Janne Hansen

OHJELMISTOARKKITEHTUURIT, ARKKITEHTONISET TYYLIT SEKÄ ARKKITEHTUURISUUNNITTELU

Tietojärjestelmätieteen
pro gradu - tutkielma
29.7.1999

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Hansen, Janne Juhani

Ohjelmistoarkkitehtuurit, arkkitehtoniset tyylit ja arkkitehtuurisuunnittelu /

Janne Hansen

Jyväskylä: Jyväskylän yliopisto, 1999.

107 s.

Tutkielma

Tässä tutkielmassa tarkastellaan ohjelmistoarkkitehtuureita. Vielä nykyäänkin ohjelmistojen arkkitehtuurisuunnittelun katsotaan olevan eräänlaista taidetta tai mustaa magiaa. Tutkielman tavoitteena onkin selvittää, mitä ohjelmistoarkkitehtuurilla, arkkitehtonisilla tyyleillä sekä arkkitehtuurisuunnittelulla varsinaisesti tarkoitetaan. Tavoitteena on myös antaa lukijalle kokonaisnäkemys ohjelmistoarkkitehtuureiden nykytutkimuksen tilasta sekä tutustuttaa lukija ohjelmistoarkkitehtuureiden keskeisimpiin sovelluskohteisiin.

Aiheeseen perehdytään sekä kirjallisuuden että osittain myös käytännön kokemuksen pohjalta. Tutkimusmenetelmä on käsitteellisteoreettinen, vaikka tutkielman jotkin osuudet ovatkin lähinnä kirjallisuustutkimusta.

Tutkielmassa pyritään selvittämään arkkitehtuurin suhteita niitä ympäröivään todellisuuteen – järjestelmän kehityksessä käytettyihin prosessimalleihin, suunnittelumenetelmiin, itse arkkitehtuuria kehittävään organisaatioon sekä kehitystyön eri sidosryhmiin.

AVAINSANAT: ohjelmistotekniikka, ohjelmistoarkkitehtuuri, arkkitehtoninen tyyli, arkkitehtuurisuunnittelu

KIITOKSET

Haluan kiittää InfoManager Oy:tä saamastani tuesta pro gradu - tutkielmani tekemiseen.

Suuret kiitokseni ovat ansainneet ohjaajani professori Markku Sakkinen, InfoManagerin ohjelmistotekniikan ammattilaiset (mm. Sampo Kuusi, Marko Lehtamo, Olli-Pekka Siikarla, Mika Vesterholm ym.) heiltä saamistani neuvoista sekä rakentavasta palautteesta. Kiitokset ja kumarrukset on myös ansainnut oikolukijani Laura Lahti.

SISÄLLYSLUETTELO

1. JOHDANTO **1**

2. TAUSTOJA JA HISTORIAA **3**

2.1. TIETOJENKÄSITTELYSTÄ TIETOJÄRJESTELMIIN 3
2.2. KILPAILUEDUN SAAVUTTAMINEN TIETOJÄRJESTELMIEN AVULLA 4
2.2.1. Kustannusten alentaminen ja tuotteen erottuminen kilpailevista tuotteista ... *4*
2.2.2. Ympäristötekijöihin vaikuttaminen *5*
2.3. OHJELMISTOKRIISI JA TAIKALUODIT 6
2.4. ARKKITEHTUURIT – TAIKALUOTI OHJELMISTOKRIISIN RATKAISEMISEKSI? 7
2.5. ARKKITEHTUUREIDEN TUTKIMUKSEN HISTORIAA 9
2.6. ARKKITEHTUUREIDEN NYKYTUTKIMUS 10

3. OHJELMISTOARKKITEHTUURIT **14**

3.1. OHJELMISTOARKKITEHTUURIN MÄÄRITELMIÄ 15
3.1.1. Arkkitehtuurin metamalli *18*
3.1.2. Mitä ohjelmistoarkkitehtuurit eivät ole *18*
3.2. MIHIN ARKKITEHTUURIA TARVITAAN? 20
3.2.1. Arkkitehtuuri sidosryhmien välisen kommunikaation välineenä *20*
3.2.2. Järjestelmän ensimmäiset suunnittelupäätökset *23*
3.2.3. Siirrettävissä ja uudelleenkäytettävissä oleva järjestelmän abstraktio *26*
3.3. ARKKITEHTUURIN LIIKETOIMINNALLINEN KIERTOKULKU 26
3.4. ARKKITEHTUURI JA PROSESSIMALLIT 28
3.4.1. Arkkitehtuuri ja ohjelmiston elinkaari *29*
3.4.2. Arkkitehtuuri ja vesiputousmalli *30*
3.4.3. Kehitysvaiheen arkkitehtuuri ja WinWin-spiraalimalli *30*
3.5. OHJELMISTOARKKITEHTUUREIDEN YHTEENVETO 37

4. ARKKITEHTONISET TYYLIT **38**

4.1. ARKKITEHTONISEN TYYLIN MÄÄRITELMIÄ 38
4.1.1. Garlanin ja Shaw'n vuoden 1994 määritelmä *38*
4.1.2. Muut yleiset arkkitehtonisen tyylin määritelmät *40*
4.2. TYYLIEN MUODOSTUMINEN 41
4.3. ESIMERKKEJÄ ARKKITEHTONISISTA TYYLEISTÄ 43
4.3.1. Putket ja suodattimet *43*
4.3.2. Kerroksittaiset järjestelmät *46*
4.3.3. Liitutaulumenetelmä *48*
4.3.4. Asiakas-palvelin -arkkitehtuuri *50*
4.3.5. Tietoabstraktiot ja oliopohjaiset järjestelmät *53*
4.3.6. Keskitetyn ohjauksen mallit *54*
4.3.7. Tapahtumapohjaiset järjestelmät *57*
4.3.8. Heterogeeniset tyylit *58*
4.3.9. Kolmitasoarkkitehtuuri esimerkkinä heterogeenisesta tyylistä *59*
4.4. JÄRJESTELMÄN KUVAAMINEN ARKKITEHTONISTEN TYYLIEN AVULLA 62
4.5. ARKKITEHTONISTEN TYYLIEN YHTEENVETO 64

5. ARKKITEHTUURISUUNNITTELU **65**

5.1. ARKKITEHTUURISUUNNITTELUN TAVOITTEITA 65
5.2. MISTÄ ARKKITEHTUURIT TULEVAT? 66
5.2.1. Varastamalla – menetelmällä – intuitiolla *67*
5.2.2. Rutiiniratkaisu vai innovaatio? *68*
5.2.3. Sovelluskehykset *68*
5.3. JÄRJESTELMÄN RAKENTEELLISEN ARKKITEHTUURIN MUODOSTAMINEN 70
5.3.1. Ylhäältä alas: rakenteisen analyysin tietovirtakaaviot *71*
5.3.2. Alhaalta ylös: komponenttipohjainen ohjelmistotuotanto *73*
5.3.3. Arkkitehtuurisuunnittelu ja oliolähestymistapa *77*
5.4. TYYLIEN KÄYTTÖ ARKKITEHTUURISUUNNITTELUSSA 79
5.4.1. Arkkitehtonisten tyylien jaottelusta *80*
5.4.2. Tyylien uudelleenkäytöstä *82*
5.4.3. Arkkitehtonisen tyylin valinnan ongelma *84*
5.4.4. Tyylistä ja vaatimuksista ohjelmiston arkkitehtuuriin *85*
5.4.5. Yhteenveto tyylien käytöstä arkkitehtuurisuunnittelun apuna *87*
5.5. MILLAINEN ON HYVÄ ARKKITEHTUURI? 87
5.6. ARKKITEHTUURISTA JÄRJESTELMÄN TOTEUTUKSEEN 88
5.6.1. Minijärjestelmän rakentaminen ja arviointi *89*
5.7. ARKKITEHTUURISUUNNITTELUN YHTEENVETO 90

6. YHTEENVETO 91

LÄHDELUETTELO 93

LIITTEET 103

1. JOHDANTO

Ohjelmistoarkkitehtuurin käsite voidaan katsoa muodostuneeksi, kun ensimmäinen ohjelma jaettiin kokonaistoiminnallisuuden toteuttaviksi komponenteiksi. Miten nämä ensimmäiset järjestelmän arkkitehtuuria koskevat suunnittelupäätökset tehdään? Ohjelmistosuunnittelijat ovat jo pitkään käyttäneet hyväkseen erilaisia malleja suunnittelupäätösten tekemiseksi. Mallien – tai arkkitehtonisten tyylien – käyttö arkkitehtuurisuunnittelun apuna on kuitenkin ollut yleensä tiedostamatonta; suunnittelijat ovat muodostaneet järjestelmän rakenteellisen jaottelun käyttämällä perusteena joko kokemuksesta hyväksi havaitsemiaan ratkaisumalleja tai yleisesti tiedossa olevia periaatteita.

Vielä nykyäänkin arkkitehtuurisuunnittelun katsotaan olevan eräänlaista taidetta – arkkitehtuureita tekevät "visionäärit, joilla on arkkitehtonisia näkemyksiä". Ohjelmistoarkkitehtuureihin tai niiden suunnitteluun liittyvää epämääräisyyttä ei myöskään helpota arkkitehtuurin tai arkkitehtonisen tyylin eksplisiittisen määritelmän puute. Mitä ohjelmistoarkkitehtuureilla ja niiden suunnittelun apuna käytettävillä arkkitehtonisilla tyyleillä varsinaisesti tarkoitetaan? Entä miten ohjelmistoarkkitehtuurit vaikuttavat niitä ympäröivään kontekstiin: järjestelmän kehityksen eri osapuoliin, järjestelmiä kehittävään organisaatioon tai kehityksessä käytettyihin prosessimalleihin? Tutkielman tavoitteena on antaa kokonaisnäkemys ohjelmistoarkkitehtuureiden tutkimusalueesta arkkitehtuureiden ymmärtämiseksi sekä käytännön hyödyntämiseksi. Tutkimusaineistona käytetään sekä kirjallisuutta että kirjoittajan omia kokemuksia. Tutkimusmenetelmä on käsitteellisteoreettinen.

Tutkielma perustuu pääasiassa seuraaviin olettamuksiin:

- Ohjelmistokriisin yhteydessä havaittu tarve hallita suurten järjestelmien monimutkaisuutta
- Järjestelmien kehittämisen eri prosessimallien teoriat kuten vesiputous- ja spiraalimalli
- Uudelleenkäytön avulla saavutettavissa oleva tuottavuuden parantuminen

Tutkielman luvussa 2 käsitellään arkkitehtuuriajattelun muodostumiseen vaikuttaneita sekä liiketoiminnallisia että ohjelmistoteknisiä syitä ja arkkitehtuureiden tutkimuksen historiaa. Luvussa 3 käsitellään ohjelmistoarkkitehtuurin määritelmiä sekä sen suhteita tietojärjestelmien kehittämisen eri käytäntöihin ja malleihin. Luku 4 käsittelee arkkitehtonisia tyylejä sekä järjestelmän arkkitehtuurin kuvaamista sen muodostamisen apuna käytettyjen tyylien avulla. Luvussa 5 keskitytään arkkitehtuurisuunnittelun eri lähestymistapoihin sekä ongelmiin. Luku 6 on tutkielman yhteenveto.

2. TAUSTOJA JA HISTORIAA

80-luvun kuluttajamarkkinoiden ympäristötekijöiden muutokset sekä kiristyvä kilpailutilanne saivat yritykset vaatimaan yhä tehokkaampia ja ennen kaikkea suurempia tietojärjestelmiä ohjelmistojen tuottajilta. Ohjelmistojen tuottajat joutuivat vaikeuksiin, kun tarjonta ei pystynyt vastaamaan kasvaneeseen kysyntään. Tämän luvun tarkoituksena on selvittää niitä syitä, jotka ovat pakottaneet hakemaan yhä uusia ratkaisuja tietojärjestelmien kehittämisen ongelmiin – ja jotka samalla johtivat ohjelmistojen arkkitehtuuriajattelun syntymiseen. Taustoihin tutustutaan sekä tietojärjestelmiä käyttävien että tietojärjestelmiä tuottavien organisaatioiden näkökulmasta.

2.1. Tietojenkäsittelystä tietojärjestelmiin

Michael Earl kutsuu kirjassaan "*Management Strategies for Information Technology*" yritysten 80-luvun lopun jälkeistä aikaa tietojärjestelmien aikakaudeksi (*Information Technology era*) erotuksena sitä edeltäneestä tietojenkäsittelyn aikakaudesta (*Data Processing era*). Tietojenkäsittelyn aikakauteen liittyvät silloiset tietokonelaitteistojen tähtitieteelliset hinnat, vanhentuneet (reikäkortti)teknologiat, tietokoneiden käyttöä leimaava insinöörihenki sekä pettymykset aikaisempiin epäonnistuneisiin tietojärjestelmien kehityshankkeisiin. Tietojärjestelmien aikakauden tunnusmerkkinä voidaan pitää tietojärjestelmien strategisen merkityksen oivaltamista; nykyisin tietojärjestelmät nähdään resursseina, jotka vaativat hallintaa kuten muutkin strategisesti tärkeät resurssit [Earl 89].

Tietojärjestelmien liiketoimintaan vaikuttava strateginen merkitys johtuu pääasiassa kahdesta syystä: itse teknologian kehityksestä mahdollistavana tekniikkana sekä yritysten muuttuvista ympäristötekijöistä.

Teknologian kehitystä vei eteenpäin laitteistojen, telekommunikaatio- ja prosessiautomaatiosektoreiden sekä ohjelmistojen nopea kehitys. Ympäristötekijöiden muutoksella tarkoitetaan lainsäädäntöjen ja käytäntöjen vapautumisesta johtuvaa kansainvälisen kil-

pailutilanteen kiristymistä sekä ihmisten arvomaailmassa tapahtuneita muutoksia. Esimerkiksi koti ja vapaa-aika ovat kasvattaneet arvostustaan kuluttajien silmissä.

Tietojärjestelmiä käytetään yritysten tuloksille asetettujen tavoitteiden saavuttamiseen sidosryhmien, asiakkaiden, omistajien ja työntekijöiden silmissä [Callon 96]. Tietojärjestelmät saattavat osoittautua juuri siksi strategiseksi aseeksi, joka määrää yrityksen menestyksen tai menetyksen *kilpailuedun* saavuttamisessa.

