KOMPONENTTIPOHJAISEEN OHJELMISTOARKKITEHTUURIIN SIIRTYMINEN

Kimmo Pylvänäinen

Opinnäytetyö
Maaliskuu 2010

Liiketalous
Luonnontieteiden ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) PYLVÄNÄINEN, Kimmo	Julkaisun laji Opinnäytetyö	Päivämäärä 07.03.2010
	Sivumäärä 40	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
KOMPONENTTIPOHJAISEEN OHJELMISTOARKKITEHTUURIIN SIIRTYMINEN

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
KARHULAHTI, Mika

Toimeksiantaja(t)
Yritys X

Tiivistelmä

Työ käsittelee ohjelmistoarkkitehtuurin merkitystä ohjelmistosuunnittelussa. Työssä selvitettiin, mitä on ohjelmistoarkkitehtuuri, ja miksi se on merkittävässä asemassa ohjelmistoprojektien suunnittelussa.

Teoriaosuudessa käytiin laajasti läpi ohjelmistoarkkitehtuurin käsitettä. Tämän lisäksi käytiin läpi myös erilaisia ohjelmistoarkkitehtuurityylejä sekä niiden pääasiallisia käyttökohteita. Työssä esiteltiin erilaisia ohjelmistoarkkitehtuurityylejä, mutta pääpaino oli komponenttipohjaisella ohjelmistoarkkitehtuurilla. Aihetta käsiteltiin toimeksiantajayrityksen tarpeiden näkökulmasta. Tutkimuksessa selvitettiin, mitä hyötyjä oikeanlaisen arkkitehtuurin valinnasta seuraa, ja miten se helpottaa ohjelmistotuotantoa.

Tutkimusosuudessa selvitettiin ja käytiin läpi niitä kokemuksia, joita toimeksiantajayrityksessä tehtyjen pilottiprojektien pohjalta on saatu. Tarkemmassa käsittelyssä oli Ajastetut tehtävät -sovellus, jonka ohjelmistoarkkitehtuurimallia käytiin läpi ja esiteltiin käytännössä.

Tärkeä näkökulma tutkimuksessa oli se, miten ohjelmistotuotantoa voidaan tehostaa oikeanlaista ohjelmistoarkkitehtuuria käyttämällä.

Tutkimuksen tulokset osoittavat, miten oikeanlaisella ohjelmistoarkkitehtuurilla on mahdollista parantaa yrityksen ohjelmistokehityksen tuottavuutta ja laatua.

Avainsanat (asiasanat)
Ohjelmistoarkkitehtuuri, ohjelmistotuotanto, komponentti

Muut tiedot

Author(s)
PYLVÄNÄINEN, Kimmo

Type of publication
Bachelor´s Thesis

Date
07032010

Pages
40

Language
Finnish

Confidential

( ) Until

Permission for web publication
(X)

Title
MOVING TO COMPONENT-BASED DEVELOPMENT

Degree Programme
Business Information Systems

Tutor(s)
KARHULAHTI, Mika

Assigned by
Company X

Abstract

The purpose of this study was to find out how choosing the right software architecture can improve the process of software development. The assigner of this study was Company X, which has made a decision to move to the component-based software development in the near future.

The theoretical part of this thesis concentrates on different kind of software architecture models. The most important architecture for this study is component-based architecture. In order to understanding the assigner's decision to change the whole way of its software development, it is essential to be aware the meaning of software architecture.

The empirical part of this study concentrates mainly on the pilot project that has already been made using the new architecture model. There are some conclusions that have been made about the new model. Also the whole new architecture model is described in every detail.

The aim of this thesis is to find out how proper software architecture design can help in software projects. It can be the key factor to make software as productive as possible.

Keywords
Component-based Development, Component, Software Architecture

Miscellaneous

SISÄLTÖ

KUVIOT

1 JOHDANTO

Opinnäytetyö käsittelee komponenttipohjaista ohjelmistoarkkitehtuuria. Tutkimuksen tavoitteena on selvittää, mitä hyötyjä komponenttiarkkitehtuurista on pienissä ja keskisuurissa ohjelmistoalan yrityksissä. Opinnäytetyössä verrataan komponenttiarkkitehtuuria vanhaan keskitettyyn, kolossaaliseen arkkitehtuuriin. Vertailemalla näitä eri toteutustapoja pyritään muodostamaan selkeä kuva komponenttimallin eduista ja hyödyistä.

Opinnäytetyön toimeksiantajana toimii keskisuomalainen ohjelmistoalan pk-yritys. Suoritin yrityksessä työharjoittelujakson, joten olen perillä yrityksen työskentely- ja toimintatavoista. Yrityksessä on käynnissä teknologian vaihto, jossa tarkoituksena on siirtyä käyttämään komponenttipohjaista ohjelmistoarkkitehtuuria. Tämän avulla pyritään pääsemään eroon vaikeasti päivitettävistä tuotteista. Komponenttiarkkitehtuurin avulla pyritään luomaan helposti hallittavia ja päivitettäviä, monikäyttöisiä ohjelmistokomponentteja. Näitä komponentteja pystytään käyttämään useassa yrityksen eri tuotteessa. Kertakäyttöisestä ohjelmistokoodista pyritään siirtymään monikäyttöisempään malliin. Tämän tutkimuksen avulla pyritään selvittämään, miten komponenttipohjainen ohjelmistoarkkitehtuuri soveltuu toimeksiantajayrityksen tarpeisiin. Nykyisessä toimintatavassa on noussut vastaan monia haasteita, jotka komponenttipohjaisella ohjelmistoarkkitehtuurilla pyritään ratkaisemaan. Tässä tutkimuksessa etsitään ratkaisua näihin ongelmiin ja pyritään katsomaan asioita komponenttipohjaisen ohjelmistoarkkitehtuurin lähtökohdista.

Toimeksiannon tavoitteena on luoda selkeä kuvaus komponenttiarkkitehtuurista niin teoreettisella tasolla kuin käytännön esimerkkeinäkin. Tämän tutkimuksen tuloksien avulla pystytään helposti osoittamaan yrityksen työntekijöille, niin nykyisille kuin tulevillekin, mitä komponenttipohjainen arkkitehtuuri tarkoittaa. Teorian lisäksi tavoitteena on käsitellä myös käytännön kokemuksia, joita komponenttiarkkitehtuurista on muodostunut teknologiavaihdon alkuvaiheiden perusteella. Opinnäytetyö toimii näin ollen perehdytysmateriaalina, jonka pohjalta muidenkin kuin yrityksen

ohjelmistoarkkitehdin ja pääsuunnittelijoiden on mahdollista saada helposti tietoa uudesta toimintatavasta.

2 TUTKIMUSASETELMA

Seuraavassa osiossa käsitellään toimeksiantajayrityksen tilannetta ja tutkimuksen tarpeita. Lisäksi käydään läpi tutkimuksessa käytettävää tutkimustapaa. Tämän jälkeen esitellään tutkimuksen rajaus ja tutkimuskysymykset. Luvun lopussa kerrotaan tutkimusmenetelmästä ja tutkimuksen tavoitteista.

2.1 Toimeksiantaja ja tutkimuksen rajaukset

Toimeksiantajana toimii keskisuomalainen ohjelmistoalan pk-yritys. Yrityksen päätuotteina ovat SaaS-mallin mukaan myytävät ohjelmistot. SaaS-lyhenne tulee sanoista Software as a Service. SaaS-ohjelmistot ovat siis ohjelmistoyrityksen itsensä kehittämiä ja myös ylläpitämiä. SaaS-ohjelmia myydään asiakkaalle palveluina, joiden käytöstä asiakkaat maksavat. Asiakkaiden ei tarvitse huolehtia ohjelmistojen asennuksesta, vaan niiden käyttö onnistuu ohjelmistoyrityksen ylläpitämille verkkosivuille kirjautumalla. Tämä helpottaakin monen asiakkaan päätöstä ohjelmiston hankkimisesta, koska varsinaista it-puolta ei yrityksessä välttämättä tarvita laisinkaan ohjelmiston käyttöönottoa varten. (CIO Software as a Service (SaaS) Definition and Solutions 2010.)

2.2 Tutkimuskysymykset

Tämän tutkimuksen aihe pyritään rajaamaan tarkasti, jotta tutkimuksesta olisi hyötyä toimeksiantajayritykselle. Tutkimuskysymykset on rajattu tarkasti etsimään vastauksia juuri toimeksiantajayritystä koskeviin aiheisiin. Tutkimuskysymysten tärkein funktio on olla suuntaviittana tutkimuksen kululle.

1. Mitä on komponenttiarkkitehtuuriin perustuva ohjelmistosuunnittelu?

2. Minkä kokoisissa ohjelmistoprojekteissa komponenttiarkkitehtuuria on järkevä käyttää?

3. Miten komponenttiarkkitehtuurin edut ilmenevät pk-yrityksessä verrattuna monoliittiseen arkkitehtuuriin?

2.3 Tutkimusmenetelmä

Opinnäytetyö tehdään kvalitatiivisen tutkimusmenetelmään perustuen. Kvalitatiivisen tutkimuksen perusteella on mahdollista tehdä kerätystä aineistosta tulkintoja ja yleistyksiä, joiden perusteella pystytään käsittelemään komponenttiarkkitehtuurin eri ominaisuuksia monipuolisesti. Tämän avulla on mahdollista tehdä havaintoja yritystä hyödyttävistä komponenttiarkkitehtuurin ominaisuuksista ja toimintatavoista.

2.4 Tutkimuksen tavoitteet

Opinnäytetyön tavoitteena on toteuttaa pk-yritykselle selvitys komponenttiarkkitehtuuriin siirtymisen hyödyistä, mahdollisista muutoksista työtapoihin sekä uusista haasteista. Konkreettisen hyödyn hahmottaminen on yksi tärkeistä tavoitteista. Komponenttimallin avulla pyritään saavuttamaan helpommin hallittavia ohjelmistokokonaisuuksia. Tutkimuksen tavoitteena on selkeästi osoittaa näitä hyötyjä tulevassa jokapäiväisessä työskentelyssä komponenttimalliin perustuvan ohjelmistoprojektin parissa.

3 OHJELMISTOARKKITEHTUURI

Tässä luvussa kerrotaan ohjelmistoarkkitehtuurista. Alussa kerrotaan ohjelmistoarkkitehtuurista yleisellä tasolla. Seuraavaksi käsitellään ohjelmistoarkkitehtuurin merkitystä. Yleisen ohjelmistoarkkitehtuurin selventämisen lisäksi luvussa käsitellään myös ohjelmistoarkkitehdin roolia ja sitä, mitä vastuita hänellä on läpi koko ohjelmistoprojektin elinkaaren ajan.

