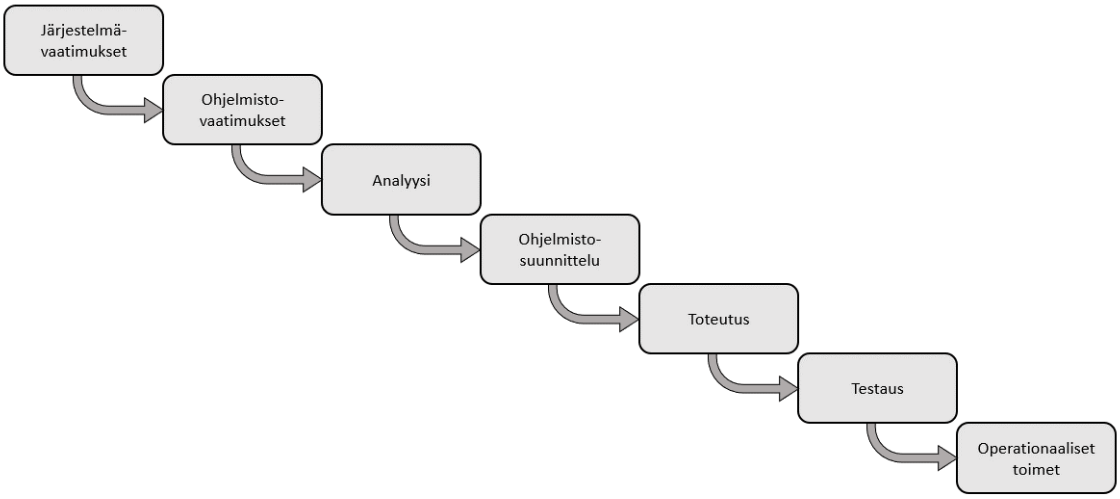
Vaikka nykypäivänä ketterät menetelmät ovatkin pinnalla ja niitä käytetään paljon ohjelmistokehityksessä, ei ketteryys kuitenkaan sovellu kaikkeen. Esimerkiksi tietoturvaltaan kriittiset järjestelmät ja niiden kehitysprojektit, jotka vaativat järjestelmällisempiä menetelmiä, voivat hyötyä edelleen perinteisestä vesiputousmallista. Tutustutaan tässä työssä yleiskuvan tasolla tähän perinteiseen vesiputousmalliin sekä ketteristä menetelmistä Scrum- ja KanBan menetelmiin.

2.3.1 Vesiputous

Vesiputousmallia pidetään usein perinteisimpänä ohjelmistokehitysmallina ja sen avulla ohjelmistokehitykseen liittyvät vaiheet ja niiden yhteydet onkin helppo ymmärtää. Mallin syntyhetkenä pidetään Roycen (1970) kirjoittamaa artikkelia, jossa hän kuvaa prosessin, jolla suuria ohjelmistojärjestelmiä kehitetään. Hän ei kuitenkaan missään vaiheessa artikkelia mainitse mallin nykyistä vesiputous nimeä, vaan nimi on kehittynyt mallille vasta myöhemmin.

Cooke (2012, s. 34) mainitsee, että malli perustuu ajatukseen, jossa kukin kehitysprosessin vaihe on asetettu järjestykseen ja ennen seuraavaan vaiheeseen siirtymistä edellinen vaihe pitää olla suoritettuna kokonaan (Kuva 2). Tämä tarkoittaisi sitä, että esimerkiksi suunnitteluvaihetta ei voida aloittaa ennen kuin kaikki ohjelmiston vaatimukset on kerätty ja selvitetty. Tämä voidaan mieltää yleiseksi käsitykseksi vesiputousmallista, vaikkakin Royce (1970, s. 328) tarkentaakin omassa kuvauksessaan mallia ehdottamalla

iteratiivisuuden lisäämistä. Tällöin projektin edetessä ja vaatimusten sekä suunnitelmien tarkentuessa voidaan tarvittaessa siirtyä iteratiivisesti vaiheiden välillä, kuitenkin pysyen vain viereisissä vaiheissa.



Kuva 2. *Vesiputousmallin vaiheet (Royce, 1970, s. 329).*

Vaikka vesiputousmalli onkin jo nykypäivänä vanhaa perua, on se silti pitänyt pintansa ohjelmistokehityksessä. Pienemmissä projekteissa ketterät menetelmät ovat korvanneet sen, mutta kuten Sommerville (2016, s. 45) mainitsee, vesiputousmalli on sopiva erityisesti turvallisuuskeskeisten ohjelmien kehittämiseen tai silloin, kun tarvitaan selkeästi jaoteltua tai jäsenneltyä kehitysprosessia. Cooke (2012, s. 35) kertoo, että vesiputousmallin kaltaisen suunnitteluvetoisen mallin tarkoituksena oli vähentää riskejä projektitoimitukseen liittyen, kun jokaisen vaiheen valmistuminen vaati johdon hyväksyntää. Todellisuudessa vesiputousmallin on todettu jopa lisäävän projektin epäonnistumisen riskiä, sillä se pakottaa laatimaan suuret määrät dokumentteja projektin alkuvaiheessa, tuoden mukanaan tähän liittyviä riskejä sekä rajoittaa vaatimusten muuttumiseen reagointia projektin edetessä. Despa (2014, s. 51) mainitsee tutkimuksessaan vesiputousmallin vahvuuksiksi helpon hallittavuuden sekä helposti ymmärrettävän prosessin. Mallin heikkouksiin kuuluu muuttuviin vaatimuksiin mukautumisen lisäksi toimivan koodin toimittaminen erittäin myöhäisessä vaiheessa projektia sekä matala kynnys sietää suunnitteluvaiheen virheitä.

2.3.2 Scrum

Kun tarkastellaan ketterää ohjelmistokehitystä, hyvin usein ensimmäisenä menetelmänä mieleen tulee Scrum, joka on lähinnä projektin hallintaan liittyvä iteratiivinen menetelmä. Se ei suinkaan ole sidottu vain ohjelmistojen kehittämiseen, vaan sitä voidaan käyttää

mihin tahansa projektiluontoiseen työhön. Sen iteratiivisen luonteen vuoksi se kuitenkin soveltuu erittäin hyvin juuri ohjelmistojen kehittämiseen sekä muihin IT-alan kehitys- ja ylläpitoprojekteihin.

Scrumin mukaiseen työskentelyyn kuuluu muutamia sille ominaisia piirteitä, kuten työn jaksottaminen muutaman viikon mittaisiin sprintteihin (engl. sprint), prosessiin kuuluvien avainhenkilöiden ja roolien tunnistamista, erilaisten projektin kulkuun liittyvien tapaamisten ja palaverien pitämistä sekä projektiin liittyvien tehtävien jakamista ja priorisointia tehtävälistojen avulla.

Oikeaoppinen Scrum prosessi vaatii siis ainakin kolmen eri roolin tunnistamista. Jokaisella Scrum projektilla tulee olla nimettynä tuoteomistaja (engl. product owner), jonka tehtävänä on vastata bisnekseltä tulevien vaatimusten täyttämisestä sekä tehtävien priorisoimisesta projektin aikana. Toinen tärkeässä roolissa oleva henkilö on Scrum-mestari (engl. scrum master), joka ohjaa projektitiimin työskentelyä sekä varmistaa oikeiden Scrum käytänteiden käytön projektin aikana ja opastaa tarvittaessa niiden käytössä. Viimeisenä roolina, tai joukkona on varsinainen Scrum tiimi (engl. scrum team), joka koostuu monialaisista osaajista, mahdollistaen itsenäisen toiminnan ja mahdollisten ongelmien ratkaisemisen tiimin sisäisin voimavaroin (Cooke, s. 44).

Projektiin kuuluvat tehtävät jaetaan Scrum prosessin mukaan tuotetta koskevaan tehtävälistaan (engl. product backlog) sekä kutakin sprinttiä koskevaan tehtävälistaan (engl. sprint backlog). Tuoteomistajan tehtävänä on vastata priorisoinnista sekä yhdessä Scrum tiimin kanssa valita, mitkä tehtävät otetaan mukaan seuraavaan sprinttiin toteutettavaksi. Sprinttien tilaa seurataan tehtävätaulun avulla, johon sprintin tehtävät on listattu ja niiden nykyisestä tilasta on mahdollista muodostaa käsitys nopeasti. Kukin sprintti kestää usein 2-4 viikkoa, jonka aikana kehitettäväksi valitut tehtävät on tarkoitus saattaa valmiiksi. Näitä sprinttejä toteutetaan iteratiivisesti, ja tuotetta kehitetään näin pala palalta valmiiksi (Cooke, s. 45-46).

Oleellisena osana Scrumin käytäntöjä ovat myös erilaiset tapaamiset ja palaverit, joihin kuuluvat päivittäiset tiimitapaamiset (engl. daily meetings), sprinttien suunnittelupalaverit (engl. sprint planning) sekä sprinttien lopussa tehtävät katselmukset (engl. sprint review). Päivittäisissä tapaamisissa tarkoituksena on käydä läpi mahdolliset ongelmakohdat sekä kehitettävänä olevien tehtävien sen hetkinen tilanne. Näihin palavereihin osallistuu koko Scrum tiimi ja ne pidetään usein aamun aikana työpäivän alussa, jolloin kaikilla tiimin jäsenillä on mahdollisuus muodostaa yleiskuva projektin kulusta. Sprinttien suunnittelupalavereissa tarkoituksena on käydä läpi tuotteen kehityslistaa ja valita sieltä seuraa-

vaan sprinttiin tulevat tehtävät sekä priorisoida ne valmiiksi kyseisen sprintin tehtävälistaan. Sprintin katselmuksessa taas esitellään kuluneen sprintin aikana tehty työ sekä käydään läpi mahdollisia kehityskohteita ja parannuksia tiimin työskentelytapoihin liittyen, jolloin niihin voidaan pyrkiä reagoimaan seuraavassa sprintissä (Cooke, s. 44-46)

Scrumin iteratiivinen työskentelytapa ja tehtävien pilkkominen pieniin, nopeasti toteutettavissa oleviin osiin, sopii erityisesti selkeästi määritettävissä oleviin projekteihin, joissa työtehtävät eivät kasva liian suuriksi kokonaisuuksiksi. Scrum soveltuu myös erittäin hyvin projektitiimille, jonka jäsenten taitotasot eivät eroa toisistaan suuresti. Näin eri tehtävät voidaan jakaa tasaisemmin jäsenten kesken ja projektin eteneminen ei esty esimerkiksi jäsenen sairastumisen vuoksi. Vastaavasti Scrum ei tietenkään sovellu ihan kaikkeen, vaan jonkin tyyppisissä projekteissa voi esiintyä haasteita. Esimerkiksi tarve muuttaa käynnissä olevan sprintin sisältöä voi aiheuttaa ongelmia, sillä sprinttien sisältö ja työmäärä on usein suunniteltu kokonaan etukäteen, joten uuden työn mukaan ottaminen saattaa vaikuttaa muiden tehtävien valmiiksi saattamiseen. Scrum prosessi on myös herkkä tiimin jäsenten vaihtuvuudelle ja uuden jäsenen liittyessä voi mennä aikaa, kunnes prosessi saadaan jälleen toimimaan sulavasti ja uusi jäsen perehdytettyä käytäntöihin (Allbee, 2018).

2.3.3 Kanban

Toinen ketteriin menetelmiin lukeutuva, työkuorman ja muutoksenhallintaan keskittyvä metodi on Kanban. Sitä voidaan käyttää itsenäisesti tai yhdistettynä muiden menetelmien kanssa. Hyvin yleisenä tapana onkin käyttää sekä Scrumia että Kanbania yhdessä. Kanbanin yhtenä perusideana on lisätä projektin näkyvyyttä ulospäin Kanban taulun avulla, jonka kautta projektin sidosryhmät voivat nähdä projektin kulun missä vaiheessa hyvänsä. Toinen Kanbanille tyypillinen piirre on yksilöiden työmäärän rajoittaminen yhteen kerrallaan tehtävään työtehtävään. Kukin tehtävä pitäisi saattaa aina ensin loppuun, ennen kuin seuraavaa tehtävää aloitetaan. Tämä tapahtuu osittain myös Kanbanin keskeneräistä työtä rajoittavien WIP-rajoitteiden (engl. work in progress) avulla. Kanban taululla oleville sarakkeille on kullekin asetettu oma WIP-rajoite, esimerkiksi kehitys- ja testausvaiheilla saattaa olla eri lukemat rajoittamassa samanaikaisten tehtävien määrää, riippuen käytettävissä olevien henkilöiden määrästä ja saraketta vastaavan työn luonteesta. Vaikka tiimissä olisikin vähemmän testaushenkilöitä, kuin kehittäjiä, niin testaustoimet voivat olla nopeampia kuin kehitystehtävät, jolloin voi olla paikallaan käyttää suurempaa WIP-rajoitetta testauksessa (Cooke, s. 54-55).

Toisin kuin Scrumissa, Kanbanissa töitä ei jaksoteta suoraisesti sprintteihin vaan menetelmässä luotetaan töiden läpivirtaukseen (engl. flow). Eli iteratiivisen luonteen sijasta

Kanbanissa seurataan tehtävien läpimenoaikaa ja valmiita tehtäviä kerätään seuraavaan julkaisuun ilman, että kehitystä olisi jaksotettu lyhyempiin aikaikkunoihin (Shalloway, 2010). Tarkoituksena on kuitenkin aina pyrkiä jakamaan tehtävät mahdollisimman pieniin kokonaisuuksiin, jolloin niiden läpimenoaika saadaan tuotua mahdollisimman alas. Läpimenoaikoja seuraamalla voidaan pyrkiä kehittämään prosessia, jos havaitaan jonkin vaiheen muodostavan pullonkauloja tehtävien läpikulussa. Kanbaniin ei myöskään kuulu mitään päivittäisiä tapaamisia, joissa projektin tilaa erikseen selvitettäisiin, vaan tilan voi tarkastaa Kanban taululta missä vaiheessa hyvänsä (Allbee, 2018).

Kanban toimii tehokkaasti, kun tiimiin kuuluu eri taitoja omaavia työntekijöitä, jolloin yhden alan erikoisosaajat voivat ottaa helposti tehtäväkseen heidän osaamistaan vastaavat tehtävät, sillä tehtävien valmistumisen ajankohdalla ei niinkään ole väliä. Kanbanissa myös uusien tehtävien lisääminen on helppoa, jopa priorisointijonon kärkeen, sillä valmiiksi suunniteltuja sprinttejä omine tehtävälistoineen ei ole, vaan tehtäviä otetaan alati muuttuvasta priorisointijonosta (Allbee, 2018).

3. VAATIMUSMÄÄRITTELY

Vaatimusmäärittely toimii ohjelmistoprojektin perustana ja onkin yksi projektin tärkeimmistä vaiheista. Sen tehtävänä on määritellä mitä eri sidosryhmiin kuuluvat käyttäjät, asiakkaat, toimittajat, kehittäjät tai yritykset haluavat uudelta järjestelmältä. Kun projektissa mukana olevia sidosryhmiä on useita, voi myös eriäviä näkemyksiä tulevalle ohjelmistolle asetettavista vaatimuksista olla useita ja osa niistä voi olla jopa ristiriidassa toistensa kanssa. Nämä ristiriidat on kuitenkin välttämätöntä selvittää, jotta voidaan siirtyä ohjelmiston varsinaiseen suunnitteluvaiheeseen (Hull & Jackson & Dick, 2005, s. 2).

Tässä luvussa käydään tarkemmin läpi vaatimusmäärittelyn roolia ohjelmistokehityksessä sekä sen eri vaiheita, metodeja ja käytänteitä. Tutustutaan myös ketteryyden tuomiin erityispiirteisiin ja vaatimuksiin sekä vaatimusmäärittelyä koskeviin standardeihin.

3.1 Vaatimusmäärittelyn rooli ohjelmistokehityksessä

Chakrabortyn, Baowalyn, Arefin ja Baharin (2012 s. 723) mukaan vaatimusmäärittely on ohjelmistokehityksen ja SDLC:n ehdottomasti tärkein vaihe. Vaatimusmäärittelyn aikana tehdyt virheet voivat olla erittäin kalliita korjata, jos ne huomataan vasta kehitystyön loppuvaiheessa. Boehm ja Papaccio (1988, s. 1466) ovat todenneet myös jo 1980-luvun lopulla, että ohjelmistokehitysprosessin myöhemmässä vaiheessa tehtävät korjaukset voivat olla jopa 50-200 kertaisesti kalliimpia, kuin jos ne olisi havaittu aikaisemmassa vaiheessa projektia. Näin ollen vaatimusmäärittelyn aikana tehdyillä virheillä tai huomioimattomilla osa-alueilla voi olla äärimmäisen vakavia seurauksia projektin lopputuloksen kannalta.

Usein myöhäisessä vaiheessa huomatut virheet vaikuttavat lisäkustannusten lisäksi myös projektin aikatauluun. Mitä myöhemmässä vaiheessa projektia virhe havaitaan, sitä enemmän sillä on potentiaalisia vaikutuksia projektin alussa tehtyihin päätöksiin. Projektin alussa päätettyä arkkitehtuuria saatetaan joutua muuttamaan ja tämä vastaavasti voi vaikuttaa rakenteellisesti ohjelmistoon, jolloin virheen korjaamiseen kuluva aika voi olla erittäinkin huomattava ja sitä kautta viivästyttää projektin valmistumista merkittävissä määrin.

Holtin, Perryn ja Brownswordin (2012, s. 1-2) mukaan vaatimusmäärittelylle on viisi tärkeää syytä. Vaatimusmäärittelyn tehtävänä on auttaa ymmärtämään vaatimukset oikein, tämä on välttämätöntä riippumatta siitä, minkälaista järjestelmää ollaan tekemässä ja

projektin onnistumisen kannalta vaatimukset onkin tärkeää ymmärtää aina täysin ja oikein. Vaatimusten tehtävänä on myös ajaa projektia, eli jokainen tehty toiminto tai toimenpide tulisi voida johtaa johonkin vaatimukseen. Vaatimusten oleellisena tehtävänä on myös määrittää tuotteen laatua, sillä lopullisen tuotteen tulisi olla aina sekä vaatimustenmukainen että tarkoituksenmukainen. Vaatimusmäärittelyssä tehtävät vaatimukset toimivat usein myös kriteereinä hyväksymistestauksessa ja tuotteen hyväksynnässä. Vaatimusmäärittely ja selkeästi määritellyt vaatimukset myös kohottavat projektin kulkuun liittyvää itsevarmuutta, sillä varmuus oikein ymmärretyistä vaatimuksista antaa myös varmuutta toteutusvaiheeseen.

3.2 Vaatimukset ohjelmistoprojektissa

Vaatimuksilla tarkoitetaan erinäisiä kuvauksia, jotka selittävät mitä palveluita järjestelmän pitäisi tarjota sekä minkälaisia rajoitteita järjestelmään mahdollisesti kohdistuu (Sommerville, 2016, s. 102). Vaatimusten tavoitteena on siis selvittää mahdollisimman tarkasti, mitä järjestelmältä odotetaan, jotta tarvittavat määritykset voidaan tehdä kehitystyötä varten.

Vaatimuksia pystytään kategorisoimaan riippuen niiden luonteesta ja Sommerville (2016, s. 105) mainitseekin, että yleensä jaottelu tehdään toiminnallisten ja ei-toiminnallisten vaatimusten kesken. Näiden kahden lisäksi Holt, Perry ja Brownsword (2012, s. 98) tarkentavat jaottelua bisnesvaatimuksilla. Bisnesvaatimuksilla tässä tarkoitetaan yleisempiä, liiketoiminnan kannalta tärkeitä vaatimuksia, kuten rahan tekeminen, markkinajohtajan aseman tavoittaminen tai asiakkaan pitäminen tyytyväisenä. Nämä eivät suoraan liity vain ohjelmistojen kehittämiseen, joten pitäydytään tässä työssä toiminnallisten- ja ei-toiminnallisten vaatimusten jaottelussa.

Vaatimustyypistä riippumatta, vaatimuksille voidaan Holtin, Perryn ja Brownswordin (2012, s. 96-97) mukaan asettaa tiettyjä edellytyksiä, joiden perusteella vaatimusta voidaan pitää ”hyvänä” vaatimuksena. Heidän mukaansa vaatimuksen tulisi olla selvästi tunnistettavissa, sillä vaatimuksilla on tapana muuttua tai kehittyä elinkaarensa aikana. Vaatimusten tulee olla myös selkeitä ja mahdollisimman yksiselitteisiä, jotta väärinkäsityksiltä vältytään. Vaatimus ei myöskään saa olla sidottuna mihinkään tiettyyn ratkaisuun ja jokaisella vaatimuksella tulee olla määritettynä omistaja, joka vastaa vaatimuksen täyttämisestä. Hyvällä vaatimuksella pitää olla myös lähde, eli kuka tai mikä tätä vaatimusta haluaa? Vaatimuksen tulee olla myös vahvistettavissa sekä validoitavissa. Tämä tarkoittaa sitä, että vaatimus on voitava todentaa ja osoittaa että se toimii oikein sekä validoida, eli osoittaa, että se on täytetty. Viimeisenä hyvän vaatimuksen kriteerinä on sen priori-

sointi, eli jokainen vaatimus pitää pystyä asettamaan järjestykseen niiden tärkeyden mukaan, sillä kaikkia vaatimuksia ei aina voida täyttää ja on tärkeää tietää missä järjestyksessä vaatimuksia pyritään täyttämään.

3.2.1 Toiminnalliset vaatimukset

Toiminnalliset vaatimukset ovat konkreettisia toimintoja, joita järjestelmän tulisi tarjota käyttäjilleen tai miten järjestelmän tulisi reagoida eri syötteisiin ja tilanteisiin. Vastaavasti toiminnalliset vaatimukset voivat toki myös määrittää, mitä järjestelmä ei missään tilanteessa saisi tehdä. Ideaalitilanteessa järjestelmän toiminnallisten vaatimusten pitäisi olla kokonaisia ja johdonmukaisia, tarkoittaen sitä, että kaikki vaatimukseen liittyvä informaatio tulisi olla määritettynä, eikä vaatimus saisi olla ristiriidassa minkään muun vaatimuksen kanssa. Kaikenlaiset epätarkkuudet vaatimuksissa voivat johtaa erimielisyyksiin asiakkaan ja kehittäjien välillä ja kehittäjillä onkin usein tapana yksinkertaistaa toteutusta, jos vaatimus ei ole riittävän selvästi määritetty. Tämä taas harvemmin on sitä, mitä asiakas on alun perin halunnut ja vaatimuksen muuttaminen tai tarkentaminen viivästyttää projektia ja kasvattaa sen kuluja (Sommerville, 2016, s. 105).

Holt, Perry ja Brownsword (2012, s. 51-52) mainitsevat, että toiminnallisten vaatimusten kuvaukset hyvin usein sisältävät jonkin verbin, kuten tee, tarjoa tai toimita. Näillä kuvauksilla pyritään yleensä siis kertomaan jokin erillinen toimenpide tai toiminta, joka järjestelmän toivotaan tarjoavan. He myös lisäävät, että toiminnallisten vaatimusten tulisi olla muotoiltu siten, että ne eivät ole riippuvaisia mistään toteutusteknologiasta, vaan ne voitaisiin toteuttaa millä teknologialla hyvänsä ja mihin järjestelmään tahansa.

