ARKKITEHTUURIKESKEINEN LÄHESTYMISTAPA TAVOITTEENA OHJELMISTON PAREMPI LAATU

Harri Karhunen

Pro gradu-tutkielma
Tietojenkäsittelytiede
Ohjelmistotekniikka

Kuopion yliopisto
Informaatioteknologian ja kauppa-tieteiden tiedekunta

Syyskuu 2003

TIIVISTELMÄ

KUOPION YLIOPISTO, Informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytiede
Ohjelmistotekniikka

KARHUNEN, HARRI J. : Arkkitehtuurikeskeinen lähestymistapa tavoitteena ohjelmiston parempi laatu

Pro gradu-tutkielma, 141 s., 8 liitettä (57 s.)

Pro gradu tutkielman ohjaaja: FT, professori(mvs) Anne Eerola
Syyskuu 2003

Avainsanat: Ohjelmistoarkkitehtuuri, arkkitehtuurin parantaminen, arkkitehtuuri- ja suunnittelumallit, komponenttiteknologia, arkkitehtuurin arvioinnin kehittäminen

Tutkielman tavoitteena on selvittää liiketoimintajärjestelmien arkkitehtuurin parantamiseen liittyviä tekijöitä huomioiden komponenttipohjainen sovelluskehitys.

Liiketoiminnan lähtökohtana ovat markkinat ja asiakkaiden esittämät vaatimukset. Asiakkaiden vaatimusten pohjalta suunnitellaan ja rakennetaan toimitettava tuote. Prosessinhallinta kilpailukykyisesti laadukkaita tuotteita rakentaen on yrityksen tärkeimpiä kilpailutekijöitä ohjelmistoteollisuudessa.

Liiketoimintaprosessit toteutetaan tänään hajautetussa ympäristössä. Hajautuksen mukanaan tuoma monimutkaisuus aiheuttaa uusia vaatimuksia järjestelmäkehitykseen. Monimutkaisuutta pyritään hallitsemaan komponenttiteknologian avulla. Komponenttipohjainen ajattelutapa tuo uusia vaatimuksia ohjelmistotuotantoon. Komponenttien keskinäisiä suhteita tutkitaan arkkitehtuurien avulla. Arkkitehtuurien avulla voidaan syntyvää järjestelmää arvioida ja parantaa sen ominaisuuksia. Arkkitehtuurisuunnittelu toteutetaan hyödyntämällä mallijärjestelmiä, joiden avulla ohjelmistojen rakennetta voidaan parantaa. Arkkitehtuurien avulla voidaan ohjelmiston rakenne dokumentoida asianosaisia kiinnostavalla tavalla. Komponentti- ja arkkitehtuuripohjainen ajattelutapa pyrkii parantamaan ohjelmiston laatua ja lisäämään uudelleenkäyttöä. Tässä tutkimuksessa selvitetään laadunparantamiseen ja uudelleenkäyttöön liittyviä tekijöitä tavoitteena lisätä tuotteen kilpailukykyä.

Tutkielmaa varten on haastateltu kolmea yritystä. Haastattelun lähtökohtana on ollut jokin yrityksen järjestelmässä oleva ongelma tai piirre. Tuloksia on peilattu arkkitehtuurikeskeisesti. Myös ongelman ratkaisu on esitetty.

Tehokas, laadukas komponenttipohjainen tuotanto vaatii koko yrityksen toimintaprosessin läpikäymisen. Yritykseltä edellytetään kykyä kypsempään toimintaan ja kykyä parantaa tuotantoprosessia. Esitetyt mallit on suunniteltu suuria yrityksiä ajatellen, pienet yritykset voivat hyödyntää arkkitehtuuri- ja komponenttikeskeistä ajattelutapaa soveltamalla malleja luovasti ja pyrkien ottamaan niistä juuri omaan liiketoimintaan sopivat piirteet.

Esipuhe

Tämä tutkielma on tehty Kuopion yliopiston tietojenkäsittelytieteen laitoksella vuonna 2003. Tutkielman ohjaajana toimi FT, professori Anne Eerola, jolle esitän parhaimmat kiitokset asiantuntevasta ohjauksesta ja avusta työn tekemisessä.

Haluan kiittää haastatteluihin osallistuneita yrityksiä ja henkilöitä heidän avustaan tutkimuksen tekemisessä.

Kuopiossa 29.9.2003

_____

Harri Karhunen

1 Johdanto 7
1.1 Yleistä
1.2 Markkinat
1.3 Arkkitehtuurit 8
1.4 Arkkitehtuurien parantaminen kilpailukeinona 9
2 Hajautetut järjestelmät 11
2.1 Yleistä 11
2.2 Hajautettujen järjestelmien periaatteita 12
2.2.1 Integraatio 12
2.2.2 Kommunikaatio 12
2.2.3 Hajautuksen arkkitehtuuri 14
2.2.4 Hajautettujen järjestelmien luokittelua 14
2.3 Middleware-ohjelmistot hajautuksen toteuttajana 15
2.3.1 OSF:n hajautettu tietojenkäsittely-ympäristö 16
2.3.2 Tapahtumamonitori (TP monitor) 16
2.3.3 MOM 17
2.3.4 Verkkopalvelut (web services) 18
2.3.5 Oliopohjaiset hajautustekniikat
2.3.6 CORBA Common Object Request Broker –arkkitehtuuri
2.3.7 Java J2EE-arkkitehtuuri
2.3.8 Microsoft .NET arkkitehtuuri
2.3.9 Yhteenveto hajautuksesta ja välitason ohjelmista 26
3 Komponentit – uudelleenkäyttö ja liiketoimintakomponentit 29
3.1 Yleistä 29
3.2 Komponenttien luokittelu ja rakeisuus 29
3.3 Komponenttien kehittäminen 32
3.4 Kehykset ja komponenttisysteemin metamalli 38
3.5 Uudelleenkäytettävät ohjelmistot ja komponentit 42
3.6 Komponentit ja integraatio 45
3.7 Yhteenveto komponenteista 47
4 Arkkitehtuurit 49
4.1 Ohjelmistoararkkitehtuurit – arkkitehtuurin näkökulmat 49
4.1.1 Yleistä 49
4.1.2 Näkökulmat 50
4.1.3 Yhteenveto 53
4.2 Arkkitehtuurityylit,-mallit ja suunnittelumallit suunnittelun välineenä 54
4.2.1 Yleistä 54
4.2.2 Arkkitehtuurityylit ja -mallit 55
4.2.3 Esimerkki arkkitehtuuri- ja suunnittelumallien käytöstä 56
4.2.4 Yhteenveto 61
4.3 Arkkitehtuuri ja sidosryhmät – dokumentoinnin haasteet 61
4.4 Vaatimusten hallinta ja ohjelmistoprosessin organisointi 62
4.4.1 Yleistä 62
4.4.2 Vaatimusten määrittely 64
4.4.3 Prosessin mallintaminen 66
4.4.4 Prosessin kypsyyden arviointi 70
4.4.5 Yhteenveto 72
4.5 Arkkitehtuurin suunnittelu 72
4.5.1 Yleistä 72
4.5.2 Arkkitehtuurin suunnittelu 73
4.5.3 Yhteenveto 77

4.6 Arkkitehtuurin parantaminen 78
4.6.1 Yleistä 78
4.6.2 Arkkitehtuurin parantaminen 78
4.6.3 Yhteenveto 80
4.7 Tuotelinja-arkkitehtuuri 80
4.7.1 Yleistä 80
4.7.2 Tuotelinja-arkkitehtuuri 81
4.7.3 Yhteenveto 82
4.8 Arkkitehtuurin dokumentointi 83
4.8.1 Yleistä 83
4.8.2 Näkymät dokumentaatioon ja dokumentoitavat kohteet 83
4.8.3 Hyvän dokumentaation tunnusmerkit 84
4.8.4 Dokumentaatio ja asianosaisten kiinnostuksen kohteet 85
4.8.5 Yhteenveto 86
4.9 Yhteenveto arkkitehtuureista 86
5 Esimerkkitapaukset 89
5.1 Yleistä – tavoitteet, tutkimusmenetelmät 89
5.2 Esimerkki 1 – haastattelu 90
5.3 Esimerkki 2 – haastattelu 91
5.4 Esimerkki 3 – haastattelu 92
5.5 Yhteenveto ja johtopäätökset haastatteluista 92
5.5.1 Haastattelujen tulokset 92
5.5.2 Yhteenveto 93
6 Tavoitteena arkkitehtuurin parantaminen 95
6.1 Tuotantoprosessin parantaminen – uudelleenkäytön lisääminen 95
6.1.1 Kohdealueen analysointi 95
6.1.2 Prosessin parantaminen 98
6.1.3 Uudelleenkäytön lisääminen 99
6.2 Laadun määrittäminen ja laadun arviointi 103
6.2.1 Laatuattribuuttien asettaminen 103
6.2.2 Arkkitehtuuri ja laatuattribuutti – laatutekijän kohdistaminen arkkitehtuuriin 106
6.3 Arkkitehtuurin arvioinnin kehittäminen 106
6.3.1 Arkkitehtuurin arviointi yleistä 106
6.3.2 SAAM 109
6.3.3 ATAM 110
6.4 Tuotteenhallinta ja tuotelinja-arkkitehtuuri keinoina lisätä uudelleenkäyttöä 113
6.4.1 Tuotteenhallinnan tehtävät 113
6.4.2 Tuotelinja-arkkitehtuuri ja uudelleenkäyttö 114
6.5 Liiketoimintakomponenttikehys ja mallintaminen keinona lisätä uudelleenkäyttöä 117
6.6 Kohdealuespesifiset sovelluskehykset (Domain-specific application frameworks) 121
6.7 Yhteenveto 123
7 Pohdinta 125
7.1 Yleistä 125
7.2 Palkanlaskentajärjestelmä esimerkkinä arkkitehtuurin käytöstä 128
7.2.1 Kuvaus järjestelmästä 128
7.2.2 Muunneltavuusanalyysi 128
7.2.3 Arkkitehtuuri 130
7.2.4 Käyttötapaukset 130
7.2.5 Aktiviteettikaavio 132

7.2.6 Komponentit 132
7.2.7 yhteenveto mallista 135
7.3 Arkkitehtuurin parantaminen kilpailukeinona 135
Lähteet 139

LIITTEET

LIITE 1. Sanasto ja käsitteet
LIITE 2. Esimerkki 1 Kuopion kaupunki sosiaali- ja terveyskeskus

LIITE 3. Esimerkki 2 Primus-oppilashallintojärjestelmä, Kuopion kaupunki oppimateriaalikeskus
LIITE 4. Esimerkki 3 Prointra-Intranet –ohjelmisto, JSOP OY
LIITE 5. Laatuattribuuttien painoarvot
LIITE 6. Laatuattribuuttien ominaisuuksien kartoitus
LIITE 7. Arkkitehtuurien mallijärjestelmiä
LIITE 8. Arkkitehtuuri- ja suunnittelumalleja staattinen kuvaus

1 Johdanto

1.1 Yleistä

Aivan liian usein on lehdistä luettavissa kritiikkiä ohjelmistoteollisuuden tuotannon laadusta. Milloin pettää laatu, milloin palvelut eivät toimi tai markkinoiden toimivuus on arvostelun kohteena [Jun03].

Ohjelmistot ovat osa jokapäiväistä elämäämme ja niiden kanssa on tultava toimeen. Olemme kehittäneet useita tapoja tulla toimeen erilaisten ohjelmistojen kanssa. Olemme tottuneet siihenkin, että ostamassamme ohjelmistotuotteessa on vikoja - vakaviakin. Usein asiakas on niin riippuvainen tuotteesta, että hän sietää melkein mitä tahansa, jotta työt saadaan edes jollain tapaa suoritetuksi.

Asiakkaat eivät ole tyytyväisiä nykytilaan. Aivan liian usein tuotteet tuodaan markkinoille keskeneräisinä. Ongelmat ovat nimenomaan laadussa. Ohjelmistojulkistukset tuodaan markkinoille puolivalmiina. Ohjelmien testaus laiminlyödään. Asiakas on testaaja [Jun03].

Tuote on saatava nopeasti markkinoille (time-to-market). Laatu koetaan tärkeäksi, mutta aikataulutus- ja resurssien kohdentamisongelmat johtavat siihen, että jostain on tingittävä. Tingitään suunnittelusta, tingitään testauksesta. Tänään asiakkaat odottavat hyötyjä investoinneistaan. Myyjän pitää pystyä osoittamaan, mitkä ovat ohjelmiston hyödyt asiakkaalle. Asiakas myös vaatii, että lupaukset pidetään. Jotta asiakkaan vaatimuksiin voitaisiin vastata, on mietittävä keinoja, joilla tuotantoprosessia voidaan kehittää niin, että valmistettava tuote saadaan tuotettua nopeammin, halvemmalla ja riittävän laadukkaasti.

1.2 Markkinat

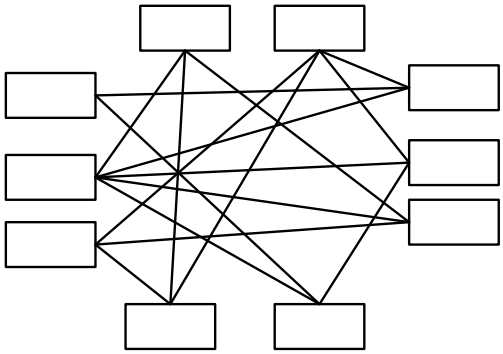
Liiketoimintasovellusten ja –palvelujen markkina-arvosta merkittävä osa tulee eri informaatiojärjestelmien toiminnallisuuden yhteensovittamisesta. Integroinnin osuus IT-panostuksista on noussut jopa 70% [Jun03]. Tämä kertoo, missä on liiketoimintasovellusten business-to-business –markkinoiden painopiste.

Markkinoilla menestytään, jos kyetään valmistamaan parempia tuotteita kuin muut ja paremmin kuin muut. Esimerkiksi, jos yritys A:n markkinoima tuote on joustavampi, helpompi integroida muihin järjestelmiin kuin yrityksen B tuote, on selvää, että A:lla on etulyöntiasema. Voidakseen valmistaa tuotteita kilpailukykyisesti yritys A on parantanut valmistusprosessiaan, jonka ansiosta tuotteen valmistusaikaa on kyetty lyhentämään. Lisäksi uudelleenkäyttöä lisäämällä on pystytty alentamaan valmistuskustannuksia. Tuotteen on oltava myös joustava, integroitavissa ja vikasietoinen. Tuotteen on oltava laadukas. Laatu on tuotteessa sen rakenteessa eli arkkitehtuurissa. Tässä tutkielmassa tutustutaan ohjelmistotuotannon eri käsitteisiin tavoitteena löytää tekijöitä, joiden avulla saataisiin tuotettua parempilaatuisia ohjelmistoja markkinoille. Puhuttaessa arkkitehtuureista (Architecture) tarkoitetaan tässä tutkielmassa ohjelmistoarkkitehtuureja (Software architecture), ellei muuta ole mainittu.

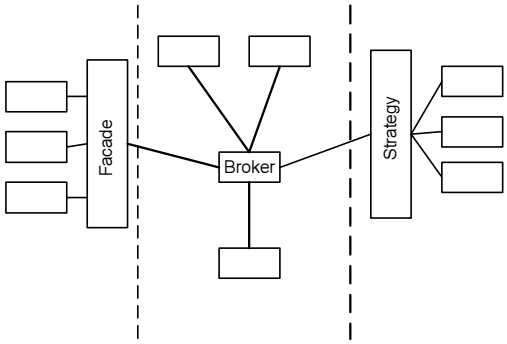
1.3 Arkkitehtuurit

Ohjelmiston suunnittelu ja rakentaminen on varsin monimutkaista. Hajautus lisää vielä monimutkaisuutta. Hajautettujen järjestelmien monimutkaisuuden hallintaan on olemassa väliohjelmistoja, sovelluspalvelimia ja sovelluskehittimiä. Ne tarjoavat hajautettuun järjestelmään tarvittavan kommunikaatioväylän, hoitavat transaktiot sekä tarjoavat tietoturvan. Hajautettujen järjestelmien viitearkkitehtuuri muodostuu kerrosarkkitehtuurista, jossa *monikerrosarkkitehtuurin (multi-tier architecture)* avulla toteutetaan *asiakas-palvelin (client-server) -tehtäviä*. Väliohjelmistojen tehtävänä on paitsi väylän tarjoaminen palvelupyynnöille, myös nimipalvelun järjestäminen, jotta asiakas ja palvelu löytävät toisensa.

Ohjelmistotuotannon siirtyessä olioajatteluun, alkoi ohjelmistojen rakenteen analysointi ja kehittäminen muodostua tärkeäksi. Jo verrattain pienen ohjelmiston olioiden väliset viittaukset voivat muodostaa monimutkaisen *spagetin (suhteiden verkon)*. Jos ohjelman rakenne aluksi onkin selkeä, niin siihen elinkaaren aikana kohdistuneet muutos- ja parannustoimet voivat aiheuttaa sen, että sen rakenne degeneroituu niin pahoin, ettei kukaan saa siitä selvää (Kuva 1). Tällöin joudutaan koko ohjelmisto toteuttamaan uudelleen. Osittain tämän takia alettiin tutkia ohjelmistojen rakennetta. Tavoitteena oli muodostaa rakenteesta sellainen, että se olisi selkeä ja sallisi monimuotoisuuden käyttämisen ohjelmiston toiminnassa ja että ohjelmiston elinikä muodostuisi mahdollisimman pitkäksi. Alettiin tutkia erilaisten mallien käyttöä ja luotiin *mallijärjestelmiä (pattern systems)*, joiden avulla ohjelmistojen rakennetta voitiin selkeyttää (Kuva 2).



Kuva 1. Esimerkki elementtien välisistä suhteista spagetti-malli

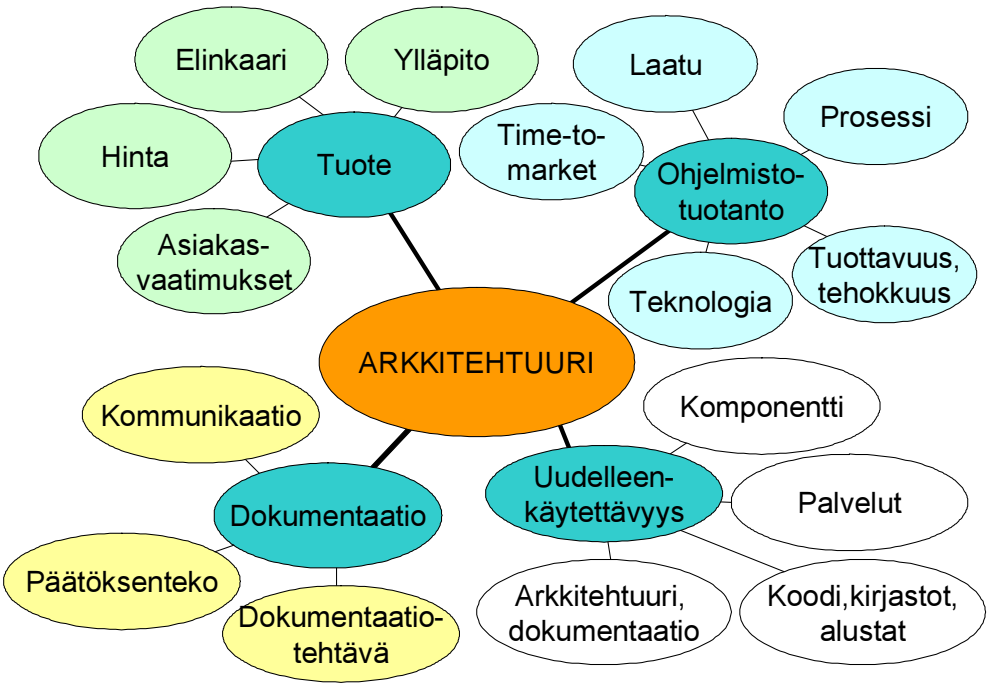


Kuva 2. Esimerkki elementtien välisistä suhteista
käyttäen ohjelmistoarkkitehtuureita ja malleja

Arkkitehtuurien avulla kuvataan rakennettavaa tuotetta. Arkkitehtuurien avulla voidaan myös analysointivaiheessa arvioida tuotteen laadullisia ominaisuuksia. Onnistunut ohjelmistotuote on tekijöidensä summa: markkinoinnin, tuotekehityksen ja asiakkaan yhteisvaikutuksen tulosta. Koska ohjelmistojen kehittäminen on kallista, tarvitaan keinoja kustannusten alentamiseen ja tuotantoaikojen lyhentämiseen. Komponenttiajattelu tehostaa uudelleenkäyttöä. Vähitellen voidaan muodostaa tuoteperheitä – tuotelinja-arkkitehtuuri, jolloin komponenttien kierrätystä voidaan tehostaa.

1.4 Arkkitehtuurien parantaminen kilpailukeinona

Tämän tutkielman tavoitteena on tutkia niitä keinoja, joita sekä asiakkaalla että ohjelmistoja tuottavalla yrityksellä on, kun ne pyrkivät parantamaan tietojärjestelmänsä tai ohjelmistonsa ominaisuuksia. Asiakas haluaa parempaa toiminnallisuutta, laatua ja integroitavuutta järjestelmiensä välillä. Tuottaja haluaa parempia ohjelmia, parempaa laatua, parempia ja nopeampia prosesseja. Tuottaja haluaa myös tuottaa ohjelmistot halvemmalla kuin kilpailijat. Kuvassa 3 on havainnollistettu *arkkitehtonisen lähestymistavan (architecture centric approach)* mukaisesti siihen liittyviä osa-alueita.



Kuva 3. Arkkitehtuuriin liittyviä käsitteitä

Tutkielmassa tutkitaan aluksi keskeisimpiä käsitteitä. Tavoitteena on selvittää teknologinen ympäristö ja arkkitehtuurien osuus tuotantoprosessissa. Arkkitehtuureista selvitetään keskeisimmät käsitteet ja arkkitehtuurin määrittelemiseen ja muokkaamiseen liittyvät tekijät. Case-osa koostuu kolmesta tapauksesta. Kaksi niistä on tuotetta käyttäviä organisaatioita ja yksi on ohjelmistoa tuottava markkinointiosasto. Tutkimusmenetelmänä on henkilökohtainen haastattelu. Haastattelun tavoitteena oli selvittää organisaation tavoitteita tuotteen suhteen: mikä oli konteksti, ongelma ja ratkaisu. Tavoitteena arkkitehtuurin parantaminen -osassa luodaan katsaus arkkitehtuurin muokkaamiseksi ja laadunhallinnan parantamiseksi. Lopuksi Pohdinta-osassa arvioidaan arkkitehtuurien merkitystä ja arkkitehtuurien parantamisen vaikutusta kilpailukykyyn.

2 Hajautetut järjestelmät

2.1 Yleistä

Tässä tutkielmassa lähtökohtana ovat yritysten tietojärjestelmät osana yrityksen liiketoimintaprosesseja. Hajautusteknologioiden avulla voidaan liiketoimintasovellukset hajauttaa tietoverkkoon. Hajautus on vaativa teknologia, jonka toteuttamiseksi on useita eri vaihtoehtoja. Yritykset voivat integroida toimintansa verkon välityksellä joko verkkopalveluiden avulla tai sovellustason integrointitekniikoiden avulla esim. oliopohjaisten väliohjelmien avulla. Toiminnallisuuden aikaansaamiseksi tarvitaan kommunikaatiota. Yritysten ja sovellusten on kyettävä kommunikoimaan keskenään. Joitakin keskeisimpiä tutkielmassa käytettyjä käsitteitä on selostettu sanastossa (Liite1).

Hajautettujen järjestelmien avulla voidaan tehtävät suorittaa hajautetusti tietoverkossa käyttäen verkossa saatavilla olevia palveluita. Uusissa järjestelmissä tietojenkäsittelyn hajautus on jossain muodossa aina mukana ja hajautuksen välineinä käytetään jonkinlaista *välitason ohjelmistoa (middleware)*. Asiakkaat ja palveluntarjoajat yhdistetään verkkopalveluiden ja välitason ohjelmien avulla.

Hajautuksen avulla saavutetaan *läpinäkyvyys (transparency)*. Systeemissä eri tehtävät ovat jakaantuneet eri koneille omiksi prosesseikseen. Prosessien välille luodaan yhteistoimintaa ja mahdollistetaan prosessien välinen kommunikaatio. Kommunikaatiokustannukset kasvavat hajautuksen myötä [Tan02], [Mul89 s. 6-9].

Hajautetun järjestelmän tulee huolehtia käyttäjänsä puolesta prosessien välisestä kommunikaatiosta, prosessien hallinnasta, nimipalvelusta, prosessien synkronoinnista, tiedon yhtenäisyydestä ja replikoinnista sekä transaktioiden hallinnasta ja virheestä toipumisesta ja turvallisuudesta. Nämä vaatimukset toteutetaan *sovelluspalvelinten (application servers)* avulla.

2.2 Hajautettujen järjestelmien periaatteita

2.2.1 Integraatio

Integroinnin tarkoituksena on mahdollistaa tiedon ja liiketoimintaprosessien jakaminen organisaatiossa siihen liittyvien sovellusten, tietovarastojen ja ihmisten kesken. Integraation suunnittelu ja hallinta on ohjelmistosuunnittelun keskeisimpiä tehtäviä. Hajautettujen järjestelmien kannalta voidaan integrointitehtävät jakaa *esitysintegraatioon (user interface level), tietovarastointegraatioon (data level), toiminnalliseen integraation* [Puu03]. Esitysintegraatiossa integrointi tapahtuu suoraan sovellusten käyttöliittymien kautta. Tietovarastointegraatiossa integraatio tapahtuu suoraan tietokantojen tai tietokannanhallintajärjestelmien kautta. Toiminnallisessa integraatiossa sovellukset integroidaan suoraan toisiinsa esimerkiksi *ohjelmointirajapintojen (API) kautta* [Puu03].

David Linthicum esittää integrointimallin, joka lähtee yrityksen omasta prosessista ja sen integroimisesta järjestelmään. Mallissa mahdollisina integrointipisteinä ovat *metodi, sovellusrajapinta* tai *tieto.* Integraatiotasot tässä mallissa ovat *tietosuuntautunut integrointimalli, sovellusrajapintasuuntautunut integrointimalli, metodisuuntautunut integrointimalli, portaalisuuntautunut integrointimalli* ja *prosessisuuntautunut integrointimalli* [Puu03], [Lin00] (kts. Liite 1).

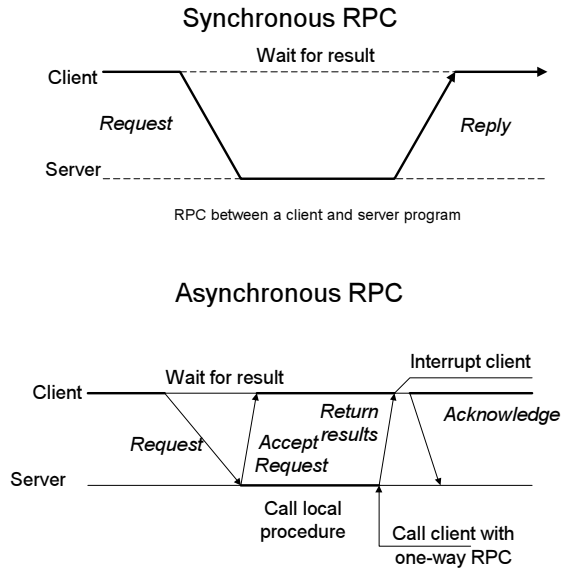
Tavoiteltavin, mutta myös vaativin integraatiomalli on prosessien välillä tapahtuva integrointi, jossa yhteisesti laadittujen sääntöjen avulla voidaan siirtää informaatioita eri organisaatioiden järjestelmien kesken. Aina tähän ei ole edellytyksiä. Usein voidaan sopia ainoastaan yhteisen tiedon muodosta. Tässä tapauksessa integrointi tapahtuu niin, että järjestelmät käyttävät yhteistä tietoa, joka voi olla vaikka toisen järjestelmän tietokanta, yhteinen tietokanta tai jaettu hakemisto, johon toinen tuottaa tiedoston ja josta toinen lukee tiedoston.

2.2.2 Kommunikaatio

Kommunikaation avulla määrätään järjestelmän ominaisuudet kuten skaalautuvuus, samanaikaisuuden hallinta ja suorituskyky. Kommunikaatio voidaan järjestää käyttäen *etäproseduurikutsua (remote procedure call RPC), etämetodikutsua (remote method invocation RMI), viestisuuntautunutta välitasoa (message-oriented middleware MOM)*

tai *tietovirtasuuntautunutta tapaa (stream-oriented)*. RPC ja RMI-tyyppiset asiakas/palvelin-yhteydet ovat yleensä synkronisia, mikä tarkoittaa sitä, että asiakas pyytää palvelua ja jää odottamaan kunnes saa vastauksen. Asynkronisessa tyylissä asiakas lähettää pyynnön ja jatkaa toimintaansa. Kun vastaus saapuu, asiakas keskeyttää toimintansa ja käsittelee tulleen vastauksen. Kuvassa 4 on esitetty synkronisen ja asynkronisen RPC:n yksinkertaistettu periaate.

Tavallinen asiakkaan ja palvelimen välinen tiedonsiirto on synkronista. Skaalautuvissa suuren volyymin järjestelmissä, joissa on asiakasmäärät vaihtelevat suuresti esimerkiksi ollen 10.000 tai 1 milj. samanaikaista käyttäjää, palvelin toimii asynkronisesti. Tällöin palvelupyynnöt tulevat käsittelysäikeelle, joka ohjaa pyynnöt jonoon odottamaan käsittelyä.



Kuva 4. RPC-kutsu synkroninen/asynkroninen [Tan02 s. 79]

Kommunikaatio voi olla joko *yhteydellinen (connection-oriented)* tai *yhteydetön (connectionless)*. Yhteys luodaan ennen tietojen lähettämistä tai sitten yhteyttä ei ole etukäteen muodostettu. Yhteydetöntä viestintää on esimerkiksi http-protokollan mukainen selainyhteys palvelimeen.

2.2.3 Hajautuksen arkkitehtuuri

Tietojenkäsittelyn perussuhde on *asiakas-palvelin –suhde (client-server –style)*. Asiakas on vailla palvelua ja kysyy sitä. Yhteys on *point-to-point* –tyyppinen. Palvelin suorittaa tehtävän saatuaan pyynnön ja vastaa pyyntöön. Hajautetussa ympäristössä tarvitaan välittäjää, jonka tehtävänä on välittää palvelupyyntö palvelimelle. Välitystehtävä toteutetaan V*älittäjä (Broker)-arkkitehtuurimallin* avulla. Palvelimen toiminta on jaettu eri kerroksiin sovelluspalvelimen avulla, jolloin syntyy monikerrosarkkitehtuuri (multi-tier architecture). Palvelimen operaatiot tuodaan asiakkaalle esimerkiksi käyttäen etäoliota, joka edustaa palvelinta asiakkaan koneessa. Tällöin käytetään *Proxy-suunnittelumallia*. Point-to-point –tyyppinen yhteys ei aina sovellu kaikkiin tapauksiin. Esimerkiksi tapahtumankäsittelyssä tarvitaan menetelmä, jossa tapahtumista kiinnostuneet saavat tiedon tapahtumista. Tämä toteutetaan *Tarkkailija (Publish-Subscribe tai Observer) –suunnittelumallin* avulla. Malli toteuttaa o*ne-to-many* –tyyppisen yhteyden osapuolten välillä.

2.2.4 Hajautettujen järjestelmien luokittelua

Hajautetut järjestelmät voidaan jaotella hajautettuihin oliopohjaisiin systeemeihin (distributed object-based system), hajautettuihin tiedostojärjestelmiin (distributed file systems), hajautettuihin dokumenttipohjaisiin systeemeihin (distributed document-based system) sekä hajautettuihin koordinaatiopohjaisiin systeemeihin (distributed coordination based system) [Tan02 s. 493-] (kts. Liite 1).

Oliopohjaisten systeemien tarkoituksena on tehtävien suorittaminen hajautettuja olioita käyttämällä, jossa etäoliolla on edustaja, jota asiakas voi käyttää tuotantoprosessissaan (kts. kohta 2.3.5). Hajautettujen tiedostojärjestelmien avulla taas voidaan yhteistä tietoa jakaa hajautetussa ympäristössä ja taata luotettava ja turvallinen toiminta. Hajautetut tiedostojärjestelmät ovat omana kerroksenaan rakennettaessa hajautettuja systeemejä ja sovelluksia. Hajautetut dokumentaatiopohjaiset systeemit pyrkivät puolestaan antamaan käyttäjälleen dokumentin sellaisena konseptina, että tieto on helposti ja yksinkertaisesti käytettävissä. Esimerkkinä tästä ovat tiedostopohjainen web-sivusto ja tietokantapohjainen Lotus Notes.

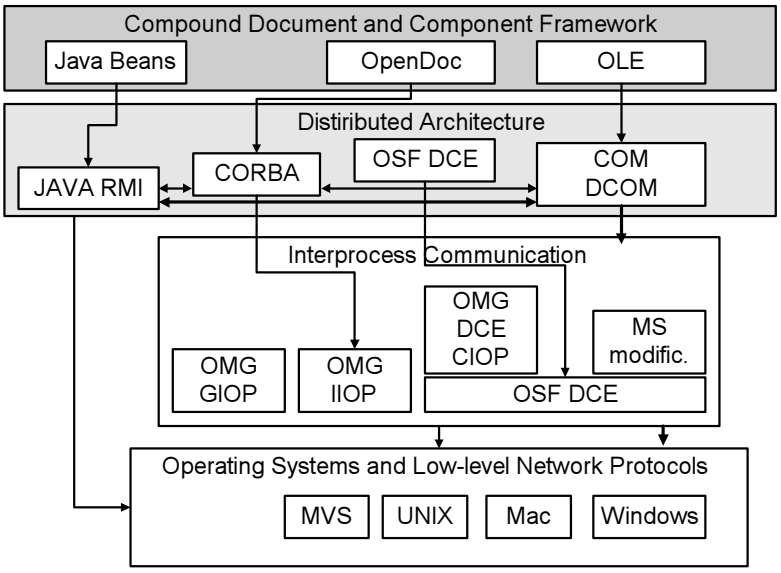
Hajautettu koordinaatiopohjainen systeemi pyrkii komponenttien sijasta koordinoimaan komponenttien välisiä toimintoja. Prosessit voivat kommunikoida keskenään tietämättä

etukäteen toisistaan. Tärkeintä koordinaatiopohjaisissa järjestelmissä on tietojenkäsittelyn ja koordinaation erottaminen toisistaan. Kommunikaatio hoidetaan yleensä asynkronisesti, mutta koordinaattorit voivat toimia myös synkronisesti. Viestienvälitys prosessilta toiselle voidaan toteuttaa koordinaatio-osassa esimerkiksi käyttämällä Tarkkailija–mallia (kts.liite 8). Yhteys voidaan hoitaa myös point-to-point –tyyppisesti. Esimerkiksi Javan Jini-arkkitehtuuri käyttää Tarkkailija-mallia ainoastaan silloin, kun palvelu alustetaan. Jinin koordinaatio-ominaisuuksien ansiosta asiakas ja palvelu löytävät toisensa verkosta. Jini käyttää kommunikointiteknogioina verkossa esimerkiksi RMI-, CORBA- tai SOAP-teknologioita Asiakas löytää palvelun palvelu API:n avulla. Jini tarjoaa palveluina esimerkiksi haku-, transaktio-, tapahtumanhallinta- ja postilaatikkopalveluita

2.3 Middleware-ohjelmistot hajautuksen toteuttajana

Välitasonohjelmistot (middleware) voidaan luokitella tietokanta- (database middleware), etäproseduurikutsu- (RPC), oliosanomanvälitin- (ORB), transaktioprosessien monitorointi- (transaction processing monitors TP) ja viestisuuntautuneisiin- (message-oriented MOM) välitason ohjelmistoihin [www010].

Hajautettujen järjestelmien arkkitehtuureilla on yhtymäkohtia ja yhdyskäytäviä, joiden avulla järjestelmät voivat kommunikoida keskenään ja joiden avulla eri järjestelmien toiminnallisuutta voidaan yhdistää (Kuva 5). Teknologiat perustuvat monelta osin samoihin perusperiaatteisiin. Näihin on määrittelyjen yhteydessä tehty valmistaja/määrittelijä-kohtaisia muutoksia ja parannuksia.

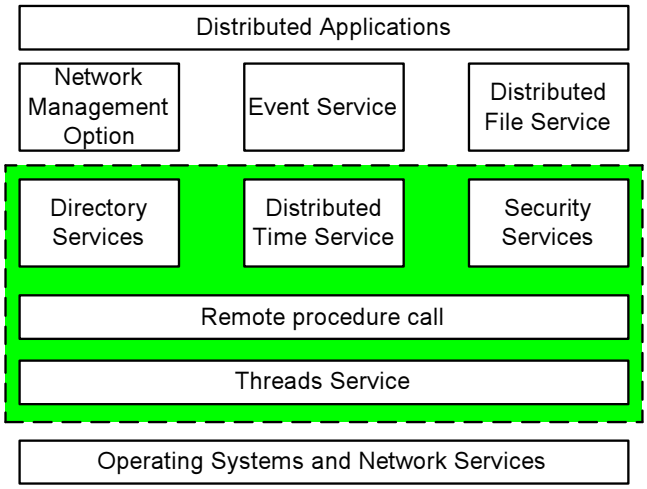


Kuva 5. Hajautettujen arkkitehtuurien teknologioiden vertailua [MoR97 s. 180]

Liiketoimintaprosessien kehittämisen kannalta mielenkiintoisimpia hajautettujen järjestelmien toteutusratkaisuja ovat hajautetut oliopohjaiset systeemit integroituna web-ympäristöön. Palvelupyyntöjä ja vastauksia hajautetussa ympäristössä liiketoimintasovellusten välillä voidaan välittää esimerkiksi käyttäen verkkopalveluita tai oliopohjaisia hajautustekniikoita.

2.3.1 OSF:n hajautettu tietojenkäsittely-ympäristö

OSF (Open Software Foundation) alkoi luoda 1980-luvulla määrittelyä ympäristön luomiseksi client-server –sovelluksille. 90-luvun alussa valmistui määrittely, joka loi verkko- ja käyttöjärjestelmän ja hajautettujen sovellusten välille palvelukerroksen palvelupyyntöjen välittämiseksi (Kuva 6). Hajautettu tietojenkäsittely-ympäristö (Distributed Computing Environment DCE) on proseduuripohjainen, eikä tue suoraan oliopohjaista ohjelmistokehitystä. DCE tarjoaa sovelluksille läpinäkyvyyden. DCE ja CORBA ovat monessa suhteessa samankaltaisia. Molemmat keskittyvät hajautettuihin asiakas/palvelin-sovelluksiin. Molempien takana on samoja organisaatioita ja yrityksiä. Molemmista on ollut toimittajaversioita 90-luvulta lähtien. Molemmat tarjoavat samoja palveluita mm. hakemisto-, aika-, säie- turvallisuus-, tapahtuma- ja verkonhallintapalvelut. Eroina ovat lähinnä CORBA:n oliopohjaisuus, dynaamisuus sekä laajempi tuki kohdealuekohtaisille toteutuksille. DCE on rakenteeltaan yksinkertaisempi kuin CORBA [MoR97 s. 187].



Kuva 6. DCE-arkkitehtuuri

2.3.2 Tapahtumamonitori (TP monitor)

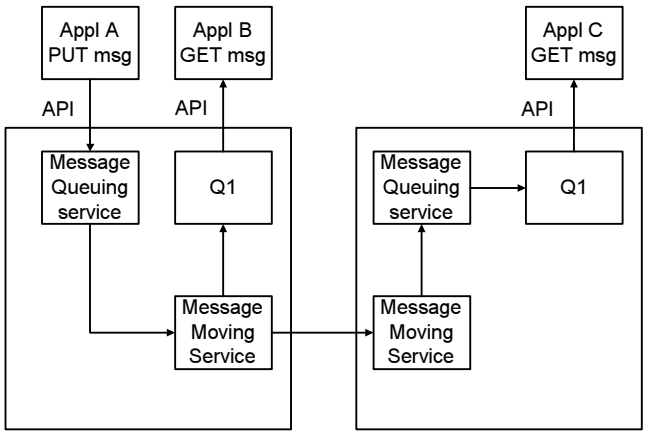
Tapahtumamonitorit (transaction processing monitor technology) tarjoavat asiakas/palvelin –kommunikointiin kapasiteettia tehokkaasti ja luotettavasti. Monitoreiden

avulla voidaan kehittää, ajaa ja hoitaa transaktioita. Asiakas on yhteydessä monitorointiohjelmaan, joka hoitaa transaktioita käyttäen API-rajapintaa. Tätä teknologiaa on käytetty 80-luvun alusta alkaen. Tapahtumamonitori hallinnoi useiden asiakkaiden prosesseja ollen julkisivuna asiakkaisiin päin ja hoitaen yhteydet palveluihin. Teknologia tarjoaa myös tehokkaan ja luotettavan *ohjelmien välisen (program-to-program)* kommunikointimahdollisuuden käyttäen eri malleja, kuten *tallenna ja lähetä (store-and-forward)* -, asynkroninen- ja RPC-mallit [www008]. Sovelluspalvelimet ovat tapahtumamonitoreja, koska ne API-rajapinnan kautta kontrolloivat asiakkaiden esittämiä palvelupyyntöjä ja välittävät pyynnöt eteenpäin palvelukomponentille [Puu03 s. 31].

2.3.3 MOM

MOM-tyyppisiä ohjelmistoja alettiin käyttää 1980-luvun loppupuolella. Ohjelmiston tarkoituksena on lisätä ohjelmistojen ja järjestelmien *yhteentoimivuutta (interoperability), siirrettävyyttä (portability)* ja *joustavuutta (flexibility)* asiakas/palvelinohjelmistoissa [www008].

MOM sopii tapahtumapohjaisiin sovelluksiin, jolloin voidaan soveltaa Tarkkailija (Publish-Subscribe) –suunnittelumallia. Oliopohjaisissa sovelluksissa mahdollistetaan sovellusten välinen point-to-point –tyyppinen viestienvaihto. Tässä mallissa esimerkiksi IBM:n mallin mukaisesti molemmissa päissä on *viestijonomanageri (message queue manager)*, jonka tehtävä on tarjota sovelluksille API-rajapinta ja jonopalvelu (Kuva 7). Managerin tehtäviin kuuluu viestin toimittaminen joko API-rajapinnan kautta paikalliselle sovellukselle tai *viestinvälityspalvelun (message moving service)* avulla etämanagerille (Broker-arkkitehtuurimalli). Tämän mallin etuna, on yksinkertainen toiminta-ajatus. API-kutsujen esimerkiksi CONN, OPEN, GET, PUT ja CLOSE –operaatioiden avulla voidaan sovellusten välillä vaihtaa viestejä. MOM on luonteeltaan asynkroninen. Ohjelmisto pyrkii nopeuteen ja jonojen käsittelymekanismin avulla luotettavuuteen. Sittemmin näihin järjestelmiin on liitetty transaktioidenhallintaominaisuuksia ja tietoturvatekijöitä [www010] .



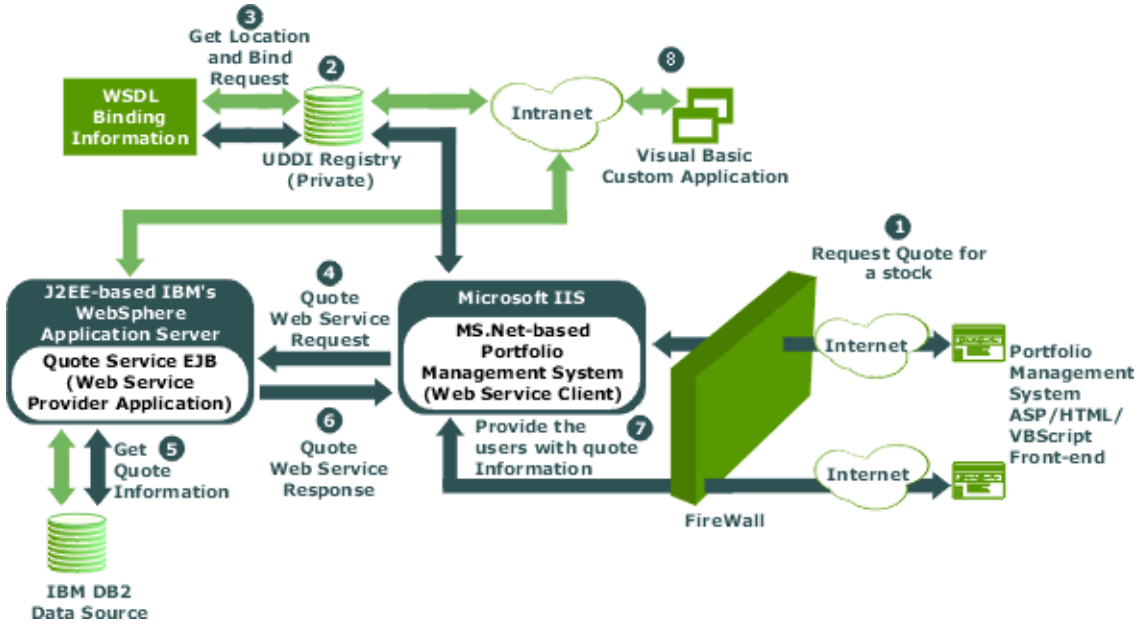
Kuva 7. Viestinvälitysprosessi [www010]

2.3.4 Verkkopalvelut (web services)

Verkkopalvelut tarjoavat yhteisen rajapinnan palvelupyyntöjen esittämiseen. Tarkoituksena on, että määrittely tarjoaa yksiselitteisen protokollan, jotta osapuolten kommunikointi onnistuisi helpommin. Tavoitteena on mahdollistaa sovellusten automaattinen kommunikointi. Verkkopalveluiden vahvuus on niiden selkeys ja helppous. Haittoina voidaan mainita, että määrittely ei tarjoa sisäänrakennettua käyttäjäntunnistusta, tietoturvaa, kuormantasausta tai virheistä toipumista. Nämä piirteet on joko rakennettava itse tai ne sisältyvät web-palvelimeen ja sovelluspalvelimeen.

Verkkopalvelun logiikka on se, että sovelluspalvelimella oleva sovellus, joka on usein jokin komponenttipohjainen tuote, haluaa lähettää sanoman toiselle sovellukselle, joka ei sijaitse omalla sovelluspalvelimella (palveluketjuesimerkki Kuva 8). Tällöin sovelluspalvelin lähettää kyselyn verkkopalveluihin käyttäen kanavan web-palvelin osaa. Kun UDDI-rekisteristä on paikallistettu palveluntarjoaja, noudetaan palveluntarjoajalta palvelua kuvaava WSDL-dokumentti. Sen jälkeen kun dokumentti on tutkittu, lähettää asiakassovellus SOAP-palvelupyynnön palveluntarjoajalle ja keskustelu alkaa osapuolten kesken. Verkkopalvelut toimivat Internetin kautta ja käyttävät http-protokollaa. Näin ollen sanoma voidaan lähettää salattuna esim. HTTPS:n avulla julkisen verkon kautta kohteena olevalle järjestelmälle sen palomuurin lävitse. Verkkopalveluiden käyttökohteeksi soveltuvat yritysten liiketapahtumat esim. sähköinen laskutus ja logistiikka sekä yrityksen sisäinen viestintä Intranet-verkossa esim. virheilmoitukset. Muina käyttökohteina ovat sovellusten hajauttaminen verkkoon [Jär02].

Liiketoimintaa palvelevat verkkopalvelut toteutetaan valmistajien *sovelluskehysten avulla (application frameworks)*. Sovelluskehykset tarjoavat kaikki tarpeelliset palvelut ja tietoturvan sekä toimivat kehitysalustoina tarjoten työkalut ja ohjelmointiympäristön hajautettujen monikerrossovellusten rakentamiseen [www004]. Käytetyimmät ratkaisut toimivat J2EE- tai .NET-alustoilla. Tarjolla on myös avoimeen koodiin perustuvia ratkaisuja.



Kuva 8. Esimerkki palvelupyynnön käsittelystä [www004]

Verkkopalvelun käyttö on perusteltua silloin, kun verkkopalvelut yksinkertaistavat arkkitehtuuria. Verkkopalvelut koostuvat joukosta yksinkertaisia avoimia standardeja, joten ne eivät sisällä sisäänrakennettuja toimintoja. Tämä helpottaa palvelujen integrointia. Tiedonvälitys on helppoa heterogeenisessa, massiivisessa hajautetussa ympäristössä. Verkkopalvelut vaativat paneutumista tietoturvaan ja sovellusten vikasietoisuuden ja skaalautuvuuden tutkimisen.

2.3.5 Oliopohjaiset hajautustekniikat

Liiketoimintakomponenttien välitason ohjelmistomarkkinat perustuvat kolmeen perusratkaisuun, joista selostetaan jäljempänä perusperiaatteet:

- CORBA on OMG:n määrittely laajojen liiketoimintasovellusten rakentamiseksi. Tämän määrittelyn mukaisia sovelluspalvelimia on markkinoilla useita.

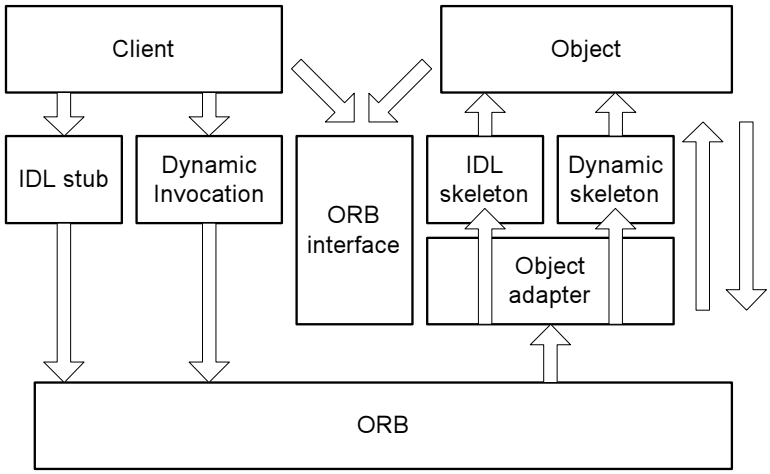
CORBA-arkkitehtuurin mukaiset järjestelmät soveltuvat parhaiten suuriin palvelukokonaisuuksiin, joissa se tarjoaa vakaan alustan sovelluksille.

- J2EE on Sunin tekemä ohjelmistoarkkitehtuurimäärittely, johon on liitetty mukaan myös verkkopalvelut. Olioteknologia perustuu Java Beans- ja EJB-komponenttimääritykseen.
- Microsoftin .NET-arkkitehtuuri on Microsoftin alustalle suunnittelu alusta verkkopalvelujen toteuttamiseksi, komponenttien kehittämiseksi ja tehtävien suorittamiseksi. Esitellään myös lyhyesti Microsoftin vanhempi COM-teknologia.

2.3.6 CORBA Common Object Request Broker –arkkitehtuuri

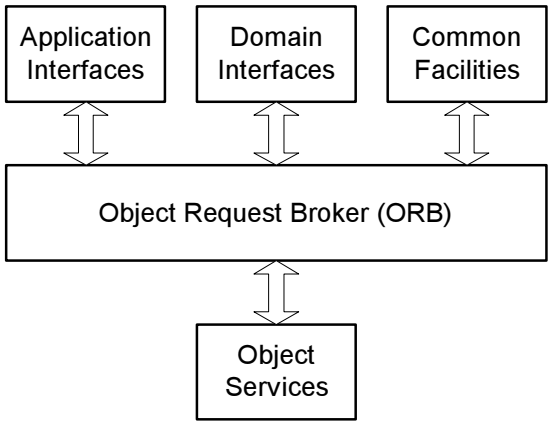
CORBA on OMG:n oliopyyntöjen määrittelyarkkitehtuurimalli eli määrittely, jonka mukaisesti ohjelmistovalmistajat voivat suunnitella ja valmistaa välitason ohjelmistoja. CORBA muodostaa alustan asiakkaan sovelluksille eli ohjelmistokomponenteille. CORBA tarjoaa oliopyyntöjen välittämiseen väyläksi *sanomanvälittäjää (object request broker ORB)*, joka on ohjelma ja jonka tehtävänä on välittää oliopyynnöt asiakaskomponentilta palvelukomponentille.

Komponenttien välisenä rajapintana käytetään *IDL:n (Interface Definition Language)* mukaista rajapintamäärittelyä. Tämä rajapintamäärittelykieli määrittelee tarkasti olioiden ja palveluiden metodit ja niiden parametrit [Tan02 s.493-]. Asiakas voi käyttää oliopalveluita staattisesti stub- ja skeleton-luokkien avulla, dynaamisesti olioviitteen avulla tai rajapintavaraston avulla silloin, kun on viite olioon, mutta ei ole tietoa olion tarjoamista rajapinnoista (Kuva 9) [NKV01 s.600]. CORBA konsepti pohjautuu *oliopohjaiseen malliin (object-oriented model)*, *avoimeen hajautuspohjaiseen ympäristöön (open distributed computing environments)* ja *komponenttien integraatioon ja uudelleenkäyttöön (component integration and reuse)* [MoR97 s. 9].



Kuva 9. CORBA:n yleinen arkkitehtuuri

CORBA:n toimintomalli (Kuva 10) koostuu neljästä ORB-väylän yhteen liittämästä osasta. *Oliopalvelut (objects services)* ovat kohdealueesta riippumattomia palveluita kuten. *nimipalvelu (naming service), kaupankäyntipalvelu (trading service)* sekä elinkaaripalvelu, tietoturva, transaktiot ja tapahtumienhallinta. Nimipalvelun tarkoitus on löytää etäoliot viitteen avulla. Kaupankäyntipalvelun avulla taas asiakas löytää olion sen ominaisuuksien mukaan. *Yleiset palvelu (common facilities)* ovat horisontaalisia rajapintoja käyttäjän sovelluksiin. *Kohdealueen rajapinnat (domain interfaces)* koostuvat kohdealuekohtaisista rajapinnoista. *Sovellusrajapinnat (application interfaces)* ovat sovelluskohtaisia palveluja liiketoimintasovelluksia varten.



Kuva 10. CORBA viitearkkitehtuurin palvelut
(OMG reference model achitecture) [www005]

Yhteistoiminnallisuus JAVA kanssa toteutetaan Javan rajapintamäärittelykielen IDLJ:n avulla. CORBA tarjoaa alustariippumattoman siirrettävyyden. CORBA tarjoaa yhteis-

toiminnallisuuden Microsoftin teknologioiden kanssa. CORBA/COM yhteistoiminnallisuuden avulla on mahdollisuutta integroida palvelinohjelmistoa asiakasohjelmistoihin OLE COM-alustoilla [MoR97 s. 84].

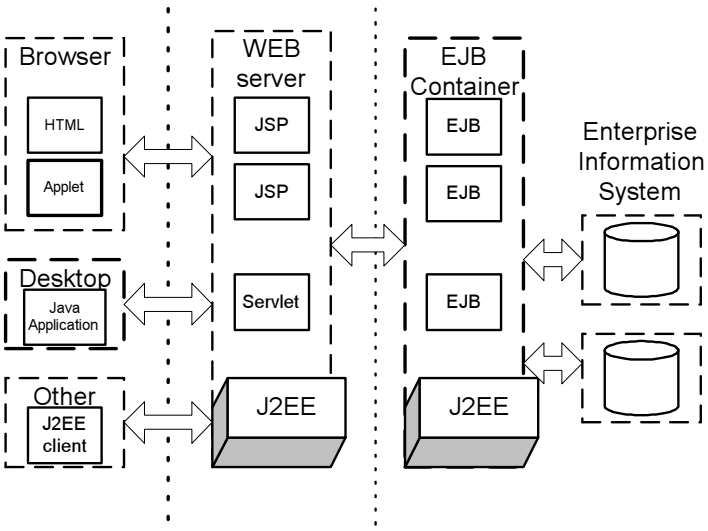
2.3.7 Java J2EE-arkkitehtuuri

JavaBeans on komponenttiteknologia komponenttien rakentamiseksi käyttäen Javaa. Papu voi olla yksittäinen luokka, jonka rajapinnat on rakennettu seuraavasti [SoW98 s. 398]:

- Olion attribuutteja voi lukea ja kirjoittaa metodien avulla (getAtribuutti(), setAttribuutti(Attribuutti), isAttribuutti()).
- Asiakas voi kutsua palveluita metodien avulla.
- Tapahtumat esittävät komponentin ilmoituksia.

Java tukee reflectio-ominaisuutensa ansiosta myöhäistä sidontaa katsomalla dynaamisesti rajapintoja ja metodeita ja kutsumalla niitä tuntematta olion staattisesti tyyppiä. JavaBeans-komponentti koostuu tyypillisesti useista luokista, joista kukin voi tarjota komponenttirajapintoja. Pysyvyys (persistence) saavutetaan sarajallistamisen (serialization) avulla. Komponentit pakataan JAR-tiedostoiksi jakelua varten [SoW98 s. 399].

Javan J2EE-arkkitehtuurin tarkoituksena on liiketoimintasovellusten toteuttaminen. *Sovelluspalvelin (EJB container)* noudattaa EJB-arkkitehtuuria, joka on Javan malli liiketoimintakomponentista, jossa sovellukset jaetaan kolmeen osaan: *komponentteihin (component), säiliöihin (container)* ja *yhdistimiin (connector)*. Säiliöitä on kahdenlaisia WEB-säiliö ja EJB-säiliö. Säiliöt ajetaan J2EE-palvelimella (Kuva 11).



Kuva 11. J2EE-arkkitehtuuri [www003]

J2EE-komponentti on riippumaton toiminnallinen ohjelmistoyksikkö. J2EE-komponentit koostuvat *sovellusasiakkaista* ja *sovelmista*, *Java-serveleteistä* ja *JSP-komponenteistä*, jotka ovat palvelimella suoritettavia WEB-komponentteja sekä *EJB-komponenteista,* jotka ovat sovelluspalvelimella ns. säiliössä (container) suoritettavia liiketoimintakomponentteja. EJB-palvelin tarjoaa palveluina EJB-komponentin luonti-, ylläpito- ja poistamismetodit, nimipalvelun, transaktioiden hallinnan, tietoturvan, tietokantayhteyden (JDBC) sekä huolehtii asiakkaan ja pavun välisistä yhteyksistä.

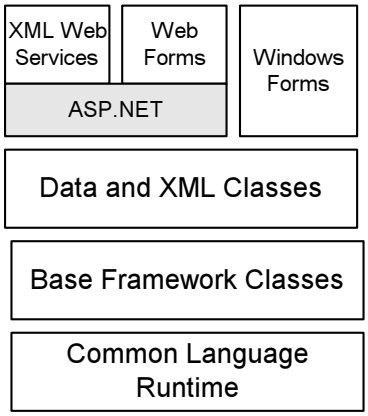
Enterprise Java Bean –komponentti (EJB bean) on määrittely komponenteista eli pavuista, jolla on kaksi rajapintaa. Home-rajapinnan avulla luodaan, etsitään ja poistetaan papuja. Remote-rajapinnan avulla voidaan käyttää pavun tarjoamia palveluja. Lisäksi papuun kuuluvat liiketoimintaoperaatiot toteuttava EJB-luokka sekä toimituskuvaus (deployment descriptor). Toimituskuvaus kuvaa pavun osat, transaktio-ominaisuuksia ja pavun turvallisuuteen liittyvää käyttäytymistä. Käyttäytymistä voidaan muokata toimitusvaiheessa [SoW98 s. 401]. Papuja on kolmenlaisia: sessio- (session bean), yksilö- (entity bean) ja viestiohjautuvia papuja (message driven bean) (kts. Liite 1).

2.3.8 Microsoft .NET arkkitehtuuri

.NET-arkkitehtuuri on Microsoftin palvelukonsepti verkkopalveluiden ja Web-pohjaisten liiketoimintasovellusten rakentamiseksi. Arkkitehtuuri pohjautuu vahvasti verkkopalveluihin, jossa palveluviestit toimitetaan verkkopalveluiden avulla. .NET-arkkitehtuurissa rajapinnat on suunniteltu uusiksi. Aiemmat Win32- ja DCOM-

rajapinnat eivät kuulu tähän teknologiaan. Olioteknologia perustuu .NET-komponenttiteknologiaan. Lisäksi aiempaan COM-komponenttiteknologiaan perustuvia komponentteja voidaan integroida tähän teknologiaan. Uudet rajapinnat mahdollistavat paremman toimivuuden myös muiden kuin Microsoftin käyttöjärjestelmien kanssa. Koska eri ohjelmointikielillä tehty koodi käännetään *CIL-välikoodiksi (common intermediate language)*, joka on natiivi kaikilla alustoilla, voidaan Windowsille tehtyjä ohjelmia ajaa myös Linuxilla. Täydellistä yhteensopivuutta ei kuitenkaan saavuteta, koska grafiikkakirjastoista ja tiedostonkäsittelystä johtuvat erot ovat edelleen olemassa käyttöjärjestelmien välillä [Hud03].

.NET Framework 1.1 ohjelmointirajapinta koostuu *Common Language Runtime (CLR)* ja .NET Framework –kirjastoista. CLR-kirjaston avulla ohjelmia voidaan kirjoittaa mm. käyttäen C, C++, C#, MS Visual Basic –kieliä. .NET Framework koostuu ASP.NET, Windows Forms ja ADO.NET –komponenteista. ASP.NET-komponentin avulla rakennetaan web-sovelluksia. Windows Forms helpottaa käyttäjärajapintojen rakentamista ja ADO.NET on tarkoitettu helpottamaan tietokantayhteyksiä (Kuva 12).



Kuva 12. .Net-arkkitehtuurin kerrosmalli

.NET-komponentteja Microsoft kutsuu palvelukomponenteiksi (service component). Palvelukomponentteja ajetaan sovelluspalvelimella. Tässä mallissa web-kerroksessa web-palvelimessa suoritetaan ASP.NET-skriptejä, joilla html-sivuille luodaan tarvittava dynaamisuus. Palvelimet vaativat toimiakseen, että niissä on common Language (CLR) ja .NET Framework –kirjastot. Tällä hetkellä puuttuvat vielä .NET-pohjaiset tietokannat, transaktiopalvelut ja viestijonopalvelut. Uudessa Windows Server 2003:ssa on ajoalusta .NET 1.1:lle ja UDDI-hakemistopalvelu. .Net-komponentit ovat *dynaamisesti linkitettyjä kirjastoja (DLL)*, joita kutsutaan ajonaikaisesti (run-time). Liiketoimintakomponentteina voidaan käyttää aiempia COM, COM+ ja DCOM-määritysten mukaisia

komponentteja, jotka ajetaan omina prosesseinaan/säikeinään lisäämällä tarvittava COM-komponentin tyyppikirjasto (type library) viitteenä .NET-ympäristöön esimerkiksi käärimällä [www002].

Vanhempi komponenttimalli perustuu COM-komponenttimalliin, jossa perusrajapintana ovat Win32-rajapinta ja sovellustason rajapintana COM+ 1.5/DCOM. Microsoftin määrittely on binääristandardi rajapinnoille. Asiakas viittaa olioon rajapinnan kautta, jota esittää osoitin solmuun (pointer to node), joka puolestaan viittaa rajapintafunktiotaulukkoon. Oliolla voi olla tila tai se voi olla tilaton. Komponentti tarjoaa joukon rajapintoja, johon asiakas viittaa edellä kuvatulla tavalla. Jokainen rajapinta tukee QueryInterface-rajapintaa, joka tiedustelee muita rajapintoja rajapintojen yksilöllisten tunnusten avulla. Julkaistua rajapintaa ei voi muuttaa. Komponentilla on rajapinta IUnknown, jonka avulla komponentti tunnistetaan.

COM-teknologia koostuu COM/DCOM-, COM+, MSMQ-, MTS- ja ActiveX-teknologioista. COM-malli on tarkoitettu asiakas- ja palvelukomponentin vuorovaikutuksen toteuttamiseen ja toteuttaa client/server-arkkitehtuurin [SzC98 s.194-217] COM-mallin lähtökohtana on asiakas ensin –periaate toisin kuin esimerkiksi CORBA:ssa, jossa periaatteena on palvelin ensin –periaate [MoR97 s. 192].

Viestin välityspalvelin (Microsoft Message Queuing MSMQ) on MOM-tyyppinen väliohjelmisto viestien välittämiseksi. Microsoft Transaction Server (MTS) -palvelin tukee transaktioita COM-pohjaisissa ohjelmissa. Säiealtaan (threads of pool) avulla voidaan kontrolloida suorituskykyä silloin, kun suuri joukko kyselyitä kohdistuu COM-olioihin. MTS tarjoaa tuen transaktioille, automaattisen säikeidenhallinnan sekä olioiden uudelleenkäytön. COM sisältää OLE-mallin dokumenttien yhdistämiseksi sekä ActiveX-kontrollit, jotka ovat joukko COM rajapintastandardeja ja joiden avulla voidaan sallia tapahtumien hallinta kontrollien avulla. ActiveX-kontrollien ominaisuudet ovat vastaavat kuin JavaBeanseilla [SoW98 s. 401-]. Komponenttimallien vertailua on esitetty Taulukossa 1.

Taulukko 1. Vertailua ActiveX/OLE/COM ja CORBA/OpenDoc [JGP97 s. 195]

Ominaisuus	ActiveX/OLE/COM	CORBA/OpenDOC
komponenttisysteemi	Tarvitaan erillinen komponenttisysteemi jokaista komponenttijoukkoa varten esimerkiksi taulukkolaskentaa varten.	ORB on yksi komponenttisysteemi, ja jokainen lisäpalvelujoukko on erillinen komponenttisysteemi. Komponenttisysteemit ovat organisoidut alisysteemeiksi tai paketeiksi (service packages).
sovellusten rakentaminen	Rakennetaan sovellukset niin, että OLE- ja DCOM-komponentteja käytetään OLE servereinä sekä ActiveX-komponentteja selaimissa ja muissa säiliöissä luoden uutta toiminnallisuutta skriptikielien avulla.	Komponentteja käytetään sijaintiriippumattomasti ORB:n kautta. Käytetään OpenDoc-komponentteja silloin, kun halutaan rakentaa rikas käyttäjärajapinta (rich UI) ja dokumenttisuuntautuneita sovelluksia.
fasadi	Fasadit sisältävät rajapintamäärittelyt ja ActiveX/OCX –komponentteja, joita voidaan mukauttaa.	Fasadit sisältävät IDL määrittelyt.
rajapinta	OLE IDL	OMG IDL
muunneltavuusmekanismit	parametrisointi, generointi, aggregaatio	perintä ja aggregaatio
hajautusmekanismi	DCOM, Internet	ORB
välitasokerros	DCOM, ActiveX, OLE	ORB, oliopalvelut, yleiset palvelut (common facilities), komponentit yhdistettyjä dokumentteja varten (compound documents)
käyttöjärjestelmätaso	Malli vaatii Windows-käyttöjärjestelmän.	ORB tarjoaa alustariippumattomuuden.

