Tampereen yliopisto

Laura Fadjukoff

TOIMIALAKOHTAINEN MALLINNUS SOVELLUSKEHITYKSESSÄ

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Outi Sievi-Korte
Tarkastaja: David Hästbacka
marraskuu 2021

TIIVISTELMÄ

Laura Fadjukoff: Toimialakohtainen mallinnus sovelluskehityksessä
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Marraskuu 2021

Sovellukset ovat nykyään olennainen osa lähes kaikkia palveluja ja tuotteita, ja uusia sovelluksia täytyykin kehittää kiihtyvällä tahdilla. Pula osaavista ohjelmoijista rajoittaa kuitenkin alan kehitystä. Ohjelmointityön helpottamiseksi ohjelmistosuunnittelu on kehittynyt jatkuvasti yhä abstraktimmaksi ja intuitiivisemmaksi. Samalla rutiininomaiset ohjelmointivaiheet ovat yhä enenevissä määrin jääneet automaattisesti suoritettaviksi, jolloin osaavien kehittäjien aika säästyy vaativampiin tehtäviin. Nykyisin on myös tarjolla niin kutsuttuja low code ja no code -sovelluskehitysalustoja, joilla sovelluksia pystyy luomaan jopa kirjoittamatta riviäkään koodia.

Yritykset voivat myös luoda itselleen omaan toimialueeseensa sopivia automaattisia kehitysratkaisuja esimerkiksi toimialakohtaisen mallinnuksen avulla. Eräs esimerkki toimialakohtaisuutta hyödyntävästä yrityksestä on jyväskyläläinen SoulCore, joka on kehittänyt omia tarpeitaan vastaavan automaattisen sovellusgeneraattorin.

Tämä tutkimus on tapaustutkimus, jonka tarkoituksena oli vertailla sovelluskehityksen tehokkuutta perinteisesti ohjelmoiden ja SoulCoren itselleen kehittämää sovellusgeneraattoria käyttäen. Perinteisen ohjelmoinnin työkaluina käytettiin Microsoftin Visual Studiota ja ASP.NET-ohjelmistokehystä.

Tutkimuksen tulokset vahvistivat, että toimialakohtainen mallinnus tehostaa sovelluskehitystä huomattavasti. Toimialakohtaista mallinnusta käyttäen tietokannan luominen ja sovelluslogiikan ohjelmointi hoituivat täysin automaattisesti käyttäjän tekemän mallin pohjalta, ja käyttöliittymän muokkaaminenkin oli huomattavasti yksinkertaisempaa kuin perinteisesti käsin ohjelmoiden. Esimerkkisovelluksen luominen toimialakohtaista sovellusgeneraattoria käyttäen vei näistä syistä alle 20 % työajasta, joka kului samaan tehtävään perinteisesti ohjelmoiden. Viikon työmäärä voitiin siten suorittaa vajaassa yhdessä työpäivässä.

Avainsanat: sovelluskehitys, toimialakohtainen mallinnus, kooditon kehitys, vertaileva tapaustutkimus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

ABSTRACT

Laura Fadjukoff: Domain-Specific Modeling in Application Development
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
November 2021

Today, applications are an essential part of almost all services and products, and new applications need to be developed at an accelerating pace. However, a lack of skilled programmers restrains the development. To make programming easier, software design has evolved to become more abstract and intuitive. At the same time, routine programming steps have increasingly been left to be performed automatically, saving the time of skilled developers for more demanding tasks. Today, so-called low code and no code application development platforms are also available, with which applications can be created even without writing a line of code.

Companies can also create automated development solutions suitable for their own domain, for example with domain-specific modeling. One example of a company using domain-specificity is SoulCore from Jyväskylä, which has developed an automatic application generator that meets its own needs.

The present study was a case study comparing the effectiveness of application development between traditional programming and using the application generator developed by SoulCore. Microsoft Visual Studio and the ASP.NET software framework were used as tools for traditional programming.

The results confirmed that domain-specific modeling greatly enhances application development and makes it more efficient. Using domain-specific modeling, database creation and application logic programming were handled fully automatically based on a user-created model, and customizing the user interface was also much simpler than with traditional manual programming. Creating a sample application with a domain-specific application generator was possible in less than 20 % of the working time required for manual programming. The weekly workload could thus be reduced to less than one working day.

Keywords: Application Development, Domain-Specific Modeling, No Code Development, Comparative Case Study

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Opintoni sujuivat vauhdikkaasti perheen perustamisen ja työssäkäynnin ohellakin, kunnes toisen lapseni syntyessä koin tarvetta priorisoida aikaa perheen kanssa, ja palata diplomityön tekemiseen myöhemmin. Aikaa onkin ehtinyt kulua, ja nyt viimeistelen tätä työtä jo viiden lapsen äitinä.

Tämä opinnäyte sai alkunsa SoulCore-yrityksen kanssa käydyistä keskusteluista. Haluan kiittää lämpimästi sen omistajaa ja toimitusjohtajaa Tiina Vestmania kaikesta tuesta työn eri vaiheissa. Lämmin kiitos myös vanhemmalle konsultille Risto Pohjoselle, joka toimi työni ohjaajana SoulCoressa ja SoulCoren pääarkkitehdille Jarno Leikkaalle. Ilman teidän tukeanne ja kärsivällisyyttänne työni valmistuminen ei olisi ollut mahdollista. Muutettuani perheineni Sveitsiin ja uusien yllätysten myötä täällä, vielä keskeneräinen opinnäyte jäi taka-alalle ja työn tekeminen joutui harmillisesti tauolle. Teidän antamanne arvokas tuki erityisesti jo valmistuneessa empiirisessä vaiheessa on kuitenkin kannustanut minua jatkamaan työn muiden vaiheiden tekemistä täällä.

Sain työlleni ohjaajaksi Outi Sievi-Kortteen, jolta sain työn käynnistyessä hyviä vinkkejä ja lähteitä. Kiitos! Ohjaustyö jäi pitkähkölle tauolle muutettuani pois Suomesta, ja halusinkin palata keskusteluun ohjaajan kanssa vasta, kun työ olisi jo edennyt niin pitkälle, että voin itsekin uskoa sen nyt todella valmistuvan. Siltä ei aina tuntunut. Kiitos omalle äidilleni siitä, että uskoit minuun myös silloin, kun itsestäni tuntui mahdottomalta enää palata tämän opinnäytteen tekemiseen ja sen valmiiksi kirjoittamiseen.

Tätä opinnäytettä on siis tehty perheen kasvamisen ja kasvattamisen ohella. Vaikka se on edennyt hitaasti, en kadu perheen ja lasten asettamista tänä aikana etusijalle elämässäni, lapset ovat aina elämässä tärkeintä. Kiitos teille jokaiselle rakkaalle: Manna, Emmi, Tilli, Nella ja Anttu sekä ihanalle miehelleni Tonille! Tiedän että tekin osaatte iloita äidin opintojen viimein päättyessä.

Zürichissä, 1.11.2021

Laura Fadjukoff

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. SOVELLUSKEHITYS 3
   2.1 Ohjelmointikielten kehittyminen 3
   2.2 Kooditon kehitys 7
   2.3 Ohjelmointityön vaatimukset 10
3. TOIMIALAKOHTAINEN MALLINNUS 13
   3.1 Toimialakohtaisuus sovelluskehityksen lähtökohtana 13
   3.2 Toimialakohtaisen mallinnuksen hyödyt 14
   3.3 Toimialakohtaisen mallinnuksen arkkitehtuuri 17
   3.4 Toimialakohtaisen mallinnuksen toteuttaminen 20
4. TUTKIMUKSEN KUVAUS 22
   4.1 Tutkimuskysymykset 22
   4.2 Tutkimusmenetelmä 22
   4.3 Automatisoitu sovellustuotanto SoulCorella 26
   4.4 Esimerkkisovellus 28
   4.5 Tutkimuksen toteutus 29
5. TUTKIMUKSEN SUORITTAMINEN 31
   5.1 Esimerkkitoteutus 1: Perinteinen ohjelmointi 31
   5.2 Esimerkkitoteutus 2: Mallista generoiden 33
   5.3 Toteutustapojen vertailu 35
6. TULOSTEN TARKASTELU 37
7. YHTEENVETO 40
LÄHTEET 41

LYHENTEET JA MERKINNÄT

DSM	engl. Domain Specific Modeling, toimialakohtainen mallinnus
MVC	engl. model-view-controller, malli-näkymä-käsittelijä-jakoon perustuva ohjelmistoarkkitehtuuri
UML	engl. Unified Modeling Language (yhdenmukaistettu mallinnuskieli), ohjelmistotuotannossa yleisesti käytetty mallinnuskieli
vähäkoodinen	engl. low code
kooditon	engl. no code

1. JOHDANTO

Nykyisessä voimakkaasti digitalisoituneessa yhteiskunnassa sovellukset ovat osa lähes kaikkia palveluita ja tuotteita. Tarvitaan yhä monimutkaisempia tietojärjestelmiä, ja ne täytyy saada tuotettua nopeasti (Krach 2019, Räty 2019). Pärjätäkseen jatkuvasti kovenevassa kilpailussa ohjelmistokehitysmarkkinoilla yritysten täytyy tehostaa ohjelmistotuotantoaan (Holtkamp 2015). Se vaatii nopeita, joustavia ja tehokkaita ratkaisuja jatkuvasti muuttuvissa toimintaympäristöissä (Sanchis ym. 2019). Sovelluskehityksen tehokkuudella on tärkeä osa tuotannon tehostamisessa (Sanchis ym. 2019). Sovelluskehitystä on tehostettu kehittämällä uusia ohjelmointikieliä, sovelluskehitysohjelmistoja, suunnittelumalleja ja tuotantomenetelmiä. Pula osaavista ohjelmoijista rajoittaa alan kehitystä (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019).

Automatisoitu sovellustuotanto on yksi tapa, jolla sovelluskehitystä voidaan pyrkiä tehostamaan (Sanchis ym. 2019). Ideana on siis kehittää ohjelma, joka pystyy tuottamaan sovelluskoodia itsenäisesti annettujen määrittelyjen perusteella vähentäen näin ohjelmoijan työtä. Kuten Sanchis ym. (2019) kuvaavat, niin sanotut "low code"- kehitysalustat tarjoavat teknologiamekanismit, jotka helpottavat ja automatisoivat ohjelmistosovellusten kehittämistä yrityksen tarpeiden tukemiseksi ja digitaalisen muutoksen edistämiseksi. Äärimmilleen automatisoidusta sovelluskehityksestä puhutaan myös nimityksellä "no code" (Caballar 2020). Low code tai no code -kehitykselle ei ole vakiintunutta suomenkielistä ilmaisua. Tässä tutkimuksessa käytän niistä käsitteitä *vähäkoodinen* ja *kooditon* sovelluskehitys.

Kooditon ja vähäkoodinen sovelluskehitys ovat jo teollisuuden tuotantokäytössä ja saaneet huomiota menestyvien yritysten käytössä, tunnettuja esimerkkejä ovat kooditonta ohjelmointiprosessia käyttävä AppGyver[1] sekä vähäkoodinen avoimen lähdekoodin tuote JHipster[2]. Eräs sovelluskehityksen automatisointiin erikoistunut yritys on jyväsky-

---

[1] https://www.appgyver.com/

[2] https://www.jhipster.tech/

läläinen vuonna 2014 perustettu SoulCore, jonka toiminta-ajatuksena on tarjota asiakkailleen joustavia ja käyttäjäystävällisiä liiketoimintasovelluksia nopeasti ja kustannustehokkaasti. SoulCore kertoo verkkosivullaan (SoulCore 2019) ideologiakseen, että se haluaa "lyhentää toimitusten läpimenoaikoja, nostaa toimitettavien sovellusten laatua ja tarjota asiakkaalle innostavan ja tehokkaan sovellustoimitusprosessin".

Tämä tutkimus on vertaileva tutkimus, jossa tarkastellaan mahdollisuuksia sovelluskehityksen tehostamiseen toimialakohtaisen automatisoinnin avulla. Tarkoituksena on selvittää, tekeekö toimialakohtainen mallinnus sovelluskehityksestä tehokkaampaa. Tutkimuksessa toteutetaan esimerkkisovellus toisaalta perinteisesti ohjelmoiden, toisaalta hyödyntäen SoulCoren kehittämää sovellusgeneraattoria, ja verrataan näiden tapojen tehokkuutta.

Tutkimuksen kohteena oleva sovelluskehitys, joka hyödyntää sovellusgeneraattoria, toimialakohtaista mallinnusta ja kooditonta ohjelmointia, on eräänlainen uusi askel sovelluskehityksen ja ohjelmointityön kehittämisen pitkässä historiassa (Woo 2020). Tutkimusraportissa käytetäänkin nimitystä "perinteinen ohjelmointi" niistä menetelmistä, joihin tämän uuden ohjelmointimenetelmän tuottamia hyötyjä verrataan. Luvussa 2 esitellään ohjelmointikielten ja ohjelmistotyön kehityskaarta kohti kooditonta sovelluskehitystä sekä nykyohjelmointityön haasteita.

Luvussa 3 määritellään toimialakohtainen mallinnus tässä tutkimuksessa tarkoitettuna sovelluskehityksen lähtökohtana sekä esitellään toimialakohtaisen mallinnuksen hyötyjä, arkkitehtuuria ja toteuttamista.

Tutkimuksen empiirinen osuus esitellään luvussa 4. Siinä esitellään tutkimuskysymykset ja tutkimusmenetelmät sekä sovellusautomaation toimintaa tutkimuksen kohdeyrityksessä SoulCoressa. Lisäksi esitellään tutkimustehtäväksi asetettu esimerkkisovellus ja vertailevan tutkimuksen kulku.

Tutkimuksen suorittaminen ensin perinteisellä ohjelmointitavalla ja sen jälkeen toimialakohtaista mallinnusta ja sovellusgenerointia hyödyntäen kuvataan luvussa 5. Luvussa 6 tarkastellaan tutkimuksen tuloksia. Lopuksi on vielä yhteenveto.

2. SOVELLUSKEHITYS

Sovelluksella tarkoitetaan tietokoneella suoritettavaa ohjelmaa, joka suorittaa tiettyjä tehtäviä. Sovelluskehityksellä rakennetaan sovelluksia, joilla pyritään parantamaan esimerkiksi yritysten liiketoimintaa, asiakaspalvelua tai yksityishenkilöiden arkea. Sovelluksia on esimerkiksi verkkosivuilla tai mobiililaitteissa, mutta myös älykelloissa tai muissa laitteissa. Jotkut ohjelmistoalan yritykset tarjoavat räätälöityä sovelluskehitystä toisille yrityksille, joiden tarpeisiin valmiina löytyvät sovellukset eivät sovi. Uusi sovellus suunnitellaan ja kehitetään asiakkaan tarpeiden ja toiveiden perusteella.

Sovelluskehitys etenee useiden eri toimintojen ja elementtien prosessina, jotka on suunniteltu ohjelmistojärjestelmien suunnitteluun, kehittämiseen ja ylläpitoon. Ohjelmistokehitysprosessin jäsentämiseksi, kuvaamiseksi ja määrittelemiseksi on suunniteltu useita ohjelmistoprosessimalleja (Kaur & Sengupta 2011). Keskeinen ohjelmistokehitysprosessin elementti on soveltuvan ohjelmointikielen valitseminen. Ohjelmointikieliä on kehitetty lukuisia eri tarkoituksiin pyrkien näin ohjelmoinnin tehostamiseen eri toimintaympäristöissä.

