Tampereen yliopisto

Aulis Leinonen

OHJELMISTOARKKITEHTUURIN SUUNNITTELU PIENESSÄ STARTUP-YRITYKSESSÄ

Tietotekniikka
Diplomityö
Toukokuu 2019

TIIVISTELMÄ

Aulis Leinonen: Ohjelmistoarkkitehtuurin suunnittelu pienessä startup-yrityksessä
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-tutkinto-ohjelma
Toukokuu 2019

Työssä tarkasteltiin ohjelmistoarkkitehtuurin suunnittelua pienissä startup-yrityksissä kirjallisuus- ja case-tutkimuksen avulla. Näiden yritysten toimintamalleja tutkitaan kirjallisuudessa harvoin. Pienissä startup-yrityksissä arkkitehtuurisuunnittelu on haastavaa rajallisten resurssien vuoksi. Työssä keskityttiin tutkimaan Lean Startup ja "Minimum Viable Product", eli MVP-metodologioiden sekä resurssivajeen vaikutusta arkkitehtuurisuunnitteluun.

Kirjallisuustutkimuksessa tarkasteltiin startup-yrityksiä ja arkkitehtuurin suunnittelun prosesseja ja siihen liittyviä konsepteja. Case-tutkimuksessa taas tutkittiin käytännössä, kuinka pienen startup-yrityksen MVP-prosessi toimii.

Työssä selvisi, että pienen yrityksen resurssit asettavat rajoitteita arkkitehtuurisuunnittelulle ja MVP:n toteuttamiselle. Suunnittelumenetelmät, jotka vaativat vähän resursseja soveltuivat parhaiten tämän tyyppisille yrityksille. Tällaisia ovat esimerkiksi arkkitehtuuriprototyypit ja erittäin kevyt dokumentaatio. Pienen startup-yrityksen tulisi myöskin harkita vaihtoehtoja MVP-metodologialle, ennen sen käyttöönottoa. Ohjelmoidun MVP-tuotteen kehittäminen vie enemmän resursseja muun muassa ei-ohjelmoituihin prototyyppeihin verrattuna.

Case-tutkimus tarkastelee vain yhden startup-yrityksen MVP-prosessia. Työssä tehtyjä päätelmiä voidaan soveltaa muihin projekteihin harkiten, vaikka tämä tutkimus on suppea. Jatkossa pienten startup-yritysten toiminnan ja arkkitehtuurisuunnitteluprosessin tutkimusta tulisi laajentaa pieniin yrityksiin, jotka sekä menestyvät että epäonnistuvat. Niiden eroavaisuuksia tulisi tutkia, jotta voitaisiin selvittää niiden menestykseen johtavat tekijät.

Avainsanat: opinnäytetyö, ohjelmistoarkkitehtuuri, startup, lean, MVP, suunnittelu

ABSTRACT

Aulis Leinonen: Designing software architecture in small startup companies
Master of Science Thesis
Tampere University
Degree Programme in Information Technology
May 2019

This thesis studied designing software architecture in small startup companies by the means of researching other literature and performing a case study. The processes and activites of these types of companies are rarely covered in literature. It is often difficult to carry out architectural design in small startup companies due to their lack of resources. The impact of Lean Startup and "Minimum Viable Product" (MVP) -methodologies on architecture design were the main focus of this study.

The literature research covered startup companies, the processes of architecture design, and the concepts relating to it. In the case study the processes of small startup companies were examined in practice.

The studies indicated that the lack of resources in small startup companies sets restrictions for the architecture design methods and realising MVPs. The methods for desiging the architecture that require the least amount of resources seem to be the best fit for these types of companies. For example, architecture prototypes and very light documentation seemed to suit this context. Small startup companies should also consider alternatives to the MVP approach before fully committing to it. This is due to the extra effort required in programming an MVP over validation methods that do not require any programming.

The case study in this thesis only covers one startup project and its MVP process. The conclusions can be applied to other projects using consideration, even though this study is limited. In the future the processes and activities of small startup companies and their architecture design process should be expanded to multiple small startup companies. The future studies should include companies that succeed and fail. This would allow the differences between them to be scrutinised and the factors that lead to their success to be found.

Keywords: thesis, software architecture, startup, lean, MVP, design

ALKUSANAT

Tämä diplomityö on tehty Tampereen yliopiston tietotekniikan laitoksen ohjelmistotuotannon yksikköön. Työn ohjaajina ja tarkastajina toimi apulaisprofessori Kari Systä sekä tekniikan tohtori Terhi Kilamo Tampereen yliopistosta.

Kiitän kaikkia projektiin osallistuneita. Erityisesti kiitän Veli-Pekka Rajalaa ja Kari Systää.

Tampereella, 29. toukokuuta 2019

Aulis Leinonen

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

artefakti	arkkitehtuurisuunnittelussa syntynyt tuotos
backlog	Scrum-metodologissa käytetty listaus jäljellä olevista tehtävistä
dokumentaatio	kirjallinen informaatio
et al.	lat. et alii tai et alieae, ja muut
refaktorointi	uudelleen ja paremmin tekeminen, huonon laadun korjaaminen
TAU	Tampereen yliopisto (engl. Tampere University)
tekninen velka	työ, joka tietoisesti sivuutetaan, mikä aiheuttaa tulevaisuudessa enemmän työtä
TUNI	Tampereen korkeakouluyhteisö (engl. Tampere Universities)
URL	verkkosivun osoite (engl. Uniform Resource Locator)
validointi	testaus

1 JOHDANTO

Nykyisin on olemassa monia menestyviä teknologia-startup-yrityksiä, kuten Uber ja Airbnb. Ne ovat niin sanottuja yksisarvisia, eli yrityksiä, jotka tuottavat verkkopalveluita tai ohjelmistoja ja ne ovat arvoltaan yli miljardi dollaria. Tällaisten yritysten olemassaolo tekee startup-maailmasta erittäin houkuttelevan yrittäjille.

Todellisuudessa startup-yrityksen aloittaminen on vaikeaa ja monet niistä epäonnistuvat. Startup-yrityksillä ei monesti ole aluksi selkeää liiketoimintamallia, eikä maksavia asiakkaita. Lean Startup on iteratiivinen prosessi, jonka tarkoituksena on yksinkertaistaa ja helpottaa startup-yrityksen aloittamista ja päästä tilanteeseen, jossa yrityksellä on sekä testattu liiketoimintamalli että maksavia asiakkaita. Ohjelmistoa kehittävät startup-yritykset käyttävät usein "Minimum Viable Product", eli MVP-metodologiaa tuotteensa kehittämiseksi Lean Startup -prosessin ohella. Siinä ohjelmistoa tuotetaan pienissä osissa. Vaikka sillä voidaan kehittää liiketoimintamallia ja tuotetta pienillä riskeillä, se aiheuttaa joitain haasteita ohjelmistokehitykseen.

Ohjelmistoarkkitehtuurilla on suuri vaikutus ohjelmistoprojektin onnistumiseen. Sopimaton arkkitehtuuri voi johtaa ohjelmiston heikkoon laatuun, taloudellisiin menetyksiin ja jopa pahimmassa tapauksessa siihen, että ohjelmisto ei valmistu ollenkaan. Erityisesti pienissä startup-yrityksissä arkkitehtuurin suunnittelu ei ole triviaalia resurssivajeen takia. Ohjelmistolle asetetut vaatimukset voivat muuttua radikaalisti kehityksen aikana Lean Startup -prosessin takia. Arkkitehtuurin suunnittelu ennen vaatimusten varmistumista kuluttaa resursseja, sillä suunnittelu - tai toteutustyötä täytyy tehdä uudestaan. Lisäksi, arkkitehtuurisuunnittelua pienen startup-yrityksen viitekehyksessä käsitellään kirjallisuudessa harvoin.

Tässä työssä tutkitaan ketterään kehitykseen startup-ympäristöön sopivia arkkitehtuurin suunnittelumenetelmiä kirjallisuustutkimuksen avulla. Työssä selvitetään myös mitä työkaluja arkkitehtuurisuunnittelua varten on olemassa sekä miten Lean Startup ja MVP-metodologiat vaikuttavat tuotekehitykseen ja arkkitehtuurisuunnitteluun käytännössä. Näitä tarkastellaan kirjallisuus- että case-tutkimuksen avulla.

Työn luvussa 2 tarkastellaan ohjelmistoarkkitehtuuria ja sen suunnittelua yleisesti. Luku 3 esittelee startup yritysten prosesseja ja 4 esitellään kuinka ohjelmistoarkkitehtuurisuunnittelu voidaan yhdistää starup-prosessiin. Luku 5 esittelee erilaisia konsepteja, jotka liittyvät arkkitehtuurisuunnitteluun ja tarkastelee niiden sopivuutta Lean Startup -prosessiin. Luku 6 käsittelee arkkitehtuurin dokumentointia, 7 arkkitehtuurin arviointia, 8

case-tutkimusta sekä sen tuloksia ja 9 sisältää analyysin tutkimuksen tuloksista ja laajuudesta. Lopuksi luku 10 kokoaa yhteen työn tärkeimmät kohdat.

2 OHJELMISTOARKKITEHTUURI

Ohjelmistoarkkitehtuurin suunnittelun tarkoituksena on parantaa ohjelmiston laatua korkean tason pohdinnan avulla. Korkean tason suunnittelupäätösten käyttö ohjelmistotuotannossa sai alkunsa 70-luvun alussa. Tällöin havaittiin kuinka modulaarisuutta ja tiedon piilottamista voidaan käyttää ohjelmiston joustavuuden ja ymmärrettävyyden aikaansaamiseksi [50]. Halu toteuttaa aina vain monimutkaisempia ohjelmistoja nopeammin ja kustannustehokkaammin silti ylläpitäen saman laatutason on sittemmin kasvattanut mielenkiintoa ohjelmistoarkkitehtuurin suunnittelua kohtaan [62]. Arkkitehtuuri toimii myös keskustelun aiheena eri sidosryhmien kanssa keskusteltaessa [6].

Ohjelmistoarkkitehtuurille ei ole muodostunut yksiselitteistä määritelmää vaan se riippuu lähestymiskulmasta. Esimerkiksi Perry ja Wolf määrittelevät sen artikkelissaan (1992) kolmiosaisesti. Siinä se koostuu

- joukosta arkkitehtuurillisia elementtejä
- periaatteista, jotka ohjaavat elementtien välisiä suhteita ja niiden ominaisuuksia
- perusteluista tiettyjen elementtien ja niiden muodostelman valintaan. [51]

Bass ja Kazman (2003) toisaalta määrittelevät seuraavasti: Järjestelmän ohjelmistoarkkitehtuuri on

- joukko rakenteita järjestelmän miettimistä varten
- ohjelmistoelementtejä ja niiden välisiä suhteita, jotka muodostavat suuremmat rakenteet
- ohjelmistoelementtien ja suhteiden ominaisuuksia [10].

Joka tapauksessa arkkitehtuurin omaksutaan koostuvan komponenteista ja niiden välisistä suhteista. Perustelut näille ovat tärkeässä osassa. Ohjelmistoarkkitehtuurille ominaista on korkea abstraktiotaso. Pienien toteutusyksityiskohtien suunnitteluun ei arkkitehtuurisuunnittelussa ole syytä kuluttaa resursseja. Sen tarkoituksena on kuvata kuinka suuremmat kokonaisuudet kytkeytyvät toisiinsa. Tässä työssä ohjelmistoarkkitehtuurilla tarkoitetaan komponentteja, niiden välisiä suhteita sekä ominaisuuksia ja perusteluja näille.

Moderneissa ohjelmistoissa suunnittelun laiminlyöminen voi johtaa ongelmiin ohjelmiston kompleksisuuden kanssa. [62] Nämä ongelmat voivat aiheuttaa jopa ohjelmiston valmistumisen estymisen ja parhaimmassa tapauksessa johtavat vain taloudellisiin menetyksiin.

Ohjelmistoarkkitehtuurin suunnittelu tavalla tai toisella on siis oleellinen osa ohjelmistotuotantoprosessia.

Laajan ohjelmiston laadulliset ominaisuudet määräytyvätkin pääosin ohjelmiston arkkitehtuurin perusteella. Suurissa järjestelmissä niiden ominaisuudet, kuten tehokkuus, saatavuus, muunneltavuus riippuvat enemmän järjestelmän yleisarkkitehtuurista, kuin koodin tasolla tehdyistä valinnoista. Esimerkiksi ohjelmointikielen, suunnitelmien yksityiskohtien, algoritmien, tietorakenteiden tai testauksen vaikutus ei ole yhtä suuri näille ominaisuuksille, kuin arkkitehtuurin. Se ei kuitenkaan tarkoita sitä, etteivätkö matalamman tason valinnat, kuten algoritmit ja tietorakenteet ole tärkeitä, mutta niillä on pienempi vaikutus järjestelmän menestymiseen. [34]

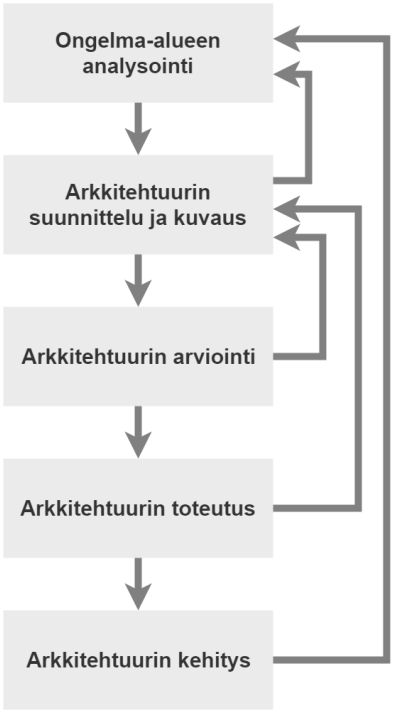
Kaikessa arkkitehtuurisuunnittelussa toistuu aina samat vaiheet. Hyvä ymmärrys sen suunnitteluprosessista ja elinkaaresta on tärkeää. Yleensä oletetaan, että arkkitehtuurisuunnitelma on luova tapahtuma, jossa ei ole tarkkaan määriteltyä prosessia. Monet suuret järjestelmät ovatkin rakentuneet tällä tavalla. Luovuuteen ja pohdintaan kannustavan prosessin seuraaminen on kuitenkin tärkeää ison järjestelmän suunnittelussa ja arvioinnissa. Kuten muillakin ohjelmistokehitykseen liittyvillä tuotoksilla on elinkaari, jossa on useita eri vaiheita ja tehtäviä. Jokaisella vaiheella on omat esiehdot sen käytölle ja sovellettavuudelle. [2, 59]

Seuraavaksi esitellään nämä vaiheet, kuten Tang et al. ovat ne kuvailleet. Ne ovat myös näkyvissä kuvassa 2.1.

1. Ongelma-alueen analysointi. Ongelma-alueen analysointi koostuu useasta alikokonaisuudesta ja tehtävästä. Sen tarkoituksena on määrittää mitkä ovat ne ongelmat, jotka pitää ratkaista. Tähän voi kuulua arkkitehtuurille asetettujen vaatimusten tutkiminen ja varmistaminen, sidosryhmien mielipiteiden kerääminen ja vaatimusten priorisointi. [2, 59]

2. Arkkitehtuurillisten päätösten suunnittelu ja kuvaus. Tässä vaiheessa pyritään tekemään tärkeimmät arkkitehtuurilliset päätökset vaatimusten perusteella. Arkkitehti voi arvioida ja päättää useamman vaihtoehdon joukosta ne, jotka ovat sopivimmat ja optimaalisimmat kyseiseen sovellukseen. Hän vastaa myös arkkitehtuurin dokumentoinnista käyttäen asiaankuuluvia dokumentointimenetelmiä. [2, 59]

3. Arkkitehtuurin arviointi. Arkkitehtuurin arvioinnissa on tarkoituksena varmistaa, että valitut ratkaisut ovat varmasti parhaiten sopivia ongelman ratkaisemiseksi ja laatuominaisuuksien takaamiseksi. Tällöin arvioidaan arkkitehdin tekemät ratkaisut vaatimuksiin nähden. [2, 59]



Kuva 2.1. *Arkkitehtuurisuunnittelu voidaan jakaa erillisiin vaiheisiin. (Muokattu [2] kuvasta.)*

4. Arkkitehtuurin toteutus. Suunnitellusta arkkitehtuurista tehdään yksityiskohtainen suunnitelma ja se toteutetaan. Ohjelmistokehittäjät tekevät useita päätöksiä, joiden tulee olla linjassa korkean tason arkkitehtuuripäätösten kanssa. Kehittäjien tulee varmistaa arkkitehtuurin suunnittelijalta, että heidän päätöksensä ovat yhteensopivia arkkitehtuurin kanssa. [2, 59]

5. Arkkitehtuurin ylläpito. Arkkitehtuurin ylläpitoon kuuluu muutoksien teko ajan myötä, kun arkkitehtuuri kehittyy parannusten ja ylläpidollisten vaatimusten takia. Nämä asettavat monia uudenlaisia vaatimuksia arkkitehtuurille ja järjestelmälle. Tiedonhallinnan näkökulmasta aiemmat päätökset uudelleenarvioidaan verraten niitä muuttuneiden vaatimuksien mahdollisiin vaikutuksiin. Tällöin tehdään uusia päätöksiä, jotka tukevat uusia vaatimuksia, mutta jotka eivät riko arkkitehtuurin yhtenäisyyttä. [2, 59]

Edellä mainitut vaiheet eivät kuitenkaan ole perättäisiä. Ne tehdään enemmänkin iteroiden tai aiemman päälle rakentaen. Tehtävät, jotka liittyvät yhteen vaiheeseen voidaan suorittaa tai niitä voidaan tehdä uudelleen samaan aikaan, kuin tehdään muihin tehtäviin liittyviä vaiheita. [2, 59]

Arkkitehtuurisuunnitelun eri vaiheisiin perehdytään työssä osissa 5, 6 ja 7. Osa 4 käsittelee arkkitehtuurisuunnittelua projektihallinnollisesta ja prosessinäkökulmasta.

3 STARTUP-PROSESSI

Startup-yritykseksi lasketaan karkeasti uudet yritykset, jotka etsivät liiketoimintamallia ja ovat kasvuhakuisia. [18] Niille on ominaista, että yrityksen luoma tuote rakennetaan validoidun oppimisen avulla. Validoidussa oppimisessa testataan liiketoimintaideaa ja tehdään tuotteeseen muutoksia tämän testauksen perusteella. Yrityksellä ei siis välttämättä ole aluksi tiedossa, mikä on heidän lopullinen myytävä tuotteensa. Tuote kehittyy ajan saatossa iteratiivisesti. [18] Muita ominaispiirteitä startup-yrityksille on niiden pienet resurssit ja nopeat kasvutavoitteet. Tutkimusta on aiemmin tehty lähinnä vakiintuneiden yritysten näkökulmasta. Pienten startup-yritysten ongelmat ovat olleet vähemmän tarkastelussa [12].

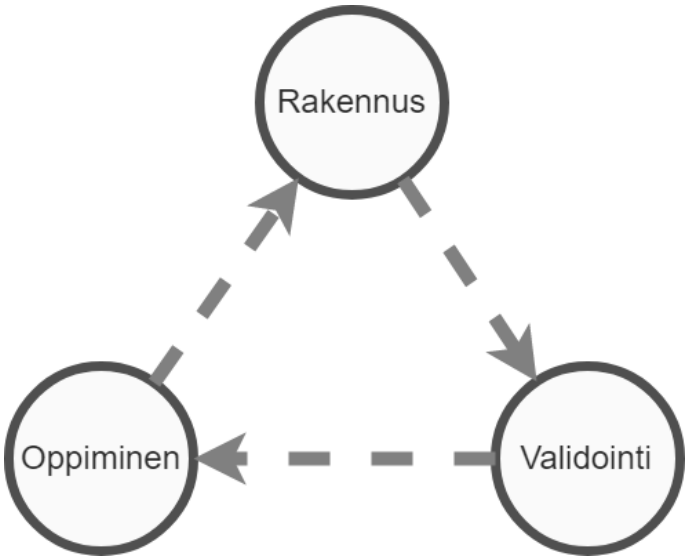
3.1 Lean Startup

Startup-yritysten aloittaminen ja tuottavaksi saaminen on sen koosta riippumatta vaikeaa. Aloituksen avuksi on kehitetty erilaisia ohjeistavia malleja, jotka esittelevät muissa startup-yrityksissä toimivaksi koettuja menetelmiä. [17] Nämä voivat yksinkertaistaa haastavaa aloitusprosessia huomattavasti.

Lean Startup on prosessimuoto, jota monet startup-yritykset käyttävät tuotekehitykseen [17, 18, 44, 60]. Se on iteratiivinen prosessi, jossa suoritetaan kolmea eri pääaktiviteettia: rakennus, validointi ja oppiminen. [52] Pääaktiviteettien suhteet on esitetty kuvassa 3.1.