3.2.2 Ei-toiminnalliset vaatimukset

Robertson J. ja Robertson S. (2006) mainitsevat, että ei-toiminnalliset vaatimukset ovat ominaisuuksia, joita järjestelmällä tulee olla, tai miten hyvin sen tulee suoriutua sille asetetuista tehtävistä. Ei-toiminnalliset vaatimukset tekevät tuotteesta kiinnostavan, käytettävän, nopean, luotettavan tai turvallisen. Ei-toiminnallisilla vaatimuksilla voidaan esimerkiksi siis määritellä vasteaikoja tai laskentojen tarkkuusrajoja. Nämä voivat olla myös vaatimuksia tietynlaisesta ulkoasusta tai vaikka jonkin tietyn lain noudattamisesta.

Sommervillen (2016, s. 107) mukaan ei-toiminnalliset vaatimukset voivat olla monesti kriittisempiä, kuin toiminnalliset vaatimukset. Hyvin usein järjestelmää voidaan käyttää ja mahdolliset ongelmat kiertää, jos toiminnallinen vaatimus ei ole täytetty halutulla tavalla. Ei-toiminnallisen vaatimuksen täyttämättä jättäminen taas voi merkitä sitä, että

koko järjestelmä voi olla käyttökelvoton. Jos esimerkiksi potilaiden terveyteen liittyvä ohjelmisto ei täytä sille asetettua luotettavuuskriteeriä, voisi sen käyttö asettaa potilaat vaaraan.

Yleinen ongelma ei-toiminnallisia vaatimuksia kirjattaessa on, että vaatimus kirjataan liian yleisellä tasolla ja vaatimus asettaa ohjelmalle vain yleisen tason tavoitteita (Sommerville, 2016), s. 109). Tämä voi olla esimerkiksi vikasietoisuuteen liittyen; ”Ohjelmiston pitää palautua yllättävästä katkoksesta nopeasti”. Tämä ei juurikaan anna kehittäjille selkeää kuvaa, miten nopeasti järjestelmän pitäisi palautua, joten suositeltavaa onkin pyrkiä sisällyttämään ei-toiminnallisiin vaatimuksiin jokin mitattavissa oleva tieto. Aiemman vaatimuksen voisi muotoilla vaihtoehtoisesti; ”Ohjelmiston pitää palautua yllättävästä katkoksesta 5 minuutin kuluessa”, jolloin vaatimusta voidaan helposti testata ja näin todentaa vaatimuksen täyttyminen.

Holt, Perry ja Brownsword (2012, s. 101) mainitsevat, että ei-toiminnaliset vaatimukset hyvin usein asettavat rajoitteita muille vaatimuksille. Heidän mukaansa yleensä on tapana vain listata ei-toiminnallisia vaatimuksia, eikä juurikaan miettiä, voisiko kyseinen vaatimus luoda rajoitteita muiden vaatimusten kannalta.

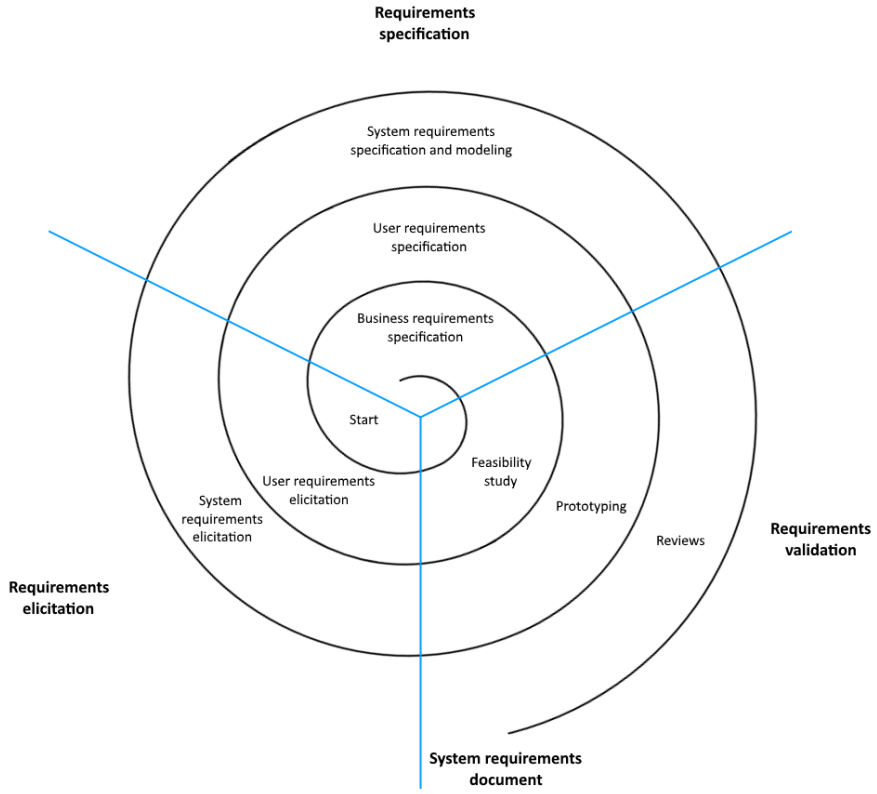
3.3 Perinteinen vaatimusmäärittelyprosessi

Vaatimusmäärittely voidaan jakaa useampaan toisistaan eroavaan aktiviteettiin tai vaiheeseen. Sommerville (2016, s. 55) jakaa vaatimusmäärittelyn kolmeen vaiheeseen; Vaatimusten esillesaanti ja analyysi, määrittelyiden teko sekä vaatimusten validointi. Näiden lisäksi vaatimusmäärittelyyn yleensä sisällytetään myös vaatimustenhallinta, jolloin voidaan todeta vaatimusmäärittelyn koostuvan viidestä eri vaiheesta; Esillesaanti, analyysi, dokumentointi, validointi sekä vaatimustenhallinta (Kuva 3). Tässä työssä käytetään edellä mainittua viisivaiheista vaatimusmäärittelyprosessia.

Kuva 3: *Vaatimusmäärittelyprosessiin kuuluvat vaiheet.*

Sommerville (2016, s. 111-112) toteaa, että vaikka vaatimusmäärittelyprosessi voidaankin jakaa selkeästi eroaviin osa-alueisiin, tehdään vaatimusmäärittelyä hyvin usein itera-

tiivisesti ja näin ollen spiraalimallin mukainen kuvaus olisi osuvampi (Kuva 4). Spiraalimallissa vaatimusmäärittelyn eri aktiviteetit toimivat lomittain ja tehtävät vaatimukset tarkentuvat vaihe vaiheelta.



Kuva 4: *Vaatimusmäärittelyprosessi kuvattuna spiraalina (Sommerville, 2016, s. 112).*

3.3.1 Esillesaanti

Kun ohjelmistoprojekti on saatu käyntiin, ja varsinainen työ ohjelmiston kehityksen suhteen voidaan aloittaa, tulee ohjelmiston elinkaarimallin mukaan vaatimusmäärittelyn vuoro. Ennen kuin mitään varsinaisia määrittelyitä voidaan dokumentoida, pitää näitä koskevat vaatimukset saada ensin selville. Vaatimusmäärittelyprosessin ensimmäinen vaihe onkin esillesaanti (engl. elicitation), jonka tarkoituksena on selvittää kaikki ohjelmistoon liittyvät odotukset, tarpeet ja rajoitteet.

Esillesaantivaiheen yhtenä tarkoituksena on siis selvittää, minkälaista työtä tulevan ohjelmiston eri kohderyhmiin kuuluvat käyttäjät tekevät, minkälaisia odotuksia heillä on uutta järjestelmää kohtaan ja miten uusi järjestelmä voisi tukea heidän työtään parhaiten. Näiden loppukäyttäjien vaatimusten esillesaannissa voi helposti ilmaantua erilaisia ongelmakohtia. Käyttäjät eivät välttämättä tiedä tarkalleen, mitä he haluavat ohjelmistolta,

ainakaan kovin yksityiskohtaisella tasolla. On helppoa antaa yleispiirteisiä toiveita järjestelmän toiminnan suhteen, mutta ohjelmiston tekeminen vaatisi yksityiskohtaisia ja tarkkoja, todistettavissa olevia vaatimuksia. Usein loppukäyttäjät myös ilmaisevat vaatimukset käyttäen omaa termistöään, joka voi hankaloittaa vaatimusten ymmärtämistä. Eri käyttäjät voivat myös ilmaista vaatimuksensa eri tavalla, joka voi johtaa ristiriitoihin tai siihen, että samaa vaatimusta pyydetään hieman eri tavoilla. Näissä tilanteissa onkin tärkeää pystyä tunnistamaan samankaltaisuudet ja ryhmittelemään samaa toiminnallisuutta tavoittelevat vaatimukset. Myös organisaation sisäiset poliittiset tekijät voivat vaikuttaa järjestelmän vaatimuksiin, kun johtoryhmän jäsenet pyrkivät esittämään vaatimuksia, joilla he voisivat vahvistaa omaa asemaansa tai vaikutusvaltaansa organisaatiossa. Yritysympäristön ja talouden dynaamisuus voivat myös aiheuttaa ongelmia vaatimusten esillesaantiprosessissa. Jatkuvasti muuttuva ympäristö voi johtaa siihen, että jo tunnistettujen vaatimusten tärkeys voi vaihdella ja uusia vaatimuksia voi ilmetä tahoilta, joita ei aiemmin oltu vielä kuultu (Sommerville, 2016, s. 112-113).

Vaatimusten esillesaantiin voidaan käyttää monia erilaisia tekniikoita. Tekniikoihin kuuluu interaktiivisia metodeja, joissa ollaan tekemissä eri käyttäjäryhmien kanssa sekä itsenäisesti tehtävää tutkimusta ja vaatimusten selvittämistä. Yleensä työskentely suoraan loppukäyttäjien kanssa on välttämätöntä, sillä onhan tavoitteena kehittää järjestelmä, joka täyttäisi mahdollisimman hyvin näiden käyttäjien tarpeet (Wiegers & Beatty, 2013).

Ensimmäisenä ja varmasti käytetyimpänä metodina on eri kohde- tai sidosryhmiin kuuluvien henkilöiden haastattelu. Haastatteluilla voidaan helposti ja nopeasti muodostaa kokonaiskuva vaatimuksista ja yksityiskohtia voidaan tarkentaa haastattelun edetessä. Yhtenä haastatteluiden etuna on mahdollisuus kysyä vaatimuksista suoraan järjestelmän loppukäyttäjiltä, ja näin saada heidän näkemyksensä mukaan jo vaatimusmäärittelyn alkuvaiheessa. Haastattelut ovat myös tärkeässä osassa ketterissä ohjelmistoprojekteissa, sillä näin asiakas saadaan helpoiten mukaan heti projektin alusta alkaen (Wiegers & Beatty, 2013).

Haastetteluiden lisänä tehokkaina metodeina ovat työpajat (engl. workshop) sekä ydinryhmät (engl. focus group). Työpajat ovat erityisen tehokkaita, sillä niissä projektin sidosryhmien avainhenkilöt voidaan saattaa yhteen ja yhdessä kehitystiimin kanssa eri vaatimuksia voidaan kehittää, tarkentaa ja jalostaa kohti projektin toteutuksen kannalta tärkeitä käyttäjävaatimuksia. Ydinryhmät taas ovat joukko valikoituja henkilöitä, joiden tehtävänä on auttaa vaatimusten esillesaannissa tuomalla kokemustaan sekä ideoitaan osaksi vaatimusten selvittämistä (Wiegers & Beatty, 2013).

Käyttäjien toiveita voi olla hankala saada selville suoraan kysymällä tai käyttäjille voi olla vaikea kuvata, miten heidän tyypillinen työtehtävä hoidetaan. Tämä voi johtua työtehtävän haastavuudesta ja siten sen vaikeasta kuvailtavuudesta tai vastaavasti siitä, että käyttäjät ovat tottuneet käyttämään heille ominaista termistöä, joka ei välttämättä ole selvä vaatimusmäärittelyn tekijälle. Tällaisissa tilanteissa käyttäjän työskentelyn tarkkailu voi osoittautua hyödylliseksi ja vaatimusmäärittelijä voi omalla kokemuksellaan ja tarkentavilla kysymyksillä havaita helposti piiloon jääviä vaatimuksia (Wiegers & Beatty, 2013).

Yhtenä tiedonkeruumenetelmänä voidaan käyttää myös kyselyitä, joissa suurempi joukko käyttäjiä vastaa ennalta määritettyihin kysymyksiin. Kyselyiden tekeminen on myös verrattain kustannuksiltaan alhainen menetelmä, joten jos tavoitteena on saada kerättyä suuremman joukon mielipide johonkin asiaan, voidaan se tehdä kyselyillä kaikkein tehokkaimmin. Kyselyiden tarkoituksena on laajentaa käsitystä käyttäjien tarpeista ja näin auttaa muodostamaan tarkemmin kuvaavia vaatimuksia järjestelmälle. Suurimpana haasteena kyselyiden tekemisessä on sopivien, riittävän selkeiden ja yksinkertaisten kysymysten laatiminen (Wiegers & Beatty, 2013).

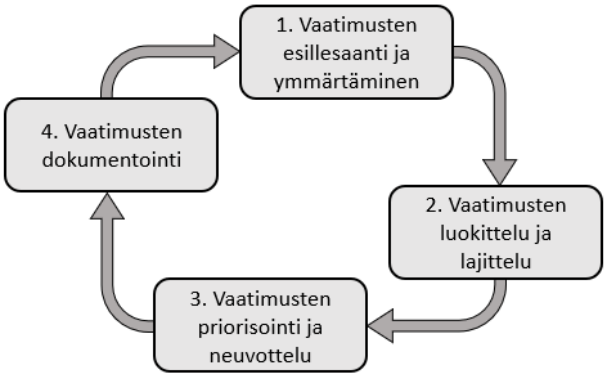
Wiegers ja Beatty (2013) mainitsevat myös staattisista menetelmistä järjestelmien välisten rajapintojen, käyttöliittymien sekä dokumentaatioiden analysoinnin. Kehitettävä järjestelmä voi olla kytköksissä muihin järjestelmiin ja näiden muiden järjestelmien tarjoamien rajapintojen analyysilla voidaan saada esille vaatimuksia tai rajoitteita, jotka vaikuttavat uuteen järjestelmään. Käyttöliittymäanalyysilla tarkoituksena on tehdä analyysia olemassa olevalle järjestelmälle, joka on tarkoitus korvata uudella. Tällä tavoin voidaan muodostaa kattava käsitys, miten nykyinen järjestelmä toimii ja mitä puutteita sen käytössä voisi olla. Dokumentaatioanalyysin avulla voidaan selvittää vanhan järjestelmän määrittelyistä, mitä toiminnallisuuksia on ehdottomasti säilytettävä ja mitkä ovat mahdollisesti jo poistuneet käytöstä. Vanhan järjestelmän dokumentointi voi helpottaa alun vaatimusten esillesaantia, kun kaikkea ei tarvitse keksiä tyhjästä, vaan voidaan turvautua osin aiempaan dokumentaatioon.

3.3.2 Analyysi

Esillesaantivaiheen jälkeen saadut vaatimukset tulee analysoida ja prosessoida. Tämä analysointivaihe mielletään usein osaksi esillesaantivaihetta ja tästä onkin kahdenlaista koulukuntaa. Osa käsittelee vaiheet yhtenä ja osa taas jakaa vaiheet erilleen. Vaatimusmäärittelyprosessin selkeyttämiseksi tässä työssä analysointivaihe käsitellään erillään.

Analyysivaiheen tarkoituksena on luokitella, käsitellä, tarkentaa ja priorisoida aiemmin esille saatuja vaatimuksia. Näillä toimilla vaatimuksia voidaan ymmärtää paremmin ja tunnistaa mahdolliset ristiriidat näiden välillä. Kun projektissa on mukana useampia sidosryhmiä, jossakin vaiheessa vaatimukset todennäköisesti ajautuvat ristiriitaan keskenään. Näiden ristiriitojen ilmetessä, kyseisten vaatimusten takana olevat sidosryhmät tulee ottaa mukaan neuvotteluun ja löytää ratkaisu, joka miellyttää kaikkia osapuolia. Neuvottelut ovatkin yksi analyysivaiheen keskeisimpiä välineitä (Ahmad, 2008, s. 683).

Myös Sommerville (2016, s. 113-114) mainitsee analyysi- ja esillesaantivaiheiden lomittaisuudesta sekä syklisyydestä. Vaatimusten määrittely alkaa niiden esillesaannista ja päättyy vaatimusten dokumentointiin, näiden vaiheiden välissä vaatimuksia analysoidaan ja tarkennetaan (Kuva 5). Vaatimusten analysointia helpottaa, jos sidosryhmiltä saadut tiedot ryhmitellään. Yhtenä ryhmittely keinona voidaan käyttää sidosryhmien tuomaa jaottelua, tai järjestelmäarkkitehtuurin tuomaa alijärjestelmien jaottelua ja jakaa vaatimukset näihin ryhmiin.



Kuva 5: *Analyysivaiheen vaatimusten luokittelu, priorisointi ja näistä neuvottelu osana iteratiivista vaatimusmäärittelyä (Sommerville, 2016, s. 113).*

Analyysivaiheen päätöksenteossa avainasemassa ovat siis priorisointi ja neuvottelut. Priorisoinnilla järjestelmän vaatimukset pyritään järjestämään niiden tärkeyden mukaan ja tunnistamaan kaikkein arvokkaimmat vaatimukset. Usein vaatimuksia tulee niin paljon, että projektin aikataulun ja budjetin puitteissa kaikkia ei ehditä mitenkään toteuttamaan, joten vaatimusten priorisointi ja tärkeimpien toimintojen löytäminen on lähes itsestäänselvyys projektin onnistumisen kannalta. Tärkein työkalu vaatimusten priorisointiin on projektin sidosryhmien väliset neuvottelut. Juuri näiden neuvotteluiden avulla voidaan

löytää yhteisymmärrys projektin kannalta tärkeimmistä vaatimuksista sekä löytää ratkaisu mahdollisten erimielisyyksien tai vaatimuksissa ilmenneiden ristiriitojen kannalta (Ahmad, 2008, s. 684).

3.3.3 Dokumentointi

Kun järjestelmälle asetetut vaatimukset on saatu selvitettyä sekä priorisoitua sidosryhmien kanssa käytyjen neuvotteluiden tuloksena, pitää ne kirjata ylös tulevaa kehitystyötä varten. Tämän dokumentointivaiheen tarkoituksena on luoda järjestelmän kehitystä tukevat varsinaiset määrittelyt. Määrittelyiden muoto, kieliasu ja niiden sisältämän informaation määrä riippuu paljon kehitysprojektissa käytettävistä menetelmistä ja käytänteistä. Wiegersin ja Beattyn (2013) mukaan oleellista on kuitenkin se, että määrittelyt sisältävät juuri sopivasti tietoa kehittäjien ja testaajien tarpeisiin. Liika informaatio saattaa peittää määrittelyn perimmäisen tarkoituksen ja taas liian vähän informaatiota antava määritys voi herättää enemmän kysymyksiä, kuin se antaa vastauksia.

Sopivan informaation määrän lisäksi määrityksille voidaan asettaa joitakin ominaisuuksia, jotka tekevät niistä hyviä. Sommervillen (2016, s. 120) mukaan ideaalitapauksessa määrittelyiden tulisi olla selkeitä, yksiselitteisiä, helppoja ymmärtää, kattavia sekä johdonmukaisia. Näiden lisäksi Wiegers ja Beatty (2013) mainitsevat, että hyvän vaatimuksen tulee olla tarkka ja oikein, mahdollinen, välttämätön, priorisoitu sekä todennettavissa. Wiegers ja Beatty (2013) jatkavat, että toisiinsa liittyvien määritysten, esimerkiksi samaan julkaisuun liittyvien, tulisi olla aiemmin mainittujen lisäksi muokattavissa sekä jäljitettävissä. Nämä viimeksi mainitut tulevat osittain muutoksen- ja vaatimustenhallinnan kautta.

Määrittelyiden varsinaiseen kirjoittamiseen voidaan käyttää useita erilaisia menetelmiä. Näistä ehdottomasti yleisin on luonnollisten kielten käyttö. Tämän lisäksi määritystä tukemaan voidaan käyttää erilaisia taulukoita ja kaavioita, joiden avulla määritystä voidaan selkeyttää sekä tarkentaa. Yleensä myös vaatimuksen tyyppi määrittelee missä muodossa kyseinen määritys voidaan tehdä. Käyttäjävaatimukset tulisi aina kirjoittaa luonnollisilla kielillä, sillä näin ne on helpompi ymmärtää projektin sidosryhmien kesken. Järjestelmävaatimukset voidaan myös kirjoittaa käyttäen luonnollisia kieliä, mutta joissakin tapauksissa voi olla hyödyllistä ottaa mukaan formaaleja tai matemaattisia metodeja tarkentamaan määritystä (Sommerville, 2016, s. 120). Näitä eri määrittelymenetelmiä käydään tarkemmin läpi luvussa 4.

Määrittelyt voidaan kirjoittaa käyttäen vapaamuotoista dokumenttipohjaa tai vaihtoehtoisesti jotakin sähköistä alustaa, jonne vaatimukset ja niihin liittyvät määrittelymateriaalit

tallennetaan. Yhtenä vaihtoehtona on myös käyttää ohjelmiston vaatimusmäärittely-dokumenttia, eli SRS-dokumenttia (engl. software requirements specification). Tämä on hieman muodollisempi dokumentointitapa koko ohjelmistoa tai järjestelmää koskeville vaatimuksille ja määrittelyille. Tarkoituksena on koota kaikki tarpeellinen yhteen dokumenttiin, jota kaikki projektin sidosryhmät voivat lukea ja käyttää. Sommervillen (2016, s. 127) mukaan tämä tarkoittaa sitä, että dokumentin tekemisessä joudutaan tekemään kompromisseja. Dokumentin tarvitsee selittää vaatimukset työn tilaajalle ymmärrettävässä muodossa, riittävän yksityiskohtaisella tasolla kehittäjille ja testaajille sekä sisältää riittävästi informaatiota järjestelmän evoluutiosta suunnittelijoita varten. SRS-dokumentin rakenne on yleensä muodollinen, sisältäen omat osionsa johdannolle, sanastolle sekä liitteille. Dokumentin pituus riippuu pitkälle siitä, miten tarkasti määrittelyt halutaan kuvata. Laajemmissa projekteissa, kun mukana on useampia toimittajia, voi olla tarpeellista kuvata järjestelmän osiin vaikuttavat vaatimukset tarkemmin. Sommerville (206, s. 127) kertoo myös, että ketteryyden näkökulmasta ajateltuna kyseistä dokumenttia voitaisiin pitää turhana. Monissa tapauksissa dokumentti ehtii vanheta jo ennen kuin se on saatu valmiiksi, sillä ketterissä menetelmissä pyritään kannustamaan muutoksiin, joihin muodolliset pitkät dokumentit eivät usein taivu. Tästä huolimatta voi olla suositeltavaa kirjata ainakin projektiin kuuluvien järjestelmien pääpiirteiset vaatimukset ja rajoitteet yhteen dokumenttiin, johon voidaan viitata pitkin projektia.