2.1 Ohjelmointikielten kehittyminen

Ohjelmointia on varhaisimmista ajoista alkaen pyritty eri tavoin kehittämään intuitiivisemmaksi ja siten helppokäyttöisemmäksi (Woo 2020). Varhaisimmat tietokoneohjelmat kirjoitettiin konekielellä eli binäärikoodina. Konekieliä pidetään ohjelmointikielten ensimmäisenä sukupolvena. Koska ihmisen on lähes mahdoton lukea tai kirjoittaa binäärikoodia, kehitettiin symboliset konekielet, joissa binäärikoodia vastaavat selkeät käskyjen nimet ja symboliset muistiosoitteet, jotka sitten käännetään binäärikoodiksi ohjelman suorittamista varten. Symbolisia konekieliä pidetään ohjelmointikielten toisena sukupolvena. Sekä ensimmäisen että toisen sukupolven ohjelmointikielet ovat koneriippuvaisia, koska eri tietokoneilla on eri konekielet. (Harsu 2012)

Ohjelmoinnin helpottamiseksi ja ohjelmien siirrettävyyden mahdollistamiseksi alettiin kehittää yleiskäyttöisempää tapaa ohjelmointiin. Periaatteena oli, että ohjelmoija kirjoittaa ratkaisun abstraktimmalla formaalilla kielellä, jonka jälkeen se käännetään ohjelmallisesti konekielelle. Tätä kutsuttiin automaattiseksi ohjelmoinniksi, koska kääntäjäohjelma suoritti varsinaisen koodauksen ohjelmoijan ohjeiden perusteella. Automaatioon on siis

pyritty ohjelmointialalla jo varsin varhain. Näin alkoivat kehittyä korkean tason ohjelmointikielet eli kolmannen sukupolven kielet, jollaisia ovat lähes kaikki nykyisin käytössä olevat ohjelmointikielet. Korkean tason ohjelmointikielet sisältävät abstrakteja kontrollirakenteita, ja uusia abstrakteja käskyjä voidaan tehdä aliohjelmien avulla. Abstraktimmalla tasolla kirjoitettuja ohjelmia on ihmisen helpompi lukea ja ymmärtää. (Harsu 2012)

Tietotekniikan käyttö on laajentunut todella moniin erilaisiin tarkoituksiin ja sitä sovelletaan jo lähes kaikkiin ihmisten ja organisaatioiden toimintoihin. Kun tavoitteena on ollut ohjelmoinnin intuitiivisuus ja helppokäyttöisyys (Woo 2020), hyvin moninaiset, mutta toisaalta tapauskohtaisesti tarkoin määritellyt tarpeet ovat johtaneet myös erilaisia käyttötarkoituksia varten kehitettyjen, monien erilaisten ohjelmointikielten kehitykseen. Tämä vaikeuttaa alan osaajien koulutusta, kun kukin kieli pitäisi siis opiskella erikseen. Kuten jo edellä todettiin, pula osaavista ohjelmoijista rajoittaakin alan kehitystä (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019). Monet tiettyihin tehtäviin kehitetyistä ohjelmointikielistä ovat kuitenkin olleet käytössä jo kymmeniä vuosia. (Sebesta 2012)

Ensimmäiset digitaaliset tietokoneet kehitettiin 1940- ja 1950-luvuilla tieteellisiä sovelluksia varten. Tuolloin tieteelliset sovellukset käyttivät suhteellisen yksinkertaisia tietorakenteita, mutta sisälsivät runsaasti liukulukujen laskutoimituksia. Yleisimmät tietorakenteet olivat taulukoita ja matriiseja ja yleisimmät kontrollirakenteet olivat silmukoita ja valintoja. Ensimmäiset tieteellisiä sovelluksia varten suunnitellut korkean tason ohjelmointikielet palvelivat siis näitä tarpeita. Erityistä huomiota kiinnitettiin tehokkuuteen, koska kielten kilpailija oli symbolinen konekieli. Ensimmäinen tieteellisiä sovelluksia varten kehitetty ohjelmointikieli oli Fortran. Sitä käytetään edelleen joissain tieteellisissä sovelluksissa, joissa tehokkuus on tärkeää, esimerkiksi avoimen lähdekoodin LAPACK[3]. (Sebesta 2012)

Tietokoneiden käyttö liiketoimintasovelluksiin alkoi 1950-luvulla (Sebesta 2012). Tarpeiden laajentuessa myös ohjelmointikieliä piti kehittää hyvin erilaisiin tarpeisiin. Kun asiantunteva kehittäjä määrittelee tietyn sovellusalueen säännöt kieleksi, hän tuottaa samalla kehitysohjeet ja parhaat käytännöt kaikkien kyseistä kieltä käyttävien kehittäjien käyttöön (Kelly & Tolvanen 2008). Liiketoiminnan kielten kehityksessä keskeistä oli yksityiskohtaisten raporttien tuottaminen, desimaalilukujen ja merkkien täsmällinen tallentaminen ja

---

[3] http://www.netlib.org/lapack/

kuvaaminen ja desimaalilukujen laskutoimitusten määrittäminen (Sebesta 2012). Näitä tarkoituksia varten kehitettiin sekä uusia tietokoneita että niihin uusia ohjelmointikieliä (Sebesta 2012). Ensimmäinen menestynyt korkean tason kieli kaupan alalle oli COBOL, jonka alkuperäinen versio ilmestyi vuonna 1960. Se on edelleen yleisimmin käytetty kieli näissä sovelluksissa (Sebesta 2012).

Tekoäly on tietokonesovellusten osa-alue, joka pyrkii matkimaan ihmismäistä ajattelua tai tekemään rationaalisia päätöksiä itsenäisesti (Russell & Norvig 2016). Tekoälysovelluksille on ominaista pikemminkin symbolisten kuin numeeristen laskelmien käyttö (Russell & Norvig 2016, Sebesta 2012). Tekoälyohjelmointi vaatii joskus enemmän joustavuutta kuin muut ohjelmointialueet, esimerkiksi mahdollisuus luoda ja suorittaa koodisegmenttejä toteutuksen aikana on tarpeen joissain tekoälysovelluksissa (Sebesta 2012). Ensimmäinen laajasti käytetty tekoälysovelluksia varten kehitetty ohjelmointikieli oli funktionaalinen kieli Lisp, joka ilmestyi vuonna 1950-luvun lopulla (Russell & Norvig 2016, Sebesta 2012). 1970-luvun alkupuolella kehitettiin vaihtoehtoinen lähestymistapa: looginen ohjelmointi Prologilla (Russell & Norvig 2016, Sebesta 2012). Nykyiset tekoälyt hyödyntävät yleensä koneoppimista, jossa ohjelmat oppivat tekemään päätöksiä itsenäisesti annetun esimerkkidatan pohjalta (Russell & Norvig 2016). Woon (2020) mukaan tekoälyä voidaan käyttää ohjelmointityön tehostamiseen esimerkiksi ennakoimalla ohjelmoinnin seuraavia vaiheita. Woo (2020) pitääkin tekoälyä ja koneoppimista hyödyntävän ohjelmointituen liittämistä nykyaikaisiin ohjelmointialustoihin seuraavana askeleena kohti päämäärää, jossa tekoäly voisi tuottaa koodia automaattisesti.

Tietokoneen käyttöjärjestelmä ja ohjelmoinnin tukityökalut tunnetaan yhdessä sen järjestelmäohjelmistona. Järjestelmäohjelmistoa käytetään melkein jatkuvasti, joten sen on oltava tehokas. Lisäksi siinä on oltava matalan tason ominaisuuksia, jotka mahdollistavat ulkoisten laitteiden ohjelmistorajapintojen kirjoittamisen. Jotkut tietokonevalmistajat kehittivät 1960- ja 1970-luvuilla koneilleen erityisiä konekeskeisiä korkean tason kieliä järjestelmäohjelmistoille. Suurin osa järjestelmäohjelmistoista on kuitenkin nyt kirjoitettu yleisemmillä ohjelmointikielillä, kuten C ja C++. UNIX-käyttöjärjestelmä on kirjoitettu lähes kokonaan C:llä, mikä on tehnyt siitä suhteellisen helpon siirtää eri koneisiin. Jotkut C:n ominaisuuksista tekevät siitä hyvän valinnan järjestelmien ohjelmointiin. Se on matalan tason kieli, suorituskykyinen eikä kuormita käyttäjää monilla turvallisuusrajoituksilla, joita järjestelmäohjelmoijat eivät usein koe tarvitsevansa. (Sebesta 2012)

Toisaalta turvallisuusrajoitusten puute mahdollistaa ohjelmointivirheiden huomaamatta jäämisen, ja joidenkin ohjelmoijien mielestä C onkin liian vaarallinen käytettäväksi

isoissa, tärkeissä ohjelmistojärjestelmissä (Sebesta 2012). Turvallisuusnäkökulma onkin tärkeä lähtökohta joissain toisissa kielissä, esimerkiksi Adassa, joka kehitettiin Yhdysvaltain puolustusministeriön toimeksiannosta 1980-luvulla. Sen kehittämiselle oli tarkat vaatimukset, ja perusajatuksena oli, että ohjelmien täytyy olla luotettavia (Harsu 2012).

Olio-ohjelmoinnin kehittyminen on ollut merkittävä askel ohjelmointikielten kehittymisessä abstraktimmiksi ja siten intuitiivisemmiksi. Oliokielissä ohjelmoija voi määritellä omia abstrakteja tietotyyppejä luokkien avulla. Ensimmäiset oliokielet kehitettiin jo 1960-luvulla. 1980-luvulla kehitettiin C++ laajentamalla C-kieltä olioilla. 1990-luvulla C++:n pohjalta kehitettiin Java, ja myöhemmin Javan ja C++:n pohjalta C#. (Harsu 2012)

Verkkosisältöjen toteutukseen käytetään monenlaisia kieliä lähtien merkintäkielistä kuten HTML yleiskäyttöisiin ohjelmointikieliin kuten Java (Sebesta 2012). Nykyaikaiset verkkosivut sisältävät monenlaisia toiminnallisuuksia ja ovat siten jo monimutkaisia ohjelmistokokonaisuuksia (Kaluža ym. 2018). Verkkosivujen ohjelmoinnissa yleisiä ovat skriptikielet kuten JavaScript ja PHP (Sebesta 2012).

GitHubin raportin (2019) mukaan sen käyttäjien keskuudessa suosituimmat ohjelmointikielet kehittäjien määrän mukaan mitattuna vuonna 2019 olivat:

1. JavaScript
2. Python
3. Java
4. PHP
5. C#
6. C++
7. TypeScript
8. Shell
9. C
10. Ruby

GitHubin (2019) mukaan ohjelmointikielen suosioon vaikuttaa tällä hetkellä vahvimmin se, että haetaan tyyppiturvallisuutta, yhteensopivuutta ja avointa lähdekoodia.

TIOBE-indeksi (TIOBE 2020) mittaa ohjelmointikielten suosituimmuutta hakukonetulosten mukaan. Tammikuussa 2020 sen mukaan suosituimmat ohjelmointikielet ovat:

1. Java
2. C
3. Python
4. C++
5. C#
6. Visual Basic .NET
7. JavaScript
8. PHP
9. Swift
10. SQL

Ohjelmointikielien kehitystyö on siis johtanut jatkuvasti uusien ohjelmointikielten syntyyn, kun eri toimialoille ja tehtäviin on kehitetty juuri niissä tehokkaasti toimivia ohjelmointikieliä. Useimpien kielten rajatun käyttötarkoituksen vuoksi minkään yhden ohjelmointikielen osaaminen ei kuitenkaan riitä ohjelmointityössä, vaan perinteinen ohjelmointityö edellyttää useimmiten useiden eri kielten hallintaa.

2.2 Kooditon kehitys

Harsun (2012) mukaan ohjelmointikielten neljäntenä sukupolvena pidetään sovelluskehittimiä eli ohjelmia, joilla pystyy generoimaan tietyn sovellusalueen ohjelmia. Niissä kieltä käytetään usein osana graafista käyttöliittymää. Harsun (2012) mukaan sovelluskehittimet ovat yleensä jonkin valmistajan kaupallisia tuotteita, minkä takia ne eivät ole laajassa käytössä.

Vastaava idea on kuitenkin nyt teknologioiden kehityksen myötä nousussa uusilla nimillä. Ns. *vähäkoodisen* (*low-code*) tai *koodittoman* (*no-code*) kehityksen alustat mahdollistavat ohjelmien luomisen jopa kirjoittamatta riviäkään koodia (Caballar 2020, Sahay ym. 2020). Koodittomassa kehityksessä huipentuu edellä kuvattu pitkään jatkunut pyrkimys kehittää ohjelmointikieliä aina vain intuitiivisemmiksi (Woo 2020). Vähäkoodinen sovelluskehitysalusta tukee nopeaa sovelluskehitystä ja sovellusten yksivaiheista käyttöönottoa, suorittamista ja hallintaa käyttäen deklaratiivisia, korkean tason ohjelmointijärjestelmiä, kuten metadata- ja mallipohjaisia ohjelmointikieliä (Vincent ym. 2019,

Silva ym. 2020). Jotkut vähäkoodiset sovelluskehitysalustat (esim. Mendix[4]) ovat alkaneet hyödyntää tekoälyä ennakoidakseen paremmin ohjelmointityön tarpeita (Woo 2020). Tällaisen dataan perustuvan älykkyyden sisällyttäminen sovelluskehitysalustaan on seuraava askel kohti automaattista koodin tuottamista ja ohjelmia, jotka ohjelmoivat itse itsensä (Woo 2020).

Markkinatutkimusyritys Forresterin raportin (Richardson & Rymer 2014) mukaan sovellusten ja sovelluskehityksen lisääntynyt vuorovaikutus asiakkaiden kanssa saa monet yritykset siirtymään vähäkoodisiin alustoihin. Toisen tutkimusyrityksen Gartnerin raportissa (Vincent ym. 2019) ennustetaan, että vuoteen 2024 mennessä vähäkoodinen sovelluskehitys kattaa jo yli 65 % sovelluskehityksestä.

Ohjelmistoalan yrityksillä on vaikeuksia löytää riittävästi osaavaa työvoimaa (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019), joten kooditon tai vähäkoodinen kehitys vastaa tähän tarpeeseen (Caballar 2020, Sahay ym. 2020). Koodittomalla kehityksellä ohjelmointitaidottoman ihmisen on mahdollista luoda samanlaisia ohjelmia kuin ohjelmoinnin ammattilaisen (Caballar 2020, Sahay ym. 2020). Koodittoman kehityksen työkalun oppiminen on huomattavasti helpompaa kuin ohjelmoinnin opiskelu (Caballar 2020). Nostamalla abstraktiotasoa, jolla ohjelmistoja kehitetään, vähäkoodiset kehitysalustat automatisoivat rutiininomaiset kehitystehtävät, mikä on tehokas apu maailmanlaajuiseen ammattimaisten ohjelmistokehittäjien puutteeseen (Silva ym. 2020). Vähäkoodisella sovelluskehityksellä on potentiaalia muuttaa dramaattisesti ohjelmistojen kehitystapaa, jolloin ainakin tietyillä osa-alueilla on mahdollista toisaalta kehittää laadukkaita ohjelmistoja kouluttamattomien kehittäjien voimin ja toisaalta nopeuttaa merkittävästi kokeneiden kehittäjien kehitysprosessia (Silva ym. 2020). Kokeneimmille ohjelmoijille jää näin aikaa keskittyä vaikeimpiin ohjelmointiongelmiin (Woo 2020).

