Enemmän kuin ohjelmointia: laskennasta käyttöä huomioiviin arkkitehtuureihin

Peter Larsson
49547
Pro gradu -tutkielma
Tulevaisuuden teknologioiden laitos
Kevät 2018

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

Ohjelmointi on ymmärtämistä: Tietojärjestelmän kehittäminen ei käsitä pelkästään työn suorittavan ohjelman luomista. On keskeistä, että ohjelmankehitys on johtanut sovellusalueen syvälliseen ymmärtämiseen; muuten tietojärjestelmä tuskin soveltuu kohdeorganisaation tarpeisiin. Järjestelmän kehityksen aikana on tärkeää, että sovellusalueen tulkinnasta käydään keskustelua kehittäjien ja organisaation välillä. (korostukset kuten alkuperäisessä tekstissä; Lehrmann Madsen ym. 1993, 3)

…ihmiskeskeisestä suunnittelusta puuttuu formalismi, joten se ei pysty järjestelmällisesti luomaan yhteyksiä käyttäjien tarpeiden ja ohjelman rakenteen välillä. (Denning & Dargan 1996, 112)

TURUN YLIOPISTO

Tulevaisuuden teknologioiden laitos

LARSSON, PETER: Enemmän kuin ohjelmointia: laskennasta käyttöä huomioiviin arkkitehtuureihin

Pro gradu -tutkielma, 87 s.
Tietojärjestelmätiede
Kevät 2018

Millainen ohjelma luodaan, on ohjelmankehityksen haastavimpia kysymyksiä. Uuden ohjelman suunnittelussa on huomioitava, millaiseksi organisaation tuleva toiminta halutaan. Jos tietotekniikalta tarvittava tuki voidaan määritellä yksiselitteisesti, voi ohjelmankehitys noudattaa rationaalista prosessia. Kuitenkin uudenlaisessa toiminnassa sekä toiminta että tarvittava tietotekniikka ovat tuntemattomia. Ongelman määritys syntyy mahdollisista ratkaisuista ja ratkaisut puolestaan ongelman määrityksestä. Ohjelman suunnittelu vaatii asiakkaan ja kehittäjän yhteistyötä.

Tutkielman lähtökohtana on ajatus, että ohjelmointi riittävän korkealla abstraktiotasolla mahdollistaa asiakkaan osallistumisen ohjelman suunnitteluun. Suunnittelulla tarkoitetaan tässä ohjelman toiminnallisuuden ja käytön muotoilua. Riittävän korkealla abstraktiotasolla viitataan ohjelmointimenetelmän käsitteisiin. Käsitteiden on oltava sellaisia, että asiakas voi kokemuksensa perusteella niitä käsitellä. Tutkielman tavoitteena on luoda viitekehys, joka määrittelee riittävän korkean abstraktiotason.

Viitekehys perustuu Markku I. Nurmisen (1988) tietotekniikan näkökulmiin. Näkökulmat kuvaavat kolmea eri käsitystä tietojärjestelmistä. Ne eroavat tavassa miten työntekijä voi vaikuttaa työhönsä. Yksi vaikuttamisen kohde on millainen tietojärjestelmä luodaan. Näkökulmia käytetään yhdistämään työntekijän vaikuttaminen ja ohjelmankehitys. Tuloksena on ohjelmankehitysprosesseja, jotka eroavat kehityksen kohteeltaan. Erot tulkitaan eri abstraktiotasoiksi: teknisestä toimintaa käsittelevään. Ohjelmankehitysprosessien perusteella luodaan mallit niitä tukevien ohjelmointimenetelmien piirteistä. Mallit muodostavat tutkielman viitekehyksen.

Viitekehys määrittelee ohjelmoinnin abstraktiotason, jossa asiakas voi osallistua ohjelmankehitykseen. Esimerkkejä viitekehyksen mukaisista ohjelmointimenetelmistä on löydetty neljästä suosituimmasta ohjelmointiparadigmasta.

Avainsanat: ohjelmistoarkkitehtuuri, ohjelmistokehitys, osallistava suunnittelu, tietojärjestelmät

UNIVERSITY OF TURKU
Department of Future Technologies

LARSSON, PETER: Beyond programming: definition of programming methods that support participatory design

Master's Thesis, 87 p.
Information Systems Science
Spring 2018

To decide what kind of software is needed is one of the hardest questions in software development. One must anticipate the consequences of the new software to ensure that they are aligned with the organizational goals. The software development process can proceed rationally if the requirements can be defined unambiguously. However, if the activity to be supported is new, then both the requirements and the software solution are unknown. The requirements are defined by the solution and the solution by the requirements. The client and the developer need to co-design both the activity and the software support.

The hypothesis of this thesis is that it is possible to program on a level of abstraction that allows the client to participate in creating the software design. Design refers here to the functional features and the user experience. The level of abstraction relates to the vocabulary used when creating the design. The vocabulary has to be such that the client can use it based on prior experience. The goal of this thesis is to develop a framework that defines such a level of abstraction.

The framework is based on three perspectives on information systems described by Markku I. Nurminen (1988). The perspectives differ in the way the employees can influence their working conditions. One area of influence is the software to be developed. By using the perspectives to combine the employees' influence and software development, different kinds of software development processes can be recognized. The key difference is the target of the development, which can be interpreted as representing a level of abstraction. Levels of abstraction range from technical to organizational. The nature of the software development processes is used to define models of programming methods. The models and the associated levels of abstraction form the framework of this thesis.

The framework defines an abstraction level of software development where the client can participate. Examples of the framework's models of programming methods have been found in four of the most popular programming paradigms.

Keywords: information systems, participatory design, software architecture, software development

SISÄLTÖ

1 Johdanto 1
2 Tutkielman menetelmä 4
2.1 Soft Systems Metodologia (SSM) 5
2.2 SSM:n työkalut 9
2.3 SSM ja tietotekniikka 11
2.4 SSM:n soveltaminen tutkielmassa 18
3 Ohjelmoinnin viitekehys 20
3.1 Ohjelmankehitysprosessien lähtökohtien aiheuttamat erot 21
3.2 Kolme näkökulmaa tietotekniikkaan 26
3.3 Ohjelmaa kehittävien toimintajärjestelmien ydinmääritelmät 30
3.4 Kolme ohjelmoinnin mallia 33
4 Viitekehyksen mukaiset ohjelmointimenetelmät 38
4.1 Ohjelmointi laskennan toteuttamisena -malli 40
4.2 Ohjelmointi todellisuuden mallintamisena -malli 47
4.3 Ohjelmointi tietokoneen käytön mahdollistamisena -malli 51
4.4 Ohjelmoinnin viitekehyksen toteutuminen 55
5 Enemmän kuin ohjelmointia 57
5.1 Enemmän kuin ohjelmointikieliä 59
5.2 Enemmän kuin ohjelmointia 61
5.3 Ohjelmistoarkkitehtuurit 64
5.4 Kirjallisuuden tuki ohjelmoinnin viitekehykselle 67
6 Yhteenveto 71
Lähteet 72

1 JOHDANTO

Tietokoneohjelman kehittämisen haasteet voidaan jakaa ohjelmoinnin luonteeseen ja tekniikkaan[1] (Brooks 1986). Ongelman ratkaisuperiaatteen määritteleminen ja ratkaisun toteutuksen päättäminen käsitteellisellä tasolla kuuluvat ohjelmoinnin luonteeseen. Ratkaisuperiaate on ongelman ratkaisun kuvaamista ohjelman suorituksen aiheuttamana muutoksena syötteestä tulokseksi. Muutoksen aikaansaava toimenpiteiden ketju, algoritmi, kuvaa toteutusta käsitteellisellä tasolla. Menetelmät, välineet ja näiden hyödyntäminen ovat teknisiä kysymyksiä. Menetelmät ovat tapoja valita ja käyttää välineitä ratkaisun toteuttamiseksi. Tietokoneohjelman kehittämisessä ohjelmoinnin luonne viittaa kysymykseen mitä, kun taas tekniikka viittaa kysymykseen miten.

Merkittävät parannukset ohjelmoinnin tuottavuuteen ovat olleet seurausta tekniikan kehityksestä (Brooks 1995). Esimerkkejä tällaisesta kehityksestä ovat mm. korkean tason ohjelmointikielet, debuggerit, kehitysympäristöt ja kehitysmenetelmät. Samansuuruisen tuottavuuden kasvun saavuttaminen kehittämällä uusia tekniikoita on kuitenkin vaikeaa (ibid.). Keskeisiin ohjelmankehityksen osa-alueisiin on jo olemassa ratkaisu. Uudet ratkaisut ovat korvaavia tai olemassa olevien muunnelmia, joten niiden tarjoama lisäetu on yleensä pieni verrattuna tekniikan alkuperäiseen hyötyyn. Ohjelmoinnin luonteeseen liittyvät ongelmat tarjoavat paremman mahdollisuuden tuottavuuden merkittävään kasvuun (Brooks 1986). Niiden ratkaisut tulevat osaksi ohjelmankehityksen tekniikoita.

Vaikeimpia ohjelmoinnin luonteeseen liittyviä ongelmia on päättäminen siitä millainen ohjelma luodaan (Brooks 1986). Ohjelmat ovat asiakkaalle työvälineitä tai työympäristöjä (Floyd 1988). Niiden merkitys tulee käytöstä tai toiminnasta, jonka ne mahdollistavat. Mahdolliset hyödyt ovat seurausta tiedonkäsittelyn automatisoinnista, tiedonjakelun tehostamisesta tai toiminnan muutoksista (Mooney ym. 1996). Hyötyjen toteutuminen voi johtua yhdestä tekijästä tai useasta tekijästä samanaikaisesti. Tietotekniikan käyttöönotto ei kuitenkaan automaattisesti tuo hyötyjä. Ilmiötä on nimetty tuottavuusparadoksiksi (Brynjolfson & Yang 1996), jossa tietotekniikan hankinnan vaikutukset eivät

---

[1] Tekniikka -käsitettä käytetään tässä työssä merkityksessä: menetelmät, välineet ja niiden käytön osaaminen (Mot Tietotekniikan Liiton Atk-sanakirja).

näy tilastollisesti yritysten tuottavuuden kasvuna.

Tuottavuusparadoksin ratkaisu löytyi tietotekniikkainvestointeja täydentävistä toimenpiteistä (Brynjolfson & Hit 2002). Toimenpiteitä olivat liiketoimintaprosessien kehittäminen, organisaatiorakenteen muuttaminen sekä innovaatiot asiakas- ja toimittajasuhteissa. Hyödyn saaminen tietotekniikan kehittämisestä vaatii tietotekniikan, organisaation ja näiden vuorovaikutuksen näkökulmien yhdistämistä (Mooney ym. 1996). Eri näkökulmien yhdistäminen onnistuu parhaiten, jos niitä kehitetään yhdessä (Bratteteig ym. 2013). Organisaation kehittäminen on kuitenkin nykyisen ohjelmankehityksen ulkopuolella (Reijonen 2003, Nurminen & Forsman 1994). Asiakas määrittelee vaatimukset ja arvioi tuloksen (Jacobson ym. 2013), mutta hän ei ole mukana ohjelman suunnittelussa (design).

Tutkielman tavoitteena on mahdollistaa asiakkaan osallistuminen ohjelman suunnitteluun. Lähtökohtana on James Cogginssin (1996) esittämä idea, että ohjelmointi riittävän korkealla abstraktiotasolla mahdollistaa asiakkaan osallistumisen. Idean esityksessä ei kuitenkaan kerrota mikä on sopiva abstraktiotaso. Tutkielman tutkimuskysymyksenä on: Mikä on sopiva ohjelmoinnin abstraktiotaso, jotta asiakas voi osallistua ohjelmankehitykseen? Tutkimusmetodi on konstruktiivinen (Hevner 2004). Tutkimuskysymykseen vastataan luomalla viitekehys, joka määrittelee sopivan abstraktiotason. Oletuksena on, että löytyy myös viitekehyksen mukaisia ohjelmointimenetelmiä.

Tutkielma koostuu kolmesta osasta. Ensimmäisessä osassa esitellään käytetty kehitysmenetelmä ja sen avulla luotu viitekehys. Menetelmä on alun perin tarkoitettu organisaatioiden toiminnan kehittämiseen. Koetut ongelmat pyritään ratkaisemaan luomalla toiminnan malleja, joissa ongelmat eivät esiinny. Malleja käytetään keskusteluvälineinä käytännön toimenpiteitä suunniteltaessa. Tutkielmassa kehitysmenetelmää sovelletaan ohjelmankehitysprosessien vertaamiseen. Tavoitteena on luoda viitekehys, jonka avulla voidaan tunnistaa asiakkaan osallistumista tukevia ohjelmointimenetelmiä. Ohjelmankehitysprosessien erot asiakkaan huomioimisessa antavat tähän tarvittavat kriteerit.

Tutkielmassa käytetään kirjallisuudesta löydetyn kolmen näkökulman mukaisia ohjel-

mankehitysprosesseja. Ohjelmankehitysprosessien kuvaukset ovat abstrakteja. Keskeistä niissä on asiakkaan osallistuminen ja ohjelmankehityksellä tavoiteltava muutos. Kolme näkökulmaa luokittelee ohjelmankehitysprosessien muutokset teknisestä asiakkaan tulevaa toimintaa kehittävään. Muutosta käytetään perusteena prosesseihin sopivien ohjelmointimenetelmien mallien luomiseen. Mallien luonnetta kuvataan ohjelmoinnin teorioilla. Kolme mallia muodostaa viitekehyksen, joka tuo esille ohjelmointimenetelmien eroja asiakkaan osallistumisen tuessa.

Toisessa osassa haetaan esimerkkejä viitekehyksen ohjelmoinnin malleja edustavista menetelmistä. Tutkimusoletuksena on, että jokaiselle mallille löytyy sopivia menetelmiä. Ohjelmoinnin teoriat konkretisoivat viitekehyksen mallien sisältämän käsityksen ohjelman ja todellisuuden suhteesta. Ne kertovat mitä asioita huomioidaan ohjelmoinnissa. Teorioita käytetään ohjelmoinnin malleja vastaavien ohjelmointimenetelmien tunnistamiseen. Käyttämällä käsitteitä, jotka kuvaavat asiakkaalle merkityksellisiä asioita, voi asiakas osallistua ohjelman suunnitteluun. Tutkielman idean mukaan ohjelmointimenetelmässä on oltava käsitteille konkreettinen representaatio ohjelman lähdekoodissa. Ajatuksena on, että jaettu käsitteistö edistää yhteistyötä.

Kolmannessa osassa haetaan kirjallisuudesta tukea viitekehyksessä esitettyyn käsitykseen ohjelmankehityksestä. Sopivan kirjallisuuden löytämiseksi hyödynnetään viitekehyksen teoreettista taustaa, jolloin voidaan verrata omia johtopäätöksiä muiden tekemiin. Lisäksi haetaan tekniikoita, joilla olisi vastaavanlaiset tavoitteet viitekehyksen menetelmien kanssa. Lopuksi pohditaan viitekehyksen teorioiden ja menetelmien jatkotutkimuksen mahdollisuuksia.

2 TUTKIELMAN MENETELMÄ

Soft Systems Metodologia[2] (jatkossa SSM; engl. Soft Systems Methodology) sai alkunsa tutkimusohjelmasta, jossa järjestelmätekniikkaa (engl. systems engineering) pyrittiin soveltamaan organisaation kehittämiseen (Checkland & Scholes 1990). Järjestelmätekniikka soveltui kuitenkin huonosti usein monimutkaisiksi ja epämääräisiksi osoittautuvien ongelmien käsittelyyn. SSM:n kehitys alkoi tästä ristiriidasta tutkimuskohteen ja menetelmän välillä. Ristiriidan ratkaisemiseksi jouduttiin muuttamaan tapaa millä järjestelmäkäsitettä hyödynnettiin.

SSM edustaa nk. pehmeää järjestelmäajattelua, jonka mukaan järjestelmäkäsitettä voidaan käyttää apuvälineenä ongelmalliseksi koettujen tilanteiden kehittämiseen (Checkland & Scholes 1990). Pehmeän järjestelmäajattelun mukaan ihmisen toiminnassa sekä ongelman määritelmät että mahdolliset ratkaisut ovat moninaisia. Yhteistä ihmisten toiminnalle eri konteksteissa on pyrkimys johonkin tavoitteeseen. SSM:ssa otettiin ihmisten tavoitteelliset toimintajärjestelmät todellisuuden tarkastelun apuvälineeksi (ibid.). Abstraktilla toimintajärjestelmän käsitteellä haluttiin tehdä ero konkreettiseen tekemiseen. Yksi seuraus toimintajärjestelmä -käsitteen käytöstä oli havainto, että sen määritelmä oli riippuvainen käytetystä näkökulmasta[3].

SSM tukee eri näkökulmien käyttämistä saman toimintajärjestelmän tulkintaan. Tut-

[2]Tutkielmassa käytetään englanninkielisestä nimestä Soft Systems Methodology suomenkielistä muotoa Soft Systems Metodologia. Tulevaisuuden tutkimuksessa on Soft Systems Methodology -nimi käännetty Pehmeäksi Systeemimetodologiaksi (Rubin 2002). Pehmeän Systeemimetodologian kuvaus, on ristiriidassa alkuperäisen käsityksen kanssa. Alkuperäisen Soft Systems Metodologian määritelmän mukaan todellisuudessa ei ole järjestelmiä (Checkland & Poulter 2006), mutta Pehmeän systeemimetodologian kuvauksessa kuitenkin toistuvasti viitataan todellisuudessa oleviin järjestelmiin (tai systeemeihin). Tutkielmassa noudatetaan alkuperäistä Soft Systems Metodologian tulkintaa ja sekaannuksen välttämiseksi ei käytetä Pehmeän systeemimetodologian käsitettä.

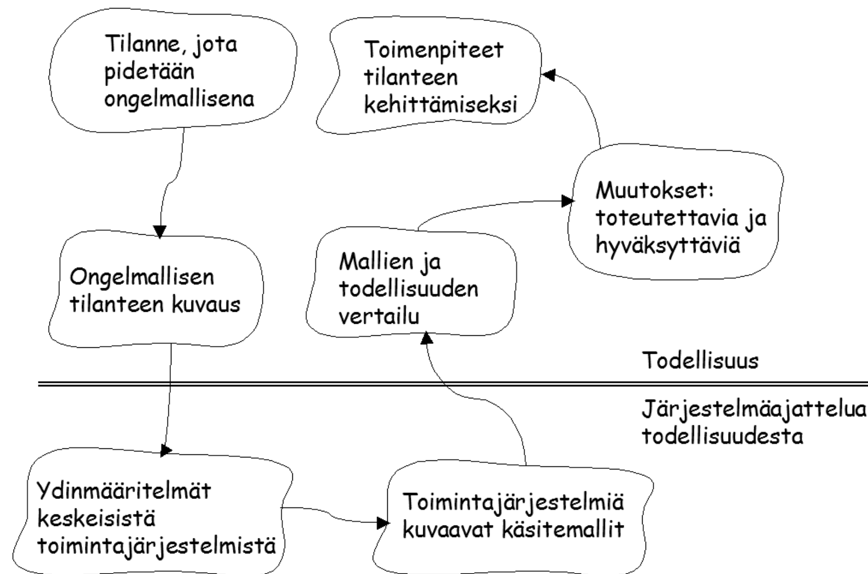
[3] Tässä näkökulmaa käytetään viittaamaan Soft Systems Metodologian (Checkland & Scholes 1990) käyttämään termiin maailmankatsomus (saks. weltanschauung, engl. worldview). Yleinen näkökulman tulkinta on suppeampi, mutta tässä käytetään sitä samassa merkityksessä kuin Markku Nurminen (1988) käytti kuvatessaan erilaisia tulkintoja ihmisen ja tietotekniikan suhteesta. Näkökulma kattaa tässä tietotekniikan kehittämisen, käytön ja tutkimuksen.

kielmassa tarkastellaan ohjelmankehitystä toimintajärjestelmänä. Ohjelmankehityksessä yhdistyy tuote- ja prosessinäkökulma (Floyd 1988). Tuotenäkökulma edustaa ohjelman kehitystä sille asetuista vaatimuksista aina varsinaiseen ohjelmaan. Prosessinäkökulma sisältää ohjelmankehitykseen osallistuvien toimijoiden välistä vuorovaikutusta ja oppimista. Asiakkaan mahdollisuus osallistua liittyy keskeisesti prosessinäkökulmaan. Muuttamalla tuote- ja prosessinäkökulman painotusten suhdetta voidaan muodostaa erilaisia näkökulmia ohjelmaa kehittäviin toimintajärjestelmiin. Tässä luvussa kuvataan SSM:n prosessi, työkalut, yhteys tietotekniikkaan ja soveltaminen tässä tutkielmassa.

2.1 Soft Systems Metodologia (SSM)

SSM on tarkoitettu muokattavaksi, jotta menetelmä soveltuu sen käytön kontekstiin. Erilaiset sovellustavat ovat myös johtaneet oivalluksiin, joiden mukaan metodologiaa on kehitetty. Ajan myötä SSM:n muoto on kuitenkin vakiintunut ja muutokset siihen ovat tapahtuneet harvemmin (Checkland 1999). SSM:n looginen seitsemänvaiheinen prosessi (Checkland 1981) kuvaa ensimmäistä versiota metodologiasta. Kehittyneemmässä versiossa loogisen prosessin rinnalla on kulttuurinen prosessi, jossa analysoidaan itse interventiota ja kohdeorganisaatiota (Checkland & Scholes 1990). SSM hyödyntäessä huomattiin myös, että sitä voitiin käyttää ajattelutapana todellisuuden jäsentämiseen (ibid.) Tässä luvussa esitellään nämä SSM:n kolme muotoa.

SSM:n prosessissa on seitsemän vaihetta ja ne voidaan jakaa kahteen kokonaisuuteen todellisuuteen ja teoriaan (ks. Kuvio 1). Prosessin lähtökohtana on ongelmallinen tilanne. Ongelmallinen -käsitteellä halutaan tehdä ero yksiselitteisiin ongelmiin, joihin on looginen ratkaisu. Aluksi pyritään keräämään mahdollisimman paljon mielipiteitä ja näkökulmia tilanteesta. Seuraavassa vaiheessa luodaan kuvaus tilanteesta. Apuna voidaan käyttää tilanne-kuvia, joilla kuvataan ongelman keskeiset tekijät ja niiden väliset suhteet. Kuvauksia käytetään keskustelun apuvälineinä. Tarkoituksena on erilaisten ongelma-aiheiden tunnistaminen.



Kuvio 1: Soft Systems Metodologian loogisen analyysin prosessin seitsemän vaihetta (Checkland & Scholes 1990). Prosessi jakaantuu kahteen osaan: todellisuuteen ja järjestelmäajatteluun siitä. Todellisuuteen liittyvässä osassa aluksi kuvataan ongelmallinen tilanne ja lopuksi päätetään tilanteen kehittämisen toimenpiteistä. Järjestelmäajattelua käytetään toimintajärjestelmämallien luomiseen, joissa ongelmia ei ole. Malleja käytetään apuvälineenä keskusteltaessa toimenpiteistä tilanteen kehittämiseksi.

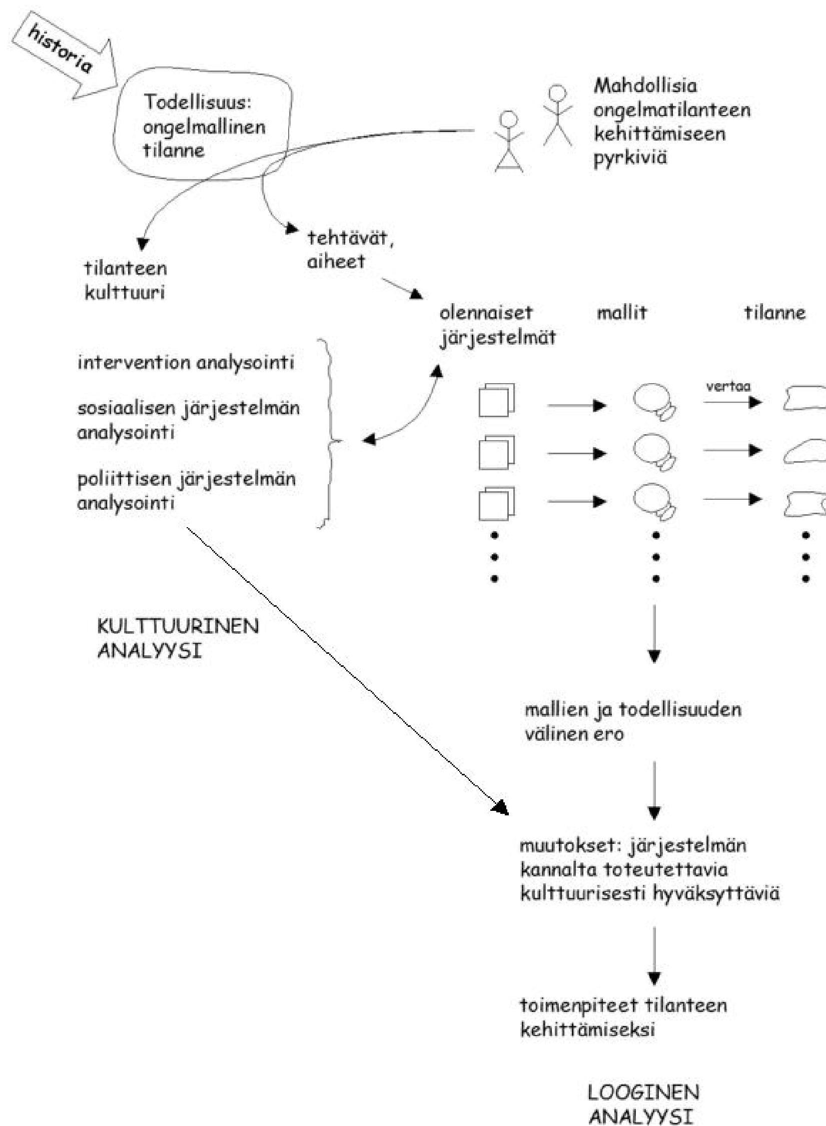
Eri ihmiset näkevät asiat eri lailla. Toiselle Greenpeace on ympäristönsuojelua, toiselle terrorismia, harjoittava organisaatio. Valitsemalla sopivia näkökulmia voidaan samaa toimintaa tulkita erilaisina toimintajärjestelminä, joissa ongelma-aiheita ei enää esiinny. Tarkasteltavaa toimintaa voidaan myös rajata kahdella tavalla, joko tehtävän mukaan organisaation yksikön sisällä tai aiheen mukaan, joka ei noudata yksikkörajoja. Näkökulma ja toiminnan idea (toimintajärjestelmän tuottama muutos) muodostavat ydinmääritelmän ytimen. Ydinmääritelmät ovat lähtökohta toimintajärjestelmien mallien luomiselle.

Toimintajärjestelmämalli on n. seitsemästä komponentista koostuva kokonaisuus. Komponentit on liitetty toisiinsa sen mukaan mitä ne tuottavat ja mitä ne itse tarvitsevat toimiakseen. Komponenttien vuorovaikutus muodostaa järjestelmän. Lisäksi voidaan tunnistaa kolme komponenttia, joilla on toimintaa ohjaava tehtävä: ensimmäinen sisältää toiminnalle asetetut kriteerit, toinen valvoo näiden noudattamista ja kolmas ohjaa toi-

mintaa. Toimintajärjestelmillä ei ole vastinetta todellisuudessa vaan ne ovat ydinmääritelmien muutosprosessien loogisia kuvauksia. Ne on tarkoitettu tukemaan keskustelua, joka huomioi erilaisia näkökulmia.

Ydinmääritelmien ja niitä vastaavien toimintajärjestelmien mallien teko edustaa prosessin teoreettista osuutta (ks. edellä Kuvio 1). Toimintajärjestelmien luontia seuraavassa vaiheessa verrataan niitä todellisuuteen. Vertailua hyödynnetään keskusteltaessa eri vaihtoehdoista ongelmalliseksi koetun tilanteen kehittämiseksi. Keskusteltaessa tulee esille eri sidosryhmien käsitys ratkaisuvaihtoehdoista. Tavoitteena on löytää ratkaisu, joka on sekä toimiva että hyväksyttävissä. Prosessin tarkoituksena voi olla joko pelkän selvityksen tekeminen tai myös ryhtyminen konkreettisiin toimenpiteisiin tilanteen muuttamiseksi. Varsinaiset toimenpiteet ovat prosessin ulkopuolella, mutta näiden jälkeen syntynyttä uutta tilannetta voidaan jälleen tarkastella SSM:a hyödyntäen.

Edellä esitetty prosessi kuvaa ensimmäistä vakiintunutta tapaa hyödyntää SSM:a. Tavoitteena ei ollut pelkästään ideoiden luonti vaan myös ongelman ratkaisevien toimenpiteiden mahdollistaminen (Checkland 1999). Toimenpiteiden onnistuminen on riippuvainen niistä ihmisistä, joiden työhön tai asemaan muutokset vaikuttavat. Alussa SSM:n harjoittajat selvittivät epävirallisesti millaiset muutokset olivat organisaation sosiaalisen ja poliittisen tilanteen mukaan mahdollisia. Organisaatiokulttuurin analysointi liitettiin kuitenkin myöhemmin osaksi SSM:a (ibid.). Kulttuurinen analyysi tuli aikaisemman loogisen analyysin rinnalle. SSM:n kehittyneempi versio koostuu näistä kahdesta keskenään vuorovaikutuksessa olevasta prosessista (ks. Kuvio 2).