2.3.9 Yhteenveto hajautuksesta ja välitason ohjelmista

Liiketoiminnan kehittäjän kannalta liiketoimintajärjestelmäkehitys tapahtuu komponenttipohjaisten ohjelmistoarkkitehtuurien avulla. Tällöin ohjelmistokehitys tapahtuu pääasiassa hajautettujen oliopohjaisten systeemien muodossa. Komponenttituotanto pohjautuu useimmiten olio-paradigmalle, mutta näin ei suinkaan tarvitse olla.

Merkille pantavaa on, että hajautettujen järjestelmien suunnitteluperiaatteet ovat pysyneet samoina 80-luvulta saakka. Mullender on luetellut periaatteet hajautettujen järjestelmien suunnittelemiseksi seuraavasti [Mul89 s. 446-447]:

- Määritellään vastuut eli mistä mikin järjestelmän ja ohjelman osa on vastuussa.
- Kytkeydytään vain, kun on tarvetta. Tai toisin päin pidä auki riittävästi yhteyksiä, jotta saavutetaan vaadittu tehokkuus.
- Tuetaan vain tarpeellisia palveluja. Kun systeemit eivät ole kytkettyinä, ylläpidetään palveluntarjonta minimissään.
- Järjestelmän tulee tarjota tunnistuspalvelut ja pystyä monitoroimaan omaa toimintaansa.
- Järjestelmän on kyettävä toipumaan virheestä siihen tilaan, jossa se oli virheen sattuessa.
- Järjestelmä on suunniteltava *skaalautuvaksi (scalability)*. Skaalautuvuus tarkoittaa esimerkiksi järjestelmän kykyä vastata asiakkaiden kyselyihin, joiden määrä voi vaihdella eri aikoina (Liite 1).
- Vältetään mekanismeja, joista voi tulla paljon virheitä (ylikuormitus). Vältetään myös mekanismeja, joiden toiminta ei ole tarkkaan tiedossa. Yksinkertaiset ja tunnetut mekanismit ovat luotettavia.
- *Sitkeys (robustness)* on kallista. Sitkeys tarkoittaa järjestelmän kykyä selvitä sen toimintaympäristössä tapahtuvista muutoksista [www013]. Tehokkuus kärsii sitkeydestä. Tehokkuuden ja sitkeyden välillä olisi löydettävä sopiva balanssi.

Hajautuksen avulla yritysten informaatiojärjestelmiä on voitu kehittää niin, että yhä uusia toimintoja on saatettu tietojenkäsittelyn piiriin. On otettu käyttöön monikerrosarkkitehtuurit, jolloin kyetään sovelluspalvelimien avulla palvelemaan erilaisia asiakkaita. Tiedon esitys ja tiedon mallintaminen kyetään erottamaan toisistaan, jolloin käyttöliittymät voidaan toteuttaa erilaisilla päätelaitteilla (terminaali, PC, käsitietokoneet, mobiilit päätelaitteet).

Yritysten, erilaisten järjestelmien ja alustojen integroimiseksi on tarjolla useita tekniikoita. Verkkopalvelut ovat yksinkertainen ja helposti rakennettava menetelmä yritysten järjestelmien integroimiseksi. Muita varsinkin 90-luvulla käytettyjä tapoja ovat olleet MOM-tyyppiset järjestelmät sekä Tapahtumamonitorointi-ohjelmat.

Hajautetut järjestelmät toteutetaan usein sovelluspalvelimia käyttäen. Tällöin valinta tehdään usein joko Java- tai Microsoft-teknologian välillä. Java-pohjaiset ratkaisut voivat pohjautua joko CORBA-määrittelyyn tai EJB-komponenttiteknologiaan. Microsoftin .NET-teknologia on uusi ja siinä on kehittynyt virtuaalikone, luokkakirjasto ja xml-tuki. Microsoft soveltuu mainiosti ratkaisuksi niille pienille ja keskisuurille yrityksille, jotka toimivat Microsoft-ympäristössä. Suurten toimialapohjaisia ratkaisuja toteuttavien yritysten ratkaisu on usein CORBA, koska määritys on laajalti käytössä suurissa organisaatioissa ja koska sovelluspalvelin- ja kehitysympäristötoimittajia on runsaasti. Jos joudutaan rakentamaan oma liiketoimintaympäristö ja luomaan toiminnallisuus itse suuressa ympäristössä, on ratkaisu luultavasti EJB, koska EJB-ratkaisu vaatii CORBA:a vähemmän resursseja. Jos taas sovelluksissa on paljon työpöytäintegraatiota ja toimistossa käytetään paljon Microsoftin teknologiaa, tuntuisi .NET paremmalta ratkaisulta.

3 Komponentit – uudelleenkäyttö ja liiketoimintakomponentit

3.1 Yleistä

Komponenttiteknologia syntyi 90-luvulla vastauksena niihin ongelmiin, joita yhä monimutkaisemmaksi käyvä ohjelmistoteknologia aiheutti. Pelkkä olioihin pohjautuva teknologia, vaikka se muuten paransikin ohjelmistojen laatua, ei riittänyt lisäämään ohjelmistojen uudelleenkäytettävyyttä ja ohjelmistoprojektien tehokkuutta. Ohjelmistotuotanto on siirtynyt komponenttipohjaiseen ajattelutapaan. Nähdään, että komponenttien avulla ohjelmistotuotantoon saadaan samaa tehokkuutta kuin muillakin teollisuuden aloilla. Komponenttien avulla tuotannossa voidaan muodostaa logistisia ketjuja ja ulkoistaa tuotantoa alihankkijoille tai hankkia osia markkinoilta. Tavoitteena on, että yhä suurempi osa komponenteista voitaisiin ostaa suoraan hyllytavarana *COTS (commercial off the shelf) -komponentteina,* jolloin tarjolla olisi laadukkaita, nopeasti saatavia ja käyttöönotettavia, monikäyttöisiä ja edullisia komponentteja.

Komponenttiteknologialle on asetettu vaatimuksia, jotka sen on täytettävä. Teknologian tulee alentaa valmistuskustannuksia, lisätä joustavuutta vastata teknologian muutoksiin, tukea modulaarisuutta ja järjestelmien integrointia sekä myös lisätä standardointia. Komponenttiteknologialla sanotaan olevan etuja, kuten parempi monimutkaisuuden hallinta, kyky parantaa prosessienhallintaa modulaarisuuden ansiosta, kyky kattaa koko ohjelmiston elinkaaren sekä kyky alentaa tuotteen valmistusaikaa (time-to-market) [HeS99 s. 21].

Näiden tavoitteiden saavuttamiseksi tarvitaan integroidut työkalut ja kehitysympäristöt. On osattava allokoida sovellustuotannon vaatimat henkilöstöresurssit ja osattava organisoida sekä liiketoimintaprosessit että tuotanto-organisaatio kilpailukykyiseksi. Lisäksi tarvitaan paremmat ohjelmistojen toteutuskäytännöt (software engineering practises) ja menetelmät arkkitehtuurin ja uudelleenkäytön edistämiseksi.

3.2 Komponenttien luokittelu ja rakeisuus

Ohjelmisto koostuu osista – *elementeistä (element)*. Elementti voi olla käännösaikainen yksikkö *moduuli (module)* tai *komponentti (component)* (kts. liite 1). Komponentti on

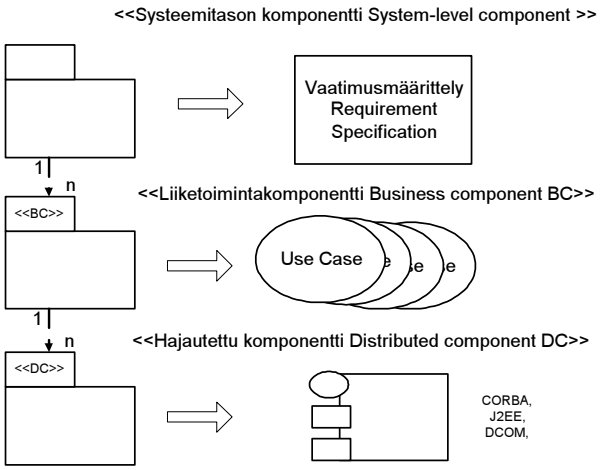
itsenäinen ohjelmiston osa, joka kuvaa ohjelmiston tilaa ajonaikana. Komponentti on olemassa ajonaikana [CleF02]. Komponentti on olemassa abstraktina käsitteenä myös analysointi- ja suunnitteluaikana. Komponentti on tarkoitettu jakelua varten. Komponentti tarjoaa palvelunsa vain *rajapintojensa (interface)* kautta. Rajapinta kätkee toteutuksen kutsujalta (caller). Rajapinta on ainoa *riippuvuus (dependency)* osapuolten kesken. Rajapinta luo *sopimuksen (contract)* osapuolten välille [HeS99].

Komponenttiarkkitehtuurin keskeisimmät tehtävät ovat [SoW98 s. 29]:

- Vuorovaikutusmallit, joiden (interchange) avulla esitellään komponentit käyttäen staattisia malleja.
- Abstraktien yhdistimien (connector) määrittely komponenttien välille, mikä suoritetaan mallintamalla komponenttien välistä yhteistoimintaa esittämällä kaava vuorovaikutukselle. Kaava voidaan kohdistaa mille tahansa tietyn tyyppiselle komponenttiparille.
- Teknologiset valinnat, joita ovat muun muassa väliohjelmistostandardin valinta esim. CORBA sekä kommunikaatiotavan päättäminen komponenttien toisiinsa liittämisen mahdollistamiseksi.

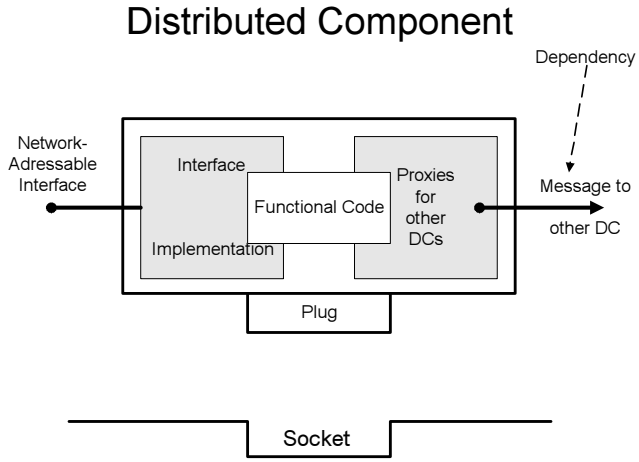
Komponenttien yhteistoiminnallisuus (interoperability) muodostetaan *komponenttisysteemien* avulla. Komponenttisysteemi voidaan rakentaa esimerkiksi käyttäen runkona komponenttikehystä, johon yksittäiset komponentit liitetään. *Komponenttikehys (component framework*) on määritelmä, johon voidaan liittää (plug) yksittäisiä komponentteja, jotka täyttävät komponenttikehyksen määrittelemät specifikaatiot. Komponenteista rakennetaan *komponenttipakkaus (component kit)* siten, että pakkauksen komponentit noudattavat yhteistä arkkitehtuuria ja yhteisiä spesifikaatioita. Komponenttipakkausten avulla voidaan lisätä komponenttien uudelleenkäyttöä [SoW98 s. 27-28].

Komponenttisysteemin tehtävät muodostuvat osatehtävistä. Komponentit on luokiteltu perustuen niiden *rakeisuuteen (granularity*) ja kokoon. Herzum on luokitellut komponentit niiden rakeisuuden mukaan: *komponenttisysteemiin (business component system eli system-level component), liiketoimintakomponentteihin (business component BC) ja hajautettuun komponenttiin (distributed component DC). Komponenttisysteemien liittoumasta (federation of system-level components)* on kyse silloin kun kaksi tai useampaa komponenttisysteemiä toimii vuorovaikutuksessa keskenään [HeS99 s. 36] (Kuva 13).



Kuva 13. Komponenttien rakeisuus ja vastaavuus suunnittelukäsitteisiin [HeS99 s. 37 mukaellen]

Mallissa hajautettu komponentti on pienimmän rakeisuuden omaava komponentti. Hajautettu komponentti on installoitu komponentin ajoympäristöön. Se voi olla esimerkiksi EJB-papu. Ajoympäristöä kutsutaan *komponenttivirtuaalikoneeksi (business component virtual machine BCVM)*. Virtuaalikone sisältää vaadittavan ohjelmistoalustan, jotta komponentti voi suorittaa tehtävänsä. Hajautettu komponentti (Kuva 14) koostuu tarjotusta rajapinnasta, pyydetyistä rajapinnoista (proxy), toiminnallisesta koodista (functional code) sekä liitin-osasta (plug) ja liitoskohdasta (socket), jolla komponentti liittyy ajoalustaan. Ajoalustana toimiva välitason ohjelmisto tai sovelluspalvelin tarjoavat komponentille liitoskohdan, jolla komponentti voi olla yhteydessä muihin komponentteihin tai rajapintoihin [HeS99], [Myk00 s. 54-60] (kts. Liite 1).



Kuva 14. DC Distributed Component – hajautettu komponentti

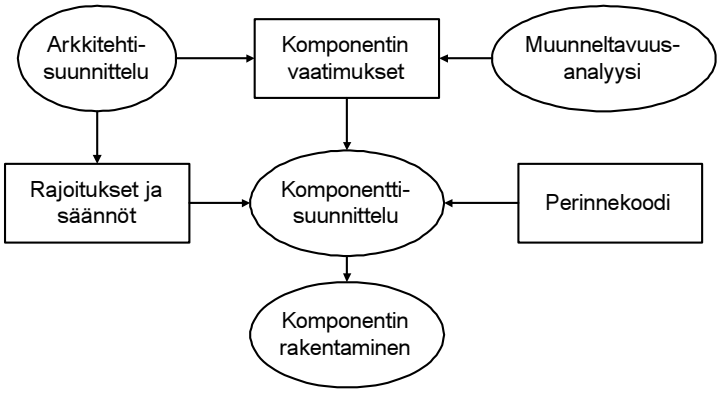
Liiketoimintakomponentti koostuu yhdestä tai useammasta hajautetusta komponentista. Liiketoimintakomponentti hoitaa yleensä tietyn rajatun liiketoimintaa palvelevan tehtä-

vän ja on jäljitettävissä johonkin käyttötapaukseen. Liiketoimintakomponentilla on erilaisia tehtäviä. Se voi olla *varuskomponentti (utility component), entiteettikomponentti (entity component), prosessikomponentti (processing component) tai apukomponentti (auxiliary component)* [Myk00 s. 59], [HeS99 s. 174] (kts. Liite 1).

Komponenttisysteemi koostuu hierarkisesti koostetuista liiketoimintakomponenteista. Hierarkiataso muodostuu prosessi (process), entiteetti (entity) ja varuste (utilities) -tasoista. Komponenttisysteemissä on yleensä ulkopuolisia yhteyksiä varten *yhdyskäytävänä (System gateway)* toimiva Fasadi-komponentti (Facade), johon on keskitetty järjestelmän kommunikointi ulkopuolisten järjestelmien kanssa. Tällöin komponenttisysteemi muodostaa systeemitason komponentin. Yhteystoiminnallisuus muiden järjestelmien kanssa tulisi huomioida jo suunnitteluaikana, jolloin kaikki vaatimukset voidaan huomioida paremmin. Jälkeenpäin voidaan yhteistoiminnallisuutta komponenttisysteemien välille rakentaa adapterin avulla. [HeS99 s.157-], [Myk00 s.60]. Liitteessä 8 on esitetty Sovitin (Adapter)-suunnittelumalli sovittimen rakentamiseksi komponenttien välille.

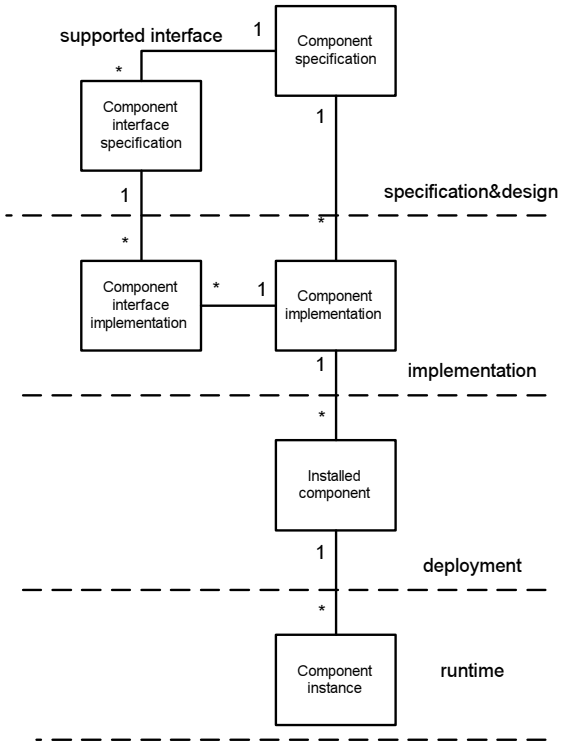
3.3 Komponenttien kehittäminen

Komponentin suunnittelussa yhdistetään komponentin vaatimukset, rajoitukset ja säännöt, perinnekoodi (lecacy code) sekä integroidaan komponentti osaksi komponenttisysteemiä. Komponentille asetettavat vaatimukset saadaan arkkitehtuurisuunnittelun tuotoksena. Sen lisäksi komponentille on tehtävä *muunneltavuusanalyysi* (Kuva 15) [Bos00 s.216]. Muunneltavuusanalyysin tarkoituksen on selvittää, millaisia ominaisuuksia komponentilta vaaditaan silloin, kun sitä ollaan sovellusta rakennettaessa asentamassa toimintaympäristöönsä.



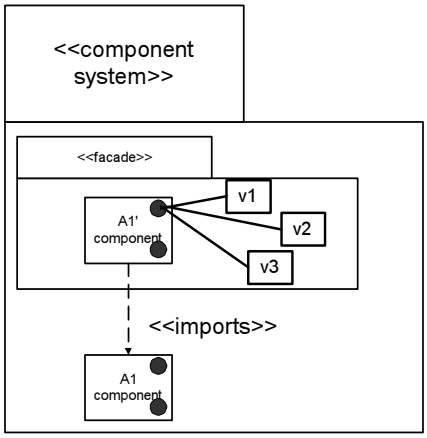
Kuva 15. Komponentin kehitysprosessi [Bos00 Figure 48 s.217]

Komponentilla on elinkaarensa aikana useita olomuotoja alkaen aina määrittelystä ja jatkuen tuotantokäyttöön saakka. Kuvassa 16 esitetään komponentin elinkaari ja sen rajapintojen muotoutuminen suunnittelun eri vaiheissa mukaellen Cheesmanin [ChD01] esittämiä ajatuksia. Määrittely- ja suunnitteluvaiheessa komponentille luodaan vaatimusmäärittely (component specification) ja kuvataan komponentin rajapinnat (component interface specification). Rakennusvaiheessa rakennetaan rajapintamäärittelyille toteutuksia (component interface implementation) ja rakennetaan komponentti (component implementation). Komponentti julkaistaan jakelukelpoisena pakettina ja asennetaan ympäristöönsä (installed component). Komponentti voidaan asentaa haluttaessa useina toteutuksina toimintaympäristöönsä. Lopulta komponentti toimii ajonaikaisessa ympäristössä (runtime environment) komponenttina (component instance). Samaa komponenttia voivat käyttää useat prosessit, jolloin järjestelmän suorituskyky säilyy korkeana tai jokainen prosessi voi alustaa komponentista oman instanssinsa. Tällöin voidaan jokaista prosessia varten luoda oma tietosaareke (isle of data), jonka avulla voidaan taata tiedon pysyvyys. Komponentti voidaan koota joko käännösaikana, linkkausaikana, alustusaikana tai suoritusaikana. Muut ovat binäärisiä paitsi käännösaikainen komponentti, joka on lähdekoodina [Kos00 s.219-].



Kuva 16. Komponentin elinkaari ja olomuodot (mukaellen [ChD01])

Komponenttia voi käyttää joko sellaisenaan (as-is) eli komponentti on konkreettinen tai sitä voi mukauttaa käyttötarpeen mukaan. Viimeksi mainitussa tapauksessa komponenttia kutsutaan abstraktiksi (abstract). Mukauttaminen tapahtuu komponentin *muunneltavuuspisteiden kautta (variation points)* [JPG97 s.95-97]. Kuvassa 17 on esitetty komponentin muunneltavuuspisteitä mustina täplinä ja mahdollisina käytettävinä muuntelun arvoina pisteeseen liittyvät elementit v1-v3. Muunneltavuuspiste tarkoittaa sitä kohtaa komponentissa, jossa komponentin ominaisuuksia voidaan mukauttaa lisäämällä muunneltavuuspisteisiin tarvittava muutos. Muunneltavuuspiste voi olla luokka, tyyppi tai käyttötapaus [JGP97]. Komponentti voi tarjota muunneltavuutena esimerkiksi toimintaalustan valinnan, käyttöliittymän valinnan tai sen avulla voidaan valita vaikka sovelluksessa käytettävä valuutta.



Kuva 17. Komponenttisysteemi, Facade-komponentti ja muunneltavuuspisteitä [JPG97 s. 100]

Komponentin vaatimuksia määriteltäessä on tutkittava, millaista muunneltavuutta komponentilta vaaditaan. Muunneltavuusanalyysin (variability analysis) tarkoituksena kartoittaa komponentille vaadittava muunneltavuus, jotta sitä voitaisiin käyttää mahdollisimman monessa tarkoituksena [Bos00 s. 225]. Muunneltavuus luodaan *perinnän, laajentamistekniikan käytön, konfiguraation, parametrien, mallien tai generoinnin avulla* (Taulukko 2).

Perinnän (inheritance) avulla komponenttia voidaan uudelleen käyttää niin, että luokan ominaisuuksia voidaan muotoilla uudelleen. Komponentti voidaan toimittaa esimerkiksi kehyksen kaltaisena niin, että komponentin suunnittelija kirjoittaa abstraktin luokan, jota voidaan käyttää *virtuaalioperaatioissa* perimällä ko. luokka ja tarkentamalla luokan toiminnallisuus. Virtuaalioperaatio voidaan kirjoittaa joko *rungon* kanssa tai ilman. Jos siinä on runko, on kyseessä *kiinnitetty operaatio*. Perintää käytettäessä komponentin ominaisuuksia voidaan erikoistaa *valkolaatikko-tekniikkaa (white-box techniques)* käyttäen [Bos00 s 225], [Kos00 s. 90].

Taulukko 2. Muuntelumekanismit [JGP97 s.102]

mekanismi	muunneltavuuspisteen tyyppi	*variantin* tyyppi	mekanismin suositeltava käyttö
perintä (inheritance)	virtuaalioperaatio	aliluokka tai alityyppi	erikoistaminen ja valittujen operaatioiden lisääminen, muiden operaatioiden säilyttäminen
laajennukset (extensions)	laajennuspiste	laajennus	On oltava mahdollista liittää useita variantteja kuhunkin laajennuspisteeseen samanaikaisesti.
käyttö (uses)	käyttöpiste	käyttötapaus	Uudelleen käyttämällä abstraktia käyttötapausta luodaan erikoistettu käyttötapaus.
konfiguroiminen (configuration)	konfiguraatiokohta (configuration item slot)	konfiguraatiokohta	Valitaan vaihtoehtoisia toimintoja ja implementaatioita.
parametrit (parameters)	parametri	rajoitettu parametri	On olemassa useita variaatiopisteitä jokaiselle muuntelupiirteelle.
mallin installointi (template instantiation)	mallin parametri	mallin instanssi	tekemällä tyypin mukauttaminen tai valitsemalla vaihtoehtoinen pala koodia
generointi (generation)	parametri tai skriptikieli (language script)	rajoitettu parametri tai ilmaisu	tekemällä laaja-alaista generointia yhdestä tai useammasta tyypistä tai luokasta ongelmaspesifisellä kielellä

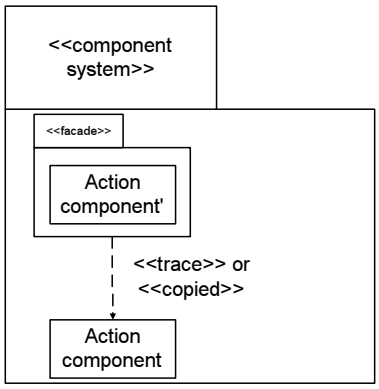
Laajennukset (extensions) ovat pieniä käyttötapauksia, tyyppejä tai luokkia, jotka laajentavat tai muokkaavat sen käyttötapauksen, tyypin tai luokan käyttäytymistä, johon ne ovat liittyneet. *Laajentamistekniikkaa (extension)* käytettäessä komponentille määrätään muunneltavuuspiste esimerkiksi *Strategia-suunnittelumallia (Strategy -design pattern)* käyttäen. Tällöin komponentin käyttäjä voi valita tarkoitukseen sopivan toiminnallisuuden. Toinen mahdollisuus laajennusten toteuttamiseen on *Kuorruttaja (Decorator)*-suunnittelumallin käyttö. Liitteessä 8 on esitelty tässä tutkielmassa mainitut suunnittelumallien luokkakaaviot ja lähdeviittaukset.

Konfiguraatio (configuration) –tekniikan avulla komponentin tarvitsema muunneltavuus tarjotaan käyttäjälle rajapintana, johon käyttäjä voi asettaa tarvittavat parametrit toiminnallisuuden mukauttamiseksi käyttäjän tarpeisiin [Bos00 s. 226]. *Component Configurator –suunnittelumallin* avulla komponentille voidaan ajonaikana sitä alustettaessa määritellä haluttuja ominaisuuksia [ScR01 s. 75]. *Mallien alustus (template instan-*

tiation) –tekniikka puolestaan mahdollistaa komponentin mukauttamisen sovellusspesifisesti niin, että se huomioi sovelluksessa käytetyt tietotyypit. *Komponentin generoinnista (component generation)* on kyse silloin, kun komponentin käyttäjä suunnittelee käyttötarkoituksen mukaisen määrittelyn soveltuvalla ohjelmointikielellä. Tämä määrittely generoidaan kääntäjällä sopivaksi komponentiksi, jota voidaan käyttää sovelluksessa.

Komponentin mukauttaminen uusiin käyttötarkoituksiin voidaan tehdä *kääreen (Wrapper Facade -suunnittelumalli)* avulla. Käärimistä käytetään liitettäessä perinnejärjestelmiä muihin järjestelmiin. Muita mukauttamistekniikoita ovat *Leikkaa ja liimaa (copy-paste)* –tekniikka, jossa komponentin koodi kopioidaan kehitettävään koodiin. Kääre toteuttaa mustalaatikko-mallia, kun taas leikkaa ja liimaa –malli on esimerkki valkolaatikko-mallista.

Fasadi (system gateway) -komponentti on paketoitu komponenttien alijoukko tai viittaus komponenttisysteemistä valittuihin komponentteihin. Jokainen fasadi tarjoaa julkisen pääsyn ainoastaan niihin osiin komponenttisysteemiä, jotka on valittu uudelleenkäytettäväksi (Kuva 18) [JPG97 s. 89]. Yksittäisten komponenttien rajapinnat toteutetaan *Facade*-suunnittelumallilla.



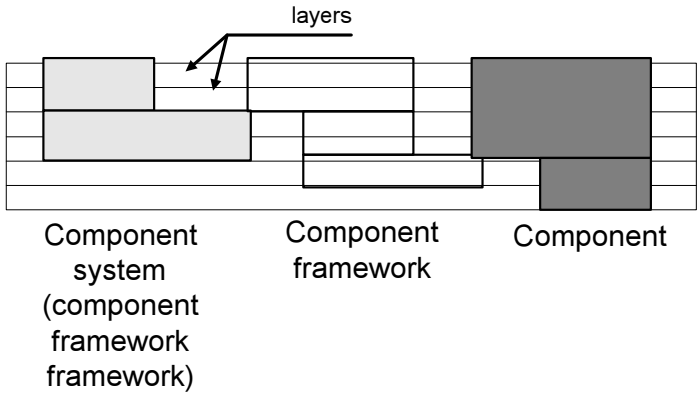
Kuva 18. <<Facade>>-komponentti komponenttisysteemissä [JPG97 s. 90]

Komponentti voidaan koostaa *Composition* tai *Whole-Part* –suunnittelumallin avulla. Käyttöliittymän tiedon ja esityksen erottamiseen käytetään *Model-view-controller* -mallia. Gamman mukaan perintää käytetään liikaa ja se vähentää joustavuutta suunnittelussa. Gamma suosittelee koostamista, koska se on joustavampi kuin staattinen perintä. Delegointia voidaan käyttää aina perinnän sijasta ja se vähentää erillisten luokkien määrää[JPG97], [GaH95].

3.4 Kehykset ja komponenttisysteemin metamalli

Kehys (Framework) on joukko luokkia, jotka muodostavat abstraktin suunnitteluratkaisun toisiinsa liittyville ongelmille (family related problems) [Bos00 s.242]. Kehysten koostamisessa noudatetaan *ns. Hollywood-periaatetta***.** Käyttäen abstrakteja luokkia ja rajapintoja saadaan kehyksiin uudelleenkäytettävyyttä. Mallit muodostavat mikroarkkitehtuureita (microarchitecture) [SzC98 s. 142]. Kehysten avulla muodostetaan alisysteemejä (subsystems). Komponentti voi koostua useasta alisysteemistä. Kehysajattelussa kehyksen voi muodostaa eri abstraktio- tai karkeustasolla oleva rakennelma, jossa kehys tarjoaa osan toiminnallisuutta ja kehyksen soveltajalle ns. kuumien pisteiden (hot spot) kautta mahdollisuuden lisätä kehykseen toiminnallisuutta ja muunneltavuutta.

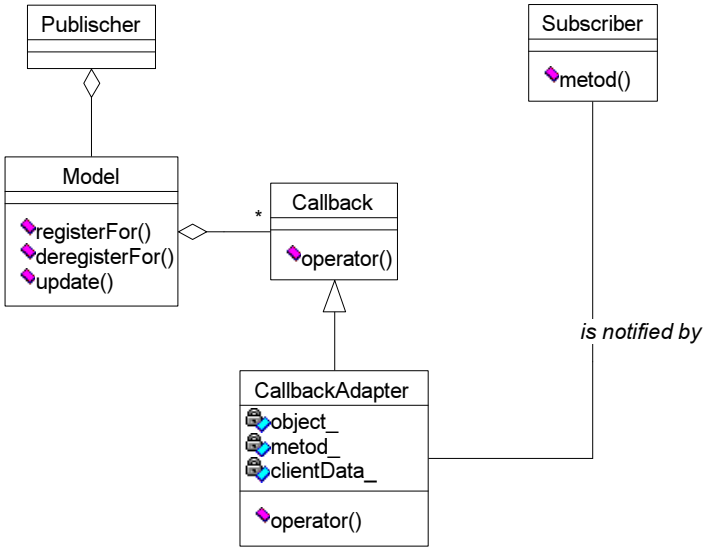
Szyperski esittelee *kerrostetun komponenttiarkkitehtuurin (tiered component architecture)*, jossa metamallin avulla kuvataan komponenttien *hierarkioita (hierarchy)*. Komponentit voivat olla osana arkkitehtuuria jossain kerroksessa (layers). Kerrostyylin avulla voidaan hallita komponenttien välisiä suhteita ja vastuita. Komponenttisysteemin *meta-malli (meta model)* koostuu *tasoista (tiers)*. Meta-mallissa on kyseessä *kaksitasomalli (second-tier)*, jossa *komponenttisysteemi (component system eli component framework framework)* koostuu *komponenttikehyksistä (component framework)*, jotka puolestaan koostuvat komponenteista. Meta-mallissa kukin taso (tier) voi kommunikoida horisontaalisesti oman tasonsa kanssa tai vertikaalisesti oman ylemmän tai alemman tason kanssa (Kuva 19). Esimerkkinä meta-mallista ovat monikerrokselliset asiakaspalvelin (multitier client-server applications) -sovellukset [SzC98 s. 277].



Kuva 19. Komponenttisysteemin metamalli [SzC98 s. 277]

Kehys tarjoaa perustoiminnallisuuden. Toiminnan räätälöinti voidaan tehdä joko periytymisen kautta tai käyttäen rajapintoja hyväksikäyttäen valmiita suunnittelumalleja. Sopivia suunnittelumalleja ovat esimerkiksi Strategia-suunnittelumalli, joka tarjoaa mahdollisuuden myöhäiseen *sidontaan (late binding),* jolloin ajon aikana voidaan valita haluttu toiminnallisuuden toteuttava olio ja sen metodi. Tällöin kehyksessä esitellään ja käytetään rajapintaa, johon suunnittelija valitsee vaihtoehdoista toteutuksen. Mustalaatikkotapauksessa erikoistajalle tarjotaan tieto vaihtoehdoista käytettäväksi. Valkolaatikkotapauksessa suunnittelija voi lisäksi luoda rajapinnan toteutukseen omia vaihtoehtoja. Etuna tämän mallin käytössä verrattuna staattiseen perintään on se, että esimerkiksi sovelluksen kokoaja voi valita sovellukseen sopivan vaihtoehdon. Jos halutaan lisätä yksittäiselle oliolle uutta toiminnallisuutta dynaamisesti tarvittaessa, kannattaa käyttää Kuorruttajaa (Decorator) Strategian tilalla.

Publish-Subscribe -mallin avulla voidaan laukaista tapahtumia, toimittaa ilmoituksia mm. tilatietoja muille olioille. Tätä mallia voisi käyttää kehyksessä tarvittavan muuntelun aikaansaamiseksi siten, että tarjotaan komponentille rajapinta, jonka avulla se voi ilmoittautua tilaajaksi. Koska on tarvetta toimittaa tapahtuma usealle tilaajalle, on parempi toteuttaa linkitys koukkumetodiin (hook methods) Strategia-mallin avulla sen sijaan, että käytettäisiin Operaatiorunko (Template Method)-suunnittelumallia. *Takaisinkutsu (Callbacks)* -mallissa Callback-elementti määrittelee olion toiminnallisen rajapinnan, jota Publisher kutsuu ja Subscriber noteeraa. CallbackAdapter saa Callback:n ilmentymän ja lähettää tapahtuman Subscriber:n metodille. Model rekisteröi, poistaa Callback:n tilaajat sekä lähettää tiedon tapahtumasta Callback:lle (Kuva 20).



Kuva 20. Takaisinkutsu (Callbacks) [FaJ00 s. 574]

Abstrakti tehdas (Abstract Factory) –suunnittelumallin avulla voidaan luodaan rajapintoja määrittelemättä konkreettisia luokkia. Olion luontioperaatio ja olion ominaisuudet määritellään dynaamisesti ajonaikana.

Tehdasmetodin (Factory Method) avulla taas luodaan olion kutsuoperaatio, muttei päätetä mistä luokasta ilmentymä luodaan. Tällöin jonkin olioin luomiseksi rajapinta, joka toteutetaan toisaalla.

Torjuntahävittäjä (Interceptor) –arkkitehtuurimallin avulla voidaan kehykseen lisätä palveluja ja laukaista palvelu tapahtuman sattuessa tapahtumaohjatusti.

Operaatiorungon (Template Method) avulla voidaan luokan toiminnallisuudesta määritellä osa ja jättää loput aliluokan toteutettavaksi. Käyttökelpoinen silloin, kun halutaan, että kehys tarjoaa osan toiminnallisuudesta ja aliluokka täydentää ajonaikana toimintaa.

Perinnekomponentit (lecacy component) liitetään kehykseen Wrapper Facade-suunnittelumallin avulla. Tällöin rakennetaan komponentin rajapinnoille adaptereita. Fasadi-mallin avulla käyttäjälle tarjotaan yksi selkeä rajapinta, joka kätkee alla olevan monimutkaisuuden. Tarjotun rajapinnan avulla sovellukseen voidaan liittää ulkopuolisia elementtejä kuten komponentteja.

Lisäksi käytetään MVC-arkkitehtuurimallia sallimaan tiedon esitykseen monimuotoisuutta.

Komponentin konfiguraattori (Component configurator)-suunnittelumallin avulla hallitaan komponenttikehykseen kuuluvien komponenttien alustaminen, prosessointi ja toiminnan lopettaminen dynaamisesti. Muita kehyksiin liittyviä malleja ovat Rekursiokooste (Composite) ja Whole-Part – suunnittelumallit, joiden avulla voidaan koostaa olioita toisista olioista.

Laajentamistekniikan (extending technique) mukaan jaettuna kehysarkkitehtuurit ovat valkolaatikko, mustalaatikko ja harmaalaatikko. Valkolaatikkokehyksessä (white-box framework) olemassa olevaa toiminnallisuutta uudelleen käytetään perimällä kehyksen perusluokkia ja ylikirjoittamalla etukäteen määritellyt koukkumetodit (hook) malleja

käyttämällä. Mustalaatikkokehyksen (black-box framework) tapauksessa laajennettavuus saavutetaan siten, että määritellään rajapintoja komponenteille, jotka voidaan liittää kehykseen käyttäen olioiden koostamista. Komponentit siis joko noudattavat tiettyä rajapintaa tai komponentit integroidaan kehykseen mallien avulla. Harmaalaatikkokehys (gray-box framework) on sekoitus ylläolevia tapauksia. Harmaalaatikko sallii laajentamisen joko perinnän avulla (staattisesti), käyttäen dynaamista sidontaa (dynamic binding) ja määriteltyjen rajapintojen kautta. On arvioitu, että jopa yli puolet kehyksistä on harmaalaatikko-tapauksia [FaJ00 s. 616].

Komponenttikehys (component framework) kutsuu, alustaa ja poistaa komponentin ajonaikana. Komponenttikehyksen tarkoituksena on tarjota komponenteille allas, jossa ne voivat toimia ja tavan altaassa olevien komponenttien yhteistoiminnan toteuttamiseksi. Komponenttikehykset voivat toimia yhteistyössä halutun toiminnallisuuden aikaansaamiseksi [ScC98 s.280]. Komponenttikehys on myös kokoelma hyödyllisiä aputoimintoja. Kehyskomponentti on räätälöitävä komponentti, joka voidaan erikoistaa joko toteutus- tai erikoistamisrajapinnan kautta [Kos00 s. 272].

Komponenttikehykset tuotelinjassa (product line) voidaan luokitella tuotespesifisiin lisäysmalleihin (product-specific extension model), standardispesifisiin lisäysmalleihin (standard-specific extension model), hienojakoisiin lisäysmalleihin (fine-grained extension model) sekä generointipohjaisiin malleihin (generator-based model) [Bos00 s. 250] (kts. Liite 1).

Oliopohjainen sovelluskehys (application framework) on osittain valmis (semicomplete) sovellus, jolla on oma kontrollinohjaus (control loop). Tällaista kehystä voidaan laajentaa asiakassovelluksiksi käyttäen koukkumetodeja (hook method) ja kuumia pisteitä (hot spot) [FaJ00 s.615]. Sovelluskehys on kohdealuespesifinen (domain-specific) kuten esimerkiksi rajapinta, hajautettu systeemi tai sovelluksen toimintaympäristö [FaJ00 s.1]. Sovelluskehykset luokitellaan *systeemin infrastruktuurikehyksiin (system infrastructure frameworks), väliohjelmisto integraatiokehyksiin (middleware integration frameworks) ja liiketoimintasovellusten kehyksiin (enterprise application frameworks*) [FaJ00 s. 616] (kts. Liite 1).

3.5 Uudelleenkäytettävät ohjelmistot ja komponentit

Uudelleenkäytettävä komponentti (reusable component) on ohjelmistotuotantoprosessin tuote, jota voidaan käyttää ohjelmistotuotteen osana myös muissa tapauksissa kuin, mihin se on alun perin suunniteltu. Jotta komponentti olisi aidosti uudelleenkäytettävä, on se dokumentoitava asianmukaisesti. Dokumentoinnin tulisi tukea komponentin *luomista, välitystä ja käyttämistä (creation, brokering, consuming)* [For02 s. 20].

Komponenttien uudelleenkäytön tasot voidaan luokitella seuraavasti [Bos00 s.215]:

- S*ysteemin versioiden uudelleenkäyttö (system versions reuse)* tarkoittaa sitä, että komponentti mukautuu systeemin nykyisiin ja tuleviin vaatimuksiin. Tätä ei pidetä varsinaisesti uudelleenkäyttönä.
- *Ohjelmiston tuotelinja (software product line)* tarkoittaa tuotelinjan komponenttien uudelleenkäyttöä tuoteperheessä.
- *Kolmannen osapuolen komponentit (third-party components)* ovat sellaisia komponentteja, joita voidaan käyttää eri organisaatioissa. Puhutaan myös hyllytavarasta (COTS). Tällaiset komponentit ovat yleensä tiukasti kohdealuesidonnaisia tai muutoin rajoittuneita.

Merkittävimmät yleiset uudelleenkäytönkohteet ovat *käyttöjärjestelmät (operating systems), varusohjelmistot (utilility programs) ja pakettiohjelmistot (packet applications)*. Pakettiohjelmistoilla tarkoitetaan julkistettuja ohjelmia [Lin00]. Yrityksen yleisiä uudelleenkäytön muotoja ovat mm. arkkitehtuurit, koodi, data, suunnittelu, dokumentaatio, mallit ja arviot, käyttäjärajapinnat (human interfaces), suunnitelmat, vaatimukset ja testitapaukset [For02 s. 19]. Sunnittelutason uudelleenkäytön tehostamisella voidaan tehostaa projektin toteutusta. Suunnittelutason uudelleenkäytön muodot ovat Szyperskin mukaan [SzC97 s. 133]:

- jaettu yhdenmukaisuus: ohjelmointikielet (sharing consistency: programming and scripting languages),
- jaetut konkreettiset ratkaisutavat: kirjastot (sharing concrete solution fragments: libraries), jaetut sopimukset: rajapinnat (sharing contracts: interfaces),
- jaetut vuorovaikutusarkkitehtuuri: mallit (sharing interaction architectures: patterns),

- jaetut alisysteemiarkkitehtuurit: kehykset (sharing subsystems architectures: frameworks) ja
- jaettu yleinen rakenne: systeemiarkkitehtuuri (sharing overall structure: system architectures).

Kaikkien näiden tekijöiden hyödyntäminen parantaa ohjelmistoprojektin kustannustehokkuutta. Ongelmana edellä luetelluissa uudelleenkäytön muodoissa on se, että ne ovat yleensä käytettävissä vain organisaatiossa sisäisesti. Organisaatioiden ja yritysten välille tarvitaan parempia keinoja toteuttaa yhteistoiminnallisuutta. Markkinoilla, jossa tieto on kaupankäynnin väline, voidaan uudelleenkäyttö saavuttaa vain liiketoiminnan keinoin. Markkinoilla binäärеinä kaupattavat komponentit soveltuvat tarkoitukseen mainiosti. Toinen markkinoitava tuoteryhmä ovat palvelut. Palveluiden markkinat koostuvat suunnittelusta, koulutuksesta, systeemityöstä, asennuspalveluista, operoinnista ja ylläpidosta. Palveluiden avulla informaatiota siirretään uudelleen käyttäen sitä yritysten välillä.

Jacobsen toteaa, että ohjelmistoprojekteissa saavutetaan 50-90%:n uudelleenkäyttöasteita. Esimerkkeinä hän mainitsee mm. AT&T:n, Microsoftin, Ericssonin, IBM:n. Uudelleenkäytön ansiosta saadaan selviä hyötyjä. Tuotteen valmistusaika (time-to-market) puolittuu, virheiden määrä alenee 89-90% ja ylläpitokustannukset alenevat 80-90% [JGP97 s. 6].

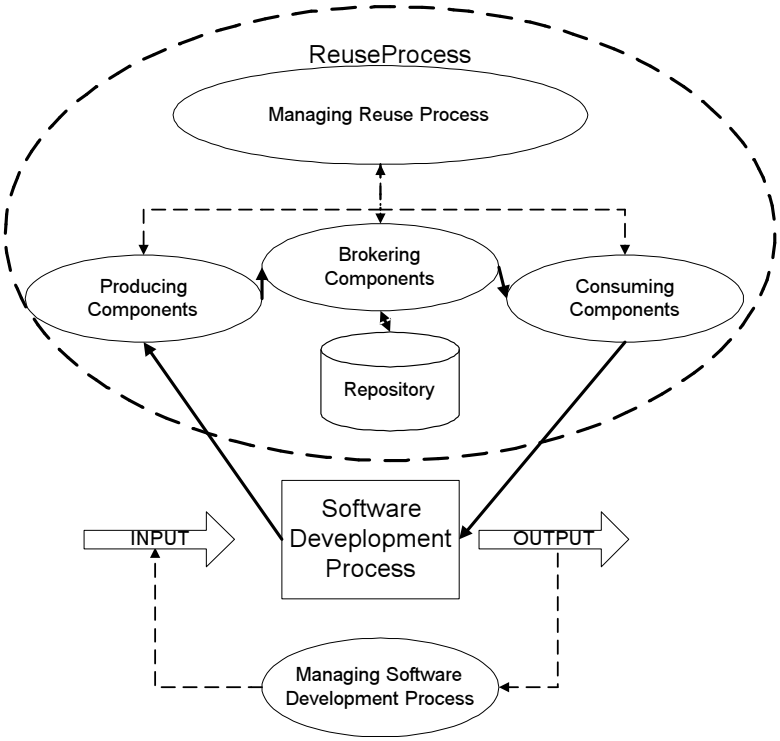
Uudelleenkäyttö vaatii muutoksia organisaation toiminnassa. Jotta uudelleenkäyttöä voitaisiin jatkuvasti lisätä, on organisaatio tutkittava toimintaansa jatkuvasti. Uudelleenkäytön sudenkuoppia voidaan välttää parantamalla ohjelmistosuunnittelumenetelmiä ja järjestämällä tuotanto vastaamaan uudelleenkäytön asettamia vaatimuksia.

Ohjelmistosuunnittelumenetelmät (software engineering methods) tarvitsevat keinoja, joilla tunnistetaan suunnittelun eri vaiheissa esiin tulevat elementit. Komponenttien uudelleenkäytön vähäisyys voi johtua siitä, että ei osata valita sopivia komponentteja, komponenttien dokumentaatio on puutteellista, kehitysympäristö ei tue komponenttikirjastojen luontia tai komponenttien ominaisuudet eivät tue uudelleenkäyttöä. Jos Komponentti on joustamaton, sitä on vaikea käyttää muihin kuin niihin tehtäviin, joihin se on ensimmäisellä kerralla suunniteltu. Sopivien työkalujen hankinta voi olla laiminlyöty. Komponenttikehitykseen ja uudelleenkäytön edistämiseen on kehitetty rajapintoja ja

sovelluskehittimiä, joiden avulla myös eri komponenttimalleilla (EJB, CORBA, .NET) tehtyjen komponenttien uudelleenkäyttö ja yhteistoiminnallisuus on helpompaa.

Jotta yritys saisi kaiken hyödyn komponenttiteknologioista on uudelleenkäyttö organisoitava. Projekteilla on oltava tieto uudelleenkäytettävistä komponenteista. Uudelleenkäytön tulee olla harkittua. Tuotteella on pitkä elinkaaari, joten valittaessa niitä komponentteja, joita käytetään, on huomioitava se, onko komponentille saatavissa tukea esim. 5-10 vuoden kuluttua.

Uudelleenkäyttösuuntautunut ohjelmistokehitys (reuse-oriented software development) - prosessimallissa mallinnetaan ohjelmistosuunnitteluprosessia, jossa on huomioitu uudelleenkäytettävyys (Kuva 21).



Kuva 21. Komponentin uudelleenkäyttö ja ohjelmistoprosessi [For02 s. 21]

Tuotannon järjestäminen (managing reuse process) tarkoittaa sitä, että arvioidaan uudelleenkäyttöä: arvioidaan, mikä on hyvää ja mitä pitäisi kehittää, sekä pyritään komponenttien hallintaan, mitataan käytettävyys ja kustannus/hyöty-suhde. Lopuksi vielä arvioidaan ihmisten johtaminen, joka tarkoittaa tekniikoiden hallintaa, koulutusta ja luovuuden hyväksikäyttöä. *Komponenttien tuotannossa (producing components)* tehdään aluksi *kohdealueen analyysi (domain analysis)* [For02], [PSM94], jonka avulla löyde-

tään uudelleenkäytettävät abstraktiot, jotka sen jälkeen tuotetaan komponenttienohjelmistoprojektissa. Tuotanto voi suuntautua ylläpitoon tai olla uustuotantoa. Prosessi voi olla korjaava, mukauttava tai täydentävä [For02], [LST78]. Komponentit on välitettävä eteenpäin. *Komponenttien välitys (brokering components)* lisää eri projektien välillä uudelleenkäyttöä, luo luottamusta olemassa oleviin komponentteihin [For02], [Tra95]. Komponenttivaraston ylläpito, arviointi ja validointi on järjestettävä. Yrityksen on ylläpidettävä tietovarastoa komponenteistaan. Tietovarastossa on dokumentoituna ne komponentit, joita yritys käyttää projekteissaan. Komponentit voivat olla omia tai ostettuja. *Komponenttivaraston (repository*) ylläpitoon liittyy viisi erillistä tehtävää [For02], [Lim98] eli arviointi, hankkiminen, sertifiointi, lisääminen ja poistaminen. Komponenttien käyttö (consuming components) tarkoittaa sitä, että komponentti löydetään, se ymmärretään, sitä muokataan, se otetaan käyttöön ja lopuksi se integroidaan tuotteeseen. Komponenttien hallinta on tärkeä osa ohjelmistotuotantoprosessia ja se liittyy läheisesti arkkitehtuurisuunnitteluun.

3.6 Komponentit ja integraatio

Merkittävä pullonkaula on eri aikakaudelta olevien järjestelmien ja erilaisten prosessien integroiminen toisiinsa. Herzum esittää järjestelmäintegraation hierarkisesti tasoina. Kehitettäessä järjestelmää tulisi kaikkien tasojen vaatimukset huomioida. Tasot ovat ylhäältä alaspäin korkeimmalta abstraktiotasolta lähtien seuraavat [Myk00 s.93], [HeS99]:

- Lähtökohtana on systeemin kehityksen elinkaariajattelu (development lifecykle viewpoint). Ylimpänä ovat kehitysprosessin liittymät (development lifecycle interfaces) eli määritellään, mitä liittymiä mahdollisesti tarvitaan kehittäessä järjestelmää.
- Toiminnallisen arkkitehtuurin näkökulmasta (functional archicture viewpoint) voidaan määritellä toiminnallinen viitemalli (functional reference model), joka lähtee siitä, että ainakin rajapintojen viitemallin tulisi olla sama. Semantiikka (semantics) kertoo tiedon tarkoituksen ja toiminnalliset liittymät (functional interfaces) tarkoittavat operaatioiden määrittelyä eli millainen liittymä on.
- Tarkasteltaessa sovellusarkkitehtuurin näkökulmasta (application architecture viewpoint) järjestelmää, käsite sovellusinfrastruktuuri (application infrastructure) tarkoittaa, mitä arkkitehtuuria ja millaisia liittymäkäytäntöjä ja suunnittelumalleja järjestelmässä käytetään.

- Tekninen arkkitehtuuri näkökulma (technical architecture viewpoint) lähtokohtana voidaan määritellä tekninen infrastruktuuri (technical infrastructure) –taso, joka lähtee siitä, kuinka teknisesti tuetaan kommunikaatiota ja tekniset liittymät (technical interfaces**)** -taso, joka kuvaa käytettävän tekniikan.

Analysointi- ja suunnitteluvaiheessa komponenttien integrointi voidaan huomioida jo alusta pitäen kehitysprosessin liittymin ja määritellä suoraan komponenttien tarjoamat ja tarvitsemat rajapinnat. Komponentin elinkaaren aikana integrointi toteutetaan myös teknisesti. Tällöin toteutetaan suunnittelu aikana määritellyt liittymät ja rakennetaan tarvittaessa adaptereita yhteensopivuuden mahdollistamiseksi. Jotta komponentti voisi toimia toisen komponentin kanssa, komponentteja on kyettävä ajamaan niin, että ne voivat lähettää toisilleen viestejä ja vastaanottaa viestejä toisilta komponenteilta. Lisäksi onnistunut integraatio edellyttää, että on olemassa sopimus rajapinnoista ja rajapintojen hyödyntämistä varten tarvittavista esi- ja jälkiehdoista. Liittymän semantiikka eli merkitys on oltava osapuolten tiedossa. Toiminnallisen viitemallin avulla voidaan yhtenäistää käytettävät rajapinnat ja rajoittaa tarvetta rakentaa adaptereita tai siltoja järjestelmien välille.

Järjestelmäintegroinnissa tulisi pyrkiä siihen, että integrointi voi tapahtua mahdollisimman korkealla abstraktiotasolla. Tällöin järjestelmät voivat suoraan käyttää toistensa palveluja. Sanotaan, että järjestelmät ovat integroidut (integrated) toisiinsa. Aina tämä ei onnistu. Usein järjestelmiä on kehitetty irrallaan toisistaan tai uusia järjestelmiä halutaan liittää olemassa olevaan järjestelmään. Tällöin integrointi tapahtuu joko *väyläpohjaisesti (bus-based)*, *sillan (brigde)* avulla tai *koordinaatiopohjaisesti (coordinated)* [HeS99 s. 211].

Silta-ratkaisu soveltuu point-to-point –tyyppisiin integrointeihin, joissa yksittäisiä järjestelmiä integroidaan toisiinsa. Silta ratkaisun etuna on sen toteutuksen helppous. Ensiksi sovitaan sanoman sisältö ja muoto (protokolla). Sen jälkeen on osapuolten vastuulla muuttaa sanomat sovitun protokollan mukaiseksi. Muunnos tapahtuu adapterien avulla. Muunnoksen jälkeen sanoma annetaan silta-elementille, joka toimittaa sen vastaanottajalle. Vastaanottaja muuttaa sanoman adapterinsa avulla itselleen sopivaksi. Haittana silta-ratkaisussa on se, että se on ikään kuin "kovakoodattu", mikä tarkoittaa sitä, että se soveltuu sellaisenaan vain kyseisiin käyttökohteeseen eikä ole yleensä yleistettävissä.

Väylän ominaisuutena on kyky integroida useita järjestelmiä toisiinsa (many-to-many). Väylä tarjoaa monimuotoisuutta siten, että kyselyn suorittaja voi väylän avulla suorittaa kyselyjä useilta palvelun tarjoajalta. Väylä ratkaisussa kyselyn lähettävä systeemi muuttaa adapterin avulla viestin väylän protokollan mukaisesti ja lähettää sen väylään halutuille vastaanottajille. Vastaanottajat muuttavat viestin itselleen sopivaksi ja vastaavat viestiin. Joustavuuden hintana ovat lisääntyvä monimutkaisuus ja silta-ratkaisua kalliimmat rakennuskustannukset. Väylä-ratkaisua käytettäessä joudutaan väylän käyttö huomioimaan siihen osallistuvissa järjestelmissä jo rakennusaikana, koska järjestelmien on oltava jollain tapaa kykeneviä yhteistoiminnallisuuteen.

Koordinaattoria käytettäessä järjestelmien välillä on koordinaattori-komponentti, jonka kautta ja avulla järjestelmät integroidaan toisiinsa. Kuten muissakin tapauksissa tarvitaan järjestelmien välille adaptereita yhteisen protokollan luomiseksi. Koordinaattorin etuna on se, että se mahdollistaa kyselyn suorittamisen usealta palveluntarjoajalta yhdellä kyselyllä, jos koordinaattori tukee tätä. Edelleen koordinaattori sallii sen, että järjestelmät voivat olla toisistaan riippumattomia ja eivätkä tiedä mitään toisistaan. Koordinaattori-ratkaisu on monimutkaisin, koska koordinaattorissa on oltava runsaasti toiminnallisuutta. Koordinaattorina voi olla erillinen sovellus, joka voi olla hankittu valmiina. Koordinaattorin käyttö on perusteltua silloin, jos on tarvetta luoda järjestelmien yhteistoimintaa varten sääntöjä eli jonkinlainen toimintaprotokolla, jonka avulla voidaan toimintaa hallita. Koordinaattorin avulla voidaan järjestelmät erottaa toisistaan. Koordinaattorin etuna on se, että siihen on väylään verrattuna helpompaa liittää jälkikäteen erilaisia järjestelmiä.

Parhaimmalta integrointikeinolta tuntuisi väylä-ratkaisu, koska se on joustava ja siihen on helpointa liittää uusia järjestelmiä. Järjestelmien yhteistoiminnan kehittäminen on mahdollista myös jälkikäteen silloin, kun järjestelmissä on löydettävissä riittävästi yhteistoiminnallisuutta.

3.7 Yhteenveto komponenteista

Komponenttien avulla on viimeisen kymmenen vuoden aikana luotu lukemattomia sovelluksia. Komponenttien avulla on lisätty uudelleenkäyttöä. Komponenttien avulla on rakennettu hajautettuja järjestelmiä. Komponenttien kuten hajautuksenkin varjopuolina on lisääntyvä monimutkaisuus. Tämän monimutkaisuuden hallintaan on kehitetty arkkitehtonisia tarkastelutapoja. Komponenttien luomiseksi on luotu sovelluskehittimiä, joi-

den tarkoituksena on eristää komponentti-spesifiset tekijät liiketoiminnan suunnittelusta. Uskotaan, että tekninen osuus voidaan jättää tekniikan hoidettavaksi. Ehkäpä tämä onnistuu, jää nähtäväksi.

Komponenttien valmistus vaatii kuitenkin organisatorisia muutoksia. On syntynyt uusia tehtäviä. Tarvitaan *komponenttisysteemiarkkitehtia*, *komponenttikehysarkkitehtia, komponenttiohjelmoijaa* sekä *komponentin kokoajaa* [SzC97 s. 337]. Samoin organisaation on muutettava toimintaansa niin, että syntyy komponentin käyttäjiä ja komponentin kehittäjiä. Tämä on tarpeen, jotta kyettäisiin allokoimaan resursseja pitkäjänteisesti uudelleenkäytettävien komponenttien kehittämiseen.

Komponenttiajattelu koostamisen välineenä säilynee ja kehittynee. Komponenttien edut ovat kiistattomat. Yritysten liiketoimintaprosessit ovat mallinnettavissa ja ne voidaan toteuttaa uudelleenkäytettävien komponenttien avulla. Komponenttiteknologian ongelmina on tähän asti ollut eri teknologioilla tehtyjen ohjelmiston osien huono laatu tai kallis yhteensovittaminen. On viitteitä siitä, että tämä muuttuu. Tulevaisuudessa erilaiset komponentit pystyvät keskustelemaan keskenään. Tämä ominaisuus on joko rakennettu niihin valmiina tai komponentilla on universaali rajapinta, jonka avulla sen ominaisuuksia voidaan käyttää. Kaikilla komponenteilla on lisäksi oltava yhteinen universaali liitoskohta, jonka avulla se voidaan liittää ajoympäristöön. Tällöin komponentille saadaan mahdollisimman laaja asiakaskunta ja päästään synnyttämään laajat komponenttimarkkinat.

4 Arkkitehtuurit

4.1 Ohjelmistoararkkitehtuurit – arkkitehtuurin näkökulmat

4.1.1 Yleistä

Ohjelmistoarkkitehtuuri kuvaa systeemin rakennetta, joka koostuu ohjelmistokomponenteista, niiden ulkoapäin näkyvistä ominaisuuksista ja niiden välisistä suhteista [Bos00 s. 11].

Arkkitehtuurin tulee perustua tehtyyn kohdealueen analyysin, jonka pohjalta valitaan paras arkkitehtuurivaihtoehto. Valintaa hankaloittavat valitettavasti useat rajoitteet: organisaation vaatimukset, kehitystiimin ammattitaito ja mieltymykset, vaihtoehtoiset teknologiat, käytettävissä olevat resurssit, aikataulu, systeemin asettamat vaatimukset sekä muut sidosryhmät. Kun alustava arkkitehtuuri on valittu, on sen sopivuus arvioitava, laatutekijät (laatuattribuutit) on asetettava kohdalleen ja parannettava arkkitehtuuria tarvittaessa. Arkkitehtuuria muokataan kunnes se tyydyttää asianosaisia.

Ohjelmistoarkkitehtuurin avulla on mahdollista arvioida rakennettavan järjestelmän ominaisuuksia. *Laatutekijöiden eli laatuattribuuttien (quality attributes)* arviointi koetaan tärkeiksi. Tarkastelemalla arkkitehtuuria voidaan arvioida esimerkiksi systeemin suorituskykyä, joustavuutta, virheensietokykyä ja sitkeyttä.

Arkkitehtuurin avulla voidaan ohjelmistosta tunnistaa erilaisia *arkkitehtuurityylejä (architectural style)*. Voidaan havaita, että ohjelmistolla on yleistyyli ja voidaan sanoa, että ohjelmisto toteuttaa tiettyä *arkkitehtuurimallia (architectural pattern)* tai siihen sisältyy useita arkkitehtuurimalleja. Näiden mallien rakentamisessa ja koostamisessa käytetään *suunnittelumalleja (design patterns)*. Arkkitehtuurien kuvaamiseksi on laadittu *mallisysteemejä (pattern systems),* joissa kuvataan ja luokitellaan mallien ominaisuuksia ja niiden välisiä riippuvuuksia. Arkkitehtuurilla on monta tehtävää: markkinointitehtävä, kehitysaikainen tehtävä, kommunikaatiotehtävä, dokumentaatiotehtävä [Smo03]. Kunkin tehtävän toteuttaminen vaatii toiminnan organisointia niin, että arkkitehtuuri on osa tuotantoprosessia ja arkkitehtuurisuunnittelun tuotokset otetaan käyttöön prosessissa [Bos00].

Arkkitehtuurin kuvauksessa *puoliformaalit menetelmät* kuten esimerkiksi *Unified Modeling Language UML*, ovat suosittuja, koska niiden avulla voidaan kuvata joustavasti ja luovasti liiketoimintaprosesseja. Tosin UML on puutteellinen arkkitehtuurin kuvauksen kannalta. Arkkitehtuurin kuvauksessa käytetään myös *arkkitehtuurin kuvauskieliä (achitecture description language ADL)*, joiden avulla voidaan formaalisti kuvata arkkitehtuuria, sen osia, osien välisiä suhteita ja osien välillä olevia esi- ja jälkiehtoja [CleF02], [Bos00 s. 12]. Tuotantoprosessissa *arkkitehtuurin arviointia (architectural assessment)* ja *arkkitehtuurin parantamismenetelmiä (architecture improvement)* on kehitettävä. Arkkitehtuurin dokumentointitehtävänä on kerätä, luetteloida ja esitellä ja julkistaa toteutuneita onnistuneita arkkitehtuuriratkaisuja. Dokumentoinnin kohteita ovat esimerkiksi kokonaiset systeemit, mallit, kehykset, komponentit ja muut mekanismit.

Komponenttiajattelun paradigma on se, että uudelleenkäytön avulla parannetaan ohjelmiston laatua ja tuotantoprosessin tehokkuutta. Tuotelinja-arkkitehtuurin avulla pyritään löytämään komponentteja, joita voitaisiin käyttää useissa samankaltaisissa tuotteissa tai komponentteja, jotka voivat tehdä samoja tehtäviä yrityksen eri systeemeissä.

4.1.2 Näkökulmat

Tietojärjestelmällä on monta sidosryhmää. Kukin tarkastelee järjestelmää omalta kannaltaan. Jokaiselle tarvitaan omiin tarpeisiin sopiva kuvaustapa. Näiden tarpeiden täyttämiseksi on kehitetty erilaisia tarkastelutapoja.

Kruchten painottaa *4+1 –näkymässä* eri lähestymistapojen merkitystä. *Käyttötapausnäkymässä (use case view)* keskitytään eri käyttötapauksiin ja kuvataan ne. *Loogisessa näkymässä (logical view)* kuvataan luokkia, rajapintoja ja toimintoja. *Toteutusnäkymässä (implementation view)* kuvataan tiedostoja, komponentteja sekä tuotteen konfiguraatiota. *Prosessinäkymä (process view)* puolestaan keskittyy prosesseihin tavoitteenaan *tehokkuus (performance)* ja *skaalautuvuus (scalability)*. *Jakelunäkymä (deployment view)* painottaa laitteistotopologiaa, jakelua ja asennustekijöitä [Kru95]. Kruchtenin näkökulmissa hyvää on niiden kattavuus. Dokumentaatiossa voidaan näiden näkökulmien avulla tuoda esiin kutakin sidosryhmää kiinnostavia kohtia ohjelmistosta. Lisäksi nämä näkymät voidaan visualisoida havainnollisesti ja tarkasti. Visualisointiin voidaan käyttää esimerkiksi UML:n notaatiota. Kruchtenin näkökulmat ovat laajasti käytössä arkkitehtuurien dokumentoinnissa.

Mary Shaw:lla on kolme näkökohtaa arkkitehtuurin rakenteeseen. *Kehysmallien (framework models)* avulla pyritään koko systeemille saamaan yksi ja sama arkkitehtuuri. Kehysnäkökohta keskittyy kohdealue-spesifiseen ohjelmistoarkkitehtuuriin tai tietovarastoon. Tällöin middleware tai kommunikaatioinfrastruktuuri on tärkeää. *Dynaamiset mallit* korostavat systeemin käyttäytymistä, mikä tarkoittaa sitä, että nämä mallit kuvaavata mm. muutoksia systeemin konfiguraatiossa, kommunikaatiossa tai muutoksia tietosisällössä. *Prosessointimallit* keskittyvät arkkitehtuurin rakenteeseen ja prosessiin tai askeleisiin, jotka ovat mukana tässä rakenteessa. Tältä kannalta arkkitehtuuri on seurausta prosessiskriptin seuraamisesta. Shaw:n näkökulmat soveltuvat komponenttien välisen vuorovaikutuksen tutkimiseen [CleF02].

Soni, Nord ja Hofmeister esittelivät (1995) arkkitehtuurinäkymiä teollisessa ympäristössä ”The Siemens Four View model for architechture”. He jakavat näkökulmat seuraavasti. *Konseptuaalinen arkkitehtuuri (conseptual architecture*) kuvaa systeemiä tärkeimpien suunnitteluelementtien pohjalta ja keskittyy niiden välisiin suhteisiin esimerkiksi tutkien komponenttien ja konnektorien välisiä suhteita. *Moduliarkkitehtuuri (module architechture)* tutkii kerrostuvia rakenteita ja niiden jakautumista toiminnallisiin osiin ja tasoihin (functional decomposition and layers). Esimerkkeinä tällaisista ovat alisysteemit ja modulit. *Toteutusarkkitehtuuri (execution architecture)* kuvaa järjestelmän dynaamista rakennetta kuten tehtäviä, säikeitä ja prosesseja. Viimeisenä näkökulmana on *koodiarkkitehtuuri (code architecture),* joka kuvaa kuinka esimerkiksi tiedostot, hakemistot, koodi, binäärit ja kirjastot on organisoitu kehitysympäristössä. Nämä näkökulmat ovat vahvasti suunnittelukeskeisiä ja soveltuvat projektia toteuttavien suunnittelijoiden tekemään ohjelmistoprojektin tarkasteluun [CleF02].