3.3.4 Validointi

Vaatimusten validoinnin tai tarkistamisen tarkoituksena on varmistua siitä, että tehdyt määrittelyt vastaavat edelleen järjestelmälle asetettuja tavoitteita ja odotuksia. Tilanteet voivat muuttua projektin edetessä ja niin voivat myös järjestelmälle asetetut toiveet. Tämän vuoksi on hyvä tarkistaa, että tehdyt määrittelyt edelleen täyttävät jonkin järjestelmälle asetetun tarpeen. Validointi voidaan ymmärtää myös siten, että tarkistetaan tehdyn määrittelyn dokumentointiin liittyviä hyvän määrityksen laatukriteereitä, joita esiteltiin dokumentointia käsittelevässä luvussa. Tämä on ehkä enemmän vain vaatimusten laadun tarkastusta ja varsinainen validointi tarkoittaa juuri vaatimusten oikeellisuuden ja tarkoituksenmukaisuuden tarkistamista (Wiegers & Beatty, 2013).

Vaatimusten validoinnissa vaatimuksille pyritään tekemään useampia erilaisia tarkistuksia. Näihin tarkistuksiin kuuluu Sommervillen (2016, s. 129) mukaan ainakin viisi erilaista tarkistusta. Vaatimuksen pätevyyttä (engl. validity) tarkistetaan vertaamalla sitä järjestelmälle asetettuihin käyttäjätarpeisiin. Aiemmin mainitusti muuttuva tilanne on voinut tehdä vaatimuksesta tarpeettoman. Toinen tarkistus liittyy johdonmukaisuuteen (engl. consistency), eli vaatimus tai määrittely ei saa olla ristiriidassa toisen kanssa, eikä samasta

asiasta kuuluisi olla useampaa määritystä. Vaatimuksen kokonaisuutta (engl. completeness) voidaan myös arvioida, eli onko määrittelyssä mainittu kaikki tarvittavat toiminnot, ominaisuudet sekä rajoitteet, jotka järjestelmän käyttäjän toimintaan vaikuttavat. Neljäntenä tarkistuksena Sommerville mainitsee vaatimuksen realistisuuden (engl. realism) arvioinnin. Tällä tarkoitetaan sitä, onko vaatimus ylipäätään mahdollista toteuttaa käyttäen olemassa olevia tekniikoita ja tietotaitoa sekä onko toteutustapa mahdollinen projektin budjetin ja aikataulun puitteissa. Viimeinen tarkistus liittyy vaatimuksen todentamiseen (engl. verifiability), jolla tarkoitetaan vaatimuksen testattavuutta. Vaatimus tulisi aina kirjoittaa siten, että sen pohjalta voidaan kirjoittaa tarvittavat testit ja siten todistaa järjestelmän täyttävän kyseisen vaatimuksen.

Tapoja vaatimusten arviointiin on monia. Määrittelyn tekijä voi toki itsekin arvioida vaatimusta aiemmin mainittujen kriteerien pohjalta, mutta yleensä on suositeltavaa käyttää apuna vertaisarviointia, jos se suinkin on mahdollista. Wiegers ja Beatty (2013) mainitsevat kolme vertaisarviointitapaa. Yksittäisarviointi, jossa yksi kollega käy tarkistamassa tehdyn määrittelyn. Tämä yleensä tehdään määrittelytyötä tekevän työpisteellä yhdessä määrittelijän kanssa. Toinen tapa on kutsua useampi kollega käymään läpi määrittelyitä samanaikaisesti ja kerätä näin palautetta. Kolmannessa määrittelyiden tekijä itse selittää tehdyt määrittelyt suuremmalle joukolle ja kerää tätä kautta palautetta. Sommerville (2016, s. 130) mainitsee näiden erilaisten katsausten lisäksi prototypoinnin sekä testitapausten koostamisen. Prototyypin avulla voidaan helposti ja nopeasti testata, täyttääkö ehdotettu ratkaisu käyttäjien tarpeet ja odotukset. Saadun palautteen avulla vaatimuksia voidaan tarkentaa ja jalostaa sekä prototyyppiä voidaan käyttää mahdollisesti tukemaan kehitystä. Testitapausten teko toimii myös hyvänä validointitapana, sillä jos testejä on vaikea tehdä, on todennäköisesti myös vaatimuksessa jotain vikaa ja sitä tulisi tarkentaa.

Wiegers ja Beatty (2013) kertovat, että vaatimusten validointiin käytettävät katselmukset voivat olla luonteeltaan formaaleja tai informaaleja. Eroina näissä on se, että formaaleista, muodollisista katselmuksista tavoitteena on tuottaa raportointia ja niissä käytetään tarkkaan määritettyä prosessia. Informaaleissa katselmuksissa ei suoraan vaadita dokumentointia tai keskustelua, jonka vuoksi eri tarkastelijat voivat tulkita saman vaatimuksen eri tavalla ajatellen, että se on silti oikein. Tämä voi aiheuttaa huomaamattomia virheitä määrittelyissä, sillä yhtenä ehtona on nimenomaan määrittelyiden yksiselitteisyys.

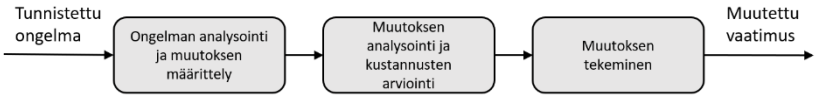
3.3.5 Vaatimustenhallinta

Ohjelmistoprojektiin liittyviä vaatimuksia kertyy projektin aikana valtavat määrät ja väistämättä ajaudutaan jossakin vaiheessa tilanteeseen, jossa vaatimuksiin on tehtävä muutoksia. Ilman minkäänlaista vaatimustenhallintaa vaatimukset voivat sekoittua toisiinsa,

tietoa voi hukkua tai vaatimuksiin tehdyt muutokset eivät välttämättä edes välity kaikille osapuolille. Vaatimustenhallinnalla on tarkoitus ylläpitää vaatimusten paikkansapitävyyttä ja mahdollistaa oikean tiedon välittyminen projektin etenemisestä vastaaville osapuolille. Tästä syystä onkin oleellista ottaa vaatimustenhallinta mukaan osaksi projektia heti alusta alkaen.

Wiegers ja Beatty (2013) mainitsevat vaatimushallinnan koostuvan neljästä osa-alueesta; versionhallinnasta, muutoksenhallinnasta, vaatimusten tilan seurannasta sekä vaatimusten jäljentämisestä. Versionhallinnan tarkoituksena on mahdollistaa vaatimuksen historian tarkastelu sekä varmistaa, että vaatimuksesta käytetään aina sen viimeisintä, ajan tasalla olevaa versiota. Muutostenhallinnalla tarkoitetaan prosessia, joka sisältää vaatimusten muutosten ehdottamisen, muutoksen vaikutusten analysoinnin, vaatimuksen tai määrittelyn varsinaisen muutoksen sekä vaatimusten muutosten mittaamisen. Vaatimusten tilan seurannalla taas on tarkoitus seurata vaatimusten tilannetta kehityksen aikana. Tähän kuuluu mahdollisten eri tilojen määrittäminen sekä jokaisen vaatimuksen tilan kirjaaminen ja seuranta projektin kuluessa. Vaatimusten tilan seurannan lisäksi tarvitaan usein vaatimusten jäljittämistä, jonka avulla on tarkoitus seurata vaatimukseen liittyvien muutosten historiaa aina vaatimuksen alkuperään saakka.

Muutoksenhallinta yleensä juontaa johonkin havaittuun ongelmaan vaatimuksessa, joka pitää korjata. Sommervillen (2016, s. 133-134) mukaan vaatimusten muutoksenhallinta voidaan jakaa kolmeen vaiheeseen (Kuva 6). Alussa ongelma analysoidaan ja muutoksesta tehdään määrittely ja ehdotus. Tarkoitus on varmistaa, että ehdotettu muutos on edelleen pätevä ja sopiva järjestelmän tarpeisiin. Seuraavaksi muutos analysoidaan tarkemmin ja pyritään selvittämään mitä vaikutuksia sillä voi olla muihin vaatimuksiin sekä mitä lisäkustannuksia tehtävään muutokseen liittyy. Kun muutosanalyysi on tehty, voidaan tehdä päätös varsinaisen vaatimuksen muuttamisesta ja siirtyä muutoksen tekemiseen. Vaatimukseen ja määrittelyyn liittyvää dokumentointia muutetaan vastaamaan uutta vaatimusta, jonka vuoksi onkin syytä ottaa huomioon mahdollisten muutosten tuleminen jo dokumentin rakennetta päätettäessä.



Kuva 6: *Muutostenhallinnan prosessi havaitusta ongelmasta uudistettuun vaatimukseen (Sommerville, 2016, s. 133)*

Tämän muutoksenhallintaprosessin oleellisena osana oleva muutosten vaikutuksen analysointi on lähes välttämätöntä, riippumatta siitä, minkä kokoisesta muutoksesta on kyse. Sen tarkoituksena on tarjota tietoa tehtävän muutoksen seurauksista ja näin ollen auttaa tekemään päätöksiä muutoksen hyväksynnän osalta. Analyysin tehtävänä on tunnistaa mitä järjestelmän komponentteja voidaan joutua lisäämään, muokkaamaan tai poistamaan sekä minkälainen työarvio näihin liittyy. Analyysi alkaa vaatimuksen muutoksen vaikutusten tunnistamisella, mihin järjestelmän osiin ja muihin vaatimuksiin se voi vaikuttaa ja minkälaisia muutoksia järjestelmän arkkitehtuuriin ja koodiin tai testeihin pitää tehdä. Myös kaikki muutokseen liittyvä materiaali tulee tunnistaa, koskien kaikkia vaatimus dokumentteja, tiedostoja tai malleja, joita voidaan joutua muuttamaan. Viimeisenä toimena analyysissa pitää arvioida kyseiseen muutokseen liittyvän työn määrää (Wiegers & Beatty, 2013).

Projektin kulun kannalta on myös tärkeää seurata vaatimusten tilaa. Tällä tilan seurannalla tarkoitetaan sitä, kun vaatimusta seurataan aina sen esittämisestä siihen saakka, kunnes se on valmis ja toteutettu osaksi järjestelmää. Vaatimusten tilan seuranta on siis merkittävänä osana myös projektin seurantaan liittyviä toimia. Erilaisia vaatimusta kuvaavia tiloja Wiegers ja Beatty (2013) mainitsevat olevan esimerkiksi ehdotettu (engl. proposed), meneillään (engl. in progress), luonnosteltu (engl. drafted), hyväksytty (engl. approved), toteutettu (engl. implemented), todennettu (engl. verified), lykätty (engl. deferred), poistettu (engl. deleted) sekä hylätty (engl. rejected). Näitä tiloja voidaan projektin tarpeeseen keksiä uusiakin tai käyttää vain muutamia, tarkoituksena on kuitenkin tarjota riittävän monipuoliset vaihdot kattavaan vaatimusten seurantaan. Nimettyjen kategorioiden käyttö voi olla myös huomattavasti kuvaavampaa, kuin pelkkien prosentuaalisten valmiusasteiden käyttö.

Vaatimustenhallintaan voi liittyä monenlaisia ongelmia, erityisesti vaatimustenhallintaan liittyvään arviointiin ja suunnitteluun, vaatimusten tilan seurantaan sekä muutostenhallintaan. Ongelmia syntyy esimerkiksi siitä, että vaatimustenhallinnasta on hyvin harvoin kokemusta yksilötasolla tai organisaatiolta puuttuu valmis, riittävän selkeä vaatimustenhallinnanmalli, jota voitaisiin seurata eri projekteissa. Tämä johtaa siihen, että esimerkiksi vaatimustenhallintaankin olennaisesti liittyviä työaika-arvioita on erittäin hankala tehdä, kun aiheeseen liittyvää kokemusta ei ole kertynyt. Myös käyttäjä- ja järjestelmävaatimusten erottaminen toisistaan on ajoittain hankalaa, joka voi johtaa siihen, että vaatimuksista ei osata tehdä toteutustavasta riippumattomia, vaan pyritään keksimään vaatimukseen sopiva ratkaisu jo heti määrittelyvaiheessa. Ongelmia voi liittyä myös organisaation tyyppiin ja harjoitettavan liiketoiminnan alaan ja siihen, miten tämä ala ohjailee

vaatimustenhallintaprosessia. Ylipäätään vaatimuksia käsitellään eri lailla, riippuen ollaanko tekemisissä tuotteen tai ohjelmiston valmistamisessa vai perustuuko liiketoiminta jonkin kolmannen osapuolen tarjoamaan valmiiseen ohjelmistoon. Ongelmia voi aiheutua myös vaatimusten tilan seurannassa, sillä esimerkiksi vaatimusten esillesaannin aikana niiden tilaa on vaikea seurata ja hetkeä, jolloin yhden aihealueen vaatimukset on kaikki saatu selville, on vaikea määrittää. Viimeisenä ongelmakohtana ovat muutosten tuomat haasteet, jolloin mikä tahansa tehty muutos todennäköisesti vaikuttaa myös muihin vaatimuksiin ja näiden vaikutuksia ja kustannuksia voi olla hankala arvioida (Dick & Hull & Jackson, 2017, s. 154-156).

3.4 Standardit

Monilla tekniikan ja teollisuuden aloilla on tapana käyttää erilaisia standardeja kuvaamaan tehtävän työn käytänteitä sekä varmistamaan työn oikeellisuutta. Standardeilla on myös tapana toimia ohjaavana tekijänä tuotteen valmistuksessa, jos valmistettavan tuotteen tarvitsee täyttää jonkin markkina-alueen vaatimukset. Ohjelmisto- ja tietotekniikassa standardeilla kuvataan nimenomaan tapoja toimia ja niiden tarkoituksena on pyrkiä yhtenäistämään teknisiä ratkaisuja ja alan käytänteitä.

Ohjelmistojen vaatimusmäärittelyä ohjaamaan on vuosien saatossa kehitetty useampia standardeja. Yhtenä suurimpana standardeja ja suosituksia määrittelevänä tahona on Institute of Electrical and Electronics Engineers eli IEEE, joka on voittoa tavoittelematon järjestö. Järjestön tarkoituksena on edistää teknologian kehitystä ihmiskunnan hyväksi (IEEE, 2019a). IEEE on julkaissut useampia standardeja liittyen vaatimusmäärittelyyn, joista merkittävimpiä ovat ainakin vuonna 1984 ensimmäisen kerran julkaistu määrittelydokumenttia koskeva IEEE 830 standardi (IEEE, 2019b) sekä myöhemmin tämän korvannut, myös muita standardeja yhdistänyt IEEE 29148 (IEEE, 2019c). Kumpaakin standardia on ajan saatossa muokattu ja näistä onkin tarjolla useampia versioita. Määrittelydokumenttia koskevasta IEEE 830 standardista ehdittiin julkaista päivitetyt versiot vuosina 1993 ja 1998, jonka jälkeen IEEE 29148 korvasi standardin vuonna 2011. Tässä työssä tarkoitus on tutustua lyhyesti tähän uusimpaan IEEE 29148 standardiin ja sen pääpiirteisiin.

Uusimmasta nykyäänkin voimassa olevasta vaatimusmäärittelyä koskevasta IEEE 29148 standardista on julkaistu kaksi versiota. Ensimmäinen vuonna 2011 ja tästä päivitetty IEEE 29148-2018 versio vuonna 2018. Standardi määrittää hyvän vaatimuksen rakenteen, kertoo vaatimusten ominaisuuksista ja piirteistä ja tarjoaa tietoa vaatimusmäärittelyn iteratiivisesta käytöstä pitkin ohjelmiston elinkaarta (IEEE, 2019c). Tämä uusi

standardi korvasi yhdistämällä saman standardin alle aiemmin käytetyt SRS-dokumenttia koskevan standardin IEEE 830-1998 (IEEE, 2019b), järjestelmien vaatimusmäärittelyä koskevan ohjeistuksen IEEE 1233-1998 (IEEE, 2019d) sekä järjestelmän ominaisuuksista kertovan ConOps (engl. concept of operations) -dokumentin rakenteen määrittäneen standardin IEEE 1362-1998 (IEEE, 2019e).

Tämä uusin nykyisin käytössä oleva vaatimusmäärittelyn standardi IEEE 29148 määrittelee joukon prosesseja, joiden avulla järjestelmien ja ohjelmistojen vaatimuksia voidaan tuottaa. Näihin prosesseihin kuuluu yleisesti vaatimusmäärittelyyn liittyvät käytännöt, liiketoimintaan ja tavoitteisiin liittyvän analysoinnin prosessi, sidosryhmien tarpeiden ja vaatimusten määrittelyn prosessi, järjestelmävaatimusten määrittelyprosessi sekä vaatimusten hallintaan liittyvä prosessi. Standardiin kuuluu myös ohjeita ja suosituksia näiden prosessien käyttöön. Prosessien lisäksi standardissa on kuvattu tarkasti vaatimusmäärittelyyn liittyvien määrittelydokumenttien vaadittava sisältö ja rakenne. Dokumentit, jotka standardissa on kuvattu ovat liiketoiminnan vaatimuksia kuvaava BRS (engl. business requirements specification), sidosryhmiltä tulevia vaatimuksia määrittelevä StRS (engl. stakeholder requirements specification), järjestelmän vaatimuksista kertova SyRS (engl. system requirements specification) sekä aiemmin mainittu, yleisempi ohjelmiston vaatimuksia määrittelevä SRS-dokumentti (IEEE, 2018).

Standardin käytön hyödyt tulevat todennäköisesti enemmän esille formaaleja käytäntöjä seuraavissa projekteissa sekä turvallisuuskriittisiä järjestelmiä kehitettäessä. Tarkan, tiettyä rakennetta seuraavan dokumentoinnin käyttö voi tuoda joissakin tilanteissa selkeyttä vaatimuksiin, mutta käytännön syistä suurta määrää tarkkaa dokumentaatiota voidaan käyttää lähinnä pääosin staattisen ja muuttumattoman tilanteen pohjalta rakennettavissa ohjelmistoissa sekä perinteistä suunnitteluvetoista vesiputousmallia käytettäessä.

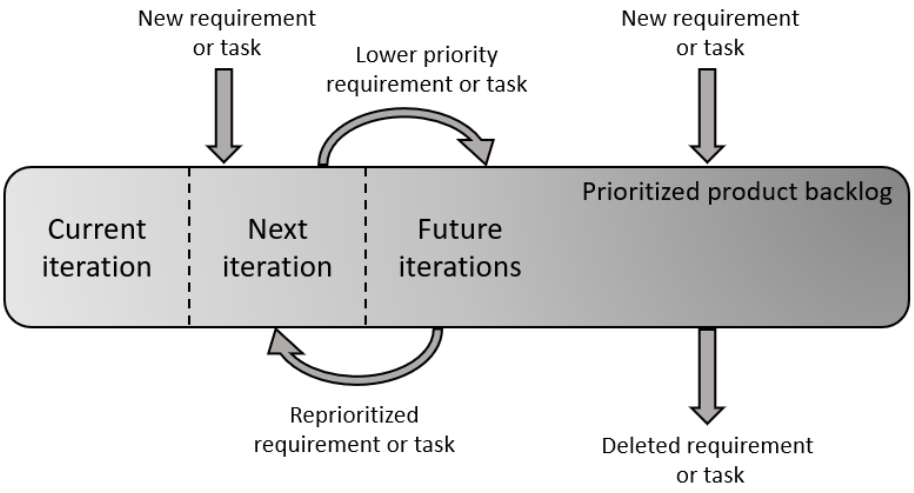
3.5 Vaatimusmäärittely ketterässä ohjelmistokehityksessä

Vaatimusmäärittely ketterässä ohjelmistokehityksessä eroaa hieman perinteisissä suunnitteluvetoisissa elinkaarimalleissa olevasta vaatimusmäärittelystä. Wiegers ja Beatty (2013) mainitsevat, että vaikka puhutaankin vaatimusmäärittelystä ketterissä menetelmissä, niin silti termi ”ketterät vaatimukset” olisi ilmaisuna väärin. Ketterät vaatimukset termi viittaisi siihen, että ketterissä menetelmissä tehtävän vaatimusmäärittelyn tuloksena saatavat vaatimukset jotenkin eroaisivat perinteisemmillä menetelmillä saaduista vaatimuksista. Kaikki saadut vaatimukset ovat kuitenkin samanlaisia, sillä kehittäjien tarvitsee tietää täsmälleen samat tiedot, riippumatta käytetystä menetelmästä.

Ketterä vaatimusmäärittely eroaa perinteisesti tehtävästä vaatimusmäärittelystä painottamalla asiakkaan mukana oloa, dokumentoinnin määrän pienentämisellä sekä iteratiivisella tavalla toimia. Cao ja Ramesh (2008) ovat tunnistaneet tutkimuksessaan seitsemän tyypillistä piirrettä vaatimusmäärittelylle ketterässä ohjelmistokehityksessä. Näitä piirteitä ovat kasvotusten kommunikointi kirjoitettujen määrittelyiden sijaan, iteratiivinen määrittely, määritysten priorisointi, muuttuvien vaatimusten hallinta jatkuvalla suunnittelulla, prototypointi, TDD (engl. test driven development) sekä erilaiset katselmukset ja hyväksymistestaus. Näistä menetelmistä kaikki, paitsi prototypointi ja TDD olivat käytössä kaikissa tutkimukseen osallistuneissa organisaatioissa.

Ketterien menetelmien iteratiivisen luonteen vuoksi asiakkaan kanssa kommunikoidaan koko projektin ajan. Tämä tarkoittaa vaatimusmäärittelyn kannalta sitä, että projektiin voidaan lisätä kokonaan uusia vaatimuksia tai muuttaa jotain vanhoista, missä vaiheessa tahansa. Ketterissä menetelmissä puhutaan usein käyttäjätarinoista (engl. user stories), jotka ajavat vaatimusten virkaa. Nämä käyttäjätarinat kuvaavat mitä käyttäjän pitäisi pystyä tekemään, joista sitten johdetaan varsinaiseen kehitystyöhön tarvittavat tehtäväkuvaukset. Käyttäjien mukanaolo tuo myös sen mahdollisuuden, että projektin iteraatioissa saatu toiminnallisuus voidaan testata ja validoida heti (Wiegers & Beatty, 2013).