Kuvio 2: Soft Systems Metodologian kehittyneemmän version rinnakkaiset kulttuurisen ja loogisen analyysin prosessit (Checkland & Scholes 1990). Kulttuurisessa analyysissä selvitetään edellytykset toiminnan kehittämiselle. Analyysi sisältää valinnan miten soveltaa SSM:a, keskeisten sidosryhmien tunnistamisen ja kartoituksen siitä mitkä toimenpiteet katsotaan mahdollisiksi. Loogisen analyysin tarkoitus on tukea päättämistä toimenpiteistä tilanteen kehittämiseksi. Tähän liittyy ongelmallisen tilanteen kuvaaminen, vaihtoehtoisten toimintajärjestelmien luominen ja toimintajärjestelmien vertaaminen todellisuuteen päätännän tukemiseksi.

Kulttuurinen analyysi alkaa samasta vaiheesta, ongelmallisesta tilanteesta, kuin looginen analyysi. Seuraavana on itse intervention analyysi ja tilanteen osapuolten tunnistaminen. Voidaan tunnistaa henkilöitä kolmessa roolissa: asiakkaan, tekijän ja omistajan. Asiakas on alun perin käynnistänyt prosessin tilanteen kehittämiseksi ja hänet on pidettävä ajan tasalla prosessin etenemisestä. Tekijän on huomioitava käytössä olevat resurs-

sit vaihtoehtoja analysoidessaan. Omistajat ovat ne joihin nykyinen tilanne ja/tai tilanteen muutos vaikuttavat. He ovat hyviä lähteitä erilaisille näkökulmille tilanteen kehittämiseksi. Tekijän voidaan katsoa itse olevan intervention omistaja ja näin hän voi soveltaa SSM:a tämän prosessin suunnitteluun.

Intervention analyysissä käsiteltiin jo rooleja, jotka liittyivät kehittämisen lähtökohtiin. Sosiaalinen analyysi käsittelee rooleja laajemmin. Roolit määrittelevät yhteisön tai organisaation jäsenten eroja. Ne voivat olla virallisia tai epävirallisia. Normit liittyvät käsityksiin siitä, miten eri rooleissa olevien ihmisten odotetaan käyttäytyvän. Arvot luovat kriteerit, joiden mukaan arvioidaan käyttäytyykö joku roolinsa mukaisesti. Roolit edustavat yhteisössä myös valtaa, joka on poliittisen analyysin kohde. Eri organisaatioissa eri asiat edustavat valtaa. Kulttuurinen analyysi poistaa rajan teorian ja todellisuuden välillä, koska luotavat järjestelmämallit perustuvat organisaatiossa eri rooleissa toimivien ihmisten käsityksiin (Checkland 1999).

SSM on tarkoitettu muokattavan menetelmäksi, joka soveltuu sen käytön kontekstiin. Samalla kun metodologiaa muokataan kontekstiin sopivaksi, on myös mietittävä sen käsitteitä ja miten ne soveltuvat tilanteen jäsentelyyn. SSM:n soveltajat huomasivat, että he alkoivat myös käyttää metodologiaa todellisuutta kuvaavana teoriana (Checkland & Scholes 1990). SSM tarjoaa ajattelutavan, jolla voidaan jäsentää havaintoja todellisuudesta. Tavoitteena ei ole todellisuuden muuttaminen vaan siitä oppiminen ja siinä asiakkaana on käyttäjä itse. SSM:n soveltaminen sijoittuu aina seitsemänvaiheisen mallin ja metodologian käyttämisen ajattelutapana väliselle skaalalle (ibid.).

2.2 SSM:n työkalut

Tutkielmassa sovelletaan SSM:a ajattelutapana (ks. edellinen luku). Käytettävät työkalut ovat samoja kuin ensimmäisessä SSM:n versioissa, ero on niiden hyödyntämistavassa. Tutkielmassa käytetään olemassa olevaa ohjelmankehityksen järjestelmämallia. Kulttuurista analysointia SSM:n toisen version muodossa ei tehdä, koska tutkielmassa hyödynnetään kirjallisuudessa esitettyjä käsityksiä ohjelmankehityksestä. Työkalujen kuvaukset ja esimerkit perustuvat Soft Systems Methodology in Action (Checkland & Scholes 1990) teokseen. SSM:n ensimmäisen version työkaluja ovat tilanne-kuvat,

ydinmääritelmät, toimintajärjestelmämallit ja toimintajärjestelmien vertailu.

Tilanne-kuvat (Rich Pictures) on tarkoitettu intervention tekijän ja interventioon liittyvien sidosryhmien väliseen viestintään. Ongelmalliseksi koetun tilanteen piirteitä voidaan esittää helpommin kuvien avulla kuin kirjallisesti, koska kuvat tuovat tilanteen osatekijöiden ja näiden suhteiden kokonaisuuden esille. Intervention tekijä voi käyttää kuvia tilanteesta muodostamansa käsityksen tarkistamiseen sidosryhmien edustajien kanssa. Tilanne-kuvien avulla on tarkoitus tuoda esille ongelma-aiheet, joita toimintajärjestelmiä luotaessa pyritään välttämään.

Toimintajärjestelmät ovat toiminnan logiikan kuvauksia. Niitä ei ole tarkoitus toteuttaa sellaisenaan, vaan ne ovat keskustelun apuvälineitä. Toimintajärjestelmän ytimen muodostaa valittu näkökulma toimintaan ja toiminnan aiheuttama muutos (syötteestä tulokseksi). Näkökulma ja muutos ovat osa toimintajärjestelmän ydinmääritelmää, joka määrittelee näiden lisäksi sidosryhmät (asiakkaat, tekijät ja omistajat) sekä toiminnalle asetettavat kriteerit. Toimintaa voidaan tarkastella useasta näkökulmasta ja riippuen näkökulmasta on toiminnan tuottama muutos erilainen. Esimerkiksi kirjasto voidaan nähdä sekä sivistystä että viihdettä tarjoavana instituutiona.

Ydinmääritelmän tekemistä tukee SSM:ssa muistisääntö ja -kaava. Ne täydentävät toisiaan ydinmääritelmän kuvauksina. Muistisääntönä toimii kirjain yhdistelmä CATWOE. Jokainen kirjain edustaa yhtä ydinmääritelmän osaa: Asiakas (Customer) on se johon muutos vaikuttaa, tekijä (Actor) toteuttaa muutoksen, muutos (Transformation) kuvaa toimintajärjestelmän prosessia, näkökulma (Worldview) määrittelee mitä toimintajärjestelmä tekee, omistaja (Owner) voi estää toiminnan ja kriteerit (Enviromental constraints) määrittelevät toiminnan puitteet. Muistikaavana toimi XYZ -järjestelmä: joka tekee X:ää käyttämällä Y:tä saavuttaakseen Z:n. Muistisääntö kuvaa ydinmääritelmän muodostavia tekijöitä, kun taas muistikaava kuvaa muutosta ja tuo lisäksi esille myös keinon sen aikaansaamiseksi.

Ydinmääritelmät ovat lähtökohtana erilaisille toimintajärjestelmien malleille. Jokaista ydinmääritelmää vastaa yksi toimintajärjestelmä. Toimintajärjestelmän mallin on toteu-

tettava ydinmääritelmän kuvaama muutos noudattaen siinä annettuja kriteereitä. Malli koostuu noin seitsemästä komponentista, jotka vähintään tarvitaan syötteen muuttamiseksi tulokseksi. Komponentit on kytketty toisiinsa, niiden tarvitsemien syötteiden ja tuottamien tulosten perusteella. Yksittäisen komponentin kuvauksessa keskeisenä on verbi, joka edustaa komponentin syötteelle tekemää muutosta. Jos prosessin ymmärtäminen vaatii tarkempaa kuvausta, voidaan jokainen komponentti esittää noin seitsemästä komponentista koostuvana toimintajärjestelmänä.

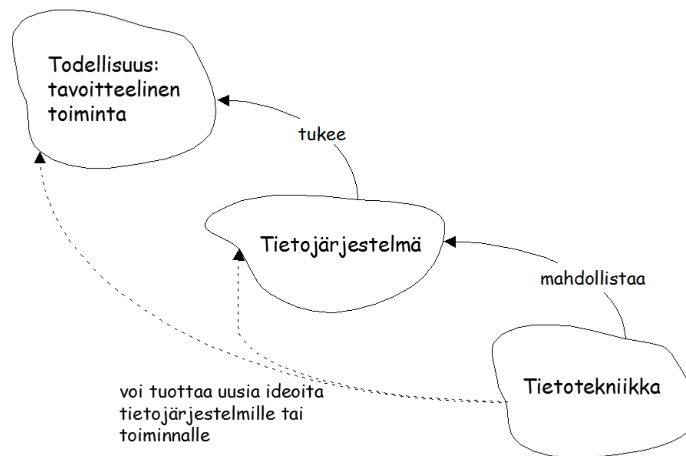
Toiminnalle asetettavia kriteereitä valvoo toimintajärjestelmässä kolme ylimääräistä komponenttia. Komponenttien voidaan ajatella olevan valvottavan järjestelmän ulkopuolella seuraten tätä kokonaisuutena. Yksi komponenteista valvoo toimintajärjestelmän toimintaa. Toinen komponentti sisältää toiminnalle annetut kriteerit, joita ovat: toteuttaako toimintajärjestelmä tarkoituksensa, riittävätkö resurssit, saavuttaako järjestelmä (pitkällä tähtäimellä) tavoitteensa. Kolmas komponentti tekee toimintaa korjaavia muutoksia, jos toiminnan on havaittu poikkeavan annetuista kriteereistä. Tavoitteellisen toimintajärjestelmän malli kuvaa kokonaisuutta, joka tehtävänsä suorittamisen lisäksi pystyy sopeutumaan muutoksiin.

Toimintajärjestelmien malleja käytetään apuvälineinä keskusteltaessa toiminnan kehittämisestä. Useamman mallin käyttö jäsentää keskustelua ja niitä voidaan myös käyttää kyseenalaistamaan vallitsevia käsityksiä tilanteesta. Toimintajärjestelmien malleja voidaan hyödyntää neljällä tavalla: epävirallinen keskustelu, määrämuotoinen keskustelu, erilaisten skenaarioiden kirjoittaminen mallien “suorittamisen” perusteella ja todellisuuden mallintaminen samassa muodossa. Määrämuotoinen keskustelu on näistä yleisin, jossa malleja käytetään kysymysten asettamiseen todellisuudesta. Kysymyksiin vastaamisen tarkoituksena on herättää keskustelua tilanteen kehittämisen keinoista.

2.3 SSM ja tietotekniikka

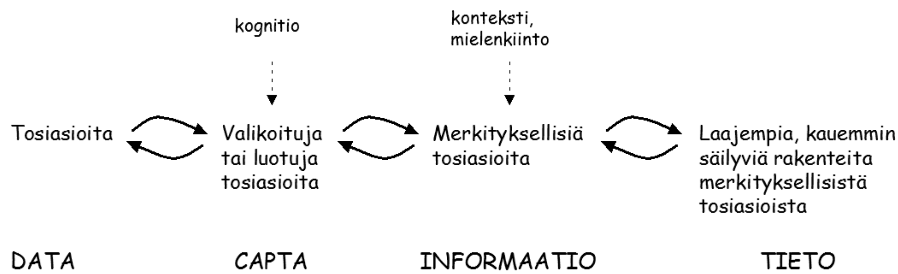
SSM:n yhteys tietotekniikkaan on organisaation toiminnan tiedontarpeiden täyttämisessä (Checkland & Poulter 2006). Toimintajärjestelmien malleja voidaan tarkastella myös tiedonkäsittelyn näkökulmasta. Jokainen toiminto tarvitsee tietoa toimiakseen ja samalla myös tuottaa sitä. Toimintajärjestelmän rinnalle voidaan mallintaa tietojärjestelmä, joka

kuvaa toimintojen välistä tiedon kulkua. Tietojärjestelmät toteutetaan nykyään useimmiten tietotekniikan avulla (ibid.). Seurauksena on kolme keskenään vuorovaikutuksessa olevaa järjestelmää (ks. Kuvio 3). Tässä luvussa esitetään miten SSM:a sovelletaan tietojärjestelmän määrittelemiseen, tietojärjestelmän kehittämiseen ja tietojärjestelmä- ja tietojenkäsittelytieteiden käsitteiden erojen selventämiseen.



Kuvio 3: Toiminnan, tietojärjestelmän ja tietotekniikan suhde (Checkland & Poulter 2006). Tavoitteellinen toiminta tarvitsee tietoa ja samalla myös tuottaa sitä. Tietojärjestelmä kuvaa toimintojen välistä tiedon kulkua. Tietojärjestelmät toteutetaan nykyään useimmiten tietotekniikan avulla. Tietotekniikan kehittäminen voi mahdollistaa uudenlaisen toiminnan.

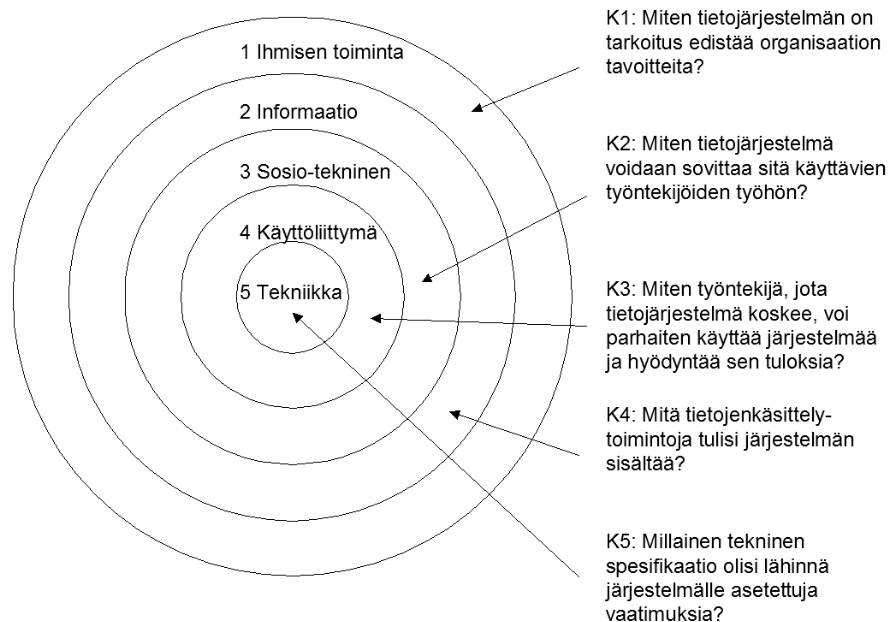
Tietojärjestelmät ovat luonnollinen kohde SSM:lle, koska tieto on edellytys tavoitteelliselle toiminnalle ja tietojärjestelmien voidaan katsoa olevan järjestelmäajattelun yksi sovellus. Tieto ja tietojärjestelmä käsitteiden merkitys on kuitenkin epäselvä. Osana SSM:n tutkimusta on myös näitä käsitteitä pyritty määrittelemään tarkemmin (Checkland & Holwell 1998). Tieto voidaan jakaa dataan, captaan ja informaatioon (ks. Kuvio 4). Data (lat. dare, antaa) edustaa tosiasioita todellisuudesta. Capta (lat. capere, ottaa) on kiinnostuksen mukaan valikoituja tosiasioita. Valinnan perustelu tarjoaa kontekstin, joka tekee näistä tosiasioista merkityksellisiä eli voidaan puhua informaatiosta. Tieto on informaatiosta koostuvia monimutkaisempia ja pysyvämpiä rakenteita.



Kuvio 4: Datan, captan, informaation ja tiedon väliset suhteet (Checkland & Holwell 1998). Tieto voidaan jakaa dataan, captaan ja informaatioon. Data edustaa tosiasioita todellisuudesta. Capta on kiinnostuksen mukaan valikoituja tosiasioita. Valinnan perustelu tarjoaa kontekstin, joka tekee näistä tosiasioista merkityksellisiä eli voidaan puhua informaatiosta. Tieto on informaatiosta koostuvia monimutkaisempia ja pysyvämpiä rakenteita.

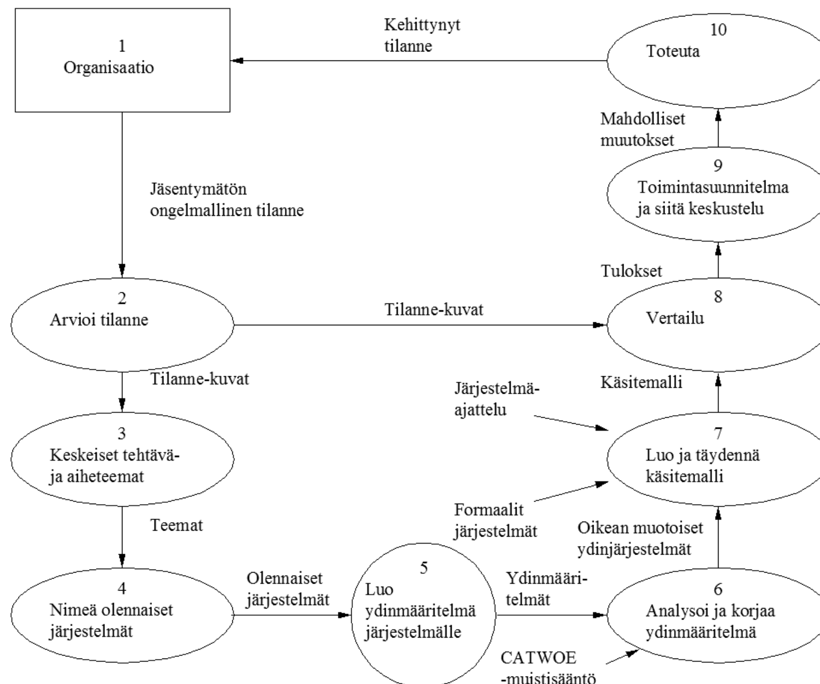
Tietojärjestelmille ei ole yhtä vakiintunutta määritelmää (King & Lyytinen 2006). SSM:n hengen mukaisesti tietojärjestelmä -käsitettä ei ole pyritty määrittelemään yksiselitteisesti vaan tarjoamaan työkalut erilaisten näkökulmien vertailuun (Checkland & Holwell 1998). Organisaatio koostuu vuorovaikutuksessa olevista ihmisistä, jotka jakavat kokonaan tai osittain yhteisen käsityksen ympäristöstään. Teknologia tukee ihmisten välistä viestintää. Ihmisten, organisaation ja teknologian vuorovaikutus muodostavat yhdessä tietojärjestelmän. Jokaisessa tietojärjestelmä -käsitteen kuvauksessa on nämä kolme tekijää sekä niiden väliset suhteet kuvattu (ibid.).

Multiview on metodologia tietojärjestelmien kehittämiseen (Avison & Wood-Harper 1990). Nimensä mukaisesti se yhdistää useamman näkökulman, erityisesti inhimillisen ja teknisen, tietojärjestelmien luomiseen. Multiview koostuu viidestä vaiheesta: ihmisen toiminta, tiedontarpeet, sosiotekninen suunnittelu, käyttöliittymät ja tekninen toteutus (ks. Kuvio 5). Kaksi ensimmäistä vaihetta painottavat enemmän analyysiä ja kolme viimeistä suunnittelua. Jokainen vaihe edustaa eri näkökulmaa tietojärjestelmiin ja tarjoaa näihin erikoistuneet menetelmät. Metodologia on usean olemassa olevan menetelmän yhdistelmä. Sitä ei ole tarkoitettu seurattavan sellaisenaan, vaan sovellettavaksi tilanteen mukaan.



Kuvio 5: Multiview metodologia (Avison & Wood-Harper 1990). Multiview on metodologia tietojärjestelmien kehittämiseen. Se yhdistää inhimillisen ja teknisen näkökulman tietojärjestelmän luomiseen. Multiview koostuu viidestä vaiheesta: ihmisten toiminnan analysointi, tiedontarpeiden analysointi, sosiotekninen suunnittelu, käyttöliittymäsuunnittelu ja tekninen toteutus.

Erityinen Multiview metodologian piirre on eksplisiittinen mahdollisuus, että tietotekniikan kehittämiseen ei ryhdytä. Mahdollisuus liittyy ensimmäiseen vaiheeseen, ihmisen toiminnan analyysiin, jossa hyödynnetään SSM:a. Multiview käyttää SSM:n ensimmäistä versioita (ks. Kuvio 6). Ihmisen toiminnan analyysissä on tarkoitus selvittää mikä on ratkaistava ongelma ja mitä sen ratkaisu sisältää. Ero SSM:n prosessiin näyttäisi olevan siinä, että Multiview pyrkii rajaamaan tarkasteltavan järjestelmän jo ydinmääritelmien perusteella. Alkuperäisessä SSM:n prosessissa verrattiin useampaa järjestelmämallia todellisuuteen. On mahdollista, että toiminnan muutos ei vaadi tietotekniikka, muuten toimintajärjestelmämalli on lähtökohta tiedon tarpeiden analysoinnista alkavalle ohjelmankehitykselle.

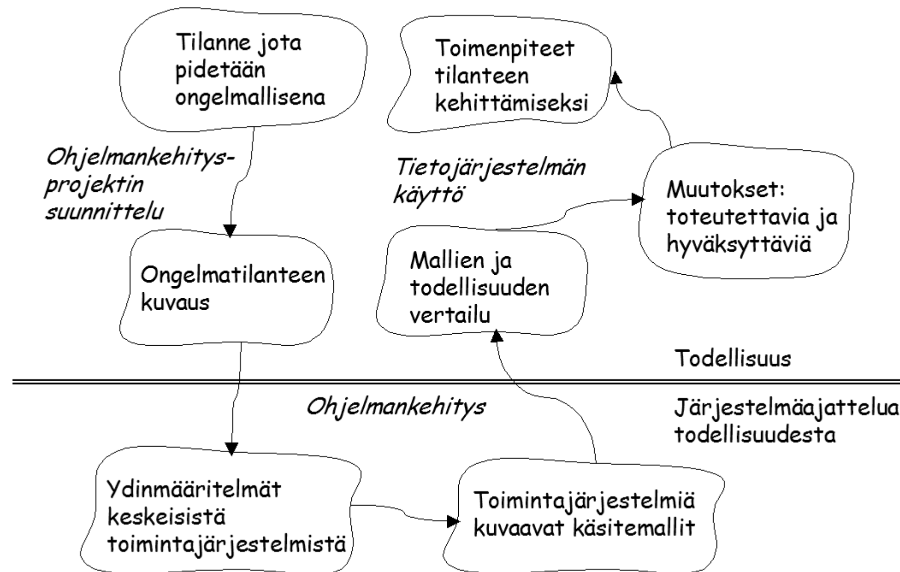


Kuvio 6: Multiview - toiminnan analyysi käyttäen Soft Systems Metodologiaa (Avison & Wood-Harper 1990). Analyysi perustuu SSM:n ensimmäiseen versioon. Prosessi noudattaa alkuperäistä seitsemän vaiheista mallia: ongelmallinen tilanne, tilanteen kuvaus, ydinmääritelmät, niiden toimintajärjestelmät, toimintajärjestelmien vertailu todellisuuteen, muutoksista päättäminen ja toteutus. Osa vaiheista on Multiview metodologiassa jaettu useampaan osaan kuin alkuperäisessä.

Edellä on esitetty Multiview metodologian vaiheet ja miten SSM:a on hyödynnetty siinä. Metodologiasta on olemassa myös uudempi versio (Avison & Fizgerald 2003). Ero abstraktilla tasolla on, että vaiheet neljä ja viisi on yhdistetty. Muutoksella on haluttu siirtyä evolutiivisenpään kehittämiseen, jossa suunnittelu, kehittäminen ja arviointi tehdään iteratiivisesti. Uudemmassa versiossa on myös muutettu vaiheiden sisältöjä, mutta SSM hyödyntäminen näyttäisi olevan ennallaan (ibid.). Multiview liittää SSM:n konkreettisesti tietojärjestelmien toteutukseen ja näin täydentää sitä.

Pekka Reijonen (2003) käyttää SSM:n käsitteitä selventämään organisaation tietokonetuettuun tietojenkäsittelyyn liittyvän kahden tieteen, tietojenkäsittely- ja tietojärjestelmätiede, eroja. Tietojenkäsittelytiede edustaa tässä ohjelmankehityksen ja tietojärjestelmätiede ohjelman käytön tutkimusta. Tutkimusalueet nähdään instituutioina, jotka ovat ihmisen jakamia vakiintuneita käsityksiä aiheesta (Berger & Luckmann 1966). SSM:n prosessikaaviota käytetään visualisoimaan ohjelman kehityksen ja käytön suhtei-

ta (ks. Kuvio 7). Ohjelmankehitys on erillinen organisaation toiminnasta (Nurminen & Forsman 1994). Ohjelma tulee osaksi tietojärjestelmää vasta kun sen käyttö on institutionalisoitunut organisaatiossa (Reijonen 2003).



Kuvio 7: Ohjelmankehityksen kuvaus käyttäen Soft Systems Metodologian kehitysprosessia mallina (Reijonen 2003). Ohjelmankehitysprojektin suunnittelu ja ohjelman käyttö osana tietojärjestelmää kuuluu todellisuuteen, kun taas ohjelmankehitys on todellisuuden (käytön ja sen kontekstin) ulkopuolella.

SSM:n ydinmääritelmiä käytetään tuomaan tarkemmin esille ohjelman kehityksen ja käytön instituutioiden erot. Ohjelmankehityksen näkökulman mukainen muutosprosessi on “ohjelman määritelmästä -> asiakkaan hyväksymäksi ohjelmaksi” (Reijonen 2003). Vastaavasti ohjelman käytön näkökulmasta muutosprosessi on “asennetusta ohjelmasta -> käytössä olevaan tietojärjestelmään” (ibid.). Ydinmääritelmien ydin, muutosprosessi, on erilainen riippuen näkökulmasta. Erot koskevat myös muita ydinmääritelmien piirteitä (ks. Taulukko 1). Tietojenkäsittelytiede ja tietojärjestelmätiede eroavat sekä ohjelmankehityksen tulkinnaltaan että sijainniltaan prosessissa. Tieteiden kohteena ovat eri asiat, joten näiden sekoittamista tulisi välttää (Reijonen 2003).

Taulukko 1: Ohjelmankehityksen ja ohjelman käytön instituutioiden erot hyödyntäen CATWOE (Customers (asiakkaat), Actors (toimijat), Transformation (muutosprosessi), Weltanschauung (näkökulma), Owners (omistajat) ja Environmental constraints (ulkoiset rajoitukset) muistisääntöä (Reijonen 2003).

CATWOE elementit	Ohjelmankehityksen instituutio	Tietojärjestelmän käytön instituutio
C (asiakkaat, johon projekti vaikuttaa)	Ohjelman toimittajan asiakkaat	Työntekijät ja asiakkaiden asiakkaat
A (toimijat, jotka toteuttavat T:n)	Kehittäjät (kehitykseen osallistuvat käyttäjät katsotaan myös kehittäjiksi)	Käyttäjät, johtajat
T (muutosprosessi syötteestä tulokseksi)	Ohjelman määritelmä -> Asiakkaan hyväksymä ohjelma	Asennettu ohjelma -> Käytössä oleva tietojärjestelmä
W (näkökulma, joka tekee T:n mielekkääksi valitussa kontekstissa)	Ohjelman kehityksen ammattilaiset luovat toimivan ohjelman ajallaan	Käytettävän tietojärjestelmän muuttaminen edistää tuotteiden ja palvelujen tuotantoa
O (omistajat, jotka voivat pysäyttää T:n)	Johtajat (ohjelman toimittajan tai asiakasyrityksen)	Asiakasyrityksen johtajat
E (järjestelmän ulkopuolella olevat tekijät, jotka otetaan annettuina)	Tekniset artefaktit, logiikan lait, kehitystyökalut, vaatimusmääritelmä, resurssit (aika, raha)	Ohjelma (myöhemmin tietojärjestelmä), normit, säännöt, proseduurit

Edellä on esitetty kolme tapaa soveltaa SSM:a tietotekniikkaan. Tilanne-kuvia käytettiin selventämään tietojärjestelmiin liittyviä käsitteitä ja näiden välisiä suhteita. SSM:n prosessia ja sen työkaluja käytettiin osana tietojärjestelmänkehitystä. Viimeisenä sovellettiin prosessimallia ja ydinmääritelmiä tuomaan esille kaksi organisaation tietokonetuettuun tietojenkäsittelyyn liittyvän tieteen eroja. Ensimmäisessä ja viimeisessä tavassa soveltaa SSM:a voidaan puhua siitä ajattelutapana. SSM tietojärjestelmän kehittämisessä edustaa mekaanisempaa tapaa. Samoja välineitä voidaan käyttää sekä konkreettiseen toimintaan että oppimiseen, ero on soveltamisessa.

2.4 SSM:n soveltaminen tutkielmassa

Pekka Reijonen (2003; ks. edellinen luku) käytti SSM:a tietojenkäsittelytieteen ja tietojärjestelmätieteen erojen selventämiseen organisaation tietokonetuetun tietojenkäsittelyn tulkinnassa. Tietojenkäsittelytiede tutki siinä ohjelman kehitystä, kun taas tietojärjestelmätiede sen käyttöä. Tutkielmassa sovelletaan Reijosen käyttämiä SSM:n menetelmiä päinvastaisessa tarkoituksessa yhdistämään ohjelman kehityksen ja käytön näkökulmat. Tavoitteena on näkökulmia yhdistämällä tunnistaa ohjelmointimenetelmiä, jotka huomioisivat tulevan käytön. Menetelmät mahdollistaisivat asiakkaan osallistumisen ohjelmankehitykseen. Samalla hän voi myös arvioida tulevan ohjelman vaikutuksia toimintaan ja tämän perusteella ehdottaa muutoksia.