Ensiksi idea liiketoimimalli kuvataan käyttäen liiketoimintamallinnuspohjaa niin että se sisältää testattavia oletuksia. Liiketoimintamallinnuspohja sisältää kaikki olennaiset asiat, joista yrittäjillä tulisi olla tarpeeksi varmuutta. [52]

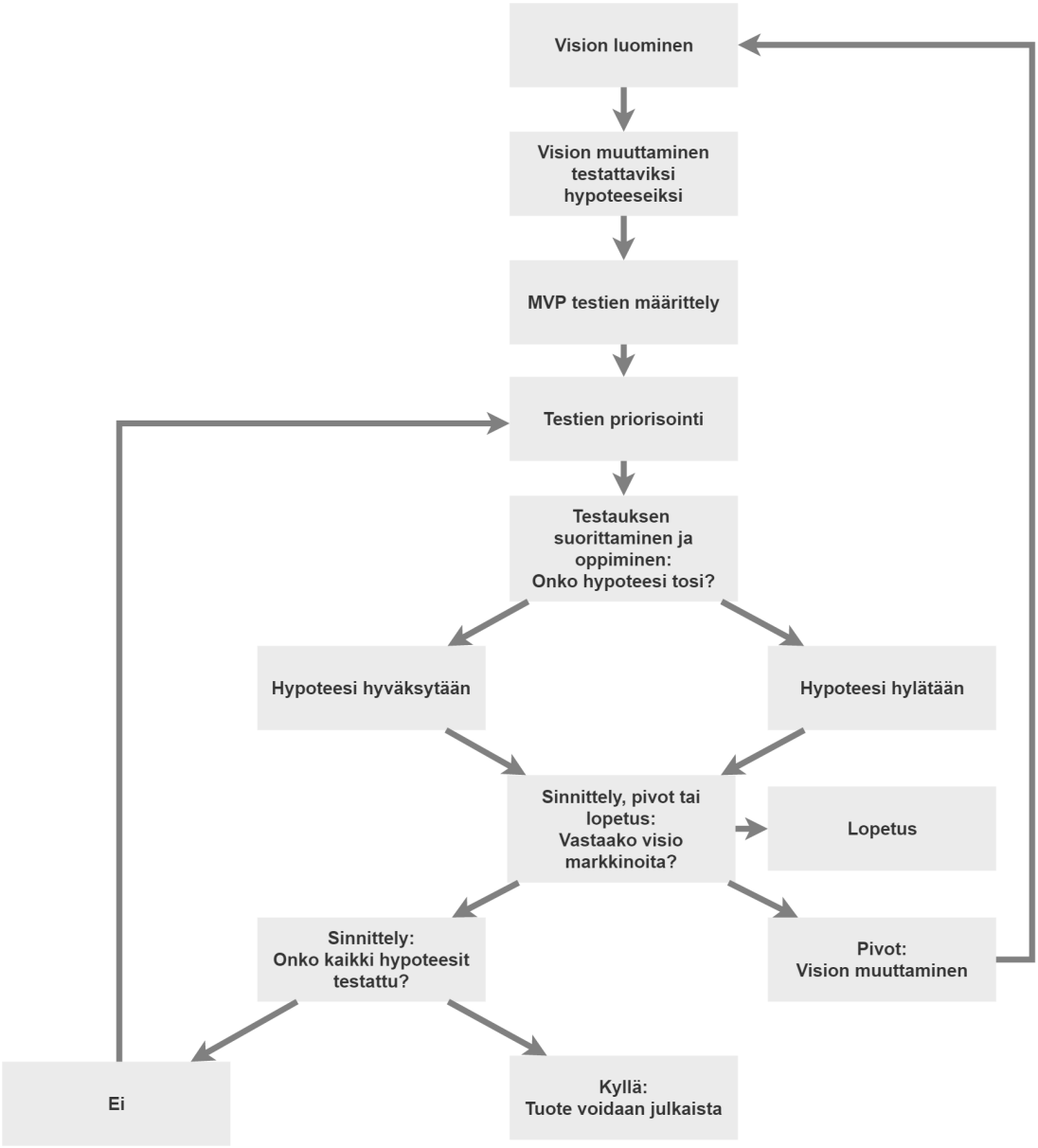
Seuraavaksi yrittäjät testaavat olettamuksia, joista liiketoimintamalli muodostuu. Testien laajuus ja tapa riippuu olettamuksien kriittisyydestä prosessin jatkon kannalta. "Minimum viable product", eli MVP on yksi olettamuksien testaustapa. [52] MVP-termillä voidaan tarkoittaa hieman eri asioita riippuen kontekstista. Tässä työssä sillä tarkoitetaan tuotteesta luotavaa ohjelmoitua versiota, joka sisältää vain vähimmäismäärän ominaisuuksia. Se toteutetaan mahdollisimman lyhyessä ajassa ja käyttäen mahdollisimman vähän resursseja, kuitenkin niin että sillä on mahdollista validoida olettamuksia. Se ei kuitenkaan ole vielä valmis tuote, vaan työkalu validointiin [60]. Validointi tulisi tehdä useimmilla asiakasryhmillä, mutta myös rahoittajilla ja alan erikoisasiantuntijoilla [17].



Kuva 3.1. *Lean Startup perustuu validoituun oppimiseen, jossa tuotetta rakennetaan osissa. (Muokattu [14] kuvasta.)*

Viimeisessä vaiheessa, kun kaikki olettamukset on validoitu, tuote on valmis markkinoille. Tällöin on selvillä, että tuotteella on asiakkaita, jotka ovat valmiita maksamaan siitä. Lean Startup -metodologian tarkoituksena on päästä tähän tilaan. [44, 52] Ennen ohjelmistokehityksen aloittamista kunnolla startup yrityksen olisi kuitenkin hyvä saada muutamia maksavia asiakkaita ja testata olettamuksia halvoilla menetelmillä [17].

Kaiken kaikkiaan Lean Startup on kaavamainen prosessi, joka sisältää pienempiä vaiheita, kuin kolmen pääaktiviteetit abstraktio pystyy kuvaamaan. Sen eri vaiheet on esitetty vielä yksityiskohtaisemmin kuvassa 3.2.



Kuva 3.2. *MVP-prosessi seuraa ennalta asetettua kaavaa. (Muokattu [44] kuvasta.)*

Ketterää kehitystä ja Scrum-prosessia voidaan hyödyntää Lean Startup -metodologian rakennusvaiheessa. Tämä koskee erityisesti liiketoimintaidean validointitapoja, joissa toteutetaan toimiva ohjelmisto testattavaksi. [14, 60] Muut testaustavat ovat yleensä keveämpiä, eivätkä ne vaadi yhtä tarkoin määriteltyä prosessia rakentamisen avuksi. Esimerkiksi paperille tehtävät käyttöliittymäprototyypit voivat valmistua yhden päivän sisällä.

Lean Startup ja MVP -metodologioiden käyttö aiheuttaa kuitenkin joitain ristiriitoja ohjelmistokehityksen yhteydessä. MVP:hen on järkevää tehdä vain sen vaatimia ominaisuuksia. Suunnitelmia ei pitäisi yleistää, eikä toteuttaa mitään mahdollista seuraavaa projektia varten. Tämä on tärkeää startup-yritysten rajoitteellisten resurssien vuoksi. Kun tuotteiden suunta ja tarkoitukset ovat vakiintuneet, yleistykset ja suunnittelu tulevaisuutta varten voivat muuttua tärkeämmiksi. Toisaalta, lyhyen tähtäimen suunnittelu voi johtaa suureen tekniseen velkaan. Arkkitehtuuriin ja toteutuksen refaktoroinnin mahdollisuuteen toisaal-

ta taas tulisi panostaa joka tapauksessa. Muun muassa tuotteesta pitää pystyä helposti poistamaan ominaisuuksia. Lisäksi kehittäjien ammattitaito ja teknologioiden skaalautuvuus tulee ottaa huomioon, sillä ne auttavat pitkällä aikavälillä. [17] Nämä kuitenkin vaativat pitkän aikavälin suunnittelua.

Lisäksi startup yritykselle on tärkeää huomioida tuotantoprosessin kehittäminen tuotekehityksen ohella. Dande et al. huomauttavat, että ajoitus prosessin kehitykselle tulee olla oikea, sillä prosessin kehittämiseen vaaditaan yleensä resursseja yrityksen kokeneimmilta ja parhaimmilta työntekijöiltä. Itse tuotteen tulisi olla etusijalla, ja prosessin parannukset tulisi tehdä kasvuvalmisteluina sekä suunnitella ja ajoittaa tarkasti. [17]

MVP ei ole ainoa työkalu olettamusten validointiin Lean Startup -metodologiaa käytettäessä. Dande et al. mukaan siihen voidaan käyttää esimerkiksi paperiprototyyppejä tai jotain muita nopeita ja halpoja menetelmiä. Pääasiana on, että idea tulee ymmärretyksi. Ohjelmistostartup-yritysten tulisikin käyttää halpoja, jopa nollabudjetin menetelmiä varmistaakseen, että ihmiset todella tarvitsevat tuotetta tai palvelua. [17] Ne kumminkin vaativat muuta osaamista, kuin tietämystä arkkitehtuurisuunnittelusta, joten niitä ei tutkita lisää tässä työssä.

Lean Startup -menetelmää käytettäessä on myös mahdollista päätyä tilanteeseen, jossa validointi ei tuota tuotteelle suotuisia tuloksia. Tällöin on mahdollista joko "sinnitellä"engl. persevere idean kanssa ja tehdä uusia testejä tai tehdä pivot. Pivot on järjestelmällinen suunnanmuutos, jonka tarkoituksena on testata uutta perustavanlaatuista hypoteesia tuotteesta ja liiketoimintamallista. [49] Pivotin aikana siis suunnitellaan tuotetta uudestaan tai vaihdetaan liiketoimintasuunnitelmaa.

3.2 MVP-ohjelmiston tuottaminen

Historiallisesti ohjelmistoja toteutettiin vesiputousmallisissa projekteissa. Niissä tuotantoprosessi jaetaan erillisiin peräkkäisiin vaiheisiin ja ohjelmistoarkkitehtuuri suunnitellaan ennen itse ohjelmointityön alkamista. Tämä prosessimalli on kuitenkin todettu olevan monessa mielessä ongelmallinen ohjelmistotuotannossa. [18]

Vesiputousmalliin yksi suurimmista ongelmista on sen huono kyky reagoida muuttuviin vaatimusmäärittelyihin. Siinä ohjelma suunnitellaan kokonaan kehityksen varhaisessa vaiheessa. Tämä sitoo paljon resursseja ennen kuin konkreettista toimivaa ohjelmaa on saatu aikaiseksi. Lisäksi koko ohjelmisto perustuu yhteen suunnitelman, jonka muuttaminen on todennäköisesti työlästä, sillä muutoksiin ei suunnitteluvaiheessa olla varauduttu. Vaatimusten muuttaminen palauttaa prosessin sen alkuvaiheeseen.

Käytännössä vaatimukset perustuvat olettamuksille, jonka takia ne muuttuvat usein. Toisaalta, vesiputousmalli yhdistetään tiiviisti arkkitehtuurin suunnitteluun, sillä perinteisesti siinä on painotettu suunnitteluvaiheen tärkeyttä.

2000-luvun alusta lähtien niin sanotun ketterän kehityksen prosessimuodot ovat yleisty-

neet. Tällaisia prosesseja ovat esimerkiksi Scrum ja Extreme Programming (XP). Niissä perinteiseen malliin verrattuna suoritetaan kaikkia ohjelmistotuotannon vaiheita aina suunnittelusta testaukseen lyhyissä iteraatioissa. [5, 24, 46, 60] Tässä työssä käsitellään ketterää kehitystä erityisesti Scrum-prosessin näkökulmasta, sillä se on teollisuudessa käytössä laajasti. Ketterä kehitys toimii joka tapauksessa paremmin startup-yrityksen luonteen kanssa vesiputousmalliin verrattuna. [18] Tämä johtuu muun muassa siitä, että se mahdollistaa nopeamman palauteketjun ja vaatii vähemmän alkuresursseja. [18, 54] Tässä työssä keskitytäänkin arkkitehtuurisuunnitteluun ketterän kehityksen osana.

Työssä käsitellään sekä ketterää ohjelmistokehitystä että Lean Startup -prosessia. Nämä molemmat ovat iteratiivisia prosesseja, mutta niissä iteraatiot tarkoittavat eri asioita. [60] Lean Startup -metodologiassa iteraatioksi kutsutaan yhtä sykliä, jossa tuotetta rakennetaan, mitataan ja siitä opitaan. Toisaalta taas ketterässä kehityksessä yksi iteraatio tarkoittaa ajoitettuja toistuvia syklejä, jonka aikana tehdään ohjelmistokehitystä. Näitä kutsutaan tässä työssä jatkossa Scrum-menetelmässä käytetyllä termillä "sprintti". Yhden Lean Startup -iteraation aikana voi siis olla useita sprinttejä.

4 ARKKITEHTUURISUUNNITTELU LEAN STARTUP -KEHITYKSESSÄ

Muutama vuosi sitten, ketterän kehityksen leviämisen myötä, kiinnostus kattavaa ja syvällistä suunnittelua kohtaan vaikutti olevan laskussa. Useasti arkkitehtuurisuunnittelu kuuluu tällaisen suunnittelun piiriin. Kirjallisuudessa mainitaan, että monet pitävät sitä jopa haitallisena ketterän kehityksen kulttuurissa [54]. On kuitenkin huomattu, että ketterissä prosesseissa arkkitehtuurisuunnittelun pois jättäminen lisää niin sanottua teknistä velkaa; se tulee myöhemmin tehtäväksi "korkojen kanssa"[11]. Yksi selitys arkkitehtuurisuunnittelun laiminlyöntiin on, että siihen ei oteta ketterissä prosesseissa kantaa, ja toimivalle tuotteelle annetaan siinä enemmän painoarvoa kuin dokumentaatiolle. Usein tuotteen kehitykselle asetetun suuren painoarvon tulkitaan tarkoittavan sitä, että dokumentaatiolla ja suunnittelulla ei ole merkitystä. Toinen ongelma on, että muun muassa yleisesti käytetyssä Scrum-metodologiassa arkkitehtuurisuunnittelulle ei ole erillistä mainintaa, ja sille ei siinä erikseen varata resursseja. [56, 63]

Ketterässä kehityksessä on Kazman et al. mukaan kaksi kilpailevaa tekijää, jotka vaikuttavat arkkitehtuurisuunnitteluun:

- toiminnallisten vaatimusten toteuttaminen lyhyellä aikavälillä
- laatutavoitteiden täyttäminen lyhellä ja pitkällä aikavälillä [11].

Nämä kilpailevat myös Lean Startup- ja MVP-metodologioita käytettäessä. Toiminnallisten vaatimusten toteutus on perustana ketterässä kehityksessä. Lean Startup -metodologiassa käyttäjälle tulee tuottaa arvoa varhaisessa vaiheessa ja usein. Laatutavoitteiden täyttäminen taas on välttämätöntä, sillä sen laiminlyöminen johtaa projektin pysähtymiseen, koska muutoksien tekemisestä tulee liian monimutkaista.

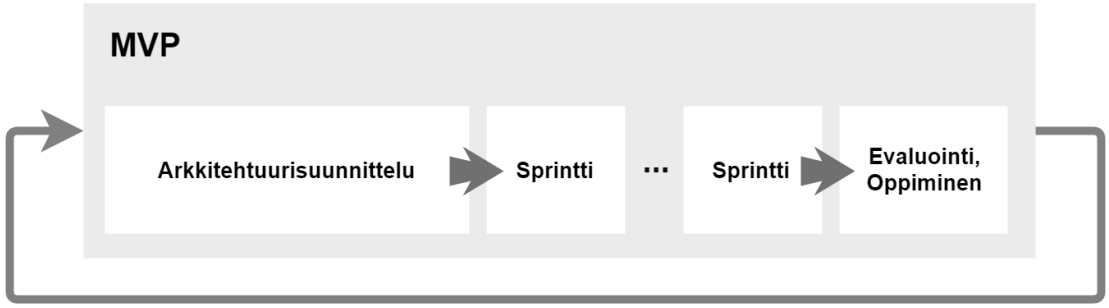
Projektin vaatima arkkitehtuurillinen suunnittelu riippuu Abrahamsson et al. [1] mukaan erittäin paljon projektin kontekstista. Kontekstiin kuuluu projektin ympäristö, esimerkiksi organisaatio tai sovelluksen toiminta-ala. Lisäksi siihen kuuluu projektin sisäiset tekijät kuten sen koko, liiketoimintamalli, työntekijöiden määrä ja fyysinen sijainti, järjestelmän ikä ja sen toimivuuden tärkeys. Myös monet muut tekijät voivat vaikuttaa tarvittavan arkkitehtuurisuunnittelun määrään. Ketterässä kehityksessä sitä tulisi tehdä vain juuri tarvittava määrä. Arkkitehtuurisuunnittelua tulisi tehdä mahdollisimman vähän, ja siihen liittyvän toiminnan tulisi olla tehokasta. Abrahamsson et al. mainitsevat, että suuret ja monimutkaiset projektit vaativat merkittävän määrän arkkitehtuurisuunnittelua, ja niissä ketteriä

menetelmiä tulisi muokata sopimaan kyseiseen kontekstiin. [1]

Seuraavissa luvuissa käsitellään arkkitehtuurin suunnittelua iteratiivisissa Lean Startup -metodologiaa seuraavissa prosesseissa.

4.1 Koko arkkitehtuurin suunnittelu etukäteen

Vaikka ketterää kehitystä harjoittavien keskuudessa on yleinen käsitys, ettei suunnittelua tulisi tehdä etukäteen, niin ketterät metodologiat eivät pakota suunnittelun ohittamiseen [64]. Arkkitehtuuri voidaankin suunnitella ketterissäkin projekteissa kokonaan etukäteen. Suunnitelmien yksityiskohdat jätetään tällöin toteutuksen aikaisiksi päätöksiksi, mutta suuremmat abstraktiot ja vuorovaikutukset päätetään ennen ketterän toteuttamisen alkua. Toisaalta Lean Startup -prosessimuotoa seurattaessa sitä ei voida kuitenkaan koskaan suunnitella kokonaan "lopulliselle"tuotteelle, sillä vain seuraavan MVP:n tavoitteet ja määrittely ovat selvillä. Tämä lähestymistapa MVP ja Scrum-metodologioihin integroituna on esitetty kuvassa 4.1.



Kuva 4.1. *Arkkitehtuurisuunnittelua seuraa sprintit, validointi ja oppiminen.*

Kuten kuvasta 4.1 voidaan havaita, arkkitehtuurisuunnittelua seuraavat ketterät toteutussprintit ja Lean Startup -metodologiaan liittyvät validointi sekä oppiminen.

Eloranta et al. huomasivat, että teollisuudessa tätä lähestymistapaa käytetään yleensä, kun tavoitteena on tehdä monimutkainen reaaliajassa toimiva sulautettu järjestelmä, eli laitteisto, jolla on jokin tietty tarkoitus. Näiden toteuttamiseen vaaditaan paljon koodia, ja niitä toteuttavilla yrityksillä on omat alustansa, joiden päälle toteutetaan tietynlaisia järjestelmiä. Eloranta et al. havaitsivat myös, että laitteiston läheisellä ohjelmoinnilla on suuri merkitys arkkitehtuurin kannalta ja sen vuoksi sitä tulee miettiä harkiten ennen ohjelmiston toteuttamista. Heidän tutkimuksessaan arkkitehtuurisuunnittelu kesti yrityksiltä puoli vuotta. [20] Tämä on kuitenkin startup-yrityksen kontekstissa aivan liian pitkä aika pelkkään suunnitteluvaiheeseen. Tuotteesta pitäisi saada käyttäjille nopeammin testattava versio, sillä muuten vielä muodostumattomista liiketoimintasuunnitelmista ja isoista alkuinvestoinneista johtuvat riskit kasvavat suuriksi tai tuotteen kehitys hidastuu liikaa.

Tätä lähestymistapaa käytettäessä toteutuksen aikana pyritään tekemään vain pieniä muutoksia arkkitehtuuriin ja niitä tekevät samat arkkitehdit, jotka aluksi suunnittelivat

sen. Ohjelmoijat eivät siis muokkaa arkkitehtuuria. [20, 43] Käytännössä arkkitehtuuri voi muuttua, kun tuotteelle asetetaan uusia vaatimuksia tai ympäröivistä tekijöistä saadaan uutta tietoa toteutus-sprinttien aikana. Alustan muutokset voivat myös vaikuttaa arkkitehtuuriin niin, että sitä joudutaan muokkaamaan. [20]

Eloranta et al. havaitsivat, että tässä lähestymistavassa on jotain hyötyjä ketterässäkin projektissa. Kun suunnittelu tehdään etukäteen, niin toiminnallisuudet saadaan pienennettyä sprinttimuotoiseen kokoon ja backlogin muodostamisesta tulee helpompaa. Muutoin ne muodostuisivat liian suuriksi toteutettavaksi yhdessä sprintissä. [20]

Arkkitehtuurin suunnittelu ennen toteutuksen aloittamista vaatii:

- paljon luottamusta tuotteen vaatimusten vakauteen
- riittävän määrään maksavia asiakkaita.

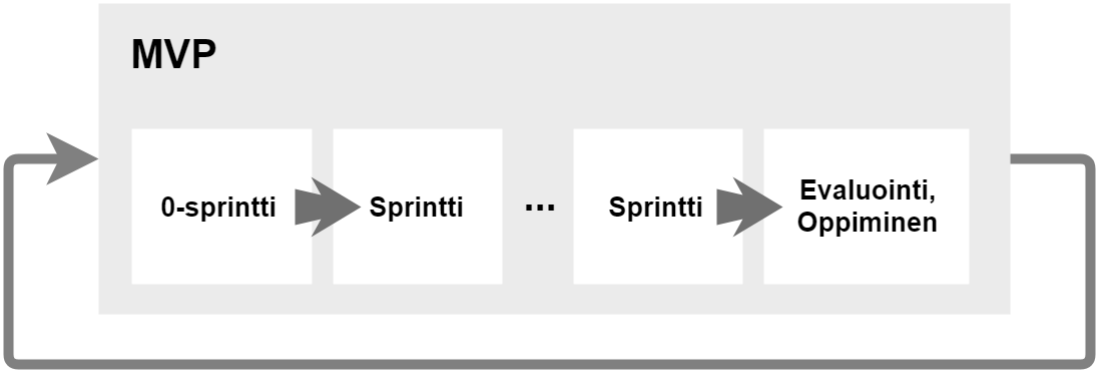
Suunnittelu etukäteen vaatisi paljon resursseja ja lisäisi epäonnistumisen riskejä, jos koko ohjelmisto toteutettaisiin kerralla. Lean Startup -prosessia seurattaessa ohjelmiston arkkitehtuuri voidaan kuitenkin suunnitella vain seuraavaa MVP:tä varten. Tällöin suunnittelu ei todellisuudessa tapahdu kokonaan etukäteen, vaan iteratiivisesti. Se tekee tästä lähestymistavasta mahdollisen vaihtoehdon Lean Startup -mallia käytettäessä. Prosessimuodoissa, joissa ohjelmistoa ei rakenneta ja testauteta pienemmissä osissa, se ei sitä välttämättä ole. Monen kuukauden pituiset suunnitteluvaiheet eivät kuitenkaan ole järkeviä startup-kontekstissa. Tämä ei ole toisaalta lähestymistavan puhtain muoto ja suunnitteluvaiheen lyhentyessä se alkaakin muistuttamaan yhä enemmän arkkitehtuurisprinttimenetelmää.

4.2 Arkkitehtuurisprintti (0-sprintti)

Ketterässä kehityksessä arkkitehtuurisuunnittelun toteuttamiseen voidaan varata ensimmäinen sprintti. Tässä lähestymistavassa toteutuksen aloittaminen suoraan ohjelmoimalla korvataankin arkkitehtuurin suunnittelulla. Tämä ei ole yhtä laaja ja yksityiskohtainen selvitys tai suunnitteluvaihe, kuin aiemmin esitellyssä lähestymistavassa. Se kuitenkin tarjoaa harkitun pohjan ohjelmiston rakenteelle. Suurin eroavaisuus ensimmäiseen menetelmään on, ettei arkkitehtuuria suunnittele erilliset suunnittelijat, vaan kehittäjät itse ja sen huomattavasti lyhyempi kesto, maksimissaan 4 viikkoa [20]. 0-sprintti ketterää MVP-metologiaa hyödyntäen on esitetty kuvassa 4.2.

