Jari Laari

OHJELMISTOARKKITEHTUURI JA SEN SUUNNITTELU: TAPAUSTUTKIMUKSENA TUOTANTOTEHOKKUUDEN SEURANTA- JA KUNNONVALVONTAJÄRJESTELMÄN ARKKITEHTUURI

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2014

TIIVISTELMÄ

Laari, Jari
Ohjelmistoarkkitehtuuri ja sen suunnittelu: tapaustutkimuksena tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän arkkitehtuuri
Jyväskylä: Jyväskylän yliopisto, 2014, 94 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaaja: Leppänen, Mauri

Tänä päivänä omakotitaloa – saati sitten pilvenpiirtäjää – ei lähdetä rakentamaan ilman kunnollisia piirustuksia. Valitettavasti sama käytäntö ei ole vielä yhtä järjestelmällisesti rantautunut ohjelmistoteollisuuteen. Vaikka ohjelmistokehitys eroaa monilta osin talonrakennuksesta, arkkitehtuurisuunnittelun tulee olla olennainen osa ohjelmistokehitystyötä. Ohjelmistoarkkitehtuurin suunnittelulle on esitetty monia suunnittelumenetelmiä. Sen sijaan menetelmien käytöstä todellisissa ohjelmistoarkkitehtuurin suunnitteluhankkeissa on olemassa vain vähän tutkimustietoa.

Tutkimuksen tarkoituksena on selvittää, millä tavalla voidaan valita ja soveltaa ohjelmistoarkkitehtuurin suunnittelumenetelmää ja arvioida tuloksena saatua ohjelmistoarkkitehtuuria. Tutkimuksessa ohjelmistoarkkitehtuuria, sen tavoitteita, kuvaustapoja, arkkitehtuurityylejä sekä suunnittelu- ja arviointimenetelmiä tutkitaan ensin kirjallisuuskatsauksen avulla. Tämän jälkeen työssä toteutetaan tapaustutkimus, jossa valitun arkkitehtuurin suunnittelumenetelmän (ADD) avulla suunnitellaan tapaustutkimuksen kohteena olevalle ohjelmistolle nykyaikainen, uudet tarpeet täyttävä, arkkitehtuuri. Lopuksi tuotettua arkkitehtuuria arvioidaan käyttämällä valittua arviointimenetelmää (ATAM) ja vertaamalla tuotettua arkkitehtuuria vanhaan arkkitehtuuriin laadullisten ominaisuuksien näkökulmasta.

Tutkimus osoittaa, että ADD-menetelmä soveltuu tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän tapaisten järjestelmien arkkitehtuurin suunnitteluun. Saadun arkkitehtuurin todetaan palvelevan kohdeorganisaation tarpeita. Tutkimusprosessia ja -mallia esitetään hyödynnettäväksi vastaavankaltaisissa tutkimushankkeissa. Suunniteltua arkkitehtuuria ehdotetaan käytettäväksi myös muiden teollisen internetin sovellutuksien arkkitehtuurin pohjana.

Tutkimus kannustaa ohjelmistoarkkitehtuurin suunnitteluun ja tarjoaa tietoa, kuinka ohjelmistoarkkitehtuurin suunnittelua voidaan toteuttaa ohjelmistokehitysprojekteissa. Tulokset tarjoavat myös hyviä lähtökohtia jatkotutkimukselle.

Asiasanat: ohjelmistoarkkitehtuuri, ohjelmistoarkkitehtuurin suunnittelu, ADD, ohjelmistoarkkitehtuurin arviointi, ATAM, tapaustutkimus

ABSTRACT

Laari, Jari
Software Architecture and its Design: A Case Study
Jyväskylä: University of Jyväskylä, 2010, 94 p.
Information Systems Science, Master's thesis
Supervisor: Leppänen, Mauri

Nowadays, it is not reasonable to build a house without first making proper designs for it. Unfortunately, the same is not true, to the same extent, in software engineering. Even if software engineering differs from house building in many respects, architecture design should be an essential part of the software development process. In the literature, a number of methods have been published for software architecture design. However, there is a scarcity of research on the use of these design methods in practice.

The purpose of this study is to find out how to choose and apply a software architecture design method and evaluate the outcomes. We first make a literature review of software architecture, architectural styles as well as architecture design and evaluation methods. Based on this, we conduct a case study in which one architecture design method (ADD) is selected, adapted and utilized to design a new software architecture for the certain legacy software. We evaluate the outcome by using one software architecture evaluation method (ATAM) and compare it to the existing architecture in terms of non-functional requirements.

The study shows that the ADD method can be applied to design, in an iterative manner, an architecture for systems similar to the target system in the study. Based on the evaluation, the new architecture is considered to satisfy needs of the organization. The research process and model built for this study are suggested to be worth considering in similar kinds of research endeavors. The new architecture could be used as a generic architecture for Internet of Things (IoT) applications.

This study encourages designing software architecture and provides information about how software architectures can be designed in practice. The results provide a good basis for further research.

Keywords: software architecture, software architecture design, ADD, software architecture evaluation, ATAM, case study

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Ohjelmistoarkkitehtuurit ovat jo verrattain vanha asia. Ohjelmistot olivat jo 1960-luvulla niin laajoja, että niiden ohjelmistokehitykseen piti ottaa mukaan rakenteen suunnittelua, jotta ohjelmistot olisivat laajennettavissa tulevaisuudessa. Dijkstra (1968) esittää aiheesta yhden varhaisimmista tutkimuksista, jossa kerrotaan ohjelmiston jakamisesta abstraktiotasoihin. Kruchtenin ym. (2006) mukaan ensimmäinen havainto ohjelmistoarkkitehtuuri-käsitteen käytöstä on vuodelta 1969 NATO:n järjestämässä konferenssissa. Ohjelmistojen edelleen laajentuessa ja monimutkaistuessa ohjelmistoarkkitehtuurit ja niiden huomioon ottaminen ohjelmistokehityksessä ovat tulleet yhä kriittisemmiksi. Liian monessa yrityksessä ohjelmistoarkkitehtuuriin kiinnitetään huomiota vasta, kun on liian myöhäistä. Ohjelmisto on saattanut vuosien saatossa rakentua pala kerrallaan ilman, että on kiinnitetty huomiota kokonaisuuteen.

Ohjelmistoarkkitehtuurilla tarkoitetaan kuvausta, josta käyvät ilmi ohjelmiston osat, niiden keskinäiset suhteet ja suhteet ympäristöön, sekä periaatteet, jotka ohjaavat ohjelmiston suunnittelua ja evoluutiota (International Organization for Standardization, 2011; Koskimies & Mikkonen, 2005). Se on järjestelmän perustuslaki, jota noudattamalla mahdollistetaan ohjelmiston selkeä rakenne. Ohjelmiston arkkitehtuurin rakentamisessa voidaan hyödyntää aiemmin hankittua tietoa. Arkkitehtuurityyli (*architectural style*) määrää järjestelmän kokonaisrakenteen. Se on malli, jonka tarkoituksena on kuvata, millä tavoin järjestelmä organisoidaan eri abstraktiotasoilla (Koskimies & Mikkonen, 2005). Arkkitehtuurityylit määrittävät, minkälaisia komponentteja ja suhteita kannattaa tietynlaisessa kohdeongelmassa käyttää. Ne ovat eräänlaisia standardiratkaisujen kuvauksia. Tunnettuja arkkitehtuurityylejä ovat esimerkiksi kerrosarkkitehtuuri (Garlan & Shaw, 1993) ja asiakas-palvelin arkkitehtuuri (Buschmann ym., 1995).

Hyvään ohjelmistoarkkitehtuuriin ei päädytä useinkaan sattumalta, vaan se on huolellisen suunnittelun tulos. Ohjelmistoarkkitehtuuriin suunnitteluun on kehitetty erilaisia suunnittelumenetelmiä. Tällaisia ovat esimerkiksi ominaisuusvetoinen arkkitehtuurisuunnittelu (Attribute-Driven Design, ADD) (Bachmann & Bass, 2001), Siemensin neljän näkymän suunnittelumenetelmä (Soni

ym., 1995) ja RUP:n 4+1:n näkymän suunnittelumenetelmä (Kruchten, 1995; Kruchten, 2004). Osa suunnittelumenetelmistä on kehitetty teollisuudessa, osa osin akateemisen työn tuloksena (Hofmeister ym., 2007). Monet suunnittelumenetelmistä on tarkoitettu laajojen ohjelmistojen arkkitehtuurien suunnitteluun. Ne edellyttävät usein laajaa etukäteissuunnittelua. Ketterän ohjelmistokehityksen kannattajat pitävät raskasta suunnittelua paheksuttavana. Sen pelätään johtavan laajaan dokumentaatioon, kasvaneeseen työmäärän ja paluuseen takaisin vanhaan ja kankeaan vesiputousmalliin. Jotkin suunnittelumenetelmät ovat raskaita käyttää, koska ne vaativat eri sidosryhmien aktiivista osallistumista prosessiin (Kruchten, 2010; Nord & Tomayko, 2006). Tästä syystä kirjallisuudessa on keskusteltu paljon siitä, miten arkkitehtuurin suunnittelua voidaan tehdä ketterän ohjelmistokehityksen yhteydessä (Faber, 2010; Nord & Tomayko, 2006; Kruchten, 2010).

Ohjelmistoarkkitehtuurin laatua tulee arvioida niin suunnitteluprosessin aikana kuin sen päätteeksikin. Koskimiehen ja Mikkosen (2005) mukaan ohjelmistoarkkitehtuurin arviointi perustuu arkkitehtuurikomponenttien ja alijärjestelmien suhteiden sekä ominaisuuksien arviointiin. Koskimiehen ja Mikkosen (2005) sekä Clementsin ym. (2003) mukaan arkkitehtuurin arvioinnissa keskeisintä on ohjelmiston laadullisten ominaisuuksien arviointi. Reekien ym. (2006) mukaan laadulliset ominaisuudet voidaan jakaa kahteen luokkaan: ajonaikaisiin (*runtime*) ja ei-ajonaikaisiin (*non-runtime*). Ajonaikaisiin ominaisuuksiin kuuluvat suorituskyky, käytettävyys, luotettavuus ja turvallisuus. Ei-ajonaikaisiin kuuluvat ylläpidettävyys, testattavuus, uudelleenkäytettävyys, konfiguroitavuus ja laajennettavuus.

Ohjelmistoarkkitehtuurin arviointia varten on kehitetty arviointimenetelmiä, jotka johdonmukaistavat arviointityötä. Suosittuja arviointimenetelmiä ovat esimerkiksi SEI-instituutissa kehitetyt ATAM (*Architecture Tradeoff Analysis Method*) (Kazman ym., 1998), SAAM (*Architecture Tradeoff Analysis Method*) (Kazman ym., 1994) ja ARID (*Architecture Tradeoff Analysis Method*) (Clements, 2000), joiden avulla virheitä ohjelmistoarkkitehtuurin suunnittelussa voidaan vähentää (Clements ym., 2003). Babarin ym. (2004) mukaan ohjelmistoarkkitehtuurin arviointimenetelmät tarjoavat suuntaviivoja, heuristiikoita ja standardeja, joita noudattamalla arkkitehtuurin toimivuus kohdeongelmassa voidaan varmistaa. Kirjallisuudessa on julkaistu ohjelmistoarkkitehtuurien arviointimenetelmien vertailuja (esim. Babar ym., 2004). Vaikka arviointimenetelmiä pidetään kustannustehokkaina (Clements ym., 2003), on osa niistä raskaita käyttää.

Ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmä tulee valita ja räätälöidä kulloistenkin tarpeiden ja rajoitteiden mukaisesti. Hofmeister ym. (2007) vertailevat viittä eri teollisuudessa kehitettyä suunnittelumenetelmää ja muodostavat sen pohjalta yhden yleistyksen ohjelmistoarkkitehtuurin suunnittelumenetelmäksi. Babar ym. (2004) ovat esittäneet vertailutaulukkoja arviointimenetelmistä, jotka voivat helpottaa sopivan arviointimenetelmän valintaa.

Kuten edellä olevasta käy ilmi, on käsitteellis-teoreettista tutkimusta ohjelmistoarkkitehtuureista, niiden suunnittelusta ja arvioinnista tehty varsin paljon. Sen sijaan empiiristä tutkimusta suunnittelumenetelmien soveltamisesta

käytännössä on tehty selvästi vähemmän. Esimerkkeinä tällaisista ovat Kazmanin ym. (1998) ja Barbaccin ym. (2003b) tapaustutkimukset, joissa on arvioitu ATAM-arviointimenetelmän (Kazman ym., 1998) sopivuutta ja toimivuutta. Jotta saadaan kokemuksia suunnittelu- ja arviointimenetelmien toimivuudesta, tarvitaan enemmän empiirisiä tutkimuksia erilaisista tilanteista. Vain tällä tavalla voidaan saada luotettavaa tietoa ja uusia ideoita menetelmien edelleen kehittämiseksi.

Tässä tutkielmassa tarkastellaan ohjelmistoarkkitehtuurin suunnittelua ja arviointia sekä tutkimuksen että käytännön näkökulmasta. Tämän tutkimuksen tutkimusongelma on:

> *Millä tavalla voidaan valita ja soveltaa ohjelmistoarkkitehtuurin suunnittelumenetelmää ja arvioida tuloksena saatua ohjelmistoarkkitehtuuria*?

Tutkimusongelma on jaettavissa seuraaviin tutkimuskysymyksiin:

- Mitä tarkoitetaan ohjelmistoarkkitehtuurilla ja millaisia ohjelmistoarkkitehtuurityylejä on olemassa?
- Millaisia ohjelmistoarkkitehtuurin suunnittelumenetelmiä on esitetty ja millä perusteilla niitä voidaan valita sovellettavaksi?
- Miten ohjelmistoarkkitehtuuria voidaan arvioida?
- Miten ohjelmistoarkkitehtuurin suunnittelumenetelmää voidaan soveltaa käytännössä?

Tutkimuksen tavoitteena on rakentaa ohjelmistoarkkitehtuureja ja niiden suunnittelua ja arviointia koskeva käsitteellinen perusta ja hyödyntää tätä käytännön ohjelmistoarkkitehtuurin suunnitteluprosessissa. Käsitteellis-teoreettinen osuus perustuu aiemmin tehtyyn tutkimukseen. Käytännön osuus toteutetaan tapaustutkimuksena (Yin, 2009; Runeson & Höst, 2009). Tapaustutkimuksen kohteena on tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän arkkitehtuurin suunnittelu- ja arviointiprosessi. Tässä tapauksessa nykyinen ohjelmistoarkkitehtuuri päivitetään vastaamaan paremmin nykyisiä sekä tulevaisuuden tarpeita. Anturipohjainen järjestelmä mittaa tuotantokoneiden toimintatilaa, tuotantotehokkuuden tunnuslukuja (*Overall equipment effectiveness*, OEE) ja koneiden kunnossapidon tunnuslukuja. Antureiden mittaama tieto prosessoidaan käyttäjäystävälliseen muotoon ja esitetään käyttäjille hyödyntäen modernia tietoliikennetekniikkaa.

Suunnittelun tuloksena syntyy ohjelmistoarkkitehtuuri, jota voidaan käyttää kohdeorganisaatiossa päätöksenteon ja suunnittelun tukena. Tehty arkkitehtuurisuunnitelma tarjoaa objektiivisen näkemyksen siitä, mihin suuntaan nykyistä ohjelmistoratkaisua tulisi viedä. Tämän lisäksi organisaatioon syntyy tutkimuksen myötä ymmärrystä ja ammattitaitoa ohjelmistoarkkitehtuurista. Luotua arkkitehtuurikuvausta voidaan käyttää laajemminkin teollisen internetin sovellutuksissa.

Tutkimus jakautuu seitsemään lukuun. Tutkielman teoreettinen osuus (luvut 2-3) vastaa kolmeen ensimmäiseen tutkimuskysymykseen. Empiirisen osuuden (luvut 4-6) avulla vastataan viimeiseen tutkimuskysymykseen. Luvus-

sa 2 esitetään ensin, mitä ohjelmistoarkkitehtuurilla tarkoitetaan ja tavoitellaan ja miten sitä voidaan kuvata. Sen jälkeen esitetään arkkitehtuurityylien kategorioinnteja ja kuvataan kuusi arkkitehtuurityyliä. Luvussa 3 käsitellään ohjelmistoarkkitehtuurin suunnittelua ja arviointia. Ensin esitetään yleiskuvaus ohjelmistoarkkitehtuurin suunnittelusta ja sen jälkeen kuvataan kahta suunnittelumenetelmää. Toiseksi kerrotaan yleisesti ohjelmistoarkkitehtuurin arvioinnista ja sen jälkeen kuvataan yhtä arviointimenetelmää. Lopuksi kerrotaan, miten ohjelmistoarkkitehtuurin suunnitteluun suhtaudutaan ketterässä ohjelmistokehityksessä.

Empiirinen osa alkaa luvusta 4. Luvussa kuvataan, empiirisen tutkimuksen tutkimusmenetelmä, tutkimusprosessi ja -malli ja tiedonkerääminen. Luvussa 5 esitetään tapaustutkimuksen tulokset. Ensin kuvataan olemassa oleva ohjelmisto ja sen arkkitehtuuri. Sen jälkeen kerrotaan, miten suunnittelumenetelmä valittiin ja miten sitä sovellettiin. Kolmanneksi kuvataan uusi arkkitehtuuri, arvioidaan sitä ja vertaillaan vanhaa ja uutta arkkitehtuuria. Luvussa 6 esitetään tapaustutkimuksen tulokset tiivistettyinä, verrataan niitä aiempiin tutkimuksiin, esitetään johtopäätökset, kerrotaan, kuinka tutkimuksen tuloksia voidaan hyödyntää ja lopuksi tarkastellaan tutkimuksen reliabiliteettia ja validiteettia. Tutkimus päättyy yhteenvetoon.

2 OHJELMISTOARKKITEHTUURI

Tässä luvussa esitellään ensin, mitä ohjelmistoarkkitehtuurilla tarkoitetaan, mitä sillä tavoitellaan ja mitä sen kuvaaminen tarkoittaa. Sen jälkeen esitetään arkkitehtuurityylien kategoriointeja ja kuvataan kuusi eri arkkitehtuurityyliä: kerrosarkkitehtuurit, tietovuoarkkitehtuurit, asiakas-palvelin-arkkitehtuurit, viestinvälitysarkkitehtuurit, malli-näkymä-ohjain-arkkitehtuurit ja tulkkipohjaiset arkkitehtuurit. Luku päättyy yhteenvetoon.

2.1 Määritelmä

Varhaisimpia merkkejä ohjelmistoarkkitehtuureista on esitelty Dijkstran (1968) tutkimuspaperissa, jossa kerrotaan, kuinka ohjelmistoa abstraktiotasoihin jakamalla voidaan luoda sille esitettävää rakennetta. Tutkiessaan käyttöjärjestelmiä hän esittelee idean ohjelmiston jakamisesta tasoihin, jotka voivat kommunikoida vain naapuritasojensa kanssa. Dijkstran (1968) kuvausta voidaan pitää yhtenä varhaisimmista merkeistä nykyaikaisesta ohjelmistoarkkitehtuurin määritelmästä. (Bass ym., 2003; Koskimies & Mikkonen, 2005.)

Ohjelmistoarkkitehtuurien kuvausta koskevassa standardissa IEEE 1471 (2011) annetaan määritelmä ohjelmistoarkkitehtuurille. Koskimies ja Mikkonen (2005, s. 18) ovat suomentaneet standardin määritelmän seuraavasti: [ohjelmistoarkkitehtuuri on perusorganisaatio] ”joka sisältää osat, niiden keskinäiset suhteet ja niiden suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota”. Standardissa määritellään lisäksi, kuinka ohjelmistoarkkitehtuuri voidaan kuvata luotettavalla tavalla sekä mitkä ovat keskeiset käsitteet ja sanasto ohjelmistoarkkitehtuurin alueella. Standardissa ei määritellä yhtään pakollista näkymää, joka täytyisi olla kaikissa arkkitehtuurikuvauksissa. Syynä tähän on standardointityöryhmän erimielisyydet ja näkymien ja kuvaustekniikoiden sovelluskohtaisuus. (International Organization for Standardization, 2011.)

Clements ym. (2002) selventävät IEEE 1471 standardin määritelmää toteamalla, että ohjelmistoarkkitehtuuri on joukko järjestelmää kuvaavia rakenteita, jotka koostuvat ohjelmistoelementeistä, niiden ominaisuuksista ja keskinäisistä suhteista.

Koskimies ja Mikkonen (2005) painottavat arkkitehtuurin tärkeää merkitystä koko ohjelmiston rakenteen kannalta: [ohjelmistoarkkitehtuuri on] ”järjestelmän perustuslaki, jota on noudatettava järjestelmää rakennettaessa. Perustuslakia saa muuttaa vain painavilla perusteilla” (Koskimies & Mikkonen 2005, s. 18).

Yhteenvetona voidaan todeta *ohjelmistoarkkitehtuurin* kuvaavan abstraktilla tasolla ohjelmiston rakenneosia ja niiden välistä vuorovaikutusta. Ohjelmistoarkkitehtuuri luo yleiskuvan tavoiteltavasta järjestelmästä piilottaen suunnittelun yksityiskohdat. Se muodostaa järjestelmän suunnittelun pohjan ja ohjaa järjestelmän jatkokehitystä pitkälle tulevaisuuteen. Ohjelmistoarkkitehtuuri ei ole yhteen kuvaustapaan tai tyyliin sidottu, vaan se voi olla hyvinkin erilainen käyttökohteesta riippuen.

2.2 Tavoite

Jokaisella ohjelmistolla on arkkitehtuuri. Kukaan ei ole ehkä suunnitellut tai dokumentoinut sitä, mutta se on syntynyt huomaamatta pala kerrallaan ohjelmistokehityksen edetessä. Bassin ym. (2003) mukaan ei ole ehdottoman hyvää tai huonoa ohjelmistoarkkitehtuuria. Arkkitehtuuri on ainoastaan enemmän tai vähemmän käyttötarkoitukseensa sopiva. Esimerkiksi lyhyttä pilotointia varten rakennettu järjestelmä ei tarvitse skaalauntuvaa ja helposti ylläpidettävää työlästä arkkitehtuuriratkaisua. Vasta 1990-luvulla on alettu paremmin ymmärtämään ohjelmistoarkkitehtuurin vaikutus ohjelmiston laatuun ja ohjelmistokehityksen läpimenoaikaan ohjelmistojen koon kasvaessa sekä järjestelmien monimutkaistuessa entisestään (Bosch, 2000).

Ohjelmistoarkkitehtuurin avulla järjestelmän toteutus ja toiminta on mahdollista jakaa osiin. Osiin jakaminen mahdollistaa toteutettavan järjestelmän eri osien tehokkaan työstämisen yhtäaikaisesti. Tällöin ohjelmistokehittäjät eri organisaatioista, eri aikavyöhykkeiltä tai eri maista voivat samanaikaisesti työskennellä kohti yhteistä tavoitetta ilman riippuvuutta muiden tekemisestä. Keskeisessä osassa on arkkitehtuurin avulla tehtävä rajapintasuunnittelu, mikä määrittää kuinka tietoa siirretään järjestelmän tai sovelluksen eri osien välillä. (Clements ym., 2002.)

Boschin (2000) mukaan hyvällä ohjelmistoarkkitehtuurisuunnittelulla voidaan vähentää ohjelmistokehityksessä syntyviä kustannuksia, parantaa ohjelmiston jatkokehitystä ja ylläpidettävyyttä sekä vähentää tuotteen kehitykseen kuluvaa aikaa (*time-to-market*). Esimerkiksi arkkitehtuurillisesti huonosti määritelty rajapinta voi aiheuttaa viivästyksiä kehitystyöhön tilanteessa jossa rajapinnan kehittäjä ja käyttäjä toteuttavat toiminnallisuutta samanaikaisesti. Täl-

löin rajapinnan käyttäjän ja kehittäjän oletukset eivät vastaa toisiaan, mikä voi aiheuttaa mittavaa tarvetta ohjelmiston uudelleenkirjoitukselle.

Bosch (2000) tiivistää ohjelmistoarkkitehtuurin tarkoituksen kolmeen näkökulmaan. Hänen mukaansa ohjelmistoarkkitehtuuri:

- kontrolloi ohjelmiston laatua ennen varsinaisen ohjelmistokehityksen alkamista,
- esittää ohjelmistosta konkreettisia malleja, joita voidaan hyödyntää keskusteluissa sidosryhmien kanssa, ja
- määrittää arkkitehtuuriset komponentit ja niiden välisen vuorovaikutuksen, mikä edesauttaa uudelleenkäytettävyyttä.

Ohjelmistoarkkitehtuurista on hyötyä useille eri ryhmille. Bachmann ym. (2000) ovat määritelleet ohjelmistoarkkitehtuurin käyttökohteita eri käyttäjäryhmien mukaan. Esimerkiksi projektipäälliköille ohjelmistoarkkitehtuuri mahdollistaa resurssien suunnittelun ja kohdistamisen, kun taas ohjelmistokehittäjille ohjelmistoarkkitehtuuri sanelee ohjelmistokomponenttien toteutusjärjestystä. Jansen (2008) on tiivistänyt ohjelmistoarkkitehtuurin käyttökohteet ja käyttötarkoitukset ohjelmistokehitysprosessin eri vaiheissa seuraavasti:

- *Suunnitelma*: Ohjelmistoarkkitehtuurin tärkein tarkoitus on rakentaa ja kuvata ohjelmiston ääriviivat. Perusrakennetta voidaan jatkojalostaa tarkempiin yksityiskohtiin, joiden pohjalta ohjelmiston rakentaminen on mahdollista.
- *Pitkän tähtäimen suunnitelma (roadmap)*: Ohjelmistoarkkitehtuuri mahdollistaa keskipitkän ja pitkän ajan suunnitelmien tekemisen, joiden avulla yritys voi varmistaa teknisen etumatkan markkinoilla.
- *Kommunikoinnin väline*: Korkean tason ohjelmistoarkkitehtuuri piilottaa suunnittelun yksityiskohdat, jolloin se voi toimia kommunikoinnin välineenä ohjelmistokehittäjien ja eri sidosryhmien (esim. asiakas, johto, rahoittajat) välillä. Näin ollen isompi määrä ihmisiä voi osallistua ohjelmistoarkkitehtuurin suunnitteluun ja parantamiseen.
- *Työnjakaja*: Ohjelmistoarkkitehtuuri jakaa järjestelmää pienempiin itsenäisesti toteutettaviin palasiin. Tämä mahdollistaa ohjelmistokehittäjien rinnakkaisen työskentelyn ohjelmistokehityksen eri vaiheissa.
- *Laadun ennakoija*: Ohjelmistoarkkitehtuurin pohjalta voidaan aikaisessa vaiheessa ennakoida tulevan ohjelmiston laatua. Ohjelmiston suunnitteluvaiheessa mahdollisten suunnitteluvirheiden tunnistaminen ja korjaaminen on huomattavasti helpompaa ja halvempaa kuin myöhemmissä vaiheissa.

2.3 Ohjelmistoarkkitehtuurin kuvaaminen

Clements ym. (2002) määrittelevät dokumentoinnin, eli kuvaamisen, dokumentaation tuottamiseksi. Näin ollen dokumentoinnin tuloksena syntyy konkreetti-

sia asioita, kuten elektronisia tiedostoja, web-sivuja, piirustuksia ja diagrammeja. Olennainen osa on asioiden jalostaminen muotoon, jota tutkimalla muut ihmiset ymmärtävät tehdyt ratkaisut.

Koskimies ja Mikkonen (2005, s 29) toteavat, että ohjelmistoarkkitehtuurikuvaus on keskeinen dokumentti, jossa arkkitehtuuri materialisoituu. Ohjelmistoarkkitehtuuri tulee kuvata siten, että muut voivat onnistuneesti käyttää sitä, ylläpitää sitä ja rakentaa ohjelmiston sen pohjalta (Clements ym., 2002 s. 31). Koska Clementsin ym. (2002) 582-sivuinen teos rakennetaan tämän kysymyksen pohjalle, käsitellään tässä ohjelmistoarkkitehtuurin kuvaamista vain hyvin pintapuolisesti.

Ohjelmistoarkkitehtuurin dokumentoinnin tarkoituksena on kertoa ohjelmistoarkkitehdin tekemät ratkaisut yksiselitteisesti. Ilman kunnollista dokumentaatiota ratkaisua ei oikeastaan ole edes olemassa, koska muut eivät voi käyttää sitä. Arkkitehtuuri tulee kuvata siten, että eri sidosryhmät voivat vaivattomasti löytää siitä tarvitsemansa informaation. (Clements ym., 2002.)

Yleensä dokumentaatiota tuotetaan koska on pakko. Dokumentaation tuottamistarve voi tulla asiakkaalta, johdolta tai yrityksen käyttämistä laatustandardeista. Usein dokumentaatiota tuotetaan vain dokumentoinnin tarpeeseen, ei esimerkiksi siksi, että se auttaa yritystä luomaan laadukkaita tuotteita ja vähentämään ohjelmistokehitykseen kuluvaa aikaa. Ohjelmistoarkkitehtuurin kuvaus tuleekin nähdä kaikkia osapuolia hyödyttävänä asiana: arkkitehdit voivat sen avulla ylläpitää ja vetää ohjelmiston suuntaviivoja, johto voi sen avulla kohdentaa resursseja ja asiakkaat osallistua ohjelmiston suunnitteluun. (Clements ym., 2002.)

Jansen (2008) on listannut tapoja, miten ohjelmistoarkkitehtuuri voidaan kuvata:

- *Luonnollinen kieli*: Useimmiten ohjelmistoarkkitehtuuri kuvataan suullisesti ja kirjallisesti tekstin muodossa. Tekstin avulla voidaan tehtyjä ratkaisuja perustella kattavasti.
- *Malli:* Mallit tarjoavat muotin, jonka avulla ohjelmistoarkkitehtuuria voidaan kuvata. Malli määrittää elementit ja vuorovaikutussuhteet, jotka ohjelmistoarkkitehtuurissa tulee kuvata. Tällä tavoin ohjelmistoarkkitehtuurin suunnittelu voidaan toteuttaa johdonmukaisesti.
- *Diagrammit*: Erilaiset kuviot mahdollistavat hankalien asiayhteyksien välisten yhteyksien esittämisen yksinkertaisesti. Kuvioiden käyttö on kätevää etenkin erilaisten abstraktiotasojen kuvaamisessa.
- *Kuvat:* Hahmotelmia ja kuvia voidaan käyttää havainnollistamaan ja selittämään käytettyjä käsitteitä.
- *Formaali kieli:* Ohjelmistoarkkitehtuuria voidaan kuvata myös formaalisti, esimerkiksi metamallien avulla. Metamallissa määritellään käsitteitä, suhteita ja merkityksiä, joiden avulla ohjelmistoarkkitehtuuri voidaan kuvata.

Ohjelmistoarkkitehtuuri materialisoidaan arkkitehtuurin kuvaustekniikoiden avulla. Kuvaustekniikat varmistavat sen, että tehdyt suunnitelmat käyttävät

ennalta määriteltyjä sääntöjä ja notaatioita. Näin kaikki suunnitelmia käyttävät ymmärtävät tarkalleen, mitä kullakin notaatiolla tarkoitetaan eri yhteyksissä. Koskimiehen ja Mikkosen (2005) mukaan kuvaustekniikat liittyvät läheisesti työkalutukeen. Työkalujen avulla voidaan tuottaa arkkitehtuurimalleja ja -näkymiä, mutta työkalut varmistavat myös, että tehdyt mallit ovat keskenään ristiriidattomia. Tunnetuimpia tekniikoita ohjelmistoarkkitehtuurien kuvauksessa ovat UML (*Unified Modeling Language)* (Booch ym., 1996), AADL (*Architecture Analysis & Design Language*) (SAE Technical Standards Board, 2004) ja SysML (*Systems Modeling Language*) (Clements ym., 2002; SysML Merge Team, 2006).

2.4 Arkkitehtuurityylejä

2.4.1 Määritelmiä ja kategoriointeja

Koskimiehen ja Mikkosen (2005) mukaan *arkkitehtuurityyli* (*architectural style*) määrää järjestelmän kokonaisrakenteen. Se on malli, jonka tarkoituksena on kuvata, millä tavoin järjestelmä organisoidaan eri abstraktiotasoilla. Shawin Garlanin (1996) mukaan arkkitehtuurityyli määrittelee komponentit, niiden väliset välikappaleet sekä säännöt, kuinka niitä voidaan yhdistellä. Clementsin ym. (2002) mukaan arkkitehtuurityyli esittää valittua arkkitehtuurillista lähestymistapaa. Heidän mukaan arkkitehtuurityyli vastaa uudelleenkäytön tarpeeseen. Jossakin yhteydessä tehty laaja suunnittelupäätös voidaan jalostaa arkkitehtuurityyliksi, jota voidaan käyttää laajemmin eri asiayhteyksissä.

Buschmannin ym. (1995) mukaan *suunnittelumalli (design pattern*) määrittelee alijärjestelmiä, niiden vastuita, sääntöjä ja ohjeita, joiden avulla voidaan esittää alijärjestelmien välisiä suhteita. Kirjoittajien mukaan suunnittelumallit helpottavat ohjelmiston perusrakenteen määrittelytyötä. Esimerkiksi kehitystyössä jokainen ohjelmistokomponentti noudattaa sovittua suunnittelumallia, jonka mukaan sille määritellään alijärjestelmät ja rajapinnat muihin ohjelmistokomponentteihin. Tällä tavoin ohjelmiston rakenne muodostuu selkeäksi ja ylläpidettäväksi. Koskimiehen ja Mikkosen (2005, s. 102) mukaan ”suunnittelumallit ovat tunnettujen, käytännössä hyväksi havaittujen ratkaisujen kuvauksia yleisiin ohjelmistojen suunnittelua koskeviin ongelmiin tietyissä tilanteissa”, eli eräänlaisia standardiratkaisujen kuvauksia.

Buschmannin ym. (1995) tekemä määritelmä suunnittelumalleista on hyvin samantapainen kuin Koskimiehen ja Mikkosen (2005) määritelmä arkkitehtuurityyleistä. Koskimies ja Mikkonen (2005, s. 125) selventävät arkkitehtuurityylin ja -suunnittelumallin eroa: ”Itse asiassa ei ole aina aivan selvää, milloin tietty ratkaisuperiaate on suunnittelumalli ja milloin arkkitehtuurityyli, erityisesti silloin kun jokin suunnittelumalli – esimerkiksi Tarkkailija-malli – yleistetään arkkitehtuurin perustaksi”. He selventävät eroa siten, että suunnittelumalli ratkaisee paikallisia ongelmia yhdenmukaisesti, kun taas arkkitehtuurityyli ku-

vaa koko järjestelmää. Koskimiehen ja Mikkosen (2005) mukaan arkkitehtuurityyli syntyy, kun suunnittelumalli skaalataan koko järjestelmän arkkitehtuurin kantavaksi periaatteeksi. Näin ollen rajanveto suunnittelumallien ja arkkitehtuurityylien välillä on hyvin häilyvä. Tässä tutkielmassa suunnittelumalleja käsitellään koko järjestelmän laajuudessa, jolloin voidaan puhua arkkitehtuurityyleistä. Myös Shaw ja Garlan (1996) myöntävät teoksessaan, että ero suunnittelumallin ja arkkitehtuurityylin välillä on hyvin hankalasti määriteltävissä. Myöhemmin arkkitehtuurityylin ja suunnittelumallin käsitteiden välille on tehty erilaisia rajanvetoja (esim. Clements ym. (2002), mutta käsitteistöä ei ole standardisoitu.

Arkkitehtuurityylejä voidaan luokitella esimerkiksi niiden toiminnan, ominaisuuksien ja ideologian tai käyttötarkoituksen pohjalta. Seuraavassa kerrotaan tarkemmin Koskimiehen ja Mikkosen (2005) sekä Buschmannin ym. (1995) luokitteluista.

Koskimies ja Mikkonen (2005) ovat jakaneet arkkitehtuurityylit kolmeen luokkaan, jossa kategorointi perustuu arkkitehtuurityylien samankaltaiseen rakenteeseen. Englanninkieliset käännökset ovat Buschmannin ym. (1995) teoksesta. Kategorointi on esitetty alla.

- Osittavat arkkitehtuurityylit
    - Kerrosarkkitehtuurit (*The Layers*)
    - Tietovuoarkkitehtuurit (*The Pipes and Filters*)
- Palveluihin perustuvat arkkitehtuurityylit
    - Asiakas-palvelin-arkkitehtuurit (*Client-Server architecture*)
    - Viestinvälitysarkkitehtuurit (*The Broker pattern)*
- Erikoisarkkitehtuurityylit
    - Malli-näkymä-ohjain-arkkitehtuurit (*Model-View-Controller, MVC*)
    - Tulkkipohjaiset arkkitehtuurit (*Reflection architecture*)

Buschmann ym. (1995) ovat esittäneet kategorointia ohjelmistoarkkitehtuurin suunnittelumalleille, johon kuitenkin sisältyy myös Koskimiehen ja Mikkosen (2005) määrittelemiä arkkitehtuurityylejä. Buschmann ym. (1995) esittävät neljä kategoriaa: jäsentävät-, hajautetut-, vuorovaikutteisuutta tukevat ja mukautuvat arkkitehtuurit. Buschmannin ym. (1995) esittämä kategorointi perustuu sekä samankaltaiseen ideologiaan (*jäsentävät arkkitehtuurit*, *hajautetut arkkitehtuurit*), että luokitteluun käyttötarkoituksen perusteella (*vuorovaikutteisuutta tukevat*, *mukautuvat arkkitehtuurit*). Alla on yhteenveto kategorioihin kuuluvista arkkitehtuureista. Suluissa olevat arkkitehtuurit kuuluvat kirjoittajien mukaan vain osittain kyseisen kategorian alle.

- Jäsentävät arkkitehtuurit *(From mud to Structure)*
    - Kerrosarkkitehtuurit
    - Tietovuoarkkitehtuurit
    - Liitutauluarkkitehtuurit (*Blackboard architecture*)
- Hajautetut arkkitehtuurit (*Distributed systems*)
    - Viestinvälitysarkkitehtuurit

    - (Tietovuoarkkitehtuurit)
    - (Ydinarkkitehtuurit) (*Microkernel system*)
- Vuorovaikutteisuutta tukevat arkkitehtuurit (*Interactive Systems*)
    - Malli-näkymä-ohjain-arkkitehtuurit
    - Presentaatio-Abstraktio-Kontrolli-arkkitehtuurit (*Presentation-Abstraction-Control, PAC*)
- Mukautuvat arkkitehtuurit (*Adaptable Systems*)
    - Ydinarkkitehtuurit
    - Tulkkipohjaiset arkkitehtuurit

Buschmannin ym. (1995) mukaan jäsentäviin arkkitehtuureihin kuuluvat kerrosarkkitehtuuri, tietovuoarkkitehtuuri ja liitutauluarkkitehtuuri. Näistä kaksi ensimmäistä kuuluu Koskimiehen ja Mikkosen (2005) esittämään ”osittavat arkkitehtuurityylit”-kategoriaan. Buschmann ym. (1995) määritelmän mukaan tämän kategorian tyylit tai mallit pyrkivät hierarkiaa rakentamalla jäsentelemään kuvattavan järjestelmän. Tämä tapahtuu jakamalla kuvattavaa asiaa pienempiin – helposti hallittaviin – paloihin. Näin ollen kategoria vastaa täysin Koskimiehen ja Mikkosen (2005) ”osittavat arkkitehtuurityylit”-kategorian ajatusta.