Ohjelmistoprosessimalleilla on erittäin tärkeä rooli ohjelmistokehityksessä, joten ne muodostavat ohjelmistotuotteen ytimen (Kaur & Sengupta 2011). Vähäkoodisella kehitysalustalla kehitysprosessi etenee seuraavasti (Sahay ym. 2020):

1. Tietomallinnus
2. Käyttöliittymän määrittely

---

[4] https://www.mendix.com/

3. Toimintalogiikan määrittely

4. Ulkoisten palvelujen integrointi

5. Sovelluksen käyttöönotto

Richardsonin ja Rymerin (2014) mukaan vähäkoodisen alustan käytöllä on monia hyötyjä ohjelmistoprosessissa:

- nopeuttaa ohjelmistotoimituksia
- helpottaa asiakkaiden osallistumista projektiin
- auttaa menestymään vähäisemmillä ohjelmointitaidoilla
- uusien ideoiden kokeileminen on helppoa ja nopeaa, ja toimivat ideat saa siirrettyä helposti tuotantoon
- toteutetun sovelluksen saa muunnettua mobiilikäyttöön napin painalluksella ilman mobiiliosaamista

Koska nykyään on olemassa monia erilaisia vähäkoodisia sovelluskehitysalustoja, voi olla haasteellista valita niistä kuhunkin tarpeeseen parhaiten sopiva vaihtoehto (Sahay ym. 2020). Sovelluksen kehitysmahdollisuudet riippuvat käytettävän vähäkoodisen sovelluskehitysalustan toiminnoista, ja alkuperäisiä vaatimuksia voidaan joutua mukauttamaan sovelluskehitysalustan tarjoamien vaihtoehtojen mukaisiksi (Sahay ym. 2020).

Vähäkoodiset kehitysalustat ovat usein pilvessä toimivia ohjelmistoalustoja, joiden avulla voi osaamistaustasta riippumatta kehittää täysimittaisia, tuotantovalmiita sovelluksia (Sahay ym. 2020). Sovellusten kehityksen ytimessä ovat mallipohjaiset suunnitteluperiaatteet, ja kehityksessä hyödynnetään pilvi-infrastruktuureja, automaattista koodinluontia ja deklaratiivisia ja graafisia korkean tason abstraktioita (Sahay ym. 2020). Sovelluskehityksen arkkitehtuuri on siis kerroksista kuten Kuva 1 osoittaa.

Kuva 1. *Vähäkoodisten sovelluskehitysalustojen kerrosteinen arkkitehtuuri (Sahay ym. 2020, s. 172)*

2.3 Ohjelmointityön vaatimukset

Elämme niin sanotun neljännen teollisen vallankumouksen aikaa (Industry 4.0), jossa vahvan tietoteknologian ja tuotannon automatisaation kasvu täydentyy kyberfyysisillä järjestelmillä (ASME 2015), esimerkkinä älyohjatut liikennevälineet. Teollisuuden nopea kehittyminen ja eri tarkoituksiin soveltuvien ohjelmointikielten runsaus vaikeuttavat kasvavaan ohjelmointialan osaajien tarpeeseen vastaamista koulutuksen keinoin. Ohjelmointitaitojen koulutusta onkin esitetty lisättäväksi laajasti teollisuuden eri ammattiryhmien koulutusohjelmiin (esim. ASME 2015) ja jo esimerkiksi suomalaisen perusopetuksen opetussuunnitelman perusteissa (2014) koodaustaidot mainitaan osana laaja-alaisia osaamistavoitteita sekä ala- että yläkoulun puolella. Vaikka maailma ympärillämme on vahvasti digitalisoitunut ja teknologian käyttö tullut osaksi jokapäiväistä elämää ja siis myös opiskelua, tuoreet tutkimukset (esim. Kaarakainen & Saikkonen 2019) kuitenkin osoittavat, että jopa tekniikan alojen opiskelijoiden digitaalisissa valmiuksissa on puutteita.

Tie kokeneeksi ohjelmointialan ammattilaiseksi on siis pitkä. Xie ym. (2019) esittävät, että opetusta ja aloittelevan ohjelmoijan taitojen kehittymistä voisi tehostaa ohjelmointitaitojen strukturoidulla opetuksella. Heidän mukaansa neljä keskeistä toisiaan tukevaa ohjelmointitaitoa ovat:

1. koodin syntaksin lukeminen ja ymmärtäminen

- Ohjelmoija pystyy koodia seuraamalla päättelemään, miten ohjelma toimii.

2. koodin kirjoittaminen syntaktisesti oikein

- Ohjelmoija pystyy kirjoittamaan virheettömiä käskyjä, jotka tuottavat odotetun lopputuloksen.

3. ohjelmointimallien tunnistaminen

- Ohjelmoija ymmärtää ohjelmoinnissa toistuvat rakenteet ja niiden tarkoitukset.

4. ohjelmointimallien käyttäminen

- Ohjelmoija pystyy ratkaisemaan ongelmia valitsemalla ja toteuttamalla sopivia ohjelmointimalleja.

Näiden perustaitojen lisäksi erinomainen ohjelmoija erottuu aloittelijasta taidoillaan virheiden löytämisessä ja korjaamisessa sekä ongelmanratkaisussa ja kyvyllään keksiä tarvittaessa uusia ohjelmointimalleja (Xie ym. 2019).

Ohjelmointityössä käytettävät käsitteelliset ja konkreettiset työkalut kuten ohjelmointityökalut ja -kielet, menetelmät ja prosessimallit vaikuttavat ohjelmointiprosessin tuloksiin ja käyttäjänsä ajattelutapaan (Wernick & Hall 2004). Ohjelmointityö edellyttää perehtymistä sekä käytettävään työkaluun että asiakkaan vaatimuksiin. Ohjelmistokehitysyritykset etsivät kilpailukykyetua kustannusten, laadun, joustavuuden, tuottavuuden ja riskien vähentämisen perusteella (Holtkamp 2015).

Holtkamp (2015) osoittaa, että ohjelmistonkehittäjältä vaadittaviin kompetensseihin ei kuitenkaan ole helppoa soveltaa yleispäteviä osaamismalleja, koska kehittämisympäristöllä on hyvin merkittävä vaikutus osaamisvaatimuksiin. Ohjelmointi on vahvasti kontekstisidonnaista eli ohjelmistokehittäjien täytyy muokata työskentelyään kulloisenkin työympäristön mukaiseksi. Ohjelmistotuotteiden laatu riippuu suuresti kehitystiimin jäsenten pätevyydestä (= tieto, taidot ja kyvyt) (Holtkamp 2015).

Vaikka ohjelmistokielten kehitys ja etenkin tekoälyn ja koodittoman ohjelmoinnin synteesi tähtää siihen, että ohjelmointityö olisi mahdollista yhä useammalle, ohjelmointityön erityisosaamista tullaan jatkossakin tarvitsemaan ratkaisemaan vaikeimpia pulmia (Woo

2020). Silva ym. (2020) selvittivät millaisia ohjelmointiongelmia vähäkoodisen sovelluskehitysalustan kanssa voi tulla. He määrittelevät kolme erityyppistä ajatusvirhettä ohjelmoijan käsityksessä siitä, miten ongelma pitäisi ohjelmoida:

- *Alihajottamisristiriidassa* (under-decomposition conflict) käyttäjä ajattelee toiminnon hoituvan yhdellä askeleella, kun se todellisuudessa vaatii useampia vaiheita. Tämä saattaa tarkoittaa ongelman monimutkaisuuden aliarviointia.

- *Ylihajottamisristiriidassa* (over-decomposition conflict) tilanne on päinvastoin, eli käyttäjä uskoo toiminnon vaativan useita vaiheita, kun se todellisuudessa hoituisi yhdellä askeleella. Tämä saattaa johtaa tarpeettoman monimutkaiseen ratkaisuun.

- *Vastaamattomuusristiriita* (no correspondence conflict) tarkoittaa sitä, että käyttäjän ajattelema vaihe ei ole järjestelmässä mahdollinen tai päinvastoin käyttäjä ei ole ottanut huomioon jotain järjestelmässä tarvittavaa vaihetta.

Ongelmanratkaisijan on ensin jäsenneltävä ongelma selkeäksi esitykseksi nykytilanteesta ja tavoitetilasta. Lisäksi merkityksettömät toimet on tunnistettava hedelmällisistä toimista näennäisesti rajoittamattomasta joukosta mahdollisia vaihtoehtoja. Luovien osaamistyön ammattilaisten, kuten tuotekehittäjien, on rutiininomaisesti käsiteltävä tällaisia tilanteita. (Björklund 2013)

Björklund (2013) vertaili tutkimuksessaan kokeneiden tuotekehittäjien ja tuotekehityksen opiskelijoiden ongelmanratkaisukykyjä tuotekehityksen ongelmiin liittyen. Hän havaitsi kokeneiden tuotekehittäjien mielikuvien olevan laajempia, syvempiä ja yksityiskohtaisempia ja sisältävän enemmän yhteenliitäntöjä ja suuntautuvan enemmän toimintaan kuin opiskelijoilla. Kokeneet tuotekehittäjät näkivät ongelmat siinä mielessä opiskelijoita haastavampina, että he ilmoittivat tarvitsevansa enemmän tietoa ratkaistakseen esitetyt ongelmat. Kokeneet tuotekehittäjät olivat tuottavampia, tuottaen paljon suuremman määrän koodia kuin opiskelijat. (Björklund 2013)

3. TOIMIALAKOHTAINEN MALLINNUS

Toimialalla tarkoitetaan tässä tutkimuksessa sellaista mahdollisimman tarkkaan rajattua joukkoa sovelluskehityksen tapauksista, että tuotettavilla sovelluksilla on paljon yhteneväisyyksiä, joita voidaan hyödyntää sovelluskehityksen automatisoinnissa. Tässä ei siis tarkoiteta esimerkiksi Tilastokeskuksen toimialaluokitusta (2008), joka puolestaan jakaa toimialat niiden sisällön ja tarkoituksen mukaan (esim. rakentaminen, koulutus, terveys- ja sosiaalipalvelut).

3.1 Toimialakohtaisuus sovelluskehityksen lähtökohtana

Ohjelmistokehityksen tehokkuutta voi parantaa lisäämällä abstraktiota. Nykyisten ohjelmointikielten abstraktiota on kuitenkaan vaikea nostaa enää merkittävästi. Korkeampi abstraktiotaso saadaan *toimialakohtaisella mallinnuksella* (domain-specific modeling, DSM), jossa ratkaisu määritellään suoraan ongelma-alan käsitteitä käyttäen ja lopullinen tuote generoidaan tästä määrittelystä (Kelly & Tolvanen 2008). Toimialakohtaisen ratkaisun arkkitehtuuri määrittyy toimialan tietämyksen ja ominaisuuksien pohjalta (Krach 2019). Kun ratkaisun määrittelyssä käytetään toimialan käsitteitä, se helpottaa ohjelmointityön ammattilaisten ja valmista sovellusta myöhemmin hyödyntävien asiakkaiden välistä vuorovaikutusta (Fowler 2010, Kelly & Tolvanen 2008).

Toimialakohtainen mallinnusratkaisu tehdään tarkasti rajatulle toimialalle, ja se rajaa pois kaikki muut sovellusalueet, eli sitä voi käyttää vain yhdenlaisiin sovelluksiin. Koska kielen ja generaattoreiden täytyy sopia vain yhdelle yritykselle tietynlaisiin tapauksiin, abstraktiotaso saadaan korkeammaksi. Mitä kapeampi sovellusala on, sitä paremmin sen saa automatisoitua. (Kelly & Tolvanen 2008)

Malli on aina kuvaus jostakin; se on systeemin abstraktio (France & Rumpe 2007, Kelly & Tolvanen 2008, Kleppe ym. 2003). Yleensä yksittäinen malli kuvaa jotakin systeemin tiettyä näkymää tai ominaisuutta, ja koko systeemin määrittelemiseksi tarvitaan useita malleja ja mallinnuskäsitteitä, jotka määrittelevät sovellusta eri näkökulmista (Kelly & Tolvanen 2008, Kleppe ym. 2003). Joskus mallia pidetään vain yksinkertaistettuna kuvauksena systeemistä ja ohjelmointikielellä ohjelmoitua koodia parempana kuvauksena todellisuudesta (Kelly & Tolvanen 2008). Se saattaa pitää paikkansa, jos kyseessä on mallinnuskieli, joka ei sisällä riittävästi informaatiota toimivan systeemin määrittelyyn, mutta toimialakohtaisten mallien kohdalla se ei pidä paikkaansa (Kelly & Tolvanen 2008).

Ohjelmistotuotannossa malleja käytetään usein ideoiden kommunikoinnin apuna ratkaisujen luonnosteluun, sovellusten suunnitteluun, toiminnallisuuksien määrittelyyn ja dokumentointiin (Tolvanen & Kelly 2016). Mallien käyttö ja ohjelmointi pidetään kuitenkin usein täysin erillään, jolloin se aiheuttaa vain lisää työtä pelkkään ohjelmointiin verrattuna (Kelly & Tolvanen 2008). Dzidek ym. havaitsivat tutkimuksessaan (2008), että ohjelmointityön suorittaminen UML-mallinnuksen kanssa oli hieman hitaampaa kuin pelkkä ohjelmointi. *Mallipohjainen kehitys* (model-driven development) tarkoittaa sen sijaan mallien käyttämistä pääasiallisena lähteenä sovellusten luomisessa ohjelmakoodin sijaan (Tolvanen & Kelly 2016). Silloin koodi voidaan generoida suoraan mallista tietyn tyyppisiin sovelluksiin erikoistuneella sovellusgeneraattorilla (Kelly & Tolvanen 2008). Mallipohjaista kehitystä käytetään paljon tuotekehityksessä ja sulautetuissa järjestelmissä, mutta se ei ole yhtä yleistä projektiperustaisessa kehityksessä kuten konsultoinnissa tai alihankintatyössä (Tolvanen & Kelly 2016). Mallipohjaisessa kehityksessä voidaan käyttää myös yleiskäyttöisiä mallinnuskieliä (esim. UML), mutta toimialakohtaiset mallinnuskielet mahdollistavat suuremman hyödyn (Tolvanen & Kelly 2016).

Vaikka yksittäistä toimialakohtaista mallinnuskieltä määritelmän mukaisesti voidaankin soveltaa vain rajoitettuun toimialueeseen, lähestymistapa sinänsä sopii moniin erilaisiin toimialueisiin (Tolvanen & Kelly 2016). Toimialakohtaista lähestymistapaa kannattaakin Kellyn ja Tolvasen (2008) mukaan käyttää aina, kun se on mahdollista, koska se tuottaa parempia tuloksia kuin yleiskäyttöiset lähestymistavat.

3.2 Toimialakohtaisen mallinnuksen hyödyt

Toimialakohtaisesta mallinnuksesta saadaan samoja hyötyjä kuin abstraktiotason nostosta yleensäkin. Tuottavuus ja laatu paranevat, monimutkaisuus saadaan piilotettua ja asiantuntemusta voidaan hyödyntää laajemmin (Fowler 2010, Kelly & Tolvanen 2008). Kehittäjä pystyy toimimaan monimutkaisempien järjestelmien parissa vähemmällä vaivalla (Kleppe ym. 2003).

Yrityksen ohjelmistokehittäjien taidoissa ja kokemuksessa on usein suuria eroja, erityisesti kun osaavista ohjelmoijista on pulaa (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019). Toimialakohtainen mallinnus on tehokas tapa hyödyntää asiantuntevien kehittäjien osaamista koko tiimin tasolla: kun asiantunteva kehittäjä määrittelee toimialan säännöt kieleksi, hän saa kehitysohjeet ja parhaat käytännöt kaikkien kehittäjien käyttöön (Kelly & Tolvanen 2008). Sama pätee generaattorin luomiseen: kun generaattorin

on luonut asiantunteva kehittäjä, sen luoma koodi on asiantuntevan kehittäjän luoman koodin tasoista (Kelly & Tolvanen 2008).