3.1 Yleistä ohjelmistoarkkitehtuurista

Ohjelmistoarkkitehtuuri tarkoittaa ohjelmiston rakennetta yleisellä tasolla. Ohjelmistoarkkitehtuurissa ei oteta kantaa ohjelmiston yksityiskohtaiseen toteutukseen. Päätarkoituksena on luoda laajat ääriviivat, joiden pohjalle ohjelmisto tulee rakentumaan. Kehitettäessä laajoja ohjelmistosovelluksia on tärkeää kiinnittää huomiota ohjelmiston rakenteen toteutukseen. Ohjelmisto on pyrittävä mallintamaan jo hyvin aikaisessa vaiheessa ohjelmistokehitystä. (Shaw 1996.)

Tässä vaiheessa ohjelmistonarkkitehtuurilla on erittäin suuri vaikutus tulevan ohjelmiston lopulliseen rakenteeseen. Ohjelmistoarkkitehtuurin avulla pystytään mallintamaan ohjelmiston rakenne. Suurien järjestelmien toteutuksessa varhainen kokonaisvaltainen mallintaminen nouseekin avainasemaan. Varhaisessa vaiheessa tapahtuva laajamittainen suunnittelu on erittäin tärkeää ohjelmiston tulevan elinkaaren kannalta. Oikein suunniteltu joustava arkkitehtuuri mahdollistaa ohjelmiston muuntumisen ja kasvamisen sen koko elinkaaren ajan. (Shaw 1996.)

3.2 Ohjelmistoarkkitehtuurin määritelmä

Yksiselitteistä määritelmää ohjelmistoarkkitehtuurille ei ole olemassa, mutta yksi tulkinta asiasta on seuraava: ohjelmiston arkkitehtuuri on koko järjestelmän rakenne, joka muodostuu komponenteista ja niiden ominaisuuksista sekä näiden komponenttien välisistä suhteista. Arkkitehtuurin komponenttikäsitettä ei pidä sekoittaa

ohjelmistokomponentin kanssa. Arkkitehtuurin määritelmän komponentti sisältää tiettyjä ominaisuuksia ja on osa suurempaa kokonaisuutta. Ohjelmistokomponentti-käsite kuvaa kokonaista ohjelmiston osaa, kokonaisuutta, joka voi sisältää useita eri komponentteja. (Shaw 1996.)

Jokaisella ohjelmistolla on olemassa jonkinlainen arkkitehtuuri, vaikka sitä ei välttämättä ole määritelty erikseen. Ohjelmiston rakenne voi olla niin yksinkertainen, että erilliseen arkkitehtuurin suunnitteluun ja kuvaamisen ei ole käytännössä tarvetta. Yksinkertaisimmillaan arkkitehtuuri koostuu ainoastaan yhdestä komponentista, joka sisältää erilaisia toimintoja. (Shaw 1996.)

Ohjelmistoarkkitehtuurin pääajatuksena on tehdä selkeä määritelmä siitä, miten suuri ohjelmistokokonaisuus koostuu useista pienemmistä osista. Pelkkä pienien osien erittely ei riitä, vaan on myös kuvattava, miten nämä eri osat liittyvät ja toimivat suhteessa toisiinsa. Erittäin suuret ohjelmistot pyritään jakamaan pienempiin komponentteihin, jotta suuri kokonaisuus on helpommin hahmotettavissa. Monitasoisesti toteutetussa arkkitehtuurissa pystytäänkin muuttamaan ohjelmiston tiettyjä osia ilman, että muutokset vaikuttavat kokonaisuuden rakenteeseen. (Shaw 1996.)

Ohjelmistoarkkitehtuuri ei ole sama asia kuin ohjelmiston suunnittelu. Ohjelmistoarkkitehtuuri on yksi erittäin tärkeä osa ohjelmiston suunnittelussa. Ohjelmistosuunnittelu on suurempi kokonaisuus, jossa ohjelmistoarkkitehtuurin osa on kuitenkin erittäin merkittävä. Ohjelmistosuunnittelussa käsitellään myös ohjelmiston yksityiskohtaista toteutusta konkreettisella tasolla, mutta ohjelmistoarkkitehtuurin tärkein päämäärä on jäsentää ohjelmisto abstraktilla tasolla selkeisiin osiin ja kokonaisuuksiin. (Shaw 1996.)

3.3 Ohjelmistoarkkitehtuurin merkittävimmät hyödyt

Ohjelmistoarkkitehtuurista on paljon hyötyä tulevan ohjelmistoprojektin onnistumisen kannalta. Bass ja muut esittävät kolme merkittävää syytä, jotka osoittavat kiistatta ohjelmistoarkkitehtuurin tärkeyden. Koko ohjelmistoprojektin onnistumisen kannalta tärkein ohjelmistoarkkitehtuurin ominaisuus on sen tarjoama tiedonvälitys. Tiedonvälityksellä tarkoitetaan arkkitehtuurin tuloksena syntyvää korkean tason abstraktiomallia tulevasta tietojärjestelmästä. Tämän mallin perusteella pystytään helposti perehdyttämään ohjelmistoprojektissa toimivat eri osapuolet tulevan järjestelmän rakenteeseen ja toimintaan. (Bass ym. 1998.)

Toinen tärkeä hyöty ohjelmistoarkkitehtuurissa on sen tarjoama suunnittelun linjaus. Arkkitehtuurissa määritelty linjaus jatkuu läpi koko projektisuunnitelman. Arkkitehtuurissa mallinnettua ohjelmistoprojektin rakennetta on mahdollista analysoida erittäin tarkkaan projektin alkupuolella. Tässä vaiheessa on mahdollista tehdä vielä myös muutoksia itse projektin rakenteeseen sekä miettiä erilaisia projektin toteutustapoja.

(Bass ym.1998.)

Kolmas merkittävä hyöty on kerran arkkitehtuurin avulla tehdyn abstraktiomallin uudelleen käyttäminen tulevissa projekteissa. Jos yrityksen tulevat ohjelmistot ovat jollain tasolla samankaltaisia jo tehtyjen kanssa, on erittäin todennäköistä, että kehitettyä arkkitehtuurimallia pystytään hyödyntämään suoraan tai kohtuullisen pienillä muutoksilla myös uusissa projekteissa.

Näiden hyötyjen perusteella pystytään yrityksessä tekemään muutaman pilottiprojektin jälkeen selkeitä linjanvetoja siitä, miten projekti etenee ja miten siinä käytettävä arkkitehtuurimalli rakentuu. Selkeästi hahmotettavan rakenteen avulla päästään eroon myös monista eri riskeistä. Ohjelmiston eri osien suunnittelun ja mallintamisen jälkeen on paljon helpompi tehdä yhteenveto siitä, mitkä osa-alueet on otettu huomioon ja mikä osa-alue on vielä suunnittelun osalta keskeneräinen. Tämän avulla pystytään välttämään suuri osa merkittävistä riskeistä, eikä mitään ikäviä yllätyksiä pääse syntymään missään ohjelmistoprojektin kehitysvaiheessa. (Bass ym. 1998.)

Ohjelmistoarkkitehtuurin hyödyt kyllä periaatteessa tiedetään yleisellä tasolla, mutta siltikään arkkitehtuuriin ei monissa ohjelmistoprojekteissa panosteta riittävän paljon. Oikean arkkitehtuurityylin valinnalla ja sen pohjalta rakennettavan abstraktiomallin perusteella vältyttäisiin monilta yleisiltä ohjelmistoprojekteissa vaanivilta sudenkuopilta. Ohjelmistoprojektien koko kasvaa jatkuvasti, ja tämän myötä myös niiden rakenne kasvaa monitasoiseksi ja monimutkaistuu. Voisikin sanoa, että kunnollinen ohjelmistoarkkitehtuurin suunnitteleminen on useiden yritysten elinehto, jotta ohjelmistoprojektit voidaan viedä kunnialla läpi.

3.4 Ohjelmistoarkkitehdin vastuut

Suurin vastuu ohjelmistoarkkitehtuurin suunnittelusta on ohjelmistoarkkitehdin harteilla. Ohjelmistoarkkitehdin asema on erittäin vastuullinen, eikä vastuu rajoitu pelkästään arkkitehtuurisuunnitelmaan, vaan vastuu on kannettava läpi koko ohjelmistoprojektin. Ohjelmistoarkkitehdin on oltava eräänlainen moniosaaja, jonka on pystyttävä ottamaan kantaa sekä perustelemaan ratkaisujaan monen eri tahon kanssa. On tiedettävä suunnittelupuolen lisäksi myös ohjelmiston teknisestä puolesta paljon, jotta ei tehdä tekniseltä puoleltaan mahdottomiksi osoittautuvia suunnitelmia.
(Bredemeyer 2000.)

Arkkitehdin vastuualue koostuu monesta eri osa-alueesta. Bredemeyerin mukaan arkkitehdin toimenkuva koostuu seuraavista osa-alueista: teknologia, liiketoimintastrategia, organisointi, konsultointi ja johtaminen. Tästä voikin tehdä selkeän johtopäätöksen, että arkkitehdin rooli vaatii paljon työkokemusta ohjelmistotuotannon eri osa-alueista. Arkkitehdin on pystyttävä myös pitämään esityksiä ja selittämään arkkitehtuurisuunnitelmiaan niin osakkeenomistajille ja hankkeen tukijoille kuin sovelluskehityspuolen henkilöstöllekin. Tämä vaatii asioiden läpikotaista ymmärtämistä erilaisten näkökulmien läpi katsottuna.

Arkkitehti toimii eräänlaisena puskurina suunnittelupuolen ja teknisen puolen välillä. Arkkitehdin merkittävimpänä tehtävänä on oikeanlaisen arkkitehtuurityylin valinta

projektin tarpeet huomioon ottaen. Tämä ei olekaan aivan yksinkertainen tehtävä, vaan se vaatii monen eri osa-alueen hallitsemista. Arkkitehdin on osattava tehdä korkean tason suunnittelua, minkä lisäksi on pyrittävä hahmottamaan myös niitä tekniikoita ja teknisiä yksityiskohtia, jotka mahdollistavat arkkitehtuurisuunnitelman käytännön toteuttamisen. (Bredemeyer 2000.)

Toimiakseen hyvin ohjelmistoarkkitehdin on oltava hyvin perillä myös ohjelmistotuotannon teknisestä puolesta. Bredemayerin ja Malanin mukaan ohjelmistoarkkitehti voi toimia myös yhtenä ongelmanratkaisijana yhdessä sovelluspuolen pääkehittäjien kanssa. Ohjelmistoarkkitehti voi kirjoittaa koodia, mutta ei mitä tahansa koodia. Tärkeintä olisi paneutua suuriin ongelmiin, jotka ovat ratkaisevia sovelluksen koko rakenteen kannalta. Kun arkkitehti on aika ajoin tekemisissä ohjelmistokoodin kanssa, hänellä säilyy parempi kosketus tekniseen puoleen, ja sen myötä arkkitehtuurisuunnitelmat pysyvät selkeämpinä teknisen puolen näkökulman kannalta. (Bredemeyer 2000.)