Tutkielma soveltaa SSM:n seitsemänvaiheista prosessia. Ongelmallisen tilanteen sijaan voi lähtökohtana olla tavoitteellista toimintaa kuvaava järjestelmämalli (Checkland & Scholes 1990). Oletuksena on, että mallille on mahdollista löytää ydinmääritelm(i)ä. Tutkielmassa tarkastellaan tietokoneohjelman kehitystä. Ohjelmankehityksen järjestelmämallina käytetään ohjelman suhdetta ympäristöön (Lehman 1980) kuvaavaa mallia. Malli kuvaa ohjelmankehityksen lähtökohdan ja tuloksen. Tutkielmassa keskeistä on ohjelmankehityksen aiheuttama muutos, joten prosessin ääripäät ovat riittäviä.

Tavoitteellisen toiminnan järjestelmämallia seuraa SSM:n prosessissa ydinmääritelmien luominen. Tutkielman kontekstissa kyseessä on ohjelmankehitystä harjoittavien toimintajärjestelmien ydinmääritelmistä. Ydinmääritelmä sisältää tiedot asiakkaista, toimijoista, muutoksesta, näkökulmasta, omistajista ja ulkoisista vaatimuksista. Näkökulmat kuvaavat erilaisia käsityksiä ongelmasta ja sen ratkaisevasta muutoksesta. Ohjelmaa kehittävän toimintajärjestelmän mallista on tunnistettu kolme eri versioita, jotka tuottavat erityyppisiä ohjelmia (Lehman 1980): S (engl. Spesification), P (engl. Problem) ja E (engl. Embedded). S-tyyppinen ohjelma toteuttaa annetun spesifikaation. P-tyyppi ratkaisee ulkoisen ongelman. E-tyypin ohjelma muuttaa organisaation toimintaa. Ohjelmankehityksen kohteiden erilaisuus tulkitaan tutkielmassa edustavan eri ydinmääritelmien muutoksia.

Kolme ohjelmatyyppiä näyttäisi sopivan Markku I. Nurmisen (1988) esittämään kolmeen tietotekniikan näkökulmaan: järjestelmäteoreettinen, sosiotekninen ja humanistinen. Alun perin näkökulmia käytettiin tietotekniikan käytön tarkasteluun, jossa ihmisen ja tietotekniikan suhdetta kuvattiin tietojärjestelmämalleilla. Järjestelmäteoreettisessa tietojärjestelmässä tekniikan toimivuus on ensisijainen. Sosioteknisessä tietojärjestelmässä tasapainoillaan tekniikan vaatimusten ja organisaation tarpeiden välillä. Humanistisessa tietojärjestelmässä lähtökohtana ovat ihmisten tarpeet. Tietojärjestelmämallit tuovat esille suhtautumisen sidosryhmiin ja tietotekniikalle asetettavat vaatimukset. Tietotekniikan näkökulmat yhdistävät ohjelmatyyppejä ja tietojärjestelmämalleja. Tietojärjestelmämallit täydentävät ohjelmaa kehittävien toimintajärjestelmien ydinmääritelmiä.

SSM:n prosessissa ydinmääritelmien perusteella luodaan toimintajärjestelmien malleja. Malleja verrataan todellisuuteen, uuden toiminnan suunnittelemiseksi, jossa ongelmia ei enää esiinny. Tutkielmassa ei luoda toimintajärjestelmien malleja, koska ohjelmointimenetelmien tunnistamiseksi tämä ei ole välttämätöntä. Ydinmääritelmiä käytetään osallistavan ohjelmankehityksen määrittelemiseen. Luomalla erilaisia ydinmääritelmiä voidaan havaita eroja asiakkaan osallistumisen mahdollisuuksissa. Ydinmääritelmän muutosprosessin on oltava riittävän korkealla abstraktiotasolla, jotta asiakas voi siihen osallistua. Muutosprosessien perusteella määritellään niitä tukevien ohjelmointimenetelmien malleja. Näin saadaan esille myös erot ohjelmointimenetelmien luonteessa.

3 OHJELMOINNIN VIITEKEHYS

Ohjelmankehitys voidaan nähdä ohjelman kielen kehittämisenä (Winograd & Flores 1986). Ohjelmoimalla määritellään kielen käsitteet ja säännöt näiden manipuloimiseen. Käsitteet ja säännöt määrittelevät millaiseen käyttöön ohjelma soveltuu ja mitkä ovat sen vaikutukset. Ohjelmointikielet ovat tarkoitettuja kuvaamaan miten tietokoneen tulisi toimia. Yhteys ohjelman lähdekoodin ja ohjelman käytön välillä on kehittäjän mielessä (ibid.). Yhteyttä voidaan nimittää ohjelman teoriaksi (Naur 1992). Teorian toteuttaminen (ohjelmointi) on tietokoneen käytön mahdollistamista.

Ohjelmat ovat asiakkaalle työvälineitä tai työympäristöjä (Floyd 1988). Niiden merkitys tulee toiminnasta, jonka ne mahdollistavat. Ohjelmankehityksessä myös asiakkaalla on teoria ohjelman käytöstä (Reisin 1992). Hänelle ohjelman teoria ilmentää tietokoneen ominaisuuksia. Käsitys siitä mitkä ominaisuudet ovat mahdollisia ja millaisia ne ovat kehittyvät ohjelmankehitysprosessin aikana. Uudet ominaisuudet viittaavat myös uusiin käytön mahdollisuuksiin. Asiakkaan osallistuminen ohjelmankehitysprosessiin tapahtuu vaikuttamalla ohjelman teorian muotoutumiseen.

Mallimonopoli kuvaa tilannetta, jossa toinen toimija omistaa mallin ja toinen yrittää arvata millainen tämä malli on (Bråten 1973). Asiakas ei tiedä millaisen ohjelman hän haluaa (Brooks 1986, Parnas & Clements 1986). Hänellä on liian suppea kokemus tietokoneohjelmista, jotta hän voisi tietää millaiset ominaisuudet ovat mahdollisia. Kehittäjä ei voi tietää millainen ohjelman tuleva käyttö on, koska se on riippuvainen toiminnasta, jonka osaksi se tulee (Suchman 1987). Uransa aikana kehittäjä törmää liian harvaan saman alan sovellukseen, muodostaakseen niistä kattavan käsityksen (Brooks 2010). Tietokoneohjelman suhteen asiakkaalla on toiminnan ja kehittäjällä ratkaisun mallimonopoli.

Mallimonopolin purkamiseksi on malliheikolle ryhmälle luotava yhtä voimakas malli kuin mallivahvalle (Bråten 1973). Saman vahvuiset mallit mahdollistavat tasavertaisen keskustelun. Ohjelman teoria toimii ohjelmankehityksessä tällaisena mallina, se edustaa asiakkaalle teknologiaa ja kehittäjälle käyttöä. Ohjelmankehitysprosessin tuloksena teo-

riasta pitäisi tulla yhteinen. Jos ongelma on tarkasti määriteltävissä, niin yhdistäminen voi noudattaa rationaalista prosessia, jossa määritelmästä seuraa haluttu toteutus. Tietotekniikka voi myös mahdollistaa uusia toimintatapoja (Checkland & Holwell 1998). Tällöin sekä ongelma että ratkaisu ovat tuntemattomia (Torvinen & Korteinen 1997). Ongelman määritys syntyy mahdollisista ratkaisuista ja ratkaisu puolestaan ongelman määrityksestä. Ohjelmankehitys vaatii silloin asiakkaan ja kehittäjän vuorovaikutusta.

James Coggins (1996) esitti, että olio-ohjelmointi käyttäen riittävän korkeaa abstraktiotasoa mahdollistaisi käyttäjän osallistumisen konkreettisesti ohjelman suunnitteluun. Sovellusaluekielet (ks. luku 4.2.) ovat ohjelmointikieliä, joiden käsitteistö ilmentää käyttökohteen semanttista mallia (Fowler 2010). Cogginsin ajatus tuo sovellusaluekielten idean yleiskäyttöisiin ohjelmointikieliin. Asiakas voi ehdottaa käsitteitä ajattelemansa toiminnan (malli) näkökulmasta. Kehittäjä voi ehdottaa käsitteitä ajatellen tietokoneen mahdollisuuksia ja ohjelman toteutusta (malli). Seuraavaksi määritellään viitekehys, jonka avulla tunnistetaan sopiva ohjelmoinnin abstraktiotaso.

3.1 Ohjelmankehitysprosessien lähtökohtien aiheuttamat erot

Tietokoneohjelma on Meir Lehmannin (1980, 1061) mukaan "todellisuuden osan tai jonkin diskurssin abstraktiota kuvaavan mallin teorian sisältämän mallin malli"[4]. Tietokoneohjelman määritelmää voidaan selventää aloittamalla tulkinta ohjelman lähtökohtana olevasta ongelmasta. Ongelman kuvaus on abstraktio todellisuudesta tai tietystä diskurssista, koska siihen on valittu keskeiset piirteet ja muut piirteet on jätetty ulkopuolelle. Ohjelmankehityksen yhteydessä ongelma määritellään spesifikaatiossa, joka sisältää myös ratkaisun periaatteet. Ohjelmointikieltä voidaan pitää mallin teoriana, jonka kaikista ongelman ratkaisevista käsitteiden yhdistelmistä valitaan yksi malli. Tietokoneohjelma on valitun mallin toteutus (mallin malli) ja ongelman ratkaisu.

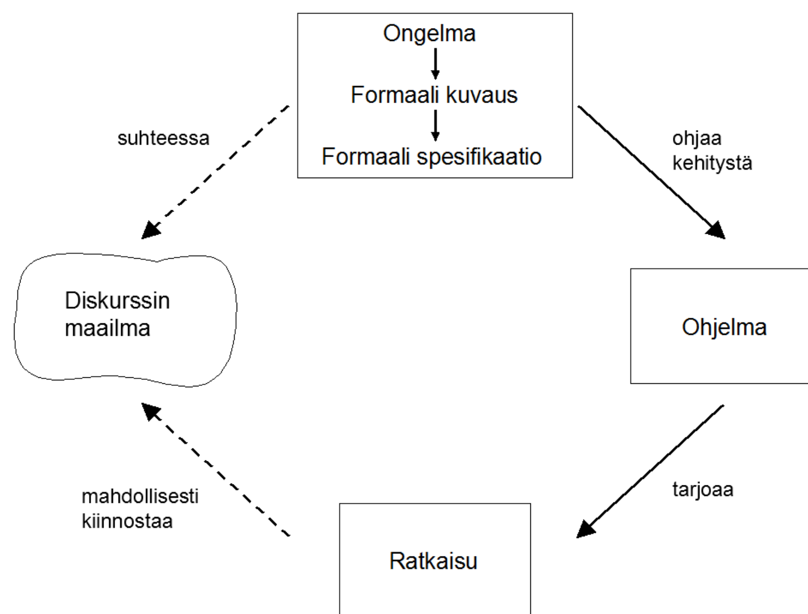
Ohjelman lähtökohta ja ratkaisun luonne kuvaavat ohjelman suhdetta ympäristöön. Lähtökohta on se ongelma, jonka ohjelman on tarkoitus ratkaista. Ratkaisu on se tulos tai

[4] "…any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse." (Lehman 1980, 1061)

hyöty, jonka ohjelma tuottaa. Ohjelmia voidaan luokitella eri tyyppeihin niiden käyttöympäristöön muodostaman suhteen mukaan. Ohjelmatyyppejä on käytetty kuvaamaan millaisia ylläpitotarpeita ohjelman käytön jatkuvuuden mahdollistaminen tuottaa (Lehman 1980). Tutkielman aiheena on ohjelmankehitys, jolloin ollaan kiinnostuneita siitä, millainen muutos johtaa tiettyyn ohjelmatyyppiin.

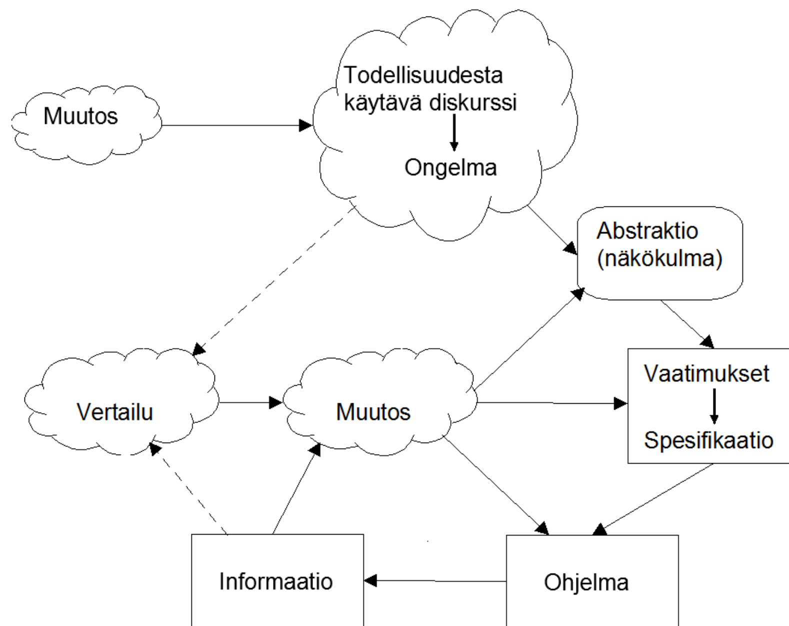
Tietokoneohjelmia voidaan luokitella käyttöympäristöön luodun suhteen mukaan S-, P- tai E-tyyppisiksi ohjelmiksi (Lehman 1980). Ensimmäinen ohjelmatyyppi on S-ohjelma englannin sanasta “spesification”, suomeksi spesifikaatio tai määritys. Toinen ohjelmatyyppi P-ohjelma, jossa lyhenne P tulee englannin sanasta “problem”, suomeksi ongelma. Kolmas ohjelmatyyppi on E-ohjelma, joka tulee englannin sanasta “embedded”, suomeksi sulautettu. Tässä luvussa kuvataan eri ohjelmatyyppien suhdetta ympäristöön ohjelmankehitysprosessin tuottamana muutoksena.

S-ohjelmat ovat tietokoneohjelmia, joiden toiminnallisuuden määrittää yksiselitteisesti spesifikaatio (Lehman 1980). Spesifikaatio voi käsitellä todellisuudessa esiintyviä asioita, mutta sen yhteys niihin on välillinen tai abstrakti. Todellisuuden sijaan spesifikaation voidaan sanoa liittyvän tiettyyn diskurssiin (ks. Kuvio 8). Spesifikaatio määrittelee ohjelman syötteen ja tulosteen suhteen. Jos syötteen ja tulosteen suhde muuttuu, on kyseessä uusi ohjelma, joka vastaa toista spesifikaatiota. Ohjelmankehityksen kannalta kehitysprosessin aiheuttama muutos on spesifikaatiosta ohjelmaksi.



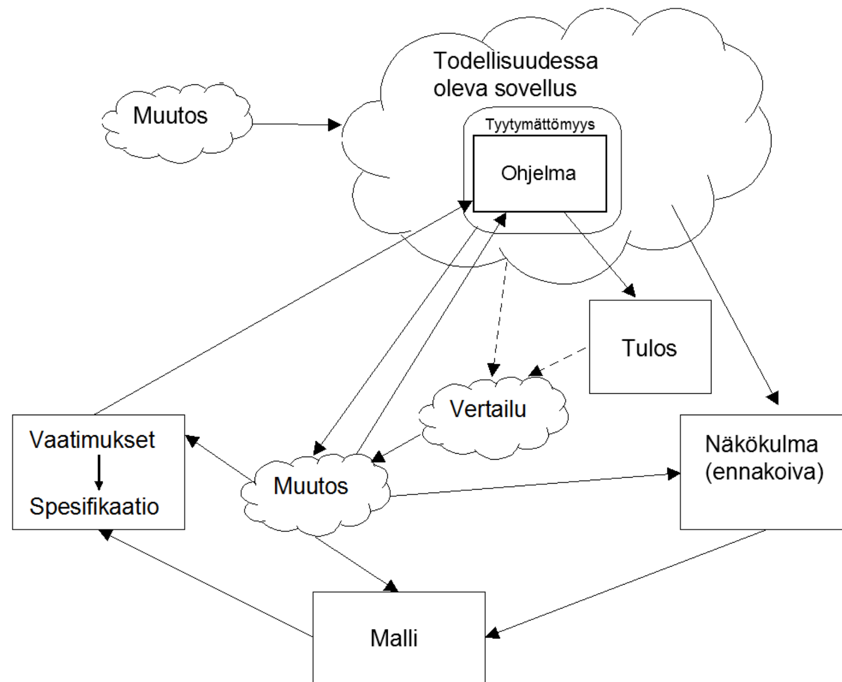
Kuvio 8: S-ohjelman (Specification = spesifikaatio) malli (Lehman 1980). S-ohjelman lähtökohtana on diskurssi, joka voi viitata johonkin todellisuuden asiaan, mutta yhteys on välillinen tai abstrakti. Ohjelma toteuttaa spesifikaation määrittelemän muutoksen syötteestä tulokseksi.

P-ohjelmat ovat tietokoneohjelmia joiden lähtökohtana on todellisuuden ongelma (Lehman 1980). Kaikkia ongelman piirteitä ei voida havaita tai ottaa huomioon. Ongelman määritys on todellisuuden abstraktio (ks. Kuvio 9). Tietokoneohjelma ei ratkaise varsinaista ongelmaa, vaan siitä tehdyn mallin. Ohjelman tuottamaa tietoa tai palvelua arvioidaan suhteessa koettuun todellisuuteen. P-ohjelmat ovat alttiita muutoksille, sitä mukaa kuin tieto ongelmasta lisääntyy tai ympäristö muuttuu (ibid.). Ohjelmankehityksen kannalta prosessin aiheuttama muutos on ongelmasta ohjelman tarjoamaksi ratkaisuksi.



*Kuvio 9: P-ohjelman (**P**roblem = ongelma) malli (Lehman 1980). P-ohjelma ratkaisee todellisuudessa olevan ongelman abstraktin mallin. Ohjelman oikeellisuus on riippuvainen siitä kuinka hyvin sen tuottama informaatio/palvelu sopii koettuun todellisuuteen. Tieto mallinnettavasta ongelmasta lisääntyy kun sitä käsitellään ja ongelma voi myös muuttua. P-ohjelmat vaativat ylläpitoa säilyäkseen relevantteina.*

E-ohjelmat ovat tietokoneohjelmia joiden tarkoituksena on mekanisoida osa ihmisen tai yhteisön toiminnasta (Lehman 1980). Toisin kuin edellä olevat ohjelmatyypit E-ohjelmat ovat osa todellisuutta, ne ovat upotettuja siihen (ks. Kuvio 10). Ohjelman malli sisältää kuvauksen myös sen tulevasta käytöstä. Osa työstä on tehtävä käyttäen tietokonetta. Valmis ohjelma on työkalu, joka sekä mahdollistaa että rajoittaa toimintaa. E-ohjelmat ovat voimakkaammin alttiita muutoksille, koska niiden yhteys toimintaan on tiiviimpi kuin aikaisemmissa ohjelmatyypeissä. Ohjelmankehitysprosessin kannalta kyseessä on muutos toiminnasta uudeksi tietokonetuetuksi toiminnaksi.



*Kuvio 10: E-ohjelman (**E**mbedded = upotettu) malli (Lehman 1980). E-ohjelma mallintaa ohjelman tulevan käytön. Ohjelmat ovat työkaluja tai työympäristöjä, joita on käytettävä työnteossa. Muutokset toiminnassa asettavat nopeammin vaatimuksia E-ohjelmille, koska niiden suhde työn tekoon on välitön.*

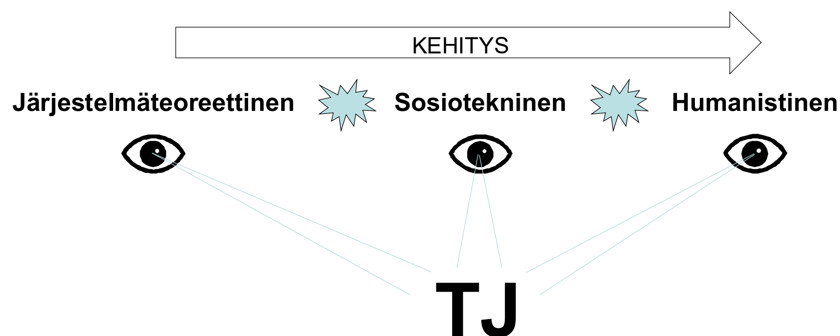
Alkuperäinen tarkoitus ohjelmatyypeillä oli tuoda esille kuinka paljon eri tyypit vaativat ylläpitoa. Tässä luvussa on tarkasteltu ohjelmatyyppejä toisesta näkökulmasta, ohjelmankehityksen aiheuttaman muutoksen edustajina. Tuloksena on saatu S-, P- ja E-tyyppiä vastaavat muutokset: spesifikaatiosta ohjelmaksi, ongelmasta ratkaisuksi ja toiminnasta tietokonetuetuksi toiminnaksi. Koska valmiin ohjelman on oltava määrämuotoinen, voidaan sanoa, että riippumatta ohjelmatyypistä, kaikki ohjelmat ovat toteutuessaan S-tyyppiä. Meir Lehman (1980) ehdottaakin, että metodologioiden kehittyessä kaikki ohjelmat luodaan S-tyyppisten ohjelmien muodostamista rakenteista.

P- ja E-ohjelmat ovat S-ohjelmien muodostamista rakenteista koostuvia. Mutta P- ja E-ohjelmien välillä ei ole samanlaista yhteyttä. E-ohjelmien on sisällytettävä kuvaus ongelmasta, joita toiminnan mekanisoinnilla pyritään ratkaisemaan. Tässä mielessä E-ohjelmat sisältävät myös P-ohjelmatyypin. E-ohjelmia ei kuitenkaan voida rakentaa pelkästään P-ohjelmista, koska P-ohjelmat eivät sisällä tulevan toiminnan kuvausta. P-ohjelmat toimivat osana E-ohjelmien sisältämää tulevan käytön mallia. Yhdessä ne kuvaavat ohjelman osana tietokoneen ulkopuolista todellisuutta (Lehman 1980). Yhdis-

telmää kutsutaan A-ohjelmaksi (engl. **A**pplication) eli sovellukseksi.

3.2 Kolme näkökulmaa tietotekniikkaan

Markku I. Nurminen (1988) käytti näkökulman käsitettä ihmisen ja tietotekniikan suhteen tarkasteluun. Näkökulma tulkittiin yhteiskunnalliseksi instituutioksi (Berger & Luckmanin 1966), joka määrittää yhteiset piirteet (Nygaard 1986) tietotekniikan käytölle, kehittämiselle ja tutkimukselle. Nurminen keskittyi tietotekniikan käyttöön, jossa ihmisen ja tietotekniikan suhde muodostaa tietojärjestelmän. Keskeisenä suhteessa on toimija (Kuutti 2003), joka määrittelee mitä ja miten tehdään. Tietojärjestelmän malli edustaa ideaalityyppiä, jossa tietyt piirteet ovat korostettuja näkökulman luonteen esille tuomiseksi (Nurminen 1988). Käyttämällä useampaa näkökulmaa Nurminen kuvasi myös kehitystä, jossa aikaisemman näkökulman ongelmat johtivat seuraavaan. Hän tunnisti kolme näkökulmaa: järjestelmäteoreettisen, sosioteknisen ja humanistisen (ks. Kuvio 11).

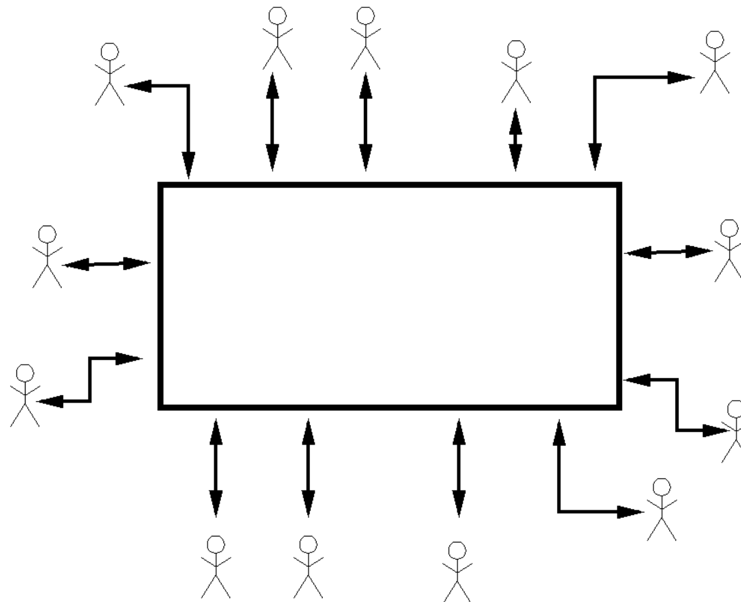


Kuvio 11: Kolme näkökulmaa tietojärjestelmiin. Ongelmat edellisen näkökulman tietojärjestelmän tulkinnan kanssa ovat johtaneet seuraavan syntyyn. Näkökulmat edustavat näin myös tietojärjestelmien käsityksen kehitystä.

Ensimmäinen näkökulma perustuu järjestelmäteoriaan (Nurminen 1988). Järjestelmä on toisiinsa vuorovaikutuksessa olevista osista koostuva kokonaisuus, jolla on ominaisuuksia joita yhdelläkään sen osalla ei ole (Bertalanfy 1969). Järjestelmäteoreettisen näkökulman voidaan katsoa alkaneen tietotekniikan käytön ensiaskelista. Tietokoneen aika oli aluksi kalliimpaa kuin ihmisen työ (Sackman 1974). Ohjelmankehityksessä resurssien taloudellinen käyttö ja tehokkuus olivat ohjaavia kriteereitä. Kehitysprojektin vaa-

timukset asetettiin etukäteen ja varsinainen suunnittelu koski ohjelman teknistä toteutusta (Nurminen 1988).

Aluksi tietokoneen käytössä oli kysymys laskennasta. Käyttäjä määritteli laskutoimituksen ja antoi tiedot, jotka hän halusi käsiteltävän tietokoneella. Tulokset palautettiin käyttäjälle. Voidaan sanoa, että käyttäjällä oli hallussaan koko tiedonkäsittely, tietokoneen ohjaamista lukuun ottamatta. Tietokoneiden kehittyessä voitiin tieto (data) tallentaa järjestelmään. Etuna oli, että useampi käyttäjä pystyi helpommin hyödyntämään tietoa omissa laskelmissa. Haittana oli, että tiedon omistajuus hävisi. Myös ohjelmat voitiin tämän kehityksen seurauksena tallentaa tietokoneelle. Tietokone "omisti" sekä tiedon että määritteli sen käsittelyn mahdollisuudet. Muutokset tietokoneessa oleviin ohjelmiin vaativat erikoisosaamista ja muutosten tarpeeseen suhtauduttiin epäilevästi (Nurminen 1988). Tietojärjestelmässä tietokone saneli miten tietoa käytettiin (ks. Kuvio 12).

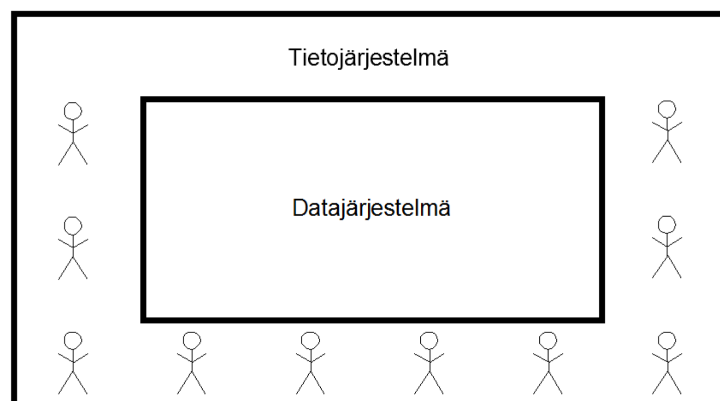


Kuvio 12: Tietojärjestelmässä tietokone omistaa tiedon ja sen käsittelyn välineet (Nurminen 1988). Tietokone määrää miten työtä tehdään ja miten viestintä tapahtuu järjestelmässä.

Järjestelmäteoreettinen näkökulma edustaa nk. kovaa järjestelmäajattelua (Checkland & Holwell 2004). Kovan järjestelmäajattelun mukaan kokonaisuutta voidaan kehittää lisäämällä siihen haluttuja ominaisuuksia uudella komponentilla tai muokkaamalla ole-

massa olevaa. Toimintaa tukevan tietotekniikan kehittäminen nähdään teknisenä ongelmana, jossa ratkaisu on oikea, kun tekniikka toimii annetun määrityksen mukaan. Ohjelmien käyttöönotto organisaatioissa toi mukanaan odottamattomia sivuvaikutuksia, jotka haittasivat organisaatioiden toimintaa (Nurminen 1988). Muutokset tietojärjestelmän osaan muuttivat kokonaisuuden ominaisuuksia.