Hajautettujen arkkitehtuurien luokkaan kuuluvat Buschmannin ym. (1995) mukaan ainoastaan viestinvälitysarkkitehtuuri, mutta se viittaa myös toisiin kategorioihin luokiteltuihin tietovuoarkkitehtuuriin, ydinarkkitehtuuriin. Tämän luokan tyylit sopivat hajautetun järjestelmän arkkitehtuurin pohjaksi.

Vuorovaikutteisuutta tukevien arkkitehtuurien joukkoon kuuluvat Buschmann ym. (1995) mukaan malli-näkymä-ohjain-arkkitehtuurit ja presentaatio-abstraktio-kontrolli-arkkitehtuurit. Molemmat näistä arkkitehtuurityyleistä tukevat järjestelmän kuvaamista, joka on vastuussa kommunikoinnista käyttäjän kanssa. Pääajatus molemmissa arkkitehtuurityyleissä on erottaa käyttöliittymä ohjelman varsinaisesta toiminnasta. Tällainen arkkitehtuuriajattelu mahdollistaa erilaisen käyttäjäkokemuksen tarjoamisen eri käyttäjäryhmille puuttumatta kuitenkaan järjestelmän perustoiminnallisuuteen. Sittemmin tämä arkkitehtuurityyliluokittelu on saanut seurakseen malli-näkymä-esittäjä (*Model-View-Presenter, MVP)*, malli-näkymä-mallinäkymä (*Model, View, ViewModel, MVVM*) sekä malli-näkymä-* (*Model-View-Whatever, MVW tai MV*)* arkkitehtuurityylit, jotka pohjautuvat ja ovat kehittyneet malli-näkymä-ohjain-arkkitehtuurista (Potel, 1996; Anderson, 2012). Näiden uusien arkkitehtuurityylien voidaan katsoa kuuluvan vuorovaikutteisuutta tukeviin arkkitehtuurityyleihin, vaikkei Buschmann ym. (1995) ole niitä määritellyt. (Buschmann ym., 1995.). Etenkin moni nykyaikaisista web-käyttöliittymien rakentamiseen tarkoitetuista JavaScript-käyttöliittymäkirjastoista pohjautuu MV*-arkkitehtuurityyliin. MV*-arkkitehtuurityyli määrittelee komponenteiksi ainoastaan mallin ja näkymän, jolloin ohjelmistokehittäjän tai kirjaston vastuulle jää määrittää muut tarvittavat komponentit.

Mukautuviin arkkitehtuureihin kuuluvat ydinarkkitehtuuri ja tulkkipohjaiset arkkitehtuurit. Ne tukevat ohjelmiston jatkuvaa laajentumista, tekniikoi-

den vaihtumista ja järjestelmälle asetettujen vaatimusten muuttumista. (Buschmann ym., 1995.)

Shaw ja Garlan (1996) määrittelevät vielä kolmannen tavan luokitella arkkitehtuurityylejä. Heidän luomansa kategoriointi pohjaa arkkitehtuurityylien käyttökohteisiin: *tietovirtajärjestelmät* (*dataflow systems*), *kutsu-ja-palautus-järjestelmät* (*call-and-return systems*), *komponenttikeskeiset järjestelmät* (*indepedent components*), *virtuaalikoneet* (*virtual machines*) ja *tietokeskeiset järjestelmät* (*data-centered systems*).

Seuraavassa esitetään kuusi arkkitehtuurityyliä Koskimiehen ja Mikkosen (2005) luokittelua mukaillen. Lähteinä on käytetty Buschmannin ym. (1995), Koskimiehen ja Mikkosen (2005) sekä Shawin ja Garlanin (1996) teoksia ohjelmistoarkkitehtuureista.

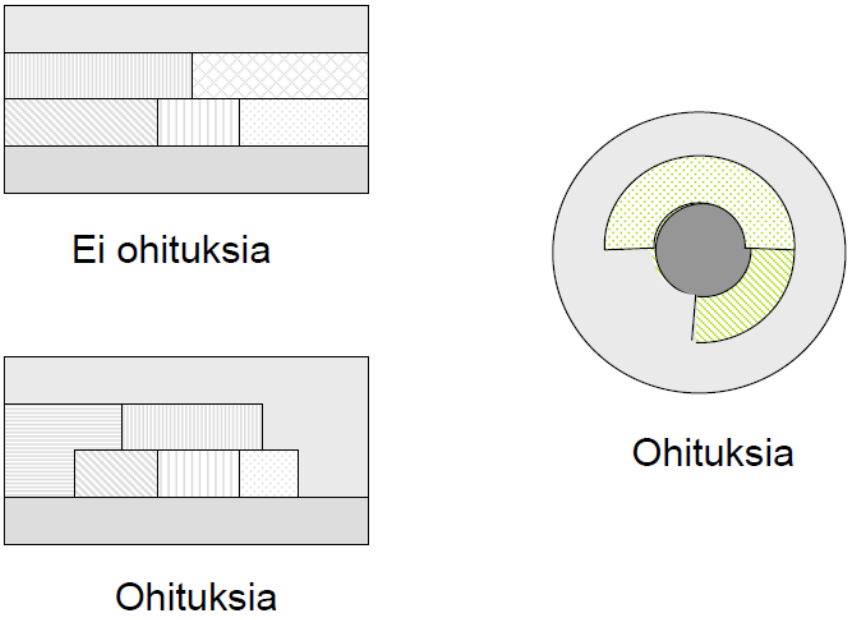
2.4.2 Kerrosarkkitehtuurit

Kerrosarkkitehtuuri on organisoitu tasoihin jonkin valitun absktratiotasojärjestelmän mukaan. Ylemmällä tasolla olevat komponentit käyttävät hyväkseen alemmalla tasolla olevia komponentteja. Kerrosarkkitehtuuri on hyvä tapa lähestyä ongelmaa, jossa suuren järjestelmän toiminnallisuutta täytyy jakaa pienempiin – ja helpommin toteutettaviin – osiin. Ihannetilanteessa jokainen kerros tarjoaa selkeän ja staattisen rajapinnan muiden tasojen käytettäväksi. Kerros voi käyttää muiden tasojen rajapintoja hyväkseen alemmilta toteutustasoilta. Tällöin kerroksen sisäistä toteutusta voidaan muuttaa, kunhan tarjottu rajapinta pysyy muuttamattomana. Näin voidaan esimerkiksi yhden kerroksen suorituskykyä parantaa puuttumatta muiden kerrosten toimintaan. (Koskimies & Mikkonen, 2005; Buschmann ym., 1995.)

Shawin ja Garlanin (1996) mukaan jotkin kerrosarkkitehtuurit ovat sellaisia, jossa kerros on tietoinen ainoastaan alemmasta kerroksesta. Hyvin usein kerrosarkkitehtuuri ei ole täysin ”puhdas”, vaan alemmalla tasolla oleva kerros voi tarvita ylemmän tason palveluita. Joissakin tapauksissa joudutaan kerroksia ohittamaan, jolloin esimerkiksi 1. tason kerros (ylempi) tarvitsee 3. tason kerroksen palveluita (alempi). Kerrosten ohittaminen voi tehdä ohjelmistoarkkitehtuurista monimutkaisemman, mutta ohittaminen voi olla välttämätöntä esimerkiksi ohjelmiston suorituskyvyn takia. Kuvio 1 esittää puhtaan kerrosarkkitehtuurin (ei ohituksia) sekä kaksi tilannetta, jossa ohituksia tapahtuu. (Koskimies & Mikkonen, 2005.)

Tietoliikenneprotokollat tarjoavat paljon esimerkkejä kerrosarkkitehtuureista. Tietoliikenneprotokollat ovat tarkasti määriteltyjä ja niiden toiminta pohjautuu hyvin usein johonkin alemman tason protokollaan. Tiedon esitysmuoto, sisältö ja tarkoitus on kaikille viesteille määritelty. Jokainen taso huolehtii omasta tehtävästään viestinvälityksessä käyttäen alemman tason palveluja. Protokollan sisäistä toteutustapaa (toteutuskieli, rakenne, algoritmit) voidaan muuttaa, kunhan protokolla tarjoaa samat palvelut muille protokollille (kerroksille). Kansainvälinen standardisoimisjärjestö ISO (*International Organization for Standardization*) on määritellyt OSI (Open Systems Interconnection Reference

Model) -mallin (1996), joka kuvaa tiedonsiirtoprotokollien seitsemän kerroksen arkkitehtuurin. (Bachmann & Bass, 2001.)



Kuvio 1: Kerrosarkkitehtuurin graafisia esityksiä (Koskimies & Mikkonen, 2005, s. 127)

Buschmann ym. (1995) selventävät kerrosarkkitehtuurin hyötyjä:

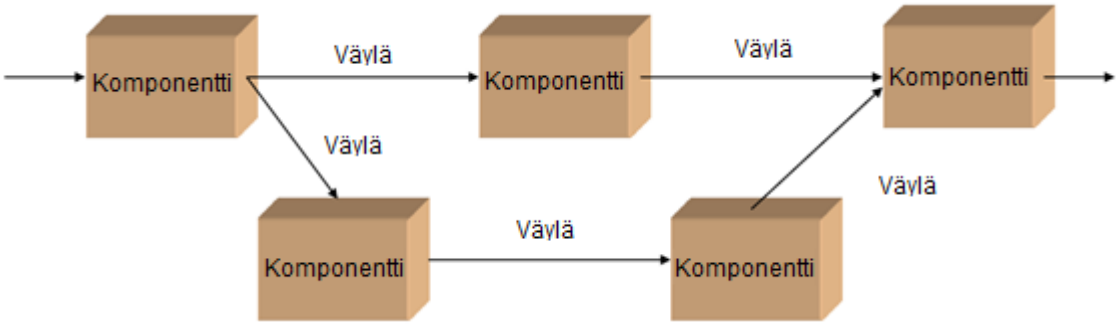
- *Kerrosten uudelleenkäytettävyys*: Ohjelmistokomponenttien uudelleenkäyttö on yksi ohjelmistokehityksen perusperiaatteista. Hyvin dokumentoitua ja toteutettua arkkitehtuurikerrosta voidaan uudelleen käyttää muissakin kuin alkuperäisessä yhteydessä.
- *Paikalliset riippuvuudet:* Hyvin määritellyt rajapinnat mahdollistavat kerroksen sisäisen uudelleen toteutuksen. Kerrokset voidaan suunnitella esimerkiksi siten, että muutos laitteistossa tai käyttöjärjestelmässä vaikuttaa vain yhden ylemmän kerroksen toteutukseen. Tämä mahdollistaa järjestelmän laajennettavuuden ja siirrettävyyden muihin yhteyksiin sekä kerrosten testauksen riippumatta muista kerroksista. Shaw ja Garlan (1996) toteavat, että muutos yhteen kerrokseen vaikuttaa parhaassa tapauksessa enintään kahteen muuhun kerrokseen. Yleensä vaikutuksen alla ovat käsiteltävän kerroksen ylä- ja alapuolella olevat kerrokset. Toisaalta muutosvaikutus muihin kerroksiin on suuri kerrosarkkitehtuurissa, jossa tapahtuu paljon kerrosten välisiä ohituksia.
- *Vaihdettavuus:* Ideaalitilanteessa yksi kerros voidaan vaihtaa kokonaan toisenlaiseen toteutukseen, jos se tarjoaa samankaltaisen toiminnallisuuden. Klassinen esimerkki on laitteen liittäminen tietoverkkoon. Tietokoneen liittäminen verkkoon voi tapahtua langallisesti tai langattomasti lukuisin erilaisin menetelmin, mutta kaikki yhteystavat tarjoavat samankaltaisen toiminnallisuuden käyttöjärjestelmälle ja muille ohjelmille, jotka toimivat tiedonsiirtokerroksen yläpuolella.

Vastaavasti Buschmann ym. (1995) selventävät kerrosarkkitehtuurin haittapuolia:

- *Vesiputousefekti:* Tasojen toiminnan muuttuminen voi aiheuttaa hallitsemattoman muutostarpeen myös muihin tasoihin. Esimerkiksi tilanteessa, jossa alhaisella tasolla tehty oletus tarvittavasta tiedonsiirtonopeudesta periytetään ylemmille tasoille, aiheuttaa muutostarvetta kaikkialle, mikäli alemmalla tasolla tiedonsiirtonopeutta joudutaankin jostakin syystä nostamaan.
- *Tehokkuus:* Kerrosarkkitehtuuri on usein tehottomampi kuin rakenne, jossa kaikki komponentit ovat välittömästi saavutettavissa (*sea of components*). Tasojen välillä tapahtuvaa tiedon siirto ja muuntaminen sopivaan muotoon ovat resursseja vaativia tapahtumia.
- *Suunnittelun vaikeus:* Vähäinen tasojen määrä ei mahdollista kaikkia kerrosarkkitehtuurin hyötyjen saavuttamista, mutta toisaalta liian suuri tasomäärä aiheuttaa monimutkaisuutta ja suoritustehottomuutta. Sopiva tasojen määrän suunnittelu kohdeongelmaan on haastavaa, mutta erittäin tärkeää ohjelmiston laadun kannalta.

2.4.3 Tietovuoarkkitehtuurit

Tietovuoarkkitehtuurin osat ovat itsenäiset komponentit (*filter*), jotka toimivat prosessointiyksikköinä, sekä niitä yhdistävät väylät (*pipe*). Buschmann ym. (1995) täydentävät määritelmää toteamalla, että komponentit lukevat tietovirtaa ja jatkojalostavat sen kuljetettavaksi seuraavalle väylälle. Jokainen komponentti toimii itsenäisenä osana isompaa järjestelmää. Ketjuttamalla komponentteja ja niitä yhdistäviä väyliä saadaan annetulla syötteellä haluttu lopputulos. Koskimies ja Mikkonen (2005) täsmentävät vielä, että tilattomasti toimivat komponentit eivät tunne tosiaan, vaan niiden välinen vuorovaikutus tapahtuu täysin väylien varassa. Kuvio 2 havainnollistaa komponenttien ja väylien toiminnan tietovuoarkkitehtuurissa.



Kuvio 2: Tietovuoarkkitehtuurin perusajatus (mukaillen Koskimies & Mikkonen, 2005 s. 132)

Koskimiehen ja Mikkosen (2005) mukaan tietovuoarkkitehtuuri sopii tilanteisiin, joissa järjestelmän toiminta on pääsääntöisesti tietovirtojen jalostamista ja prosessointia.

Tietovuoarkkitehtuurit tarjoavat merkittäviä hyötyjä. Alla oleva esitys perustuu seuraaviin lähteisiin: Shaw ja Garlan (1996), Buschmann ym. (1995) sekä Koskimies ja Mikkonen (2005). Tietovuoarkkitehtuurien hyötyjä ovat:

- *Ei tarvetta tiedon välivarastoinnille:* Kommunikointi erillisten komponenttien ja väylien välillä on mahdollista toteuttaa ilman tiedon hidasta ja virhealtista välivarastointia tiedostojärjestelmään. Välivarastointi ja väliaikaistiedon tutkiminen on kuitenkin mahdollista niin sanotun T-risteystoteutuksen avulla, jossa tieto haarautetaan useammalle komponentille (Buschmann ym., 1995).
- *Tiedon jalostus askel kerrallaan:* Järjestelmän toiminta on helposti ymmärrettävissä yksinkertaisia askelia seuraamalla (Shaw & Garlan, 1996). Koskimies ja Mikkonen (2005) täydentävät toteamalla, että vaikeasti toteutettavat ja hankalasti ymmärrettävät tietojenkäsittelytehtävät voidaan tietovuoarkkitehtuurin avulla ymmärtää ja toteuttaa hallitusti.
- *Uudelleenkäytettävyys:* Tietovuoarkkitehtuuri tukee uudelleenkäytettävyyttä. Tällöin on periaatteessa mahdollista yhdistää mitkä tahansa kaksi komponenttia toisiinsa. Tietovuoarkkitehtuuri sallii komponenttien vapaan uudelleen sijoittelun järjestelmän eri osiin (Shaw & Garlan, 1996).
- *Rinnakkaislaskennan tehokkuus:* Tietovuoarkkitehtuuri tukee luonnostaan rinnakkaista suoritusta. Jokainen komponentti toimii itsenäisesti, joten ne voivat voi suorittaa tehtäviä rinnakkaisesti (Shaw & Garlan, 1996).
- *Helppo kehitystyö ja ylläpidettävyys:* Johtuen komponenttien rajoitetusta riippuvuudesta toisiinsa nähden, Shaw ja Garlan (1996) esittävät, että tietovuoarkkitehtuurin päälle rakennettu järjestelmä on helposti laajennettavissa ja ylläpidettävissä. Uusia komponentteja on vaivatonta lisätä, koska muutokset vaikuttavat vain hyvin vähän muihin komponentteihin. Olemassa olevien komponenttien päivitys tai korvaaminen on myös helposti mahdollista, jos komponentin käyttämät tai tarjoamat ulkoiset palvelut eivät muutu.

Uudelleenkäytettävyys ja rinnakkaislaskennan tehokkuus nostetaan kaikissa kolmessa teoksessa keskeisiksi hyödyiksi. Nämä tekevät siitä sopivan esimerkiksi ohjelmointikielen kääntäjän arkkitehtuuriksi (Koskimies & Mikkonen, 2005). Koskimies ja Mikkonen (2005) huomauttavat, ettei tietovuoarkkitehtuuri sovi interaktiivisten järjestelmien rungoksi sen tilattoman luonteen takia. Buschmann ym. (1995) selventävät tietovuoarkkitehtuurin haittapuolia:

- *Tilatiedon jakaminen:* Tietovuoarkkitehtuuri ei rakenteeltaan sovellu tilatiedon jakamiseen. Näin ollen esimerkiksi ison asetustietomäärän levittäminen komponenteille on kallista ja joustamatonta.
- *Rinnakkaislaskennan tehokkuus on usein illuusiota:* Suurin osa komponenteista kuitenkin käsittelee syötteen kokonaan ennen kuin lähettää sitä paloittain eteenpäin. Lisäksi tiedonsiirto väylien välityksellä voi olla hidasta, jolloin parempi ratkaisu olisi toteuttaa kaikki toiminnallisuus yhdessä komponentissa.

- *Tiedon muuttamisen tehottomuus:* Unix-järjestelmän putkioperaattori perustuu väylien käyttöön. Esimerkiksi ohjelma, joka käyttää unix-järjestelmän putkioperaattoria laskutoimituksien toteuttamiseen, joutuu taustalla oleva toteutuksessa muuttamaan ASCII-merkkejä numeroiksi ja toisin päin jokaisessa komponentissa.
- *Virheiden käsittely:* Virheidenkäsittely on syytä toteuttaa koko järjestelmän laajuudessa ennalta sovitulla strategialla. Virheiden tunnistaminen, niistä palautuminen ja tiedottaminen muille komponenteille on haastavaa, koska arkkitehtuurin toiminta perustuu yksisuuntaisiin viestinvälitysketjuihin.

2.4.4 Asiakas-palvelin arkkitehtuurit

Asiakas-palvelin-arkkitehtuurissa pääkomponentit ovat resurssien hallitsija (palvelin) ja resurssien käyttäjä (asiakas). Asiakas voi kutsuilla pyytää palvelimelta haluttua resurssia, jonka toimittamisesta asiakkaalle palvelin vastaa itsenäisesti. Näin palvelin kätkee asiakkaalta toiminnan varsinaisen toteutuksen.

Koskimiehen ja Mikkosen (2005) mukaan arkkitehtuurin toiminta perustuu usein istuntoihin (*session*). Palvelimen tehtävä on yleensä passiivisena odottaa asiakkaan yhteydenottoa. Yhteydenmuodostamisen jälkeen suoritetaan palvelimella asiakkaan pyytämä palvelukokonaisuus. Tämän jälkeen istunto päätetään asiakkaan tai palvelimen toimesta. Asiakkaat ja palvelimet toimivat erillisissä prosesseissa, mikä mahdollistaa toiminnan hajauttamisen.

Buschmann ym. (1995) luokittelevat asiakas-palvelin arkkitehtuurin viestinvälitysarkkitehtuurin alle, jonka erityistapauksena se toimii.

Koskimies ja Mikkonen (2005) pitävät asiakas-palvelin-arkkitehtuuria yleisimmin käytettynä arkkitehtuuriratkaisuna. Laajalle levinnyttä arkkitehtuuriratkaisua käytetään muun muassa tietovarasto-, sovellus-, sähköposti-, web-palvelimissa. Koskimies ja Mikkonen (2005) tuovat esille asiakas-palvelin arkkitehtuurin hyötyjä:

- *Selkeä työnjako:* Asiakas lähettää palvelimelle pyyntöjä, joihin palvelimen tulee vastata ennalta sovitusti. Näin ollen virheiden etsiminen jommastakummasta päästä on yksinkertaisempaa, koska tiedetään jo ennalta, millä tavoin kommunikointi tulisi tapahtua.
- *Hajautettavuus:* Koska asiakas ja palvelin ovat toisistaan riippumattomia voi yksi palvelin palvella useampaa asiakasta. Asiakkaiden palveleminen voidaan hajauttaa siten, etteivät asiakkaat ole edes tietoisia toisistaan.

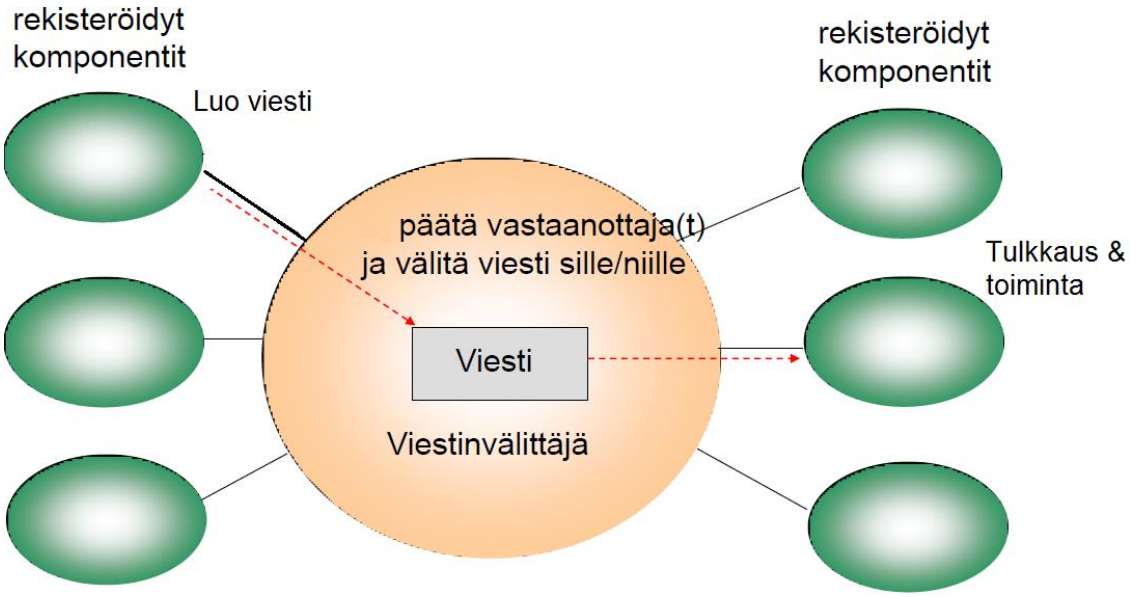
Koskimies ja Mikkonen (2005) sekä Buschmann ym. (1995) tuovat esille asiakas-palvelin arkkitehtuurin haittoja:

- *Tehottomuus*: Etämetodikutsu voi olla huomattavasti hitaampi kuin paikallinen kutsu. Asiakkaalla voi myös olla jo valmiina kaikki tarvittava tieto, mutta se lähettää sen ainoastaan palvelin-komponentille prosessoitavaksi. Joissakin tapauksissa tehokkaampaa olisi suorittaa tehtävä paikallisesti.

- *Rajapintojen muuttaminen*: Yhteisesti sovitun rajapinnan muuttaminen, esimerkiksi rajapinnan selkeyttämisen takia, aiheuttaa muutostarpeen sekä asiakas- että palvelinkomponentille. Tällöin vanhaa tapaa liikennöidä ei välttämättä voida enää tukea, mikä aiheuttaa päivitystarpeen kaikille asiakkaille, jotka ovat yhteydessä palvelimeen.

2.4.5 Viestinvälitysarkkitehtuurit

Viestinvälitysarkkitehtuurin toiminta perustuu viestinvälittäjäkomponentin toimintaan, joka on vastuussa tiedon välittämisestä eri komponenttien välillä (Buschmann ym., 1995). Erona asiakas-palvelin-arkkitehtuuriin on se, ettei viestinvälitysarkkitehtuurissa komponenttien rooleja ole määritelty (Koskimies & Mikkonen, 2005). Kuvio 3 esittää viestinvälitysarkkitehtuurin toiminnan. Komponentit rekisteröityvät viestinvälittäjälle ja kertovat mistä tiedosta ovat kiinnostuneet. Viestinvälittäjälle rekisteröitynyt komponentti luo viestin ja lähettää sen viestinvälittäjälle välitettäväksi. Viestinvälittäjä päättää komponentilta saamansa tiedon perusteella oikean vastaanottajan ja lähettää sen sille. Tämän jälkeen vastaanottaja tulkitsee viestin ja toimii määrittelyn mukaisesti.



Kuvio 3: Viestinvälitysarkkitehtuurin toiminta (mukaillen Koskimies & Mikkonen, 2005 s. 132)

Viestinvälitysarkkitehtuuri sopii tilanteisiin, jossa tulevien komponenttien määrää, laatua tai niiden välittämää tietoa ei ennalta tiedetä. Tällöin viestinvälitysarkkitehtuurin avulla rakennetaan rajapinta, jonka kautta kaikki komponentit kommunikoivat toisilleen. Koskimiehen ja Mikkosen (2005) mukaan viestinvälitysarkkitehtuurin toteutus voidaan rakentaa esimerkiksi komponenttien rekisteröimisellä. Komponentti voi ilmoittaa viestinvälittäjälle olevansa kiinnos-

tunut tietynlaisesta informaatiosta ja kun jokin muu komponentti välittää sopivaa tietoa, hoitaa viestinvälittäjä sen oikeille tahoille.

Kalja ym. (2005) esittävät työssään Viron valtion tietohallinnon parhaita käytänteitä. Työssä esitetään Viron X-road palveluväylä, joka on mahdollistanut maahan kustannustehokkaat julkiset tietojärjestelmäratkaisut. Palveluväylä yhdistää suuren määrän julkisia ja yksityisiä palveluja. Palveluväylän idea pohjautuu viestinvälitysarkkitehtuuriin, jossa palveluväylä toimii viestinvälittäjänä eri komponenttien välillä. Kansalaiset ja palveluntarjoajat voivat palveluväylään yhdistetyn tietojärjestelmän avulla hakea joustavasti tietoa monista eri lähteistä, kuten väestörekisteristä, kaupparekisteristä ja kiinteistörekisteristä.

Buschmann ym. (1995) listaavat viestinvälitysarkkitehtuurin eduiksi seuraavat:

- *Sijainnin avoimuus*: Tiedon tarvitsijan ei tarvitse tietää tiedon tarjoajan sijaintia, koska tiedonvälittäjä on vastuussa pyyntöjen ja vastausten välittämisestä. Näin ollen pyyntöön vastaaja voi välittää tiedon eteenpäin välitettäväksi tiedonvälittäjälle, välttämättä tietämättä, mistä pyyntö oli peräisin.
- *Komponenttien vaihdettavuus ja korvattavuus*: Komponenttien sisäistä rakennetta voidaan parantaa ja kehittää puuttumatta muiden komponenttien rakenteeseen, mikäli rajapinnat komponentin ulkopuolelle pysyvät samana. Myös viestinvälittäjän rakennetta voidaan muuttaa, mikäli myös uusi ratkaisu tarjoaa samat palvelut kuin aiemmin toteutettu.
- *Siirrettävyys*: Viestinvälitysarkkitehtuuri voi kätkeä komponenteilta käyttöympäristön, kuten käyttöjärjestelmän ja tietoliikenneyhteydet. Tämä mahdollistaa arkkitehtuurilla toteutetun järjestelmän viemisen muihin käyttöympäristöihin pienemmällä vaivalla kuin järjestelmän, jonka komponentit riippuvat käyttöjärjestelmän tarjoamista palveluista.
- *Viestinvälittäjien yhteensopivuus*: Viestinvälitysarkkitehtuurilla toteutetut järjestelmät voivat kommunikoida keskenään viestinvälittäjien välityksellä yhteisen rajapinnan välityksellä. Tällaista kahden viestinvälittäjän välistä rajapintaa kutsutaan sillaksi.
- *Uudelleenkäytettävyys*: Rakennettaessa uutta palvelua viestinvälitysarkkitehtuurilla toteutettuun järjestelmään voidaan sen toiminnassa käyttää hyväksi jo olemassa olevia komponentteja tai niiden palveluita. Esimerkiksi uusi raportointijärjestelmä voi hyödyntää olemassa olevien tietokantojen palveluita, sekä esimerkiksi aiemmin rakennettua tulostusjärjestelmää.

Buschmann ym. (1995) listaavat viestinvälitysarkkitehtuurin heikkoudeksi seuraavat:

- *Rajoitettu tehokkuus*: Viestinvälitysarkkitehtuuria käyttävät sovellukset ovat usein hitaampia kuin järjestelmät, jossa toiminnallisuus on staattisesti määritelty. Syynä on viestinvälittäjä, joka muodostaa välikomponentin perinteiseen asiakas-palvelin-arkkitehtuuriin verrattuna.

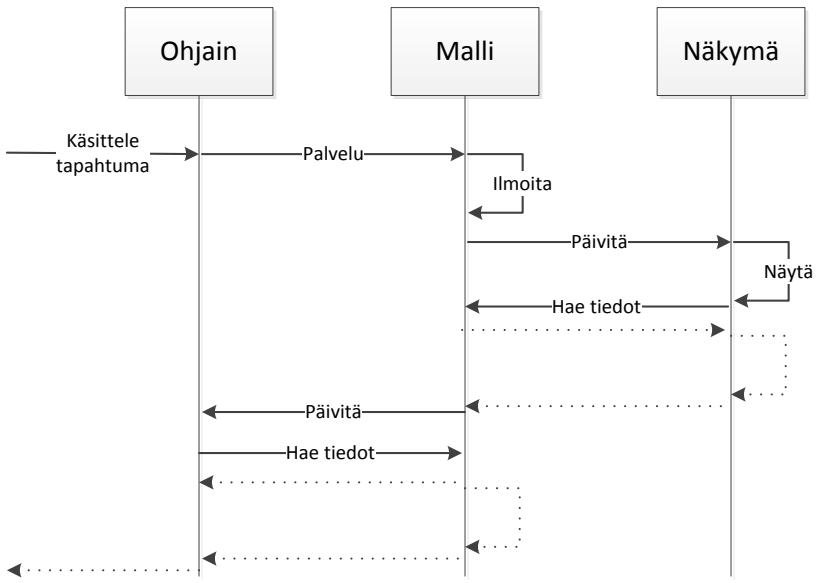
- *Vikasietoisuus*: Koska yksittäinen viestinvälittäjäkomponentti on vastuussa kaikesta tiedonsiirrosta komponenttien välillä, viestinvälittäjään aiheutuva vika pysäyttää kaiken liikenteen. Toki järjestelmään voidaan rakentaa useampi viestinvälittäjä, mikä taas voi heikentää arkkitehtuurin selkeyttä.

2.4.6 Malli-näkymä-ohjain-arkkitehtuurit

Malli-näkymä-ohjain-arkkitehtuuri (*Model-View-Controller, MVC*) koostuu kolmesta komponentista: mallit (*model*), näkymät (*view*) ja ohjaimet (*controller*) (Koskimies & Mikkonen, 2005). Mallit huolehtivat tiedon tallentamisesta, ylläpidosta ja käsittelystä, eli käytännössä ne ovat kuvaus tiedosta sisältäen toiminnallisuutta tiedon käsittelyyn. Näkymät määrittelevät käyttöliittymän ulkoasua ja sijoittelua. Ohjaimet toimivat rajapintana mallien ja näkymien välillä vastaanottaen käyttäjältä ja muualta järjestelmästä tulevia käskyjä. Hyvin usein ohjaimet mallien kanssa muodostavat ohjelman toiminnan kannalta olennaisen liiketoimintalogiikan.

Buschmannin ym. (1995) mukaan komponenttijaolla tietoa käsitellään kolmella eri tavalla: sitä prosessoidaan malli-komponentissa, syötetään ulos näkymien kautta sekä syötetään sisään ohjaimen kautta. Tällaisella komponenttijaolla voidaan ohjelman rakenne jakaa hallittaviin, kokonaisuuksiin, jolloin komponenttien välillä on myös selkeä työjako.

Kuvio 4 esittää komponenttien välisen työjaon ja niiden väliset tapahtumat esimerkkitapauksessa. Ohjain ottaa vastaan tapahtuman käyttäjältä tai järjestelmältä ja vie sen edelleen käsiteltäväksi ja näytettäväksi mallin kautta näkymään. Muutokset näkymässä voivat aiheuttavat tilan muutoksen, jotka heijastuvat ohjaimelle.



Kuvio 4: Malli-näkymä-ohjain arkkitehtuurin komponentit ja niiden väliset tapahtumat (mukaillen Buschmann ym., 1995 s. 130)

Buschmann ym. (1995) määrittelevät malli-näkymä-ohjain-arkkitehtuurin interaktiivisten järjestelmien kenties tunnetuimmaksi arkkitehtuurityyliksi. Arkkitehtuurityyli on kehitetty Smaltalkin yhteydessä jo 1970-luvulla, mutta sitä on alettu hyödyntämään web-järjestelmissä vasta hiljattain. Jazayeri (2007) korostaakin malli-näkymä-ohjain-arkkitehtuurin olevan yksi tämän päivän web-kehityksen tärkeimmistä suuntauksista.

Buschmann ym. (1995) esittävät seuraavia hyötyjä malli-näkymä-ohjain-arkkitehtuurille:

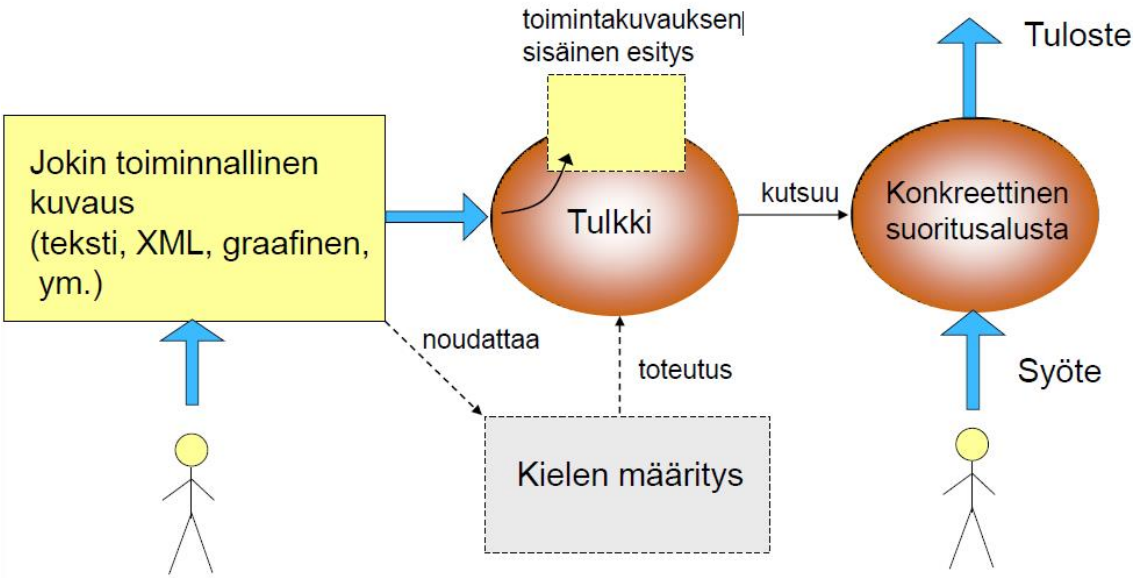
- *Usean näkymän mahdollisuus*: Tiedon esitystapa on irrotettu kokonaan varsinaisesta tiedosta. Tämä mahdollistaa useita eri esitystapoja samalle tiedolle. Useita näkymiä voidaan käsitellä rinnakkain ajonaikana.
- *Synkronoidut näkymät*: Arkkitehtuuri huolehtii siitä, että käyttäjän tai järjestelmän tekemät käskyt vaikuttavat kaikkiin tarvittaviin komponentteihin. Näin itsenäiset näkymät ja ohjaimet pysyvät synkronoituna.
- *Komponenttien välinen riippumattomuus*: Komponenttien välinen työjako mahdollistaa ohjaimien ja mallien vaihtamisen koskematta muihin komponentteihin. Näkymäkomponentin käyttäytymistä voidaan ohjailla myös ajonaikaisesti. Tämä ominaisuus mahdollistaa Koskimiehen ja Mikkosen (2005) mukaan mallin uudelleenkäytön. Kirjoittajien mukaan tämä tekee arkkitehtuurityylin käyttökelpoiseksi graafisten käyttöliittymien suunnittelussa.
- *Käyttäjäkokemuksen vaihdettavuus*: Koska näkymät on irrotettu muusta toteutuksesta, voidaan niitä kustomoida käyttäjäkohtaisesti. Tämä tekee myös järjestelmän kehityksestä suoraviivaisempaa: ohjelmistokehittäjät voivat keskittyä liiketoimintalogiikan toteuttamiseen ja graafikot käyttäjäkokemuksen kehittämiseen.

Buschmann ym. (1995) esittävät seuraavia malli-näkymä-ohjain-arkkitehtuurin haittapuolia:

- *Kasvanut kompleksisuus:* Yksinkertaisista valikoista ja tekstielementeistä koostuvaan yksinkertaiseen käyttöliittymään malli-näkymä-ohjain-arkkitehtuurin hyödyntäminen kasvattaa järjestelmän kompleksisuutta, ilman mainittavia hyötyjä.
- *Mahdollisuus liiallisiin päivityskutsuihin:* Kaikki näkymät eivät ole kiinnostuneita kaikista muutoksista mitä mallissa tapahtuu, jatkuvat päivityskutsut näkymältä mallille voivat olla turhia. Esimerkiksi piilotettua näkymää tarvitsee päivittää vasta, kun se on takaisin näkyvissä käyttäjälle.
- *Näkymän tehoton pääsy tietoon:* Malli-komponenteista riippuen näkymä saattaa joutua tekemään useita kyselyitä eri malleille saadakseen haluamansa tiedon. Muuttumatonta tietoa saatetaan joutua päivittämään useaan kertaan, ellei näkymä-komponentti osaa puskuroida tietoa.