Herzum ja Simms kuvaavat neljä näkymää seuraavasti. Ensiksi *tekninen arkkitehtuuri (technical architecture)* keskittyy komponenttien ajoympäristöön, joukkoon työkaluja, käyttäjän rajapinnan kehykseen ja muihin teknisiin palveluihin (services/facilities) ja toimintoihin, joita tarvitaan kehitettäessä ja ajettaessa komponenttipohjaista systeemiä. S*ovellusarkkitehtuuri (application architecture)* on kiinnostunut joukosta arkkitehtonisia päätöksiä, malleista, suuntaviivoista ja vaadituista standardeista, joita tarvitaan rakennettaessa komponenttipohjaista systeemiä. *Projektiarkkitehtuuri (project management architecture)* koostuu niistä elementeistä mm. konsepteista, suuntaviivoista, periaatteista ja johtamisen ja kommunikoinnin työkaluista, joita tarvitaan rakennettaessa

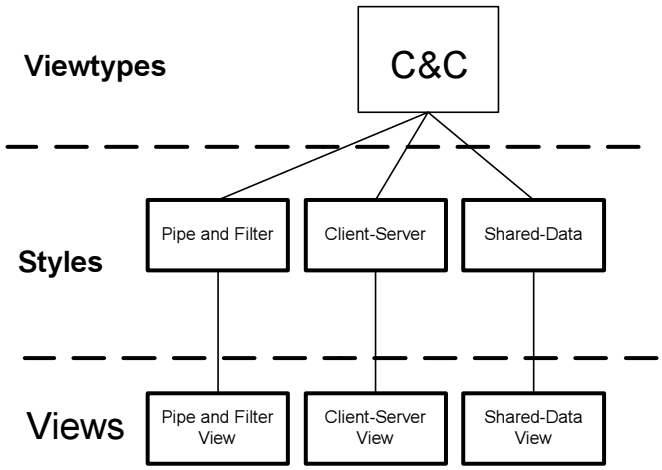
skaalautuva, suuri järjestelmä yhdessä suuren projektiryhmän kanssa. *Toiminnallinen arkkitehtuuri (functional architecture)* määrittelee ja kuvaa rakennettavan arkkitehtuurin. Herzumin näkökulmat pohjaavat vahvasti hänen integraatiomalliinsa, jossa tutkitaan eri abstraktiotasoille hajautettua integraatioprosessia [CleF02], [HeS99 s. 73].

Näkymä (view) on joukko systeemin elementtien esityksiä ja niiden välisiä suhteita [CleF02 s.13]. Paul Clements luokittelee näkymät jakaen ne hierarkisesti *näkymätyyppeihin (viewtypes), tyyleihin (style)* ja *näkymiin (views)* [CleF02 s. 18]. Näkymätyyppejä ovat moduulinäkymätyyppi (module viewtype), komponentti ja yhdistin –näkymätyyppi (component and connector viewtype) sekä allokointinäkymätyyppi (allocation viewtype).

Tässä tarkastelutavassa moduulinäkymätyyppi on joukko luokkia tai mikä tahansa koosteen koodissa. Moduuleilla on suhteita toisiinsa. Tähän liittyvät tyylit (styles) seuraavasti [CleF02 s. 35]:

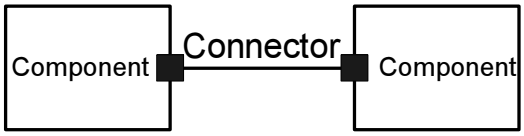
- Osiinjako-tyyli (decomposition style) jakaa tehtävät systeemeihin ja alisysteemeihin.
- Käyttää-tyyli (uses style) kertoo moduulien välisistä riippuvuuksista.
- Yleistys-tyyli (generalization style) kertoo koodiyksiköiden hierarkisuudesta ja kerrostyyli (layered-style) organisoi koodin erillisiin kerroksiin. Tämä tyyli tukee siirrettävyyttä, muunneltavuutta.

Kuvassa 22 on esitetty esimerkkinä komponentti ja yhdistin -näkymien hierarkia.



Kuva 22. Näkymien hierarkia esimerkkinä komponentti ja yhdistin (C&C) -näkymätyyppi [CleF02 s.13]

Komponentti ja yhdistin (C&C) –näkymätyyppi kuvaa ajonaikaista käyttäytymistä. Komponentteja voivat olla oliot, prosessit, entiteetit tai oliokokoelmat. Komponentteja liittävät toisiinsa yhdistimet. Yhdistimiä voivat olla putket (pipes), varastot (repositories), liitoskohdat (sockets) sekä väliohjelmistot (middleware). Kuvassa 23 on kuvattu periaatepiirroksen avulla kahden komponentin liittyminen toisiinsa [CleF02]. Liittäminen voidaan toteuttaa esimerkiksi käyttäen putki-suodatin (pipes and filters), jaettu tietovarosto (shared-data), tilaaja-tuottaja (publish-subscribe), asiakas-palvelin (client-server), vertaisverkko (peer to peer) tai kommunikoivat prosessit (communicating-processes) –tyylejä (kts. Liite 7).



Kuva 23. Komponentti ja yhdistin

Allokointinäkymä-tyypissä taas keskitytään tarkastelemaan ohjelmistoyksiköiden kiinnittymistä ympäristön elementteihin. Esimerkkinä laitteisto (hardware), tiedostojärjestelmät (file systems), kommunikointi tai kehystiimit. Tyylejä on kolmenlaisia. *Sijoittelutyyli (deployment style)* sijoittelee prosessit laitteistoelementteihin esimerkiksi prosessoiviksi solmuiksi, kommunikointikanaviksi, muistivarastoiksi tai tietovarastoiksi. *Toteutustyyli (implementation style)* kohdentaa moduulit kehitysympäristöön. Ja lopuksi *työtehtävä-tyyli (work assigment style)* on apuna, kun moduuleita kohdennetaan kehitystiimeihin. Tällä tyylillä kerrotaan, kuka on vastuussa mistäkin [CleF02].

4.1.3 Yhteenveto

Valittava arkkitehtuurinäkökulma riippuu tarpeesta. Tarkastelu tehdään sidosryhmän tarpeiden pohjalta. Asiakasta kiinnostavat eri asiat kuin ohjelmoijaa. Näkökulma yhdistelemällä systeemistä saadaan parempi ja tarkempi kuva. Ajatellaan esimerkiksi, että loogisessa näkymässä kuvataan järjestelmää komponenttikaavion ja luokkakaavioiden avulla. Tällöin saadaan staattinen kuvaus käännösaikaisesta tilanteesta. Näkymä kertoo, mistä komponenteista, moduuleista ja luokista eli elementeistä järjestelmä koostuu ja riippuvuuksista elementtien välillä. Mutta se ei kerro riittävästi ajonaikaisesta käyttäytymisestä. Dynaaminen käyttäytyminen on esitettävä erikseen. Kuvataan komponenttien ja olioiden välistä vuorovaikutusta sekä komponentin tai olion tilaa prosessin eri vai-

heissa. Tämän lisäksi tarvitaan vielä näkökulma prosesseihin. Prosessit koostuvat järjestelmän tehtävästä ja käyttäjän ja järjestelmän vuorovaikutussuhteesta. Näitä kuvataan erilaisten prosessikaavioiden avulla.

4.2 Arkkitehtuurityylit,-mallit ja suunnittelumallit suunnittelun välineenä

4.2.1 Yleistä

Rakennusarkkitehtuureissa on löydettävissä erilaisia tyylejä, yksityiskohtia ja säännönmukaisuuksia, jotka ovat ominaisia kullekin aikakaudelle, rakennuksen käyttötarkoitukselle tai arkkitehdille. Ohjelmistoissa on löydetty vastaavanlaisia tyylejä.

Erilaisia malleja sovelletaan yleisesti esimerkiksi kehitysprosesseissa, organisaatioiden kehittämisessä, järjestelmäarkkitehtuureissa, analysointimalleissa, olevien kehysten käytössä ja uusien kehysten kehittämisessä [www011].

Ohjelmistoarkkitehtuurikuvauksissa on luotu malliratkaisut tavanomaisimpien ongelmien ratkaisemiseksi. Mallikirjastojen tarkoituksena on toimia käsikirjana parasta tai riittävän hyvää mallia etsittäessä ja sovellettaessa. Ajatuksena on, että alan paras osaaminen olisi mallien avulla myös kokemattomien ohjelmistosuunnittelijoiden käytössä. Jokainen malli tarvitsee toimiakseen selkeät käytännön soveltamisohjeet [Bos00]. Mallien välillä on riippuvuussuhteita. Yhden mallin käyttö johtaa toiseen malliin. Mallit muodostavat *mallijärjestelmiä (pattern language)* [Kos00 s. 246].

Luokitellut mallit esitetään *mallin (pattern)* avulla. Mallissa esitettävä asiayhteys eli *konteksti (context)* on tilanne, jossa ongelma esiintyy. Asiayhteys voi olla hyvinkin yleinen, toisaalta asiaa voi lähestyä listaamalla tilanteet, joissa ongelmat esiintyvät. *Ongelma (problem)* kuvaa kontekstissa ilmenevän ongelman. Käsitellään ongelmaa, joka tässä kontekstissa esiintyy toistuvasti ja on tärkeää. Aloitetaan yleisellä ongelmanmäärittelyllä, määrittelemällä hyvin yleisellä tasolla, mitä pitäisi ratkaista. Tätä yleistä määrittelyä täydennetään joukolla *voimia (forces)*. Voimilla tarkoitetaan kaikkia niitä ongelman piirteitä, jotka vaikuttavat ongelmaan ja jotka pitäisi huomioida, kun ongelma ratkaistaan kuten esim. toteutettavat vaatimukset, toteutettavat rajoitteet, toivottavat ominaisuudet. Voimat ovat avain ongelman ratkaisuun. Jos esimerkiksi tietoturvan asettamia rajoitteita ei huomioida, voi ohjelmiston käyttöarvo olla nolla, vaikka se täyttäisi-

kin muut vaatimukset. *Ratkaisu (solution)* on käytännössä kokeiltu ja testattu ratkaisu ongelmaan. Esitetään ratkaisu ongelmaan tai paremminkin ratkaisu, joka tasapainottaa voimat [GaH95 s.6].

4.2.2 Arkkitehtuurityylit ja -mallit

Mallien luokittelun käsitteissä on horjuvuutta: toisaalla [Bos00] puhutaan arkkitehtityyleistä ja toisaalla [BuR96], [ScR01] puhutaan arkkitehtuurimalleista. Kuitenkin tarkoitetaan jotakuinkin samaa. Noudatetaan Boschin esittämää, joten arkkitehtuurityyli määrittelee koko systeemin viitearkkitehtuurin.

Arkkitehtuurityyli (architectural style) kuvaa järjestelmän tai sen osan perusrakenteen. Sanomme, että ohjelmisto noudattaa esim. kerrostyyliä. *Arkkitehtuurimalli (architectural patterns)* määritellään niin, että se ei ole määräävä, vaan voi sulautua arkkitehtuurityyliin ja täydentää sitä. Arkkitehtuurimalli tavallisesti määrää arkkitehtuurin säännöt, jotka määrittelevät kuinka systeemi käsittelee jotain toiminnallista piirrettä. *Suunnittelumallin (design patterns)* [GaH95] vaikutus kohdistuu rajoitettuun joukkoon luokkia ohjelmistoarkkitehtuurissa. Lisäksi yksi luokka voi olla mukana useassa suunnittelumallissa ilman, että se aiheuttaisi epäjohdonmukaisuutta. Ohjelmointikielen erikoispiirteitä voidaan hallita *idioomien (idioms)* avulla (kts. liite 1).

Mary Shaw määrittelee perusarkkitehtuurityylejä järjestelmän korkean tason abstraktioiden luomiseksi (high-level system design) ja ohjeet arkkitehtuurin dokumentoinniksi. Mallin dokumentointi sisältää ongelman ja kontekstin määrittelyn, ratkaisun, tarvittavat kaaviot tyypillisestä mallista, merkittävät variantit sekä joukon esimerkkejä, joita on sovellettu malliin (Liite 7). Buschmann on laatinut oman mallisysteeminsä, johon sisältyy malleja korkean tason (high-level) arkkitehtuurimalleista pienempää rakeisuutta kuvaaviin suunnittelumalleihin ja idioomeihin [BuR96]. Buschmann luokittelee [BuR96] arkkitehtuurit ja suunnittelumallit niiden toiminnallisuuden mukaisesti siten, että luokittelu kertoo mallin toiminnallisen tarkoituksen (Liite 7). Esimerkiksi hajautusarkkitehtuurit-luokittelu kuvaa arkkitentuurimalleja, joiden avulla toteutetaan toimintojen hajautus verkossa. Merkille pantavaa on, että yhden mallin avulla ratkaistaan vain yksi ongelma. Esimerkiksi Broker-mallin avulla komponentteja voidaan hajauttaa eri osoiteavaruuksiin ja mahdollistetaan ne keinot, joiden avulla komponentit löytävät toisensa. Sen jälkeen on kuitenkin ratkaistava tarvittava yhteistoiminnallisuus eli esimer-

kiksi se, kuinka komponentit kommunikoivat ja minkä muotoista tietoa ne vaihtavat keskenään.

Samanaikaisuuden ja verkotettujen olioiden hallintaan koottu mallijärjestelmä keskittyy kommunikaation ylläpitoon sellaisissa järjestelmissä, joissa vaaditaan skaalautuvuutta ja kuormansietoa ja samanaikaisuuden hallintaa. Tämä mallijärjestelmä täydentää aiempaan Buschmannin laatimaa järjestelmää (Liite 7 Taulukko 2). Samanaikaisuus ja eri osoiteavaruuksiin verkottaminen antavat suunnittelijalle runsaasti haasteita. Palvelu on kyettävä löytämään, palvelupyyntö on kyettävä käsittelemään tehokkaasti, turvallisesti ja luotettavasti ja vastaus on kyettävä toimittamaan asiakkaalle. Mallien avulla pyritään auttamaan suunnittelijaa palvelun tavoittamisessa ja konfiguroinnissa (service access and configuration), tapahtumankäsittelyssä (event handling), synkronoinnissa (synchronization) sekä samanaikaisuuden hallinnassa (concurrency) [ScR01 s. 1].

Martin Fowler on luokitellut kirjassaan liiketoimintasovellusten arkkitehtuurimalleja. Fowler keskittyy liiketoimintasovellusten hallintaan kerrosarkkitehtuurien avulla sekä siihen, kuinka järjestää kohdealueen toimintalogiikka ja kuinka sitoa se tietokannanhallintajärjestelmiin. Lisäksi Fowler esittää malleja web-sovellusten hallintaan ja hajautettujen järjestelmien suunnitteluperiaatteita [www012].

Suunnittelumallien perustapaukset on koottu Design Patterns - Elements of Reusable Object Oriented Software -teokseen [GaH95]. Javan Enterprise-liiketoimintakomponentteihin erikoistuen on koottu suunnittelumalleja, jotka tarjoavat malleja transaktioiden, hajautuksen, samanaikaisuuden ja tietokannan hallintaan (Liite 7 Taulukko 3).

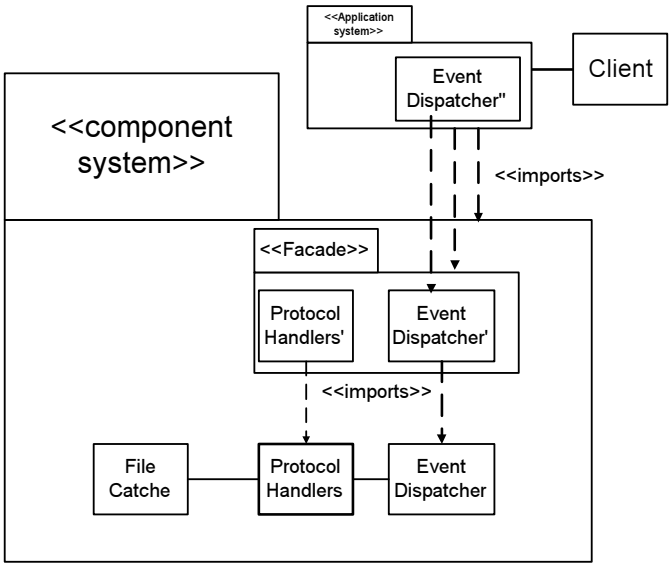
Liiketoimintakomponenttien suunnittelussa ei siis tarvitse normaalitapauksessa kiinnittää huomiota komponenttien väliseen arkkitehtuuriin eikä alustaan. Riittää, kun valitsee haluamansa kommunikointimuodon. Implementointivaiheessa valitaan sitten esimerkiksi tarkoitukseen sopiva EJB-komponentti.

4.2.3 Esimerkki arkkitehtuuri- ja suunnittelumallien käytöstä

Esimerkkinä eri arkkitehtuuri- ja suunnittelumallien käytöstä esitellään JAWS web-palvelinprojekti, jossa arkkitehtuurisuunnittelussa on käytetty suunnittelumalleja. JAWS on sekä WEB-palvelin että sovelluskehys. Kuvassa 24 on esitetty tyypillinen ympäristö WEB-palvelun järjestämiseksi. Mallissa jakelukelpoista sovellusta kuvaa Application

system –paketti, joka on valmis, konfiguroitu jakelupaketti. Komponenttisysteemi (component system) –paketti kuvaa uudelleenkäytettävää komponenttia, joka erikoistetaan valmistuksen aikana. Komponenttisysteemi tarjoaa palvelurajapinnan Facade-komponentin kautta.

WEB-palvelimen toiminta tapahtuu niin, että asiakas esittää pyynnön selaimen kautta palvelimelle. Pyyntö saapuu palvelimelle (application system) Event Dispatcher-komponentille. Event Dispatcher purkaa pyynnön ja koordinoi liitoskohdan (socket) kahvoja (handles) ja säikeitä, joita käytetään vastaanottamaan ja prosessoimaan sivupyyntöjä. Jokainen sivupyyntö prosessoidaan Protocol Handler –komponentissa, joka jäsentää ja kirjoittaa pyynnön lokiin, noutaa tiedoston tilatiedot (status) ja päivittää File Cache –komponentin, lähettää tiedoston takaisin asiakkaalle ja poistaa kaikki pyyntöön liittyvät resurssit. File Cache -komponentti toimii virtuaalisena tiedostojärjestelmänä, jossa sijoitetaan pyydettyjä sivuja hierarkisesti siten, että useimmin pyydetyt sivut ovat noudettavissa sieltä. Tämä menettelyjä vähentää tiedostojärjestelmään kohdistuvia kyselyjä ja parantaa siten tehokkuutta [ScR01 s. 24].



Kuva 24. Tyypillinen Web-systeemi ja sen komponentit, mukaillen [ScR01 s. 24]

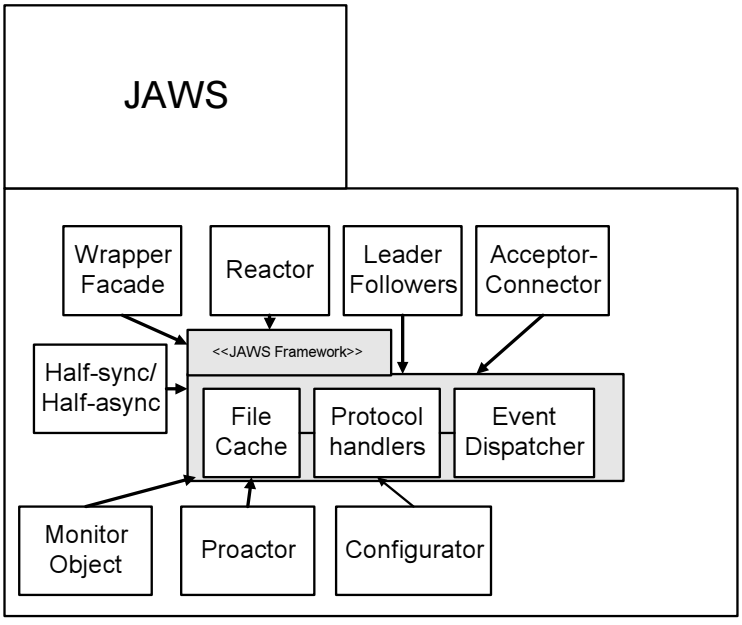
Mallien avulla voidaan välttää tavallisimmat ongelmat rakennettaessa web-palvelinta. Ongelmat ovat yleensä matalan tason yksityiskohtia ja aiheuttavat rajoituksia siirrettävyyteen ja tekevät kyseisen palvelimen konfiguroinnin vaikeaksi. Ongelmien ratkaisuina on tarjolla useita erilaisia vaihtoehtoja, joiden avulla ominaisuuksia voidaan säädellä

tarpeiden mukaisesti. Samanaikaisuusmalleja (concurrency models) ovat säie per pyyntö (thread per request) – tai säieallas (thread pool) –malli. Tapahtuman purkumallit (event demultiplexing models) ovat joko synkronisia tai asynkronisia. Tiedoston säilytysmalleja (file caching models) ovat taas last in first out (lifo) tai first in first out (fifo) -mallit. Protokollia voivat olla esimerkiksi http/1.0 tai http/1.1 -protokollat.

JAWS-projektin haasteita ovat [ScR01 s. 27]:

- pyrkimys irrottautua matalantason rutiineista
- protokollan käsittelyn ja tapahtumanhallinnan erottaminen
- suorituskyvyn parantaminen säikeiden avulla
- synkronoidun kyselyjonon käyttäminen
- palvelimen säikeistyksen minimointi
- pyrkimys tehokkaaseen asynkronisen viestintään (input/output)
- luoda konfiguroitavissa oleva palvelin

JAWS-projektissa palvelimen komponenttien implementointi toteutettiin kahdeksan mallin avulla. (Kuva 25). Malleista on kuvattu staattiset luokkamallit liitteessä 8.



Kuva 25. JAWS Framework, Framework-komponentit ja komponenttien implementointi mallien avulla mukaillen [ScR01 s. 27-41]

JAWS-kehys implementoidaan arkkitehtuuri- ja suunnittelumallien avulla [ScR01 s. 24-42]:

- Wrapper Facade–mallin avulla vältytään matalan tason API-kutsujen suora käsittely ja voidaan nostaa abstraktiota. JAWS:ssa mallia käytetään siten, että sen avulla kehyksen komponentit voivat suorittaa tehtävänsä erilaisilla käyttöalustoilla kuten esimerkiksi UNIX-, Windows- tai Linux-alustoilla.
- Reactor-mallin ja Acceptor-Connector-mallin avulla erotetaan tapahtumankäsittely ja protokollanhallinta toisistaan. Reactor-malli toimii synkronisesti. Asiakaspyyntö tulee palvelimelle. Synkroninen tapahtumanpurkaja huomaa tulleen pyynnön ja ilmoittaa siitä Reactor-komponentille, joka puolestaan purkaa ja lähettää nämä tapahtumat nimitetylle tapahtumankäsittelijälle edelleen prosessoimista varten. Tapahtumankäsittelyssä luodaan ja liitetään tapahtumankäsittelijä pyyntöä varten. Sen jälkeen luotu http_handler-käsittelijä vastaanottaa ja käsittelee koukkumetodinsa avulla tulleen pyynnön.
- Half-sync/Half-async -mallin avulla voidaan skaalata palvelimen suorituskykyä prosessoitaessa eri HTTP pyyntöjä samanaikaisesti useissa säikeissä. Tämä malli määrittelee kaksi prosessointitasoa – yhden asynkronisen ja yhden synkronisen tason – mukaanlukien jonotason (queue), joka ansiosta palveluita voidaan vaihtaa eri kerrosten välillä. Malli sallii synkronisten palveluiden kuten http:n prosessoinnin samanaikaisesti (concurrently) omissa työsäikeissään. http:n prosessointisäikeen ns. säiealtaan tehtävänä on ottaa kyselyjonosta (request queue) sivupyyntö ja käsitellä se ja lähettää vastaus I/O-porttiin. Tämä menettely on käyttöjärjestelmä sidonnainen. Kun työsäie haluaa lähettää vastuksen asiakkaalle, esitetään palvelimen taholta käyttöjärjestelmälle pyyntö uuden yhteyden luomiseksi (endpoint) asiakasta varten. Sen jälkeen luotu kahva on työsäikeen käytettävissä, ja kommunikointi asiakkaan kanssa voi alkaa eli sivu lähetetään asiakkaalle. Asynkroniset palvelut suoritetaan kokoamalla tapahtumat käsiteltäväksi pyyntöjonoa varten. Menettely voi olla esimerkiksi tapahtua siten, että Reactor-mallin avulla luetaan ja kootaan tapahtumanlähteeseen (soketti-abstraktio) tuleva pyyntö. Pyyntö lähetetään tapahtumankäsittelijöille, jotka laittavat vastauksen pyyntöjonoon (request queue) odottamaan lähettämistä. Sen jälkeen jonosta kukin työsäie saa vuorollaan valmistetun sivun ja voi lopuksi lähettää sivun takaisin asiakkaalle.

- Monitor Object-mallin soveltaminen *synkronoituun kyselyjonoon (synchronized request queue)* synkronoi metodin suorittamisen ja takaa sen, että vain yksi metodi ajetaan kerrallaan palvelimen kyselyjonossa.
- Leader/Followers -mallin tarkoituksena on palvelimen säiekustannusten minimoiminen. Tämä arkkitehtuurimalli tarjoaa tehokkaan samanaikaisuutta tukevan mallin, jossa useat säikeet vuorottelevat jakaessaan tapahtumanlähteitä. Tämä malli eliminoi tarpeen erottaa säikeenä suoritettava Reactor-elementti ja synkronoitu kyselyjono (request queue) JAWS palvelimessa.
- Proactor-malli tarjoaa keinot käsitellä asynkronisesti kyselyjä ja vastauksia niissä käyttöjärjestelmissä, jotka tukevat asynkronista input/output-käsittelyä tehokkaasti. Tämä arkkitehtoninen malli rakentuu samanaikaisista tapahtumaohjatuista palvelinsovelluksista, jotka vastaanottavat ja prosessoivat kyselyjä useilta asiakkailta asynkronisesti. Sovelluspalvelut on jaettu kahteen osaan. Ensiksi asynkroniset operaatiot hyväksyvät yhteyksiä ja vastaanottavat asiakkaan HTTP GET pyyntöjä. Toiseksi käsittelijä prosessoi asynkronisen operaation tulokset välittäen tiedoston asiakkaalle sen jälkeen, kun asynkroninen yhteysoperaatio on valmis. Aivan kuten Reactor-mallissa Proactor-malli irrottaa tapahtuman purkamisen ja tapahtumankäsittelyn palvelinsovelluksen logiikasta. Pääasiallinen ero on siinä, että Proactor käsittelee loppuunsaatettuja tapahtumia, jotka ovat tuloksia *asynkronisista* operaatioista, kun taas Reactor käsittelee *synkronisten* operaatioiden laukaisemia tapahtumia.
- Component Configuration –malli laajentaa WEB-palvelimen konfiguroitavuutta. Tämän mallin avulla voidaan sovelluksen komponentit linkittää ja poistaa linkityksestä ajonaikana. Uudet lisäpalvelut voidaan siten lisätä ilman, että olisi modifioitava, uudelleenkäännettävä, staattisesti uudelleenlinkitettävä tai ajettava alas ja uudelleenladattava ajettavaa sovellusta.
- Lisäksi on käytetty muita malleja, joilla lisätään joustavuutta ja modulaarisuutta (flexibility ja modularity). Thread-Safe Inteface ja Strategized Locking-mallit auttavat minimoimaan lukkiumia JAWS:n virtuaalisessa tiedostojärjestelmässä. Käytettyjä suunnittelumalleja JAWS:ssä ovat Adapter-, Bridge-, Factory Method-, Iterator-, Template Method- ja Proxy-suunnittelumallit. Muita tarvittavia ovat Singleton-, State- ja Strategy-mallit.

4.2.4 Yhteenveto

Hajautettujen ja verkottuneiden sovellusten suunnittelu on verraten monimutkaista. Ratkaistavia ongelmia on paljon. Kilpailukykyinen tuotanto ei onnistu millään ad-hoc –menetelmällä. Tarvitaan selkeä kartta. Tarvitaan selkeä rakenne. Arkkitehtuurien malliratkaisujen avulla saavutetaan arkkitehtuuriin selkeyttä ja syntyvään koodiin ryhtiä. Mallit, kehykset ja komponentit luovat selkeän ohjelmistorakenteen. Selkeys on ennuste siitä, että ohjelmiston elinkaari on pitkä ja se kestää hyvin elinkaarensa aikana tapahtuvan kehityksen ilman, että ohjelmiston rakenne degeneroituu sen suunnitellun elinkaaren aikana.

4.3 Arkkitehtuuri ja sidosryhmät – dokumentoinnin haasteet

Arkkitehti valitsee dokumentoitavan näkymän. Ongelmana on, mikä näkymä valitaan dokumentaatioon. Jotta voitaisiin päättää mitä näkymiä ja milloin ja minkä tasoisia tarvitaan, on tiedettävä ketä varten dokumentoidaan, mitä resursseja on käytettävissä, mikä on budjetti, mikä on aikataulu ja ketkä ovat tärkeitä asianosaisia. Dokumentaatiosta luodaan jokaista asianosaista varten räätälöity paketti.

Asianosaisia (stakeholders) on projektissa useita ja heidän välillään voi olla esimerkiksi seuraavia tehtäviä. Projektinjohto, arkkitehti ja asiakas neuvottelevat ohjelmistovaatimukset. Arkkitehti ja suunnittelijat määrittelevät, mitä komponentit sisältävät, mikä on suorituskyky ja mikä ajoaikainen resurssien kulutus. Toteuttajat tarvitsevat tietoa siitä, mitä rajoituksia on ja mitä vapauksia heille suodaan kehitystoimissa. Testaajat ja integroijat saavat tietoa elementtien käyttäytymisestä ja määrittelyn siitä kuinka eri elementit sopivat yhteen. Ylläpitäjille arkkitehtuurin dokumentaatio on lähtökohta ylläpitoprosessin toiminnoille.

Dokumentaatio valottaa niitä alueita, joihin mahdolliset muutokset vaikuttavat. Muiden yhteistoimintaa harjoittavien ulkopuolisten systeemien suunnittelijat tarvitsevat dokumentaatiota, jotta voivat määritellä tarjottavia ja vaadittavia palveluita ja niiden protokollia. Projektipäälliköille dokumentaatio on lähtökohta kehitystiimin rakentamisessa, ohjelmiston muotoilussa ja projektisuunnittelussa sekä eri tiimien edistymisen seurannassa. Tuotelinjapäälliköt voivat havaita, tarvitaanko tuotelinjaan uusia jäseniä. Ja lopuksi laadunvarmistusryhmälle dokumentaatio on lähtökohtana tarkistuslistoille, jotta

voidaan varmistua siitä, että toteutus on onnistuneesti suoritettu arkkitehtuurikuvausten mukaisesti [CleF02] .

Sidosryhmien vaatimukset poikkeavat toisistaan. Informaation sisältö kutakin varten on suhteutettava projektin kokoon. Dokumentaatio on laadittava niin, että tarvittava informaatio on helposti irrotettavissa ja tulostettavissa. Kyse on myös tiedottamisesta. Yrityksen laatukäsikirjan tulee sisältää määräykset yrityksen tiedotuspolitiikasta. Siinä tulee määritellä arkkitehtuurista, mitä dokumentoidaan, kenelle dokumentoidaan ja kuka on vastuussa dokumentoinnista ja sen jakelusta. Ei voida ajatella, että joka projektissa jouduttaisiin erikseen miettimään sitä, kuinka dokumentaatio laaditaan. Toiminnan tulee olla organisoitua.

4.4 Vaatimusten hallinta ja ohjelmistoprosessin organisointi

4.4.1 Yleistä

Parantaessaan toimintaansa ja kilpailukykyään yritys joutuu tarkastelemaan toimintaansa yhtenä kokonaisuutena. Ensiksi tarkastellaan liiketoiminnan tilaa. Yrityksen organisaation toiminnan on muututtava (transformation) jäykästä organisaatio- ja osastojaosta prosessikeskeiseksi. Käytetään ilmausta *reengineering,* jolla tarkoitetaan toiminnan uudistamista. Organisaation toiminnan muutos mahdollistaa myös ohjelmistotuotantoprosessin kehittämisen. Ohjelmistoprosessi koostuu ohjelmiston luokittelemisesta, ohjelmistomoduulien määrittelemisestä, laadun määrittelemisestä sekä ohjelmistotuotantoprosessin luomisesta. Ohjelmistokehityksen lähtökohtana on kohdealueen analyysi (domain analysis). Analyysin avulla löydetään liiketoimintaprosessit, joista voidaan mallintaa ohjelmistoprosessit ja tehdä päätöksiä kehityskelpoisista ohjelmistoprojekteista [Ber95].

Ohjelmistotuotannon hallitsemiseksi on tuotteen vaatimustenkäsittely (requirements management) kyettävä hallitsemaan ja prosessi osattava organisoida kilpailukykyisesti. Vaatimuksia asettavat kaikki asianosaiset. Asiakkaan vaatimusten löytäminen, analysointi ja hyväksyttävien vaatimusten valitseminen projektiin muodostavat yhden kokonaisuuden. Muita vaatimuksia esittävät projektin johto, projektihenkilöstö, käyttäjät, ylläpito ja ulkopuoliset sidosryhmät. Myös nämä muut vaatimukset on pystyttävä käsittelemään. Vaatimusten asettajien vaatimukset ovat usein ristiriidassa keskenään. Yhden

asianosaisen vaatimuksen täyttäminen toisaalla voi heikentää toisen asianosaisen vaatimusten toteuttamista.

Ohjelmistoprojekti on perinteisesti nähty yhtenä kokonaisuutena, joka on toimitettu sovittuun ajankohtaan mennessä asiakkaalle. Tämä lähestymistapa on luonteva tavanomaisten ja verrattain pienten projektien toteuttamisessa. Ongelmana tässä lähestymistavassa on se, että toimitettava projekti on yksi kokonaisuus ja se toimitetaan yhdellä kertaa. Lisäksi koska sovittava asia on verraten yksinkertainen, on siitä voitu tehdä kiinteähintainen sopimus osapuolten kesken. Tällainen sopimuskäytäntö sisältää suuria riskejä. Neuvotteluvarana osapuolten kesken on toimitusaika ja hinta. Tämä johtaa vääjäämättä siihen, että hanke ei saa riittävästi resursseja ja aikataulu jää tiukaksi. Jokaisessa projektipalaverissa asialistalla joudutaan keskustelemaan aikataulusta ja miettimään keinoja kuinka aikatauluviiveet saataisiin kiinni. Koska maksimikate saadaan minimimiehityksellä, on suuri vaara, että projekti ei saa tarvitsemiaan resursseja. Tällöin suunnitteluvaiheessa ja testausvaiheessa hanke ei saa riittäviä resursseja. Laadusta ei puhu kukaan. Tällöin onnistunut projekti on sellainen, että se jollain keinolla saadaan toimitettua asiakkaalle.
Mitä asiakas saa? Asiakas saa luultavasti ohjelman, joka juuri ja juuri täyttää asiakkaan sopimukseen kirjaamat vaatimukset. Kaiken lisäksi ohjelmatoimitus myöhästyy. Ohjelma ehkä saadaan toimimaan ajoissa, mutta havaittujen virheiden ja niiden korjausten takia, ohjelma saadaan tuotantokäyttöön myöhässä.

Edellä kuvattu on tiedossa. Sitä on yritetty ratkaista laatimalla *projektille osatavoitteita (milestones)*, joiden avulla toimituksen etenemistä voidaan vaiheistaa. Toisena keinona pyritään parantamaan vaatimustenhallintaa. Kyetään arvioimaan, mitkä ovat vaatimukset: *toiminnalliset ja ei-toiminnalliset vaatimukset (functional and non-functional requirements)*. Hallitaan myös systeemin asettamat vaatimukset (kts. 4.4.4 CMM taso 2).

Vaatimustenhallinnan lisäksi on projektinhallinta järjestettävä asianmukaisesti. Tuotantoprosessi on suunniteltava siten, että se ottaa huomioon laatutekijät ja niiden merkityksen tuotteessa. Yksi ratkaisu on *iteratiivinen prosessin suunnittelu (iterative process planning)* [Roy98 s.139]. Lähtökohtana on arkkitehtuuriin nojaava ajattelu ja pyrkimys jakaa hanke alitehtäviin siten, että niitä voidaan toteuttaa rinnakkain ja samanaikaisesti. Hanke ja sen osatehtävät toteutetaan iteratiivisesti, jolloin tehtävän tulos parantuu ite-

roimalla. Tällöin joudutaan kohdistamaan toimenpiteitä prosessin parantamiseen (kts. 4.4.4 CMM taso 3).

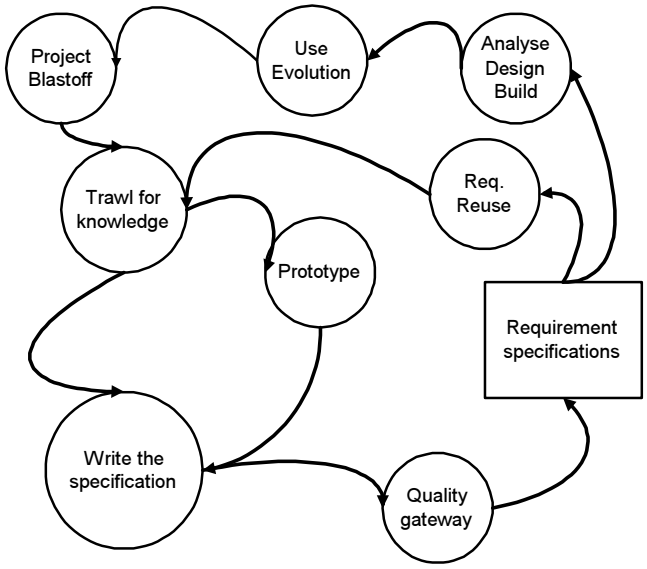
Yrityksellä tulee olla laatupolitiikka ja asianmukainen laatukäsikirja, johon on kirjattu yrityksen keskeiset tavoitteet ja keinot tavoitteisiin pääsemiseksi. Laadunhallinnassa toiminnan arviointi ja mahdollisten kehityskohteiden etsiminen voidaan suorittaa vertaamalla omaa toimintaa johonkin vertailukohtaan. Projektin on kyettävä mittaamaan ja arvioimaan toimintaansa. Projektin kypsyyttä voidaan mitata esimerkiksi Capability Maturity Model (SEI CMM) mallin avulla [Roy98 s. 363].

4.4.2 Vaatimusten määrittely

Vaatimusten määrittelyssä voidaan erottaa kaksi tehtävää: vaatimusten kerääminen ja toteutettavien vaatimusten valinta. Rakennettavan systeemin vaatimukset esittää viime kädessä asiakas. Asiakkaan vaatimusten lisäksi vaatimuksia esittävät loppukäyttäjät, ylläpito, suunnitteluorganisaatio sekä ulkopuoliset henkilöt. Järjestelmää rakennettaessa valitaan *kerättyjen vaatimusten (requirements capture)* pohjalta ne vaatimukset, joiden perusteella järjestelmä rakennetaan. Vaatimukset kirjataan vaatimusmäärittelyyn, jonka asiakas hyväksyy.

Aina ei ole valmiutta laatia hankkeelle valmista muuttumatonta määrittelyä, vaan määrittely elää ja kehittyy projektin kuluessa. Tämän vuoksi hanke on pystyttävä organisoimaan niin, että projektin kuluessa tulleet kehitys- ja parannusehdotukset voidaan toteuttaa. Vaatimusmäärittelyprosessi kestää jossain muodossa läpi tuotteen koko iän, rakentamispäätöksestä ohjelmiston poistamiseen käytöstä (Kuva 26). *Projektin aloitusvaiheessa (blastoff)* kuvataan konteksti, määritellään tavoitteet ja tarkoitus, kirjataan asianosaiset, määritellään liiketoimintatapahtumat sekä projektille asetettavat rajoitukset esim. olemassa olevat perinnejärjestelmät tai organisatoriset rajoitukset. *Perehtymisvaiheessa (trawl for knowledge)* perehdytään tuotteeseen, laaditaan käyttötapausesimerkkejä, tutkitaan ja löydetään potentiaaliset vaatimukset ja käyttötapaukset. Vaatimuksia joudutaan joskus havainnollistamaan ja miettimään, mikä on oleellista ja mikä on vähemmän tärkeää. Apuna vaatimusten muotoilemisessa voidaan käyttää esimerkiksi prototyyppejä (prototype) tai skenaarioita (scenario). Protoilun avulla tuotteesta tai jostain sen osa-alueesta rakennetaan malli, jonka avulla voidaan havainnollistaa käsiteltävää asiaa. Skenaarion avulla voidaan käsiteltävää työtä havainnollistaa mallintamalla sitä käsitteellisin keinoin käyttäen syötteenä esimerkiksi havaintomateriaalia.

Kaikki potentiaaliset vaatimukset *kirjataan vaatimusmäärittelyyn (write the specification),* jonka jälkeen suoritetaan *laadullinen analyysi (quality gateway),* jonka tarkoituksena on punnita löydetyt vaatimukset ja kirjata valitut vaatimukset formaalisti laadittavaan määrittelyyn. Vaatimusmäärittelyyn kirjattavan vaatimuksen tulisi olla sisällöltään täsmällinen, tarkka ja hyödyllinen. Esimerkiksi, jos jokin laadullinen vaatimus suorituskyvystä on "suorituskyvyn tulee olla mahdollisimman hyvä", ei tällaista kannata kirjata vaatimusmäärittelyyn, koska sitä ei voi todentaa. Jokaisen vaatimuksen osalta testataan, onko se täydellinen, jäljitettävä, tarpeellinen, ristiriidaton, elinkelpoinen, ratkaistavissa, onko vaatimus kohtuullinen sekä kuuluko vaatimus sovittuun toimitukseen [Rob99 s. 181].



Kuva 26. Vaatimustenmäärittelyprosessi mukaillen [Rob99 s. 11]

Vaatimusten perusteella rakennettavalle systeemille laaditaan käyttötapaukset. Käyttötapaukset dokumentoidaan. Käyttötapauksen suunnittelu muodostaa jäljitettävän ketjun läpi koko suunnitteluprosessin, jolloin dokumentoitua käyttötapausta voidaan myöhemmin käyttää uudelleen toisessa projektissa.

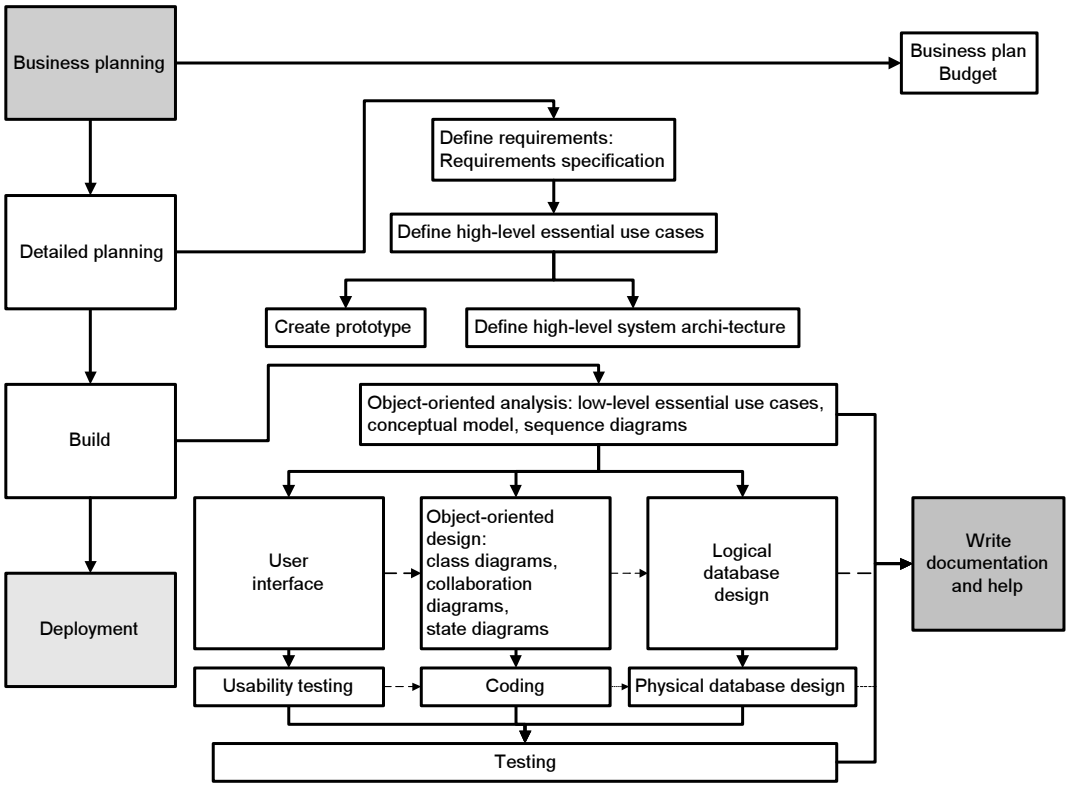
Arkkitehtuurisuunnittelun hankalin vaihe on asiakkaan vaatimusten muuttaminen arkkitehtuuriksi. Toiminnallisten vaatimusten kuvaaminen on helpointa. Asiakas osaa yleensä kuvata, mitä toimintoja tarvitaan ja miten ne suoritetaan. Laadullisten ominaisuuksien tunnistaminen, arvioiminen ja niiden muuttaminen toiminnallisiksi ominaisuuksiksi on vaikeinta.

4.4.3 Prosessin mallintaminen

Herzumin mukaan on komponenttipohjaisen liiketoimintaprosessin kehitysprosessissa otettava huomioon seuraavia tekijöitä: komponenttipohjainen lähestymistapa (component-centric approach), arkkitehtoninen lähestymistapa (architecture-centric approach), komponenttien autonomia (focus on autonomy of components), yhteistyönäkökulma (approach to collaborations), iteratiivinen luonne (iterative nature), jatkuva integraatio (continuous integration) sekä riskienhallinta (risk management) [HeS99 s. 51] (kts. Liite 1).

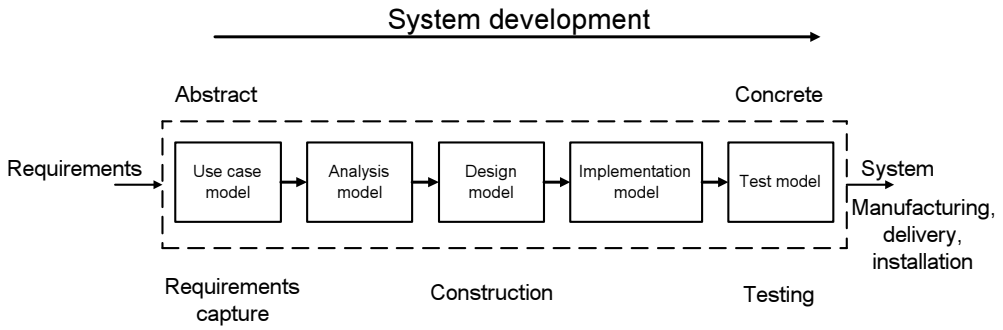
Eräs malli kuvaa liiketoimintapohjaisen ohjelmistokehityksen elinkaaren neljällä vaiheella, jotka ovat liiketoiminnan suunnittelu (business planning), yksityiskohtien suunnittelu (detailed planning), rakentamisvaihe (build) sekä toimitusvaihe (deployment). Tämä malli lähtee liiketoiminnan suunnittelusta, josta saadaan vaatimukset ohjelmistoprojektille. Malli kuvaa analyysi- ja suunnitteluvaihetta (detailed planning) prosessina, jossa korkean tason käyttötapauksista muodostetaan protyyppejä ja korkean tason arkkitehtuuri. Nämä käyttötapaukset jaetaan osiin tarkemmiksi käyttötapauksiksi (Kuva 27) [Gra01 s. 30].

Suunnittelussa korostuu vaatimustenhallinta ja -määrittely sekä arkkitehtuurin luonnostelu. Protoilun avulla tulisi pyrkiä siihen, että ennen rakentamisvaihetta voitaisiin arvioida suoritettuja olettamuksia ja varmistua siitä, että vaihtoehdoista valitaan paras mahdollinen. Käytettävyys ymmärretään lähinnä käyttöliittymän käytettävyydeksi, mikä on liian suppea näkökulma [Imm03a].



Kuva 27. Ohjelmiston elinkaaren toimenpiteet liiketoimintasuunnitelmasta toimitusvaiheeseen saakka mukaillen [Gra01 s. 30]

Ohjelmiston elinkaareen kuuluvat lisäksi käytönaikainen ylläpito ja evoluutio. Ylläpidon tarkoituksena on varmistua siitä, että ohjelmisto säilyy toimintakykyisenä mahdollisimman pitkään. Siihen kuuluvat tuotteenhallinta, havaittujen virheiden korjaus, pienet tuotteen parannustoimet. Aika-ajoin tuotteeseen suoritetaan kehitystoimenpiteitä kehitysprojektin avulla ja tuotteesta julkaistaan uusi versio. Versionhallinnan avulla varmistetaan eri versioiden yhteensopivuus ja asiakkuudenhallinta siten, että tiedetään, mitä versioita on milläkin asiakkaalla. Perinteinen vesiputousmallin mukainen (Kuva 28) kuvaus lähtee vaatimuksista päätyen valmiiseen tuotteeseen.



Kuva 28. Ohjelmistoprosessi [JGP97 s. 56]

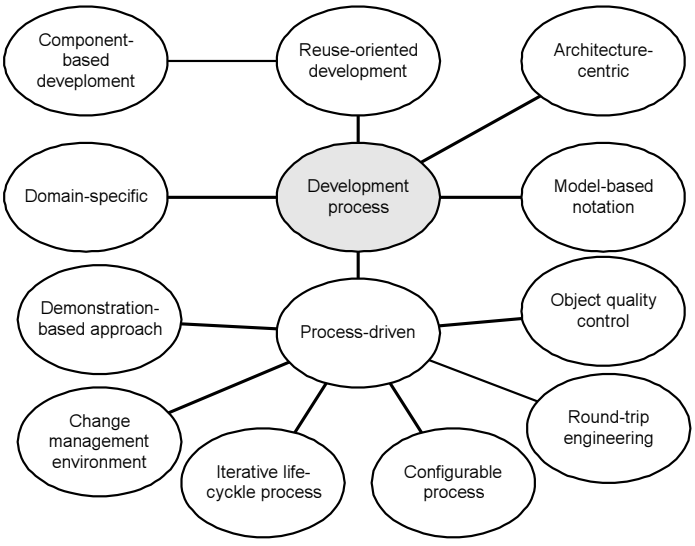
Prosessin mallintamisen avulla toiminta voidaan jakaa helpommin käsiteltäviksi osatehtäviksi. Perinteisin ns. vesiputousmalli toteuttaa tuoteprojektin yhtenä alusta loppuun tapahtuvana prosessina, jossa lähdetään vaatimusten määrittelystä ja analysoinnista, suunnittelun ja mennään implementoinnin ja testauksen kautta käyttöön ja ylläpitoon [Roy98]. Organisaation tulisi toiminnassaan noudattaa systemaattisesti ns. *"Best practice"-menettelyä* (kts. Liite1). Virheenä on pidettävä sitä, jos huomio kohdistetaan vain meneillään olevaan projektiin. Huomion tulisi kohdistua kohdealueeseen kokonaisuutena [JGP97 s. 8].

Tuotantoprosessin tulee tukea uudelleenkäytettävyyttä, olla arkkitehtuuria hyödyntävä sekä tarkastella kohdealuetta ja käyttää hyväksi havaittuja komponentteja. Prosessia tulee tarkastella useasta näkökulmasta. *Arkkitehtuuri ensin lähestymistapa (architecture-first approach)* on keskeinen suunnitteluelementti, jossa jaetaan arkkitehtuuri komponentteihin huomioiden elementtien uudelleenkäytettävyys. *Uudelleenkäyttösuuntautunut kehittäminen (reuse-oriented development)* on yksi tapa parantaa tuottavuutta. Tällöin vaatimusmäärittelyyn tehdään komponenttianalyysi, jossa tutkitaan, mitä projektiin sopivia komponentteja komponenttivarastosta löytyy. Komponenttianalyysi liittyy läheisesti arkkitehtuurinmäärittelyprosessiin. Komponenttianalyysin pohjalta tehdään vaatimusmäärittelyyn tarvittavia muutoksia, jotta saatavilla olevia komponentteja voitaisiin käyttää projektissa. Tämän jälkeen systeemi suunnitellaan ottaen huomioon uudelleenkäytettävyys [SOM01 s. 49]. *Kohdealuespesifinen (domain-specific)* tarkastelutapa lähtee siitä, että uudelleenkäyttö on kannattavinta kohdealueen sisällä eikä kohdealueiden välillä. *Prosessisuuntautunut (process-driven)* tarkastelutapa lähtee siitä, että aivan aluksi on luotava prosessi, joka on toistettavissa, jotta sen lopputulos voidaan ennustaa. Prosessia pyritään parantamaan ja sen kypsyyttä arvioimaan [JPG97 s. 25]. *Iteratiivinen elinkaariprosessi (iterative life-cyckle process)* sisältää myös riskien aikaisen hallinnan. Iteratiivisuutta käytetään keskeisenä riskienhallinnan elementtinä.

Suunnittelumetodeja muutetaan suosimaan *komponenttipohjaista kehittämistä (component-based development)*. Koodipohjaisesta ajattelusta on päästävä komponenttipohjaiseen ajatteluun, jotta omaa generoitavaa koodia ja asiakaskohtaista räätälöintiä voitaisiin vähentää. Tämä on tärkein teknologinen elementti, jonka avulla ohjelmiston rakennetta voidaan yksinkertaistaa. Muutosten hallitsemiseksi perustetaan *muutostenhallintaympäristö (change management environment)*. Korostetaan muutoksen vapautta sellaisten työkalujen avulla, jotka tukevat *edestakaista suunnittelua (round-trip enginee-*

ring). Tämä on välttämätöntä automatisoitaessa ja synkronoitaessa eri muotoisen informaation suunnittelua mm. määrittelyä, mallintamista, lähdekoodia, ajokelpoista koodia ja testitapauksia. Suunnittelutekijät mallinnetaan *mallipohjaisella notaatiolla (model-based notation)* käyttäen esimerkiksi UML-pohjaisia ohjelmistokuvaustapoja. Luodaan *prosessiin laatukontrolli (Objective quality control)* ja arvioidaan kehitystä. Käytetään *protoilua (demonstration-based approach)* arvioitaessa tehtyjä ratkaisuja. Luodaan prosessi *konfiguraatioiden hallitsemiseksi (configurable process)* [Roy98 s. 63] (kts. liite 1).

Kuvassa 29 on kuvattu ohjelmistokehitysprosessin eri osatekijät. Lähestymistapa ohjelmistoprojektiin koostuu näistä näkökulmista. Painotukset voivat vaihdella projektista toiseen. Yhdessä projektissa voidaan painottaa komponenttien uudelleenkäytettävyyttä, toisessa projektissa päähuomio kiinnitetään arkkitehtuuriin ja pyritään arkkitehtuurin muokkaamisen kautta saavuttamaan vaadittavat ominaisuudet.



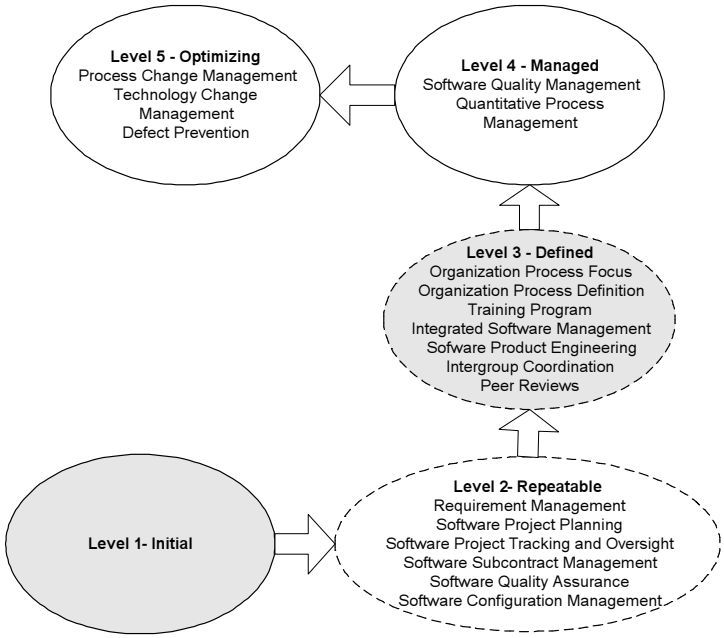
Kuva 29. Ohjelmiston kehitysprosessin näkökulmat

Prosessin mallintamisen avulla projektille on luotu prosessi, jolla on aikaulottuvuus, tehtävät ja niiden aikataulu, tehtävien jakaantuminen osatehtäviksi ja niiden suoritustavoitteet, suunniteltu iterointi, välitavoitteet ja niiden seuranta sekä projektin budjetointi ja miehitys. Esimerkkinä tällä tavoin toimivista prosessimalleista on esimerkiksi RUP (Rational Unified Process) [www006].

4.4.4 Prosessin kypsyyden arviointi

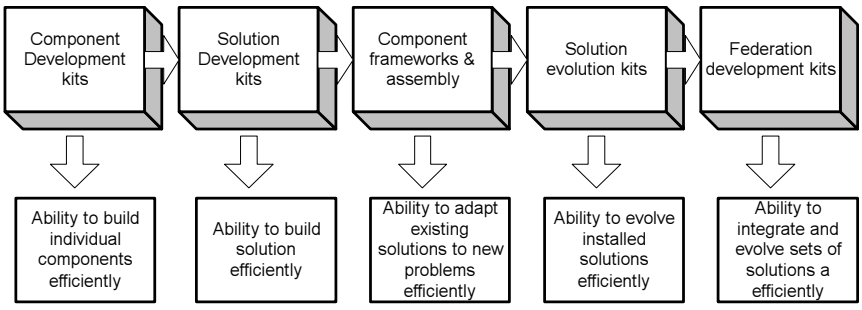
Organisaation tulee kyetä arvioimaan toimintaansa. Mikä on hyvää, mikä on huonoa ja mitä tulisi kehittää. ISO 9000 –standardisarja on kehys, jonka avulla yritys voi järjestää tuotantoaan. Standardi perustuu sertifiointiin ja jatkuvaan toiminnan arviointiin eli auditointiin, jonka avulla pyritään varmistumaan siitä, että toiminta täyttää sertifioinnin vaatimukset. Kypsyyden arvioimiseksi on kehitetty eri menetelmiä. Capability Maturity Model (CMM) –malli on yksi käytetty menetelmä kypsyyden arvioimiseksi. CMM sopii erikoisesti suurten projektien kypsyyden arviointiin. CMM-mallissa toimintaa tarkastellaan viidellä tasolla [Roy98 s. 363]. ISO:n työryhmä on kehittänyt SPICE-mallin, joka pohjautuu mm. CMM:ään ja jonka tarkoituksena on arvioida ja parantaa ohjelmistotuotantoprosessia luomalla viitemalli ja kehys, joiden avulla prosessia voidaan arvioida [EDM98].

CMM:n tasolle yksi (initial) pääsee jokainen yritys. Arvioidaan, että 70% yrityksistä on tällä tasolla. Ominaista tälle tasolle ovat kypsymättömät prosessit ja siitä johtuen valmius ottaa suuria riskejä toiminnassa. On selvää, että tällä tasolla toimiva yritys ei kykene toimimaan tehokkaasti ja innovatiivisesti hyödyntäen komponenttiteknologiaa tai arkkitehtuuria. Yrityksen tulisi vähintään kyetä hallitsemaan vaatimustenhallinta, arvioimaan ja parantamaan laatua sekä hallitsemaan projektinhallinnan perusasiat. Tasolle kaksi päästään, jos toiminta on toistettavissa (repeatable), ohjelmistotuotanto on kyetty organisoimaan esimerkiksi vesiputousmallin mukaisesti ja katselmointi ja sopimuskäytännöt osataan. Tällä tasolla arvioidaan olevan 15% yrityksistä. Kolmannella tasolla taas toiminta on määriteltyä (defined), jolloin toiminta on vakaata, ennustettavaa ja toiminnassa tapahtuu kehittymistä. Mielenkiinnon kohteina ovat mm. organisaation kehittäminen ja prosessien määritteleminen. Tällä tasolla voidaan odottaa projektien hallintaa niin, että hallitaan paremmin kustannukset, aikataulutus ja laadunhallinta. Tällä tasolla on alle 10% yrityksistä. Kun toiminta on johdettua (managed), ollaan tasolla neljä. Luonteenomaista tälle tasolle on kyky mitata ja analysoida prosessia, laatujohtaminen ja virheenhallinta. Vain alle 5% yrityksistä on tällä tasolla. Viidentenä ja ylimmäisenä tasona on kyky optimoida toimintansa (optimizing), jolloin ollaan jo siinä pisteessä, että yritys kykenee hyödyntämään teknologiset innovaatiot, on innovoiva itsekin ja hallitsee muutosjohtamisen. Vain alle 1% yrityksistä yltää tälle tasolle (Kuva 30) [Roy98 s. 363]



Kuva 30. CMM-malli kypsyystasot [Jal99 s. 7]

Herzum jakaa komponenttipohjaisen tuotannon kypsyyden viiteen vaiheeseen, jossa tasot kuvaavat yrityksen ongelman ratkaisukykyä. Alimmalla tasolla hallitaan komponenttiteknologia. Toisella tasolla saavutetaan kyky rakentaa ratkaisuja uusille kohdealueille tehokkaasti. Kolmannella tasolla yrityksellä on kyky mukauttaa olemassa olevat ratkaisut uusiin ongelmiin tehokkaasti. Mahdollisesti voidaan käyttää komponenttikehyksiä, komponenttimalleja, komponenttikirjastoja ja jopa esiasennettuja ratkaisuja jokaista kohdealuetta varten. Neljännellä tasolla ollaan, kun on kyky kehittää asennettuja sovelluksia tehokkaasti. Ja lopuksi viidennelle tasolle päästään, kun on saavutettu kyky integroida ja kehittää ratkaisujoukkoja useita yhdessä toimivia systeemeitä varten (Kuva 31) [HeS99 s. 30].



Kuva 31. Komponenttituotannon kypsyystasot [HeS99 s. 30]

Kypsyydenarviointimallien avulla organisaatio voi arvioida toimintaansa ja miettiä keinoja, joilla toimintaa voisi parantaa. Organisaatioiden toiminnassa ongelmana on se, kuinka organisaatio jatkuvasti kykenisi tutkimaan toimintaansa ja uusiutumaan. Organi-

saatiot ovat luonnostaan konservatiivisia. Tehtävät halutaan tehdä, kuten ne on ennenkin tehty. Muutosvastarinta on suuri varsinkin, jos uuden toimintatavan hyötyjä ei kyetä näkemään. Tästä johtuen organisaation muutosvastarinta on huomioitava kiinnittämällä huomiota sisäiseen markkinointiin. Toiminnan muutokset on pystyttävä perustelemaan ja syntyvät hyödyt osoittamaan myös yksilötasolla. Toiminnan kehittäminen vaatii huolellista suunnittelua, riittävää panostamista ja johdon sitoutumista hankkeeseen.

4.4.5 Yhteenveto

Vaatimustenhallinta ja projektinhallinta vaativat koko organisaation kattavaa suunnittelua. Vaatimustenhallinta ei tarkoita ainoastaan, että vaatimukset löydetään ja osataan laatia asianmukainen dokumentti suunnittelun pohjaksi. Vaatimustenhallinta tarkoittaa myös sitä, vaatimuksia kyetään käsittelemään ja täsmentämään jopa muuttamaan vaatimuksia projektin aikana, jos se katsotaan järkeväksi.

Ohjelmistoprojekti toteutetaan tiimityönä, se koostuu useista osatehtävistä ja siinä käytetään useita elementtejä. Prosessia tulee kyetä parantamaan, komponenttien ja prosessien uudelleenkäyttö tulee maksimoida, projektin resurssit osata käyttää ja projektinhallinta hoitaa. Perinteisestä vesiputousmallista on siirrytty iteroivimpiin, kehittävämpiin (iterative, incremental) toimintatapoihin. Organisaatiot arvioivat toimintansa kypsyyttä ja pyrkivät parantamaan toimintamallejaan.

4.5 Arkkitehtuurin suunnittelu

4.5.1 Yleistä

Ohjelmistotuotantoprosessissa on ensiksi suoritettava kohdealueen analyysi. Analyysi perustuu yleensä johonkin malliin eli metodiin. Ideaalinen lähestymistapa analyysissä on, että suunnittelijat saisivat siitä apua ohjelmistokehitysprosessin jokaiseen vaiheeseen ja voisivat valita sopivia komponentteja uudelleenkäytettäväksi [For02 s. 25].

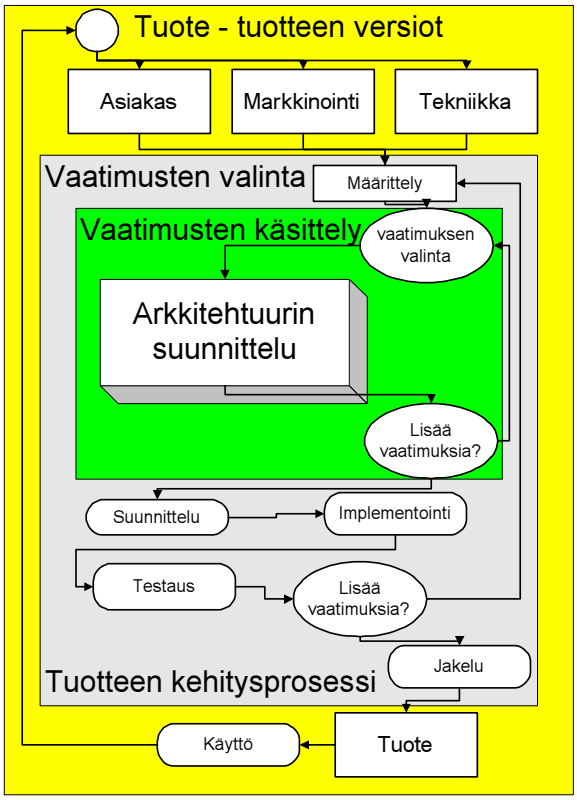
Valmiin ohjelmiston arkkitehtuuri on monen tekijän summa. Alustava suunnitelma on harvoin riittävän hyvä, joten sitä joudutaan parantamaan. Lopulta on aikaansaatu riittävän hyvä osapuolia tyydyttävä arkkitehtuuri, jonka pohjalta suunnittelua voidaan jatkaa.

4.5.2 Arkkitehtuurin suunnittelu

Boschin mukaan arkkitehtuurin suunnittelu koostuu kolmesta tehtävästä: toimintopohjaisen arkkitehtuurin suunnittelusta (functionality-based architectural design), arkkitehtuurin arvioimisesta (assessing software architecture) ja arkkitehtuurin muokkaamisesta (architecture transformation) [Bos00 s. 32].