2.2. Kilpailuedun saavuttaminen tietojärjestelmien avulla

Kilpailuedun saavuttamista tietojärjestelmien avulla tutkivat mm. Porter ja Millar [1985] artikkelissaan *"How information gives you competitive advantage"*. Artikkelissa esitetään neljä pääasiallista tapaa, miten tietojärjestelmien avulla voidaan saavuttaa etumatkaa kilpailijoihin:

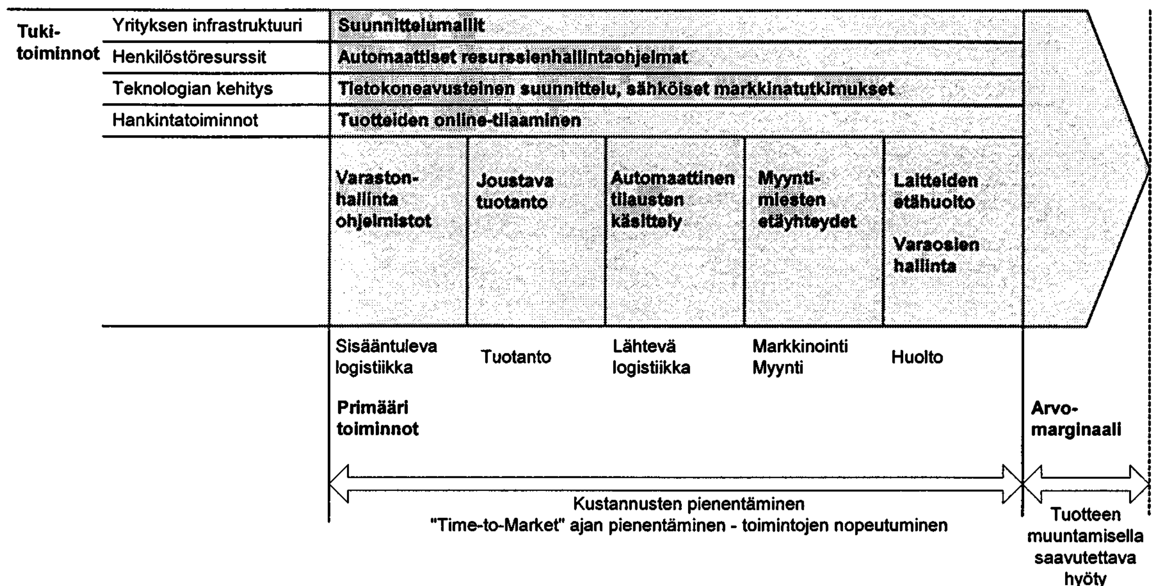
1. Tuotantokustannusten alentaminen
2. Tuotteen erottuminen edukseen muista vastaavista tuotteista (*enhancing product differentiation*)
3. Kilpailun pelisääntöjen (ja mittakaavan) muutos
4. Uusien liiketoiminta-alueiden luominen

2.2.1. Kustannusten alentaminen ja tuotteen erottuminen kilpailevista tuotteista

Tuotteen tuotantokustannusten alentamisesta ja tuotteen erottumisesta saavutettavia hyötyjä voidaan kuvata Porterin arvoketjun (*value chain*) avulla, joka soveltuu minkä tahansa toimialan arvioimiseen [Callon 96]. Arvoketju-ajattelutavan mukaan yrityksen liiketoiminta jakautuu liiketoiminnan muodostaviin teknologisiin ja taloudellisiin arvotoimintoihin (*value activities*). Yritys tuottaa voittoa, mikäli asiakkaan tuotteista maksama arvo on suurempi, kuin yritys on kuluttanut arvon tuottamiseen liittyviin toimintoihin. Menestyäkseen kilpailussa yrityksen on siis joko saatava minimoiduksi arvon

tuottamiseen kulutettu raha tai vaihtoehtoisesti pyrittävä muuntamaan tuotetta siten, että se eroaa kilpailevista ja korvaavista tuotteista niin, että asiakas maksaisi paremman hinnan, eli tuotteella olisi suurempi arvo.

Arvoketju koostuu *primääritoiminnoista*, kuten logistiikka, tuotanto, markkinointi ja myynti sekä huolto, ja *tukitoiminnoista*, kuten yrityksen infrastruktuuri, teknologian kehitys, henkilöstöresurssien hallinta jne. Tietojärjestelmät vaikuttavat tämän arvoketjun muodostumisen jokaiseen kohtaan. Tietojärjestelmien käyttö muuttaa sekä itse toimintojen suoritustapoja että niitä tapoja, joilla nämä toiminnot liittyvät toisiinsa. Muuntamalla tuotetta tietotekniikan avulla voidaan kasvattaa kuluttajan havaitsemaa tuotteen arvoa (siis kuluttajan maksamaa hintaa) tai pyrkiä pienentämään arvotoimintojen suorittamisen kustannuksia [Porter & Millar 85]. Kuva 1 havainnollistaa tietotekniikan vaikutuksia Porterin arvoketjuun.



Kuva 1: Esimerkki tietotekniikan vaikutuksista Porterin arvoketjuun. Muunneltu [Porter & Millar 85]:stä

2.2.2. Ympäristötekijöihin vaikuttaminen

Tuotteen muuntamisen ja kustannusten pienentämisen lisäksi tietotekniikkaa voidaan Porterin ja Millarin [1985] mukaan käyttää kilpailuaseena liiketoiminnan mittakaavan

muuttamisessa. Kilpailutilanteen mittakaavan muutosta voidaan pitää eräänlaisena kilpailun pelisääntöjen muuttamisena oman edun saavuttamiseksi. Esimerkkinä artikkelissa mainitaan Wall Street Journal-lehti, joka teki pioneerityötä sivujen siirtotekniikan kehittämisessä. Lehden paikallisia versiota myytiin ja painettiin (vuonna 1985) seitsemässätoista eri maassa. Pääosa kaikkien kieliversioiden artikkeleista kirjoitettiin keskitetysti vain kerran ja siirrettiin suoraan tietotekniikan avulla painettavaksi eri maihin. Tietotekniikkaa käytettiin siis kilpailun mittakaavan muuttamisessa paikallisesta kilpailusta kansainväliseksi kilpailuksi [Porter & Millar 85].

Tietotekniikka on myös luonut kokonaan uusia liiketoiminta-alueita – esimerkkinä viime aikojen puhelinyhtiöiden muuttuminen puhelinlinjoja toimittavista yrityksistä tietoliikenneyhteyksien toimittajiksi. Yritysten on ollut siis mahdollista kehittää liiketoimintamuotoja, joita aikaisemmin ei ole ollut edes olemassa.

2.3. Ohjelmistokriisi ja taikaluodit

Tietojärjestelmiä käyttävien organisaatioiden lisäksi myös ohjelmistojen tuottajat ovat olleet vaikeuksissa. Vuonna 1968 NATOn seminaarissa esitetty "ohjelmistokriisi" kertoi tilasta, johon ohjelmistojen tuottajat olivat ajautuneet. Ohjelmistokriisillä tarkoitettiin niitä ongelmia, jotka seurasivat rakennettavien järjestelmien koon jatkuvasta kasvamisesta. Ongelmana ei ollut rakennettujen ohjelmien toimivuus, vaan puute kurinalaisista menetelmistä joiden avulla pystyttäisiin toisaalta hallitsemaan suurten ohjelmistojen monimutkaisuutta, toisaalta taas nopeuttamaan ohjelmistojen kehitysprosessia niin, että kasvavaan ohjelmistojen kysyntään voitaisiin vastata järkevässä ajassa. Tätä vuoden 1968 konferenssia pidetään nykyaikaisen ohjelmistotekniikan (*software engineering*) perustamistilaisuutena [Haikala & Märijärvi 98] [Pressman 97] [Krueger 92] [Garlan & Shaw 96].

Frederick Brooksin artikkelin "*No Silver Bullet – Essence and Accidents of Software Engineering*" [Brooks87] mukaan ohjelmistojen kehittämiseen liittyvät ongelmat johtuvat itse ohjelmistojen kehitystyön olemuksesta (*essence*) sekä satunnaisista voitetta-

vissa olevista ongelmista (*accidents*). Vakavampia olemukseen liittyviä ongelmia ovat ohjelmistojen luontainen monimutkaisuus, ohjelmistojen yhteensopimattomuus toistensa kanssa, ohjelmistojen vaikea muunnettavuus sekä ohjelmien näkymättömyys.

Luontaisella monimutkaisuudella tarkoitetaan kehitystyössä käytettävien rakenteiden monimutkaisuutta. Kahta samanlaista rakennetta ei ole, ja rakenteet kytkeytyvät toisiinsa hyvin monella eri tasolla. Monimutkaisuus tekee järjestelmän ymmärtämisen, kuvaamisen ja testaamisen äärimmäisen vaikeaksi.

Yhteensopivuuden ongelmalla tarkoitetaan järjestelmien toisiinsa liittämisen vaikeutta. Mitään standardeja järjestelmien ja komponenttien liittymätapoja ei ole olemassa. Jokainen liittymä on yksilöllinen.

Muunnettavuuden ongelmalla tarkoitetaan sitä, että koska ohjelmistoa voidaan muuntaa, sen muuntamiselle löytyy jatkuvaa tarvetta. Ohjelman tai tietojärjestelmän elinkaarta voidaan jatkaa alunperin suunniteltua pidemmäksi muuttamalla ohjelmistoa uusia tarpeita vastaavaksi. Vastakohtana artikkelissa mainitaan auton rakenne, joka todennäköisesti pysyy koko sen elinkaaren ajan täsmälleen samanlaisena (neljä pyörää, kori, yksi moottori jne.)

Ohjelmia ei voi nähdä. Ihmisen yksi tärkein tapa asioiden ymmärtämiseksi on konkreettinen näkeminen – visualisointi, joka ohjelmistojen yhteydessä on mahdotonta. Ymmärtämisen lisäksi näkymättömyys vaikeuttaa ohjelmiston rakenteesta keskustelemista toisten osapuolten kanssa. [Brooks 87]

2.4. Arkkitehtuurit – taikaluoti ohjelmistokriisin ratkaisemiseksi?

Brooksin mukaan mitään taikaluotia – yksittäistä ohjelmistokriisin ratkaisevaa tekijää – ei ole olemassa. Parhaat keinot yrittää ratkaista tietojärjestelmien olemuksesta johtuvat ongelmat ovat toistaiseksi olleet komponenttien ostaminen itse rakentamisen sijaan, järjestelmän vaatimusten asteittainen jalostaminen prototyyppimenetelmää käyttäen,

ohjelmiston kasvattaminen eli inkrementaalinen rakentaminen ja loistavat suunnittelijat järjestelmän rakentajina [Brooks 87].

> *"Uskon, että ohjelmistojen rakentamisen vaikeus liittyy ohjelmiston käsitteellisen rakenteen määritykseen, suunnitteluun ja testaukseen, ei itse rakenteen esittämiseen ja esityksen testaamiseen."*
>
> - Fred Brooks: "No Silver Bullet", 1987

Ohjelmistoarkkitehtuureita ei kannata pitää tietojärjestelmien kehittämisen kaikkia ongelmia ratkaisevana taikaluotina – mikään yksittäinen tekniikka tai ajattelutapa tuskin yksinään tulee ratkaisemaan ohjelmistokriisin ongelmia. Arkkitehtuureita voidaan kuitenkin pitää askeleena oikeaan suuntaan; arkkitehtuurit käsittelevät niitä lähestymistapoja, jotka Brooks mainitsi artikkelissaan lupaaviksi yrityksiksi voittaa ohjelmistojen kehittämiseen liittyvät ongelmat:

- Järjestelmien monimutkaisuuden hallinta korkean tason abstraktioiden ja niiden potentiaalisen uudelleenkäytön avulla [Medvidovic & Taylor 98].
- Järjestelmälle asetettujen vaatimusten asteittainen jalostaminen (Boehmin MBASE-menetelmä, jossa elinkaariarkkitehtuuria pidetään kehitysvaiheiden elinkaarivaatimusten "henkilöitymänä" [Boehm ym. 98]).
- Kuinka hyväksi havaittujen arkkitehtonisten tyylien tutkimuksen ja opetuksen avulla saadaan aikaan tietojärjestelmien kehitystyössä tarvittavia loistavia suunnittelijoita – järjestelmäarkkitehteja [Garlan & Shaw 96].

Tässä tutkielmassa tarkastellaan, kuinka ohjelmistoarkkitehtuureiden avulla voidaan ratkaista tai ainakin helpottaa tietojärjestelmien rakentamisen olemuksesta johtuvia ongelmia. Tutkielmassa lähestytään ongelmaa ohjelmiston tuottajan näkökulmasta.

2.5. Arkkitehtuureiden tutkimuksen historiaa

Vaikka arkkitehtuureiden tutkimus onkin nykyisin saavuttanut melko suuren suosion tieteenharjoittajien keskuudessa, on hyvä huomata tutkimuksen juurten ulottuvan kuitenkin hieman kauemmas menneisyyteen. Arkkitehtuureiden nykytutkimuksessa on hyvin monella tapaa "löydetty uudelleen" esim. Frederick Brooksin, Edsger Dijkstran ja David Parnasin jo neljännesvuosisata sitten esittämiä periaatteita [Bass ym. 98].

Ensimmäisiä arkkitehtuuriajattelun muodostumiseen vaikuttaneita tutkimuksia oli Dijkstran [1968] artikkeli "The structure of the 'T.H.E' multiprogramming system.". Artikkeli käsitteli käyttöjärjestelmien muodostamista kerroksittaisen mallin avulla (*layered system*), jossa jokainen järjestelmän kerros saa rajapintojen välityksellä olla yhteydessä vain sen ylä- tai alapuolella olevaan kerrokseen. Dijkstran mielestä kannatti kiinnittää huomiota siihen miten ohjelmisto on muodostunut, eikä vain siihen, tuottaako ohjelma ajonaikaisesti oikean tuloksen. Kerroksittaisen mallin käyttö yleistyi muuhunkin käyttöön kuin pelkkään käyttöjärjestelmien rakentamiseen [Clements & Northrop 96]. Kerroksittaista mallia pidetään nykyään tyypillisenä esimerkkinä järjestelmän rakentamisessa käytetystä arkkitehtonisesta tyylistä [Garlan & Shaw 96] [Garlan & Perry 95] [Bass ym. 98].

Dijkstran artikkelin lisäksi arkkitehtuurin käsitteen muodostumiseen johtaneita tutkimuksia olivat kolme Parnasin artikkelia: moduulisuunnitteluun, kapselointiin ja tiedon piilottamiseen liittyvä artikkeli "On the Criteria for Decomposing Systems into Modules", vuoden 1974 ohjelmistojen hierarkkisia rakenteita käsitellyt tutkimus "On a 'Buzzword': Hieratchical Structure" ja vuoden 1976 ohjelmaperheitä ja ohjelmistotuotannon tuotelinjoja tutkinut artikkeli "On the Design and Development of Program Families" [Clements & Northrop 96].

Järjestelmien kehitystyötä tutkittaessa voidaan havaita ohjelmointikielten ja työkalujen abstraktiotason nousemista sekä ohjelmistojen rakentamiseen käytettävien komponenttien käsitteellisen koon kasvamista. Abstraktiotason nousemista voidaan havaita myös

kehitysmetodeissa: abstraktien tietotyyppien käyttö nostaa abstraktiotasoa ylöspäin kooditasolta [Garlan & Shaw 94].

Kaikkia edellä mainittuja käytännön elämän ilmiöiden havaintoja ja tieteellisiä tutkimuksia yhdistää yhteinen piirre: ohjelmistojen ja tietojärjestelmien tarkastelun tason jatkuva nouseminen ensin kooditasolta suunnittelun tasolle ja edelleen ylöspäin kohti arkkitehtonisen tarkastelun tasoa.