Järjestelmäteoreettisen näkökulman mukaisessa ohjelmankehityksessä valmiin ohjelman tahattomat vaikutukset työn tekemiseen aiheuttivat ongelmia. Ohjelmankehitystä muutettiin ottamaan huomioon myös ihmisten toiminta. Teknisen järjestelmän rinnalle tuli ihmisten muodostama yhteisöllinen järjestelmä. Lähestymistapa edustaa sosioteknistä näkökulmaa (Nurminen 1988). Tavoitteena oli kahden järjestelmän samanaikainen optimointi (Mumford 1985). Perusteluna oli, että organisaatiolle tehokkainta on, kun sekä yhteisöllinen että tekninen järjestelmä toimivat hyvin. Tietojärjestelmä koostui kahdesta järjestelmästä ihmisten tietojärjestelmästä ja teknisestä datajärjestelmästä (ks. Kuvio 13). Ero tiedon ja datan välillä on, että tieto vaatii merkityksen antamisen, joka on mahdollista vain ihmiselle (Nurminen 1988; Checkland & Holwell 1998).



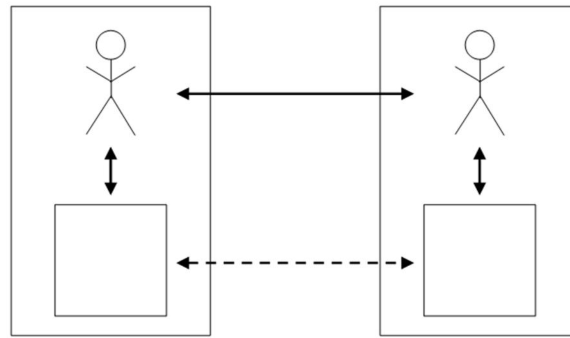
Kuvio 13: Datajärjestelmä ja tietojärjestelmä (Nurminen 1988). Ihmisten tietojärjestelmä ja tietotekninen datajärjestelmä muodostavat yhdessä sosioteknisen järjestelmän. Sosioteknistä järjestelmää suunniteltaessa huomioidaan ihmisten työ, mutta toteutettu datajärjestelmä määrittelee miten sitä käsitellään ja sen muuttaminen vaatii uuden järjestelmän kehittämisen.

Sosioteknisen näkökulman mukaisessa ohjelmankehityksessä oli kyseessä kahden järjestelmän, yhteisöllisen ja teknisen, rinnakkaisesta kehittämisestä. Teknisten tavoitteiden lisäksi huomioitiin työhön liittyviä tekijöitä kuten työelämän laatu sekä työntekijöi-

tä lähellä oleva päätäntä (Munkvold 2000). Keskeisenä mahdollistavana tekijänä oli työntekijöiden osallistuminen tulevan työn suunnitteluun. Ohjelmankehitysprosessi eteni kahtena rinnakkaisena prosessina, yhteisöllisenä ja teknisenä, joiden tulokset yhdistettiin vertaamalla vaihtoehtoja sekä hakemalla yhteensopivia ratkaisuja. Lopputuloksena oli kehitettävän järjestelmän suunnitelma. Kehitysprosessiin kuului myös valmiin ohjelman arviointi, jotta siitä voitaisiin oppia seuraaviin kehitysprojekteihin.

Sosiotekninen näkökulma näyttäisi huomioivan ihmisen tarpeet ohjelman kehityksessä. Näkökulman mukaisissa kehitysmenetelmissä on kuitenkin piirteitä, joiden seurauksena ohjelman käytöstä syntyvällä tietojärjestelmällä on samat ongelmat kuin järjestelmäteoreettisessa näkökulmassa. Tekniselle järjestelmälle annetaan yhtä suuri merkitys kuin yhteisölliselle. Usein teknologisten ratkaisujen perustelu on helpompaa kuin enemmän tulkinnanvaraisten yhteisöllisten (Nurminen 1988). Teknisen järjestelmän toteutus tapahtuu toiminnasta irrallisena (Nurminen & Forsman 1994). Arvioinnissa mahdolliset puutteet voidaan todeta, mutta ohjelmaa ei enää muuteta. Työntekijät joutuvat sopeutumaan valmiin ohjelman toimintatapaan ja sen käytön vaikutuksiin muuhun työhön.

Toisin kuin järjestelmäteoreettisella ja sosioteknisellä näkökulmalla, humanistisella näkökulmalla ei ole konkreettista ilmentymää. Näkökulmia edustamaan on käytetty tietojärjestelmää ideaalityyppinä. Ideaalityypin avulla voidaan myös käsitellä humanistista näkökulmaa. Humanistisen näkökulman mukaista tietojärjestelmää Nurminen (1988) nimitti ihmisen kokoiseksi (engl. ”Human Scale Information System”). Siinä jokaisella työntekijällä on oma datajärjestelmänsä. Järjestelyn ei tarvitse olla fyysinen, vaan riittää että työntekijällä on mahdollisuus kontrolloida tietoaan sekä sen käsittelyyn tarvitsemiaan työkaluja. Tietojärjestelmä ei synny tietotekniikan, vaan ihmisten välisen yhteistyön kautta (ks. Kuvio 14). Ihmisten oppima toimintapa luo tietojärjestelmän (Reijonen 2003). Humanistisen näkökulman mukaan tietojärjestelmä ei vaadi tietotekniikkaa toimiakseen.



Kuvio 14: Ihmisen kokoisen tietojärjestelmän viestinnän rakenne (Nurminen 1988). Tietojärjestelmässä jokaisella työntekijällä on oma datajärjestelmä, jonka sisältämät tiedot ja työkalut ovat hänen valitsemia. Tietojärjestelmä muodostuu toisiinsa yhteydessä olevista työntekijöistä (ilman tietokoneen väliintuloa). Myös datajärjestelmien tietoa voidaan siirtää työntekijältä toiselle, mutta transaktio tapahtuu vain työntekijän toiminnan kautta.

Humanistisen näkökulman mukaisessa ohjelmankehityksessä käyttäjä on aloitteen tekijä (Nurminen 1988). Esimerkkejä ohjelmankehityksestä, jossa käyttäjä voi olla aloitteen tekijä, ovat protoilu ja käyttäjälle räätälöidyt ohjelmointivälineet (ibid.). Protoilussa ohjelmasta luodaan iteratiivisesti yhä kehittyneempiä toiminnallisia malleja (prototyyppejä), joita käyttäjä arvioi ja joihin hän voi ehdottaa parannuksia. Prototyyppiä ei ole tarkoitusta käyttää, vaan kun oikeanlainen toiminnallisuus on löydetty, toteutetaan varsinainen ohjelma. Käyttäjälle räätälöidyt ohjelmointivälineet mahdollistavat hänen itsensä luoda sekä muokata käyttämiään sovelluksia. Käyttäjällä on mahdollisuus päättää työvälineistä ja niiden toiminnallisuudesta. Samalla hän joutuu ottamaan vastuutta omasta työstään ja sen soveltumisesta organisaatioon kokonaisuutena (Nurminen 1988).

3.3 Ohjelmaa kehittävien toimintajärjestelmien ydinmääritelmät

Soft Systems Metodologian (SSM) ydinmääritelmä kuvaa toimintajärjestelmän keskeisiä piirteitä. Valittu näkökulma toimintajärjestelmään ja toiminnan aiheuttama muutos muodostavat ydinmääritelmän ytimen. Muutos kuvaa toimintajärjestelmän tarkoitusta, joka vaihtelee näkökulman mukaan. Näkökulmia tietotekniikkaan käytetään tutkielmassa yhdistämään ihmisen ja tietotekniikan suhdetta kuvaavia tietojärjestelmämalleja ohjelmatyyppien edustamaan muutokseen. Tuloksena on ohjelmaa kehittävien toimintajärjestelmien ydinmääritelmiä, jotka eroavat tavassa tukea kehittäjän ja asiakkaan yhteistyötä. Yhdistämisen apuna käytetään CATWOE muistisääntöä (ks. luku 2.2). Muisti-

sääntö sisältää asiakkaat, toimijat, muutoksen, näkökulman, omistajat ja kriteerit.

Järjestelmäteoreettisessa näkökulmassa tuleva käyttäjä oli ohjelmankehitysprosessin ulkopuolella. Ohjelman käytöstä syntyvässä tietojärjestelmässä käyttäjä joutui sopeutumaan ohjelman vaatimiin toimintatapoihin. S-ohjelman määrittelee spesifikaatio ja myös sen toimivuus todennetaan siitä. Spesifikaation yhteys todellisuuteen on välillinen tai abstrakti. Ohjelma ei ole kuvaus todellisuudesta vaan spesifikaatiosta. Järjestelmäteoreettisen näkökulman käsitys ohjelmankehityksestä näyttäisi vastaavan S-ohjelman mukaista kehitysprosessia. Ydinmääritelmän voidaan katsoa edustavan teknologista determinismiä, jossa ihminen sopeutuu teknologiaan (Nurminen 1988).

Järjestelmäteoreettisen ohjelmankehityksen näkökulman ydinmääritelmällä (ks. Taulukko 2 vasen sarake) todellisuuden kuva on objektiivinen (W). Sen mukaan ohjelmankehitys on muutosprosessi (T), jossa spesifikaatio muutetaan ohjelmaksi. Käyttäjä (C) voi osallistua ohjelman ominaisuuksien määrittelyyn, mutta hän ei voi vaikuttaa ohjelman kehitykseen. Kehitystyön tekevät kehittäjät (A) ilman ulkopuolista puuttumista. Asiakasyrityksen johtajat (O) voivat pysäyttää kehitystyön, jos syntyvä ohjelma ei näytä vastaavan vaatimuksia (E) tai kehitystyö käyttää enemmän resursseja (E) kuin sovittu.

Sosioteknisessä näkökulmassa käyttäjät olivat mukana määrittelemässä ohjelmalle asetettavia vaatimuksia teknisten vaatimusten rinnalla. Kyseessä oli kahden järjestelmän suunnittelu: yhteisöllisen tietojärjestelmän ja teknisen datajärjestelmän. Käyttäjät määrittelivät ongelman, jonka teknisen järjestelmän oli ratkaistava. Sosiotekninen näkökulma sisälsi myös ohjelman arvioinnin. P-ohjelman lähtökohtana oli tietokoneen ulkopuolinen ongelma. Ratkaisun oikeellisuuden kriteerinä oli miten hyvin ohjelman tuottama informaatio tai palvelu ratkaisi ongelman. P-ohjelmatyypin määrittelemä muutos näyttäisi sopivan sosiotekniseen näkökulmaan ja sen tietojärjestelmämalliin.

Sosioteknisen ohjelmankehityksen ydinmääritelmä (ks. Taulukko 2 keskimmäinen sarake) näyttäisi sisältävän dualistisen (W) näkökulman todellisuuteen. Ohjelmankehityksessä pyritään yhteisöllisen ja teknisen järjestelmän rinnakkaiseen optimointiin. Kehitysprosessi tuottaa ratkaisun (T) tietokoneen ulkopuoliseen ongelmaan. Kehittäjät neu-

vottelevat ongelman määrittelystä käyttäjien (C) kanssa, mutta varsinainen toteutus on heidän (A) varassa. Vaikka käyttäjät osallistuvat projektiin vaikuttaa siltä, että projekti on usein käynnistetty johdon (O) taholta, jolloin sen lopettaminen ei ole käyttäjien toimesta mahdollista. Keskeiseksi onnistumisen kriteereiksi muodostuu kuinka hyvin (E) valmis ohjelma ratkaisee ongelman.

Humanistinen näkökulma asettaa käyttäjän keskipisteeseen. Hän ei ole pelkästään vaatimusten määrittelijä, vaan hän on aktiivisesti mukana ohjaamassa ohjelmankehityksen kulkua. Tietojärjestelmä syntyy kun useampi ohjelman käyttäjä on yhteydessä toisiinsa. Ohjelman oikeellisuuden määrittelee sen käytön tuottama hyöty. E-ohjelmat ovat osa todellisuutta. Ohjelmatyyppi huomioi myös tulevan toiminnan, jota ohjelman on tarkoitus tukea. Käyttäjä voi vaikuttaa tällaiseen ohjelmankehitykseen, koska se perustuu hänen tuntemaan todellisuuteen. E-ohjelma tyyppi näyttäisi sopivan Humanistiseen näkökulmaan, jossa asiakas on mukana, ei pelkästään määrittelemässä vaan myös kehittämässä tietojärjestelmää.

Humanistisen ohjelmankehityksen ydinmääritelmällä on subjektiivinen (W) näkökulma todellisuuteen (ks. Taulukko 2 oikea sarake). Sen mukaan käyttäjä (C) määrittelee millaisen ohjelman hän tarvitsee työssään. Kyseessä on toiminnan muuttamisesta tietokonetuetuksi toiminnaksi (T). Käyttäjä ei pelkästään toimi määrittelijänä vaan hän (A) on aktiivisesti mukana kehityksessä kehittäjien (A) rinnalla. Vaikka käyttäjä päättääkin millainen ohjelma luodaan, on yhä luultavaa, että organisaatioissa halutaan varmistua, että tuotannon välineet ovat yhtenäisiä (Nurminen 1988). Näin johto (O) yhä päättää viimekädessä. Ohjelmankehityksen onnistumisen määrittelee sen käytön toimintaan tuottama hyöty (E).

Taulukko 2: Kolme ohjelmankehityksen ydinmääritelmää.. Ydinmääritelmät kuvaavat eroja tulevan käytön huomioimisessa ohjelmankehityksessä.

		Järjestelmäteoreettinen ohjelmankehitys	Sosiotekninen ohjelmankehitys	Humanistinen ohjelmankehitys
Asiakas	(C)	asiakkaan työntekijät	asiakkaan työntekijät	asiakkaan työntekijät
Toimija	(A)	kehittäjät	kehittäjät	kehittäjät ja asiakkaan työntekijät
Muutos	(T)	spesifikaatio => ohjelma	ongelma => ratkaisu	toiminta => tietokonetuettu toiminta
Näkökulma	(W)	objektiivinen	dualistinen	subjektiivinen
Omistaja	(O)	asiakasyrityksen johtajat	asiakasyrityksen johtajat	asiakasyrityksen johtajat
Kriteerit	(E)	ohjelma toteuttaa spesifikaation	ratkaisee ongelman tyydyttävällä tavalla	ohjelman käytön tarjoama hyöty toiminnalle

Tutkielman tavoitteena on toimintajärjestelmä, jossa asiakas ja kehittäjä yhdessä kehittäisivät asiakkaan toimintaa tukevaa ohjelmistoa. Järjestelmäteoreettisessa ohjelmankehityksessä asiakas asettaa vaatimuksia ohjelmalle, mutta hän ei osallistu varsinaiseen suunnitteluun tai kehitykseen. Sosioteknisessä ohjelmankehityksessä asiakas ja kehittäjät neuvottelevat ongelman määrityksestä ja ratkaisusta. Ohjelman kehitys perustuu suunnitelmaan, mutta asiakas ei voi suunnitteluvaiheen jälkeen enää arvioida ohjelmankehityksen seurauksia tai pyytää siihen muutoksia. Humanistisessa ohjelmankehityksessä asiakas on mukana myös ohjelmankehityksen aikana. Hän voi arvioida ohjelmankehityksen vaikutuksia tulevaan toimintaan ja ehdottaa tarvittavia muutoksia. Viimeinen ydinmääritelmä vastaa tutkielman tavoitetta.

3.4 Kolme ohjelmoinnin mallia

Edellisessä luvussa esiteltiin kolme ohjelmankehityksen toimintajärjestelmän ydinmääritelmää. Näistä Humanistinen ohjelmankehitys mahdollisti asiakkaan osallistumisen ohjelman kehittämiseen. Esittämällä kolme erilaista ydinmääritelmää on voitu havaita niiden eroja. Vastaavasti tässä luvussa luodaan ydinmääritelmien mukaisia ohjelmointimenetelmien malleja. Mallien luonnetta havainnollistetaan ohjelmointimenetelmien

teorioilla. Koska tutkielman lähtökohtana oleva idea koski olio-ohjelmointia, haetaan teoriat sen piiristä. Mallien luomiseen käytetään Soft Systems Metodologian kaavaa XYZ: järjestelmä joka tekee X:ää käyttämällä Y:tä saavuttaakseen Z:n (ks. Luku 2.2). X ja Z edustavat ydinmääritelmän muutosta. Y edustaa ohjelmointimenetelmän mallia.

Järjestelmäteoreettisessa ohjelmankehityksessä muutos on spesifikaatiosta ohjelmaksi. Asiakas voi vaikuttaa spesifikaatioon esittämällä ohjelmaan liittyviä vaatimuksia. Kuitenkaan hän ei voi vaikuttaa siihen millainen (design) ohjelma luodaan. Kehittäjät toteuttavat ohjelman spesifikaation perusteella. Myös lopputulos arvioidaan suhteessa tähän. Ydinmääritelmän muutos ja siinä käytettävä menetelmä voidaan esittää XYZ -kaavalla seuraavasti: Toteuta ohjelman spesifikaatio (X) kirjoittamalla tietokoneelle ohjelma (Y), joka laskee spesifikaation mukaisesti annetun syötteen tuloksen (Z). Ohjelmoinnin mallina on laskennan toteuttaminen. Jos ongelma on algoritmisesti ratkaistavissa, voidaan laskutoimitus toteuttaa kaikilla Turing-täydellisillä ohjelmointikielillä.

Olio-ohjelmoinnin alkuperäinen tavoite oli tarjota kieli tietokoneen ulkopuolisten järjestelmien kuvaamiseen ja simulointiin (Kristensen ym. 2007). Simuloinnin mahdollistamiseksi oli kielellä oltava ominaisuudet myös tietokoneen ohjaamiseen. Turingin kone (Turing 1936) on abstrakti algoritmia toteuttavan koneen malli. Sitä käytetään määritelmänä tietokoneen laskentakyvystä. Ohjelmointikielet, jotka voivat toteuttaa vastaavat laskutoimitukset, ovat yhtä ilmaisukykyisiä eli Turing-täydellisiä (Scott 2009). Ensimmäinen olio-ohjelmointikieli sisälsi Turing konetta vastaavat ominaisuudet. Turingin konetta käytetään Ohjelmointi laskennan toteuttamisena -mallin teoriana.

Sosioteknisessä ohjelmankehityksessä muutos on ongelmasta ohjelman tarjoamaksi ratkaisuksi. Asiakas määrittelee ongelman, jonka tietokoneohjelman on tarkoitus ratkaista. Neuvottelua käydään ongelman ja teknologisten vaatimusten välillä, ennen toteutusta. Lopuksi luotu ohjelma arvioidaan, mutta arviointi ei enää muuta toteutusta. Ydinmääritelmän sisältämä muutos ja siinä käytettävä menetelmä voidaan esittää XYZ -kaavalla seuraavasti: Ratkaise ongelma (X) luomalla tietokoneelle siitä malli (Y), jonka käsittely tuottaa halutun informaation tai palvelun (Z). Ohjelmoinnin mallina on todellisuuden mallintaminen. Skandinaavinen lähestymistapa olio-ohjelmointiin tulkitsee oh-

jelmoinnin todellisuuden mallintamiseksi (Sørgaard 1988).

Skandinaavinen olio-ohjelmointi sisältää sekä ohjelmoinnin että mallintamisen (Lehrmann Madsen & Møller-Pedersen 2010). Mallintaminen katsotaan olevan ohjelmointia korkeammalla abstraktiotasolla ja näin se on myös konkreettista ohjelman rakenteiden luomista. Olio-ohjelmoinnin käsitteet luokka ja olio tukevat tällaista ohjelmankehitystä. Luokkia voidaan käyttää mallin muodostamiseen olemassa olevan tai kuvitellun (esim. tulevan) todellisuuden ilmiöistä ja niiden välisistä suhteista. Oliot ovat näiden mallien fyysisiä esiintymiä (ovat olemassa konkreettisesti tietokoneen muistissa) ohjelman suorituksen aikana. Skandinaavista olio-ohjelmointia käytetään Ohjelmointi todellisuuden mallintamisena -mallin teoriana.

Humanistisessa ohjelmankehityksessä muutos on toiminnasta tietokonetuetuksi toiminnaksi. Ohjelmankehityksen lähtökohtana on asiakkaan kokema tarve. Asiakas osallistuu aktiivisesti ohjelman kehittämiseen. Ohjelman hyödyllisyys arvioidaan sen käytön kautta. Ydinmääritelmän sisältämä muutos ja siinä käytettävä menetelmä voidaan esittää XYZ -kaavalla seuraavasti: Tue toimintaa (X) luomalla ohjelma (Y), jonka käyttö mekanisoi toiminnan osan (Z). Ohjelmoinnin mallina on käytön mahdollistaminen. Olio-ajattelussa (West 2004) käyttäjän ajatellaan olevan vuorovaikutuksessa tietokoneohjelman sisältämien olioiden kanssa. Oliot voidaan tulkita työkaluiksi (Züllighoven 2005) tai kokonaisuutena työympäristöksi (West 2004).

Olio-ajattelussa keskitytään olioiden tarjoamiin tietoihin ja palveluihin (West 2004). Käyttäjä on yhteydessä käyttöliittymän kautta ohjelman olioihin. Ne hyödyntävät vuorostaan muiden olioiden palveluja. Keskeiset oliot saadaan kontekstista (ongelmasta), johon tietokoneohjelma on tarkoitettu ratkaisuksi. Olio-ajattelu on ongelmakeskeistä, tietokoneen ohjelmoinnissa yleisen ratkaisukeskeisen ajattelun sijaan (ibid.). Oliot tarjoavat yhteisen kielen kehittäjän ja käyttäjän välille (West 2004, Coggins 1996). Olio-ajattelu siirtää huomion olioiden kuvaamista asioista olioiden tarjoamiin palveluihin. Sitä käytetään Ohjelmointi käytön mahdollistamisena -mallin teoriana.

Tutkielman lähtökohtana oli idea, että olio-ohjelmointi riittävän korkealla abstraktiota-

solla mahdollistaisi asiakkaan osallistumisen ohjelman kehittämiseen. Ohjelmankehityksen toimintajärjestelmissä (ks. ed. luku) todettiin eroja asiakkaan osallistumisen tuessa. Niiden ydinmääritelmien muutokset edustivat eri abstraktiotasoja. Tässä luvussa luotiin ohjelmointimenetelmien malleja näille abstraktiotasoille (ks. Taulukko 3). Laskennan toteuttamisen kohderyhmänä ovat kehittäjät. Todellisuuden mallintaminen perustuu asiakkaan kokemaan todellisuuteen, joten tähän hän voi ottaa kantaa. Kuitenkin vasta käytön mahdollistamisessa yhdistyy ongelma ja tietotekniikan tarjoama ratkaisu. Asiakas voi tällä tasolla vaikuttaa siihen millainen ohjelma tehdään ja näin myös siihen millaiseksi tuleva toiminta muodostuu.

Taulukko 3: Kolme ohjelmoinnin mallia. Mallien lisäksi esitetään toimintajärjestelmän ydinmääritelmä, ydinmääritelmän muutos jonka mallin on ajateltu mahdollistavan sekä mallin teoria.

Ohjelmoinnin malli	Toimintajärjestelmän ydinmääritelmä	Muutos	Teoria
laskennan toteuttaminen	Järjestelmäteoreettinen ohjelmankehitys	spesifikaatio => ohjelma	Turingin kone
todellisuuden mallintaminen	Sosiotekninen ohjelmankehitys	ongelma => ratkaisu	Skandinaavinen olio-ohjelmointi
käytön mahdollistaminen	Humanistinen ohjelmankehitys	toiminta => tietokonetuettu toiminta	Olio-ajattelu

Ohjelmoinnin mallit määrittelevät millä abstraktiotasolla ohjelmointimenetelmän käsitteet ovat. Käytön mahdollistaminen vastaa tutkielman tavoitteena olevaa tasoa. Asiakas voi siinä kuvitella mitä ohjelma mahdollistaisi ja mitkä olisivat sen vaikutukset toimintaan. Tietokoneohjelman mahdollisuuksia ei kuitenkaan voida arvioida ilman käytön kohdetta. Myös todellisuuden mallintamisen malli on tarpeellinen (ks. luku 3.1.). Kehittäjän näkökulmasta ohjelma on voitava toteuttaa, joten tarvitaan myös laskennan toteuttamisen malli. Kaikkia em. malleja tarvitaan, yhdessä ne muodostavat asiakkaan osallistumisen mahdollistamisen viitekehyksen. Ero perinteiseen ohjelmankehitykseen on, että laskennan toteuttamisen lisäksi myös asiakkaan ymmärtämät mallit esitetään eksplisiittisesti lähdekoodissa.

On mahdollista, että myös muissa paradigmoissa, kuin olio-ohjelmointi, on viitekehystä vastaavia menetelmiä. Niitä pyritään siksi löytämään myös kolmesta muusta suositusta ohjelmointiparadigmasta (Bal & Grune 1994): imperatiivinen, funktionaalinen ja logiikka. Oletuksena on, että saman ohjelmointiparadigman sisällä voidaan yhdistää eri ohjelmoinnin mallien mukaiset menetelmät. Seuraavassa luvussa pyritään ohjelmoinnin teorioiden avulla löytämään viitekehyksen mukaisia menetelmiä. Menetelmien kuvauksissa pyritään tuomaan esille kehittämisen abstraktiotaso, sen esitys lähdekoodissa ja menetelmän suhde käytettyyn teoriaan.

4 VIITEKEHYKSEN MUKAISET OHJELMOINTIMENETELMÄT

Edellisessä luvussa luotiin tutkielman tavoitteena oleva viitekehys. Viitekehys koostuu kolmesta mallista, jossa ohjelmointi tulkitaan laskennan toteuttamisena, todellisuuden mallintamisena ja käytön mahdollistamisena. Viimeinen edustaa tavoiteltua abstraktiotasoa, jossa asiakas voi osallistua ohjelmankehitykseen. Mallit edustavat myös ohjelmankehityksen evoluutiota, jossa uudempi tapa korvasi aikaisemman, mutta viitekehyksen malleja luotaessa kävi ilmi, että teknisesti on kuitenkin huomioitava kaikki kolme. Tutkielman oletuksena on, että löytyy myös esimerkkejä viitekehyksen mukaisista ohjelmointimenetelmistä. Näiden tunnistamista helpottamaan valittiin kolmeen ohjelmoinnin malliin sopivaa teoriaa: Turingin kone, Skandinaavinen olio-ohjelmointi ja Olio-ajattelu.

Alun perin Turingin kone (TK) luotiin matematiikan perusteisiin liittyvän päätösongelman (Entscheidungsproblem) tutkimiseksi (Turing 1936). Päätösongelmassa haetaan mekaanista matemaattista proseduuria, algoritmia, jolla voidaan päättää onko lause todistettavissa vai ei. Turing todisti tällaisen algoritmin mahdottomuuden osoittamalla sen johtavan ristiriitaan (sama lause on sekä tosi että epätosi). Todistukseen hän tarvitsi täsmällisen tavan esittää algoritmeja, jota varten hän kehitti TK:en. TK on formaali kieli algoritmeja suorittavan koneen toiminnan kuvaamiseen. Vuonna 1936 julkaistiin useita kieliä, jolla voitiin määritellä algoritmeja (Gandy 1988). Lambda-laskenta (Church 1936), laskennan malli (Post 1936), rekursiiviset funktiot (Kleene 1936) ja TK osoittautuivat kaikki ekvivalenteiksi (Robinson 2000). TK on kuitenkin saanut erityisen aseman sen käsitteiden ja toiminnan selkeyden takia (Soare 2016).

Turingin konetta (TK) käytetään määritelmänä ohjelmointikielten laskentakyvystä (Scott 2009). Se määrittää mitä niillä voi laskea ja mitä ei. Koska kaikki nk. yleiskäyttöiset ohjelmointikielet pystyvät samoihin laskutoimituksiin, ei tässä suhteessa ole väliä minkä kielen valitsee. Vaikka ohjelmointikielet ovat yhtä ilmaisukykyisiä, eroavat ne kuitenkin eri ongelmien ratkaisujen yksinkertaisuudessa (Perlis 1982). Tässä luvussa esitellään ohjelmoinnin mallien esimerkkejä neljästä merkittävimmästä paradigmasta

(Bal & Grune 1994): imperatiivinen, funktionaalinen, logiikka ja olio. Paradigmat tulkitaan tässä tutkielmassa, Bertrand Meyerin (2009) esimerkkiä seuraten, ohjelmointimenetelmiksi.

Skandinaavinen olio-ohjelmointi sai alkunsa järjestelmien simuloimisesta. Tarvittiin kieli, jolla voitiin kuvata simuloitava järjestelmä ja määritellä miten se suoritetaan tietokoneella. Skandinaavinen olio-ohjelmointi kehittyi kolmen kielen kautta ja samalla simuloinnin tavoite muuttui ohjelmoinniksi mallintamisena. SIMULA I (SIMUlation LAnguage) oli järjestelmien simuloimiseen tarkoitettu ohjelmointikieli (Dahl & Nygaard 1966). SIMULA 67 (käytetään myös pelkkää SIMULA nimeä) oli yleiskäyttöinen ohjelmointikieli, jonka erityispiirteenä oli tuki sovellusaluekielille (Dahl ym. 1968). Yksi tällainen sovellusalue oli järjestelmien simulointi. SIMULA 67 oli ensimmäinen ohjelmointikieli, jossa käytettiin olion ja luokan käsitteitä. Tästä alkoi olio-ohjelmointi ja SIMULA 67 perillisinä voidaan pitää sekä C++ (Stroustrup 1994) että Smalltalk (Kay 1993) ohjelmointikieliä, joista muut olio-kielet ovat ottaneet mallia.