Ketterille menetelmille tyypillinen muutoksiin kannustaminen luonnollisesti vaikuttaa suoraan myös vaatimusmäärittelyyn ja vaatimustenhallintaan. Iteratiivisissa ketterissä menetelmissä (Kuva 7) vaatimuksia hallitaan käyttämällä tehtävälistoja (engl. backlog), joihin projektin vaatimukset kerätään. Tässä listassa olevia vaatimuksia voidaan muokata, lisätä ja poistaa missä vaiheessa projektia tahansa ja vaatimusten priorisointia voidaan muokata muuttuvien tarpeiden seurauksena. Tämä priorisoitu lista siis elää koko ajan pitkin projektia ja kuhunkin iteraatioon sisältyvät tehtävät valitaan priorisoinnin mukaan. Uusia vaatimuksia on mahdollista ottaa heti mukaan käynnissä olevaan iteraatioon, mutta suositeltavaa olisi pitäytyä jo päätetyissä tehtävissä ja ottaa uusi vaatimus mukaan vasta seuraavassa iteraatiossa. Jos halutaan tehostaa uusiin vaatimuksiin reagointia ja mahdollistaa näiden ottaminen mukaan iteraatioihin mahdollisimman nopeasti, voidaan pyrkiä lyhentämään iteraatioiden pituutta. Tällä tavoin kehittäjille annetaan rauha keskittyä toteuttamaan iteraatioon valitut tehtävät ja silti uusien vaatimusten lisäämistä toteutettavaksi saadaan nopeutettua (Wiegers & Beatty, 2013).



Kuva 7: *Vaatimustenhallinta iteratiivisissa ketterissä menetelmissä (Wiegers & Beatty, 2013).*

Iteratiivisissa menetelmissä vaatimusmäärittelyä tehdään osana jokaista iteraatiota. Yleensä vain korkeimman tason vaatimukset selvitetään projektin alussa, joita sitten tarkennetaan osana iteraatioita. Tällä tavoin vaatimukset saadaan pidettyä ajankohtaisina ja ne vastaavat tilaajan sen hetkisiä toiveita ja tarpeita. Cao ja Ramesh (2008) mainitsevat tutkimuksessaan iteratiivisuuden hyödyiksi tyydyttävämmän asiakassuhteen kehittymisen sekä asiakkaan helpon tavoitettavuuden johdosta saatavat selkeämmät vaatimukset. Iteratiivisuuden haasteiksi he mainitsevat kustannusten ja aikataulun arvioinnin, minimaalisen dokumentoinnin määrän sekä ei-toiminnallisten vaatimusten huomioimatta jättämisen projektin alkuvaiheessa. Yleensä projektin alkuvaiheessa keskitytään liikaa keskeisimpien toimintojen toteuttamiseen, jolloin järjestelmän toimintaa koskevat vaatimukset voivat jäädä taka-alalle.

Ketteriä menetelmiä käyttävissä projekteissa käytetään usein erilaisia työkaluja projektin ja vaatimusten seurantaan. Wiegers ja Beatty (2013) mainitsevat projektin seurantaan ja hallintaan käytettävistä työkaluista erilaiset käyttäjätarinoiden tilan seurantaan liittyvät työkalut sekä kunkin iteraation tai projektin kulun seurantaan käytettävät edistymiskäyrät (engl. burndown chart). Näiden käyrien avulla voidaan kuvata sekä projektin arvioitua työmäärää että kunkin iteraation toteutunutta työmäärää. Näitä työaika-arvioita kuvataan usein käyttäjätarinoihin sidotuilla pisteillä. Suuremmat tehtävät ovat useamman pisteen arvoisia ja näin kestävät pidempään. Edistymiskäyrän avulla voidaan seurata projektin toteutunutta kulkua verrattuna arvioituun edistymiskäyrään ja näin tehdä päätöksiä iteraatioihin tulevasta työmäärästä. Esimerkiksi jos näyttää siltä, että suunniteltu työmäärä on kussakin iteraatiossa lian suuri, voisi olla järkevää pienentää iteraatioihin mukaan otettavaa työmäärää, jolloin projektin kustannusten ja keston arviointi voidaan muuttaa

realistisemmaksi. Vaatimuksiin liittyvään muutostenhallintaan käytetään usein jotakin tikettien keruujärjestelmää, johon vaatimukset kootaan ja säilötään. Jotkin työkalut tarjoavat valmiin muutostenhallintaprosessin, jossa työkalun avulla voidaan luoda ja hallita vaatimukseen liitettäviä muutospyyntöjä (engl. change request). Lisäksi työkaluun sisältyvien raportointimahdollisuuksien avulla voidaan seurata tapahtuvien muutosten määrää ja vaikutuksia kehitykseen.

4. MÄÄRITTELYMENETELMÄT OHJELMISTOKEHITYKSESSÄ

Vaatimusten dokumentointia varsinaisiksi määrittelyiksi voidaan tehdä monella tapaa. Yleisesti käytetään luonnollisia kieliä, jolloin määrittelyt kuvataan tekstimuotoisina selosteina käyttäen apuna kuvia ja taulukoita. Jos luonnollisten kielten käyttö todetaan riittämättömäksi, tai määritysten oikeellisuus on äärimmäisen tärkeää, voidaan käyttää formaaleja määrittelymenetelmiä. Näiden avulla vaatimukset ja määrittelyt voidaan todentaa jo ennen varsinaista kehitystyötä ja mahdolliset virheet havaita aikaisessa vaiheessa. Luonnollisten kielten ja formaalien määrittelymenetelmien välimaastoon jäävät puoliformaalit menetelmät, kuten UML-kaaviot, joita voidaan käyttää esimerkiksi luonnollisten kielten lisänä. Näin voidaan varmistua määritysten oikeellisuudesta sekä tehdä määrittelyistä helpommin ymmärrettäviä ja luettavampia

Tässä luvussa on tarkoitus tutustua eri määrittelymenetelmiin ohjelmistokehityksessä. Formaaleista menetelmistä tutustutaan perinteikkään tietotekniikkayrityksen IBM:n (engl. International Business Machines Corporation) kehittämiin Z-notaatioon ja VDM:ään (engl. Vienna Development Method). Näiden lisäksi tutustutaan puoliformaaleista menetelmistä UML-kielen sekä informaaleista menetelmistä luonnollisten kielten käyttöön osana määrittelytyötä.

4.1 Formaalit määrittelymenetelmät

Formaaleiksi menetelmiksi kutsutaan järjestelmällisiä matematiikkaan perustuvia kuvaustapoja, jotka muodostuvat abstraktin algebran, diskreetin matematiikan sekä logiikan ympärille. Menetelmissä matematiikan ja loogisen päättelyn avulla kuvataan ja määritellään jokin ominaisuus, joka järjestelmän tulisi toteuttaa. Tämän tehdyn kuvauksen lopputulema on aina tosi, jos se voidaan johtaa määrittelyssä asetetusta lähtötilanteesta käyttämällä loogisia operaatioita. Kun formaalit menetelmät pohjautuvat matematiikkaan ja loogiseen päättelyyn, voidaan määrittelyiden tarkistaminen tehdä tarvittaessa koneiden avulla, jolloin ihmiselementin tuoma virheen mahdollisuus häviää ja tuloksena saadaan jo määrittelyvaiheessa todennettu, toimivaksi todistettu vaatimus (Alagar & Periyasamy, 2011, s. 23).

Formaalit menetelmät voivat osoittautua hyödyllisiksi erityisesti monimutkaisia järjestelmiä suunniteltaessa ja kuvattaessa, jolloin toiminnoista voidaan muodostaa yksityiskohtaiset ja valmiiksi testatut mallit. Hull, Jackson ja Dick (2005, s. 70) mainitsevat, että

formaalit menetelmät soveltuvat erityisen hyvin kriittisten järjestelmien määrittelyyn, jolloin järjestelmällisten matemaattisten menetelmien käytöstä muodostuvat kustannukset ovat oikeutettuja saatuihin hyötyihin verrattuna. Formaalien menetelmien käyttö ei rajoitu myöskään vain vaatimusmäärittelyyn, vaan niitä voidaan käyttää kaikissa ohjelmistokehitysprosessin eri vaiheissa. Alagar ja Periyasamy (2011, s. 24) kertovat, että nykypäivänä formalisointia voidaan käyttää esimerkiksi ohjelman käyttäytymisen määrittämiseen, suunnittelun tukena tai toimintojen määrittelyssä. Varsinainen formaaliuden aste ja käytettävät metodit riippuvat projektin tyypistä, sen koosta sekä tehtävän sovelluksen käyttötarkoituksesta ja sovellusalueesta (engl. application domain). Myös Alagar ja Periyasamy mainitsevat hyöty-kustannussuhteen arvioinnin tärkeänä vaiheena ennen formaalien menetelmien tuomista osaksi kehitysprosessia. Saavutettu hyöty voi olla vähäistä verrattuna mahdollisesta menetelmien käyttöönotosta ja koulutustarpeista, tai tarvittavista työkaluista muodostuviin kustannuksiin verrattuna.

Alagar ja Periyasamy (2011, s. 26-27) jatkavat, että formaalien menetelmien valintaan ja aiemmin mainittuun hyötykustannussuhteeseen vaikuttavat erityisesti tehtävän ohjelman tyyppi, koko ja rakenne, vaadittavan formaaliuden taso, mahdollisuus rajata formaalien menetelmien käyttöä projektin sisällä sekä formaaleja menetelmiä tukevat työkalut.

Ensimmäisenä ja ehkä oleellisimpana vaikuttajana formaalien menetelmien soveltuvuuteen ja kustannushyötysuhteeseen vaikuttaa siis kehitettävän ohjelman tyyppi sekä sen koko ja rakenne. Kaikkien ohjelmistojen kehittämiseen menetelmät eivät automaattisesti sovellus, vaan näitä projektin piirteitä tulee pohtia ennen valinnan tekemistä. Esimerkiksi yksinkertaisten ja pienten ohjelmistojen kehittämiseen ei välttämättä vaadita formaalien menetelmien käyttöä tai niistä ei saavuteta riittävästi hyötyä. Vastaavasti turvallisuuskriittiset ja monimutkaiset sekä suuret helposti pienempiin kokonaisuuksiin ja komponentteihin pilkottavissa olevat ohjelmistot voivat hyötyä formaalien menetelmien käytöstä. Näissäkin tapauksissa formaalien menetelmien käyttö voidaan rajata vain kaikkein kriittisimpiin ja tärkeimpiin komponentteihin ja niiden suunnitteluun sekä määrittelyyn (Alagar & Periyasamy, 2011, s. 26).

Projektin tyypistä riippuen siltä vaadittava formaaliuden taso voi vaihdella. Joissakin tapauksissa voidaan vaatia täsmällisempää ja tarkempaa formaalien menetelmien käyttöä ja jotakin tiettyä menetelmän mukana tuomaa työkalua tai ominaisuutta, kuten täsmällistä semantiikan analysointia tai mekaanista formaaliuden tarkistusta. Alagarin ja Periyasamyn (2011, s. 27) mukaan formaaliuden tasot voidaan jakaa karkeasti kolmeen luokkaan, joissa formaaliuden taso kasvaa korkeammaksi. Ensimmäisellä tasolla luonnollisilla kielillä, kaavioilla ja ei-matemaattisilla menetelmillä tehty määrittely muutetaan

loogisia ja matemaattisia operaatioita käyttävälle kielelle. Toisella tasolla käytetään formaalia määrittelykieltä ja siihen liittyviä työkaluja määrittelyn syntaksin analysointiin ja esitystavan selkeyttämiseen. Määrittelykielten mukana tuomaa datarakenteiden abstrahointia voidaan myös käyttää. Kolmannella tasolla hyödynnetään kaikkia formaalien määrittelymenetelmien tarjoamia mahdollisuuksia. Määrittelyt tehdään käyttämällä formaalia määrittelykieltä ja määrittelyiden analysointiin, seurantaan ja todentamiseen käytetään näitä varten kehitettyjä työkaluja.

Formaalien menetelmien käyttöä projektissa voidaan rajata eri tavoilla. Usein ei suinkaan ole tarvetta hyödyntää formaaleja menetelmiä kaikessa projektitoiminnassa ja kehityksen eri vaiheissa. Rajaamista voidaan tehdä esimerkiksi rajaamalla formaalien menetelmien käyttö vain tiettyyn vaiheeseen ohjelmistokehitysprosessia, kuten vaatimusmäärittelyyn, jolloin tuotetun ohjelmiston laatua voidaan pyrkiä parantamaan havaitsemalla mahdolliset virheet aikaisessa vaiheessa. Toinen mahdollinen rajaustapa on käyttää formaaleja menetelmiä vain tiettyjen komponenttien kehittämisessä. Tällöin menetelmiä voidaan käyttää esimerkiksi vain kaikkein turvallisuuskriittisimmissä komponenteissa, joiden testaaminen on äärimmäisen tärkeää. Järjestelmän todentaminen voi myös vaatia formaalien menetelmien käyttöä, jos järjestelmälle on asetettu jokin kriittinen vaatimus, joka pitää todentaa, voidaan sen määrittelyyn ja todentamiseen käyttää formaaleja menetelmiä (Alagar & Periyasamy, 2011, s. 27).

Viimeisenä huomioitavana osana formaaleja menetelmiä valittaessa on näiden vaatimat työkalut. Formaaleja menetelmiä ei voida käyttää tehokkaasti vain kynällä ja paperilla vaan syntaksin tarkistukseen, semantiikan analysointiin ja teoreeman todistamiseen tarvitaan ohjelmallisia työkaluja (Alagar & Periyasamy, 2011, s. 27).

Kun päätös formaalien menetelmien käyttöönotosta on tehty, pitää sitä koskevat dokumentoinnin ja kommunikoinnin säännöt sopia projektitiimin kesken sekä tiimin jäsenille tarjota tarvittavaa koulutusta. Formaalien menetelmien käyttötapa ja käytänteet voivat vaihdella projektin tai kehitysprosessin vaiheiden mukaan. Otetaan tässä kohtaa kuitenkin huomioon vain vaatimusmäärittelyyn kuuluva työnkuva ja mitä formaaleihin menetelmiin liittyviä käytänteitä vaatimusmäärittelyyn kuuluu.

Formaaleja menetelmiä käytettäessä vaatimusmäärittelyyn kuuluu sekä määrittelytyötä että tehtyjen määrittelyiden analysointia. Molemmissa tehtävissä vaaditaan laaja-alaista osaamista valitun menetelmän käytöstä. Määrittelyitä kirjoittaessa myös laaja kokemus menetelmän syntaksista sekä abstrahoinnista on tarpeen, kuin myös tietämys määriteltävästä prosessista tai toiminnosta. Määrittelyn analysoinnilla tarkoitetaan tehdyn mää-

rittelyn todentamista ja epäjohdonmukaisuuksien ratkaisemista. Analysoinnin tavoitteena on todistaa tehty määrittely oikeaksi ja johdonmukaiseksi. Vaatimusmäärittelytiimin tehtävät eivät pääty valmiin määrittelytyön jälkeen, vaan myöhemmässä vaiheessa heidän tehtävänä on auttaa kehitystiimiä mahdollisissa ongelmatilanteissa tehtyjen määrittelyiden ymmärtämisessä. Koska formaaleilla menetelmillä tehtyjä määrittelyitä voidaan käyttää testaamisen apuna, on vaatimusmäärittelytiimin tehtävänä myös auttaa testaustiimiä toiminnallisten testien luonnissa (Alagar & Periyasamy, 2011, s. 29).

Yhtenä keskeisimmistä tavoitteista formaaleja menetelmiä käytettäessä on virheiden aikainen tunnistaminen. Vaikka vaatimus olisi määritelty ja todistettu oikeaksi formaaleilla menetelmillä, ei se välttämättä takaa sitä, että se toteutusvaiheen jälkeen on vielä tehty oikein. Tästä huolimatta määrittelyvaiheessa tehdyt virheet saattavat kantaa kauaskantoisia vaikutuksia, joten on äärimmäisen tärkeää pyrkiä kitkemään kaikki mahdolliset virheet aikaisessa vaiheessa. Määrittelyiden analysointivaiheessa voidaan pyrkiä löytämään eri tyyppisiä virheitä katselmusten avulla, kuten puuttuvia tai virheellisiä vaatimuksia, formaalin kielen väärinkäytöstä johtuvia virheitä, loogisia virheitä määrittelyn oikeellisuuden todistamisessa, rajoitusten vääränlainen käyttö oikeellisuuteen vaikuttavissa kriteereissä sekä vääränlainen vaatimusten kääntäminen formaalille kielelle. Virheiden kitkemisen tavoitteena on siis pyrkiä validoimaan määrittelyt ennen kuin niitä aletaan toteuttamaan kehitystiimin voimin (Alagar & Periyasamy, 2011, s. 29).

4.1.1 Formaalien menetelmien piirteet ja ominaisuudet

Formaaleille menetelmille voidaan tunnistaa joitakin yhteisiä piirteitä, jotka ovat tunnusomaisia lähes kaikille formaaleille menetelmille. Alagar ja Periyasamy (2011, s. 38-40) mainitsevat kuusi eri ominaisuutta, jotka esiintyvät useimmissa formaaleissa määrittelykielissä. Näitä ominaisuuksia ovat menetelmän luonteeseen kuuluva formaalius, abstrahoinnin käyttö työkaluna, modulaarisuus, epädeterministisyys, pääteltävyys osana mallin käytöksen analysointia sekä mahdollisuus aikasidonnaiseen käyttäytymisen mallintamiseen.

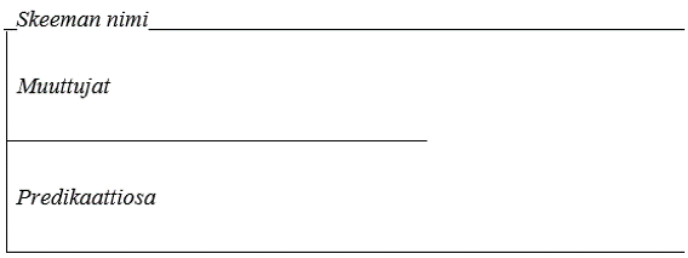
Formaaleista menetelmistä puhuttaessa luonnollisesti selkein erottuva piirre on menetelmän *formalismi*, eli matematiikan ja selkeän rakenteen ja syntaksin tuoma johdonmukaisuus, jolloin kielellä tehtyjä selkeästi strukturoituja määrittelyitä on helppo ymmärtää ja seurata. Toinen usein formaaleihin menetelmiin kuuluva piirre on *abstrahointi*. Abstrahoinnin avulla voidaan luoda määrittelyitä, jotka eivät ole kytköksissä mihinkään arvoihin tai ennalta määritettyyn dataan, vaan luodut mallit tai teoriat muodostetaan manipuloimalla informaatiota loogisella tasolla ja käyttämällä kielten tarjoamia abstrakteja operaa-

tioita olioiden luomiseen, poistamiseen, käsittelyyn sekä näiden välisten relaatioiden luomiseen. Kolmas formaaleihin menetelmiin liittyvä piirre on *modulaarisuus*, jolla tarkoitetaan mahdollisuutta muodostaa monimutkaisia ja suuria kokonaisuuksia pienempiä kokonaisuuksia koskevista määrittelyistä, käyttämällä rikastamista (engl. enrichment) ja koostamista (engl. composition). Abstrahointi mahdollistaa myös *epädeterministisyyden*, joka voidaan mieltää myös yhdeksi yhteiseksi piirteeksi. Tällä tarkoitetaan sitä, että kieli tarjoaa epädeterministiset rakenteet datan epäsuoraan käsittelyyn sekä rajattoman määrän vaihtoehtoja ennalta määriteltyjen tapahtumien joukosta, joka taas mahdollistaa entistä enemmän vapauksia suunnitteluvaiheessa. Eräs toinen formaaleille menetelmille tyypillinen piirre on *pääteltävyys*, eli kieli mahdollistaa mallin käytöksen pääteltävyyden käyttämällä sille asetettuja lakeja ja ennalta määriteltyjä tapahtumia. Viimeisenä piirteenä on mahdollisuus *aikasidonnaisuuteen*, eli mallinnuksessa voidaan määrittää jokin aikavaatimus, joka toteuttaa kyseisen tapahtuman (Alagar & Periyasamy, 2011, s. 38-40).

4.1.2 Z-notaatio

Z-notaatio on ensimmäisen kertaluvun predikaattilogiikkaan ja joukko-oppiin perustuva formaali määrittelykieli. Notaation avulla määrittelyn kohteena olevaa dataa voidaan esittää joukkoina, assosiaatiotauluina (engl. map), monikkoina (engl. tuple), relaatioina sekä karteesisina tuloina. Vastaavasti taas tämän datan manipulointiin voidaan käyttää erilaisia kieleen kuuluvia matemaattisia funktioita ja operaatioita (Dick & Hull & Jackson, 2017, s. 70-71).

Kielellä tehdyt määrittelyt esitetään kaavioina, joita kutsutaan skeemoiksi (engl. schema). Tavoitteena on pitää skeema pienenä sekä helposti luettavana laatikkona, joka koostuu kahdesta osasta (Kuva 8). Skeeman ylemmässä osassa esitellään skeemaan liittyvät muuttujat ja alemmassa predikaattiosassa määritetään esiteltyjen muuttujien suhteita. Varsinaiset Z-notaatiolla tehdyt määrittelyt koostuvat tavallisesti useamman skeeman kokoelmista, jossa yksittäiset skeemat määrittelevät jonkin entiteetin ja nämä yhdessä skeemojen välisten suhteiden avulla muodostavat määrittelyn, jota voidaan käyttää toiminnon kehittämisessä (Dick & Hull & Jackson, 2017, s. 70-71).



Kuva 8: *Z skeeman rakenne (Dick & Hull & Jackson, 2017, s. 71).*

Z-notaatio on alun perin kehitetty IBM United Kingdom Laboratories Limitedin ja Oxfordin yliopiston tietotekniikan laboratorion yhteistyöprojektina, jossa ranskalainen Jean-Raymond Abrial on toiminut kielen alullepanijana (King & Sørensen & Woodcock, 1988). Tämän jälkeen kieltä on kehitetty vuosien saatossa eteenpäin ja vuonna 2002 siitä julkaistiin ISO/IEC 13568 standardi (ISO/IEC, 2002).

Ajan saatossa Z-notaatiota on jatkettu erilaisilla oliopohjaisilla variaatoilla. Alagar ja Peryasamy (2011, s. 539) mainitsevat esimerkkeinä näistä MooZ, ZEST, Z++ ja Object-Z laajennukset. He jatkavat, että näistä mainituista variaatioista Object-Z on saavuttanut eniten suosiota sen laajemman tarjolla olevan kirjallisuuden sekä saatavilla olevien työkalujen vuoksi. Näiden laajennusten tavoitteena on ollut parantaa proseduraalisen Z-kielen soveltuvuutta olioiden määrittämiseen ja suunnitteluun tuomalla siihen mukaan luokkien käsittelyn sekä joukon oliopohjaista ajattelumallia tukevia ominaisuuksia.