Arkkitehtuurisprintit ovat Eloranta et al. tutkimuksen mukaan yhtä pitkiä kuin muutkin sprintit, eli 1-4 viikkoa. Niiden aikana järjestelmän pääsuunnittelijat suunnittelevat alustavan arkkitehtuurin kehittäjätyöryhmän kanssa. Alustava arkkitehtuuri ei usein kuitenkaan ole pysyvä ja se kehittyy ja muuttuu ohjelmiston kehityksen yhteydessä. [20]

Arkkitehtuuriin tulisi Abrahamsson et al. mukaan keskittyä tarpeeksi varhaisessa vaiheessa, sillä arkkitehtuuri sisältää joukon tärkeitä päätöksiä ohjelmiston rakenteesta ja sen toiminnasta. Niitä on vaikea poistaa, muuttaa tai refaktoroida, joka Abrahamsson et al.



Kuva 4.2. *Arkkitehtuurisprinttiä käytettäessä ensimmäinen sprintti on omistettu arkkitehtuurisuunnittelulle, tämän jälkeen kehitys jatkuu iteroiden.*

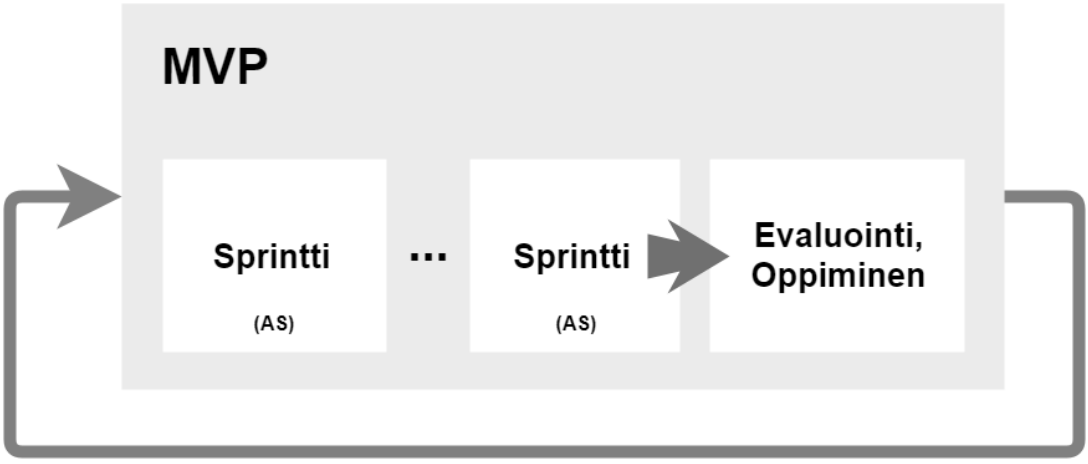
mukaan tarkoittaa sitä, että arkkitehtuurisuunnittelu tulisi ottaa käsittelyyn ensimmäisissä ketterissä iteraatioissa. He kehottavat käyttämään 0-sprintti lähestymistapaa ketterissä projekteissa. [1] Kevyt tutkinta aluksi vähentää riskiä sille, että järjestelmästä kehittyy ajan saatossa liian monimutkainen. Uudelle MVP:lle voidaan suunnitella arkkitehtuuria aina edellisen version päälle. Arkkitehtuurin kokonaiskuva voi kuitenkin pysyä hallinnassa paremmin kuin ilman erillistä suunnittelua, sillä MVP-iteraatioiden välillä pysähdytään pohtimaan, onko se vielä käyttökelpoinen ja tukeeko se seuraavaa MVP:tä. Tällöin esimerkiksi pivot-kohdassa arkkitehtuuria voidaan analysoida ja refaktoroida systemaattisesti ja tarpeen mukaisesti. Toisaalta se ei vaadi suuria alkuinvestointeja, eikä muutoksia arkkitehtuuriin ei tarvitse pelätä huolellisesti suunniteltuun, mutta pois heitettyyn ohjelmistoon hukattujen resurssien takia.

4.3 Arkkitehtuurin suunnittelu sprintin aikana

Kolmas vaihtoehto on suunnitella ohjelmistoarkkitehtuuri toteutuksen yhteydessä. Tässä kehittäjät suunnittelevat itse arkkitehtuurin. Ensimmäisten sprinttien aikana on tarkoitus sekä suunnitella arkkitehtuuria että tuottaa julkaistavia ominaisuuksia. Jokaisen sprintin aikana arkkitehtuuri, joka liittyy siinä tehtäviin ominaisuuksiin, hiotaan ja aikaisemmin toteutettuja ominaisuuksia refaktoroidaan, mikäli se on tarpeellista. Arkkitehtuurisuunnittelu sprinttien sisällä MVP:tä rakennettaessa esitetty kuvassa 4.3.

Eloranta et al. mukaan yritykset, jotka suunnittelevat arkkitehtuuria ainoastaan sprinttien sisällä koostuvat juuri sen alan kokeneista kehittäjistä. Ongelmia voi syntyä, jos kehittäjät eivät ole kokeneita omalla osa-alueellaan, mikä voi johtaa koko projektin epäonnistumiseen. Tällöin arkkitehtuuri ei vastaa vaatimuksia ja sitä pitää refaktoroida uudelleen ja uudelleen. Eloranta et al. huomasivat, että lähestymistapa lisää joka tapauksessa refaktoroinnin tarvetta, sillä suunnitelmia tehdessä ei oteta huomioon kokonaiskuvaa. [20]

Sturtevant huomauttaa, että ketterässä prosessissa ei-ketterien arkkitehtuurien käyttö johtaa arvottoman ohjelmiston tuottamiseen. Arkkitehtuurista muodostuu hänen mukaan-



Kuva 4.3. *Arkkitehtuurisuunnittelu voidaan jättää yksittäisiin sprintteihin.*

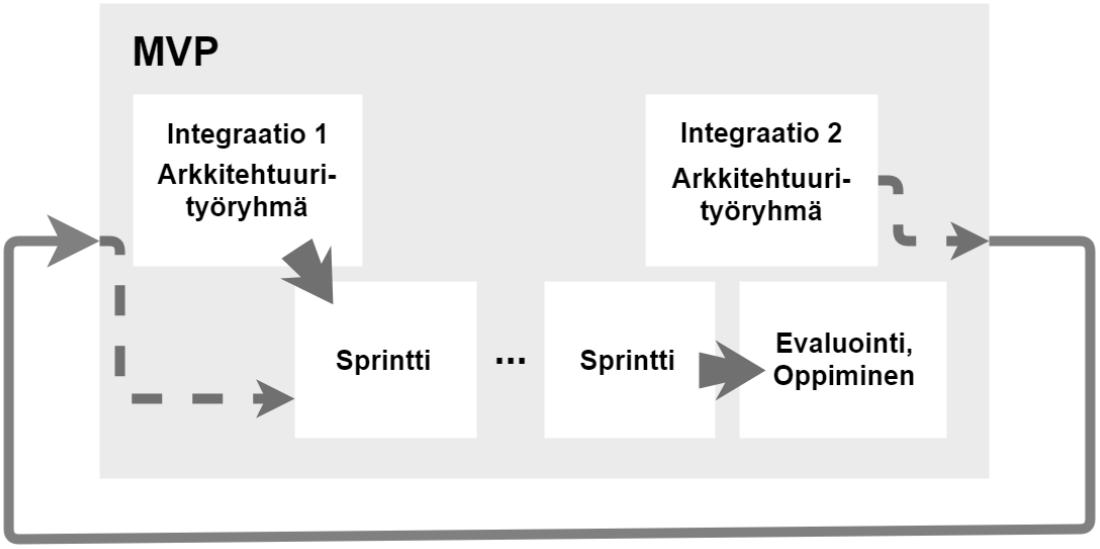
sa suurin este ohjelmistokehitykselle. Ketterä kehitys ja ketterä arkkitehtuuri tulisi olla tasapainossa. [56]

Lean-lähestymistapa yhdistettynä ohjelmistokehitykseen voi joissakin tapauksissa johtaa ongelmiin. Monet ketterät menetelmät, kuten suunnittelupokeri, backlog ja niin edelleen, eivät torju ohjelmiston rakenteellisia ongelmia. Jos ohjelmoijat eivät helposti pysty ymmärtämään tai muokkaamaan koodia, he eivät pysty vastaamaan validoinnissa vahvistettujen tai hylättyjen olettamuksien aiheuttamiin muutoksiin. Tekninen velka johtaa asiakkaille tuotettavan arvon tyrehtymiseen. [56] Tämän vuoksi pelkkä sprintin aikana tehtävä suunnittelu voi niin ikään johtaa ongelmiin startup-yrityksessä. Vaikka tuote saataisiin nopeasti markkinoille, voi MVP-iteraatioiden tekeminen jossain vaiheessa muuttua niin työlääksi, että yritys ei pysy enää kilpailussa mukana tai se ei lopulta pysty täyttämään asiakkaiden vaatimuksia ja toiveita.

4.4 Erillinen arkkitehtuurityöryhmä

Viimeinen yleisesti käytetty vaihtoehto [20] on erillisen arkkitehtuurityöryhmän käyttäminen. He suunnittelevat arkkitehtuuria samaan aikaan kun kehittäjät toteuttavat ohjelmistoa. Nämä molemmat työryhmät toimivat toisistaan kokonaan erillään, mutta myös sekaryhmät voivat olla mahdollisia. Arkkitehtuurityöryhmään voi kuulua kehittäjätyöryhmän jäseniä.

Arkkitehtuurityöryhmän jäsenille asetetaan merkkipaaluja. Merkkipaalut ovat ajankohtia, jolloin tietyt arkkitehtuurin osat tulee olla valmiina. Jokainen ohjelmajulkaisu sisältää yleensä yhden merkkipaalun ja tällöin seuraavan julkaisun arkkitehtuuri tulee olla selvillä. [20] Lean Startup ja MVP -metodologioita käytettäessä tämä voisi tarkoittaa seuraavan MVP:n arkkitehtuuria. Kuvassa 4.4 on esitetty erillinen arkkitehtuurityöryhmä MVP-metodologian osana.



Kuva 4.4. *Arkkitehtuurityöryhmän integraatioon MVP-kehityksessä on monia vaihoehtoja.*

Kun merkkipaalu on saavutettu, arkkitehtuurityöryhmä esittelee suunnitelmansa, joita voi olla useita. Kehittäjätyöryhmästä päättävät henkilöt ja mahdollisesti asiakkaat arvioivat suunnitelmaehdotukset ja päättävät millä suunnitelmalla jatketaan kehittämistä. [20] Kuten kuvasta 4.4 voidaan huomata, niin merkkipaaluille ei kuitenkaan ole luontaista paikkaa MVP-metodologiaa käytettäessä. Se voidaan asettaa esimerkiksi joko MVP:n sisäisesti tai ennen seuraavaa MVP-iteraation tekemistä.

Eloranta et al. tutkimukseen osallistuneet yritykset yleensä käyttivät tätä menetelmää, jotta he eivät sotkisi Scrum-prosessia millään tapaa. [20] Startup-yrityksessä ei kuitenkaan välttämättä ole resursseja erilliseen arkkitehtuuritiimiin ja ketterien menetelmien järjestelmällinen seuraaminen ei ole olennaista. Tämän lisäksi sen integrointitapa täytyy erikseen päättää, kun arkkitehtuurityöryhmää käytetään Lean Startup ja MVP-metologian kanssa. Arkkitehtuurityöryhmä voi tehdä suunnittelua seuraavaa MVP:tä varten. Tällöin toisaalta syntyy mahdollisuus sille, että heidän suunnitelmansa menevät hukkaan, jos validoinnin perusteella päätetään tehdä pivot.

Näiden syiden vuoksi tämä lähestymistapa ei vaikuttaisi sellaisenaan sopivalta aloittelevalle startup-yritykselle.

4.5 Joukkoistaminen

Joukkoistaminen engl. crowdsourcing on suhteellisen uusi ilmiö ohjelmistokehityksessä. Termi esiteltiin alun perin vuonna 2006 kuvaamaan hajautettua ongelmanratkaisumallia, jossa internetin välityksellä ihmiset ratkovat ongelmia. Sittemmin joukkoistamiseen liittyen on tehty paljon tutkimusta ja sitä on harjoitettu ohjelmistokehityksen yhteydessä. [45]

Joukkoistettussa ohjelmistokehityksessä engl. crowdsourced software engineering yritys

hyödyntää resursseja verkossa erinäisiin ohjelmistokehityksen tehtäviin, kuten ohjelmointiin, testaukseen tai suunnitteluun. Joukkoistamisen väitetään vähentävän kehitysaikaa, sillä se lisää rinnakkaista toimintaa. Se voi lisäksi vähentää kustannuksia ja virheitä ohjelmistossa. [39, 55] Joukkoistamiseen ohjelmistokehityksessä on alustoja kuten TopCoder ja Upwork. [29, 45] Tämän työn puitteissa mielenkiintoisinta on joukkoistamisen käyttäminen suunnitteluvaiheessa, sillä työ käsittelee ohjelmistoarkkitehtuurin suunnittelua.

Mao et al. mukaan, kun joukkoistetaan suunnitteluvaihetta, tulee ensin tehdä muutamia päätöksiä esimerkiksi

- mitä joukkoistetaan
- kuinka pieniä ovat tehtävät
- mitä taitoja tehtävän suorittaminen vaatii sen tekijöiltä
- kuinka tekijöitä kannustetaan suunnitteluun
- miten ja kuinka suurta joukkoa tekijöitä halutaan [45].

Näiden päätösten suunnittelu voi vaatia oman aikansa ja huonosti suunniteltu joukkoistettu tehtävä voi mennä hukkaan.

Joissakin edellä mainituista alustoista on ominaisuuksia ohjelmistoarkkitehtuurin joukkoistamiseen. Esimerkiksi TopCoder on yksi yleisesti käytetyistä alustoista, jotka tukevat sen suunnittelua [45]. Niissä on kuitenkin joitakin rajoitteita, kun yhdistetään ja kehitetään useiden suunnittelijoiden tekemiä suunnitelmia. Useampien suunnitelmien yhdistely on kuitenkin hyödyllistä [40]. [45] Joukkoistaminen itsessään vaatii suhteellisen paljon resursseja sen suunnitteluun, valittujen tuotosten palkitsemiseen ja tulosten tarkastamiseen. Sen käyttämiseen vaadittujen resurssien vuoksi se ei välttämättä ole mahdollinen ohjelmistostartup-yrityksen alkuvaiheissa. Ehkä ohjelmistolle asetettujen vaatimusten vakiintuessa tai erittäin suurta uutta ominaisuutta tai muutosta tehtäessä joukkoistamista voitaisiin hyödyntää järkevästi.

Taulukko 4.1 esittää ominaisuuksia eri lähestymistavoille arkkitehtuurisuunnitteluun MVP-prosessin osana.

Taulukko 4.1. *Lähestymistavat eroavat toisistaan monella tavalla.*

Lähestymistapa	Suunnittelun kesto	Kattavuus	Yhteensopivuus MVP-prosessiin
Suunnittelu etukäteen	Pitkä	Erittäin kattava	Huono
0-sprintti	Keskipitkä	Kattava	Sopivin
Sprintin aikana	Ei erillistä suunnitteluvaihetta	Vähäinen	Keskinkertainen
Arkkitehtuuri-työryhmä	Pitkä	Erittäin kattava	Huono
Joukkoistaminen	Keskipitkä/Pitkä	Erittäin kattava	Keskinkertainen

Kuten taulukosta 4.1 voidaan huomata, 0-sprintti-lähestymistapa sopii ominaisuuksiltaan parhaiten MVP-prosessin kanssa käytettäväksi kirjallisuustutkimuksen perusteella. Se on kompromissi kattavan suunnittelun ja resurssien kulutuksen välillä.

5 YLEISET ARKKITEHTUURISUUNNITTELUN JA TOTEUTUKSEN KONSEPTIT

Ohjelmistoarkkitehtuurin suunnittelu vaatii resursseja ja osaamista usealta eri osa-alueilta. Startup-yrityksestä ei näitä välttämättä löydy. Seuraavissa luvuissa käydään läpi eri lähestymistapoja arkkitehtuurisuunnitteluun ja niiden soveltuvuutta ohjelmistokehitykseen startup-yrityksen rajoitteiden puitteissa. Tässä käsiteltävät menetelmät ja työkalut koskevat Tang et al. määritelmän vaiheita:

- 2 arkkitehtuurillisten päätösten suunnittelu ja kuvaus
- 4 arkkitehtuurin toteutus.

Jotkin niistä vaikuttavat molemmissa vaiheissa ja toiset taas toimivat pelkästään suunnittelun apuna. Konseptit, jotka auttavat molemmissa esitellään ensin. Osan lopussa tehdään yhteenveto eri konsepteista.

5.1 Ohjelmarungot

Ohjelmarungot tarkoittavat ohjelman korkean tason rakentamista ja oikean toiminnallisuuden korvaamista tynkätoiminnoilla, kuten tyhjillä funktioilla. Näitä tynkätoimintoja voidaan sitten muuttaa oikeaksi toiminnallisuudeksi asteittain iteraatioiden ja sprinttien aikana. Ohjelmarungon tekeminen auttaa suunnittelua ylhäältä alaspäin. Siinä osittain toimivassa järjestelmässä on kokonainen korkean tason rakenne suunniteltuna ja ohjelmoituna. Ohjelmarunko voi kehittyä myös prototyyppien luonnin seurauksena, jolloin prototyypin rakenteen päälle tehdään lopullinen ohjelmisto [7].

Ennen ohjelman laajamittaista toteuttamista, sen tekninen perusta täytyy päättää. Tekniseen perustaan kuuluu tekniset palvelut, joiden päälle ohjelmistoa rakennetaan ja ohjelmistokehykset toiminnallisten rakennusosasten tekemiseen. Ohjelman toimintalogiikan tulisi olla mahdollisimman itsenäinen näistä rakennuspalikoista. Teknisen perustan tekeminen ei ole mahdollista ilman tietoa toiminnallisten rakennusosasten tarpeista. Tämä on yksi syy ohjelmarungon rakentamiselle, sillä ohjelmarunko osoittaa selkeästi, kuinka järjestelmä voidaan tehdä, jotta se mukautuu arkkitehtuuriin. [62]

Ohjelmarungon rakenne vastaa ohjelmiston lopullista arkkitehtuuria. Toisaalta yksittäiset komponentit eivät tarjoa kokonaista toteutusta sen toiminnoille. Ne yleensä sisältävät toi-

minnallisuudet jollekin tietylle käyttötapaukselle. Mitään muita toimintoja ei ole vielä toteutettu tai ne on määritelty tilapäisillä ratkaisuilla. Ohjelmarungot tarjoavat ajettavan pohjan projektin mahdollisimman varhaisessa vaiheessa. Järjestelmä sisältää kaikki oleelliset komponentit, vaikka niitä ei olisikaan toteutettu kokonaan. Järjestelmän toiminnallisuus onkin sen toissijainen tarkoitus. [62]

Yksittäisiä komponentteja voidaan kehittää pidemmälle, kun ohjelmarunko on tehty koko järjestelmälle. Kaiken aikaa on käytössä ajettava ohjelma, joka toimii jollain tavalla. Lisäksi kehityksen varhaisissa vaiheissa voidaan keskittyä järjestelmän kriittisiin tai monimutkaisiin osiin ja riskejä voidaan ehkäistä. Runko mahdollistaa myös sen, että kehittäjät voivat keskittyä toimintalogiikan kehittämiseen järjestelmän rakenteen sijaan, sillä se on jo olemassa. [7, 62]

Ohjelmarunkoja voidaan käyttää myös sen varmistamiseksi, että suunniteltu arkkitehtuuri varmasti toteutuu. Vaikka arkkitehtuuri olisi suunniteltu tietynlaiseksi, kehittäjät eivät välttämättä osaa tulkita sitä oikein tai jostakin muusta syystä se toteutetaan suunnitelmista eroavalla tavalla. Jos ohjelmaan on runko jo toteutettu, kehittäjien ei tarvitse huolehtia arkkitehtuurista ja he voivat toteuttaa ohjelmistoa rungon asettamissa rajoissa. [62]

5.2 Ohjelmistokehykset

Ohjelmistokehys engl. software framework on rakenne, joka ohjaa millaisia sovelluksia voidaan tai tulisi rakentaa ja kuinka ne sovitetaan toisiinsa [19, 53]. Sen tarkoituksena on yksinkertaistaa kehitysympäristöä, mikä auttaa ohjelmoijia keskittymään vaatimusten toteuttamiseen. Se tarjoaa kehittäjille useasti tarvittavia työkaluja tai funktioita, joka vähentää toistuvia tehtäviä. Kehykset sisältävät ohjelmia, määrittävät rajapintoja tai tarjoavat työkaluja kehyksen käyttöä varten. Kehyksen tarkoitus voi olla määrittää joukko funktioita järjestelmässä ja kuinka ne kytkeytyvät toisiinsa, käyttöjärjestelmän eri tasoja, ohjelmiston alijärjestelmän tasoja ja niin edelleen. Ohjelmistokehys voi olla konkreettinen tai konseptuaalinen alusta, joka toteuttaa yleiskäyttöisiä toiminnallisuuksia, joita ohjelmoija voi erikoistaa tai korvata. [4, 19]

Ohjelmistokehyksellä ja ohjelmarungolla on monia yhtenäisyyksiä. Suurin ero niillä on kuitenkin se, että ohjelmarungon tarkoitus on määrittää ohjelmiston rakenne ja kuinka komponentit ovat vuorovaikutuksessa keskenään. Kehyksien tarkoitus on sen sijaan tarjota hyödyllisiä tai muuten usein käytettäviä yleisiä toimintoja, mutta ne voivat myös tarjota näkemyksen rakenteelle. Ohjelmistokehykset voivat usein olla myös kolmannen osapuolen tekemiä ja ne ovat yleiskäyttöisiä, kun ohjelmarungot taas ovat yrityksen sisäisiä ja tarkoitettuja yhden projektin käyttöön.