2.4.7 Tulkkipohjaiset arkkitehtuurit

Tulkkipohjaisissa arkkitehtuureissa järjestelmälle annetaan syötteenä toiminnallisia kuvauksia, joita välittää eteenpäin suoritusalustan suoritettavaksi (Koskimies & Mikkonen, 2005). Koskimiehen ja Mikkosen (2005) mukaan järjestelmä voi olla sellainen, joka tarjoaa joukon peruspalveluita, mutta vasta ajonaikana selviää kuinka niitä yhdistellään lopputuloksen saamiseksi. Tällöin saavutetaan riippumattomuutta esimerkiksi eri alustoista, tai voidaan päättää vasta ajonaikana, millä tavoin järjestelmän tarjoamia peruspalveluita yhdistellään saadun syötteen perusteella. Alustariippumattomuus saavutetaan Koskimiehen ja Mikkosen (2005) mukaan sillä, että abstrakti suoritusalusta (tulkki) erotetaan konkreettisesta. Tulkkiarkkitehtuurin idea on pelkistetty kuvioon 5. Asiakas lähettää sovitun rakenteen tai kielen mukaisen käskylauseen tulkille, joka oman sisäisen toteutuksen mukaan kutsuu ja käyttää suoritusalustaa. Suoritusalusta voi tämän jälkeen palauttaa tuloksen suoraan asiakkaalle.



Kuvio 5: Tulkkiperusteisen arkkitehtuurin perusrakenne (mukaillen Koskimies & Mikkonen, 2005 s. 147)

Shawin ja Garlanin (1996) mukaan tulkkipohjaisia arkkitehtuureja käytetään laajalti virtuaalikoneiden rakentamisessa. Koskimies ja Mikkonen (2005) esittävätkin, että esimerkiksi virtuaalikoneisiin perustuvat ohjelmointijärjestelmät, kuten Java, ovat tulkkipohjaisen arkkitehtuurien sovellutuksia.

Koskimies ja Mikkonen (2005) nostavat esille tulkkipohjaisten arkkitehtuurien hyödyt:

- *Looginen suoritusympäristö on erillinen kokonaisuus*: Esimerkiksi SQL-kieltä tukeva tietokannanhallintajärjestelmä on mahdollista toteuttaa tulkkipohjai-

sella arkkitehtuurilla, mikä mahdollistaa sovelluksen siirtämisen tietokanta-järjestelmästä toiseen.
- *Tulkittava kieli ja sen merkitys ovat muutettavissa*: Esimerkiksi kielen laajentuminen ei tee vanhalla kielellä kirjoitettuja toimintakuvauksia toimimattomaksi, ellei toteutusta muuteta tai rakenteita vastaavia luokkia ja toteutuksia poisteta.

Vastaavasti Koskimies ja Mikkonen (2005) teoksesta poimittuja haittapuolia ovat:

- *Suoritustehokkuuden heikkeneminen:* Tulkkipohjaisen arkkitehtuurin suoritusaika voi olla huomattavasti natiiviohjelmaa suurempi, koska tulkittavan esityksen muodostaminen ja tulkitseva suoritus vievät enemmän resursseja kuin natiiviohjelma.
- *Tilankäyttö:* Koskimies ja Mikkonen (2005, s. 148) mukaan ”suoritettavan olioesityksen tilavaatimus voi olla suurehko: se saattaa viedä huomattavasti enemmän tilaa kuin vastaava merkkijonoesitys tai konekoodi”. Näin ollen yksinkertainenkin tehtävä voi vaatia paljon tilaa toimiakseen.

2.4.8 Yhteenveto arkkitehtuurityyleistä

Edellä on kuvattu kuusi tunnetuinta arkkitehtuurityyliä ja kerrottu niihin liittyvistä hyödyistä ja haittapuolista. Valinta esiteltyjen arkkitehtuurityylien välillä voidaan tehdä esimerkiksi niiden toiminnan, ominaisuuksien, ideologian, käyttötarkoituksen tai hyötyjen pohjalta. Etenkin laajassa järjestelmässä arkkitehtuurityyli täytyy pelkistää paikalliseksi suunnittelumalliksi, joka ratkaisee ainoastaan järjestelmän paikallisia ongelmia, koska koko järjestelmän ideologiaa ei yleensä voida kiteyttää yhteen arkkitehtuurityyliin.

Taulukko 1 esittää yhteenvedon arkkitehtuurityylien toimintaperiaatteista ja muutaman esimerkin käyttökohteista. Jokaisella arkkitehtuurityylillä on omaleimainen toimintaperiaate, joka tukee sen käyttöä tietyissä käyttökohteissa. Esimerkiksi tietoliikenneprotokollat (OSI, TCP/IP-viitemalli) perustuvat usein kerrosarkkitehtuuriin. Tietovuoarkkitehtuurit sopivat tietovirtojen jalostamiseen sekä prosessointiin ja esimerkiksi unix-järjestelmien putkioperaattori (*pipe operator*) käyttää tätä hyväkseen. Asiakas-palvelin arkkitehtuuria esiintyy etenkin web- ja sähköpostipalvelimissa. Viestinvälitysarkkitehtuuria voidaan käyttää yksinkertaistamaan laajoja ja monimutkaisia integraatioita vaativia järjestelmiä. Näkymä-malli-ohjain arkkitehtuurit eri variaatioineen sopivat käyttöliittymien rakentamiseen. Tulkkipohjaisia arkkitehtuurityylejä voidaan käyttää esimerkiksi virtuaalikoneissa ja tietokannanhallintajärjestelmissä. Ei ole kuitenkaan täysin poissuljettua, ettei jossakin tilanteessa esimerkiksi web-käyttöliittymää voisi rakentaa viestinvälitysarkkitehtuuria käyttäen. On tärkeää tunnistaa ja omaksua arkkitehtuurityylin toimintaperiaate, jotta voi ymmärtää millaisia ongelmia sillä voi ratkoa. Arkkitehtuurityylien käyttö helpottaa

ohjelmistojen suunnittelua, koska valmiina voi olla jo tietoa siitä, mikä arkkitehtuurillinen lähestymistapa toimii parhaiten kohdeongelmassa.

Taulukko 1: Yhteenveto arkkitehtuurityyleistä

Arkkitehtuurityyli	**Toimintaperiaate**	**Käyttökohteita**
Kerrosarkkitehtuurit	Organisoitu tasoihin abstraktiojärjestelmän mukaan	Tietoliikenneprotokollat
Tietovuoarkkitehtuurit	Itsenäiset komponentit välittävät tietoa väylien kautta	Tilattomien tietovirtojen jalostaminen ja prosessointi
Asiakas-palvelin- arkkitehtuurit	Resurssien tarjoaja (palvelin) tarjoaa palveluja asiakkaalle	Sähköposti ja Web -palvelimet
Viestinvälitysarkkitehtuurit	Viestinvälittäjäkomponentti välittää tietoa rekisteröityneiden komponenttien välillä	Laajoja integraatioita vaativat järjestelmät
Näkymä-malli-ohjain-arkkitehtuurit	Itsenäisten mallien, näkymien ja ohjaimien toiminta	Käyttöliittymien rakentaminen
Tulkkipohjaiset arkkitehtuurit	Abstraktin suoritusalustan (tulkki) erottaminen konkreettisesta	Virtuaalikoneet

2.5 Yhteenveto

Tässä luvussa esitettiin ohjelmistoarkkitehtuuria ja arkkitehtuurityylejä. Esitetyn perusteella lukija voi muodostaa kuvan ohjelmistoarkkitehtuurista ja eri arkkitehtuurityyleistä.

Ensimmäisessä alaluvussa esiteltiin määritelmiä ohjelmistoarkkitehtuurikäsitteelle ja esitettiin sille oma määritelmä. Tämän mukaan ohjelmistoarkkitehtuuri on järjestelmän perustuslaki, joka koostuu joukosta järjestelmää kuvaavia rakenneosia, niiden ominaisuuksia ja keskinäisiä suhteita. Luvussa kuvattiin myös ohjelmistoarkkitehtuurin historiaa ja varhaisimpia merkkejä Dijkstran (1968) tutkimuspaperia mukaillen.

Toisessa alaluvussa kerrottiin, mitä ohjelmistoarkkitehtuurilla tavoitellaan ja miten sitä käytetään ohjelmistoprojektin eri vaiheissa. Luvussa tuotiin esille ohjelmistoarkkitehtuurin hyötyjä, joita ovat: ohjelmistokehityksen vähentyvät kustannukset, ohjelmiston jatkokehityksen ja ylläpidettävyyden parantuminen sekä tuotteen kehitykseen kuluvan ajan vähentyminen. Lisäksi kerrottiin ohjelmistoarkkitehtuurin käyttökohteista, joita olivat: suunnittelun apuväline, kommunikoinnin mahdollistaja, työnjakaja ja laadun ennakoija.

Kolmannessa alaluvussa kerrottiin ohjelmistoarkkitehtuurien kuvaamisesta. Ohjelmistoarkkitehtuurin kuvaaminen tunnistettiin tärkeäksi asiaksi, koska muuten tehdyt suunnitelmat jäävät ajatustasolle yksittäisen ihmisten muistin varaan. Luvussa kerrottiin ohjelmistoarkkitehtuurin ilmentymistapoja, joita ovat luonnollinen kieli, malli, diagrammit, kuvat ja formaali kieli (Jansen, 2008). Luvussa kerrottiin lyhyesti ohjelmistoarkkitehtuurin kuvaustekniikoista, joista yksi tunnetuimmista on UML.

Neljännessä alaluvussa määriteltiin arkkitehtuurityyli järjestelmän kokonaisarkkitehtuuria kuvaavaksi periaatteeksi. Suunnittelumallin ja arkkitehtuurityylin eroa selvennettiin siten, että suunnittelumalli ratkaisee paikallisia ongelmia yhdenmukaisesti, kun taas arkkitehtuurityyli kuvaa koko järjestelmää. Käsitteiden jälkeen esiteltiin arkkitehtuurityylien kategoriointeja ja kuusi arkkitehtuurityyliä, jotka voivat olla myös erilaisten ongelmien standardiratkaisujen kuvauksia. Arkkitehtuurityyleistä kuvattiin: kerrosarkkitehtuurit, tietovuoarkkitehtuurit, asiakas-palvelin-arkkitehtuurit, viestinvälitysarkkitehtuurit, mallinäkymä-ohjain-arkkitehtuurit ja tulkkipohjaiset arkkitehtuurit. Lopuksi esitettiin yhteenveto arkkitehtuurityylien toimintaperiaatteesta ja esimerkkejä käyttökohteista.

3 OHJELMISTOARKKITEHTUURIN SUUNNITTELU JA ARVIOINTI

Tässä luvussa kerrotaan arkkitehtuurivetoisen ohjelmistokehityksen suunnittelusta ja arvioinnista. Ensin kuvataan ohjelmistoarkkitehtuurin suunnittelua ja kaksi erilaista suunnittelumenetelmää. Sen jälkeen kerrotaan ohjelmistoarkkitehtuurin arvioinnista ja kuvataan yksi esimerkkimenetelmä arvioinnin toteuttamiseksi. Lopuksi kirjallisuudesta poimittujen esimerkkien avulla kerrotaan, kuinka ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmiä voidaan käyttää ketterän ohjelmistokehityksen kontekstissa.

3.1 Ohjelmistoarkkitehtuurin suunnittelu

Ohjelmistoarkkitehtuuri voi syntyä huomaamatta ohjelmistokehityksen edetessä tai sitä voidaan suunnitella järjestelmällisesti ennen varsinaisen kehitystyön aloittamista. Seuraavassa esitetään ohjelmistoarkkitehtuurin suunnittelua menetelmien avulla.

3.1.1 Johdanto ohjelmistoarkkitehtuurin suunnitteluun

Albin (2003) määrittää suunnittelun (*design*) työvaiheeksi, jonka aikana etsitään ja löydetään ratkaisuja ongelmiin. Hänen mukaansa suunnittelumenetelmä on työkalu, joka helpottaa ratkaisujen etsimisessä. Albinin (2003) mukaan arkkitehtuurisuunnittelu on luonnollinen jatke ohjelmistokehityksen suunnitteluprosessille. Bassin ym. (2003) näkemyksen mukaan ohjelmistoarkkitehtuurin suunnittelumenetelmä sisältää tietoa siitä, kuinka ohjelmiston laadulliset vaatimukset voidaan kerätä ja kuinka vaatimukset voidaan huomioida arkkitehtuurissa erilaisten taktiikoiden ja mallien avulla.

Ohjelmistoarkkitehtuurin suunnittelun tarkoituksena on jakaa järjestelmä komponentteihin, jotka kommunikoivat keskenään. Ohjelmistoarkkitehtuuri syntyy liiketoimintavaatimusten sekä teknisten ja laadullisten vaatimusten

myötä (Bass ym., 2003). Ohjelmistokomponenttien ja niiden välisen kommunikaation tehtävä on toteuttaa nämä vaatimukset (Albin, 2003).

Ohjelmistoarkkitehtuurin suunnittelumenetelmillä on samat hyödyt kuin ohjelmistoarkkitehtuurin suunnittelumallilla: ne ovat käytännössä hyväksi havaittujen ratkaisujen kuvauksia tietyssä tilanteessa (Koskimies & Mikkonen, 2005). Niiden avulla ohjelmistoarkkitehti saa selkeät työvaiheet, joita noudattamalla ohjelmistoarkkitehtuuri suurella todennäköisyydellä täyttää sille asetetut vaatimukset.

Hofmeister ym. (2007) ovat tutkineet viitta teollisuudesta lähtöisin olevaa arkkitehtuurin suunnittelumenetelmää. Tutkimukseen valitut suunnittelumenetelmät olivat:

- Ominaisuusvetoinen arkkitehtuurisuunnittelu (*Attribute-Driven-Design*) (Bachmann & Bass, 2001)
- Siemens 4:n näkymän suunnittelumenetelmä (*Siemens' 4 views*) (Soni ym., 1995)
- RUP:n 4+1:n näkymän suunnittelumenetelmä (*RUP's 4+1 Views)* (Kruchten, 1995; Kruchten, 2004)
- BAPO-menetelmä (*Business Architecture Process and Organization*) (America ym., 2000)
- ASC-menetelmä (*Architectural Separation of Concerns*) (Ran, 2000)

Kirjoittajat ovat analysoineet suunnittelumenetelmien yhtäläisyydet ja tämän perusteella luoneet yleisen mallin, joka kuvaa ohjelmistoarkkitehtuurin suunnitteluprosessia. Tutkimuksessa kiinnitettiin huomiota menetelmien toimintoihin (*activities*) ja tuotoksiin (*artifacts*). Käytännössä suunnittelumenetelmän toiminnot ovat työvaiheita, joiden avulla tuotos, eli arkkitehtuuri saadaan aikaan.

Hofmeister ym. (2007) ovat tunnistaneet viidestä menetelmästä paljon yhtäläisyyksiä, vaikka menetelmät ovat kehitetty toisistaan riippumatta. Kirjoittajat kertovat menetelmien kolmesta yhtäläisyydestä, jotka ovat: menetelmien perustuminen laadullisiin vaatimuksiin, usean näkymän huomiointi ja tuloksen iteratiivinen arviointi eri pisteissä. Menetelmien toiminnan perustuminen laadullisiin vaatimuksiin selittyy sillä, että suuren järjestelmän toimintaa ei voida varmistaa ainoastaan suunnittelemalla yksittäisten ohjelmistokomponenttien toimintaa ottamatta huomioon kokonaisuutta. Laadulliset vaatimukset varmistavat, että ohjelmistokomponentit yhdessä muodostavat käyttökelpoisen järjestelmän. Usean näkymän huomioimisella arkkitehtuurilla tarjotaan eri osapuolia hyödyttävää tietoa. Ohjelmistosuunnittelija tarvitsee arkkitehtuurista erilaista tietoa kuin johto. Usean näkymän tarjoaminen voi liittyä myös eri abstraktiotasojen kuvaamiseen, missä järjestelmä ensin kuvataan karkealla tasolla ja sen jälkeen paneudutaan yksityiskohtiin. Menetelmään integroidun arvioinnin avulla varmistutaan arkkitehtuurin ajantasaisuudesta ohjelmistokehityksen eri vaiheissa. (Hofmeister ym. 2007.)

Vastaavasti Hofmeister ym. (2007) ovat löytäneet viidestä esitellystä suunnittelumenetelmästä erovaisuuksia. Menetelmien painotukset arkkitehtuu-

rin tekemiseksi ovat erilaisia. Esimerkiksi RUPin (Kruchten, 2004; Kruchten, 1995) keskeinen ajatus on inkrementaalinen kehitys, kun taas ominaisuusvetoinen arkkitehtuurisuunnittelu (Bachmann & Bass, 2001) nojaa arkkitehtuurityylien käyttöön. Suunnittelumenetelmät käyttävät myös erilaisia lähestymistapoja arkkitehtuurin saavuttamiseen. Ominaisuusvetoinen arkkitehtuurisuunnittelu (Bachmann & Bass, 2001) nojaa skenaarioihin, RUP (Kruchten, 2004; Kruchten, 1995) riskien poistamiseen ja ASC-menetelmä (Ran, 2000) arkkitehtuurillisesti merkittävien vaatimusten tunnistamiseen. Suunnittelumenetelmien soveltamisala voi vaihdella. Esimerkiksi ominaisuusvetoinen arkkitehtuurisuunnittelu (Bachmann & Bass, 2001) tarjoaa työvaiheet arkkitehtuurillisten vaatimusten valitsemiseksi, mutta ei tarkemmin selitä kuinka vaatimuksia kerätään. Siemens 4 näkymän suunnittelumenetelmä (Soni ym., 1995) tarjoaa taas työkaluja muidenkin vaatimusten keräämiseksi. Muita eroavaisuuksia menetelmissä ovat alkuperäinen tarkoitus, painotukset ja ulottuvuus, eli millaisissa eri yhteyksissä sitä voidaan käyttää.

Seuraavassa esitetään ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä ja RUP 4+1 näkymän suunnittelumenetelmä. Menetelmät ovat valittu niiden tunnettavuuden mukaan.

3.1.2 Ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä

Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän (*Attribute-Driven Design, ADD*) tarkoituksena on tarjota arkkitehtuurin suunnitteluun menetelmä, jonka avulla voidaan täyttää järjestelmältä vaaditut laadulliset ja toiminnalliset vaatimukset sekä suunnittelulle asetetut rajoitteet (Bachmann & Bass, (2001). Bachmannin ja Bassin (2001) mukaan menetelmä voidaan nähdä laajennuksena esimerkiksi RUP-ohjelmistokehitysprosessiin (*Unified Software Development Process*) (Kruchten, 2004). Ominaisuusvetoisen arkkitehtuurisuunnittelun avulla voidaan järjestelmälle luoda korkean tason arkkitehtuuri, jonka jälkeen kehitystyötä voidaan yksityiskohtaisemmin jatkaa RUP-ohjelmistokehitysprosessia mukaillen. Menetelmä on kehitetty SEI-instituutissa.

Alkuperäinen ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä (Bass ym., 2003; Bachmann & Bass, 2001) sisältää kahdeksan vaihetta, jotka johtavat koko järjestelmän kattavaan arkkitehtuurisuunnitelmaan. Työvaiheet suoritetaan iteratiivisesti siten, että jokainen ohjelmiston osa tulee käsitellyksi valitulla tarkkuudella. Sittemmin Wojcik ym. (2006) ovat esitelleet menetelmästä uuden version, joissa työvaiheita on tarkennettu ja menetelmän opettelua ja käyttöä on helpotettu. Tässä työssä käytetään uudempaa versiota, koska se on kehitetty alkuperäisestä menetelmästä saadun palautteen perusteella (Wojcik ym., 2006). Kuvio 6 esittää yhteenvedon menetelmästä.

Wojcik ym. (2006) esittävät ominaisuusvetoista arkkitehtuurisuunnittelumenetelmää havainnollistavassa dokumentissa tarkastuslistan, mitä tehtäviä tulee suorittaa kussakin työvaiheessa. Myös lyhyt sanasto esitetään lukijalle. Sekä sanasto, että tarkastuslistat helpottavat menetelmän opettelua ja käyttöä.

Kuvio 6: Ominaisuusvetoinen arkkitehtuurisuunnittelun lähtötiedot, työvaiheet ja lopputulos (mukaillen Wojcik ym., 2006 s. 17)

Ominaisuusvetoinen arkkitehtuurisuunnittelun vaiheet kuvataan alla Wojcikin ym. (2006) mukaan:

1. *Varmistu vaatimuksien riittävyydestä*: Menetelmän edellytys on näkemys järjestelmältä vaaditusta vaatimuksista, jotka ovat riittävän laadukkaita ja priorisoituja. Menetelmää käytettäessä vaatimukset jaotellaan kolmeen kategoriaan: toiminnallisiin vaatimuksiin, suunnitteluraajoituksiin ja laatuvaatimuksiin.

2. *Valitse kuvattava elementti*: Tässä vaiheessa valitaan, mitä järjestelmän osa-aluetta kuvataan. Ensimmäisenä elementtinä kuvataan järjestelmä kokonaisuutena, jonka jälkeen voidaan siirtyä pienempiin osiin. Työkulku voi edetä

esimerkiksi seuraavasti: järjestelmä -> alijärjestelmä -> moduuli -> alimoduuli. Haluttava esitystaso on aina riippuvainen arkkitehtuuritasosta, joka halutaan saavuttaa. Mikäli järjestelmä halutaan kuvata vain ylemmällä tasolla, riittää menetelmän vaiheiden läpikäynti yhden kerran.

Wojcikin ym. (2006) mukaan kuvattavien järjestelmien järjestyksen valinta voi perustua esimerkiksi arvioon osajärjestelmän arkkitehtuurin toteutuksen vaikeudesta ja riskeistä, liiketoimintavaatimuksista tai organisaation määrittelemistä kriteereistä. Näin ollen valinta sen suhteen, kumpi kahdesta osajärjestelmästä kuvataan ensin, voidaan tehdä sen perusteella, kummalla osajärjestelmistä on määritelty paremmat vaatimukset ja on näin ollen helpompi toteuttaa. Toisaalta liiketoimintavaatimukset ja organisaatiokriteerit voivat vaatia juuri päinvastaisen osajärjestelmän mallintamista.

3. *Tunnista arkkitehtuuriin vaikuttavat vaatimukset***:** Wojcikin ym. (2006) mukaan tässä vaiheessa vaatimukset priorisoidaan sen mukaan, kuinka suuri vaikutus niillä on järjestelmän arkkitehtuuriin. Luokittelu voidaan tehdä esimerkiksi kategorioinnilla ”korkea vaikutus”, ”keskitasoinen vaikutus” tai ”vähäinen vaikutus”.

   Vaatimukset, jotka sidosryhmät ovat arvioineet tärkeiksi ja jotka arkkitehti on tunnistanut arkkitehtuuria ohjaavaksi, määritellään mahdollisiksi arkkitehtuuria ohjaaviksi vaatimuksiksi (*candidate architectural drivers*).

4. *Valitse suunnitteluratkaisut, jotka täyttävät arkkitehtuurilliset vaatimukset:* Tässä vaiheessa valitaan arkkitehtuurisuunnitelmassa ilmentyvät arkkitehtuurielementit ja niiden suhteet. Tämän saavuttamiseksi voidaan vertailla eri arkkitehtuurityylejä ja niiden sopivuutta arkkitehtuuria ohjaaviin vaatimuksiin. Lisäksi käytetään löydettyjä rajoitteita ja laadullisia ominaisuuksia, jotta oikeat elementit löydetään.

5. *Toteuta arkkitehtuurielementit ja jaa vastuut*: Tässä pureudutaan syvemmälle arkkitehtuurielementteihin ja niiden sisäiseen toiminnallisuuteen. Tämän perusteella määritellään elementtien välisiä suhteita ja roolijakoa ottaen huomioon aiemmin määritellyt arkkitehtuurilliset vaatimukset. Tässä vaiheessa selvitetään, millä tavoin arkkitehtuurielementit ovat riippuvaisia toisistaan.

6. *Määritä rajapinnat arkkitehtuurielementeille*: Aiemmalla tasolla on selvitetty mitkä ovat riippuvuudet eri arkkitehtuurielementtien välillä. Tässä vaiheessa määritellään, mitä arkkitehtuurielementti tarvitsee muualta ja mitä se tarjoaa muille. Riippuen kuvattavasta tasosta lopputuloksena voi olla erilaisia rajapintasuunnitelmia, kuten informaatiota siitä, mitä tietoa arkkitehtuurielementtien välillä liikkuu, tai tietoa siitä, kuinka virheitä käsitellään arkkitehtuurielementtien välillä.

7. *Varmista ja jalosta vaatimukset*: Ominaisuusvetoisen arkkitehtuurisuunnittelun lähes jokaisessa vaiheessa päätöksiä tehdään järjestelmältä vaadittuihin vaatimuksiin peilaten. Tässä vaiheessa varmistutaan vielä siitä, että toiminnalliset ja laadulliset vaatimukset saavutetaan kuvatulla arkkitehtuurilla. Koska vaatimukset määritellään hyvin usein koko järjestelmälle, tässä vaiheessa varmistutaan siitä, että vaatimukset otetaan huomioon tarvittavilta osin osajärjestelmien suunnittelun yhteydessä.

8. *Toista vaiheet 2-7 seuraavalle elementille:* Wojcikin ym. (2006) mukaan seuraavaksi siirrytään seuraavan elementin käsittelyyn. Se voi olla nykyisellä kierroksella luodun elementin lapsi, tai aiemmin luotu elementti. Päätös seuraavasta kuvattavasta elementistä tehdään kohdassa 2.

Työvaiheiden lopputuloksena on arkkitehtuurisuunnitelma, jossa elementeille on määritelty roolit, vastuut, ominaisuudet sekä yhteydet muihin elementteihin.

3.1.3 RUP 4+1 mallin suunnittelumenetelmä

RUP-ohjelmistokehitysprosessin (*IBM Rational Unified Process*) tarkoituksena on hallita ohjelmistokehitysprosessin tehtäviä ja vastuita. Sen päämääränä on tuottaa hyvälaatuinen ja käyttäjiä tyydyttävä ohjelmisto, joka on toteutettu pysyen aikataulussa ja budjetissa. (Kruchten, 2004.)

RUP suosittaa arkkitehtuurin suunnittelumenetelmän pohjaksi Kruchtenin (1995) 4+1 -mallia (Hofmeister ym., 2007; Kruchten, 2004). Kruchtenin 4+1 -malli ei esitä selkeitä työvaiheita ohjelmistoarkkitehtuurin tuottamiseksi, vaan siinä kuvataan, millaisia eri näkymiä arkkitehtuuri voi tarjota. Kruchtenin (1995) malli koostuu fyysisestä näkymästä, loogisesta näkymästä, prosessinäkymästä, kehitysnäkymästä ja skenaarionäkymästä.

4+1- mallin *fyysinen näkymä* käsittää järjestelmän laiteläheiset komponentit, niiden yhteydet ja muut asiat, joista laitteistoa ylläpitävät tahot ovat kiinnostuneita. *Looginen näkymä* luokittelee järjestelmän toiminnallisuuden eri rakenneosien hoidettavaksi. *Prosessinäkymässä* toiminta jaetaan prosesseihin, jossa kuvataan tapahtumaketjut, jotta järjestelmän suorituskykyä ja skaalautuvuutta voidaan arvioida. *Kehitysnäkymä* on tarkoitettu ohjelmistokehittäjille, ja se kuvaa järjestelmän jaon esimerkiksi moduuleihin tai luokkiin. *Skenaarionäkymä* nitoo muut näkymät yhteen. (Hofmeister ym., 2007.)

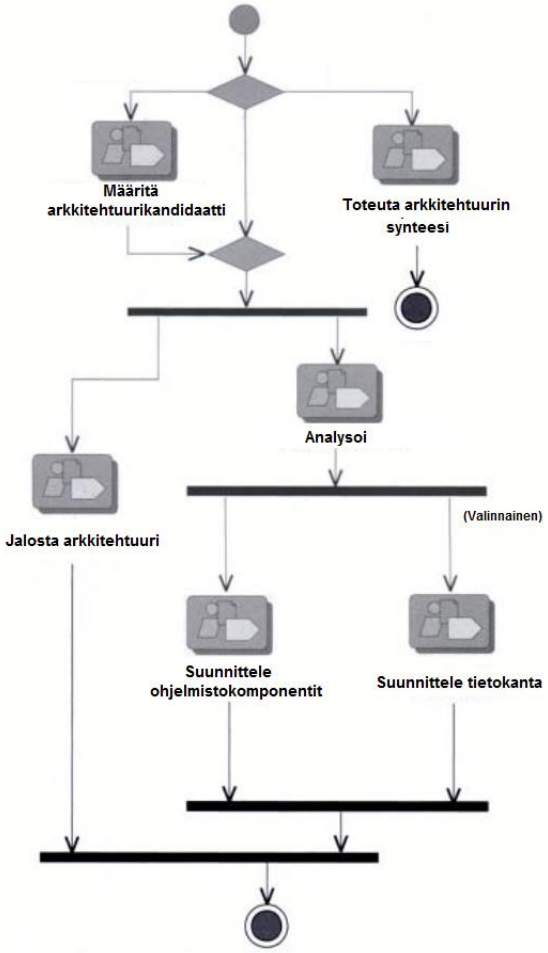
Kruchten (2004) esittää RUP-menetelmän mukaisen ohjelmistoarkkitehtuurin suunnittelumenetelmän. Käytännössä suunnittelu on kuitenkin vahvasti hajautettu RUP-menetelmän eri vaiheisiin. *RUP 4+1 -mallin suunnittelumenetelmä* koostuu kolmesta päävaiheesta. Vaiheet on esitelty alla Kruchtenin (2004) ja Hofmeister ym. (2007) mukaan:

1. *Määritä arkkitehtuurikandidaatti*: Käyttötapausanalyysiä käyttämällä selvitetään arkkitehtuurillisesti tärkeät vaatimukset. Vaatimusten määrittämisen jälkeen voidaan luoda arkkitehtuurikandidaatti seuraavia vaiheita varten.

Arkkitehtuurikandidaatti voi sisältää alustavan hahmotelman arkkitehtuurin elementeistä ja toteutettavista abstraktiotasoista. Tämän vaiheen aktiiviset toimijat ovat ohjelmistoarkkitehti, joka suorittaa arkkitehtuurianalyysiä, sekä ohjelmistosuunnittelija, joka tekee käyttötapausanalyysiä.

2. *Suorita arkkitehtuurin synteesi*: Vaiheessa todennetaan arkkitehtuurin toteuttamiskelpoisuus saatujen vaatimusten pohjalta. Edellisessä vaiheessa huomioitujen käyttötapauksien lisäksi tässä kohdassa huomioidaan muut toiminnalliset ja ei-toiminnalliset vaatimukset. Eri toiminnallisia ja laadullisia vaatimuksia yhdistelemällä muodostetaan malli tai prototyyppi, joka kuvaa arkkitehtuuria. Lopputuloksena on tarkempi visio tulevasta arkkitehtuurista.
3. *Jalosta arkkitehtuuria*: Arkkitehtuurista tunnistetaan tarvittavat elementit (luokat, prosessit), jotka yhdistetään kokonaisuudeksi. Tässä vaiheessa tunnistetaan mahdolliset suunnittelumallit ja taktiikat, jotka ratkaisevat havaittuja ongelmia. Lopuksi tuotettu arkkitehtuuri katselmoidaan.

Kuvio 7 esittää työvaiheiden kulun. Kuviosta voidaan huomata, että arkkitehtuurin synteesin toteuttamisen jälkeen voidaan edetä suoraan ohjelmiston kehittämisvaiheeseen. Vaihtoehtoisesti ensin määritellään arkkitehtuurikandidaatti, jonka analyysin pohjalta toteutetaan tarkempi ohjelmistoarkkitehtuuri, sekä suunnitellaan ohjelmistokomponentit ja tietokantarakenteet.



Kuvio 7: RUP-pohjaisen ohjelmistoarkkitehtuurisuunnitteluprosessin työvaiheet (mukaillen Kruchten, 2004 s. 181)

3.2 Ohjelmistoarkkitehtuurin arviointi

Ohjelmistoarkkitehtuurin suunnittelun tuloksena saatu arkkitehtuuri tulee arvioida. Arvioinnin tarkoituksena on varmistaa ohjelmistoarkkitehtuurin soveltuvuus käyttökohteeseen. Seuraavassa esitellään ohjelmistoarkkitehtuurin arviointia ensin yleisesti ja sen jälkeen kuvataan käytettäväksi yksi arviointimenetelmä.

3.2.1 Lyhyesti arvioinnista ja arviointimenetelmistä

Koskimiehen ja Mikkosen (2005) mukaan ohjelmistoarkkitehtuurin arviointi perustuu arkkitehtuurikomponenttien ja alijärjestelmien suhteiden ja niiden ominaisuuksien arviointiin. Koskimies ja Mikkonen (2005, s. 221) korostavat arkkitehtuurin arvioinnin haastavuutta: ”arkkitehtuurin arviointi eroaa monien muiden teknisten vaihetuotteiden katselmoinnista sikäli, että sen laatu ei perustu puhtaasti suunnitteluaikaisiin teknisiin ansioihin, vaan myös sen kykyyn täyttää sille määritellyt pidemmän tähtäyksen tavoitteet”. Käytännössä arkkitehtuurin onnistumista voidaan siis arvioida täysin luotettavasti vasta ohjelmiston elinkaaren loppupäässä. Oleellista arkkitehtuurin arvioinnissa on tunnistaa ohjelmiston tulevia asiakas- ja laajentumistarpeita, joihin arkkitehtuurin tulee pystyä vastaamaan.

Babarin ym. (2004) mukaan ohjelmistoarkkitehtuurin arviointimenetelmät tarjoavat suuntaviivoja, heuristiikoita ja standardeja, joita noudattamalla saavutetaan menetelmän tarkoitus. Kirjoittajien mukaan arviointimenetelmän tulee määrittää, mitä se vaatii toimiakseen, mitkä ovat askeleet tuloksen saavuttamiseksi ja mikä menetelmän tulos oikeastaan on. Babar ym. (2004) nostaa esille arviointimenetelmien puutteellisen käyttöohjeistuksen, vaikka suurin osa arviointimenetelmistä tarjoaa kuitenkin karkean tason selityksen menetelmän toiminnasta.

Arviointimenetelmät vähentävät ohjelmistokehityksen riskejä ja ne ovat suhteellisen edullisia toteuttaa. Parhaimmillaan ne vähentävät toteutus- tai käyttöönottovaiheessa toteutuvia virheitä, joiden korjaaminen on erittäin kallista. Koska ohjelmistoarkkitehtuuri kuvaa ohjelmiston luonnetta ennen toteutuksen varsinaisen aloittamista, voidaan virheet ja rajoitteet tunnistaa jo suunnitteluvaiheessa. (Clements ym., 2003.)

Koskimies ja Mikkonen (2005) sekä Clements ym. (2003) korostavat arkkitehtuurin arvioinnissa ohjelmiston laadullisten ominaisuuksien arviointia. Reekien ym. (2006) mukaan laadulliset ominaisuudet voidaan jakaa kahteen luokkaan, ajonaikaisiin (*runtime*) ja ei-ajonaikaisiin (*non-runtime*). *Ajonaikaisiin laadullisiin ominaisuuksiin* kuuluvat suorituskyky, käytettävyys, luotettavuus ja turvallisuus. *Ei-ajonaikaiset ominaisuudet* koskettavat järjestelmää laajemmin. Niihin kuuluvat: ylläpidettävyys, testattavuus, uudelleenkäytettävyys, konfiguroitavuus ja laajennettavuus. Alla on kullekin näistä ominaisuuksista annettu lyhyt kuvaus:

- *Suorituskyky*: Suorituskykyä voidaan mitata esimerkiksi sillä, kuinka paljon resursseja (muisti, CPU-aika, tilavaatimus) jokin ohjelmisto tai sen osa vaatii.
- *Käytettävyys*: Kuvaa sitä, kuinka helppokäyttöinen ohjelmisto, järjestelmä tai verkkosivusto on käyttäjän kannalta.
- *Luotettavuus*: Luotettavuutta voidaan arvioita sen perusteella, kuinka usein ohjelmisto epäonnistuu suorittamaan vaaditun tehtävän.
- *Turvallisuus*: Ohjelmiston ominaisuus, joka varmistaa, että järjestelmää voidaan käyttää sillä tavoin kuin sen on tarkoitus toimia. Turvallisuuteen liittyvät oleellisesti käyttäjäautentikointi ja tiedon salaaminen.
- *Ylläpidettävyys*: Kuvaa, kuinka helposti järjestelmää voidaan laajentaa. Käytännössä tähän vaikuttavat esimerkiksi ohjelmistokoodin rakenne, rajapintamääritykset ja dokumentaatio.
- *Testattavuus:* Esittää, millä tavoin ohjelmistoa voidaan testata, eli miten sen laadunvarmistus on mahdollista suorittaa. Automaattista testattavuutta voidaan jakaa edelleen yksikkötesteihin, integraatiotesteihin ja järjestelmätesteihin.
- *Uudelleenkäytettävyys:* Kuvaa, kuinka hyvin ohjelmistoratkaisut tukevat ohjelmistokoodin uudelleenkäyttöä. Esimerkiksi samaa tietokantayhteyttä voidaan käyttää niin käyttöliittymässä kuin palvelinlogiikan toteuttamisessa.
- *Konfiguroitavuus:* Ilmentää, kuinka hyvin järjestelmää voidaan konfiguroida eri tapauksiin. Konfiguroitavuutta toteuttavat yleensä ohjelmiston konfiguraatiotiedostot, jotka ovat määriteltävissä erilaisia tapauksia varten.
- *Skaalautuvuus:* Tarkoittaa, kuinka helposti järjestelmän suorituskykyä voidaan kasvattaa häiritsemättä järjestelmän toimintaa. Skaalautuvuuden avulla voidaan samalla järjestelmällä esimerkiksi palvella useampia asiakkaita.

Kazman ym. (1998) kiinnittää huomiota arkkitehtuurin laadullisten ominaisuuksien vertailuun, koska laajassa ohjelmassa esimerkiksi tehokkuus, saatavuus ja muokattavuus riippuvat enemmän ohjelmiston arkkitehtuuriratkaisuista kuin yksittäisen ohjelmistokomponentin toiminnasta.