2.6. Arkkitehtuureiden nykytutkimus

Arkkitehtuureita ovat viime aikoina tutkineet sekä käytännön tasolla toimivat ohjelmistosuunnittelijat että varsinaiset tieteenharjoittajat. Arkkitehtuureihin liittyviä asioita on käsitelty moduulien rajapintakielien määrittelyssä, sovellusaluekohtaisten arkkitehtuureiden (*domain specific architectures*) tutkimuksessa, ohjelmistojen uudelleenkäytön yhteydessä jne. [Garlan & Shaw 96].

Arkkitehtuureiden tutkimuksen ongelmana voidaan Garlanin ja Shawin [1996] mukaan pitää sitä, että tutkimusta tehdään päällekkäisesti pienissä ryhmissä toisistaan tietämättä useassa eri paikassa. Ongelmaa on yritetty ratkaista perustamalla työryhmiä, joissa arkkitehtuureiden tutkijat voisivat jakaa tietojaan nykytutkimuksen tilasta ja käynnissä olevasta tutkimuksesta. Työryhmät ovat saaneet osakseen varsin suurta kiinnostusta [Garlan & Shaw 96].

Esimerkiksi IEEE Architecture Working Group:in (AWG) tavoitteena on ollut "ottaa kokonaisvaltaisesti kantaa arkkitehtuureihin ohjelmistopainotteisten järjestelmien sovelluskohteena, muodostaa käsitteellinen rakenne ja sanasto arkkitehtuureista keskustelemiseen, tunnistaa ja tehdä tunnetuksi hyviä arkkitehtonisia rakenteita sekä antaa käytäntöjen kehittyä relevantteina teknologioina" [AWG 99].

Garlanin ja Shawin [1996] mukaan arkkitehtuureiden tutkimus on suuntautumassa arkkitehtuureiden kuvauskielten kehittämiseen, arkkitehtonisen käytännön kokemusmaail-

man muodollistamiseen niin, että siitä voidaan keskustella ja mahdollisesti opettaa edelleen, spesifisten sovellusalueiden arkkitehtuureiden sovelluskehysten luomiseen sekä arkkitehtuurin formalisointiin [Garlan & Shaw 96].

Arkkitehtuurien tutkimuksessa nojaudutaan pääasiassa seuraaviin uskottaviin oletuksiin, vaikka niitä ei olekaan ehdottomasti todistettu [Clements 96]:

- Järjestelmiä voidaan rakentaa nopeasti ja kustannustehokkaasti käyttämällä (tai luomalla) suuria ulkoisesti kehitettyjä komponentteja. Aikaisemmat paradigmat ovat keskittyneet päätoimintonaan koodiriveissä mitattavaan ohjelmointiin; arkkitehtuureihin perustuvalla järjestelmän kehittämisellä tarkoitetaan komponenttien käyttöön perustuvaa ohjelmistotuotantoa. Keskeiseksi käsitteeksi nousee komponenttien yhdistäminen.
- Järjestelmän arkkitehtuurin tutkimisen avulla voidaan ennustaa tiettyjä järjestelmän ominaisuuksia jo ennen suunnittelu- tai toteutusvaihetta.
- Ohjelmistotuotannossa voidaan muodostaa kokonaisia tuotelinjoja ohjelmaperheen viitearkkitehtuurin käytön avulla. Korkean tason uudelleenkäyttö mahdollistetaan arkkitehtonisen suunnittelun avulla.
- Komponentin toiminnallisuus voidaan (ja on järkevää) erottaa sen liittymistä. Perinteisesti komponenttien yhteydessä mielenkiinto on kohdistettu vain sen toiminnallisuuteen
- “Vähemmän on enemmän”. Tietojärjestelmä voidaan muodostaa useilla eri tavoilla, mutta suunnitteluavaruuden rajoittaminen muutamaan vaihtoehtoon on Garlanin ja Shawin [1996] mukaan järkevää, koska sen avulla voidaan parantaa uudelleenkäyttöä, lyhentää käytetyn rakenteen valintaan kulutettua aikaa (kaikkien kombinaatioiden testaaminen ja arviointi vie aikaa), sekä parantaa komponenttien välistä toimintaa.

Lupaavinta kehitystä voidaan Clementsin [1996] mukaan odottaa seuraavilta osa-alueilta:

- Arkkitehtuureiden suunnittelu tai valinta, miten valita arkkitehtuuri, joka toteuttaa joukon toiminnallisia, suorituskyvyllisiä ja laadullisia vaatimuksia?

- Arkkitehtuureiden esitystapa, miten arkkitehtuuri pitäisi kuvata mallien avulla?
- Arkkitehtuureiden arviointi ja analyysi, kuinka arkkitehtuuria voi käyttää järjestelmän ominaisuuksien arviointiin ja ennustamiseen?
- Arkkitehtuureihin perustuvien kehitysmetodien muodostaminen ja kehittäminen
- Arkkitehtuuriin palauttaminen, vanhojen järjestelmien (esim. perinteiset COBOL-keskuskoneet) muuntaminen uusia tarpeita vastaavaksi.

Taylorin ja Medvidovicin [1998] mukaan arkkitehtuureiden tutkimuksen "ensimmäinen sukupolvi" on kehittynyt niin pitkälle, että arkkitehtuurin todellisia vaikutuksia voidaan alkaa arvioida. Kirjoittajien mukaan arkkitehtuuria leimaa vielä toistaiseksi turha akateemisuus, ja hyvin vähän kehitetystä teknologiasta on siirtynyt tutkimuksesta käytäntöön. Toistaiseksi sovelluskohteita joilla arkkitehtuureiden avulla olisi saatu kaikkein suurimpia hyötyjä on jäänyt käyttämättä – toisilla sovellusalueilla on taas luvattu liikoja [Medvidovic & Taylor 98].

Arkkitehtuureita tutkitaan pääasiassa kahdessa eri tutkimuslaitoksessa: Carnegie-Mellon Universityn Software Engineering Institutessa (CMU/SEI), ja University of Southern Californian Center for Software Engineeringissä (USC/CSE).

SEI:n pääasialliset tutkimuskohteet ovat ohjelmistotuotannon hallinta (Software Engineering Management Practices) ja ohjelmistotuotannon tekniikat (Software Engineering Technical Practices). Arkkitehtuureiden tutkimus kuuluu SEI:n ohjelmistotuotannon tutkimuksen alaisuuteen. Tutkielmassa viitattuja CMU:ssä tai SEI:ssä nykyisin (kesäkuu 99) vaikuttavia henkilöitä ovat mm.

- Len Bass (*Senior Member of the Technical Staff*)
- Paul C. Clements (*Senior Member of the Technical Staff*)
- Rick Kazman (*Senior Member of the Technical Staff*)
- Linda M. Northrop (*Director, Product Line Systems Program*)
- Mary Shaw (*A.J. Perlis Professor, Associate Dean for Professional Programs*)
- David Garlan (*Associate Professor*)

University of Southern Californian Center for Software Engineeringin (USC/CSE) nykytutkimus keskittyy ohjelmistoarkkitehtuureiden lisäksi COCOMO-mallin kehittämiseen, WinWin-spiraalimallin tutkimukseen, MBASE-lähestymistavan tutkimukseen sekä muutamaan muuhun eritysalueeseen. CSE:n arkkitehtuuritutkimukseen liittyy arkkitehtuurin suhteiden tarkastelu mm. MBASE-lähestymistapaan, WinWin-spiraalimalliin ja ohjelmistotuotannon prosessimalleihin yleensä. Tutkimuksessa viitattuja CSE:läisiä ovat mm.

- Barry Boehm (*TRW Professor of Software Engineering*) CSE:n perustaja
- Nenad Medvidovic (*Assistant Professor*)
- Alexander Egyed (*Research Assistant*)
- Cristina Gacek (*PhD, Reseach Associate*)
- Dan Port (*PhD, Reseach Associate*)
- Ahmed Abd-Allah (*PhD*)

Seuraavassa kappaleessa siirrytään tutkimaan ohjelmistoarkkitehtuureiden kirjallisuudessa esitettyjä määritelmiä sekä suhteita tietojärjestelmien kehityksessä käytettyihin menetelmiin ja prosessimalleihin.

3. OHJELMISTOARKKITEHTUURIT

Main Entry: **ar·chi·tec·ture**
Pronunciation: 'är-k&-"tek-ch&r
Function: *noun*
Date: 1555

1 : the art or science of building; specifically : the art or practice of designing and building structures and especially habitable ones
2 a : formation or construction as or as if as the result of conscious act <the architecture of the garden>
b : a unifying or coherent form or structure <the novel lacks architecture>
3 : architectural product or work
4 : a method or style of building
5 : the manner in which the components of a computer or computer system are organized and integrated

- Webster Dictionary 1999

Ohjelmistojen koon ja monimutkaisuuden kasvaessa järjestelmän kokonaisrakenteen hallinta on muuttunut tärkeämmäksi kuin esim. algoritmien suunnittelu. Järjestelmäkokonaisuuden muodostavat mm. järjestelmän komponenttirakenne, järjestelmän toimintaa ohjaavat kontrollirakenteet, tiedonsiirtoprotokollat, tietojen suojaus ja tiedonsiirron synkronointi, komponenttien toteuttama toiminnallisuuden jakautuminen, järjestelmän fyysinen jakautuminen eri koneille, järjestelmän skaalautuvuus ja suorituskyky, järjestelmän ajan myötä tapahtuva evoluutio ja ja järjestelmän toteutusvaihtoehtojen välillä tehtävä valinta. Tätä järjestelmän tai ohjelmiston tarkastelun tasoa sanotaan ohjelmiston arkkitehtoniseksi tasoksi [Garlan & Shaw 96] [Garlan & Perry 95] [Perry & Wolf 92] [Garlan & Shaw 94].

Tämä luku tutkii ohjelmistoarkkitehtuurin käsitettä ja sen suhteita muihin tietojärjestelmien kehitystyön käytäntöihin. Ensimmäisessä kappaleessa esitetään ja analysoidaan ohjelmistoarkkitehtuurin kirjallisuudessa esitettyjä määritelmiä. Seuraavissa kappaleissa etsitään syitä arkkitehtuureiden tarpeellisuudelle järjestelmien kehitystyössä ja todetaan ohjelmistoarkkitehtuurin vaikuttavan myös takaisin sitä kehittävään organisaatioon. Lopulta tutkitaan, miten arkkitehtuuri ja sen muodostaminen liittyy järjestelmien kehittämistyön prosesimalleihin.

3.1. Ohjelmistoarkkitehtuurin määritelmiä

Tässä kappaleessa käsitellään kirjallisuudessa esitettyjä ohjelmistoarkkitehtuurin määritelmiä. Vaikka määritelmiä löytyy useita, mitään yleisesti hyväksyttyä ohjelmistoarkkitehtuurin eksplisiittistä määritelmää ei ole olemassa [Garlan & Perry 95] [Boehm ym. 98] [Bass ym. 98].

Arkkitehtuuri-termiä käytetään tietojärjestelmien yhteydessä hyvin laajasti kuvaamaan jotakin seuraavista vaihtoehdoista [Garlan & Perry 95]:

1. Yksittäisen ohjelmiston arkkitehtuuria (*instance of architecture*)
2. Arkkitehtonista tyyliä (*architectural style*)
3. Arkkitehtuureiden tutkimusta (*study of architecture*)

Tässä tutkielmassa käytetään termiä 'ohjelmistoarkkitehtuuri' kuvaamaan yksittäisen ohjelman tai järjestelmän arkkitehtuuria. Joissakin yhteyksissä tutkielmassa käytetään myös termejä ohjelma**n**, ohjelmisto**n** tai järjestelmä**n** arkkitehtuuri joilla kaikilla tarkoitetaan samaa asiaa – arkkitehtuurin yksittäistä esiintymää eli instanssia.

Ohjelmistoarkkitehtuurille mainitaan kirjallisuudessa kolme erilaista perusmääritelmää, jotka sisällyttävät ohjelmiston arkkitehtuuriin hieman eri ominaisuuksia. Ohjelmistoarkkitehtuurin (tai ohjelmiston arkkitehtuurin) perusmääritelmään tyypillisesti kuuluu komponentit ja niiden väliset liittymät [Garlan & Shaw 94]:

Yksittäisen järjestelmän arkkitehtuuri on joukko tietojenkäsittelyä suorittavia komponentteja – tai lyhyemmin komponentteja – ja näiden komponenttien välisten vuorovaikutusten kuvauksia – liittymiä.

Toisessa perusmääritelmässä katsotaan arkkitehtuuriin kuuluvan edellisten lisäksi sekä muoto että arkkitehtuurin muodostumisen sanallisesti kuvaava perustelu (*rationale*). Alla Perryn ja Wolfin [1992] määritelmä ohjelmiston arkkitehtuurista:

Ohjelmiston arkkitehtuuri =

{ elementit, muoto, perustelu }

eli ohjelmistoarkkitehtuuri muodostuu joukosta arkkitehtonisia (tai jos halutaan, suunnittelu-) elementtejä, joilla on tietty muoto.

Edellä mainitussa määrittelyssä arkkitehtonisiksi elementeiksi lasketaan tiedonkäsittelyä suorittavat elementit, elementtien väliset liittymät (eli edellisen määritelmän komponenttien väliset liittymät) sekä tietoelementit. Muodolla tarkoitetaan elementtien välisiä painotettuja ominaisuuksia, esimerkiksi joidenkin suhteiden tai ominaisuuksien keskeistä tärkeyttä arkkitehtuurin muodostumisessa. Arkkitehtuurin perustelu sisältää tiedot arkkitehtuurin muodostamisessa tehdyistä ratkaisuista – perustelu "vangitsee" arkkitehtuurin olennaisimmat piirteet tyylistä, elementtien valinnoista ja muodosta [Perry & Wolf 92].

Ehkä kattavimmassa ohjelmistoarkkitehtuurin määritelmässä Gacek ym. [1995] määrittelevät ohjelmiston arkkitehtuuriin kuuluvan edellä mainittujen ominaisuuksien lisäksi vielä sidosryhmien järjestelmälle asettamat vaatimukset sekä selvityksen siitä, miten ohjelmiston arkkitehtuuri toteuttaa nämä vaatimukset.

Ohjelmistoarkkitehtuuri koostuu seuraavista osa-alueista:

- *Joukko ohjelmistoja ja järjestelmäkomponentteja, niiden välisiä liittymiä ja järjestelmälle asetettuja (esim. laadullisia ja tehokkuutta määrääviä) rajoitteita*
- *Joukko sidosryhmien järjestelmälle asettamia vaatimuksia*
- *Selvitys siitä, miten nämä komponentit, liittymät ja rajoitteet määrittelevät ohjelmiston, joka toteuttaa sille asetetut vaatimukset*

[Gacek ym. 95]

Ohjelmistoarkkitehtuureita käsitellään useissa muissakin tutkimuksissa ja kirjoissa. Niissä harvoin kuitenkaan pyritään määrittelemään itse ohjelmistoarkkitehtuurin käsitettä, vaan ohjelmistoarkkitehtuuri kuvataan arkkitehtuurisuunnittelun tulokseksi.