4 ARKKITEHTUURITYYLIT

Tässä luvussa selvitetään erilaisia ohjelmistoarkkitehtuurin tyylisuuntia. Luvussa keskitytään kuitenkin vain tutkimuksen kannalta tärkeimpiin tyylisuuntiin, joten läpileikkaus ei ulotu läheskään kaikkiin eri ohjelmistoarkkitehtuurityyleihin. Tämän jälkeen käydään yksityiskohtaisemmin läpi erilaisia ohjelmistoarkkitehtuurin tyylisuuntauksia. Pääasiallisena käsittelyn kohteena on komponenttipohjainen ohjelmistoarkkitehtuuri, mutta tähän liittyvät kiinteänä osana tutkimuksen kannalta myös kerroksittainen ohjelmistoarkkitehtuuri ja malli-näkymä-ohjain-arkkitehtuurityyli.

4.1 Yleistä arkkitehtuurityyleistä

Ohjelmistoarkkitehtuurit voidaan jakaa erilaisiin tyyleihin, joilla on huomattavia keskinäisiä eroja. Tässä tutkimuksessa käsitellään pääosin komponenttipohjaista arkkitehtuurityyliä. Ohjelmistoarkkitehdin on pystyttävä valitsemaan kulloiseenkin projektiin parhaiten sopiva arkkitehtuurityyli, jotta sovelluksen rakenne saadaan optimoitua vastaamaan projektin tarpeita. Arkkitehtuurityylien käyttöä pitää harkita käyttötarkoitukseen mukaan. Arkkitehtuurien rakenteellisten erojen vuoksi ne soveltuvat erilaisiin projekteihin. Jos ohjelmistoprojekti on kevyt eikä tämän vuoksi vaadi välttämättä kovin monitasoista rakennetta, on syytä valita jokin kevyt ohjelmistoarkkitehtuurityyli. Suurempien järjestelmien kanssa taas on syytä valita monitasoisempi ratkaisu, jotta sovellus saadaan jaettua useaan pienempään, helpommin hallittavaan kerrokseen.

4.2 Kerroksittainen ohjelmistoarkkitehtuurityyli

Tässä luvussa käydään tarkemmin läpi kerrosarkkitehtuurityyliä. Yleisesittelyn lisäksi kerrosarkkitehtuurista käydään läpi, mitä hyötyjä siitä on ja minkälaisiin ohjelmistoprojekteihin se soveltuu. Tämän lisäksi on myös hyvä tarkastella, mitä vaatimuksia ja rajoitteita kerrosarkkitehtuurityyli asettaa ohjelmistoprojektin suunnittelulle ja vaatimuksille.

4.2.1 Kerroksittaisen ohjelmistoarkkitehtuurin määritelmä

Kerroksittainen ohjelmistoarkkitehtuurityyli koostuu nimensä mukaisesti useasta eri komponentista, joita kutsutaan kerroksiksi (layer). Jokainen kerros muodostuu ainoastaan yhdestä komponentista. Bachmann ja muut määrittelevät, että kerroksittainen arkkitehtuurityyli, niin kuin kaikki muutkin arkkitehtuurityylit, kuvaa suuren ohjelmiston jakamista useaan pienempään osaan. Kerroksittaisessa ohjelmistoarkkitehtuurissa jokainen kerros mielletään itsenäiseksi virtuaalikoneeksi. Näiden virtuaalikoneiden pohjalta voidaan järjestelmä jakaa suppeampiin osiin, mikä helpottaa järjestelmän jäsentämistä. Tästä seuraa järjestelmän helpompi ymmärrettävyys. (Bachmann ym. 2000.)

Bachmannin ja muiden mukaan virtuaalikone koostuu joukosta ohjelmia tai ohjelman osia, jotka yhdessä tarjoavat palveluja. Näitä palveluja voivat muut ohjelmat käyttää tietämättä, miten palvelut on toteutettu. Jokaisella komponentilla voi olla tietty erikoisala, jonka perusteella se tarjoaa palvelua toiselle kerrokselle. Kerros voi toimia ainoastaan tietokantapalveluna, joka tarjoaa toisen kerroksen pyytämiä tietoja. Kun kerrosten välillä on selvät hierarkiat siitä, miten tieto liikkuu, se lisää huomattavasti myös koko ohjelmiston tietoturvallisuutta. Jos ohjelmisto on tietokantaan yhteydessä vain yhtä tarkasti määriteltyä reittiä pitkin, hyökkäyksille mahdollistavien aukkojen määrä on automaattisesti paljon pienempi kuin järjestelmässä, jossa kaikki eri komponentit toimivat yhdessä suuressa kaaoksessa. (Bachmann ym. 2000.)

4.2.2 Kerroksittainen arkkitehtuuri käytännössä

Eräs tunnetuimmista kerroksittaisen ohjelmistoarkkitehtuurin toteutuksista on 1980-luvulla kehitetty OSI-malli, Open System Interconnection Reference Model. OSI-malli kuvaa eri tiedonsiirtoprokollien keskinäisiä yhteyksiä. OSI-malli rakentuu seitsemästä eri kerroksesta. OSI-malli on kansainvälinen ISO:n, International Organization of Standardization, hyväksymä standardi, jonka perusteella verkkoprotokollien on toimittava. (Zimmermann 1980.)

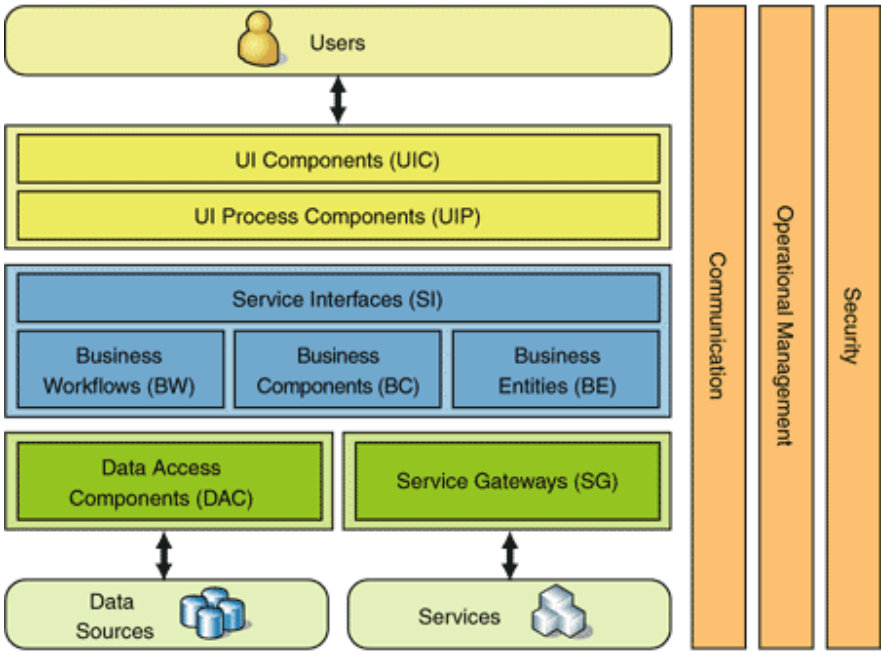
KUVIO 1. ISO:n OSI-malli (Zimmermann 1980)

Kuviosta näkee selvästi, että jokaisella kerroksella on oma selkeä paikkansa. Tieto siirtyy kerrokselta toiselle aina samaa reittiä pitkin. Jokaisella kerroksella on oma tärkeä roolinsa, jotta tieto kulkee nopeasti ja esteettömästi. Kerroksilla on kiinteä ja osin jäykkäkin hierarkia. Tieto siirtyy aina samaa reittiä kerrokselta toiselle hierarkian mukaan.

Kerroksittaisessa arkkitehtuurissa voi kerrosten välillä olla kahta erilaista määrittelyä säännöissä. Tiukempaa jaottelua kutsutaan täysin *puhtaaksi* (strict) kerroksittaisuudeksi. Kerroksittaisessa arkkitehtuurissa ohjelmisto on jaettu selkeisiin kerroksiin, jotka tarjoavat erilaisia palveluja toisilleen. Kerrosten välille on luotu tarkat säännöt, joiden

perusteella kerrosten välinen vuorovaikutus toimii. Tämä tarkoittaa sitä, että kerros N ja kerros N+1 toimivat yhteydessä toisiinsa ja välittävät palveluitaan. Kerros N+2 ei siis voi olla suoraan yhteydessä kerrokseen N, vaan se lähettää palvelukutsun kerrokselle N+1, joka taas välittää kutsun eteenpäin kerrokselle N. Kerros N palauttaa halutun palvelun kerrokselle N+1, josta se menee takaisin kerrokselle N+2. Esimerkkinä tiukasta kerroksellisuudesta toimii jo aikaisemmin esitelty OSI-malli. (Garlan & Shaw 1998.)

Toinen lähestymistapa kerroksittaiseen arkkitehtuuriin on *väljä* (relax) kerroksittaisuus. Väljä kerroksellisuus tarkoittaa, että yläpuolella oleva kerros pystyy pyytämään palveluita *kaikilta* sen alapuolella olevilta kerroksilta. Kerroksellisuus säilyy, mutta se ei ole ylhäältä alaspäin yhtä tarkasti määriteltyä kuin puhtaassa kerroksellisessa mallissa. Alapuolen kerrokset välittävät edelleen pyydetyn tiedon yläpuolen kerroksiin tarkassa järjestyksessä kerrokselta aina seuraavalle. (Microsoft MSDN Three layered services application, 2010)



KUVIO 2. Väljä 3-kerroksinen ohjelmistoarkkitehtuuri (Microsoft MSDN Three layered services application, 2010)

Kuviossa 2 on esitelty 3-kerroksinen ohjelmiston arkkitehtuuri. Ohjelmisto on jaettu kolmeen eri kerrokseen, ja keltaisella on merkitty esityskerros (Presentation layer), sinisellä toimintalogiikkakerros (Business layer) ja vihreällä tietokerros (Data layer). Esityskerros tarjoaa ohjelmiston käyttöliittymän (User Interface). Käyttöliittymä on käyttäjälle näkyvä osa. Esimerkiksi SaaS-ohjelmistossa se voi olla ohjelmiston selaimeen avautuva html-sivu, joka näkyy asiakkaan selaimessa. Tämä voidaan toteuttaa esimerkiksi ASP.NET- tekniikalla. ASP.NET on Microsoftin kehittämä web-sovelluskehys, joka perustuu .NET-ohjelmistokomponenttikirjastoihin. ASP.NET- tekniikalla toteutetut web-sivut rakentuvat luontevasti ASP.NET MVC -mallin mukaan, eli web-sivut noudattavat malli-näkymä-ohjain-arkkitehtuuria. (Microsoft ASP.NET Overview n.d.)