Ohjelmistoarkkitehtuurin arviointia varten on esitetty erilaisia arviointimenetelmiä. Arviointimenetelmät kuvaavat työvaiheet, joiden avulla ohjelmistoarkkitehtuuri voidaan arvioida luotettavasti. Ennalta määriteltyjen työvaiheiden ansiosta arviointi on toistettavissa eri asiayhteyksissä samankaltaisesti (Clements ym., 2003). Koskimiehen ja Mikkosen (2005) esittävät kysymyksiä, joihin ohjelmistokehityksen arviointimenetelmät vastaavat:

- Sopiiko suunniteltu arkkitehtuuri järjestelmälle?
- Mitkä vaihtoehtoisista arkkitehtuureista soveltuvat parhaiten järjestelmälle, ja miksi?
- Miten hyvä tulee olemaan järjestelmän jokin tietty laadullinen ominaisuus, olettaen että järjestelmä toteutetaan järkevästi?

Babari ym. (2004) ovat tutkineet ohjelmistoarkkitehtuurin arviointimenetelmiä ja rakentaneet kehyksen, jota voidaan käyttää sopivan arviointimenetelmän va-

litsemiseksi. He ovat ottaneet vertailuun kahdeksan vakiintunutta arviointimenetelmää, jotka pohjautuvat skenaarioihin. Menetelmät ovat:

- SAAM (*Scenario based Architecture Analysis Method*) (Kazman ym., 1994)
- ATAM (*Architecture Tradeoff Analysis Method*) (Kazman ym., 1998)
- ARID *(Active Reviews for Intermediate Design*) (Clements, 2000)
- SAAMER (*SAAM for Evolution and Reusability*) (Lung ym., 1997)
- ALMA (*Architecture-Level Modifiability Analysis*) (Bengtsson ym., 2004)
- ALPSM (*Architecture-Level Prediction of Software Maintenance*) (Bengtsson & Bosch, 1999)
- SBAR (*Scenario-Based Architecture Reengineering*) (Bengtsson & Bosch, 1998)
- SAAMCS (*SAAM for Complex Scenarios*) (Lassing ym., 1999)
- ISAAMCR (*Integrating SAAM in domain-Centric and Reuse-based development*) (Molter, 1999)

Babarin ym. (2004) tutkimuksessa on kiinnitetty huomiota arviointimenetelmien kypsyystasoon, prosesseihin, työvaiheisiin, tavoitteisiin, laadullisten ominaisuuksien käyttöön, suoritusajankohtaan, arvioinnin painopisteisiin, sidosryhmien osallistumiseen, työkalutukeen ja resurssivaatimuksiin. ATAM ja ALMA –menetelmät on todettu kaikkein jalostetuimmiksi, koska niitä on käytetty ja arvioitu useissa eri käyttökohteissa ja –ympäristöissä.

Babarin ym. (2004) mukaan suurin osa arviointimenetelmistä tarjoaa hyvän ohjeistuksen sille, kuinka menetelmiä tulee käyttää ohjelmistoarkkitehtuurin arvioinnissa. Useimmat arviointimenetelmistä tarjoavat tekniikoita laadullisten ominaisuuksien sekä skenaarioiden luomiseen ja arviointiin. Lähes kaikki menetelmät tunnistavat eri sidosryhmien tärkeyden arviointiprosessissa. Eroavaisuuksia on ainoastaan painotuksissa sen suhteen, millaisella kokoonpanolla arkkitehtuuria arvioidaan.

Babar ym. (2004) toteavat ATAM-menetelmän (Kazman ym., 1998) olevan ainoa, joka tarjoaa kattavan prosessituen menetelmän käyttöön. ATAM-menetelmä ottaa huomioon myös ei-teknisiä asioita, kuten sosiaaliset näkökulmat. Kirjoittajien mukaan arviointimenetelmien tarjoama työkalutuki on olematonta. Lisäksi menetelmät tarjoavat heikosti tietoa vaadituista henkilöresursseista muutamaa poikkeusta (SAAM, ATAM, ARID) lukuun ottamatta.

Seuraavassa esitetään esimerkkinä ATAM-arviointimenetelmä (*Architecture Tradeoff Analysis Method*) (Kazman ym., 1998). Menetelmä on valittu, koska se on Babarin ym. (2004) mukaan ARID-menetelmän (Clements, 2000) ohella ainoa, joka tarjoaa riittävän ohjeistuksen arviointiprosessin eri vaiheille. Verratuista menetelmistä se on myös kattavimmin testattu eri käyttöympäristöissä.

3.2.2. ATAM-arviointimenetelmä

ATAM-arviointimenetelmä (Kazman ym., 1998) on SEI-instituutissa (*Software Engineering Institute*) kehitetty ohjelmistoarkkitehtuurin arviointimenetelmä. Se pohjautuu SAAM-menetelmään *(Software Architecture Analysis Method*), joka on

julkaistu vuonna 1994 (Kazman ym., 1994). Menetelmän avulla arvioidaan ohjelmistoarkkitehtuurissa tehtyjä ratkaisuja laadullisten ominaisuuksien perusteella, jotta saadaan tietoa siitä, täyttääkö arkkitehtuuri sille asetetut tavoitteet konkretisoituessaan.

Menetelmän käyttö perustuu esimerkkitilanteisiin (skenaarioihin). Eri sidosryhmät, kuten kehittäjät, projektipäälliköt, asiakkaat ja ohjelmistoarkkitehdit luovat omasta näkökulmastaan todennäköisiä käyttötapauksia, muutostilanteita, ylläpidollisia tehtäviä ja rasitusskenaarioita. Näiden esimerkkitilanteiden pohjalta arvioidaan, kuinka hyvin järjestelmä pystyy arkkitehtuurillisesti vastaamaan niihin. (Koskimies & Mikkonen, 2005.)

ATAM-menetelmä koostuu neljästä vaiheesta. Alun perin Kazman ym. (1998) ovat nimenneet vaiheet seuraavasti: skenaarioiden ja vaatimusten kerääminen, arkkitehtuurinäkymien ja skenaarioiden toteuttaminen, mallin rakentaminen ja analysointi sekä päätöksien teko. Clements ym. (2003) ovat nimenneet vaiheet eri tavalla: valmistelu, alustava arviointi, lopetus ja seuranta. Käytämme tässä Koskimiehen ja Mikkosen (2005) esittämää luokittelua, joka on koostettu edellä mainituista teoksista. Koskimiehen ja Mikkosen (2005) nimeämät vaiheet eivät ole suoria käännöksiä, vaan nimet ovat annettu vaiheen sisällön perusteella. Alla on kuvattu vaiheet ja niiden aktiviteetit Koskimiehen ja Mikkosen (2005) mukaan:

1. *Esittelyosio*: Vaiheessa kuvataan arviointimenetelmän tarkoitus arviointiin osallistuville. Tämän jälkeen kerrataan järjestelmän olennaiset vaatimukset liiketoiminnan kannalta sekä mitä rajoitteita (tekniset, taloudelliset, poliittiset) on tähän mennessä havaittu. Toimintaympäristön kuvaamisen jälkeen pääarkkitehti kuvaa arkkitehtuurin ja tekniset ratkaisut sillä tasolla, että kaikki osallistujat ymmärtävät tehdyt ratkaisut.
2. *Analyysiosio*: Kolmesta askeleesta koostuvan analyysiosion ensimmäisessä askeleessa tunnistetaan arkkitehtuuriratkaisut, jotka tukevat laadullisten vaatimusten täyttymistä. Toisessa askeleessa laatuominaisuuksiin liitetään esimerkkitilanteita, joissa laadullisen ominaisuuden vaikutus tulee käytännössä esille. Esimerkkitilanteet painotetaan sen mukaisesti, kuinka tärkeä niiden toteutus on järjestelmän kannalta ja kuinka vaikea ne on toteuttaa. Laadullisten ominaisuuksien ja esimerkkitilanteiden yhteyksistä muodostetaan laatupuu (*utility tree).* Kolmannessa askeleessa skenaarioiden ja laadullisten ominaisuuksien väliin lisätään sitä tukevat arkkitehtuuriratkaisut. Askeleen lopputuloksena muodostetaan laatupuu, jossa kuvataan laadullinen ominaisuus (esimerkiksi: saavutettavuus), sen tarkenne (esimerkiksi: laitevika) ja skenaario (esimerkiksi: käynnistä laite uudestaan minuutin sisällä laitteen rikkoutuessa). Laatupuun perusteella voidaan tunnistaa järjestelmän eri osien riskejä, turvallisia ratkaisuja, herkkyyskohtia ja tasapainokohtia.
3. *Testausosio*: Aiemmissa vaiheissa luodut skenaariot ovat pääosin järjestelmän kehittäjien tekemiä. Tässä vaiheessa sidosryhmiltä otetaan vastaan skenaarioita esimerkiksi käyttötilanteista, muutostilanteista tai rasitustilanteista. Mikäli sidosryhmien luomia skenaarioita ei löydy laatupuusta, tulee ne lisätä sinne ja analysoida vaikutukset järjestelmään. Uusi laatupuu analysoi-

daan ja arkkitehti kertoo, miten arkkitehtuuri vastaa skenaarioihin. Testausosion viimeisessä askeleessa voidaan testausosio käydä iteroiden läpi eri osanottajajoukolla, mikäli uusia skenaarioita ei löytynyt.

4. *Raportointiosio*: Arvioinnin tulokset esitetään kaikille osanottajille. Tuloksena on ohjelmistoarkkitehtuurin kipupisteiden tunnistus ja riskien tunnistus. Analyyseistä voidaan koostaa myös riskiteemoja, jossa kuvataan riskejä, joiden taustalla on sama (arkkitehtuurillinen) syy.

ATAM-menetelmä on sellaisenaan varsin raskas: Koskimiehen ja Mikkosen (2005) mukaan koko arviointiprosessi kestää noin kolme päivää ja vaatii eri sidosryhmien sitoutumista prosessiin. Se ei tämän takia välttämättä sovi sellaisenaan esimerkiksi pienen ohjelmiston ketterän kehityksen avuksi.

3.3 Arkkitehtuurin suunnittelu ja arviointi ketterässä ohjelmistokehityksessä

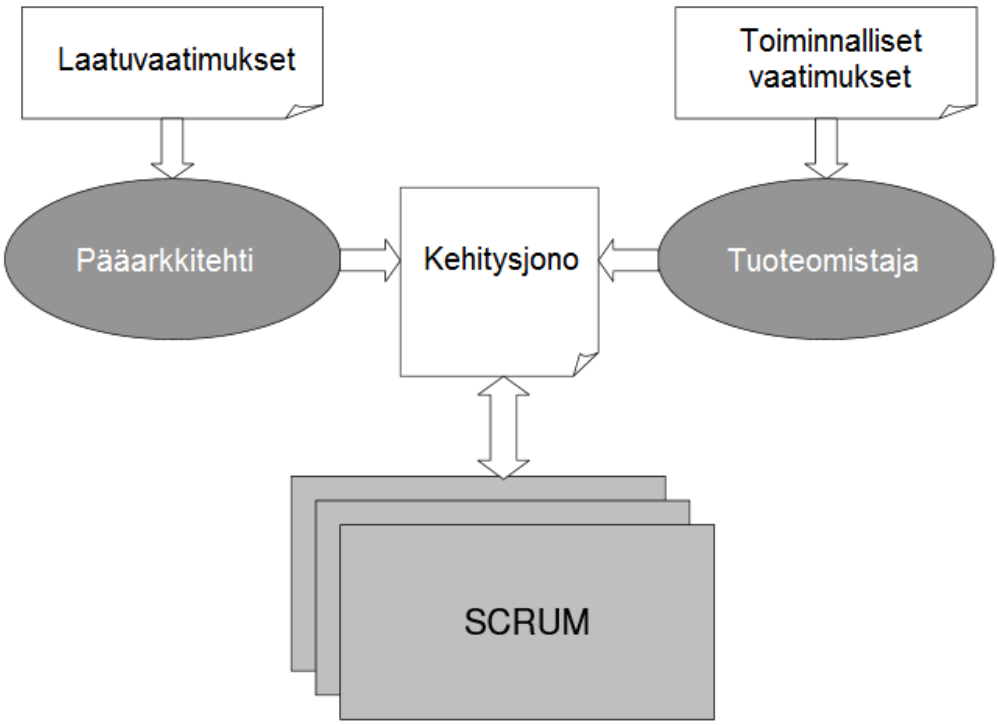
Ohjelmistoarkkitehtuurin suunnittelun asema ja ajoitus ohjelmistokehitysprosessin suhteen jakaa mielipiteitä. Ketterän ohjelmistokehityksen kannattajien keskuudessa saatetaan pitää ohjelmistoarkkitehtuurin laajamittaista etukäteissuunnittelua paheksuttavana asiana, joka ei sovi ketterän kehityksen periaatteisiin. Ohjelmistoarkkitehtuurin suunnittelumenetelmien käytön pelätään johtavan laajaan dokumentaatioon, kasvaneeseen työmäärään ja paluuseen takaisin vanhaan ja kankeaan vesiputousmalliin. (Kruchten, 2010; Nord & Tomayko, 2006)

Ketteryyden ja arkkitehtuurin suunnittelun suhdetta ovat yleisellä tasolla tarkastelleet Abrahamsson ym. (2010), Akbari ja Sharifi (2012), Hadar ja Sherman (2012), Keuler, Wagner ja Winkler (2012), Kruchten (2010) sekä Durdik (2011). Abrahamssonin ym. (2010) mukaan arkkitehtuuri ja ketteryys eivät välttämättä ole toistensa vastakohtia. Kirjoittajien mukaan arkkitehtuurin suunnittelu ketterässä kontekstissa rakentuu ei-toiminnallisten vaatimusten huomioimiseen. Heidän mukaan hyvin usein ei-toiminnalliset vaatimukset sivutetaan ketterässä kehityksessä, koska ne eivät suoraan vaikuta toteutettaviin ominaisuuksiin. Olennainen osa kehitystyötä on siis kiinnittää huomiota ei-toiminnallisiin vaatimuksiin arkkitehtuurin näkökulmasta. Hadar ja Sherman (2012) ovat tutkineet ohjelmistoarkkitehtien toimintaa ketterässä ja perinteisessä ohjelmistokehityksessä. Oleellisin ero vaikuttaa olevan siinä, että ketterät ohjelmistoarkkitehdit osallistuvat ohjelmiston kehitykseen myös vaatimusmäärittelyn jälkeen. Durdik (2011) on käsitellyt arkkitehtuurin kuvaamista ja kuvausprosessia ketterässä kehityksessä.

Ketterän ohjelmistokehityksen ja arkkitehtuurisuunnittelun integrointia menetelmätasolla ovat tarkastelleet XP-menetelmän yhteydessä Nord, Tomayako ja Wojcik (2006), Breivold ym. (2010) sekä Scrum-menetelmän yhteydessä Faber (2010), Eloranta ja Koskimies (2011) sekä Isham (2008). Madisonin (2010) esitys koskee Scrumia ja XP:tä. Akbari ja Sharifi (2012) ovat tutkineet perinteis-

ten arviointimenetelmien käyttöä ketterässä kehityksessä. Keuler ym. (2012) esittelevät viitekehyksen ketterään ohjelmistokehitykseen, jonka avulla ohjelmistokehitykseen tuodaan mukaan arkkitehtuurin suunnittelua. Esitetty viitekehys ei ole arkkitehtuurin suunnittelumenetelmä, vaan se on tarkoitettu kehittäjien työkaluksi tukemaan arkkitehtuuria huomioivaa ohjelmistokehitystä. Seuraavassa kerrotaan lähemmin Faberin (2010) ja Nordin ym. (2006) esityksistä.

Faber (2010) esittää konkreettisen keinon huomioida ohjelmistoarkkitehtuuri ketterässä kehityksessä. Scrum-viitekehykseen (Schwaber & Beedle, 2002) perustuen hän esittää mallin, jossa tuotejonon (*Product backlog*) priorisoinnista vastaavan tuoteomistajan (*Product Owner*) lisäksi priorisoinnista vastaa pääarkkitehti (*Architecture Representative*). Käytännössä pääarkkitehti on vastuussa kehitettävän järjestelmän laatuvaatimuksista, niiden keräämisestä ja priorisoinnista. Tuoteomistaja ja pääarkkitehti priorisoivat tiiviissä yhteistyössä toteutettavia asioita. Kuvio 8 esittää, kuinka pääarkkitehti ja tuoteomistaja osallistuvat Scrum-pohjaiseen ohjelmistokehitysprosessiin. Tällä tavoin toimittaessa järjestelmän arkkitehtuurillinen rakenne otetaan jatkuvasti huomioon uusien ominaisuuksien suunnittelun yhteydessä, jolloin järjestelmän jatkokehitys on tulevaisuudessa helpompaa.



Kuvio 8: Periaatekuva pääarkkitehdin roolista Faberin esittämässä Scrum-muunnelmassa. (mukaillen Faber, 2010 s. 11)

Nord ja Tomayko (2006) ovat tutkineet arkkitehtuurin suunnittelumenetelmien ja ketterän kehityksen menetelmien yhdistämistä. He ovat keskittyneet siihen, millä tavoin ominaisuusvetoista arkkitehtuurisuunnittelumenetelmää (Bachmann & Bass, 2001), Extreme Programming (XP) -menetelmää (Beck, 2000) sekä

kahta arkkitehtuurin arviointimenetelmää (ATAM (Kazman ym., 1998), CBAM (Bass ym., 2003)) voidaan soveltaa yhdessä. Nord ja Tomayko (2006) toteavat, että arkkitehtuurin suunnittelu- ja arviointimenetelmät voivat parantaa XP:n käytäntöjä. Näiden menetelmien käyttö tekee ohjelmiston kehittämisestä helpompaa ja johdonmukaisempaa. Nord ja Tomayko (2006) painottavat, että ominaisuusvetoisen arkkitehtuurisuunnittelun tarkoituksena on sellaisen kokonaisarkkitehtuurin suunnittelu, joka täyttää laatuvaatimukset. Kokonaisarkkitehtuuri onkin asia, joka hyvin usein unohtuu XP-kehittäjiltä, jolloin seurauksena syntyy kehitystyön edetessä hankalasti laajennettava ja epävarma järjestelmä. Nord ja Tomayko (2006) esittävät tavan, jolla ominaisuusvetoinen arkkitehtuurisuunnittelu ja ohjelmistoarkkitehtuurin arviointimenetelmät voidaan integroida osaksi XP-menetelmän eri vaiheita:

- *Suunnittelu ja kertomukset*: XP:n olennaiset rakennuspalat, eli käyttäjäkertomukset, täydennetään laadullisilla vaatimuksilla. Menetelmänä laadullisten ominaisuuksien keräämisessä voi olla a esimerkiksi QAW-menetelmä (Quality Attribute Worshop) (Barbacci ym., 2003a). Menetelmä on hyvin samankaltainen kuin ATAM-arviointimenetelmän (Kazman ym., 1998) skenaarioiden kerääminen ja analysointi.
- *Toteutus:* Ominaisuusvetoista arkkitehtuurisuunnittelumenetelmää soveltamalla luodaan korkean tason arkkitehtuuri, joka mahdollistaa tulevien ominaisuuksien vaikutuksen arvioinnin ohjelmiston rakenteeseen. Arkkitehtuuria suunnitellaan ainoastaan pakollinen määrä, kuitenkin vähintään siten että järjestelmä tavoittaa ennalta määritellyt laadulliset vaatimukset.
- *Analysointi ja testaus:* Suunnitelmien analysointi ATAM ja CBAM -menetelmillä mahdollistaa aikaisen palautteen saamisen. Arkkitehtuurin arviointi voi tuoda esille riskejä, tarvetta kompromissien tekoon tai arvioita arkkitehtuurillisten ratkaisujen vaikutuksesta ohjelmiston kehitykseen kuluvaan aikaan, ennen kuin varsinainen toteutus on kerinnyt alkaa.

3.4 Yhteenveto

Tässä luvussa käsiteltiin ohjelmistoarkkitehtuurin suunnittelua ja arviointia. Ohjelmistoarkkitehtuuri voi syntyä vähitellen ohjelmistokehityksen edetessä tai sitä voidaan suunnitella johdonmukaisesti ohjelmistoarkkitehtuurin suunnittelumenetelmien avulla. Johdonmukaisella suunnittelulla voidaan saavuttaa todellista hyötyä, koska yleensä tällöin tehty arkkitehtuuri vastaa ohjelmistolle asetettuja laadullisia vaatimuksia ja palvelee paremmin tulevia tarpeita. Luvussa esitettiin kaksi ohjelmistoarkkitehtuurin suunnittelumenetelmää, ominaisuusvetoinen arkkitehtuurisuunnittelu sekä RUP 4+1 mallin suunnittelumenetelmä.

Ohjelmistoarkkitehtuurin suunnittelun lisäksi luvussa kerrottiin ohjelmistoarkkitehtuurin arvioinnista. Ohjelmistoarkkitehtuurin arvioinnin todettiin olevan haastava tehtävä, joka perustuu laadullisiin vaatimuksiin. Ohjelmisto-

arkkitehtuurin arvioinnin merkitystä tuotteen laadun takaajana korostettiin ja tarjottiin esimerkkejä arvioinnin suorittamiseksi. Luvussa esitettiin SEI-instituutin kehittämä ATAM-arviointimenetelmä, joka on yksi tunnetuimmista ja käytetyimmistä ohjelmistoarkkitehtuurin arviointimenetelmistä.

Lopuksi kerrottiin, kuinka ohjelmistoarkkitehtuurin suunnittelua ja arviointia voidaan hyödyntää ketterässä kehityksessä. Luvussa esitettiin Faberin (2010) näkemys laadullisten vaatimusten hyödyntämisessä Scrum-viitekehyksessä sekä Nordin ja Tomaykon (2006) tutkimus suunnittelu- ja arviointimenetelmien hyödyntämisestä Extreme Programming (XP) –menetelmän yhteydessä.

4 TAPAUSTUTKIMUKSEN TOTEUTTAMINEN

Tutkielman empiirisen osan tavoitteena on selvittää, kuinka ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmiä voidaan soveltaa käytännössä. Tutkimuksessa valitaan yksi ohjelmistoarkkitehtuurin suunnittelumenetelmä, jota sovittamalla ja käyttämällä luodaan kohdeorganisaation järjestelmälle uusi ohjelmistoarkkitehtuuri. Saatua lopputulosta ja suunnitteluprosessin sujuvuutta arvioidaan kehittämisehdotuksien saamiseksi. Lopputuloksen arvioimisessa ja vahventamiseksi käytetään apuna ohjelmistoarkkitehtuurin arviointimenetelmää. Tässä luvussa kerrotaan tutkimusmenetelmästä, tutkimuskohteesta, tutkimusprosessista ja -mallista sekä tiedonkeräämisestä.

4.1 Tutkimusmenetelmä

Tähän tutkimukseen on tutkimusmenetelmäksi valittu tapaustutkimus. Järvinen ja Järvinen (2011) määrittelevät tapaustutkimuksen luonteeltaan tapausta kuvailevaksi, teoriaa testaavaksi tai teoriaa luovaksi. Tapaustutkimuksen avulla voidaan selvittää, millaisia käsiterakenteita, malleja tai teorioita yksittäinen tapaus voi paljastaa. On toki mahdollista, ettei mitään uutta löydy, mutta siltikin syntyy tietoa siitä, millainen tutkittava asia on. Yin (2009) määrittelee tapaustutkimuksen keinoksi, jonka avulla voidaan tutkia empiirisiä aiheita ennalta suunnitellun menetelmän avulla. Yinin (2009) mukaan tapaustutkimus voi vastata kysymyksiin ”miten”, ”miksi” ja ”mitä”. Se sopii myös kuvailevaan tutkimukseen ja arviointitutkimukseen (Järvinen & Järvinen, 2011).

Runeson ja Höst (2009) pitävät tapaustutkimusmenetelmää erityisen sopivana ohjelmistotuotantoon liittyviin tutkimuksiin. Kirjoittajien mukaan ohjelmistotuotannon (*software engineering*) tutkimuksessa tapaustutkimukset eivät ole kuitenkaan saavuttaneet vielä niin vakiintunutta ja yleisesti tunnistettua asemaa kuin tietojärjestelmätieteen tutkimuksessa (*information systems research*). Runesonin ja Höstin (2009) mukaan alan tutkimukset yleensä käsittelevät jotakin nykyajan ilmiötä luonnollisessa kontekstissaan. Heidän mukaansa tapaus-

tutkimus voi olla tutkiva (*exploratory*), kuvaileva (*descriptive*), selittävä (*explanatory*) tai parantamiseen tähtäävä (*improving*). Tutkiva tapaustutkimus selvittää, mitä ympäristössä tapahtuu, ja sen tarkoituksena on luoda uusia käsityksiä, ideoita ja hypoteeseja ilmiöistä. Kuvaileva tapaustutkimus nimensä mukaisesti kuvailee tilannetta tai ilmiötä. Selittävä tapaustutkimus pyrkii löytämään selityksiä tilanteeseen tai ongelmaan, mikä tapahtuu yleensä kausaalisuhteita analysoimalla. Parantamiseen tähtäävä tapaustutkimus pyrkii kehittämään ilmiötä jostakin näkökulmasta katsoen.

Tässä tutkimuksessa käsiteltävä ilmiö on ohjelmistoarkkitehtuurin suunnittelumenetelmän hyödyntäminen käytännössä. Tutkimus sijoittuu ohjelmistokehityksen alueelle, koska siinä käsitellään ohjelmistokehityksessä tapahtuvia prosesseja. Tapaustutkimus on luonteeltaan tapausta kuvaileva, tutkiva ja jossain määrin parantamiseen tähtäävä (vrt. Yin, 2009; Runeson & Höst, 2009). Tutkimuksessa kuvaillaan olemassa olevaa ohjelmistoarkkitehtuuria ja tutkitaan mitä tapahtuu, kun sitä muutetaan suunnitteluprosessin aikana. Suunnitteluprosessia varten valitaan käyttöön yksi suunnittelumenetelmä. Toissijaisesti tässä tutkimuksessa pyritään myös kehittämään kohdeyrityksen arkkitehtuurin suunnittelua.

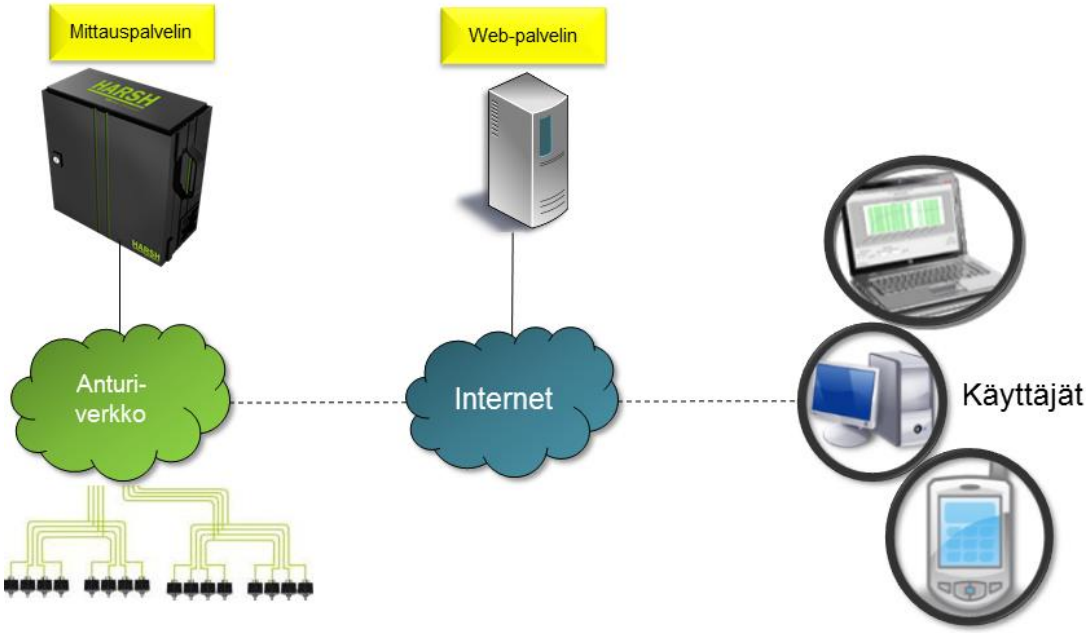
Kohdeorganisaatio ei ole aivan luonnollisessa tilassaan, koska tutkija auttaa ohjelmistoarkkitehtuurin suunnitteluprosessin läpiviennissä ulkopuolisena asiantuntijana. Tutkijan osallistuminen sekä prosessin läpivientiin, että prosessin ja tulosten analysointiin voi vaikuttaa tutkimuksen validiteettiin. Toisaalta tutkija voi näin toimiessaan saada ensikäden havaintoja kuinka prosessi käytännössä toimii.

4.2 Tutkimuskohde

Tapaustutkimuksessa voidaan tarkastella yhtä tapausta tai useita tapauksia (Järvinen & Järvinen, 2011). Tässä tutkimuksessa tapauksen kohteena on tietyn ohjelmistotuotteen arkkitehtuurin suunnitteluprosessi. Ohjelmistotuotetta kutsutaan jatkossa WBS-järjestelmäksi. Tässä tutkimuksessa käsitellään vain yhtä tapausta, koska yritykseen ja sen ohjelmistokokonaisuuteen perehtyminen vievät resursseja. Tämän lisäksi yrityksessä tutkijan toimesta suoritettava ohjelmistoarkkitehtuurin suunnittelu on aikaa vievää.

Tapaustutkimus tehdään organisaatiossa, joka kehittää palveluratkaisua teollisuuden tarpeisiin. WBS-järjestelmä koostuu ethernet-tietoverkkoon liitettävistä antureista sekä laitteisto- ja ohjelmistoratkaisuista. Anturien tuotantokoneista mitatun tiedon avulla asiakasyritykset voivat seurata web-käyttöliittymän avulla laitteidensa käyttöastetta (tehokkuutta) sekä suorittaa ennakoivaa kunnonvalvontaa laitteistorikkojen ja tuotantokatkojen välttämiseksi. Ohjelmiston ja antureiden avulla tehtävän käyttöasteen valvonnan avulla asiakasyritykset voivat parantaa laitteiden tehokasta käyttöaikaa ilman työläitä integraatioita laitteiden automatiikkaan. Tämän vuoksi tarjottava ohjelmisto- ja laiteratkaisu sopii hyvin erilaisiin käyttötilanteisiin akkuporakoneesta suuriin

paperikoneisiin. Jatkuvan seurannan ja analysoinnin avulla tuotannon tehokkuuden kehittäminen ja kunnonvalvonta on mahdollista toteuttaa. Kuvio 9 esittää yleiskuvan järjestelmän rakenteesta. Anturien välittämä tieto viedään ethernet-verkossa mittauspalvelimen kautta internetiin kytkettyyn web-palvelimeen, josta käyttäjät voivat hakea tarvitsemansa tiedon eri päätelaitteilla. Tarkempi kuvaus ohjelmiston arkkitehtuurista ja toiminnasta esitetään luvussa 5.1.



Kuvio 9: Anturipohjaisen tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän toimintaperiaate

Yrityksen sadasta henkilöstä noin kymmenen työskentelee WBS-järjestelmän kehityksessä. Tiimiin kuuluu ohjelmistokehittäjien lisäksi laiteratkaisuista ja myynnistä vastaavia henkilöitä. Ohjelmistokehitystä suoritetaan Scrum-viitekehystä (Schwaber & Sutherland, 2013) mukaillen.

WBS-järjestelmän laite- ja ohjelmistoratkaisun kehittäminen on alkanut vuonna 2006. Ohjelmistoratkaisun kehittämisen aktiivinen vaihe on kuitenkin alkanut vuodesta 2012, jolloin ohjelmistokehitystiimin kokoa on alettu kasvattamaan. Nykyinen ohjelmistokokonaisuus on syntynyt erilaisten liiketoiminta-ajatusten perusteella pala kerrallaan.

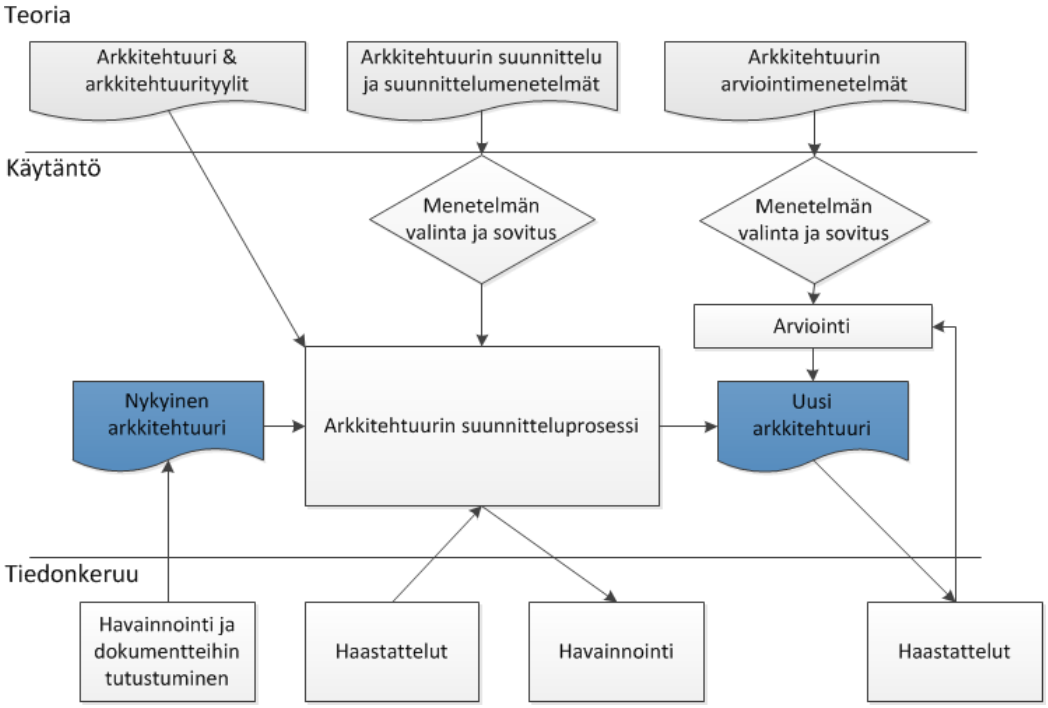
Nyt yrityksessä ollaan suunnittelemassa ohjelmistotuotteen siirtämistä pilvipalveluun, jolloin sitä voidaan nykyistä paremmin tarjota palveluliiketoimintamallin (*Software as a Service, SaaS*) avulla. Tämän vuoksi yrityksessä halutaan selvittää, minkälaisia arkkitehtuurillisia ratkaisuja ohjelmistoon täytyy tehdä, jotta ohjelmiston tarjoaminen palvelupohjaisesti web-teknologioiden avulla on mahdollista. Arkkitehtuurisuunnittelun tavoitteena on mahdollistaa nämä tarpeet ja luoda suunnitelma ohjelmistokokonaisuuden tulevaisuudelle.

Järvinen ja Järvinen (2011) suosittelevat tutkimuskohteen valintaa sen perusteella, kuinka informatiivisesti edustava se on. Organisaatio on muutostilanteessa, jossa siltä vaaditaan johdonmukaista arkkitehtuurin suunnittelua. Orga-

nisaatiolla on aito tarve saada tietoa arkkitehtuurin suunnittelusta. Tapaustutkimuksen kohteeksi valittu arkkitehtuurin suunnittelu tarjoaa mahdollisuuden tutkia, miten arkkitehtuurin suunnittelu kannattaisi tehdä ko. ohjelmistotuotteen osalta. Näin organisaatiossa tapahtuvaa prosessia on mahdollista tutkia. Tämän lisäksi toteutettava tutkimus antaa tietoa, kuinka ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmiä voidaan hyödyntää muissa ko. organisaation kehityshankkeissa ja mahdollisesti vastaavien PK-yritysten ketterän ohjelmistokehityksen konteksteissa.

4.3 Tutkimusprosessi ja –malli

Tutkimus voidaan jäsentää kolmeen kokonaisuuteen, teoriaa, käytäntöä ja tiedonkeruuta koskeviin kokonaisuuksiin (Kuvio 10). Teoriaosuuden ja tiedonkeruun tarkoituksena on tukea käytännön tekemistä. Käytännön tekemisen yhteydessä toteutetulla havainnoinnilla ja haastatteluilla saadaan tietoa koko prosessin onnistumisesta, jolloin tehdystä työstä voidaan löytää parannuskohteita.



Kuvio 10: Tutkimusprosessi

Ennen tämän tutkimuksen aloittamista kohdeyrityksessä on havaittu nykyisen arkkitehtuurin joustamattomuus uusia vaatimuksia ajatellen. Tämän takia yrityksessä on nähty tarve ohjelmiston uudelle rakenteelle ja sitä kautta ohjelmistoarkkitehtuurin suunnittelulle.

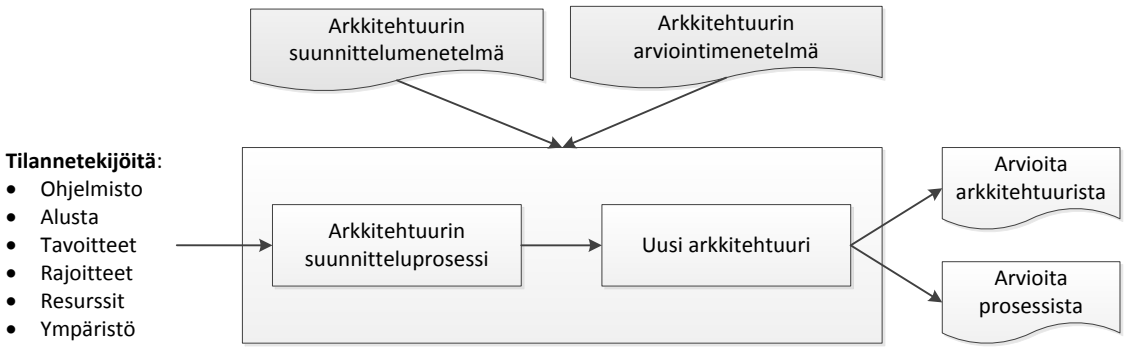
Tutkimustyö on alkanut tutustumalla ohjelmistoarkkitehtuuria, arkkitehtuurityylejä, arkkitehtuurin suunnittelua ja arviointia koskevaan kirjallisuuteen. Tällä tavalla on saatu konkreettisia työkaluja ja ajatuksia siitä, kuinka arkkitehtuurillisia ongelmia voidaan ratkaista. Tätä kokonaisuutta koskevia tuloksia on esitetty luvuissa 2 ja 3.

Toisessa vaiheessa on tutustuttu nykyiseen ohjelmistoarkkitehtuuriin dokumenttien ja omien havaintojen perusteella. Tämä perehtyminen on ollut pitkäaikainen prosessi ja on käsittänyt käytännön ohjelmistokehityksen tekemistä kohdeyrityksessä. Näin tutkijalle on syntynyt konkreettinen kuva nykyisestä ohjelmistoarkkitehtuurista useammasta eri näkökulmasta.