Ohjelmistokehykset voidaan luokitella kahteen eri luokkaan: mustiin ja valkoisiin laatikoihin. Niiden pääasiallisena erona on, kuinka paljon ohjelmoijan tulee tietää ja määrittää kehyksen sisäistä toimintaa. Mustassa laatikossa kehyksen käyttäjälle avataan vain komponenttien ulkoiset rajapinnat, eikä hänen tarvitse välittää sen sisäisestä toiminnas-

ta. Valkoisessa laatikossa sen sijaan ohjelmoijan tulee tietää ja ymmärtää, kuinka kehys toimii sisäisesti. Musta-laatikko-kehykset ovat pääasiassa helpompi ymmärtää ja oppia, kuin valko-laatikko-kehykset. Toisaalta ne eivät monesti ole yhtä joustavia. [33]

Ohjelmistokehykset voivat olla suuria kokonaisuuksia, jotka määrittelevät kokonaan ohjelmiston rakenteen. Tällöin kehyksen käyttäminen määrittää ohjelmiston arkkitehtuurin melkein kokonaan. Mikäli kehys sisältää oletuksia tai määrää arkkitehtuuria, sen valinta voi osoittautua erittäin tärkeäksi. Ohjelmistokehyksen määräämien ratkaisujen laatu määrää suoraan sillä tuotettavan ohjelmiston laatua. [15] Hyvin suunniteltu ja tilanteeseen sopiva kehys voi huomattavasti nopeuttaa ohjelmistokehitystä ja ohjelmiston laatua, kun taas huonosti tilanteeseen sopiva kehys voi osoittautua käyttökelvottomaksi. Kehykset voivat toisaalta myös keskittyä helpottamaan jonkun tietyn osa-alueen toteuttamista, jolloin se huomioidaan yhtenä komponenttina arkkitehtuurissa. Tällöinkin sen yhteensopivuutta tulisi tutkia esimerkiksi arkkitehtuuriprototyypein.

Kehykset ovat Vogel et al. mukaan hyviä tuottamaan toteutuksia, jotka vastaavat arkkitehtuurisuunnitelmia. Kehykset antavat niiden käyttäjille toteutusohjeiston rajapinnoin ja abstraktein toteutuksin. Rajapinnat ja abstraktit toteutukset vastaavat arkkitehtuuria, joka on tehty ja niinpä ne varmistavat, että toteutettu ohjelma vastaa suunnitelmaa. [62]

Kehykset voivat laajuudestaan riippuen olla hyvä työkalu arkkitehtuurisuunnittelun yhteydessä. [15] Se toimii jokseenkin samalla tavalla, kuin referenssiarkkitehtuuri ja runko yhdistettynä. Se voi määrittää jonkin tietyn rakenteen, jonka puitteisiin ohjelmisto rakentuu sekä konventioita, joita koodin tulee seurata. Niiden käyttö voikin vähentää arkkitehtuurisuunnittelun määrää huomattavasti.

Kehysten käyttöön startup ympäristössä ei ole suoria esteitä. Hyvin valitun kehyksen käyttö ei yleensä lisää merkittävästi alkutyömäärää tai suunnittelua, varsinkin jos ne ovat kolmannen osapuolen tekemiä ja ne integroidaan yrityksen omaan tuotteeseen. Jos kolmannen osapuolen kehyksen valitsee huonosti, sen sovittaminen järjestelmään voi osoittautua vaikeaksi. Oman kehyksen luominen startup-kontekstissa sen sijaan vie liikaa resursseja, sillä se vaatii paljon suunnittelua sekä yleistävien ratkaisujen etsimistä. Sopivien kehysten löytäminen ja valkoisen laatikon ohjelmistokehyksien opettelu voi niin ikään viedä aikaa. Sopiva kolmannen osapuolen kehys kuitenkin yksinkertaistaa ja nopeuttaa tehtäviä sekä suunnittelu- että toteutusvaiheessa. Tämän vuoksi niiden käyttö on joka tapauksessa suotavaa.

5.3 Arkkitehtuuriprototyypit

Arkkitehtuurillisten prototyppien luonti tarkoittaa suoritettavan koodin tekemistä ja käyttöä niihin sidosryhmiä kiinnostaviin aiheisiin, jotka liittyvät järjestelmän arkkitehtuuriin ja laatuvaatimuksiin [8].

Bardram et al. jakavat arkkitehtuurilliset prototyypit niiden tarkoitusperän mukaan. Tyyp-

pejä on 3:

1. tutkivat prototyypit engl. exploratory prototype
2. kokeilevat prototyypit engl. experimental prototype
3. kehittävät prototyypit engl. evolutionary prototype [7].

Tutkivia prototyyppejä tehdään, jotta voidaan selvittää arkkitehtuurin suunnitteluavaruus. Tällöin tehdään useita vaihtoehtoisia prototyyppejä, jotka analysoidaan ja ajetaan, jotta saadaan aikaiseksi optimaalinen ratkaisu ongelmaan. Kokeilevien prototyyppien tarkoitus on arvioida jokin tietty arkkitehtuurillinen päätös. Tällöin toteutetaan yleensä vain yksi arvioitava prototyyppi. Kehittäviä prototyyppejä valmistetaan usean prototyypin sarjana, jossa jokaisen prototyypin pohjalta rakennetaan uusi prototyyppi. Usein tällaisten prototyyppien valmistus johtaa ohjelmistorungon muodostumiseen. [7]

Prototyyppejä voidaan kehittää moniin tarkoituksiin. Arkkitehtuurillisen prototyypin tulee täyttää 5 eri vaatimusta. [7]

1. Sen tulee opettaa arkkitehtuurillisten päätösten vaikutuksista järjestelmään.
2. Sen tulee ottaa huomioon laatuvaatimukset. Useasti tarkoituksena on selvittää mittaamalla, kuinka ratkaisut vaikuttavat ohjelmiston laatuun.
3. Se ei sisällä juuri hyödyllistä toiminnallisuutta liiketoiminnalle tai loppukäyttäjälle.
4. Sen rakentamisen tarkoituksena on ehkäistä tai arvioida arkkitehtuurillisen päätöksen aiheuttamia riskejä.
5. Se auttaa välittämään arkkitehtuurillista tietämystä. Käytännössä sitä voidaan esimerkiksi jatkossa käyttää ohjelmistoarkkitehtuurin opetustyökaluna.

Prototyyppien tekeminen mahdollistaa tärkeiden arkkitehtuurillisten päätösten tasokkaamman tarkastelun. Ennen kokeilua ennalta arvaamattomat osa-alueet voivat olla vaikeita tai mahdottomia ennustaa. Tämän vuoksi prototyyppien tekeminen voi olla järkevää suunnitelmien arvioimiseksi muiden menetelmien ohella. [16]

Arkkitehtuurillisia prototyyppejä käytetään teollisuudessa työkaluna suunnitelmien tekemiseen ja arvioimiseen. Kokeneetkin arkkitehdit luottavat suoritettavaan ohjelmakoodiin enemmän kuin keskusteluun, kuten esimerkiksi aivoriiheen, erinäisiin katselmuksiin tai loogisiin argumentteihin. Christensen et al. pohdinnan mukaan prototyyppejä ei välttämättä hyödynnetä aina kun siihen olisi järkevä mahdollisuus. Esimerkiksi toiminnallisuuksien tekeminen prototyyppiin saattaa saada sen vaikuttamaan kuluja lisäävältä toimenpiteeltä, vaikka sillä voitaisiin testata arkkitehtuuriin liittyviä ongelmakohtia. [16]

Eloranta et al. tutkimuksen mukaan arkkitehtuuriprototyyppejä käytetään erityisesti projekteissa, joissa arkkitehtuuri suunnitellaan etukäteen ennen ohjelmaiteraatioiden tekemistä. Tällöin yleensä kokeillaan erilaisia lähestymistapoja ja teknologioita, joiden päälle ohjelmiston arkkitehtuuria rakennetaan. Toisaalta, jos arkkitehdit ovat myös ohjelmistokehittäjiä, alkaa projektimuoto muistuttaa enemmän sprintin sisäisen arkkitehtuurisuunnittelun mallia. [20] Startup-ympäristössä erilaisten prototyyppien tekeminen vaikuttaisi

olevan joissakin tilanteissa järkevä ratkaisu. Jos käyttöön valitaan uusia teknologioita, joita kehittäjät eivät ole aiemmin käyttäneet, voi prototyyppi selventää monia teknologiaan liittyviä seikkoja. Prototyypistä voidaan tehdä tarpeeksi karkea yksityiskohtien mallintamisen sijaan, jotta sen tekeminen ei vaadi suuria resursseja.

5.4 Arkkitehtuurilliset kehykset

Tässä ja seuraavissa aliluvuissa käsitellään pelkästään suunnittelun aikaisia työkaluja.

Suunnittelumallit kuvaavat tavan, jolla voidaan ratkaista jokin yleinen ohjelmistorakenteen ongelma. Suunnittelumallit voivat olla riippuvaisia joistain teknologioista tai ohjelmistotuotannon alasta. Esimerkiksi Microsoftin kehittelemät toiminta- ja suunnittelumallit ovat teknologiariippuvaisia kun taas yleisesti olio-ohjelmoinnissa käytössä olevat Gang of Four -mallit eivät riipu mistään tietystä teknologiasta [6, 30].

Arkkitehtuurillinen kehys engl. architectural framework on suunnittelumallia laajempi kokonaisuus. Siihen sisältyy joukko arkkitehtuurilliseen kuvaukseen liittyviä yleisiä käytäntöjä, jotka ovat vakiintuneet jollain tietyllä alalla tai sidosryhmäyhteisössä. Se ottaa huomioon jotkin yhteiset kiinnostuksen kohteet ja niitä rajaavat lähestymiskulmat. Esimerkkejä arkkitehtuurillisista kehyksistä ovat TOGAF (The Open Group Architectural Framework), MoDAF ja the Zachman Framework. [6, 48]

Arkkitehtuurillinen kehys on siis kokonaisuus, johon kuuluu jokin tietty tapa kuvata ja käsitellä ohjelmistoarkkitehtuuria ja ohjelmistoa itseään. Muun muassa TOGAF määrää useita eri "näkymiä", joissa ohjelmistoa tulee kuvata [48]. Näissä on kuitenkin välillä erittäin paljon erilaisia näkymiä, jotka tulisi toteuttaa ohjelmistoa varten. TOGAF ehdottaakin 7 eri näkymän tekemistä järjestelmälle [48, 61]. MVP-metodologiaa käytettäessä näin tarkka kuvaus järjestelmästä voi olla liiallista. Mikäli arkkitehtuurillinen kehys pakottaa raskaaseen prosessiin, niin se ei sovi MVP:n tuottamiseen.

Lean Startup -prosessiin kantaa ottavia arkkitehtuurillisia kehyksiä on haastava löytää [57]. Tämän työn puitteissa ei löytynyt kehystä, joka olisi keskittynyt arkkitehtuurin suunnitteluprosessiin Lean Startup -ympäristössä. Monesti kehykset kuvaavat isojen ja vakiintuneiden yritysten ja organisaatioiden tarpeita, eikä pienenten startup-yritysten rajoitteita oteta niissä huomioon [48, 57, 61]. Tämä näkyy muun muassa edellä mainitussa arkkitehtuurin tarkassa dokumentoinnissa ja suurille organisaatioille sopivien prosessien ehdottamisena. Arkkitehtuurillisten kehysten valinnassa ja yksilöinnissä tulisi siis pohtia eri ohjeiden tarkoituksia ja sovellettavuutta juuri kyseisessä tilanteessa.

5.5 Referenssiarkkitehtuurit

Referenssiarkkitehtuurit ovat toimivaksi todettuja arkkitehtuureja tai niiden osia, joita voidaan käyttää uudelleen. Referenssiarkkitehtuuri eroaa kehyksestä siinä, että se tarjoaa jonkin toimivan arkkitehtuurin osan, kun kehys taas kuvaa jonkin arkkitehtuurisuunnitteluprosessin. Referenssiarkkitehtuuri ei siis ota suunnitteluprosessiin kantaa. Ne ovat yleensä alakohtaisia ja niitä voidaan joutua muokkaamaan kyseiseen ohjelmistoon sopivaksi esimerkiksi jättämällä joitain osia pois tai muokkaamalla niitä. Ne voivat myös olla osittain tarkoituksella puutteellisia, jolloin niitä täydennetään kyseistä ohjelmistoa varten.

Teollisuudessa referenssiarkkitehtuureja käytetään konkreettisten arkkitehtuurien suunnitteluun. Niitä ei ole tarkoitettu kovin erikoistuneisiin sovelluksiin, mutta ne tarjoavat yleisiä suunnitelmia ja suunnittelupäätöksiä. Ohjelmistokehittäjät voivat käyttää niitä suunnitelmiensa pohjana ja muokata niitä omaan ohjelmistoonsa sopivaksi. Niinpä ne tarjoavat osittain tai kokonaan toteutettuja suunnittelutuotoksia, jotka on tarkoitettu tiettyihin käyttötilanteisiin. Referenssiarkkitehtuureja voidaankin pitää uudelleenkäytettävänä arkkitehtuurillisena tietona, joka pitää muun muassa sisällään:

- suunnitteluohjeita
- hyviä käytäntöjä
- standardeja
- arkkitehtuurillisia tyylejä.

Lisäksi ne tukevat standardien muodostumista ja järjestelmien yhteensopivuutta, sillä ne voivat perustua samalle pohjalle. [23]

Galster et al. listaavat referenssiarkkitehtuureille kolme hyötyä. Ne esitellään seuraavaksi.

Referenssiarkkitehtuurit yksinkertaistavat suunnittelutehtäviä. Jotkin arkkitehtuurin suunnitteluun liittyvät tehtävät ovat helpompia, jos niissä käytetään apuna referenssiarkkitehtuureja. Esimerkiksi suunnittelun aloitus ja jatkaminen ovat yksinkertaisempia, jos niitä lähdetään tekemään referenssiarkkitehtuurin päälle. Ne voivat mahdollistaa ketterässä ympäristössä eri vaihtoehtojen kokeilun referenssiarkkitehtuurin rajoissa. [23]

Referenssiarkkitehtuurit lisäävät vapautta kehitystyöryhmissä. Galseter at al. huomasivat, että kehittäjät pystyvät työskentelemään usean eri projektin kanssa samanaikaisesti tai vaihtamaan nopeasti niiden välillä, jos ne käyttävät samaa referenssiarkkitehtuuria. Lisäksi se tukee kommunikaatiota työryhmän sisällä, koska heillä on yhteisymmärrys yleisistä arkkitehtuurillisista ideoista. Tämä vuorostaan tukee monialaisia työryhmiä, joita suositaan ketterässä kehityksessä. [23]

Referenssiarkkitehtuurit tuovat huomiota ohjelmiston arkkitehtuuriin. Ketterissä menetelmissä, joissa ohjelmiston rakennetta ei aseteta kovin tärkeäksi, referenssiarkkitehtuurien käyttö lisää huomiota ohjelmiston arkkitehtuuriin. Se tuo esiin laatutavoitteiden tärkeyttä ketteriin prosesseihin aiheuttamatta merkittäviä lisäkustannuksia. Lisäksi se vähentää

liiallista dokumentointia. [23]

Referenssiarkkitehtuureihin liittyy kuitenkin joitain rajoitteita. *Kehittäjien henkilökohtaiset mieltymykset* voivat olla referenssiarkkitehtuureja vastaan. Galster et al. tutkimuksessa osa kehittäjistä ei pitänyt niiden käyttämisestä sillä ne rajoittavat luovuutta ja vapautta päätöksiä tehtäessä. [23]

Referenssiarkkitehtuurit vaativat ylläpitoa. Referenssiarkkitehtuurien ylläpito voi viedä liikaa resursseja etenkin, jos ohjelmistojulkaisun tekeminen on tärkeää. Sen päivittäminen ei myöskään tarjoa välitöntä arvoa asiakkaille, mutta toisaalta se on hyödyllistä ohjelmistoa tuottavalle yritykselle. [23]

Galster et al. tutkimuksen mukaan niitä voidaan käyttää ketterissä ohjelmistoprosesseissa suunnittelun apuna. [23] Toisaalta, muun muassa referenssiarkkitehtuurien hyöty arkkitehtuurin esiin tuomisen työkaluna voi olla erityisesti pieni, jos arkkitehtuuriin panostetaan muutenkin. Esimerkiksi 0-sprinttilähestymistavassa yksi iteraatiokierros käytetään pelkästään arkkitehtuurin suunnitteluun. Tässä vaiheessa on jo huomattu, että arkkitehtuuri on tärkeä osa ohjelmiston onnistumista.

Referenssiarkkitehtuureja voitaisiin käyttää startup-yrityksessä arkkitehtuurin pohjana kaikissa lähestymistavoissa, mikäli sopiva sellainen löytyy kyseiselle sovellukselle. Tällöin saadaan jokin aloituskohta arkkitehtuurisuunnitelmille. Vaihtoehto referenssiarkkitehtuurille olisi esimerkiksi kattavan ohjelmistokehyksen käyttäminen. Ohjelmistokehykset käsitellään kohdassa 5.2.

Yhteenvetona kaikista esitellyistä konsepteisa taulukko 5.1 esittää, missä vaiheissa mitäkin konseptia voidaan hyödyntää ja kuinka ne sopivat pienille startup-yrityksille niiden resurssien käytön puolesta.

Taulukko 5.1. *Monet esitellyistä konsepteista sopivat MVP-metodologian kanssa käytettäväksi.*

Konespti	Vaihe	Sopivuus pienille startup-yrityksille
Ohjelmarungot	Suunnittelu ja toteutus	Hyvä
Ohjelmistokehykset	Suunnittelu ja toteutus	Hyvä
Arkkitehtuuriprototyypit	Suunnittelu ja toteutus	Hyvä
Arkkitehtuurilliset kehykset	Suunnittelu	Huono/Keskinkertainen
Referenssiarkkitehtuurit	Suunnittelu	Hyvä

Kuten taulukko 5.1 kertoo, suurin osa esitellyistä menetelmistä on hyödynnettävissä MVP-prosessissa arkkitehtuurisuunnittelun apuna. Erityisesti ohjelmistokehykset, ohjelmarungot ja arkkitehtuuriprototyypit vaikuttaisivat olevan hyödyllisiä, sillä niitä voidaan käyttää sekä suunnittelu että toteutusvaiheessa. Niiden käyttö ei myöskään kuluta paljon resursseja.

Menetelmistä huonoiten MVP-kehitykseen sopii arkkitehtuurilliset kehykset. Ne eivät tue itse toteutusta ja monet niistä on suunniteltu suurten yritysten käyttöön, joka tekee niistä raskaita. Tällöin ne eivät suoraan tue Lean-ajattelumallia pienissä yrityksissä. Lean-ajattelumallia tukevia kehyksiä on kuitenkin olemassa ja mikäli ne ovat tarpeeksi keveitä pienelle yritykselle, niitä voidaan hyödyntää suunnittelun apuna.

6 ARKKITEHTUURIN DOKUMENTOINTI

Arkkitehtuurisuunnittelussa tietoa luodaan ja käytetään paljon. Arkkitehtuurillisesta tiedosta tärkeä osa koskee valittuja ratkaisuja komponenttien ja yhteyksien kannalta. Myös päätökset, jotka johtavat tiettyyn ratkaisuun ovat oleellinen osa arkkitehtuurillista tietoa. Tämä tieto on joko hiljaista engl. tacit knowledge tai avointa engl. explicit knowledge. [6] Hiljaisella tiedolla tarkoitetaan tietoa, jota ei välttämättä tiedosteta siinä hetkessä tai lausuta ääneen. Avoin tieto taas on tietoa, joka on avattu myös muille, esimerkiksi keskustelun tai dokumentoinnin yhteydessä. Molempia tiedon tyyppejä syntyy arkkitehtuurisuunnittelun aikana.

Dokumentointi on tärkeässä roolissa kaikissa Tang et al. määrittämissä ohjelmistoarkkitehtuurin suunnitteluvaiheissa. Dokumentaatiota joko luetaan tai sitä tuotetaan riippuen kyseessä olevasta vaiheesta ja yhdessä vaiheessa voidaan tehdä molempia.

Kun puhutaan ohjelmistoarkkitehtuurin dokumentoinnista, viitataan avoimeen arkkitehtuuritietoon. Siinä kirjataan ylös ja tehdään muille tiettäväksi esimerkiksi, mitä on keskusteltu tapaamisissa, tieto jota on saatu projektin ulkopuolelta, aikaisemmat kokemukset yleensä ja erityisesti mitä päätöksiä on juuri tehty. [6]

Arkkitehtuurillinen tieto koostuu:

- arkkitehtuurillisesta suunnitelmasta
- suunnittelupäätöksistä
- perusteluista.

Perustelut sisältävät olettamukset, kontekstin ja muut tekijät, jotka yhdessä määrittävät miksi juuri kyseinen lopullinen päätös on sellainen kuin se on. Yleensä muu kuin arkkitehtuurillinen suunnitelma pysyy hiljaisena tietona. Avoin kuvaus arkkitehtuurillisesta tiedosta on hyödyllistä laadukkaiden järjestelmien rakentamisessa ja kehittämisessä. [6, 38]

6.1 Dokumentoinnin haasteet

Kaikkea tietoa ei koskaan voi dokumentoida, eikä se olisi edes järkevää. Jansen et al. kokosivat listan dokumentoinnin haasteista. [32]

Dokumentaation *ymmärrettävyys* on vaikeaa taata. Kirjoittajan tarkoituksesta katoaa aina jotakin, kun joku muu lukee sen. Joidenkin asiakirjojen ymmärrettävyydestä tulee haas-

tavampaa, kun dokumentaatio laajenee järjestelmän monimutkaistuessa ja kasvaessa. Arkkitehdin käyttämä kieli ja konseptit eivät välttämättä ole kaikkien ymmärrettävissä erityisesti silloin, jos sidosryhmillä on eri taustatiedot. Pelkästään dokumentaation lukeminen johtaa usein epäselvyyksiin ja tulkintojen eroavaisuuksiin, vaikka hyvät viitteet ja sanastot helpottavatkin ymmärrettävyyttä. [32]

Olennaisen arkkitehtuuritiedon löytäminen laajasta arkkitehtuuridokumentaatiosta on usein ongelmallista. Tieto on usein ripoteltu useisiin asiakirjoihin. [10] Ensimmäinen haaste on asiaankuuluvien asiakirjojen löytyminen järjestelmää kuvaavien asiakirjojen joukosta. Niiden jakaminen sähköpostissa tai jaetuissa kansioissa monimutkaistaa oikeiden tietojen löytämistä. Tällöin eri ihmisillä on eri versiot samasta asiakirjasta. Toinen ongelma on halutun tiedon löytäminen näiden asiakirjojen sisältä. Vaikka dokumentaation selkeä rakenne, sanasto ja sisällysluettelo auttavat, ohjelmistoarkkitehtuuria koskevista asiakirjoista ei ole mahdollista löytää tiedon tarkkaa paikkaa. [32]

Jäljitettävyyden tuottaminen useiden asiakirjojen välillä on vaikeaa. Käytännössä usein on epäselvää, kuinka asiakirjat liittyvät toisiinsa. Muun muassa määrittely- ja arkkitehtuuriasiakirjojen väliltä voi olla vaikeaa löytää toisiinsa liittyviä tietoja. Tekstit ja taulukot eivät pysty täysin kuvaamaan kaikkia suhteita. Kaaviot, kuten näkymät tai mallit, ovat tehokkaampia suhteiden kuvaamisessa asiakirjojen sisällä ja niiden välillä. Mallien ja näkymien merkitykset eivät yleensä ole eksplisiittisiä ja siksi heikentävät ymmärrettävyyttä. [28]

Muutosten vaikutuksen arvioiminen koko järjestelmään on usein tarpeen. Siksi kun tehdään arkkitehtuurillisia päätöksiä, pitää analysoida mihin osiin arkkitehtuuria muutokset vaikuttavat. Dokumentaatiossa nämä päätökset ja niiden suhteet eivät yleensä ole esitetty sanallisesti, joten muutosten vaikutusten analysointi luotettavasti on usein erittäin vaikeaa. Jäljitettävyyden puuttuminen arkkitehtuurielementtien välillä pahentaa ongelmaa. [32, 58]

Arkkitehtuuridokumentaation tulee pitää ajan tasalla. Suurissa ja monimutkaisissa järjestelmissä muutoksia tulee usein ja arkkitehtuuridokumentaation päivittämisen hinta on joskus kohtuuton. Dokumentaatio vanhentuukin siksi nopeasti ja eri sidosryhmät, kuten kehittäjät ja ylläpitäjät eivät enää pidä dokumentaation tietosisältöä oikeellisena. [32, 42]

Lisäksi ongelmana voi olla se, että arkkitehtuurillista tietoa syntyy niin paljon, että sen kaiken dokumentoimiseen menisi erittäin paljon resursseja. Ei ole kovin tehokasta, jos aikaa kuluu tiedon tallentamiseen, eikä ohjelmisto etene. Lisäksi oman haasteensa asettaa se, että dokumentoinnista ei ole hyötyä sen tekijälle juuri sillä hetkelle [41]. Lisäksi ongelmana on tiedon hakeminen ja visualisointi sen dokumentoinnin jälkeen. Vaikka tieto olisi kirjattu johonkin ylös, se voi helposti hukkua muun sillä hetkellä epäoleellisen tiedon sekaan. [6]

Ohjelmistokehityksen ketterissä malleissa vaivannäön vähentäminen ja yksinkertaistus johti ajatukseen, että arkkitehtuuridokumentaation tulisi sisältää vain kriittisimmät osa-alueet. Näihin kuuluu esimerkiksi arkkitehtuurillisesti tärkeät vaatimukset tai kaaviot, jotka esittävät kriittisimpiä näkymiä [3]. Näkymiä käsitellään tarkemmin luvussa 6.2. Seuraavis-

sa luvuissa tutkitaan vertailun vuoksi sekä yleisimmin käytettyjä perinteisiä että uusimpia menetelmiä dokumentoida ohjelmistoarkkitehtuuria.