Toimintalogiikkakerrokseen sisältyvät kaikki ohjelmiston toiminnallisuudet. Tämä pitää sisällään esimerkiksi ohjelmiston toiminnalliset metodit. Nämä käskyt pyytävät haluamansa tiedon tietokerrokselta. Toimintalogiikkakerroksen voi toteuttaa esimerkiksi .NET- yhteensopivalla ohjelmointikielellä. .NET Framework on Microsoftin kehittämä ohjelmistokomponenttikirjasto, joka on yhteensopiva useiden eri ohjelmointikielien kanssa. .NET Frameworkia käytetään Microsoftin Visual Studio -sovelluskehitysympäristössä. (Microsoft .NET Framework Conceptual Overview n.d.)

Tietokerros säilöö ja tarjoaa muun sovelluksen pyyntöjä vastaavan tiedon muille kerroksille. Tietokerros on kiinteässä yhteydessä erillisiin tietokantoihin. Tietokerros voi koostua pienemmissä järjestelmissä tietokantojen sijasta esimerkiksi XML-toteutuksesta (eXtensible Markup Language). Tietokerroksen toteutuksessa voidaan käyttää esimerkiksi ADO.NET- tekniikkaa. (Microsoft ADO.NET n.d.)

ADO.NET on Microsoftin kehittämä luokkakirjasto, jonka pääasiallisena tehtävänä on toimittaa tietoa eri palveluille. ADO.NET:llä pystytään kehittämään palveluita, jotka toimittavat erilaista tietoa sitä tarvitseville palveluille. ADO.NET:n komponenttien avulla onnistuu tiedonhaku niin XML-aineistosta kuin myös tietokannoista. (Microsoft ADO.NET n.d.)

XML on merkintäkieli, jonka avulla tiedon merkitys pystytään kuvaamaan tiedon sekaan. XML ei siis ole ohjelmointikieli. XML:n avulla pystytään rakentamaan monikerroksisia tietorakenteita, joiden avulla sisältö saadaan yhdenmukaiseen tallennusmuotoon. XML:n avulla pystytään toteuttamaan erilaisten järjestelmien integraatioita helpommin sen yhteensopivuuden ansiosta. (W3C Extensible Markup Language n.d.)

4.2.3 Kerroksittaisen arkkitehtuurin edut

Kerroksittaisesta arkkitehtuurista on useita hyötyjä, jotka ovat monen järjestelmän rakenteen selkeyttämisen kannalta välttämättömiä. Kerroksellisuuden avulla pystytään jäsentämään erittäin monimutkainen ja laaja järjestelmä moneen pienempään kerrokseen. Tämän avulla pystytään hahmottamaan järjestelmän rakenne huomattavasti helpommin, koska kerrokset ovat selkeitä omia kokonaisuuksiaan. Garlan ja Shaw ilmaisevat tämän siten, että kerroksittainen arkkitehtuuri mahdollistaa suunnittelun jakamisen erinäisiin osioihin kasvavan abstraktiotason mukaan. (Garlan & Shaw 1998.)

Kerroksittainen arkkitehtuuri mahdollistaa hyvän uudelleenkäytettävyyden. Itse arkkitehtuuri ei ota kantaa siihen, millä tekniikalla järjestelmä toteutetaan. Siitä huolimatta hyöty tulee esille siinä, että jokainen kerros voidaan toteuttaa sille tarpeelliseksi todetulla tekniikalla. Koska kerrokset toimivat itsenäisesti, niin muutosten tapahtuessa joudutaan pääosin korjaamaan vain yhtä tiettyä kerrosta, vaikka muutokset olisivat suuriakin. Esimerkiksi jonkin kerroksen toteutustekniikan vaihtaminen onnistuu ilman, että se aiheuttaa muutoksia muualle kuin muutettavaan kerrokseen. Kerroksittainen arkkitehtuuri on vahvasti kytköksissä nykyisiin monimutkaisiin ohjelmistorakenteisiin. Joissain tapauksissa se on määritelty tarkemmin kuin toisissa, mutta silti se on laajalti käytössä.

4.3 Malli-näkymä-ohjain-arkkitehtuurityyli

Tässä luvussa käsitellään malli-näkymä-ohjain-ohjelmistoarkkitehtuuria samaan tapaan kuin aikaisemmassa luvussa kerroksittaista ohjelmistoarkkitehtuurityyliä. Yleisen selvityksen lisäksi on tärkeää käydä läpi malli-näkymä-ohjain-ohjelmistoarkkitehtuurin rakennetta. Tämä lisäksi selvitetään, minkälaisiin sovelluksiin malli-näkymä-ohjain-arkkitehtuuri soveltuu kaikista parhaiten.

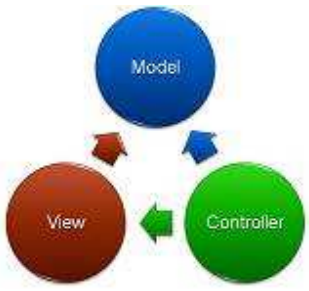
4.3.1 Yleistä malli-näkymä-ohjain-arkkitehtuurista

Malli-näkymä-ohjain-ohjelmistoarkkitehtuuri (Model-View-Controller) kuuluu vuorovaikutteisia järjestelmiä tukevaan tyyliin. Malli-näkymä-ohjain-arkkitehtuuri soveltuu erityisen hyvin järjestelmiin, joissa on pystyttävä luomaan vuorovaikutusta ihmisen ja järjestelmän välillä. Vuorovaikutus ihmisen ja järjestelmän välillä tapahtuu yleensä graafisen käyttöliittymän kautta. Malli-näkymä-ohjain-arkkitehtuuria käytetään usein web-pohjaisissa sovelluksissa. Graafisena käyttöliittymänä käyttäjälle toimivat yleensä sovelluksen web-sivut. (MSDN: Microsoft patterns & practices, Model-View-Controller 2009.)

Usein ohjelmistosovelluksen päätarkoituksena on palauttaa tietoa tietovarastosta, esimerkiksi tietokannasta, ja näyttää se käyttäjälle. Kun käyttäjä muuttaa tietoja, järjestelmä tallentaa muutokset. Onkin luonnollista ohjelmoida järjestelmä siten, että käyttöliittymä ja tiedontallennus toimivat kiinteästi yhdessä, jotta vältytään ylimääräiseltä ohjelmoinnilta. Tästä muodostuu kuitenkin ongelmaksi se, että käyttöliittymä muuttuu paljon useammin kuin järjestelmän tiedontallennusosio. Käyttöliittymässä voi olla lukuisia erilaisia näkymiä, joilla käyttäjä toimii, mutta tiedontallennuksessa voi olla usein kysymys ainoastaan muutamasta erilaisesta toiminnosta, joiden avulla muuttunut tieto tallennetaan tietokantaan. Lisäksi toiseksi ongelmaksi muodostuu sovelluksen toimintalogiikkakerros, joka toimii erillään järjestelmän tiedontallennuksesta. Näihin ongelmiin malli-näkymä-ohjain-arkkitehtuuri pyrkii tarjoamaan ratkaisun. Tavoitteena on tehdä jokaisesta osa-alueesta oma komponenttinsa, johon voidaan tehdä muutoksia

helposti, ilman että muut osa-alueet siitä kärsivät. (MSDN: Microsoft patterns & practices, Model-View-Controller 2009.)

Järjestelmän käyttäjälle näkyvä osa, eli käyttöliittymä, voi muuttua usein. Järjestelmään saatetaan lisätä uusia sivuja tai sivujen ulkomuotoa muutetaan. Tällöin on erityisen paljon hyötyä siitä, että käyttöliittymä on eriytetty omaksi komponentiksi. Muutokset voidaan tehdä usein muuta sovellusta kevyempään käyttöliittymäkoodiin helposti, ilman että koko järjestelmän toimintalogiikkaa tarvitsee alkaa muuttaa käyttöliittymämuutosten vuoksi. Jos järjestelmän eri osat, kuten käyttöliittymä, toimintalogiikka ja tietojen tallennus, olisi yhdistetty yhdeksi komponentiksi, pienetkin muutokset vaatisivat ohjelmakoodiin tarkkaa läpikäyntiä monessa eri paikassa. Tämän lisäksi sovellusta jouduttaisiin testaamaan läpikotaisin, ennen kun voitaisiin varmistua sen toiminnasta. Kun komponentit ovat eriytetty omiksi kokonaisuuksiksi, niitä pystytään testaamaan pienissä osissa, minkä avulla pystytään varmistumaan pienien osien toiminnasta ja voidaan siirtyä aina ylemmän tason testaukseen, kun sen alapuolinen taso on todettu toimivaksi. (Microsoft Asp.Net MVC Overview 2010.)



KUVIO 3. Malli-näkymä-ohjain-arkkitehtuurin komponentit (Microsoft Asp.Net MVC Overview 2010.)

Malli-komponentti (model) -tehtävänä on huolehtia sovelluksen tiedonvälityksestä. Malli-komponentti kuuntelee ohjain-komponentin käskyjä ja hakee niiden mukaan tiedon tietokannasta ja välittää sen eteenpäin ohjaimelle, joka välittää tiedot eteenpäin. (Microsoft Asp.Net MVC Overview 2010.)

Ohjain-komponentti (controller) sisältää kaikki sovelluksen toiminnallisuudet. Ohjain välittää käskyjä mallille, joka palauttaa ne lopulta käyttöliittymään haluttuun näkymään.

Näkymä-komponentti (view) huolehtii tiedon ja toimintojen näyttämisestä käyttöliittymään. Näkymä ei sisällä mitään toiminnallisuuksia, vaan se on esimerkiksi tiedonsyöttökenttiä ja valikoita käyttöliittymässä, joihin käyttäjät voivat syöttää tietoja. Näkymä vie tiedot ohjaimelle, joka lopulta päättää, miten syötteisiin tulee suhtautua. (Microsoft Asp.Net MVC Overview 2010.)

4.3.2 Malli-näkymä-ohjain-arkkitehtuurin edut

Microsoftin kehittämä ASP.NET MVC Framework tarjoaa joustavan valmiin rakenteen, jonka pohjalta pystytään luomaan rakenteellisesti haasteellisiakin web-sovelluksia helpommin. ASP.NET MVC Frameworkin avulla rakenne pilkotaan ensiksi kolmeen eri osaan: malli-, näkymä- ja ohjain-osioihin, mikä selkeyttää sovelluksen rakennetta jo erittäin paljon. (ASP.NET MVC Overview n.d.)