Kolmannessa vaiheessa yrityksessä toteutetaan arkkitehtuurisuunnittelu valitulla arkkitehtuurin suunnittelumenetelmällä. Yrityksen henkilöstöä ja asiakkaita haastatellaan ja käytetään apuna uuden arkkitehtuurin tuottamiseksi. Arkkitehtuurin suunnittelussa käytetään apuna havaintoja, jotka ovat syntyneet työskenneltäessä yrityksessä. Samalla omaa ja muiden työskentelyä havainnoidaan suunnittelumenetelmän toteuttamisen eri vaiheissa.

Tuotettava ohjelmistoarkkitehtuuri arvioidaan käyttämällä valittua ohjelmistoarkkitehtuurin arviointimenetelmää. Yrityksen henkilöstöä haastatellaan ohjelmistoarkkitehtuurin onnistumisen arvioimiseksi. Oleellisessa osassa arkkitehtuurin arvioinnissa ovat eri asiakasskenaariot, joita vasten tehtyä arkkitehtuuria voidaan arvioida. Lopuksi tehty työ ja kokemukset raportoidaan tutkielmassa.

Tutkimusmalli voidaan esittää pääpiirteittäin kuvion 11 tapaan. Tutkimuksessa toteutetaan ohjelmistoarkkitehtuurin suunnitteluprosessi, jonka seurauksena syntyy uusi ohjelmistoarkkitehtuuri. Suunnitteluprosessissa käytetään apuna hankittua tietämystä ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmistä. Suunnitteluprosessi on organisaatiossa aina ainutkertainen, sillä siihen vaikuttavat lukuisat eri tilannetekijät. Tilannetekijöinä ovat suunnittelun kohteena oleva ohjelmisto, mahdollisesti kiinnitetty alusta, suunnittelulle asetetut tavoitteet ja -rajoitteet, suunnitteluun käytettävissä olevat resurssit sekä organisationaalinen ja tekninen ympäristö, jonka osana kehitettävän ohjelmiston on määrä olla. Tutkimuksessa saadaan arvioita ohjelmistoarkkitehtuurin suunnitteluprosessin ja arkkitehtuurin laadusta.



Kuvio 11: Tutkimusmalli

4.4 Tiedonkerääminen

Yin (2009) sekä Runeson ja Höst (2009) suosittavat tiedonkeruuta useasta eri lähteestä. Lähteitä voivat olla dokumentit, arkistot, haastattelut, vapaa havainnointi, osallistuva havainnointi ja fyysiset luomukset. Usean tietolähteen käytön avulla parannetaan tutkimuksen validiteettia. Tässä tutkimuksessa tietolähteinä käytetään dokumentteja, haastatteluja ja havainnointia (vrt. Kuvio 10). Tiedonkeruu on tapahtunut aikavälillä toukokuu 2014 – marraskuu 2014. Tiedonkeräämisen tarkoituksena on saada tietoa siitä, millainen ohjelmiston arkkitehtuurin tulisi olla, kuinka onnistunut analyysin kohteena oleva arkkitehtuuri on ja kuinka onnistunut suunnitteluprosessi oli.

Tiedonkeruu jakautuu kolmeen osa-alueeseen, jotka on toteutettu ajallisesti peräkkäin. Jokaisessa osa-alueessa käytetään useampaa tiedonkeruumenetelmää. Tiedonkeruun osa-alueet ovat:

- tärkeimpien laadullisten ja toiminnallisten vaatimusten selvittäminen ohjelmistoarkkitehtuurin suunnittelemiseksi,
- toteutetun ohjelmistoarkkitehtuurin arviointi,
- toteutetun prosessin arviointi.

Ohjelmiston laadullisten ja toiminnallisten vaatimusten selvitystyö on aloitettu toukokuussa 2014 havainnoimalla ohjelmistoa ja liiketoimintaympäristöä. Aktiivisin vaihe havainnoinnissa on ollut touko-elokuu 2014, jolloin tutkija työskenteli päätoimisesti kohdeorganisaatiossa. Havainnoinnin avulla keskityttiin löytämään nykyisen arkkitehtuurin ongelmakohtia sekä havaitsemaan asiakasvaatimuksia. Tulevia tarpeita pyrittiin virallisten teiden lisäksi havaitsemaan ns. kahvipöytäkeskusteluista, jossa järjestelmän hyödyntämismahdollisuuksia eri käyttökohteissa pohdittiin. Tehtyjä havaintoja pyritään vahvistamaan ja uusia havaintoja keräämään haastattelemalla asiakkaan edustajia ja ohjelmistokehittäjiä. Havaintojen ja haastatteluiden perusteella kootaan vaatimuslista, jota käytetään arkkitehtuurin suunnittelussa. Vaatimusten selvittämiseen käytetty haastattelupohja on esitetty liitteessä 1.

Ohjelmistoarkkitehtuurin suunnitteluprosessin tuottama tuotos eli *arkkitehtuuri arvioidaan*. Arvioinnin keskeisenä metodina on arkkitehtuurin arviointimenetelmän käyttö. Tehtyä arkkitehtuuria arvioidaan laadullisten ominaisuuksien perusteella, jotka jaetaan Reekien ym. (2006) mukaan kahteen luokkaan: ajonaikaisiin ja ei-ajonaikaisiin. Tuotettu ohjelmistoarkkitehtuuri annetaan tämän lisäksi arvioitavaksi yrityksen johdolle ja ohjelmistokehittäjille. Heidän näkemyksiään ohjelmistoarkkitehtuurista kerätään haastatteluilla.

Lopuksi koko *toteutetun suunnitteluprosessin toiminta arvioidaan*. Arvioinnin mittareina ovat suunnitteluprosessin tuotoksen laatu sekä käytetyn prosessin toiminta, tehokkuus ja sopivuus tähän tapaustutkimukseen. Tässä yhteydessä käytetään havaintoja prosessin eri vaiheista sekä haastatteluja suunnitteluun osallistuneilta.

Riskitekijänä haastatteluissa on kehitystiimin pieni koko. Lisäksi asiakkaiden kiinnostus prosessin eri vaiheisiin voi olla vaihtelevaa. Kohdeorganisaation tavoitteet määrittelevät sen, missä määrin ohjelmistokehitystiimin ja asiakkaiden resursseja voidaan tutkimuksessa käyttää. Tästä syystä suurin osa tiedonkeruusta tulee keskittymään ensimmäiselle osa-alueelle, eli laadullisten vaatimusten keräämiseen.

5 TULOKSET

Tässä luvussa esitetään tapaustutkimuksen kohteena olleen ohjelmiston aiempi arkkitehtuuri, kerrotaan toteutusta ohjelmistoarkkitehtuurin suunnittelusta, raportoidaan suunnittelun tuloksena syntyneestä arkkitehtuurista, arvioidaan tuotettua arkkitehtuuria sekä vertaillaan vanhaa ja uutta arkkitehtuuria laadullisten ominaisuuksien avulla. Esityksessä palveluntarjoajalla tarkoitetaan kohdeorganisaatiota ja asiakkailla palveluntarjoajan asiakkaita, eli palvelun käyttäjiä.

5.1 Nykyinen ohjelmistoarkkitehtuuri

Nykyinen ohjelmistokokonaisuus koostuu neljästä pääkomponentista: anturit, mittauspalvelin, web-käyttöliittymä (WebUI) sekä konfigurointityökalu. Lähiverkkoon liitettäviä anturimalleja on kaksi. CM301-anturi mittaa kiihtyvyyttä ja lämpötilaa. Toisessa anturimallissa (CM300) on kolme analogista ja kolme digitaalista sisäänmenoa, joihin voidaan liittää käytännössä mitä tahansa muita antureita (esim. paine-, paikka-, uv- tai voima-anturi). Anturit ovat PoE-liitännäisiä (*Pover Over Ethernet*), joten ne eivät tarvitse ulkoista virtaa toimiakseen. Anturit ovat riippumattomia muista ohjelmistoratkaisuista, ja niitä voidaan konfiguroida web-selaimen välityksellä. Nykyisenkaltaisen anturien kehitystyö on aloitettu vuonna 2007.

Aiemmin mittauspalvelimen kaltaista roolia toteutti Windows-työpöytäohjelmisto, jonka avulla käyttäjä pystyi lataamaan mittaustietoa omalle tietokoneelleen, analysoimaan sitä sekä muodostamaan mittaustiedon perusteella raportteja. Työpöytäohjelmiston kehitys aloitettiin vuonna 2010. Asiakkaiden halutessa jatkuvaa varmatoimista seurantaa anturitiedon perusteella, päätettiin vuonna 2012 aloittaa itsenäisen mittauspalvelimen kehitys Linux-käyttöjärjestelmän päälle. Työpöytäohjelmiston rooliksi jäi toimia konfigurointityökaluna Linux-pohjaiselle mittauspalvelimelle.

Nykyisenkaltaisen Linux-käyttöjärjestelmän päälle rakennetun mittauspalvelimen tehtävänä on kerätä anturilta mittaustietoa ja analysoida sitä käyttäjän tekemän konfiguraation mukaisesti. Konfigurointi käsittää mm. antureiden asentamisen, analyysifunktioiden määrittelyn, haluttujen raporttien määrittelyä, hälytysraja-arvojen asentamista, hälytyksien vastaanottajien määrittelyä sekä WebUI:n toiminnallisuuden ja ulkoasun määrittelyä. Mittauspalvelin lähettää käyttäjän määrittämät, mittaustiedosta lasketut avainluvut ja muun tarvittavan konfiguraation eteenpäin WebUI:lle, josta käyttäjä voi käydä katsomassa reaaliaikaista avainlukutietoa, avainlukuhistoriaa ja avainluvuista koostettuja raportteja. Mittaustietoa (raakadata) syntyy niin suuri määrä, ettei sitä ole mahdollista tarkastella WebUI:sta. Tämän takia oleellista on laskea mittaustiedosta käyttäjää kiinnostavat avainluvut. Mittauspalvelin voi lähettää sähköposti- ja tekstiviestihälytyksiä, mikäli käyttäjän anturille määrittämät raja-arvot täyttyvät, tai jos mittauspalvelimen toiminnassa on häiriöitä. Mittauspalvelimella tulee olla yhteys antureille, joten käytännössä ne on sijoitettava samaan fyysiseen lähiverkkoon. Vaihtoehtoisesti anturin ja mittauspalvelimen välinen yhteys voidaan toteuttaa internetin yli salatusti esimerkiksi VPN-ratkaisulla. Tietoliikenneyhteyksien nopeus ja epävarmuus pakottavat useissa asiakastilanteissa mittauspalvelimen fyysisesti anturien lähelle.

Asiakkaiden halutessa reaaliaikaista tietoa antureilta eri päätelaitteille aloitettiin vuonna 2013 web-käyttöliittymän (WebUI) rakentaminen. Aiemmin järjestelmäkokonaisuus käsitti ainoastaan konfigurointityökalun ja mittauspalvelimen, joka lähetti sähköpostitse tai tekstiviestein tarvittavia hälytyksiä ja raportteja käyttäjälle. WebUI voi sijaita samalla fyysisellä palvelimella mittauspalvelimen kanssa. Useissa asiakastapauksissa WebUI sijaitsee yrityksen oman lähiverkon ulkopuolella, jotta siihen voidaan ottaa luotettavasti yhteyttä mistä päin internetiä tahansa. Reaaliaikaisen avainlukutiedon lisäksi asiakas voi tarkastella avainlukutiedon historiaa ja muodostettuja raportteja. Näiden ominaisuuksien avulla asiakas voi saada tietoa koneidensa käyttöasteesta (tehokkuus) tai suorittaa laitteilleen ennakoivaa kunnonvalvontaa. Kuvio 12 esittää yhden näkymän reaaliaikaiseen avainlukutietoon. Kuviossa esitetyssä näkymässä suoritetaan ennakoivaa kunnonvalvontaa. Mitattavia suureita ovat esimerkiksi kiihtyvyyspiikit eri kohdista moottoria, lämpötilat, kierrosluku ja paine. Näiden avainlukutietojen ja avainlukuhistorian perusteella asiantuntijat voivat tunnistaa häiriötilanteita tuotantokoneesta. Asiakkaan saama hyöty järjestelmän käyttämisestä muodostuu tilanteessa, jossa tuleva laiterikko tai poikkeustila voidaan ohjelmiston ansiosta havaita ennakolta. Tällöin vältetään kalliit tuotantokatkot.

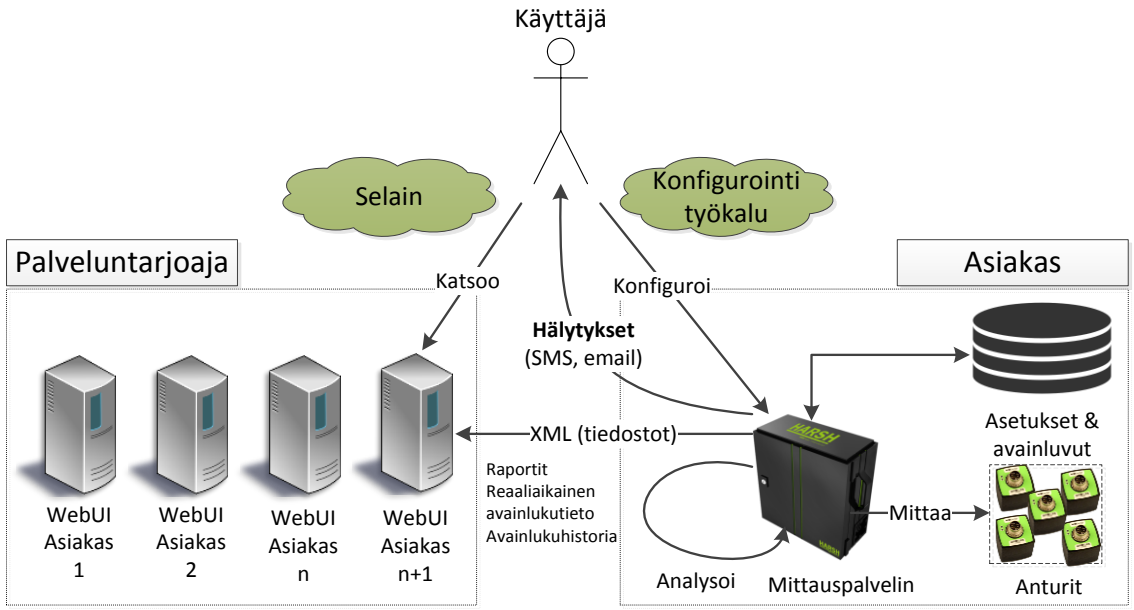
Edellä olevan kuvauksen perusteella voidaan todeta oleellisten ohjelmistokokonaisuuksien rakentuneen erilaisten tarpeiden pohjalta vähitellen. Alkuperäinen liiketoimintaidea oli toimia laitteistotoimittaja verkkoliitäntään kytkettäville teollisuuden tärinäantureille. Pian kuitenkin huomattiin, etteivät asiakkaat tarvitse itsenäistä anturia, ellei siihen liity jotakin ohjelmistoratkaisua ja palvelua tuotannon tilan seuraamiseen. Tämän vuoksi aloitettiin työpöytäkäyttöisen analysointiohjelmiston kehittäminen vuonna 2010 tukemaan anturirat-

kaisua. Samoin esimerkiksi vuonna 2007 antureiden kehitystyötä aloitettaessa ei ollut tietoa Linux-pohjaisesta mittauspalvelimesta, eikä vuonna 2010 työpöytäkehityksen aloituksen yhteydessä vielä osattu odottaa tarvetta web-käyttöliittymälle (WebUI).

Kuvio 12: Kuvakaappaus web-käyttöliittymän karttanäkymästä

Kuvio 13 esittää yleiskuvan ohjelmistoarkkitehtuurista. Kuviosta on nähtävissä lähtötilanteen oleellisin ongelma: käyttäjä suorittaa järjestelmän päivittäistä käyttöä web-käyttöliittymän avulla, mutta joutuu muutoksia tehdessään järjestelmän konfiguraatioon käyttämään työpöytäohjelmistoa. Syynä tähän on ohjelmiston menneisyys: asiakas on halunnut järjestelmään liitettävän tietoa näyttävän web-käyttöliittymän, eikä toteutusvaiheessa ole nähty akuuttia tarvetta konfiguroinnin siirtämiseen samalla WebUI:lle. Nykyiseen WebUI:hin voitaisiin tuoda konfigurointiominaisuutta, mutta nykyinen rajapinta mittauspalvelimen ja WebUI:n välillä ei mahdollista tiedonsiirtoa WebUI:lta mittauspalvelimelle. Mittauspalvelin voi lisäksi olla eristettynä palomuurin takana asiakkaan lähiverkossa. Sen takia tiedonsiirron toteuttaminen on käytännössäkin haastavaa. Kahden eri järjestelmän (konfigurointityökalu ja WebUI) käyttäminen heikentää järjestelmän käytettävyyttä: käyttäjän ei ole mahdollista muokata konfiguraatiota siellä, missä se tapahtuisi kaikista helpoiten. WebUI ei kerää saamaansa tietoa tietokantoihin, vaan käyttää mittauspalvelimen tarjoamia staattisia XML-tiedostoja. Tämän takia tietoa on hankalaa käsitellä ja jalostaa WebUI:n päässä.

Nykyisessä ratkaisussa eri asiakkaiden WebUI:t sijaitsevat usein fyysisesti samalla palvelimella. Niillä ei ole kuitenkaan yhteyttä toisiinsa, vaan jokainen käyttöliittymä toimii täysin itsenäisesti muista riippumatta. Nykyisessä arkkitehtuurissa ei ole otettu huomioon asiakkaita, jotka haluavat monen mittauspalvelimen tuottaman tiedon yhteen web-käyttöliittymään. Tällaista ominaisuutta voivat vaatia yritykset, joilla on toimipisteitä useassa paikassa ja jotka haluavat seurata tuotannon tilaa keskitetysti. Arkkitehtuuri ei siis mahdollista järjestelmän skaalautumista suureen mittakaavaan.

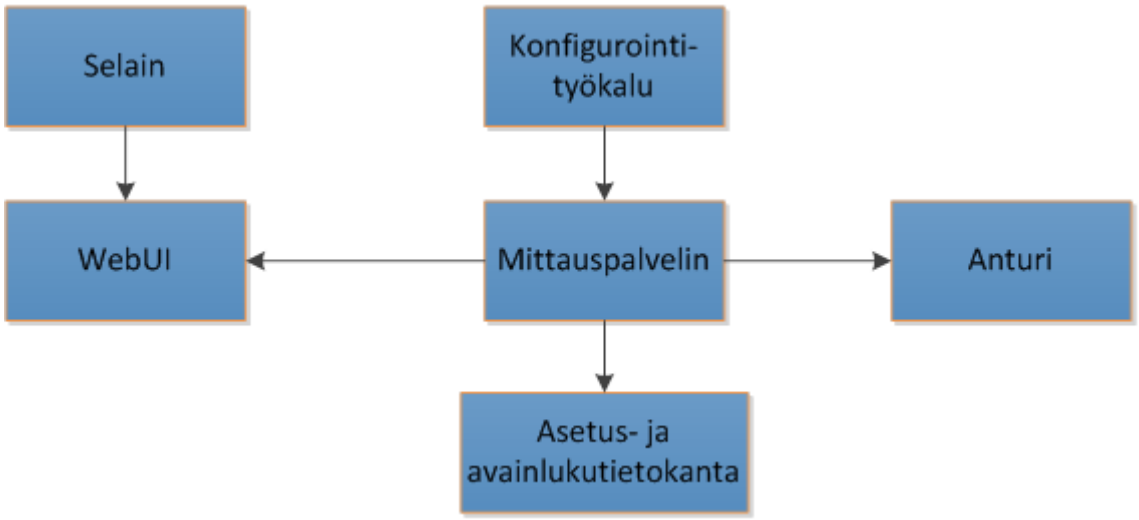


Kuvio 13: Nykyinen ohjelmistoarkkitehtuuri

Arkkitehtuurityylillisesti ohjelmistokokonaisuus noudattelee asiakas-palvelin-arkkitehtuurin (Buschmann ym., 1995) periaatteita. Anturi odottaa palvelimen roolissa mittauspalvelimen (asiakas) pyyntöä toimittaa mittaustietoa. Ladattu mittaustieto tallennetaan mittauspalvelimen tietokantaan, jota voidaan kuvata passiivisena palvelimena. Tietokanta odottaa siis mittauspalvelimen yhteydenottopyyntöjä. Mittauspalvelin odottaa passiivisena konfigurointityökalun yhteydenottoa, joten mittauspalvelin toimii tässä rajapinnassa palvelimena. Asiakkaan käyttämä konfigurointityökalu toimii asiakkaan roolissa ottaessaan yhteyksiä mittauspalvelimeen. Mittauspalvelimen ja web-palvelimen toiminta perustuu mittauspalvelimen yhteydenottoihin, joita WebUI odottaa. Näin ollen WebUI toimii palvelimen ja mittauspalvelin asiakkaan roolissa. Käyttäjän selain toimii asiakkaan roolissa muodostaen yhteyden WebUI:lle. Kuvio 14 esittää yhteenvedon tilanteesta eri osapuolien välillä. Nuolen suunta kertoo yhteydenoton suunnan, eli kumpi komponentti toimii palvelimena ja kumpi asiakkaana.

Koskimiehen ja Mikkosen (2005) sekä Buschmann ym. (1995) mainitsemista asiakas-palvelin-arkkitehtuurin haitoista (vrt 2.4.4) on organisaatiossa havaittu ongelmia, joita syntyvät, kun mittauspalvelimen tarjoamaa rajapintaa konfigurointityökalulle joudutaan muuttamaan. Pahimmassa tapauksessa rajapin-

nan muuttuessa joudutaan ensin päivittämään asiakkaan käytössä oleva mittauspalvelin, jonka jälkeen joudutaan päivittämään kaikki asiakkaan käytössä olevat konfigurointityökalut uusimpaan versioon. Mittauspalvelimen ja WebUI:n päivitettävyys on helpompaa, koska molemmat ohjelmistot ovat palveluntarjoajan hallinnassa ja ylläpidettävissä.



Kuvio 14: Yhteydenotot eri komponenttien välillä

5.2 Ohjelmistoarkkitehtuurin suunnittelu

Tässä alaluvussa kuvataan ensin suunnittelumenetelmän valintaa ja sen jälkeen esitetään, kuinka valittua suunnittelumenetelmää sovellettiin käytännössä tässä tapaustutkimuksessa.

5.2.1 Suunnittelumenetelmän valinta

Luvussa 3 todettiin, että osa ohjelmistoarkkitehtuurin suunnittelumenetelmistä on kehitetty teollisuudessa (Hofmeister ym., 2007) ja jotkut menetelmistä ovat raskaita käyttää vaatien muiden muassa eri sidosryhmien aktiivista osallistumista prosessiin (Kruchten, 2010; Nord & Tomayko, 2006). Ohjelmistoarkkitehtuurin raskasta suunnittelua ennalta saatetaan pitää jopa paheksuttavana, koska se ei sovi ketterän ohjelmistokehityksen periaatteisiin. Ohjelmistoarkkitehtuurin suunnittelumenetelmien käytön pelätään johtavan laajaan dokumentaatioon, kasvaneeseen työmäärän ja paluuseen takaisin vanhaan ja kankeaan vesiputousmalliin.

Kohdeorganisaatio toimii ketterän ohjelmistokehityksen periaatteiden mukaisesti, joten ohjelmistoarkkitehtuurin järeään etukäteissuunnitteluun suunnittelumenetelmiä hyväksikäyttäen suhtaudutaan varauksellisesti. Historia kohdeorganisaatiossa on kuitenkin paljastanut tarpeen järjestelmälliseen suunnitteluun. Tulevia vaatimuksia ei ole pystytty ennakoimaan, mikä on joh-

tanut turhaan työhön, heikentyneeseen käytettävyyteen ja monimutkaisiin ratkaisuihin. Liiketoiminta-ajatukset ja asiakastarpeet ovat pystytty toteuttamaan, mutta samalla arkkitehtuuri ja ohjelmistokokonaisuus ovat rämettyneet. Kohdeorganisaatiossa on tämän takia tunnistettu tarve ohjelmistoarkkitehtuurin suunnittelulle. Arkkitehtuurisuunnittelun tekemistä tässä vaiheessa puoltaa ohjelmistojen siirtyminen entistä enemmän web-pohjaisiksi. Murros on jo tapahtunut kuluttajasovelluksissa ja teollisuus seuraa vähitellen perässä. Toinen vallitseva trendi on tuotteiden myyminen kiinteähintaisten sopimusten sijaan palveluliiketoiminnan avulla. Nykyisen arkkitehtuurin vaatimasta erillisestä työpöytäkäyttöisestä konfiguraatiosovelluksesta halutaan päästä eroon. Syyt työpöytäohjelmistosta luopumiseen ovat ennen kaikkea käyttökokemuksen parantaminen, mutta myös web-teknologioiden mahdollistama käyttöjärjestelmä- ja paikkariippumattomuus sekä teknologian kehittymisen mahdollistama nopeampi kehitystyö. Aiemmat oletukset siitä, että web-käyttöliittymä on vain tiedon näyttämistä varten, ei mahdollista saumattomasti konfiguroinnin siirtämistä web-käyttöliittymään. On nopeampaa ja järkevämpää suunnitella ohjelmisto-osien roolit uudestaan ja tehdä tarvittava uudelleenkirjoitus ohjelmistolle.

Ohjelmiston tärkeimmäksi tavoitteeksi ja visioksi määriteltiin kustomoitava ja skaalauntuva web-palvelukokonaisuus, jonka avulla järjestelmän käyttäminen ja konfigurointi on käyttäjäystävällisesti mahdollista. Tavoiteltava arkkitehtuuritaso määriteltiin järjestelmän *kokonaisarkkitehtuurin kuvaamiseksi*. Tällöin ohjelmiston tärkeimmät osat, niiden suhteet ja tehtävät kuvataan yleisellä tasolla puuttumatta toteutuksen yksityiskohtiin. Tällaisen arkkitehtuuriluonnoksen avulla kohdeorganisaation on mahdollisuus arvioida tarvittavia resursseja ja muutosvaatimuksia vision saavuttamiseksi. Liian tarkkaa ennalta suunnittelua ei haluttu tehdä, koska tällaiset suunnitelmat vanhentuvat hyvin nopeasti. Ohjelmistoarkkitehtuurin suunnitteluun vaikuttavat tilannetekijät, jotka on esitetty taulukossa 2. Tässä tapaustutkimuksessa tilannetekijöiksi on tunnistettu ohjelmisto, alusta, tavoitteet, rajoitteet, resurssit ja ympäristö.

Taulukko 2: Ohjelmistoarkkitehtuurin suunnittelun tilannetekijät

Tilannetekijä	**Selite**
Ohjelmisto	Tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmä
Alusta	Ohjelmistoratkaisu toteutetaan avoimen lähdekoodin ohjelmointikielillä
Tavoitteet	Järjestelmän kokonaisarkkitehtuurin kuvaaminen
Rajoitteet	Arkkitehtuurin suunnitteluun voidaan käyttää kohdeorganisaation työntekijöiden ja asiakkaiden haastatteluja maksimissaan 2 h/ henkilö. Uuden ratkaisun tulee pohjautua mahdollisimman paljon aiempaan järjestelmään (koodin uudelleenkäyttö). Muut rajoitteet on kuvattu kohdassa 5.2.2.
Resurssit	Arkkitehtuurin suunnitteluun voidaan käyttää kohdeorganisaation työntekijöiden apua haastatteluiden muodossa. Suunnittelun toteuttamisesta vastaa tutkija. Arkkitehtuurin toimeenpanemisesta vastaa yrityksen tuotekehitysyksikkö (3 ohjelmistokehittäjää) ja ulkopuoliset asiantuntijat.
Ympäristö	Ohjelmistokehitystä tehdään Scrum-viitekehystä mukaillen

Ohjelmistoarkkitehtuurin suunnittelua varten päädyttiin käyttämään jotakin olemassa olevaa suunnittelumenetelmää, koska kohdeorganisaatiossa ei ole aiempia toimintatapoja kokonaisvaltaiseen arkkitehtuurisuunnitteluun. Kohdeorganisaatiossa haluttiin edetä suunnitteluprosessissa johdonmukaisesti ja järjestelmällisesti, jotta lopputulos olisi varmasti vaatimukset täyttävä. Suunnittelumenetelmien käytöllä on lukuisia hyötyjä (Hofmeister ym., 2007), joista tärkein on ohjelmiston laadullisten ominaisuuksien täyttäminen.

Kohdeorganisaatio ei asettanut rajoitteita tai toiveita suunnittelumenetelmän valintaan, joten sopivan menetelmän valinta jäi tutkijan tehtäväksi. Suunnittelumenetelmän valintaan toki vaikutti kohdeorganisaation tutkijan työlle asettamat tavoitteet, joista tärkeimpänä oli uuden järjestelmän vaatimusten selvittäminen ja kokonaisarkkitehtuurin kuvaaminen. Arkkitehtuurin suunnitteluun oli käytössä rajoitetusti tutkijan ja kohdeorganisaation henkilökunnan resursseja, joten suunnittelumenetelmän käyttö ei esimerkiksi saanut perustua usean päivän workshop-tilaisuuksiin. Menetelmän tuloksena vaadittiin arkkitehtuuri, joka rajoitteiden puitteissa täyttää luotettavasti järjestelmälle asetetut vaatimukset. Menetelmän toivottiin olevan tarpeeksi kypsä ja sellainen, että sen käyttöön ja soveltamiseen on riittävä ohjeistus, ja että sitä on todistetusti onnistuneesti käytetty muissa ohjelmistoprojekteissa.

Sopivan suunnittelumenetelmän löytämiseksi tutustuttiin useisiin teoksiin ja tutkimuksiin. Tärkeimpänä näistä on aiemmin kuvattu Hofmeisterin ym. (2007) teos viiden teollisuudesta lähtöisin olevan suunnittelumenetelmän vertailusta. Käytännön toteutuksia suunnittelumenetelmien käytöstä tutkittiin mm. Bassin ym. (2003), Cervantesin ym. (2013) sekä Woodin (2007) esityksistä. Lisäksi tutustuttiin ohjelmistoarkkitehtuurin suunnitteluun ketterän ohjelmistokehityksen kontekstissa Abrahamssonnin ym. (2010), Akbarin ja Sharifin (2012), Faberin (2010), Hadarin ja Shermanin (2012), Kruchtenin (2010), Keulerin ym. (2012), sekä Nordin ja Tomaykon (2006), tutkimusten pohjalta.

Kirjallisuuteen tutustumalla päädyttiin vertailemaan kolmea menetelmää, jotka arvioitiin kohdeongelmaan sopiviksi: ominaisuusvetoista arkkitehtuurisuunnittelumenetelmä (Wojcik ym., 2006), RUP 4+1 mallin suunnittelumenetelmää (Kruchten, 1995; Kruchten, 2004) sekä Faberin (2010) mallia ohjelmistoarkkitehtuurin suunnittelusta Scrum-viitekehyksen yhteydessä. RUP 4+1 mallin suunnittelumenetelmää pidettiin kattavana, mutta saadakseen sen käytöstä täyden hyödyn tulisi kohdeorganisaation ohjelmistokehityksen jäljitellä muiltakin osin RUP-prosessikehystä (Kruchten, 2004), jota voidaan pitää raskaana menetelmänä. Faberin (2010) mallia pidettiin mielenkiintoisena sen Scrum-taustan takia, mutta menetelmän kuvaus on hyvin pintapuolinen. Faberin malli ei varsinaisesti ole hyvin kuvattu menetelmä, vaan ainoastaan ajatus siitä, kuinka arkkitehtuuri voitaisiin ottaa huomioon Scrum-viitekehystä käytettäessä. Malli sopii parhaiten käytettäväksi ohjelmiston ylläpitovaiheessa, koska se ei tarjoa työkaluja suunnitteluun. RUP 4+1 mallin suunnittelumenetelmän tai Faberin (2010) mallin käytöstä arkkitehtuurin suunnittelussa ei löydetty tapaustutkimuksia.

Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän käytöstä löydettiin menetelmää tukevia tapaustutkimuksia (mm. Cervantes ym. (2013) ja Wood (2007)) ja muita menetelmän käyttöä havainnoivia dokumentteja. Kirjallisuuteen (mm. Hofmeister ym. (2007)) perehtymällä todettiin ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmän olevan tällä hetkellä vakiintunein suunnittelumenetelmä. Menetelmästä oli saatavissa tarpeeksi dokumentaatiota ja sen esitys vaikutti selkeältä. Menetelmän opetteleminen ja hyödyntäminen eivät vaikuttaneet liian raskaalta. Menetelmän käytöstä oli saatavissa jo valmiiksi tapaustutkimuksia, jotka tukivat ja helpottivat tämän tapaustutkimuksen toteuttamista. Ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä on kohdistettu suuriin ja vaativiin ohjelmistoprojekteihin, mutta sen arvioitiin sopivan myös tämän tutkimuksen toteuttamiseen. Näiden seikkojen takia tähän tutkimukseen suunnittelumenetelmäksi valittiin ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä.

5.2.2 Suunnitteluprosessi

Seuraavassa kerrotaan yleisellä tasolla ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän (Wojcik ym., 2006) soveltamisesta tässä tapaustutkimuksessa. Kuvailusta on jätetty pois sellaisia laadullisia ja toiminnallisia vaatimuksia, rajoitteita sekä arkkitehtuuripäätöksiä, jotka lukeutuvat yrityssalaisuuden piiriin. Tekstin avulla on kuitenkin mahdollista saada käytännön tietoa siitä, kuinka suunnittelumenetelmää sovellettiin.

Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän (*Attribute-Driven Design, ADD*) (Wojcik ym., 2006) lähtökohtana on ohjata arkkitehtuurin suunnittelua siten, että tuloksena saavutettava järjestelmä täyttää vaaditut laadulliset ja toiminnalliset vaatimukset sekä suunnittelulle asetetut rajoitteet. Menetelmän avulla kuvattava järjestelmä käsitellään iteratiivisesti osakomponentti kerrallaan. Työvaiheiden lopputuloksena on arkkitehtuurisuunnitelma, jossa elementeille on määritelty roolit, vastuut, ominaisuudet, sekä yhteydet muihin elementteihin. Tarkemmin menetelmää on kuvattu kohdassa 3.1.2.

Suunnittelun ensimmäisessä vaiheessa *(Varmistu vaatimusten riittävyydestä*) kerättiin suunnitteluun vaikuttavia vaatimuksia kohdeorganisaation henkilöltä ja asiakkaalta. Tässä tiedonkeruussa oleelliseksi muodostui teollisuuden toimintaympäristön rajoitteiden tunnistaminen. Jotkut asiakkaat haluavat kaiken kriittisen tiedon omaan hallintaan, eikä tiedon siirtäminen kokonaan esimerkiksi pilvipalveluihin ole vaihtoehto. Tämä vaatimus on vaikuttanut eniten sekä nykyiseen että uuteen arkkitehtuuriin yhdessä edellä kuvatun vision kanssa. Tutkija koosti vaatimuksista ja rajoitteista listan, jota tuoteomistaja kommentoi ja asetti jokaiselle elementille tärkeyden (korkea - H, keskisuuri - M, pieni - L). Lista tuoteomistajan määräämien prioriteettien kanssa esitetään alla:

1. Web-käyttöliittymä, jonka kautta järjestelmän konfigurointi ja käyttäminen on mahdollista
    a) Reaaliaikainen näkymä avainlukutietoon (vrt. Kuvio 12) (**H)**

   b) Käyttäjän reagointimahdollisuus järjestelmän tuottamiin hälytyksiin sekä hälytyshistorian selailu (**H)**
   c) Avainlukutiedosta koostettujen raporttien katsomismahdollisuus (**M**)
   d) Käyttäjän mahdollisuus konfiguroida järjestelmän perustoiminnallisuutta (mm. hälytyksien vastaanottajat, aktiiviset hälytykset, näkymät, pääsyoikeudet) (**M**)
   e) Käyttäjän mahdollisuus konfiguroida järjestelmään liitettäviä mittauspalvelimia, antureita, analyysifunktioita sekä niiden nimiä ja muita parametreja (**L**)
2. Tietoliikennerajoitteet
   a) Järjestelmäkokonaisuuksien tulee toimia itsenäisesti tietoliikenneyhteyden katketessa (**H**)
   b) Järjestelmän tulee kyetä suorittamaan kriittiset hälytykset myös tilanteessa, jossa tietoliikenneyhteyttä ei ole käytettävissä (**H**)
   c) Tietoliikenneyhteys asiakkaan ympäristöstä internetiin voi olla hidas ja epävakaa (**H**)
3. Järjestelmän luotettavuus ja reaaliaikaisuus
   a) Hälytykset on lähetettävä muutaman sekuntien kuluessa havaitusta hälytystilasta (**H**)
   b) Käyttäjälle näytettävän mittaustiedon täytyy olla reaaliaikaista (**H**)
   c) Käyttäjälle näytettävän mittaustiedon täytyy olla luotettavaa (**H**)
   d) Järjestelmän sisäisistä ongelmista täytyy indikoida käyttäjää ja ylläpitäjää (**L**)
4. Joustava konfigurointi eri asiakastilanteita ja vaatimuksia varten (**M**)
5. Järjestelmän rakentuminen nykyisen ohjelmistoratkaisun varaan (pl. WebUI) (**L**)
6. Järjestelmän skaalautuvuus ja ylläpidettävyys tulee olla hyvällä tasolla (**M**)

Edellä esitetyt koko järjestelmää koskevat vaatimukset koskevat tuotteen MVP-ratkaisua (*Minimum viable product*), joka pitää sisällään liiketoiminnan kannalta oleellisimmat toiminnot ja rajoitteet. Tutkijan työtä helpotti olemassa oleva ohjelmisto, jonka toiminnalliset vaatimukset ovat pääosin myös uuden ohjelmiston vaatimuksia. Näitä vaatimuksia ei kokonaisuudessaan tässä listata, sillä ne eivät ohjaa järjestelmän arkkitehtuurin suunnittelua.