6.2 Näkymämallit

Organisaatiot voivat käyttää kolmea eri strategiaa tietonsa hallintaan: henkilöinti, koontaminen ja hybridistrategia. Hybridistrategiassa yhdistyy ominaisuuksia kummastakin aiemmasta strategiasta. [6]

Henkilöintistrategiassa dokumentoinnin kohteena on tieto siitä, kuka tietää kyseisestä asiasta. [6] Lean ja ketterät ajattelumallit vaikuttaisivat suosivan tätä menettelyä, koska järjestelmän kuvaus itsessään ei omaa korkeaa prioriteettia. Toisaalta, pienissä startup-yrityksissä ei ole paljon työntekijöitä. Tällöin ei välttämättä ole tarvetta erilliselle strategialle tiedon omaavan henkilön muistamiseen tulevaisuudessa. Yleensä joku toinen yrityksessä osaisi heti sanoa, kuka tietää asiasta, mikäli hän ei itse muista/tiedä sitä. Tässä on tietysti riskinä se, että työntekijä lähtee yrityksestä. Tällöin hän ei enää voi toimia tiedon lähteenä ja tieto häviää.

Koontaminen engl. codification vastaa klassista dokumentaatiota, eli se on järjestelmää kuvaavaa tietoa. Se voi olla tekstiä, kaavioita tai muulla tavalla järjestelmää koskevaa tallennettua tietoa. Koontamiseen on enemmän työkaluja kuin henkilöintiä varten ja se oli suuressa suosiossa aikaisempina vuosikymmeninä [6]. Osan alussa esiteltyjen ongelmien takia klassinen dokumentaatio ei kuitenkaan enää ole kovin suosittu tiedon tallentamistapa ketterissä menetelmissä. Muun muassa Scrumissa sille ei ole asetettu erillisiä käytäntöjä [5].

Koontamisesta voi kuitenkin olla joitain hyötyjä myös ketterää MVP-metodologiaa käytettäessä. Vaikka tietoa ei tallennettaisi sen pitkäaikaisen muistamisen vuoksi, voi erilaisen tekstin tai kaavioiden tuottaminen toimia ajattelua ohjaavana ja keskustelua herättävänä toimintana. Seuraavissa kappaleissa käsitellään erilaisia koontamisen työkaluja.

Modernit ohjelmistoarkkitehtuurisuunnittelun toimintamallit luottavat vieläkin Perryn ja Wolfin periaatteisiin. Arkkitehtuuri koostuu elementeistä, muodosta ja perusteluista. Elementit ovat jokaisen arkkitehtuurillisen kuvauksen päärakennusosa, sillä ne määräävät komponentit ja suhteet. Ei-toiminnalliset ominaisuudet määräävät lopullisen muodon arkkitehtuurille. Monet eri muodot samoilla toiminnallisuuksilla ovat mahdollisia. Syyt juuri tietyn muodon valitsemiseen ovat arkkitehtuurin syvintä olemusta, mutta ne usein jätetään liian vähälle huomiolle arkkitehtuurisuunnittelun aikana. Tämä johtuu siitä, että perustelut ovat arkkitehdin mielessä hiljaisena tietona, jota ei dokumentoida käytettävään muotoon. [6]

Ohjelmistoarkkitehtuuri on muodostunut suunnitteluprosessin tuotokseksi, joka sisältää tärkeitä päätöksiä:

- ohjelmistojärjestelmän järjestyksestä

- ohjelmiston rakenteellisista elementeistä ja niiden rajapinnoista, joiden vuorovaikutusten avulla määritetään ohjelmiston toiminta
- näiden elementtien yhdistämisestä suuremmiksi osakokonaisuuksiksi.

Vuosien ajan käytäntö ja tutkimus ovat keskittyneet eri perspektiiveihin, joita kutsutaan arkkitehtuurillisiksi näkymiksi. Nämä näkymät, jotka kuvaavat eri sidosryhmien mielenkiinnon kohteita tarjotaan joukkona yhdenmukaistettuja kuvauksia yhtenäisellä ja loogisella tavalla. Niitä käytetään myös tapana kommunikoida arkkitehtuuria. [6]

90-luvun alussa monet ryhmät ympäri maailmaa tajusivat, että laaja ja monimutkainen ohjelmistojärjestelmä koostuu useasta keskenään kietoutuneesta rakennelmasta. Aiemmat yritykset kuvata koko arkkitehtuuria yhden kaavion avulla ei ole mahdollista. Eri sidosryhmät ovat kiinnostuneita arkkitehtuurin eri osa-alueista ja tarkastelevat ohjelmistoa eri näkökulmista. Jokainen osa-alue voidaan kuvata erillisenä näkymänä eri sidosryhmiä varten. Näkymät ovat projektioita, abstraktioita tai yksinkertaistuksia monimutkaisemmasta kokonaisuudesta. [6]

Arkkitehtuurillinen näkymä on osittainen esitys tarkoin määriteltyjen arkkitehtuurillisten kiinnostuksen kohteiden joukosta. Näkökulma on joukko toimintatapoja jonkin näkymän rakennukseen, tulkintaan ja käyttöön. Tavallaan näkökulma on näkymälle, sama kuin mitä selitys on karttamerkinnöille. Vastaavuudet eri näkymien välillä ovat tärkeässä osassa, eli missä suhteessa elementit ovat usean näkymän välillä. [6]

Viimeisen 30 vuoden ajan tietotekniikan tutkijat ovat yrittäneen kehittää arkkitehtuurillisia kuvauskieliä. Nämä kuvauskielet kirjaavat ja esittävät ohjelmistojärjestelmän oleellisia elementtejä. [6] Viime vuosina UML on selkeästi laajimmin käytetty kuvauskieli. Sitä käytetään esimerkiksi notaationa esityksessä. [6]

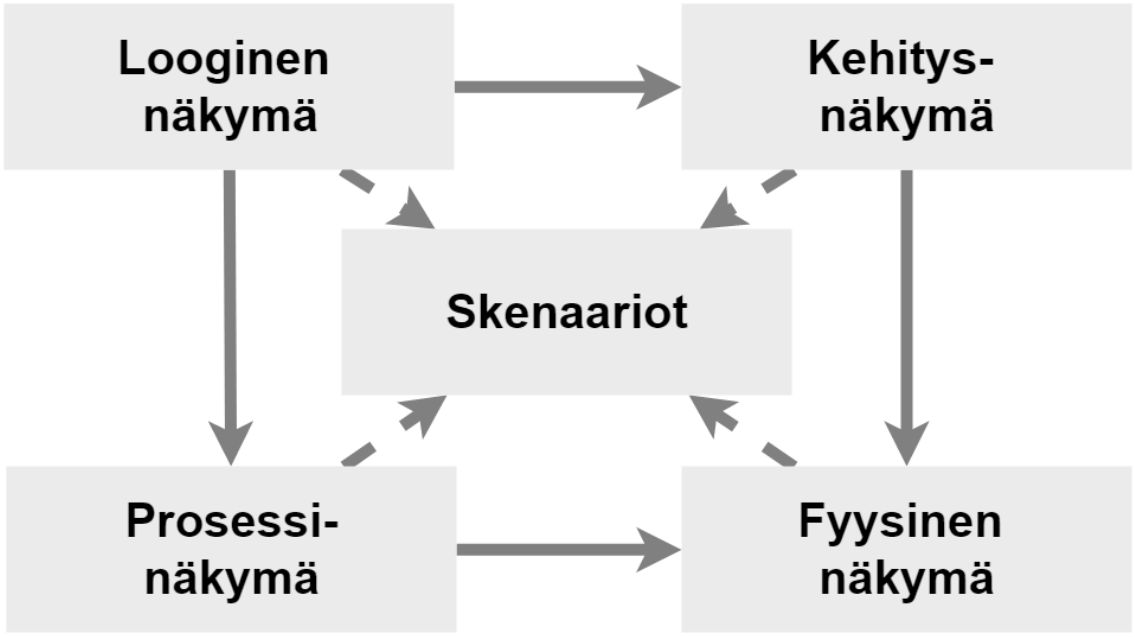
6.2.1 4+1-näkymämalli

4+1-näkymämalli on yksi teollisuuden laajasti käyttämistä ohjelmistoarkkitehtuurin esitysrakenteista [47]. Se syntyi tarpeesta saada looginen, selkeä ja luettava mallinnus ohjelmistoarkkitehtuurista. Sen tarkoitus on myös vähentää arkkitehdin tekemiä virheitä, kuten yhden elementin liian tarkkaa määrittelyä ja saada ajattelemaan arkkitehtuuria useammalta kannalta. [37]

4+1 näkymän mallissa ohjelmistoarkkitehtuuri jaetaan viiteen eri näkymään. Jokainen näkymä kohdentaa kuvauksen tiettyihin järjestelmän eri sidosryhmiä kiinnostaviin kohteisiin. Eri näkymät eivät ole itsenäisiä, vaan niiden elementit saattavat esiintyä useammassa näkymässä. [37]

Mallissa arkkitehtuuria kuvataan loogisen-, kehitys-, prosessi-, fyysisen ja skenaarionäkymän avulla. Nämä näkymät on esitetty kuvassa 6.1.

Loogisen näkymän tehtävänä on tukea toiminnallisia vaatimuksia, eli palveluita, joita jär-



Kuva 6.1. *4+1 näkymän mallin eri näkymät ovat suhteessa toisiinsa ja skenaarioihin. (Muokattu [37] kuvasta.)*

jestelmän pitäisi tarjota loppukäyttäjilleen. Suunnittelijat jakavat järjestelmän joukkoon tärkeimpiä abstraktioita, jotka ovat otettu ongelma-alueelta. Nämä abstarktiot ovat olioita tai olioluokkia, jotka käyttävät abstraktion, kapseloinnin ja perinnän periaatteita. Toiminnallisen analyysin tukemisen lisäksi jakamisen aikana tutkitaan mekanismit ja suunnitteluelementit, jotka jaetaan koko järjestelmän kanssa. Loogisen näkymän esittämiseen käytetään yleensä luokkakaaviota. [37]

Prosessinäkymä ottaa huomioon joitain ei-toiminnallisia vaatimuksia, kuten tehokkuuden ja järjestelmän saatavuuden. Siinä käsitellään rinnakkaisuutta ja tietokoneiden välistä jakamista ja järjestelmän eheyttä. Prosessinkäymä myös määrittää mitkä säikeet ajavat mitkäkin loogisen näkymän olioiden toiminnot. [37]

Suunnittelijat kuvaavat prosessinäkymässä järjestelmää usealla abstraktiotasolla, jotka kaikki käsittelevät tiettyä asiaa. Korkeimmalla abstraktiotasolla siinä kuvataan joukkoa itsenäisesti ajettavia kommunikoivien ohjelmien loogisia verkkoja. Ne on jaettu joukolle laitteistoresursseja, jotka ovat niin ikään yhdistetty keskenään väylällä, lähiverkon tai laajaverkon avulla. Useita loogisia verkkoja voi olla olemassa samaan aikaan. Ne voivat myös käyttää samoja fyysisiä resursseja. [37]

Prosessinäkymässä prosessi on joukko tehtäviä, jotka muodostavat ajettavan yksikön. Ne esittävät tasoa, jossa prosessinäkymää voidaan taktisesti ohjailla (käynnistää, palauttaa, uudelleen konfiguroida, sammuttaa jne.). Lisäksi prosesseja voidaan kopioida järjestelmän tehokkuuden kasvattamiseksi tai saatavuuden takaamiseksi. [37] Kuten muitakin näkymiä, prosessinäkymää nykyään kuvataan UML kaavioiden avulla. [47]

Fyysinen näkymä ottaa huomioon järjestelmän ei-toiminnalliset vaatimukset, kuten sen saatavuus, luotettavuus tai virheensietokyky, tehokkuus ja skaalautuvuus. Ohjelmistoa

ajetaan tietokoneverkossa. Loogisessa, prosessi- ja kehitysnäkymässä tunnistetut elementit jaetaan eri tietokoneille. [37]

Kehitysnäkymä keskittyy kuvaamaan ohjelmistomoduulien järjestystä ohjelmistokehitysympäristössä. Ohjelmisto on pakattu pieniin osiin, kuten kirjastoihin tai alijärjestelmiin. Niitä voidaan kehittää yhden tai useamman kehittäjän toimesta. Alijärjestelmät muodostavat tasoja sisältävän hierarkian, jossa jokainen taso tarjoaa kapean ja hyvin määritellyn rajapinnan sen yllä olevia tasoja varten. [37]

Kehitysnäkymä ottaa huomioon sisäiset vaatimukset, jotka liittyvät kehityksen helppouteen, ohjelmiston hallintaan, uudelleenkäyttöön sekä ohjelmointikielen ja työkalujen asettamat vaatimukset. Kehitysnäkymä tukee vaatimusten ja työn jakamista kehitystyöryhmille ja tukee kustannusarviointia, suunnittelua, projektin etenemisen seurantaa, kannettavuutta ja tietoturvaa. Se on aloituskohta uuden tuotelinjaston perustamiselle. [37]

Kruchten ehdottaa, että kehitysnäkymä esitettäisiin moduuli ja alijärjestelmädiagrammien avulla. Ne näyttävät järjestelmän vienti- ja tuontisuhteet. Kehitysnäkymä voidaan kuvata kokonaan vain, kun kaikki ohjelmistoelementit ovat tunnistettu. [37]

Kruchten kehottaa käyttämään 4-6 tasoa alijärjestelmien kuvaukseen. Alijärjestelmien tulisi riippua vain toisista järjestelmistä, jotka ovat samalla tai alemmalla tasolla. [37]

Käyttötapausnäkymässä otetaan tarkasteluun pieni joukko tärkeimpiä käyttötapauksia. Niiden avulla varmistetaan, että neljä muuta näkymää toimivat saumattomasti keskenään. Jokaiselle käyttötapaukselle tehdään vastaavat toimintakuvaukset. Skenaariot ovat jossain määrin abstraktioita kaikkein tärkeimmistä vaatimuksista. [37]

Käyttötapausnäkymä ei tuo uutta tietoa elementtien muodossa. Tämän vuoksi mallin nimessä on "+1". Sillä on kuitenkin oma tehtävänsä elementtien havaitsemisessa ja arkkitehtuurin validoinnissa. [37]

Kruchten ehdotti käytettäväksi erilaisia kuvaustapoja eri näkymille [37]. UML-merkintätapa on julkaisun jälkeen levinnyt niin vallitsevaksi tavaksi kuvata ohjelmistoja, että on perusteltua käyttää sitä vaihtoehtoisten kuvaustapojen sijaan. [47]

Vaikka mallin nimi on 4+1, Kruchten huomaa, että arkkitehtuurit eivät aina tarvitse 5:ttä eri näkymää. Hänen mukaansa näkymät, joita ei ole tarpeellista tehdä voidaan jättää pois kuvauksesta. Esimerkiksi fyysistä näkymää ei tarvitse tehdä, mikäli käytössä on vain yksi prosessi. Pienissä projekteissa voi myös olla mahdollista yhdistää looginen ja kehitysnäkymä. [37]

MVP:n tapauksessa saattaisi olla hyödyllisempää käyttää resurssia johonkin muuhun kuin kaikkien näkymien tekemiseen. 4-6 eri tason kuvaus voi myös kuvata järjestelmää liian tarkasti. Järjestelmä todennäköisesti muuttuu kehityksen aikana ja useasti vaihtuvien komponenttien kuvaus vain haittaa kehitystä, kun dokumentaatiota joudutaan päivittämään. Järjestelmää voidaan kuitenkin kuvata korkealla abstraktiotasolla tiedon säilyttämiseksi ja eri näkymien pohtimisesta saavutettavien hyötyjen vuoksi. MVP voitaisiin

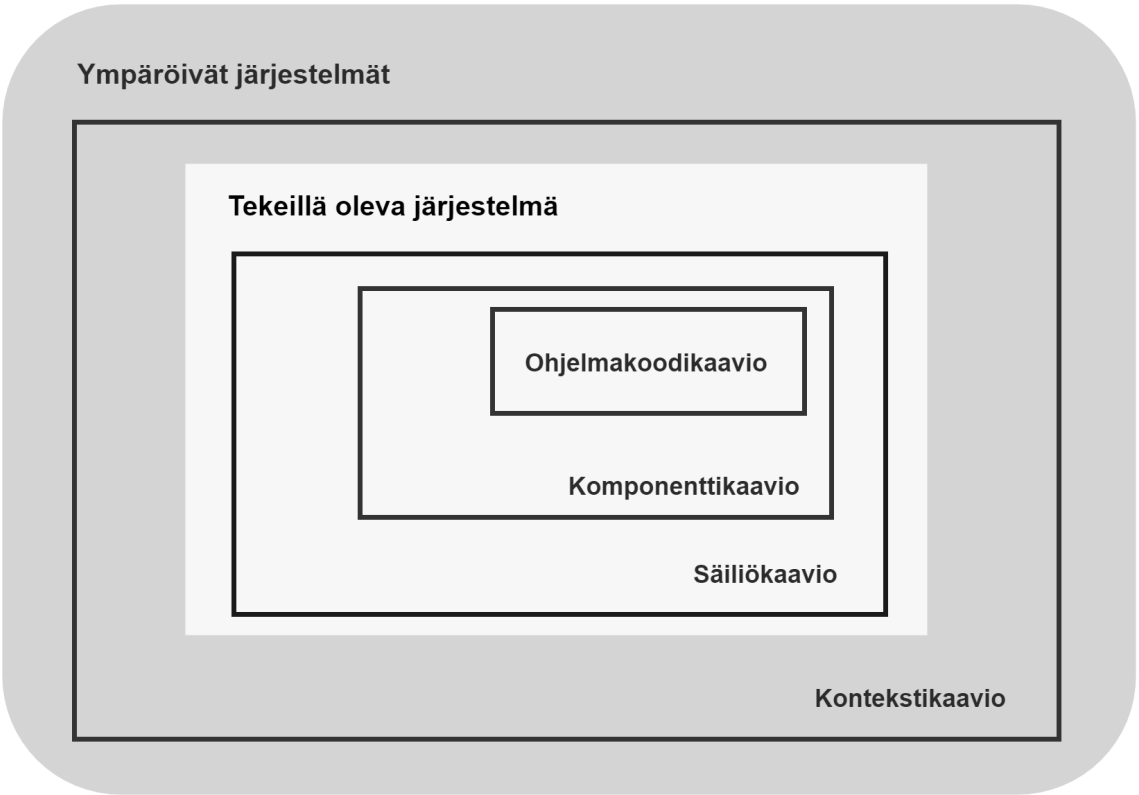
laskea pieneksi projektiksi, ja sitä kuvattaessa 4+1 mallilla onkin perusteltua yhdistää looginen ja kehitysnäkymä, ellei niitä erikseen tarvita.

6.2.2 C4-näkymämalli

C4-näkymämalli koostuu neljästä eri näkymästä. Sen tarkoituksena on kuvata vain tarvittava määrä arkkitehtuurista niin, että sen avulla voidaan kommunikoida sidosryhmien kanssa ja että kehittäjätyöryhmä ymmärtää arkkitehtuurin ilman yksityiskohtia, jotka muuttuvat, kun arkkitehtuuri muuttuu. [13, 31]

Malli käyttää tiettyjä abstraktiotasoja ohjelmiston staattisen rakenteen kuvaamiseksi. Siinä staattiseen rakenteeseen kuuluu järjestelmän erinäiset "säiliöt", kuten ohjelmat, tietovarastot, mikropalvelut ja niin edelleen. Lisäksi niihin kuuluu komponentit ja ohjelmakoodi. [13]

C4-malliin kuuluu nimensä mukaan neljä eri tasoa. Jokainen taso kuvaa järjestelmää hieman tarkemmin, kuten kaaviosta 6.2 voidaan huomata. Tasot kuvataan seuraavaksi.