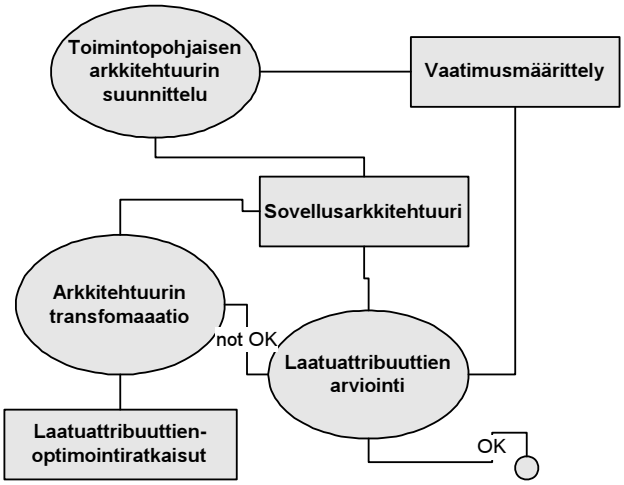
Arkkitehtuurisuunnittelussa lähtökohtana on *top-down* –ajattelu. Tämä tarkoittaa sitä, että kohdealuetta tarkastellaan aluksi kokonaisuutena. Järjestelmä jaetaan pienempiin osiin ja osat analysoidaan ja jaetaan pienempiin osakokonaisuuksiin. Lopputuloksena on eri karkeusasteen elementtien muodostama kuvaus ohjelmistosta.

Boschin mukaan arkkitehtuurin huomioiva ohjelmiston suunnitteluprosessi koostuu neljästä sisäkkäisestä vaiheesta, joissa kussakin kehitys tapahtuu iteroiden. Kuvassa 32 esitetään suunnitteluprosessi ja kuvassa 33 sisin vaihe [Bos00 luku2 s. 23]. Kahta sisintä vaihetta sanotaan *laatuattribuutti suuntautuneeksi ohjelmistosuunnittelumenetelmäksi (quality attribute-oriented software architecture design method)*.



Kuva 32. Arkkitehtuurin suunnitteluprosessin konteksti [Bos00 s. 24]

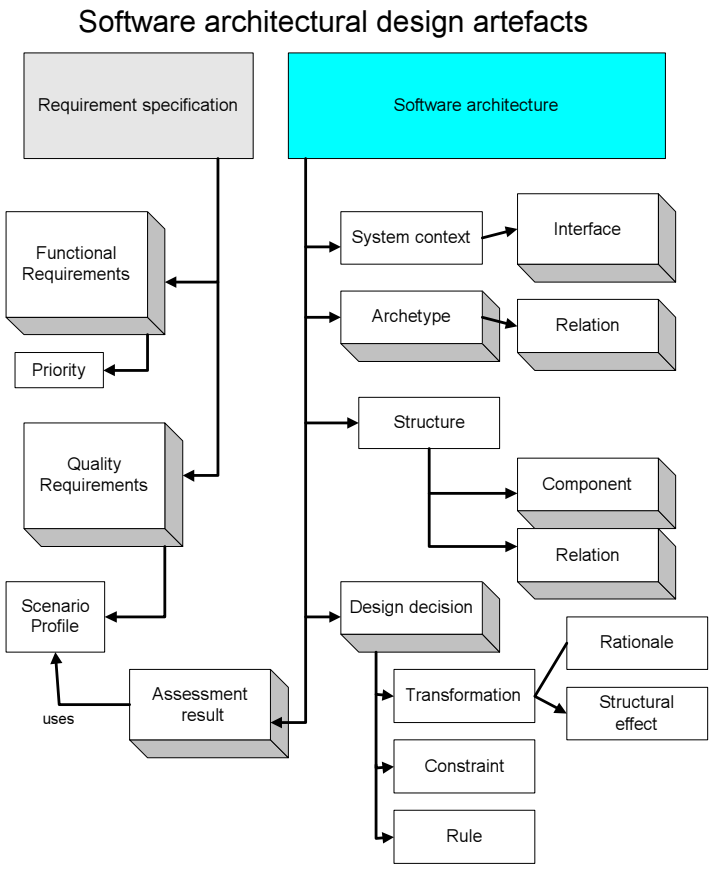
Uloin vaihe kuvaa tuotetta: eri versioita, asiakkaita, markkinointiosastoa ja suunnittelijoita. Uloin vaihe luo tuotteelle vaatimukset, jotka tuotteella ovat olemassa (Kuva 33). Uloin vaihe kuvaa tuotteen elinkaarta asiakkaan kannalta.



Kuva 33. Arkkitehtuurin suunnittelu [Bos00 s. 32]

Seuraavaksi vaiheessa kaksi laaditaan vaatimusmäärittely, josta voidaan valita vaatimuksia jatkokäsittelyyn. Tässä vaiheessa siis määritellään vaatimukset ja sisemmät prosessit laativat sen pohjalta soveliaan arkkitehtuurin. Tämän jälkeen suunnitellaan tuote, implementoidaan ja testataan. Tuloksena on tuote. Katsotaan onko tuote valmis jakelua varten, ja jos ei ole, palataan vaatimusmäärittelyyn, jota täydennetään. Vaatimusmäärittelyn muutosten perusteella parannetaan arkkitehtuuria. Tätä jatketaan kunnes tuote on valmis jakeluun ja markkinoille (Kuva 32). Tämä vaihe kuvaa tuotteen suunnittelu ja valmistusprosessia.

Vaatimusmäärittelystä otetaan vaatimuksia kolmatta vaihetta varten. Vaatimukset valitaan siten, että ensiksi valitaan määräävin vaatimus ja jatketaan kunnes vaatimukset on käyty lävitse. Tämän vaiheen valittu vaatimus on syötteenä seuraavalle vaiheelle, jossa suunnitellaan ja muokataan arkkitehtuuri (Kuva 32 ja Kuva 33).



Kuva 34. Arkkitehtuurin suunnittelutekijät [Bos00 s. 26]

Neljännessä vaiheessa suunnitellaan ensiksi *toiminnallinen arkkitehtuuri (functionality-based architectural design)* eli määritellään se arkkitehtuuri, joka parhaiten toteuttaa vaaditun toiminnallisuuden. Toiminnallista arkkitehtuuria määritellessä määritellään ensiksi *systeemin konteksti (system context)*. Kontektista määritellään rajapinnat ulkoisiin systeemeihin. Liitetään toiminnalliset ja laadulliset vaatimukset jokaiseen rajapintaan. Lisäksi, jos määritellään *tuotelinjaa (product line),* määritellään eri tuotteita varten tarvittava rajapintojen muunneltavuus (variablity). Tässä voidaan käyttää apuna jo luotua kohdealueen analyysia (domain analysis). Kontekstista etsitään *arkkityyppejä (archetypes)*. Arkkityypit ovat korkeamman abstraktiotason karkeita yleistyksiä, jotka voidaan löytää kyseisestä kontekstista. Kontekstista voi löytyä useita kandidaatteja arkkityypeiksi. Tällöin joudutaan valitsemaan kehityksen kannalta sopivimmat työskentelyn pohjaksi. Arkkityyppien joukko muodostaa karkean arkkitehtuurin. Jaetaan tämä arkkitehtuuri sopiviin komponentteihin. Jo tässä vaiheessa voidaan katsoa, voidaanko käyttää olemassa olevia komponentteja hyödyksi. Sanotaan, että yrityksellä on komponenttivarasto (repository) [HeS99]. Tällöin muodostuu alustava sovellusarkkitehtuuri (Kuva 33 ja 34). Komponentteja muodostettaessa tutkitaan, että kontekstikaavion rajapinnat voidaan kohdentaa komponentteihin. Komponentit tulee voida liittää johonkin kohdealuee-

seen. Kohdealueita on joko sovelluskohtaisia *kohdealueita (application domain*) tai *tietojenkäsittelytieteen mukaisia kohdealueita (computer science domain)*. Jälkimmäisillä tarkoitetaan vaikkapa käyttöjärjestelmää tai tiedostojärjestelmää. On hyödyllistä tutkia järjestelmää siten, että jakaa sen toiminnallisuutta eri abstraktiotasoille. On tarpeellista tutkia myös, olisiko kohdealueessa (domain entities) jotain sellaista, jonka mukaisesti on tarpeellista suorittaa osiinjakoa. Ja lopuksi, jos tunnistettuja arkkityyppejä on useissa paikoissa järjestelmää, voi olla järkevää, että tälläisestä *arkkityypista (archetype instantiations)* tehdään oma komponenttinsa [Bos00 s. 62-63]. Tässä vaiheessa tarkastelu on vielä kohdealue-spesifinen eikä ratkaisu-spesifinen. Komponenttien välisten kytkösten lukumäärä tulisi pitää mahdollisimman vähäisenä *(low coupling principle)*. Tämän jälkeen muodostettu karkea arkkitehtuuri, jossa systeemi on jaettu karkealla tasolla komponentteihin, jaetaan rekursiivisesti osakomponentteihin *(describing system instantiations*) [Bos00 s. 66-67].

Arkkitehtuurin arvioinnin tarkoituksena on tutkia, kuinka hyvin toiminnallisten vaatimusten pohjalta rakennettu arkkitehtuuri vastaa laadullisia vaatimuksia. Jos arkkitehtuuri ei täytä vaatimuksia, on arkkitehtuuria parannettava eli muokattava, jotta se täyttäisi asetetut vaatimukset. Arviointiin voidaan käyttää erilaisia tekniikoita: laadullinen arviointi (qualitative assessment), määrällinen arviointi (quantitative assessment) sekä teoreettisen maksimin arviointi (assessment of theoretical maximum) [Bos00 s. 80]. Laadullinen arviointi tarkoittaa sitä, että verrataan kahta arkkitehtuurimallia valittuun laatuattribuuttiin ja vertailun tuloksena saadaan tieto, kumpi täyttää paremmin vaatimukset. Määrällinen arviointi tarkoittaa sitä, että laatuattribuuteille on annettu määrällisiä vaatimuksia ja verrataan, mikä arkkitehtuuri täyttää parhaiten vaatimukset. Ongelmana tässä kuten muissakin arviointimenetelmissä on se, että laadulliset ominaisuudet ovat ristiriidassa keskenään (suorituskyky, joustavuus jne.). Jos laatutekijöiden keskinäinen tärkeys on arvioitu, on vertailu helpompaa. Tällöin voidaan vertailussa painottaa tärkeimpiä laatutekijöitä. Teoreettisen maksimin tai minimin arviointi tarkoittaa sitä, että arvioidaan jonkin laatutekijän maksimi- tai minimiarvot ja verrataan näitä arvoja tarkastelun kohteena olevan arkkitehtuurin arvoon.

Arkkitehtuurin arvioinnin ongelmia varsinkin analyysivaiheessa ovat järjestelmän käyttäytymisen hahmottaminen toiminnan eri tilanteissa. *Profilointi (profile)* –tekniikka on yleisesti käytetty tekniikka niissä tapauksissa, jossa jotakin ominaisuutta halutaan tutkia, silloin kun profiili muodostuu *joukosta skenaarioita (set of scenarios)*. Ylläpidettävyyt-

tä voidaan arvioida esimerkiksi muutosprofiilin *(change profile)* kautta ja sitkeyttä virhesyötteiden kautta. Muita profiileita ovat esimerkiksi joukko käyttäjäskenaarioita eli *käyttäjäprofiili (usage profile)*, joka tutkii käyttäjän syötteitä tai *hazardi-skenaariot (hazard scenarios)*, jolla taas tutkitaan turvallisuutta (safety). Arviointi on tällöin *skenaario-pohjaista arviointia (scenario-based assessment)*. Arkkitehtuuria arvioidaan ensiksi *vaikuttavuusanalyysin (impact analysis)* avulla, jossa tutkitaan skenaarioprofiilin vaikutusta valittuun arkkitehtuuriin. Analyysin tuloksena saadaan tuloksena *dataa*, jonka perusteella saadaan aineistoa laatuattribuuttien arvioimiseksi [Bos00 s.82].

Toinen mahdollisuus on *käyttää simulointia (simulation-based assessment)*, jossa mallinnetaan järjestelmää ja tutkitaan sen käyttäytymistä. Tässä voidaan käyttää *protoilua (prototype)*. Joissain tapauksissa voidaan laatutekijöitä arvioida *matemaattisen mallin (mathematical model-based assessment)* avulla esimerkkinä jotkin reaaliaikaiset järjestelmät, joissa tutkitaan syötteen, käsittelyn ja vastauksen reaaliaikaisuutta. Usein *kokemuspohjainen (experience-based assessment)* arviointi on tehokasta varsinkin silloin, kun kohdealueelle on jo tehty vastaavanlaisia sovelluksia [Bos00 s.79].

Alustava arkkitehtuuri ei kelpaa sellaisenaan, vaan sitä on parannettava, jotta se täyttää myös laadulliset vaatimukset. Laatuvaatimusten huomioimista arkkitehtuurisuunnittelussa kutsutaan *arkkitehtuurin muuttamiseksi (architecture transformation)* eli *transformaatioksi.* Transformaation avulla voidaan arkkitehtuuria parantaa [Bos00 s. 109].

4.5.3 Yhteenveto

Arkkitehtuurin suunnitteluvaihe koostuu kolmesta osasta: toiminnallisen arkkitehtuurin suunnittelusta, arkkitehtuurin arvioimisesta ja arkkitehtuurin transformaatiosta. Toiminnallisen arkkitehtuurin suunnittelussa kohdealueen analyysin perusteella löydetään sopivat komponentit, joiden pohjalta muodostetaan alustava sovellusarkkitehtuuri. Suunnitteluprosessin tarkoituksena on löytää kohdealueesta kaikki merkittävät arkkitehtuuriin vaikuttavat tekijät. Muodostaa niiden pohjalta yleiskuva arkkitehtuurista. Muokata arkkitehtuuria, kunnes sen katsotaan olevan riittävän hyvä, jotta se täyttää asiakkaan vaatimukset. Parasta tässä mallissa on iteratiivisuus ja sen pohdiskeleva luonne. Tämä menetelmä on parhaimmillaan silloin, kun toimitaan uudella ja oudolla kohdealueella tai kun ollaan rakentamassa tuotelinjaa.

Jos kohdealue ja sovellus on läpikotaisin tuttu, on luultavaa, että kustannus- ja aikataulusyistä, tämän tyyppistä lähestymistapaa ei käytetä, vaan mennään suoraan detaljeihin ja hiotaan niitä. Ohjelmistoista tehdään niiden elinkaaren aikana uusia versioita, virheitä korjataan, uusia ominaisuuksia lisätään jne. Muutokset tehdään vanhaan koodiin. Muutokset rikkovat vähitellen ohjelman rakenteen. Ohjelmisto rappeutuu. Jossain vaiheessa tullaan tilanteeseen, jolloin on edullisempaa tehdä kokonaan uusi ohjelma, kuin päivittää vanhaa. Tällöin kannattaa kohdealueesta tehdä uusi analyysi ja katsoa, mitä muutoksia vaatimuksiin on syytä tehdä. Uutta rakennettaessa tulisi prosessin lähtökohtana olla kohdealueen analyysi ja arkkitehtuurisuunnittelu tulisi tehdä uuden analyysin pohjalta.

4.6 Arkkitehtuurin parantaminen

4.6.1 Yleistä

Jos arkkitehtuuri ei täytä laadun osalta vaatimuksia, on sitä parannettava. Tätä arkkitehtuurin muokkausta nimitetään *arkkitehtuurin transformaatioksi*. Siinä lähdetään olevasta sovellusarkkitehtuurista liikkeelle ja muokataan arkkitehtuuria ja lopputuloksena on uusi arkkitehtuuri. Arkkitehtuuria parannettaessa on otettava kaikki se tietämys, joka organisaatiolla niistä keinoista, joita tarvitaan kohdealueen sovelluksen arkkitehtuurin muokkaamiseksi. Muokkaamisessa apuna käytetään organisaation kokemusta sekä omia itse laadittuja mallisysteemejä, malleja tai yleisiä mallisysteemejä. Abstraktion eri tasoilla käytetään arkkitehtuurityylejä, malleja ja suunnittelumalleja muokkaamisen välineenä. Muokkaamisen aikana muutetaan laatutekijät toiminnallisuudeksi ja jaetaan asetetut laatuvaatimukset komponenteille.

4.6.2 Arkkitehtuurin parantaminen

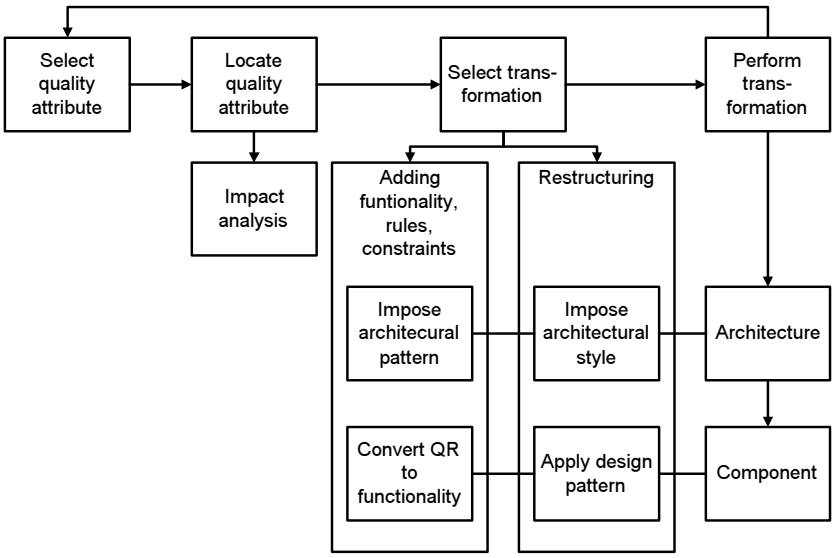
Arkkitehtuurin muokkaaminen kohdistuu sekä yksittäisiin komponentteihin, että koko arkkitehtuuriin. Arkkitehtuuria muokattaessa toiminnallisuutta voidaan määritellä tai lisätä valitsemalla sopiva arkkitehtuurimalli. Näin voidaan esimerkiksi hajauttaa toimintaa Broker-arkkitehtuurimallin avulla. Arkkitehtuurin parantamisen eräs tärkeimpiä tehtäviä on pystyä määrittelemään kullekin laatutekijälle sen vaikutuskohde eli ne komponentit, jotka rakentavat kysein piirteen. Esimerkiksi määriteltäessä suorituskykyä on pystyttävä löytämään ne komponentit ja niiden väliset suhteet, jotka muodostavat prosessissa pullonkaulat eli kuormittuvat eniten ohjelmistoa suoritettaessa [Bos00 s. 114]. Kuvassa 35 on kuvattu transformaatioprosessi.

Muokkaamisprosessissa tunnistetaan ensimmäisenä ne laatutekijät, joita valittu arkkitehtuuri ei täytä. Laatuattribuutit on priorisoitu ja siten saadaan käsittelyjärjestys. Laatuattribuuteista yleensä neljä tai viisi on muita tärkeämpiä ja ne ovat järjestelmälle erittäin tärkeitä. Listataan myös ne attribuutit, joiden vaatimukset järjestelmä täyttää [Bos00 s. 112].

Toisena vaiheena pyritään laatuvaatimukset kohdistamaan tietyille komponenteille tai tiettyyn kohtaan arkkitehtuuria. Komponentit toteuttavat toiminnallisuuden. Myös laatutekijät on kyettävä kohdentamaan oikeisiin komponentteihin. Kohdentamisen avulla voidaan löytää järjestelmän pullonkauloja (bottle-neck). Arkkitehtuurin arvioinnin avulla on saatu osviittaa siitä, mikä osatekijä on määräävä kullekin laatutekijälle. Usein skenaarioissa on havaittu, että yksi ja sama komponentti on mukana useassa skenaariossa. Jos ohjelmistovaatimuksiin tehdään muutoksia, on tämä komponentti erityisen herkkä kaikille muutoksille [Bos00 s. 113].

Kolmantena vaiheena valitaan se muokkaamisen kohde, joka ratkaisee arkkitehtuurin ongelmapisteet (problem spots). Tavallisesti tarvitaan useita muokkaamisoperaatioita. Hankaluutena tässä on, se että eri laatuvaatimukset ovat ristiriidassa keskenään. Yhden laatutekijän huomioiminen voi huonontaa toista tekijää. Pyritään valitsemaan sellainen transformaatio, joka ei vaikuta haitallisesti muihin ominaisuuksiin [Bos00 s. 115]. Transformaatio voi esimerkiksi lisätä toiminnallisuutta arkkitehtuuriin tai yksittäisiin komponentteihin (added functionality, rules or constraints) tai muotoilla arkkitehtuuria tai komponentteja (restructuring).

Lopuksi toteutetaan transformaatio. Lisättävä toiminnallisuus voi vaikuttaa sovellettavaan arkkitehtuurimalliin. Voi olla, että sitä joudutaan muuttamaan. Komponentit toteuttavat toiminnallisuuden. Laatuattribuutit muutetaan toiminnallisuudeksi siten, että pyritään kohdistamaan vaatimus niihin komponentteihin, joiden vaikuttavuusanalyysin perusteella uskotaan osallistuvan k.o. laatutekijän toteuttamiseen. Jos havaitaan, että valittu arkkitehtuuri tai jokin sen osa ei täytä vaatimuksia, joudutaan arkkitehtuuria muotoilemaan (restructuring). Tällöin voidaan joutua määrittelemään uudelleen käytettävä arkkitehtuurityyli. Uusi tyyli voi vaikuttaa valittuihin arkkitehtuurimalleihin. Mallit koostuvat komponenteista. Komponenttien ominaisuuksia voidaan muokata suunnittelumallien avulla. Arkkitehtuurin muokkausvaiheet on syytä dokumentoida, jotta jäljitettävyys säilyy ja voidaan palata tarvittaessa aikaisempiin versioihin [Bos00 s. 112-116].



Kuva 35. Transformaatio ja taksonomia (taxonomy of transformation process) [Bos00 s. 114 mukaellen]

4.6.3 Yhteenveto

Arkkitehtuurin muokkaamisen avulla voidaan toteuttaa ohjelmiston laatuvaatimukset. Koska muokkaaminen perustuu hyväksi havaittujen mallien hyödyntämiseen, parantaa mallien soveltaminen arkkitehtuuria. Muokkaamisoperaatio kohdentaa vaatimukset komponenteille. Koko ohjelmiston muokkaaminen vaatii useita muokkaamisoperaatioita, koska ohjelmisto on monimutkainen ja laatuvaatimuksia on paljon. Muokkaamisen avulla parhaassa tapauksessa ohjelmiston laatua voidaan parantaa, mutta pahimmassa tapauksessa jonkin ominaisuuden toteuttaminen saa aikaan sen, että jokin toinen ominaisuus ei toteudu toivotulla tavalla. Esimerkiksi, jos järjestelmälle halutaan erittäin hyvä suorituskyky, voi sen ylläpidettävyys, käytettävyys tai joustavuus olla todella heikko. Toteutuva järjestelmä on siis monen tekijän kompromissi.

4.7 Tuotelinja-arkkitehtuuri

4.7.1 Yleistä

Tehokas ohjelmistotuotanto pyrkii lisäämään uudelleenkäyttöä aina, kun se on mahdollista. Bosch luokittelee arkkitehtuurin käytön kolmeen kategoriaan: *räätälöidyt järjestelmät*, *tuotelinja-arkkitehtuurit* (*PLA, product line architecture)* ja *komponenttimarkkinoiden vakioratkaisut* [Bos00 s. 162].

Räätälöidyt järjestelmät toteutetaan ohjelmistoprojekteina. Projektointi järjestetään projektiryhminä ja projektin toteutus vaihtelee projekteittain. Räätälöidyissä projekteissa uudelleenkäyttö on hankalinta, koska projektit voivat olla aina erilaisia, projektien läpimenoajat ovat nykyisin todella lyhyitä. Toisaalta jos uudelleenkäyttöä voidaan lisätä projektissa, on sillä merkittävä vaikutus projektin talouteen ja tulosten laatuun.

Ohjelmistoilla voi olla samankaltainen rakenne ja toiminnallisuus. Tällöin ohjelmat muodostavat ohjelmistoperheen (software product line family). Tämän perheen yhteistä arkkitehtuuria kutsutaan tuotelinja-arkkitehtuuriksi (PLA, product line architecture). Tuotelinja-arkkitehtuurin eduiksi mainitaan lisääntynyt luotettavuus, projektien ennustettavuus, lyhentynyt tuotantoaika, selkeämpi ohjelmistotuotanto, parempi tuottavuus, parempi riskien hallinta sekä prototyyppien helpompi käyttö [Kos00 s. 200].

Kolmantena kategoriana oleva komponenttimarkkinoiden vakioratkaisut liittyvät kohdealueen arkkitehtuurin standartointiin siten, että tällä alueella voidaan käyttää sopimuksen mukaisia komponentteja, komponenttien rajapintoja ja riippuvuuksia komponenttien välillä. Standardointi on komponenttimarkkinoiden syntymisen edellytys [Bos00 s. 163].

4.7.2 Tuotelinja-arkkitehtuuri

Tuotelinja-arkkitehtuurinäkökulma voidaan jakaa kolmeen kategoriaan. Arkkitehtuuri, komponentti ja systeemi muodostavat ensimmäisen kategorian. Tällöin arkkitehtuuri suunnitellaan siten, että se rakentuu tuoteperheen ympärille ja erittelee ne elementit, jotka ovat yhteisiä kaikille tuotelinjassa ja ne, jotka ovat kullekin tuotteelle yksilöllisiä. Arkkitehtuuriin liittyvät tuotelinjan yhteiset komponentit ovat kooltaan melko suuria, jotta tuotelinja voitaisiin muodostaa järkevällä tavalla *(component system engineering)*. Lopuksi on tarkasteltava muodostettavaa tuotelinja-arkkitehtuuria ja systeemiarkkitehtuuria. Tarkoituksena on sopeuttaa komponentit ja komponenttien väliset suhteet systeemiarkkitehtuuriin *(application engineering)* [Bos00 s. 184].

Toinen ulottuvuus koostuu liiketoiminnasta, organisaatiosta, prosessista ja teknologiasta. Nämä ulottuvuudet pyrkivät lisäämään komponenttien uudelleenkäyttöä. Teknologiset päätökset uudelleenkäytön lisäämiseksi edellyttävät mm. oliopohjaisten kehysten käyttöä, tekniikoiden ylläpidettävyyden arviointia sekä laatuattribuuttien hallintaa.

Kolmas ulottuvuus muodostuu kohdealueen elinkaaren kattavien tehtävien hallinnasta organisaatiossa. Tähän sisältyvät kehittäminen, toimittaminen ja evoluutio. *Kehittäminen (development)* kattaa koko tuotelinja-arkkitehtuurin elinkaaren. Tähän vaiheeseen liittyvät analyysit, arkkitehtuurin suunnitteleminen ja arvioiminen. *Toimitusnäkökohta (deployment)* ottaa kantaa siihen, että tuotelinja-arkkitehtuuri on mukautettava siten, että syntyy jakelukelpoisia tuotteita. Tuotelinja-arkkitehtuuri ja uudelleenkäytettävät komponentit sijoitetaan kehityksen aikana tuotteen arkkitehtuuriin. Tuotelinja-arkkitehtuurin jakelussa on kyse siitä, että arkkitehtuuri konfiguroidaan tuotteen vaatimusten mukaiseksi. Tämän seurauksena muodostuu tuotteelle sen oma arkkitehtuuri. Mukautettuun tuotteen arkkitehtuuriin liitetään tuotelinja-arkkitehtuurin mukaiset komponentit, joihin lisätään tuotespesifiset konfiguraatiot sekä rakennetaan tuotespesifiset komponentit ja integroidaan elementit yhteen. Lopuksi pakataan komponentti jakelupakkaukseen ja tuote on valmis julkaistavaksi. Tuotteet kehittyvät ja muuttuvat. *Evoluutio (evolution)* on otettava huomioon tuotelinjassa. Tuotteen kehitystarpeet aiheuttavat muutospaineita komponentteihin. Tällöin joudutaan tutkimaan sopiiko muutos tuotelinjaan tai pitäisikö muuttuva komponentti irrottaa tuotelinjasta vai olisiko tuotelinjaa muutettava siten, että evoluutio on hallinnassa [Bos00 s. 166].

4.7.3 Yhteenveto

Tuotelinjan rakentamisessa haasteena on yhteisten tuotelinjaan sopivien elementtien ja toteutustapojen löytäminen. Tuotelinjan arkkitehtuuri tulisi tehdä niin, että syntyvät elementit ovat riittävän yleisiä, jotta niitä voidaan erikoistaa tuotteita rakennettaessa. Variaatiopisteet komponenteille on suunniteltava niin, että tuotteiden rakentaminen onnistuu mahdollisimman pitkälle varioimalla komponentteja. Mikäli tuotelinjassa on löydettävissä yhteinen runko eli yleisiä ominaisuuksia, jotka ovat samoja kaikille, kannattaa rakentaa komponenttikehys. Kehysten avulla voidaan komponenttisysteemiä mukauttaa vastaamaan kunkin tuotteen vaatimuksia.

Tuotelinjaa rakennettaessa on huomioitava tuotelinjan elinkaari liiketoiminnan analyysistä yksittäisten tuotteiden ylläpitoon ja kehitykseen saakka. Haasteina tuotelinjan rakentamisessa ja ylläpidossa ovat tuotteiden kehittämistarpeet ja rakennetun tuotelinja kyky mukautua uusiin vaatimuksiin.

4.8 Arkkitehtuurin dokumentointi

4.8.1 Yleistä

Arkkitehtuurin dokumentaatio alkaa projektin alkaessa ja loppuu joko projektin luovutuksen yhteydessä tai kun tuote saavuttaa elinkaarensa pään. Dokumentaation tehtävänä on rekisteröidä tehdyt luonnokset, eri suunnitteluratkaisut ja vaiheet asianosaisia varten. Dokumentaation tulee olla jäljitettävissä. Luonnosten ja eri työvaiheiden dokumentointi on tärkeää siksi, että voidaan tarvittaessa palata aikaisempiin versioihin ja luonnoksiin. Tämä on tärkeää siitäkin syystä, että harhapoluille eksyttäessä, voidaan palata takaisin ja etsiä uusia polkuja. Sanana arkkitehtuuri on varsin onnistunut. Analogia rakennusten arkkitehtuuriin on suoraviivainen. Rakennusarkkitehtuuri kuvaa pääpiirustusten ja työselityksen avulla rakennettavan rakennuksen yleispiirteisesti ja antaa yleiskuvan rakennuksesta. Suunnitelmasta voidaan löytää erilaisia tyylejä ja malleja aivan kuten ohjelmistoarkkitehtuureista. Rakennuksen arkkitehtuurisuunnitelmat tarvitsevat työselityksen, jossa tulkitaan ja annetaan tarpeellista informaatiota suunnitelmien ymmärtämiseksi ja tulkitsemiseksi.

Ohjelmistoarkkitehtuurin kuvauksen tulee kuvata ohjelmistoelementit eri näkökulmista ja sisällyttää dokumentaatioon riittävästi selittävää informaatiota. Hyvän dokumentaation tarkoituksena on olla opetuksen välineenä, kommunikoinnin välineenä, neuvottelun välineenä ja systeemianalyysin pohja. Suorituskykymittaajille arkkitehtuuri on malli, jonka mukaan arvioida ja simuloida järjestelmää. Arkkitehtuurin avulla arvioidaan myös laatuattribuutteja.

4.8.2 Näkymät dokumentaatioon ja dokumentoitavat kohteet

Arkkitehti valitsee dokumentoitavat näkymät. Dokumentaatio voi muodostua yhdestä tai useammasta dokumentista. Jokaisen dokumentin tulisi sisältää ainakin päivämäärä ja status (draft, baseline, version number jne), organisaation nimi, muutoshistoria ja yhteenveto.

Dokumentaatio, jota sovelletaan kaikkiin näkymiin, sisältää johdanto-osan koko dokumenttipakettia varten, jossa on lukijan opas, jonka avulla asianosaiset löytävät halutun informaation nopeasti. Johdannossa on kuvaus siitä, kuinka eri näkymät ovat suhteessa toisiinsa ja systeemin kokonaiskuvaus sekä rajoitukset ja perustelut koskien koko arkki-

tehtuuria. Edelleen siinä on vaadittava informaatio koko dokumentin ylläpitämiseksi [CleF02].

Näkymän vakiodokumentointi koostuu seitsemästä osasta [CleF02]. Esitysdokumentti (primary presentation) esittää elementit ja niiden väliset suhteet tavallisesti graafisena esityksenä. Siihen kuuluu elementtiluettelo (element catalog), kuvaus elementeistä ja niiden suhteista, ominaisuuksista, rajapinnoista ja käyttäytymisestä. Lisäksi tarvitaan yhteyskaavio (context diagram), joka esittää yhteydet muihin systeemeihin, muunneltavuus opas (variability guide), jossa selitetään ne (built-time) rakennusaikaiset mekanismit, jotka ovat saatavilla arkkitehtuurin räätälöintiä varten. Siinä on oltava selvitys arkkitehtuurin taustasta (architectural background) ja perustelut (rationale), miksi valittua arkkitehtuuria on käytetty. Muina dokumentteina arkkitehtuuria kuvattaessa ovat analyysin tulokset (analysis results) esimerkiksi suorituskykyanalyysi, turvallisuusanalyysi ja luettelo modifiointia varten sekä arkkitehtuurin suunnittelun kuluessa tehdyt oletamukset (assymptions). Muuta informaatiota ovat mahdolliset näkymään liittyvät muut dokumentit (related view packets).

4.8.3 Hyvän dokumentaation tunnusmerkit

Dokumentoinnin ainoa arvon mittari on sen käyttöarvo. Alla on lueteltu hyvän ja huonon dokumentaation piirteitä. Hyvälle dokumentaatiolle on tunnusomaista selkeys, keveys ja kyky dokumentoida käyttäjä huomioiden.

Huonon dokumentaation piirteitä [SoW98 s. 189]:

- Dokumentaatiota ei ole tehty lainkaan.
- Dokumentaatio ei sisällä lainkaan koodia. Dokumentaatio tehdään yhdessä koodauksen kanssa. Jos dokumentaatio tehdään ennen ohjelmointia, se sisältää ainoastaan käyttäjän vaatimukset eikä lainkaan ohjelmoinnin aikana esiin tulleita piirteitä.
- Dokumentaatio on tehty pelkästään käyttäen kaavioita ja piirroksia. Kaaviot vaativat aina lisäinformaatiota, selityksiä ja dokumentteja.
- Seinätaulutekniikka väärin käytettynä siten, että seinät täytetään suurilla piirroksilla, joissa on satoja pieniä laatikoita, voi olla vaikuttavaa, mutta enemminkin sotkee kuin selkeyttää hanketta.

- Suuri, paksu, muodollinen dokumentti, joka on tehty mallin mukaan täyttämällä malliin projektin tiedot. Mallit ovat hyviä, mutta dokumentaatio tulisi mukautua kohteeseensa niin, että siitä löytää haluamansa piirteet.

Hyvän dokumentaation tunnuspiirteitä ovat [SoW98 s. 189]:

- Dokumentilla on selkeä rakenne.
- Dokumentti on sopivasti sekoitettuna formaaleja ja epäformaaleja kuvaustapoja. Jokaisessa kappaleessa tulisi olla selkokielinen kuvaus ja muodollisempi osa.
- Dokumentissa on selkeä sanasto.

Hyvän dokumentaation lähtökohta on, että abstraktiotaso valitaan sopivasti [SoW98 s. 190]:

- Yhdistetään useita olioita yhdeksi.
- Käsitellään useita toimintoja (actions) yhtenä toimintojonona (action flow).
- Unohdetaan kuinka toiminnot on alustettu.
- Mallinnetaan kaikki käyttökelpoiset konseptit jopa abstraktit olioina. Spesifikaatiossa esiintyvien tyyppien ei tarvitse esiintyä implementaatiossa.
- Luonnehditaan toimintoja niiden päämäärien mukaan, ei niinkään sen mukaan kuinka ne toimivat

4.8.4 Dokumentaatio ja asianosaisten kiinnostuksen kohteet

Kiinnostuksen kohteet ovat esimerkiksi seuraavat: *Projektipäällikkö (project manager)* on kiinnostunut aikataulusta, resurssoitavista tehtävistä ja ehkä mahdollisuudesta julkaista systeemistä jokin osajoukko kaupallisista syistä. *Projektin jäsen (member of the development team)* tarvitsee omaan tehtäväänsä liittyvät tiedot, yleistiedot sekä marssijärjestyksen työn suorittamiseksi. *Testaajat ja integraattorit (testers and integrators)* tarvitsevat tiedot testattavista tai integroitavista osista. Elementin yksikkötestaaja tahtoo saada saman informaation kuin elementin kehittäjäkin, jotta testaaja tietäisi, kuinka elementti käyttäytyy. Mustalaatikko-testaaja (black box testing) tahtoo saada elementin rajapintamäärittelyt. *Muiden järjestelmien suunnittelijat (designers of other systems)* tarvitsevat arkkitehdilta joukon tarjottuja ja vaadittuja rajapintamäärittelyjä sekä ohjeet näiden operaatioiden soveltamiseksi. *Ylläpitäjät (maintainers)* käyttävät arkkitehtuuria ylläpidon aloituspisteenä ja tarvitsevat tietoa siitä, mihin alueisiin kukin muutos vaikuttaa. Ylläpitäjille on syytä toimittaa sama dokumentaatio kuin kehittäjillekin (develo-

pers). *Asiakkaat (customers)* ovat kiinnostuneita siitä, toteuttaako järjestelmä toiminnalliset ja laadulliset vaatimukset. *Loppukäyttäjät (end users)* eivät tarvitse arkkitehtuuria työssään, mutta he ovat kiinnostuneita siitä, mitä milloinkin tapahtuu ja kuinka he voivat käyttää ohjelmaa tehokkaammin tutkimalla arkkitehtuuria. *Analysoijat (analysts)* haluavat tietää sen kuinka hyvin systeemi täyttää laatuvaatimukset. *Uusi asianosainen* (*new stakeholder)* tarvitsee ainakin yhteyskaavion, tiedot arkkitehtuurien rajoituksista, yleiskatsauksen ja perustason tietopaketin systeemistä. *Tuleville arkkitehdeille* (*future architects)* on syytä jakaa kaikki tieto [CleF02 s. 290].

4.8.5 Yhteenveto

Ohjelmistoprojektiin osallistuu useita sidosryhmiä. Informaatiolla on selvä raja, joka kulkee yrityksen ja ulkomaailman välillä. Ulkopuolisille sidosryhmille luovutettava dokumentaatio on huomattavasti rajoitetumpaa ja keskittyy yleisluontoisen informaation antamiseen ja tarkoin rajattuun yhteiseen tehtävään, kuin yrityksen projektiryhmälle annettava informaatio. Informointi riippuu siis suoritettavasta tehtävästä. Sovitaan, että arkkitehti vastaa dokumentoinnista. Tällöin arkkitehdin vastuulla on laatia jokaiselle dokumentille jakeluohje. Jakeluohjeen avulla jokainen saa hänelle kuuluvan informaatiopaketin hankkeesta. Tavoitteena ja haasteena dokumentoinnissa on tuottaa riittävän laadukasta informaatiota, oikea-aikaisesti, oikeille henkilöille. Dokumentointi ja dokumentoinnin jakelu liittyvät läheisesti projektinhallintaan.

4.9 Yhteenveto arkkitehtuureista

Yritysten tietojärjestelmissä on eri aikakausina ollut erilaisia trendejä. 80-luvulla alkaneen Microsoft-vetoisen PC-valtakauden tuloksena sovelluksia ja tietoja hajautettiin käyttäjien PC-laitteisiin ja työpöydille. Verkkokäyttöjärjestelmien avulla tietoja ja ohjelmia voitiin tallentaa resursseiksi palvelimille. Palvelimet toimivat joko tulostuspalvelimina tai tiedostopalvelimina. Tuloksena tästä oli tiedon yhtenäisyyden häviäminen ja ylläpidon tehottomuus. Tarvittiin keskittämistä. Perinteinen keskuskonemalli, jossa prosessit toteutettiin palvelimella ja jossa tieto ja resurssit tarjottiin palvelimen kautta käyttäjän laitteen ollessa ainoastaan terminaali, oli malli rakennettaessa uutta hajautettua ympäristöä. Tieto keskitettiin tietokannanhallintajärjestelmille (DBMS). Yhteinen tieto kopioitiin (replicate) tarvittaessa toisille palvelimille. Asiakas/palvelin-suhteessa asiakas-ohjelmassa oli aluksi runsaastikin toiminnallisuutta ja palvelinpuoli keskittyi tiedonhallintaan. Ylläpidon kannalta tilanne ei ollut tyydyttävä. Tehokkuutta toimintaan

saatiin siten, että asiakkaan laitteen toiminnallisuuden tuli olla minimissä tarjoten ainoastaan graafisen käyttöliittymän. Verkko-selaimesta on muodostunut universaali käyttöliittymä, joka tarjoaa sekä kehittäjälle että käyttäjälle standardin toimintaympäristön.

Verkossa hajautettuina olevien sovellusten yhteistoimintaa on harrastettu väliohjelmistojen avulla aina 1980-luvun lopulta saakka. Komponenttipohjaisissa liiketoimintasovelluksissa on alettu käyttää yhä useammin sovelluspalvelimia. Sovelluspalvelimet paitsi että toimivat väliohjelmistoina, tarjoavat komponenttisysteemeille peruspalvelut sekä lisäksi ajoalustan toimintaa varten. Tällöin muodostuvassa monikerros-arkkitehtuurissa tietokannanhallintajärjestelmä vastaa tiedosta, sovelluspalvelin toimii komponenttien ajoalustana, web-palvelin rakentaa asiakasta varten liittymän ja asiakas toimii omassa kerroksessa joko selaimen tai asiakassovelluksen avulla. Yritysohjelmistojen välinen automaattinen kommunikointi hoidetaan yhä useammin verkkopalveluiden avulla. Aiemmin kommunikaatio hoidettiin yleensä viestinvälitys-palvelinten avulla. Verkkopalvelimen avulla saavutetaan sijaintiriippumattomuus ja yhteinen kommunikointiprotokolla.

Sovelluskehitys oli liian kallista. 90-luvulla sovelluskehitystä haluttiin tehostaa ja ohjelmistoja alettiin koostaa uudelleenkäytettävistä komponenteista. Sinänsä ajatus ei ollut uusi, uudelleenkäyttöä on harrastettu laajasti aina, kun se on ollut mahdollista. Uutta oli se, että komponenttien käytön edistämiseksi luotiin määrittelyjä, joilla pyrittiin rakentamaan kaupalliset markkinat ja edistämään komponenttien käyttöä. Komponenttipohjaisuus on 2000-luvulla ohjelmistotuotannon päätrendi yhdessä hajautettujen järjestelmien ja oliosuuntautuneen ajattelun kanssa.

Komponenttien koostamisen ja komponenttien toiminnan esittämiseksi syntyi käsite komponenttiarkkitehtuuri. Arkkitehtuurin tavoitteena on eri näkökulmista lähtien kuvata suunniteltavan ohjelmiston rakennetta. Pian arkkitehtuurista löydettiin säännönmukaisuuksia eli malleja. 90-luvulla arkkitehtuureista luotiin eri käyttötarkoituksia varten eri rakeisuuden omaavia malleja, joita kutsutaan arkkitehtuurityyleiksi, -malleiksi ja suunnittelumalleiksi. Luodut mallit ovat vuorovaikutuksessa keskenään ja niistä muodostettiin mallisysteemejä. Arkkitehtuureita suunniteltaessa näitä malleja voitiin hyväksikäyttää ja lisätä luotavan ohjelmiston ymmärrettävyyttä. Hajautuksen, lisääntyneen vuorovaikutuksen ja uusien käyttäjävaatimusten takia ohjelmistojen monimutkaisuutta voidaan hallita paremmin arkkitehtuurikuvausten avulla. Arkkitehtuurikuvaukselle on tul-

lut myös informatiivisia tehtäviä. Kuvaus toimii päätöksenteon välineenä ja dokumentoi rakennettavan systeemin.

Ohjelmistoprojektin hankaluutena on se, että muodostuva tuote nähdään vasta, kun se voidaan ajaa lopullisessa ympäristössään. Ohjelmistoa tulee kyetä arvioimaan jollain tavalla. Arkkitehtuuri tarjoaa tähän välineen. Arkkitehtuuri suunnitellaan hankkeen alkuvaiheessa. Arkkitehtuuri rakennetaan mallien avulla tai niitä muokaten. Mallien toiminta tunnetaan. Arkkitehtuurin avulla voidaan arvioida jonkin ominaisuuden toteutumista ja suorittaa tarvittavia korjauksia halutun toiminnallisuuden saavuttamiseksi.

Käytännössä vaikeuksia tuottaa jonkin tietyn ominaisuuden kohdistaminen oikealle komponentille. Voi käydä niinkin, että puutteellisten tietojen takia joku laadullinen ominaisuus kohdistetaan väärälle komponentille. Tällöin havaitsematta jäänyt ongelma voi siirtyä jopa asiakkaalle menevään tuotteeseen. Toiminta on organisoitava niin, että väärät päätökset, tehdyt virheet ja kehitystä kaipaavat piirteet tulisivat ilmi. Ongelma liittyy projektinhallintaan ja organisaation kykyyn toimia ja arvioida toimintaansa. Toiminnan tulisi olla jollain lailla iteratiivista, jotta jo tehtyä työtä arvioitaisiin eri näkökulmista useaan otteeseen. Näin löydettäisiin kehittämistä edellyttävät osat ja piirteet

Ohjelmistohankkeen tavoitteet muodostuvat kaikkien sidosryhmien tarpeista. Tavoitteena on parempi tuote. Suunnittelun tulee kyetä tuottamaan laatua ja kilpailukykyä. Liiketoimintakomponenttien kehittämisessä tämä tarkoittaa kustannustietoista, laatuorientoitunutta ja uudelleenkäyttöön pyrkivää tapaa. Jos yrityksellä on mahdollisuuksia muodostaa tuotelinja, se kannattaa. Tuotelinjan avulla voidaan uudelleenkäyttöä lisätä. Tuotelinja ei sovellu pieniin yksittäistapauksiin, koska sen kehittäminen on kallista. Sen sijaan jos yrityksellä on useita tuotteita, joilla on samantyyppisiä ominaisuuksia, voidaan niistä etsiä sopivia toimintoja, joista voidaan rakentaa komponentteja tuotelinjaan.

5 Esimerkkitapaukset

5.1 Yleistä – tavoitteet, tutkimusmenetelmät

Esimerkkitapausten tavoitteena oli selvittää niitä arkkitehtuureissa olevia mahdollisuuksia tai ominaisuuksia, joita yritykset käyttävät pyrkiessään parantamaan tuotteidensa tai informaatiojärjestelmiensä kilpailukykyä. Koska lähtökohtana oli reaalimaailma, ei voida olettaa, että yritykset käyttäisivät täysimääräisesti kaikkia niitä mahdollisuuksia, joita ohjelmistoarkkitehtuurit tarjoavat. Usein jo hyvinkin yksinkertaisella oivalluksella ja käytännön kokemuksilla saadaan aikaan merkittäviä parannuksia.

Tutkimusmenetelmäksi on valittu haastattelututkimus, koska haastateltavien arkkitehtuurien tuntemuksesta ei etukäteen ollut tietoa. Haastattelun avulla pyrittiin hahmottamaan toimintaympäristö, siinä esiintyvät ongelmat ja ongelmien ratkaisutavat. Haastattelu suoritettiin vapaamuotoisen keskustelun avulla ja se kesti noin tunnin. Haastattelussa luodattiin haasteltavan järjestelmälle asettamia vaatimuksia, jotka luokiteltiin toiminnallisiin ja ei-toiminnallisiin vaatimuksiin.

Haastateltavilta kysyttiin ensiksi kohdealueeseen liittyvin kysymyksin konteksti eli tilannekuvaus haastateltavasta aiheesta. Sen jälkeen keskusteltiin järjestelmälle asetettavista toiminnallisista vaatimuksista. Toiminnallisten vaatimusten jälkeen siirryttiin haastattelun vaativimpaan osaan eli ei-toiminnallisiin (laadullisiin) vaatimuksiin. Tavoitteena oli selvittää, kuinka haastateltavat asettivat laatuvaatimuksia ja arvioivat niitä. Taustana tässä oli se, että arkkitehtuurin suunnittelun ja muokkaamisen avulla voidaan arvioida tulevan järjestelmän ominaisuuksia jo ennen kuin se on rakennettu.

Ensin kartoitettiin toiminnallisia vaatimuksia, joita olivat ohjelmiston tehtävät järjestelmässä. Oliko ohjelma tarkoitettu esimerkiksi eräajoon, loppukäyttäjän kyselyiden ja päivitysten palvelutehtäviin, toisten ohjelmien palvelutehtäviin sekä käyttöliittymäksi perinnejärjestelmiin. Merkittävää on myös, mitä ominaisuuksia ohjelmassa on oltava, mitä olisi hyvä olla tai mitä ominaisuutta ei tarvita. Mahdolliset optiot kiinnostavat myös. Systeemin asettamat vaatimukset kuten käyttöjärjestelmät, laitekokoonpanot, järjestelmäriippumattomuus tai mobiilikäyttöympristöt kirjattiin ylös.

Arkkitehtuurin määrittelyn selvittäminen on vaikeaa jopa mahdotonta johtuen siitä, että siihen liittyy liikesalaisuuksia, joiden perusteita ei haluta paljastaa ulkopuolisille. Yrityksen on syytä selvittää itselleen, millaisia mallisysteemejä se ensisijaisesti aikoo käyttää. Mielenkiintoista on myös tieto, löytyikö mallisysteemistä sovellusvalmiita malleja, kuinka valittuja malleja muokattiin sekä mitä arkkitehtuuri- tai suunnittelumalleja käytetään. Suunnitteluvaiheen arkkitehtoniset ratkaisut tulisi tehdä niin, etteivät ne sido yritystä mihinkään tiettyyn teknologiaan, vaan toteutusvaiheessa voidaan valita yritykselle parhaiten sopiva teknologinen ratkaisu.

Arkkitehtuurin parantaminen kartoitettiin niin, että katsottiin, mikä on nykytilanne ja mikä on tuleva tilanne. Jos arkkitehtuuria muutetaan, ja uusi arkkitehtuuri tyydyttää asianosaisia, voidaan sanoa, että arkkitehtuuri on parantunut.

Tutkittuja tapauksia oli kolme, joista kustakin on oma selvityksensä jäljempänä. Järjestelmät olivat kooltaan ja käyttötarkoituksiltaan suurestikin toisistaan poikkeavia. Toteutustavat ja problematiikka muistuttivat monilta osin toisiaan. Kaksi haastateltavista oli tietojärjestelmän käyttäjiä, jotka tekevät järjestelmän kehitystä ja muokkausta yhdessä toimittajan kanssa. Kolmas haastateltava oli omaa valmisohjelmistoaan markkinoiva yritys, joka suorittaa markkinoinnin suoramarkkinointina ja tekee itse tuotekehitystä yhdessä asiakkaidensa kanssa.

5.2 Esimerkki 1 – haastattelu

Ensimmäisenä haastattelun kohteena oli terveystoimen tietojärjestelmän uusimisprojekti. Haastattelusta on laadittu raportti (Liite 3), jossa on kartoitettu konteksti, ongelma ja ratkaisu. Haasteltava on loppukäyttäjä, jonka tavoitteena on hankkia ohjelmat valmiina. Ohjelmistoon tehdään kehittämistä lähinnä ylläpidon kautta. Ohjelmisto on toimintakriittinen ja koko organisaation toiminta nojaa siihen. Hankkeen toteuttamiseksi perustettiin projekti, joka kartoitti perusteellisesti hankkeen vaatimukset ja tutustui tarjolla oleviin toimittajaehdokkaisiin. Valittujen toimittajakandidaattien tuotteisiin tutustuttiin referenssien avulla ja sen jälkeen suoritettiin koekäyttö asiakasorganisaation omin laittein. Koekäytössä eri käyttäjäryhmät tutustuivat vaihtoehtoihin ja raportoivat havainnoistaan perusteellisesti laaditun kysymyslomakkeen avulla. Hankintaprosessissa sekä toiminnalliset että käytettävyyteen liittyvä tekijät punnittiin tarkkaan.

Ongelmana valmistuotteen ostamisessa on laatutekijöiden arvioinnin vaikeus. Koska ohjelmistoarkkitehtuurista ei ole käytettävissä tarvittavaa dokumentaatiota, ei laadun arviointia voida suorittaa luotettavasti. Koekäytössä laatutekijöiden testaus on erittäin vaikeaa. Esimerkiksi suorituskykymittauksen tekeminen vaatisi, että kaikki ohjelmiston osatekijät olisivat tuotantokäytössä. Järjestelmän luotettavuutta on myös erittäin vaikea arvioida. Laatutekijöille kannattaa, jos järkeviä mittareita löytyy, antaa arvot, jotka toimittajan tulee täyttää. Laatu jää joka tapauksessa asiakkaan riskiksi, koska laadun asiakas voi todeta vasta, kun tuote on toiminnassa. Vaikka havaittaisiinkin, että hankittu tuote ei täytä kaikkia vaatimuksia, voidaan tuote silti joutua ottamaan käyttöön. Jos ohjelmaa ei tällöin hyväksytä käyttöön, voi syntyvä haitta olla huomattavasti suurempi sinä tapauksessa, että toimittajaa vaihdetaan.

Ongelmana asiakkaalla olivat organisaatiossa hajallaan olevat sovellukset ja tiedon yhtenäisyyden puute. Samaa tietoa käsitellään monessa paikassa monta kertaa. Ratkaisuksi etsittiin keskitetty ratkaisu, jossa organisaatio käsittelee keskitettyä tietokannanhallintajärjestelmässä olevaa tietoa. Toteutettu ratkaisun viitearkkitehtuuri on monikerrosarkkitehtuuri, jossa sovellukset ajetaan sovelluspalvelimella. Järjestelmä integroituu muihin järjestelmiin esimerkiksi työpöytäintegraation avulla tai sanomarajapintojen avulla (esim. HL7), jolloin esimerkiksi perinnejärjestelmät voidaan kääriä kääreen avulla. Tilaaja on varsin tyytyväinen hankintaansa ja odottaa sille pitkää 15-20 vuoden elinkaarta.

5.3 Esimerkki 2 – haastattelu

Toisena tapauksena oli alun perin yhdenkäyttäjän sovellus, joka toimii käyttäjän PC:ssä (Liite 4). Haasteltava oli ohjelmiston loppukäyttäjä. Ohjelmiston avulla suorittiin oppilasarvostelu oppilaitoksissa. Oppilaitoksia oli useita. Jokaiseen kouluun asennettiin ohjelma ja rekisteri. Ohjelman monipuolisten ominaisuuksien ansioista jokainen koulu muokkasi rekisteriä mieleisekseen. Tiedon yhtenäisyys katosi. Mikrotukea työllisti yksittäiskoneissa olevien sovellusten ajan tasalla pito, virheidenkorjaus ja käyttäjätuki. Koulupiirin tuli laatia oppilasarvostelusta raportteja ministeriölle ja muille viranomaisille. Tämä oli vaikeaa, koska eri koulujen tiedot olivat epäyhtenäisiä.

Ratkaisuksi ongelmiin muodostui tiedon keskittäminen. Toteutettiin hajautettu asiakas/palvelin (client/server) -pohjainen järjestelmä, jossa kaikkien koulujen tieto keskitettiin palvelimelle, jossa toimii palvelinsovellus. Asiakkaat ovat yhteydessä palveluun asiakas-sovelluksen avulla. Ratkaisu ratkaisi kaikki ongelmat. Tieto on yhtenäistä, tie-

dosta saadaan haluttuja raportteja, mikrotukea tarvitaan vähemmän ja mikä tärkeintä koulujen ei tarvitse käyttää aikaa sovellusten ylläpitoon. Toteutettuun ratkaisuun käyttäjä on erittäin tyytyväinen.

5.4 Esimerkki 3 – haastattelu

Kolmantena haastateltiin oma sovellustaan myyvää yritystä (Liite 5). Tuotteena oli portaaliohjelmisto, joka on tarkoitettu yrityksen informaation käsittelyyn ja viestintään. Sekä myyjä että asiakkaat ovat tyytyväisiä tuotteeseen. Ohjelmisto toimii Microsoftin alustalla. Yritys miettii parhaillaan ohjelmistosta uutta versiota, jossa se siirrettäisiin uudelle alustalle. Vaihtoehtoja on kaksi joko ohjelmisto suunnitellaan J2EE-alustalle tai muunnetaan Microsoftin .NET-alustalle. Yrityksellä on kokemusta J2EE-alustasta, joten alustan vaihtaminen ei sinänsä muodostu kynnykseksi. J2EE-alustan puolesta puhuu se, että sen avulla ohjelmistoa voidaan käyttää käyttöjärjestelmäriippumattomasti. J2EE-ratkaisu voi tuoda uusia asiakkaita ja lisenssimaksujen alentumisen ansiosta tuotteesta saa paremman hinnan tai hintakilpailukykyä voidaan lisätä. .NET-ratkaisun puolesta puhuu se, että kyseinen alusta on ominaisuuksiltaan J2EE:n veroinen. Edelleen olemassa olevien asiakkaiden on helppo siirtyä uuteen ohjelmistoon, koska aiemmat kokemukset voidaan hyödyntää. Ja viimeiseksi aiemman ohjelmiston uudistaminen .NET-ympäristöön on helpompaa kuin siirtyminen J2EE-ratkaisussa. Yritys ei haastattelutilanteessa ole tehnyt tuotekehityspäätöstä.

5.5 Yhteenveto ja johtopäätökset haastatteluista

5.5.1 Haastattelujen tulokset

Kaikki haastateltavat kokivat ohjelmistokehityksen tärkeiksi. Jokaisella oli ongelma, johon haettiin ratkaisua. Ongelmat olivat hyvin konkreettisia: sovellusten yhteistoiminta, integraation puute, tiedon hajautus ja pirstaleisuus, teknologian kehitys ja muunneltavuus. Järjestelmien vaatimusten hallinnassa toiminnallisten vaatimusten etsiminen, valitseminen ja arvioiminen oli yleensä hallinnassa. Samoin käyttäjälle näkyvät ominaisuudet kuten käytettävyys tutkittiin.

Laadullisten ominaisuuksien arviointia ei joko tehdä ollenkaan tai jos tehdään, ei ominaisuuksien arvioimiseen ole luotettavia mittareita. Laadullisten tekijöiden arviointi on asiakasyritykselle usein ylivoimainen tehtävä. Tutustuminen tuotteeseen nojaa myyjän argumentteihin ja myyjän toimittamaan infoon. Myyjän referensseihin tutustuminen voi

auttaa jonkin verran. On huomattava, että referenssikohteelta ei aina välttämättä saada todenmukaista tietoa, vaan tieto on aina jotenkin suodatettua. Ohjelmiston koekäyttö parantaa asiaa. Olisi hyvä, jos ohjelmaa voisi käyttää omassa ympäristössä, tällöin voidaan varmistua ainakin siitä, että se toimii omassa laitearkkitehtuurissa. Testikäytön ongelmana ovat asentamisen työmäärä, testitapausten puute ja kuormitustestauksen suorittamisen ongelmat.

Tilanne on kokonaan toinen ohjelmistoja tuottavalla yrityksellä. Yritys voi tutkia vapaasti kehitettävän ohjelmiston elementtejä eli omien elementtien rakenteita ja niiden liittymistä ostettuihin komponentteihin. Omia komponentteja voi arvioida laatutekijöiden suhteen eli tutkia sitä, mihin komponenttiin jokin laatutekijä kohdistuu, ja arvioida, millä toimenpiteillä arkkitehtuuri vastaisi asetettuja vaatimuksia. Varsinkin laatutekijöiden painotus on tällöin tärkeää. Jos esimerkiksi tehottomuus johtuu pääosin käytetystä väliohjelmistosta, ei tilannetta voida korjata, vaikka oman liiketoimintakomponentin algoritmeja tai tietokantayhteyksiä tehostettaisiin tai olioviittauksia rajoitettaisiin.

Käyttöjärjestelmä-, väliohjelmisto- ja sovellustason ohjelmistojen uudelleenkäyttö on laajaa. Ohjelmistot ostetaan usein ulkopuolisilta toimittajilta. Tilanne on siis tältä osin aivan sama kuin lopputuotteen ostaneella asiakkaalla. Myyjä on lähes ainoa tietolähde. Komponentti tai systeemi voi olla mustalaatikko. Tämä on koko ohjelmistoteollisuuden ongelma. Ostettiinpa toimisto-ohjelma tai tuotannonhallintaohjelmisto tai vaikkapa sanomanvälitin, joudutaan luottamaan myyjään, ja myyjä ei yleensä ota vastuuta koko tuotteesta. Integroiduissa tuotteissa "vastuunkantajia on lähes mahdoton löytää". Räätälöidyissä ohjelmissa voidaan sopimuskäytännön avulla varmistua joistakin ominaisuuksista.

5.5.2 Yhteenveto

Yritykset hyötyvät arkkitehtuurin arvioimisesta. Kysehän on myös laadusta, laadun arvioimisesta. Laadun arvioinnin voisi aloittaa vaikka siten, että muodostaa luettelon tärkeimmistä laatuattribuuteista ja arvioi kunkin laatuattribuutin tärkeyttä yrityksen ohjelmistoarkkitehtuurille (Liite 5 ja Liite 6). Kutakin laatutekijää voisi sitten arvioida erikseen ja tutkia sen vaikutuksia ohjelmistoon ja etsiä keinoja, joilla laatua voidaan parantaa arkkitehtuurin keinoin. Arvioinnissa voi käyttää kokemusta, protoilua tai simulointia. Jos edellä mainittujen keinojen avulla ei uskota saavutettavan riittävän hyvää käsitystä asiasta, kannattaa muodostaa profiileita. Profiili on kokoelma skenaarioita. Profiili

on muodostettu niin, että se koostuu määräävimpien laatutekijöiden pohjalta tehdyistä skenaarioista. Skenaarioiden pohtimisessa kannattaa käyttää apuna esimerkiksi UML:n mukaisia esityksiä (tilakaavio, sekvenssikaavio, aktiviteettikaavio, sijoittelukaavio) [Fow99].

Haastattelujen perusteella haastattelijalle muodostui käsitys, että arkkitehtuurin arvioinnista hyötyvät sekä asiakas että ohjelmistotoimittaja. Tämän vuoksi olisi hedelmällistä, että arkkitehtuurin dokumentaatiota käytettäisiin asiakkaan ja toimittajan väliseen kommunikaatioon nykyistä laajemmassa mittakaavassa. Dokumentaatio on laadittava siten, että sen näkökulma on asiakaslähtöinen. Tällöin esitetty aineisto palvelee päätöksentekoa ja siitä on välitön hyöty osapuolille [Bos00].

6 Tavoitteena arkkitehtuurin parantaminen

6.1 Tuotantoprosessin parantaminen – uudelleenkäytön lisääminen

6.1.1 Kohdealueen analysointi

Liiketoiminnan analysointi (analysis of business activities) tutkii yrityksen liiketoimintaprosesseja. Analyysin perusteella löydetään olemassa olevia hyviä tai kehittämistä kaipaavia prosesseja. Voidaan myös lähteä puhtaalta pöydältä ja suunnitella prosessi kokonaan uudelleen. Kun on kartoitettu toiminnan kannalta tarpeelliset prosessit, suoritetaan kullekin prosessille tarpeellinen vaatimusmäärittely. Prosessin vaatimukset voidaan asettaa kolmella tavalla: Uusi prosessi korvaa jonkun aiemman prosessin. Vastaavanlaisia prosesseja on muualla organisaatiossa: Samanlaisia prosesseja on kuvattu kirjallisuudessa. Analyysi tuottaa dokumentaatiota. Määritellään prosessin tarkoitus, toimintaympäristön mallinnus, käyttäjäskenaariot, prosessin tulosteet ja raportit, olemassa olevat manuaalit ja käyttäjien kokemukset. Prosessille tehdään riskikartoitus ja kartoitetaan *kriittiset menestystekijät (critical success factors CSF).* CSF-kartoituksen avulla selvitetään vaadittavat tulokset ja tavoitteet. Tällaisia ovat esimerkiksi markkinaosuuden kasvattaminen tai varaston kieronopeuden kasvattaminen. Kartoittaminen suoritetaan prosessin avainhenkilöiden haastattelun avulla tai ryhmähaastattelujen avulla. Muita avustavia tietojenkeruumenetelmiä ovat erilaiset kyselyt, palautteet ja prosessien mittaustiedot [Ber95 s. 103].

Kohdealueen analyysin (DA domain analysis) tarkoituksena on paitsi analysoida kohdealuetta myös löytää uudelleenkäytettäviä komponentteja kohdealueen ongelmien ratkaisemiseksi. Ongelmana komponenttipohjaisessa kehityksessä voidaan nähdä se, että kohdealueen analyysi tehdään pintapuolisesti ymmärtämättä kohdealuetta riittävän syvällisesti ja että siinä keskitytään ainoastaan komponenttien etsimiseen. Kohdealueen analysoinnissa on oltava käytettävissä riittävä perehtyneisyys kohdealueen erityispiirteisiin. Analyysin tasoa parantaa, jos ohjelmistoarkkitehti on perehtynyt kohdealueen erityisominaisuuksiin. Jos näin ei ole, tarvitaan analyysiin riittävä panos kohdealueeseen perehtyneen asiantuntijan taholta ja lisäksi se, että arkkitehdillä on käsitys kohdealueen erityispiirteistä. Analyysin pohjalta löydetään käyttökelpoisia komponentteja. Tehokas tuotanto edellyttää, että tuotannossa kyetään käyttämään uudelleen olemassa olevia

komponentteja. On olemassa lukuisa joukko erilaisia tekniikoita, joilla tuottavuutta voidaan parantaa. Kaikkien tekniikoiden ongelma on se kuinka ne saadaan organisaatiossa tehokkaaseen käyttöön niin, että niitä käytetään jokapäiväisessä tuotannossa. Metodit ja menetelmät on integroitava yrityksen tuotantoprosessiin [For02 s. 152].

Yksi mielenkiintoinen tapa analysoida kohdealuetta on tehdä se käyttäen *hierarkkista kohdealueen analyysia (hierarchical domain analysis HDA)*, jossa analysoidaan kohdealuetta vaiheittain ja asteittain (Taulukko 3). Kohdealueeksi valitaan koko yritys ja sen tuotantoprosessit. Ensimmäinen vaihe keskittyy yrityksen pääpiirteiden analysointiin. Analyysi integroidaan osaksi tuotantoprosessia [For02 s. 152]. Mallissa ensimmäinen vaihe tehdään vain kerran. Toisessa vaiheessa käydään lävitse meneillään oleva kehityshanke ja tunnistetaan ne alikohdealueet, joihin se kuuluu. Kolmannessa vaiheessa tunnistetaan ne komponentit, jotka tarvitaan liiketoiminnan ja käyttäjien tarpeiden tyydyttämiseksi. Komponentit luokitellaan, jotta luokittelun avulla komponentit voidaan löytää myös seuraavissa projekteissa. Tässä vaiheessa löydetään uudelleenkäytettävät komponentit ja tunnistetaan mahdolliset uudelleenkäytön kohteet kohdealueessa. Hierarkiassa kuljetaan taaksepäin jokaisen tunnistettavan komponentin osalta, jotta saataisiin selville, mitä mahdollisuuksia uudelleenkäyttöön muilla alikohdealueilla on. Neljännessä vaiheessa tutkitaan mahdollisia ratkaisuja ja projektin toteutustapoja tarkemmin. Valitut ratkaisut rajoittavat komponenttien mahdollisuuksia kommunikoida keskenään. Kommunikointi tapahtuu joko väliohjelmistojen tai järjestelmäohjelmistojen kautta.