SIMULA I ja SIMULA 67 olivat ALGOL 60 kielen laajennuksia (Dahl & Nygaard 1966, Dahl ym. 1968). BETA kielen kehityksen tavoitteena oli luoda SIMULA kieliä yksinkertaisempi ohjelmointikieli, joka säilyttäisi erityispiirteet, mutta korvaisi ALGOL 60:n rakenteet. Uudella kielellä oli kaksi keskeistä piirrettä: koko ohjelmointikieli perustui kuvion (engl. pattern) käsitteeseen ja ohjelmointi tulkittiin mallintamiseksi (Kristensen ym. 2007). Tutkielmassa luodun viitekehyksen toisena mallina on ohjelmointi mallintamisena. SIMULA-kieliä ei enää käytetä, mutta mallintamisen esimerkkejä voidaan hakea nykyisistä olio-ohjelmointikielistä tai muista ohjelmointiparadigmoissa.

Olio-ajattelussa (West 2004) maailma nähdään koostuvan olioista. Ohjelmankehityksellä pyritään luomaan olioita, jotka tukevat toimintaa tietokoneen ulkopuolella. David West (2004) asettaa olio-ajattelun keskeiseksi XP (Extreme Programming) -menetelmän osaksi. Hän myöntää kuitenkin, että XP-menetelmän (Beck 1999) yhteydessä ei erityisesti mainita olio-ajattelua ja olioihin viitataan yleisestikin vähän. Näin on ehkä oikeampi nähdä XP-menetelmä olio-ajattelun innoittajana tai sovelluskohteena. Olioajattelua ja XP -menetelmää yhdistävät lähtökohta, arvot ja käytännöt (West 2004). Yhteinen

lähtökohta liittyy XP-menetelmän luojaan Kent Beckiin, jonka olio-ohjelmointiin liittyvää työtä hyödynnetään olio-ajattelussa. Arvot liittyvät kehittäjien ja asiakkaan yhteistyötä tukeviin ohjelmankehitystapoihin, jotka sopivat hyvin olio-ajatteluun. Käytännöt ovat teknisiä menetelmiä, joita voidaan käyttää myös olio-ohjelmoinnissa.

Ketterien menetelmien, kuten XP, vastakohtana ovat prosessikeskeiset ohjelmankehitysmenetelmät (Boehm & Turner 2003). Vastaavasti olio-ajattelun vastakohtana voidaan pitää tietokonekeskeistä (ks. luku 3.1.) ajattelua (West 2004). Tietokonekeskeisessä ajattelussa ongelmaa tarkastellaan ratkaisun toteutukseen käytettävien keinojen kautta. Keinot ohjaavat mitä asioita ongelmasta valitaan ja miten ne tulkitaan. Painopiste on tulevassa ohjelmatuotteessa ja sen suunnittelussa. Olio-ajattelussa pyritään samanlaiseen asiakaskeskeisyyteen ja muutoshallintaan kuin ketterissä menetelmissä. Erityisesti tulkitaan olioiden mallintavan ongelma-aluetta, ei tulevaa ohjelmaa. Ratkaisu syntyy ohjelman olioiden ja sen ulkopuolella olevien olioiden (esim. ihmisten) välisestä viestinnästä.

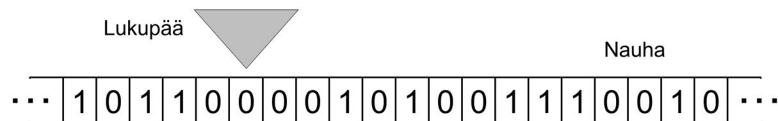
Tässä luvussa esitellään kolmen ohjelmoinnin mallin esimerkkejä neljästä suosituimmasta ohjelmointiparadigmasta.

4.1 Ohjelmointi laskennan toteuttamisena -malli

Turingin kone (Turing 1936) toimii määritelmänä ohjelmointikielten laskentakyvystä (Scott 2009). Alan Turing (1936) otti koneen kehittämisen lähtökohdaksi laskemisen ruutupaperille, jota hän yksinkertaisti kunnes laskenta voitiin tehdä täysin mekaanisesti. Paperi voitiin korvata riittävän pitkällä ruutuihin jaetulla nauhalla. Jokainen ruutu voi sisältää yhden merkin ja kone laskee ruutu kerrallaan. Ihminen voi luottaa ongelman ymmärrykseensä laskiessaan, mutta kone tarvitsee yksiselitteiset säännöt. Sääntöjen suorittaminen alkaa ensimmäisestä säännöstä ja ensimmäisestä ruudusta. Jokainen sääntö viittaa myös seuraavaan sääntöön, paitsi sääntö, joka määrittelee laskutoimituksen valmiiksi. Sääntöjen valintaan vaikuttaa myös kohdalla olevan ruudun sisältö ja eri sisältöjä varten on säännöistä omat versiot. Numeroiksi valittiin binääriluvut, koska niiden käsittelyyn tarvitaan vähemmän sääntöjä. Laskemisen yksinkertaistuksesta syntyi kuvitteellinen äärettömällä nauhalla laskeva kone, nk. Turingin kone (ks. Kuvio 15).

Säännöt

Tila	Luettu merkki	Kirjoitettava merkki	Siirtyminen Vas/Oik/Paik	Seuraava tila
T_1	0	0	Oik	T_1
T_1	1	1	Oik	T_2
T_1	$Loppu$	Ei	$Paik$	$Pysähdy$
T_2	0	0	Oik	T_2
T_2	1	1	Oik	T_3
T_2	$Loppu$	$Pariton$	$Paik$	$Pysähdy$
T_3	0	0	Oik	T_3
T_3	1	1	Oik	T_2
T_3	$Loppu$	$Parillinen$	$Paik$	$Pysähdy$



Kuvio 15: Alan Turingin (1936) keksimä kuvitteellinen tietokone, joka voi toteuttaa kaikki laskettavissa olevat laskutoimitukset. Se koostuu äärettömästä nauhasta, joka on jaettu ruutuihin ja jokainen ruutu voi sisältää yhden merkin. Tässä esimerkissä käytetään 0, 1 Loppu, Pariton ja Parillinen merkkejä. Koneessa on lukupää, joka lukee, kirjoittaa ja liikkuu nauhalla. Koneen toiminnan määrittelevät säännöt, jotka ohjaavat lukupään toimintaa.

Turingin koneen säännöt muodostavat ohjelmointikielen. Ohjelmointikielillä on sekä matemaattinen että fyysinen merkitys (Turner 2014, Irmak 2012, Colburn & Shute 2007, Colburn 2000, Moor 1978). Matemaattinen malli määrittelee mitä laskutoimitus merkitsee ja fyysinen toteutus mahdollistaa laskennan automatisoinnin sekä tietokoneen laitteiston hyödyntämisen. Turingin kone on täysin matemaattinen konstruktio, vaikkakin käyttää konetta laskennan mallina. Samalla säännöllä ja ruudun sisällöllä, on tulos aina täsmälleen sama. Fyysisen tietokoneen toiminnan ohjaaminen tuo mukanaan monimutkaisuuden ja enää ei ole yhtä helppoa päätellä mikä on tulos. Toisaalta, laitteen ohjaaminen tuo mahdollisuuksia, joita matemaattisella mallilla ei ole. Ohjelmointikielen käsitteet voivat joko olla lähempänä matemaattista mallia tai tietokoneen rakennetta.

Imperatiivisten ohjelmointikielten malli perustuu nykyisten tietokoneiden yleiseen rakenteeseen, von Neumannin (1945) ehdottamaan arkkitehtuuriin. Sen tavoite oli yksinkertaistaa tietokoneen ohjaaminen korvaamalla fyysinen, kytkimillä ja johdotuksilla toteutettu, konekielen komentoihin perustuvalla ohjelmalla. Komennot ja käsiteltävä tieto sijaitsee tietokoneen muistissa. Tietokoneen toiminta perustuu sykliin, jossa komento noudetaan muistista, tulkitaan (haetaan esim. käsiteltävä tieto muistista), suorite-

taan ja aloitetaan alusta siirtymällä seuraavaan komentoon. Korkeantason imperatiiviset ohjelmointikielet abstrahoivat tämän prosessin poistamalla tarpeen käsitellä muistia eksplisiittisesti (ks. Kuvio 16). Ohjelma kirjoitetaan peräkkäisille riveille ja eteneminen tapahtuu ylhäältä alas, jollei ohjelman tekstissä viitata aikaisempaan kohtaan. Viittaukset perustuvat nimiin tai tekstin rakenteeseen. Tietoa tallennetaan muistiin ja haetaan sieltä käyttäen muistia edustavia nimiä, nk. muuttujia.

```
int gcd(int a, int b) {
     while (a != b) {
          if (a > b) a = a - b;
          else b = b - a;
     }
     return a;
}
```

Kuvio 16: Suurimman yhteisen tekijän laskeva funktio toteutettuna C-ohjelmointikielellä (Scott 2009). Funktiolle annetaan syötteenä kaksi kokonaislukua a ja b. Jos a ja b ovat yhtä suuria, palautetaan a suurimpana yhteisenä tekijänä. Jos a ja b ovat eri suuria, siirrytään while-silmukkaan, jossa verrataan lukuja keskenään. Suuremmasta luvusta vähennetään pienempi luku. Silmukkaa suoritetaan kunnes molemmat luvut ovat samansuuruisia. Silmukan suoritus pysähtyy ja luku a palautetaan suurimpana yhteisenä tekijänä.

Vuonna 1957 julkaistu Fortran oli ensimmäisiä imperatiivisia ohjelmointikieliä, joka keskittyi laskemiseen (Backus 1978). C-ohjelmointikielen, ensimmäinen versio 1972, tavoitteena oli tietokonelaitteiston ohjaaminen (Ritchie 1996). Molemmat kielet ovat yhä käytössä ja ne kuvaavat imperatiivisten kielten kaksinaista roolia. Proseduurit ja funktiot mahdollistavat ohjelmoinnin abstraktiotason korottamisen (Jipping & Bruce 2014). Imperatiivisiin ohjelmointikieliin on tuotu myös uusia käsitteitä, kuten oliot (Stroustrup 1984) ja ensiluokkaiset funktiot (Gosling ym. 2015). Muistin käsittely ja sen ohjaaminen on kuitenkin yhä imperatiivisen ohjelmoinnin ytimessä. Malli on konekeskeinen, korkeammasta abstraktiotasosta huolimatta, joka mahdollistaa tehokkaiden ratkaisujen hakemisen ja tietokonelaitteiston luontevan ohjaamisen. Nykyisistä ohjelmointikielistä imperatiiviset kielet ovat eniten käytettyjä (Jipping & Bruce 2014, Scott 2009, Bal & Grune 1994).

Funktionaalisten ohjelmointikielten mallina on Alonzo Churchin (1936) kehittämä Lambda-laskenta (Hudak 1989). Lambda-laskenta on formalismi algoritmien esittämi-

seen. Mallina siinä on matemaattinen funktio, jonka otsikon määrittelemät symbolit korvataan funktion syötteillä. Myös tiedot esitetään funktiona esim. luonnolliset luvut alkavat nolla funktiosta ja sitä seuraavat luvut saadaan seuraajafunktiolla. Laskeminen perustuu funktioiden sijoittamiseen funktioiden sisälle. Toisto voidaan toteuttaa käyttäen rekursiota, joka tekee Lambda-laskennasta ilmaisukyvyltään Turingin konetta vastaavan. Käytännössä funktionaalisissa ohjelmointikielissä luvut lasketaan, mutta myös näissä kerran ilmoitettu symbolin arvo on määrittelyalueen sisällä sama. Laskutoimituksessa lähtöarvot pysyvät ja tuloksena on uusi arvo. Tietokoneen toimintaa ohjaavat matematiikan lait (ks. Kuvio 17).

```
(define gcd
     (lambda (a b)
       (cond   ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a)))))
```

Kuvio 17: Suurimman yhteisen tekijän laskeva funktio gcd, toteutettuna Scheme-ohjelmointikielellä (Scott 2009). Define avainsana asettaa termin (anonyymi funktio) nimeksi gcd. Anonyymifunktiolla on kaksi parametria. Sen sisältämä ehto-funktio vertaa arvoja kolmeen ehtoon. Ensimmäisessä ehdossa arvot ovat yhtä suuria, jolloin ensimmäinen arvo edustaa suurinta yhteistä tekijää. Toisessa ehdossa ensimmäinen arvo on toista suurempi. Kolmas ehto on tapausta varten, jossa edelliset ehdot eivät toteutuneet. Tässä sen merkitys on, että toinen annetuista arvoista on ensimmäistä suurempi. Toisen tai kolmannen ehdon toteutuessa ehto-funktion tuloksena on gcd-funktio, jonka parametreina ovat isomman ja pienemmän arvon erotus sekä pienempi arvo. gcd-funktiota lasketaan rekursiivisesti kunnes parametrit ovat yhtä suuria, jolloin ensimmäinen ehto täyttyy.

John McCarthyn (1960) luoma LISP on ensimmäinen funktionaalinen ohjelmointikieli ja toiseksi vanhin (Fortran on vanhempi) yhä käytössä oleva ohjelmointikieli (McCarthy 1981). LISP on proseduraalinen kieli ja esim. rekursioita ohjataan ehtolauseella. Toinen funktionaalisten ohjelmointikielten toimintamalli sai alkunsa ML-kielestä (Gordon ym. 1978), jossa laskutoimituksen eteneminen perustuu hahmon tunnistukseen (pattern matching). Funktioita voidaan luoda erilaisia tapauksia (syötteiden arvoja) varten ja erityisesti rekursion päättyminen on ratkaistu funktiolla, joka ei enää kutsu itseään. Hahmontunnistuksen hyödyntäminen mahdollistaa ohjelmien kirjoittamisen, jotka muistuttavat matemaattista määritelmää. LISP ja ML-kielissä on imperatiivisia komentoja tietoko-

neen ohjaamiseen, mutta esim. Haskell (Hudak ym. 2007, Hudak ym. 1992) ohjelmointikielessä nämäkin on pyritty upottamaan matemaattiseen kontekstiin. Funktionaalisten kielten etuna on korkeampi modulaarisuus, johtuen mm. sivuvaikutuksettomien funktioiden helposta yhdistelystä (Hu ym. 2015, Hughes 1989). Funktionaalisten ohjelmointikielten suosio on kasvanut ja moneen ohjelmointikieleen on lisätty näiden piirteitä (Hu ym. 2015).

Nykyinen logiikkaohjelmointi syntyi luonnolliseen kieleen perustuvien käyttöliittymien tutkimuksesta (Colmerauer ja Roussel 1996). Käyttöliittymän toteutuksessa oli selvitettävä mitä syötteenä saatu lause merkitsee ja siitä päätellä sopiva vastaus. Lauseen merkityksen selvittämiseksi on tunnistettava sen sanat ja näiden muodostamat rakenteet (Indurkhya 2014). Vastaus perustuu yleisesti syötteen rakenteesta johdettuun vastineeseen sekä erityisesti sanojen antamiin yksityiskohtiin. Logiikkaohjelmoinnissa yhdistyi luonnollisen kielen käsittely ja teoreemojen automaattinen todistaminen (Colmerauer ja Roussel 1996). Myös logiikkaohjelmoinnissa tunnistetaan lauseen sanoja ja rakenteita, mutta tavoitteena on selvittää onko lause totta tai millä edellytyksillä se voisi olla (ohjelman puitteissa). Teoreemojen automaattinen todistaminen tarjoaa tähän mekanismin ja se vastaa luvun muiden paradigmojen laskemista.

Logiikkaohjelmoinnissa (esim. Bramer 2013) syötteenä saatu lause on totta, jos se voidaan muodostaa ohjelman lauseista. Ohjelman lauseet voivat olla tosiasioita tai sääntöjä. Tosiasia voi olla vakio (jokin merkkijono) tai predikaatti (vakio), joka määrittelee yhden tai useamman vakion yhdistelmän (predikaatin jälkeen suluissa). Näiden merkitys perustuu käyttäjän tulkintaan. Säännön muodostaa pää ja runko. Pää on yksittäinen lause, joka on tosi, jos rungon lauseet ovat tosia. Rungon lauseet ovat tosia, jos ne löytyvät ohjelmasta. Säännöt voivat sisältää myös muuttujia ja näille on löydyttävä vastaava vakio ohjelman lauseista. Muuttujan arvon asettaminen vaikuttaa sen kaikkiin esiintymiin säännössä. Ohjelman suorituksessa tulkki käy ohjelmaa lävitse hakien lausetta, joka vastaa syötettä (ks. Kuvio 18). Jos löytyy vastaava tosiasia, on syöte totta, jos taas löytyy vastaava sääntö, haetaan sen rungon lauseille vastineita ohjelmasta. Syötteen ollessa predikaatti voi se määritellä millaisia muuttujia lauseissa on oltava. Jos lauseen rungon määriteltyjen muuttujien yhdistelmä johtaa ristiriitaan, niin palataan taaksepäin ja ko-

keillaan uudella yhdistelmällä. Hakua ja palaamista jatketaan, kunnes kaikki yhdistelmät on käyty lävitse. Jos syötettä vastaavaa yhdistelmää ei ole löytynyt, on lause epätosi.

```
gcd(A, B, G) :- A = B, G = A.
gcd(A, B, G) :- A > B, C is A - B, gcd(C, B, G).
gcd(A, B, G) :- B > A, C is B - A, gcd(C, A, G).
```

Kuvio 18: Kahden luvun suurimman yhteisen tekijän hakuun liittyvät lauseet, toteutettuna Prolog-ohjelmointikielellä (Scott 2009). Suurin yhteinen tekijä saadaan kyselyllä gcd(A, B, G), jossa A sekä B ovat kokonaislukuja ja G edustaa suurinta yhteistä tekijää, jota ei vielä tiedetä. Vastausta haetaan käymällä lauseet lävitse ylhäältä alas. Pyritään löytämään kyselyä vastaava lause (tässä esimerkissä lauseet ovat sääntöjä). Jos lause vastaa kyselyä, kaikille muuttujilla on voitu antaa arvot, niin on löydetty vastaus. Jos lause vastaa kyselyä osittain, tehdään uusi haku alusta käyttäen osatulosta. Ensimmäinen sääntö määrittelee tilanteen, jossa suurin yhteinen tekijä on löytynyt. Tällöin A on yhtä suuri kuin B, jolloin suurin yhteinen tekijä G on yhtä suuri kuin A. Toinen lause kuvaa sääntöä, jossa A on suurempi kuin B. Jos näin on, niin C saa arvoksi A:n ja B:n erotuksen. Tehdään uusi haku jossa gcd:n arvoina on C, B ja G. Kolmas lause kuvaa sääntöä, jossa B on suurempi kuin A. Jos näin on, niin C saa arvoksi B:n ja A:n erotuksen. Tehdään uusi haku jolloin gcd:n arvoina on C, A ja G. Hakuja jatketaan kunnes suurin yhteinen tekijä on löytynyt.

Muista tässä luvussa esitetyistä ohjelmointiparadigmoista poiketen, logiikkaohjelmoinnissa ei määritellä algoritmia eksplisiittisesti vaan ohjelmistotulkki muodostaa sen tosiasioiden ja sääntöjen ketjun avulla. Ohjelmat voivat näin olla lyhyempiä ja selkeämpiä (Bratko 2011), koska ohjelman suorituksen kulkua ei tarvitse eksplisiittisesti määritellä. Matemaattinen malli on logiikan, mutta ohjelman lauseiden peräkkäinen suoritus viittaa tietokoneeseen. Maarit Harsu (2005) kritisoi eksplisiittistä suoritusjärjestystä sekä muita logiikkaan kuulumattomia ominaisuuksia, pitäen näitä puutteina. Logiikkaohjelmointi tarjoaa kuitenkin tehokkaan tavan käsitellä monimutkaisuutta syötteessä tai syötteen tulkinnassa. Käsittelyä helpottaa, ongelman ratkaiseminen asioiden välisiä suhteita määrittelemällä ja hakuun perustuvalla laskennalla (Scott 2009).

Olio-ohjelmointi kehittyi järjestelmien simuloimiseen tarkoitetusta ohjelmointikielestä, SIMULA I (Dahl & Nygaard 1966). Simulaatio muodostui pseudorinnakkaisista prosesseista (proseduureista), joiden suoritus voitiin pysäyttää ja siirtää toiselle prosessille. Muilla prosesseilla oli pääsy pysäytetyn prosessin muuttujiin. Prosessi määriteltiin toi-

mintomääreellä, josta voitiin luoda useita prosesseja. SIMULAn seuraavassa versiossa prosessi ja toimintomääre nimettiin olioksi ja luokaksi. Vallitsevassa olio-ohjelmoinnin tulkinnassa (Armstrong 2006) on keskitytty muihin piirteisiin ja rinnakkaisuus on jätetty pois. Imperatiivissa olio-kielissä, kuten C++, olio on ulkopuolisten operaatioiden kohteena, jotka muuttavat sen arvoja (Jipping & Bruce 2014). Olio-laskennan (Abadi & Cardelli 1996) malli perustuu funktionaaliseen ohjelmointiin, jossa rakenteelliset piirteet edustavat olio-ohjelmointia. Poikkeuksena em. on Smalltalk (Kay 1993), jonka idea perustuu ajatukseen verkottuneista laskennan yksiköistä (ks. Kuvio 19).

```
gcd: other
(self = other)
     ifTrue:  [↑ self]
(self < other)
     ifTrue:  [↑ self gcd: (other - self)]
     ifFalse: [↑ other gcd: (self - other)]
```

Kuvio 19: Kahden luvun suurimman yhteisen tekijän laskemista kuvaava metodi, toteutettuna Smalltalk-ohjelmointikielellä (muokattu Scott 2009). gcd on kokonaisluku-olion metodi, jolla on yksi parametri (other). Ensimmäinen viesti (=) isäntäoliolle (self) testaa onko syötteenä saatu olio (other) yhtä suuri. Vastauksena on viittaus totuusarvo-olioon (tosi tai epätosi). Jos totuusarvo-olio edustaa arvoa tosi, niin se suorittaa ifTrue: -viestin parametrina olevan ohjelmalohkon. Ohjelmalohko palauttaa (↑) tässä tapauksessa viittauksen isäntäolioon (self) ja metodin suoritus päättyy. Toinen viesti (<) isäntäoliolle testaa onko syötteenä saatu olio (other) suurempi. Vastauksena on totuusarvo-olio, jolle lähetetään ensin viesti ifTrue:, jos vastaavaa metodia ei ole eli kyseessä on totuusarvo epätosi, niin lähetään viesti ifFalse:. Viestin parametrina on ohjelmalohko, joka sisältää palautuksen (↑) ja viestin. ifTrue: tapauksessa lähetetään isäntä-oliolle viesti gcd, jonka parametrina on viittaus syötteenä saadun olion ja isäntä-olion erotuksen muodostamaan olioon. ifFalse: tapauksessa lähetetään viesti syötteenä saadulle oliolle ja parametrina viittaus isäntä-olion ja syötteenä saadun olion erotuksen muodostamaan olioon. Rekursio päättyy kun arvot ovat yhtä suuret, jolloin palautetaan viittaus viimeiseen olioon kutsuketjua pitkin.

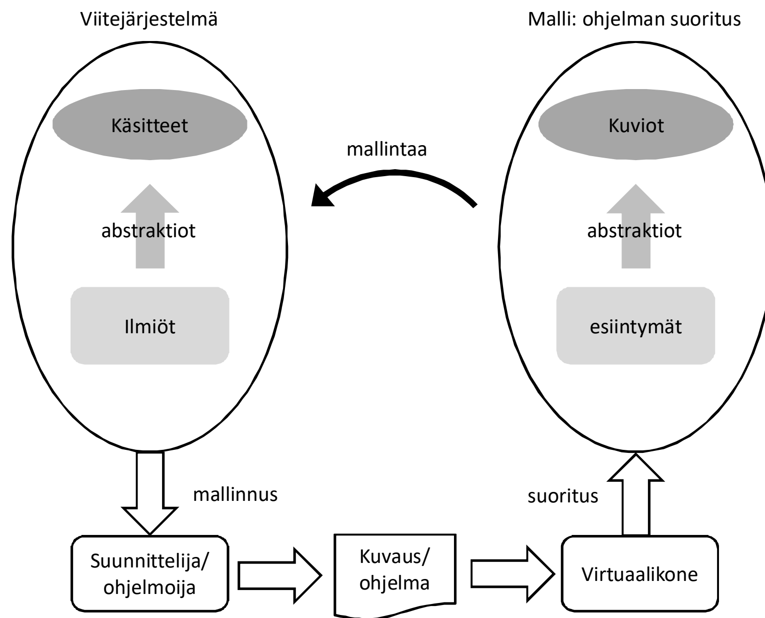
Smalltalk oli ensimmäinen yleisessä käytössä oleva puhdas olio-ohjelmointikieli (Kay 1993). Se perustui ajatukseen, että tietokoneohjelma voidaan kuvata pelkästään olioista koostuvana rekursiivisena rakenteena, jonka toiminnallisuus perustuu olioiden väliseen viestintään. Olio edustaa tietokonetta, jonka seurauksena jokaisella ohjelman oliolla on käytettävissään kaikki tietokoneen ominaisuudet. Olio-ohjelmointi on tämän tulkinnan mukaan tietokoneen ohjaamista korkeammalla abstraktiotasolla, mutta malli ei ole fyy-

sinen (laitteisto) tai matemaattinen. Siinä ei määritellä miten laskenta toteutetaan tai mitä se on, vaan laskentaa edustaa palvelujen pyytäminen ja tarjoaminen. Smalltalk-kielestä on monta versiota ja ainut merkittävä toinen tämän mallin mukainen ohjelmointikieli on Self (Ungar & Smith 1987). Kielenä Smalltalk on marginaalinen, mutta se on vaikuttanut olio-ohjelmoinnin leviämiseen, ja sen ohjelmointitapaa voidaan noudattaa muissa olio-kielissä, vaikka niiden laskennan malli ei sellaiseen ohjaa.

Ohjelmointikielillä on sekä matemaattinen että fyysinen tulkinta, riippumatta siitä kumpaa niiden syntaksi lähemmin muistuttaa. Turnerin (2014) mukaan, ohjelman muodostavien käsitteiden matemaattinen merkitys mahdollistaa ohjelman vertaamisen annettuun spesifikaatioon. Tutkielmassa on käytetty esimerkkinä kahden kokonaisluvun suurimman yhteisen tekijän laskevaa kaavaa, joka on toteutettu eri paradigmojen kielillä (ks. edellä Kuviot 16-19). Vaikka merkitys on kaikissa sama, ovat toteutukset erilaisia. Ohjelmointikielet ovat laskennan määrittelyn formalismeja, jolloin niiden käsitteet liittyvät siihen. Ne kaikki toteuttavat Turingin konetta vastaavat ominaisuudet.

4.2 Ohjelmointi todellisuuden mallintamisena -malli

Ohjelmointi todellisuuden mallintamisena -mallissa todellisuus esitetään eksplisiittisesti ohjelman lähdekoodissa. Käytettyjen käsitteiden ja niiden suhteiden tulisi vastata asiakkaan käsitystä. Todellisuuden mallintamisen teoriaksi valittiin Skandinaavinen olio-ohjelmointi, jonka avulla pyritään tunnistamaan vastaavia ohjelmointimenetelmiä. Skandinaavinen olio-ohjelmointi kehittyi kolmen ohjelmointikielen kautta, näistä viimeisessä, BETA-kielessä, tulkitaan ohjelmointi mallintamisena. Tavoitteena oli käsiteltävän ongelman ymmärtäminen: ”Ymmärtäminen ohjelmoinnissa tarkoittaa sovelluskohteen mallintamista, muuten ohjelmointi redusoituu pelkäksi koneen ohjaamiseksi” (Lehrmann Madsen & Møller-Pedersen 2010). Mallintamisen kohteeksi valittu todellisuuden osa nähtiin viitejärjestelmänä, jota tietokoneohjelma kuvasi (ks. Kuvio 20). Mallina pidettiin ohjelman suoritusta, ei sitä kuvaavaa lähdekoodia.



Kuvio 20: Viitejärjestelmän, mallijärjestelmän ja sitä toteuttava ohjelman suhde (Lehrmann Madsen & Møller-Pedersen 2010). Viitejärjestelmä on kohteen abstrakti kuvaus, johon on valittu ne piirteet joita katsotaan tärkeiksi. Mallijärjestelmä on viitejärjestelmän kuvauksen formalisointi. Ohjelma on mallijärjestelmän toteutus, jossa formaalikuvaus on muutettu suoritettavaksi ohjelmaksi. Vasta ohjelman suoritus mallintaa viitejärjestelmää.

BETA-kielessä mallintaminen sisälsi (Lehrmann Madsen ym. 1993): viitejärjestelmän, mallijärjestelmän ja sen kuvauksen. Näitä vastaavat ohjelmankehityksessä analyysin, suunnittelun ja toteutuksen vaiheet. Analyysivaiheen tavoitteena on sovelluskohteen ymmärtäminen. Siihen kuuluu viitejärjestelmän kuvaaminen käyttäen todellisuuden käsitteitä. Käsitteet ovat abstraktioita, koska niihin valitaan vain ne piirteet, jotka katsotaan keskeiseksi. Viitejärjestelmää ei ole olemassa, vaan se on valitun näkökulman mukainen tulkinta todellisuudesta. Suunnitteluvaihe määrää millainen ohjelmasta tulee, muuttamalla viitejärjestelmän piirteet formaaliksi mallijärjestelmän määritelmäksi. BETA-kielen käsitteitä käytettäessä suunnittelun tuloksena on abstraktiohjelma. Toteutusvaiheessa täydennetään käsitteitä, jotta ne voidaan suorittaa tietokoneella. BETA-kielessä mallintaminen oli huomioitu kieltä suunniteltaessa, muissa ohjelmointikielissä mallintaminen voidaan toteuttaa erilaisilla menetelmillä.