Malli-näkymä-ohjain-arkkitehtuuri on suhteellisen monimutkainen arkkitehtuuri, joten pitää miettiä, milloin sen käyttö on järkevää. Kun suunnitelmat sovellusta varten on tehty ja tiedetään, että sovellus tulee olemaan laaja ja sisältämään useita toiminallisuuksia, tiedonsyöttökenttiä, tiedon tallennusta ja hakemista, voidaan alkaa harkita malli-näkymä-ohjain-arkkitehtuuriin siirtymistä. Alussa työmäärä on suuri, jotta rakenne saadaan toimivaksi, mutta tämän jälkeen uusien näkymien tekeminen valmiiden ohjain- ja malli-osioiden pohjalta on helppoa. Kun sovelluksen koko kasvaa samalla dramaattisesti lisäten uusien sivujen tarvetta, pystytään malli-näkymä-ohjain-arkkitehtuurin avulla pitämään rakenne kuitenkin hallittavana. (ASP.NET MVC Overview n.d.)

Sovellusten laajuuden kasvaessa niiden parissa työskentelevien työntekijöiden määrä luonnollisesti kasvaa. Malli-näkymä-ohjain-arkkitehtuurin ansiosta saman sovelluksen parissa työskentelevien kesken pystytään jokaiselle jakamaan sovelluksesta helposti oma osa, esimerkiksi jokin näkymä tai ohjaimen osa. Tällöin vältytään jatkuvilta konflikteilta ohjelmointikoodissa, joita saattaa helposti syntyä, jos monta henkilöä työskentelee samojen tiedostojen parissa.

4.4 Komponenttipohjainen arkkitehtuuri

Tässä luvussa kerrotaan komponenttipohjaisesta arkkitehtuurista. Tärkeää on määritellä, mikä on komponentti ja miten se sijoittuu tutkimuksessa käytettäviin muihin arkkitehtuurityyleihin. Komponenttipohjainen arkkitehtuurin esittely ja sen käyttötarkoitusten läpikäynti ovat tämän tutkimuksen kannalta kaikista tärkeimmät asiat. Komponenttiarkkitehtuuri on laajalti käytössä ohjelmistojen rakenteessa. Tämän lisäksi jokainen komponentti voi pitää sisällään vielä omanlaisensa arkkitehtuurin. Käsitteet menevät hieman ristiin, joten on syytä selvittää, mikä on komponentin rooli esimerkiksi kerroksittaisessa arkkitehtuurityylissä.

4.4.1 Yleistä komponenttipohjaisesta arkkitehtuurista

Komponenttipohjainen ohjelmointi (Component-oriented programming) ja komponettipohjainen ohjelmistokehitys nousivat kuumiksi puheenaiheiksi 1990-luvun loppupuolella, kun haluttiin parantaa ohjelmistotuotannon tuottavuutta. Keksittiin, että on kannattavaa pyrkiä kierrättämään koodia mahdollisimman paljon sen sijaan, että aina aloitettaisiin sovellusten tekeminen tyhjältä pöydältä. Ohjelmistosovellukset koostuvat kuitenkin pääosin aina joiltain osin samoista osa-alueista. Esimerkkeinä tästä toimii esimerkiksi käyttäjänhallinta tai sovellukseen kirjautuminen eli sessioiden hallinta. (Pree 1997.)

Ohjelmistojen uudelleenkäyttö (software reuse) on muodostunut erittäin kannattavaksi toimintatavaksi. Jos yrityksellä on kokemusta jo useasta erilaisesta ohjelmistoprojektista, pystytään näiden ohjelmistoprojektien eri osista, komponenteista, rakentamaan myös aivan uusia ohjelmistoja. Ohjelmistojen uudelleenkäytön tärkein hyöty on se, että aina ei tarvitse ohjelmistoprojektia aloittaa koodin kirjoituksessa nollatilanteesta. Kun ohjelmistoja rakennetaan jo vanhoista, itse kehitetyistä osista, tarkoitetaan koostavaa uudelleenkäyttöä. Tällä periaatteella tullaan toimimaan myös toimeksiantajayrityksessä. Tutkimusten mukaan jopa 40 % - 80 % ohjelmistokoodista on uudelleen käytettävissä sovelluksesta toiseen. (Sametinger 1997.)

4.4.2 Komponenttipohjaisen arkkitehtuurin edut

Komponenttipohjaisessa ohjelmistoarkkitehtuurissa on monia merkittäviä etuja. Komponenttiarkkitehtuuriin siirtymistä on harkittava tarkoin. On otettava huomioon, minkälaisia projekteja yritys on tehnyt tai mahdollisesti tulee tekemään. Jos yritys tulee kehittämään useita ohjelmistoprojekteja, komponenttipohjainen arkkitehtuuri on varteenotettava vaihtoehto. Komponenttipohjaisen arkkitehtuurin myötä ohjelmistokoodista tulee paremmin uudelleenkäytettävää, josta on merkittävä hyöty kehitettäessä useita eri ohjelmistoprojekteja. Jos yritys toimii useiden vuosien ajan vain muutamassa projektissa tai tekee räätälöityjä ohjelmistoratkaisuja asiakkaiden tilauksesta, komponenttiarkkitehtuuri ei välttämättä ole oikea ratkaisu. Parempiin tuloksiin saatetaan päästä kevyemmällä arkkitehtuurimallilla.

Useita projekteja kehitettäessä komponenttiarkkitehtuurin edut tulevat parhaiten esille. Kun ohjelmoijat ovat työskennelleet joidenkin komponenttien parissa jo useassa edellisessä projektissa, komponenttien laatu on huomattavasti parempi kuin aivan uuden komponentin kohdalla. Mahdollisia virheitä on korjattu jo useaan kertaan, ja ihannetilanteessa toimiva ja erityisesti testattu komponentti pystytään liittämään uuteen järjestelmään helposti ja vaivattomasti. (Sametinger 1997.)

Komponenttiarkkitehtuurin myötä ohjelmiston kehitys nopeutuu huomattavasti, koska ohjelmistot koostetaan erilaisista komponenteista, joita voidaan kehittää rinnakkain. Samojen komponenttien käyttö useissa eri sovelluksissa tehostaa ohjelmistokehitystä, koska niiden avulla pystytään rakentamaan ohjelmiston runko monilta eri osa-alueilta valmiiksi. Sovelluskohtaiset erikoistoiminnot ovat tietenkin jokaisessa ohjelmistoprojektissa omia kokonaisuuksiaan, mutta niiden ympärille rakentuva sovelluksen runko on hyvin usein toteutettavissa jo aiemmin kehitetyillä ohjelmistokomponenteilla. (Sametinger 1997.)

Komponenttipohjaisen arkkitehtuurin avulla on helpompi määrittää projektissa työskentelevien henkilöiden tarve sekä heidän tehtävänsä. Työnjakoa voidaan suunnitella esimerkiksi jakamalla tehtäviä komponenttien mukaan. Jokaisen eri komponentin parissa

voi työskennellä tehokkaasti yhdestä useampaa henkilöä, toki riippuen projektin ja komponenttien laajuudesta. Tämän avulla saadaan projektissa työskentelevien henkilöiden työtehokkuutta kasvatettua, koska saadaan jokaiselle työntekijälle helposti määriteltävä vastuualue. Henkilökohtainen vastuu itse kehittämästään komponentista voi jatkua myös muihin projekteihin, joissa komponenttia tullaan mahdollisesti käyttämään. Komponenttien jatkokehitys on näin ollen huomattavasti helpompaa, koska jokaiseen komponenttiin löytyy oma asiantuntija, joka on täysin perillä komponentin rakenteesta ja toiminnasta. (Jaaksi 1998.)

Ohjelmistojen laatua pystytään tarkastelemaan ISO 9126 -standardin perusteella. Ohjelmistoissa on kuusi ominaisuutta, joiden perusteella sen laatu pystytään määrittelemään: toiminnallisuus, ylläpidettävyys, käytettävyys, luotettavuus, tehokkuus ja siirrettävyys. Komponenttipohjainen ohjelmistokehitys tukee kaikkien näiden osa-alueiden kehitystä. (ISO/IEC 9126 and 14598 integration aspects: A Brazilian viewpoint 2000.)

Tärkeimmät komponenttipohjaisen ohjelmistokehityksen edut saavutetaan ohjelmistokomponenttien uudelleenkäytöllä, jonka myötä ohjelmistoihin voidaan luoda nopeammin uusia toiminnallisuuksia. Komponenttipohjaisen rakenteen avulla saavutetaan myös hyvä ylläpito, koska komponenttien tilalle voidaan helposti vaihtaa jokin erilainen, paremmin käyttötarkoitukseen sopiva komponentti, jos ongelmia havaitaan. (Veryard 1997.)

Jokaiseen ohjelmistoon vaaditaan tietyt standardikomponentit, joiden ympärille ohjelmiston perusrakenne rakentuu. Tämä parantaa erityisesti ohjelmistojen käytettävyyttä. Yrityksessä voidaan luoda helposti tuoteperheitä, joihin kuuluvat ohjelmistot ovat ulkoasultaan ja käytettävyydeltään lähellä toisiaan. Kun ohjelmisto rakennetaan samoista standardikomponenteista, siitä seuraa väistämättä samankaltainen yhtenäinen linja koko ohjelmistoperheen kesken. Asiakkaiden on helppo käyttää ohjelmistoja ja hankkia myös uusia saman tuoteperheen ohjelmistoja, koska niiden käyttö on pääpiirteittäin tuttua jo entuudestaan. (Veryard 1997.)

Ohjelmistojen tehokkuutta komponenttipohjainen ohjelmistokehitys parantaa merkittävästi. Standardikomponentit saadaan optimoitua tehokkaasti toimiviksi monen eri kehitysrupeaman aikana. Tämän vuoksi pystytään keskittymään sovelluskohtaisesti uusien komponenttien tehokkuuden parantamiseen. Tehokkuuden parantamiseen liittyy läheisesti myös ohjelmiston luotettavuus. Standardikomponentteihin voidaan pääsääntöisesti luottaa, minkä ansiosta ongelmien paikantaminen helpottuu huomattavasti. Ongelmat pystytään rajaamaan helposti hyvinkin pienelle alueelle johtuen useiden eri komponenttien aikaisemmasta kehityskaaresta. Ohjelmiston joustavuus myös paranee huomattavasti komponenttiarkkitehtuurin myötä. Ohjelmistot eivät enää ole jäykkiä kokonaisuuksia, vaan ne voivat sopeutua helposti muuttuviin vaatimuksiin. Osia voidaan vaihtaa ja kehittää helposti, ilman että ohjelmiston rakenne ja toimivuus siitä kärsii. (Veryard 1997.)

4.4.3 Komponenttipohjaisen arkkitehtuurin haasteet

Komponenttipohjaisessa ohjelmistoarkkitehtuurissa on paljon hyviä puolia, mutta on myös syitä, jotka asettavat sen käytölle omat haasteensa. Komponenttipohjainen ohjelmistoarkkitehtuuriin siirtyminen luo tiettyjä vaatimuksia, jotka on otettava huomioon ohjelmistoja suunniteltaessa. Kokonaan uudenlaiseen arkkitehtuuriin siirtyminen on tietysti jokaiselle ohjelmistoalan yritykselle suuri haaste. Uusi arkkitehtuuri vaikuttaa ohjelmiston suunnittelemiseen ja toteutukseen erittäin merkittävästi. Haasteet voidaan jakaa kahteen eri kategoriaan, teknisiin ongelmiin ja organisationaalisiin ongelmiin.