Vaatimukseen 4 (*Joustava konfigurointi eri asiakastilanteita ja vaatimuksia varten)* vaikuttavia asiakastapauksia liittyen tietoliikenneverkkoon tunnistettiin kolmenlaisia:

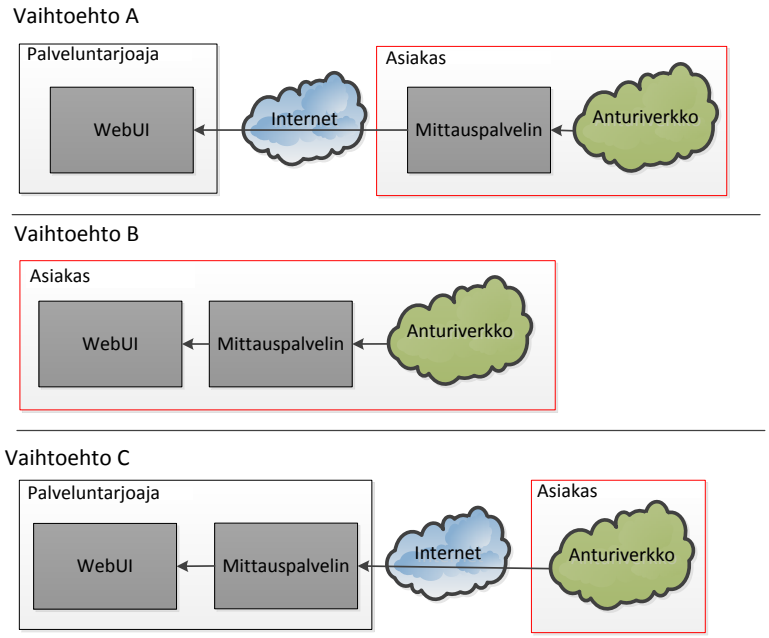
- *Rajoitettu yhteys*: Asiakkaan ympäristöstä on palomuurilla rajoitettu yhteys internetiin. Tällöin yhteyden ottaminen internetistä päin asiakkaan ympäristöön ei ole mahdollista.
- *Rajoittamaton yhteys:* Asiakas konfiguroi tietoliikenneverkkonsa sillä tavoin, että yhteydenotot internetistä päin palveluntarjoajan laitteisiin ovat mahdollisia. Vaihtoehtoisesti palveluntarjoaja voi luoda järjestelmälle oman yhteyden internetiin esimerkiksi 3G-verkon avulla.

- *Ei yhteyttä*: Joissakin tapauksissa asiakkaan tietoliikenneympäristö on eristetty julkisesta internetistä kokonaan. Tällöin ylläpidolliset työt voidaan hoitaa esimerkiksi VPN-yhteyden avulla.

Arkkitehtuurin suunnittelussa tunnistettiin kolme erilaista asiakastapausta (*vaatimus 4*) tiedon säilyttämisen näkökulmasta. Asiakkailla on erilaisia käytäntöjä siihen, missä fyysisessä sijainnissa heidän tuotantoa koskevaa tietoa voidaan säilyttää. Uuden arkkitehtuurin tulee myös tukea samoja asiakastapauksia kuin vanhan arkkitehtuurin. Tiedon säilyttämisen vaihtoehtoja nykyisessä järjestelmässä ovat:

- *Vaihtoehto A - Avainluvut pilvipalvelussa:* Mittauspalvelin on asiakkaan verkossa, mutta avainlukutieto siirretään palvelutarjoajan palvelimille, jossa WebUI sijaitsee. Tällöin WebUI-palveluun on luotettava yhteys kaikkialta internetistä. Raakadata sijaitsee kuitenkin asiakkaan mittauspalvelimella
- *Vaihtoehto B - Tieto asiakkaalla:* Asiakas ei halua siirtää tuotantonsa kannalta oleellista tietoa oman fyysisen sijaintinsa ulkopuolelle. Tällaisissa tapauksissa mittauspalvelin ja WebUI ovat asiakkaan tiloissa. WebUI-käyttöliittymään ei välttämättä haluta mahdollistaa yhteyttä internetistä käsin, vaan se on käytettävissä vain asiakkaan sisäverkosta käsin.
- *Vaihtoehto A – Kaikki tieto pilvipalvelussa:* WebUI ja mittauspalvelin ovat molemmat palveluntarjoajan ylläpidossa. Tällöin asiakkaan ympäristössä sijaitsee ainoastaan anturit, joihin mittauspalvelin ottaa yhteyden salatun yhteyden ylitse. Kaikki anturitieto tallennetaan palveluntarjoajan pilvipalveluun.

Vaihtoehdot mittauspalvelimen ja web-palvelun sijoittamiseen on kuvattu kuviossa alla. Mikäli WebUI sijoitetaan palveluntarjoajan ympäristöön, voidaan samalla fyysisellä laitteella palvella useampaa eri asiakasta.



Kuvio 15 Mittauspalvelimen ja web-palvelimen sijainnit asiakastapauksissa

Ensimmäinen iteraatio

Suunnittelumenetelmän mukaisesti seuraavat vaiheet suoritettiin useammassa iteraatiossa. Ensimmäisen iteraation toisessa työvaiheessa (*Valitse kuvattavat elementit*) valitaan kuvattava elementti. Ensimmäisellä iteraatiokierroksella kuvattava elementti on järjestelmä kokonaisuudessaan, sillä muita vaihtoehtoja ei ole.

Menetelmän kolmannessa vaiheessa (*Tunnista arkkitehtuuriin vaikuttavat vaatimukset*) tunnistetaan ja priorisoidaan arkkitehtuuriin vaikuttavat vaatimukset ja rajoitteet. Tuoteomistaja on aikaisemmin priorisoinut vaatimukset ja rajoitteet, joten tässä vaiheessa tutkija priorisoi vaatimukset sen mukaan, millainen niiden odotettava vaikutus arkkitehtuuriin on. Taulukko 3 esittää yhteenvedon tuoteomistajan ja tutkijan tekemistä prioriteeteista. Koska alakohtien välinen prioriteettivaihtelu on pientä, päätettiin yksinkertaisuuden vuoksi tässä kohtaa vaatimuksia ja rajoitteita käsitellä kokonaisuuksina ilman edellä mainittuja alakohtia. Näin ollen tässä vaiheessa arkkitehtuuria ohjaaviksi vaatimuksiksi valittiin kaikki viisi yläkategoriaa, eli 1) web-käyttöliittymä, jonka kautta järjestelmän konfigurointi ja käyttäminen on mahdollista, 2) tietoliikennerajoitteet, 3) järjestelmän luotettavuus ja reaaliaikaisuus, 4) joustava konfigurointi eri asiakastilanteita ja vaatimuksia varten ja 5) järjestelmän rakentuminen nykyisen ohjelmistoratkaisun varaan sekä 6) skaalautuvuus ja ylläpidettävyys. Myös Wojcik ym. (2006) suosittavat vaatimusten ja rajoitteiden karsimista 5-7 tärkeimpään.

Taulukko 3: Vaatimusten ja rajoitteiden priorisointi ominaisuusvetoisen arkkitehtuurisuunnittelun kolmannessa vaiheessa

Vaatimus / rajoite	**Liiketoiminnallinen arvo**	**Arkkitehtuurillinen merkitys**
1a (tilannenäkymä)	H	L
1b (hälytykset)	H	L
1c (raportit)	M	L
1d (peruskonfigurointi)	M	L
1e (järjestelmäkonfigurointi)	L	M
2a (itsenäinen toiminta)	H	H
2b (kriittiset hälytykset)	H	H
2c (hidas yhteys)	H	H
3a (hälytyksiin reagointiaika)	H	M
3b (mittaustiedon reaaliaikaisuus)	H	M
3c (mittaustiedon luotettavuus)	H	M
3d (käyttökatkojen indikonti)	L	M
4 (asiakastilanteiden tukeminen)	M	H
5 (pohjautuminen nykyiseen ohjelmistoon)	L	H
6 (skaalautuvuus ja ylläpidettävyys)	M	H

Neljännessä vaiheessa (V*alitse suunnitteluratkaisut*) tarkoituksena on määrittää suunnittelupäätökset (suunnittelumenetelmät, arkkitehtuurityylit ja taktiikat), jotka täyttävät arkkitehtuurillisesti merkittävät vaatimukset. Cervantes ym. (2013) ovat tässä vaiheessa konkretisoineet arkkitehtuurisuunnitelmaa määrittelemällä ohjelmistokehykset (esim: Java Spring), joita käyttämällä ratkaistaan suunnitteluongelmat. Koska tässä tapauksessa tarkoituksena on kuvata järjestelmää korkealla tasolla, ei tässä kohdassa määritelty käytettäviä ohjelmistokehyksiä. Vaatimuksista tunnistettiin kaksi oleellista elementtiä tämän vaiheen kannalta. Järjestelmältä halutaan web-käyttöliittymä, johon voidaan ottaa yhteys internetistä. Lisäksi tarvitaan järjestelmä, joka huolehtii mittaamisesta antureilta sekä hälytyksien teosta ilman jatkuvatoimista tietoliikenneyhteyttä. Tästä syystä luotiin kaksi komponenttia: web-palvelin ja mittauspalvelin, jotka toimivat toisistaan riippumatta tietoliikennekatkojen tapahtuessa. Suunnittelun lähtökohdaksi tietoliikenteen näkökulmasta otettiin asiakas-palvelin arkkitehtuurityyli, sillä kaikissa asiakastapauksissa mittauspalvelimella on mahdollisuus ottaa yhteys web-palvelimeen päin. Toinen vaihtoehtoinen arkkitehtuurityyli näiden järjestelmien välillä olisi voinut olla viestinvälitysarkkitehtuurityyli. Tällainen ratkaisu skaalautuisi nykyistä paremmin useamman mittauspalvelimen ja web-palvelimen ympäristöihin, mutta se olisi luonut järjestelmästä monimutkaisemman. Useissa tapauksissa yhdellä asiakkaalla on kuitenkin vain yksi web-palvelin ja yksi mittauspalvelin.

Menetelmän viidennessä vaiheessa (*Toteuta arkkitehtuurielementit ja jaa vastuut*) tällä iteraatiokierroksella määriteltiin web-palvelimen ja mittauspalvelimien väliset vastuut. Koska käyttäjällä täytyy olla pääsy avainluku- ja asetustietoon tilanteessa, jossa yhteys mittauspalvelimeen on katkennut (*vaatimus 2*), päätettiin web-palvelimen huolehtivan avainlukujen ja asetustiedon tallentamisesta ja ylläpidosta. Mittauspalvelimen vastuuksi jätettiin mittaustiedon hakeminen antureilta sekä mittauksista koostetun avainlukutiedon tallentaminen väliaikaiseen tietokantaan, josta ne siirretään web-palvelimen tietokantaan. Web-palvelimen tehtävä on tuottaa käyttäjälle raportit ja hälytykset. Tietoliikenteen näkökulmasta web-palvelin odottaa mittauspalvelimen yhteydenottoja, eli se toimii palvelimena. Mittauspalvelimen tehtävä on taas muodostaa asiakkaan roolissa yhteys web-palvelimelle. Koska mittauspalvelimen rooli on nykyisenkaltainen muutamia poikkeuksia (hälytykset, raportit) lukuun ottamatta, tulee myös vaatimus 5 (*Järjestelmän rakentuminen nykyisen ohjelmistoratkaisun varaan*) täytetyksi. Uuden mittauspalvelimen toteutuksessa voidaan siis käyttää suurilta osin vanhan arkkitehtuurin mittauspalvelimen ohjelmakoodia hyväksi. Web-palvelimen rooli tulee muuttumaan nykyisestä WebUI:sta niin paljon, että se on käytännössä alusta alkaen uudelleenkirjoitettava ottaen huomioon uudet vaatimukset.

Kuudennessa vaiheessa (*Määritä rajapinnat arkkitehtuurielementeille)* määriteltiin rajapintoja. Ensimmäisellä iteraatiokierroksella määriteltiin mittauspalvelimen ja web-palvelimen välinen rajapinta. Rajapinnan tärkeimmät tehtävät ovat käskyjen vieminen web-palvelimelta mittauspalvelimelle ja avainlukutiedon kuljettaminen web-palvelimelle. Rajapinnan tulee mahdollistaa yhteyskat-

kojen havaitseminen, jotta käyttäjää voidaan varoittaa asianmukaisesti (*vaatimus 3d*). Rajapinnan tulee mukautua erilaisiin asiakastilanteisiin (*vaatimus 4*), kuten mittauspalvelimen sijaintiin palomuurin takana. Rajapintaan oleellisesti vaikuttavaksi tekijäksi havaittiin myös vaatimukset 3 (t*ietoliikennerajoitteet*) ja 4 (*järjestelmän luotettavuus ja reaaliaikaisuus*). Rajapinnan tulisi siis palautua ongelmitta yhteyskatkoista ja toimia mahdollisimman reaaliaikaisesti. Yhteyskatkojoen sattuessa avainluvut täytyy kuitenkin siirtää mittauspalvelimelle tallennettavaksi, vaikka ne eivät enää olisikaan reaaliaikaisia.

Ensimmäisen iteraatiokierroksen viimeisessä (*Varmista ja jalosta vaatimukset*) vaiheessa varmistuttiin siitä, että laadulliset ja toiminnalliset vaatimukset ja rajoitteet kuljetetaan eteenpäin iteraatiokierroksella luotuihin elementteihin. Ensimmäisellä iteraatiokierroksella web-palvelimen vastuulle kuljetettiin huolehtiminen järjestelmän konfiguraation säilyttämisestä ja tiedon näyttämisestä (*vaatimus 1*). Rajapinnan vastuulle vietiin sitä koskettavat vaatimukset 2 (t*ietoliikennerajoitteet*) ja 3 (*järjestelmän luotettavuus ja reaaliaikaisuus*). Vaatimuksesta 1 johdettiin lisäksi uusi vaatimus mittaustiedon tallentamisesta ja kuljettamisesta web-palvelimelle, siten että luotettavuus ja reaaliaikaisuus tulevat huomioiduiksi (*vaatimukset 3a, 3b, 3c*). Nämä vaatimukset määrättiin mittauspalvelimen vastuulle. Vaatimus 6 (*skaalautuvuus ja ylläpidettävyys*) tunnistettiin sellaiseksi, joka vaikutta kaikkiin tässä vaiheessa luotuihin arkkitehtuurikomponentteihin.

Ensimmäisellä iteraatiokierroksella luotiin siis kolme elementtiä, web-palvelin, mittauspalvelin ja niiden välinen rajapinta. Web-palvelimen tehtävänä on avainlukutiedon näyttäminen sekä raporttien ja hälytyksien generointi. Mittauspalvelimen tehtävänä on mitata anturilta tietoa, koostaa siitä avainlukutietoa ja kuljettaa se rajapinnan kautta web-palvelimelle.

Toinen iteraatio

Koska mittauspalvelin perustuu pitkälti vanhan järjestelmän varaan, jää tässä vaiheessa ainoaksi kuvattavaksi järjestelmäksi web-palvelin. Rajapintaa web-palvelun ja mittauspalvelimen välillä voidaan ajatella myös järjestelmäosana, mutta sitä ei tässä yhteydessä tarkemmin kuvata. Rajapinnan käytännön toteuttamista ohjaavat hyvin paljon käytettävät ohjelmistokehykset, joihin ei tässä tutkimuksessa haluttu ottaa kantaa. Toinen iteraatiokierros aloitettiin siis valitsemalla kuvattavaksi järjestelmäksi web-palvelin. Mittauspalvelinta ei tässä kuvata tarkemmin, sillä sen toiminnallisuutta tullaan muokkaamaan vain siltä osin, että se mahdollistaa avainlukujen ja muun informaation siirtämisen web-palveluun sopivalla yhteystavalla. Uusia arkkitehtuurillisia ratkaisuja ei siis tulla tekemään mittauspalvelimen osalta.

Aiemmassa iteraatiossa web-palvelimen vaatimukseksi määrättiin järjestelmän konfiguraation säilyttäminen ja tiedon näyttäminen (*vaatimus 1*). Web-palvelimen täytyy toimia itsenäisesti tilanteessa, missä tietoliikenneyhteys mittauspalvelimeen on katkennut (*vaatimukset 2a & 2c*). Itsenäinen toiminta tarkoittaa käytännössä sitä, että käyttäjä saa yhteyden web-palvelimeen ja saa tiedon katkenneesta yhteydestä mittauspalvelimeen. Tästä huolimatta käyttäjän täytyy pystyä katselemaan järjestelmään aiemmin saapunutta tietoa. Web-palvelimen sisäisessä toteutuksessa tulee ottaa myös huomioon skaalautuvuus ja ylläpidet-

tävyys. Web-palvelimen osalta tämä tarkoittaa mahdollisimman selkeiden sisäisten rajapintojen käyttämistä, jolloin järjestelmät rajapintojen päissä ovat helposti vaihdettavissa toisiin.

Web-palvelu jaettiin skaalautuvuuden nimissä kahteen osaan. Skaalautuvuuden takia (*vaatimus 6*) web-palvelusta haluttiin erottaa asiakkaalle näkyvä palvelu sekä järjestelmän sisäinen toiminnallisuus. Arkkitehtuurityylilliseksi lähestymistavaksi valittiin kerrosarkkitehtuuri, jossa alimpana kerroksena on tietokanta- ja rajapintayhteydestä huolehtiva kerros ja ylempänä tiedon esityskerros. Asia voidaan mieltää myös asiakas-palvelin arkkitehtuuriksi, sillä ylempi kerros pyytää alemman kerroksen palveluita. Enemmän kyse on kuitenkin kerrosarkkitehtuurista, sillä siinä kerrosten välillä on selkeämpi looginen riippuvuussuhde.

Viidennessä työvaiheessa kerrosarkkitehtuurin osat nimettiin ytimeksi ja web-käyttöliittymäksi. Web-käyttöliittymän tehtävänä on tarjota käyttäjälle käyttäjäystävällinen käyttöliittymä, joka täyttää vaatimuksen 1 (*Web-käyttöliittymä, jonka kautta järjestelmän konfigurointi ja käyttäminen on mahdollista*). Web-palvelimen ytimen tehtävänä on huolehtia tietokantayhteydestä ja kommunikoinnista rajapinnan kautta mittauspalvelimen kanssa. Se huolehtii hälytyksien ja raporttien generoinnista. Web-käyttöliittymä käyttää alemman kerroksen palveluja saadakseen yhteyden tietokantaan.

Kuudennessa vaiheessa rajapintojen näkökulmasta päädyttiin käyttämään REST-arkkitehtuuria (*Representational State Transfer)* (Fielding, 2000), joka on web-palveluissa yleisesti käytetty tapa erottaa tiedon esityskerros ja muu sovelluslogiikka. REST-arkkitehtuuri antaa suuntaviivoja ja rajoitteita, joita noudattamalla käyttäjälle voidaan tarjota skaalautuva ja tilaton rajapinta. Moni ohjelmistokirjasto tarjoaa valmiiksi työkaluja REST-arkkitehtuurin mukaisen rajapinnan tarjoamiseen tai käyttämiseen. Käytännössä lähes kaikki olemassa olevat REST-toteutukset käyttävät HTTP-protokollaa ja sen metodeita (GET, PUT, POST, DELETE) tiedonsiirrossa. Tässä tapauksessa REST-arkkitehtuuri erottaa web-palvelimen käyttöliittymän ja ytimen toisistaan. Tämä mahdollistaa esimerkiksi tietokantajärjestelmän vaihtamisen toiseen, jos se on skaalautuvuuden ja ylläpidettävyyden (*vaatimus 6*) näkökulmasta perusteltua.

Toisen iteraatiokierroksen viimeisessä vaiheessa vaatimus 6 (*skaalautuvuus & ylläpidettävyys*) kuljetettiin sekä Web-palvelimen ytimelle, että käyttöliittymälle. Molemmat elementit myös toteuttavat omalta osaltaan vaatimusta 1 käyttöliittymän tarjoamisesta. Vaatimus 1 pilkottiin siten, että web-käyttöliittymän tehtävä on tarjota näkymä ja konfiguraatiomahdollisuus tietoon, kun taas web-palvelimen ytimen tehtävä on ylläpitää ja säilyttää mittauspalvelimelta tullutta tietoa sekä järjestelmän konfigurointitietoa. Tietoliikenneyhteyden katkeamisen web-käyttöliittymän ja web-palvelimen ytimen välillä todettiin olevan epätodennäköinen tilanne, sillä osat toimivat yleensä samalla fyysisellä palvelimella. Tästä syystä vaatimus 2A (*Järjestelmäkokonaisuuksien tulee toimia itsenäisesti tietoliikenteen katketessa*) jätettiin huomioitta. Kerrosarkkitehtuuriajattelun mukaisesti ylempi kerros ei voi toimia ilman alempaa kerrosta. Toki itsenäinen web-käyttöliittymä voi kertoa käyttäjälle yhteyskatkosta, mutta sillä

ei ole mahdollisuutta puskuroida tai säilyttää tietoa pitkän tietoliikennekatkon varalta.

Kolmas iteraatio

Kolmas iteraatio aloitettiin valitsemalla seuraava kuvattava järjestelmä. Vaihtoehdot olivat web-palvelimen ydin, web-palvelimen käyttöliittymä sekä mittauspalvelimen ja web-palvelimen välinen rajapinta. Koska web-palvelimen rajapinnan kuvaaminen oli aiemmin suljettu pois, jäi jäljelle kaksi vaihtoehtoa. Näistä vaihtoehdoista päädyttiin kuvaamaan web-palvelimen käyttöliittymä, sillä sen arkkitehtuurilliset vaatimukset olivat jo hyvin selvillä. Web-palvelimen ytimen kuvaamiseen ei katsottu olevan aihetta, sillä se rakennetaan jonkin ohjelmistokehyksen päälle, mikä käytännössä määrää käytettävät arkkitehtuurityylit. Teknisiin yksityiskohtiin ei tässä työssä haluttu mennä.

Web-käyttöliittymän toiminnallisena vaatimuksena on tarjota näkymä ja konfigurointimahdollisuus järjestelmän sisältämään tietoon (*vaatimus 1*). Tämän lisäksi toteutettavan käyttöliittymän tulee olla skaalautuva ja helposti ylläpidettävä (*vaatimus 6).* Vaatimus 4 (*Joustava konfigurointi eri asiakastilanteita ja vaatimuksia varten)* vaikuttaa web-käyttöliittymään sillä tavoin, että käyttöliittymä täytyy olla konfiguroitavissa ja modifioitavissa eri käyttäjiä varten. Käytännössä tämä tarkoittaa esimerkiksi asiakkaan logon näyttämistä käyttöliittymässä ja järjestelmästä muodostetuissa raporteissa.

Kolmannessa työvaiheessa vaatimusten perusteella valittiin arkkitehtuurityyliksi MV*-arkkitehtuuri. Muita arkkitehtuurityylejä ei edes harkittu, sillä käytännössä kaikki nykyiset web-kehitykseen tarkoitetut ohjelmistokehykset pohjautuvat MV*-arkkitehtuuriin. MV*-arkkitehtuurin avulla varmistetaan erilaiset käyttöliittymät eri asiakkaille sekä erilaiset tiedon esittämistavat eri käyttäjäryhmille. Valittu arkkitehtuurityyli sallii käyttöliittymän erottamisen varsinaisesta sovelluslogiikasta.

MV*-arkkitehtuurityyli ja käytettävä ohjelmistokehys määräävät tarkemmin arkkitehtuurilliset komponentit, joten neljännessä vaiheessa komponentteja ei lopullisesti määritetty. Suurin osa ohjelmistokehyksistä sisältää ainakin mallin ja näkymän, mutta muut käytettävät komponentit riippuvat ohjelmistokehittäjän tekemistä ratkaisuista. Tässä vaiheessa muita komponentteja ei haluttu määritellä, ettei arkkitehtuuri ohjaa liiaksi käytännön tekemistä. Todennäköistä on, että valmiissa ohjelmistossa web-käyttöliittymä koostuu neljästä komponentista. Palvelukomponentti huolehtii REST-rajapinnan toteuttamisesta, malli tarjoaa käyttöliittymälogiikkaa, näkymä huolehtii käyttöliittymäelementeistä ja ohjain toimii rajapintana mallin ja näkymän välillä.

Kolmannen iteraation kuudennessa vaiheessa web-käyttöliittymän sisäisiä rajapintoja ei lähdetty tarkemmin määrittelemään. Rajapintojen määrittäminen jätettiin ohjelmistokehittäjän tehtäväksi, sillä kyse on yhden järjestelmäkokonaisuuden hyvin tarkasta sisäisestä toteutuksesta.

Viimeisessä vaiheessa vaatimuksia ei myöskään tarkennettu luoduille elementeille. Käytännössä web-käyttöliittymän sisäisillä komponenteilla on samat vaatimukset kuin isäntäkomponentilla.

Yhteenveto iteraatiokierroksista on esitetty taulukossa 4. Sarakkeina ovat suunnittelua ohjanneet vaatimukset (1-6), kuvattava elementti, suunnittelupäätös sekä luodut elementit. Suluissa on esitetty, mihin ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän työvaiheeseen kyseinen taulukon sarake liittyy. Riveinä ovat toteutetut iteraatiot (I-III). Tässä työssä keskityttiin koko järjestelmän ja web-palvelimen toiminnan kuvaamiseen. Ongelmana nykyisessä arkkitehtuurissa on web-käyttöliittymän ja mittauspalvelimen roolijako sekä web-palvelimen tehtävät, mistä syystä esimerkiksi mittauspalvelimen ja anturin rajapintaa tai tehtäviä ei tässä käsitellä. Arkkitehtuurissa ei mittauspalvelinta lähdetty pilkkomaan pienempiin palasiin, sillä nykyinen järjestelmä skaalautuu uuteen käyttötarkoitukseen melko pienellä vaivalla. Iteraation seuraavat vaiheet voisivat koskea rajapinnan sekä mittauspalvelimen sisäisen toteutuksen kuvaamista, mikäli niiden sisäistä toiminnallisuutta halutaan kuvata ja arvioida.

Taulukko 4: Suunnitteluprosessin iteraatiokierrokset

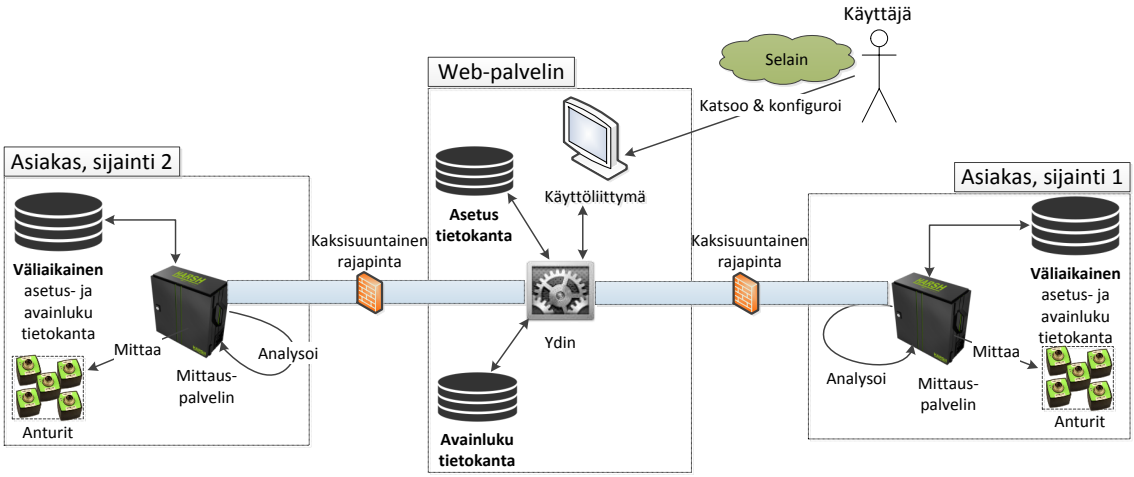
Iteraatio	**Vaatimus (1,3,7)**	**Kuvattava elementti (2)**	**Suunnittelupäätös (4)**	**Luodut elementit (5)**
I.	1, 2, 3, 4, 5, 6	Koko järjestelmä	Asiakas-palvelin-arkkitehtuuri	Mittauspalvelin, web-palvelin, rajapinta
II.	1, 2a, 2c, 4 6	Web-palvelin	REST, kerrosarkkitehtuuri	Ydin, käyttöliittymä
III.	1, 4, 6	Web-käyttöliittymä	MV*	Vähintään malli ja näkymä

5.3 Uusi ohjelmistoarkkitehtuuri

Uuden ohjelmistoarkkitehtuurin päätavoitteena on ollut mahdollistaa järjestelmän käyttäminen ja konfigurointi yhden skaalauntuvan ja käyttäjäystävällisen web-palvelun kautta. Suunnitteluvaiheessa konfiguroinnista tunnistettiin kaksi tasoa: järjestelmän peruskonfiguroitavuus ja järjestelmätason konfigurointi. Loppukäyttäjä tekee järjestelmän peruskonfigurointia, kuten määrää hälytyksien vastaanottajia sekä kustomoi näkymät halutunlaisiksi. Palveluntarjoaja huolehtii järjestelmäkonfiguroinnista, joka sisältää esimerkiksi anturien ja analyysfunktioiden toiminnan määrittelyn. Järjestelmän ensimmäisessä toteutusvaiheessa järjestelmäkonfiguraatio päätettiin jättää pois web-käyttöliittymästä, sillä se voidaan tehdä erillisen konfigurointityökalun avulla, jota loppuasiakkaan ei tarvitse käyttää. Tulevaisuuden tavoite on kuitenkin, että kaikki konfigurointi voitaisiin tehdä web-käyttöliittymästä.

Rajoitteet ja toiminnalliset vaatimukset huomioiden rakennettu uusi arkkitehtuuri on esitetty kuviossa 16. Arkkitehtuuri on esitys asiakastapauksesta,

jossa mittauspalvelin sijaitsee asiakkaan tietoliikenneverkossa ja web-palvelin palveluntarjoajan tietoliikenneverkossa. Esitetty arkkitehtuuri mahdollistaa useammasta mittauspalvelimesta haetun tiedon koostamisen keskitetysti yhteen paikkaan. Tällöin asiakas saa yhdestä web-palvelusta koko yrityksensä tuotantoa koskevan tilannetiedon.



Kuvio 16: Ohjelmiston uusi arkkitehtuuri

Järjestelmän roolit ja toimintaperiaate uudessa arkkitehtuurissa on määritelty seuraavasti:

- *Mittauspalvelin*: Mittauspalvelimen roolia on kavennettu aiemmasta arkkitehtuurista siten, että se on vastuussa ainoastaan raakadatan hakemisesta anturilta sekä sen muuttamisesta avainlukutiedoksi käyttäjän konfiguraation perusteella. Tämä mahdollistaa tärkeimmän liiketoimintalogiikan siirtämisen palveluntarjoajan helpommin saavutettavaan ja ylläpidettävään web-palveluun. Kriittiset hälytykset voidaan kuitenkin edelleen lähettää suoraan mittauspalvelimelta SMS-liittymän avulla. Mittauspalvelin hakee konfiguraation web-palvelimelta. Mittauspalvelimella on paikallinen konfiguraatiotietokanta, jotta se voi toimia itsenäisesti verkkoyhteyden katketessa ulkomaailmaan. Mittauspalvelin tallentaa avainlukuja paikalliseen tietokantaan, josta se siirretään rajapinnan kautta tasaisin väliajoin web-palvelimelle. Verkkoyhteyden katketessa avainlukutiedot siirretään web-palvelimelle, kun yhteys palautuu. Raakadataa voidaan siirtää käyttäjän pyynnöstä web-palvelimelle, mutta sitä ei tehdä automaattisesti siitä aiheutuvan suuren tietoliikennemäärän takia.
- *Web-palvelin*: Web-palvelimen roolina on ylläpitää avainlukutietokantaa, jota voidaan koostaa asiakkaalle useasta eri lähteestä. Web-palvelin huolehtii hälytyksien ja raporttien generoinnista käyttäjän konfiguraation perusteella. Web-palvelin tarjoaa käyttöliittymän konfiguraation muokkaamiseen ja asiakasta kiinnostavan tiedon näyttämiseen. Rajapinta web-palvelimen ytimen ja käyttöliittymän (WebUI 2.0) välillä toteutetaan REST-tekniikan avulla.

Arkkitehtuurityylillisesti järjestelmää ei kuvaa mikään yksittäinen arkkitehtuuri. Esitetty arkkitehtuuri on yhdistelmä kerrosarkkitehtuuria, REST-arkkitehtuuria, asiakas-palvelin arkkitehtuuria, viestinvälitysarkkitehtuuria, tietovuoarkkitehtuuria ja MV*-arkkitehtuuria (Taulukko 5).

Taulukko 5: Uuden arkkitehtuurin arkkitehtuurityylit

Osapuolet	**Arkkitehtuurityyli**
Anturi (palvelin) – mittauspalvelin (asiakas)	Asiakas-palvelin arkkitehtuuri, tietovuoarkkitehtuuri
Mittauspalvelin (asiakas) – Web-palvelimen ydin (palvelin)	Asiakas-palvelin arkkitehtuuri, tietovuoarkkitehtuuri
Web-palvelimen ydin	Viestinvälitysarkkitehtuuri
Web-palvelimen ydin ja käyttöliittymä	Kerroarkkitehtuuri ja REST-arkkitehtuuri
Web-käyttöliittymä	MV*-arkkitehtuuri
Selain (asiakas) – Web-käyttöliittyä (palvelin)	Asiakas-palvelin arkkitehtuuri

Tietoliikenteen näkökulmasta anturi ja mittauspalvelin toimivat asiakas-palvelin arkkitehtuurin mukaisesti. Anturi odottaa mittauspalvelimen käskyä mitata tietoa ja toimittaa sen mittauspalvelimelle. Asiakas-palvelin arkkitehtuurityylin haitoista *tehottomuus* ei toteudu, sillä komponenttien välissä on aina joka tapauksessa tietoliikenneyhteys. Sisäisiä etäproseduurikutsuja ei siis suoriteta. Anturin tarjoama rajapinta mittauspalvelimelle on jo nyt hyvin määritelty, joten asiakas-palvelin-arkkitehtuurin haitoista *rajapintojen muuttumista* ei ole odotettavissa. Mittaustoimenpidettä voidaan kuvata yksinkertaistetusti myös tietovuoarkkitehtuurin näkökulmasta, jossa ensimmäinen komponentti on anturi, joka välittää sen väylän (tietoliikenneyhteys) kautta mittauspalvelimelle. Seuraava komponentti tästä eteenpäin olisi web-palvelin, johon tieto kuljetetaan tietoliikenneyhteyden (väylä) välityksellä. Kyse ei ole kuitenkaan näin selkeistä ja suoraviivaisista operaatioista, minkä takia asiakas-palvelin arkkitehtuurityyli selittää paremmin anturin ja mittauspalvelimen välistä yhteyttä.

Mittauspalvelimen ja web-palvelimen välillä vallitsee tietoliikenteen näkökulmasta asiakas-palvelin arkkitehtuuri. Web-palvelin odottaa mittauspalvelimen yhteydenottoa, jonka jälkeen muodostetaan rajapintayhteys. Rajapintayhteydessä molemmat komponentit voivat viestiä toisilleen viestinjonon periaatteella. Asiakas-palvelin arkkitehtuurin näkökulmasta molemmilla komponenteilla on selkeä työnjako ja tehtävä, jota ne toteuttavat lähes itsenäisesti. Tiedonsiirto rajapinnan kautta on välttämätöntä, jotta käyttäjälle voidaan tarjota reaaliaikaista avainlukutietoa. Web-palvelimen ytimen viestien vastaanottajaa voi ajatella myös viestinvälitysarkkitehtuurin näkökulmasta, sillä se saa mittauspalvelimelta erilaisia viestejä, joille täytyy päättää oikea taho. Tällä ajattelumallilla ydin skaalautuu hyvin erilaisiin tarpeisiin sen mukaan, millaista tietoa mittauspalvelin välittää.

Web-palvelimen ytimen ja käyttöliittymän välillä vallitsee kerrosarkkitehtuuri. Käyttöliittymä on riippuvainen ytimen palveluista, joten kyse on kahdesta kerroksesta koostuvasta kerrosarkkitehtuurista, jossa ohituksia ei tapahdu. Käyttöliittymästä ei siis ole suoraa yhteyttä esimerkiksi tietokantaan, vaan ydin

toimii kerroksena välissä. REST-arkkitehtuurityyliä käytetään tiedonsiirrossa kerrosten välillä. Kerrosarkkitehtuurityylin hyödyistä konkretisoituvat kerrosten *uudelleenkäytettävyys* ja *paikalliset riippuvuudet*. Ytimen tarjoamaa REST-rajapintaa voidaan käyttää muissakin yhteyksissä kuin käyttöliittymässä. Rajapinnan ansiosta web-käyttöliittymä ei muodosta suoraa yhteyttä tietokantaan. Tämän takia tietokanta voidaan vaihtaa kokonaan toiseen ilman muutoksia web-käyttöliittymän toimintaan. Pelkästään käyttöliittymän tehokkuutta ajatellen ratkaisu, jossa web-palvelin on suorassa yhteydessä tietokantaan, voisi olla nopeampi. Skaalautuvuuden takia on kuitenkin parempi luoda välikerros hyödyntäen tasojen välissä REST-arkkitehtuurityyliä.

Web-käyttöliittymän sisäistä arkkitehtuuria kuvaa MV*-arkkitehtuurityyli, jota käytetään yleisesti web-käyttöliittymien rakentamisessa. Web-käyttöliittymän toteuttaminen MV*-arkkitehtuurilla mahdollistaa erilaisten näkymien tarjoamisen samaan tietoon. Tämä on olennaista, sillä esimerkiksi tehtaanjohtaja on kiinnostunut erilaisesta tiedosta ja esitysmuodosta kuin tuotantotyöntekijä. MV*-arkkitehtuurityyli mahdollistaa myös käyttäjäkokemuksen kustomoinnin eri asiakkaille.

Selaimen ja web-käyttöliittymän välillä on tietoliikenteen näkökulmasta kyse asiakas-palvelin arkkitehtuurista. Asiakas-palvelin arkkitehtuurin haitoista huomioonotettava on *rajapintojen muuttuminen*. Asiakkaan selaimesta riippuen web-käyttöliittymän sisältö saatetaan selaimessa tulkita eri tavoilla. Selaimien uudet versiot saattavat tulkita rakennetun ohjelmistokoodin eri tavalla kuin selaimen vanhemmat versiot. Tähän voidaan arkkitehtuurillisesti varautua jakamalla web-käyttöliittymä MV*-arkkitehtuurityylin mukaisesti näkymiin ja muuhun logiikkaan.

5.4 Ohjelmistoarkkitehtuurin arviointi ja vertailu aiempaan

Tässä alaluvussa kuvataan arkkitehtuurin arviointia tässä tapaustutkimuksessa sekä vertaillaan uutta ja vanhaa arkkitehtuuria ajonaikaisten ja ei-ajonaikaisten ominaisuuksien perusteella. Vertailua esitetään Reekien ym. (2006) laadullisten ominaisuuksien luokittelun perusteella. Laadulliset ominaisuudet on luokiteltu ajonaikaisiin (suorituskyky, käytettävyys, luotettavuus ja turvallisuus) ja ei-ajonaikaisiin (ylläpidettävyys, testattavuus, uudelleenkäytettävyys, konfiguroitavuus ja laajennettavuus) ominaisuuksiin.

5.4.1 Arkkitehtuurin arviointi

Luvussa 3 on esitelty sovellusarkkitehtuurin arviointimenetelmiä. Tässä tapaustutkimuksessa työstetyn arkkitehtuurin arvioinnille asetettiin tavoitteeksi toimia arkkitehtuurin laadunvarmistajana. Käytännössä arviointiprosessin tavoitteena on varmistua, että suunniteltu arkkitehtuuri sopii käyttökohteeseensa. Rajoitteena ovat kohdeorganisaation työntekijöiden resurssit, joita voidaan

käyttää vain haastatteluiden muodossa. Valittavan arviointimenetelmän toivottiin olevan vakiintunut, nopeasti omaksuttavissa ja sopivan pienen mittakaavan ohjelmistokehitysprojekteihin.