Kuva 6.2. *C4:n koostuu neljästä eri tasosta.*

Taso 1 Järjestelmän kontekstikaavio Järjestelmän korkeimman tason kuvaus, C4-mallissa on järjestelmäkontekstikaavio. Siinä näkyy ohjelmisto ja kuinka se sijoittuu ympäristöönsä käyttäjien ja muiden ulkopuolisten järjestelmien suhteen. [13]

Taso 2 Säiliökaavio Toisen tason kuvaus siirtyy lähemmäksi järjestelmää. Siinä kuvataan ohjelmiston eri säiliöt, kuten edellä mainitut ohjelmat, tietovarastot ja niin edelleen, joista

ohjelma koostuu. Päätökset eri teknologioista tulevat esiin tässä kaaviossa. [13]

Taso 3 Komponenttikaavio Komponenttikaavio kuvaa jokaista säiliötä yksi kerrallaan tarkemmin ja näyttää sen sisällä olevat komponentit. Nämä komponentit tulisi olla oikeita abstraktioita koodista. [13]

Taso 4 Ohjelmakoodi C4-mallissa alin taso kuvaa ohjelmakoodia. Se on valinnainen näkymä, jonka voi tehdä, mikäli se on oleellista. Siinä esitetään yksi komponentti kerrallaan ja kuinka se toteutetaan. Siinä voi esimerkiksi näkyä mistä luokista ja toteutusyksityiskohdista kyseinen komponentti muodostuu. [13]

C4-malli ei pakota käyttäjää hyödyntämään mitään tiettyä merkintätapaa. Brown mainitsee, että esimerkiksi UML on merkintätapana sopiva C4-mallille. Hän kuitenkin suosittelee, että jokainen elementti sisältäisi tiettyjä ominaisuuksia, jotka helpottaisivat kaavioiden lukemista ja ehkäisisivät epäselvyyksiä.

C4-kuvaus on keveämpi tapa kuvata järjestelmää, kuin 4+1 näkymän malli esimerkiksi siksi, että siitä puuttuu ainakin yksi näkymä. C4-kuvaus keskittyy kuvaamaan järjestelmää eri kannoilta kuin 4+1 näkymän malli. Esimerkiksi C4-näkymä ei kuvaa käyttötapauksia millään tavalla. Tilanteesta riippuen kumpikin malli voi sopia MVP:n arkkitehtuurin kuvaamiseksi. C4-malli voi olla helpompi muokata kevyemmäksi, sillä alempia tasoja voidaan helposti vain jättää kuvaamatta. 4+1 mallista on vaikeampi jättää näkymiä pois, sillä kaikilla näkymillä on selkeästi määritelty tarkoitus. Tämän vuoksi ehkä C4-malli on käyttökelpoisempi malli ketterässä kehityksessä ja MVP:tä luotaessa.

6.3 Arkkitehtuuripäätöskuvaukset

Viime vuosikymmenen aikana on alettu käyttämään arkkitehtuuripäätöskuvauksia [25] engl. architecture decision description. Niissä keskitytään tallentamaan yksityiskohtaista tietoa järjestelmää koskevista suunnittelupäätöksistä ja niiden suhteista. [6]

Arkkitehtuuripäätöskuvaukseen kuuluu seuraavat osat:

- päätös itsessään
- perustelu
- laajuus
- tila
- päätöksen laatija
- aikaleima
- muutoshistoria
- kategoriat
- hinta
- riskit

- läheisesti yhteen kuuluvat päätökset
- suhteet ulkoisiin tuotoksiin. [6].

Tällaisenaan arkkitehtuuripäätöskuvauksissa on erittäin paljon kirjattavaa ja ylläpidettävää. Kaikkien päätösten kirjaus voi viedä pienellä työryhmällä paljon resursseja. Hadar et al. esittävät abstraktin arkkitehtuurimäärittelydokumentaation vaihtoehtona perinteisille dokumentointitavoille ja arkkitehtuuripäätöskuvauksille. [26]

Abstrakti arkkitehtuurimäärittelydokumentti sisältää sen hetkiseen julkaisuun kaikkein relevanteimman ja ajan tasalla olevan tiedon. Se sisältää myös tietoa tuotteen olemassa olevan arkkitehtuurin tilasta. Sisältää ns. hissipuheista lainattuja elementtejä. Dokumentti koostuu neljästä suuremmasta kokonaisuudesta:

- tuotteen yleiskatsaus
- tuotteen tavoitteet seuraavaa julkaisua varten
- yleiskatsaus tuotteen arkkitehtuurista kokonaisuudessaan yhdessä kyseisen sprintin muutosten kanssa
- ei-toiminnalliset vaatimukset.

Jotta kuvaus pysyisi lyhyenä ja kohdennettuna, Hadar et al. ehdottavat, että dokumentti pidettäisiin kaavamaisena. Arkkitehdit lisäävät asianmukaisen tiedon esimääriteltyihin kenttiin rönsyilevän vapaamuotoisen tekstin kirjoittamisen sijaan. [26]

Hadar et al mukaan abstraktilla arkkitehtuurimäärittelydokumentilla on monia etuja perinteiseen dokumentaatioon verrattuna. Se helpottaa arkkitehtien yhteisen kielen muodostumista ja yksinkertaistaa arkkitehtien ja sidosryhmien välistä kommunikointia. Se esimerkiksi tehostaa tiedon esitystapaa, jolloin arvioijat voivat keskittyä tietosisältöön ja suunnittelupäätöksiin. Muuten pitkät kerronnalliset toteamukset hämärtävät arkkitehtuurillista rakennetta. Hadar et al. väittävät, että lyhyen ja tiettyyn kohteeseen suunnatun tekstin arvostelu johtaa rakentavaan ja kohdennettuun kommentointiin. Arvostelijat voivat helposti seurata eri julkaisujen välillä tehtyjä arkkitehtuurin muutoksia ja arvioida arkkitehtuurin kehitystä pitkällä aikavälillä. [26]

6.4 Arkkitehtuurihaiku

Arkkitehtuurihaiku engl. architecture haiku on dokumentointitapa, joka on luotu turhan dokumentoinnin vähentämiseen. Siinä arkkitehtuurin tärkeimmät asiat kuvataan yhdellä paperilla. Yhden paperiarkin rajan asettaminen arkkitehtuurin kuvaukselle pakottaa arkkitehdit keskittymään suunnitelmien tärkeimpiin osa-alueisiin. Tällöin arkkitehdeille ei niin ikään jää tilaa tuhlattavaksi ylimääräisillä yksityiskohdilla. [36]

Arkkitehtuurihaikun tulisi sisältää

- lyhyt esittely ratkaisusta yleensä
- lista tärkeistä teknisistä rajoitteista
- korkean tason tiivistelmä toiminnallisista vaatimuksista
- priorisoitu lista laatuattribuuteista
- lyhyt selitys suunnittelupäätöksistä
- lista arkkitehtuurillisista tyyleistä ja suunnittelumalleista
- vain kaaviot, jotka lisäävät merkitystä muun sivulla olevan tiedon lisäksi. [36]

Arkkitehtuurihaikun tiivistämiseksi voidaan käyttää kognitiivisia laukaisijoita engl. cognitive trigger. Päätösten yksityiskohtaisen kuvaamisen sijaan voidaan olettaa, että lukija tietää jo paljon tarvittavaa taustatietoa. Kaksi yleistä kognitiivista laukaisijaa ovat arkkitehtuurilliset tyylit ja mallit. Esimerkiksi "kerros” voi kertoa jo paljon rakenteesta. Yhdellä sanalla tai ilmauksella voidaankin esittää paljon informaatiota lyhyesti. Näiden sanojen ja ilmausten merkitystä ei tarvitse toistaa dokumentaatiossa, mutta sitä lukevat henkilöt voivat silti ymmärtää dokumentin. [36]

Arkkitehtuurihaiku voi olla tehokas työkalu dokumentoinnin tehostamiseksi. Se ei kuitenkaan ole aina oikea työkalu joka projektiin tai tapaukseen. Kaikki projektin sisällä eivät välttämättä tiedä kognitiivisia laukaisijoita tai terminologian merkitys vaihtelee henkilöstä toiseen. Terminologian ongelmat voidaan kuitenkin ratkaista keskustelemalla termeistä ja päättämällä yhteiset termit, joita käytetään. [36]

Arkkitehtuurihaiku ei auta myöskään eri arkkitehtuurillisten vaihtoehtojen läpi käymisessä. Sen tehtävänä on toimia vain tiedon tallennustyökaluna. Se voi jopa olla haitallinen, koska se rajoittaa luovuutta, eikä anna tilaa eri vaihtoehtojen tutkimiselle. Arkkitehtuurihaiku tulisi kirjoittaa vasta, kun kaikki eri vaihtoehdot on tutkittu perusteellisesti ja ongelman ratkaisutapa on selvillä. [36]

Arkkitehtuurihaiku voi olla startup-yritykselle hyödyllinen menetelmä tiedon pitempiaikaiseen säilyttämiseen. Sen tekeminen ja ylläpito vaatii huomattavasti vähemmän resursseja, kuin esimerkiksi eri näkymämallit. Se ei kuitenkaan takaa yhtä luotettavaa ja ymmärrettävää dokumentaatiota. Arkkitehtuurihaiku luottaa arkkitehtien muistiin ja kognitiivisiin laukaisijoihin, tämän vuoksi se ei välttämättä toimi tiedon välittäjänä kasvavassa startup-yrityksessä. Uusille työntekijöille tulee selittää, mitä eri termeillä tai ilmauksilla tarkoitetaan. Myös ajan myötä arkkitehdit voivat unohtaa niiden tarkoitukset, jolloin haiku muuttuu vaikeaselkoiseksi.

Arkkitehtuurihaiku tällaisessa muodossa ei suoraan ota kantaa Lean Startup -ajatusmalliin. Asiat, jotka sisältyvät siihen kuvaavat vain ohjelmiston arkkitehtuuria, eikä esimerkiksi MVP-iteraation tarkoitusta ole sisällytetty siihen. Sitä voitaisiin mahdollisesti muokata sisällyttämään asioita, jotka liittyvät erityisesti Lean Startup -prosessiin. Esimerkiksi validointiin liittyviä kysymyksiä voisi tuoda esiin myös arkkitehtuurihaikussa. Asioita, jotka

voitaisiin sisällyttää arkkitehtuurihaikuun Lean Startup -menetelmän yhteydessä olisi mielenkiintoista selvittää jatkotutkimuksilla. Esimerkiksi ongelmana voi olla se, riittääkö sivu kuvaamaan sekä arkkitehtuuria että liiketoimintamallin testausta ja tulisiko arkkitehtuurihaikujen eri versioita tulisi säilyttää myöhempää tarkastelua varten.

7 ARKKITEHTUURIN ARVIONTI

Arkkitehtuurin muuttaminen ohjelmiston toteuttamisen jälkeen voi vaatia paljon resursseja. Arkkitehtuurin arvioinnilla ennen ohjelmiston toteuttamista voidaan tehdä johtopäätöksiä, siitä kuinka se pystyy täyttämään asetetut vaatimukset. Siinä voidaan tarkastella myös refaktoroinnin tarvetta.

Arkkitehtuurin arvioinnin yksi suurimmista hyödyistä on sen kyky huomata ongelmat ja riskit aikaisessa vaiheessa. Tällöin ne voidaan helposti korjata tai niiden vaikutuksia voidaan lieventää verrattuna esimerkiksi vasta toteutetun ohjelmiston testauksessa havaittaviin ongelmiin. Lisäksi arkkitehtuurin arviointi kannustaa kommunikoimaan eri sidosryhmien kanssa tilanteissa, jolloin niin ei muuten tehtäisi. Arviointia voidaan parhaassa tapauksessa käyttää osana dokumentointiprosessia. [27]

Vaikka arkkitehtuurin arvioinnilla on monia hyötyjä, sitä ei käytetä laajasti teollisuudessa. Monet yritykset ovat tietoisia sen hyödyistä, mutta hyvin harvat käyttävät sitä. Lisäksi ketterät menetelmät eivät kannusta arkkitehtuurin arviointiin, sillä niihin yleensä kuuluu paljon aikaa ja resursseja. [9] Ketterissä prosesseissa arkkitehtuurianalyysin suorittamiseen on omat haasteensa, koska arkkitehtuuria suunnitellaan tyypillisesti osissa jatkuvasti. Tämän vuoksi arkkitehtuurianalyysin itsensä tulee pystyä analysoimaan osittaista arkkitehtuuria sekä sitä pitää pystyä suorittamaan osissa. Joitain perinteisiä analysointimenetelmiä joudutaankin siis muokkaamaan ketterään projektiin sopivaksi. [9, 27]

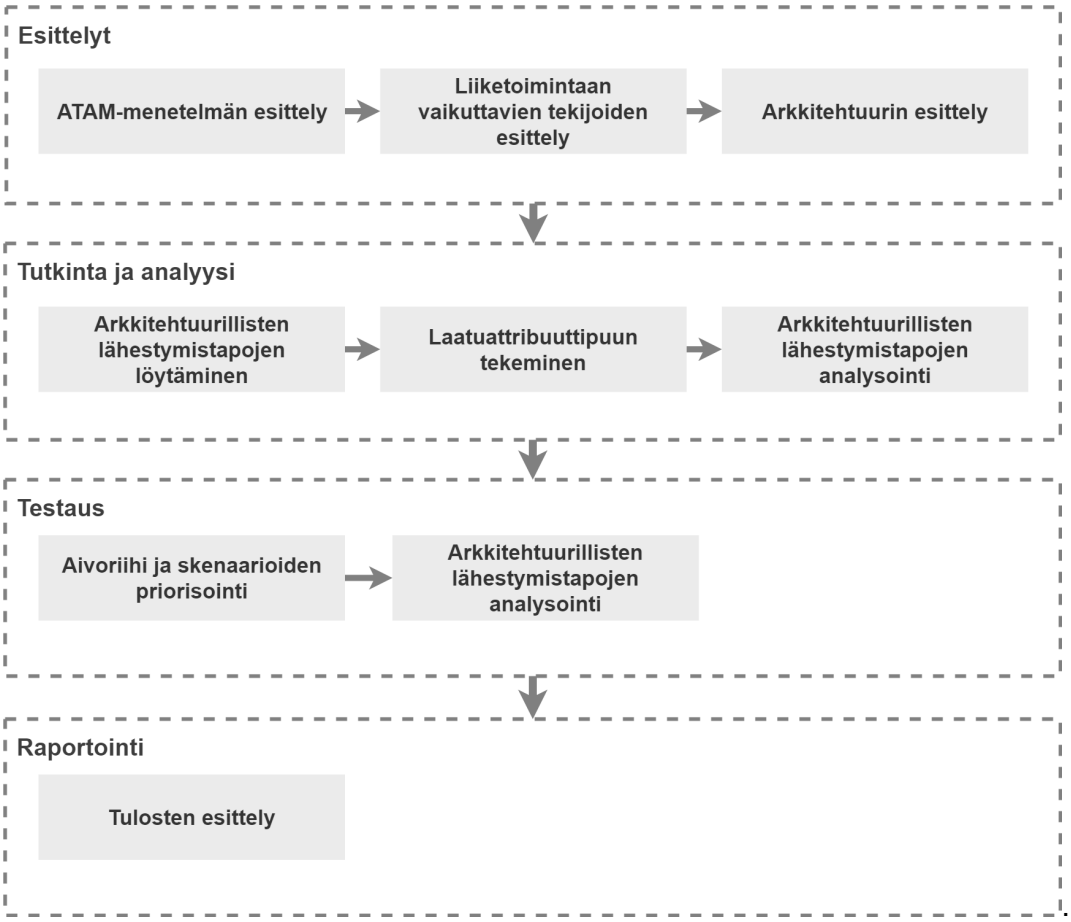
Ohjelmistostartup-yrityksen viitekehyksessä olisi erittäin suotuisaa, jos MVP-prosessin aikana voitaisin luoda iteratiivisesti arkkitehtuuri, joka tukee ohjelmistokehityksen etenemistä ja laatuvaatimusten täyttymistä. Jos arkkitehtuurin arvioinnilla voidaan saavuttaa tällainen prosessi, niin se voisi olla hyödyllinen vaihe MVP:n suunnittelussa. Tässä luvussa esitelläänkin arkkitehtuurin arviointitapoja ja tarkastellaan niiden sopivuutta ketterään kehitykseen startup-ympäristössä.

7.1 ATAM

Architecture Tradeoff Analysis Method eli ATAM on yksi yleisemmin teollisuudessa käytetyistä arkkitehtuurin arviointimenetelmistä. [21, 65] Sen kehittivät Kazman et al. vuonna 2000 tavaksi arvioida ohjelmiston arkkitehtuuria usean eri kilpailevan laatuattribuutin suhteen [34]. ATAM:in yleisyyden vuoksi se otetaan tässä työssä lähempään tarkasteluun.

ATAM tehtiin halusta luoda järkiperäisiä päätöksiä kilpailevien arkkitehtuurien välillä, jotka pohjautuvat hyvin dokumentoituihin järjestelmän attribuuttien analyyseihin, keskittyen tasapainotettavien valintojen löytämiseen. Kehityksen varhaisessa vaiheessa ATAM toimii vaatimusten täsmentämisen työkaluna. [34]

Menetelmän tarkoituksena on edesauttaa kommunikaatiossa sidosryhmien välillä, selventää ja täsmentää vaatimuksia sekä tarjota rungon yhtäaikaiseen arkkitehtuurin suunnitteluun ja analyysiin. [34]



Kuva 7.1. *ATAM koostuu 9 eri vaiheesta.*

ATAM-menetelmä koostuu neljästä pääosasta:

- esittelyt
- tutkinta ja analyysi
- testaus
- raportointi.

Ne voidaan jakaa vielä yhteensä 9:ään hienojakoisempaan vaiheeseen. [35] Niitä käsitellään seuraavaksi.

Esittelyt

Vaihe 1 — ATAM-menetelmän esittely. ATAM aloitetaan esittämällä menetelmä kaikille

sidosryhmille. [35]

Vaihe 2 — Liiketoimintaan vaikuttavien tekijöiden esittely Toisessa vaiheessa projektipäällikkö kuvaa liiketoiminnalliset tavoitteet, jotka ovat kehitystyön motivoivia tekijöitä. Hän myös kuvaa, mitkä ovat arkkitehtuuriin vaikuttavat taloudelliset tekijät. [35]

Vaihe 3 — Arkkitehtuurin esittely. Kolmannessa vaiheessa arkkitehti esittelee alustavan arkkitehtuurisuunnitelman. Esittely keskittyy kuvaamaan arkkitehtuuria liiketoimintatavoitteiden näkökulmasta ja sitä kuinka arkkitehtuuri tukee liiketoimintatavoitteita. [35]

Tutkinta ja analyysi

Vaihe 4 — Arkkitehtuurillisten lähestymistapojen löytäminen. Tutkinnan ja analyysin ensimmäisessä vaiheessa arkkitehti selvittää eri arkkitehtuurilliset lähestymistavat tilanteeseen. Tässä vaiheessa niitä ei kuitenkaan vielä analysoida. [35]

Vaihe 5 — Laatuattribuuttipuun tekeminen. Laatuvaatimukset, jotka muodostavat järjestelmän hyödyllisyyden (kuten tehokkuus, saatavuus, tietoturva jne.) tuodaan esiin. Niistä tehdään määrittelyt skenaariotasolla ja niiden syyt ja vasteet merkataan sekä priorisoidaan. [35]

Vaihe 6 — Arkkitehtuurillisten lähestymistapojen analysointi. Vaiheen 5 tärkeiden prioriteettien pohjalta valitaan arkkitehtuurit, jotka tukevat näitä ominaisuuksia. Tämän jälkeen ne analysoidaan ja löydetään arkkitehtuurilliset riskitekijät, herkkyyskohdat ja kompromissit. [35]

Testaus

Vaihe 7 — Aivoriihi ja skenaarioiden priorisointi. Vaiheessa 5 löydettyjen skenaarioiden pohjalta tehdään suurempi joukko skenaarioita koko sidosryhmäjoukon avulla. Näille skenaarioille asetetaan prioriteetti äänestyksellä. Tähän äänestykseen osallistuu kaikki sidosryhmät. [35]

Vaihe 8 - Arkkitehtuurillisten lähestymistapojen analysointi. Tässä vaiheessa iteroidaan uudelleen vaihetta 6. Suurimmalla prioriteetillä olevat skenaariot toimivat testitapauksina lähestymistapojen analyysissä. Nämä testitapaukset voivat auttaa löytämään uusia lähestymistapoja, riskejä, herkkyyspisteitä tai kompromisseja. Ne dokumentoidaan analyysin aikana.

Raportointi

Vaihe 9 - Tulosten esittely. Viimeisessä vaiheessa ATAM-ryhmä esittelee löytönsä sidosryhmille ja kirjoittaa mahdollisesti siitä raportin, josta käy ilmi ehdotetut strategiat. [35]

ATAMissa suunnittelu tehdään toteutuksesta riippumattomana erillisenä tapahtumana. Tämä lisää etukäteissuunnittelua, jos suunnitelmien oletetaan olevan valmiita toteutuksen alkaessa. Tästä syystä se ei sovi yhtä hyvin ketterään prosessiin, kuin iteratiiviset ja toteutuksen kanssa samaan aikaan tehtävät analyysimenetelmät. Ongelmallista on myös se, että aikaisemmin toteutetusta ohjelmistosta opittuja asioita ei oteta erikseen

huomioon. Voisi olla hyödyllistä tietää aikaisemmin opitun tiedon pohjalta, voiko kyseisellä ratkaisulla päästä asetettuihin laatuvaatimuksiin. Vaikkakin, jos arkkitehdit ovat olleet tekemässä toteutusta, he voivat todennäköisesti tuoda havaintoja esiin itse analyysin aikana.

Kehittyvän järjestelmän arkkitehtuuri ketterässä projektissa luo monia haasteita. Kazman et al. ehdottavat laatuattribuuttien pohtimista muiden kehitykseen liittyvien tapaamisten yhteydessä. Esimerkiksi Scrumissa retrospective-tapaamisessa laatuattribuutteihin liittyviä puutteita voidaan nostaa esiin tai sprintin suunnittelupalaverissa arkkitehti voi esittää laatuattribuutteihin liittyviä korjauksia. Tässä pystytään esittämään myös hyvin miksi arkkitehtuuriin liittyviä korjaus- tai parannustoimenpiteitä tulisi tehdä. [11]

ATAMissa aivoriihitapahtumiin tulisi osallistua kaikkien sidosryhmien edustajat, 4-5 arvioijaa ja 5-10 projektiin liittyvää henkilöä. [35]

ATAM on luonteeltaan erittäin raskas prosessi. Se vaatii paljon taustatietoja, siihen kuluu paljon resursseja ja sitä on sen vuoksi pidetty pienissä projekteissa ja yrityksissä mahdottoman ratkaisuna. Viime vuosina sitä on kuitenkin yritetty tuoda ketterään prosessiin mukaan poistamalla tai muokkaamalla eri vaiheita [11, 22]. Mitkä vaiheet poistetaan? ATAM on sellaisenaan suositeltava prosessi vain, kun kyseessä on vakiintunut ja resurssirikas yritys ja projekti, joissa syvällinen arkkitehtuurianalyysi on oleellinen laadun takaamiseksi. Räätälöitynä se voi sopia ketterään projektiin.