HDA:n avulla löydetään mahdollisesti sopivia komponenttiehdokkaita ja tutkitaan voisiko samoja elementtejä käyttää useammassa alikohdealueessa. HDA sopii suuremmille organisaatioille, joille koosta johtuen muodostuu hierarkisia rakenteita. Suurissa organisaatioissa on tapahtunut kehitystä, jossa eri mm. osastoilla on sovelluksissa käytetty komponentteja, ohjelmistoja koordinoimatta kehitystä yrityskohtaisesti. Tällöin joudutaan ongelmiin tiedon yhtenäisyyden (consistency) ja yhteentoimivuuden (interoperability) kanssa, josta aiheutuu tarvetta suorittaa järjestelmien välillä intergraatiota.

Taulukko 3. Hierarkinen kohdealueen analyysi [For02 s. 155]

1.vaihe: Luo yritykselle malli (domain model).	1.1 Tunnista yrityksen pääprosessit. 1.2 Määrittele ohjelmistokehityksen perusteet. 1.3 Luo alikohdealueiden (sub-domain) hierarkiat lähtien liiketoimintanäkökohdista. 1.4 Luo alikohdealueiden hierarkiat lähtien ohjelmistonäkökohdista.
2. vaihe: Määrittele alikohdealueet (sub-domain).	2.1 Määrittele tämän hetkisen kehitysprojektin rajat. 2.2 Löydä alikohdealueiden informaation lähteet (information sources). 2.3 Määrittele alikohdealueiden ongelmat ja vaatimukset.
3. vaihe: Analysoi liiketoimintanäkökohdat (analyze business aspect).	3.1 Määrittele liiketoimintaprosessit. 3.2 Määrittele avainkonseptit. 3.3 Määrittele tarvittavat palvelut. 3.4 Analysoi ja tunnista palvelut. 3.5 Luokittele palvelut.
4. vaihe: Analysoi ohjelmistonäkökohdat (analyze software aspect).	4.1 Määrittele kerrokset ohjelmistonäkökohdille. 4.2 Määrittele käytettävät ohjelmistoratkaisut. 4.3 Luokittele löydetyt sovellukset.
5. vaihe: Käytä analyysin tuloksia (use domain analysis results).	

Kartoituksen avulla löydetään prosessit, joilla on samoja piirteitä niin, että ne voidaan suorittaa yhdenmukaisesti. Yhdenmukainen tuotanto mahdollistaa tiedon yhtenäisyyden, samaa tietoa ei tarvitse kirjata kahta kertaa tai sama päivitys on kaikkien käytettävissä. Tiedon yhtenäisyys vaatii myös replikoinnin suunnittelun, jotta tehokkuus säilyttäen voitaisiin taata kaikille riittävän tuore päivitetty tieto. Yrityksissä ja organisaatioissa törmätään usein tilanteisiin, joissa käytössä oleva tietojärjestelmä muodostaa spagetin, jossa kukin yksikkö on hankkinut omaa käyttöään varten juuri kyseiselle yksikölle parhaan vaihtoehdon. Yrityksen kannalta tilanne voi olla huono. Integrointikustannukset voivat olla liian suuria. Varsinkin käyttäjän koneisiin hankittujen sovellusten ja käyttäjän laitteissa olevan tiedon integrointi voi olla pulmallista. Tavoitteena tulisi olla integraatiopisteiden minimointi niin, että yhteistoiminnallisuus sovellusten välillä tapahtuu palvelinkerroksessa ja tietokannanhallintajärjestelmä on suunniteltu siten, että samaa tietoa voidaan käyttää useissa sovelluksissa. Työpöytätason integrointi ei ole ratkaisuna suositeltava siksi, että se rajoittaa yhteistoiminnallisuutta käytettäessä eri tyyppisiä pää-

telaitteita ja käyttöjärjestelmiä. Lisäksi sovellustason yhteistoiminnallisuus tarjoaa paremman tietoturvan.

HDA:n avulla yritys voi luoda strategian prosessiensa hallintaa varten. Suoritetussa kartoituksessa selvitetään yrityksen tuotantoprosessit, käytetyt ohjelmisto, komponentit ja muut elementit. Kartoituksen perusteella päätetään mitä prosesseja uusitaan, kuinka tieto järjestetään, mitkä prosessit säilytetään tai mitä kehitetään sekä kuinka prosessien välinen integrointi suoritetaan.

6.1.2 Prosessin parantaminen

Ohjelmistotuotantoa harjoittavilla yrityksillä on yleensä ISO:n (9001 palveluille, 9000-3 ohjelmistoyrityksille) mukainen laatujärjestelmä. Laatujärjestelmän olemassaolo merkitsee sitä, että yritys on miettinyt ja suunnitellut toimintansa. Prosessien ja tuotteiden laadunvalvontaa on analysoitu ja laatukäsikirja on luotu ohjeeksi päivittäistä toimintaa varten. ISO 9001:ssa on määritelty parikymmentä klausuulia. Klausuuleja on esimerkiksi laadusta, tarkastuksista, korjaavista toimenpiteistä, katselmoinnista, koko tuotantoprosessista aina vaatimusmäärittelyn suorittamisesta valmiin tuotteen toimittamiseen saakka [Jal99 s. 325-].

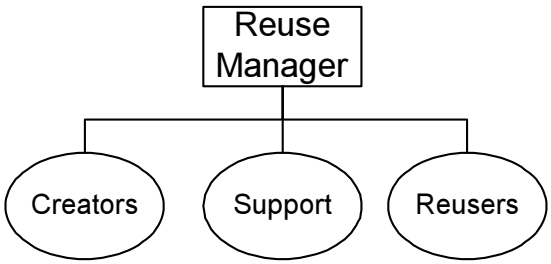
Tuotannon kypsyyden arvioimiseksi ja parantamiseksi yritykset tarvitsevat jonkinlaisia menetelmiä esimerkiksi CMM tai SPICE. Kypsyyden analysoimiseksi suoritetaan *laatujärjestelmästä analyysi (gap analysis),* jolla pyritään selvittämään millä CMM:n tasolla jokin ISO:n prosessi on. Kun prosessit on käyty lävitse, saadaan yleiskuva siitä, millä tasolla yrityksen toiminta on. Analyysin perusteella voidaan miettiä, onko tarvetta tai resursseja parantaa tuotantoprosessia pyrkimällä ylemmälle tasolle. Tuotannon kypsyyden parantaminen on resurssikysymys. Se vaatii aikaa, rahaa ja työvoiman allokointia. Tähän ei ole aina valmiutta [Jal99 s. 24 ja s. 331].

Yrityksen, joka pyrkii parantamaan prosessin hallintaa kuten aikataulutusta, kustannustehokkuutta tai laadunhallintaa (time-to-market, cost, quality), tulisi pyrkiä ainakin CMM:n kypsyystasolle kolme [Roy98 s. 364]. Tällöin yrityksellä on eväät määritellä toiminta niin, että tuotantoprosessit voidaan rakentaa uudelleen tai muokata niitä ja siirtyä komponenttipohjaiseen tuotantoon, jossa otetaan huomioon sekä komponenttinäkökulma että arkkitehtoninen näkökulma.

6.1.3 Uudelleenkäytön lisääminen

Jacobson esittelee *RSEB (Reuse software engineering business)* -mallin uudelleenkäytön lisäämiseksi [JGP97]. Mallissa organisaatioon luodaan komponenttien tuottajatiimejä (creators), komponenttien käyttöä tukeva tiimi (support) sekä komponenttien käyttäjäryhmiä (reusers). Komponenttien tuotanto ja käyttö on erotettava toisistaan, koska tuotanto on hitaampaa, kalliimpaa ja pyrkii parantamaan uudelleenkäyttöä (Kuva 36). Uudelleenkäyttäjät ovat projekteja toteuttavia ryhmiä, jotka toteuttavat projekteja pyrkien saavuttamaan projektin tavoitteet. Komponentin tuottajan ja kuluttajan tavoitteiden ristiriitaisuuksien takia tulisi pyrkiä siihen, ettei ainakaan samassa projektissa mikään projektiryhmä joudu toteuttamaan molempia rooleja [JPG97 s. 20].

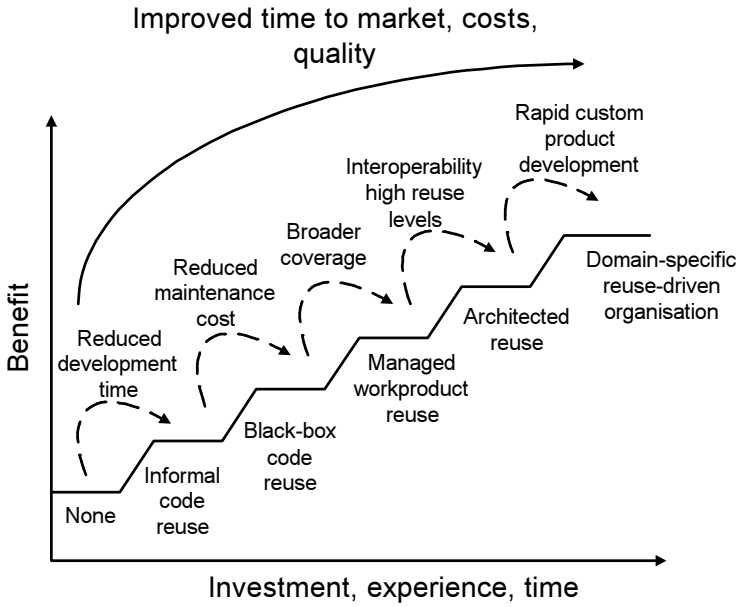
Jotta komponenttien käytössä ja uudelleenkäytössä yleensäkin olisi jotain mieltä, tulisi luotua komponenttia käyttää ainakin kolme tai neljä kertaa, jotta siihen uhrattuja kustannuksia saadaan takaisin. Uudelleenkäytettävän komponentin variointi on halpaa. Jacobson arvioi, että uudelleenkäyttö variointeineen on neljä kertaa halvempaa kuin kokonaan uuden luominen. Edelleen tuotantoprosessin muuttaminen niin, että se mahdollistaa uudelleenkäytön kestää useita vuosia ja siitä koituu taloudellista hyötyä vasta prosessin loppuvaiheessa [JPG97 s. 23].



Kuva 36. Uudelleenkäytön organisointi [JPG97 s. 20]

Uudelleenkäytön lisääminen uudelleenkäytön abstraktiota kasvattamalla parantaa tuotteen markkinoitavuutta eli se saadaan nopeammin markkinoille, voidaan käyttää testattuja ominaisuuksia ja tuote saadaan markkinoille edullisemmin. Abstraktion alimmalla tasolla on koodin uudelleenkäyttö. Jo sillä saavutetaan edistystä projektin toteuttamisessa. Arkkitehtuurin uudelleenkäytöllä saavutaan jo merkittäviä etuja. Tällöin ovat käytettävissä hyväksi havaitut ratkaisut sekä organisaation itsensä kehittämät että mallijärjestelmien tuomat vaihtoehdot. Arkkitehtuurin avulla voidaan kommunikoida asiakkaan, alihankkijan ja oman organisaation kesken. Jäljitettävyytensä ansiosta arkkitehtuurin hyväksikäytöllä voidaan nostaa tehokkuutta. Tehokkainta ohjelmistotuotanto on silloin,

kun on mahdollista organisoida tuotanto niin, että se on kohdealuespesifistä ja komponenttisuuntautunutta. Tällöin puhutaan uudelleenkäyttösuuntautuneesta tuotannosta (reuse-driven) [JPG97 s. 21] (Kuva 37).



Kuva 37. Inkrementaalinen uudelleenkäytön omaksuminen [JGP97 s. 21]

Tuotantoprosessi voidaan jakaa osiin niin, että se koostuu sovellusperheen, valmiin sovelluksen sekä komponenttisysteemin rakentamisosasta. Lisäksi tarvitaan toimenpiteet tuotannon järjestämiseksi. Kuvassa 38 on esitetty malli tuotannon järjestämiseksi.

Mallissa *sovellusperheen rakentaminen (application family engineering AFS)* -prosessi tarkoittaa sitä, että analysoitu kohdealue tai liiketoiminta jaetaan riittävän yleisellä tasolla kerrosarkkitehtuuriksi siten, että siitä voidaan muodostaa alisysteemeitä. Tavoitteena on luoda arkkitehtuurit alisysteemien ja komponenttisysteemien kerroksille, julkisivuille ja rajapinnoille siten, että ne tukevat koko sovellusperhettä. RSEB toteutetaan iteratiivisesti ja inkrementaalisesti niin, että eri prosessit tapahtuvat rinnakkain ja vuorovaikutuksessa keskenään (AFS, ASE ja CSE). AFS kehittää ja ylläpitää kerrosarkkitehtuuria, joka esitetään ylisysteeminä (superordinary) sisältäen tärkeimmät käyttötapaus-, analyysi-, suunnittelu-, jakelu- ja testimallit. Toteutustapa on iteratiivinen: ensiksi käsitellään tärkein käyttötapaus ja sitten seuraava jne. Kehityksen aikana malliin liitetään COTS-komponentit ja perinnejärjestelmät [JGP97].

Sovelluksen rakentaminen (application system engineering ASE) –prosessi valitsee, erikoistaa ja asentaa komponenttisysteemeissä olevia komponentteja rakennettavaan asia-

kassovellukseen sekä rakentaa tarvittavat laajennukset ja suorittaa tarvittavan räätälöinnin. Mallissa valmistaja (manufacturer) vastaanottaa uuden version sovelluksesta sen jälkeen, kun se on kehitetty valmiiksi ja tuottaa ja toimittaa täydellisen sovelluksen asiakkaalle.

Komponenttisysteemin rakentaminen (component system engineering CSE) –prosessi tarkoittaa sitä, että suunnitellaan, rakennetaan ja pakataan komponentteja komponenttisysteemiin. Prosessi käyttää sopivaa koodia, mallipohjia, malleja, sanastoja, dokumentteja ja asiakaskohtaisia työkaluja. Prosessi alkaa tarpeesta luoda komponenttisysteemi. Aluksi kerätään ja analysoidaan aineistoa. Tavoitteena on luoda malli, joka on konsistentti ja joka ilmaisee eksplisiittisesti perusominaisuuden ja muunneltavuuden siinä sovellussarjassa, joka uudelleenkäyttää näitä komponentteja. Komponentti on joko konkreettinen (as-is) tai erikoistettavissa (abstract). Komponentinrakentaja luo arkkitehtuurin, lisää sen suunnitelmaan ja rakentaa ja testaa komponenttisysteemin. Prosessi sisältää sertifioinnin ja komponenttisysteemin pakkaamisen uudelleenkäyttäjiä varten [JPG97 s. 41-44].

RSEB:ssä *toiminnan johtamisessa (managing the reuse business)* on kaksi vaihetta. Ensimmäisenä RSEB on otettava käyttöön ja toiseksi toiminta on kyettävä järjestämään tehokkaasti. Käyttöönotto on syytä suorittaa vaiheittain. Aluksi luodaan liiketoiminnalle tavoitteet ja säännöt. Luotu aineisto määrittelee alustavasti liiketoimintaprosessit, arkkitehtuurin, organisaation ja uudelleenkäytön tavoitteet. Seuraavaksi arvioidaan uutta liiketoimintaa. Kuvataan korkealla tasolla visio uudesta arkkitehtuurista, ohjelmistoliiketoimintaprosesseista ja organisaatiosta. Tavoitteet dokumentoidaan. Laaditaan joitakin siirtymäsuunnitelmia. Sen jälkeen arvioidaan nykyistä liiketoimintaa, arkkitehtuuria, prosesseja ja organisaatiota. Huomioiden edelliset kartoitukset kehitetään haluttu ohjelmistonkehitysprosessi (software engineering process), toimintaympäristö ja valitaan työkalut. Tämän jälkeen implementoidaan uusi toimintamalli liiketoimintaan. Kun prosessi on toiminnassa, suoritetaan jatkuvaa prosessin arviointia ja parantamista [JPG97 s. 343-344].

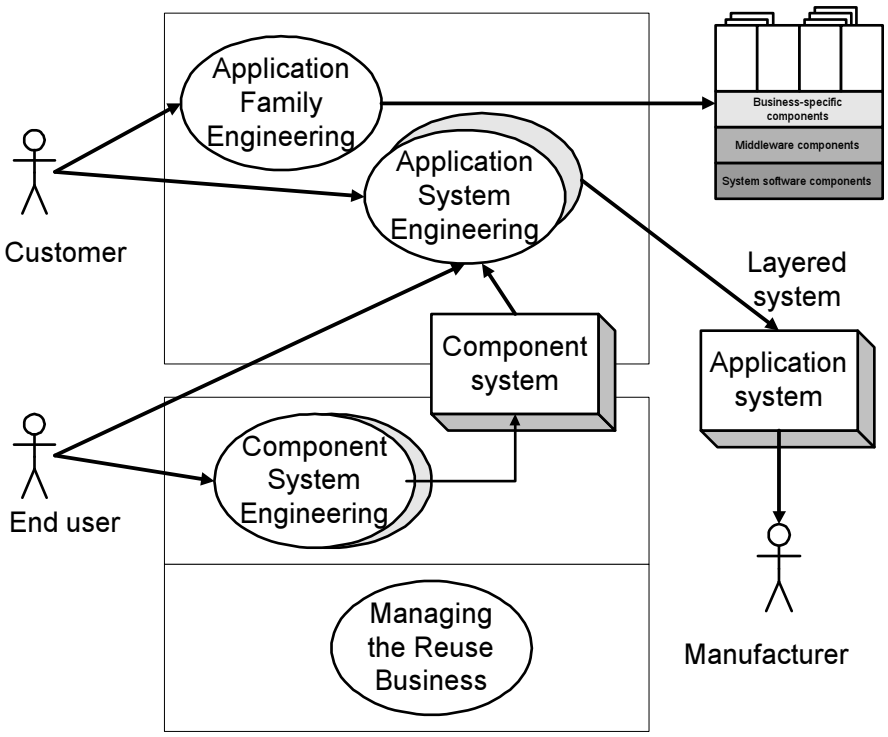
Toiseksi projektin johtaminen on erittäin tärkeää. Jotta RSEB-liiketoiminta voisi ylipäätään onnistua, tarvitaan sitoutumista. Esimerkkinä tässä on yrityksen ylin johto, jonka on sitouduttava sata prosenttisesti hankkeeseen. Toteutettaessa liiketoimintaa ilmenee

useita ongelmia. Ongelmia aiheuttavat projektien ja komponenttien erilainen aikaskaala, josta aiheutuu jännitteitä kehitystiimien välille.

Johtamisessa tärkeää on kyetä koordinoimaan eri tiimien toimintaa siten, että tiedetään, mitä missäkin tapahtuu ja että eri tiimit tietävät toistensa tekemisistä riittävästi. On varmistuttava siitä, että liiketoiminnan tavoitteet ovat kaikilla tiedossa. On osattava ihmisten ja organisaation johtaminen. Konfliktit on kyettävä ratkaisemaan niiden ilmaantuessa. Esimerkiksi, jos komponenttisysteemitiimi ei kykene toimittamaan sovellustiimille ajoissa haluttuja komponentteja, voidaan ongelma ratkaista esimerkiksi viivyttämällä toimitusta, toimittamalla komponenttisysteemi vajavaisin ominaisuuksin ja täydentämällä sitä myöhemmin tai toimittamalla sovellus, jossa on sovittua vähemmän toiminnallisuutta. Hätätapauksessa ohjataan resursseja ohjelmistokehitykseen ja räätälöidään jokin ratkaisu, jotta toimitus tapahtuu ajallaan.

Onnistunut johtaminen tarvitsee mittarit, joilla voidaan mitata toimintaa ja suorittaa tarvittavat ohjaustoimet. RESB:ssä muun ohjauksen lisäksi on kyettävä arvioimaan, uudelleenkäytön tuloksellisuutta. Eräs tapa suorittaa mittauksia on muuttaa resursseja euroiksi. Siirrytään sisäiseen laskutukseen, jossa komponenttien käytöstä eri tiimien välillä suoritetaan laskutusta. Tällöin kustannusten ja tehokkuuden arviointi on mahdollista. Ongelmana tässä on siirtohinnan määrittely, joka aiheuttaa kitkaa organisaatiossa. Toinen tapa arvioida tehokkuutta on ylläpitää kirjanpitoa komponenttivaraston käytöstä. Tällöin kirjataan projektien käyttämät komponentit ja komponenttien käyttökerrat. Käyttökertojen avulla saadaan selville komponentin käytön yksikköhinta, jota voidaan verrata budjetoituun arvoon.

Komponenttien uudelleenkäytön edistämiseksi tulisi komponenttien olla helposti käytettäviä, lunastaa lupauksensa ja olla luotettavia. Komponenttien sertifiointi tarkastuksineen varmistaa sen, että komponenttisysteemin kaikki osat on tarkastettu ja testattu. Luottamusta voidaan kasvattaa myös asiallisen dokumentaation avulla. Tällöin komponentista laaditaan käyttöohje, tuotetiedot sekä liitetään oheen testausmateriaali.



Kuva 38. Uudelleenkäytön elementit [JGP97 s. 41]

6.2 Laadun määrittäminen ja laadun arviointi

6.2.1 Laatuattribuuttien asettaminen

Ohjelmiston laadusta on parikin määritelmää (IEEE 1061 ja ISO/IEC Draft 9126-1), jotka määrittelevät keskeisimpiä laatuattribuutteja. Laatutekijät luokitellaan kuuteen kategoriaan, joilla on alaluokituksia. Nämä pääluokat ovat toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys [DoN02 s.639].

Yrityksen prosesseista muodostetaan kokonaisuuksia, joihin lisätään mahdollisesti uusia prosesseja. Prosesseja kootaan sopiviksi kohdealueiksi tai prosesseja jaetaan osaprosesseiksi. Saadaan muodostettua ohjelmistoprojektia varten kohdealue. Muodostetusta kohdealueesta saadaan vaatimukset vaatimusmäärittelyä varten. Määrittelyn laadulliset (qualitative) vaatimukset saadaan joko kohdealueesta tai kontekstista. Laadulle tulee asettaa mittarit. Mittareiden tulee olla sellaisia, että tavoitteen toteutuminen voidaan jotenkin todentaa. Laatua arvioidaan tuotteen elinkaaren kuluessa. Eräs ongelma on se, että laatukäsite vaihtelee tuotteen elinkaaren mukaan. Esimerkiksi tehokkuutta arvioidaan arkkitehtuurin suunnitteluvaiheessa eri mittareilla kuin rakentamisvaiheessa, jolloin on jo käytettävissä simulointimalleja tai prototyyppejä. Arkkitehtuurivaiheessa ei voida varmasti sanoa, mikä on tuotannon läpimenoaika kuormituksella X, mutta voidaan sanoa millainen arkkitehtuuri olisi paras valitulle attribuutille. Sen sijaan tuotanto-

vaiheessa voidaan suorittaa suorituskykymittauksia jollain tasolla ja todeta, että ajassa X tuotannon läpäisee Y tapausta.

Laatuattribuuttien asettamisessa joudutaan määräämään kaksi ominaisuutta: laatutekijän tärkeys ja arvo laatutekijälle. Vaikka arkkitehtuurin arviointivaiheessa ei kaikkia määrällisiä vaatimuksia voidakaan toteuttaa, olisi hyvä asettaa myös määrällisiä (quantitative) vaatimuksia. Määrällisten laatutekijöiden toteuttaminen voidaan tällöin huomioida jo arkkitehtuurin suunnitteluvaiheessa iterointien aikana.

Laadullinen vaatimus on boolean-tyyppinen, joko vaatimus saavutetaan tai ei. Määrällinen vaatimus on sellainen vaatimus, joka voidaan todentaa. Vaatimus joko alitetaan tai ylitetään. Määrällisen vaatimuksen toteutumisesta on asianosaisten helpompi päästä yksimielisyyteen. Laadullisen ominaisuuden hankaluus on sen epämääräisyys. Koska tavoite joko toteutuu tai ei, jää väliin jää aina jonkun verran harmaata aluetta, josta muodostuu helposti asianosaisten kesken tulkintoja.

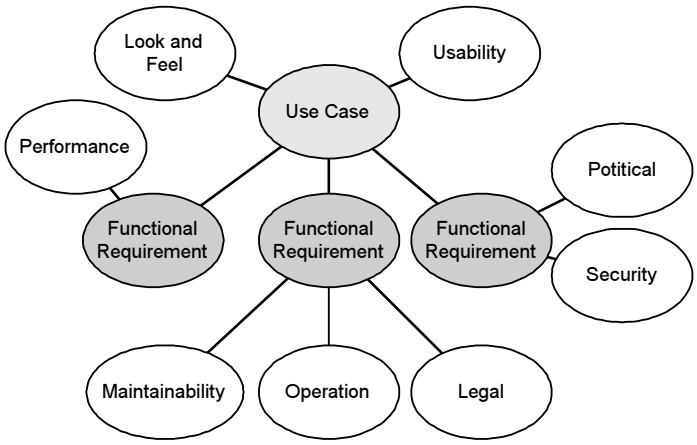
Laadullisten vaatimusten määrittely ja mittareiden asettaminen kuuluu vaatimusmäärittelyn laatimiseen (requirement specification). Karkeasti yksinkertaistaen voi todeta, että hyvä määrittely tuottaa hyvän tuotteen ja huonosta määrittelystä ei synny mitään paitsi taloudellisia menetyksiä.

Toiminnallisuus esitetään käyttötapausten avulla. Kullakin käyttötapauksella on omat ei-toiminnalliset vaatimukset. Nämä vaatimukset muodostuvat joukosta kaikkien asianosaisten vaatimuksia (Kuva 39) [Rob99 s. 114-].

Tuntuma (look and feel)-vaatimukset kuvaavat sitä, miltä tuote näyttää ja tuntuu. Esimerkiksi vaatimuksia voivat olla seuraavat. "Tuotteen käyttöliittymätuntuman on oltava samanlainen kuin muissa yhtiön tuotelinjan tuotteissa" tai "tuotteen oltava käyttökelpoinen kaikilla graafisilla työpöydillä (Windows, Motif, Gnome, KDE)" [Rob99 s. 114-]. Käytettävyysvaatimuksien (usability) tarkoituksena on kehittää tuote huomioiden sen käyttäjäkunta. Käyttäjä voi olla esimerkiksi satunnainen asiakas, jatkuvasti laitetta käyttävä ei-ammattilainen tai ammattilainen. Vaatimus voisi olla vaikka seuraava: "Ohjelmiston tulee olla opittavissa kahdessa viikossa" [Rob99 s. 114-]. Suorituskyky (performance) mittaa jonkin tehtävän suorittamista. Mittaaminen tapahtuu määrällisten arvojen perusteella. Esimerkiksi vaatimus voi olla: "Tapahtumankäsittely saa kestää korkeintaan

30 sekuntia kaikissa oloissa".[Rob99 s.114-]. Toiminnalliset (operational) vaatimukset kuvaavat toimintoympäristöä. Vaatimukset kattavat toimintoympäristön, käyttäjien toiminnalliset ehdot, yhteistyössä toimivat järjestelmät sekä määräykset tuotteiden siirrettävyydestä. Esimerkkinä on vaikkapa: "Tuotetta on voitava käyttää vaikka muut yhteistoiminnassa olevien järjestelmien palvelut eivät ole saatavilla". Ylläpidettävyysvaatimukset (maintainability) kattavat tuotteeseen sen elinkaaren aikana kohdistuvat vaatimukset. Näitä vaatimuksia ilmenee organisaatiolta, ympäristöltä, liiketoiminnan säännöistä sekä mahdollisesti lainsäädännöstä. Hyvä vaatimus on esimerkiksi: "Tuotteen tulee olla versioitavissa eri kielille". Turvallisuusvaatimukset (security) kattavat tietoturvaan liittyvän alueen, joita ovat luottamuksellisuus (confidentially), saavutettavuus (availability) sekä eheys (integrity) [Rob99 s. 114-].

Asiakkaan ja käyttäjien kulttuuriset ja poliittiset vaatimukset (cultural and political requirements) on huomioitava. Nämä vaatimukset sanelevat sopivat ja sopimattomat toimintatavat kuten esimerkiksi sen, missä järjestyksessä jokin työ on suoritettava. Lainsäädännölliset vaatimukset (legal) on myös huomioitava. Esimerkkinä vaikkapa seuraava: "Ohjelmiston on noudatettava kaikilta osin voimassa olevaa lainsäädäntöä ja erityisesti kirjanpitolakia ja asetuksia.".



Kuva 39. Käyttötapauksen ei-toiminnalliset vaatimukset [Rob99 s. 112]

Hyvä vaatimusmäärittely on laadittu niin, että kertoo asiakkaan vaatimukset täsmällisesti, mutta ei rajoita tarpeettomasti suunnittelijan vapautta hänen etsiessään asiakkaan tarpeisiin parasta mahdollista ratkaisua.

6.2.2 Arkkitehtuuri ja laatuattribuutti – laatutekijän kohdistaminen arkkitehtuuriin

Ohjelmiston vaatimukset muodostuvat toiminnallisista ja laadullisista vaatimuksista. Toiminnalliset vaatimukset ovat toteutettavia kohdealuespesifisiä vaatimuksia, jotka sovelluksen on toteutettava. Toiminnallisen vaatimuksen toteuttaa jokin alisysteemi tai komponenttisysteemi. Toiminnallisuus on siis jäljitettävissä. Laatuvaatimukset voidaan luokitella kehityksen aikaisiin laatuvaatimuksiin (development QR) ja toiminnallisiin laatuvaatimuksiin (operational QR). Kehitysnäkökohta lähtee rakentamisnäkökulmasta, jossa ylläpidettävyys, uudelleenkäytettävyys, joustavuus ja havainnollisuus ovat tärkeässä asemassa. Toiminnalliselle näkökulmalle taas tärkeitä ovat suorituskyky, luotettavuus, sitkeys ja vikasietoisuus. Yleensä laatuvaatimuksia ei voida kohdistaa tietylle osalle ohjelmistoa, vaan vaatimuksen kattaa koko ohjelmisto. Sen sijaan voidaan arvioida, mikä arkkitehtuurissa vaikuttaa eniten laatuvaatimuksen toteuttamisessa [Bos00].

Transformaatioprosessissa lähtökohtana on tilanne, jossa laatuattribuutin arviointi tuottaa tuloksen, johon ei olla tyytyväisiä. Tällöin lähtien olevasta arkkitehtuurista muokataan uusi arkkitehtuuri. Uusi arkkitehtuuri tyydyttää laatuvaatimuksen. Muokkaaminen tehdään lähtien ylimmältä abstraktiotasolta eli koko järjestelmän arkkitehtuurista (architectural style). Katsotaan ensiksi, onko siihen syytä tehdä muutoksia. Seuraavaksi edetään arkkitehtuurimalleihin ja katsotaan tarvitsiko siellä tehdä muutoksia. Jos vaadittava muutos on paikallinen, kohdistuu esimerkiksi yhteen komponenttiin ja sen yhteyksiin muihin komponentteihin, voi riittää, että sovelletaan toista suunnittelumallia. Laatutekijä pyritään muuttamaan toiminnallisuudeksi. Esimerkkinä tästä ovat virheenkäsittelytoiminnot. Koska vaatimuksia on vaikea kohdistaa komponenteille, voidaan vaatimus jakaa esimerkiksi alisysteemeille osittamalla.

6.3 Arkkitehtuurin arvioinnin kehittäminen

6.3.1 Arkkitehtuurin arviointi yleistä

Arkkitehtuurinen ajattelutapa ei ole suinkaan uusi asia. Jo niinkin varhain kuin 70-luvalla Parnas pohti modularisaation etuja ja tiedonkätkentää ohjelmiston modulaarisuuden avulla. Tavoitteena oli tällöin parantaa joustavuutta ja yhtenäisyyttä. Lisäksi modulaarisuus-ajattelu paransi ohjelmien selkeyttä vähentäen ja tuoden ilmi prosessien väliset kytkennät. Ohjelmistotuotannossa on aina mukana riski. Riski on kustannustekijä, aikataulutekijä sekä osaamistekijä. Riskianalyysin avulla pyritään kartoittamaan näh-

tävissä ja ennakoitavissa olevat riskit. Ohjelmistossa riskikartoitus kattaa myös arvioinnin siitä, kuinka rakennettava ohjelmisto pystyy kattamaan asetetut vaatimukset [DoN02 s. 638]. Ohjelmisto on aineeton hyödyke ja ohjelmistoprojektin alussa ohjelmistoa tai jotain sen ominaisuutta ei yleensä ole olemassa. Tarvitaan menetelmiä, joiden avulla ohjelmiston rakennetta ja arkkitehtuuria pystytään arvioimaan. Analysointiin on useita tapoja.

Arkkitehtuurin arvioimisen ongelmana on se, että mittaaminen on hankalaa, koska ohjelmistoa ei ole vielä rakennettu. Ohjelmistoarkkitehtuurin analysointitekniikat voidaan luokitella kahteen kategoriaan: *arkkitehtuuripohjaisiin (architecture oriented)* tai *laatuattribuuttisuuntautuneisiin (quality attribute oriented)*. Laatupohjaiset tekniikat jaetaan kahteen luokkaan: laadulliseen ja määrälliseen. Laadullinen tarkastelu vertailee kahta arkkitehtuuria ja päättää kumpi ehdokas täyttää laatuvaatimuksen paremmin. Määrällinen arviointi pyrkii ennustamaan tarkasteltavan arkkitehtuurin todellista reaali-aikaista käyttäytymistä. Arkkitehtuuripohjaista arviointia käytetään sen jälkeen, kun arkkitehtuuri on valmis. Laatuarviointia käytetään arkkitehtuurisuunnittelun aikana osana iterointia [BoB01].

Arkkitehtuuripohjaisten arviointitekniikoissa voidaan tarkastelu suorittaa kahdella tavalla joko kysyen (questioning techniques) tai mittaamalla (measuring techniques). Kysyvissä tekniikoissa muodostetaan laadullisia kysymyksiä, joiden avulla pyritään selvittämään haluttuja ominaisuuksia. Tätä tekniikkaa käytetään skenaarioissa, kyselyissä tai tarkistuslistoissa. Kyselyn vastauksia voidaan arvioida esimerkiksi viisi portaisella arvioinnilla (++,+,+/-,-,--). Mittaavat tekniikat tutkivat tiettyä ominaisuutta tai vastaavat tiettyyn kysymykseen. Mittaavia tekniikoita ovat esimerkiksi mittaukset, simulaatiot, prototyypit ja kokemusperäinen mallintaminen [DoN02 s. 640].

Useimmat vakavasti otettavista analyysimenetelmistä käyttävät skenaarioita. Skenaarioiden avulla voidaan ohjelmiston laatua tulkita havainnollisesti ja helposti ymmärrettävästi. Skenaarion avulla kuvattava laatutekijä on konkreettisempi ja rakennettavaan systeemiin spesifisempi kuin kyseisen laatutekijän yleinen kuvaus. Skenaariot ovat joukko järjestelmän käyttöjä tai modifikaatioita. Muutos kuvaa sitä kuinka järjestelmä käyttäytyy, kun komponentti suorittaa jonkun toiminnon, kuinka jonkin komponentin lisäys vaikuttaa toimintaan tai suoritetaan kytkentä kahden komponentin välille tai tehdään jokin näiden tekijöiden yhdistelmä. Arviointeja suoritettaessa arvioidaan ominai-

suuksia asianosaisten (stakeholders) kannalta. Tutkitaan kunkin roolin kuten kehittäjän, ylläpitäjän tai loppukäyttäjän kannalta, miltä skenaarion tulokset näyttävät hänen kannaltaan [DoN02 s. 641].

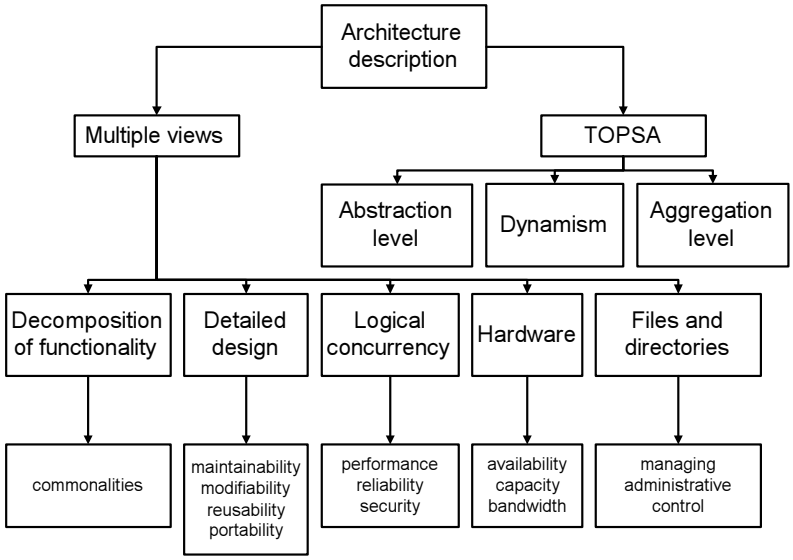
Arvointimenetelmiä ovat muun muassa skenaariopohjaiset arviointimenetelmät (scenario-based architecture analysis method SAAM) eri versioineen, arkkitehtuurin kompromissianalyysi (architectere trade-off analysis method ATAM), skenaariopohjainen arkkitehtuurin uudistaminen (scenario-based architecture reengineering SBAR) [DoN02]. SBAR:n periatteita on esitelty kohdassa 4.5.2.

Ohjelmistosuunnittelu sisältää aina arviointia. Laadun arviointia tapahtuu jatkuvasti tuotantoketjussa tuotteen koko elinkaaren aikana. Arkkitehtuurin arvioinnin avulla pyritään käyttämään hyväksi kaikki se kokemus ja tietämys, joka organisaatiolla on käytettävissään. Jos arkkitehtuurin arviointi ei ole systemaattista, joudutaan helposti siihen tilanteeseen, jossa tuotantovaiheeseen pääsee keskeneräisiä jopa huonoja arkkitehtuurimalleja. Tällöin arviointi suoritetaan rakennettuja osien avulla testaamalla tai protoilemalla valmiita osia. Jos prosessi päästetään tähän pisteeseen, voidaan ennustaa, että projektin talous, aikataulu ja ennen kaikkea laatu eivät vastaa projektille asetettuja toiveita. Voisi sanoa, että toiminta on tällöin ad-hoc –tyyppistä. Tälläkin tavalla saadaan tulosta aikaan silloin, kun toteutettavana on organisaatiolle tutun tuotteen ylläpito- ja parannustehtäviä.

Ohjelmistoarkkitehtuuri kuvaa rakennettavaa ohjelmistoa käyttäen useita erilaisia näkymiä (multiple views). Eri näkymien avulla pyritään auttamaan monimutkaisuuden hallinnassa (complexity). Näkymien avulla voidaan kuvata staattista rakennetta, dynaamista käyttäytymistä, fyysistä rakennetta ja sijoittelua sekä systeemi kehittymistä [DoN02].

Laadun arviointia arkkitehtuurin perusteella voidaan kuvata (Kuva 40) siten, että arviointia varten valitaan näkymiä ja näkymiin liittyviä ominaisuuksia. *Ohjelmistoarkkitehtuurin formaali ominaisuuksien systematiikka (taxonomy of formally-defined orthogonal software architecture TOPSA)* laajentaa arkkitehtuurin määritelmää. TOPSA sisältää kolme dimensiota: abstraktiotason (käsitteellinen, toteuttava) (abstraction: conseptual, realization), dynaamisuustason (staattinen, dynaaminen) (dynamism: static, dynamic) sekä ryhmittelytason (aggregation level). Sekä TOPSA että näkymät täydentävät toisi-

aan. Eri näkymien avulla voidaan tarjota esimerkkejä näistä dimensioista. Analyysissä TOPSA:n avulla voidaan etsiä sopivimpia laatuattribuutteja [DoN02 s. 640].



Kuva 40. Arkkitehtuurin kuvaus ja suhteet laatutekijöiden arvioimista varten [DoN02 s. 640]

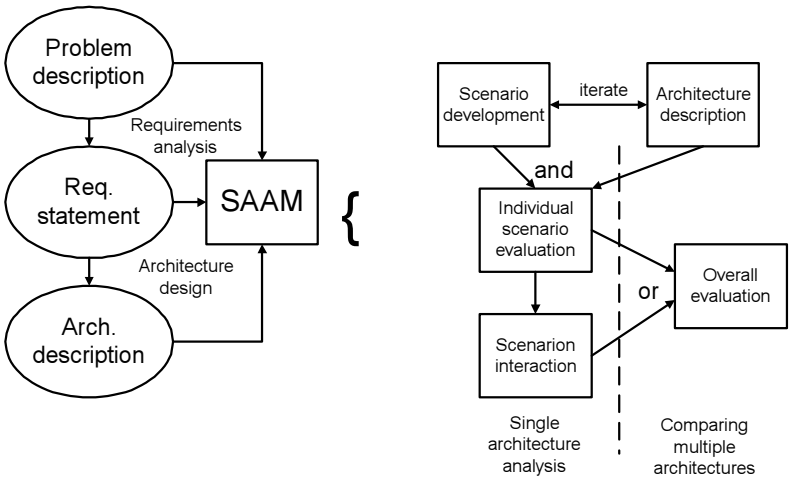
Esitellään lyhyesti kaksi tapaa suorittaa arviointeja. SAAM on näistä yksinkertaisempi, jossa skenaarioiden avulla perusarkkitehtuuria verrataan haluttuihin ominaisuuksiin. SAAM soveltuu paremmin ensimmäiseksi arviointimenetelmäksi. Toisena esitellään ATAM, joka on kehittyneempi iterointiin perustuva malli. ATAM:a voidaan käyttää silloin, kun organisaatio käyttää iteroivaa inkrementaalista ohjelmistokehitystapaa.

6.3.2 SAAM

SAAM on kehitelty 90-luvun alkupuolella ja sen tarkoituksena antaa parempi ymmärrys arkkitehtuurin yleisistä periaatteista. Tavoitteena on ollut, että arkkitehtuurisuunnittelun alkuvaiheessa luodun arvioinnin perusteella tehtyjen korjausten avulla voidaan arkkitehtuuria parantaa kustannustehokkaasti. SAAM ohjaa arkkitehtuurin tarkastustoimintaa etsimään mahdollisia ongelmakohtia (trouble spots). Perusarkkitehtuuria verrataan halutun tuotteen ominaisuuksiin. Perustekniikkana käytetään skenaarioita. Skenaariot kuvaavat niitä toimintoja, joita järjestelmän tulee tukea ja mitä muutoksia järjestelmään tulee tehdä. Menetelmä konkretisoi laatuattribuutit. Laatutekijät muutetaan siis toiminnallisuudeksi. SAAM pyrkii ottamaan huomioon eri asianosaisten vaatimukset, jotta luotava systeemi vastaisi koko elinkaarensa ajan asianosaisten vaatimuksia [DoN02].

Menetelmää käytetään arkkitehtuurin siihen versioon, joka on riittävän kypsä arvioitavaksi. Tässä vaiheessa arkkitehtuurikuvauksen tulee olla sellainen, että asianosaiset voivat omaksua sen helposti. Arkkitehtuurista kuvataan toiminnallisuus, rakenne ja allokointi. Järjestelmästä laaditaan myös lyhyt kirjallinen selvitys. Arkkitehtuurista kuvataan staattinen (static representation) ja dynaaminen esitys (dynamic representation). Staattinen esitys sisältää tietojenkäsittelyn, komponentit, datan ja kontrollin kytkennät (data and control connection). Dynaaminen esitys käsittelee sitä, kuinka järjestelmä käyttäytyy ajan kuluessa [DoN02].

Menetelmässä syötteeksi saadaan ongelmien kuvaus, vaatimusmäärittely ja arkkitehtuurin kuvaukset (Kuva 41). Asianosaiset osallistuvat skenaarioihin. SAAM tutkii, mitkä arkkitehtoniset elementit vaikuttavat skenaariossa. Skenaarion avulla löydetään muutoksia vaativat elementit. Lopuksi skenaarioille annetaan painot, jotta saadaan vaadituille muutoksille tärkeysjärjestys. Tulosten tulkinnassa vaikeuksia voi ilmetä silloin, kun erilliset skenaariot toimivat vuorovaikutussuhteessa keskenään. Tällöin toiminnallisuuden erottaminen voi olla vaikeaa. SAAM ei anna määrällisiä tuloksia. SAAM:n katsotaan olevan kypsä arviointimenetelmä. Sitä on käytetty mm. globaaleissa informaatiosysteemeissä, lennonvalvontajärjestelmissä, käyttöliittymien rakentamisessa ja sulautetuissa järjestelmissä [DoN02].



Kuva 41. SAAM syötteet ja aktiviteetit [DoN02 s. 648]

6.3.3 ATAM

ATAM on kehitetty SAAM:n pohjalta käsittelemään yksittäisiä laatuattribuutteja kuten muunneltavuus (modifiability), turvallisuus (security), suorituskyky (performance) ja

saavutettavuus (availability). ATAM rinnastetaan spiraalimalliin. ATAM:n tarkoituksena on tarjota tapa ymmärtää arkkitehtuurin kyky selvitä tilanteesta, jossa sen on täytettävä useita kilpailevia laatuattribuutteja. ATAM tunnistaa tarvittavan kompromissin (trade-off) useiden attribuuttien suhteen ennen kuin systeemi kehitetään. Myös ATAM vaatii kaikkien asianosaisten osallistumista prosessiin. Arkkitehtuurikuvausta rajoittavat perinnejärjestelmät, yhteistoiminnallisuus ja aikaisempien projektien virheet. Arkkitehtuurikuvauksen näkökulmina käytetään Kruchtenin 4+1-näkymiä [DoN02].

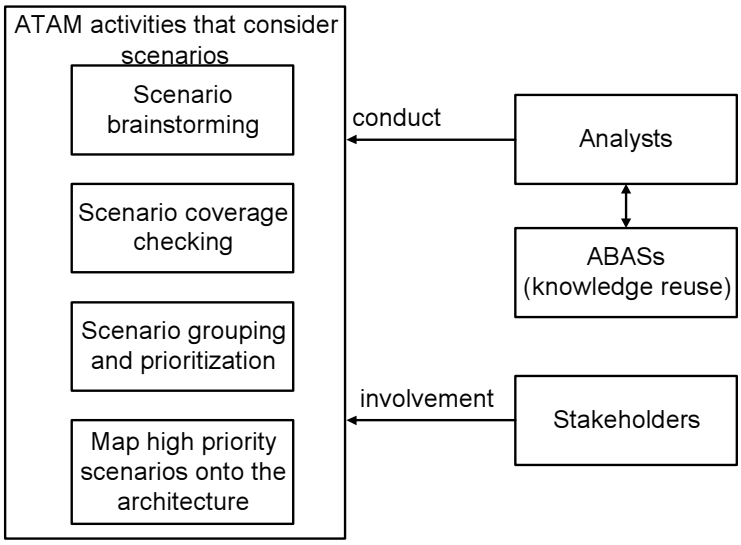
ATAM toimii kehyksenä erilaisille arviointitekniikoille, joita tarvitaan erilaisten laatuattribuuttien käsittelemiseksi. Se integroi yksilöllisen mallin jokaiselle attribuutille. Toinen tekniikka on skenaarioiden käyttö. Käytetään kolmenlaisia skenaarioita järjestelmän tutkimiseksi. Käyttötapausten (Use cases) avulla tutkitaan tavallisimpia käyttötapauksia. *Kehitysskenaarion (growth scenario*) avulla tutkitaan muutosten vaikutusta arkkitehtuuriin. *Tutkivaskenaario (exploratory scenario)* tutkii äärimmäisiä muutoksia ja sitä, kuinka se vaikuttaa järjestelmään [DoN02].

ATAM soveltaa myös laadullisen analyysin heuristiikkoja (qualitative analysis heuristics), jotka periytyvät attribuutti-pohjaisesta arkkitehtuuri tyylistä (attribute-based architecture style ABAS).

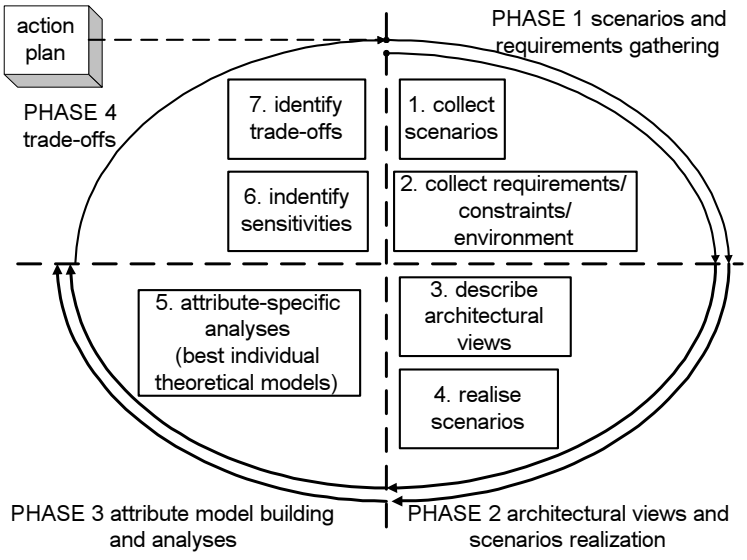
Menetelmä jaetaan neljään päävaiheeseen. Ensimmäisessä vaiheessa kootaan vaatimukset ja skenaariot. Toisessa vaiheessa ovat vuorossa arkkitehtuurin kuvaukset ja skenaarioiden toteuttaminen. Kolmannessa vaiheessa rakennettaan attribuuttimalli ja suoritetaan attribuuttipohjaiset analyysit. Viimeisessä vaiheessa löydetään ja tutkitaan vaadittavat kompromissit (trade-offs) ja sensitiivisyyspisteet. Ensin attribuuttianalyysit luodaan ja malleja analysoidaan riippumattomasti ja sen jälkeen mallit vaihtavat tietoja. Attribuuttianalyysit toimivat toisistaan riippuvasti, koska jokainen attribuutti vaikuttaa toiseen attribuuttiin. *Herkkyyspiste (sensitivity point)* on komponentti, joka on kriittinen jonkin ominaisuuden saavuttamisessa. *Kompromissipiste (trade-off point)* on ominaisuus, joka vaikuttaa yhteen tai useampaan ominaisuuteen ja on herkkyyspiste ainakin yhdelle attribuutille [DoN02].

Arkkitehtuurisuunnittelun aikana ATAM tarjoaa iteratiivisen prosessin parantamisen (iterative improvement). Vaatimukset polveutuvat asianosaisten haastattelusta, toimintaympäristöstä ja käyttäytymismalleista. Koska attribuutit ovat keskenään ristiriidassa,

jokainen olettamus on mahdollisena kohteena tarkastuksen, kelpuuttamisen tai kysymyksen asettamisen suhteen mahdollisena ATAM:n tuloksena. Kun kaikki toimet on toteutettu, verrataan analyysin tuloksia vaatimuksiin. Jos systeemin ennustettu käyttäytyminen on riittävän lähellä asetettuja vaatimuksia, voidaan siirtyä detaljikohtaiseen suunnitteluun. Jos ilmaantuu ongelmia, toteutetaan *toimintasuunnitelma (action plan)* arkkitehtuurin, mallin tai vaatimusten muuttamiseksi. Tällöin aloitetaan toinen iterointikierros. ATAM ei vaadi, että kaikkia attribuutteja käsiteltäisiin samanaikaisesti, riittää, kun määräävimmät attribuutit ovat mukana käsittelyssä. Tämä menettely keventää arviointiprosessia (Kuva 42 ja Kuva 43).



Kuva 42. ATAM toimintoja koskien skenaarioita [DoN02 s. 645]



Kuva 43. ATAM:n vaiheet [DoN02 s. 645]

6.4 Tuotteenhallinta ja tuotelinja-arkkitehtuuri keinoina lisätä uudelleenkäyttöä

6.4.1 Tuotteenhallinnan tehtävät

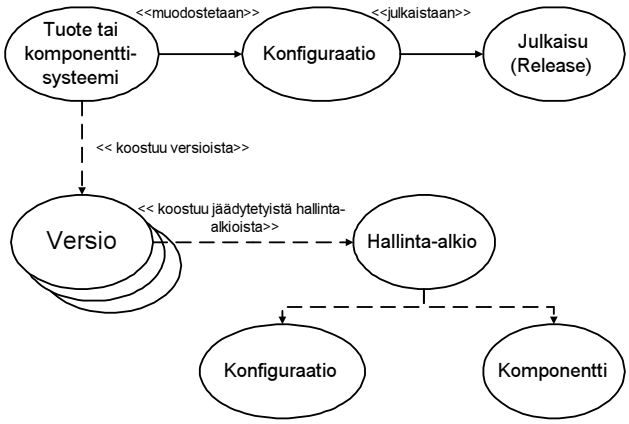
Tuotteet kehittyvät ja evoluution aikana tuotteesta kehittyy uusia kaupallisia versioita. Uudet versiot korvaavat vanhoja tai luotu uusi versio voi olla kokonaan uusi tuote. Tuotteenhallinnan tehtävänä on tässä kehityksessä hallita komponenttien ja komponenttisysteemien eri versioita. Tuotteenhallintaan kuuluvat menetelmät, joilla hallitaan konfiguraatioita sekä toimintatavat, joita noudatetaan tuotteenhallinnassa [HaM01 s. 238].

Tuotteenhallinta (configuration management) sovelluskehityksessä keskittyy ylläpitämään rekisteriä jakelukelpoisten sovellusten, komponenttisysteemien ja tuote- ja tuotelinja-arkkitehtuurien eri versioista ja näiden versioiden välisistä suhteista. Tärkein tuotteenhallinnan tehtävistä on ylläpitää tietoa, mitä versioita missäkin tuotteessa on käytetty ja mitä versioita milläkin asiakkaalla on. Jäljitettävyys (traceability) saavutetaan organisoimalla tuotanto niin, että kehityskaaren eri versiot käsitellään organisaatiossa yhdenmukaisesti niin, ettei synny tilannetta, jossa ei tiedetä mitä versiota jossain kokoonpanossa on käytetty.

Versionhallinnan tarkoituksena on sekä jäljitettävyyden hallinta että tuotannon hallinta niin, että noudatetaan sovittuja käytäntöjä versioiden hallinnasta, versioiden muutosten hallinnasta (change management). Versiot muodostavat versiopuun, joka muodostaa katkeamattoman ketjun tuotteen eri versioiden välille. Versioita muodostetaan esimerkiksi tuotteelle, komponenteille, komponenttisysteemeille sekä tuotelinjan arkkitehtuurille.

Asiakkaalle toimitettava pakettiohjelma julkaistaan (release). Julkaistavan ohjelman kaikki elementit on jäädytetty (freeze) ja julkaistava tuote muokataan jakelukelpoiseksi. Julkaisu voi sisältää tarpeellisia konfiguraatiotiedostoja, tiedostoja, asennusohjelman, asiakasdokumentaation sekä jakeluelementtejä kuten pakkaus, markkinointitoimenpiteitä ja tuotetuen. Julkaisu on muodostettava konfiguraatio, joka on koostettu tuotteen eri elementtien versioista kuten uudelleenkäytettävät komponentit, tuotespesifiset elementit sekä mahdollisesti aiemman tuoteversion elementit. Koostettava tuote muodostuu versioista. Versiot ovat jäädytettyjä hallinta-alkioita (Kuva 44).

Tuotteenhallinta lisää uudelleenkäyttöä. Tuote on voitu rakentaa niin, että siinä on otettu huomioon erilaiset asiakasvaatimukset ja tällöin tuotteesta voidaan räätälöidä asiakaskohtaisia konfiguraatioita. Tuotteenhallinta on läheisessä yhteydessä tuotelinjaan. Tuotelinjan avulla etsitään yhteisiä tekijöitä yrityksen tuotteiden asiakasversioille, jotta voitaisiin muodostaa yhteinen tuotearkkitehtuuri ja myöhemmin tuotelinja. Tuotteenhallinta rekisteröi tuotteeseen ja tuotantoon kohdistuvat tapahtumat. Tuotteenhallinnasta voidaan kartoittaa versionhallinnan kautta tarpeelliset variointikohdat. Variointikohtien löytäminen auttaa tuotelinjan rakentamisessa ja tarpeelliset kehityspisteet löydetään.



Kuva 44. Tuotteenhallinnan käsitteet mukaillen [HaM01 s. 241]

6.4.2 Tuotelinja-arkkitehtuuri ja uudelleenkäyttö

Jacobsonin (kohdassa 6.1.3) mallintama sovellusperheisiin pohjautuva uudelleenkäyttöä edistävä (RSEB) sovellusten rakentamismalli mallintaa komponenttipohjaista sovellustuotantoa [JGP97 s. 41]. Bosch on edelleen kehittänyt mallia tuotelinja-malliksi (software product lines PLA) [Bos00 s. 161]. Molemmissa malleissa on sama lähtökohta. Liiketoiminnan analysoinnin avulla löydetään yhteisiä riittävän yleisellä tasolla olevia piirteitä tuoteperheiden ja tuotelinjojen rakentamista varten.

Tuotelinjan perustaminen vaatii tuotelinja-arkkitehtuuria, joka tukee yksittäisten tuoteperheen tuotteiden toiminnallisia ja laadullisia vaatimuksia. Ongelmaksi voi muodostua tapa, jolla siirrytään tuotelinjaan. Jos tuotelinjaa varten perustetaan oma ryhmä, joka pala palalta analysoiden olevaa toimintaympäristöä muodostaa tuotelinjan ja tuotelinjaa otetaan käyttöön, voidaan riskit minimoida, mutta kustannukset ja time-to-market maksimoida. Jos sen sijaan rakennetaan tuotteelle kokonaan uusi tuotelinja, voidaan tuotelinja muodostaa tuotteen ehdoilla siten, että uudelleenkäytettävyys voidaan maksimoida.

Riskit tässä tavassa ovat suuremmat. Etuna tässä taas on rakentamisen nopeus. Se kumpaa tapaa kannattaa käyttää, riippuu yrityksestä. Jos käytössä oleva tuotanto on laadukasta ja täyttää komponenttipohjaisen tuotannon vaatimukset ja pyrkimyksenä on ollut luoda joustavia rakenteita esimerkiksi käyttäen komponenttikehyksiä tai vastaavia, voisi ajatella, että vanhaa käyttäen kannattaa kehittää uutta. Jos taas vanhaa ei voi käyttää ja teknologiaa on uusittava joka tapauksessa, ei kai liene muuta mahdollisuutta kuin rakentaa kaikki alusta uusiksi. Hyödynnettävissä tässäkin tapauksessa on yrityksen kohdealuetietämys [Bos00 s. 166].

Joskus olemassa oleva tuotelinja kaipaa uudistamista. On voinut käydä niin, että tuotelinja ei enää vastaa liiketoiminnan vaatimuksia ja sitä on parannettava. Aivan kuin siirtymisessä tuotelinjaan myös tässäkin tapauksessa voidaan valita joko evoluutio kautta tapahtuva vähittäinen siirtyminen uuteen tai kerralla muodostettava tuotelinja.

Uuden rakentaminen jäädyttää olemassa olevan tuotelinjan tai tuotteen kehityksen ja siihen tehdään enää vain ylläpitotoimintoja. Tämä voi olla yrityksen kannalta hankalaa, jos uuden kehittäminen kestää kauan. Yritys voi menettää asemaansa markkinoilla joutuessaan kilpailemaan vanhentuneilla tuotteilla asiakkaista. Sen sijaan, jos voidaan toimia markkinoilla niin, että asiakkaille voidaan toimittaa uusia versioita ja vastata asiakkaiden vaatimuksiin evolutiivisen toiminnan kautta, on riski tuotteen vanhenemisesta pienempi. Strategian valinta on erittäin tärkeää. Jos ajatellaan, että tuotelinjaan siirtyminen kestää 1-3 vuotta, on asiakasriski merkittävä. Sen sijaan jos yrityksellä on jo toimiva tuotelinja, tuotelinja uusiminen voidaan tehdä nopeastikin. Jos arvioidaan, että tuotelinja uusiminen voitaisiin tehdä alle vuodessa, on asiakasriski jo huomattavasti pienempi.

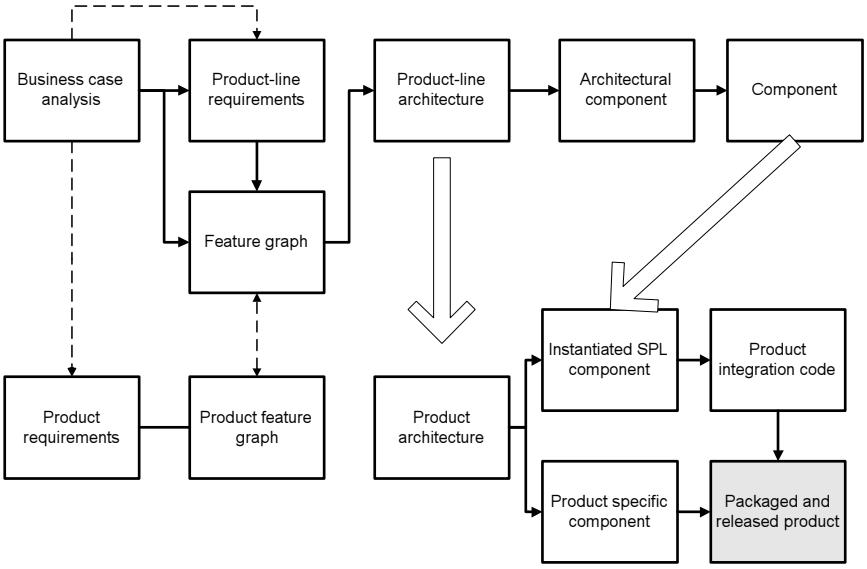
Lähtökohtana tuotelinjan (kuten minkä tahansa muunkin) perustamisessa ja kehittämisessä on suoritettu *liiketoiminta-analyysi (business case analysis)*. Liiketoiminta-analyysin pohjaksi on asetettava ensiksi kysymyksiä. Miksi tarvitaan tuotelinja-arkkitehtuuria? Mitä hyötyä saadaan tuotelinjasta? Miksi kannattaa uhrata resursseja tuotelinjaa? Jos vastaukset ovat myönteisiä ja todetaan, että järkevä tuotelinja on kehitettävissä, kannattaa analyysiä jatkaa. Tuotelinja on kannattava, jos säästetään kustannuksissa, tuotteen valmistusaika lyhenee ja jos tuotteen valmistukseen tarvittava henkilöstöresurssien määrää saadaan alennettua. Tuotelinja avulla on voitava parantaa tuotteiden laatua. Laadun parantaminen tapahtuu ensisijassa uudelleen käyttämällä testattu-

ja, luotettavia komponentteja ja komponenttisysteemejä. Analyysissä kartoitetaan olemassa oleva tilanne, ennakoidaan tuleva tilanne sekä tehdään investointilaskelmia siirtymisestä tuotelinjaan. Analyysin tuloksena saadaan selville tuotelinjan mahdolliset hyödyt ja haitat, joiden perusteella voidaan tehdä tarpeelliset johtopäätökset ja voidaan laatia tuotelinjalle asetettavat vaatimukset [Bos00 s. 189].

Vaatimuksista muodostetaan riippuvuusgraafi (feature graph). Riippuvuudet muodostetaan niin, että analysoiduista vaatimuksista valitaan tuotelinjaan soveltuvat vaatimukset (scoping). Riippuvuusgraafi kartoittaa tuotelinjan ominaisuuksien (feature) sisältymisen yksittäiseen tuotteeseen [Bos00 s. 194].

Tuotelinjaan on valittava myös siihen sisällytettävät tuotteet ja tuotteiden alustavat ominaisuudet. Tehtävä voi olla hankalakin, jos kyseessä on uusi kohdealue tai aivan uusi tuotelinja. Tuotevaatimuksista muodostuu tuotteen ominaisuusgraafi (product feature graph), joka huomioidaan tuotelinjan riippuvuusgraafia rakennettaessa. Ristiriitoja voivat aiheuttaa tuotelinjalle valittavat ja yksittäiselle tuotteelle haluttavat ominaisuudet. Ristiriidat kannattaa käsitellä ennen, kuin aletaan muodostaa arkkitehtuuria, koska ratkaisemattomat ristiriidat on ratkaista jossain vaiheessa. On edullisempaa suunnitella ratkaisu prosessin alkuvaiheessa, kun kaikki käsitteet ovat vielä abstrakteja. Ristiriitoja ei saisi missään tapauksessa lykätä rakentamisvaiheeseen [Bos00 s. 197].

Vaatimuksista saadaan tuotelinjan käyttötapaukset. Kullekin tuotteelle on määritelty vastaavalla abstraktiotasolla olevat käyttötapaukset. Tutkitaan, mitä muunneltavuutta tarvitaan, jotta tuotelinjan ja tuotteiden riippuvuudet kohtaisivat halutulla tavalla. Iteroidaan tarvittaessa. Tämän jälkeen voidaan suunnitella tuotelinjan arkkitehtuuri ja yksittäisten tuotteiden arkkitehtuuri. Suunnitellaan ja rakennetaan alisysteemit elementteineen sekä tuotelinjan että tuotteiden osalta. Arkkitehtuurisuunnittelussa suoritetaan arkkitehtuurin arviointia ja parannetaan arkkitehtuuria, jotta se vastaa laatuvaatimuksia. On huomattava, että tuotelinjan arkkitehtuurin on oltava karkealla tasolla valmiina ennen kuin yksittäisten tuotteiden rakentamista voidaan aloittaa. Ainakin on oltava päätettynä tuotelinjan jakaminen komponentteihinsa ja komponenttien tarjoamat rajapinnat kuvauksineen. Lopuksi paketoidaan ja julkaistaan sovellusohjelma (Kuva 45).



Kuva 45. Tuotelinja-arkkitehtuurin rakentamisprosessi [Bos00 Figure 41 s. 173]

6.5 Liiketoimintakomponenttikehys ja mallintaminen keinona lisätä uudelleenkäyttöä

Liiketoimintakomponenttikehyksen avulla voidaan kehittää liiketoimintasovelluksia. Esitellään malli ComponentAA, jonka mukaisesti voidaan uudelleenkäyttää keskikokoisia komponentteja (medium-size component). Malli on kehysten, mallien sekä metodologian yhdistelmä. Komponenttikehysjoukon ja siihen liittyvien mallien avulla voidaan EJB-papujen avulla kehittää tähän arkkitehtuuriin perustuvia liiketoimintajärjestelmiä. Tämän mallin etuina on käytännönläheisyys, suora liitäntä toteutukseen. Vaikka malli on tarkoitettu suurille yrityksille, soveltuu se myös pienille yrityksille silloin, kun yrityksen toimintaympäristössä kohdealueilta löytyy riittävästi samankaltaisuuksia hyödynnettäväksi, jotta voidaan rakentaa komponenttivarastoa uudelleenkäytettäväksi [GYH00].