Sopivaa arviointimenetelmää etsittiin tutustumalla Babarin ym. (2004) teokseen, jossa vertaillaan yhdeksää eri menetelmää. Vertailun mukaan ainoastaan SAAM (Kazman ym., 1994), ATAM (Kazman ym., 1998) ja ALMA –menetelmät (Bengtsson ym., 2004) ovat saavuttaneet sopivan kypsyystason (*Refinement maturity stage*), jossa menetelmän luotettavuutta on koeteltu useissa eri tapauksissa. Näistä ainoastaan ATAM-menetelmä on selkeästi tässä kypsyysvaiheessa. ALMA-menetelmän tarkoituksena on ainoastaan varmistaa järjestelmän ylläpidettävyys, joten se ei sovellu tähän tapaukseen. Tässä tapauksessa halutaan varmistua kaikkien laadullisten ominaisuuksien toteutumisesta. Jäljellejääneistä ATAM ja SAAM –menetelmät ovat hyvin samankaltaisia, sillä molemmat on kehitetty SEI-instituutissa. Arviointimenetelmäksi päätettiin kuitenkin valita ATAM-menetelmä sillä perusteella, että se on saavuttanut hieman vakiintuneemman aseman kuin SAAM-menetelmä. Se ottaa huomioon kaikkien laadullisten ominaisuuksien arvioinnin, kun taas SAAM keskittyy enemmän ohjelmiston muokattavuuden varmistamiseen. ATAM-menetelmä on kehitetty SAAM-menetelmän pohjalta, joten sen uskotaan korjanneen mahdolliset aiemman menetelmän ongelmat.

Uuden arkkitehtuurin laatu vahvistettiin ATAM-arviointimenetelmää soveltamalla. Menetelmän avulla arvioidaan ohjelmistoarkkitehtuurissa tehtyjä ratkaisuja laadullisten ominaisuuksien perusteella. Tämän avulla saadaan tietoa siitä, täyttääkö arkkitehtuuri sille asetetut tavoitteet konkretisoituessaan. Arviointi ATAM-menetelmän avulla perustuu esimerkkitilanteiden (skenaariot) luomiseen, joita vasten arkkitehtuuria testataan. Menetelmä pitää sisällään neljä vaihetta (esittelyosio, analyysiosio, testausosio, raportointiosio), joiden läpivientiin sanotaan menevän noin kolme päivää menetelmän käyttöön osallistuvilta. Tarkemmin ATAM-menetelmää on kuvattu luvussa 3.2.

Tässä tapauksessa arkkitehtuurin arvioimiseksi ei järjestetty erillistä workshop-tilaisuutta, jota menetelmä suosittaa. Arkkitehtuurin suunnittelu- ja arviointi tapahtuivat tapaustutkimuksessa osittain yhtä aikaa. Menetelmän suosituksista käytettiin arkkitehtuurin arviointia skenaarioita vasten, joita kerättiin yksityishaastatteluilla kohdeorganisaation johdolta ja ohjelmistokehitykseen osallistuvilta henkilöiltä. Tärkeimpinä skenaarioina toimivat eri asiakastapaukset, jotka on kuvattu aiemmin luvussa 5.2.2. Syynä kevyeen arviointiprosessiin oli se, että tehty ohjelmistoarkkitehtuuri kuvaa järjestelmää hyvin korkealla tasolla ja lisäksi arkkitehtuuri on jo valmiiksi muodostettu eri sidosryhmien näkemyksistä. Kohdeorganisaatiolla ei ollut mahdollisuutta koota kaikkia sidosryhmiä samaan paikkaan suorittamaan arviointia.

Arkkitehtuurin arvioinnista asiakastapauksia vasten saatiin seuraavia tuloksia, jotka tukevat esitetyn arkkitehtuurin soveltumista erilaisiin asiakastapauksiin. Taulukko 6 esittää asiakastapauksien kuvaukset ja selityksen, millä tavoin arkkitehtuuri tukee asiakastapauksen toteuttamismahdollisuutta. Koska asiakastapaukset olivat myös arkkitehtuurin suunnittelussa huomioituja seik-

koja, olivat taulukossa esitetyt tulokset ennakoitavissa. Tuotetun arkkitehtuurin todettiin soveltuvan hyvin kaikkien kolmen asiakastapauksien toteuttamiseen.

Taulukko 6: Arkkitehtuurin arviointi asiakastapauksia vasten

Asiakas-skenaario	**Kuvaus**	**Arkkitehtuurin tuki skenaarioon**
Vaihtoehto A	Mittauspalvelin on sijoitettu asiakkaan ympäristöön. Avainluvut viedään palveluntarjoajan ylläpitämään web-palveluun.	Kaksisuuntainen rajapinta mahdollistaa mittauspalvelimen kommunikaation web-palvelimelle myös palomuurin takaa
Vaihtoehto B	Mittauspalvelin ja web-palvelin on sijoitettu asiakkaan ympäristöön, johon palveluntarjoajalla on rajattu pääsy	Yksi web-palvelin palvelee yhtä asiakasta, joten sen sijoituspaikka ei ole riippuvainen muista järjestelmistä tai tietokannoista. Tarvittaessa mittauspalvelin ja web-palvelin voivat sijaita samalla fyysisellä palvelimella.
Vaihtoehto C	Mittauspalvelin ja web-palvelin ovat sijoitettu palveluntarjoajan ylläpitoon, jolloin ainoastaan anturit sijaitsevat asiakkaan fyysisessä ympäristössä	Mittauspalvelimen ja anturien fyysisellä sijainnilla ei ole väliä, kunhan niiden välillä on tietoliikenneyhteys. Arkkitehtuuri ja rajapinnan yhteydenottomuodot tukevat mittauspalvelimen ja web-palvelimen sijoittelua tarvittaessa samalla loogisella palvelimelle palveluntarjoajan ylläpitoon.

Asiakastapausskenenaarioiden lisäksi arkkitehtuuria arvioitiin järjestelmän suorituskyvyn ja skaalautuvuuden näkökulmista. Arvioinnin mukaan järjestelmän arkkitehtuuri ei ole este suorituskyvylle tai skaalatutuvuudelle, vaan kyse on ennen kaikkea ohjelmistokehitysvaiheessa tehtävistä teknologiavalinnoista. Taulukko 7 esittää skenaariot sekä selityksen siitä, kuinka arkkitehtuuri tukee skenaarioiden toteutumista. Tässä yhteydessä ei voida täysin varmistua siitä, että järjestelmä tukee esimerkiksi useita satoja käyttäjiä tai useita eri mittauspalvelimia. Arkkitehtuurin näkökulmasta suorituskyvylle ei nähty estettä, joten käytännön suorituskyky riippuu ohjelmistokehityksessä tehdyistä valinnoista. Arkkitehtuuri pyrkii kuitenkin tukemaan suorituskykyä eri skenaarioissa. Järjestelmän suorituskykyä voitaisiin arvioida paremmin, mikäli eri ohjelmisto-osissa käytettävät ohjelmistokehykset olisivat jo suunnitteluvaiheessa selvillä.

Uuden arkkitehtuurin esiversiossa kaikki hälytystoiminnallisuus oli siirretty web-palvelimen vastuulle. Arvioinnin aikana todettiin kuitenkin, että myös mittauspalvelimella täytyy olla edelleen mahdollisuus ainakin SMS-hälytyksien tekoon kriittisissä hälytystilanteissa. Tällöin hälytyksien perillemeno ei ole kiinni web-palvelimen tai tietoliikenneyhteyksien toiminnasta. Arkkitehtuurin arvioinnissa ei löydetty muita sellaisia puutteita tai riskejä, jotka olisivat vaikuttaneet arkkitehtuuriratkaisujen muuttumiseen. Ennen kaikkea syynä tähän on se, että arkkitehtuuri on kuvattu korkealla tasolla. Tehdyt ratkaisut on tehty yhdessä arvioijien kanssa, mikä saattaa myös heikentää arvioinnin luo-

tettavuutta. Luotettavimpia tuloksia olisi mahdollisesti saatu tutkimusjärjestelyllä, jossa arkkitehtuurin arviointia suorittavat eri henkilöt kuin suunnitteluun osallistuvat. Arkkitehtuurin suunnittelun aikana tehtyjä ratkaisuja arvioitiin samanaikaisesti suunnitteluvaiheessa, mikä on myös voinut vaikuttaa näihin tuloksiin.

Taulukko 7: Arkkitehtuurin arviointi kuormitusskenaarioita vasten

Skenaario	**Arkkitehtuurin tuki ja riskit skenaarioon**
Paljon (100) yhtäaikaisia käyttäjiä yhdessä web-käyttöliittymässä	REST-rajapinnan ansiosta web-käyttöliittymä voidaan siirtää erilliselle fyysiselle palvelimelle. Tarvittaessa web-palvelimen tietokanta voidaan vaihtaa suorituskykyisempään muuttamatta käyttöliittymää.
Suuri (10) määrä mittauspalvelimia yhdellä asiakkaalla	Web-palvelin voidaan sijoittaa pilvipalveluun, jossa sen käytössä olevia resursseja voidaan säätää dynaamisesti. Kyse on kuitenkin ennen kaikkea rajapinnan tehokkuudesta ja resurssivaatimuksista.
Satoja eri avainlukuja web-palvelussa	Web-palvelin voidaan sijoittaa pilvipalveluun, jossa sen käytössä olevia resursseja voidaan säätää joustavasti. Käytettävät tietokantaratkaisut ratkaisevat lopulta kuinka paljon avainlukuja voidaan tallentaa ja esittää. Rajapinta sekä tietoliikenneyhteydet voivat myös asettaa rajoitteita sille, kuinka paljon avainlukuja mittauspalvelin voi reaaliaikaisesti syöttää eteenpäin.

5.4.2 Ajonaikaisten ominaisuuksien mukainen vertailu

Seuraavaksi verrataan uutta ja vanhaa ohjelmistoarkkitehtuuria toisiinsa Reekien ym. (2006) esittämien ajonaikaisten ominaisuuksien mukaisesti. Näitä ominaisuuksia ovat suorituskyky, käytettävyys, luotettavuus ja turvallisuus. Ohjelmiston *suorituskykyä* voidaan arvioida monella eri tavalla. Suorituskykyarviointia voidaan tehdä esimerkiksi arvioimalla, kuinka nopeasti tuotantokoneelta tapahtuva muutos välitetään tietyn raja-arvon ylittyessä käyttäjälle hälytyksen välityksellä. Muita mahdollisia mittauskohteita ovat, kuinka nopeasti käyttäjä saa yhteyden järjestelmään tai kuinka nopeasti jokin tietty raportti muodostuu. Tässä tapaustutkimuksessa ei lähdetty tarkemmin määrittelemään mitattavia suureita, vaan suorituskykyä pyrittiin arvioimaan tärkeimpien ominaisuuksien perusteella. Tärkeimmät suorituskyvyn mittarit ovat tässä reaktioaika hälytyksiin sekä raporttien generointiaika. Uudessa arkkitehtuurissa hälytystoiminnallisuuden siirtäminen anturin lähellä sijaitsevasta mittauspalvelimesta web-palvelimelle voi heikentää hälytyksien reaktioaikaa. Toimiva internetyhteys ja rajapinta varmistavat kuitenkin lähes välittömän avainluvun tiedonsiirron web-palvelimelle, jossa hälytysten tila tarkistetaan. Mittauspalvelimelle on tästä syystä jätetty kuitenkin varaus SMS-hälytyksien tarjoamiseen, koska ne eivät ole tietoliikenneyhteydestä riippuvia. Joidenkin raporttien muodostaminen voi olla raskasta. Raportointiominaisuuden siirtäminen web-palvelimen vastuulle tuo suorituskykyetua tilanteessa, jossa web-palvelin on sijoitettu palveluntarjo-

ajan konesaliin tai pilvipalveluun. Tällöin käytössä on yleensä enemmän resursseja kuin asiakkaan ympäristössä toimivassa mittauspalvelimessa.

Uuden arkkitehtuurin lähtökohdaksi on otettu *käytettävyyden* parantaminen. Tässä yhteydessä käytettävyyttä tarkasteltiin loppukäyttäjän käyttäjäkokemuksen helppokäyttöisyyden ja nopeuden perusteella. Vanhan arkkitehtuurin tapauksessa käytettävyys on heikentynyt, mikäli tietoliikenneyhteys mittauspalvelimeen on ollut heikko. Uuden arkkitehtuurin ansiosta konfigurointi tehdään web-palvelun kautta, joka on aina tavoitettavissa johtuen sen sijainnista asiakasympäristön ulkopuolella. Kahden eri järjestelmän (WebUI, konfigurointityökalu) yhdistäminen yhdeksi web-palveluksi edistää käytettävyyttä, koska asiakas ei enää tarvitse erikseen asennettavia ohjelmistoja käyttääkseen palvelua. Uuden järjestelmän lopullinen käyttökokemus riippuu kuitenkin tehdyistä teknologiavalinnoista ja ohjelmistokehityksestä, joten käyttökokemusta on tässä yhteydessä mahdotonta arvioida kokonaisvaltaisesti.

Hälytyksien siirtäminen web-palvelimen tehtäväksi voi heikentää järjestelmän *luotettavuutta*. Aiemmin hälytystoiminnallisuus on toteutettu mittauspalvelimella, joka on lähettänyt sähköposti- ja tekstiviestit välittömästi hälytyksen havaittuaan. Uudessa järjestelmässä avainluvut ensin siirretään web-palvelimelle, jonka jälkeen luodaan tarvittavat hälytykset. Tietoliikenneyhteysongelmien vuoksi hälytyksien lähettäminen voi viivästyä. Arkkitehtuurin arvioinnin yhteydessä mittauspalvelimelle jätettiin kuitenkin mahdollisuus tehdä edelleen kriittiset hälytykset esimerkiksi SMS-yhteyden avulla.

Turvallisuuden kannalta kriittisin komponentti on asiakkaan tiloissa sijaitseva mittauspalvelin. Uudessa arkkitehtuurissa tietoa ei enää ylläpidetä pitkiä aikoja mittauspalvelimella, vaan se yleensä siirretään keskitetylle web-palvelimelle, jolloin tiedon turvallisuus on helpompi taata. Varmuuskopiointi on uuden arkkitehtuurin myötä helpompaa toteuttaa, koska web-palvelin yleensä sijaitsee palveluntarjoajan tiloissa nopeiden tietoliikenneyhteyksien päässä. Aiemmin tiedon sijainti mittauspalvelimella hitaan tietoliikenneyhteyden varassa teki varmuuskopioinnista hankalaa. Uudessa arkkitehtuurissa web-palvelimen ja mittauspalvelimen välillä on tarkoitus käyttää vahvaa salausta, kuten tällä hetkellä WebUI:n ja mittauspalvelimen välillä.

5.4.3 Ei-ajonaikaisten ominaisuuksien mukainen vertailu

Tässä kohdassa verrataan vanhaa ja uutta ohjelmistoarkkitehtuuria ei-ajonaikaisten ominaisuuksien mukaisesti. Niitä ovat Reekien ym. (2006) mukaan ylläpidettävyys, testattavuus, uudelleenkäytettävyys, konfiguroitavuus ja laajennettavuus. *Ylläpidettävyys* paranee uuden järjestelmän myötä. Aiemmin konfigurointityökalun ja mittauspalvelimen muuttunut rajapinta vaati loppukäyttäjän toimia konfigurointityökalun päivittämiseksi. Uudessa arkkitehtuurissa päivittäminen on helpompaa, koska sekä mittauspalvelin että web-palvelin ovat käyttäjästä riippumattomia. Ohjelmistokoodin ylläpidettävyyteen uudella arkkitehtuurilla ei ole vaikutusta.

Arkkitehtuuritasolla *testattavuuteen* ei tule muutoksia. Testausta on suoritettu yksikkötesteillä, joita myös uusi arkkitehtuuriratkaisu tukee. Vanhalle WebUI:lle ei ole kirjoitettu testejä, mutta uudet web-käyttöliittymäkehykset tukevat testaamista. Arkkitehtuureilla ei ole mainittavaa eroa myöskään *uudelleenkäytettävyyden* osalta.

Uusi arkkitehtuuri mahdollistaa usealta eri mittauspalvelimelta koostetun tiedon esittämistä asiakkaalle yhdessä web-palvelussa. Näin sen *skaalautuvuus* on parempi vanhaan arkkitehtuuriin verrattuna. Web-palvelimen ytimen ja web-käyttöliittymän erottaminen REST-rajapinnan avulla mahdollistaa niiden sijoittamisen tarvittaessa fyysisesti eri palvelimille, joten palvelu mahdollistaa suurenkin kuormituksen.

Uuden järjestelmän myötä järjestelmän *konfiguroitavuus* paranee. Aiemmin käyttäjän halutessa muokata esimerkiksi avainlukumoduulin sijaintia karttanäkymässä (vrt. Kuvio 12) käyttäjä on joutunut käyttämään työpöytäkäyttöistä konfigurointityökalua parametrien syöttämiseen. Web-teknologioiden käyttö mahdollistaa sijainnin muuttamisen suoraan karttanäkymässä ilman erillistä konfigurointityökalua. Järjestelmätasolla konfiguroitavuuteen ei tule muutoksia, vaan järjestelmä voidaan edelleen asentaa erilaisiin asiakasympäristöihin.

Uuden järjestelmän rakentamisen lähtökohtana on ollut parempi *laajennettavuus*. Vanha järjestelmä ei suoraan tue esimerkiksi useampaa mittauspalvelinta, eivätkä tehdyt rajapintaratkaisut skaalaudu isolle asiakasmäärälle. Uudessa järjestelmässä arkkitehtuurin skaalautuvuutta on testattu erilaisia laajentumisskenaarioita vasten (vrt. Taulukko 7), joten sen laajennettavuus on parempi.

5.5 Yhteenveto tuloksista

Tässä luvussa esitettiin tapaustutkimuksen tulokset. Ensimmäisessä alaluvussa esiteltiin nykyinen ohjelmistoarkkitehtuuri, joka koostuu neljästä komponentista, jotka ovat anturit, mittauspalvelin, WebUI ja konfigurointityökalu. Ohjelmistokokonaisuuden rakentumista pala kerrallaan kuvattiin historian valossa. Ohjelmiston arkkitehtuurin ongelmakohtia kerrottiin, joista tärkeämpinä ovat käytettävyysongelmat ja skaalautuvuuden puuttuminen. Lopuksi todettiin, että nykyisessä arkkitehtuurissa määrittävimpänä on asiakas-palvelin-arkkitehtuurityyli.

Toisessa alaluvussa esitettiin ohjelmistoarkkitehtuurin suunnittelumenetelmän valintaa ja suunnitteluprosessia. Ensin kuvattiin suunnitteluun vaikuttavaa kontekstia, tavoitetta ja tilannetekijöitä. Tarjolla olleista suunnittelumenetelmistä valittiin ominaisuusvetoinen arkkitehtuurisuunnittelu, koska se sisältää selkeästi kuvatun prosessin ja on saavuttanut vakiintuneen aseman. Suunnittelumenetelmän valinnan jälkeen kuvattiin suunnitteluprosessi ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän vaiheiden ja iteraatioiden mukaisesti. Tuloksena syntyi arkkitehtuuri, jonka keskeisimmät osat ovat mittauspalvelin, web-palvelin sekä niitä yhdistävä rajapinta. Web-palvelimelta erotettiin lisäksi ydin ja web-käyttöliittymä.

Suunnitteluprosessin tuloksena saatu ohjelmistoarkkitehtuuri esitettiin kolmannessa alaluvussa. Tässä yhteydessä kerrottiin arkkitehtuurikomponenttien muuttuneista rooleista ja arkkitehtuurityyleistä. Ohjelmiston toimintalogiikkaa siirrettiin entistä enemmän web-palvelimen vastuulle. Mittauspalvelimen tehtäväksi jäi tiedon mittaaminen antureilta sekä mittaustiedosta koostettujen avainlukujen välitys web-palvelimelle. Ohjelmiston hälytys- ja raportointitoiminnallisuus siirrettiin web-palvelimen tehtäväksi. Lopuksi kerrottiin uudessa arkkitehtuurissa esiintyvistä arkkitehtuurityyleistä. Uusi arkkitehtuuri sisältää piirteitä REST-arkkitehtuurista, asiakas-palvelin arkkitehtuurista, viestinvälitysarkkitehtuurista, tietovuoarkkitehtuurista ja MV*-arkkitehtuurista.

Lopuksi suoritettiin arkkitehtuurin arviointi ja nykyisen sekä uuden arkkitehtuurin vertailua laadullisten ominaisuuksien perusteella. Arviointimenetelmänä käytettiin ATAM-menetelmää, sillä se on saavuttanut vakiintuneen aseman ja on selkeästi kuvattu, sekä nopeasti omaksuttavissa. Arkkitehtuurin arvioinnissa käytettiin arkkitehtuurin testaamista skenaarioita vasten. Testattavat skenaariot koskivat eri asiakastapauksia ja järjestelmän suorituskykyä. Uutta ja vanhaa arkkitehtuuria vertailtiin Reekien ym. (2006) laadullisten ominaisuuksien luokittelun perusteella. Uuden arkkitehtuurin todettiin arkkitehtuuritasolla olevan ajonaikaisten ominaisuuksien (suorituskyky, käytettävyys, luotettavuus ja turvallisuus) perusteella parempi, vaikka ominaisuuksia on arkkitehtuuritasolla haastavaa vertailla. Ei-ajonaikaisten ominaisuuksien (ylläpidettävyys, testattavuus, uudelleenkäytettävyys, konfiguroitavuus ja laajennettavuus) perusteella uusi arkkitehtuuri on helpommin ylläpidettävissä, konfiguroitavissa ja laajennettavissa.

6 POHDINTA

Tämän tapaustutkimuksen tarkoituksena oli selvittää, millä tavalla voidaan valita ja soveltaa ohjelmistoarkkitehtuurin suunnittelumenetelmää ja arvioida tuloksena saatua ohjelmistoarkkitehtuuria. Tapaustutkimuksen kohteena oli tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän (WBS) arkkitehtuurin suunnittelu. Seuraavaksi esitetään ensin tiivistetysti tutkimuksen tulokset, verrataan niitä aiempaan tutkimukseen ja vedetään johtopäätöksiä. Toiseksi kerrotaan tutkimuksen tulosten hyödyntämismahdollisuuksista. Kolmanneksi tarkastellaan tulosten reliabiliteettia ja validiteettia.

6.1 Tulokset ja johtopäätökset

Tutkimuksen tuloksia käsitellään tutkimusmallin (Kuvio 11) mukaisesti kolmessa osassa: suunnittelukonteksti, ohjelmistoarkkitehtuurin suunnittelu ja uusi ohjelmistoarkkitehtuuri.

6.1.1 Suunnittelukonteksti

Tapaustutkimus toteutettiin yrityksessä, joka tekee ohjelmistoratkaisua teollisuuden kunnonvalvontaan ja tuotantotehokkuuden seurantaan. Ohjelmistoratkaisun parissa työskentelee noin kymmenen henkeä ohjelmistokehityksessä, myynnissä ja asiakastuessa.

Ohjelmiston nykyinen arkkitehtuuri on esitetty kuviossa 13. Arkkitehtuurin pääkomponentit anturit, mittauspalvelin, web-käyttöliittymä (WebUI) ja työpöytäkäyttöinen konfigurointityökalu. Anturien välittämä tieto viedään ethernet-verkossa mittauspalvelimen kautta internetiin kytkettyyn web-käyttöliittymään, josta käyttäjät voivat hakea tarvitsemansa tiedon eri päätelaitteilla. Käyttäjän palvelusta saama arvo välittyy web-käyttöliittymän kautta. Järjestelmä on rakennettu siten, että käyttäjän kannalta kriittinen tieto säilytetään ja jalostetaan mittauspalvelimella. Tästä syystä nykyisen web-käyttöliittymän

ominaisuudet ovat hyvin rajatut, sillä se on suunniteltu näyttämään vain avainlukutietoa käyttäjälle. Aidossa web-palvelussa tieto täytyy olla lähempänä käyttäjää, eikä eristettynä palomuurin ja hitaan tietoliikenneyhteyden takana sijaitsevaan mittauspalvelimeen. Järjestelmän konfigurointi suoritetaan mittauspalvelimelle konfiguraatiotyökalun avulla. Mittauspalvelin välittää tarvittavat tiedot web-käyttöliittymälle. Käyttäjä ei pysty tekemään konfiguraatiomuutoksia web-käyttöliittymän kautta, vaan joutuu käyttämään työpöytäkäyttöistä konfigurointityökalu tehdäkseen muutoksia järjestelmän toimintaan. Käytännössä loppukäyttäjät eivät tee konfiguraatiomuutoksia, vaan niistä vastaa tällä hetkellä yrityksen asiakastuki.

Teollisuuden ohjelmistot ovat siirtymässä kuluttajapalveluiden ohella entistä enemmän www-sovelluksiksi. Toinen suuntaus on kiinteähintaisten laitteiden ja palveluiden myyminen palveluliiketoiminnan avulla. Kohdeorganisaation tavoitteena on myydä asiakkaiden tuotannon tehokkuuden parantamiseksi alusta, jossa ohjelmisto- ja laiteratkaisut toimivat välineenä. Nykyinen ohjelmistoarkkitehtuuri ei tue näitä ajatuksia, sillä nykyinen web-käyttöliittymä ja järjestelmän kokonaisarkkitehtuuri ovat hankalasti laajennettavissa ja skaalattavissa eri käyttötapauksiin. Kohdeorganisaatio haluaa olla mukana teollisuuden muutoksessa tarjoamalla asiakkaille reaaliaikaista tilatietoa heidän tuotannostaan helppokäyttöisen web-käyttöliittymän avulla. Käyttäjäkonfiguroinnin ja järjestelmäkonfiguroinnin tuominen web-käyttöliittymään mahdollistaa tuotteen myymisen jälleenmyyjien kautta. Tällä hetkellä konfigurointityökalun käyttäminen vaatii järjestelmän toiminnan erityistä tuntemusta, minkä takia järjestelmän käyttöönotto asiakastilanteessa täytyy käytännössä tehdä kohdeorganisaation toimesta. Järjestelmästä halutaan tehdä niin helppokäyttöinen, että loppukäyttäjä tai tuotteen jälleenmyyjä voivat ottaa sen itsenäisesti käyttöön. Tilanne kilpailijoihin verrattuna on kuitenkin erinomainen. Kilpailijoiden vastaavien järjestelmien käyttöönotto asiakaskohteissa voi viedä useita viikkoja tai kuukausia, mikäli järjestelmän toiminta perustuu ohjelmistointegraatioiden tekemiseen laitteiden automaatiorajapintaan.

Tähän tapaukseen liittyy monia asioita, joita voidaan tunnistaa tapahtuvan myös muissa ohjelmistokehitystä tekevissä organisaatioista. Ohjelmistokehityksen pääpaino on yleensä asiakastarpeiden, eli ominaisuuksien, toteuttamisessa, mistä syystä arkkitehtuuri rämettyy vähitellen. Teknologioiden vaihtuessa ja uudistuessa aiemmin tehdyt päätökset vanhentuvat ja saattavat jopa muuttua tietoturvauhkiksi. Ohjelmistoon syntyy korjausvelkaa, jota ei yleensä järjestelmällisesti pyritä tai ehditä poistamaan. Näistä syystä ohjelmistoa päädytään päivittämään yleensä liian myöhään, jolloin alkuperäinen arkkitehtuuri ei välttämättä ole enää sovelias tukemaan ohjelmiston uusia tarpeita.

Tähän tapaukseen liittyy tilannetekijöitä, jotka koskevat vain tätä tapausta. Tässä tapauksessa arkkitehtuurinsuunnittelu tehdään pääasiassa vain yhden henkilön toimesta. Tämän lisäksi tutkija toimii organisaation ulkopuolelta käsin, mutta hänellä on kuitenkin hyvä tietämys kohdeorganisaatiosta ja ohjelmistosta. Samanlaista kokemusta ja näkemystä ei esimerkiksi olisi tehtävään palkatulla ulkopuolisella konsultilla, koska hän ei voisi perehtyä ohjelmiston rakenteeseen,

historiaan ja ratkaisujen taustoihin yhtä kattavasti. Koska tutkijalla ei ole kattavaa kokemusta eri ohjelmistokehyksistä ja ohjelmistokielistä, on niiden käsittely jätetty pois tästä tutkimuksesta. Arkkitehtuuriratkaisuissa voitaisiin nojata enemmän ohjelmistokehyksiin kuten Cervantes ym. (2013) ovat tehneet. Toisaalta tämän tutkimuksen tulokset voivat pysyä ajattomampana, koska ohjelmistokehykset ja ohjelmistokielet kehittyvät nopealla vauhdilla.

6.1.2 Ohjelmistoarkkitehtuurin suunnittelu

Alan peruskirjallisuuteen (Shaw & Garlan, 1996; Bass ym., 2003; Clements ym., 2002; Clements ym., 2003) tutustumalla päätettiin hyödyntää jotakin ohjelmistoarkkitehtuurin suunnittelumenetelmää. Keskeisenä lähteenä suunnittelumenetelmän valinnassa oli Hofmeisterin ym. (2007) tutkimus neljän suunnittelumenetelmän vertailusta. Tässä tutkimuksessa ohjelmistoarkkitehtuurin suunnittelussa päätettiin hyödyntää ominaisuusvetoista arkkitehtuurisuunnittelumenetelmää (ADD), koska se oli vertailluista menetelmistä vakiintunein ja käytetyin, sekä koska sen dokumentaatio sisälsi hyvän ohjeistuksen menetelmän käytöstä.

Menetelmän mukaisesti suunnittelun kohteeksi kulloinkin valitut järjestelmäosat kuvattiin iteraatiokierroksilla. Tässä tapauksessa iteraatiokierroksia käytiin läpi kolme. Suunnittelu alkoi tärkeimpien vaatimusten ja rajoitteiden kokoamisella, minkä nykyisen ohjelmiston tuoteomistaja suoritti yhdessä tutkijan kanssa. Tuoteomistaja antoi järjestelmän rakentamista varten tärkeimmät vaatimukset ja rajoitteet, joita käytettiin arkkitehtuurisuunnittelun pohjana. Tässä vaiheessa ei koostettu täydellistä vaatimuslistaa, vaan järjestelmää käsiteltiin MVP-tuotteen periaatteiden mukaisesti. Ensimmäisellä iteraatiokierroksella kuvattiin järjestelmä kokonaisuudessaan, jolloin syntyivät mittauspalvelin, web-palvelin ja rajapintaelementti. Toisella iteraatiokierroksella kuvattiin ensimmäiseltä iteraatiokierrokselta syntynyt web-palvelin, josta erotettiin web-käyttöliittymä ja web-palvelimen ydin. Viimeisellä iteraatiokierroksella määriteltiin vielä web-käyttöliittymän sisäistä arkkitehtuuria. Lopputuloksena saatiin ohjelmistoarkkitehtuuri, jonka tärkeimmät komponentit sekä roolit ja komponenttien väliset rajapinnat oli määritelty.

Tämän prosessin arvioinnin mittareina ovat suunnitteluprosessin tuotoksen laatu sekä käytetyn prosessin toiminta, tehokkuus ja sopivuus tähän tapaustutkimukseen. Lisäksi arvioidaan sidosryhmien huomioonottamista. Tuotteen laatu on näistä prosessin lopullinen laadun mittari ja sitä arvioidaan seuraavassa kohdassa.

Menetelmän käytöstä pienen ohjelmiston kuvaamisessa teki haastavan vaatimuksien kuljetus pääelementeiltä lapsielementeille (ADD, työvaihe 8). Menetelmän dokumentaatio ei suoraan suosittanut, kuinka tämä vaihe voitaisiin käytännössä toteuttaa. Menetelmä ei tarjonnut työkalutukea prosessin läpiviemiseen, mikä olisi voinut helpottaa arkkitehtuurisuunnittelun toteutusta. Työkalun avulla useista kymmenistä iteraatioista koostuvasta arkkitehtuurista voitaisiin helposti luoda johtopäätösketju, jonka avulla voidaan selvittää, mikä on aiheuttanut esimerkiksi jonkin tietyn elementin luomisen. Nyt tällaisten pää-

tösketjujen luominen on hankalaa, sillä menetelmän dokumentaatio ei tarkasti kuvaa, millaisia merkintöjä työvaiheissa täytyy tehdä. Toisaalta tämä jättää sijaa menetelmän soveltamiseen eri kohdeongelmissa. Yksinkertaisissa arkkitehtuureissa päätösketjut ovat helposti nähtävillä ja selvitettävissä, mutta isossa projektissa on ensiarvoisen tärkeää selvittää, mitkä ja millä tasolla tehdyt päätökset vaikuttavat tiettyihin alemman tason elementteihin ja valittuihin taktiikoihin.

Menetelmän käyttö auttoi tutkijaa perustelemaan tehtyjä arkkitehtuurivalintoja. Menetelmä tarjosi systemaattisen lähestymistavan prosessin läpivientiin. Vaikka alussa prosessi tuntui epäselvältä ja tuloksen saaminen epävarmalta, auttoi selkeä vaiheittainen toimintatapa lähestymään ongelmaa askel kerrallaan. Menetelmän käyttö vaatii hyvän perustietämyksen siitä, kuinka arkkitehtuurillisia ongelmia voidaan ratkaista tai millaisia ongelmia arkkitehtuurissa voi ylipäätään olla. Menetelmän käytössä haastavinta onkin juuri työvaihe 4, jossa arkkitehtuurilliset vaatimukset pyritään täyttämään käyttämällä jotakin taktiikkaa. Taktiikka voi esimerkiksi olla arkkitehtuurityyli tai jokin ohjelmistokehys. Koska tämä vaatii paljon käytännön tietoa ohjelmistokehyksistä ja arkkitehtuurityyleistä, on suunnitteluprosessi haastava henkilölle, jolla ei ole aiempaa kokemusta näistä asioista. On siis olemassa perusteltu syy sille, miksi ohjelmistoarkkitehdit yleensä ovat harmaahiuksisia.

Prosessin tehokkuutta voidaan arvioida tuotoksen ja kulutettujen resurssien suhteella. Arkkitehtuurisuunnitteluun käytetyistä tunneista ei pidetty tarkkaa kirjaa, mutta tuotos syntyi suhteellisen nopeassa ajassa sen jälkeen, kun varsinaiseen työhön päästiin. Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän selkeästi kuvatut työvaiheet auttoivat menetelmän nopeaa omaksumista. Ajallisesti haastavampaa oli etukäteistutustuminen ohjelmistoarkkitehtuuriin, arkkitehtuurityyleihin ja suunnittelumenetelmiin.

Ominaisuusvetoinen arkkitehtuurisuunnittelu on kehitetty suuriin ohjelmistoprojekteihin, joten sen käyttäminen pienessä ohjelmistossa ketterän kehityksen kontekstissa voidaan kyseenalaistaa. Tässä työssä suunniteltiin kokonaisarkkitehtuuri kuvatulle järjestelmälle. Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän avulla arkkitehtuuri voidaan suunnitella hyvin tarkalla tasolla, eikä menetelmän pääpaino ole kokonaisarkkitehtuurin kuvaamisessa. Menetelmä oli kuitenkin sovellettavissa myös tällaiseen tapaukseen.

Valittu menetelmä ei ota kattavasti huomioon arkkitehtuurisuunnittelun sosiaalista kontekstia (Hofmeister ym., 2007). Menetelmän esimerkkikuvauksessa (Wojcik ym., 2006) arkkitehtuurisuunnittelua hajautetaan eri alojen asiantuntijoille. Tässä tapaustutkimuksessa sidosryhmiltä (asiakas, asiakastuki, ohjelmistokehittäjät) selvitettiin vaatimuksia ja rajoitteita, joiden perusteella tutkija teki arkkitehtuuripäätökset. Tehdyt arkkitehtuuripäätökset varmennettiin ohjelmistokehittäjillä. Tällainen toimintatapa ei ole menetelmän suositus, vaan se muodostui menetelmän käytön yhteydessä.

Yhteenvetona voidaan todeta, että ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä on selkeästi kuvattu ja skaalautuva ohjelmistoarkkitehtuurin suunnittelumenetelmä. Se tukee arkkitehtuurin suunnittelua, mutta vaatii taakseen paljon asiantuntemusta taktiikoista (esim. ohjelmistokehykset, ark-

kitehtuurityylit), joiden avulla arkkitehtuurillisia ongelmia voidaan ratkaista. Toteutettu prosessi oli suoraviivainen ja nopea, jonka lopputuloksena saatiin laadukas tuotos aikaan. Ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä tarjosi johdonmukaisen lähestymistavan suunnitteluongelmien ratkaisemiseksi, mikä tuki prosessin onnistumista.

6.1.3 Uusi ohjelmistoarkkitehtuuri

Uuden ohjelmistoarkkitehtuurin pääosat ovat mittauspalvelin, web-palvelin ja niitä yhdistävä rajapinta. Mittauspalvelimen tehtävänä on hakea antureilta mittaustietoa ja jalostaa siitä avainlukutietoa konfiguraation mukaisesti. Avainluvut siirretään kaksisuuntaisen rajapinnan kautta web-palvelimen tietokantaan. Asiakas voi web-käyttöliittymän kautta muodostaa haluamiaan raportteja ja hälytyksiä. Oleellinen lisäys vanhaan arkkitehtuuriin on web-palvelimen tietokanta, johon avainlukutietoa voidaan koostaa useammalta eri mittauspalvelimelta.

Arkkitehtuurityylillisesti järjestelmä ei noudata mitään yksittäistä periaatetta. Käytetyt arkkitehtuurityylit ovat siis Koskimiehen ja Mikkosen (2005) mukaisesti typistetty paikallisiksi suunnittelupäätöksiksi. Järjestelmän uusien osien osalta mittauspalvelimen ja web-palvelimen rajapinta muistuttaa asiakaspalvelin arkkitehtuurityyliä. Tämän arkkitehtuurityylin periaatteita mukaillen rajapinnasta saadaan selkeä, minkä ansiosta mittauspalvelinta ja web-palvelinta voidaan kehittää samanaikaisesti. Web-palvelimen ytimen ja käyttöliittymän välillä vallitsee kerrosarkkitehtuuri, jonka rajapinta toteutetaan REST-arkkitehtuurityylin mukaisesti. Kerrosarkkitehtuurin avulla käyttöliittymä erotetaan sovelluslogiikasta ja tietokannoista, mikä mahdollistaa paremman skaalautuvuuden ja ylläpidettävyyden (Buschmann ym., 1995). REST-arkkitehtuurin avulla saadaan selkeä, nopea ja hyvin määritelty rajapinta, jonka avulla järjestelmän tietoa on mahdollista viedä kolmannen osapuolen järjestelmiin.