Z-notaatio on vahvasti tyypitetty kieli, eli jokaiselle kielellä esitellylle muuttujalle, vakiolle tai lausekkeelle pitää olla määritettynä jokin tyyppi. Z-kielessä käytetään kahdenlaisia tyyppejä; yksinkertaisia tyyppejä (engl. simple types), ja yhdistelmätyyppejä (engl. composite types). Yksinkertaisiin tyyppeihin lukeutuvat kielessä jo määritellyt primitiiviset tyypit, kuten kokonaisluvut sekä käyttäjän määrittelemät tyypit. Yhdistelmätyyppejä taas ovat joukkojen, monikoiden ja karteesisten tulojen avulla määritellyt tyypit. Muita Z-kielellä tehtyjen määrittelyiden esittämisessä ja syntaksissa käytettäviä ominaisuuksia tai esitystapoja ovat lyhentäminen (engl. abbreviation), relaatiot ja funktiot, jonot sekä niin sanotut laukut (engl. bags). Lyhentämisellä tässä tarkoitetaan toistuvasti käytettävälle pitkälle lausekkeelle annettavaa nimeä, jota käytetään mallissa lausekkeen sijasta. Laukut taas ovat Z-notaatiolle ominainen esitystapa elementtien joukoille (Kuva 9), joissa kunkin elementin esiintymismäärä on kirjattu mukaan laukkuun (Alagar & Periyasamy, 2011, s. 461-468).

$$berrybag == \{strawberry \mapsto 4,\ blueberry \mapsto 2,\ raspberry \mapsto 3\}$$

Kuva 9: *Z-notaation laukuissa elementtien esiintymismäärät on sisällytetty mukaan joukkoon.*

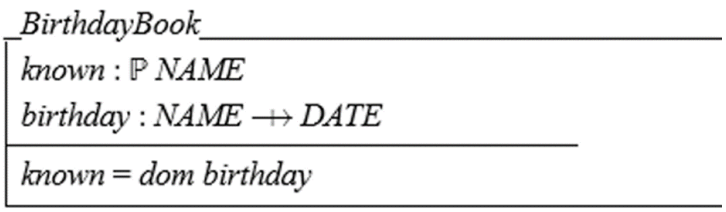
Tarkastellaan vielä joitakin aiemmin mainittuja Z-notaation piirteitä esimerkin avulla ja samalla perehdytään vielä muutamaan esimerkkien ymmärtämisen kannalta välttämättömään merkintätapaan. Kaikkia Z-notaatioon kuuluvia ominaisuuksia ja piirteitä on mahdotonta käydä läpi tämän työn puitteissa, mutta tavoitteena on kuitenkin mahdollistaa perustason ymmärrys kielen käytöstä määrittelyn työkaluna.

Esimerkissä havainnollistetaan yksinkertaisen syntymäpäivien tallentamiseen käytettävän sovelluksen toimintaa. Kyseinen esimerkki pohjautuu Michael Spiveyn (1992) koostamasta Z-notaation käsikirjasta löytyvään syntymäpäiväkirjan määrittelyä koskevaan esimerkkiin. Ensimmäisenä esitellään määrittelyssä käytettävät tyypit, joita ovat nimi sekä päivämäärä. Määrittelyssä ei oteta kantaa siihen, missä muodossa kyseiset arvot ovat, vaan kaikki nimet ja päivämäärät esitellään yleisinä perustyyppeinä (Kuva 10).

$$[NAME, DATE]$$

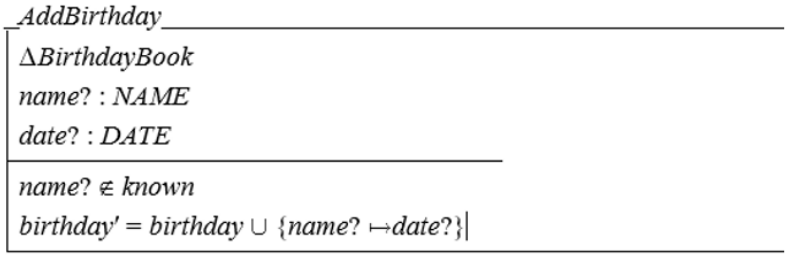
Kuva 10: *Syntymäpäiväkirjan nimiä ja päivämääriä (Spivey, 1992, s. 3).*

Ensimmäinen skeema, joka järjestelmästä muodostetaan, on sen tilaa kuvaava malli (Kuva 11). Tässä siis ylemmässä osassa esitellään järjestelmän tilalle keskeiset muuttujat ja funktiot. Ensimmäisellä rivillä muodostetaan joukko kaikista järjestelmään lisätyistä nimistä *known* nimisen muuttujan alle. $\mathbb{P}$-merkinnällä ilmaistaan, että kyseessä on potenssijoukko (engl. power set), joka tässä tilanteessa tarkoittaa vain sitä, että järjestelmässä olevien nimien määrää ei ole ennalta määritelty. Toisella rivillä määritetään birthday niminen funktio, joka palauttaa sille annettua nimeä vastaavat päivämäärät. Funktioiden esittelyssä käytetään erilaisia nuolta kuvaavia merkintöjä, tässä tapauksessa käytetty $\pfun$ -merkintä tarkoittaa sitä, että kyseessä on osittaisfunktio (engl. partial function). Alemmassa predikaattiosassa määritetään ylemmässä osassa esiteltyjen muuttujien ja lausekkeiden suhteita. Tässä tapauksessa asetetaan järjestelmän tilasta kertova invariantti, että joukko *known* on sama, kuin *birthday* funktion määrittelyjoukko (engl. domain), eli nimien joukko, joita vasten kyseinen funktio voidaan ajaa.



Kuva 11: *Syntymäpäiväkirjan tilaa kuvaava skeema (Spivey, 1992, s. 3).*

Esitellään vielä esimerkkinä yhden metodin kuvaaminen Z-notaation avulla (Kuva 12). Metodin on tarkoitus kuvata uusien syntymäpäivien lisäämistä järjestelmään. Tässä ensimmäisellä rivillä oleva $\Delta BirthdayBook$ viittaa siihen, että yllä mainittuun järjestelmän tilaan kohdistuu muutos. Lisäksi se esittelee kaikki kyseisessä skeemassa esitellyt muuttujat ja näistä kaksi eri variaatiota. Näiden eri variaatioiden on tarkoitus kuvata kyseisen muuttujan tilaa ennen metodin aikaansaamaa muutosta sekä sen jälkeen. Muutoksen jälkeistä muuttujaa kuvataan lisäämällä ’-merkki muuttujan nimen perään. Toinen esimerkissä käytetty uusi merkintätapa ovat kysymysmerkit. Kysymysmerkkien avulla ilmaistaan kyseisten muuttujien olevan syötteitä, eli tässä tapauksessa metodille annetaan nimi ja päivämäärä. Skeeman predikaattiosassa asetetaan ensin esiehto, että annettu nimi ei saa vielä sisältyä tunnettuihin nimiin. Jos tämä esiehto täyttyy, voidaan jatkaa seuraavalle riville, jossa birthday funktion tila päivitetään vastaamaan järjestelmän uutta tilaa. Tämän birthday funktion metodin ajon jälkeistä tilaa kuvataan nyt ’-merkillä ja sen olemassa olevaa laajuutta jatketaan uudella nimi ja päivämäärä parilla.



Kuva 12: *Uuden syntymäpäivän lisäämistä kuvaava skeema (Spivey, 1992, s. 4).*

Kuten aiemmin mainittiin, ei formaalien menetelmien käyttäminen ole mielekästä pelkästään kynällä ja paperilla. Tätä varten Z-notaation käyttöön on saatavilla useampia työkaluja, joista esimerkkeinä voidaan mainita Microsoft Wordiin saatavilla oleva lisäosa Z Word Tools (Hall, 2008) sekä erillinen Z-notaatiolla tehtyjä määrittelyitä varten tehty ohjelmisto CZT: Community Z Tools (Community Z Tool Project, 2003). Ensimmäisenä mainittu lisäosa Wordiin on verrattain kevyt työkalu Z-notaation käyttöön ja tarjoaakin lähinnä erilaisia valmiiksi määritettyjä merkkejä, joiden avulla varsinaiset määrittelyt voi-

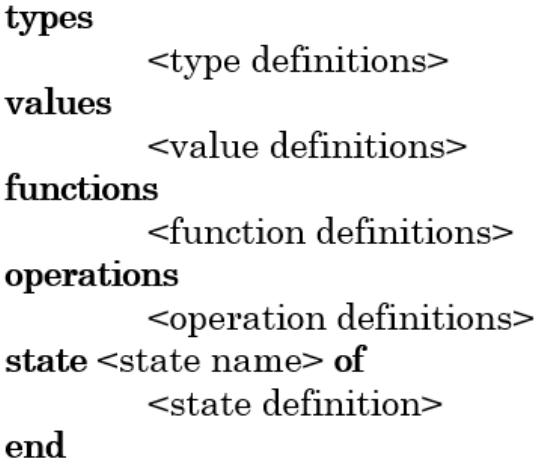
daan muodostaa suoraan Wordissa. CZT taas on avoimen lähdekoodin Java ohjelmistokehys, mutta tarjolla on myös erillinen työkalu, joka pohjautuu Eclipsen ohjelmointiympäristöön (Eclipse foundation, 2019). CZT:n avulla voidaan luoda Z-notaatiolla tehtyjä määrittelyitä sekä muokata, tarkastaa ja animoida näitä (Community Z Tool Project, 2003).

4.1.3 VDM

VDM, eli Vienna Development Method on Z-notaation rinnalla yksi tunnetuimmista formaaleista menetelmistä. Se on saanut alkunsa IBM:n Itävallan Wienissä sijaitsevassa laboratoriossa 1970-luvulla. Kieli on kehitetty Meta-IV määrittelykielen pohjalta, jota käytettiin tuolloin kyseisessä laboratoriossa PL/I -ohjelmointikielen semantiikan määrittelyyn. VDM-kieltä määrittelykielenä käytettäessä puhutaan usein VDM-SL -kielestä, jossa SL-pääte tulee sanoista ”specification language”, eli määrittelykieli (Spivey, 1988, s. 10; Alagar & Periyasamy, 2011, s. 405).

VDM-kielestä on olemassa eri variaatioita suunnattuna eri tarkoituksiin. Näistä yksi on VDM++, joka on suunniteltu käytettäväksi rinnakkaisia ja reaaliaikaisia toimintoja sisältävien oliopohjaisten järjestelmien määrittelyssä. Kyseisessä kielessä on jatkettu VDM-SL-kieltä lisäämällä siihen luokka- ja olioajattelua tukevia elementtejä. Toinen variaatio VDM-kielestä on VDM-RT (engl. VDM real time language), jota käytetään reaaliaikaisten sulautettujen ja hajautettujen järjestelmien mallintamiseen ja analysointiin. Tämä kieli on tosiasiassa vain VDM++ kielen päälle tehty laajennus, eikä siten johda suoraan alkuperäisestä VDM-SL kielestä (Overture, 2018, s. 1-2). VDM-kielestä on julkaistu myös ISO/IEC 13817-1 standardi vuonna 1996. Kyseinen standardi tarjoaa kaiken tarvittavan kielen syntaksista ja semantiikasta. Mukana on sekä kielen matemaattinen syntaksi että ASCII merkkejä käyttävä niin kutsuttu ”interchange”- syntaksi (ISO/IEC, 1996).

VDM-kielellä tehdyt määrittelyt koostuvat eri lohkoista, joissa kussakin esitellään kyseiseen lohkoon liittyviä määrityksiä. Jokainen lohko on eriytetty käyttäen lihavoituja avainsanoja (Kuva 13). Näiden lohkojen järjestyksellä ei ole määrittelyn kannalta merkitystä, eikä kaikkien lohkojen käyttöä vaadita määrittelyiden tekemiseen (Alagar & Periyasamy, 2011, s. 405-407).



Kuva 13: *VDM-kielellä tehdyssä määrittelyssä käytettäviä lohkoja (Alagar & Periyasamy, 2011, s. 406).*

Ensimmäinen kuvassa 13 esitetty lohko on types, joka sisältää määrittelyssä tarvittavat tietotyypit ja niiden esittelyt. Z-notaation tapaan VDM sisältää yksinkertaisia tyyppejä ja yhdistelmätyyppejä. Yksinkertaisiin tyyppeihin lukeutuvat kieleen kuuluvat primitiiviset tyypit, kuten kokonaisluvut, luonnolliset luvut, reaaliluvut ja rationaaliluvut sekä totuusarvot, kirjainmerkit (engl. character) ja token tyyppi. Näistä tyypeistä erikoisimpana on token, jonka avulla voidaan esittää laskettavissa oleva kokoelma toisistaan eriäviä tokeneiksi kutsuttuja arvoja. Tokeneiden avulla voidaan pyrkiä abstrahoimaan muuttujan tyyppiä, jos sen tarkka määrittely toisi vain turhaa kompleksisuutta tehtävään määrittelyyn tai jos halutaan jättää muuttujan tyypin määrittely vasta myöhempään vaiheeseen. Overturen (2018) tarjoaman VDM-kieltä koskevan oppaan mukaan ainoita operaatioita, joita token tyyppisten muuttujien välillä voidaan suorittaa, ovat totuusarvon palauttavat yhtäläisyysvertailut. Yksinkertaisten tyyppien lisäksi VDM-kielessä on niin ikään Z-kielen tapaan yhdistelmätyyppejä, jotka rakentuvat määrittelyssä aiemmin jo esiteltyjen tyyppien pohjalta. VDM-kielessä käytettäviä yhdistelmätyyppejä ovat yhdisteet (engl. union), joukot, sarjat, assosiaatiotaulut, tietueet (engl. record), karteesiset tulot sekä funktiotyypit (Alagar & Periyasamy, 2011, s. 409-416).

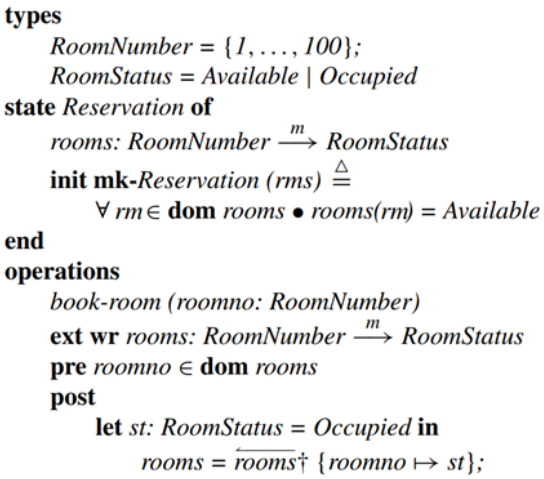
Values lohkoa käytetään globaalien arvojen ja vakioiden esittelyyn. Tämän lohkon sisällä esiteltyjä globaaleja arvoja voidaan siis käyttää missä tahansa määrittelyn eri tasoilla. Näitä arvoja käytettäessä tulee kuitenkin huomioida, että vaikka VDM-kielessä lohkojen järjestyksellä ei varsinaisesti ole merkitystä, pitää arvojen olla asetettuna ennen niiden käyttöä myöhemmissä lohkoissa. Tämä tulee siitä, että jos määrittely halutaan suorittaa käyttämällä jotakin VDM-tulkkia, pitää arvojen olla asetettuna ennen niiden käyttöä (Overture, 2018, s. 97).

Kolmas kuvassa 13 esitetty lohko on functions osio, jota käytetään määrittelyyn kuuluvien funktioiden esittelyyn. Alagarin ja Periyasamyn (2011, s. 422-424) mukaan näitä funktioita on VDM-kielessä neljän tyyppisiä; implisiittisiä funktioita, eksplisiittisiä funktioita sekä korkeamman asteen funktioita ja polymorfisia funktioita. Fitzgeraldin, Larsenin, Mukherjeen, Platin ja Verhoefin (2005, s. 14) mukaan implisiittiset funktiot määritellään esittelemällä ominaisuudet, jotka funktion tulisi toteuttaa, tarkan laskentatavan määrittämisen sijaan. Vastaavasti eksplisiittisissä funktioissa esitellään tarkka algoritminen laskentamenetelmä, miten annetuista parametreista saadaan tavoiteltu lopputulos. Korkeamman asteen funktiot, joita kutsutaan myös curryfunktioiksi (engl. curried functions), ovat funktioita, joiden tuloksena on arvon sijasta toinen funktio. Polymorfiset funktiot ovat taas geneerisiä funktioita, joiden määrittely ei riipu funktiolle annettavista parametreista. Sen sijaan määrittely voidaan tehdä antamalla funktiosta pohja, joka voidaan instantioida myöhemmin sopivilla parametreilla.

VDM-määrittelyiden operations osiota käytetään esittelemään määrittelyihin liittyvät operaatiot. Operaatiot ovat funktioiden kaltaisia toimintoja, mutta mainittavana erona näillä on se, että operaatiot sisältävät osion ulkopuolisten viittausten esittelyyn, sekä lohkon virhetilanteiden määrittelyyn. Myös operaatioita voi olla implisiittisiä ja eksplisiittisiä. Tässäkin tilanteessa implisiittisyydellä tarkoitetaan sitä, että operaatio esitellään ilman tarkan toteutustavan määrittelyä ja eksplisiittisyydellä sitä, että operaatiolle annetaan tarkka algoritmi, jonka perusteella syötteitä ja instantioituja arvoja käsitellään (Fizgerald, ym. 2005, s. 14) (Alagar & Periyasamy, 2011, s. 424-426).

Kuvassa 13 viimeisenä esitelty osio on tilan esittelyyn käytetty state lohko. Sen tarkoituksena on esitellä ongelma-alueeseen (engl. problem domain) liittyvät oliot ja näin muodostaa malli määrittelyn kohteena olevasta toiminnosta. Tiloihin voidaan kohdistaa muutoksia käyttämällä aiemmin mainittuja operaatioita (Alagar & Periyasamy, 2011, s. 417-419).

Tarkastellaan vielä VDM-kielen käyttöä määrittelyssä esimerkin avulla. Esimerkki pohjautuu Alagarin ja Periyasamyn (2005, s. 406) kirjassa esittelemään hotellihuoneiden varausjärjestelmää koskevaan VDM-määrittelyyn (Kuva 14).



Kuva 14: *Hotellin varausjärjestelmän VDM-määrittely (Alagar & Periyasamy, 2005, s. 406).*

Ensimmäisenä esimerkissä muodostetaan tarvittavat tyypit. Huoneiden varauksen kannalta olennaisia ovat huoneen numero, tässä tapauksessa väliltä 1-100, sekä huoneen tila, onko se varattu vai vapaa. Tämän jälkeen määritellään varauksen tilaa koskevat ominaisuudet. Tässä tapauksessa muodostetaan kokoelma "rooms" kaikista huoneista, jossa yhdistetään kukin huonenumero kyseisen huoneen tilaan mappaamalla. Samalla varauksen tilassa muodostetaan tätä varausta koskeva kaava (engl. pattern), jossa varmistetaan huoneen olevan vapaa. Lopuksi määrittelyssä kuvataan huoneen varaamista koskeva operaatio "book-room", joka muokkaa edellä esitettyä järjestelmän varauksia koskevaa tilaa. Operaatiossa viitataan tilassa muodostettuun rooms-kokoelmaan, jota vasten esiehto tarkistetaan. Operaatiolle annetun huonenumeron tulee kuulua olemassa oleviin huoneisiin. Jos esiehto täyttyy, suoritetaan jälkiehdon operaatio, jossa huoneen tila muutetaan varatuksi ja järjestelmän rooms kokoelmaa päivitetään sen mukaiseksi.

VDM-kielen tehokas käyttö määrittelyiden tekemiseen vaatii Z-notaation lailla työkalujen hyödyntämistä. Tarjolla onkin useita työkaluja, joista tällä hetkellä nimekkäin on todennäköisesti avoimen lähdekoodin ohjelmistoympäristö Overture tool (Overture, 2020a). Ohjelma on kirjotettu Javalla ja perustuu Eclipse alustaan. Overtureen on tarjolla erilaisia laajennuksia, jotka muokkaavat työkalua sopivammaksi tietyn tyyppisten järjestelmien suunnitteluun. Yhtenä esimerkkeinä näistä on Crescendo tool (Overture, 2020b), joka keskittyy käyttämään yhteissimulointia (engl. co-simulation) kyberfyysisten järjestelmien (engl. kyber-physical systems) suunnitteluun ja mallintamiseen. Toinen tarjolla oleva laajennus on Symphony tool (Overture, 2020c), jota käytetään useammasta järjestelmästä koostuvien järjestelmien (engl. system of systems) mallintamiseen. VDM-kielellä tehtyjä

määrittelyjä on mahdollista tehdä myös LaTeXilla, käyttämällä sille tehtyjä makroja (Lines, 1995).

4.2 Puoliformaalit ja informaalit määrittelymenetelmät

Formaalien menetelmien ulkopuolelle jääviä määrittelymenetelmiä kutsutaan puoliformaaleiksi ja informaaleiksi määrittelymenetelmiksi. Puoliformaalit menetelmät tarjoavat jonkin asteista formaaliutta, mutta niitä ei lueta varsinaisten formaalien menetelmien joukkoon. Näistä hyvänä esimerkkinä toimii Yhtenäistetty mallinnuskieli, eli UML (engl. unified modeling language), joka tarjoaa järjestelmällisen tavan kuvata ohjelmiston toimintaa eri kaavioiden avulla. UML ei ole varsinaisesti määrittelykieli, vaan sitä käytetään monesti ohjelmistojen käyttäytymisen dokumentoinnissa, tästä huolimatta se voi kuitenkin toimia myös tehokkaana työkaluna määrittelyitä tehtäessä. Informaaleilla menetelmillä taas tarkoitetaan luonnollisten kielten ja taulukoiden sekä kuvaajien käyttöä ohjelmiston toiminnan määrittelyssä. Tutustutaan tässä luvussa sekä UML kielen käyttöön määrittelyitä tehtäessä, että luonnollisten kielten asemaan tämän hetken vaatimusmäärittelyssä.

4.2.1 UML

Ihminen on tottunut mallintamaan ja visualisoimaan asioita aikojen alusta saakka. Visualisoimalla voidaan pyrkiä helpottamaan vastapuolta ymmärtämään vaikeastikin hahmotettavia yksityiskohtia. Mallintamisesta onkin tullut perustyökalu kaikilla insinöörialoilla, sillä sen avulla voidaan havainnollistaa esimerkiksi sillan arkkitehtuuria tai analysoida tuulen vaikutuksia rakennelmiin. Tieto- ja ohjelmistotekniikassa mallintamisella pyritään kuvaamaan usein jokin järjestelmän osa tai tapahtuma. Mallintamisella onkin merkittävä rooli ohjelmistojen suunnittelussa ja kattavan käsityksen muodostamisessa kehitettävästä järjestelmästä. Ohjelmistokehityksen kannalta ajateltavalla mallintamisella on neljä tehtävää. Sillä pyritään visualisoimaan järjestelmää ja sen toimintaa, sen avulla määritellään kehitettävän järjestelmän rakennetta ja käyttäytymistä, niiden avulla voidaan muodostaa järjestelmän kehitystä ohjaavat raamit ja viimeisenä niiden avulla voidaan dokumentoida toteutuneen järjestelmän toimintaa (Booch & Jacobson & Rumbaugh, 2005).