7.2 DCAR

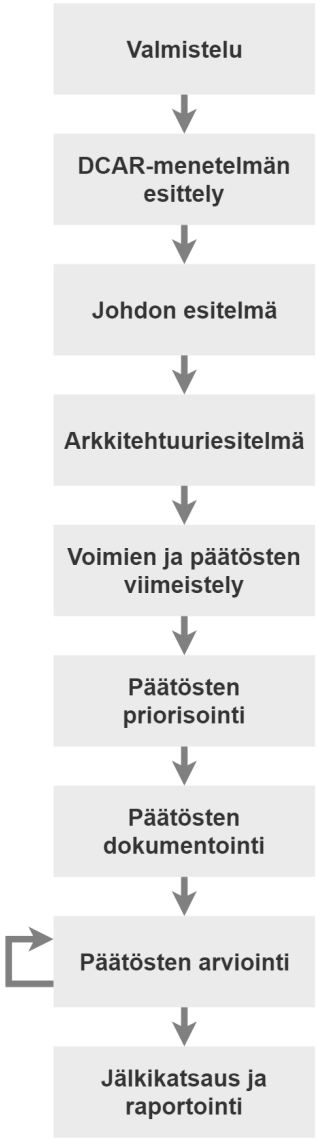
Decision centric architecture review eli DCAR on ATAM-menetelmää uudempi ja kevyempi menetelmä. Se perustuu ajallisesti ja resurssien puolesta kevyeen malliin ja tukee yksittäisten päätösten arviointia mahdollistamalla niiden testauksen ja dokumentoinnin. [27]

DCAR on suunniteltu olemaan kevyt ja ketterän kehityksen kanssa yhteensopiva prosessi. Sen kesto on noin puoli työpäivää ja vaatii 3:sta 5:een osallistujaa. Tämän vuoksi se sopii pienempiin projekteihin, joissa ei ole resursseja syvällisen arkkitehtuurianalyysin suorittamiseen. [27]

DCAR perustuu arkkitehtuurillisiin päätöksiin ja voimiin. Arkkitehtuurilliset päätökset on esitelty luvussa 6.3. Voima engl. decision force on mikä tahansa ei-triviaalinen vaikutus arkkitehtiin, joka etsii ratkaisua arkkitehtuurilliseen ongelmaan. DCAR prosessi aloitetaan valitsemalla joukko analysoitavia päätöksiä, joita arvioidaan käyttäen asiaankuuluvien projekti- ja yritysvoimien kautta.

Kuten ATAM DCAR koostuu 9:stä vaiheesta, jotka on esitelty kuvassa 7.2. Vaiheet ovat kuitenkin kestoltaan lyhyempiä ja vaativat vähemmän osallistujia. [27]

Vaihe 1 - Valmistelu. Ennen tapaamisen aloitusta sovitaan sen ajankohdasta ja järjestelmäarkkitehti valmistelee esitelmän, joka sisältää tärkeimmät arkkitehtuurilliset vaatimukset, korkean tason näkymät ja käytetyt lähestymistavat,kuten suunnittelumallit ja tyylit,



Kuva 7.2. *DCAR on 9-osainen mutta ATAM-menetelmää kevyempi prosessi. (Muokattu [27] kuvasta.)*

sekä teknologiat. Johdon ja asiakkaan edustajat valmistelevat esityksen, joka kuvaa ohjelmistoa tuotteena, sen alan, liiketoimintaympäristön ja tuotteen kehitystä ajavat liiketoimintavaatimukset ja rajoitteet. [27]

Ennen DCAR-tapaamista siihen osallistuvien tulisi tutkia heille annettua materiaalia löytääkseen mahdolliset arkkitehtuurilliset päätökset ja voimat.

Vaihe 2 - DCAR-menetelmän esittely. Arvointisessio alkaa menetelmän esittämisellä. Siinä tulisi esitellä aikataulu, DCAR-mene-telmän vaiheet, arvioinnin laajuus, mahdolliset seuraukset sekä osallistujien roolit ja vastuut. [27]

Vaihe 3 - Johdon esitelmä. Johdon tai asiakkaan edustaja antaa lyhyen esitelmän, jonka hän on valmistellut ennen tilaisuutta. Esityksen tulisi kestää 15-20 minuuttia. Sen tehtävänä on tuoda esiin liiketoimintaan liittyvät voimat, jotka pitää ottaa arvioinnin aikana huomioon. [27]

Vaihe 4 - Arkkitehtuuriesitelmä. Johtava arkkitehti esittelee arkkitehtuurin tilaisuuteen osallistuville käyttäen valmistelemaansa materiaalia. Tähän tulisi riittää 45-60 minuuttia. Tarkoituksena on antaa kaikille osallistujille yleiskuva arkkitehtuurista. Esitelmän tulisi olla erittäin vuorovaikutteinen, eli osallistujien tulisi pystyä saamaan vastauksia heidän kysymyksiinsä. Tämän vaiheen aikana arvioijat uusivat ja täyttävät listaansa arkkitehtuurillisista päätöksistä, jotka he aloittivat ensimmäisessä vaiheessa. Arkkitehtuurillisten päätösten tunnistaminen vaatii kuitenkin hieman kokemusta. [27]

Arkkitehtuurillisten päätösten tallentamisen lisäksi arvioijat muuttavat ja täydentävät listaansa voimista, jotka he ovat tunnistaneet aikaisemmista vaiheista. Voimat voidaan dokumentoida epävirallisissa toteamuksissa. [27]

Vaihe 5 - Voimien ja päätösten viimeistely. Arvioijat ovat luoneet alustavan listan arkkitehtuurillisista päätöksistä ja voimista. Tässä vaiheessa täsmennetään arkkitehtuurillisia päätöksiä ja niiden suhteita lisäksi viimeistellään ja tarkistetaan päätöksiin liittyvät voimat. Täsmennyksen apuna yksi arvioijista luo heti prosessin alussa päätöksien suhteista kaavion, jota hän päivittää aiempien vaiheiden ajan. [27]

Vaihe 6 - Päätösten priorisointi. Yleensä edellisessä vaiheissa löydetään liian monta päätöstä keskusteltavaksi yhden tilaisuuden aikana. Tämän vuoksi sidosryhmät joutuvat valitsemaan, mitkä päätökset arvioidaan seuraavien vaiheiden aikana. Kriteerit päätösten valitsemiseen riippuvat kontekstista, mutta riskejä ja suuria kuluvaikutuksia sisältävät päätökset tulisi ainakin arvioida. [27]

Heesch et al. ehdottavat pisteisiin perustuvaa valintamenetelmää. Jokainen osallistuja saa 100 pistettä, jotka he voivat jakaa päätöksien välillä vapaasti, perustuen sovittuihin kriteereihin. Päätöksen pisteet lasketaan sitten yhteen ja jokaisen henkilön asettamat pistemäärät perustellaan ja niistä keskustellaan. Eniten pisteitä saaneet päätökset jatkavat seuraaviin vaiheisiin arvosteltavaksi. Heesch et al. kertovat että 7-10 päätöksestä voidaan keskustella tehokkaasti puolessa päivässä. [27]

Vaihe 7 - Päätösten dokumentointi. Päätöksiä dokumentoitaessa arkkitehti ja muut tilaisuuteen osallistuvat dokumentoivat päätökset, jotka saivat korkeimmat luokitukset edellisessä vaiheessa. Jokainen henkilö valitsee 2 tai 3 päätöstä, joista heillä on osaamista. Ne dokumentoidaan kuvaamalla:

- sovellettu ratkaisu
- ongelma, jonka se ratkaisee
- tiedossa olevat vaihtoehdot
- voimat, jotka tulee ottaa huomioon, kun päätöstä arvioidaan.

Sidosryhmät käyttävät edellisessä vaiheessa koottua listaa erinäisistä voimista varmistaakseen, etteivät ne unohdu. He voivat kuitenkin keksiä uusia voimia vielä tässä vaiheessa. [27]

Vaihe 8 - Päätösten arvioint.i Dokumentoinnin jälkeen seuraava tehtävä on päätösten

arviointi. Arviointi aloitetaan tärkeimmästä päätöksestä. Sen dokumentoinut osallistuja esittelee sen lyhyesti, jonka jälkeen osallistujat haastavat päätöksen etsimällä lisävoimia päätöstä vastaan. Dokumentaatiota voimista ja päätösten välisistä suhteista päivitetään koko ajan tässäkin vaiheessa. Kaikki osallistujat päättävät yhdessä, mikäli voimat päätöstä vastaan ovat voimakkaampia kuin sen puolesta. [27]

Lopulta kaikki osallistujat päättävät äänestämällä päätöksen tilasta. Päätös voi olla hyvä, hyväksyttävä tai siitä pitää keskustella uudestaan. Äänestykseen voidaan käyttää esimerkiksi värikoodeja ilmaisemaan eri mielipiteitä. [27]

Koko keskustelun ajan arvioijat kirjaavat muistiinpanoihinsa mahdolliset ongelmat tai riskit, jotka tulevat esiin keskustelussa. Jokaiselle päätökselle tulisi antaa keskusteluaikaa noin 15-20 minuuttia, koska keskustelun laatu yleensä heikkenee sitä jatkettaessa. [27]

Vaihe 9 - Jälkikatsaus ja raportointi. Kun kaikki valitut päätökset on arvioitu, muistiinpanot ja tuotokset kerätään yhteen. Niitä voidaan käyttää arviointiraportin luomiseen, jonka arvioijatyöryhmä kirjoittaa. Raportista keskustellaan arkkitehdin kanssa ja arvioijaryhmä voi vielä muokata raporttia sen perusteella. Arvioijien tulisi kirjoittaa raportti mahdollisimman pian, sillä näin ajatukset ovat heillä vielä paremmin muistissa, kuin pitkän ajan jälkeen.

DCAR sopii todennäköisesti startup-yritykselle ATAM-menetelmää paremmin. Tämä johtuu siitä, että se on huomattavasti keveämpi operaatio. Se on mahdollisesti vieläkin liian raskas toteutettavaksi joka sprintin tai iteraation yhteydessä 9:n eri vaiheen ja monien eri osallistujaroolien vuoksi. Lisäksi, erillisen raportin kirjoittaminen voi olla hidasta ja kehittäjien mielestä epämiellyttävää.

Lean Startup -prosessissa DCAR-arviointimenetelmän käyttö olisi mahdollisesti hyödyllistä siinä vaiheessa, kun validointi on osoittanut liiketoimintamallin tarpeeksi toimivaksi. Tällöin voitaisiin arvioida, onko MVP:n arkkitehtuuri vielä sopiva lopulliselle tuotteelle. Mikäli arviointi osoittaa, että arkkitehtuuria täytyy muuttaa vain hieman, MVP:n koodia voitaisiin refaktoroida uusien suunnitelmien mukaiseksi. Ohjelmisto voitaisiin myös ohjelmoida tässä vaiheessa uudelleen alusta, jos taas refaktorointi vaikuttaisi vievän sitä enemmän resursseja. Tämän työn puitteissa tätä mahdollisuutta ei pystytty tutkimaan esimerkiksi case-tutkimuksessa, sillä siinä ei päästy tilanteeseen jossa liiketoimintamalli olisi validoitu. Tämä olisi kuitenkin mielenkiintoinen jatkotutkimusaihe pienen startup-yrityksen kannalta. Arvoinnin tekeminen validoidun liiketoimintaidean jälkeen voisi nopeuttaa ohjelmistokehitystä ja parantaa lopullisen tuotteen laatua.

Taulukko 7.1. *ATAM ja DCAR erovat resurssien kulutukseltaan.*

Ominaisuus	ATAM	DCAR
Kesto	3 päivää	Puoli päivää
Osallistujia	10-20	3-5
Toimii dokumentoinnin apuna	Kyllä	Kyllä
Kattavuus	Erittäin kattava	Kattava

Taulukossa 7.1 on esitetty ATAM ja DCAR -menetelmien eroavaisuuksia. Siitä voidaan huomata, että DCAR sopii paremmin pieniin ja ketteriin projekteihin.

8 CASE-TUTKIMUS

Kirjallisuustutkimuksen yhteydessä toteutettiin pienimuotoinen case-tutkimus yhden ohjelmistoprojektin toiminnasta. Case tutkimus seurasi projektin toimintaa noin puolen vuoden ajan. Projektissa kehitettiin uudenlaista rakennusalan verkkopalvelua. Se tapahtui MVP:tä käyttäen ja projekti sisälsi monia startup-yrityksen piirteitä. Muun muassa siinä oli tavoitteena löytää toimiva liiketoimintamalli ja maksavia asiakkaita erilaisten sidosryhmien kanssa keskustellen. Sen pohjalta ei kuitenkaan perustettu erillistä yritystä. Tutkimuksen aikana seurattiin:

- miten MVP-prosessi toimi
- kuinka ohjelmistoarkkitehtuuria suunniteltiin ja toteutettiin projektissa
- millaisia tuloksia suunnittelumenetelmät tuottivat.

Case-tutkimuksen kohteena oleva projekti oli resursseiltaan erityisen pieni. Siihen kuului vain yksi rakennusalan asiantuntija ja yksi ohjelmoija, jotka työskentelivät projektin parissa vapaa-ajallaan. Lisäksi projektissa oli ulkoinen toimija, joka tuotti ulkoasun tuotteelle ja erilliset esittelysivut verkossa. Joka tapauksessa työryhmä oli huomattavasti alle Dande et al. [17] suositteleman määrän. Ennen tutkimusta oli tehty joitain alustavia käyttöliittymäpiirroksia sovelluksesta erillisessä Demolan organisoimassa ideointiprojektissa. Näiden pohjalta aloitettiin MVP:n luominen. Ideointiprojektin tarkoituksena ei ollut liiketoimintamallin luominen, vaan keksiä mahdollisia ominaisuuksia järjestelmälle. Tämän vuoksi siis MVP-prosessi oli tärkeä projektille.

Seuraavaksi käsitellään tutkimuksessa ilmenneet havainnot. Ensiksi esitellään havainnot, jotka liittyvät prosessiin.

Tärkein havainto tutkimuksessa oli, että ohjelmoitu MVP on liian raskas menetelmä näin alkuvaiheessa olevalle projektille. Käyttöliittymäprototyypit toimivat huomattavasti paremmin sidosryhmien kanssa keskusteltaessa, kuin ohjelmoitu prototyyppi. Ilman ohjelmakoodia toteutettu käyttöliittymäprototyyppi pystyy esittämään saman asian, kuin oikea toteutus. Toiminnallisuuden tekeminen edes jollain tasolla käyttää lopulta paljon enemmän resursseja, kuin käyttöliittymäkuvien luominen. MVP:tä tehtäessä tulee välittömästi päättää ratkaisut moniin perustavanlaatuisiin kysymyksiin, kuten työkalujen ja teknologioiden valintoihin. Lisäksi, jos MVP:tä on tarkoitus käyttää validoinnin jälkeen, sitä tehtäessä tulee ottaa huomioon erinäisiä ominaisuuksia kuten ylläpidettävyys, tietoturva, saatavuus, laajennettavuus ja niin edelleen.

MVP:n kehitystä ohjasivat erilaiset ohjelmisto- ja liiketoimintamallien esittelyt mahdollisia asiakkaita ja muita sidosryhmiä varten. Esittelyitä varten MVP:hen valmisteltiin tiettyjä ominaisuuksia ja niiden perusteella valittiin seuraaviin tapaamisiin muita ominaisuuksia testattavaksi. MVP toimi eräänlaisena prototyyppinä ja visualisointina tulevasta tuotteesta, vaikka se ei sisältänytkään useita toiminnallisia ominaisuuksia. Toisaalta, myös käyttöliittymäkuvia käytettiin sovelluksen esittelyyn sidosryhmille. Ne herättivät samalla tavalla kiinnostusta tuotetta kohtaan, kuin MVP.

Arkkitehtuurisuunnittelu tapahtui osittain myös tapaamisten ohjaamana. Pienessä työryhmässä arkkitehtuurin suunnitteluun ei jäänyt aikaa enää siinä vaiheessa, kun MVP:hen täytyi saada uusia toimintoja. Toisaalta, kun sidosryhmätapaamisia ei ollut sovittuna, arkkitehtuuria voitiin tutkia ja pohtia enemmän.

Sprinttimuotoisen aikataulun seuraaminen osoittautui myös ongelmalliseksi. Pienen resurssimäärän vuoksi projektin tekijöiden oli mahdoton käyttää mitään yksityiskohtaista aikataulutusta. Projektissa oli suunniteltu käytettävän 0-sprintti arkkitehtuurisuunnittelun lähestymistapaa ja siihen oli varattu kuukausi. Aikataulutusta arkkitehtuurisprintille ennen ja vielä sprintinkin aikana oli vaikea ennustaa vakiintumattomien työaikojen vuoksi. Joinain hetkinä projektia pystyttiin tekemään paljon ja toisina hetkinä se taas lähes pysähtyi. Lopulta arkkitehtuurisprintti päättyi siihen, että eräs sidosryhmä ilmoitti haluavansa MVP:tä esiteltävän heille. Arkkitehtuurin lisäsuunnittelulle ei tässä vaiheessa enää ollut aikaa ja kaikki panos tuli laittaa toimivan järjestelmän tuottamiseksi. Näin epävakaassa projektissa 0-sprinttimenetelmä ei sellaisenaan siis toiminut odotetulla tai hallitulla tavalla.

Toinen ongelma arkkitehtuurisuunnittelun aikatauluttamisessa oli itse suunnittelun laajuuden arvioiminen. Aluksi projektia tekevillä ei ollut tietoa paljonko jonkin tietyn asian selvittämiseen kuluisi aikaa. Tällöin oli vaikea ennustaa olisiko esimerkiksi kahden viikon vai kuukauden pituinen 0-sprintti lyhyt tai pitkä aika suunnittelulle, vaikka työpanos olisikin pysynyt koko sen ajan vakiona. Toisaalta joidenkin perusasioiden selvittämiseen ei tarvinnut käyttää ollenkaan aikaa, sillä mielipiteet ja aiemmat kokemukset ratkaisivat niiden valinnan. Näin kävi esimerkiksi joidenkin ohjelmistokehysten kanssa. Toiset päätökset veivät huomattavasti enemmän aikaa, sillä ne vaativat uusien menetelmien opettelua. Uuden asian opetteluun käytetyn ajan arvioiminen on vaikeaa. Aiemmin epäonnistuneiden arviointien perusteella voidaan mahdollisesti kuitenkin luoda tarkempia arviointeja tulevaisuudessa. Tämän vuoksi jokin arviointi tehtävien kestolle voi olla hyväksi.

Tutkimuksessa huomattiin myös arkkitehtuurisuunnitteluun vaikuttavia tekijöitä.

MVP:tä tehdessä päätökset tapahtuivat projektin näkökulmasta epäsuotuisassa järjestyksessä. Esimerkiksi järjestelmän yleinen rakenne ja monet teknologiat, kuten pilvipalvelualustat ja ohjelmistokehykset, oli päätettävä projektin alussa, jotta verkossa toimiva MVP voitiin julkaista ja esittää sidosryhmille. Nämä olivat tärkeitä päätöksiä, sillä MVP:tä pyrittiin käyttämään jatkossa validoinnin jälkeen.

Projektin alussa joudutaan tekemään myös olettamuksia siitä, mitä ominaisuuksia arkkitehtuurilta vaaditaan. Jotkin ratkaisut voivat antaa arkkitehtuurille ominaisuuksia, jotka

hyödyttävät kehittäjiä juuri sillä hetkellä ja toiset taas vaikuttavat myöhemmin tulevaisuudessa. Esimerkiksi mahdollisimman laajennettava ja muokattava arkkitehtuuri voi olla työläs toteuttaa, mutta arkkitehtuuri jonka toteuttaminen vie vähän resursseja taas voi olla vaikeasti muokattavissa tai julkaistavissa. Tällainen valinta jouduttiin case-tutkimuksessa tekemään esimerkiksi muokattavan microservice-arkkitehtuurin ja nopeammin toteutettavan monoliittisen arkkitehtuurin valinnan välillä.

MVP:tä aloitettaessa joudutaan heti tekemään monta perustavanlaatuista valintaa sen arkkitehtuurista. Sitä vastoin projektin suunta ja tarkoitus on tässä vaiheessa vähiten tiedossa, vaikka jotkin tietyt toiminnot olivatkin MVP:hen jo valittu. Vasta liiketoimintamallin validoinnin jälkeen voidaan tietää ohjelmiston lopulliset toiminnot ja tarkoitus. Ennen sitä on aina mahdollista joutua tekemään pivot ja muuttamaan ohjelmistoa. Jos pivotissa muutetaan ohjelmiston toiminnallisuuksia paljon, niin arkkitehtuuria voidaan joutua muuttamaan ja työtä menee hukkaan. Esimerkiksi sovelluksen vaihto mobiilisovelluksesta verkkosivustoksi voi vaatia arkkitehtuurin uudelleen tekemistä. Yrityksen liiketoimintamalli voi myös jossain määrin vaikuttaa ohjelmistoon ja sen arkkitehtuurin. Muun muassa liiketoimintamalli voi aluksi perustua suoran hyödyn tuottamiseen asiakkaalle. Hänelle saatetaan luvata, että hän joutuu maksamaan tuotteesta vain, jos tuote saavuttaa jonkin merkkipaalun. Näitä merkkipaaluja täytyy mitata ohjelmistossa. Mittausta ja sen tuloksia voidaan joutua käsittelemään arkkitehtuurin tasolla, sillä se voi muodostua poikkiohjelmalliseksi ongelmaksi, ja se on kriittistä liiketoimininalle. Yrityksen liiketoimintamallin muokkaaminen tästä esimerkiksi perinteiseksi kertamaksulliseksi ohjelmistoksi voi tarkoittaa sitä, että edelliseen liiketoimintamallin vaatimiin ominaisuuksiin on kulutettu paljon resursseja tiedon keräämistavan, säilytystavan ja prosessoinnin suunnittelun sekä toteutuksen muodossa.

Riippuen MVP:n käyttötavasta Lean Startup -menetelmän validointivaihe piti ottaa arkkitehtuurisuunnittelussa huomioon. Mittaamiseen liittyvät ominaisuudet muodostuivat siis yhdeksi tärkeäksi arkkitehtuurin läpileikkaavaksi osaksi. Tämä vastasi läheisesti lokikirjanpidon luomista, säilyttämistä ja analysointia. Esimerkiksi tutkimuksen projektissa osa MVP:tä julkaistiin verkossa erikseen, minkä vuoksi käyttäjien reaktioita siihen ei voitu aina seurata paikan päällä. Erinäisten painikkeiden painalluskertoja tuli siksi erikseen tallentaa myöhempää analyysiä varten. Tallennuksen toteuttaminen vaati arkkitehtuurillista pohdintaa, jota ei muutoin olisi tarvinnut tehdä. Validointivaiheen huomioon ottaminen arkkitehtuurissa siis lisää työn määrää.