Jackson System Development (JSD; Jackson 1983) on menetelmä järjestelmien kehittämiseen, joka sai alkunsa rakenteellisen ohjelmoinnin menetelmästä Jackson Structured Programming (JSP; Jackson 2002; 1975). JSP-menetelmässä, käyttämällä peräkkäis-,

valinta- ja toistorakennetta, voidaan proseduurissa eksplisiittisesti kuvata syötteen ja tulosteen tietorakenteita. Etuna on oikeellisuuden todentamisen helpottaminen: ohjelman rakenne vastaa syötettä/tulostetta ja jokaista alirakennetta käsitellään yhdessä paikassa. On kuitenkin mahdollista, että syötteen ja tulosteen rakenteet ovat ristiriidassa. Ristiriitoja voidaan ratkaista käsittelemällä poikkeuksia aliohjelmilla. Idea johti ajatukseen järjestelmien kuvaamisesta aliohjelmista muodostuvalla hierarkialla (Jackson 1983).

JSD-menetelmä (Jackson 1983) nostaa abstraktiotason yksittäisestä proseduurista samaan asiaan (esim. kirjaston asiakas) liittyvien proseduurien hierarkiaan. Proseduurit edustavat tapahtumia, jotka ovat ajallisesti järjestettyjä ja järjestystä ohjataan kontrollirakenteilla. Proseduurit ovat aliproseduureja ohjelmassa, jonka suoritus kestää asian käsittelyn elinkaaren. JSD-menetelmän lähtökohtana on todellisuuden mallista luotu simulaatio, jota hyödynnetään järjestelmän tulosten laskemisessa. Mallintaminen nähdään matemaattista todistamista tai ohjelman testaamista korvaavana keinona, ohjelman oikeellisuuden varmistamiseksi (ibid.).

Sovellusaluekielet (engl. Domain-Specific Languages) ovat rajatun sovellusalueen käsitteistä ja niiden käytön säännöistä koostuvia ohjelmointikieliä (Fowler 2010). Ne voidaan toteuttaa mm. esiprosessorilla, kääntämällä, tulkkaamalla tai käyttäen isäntäkielen käsitteitä (Mernik ym. 2005). Toteutus on mahdollista kaikilla yleiskäyttöisillä ohjelmointikielillä, mutta funktionaalisten ohjelmointikielten piirteet näyttäisivät sopivan tähän erityisen hyvin (Hudak 1996; Abelson & Sussman 1996). Haskellin monipuolinen tyypitys, ensisijaiset funktiot ja sivuvaikutuksettomuus tukee sovellusaluekielen muodostamista kielen omilla käsitteillä (Thompson 2011). LISP-sukuisten kielten syntaksin puurakenne ja makro-esiprosessori tekee sovellusaluekielten kirjoittamisen helpoksi (Steele & Gabriel 1993).

Scheme-kieli (LISP-kielen murre) koostuu pienestä määrästä ydinkäsitteitä, joista kaikki muut käsitteet on luotu (Dybvig 2003). Syntaksin laajentamisen mahdollistaa Schemen makroksi kutsuttu piirre. Esiprosessori muuttaa kaikki makrot ydinkäsitteiksi. Myös ohjelmoija voi luoda uusia käsitteitä makroilla. Makro on sääntö, joka määrittelee

mallin syntaksista, jolla käsite korvataan. Malli sisältää aukkoja, joihin sijoitetaan käsitteen parametrit. Scheme antaa hyvän lähtökohdan kielikeskeiselle-ohjelmoinnille (Felleisen ym. 2015), jossa luotu kieli koostuu ongelma-alueen käsitteistä ja toiminnoista. Ohjelmointi on kielen toteuttamista ja käyttämistä ongelman ratkaisuun (Ward 1994). Valmis ohjelma muodostaa mallin todellisuuden ongelman ratkaisusta.

Kehykset (frames) on menetelmä, jolla tiedon kokonaisuuksia ja niiden välisiä yhteyksiä voidaan esittää logiikkaohjelmointikielissä (Bratko 2011, Chen ym. 1986). Menetelmä kehitettiin alun perin tekoälyn tutkimuksessa tietämyksen mallintamiseen. Marvin Minsky (1974) kritisoi vallassa olevaa ihmisen tiedonkäsittelyn käsitystä liian yksityiskohtaiseksi, erityiseksi ja järjestymättömäksi. Arkipäiväisen ajattelun tehokkuuden selittämiseksi, on kerralla käsiteltävien tietoyksiköiden oltava isompia ja järjestetympiä. Hän esitti vaihtoehtoiseksi tietoyksikön malliksi kehystä, joka edustaa ihmisen muistaman tilanteen stereotyyppistä mallia. Logiikkaohjelmoinnissa kehys on tietorakenne, joka kuvaa tiettyyn asiaan liittyviä tietoja, toimintoja ja sen suhteita muihin asioihin.

Kehys on tietorakenne, joka koostuu lokeroista (Bratko 2011). Lokeroilla on nimi ja ne voivat sisältää yksinkertaisia arvoja, viittauksia muihin kehyksiin tai proseduureja. Proseduurit voivat laskea lokeron arvon hyödyntäen muuta informaatiota esim. perustuen kehyksen muiden lokeroiden arvoihin. Kehykset voivat myös periä toisen kehyksen ominaisuuksia, jolla voidaan esittää tiedon yleisempiä ja erityisempiä muotoja. Kehysten alkuperäinen idea oli selittää ihmisen tietojenkäsittelyä, mutta myöhäisempi tutkimus on keskittynyt tiedon rakenteelliseen esittämiseen (Fikes & Kehler 1985). Logiikkaohjelmoinnin faktat ja säännöt sopivat luonnollisella tavalla todellisuuden mallintamiseen (Evans 2004). Kehykset laajentavat tätä ominaisuutta mahdollistaen kokonaisuuksien ja näiden suhteiden esittämisen.

Domain-Driven Design (Evans 2004) on menetelmä (jatkossa DDD-menetelmä) monimutkaisuuden hallintaan ohjelmankehityksessä. Sekä sovellusalueen monipuolisuus, että ohjelman koko, voivat lisätä monimutkaisuutta. DDD-menetelmä hyödyntää olio-ohjelmoinnin mahdollisuuksia ja ottaa kantaa sekä ohjelman tekniseen toteutukseen, että ohjelmankehitysprosessiin (tässä keskitymme ensiksi mainittuun). Teknisenä lähtökoh-

tana on lähdekoodiin kuvaama todellisuuden malli, jonka ympärille muu ohjelma rakentuu nk. monitasoarkkitehtuurin (esim. Fowler 2002) mukaan. Arkkitehtuurin tasoja voivat olla esim. käyttöliittymä, sovellus, todellisuuden malli ja infrastruktuuri. DDD-menetelmässä todellisuuden mallin perusteella toteutetaan muut tasot.

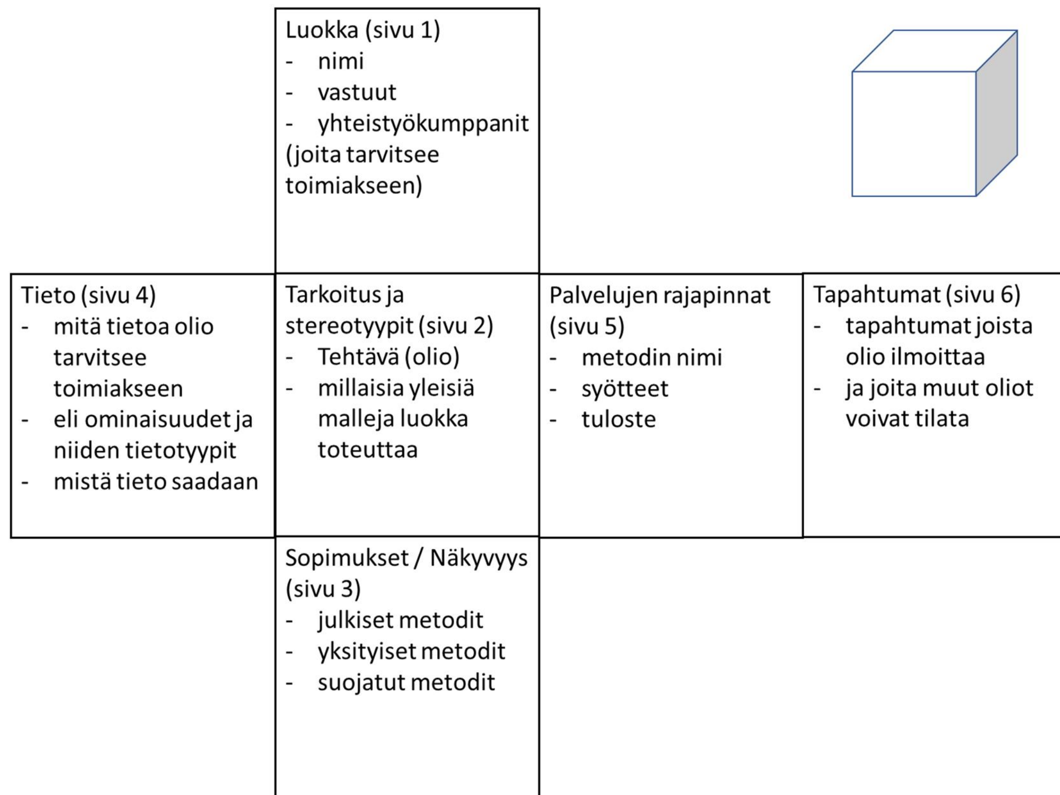
DDD-menetelmän keskeiset mallintamisen luokat ovat entiteetti, arvo-olio ja palvelu (Evans 2004). Entiteetit edustavat kohdealueen asioita, joilla on oma identiteetti ja joiden tila voi muuttua. Arvo-olioilla merkityksen antaa niiden edustama arvo tai arvojen yhdistelmä ja niiden tila on muuttumaton. Niitä käytetään Entiteettien arvojen määrittelemiseen. Palvelut ovat olioita, jotka suorittavat jonkin operaation, mutta niiden tila ei muutu. Näitä käytetään silloin kun toiminnallisuus ei luontevasti liity johonkin Entiteettiin tai Arvo-olioon. DDD-menetelmää voidaan soveltaa myös muissa ohjelmointiparadigmoissa (Wlaschin 2017).

Luvun lähtökohtana oli Skandinaavinen olio-ohjelmointi, jossa ohjelmointi tulkittiin mallintamiseksi. Ohjelman ajonaikaiset oliot toimivat todellisuuden mallina. Myös esitetyissä ohjelmointiparadigmoissa löytyi menetelmiä, jossa todellisuutta mallinnetaan eksplisiittisesti. Imperatiivisessa paradigmassa ulkoista toimintaa kuvataan proseduurien hierarkialla. Funktionaalisessa paradigmassa luotiin uudet käsitteet, jotka muodostivat kohdealuetta kuvaavan kielen. Logiikan kehykset järjestivät tiedot, toiminnot ja asioiden väliset suhteet kuvaamaan todellisuutta. Olio-ohjelmoinnissa mallintaminen perustui kolmen tyyppisiin olioihin: entiteetteihin, arvoihin ja palveluihin. Kaikissa paradigmoissa voidaan soveltaa niiden omia käsitteitä todellisuuden mallintamiseen.

4.3 Ohjelmointi tietokoneen käytön mahdollistamisena -malli

Ohjelmointi käytön mahdollistamisena -mallissa kuvataan lähdekoodissa eksplisiittisesti vuorovaikutus käyttäjän kanssa. Kuvaus määrittelee, miten vastataan käyttäjän toimiin ja miten ohjataan ohjelmaa tämän perusteella. Olio-ajattelussa toiminnan simulointi ohjaa olioiden löytämistä ja määrittelyä. Oliot tunnistetaan niiden tarjoamien ja käyttämien palveluiden kautta. Keskitetty koordinaatio korvataan hajautetulla vastuulla ja yhteistyöllä. Kun jokainen olio vastaa omasta toiminnastaan vältetään toimintojen toistamista ja joustamattomia sidonnaisuuksia. Olio-kuutio (West 2004) kiteyttää olio-

ajattelun oliokäsityksen (ks. Kuvio 21). Lähtökohtana ovat CRC-kortit (Beck & Cunningham 1989), jotka alun perin kehitettiin olio-ohjelmoinnin opetukseen.



Kuvio 21: Oliokuutiota käytetään apuna sovelluksen jakamiseksi toistensa kanssa vuorovaikutuksessa oleviin olioihin. Kuution kuusi pintaa kertoo olion vastuista/yhteistyöstä, toiminnan tarkoituksesta, metodien näkyvyydestä, tiedon tarpeista, palvelujen rajapinnoista ja millaisista tapahtumista olio tiedottaa.

CRC-kortin (Beck & Cunningham 1989) sisältämä luokan nimi sekä sen tarjoamat ja käyttämät palvelut muodostavat kuution yhden sivun. Rebecca Wirfs-Brock ja Alan McKean (2003) lisäsivät CRC-kortteihin olioiden tarkoituksen kuvauksen ja stereotyypit, joka muodostavat kuution toisen sivun. Olio-ajattelun kehittämisessä on määritelty neljä muuta sivua: palvelujen käytön rajoitukset, mitä tietoja tarvitaan palvelujen tarjoamiseen, millaiset ovat palvelujen rajapinnat ja mitä tapahtumat kertovat muille olion tilan muutoksista (West 2004). Olio-kuutio toimii mallina olion kyvystä olla vuorovaikutuksessa toisten olioiden kanssa. Olio-ajattelussa ohjelman ulkopuoliset toimijat tulkitaan myös olioiksi, joten ajattelu sisältää myös vuorovaikutuksen esim. ihmisten kanssa.

Interaction programming (jatkossa IP) on formaali menetelmä ohjelman ja käyttäjän

vuorovaikutuksen ohjelmointiin (Thimleby 2007). Vuorovaikutuksen kuvaaminen on ohjelman tilojen ja käyttäjän toimenpiteiden aiheuttamien tilojen välisten siirtymien esittämistä. IP-menetelmässä käytetään tilakonetta vuorovaikutuksen ohjelmointiin. Tuloksena on vuorovaikutuksen malli lähdekoodissa, joka sijoittuu monitasoarkkitehtuurissa (esim. Fowler 2002) käyttöliittymä- ja sovellustason väliin. Tilakone voidaan vielä esittää eksplisiittisesti matriisina, jossa tilat ovat sarakkeita ja niissä mahdolliset toimenpiteet rivejä. Tilakoneen ongelmana on, että tilojen määrä kasvaa nopeasti valtavaksi, kun muuttujia lisätään.

Puskuritilakoneessa (engl. buffer automata) tilojen ja siirtymien mallintaminen on korvattu laskukaavalla (Thimleby ym. 2011). Perinteisessä tilakoneessa muutos missä tahansa muuttujassa johtaa aina uuteen tilaan. Puskuritilakone koostuu useammasta puskurista ja vain osassa puskureista arvojen muutos vaihtaa tilaa esim. erotetaan lomakkeen esittäminen sen kenttien arvoista. Koska tilat lasketaan puskurien arvojen perusteella, ei tilojen siirtymiä voi suoraan havaita. Puskuritilakoneen malli on formaali, joten lähdekoodista voidaan kuitenkin generoida kaavio tilaan vaikuttavien puskurien arvoyhdistelmistä (Gimblett & Thimleby 2010).

Funktionaalisten LISP ja ML kielistä mallia ottavien kielten piirissä on kehitetty kaksi toisistaan poikkeavaa tapaa käsitellä käyttäjän ja ohjelman vuorovaikutusta. Scheme-kielessä on mahdollista tallentaa ohjelman jatkumo (engl. continuation), joka sisältää tilan sekä jäljellä olevan ohjelman (Dybvig 2003). Ohjelman jatkumoa on hyödynnetty web-palvelimen ja käyttäjän välisen vuorovaikutuksen hallintaan (Krisnamurthi ym. 2007). Vuorovaikutus kuvataan proseduurina, joka sisältää pisteitä, joissa jatkumo tallennetaan ja kontrolli siirtyy web-palvelimelta selaimelle (käyttäjälle). Kun otetaan uudestaan yhteys web-palvelimeen, niin ohjelman suoritus jatkaa tallennetusta kohdasta. Etuna on, että ohjelman eteneminen on luontevaa ja tilan säilyminen automaattista.

Elm on vuorovaikutteisiin ohjelmiin erikoistunut kieli, joka on ottanut mallia ML:n rinnakkaislaskennan murteesta ja jonka kääntäjä on toteutettu Haskell-kielellä (Chaplicki & Chong 2013). Kielessä on ominaisuudet graafisen käyttöliittymän ja vuorovaikutuksen määrittelyyn funktionaalisesti. Käyttäjän tai ulkoisen tekijän aiheuttama muutos

tulkitaan signaaliksi, jota voidaan käsitellä funktiolla (nk. funktionaalinen reaktiivinen ohjelmointi). Signaalit ovat diskreettejä ja uusi signaali suorittaa funktion uudelleen, joka mahdollistaa esim. käyttöliittymän arvojen päivittämisen. Scheme ja Elm tarjoavat deklaratiivisen tavan käsitellä vuorovaikutusta. Tilan eksplisiittisen hallinnan sijaan voidaan keskittyä ulkoiseen vuorovaikutukseen vastaamiseen.

B-Prolog rajoitelogiikkaohjelmointikielelle luotu korkean tason grafiikkakirjasto CGLIB (Zhou 2001), mahdollistaa sekä ruudulle piirtämisen että luodun grafiikan vuorovaikutteisen käsittelyn. Grafiikkakirjaston toteutuksessa hyödynnetään B-prologin rajoitteiden ratkaisuominaisuuksia (constraint solving) ja toimintasääntö (action rule) rakennetta. Rajoitteiden käsittely laajentaa logiikkaohjelmointia koskemaan muita arvoja kuin totuusarvoja (Gavanelli & Rossi 2010). Ongelma ratkaistaan hakemalla muuttujille arvot, jotka noudattavat annettuja rajoitteita. Rajoiteohjelmointi sopii hyvin logiikkaohjelmoinnin laajennukseksi (ibid.), koska muuttujille haetaan arvot niiden tietotyyppien arvoalueelta (vrt. Prologissa ohjelman lauseista) ja virheellisen yhdistelmän syntyessä palataan taaksepäin kokeilemaan uutta yhdistelmää.

CGLIB-kirjasto (Zhou 2001) tarjoaa käsitteet graafisten elementtien luontiin ja käsittelyyn sekä rajoitteet, joiden avulla voidaan määritellä elementtien sijoittelu ruudulla. Rajoitteiden käyttö vähentää graafisten elementtien asemoinnin vaivaa, koska ohjelmoijan on vain määriteltävä välttämättömät ehdot, loput laskee järjestelmä, huomioiden elementtien mitat ja keskinäiset sijainnit. Vuorovaikutus käyttäjän kanssa määritellään käyttäen toimintasääntöjä. Toimintasääntö määrittelee agentin, joka seuraa tiettyjä tapahtumia ja sisältää toimenpiteet, jotka suoritetaan tapahtuman toteutuessa. Tapahtumasta voidaan myös saada tietoja, joita hyödynnetään toimenpiteissä. Agentteja voidaan suorittaa peräkkäin ja rinnakkain. Ne voivat myös viestiä keskenään, joten tapahtuma voi esim. johtaa muutoksiin useassa graafisessa elementissä.

Tools and Materials Approach (jatkossa TMA; Lilienthal & Züllighoven 1997) on menetelmä käytön laatuun keskittyvien sovellusten luomiseen. Sovelluksella viitataan tässä sovellusalueeseen, ei niinkään ohjelmatuotteeseen. Kehittäjän tulisi ymmärtää käyttäjän työtehtävät ja ohjelman tulisi sisältää sovellusalueen keskeisiä asioita edustavat käsit-

teet. Tehtävillä ja käsitteillä on vastaavuus ohjelman komponenteissa. Vastaavuus tukee käyttäjää työn tekemisessä ja mahdollistaa työn organisoinnin tilanteen vaatimalla tavalla (ibid.). TMA on olio-ohjelmointiin perustuva menetelmä, jossa vastaavuus sovellusalueen ja ohjelman olioiden välillä saavutetaan käyttämällä yhdistävän teeman mukaisia vertauskuvia esim. automaatio, materiaali, työkalu ja työympäristö.

Työkalun, materiaalin, automaatin ja työympäristön vertauskuvat kuuluvat asiantuntijatyöhön (Lilienthal & Züllighoven 1997). Työkalu edustaa välineitä tai keinoja, joilla työtä tehdään. Materiaalit ovat työnkohteita ja ne voivat olla sekä fyysisiä että käsitteellisiä. Jos työvaiheet ovat mekaanisia, ja niille voidaan antaa täsmälliset säännöt, voidaan prosessi automatisoida. Työympäristö sisältää työkalut sekä materiaalit ja mahdollistaa työn järjestämisen mielekkäällä tavalla. Asiantuntijatyön vertauskuvien ja olio-ohjelmoinnin yhteys on TMA-menetelmässä toteutettu käyttäen ohjelmoinnin suunnittelumalleja (Züllighoven 2005). Suunnittelumallit ovat hyväksi havaittuja ratkaisuja usein toistuviin ongelmiin (Gamma ym. 1995). TMA-menetelmän mallit tarjoavat yleisen rungon sovellusalueen työn kohteen ja välineiden kuvaamiseen ohjelmassa.

Olio-ajattelu kuvasi olioiden välistä vuorovaikutusta tarjottujen ja käytettyjen palvelujen sekä tapahtumailmoitusten kautta. Imperatiivisessa mallissa tilakoneen siirtymät ja niitä käynnistävät tapahtumat edustivat vuorovaikutusta. Funktionaalisissa kielissä oli kaksi erilaista tapaa reagoida vuorovaikutukseen. Scheme kielessä ohjelman suoritus voitiin pysäyttää, kontrolli palauttaa käyttäjälle ja jatkaa suoritusta kun syöte oli saatu. Elm kielessä signaalien muutokset johtavat funktioiden suorittamiseen uudelleen, jolloin voidaan reagoida ulkoisiin tapahtumiin. Logiikkaohjelmoinnissa toimintasäännöt mahdollistavat käyttäjän palautteeseen reagoinnin. Olio-paradigmassa käytettiin työnteon vertauskuvia toteuttavia suunnittelumalleja. Vuorovaikutuksen mekanismit ovat paradigmoissa erilaiset, mutta yhteistä niille on eksplisiittiset rakenteet ulkoisiin tapahtumiin reagoinnin määrittelyyn.

4.4 Ohjelmoinnin viitekehyksen toteutuminen

Tutkimusoletuksena oli, että ohjelmoinnin viitekehyksen kolmelle mallille löytyy myös niitä toteuttavat menetelmät. Menetelmien tunnistamisen apuvälineenä käytettiin malle-

ja vastaavia ohjelmoinnin teorioita. Teorioiden toteutuksia haettiin neljästä ohjelmointiparadigmasta: imperatiivinen, funktionaalinen, logiikka ja olio. Viitekehyksen jokaiselle mallille löytyi toteutus kaikista paradigmoista ja näin myös tutkimusoletus täyttyi. Löydetyt menetelmät ilmentävät oman paradigmansa luonnetta, vaikka ohjelmointikielten yleiskäyttöisyydestä johtuen olisi mahdollista simuloida toisen paradigman käsitteitä. Käytön mahdollistamisen -mallin mukainen Tools & Materials Approach otti huomioon myös viitekehyksen muut mallit. Näin voidaan sanoa, että mallien yksittäisten esimerkkien lisäksi, on löydetty menetelmä, joka toteuttaa viitekehyksen kokonaisuudessaan.

5 ENEMMÄN KUIN OHJELMOINTIA

Luvussa haetaan kirjallisuudesta tukea luodulle ohjelmoinnin viitekehykselle ja sen ohjelmankehityksen tulkinnalle. Viitekehys määrittelee, että asiakkaan osallistumisen ohjelman suunnitteluun mahdollistamiseksi tulisi ohjelman lähdekoodissa löytyä käsitteet, jotka mallintavat työn kohdetta ja sen käsittelyä. Christiane Floyd (2002) kuvaa sosiologisen teorian näkökulmasta tietokoneartefaktia (sisältää tietokoneen laitteiston ja ohjelmat) osana ihmisen toimintaa, jossa keskeinen kysymys on, miten toiminta voidaan digitalisoida. Hän käyttää operaation käsitettä ihmisen toiminnan tulkitsemiseksi muodossa, joka soveltuu tietokoneella automatisoitavaksi.

Operaation käsite yhdistää yhteisöllisen, teknisen ja formaalin maailman. Järjestäytyneessä ihmisten toiminnassa (mm. sotavoimissa, lääketieteessä tai yrittämisessä) se viittaa etenemiseen tilanteessa ennalta suunnitellulla tavalla. Teknologiassa se voi tarkoittaa koneen toimintaa tai sen ohjaamista. Matematiikassa se viittaa laskutoimituksen osaan, joka on formaali ja yksiselitteinen. Operaatioon liittyy sekä toteutus että kuvaus. Toteutus on osa yksittäisen ihmisen tai ryhmän toimintaa. Kuvaus mahdollistaa operaation etenemisen ennalta määritellyllä tavalla, jota voidaan suunnitella, opettaa ja valvoa.

Kuvaus mahdollistaa operaation erottamisen sen soveltamisesta. Operaatio saa kuvauksen kautta nimen ja määritelmän, joten sitä voidaan verrata muihin operaatioihin. Operatiivinen rakenne yhdistää yksittäiset operaatiot ajan, logiikan ja kausaalisuuden suhteen yhteiseen päämäärään. Rakennetta voidaan itsessään pitää operaationa, jolloin operaatioita voidaan käyttää toiminnallisten hierarkioiden määrittelyyn. Operaation toteuttaminen vaatii toiminnan sopeuttamista sen asettamiin vaatimuksiin ja rajoituksiin. Tietokoneartefaktissa operatiivista rakennetta edustaa algoritmi, joka ohjaa tietokonetta.

Tietokoneartefaktin erityispiirre on, että se sisältää sekä operaation kuvauksen että suorituksen. Ihmisen toiminnassa nämä ovat eriytettyjä, koska kuvaus vaatii ulkopuolisen tarkkailijan. Tietokoneartefaktin ohjelma mahdollistaa operatiivisen rakenteen määrittämisen, joka liittää sen tekniseen, yhteisölliseen tai näitä ympäröivään maailmaan. Teknisenä artefaktina se voi suorittaa ohjelmansa määrittelemät operaatiot. Tieto mää-

rittelee tietokoneen operaatiot ja on samalla niiden kohteena. Operatiivinen rakenne on esitettävä eksplisiittisesti tietokoneen (ohjelmointi)kielellä. Koska tietokoneartefaktin kuvauksen kohteena on tieto, voidaan sillä mallintaa tietokoneen ulkopuolisten operaatioiden kuvauksia ja automatisoida näiden käsittelyä. Tietokoneartefaktin laitteiston kautta tiedon käsittelyllä voi olla myös vaikutusta ulkoiseen maailmaan esim. 3D tulostus korvaa perinteisen fyysisen materiaalin työstämisen.

Tietokonemaailman ulkopuolisia operatiivisia rakenteita voidaan toteuttaa tietokoneella määrittelemällä ne yksityiskohtaisesti. Jos kohteen operatiivista rakennetta ei ole kuvattu, on luotava rakenne, joka mallintaa kohteen keskeisiä toiminnallisuuksia. Sekä olemassa olevista että uusista operaatioista on luotava malli, jotta ne olisi käsiteltävissä tietokoneella. Tietokoneella on erilaisia tyylejä, joilla mallintaminen voidaan toteuttaa (Floyd 2002): proseduraalisessa käytetään proseduureja (imperatiivinen paradigma) tai funktioita (funktionaalinen paradigma), olio-tyylissä oliot ovat lähtökohta tiedon ja niihin liittyvien operaatioiden mallintamiselle, ja rajoite-tyylissä (tutkielmassa logiikkaohjelmointi) määritellään haluttu tulos, jossa algoritmi jää tietokoneen selvitettäväksi.

Christiane Floyd (2002) yhdisti yhteisöllisen, teknisen ja formaalin maailman operaation käsitteellä. Operaatioon liittyy sekä kuvauksen että toteutuksen ulottuvuudet. Tietokone sisälsi nämä molemmat. Koska tietokoneen operaatiot kuvaavat tietoa, voidaan sen oman toiminnan määrittelyn lisäksi mallintaa ulkopuolisen maailman operaatioita. Tutkielmassa on esitelty menetelmiä, jossa tietokoneen ohjaamisen lisäksi on voitu mallintaa eksplisiittisesti ulkopuolista maailmaa. Edellä esitetty sosiologiaan perustuva tulkinta tietokoneesta antaa teoreettisen perustelun tämän tutkielman mallintamisen tavoitteelle. Christiane Floyd (ibid.) mainitsee myös tutkielmassa käytetyt ohjelmointiparadigmat mallintamiseen soveltuvina välineinä.