Esimerkkinä edellisestä: "Arkkitehtuurisuunnittelulla tarkoitetaan järjestelmän jakautumista alijärjestelmiksi ja niiden välisiksi ohjaus- ja yhteydenpitomenetelmiksi" [Sommerville 95]. Toisena esimerkkinä voidaan pitää Pressmanin [1997] samansuuntaista määritelmää: "Arkkitehtuurisuunnittelussa järjestelmä jaetaan moduuleiksi ja niiden välisiksi ohjausrakenteiksi." Arkkitehtuurisuunnittelun kautta saatavat ohjelmistoarkkitehtuurin määritelmät tyypillisesti palautuvat johonkin em. kolmesta perusmääritelmästä tai johonkin niiden yhdistelmään.

Tässä tutkielmassa ohjelmiston arkkitehtuuri käsitetään Garlanin ja Shaw'n [1994] määritelmän mukaisesti järjestelmän muodostaviksi komponenteiksi ja niiden välisiksi liittymiksi. Jokaisella järjestelmällä on arkkitehtuuri [Garlan & Shaw 96] [Garlan & Perry 95] riippumatta siitä, onko arkkitehtuurin muodostumiselle järjellisiä perusteluita vai ei (vrt. [Perry & Wolf 92]). Gacek ym. [1995] ajatus arkkitehtuuriin kuuluvista järjestelmän tavoitteista, rajoitteista ja vaatimuksista on ymmärrettävissä ohjelmistoarkkitehtuureiden käytön kautta, mutta em. piirteiden lisääminen ohjelmistoarkkitehtuurin palvelee paremminkin arkkitehtuurin muodostamisprosessin kuin itse ohjelmistoarkkitehtuurin eksplisiittisen määritelmän tarpeita.

Ohjelmistoarkkitehtuureille voidaan yleistää edellisten määritelmien perusteella muutamia tunnusomaisia piirteitä [Garlan & Perry 95]:

Arkkitehtuuritasolla järjestelmää ei tarkastella algoritmien tai tietorakenteiden näkökulmasta vaan sen toiminnallisten kokonaisuuksien, osien suhteiden ja niiden välisten vuorovaikutusten karkean tason kuvauksena (*Focus of Concern* – mielenkiinnon abstraktion taso).

Järjestelmää ei tarkastella koodiin sidoksissa olevina ohjelmamoduuleina vaan kokonaisen toiminnallisen järjestelmän muodostavina komponentteina ja niiden välisinä liittyminä. (*Nature of Representation* – esitystapa)

Ohjelmistoarkkitehtuureiden yhteydessä erotetaan arkkitehtuurin yksittäinen esiintymä arkkitehtonisesta tyylistä (*Instance versus Style* – arkkitehtuurin esiintymä vastaan arkkitehtoninen tyyli). Arkkitehtonisia tyylejä käsitellään tarkemmin tutkielman luvussa 4.

Suunnittelumenetelmät ja ohjelmiston arkkitehtuuri eroavat toisistaan vaikka niitä käytetäänkin saman lopputuloksen saavuttamiseksi; molempien avulla pyritään saamaan aikaan ohjelmiston vaatimukset toteuttava järjestelmä, mutta suunnittelumenetelmät ja arkkitehtuuri eroavat siinä kuinka tähän lopputulokseen päästään. (*Design Methods versus Architecture* – suunnittelumenetelmät vastaan arkkitehtuuri). Suunnittelumenetelmien ja arkkitehtuurin eroja käsitellään hieman myöhemmin tämän tutkielman kappaleessa 5.3.3. arkkitehtuurisuunnittelun yhteydessä. [Garlan & Perry 95]

3.1.1. Arkkitehtuurin metamalli

Grady Boochin esityksessä "Software Architecture and the UML" [Booch 99] esitetty arkkitehtuurin metamalli pyrkii selventämään arkkitehtuuriin liittyvien eri käsitteiden suhteita ja koostumuksia. Vaikka liitteessä 2 esitetty arkkitehtuurin metamallin kaavio esittääkin asioita UML:n ja Rational Rose-ohjelmiston käytölle suotuisassa valossa, siitä voidaan kuitenkin hyvin havaita esimerkiksi ohjelmiston arkkitehtuurin ja arkkitehtonisten tyylien suhteet toisiinsa.

3.1.2. Mitä ohjelmistoarkkitehtuurit eivät ole

Arkkitehtuurin käsitettä on vielä toistaiseksi leimannut eräänlainen uuden tutkimusalueen liiallinen innostuneisuus ja odotuksellisuus; arkkitehtuureille on osoitettu epärealistisia odotuksia. Pettymysten jälkeen on huomattu, että ohjelmistoarkkitehtuurit eivät olekaan Brooksin "taikaluoti" [Medvidovic & Taylor 98].

Arkkitehtuuria kuvataan usein "järjestelmän yleisrakenteena". Tämä määritelmä ei pidä paikkansa siinä mielessä, että sen mukaan järjestelmässä olisi vain yksi rakenne. Todellisuudessa arkkitehtuurissa voidaan havaita useita samanaikaisia päällekkäisiä sekä hierarkkisia rakenteita [Bass ym. 98] [Booch 99].

Arkkitehtuuri sekoitetaan usein virheellisesti arkkitehtuurin kuvauskieliin (*ADL – architecture description language*). Arkkitehtuureiden kuvauskieliä voidaan pitää kuitenkin vain mahdollistavana tekniikkana arkkitehtuureiden hyödyntämiselle [Medvidovic & Taylor 98], samoin kuin esim UML:n notaatiota voidaan pitää olioiden kuvaamisen mahdollistavana tekniikkana.

Ohjelman suunnitteluvaiheen (*program design*) mallien ja arkkitehtuurin erona voidaan pitää tarkastelun ja näkökulman tasoeroa [Medvidovic & Taylor 98] [Booch 99]. Järjestelmän arkkitehtuuri tarkastelee järjestelmää merkittävästi korkeammalta (abstraktimmalta) tasolta.

Ohjelmistoarkkitehtuurin sekoittaminen arkkitehtuuriin perustuvaan järjestelmän analysointiin johtuu Medividovicin ja Taylorin mukaan siitä, että järjestelmän aikaisen kehitysvaiheen analysointi oli ensimmäisiä arkkitehtuureiden käytännön sovelluskohteita [Medvidovic & Taylor 98].

Boochin [1999] mukaan arkkitehtuuria pidetään myös virheellisesti samana asiana kuin infrastruktuuri ja luullaan, että ohjelmiston arkkitehtuuri syntyy yhden ihmisen työpanoksen tuloksena. Arkkitehtuuria pidetään lisäksi yksiulotteisena käsitteenä, eli yksi näkymä (kaavio) riittää kuvaamaan koko arkkitehtuurin.

Muita harhakäsityksiä ovat käsitys, että arkkitehtuuri kuvaa pelkästään järjestelmän loogisen (topologisen) rakenteen tai että ohjelmiston arkkitehtuuria ei voida mitenkään mitata tai validoida järjestelmän vaatimuksiin nähden. Lisäksi arkkitehtuuria pidetään sekä tieteenä että taiteena [Booch 99]!

3.2. Mihin arkkitehtuuria tarvitaan?

> Mikäli projekti ei ole saavuttanut järjestelmäarkkitehtuuria ja sen muodostumisen perustelevaa kuvausta, projektin ei pitäisi edetä täysimittaiseen järjestelmän kehitysvaiheeseen. Arkkitehtuurin määritys projektin tuotoksena mahdollistaa sen käytön läpi koko kehitys- ja ylläpitoprosessin.
>
> - Barry Boehm, 1995

Edellä oleva lainaus [Clements & Northrop 96] kuvaa Barry Boehmin näkemystä ohjelmiston arkkitehtuurin muodostamisen tärkeydestä järjestelmien kehityksessä. Tämä kappale käsittelee ohjelmistoarkkitehtuurin tyypillisiä sovelluskohteita – mihin arkkitehtuuria tarvitaan?

Arkkitehtuureita voidaan pitää tietojärjestelmien kehittämisen näkökulmasta tärkeinä niiden kolmen pääasiallisen ominaisuuden takia [Clements & Northrop 96] [Bass ym. 98]:

1. Arkkitehtuuri toimii sidosryhmien välisenä kommunikaation välineenä.
2. Arkkitehtuuri sisältää ensimmäiset järjestelmää koskevat suunnittelupäätökset.
3. Arkkitehtuuri on siirrettävissä (ja potentiaalisesti uudelleenkäytettävissä) oleva järjestelmän abstraktio.

3.2.1. Arkkitehtuuri sidosryhmien välisen kommunikaation välineenä

Rakennusten arkkitehtuurisuunnittelussa kaikilla eri sidosryhmien edustajilla – kuten toimeksiantajalla, sisustussuunnittelijalla, ympäristösuunnitelijalla, sähköinsinöörillä, LVI-insinööreillä – on eri näkemys rakennuksen arkkitehtuurista. Osapuolten eri näkemykset liittyvät kiinteästi toisiinsa ja muodostavat yhdessä kokonaisuuden – rakennuksen arkkitehtuurin [Clements & Northrop 96]. Ohjelmistoarkkitehtuuria voidaan pitää monilta osin hyvin samankaltaisena kuin rakennusarkkitehtuuria.

Tutkielman edellisessä ohjelmistoarkkitehtuurin määrittelyosuudessa todettiin Gacekin ym. [1995] määritelmän tuoneen lisäarvoa aikaisemmille määritelmille nimenomaan ohjelmiston arkkitehtuurin liittämisessä järjestelmän vaatimuksiin ja *sidosryhmien tarpeisiin* [Gacek ym. 95]. Taulukossa 1 on kuvattu sekä järjestelmän kehityksen eri sidosryhmiä että sidosryhmän mielenkiinnon kohteita.

Sidosryhmä	Mielenkiinnon kohteet
Asiakas	• Aikataulut ja budjettiarviot • Ohjelmiston järkevyyden ja riskien arviointi • Vaatimusten jäljitettävyys • Edistyksen mittaaminen
Käyttäjä	• Käyttöskenaarioiden ja vaatimusten yhdenmukaisuus • Laajennettavuus • Toimintakyky, luotettavuus, yhteensopivuus jne.
Arkkitehti ja järjestelmäsuunnittelija	• Vaatimusten jäljitettävyys • Tuki tradeoff-analyyseille • Järjestelmän täydellisyys, arkkitehtuurin vakaus
Kehittäjä	• Riittävän yksityiskohtaiset tiedot suunnittelua varten • Viite komponenttien valintaa ja rakentamista varten • Yhteensopivuuden säilyttäminen olemassaolevien järjestelmien kanssa
Ylläpitäjä	• Ohjeet ohjelmiston muuntamiseksi • Ohjeet arkkitehtuurin kehittämiseksi • Yhteensopivuuden säilyttäminen olemassaolevien järjestelmien kanssa

Taulukko 1: Ohjelmistoarkkitehtuurin merkitys eri sidosryhmille [Gacek ym. 95]

Järjestelmän kehittämisessä kaikki sidosryhmät – asiakkaat, käyttäjät, projektipäälliköt, ohjelmoijat – ovat kiinnostuneita järjestelmän eri piirteistä, joihin ohjelmistoarkkitehtuuri vaikuttaa. Käyttäjiä kiinnostaa järjestelmän tuleva luotettavuus sekä järjestelmän yhteensopivuus muiden järjestelmien kanssa. Asiakasta kiinnostaa, voidaanko arkki-

tehtuurin mukainen järjestelmä toteuttaa aikataulussa ja budjetin sallimissa rajoissa. Projektipäällikköä kiinnostaa aikataulun ja budjetin lisäksi, miten työtä voidaan jakaa eri kehitystyötä tekeville ryhmille, kehittäjää kiinnostaa, miten tämä kaikki voidaan toteuttaa toimivaksi koodiksi saakka. Ohjelmiston arkkitehtuuria voidaan pitää *yhteisenä kielenä*, jonka avulla osapuolet voivat ilmaista järjestelmää koskevat tarpeensa. Ilman arkkitehtuuria sidosryhmien yhteisenä kielenä laatuun ja käytettävyyteen vaikuttavia ensimmäisiä suunnittelupäätöksiä on hyvin vaikea tehdä suurten järjestelmien kehittämisessä [Clements & Northrop 96] [Bass ym. 98] [Gacek ym. 95].

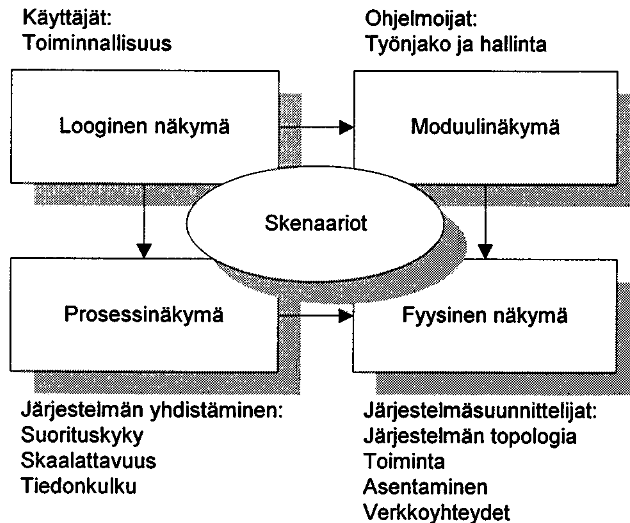
Arkkitehtuuri pitää pystyä myös kuvaamaan eri sidosryhmien käyttöä varten. Rakennusarkkitehtuurin yhteydessä tuleva rakennus kuvataan sen pohjapiirustuksena, LVI-teknisiä asioita kuvaavina putkistokaavioina, sähköteknisenä kaaviona jne. Aivan kuten rakennusarkkitehtuurissakin, ohjelmistoarkkitehtuurin kaikkien eri piirteiden esittämiseen vaaditaan useita näkymiä. Arkkitehtuuri on moniulotteinen kokonaisuus, jonka kuvaaminen vain yhtä näkymää käyttäen on mahdotonta. Arkkitehtuurin täydellisestä kuvaamisesta ei kuitenkaan vielä olla päästy yhteisymmärrykseen [Clements & Northrop 96]. Kuvausmenetelmien kehittämisen lupaavana alkuna on pidetty 4+1 näkymän mallia, joka muodostuu neljästä eri näkymästä sekä arkitehtuurin keskeisten ominaisuuksien toiminnan kuvaavista esimerkkiskenaarioista [Clements & Northrop 96] [Kruchten 95] [Booch 99] [Rational 99]. Näkymät ovat:

1. Käsitteellinen tai looginen näkymä
2. Moduulinäkymä
3. Prosessinäkymä
4. Fyysinen näkymä

+ Eri kuvaukset yhdistävät ja niiden toimintaa kuvaavat skenaariot.