Toimialakohtaisen mallinnuksen tuottavuushyödystä on vaikea löytää tarkkaa tutkimustietoa, koska yrityksillä ei yleensä ole varaa järjestää tutkimusasetelmia ja niiden käyttämät ohjelmointimenetelmät kuuluvat usein yrityssalaisuuden piiriin (Kelly & Tolvanen 2008, Vestman 2019). Tapaustutkimuksen raportoinnissa on myös ongelmansa: todella hyvät tulokset kuulostavat helposti mainostukselta ja yksityiskohtien kertominen paljastamatta yrityssalaisuuksia on hankalaa (Kelly & Tolvanen 2008). Eräs toimialakohtaisen mallinnusratkaisun toteuttamiseen käytettävä työkalu on MetaEdit+[5], jonka on eri tutkimuksissa havaittu lisäävän tuottavuutta 400–1000 % (Kärnä ym. 2009, MetaCase 2000, Puolitaival ym. 2011, Safa 2007). Usein tuottavuushyöty on niin ilmeinen (tehtävä, jonka suorittaminen perinteisin menetelmin vaati päiviä, onnistuu toimialakohtaisella mallinnuksella vain minuuteissa), että yritysten kiinnostus laajemman tai täsmällisemmän vertailun tekemiseen on pientä (Kelly & Tolvanen 2008). Lisäksi tutkimuksissa usein mitataan toimialakohtaisella mallinnuksella kuluvaa kehitysaikaa siinä vaiheessa, kun työkalu on kehittäjille uusi. Vaikka siinäkin tapauksessa tuottavuushyöty on selkeä, on odotettavissa, että se olisi vielä parempi, kun käyttäjät ovat perehtyneet mallinnukseen paremmin (Kelly & Tolvanen 2008).

Nopeat tuotantoajat toimialakohtaisella mallinnuksella aiheuttavat Kellyn & Tolvasen (2008) mukaan monia hyötyjä:

- lyhyempi aika markkinoille: tuote ehtii markkinoille ennen kilpailijoita
- nopeampi asiakaspalautekierto
- matalammat kehityskustannukset
- muutokset mahdollisia myöhäisessäkin vaiheessa
- mahdollisuus palvella sellaisia asiakkaita, joiden palveleminen perinteisesti koodaten olisi kustannusnäkökulmasta yksinkertaisesti liian epätehokasta
- pienempi tarve alihankinnalle/ulkoistamiselle

[5] https://www.metacase.com/products.html

Ylläpitovaiheeseen käytetään merkittävä osa sovelluskehityksen vaivasta, kustannuksista ja ajasta. Yksi osa ylläpitovaihetta ovat ohjelmointivirheiden korjaukset, mutta kun koodi on automaattisesti generoitua, monet virheet vältetään jo sillä. Jos malleissa on virheitä tai niihin halutaan tehdä muutoksia uusien tai muuttuneiden vaatimusten vuoksi, saadaan generoinnilla sama tuottavuushyöty kuin alkuperäisessä sovelluksen luontivaiheessa. Joskus ympäristön tai alustan muutokset vaativat muutoksia generaattoriin, jolloin mallia muuttamatta saadaan luotua sovellus uudelle alustalle. (Kelly & Tolvanen 2008)

Laatu on tuottavuutta hankalammin mitattava kriteeri. Yksi oleellinen ohjelmiston laatua mittaava kriteeri on sen luotettavuus ja toimintavarmuus kaikissa tilanteissa, mihin liittyy vähäinen virheiden määrä (Bellairs 2019, Xie ym. 2019). Virheet vaikuttavat tietysti myös tuottavuuteen – kehittäjien aikaa menee virheiden etsimiseen ja korjaamiseen (Fowler 2010). Yrityksillä ei valitettavasti ole yleensä resursseja tehdä tutkimusta siitä, paljonko virheet vähenevät toimialakohtaiseen mallinnukseen siirryttäessä (Kelly & Tolvanen 2008). On kuitenkin useita syitä, miksi toimialakohtainen mallinnus vähentää virheitä. Toimialakohtaiset mallinnuskielet voivat sisältää toimialakohtaisia oikeellisuussääntöjä, jotka tekevät ei-toivottujen määritysten tekemisen mahdottomaksi (Fowler 2010, Kelly & Tolvanen 2008). Koska malli on niin lähellä asiakasvaatimuksia ja asiakkaallekin ymmärrettävä, mallinnus voidaan tehdä jopa asiakkaan kanssa yhteistyössä, jolloin väärinymmärryksiä tai virheitä mallinnuksessa ei tule niin helposti (Fowler 2010, Kelly & Tolvanen 2008). Kun sovelluksen muuntaminen mallista koodiksi tehdään automaattisesti generoiden, vältetään tässä vaiheessa syntyvät virheet. Virheitä on tietenkin olemassa myös toimialakohtaisessa mallinnuksessa. Ne ovat kuitenkin vähemmän satunnaisia kuin käsin koodatussa sovelluksessa (Kelly & Tolvanen 2008). Jos virhe on mallissa, mallin korjaaminen ja uudelleen generointi riittää. Jos virhe on generaattorissa, sen korjaaminen vaikuttaa kaikkiin generoitaviin sovelluksiin – kerran korjattu virhe ei enää toistu (Kelly & Tolvanen 2008).

Kellyn & Tolvasen (2008) mukaan toimialakohtaisella mallinnuksella on monenlaisia lopputuloksen laatua parahtavia vaikutuksia:

- Sovellusarkkitehtuuri: Arkkitehtuuriset säännöt tulevat käyttöön automaattisesti suunnitteluvaiheessa. Yksittäisen kehittäjän ei ole mahdollista poiketa säännöistä.

- Koodaus: Kirjoitusvirheistä tai puutteellisista viittauksista tai alustuksista johtuvia tyypillisiä virheitä ei tule, kun koodi luodaan automaattisesti. Koodi on myös automaattisesti tyylillisesti samanlaista.

- Uudelleenkäyttö: Käsin kirjoitetussa koodissa on yleistä, että kehittäjät kirjoittavat uusia toteutuksia samaan toiminnallisuuteen, joka on jo toteutettu toisaalla. Toimialakohtainen mallinnusjärjestelmä voi käyttää saatavilla olevia malleja tai viitata olemassa olevaan koodiin automaattisesti.

- Toimialakohtaiset kelpoisuusvaatimukset: Mallinnuskieli tai generaattori voidaan suunnitella varmistamaan, että lopputulos sisältää halutut ominaisuudet.

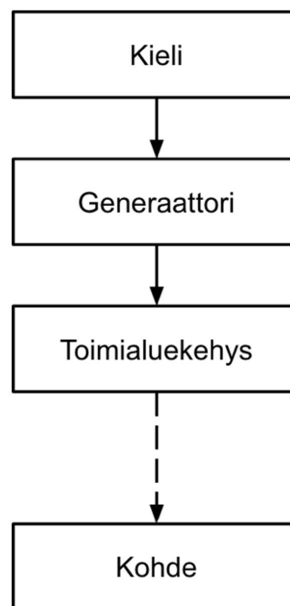
- Muutosten ja uudelleenmuotoilun vaikutukset: Muutokset määrittelyihin on helpompi tehdä toimialakohtaisella tasolla.

- Dokumentaatio: Täsmällistä ja ajantasaista dokumentaatiota pidetään laadukkaan tuotantoprosessin merkkinä. Dokumentaatio voidaan luoda automaattisesti samalla tavalla kuin koodikin.

Muutosten tekemisen helppoutta tutkivat esimerkiksi Sadrieh ja Bahri (2014) ja osoittivat, että toimialakohtaisella mallinnuksella toteutettuun sovellukseen on helpompaa ja yksinkertaisempaa tehdä myöhemmin tarvittavia muutoksia verrattuna perinteisesti ohjelmoituihin sovelluksiin. Sovelluksen muutosten tekemiseen vaadittava aika oli perinteisellä tavalla ohjelmoiden noin 2,5-kertainen ja lisättyjen tai muutettujen koodirivien määrä jopa yli nelinkertainen toimialakohtaiseen mallinnukseen verrattuna.

3.3 Toimialakohtaisen mallinnuksen arkkitehtuuri

Ohjelmistokehitysprosessi vaatii aina ongelman analysointia, ratkaisun keksimistä ja ratkaisun esittämistä ohjelmointikielellä (France & Rumpe 2007). Ohjelmoinnilla pyritään siis rakentamaan silta *ongelma-alueen* ja *ratkaisualueen* välille (France & Rumpe 2007, Kelly & Tolvanen 2008). Toimialakohtaisessa mallinnusprosessissa ongelma-aluetta kuvataan korkeatasoisella mallilla, joka muunnetaan toimivaksi systeemiksi automaattisesti generoiden (France & Rumpe 2007, Kelly & Tolvanen 2008). Toimialakohtaisen mallinnuksen kolmitasoista arkkitehtuuria kuvataan Kuvassa 2.



Kuva 2. *Toimialakohtaisen mallinnuksen arkkitehtuuri (Kelly & Tolvanen 2008, s. 64)*

Kelly ja Tolvanen (2008) määrittelevät siis toimialakohtaisen mallinnuksen arkkitehtuuriin kolme eri osa-aluetta:

1. Toimialakohtaisella mallinnuskielellä kuvataan ongelma-aluetta. Malleissa käytettävät elementit vastaavat toimialan maailmaa koodin maailman sijaan. Mallinnuskielet seuraavat toimialan abstraktioita ja merkityksiä, jolloin mallintajat pystyvät työskentelemään suoraan toimialan käsitteiden kanssa.

2. Generaattori kuvaa mallin ratkaisualueelle. Generaattori määrittelee, miten informaatio poimitaan mallista ja muunnetaan koodiksi. Yksinkertaisimmillaan tietty mallinnussymboli vastaa aina tiettyä koodia, jonka parametreina on symboliin liitetyt arvot. Generaattori voi myös luoda erilaista koodia riippuen symbolin arvoista ja suhteista muiden symbolien kanssa. Tavoitteena on, että generaattorin luomaa koodia ei tarvitse käsin muokata (Fowler 2010), joten luotu koodi on vain sivutuote, joka käännetään kääntäjällä valmiiksi ohjelmaksi.

3. *Toimialuekehys* (domain framework) on rajapinta generoidun koodin ja sovellusalustan välillä. Joissain tapauksissa erillistä kehyskoodia ei tarvita, vaan generoitu koodi toimii suoraan sovellusalustalla. Kehyskoodia kannattaa usein kuitenkin tehdä generoidun koodin yksinkertaistamiseksi.

Kohdeympäristö (target environment) on alusta, jolla sovellus pyörii. Kohdeympäristö koostuu erilaisista infrastruktuurikerroksista (Kuva 3). Ainakin joitain näistä kerroksista tarvitaan riippumatta siitä, onko sovellus ohjelmoitu käsin vai generoitu automaattisesti. Kaikki nämä kerrokset ovat olemassa parantaakseen kehittäjän tuottavuutta. Ne lisäävät abstraktiota toteutuspuolella, kun taas toimialakohtainen mallinnuskieli lisää abstraktiota ongelma-alan puolella. Äärimmäisillään kohdeympäristö voi olla pelkkä laitteisto, mutta tyypillisemmin siihen kuuluu komponenttikehyksiä, kirjastoja tai itsenäisiä ohjelmia. Toimialakohtaista mallinnusta käytettäessä sovelluskehittäjän ei tarvitse olla tietoinen kohdeympäristön tasoista, vaan niiden käyttö tulee automaattisesti. Generaattorin kehittäjän täytyy tietenkin tuntea kohdeympäristö tarkasti. (Kelly & Tolvanen 2008)

Kuva 3. *Kohdeympäristön kerrokset (Kelly & Tolvanen 2008, s. 86)*

Useimmissa tapauksissa optimaalinen tapa parantaa kohdeympäristön käyttöä on rakentaa asianmukainen toimialuekehys sen päälle. Tämä toimialuekehys voi sisältää komponentteja tai muita ohjelman osia, joita tarvitaan usein valitun toimialueen koodissa. Ne kehitetään yleensä manuaalisesti, kuten kohdeympäristön kirjastot ja kehykset. Toimialuekehyksen koodilla on erilaisia tarkoituksia (Kelly & Tolvanen 2008):

- Poistaa toistoa generoidusta koodista. Sovelluksissa on yleensä valitulle toimialueelle tyypillisiä rakenteita. Sen sijaan, että ne sisällytettäisiin generaattoriin, ne voidaan lisätä toimialuekehykseen, josta generoitu koodi voi niitä kutsua. Se pitää generaattorin yksinkertaisempana.

- Tarjoaa rajapinnan generaattorille. Toimialuekehys määrittelee oletetun muodon koodigeneroinnin tulokselle. Se ei ole määritelty konkreettisena koodina, vaan mallineena, jonka generaattori sitten tuottaa.

- Integroi olemassa olevan koodin kanssa. Toimialuekehystä voidaan käyttää kutsumaan kirjaston palveluja ja rajapintoja sen sijaan, että niitä kutsuttaisiin suoraan generoidussa koodissa.

- Piilottaa kohdeympäristön ja toteutusalustan. Toimialuekehystä voidaan käyttää tukemaan eri toteutusalustoja. Mallit ja generoitu koodi voivat siten olla samoja ja toimialuekehyksen valinta määrittää toteutusalustan.

3.4 Toimialakohtaisen mallinnuksen toteuttaminen

Toimialakohtaisen mallinnuksen hyötyjä ei saa ilmaiseksi, koska kielen abstraktiot ja työkalut kehityksen automatisoimiseksi on ensin kehitettävä ja myöhemmin ylläpidettävä (Tolvanen & Kelly 2016). Siihen tarvitaan osaavaa kehittäjää, ja toimialakohtaisen mallinnuksen laatua parantava vaikutus johtuukin oikeastaan siitä, että generaattorin kautta suurempi osa valmiista koodista on käytännössä kokeneemman ohjelmoijan eli generaattorin luojan kirjoittamaa (Kelly & Tolvanen 2008).

Valmiita DSM-ratkaisuja ei yleensä ole olemassa, koska täsmälleen sama toimialakohtainen ratkaisu ei yleensä sovi eri yrityksille (Kelly & Tolvanen 2008). Erot ongelma-alassa, tausta-arkkitehtuureissa, kohdeympäristöissä ja ohjelmointikielissä aiheuttavat sen, että yrityksen on rakennettava oma DSM-ratkaisunsa (Kelly & Tolvanen 2008). Kapea keskittymisalue saadaan yleensä parhaiten määriteltyä yksittäisen yrityksen sisällä, jolloin toimialakohtaisen mallinnuksen hyödyt tulevat parhaiten esiin (Kelly & Tolvanen 2008). Yrityksen sisälläkin yksittäinen toimialakohtainen mallinnusratkaisu kattaa yleensä vain pienen osa-alueen (Kelly & Tolvanen 2008) ja samassa yrityksessäkin saatetaan siis tarvita useampia eri toimialakohtaisia mallinnuskieliä (Tolvanen & Kelly 2016). Jos toimiala pysyy samana, hyvin luotua toimialakohtaista mallinnuskieltä ei tarvitse enää muokata (Tolvanen & Kelly 2016). Käytännössä toimialueet muuttuvat kuitenkin aina, joten on tärkeää, että toimialakohtaisen mallinnuskielen päivittäminen onnistuu helposti (Tolvanen & Kelly 2016).