Tutkielman tavoite mallintaa kohdealuetta saa tukea Christiane Floydin (2002) tulkinnasta. Jos tietokoneohjelman tavoite on olla osa ihmisten maailmaa, tulisi sen mallintaa sitä, mutta voidaanko perustella tutkielman viitekehyksen mukainen mallien jaottelu ja tavoite esittää mallit eksplisiittisesti lähdekoodissa. Tutkielman viitekehyksen taustateorioihin kuuluvassa Terry Winogradin ja Fernando Floresin (1986) kirjassa tulkitaan oh-

jelmankehitys kielen kehittämisenä. Kirjan mukaan perinteiset ohjelmointikielet sopivat huonosti kuvaamaan ohjelmien tarkoitusta ja käsitteellistä rakennetta. Ratkaisuna esitetään ensimmäisen kirjoittajan erillisessä artikkelissa (Winograd 1979) visioimia järjestelmänmäärittelykieliä. Artikkeli ja sen idea, aloittaa julkaisujen ketjun, josta löytyy vastaus kahteen vielä avoimeen kysymykseen ja vahvistusta mallintamisen tavoitteelle.

5.1 Enemmän kuin ohjelmointikieliä

Terry Winograd (1979) esitti lähes 40 vuotta sitten, että tulevaisuudessa ohjelmointikielet eivät riitä laajojen tietoteknisten järjestelmien monimutkaisuuden hallintaan. Tietokoneilla ratkottavat ongelmat eivät enää olisi matemaattisen selkeitä, vaan tietokoneet olisivat osa monimutkaisia sulautettuja järjestelmiä. Ohjelmankehitys tapahtuisi ohjelmointikieliä korkeammalla tasolla. Keskeistä ei enää olisi ohjelmien kehittäminen vaan olemassa olevien ohjelmien yhdistäminen, muokkaaminen ja selittäminen. Winograd (ibid.) esitti ratkaisuksi ohjelmankehityksen monimutkaisuuden hallintaan vision ohjelmointijärjestelmistä.

Korkeantason ohjelmointikielet abstrahoivat ohjelmien kirjoittamisen tietokoneen rakenteen yksityiskohdista ja ohjelmointijärjestelmät vastaavasti ohjelmat tietokoneen toiminnasta. Ohjelmoinnin tulisi olla deklaratiivista (mitä) imperatiivisen (miten) sijaan. Komentoketjut korvattaisiin prosessien ja niiden kohteiden määrittelyillä. Painopisteenä olisi ohjelman käyttäytyminen: mitä tehdään, missä järjestyksessä ja miten tämä vaikuttaa vuorovaikutukseen. Ohjelma voi olla vuorovaikutuksessa käyttäjien, muiden tietokoneiden tai fyysisten järjestelmien kanssa. Vuorovaikutus ohjaisi tietokoneen toimintaa formaalin matemaattisen määritelmän sijaan.

Ohjelman lähdekoodissa muuttujien ja proseduurien nimeäminen sekä ohjelman tilan varmistukset, kuvaavat mitä ohjelman suoritus koskee. Mutta, jos lähdekoodin toimintaa selostettaisiin, niin kuvaukseen kuuluisi paljon mitä koodissa ei näy. Kommentoimalla voidaan tarjota kuvaus ohjelmanpätkän paikallisesta toiminnasta, erillinen dokumentaatio kertoo ohjelmasta kokonaisuutena ja laajoissa järjestelmissä on lisäksi dokumentoitu erilaiset protokollat ja ohjelmoinnin käytännöt. Kuvaukset ovat kuitenkin usein hajallaan eri paikoissa. Ohjelmointijärjestelmän tavoitteena on yhdistää irralliset kuvaukset

yhtenäiseksi kokonaisuudeksi. Ymmärtämistä korostettaisiin ohjelmoinnin sijaan.

Ohjelmointijärjestelmän komponenttien kuvaukset koskisivat kolmea eri osa-aluetta: aihe, vuorovaikutus ja toteutus. Aihe koskee järjestelmän tarkoitusta. Ilman kohdealueen kuvausta ei käsiteltävällä tiedolla ole merkitystä. Mitkä tiedot kohdealueesta valitaan, vaihtelee ohjelman käyttötarkoituksen mukaan. Jokaisen toiminnassa olevan järjestelmän voidaan katsoa olevan vuorovaikutuksessa ympäristönsä kanssa. Vuorovaikutuksen komponentit eivät ole sidottuja tiettyyn aiheeseen, koska saman aiheen kanssa voidaan olla eri tavalla vuorovaikutuksessa. Sisällön kuvaus vuorovaikutuskomponentissa koskee sen teknistä toteutusta ja varsinaista vuorovaikutusta edustaa komponentin rajapinta ulkoiseen ympäristöön. Toteutuksen osa-alueen ajatellaan perinteisesti sisältävän tietokoneen suoritusta ohjaavat komennot. Ohjelmointijärjestelmässä määriteltäisiin myös komentojen yhdistelmien merkitykset ja seuraukset. Tähän kuuluisi erityisesti ne kommentit, joita lisätään koodin myöhempää ymmärtämistä helpottamaan.

Winograd (1979) ajatteli ohjelmointijärjestelmän toimivan korkeammalla abstraktiotasolla kuin komennot. Mallina oli tekoälyjärjestelmien tavoite organisoida tietoa ja sen käsittelytapoja. Eri abstraktiotasolla olevia prototyyppejä käytettäisiin tiedon ja prosessien kuvaamiseen. Kuvaukset olisi tallennettu muodossa, joka mahdollistaa niiden hakemisen, käsittelyn ja esittämisen. Niistä voisi yhdistelemällä luoda uusia ohjelmia. Yleisten ominaisuuksien lisäksi löytyisi valmiiden komponenttien kirjasto aikaisemmin kuvatusta kolmesta osa-alueesta. Ongelman ratkaisussa ohjelmoija hyödyntäisi valmiiden komponenttien ominaisuuksia ja loisi uusia näitä yhdistelemällä. Ohjelmointijärjestelmän kehitysympäristö tukisi kuvauksien käsittelyä ja pyrkisi automatisoimaan uusien korkeamman tason komponenttien määritelmiin sopivien kuvauksien löytämistä.

Ohjelmointijärjestelmä oli visio, jonka nähtiin vaativan kehitystä kolmella osa-alueella: ohjelmien kuvauskieli, prosessien määrittely ja kehitysympäristö. Ohjelmien kuvauskielen tulisi keskittyä niihin piirteisiin, jotka antavat ohjelmoijalle hyvän käsityksen ongelmasta ja sen ratkaisusta. Alphard ohjelmointikieli mainitaan esimerkkinä vastaavasta tavoitteesta. Prosessin määrittelyyn liittyvät yleiset tietojenkäsittelyn komponentit kuten algoritmit, tietorakenteet ja protokollat tulisi tarjota valmiina kirjastona. Esimerkkinä

tällaisesta työstä mainitaan mm. modulaarisuudessa CLU-kieli, tietorakenteissa LISP ja APL, komponenttien välisessä viestinnässä Simula ja Smalltalk. Ohjelmointijärjestelmä digitaalisten järjestelmien luomiseen on itsessään monimutkainen järjestelmä. LISP-ohjelmointikielen laatuista kehitysympäristöä esitettiin tavoitteeksi.

Ohjelmointikielten virtuaalikoneiden, kuten Java JVM (Java Virtual Machine) ja C# CLR (Common Language Runtime) ympärille syntyneet ekosysteemit sekä niiden kehitysympäristöt, kuten esimerkiksi Eclipse ja Microsoft Visual studio, voidaan pitää ohjelmointijärjestelmän toteutuksina. Komponenttien kuvaukset ovat kuitenkin keskeisesti toteutukseen liittyviä, joten Terry Winogradin (1979) visio ei täysin toteudu. Tutkielmassa luodun ohjelmoinnin viitekehyksen jaottelun voidaan katsoa vastaavan ohjelmointijärjestelmän komponenttien osa-alueita. Visiossa haettiin esimerkkejä ohjelmointikielistä, kun taas tutkielmassa on löydetty aiheen ja vuorovaikutuksen komponentteja ohjelmointikieliä hyödyntävistä menetelmistä. Tutkielman viitekehyksen mukaiset menetelmät ovat askel kohti visioitua ohjelmointikieliä korkeammalla tasolla olevaa ohjelmankehitystä.

5.2 Enemmän kuin ohjelmointia

Edellisessä luvussa esitetty ohjelmointijärjestelmä oli visio kehittyneemmästä tavasta luoda ohjelmia. Ohjelmointijärjestelmät olisivat ratkaisu monimutkaisuuden hallintaan, joissa tietokoneen toiminnan määrittelyn sijaan käsiteltäisiin ongelma-aluetta. TEDIUM (Blum & Sigilito 1985) on Winogradin (1979) vision toteutus. Se on työkalu tietokantapohjaisten tietojärjestelmien määrittelyyn ja automaattiseen luontiin. Määrittely tapahtuu sovellusalueen kontekstissa, josta luodaan automaattisesti tarvittavat ohjelmat. Ratkaisu mahdollistaa adaptiivisen suunnittelun, jossa kokemuksen lisääntyessä voidaan järjestelmää kehittää edelleen, määrittelyjä muuttamalla ja lisäämällä. TEDIUM oli tuotantokäytössä, mutta samalla kehitettiin myös ohjelmointijärjestelmän teoriaa.

Lähtökohtana oli ohjelmistotuotantoprosessi, jonka katsottiin sisältävän kolme siirtymää: koetuista tarpeista vaatimusmääritelmäksi, vaatimusmääritelmästä ohjelman lähdekoodiksi ja ohjelman lähdekoodista suoritettavaksi ohjelmaksi. Vaatimusmääritelmän ja ohjelman lähdekoodin merkityksen vastaavuus voidaan verifioida, mutta hyödyllisyys

voidaan arvioida vain ohjelmaa käyttämällä. Julkaistu ohjelma muuttaa myös työntekoa, joka voi synnyttää uusia tarpeita. Suunnittelijalle tämä on haasteellista, koska seuraukset on vaikea nähdä etukäteen. Tarvittaisiin mahdollisuus iteratiiviseen suunnitteluun. TEDIUMin ratkaisu oli formaali vaatimusmääritelmä, josta ohjelma luotiin automaattisesti. Näin muutokset voidaan tehdä ohjelman käytön (vaatimusten) kontekstissa.

TEDIUMin lähtökohtana oli, että käyttäjä tulkitaan suunnittelijaksi, joten vaikka ohjelmointijärjestelmä abstrahoi teknisistä yksityiskohdista, niin ratkaisun määrittelystä ei tingitä. Se poikkeaa siksi naiiveista automaattisista sovelluskehittimistä, koska se vaatii käyttäjältä sitoutumista järjestelmän oppimiseen (Blum & Sigilito 1985). TEDIUM koostuu kolmesta moduulista: suunnittelu-moduulia käytetään vaatimusmäärittelyn muodostamiseen, muunnos-moduuli muuttaa määrittelyn ohjelmointikielelle ja täydentää puuttuvat yksityiskohdat, ja sovellusgeneraattori-moduuli luo lopullisen ohjelman. Jokainen moduuli vastaa yhtä ohjelmistotuotannon siirtymää, joista ensimmäinen vaatisi suunnittelijaa ja kaksi muuta on automatisoitu.

Jokainen TEDIUMin moduuli sisältää ohjelman kuvauksen käsittelyä. Suunnittelumoduulissa on kaksi kuvauksen tasoa. Ensimmäinen taso on epäformaali, sen tarkoituksena on tukea sovellukselle asetettavien vaatimusten kirjaamista ja niihin liittyvien tietorakenteiden sekä prosessien hahmottelua. Tuloksena on luonnos, josta toisella tasolla suunnittelija tekee sovelluksen formaalin määritelmän. Määritelmä sisältää tekstipohjaiset kuvaukset tietomallista ja ohjelman toiminnasta (Blum & Sigilito 1985). Näitä voidaan pitää sovellusaluekielinä (ks. Luku 4.2). Kuvausten perusteella generoidaan ohjelmakoodi, joka sisältää tietorakenteet, algoritmit sekä vuorovaikutukseen tarvittavat komennot. Lopuksi ohjelman lähdekoodi käännetään binääriseksi ohjelmaksi.

TEDIUMin lähtökohta suunnitteluun on adaptiivinen, jossa ohjelmistotekniikka nähdään ongelman ratkaisuna tuotteen valmistamisen sijaan (Blum 1993). Ohjelman julkaisu, muuttaa ympäristöä josta alun perin vaatimukset ovat syntyneet, joka voi puolestaan vaatia muutoksia ohjelmaan. Lehman (1980) kutsuu tällaisia ohjelmia E-tyyppisiksi (Blum 1993). Ohjelman lähdekoodia on helppo muuttaa, mutta tästä ei seuraa että oikeiden (ongelma-alue) muutosten tekeminen olisi helppoa. Sattumanvarainen ohjelman

rakenne tai toiminnallisuuden ja optimoinnin sekoittaminen vaikeuttaa muutosten tekemistä. TEDIUMissa muokkaukset tehdään vaatimusmääritelmään, jota voidaan pitää nykyisen ongelman ratkaisun kuvauksena. Edellytykset tällaiselle järjestelmälle on kypsä teknologia ja riittävä ymmärrys sen soveltamisesta ongelma-alueeseen.

Kun vuonna 1980 toimittajan tuki John Hopkins syöpäkeskuksen tietojärjestelmän ohjelmointikielelle päättyi, tarvittiin uusi järjestelmä. Luotu järjestelmä (Blum 1993) toteutti TEDIUM-mallia. Järjestelmään kuului 200 päätettä, joita käytti 100 työntekijää. Kehitystyön jälkeen 4 suunnittelija/käyttäjää vastasivat järjestelmän jatkokehityksestä. Järjestelmään oltiin yleisesti tyytyväisiä ja siihen luotettiin potilaiden terveyteen liittyvissä päätöksissä (Blum 1993). Sovellusten määrä järjestelmässä kasvoi 10% vuosivauhtia n. 3700 vuonna 1983 aina n. 9300 vuonna 1992. 70 % sovellusten määritelmistä tehtiin muutoksia alle vuosi ensimmäisestä julkaisusta ja tekijä oli 44 % tapauksista muu kuin alkuperäinen suunnittelija. Tietomallin tauluille vastaava prosentit olivat 26 % ja 38 %. Luvut osoittavat että järjestelmää on pystytty kehittämään ja myös olemassa olevia määritelmiä muokkaamaan. Lisäksi muokkaaminen on onnistunut myös henkilöiltä, jotka eivät alun perin luoneet kyseistä järjestelmän osaa.

Blum (1996) nimittää ongelmalähtöistä ohjelmistokehitystä ”Enemmän kuin ohjelmoinniksi”, koska kehittäminen tapahtuu ongelman ja sen ratkaisun suunnittelun tasolla. Suunnittelu ei koske teknologiaa (tuotetta) vaan ihmisten toimintaympäristön muuttamista tuomalla siihen uusia mahdollisuuksia. Kehittäjät luovat teknisen ympäristön, jolla käyttäjät suunnittelevat ongelman ratkaisun, joten yhteistyötä voidaan luonnehtia osallistavaksi suunnitteluksi. Blum (1996) käyttää Markku I. Nurmisen (1988) kolmea tietojärjestelmänäkökulmaa suunnittelun osatekijöiden luokitteluun: tekniseen (järjestelmä), käyttöliittymä (sosiotekninen) ja käytön (humanistinen) kategoriaan. Hän yhdistää kolme suunnittelukategoriaa, jossa tekninen on osa käyttöliittymää ja käyttöliittymä puolestaan edellytys käytön huomiointiin. TEDIUMin suunnitteluominaisuudet edustavat em. kategorioita.

TEDIUM vastaa tutkielman viitekehystä. Se sisältää eksplisiittisesti kolme ohjelmoinnin mallia, jotka jäsentävät sovelluksen rakennetta. Valmista ohjelmaa luonnehditaan

Lehmanin (1980) E-tyypin mukaiseksi, joka oli tutkielmassa se taso jolla käyttäjä pystyi arvioimaan ohjelman vaikutusta omaan työntekoonsa. Blum (1996) käyttää myös Markku I. Nurmisen (1998) tietojärjestelmän näkökulmia ohjelman rakenteiden luokitteluun. Hän kuitenkin yhdistää näkökulmat ohjelmiin eri tavalla kun tutkielmassa. Ero on kuitenkin lähinnä teknisessä jaottelussa ja samat asiat löytyvät ohjelmasta. Yksi syy eroon voi olla, että ohjelmointijärjestelmä on tarkoitettu suunnittelija/käyttäjälle, kun taas tutkielmassa on ajateltu käyttäjän ja kehittäjän yhteistyötä. TEDIUM koski tietokantapohjaisia ohjelmistoja, mutta tutkielmassa sovellustyyppiä ei ole rajattu.

5.3 Ohjelmistoarkkitehtuurit

Jason Baragryn ja Karl Reedin (1998) mukaan ohjelmistoarkkitehtuurien tutkimusta vaivaa ristiriitaiset tulkinnat. Syynä on oletus, että ohjelmistotekniikka vastaa konkreettisia (fyysisiä) asioita valmistavia aloja, joissa tuotannon tulos ilmentää suunnitteluvaiheessa määriteltyä arkkitehtuuria. Ohjelmistonkehityksen teoreettinen tarkastelu tuo kuitenkin esille erot fyysisestä valmistuksesta. Kirjoittajien toisessa artikkelissa (Baragry & Reed 2001) esitetään, että ohjelmistoarkkitehtuurin käsite tulisi perustua ohjelmistotekniikan uuteen tulkintaan. Blumin (1996; ks. edellinen luku) teoreettista työtä käytetään esimerkkinä vastaavasta näkemyksestä.

Paul Clements (1996, viitattu Baragry & Reed 1998) esittää viisi tekijää, jotka aiheuttavat ohjelmistoarkkitehtuurin määritelmän epämääräisyyden:

- puolestapuhujat tulkitsevat käsitettä omien alojensa näkökulmasta
- tutkimus seuraa käytäntöä, sen ohjaamisen sijaan
- tutkimusala on uusi
- alaa määrittelevien käsitteiden epätäsmällisyys ja tulkinnanvaraisuus
- arkkitehtuurin käsitettä käytetään huolimattomasti ja sen suhde ohjelmistotekniikkaan hämärtyy

Ohjelmistoarkkitehtuuri perustuu oletukseen, että ohjelmistotekniikka vastaa luonteeltaan muita tekniikan aloja. Jason Baragryn ja Karl Reedin (1998) mukaan eroavaisuudet perinteisiin ”suunnittele ja valmista” aloihin ovat keskeisempiä kuin yhtäläisyydet. Ohjelmistoilla ei ole fyysistä ilmentymää, jonka perusteella voitaisiin verrata suunnittelus-

sa käytettyjä abstraktioita valmiin tuotteen rakenteeseen. Vertaamisen vaikeus haittaa arkkitehtuurin hyödyn arviointia ja sen esille tuontia. Ohjelmistokehityksen tuloksena on sekä staattinen lähdekoodi että dynaaminen suoritettava ohjelma. Useimmilla tekniikan aloilla, ei ole tällaista eroa tuotteen muodon ja suorituksen välillä.

Ohjelmistotekniikassa arkkitehtuurin rooli on poikkeava perinteisiin tekniikan aloihin verrattuna (Baragry & Reed 1998). Yhteistä on erilaisten arkkitehtuurin kuvausten käyttäminen. Perinteisillä aloilla kuvaukset ovat näkymiä samaan kokonaisuuteen. Esimerkkinä voidaan pitää talonrakennusta, jossa voi olla erilliset piirrokset rakenteelle, putkistolle ja sähkölle. Ohjelmistotekniikassa arkkitehtuuri voi kuitenkin kuvata joko asiakkaan ongelman ratkaisuperiaatetta, lähdekoodin organisointia tai tietokoneen fyysistä toimintaa. Vasta lähdekoodin kääntäminen ohjelmaksi ja ohjelman suorittaminen tietokoneella, tuottaa ratkaisun. Ohjelman suoritukseen liittyy lisäksi tietokone, käyttöjärjestelmä ja muut ohjelmistot, joiden toiminta on lähdekoodin välittämää kuvaa laajempi.

Tutkimuksessa, jossa vertailtiin vakionopeudensäätimien ohjelmistojen ja laitteistojen arkkitehtuureja, havaittiin mallintamismenetelmien olevan molemmissa keskeisiä (Baragry 1996). Erona oli mitä mallintaminen koski: laitteistoissa suunniteltiin millaisilla fyysisillä komponenteilla ratkaisu voitaisiin toteuttaa, kun taas ohjelmistoissa ratkaisu toteutetaan mallia kehittämällä. Fyysisillä tekniikan aloilla komponenttien tuntemus ja yhdistämisen tavat ovat osa alan ammattiosaamista, joka em. tutkimuksessa tuli esille saman arkkitehtuurityylin hyödyntämisenä. Ohjelmistotekniikassa ohjelmoija voi käyttää monenlaisia tyylejä ja em. tutkimuksessa voitiin tunnistaa viisi erilaista arkkitehtuuria. Ohjelmistotekniikan haasteena on mihin perustaa arkkitehtuurin valinta.

Ohjelmistotekniikan arkkitehtuurin yhteydessä abstraktio merkitsee eri asiaa kuin perinteisillä tekniikan aloilla (Baragry & Reed 2001). Perinteisesti abstraktio tarkoittaa yksityiskohtien vähentämistä, tietyn näkökulman esille tuomiseksi. Ohjelmistotekniikassa abstraktio muistuttaa enemmän filosofian ja psykologian tulkintoja, jossa abstraktio on konsepti, joka edustaa joukkoa muita konsepteja. Esimerkiksi hedelmä edustaa omenaa, appelsiinia, banaania, jne., mutta on näistä erillinen. Ohjelmistotekniikan eri arkkitehtuurien yhteys syntyy tulkitsemalla toisen arkkitehtuurin konsepteja käyttäen käsillä

olevan arkkitehtuurin käsitteitä. Vaatimusmääritelmä toteutetaan lähdekoodin käsitteillä ja lähdekoodin käsitteet puolestaan tulkitaan tietokoneen toimintana.

Ohjelmistoarkkitehtuurin ongelmien korjaamiseksi tarvitaan uusi teoreettinen perusta ohjelmistotekniikalle (Baragry & Reed 2001). Ohjelmistotekniikassa hyödynnetään konseptin, abstraktion, teorian ja mallin käsitteitä. Näitä tutkitaan filosofiassa ja psykologiassa, joista Jason Baragry (2000) löysi kaksi erilaista näkökulmaa ohjelmistotekniikkaan. Perinteinen artefaktin valmistamisen näkökulma perustuu käsitykseen jaetusta todellisuudesta, jota ihminen tulkitsee tunnistamalla ja luokittelemalla todellisuuteen liittyviä käsitteitä. Uudessa todellisuuden mallintamisen näkökulmassa ihmisen todellisuuden tulkinta perustuu aikaisemmin koettuun, jolloin kokemus on subjektiivinen. Ristiriitojen ilmetessä muokataan todellisuuden mallia sisältämään uudet kokemukset.

Todellisuuden mallintamisen näkökulman mukaan ei ole objektiivista todellisuutta, vaan se on aina tulkittua, subjektiivista. Näkökulma näyttäisi voivan selittää ohjelmistotekniikassa ohjelmakoodin uudelleen käytön ja arkkitehtuurin liittyviä ilmiöitä. Ohjelmistokirjastoja ja suunnittelumalleja tulee hyödynnettyä enemmän, jos niihin tutustutaan ennen ohjelman suunnittelua. Tietotekniikan ulkoisiin käsitteisiin viittaavia ratkaisuja on vaikeampi uudelleen käyttää, kuin tekniikkaan liittyviä. Ongelman ratkaisua voidaan kuvata monella eri tavalla. Ratkaisuja voidaan myös toteuttaa monella eri tavalla ja samaa toteutusta voidaan hyödyntää erilaisiin ratkaisuihin. Jason Baragryn (2000) mukaan, ohjelmistotekniikkaa tulisi tulkita ongelman ratkaisevan teorian kehittämisenä, tuotteen valmistamisen sijaan.

Jason Baragry (2000) esittää kirjallisuudesta useita Todellisuuden mallintamista tukevia näkökantoja mm. Amblerin ym. (1992), Lawsonin (1990), Lehmanin (1980), Hirscheimin ja Kleinin (1989), Winogradin ja Floresin (1986), Dahlbom ja Mathiassenin (1993), Naurin (1992) ja Blumin (1996) työt mainitaan. Monet em. kirjoittajista käsittelevät jotain osa-aluetta todellisuuden mallintamisen näkökulmaa vastaavasti, mutta Naur (1992) ja Blum (1996) käsittelevät ohjelmistotekniikkaa kokonaisuutena (Baragry 2000). Jason Baragry (2000) nostaa esille myös Blumin (1996) kehittämän TEDIUMin näkökulman käytännön toteutuksena, mutta huomauttaa, että ei ole selvää miten ratkai-

sua voitaisiin laajentaa ohjelmistotekniikkaan yleensä.

Todellisuuden mallintamisen näkökulma näyttäisi tukevan tutkielman viitekehyksen ideaa. Jason Baragryn (2000) mainitsi näkökulmaa tukeviksi Lehmanin (1980), Winogradin ja Floresin (1986), ja Naurin (1992) teoksia, joita on myös käytetty viitekehyksen luomisessa. Jason Baragryn ja Karl Reedin (1998; 2001) tutkimuksen perusteella voidaan katsoa, että tutkielman viitekehys määrittelee menetelmiä joilla luodaan ohjelmistoarkkitehtuureja. Viitekehys voisi toimia välittäjänä ohjelmistotekniikan kolmen arkkitehtuurin välillä, koska vaatimusten ja toteutuksen mallit on esitetty lähdekoodissa. Jason Baragryn (2000) mukaan on epäselvää, miten todellisuuden mallintamisen näkökulmaa edustavaa TEDIUMia (Blum 1996), voitaisiin soveltaa rajattua kohdealuetta yleisemmin. Tutkielman viitekehyksen mukaisissa menetelmissä ei em. rajoitetta ole.

5.4 Kirjallisuuden tuki ohjelmoinnin viitekehykselle

Tutkielman viitekehyksen ideana on, että asiakas ja kehittäjä luovat yhteisen käsityksen kehitettävän ohjelman käytöstä, jota edustaa lähdekoodissa eksplisiittiset mallit. Malleihin kuuluu toiminnan kohde, sen käsittely vuorovaikutuksessa tietokoneen kanssa ja em. toteuttavat ohjelmointikielen käsitteet. Viitekehyksen mukaisia ohjelmointimenetelmiä on löytynyt neljästä ohjelmointiparadigmasta, jotka eksplisiittisesti tai implisiittisesti edustavat em. kolmea mallia. Teoriaa on tutkielmassa hyödynnetty viitekehyksen luontiin, mutta tässä luvussa on katsottu löytyykö viitekehykselle tukea laajemmin ja voisiko se olla hyödyllinen näissä konteksteissa.

Tietokoneen suorittama ohjelma on operatiivisen rakenteen kuvaus ja toteutus. Ohjelman operaatioiden kohteena on tieto, jolloin niillä voidaan mallintaa ulkopuolisia operaatioita. Christiane Floyd (2002) käytti operaation teoreettisena lähtökohtana sosiologiaan kuuluvaa kulttuurihistoriallista (esim. Engeström 2004) toiminnan teoriaa. Teoria tuo mukanaan ihmisen toiminnan välineellisyyden, jossa työkalut välittävät tekoja ja symbolit tietoa. Toimintaan liittyy myös yhteisö, jonka jäseniä ohjaavat säännöt ja työnjako. Kehitys näkyy toiminnassa välineiden käytön kulttuurisena muutoksena, kun taas yksilössä se ilmenee oppimisena. Toiminnan teoria asettaa tietokoneen ja ohjelmien kehittämisen osaksi yhteisöllistä toimintaa.

Toiminnan teoriaa on hyödynnetty ohjelmistotekniikassa viitekehyksenä toiminnan ja tekniikan rinnakkaiseksi kehittämiseksi (Luukkonen ym. 2013). Tarkastelu on kuitenkin abstraktia, joten ohjelmiston ominaisuuksien määrittelyyn on käytettävä muita menetelmiä. Osallistavan suunnittelun menetelmät parantavat asiakkaan mahdollisuutta ilmaista mitä hän haluaa, mutta samalla niissä näyttäisi jäävän teknologia huomioimatta (Korsgaard ym. 2016). Christiane Floydin (2002) operaatioiden mallintamisessa tietokoneen ulkopuolisten operaatioiden kuvaukset muutetaan tietokoneen operaatioiksi. Mallintaminen tukee asiakasta ohjelman ominaisuuksien määrittelyssä ja kehittäjää toiminnan huomioimisessa. Toiminnan teorian mukaan uuden työkalun käyttöönotto voi johtaa tarpeisiin muuttaa toimintaa (Engeström 2004). Tutkielman viitekehyksen mukaisissa menetelmissä operaatioiden mallit ovat eksplisiittisiä, jolloin voidaan ennakoida uuden työkalun vaikutuksia.

Baragryn ja Karl Reedin (1998; 2001) tutkimuksen perusteella voidaan katsoa, että tutkielman viitekehys määrittelee menetelmiä joilla luodaan ohjelmistoarkkitehtuureja. Erityistä viitekehyksen arkkitehtuureissa on asiakkaan vaatimusten mallintaminen ja arkkitehtuurin eksplisiittinen esittäminen lähdekoodissa. Vaatimusten ja arkkitehtuurin yhdistämistä on kuvattu Kahden huipun mallilla (engl. Twin Peaks model), jossa kehittäminen etenee huipulta toiselle laskeutuvana spiraalina sekä tarkentuvien vaatimusten että luotujen arkkitehtuuristen ratkaisujen kautta (Cleland-Huang ym. 2013; Nuseibeh 2001). Viitekehyksen menetelmien käsitteet mahdollistavat uusien ominaisuuksien arvioinnin sekä toiminnallisesta että teknisestä näkökulmasta ennen toteutusta.