Kun kehityksessä käytetään oliomenetelmää, käsitteellinen eli *looginen näkymä* kuvaa ohjelmiston luokkakaaviona. *Moduulinäkymän* avulla kuvataan järjestelmän staattinen jakautuminen kehitysympäristön ohjelmamoduuleiksi. *Prosessinäkymä* kuvaa järjestelmän samanaikaisuudet ja järjestelmän synkronointiin liittyvät piirteet. *Fyysisellä näkymällä* kuvataan järjestelmän laitteistosuhteet sekä järjestelmän mahdollinen hajautus

usean koneen välillä. Edellisten näkymien havainnollistamiseen käytetään muutamia keskeisiä käyttötilanteita kuvaavia kaavioita eli *skenaarioita.* 4+1 näkymän menetelmää tukee käytössä olevista kehitysvälineistä ainakin Rational Softwaren *Rational Rose.*



Kuva 2: 4+1 näkymän menetelmä arkkitehtuurin kuvaamiseksi [Kruchten 95]

Arkkitehtuurin kuvauksessa voidaan käyttää myös arkkitehtuureiden kuvauskieliä (*ADL – Architure Description Language*) [Gacek ym. 95]. Kuvauskieliä ei tämän tutkielman piirissä käsitellä tarkemmin aihepiirin laajuuden vuoksi, vaan tyydytään toteamaan, että niitä löytyy useita eri käyttötarkoituksia varten. Liitteessä 1 esitetään Gacekin ym. [1995] näkemys eri arkkitehtuureiden kuvauskielien sovelutuvuudesta järjestelmän arkkitehtuurin eri piirteiden kuvaukseen, sekä Cristina Gacekin [1994] arkkitehtuurin kuvaamisen eri näkymien tarvetta havainnollistava malli.

3.2.2. Järjestelmän ensimmäiset suunnittelupäätökset

Arkkitehtuurisuunnittelussa tehdään ensimmäiset ja vaikeat järjestelmän kokonaisrakennetta koskevat päätökset. Näiden päätösten myöhempi muuttaminen on erittäin vaikeaa, ja niillä on vaikutuksia koko rakennettavan ohjelmiston elinkaaren ajan [Bass ym. 98].

Järjestelmän viat ovat sitä kalliimpia korjata, mitä aikaisemmassa elinkaaren vaiheessa ne syntyvät [Pressman 97] [Haikala & Märijärvi 98]. Koska arkkitehtuurisuunnittelu on järjestelmän ensimmäinen suunnitteluvaihe, sen aikana järjestelmään syntyneet viat ovat vaikeimmat havaita ja korjata. Väärän arkkitehtuurin valinnalla saatetaan aiheuttaa katastrofaaliset seuraukset rakennettavan järjestelmän onnistumiselle [Clements & Northrop 96].

Ensimmäiset suunnittelupäätökset määrittelevät järjestelmän toteutuksen rajoitteet. Järjestelmä jaetaan komponentteihin ja niitä yhdistäviin rajapintoihin. Tämän jälkeen jokaisen komponentin on toimittava arkkitehtuurin määrittelemällä tavalla käyttäen arkkitehtuurissa määriteltyjä rajapintoja. Toteutuksen rajoitteet vaikuttavat myös koko projektin laajuiseen resurssien allokointiin. Yksittäisten komponenttien kehittäjien ei tarvitse tietää arkkitehtuurin muodostamisessa käytettyjä perusteita. Perusteilla tarkoitetaan niitä päätöksiä, joissa arkkitehtuurisuunnittelun aikana esimerkiksi uhrattin hieman järjestelmän vasteajasta ylläpidettävyyden saavuttamiseksi. Arkkitehdin ei toisaalta tarvitse tietää komponentin toteutuksessa käytetyistä algoritmeista tai monimutkaisista tietorakenteista. [Clements & Northrop 96]. Resursseiksi, joihin arkkitehtuuri vaikuttaa, voidaan laskea ihmisten lisäksi myös kehityksessä käytetyt ohjelmisto- ja laitteistotyökalut sekä uudelleenkäytettävät ohjelmistokomponentit [Pressman 97].

Arkkitehtuuri määrittelee projektin organisaatiorakenteen. Järjestelmä jaetaan arkkitehtuurisuunnittelussa moduuleiksi. Järjestelmän moduulirakennetta pidetään taas yleisenä järjestelmän toteutusryhmien muodostamisen perusteena. Tämä työnjako vaikuttaa edelleen eri toteuttavien ryhmien aikatauluttamiseen, projektin budjetointiin sekä kommunikaatiossa käytettyihin neuvottelukanaviin (palaverit, yhteiset levytilat, ilmoitustaulut jne.). Toteuttavan organisaation muodostamisen jälkeen ryhmien rakennetta on vaikeaa tai jopa mahdotonta muuttaa [Bass ym. 98].

Arkkitehtuuri joko edesauttaa tai estää järjestelmän laadullisten ominaisuuksien muodostumista. Suurien järjestelmien arkkitehtuurisuunnittelussa päätetään hyvin pitkälle, toteuttaako rakennettava järjestelmä sille asetetut laadulliset odotukset [Bass ym. 98]. Järjestelmän laadulliset ominaisuudet voidaan jakaa ominaisuuksiin, joita voi mitata

järjestelmän ajoa tutkimalla (suorituskyky, turvallisuus, luotettavuus ja toiminnallisuus), ja ominaisuuksiin, joita voi arvioida järjestelmän kehitys- ja ylläpitotyön tutkimisen avulla [Clements & Northrop 96].

Järjestelmän laadun ennustaminen sen arkkitehtuurin tutkimisen avulla. Järjestelmän tulevia ominaisuuksia voidaan arvioida ja ennustaa arkkitehtuurin analyysimenetelmien (*SAAM – Software Architecture Analysis Methods*) avulla, ennen kuin järjestelmää ryhdytään toteuttamaan [Bass ym. 98]. Tässä tutkielmassa ei käsitellä arkkitehtuureiden analyysimenetelmiä aihepiirin laajuuden vuoksi.

Arkkitehtuurin avulla voidaan suunnitella ja hallita mahdollisia järjestelmään kohdistuvia muutoksia. Arviot järjestelmän ylläpitovaiheen elinkaarikustannusten osuudesta vaihtelevat tyypillisesti 60 ja 80 prosentin välillä [Haikala & Märijärvi 98] [Bass ym. 98] [Kuusi 99]. Arkkitehtuuri jakaa ohjelmistoon kohdistuvat muutokset kolmeen osaan: paikallisiin, ei-paikallisiin ja arkkitehtuuritason muutoksiin. Paikallisella muutoksella tarkoitetaan sitä, että järjestelmään haluttu muutos voidaan suorittaa muuttamalla vain yhden komponentin sisäistä toteutusta. Ei-paikallinen muutos koskee järjestelmän useaa komponenttia mutta ei itse järjestelmän arkkitehtuuria. Arkkitehtuuritason muutos vaikuttaa järjestelmän muodostavien komponenttien välisiin vuorovaikutuksiin ja sitä kautta jokaiseen järjestelmän komponenttiin. Hyvä arkkitehtuuri ennakoi järjestelmän toimintaan tulevia muutoksia niin, että ne kohdistuvat vain paikalliselle tasolle. Järjestelmään kohdistuvat muutokset tai muutostoiveet ovat arvioitavissa ja hallittavissa arkkitehtuureiden avulla [Bass ym. 98].

Arkkitehtuuri auttaa evoluution kautta tapahtuvassa järjestelmän kehittämisessä. Arkkitehtuurin määrittelyn jälkeen järjestelmästä voidaan rakentaa testattavissa oleva ajettava protoversio [Bass ym. 98]. Komponentit voidaan toteuttaa aluksi "tyhjinä" jolloin aluksi toteutetaan pelkät sovitut rajapinnat; rajapintojen takana oleva sisäinen toteutus on pelkkä simulaatio. Simulaation avulla voidaan testata rajapintojen ja moduulijaon toimivuutta (siis itse arkkitehtuuria) mahdollisimman aikaisessa kehitysvaiheessa ongelmien tunnistamiseksi.

3.2.3. Siirrettävissä ja uudelleenkäytettävissä oleva järjestelmän abstraktio

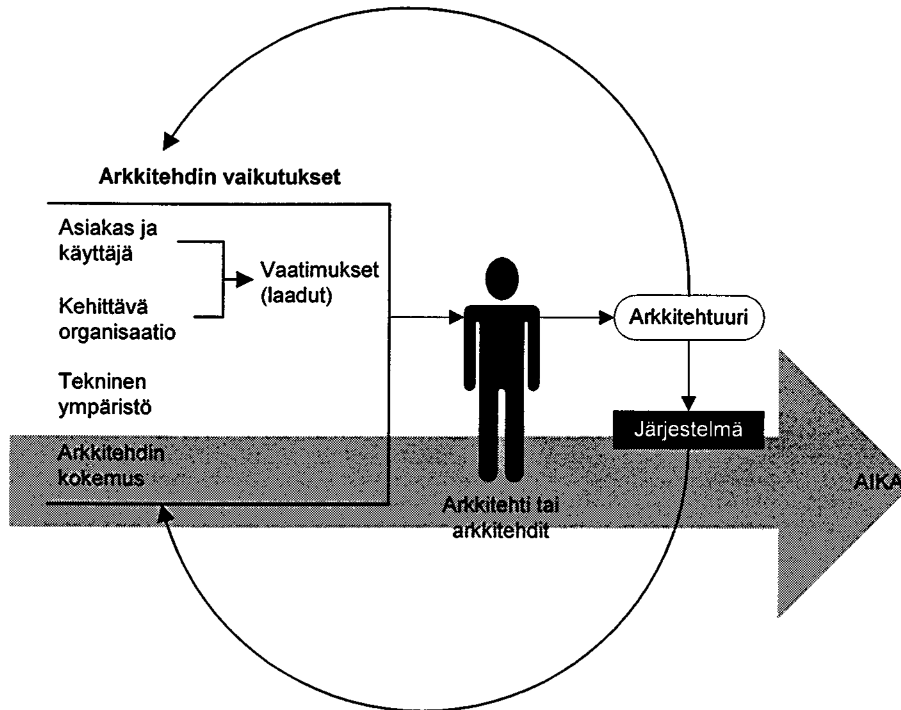
Arkkitehtuureiden uudelleenkäyttöä pidetään yhtenä keskeisempänä arkkitehtuureiden hyödyntämiskeinona. Mitä aikaisemmin uudelleenkäyttöä voidaan hyödyntää, sitä suuremmat säästöt on saavutettavissa tuotteen kehitykseen käytetyssä ajassa ja sen kehityskustannuksissa [Krueger 92] [Bass ym. 98]. Lisäksi järjestelmän laatu paranee, jos uudelleenkäytettävät komponentit ovat itsessään laadukkaita (*reusing quality artifacts* [Krueger 92]). Arkkitehtuureiden uudelleenkäyttö tapahtuu järjestelmän kehitysvaiheen alkuvaiheissa, joten uudelleenkäytöllä voidaan saavuttaa merkittäviä kustannusten säästöjä.

Arkkitehtuureiden uudelleenkäyttö liittyy toisalta arkkitehtonisen tyylin käsitteeseen, jota käsitellään tutkielman seuraavassa luvussa, sekä tyylien käyttöön osana arkkitehtuurisuunnitteluprosessia, jota käsitellään tutkielman luvussa 5.

Tässä kappaleessa käsiteltiin ohjelmistoarkkitehtuureiden tarpellisuuden syitä järjestelmän kehityksen näkökulmasta. Seuraavaksi tutkitaan arkkitehtuurin liiketoiminnallisen kiertokulun käsitteen [Bass ym. 98] avulla, mitkä tekijät vaikuttavat arkkitehtuurin muodostamiseen ja kuinka arkkitehtuuri vaikuttaa takaisin sen muodostumiseen alunperin vaikuttaneisiin tekijöihin.

3.3. Arkkitehtuurin liiketoiminnallinen kiertokulku

Arkkitehtuurin liiketoiminnallisessa kiertokulussa (ABC - *Architecture Business Cycle*) [Bass ym. 98] kuvataan, miten arkkitehtuurin ympäristö (eli konteksti jossa arkkitehtuuria kehitetään) vaikuttaa kehitettävän järjestelmän arkkitehtuuriin, ja toisaalta taas, miten järjestelmän arkkitehtuuri vaikuttaa takaisin ympäristöönsä.



Kuva 3: Arkkitehtuurin liiketoiminnallinen kiertokulku [Bass ym. 98]

Järjestelmän liiketoiminnalliset tavoitteet, järjestelmän vaatimukset, arkkitehdin kokemus, järjestelmän arkkitehtuuri ja sen mukaisesti muodostettu järjestelmä muodostavat toisiinsa vaikuttavia palautesilmukoita. Liiketoiminnassa pyritään hyödyntämään aikaisemmin rakennettuja järjestelmiä ja niiden arkkitehtuureita kasvun ja kilpailuedun saavuttamiseksi. Arkkitehtuurin liiketoiminnallinen kiertokulku toimii seuraavasti [Bass ym. 98]:

Arkkitehtuuri vaikuttaa arkkitehtuuria kehittävään organisaatiorakenteeseen. Arkkitehtuuri kuvaa kokonaisjärjestelmän muodostavat komponentit. Nämä komponentit taas hyvin pitkälle määrittelevät, kuinka työ tulee jakaa, ja työnjako taas määrää, miten kehittämisprojekti tulee organisoida. Projektin aikataulut ja budjetti laaditaan tyypillisesti komponenttia kehittävien työryhmien tarpeiden mukaisesti.

Arkkitehtuuri voi vaikuttaa organisaation tavoitteisiin. Onnistunut järjestelmä (tai arkkitehtuuri) voi saada aikaan merkittävää kilpailuetua, tai sen avulla voidaan luoda kokonaan uusia liiketoiminta-alueita [Bass ym. 98] [Porter & Millar 85] [Callon 96]. Arkki-

tehtuuria voidaan pitää myös organisaatioden tietojärjestelmiin kumuloituvana hyödynnettävissä olevana pääomana [Garlan & Perry 95].

Arkkitehtuuri voi vaikuttaa asiakkaiden järjestelmälle asettamiin vaatimuksiin. Arkkitehtuureiden avulla asiakkaalle voidaan joskus tarjota mahdollisuus hankkia päivitetty järjestelmä (samaan arkkitehtuuriin perustuen) luotettavammin, halvemmin ja nopeammin, kuin mitä järjestelmän rakentaminen alusta asti olisi vaatinut.

Järjestelmän rakentamisprosessi vaikuttaa arkkitehdin kokemuksiin. Jos järjestelmän rakentaminen onnistui hyvin kyseistä arkkitehtuuria käyttämällä, arkkitehti valitsee todennäköisesti saman lähestymistavan seuraavallakin kerralla. Jos taas arkkitehtuurivalinta osoittautuu huonoksi, seuraavalla kerralla sitä osataan välttää.

Joskus arkkitehtuurit muuttavat koko järjestelmien rakentamisen kulttuuria. Esimerkkinä ovat 60- ja 70-lukujen ensimmäiset relaatiokannat tai kääntäjien generointiohjelmat, jotka näyttivät arkkitehtonisesti esimerkkiä kaikille niiden jälkeen rakennetuille kääntäjille tai relaatiokannoille [Bass ym.98].

Ohjelmistoarkkitehtuureiden vaikutusta voidaan siis pitää kaksisuuntaisena. Arkkitehtuurin muodostamiseen vaikuttavat järjestelmän vaatimukset, laadulliset tavoitteet, tekninen ympäristö sekä arkkitehdin kokemusmaailma. Toisaalta arkkitehtuuri vaikuttaa takaisin edellä mainittuihin sen muodostumiseen vaikuttaneisiin tekijöihin. Kaksisuuntaista vaikutusta voidaan havaita myös tutkimalla arkkitehtuurin ja järjestelmien kehityksessä käytettävien prosessimallien suhteita.