Yksi tietotekniikassakin yleisesti käytetty mallinnuskieli on UML. Se on graafinen notaatio, jonka avulla voidaan määrittää, visualisoida ja dokumentoida sekä mallintaa ohjelmiston eri toimintoja ja vaiheita. Mallinnuskielen tarkoituksena on standardoida eri mallinnustehtävissä tarvittavat kuvaukset yhden yhteisen kielen alle. Koska UML on niin moniulotteinen, käytetään sitä ohjelmistokehityksessä lähes kaikissa sen eri vaiheissa. Sen

kaavioita voidaan käyttää apuna vaatimusten esillesaannissa, määrittelyssä, suunnittelussa sekä dokumentoinnin eri vaiheissa. Määrittelymenetelmistä puhuttaessa UML sijoittuu formaalien ja informaalien menetelmien välimaastoon. Se ei luonnollisesti kuulu formaalien menetelmien joukkoon, mutta sen sisältämien formaalien piirteiden vuoksi sitä voidaan pitää hieman formaalimpana vaihtoehtona informaaleille menetelmille. Laplante (2004, s. 146) mainitsee, että UML:n asemasta on kiistelyä, onko se varsinaisesti puoliformaali menetelmä vai ei. Joidenkin mielestä sen sisältämät pseudomatemaattiset piirteet oikeuttavat sille puoliformaalin aseman, kun toiset taas epäävät asemaa kielen sisältämien puutteiden ja epäjohdonmukaisuuksien vuoksi.

UML on saanut alkunsa 1990-luvulla leikkisästi kolmeksi amigoksi kutsutun kolmikon, eli Grady Booschin, Jim Rumbaughin ja Ivar Jacobsonin toimesta. Kyseiseen aikaan ohjelmistojen mallintamiseen sekä analysointiin oli tarjolla useampia eri metodeja ja notaatioita, mutta mitään yhteistä standardia ei kuitenkaan ollut vielä kehitetty. Tämän seurauksena idea yhteisestä mallinnuskielestä sai alkunsa ja tarkoituksena oli yhdistää Booschin kehittämä Boosch-metodi, Rumbaughin kehittämä OMT-metodi (engl. object modeling technique) sekä Jacobsonin kehittämä OOSE-notaatio (engl. object oriented software engineering) yhdeksi yhteiseksi kieleksi. Vuoden 1996 lopulla UML-kielen versio 0.9 oli valmis ja sen saavuttaman huomion myötä standardoinnin saralla toimiva OMG (engl. object management group) tarjoutui tukemaan kielen valmistumista. Vuoden 1997 tammikuussa kyseinen työ saatiin päätökseen ja UML kielen versio 1.0 oli valmis. Samaan aikaan toinen yritysten ryhmittymä toimitti standardointikomitealle oman ehdotelmansa UML-kieleksi. Tätä ei kuitenkaan tulkittu kilpailevana ehdotelmana, vaan enemmänkin täydentävänä. Tämä johti siihen, että ryhmittymät jatkoivat työskentelyä yhdessä ja yhteistyön tuloksena saatiin valmiiksi UML 1.1 syyskuussa 1997. Tämänkin jälkeen kieltä kehitettiin eteenpäin aina versioon 1.5 asti (Pender, 2003). Myöhemmin kielestä on tehty myös uudistettu UML 2.0 versio, joka saatiin valmiiksi 2003 ja julkaistiin virallisesti vuonna 2005 (Weilkiens & Oestereich, 2006, s. 4).

UML 2.0 kielestä on julkaistu vuonna 2012 kahteen osaan jaettu ISO/IEC standardi. Ensimmäinen ISO/IEC 19505-1:2012 osa koskee UML-kielen infrastruktuuria (ISO/IECa, 2012) ja toinen ISO/IEC 19505-2:2012 taas koskee kielen superstruktuuria (ISO/IECb, 2012). Yhdessä ne muodostavat kokonaisen UML 2.0 standardin.

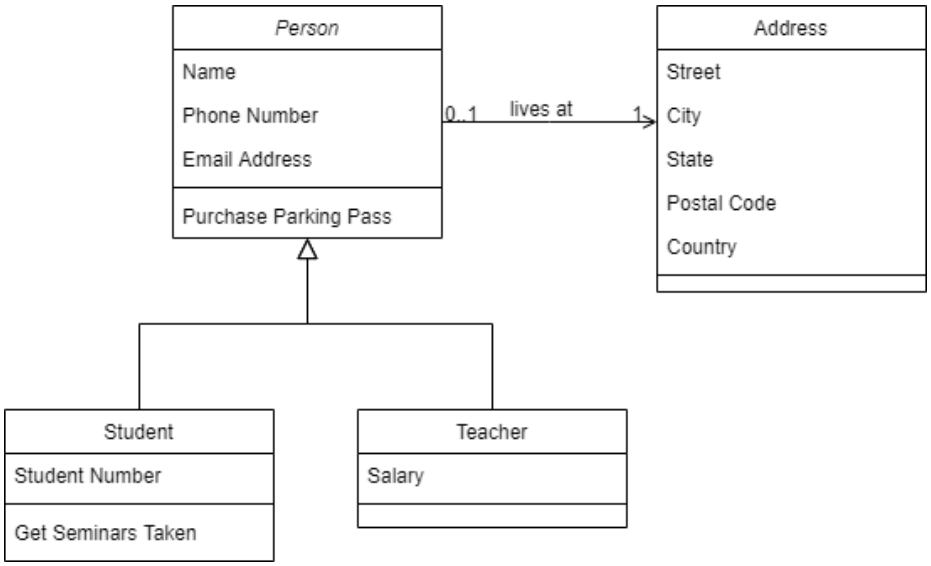
UML 2.0 kieli jakaa käytettävissä olevat kaaviot kahteen kategoriaan. Rakennekaavioihin (engl. structural diagrams) ja käyttäytymiskaavioihin (engl. behavioral diagrams). Rakennekaavioita käytetään ohjelmiston rakenteen suunnitteluun ja esittämiseen, kuten siihen miten jokin olio on relaatiossa toisen kanssa. Käyttäytymiskaavioilla voidaan taas kuvata

sananmukaisesti ohjelman käyttäytymiseen liittyviä tapahtumia, kuten esimerkiksi ohjelman sisäisiä tilanmuutoksia (Pilone & Pitman, 2005).

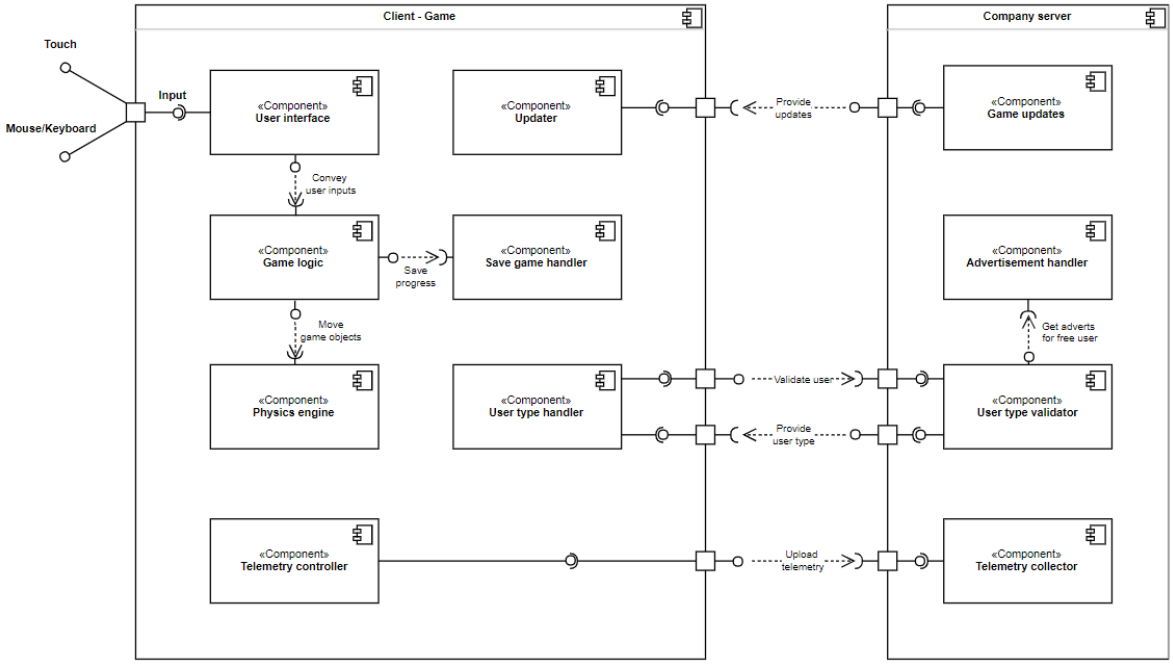
Rakennekaavioita kieleen kuuluu yhteensä seitsemän kappaletta; luokkakaaviot (engl. class diagrams), komponenttikaaviot (engl. component diagrams), koostekaaviot (engl. composite structure diagrams), sijoittelukaaviot (engl. deployment diagrams), pakkauskaaviot (engl. package diagrams) sekä oliokaaviot (engl. object diagrams) (Pilone & Pitman, 2005). Näiden lisäksi UML 2.0:aan kuuluvat profiilikaaviot (engl. profile diagrams), joiden tarkoituksena on mahdollistaa UML-kielen laajentaminen tarjoamalla mahdollisuuden luoda omia stereotyyppejä, merkittyjä arvoja ja rajoitteita (Uml-diagrams, 2020).

Käyttäytymiskaavioihin taas lukeutuvat aktiviteettikaaviot (engl. activity diagrams), kommunikointikaaviot (engl. communication diagrams), kokoavat vuorovaikutuskaaviot (engl. interaction overview diagrams), sekvenssikaaviot (engl. sequence diagrams), tilakaaviot (engl. state machine diagrams), ajoituskaaviot (engl. timing diagrams) sekä käyttötapauskaaviot (engl. use case diagrams) (Pilone & Pitman, 2005).

Jokaisella näistä UML-kaavioista on omat käyttötarkoituksensa. Ohjelmistokehitystä tarkasteltaessa rakennekaavioista tunnetuimpia ja laajimmin käytettyjä ovat todennäköisesti luokkakaaviot (Kuva 15) sekä komponenttikaaviot (Kuva 16). Luokkakaavioiden avulla voidaan helposti visualisoida kehitettävän järjestelmän rakennetta ja toimintaa kuvaamalla ohjelmaan kuuluvia luokkia sekä näiden välisiä rajapintoja. Vastaavasti komponenttikaavioiden avulla voidaan havainnollistaa järjestelmän muodostumista ja sen riippuvuuksia komponenttitasolla. Komponenttitasolla kuvaaminen voi hieman helpottaa kokonaiskuvan muodostamista järjestelmästä, kun pienemmistä elementeistä muodostuvat järjestelmän osat on voitu ryhmittää yhden helpommin hallittavan kokonaisuuden, eli komponentin alle (Pilone & Pitman, 2005).



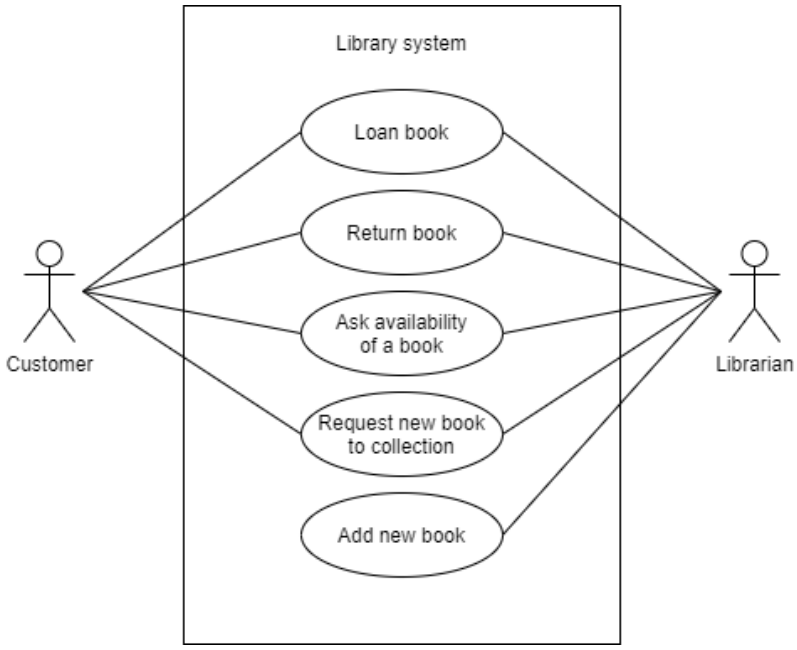
Kuva 15: *Esimerkki luokkakaaviosta, jossa esitetään henkilön tietojen muodostumiseen liittyvät luokat kouluympäristössä.*



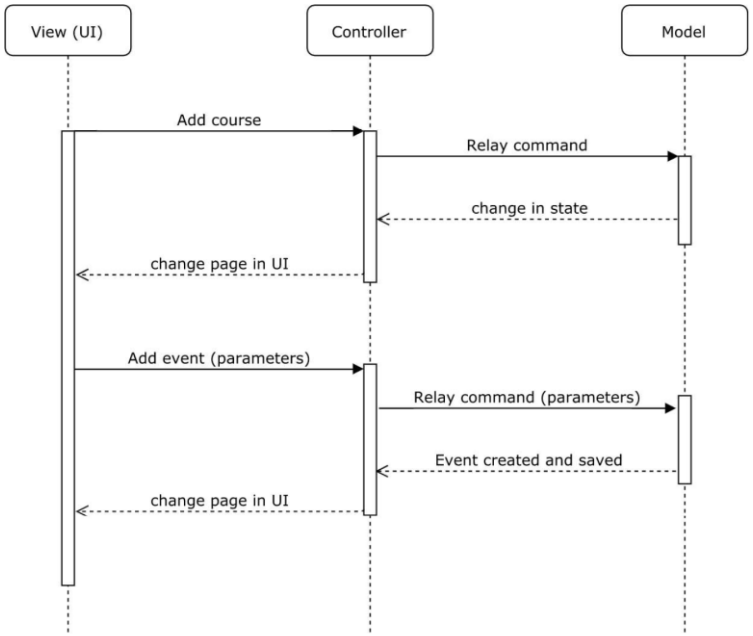
Kuva 16: *Esimerkki komponenttikaaviosta, joka kuvaa videopelin rakenteen muodostumisen eri komponenteista.*

Vastaavasti käyttäytymiskaavioista esimerkeiksi voidaan ottaa vaatimusmäärittelyssäkin hyödyllisenä apuna olevat käyttötapauskaaviot (Kuva 17) sekä sekvenssikaaviot (Kuva 18). Käyttötapauskaavioilla voidaan havainnollistaa toiminnallisten vaatimusten muodostumista käyttäjän näkökulmasta, ilman puuttumista varsinaiseen toteutukseen ja sen teknisiin yksityiskohtiin. Käyttötapauskaaviot voivat koostua kolmenlaisista elementeistä; nimetyistä toiminnallisuuksista eli käyttötapauksista (engl. use cases), henkilöistä tai toi-

mijoista, jotka käynnistävät tapahtuman (engl. actors) sekä elementeistä, joiden vastuulla kyseisen toiminnan toteuttaminen on (engl. subjects). Sekvenssikaavioissa tapahtumat voidaan esitellä yksityiskohtaisemmin aikajärjestyksessä. Näiden kaavioiden tarkoituksena on yleensä havainnollistaa eri toimijoiden välinen viestinvaihto sekä tätä kautta myös tehtävien roolitus (Pilone & Pitman, 2005).



Kuva 17: *Esimerkki käyttötapauskaaviosta, jossa kuvataan kirjaston lainausjärjestelmän toimintaa.*



Kuva 18: *Esimerkki sekvenssikaaviosta, joka kuvastaa viestinvaihtoa MVC-arkkitehtuurin mukaisessa ohjelmassa.*

UML-kieli ei välttämättä yksistään riitä toimimaan määrittelyiden ainoana kuvaustapana. Tehokkaimmillaan se on juuri silloin, kun sitä käytetään tukemaan luonnollisilla kielillä tehtyjä määrittelyitä. Koska UML-kieli on niin joustava, soveltuu se lähes mihin tahansa tilanteeseen. On tarve sitten tukea vaatimusten esillesaantia tai dokumentoida jokin vaatimus määrittelynä, on siihen todennäköisesti tarjolla jokin tilanteeseen sopiva UML-kaavio.

Pilone ja Pitman (2005) mainitsevat neljä UML-kaavioiden käyttöön liittyvää muistisääntöä. Ensimmäinen näistä on se, että UML-kielessä lähes kaikki on valinnaista. Tämä tarkoittaa sitä, että vaikka kieli tarjoaa runsaasti eri vaihtoehtoja mallinnettavan kohteen kuvaamiseen, ei kaikkia näitä vaihtoehtoja tarvitse käyttää. Tarkoituksena on pyrkiä käyttämään vain tarpeellinen osa tarjolla olevasta syntaksista ja pitämään malli mahdollisimman yksinkertaisena. Toinen näistä muistisäännöistä on se, että UML-mallit ovat harvoin kokonaan valmiita. Tämä on osittain ensimmäisen kohdan johdannainen ja malleja yleensä työstetään iteroimalla sekä kehittämällä mallia eteenpäin kerätyn palautteen pohjalta. Mallin ei siis tarvitse olla täydellinen, mutta kaiken järjestelmän lopulliseen suunnitteluun vaikuttavan pitäisi ilmetä mallista. Kolmantena kohtana Pilone ja Pitman mainitsevat UML-kielen olevan tarkoituksella suunniteltu siten, että se on avoin tulkinnalle. Tämä tarkoittaa sitä, että kaikki kielen tarjoamat ominaisuudet eivät ole yksiselitteisiä, vaan organisaatioilla on mahdollisuus kehittää omat käytänteet kyseisten ominaisuuksien käyttöön. Tästä johtuen onkin suositeltavaa, että organisaation käytänteistä muodostetaan ohjesäännöt, joiden avulla kaavioiden tulkinta olisi helpompaa. Viimeisenä kohtana mainitaan UML-kielen tarjoama laajennettavuus. Kieleen on sisällytetty mekanismeja, joiden avulla kieltä voidaan jalostaa sopivammaksi organisaation tarpeisiin. Näitä ovat jo aiemmin tekstissä mainitut profiilit, joiden avulla kieleen voidaan lisätä koristeita (engl. adornments), rajoitteita sekä stereotyyppejä.

4.2.2 Informaalit määrittelymenetelmät

Formaalien ja puoliformaalien menetelmien ulkopuolelle jäävillä informaaleilla menetelmillä tarkoitetaan usein luonnollisten kielten käyttöä määrittelyssä. Luonnollisten kielten lisäksi informaaleihin menetelmiin lukeutuvat erilaiset kaaviot, taulukot sekä kuvaajat, joiden avulla määrittelyä voidaan pyrkiä selventämään. Laplante (2004, s. 164) kiteyttää hyvin, että määrittelymenetelmä on informaali, jos sitä ei voida kokonaisuudessaan muuntaa täsmälliseksi matemaattiseksi notaatioksi.

Ohjelmistokehityksessä juuri informaalit menetelmät ovat ylivoimaisesti käytetyimpiä. Tämä johtuu todennäköisesti siitä, että niiden käyttö ei vaadi minkäänlaista erityiskoulutusta määrittelyn tekijältä tai valmiin määrittelyn lukijalta. Lisäksi luonnollisten kielten

käyttöä puoltaa niiden käytön nopeus. Määrittelyt ovat tehtävissä nopeasti ja tehokkaasti, jonka vuoksi niitä käytetään runsaasti tämän päivän ketterissä ohjelmistokehitysmenetelmissä. Ketterästi ja nopeasti tehtyjä määrittelyjä voidaan jatkojalostaa ja kehittää eteenpäin iteratiivisesti muutostarpeiden ilmaantuessa.

Luonnollisilla kielillä tehdyt määrittelyt ovat muotoilultaan ja rakenteeltaan usein hyvin vapaita, eikä niiden tekoon ole mitään ennalta määritettyä ohjaavaa mallia. Tämä voi johtaa helposti siihen, että määrittelyistä tulee hankalasti luettavia ja epäselviä tai virheellisiä. Sommerville (2016, s. 122) esittääkin viisi ohjesääntöä luonnollisilla kielillä tehtäviin määrittelyihin, joiden avulla voidaan välttyä mahdollisilta väärinymmärryksiltä. Ensimmäinen näistä on se, että organisaatiossa kehitettäisiin yhtenäinen malli, jonka pohjalta kaikki määrittelyt tehdään. Kun kaikissa määrittelyissä on käytetty samaa formaattia, on niiden ymmärtäminen ja tarkistaminen helpompaa. Toinen kohta on käyttää määrittelyissä aina samaa muotoilua pakollisten ja toivottujen vaatimusten erottamiseksi. Sommerville myös suosittelee tekstimuotoilujen, kuten lihavoinnin, kursivoinnin tai värien käyttämistä tärkeimpien kohtien korostamisessa. Määrittelyitä tehtäessä olisi myös suositeltavaa pyrkiä välttämään liian teknistä sanastoa. Kaikki määrittelyä lukevat tahot eivät välttämättä ymmärrä kyseistä sanastoa, joten lyhenteiden ja teknisten sanojen käyttö voi johtaa uusiin väärinymmärryksiin. Viimeisenä kohtana Sommerville mainitsee vaatimusten perustelun määrittelyitä tehtäessä. Määrittelyihin tulisi kirjata keneltä ja millä perusteilla kyseinen vaatimus on esitetty. Näin myöhemmin tarvittavien lisätietojen saanti on helpompaa.

Vapaasti muotoiltujen määrittelyiden vastineeksi tarjolla on myös niin kutsutut strukturoidut määrittelyt, joissa luonnollisilla kielillä tehty määrittely on pyritty muodostamaan jonkin standardoidun tavan mukaisesti. Strukturoiduissa määrittelyissä käytetään yleistä dokumenttipohjaa, jolloin määrittelyt noudattavat yleistä totuttua kaavaa ja näin niiden hallinta ja käsittely on helpompaa. Strukturoiduilla määrittelyillä ei yksistään voida kitkeä mahdollisten väärinymmärrysten syntymistä, joten lisäinformaation lisääminen luonnollisilla kielillä kirjoitettujen selosteiden yhteyteen kaavioiden, taulukoiden, kuvaajien tai mallien muodossa on suositeltavaa (Sommerville, 2016, s. 122-124).

Vaikka luonnollisten kielten käyttö määrittelyiden tekemiseen onkin kaikkein yleisintä (Luisa & Mariangela & Pierluigi, 2004), on niitä kohtaan esitetty runsaasti kritiikkiä. Yhtenä kritiikin aiheena on luonnollisten kielten käytöstä seuraava epäselvyys. Määrittelyn kirjoittajalla ja lukijalla pitäisi olla samanlainen sanasto ja käsitys käytetystä kielestä, minkä vuoksi määrittely voidaan helposti ymmärtää väärin. Tämä ei rajoitu pelkästään aiemminkin mainittuun tekniseen sanastoon, vaan normaalissa kielessäkin esiintyy runsaasti samankaltaisia sanoja, jotka voivat johtaa epäselvyyksiin määrittelyissä. Tämä

johtaa myös siihen, että sama asia voidaan selittää useammalla eri tavalla, jonka vuoksi samasta asiasta voidaan huomaamatta luoda useampi eri vaatimus ja määrittely. Samasta syystä määrittelyiden ryhmittely voi olla haastavaa, jonka seurauksena muutoksien ilmaantuessa joudutaan käymään pahimmassa tapauksessa kaikki vaatimukset läpi muutoksen seurausten kartoittamiseksi (Sommerville, 2020).