Toimialakohtaisen mallinnuskielen luomista pidetään yleensä hankalana ja kalliina. Uskotaan myös, että toimialakohtaisen kielen luominen on mahdollista vain, kun toimiala ei

muutu, ja että mallipohjainen kehitys ei skaalaudu. Hyvällä nimenomaan toimialakohtaisen mallinnuskielen luomiseen tarkoitetulla työkalulla nuo käsitykset eivät kuitenkaan välttämättä pidä paikkaansa. (Tolvanen & Kelly 2016)

Jotta mallipohjainen kehitys olisi skaalautuvaa, mallinnuksessa käytettävän työkalun täytyy pystyä tehokkaasti käsittelemään isojakin malleja ja sen on tuettava useiden samanaikaisten mallintajien työskentelyä samoilla malleilla. Jos toimialue pysyy samana, hyvin luotua toimialakohtaista mallinnuskieltä ei tarvitse enää muokata. Käytännössä toimialueet muuttuvat kuitenkin aina, joten on tärkeää, että toimialakohtaisen mallinnuskielen päivittäminen onnistuu helposti. (Tolvanen & Kelly 2016)

El Kouchen ym. (2012) tutkivat toimialakohtaisten kielten luomiseen tarkoitettujen sovellusten tehokkuutta ja havaitsivat, että kielen luominen MetaEdit+:lla oli selvästi nopeampaa kuin muilla sovelluksilla. MetaEdit+:lla toimialakohtaisen mallinnuskielen luomiseen käytettävä kieli GOPPRR on toimialakohtainen kieli itsekin, luotu varta vasten toimialakohtaisten mallinnuskielten luomiseen (Tolvanen & Kelly 2016). Näin toimialakohtaisen mallinnuskielen luominen voikin olla helppoa ja nopeaa. Kielen luomisen helppoudesta on suuri hyöty, koska silloin kieltä voivat olla luomassa samat henkilöt, jotka sitä myös käyttävät (Tolvanen & Kelly 2016). Se mahdollistaa reflektion kielen käytöstä kielen määritelmään, eli kieltä pystytään muokkaamaan sen perusteella, mitä sen käytössä havaitaan (Tolvanen & Kelly 2016). MetaEdit+:lla kielen kehittäminen ja käyttäminen onnistuu yhtäaikaisesti samassa työkalussa (Tolvanen & Kelly 2016).

4. TUTKIMUKSEN KUVAUS

Tutkimus on tapaustutkimuksena toteutettava empiirinen vertailututkimus, jossa esimerkkisovellus toteutetaan kahdella eri tavalla ja verrataan onnistuneen lopputuloksen saamiseksi käytettyä aikaa.

4.1 Tutkimuskysymykset

Tämän tutkimuksen tarkoituksena on vertailla perinteisen sovelluskehityksen ja toimialakohtaista mallinnusta ja automatisointia soveltavan sovelluskehityksen eroavuuksia. Keskeinen tutkimuskysymys on: Tekeekö toimialakohtainen mallinnus sovelluskehityksestä tehokkaampaa?

Kuten edellä on todettu, ohjelmointityö on ohjelmistokehityksen kriittinen resurssi (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019). Siksi sovelluskehityksen tehokkuus ilmaistaan tässä tutkimuksessa määritellyn tavoitteen mukaisen sovelluksen ohjelmointiin kuluvana työaikana. Kelly (2013) myös toteaa, että käytetty aika on hyvin mitattavissa oleva suure, joka toimii myös eri kielten välisissä vertailuissa. Työhön käytettävän ajan mittaaminen ottaa huomioon myös sen työn, joka ei näy lopullisessa koodissa (Kelly 2013).

Tutkimuksessa tarkastellaan myös perinteisen ohjelmoinnin ja vähäkoodisen, mallipohjaista sovellusgeneraattoria hyödyntävän sovelluskehitysprosessin eroja sekä sitä, missä sovelluskehitysprosessin vaiheissa mahdollisia ajankäytön ja tehokkuuden eroja erityisesti ilmenee.

4.2 Tutkimusmenetelmä

Tämä tutkimustyö voidaan määritellä soveltavaksi tekniseksi tutkimukseksi. Nunamaker, Chen ja Purdin (1991) määrittelevät, että perustutkimuksessa kehitetään ja testataan teorioita ja hypoteeseja tutkijan älyllisen mielenkiinnon pohjalta pikemminkin kuin käytännön syistä. Soveltava tutkimus puolestaan on tietämyksen soveltamista ongelmien ratkaisemiseksi; tämän tutkimuksen tapauksessa haluttiin selvittää, tehostaako toimialakohtainen mallinnus sovelluskehitystä. Tekninen tutkimus eroaa puhtaan tieteellisestä tutkimuksesta kokeidensa ja aiheidensa laajuudessa, ja insinöörin lähestymistavassa "jonkin saaminen toimimaan" on usein välttämätöntä. Niin tässäkin tapauksessa: tutki-

mus toteutettiin käytännön ohjelmointityön avulla. Insinöörin ja puhtaan tutkijan käyttämien menetelmien välillä ei Nunamakerin ym. mukaan kuitenkaan ole loogista eroa. Molemmilla puolilla tutkijat ovat kiinnostuneita vahvistamaan teoreettiset ennustuksensa. Ogundare (2017) erottelee tieteellistä ja teknistä paradigmaa noudattavat tutkimusprosessit ja toteaa, että prosessien tuottama tieto on luonteeltaan erilaista: tieteellisten teorioiden esittämä tieto perustuu kontrolloituihin empiirisiin prosesseihin, kun taas tekniset metodit johdetaan ensisijaisesti kokemuksesta.

Nunamaker ym. (1991) erottavat toisistaan myös arviointi- ja kehitystutkimuksen kahtena eri tutkimusmenetelmätyyppinä, jotka on suunnattu ongelmien ratkaisemiseen: arvioiva ja kehittävä. Kehitystyyppinen tutkimus pyrkii etsimään ja muodostamaan ohjeita ja malleja parempaan toimintatapaan. Kehittämistyössä käytetään systemaattisesti tieteellistä tietoa hyödyllisten materiaalien, laitteiden, järjestelmien tai menetelmien tuottamiseksi, mukaan lukien prototyyppien ja prosessien suunnittelu ja kehittäminen. Näin kehitetään mm. uusia ohjelmointiin soveltuvia toimintamalleja. Tämän tutkimuksen tapauksessa on paremminkin kyse arviointitutkimuksesta, jossa jo kehitettyä uutta toimintamallia ja sen etuja perinteisiin malleihin halutaan tutkimuksen kautta analysoida ja vertailla. Nunamaker ym. (1991) toteavat, että joissakin tutkimuksissa teknologiaa käsitellään usein liian yksinkertaistaen muuttujana, joka joko on läsnä tai ei ole. Esimerkiksi oletetaan, että kaikkien laskentataulukkojen tai tekstinkäsittelyohjelmien hyväksyttävyys käyttäjälle on sama. Tässä tutkimuksessa lähtökohtana on päinvastoin arvioida ja vertailla teknologioiden toimivuutta ja tehokkuutta. Tavoitteena on saada todisteita tukemaan tai kumoamaan toimialakohtaisen mallinnuksen kehitystyössä oletetut hypoteesit tämän uuden toimintamallin hyödyistä.

Ogundare (2017) pitää teknistä tieteen lähestymistapaa perinteisestä tieteellisestä lähestymistavasta eroavana. Hänen mukaansa teknisen tieteen mukainen (engineering paradigm) tutkimuksen toteutusprosessi eroaa tieteellisestä (scientific paradigm) monella tavalla (Taulukko 1).

Taulukko 1. Tieteellisen ja teknisen paradigman erot (Ogundare 2017, s. 169)

Tieteellinen paradigma	Tekninen paradigma
1. Määrittele ongelma	1. Määrittele ongelma
2. Tunnista muuttujat	2. Löydä faktat
3. Muotoile hypoteesi	3. Määritä parametrit
4. Määrittele metodologia	4. Määritä hyväksymiskriteerit
5. Suunnittele koe	5. Määritä järjestelmän rajat
6. Suorita koe	6. Suunnittele prototyyppi
7. Testaa hypoteesi	7. Rakenna prototyyppi
8. Analysoi tulokset	8. Testaa
9. Muodosta johtopäätökset	9. Verifioi
10. Esittele tulokset	10. Esittele tulokset

Myös Nunamaker ym. (1991) esittävät, että tietojärjestelmätutkimukseen soveltuu monimetodinen lähestymistapa, johon voi kuulua seuraavia elementtejä:

- Järjestelmien kehittäminen: prototyypit, tuotekehitys, teknologiansiirto
- Kokeilu: tietokonesimulaatiot, kenttäkokeet, laboratoriokokeet
- Havainnointi: tapaustutkimukset, kyselytutkimukset, kenttätutkimukset
- Teorian rakentaminen: käsitteelliset viitekehykset, matemaattiset mallit ja metodit

Näistä elementeistä tässä tutkimuksessa hyödynnettiin prototyyppien rakentamista ja kokeilua sekä systemaattista havainnointia. Taulukossa 2 verrataan tämän tutkimuksen prosessia Nunamakerin ym. (1991) esittämään järjestelmäkehitystutkimuksen prosessiin. Prototyypin rakentamisessa hyödynnettiin SoulCorella aiemmin tehtyä tuotekehitystä.

Taulukko 2. *Tutkimuksen viitekehys Nunamakerin ym. (1991) järjestelmäkehitystutkimuksen prosessin mukaan esitettynä*

Nunamakerin ym. järjestelmäkehitystutkimuksen prosessi	Tämän tutkimuksen prosessi
Rakenna käsitteellinen viitekehys	
• Aseta mielekäs tutkimuskysymys • Tutki järjestelmän toiminnallisuuksia ja vaatimuksia • Ymmärrä järjestelmän rakentamisen prosessit/menettelyt • Tutki asiaan liittyvää eri alojen tutkimusta uusien ideoiden ja lähestymistapojen löytämiseksi	• Tutkimuskysymyksen määrittäminen yhdessä kohdeyrityksen kanssa • Järjestelmään ja sen toiminnallisuuksiin ja toimintaperiaatteisiin tutustuminen yrityksessä • Asiaan liittyvään tutkimustietoon tutustuminen tutkimuksen viitekehykseksi
Kehitä järjestelmäarkkitehtuuri	
• Kehitä ainutlaatuinen arkkitehtuurisuunnitelma laajennettavuuteen, modulaarisuuteen jne. • Määrittele järjestelmän komponenttien toiminnot ja niiden keskinäiset suhteet	• Esimerkkisovelluksen valitseminen ja suunnitteleminen • Esimerkkisovelluksen vaatimusten listaaminen
Analysoi & suunnittele järjestelmä	
• Suunnittele tietokantakaavio ja -prosessit järjestelmän toimintojen suorittamiseksi • Suunnittele vaihtoehtoisia ratkaisuja ja valitse niistä yksi	• Tietorakennekaavion piirtäminen sovelluksen tarkemmaksi määrittelyksi • Vertailutoteutuksen työkalujen valitseminen
Rakenna (prototyyppi)järjestelmä	
• Opi käsitteistä, viitekehyksestä ja suunnittelusta järjestelmän rakentamisprosessin kautta • Oivalla järjestelmän ongelmista ja monimutkaisuudesta	• Esimerkkitoteutuksen ohjelmointi sekä perinteisesti että toimialakohtaista mallinnusta käyttäen • Esimerkkisovellusten tekemiseen käytetyn ajan mittaaminen • Toteutusvaiheessa eteen tulevien ongelmien kirjaaminen
Havainnoi ja arvioi järjestelmää	
• Havainnoi järjestelmän käyttöä tapaustutkimuksilla ja kenttätutkimuksilla • Arvioi järjestelmää laboratoriokokeilla ja kenttäkokeilla • Kehitä uusia teorioita/malleja havainnointiin ja kokeiluihin perustuen • Vahvista opitut kokemukset	• Valmiiden esimerkkisovellusten vertailu • Esimerkkisovellusten tekemiseen käytettyjen aikojen vertailu • Esimerkkitoteutusten tekemiseen liittyvien kokemusten analysointi • Johtopäätösten tekeminen

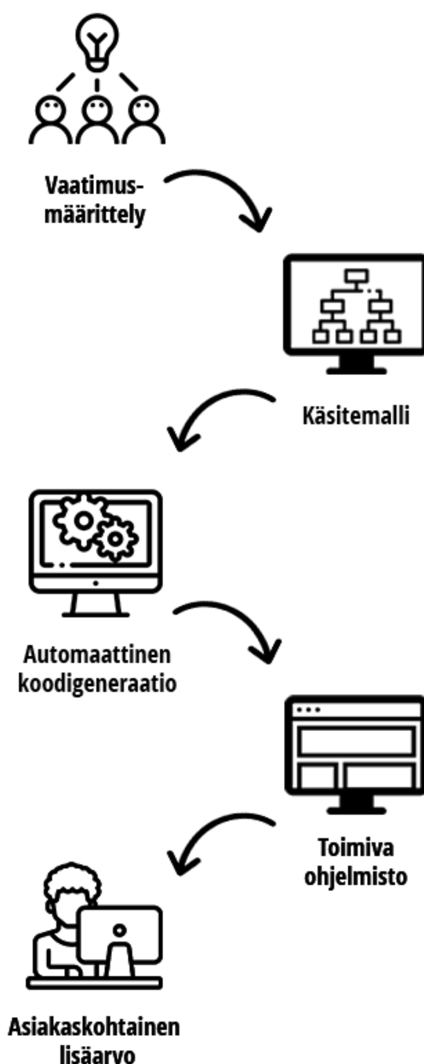
4.3 Automatisoitu sovellustuotanto SoulCorella

SoulCoren perustajaosakkaan ja toimitusjohtajan Tiina Vestmanin (2019) mukaan yrityksen toiminta-ajatuksena on alusta alkaen ollut liiketoimintajärjestelmien tehokas tuottaminen automatisoinnin kautta. Yritys ei ole keskittynyt minkään tietyn tilastokeskuksen (2008) toimialan sovelluksiin, sillä Vestmanin mukaan korkeammalta abstraktiotasolta katsottuna liiketoimintajärjestelmät ovat kaikki samanlaisia – ainoa ero on se, millaisia tietoja asiakas järjestelmässä haluaa käsitellä. Koska tuotettavat sovellukset siis näin määriteltynä ovat riittävän samankaltaisia, ne on käytännöllistä suorittaa tässä tutkimuksessa määriteltyä toimialakohtaisuutta hyödyntäen ja automaattisesti generoiden. Vestmanin mukaan generoinnin rajat eivät ole koskaan tulleet vastaan niin, että jotakin asiakkaan toivomaa asiaa ei olisi ollut mahdollista tehdä. Joissain tapauksissa kuitenkin esim. käyttöliittymää muokataan käsin ohjelmoimalla.

SoulCoren verkkosivujen (2019) mukaan yrityksen missiona on ”rakentaa laadukkaita ja moderneja sovelluksia tuotemaisella tehokkuudella ja nopeilla läpimenoajoilla verrattuna perinteiseen sovellustuotantoon”. Vestmanin (2019) mukaan se tarkoittaa sitä, että yritys kilpailee lähinnä valmistuotteita vastaan, koska vastaavien järjestelmien ohjelmoiminen käsin olisi liian aikaa vievää ja kallista. Valmistuotteisiin verrattuna automatisoidun toimialakohtaisen sovellustuotannon etuna on tietenkin se, että sovelluksesta voidaan tehdä juuri asiakkaan toiveita vastaava.