Mallin tavoitteena on lisätä tuottavuutta systeemin kehityksessä siten, että vältetään projektien kehittäminen aina alusta aloittaen (from the scratch). Arkkitehtuuri tarjoaa yleisen rakenteen ohjelmistolle kuvaamalla komponenttien roolit arkkitehtuurissa. Sovellusarkkitehtuuri pohjautuu MVC-arkkitehtuurille, jossa rooleja ovat malli (model), näkymä (view) ja kontrolli (control). Poikkeuksena on kuitenkin se, että tämä malli on suunniteltu tunnistamaan eri komponentit ja käyttämään komponenttiteknologiaa hyödyksi esim. EJB:tä ja sen suunnittelumalleja.

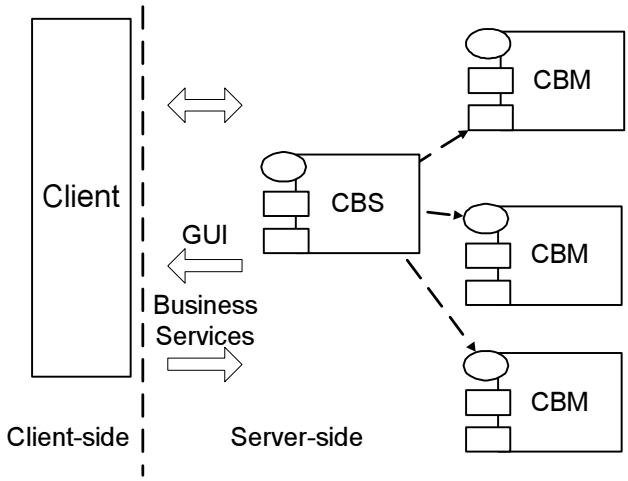
Arkkitehtuuri koostuu kolmesta kerroksesta (three tiers): yksi asiakaskoneella (client-side) ja kaksi palvelimella (server-side). Palvelinkerroksessa tarvitaan kaksi tasoa, koska asiakkaalle tarjottavat palvelut voidaan ryhmitellä ylimääräisen kerroksen avulla. Näin saavutetaan parempi tiedon yhtenäisyys (integrity). Arkkitehtuurin lähtökohtana on *tietokeskeisyys (data-centric approach)*, mikä tarkoittaa sitä, että järjestelmän mallinnus keskittyy liiketoimintaentiteettien keskinäisten riippuvuksien, tiedon yhtenäisyyden ehtojen analysointiin ja suunnitteluun eikä niinkään liiketoimintojen analysointiin. Toinen lähtökohta on *vaiheistetettu lähestymistapa (phased approach)*, mikä tarkoittaa sitä, että malli sisältää notaation vaiheista ja toiminnoista, joita tarvitaan sovellusten kehitysprojekteissa. Kehitysprosessi on iteroiva ja inkrementaalinen [GYH00].

Komponenttien kehittämisessä hankaluutena on löytää sopiva komponenttien rakeisuus. Rakeisuus voisi olla tietokeskeisessä ajattelussa tieto (item level), tietue (record level) ja alisysteemi (subsystem level). Komponentti voi olla tarkoitukseensa joko liian suuri tai liian pieni. Hankkeissa tästä aiheutuu hankaluuksia. Voi olla hankalaa löytää komponentille oikea rakeisuus, jotta siitä muodostuisi aidosti uudelleenkäytettävä ja se ei olisi riippuvainen yleisesti sovellusarkkitehtuurista. Toinen kustannuksia aiheuttava seikka on, että on vaarana, että kehitetään eri kohdealueilla komponentteja, jotka ovat ominaisuuksiltaan samanlaisia, kuin jossain muualla kehitetyt komponentit ja tästä syystä samaa komponenttia suunnitellaan yhä uudestaan ja uudestaan [GYH00].

ComponentAA-kehyksessä on kaksi luonteenpiirrettä: keskikokoiset komponentit ja abstraktit komponentit. Näiden piirteiden avulla pyritään ratkaisemaan granulaarisuuteen liittyvät ongelmat. Keskikokoinen komponentti kuvaa ainakin yhtä tietokannan tietuetta. Abstraktien komponenttien käytön avulla voidaan kuvattavaa aluetta mallintaa yleisemmin ja erikoistaa se kohdealueen liiketoimintaan. Tämä tapahtuu perimällä abstraktimalli ja lisäämällä siihen tarpeellisia sääntöjä. Abstraktia mallia voi myös käyttää sellaisenaan (as-is) [GYH00].

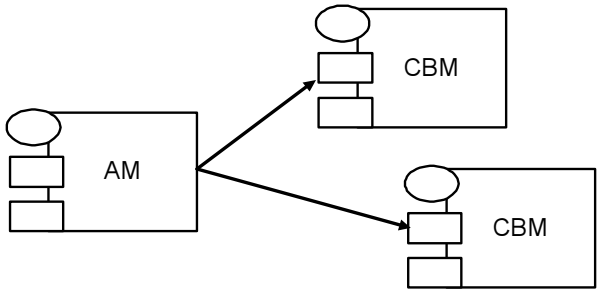
Sovellus koostuu kolmesta osasta: asiakasosasta, liiketoimintapalveluista (core business services CBS) ja liiketoimintamalleista (core business models CBM) (Kuva 46). Asiakkaan sovellus toimii asiakaskoneessa ja sovelluspalvelimella ovat CBS- ja CBM-komponentit. CBS-komponentti toimii julkisivuna ja tarjoaa asiakassovellukselle kaikki sen tarvitsemat rajapinnat ja palvelut. Esimerkiksi käytettäessä EJB-arkkitehtuuria CBS

asennetaan sessio-papuna (session bean). CBM-komponenteissa on tarvittava toiminnallisuus. CBM-komponentti kätkee tiedon mallilta. CBM-komponentti implementoidaan joukkona sessiopapuja ja CBM sisältää myös useita entiteettipapuja (entity bean) [GYH00].



Kuva 46. Sovellusarkkitehtuuri mukaillen [GYH00]

Komponenttiabstraktion avulla tarkastellaan CBM-malleja tarkoituksena etsiä eri malleista samankaltaisuuksia, joita voisi yleistää kattamaan useampia CBM-malleja. Tavoitteena on luoda abstrakteja malleja (AM), joita voisi käyttää yleisesti luotaessa uusia CBM-malleja. Abstraktimalli sisältää osan toiminnallisuudesta ja osa rakennetaan kohdespesifisesti (Kuva 47) [GYH00].



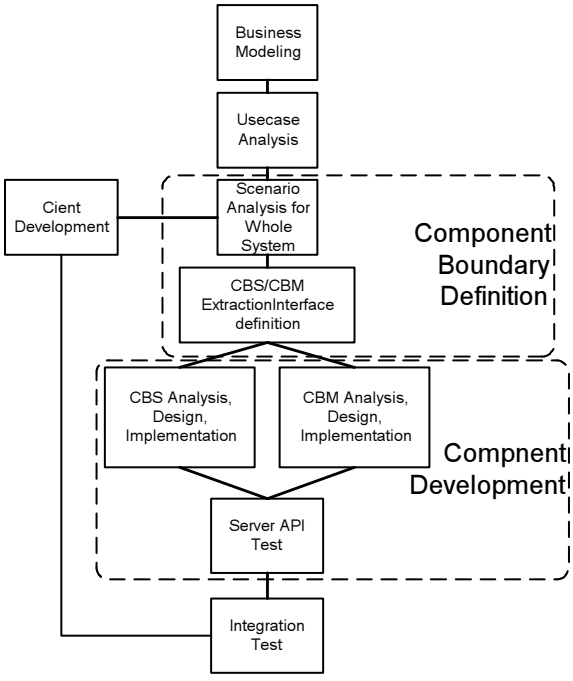
Kuva 47. Abstraktit komponentit mukaillen [GYH00]

Mallintamisen metodologia kohtaa tavanomaiset ongelmat komponenttien kehittämisessä (kts. 6.1.3). Samanaikaisesti joudutaan kehittämään sekä uudelleenkäytettäviä komponentteja että sovellusta. Tämän takia joudutaan määrittelemään huolellisesti komponentin ja tuotteen väliset ominaisuudet, sidokset ja rajapinnat. Komponentit rää-

tälöidään (customize) asiakkaita varten myös pakkaamalla, jotta niiden käyttö olisi joustavaa [GYH00].

Ratkaisuna yllä oleviin ongelmiin mallissa on käytetty suunnittelumalleja esimerkiksi EJB-malleja. Edelleen komponentit identifioidaan mahdollisimman aikaisessa vaiheessa, jotta komponentin sisäinen suunnittelu ja sovelluksen suunnittelu voidaan erottaa. Kuvassa 48 on kuvattu mallin mukainen komponentti- ja sovelluskehitys. Mallissa määritellään käyttötapaus-analyysin (use case analysis) pohjalta skenaarioiden avulla koko systeemi ja löydetään komponentit, joille määritellään rajapinnat ja tarvittavat toimintaehdot (esi- ja jälkiehdot). Tämän jälkeen voidaan sekä sovelluksen että komponenttien kehittämistä jatkaa rinnakkain [GYH00].

Edelleen prosessin on oltava inkrementaalinen, jossa toiminnot (activities) toteutetaan vaiheittain (phases). Toiminnot luokitellaan: systeemin suunnittelu, vaatimusten analysointi, määrittelyjen analysointi, systeemin arkkitehtuurin suunnittelu, systeemin infrastruktuurin suunnittelu, ohjelmiston suunnittelu, koodaus, testaus ja projektinjohto. Vaiheet taas ovat: suunnitteluvaihe, analyysivaihe, rakennusvaihe ja testausvaihe [GYH00].



Kuva 48. Komponentti-pohjainen kehitysprosessi [GYH00 fig. 6]

Kehysten avulla saavutettiin merkittäviä etuja. Komponenttikehyksen avulla on voitu vähentää kirjoitettavan koodin määrää. Suunnitteluvaiheessa resurssienkäyttö on huomattavasti vähäisempää käyttäen kehyksiä ja siihen liittyviä suunnittelumalleja. EJB-teknologia koettiin helpoksi oppia. Pystyttiin luomaan riippumattomia komponentteja käyttäen CBM-komponentteja [GYH00].

6.6 Kohdealuespesifiset sovelluskehykset (Domain-specific application frameworks)

Sovelluskehysten tehokas käyttö edellyttää, että kehys dokumentoidaan asianmukaisesti. Dokumentaatiosta ovat kiinnostuneita kehyksen kehittäjä (framework developer) ja sovelluksen kehittäjä (application developer). Kumpikin tarvitsee itselleen sopivaa informaatiota (taulukko 4) [FaJ s. 619].

Taulukko 4. Sovelluskehyksen dokumentaatio ja kiinnostuksen kohteet [FaJ00 s. 619]

Dokumenttityyppi	kehyksen kehittäjä	sovelluksen kehittäjä
Esimerkki		kyllä
Keittokirja ja reseptit		kyllä
Suunnittelumalli	kyllä	
Rajapintasopimuket	kyllä	kyllä
Viitemanuaali		kyllä
Kehyksen yleiskatsaus		kyllä
Muita dokumentteja	mallinnuskaaviot	Käyttöohje
		Ohjeet virheiden varalle

Taulukossa 4 on kerrottu sovelluskehyksen dokumenttityyppejä. Esimerkkeinä voivat olla vaikkapa lähdekoodiesimerkit, joiden avulla sovellusta voidaan laajentaa tai mukauttaa halutunlaiseksi. Keittokirja (cookbook) on kokoelma reseptejä (recipes), jotka kuvaavat askel askeleelta jonkin tehtävän suorittamista. Rajapinta sopimus kuvaa kehyksen rajapinnan ja sen esi- ja jälkiehdot sekä käytön. Kehyksen viitekäsikirja (reference manual) sisältää kuvauksen luokista, metodeista ja vastuista. Yleiskatsaus (framework overview) kuvaa kehystä siten, että kehittäjä voi sen perusteella aloittaa kehyksen opiskelun. Yleiskatsaus on laadittu siten, että myös muut kuin yllämainitut saavat sen perusteella yleiskuvan kehyksestä. Muita dokumentteja ovat suunnittelumuistiinpanot, käyttöohje, ohjelmoijan manuaali (programmer's manual), ohjeet virheiden varalle (troubleshooting quide), suorat luokka- ja metodiviittaukset (online class and method references) sekä asennusopas (installation guide) [FaJ00 s. 620]

Kehysten kehittäminen vaatii mallinnusvälineet (esim. Rose), visualisointivälineet (esim. Visual Caf´e) sekä tietokannanhallintajärjestelmän (kuten Oracle tai MySQL).

Joillekin kohdealueille on saatavilla myös sovelluskehittimiä, joiden avulla sovelluskehyksestä voidaan kehittimen mallinnusvälineiden avulla luoda sovelluksia. Esimerkiksi Esimerkki 1:ssä on käytetty ohjelmiston kehitysympäristönä Compuware Corporationin tekemää sovelluskehitintä. Suunniteltaessa sovelluskehyksen hankintaa joudutaan allokoimaan resursseja myös koulutukseen. Vaadittava aika voi olla useita kuukausiakin, mutta sovelluskehitin lienee yleensä opittavissa alle kuukaudessa [FaJ00 s. 624].

Kehyksen suunnittelija tarvitsee suunnittelumalleja jatkuvasti työssään. Jotta kehys olisi toimiva ja muunneltava ja ylipäätänsä sovelluksen kehittäjän käytettävissä, luodaan kehykseen toiminnallisuutta suunnittelumallien avulla. Useimmiten käytetyt suunnittelumallit ovat: Strategia, Komento, Silta, Tehdas ja Ainokainen –suunnittelumallit (kts. Liite 8). Arvioidaan, että kehyksen koodista noin 40% koostuu suunnittelumalleista [FaJ00 s. 624].

Joitakin ohjeita kehyksen rakentamiseksi [FaJ00 s. 625]:

- Kehysten ja luokkien välillä on suuri ero. Kehys koostuu luokista ja kehys on uudelleenkäytettävissä.
- Kuumien pisteiden ja suunnittelumallien käyttö on oiva työkalu kehystä rakennettaessa.
- Kehyksen tulisi toimia standardilla alustalla aina, kun se on mahdollista.
- Protoilu on tärkeää.
- Suunnittelu vaatii iterointia.
- Kehysten kehittäminen on ammattilaisten työtä.
- Hyvä dokumentaatio on tärkeää. Huonosti dokumentoitu kehys on hyödytön.
- Koulutus on suunniteltava ja sovittava perusmetodologioista.
- Kehyksen komponenttien välillä tulisi olla mahdollisimman vähän riippuvuuksia, jotta saavutettaisiin mahdollisimman suuri joustavuus.
- Käytetään mahdollisimman paljon erilaisia malleja.
- Suunnitteluryhmän oikea koko on tärkeää.
- Suunnitteluvaiheessa ei pidetä kiirettä.
- Suunnitteluvaiheessa ei pitäisi miettiä implementointia. Riittävä abstraktio on tärkeää.
- Yksinkertainen on kaunista.

Ohjeita kehyksen kaupallistamista varten [FaJ00 s. 627]:

- Asiakkaan vaatimukset ja kehyksen soveltuvuus on varmistettava suunnitteluvaiheessa.
- Kehyksen käyttöönotto vaatii kärsivällisyyttä.
- Kehys lisää luotettavuutta ja stabiilisuutta. Asiakkaan on hyvä tietää tämä.
- Sovellusten kehittäjille tarjotaan kattavat virheenjäljityskoukut (debug hooks) ohjelmiston testaamiseksi, jotta virheen laatu selviää.
- Kehittimeen lisättävän sovelluskohtaisen koodin määrän on oltava mahdollisimman pieni.
- Asiakas on eristettävä kehykseen mahdollisesti tehtävistä muutoksista sen elinkaaren aikana.
- Suorituskyky on tärkeää.

Sovelluskehysten teknologia kehittyy parhaillaan. Tarvitaan standardointia esimerkiksi dokumentaation, integraation ja terminologian suhteen. Kehysten kehittäminen, rakentaminen ja kehyksen opetteleminen on kallista. Jotta kehystä ylipäätään kannattaisi rakentaa ja käyttää, tulisi sillä olla useita käyttökertoja. Ainutkertaisissa pienissä ohjelmissa perinteinen olio-ohjelmointi sovelluskehittimien avulla suoritettuna on tehokkaampaa.

6.7 Yhteenveto

Arkkitehtuurin parantaminen koostuu kohteen analysoimisesta, mallintamisesta ja mallin arvioimisesta ja parantamisesta. Arkkitehtuurin hyväksikäyttö vaatii tuotantoprosessin muokkausta jotta se tukee kehittävää työskentelytapaa. Arkkitehtuurin parantaminen koostuu kahdesta asiasta: laadun parantamisesta ja uudelleenkäytön lisäämisestä. Laadun parantaminen arkkitehtuurin keinoin painottuu mallisysteemien käyttöön ja arkkitehtuurin arviointiin Uudelleenkäyttö kilpailukyvyn parantamiseksi on ratkaisu kustannusongelmaan. Tällöin komponenttiteknologiaa voidaan hyödyntää tehokkaasti. Keinoina uudelleenkäytön lisäämiseksi voidaan nähdä sekä kehystekniikka (komponenttikehykset) että tuotelinjan täysimääräinen käyttö. Pienessä mitassa myös muunneltavien komponenttien rakentaminen voi olla kannattavaa. Komponenttitekniikka lisää myös säästöjä, jotka kohdistuvat tuotteen koko elinikään. Tuotehallinta voidaan hoitaa järkevästi, kun tuote koostuu komponenteista. Tällöin voidaan asiakaspäivitykset hoitaa komponenttitasolla.

Tuotelinja-malli on soveltuvin yrityksille, jotka tuottavat ja markkinoivat tuotteitaan. Tuotelinja voi sopia myös muunlaisille yrityksille. Asiakasprojekteja toteuttava yritys voi muodostaa asiakaskuntansa pohjalta tuotelinjan, jossa tuotteistetaan projektit ja etsitään eri projekteista tuotelinjaan sopivia elementtejä. Elementit voivat täyttä osittain tuotelinjan komponenttipohjaisen ajattelun tunnusmerkit. Uudelleenkäytettäviä elementtejä voivat olla erilaiset käyttötapaukset, joihin voidaan suunnitella variointia tai sitten valmiit alemmalla abstraktiotasolla olevat yksittäisten komponenttien toteutukset. Loppukäyttäjät voivat perustaa oman tuotelinjan, jossa omasta kohdealueesta etsitään tuotelinjaan soveltuvia elementtejä eli tekijöitä joiden halutaan olevat yhteisiä läpi koko organisaation. Tämän jälkeen voidaan rakentaa omaa tietojärjestelmää toteuttaen näin luotua visiota. Ongelmana kahdella viimeisellä on toiminnan projektiluontoisuus ja vaikeus luoda ja panostaa yhteisten ominaisuuksien kehittämiseen [Bos00 s. 169].

7 Pohdinta

7.1 Yleistä

Parhaiten komponenttien käyttö ja markkinoiden luominen on onnistunut työpöydän graafisten ympäristöjen toteuttamisessa esimerkkinä ActiveX-komponentit. Esimerkiksi Visual Basic-komponentteja voi ostaa verkosta [www001].

COTS-liiketoimintakomponenttien osalta tilanne on ongelmallisempi. Komponentteja on saatavilla, mutta aitoja laajoja yleisiä komponenttimarkkinoita ei ole syntynyt. Jotta komponenttien massamarkkinat voisivat syntyä, komponentit on löydettävä helposti, niiden tuotetietojen tulee olla helposti luettavissa ja komponentteihin on voitava luottaa. Ongelma on siinä, että komponentin kuluttajan (reuser) ja tuottajan (creator) vaatimukset poikkeavat toisistaan. Kuluttaja haluaa yleisiä helposti asennettavia ja muiden komponenttien kanssa keskusteluun pystyviä, pitkäikäisiä tuotteita. Lisäksi asiakas haluaisi, että komponentit ovat keskenään vaihdettavissa. Tuottajan lähtökohtana on jatkuva asiakassuhde ja sitä kautta jatkuva maksuvirta. Vaihdettavuus sitä paitsi johtaa siihen, että toimittaja joutuisi kilpailemaan muiden toimittajien kanssa. Tämän vuoksi yleisten vaihdettavien komponenttien tuottaminen ei kiinnosta. Tuottajalle on edullista, jos asiakas saadaan sidottua siten, että hankittuaan komponenttipohjaisen tuotteen asiakas on sidottu investointiinsa.

Pitkällä aikavälillä juoksussa asiakas voittaa. Näin on käynyt systeemiohjelmistojen, väliohjelmistojen ja varusohjelmistojen kanssa. Markkinoilla on aina oma logiikkansa. Kaupalliset liiketoimintakomponenttimarkkinat syntyvät aikanaan, kun kysyntä ja tarjontatekijät kohtaavat. Tarjonta luo kysyntänsä. Kun markkinoille tulee riittävästi tarjontaa kaiken kokoisilta yrityksiltä ja kun liiketoimintakomponenttien hinta/laatusuhde saadaan kohdalleen, tulee saataville riittävästi tavaraa ostettavaksi. Tähän asti myös teknologia on asettanut esteitä. Myytävät komponentit vaativat tietynlaisen virtuaalikoneen (Business Component Virtual Machine BCVM [HeS99 s. 40]), josta johtuu, että komponenttia on hankala käyttää muiden virtuaalikoneiden kanssa. Tarvitaan universaalivirtuaalikone.

Arkkitehtuurin parantaminen on tehokkainta, kun se tapahtuu mahdollisimman korkealla abstraktiotasolla. Tässä tutkielmassa haastatellut henkilöt pyrkivät parantamaan tietojärjestelmänsä palvelukykyä parantamalla arkkitehtuuria yleisellä tasolla esimerkiksi parantamalla prosessia. Loppukäyttäjä voi kehittää ohjelmistonsa arkkitehtuuria parantamalla yleistä liiketoiminta-arkkitehtuuria ja laitearkkitehtuuria sekä kehittämällä käyttämiensä liiketoimintasovellusten ominaisuuksia yhdessä toimittajan kanssa (Kuva 49).

Kuva 49. Tietojärjestelmän toiminnan arviointia

Haastattelututkimuksen kahdessa ensimmäisessä tapauksessa (Esimerkki1 ja Esimerkki 2) toiminnassa oleva tietojärjestelmä ei täyttänyt toiminnalle asetettuja vaatimuksia. Ratkaisuna ensimmäisessä tapauksessa oli eri prosessien parempi yhteistoiminnallisuus. Arkkitehtuuria parannettiin luomalla ensimmäisessä tapauksessa monikerrosarkkitehtuurin mukainen keskitetty järjestelmä ja toisessa client/server-järjestelmä. Arkkitehtuuria kehittämällä pystyttiin tarjoamaan parempi palvelu ja takaamaan tiedon yhtenäisyys. Kolmannessa tapauksessa mietittiin, jatketaanko evoluutiota nykyisellä pohjalla vai rakennettaisiinko kokonaan uusi järjestelmä siirtämällä nykyinen ohjelmisto uudelle alustalle. Haastattelujen perusteella voi todeta, että jo verrattain pienillä muutoksilla voidaan järjestelmän ominaisuuksia parantaa.

Arkkitehtuurin parantamista voidaan suorittaa joko valmiille tuotteelle tai tuotteelle, jota ollaan luomassa. Ensimmäisessä tapauksessa arkkitehtuurin arviointi koostuu mittaamalla saadusta aineistosta. Tällöin tuotteen jotain ominaisuutta voidaan parantaa. Esimerkiksi käytettävyyttä voidaan parantaa karsimalla hierarkioita ja parantamalla loogisuutta sekä pohtimalla, vaaditaanko muutoksia arkkitehtuuriin ja mitä muita vaikutuksia muutoksella on. Valmiissa tuotteessa muutokset ovat evolutionäärisiä, ne tehdään pienin askelin ja muutosten aiheuttamat riskit ovat pieniä. Kun ollaan luo rakentamassaa uutta tuotetta, pohdittavia asioita on paljon ja riskit ovat suuret. Arkkitehtuuri suunnittelun merkitys korostuu. Arkkitehtuurin parantamisen merkitys kilpailukeinona aiheutuu saavutetusta paremmasta kilpailuasemasta. Tuotteen elinikä kasvaa, tuote on hallitumpi ja tuotteen ominaisuudet on arvioitu. Näin asiakkaalle pystytään antamaan oikeampi

kuva tuotteen ominaisuuksista. Arkkitehtuurin parantaminen on tehokkainta silloin, kun voidaan arvioida vapaasti esimerkiksi rakennettaessa uutta. Kun muokkaaminen voidaan tehdä ilman rajoituksia, voidaan tuotteen ominaisuuksia parantaa ennen kuin siirrytään suunnitteluun. Ohjelmistoprojektilla tulisi olla määriteltynä tavoitearkkitehtuuri, johon pyritään. Tavoitearkkitehtuuri on määritelty siten, että sen avulla saavutetaan keskeisimmät tavoitteet.

Arkkitehtuurin parantamisen avulla järjestelmä vastaa paremmin asetettuja vaatimuksia. Toinen yhtä tärkeä vaatimus on tehostaa tuotantoa siten, että kustannukset saadaan alas. Ratkaisuna tähän on kehitysprosessin tehokkaampi toiminta ja se edellyttää uudelleenkäytettävien elementtien hyödyntämistä. Uudelleenkäyttö on sitä tehokkaampaa, mitä korkeammalla tasolla se tapahtuu. Esimerkiksi tavoittelemisen arvoinen malli uudelleenkäytön lisäämiseksi on liiketoimintakomponenttitasoinen kehys (BC-level framework). Tämän käyttö edellyttää jo kehittynyttä komponenttituotantoprosessia, jotta komponenttien yhteistoimintaa saadaan aikaiseksi. Komponenttikehysten käyttö on tehokasta, koska kehys tarjoaa komponentille rajapinnan kehyksen tarjoamiin palveluihin ja yhteistoimintaan muiden komponenttien kanssa. Aina ei kehysten käyttö ole mahdollista tai järkevää. Tällöin kannattaa pyrkiä käyttämään malleja, joiden avulla voidaan luoda komponenttien välille vuorovaikutusta (collaboration pattern). Komponentin sisällä taas yhteistoiminta kannattaa luoda sekä kehysten avulla että käyttämällä olioohjelmoinnin suunnittelumalleja.

Komponenttikehysten käytön kehittäminen vaatii lisätutkimusta. Tarvetta on tekniikkariippumattomista määrittelyistä, joiden avulla myös pienemmät yritykset voivat rakentaa tekniikkariippuvia komponenttikehyksiä ja käyttää niitä sovelluksissaan. Tavoitteena määrittelyssä tulee olla se, että ne ovat yleistettävissä ja modifioitavissa kulloiseenkin käyttötarkoitukseen. Komponenttikehys on tällöin pohjana rakennettaessa tuotelinjaa. Tavoitteena tuotelinjan käyttämisessä on lisätä uudelleenkäyttöä ja parantaa pienten yritysten kilpailuasemaa. Määrittelyn tulee olla avoin ja hyvin dokumentoitu, joten se soveltuu luontevasti esimerkiksi yliopiston perustutkimusohjelmaan. Määrittely on myös pilotoitava yhteistyökumppanin kanssa, jotta sen käyttökelpoisuus voidaan todeta ja saadaan vuorovaikutusta yliopiston ja yritysten kanssa.

7.2 Palkanlaskentajärjestelmä esimerkkinä arkkitehtuurin käytöstä

7.2.1 Kuvaus järjestelmästä

Tässä esimerkissä tutkitaan pienen alle sadan hengen yrityksen tuotteen ominaisuuksien parantamista siten, että oliopohjaiseen toteutustapaan tottunut yritys pyrkii siirtymään jossain määrin komponentti- ja arkkitehtuuripohjaiseen ajatteluun. Esimerkissä on korostettu hyötyjen välitöntä yhteyttä toimintaan.

Kuvitteellisessa ohjelmistoyrityksessä A oli markkinoitu palkanlaskenta-ohjelmaa, jossa oli useita satoja asiakastoimituksia. Ohjelmisto oli toteutettu monoliittisesti käyttäen olio-ohjelmoinnin periaatteita. Aluksi ohjelmisto oli koodattu C++-kehitysympäristössä, mutta nyt järjestelmä on toteutettu Javalla. Asiakkaille oli toimitettu järjestelmää sekä Unix/Linux- että Windows-ympäristöihin. Käyttöliittymiä oli rakennettu sekä työpöytäsovelluksina että selainversiona. Asiakastoimituksissa palkkarekisteri toimii useilla tietokannoilla. Vaikka palkanlaskentaohjelma on parametriohjattu, joudutaan siihen toimittamaan vuosittain versiopäivityksiä, jotta se täyttää lakien ja asetusten sekä työehtosopimusten määräykset.

Asiakkailla on useita kymmeniä erilaisia konfiguraatioita. Versiopäivitysten jakelu koetaan ongelmaksi. On tapahtunut sellaisia virheitä, että asiakkaille on toimitettu väärä päivitysversio.

Tavoitteeksi asetettiin sellaisen ohjelmiston luominen, jossa selvitään yhdellä päivitysversiolla versioiden jakelun yhteydessä ja jossa asiakkaan ympäristössä olevien konfigurointiparametrien avulla voidaan ottaa asiakasspesifiset piirteet käyttöön.

Suunnitellaan ohjelmisto käyttäen uudelleenkäytettäviä komponentteja. Koska ongelmia aiheuttavat erilaiset käyttöliittymät, tietokannat ja perinnejärjestelmät, suunnitellaan komponentit niin, että ohjelmisto voidaan mukauttaa asiakkaan ympäristöön.

7.2.2 Muunneltavuusanalyysi

Käyttöliittymäkomponentit sisältävät kaikki mahdolliset käyttötapaukset. Otettaessa komponenttia käyttöön luetaan konfiguraatiotiedoista valittava vaihtoehto ja käyttöliit-

tymä mukautuu valinnan mukaisesti. Tällä menettelyllä kaikille asiakkaille voidaan toimittaa sama komponentti. Asiakkaan käyttöympäristö konfiguroidaan asennuksen yhteydessä.

Palkanlaskenta-ohjelmiston palkkarekisteriä varten luodaan jokaista vaihtoehtoista tietokantaa varten yhdyskäytävä, joka hoitaa yhteyttä tietokantaan. Muille kuin tuetuille kannoille rakennetaan asiakaskohtaiset adapterit. Hajautettu tietokantakomponentti (DBMS) tarjoaa rajapintana metodit, jotka tulostavat kyselyt XML-koodattuna yhdyskäytävää tai adapteria varten.

Järjestelmä tuottaa aineiston ulkopuolisia järjestelmiä varten XML-koodattuna. Jokaista erilaista järjestelmää varten luodaan oma kääre, jonka avulla voidaan hoitaa kommunikaatio järjestelmien kesken. Yhteys on tapahtumaohjattu. Palkanlaskennan tapahtumankäsittelijä kutsuu perinnejärjestelmää ja toimittaa järjestelmään kääreen muokkaamana metodi- tai proseduurikutsun, joka toteuttaa halutun toiminnon esimerkiksi kirjaa palkkatapahtuman kirjanpitoon. Kääre otetaan käyttöön alustettaessa komponenttia. Yhdyskäytävä-komponentti (gateway) kommunikoi kääreen kanssa tarvittavan toiminnallisuuden aikaansaamiseksi yhteistyötä tekevän järjestelmän kanssa. Tarvittavia toimenpiteitä ovat esimerkiksi kirjanpitoon tehtävät palkkakirjaukset, kassaan tehtävät maksumääräykset kuten palkanmaksu-, sotu-tilitys-, vakuutusmaksu- ja ay-maksumääräykset.

Tässä esimerkissä kuvataan yksinkertaistaen palkanlaskentajärjestelmän suunnitteluperiaatteita. Kuvataan järjestelmän uudelleenkäytettävät komponenttisysteemit sekä järjestelmän tarjoama muunneltavuus siihen liittyviä järjestelmiä varten. Järjestelmän rakennetta kuvataan korkealla abstraktiotasolla arkkitehtuurityylien ja –mallien avulla. Joitakin yksityiskohtia selvennetään suunnittelumallien avulla, mutta muutoin järjestelmän rakenne ei ole tutkiskelun kohteena.

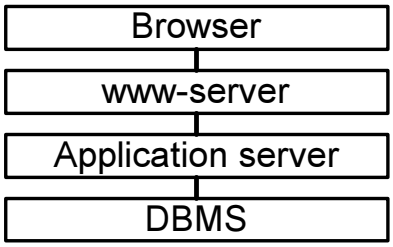
Asiakassovellukset kootaan käyttäen systeemitason komponenttia ja lisäämällä tarvittava asiakaspesifinen koodi adapterien muodossa. Komponenttimääritys ei ota kantaa käytettyyn väliohjelmistoon tai ohjelmointikieleen, vaan valmistaja suorittaa tarvittavan ohjelmoinnin ja asiakasohjelmistopakettien kokoamisen. Suunnittelussa käytetään olioparadigmaa.

Palkanlaskentajärjestelmä on riippumaton alustasta, väliohjelmistosta, tietokannanhallintajärjestelmästä sekä käyttöliittymästä. Ajatuksena on, että ohjelmisto koostetaan

asiakaskohtaisesti vastaamaan asiakkaan tarpeita. Tuotteen kilpailukyky perustuu komponenttien tarjoamaan muunneltavuuteen sekä siihen, että kaikille asiakkaille voidaan asiakaspäivitysten yhteydessä toimittaa sama päivitys.

7.2.3 Arkkitehtuuri

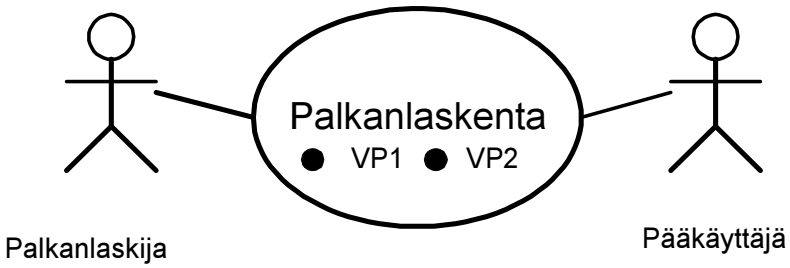
Järjestelmä rakentuu järjestelmätason komponenttista, josta asennusohjelman avulla mukautetaan asiakasohjelma. Palkanlaskentatehtävät on jaoteltu liiketoimintatason komponentteihin. Järjestelmän viitearkkitehtuuri noudattaa monikerrosarkkitehtuuria. Käyttöliittymä voi olla joko selainpohjainen tai käyttöliittymäkomponenttina (Kuva 50).



Kuva 50. Viitearkkitehtuuri

7.2.4 Käyttötapaukset

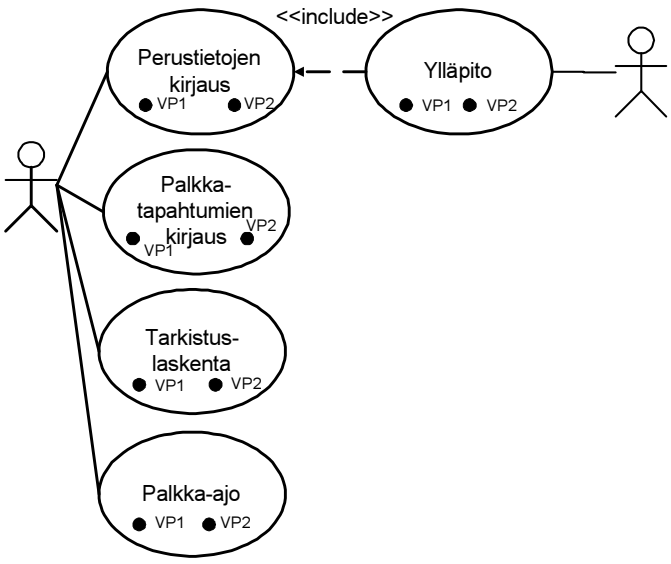
Järjestelmän pääkäyttötapaus on kuvattu kuvassa 51. Järjestelmän systeemitason komponentti tarjoaa muunneltavuutta: käyttöliittymää (VP1) sekä asiakkaan tietokannanhallintajärjestelmää varten on olemassa variaatiopiste (VP2). Muunneltavuutta kirjanpitoa ja kustannuslaskentaa, tilityksiä ja palkanmaksua varten tarjotaan erillisten kääreiden tai adaptereiden avulla.



Kuva 51. Palkanlaskentajärjestelmän pääkäyttötapaus

Järjestelmän pääkäyttötapaus jakaantuu alisysteemeihin (Kuva 52). Alisysteemit koostuvat seuraavasti:

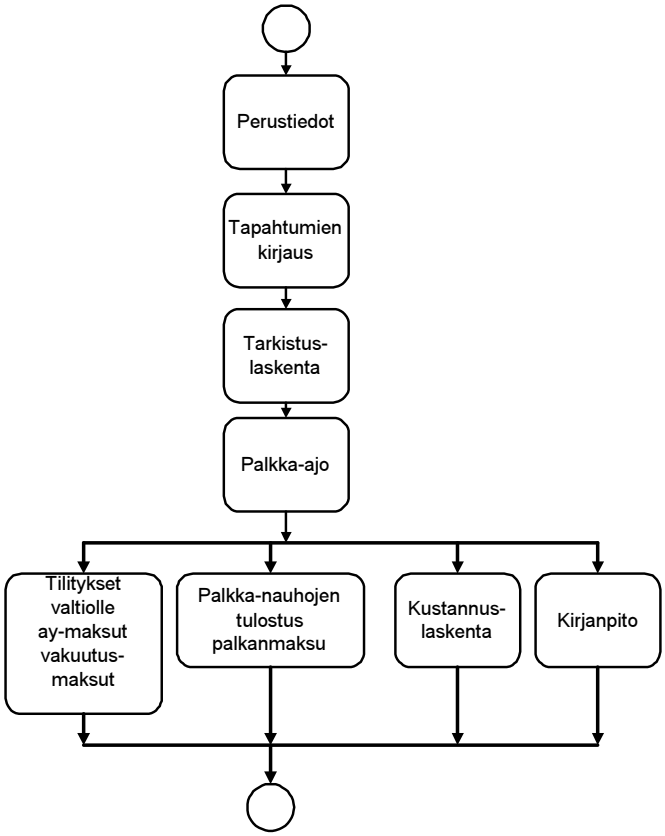
- Ylläpitokomponentin avulla ylläpidetään järjestelmän yleisiä toimintoja ja yleisiä järjestelmäkohtaisia tietoja. Päivittää palkkarekisteriä.
- Perustietojen kirjauskomponentin tarkoituksena on suorittaa henkilötietojen ylläpitotehtäviä. Tällaisiä ovat esimerkiksi verotaulukot, maksupäivärekisteri, käyttöoikeustiedot kuten käyttäjätunnukset sekä tarvittavat kirjanpito ja maksatustiedot. Päivittää palkkarekisteriä.
- Palkkatapahtumien kirjaus –komponentin avulla kirjataan palkanmaksutapahtuman palkanmaksutiedot. Kirjattavia tapahtumia ovat esimerkiksi tuntiilmoitukset, matkalaskut, erilaiset työkalukorvaukset ja muut erikoistiedot esimerkiksi peruspalkasta tapahtuvat poikkeamat. Kirjauksen tehtävänä on luoda tapahtumaluettelo palkanmaksua varten.
- Tarkistuslaskenta suorittaa palkka-ajon koeajon, joka toimii tarkistuslistana ja jonka pohjalta voidaan tehdä muutoksia palkka-ajoa varten.
- Palkka-ajo toteuttaa palkanmaksun ja laatii tapahtumalistan pohjalta maksatusluettelon, jonka avulla tulostetaan mm. palkkapussit. Edelleen laaditaan kirjanpitoon ja kustannuslaskentaan tarvittava kirjausaineisto. Kassanhoitoa varten laaditaan maksatustiedostot valtiolle, ay-liitoille, vakuutusyhtiöille sekä työntekijöiden maksatustapahtumista maksatustiedosto pankkia varten. Maksatustiedostot ovat xml-muotoisia tiedostoja.



Kuva 52. Palkanlaskentajärjestelmän käyttötapaukset

7.2.5 Aktiviteettikaavio

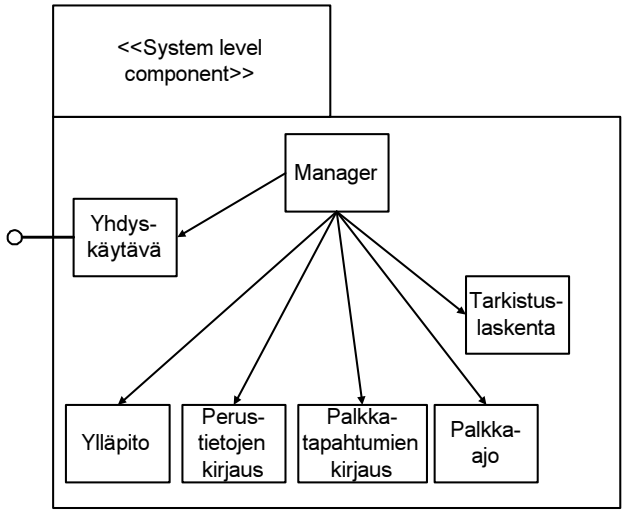
Järjestelmän toimintaa kuvaa aktiviteettikaavio (Kuva 53). Kaaviossa perustietojen-kirjaus on kuvattu samaan kaavioon, vaikka kyseessä on erillinen tapahtuma. Tapahtumaketju etenee järjestyksessä kaavion mukaisesti ja valmis palkka-ajo voidaan perua palauttamalla palkka-ajon yhteydessä otettu varmuuskopio.



Kuva 53. Palkanlaskennan aktiviteettikaavio

7.2.6 Komponentit

Käydään lävitse käyttötapausten komponentit ja joitakin niihin liittyviä periaatteita. Jokaisesta käyttötapauksesta luodaan liiketoimintakomponentti (komponenttisysteemi). Järjestelmä sisältää viisi uudelleenkäytettävää komponenttisysteemiä. Asiakassovellukset pakataan systeemitason komponentiksi, johon liitetään asiakaskohtainen koodi esimerkiksi tarvittavat yhteydet muihin järjestelmiin. Järjestelmätason komponentti on kuvattu kuvassa 54, jossa Yhdyskäytävä-komponentti hoitaa kaikki yhteydet ulkopuolisiin järjestelmiin ja Manager-komponentti vastaa järjestelmän toiminnasta.



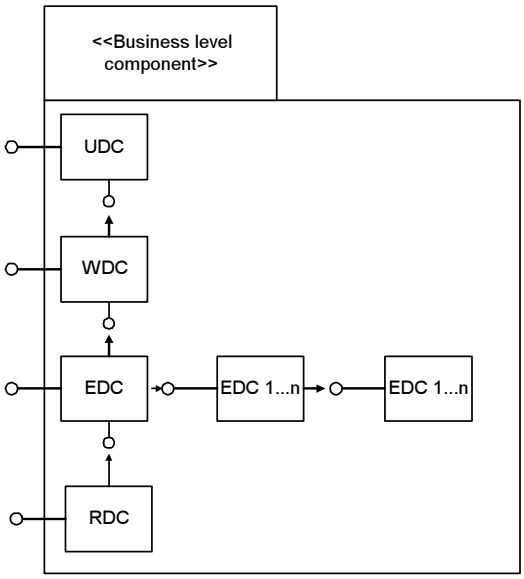
Kuva 54. Järjestelmätason komponentti ja liiketoimintakomponentit (notaatio mukaellen [HeS99])

Tarkennetaan kuvassa 54 esitettyjen liiketoimintakomponenttien tehtäviä. Järjestelmän tarjoamat liittymät muihin tietojärjestelmiin tapahtuvat ainoastaan yhdyskäytävän kautta. Esimerkiksi palkka-ajon yhteyksiä palkkarekisteriin ylläpidetään yhdyskäytävän kautta. Yhdyskäytävän vastuulla on tällöin yhteyden luominen, ylläpitäminen ja yhteyden katkaiseminen. Yhteydet muihin järjestelmiin hoidetaan niin, että yhdyskäytävä on vastuussa yhteystyötä tekevän ohjelmiston kanssa kommunikoinnista siten, että yhdyskäytävä alustaa kääreen, hoitaa viestinnän kääreen kanssa ja päättää kommunikoinnin.

Manager-komponentti ylläpitää järjestelmän muiden komponenttien tilaa. Tämä komponentti tarjoaa käyttäjälle yhdyskäytävän järjestelmän palveluihin. Manager alustaa muut komponentit ohjelmiston asennuksen yhteydessä asetettujen parametrien avulla. Manager-komponentilla on käyttöliittymä ja se tarjoaa päävalikon käyttäjille.

Muut liiketoimintakomponentit suorittavat varsinaiset palkkahallinnon tehtävät, jotka on kuvattu aktiviteettikaavion yhteydessä ja joista on selostus kohdassa 7.2.5.

Liiketoimintakomponentit koostuvat hajautetuista komponenteista, jotka on rakennettu kuvassa 55 esitettyjen periaatteiden mukaisesti. Komponentin arkkitehtuuri noudattaa kerrosarkkitehtuuria.



Kuva 55. Liiketoimintakomponentti

Kaikki käyttöliittymät on rakennettu MVC-mallin mukaisesti. Komponenttiin on rakennettu näkymään vaihtoehdot valmiiksi käyttöliittymätyyppejä varten (Windows, Motif sekä selain-liittymä). Tällöin asiakasohjelmaan installoidaan haluttu liittymä toteuttamalla abstrakti rajapinta näkymän ja kontrollerin toteuttamiseksi. Näkymän valinta on toteutettu Strategia-suunnittelumallin avulla, jolloin valitaan ainoastaan tyyppi. Kommunikaatio on oletusarvoisesti toteutettu väliohjelmiston tarjoamana palveluna esim. RPC tai RMI-tyyppisesti riippuen valmistajasta. Käyttäjän ja järjestelmän yhteistoiminta on kysely/vastaus-tyyppistä, joten tehokkuus ei ole määräävin tekijä vaan käytettävyys. Käyttöliittymä on toteutettu thin-client –periaatteella, jolloin myös työpöytäsovellus-tapauksessa käyttöliittymäkomponentti on sovelluspalvelimella ja asiakkaan client-sovellus kutsuu käyttöliittymää etäoliona.

Liiketoimintakerroksen komponentit (EDC) toteuttavat Putki- ja suodatin –mallia palkanlaskentaprosessin ja tarkistuslaskennassa, jolloin käyttäjä antaa tarvittavat ohjaustiedot ja käynnistää prosessin. Komponenttien sisällä palkanlaskentatapahtumat toteutetaan suoraviivaisesti putki- ja suodatin –mallin avulla. Vikasietoisuus toteutetaan niin, että koko palkanlaskenta tapahtumasta luodaan yksi transaktio, joka joko toteutetaan kokonaan tai perutaan kokonaan. Palkanlaskentaa kontrolloi kontrolli-komponentti, jonka tehtävänä on todeta jokaisen vaiheen onnistuminen ja tarvittaessa keskeyttää ajo. Lopuksi kontrolli-komponentti on yhteydessä muihin järjestelmiin ja siirtää laskennan tuloksen kirjanpitoon, kustannuslaskentaa ja suorittaa tilitykset.

Palkkarekisterin tietokantaliityntäkomponentti (RDC) on suunniteltu noudattaen Strategia-mallia, jossa asennuksen yhteydessä annetun parametrin perusteella valitaan soveltuva kommunikaatio tietokannanhallintajärjestelmään.

7.2.7 yhteenveto mallista

Käyttämällä uudelleenkäytettäviä komponentteja voidaan asiakaskohtaiset toimitukset koostaa nopeasti. Parasta mallissa on se, että se ei rajoita toteutusta. Käyttöliittymä voidaan suunnitella halutulla tavalla, tietokannaksi voidaan valita mikä tahansa ja järjestelmä voidaan liittää vapaasti perinnejärjestelmiin. Järjestelmältä vaaditaan suorituskykyä ainoastaan palkanlaskentavaiheessa. Perinteisesti eräajona suoritettu palkanlaskenta ja sen kesto on ollut järjestelmän arvioinnin eräitä keskeisimpiä mittareita.

Esimerkki ei ota kantaa siihen kuinka tuotanto organisoidaan, jotta saavutettaisiin tehokas tuotanto (time-to-market). Tehokkuus paranee uudelleenkäytön ja selkeämmän ohjelmistorakenteen avulla. Komponenttituotannon aloittaminen kysyy resursseja ja vaatii pitkäjänteisyyttä. Hyvä tuotteen laatu ei synny pelkästään komponentteja käyttäen, vaan vaaditaan asianmukainen jatkuva laatukontrolli laadunarviointi- ja laadunparantamistoimintoineen.

Jotta ohjelma voisi kilpailla pakettiohjelmien kanssa, on siitä luotava eri versioita erilaisille alustoille. Tavoitteen tällöin on, että uudelleenkäytön avulla tuotteen hinta saataisiin kilpailukykyisemmäksi. On rakennettava myös liittymiä asiakkaiden omia ohjelmia varten.

7.3 Arkkitehtuurin parantaminen kilpailukeinona

Herzumin [HeS99], Jacobsenin [JGP97] sekä Boschin [Bos00] esittämät mallit ja esimerkit mallintavat suurten organisaatioiden pyrkimystä parantaa tuotantoprosesseja käyttämällä arkkitehtuuri- ja komponenttisuuntautunutta ajattelutapaa. Herzum mallintaa käsitteellä *komponenttitehdas (business component factory)* prosessia, jossa tuotetaan laadukkaita liiketoimintaohjelmistoja ja joka käsittää koko tuotteen elinkaaren [HeS99 s. 61]. Komponenttitehtaan pystyttäminen on luontevaa suurille yrityksille, joilla on paljon erilaisia tuotteita ja joilla on käytössään sekä taloudellisia että henkilöresursseja.

Valtaosa yrityksistä on pieniä alle 300 hengen ohjelmistotaloja, joille komponenttitehtaan pystyttäminen voi olla ongelma. Pienissä yrityksissä rajoituksena on resurssien vähäisyys. Jos ohjelmistotyöntekijän tehokkuus on esimerkiksi 80% eli hänen työajastaan kohdistuu kyseinen osuus projekteille ja loput on yleistä yhteistä toimintaa, kuten esimerkiksi koulutusta ja toimistorutiineja, on vaikea nähdä mahdollisuuksia komponenttitehtaan käynnistämiseen. Näissä yrityksissä ongelmien ratkaisutapa on usein käytännönläheinen (low-level). Yritykset käyttävät komponenttiteknologiaa rakentaessa hajautettuja järjestelmiä, mutta lähestymistapa on usein hyvin rakentamiskeskeinen. On hankalaa nähdä kuinka prosessi, jonka sisään ajaminen voi kestää useitakin vuosia, voisi onnistua. Ratkaisuna voi olla, että komponenttitehdas otetaan osittain käyttöön. Yleensä yrityksissä on laatujärjestelmä, joten periaatteessa yrityksillä on tavoitteena kypsempi tuotantotapa (CMM). Komponenttien rakentaminen osataan, oliopohjaisten järjestelmien suunnittelu osataan ja projektien toteuttamisessa on saavutettu sellainen varmuus, että opittuja ja sovittuja käytäntöjä käyttäen asiakasprojekti saadaan valmiiksi sovitussa ajassa. Ongelmana on ohjelmistosuunnittelun taso. Osataan suunnitella järjestelmiä, mutta ei järjestelmätason komponentteja. Osataan integroida järjestelmiä, mutta liiketoimintakomponenttien hallintakykyä ei ole. Tiukka käytännönläheisyys aiheuttaa sen, että tuotannossa ei saavuteta maksimaalista tehokkuutta ja laatua. Koska näkökulma on käytännön läheinen, organisaatio ei suorita kunnollista arkkitehtuurisuunnittelua ja eikä määrittele systeemitason vaatimuksia muunneltavuudelle. Tästä seuraa, että koska näkökulma keskittyy rakentamiseen, ratkaisut ovat liian teknisiä ja usein niitä voidaan soveltaa vain rakennettuun kohdealueeseen. Pienen yrityksen ratkaisu komponenttitehtaan pystyttämiseen voisi olla esimerkiksi *mallioppimisen* käyttäminen. Luodaan jostain keskeisestä projektista malli, jota voidaan soveltaa myös muissa hankkeissa. Mallin toteuttamiseksi ostetaan tarvittaessa tietotaitoa. Kun malli on saatu toteutettua alusta loppuun, voidaan tuloksia ottaa käyttöön. Etuna malliajattelussa on se, että sen avulla voidaan pieninkin resurssein parantaa toiminnan laatua. Mallin avulla ei toki voida luoda komponenttitehdasta, mutta sen avulla voidaan joitakin piirteitä ottaa käyttöön. Organisaatio voi muokata toimintaansa, ajattelussa voidaan painottaa enemmän analyysiä ja komponentti- ja arkkitehtuurikeskeistä otetta.

Pienet yritykset kilpailleessaan suurten kanssa menestyvät usein kohtuullisen hyvin, koske ne toimivat joustavasti, niillä on pienemmät yleiskulut ja ne lähtevät asiakaslähtöisestä ongelmanratkaisusta sekä niillä on selkeästi rajattu kohdealue. Pienillä yrityksil-

lä on ongelmana saavuttaa tasalaatuisuus, resurssienhallinta ja tehokkuus. Samoin henkilöstön vaihtuvuus voi olla todella suuri ongelma. Pienten yritysten tulisi löytää tai rakentaa itse omiin tarpeisiin soveltuvia komponentti- ja sovelluskehyksiä ja hyödyntää teknologian tarjoamat mahdollisuudet sovelluskehittimien käytössä.

Suurten yritysten vahvuutena on niiden kyky panostaa tuotekehitykseen ja suunnitteluun, jonka ansiosta ne pystyvät tehokkaampaan toimintaan. Komponenttitehtaan ansiosta tuotannossa voidaan käyttää tehokkaasti uudelleenkäytettäviä elementtejä. Suuret yritykset keskittyvät myös laajempiin toimialueisiin, joista on helpompi muodostaa tuoteperheitä ja soveltaa tuotelinja-arkkitehtuuria.

Tulevaisuudessa näyttäisi käyvän niin, että markkinoille tulee yhä uusia komponentteihin pohjautuvia muunneltavia ratkaisuja, jotka ovat pakettiohjelmistojen kaltaisia. Komponentteja käsitellään, yhdistellään ja mukautetaan komponentti- ja sovelluskehysten ja sovelluskehittimien avulla. Nämä komponentit ovat rakeisuudeltaan joko kohdealuespesifisiä tai yleisiä sekä joko systeemitason komponentteja tai liiketoimintakomponentteja. Pienten yritysten tehtäväksi näyttäisi jäävän näiden komponenttien mukauttaminen asiakkaiden tarpeisiin sekä niihin liittyvien konsultointitehtävien, kuten asennus-, ylläpito-, koulutus- ja järjestelmäkehitystehtävien hoitaminen. Voi olla hyvinkin, että markkinoiden kannalta hajautettu komponentti on liian pieni kaupattavaksi (paitsi yleisten esim. GUI-komponenttien muodostamien komponenttikirjastojen osana) ja markkinat syntyvät kohdealuespesifisten liiketoimintakomponenttien ympärille. Teknologiaa käytettäneen siten, että liiketoimintakomponentteja yhdistellään ja muokataan kehysten ja kehittimien avulla.

Mitä yritys voi saavuttaa, jos se käyttää arkkitehtuurin parantamista kilpailutekijänä? Mitä hyötyjä yritykselle on panostaa arkkitehtuurisuunnitteluun, arkkitehtuurin arvioimiseen? Miksi yrityksen kannattaa kehittää organisaatiotaan niin, ett se vastaa paremmin arkkitehtuuripohjaisen suunnitteluprosessin haasteita?

Tavoitteina olivat parannukset tuotannon läpimenoajassa (time-to-market), hinnassa (price) ja laadussa (quality) Näistä tavoitteista ensiksi mainittu saavutetaan organisoimalla tuotantoprosessi modernilla tavalla (2.8.2). Hintakilpailukyky saavutetaan uudelleenkäytettävien komponenttien avulla. Tuotelinjan tai komponenttikehysten käyttö ovat esimerkiksi hyviä keinoja lisätä uudelleenkäyttöä. Jos aiottuun tarkoitukseen on

saatavissa sopiva sovelluskehitin, kannattaa sitä harkita. Arkkitehtuurin arvioinnin avulla löydetään ohjelmiston rakenteesta ne ongelmapaikat, jotka voivat aiheuttaa pullonkauloja ohjelmiston tuotantokäytössä. Arvioinnin avulla voidaan ennustaa esitettyjen laatuvaatimusten toteutumista ja suorittaa tarvittavia muutoksia arkkitehtuuriin.

Tulevaisuus on kehysten ja nimenomaan komponentti- ja sovelluskehysten. Jossain vaiheessa päästäneen siihen, että ohjelmointikielten käytöstä siirrytään kokonaan sovelluskehittimien käyttöön. Tällöin sovelluskehittimet tarjoavat tarvittavan tuen matalan tason rutiinien hoitamiseen. Liiketoimintasovellustenkehittäjä kokoaa sovelluksen komponenteista ja laatii siihen sovelluskehittimen avulla tarpeelliset asiakaskohtaiset muutokset.

Tässä tutkielmassa on tarkasteltu ohjelmistoarkkitehtuureita ja yrityksen tuotantoprosessia laajasta näkökulmasta. Tarkoituksena on ollut tuoda esille arkkitehtuurikeskeisen tarkastelutavan vaatimukset yrityksen toimintaan. Arkkitehtuuri vaikuttaa lähes kaikkeen yrityksen toiminnassa. Vaikutuksen ulottuvat organisaation rakenteesta jokapäiväiseen toimintaan. Tavoitteena on kyky tuottaa teollisesti komponentteja asiakkaiden tarpeisiin. Herzumin malli komponenttitehtaasta sopii lähes sellaisenaan suurten yritysten tarpeisiin. Tarvetta on sen sijaan luoda menetelmä komponenttitehtaan luomiseksi pienissä yrityksissä. Mallin tulisi olla riittävän yleinen, jotta sitä voitaisiin soveltaa ja kaupallistaa laajemmaltikin. Mallin tulisi olla sellainen, että se mahdollistaa mallioppimisen ja sitä voitaisiin soveltaa yrityksissä konsulttivetoisesti, koska pienillä yrityksillä ei ole tarpeellista tietotaitoa komponenttitehtaan pystyttämiseksi.

Integrointikustannukset muodostavat valtaosan yritysten tietotekniikkakustannuksista. Järjestelmien välinen integraatio on tärkeää ja siihen tarvitaan parempia välineitä. Ongelmana on usein se, että järjestelmät ovat liian erilaisia, jotta voitaisiin luoda täsmällinen ja monikäyttöinen tekniikkavetoinen ratkaisu. Tarvetta on yleiselle tekniikkariippumattomalle määrittelylle, jonka avulla voidaan integroida erilaisia järjestelmiä toisiinsa. Mallin tulisi pohjautua arkkitehtuuri- ja suunnittelumalleihin ja sen tulisi muodostaa riittävän joustava komponenttikehys siten, että sitä voidaan soveltaa käytäntöön. Tavoitteena on, että yritykset kaupallistavat määrittelyn luomalla siitä tekniikkariippuvia määrittelyjä.

Lähteet

[Ber95] Alfs Berztiss: *Software Methods for Business Reengineering*, Springer, 1995.

[BoB01] Bosch Jan, Bengtsson PerOlof: Assessing Optimal Software Architecture, *Fifth European Conference on Software Maintenance and Reengineering*, p.168, Maaliskuu, 2001.

[Bos00] Bosch Jan: *Design and Use of Software Architectures*, Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000.

[BuR96] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M.: *Pattern-oriented Software Architecture a System of Patterns Volume1*, Wiley, 1996.

[ChD01] Cheesman, J. and Daniels J.: *UML Components a simple process for specifying component-based software*, Addison-Wesley, 2001.

[CleF02] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nod, R., Stafford, J.: *Documenting Software Architectures*, 2002.

[DoN02] Dobrica, L., Niemelä, E.: A Survey on Software Architecture Analysis Methods, *IEEE Transaction on Software Engineering*, VOL. 28, no. 7, heinäkuu 2002.

[EDM98] Eman El Khaled, Drouin Jean-Normand and Melo Walce´lio: *SPICE the Theory and Practice of Software Improvement and Capability Determination,* IEE Computer Society Press, 1998

[FaJ00] Fayad, M., Johnson, R.: *Domain-Specific Application Frameworks*, Wiley, 2000.

[For02] Forsell, M.: *Improving Component Reuse in Software Development*, Jyväskylän yliopisto, 2002.

[Fow99] Fowler, M.: UML Distilled: *A Brief Guide to the Standard Object Modeling Language (2nd Edition)*, Addison-Wesley, 1999.

[GaH95] Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides: *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley (IT Press), 1995.

[Gra01] Grand, M.: *JAVA Enterprise Design Patterns Vol 3*, Wiley, 2001.

[GYH00] Ginbayashi J., Yamamoto R., Hashimoto K.: Business Component

Framework and Modelling Method for Component-Based Application Architecture, *Fourth International Enterprise Distributed Object Computing Conference (EDOC'00)* p. 184, IEEE, 2000.

[HaM01] Haikala, Märijärvi: Ohjelmistotuotanto, Satku, 2001.

[HeS99] Herzum P. Sims O.: *Business Component Factory*, Wiley, 1999.

[Hud03] Hudson, P.: Linux.NET, *Linux Format/ LinuxPro*, June 2003, 2003.

[Imm03] Immonen, M.: *Suunnittelumallit –erikoistyö*, Kuopion yliopisto, 2003.

[Imm03a] Immonen, M.: *Käytettävyyden suunnittelu ja rakentaminen ohjelmistoprosessissa*, Kuopion yliopisto, 2003.

[Jal99] Jalote P. : *CMM in Practice : Processes for executing software project at Infosys*, Addison-Wesley, 1999.

[JGP97] Jacobson I., Griss M., Johnson P.: *Software Reuse (Architecture, Process, and Organisation for business Success)*, Addison-Wesley, 1997.

[Jun03] Junkkari, M.: Asiakkaasta tuli ohjelmistoalan kuningas, *Helsingin Sanomat 25.5.2003* (2003), E1-E2.

[Jär02] Järvinen, J.: *Hajautetut verkkopalvelut*, Docendo, 2002.

[Kos00] Koskimies, K.: *Oliokirja*, Satku, 2000.

[Kru95] Kruchten, P.B.: The 4+1 View Model Architecture, *IEEE Software,* November 42-50, pp. 131-183, 1995.

[Lim95] Lim, W.: *Managing Software Reuse*, Prentice Hall, 1998.

[Lin00] Linthicum, D.: *Enterprise Application Integration*, Addison-Wesley, 2000.

[LST78] Linetz B., Swanson E., Tompkins G.: Characteristics of Application Software Maintanance, *Communication of ACM*, vol 24, no., June 1995.

[MoR97] Mowbray T., Ruh W.: *Inside CORBA distributed object standards and applications*, Addison-Wesley, 1997.

[Mul89] Mullender, S.: *Distributed Systems*, ACM Press, 1989.

[Myk00] Mykkänen, J.: Komponentti-FixIT, *Terveydenhuollon komponenttipohjainen sovellustuotanto - toiminnallisuus, arkkitehtuuri, siirytmästrategiat ja välineet*, Kuopion Yliopisto, 2000.

[NKV01] Niskanen, P., Kontio, M., Vierimaa, K.: *Inside Enterprise Java*, IT Press, 2001.

[PSM94] Prieto-Dfaz R. , Schäfer W., Matsumoto M.: Software Reusability, *Ellis Horwood*, 1-16, 1994.

[Puu03] Puumalainen, K.: *Pro gradu-tutkielma, Tietojärjestelmien integrointi*

yritysten välisen liiketoiminnan kehittäjänä, Kuopion yliopisto, 2003.

[Ran01] Rannanheimo, J.: *Pro gradu-tutkielma: Sovelluspalvelimet komponentti-infrastruktuurissa*, Kuopion yliopisto, 2001.

[Rob99] Robertson S., Robertson J.: *Mastering the Requirements Process*, Addison-Wesley, 1999.

[Roy98] Royce W.: *Software Project Management*, Addison-Wesley, 1998

[ScR01] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: *Pattern-oriented Software Architecture a System of Patterns Volume2*, 2001.

[ShD95] Shaw, M., DeLine, R., Klein, D., Ross, T.,Young, D., Zelesnik, G.: Abstractions for Software Architecture and Tools to Support Them. *IEEE Transaction on Software Engineering*, May 1995.

[Smo03] Smolander, Kari: *On the Role of Architecture in Systems Development*, Acta Universitatis Lappeenranta, 2003.

[SoM01] Sommerville Ian: *Software Engineering 6th Edition*, Addison-Wesley/ Pearson Education, 2001.

[SoW98] D'Souza Francis, Wills Alan: *Objects, Components and Frameworks with UML*, Addison-Wesley, 1998.

[SzC98] Szyperski, Clements P.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Tan02] Tanenbaum, A., van Steen, M.: *Distributed Systems Principles and Paradigms*, Prentice Hall, 2002.

[Tra95] Tracz W.: *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison-Wesley, 1995.

[www001] http://www.freedownloadscenter.com/Programming/Visual_Basic_Components_A-G/ (21.7.2003).

[www002] http://msdn.microsoft.com/visualc/using/understanding/interop/default.aspx?pull=/library/en-us/dndotnet/html/dotnetperftechs.asp (19.5.2003).

[www003] http://java.sun.com/j2ee/overview2.html (21.7.2003)

[www004] http://www.webservicesarchitect.com/content/articles/samtani04.asp (1.7.2003).

[www005] http://www.cs.wustl.edu/~schmidt/corba-overview.html (20.5.2003.

[www006] http://www.rational.com/corpinfo/practices.jsp (23.5.2003).

[www007] http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/vit/ftp/pdf/PLoP95.pdf (19.3.2003).

[www008] http://www.sei.cmu.edu/ (9.6.2003).

[www009] http://www.byte.com/art/9709/sec5/art1.htm#097cova2 (9.6.2003).

[www010] http://www.tml.hut.fi/Opinnot/Tik-110.551/1997/mqs.htm (6.6.2003).

[www011] http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html (30.5.2003).

[www012] http://martinfowler.com/eaaCatalog/ (30.5.2003).

[www013] http://discuss.santafe.edu/robustness/stories/storyReader$9 (28.8.2003).

[www014] http://www.compuware.com/products/uniface/default.htm (3.9.2003)

[www015] http://www.starsoft.fi/ (2.4.2003)

[www016] http://www.uwasa.fi/samjay/b_admin/projects/atrapo.html#Tutkimuksen (2.4.2003)

[www017] http://www.prointra.net/index.asp (2.4.2003)

[www018] http://www.microsoft.com/resources/practices/ (22.8.2003)

[www019] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html (22.8.2003)

LIITE1. SANASTO JA KÄSITTEET

ACID (Atomicity Consistency Isolation Durability) tarkoittaa transaktion ominaisuuksia. Kaikki transaktiot jakavat nämä ominaisuudet: atomisuus, samanaikaisuus, eristyneisyys ja pitkäikäisyys [Bos00 s. 137]. Tehtävä suoritetaan kokonaan tai ei ollenkaan. Järjestelmä on aina yhtenäisessä tilassa. Eri transaktiot eivät vaikuta toisiinsa. Onnistuneen transaktion tila jää pysyväksi [Ran01].