5. FORMAALIT MENETELMÄT KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

Formaalit menetelmät ovat olleet osa ohjelmistokehityksessä käytettävää menetelmien kirjoa jo vuosikymmeniä. Tästäkin huolimatta niiden käyttö nykypäivänä on verrattain vähäistä, lähes marginaalista. Tutkitaan tässä luvussa hieman tarkemmin formaalien menetelmien käyttöastetta nykypäivänä kirjallisuusselvityksen avulla sekä tutustutaan myös formaalien menetelmien nykytilanteeseen ja mahdollisiin tulevaisuudennäkymiin. Diplomityö tehtiin osana toimeksiantajayrityksen ohjelmistoprojektia, johon liittyen ketterien menetelmien mukaisia määrittelymenetelmiä tutkittiin. Tutustutaan kirjallisuusselvityksen ohella case-tutkimuksen avulla projektissa toteutuneeseen määrittelytyöhön sekä projektin luonteen vaikutuksiin määrittelytyön tekemisessä.

5.1 Formaalien menetelmien tuomat hyödyt ja haasteet

Formaalien menetelmien käytöllä pyritään usein tavoittelemaan erinäisiä hyötyjä. Yleensä nämä saadut hyödyt voidaan yhdistää jollakin tavalla ohjelmiston laadun parantamiseen. Siksi onkin tavallista, että formaaleja menetelmiä käytetään nimenomaan, jos ohjelmiston oikeellisuus tai turvallisuus on kriittistä.

Ohjelmiston laadun kannalta kriittisimpiä vaiheita ovat tyypillisesti vaatimusmäärittely sekä testaaminen. Juuri näihin ohjelmistokehitysprosessin vaiheisiin liittyen Batra, Malik ja Dave (2013) nostavatkin tutkimuksessaan formaalien menetelmien hyödyiksi mahdollisuuden mitata ohjelman oikeellisuutta, aikaisen virheiden havaitsemisen, takuun oikeellisuudesta, virhealttiuden vähenemisen, abstrahoinnin lisääntymisen, analysoinnin tehostamisen, ohjelman luotettavuuden lisääntymisen sekä määrittelyistä suoraan saatavat testitapaukset. Formaalien menetelmien vaikutuksesta koodin laadun parantamisessa on tehty runsaasti tutkimuksia, mutta pelkkä formaalien menetelmien käyttäminen ei tarkoita automaattisesti laadukkaampaa ohjelmistoa. Pfleeger ja Hatton (1997) toteavat tutkimuksessaan, että tosiasiassa laadukasta koodia tavoiteltaessa vaaditaan formaalien menetelmien käytön lisäksi myös koodin staattista analysointia sekä kattavat yksikkötestit. Vaikka formaaleilla menetelmillä onkin tärkeä rooli määrittelyiden ja testien oikeellisuuden varmistamisessa, eivät ne yksistään riitä laadukkaan ohjelmiston tuottamiseen.

Formaaleilla menetelmillä voidaan siis todistetusti saavuttaa merkittäviä hyötyjä määrittelyn ja testauksen saralla, tosin nämä hyödyt eivät kuitenkaan tule ilman rajoitteita ja

haasteita. Batra, ym. (2013) tuovat esille tutkimuksessaan joitakin formaalien menetelmien käyttöön liittyviä huomioita. Yksi näistä liittyy formaaleilla menetelmillä tehtävien määrittelyiden oikeellisuuteen. Tyypillisesti jossakin vaiheessa vaatimusmäärittelyä joudutaan turvautumaan luonnollisten kielten käyttöön, jonka vuoksi mahdolliset epäjohdonmukaisuudet voivat periytyä myös formaaleilla menetelmillä tehtyihin määrittelyihin. Tämän vuoksi takeita tehtyjen määrittelyiden oikeellisuudesta ei aina voida antaa. Muita ongelmakohtia heidän mukaansa liittyy ohjelman oikeellisuuden todentamiseen, eli vastaako toteutettu ohjelma annettua määrittelyä. Lisäksi formaaleille menetelmille on tyypillistä käyttää todistamista apuna määrittelyiden oikeellisuuden todentamisessa ja myös näiden todistusten oikeellisuuteen voi liittyä joitakin ongelmia. Viimeisenä rajoitteina Batra, ym. (2013) mainitsevat formaalien kielten kompleksisuuden sekä ohjelman toimintaympäristön kuvauksen puuttumisen, eli ohjelman oikeellisuuden tarkistukseen ei voida liittää käytettävän ympäristön ominaisuuksia.

Knight (1998) mainitsee formaalien menetelmien käyttöön liittyviksi haasteiksi menetelmien integroimisen yhtäaikaisesti kaikkiin ohjelmistokehityksen elinkaaren vaiheisiin, formaaleilla menetelmillä tehtyjen analyysien sisällyttäminen järjestelmätason luotettavuusanalyysiin sekä formaaleja menetelmiä tukevien työkalujen puutteen. Jotta formaalien menetelmien käytöstä saataisiin kaikki hyöty irti, tulisi niitä käyttää kaikissa ohjelmistokehityksen vaiheissa. Monet näistä ongelmista ovat edelleen läsnä tänäkin päivänä, joskin työkalujen saatavuus on voinut parantua ajan saatossa. Myöskin ketterät menetelmät voivat tuoda omat vaatimuksensa formaaleille menetelmille ja niiden käytölle.

Yhtenä formaalien menetelmien rajoittavana tekijänä ja yleistymisen esteenä on myös kielten korkea oppimiskynnys (Wolff, 2012). Koulutustarpeen myötä lisääntyvien kustannusten ja toimitusaikataulujen venymisen riskit eivät kannusta yrityksiä ottamaan formaaleja menetelmiä käyttöön, eikä määrittely- ja analyysivaiheissa vaadittava lisätyö välttämättä kanna hedelmää kehitysprosessin myöhemmissä vaiheissa.

5.2 Formaalit menetelmät ja ketteryys

Ketterät ohjelmistokehitysmenetelmät tuovat oman lähestymistapansa formaaleihin menetelmiin. Ketterille menetelmille tyypillinen jatkuva muutoksiin reagoiminen ei suoraan sovellu formaalien menetelmien normaaliin käyttötapaukseen. Formaalien menetelmien käyttö yleisimmin sisältää runsaasti etukäteen tehtävää dokumentointia ja analysointia, kun toiminnallisuutta pitää todistaa oikeaksi. Ketterissä menetelmissä vastaavasti dokumentointi pyritään jättämään minimiin, jotta säilytetään mahdollisuus reagoida nopeasti ja tehokkaasti mahdollisiin muutoksiin projektin edetessä. Löwe (2010) mainitseekin yhdeksi ongelmakohdaksi formaalien menetelmien käytöstä ketterissä menetelmissä juuri

jatkuvien muutosten myötä aiheutuvan koodin refaktoroinnin lisäksi tehtävän formaalien todistusten uudelleen kirjoittamisen. Jatkuva tarve muutoksien tekemiseen voi syödä formaalien menetelmien käytöstä saavutettavia hyötyjä, jos muutosten mukana pysymiseen vaaditaan paljon työtä.

Formaaleilla menetelmillä on kuitenkin hyötynsä tuvallisuuskriittisiä järjestelmiä kehitettäessä sekä analysoinnissa ja tarkistamisessa. Tyypillisesti tämänkaltaisissa projekteissa ei turvauduta ketteriin menetelmiin juuri tarvittavien formaalien piirteiden ja tarkkaan analysoitujen ja tarkistettujen vaatimusten vuoksi. Mahdollisuus kuitenkin ketterien menetelmien sekä formaalien määrittelymenetelmien yhdistämiseen olisi ja tällä tavalla molempien maailmojen tarjoamien etujen hyödyntäminen projekteissa mahdollista. Wolff (2012) ehdottaa tutkimuksessaan kevyempää lähestymistapaa formaalien menetelmien käyttöön, jolloin kehitysprosessia voitaisiin tehostaa ketterillä menetelmillä, mutta samalla saada ohjelmistoon formaalisti varmennettu toiminnallisuus. Tämänkaltainen kevyempi lähestymistapa tarkoittaisi tiimiä, jossa yhdistyisi sekä perinteisen ohjelmistokehitystavan omaavia kehittäjiä että formaaleja menetelmiä käyttäviä kehittäjiä. Kyseisessä tiimissä turvallisuuden kannalta kriittiset tehtävät voitaisiin toteuttaa formaaleilla menetelmillä ketterässä ympäristössä ja samalla muu tiimi altistettaisiin formaalien menetelmien käytölle, jolloin niiden tunnettavuus tiimin sisällä paranisi tiedonjaon ansiosta.

Valtaosa kehitettävästä ohjelmistosta ei kuitenkaan ole turvallisuuskriittistä, vaan pääosin joko kuluttajien käyttöön suunnattuja tai liiketoiminnan tukemiseen tarkoitettuja sovelluksia. Näiden kehittämisessä ehdottoman oikeellisuuden sijaan tärkeämpää on toimittaa vaatimukset täyttävä sovellus mahdollisimman tehokkaasti, jolloin formaaleille menetelmille ei nähdä tarvetta. Formaalien menetelmien käyttöä tämän päivän ohjelmistokehitysprojekteissa on tutkittu jonkin verran. Meksikossa Fernández-y-Fernándezin (2014) toimesta suoritettu kyselytutkimus osoittaa, että formaaleja menetelmiä ei käytetty juuri ollenkaan kyselyyn osallistuneiden yritysten joukossa. Käytetyimpinä menetelminä olivat juuri yleisimmät ketterät menetelmät, kuten Scrum ja XP.

Päällimmäisenä syynä formaalien menetelmien käytön vähyyteen ketterissä menetelmissä todennäköisesti onkin se, ettei niiden sisällyttäminen jatkuvasti muuttuviin tilanteisiin ole helppoa. Myös korkea oppimiskynnys ja koulutuksen keskittyminen informaalien menetelmien käyttöön vaikuttaa varmasti. Kaiken tämän lisäksi useimmiten ohjelmistojen laatua ennen tavoitellaan käyttötarkoituksen täyttämistä, eikä välttämättä niinkään välitetä, jos ohjelmisto ei olekaan ehdottoman virheetöntä.

Yksi syy formaalien menetelmien käytön vähyyteen ketterissä menetelmissä voi olla myös se, että tyypillisesti tavoitteena on kehittää asiakkaan tarpeita vastaava ohjelmisto

mahdollisimman nopeasti, tehokkaasti ja edullisesti. Oikeilla ketterillä menetelmillä voidaan tuottaa riittävän laadukasta ohjelmistoa useimpien asiakkaiden tarpeisiin, joten formaalien menetelmien käytölle ei nähdä tarvetta. Tähän yhdistettynä aiemmin mainittu formaalien menetelmien korkea oppimiskynnys sekä koulutuksen tarpeesta muodostuvat lisäkustannukset eivät ainakaan kannusta ottamaan formaaleja menetelmiä käyttöön tavallisissa projekteissa.

5.3 Formaalien menetelmien tulevaisuudennäkymät

Formaalien menetelmien käyttö jakaa mielipiteitä kehittäjien keskuudessa. Vaikka niiden tuomista hyödyistä on kiistatonta näyttöä, niin samalla niiden korkea oppimiskynnys ja maine vain asiaan erikoistuneiden insinöörien työkaluna on johtanut siihen, että menetelmiä käytetään vähän, varsinkin pienemmissä kehitysprojekteissa. Jotta formaalien menetelmien käyttö voisi lisääntyä, pitäisi niitä kehittää vieläkin helpommin lähestyttävään muotoon.

Heitmeyer (2005) on myös pohtinut asiaa paperissaan ja hänen mukaansa formaalien menetelmien yleistyminen vaatisi parannuksia neljällä eri sektorilla. Ensimmäinen näistä on formaalien kielten kehittäminen, jonka myötä kielistä tehtäisiin paremmin nykyajan tarpeisiin vastaavia sekä houkuttelevampia. Tämä voitaisiin saavuttaa korostamalla käytettävyyttä graafisten käyttöliittymien avulla, tai kehittämällä uusia kieliä, joiden käyttökohde ja sitä myöden ominaisuudet olisivat rajattu tarkasti. Toinen hänen mainitsemansa kehityskohde on määrittelyiden ja mallien laadun parantaminen. Monessa tapauksessa formaaleilla menetelmillä tehtyjen mallien ja määrittelyiden rakenne saattaa kehittyä turhan monimutkaiseksi ja abstrahoinnin käyttö unohtua. Tämän seurauksena määrittelyistä tulee sekavia ja suuria, jonka lisäksi niihin on sisällytetty hyvin usein toteutukseen liittyviä yksityiskohtia. Yhtenä ratkaisuna tämän ongelman ratkaisemiseen voisi olla koulutus, jonka myötä hyvän ja selkeän määrittelyn rakentuminen selventyisi sekä yleistyisi. Formaalit menetelmät tarjoavat keinon muodostaa toimintansa osalta tarkistettua koodia suoraan määrittelyistä. Monet tällaiset metodit generoivat kuitenkin osittain ongelmallista koodia, jota ei voida käyttää lopullisessa ohjelmistossa. Nämä metodit vaativat siis kehitystä, jolloin voitaisiin tuottaa automaattisesti määrittelyiden avulla tietoturvallista ja tehokasta sekä suoraan käyttökelpoista koodia. Viimeisenä huomiona Heitmeyer mainitsee koodin tarkistukseen käytettävät menetelmät ja niiden kehitystarpeen. Manuaalisesti tuotettua koodia pitäisi pystyä tarkistamaan automaattisesti vaatimuksia vasten ja täten todistamaan, että koodi täyttää sille asetetut vaatimukset.

Yhtenä mahdollisena keinona lisätä formaalien menetelmien käyttöä voitaisiin pitää kevyitä formaaleja menetelmiä (engl. lightweight formal methods), joiden tarkoituksena on

laskea formaalien menetelmien käytöstä muodostuvia kustannuksia jakamalla tehtävää formalisointia pienempiin osiin. Sen sijaan, että formalisointia käytettäisiin kauttaaltaan projektissa, voitaisiin pyrkiä hyödyntämään niitä formaalien menetelmien osia, joiden käytöstä voidaan saada helposti hyötyjä projektin laadun parantamisessa (Jackson & Wing, 1996). Näiden kevyempien menetelmien ympärillä on kuitenkin paljon epäselvyyttä ja selkeän kevyen menetelmän määritelmän puuttuminen vaikeuttaa niiden omaksumista. Zamansky, Spichkova, Rodriguez-Navas, Herrmann ja Blech (2018) ovatkin koostaneet tutkielmassaan kehyksen, jonka avulla sopivan kevyen menetelmän valinnan pitäisi olla helpompaa. He myös toteavat, että selkeästi eniten käytetty kevyempi menetelmä on Alloy-määrittelykieli (Jackson, 2002).

Formaalien menetelmien ympärillä on edelleen tänäkin päivänä tiivis yhteisö ja menetelmiin liittyvää tutkimusta tehdään runsaasti. FME (engl. formal methods europe) -järjestö järjestää vuosittain useitakin konferensseja sekä työpajoja ympäri maailmaa, joiden avulla pyritään edistämään formaalien menetelmien tutkimusta ja käyttöä (FME, 2020). Kokonaan uusia menetelmiä kehitetään myös jonkin verran ja menetelmien käyttöä pyritään lisäämään ketterissä menetelmissä. Yhtenä esimerkkinä voidaan mainita oliopohjaisten järjestelmien määrittelyyn käytettävä rCos (engl. refinement of component and object systems), jonka käytöstä osana ketterää ohjelmistokehitystä Zuo, Yang ja Chen (2010) ovat tehneet esityksen. Siinä rCos-mallinnusta käytettiin järjestelmäanalyysissa ja järjestelmäsuunnittelussa sekä testitapausten luonnissa.

Vaikka formaalien menetelmien parissa pyritään työskentelemään aktiivisesti ja tieteellinen yhteisö sen ympärillä on edelleen voimissaan, ei menetelmien käytön yleistymiselle näy merkittäviä kehitysaskelia. Wayne (2019) kirjoittaakin kattavasti blogissaan formaaleihin menetelmiin liittyvistä ongelmista, joiden vuoksi menetelmien käyttö on edelleen niin vähäistä. Yhtenä merkittävänä kohtana hän mainitsee todistamisen vaikeuden, jos vaatimukseen liittyvä toiminnallisuus halutaan todistaa oikeaksi, vaatii se ohjelmistokehityksessä kaikkien mahdollisten tapahtumien huomioimisen, jonka myötä tehtävistä todistuksista tulee usein monimutkaisia ja haastavia. Hän myös mainitsee, että formaalien menetelmien käytön vähyys on laajalti kulttuurista johtuva ongelma. Formaalien menetelmien ei uskota tuovan niiden tarjoamaa hyötyä, eikä suurin osa alan kehittäjistä ole edes kuullut formaaleista menetelmistä.

5.4 Case-tutkimus: Konecranes

Diplomityö tehtiin osana työn toimeksiantajana toimineen Konecranesin ohjelmistokehitysprojektia, jonka ohessa oli tarkoitus tutkia vaatimusmäärittelyyn liittyviä käytäntöjä

sekä määrittelymenetelmiä ketterässä ohjelmistokehityksessä. Konecranes on kansainvälinen silta- ja satamanostureita valmistava yritys. Valmistettava nosturi joudutaan useissa tapauksissa konfiguroimaan asiakkaan tarpeisiin, jota varten yhtiöllä on ollut käytössä useampi eri tuote- tai myyntikonfiguraattori. Käytetyt konfiguraattorit ovat pääasiassa yhtiön itse kehittämiä, joko osittain tai kokonaan.

Projekti, jonka määrittelytyöhön diplomityössä keskityttiin, oli yhtiön yhden pisimpään käytössä olleen myyntikonfiguraattorin modernisointiprojekti. Projektin tarkoituksena on kehittää uusi järjestelmä korvaamaan vanha, jo yli kaksikymmentä vuotta käytössä ollut järjestelmä. Vanha ohjelmisto kattoi käytännössä koko myyntiprosessin tuotteen konfiguroimisesta aina tarjouksen ja tilauksen tekemiseen asti. Uuden järjestelmän oli tarkoitus jakaa tämä prosessi tuotteen tekniseen konfigurointiin ja myyntitapahtumaan kuuluviin toimintoihin. Teknistä konfigurointia varten kehitettäisiin uusi järjestelmä ja tarjoussekä tilausprosessia hallittaisiin yhtiön ennestään käyttämällä järjestelmällä, jota laajennettaisiin täyttämään siihen kohdistuvat uudet vaatimukset.

Diplomityön puitteissa selvitettiin mitä toimintoja kuhunkin järjestelmään tulisi toteuttaa ja varsinainen määrittelytyö keskittyi myyntiin käytettävän järjestelmän uusien toimintojen toteuttamiseen. Projektin aikana hyödynnettiin monipuolisesti vaatimusmäärittelyn eri menetelmiä, joista erityisen tehokkaaksi havaittiin prototypointi. Prototypoinnin avulla esillesaannin iteraatioiden välissä voitiin havainnollistaa kerättyjä ideoita konkreettisessa muodossa ja näin herättää uusia ideoita sekä todentaa aiempien vaatimusten toimivuus. Muuten vaatimusten esillesaantiin käytettiin haastatteluita, työpajoja sekä vanhan ohjelmiston toimintojen kartoitusta. Vaatimusten analyysia tehtiin usein heti esillesaantivaiheen yhteydessä, esimerkiksi työpajan aikana.

Vaatimusten varsinainen dokumentointi ja määrittelyiden teko toteutettiin luonnollisilla kielillä. Taulukoiden, kirjoitettujen kuvausten sekä havainnollistavien kuvankaappausten avulla määrittelyistä pyrittiin tekemään selkeitä ja johdonmukaisia. Projektin aikana oli tarkoitus käyttää formaaleja menetelmiä jonkin osa-alueen tai toiminnon määrittelyssä, jolloin olisi saatu konkreettinen esimerkki niiden toimivuudesta ketterässä ohjelmistokehityksessä. Suuri osa tehdyistä määrittelyistä kuitenkin pohjautui käyttöliittymään tehtäviin muutoksiin tai lisäyksiin, jolloin formaalien menetelmien käytöllä ei koettu saavutettavan mitään merkittävää lisähyötyä. Myös formaalien menetelmien korkea oppimiskynnys vaikutti käytettävien menetelmien valintaan. Jotta formaalien menetelmien täysi hyöty saataisiin irti, pitäisi näiden käyttö hallita moitteetta. Formaalien menetelmien käyttö olisi myös edellyttänyt osaamista kehitystiimiltä, joka taas olisi vaatinut lisäkoulutusta.

Tehdyt määrittelyt painottuivat siis diplomityön aikana käyttöliittymään tehtäviin muutoksiin, joiden toteuttaminen oli luontevampaa käyttäen informaaleja menetelmiä. Formaalien menetelmien käyttö olisi voinut soveltua paremmin teknisempään määrittelyyn, jolloin ohjelmiston logiikkaa olisi voitu määrittää formaalisti. Projektin luonteen vuoksi tähänkään ei varsinaisesti soveltuvia kohteita löytynyt, sillä tavoitteena oli pyrkiä käyttämään vanhan ohjelmiston komponentteja nostureiden tekniseen laskentaan sellaisenaan, eikä niiden uudelleenkirjoittamiselle ja siten formaalille määrittelylle ollut projektin tässä vaiheessa tarvetta. Formaalit menetelmät voisivat olla omiaan juuri teknistä laskentaa suorittaville moduuleille, jolloin laskenta voitaisiin todentaa formaalisti ja olla varmoja laskennan lopputuloksen oikeellisuudesta.

Projektissa oli siis kyse vanhan ohjelmiston modernisoinnista, jolloin tarve ohjelmiston eri toiminnoille pysyy edelleen samana. Tarkoituksena on vain kehittää uusi järjestelmä, käyttäen uusimpia tekniikoita, joka täyttää kaikki ne samat vaatimukset, joita vanhallekin järjestelmälle on asetettu. Tämänkaltaisissa projekteissa vanhan ohjelmiston rooli uuden ohjelmiston määrittelyssä on merkittävä. Vanha ohjelmisto toimii hyvänä lähtökohtana ja sen avulla voidaan suhteellisen helposti muodostaa käsitys uutta järjestelmää koskevista vaatimuksista ja näin nopeuttaa vaatimusten esillesaantia. Vastaavanlainen tilanne voisi antaa myös mahdollisuuden formaalien menetelmien käyttöön. Vanhasta ohjelmasta voitaisiin muodostaa formaali kuvaus, joka selkeyttäisi yleiskuvaa vanhan järjestelmän tilasta sekä ominaisuuksista. Samalla kukin toiminnallisuus tulisi todennettua ja mahdolliset epäkohdat tulisi havaittua aikaisessa vaiheessa. Formaali kuvaus ja sen myötä saatu käsitys vanhan järjestelmän toiminnoista voisi myös toimia projektia ohjaavana tekijänä ja uuden järjestelmän kehitettäviä osia voitaisiin priorisoida helpommin. Formaalin kuvauksen tuottaminen kokonaisesta järjestelmästä vaatisi kuitenkin valtavat määrät resursseja sekä projektin alkuvaiheeseen sijoittuvaa selvitystyötä.