4.4.4 Tekniset ongelmat

Tekniset ongelmat ovat uuden ohjelmiston kehityksen alkuvaiheessa laajalti esillä. Yrityksen vasta siirtyessä komponenttipohjaiseen arkkitehtuuriin sillä ei vielä ole kehitettynä valmiiksi mitään komponentteja. Komponenttien täydellisen puutteen vuoksi on suuri kynnys aloittaa uuden ohjelmiston rakentaminen täysin tyhjästä. Komponenttiarkkitehtuuri ei välttämättä sovellu hyvin kovin pienille yrityksille, joilla on vain muutama eri ohjelmistotuote, joita ne kehittävät. Uuden komponenttipohjaisen ohjelmiston kehittäminen vie huomattavasti enemmän aikaa ja resursseja kuin vastaavan

kokoisen ohjelmiston kehittäminen perinteisin menetelmin ilman uudelleenkäytettävien komponenttien hyödyntämistä. Uudelleenkäytettäviä ohjelmistokomponentteja ei yleensä synny projektien yhteydessä, ellei siihen kiinnitetä erityistä huomiota. Ohjelmistoissa käytetään usein liian yksityiskohtaisia ratkaisuja, jotta niitä pystyttäisiin käyttämään jatkossa myös muissa sovelluksissa. (Sametinger 1997.)

Suurin tekninen haaste komponenttien uudelleenkäytettävyyden suhteen on niiden rajapintojen kuvauksessa. Rajapintojen kuvaukset jäävät yleensä puutteelliseksi, jotta komponentista saataisiin helposti uudelleenkäytettävä. Yleinen tapa rajapintakuvauksessa on kuvata vain komponentin sisältämät metodit sekä ainoastaan metodien sisäiset parametrit ja niiden käyttämät palautustyypit. Jotta komponenttia pystyttäisiin hyödyntämään helpommin, pitäisi lisäksi kuvata komponentin yleistä toimintaa huomattavasti paremmin ja yksityiskohtaisemmin. Komponenttien uudelleen käyttäminen on erittäin vaikeaa, jos rajapintakuvauksen perusteella saadaan komponentin metodien toiminnasta väärä kuva. Komponentti saattaa toimia täysin oikealla tavalla, mutta jos sitä käytetään väärällä tavalla, jää toimiva lopputulos saavuttamatta. Komponenttien rajapintakuvauksissa olisi myös syytä kuvata komponentin ei-toiminnallisia ominaisuuksia. Rajapintakuvauksesta olisi hyvä selvitä helposti esimerkiksi se, kuinka paljon komponentti kuluttaa muistia ja mitä muita resursseja se vaatii. Tällä tavoin saataisin helpommin muodostettua yleiskuva siitä, onko jokin komponentti sopiva tiettyyn ohjelmistoon. (Szyperski 1998.)

Yhdeksi komponenttipohjaisen arkkitehtuurin tekniseksi ongelmaksi voi muodostua sen hajautettu rakenne. Järjestelmä voi koostua useasta eri komponentista, jotka ovat kaikki määrätty käyttämään omia tietokantojaan. Tämä asettaa suuria haasteita, jos näistä kaikista tietokannoista pitää pyrkiä muodostamaan keskitetty tiedonhaku esimerkiksi raporttia varten. Useaan tietokantaan yhtä aikaa tehtävien tietokantakyselyiden rakenne muuttuu helposti erittäin monimutkaiseksi, ja lopulta niiden tekeminen ei enää järkevällä työmäärällä onnistu. Ongelmia ei vielä ilmene, jos käytössä on ainoastaan muutama eri tietokanta. Jos raportteja tarvitaan usean tietokannan sisältävästä järjestelmästä, on siihen mietittävä jotain erillistä ratkaisua. Tietokantojen sisältö voidaan esimerkiksi ajaa

johonkin ulkoiseen järjestelmään, joka on varta vasten rakennettu tiedon tarkempaa analysointia varten.

4.4.5 Organisationaaliset ongelmat

Komponenttipohjaiseen arkkitehtuuriin siirtyminen voi olla hankalaa perustella yhtiön johdolle. Jos asiakkaille myydään ohjelmistoja tiukalla aikataululla, yrityksen johdolla ei välttämättä riitä ymmärrystä siihen, miksi ohjelmistoja pitäisi suunnitella monimutkaisemmin ja enemmän resursseja sekä aikaa vieden. Perinteisien ohjelmistoprojektien elinkaarta ei ole suunniteltu uudelleenkäytettävyyttä huomioiden. Komponenttipohjaisissa ohjelmistoprojekteissa kuluu aikaa ja resursseja luonnollisesti niiden uudelleenkäytettävyyden parantamiseen, mikä ei välttämättä ole selkeästi hahmotettava resurssitarve. Se on kuitenkin pakollista, jotta komponenttipohjaista ohjelmistosuunnittelua pystytään jatkossakin toteuttamaan kannattavasti. (Pree 1997.)

Yksi merkittävä aspekti komponenttia suunniteltaessa on ottaa huomioon myös se, kannattaako komponenttia kehittää itse vai olisiko parempi ja edullisempi vaihtoehto hankkia komponentti alihankkijan kautta. Joissain tapauksissa saattaa komponentin hankkiminen ulkopuoliselta toimittajalta olla erittäin hyvä ja edullinen ratkaisutapa, mutta siinä on myös tietyt riskit. Täysin omaan järjestelmään sopivan komponentin löytäminen voi osoittautua erittäin vaikeaksi. Komponentin toimittaja saattaa ajautua talousvaikeuksiin tai lopettaa komponentin kehittämisen. Tämä saattaa johtaa komponenttiin liittyvän tuen täydelliseen loppumiseen sekä komponentin kehityksen loppumiseen. (Salonen 1998.)

5 TEKNOLOGIAN VAIHTO

Tässä luvussa käsitellään niitä syitä, joiden takia teknologian vaihtaminen tuli opinnäytetyön toimeksiantajayrityksessä ajankohtaiseksi. Luvussa käsitellään tarkemmin, minkälainen oli alkutilanne, jonka vuoksi alettiin harkita uudenlaista ohjelmistoarkkitehtuurimallia. Lisäksi käydään läpi hieman myös vanhaa toimintatapaa ja sen asettamia rajoituksia yrityksen kasvulle.

5.1 Vanhan teknologian rajoitukset

Opinnäytetyön toimeksiantajayritys tarjoaa asiakkailleen pääosin SaaS-liiketoimintamalliin perustuvia palveluita. Kehitettävät ohjelmistot ovat web-pohjaisia sovelluksia, joista myydään asiakkaille vain käyttöoikeuksia. Asiakkaan ei tarvitse huolehtia ohjelmiston toimintaympäristöstä eikä asennuksista. Asiakkaan kanssa tehdään sopimus siitä, mitä ohjelmistoja tarvitaan, sekä näiden ohjelmistojen maksullisista lisäominaisuuksista. Kun sopimus on tehty, asiakas saa omat käyttäjätunnuksensa, joiden avulla ohjelmistojen käyttö onnistuu heti. Tämä toimintamalli on asiakkaan kannalta erittäin edullinen, koska mittavia laitehankintoja ei tarvitse tehdä eikä ohjelmistojen asennuksesta tai päivityksestä tarvitse huolehtia. Ohjelmistojen käyttö onnistuu helposti internet-yhteyden ja web-selaimen avulla omalta tietokoneelta. Aina kun asiakas kirjautuu palveluun, on hänellä käytössä ohjelmiston uusin versio.

Vuosien varrella kehitettyjen ohjelmistojen koko on paisunut huomattavasti, koska kehityksen myötä on ohjelmistoihin toteutettu runsaasti uusia toiminnallisuuksia. Ohjelmistojen arkkitehtuuri on pääosin ollut monoliittinen kokonaisuus. Ensin ohjelmistosta on tehty versio, jossa on ollut ainoastaan tiettyjä perusominaisuuksia. Tähän päälle on lisätty pikkuhiljaa paljon uusia ominaisuuksia, joiden myötä ohjelmisto on paisunut ja paisunut. Alussa monoliittinen toteutus toimii, koska sitä pystytään vielä testaamaan helposti. Aina uusia ominaisuuksia lisättäessä ongelmaksi muodostuu se, mihin osiin nämä uudet ominaisuudet vaikuttavat vanhassa koodissa. Voi hyvinkin olla, että uudet ominaisuudet rikkovat joitain vanhoja ohjelman osia. Koska järjestelmät ovat kasvaneet laajoiksi kokonaisuuksiksi, on niiden testaaminen ja ylläpidettävyys

muodostuneet joissain määrin suuriksikin kompastuskiviksi. On mahdollista toteuttaa myös ylläpidettäviä ja erityisesti testattavia laajoja sovelluksia, mutta siinä kohtaa mukaan astuu vahvasti ohjelmistoarkkitehtuurin muuttaminen.

Laajat ohjelmistot on pyrittävä jakamaan useisiin pienempiin osiin ja kerroksiin. Tässä vaiheessa alkavat pienemmät ja helpommin hallittavat ohjelmistokomponentit vaikuttaa erittäin hyvältä vaihtoehdolta.

5.2 Uuden ohjelmistoarkkitehtuurin valinta

Uuden ohjelmistoarkkitehtuurin valinta voi olla erittäin hankalaa, ellei ole tarvittavaa tietotasoa tai kokemusta erilaisista ohjelmistoarkkitehtuureista. On syytä miettiä niitä näkökulmia, joiden ansiosta komponenttipohjainen ohjelmistoarkkitehtuuri osoittautuisi juuri oikeaksi toimintatavaksi toimeksiantajayrityksen liiketoimintamalliin ja soveltuisi parhaiten SaaS-toimintamallin tueksi. Ohjelmistojen muuttamisessa uuden teknologian pohjalta toimiviksi on valtava työmäärä, eikä tämänkaltaisia päätöksiä pidä tehdä ilman vahvoja perusteita.