Arkkitehtuuri (architecture) kuvaa ohjelmiston rakennetta ja sen koostumista komponenteista ja näiden komponenttien välisiä suhteita [Bos00 s. 11].

ADL Arkkitehtuurin kuvauskieli (Architecture Description Language) kuvaa ohjelmiston arkkitehtoonisia elementtejä ja niiden välisiä suhteita [CleF02 s. 469].

Apukomponentti (auxiliary component) avustaa systeemin kehitystä tai on apuna ajon aikana ja osa systeemiä. Esimerkkinä ovat vaikkapa monitorointikomponentti tai tietokantamanageri [Myk00 s. 59], [HeS99 s. 181].

Arkkitehtoninen lähestymistapa (architecture-centric approach) tarkoittaa sitä, että koko kehitysprosessi on arkkitehtuurikeskeistä siten, että jokaisessa kehitysvaiheessa lähtökohtana on arkkitehtoninen lähestymistapa [HeS99 s. 52].

Arkkitehtuurimalli (achitectural pattern) ilmaisee joitain arkkitehtuurin sääntöä, joka määrittelee sen, kuinka systeemi käsittelee jonkin toiminnallisen ominaisuuden [Bos00 s. 38]

Arkkitehtuurityyli (architectural style) on erittely elementti- ja suhdetyypeistä, joilla on yhdessä joukko rajoituksia ja kuvaus, siitä kuinka niitä voidaan käyttää [CleF02 s. 469].

Asianosaiset (stakeholders) ovat hankkeesta kiinnostuneet henkilöt ja organisaatiot [Rob99 s. 939]. Asianosaisia ovat esimerkiksi asiakas, loppukäyttäjä, johtoryhmä, liiketoimintaekspertit, teknologian hyväksikäyttäjät, systeeminkehittäjät, markkinointi, tuotepäälliköt, testaajat ja laadunvarmistus, lakimiehet, käytettävyysekspertit ja alihankkijat [Rob99 s. 358].

Best Practice [www006] määrittelevät kuusi käytäntöä, jotka on havaittu toimiviksi ohjelmistoprojekteissa: iteratiivinen ohjelmistokehitys, vaatimustenhallinta, komponenttiarkkitehtuurien käyttö, visuaalinen mallintaminen, jatkuva laadunarviointi, muutosten hallinta.

COM/DCOM/COM+ on Microsoftin komponenttimalli komponenttipohjaisten ohjelmistojen rakentamiseksi Windows-käyttöjärjestelmäalustalle.

CORBA (Common Object Request Broker Architecture) on OMG:n (Object Management Group) määrittely komponenttimallista, jossa kommunikointi elementtien välillä tapahtuu ORB:n (Object Request Broker) avulla.

DCE (Distributed Computing Environment) Open Groupin määrittelemä ympäristö etämetodikutsujen käsittelemiseksi käytössä Microsoftin käyttöjärjestelmissä. Muita RPC:tä käsitteleviä määritelmiä ovat esimerkiksi SunRPC [Tan02 s. 80].

EJB (Enterprise Java Bean) on Java-kieleen perustuva komponenttiteknologia komponenttien rakentamiseksi ja ajamiseksi sovelluspalvelimella sen EJB-säiliössä.

Elementti (element) on arkkitehtuurinen rakennuspalikka, jolle on luontaista jokin näkökulma tai tyyli (viewtype/style). Elementti voi olla esimerkiksi moduuli tai komponentti [CleF02 s. 470].

Entiteettikomponentit (entity component) esittävät varsinaista liiketoimintatapahtumaa vaikkapa laskua, tilausta, palkanmaksutapahtumaa ja ovat yleensä kohdealueesta riippumattomia [Myk00 s.59], [HeS99 s. 178].

Generointipohjainen lisäysmalli (generator-based model) generoi yleisen kehyksen avulla annettujen syötteiden pohjalta ajokelpoisen komponentin, jossa on yleiset rajapinnat. Etuina tässä mallissa ovat joustavuus ja uudelleenkäytettävyys. Haittana ovat taas alentunut kehityskelpoisuus ja tuotekohtaisten laajennusten monimutkaisuus [Bos00 s. 255].

Hajautettu dokumenttipohjainen systeemi (distributed document-based system) perustuu asiakkaan tekemiin dokumenttikyselyihin. Kyseessä on asiakas/palvelinsovellus, jossa asiakas tekee kyselyn tiettyyn URL-osoitteeseen ja jää odottamaan vastausta. Palvelin suorittaa pyynnön saatuaan sen ja palauttaa tuloksen asiakkaalle. Toimintatapa on asynkroninen ja protokollana on yhteydetön protokolla esimerkiksi HTTP, jossa jokainen pyyntö on oma kokonaisuutensa. Tärkeimmät sovellukset ovat WWW ja Lotus Notes [Tan02 s. 647].

Hajautetut järjestelmät (distributed systems) ovat kokoelma itsenäisiä tietokoneita, jotka näkyvät käyttäjälleen yhtenäisenä järjestelmänä [Tan02 s. 2].

Hajautettu koordinaatiopohjainen systeemi (distributed coordination based system) tarkoittaa sellaista mallia, jossa hajautus on suoritettu komponenttitasolla. Tässä mallissa keskitytään komponenttien toimintojen koordinoimiseen. Hajautetuissa järjestelmissä on piirteitä, joissa toteutetaan tätä tapaa. Hajautettu järjestelmä voidaan nähdä joukkona prosesseja, jossa prosessit suoritetaan periaatteessa muista prosesseista riippumattomasti. Tässä mallissa koordinaatio-osa hoitaa kaiken kommunikaation ja yhteistyön prosessien välillä. Esimerkkinä koordinaatiopohjaisesta systeemistä ovat TIBCO Rendezvous ja Sun Microsystemsin Jini [Tan02 s. 699].

Hajautetut oliopohjaiset systeemit (distributed object-based system) keskittyvät käsittelemään hajautettuja olioita. Olion hajautus on kätketty sen rajapintaan, jota tarjotaan käyttäjälle. Välitason ohjelmien tarkoituksena on tehdä hajautus käyttäjälle näkymättömäksi. Esimerkkeinä tästä ovat CORBA, DCOM ja Java [Tan02 s. 493].

Hajautetut tiedostojärjestelmät (distributed file systems) sallivat useiden prosessien jakaa tiedostoja turvallisesti ja luotettavasti. Hajautettu tiedostojärjestelmä on usein osa hajautettua järjestelmää ja sovelluksia. Tiedostojärjestelmä on riippumaton käyttöjärjestelmästä tai verkkokäyttöjärjestelmästä. Tämä mahdollistaa esimerkiksi sen, että Windows-järjestelmä voi kommunikoida Unix-tiedostopalvelimen kanssa. Näissä järjestelmissä käytetään tiedostojen käsittelyssä joko RPC-menettelyä, jossa tiedosto pysyy pal-

velimella ja päivittyy siellä tai upload/download-menettelyä, jossa tiedosto siirretään asiakkaan koneelle [Tan02 s. 575]. Tiedostojen ja resurssien hallintaan eri koneiden ja eri käyttöjärjestelmien välillä käytetään verkkokäyttöjärjestelmiä esimerkiksi Windows, Novell ja Unix/Linux. Palvelimet voivat erilaisten ohjelmien avulla tarjota resurssejaan muiden käyttöön. Esimerkiksi Linuxin Samba-ohjelman avulla voidaan osia Linuxin tiedostojärjestelmästä tarjota Windows-koneiden käyttöön [Tan02 s. 493].

Hienojakoisessa lisäysmallissa (fine-grained extension model) kehykseen lisätään suuri joukko pieniä joko tuote-spesifisiä tai yleisiä moduuleita. Etuna tässä mallissa on suuri joustavuus ja lisäosien uudelleenkäytettävyys. Haittana on taas monimutkaisuus. [Bos00 s. 253].

Jatkuva integraatio (continuous integration) on mahdollista läpi koko kehitysprosessin. Tämä tarkoittaa sitä, että liiketoimintojen suunnittelija voi kehittää itsenäisesti kyseistä liiketoimintaa ja integroitavuuden ansiosta tehtyjä suunnitteluratkaisuja voidaan testata ympäristössä palautteen saamiseksi (feed-back) [HeS99 s.52].

Idoomi (idiom) on matalan tason ohjelmistoriippuvainen malli. Idoomi kuvaa kuinka jokin piirre komponentissa tai niiden välillä toteutetaan kieliriippuvalla tavalla [BuR96 s. 14].

IPC (interprocess communication) tarkoittaa samassa koneessa tapahtuvaa eri prosessien välistä tietojenvaihtoa. Toteutetaan esimerkiksi tiedostojen, putkien (pipes), liitoskohdan (socket), opastimen (semaphores) tai jaetun muistin avulla (shared memory) [www003].

Iteratiivinen luonne (iterative nature) tarkoittaa sitä, että koko kehitysprosessi, joka kykenee kehittämään ja toteuttamaan itsenäisesti yksittäisiä komponentteja, on luonteeltaan iteratiivinen ja inkrementaalinen [HeS99 s. 52]

Kehys, Ohjelmistokehys (framework) on luokkien tai rajapintojen kokoelma, joka toteuttaa tietyn ohjelmistoperheen perusarkkitehtuurin. Sovellus tai komponentti tehdään kehyksestä *erikoistamalla erikoistamisrajapinnan* avulla. Jos kehyksen erikoistaminen luo uuden sovelluksen, on kyseessä *sovelluskehys (application framework)*. Jos taas kehyksen erikoistamisesta on tuloksena komponentti, on kyseessä *kehyskomponentti (framework component)* [Kos00 s. 260]. *Sovelluskehitin (application generator)* on järjestelmä, joka generoi korkean tason sovelluspecifikaatiosta ajokelpoisen sovelluksen (esim. Jbuilder) [Kos00 s. 262].

Komponentti (component) on kooste, jolla rajapinnat ja yksiselitteiset riippuvuudet. Komponentti voidaan jakaa itsenäisesti ja sitä voivat kolmannet osapuolet käyttää koostaessaan omia ohjelmistojaan [Bos00 s. 214]. Komponentti on joukko samanaikaisesti toimitettavia atomisia komponentteja. Atominen komponentti (atomic component) on moduuli tai joukko resursseja [SzC98 s. 276].

Komponenttien autonomia (focus on autonomy) tarkoittaa sitä, että kehitysprosessissa kiinnitetään huomio eri rakeisuuden omaavien komponenttien autonomisuuteen [HeS99 s. 52]. Tämä tarkoittaa sitä, että komponentin tulisi olla itsenäinen, riippumaton koko kehityskaatensa ajan. Riippuvuudet syntyvät vain rajapinnoista ja esi- ja jälkiehdoista.

Komponenttikehys (component framework) on ohjelmistoyksikkö, joka tukee komponenttien kokoamista tietyn standardin mukaisesti ja sallii näiden komponenttien ilmentymien asentamisen (plug) komponenttikehykseen. Komponenttikehys asettaa ympäristön rajoitukset komponentin ilmentymille ja säätelee komponentti-instanssien vuorovaikutusta [SzC98 s. 280].

Komponenttisuuntautunut ohjelmointi (COP component-oriented programming) tukee polymorfismia, modulaarista kapselointia (modular encapsulation), myöhäistä sidontaa (late binding) ja kutsumista (loading) ja turvallisuutta (type and module safety) (Szyperski,1995) [SzC98 s. 306]. **Komponenttisuuntautunut lähestymistapa (component-centric approach)** lähtee siitä, että kehityprosessi on komponenttikeskeistä siten, että se keskittyy käsittelemään eri karkeustasolla (granularity) olevia komponentteja ohjelmistokehityksen eri vaiheissa [HeS99 s. 519].

Laatuattribuuttisuuntautunut ohjelmistoarkkitehtuurisuunnittelumenetelmä (quality attribute-oriented software architecture design method) koostuu kahdesta iteratiivisesta prosessista. Sisimmässä iteraatiossa suunnitellaan, arvioidaan ohjelmistoarkkitehtuuri a muunnetaan arkkitehtuuri vastaamaan laatuvaatimuksia. Uloin iteraatio koostuu vaatimusten valinnasta ja niiden mukaan ottamisesta arkkitehtuurisuunnitteluun [Bos00 s.25].

Light-weight RPC tarkoittaa sitä, että omassa osoiteavaruudessa tapahtuva kommunikointi IPC voidaan hoitaa lightweight RPC –menettelyn avulla, jos käyttöjärjestelmä tukee sitä [Tan02 s. 78].

Liiketoimintasovelluskehys (enterprise application framework) kattaa laajan sovelluksen kohdealueen, kuten telekommunikaatio tai pankkitoiminta. Liiketoimintasovelluskehys sisältää sekä väliohjelmistointegraatio- että infrastruktuurikehykset omina alapuolella olevina tasoinaan [Faj00 s. 616].

Mallisysteemi (pattern systems) on kokoelma ohjelmistoarkkitehtuurimalleja, jotka yhdessä määräävät ohjelmistokehityksen suuntaviivat implementointia, yhdistelyä ja käyttöä varten [BuR96 s. 361]. *Mallijärjestelmä (pattern language)* on kokoelma järjestettyjä malleja niin, sen avulla voidaan systemaattisesti suunnitella jokin kokonaisuus.

Metodisuuntautunut integrointimalli tapahtuu kytkemällä sovellukset yhteen siten, että jaettavia metodeja voidaan käyttää integroinnissa. Sovelluskehysten avulla voidaan suorittaa aliohjelmistojen integrointia toimivaksi kokonaisuudeksi. Metodisuuntautunut integrointimalli käyttää esimerkiksi yhteistä sovelluspalvelinta, hajautettua komponenttia tai sovelluskehystä.

Moduuli (module) on ohjelmiston toteutus, joka kokoaa tietyn toiminnallisuuden ykdeksi yksiköksi. Moduuli voi olla esimerkiksi joukko luokkia, käännösyksikkö [CleF02 s. 470]. Moduuli on joukko luokkia tai ei-olio-pohjaisia rakenteita, kuten proseduureja tai funktioita [SzC98 s. 276].

MOC (Message-Oriented Communication) viestisuuntautunut kommunikointi on tapa toimittaa viesti lähettävältä asiakkaalta (sending host) eri koneilla olevien palvelimien (communication servers) välityksellä vastaanottajalle (receiving host). Kommunikointi voi olla persistenttiä eli jokaisessa vaiheessa varmistetaan, että viesti on tallennettu. Kommunikointi on hetkellistä (transient), jolloin viesti tallennetaan vain niin kauaksi aikaa, kuin sitä tarvitaan. Kommunikointi voi olla asynkroonista, jolloin lähettäjä jatkaa

toimintaansa heti, kun on saanut toimitettua viestin lähimmälle palvelimelle. Synkronisessa kommunikointitavassa lähettäjä on kiinnittyneenä (block) siihen saakka, kunnes viesti on toimitettu vastaanottajalle [Tan02 s. 100].

MOM (Message-Oriented Middleware) on välitason ohjelmistopalvelu (message-queuing-systems), jonka avulla tarjotaan tuki pysyvälle persistentille asynkroniselle kommunikaatiolle. Tämä palvelu takaa sen, että viesti siirtyy luotettavasti palvelimelta toiselle ja asynkronisen toimintatavan ansiosta saavutetaan tarvittava tekokkuus ja skaalautuvuus [Tan02 s. 108].

Muunneltavuuspiste (variation point) tarkoittaa kohtaa, jossa muunneltavuus voi tapahtua luokassa, tyypissä tai käyttötapauksessa [JGP97 s. 440].

.NET on Microsoftin komponenttimalli oliopohjaisten hajautettujen komponenttien rakentamiseksi. Malli on suunniteltu erikoisesti web-ympäristöön ja sisältää verkkopalvelut ja vahvan tuen XML:lle. Aiemmat COM-teknologian mukaiset komponentit voidaan liittää uuteen teknologiaa käärimällä.

Porttaalisuuntautunut integrointimalli tarkoittaa palvelun tarjoajan tarjoamaa verkkopalvelua (joko web-pohjaisesti tai tietoverkon avulla), jonka avulla palvelua tarvitseva pääsee palveluun [Puu03], [Lin00].

Prosessikomponentti (processing component) esittää kohdealuespesifistä liiketoimintaprosessia tai liiketoimintoa [Myk00 s. 59], [HeS99 s. 180].

Prosessisuuntautuneessa integraatiossa tuotantoprosessit yhdistetään niin, että ne toteutetaan sovellusintegraation ja hallintatyökalujen avulla [Puu03], [Lin00].

Rajapintaa (interface) käytetään kuvaamaan olion, luokan tai muun entiteetin kuten alisysteemin ulospäin näkyvää käyttäytymistä [JGP97 s. 155].

Resurssi (resource) on tyypitettyjen yksiköiden (typed items) jäädytetty (frozen) kokoelma.

Riskienhallinta (risk management) tarkoittaa kaikkia niitä toimintoja, joilla pyritään arvioimaan ja hallitsemaan tuotteen elinkaaren aikana ilmaantuvia ongelmia. Riskienhallinnan avulla saadaan selville ne projektin riskit, jotka ovat merkityksellisiä esimerkiksi asiakkaalle ja toimittajalle. Näin kyetään lieventämään riskien vaikutuksia hankkeessa [Jal99 s. 41].

RMI (Remote Method Invocation) on olioiden välitykseen perustuva etäoliotekniikka. Tällä tekniikalla voidaan välittää asiakkaan ja palvelimen välillä oliopalvelupyyntöjä ja vastauksia (client/server-sovellus). RMI toteutetaan yleensä synkronisesti. Metodeja voidaan käyttää rajapintojen kautta. Rajapintamäärittelyn avulla generoidaan asiakaalle etäolion edustaja (proxy) ja etäpalvelimeen luuranko (skeleton) -olio. Proxyn tehtävänä on edustaa etäoliota asiakaspäässä, marshaloida oliopyyntö (pakata kutsu) ja lähettää se skeleton oliolle, joka purkaa oliopyynnön ja kyselyn [Tan02 s. 85-].

RPC (Remote Procedure Call) etämetodikutsu on menetelmä metodin parametrien siirtämiseksi etäkoneeseen, jossa metodi suoritetaan ja tulos palautetaan kysyvän tietokoneen prosessille. Kyselyn suorittaja keskustelee omassa koneessa oleva etäproseduurin edustajan kanssa ja on tietämätön etämetodin sijainnista. Paikallisessa koneessa ole-

va etäproseduurin edustaja (stub) marshaloi (pakkaa) viestin ja siirtää sen etäkoneessa olevalle proseduurille (server stub), joka purkaa viestin ja suorittaa kyselyn proseduurilta. Yleensä RPC on synkronista, mutta se voi olla aivan hyvin myös asynkronista. DCE RPC:ssä etäproseduurin asiakkaalle antamat rajapinnat kuvataan IDL-kielellä (rajapintojen kuvauskielellä), jonka pohjalta ohjelmointikielen kääntäjät osaavat generoida tarpeelliset asiakkaan korvike (stub) ja palvelimen korvike (server stub) -proseduurit [Tan02 s. 68-79].

Skaalatuvuus (scalability) tarkoittaa sitä, kuinka helposti järjestelmä tai komponentti voidaan mukauttaa sopimaan kulloiseenkin ongelma-alueeseen [www008].

Sitkeys (robustness) tarkoittaa järjestelmän kykyä ylläpitää toimintaansa silloinkin, kun sen sisäinen tila tai rakenne tai ulkoinen toimintaympäristö muuttuu. Sitkeys on suunnitteluperiaate, jonka tarkoituksena on luoda mahdollisimman stabiileja systeemejä [www013].

SOAP (Simple Object Access Protocol) on alunperin Microsoftin ja IBM:n laatima ehdotus, joka on tällähetkellä W3C:ssä suositusvaiheessa. SOAP on viestikehys (messaging framework), joka on XML:ään perustuva tiedonvälitysprotokolla ja joka on suunniteltu toimimaan hajautetussa ympäristössä. SOAP viestejä kuljetetaan yleensä http-protokollan päällä, tosin mikään ei estä käyttämästä muitakin protokollia. Viesti koostuu XML:n mukaisesta merkkauksesta, jossa <Envelope>-tagin sisällä on valinnainen <header>-elementti, jonka avulla voidaan ilmaista sovelluksen yleisiä vaatimuksia vaikkapa käyttäjäntunnistukseen liittyviä sekä <body>-tagilla ilmaistuna haluttu viesti esimerkiksi metodikutsu, virheilmoitus tai muuta haluttua tietoa. SOAP on kirjekuori viestille, johon kuuluvat siis kirjekuoren otsikko ja runkomerkintä [Jär02 s. 79].

Sovellusrajapintasuuntautunut integrointi (application integration) tapahtuu sovellusten rajapintojen välillä (API) [Puu03], [Lin00].

Sovelluspalvelin (application server) on ohjelmisto, joka muodostaa loogisen yhteyden käyttäjän asiakassovelluksen ja tietokannan tai tietokannanhallintajärjestelmän (DBMS Database Management System) välille [Ran01 s. 35].

Standardispesifisessä lisäysmallissa (standard-specific extension model) toiminnallisuutta lisätään standardiosien avulla lisäämällä standardiosia kehykseen ja luomalla kehyksen toiminnallisuus standardiosien avulla. Standardiosat voivat olla esimerkiksi tiedostojärjestelmiä, kommunkointiprotokollia, graafisia komponentteja. Käyttäjälle tarjotaan muuttumaton rajapinta. Haittoina tässä ovat standardiosien uudelleenkäytön puute, tuotespesifisten lisäysten puute ja heikentynyt ylläpidettävyys. Jokainen asiakaskomponenttiin tehtävä muutos vaatii muutoksen kehyksen tarjoamiin rajapintoihin [Bos00 s. 251].

Suunnittelumalli (design pattern) esittää perusratkaisun ongelmaan, joka esiintyy yhä uudestaan ja uudestaan. Suunnittelumallia voi käyttää yhä uudestaan. Suunnittelumalli on kuvaus keskenään vuorovaikutuksessa olevista olioista ja luokista, jotka on muotoiltu ratkaisemaan tietyssä yhteydessä esiintyvä yleinen suunnitteluongelma [GaH95 s. 3].

Systeemin infrastruktuurikehys (system infrastructure frameworks) käsittää kokoelman kehyksiä, jotka tukevat käyttöjärjestelmää, käyttöliittymiä, kommunikointia ja kielten prosessointia. Esimerkiksi MFC (Microsoft Foundation Classes) on tällainen [FaJ00 s. 616].

Tietosuuntautuneessa integroinnissa integrointi tehdään suoraan tietovarastojen kesken eikä se tapahdu sovellustasolla [Puu03], [Lin00].

Tuotespesifinen lisäysmalli (product-specific extension model) tarjoaa yhteisen rajapinnan kautta joukon operaatioita ja rajapintoja komponentin käyttäjän käyttöön. Kehys sisältää rajapinnan ja kehyksen rungon, johon voidaan implementoida erilaisia yksittäisiä tuotteita kehyksen tarjoaman rajapinnan kautta. Etuna tässä mallissa on yksinkertaisuus. Riittää, kun ylläpidetään tuotteeseen liittyvää moduulia. Haittana on tuotekohtaisten moduulien heikko uudelleenkäytettävyys [Bos00 s. 250].

UDDI (Universal Description, Discovery & Integration) on menettely, jolla verkkopalvelut pyritään paikallistamaan (http://www.uddi.org). Kyseessä on rajapinta, jonka avulla sovelluskehittäjät voivat julkaista palveluita niin, että ne ovat muiden verkkopalveluita käyttävien saatavilla. Toimiakseen tämä standardi tarvitsee palvelimen, jossa on rekisteri tarjotuista palveluista. Palvelua haluava voi tämän rekisterin avulla tiedustella haluamaansa palvelua [Jär02 s. 98].

Varuskomponentit (utility component) ovat usein uudelleenkäytettäviä palveluita. Näitä komponentteja voidaan käyttää eri kohdealueilla [Myk00 s. 59], [HeS99 s. 174].

Variantti (variant) tarkoittaa tyypinomaista rakennetta kuten käyttötapaus, oliotyyppi tai luokka. Variantti asennetaan sopiva muunneltavuuspisteeseen (variation point) ja se erikoistaa abstraktin tyypin tai luokan [JGP97 s. 440].

Verkkopalvelut (web services) tarkoittavat sovelluksia, jotka voivat keskustella toisten verkkopalveluiden kanssa käyttäen standardeja protokollia ja tiedonvälityskanavanaan Internetiä [Jär02 s. 2].

Väliohjelmisto (middleware) on verkkokerroksen päällä oleva ohjelmisto, jonka tarkoituksena on tarjota yleisiä palveluita tukien hajautukseen läpinäkyvyyttä [Tan02 s. 23].

Väliohjelmistoinfrastruktuurikehys (middleware infrastructure frameworks) taas pyrkii parantamaan ohjelmiston modulaarisuutta toimittaessa hajautetussa ympäristössä. Esimerkkinä tästä ovat erilaiset ORB-alustat kuten VisiBroker ja Orbit. Väliohjelmistoinfrastruktuurikehykseen sisältyy alla olevana tasona systeemin infrastruktuurikehys [FaJ00 s. 616].

WSDL (Web Service Description Language) on kieli, jota käytetään kuvaamaan jotakin tiettyä SOAP-palvelua. WSDL-dokumentti kertoo palvelun osoitteen, palvelun metodit ja palveluiden kutsuparametrit tietotyyppeineen [Jär02 s.3].

Yhteistyönäkökulma (approach to collaborations) tarkoittaa sitä, että vaikka komponenteille on taattava kehityksen aikainen itsenäisyys, on kehitettävä myös koko systeemin yhteistyökykyä [HeS99 s. 52]. Tämä tarkoittaa sitä, että komponenttien yhteistoiminta hoidetaan siten, että järjestelmän kiinteys taataan kehityksen kaikissa vaiheissa ja toisaalta taataan komponenttien itsenäisyys, jotta rinnakkainen komponenttikehitys olisi mahdollista.

LIITE 2

Arkkitehtuurin parantaminen kilpailukeinona

CASE 1

KUOPION KAUPUNKI SOSIAALI- JA TERVEYSKESKUS

dokumentti	selite	versio	tekijä	pvm
KuSosTervRaportti.doc	Haastattelu 7.1.2003 /Ahola	1.3	HK	9.4.2003

Harri Karhunen
Kuopion yliopisto
Kuopiossa 9. huhtikuuta 2003

Sisällysluettelo

1 Yleistä 152
2 Haastattelu 152
3 Konteksti 152
4 Ongelma 152
5 Ratkaisu 152
6 Vaatimukset 152
7 Laatuvaatimukset 153
8 Käyttötapaukset 154
9 Järjestelmäintegraatio 154
10 Järjestelmän asettamat vaatimukset 155
11 Valinta 155
11.1 Pegasos ominaisuudet 155
11.1.1 Tiedot ohjelmistotoimittajasta 155
11.1.2 Ohjelmistoarkkitehtuuri 155
11.1.3 Järjestelmäintegraatio 156
12 Käyttöönotto 157
13 Ylläpito 157
14 Tuotteen tietoja 157
15 Toteutuneet vaatimukset 157
16 Lopuksi 157

1 Yleistä

Haastattelun tavoitteena on kartoittaa tietojärjestelmän hankinta- ja käyttöönottoprosessin vaatimusten määrittelyä, rakentuvaa arkkitehtuuria sekä tutkia uuden arkkitehtuurin aiheuttamia muutoksia organisaation toiminnassa.

2 Haastattelu

Haastattelu tehtiin 7.1.2003 Kuopion sosiaali- ja terveyskeskuksessa. Haastateltava oli Juhani Ahola.

3 Konteksti

Tarkasteltava projekti oli terveystoimen tietojärjestelmän uusimisprojekti. Kuopion kaupunki on uusinut tietojärjestelmänsä vuonna 2000 alkaneen valintaprojektin mukaisesti, jossa määriteltiin järjestelmälle asetettavat vaatimukset. Markkinakartoitus tehtiin vuonna 1999, jonka perusteella valittiin toimittajaehdokkaat. Järjestelmän tehtävänä on hoitaa perusjärjestelmälle asetettavat tehtävät, jotka jakautuvat avohoitoon, laitoshoitoon, kotihoitoon ja psykososiaaliseen työhön. Projektissa kartoitettiin nykytilanne, kehitettiin toimintaprosesseja, muodostettiin valintakriteerit, suoritettiin valittujen toimittajien esittelyt, suoritettiin tarjouskyselykierros, arvioitiin tarjoukset ja toimittajien kypsyys, tehtiin käyttöönottosuunnitelmat ja hoidettiin sopimusneuvottelut

4 Ongelma

Sosiaali- ja terveyskeskuksen tietojenkäsittelytehtävien hoitaminen on hoidettu useilla sovelluksilla, joiden välinen kommunikointi (tietojenvaihto) on ollut vähäistä. Tietoa on jouduttu kirjaamaan moneen kertaan ja tietojensiirto sähköisesti ohjelmistojen kesken on koettu ongelmaksi.

5 Ratkaisu

Terveydenhuollossa tietojärjestelmien toiminnan kehittämisessä on otettava huomioon järjestelmien välinen integraatio. Integraatio voidaan joko hoitaa niin, että sovellukset kommunikoivat eri tekniikoilla viestien välityksellä tai sovelluksia kehitetään niin, että ne voivat toimia yhteistoiminnassa. Yhteistoiminnallisuus tarvitsee sekä tekniset että semanttiset säännöt. Tiedolla on muoto ja merkitys. Nämä tekijät tekevät järjestelmäintegraatiosta vaativan ja aikaa vievän.
Ratkaisuna on hajautettu tietojenkäsittely käyttäen sellaista ohjelmistoratkaisua, joka mahdollistaa ohjelmistokomponenttien hankinnan valmiina ohjelmistotoimittajalta ja johon muut jo olemassa tietojärjestelmät voidaan integroida. Järjestelmien välinen tietojenvaihto tapahtuu viestien muodossa (HL7 tai XML), jolloin väliin voidaan luoda adapteri tai perinnejärjestelmiin kääre (Wrapper) viestien siirron mahdollistamiseksi.

6 Vaatimukset

Järjestelmälle asetettiin toiminnalliset vaatimukset, jotka muodostettiin keräämällä tietoja käytössä olevasta järjestelmästä ja kartoittamalla toimintaprosessit. Prosessien kuvauksissa käytettiin SA-menetelmän mukaisia prosessikaavioita. Prosessikuvaus tehtiin

ulkopuolisen konsultin toimesta kaikista prosesseista (SA-menetelmä vuokaavio: syöte/tulos) ja liitettiin tarjouspyyntöön.

Hankkeen riskit kartoitettiin riskianalyysin avulla, jossa pyrittiin tunnistamaan tärkeimmät riskit ja toimet, joilla kukin riski on vältettävissä.

Järjestelmän toiminnalliset vaatimukset kartoitettiin keräämällä projektiryhmän tarpeet ja kehittämiskohteet. Tämä liittyi läheisesti toimintaprosessien kehittämiseen.
Lopputuloksena näistä molemmista (toimintaprosessit ja toiminnalliset vaatimukset) saatiin muodostettua valintakriteeristö, jota käytettiin koekäytön aikana.

Järjestelmän ominaisuuksia ja vaikutelmia kartoitettiin koekäyttämällä kutakin järjestelmää ja suorittamalla käyttäjien kyselytutkimus, jossa arvioitiin mm. käytettävyyttä.
Toiminnalliset vaatimukset on arvioitu kullekin toiminnalle erikseen laaditulla 250:n kysymyksen sarjalla, jossa vastausarvot ovat väliltä 1-5. Yhteensä 6 x 250 kysymystä. Vastauksista on laskettu pisteet ja tulokset esitetty graafisesti.

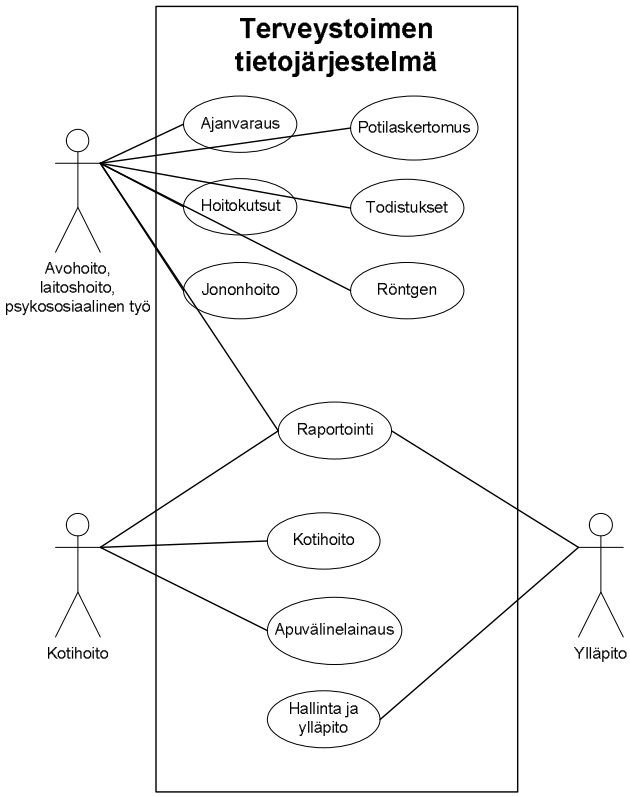
Järjestelmälle ei laadittu mitattavia laatuvaatimuksia, vaan laadun arviointia suoritettiin tutustumalla referensseihin (Espoo, Turku), joita lisäksi täydennettiin koekäytössä saaduilla kokemuksilla. Ostajan on vaikea arvioida laatutekijöitä valmiista ohjelmistosta pelkän testiaineiston perusteella. Kattava laatuselvitys vaatisi runsaasti aikaa ja panostusta, ja käytännössä sellaisen tekeminen on mahdotonta. Vertailun suorittamiseksi tulisi voida käyttää vastaavanlaista järjestelmää käytännön tilanteissa. Siksi referenssien merkitys ja toimittajan luotettavuus korostuu ohjelmistovalinnassa.

7 Laatuvaatimukset

Koska kyseessä on valmisohjelma, laatutekijöitä on jouduttu arvioimaan välillisesti: Suorituskykyä arvioitiin välillisesti koekäytön yhteydessä sekä toimittajan antamien suorituskykytietojen avulla. Ylläpidettävyydestä ei ollut mahdollista tehdä teknistä arviota. Tukeuduttiin referensseihin (Turku ja Helsinki), jotka olivat jo tehneet hankintapäätöksen.
Siirrettävyyden osalta ennakkoehdoksi asetettiin, että järjestelmän on toimittava kaupungin laitearkkitehtuurissa. Toipumisesta ei tehty arvioita, koska edellytyksiä sellaisen tekemiseen ei ollut. Samanaikaisuutta tutkittiin koekäytön yhteydessä, jolloin joitakin lukkiutumia ilmeni. Skaalautuvuusvaatimuksia arvioitiin mahdollisia käyttäjämääriä kartoittamalla.

Järjestelmä on toimintakriittinen ja sen tulee olla toiminnassa kaikissa tilanteissa. Tällöin korostuvat toimintavarmuus, turvallisuus, tietoturva sekä toipuminen virhetilanteissa. Koska järjestelmä käsittelee potilaiden tietoja, on tietoturvalla aivan erityinen painoarvo.

8 Käyttötapaukset



Kuva 1. Käyttötapauskaavio

Päätoiminnot:

- ajanvaraus (työohjelmat)
- potilaskertomus
- todistukset
- laskutus
- hoitokutsut
- raportointi
- järjestelmän hallinta ja ylläpito
- jononhoito
- kotihoito (kotisairaanhoito/kodinhoito)
- liittymät muihin ohjelmistoihin
- tietosuoja
- apuvälinelainaus
- röntgen (ei kuvankäsittelyä)

9 Järjestelmäintegraatio

Järjestelmä integroituu muihin järjestelmiin point-to-point –yhteyden avulla.

10 Järjestelmän asettamat vaatimukset

Järjestelmän on oltava yhteensopiva alustaksi tulevan Kuopion kaupungin ATK-keskuksen laitearkkitehtuuriin.

11 Valinta

Toimittajaksi valittiin Novo ohjelmallaan Pegasos.

11.1 Pegasos ominaisuudet

11.1.1 Tiedot ohjelmistotoimittajasta

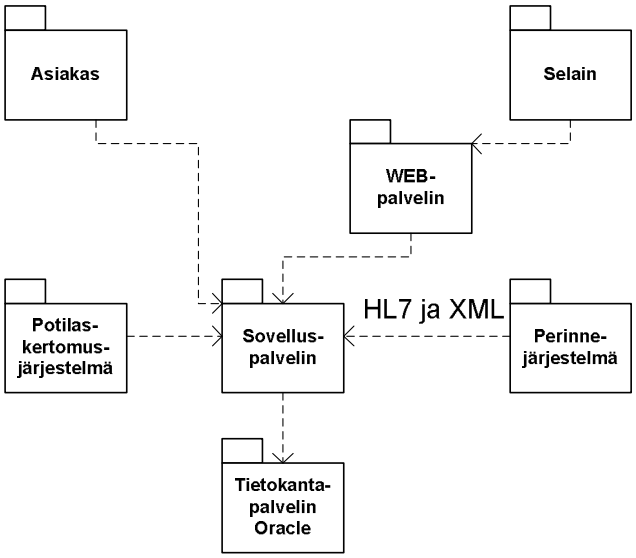
Novo Oyj on pitkää ohjelmistoalalla toiminut yritys, jolla on pitkäaikainen kokemus julkishallinnon ohjelmistoista. Julkishallinnon osuus liikevaihdosta on 40%:n luokkaa. Novo on toimittanut Pegasos-potilaskertomusjärjestelmän mm. Helsinkiin (2500 työasemaa), Tampereelle (yli 2000 työpistettä) ja Turkuun sekä Tornioon. Helsingin hankkima ohjelmisto sisältää ohjelmistoja mm. sairauskertomusten ylläpitoon, ajanvaraukseen, tilastointiin, potilaslaskutukseen ja reskontraan, väestötietoihin, terveysneuvontaan, fysioterapiaan sekä lääkitystietojen päivitykseen.

11.1.2 Ohjelmistoarkkitehtuuri

Ohjelmiston kehitysympäristönä on Compuware Corporationin tekemä Uniface Universal Request Broker Architecturen mukainen sovelluskehitin.
Ohjelmiston middleware-alusta Compuware Corporationin tekemä Broker-arkkitehtuurimallin mukainen Uniface Universal Request Broker Architecture -malliin pohjautuva ratkaisu. Valittavissa on arkkitehtuurit joko yleisen CORBA-määrittelyn mukaisesti alustariippumattomiin vaihtoehtoihin tai Microsoft-sidonnaiset COM+ tai .NET-arkkitehtuuriin pohjautuvat mallit. Middlewarena voidaan käyttää esimerkiksi IIOP, RMI, XML tai BEA TUXEDO [www014].

Käyttöliittymä voidaan rakentaa joko sovelluspohjaisen liittymän avulla tai selainpohjaisesti. Selaimen avulla tuettuina ovat tavallisimmat web-palvelimet: Microsoft IIS, iPlanet Enterprise Edition, WebSphere, Apache, Lotus Domino R5. Arkkitehtuurin viitemallina Pegasoksessa on kolmitasoinen kerrosarkkitehtuurimalli.

Kuvassa 2 on kuvattu arkkitehtuurin viitemalli käyttäen UML:n mukaista pakettikaavioesitystä. Näkökulmana on arkkitehtuurin kerrosmalli, jossa asiakkaan käyttöliittymäkerrosta kuvataan asiakas- ja selainkomponenteilla. Liiketoimintakerroksena ovat web-palvelin ja sovelluspalvelin, joka kuvaa Pegasos-ohjelmistoa. Pegasokseen liittyvänä kuvataan omana komponenttinaan potilaskertomusjärjestelmä. Muut integroitavat ohjelmistot kuvataan Perinnejärjestelmä-komponentissa. Resurssikerroksessa on kuvattu tietokantapalvelinohjelmisto.



Kuva 2. Arkkitehtuurin viitemalli

Valittu ohjelmistoarkkitehtuuri on skaalautuva ja sovelluskehys tarjoaa perustietoturvan. Middleware-ohjelmisto parantaa ylläpidettävyyttä, jolloin ohjelmistotoimittaja voi keskittyä liiketoimintakomponenttien kehittämiseen. Alusta mahdollistaa hyvinkin joustavan komponenttimallin käytön. Suorituskyky kärsii hajautetun arkkitehtuurin takia, mutta ilman toimivaa komponenttialustaa ei ole mahdollista rakentaa luotettavaa, ylläpidettävää, portattavaa ja skaalautuvaa järjestelmää.

11.1.3 Järjestelmäintegraatio

Järjestelmästä on rakennettu liittymiä toisiin järjestelmiin. Liittymiä on tällä hetkellä noin 25 kpl. Uusia liittymiä rakennetaan tarpeen mukaan. Liittymiä on tehty mm. taloushallintoon, laboratorioon, lähetejärjestelmään ja sairaanhoitojärjestelmään. Muita liittymiä ovat kuvantamisliittymät, spirometri. Myös työpöytäintegraatiota on suoritettu. Liittymät joudutaan määrittelemään tapauskohtaisesti käyttäen viestityyppeinä joko HL7 tai XML-spesifikaatiota. Lisäksi on erilaisia tiedostonvälitystehtäviä. Järjestelmäintegraatiota on tutkittu PlugIT-projektissa, jossa tutkitaan terveydenhuollon järjestelmien integrointimenetelmiä ja rakennetaan menetelmä integraatiomäärittelyjen suorittamiseksi, jota voidaan käyttää hyödyksi integraatioiden toteuttamisessa.

Järjestelmäintegraatio on yksi terveydenhuollon sovellusten tärkeimpiä kehityskohteita. Yhteistyö sairaanhoitopiirien, hoitoyksiköiden, hoitohenkilöstön kesken on pystyttävä järjestämään nykyistä paremmin. Järjestelmäintegraation kautta on saavutettavissa merkittäviä säästöjä. Jo yksistään potilaskertomuksen sähköinen käsittely (sähköinen allekirjoitus) terveyskeskuksen ja sairaalan välillä, säästää kustannuksia ja jouduttaa hoitotoimenpiteitä. Samoin tieto potilaalle tehdyistä kokeista auttaa vähentämään turhia kokeita.

12 Käyttöönotto

Asennus suoritettiin pilottihankkeena alue kerrallaan. Asennuksessa ei ilmennyt suurempia ongelmia. Järjestelmä ollut käytössä 2 vuotta.

13 Ylläpito

Järjestelmän ylläpito hoidetaan tehdyn ylläpitosopimuksen mukaisesti, johon sisältyy tarvittava puhelinneuvonta. Virheidenkorjaus hoidetaan siten, että kerätään havaitut virheet ja korjataan ne päivitysversioissa. Päivitysversiot toimitetaan 2-3 kertaa vuodessa. Ylläpidossa kehitetään laajoja uudistuksia yhteistyössä käyttäjän kanssa tavoitteena luoda entistä parempi tuote. Ylläpidossa ei kuitenkaan etsitä uusia käyttökohteita.

14 Tuotteen tietoja

Tuotteen elinkaari on 10-20 vuotta. Tuote on toimintakriittinen 24/7, joten siihen sisältyy automaattiset hälytykset. Tiedonhallintajärjestelmänä on Oracle.

15 Toteutuneet vaatimukset

Tuote vastaa asetettuja tavoitteita. Palvelimen kapasiteettia on kasvatettu, jotta haluttu suorituskyky saavutetaan. Käyttäjien taholta toiveena on parempi käytettävyys. Ongelmaksi on koettu suuri peräkkäisten ikkunoiden jono käyttöliittymissä, mutta asia korjaantunee tulevissa versioissa.

16 Lopuksi

Tällainen hyvin käyttäjäspesifinen ohjelmisto kuten Pegasos kehittyy vuosien kuluessa käyttäjiltä ja ylläpidosta tulleiden impulssien mukaisesti. Hankinnassa vaikka se olisi valmisteltu kuinka huolellisesti tahansa, ei voida ottaa huomioon kaikkia käytössä ilmeneviä seikkoja. Käytännössä ilmenevät kehitystoiveet toteutuvat uusissa versioissa. Asiakas ja ohjelmistotoimittaja ovat tiiviissä yhteistyössä keskenään. Molemmat hyötyvät yhteistyöstä: asiakas voi tehostaa toimintaansa kehittyvän ohjelmiston ansiosta ja toimittaja saa paremmin asiakkaan vaatimuksia vastaavan tuotteen.

LIITE 3

Arkkitehtuurin parantaminen kilpailukeinona

CASE 2

PRIMUS-OPPILASHALLINTOJÄRJESTELMÄ KUOPION KAUPUNKI OPPIMATERIAALIKESKUS

dokumentti	selite	versio	tekijä	pvm
Primus_Raportti.doc	Haastattelu 9.1.2003 /Aalto	1.0	HK	9.4.2003

Harri Karhunen
Kuopion yliopisto
Kuopiossa 9. huhtikuuta 2003

Sisällysluettelo

1 Yleistä 160
2 Haastattelu 160
3 Konteksti 160
4 Tiedot ohjelmistotoimittajasta (toimittajan ilmoittamat) 160
5 Ongelma 162
6 Ratkaisu 162
7 Toiminnallisia vaatimuksia 163
8 Laatuvaatimukset 163
9 Järjestelmäintegraatio 164
10 Järjestelmän asettamat vaatimukset 164
11 Ohjelmistoarkkitehtuuri 164
12 Toteutuneet vaatimukset 165

1 Yleistä

Haastattelun tavoitteena on kartoittaa käytössä olevan oppilasarvostelutietojärjestelmän käyttö- ja kehitystoimenpiteitä pyrittäessä parantamaan ohjelman ylläpidettävyyttä ja käytettävyyttä.

2 Haastattelu

Haastattelu tehtiin 9.1.2003 Kuopion kaupungin oppimateriaalikeskuksessa (OM-keskus). Haastateltava oli Tapio Aalto.

3 Konteksti

Kuopion kaupungin opetustoimen alaiset oppilaitokset suorittavat oppilasarvostelunsa StarSoft Oy:n [www00] toimittamalla Primus-ohjelmistolla. Ohjelmisto on rekisteripohjainen alunperin yhdelle koneella rakennettu PC-ratkaisu. Tällä hetkellä on käytössä versio 3 ja parhaillaan ollaan ottamassa käyttöön versiota 4. Versio 3 on 16-bittinen ohjelmisto ja versio 4 32-bittinen. Versio 4 toteuttaa client/server-arkkitehtuurin, jossa palvelinosa sijaitsee palvelimella ja palvelee useita käyttäjiä. Versio 4 mahdollistaa yhteisen usean koulun sisältävän rekisterin luomisen monioppilaitosversiossa. Kuopio on ottamassa (tai jo ottanut) käyttöön monioppilaitosversiota.

4 Tiedot ohjelmistotoimittajasta (toimittajan ilmoittamat)

StarSoft Oy on vuodesta 1987 toiminut suomalainen ohjelmistotalo, joka kehittää ja tuottaa kouluhallinnon ohjelmia. StarSoftilla on nyt kaksi tuotelinjaa: **Kouluhallinnon** ohjelmien muodostaman ohjelmistokokonaisuuden rinnalle StarSoft on kehittänyt kokonaan uuden tuotelinjan, **tapahtumanhallintaohjelmat**. Kouluhallinnossa StarSoft on markkinajohtaja Suomessa. Ressun avulla hallitaan koko toimintaympäristöä, josta käytämme termiä tapahtumanhallinta. Ressusta on kaksi perusversiota, joista toinen esitellään Starsoft:n sivuilla [www15].

4.1 Ohjelmien ominaisuuksia

Primus 3:

- Käytössä on Windows-alusta.
- Ohjelma on 16-bittinen.
- Ohjelmisto oli alun perin yhden PC:n käyttäjäversio, jolloin verkossa käytettäessä käyttäjä latasi omalle työasemalleen myös rekisteritiedot. Ohjelmiston viitearkkitehtuuri on nykyisin client/server-pohjainen, jolloin client-osaa palvellaan Windows-palvelimella olevalla server-osalla.
- Ohjelmisto on rekisteripohjainen. Ohjelmisto on monoliittinen niin, että ohjelmistopäivitykset ovat vaatineet toisinaan myös rekistereiden konversioita.
- Ohjelmisto on lähiverkossa toimiva PC-pohjainen ratkaisu.
- Ohjelman päätehtävänä oppilasarvostelu, joka sisältää mm. oppilastiedot, kurssitiedot ja oppilaan arvostelutiedot.
- Yhteistoiminta opetuksensuunnitteluohjelman (Kurre) kanssa mahdollistaa tietojenvaihdon opetuksensuunnittelun ja arvostelun synkronoimiseksi. Tietojenvaihto tapahtuu erillisen dialogin avulla. Tämän tietojenvaihdon avulla laaditaan

opetussuunnitelmat ja päivitetään laaditut suunnitelmat oppilastietoihin (Primus). Tällä menettelyllä voidaan hoitaa mm. oppilaiden siirtäminen toiselle luokalle.

Primus 4:

- Käytössä on Windows-alusta.
- Ohjelma on 32-bittinen.
- Ohjelmisto on rekisteripohjainen.
- Ohjelmiston viitearkkitehtuuri on 3-kerrosarkkitehtuuri, koostuen asiakasosasta, web-palvelinosasta sekä sovellus- ja tietokantaosasta.
- Palvelinosa on palvelimella ja nopeassa verkossa myös asiakasosa kannattaa asentaa palvelimelta ladattavaksi.
- WWW-osa on toteutettu erillisellä Wilma-komponentilla. Komponentti on toteutettu Perl-skriptillä, joten haluttaessa käyttää erillistä www-palvelinta tarvitaan esim. Linux-ympäristössä Apache-ohjelmisto ja Perl-tulkki. Erillistä www-palvelinta ei tarvita välttämättä, vaan komponentti toimii myös palvelimena. Wilma-komponentin avulla oppilas voi selaimen kautta tutustua kurssitarjontaan (kurssitarjotin) ja valita mieleisiään kursseja (lukiotasolla) sekä katsella omaa työjärjestystään. Tulosteena oppilas saa opintorekisteriotteen suorittamistaan kursseista. Opettaja voi suorittaa oppilasarvioinnin selaimen kautta. Palauteosion kautta voidaan antaa palautetta. Palautteista voidaan tulostaa luokiteltuja raportteja.
- Käyttöliittymä noudattaa Windows-asettelumallia
- Aiempaan versioon parannuksena on se, että tietokantaosa on omana komponenttinaan, jolloin ohjelmistopäivitykset eivät vaadi muutoksia rekistereihin.
- Ohjelmisto on lähiverkossa toimiva aidosti client/server-pohjainen ratkaisu, jolloin palvelinosa on verkossa Windows-palvelimella ja asiakasosa asiakkaan työasemassa.
- Ohjelman päätehtävänä oppilasarvostelun suorittaminen, joka sisältää mm. oppilastiedot, kurssitiedot ja oppilaan arvostelutiedot.
- Ohjelmistosta on liittymät Kurreen.
- Ohjelmistossa on ominaisuuksina mm. ylioppilaskirjoitusten tulosteet, lukiotodistuksia sekä erikoiskokeiden todistuksia.
- Yhteistoimintaa on mm. seuraavien ulkopuolisten järjestelmien kanssa tilastokeskus, YTL ja Kela.
- Ohjelmisto ja työpöytäintegraatio on mahdollista mm. Excelin, leikepöydän, tekstinkäsittelyohjelmien, erilaisten kuntaohjelmien (Kuntiainen, Pegasos, Primus) sekä Reitti-GIS -ohjelman kanssa.

Monioppilaitosversio versio Primus 4:

- Käytössä ovat Windows- ja Linux-alustat.
- Ohjelma on 32-bittinen.
- Ohjelmisto on rekisteripohjainen.
- Perusominaisuudet ovat kuten versiossa 4.
- Eri oppilaitosten rekisterit ovat samassa rekisterissä.
- Etuina ovat mm. suurempi joustavuus oppilaiden käsittelyssä sekä se, että duplikaatteja ei enää esiinny.
- Haittoina ovat järjestelmän katkokset, vaadittava yhdenmukaisuus oppilaitosten kesken sekä vaatimus keskitetystä tietoturvasta.

- Monioppilaitosversio mahdollistaa esim. koulupiirin laatimien julkisten tulostusten ja yhteenvetojen laatimisen, koska ajot voidaan suorittaa yhdestä kannasta.

Ohjelmistoon liittyy erillisenä ohjelmiston Kurre-opetuksensuunnitteluohjelma, jolla on yhteys tiedonsiirron avulla Primukseen. Kurren tarkoituksena on suunnitella oppilaiden opetus, opetussuunnitelmat, opetusryhmät jne.

5 Ongelma

Kullakin koululla oli esim. kansliassa yhdellä PC-laitteella Primus-ohjelma (Primus w3). Ohjelma oli rekisteripohjainen, joten koordinaation puutteen johdosta jokainen koulu alusti rekisterinsä omalla tavallaan. Myös tulosteet kukin koulu konfiguroi tavallaan. Ohjelmiston käyttö paikallisverkossa tapahtui siten, että käyttäjä käynnistää jaetulta levyltä ohjelman, joka latautuu käyttäjän koneelle ja lukee rekisterin tiedot käyttäjän työtilaan.

Tämän ratkaisun etuna oli sen koululle antama suuri vapaus. Ohjelmaa voitiin muokata niin kuin haluttiin ja rekisterit voitiin rakentaa juuri sellaisiksi kuin haluttiin. Tämän ratkaisun haittana oli liika hajautus. Yksittäisten koulujen rekistereistä ei voitu listata useiden koulujen yhteenvetoraportteja. Tietojen yhtenäisyys (consistency) puuttui ja lisäksi tiedot olivat yksittäisillä koneilla. Järjestely aiheutti lisäksi mikrotuelle runsaasti turhaa työtä.

6 Ratkaisu

OM-keskus ja ohjelmistotoimittaja alkoivat kehittää ohjelmaa siten, että siinä voitaisiin paremmin hyödyntää Kuopion kaupungin tietoverkkoa, sekä muuttaa olevaa tilannetta niin, että saavutettaisiin parempi yhteistoiminnallisuus. Ratkaisuksi muodostui vähittäinen siirtyminen asiakas/palvelin-arkkitehtuuriin. Aluksi tehtiin niin, että yhden käyttäjän versio siirrettiin yhteiselle palvelimelle, josta verkossa olevat käyttäjät (koulut) latasivat omat versionsa koneilleen. Parhaillaan ollaan siirtymässä keskitettyyn client/server-arkkitehtuuriin ottamalla käyttöön monioppilaitosversio.

6.1 Versiohistoria

Primus versio 3 on ollut aluksi jokaisella koululla omilla koneilla, jolloin koordinoinnin puuttuessa jokainen koulu on luonut tietokenttiä ja raportteja omaan tyyliinsä. Noin vuosi sitten otettiin käyttöön yksi yhteinen palvelin, johon on siirretty kaikki erilliset ohjelmat (yhteensä 46 koulua). Samalla on sovittu kannoille ja raporteille yhteinen tietosisältö ja muoto. Tällöin virasto saa ohjelmasta tarpeelliset virastotulosteet koontina tilastointiaan varten ajamalla erillisen raporttiohjelman. Käyttäjät lataavat sovelluksen työpöytäkuvakkeen avulla. Ohjelman asiakasosa haetaan palvelimelta käyttäjän työtilaan.
Ohjelmisto on toteutettu kaksi-kerrosarkkitehtuurilla, jossa käytetään omaa tietokantaa, jolloin tietokantamangeri on osa sovelluspalvelua. Tietokannan rekisteri koostuu kahdesta tiedostosta (määrittelyt *.ktn.prd ja data *.prd).

Primus versio 4 otetaan käyttöön vuoden 2003 aikana niin sanottuna monioppilaitosversiona, jossa tietokannat yhdistetään yhdeksi kannaksi. Näin hallinta on helpompaa.

Valtaosa (96%) kouluista on siirtynyt versiosta 3 versioon 4 suoraan tietokanta konversion kautta, mutta Kuopio on tehnyt toisin. Kuopiossa kouluasteen sisällä on käytössä samanlaiset tietomäärittelyt. On muodostettu projektiorganisaatioita. Projektiorganisaatio suorittaa määrittelyt ja tutkii tarpeet. Kartoitusten perusteella on luotu testikanta. Testikantaa on parannettu iteratiivisesti. Määrittelyn tulokset toimitettu on Starsoft Oy:lle, joka suorittaa ohjelmapäivitykset ja konversiot. Tällä hetkellä Kuopion ATK-keskuksessa on vuokrattuna palvelin, jossa pyörii 46 prosessia omissa kannoissaan. Virastoversion testaus alkaa ja laitetaan tuotantoon vuonna 2003.

7 Toiminnallisia vaatimuksia

Toteutuneita ominaisuuksia ovat:

- koulukohtaiset määritykset mahdollisia
- arkistoon tulostaminen

Käyttäjien esittämiä toiminnallisia ominaisuuksia ovat:

- Kentältä tulee uusia toiveita mm. julkisia kortteja, joita tavallinen käyttäjä voi vain lukea ko. tietueita.
- On oltava mahdollisuus tehdä tietuekohtaisia muutoksia koulutasolla.
- On voitava suorittaa tietuekohtainen kopiointi.

8 Laatuvaatimukset

Laadullisia kriteerejä ovat mm. seuraavat:

- nopea tiedonhaku
- asennuspaketti (verkosta) kolme hakemistoa

Haasteltava subjektiivinen arvio keskeisimmistä ohjelmistolle asetettavista laatutekijöistä kouluarvosanoin :

laatutekijä	painoarvo 1-10
suorituskyky	10
ylläpito	7
siirrettävyys	9
luotettavuus (varmuuskopiot ajastettuna, transaktioloki)	9
toipuminen (muutokset suoraan levylle)	9
turvallisuus	7
tietoturva	7
salaus	7
samanaikaisuuden hallinta	8
konfiguroitavuus (staattiset ini + kenttiä dynaamisesti)	8
käytettävyys (ruudulla paljon tietoa)	8
joustavuus (sopii moniin tarkoituksiin – yliopistokäyttöön)	8
skaalautuvuus	8
pysyvyys	8
transaktioidenhallinta	9
testattavuus (kehityskanta,testikanta)	9
modulaarisuus (koulukohtaiset moduulit nyt, client 2 tiedostoa, kuvake työpöydällä)	

Laadun arviointimenettely suoritetaan seuraavasti:

- Aloite tulee käyttäjiltä.
- Haluttuja piirteitä ja ominaisuuksia arvioidaan simuloimalla.
- Analysointi johtaa halutun ominaisuuden tilaukseen.
- Saatu päivitys testataan.
- Arviointi perustuu pääasiassa kokemukseen.
- Käyttäjäkyselyitä ei ole tehty – kokemus perustuu vuorovaikutukseen.

Ohjelmiston päätoiminnot (monioppilaitosversio) ovat:

- koulukohtainen oppilasarvostelu
- koulukohtaiset raportit
- ylläpito
- virastolle yhteenvetoraportit

9 Järjestelmäintegraatio

Järjestelmästä on tiedonsiirtoyhteydet työpöytäintegraationa: Exceliin, tekstinkäsittely-ohjelmaan, Kurreen ja leikepöydälle. Tiedonsiirtointegraatiota on toteutettu mm. Pegasoksen, tilastokeskuksen, väestörekisterikeskuksen ja Kelan kanssa.

10 Järjestelmän asettamat vaatimukset

Järjestelmän on oltava yhteensopiva alustaksi tulevan Kuopion kaupungin ATK-keskuksen laitearkkitehtuuriin.

11 Ohjelmistoarkkitehtuuri

Kuvassa 1 ja 2 on kuvattu arkkitehtuurin viitemalli käyttäen UML:n mukaista pakettikaavioesitystä.

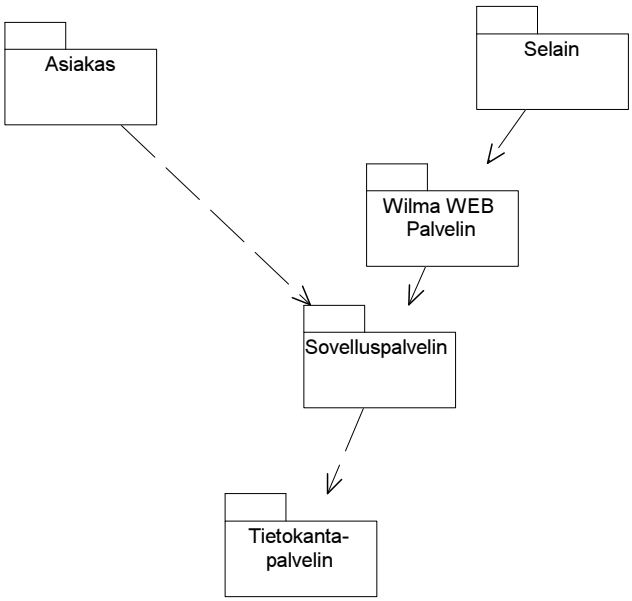
11.1 Primus versio 3 client/server arkkitehtuuri

Näkökulmana on versiossa 3 on asiakas/palvelin-arkkitehtuuri, jossa asiakkaan käyttöliittymä kuvataan Client-komponenteilla. Liiketoimintakerroksessa on Server-komponentti, joka toimii palvelimena.

Kuva 1. viitearkkitehtuuri versio 3

11.2 Primus versio 4 monioppilaitosversio kolmikerrosarkkitehtuuri

Näkökulmana versiossa 4on arkkitehtuurin kerrosmalli, jossa asiakkaan käyttöliittymä kuvataan Asiakas- ja Selain-komponenteilla. Käyttäjä voi käyttää ohjelmistoa joko Asiakas-komponentin kautta tai selaimen kautta. Liiketoimintakerroksessa ovat web-palvelin ja sovelluspalvelin, joka kuvaa Primus-server-komponenttia.



Kuva 2. viitearkkitehtuuri versio 4

12 Toteutuneet vaatimukset

StarSoft esittelee sivuillaan Vaasan yliopiston tekemää asiakastyytyväisyystutkimusta [www16], jossa tutkittiin asiakkaille lähetetyn kyselyn avulla asiakastyytyväisyyttä. Tutkimus tehtiin vuonna 1996 lähettämällä noin tuhannelle asiakkaalle kysely. Vastausprosentti oli 40%. Kyselyn mukaan kyselyn vastanneet asiakkaat ovat tyytyväisiä.

Haastattelun perusteella voidaan todeta, että viitearkkitehtuurissa tehtyjen muutosten jälkeen Kuopiossa ollaan tyytyväisiä ohjelmaan.

Arkkitehtuurimuutoksilla on saavutettu seuraavat ominaisuudet:

- Keskityksen ja tiedon konsistenssin parantumisen avulla ohjelmiston ylläpito on parantunut. Erityisesti mikrotuen tarve on vähentynyt, koska ohjelmisto ja laitetuki voidaan hoitaa keskitetysti OM-keskuksessa. Ylläpito voi keskittyä oppilaitoksissa käyttäjäkoulutukseen. Kouluista puuttuu usein tarvittava tietotekninen osaaminen, jota vaaditaan ohjelmiston ylläpitämiseksi.
- Ohjelmistoa on kehitetty koko ajan, jolloin käyttäjien tarpeita on voitu huomioida paremmin.
- Haittoja keskitetystä mallista ei haastateltava löytänyt.
- Ohjelmiston versio 4:n selainkäytön mahdollistavat ominaisuudet parantavat ohjelman käytettävyyttä mahdollistaen mm. etätyöskentelyn.

LIITE 4

Arkkitehtuurin parantaminen kilpailukeinona

CASE 3

PROINTRA-INTRANET

-OHJELMISTO

JSOP OY

dokumentti	selite	versio	tekijä	pvm
JSOP_Raportti.doc	Haastattelu Matti Koskivirta 9.1.2003	1.0	HK	9.4.2003

Harri Karhunen
Kuopion yliopisto
Kuopiossa 9. huhtikuuta 2003

Sisällysluettelo

1 Yleistä 168
2 Haastattelu 168
3 Konteksti 168
4 Tiedot ohjelmistosta 168
5 Ongelma 169
6 Ratkaisu 170
7 Järjestelmäintegraatio 170
8 Ohjelmistoarkkitehtuuri 170
9 Uuden version toteutus ja kehitettävää 170

1 Yleistä

Haastattelun tavoitteena on kartoittaa tuotannossa olevan Intranet- ja Internet-porttaaliohjelmiston tuotekehitysnäkemyksiä, joiden tavoitteena on parantaa ohjelman markkinoitavuutta.

2 Haastattelu

Haastattelu tehtiin 9.1.2003 JSOP:ssä Kuopiossa haastateltava Matti Koskivirta.

3 Konteksti

ProIntra-ohjelmiston tarkoituksena on palvella käyttäjää ja tarjota käyttäjälle pääsy hänen tarvitsemaansa tietoon. Ohjelmisto on toteutettu ns. porttaaliratkaisuna. Käyttäjän käyttöliittymänä sivustoon on selain. Porttaali voi toimia joko Intranetissa, Internetissä tai Extranetissä.

4 Tiedot ohjelmistosta

ProIntra-tuote on ollut markkinoilla kolmisen vuotta. Myynnissä on tällä hetkellä versio 3. Ohjelmisto on valmisohjelma, jota pyritään joko myymään tai vuokraamaan asiakkaalle. Halutessaan asiakas voi vuokrata ohjelman käyttöoikeiden ja levytilaa JSOP:n ylläpitämästä web-hotellista. Tällöin asiakkaan palveluprosessit pyörivät web-hotellin palvelimella.

ProIntran tarjoamat palvelut ovat:

- sisällöntuotanto sisältäen mm. aineistopankin, kuvagallerian ja html-sisältösivun
- metatietokanta tiedon jäsentämiseen ja luokittelemiseen
- lisätoimintoina mm. keskustelupalsta, aloitelaatikko, linkkilistat, ilmoittautumislomakejärjestelmä, resurssivaraus (esim. kokoushuoneiden yms.), kalenteri (käyttäjä, ryhmä, yritys), toimeksiantojen hallinta, projektienhallinta, verkko-opisto

Ohjelmisto on komponenttipohjainen ja toteutettu Microsoft-alustalle. Ohjelmiston [www17] viimeisin versio on 3.11, jossa on mukana SOAP-rajapinta.

4.1 Toiminnalliset ominaisuudet

Toiminnallisia ominaisuuksia kehittävät tuotteenhallintayksikkö yhdessä tuotevastaavan kanssa. Saatu asiakaspalaute pyritään hyödyntämään tuotekehityksessä. Lisäksi kehitellään omia ideoita. Käyttäjäpäivät koetaan tärkeiksi, silloin voidaan kartoittaa tarvetta uusille piirteillä ja asiakkaiden maksuhalukkuutta. Tuotekehityksen lähtökohtana ovat aiemmat versiot, joiden päälle uudistukset rakennetaan.