Kohdeorganisaation asiakastukihenkilöt ja ohjelmistokehittäjät arvioivat uutta arkkitehtuuria. ATAM-arviointimenetelmän (Kazman ym., 1998) käytännöistä poimittiin käyttöön arkkitehtuurin testaaminen skenaarioilla. Myös Nord ja Tomayko (2006) poimivat ketterään XP-ohjelmistoprojektiinsa ATAM-menetelmästä arkkitehtuurin testaamisen skenaarioita vasten. Arvioinnin seurauksena elementtien tehtäviä ja roolijakoa täydennettiin ja täsmennettiin saatujen kommenttien perusteella. Arviointia suoritettiin lisäksi testaamalla arkkitehtuuria erilaisia asiakasskenaarioita ja suorituskykytilanteita vasten. Tässä arvioinnissa arkkitehtuurin todettiin läpäisevän testatut skenaariot, vaikkakin lopullinen suorituskyky riippuu ohjelmistokehityksestä ja käytetyistä tekniikasta. Arkkitehtuurillisesti ei kuitenkaan tunnistettu asioita, jotka olisivat aiheuttaneet ongelmia skenaariotilanteissa.

Koskimiehen ja Mikkosen (2005) mukaan ohjelmistoarkkitehtuurin suunnittelua voidaan pitää onnistuneena, mikäli luodun arkkitehtuurin avulla on mahdollista toteuttaa kuvattava järjestelmä siten, että se täyttää toiminnalliset ja laadulliset vaatimukset ottaen huomioon rajoitteet sekä tulevaisuuden laajen-

tumistarpeet. Arkkitehtuurin suunnittelu ei ole itseisarvo, vaan sen tarkoituksena on ennalta tunnistaa riskit ja mahdollistaa ohjelmiston toteutus pysyen aikataulussa ja budjetissa.

Kohdeorganisaatiolle luotu arkkitehtuurisuunnitelma sai hyvän vastaanoton. Sen koettiin tuovan esille nykyisen arkkitehtuurin ongelmia ja havainnollistavan riittävällä tasolla tulevaisuuden kehityssuuntaa. Arkkitehtuuri ilmentää johdon näkemystä siitä, mihin suuntaan ohjelmistoratkaisua tulisi viedä. Tehty arkkitehtuuri rakennettiin nykyisen arkkitehtuurin pohjalta, joten täysin uuden ja erillisen järjestelmän rakentamisesta ei ollut kyse. Vasta tulevaisuudessa voidaan luotettavasti arvioida, onko arkkitehtuurissa ollut olennaisia virheitä, jotka olisi täytynyt huomata jo arkkitehtuurin suunnitteluvaiheessa. Tässä tutkimuksessa on kuitenkin tehty voitava uuden arkkitehtuurin luotettavuuden ja laadun varmistamiseksi.

6.2 Tutkimuksen hyödyntäminen

Seuraavassa kerrotaan, kuinka tämän tutkimuksen tuloksia voidaan käyttää hyödyksi. Ensin kerrotaan, kuinka kohdeorganisaatio voi hyödyntää tutkimuksen tuloksia. Tämän jälkeen kerrotaan, miten tutkimuksen tuloksia voidaan hyödyntää muissa yhteyksissä.

6.2.1 Tulosten hyödyntäminen kohdeorganisaatiossa

Tehdyn arkkitehtuurisuunnitelman avulla kohdeorganisaation on mahdollista objektiivisesti arvioida, onko ohjelmistokehityksen pääpainon siirtäminen uuden arkkitehtuurin avulla web-teknologioiden päälle järkevää. Tehdyt suunnitelmat toimivat pohjamateriaalina, kun yrityksen johto tekee strategista suunnittelua tulevaisuutta varten. Mikäli ohjelmistoa lähdetään kehittämään tässä tutkimuksessa esitettyyn suuntaan, toimivat tutkimuksen yhteydessä syntyneet dokumentit ohjelmiston suunnittelun lähtökohtina. Arkkitehtuurin suunnitteluprosessin tulosta voidaan siis suoraan hyödyntää organisaatiossa.

Kohdeorganisaatioon on tapaustutkimuksen toteuttamisen aikana syntynyt osaamista ohjelmistoarkkitehtuurin suunnittelusta ja huomioimisesta. Työntekijöitä on pyritty osallistuttamaan suunnitteluun, minkä ansiosta he saattavat jatkossa olla valveutuneempia ajattelemaan ohjelmistokehitystä arkkitehtuurin ja kokonaisuuksien kautta. Tämän tapaustutkimuksen toimintatapaa voidaan myöhemmin soveltaa kohdeorganisaatiossa vastaavissa suunnittelutilanteissa. Nykyisen arkkitehtuurin ongelmakohtien esilletuonti toimii palautekanavana ja kannustaa ohjelmistokehittäjiä ja johtoa kiinnittämään huomiota arkkitehtuuriin aikaisessa vaiheessa.

Tutkielman yhteydessä kuvattu arkkitehtuuri, ohjelmiston toimintaperiaate ja historia voivat toimia perehdytysmateriaalina kohdeorganisaation uusille työntekijöille. Uusi arkkitehtuuri voi toimia myös osana myyntimateriaalia, kun

ohjelmiston toimintaperiaatetta esitellään asiakkaille. Lisäksi arkkitehtuuri selventää ja määrittää käsitteitä, jolloin esimerkiksi ohjelmistokehittäjät ja johto voivat puhua asioista samoilla termeillä.

Kohdeorganisaatio voi aloittaa luvussa 3.3 esitetyn Faberin (2010) mallin soveltaminen, mikäli se koetaan hyödylliseksi. Malli esittää Scrum-viitekehyksen mukaisen tuoteomistajan lisäksi pääarkkitehdin nimeämistä, joka ottaa huomioon ohjelmiston arkkitehtuurilliset vaatimukset suunniteltaessa sprintin kehitysjonoa. Tällä tavoin kohdeorganisaatio nimeää vastuuhenkilön, jonka tehtävänä on varmistaa ohjelmiston laadullisten ominaisuuksien ja hyvän arkkitehtuurillisen lähestymistavan toteutuminen.

6.2.2 Tutkimuksen hyödyntäminen muissa yhteyksissä

Vaikka ohjelmistoarkkitehtuureja koskevia tutkimuksia on runsaasti, niiden käytöstä todellisissa tilanteissa on huomattavasti vähemmän. Tässä tutkielmassa luotu tutkimusprosessi (Kuvio 10) on hyödynnettävissä muissa tapaustutkimuksissa, joiden tarkoituksena on selvittää ohjelmistoarkkitehtuurin suunnittelumenetelmän hyödyntämismahdollisuuksia. Tutkimusprosessi voi toimia myös prosessikaaviona organisaatiossa, joka haluaa johdonmukaisen lähestymistavan arkkitehtuurin suunnitteluun. Sen mukaisesti organisaatio ensin tutustuu arkkitehtuuriin, arkkitehtuurin suunnitteluun ja arviointiin ja näiden tietojen pohjalta toteuttaa arkkitehtuurin suunnitteluprosessin. Taustaaineistona arkkitehtuuriin ja suunnittelumenetelmiin voi toimia tämän tutkielman kirjallisuusosio.

Tutkimuksessa luotu arkkitehtuurikuvaus (Kuvio 16) on hyödynnettävissä muissakin teollisen internetin ohjelmistoissa, joiden toimintaperiaate on samankaltainen. Teollisen internetin laitteiden ja ohjelmistojen toimintaperiaate mukailee tässä työssä kuvatun ohjelmiston toimintaperiaatetta. Usein käyttäjällä on anturi tai muu laite, joka lähettää jatkuvasti tietoa internetiin käsiteltäväksi. Käsittely tapahtuu usein pilvipalvelussa ja käyttäjälle tarjotaan web-käyttöliittymä jalostettuun tietoon. Eri antureilta, käyttäjiltä ja muista järjestelmistä saatua tietoa voidaan yhdistää samaan tietokantaan, jolloin kaikki hyötyvät avoimesta tiedon jakamisesta. Vaikka tässä tutkielmassa esitetyssä arkkitehtuurissa tieto on asiakkaan kontrolloitavissa ja omistuksessa, voidaan arkkitehtuuria käyttää keskitetyn järjestelmän arkkitehtuurin pohjana.

Tulosten suoraviivaiseen yleistämiseen ei tapaustutkimuksen luonteesta johtuen ole mahdollisuutta. Muissa organisaatioissa voidaan kuitenkin samantapaisessa tilanteessa ottaa oppia tässä tutkielmassa esitetystä toimintatavasta. Tutkielma toimii esimerkkinä siitä, millaisessa tilanteessa ketterän kehityksen periaatteiden mukaisesti toimiva organisaatio voi hyödyntää arkkitehtuurin suunnittelumenetelmiä. Tutkielma kannustaa onnistuneen prosessin kautta muita organisaatioita järjestelmälliseen arkkitehtuurin suunnitteluun. Huomionarvoista tässä tapauksessa on, että asiakkaat, liiketoiminta-ajatus ja vaatimukset olivat jo lähes valmiina, jolloin tutkijan tehtäväksi jäi näitä tekijöitä hyväksikäyttäen luoda ohjelmistolle arkkitehtuuri. Organisaatiossa, joka alkaa

kehittää ohjelmistoa ”puhtaalta pöydältä”, ei välttämättä ole mahdollista edetä näin johdonmukaisesti, sillä vaatimukset ja asiakastarpeet voivat muuttua radikaalisti lyhyessä ajassa.

6.3 Realibiteetin ja validiteetin arviointi

Tutkimuksen laatua arvioidaan reliabiliteetin ja validiteetin näkökulmasta. Yin (2009) on jakanut validiteetin rakennevaliditeettiin, sisäiseen validiteettiin ja ulkoiseen validiteettiin.

Tutkimuksen *realibiteetti* varmistaa, että tutkimusaskeleet, esimerkiksi tiedonkeruu ja -analysointi, voidaan toistaa siten, että saadaan samat tulokset kuin ensimmäisellä suorituskerralla (Runeson & Höst, 2009; Yin, 2009). Toisin sanoen realibiteetti vahvistaa tutkijan riippumattomuuden tutkimustiedosta ja tutkimustuloksista, eli toisen tutkijan toistaessa tutkimuksen tulisi hänen saada samat tutkimustulokset kuin ensimmäisellä suorituskerralla. Runesonin ja Höstin (2009) mukaan tutkimuksen realibiteetti kärsii esimerkiksi tilanteessa, jossa tiedon analysointi tai tutkimuskysymykset on kuvattu epäselvästi. Yin (2009) muistuttaa, että reliabiliteetin arvioimista on saman tapauksen uudelleentoteutus, ei samankaltaisten tulosten saaminen toisessa tapauksessa. Tutkimuksen realibiteetti voidaan varmistaa riittävällä dokumentaatiolla (Yin, 2009). Yin (2009) painottaa selkeiden askelien esittämistä, joiden avulla tutkimus on mahdollista toistaa.

Tässä työssä on pyritty selkeästi kertomaan, mitä missäkin vaiheessa on tehty ja mihin johtopäätöksiin tehdyt valinnat perustuvat. Realibiteetin toteutumista edesautetaan tutkimusmenetelmän, -prosessin ja -kohteen selkeällä kuvaamisella. Tutkimuksen lähtökohtana ollut arkkitehtuuri on kuvattu siten, että sen toimintaperiaate on helposti ymmärrettävissä. Arkkitehtuurin ongelmia on selvitetty, jotta lukija ymmärtää syyt, mitkä ovat johtaneet uuden arkkitehtuurin tarpeeseen. Tuloksena saatu arkkitehtuuri on esitetty lukijalle ja siihen johtaneet ratkaisut on kuvattu tutkielmassa.

Realibiteetin valossa ei ole kiistatonta, että toinen tutkija astuessaan samaan organisaatioon ja tapaukseen päätyisi samanlaiseen arkkitehtuuriratkaisuun ja havaintoihin kuin tässä tutkimuksessa. Tähän vaikuttavat tutkijan kokemus ohjelmistokehityksestä, kohdeorganisaation tuntemus ja valittu menetelmä arkkitehtuurin toteuttamiseksi. Arkkitehtuurin suunnittelu on tuonut kohdeorganisaation työntekijöille tietoa nykyisestä arkkitehtuurista ja pakottanut heitä ajattelemaan asioita arkkitehtuurin kannalta. Tästä syystä he saattavat vaatimuksia määrittäessään ja arkkitehtuuria arvioidessaan päätyä toisiin ratkaisuihin, mikäli prosessi suoritettaisiin uudestaan. Tutkimuksen reliabiliteettiin voi vaikuttaa tiedon tulkitseminen tutkimusasetelmaan sopivalla tavalla. Tutkija saattaa huomaamattaan johdatella haastateltavia sanomaan haluamiaan asioita. Esimerkiksi arkkitehtuurin arvioinnin yhteydessä haastattelijan tekemää manipulointia on pyritty välttämään siten, että arkkitehtuuri on lähetetty

haastateltavalle etukäteen ja häntä on pyydetty tekemään siitä havaintoja ennalta.

Tutkimuksen validiteetti tarkoittaa tutkimuksen tulosten uskottavuutta (*trustworthiness*), eli missä määrin tulokset ovat tosia ja riippumattomia tutkijan subjektiivista näkemyksistä (Runeson & Höst, 2009, s. 153). Validiteettia voidaan tarkastella monesta näkökulmasta. Yin (2009) on koonnut kirjallisuudesta näkökulmia, joiden avulla tulosten validiteettia voidaan arvioida. Näitä on käytetty empiirisessä sosiaalitieteessä, mutta hänen mukaansa ne sopivat myös tapaustutkimuksiin. Validiteetin arvioinnin mittarit ovat rakennevaliditeetti, sisäinen validiteetti ja ulkoinen validiteetti. Runeson ja Höst (2009) suosittavat samojen näkökulmien käyttämistä ohjelmistotuotannon tutkimuksen validiteetin arvioinnissa.

Rakennevaliditeetti tarkoittaa, kattavatko valitut mittarit tutkittavan kohteen riittävällä tarkkuudella (Yin, 2009; Järvinen & Järvinen, 2011). Runeson ja Höst (2009) antavat esimerkin tilanteesta, jossa rakennevaliditeetti on vaarassa. Tilanne voi syntyä esimerkiksi silloin, kun haastateltava ja haastattelija tulkitsevat asioita eri tavalla. Syynä erilaisiin tulkintoihin voi olla henkilöiden erilainen tausta, koulutustaso tai muut tekijät, jotka tässä tapauksessa häiritsevät rakennevaliditeettia. Yinin (2009) mukaan rakennevaliditeetti voidaan varmistaa luomalla todisteketjuja ja käyttämällä useita eri tietolähteitä johtopäätöksien tukena.

Tässä tutkimuksessa haastatteluihin valittiin henkilöitä kohdeorganisaation suosituksien perusteella. Suoria haastatteluja tehtiin ainoastaan yhdelle kohdeorganisaation asiakkaalle, mutta asiakasvaatimuksia kerättiin lisäksi henkilöiltä, jotka ovat tiiviissä yhteydessä asiakkaisiin. Kohdeorganisaatiosta haastatteluihin voitiin valita henkilöt tutkijan näkemyksen perusteella. Haastateltavat henkilöt olivat johto, tuotteen myyjä, asiakastukihenkilöt ja ohjelmistokehittäjät. Tältä osin kaikki sidosryhmät kohdeorganisaation sisältä tulivat edustetuksi. Rakennevaliditeetin näkökulmasta asiakkaiden ja loppukäyttäjien parempi huomioonottaminen olisi parantanut tulosten luotettavuutta. Nyt asiakkaiden ja loppukäyttäjien tarpeet ja vaatimukset kerättiin yhdeltä asiakkaalta sekä asiakastukihenkilöiltä. Tutkijan vaikutusta haastateltavien mielipiteisiin (*reactive bias*) ei tutkimuksen yhteydessä havaittu. Tutkija on toiminut yrityksessä aiemmin ja oli kaikille haastateltaville jo ennalta tuttu. Tästä syystä haastateltavilla ei ollut tarvetta asioiden (huomaamattomalle) vääristelylle tai kaunistelulle.

Tässä tutkimuksessa uutta ja vanhaa arkkitehtuuria arvioitiin Reekien ym. (2006) laadullisten ominaisuuksien luokittelun perusteella sekä haastatteluista kerättyjen skenaarioiden perusteella. Skenaariot liittyivät selkeästi laadullisiin ominaisuuksiin. Tämän takia haastateltavilta kerätyt, arkkitehtuuria testaavat skenaariot käsiteltiin laadullisia ominaisuuksia arvioitaessa. Näin arkkitehtuurin arvioinnissa käytettiin kahta mittaria: 1) tutkijan ja haastateltavien havaintoja arkkitehtuurien laadullisista ominaisuuksista sekä 2) arkkitehtuurin toimivuutta eri skenaarioissa. Koska laadulliset ominaisuudet on kirjallisuudessa

laajasti tunnistettu arkkitehtuuriin laatuun vaikuttaviksi asioiksi, voidaan tutkielman rakennevaliditeetin olevan tältä osin kunnossa.

Sisäinen validiteetti arvioi kausaalisuhteiden luotettavuutta (Yin, 2009). Esimerkiksi tutkijan havaitessa kahden muuttujan välisen yhteyden on olemassa riski, että muuttujien välissä on kolmas huomaamaton muuttuja, eikä muodostettu kausaalisuhde ole näin luotettava (Runeson & Höst, 2009). Yinin (2009) mukaan sisäisen validiteetin arviointi ei sovi kuvailevaan tutkimukseen, vaan se on tarkoitettu tutkimuksiin, jotka tutkivat kausaalisuhteita (vrt. selittävä tutkimus). Tästä syystä rakennevaliditeetin arviointi ei relevanttia tässä yhteydessä.

Ulkoisella validiteetilla tarkoitetaan tutkimustulosten yleistettävyyttä. Kyse on siis siitä, saadaanko samanlaisella tutkimusasetelmalla vastaavassa tilanteessa samanlaisia tuloksia. Tutkimuksessa esimerkiksi otoskoon lisääminen ei lisää validiteettiä, vaan reliabiliteettia. Syynä tähän on muuttunut otoksenottamisprosessi, jolloin tutkimukset eivät ole enää keskenään vertailtavia ulkoisen validiteetin näkökulmasta. (Järvinen & Järvinen, 2011.)

Tässä tapauksessa ulkoisen validiteetin näkökulmasta voidaan pohtia, minkälaisessa tilanteessa tutkielman tulokset olisivat valideja. Tässä tutkimuksessa tutkimuskohteena on ollut yksi arkkitehtuurin suunnitteluprosessi, joten lähtökohtaisesti tuloksia on vaikeaa toistaa samoin tuloksin missään muussa tilanteessa. Yrityksen historia, arkkitehtuuri ja kehityskulku sekä tutkimuksen tilannetekijät ovat uniikkeja, joten vastaavanlaista kontekstia on vaikea löytää. Toisaalta tässä tutkimuksessa sovelletuista toimintatavoista, esitetyistä perusteista ja saaduista tuloksista on mahdollisuus oppia ja käyttää näitä tietoja hyödyksi muissa, vastaavan kaltaisissa tilanteissa kulloisenkin harkinnan mukaisesti

Yhteenvetona voidaan todeta, että vaikka tässä tutkimuksessa voidaan tunnistaa tulosten reliabiliteettia ja validiteettia koskevia uhkia, on näitä uhkia onnistuttu pienentämään monenlaisin toimin. Tutkimuksen ratkaisujen perusteiden, kulun ja tulosten perusteellinen raportointi edesauttaa reliabiliteetin saavuttamista. Arkkitehtuurin suunnittelun lähtökohtien ja tuloksen arvioiden esille saamiseksi on käytetty kohtalaisen kattavaa joukkoa sidosryhmien edustajia. Tutkijan toimiminen sekä suunnitteluprosessin toteuttajan että tutkijan rooleissa heikentää tutkimuksen johtopäätösten luotettavuutta, mutta on vähentänyt väärinymmärryksen mahdollisuuksia ja parantanut luodun arkkitehtuurin kykyä vastata kohdeorganisaation tarpeeseen. Tutkimustulosten yleistettävyys on muiden tapaustutkimusten tapaan hyvin rajoittunutta. Puutteista huolimatta tutkimusta voidaan pitää tapaustutkimukseksi varsin luotettavana.

7 YHTEENVETO

Tämän tutkimuksen tarkoituksena oli selvittää, millä tavalla voidaan valita ja soveltaa ohjelmistoarkkitehtuurin suunnittelumenetelmää ja arvioida tuloksena saatua ohjelmistoarkkitehtuuria.

Ohjelmistoarkkitehtuuria, sen suunnittelua ja arviointia varten hankittiin tietoa kirjallisuuskatsauksen avulla. Kirjallisuuskatsauksella rakennettiin käsitteellistä perustaa ohjelmistoarkkitehtuurille, sen suunnittelulle ja arvioinnille. Tutkimuksessa kuvattiin ohjelmistoarkkitehtuuria, sen tavoitteita ja kuvaamistapoja. Lisäksi esiteltiin kuusi arkkitehtuurityyliä (kerrosarkkitehtuurit, tietovuoarkkitehtuurit, asiakas-palvelin-arkkitehtuurit, viestinvälitysarkkitehtuurit, malli-näkymä-ohjain-arkkitehtuurit ja tulkkipohjaiset arkkitehtuurit) sekä niiden hyötyjä ja haittoja. Tämän jälkeen esiteltiin ohjelmistoarkkitehtuurin suunnittelua ja arviointia sekä menetelmiä niiden suorittamiseen. Suunnittelua ja arviointia käsiteltiin myös ketterän kehityksen kontekstissa.

Tutkimuksen empiirisessä osassa toteutettiin tapaustutkimus, jossa kohdeorganisaation nykyisen ohjelmistoratkaisun pohjalta rakennettiin uusi arkkitehtuuri ominaisuusvetoista arkkitehtuurisuunnittelumenetelmää (Attribute-Driven Design, ADD) hyväksikäyttäen. Tapaustutkimuksen kohteena oli tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmän arkkitehtuurin suunnittelu. Ohjelmiston ja anturipohjaisen ratkaisun avulla kohdeorganisaation asiakkaat voivat suorittaa kunnonvalvontaa tuotantokoneille ja seurata tuotantonsa tehokkuutta ja tilaa. Aiempaan arkkitehtuurin verrattuna tuotettu arkkitehtuuri sallii paremman skaalautuvuuden, kehittämistyön helpottumisen ja varmuuskopioinnin parantumisen. Uusi arkkitehtuuri skaalautuu siis paremmin sekä nykyisiin että tuleviin tarpeisiin. Tuotettua arkkitehtuuria arvioitiin ATAM-arviointimenetelmän avulla, ja sitä verrattiin vanhaan arkkitehtuurin laadullisten ominaisuuksien suhteen.

Ominaisuusvetoinen arkkitehtuurisuunnittelumenetelmä todettiin sopivaksi menetelmäksi pienen mittakaavan ohjelmistokehitysprojektiin. Menetelmä tarjosi selkeät työvaiheet, joita seuraamalla on mahdollista suunnitella järjestelmän arkkitehtuuri. Ominaisuusvetoisen arkkitehtuurisuunnittelumenetelmän, kuten myös muidenkin vastaavien menetelmien, käyttäminen vaatii

kokemusta ja taustatietoa arkkitehtuurityyleistä, ohjelmistokehyksistä, sekä muista erilaisista tekniikoista ja taktiikoista. Suunnittelumenetelmät ovat kohdennettu suuren mittakaavan toimintavarmuutta vaativien ohjelmistojen suunnitteluun, mutta niitä voidaan hyödyntää myös muissa yhteyksissä. Suunnittelumenetelmien hyödyt tulevat kuitenkin selkeämmin esille suuremmissa projekteissa. Tutkimuksen perusteella päättelyketjut kokonaisvaatimuksista yksittäisen elementin vaatimuksiin syntyvät selkeästi etenkin suurempia järjestelmiä kuvatessa, mutta tällöin myös arkkitehtuurikuvauksen koko kasvaa. Suuren arkkitehtuuriprosessin toteuttaminen on menetelmän dokumentointiavaatimuksien takia raskasta. Pienemmissä, muutaman iteraatiokierroksen, arkkitehtuuritoteutuksissa päättelyketjut ovat lyhyempiä ja selkeästi nähtävillä ilman mittavaa dokumentaatiota.

Tuotetun arkkitehtuurin avulla kohdeorganisaatio voi arvioida, onko ohjelmistokehityksen pääpainon siirtäminen web-teknologioiden päälle järkevää. Tuotettu arkkitehtuuri toimii johdon työkaluna ja suunnittelun tukena, mikäli tässä tutkielmassa tuotetut arkkitehtuuri valitaan ohjelmistokehityksen pohjaksi. Tämän lisäksi tässä tutkimuksessa tuotettu arkkitehtuuri voi palvella laajemminkin teollisen internetin sovelluksia, jonka erikoistapaus tässä tutkimuksessa kuvattu tuotantotehokkuuden seuranta- ja kunnonvalvontajärjestelmä on. Arkkitehtuurin lisäksi tutkielmassa luotu tutkimusprosessi ja tutkimusmalli ovat hyödynnettävissä muissa tutkimuksissa, joissa tarkoituksena on selvittää ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmien toimintaa.

Suunnittelumenetelmän valinta tehtiin tässä tutkimuksessa nojautuen muutamaan tutkimukseen. Tämän perusteella on haastavaa todeta, onko vertailuun valitut suunnittelumenetelmät sopivimmat juuri tähän tapaukseen. Tässä tutkimuksessa kuvattiin ainoastaan järjestelmän kokonaisarkkitehtuuri. Tämä voi rajoittaa tutkimuksesta saatavia havaintoja. Tutkimuksessa ei voida täysin luotettavasti arvioida tuotetun arkkitehtuurin toimivuutta kohdeongelmassa, sillä arkkitehtuurin pohjalta suunniteltua ohjelmistoa ei ole vielä toteutettu.

Jatkotutkimuksissa arkkitehtuurin suunnittelua voitaisiin tehdä järjestelmälle, jonka arkkitehtuuria halutaan kuvata myös alemmilla tasoilla. Lisäksi arkkitehtuurin toimivuutta voitaisiin arvioida sen jälkeen, kun sen pohjalta toteutettu ohjelmisto on otettu käyttöön. Samalle ohjelmistolle voitaisiin tehdä useampi arkkitehtuurikuvaus käyttämällä eri suunnittelumenetelmiä, jolloin saataisiin konkreettista vertailutietoa suunnittelumenetelmien toiminnasta. Käsitteellis-teoreettista tutkimusta ohjelmistoarkkitehtuureista, niiden suunnittelusta ja arvioinnista on tehty varsin paljon. Sen sijaan empiiristä tutkimusta suunnittelumenetelmien soveltamisesta käytännössä on selvästi vähemmän. Kattavia tapaustutkimuksia ohjelmistoarkkitehtuurin suunnittelu- ja arviointimenetelmien käytöstä ketterässä ohjelmistokehityksessä on tämän tutkimuksen lisäksi vähän. Sopivia jatkotutkimusaiheita voivat olla näiden suunnittelusuuntautuneiden arkkitehtuurin suunnittelu- ja arviointimenetelmien sovittaminen ja tiukempi integrointi ketterään ohjelmistokehitykseen. Erityisen tärkeää tämänkaltaisessa jatkotutkimuksessa olisi kuvata menetelmät ja työvaiheet yksinkertaisesti ja selkeästi, jotta pienikin organisaatio voi hyödyntää niitä.

LÄHTEET

Abrahamsson, P., Babar, M. A. & Kruchten, P. (2010). Agility and Architecture: Can they Coexist? *Software, IEEE* 27 (2), 16-22.

Akbari, F. & Sharafi, S. M. (2012). A Review to the Usage of Concepts of Software Architecture in Agile Methods. Teoksessa *Instrumentation & Measurement, Sensor Network and Automation (IMSNA), 2012 International Symposium on* (s. 389-392). IEEE.

Albin, S. T. (2003). *The Art of Software Architecture: Design Methods and Techniques.* Indianapolis: John Wiley & Sons.

America, P., Obbink, H., van Ommering, R. & van der Linden, F. 2000. COPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering. Teoksessa *Software Product Lines.* New York: Springer, 167-180.

Anderson, C. (2012). *Pro Business Applications with Silverlight 5.* New York: Springer.

Babar, M. A., Zhu, L. & Jeffery, R. (2004). A Framework for Classifying and Comparing Software Architecture Evaluation Methods. Teoksessa *Software Engineering Conference, 2004. Proceedings. 2004 Australian* (s. 309-318). Los Alamitos: IEEE Computer Society.

Bachmann, F. & Bass, L. (2001). Introduction to the Attribute Driven Design Method. Teoksessa *Proceedings of the 23rd International Conference on Software Engineering* (s. 745-746). Toronto: IEEE Computer Society.

Bachmann, F., Bass, L., Carriere, J., Clements, P. C., Garlan, D., Ivers, J., Nord, R. & Little, R. (2000). *Software Architecture Documentation in Practice: Documenting Architectural Layers* (3-2000). Pittsburgh: Carnegie Mellon University.

Barbacci, M. R., Ellison, R. J., Weinstock, C. B. & Wood, W. G. (2003a). *Quality Attribute Workshops (QAWs)* (CMU/SEI-2003-TR-016). Pittsburgh: Software Engeering Institute.

Barbacci, M., Clements, P. C., Lattanze, A., Northrop, L. & Wood, W. (2003b). *Using the Architecture Tradeoff Analysis Method (ATAM) to Evaluate the Software Architecture for a Product Line of Avionics systems: A Case Study* (CMU/SEI-2003-TN- 012). Pittsburgh: Software Engineering Institute.

Bass, L., Clements, P. & Kazman, R. (2003). *Software Architecture in Practice.* (2. painos). Pittsburgh: Addison Wesley.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Boston: Addison-Wesley Professional.

Bengtsson, P. & Bosch, J. (1999). Architecture Level Prediction of Software Maintenance. Teoksessa *Proceedings of the Third European Conference on Software Maintenance and Reengineering,* (s. 139-147). Washington: IEEE.

Bengtsson, P. & Bosch, J. (1998). Scenario-based Software Architecture Reengineering. Teoksessa *Proceedings. Fifth International Conference on Software Reuse* (s. 308-317). Los Amitos: IEEE.

Bengtsson, P., Lassing, N., Bosch, J. & van Vliet, H. (2004). Architecture-level Modifiability Analysis (ALMA). *Journal of Systems and Software* 69 (1), 129-147.

Booch, G., Rumbaugh, J. & Jacobson, I. (1996). The Unified Modeling Language. *Unix Review* 14 (13), 5-35.

Bosch, J. (2000). *Design and use of Software Architectures: Adopting and Evolving a Product-line Approach.* Great-Britain: Pearson Education.

Buschmann, F., Meunier, R., Rohnert, H. & Stal, M. (1995). *Pattern-Oriented Software Architecture - A Pattern System.* England: Wiley.

Cervantes, H., Velasco-Elizondo, P. & Kazman, R. (2013). A Principled Way to Use Frameworks in Architecture Design. *Software, IEEE* 30 (2), 46-53.

Clements, P. C. (2000). *Active Reviews for Intermediate Designs* (CMU/SIE-2000-TN-009). Pittsburg: Software Engineering Institute.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J. & Little, R. (2002). *Documenting Software Architectures: Views and Beyond.* Boston: Pearson Education.

Clements, P., Kazman, R. & Klein, M. (2003). *Evaluating Software Architectures.* Indiana: Pearson Education, Inc.

Dijkstra, E. W. (1968). The Structure of the "THE" Multiprogramming System. *Communications of the ACM* 11 (5), 341-346.

Durdik, Z. (2011). Towards a Process for Architectural Modelling in Agile Software Development. Teoksessa *Proceedings of the Joint ACM SIGSOFT Conference--QoSA and ACM SIGSOFT Symposium--ISARCS on Quality of Software Architectures--QoSA and Architecting Critical Systems--ISARCS* (s. 183-192). New York: ACM.

Eloranta, V. & Koskimies, K. (2011). *Sulava Scrum Survey Report* (978-952-15-2661-9). Tampere: Tampereen teknillinen yliopisto.

Faber, R. (2010). Architects as Service Providers. *Software, IEEE* 27 (2), 33-40.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine.

Garlan, D. & Shaw, M. (1993). An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering* 1, 1-39.

Hadar, I. & Sherman, S. (2012). Agile vs. Plan-driven Perceptions of Software Architecture. Teoksessa *5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)* (s. 50-55). IEEE.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A. & America, P. (2007). A General Model of Software Architecture Design Derived from Five Industrial Approaches. *Journal of Systems and Software* 80 (1), 106-126.

International Organization for Standardization (2011). ISO/IEC 42010: IEEE Std 1471-2000: Systems and Software Engineering — Recommended Practice for Architectural Description of Software-Intensive Systems. Haettu 21.9.2014 osoitteesta: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45991.

International Organization for Standardization. (1996). ISO/IEC 7498-1:1994 - Information Technology -- Open Systems Interconnection -- Basic Refer-

ence Model: The Basic Model. Haettu 10.10.2014 osoitteesta: http://www.iso.org/iso/catalogue_detail.htm?csnumber=20269.

Isham, M. (2008). Agile Architecture is Possible You First Have to Believe! Teoksessa *Agile, 2008. AGILE'08. Conference* (s. 484-489). IEEE.

Jansen, A. G. J. (2008). *Architectural Design Decisions*. University of Groningen.

Järvinen, P. & Järvinen, A. (2011). *Tutkimustyön metodeista.* Tampere: Opinpajan Kirja.

Jazayeri, M. (2007). Some Trends in Web Application Development. Teoksessa *Proceedings of Future of Software Engineering, (FOSE'07)* (s. 199-213). IEEE.

Kalja, A., Reitsakas, A. & Saard, N. (2005). eGovernment in Estonia: Best Practices. *Technology Management: A Unifying Discipline for Melting the Boundaries* , 500-506.

Kazman, R., Bass, L., Webb, M. & Abowd, G. (1994). SAAM: A Method for Analyzing the Properties of Software Architectures. Teoksessa *Proceedings of the 16th International Conference on Software Engineering* (s. 81-90). Los Alamitos: IEEE Computer Society Press.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. & Carriere, J. (1998). The Architecture Tradeoff Analysis Method. Teoksessa *Proceedings on Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98).* (s. 68-78). Monterey: IEEE.

Keuler, T., Wagner, S. & Winkler, B. (2012). Architecture-aware Programming in Agile Environments. Teoksessa *Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*(s. 229-233). IEEE.

Koskimies, K. & Mikkonen, T. (2005). *Ohjelmistoarkkitehtuurit.* Tampere: Talentum Oy.

Kruchten, P. (2010). Software Architecture and Agile Software Development: a Clash of two Cultures? Teoksessa *ACM/IEEE 32nd International Conference on Software Engineering* (s. 497-498). New York: IEEE.

Kruchten, P. (2004). *The Rational Unified Process: an Introduction.* Boston: Pearson Education.

Kruchten, P. B. (1995). The 4+1 View Model of Architecture. *Software, IEEE* 12 (6), 42-50.

Kruchten, P., Obbink, H. & Stafford, J. (2006). The Past, Present, and Future for Software Architecture. *Software, IEEE* 23 (2), 22-30.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1999). On Software Architecture Analysis of Flexibility, Complexity of Changes: Size isn't Everything. Teoksessa *Proceedings of Second Nordic Software Architecture Workshop NOSA* (s. 1103-1581). Blekinge Institute of Technology.

Lung, C., Bot, S., Kalaichelvan, K. & Kazman, R. (1997). An Approach to Software Architecture Analysis for Evolution and Reusability. Teoksessa *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research* (s. 15-26). Toronto: IBM Press.

Madison, J. (2010). Agile Architecture Interactions. *Software, IEEE* 27 (2), 41-48.

Molter, G. (1999). Integrating SAAM in Domain-centric and Reuse-based Development Processes. Teoksessa *Proceedings of the 2nd Nordic Workshop on Software Architecture* (s. 1-10). Ronneby: Citeseer.

Nord, R. L. & Tomayko, J. E. (2006). Software Architecture-centric Methods and Agile Development. *Software, IEEE* 23 (2), 47-53.

Potel, M. (1996). *MVP: Model-View-Presenter the Taligent Programming Model for c and Java.* Nevada City: Taligent Inc.

Ran, A. (2000). ARES Conceptual Framework for Software Architecture. Teoksessa M. Jazayeri, A. Ran & F. van der Linden (toim.) *Software Architecture for Product Families Principles and Practice.*(s. 1-29). Boston: Addison Wesley.

Reekie, J., McAdam, R. & McAdam, R. (2006). *A Software Architecture Primer.* Sydney: Angophora Press.

Runeson, P. & Höst, M. (2009). Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14 (2), 131-164.

SAE Technical Standards Board (2004). *Architecture Analysis & Design Language (AADL)* (AS5506). SAE International.

Schwaber, K. & Beedle, M. (2002). *Agile Software Development with Scrum.* Upper Saddle River: Prentice Hall.

Schwaber, K. & Sutherland, J. (2013). The Scrum Guide™ - The Definitive Guide to Scrum: The Rules of the Game. Haettu 27.10.2014 osoitteesta: http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf#zoom=100.

Shaw, M. & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline.* New York: Prentice Hall Englewood Cliffs.

Soni, D., Nord, R. L. & Hofmeister, C. (1995). Software Architecture in Industrial Applications. Teoksessa Proceedings of *17th International Conference on Software Engineering (ICSE 1995)* (s. 196-196). Washington: IEEE.

SysML Merge Team (2006). *Systems Modeling Language (SysML) Specification* (ad/2006-03-01). SysML Open Source Specification Project.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R. & Wood, B. (2006). *Attribute-Driven Design (ADD), Version 2.0* (CMU/SEI-2006-TR-023). Pittsburgh: DTIC Document.

Wood, W. G. (2007). *A Practical Example of Applying Attribute-Driven Design (ADD),Version 2.0* (CMU/SEI-2007-TR-005). Pittsburgh: Carnegie Mellon University.

Yin, R. K. (2009). *Case Study Research: Design and methods.* Beverly Hills CA: Sage Publications.

LIITE 1 HAASTATTELURUNKO

Haastattelurunko, jota käytettiin ohjelmistokehittäjien ja asiakastuen haastatteluihin vaatimusten ja nykyisen arkkitehtuurin ongelmien selvittämiseen.

1. Tunnistatko joitakin tarpeita mitä ei ole pystytty toteuttamaan nykyisellä ohjelmistoarkkitehtuurilla?
2. Mitä pyyntöjä asiakkailta on tullut, joita ei ole vielä toteutettu, tai joita ei ole pystytty toteuttamaan nykyisellä ohjelmistolla?
3. Kerro näkemyksesi siitä, mikä on asiakkaille nykyisen järjestelmän käytössä oleellisinta?
4. Mitä hyötyjä asiakas järjestelmän avulla saa?
5. Mikä on näkemyksesi siitä, ovatko asiakkaat valmiita luovuttamaan järjestelmän datan keskitettyyn pilvipalveluun?
6. Tuleeko mieleesi muita arkkitehtuuriin liittyviä hyötyjä ja haasteita, joita ei tämän haastattelun yhteydessä ole tullut esille?