Sovellustuotannossa käytetään yrityksessä kehitettyä mallintamiseen ja automatisointiin perustuvaa sovellustuotantomallia. Sen automatisoidun sovellustuotannon teknologian ytimenä on SoulGen-generaattori, joka tekee visuaalisella mallinnuskielellä mallinnetun käsitemallin pohjalta sovelluksen ohjelmakoodin tietokantoineen ja käyttöliittymineen (Kuva 4). Tämän vaiheen tavoitteena on poistaa toisteista, matalan arvon käsityötä. Generoinnin tuloksena syntyy toimiva ohjelmisto, jota voidaan kehittää edelleen asiakaskohtaisten tarpeiden perusteella.

Sovellusgeneraattorilla tuotetaan asiakkaille sekä modulaarisia valmisratkaisuja että sovelluksia asiakkaan tarpeen mukaisiin laajoihin ratkaisukokonaisuuksiin. Ratkaisuja voi myöhemmin laajentaa uusilla toiminnallisuuksilla tai esimerkiksi tietoturvaan, suorituskykyyn tai käytettävyyteen liittyvillä päivityksillä.



Kuva 4. *Automatisoitu sovellustuotanto SoulCorella (SoulCore 2019)*

Malli vastaa Sahayn ym. (2020) kuvausta vähäkoodisen sovelluskehityksen toimintamallista. Sen mukaan vähäkoodisen kehitysalustan voi jakaa kolmeen tasoon, joista käyttäjälle näkyvä taso on sovellusmallinnin, graafinen ympäristö, jolla käyttäjä voi määritellä sovelluksensa. Erilaisten työkalujen ja pienoisohjelmien avulla käyttäjä määrittelee siinä sovelluksen tietomallin, toimintalogiikan, käyttöoikeudet jne. Palvelintaso sisältää koodin generoinnin, kääntämisen ja optimoinnin sekä erilaisia palveluja mukaan lukien versionhallinta, käyttöönottopalvelut ja ohjelman toiminnan seuraaminen ja kirjaaminen. Kolmas taso on alustaan integroidut ulkoiset palvelut, esimerkiksi tietokantapalvelin ja ulkoiset ohjelmointirajapinnat. Kehitysalustaa käyttämällä sovelluksen kehittäjän ei siis tarvitse huolehtia sovelluksen matalan tason arkkitehtuurin yksityiskohdista, esimerkiksi

tietokannan tyypistä, tietojen eheydestä tai kyselyjen optimoinnista, käyttäjien todennuksesta ja tietoturvallisuudesta eikä kuormituksen tasapainosta.

Vähäkoodisen sovelluskehityksen hyötyjä on kuvattu useissa tutkimuksissa, kuten edellä on kuvattu. Automatisoidun sovellustuotannon hyödyt ovat SoulCoren (2019) mukaan seuraavat:

- Radikaalisti lyhyemmät toimitusten läpimenoajat
- Asiakkaamme näköiset, käyttäjäystävälliset ja kustannustehokkaat sovellukset
- Osallistava ja innostava sovelluksen toteutusprosessi
- Työläät ominaisuudet automaattisesti osana sovelluksia
- Sovellusten laadun ja käytettävyyden jatkuva parantaminen
- Sovelluksen laajentaminen ja modernisointi samoin menetelmin

4.4 Esimerkkisovellus

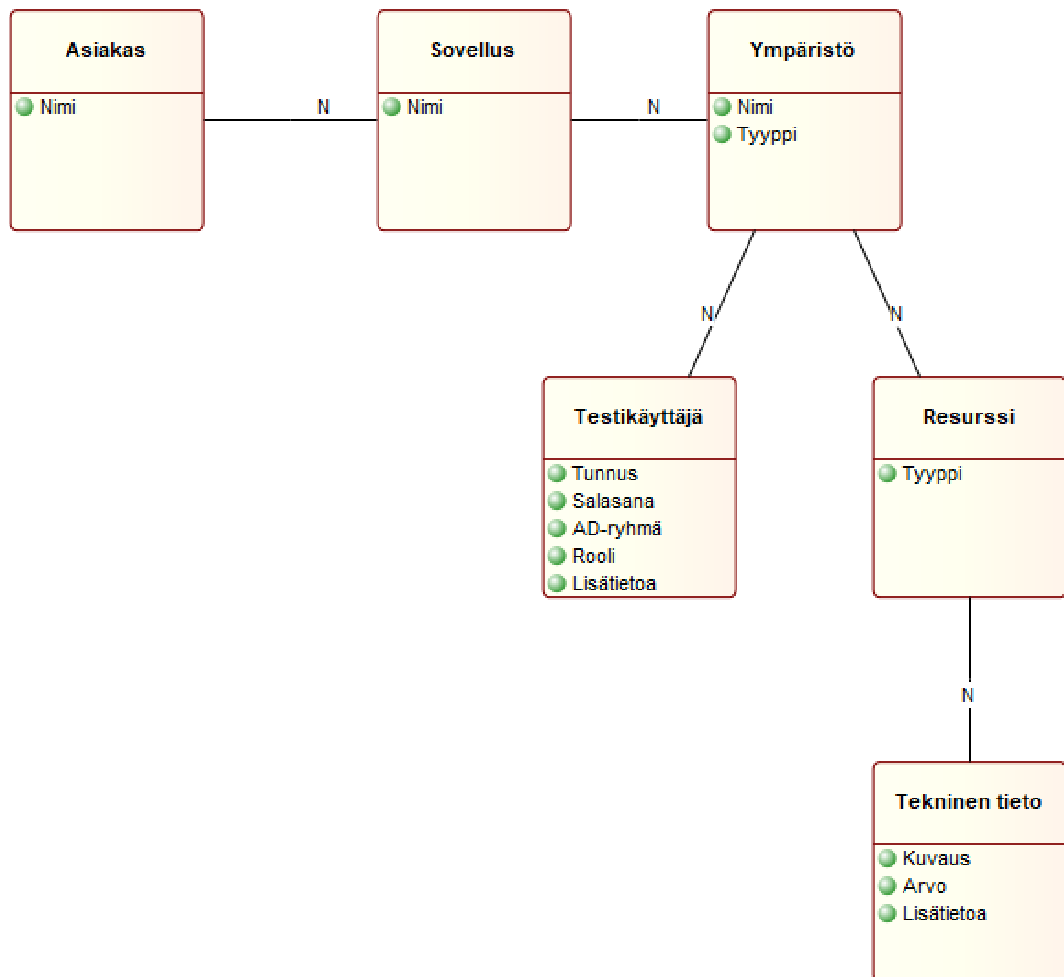
Esimerkkisovellukseksi valittiin yhdessä SoulCoren asiantuntijoiden kanssa verkkosovellus, jolla voi hallita yrityksen asiakasprojektien tietoja. Tämä koettiin yrityksessä mielekkääksi kehittämiskohteeksi, sillä se vastaa täysin sitä, millaisia sovelluksia yrityksessä tuotetaan, joskin pienemmässä koossa, koska suuren sovelluksen toteuttamiseen tällaista tutkimustyötä varten ei olisi resursseja.

Sovelluksen tietoihin liittyvät käsitteet ovat puumaisesti hierarkkisia, ja ne on esitelty tarkemmin Kuvassa 5.

Sovellukselle tehtiin seuraava vaatimusmäärittely:

- Verkkosovellus listaa asiakkaat, niille kullekin kehitetyt sovellukset ja sovellusten käyttöönottoympäristöt.
- Sovellukseen voidaan lisätä uusia asiakkaita, sovelluksia ja ympäristöjä. Niitä voidaan myös muokata tai poistaa.
- Listaa kunkin käyttöönottoympäristön testikäyttäjät ja resurssit sekä kuhunkin resurssiin liittyvät tekniset tiedot (kuvaus, arvo ja mahdollisuus lisätietoihin).

- Jokaisesta testikäyttäjästä esitetään tarvittavat tiedot (tunnus, salasana, AD-ryhmä, rooli, mahdollisuus lisätietoihin).
- Kaikkia edellä mainittuja voidaan lisätä, muokata tai poistaa.



Kuva 5. *Esimerkkisovelluksen tietorakennekaavio*

Sovelluksen tulisi esittää tiedot kahtena sovellusnäkymänä. Ensimmäinen näkymä sisältäisi puumaisen listan asiakkaista, sovelluksista ja ympäristöistä. Toinen näkymä sisältäisi yhden valitun ympäristön tiedot. Muita vaatimuksia käyttöliittymälle ei asetettu.

4.5 Tutkimuksen toteutus

Tutkimus suoritettiin siis ohjelmoimalla esimerkkisovellus kahdella eri tavalla:

1. perinteisesti ohjelmoiden sekä
2. toimialakohtaista mallinnusta ja SoulCoren kehittämää sovellusgeneraattoria käyttäen.

Perinteisen ohjelmoinnin työkaluiksi valittiin Microsoft Visual Studio ja ASP.NET-teknologia. ASP.NET on Microsoftin luoma verkkosovellusten kehittämiseen tarkoitettu ohjelmistokehys (Ryan ym. 2013). Perusteena valinnalle oli se, että ilman sovellusgeneraattoria nämä olisivat mahdollisia ohjelmointityökaluja kyseisen ratkaisun ohjelmointiin, ja yrityksen rakentama sovellusgeneraattori perustuu näihin samoihin työkaluihin. Yhtenä valintaperusteena oli myös se, että tutkimuksen tekijällä ei ollut aiempaa kokemusta näiden teknologioiden käytöstä. Näin ollen vertaileva tutkimus oli mahdollista rakentaa siten, että sekä perinteisen että generoidun sovelluskehittämisen osalta sovelluskehittäjänä toimi noviisitasoinen ohjelmistonkehittäjä. Sovelluslogiikka ohjelmoitiin C#:lla, joka on yksi nykyisin yleisimmin käytettyjä ohjelmointikieliä (GitHub 2019, TIOBE 2020).

Toimialakohtaisen mallinnuksen työkaluna käytettiin SoulCoren sovellusgeneraattoria, jota on kuvattu tarkemmin luvussa 4.2. Perinteinen toteutus päätettiin tehdä ensin, jottei generoimalla tuotettu lopputulos ohjaisi toteutusta liikaa.

Sovelluskehityksen havainnoimiseksi ja seuraamiseksi päätettiin systemaattisesti kirjata muistiin, kuinka paljon aikaa ohjelmoinnin toteuttamisen eri vaiheet kestävät. Aikakirjanpidossa käytetty aika päätettiin kirjata sen mukaan, onko se käytetty tietokannan luomiseen, sovelluslogiikan ohjelmointiin, käyttöliittymän ohjelmointiin vai mallin luomiseen.

Kun ohjelmoija ei ole aiemmin käyttänyt työssä käytettäviä työkaluja, tehtävän suorittamiseen kuluvaan aikaan tulee mukaan myös opetteluun kuluva aika (Kelly 2013). Tämän vaikutuksen vähentämiseksi käytettäviin työkaluihin tutustuttiin hieman etukäteen. ASP.NET-teknologiaan tutustuttiin muutaman tutoriaalin avulla ja SoulCoren mallinnusta esiteltiin lyhyesti. Muuta aiempaa kokemusta kummastakaan toteutustavasta ei ollut. Kelly (2013) huomauttaa lisäksi, että tehtävän suorittamiseen kuluva aika on mittarina luotettavimmillaan silloin, kun tekijän osaaminen riittää helposti tehtävän suorittamiseen, ja varoittaa, että käytetty aika lähestyy ääretöntä, mikäli tehtävä on tekijän kykyjen rajoilla. Siksi tässä tutkimuksessa molempien esimerkkisovellusten suorittamisen aikana apua voi pyytää kokeneemmalta ohjelmoijalta, jotta tehtävien suorittaminen ei pääse jumiutumaan osaamattomuuden vuoksi.

5. TUTKIMUKSEN SUORITTAMINEN

Tässä luvussa kerrotaan esimerkkisovelluksen toteutuksesta ensin perinteisesti ohjelmoiden ja sitten SoulCoren sovellusgeneraattoria käyttäen. Lopuksi vertaillaan näitä toteutustapoja.

5.1 Esimerkkitoteutus 1: Perinteinen ohjelmointi

Sovellus toteutettiin Microsoftin ASP.NET-teknologialla Visual Studiolla. Tietokanta tehtiin SQL Server Management Studiolla.

Toteutus aloitettiin luomalla tietokanta SQL Server Management Studiolla SQL-lauseita käyttäen ja yhdistämällä tietokanta Visual Studiolla luotuun ASP.NET-projektiin. Tietokanta luotiin tietomallin pohjalta (esitelty luvussa 4.4). Tietokannan luomiseen aikaa meni 2 tuntia. Myöhemmin tietokannasta jouduttiin korjaamaan virheitä ja tekemään muita pieniä muutoksia, joiden vuoksi kokonaisuudessaan tietokannan tekemiseen käytetty aika oli 5,5 tuntia.

Sovelluslogiikka ohjelmoitiin MVC-arkkitehtuuria (model-view-controller eli malli-näkymä-käsittelijä) käyttäen. MVC-arkkitehtuuri on erityisesti graafisten käyttöliittymien ohjelmointiin tarkoitettu suunnittelumalli. Siinä ohjelmoitava sovellus jaetaan osiin siten, että **malli** sisältää ohjelmassa tarvittavan tiedon ja logiikan sen muuttamiseen, **näkymä** näyttää tiedon käyttäjälle ja **käsittelijä** muuntaa käyttäjän näkymässä antamat käskyt mallin muutostoimenpiteiksi (Leff & Rayfield 2001). Esimerkkisovelluksessa malli ja käsittelijä ohjelmoitiin C#:lla, ja tämä osuus laskettiin sovelluslogiikan ohjelmoinniksi, johon kului 12,5 tuntia.

Näkymien ohjelmoinnissa käytettiin HTML-kieltä ASP.NET-lisäyksillä. Visual Studion ASP.NET-työkaluilla saatiin luotua automaattisesti käyttöliittymäpohjat uusien kohteiden lisäämiselle ja tietojen muokkaamiselle, mutta koska käyttöliittymästä haluttiin dynaaminen ja samalla sivulla toimiva, ne korvattiin lopulta JavaScript-koodilla, eli käytettiin tähän tarkoitukseen kehitettyä yleisesti käytössä olevaa ohjelmointikieltä (GitHub 2019, TIOBE 2020). Se lisäsi merkittävästi käyttöliittymän ohjelmointiin kulunutta aikaa, joka lopulta oli 16,5 tuntia.

Sovelluksen valmistuttua se esiteltiin yrityksen toimihenkilöille ja todettiin vaatimusmäärittelyitä vastaavaksi. Sovelluksen ulkonäköä esitellään Kuvissa 6 ja 7.