Aiemmin on huomattu, että ohjelmistojen laajuus on osoittautunut ongelmaksi. Ohjelmistojen jakaminen pienempiin osakokonaisuuksiin onnistuu loistavasti komponenttiarkkitehtuurilla. Toimeksiantajayrityksessä ei aikaisemmin ole kiinnitetty kovinkaan suurta huomiota ohjelmistokoodin uudelleenkäyttöön. Totta kai tiettyjä koodin osia on käytetty useissa eri sovelluksissa, mutta tämä voi koitua suureksi riskiksi. Jos kopioitavaa koodia ei ole testattu tarpeeksi hyvin, se saattaa toimia uudessa ympäristössä aivan eri tavalla kuin alun perin oli suunniteltu. Testauksen puutteen vuoksi koodiin jääneet virheet tietysti monistuvat sitä mukaa, kun koodia käytetään uusissa ympäristöissä. Komponenttipohjainen ohjelmistoarkkitehtuuri tarjoaa tässä tilanteessa erittäin hyvän ratkaisun. Kun uusiin ohjelmistosovelluksiin liitetään jokin tietty komponentti, se tapahtuu keskitetysti. Liitettävä komponentti on aina sama jokaisessa sovelluksessa, joten komponenttiin jääneet mahdolliset virheet eivät tässä tapauksessa monistu. Virheiden korjaaminen hoituukin paljon helpommin, koska voidaan selkeästi määrittää tietty komponentti, joka vaatii korjauksia. Kun virhe korjataan tietyssä

komponentissa, se korjautuu luonnollisesti joka paikassa, jossa kyseistä komponenttia käytetään.

Komponenttipohjainen ohjelmistoarkkitehtuuri istuu SaaS-liiketoimintamalliin erittäin hyvin. Kun yritys luo ohjelmistoja usean eri asiakkaan käyttöön, on helposti muodostettavissa tiettyjä yleiskomponentteja, joita tullaan käyttämään lähestulkoon jokaisessa yrityksen sovelluksessa. Näiden komponenttien avulla pystytään luomaan helposti tunnistettava oma tyyli jokaisen sovelluksen välillä, ja tämän avulla asiakkaan on helppoa ja luontevaa ottaa käyttöön myös yrityksen muita tuotteita, jos yksi tai useampi on jo entuudestaan tuttu. Käyttöliittymäkomponenteilla on mahdollista luoda koko sovellusperheeseen yhtenäinen käyttöliittymä ja ulkoasu. Käyttöliittymän yhtenäisyyteen ja helppokäyttöisyyteen on syytä paneutua ohjelmistoja kehitettäessä huolellisesti, jotta käytön helppous välittyy lopulta asiakkaalle asti. Käyttöliittymän tultua tutuksi asiakkaan olisi mahdollista pyrkiä käyttämään myös yrityksen muita tuotteita saman käyttölogiikan perusteella.

SaaS-liiketoimintamalli perustuu siihen, että ohjelmistoon lisätään useiden eri iteraatiokierrosten kuluessa uusia ominaisuuksia. Ohjelmistot eivät ole täysin tietylle asiakkaalle räätälöityjä, koska ohjelmistoja käyttävät useat eri asiakkaat. Kuitenkin asiakaspalautteen perusteella saaduista uusista ominaisuustoivomuksista pystytään valitsemaan sellaiset, joista on hyötyä usealle eri asiakkaalle. Uusia ominaisuuksia pystytään lisäämään ohjelmiston uusiin versioihin helpommin kuin aikaisemmin. Kun muutokset rajoittuvat vain tiettyihin komponentteihin, pystytään ne testaamaan ja näin ollen varmistumaan siitä, että ohjelmiston muut osat toimivat riippumatta uusista ominaisuuksista.

6 UUSI ARKKITEHTUURIMALLI KÄYTÄNNÖSSÄ

Tässä luvussa käsitellään toimeksiantajayrityksen arkkitehtuurimallia. Arkkitehtuurimalli perustuu komponenttipohjaiseen toteutustapaan, mutta se pitää sisällään myös kerroksellista arkkitehtuurirakennetta. Tässä luvussa käsitellään arkkitehtuurimallia yrityksessä tehdyn pilottiprojektin pohjalta.

6.1 Arkkitehtuurin rakenne

Teknologian vaihdon myötä vaihtuu arkkitehtuurimalli, joka täytyy suunnitella toimivaksi, jotta uuteen toimintatapaan siirtymisestä koituu hyötyä pitkällä aikavälillä. Komponenttipohjaiseen arkkitehtuuriin siirtyminen on haastavaa, koska jokainen ohjelmistokomponentti pitää sisällään vielä arkkitehtuurisen rakenteen, jonka pohjalta se koostuu.

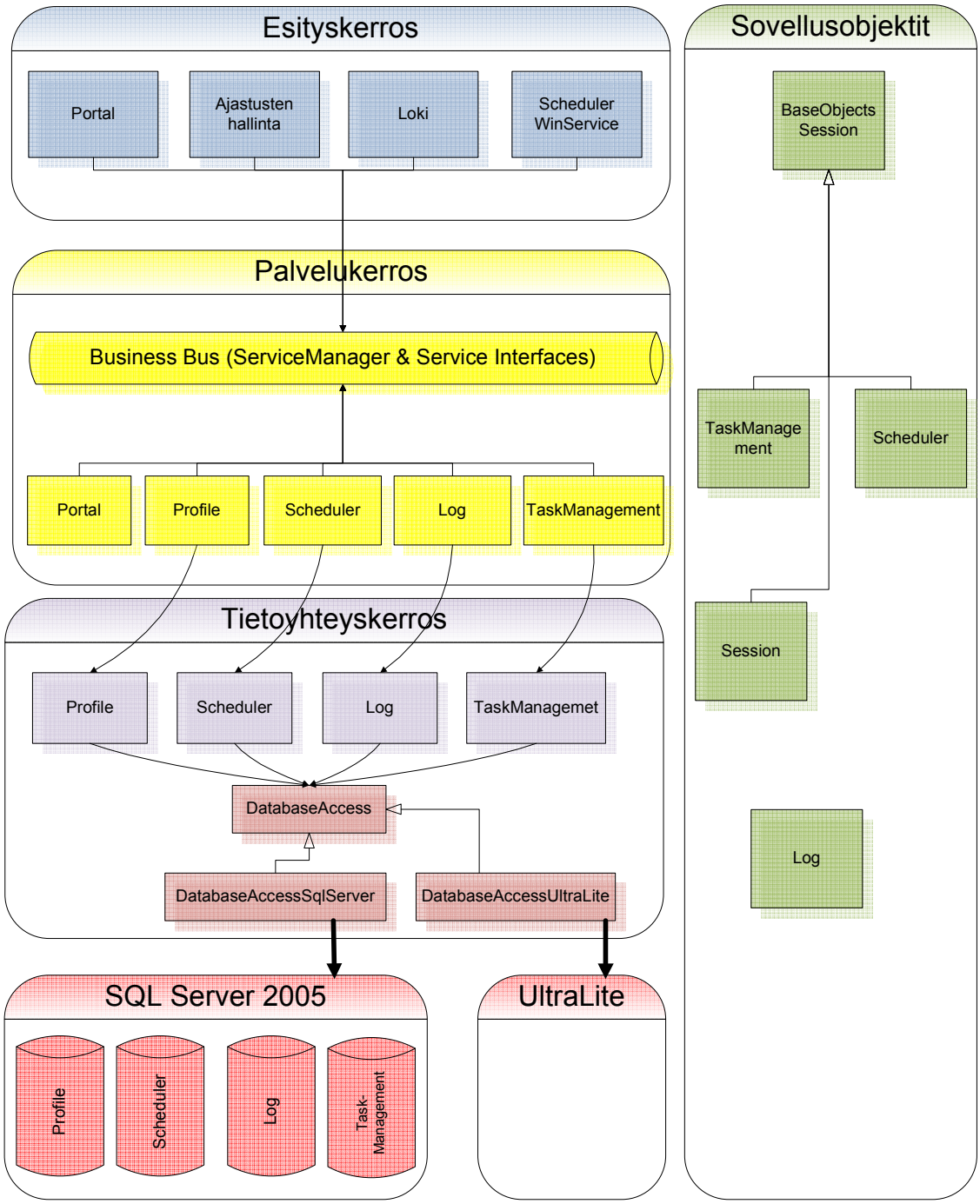
Ohjelmiston rakennetta on hyvä alkaa purkaa ylhäältä käsin. Ohjelmisto rakentuu useista eri ohjelmistokomponenteista, jotka tarjoavat palveluitaan toisilleen. Nämä komponentit ovat sisäiseltä rakenteeltaan jaettu kerroksellisen arkkitehtuurimallin mukaisesti. Lisäksi kerrosarkkitehtuurin päällä toimii vielä malli-näkymä-ohjain-arkkitehtuuri, joka muodostaa käyttäjälle näkyviin sovelluksen käyttöliittymän.

Arkkitehtuurin valinta ei aseta rajoituksia sille, millä tekniikalla järjestelmä lopulta toteutetaan. Sen tarkoituksena on vain antaa tietynlaiset suuntaviivat ja rajoitukset, joiden pohjalta tulee toimia. Kun projektia tarkastellaan hieman tarkemmasta näkökulmasta, on jo syytä kiinnittää huomiota siihen, miten ja millä tekniikoilla se toteutetaan.

6.2 Ajastetut tehtävät -sovellus

Toimeksiantajayrityksessä pilottiprojektina uuden arkkitehtuurin sisäänajolle toteutettiin sovellus, jonka tarkoituksena on asettaa ajastettuja tehtäviä. Tämän sovelluksen avulla voidaan esimerkiksi lähettää tietty sähköposti tiettyyn aikaan asiakkaalle. Tämän

sovelluksen ominaisuudet pystytään tulevaisuudessa helposti integroimaan muihin sovelluksiin. Näin pystytään luomaan muistutuksia tai muita tapahtumia toteutumaan juuri tiettynä ajankohtana automaattisesti ennalta määrätyllä tavalla.



KUVIO 4. Ajastetut tehtävät -sovelluksen arkkitehtuurimalli

Sovellus koostuu useista eri komponenteista, jotka on rakennettu kerrosarkkitehtuuriin perustuen. Komponentit on jaettu kerroksellisesti kolmeen eri kerrokseen. Käyttäjälle näkyvä käyttöliittymä muodostuu esityskerroksesta. Esityskerros voidaan toteuttaa millä tahansa tekniikalla. Tässä sovelluksessa on ajastusten hallinta -näkymä, joka näytetään asiakkaalle web-sivun kautta. Sovelluksen palveluihin voi jokin muu sovellus olla yhteyksissä vaikka web-servicen kautta, jolloin esityskerrokseen määritellään ainoastaan rajapintakutsut. Näiden rajapintakutsujen avulla pystytään sovellus integroimaan yhteensopivaksi toisen käyttöympäristön tarpeisiin.