6. ANALYYSI

Formaalien menetelmien käyttö koodin laadun parantamiseen on aiheena mielenkiintoinen. Ajatus tehtävien määrittelyiden todentamisesta ja todistamisesta ennen varsinaisen toteutuksen aloittamista, vaikuttaen siten tehtävään toteutustyöhön sekä tehokkuuden että laadun näkökulmasta vaikuttaa lupaavalta. Todellisuudessa kuitenkin formaalien menetelmien heikko tunnettavuus ja niiden käyttöön liittyvät ongelmat ovat vaikuttaneet siihen, että niiden käyttö ohjelmistokehityksessä on erittäin vähäistä. Tämä on sinänsä harmillista, sillä formaalien menetelmien käytöstä saatavista hyödyistä on kiistatonta näyttöä.

Yhtenä erittäin hyvänä esimerkkinä voidaan pitää formaaleista määrittelyistä saatavia valmiita testitapauksia. Vaikka kulttuuri testaamisen ympärillä onkin jo kehittynyt parempaan suuntaan ja testaamiseen kiinnitetään enemmän huomiota, kohdellaan silti perusteellisempaa testaamista usein pienemmällä prioriteetilla muuhun kehitykseen verrattuna. Poikkeuksena tähän ovat kuitenkin isommat kehitysprojektit, joissa usein perustetaan erillinen testaustiimi, joka vastaa järjestelmän perusteellisesta testaamisesta. Formaaleilla menetelmillä saatavat valmiit testitapaukset voisivat siis edesauttaa testaamisen määrää ja näin edelleen kehittää ohjelmistojen laatua kohti parempaa.

Nykypäivän ketterät ohjelmistokehitysmenetelmät tuovat omat haasteena formaalien menetelmien käyttöön. Haasteista huolimatta niiden käyttöä ja kehitysprosessin osaksi ottamista voitaisiin hyvin harkita, vaikkakin rajatusti. Koko projektin kehitystä ei tarvitsisi formalisoida, vaan voitaisiin pyrkiä hyödyntämään formaalien menetelmien tuomaa tarkistusta vain toiminnaltaan ja tietoturvaltaan kriittisten osien tai komponenttien kehittämiseen. Näin ketterien menetelmien tuoma mahdollisuus reagoida muuttuviin vaatimuksiin ja nopeus säilyisivät muuttumattomina suurimmassa osassa projektia. Samalla formaalien menetelmien käyttöön liittyvää osaamista ei välttämättä vaadittaisi koko tiimiltä ja tarvittava etukäteen tehtävä dokumentaatio voitaisiin pitää vähäisenä. Näin formaalien menetelmien rajatun käytön myötä muodostuvat lisäkustannukset voitaisiin pitää kohtuullisina ja samalla kuitenkin parantaa kehitettävän ohjelmiston laatua huomattavasti.

Yksi tämänkin työn puitteissakin esille noussut käyttötapaus formaaleille menetelmille voisi olla vanhan ohjelmiston kuvaaminen formaalisti. Erityisesti samanlaiseen toiminnallisuuteen tähtäävässä modernisointiprojektissa tästä voisi olla hyötyä. Tässäkään tapauksessa täydellinen formalisointi ei välttämättä olisi ketteryyden kannalta otollisinta, vaan voitaisiin pyrkiä todentamaan formaalisti vanhasta järjestelmästä ne osat, joiden

oikeellisuudesta olisi eniten hyötyä. Formaalit menetelmät eivät myöskään sovellu samalla tavalla kaikkien järjestelmän osien määrittelyyn vaan eniten hyötyä saadaan, kun esimerkiksi ohjelmiston toiminnan kannalta kriittinen logiikka ja laskenta kuvataan formaalisti. Käyttöliittymän yksityiskohtiin tähtäävään määrittelyyn luonnolliset kielet soveltuvat huomattavasti paremmin.

Formaalien menetelmien käytön yleistymisen merkittävimpänä esteenä on todennäköisesti niiden korkea oppimiskynnys. Menetelmien sisältyminen koulujen tarjoamaan koulutukseen on vähäistä ja saatavilla oleva materiaali ajoittain hankalasti tulkittavaa tai vähäistä. Myös uudemman, relevantin materiaalin löytäminen osoittautui tämän työn puitteissa haastavaksi. Formaalien menetelmien käytön perusteisiin perustuvaa laadukasta materiaalia ei juurikaan ole, vaan menetelmien käyttöä pitäisi opetella vain muutaman esimerkin ja notaatioihin syventyvien oppaiden avulla.

Helposti omaksuttavan materiaalin vähäinen määrä ja erilaisten formaalien kielten valtava kirjo voi johtaa siihen, että formaalien menetelmien käyttöä ei osata edes harkita. Selkeää vertailua eri menetelmien välillä ei ole ja siksi juuri omaan projektiin soveltuvan menetelmän valinta on hankalaa. Jotkin menetelmiin perehtyneet asiantuntijat tarjoavat apuaan konsultaation muodossa, jolloin formaalien menetelmien käyttöönottoa projektissa voitaisiin nopeuttaa. Todennäköisesti tällaisten asiantuntijoiden kohdalla puhutaan kuitenkin määrällisesti vain kourallisesta eri henkilöitä ympäri maailman, joten esimerkiksi Suomessa toteutettavan pienemmän projektin kohdalla ei ole edes realistista tavoitella heidän palveluitaan. Tehokas konsultointi vaatisi paikan päällä olemista ja täten kustannukset voisivat nousta kynnyskysymykseksi.

Tällä hetkellä ketterissä menetelmissä käytettävien luonnollisten kielten asema näyttää perustellulta. Huolella tehty vaatimusmäärittely, käytettävästä formaaliuden tasosta riippumatta, auttaa kehittämään useissa tapauksissa riittävän laadukasta ja toiminnaltaan luotettavaa koodia. Formaalien menetelmien käyttöä ei tällä hetkellä pidetä millään tapaa välttämättömänä ja usein käytettävyyden ja luotettavuuden osalta riittäviä tuloksia saadaan aikaan ilmankin formaaleja menetelmiä. Niin kauan kuin formaalien menetelmien käyttöön ja oppimiseen liittyy ongelmia, ei voida odottaa niiden nopeaa yleistymistä.

7. YHTEENVETO

Tämän diplomityön aiheena käsiteltiin ohjelmistokehitykseen ja erityisesti nykypäivänä vakiintuneeseen asemaan nousseiden ketterien menetelmien yhteydessä käytettäviä määrittelymenetelmiä. Määrittelymenetelmillä tarkoitetaan vaatimusten dokumentointiin ja kirjaamiseen käytettäviä tekniikoita, jotka voidaan jakaa karkeasti kolmeen kategoriaan. Näihin lukeutuvat, tämänkin työn puitteissa esitellyt formaalit menetelmät, puoliformaalit sekä informaalit menetelmät. Formaalit menetelmät ovat matemaattista syntaksia ja loogista päättelyä sisältäviä menetelmiä, joiden tarkoituksena on edesauttaa määrittelyiden oikeellisuutta ja tehostaa virheiden havainnointia aikaisessa vaiheessa. Puoliformaaleilla menetelmillä tarkoitetaan menetelmiä, jotka sisältävät jonkin tasoisia formaaleja piirteitä, mutta eivät kuulu virallisten formaalien menetelmien joukkoon. Yhtenä tällaisena menetelmänä voidaan pitää tässä työssä käsiteltyä UML-mallinnuskieltä. Informaaleilla menetelmillä taas tarkoitetaan näiden joukkojen ulkopuolelle jääviä menetelmiä, eli käytännössä luonnollisten kielten käyttöä määrittelyiden tekemiseen.

Työn tekeminen osana Konecranesin ohjelmistoprojektia mahdollisti havainnoinnin oikean kehitysprojektin puitteissa. Ketterillä menetelmillä toteutettu projekti toimi hyvänä pohjana vaatimusmäärittelyiden käytänteitä tutkittaessa ja tarjosi erinomaisen ympäristön vaatimusmäärittelyyn kuuluvien käytäntöjen tarkasteluun. Työn tavoitteena olikin selvittää mitä käytäntöjä ja menetelmiä nykypäivänä käytetään sekä tutustua formaalien menetelmien käyttöön ja asemaan tämän päivän ketterissä ohjelmistoprojekteissa.

Työ koostui kuudesta eri luvusta. Johdannon jälkeisessä, toisessa luvussa tarkoitus oli antaa lukijalle yleistason käsitys ohjelmistokehityksestä sekä tähän kuuluvasta ketterästä lähestymistavasta. Ohjelmistokehitysmenetelmistä tutustuttiin perinteikkääseen vesiputousmalliin sekä ketteriin menetelmiin kuuluviin Scrumiin ja Kanbaniin. Seuraavassa luvussa perehdyttiin tarkemmin vaatimusmäärittelyn eri vaiheisiin ja käytäntöihin. Työssä esiteltiin vaatimusmäärittely viisivaiheisena prosessina vaatimusten esillesaannista, analyysin kautta dokumentointiin ja validointiin sekä lopulta vaatimustenhallintaan. Prosessin lisäksi työssä tutustuttiin myös vaatimusmäärittelyä ohjaavien standardien nykytilaan sekä ketteryyden tuomiin erityispiirteisiin ja vaikutuksiin. Työn neljännessä luvussa tutustuttiin syvemmin vaatimusmäärittelyssä käytettäviin määrittelymenetelmiin. Luku on jaettu kolmeen osaan käyttäen menetelmille ominaista jakoa formaalien-, puoliformaalien- ja informaalien menetelmien kesken. Formaaleja menetelmiä käsittelevän osan tarkoituksena oli perehtyä tarkemmin menetelmille ominaisiin piirteisiin sekä tutustua tarkemmin konkreettisten esimerkkien avulla kahteen formaaliin määrittelykieleen; Z-

notaatioon ja Vienna development methodiin. Puoliformaaleista menetelmistä työssä otettiin esille UML-kielen käyttö määrittelyiden tekemisessä. Lukijalle pyrittiin tarjoamaan peruskäsitys kielen tarjoamista kaavioista sekä näiden soveltuvuudesta vaatimusmäärittelyn työkaluna. Viimeisenä luvussa tutustutaan kaikkein yleisimmin käytettyihin informaaleihin menetelmiin, eli luonnollisiin kieliin. Viidennessä luvussa perehdytään tarkemmin ketterään lähestymistapaan formaaleissa menetelmissä. Tavoitteena on muodostaa käsitys formaalien menetelmien nykytilasta ja käyttöasteesta sekä soveltuvuudesta ketteriä menetelmiä hyödyntäviin projekteihin. Kuudennessa luvussa työn tuloksena saatuja havaintoja analysoitiin ja pohdittiin tarkemmin.

Tutkimus toteutettiin pääasiassa kirjallisuusselvityksen avulla, jonka tueksi pyrittiin tuomaan omia havaintoja diplomityön toimeksiantajan Konecranesin ketterillä menetelmillä toteutetun ohjelmistoprojektin aikana tehdystä vaatimusmäärittelystä. Kirjallisuusselvityksessä pyrittiin keskittymään tuoreimpien tutkimusten antamiin havaintoihin.

Vaikka formaaleja menetelmiä on käytetty ja kehitetty jo yli puoli vuosisataa, eivät ne ole missään vaiheessa yleistyneet valtavirran käyttämiksi työkaluiksi. Tuoreimpien tutkimusten ja asiantuntijoiden mielipiteiden mukaan informaalit menetelmät ovat edelleen johtavassa asemassa, varsinkin ketterillä menetelmillä ohjelmistoja kehitettäessä. Formaalien menetelmien käytöstä onnistuneesti ketterissä menetelmissä on vain hieman näyttöä. Yhtenä ratkaisuna formaalien menetelmien käytön lisäämiseen on ehdotettu formaalien menetelmien kevennettyä käyttötapaa, jossa kehitystiimi koostuisi sekä formaaleja menetelmiä taitavista kehittäjistä että perinteisiä menetelmiä käyttävistä kehittäjistä. Tällaisella tiimillä voitaisiin hyödyntää formaaleja menetelmiä turvallisuuskriittisten osien kehittämisessä ja samalla lisätä tietämystä formaaleista menetelmistä niistä vähemmän tietäville tiimijäsenille.

Konecranesin ohjelmistoprojekti antoi mahdollisuuden todeta käytännössä, millaisia menetelmiä ketterässä ohjelmistokehityksessä käytetään. Tarkoituksena oli alun perin soveltaa formaaleja menetelmiä jonkin osa-alueen määrittelyssä, mutta sopivaa määrittelykohdetta oli kuitenkin vaikea löytää. Perinteiset luonnollisilla kielillä toteutetut määrittelyt ovat helpommin lähestyttäviä sekä ymmärrettäviä, eikä niitä varten vaadita mitään erityisosaamista. Tästäkin syystä on ymmärrettävää, että jos ei formaaleille menetelmille ole projektin luonteen puolesta ehdotonta tarvetta, käännytään helpommin luonnollisten kielten käyttöön. Projektissa merkittävässä roolissa toimi vanha ohjelmisto, jonka uudistamiseen kehitettävällä järjestelmällä tähdättiin. Vanhan ohjelmiston olemassaolo merkitsi sitä, että tulevan järjestelmän vaatimusten raamit olivat selvät jo alusta alkaen, pitihän sen korvata vanha järjestelmä. Myös vaatimusten selvittämiseen tämä toi tietynlaista etua, sillä uuteen järjestelmään osoitettavia vaatimuksia voitiin johtaa suoraan vanhasta

ohjelmistosta ja sen käyttötapauksista. Yhtenä mahdollisena formaalien menetelmien käyttökohteena olisikin voinut olla vanhan ohjelmiston formaali kuvaus, jonka avulla järjestelmän toiminnoista olisi saatu kattava käsitys ja samalla toiminnallisuus olisi tullut todennettua formaalisti. Tämänkaltainen prosessi olisi kylläkin vaatinut valtavasti resursseja ja etukäteen suoritettavaa työtä, joka sotii ketterillä menetelmillä toteutettavan projektin perusajatusta vastaan.

Formaalien menetelmien ympärillä toimii edelleen aktiivinen asiantuntijoiden yhteisö, jonka pyrkimyksenä on kehittää menetelmiä sekä edesauttaa menetelmien laajempaa käyttöönottoa. Tästäkin huolimatta on erittäin todennäköistä, että formaalien menetelmien käyttö ei tule lisääntymään lähiaikoina merkittävästi, vaan luonnollisten kielten asema ketterien menetelmien määrittelykielenä pysyy vakaana.

Formaalien menetelmien saralla olisi varmasti runsaasti tutkittavaa ja menetelmien tuominen mukaan ketterään ohjelmistokehitykseen voisi ratkaista moniakin ohjelmistojen laatuun liittyviä ongelmia. Tämä kuitenkin vaatisi ehkä kokonaan uuden kielen kehittämistä, joka ratkaisisi kaikki formaaleja menetelmiä piinaavat ongelmat. Jotta menetelmien käyttö yleistyisi, pitäisi tarjolla olla helposti lähestyttävä, pienemmän oppimiskynnyksen omaava ja laadukkaiden ohjelmistojen tukema formaali menetelmä.

Formaaleihin menetelmiin liittyvää tutkimusta olisi hyvä edelleen jatkaa. Jatkotutkimuksena formaalien menetelmien nykytilanteeseen liittyen voitaisiin pyrkiä selvittämään, mikä on formaalien menetelmien käyttöaste nykypäivänä. Tutkimuksessa voitaisiin selvittää, minkälaisia projekteja formaaleilla menetelmillä toteutetaan ja millä perusteilla formaalit menetelmät on valittu.

LÄHTEET

Agile Alliance. (2001), Agile manifesto (viitattu 2.9.2019): http://agilemanifesto.org/

Ahmad, S. (2008), Negotiation in the requirements elicitation and analysis process, IEEE

Alagar, V. S. & Periyasamy, K. (2011), Specification of software systems, Springer

Ali Babar, M. & Brown, A. W. & Mistrik, I. (2013), Agile software architecture: Aligning agile processes and software architectures, Elsevier

Allbee, B. (2018), Hands-on software engineering with Python, Packt publishing

Batra, M. & Malik, A. & Dave, M. (2013), Formal methods: benefits, challenges and future direction, JGRCS

Boehm, B. W. & Papaccio, P. N. (1988), Understanding and controlling software costs, IEEE

Booch, G. & Jacobson, I & Rumbaugh, J (2005), Unified modelling language user guide, Addison Wesley

Cao, L. & Ramesh, B. (2008), Agile requirements engineering practices: An empirical study, IEEE

Chakraborty, A. & Baowaly, M. K. & Arefin, A. & Bahar, A. N. (2012), The role of requirement engineering in software development life cycle, CIS Journal

Community Z Tool Project (2003), CZT:Community Z Tools, saatavissa (viitattu 18.12.2019): http://czt.sourceforge.net/manual.html

Cooke, J. L. (2012), Everything you want to know about Agile: How to get Agile results in a less-than-agile organization, IT Governance

Despa, M. L. (2014), Comparative study on software development methodologies, Database systems journal

Eclipse foundation (2019), Eclipse IDE, saatavissa (viitattu 18.12.2019): https://www.eclipse.org/eclipseide/

Fitzgerald, J. & Larsen, P. G. & Mukherjee, P. & Plat, N. & Verhoef, M. (2005), Validated designs for object-oriented systems, Springer

Fernández-y-Fernández, C. A. (2014), Integrating formal methods into traditional practices for software development: an overview, Arxiv

FME (2020a), About FME, saatavissa (viitattu 13.4.2020): http://www.fmeurope.org/about/

FME (2020b), Upcoming FM Conferences, saatavissa (viitattu 13.4.2020): http://www.fmeurope.org/feature/upcoming_conferences/

Hall, A. (2008), Z Word Tools, saatavissa (viitattu 18.12.2019): https://sourceforge.net/projects/zwordtools/

Heitmeyer, C. L. (2005), Developing high quality software with formal methods: what else is needed?, ResearchGate

Holcombe, W. M. L. (2008), Running an Agile Software Development Project, Wiley

Holt, J. & Perry, S. A. & Brownsword, M. (2012), Model-based requirements engineering, The Institute of engineering and technology

Hull, E. & Jackson, K. & Dick, J. (2017), Requirements engineering 2nd edition, Springer

IEEE (2018), IEEE 29148-2018 – Systems and software engineering – life cycle processes – Requirements engineering, IEEE

IEEE (2019a), IEEE at a glance, saatavissa (viitattu 13.10.2019): https://www.ieee.org/about/today/at-a-glance.html

IEEE (2019b), IEEE 830-1998 Recommended Practice for Software Requirements Specifications, saatavissa (viitattu 13.10.2019): https://ieeexplore.ieee.org/document/720574

IEEE (2019c), IEEE 29148-2018 International Standard – Systems and software engineering -- Life cycle processes -- Requirements engineering, saatavissa (viitattu 13.10.2019): https://ieeexplore.ieee.org/document/8559686

IEEE (2019d), IEEE 1233-1998 Guide for developing system requirements specifications, saatavissa (viitattu 13.10.2019): https://ieeexplore.ieee.org/document/741940

IEEE (2019e), IEEE 1362-1998 Guide for Information Technology – System Definition - Concept of Operations (ConOps) Document, saatavissa (viitattu 13.10.2019): https://ieeexplore.ieee.org/document/761853

ISO/IEC. (2002), ISO/IEC 13568

ISO/IEC. (1996), ISO/IEC 13817-1

ISO/IECa. (2012), ISO/IEC 19505-1

ISO/IECb. (2012), ISO/IEC 19505-2

Jackson, D. & Wing, J. (1996), Lightweight formal methods, Springer

King, S. & Sørensen, I. H. & Woodcock, J. (1988), Z: Grammar and concrete and abstract syntaxes version 2.0, IBM United Kingdom Laboratories Ltd.

Knight, J. C. (1998), Challenges in the Utilization of Formal Methods, Springer

Laplante, P. A. (2004), Real-time systems design and analysis 3rd edition, IEEE

Leon, A. (2015), Software configuration management handbook 3rd edition, Artech house

Lines, K. J. (1995), VDM-SL LaTeX macros, saatavissa (viitattu 25.2.2020): ftp://ftp.npl.co.uk/pub/latex/macros/vdm-sl/

Luisa M. & Mariangle F. & Pierluigi N. I. (2004), Market research for requirements analysis using linquistic tools, Springer

Löwe, M. (2010), Formal methods in agile development, ECEASST

Overture (2018), VDM-10 Language manual, saatavissa (viitattu 3.1.2020): https://raw.githubusercontent.com/overturetool/documentation/master/documentation/VDM10LangMan/VDM10_lang_man.pdf

Overture (2020), Overture tool – overview, saatavissa (viitattu 24.2.2020): http://overturetool.org/

Pender, T. (2003), UML Bible, Wiley

Pfleeger, S. L. & Hatton, L. (1997), Do formal methods improve code quality?, IEEE

Pilone, D. & Pitman, N. (2005), UML 2.0 in a Nutshell, O'Reilly Media Inc

Sommerville, I. (2016), Software engineering 10th edition, Pearson

Sommerville, I. (2020), Natural language requirements, saatavissa (viitattu 25.2.2020): https://iansommerville.com/software-engineering-book/web/natural-language/

Shalloway, A. (2010), The real differences between Kanban and Scrum, saatavissa (viitattu 24.11.2019): https://www.netobjectives.net/blogs/real-differences-between-kanban-and-scrum

Spivey, J. M. (1988), Understanding Z: A specification language and its formal semantics, Cambridge U. P

Spivey, J. M. (1992), The Z-notation: A Reference Manual, Prentice Hall International (UK) Ltd

Uml-diagrams. (2020), UML Profile diagrams, saatavissa (viitattu 3.2.2020): https://www.uml-diagrams.org/profile-diagrams.html

Wayne, H. (2019), Why don’t people use formal methods?, saatavissa (viitattu 13.4.2020): https://www.hillelwayne.com/post/why-dont-people-use-formal-methods

Wiegers, K. E. & Beatty, J. (2013), Software requirements 3rd edition, Microsoft Press

Weilkiens, T. & Oestereich, B. (2006), UML 2 Certification guide: Fundamental and intermediate exams, Elsevier science & technology

Wolff, S. (2012), Scrum goes formal: agile methods for safety-critical systems, IEEE

Royce, W. (1970), Managing the development of large software systems, TRW

Zamansky, A. & Spichkova, M. & Rodriquez-Navas, G. & Herrmann, P. & Blech, J. O. (2018), Towards classification of lightweight formal methods, Arxiv

Zuo, A. & Yang, J. & Chen, X. (2010), Research of agile software development based on formal methods, IEEE