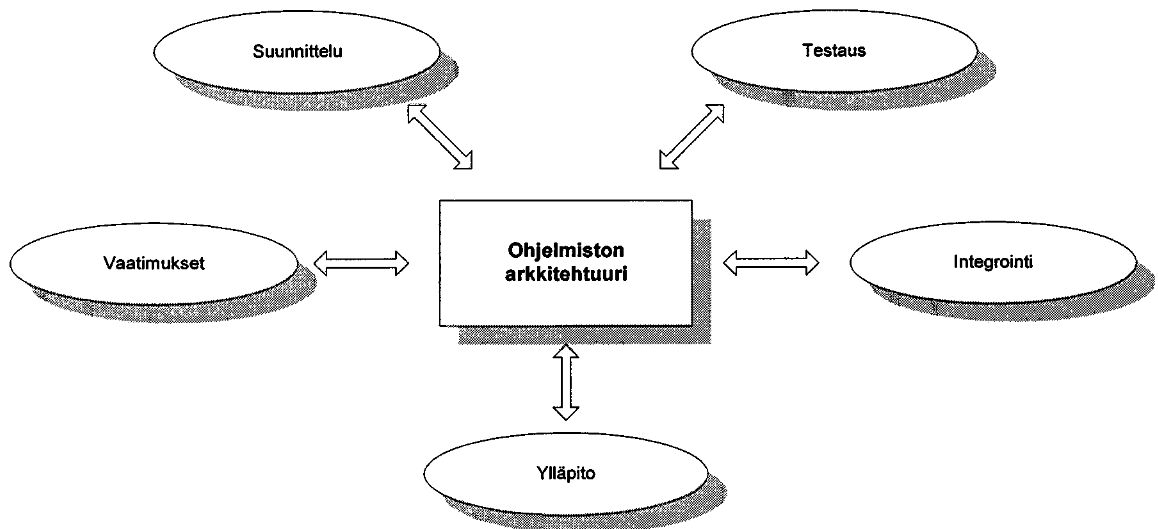
3.4. Arkkitehtuuri ja prosessimallit

Seuraavissa kappaleissa ohjelmistoarkkitehtuuria tutkitaan sekä ohjelmiston elinkaaren että kehitystyössä yleisesti käytettyjen prosessimallien kautta. Arkkitehtuurin liiketoiminnallisen kiertokulun yhteydessä on jo todettu arkkitehtuurin vaikutuksen olevan kaksisuuntainen. Tulevissa kappaleissa voidaan havaita arkkitehtuurin vaikuttavan sitä

ympäröivään maailmaan ja ympäröivän maailman vaikuttavan takaisin muodostettavaan arkkitehtuuriin – järjestelmän kaikkien eri kehitysvaiheiden aikana.

3.4.1. Arkkitehtuuri ja ohjelmiston elinkaari

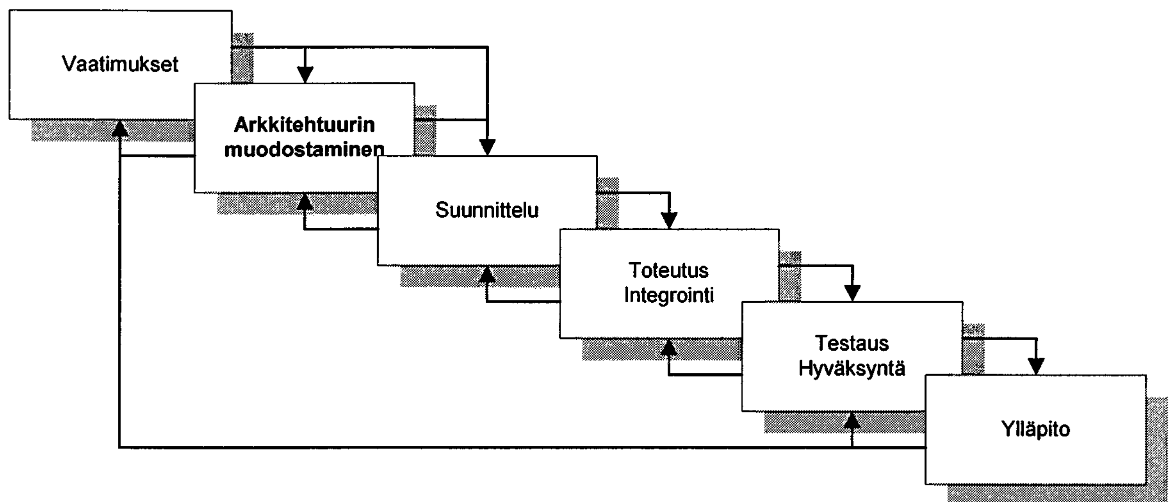
Ohjelmiston arkkitehtuuri vaikuttaa järjestelmän kehittämiseen ja laatuun koko ohjelmiston elinkaaren ajan [Gacek ym. 95] [Bass ym. 98] (Kuva 4). Arkkitehtuurin vaikutus on kaksisuuntainen. Arkkitehtuuriin vaikuttavat esimerkiksi järjestelmän vaatimukset, mutta arkkitehtuurin suunnittelussa ja valinnassa tehdyt päätökset (esim. järjestelmän suorituskykyyn kohdistuneet uhraukset järjestelmän ylläpidettävyyden saavuttamiseksi) vaikuttavat takaisin järjestelmän vaatimuksiin. Samoin järjestelmän suunnitteluvaihe, testaus, integrointi ja ylläpito asettavat arkkitehtuurille vaatimuksia, ja arkkitehtuuri vaikuttaa takaisin jokaiseen eri ohjelmiston elinkaaren vaiheeseen.



Kuva 4: Järjestelmän arkkitehtuurimääritelmän elinkaari [Gacek ym. 95]

3.4.2. Arkkitehtuuri ja vesiputousmalli

Ohjelman arkkitehtuurin muodostaminen sijoittuu järjestelmän kehittämisen prosessia kuvaavassa vesiputousmallissa vaatimusmäärityksen ja suunnittelun väliin [Gacek 94]. Kuvan 5 vesiputousmallin takaisinpäin suuntautuvat nuolet tarkoittavat Winston Roycen vuonna 1970 alkuperäisesti mahdollistamia eri kehitysvaiheiden takaisinkytkentöjä, vaikka mallin yleisessä soveltamisessa ei takaisinkytkentää yleensä sallitakaan. Mallia, jossa takaisinkytkentä ei ole sallittua, kutsutaan lineaariseksi peräkkäiseksi kehitysmalliksi (*linear sequential model*) [Pressman 97].



Kuva 5: Arkkitehtuurin muodostaminen ja vesiputousmalli [Gacek 94]

3.4.3. Kehitysvaiheen arkkitehtuuri ja WinWin-spiraalimalli

Spiraalimalliin perustuvassa järjestelmien kehityksessä arkkitehtuurin katsotaan muodostuvan iteraatioiden avulla järjestelmälle asetettujen vaatimusten ja laadullisten tavoitteiden perusteella [Boehm 95] [Boehm & Port 98] [Booch 99]. Tässä kappaleessa kuvattavat käsitteet ja menetelmät liittyvät laajemmin ottaen Boehm-koulukunnan tutkimaan järjestelmien kehittämisen MBASE-lähestymistapaan[1].

---

[1] Boehm-koulukunnalla tarkoitetaan USC/CSE:ssä Boehmin johdolla tehtävää laajahkoa MBASE-lähestymistapaa (*Model Based Architecting and Software Engineering*) koskevaa tutkimusta. Koska kehitysvaiheen arkkitehtuuri ja arkkitehtuurisuunnittelu liittyvät

Spiraalimallin avulla tapahtuvan arkkitehtuurin iteratiivisen muodostamisen käsittelemiseksi on ensin määriteltävä kehitysprosessin kolme "ankkurointipistettä" eli "virstanpylvästä". Virstanpylväällä tarkoitetaan jonkin projektin välivaiheen saavuttamista ja välivaiheen tuotosten valmistumista. Ankkurointipisteiden jälkeen kuvataan itse Win-Win-spiraalimalli, jota voidaan pitää arkkitehtuurisuunnittelua varten laajennettuna spiraalimallina. Edellisten määrittelyiden jälkeen voidaan kuvata kuinka kehitysvaiheen arkkitehtuuri muodostetaan iteratiivisesti kehitysvaiheen tavoitteiden avulla.

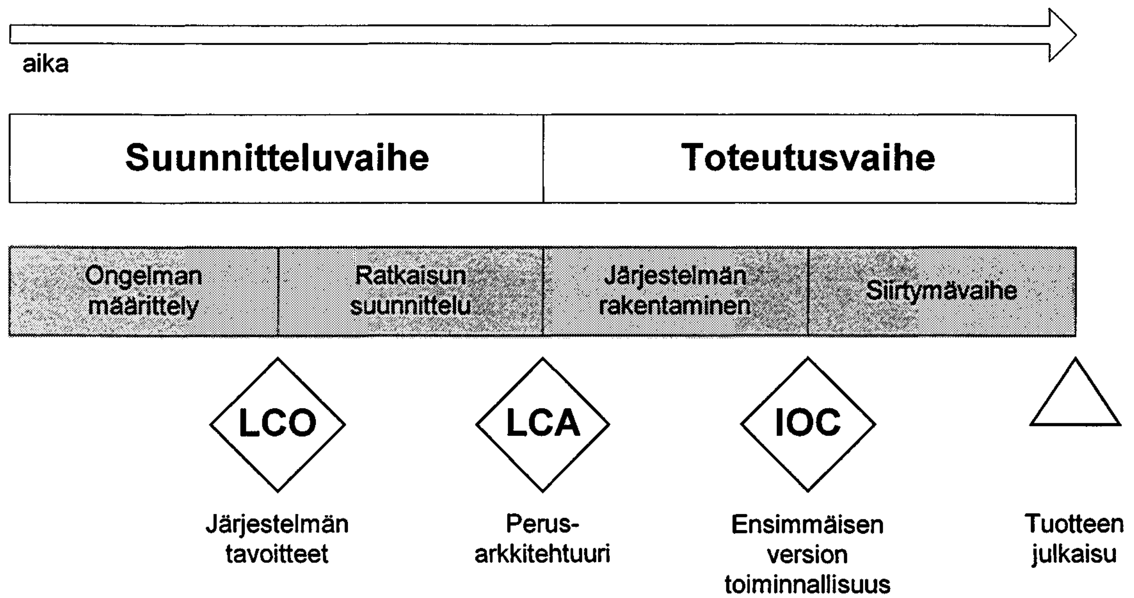
Ankkurointipisteellä tarkoitetaan jonkin kehitysprosessin virstanpylvään saavuttamista. Virstanpylvään saavuttamiseen voi liittyä joidenkin konkreettisten tuotosten valmistuminen. Kehittämisprojektin kolmeksi tärkeäksi ankkurointipisteeksi esitetään seuraavat [Boehm 95] [Boehm & Port 98] [Booch 99]:

1. Kehitysvaiheen tavoitteiden määritelmä (*LCO – life cycle objectives*)
2. Kehitysvaiheen arkkitehtuurin määritelmä (*LCA – life cycle architecture*)
3. Ensimmäisen version toiminnallisuuden määritelmä (*IOC – initial operational capacity*)

Kuvassa 6 esitetään kehittämisprojektin eri ankkurointipisteiden ajallinen suhde järjestelmän kehittämisen prosessimalliin.

---

keskeisesti ko. menetelmään, MBASE-lähestymistapa on kuvattu lyhyesti liitteessä 3. Katso myös kappale 2.6., arkkitehtuureiden nykytutkimus.



Kuva 6: Järjestelmän kehityksen ankkurointipisteet. Muunneltu [Boehm & Port 98]:sta.

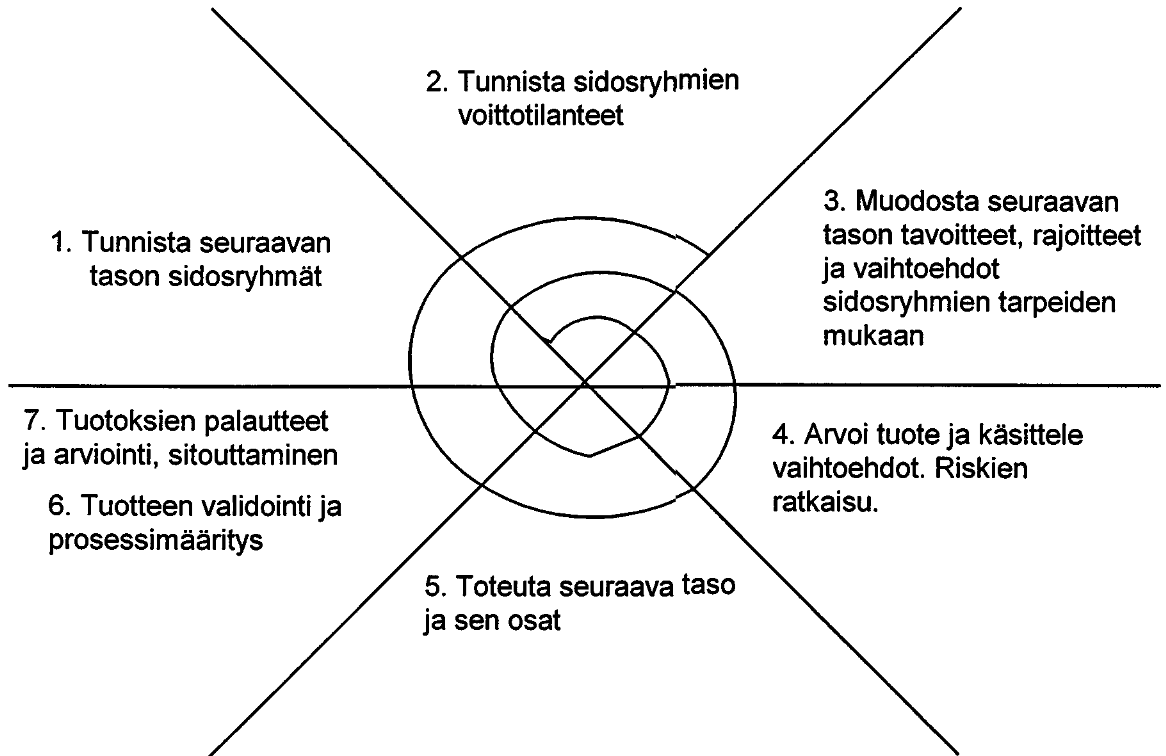
Kehitysvaiheen tavoitteiden määrittely (LCO) saadaan järjestelmän ongelmamäärityksen tuloksena. Vesiputousmalliin suhteutettuna tällä tarkoitettaisiin järjestelmän vaatimusmääritystä. Tavoitteiden määrittelyn aikana rajataan järjestelmän kattavuus: mitä järjestelmään tässä kehitysvaiheessa sisällytetään ja mitä ei. Lisäksi sen avulla varmistetaan organisaation liiketoiminnallinen sitoutuminen järjestelmän kehittämiseen vähintään kehitysvaiheen arkkitehtuurin suunnitteluvaiheen ajaksi [Boehm 95] [Boehm ym. 98].