Sovelluksen esityskerroksen näkymä voidaan esittää monella tavalla käyttötarkoituksesta riippuen. Ajastusten hallinnan kautta käyttäjä pystyy suoraan olemaan yhteyksissä tehtäviin. Toisena näkymänä voidaan käyttää esimerkiksi tapahtumalokia, joka on täysin oma komponenttinsa, mutta sen avulla pystytään tarkastelemaan, mitä ajastuksen hallinta -sovelluksessa on tapahtunut. Ajastettujen tehtävien hallintakäyttöliittymä, joka näkyy palveluun kirjautuville käyttäjille, on toteutettu ASP.NET MVC:llä. Esityskerroksessa siis käytetään tässä tapauksessa malli-näkymä-ohjain-arkkitehtuuria, jonka pohjalle web-käyttöliittymä rakentuu.

Eri komponenttien integraatio samaan järjestelmään tapahtuu käytännössä palvelukerroksessa ja tietokerroksessa. Esityskerros on yhteydessä palvelukerrokseen. Palvelukerroksessa toimii Business Bus, joka pitää sisällään palveluiden kutsut sekä palvelurajapinnat. Tämän välittäjän hyöty tulee esiin siinä, että kaikki määritykset ja viitteet täytyy tehdä ainoastaan yhteen paikkaan. Kaikkien komponenttien pyynnöt kulkevat tämän kautta, joten säästytään suurelta määrältä turhia päällekkäisiä viittauksia eri komponenttien rajapintoihin ja palvelukutsuihin.

Sovellusobjektit sisältävät luokat, joihin on määritetty ohjelmistossa käytettävät oliot. Näissä olioluokissa ei ole määritelty olioiden toimintoja. Luokkiin on määritelty olioiden attribuutit ja ainoastaan näihin attribuutteihin liittyvien tietojen asetusmetodit. Sovelluksesta löytyy perusolio, BaseObject, joka pitää sisällään kaikkien muiden olioiden perusattribuutit. Kaikki muut oliot perustuvat BaseObjectiin, jonka jokainen olio perii.

Muut oliot sisältävät niille ominaisia lisäominaisuuksia. Luokkien perinnällä säästytään huomattavalta työmäärältä, varsinkin jos erilaisia olioita on lukuisia.

Tietokerroksessa sijaitsevat tietokantaan yhteydessä olevat luokat ja metodit. Palvelukerroksesta tulevat palveluiden kutsut tietokerrokselle, joka edelleen välittää pyynnöt tietokantaan DatabaseAccess-luokan kautta. DatabaseAccess-luokkaan on tarkasti määritelty, minkä muotoisina sql-kyselyt tietokantaan menevät. DatabaseAccess-luokan käytöllä saavutetaan se, että käytettävä tietokanta on helposti vaihdettavissa toiseen. Tällä hetkellä käytettävä tietokanta on Microsoft SQL Server 2005. Tämä rakenne, jossa käytetään vielä erillistä DatabaseAccess-luokkaa, monimutkaistaa rakennetta jonkin verran.

Erillisen DatabaseAccess-luokan käyttö vaatii lisäksi lisää kirjoitettavaa koodia, mutta siitä on suuri hyöty tulevaisuutta ja järjestelmän laajentamista ajatellen. DatabaseAccess-luokkaan voidaan kirjoittaa omat luokkatiedostot jokaista eri tietokantatyyppiä varten. Tämän avulla pystytään tietokanta vaihtamaan helposti käyttötarkoitusta paremmin vastaavaksi. Sen sijaan, että kaikki tietokantakyselyt olisi tiukasti kirjoitettu jokaiseen tiedonhakumetodiin ainoastaan yhtä tietokantaa tukevaksi, pystytään DatabaseAccess-luokkien avulla hallitsemaan tietokantakyselyt usealle erilaiselle tietokantatyypille.

Tulevaisuudessa tärkein toisentyyppisen tietokannan käyttökohde on mobiilipalvelut. Rinnalle voidaan ottaa käyttöön esimerkiksi Sybase UltraLite -tietokanta, joka on suunniteltu mobiilipalveluita varten. Tämän uuden arkkitehtuurimallin myötä mobiilisovelluksen teko on huomattavasti vaivattomampaa kuin aikaisemmin. Palvelukerros ja tietokerros voidaan pitää ennallaan, ja ainoastaan esityskerrokseen on tehtävä oma käyttöliittymänsä mobiilisovellukselle. Tässä huomataan, miten komponenttipohjainen ohjelmistoarkkitehtuuri tukee uudelleenkäyttöä ja korvattavuutta.

Tämän pilottiprojektin avulla saatiin positiivisia tuloksia ohjelmistokomponenttien uudellenkäyttöä ajatellen. Projektin tuloksena syntyi mm. loki- ja käyttäjänhallintakomponentit. Lokikomponentilla pystytään seuraamaan ohjelmassa tapahtuvia tietokantakutsuja, sekä erinäisiä palvelukutsuja. Tämä on yleishyödyllinen

komponentti järjestelmän toiminnan seuraamiseen. Käyttäjänhallintakomponentilla huolehditaan käyttäjien kirjautumisesta järjestelmään, sekä käyttäjätietojenhallinnasta. Nämä komponentit ovat täysin uudelleenkäytettävissä tulevissa projekteissa. Tämän avulla säästytään suurelta työmäärältä, koska komponentin liittäminen uuteen ohjelmistoon onnistuu vaivattomasti.

7 POHDINTA

Ohjelmistokehitys on vuosien saatossa mennyt kohti yhä monimutkaisempia tietojärjestelmiä. Järjestelmien suuri koko ja laajuus asettavat monia haasteita ohjelmistokehittäjille ja varsinkin ohjelmistoarkkitehdeille. Ohjelmistoja voidaan kehittää äärettömän monella eri tavalla. Näkökulmasta riippuen ohjelmistosuunnittelun haasteet voivat tuntua erilaisilta. Suuria ja monimutkaisia ohjelmistokokonaisuuksia on erittäin hankala alkaa lähestyä ilman kunnollista suunnittelua. Tässä suunnittelussa yksi suurimmista ja tärkeimmistä kokonaisuuksista on oikeanlaisen ohjelmistoarkkitehtuurin suunnittelu ja määrittely.

Ohjelmistoarkkitehtuurin avulla voidaan muuttaa erittäin laajakin kokonaisuus helposti hallittavaksi, eri osista koostuvaksi sovellukseksi. Toisaalta vääränlaisella tai kehnolla ohjelmistoarkkitehtuurilla pystytään myös kaatamaan lupaava ohjelmistoprojekti. Ohjelmistoarkkitehtuurin valinnassa on monia näkökulmia, joita on otettava huomioon, jotta saavutetaan toimiva kokonaisuus. Pitää ottaa huomioon projektin tarpeet. Miten raskaan arkkitehtuurin tuleva projekti tarvitsee? Tullaanko projektia kehittämään jatkossa, vai onko kyseessä kertaluontoinen toimitus asiakkaalle? Syntyykö käytettävällä arkkitehtuurilla laadukas ohjelmisto?

Yrityksellä, jolla ei ole ollut käytössä selkeää ohjelmistoarkkitehtuuria, on erittäin suuri kynnys siirtyä käyttämään tiukasti määriteltyä uutta arkkitehtuuria. Ohjelmiston voi varmasti tehdä helposti ilman tarkasti määriteltyä arkkitehtuuria, mutta tämä puute alkaa tietyssä vaiheessa vaatia veronsa. Pienissä projekteissa tätä ongelmaa ei välttämättä pääse ilmenemään, mutta laajemmissa järjestelmissä siihen törmääminen on väistämätöntä.

Tutkimuksessa kävi hyvin ilmi eri perusteita, minkä vuoksi komponenttipohjainen arkkitehtuuri on juuri sopiva opinnäytetyön toimeksiantajayrityksen tarpeita ajatellen. SaaS-periaatteeseen perustuva liiketoiminta tukee mainiosti komponenttipohjaista arkkitehtuuria. On suuri etu, että useaa omaa ohjelmistoa pystytään kehittämään rinnakkain. Se, että ohjelmistoja ei asenneta käyttäjille, vaan käyttäjät käyttävät niitä

toimeksiantajayrityksen internetsivujen kautta, korostaa erityisesti komponenttiarkkitehtuurin hyötyjä. Jos löydetään joitain virheitä, ne pystytään korjaamaan päivittämällä yhden komponentin tiettyä osaa, jossa virhe ilmenee. Tämän korjauksen avulla virhe korjaantuu saman tien myös kaikkiin muihin samaa komponenttia käyttäviin sovelluksiin.

Jos yritys kehittää paljon samantyyppisiä ohjelmistoprojekteja, olisi syytä miettiä, miten toimintaa saataisiin tehostettua. Tähän yhtenä ratkaisuna on koodin uudelleenkäyttö. Komponenttiarkkitehtuuri toimii tästä näkökulmasta tarkasteltuna paremmin kuin hyvin. Komponenttien uudelleenkäyttö saattaa säästää vuosien kuluessa valtavan määrän työtunteja.

LÄHTEET

ASP.NET MVC Overview. 2010.

Http://www.asp.net/learn/mvc/tutorial-01-cs.aspx.

Bass, L., Clements, P. & Kazman, R. (1998) *Software Architecture in Practice*. Addison Wesley Longman, inc., Reading, Massachusetts.

Bredemeyer, D. & Malan, R. (2000) *The Role of the Software Architect.*
Http://www.bredemeyer.com/pdf_files/role.pdf.

CIO Software as a Service (SaaS) Definition and Solutions. 2010.
Http://www.cio.com/article/109704/Software_as_a_Service_SaaS_Definition_and_Solutions.

Zimmermann, H. IEEE Transactions on Communications, vol. 28, no. 4, April 1980.

ISO/IEC 9126 and 14598 integration aspects: A Brazilian viewpoint. The Second World Congress on Software Quality, Yokohama, Japan, 2000.

Jaaksi, A. From Objects to Components. 1998.

MSDN: Microsoft patterns & practices, Model-View-Controller, 2009
Http://msdn.microsoft.com/en-us/library/ms978748.aspx

Microsoft .NET Framework Conceptual Overview, 2010
Http://msdn.microsoft.com/fi-fi/library/zw4w595w%28en-us,VS.100%29.aspx

Microsoft ADO.NET, 2010

Http://msdn.microsoft.com/en-us/library/aa286484.aspx

Microsoft ASP.NET Overview, 2010

Http://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx

Microsoft MSDN Three layered services application, 2010
Http://msdn.microsoft.com/en-us/library/ms978689.aspx

Pree Wolfgang: Component-Based Software Development – A New Paradigm In Software Engineering?, Software – Concepts and Tools, 1997

Salonen Petri: Quality of Objects and Components, Merito Forum Oy, 1998

Sametinger Johannes: Software Engineering with Reusable Components, Springer-Verlag, 1997

Shaw, M. (1996) *Some Patterns for Software Architectures* Proceedings of the Second Pattern Languages of Program Design Workshop, Addison-Wesley, Pennsylvania.

Szyperski Clemens: Component Software – Beyond Object-Oriented Programming, Addison-Wesley, 1998

Veryard Richard: Component-Based Development, Veryard projects, 1997