4.2 Laadulliset ominaisuudet

Laadullisten ominaisuuksien arviointi on hankalaa. Laatuattribuuttien arviointi (kuten esimerkiksi suorituskyky, käytettävyys, turvallisuus, ylläpidettävyys, siirrettävyys, konfiguroitavuus) koettiin tärkeäksi. Laadun mittaus tapahtuu käyttäjäasteella saadun palautteen perusteella. Tuotetta on pyritty testaamaan työkaluilla. Tällä hetkellä testaus

tapahtuu pääasiassa koodauksen yhteydessä. Yritys pyrkii toiminnoissaan sisäiseen asiakkuuteen, jolla uskotaan parannettavan laatua. Toimitusprosessin toimivuus on testattu. Tavoitteena päästä siihen, että vain asiakastoimituksen yhteydessä testataan toimitettu asennettu tuote. Laatuattribuuttien toteutumisen arviointi ennakolta perustuu tällä hetkellä kokemusperäiseen tietoon. Testausmenetelmiä arvioinnin parantamiseksi kehitellään.

4.3 Laitevaatimukset

Ohjelmiston laitevaatimukset ovat seuraavat:

Palvelin (minimi 25-500 käyttäjälle)

- Intel Celeron Suoritin 850MHz, 128Kt välimuisti / vastaava
- 512MB SDRAM
- 10/100 Mbps verkkokortti
- kiintolevy nro.1: 20Gt IDE / SCSI
  kiintolevy nro.2: 20Gt IDE / SCSI
  mieluummin raid 5 konfiguraatio, ide levyillä raid 0
- suositus varmuuskopiointiin: DLT –asema tai levyvarmistin

Työasema

- Normaali toimisto-PC varustettuna verkkoyhteydellä
- ProIntra tukee näytön resoluutiota 800 x 600 – 1600 x 1200

4.4 Ohjelmistovaatimukset

Palvelin

- Microsoft Windows 2000 Server
- Microsoft Internet Information Server 5.0
- Borland Interbase 6 Server tietokanta
- Virustorjunta
- Mahdollinen varmuuskopiointiohjelmisto, jos ei käytetä NT:n omaa varmistusohjelmistoa.

Työasema

- Normaali toimisto-PC varustettuna Internet Explorer 4.01 (Suositus IE 6.0, sisältösivueditori vaatii IE 5.5 tai uudemman) tai Netscape 4.08 tai uudemmat versiot.
- Single sign-on –ominaisuus vaatii Internet Explorer –selaimen käyttöä ja Windows-käyttöjärjestelmää.

5 Ongelma

Ohjelmiston toimintaan ollaan tyytyväisiä. Ohjelmiston laadulliset ominaisuudet ovat kohdallaan. Ohjelmisto on käyttöjärjestelmäriippuvainen. Tämä saattaa karsia osan asiakkaista. Lisäksi arkkitehtuuria kehittämällä ohjelmiston skaalautuvuus raskaampiin

sovelluksiin paranee. Ohjelmiston kehittämiseen vaikuttavat asiakaskunnan segmentointi eli se kenelle ja minkä kokoisille organisaatioille ohjelmistoa aiotaan myydä.

6 Ratkaisu

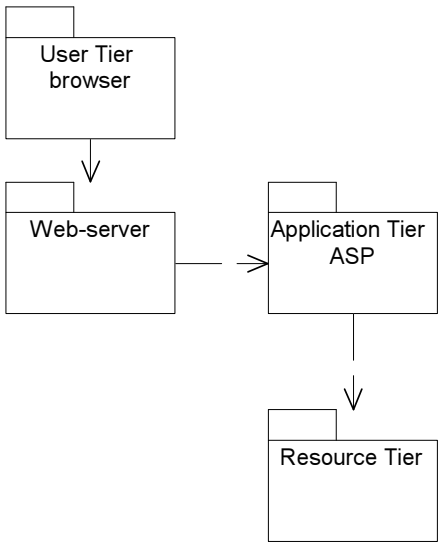
Parhaillaan mietitään ohjelmistolle jatkokehityshanketta. Tuleva versio 4 pohjautunee joko Java-pohjaiseen J2EE-arkkitehtuuriin tai Microsoftin .NET-arkkitehtuuriin. J2EE-arkkitehtuurin etuina ovat laitteistoriippumattomuus, mahdollisuus hyödyntää komponenttimarkkinoita sekä jo jonkin aikaan tuotannossa ollut alusta. Microsoftin etuina voidaan pitää Microsoft-yhteensopivuutta sekä aiemman projektin dokumentaation uudelleenkäytettävyyttä.

7 Järjestelmäintegraatio

Järjestelmä tarjoaa yhteyden muihin järjestelmiin XML-pohjaisesti SOAP-määrittelyjen mukaisesti.

8 Ohjelmistoarkkitehtuuri

Ohjelmisto on toteutettu 3-kerrosarkkitehtuurilla.



Kuva 1. viitearkkitehtuuri

Kuvassa 1 on kuvattu arkkitehtuurin viitemalli käyttäen UML:n mukaista pakettikaavioesitystä. Näkökulma on arkkitehtuurin kerrosmalli, jossa asiakkaan käyttöliittymä kuvataan selainkomponentilla. Liiketoimintakerroksessa on kuvattuna web-palvelin sekä tarvittava toiminnallisuus omana komponenttinaan. Resurssikerrokseen on sijoitettu tietokannanhallintajärjestelmä.

9 Uuden version toteutus ja kehitettävää

Uusi versio toteutetaan vähitellen korvaamalla moduuli kerrallaan uudella. Tuotteen eliniäksi arvioidaan 5-10 vuotta. Dokumentaation kehittäminen koettiin tärkeäksi. Testaustoiminnan kehittäminen ja laadun parantamistoimenpiteiden kehittäminen koettiin tärkeäksi.

LIITE 5. Laatuattribuuttien painoarvot

	attribuutti		1	2	3	4	5	6	7	8	9	10
1	suorituskyky (performance)											
2	ylläpidettävyys (maintanance)											
3	siirrettävyys (portability)											
4	luotettavuus (reliability)											
5	toipuminen (recovery)											
6	turvallisuus (safety)											
7	tietoturva (security)											
8	salaus (encrytion)											
9	samanaikaisuuden hallinta (concurrency)											
10	konfiguroitavuus staattinen (configuration static)											
11	konfiguroitavuus dynaamineen (configuration dynamic)											
12	käytettävyys (usability)											
13	joustavuus (flexibitity)											
14	skaalautuvuus (scalability)											
15	pysyvyys (persistence)											
16	transaktioidenhallinta (transaction management)											
17	testattavuus (testability)											
18	modulaarisuus (modularity)											

LIITE 6. Laatuattribuuttien ominaisuuksien kartoitus

Kuinka laatuattribuutin ominaisuudet asetettiin ja millaisiksi ne muodostuivat?

	attribuutti	kuinka ?	millainen ?
1	suorituskyky (performance)		
2	ylläpidettävyys (maintanance)		
3	siirrettävyys (portability)		
4	luotettavuus (reliability)		
5	toipuminen (recovery)		
6	turvallisuus (safety)		
7	tietoturva (security)		
8	salaus (encryption)		
9	samanaikaisuuden hallinta (concurrency)		
10	konfiguroitavuus staattinen (configuration static)		
11	Konfiguroitavuus dynaamineen (configuration dynamic)		
12	Käytettävyys (usability)		
13	joustavuus (flexibitity)		
14	Skaalautuvuus (scalability)		
15	pysyvyys (persistence)		
16	transaktioidenhallinta (transaction management)		
17	testattavuus (testability)		
18	modulaarisuus (modularity)		

LIITE 7. Arkkitehtuurien mallijärjestelmiä (Pattern language)

Sisällysluettelo

1 Yleistä 175
2 Taulukko 1. mallijärjestelmä (Pattern language) arkkitehtuurien luokittelu ja taksonomia esimerkki [BuR96] 175
3 Taulukko 2. Mallijärjestelmä samanaikaisten ja verkotettujen järjestelmien rakentamiseksi [ScR01 s.525] 176
4 Taulukko 3. Java Enterprise Design Patterns [Gra01] 177
5 J2EE-mallijärjestelmä (Core J2EE Pattern Catalog) 178
6 Taulukko 4. Mary Shaw:n luokittelemat perusarkkitehtuurit [www007] 179

1 Yleistä

Tässä liitteessä on esitetty esimerkkinä neljä arkkitehtuurien mallijärjestelmää (Taulukko 1-4). Mallien luokitteluja on muitakin esimerkiksi Microsoftin [www018] ja Sunin sivustoilta löytyy malliesimerkkejä (kuva 1) [www019].

2 Taulukko 1. mallijärjestelmä (Pattern language) arkkitehtuurien luokittelu ja taksonomia esimerkki [BuR96]

Luokittelu	Arkkitehtuurimalli
Jäsennysarkkitehtuurit (From Mud to Structure)	Kerrosarkkitehtuuri (Layer Architechture) Putket ja suodattimet (Pipes and Filters) Liitutaulu (Blackboard)
Hajautusarkkitehtuurit (Distributed Systems)	Välittäjä (Broker)
Vuorovaikutusarkkitehtuurit (Interactive Systems)	MVC PAC
Muokattavat (Adaptive Systems)	Microkernel Reflection dynaamisesti mukautuva arkkitehtuuri
Rakenteinen osiinjako (Structural Decomposition)	Whole-Part
Työn organisointi (Organisation of Work)	Master-Slave
Saantikontrolli (Access Control)	Proxy
Hallinta (Management)	Command Processor View Handler Counter Pointer
Kommunikointi (Communication)	Forward-Receiver Client-Dispatcher-Server Publish-Subscribe

3 Taulukko 2. Mallijärjestelmä samanaikaisten ja verkotettujen järjestelmien rakentamiseksi [ScR01 s.525]

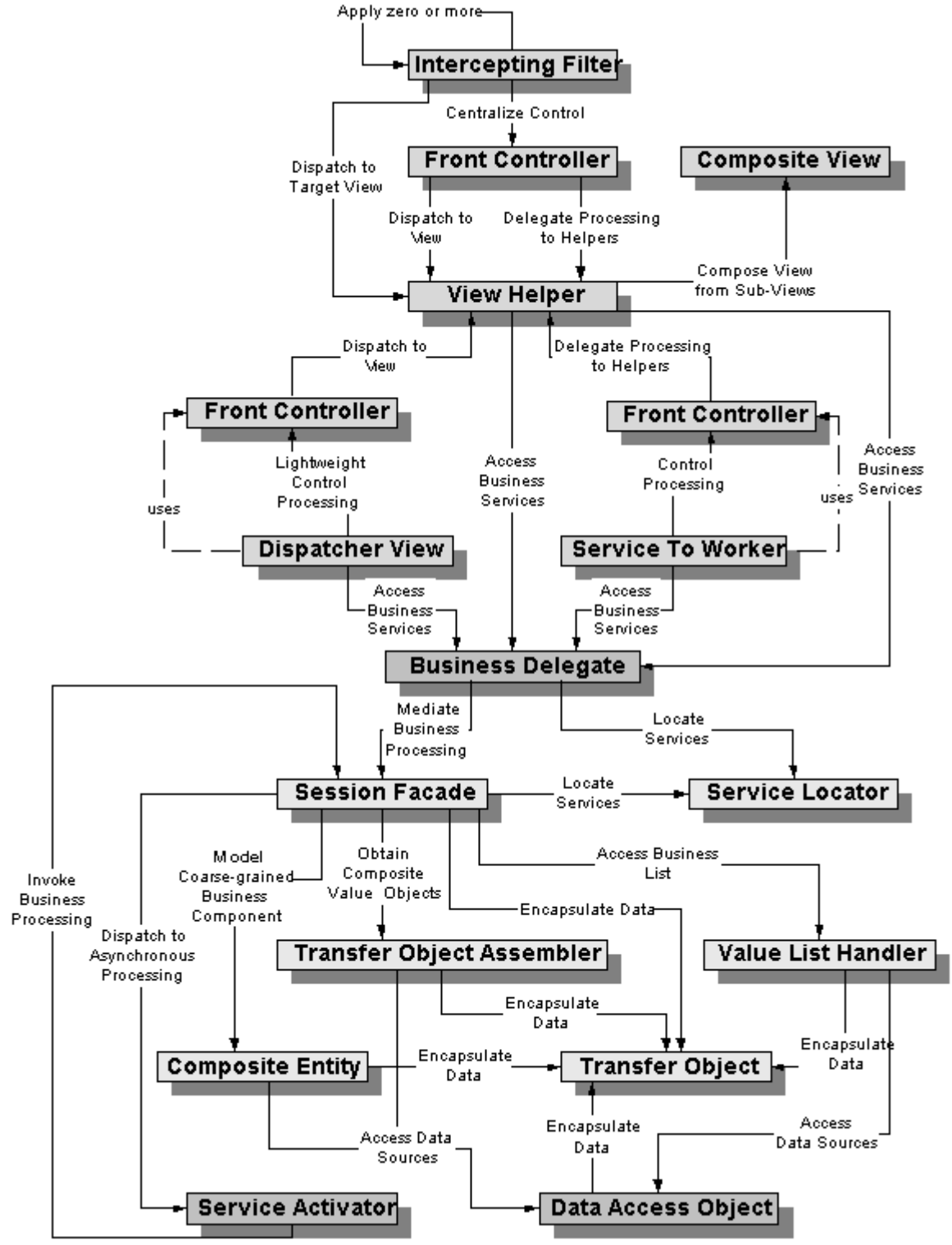
Luokittelu	Arkkitehtuurimalli (Architectural Pattern)	Suunnittelumalli (Design Pattern)	Idioomi (Idiom)
Perusarkkitehtuurit (Base-line Architecture)	Broker Layers Microkernel		
Kommunikaatio (Communication)	Pipes and Filters	Abstract Session Command Processor Forwarder-Receiver Observer Remote Operation Serializer	
Alustus (Initialization)		Activator Client-Dispatcher-Server Evictor Locator Object Lifetime Manager	
Palveluun pääsy ja konfiguraatio (Service Access and Configuration)	Interceptor	Component Configurator Extension Interface Half Object plus Protocol Manager-Agent Proxy Wrapper Facade	
Tapahtuman käsittely (Event Handling)	Proactor Reactor	Acceptor-Connector Asynchronous Completion Token Event Notification Observer Publish-Subscribe	
Synkronointi (Synchronization)	Object Synchronizer	Balking Code Locking Data Locking Guarded Suspension Double-Checked Locking Optimization Reader/Writer Locking Specific Notification Strategized Locking Thread-Safe Interface	Scoped Locking
Samanaikaisuus (Concurrency)	Half-Sync/Half-Async Leader/Followers	Active Object Master-Slave Monitor Object Producer-Consumer Scheduler Two-Phase Termination Thread-Specific Storage	

4 Taulukko 3. Java Enterprise Design Patterns [Gra01]

Luokittelu	Suunnittelumalli
Transaktiomallit (Transaction patterns)	ACID transaction Composite transaction Two phase commit Audit trail
Hajautetut arkkitehtuurimallit (Distributed architecture patterns)	Shared object Object request broker Object replication Redundant independent objects Prompt repair Mobile agent Demilitarized zone Process pairs
Hajautetut tietojenkäsittelymallit (Distributed computing patterns)	Object identifier Registry Protection proxy Publish-subscribe Retransmission Mailbox Heavyweight/lightweight Heartbeat Connection multiplexing
Samanaikaisuuden hallintamallit (Concurrency patterns)	Session object Lock file Static locking order Optimistic concurrency Thread pool Ephemeral cache item Transaction state stack
Väliaikaisuudenhallintamallit (Temporal patterns)	Time server Versioned object Temporal property
Tietokantamallit (database patterns)	Persistence layer CRUD Stale object Type conversion IsDirty Lazy Retrieval

5 J2EE-mallijärjestelmä (Core J2EE Pattern Catalog)

Kuvassa 1 on esitetty J2EE:n mukainen mallijärjestelmä [www019].



Kuva 1. J2EE-mallijärjestelmä (Core J2EE Pattern Catalog)

6 Taulukko 4. Mary Shaw:n luokittelemat perusarkkitehtuurit [www007]

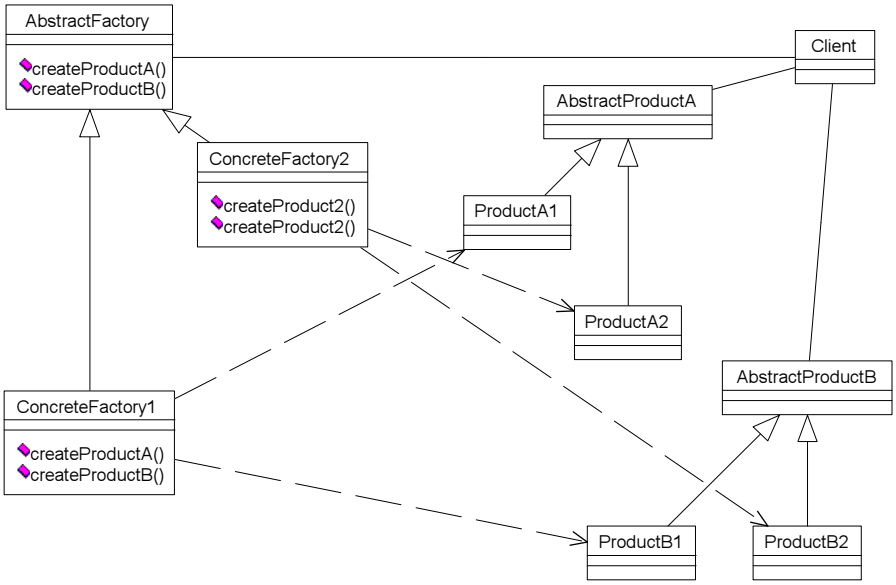
Arkkitehtuurimalli	Mallin selite
Putki ja suodatin (Pipeline architectural pattern)	Malli sovellusten välisen datavirran käsittelyyn.
Tietoabstraktio (Data abstraction architectural pattern)	Malli hoitaa managereiden avulla esitettävän informaation suojaamisen ja tunnistamisen. Komponentteina tässä esiintyy managereita ja konnektoreina proseduurikutsuja.
Kommunikoivat prosessit (Communicating processes architectural pattern)	Malli soveltuu prosessien välisten viestien järjestämiseen.
Implisiittinen kutsuminen (Implicit invocation architectural pattern)	Malli soveltuu sellaisten prosessien tapahtumien välittämiseen, jotka ovat rekisteröityneet kuuntelijoiksi (vrt. Publish-Subscribe).
Tietovarasto (Repository architectural pattern)	Tietovarasto soveltuu sellaisten sovellusten toteutusmalliksi, joiden tarkoituksena on keskitetysti ylläpitää ja jakaa tietoa.
Tulkki (Iinterpreter architectural pattern)	Tulkki toimii siltana eri elementtien välillä tulkaten tiedon vastaanottajalle sopivaksi.
Pääohjelma/aliohjelma (Main program and subroutines architectural pattern).	Malli muodostaa hierarkian ohjelman osien välille
Kerros (Layered architectural pattern)	Malli luokittelee palveluiden välille hierarkian.

LIITE 8. ARKKITEHTUURI- JA SUUNNITTELUMALLEJA STAATTINEN KUVAUS

Abstraktitehdas (Abstract Factory) –suunnittelumalli 95
Acceptor-Connector –suunnittelumalli 95
Ainokainen (Singleton)-suunnittelumalli 95
Edeltäjä-Seuraaja (Leader/Followers) –arkkitehtuurimalli 98
Edustaja (Proxy)-suunnittelumalli 99
Julkisivu (Facade tai Wrapper Facade)-suunnittelumalli 103
Half-Sync/Half-Async –arkkitehtuurimalli 103
Iteraattori (Iterator) –suunnittelumalli 106
Komento (Command)-suunnittelumalli 106
Komponentin konfiguraattori (Component configurator)-suunnittelumalli 106
Kuorruttaja (Decorator)-suunnittelumalli 109
Monitor Object –suunnittelumalli 110
MVC-arkkitehtuurimalli 113
Operaatiorunko (Template Method)-suunnittelumalli 113
Proactor-arkkitehtuurimalli 114
Reactor-arkkitehtuurimalli 117
Rekursiokooste (Composite)- suunnittelumalli 121
Silta (Bridge)-suunnittelumalli 123
Sovitin (Adapter) –suunnittelumalli 125
Strategia (Strategy) –suunnittelumalli 125
Tarkkailija (Observer tai Publish-Subscribe)-suunnittelumalli 128
Tehdasmetodi (Factory Method)-suunnittelumalli 128
Tila (State)-suunnittelumalli 128
Torjuntahävittäjä (Interceptor)-arkkitehtuurimalli 130
Välittäjä (Broker)-arkkitehtuurimalli 130
Whole-Part –suunnittelumalli 132

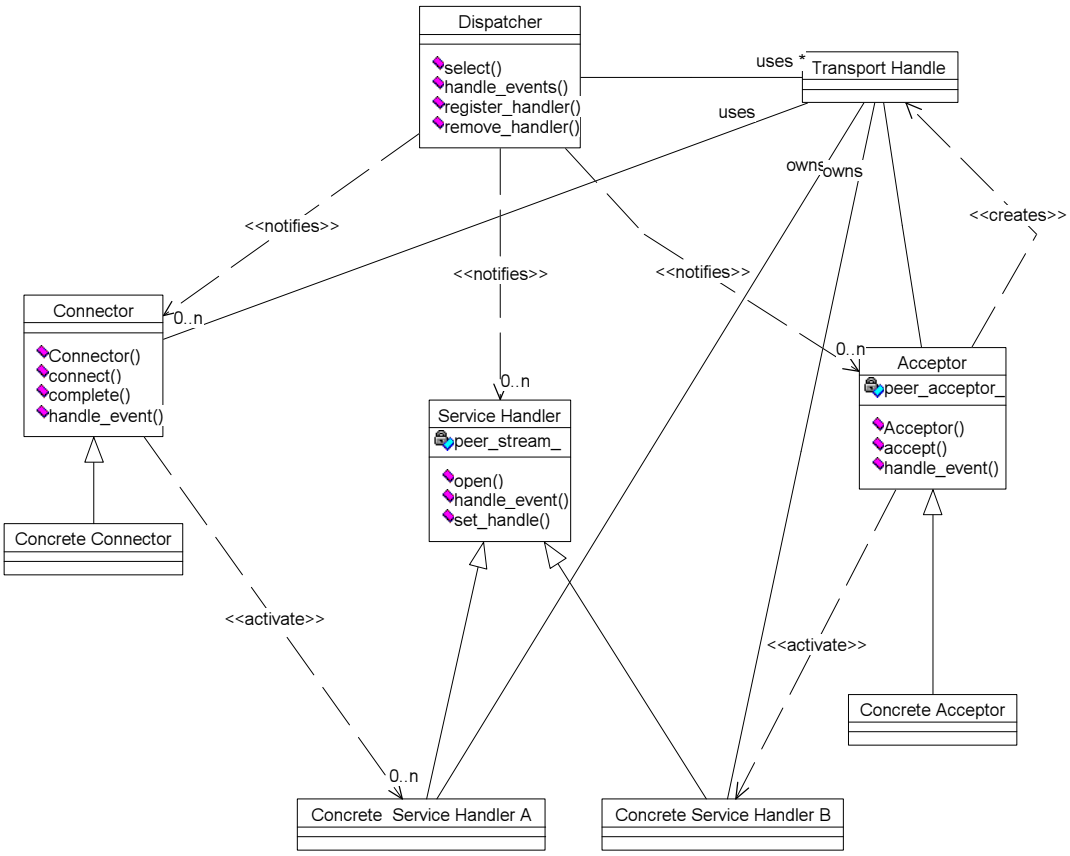
Abstraktitehdas (Abstract Factory) –suunnittelumalli

Tuottaa rajapinnan, jolla luodaan toisiinsa liittyvien olioiden muodostamia olioperheitä määrittelemättä olioiden konkreettisia luokkia [GaH95 s. 87].



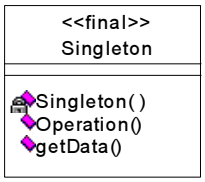
Acceptor-Connector –suunnittelumalli

Acceptor-Connector Design Pattern irrottaa yhteyden ja alustuksen yhteistyötä tekevistä vertaispalveluista (peer services) verkotetussa systeemissä siitä prosessoinnista, jonka ne esittävät kerran liityttyään ja alustettuaan. Tämä malli sallii sovellusten konfiguroida yhteystopologioitaan sillä tapaa, että ne ovat suuressa määrin riippumattomia tarjoamistaan palveluista. Malli voidaan kerrostaa Reactor- tai Proactor-mallien huipulle hoitamaan tapahtumia, jotka liittyvät yhteyden perustamiseen palveluiden välillä [ScR01 s. 285].



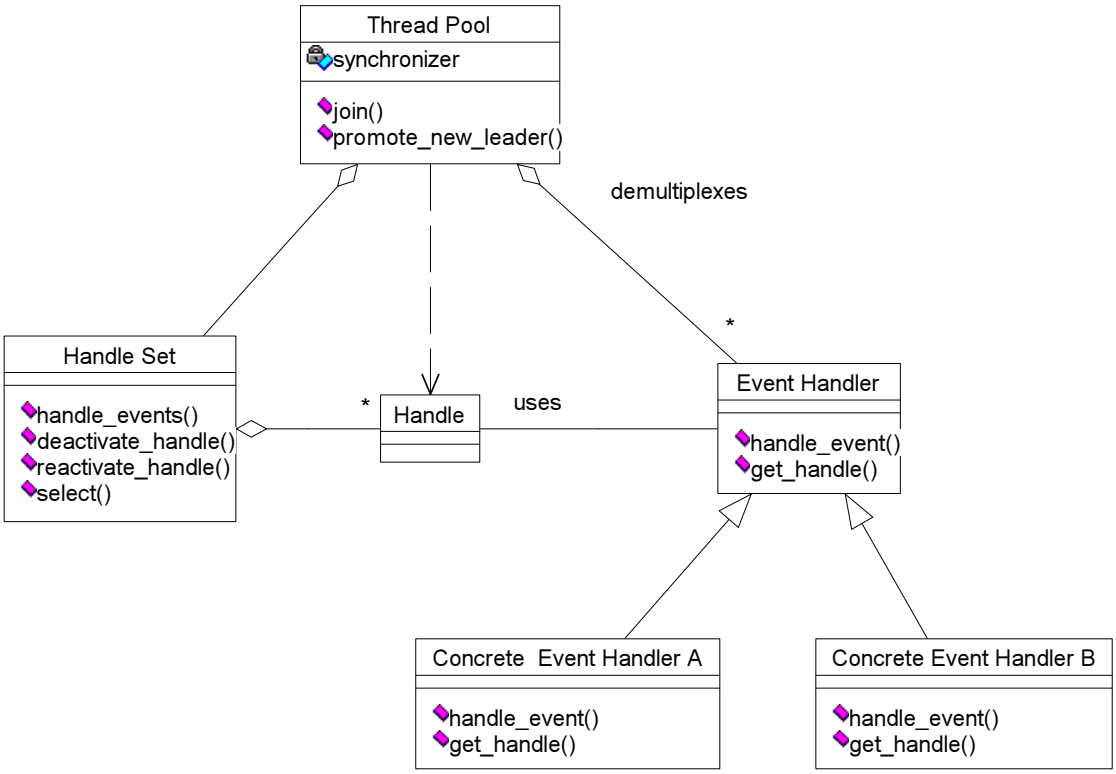
Ainokainen (Singleton)-suunnittelumalli

Varmistaa, että luokasta luodaan vain yksi ilmentymä[GaH95 s. 127], [Imm03].



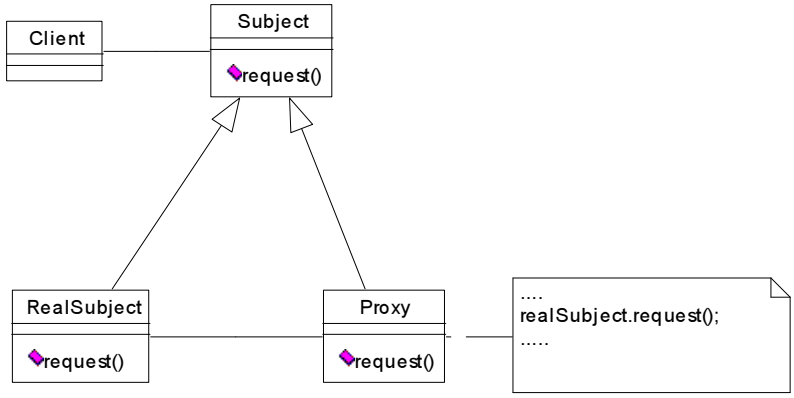
Edeltäjä-Seuraaja (Leader/Followers) –arkkitehtuurimalli

Leader/Followers Architectural Pattern tarjoaa tehokkaan samanaikaisuusmallin, jossa useat säikeet jakavat vuorollaan joukon tapahtumanlähteitä. Malli havaitsee, purkaa, lähettää ja prosessoi tapahtumanlähteen toimittamia palvelupyyntöjä. Tätä mallia voidaan käyttää Half-sync/Half-async- ja Active Object -mallien asemesta suorituskyvyn parantamisessa silloin, kun ei ole olemassa rajoituksia synkronoinnissa prosessoitaessa kyselyjä säiepoolista [ScR01 s.447].



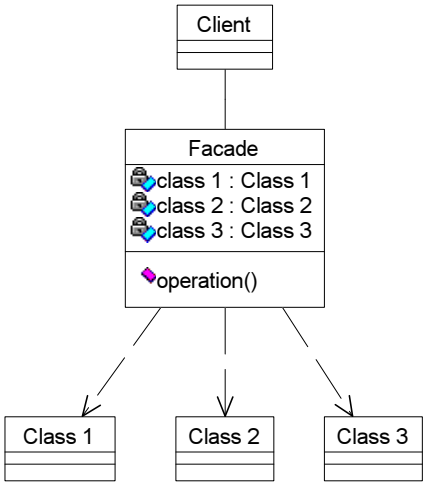
Edustaja (Proxy)-suunnittelumalli

Tuottaa oliolle korvikkeen tai paikanpitäjän, joka kontrolloi olioon kohdistuvia pyyntöjä [GaH95 s. 207], [Imm03].



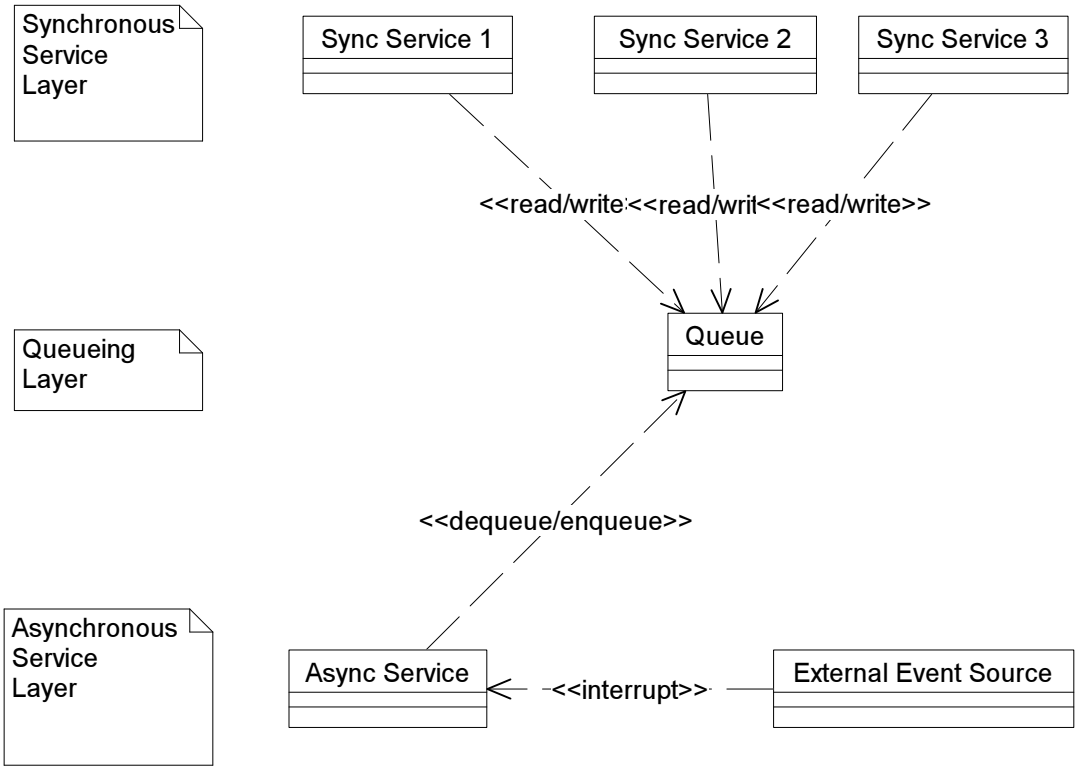
Julkisivu (Facade tai Wrapper Facade)-suunnittelumalli

Tarjoaa yhtenäisen rajapinnan alijärjestelmän rajapintojen joukolle [GaH95 s. 185].



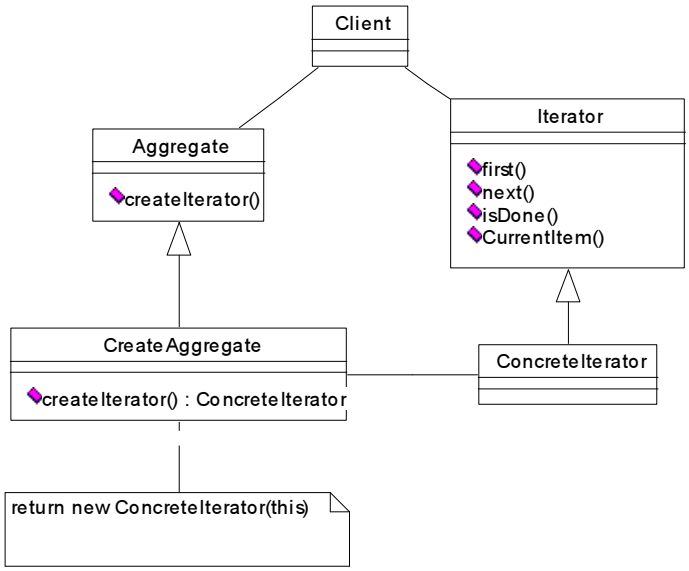
Half-Sync/Half-Async –arkkitehtuurimalli

Half-Sync/Half-Async Architectural Pattern irrottaa asynkronisen ja synkronisen prosessoinnin samanaikaisessa systeemissä, yksinkertaistaa ohjelmointia vähentämättä suorituskykyä kohtuuttomasti. Malli esittelee kaksi sisäisesti kommunikoivaa tasoa, yksi asynkronisille ja yksi synkronisille palveluille. Jonotuskerros (Queuing layer) välittää kommunikaatiota tasojen välissä [ScR01 s. 423].



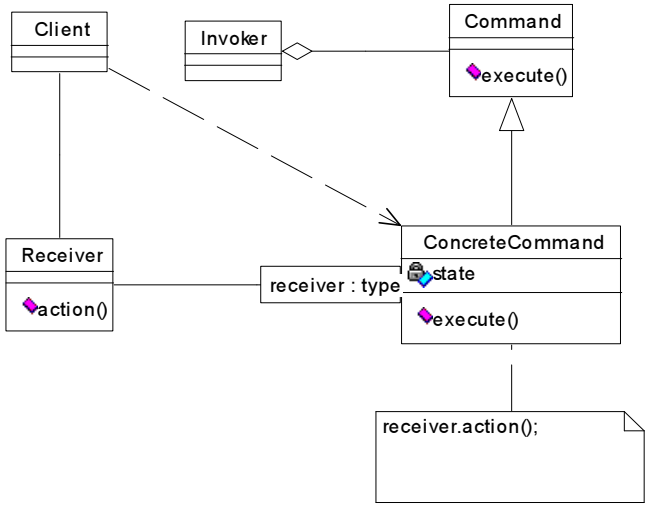
Iteraattori (Iterator) –suunnittelumalli

Tarjoaa tavan kokoelman läpikäymiseen paljastamatta kokoelman sisäistä esitystapaa [GaH95 s. 257].



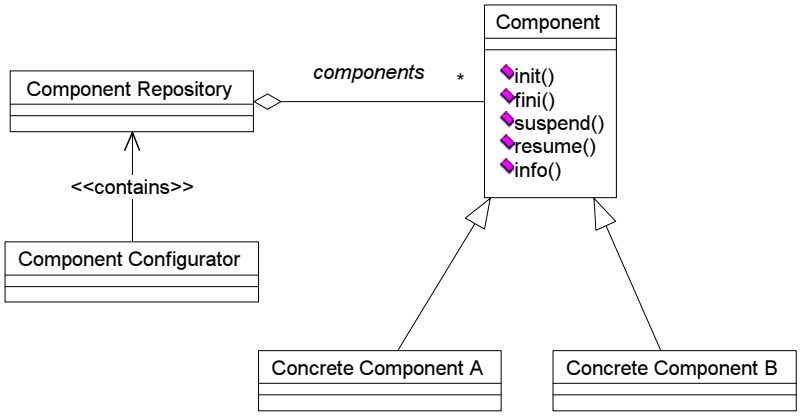
Komento (Command)-suunnittelumalli

Kapseloi olion pyynnöksi [GaH95 s. 233].



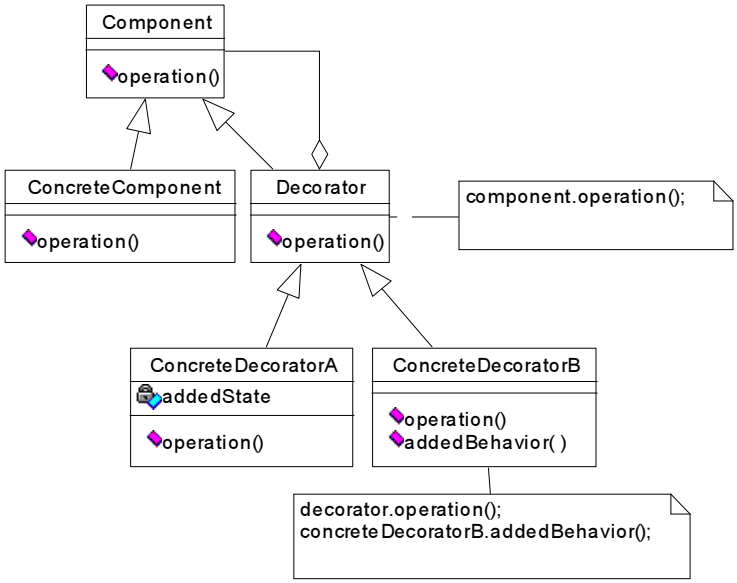
Komponentin konfiguraattori (Component configurator)-suunnittelumalli

Komponentin konfiguraattori (Component configurator)-mallin avulla hallitaan komponenttikehykseen kuuluvien komponenttien alustaminen, prosessointi ja toiminnan lopettaminen dynaamisesti[ScR01 s. 75].



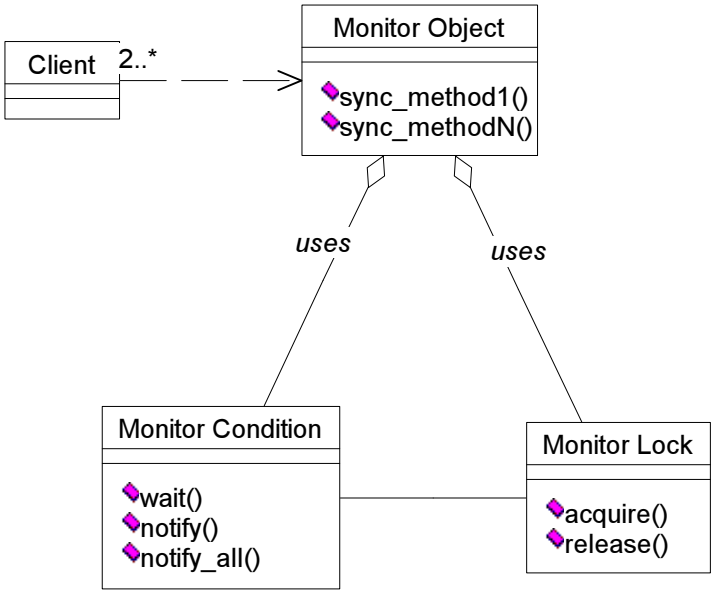
Kuorruttaja (Decorator)-suunnittelumalli

Lisää oliolle dynaamisesti uusia vastuita [GaH95 s. 175], [Imm03].



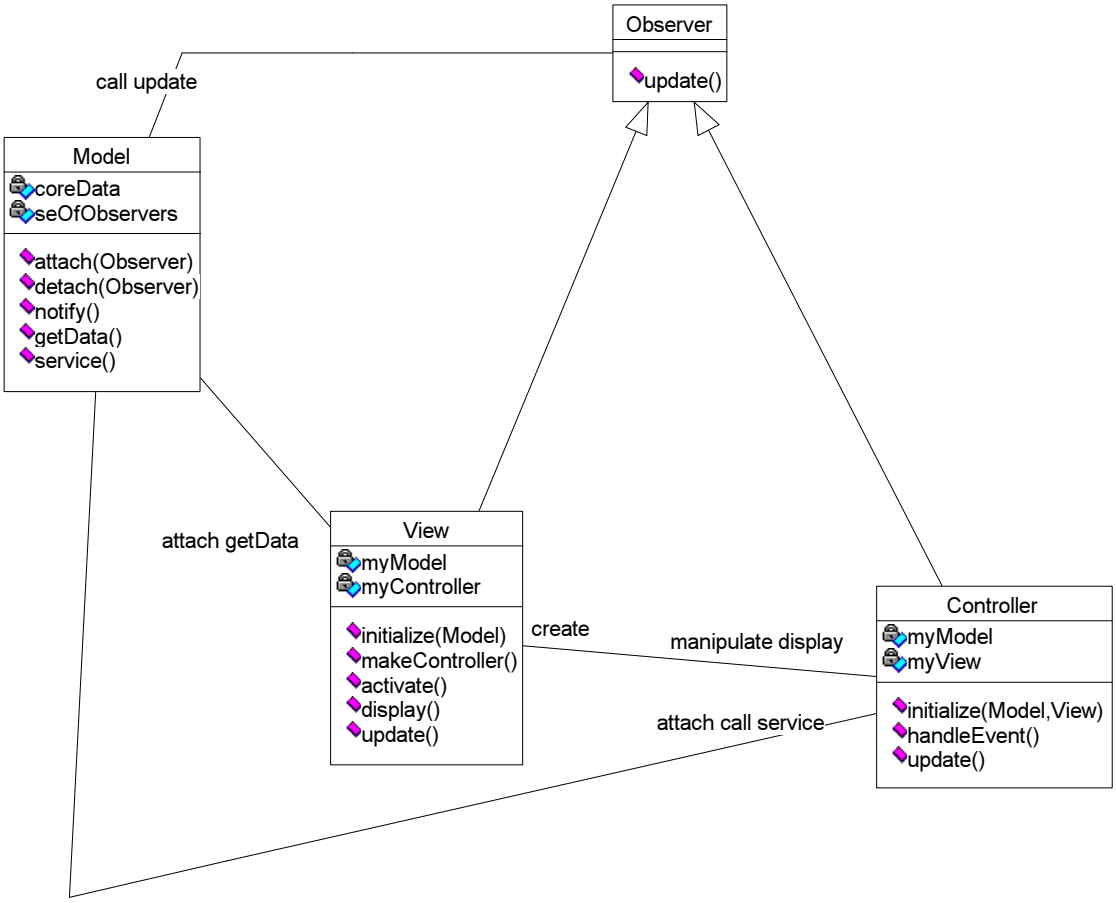
Monitor Object –suunnittelumalli

Monitor Object Design Pattern synkronoi samanaikaisen metodin toteutuksen varmistaakseen, että vain yksi metodi ajallaan ajaa oliossa. Se myös sallii olion metodien ja toteutusjaksonsa yhteistoiminnallisesti. Tunnettu myös Thread-safe Passive Object [ScR01 s. 399].



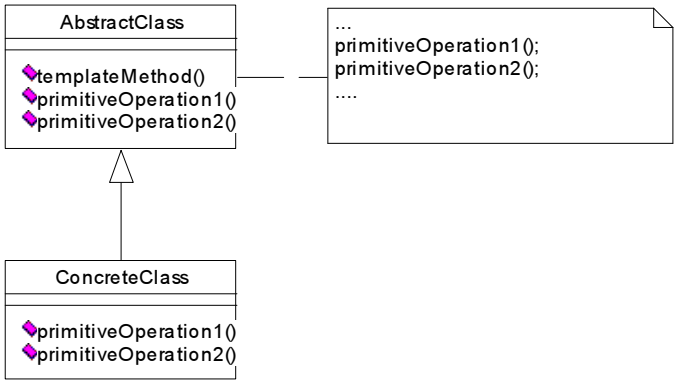
MVC-arkkitehtuurimalli

Vuorovaikutteiset sovellukset, joilta vaaditaan joustavaa käyttöliittymää [BuR96 s. 125].



Operaatiorunko (Template Method)-suunnittelumalli

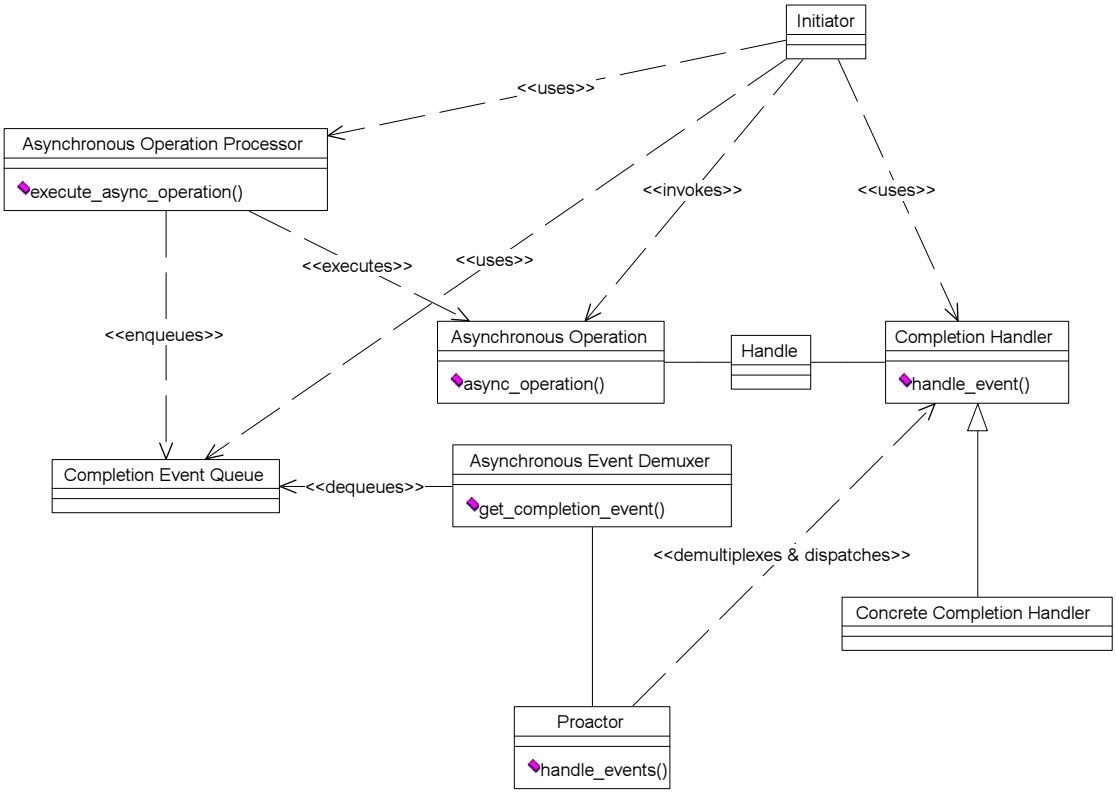
Määrittelee algoritmin rungon, mutta jättää osia aliluokkien täydennettäväksi [GaH s. 325].



Proactor-arkkitehtuurimalli

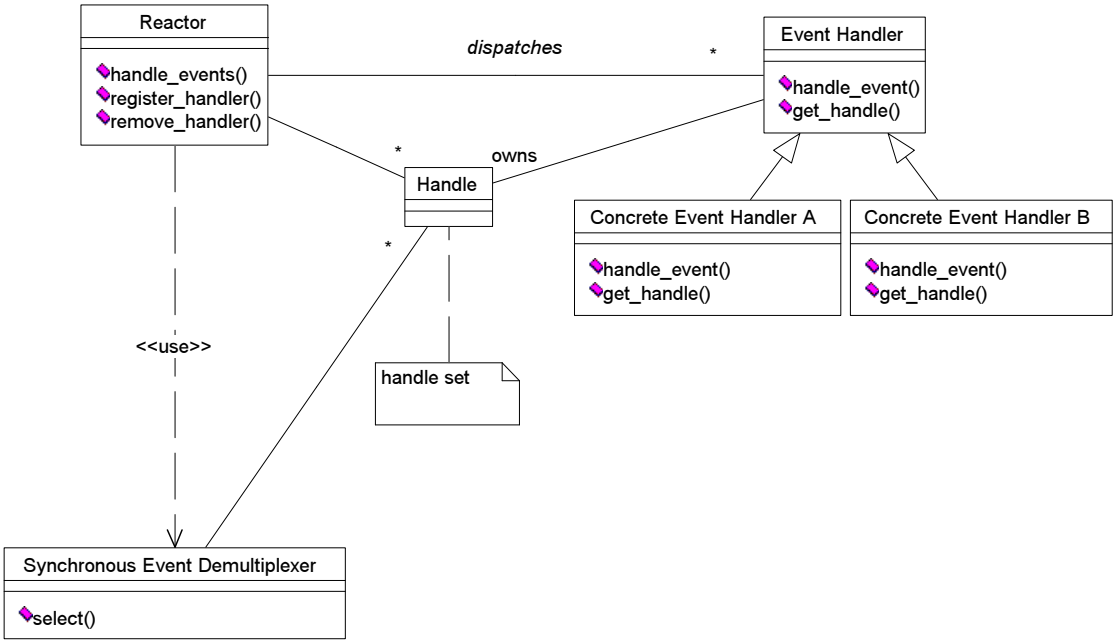
Proactor-arkkitehtuurimalli tukee asynkronista I/O:ta tehokkaasti sallimalla tapahtuma-ohjattujen sovellusten tehokkaasti purkaa ja lähettää palvelupyyntöjä, jotka laukaustaan asynkronisten operaatioiden päättäminä. Tämä malli tarjoaa suorituskykyä samanaikaisissa prosesseissa ilman, että siitä koituu muita hankaluuksia.

Proactor-malli aloittaa kontrollin ja tietovirran sovelluksessa alustamalla yhden tai useamman asynkronisen operaatiopyynnön "proaktiivisesti" asynkronisena operaatioprosessorina. Kun nämä asynkroniset operaatiot ovat valmiit, asynkroninen operaatioprosessori ja suunniteltu Proactor-komponentti tekevät yhteistyötä purkaakseen valmiit tulostapahtumat yhteistyötä tekeville tulostenkäsittelijöille ja lähettävät näiden käsittelijöiden koukkumetodit. Sen jälkeen, kun päättämistapahtuma on prosessoitu, tulostenkäsittelijä voi alustaa uuden asynkronisen kyselyn proaktiivisesti [ScR01 s. 176].



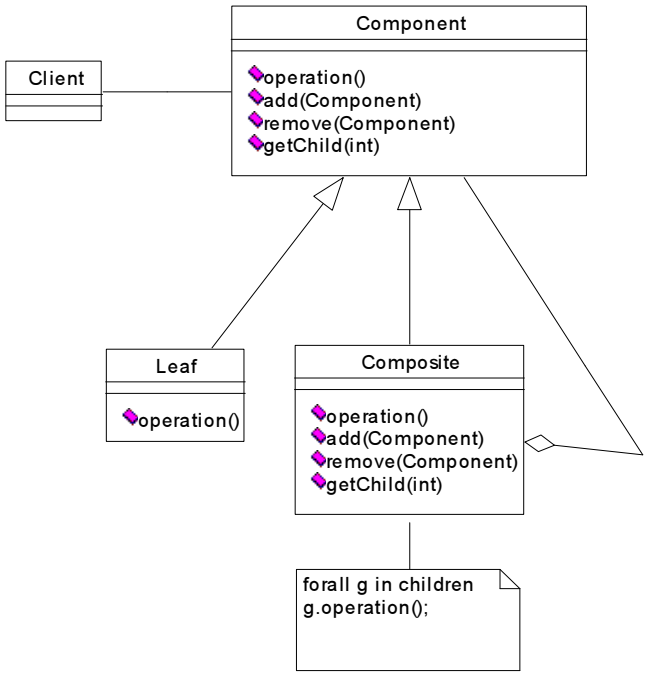
Reactor-arkkitehtuurimalli

Reactor-arkkitehtuurimalli sallii sovellusten purkaa asiakaspyyntöjä tapahtumaohjatusti ja lähettää eteenpäin asiakkaiden esittämät palvelupyynnöt. Reactor-malli kääntää vuonohjauksen (flow control) sovelluksessa Hollywood-periaatteen mukaisesti. Reactor-komponentti odottaa tapahtumaa synkronisesti. Sen jälkeen Reactor purkaa tapahtuman ja liittää siihen tapahtumankäsittelijän, joka saa puretun tapahtuman käsiteltäväkseen ja lopuksi lähettää sopivan koukkumetodin (hook method) tapahtumankäsittelijälle. Sovellussuunnittelijan tarvitsee ainoastaan toteuttaa konkreettinen tapahtumankäsittelijä. Kehittäjä voi uudelleenkäyttää Reactor-komponentin purkamis- ja lähettämismekanismia [ScR01 s. 176].

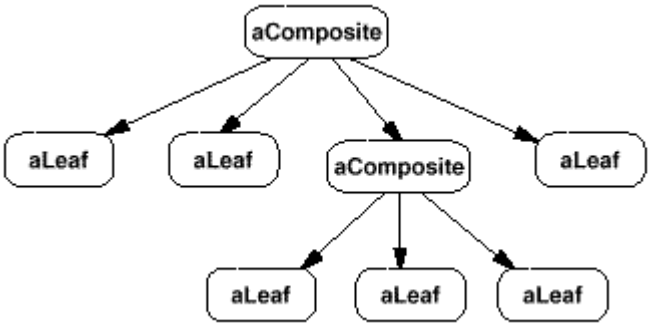


Rekursiokooste (Composite)- suunnittelumalli

Malli esittää oliot rekursiivisesti koostettuna puurakenteena (part-whole hierarchy) [GaH95 s. 163].

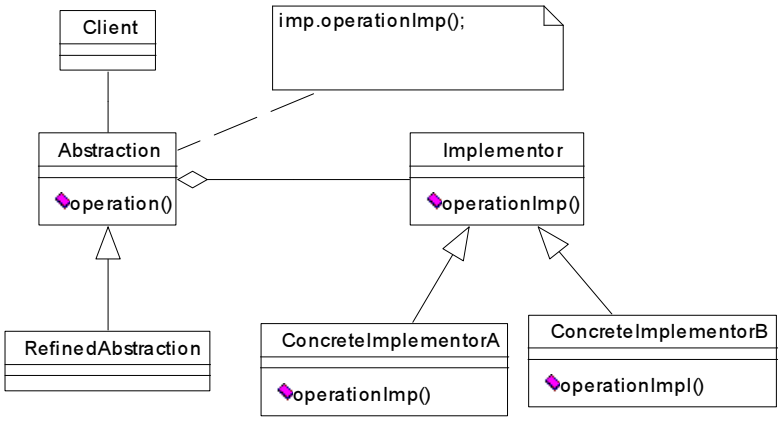


Tyypillinen oliorakenne voi näyttää tältä:



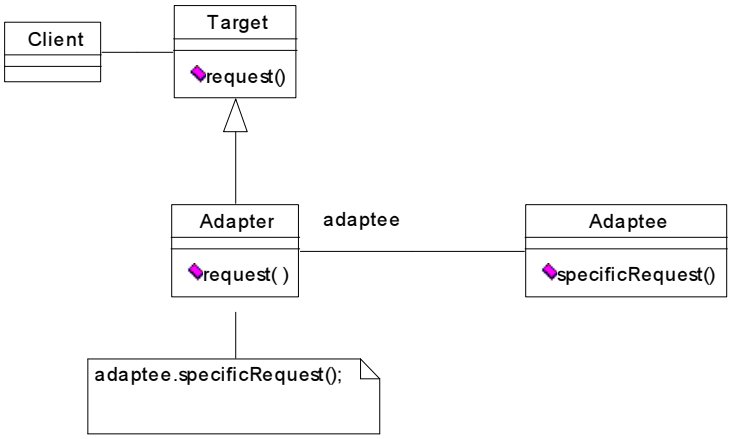
Silta (Bridge)-suunnittelumalli

Erottaa rajapinnan toteutuksesta, jolloin kumpaakin voidaan muuttaa toisistaan riippumatta [GaH95 s. 151].



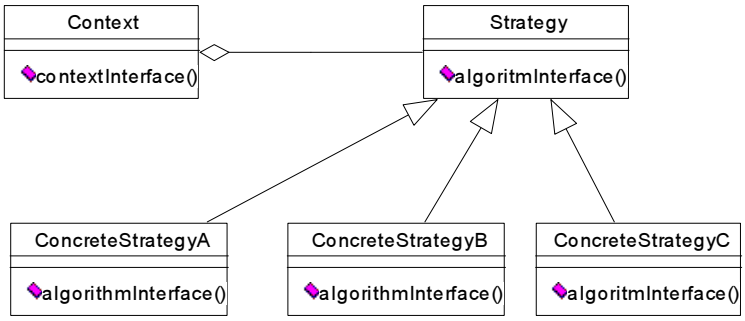
Sovitin (Adapter) –suunnittelumalli

Konvertoi rajapinnan toiseksi rajapinnaksi, joka vastaa sovelluksen tarpeita [GaH95 s. 139].



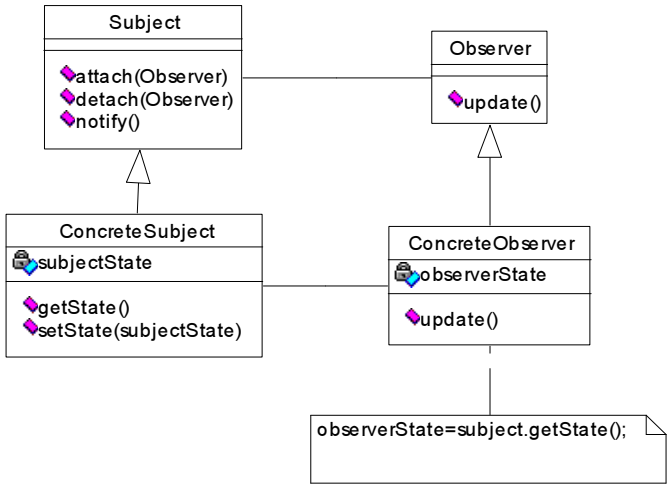
Strategia (Strategy) –suunnittelumalli

Määrittelee algoritmiperheen ja tekee niistä keskenään vaihdettavia [GaH95 s. 315].



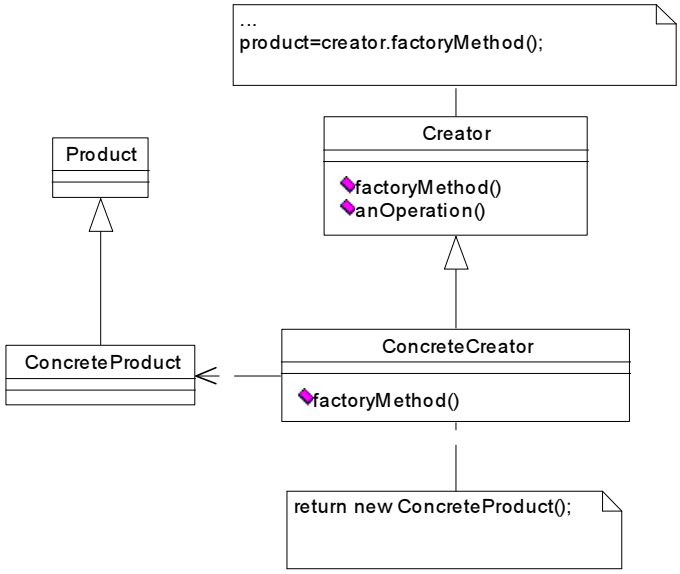
Tarkkailija (Observer tai Publish-Subscribe)-suunnittelumalli

Määrittelee olioiden välille yksi moneen –riippuvuuden siten, että kun yhden olion tila muuttuu, siitä riippuvat oliot saavat ilmoituksen ja päivittyvät automaattisesti [GaH95 s. 293].



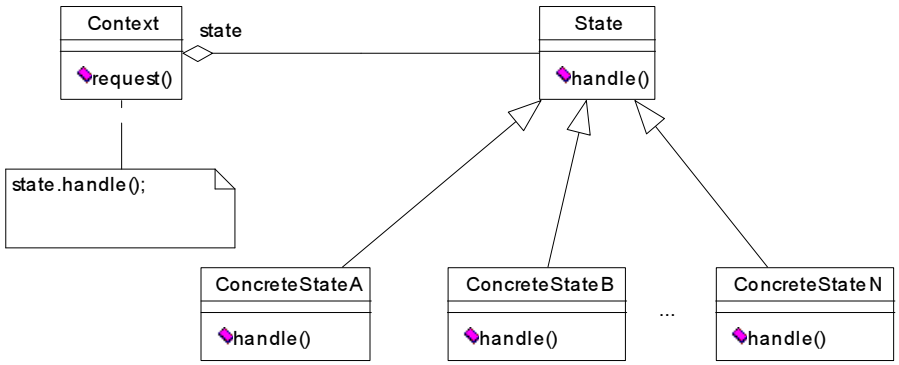
Tehdasmetodi (Factory Method)-suunnittelumalli

Määrittelee olioin luontioperaation kutsumuodon, mutta jättää aliluokkien tehtäväksi päättää, mistä luokasta ilmentymä luodaan [GaH95 s. 107].



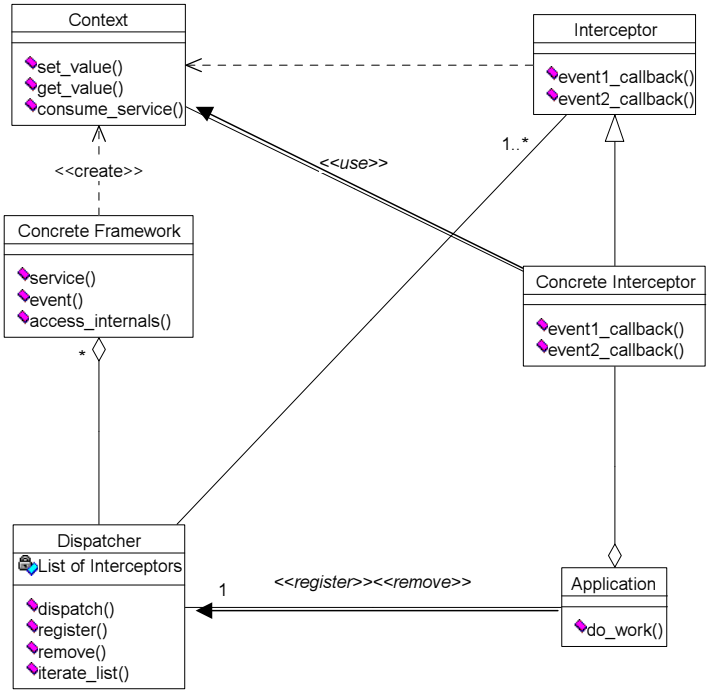
Tila (State)-suunnittelumalli

Malli saa olion muuttamaan käyttäytymistään, kun sen tila muuttuu [GaH95 s. 305].



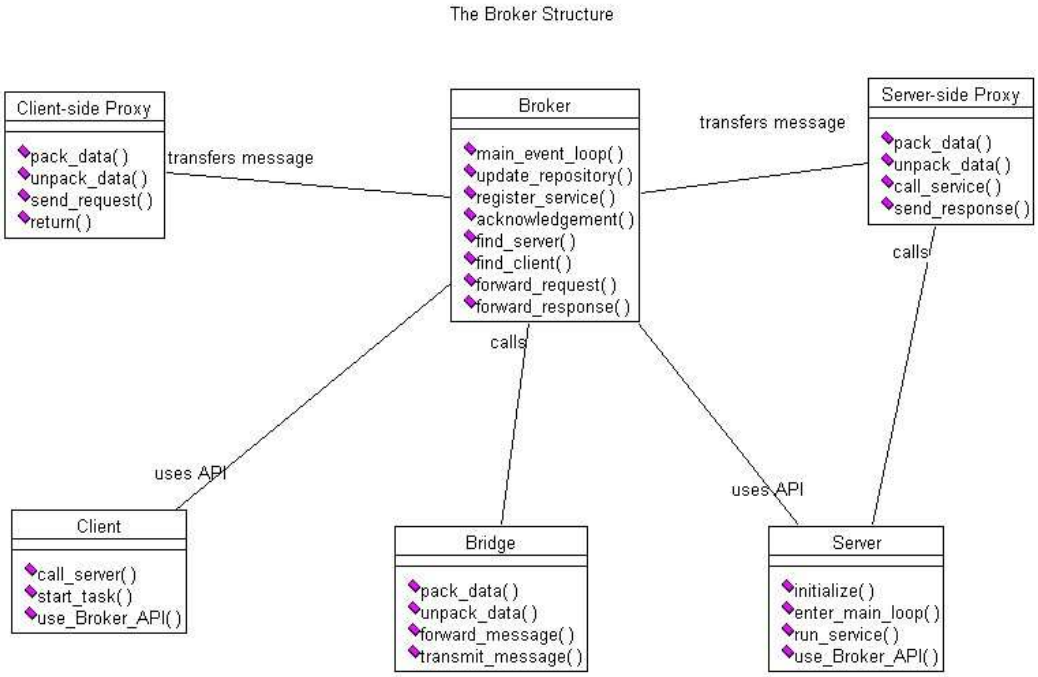
Torjuntahävittäjä (Interceptor)-arkkitehtuurimalli

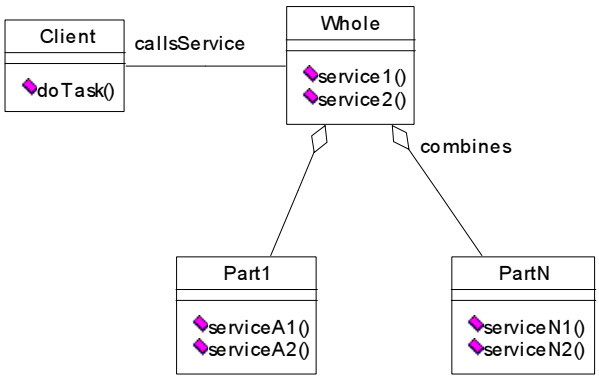
Torjuntahävittäjä (Interceptor)-arkkitehtuurimallin avulla voidaan kehykseen lisätä palveluja ja laukaista palvelu tapahtumaohjatusti [ScR01 s. 109].



Välittäjä (Broker)-arkkitehtuurimalli

Voidaan käyttää hajautetuissa järjestelmissä hajautetuissa komponenteissa muodostamaan vuorovaikutus etäkutsun avulla [BuR96 s. 99].



Whole-Part –suunnittelumalli

Mallin avulla voidaan komponentteja koostaan niin, että ne muodostavat yhden semanttisen yksikön [BuR96 s. 225].