Toisena kehitysvaiheen virstanpylväänä pidetään *kehitysvaiheen arkkitehtuurin muodostamista* (*LCA- life cycle architecture*). Tässä vaiheessa muodostetaan kehitysvaiheen tavoitteet toteuttava järjestelmän arkkitehtuurin suunnitelma. Jälleen vesiputousmalliin suhteutettuna tällä tarkoitettaisiin suunnitteluvaiheen ensimmäisiä järjestelmän rakennetta koskevia päätöksiä. Arkkitehtuurin avulla johto voi varmistaa asetettujen tavoitteiden toteutumisen ja päättää sitoutumisestaan järjestelmän kehitykseen. Kehitysvaiheen arkkitehtuuria voidaan siis pitää kehitysvaiheen tavoitteiden toteutuksen suunnitelmana [Boehm 95] [Boehm ym. 98].

Viimeisen virstanpylvään - *ensimmäisen version toiminnallisuuden määritelmä* (IOC) – keskeinen sisältö on toteuttaa arkkitehtuuri käytännössä. Lisäksi IOC:n sisällöksi mainitaan ohjelmiston käyttöönoton valmistelu, ostettujen komponenttien lisensointi, ohjelmiston dokumentointi jne., tulevan asennusympäristön valmistelu sekä käyttäjien tukitoimintojen suunnittelu ja perustaminen [Boehm 95].

WinWin-spiraalimalli on alkuperäisen spiraalimallin arkkitehtuurisuunnittelun huomioon ottava laajennus. Alkuperäisen vuoden 1988 spiraalimallin mukainen ohjelmiston kehitys alkoi tulevan iteraatiokierroksen tavoitteiden, rajoitteiden ja vaihtoehtojen määrittelystä. Sen käytön ongelmaksi muodostui kuitenkin, mistä nämä tavoitteet ja rajaukset saataisiin. Ongelmat korjattiin muodostamalla laajennettu WinWin-spiraalimalli [Boehm 95].

WinWin-spiraalimallin mukainen järjestelmän kehitys alkaa seuraavan iteraatiokierroksen sidosryhmien tunnistamisesta ja jatkuu sidosryhmien voittotilanteiden (*Win-Win-situation*) tunnistamiseen. Tämän jälkeen määritellään sidosryhmien tarpeiden mukaiset järjestelmän iteraatiokierroksen tavoitteet, rajoitteet ja kehitysvaihtoehdot. Näiden vaiheiden jälkeen WinWin-spiraalimalli jatkuu kuten alkuperäinen spiraalimalli: arvioidaan riskit, toteutetaan, arvoidaan lopputulos ja sitoutetaan sidosryhmät tulevaan kierrokseen [Boehm 95] [Boehm ym. 98]. WinWin-spiraalimallin iteraatiokierroksen vaiheet ovat yhdenmukaisia kappaleessa 3.1. esitetyn boehmilaisen ohjelmistoarkkitehtuurin määritelmän [Gacek ym. 95] kanssa. Määritelmässähän laskettiin ohjelmiston arkkitehtuuriin kuuluvan mm. järjestelmän sidosryhmien asettamat tavoitteet ja rajoitteet – "arkkitehtuuri ei koskaan esiinny yksinään vaan on sidoksissa järjestelmän vaatimuksiin jne." [Gacek ym. 95].



Kuva 7: Win-Win spiraalimalli [Boehm 95]

Boehmin ja Portin [1998] mukaan LCO ja LCA -ankkurointipisteitä voidaan käyttää myös spiraalimallin mukaisen järjestelmän iteratiivisen kehittämisen yhteydessä. Taulukko 2 sisältää esimerkin järjestelmän kehityksen alkuvaiheeseen soveltuvista iteraatiovaiheista. Spiraalimallin käytön yhteydessä kehitysvaiheen arkkitehtuurin muodostamisen keskeisenä ajatuksena pidetään sekä kehitysvaiheen tavoitteiden että kehitysvaiheen arkkitehtuurin samanaikaista iteratiivista muodostamista. Taulukossa 2 on kuvattu miten itse kehitysvaihe, kehitysvaiheen tavoitteet (LCO) ja kehitysvaiheen arkkitehtuuri (LCA) liittyvät toisiinsa.

Kehitysvaihe	Kehitysvaiheen tavoitteet (LCO)	Kehitysvaiheen Arkkitehtuuri (LCA)
Toiminnallisen konseptin kehittäminen	• Järjestelmän kattavuus ja korkean tason tavoitteet - Järjestelmän rajaus	• Järjestelmän tarkoitusten kuvaus ja kattavuus kehitysvaiheittain • Toiminnallisen konsep-

	- Ympäristön parametrit ja oletukset - Evoluution määrittely (kehitysvaiheet) • Toiminnallinen konsepti	tin kuvaus inkrementaalista kehitysvaihetta kohden
Järjestelmän prototyypit	• Tärkeimpien käyttötapojen tunnistaminen • Kriittisten riskien ratkaisu	• Käyttötapausten kuvaus • Tärkeimpien riskien ratkaisu
Vaatimusten määritys	• Korkean tason toiminnat, liittymät, laatu, ja järjestelmän ominaisuudet kuten: - Kasvuvektorit - Priorisoinnit • Sidosryhmien samanaikaisten tarpeiden tunnistaminen oleennaisissa ongelmissa	• Funktioiden, liittymien ja laatuattribuuttien määrittely per kehitysvaihe. - Tulevaisuudessa selvitettävien osuuksien tunnistaminen (*TBD, to be determined items*) • Sidosryhmien samanaikaisten tarpeiden priorisointi
Järjestelmän ja ohjelmiston arkkitehtuurin määritelmä	• Ainakin yhden sopivan arkkitehtuurin määritteleminen - Fyysisten ja loogisten elementtien ja suhteiden tunnistaminen - Uudelleenkäytettävien komponenttien tunnistaminen • Mahdottomien arkkitehtuureiden tunnistaminen	• Arkkitehtuurin valinta ja kehitys inkrementaalista kehitysvaihetta kohden - Fyysiset ja loogiset komponentit, liittymät, konfiguraatiot, rajoitteet - Valmiit komponentit (COTS), uudelleenkäytön mahdollisuuksien valinta - Sovellusalueen arkkitehtuurin ja arkkitehtuu-

		rillisten tyylien valinta • Arkkitehtuurin ajan myötä tapahtuvan kehityksen tunnistaminen
Kehitysvaiheiden suunnitelman määrittely	• Kehitysvaiheen sidosryhmien tunnistaminen - Käyttäjät, asiakkaat, kehittäjät, ylläpitäjät, suuri yleisö, muut? • Kehityksessä käytetyn prosessimallin tunnistaminen - Korkean tason vaiheet, kehitystasot (inkrementit) • Korkean tason Miksi, Mitä, Kuka, Milloin, Missä, Kuinka, Kuinka Paljon (WWWWWHH) vaiheittain tunnistaminen	• WWWWWHH suunnitelma per vaiheen alkuperäisen toiminnallisen kapasiteetin suunnitelma (IOC) - Osittainen kehitysmalli, tärkeimpien TBD:iden määrittely myöhempiä kehitysvaiheita varten.
Järjestelmän arvio	• Ylläolevien elementtien yhdenmukaisuuden varmistaminen - Analyysit, mittaukset, prototyypit, simulaatiot jne. - Liiketoiminnallisen hyödyn analysointi vaatimusten ja arkkitehtuurin perusteella	• Edellisten yhtenäisyyden varmistaminen • Kaikki suurimmat riskit ratkaistu riskinhallintasuunnitelman avulla

Taulukko 2: LCA ja LCO kehitysprosessin ankkurointipisteet [Boehm & Port 98]

Spiraalimalliin perustuvassa järjestelmien kehityksessä arkkitehtuurin katsotaan muodostuvan iteratiivisesti sidosryhmien järjestelmälle asettamien vaatimusten ja laadullisten tavoitteiden perusteella. Iteraatiokierros alkaa seuraavan tason sidosryhmien tarpeiden tunnistamisella, joiden avulla saadaan määritellyksi kehitysvaiheen tavoitteet. Kehitysvaiheen arkkitehtuurilla tarkoitetaan arkkitehtuurisuunnitelmaa, joka toteuttaa aikaisemmin määritellyt järjestelmän tavoitteet. Iteraatioita jatketaan, kunnes järjestelmä on valmis. Välivaiheina voidaan pitää järjestelmän kaikkien tavoitteiden määrityksen saavuttamisen virstanpylvästä (LCO), tavoitteet toteuttavan arkkitehtuurisuunitelman muodostamista (LCA) ja arkkitehtuurin käytännössä toteuttavan ensimmäisen toiminnallisen version virstanpylvästä (IOC).

3.5. Ohjelmistoarkkitehtuureiden yhteenveto

Ohjelmistoarkkitehtuurille voidaan kirjallisuudesta löytää useita laajuudeltaan vaihtelevia määritelmiä. Määritelmiä yhdistävänä piirteenä voidaan pitää ohjelmiston arkkitehtuurin muodostumista ohjelmiston toteuttavista komponenteista ja niiden välisistä liittymistä.

Ohjelmistoarkkitehtuureita pidetään tärkeinä, koska ne toimivat järjestelmän kehittämisen eri sidosryhmien välisenä kommunikaatiomenetelmänä, sisältävät järjestelmän ensimmäiset suunnittelupäätökset sekä toimivat mahdollisesti siirrettävänä ja uudelleenkäytettävänä järjestelmän abstraktiona.

Ohjelmistoarkkitehtuurin vaikutus on kaksisuuntainen: järjestelmän arkkitehtuuri vaikuttaa järjestelmää kehittävään organisaatioon – ja päin vastoin. Kaksisuuntainen vaikutus on havaittavissa tutkimalla sekä ohjelmiston elinkaarta että järjestelmän kehittämisen prosessimalleja.

4. ARKKITEHTONISET TYYLIT

Tämän luvun tarkoituksena on määritellä arkkitehtonisen tyylin käsite ja tyylien tyypillisiä sovelluskohteita. Aihetta lähestytään ensin kirjallisuudessa esitettyjen arkkitehtonisen tyylin eri määritelmien kautta, jonka jälkeen kappaleessa 4.2. käsitellään, miten arkkitehtoniset tyylit muodostuvat järjestelmien kehitystyössä hyväksi havaittujen ratkaisumallien seurauksena.

Kappale 4.3. esittää esimerkkejä yleisimmistä arkkitehtonisista tyyleistä ja käsittelee niiden soveltuvuutta arkkitehtuurisuunnittelussa esiintyvien eri tyyppisten ongelmien ratkaisemiseen. Tämän jälkeen tutkitaan vielä järjestelmän arkkitehtuurin kuvaamista arkkitehtonisten tyylien avulla.

4.1. Arkkitehtonisen tyylin määritelmiä

Arkkitehtoniselle tyylille löytyy kirjallisuudesta useita eri määritelmiä. Websterin sanakirja [Webster 99] määrittelee sanan tyyli seuraavasti: *"tietty tapa tai tekniikka, miten jokin asia on tehty, luotu tai esitetty."* Edellisestä johtaen arkkitehtonisen tyylin määritelmänä voitaisiin pitää: "tietty tapa tai tekniikka, miten jokin tietojärjestelmien kehittämisen ongelma ratkaistaan." Seuraavissa kappaleissa esitetään ja arvioidaan kirjallisuudessa esitettyjä arkkitehtonisen tyylin määritelmiä.

4.1.1. Garlanin ja Shaw'n vuoden 1994 määritelmä

Arkkitehtuureita käsittelevässä kirjallisuudessa ehkä suosituimmaksi arkkitehtonisen tyylin määritelmäksi on muodostunut Garlanin ja Shaw'n [1994] esittämä määritelmä, jossa määritellään ensin yksittäinen arkkitehtuurin esiintymä:

Yksittäisen järjestelmän arkkitehtuuri on joukko tietojenkäsittelyä suorittavia komponentteja – tai lyhyemmin komponentteja – ja näiden komponenttien välisten vuorovaikutusten kuvauksia – liittymiä.

Ja arkkitehtonisen tyylin määritelmä edellistä apuna käyttäen:

Arkkitehtoninen tyyli määrittelee yksittäisen arkkitehtuurin esiintymän yhteydessä käytetyn komponenttien ja niiden välisten liittymien sanaston sekä joukon rajoitteita sille, kuinka nämä komponentit ja liittymät voidaan yhdistää [Garlan & Shaw 94].

Garlan ym. analysoivat edellisen määritelmän pohjalta saatavaksi neljä arkkitehtonisen tyylin määrittelemää piirrettä [Garlan ym. 94]:

1. Tyyli määrittelee tyyliin kuuluvat komponentit ja niiden väliset liittymät kuvaavan sanaston. Sanastolla tarkoitetaan tyylien yhteydessä käytettyjä osakokonaisuuksien nimiä kuten asiakas, palvelin, putki ja suodatin tai kerroksittainen rakenne.
2. Tyyli määrittelee tyyliin kuuluvien komponenttien yhdistämisen säännöstön (*configuration rules*). Säännöstö on joukko topologisia rajoitteita, jotka määräävät, kuinka tyyliin kuuluvia komponentteja on sallittu yhdistää. Säännöstö voi esimerkiksi kieltää taaksepäin suuntautuvan palautteen tietyn tyyppisen putki ja suodatin-menetelmän rakenteen yhteydessä – tieto virtaa putkessa vain yhteen suuntaan (putki ja suodatin -menetelmä on selvitetty tarkemmin kappaleessa 4.3.1.).
3. Kun tyylin muodostavat komponentit yhdistetään tyylin säännöstön määrittelemällä tavalla, niiden avulla saadaan aikaan jonkin toiminnon suorittava kokonaisuus. Tyyli määrittelee siis semanttisen tulkinnan.
4. Tyylit määrittävät tyyliin käyttöön perustuvan järjestelmän arviointiin soveltuvat analyysit. Esimerkiksi järjestelmästä, joka on toteutettu asiakas-palvelin - arkkitehtuuria käyttäen, voidaan analysoida, kuinka järjestelmä tunnistaa ja estää mahdolliset tietojen lukitustilanteet (*synchronization and deadlock detection*).

4.1.2. Muut yleiset arkkitehtonisen tyylin määritelmät

Tähän kappaleeseen on koottu muut arkkitehtonisen tyylin yleisesti kirjallisuudessa käytetyt määritelmät. Jokainen määritelmä lähestyy tyylin käsitettä hieman eri lähtökohdasta: joko alhaalta ylös, ylhäältä alaspäin tai johtamalla tyylin määritelmän tyylien varsinaisen käyttötarkoituksen kautta. Vaikka yhtäkään allaolevista määritelmistä ei pidetä täydellisenä, yhdessä ne antavat kuitenkin arvokasta näkemystä arkkitehtonisten tyylien ymmärtämiseksi.

Perryn ja Wolfin [1992] määritelmän lähestymistapa on alhaalta ylös; arkkitehtuurin määritelmän avulla todetaan arkkitehtonisen tyylin olevan abstraktiotasoltaan korkeammalla kuin arkkitehtuurin konkreettinen instanssi.