Ohjelmistolla on arkkitehtuuri, riippumatta siitä onko sitä suunniteltu tietoisesti vai onko se syntynyt ongelmanratkaisun sivuvaikutuksena. Käytettyä arkkitehtuuria on vaikea havaita lähdekoodista, jolloin on tutkittu tapoja tämän yhteyden kuvaamiseen (Javed & Zdun 2014). Kuvaukset perustuvat yhteyksien dokumentaatioon ja työkaluihin, jotka helpottavat muutosten hallintaa. Koska vaatimusmäärittelyn toteutuminen on riippuvainen ohjelmistoarkkitehtuurista, on myös kehitetty keinoja dokumentoida näitä yhteyksiä (Alebrahim 2017; Avgeriou ym. 2011). On myös ehdotettu, että arkkitehtuurin voisi kirjata suoraan lähdekoodiin (Christensen & Hansen 2011; Woods & Rozanski 2010).

Tutkielmassa mennään askel pidemmälle, viitekehyksen menetelmissä toiminnallisia vaatimuksia kuvaava arkkitehtuuri on eksplisiittinen ohjelman lähdekoodissa.

Bruce I. Blumin ja Vincent G. Sigilliton (1985) luoma TEDIUM-ohjelmointijärjestelmä, joka mahdollisti loppukäyttäjien kehittää tietojärjestelmää itse, vastaa tutkielman viitekehystä. Anna-Liisa Syrjänen ja Kari Kuutti (2011) pitivät Bruce I. Blumin (1996) työtä loppukäyttäjän kehittämisen tukemiseksi uraauurtavana. He katsovat kuitenkin loppukäyttäjän kehitysmenetelmien puutteeksi, että ohjelmiston muutosten vaikutuksia organisaation toimintaan ei huomioida. Ratkaisuna he esittävät kulttuurihistorialliseen toiminnan teoriaan perustuvan muutoslaboratorion (Virkkunen & Newnham 2013) ja loppukäyttäjän kehittämisen yhdistämistä. Kehitys koskisi tällöin sekä teknologiaa, että toimintaa (Syrjänen & Kuutti 2011). Blumin ja Sigilliton (1985) ohjelmointijärjestelmä sopii tutkielman viitekehykseen, joten tämä viittaa mahdollisuuteen käyttää yhdistelmässä myös viitekehyksen muita menetelmiä.

Muutoslaboratorio on työn kehittämiseen käytetty iteratiivinen interventiomenetelmä (Daniels 2008). Siinä työntekijät tutkijoiden tuella pohtivat nykyisen toiminnan ristiriitoja, miten työ on muuttunut aikaisemmasta ja millaisessa työn järjestelyssä ristiriitoja ei olisi. Apuna toiminnan jäsentelyyn käytetään toiminnan teorian kaaviota, joka sisältää toimijan, säännöt, työkalut, yhteisön, työnjaon ja toiminnan kohteen. Lisäksi kirjataan erikseen toiminnan kehittämisen ideoita ja työkaluja. Anna-Liisa Syrjänen ja Kari Kuutti (2011) esittivät, että työntekijät voisivat itse kehittää tietoteknisiä työkaluja muutoslaboratorion tulosten perusteella. Tutkielman idean mukaan loppukäyttäjäkehittäjän sijaan kehitys tapahtuisi työntekijöiden ja ohjelmoijien yhteistyönä, jossa operaation malli toimisi rajakohteena (Star & Griesemer 1989). Rajakohteena operaation malli mahdollistaisi tasapuolisen keskustelun kehitettävistä ohjelmiston ominaisuuksista.

Muutoslaboratorion ja tutkielman viitekehyksen menetelmien yhdistelmällä, voitaisiin kehittää sekä toimintaa että ohjelmistoa rinnakkain. Iteratiivisuus on nykyisen ohjelmistokehityksen vallitseva piirre (Ralph 2015), joten muutoslaboratorioistunnot voisi liittää ohjelmistokehitysprosessin pisteisiin, joissa arvioidaan iteraatiossa kehitettyä ja suunnitellaan seuraavaa. Teorian kannalta tällainen asetelma mahdollistaisi tutkielman

Bråtenin (1973) mallimonopolin ratkaisun sovelluksen, jossa työntekijä ja ohjelmoija luovat yhteistä ohjelman teoriaa omien malliensa näkökulmasta, empiiristä tutkimista. Teknisesti kyseessä olisi menetelmien soveltamisesta viitekehyksen tavoitteiden mukaisesti, joka onnistuessaan olisi ohjelmistokehityksen luonteeseen (Brooks 1986) vaikuttava muutos. Toiminnan kehittäminen tulisi ohjelmistokehityksen kohteeksi.

6 YHTEENVETO

Tutkielman tavoitteena oli mahdollistaa, että asiakas ja ohjelmoija voivat kehittää ohjelmaa yhdessä. Asiakas olisi mukana valitsemassa ongelman ratkaisuja ja vaikuttamassa millaisiksi ne muotoutuvat. Osallistuminen on mahdollista vain, jos ohjelmasta käytävä keskustelu perustuu asiakkaan tuntemaan toimintaan. Ohjelmoijan kannalta haastavaa on hänelle tuntemattomaan toimintaan liittyvien ongelmien ratkaiseminen. Hän osaa parhaiten lähestyä ongelmaa tietotekniikan ominaisuuksien kautta. Tutkielma pyrki löytämään ohjelmointimenetelmiä, jotka tukisivat jaetun käsityksen muodostamista toimintaa tukevasta ohjelmasta.

Asiakkaan ja ohjelmoijan yhteistyötä tukevien ohjelmointimenetelmien löytämiseksi luotiin viitekehys, joka määritteli tällaisten menetelmien piirteet. Tutkimusoletuksena oli, että tällaisia menetelmiä olisi myös olemassa. Viitekehys koostuu kolmesta mallista: toiminnan kohde, sen käsittely vuorovaikutuksessa tietokoneen kanssa ja em. toteuttavat ohjelmointikielen käsitteet. Kaksi ensimmäistä ovat sellaisia, että asiakas voi ottaa niihin kantaa. Malleilla piti myös olla eksplisiittinen esitys ohjelman lähdekoodissa, jotta kehittämistä voitaisiin pitää yhteisenä. Viitekehyksen mallien mukaiset menetelmät löytyivät neljästä merkittävimmästä ohjelmointiparadigmasta. Näistä olio-ohjelmoinnista löytyi lisäksi menetelmä, joka sisälsi viitekehyksen kaikki mallit.

Verrattaessa tutkielmassa luotua viitekehystä taustakirjallisuuteen syntyi idea asiakkaan ja kehittäjän yhteistyötä tukevasta ohjelmankehitysprosessista. Sekä ohjelman että toiminnan kehittämiseen löytyy menetelmiä, jotka etenevät iteratiivisina prosesseina. Yhdistämällä iteraatioiden alkupisteet, voitaisiin kehittää toimintaa ja sitä tukevaa ohjelmistoa samanaikaisesti. Tutkielman menetelmät tukisivat prosessien yhdistämistä, koska niiden avulla voitaisiin keskustella tulevasta tietokonetuetusta toiminnasta. Kehitysprosessi mahdollistaisi asiakkaan ja kehittäjän yhteisen käsityksen muodostumisen empiirisen tutkimisen. Asiakkaan ja kehittäjän organisaatiot saisivat tästä menetelmän, joka edistäisi paremmin tai pienemmällä panostuksella toimintaa tukevien ohjelmistojen kehittämistä. Koska prosessimallit ovat olemassa ja tutkielman viitekehyksen mukaisia menetelmiä löytyi, olisi tässä mahdollisuus jatkotutkimukselle.

LÄHTEET

Abadi, M. & Cardelli, L. (1996). A Theory of Objects. New York: Springer.

Abelson, H. & Sussman, G. J. (1996). Structure and interpretation of computer programs. 2nd ed. Cambridge: MIT Press.

Alebrahim, A. & Heisel, M. (2017). Bridging the Gap Between Requirements Engineering and Software Architecture. Wiesbaden: Springer.

Ambler, A. L., Burnett, M. M. & Zimmerman, B. A. (1992). Operational versus definitional: A perspective on programming paradigms. Computer, 25(9), s. 28-43.

Armstrong, D. J. (2006). The quarks of object-oriented development. Communications of the ACM, 49(2), s. 123-128.

Avgeriou, P., Grundy, J., Hall, J.G., Lago, P. & Mistrík, I. (toim.) (2011). Relating software requirements and architectures. Heidelberg: Springer.

Avison, D. & Fitzgerald, G. (2003). Information systems development: methodologies, techniques and tools. Maidenhead: McGraw Hill.

Avison, D.E. & Wood-Harper, A.T. (1990). Multiview: An Exploration in Information Systems Development. Oxford: Blackwell.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. Communications of the ACM 21(8), s. 613-641.

Bal, H.E. & Grune, D. (1994). Programming language essentials. Amsterdam: Addison-Wesley.

Baragry, J. (1996). An initial comparison of software and engineering designs of auto-

motive cruise control systems. Teoksessa Proceedings of Australian Software Engineering Conference. New York: IEEE, s. 192-202.

Baragry, J. (2000). Understanding software engineering: from analogies with other disciplines to philosophical foundations. B.Sc. (Hons). La Trobe University.

Baragry, J. & Reed, K. (1998). Why is it so hard to define software architecture? Teoksessa Proceedings of Software Engineering Conference Asia Pacific. New York: IEEE, s. 28-36.

Baragry, J. & Reed, K., 2001. Why we need a different view of software architecture. Teoksessa Proceedings of Working IEEE/IFIP Conference on Software Architecture. New York: IEEE, s. 125-134.

Beck, K. (1999). Extreme programming explained: embrace change. Boston: Addison-Wesley.

Beck, K. & Cunningham, W. (1989). A laboratory for teaching object oriented thinking. ACM Sigplan Notices 24(10), s. 1-6.

Berger, P. L. & Luckmann, T. L. (1966). The Social Construction of Reality - A Treatise in the Sociology of Knowledge. New York: Penguin Books.

von Bertalanfy, L. (1969). General Systems Theory: Foundations, Development, Applications - Revised Edition. New York: George Brazilier.

Blum, B.I. (1996). Beyond programming: to a new era of design. New York: Oxford University Press.

Blum, B.I. & Sigillito, V. G. (1985). Some philosophic foundations for an environment for system building. Teoksessa Proceedings of the 1985 ACM annual conference on The range of computing: mid-80's perspective, s. 516-523.

Boehm, B. & Turner, R., (2003). Balancing agility and discipline: A guide for the perplexed. Boston: Addison-Wesley.

Bramer, M. (2013). Logic programming with Prolog. London: Springer.

Bratko, I. (2011). Prolog programming for artificial intelligence, 4. painos. London: Pearson.

Bratteteig, T., Bødker, K., Dittrich, Y., Mogensen, P. H. & Simonsen J. (2013). Methods: Organising principles and general guidelines for Participatory Design Projects. Teoksessa Simonsen, J. & Robertson, T. (toim.) International Handbook of Participatory Design. New York: Routledge.

Brooks, F. P. (1986). "No Silver Bullet — Essence and Accident in Software Engineering". Teoksessa Kugler, H., -J. (toim.) Information processing 1986, Proceedings of the IFIP Tenth World Computing Conference. Amsterdam: Elsevier, s. 1069-1076.

Brooks, F. P. (1995). The Mythical Man-Month: essays on software engineering. Boston: Addison-Wesley.

Brooks, F. P. (2010). The Design of Design: Essays from a Computer Scientist. London: Pearson.

Brynjolfson, E. & Hit, L. (2002). Computing Productivity: Firm-Level Evidence. MIT Working paper 4210-01, http://ebusiness.mit.edu/erik/Brynjolfsson-Hitt-Computing%20Productivity.doc, luettu 13.2.2018.

Brynjolfson, E. & Yang, S. (1996). Information Technology and Productivity: A Review of the Literature. Advances in Computers 43, s. 179-214.

Bråten, S. (1973). Model Monopoly and Communication: Systems Theoretical Notes

On Democratization. Acta Sociologica 16, s. 98-107.

Czaplicki, E., & Chong, S. (2013). Asynchronous functional reactive programming for GUIs. ACM SIGPLAN Notices 48(6), s. 411-422.

Checkland, P. (1981) Systems thinking, Systems practice. Chichester: John Wiley & Sons.

Checkland, P. (1999). Soft Systems Methodology: a 30-year retrospective. Lisätty teokseen Checkland, P. & Scholes, J. (1990). Soft Systems Methodology in Action. Uusintapainos syyskuu 2005. Chichester: John Wiley & Sons, A1-A65.

Checkland, P. & Holwell, S. (1998) Information, Systems and Information Systems: making sense of the field. Chichester: John Wiley & Sons.

Checkland, P. & Holwell, S. (2004). "Classic" OR and "soft" OR - an asymmetric complementarity. Teoksessa P. Michael (toim.) Systems modelling. Chichester: John Wiley & Sons.

Checkland, P. & Poulter, J. (2006). Learning for Action: A Short Definitive Account of Sost Systems Methodology and its use for Practitioners, Teachers and Students. Chichester: John Wiley & Sons.

Checkland, P. & Scholes, J. (1990). Soft Systems Methodology in Action. Chichester: John Wiley & Sons.

Chen, H. H., Lin, I. P. & Wu, C. P. (1986). A logic programming approach to frame-based language design. Teoksessa Proceedings of 1986 ACM Fall joint computer conference. New York: IEEE, s. 223-228.

Church, A. (1936). An unsolvable problem of elementary number theory.
American Journal of Mathematics 58, s. 345-363.

Coggins, J. M. (1996). Subject-Oriented Programming. Teoksessa G. H. Jacoby & J. Barnes (toim.) Astronomical Data Analysis Software and Systems V, ASP Conference Series 101.

Christensen, H. B., & Hansen, K. M. (2011). Towards architectural information in implementation (NIER track). Teoksessa Proceedings of the 33rd International Conference on Software Engineering. New York: ACM, s. 928-931.

Cleland-Huang, J., Hanmer, R. S., Supakkul, S., & Mirakhorli, M. (2013). The twin peaks of requirements and architecture. IEEE Software, 30(2), s. 24-29.

Colburn, T. (2000). Philosophy and computer science. New York: London.

Colburn, T., & Shute, G. (2007). Abstraction in computer science. Minds and Machines, 17(2), s. 169-184.

Colmerauer, A., & Roussel, P. (1996). The birth of Prolog. Teoksessa History of programming languages-II. New York: ACM, s. 331-367.

Dahl, O. J. & Nygaard, K. (1966). SIMULA: an ALGOL-based simulation language. Communications of the ACM 9(9), s. 671-678.

Dahl, O. J., Myhrhaug, B. & Nygaard, K. (1968). Some features of the SIMULA 67 language. Teoksessa Proceedings of the second conference on Applications of simulations, s. 29-31.

Dahlbom, B., & Mathiassen, L. (1993). Computers in context: The philosophy and practice of systems design. Cambridge: Blackwell.
Daniels, H. (2008). Vygotsky and research. New York: Routledge.

Denning, P. & Dargan, P. (1996). Action-Centered Design. Teoksessa Winograd, T.

(toim.) Bringing Design to Software. New York: Addison Wesley, s. 105-120.

Dittrich, Y., Floyd, C., & Klischewski, R. (toim.). (2002). Social Thinking-Software Practice. Cambridge, MA: Mit Press.

Dybvig, R. K. (2003). The SCHEME programming language. Cambridge, MA: Mit Press.

Engeström, Y. (2004). Ekspansiivinen oppiminen ja yhteiskehittely työssä. Tampere: Vastapaino.

Evans, E. (2004). Domain-driven design: tackling complexity in the heart of software. Upper Saddle River: Addison-Wesley.

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., & Tobin-Hochstadt, S. (2015). The Racket manifesto. Teoksessa LIPIcs-Leibniz International Proceedings in Informatics 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Fikes, R., & Kehler, T. (1985). The role of frame-based representation in reasoning. Communications of the ACM, 28(9), s. 904-920.

Floyd, C. (1988). A Paradigm Change in Software Engineering. ACM SIGSOFT, Software Engineering Notes 13(2), s. 25-38.

Fowler, M. (2002). Patterns of enterprise application architecture. Boston: Addison-Wesley.

Fowler, M. (2010). Domain-Specific Languages. Boston: Pearson Education.

Gandy, R. (1988). The Confluence of Ideas in 1936. Teoksessa R. Heerken (toim.) The Universal Turing Machine - A Half-Century Survey. New York: Oxford University

Press, s. 51-111.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Gavanelli, M., & Rossi, F. (2010). Constraint logic programming. In A 25-year perspective on logic programming. Berlin: Springer, s. 64-86.

Gimblett, A., & Thimbleby, H. (2010). User interface model discovery: towards a generic approach. Teoksessa Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems. New York: ACM, s. 145-154.

Gordon, M., Milner, R., Morris, L., Newey, M., & Wadsworth, C. (1978). A metalanguage for interactive proof in LCF. Teoksessa Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. New York: ACM, s. 119-130.

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). The Java Language Specification (Java SE 8 Edition). California: Oracle.

Harsu, M. (2005). Ohjelmointikielet: Periaatteet, käsitteet, valintaperusteet, Helsinki: Talentum.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design Science in Information Systems Research. MIS Quarterly 28(1), s. 75-105.

Hirschheim, R. & Klein, H. K. (1989). Four paradigms of information systems development. Communications of the ACM 32(10), s. 1199.

Hu, Z., Hughes, J., & Wang, M. (2015). How functional programming mattered. National Science Review, 2(3), s. 349-370.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. ACM Computing Surveys 21(3), s. 359-411.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J. & Kieburtz, D. (1992). Report on the programming language Haskell: a non-strict, purely functional language version 1.2. ACM SIGPLAN notices, 27(5), s. 1-164.

Hudak, P. (1996). Building domain-specific embedded languages. ACM Computing Surveys (CSUR), 28(4), 196.

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. (2007). A history of Haskell: being lazy with class. Teoksessa Proceedings of the third ACM SIGPLAN conference on History of programming languages ACM, s. 12-1.

Hughes, J. (1989). Why functional programming matters. The Computer journal, 32(2), s. 98-107.

Indurkhya, N. (2014). Natural Language Processing. Teoksessa Gonzalez, T. & Diaz-Herrera, J. Computing handbook: computer science and software engineering (3. painos). Chapman & Hall/Crc.

Irmak, N. (2012). Software Is An Abstract Artifact. Grazer Philosophische Studien, 86(1), s. 55-72.

Jacobson, I., Ng, P. W., McMahon, P.E., Spence, I. & Lidman, S. (2013). The Essence of Software Engineering: Applying the SEMAT Kernel. New Jersey: Pearson.

Jackson, M. A. (1975). Principles of program design. London: Academic press.

Jackson, M. A. (1983). System development. Englewood Cliffs: Prentice-Hall.

Jackson, M. (2002). JSP in Perspective. Teoksessa Software pioneers. Berlin: Springer,

s. 480-493.

Javed, M. A., & Zdun, U. (2014). A systematic literature review of traceability approaches between software architecture and source code. Teoksessa Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM, 16.

Jipping, M. J. & Bruce, K. (2014). Imperative Language Paradigm. Teoksessa Gonzalez, T. & Diaz-Herrera, J. Computing handbook: computer science and software engineering (3. painos). Chapman & Hall/Crc.

Kay, A. C. (1993). The early history of Smalltalk. ACM SIGPLAN Notices 28(3), s. 69-95.

King, J. L. & Lyytinen, K. (2006). Information Systems: The State of the Field. West Sussex: John Wiley & Sons.

Kleene, S. C. (1936). General recursive functions of natural numbers. Mathematische Annalen, 112(1), s. 727-742.

Korsgaard, H., Klokmose, C. N., & Bødker, S. (2016). Computational alternatives in participatory design: putting the t back in socio-technical research. Teoksessa Proceedings of the 14th Participatory Design Conference: Full papers-Volume 1. New York: ACM, s. 71-79.

Krishnamurthi, S., Hopkins, P. W., McCarthy, J., Graunke, P. T., Pettyjohn, G., & Felleisen, M. (2007). Implementation and use of the PLT Scheme web server. Higher-Order and Symbolic Computation, 20(4), s. 431-460.

Kristensen, B. B., Madsen, O. L. & Møller-Pedersen, B. (2007). The when, why and why not of the BETA programming language. Teoksessa Proceedings of the third ACM SIGPLAN conference on History of programming languages, s. 10-1--10-57.

Kuutti, K. (2003). Searching Knowledge for Design — Nurminen's "Humanistic Perspective" Revisited. Teoksessa Timo Järvi and Pekka Reijonen (toim.) People and Computers: Twenty-one Ways of Looking at Information Systems. Turku Centre for Computer Science, TUCS General Publication 26, s. 29-40.

Lawson, B. (1980). How Designers Think. London: The Architectural Press Ltd.

Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE 68(9).

Lehrmann Madsen, O. & Møller-Pedersen, B. (2010). A Unified Approach to Modeling and Programming. Teoksessa D.C. Petriu, N. Rouquette, Ø. Haugen (toim.) MODELS 2010, Part I, LNCS 6394. Berlin: Springer, s. 1-15.

Lehrmann Madsen, O., Møller-Pedersen B. & Nygaard, K. (1993). Object-Oriented Programming in the BETA Programming Language. New York: Addison-Wesley.

Lilienthal, C., & Züllighoven, H. (1997). Application-oriented usage quality: the tools and materials approach. Interactions, 4(6), s. 35-41.

Luukkonen, I., Toivanen, M., Mursu, A., Saranto, K. & Korpela, M. (2013). Researching an activity-driven approach to information systems development. Teoksessa Handbook of research on ICTs and management systems for improving efficiency in healthcare and social care. Hershey: IGI-Global, s. 431-450.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. Communications of the ACM 3(4), s. 184-195.

McCarthy, J. (1981). History of LISP. In History of programming languages I. New York: ACM, s. 173-185.

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM computing surveys, 37(4), s. 316-344.

Meyer, B. (2009). Software Architecture: Object-Oriented Versus Functional. Teoksessa Spinellis, D. & Gousios, G. (toim.) Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design. Sebastopol: O'Reilly Media, s. 315-348.

Minsky, M. (1974). A framework for representing knowledge. Cambridge, MA: MIT.

Mooney, J. G., Gurbaxani, V. & Kramer, K. L. (1996). A Process Oriented Framework for Assessing the Business Value of Information Technology. ACM SIGMIS Database 27(2), s. 68-81.

Moor, J. H. (1978). Three myths of computer science. The British Journal for the Philosophy of Science, 29(3), s. 213-222.

Mot Tietotekniikan Liiton Atk-sanakirja. Helsinki: Kielikone. <https://mot.kielikone.fi> 14.2.2018.

Mumford, E. (1985). Defining System Requirements to Meet Business Needs: A Case Study Example. The Computer Journal 28(2), s. 97-104.

Munkvold, B. E. (2000). Tracing the Roots: The Influence of Socio-Technical Principles on Modern Organisational Change Practices. Teoksessa E. Coakes, D. Willis & R. Lloyd-Jones (toim.) The New SocioTech. London: Springer.

Naur, P. (1992). Programming as Theory Building. Teoksessa Computing: A Human Activity. New York: ACM Press/Addison-Wesley, s. 37-48.

von Neumann, J. (1945). First Draft of a Report on the EDVAC. IEEE Annals of the History of Computing 1993, 15(4), s. 27-75.

Nurminen, M. I. (1988). People or Computers: Three Ways of Looking at Information Systems. Lund: Studentlitteratur.

Nurminen, M. I. & Forsman, U. (1994). Reversed Quality Life Cycle Model. Teoksessa G. E. Bradley & H. W. Hendrick (toim.) Human factors in organizational design and management – IV. Amsterdam: Elsevier, s. 17-27.

Nuseibeh, B. (2001). Weaving Together Requirements and Architectures. Computer 34 (3), s. 115-117.

Nygaard, K. (1986). Program Development as a Social Activity. Teoksessa Proceedings from the IFIP 10th World Computer Congress. 1.8-5.8.1986. Dublin Ireland, s. 189-198.

Parnas, D. & Clements, P. (1986). A Rational Design Process: How and why to fake it. IEEE Transactions on Software Engineering archive 12(2), s. 251-257.

Perlis, A. J. (1982). Epigrams on programming. SIGPLAN Notices, 17(9), s. 7-13.

Post, E. L. (1936) Finite Combinatory Processes-Formulation 1. The Journal of Symbolic Logic 1(3), s. 103-105.

Wirfs-Brock, R., & McKean, A. (2003). Object design: roles, responsibilities, and collaborations. Boston: Addison-Wesley.

Ralph, P. (2015). The sensemaking-coevolution-implementation theory of software design. Science of Computer Programming, 101, s. 21-41.

Reijonen, P. (2003). Software Development and IS Use. Teoksessa Timo Järvi and Pekka Reijonen (toim.) People and Computers: Twenty-one Ways of Looking at Information Systems. Turku Centre for Computer Science, TUCS General Publication 26, s. 305-319.

Reisin, F. M. (1992). Anticipating reality construction. Teoksessa C. Floyd, R. Keil Slawik, R. Budde & H. Züllighoven (toim.) Software Development and Reality Construction. New York: Springer, s. 312-325.

Ritchie, D. M. (1996). The development of the C programming language. In History of programming languages II. New York: ACM, 671-698.

Robinson, J. A. (2000). Computational logic: Memories of the past and challenges for the future. Computational Logic — CL 2000 Lecture Notes in Computer Science 1861, s. 1-24.

Rubin, A. (2002). Pehmeä systeemimetodologia tulevaisuudentutkimuksessa. Teoksessa Kamppinen, M., Kuusi, O. & Söderlund, S. (toim.) Tulevaisuudentutkimus: Perusteet ja Sovelluksia. Helsinki: Suomalaisen Kirjallisuuden Seura.

Sackman, H. (1974). Computers and Social Options. Teoksessa E. Mumford and H. Sackman (toim.) Human Choice and Computers. Amsterdam: North-Holland.

Scott, M. L. (2009). Programming Language Pragmatics. Burlington: Morgan Kaufmann Publishers.

Soare, R. I. (2016). Turing computability: Theory and applications. Berlin: Springer.

Star, S. L., & Griesemer, J. R. (1989). Institutional ecology, translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. Social studies of science, 19(3), s. 387-420.

Steele Jr, G. L. & Gabriel, R. P. (1993). The evolution of Lisp. ACM SIGPLAN Notices 28(3), s. 231-270.

Stroustrup, B. (1984). The C++ programming language: reference manual (No. BELL-CSTR-108). Murray Hill: AT&T Bell Laboratories.

Stroustrup, B. (1994). The design and evolution of C++. Boston: Addison-Wesley.

Suchman, L.A. (1987). Plans and situated actions: the problem of human-machine communication. New York: Cambridge University Press.

Syrjänen, A.-L. & Kuutti, K. (2011). From technology to domain: the context of work for end-user development. Teoksessa Proceedings of the 2011 iConference, s. 244-251.

Sørgaard, P. (1988). Object-Oriented Programming and Computerised Shared Material. Proceedings of the European Conference on Object-Oriented Programming, s. 319-334.

Thimbleby, H. (2007). Press On: Principles of Interaction Programming. Boston: MIT Press.

Thimbleby, H., Gimblett, A., & Cauchi, A. (2011). Buffer Automata: A UI architecture prioritising HCI concerns for interactive devices. Teoksessa Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. New York: ACM, s. 73-78.

Thompson, S. (2011). Haskell: the Craft of Functional Programming. Boston: Addison-Wesley.

Torvinen, V. & Korteinen, B. (1997). Problem formulation in IS development methodologies: Towards a constructive view through a deconstructive approach. Teoksessa R. Galliers, C. Murphy, H. Hansen, R. O'Callaghan, S. Carlsson & C. Loebbecke (toim.) Proceedings of the 5th European Conference on Information Systems II, s. 647-658.

Turing, A. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. Teoksessa Proceedings of the London Mathematical Society 42(2), s. 230-265.

Turner, R. (2014). Programming languages as technical artifacts. Philosophy & Technology, 27(3), s. 377-397.

Ungar, D., & Smith, R. B. (1987). Self: The power of simplicity. ACM SIGPLAN Notices 22(12), s. 227-242.

Virkkunen, J., & Newnham, D. S. (2013). The change laboratory. A tool for collaborative development of work and education. Rotterdam: Sense Publishers.

Ward, M. (1994). Language Oriented Programming. Software – Concepts and Tools 15, s. 147-161.

West, D. (2004). Object Thinking. Redmond: Microsoft Press.

Winograd, T. (1979). Beyond programming languages. Communications of the ACM 22 (7), s. 391-401.

Winograd, T. & Flores, F. (1986). Understanding Computers and Cognition: a New Foundation for Design. Norwood: Ablex Publishing Corporation.

Wlaschin, S. (2017). Domain Modeling Made Functional. Dallas: Pragmatic Bookshelf.

Woods, E. & Rozanski, N. (2010). Unifying software architecture with its implementation. Teoksessa Proceedings of the Fourth European Conference on Software Architecture: Companion Volume. New York: ACM, s. 55-58.

Zhou, N. F. (2001). Authoring graphics-rich and interactive documents in CGLIB: a constraint-based graphics library. Teoksessa Proceedings of the 2001 ACM Symposium on Document engineering. New York: ACM, s. 28-37.

Züllighoven, H. (2005). Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools & Materials Approach. San Fransisco: Morgan

Kaufmann.