Arkkitehtuuriprototyypit osoittautuivat toimivaksi työkaluksi suunnitteluun tässä projektissa. Uusien teknologioiden ja ideoiden kokeilu prototyypin avulla antoi projektia tekeville paljon tarkemman kuvan niistä, kuin muunlainen tutkinta. Muun muassa kehysten vakaus ja tehokkuus tulivat ilmi vasta toimivaa ohjelmakoodia ajettaessa. Hyväksi koetusta arkkitehtuuriprototyypistä kehittyikin lopulta runko MVP:lle. Joka tapauksessa, jos jokin teknologia on uusi tai epäselvä kehittäjille, arkkitehtuuriprototyypin tai ohjelmarungon tekeminen auttaa kartoittamaan, sopiiko se lopulta tarkoituksiin ja kehittäjille. Ohjelmistokehyksen valinnan huomattiin olevan erittäin subjektiivinen prosessi. Kehittäjän omat

tottumukset ja taidot tietyistä toimintatavoista ohjaavat pitkälti valintaa.

Ohjelmistokehykset niin ikään auttoivat suunnittelussa huomattavasti. Projektissa toteutettiin verkkopalvelu, jonka tekemistä varten on olemassa kattavia kehyksiä. Ne tarjoavat yleisesti käytettyjä funktioita ja hyväksi koetun rakenteen. Kattavien kehysten vuoksi MVP:tä varten tarvittavan arkkitehtuurisuunnittelun määrä väheni merkittävästi. Kehysten suurin haaste oli niiden opetteluun kulunut aika. Kehyksistä tulee tietää ainakin niiden hyödyt, haitat ja kuinka ne voidaan yhdistää muihin komponentteihin, jotta ne voidaan valita tietoisesti. Joillain tietyillä ohjelmistokehityksen aloilla, kuten verkkopalvelut, on olemassa erittäin monta vaihtoehtoista ratkaisua. Näiden vertailu vaatii paljon tutkimista. Ohjelmistokehysten säästämät resurssit kuitenkin kompensoivat niiden opetteluun kuluvan ajan ja ne olivat yleisesti hyödyllisiä. Ohjelmistokehysten käyttö on siis suotavaa MVP:tä tehdessä.

Pienen resurssimäärän vuoksi arkkitehtuurin arviointimenetelmät osoittautuivat liian raskaiksi. Yhdistettynä tähän ja paineeseen luoda toimivaa ohjelmistoa, arviointimenetelmien käyttöön ei jäänyt aikaa. Toisaalta, ohjelmistokoodia ei tehty niin paljon, että suuret muutokset arkkitehtuuriin olisivat muodostuneet haastaviksi, jos niitä olisi pitänyt tehdä. Ehkä arviointimenetelmien käyttö olisi tullut tärkeämmäksi projektin edetessä.

Pienen työryhmän vuoksi vain suullinen keskustelu arkkitehtuurista oli yleensä tarpeellista. Erillistä dokumentaatiota ei varsinaisesti tarvittu suunnitelmien muistamiseksi tai jakamiseksi. Muutamissa tapauksissa kaavioiden piirtäminen auttoi silti havainnollistamaan suunnitelmia. Kokonaisen 4+1- tai C4-mallin tekeminen koettiin vievän liikaa aikaa, sillä monet näkymät määräytyivät suoraan jonkin ohjelmistokehyksen asettamalla tavalla. Sidosryhmät eivät myöskään olleet kiinnostuneita niistä. Arkkitehtuuripäätöskuvaukset osoittautuivat projektille niin ikään liian raskaiksi ja kankeiksi. Toisaalta, tutkimuksen seurantajakso ei ollut niin pitkä, että tärkeää arkkitehtuurillista tietoa olisi päässyt unohtumaan. Tämän vuoksi myöskään arkkitehtuurihaikujen ei koettu tuovan lisäarvoa.

9 TUTKIMUKSEN ANALYSOINTI JA KATTAVUUS

Työssä tehty kirjallisuusselvitys esittelee yleisesti käytettyjä ja uusia arkkitehtuurin suunnitteluun liittyviä konsepteja. Se ei kuitenkaan kata läheskään kaikkia menetelmiä. Kirjallisuudessa esiintyy näiden lisäksi monia menetelmiä, joita käytetään yleisesti sekä teollisuudessa että vain tutkimuksen tasolla. Niitä kaikkia ei tässä työssä pystytty esittelemään ja testaamaan.

Työn aikana suoritettu case-tutkimus tarkastelee vain yhtä projektia lyhyellä aikavälillä. Sen vuoksi se ei ole kovinkaan kattava ja siinä tehtyjä päätelmiä ei välttämättä voi yleistää jokaiseen pienen startup-yrityksen tapaukseen. Esimerkiksi dokumentaation merkitystä projektille pitkien aikajaksojen yli ja erilaisten menetelmien vertailua ei tässä työssä pystytty tekemään. Tulevaisuudessa startup-yrityksiä tulisi tutkia pidempikestoisesti ja useassa erilaisessa pienessä yrityksessä tai projektissa, jotta tässä työssä esitetyt väitteet voitaisiin osoittaa yleispäteviksi. Lisäksi menestyvien ja epäonnistuvien yritysten eroavaisuuksia tulisi tutkia, jotta voitaisiin selvittää niiden tuloksiin johtavia tekijöitä.

Pienen startup-yrityksen tulisi miettiä tarkkaan, onko MVP paras lähestymistapa liiketoimintasuunnitelman validoimiseksi ja maksavien asiakkaiden löytämiseksi. Vaikka tutkimus ei olekaan kovin laaja se osoittaa vahvasti, että MVP voi olla aloittelevalle yritykselle raskas prosessi. Muut menetelmät, kuten käyttöliittymäkuvat ja paperiprototyypit, yhdistettynä Lean Startup -metodologiaan voivat tuottaa samat tulokset nopeammin liiketoimintamallin validoimisessa, kuin ohjelmoitu MVP.

Jos MVP päätetään toteuttaa, arkkitehtuuria olisi hyvä tutkia edes hieman ennen MVP:n toteuttamista. Ainakin teknologioiden ja kehysten yhteensopivuus tulisi varmistaa ennen ohjelmoimista. Siihen voidaan käyttää esimerkiksi arkkitehtuuriprototyyppiä, joka sisältää arkkitehtuurillisesti tärkeimmät komponentit. Näin voidaan välttyä yllättäviltä ongelmilta toteutusvaiheessa.

Arkkitehtuuridokumentaation kirjoittaminen yksityiskohtaisesti ei ole tärkeää nopeasti toteutettavissa, työntekijämäärältään erittäin pienissä projekteissa. Tutkimuksessa ei kuitenkaan kyetty selvittämään dokumentaation tärkeyttä pitkällä aikatähtäimellä. Esimerkiksi olisi mielenkiintoista selvittää dokumentaation tärkeys tilanteessa, kun startup-yritys onnistuneesti lanseeraa tuotteensa markkinoille ja yritys kasvaa nopeasti. Tällöin yleensä yritykseen palkataan uusia työntekijöitä ja heidän täytyy ymmärtää sovelluksen arkkitehtuuri. Onko suullinen selitys siinä tilanteessa silti toimivin ratkaisu vai tulisiko tieto jakaa dokumentaation avulla?

Myöskin arkkitehtuurin arviointi vaikuttaisi olevan pienissä aloittelevissa startup yrityksissä hyödytöntä case-tutkimuksen perusteella, sillä arviointi vaatii suhteellisen paljon resursseja. Kirjallisuusanalyysin perusteella DCAR on kuitenkin startup yritykselle ATAM-menetelmää parempi vaihtoehto, silloin kun arkkitehtuuriarvioinnin koetaan olevan hyödyllistä. Tämä ajankohta olisi syytä selvittää jatkotutkimuksilla, jotta startup yritykset voisivat selkeämmin suunnitella ja ajoittaa arkkitehtuuriarvioinnit, ennen kuin se on liian myöhäistä. Mahdollisesti vaihe, jossa liiketoimintamalli on todettu toimivaksi, olisi syytä tehdä arkkitehtuuriarviointi.

10 YHTEENVETO

Arkkitehtuurisuunnittelua varten on olemassa useita hyödyllisiä työkaluja. Sekä kirjallisuustutkimus että työssä tehty case-tutkimus viittasivat siihen, että erityisesti arkkitehtuuriprototyypit ja ohjelmistokehykset ovat hyviä työkaluja MVP-tuotteen arkkitehtuuria luotaessa.

MVP-metodologia tuotekehityksessä vaikuttaa suunnitteluun merkittävästi. Siinä tuotetta kehitetään osissa ja iteratiivisesti. Kirjallisuustutkimuksen mukaan MVP:tä kehitettäessä arkkitehtuurisuunnittelulle tulisi varata hieman aikaa ennen toteutuksen alkua. Suunnittelu voidaan yhdistää prosessiin esimerkiksi arkkitehtuurisprintti-lähestymistapaa käyttäen, jossa ensimmäinen ketterä iteraatio omistetaan pelkästään arkkitehtuurisuunnittelulle. Tällä vältytään suurimmilta arkkitehtuurisuunnittelun laiminlyömisen aiheuttamilta ongelmilta. Pitkäkestoiset suunnitteluvaiheet taas vievät liikaa resursseja, vaikka niillä voitaisiinkin tehdä parempi arkkitehtuuri.

Pienen yrityksen resurssit asettavat rajoitteita arkkitehtuurisuunnittelulle ja MVP:n toteuttamiselle. Esimerkiksi arkkitehtuurin arviointi ei vaikuttaisi olevan mahdollista pienelle aloittelevalle startup-yritykselle. Ne vaativat useita osaajia, joita pienistä startup-yrityksistä ei välttämättä löydy.

Työssä huomattiin, että pienen startup-yrityksen ei välttämättä kannata testata liiketoimintamallinsa olettamuksia MVP-tuotetta käyttäen. Ei-ohjelmoituun prototyyppiin verrattuna se vaatii suhteellisen raskasta ohjelmointia, eikä kuitenkaan tuo riittävästi lisäetuja. Tutkimuksen perusteella on suotavaa käyttää ei-ohjelmoituja prototyyppejä tai käyttöliittymäkuvia aina, kun niillä voidaan tarpeeksi hyvin kuvata liiketoimintaidea tai tuote sidosryhmille.

Työssä tehty kirjallisuus ja case-tutkimus tutkii ohjelmistoarkkitehtuurin suunnittelua pienten startup-yritysten näkökulmasta ja pienellä aikavälillä. Sen vuoksi tässä työssä esitettyjä havaintoja ja päätelmiä ei voida yleistää kaikkiin Lean Startup -prosessia hyödyntäviin yrityksiin. Niitä voidaan kuitenkin käyttää harkiten ja projektiin soveltaen. Jatkossa pienten startup-yritysten toiminnan ja arkkitehtuurisuunnitteluprosessin tutkimusta tulisi laajentaa moniin pieniin yrityksiin. Näin saataisiin tietoa menestyvistä ja epäonnistuvista yrityksistä. Niiden eroavaisuuksia tulisi tutkia, jotta voitaisiin selvittää menestykseen tai epäonnistumiseen johtavia tekijöitä.

LÄHTEET

[1] P. Abrahamsson, M. A. Babar ja P. Kruchten. Agility and Architecture: Can They Coexist? *IEEE Software* 27.2 (maaliskuu 2010), 16–22. ISSN: 0740-7459. DOI: 10.1109/MS.2010.36.

[2] M. Ali Babar, A. W. Brown ja I. Mistrík. *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Amsterdam;Boston; Elsevier/Morgan Kaufmann, 2014.

[3] S. W. Ambler ja I. Books24x7. *Agile modeling: effective practices for eXtreme programming and the unified process*. 1. Aufl. New York: Wiley, 2002.

[4] Ó. Andri Ragnarsson. Importance of Design Patterns and Frameworks for Software Development (toukokuu 2019).

[5] A. Azanha, A. R. T. T. Argoud, J. B. d. Camargo Junior ja P. D. Antoniolli. Agile project management with Scrum. *International Journal of Managing Projects in Business* 10.1 (2017), 121–142.

[6] M. A. Babar, T. Dingsøyr, P. Lago ja H. v. Vliet. *Software Architecture Knowledge Management: Theory and Practice*. 1. Aufl. Berlin, Heidelberg: Springer-Verlag, 2009.

[7] J. E. Bardram, H. B. Christensen, A. V. Corry, K. M. Hansen ja M. Ingstrup. Exploring Quality Attributes Using Architectural Prototyping. Vol. 3712. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, 155–170.

[8] J. Bardram, H. Christensen ja K. Hansen. Architectural prototyping: An approach for grounding architectural design and learning. Heinäkuu 2004, 15–24. ISBN: 0-7695-2172-X. DOI: 10.1109/WICSA.2004.1310686.

[9] L. Bass ja R. L. Nord. Understanding the Context of Architecture Evaluation Methods. IEEE, 2012, 277–281.

[10] L. Bass, P. Clements ja R. Kazman. *Software architecture in practice*. 2nd. Boston: Addison-Wesley, 2003.

[11] S. Bellomo, I. Gorton ja R. Kazman. Toward Agile Architecture: Insights from 15 Years of ATAM Data. *IEEE Software* 32.5 (2015), 38–45.

[12] V. Berg, J. Birkeland, A. Nguyen-Duc, I. O. Pappas ja L. Jaccheri. Software startup engineering: A systematic mapping study. *The Journal of Systems Software* 144 (2018), 255–274.

[13] S. Brown. *The C4 Model for Software Architecture*. The C4 Model, verkkosivu. URL: https://www.infoq.com/articles/C4-architecture-model (viitattu 01. 05. 2019).

[14] B. Burns. *Lean Startup vs. Agile*. infoq.com, verkkosivu. URL: https://www.infoq.com/articles/C4-architecture-model (viitattu 02. 05. 2019).

[15] H. Cervantes, P. Velasco-Elizondo ja R. Kazman. A Principled Way to Use Frameworks in Architecture Design. *IEEE Software* 30.2 (2013), 46–53.

[16] H. B. Christensen ja K. M. Hansen. An empirical investigation of architectural prototyping. *The Journal of Systems Software* 83.1 (2010), 133–142.

[17] A. Dande, V.-P. Eloranta, H. Hadaytullah, A.-J. Kovalainen, T. Lehtonen, M. Leppänen, T. Salmimaa, M. Syeed, M. Vuori, C. Rubattel, W. Weck ja K. Koskimies. *Software Startup Patterns - An Empirical Study*. Tampere University of Technology. Department of Pervasive Computing, 2014.

[18] S. Denning. Updating the Agile Manifesto. *Strategy Leadership* 43.5 (2015).

[19] N. M. Edwin. Software Frameworks, Architectural and Design Patterns. *Journal of Software Engineering and Applications* 7.8 (2014), 670–678.

[20] V.-P. Eloranta. *Techniques and Practices for Software Architecture Work in Agile Software Development*. Tampere University of Technology, 2015.

[21] D. Falessi, G. Cantone, R. Kazman ja P. Kruchten. Decision-making techniques for software architecture design: A comparative survey. *ACM Computing Surveys (CSUR)* 43.4 (2011), 1–28.

[22] S. Farhan, H. Tauseef ja M. A. Fahiem. Adding Agility to Architecture Tradeoff Analysis Method for Mapping on Crystal. Vol. 4. IEEE, 2009, 121–125.

[23] M. Galster ja S. Angelov. Understanding the use of reference architectures in agile software development projects. *European Conference on Software Architecture*. Springer. 2015, 268–276.

[24] E. George. Agile Project Management: Scrum, eXtreme Programming, and Scrumban. Elsevier, 2016, 1–1. ISBN: 0128023228;9780128023228;

[25] S. Gerdes, S. Jasser, M. Riebisch, S. Schröder, M. Soliman ja T. Stehle. Towards the essentials of architecture documentation for avoiding architecture erosion. ACM, 2016, 1–4.

[26] I. Hadar, S. Sherman, E. Hadar ja J. J. Harrison. Less is more: Architecture documentation for agile development. IEEE, 2013, 121–124.

[27] U. van Heesch, V.-P. Eloranta, P. Avgeriou, K. Koskimies ja N. Harrison. Decision-Centric Architecture Reviews. *IEEE Software* 31.1 (2014), 69–76.

[28] C. Hofmeister, R. L. Nord ja D. Soni. Global Analysis: moving from software requirements specification to structural views of the software architecture. *IEE Proceedings - Software* 152.4 (2005), 187.

[29] *How it works - Working with and being part of the Topcoder Community*. Topcoder, verkkosivu. URL: https://www.topcoder.com/how-it-works/faqs/ (viitattu 24. 05. 2019).

[30] S. Hussain, J. Keung, A. A. Khan ja K. E. Bennin. Correlation between the Frequent Use of Gang-of-Four Design Patterns and Structural Complexity. IEEE, 2017, 189–198.

[31] IBM. *Document architectures by using the C4 model*. IBM, verkkosivu. URL: https://www.ibm.com/cloud/garage/practices/code/c4-model-for-software-architecture (viitattu 01. 05. 2019).

[32] A. Jansen, P. Avgeriou ja J. S. van der Ven. Enriching software architecture documentation. *The Journal of Systems Software* 82.8 (2009), 1232–1248.

[33] R. E. Johnson ja B. Foote. Designing Reusable Classes. 2001.

[34] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson ja J. Carriere. The architecture tradeoff analysis method. *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*. IEEE. 1998, 68–78.

[35] R. Kazman, M. Klein ja P. Clements. *ATAM: Method for architecture evaluation*. Tekninen raportti. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000.

[36] M. Keeling. Architecture haiku: a case in lean documentation. *IEEE Software* 32.3 (2015), 35.

[37] P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software* 12.6 (1995), 42–50.

[38] P. Kruchten, P. Lago ja H. van Vliet. Building Up and Reasoning About Architectural Knowledge. Vol. 4214. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 43–58.

[39] K. R. Lakhani, D. A. Garvin ja E. Lonstein. Topcoder (a): Developing software through crowdsourcing. *Harvard Business School General Management Unit Case* 610-032 (2010).

[40] T. D. LaToza ja A. van der Hoek. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software* 33.1 (2016), 74–80.

[41] L. Lee ja P. Kruchten. Capturing Software Architectural Design Decisions. IEEE, 2007, 686–689. ISBN: 0840-7789.

[42] T. C. Lethbridge, J. Singer ja A. Forward. How software engineers use documentation: the state of the practice. *IEEE Software* 20.6 (2003), 35–39.

[43] S. C. Lo ja N. Chen. *IEEE 42010 and Agile Process- Create Architecture Description through Agile Architecture Framework*. Copyright - Copyright The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) 2017; Last updated - 2019-02-27. 2017. URL: https://libproxy.tuni.fi/login?url=https://search-proquest-com.libproxy.tuni.fi/docview/2139493407?accountid=14242.

[44] Y. Mansoori. Enacting the lean startup methodology. *International Journal of Entrepreneurial Behavior Research* 23.5 (2017), 812–838.

[45] K. Mao, L. Capra, M. Harman ja Y. Jia. A survey of the use of crowdsourcing in software engineering. *The Journal of Systems Software* 126 (2017), 57–84.

[46] M. Marchesi, G. Succi ja SpringerLink. *Extreme Programming and Agile Processes in Software Engineering: 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003, Proceedings*. Vol. 2675;2675.; Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[47] S. Meng, Z. Pan, W. Li, S. Xie, C. Liu, K. He ja H. Yang. The "4+1"view model on safe home system architecture. IEEE, 2010, 352–355. ISBN: 2327-0586.

[48] D. Minoli ja I. Books24x7. *Enterprise Architecture A to Z: Frameworks, Business Process Modeling, SOA, and Infrastructure Technology*. 1. painos. London: Auerbach Publications, 2008.

[49] O. P. *Learn about Lean Startup*. Open class rooms, verkkosivu. URL: https://openclassrooms.com/en/courses/4544561-learn-about-lean-startup/4716471-learn-how-and-when-to-pivot (viitattu 07. 05. 2019).

[50] D. Parnas. *On the criteria to be used in decomposing systems into modules*. 1972.

[51] D. E. Perry ja A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), 40–52.

[52] E. Ries. *The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses*. New York: Crown Business, 2011. ISBN: 9780307887894;0307887898;

[53] M. Rouse. *Definition framework*. Techtarget, verkkosivu. URL: https://whatis.techtarget.com/definition/framework (viitattu 01. 05. 2019).

[54] E.-M. Schön, J. Thomaschewski ja M. J. Escalona. Agile Requirements Engineering: A systematic literature review. *Computer Standards Interfaces* 49 (2017), 79–91.

[55] K.-J. Stol ja B. Fitzgerald. Two's company, three's a crowd: a case study of crowdsourcing software development. *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, 187–198.

[56] D. Sturtevant. Modular Architectures Make You Agile in the Long Run. *IEEE Software* 35.1 (2018), 104–108.

[57] *Survey of Architecture Frameworks*. iso-architecture.org, verkkosivu. 2018. URL: http://www.iso-architecture.org/ieee-1471/afs/frameworks-table.html.

[58] A. Tang, M. A. Babar, I. Gorton ja J. Han. A Survey of the Use and Documentation of Architecture Design Rationale. IEEE, 2005, 89–98.

[59] A. Tang, P. Avgeriou, A. Jansen, R. Capilla ja M. Ali Babar. A comparative study of architecture knowledge management tools. *The Journal of Systems Software* 83.3 (2010), 352–370.

[60] H. Terho, S. Suonsyrjä ja K. Systä. Understanding the Relation between Iterative Cycles in Software. *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017. ISBN: 978-0-9981331-0-2.

[61] *THE TOGAF® STANDARD, VERSION 9.2*. The Open Group, verkkosivu. 2019. URL: http://www.intel.com/content/www/us/en/history/historic-timeline.html (viitattu 05. 07. 2019).

[62] O. Vogel, I. Arnold, A. Chughtai ja T. Kehrer. *Software Architecture : A Comprehensive Framework and Guide for Practitioners*. Berlin, Heidelberg: Springer, 2011.

[63] M. Waterman. Agility, Risk, and Uncertainty, Part 1: Designing an Agile Architecture. *IEEE Software* 35.2 (2018), 99–101.

[64] D. Weyns, R. Mirandola, I. Crnkovic, I. för datavetenskap (DV), Linnéuniversitetet ja F. för teknik (FTK). *Software Architecture : 9th European Conference, ECSA*

2015, Dubrovnik/Cavtat, Croatia, September 7-11, 2015. Proceedings. 1st 2015. Vol. 9278. Cham: Springer International Publishing AG, 2015.

[65] A. Zalewski ja S. Kijas. Beyond ATAM: Early architecture evaluation method for large-scale distributed systems. *The Journal of Systems Software* 86.3 (2013), 683–697.