Application name

Asiakkaat

Asiakas1
Muuta nimeä Poista asiakas

Sovellukset

Sovellus1
Muuta nimeä Poista sovellus

Ympäristöt

abcdemo	Tuotanto
asdf	Kehitys

Lisää uusi ympäristö

Sovellus2

Lisää uusi sovellus

Asiakas2

Asiakas4

Asiakas5

Lisää uusi asiakas

© 2018

Kuva 6. *Esimerkkitoteutus 1: sovelluksen ensimmäinen sivu*

Application name

abcdemo

Tuotanto

Muokkaa ympäristöä Poista ympäristö

Testikäyttäjät

tunnus	salasana	AD-ryhmä	rooli	lisätietoa		
laura	qwerty		admin		Muokkaa	Poista
asdf	ghjk				Muokkaa	Poista

Lisää uusi testikäyttäjä

Resurssit

resurssi	kuvaus	arvo	lisätietoja		
Tietokantapalvelin Muokkaa Poista	jotain	jotain		Muokkaa	Poista
	asdf	asfd		Muokkaa	Poista
	Lisää uusi tekninen tieto				
Website Muokkaa Poista	aaaa	bbbb		Muokkaa	Poista
	cccc	dddd		Muokkaa	Poista
	Lisää uusi tekninen tieto				

Lisää uusi resurssi

Takaisin

© 2018

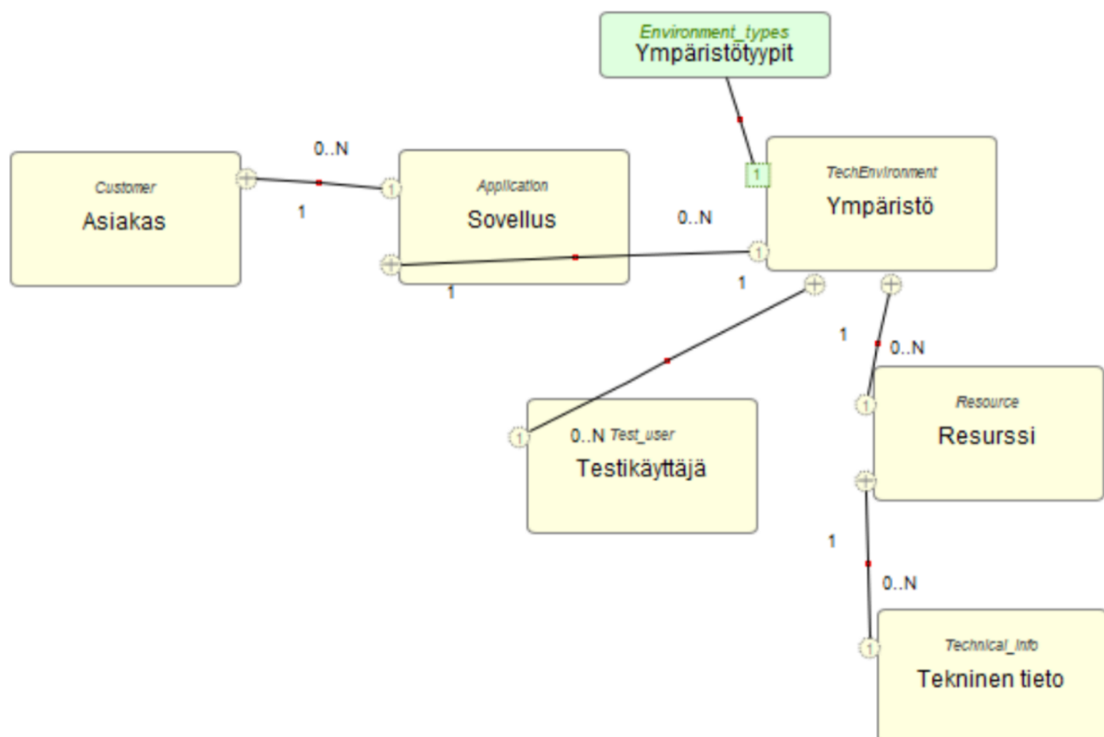
Kuva 7. *Esimerkkitoteutus 1: sovelluksen toinen sivu*

Aikaa sovelluskehitykseen kului yhteensä 36 tuntia, josta 5,5 tuntia tietokantaan, 12,5 tuntia sovelluslogiikkaan, 16,5 tuntia käyttöliittymään ja 1,5 tuntia muuhun (projektin luominen yms.).

5.2 Esimerkkitoteutus 2: Mallista generoiden

Sovellus toteutettiin SoulCoren generointisysteemillä. SoulGen Modelerilla tehdään malli, josta MetaEdit+:lla tehty generaattori tekee XML-koodia. Javalla tehty SoulGen-generaattori tekee XML-koodista projektissa tarvittavan koodin (HTML, JavaScript, C#, tietokantakoodi).

Ensin SoulGen Modelerilla luotiin tietomalli. Malli luotiin esimerkkisovelluksen määrittelyvaiheessa tehdyn tietorakennekaavion pohjalta. Jokaiseen käsitteeseen listattiin tarvittavat ominaisuudet, ja käsitteet liitettiin toisiinsa liitoksilla, jotka määrittävät mallista luotavan ohjelman toimintaa. Malli on nähtävissä Kuvassa 8. Mallin luomiseen meni aikaa 1,25 tuntia.

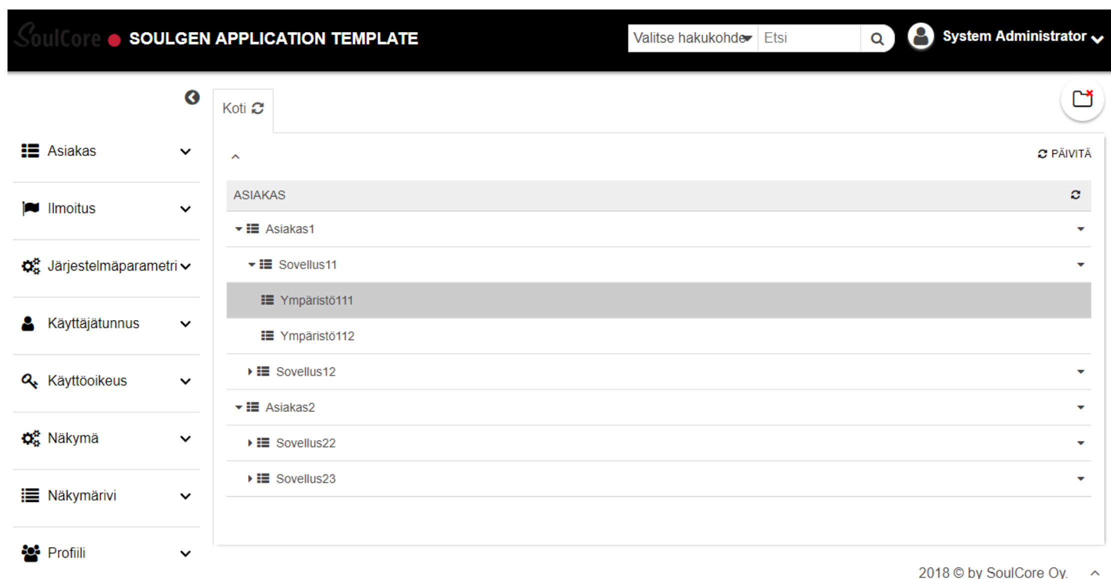


Kuva 8. *SoulGen Modelerilla tehty tietomalli*

SoulGen Modeler generoi mallista XML-koodin, joka syötettiin SoulGen-generaattorille. SoulGen-generaattori tuotti valmiin ohjelmakoodin, minkä jälkeen ohjelma oli valmis käytettäväksi selaimessa. Varsinaiseen generointiin kulunut aika oli mitättömän pieni, mutta tarvittavien ohjelmien säätämisessä meni jonkin verran aikaa, joka on laskettu luokkaan muu (yhteensä 3 tuntia).

Valmiin ohjelman käyttöliittymää piti vielä hieman muokata, jotta se näyttäisi halutut asiat halutulla tavalla. Näkymää pystyi muokkaamaan helposti suoraan ohjelman kautta SQL-lauseiden avulla. Käyttöliittymän muokkaamiseen kului 2,25 tuntia.

Sovelluksen lopullinen ulkonäkö on nähtävillä Kuvissa 9 ja 10. Sovelluksen valmistuttua se jälleen esiteltiin yrityksen toimihenkilöille ja todettiin vaatimusmäärittelyitä vastaavaksi.



Kuva 9. *Esimerkkitoteutus 2: sovelluksen ensimmäinen sivu*

Kuva 10. *Esimerkkitoteutus 2: sovelluksen toinen sivu*

Aikaa kului yhteensä 6,5 tuntia, josta 1,25 tuntia mallin luomiseen, 2,25 tuntia käyttöliittymän muokkaamiseen ja 3 tuntia muuhun (projektin luominen yms.).

5.3 Toteutustapojen vertailu

Eri toteutustavoilla tehdyistä ohjelmista ei tullut täsmälleen samanlaisia, vaikka ne sisältävätkin samat ominaisuudet. Molemmat todettiin vaatimuksia vastaaviksi ja riittävän samankaltaisiksi, jotta niiden vertailu on mielekästä.

Taulukosta 3 nähdään, että sovelluksen tekeminen toimialakohtaisella mallinnuksella oli huomattavasti nopeampaa kuin perinteisesti ohjelmoiden. Kuten Taulukko 3 osoittaa, perinteinen ohjelmointi ja mallin generointi ovat luonteeltaan hyvin erilaisia tapoja sovelluskehitykselle. Perinteinen ohjelmointi pohjautuu tietokannan rakentamiseen ja sovelluslogiikan ohjelmointiin, joihin ajallisesti kului suurin osa ohjelmoijan työajasta. Toinen ajallisesti vaativa tehtäväalue on käyttöliittymän rakentaminen.

Taulukko 3. *Esimerkkisovelluksen toteutustapojen vertailu*

	Perinteinen ohjelmointi (tuntia)	Mallista generointi (tuntia)	Ero (tuntia)
Tietokannan luominen	5,5	-	-5,5
Sovelluslogiikan ohjelmointi	12,5	-	-12,5
Mallin luominen	-	1,25	+1,25
Käyttöliittymän muokkaaminen	16,5	2,25	-14,25
Muu (projektin luominen yms.)	1,5	3,0	+1,5
Yhteensä	**36,0**	**6,5**	**-29,5**

Mallista generoidessa voitiin kokonaan ohittaa perinteiseen ohjelmointiin kuuluva tietokanta- ja sovelluslogiikkatyö. Sen sijaan kehitystyö lähti liikkeelle mallin luomisesta, joka edellytti huomattavasti vähäisempää ajallista panostusta. Mallista generoidun sovelluksen käyttöliittymää piti myös hieman muokata, mutta se ei vienyt läheskään niin paljon aikaa kuin perinteisesti ohjelmoiden, koska käyttöliittymän toiminnot olivat generoidussa sovelluksessa valmiina, ja näkymää pystyi muokkaamaan suoraan sovelluksessa yksinkertaisten SQL-lauseiden avulla. Esimerkkitapauksessa käyttöliittymien muokkaamiseen liittyvä ajallinen ero oli yli 14 tuntia eli noin kaksi työpäivää; prosentuaalisesti säästyi 86 % työajasta verrattuna perinteiseen ohjelmointiin.

Muut tehtävät kuten projektin luominen veivät perinteisessä menetelmässä suhteellisen pienen ajan, 1,5 tuntia eli noin 4 % kokonaisajasta. Mallista generoidessa muut tehtävät veivät huomattavasti suuremman osan muutoin tehostuneesta sovelluskehityksestä, 3 tuntia eli noin 46 %. Tätä eroa saattaa selittää osaltaan se, että generointiin tarvittiin useampi sovellus, jotka piti saada toimimaan yhteen ja jotka vaativat yrityksen sisäisen verkon käyttämistä. Ero tähän luokkaan kuluneiden tuntien osalta toteutustapojen välillä on kuitenkin ajallisesti sen verran pieni, että kyse voi hyvin olla myös satunnaisvaihtelusta, ja pitkälle meneviä johtopäätöksiä siitä ei voi vetää.

6. TULOSTEN TARKASTELU

Tämä tutkimus oli siis vertaileva tutkimus, jossa tarkasteltiin sovelluskehityksen tehostamisen mahdollisuuksia toimialakohtaisen automatisoinnin avulla. Ohjelmointia on läpi sen historian pyritty tehostamaan ja tekemään intuitiivisemmaksi, ja sovellusgeneraattoria, toimialakohtaista mallinnusta ja kooditonta ohjelmointia hyödyntävä sovelluskehitys pyrkii eteenpäin tässä kehityksessä (Woo 2020). Tutkimuksessa selvitettiin toimialakohtaisen automatisoinnin hyötyjä toteuttamalla esimerkkisovellus kahdella eri menetelmällä: (1) ensin perinteisesti ohjelmoiden ja seuraavaksi (2) hyödyntäen SoulCoren kehittämää sovellusgeneraattoria. Koska ohjelmointityö on sovelluskehittämisen kriittinen tekijä ja erityisesti kokeneista ohjelmistonkehittäjistä on pulaa (Indeed 2016, Metsä-Tokila 2017, Ojala 2019, Räty 2019), tutkimuksessa haluttiin selvittää noviisitasoisen ohjelmistonkehittäjän sovelluksen kehittämiseen tarvitsemaa työaikaa ja ohjelmointityön tehostamismahdollisuuksia käyttämällä vähäkoodista menetelmää ja toimialakohtaista mallinnusta.

Sovelluksen tekeminen toimialakohtaista mallinnusta käyttäen oli odotetusti nopeampaa kuin perinteisesti ohjelmoiden. Kokonaisajankäyttöä vertailemalla havaittiin, että sovelluksen tuottaminen sovellusgeneraattorilla vei peräti 80 % vähemmän aikaa kuin sovelluksen ohjelmointi perinteiseen tapaan. Kun perinteinen ohjelmointiprosessi käynnistyy tietokannan luomisella ja sovelluslogiikan ohjelmoinnilla, nämä vaiheet olivat kokonaan tarpeettomia ja voitiin ohittaa, kun ohjelmointityö suoritettiin koodittomasti sovellusgeneraattorilla. Sahayn ym. (2020) esittämän vähäkoodisen kehitysprosessin mukaisesti kooditon ohjelmointityö käynnistyy tietomallinnuksella.

Sovellusgeneraattoria hyödyntäen ohjelmointityöaikaa säästyi huomattavasti myös käyttöliittymän muokkaamisesta. Sovellusgeneraattorilla luodun ohjelman näkymän muokkaaminen onnistui helposti sovelluksessa itsessään muutaman SQL-lauseen avulla. Perinteisen koodauksen menetelmäksi valittiin tunnettu ja yleisesti käytetty ohjelmointikieli C# (GitHub 2019, TIOBE 2020). Sen lisäksi käyttöliittymän rakentamiseen jouduttiin kuitenkin käyttämään toista ohjelmointikieltä JavaScriptiä, joka on myös yleisimmin käytettyjen ohjelmointikielten joukossa (GitHub 2019, TIOBE 2020). Käyttöliittymän muokkaamiseen kului näin aikaa yli seitsenkertaisesti verrattuna sovellusgeneraattorilla automaattisesti luodun käyttöliittymän muokkaamiseen. Kahden erilaisen ohjelmointikielen käyttö osoittaa, että myös monet perinteisessä ohjelmoinnissa yleisesti käytetyt kielet

ovat käyttötarkoitukseltaan suhteellisen kapearajaisia, ja käytännön ohjelmointityö edellyttää siksi usein ohjelmointiosaamista usealla ohjelmointikielillä. Tämä luonnollisestikin tuottaa lisähaasteita ohjelmointityön ammattilaisten koulutukselle ja osaajien rekrytoinnille.

Vaikka sovellusgeneraattorilla toteutettu ohjelmointiprosessi sisälsi mallin luomisen, jota perinteisessä prosessissa ei tarvittu lainkaan, tämä vaihe osoittautui hyvin nopeaksi (vain 1,25 tuntia). Sovellusgeneraattorilla toteutettu työ sisälsi myös jonkin verran perinteistä ohjelmointiprosessia enemmän pieniä "muu ohjelmointityö" -luokkaan määriteltyjä tehtäviä, joiden kokonaismäärä oli kuitenkin molemmissa prosesseissa vähäinen. Tutkimus vahvisti Holtkampin (2015) väitteen ohjelmointityön vahvasta kontekstisidonnaisuudesta. Perinteinen ohjelmointi ja mallin generointi olivat luonteeltaan hyvin erilaisia sovelluskehityksen tapoja, ja sovelluskehittäjän oli muokattava työskentelyään kulloinkin käytettävän työympäristön mukaiseksi.

Kellyn (2013) mukaan tehtävän suorittamiseen kuluva aika on mittarina luotettavimmillaan silloin, kun tekijän osaaminen riittää helposti tehtävän suorittamiseen, mutta käytetty aika lähestyy ääretöntä, mikäli tehtävä on tekijän kykyjen rajoilla. Mikäli tekijä ei ole aiemmin käyttänyt kyseisiä työkaluja, tehtävän suorittamiseen kuluvaan aikaan tulee mukaan myös opetteluun kuluva aika (Kelly 2013). Näiden seikkojen vaikutusta pyrittiin tämän tutkimuksen tapauksessa vähentämään siten, että ennen esimerkkisovelluksen suorittamista tehtiin muutama ASP.NET-teknologiaa käsittelevä tutoriaali kyseisten teknologioiden oppimiseksi, ja SoulCoren mallinnusta esiteltiin lyhyesti. Muuta aiempaa kokemusta kummastakaan toteutustavasta ei ollut. Lisäksi molempien esimerkkisovellusten suorittamisen aikana apuna oli kokeneempi ohjelmoija, joten tehtävien suorittaminen ei päässyt jumiutumaan osaamattomuuden vuoksi. Kokemattomuudella oli varmasti vaikutusta tutkimuksen tulokseen, mutta koska eri toteutustavat olivat molemmat ennestään tuntemattomia ja havaittu ero käytetyissä ajoissa niin suuri, tulosta voidaan kuitenkin pitää hyvin suuntaa antavana. Kelly ja Tolvanen (2008) huomauttavat, että on odotettavissa, että tuottavuushyöty olisi suurempi, kun käyttäjä olisi perehtynyt mallinnukseen paremmin.

Tutkimuksen rajoitteena on sen pienuus – vain yksi ohjelmoija ja yksi sovellus. Kun tutkitaan vain yhden ohjelmoijan tulosta ja kokemuksia, on mahdollista, että jonkun toisen ohjelmoijan kyseessä ollessa tulokset ja kokemukset olisivat olleet erilaiset. Vain yhden esimerkkisovelluksen käyttäminen tutkimuksessa jättää myös epäselväksi, kuinka sa-

manlaisia tulokset olisivat olleet jonkin eri sovelluksen tapauksessa. Toisaalta kaikkiin vastaaviin sovelluksiin liittyy samankaltaiset kehitysvaiheet, ja SoulCore on erikoistunut juuri tietyntyyppisten sovellusten ohjelmointiin. Esimerkkisovellus pyrittiin valitsemaan mahdollisimman edustavaksi esimerkiksi siitä, millaisia sovelluksia SoulCore tekee, mutta käytettävissä olevien resurssien rajallisuuden vuoksi se on toki paljon pienempi. On todennäköistä, että laajemmassa sovelluksessa toteutustapojen ero olisi tullut vielä selkeämmin esille.

Tutkimuksen tuloksista ja niiden merkityksestä käytiin keskustelu asiakasyrityksen johdon kanssa. SoulCoren toimitusjohtajan Tiina Vestmanin (2019) mukaan tutkimuksen tulos antaa hyvin suuntaa erosta, mutta vertailun ongelmana on esimerkkisovelluksen pienuus sekä se, ettei käsin tehty sovellus sisällä kaikkia niitä ominaisuuksia, joita automaattisesti generoituun sovellukseen tulee automaattisesti, esimerkiksi kieleistys, lokitus ja tietosuojaominaisuudet. Hän uskoo, että noiden ominaisuuksien toteuttaminen tekisi vielä suuremman eron käsin ohjelmoituun ja automaattisesti generoituun sovellukseen käytettyjen aikojen välille, ja toivoo, että sellaisen vertailun tekeminen olisi joskus mahdollista.

7. YHTEENVETO

Tutkimuksessa vertailtiin verkkosovelluksen ohjelmointia kahdella eri tavalla: ensin perinteisesti ohjelmoiden käyttäen Microsoftin ASP.NET-teknologiaa ja sitten toimialakohtaiseen mallinnukseen perustuvaa sovellusgeneraattoria käyttäen. Vertailussa keskityttiin vertailemaan sovelluksen toteuttamiseen kuluvaa aikaa, koska sen katsottiin parhaiten kuvaavan ohjelmistokehityksen tehokkuutta. Sovelluksen luomisen sovellusgeneraattorilla havaittiin olevan selvästi nopeampaa kuin perinteisesti ohjelmoiden.

Sovelluksen tuottaminen sovellusgeneraattorilla vei peräti 80 % vähemmän kokonaistyöaikaa kuin sovelluksen ohjelmointi perinteiseen tapaan. Vähäkoodisella prosessilla sovellusgeneraattorilla ohjelmoiden ohjelmointityöaikaa säästyi erityisesti tietokannan luomisesta ja sovelluslogiikan ohjelmoinnista, joilla työvaiheilla perinteinen ohjelmointiprosessi käynnistyy, mutta jotka voitiin sovellusgeneraattorilla kokonaan ohittaa. Lisäksi sovellusgeneraattoria hyödyntävässä ohjelmoinnissa ohjelmointityöaikaa säästyi huomattavasti käyttöliittymän muokkaamisvaiheessa. Perinteiseen ohjelmointiin verrattuna sovellusgeneraattorilla tehtävä prosessi käynnistyi mallin luomisella, jota perinteisessä prosessissa ei siis tarvita lainkaan. Tämä vaihe osoittautui kuitenkin varsin nopeaksi. Sovellusgeneraattorilla toteutettu työ sisälsi myös jonkin verran perinteistä ohjelmointiprosessia enemmän pieniä "muu ohjelmointityö" -luokkaan määriteltyjä tehtäviä, joiden kokonaismäärä oli kuitenkin molemmissa prosesseissa vähäinen.

Perinteinen ohjelmointi ja mallin generointi olivat luonteeltaan hyvin erilaisia sovelluskehityksen tapoja, ja sovelluskehittäjän oli muokattava työskentelyään kulloinkin käytettävän työympäristön mukaiseksi. Tutkimus osoitti, että myös noviisitasoinen ohjelmistonkehittäjä kykenee lyhyen perehdytyksen jälkeen hyödyntämään vähäkoodista menetelmää ja toimialakohtaista mallinnusta hyödyntävää sovellusgeneraattoria tehokkaasti, ja että ohjelmointityötä on mahdollista sitä käyttäen huomattavastikin tehostaa.

LÄHTEET

The American Society of Mechanical Engineers (ASME). (2015). Industry 4.0. A discussion of qualifications and skills in the factory of the future. A German and American perspective. New York: The American Society of Mechanical Engineers. Saatavissa (viitattu 13.3.2021): https://www.vdi.de/ueber-uns/presse/publikationen/details/industry-40-a-discussion-of-qualifications-and-skills-in-the-factory-of-the-future-a-german-and-american-perspective

Bellairs, R. (2019). What is code quality? And how to improve code quality. Coding Best Practices / Static Analysis. Saatavissa (viitattu 12.10.2021): https://www.perforce.com/blog/sca/what-code-quality-and-how-improve-code-quality

Björklund, T. (2013). Initial mental representations of design problems: Differences between experts and novices. Design Studies 34, s. 135–160.

Caballar, R. D. (2020). Programming Without Code: The Rise of No-Code Software Development. IEEE Spectrum. Saatavissa (viitattu 24.10.2020): https://spectrum.ieee.org/tech-talk/computing/software/programming-without-code-no-code-software-development

Dzidek, W. J., Arisholm, E. & Briand, L. C. (2008). A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. IEEE Transactions on Software Engineering. Vol.34(3), s. 407–432.

El Kouhen, A., Dumoulin, C., Gérard, S. & Boulet, P. (2012). Evaluation of Modelling Tools Adaptation. CNRS HAL. Saatavissa (viitattu 10.10.2021): https://hal.archives-ouvertes.fr/file/index/docid/706701/filename/Evaluation_of_Modeling_Tools_Adaptation.pdf

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

France, R. & Rumpe, B. (2007). Model-driven Development of Complex Software: A research Roadmap. Future of Software Engineering, IEEE Computer Society.

GitHub (2019). The State of the Octoverse. Saatavissa (viitattu 21.11.2019): https://octoverse.github.com/

Harsu, M. (2012). Ohjelmointikielet – Periaatteet, käsitteet, valintaperusteet. Saatavissa (viitattu 14.11.2019): http://www.cs.tut.fi/~popl/nykyinen/Ohjelmointikielet-harsu.pdf

Holtkamp, P. (2015). Competency Requirements of Global Software Development: Conceptualization, Contextualization, and Consequences. Jyväskylä Studies in Computing 221. Jyväskylän yliopisto.

Indeed (2016). Is the War for Tech Talent Hurting Innovation? Hiring Managers, Recruiters Respond. Saatavissa (viitattu 31.10.2020): https://www.indeed.com/lead/impact-of-tech-talent-shortage

Kaarakainen, M-T. & Saikkonen, L. (2019). Tekniikan alojen opiskelijoiden digitaaliset valmiudet suhteessa työelämän ja opintojen muuttuviin osaamisvaatimuksiin. Ammattikasvatuksen aikakauskirja, 21(4), s. 26–44.

Kaluža, M., Troskot, K. & Vukelić, B. (2018). Comparison of front-end frameworks for web applications development. Zbornik Veleučilišta u Rijeci, 6(1), s. 261–282.

Kaur, R. & Sengupta, J. (2011). Software Process Models and Analysis on Failure of Software Development Projects. International Journal of Scientific & Engineering Research. Vol.2(2).

Kelly, S. (2013). Empirical Comparison of Language Workbenches. Proceedings of the 2013 ACM workshop on Domain-specific modeling, s. 33–37.

Kelly, S. & Tolvanen, J. (2008). Domain-specific modeling: enabling full code generation. John Wiley & Sons.

Kleppe, A., Warmer, J. & Bast, W. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional.

Krach, S. (2019). Towards domain-specific extensibility of quality-aware software architecture meta-models. 10th Symposium on Software Performance. Saatavissa (viitattu 20.10.2021): https://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Krach.pdf

Kärnä, J., Tolvanen, J. & Kelly, S. (2009). Evaluating the use of domain-specific modeling in practice. Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling. Saatavissa (viitattu 11.10.2021): http://www.dsmforum.org/events/dsm09/Papers/Karna.pdf

Leff, A. & Rayfield, J. T. (2001). Web-application development using the Model/View/Controller design pattern. Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference.

MetaCase (2000). Case Study: MetaEdit+ Revolutionized the Way Nokia Develops Mobile Phone Software. Saatavissa (viitattu 11.10.2021): www.metacase.com/papers/MetaEdit_in_Nokia.pdf

Metsä-Tokila, T. (2017). Kasvun mahdollistajat – ohjelmistoala ja tekninen konsultointi. Työ- ja elinkeinoministeriön toimialaraportti. Saatavissa (viitattu 31.10.2020): https://julkaisut.valtioneuvosto.fi/bitstream/handle/10024/80868/Ohjelmistoala_ja_tekninen_konsultointi.pdf

Nunamaker, J., Chen, M. & Purdin, T. (1991). Systems Development in Information Systems Research. Journal of Management Information Systems. Vol.7(3), s. 89–106.

Ogundare, O. (2017). How Do You Know What You Know: Epistemology in Software Engineering. Journal of Software Engineering and Applications, 10(02), s. 168–173. Saatavissa (viitattu 27.2.2021): https://www.scirp.org/pdf/JSEA_2017022415054805.pdf

Ojala, P. (2019). Ohjelmistoalan osaajapula, alan vaatimukset ja työllistyminen alalle. ePooki. Oulun ammattikorkeakoulun tutkimus- ja kehitystyön julkaisut 75. Saatavissa (viitattu 31.10.2020): https://www.theseus.fi/bitstream/handle/10024/262617/ePooki%2075_2019.pdf

Perusopetuksen opetussuunnitelman perusteet. (2014). Opetushallituksen määräys 22.12.2014. Saatavissa (viitattu 13.3.2021): https://www.oph.fi/fi/koulutus-ja-tutkinnot/perusopetuksen-opetussuunnitelman-perusteet

Puolitaival, O., Kanstrén, T., Rytky, V. & Saarela, A. (2011). Utilizing Domain-Specific Modelling Software Testing. The 3rd International Conference on Advances in System Testing and Validation Lifecycle. Saatavissa (viitattu 11.10.2021): https://www.researchgate.net/profile/Teemu-Kanstren/publication/266523830_Utilizing_Domain-Specific_Modelling_for_Software_Testing/links/54cd3e5f0cf29ca810f7d98b/Utilizing-Domain-Specific-Modelling-for-Software-Testing.pdf

Richardson, C. & Rymer, J. R. (2014). New Development Platforms Emerge For Customer-Facing Applications. Forrester.

Russell, S. & Norvig, P. (2016). Artificial Intelligence: A Modern Approach, 3rd ed. Pearson Education Limited.

Ryan, W., de Kort, W. & Milton, S. (2013). Developing Windows Azure and Web Services. Pearson Education.

Räty, P. (2019). Automaatiosta vauhtia ohjelmointityöhön. Tivi 09/2019, s. 36.

Sadrieh, A. & Bahri, P. (2014). Novel Domain-Specific Language Framework for Controllability Analysis. Computer Aided Chemical Engineering. Vol.33, s. 559–564.

Safa, L. (2007). The making of user-interface designer a proprietary DSM tool. In 7th OOPSLA workshop on domain-specific modelling (DSM). Saatavissa (viitattu 11.10.2021): http://dsmforum.org/events/DSM07/papers/safa.pdf

Sahay, A., Indamutsa, A., Di Ruscio, D., & Pierantonio, A. (2020). Supporting the understanding and comparison of low-code development platforms. 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE.

Sanchis, R., García-Perales, Ó., Fraile, F. & Poler, R. (2019). Low-Code as Enabler of Digital Transformation in Manufacturing Industry. Applied Sciences, 10(1).

Sebesta, R. W. (2012). Concepts of Programming Languages, 10th ed. Pearson.

Silva, C., Vieira, J., Campos, J. C., Couto, R. & Ribeiro, A. N. (2020). Development and Validation of a Descriptive Cognitive Model for Predicting Usability Issues in a Low-Code Development Platform. Human Factors.

SoulCore (2019). Yrityksen verkkosivu. Saatavissa (viitattu 12.11.2019): https://www.soulcore.fi/

Tilastokeskus (2008). Toimialaluokitus. Saatavissa (viitattu 12.10.2021): https://www.stat.fi/fi/luokitukset/toimiala/

TIOBE (2020). TIOBE Index for January 2020. Saatavissa (viitattu 9.1.2020): https://www.tiobe.com/tiobe-index/

Tolvanen, J. & Kelly, S. (2016). Model-Driven Development Challenges and Solutions – Experiences with Domain-Specific Modelling in Industry. Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, s. 711–719.

Vestman, T., toimitusjohtaja, SoulCore. Haastattelu 30.12.2019.

Vincent, P., Iijima, K., Driver, M., Wong, J. & Natis, Y. (2019). Magic Quadrant for Enterprise Low-Code Application Platforms. Gartner.

Wernick, P. & Hall, T. (2004). Can Thomas Kuhn's paradigms help us understand software engineering? European Journal of Information Systems, 13(3), s. 235-243.

Woo, M. (2020). The Rise of No/Low Code Software Development – No Experience Needed? Elsevier Public Health Emergency Collection. Saatavissa (viitattu 12.10.2021): https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7361109/

Xie, B., Loksa, D., Nelson, G., Davidson, M., Dong, D., Kwik, H., Hui Tan, A., Hwa, L., Li, M. & Ko, A. (2019). A theory of instruction for introductory programming skills. Computer Science Education, 29(2-3), s. 205-253.