Jos arkkitehtuuri on arkkitehtonisten elementtien muodollisesti määritelty järjestys, arkkitehtoninen tyyli on se, mikä abstrahoi nämä elementit ja niiden piirteet muista spesifisistä tyyleistä. Arkkitehtoninen tyyli on vähemmän rajoittava ja vähemmän täydellinen määritelmä kuin yksittäisen tietyn arkkitehtuurin esiintymän määritelmä [Perry & Wolf 92].

Seuraava Garlanin ja Perryn [1995] hieman vajavainen määritelmä on jälleen ylhäältä alaspäin lähestyvä tyylin määritelmä:

Arkkitehtoninen tyyli määrittelee muodon ja rakenteen rajoitteet joukolle arkkitehtuurin esiintymiä [Garlan & Perry 95].

Arkkitehtonisen tyylin voi määritellä myös sen käyttötarkoituksen kautta [Garlan ym. 94]:

Arkkitehtoninen tyyli on järjestelmien muodostamisessa käytetty joukko toistuvia rakenteita, joita voidaan rutiininomaisesti käyttää tietyn tyyppisten ongelmien ratkaisemiseksi.

Edellisen määritelmän huonona puolena voidaan pitää sitä, että sen perusteella arkkitehtonisena tyylinä voitaisiin pitää myös suunnittelumalleja (design patterns) [Gamma ym. 94]. Suunnittelumallit ovat käsitteellisesti hyvin lähellä arkkitehtonisen tyylin käsitettä, mutta niillä on käytönnössä muutamia merkittäviä eroja. Arkktehtonisten tyylien suhteita muihin uudelleenkäytön mekanismeihin on käsitelty tarkemmin kappaleessa 5.4., tyylien uudelleenkäytön yhteydessä.

Esitetään analogian vuoksi vielä arkkitehtoniset tyylit rakennusarkkitehtuureihin rinnastava määritelmä [Bass ym. 98]:

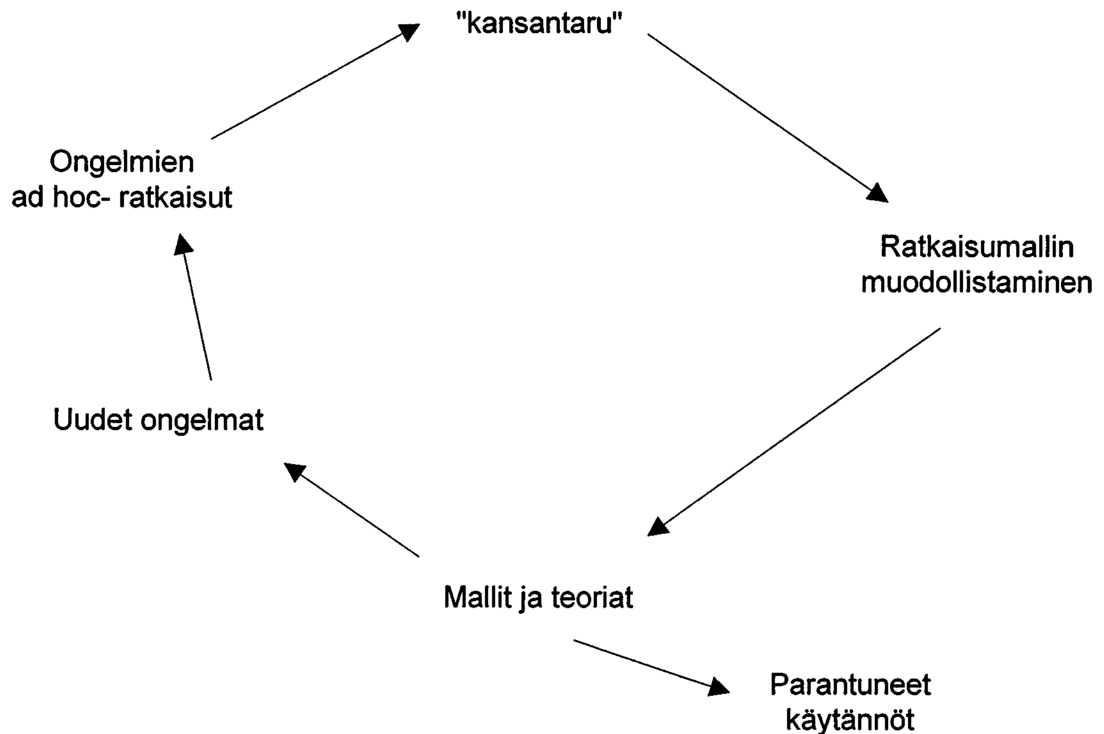
Ohjelmiston arkkitehtonisella tyylillä tarkoitetaan suurinpiirtein samaa asiaa kuin rakennusarkkitehtuurissakin. Tyyli sisältää muutamia avainpiirteitä ja sääntöjä, miten näitä avainpiirteitä voidaan yhdistellä, jotta arkkitehtoninen yhtenäisyys säilyy.

Tämän tutkielman tarpeisiin käytetään Garlanin ja Shawin [1994] määritelmää, jossa arkkitehtonisen tyylin todetaan määrittelevän tyylin sanaston, tyylin muodostavien komponenttien ja niiden välisten liittymien rakenteen rajoitteet ja tyylin semantiikan [Garlan & Shaw 94]. Tutkielmassa käsitellään lisäksi tyylejä niiden pääasiallisen käytön kautta; tyyliä pidetään joukkona uudelleenkäytettäviä rakenteita, joita voidaan käyttää tiettyjen järjestelmän toiminnallisten ja laadullisten tavoitteiden saavuttamiseksi.

4.2. Tyylien muodostuminen

Tämä kappaleessa tutkitaan arkkitehtonisten tyylien muodostumista. Ohjelmistoarkkitehtuureiden ja arkkitehtuuriajattelun yksi tärkeimpiä saavutuksia on erilaisten yleisten ratkaisumallien löytäminen ja uudelleenkäyttö järjestelmien muodostamisessa ja niiden sisäisessä organisoinnissa. Useat näistä ratkaisumalleista – arkkitehtonisista tyyleistä – ovat muodostuneet vuosien kuluessa kun ohjelmistojen kehittäjät ovat uudelleenkäyttäneet hyväksi havaitsemiaan ratkaisumalleja ratkaistakseen tietyntyyppisiä järjestelmien kehittämisen ongelmia [Garlan & Shaw 96]. Arkkitehtonisten tyylien tärkein hyöty on

juuri näiden hyväksi havaittujen rakenteiden rutiininomainen uudelleenkäyttö uusien järjestelmien arkkitehtuureiden muodostamisessa [Garlan ym. 94].



Kuva 8: Arkkitehtonisen tyylin muodostuminen [Garlan & Shaw 96]

Kuvassa 8 on esitetty arkkitehtonisen tyylin muodostumisen vaiheet. Tyylin muodostuminen alkaa kun uusi ongelma ratkaistaan millä tahansa käytettävissä olevalla keinolla, intuition tai "raa'an voiman" avulla (ad hoc -ratkaisu). Saman ongelman joutuvat ratkaisemaan yleensä useat toisistaan tietämättömät suunnittelijat, jolloin ongelman ratkaisemiseksi muodostuu epäilemättä useita erilaisia ratkaisutapoja. Ongelma siis ratkaistaan – ja ratkaistaan taas uudelleen usean eri suunnittelijan toimesta.

Kun suunnittelijat pohtivat tietyn ongelman eri ad hoc -ratkaisuvaihtoehtojen hyviä ja huonoja puolia, hyväksi havaitut ratkaisumallit siirtyvät vähitellen "kansantaruiksi", joista puhutaan käytävillä ja kahvihuoneissa. Hyvän ad hoc -ratkaisun siirtyminen kansantaruksi saattaa kestää hyvinkin kauan, koska suunnittelija ei välttämättä tunne tarvetta kertoa edelleen toimivasta ratkaisusta.

Kun kansantaruun perustuvasta ratkaisumallista tulee ajan myötä yhä systemaattisemmin käytetty, kansantaru muodollistuu – ratkaisumalli kirjataan heuristiikaksi ja joukoksi toteuttamisen sääntöjä. Seuraavassa tyylin kehitysvaiheessa muodollistettu menetelmä on riittävän kehittynyt muunnettavaksi tyyliä koskeviksi malleiksi ja teorioiksi. Malleihin lisätään riittävä matemaattinen tausta. Mallien muodostuminen auttaa parantamaan vallitsevia käytäntöjä, ja käytännöt edelleen parantamaan malleja [Garlan & Shaw 96].

Seuraava kappale esittelee muutamia alunperin järjestelmien kehittämisen kansantaruista muodostuneita ratkaisumalleja tietyntyyppisten vakio-ongelmien ratkaisemiseksi.

4.3. Esimerkkejä arkkitehtonisista tyyleistä

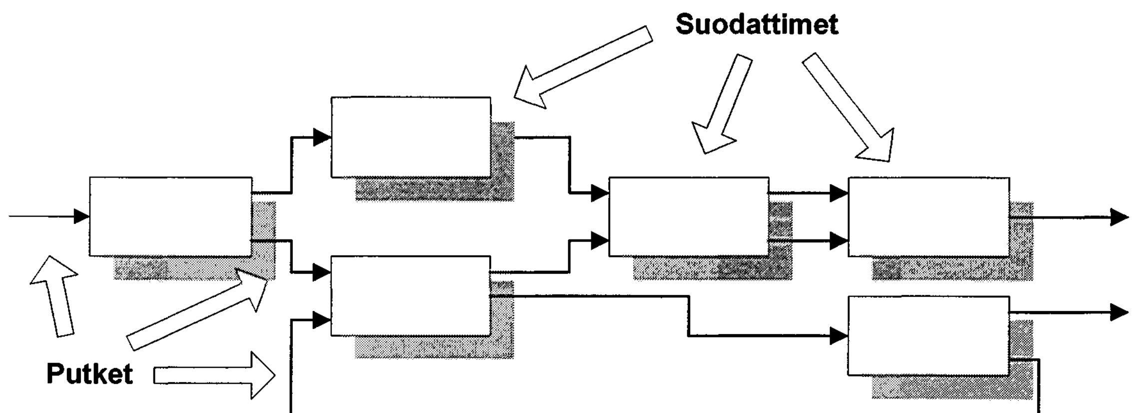
Arkkitehtonisia tyylejä ja niiden käyttöä lienee helpoin ymmärtää esimerkkien kautta. Tässä kappaleessa kuvataan muutamia yleisesti tunnistettuja ja järjestelmien kehityksessä käytettyjä arkkitehtonisia tyylejä.

Arkkitehtoniset tyylit jaottuvat yleensä neljään eri ryhmään [Shaw 95]: oliopohjaisiin arkkitehtuureihin, joiden keskeinen ajatus on tiedon piilottaminen, tilapohjaisiin arkkitehtuureihin kuten liitutaulumenetelmä, palaute- ja kontrollipohjaisiin arkkitehtuureihin (putket ja suodattimet) ja arkkitehtuureihin, jotka korostavat järjestelmän reaaliaikaisia ominaisuuksia (keskitetyn kontrollin järjestelmän kontrollerimalli).

4.3.1. Putket ja suodattimet

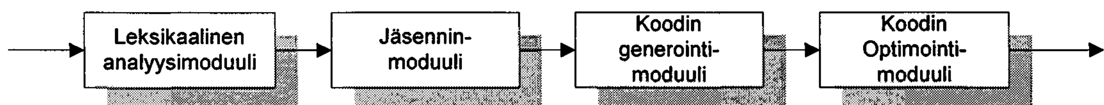
Putket ja suodattimet ovat järjestelmän rakenteellisen muodostamisen malli. Putki ja suodatin -menetelmässä jokaisella komponentilla on joukko sisääntuloja (inputs) joista komponentti lukee syötteensä sekä joukko ulostuloja (outputs) johon komponentti toimittaa tulosteensa [Garlan & Shaw 96] (Kuva 9). Tiedon käsittely tyylin yhteydessä tapahtuu yleensä inkrementaalisesti, eli lopputulos muodostuu erilaisten välivaiheiden kautta.

Komponenttien välisiä tietovirtoja kutsutaan putkiksi, koska ne toimivat välittäjinä komponenttien sisään- ja ulostulojen välillä. Itse komponentteja taas kutsutaan suodattimiksi, koska sisääntulevasta putkesta luetaan jotain ja komponentit "suodattavat" tiedon toiseen muotoon ja toimittavat sen edelleen seuraavaan ulostuloon. Tyylin tunnetuimmat erikoistapaukset ovat "putkistot" (pipelines), jotka rajoittavat komponenttien topologian peräkkäiseksi (kuvan 10 kääntäjäesimerkki), ja "rajoitetut putket" jossa putkessa samanaikaisesti sijaitsevan tiedon määrää on rajoitettu jollakin tavalla.



Kuva 9: Putket ja suodattimet [Garlan & Shaw 96]

Eräs yleinen putki ja suodatin -menetelmää käyttävä esimerkki on kääntäjien muodostamisessa käytetty viitemalli (*reference model*), jossa kääntäjän muodostavat peräkkäiset putkilla yhdistetyt leksikaalisen analyysin suorittava moduuli, jäsenninmoduuli (*parser module*), koodin generointimoduuli ja lopulta koodin optimointimoduuli (viitemalleja käsitellään hieman tarkemmin arkkitehtuurisuunnittelun yhteydessä tutkielman luvussa 5).



Kuva 10: Putki ja suodatin-rakenne kääntäjän arkkitehtonisena tyylinä

Käytetään putkien ja suodattimien inkrementaalista toimintaa kuvaavana esimerkkinä C-kielen kääntäjää. Kääntäjän toiminta alkaa leksikaalisesta analyysista, joka etsii annetusta ASCII-muotoisesta tietovirrasta erilaisia avainsanoja (*token*). Avainsanoja ovat mm. C-kielen avainsanat `int`, `main`, `void` ja `for`. Avainsana voi olla myös vertailumerkki (==, <=, >=), muuttujaan sijoitettava kokonaisluku (2, -2) jne. Leksikaalisen analyysin suorittava moduuli suodattaa alkuperäisestä ASCII-muotoisesta tekstitiedostosta avainsanoja seuraavan moduulin käyttöön.

Leksikaalisen analyysin suorittava moduuli antaa nämä avainsanat jäseninmoduulille, joka muodostaa niistä semanttisen tulkinnan (eli mitä löydetyillä avainsanoilla tarkoitetaan). Semanttinen tulkinta on tyypillisesti eräänlainen puurakenne. Semanttisen analyysin aikana koodista tunnistetaan mm. lohkorakenteet. Esim. C-kielen lohkorakenne

```
int main (void)
{
    ...
}
```

muodostaa semanttisen kokonaisuuden nimeltä pääohjelma. Jäsennin suodatti siis leksikaalisen analyysin avainsanoista semanttisen tulkinnan.

Jäseninmoduuli toimittaa muodostamansa puumaisen esityksen koodin semantiikasta (jäsennys) koodin generointimoduulille, joka muodostaa annetusta puusta kohdekoodin. Luotu kohdekoodi annetaan vielä edelleen koodin optimointimoduulille, joka poistaa koodiin jääneet tarpeettomat osuudet, esim. tyhjät silmukat, kuten alla olevassa esimerkkikoodissa:

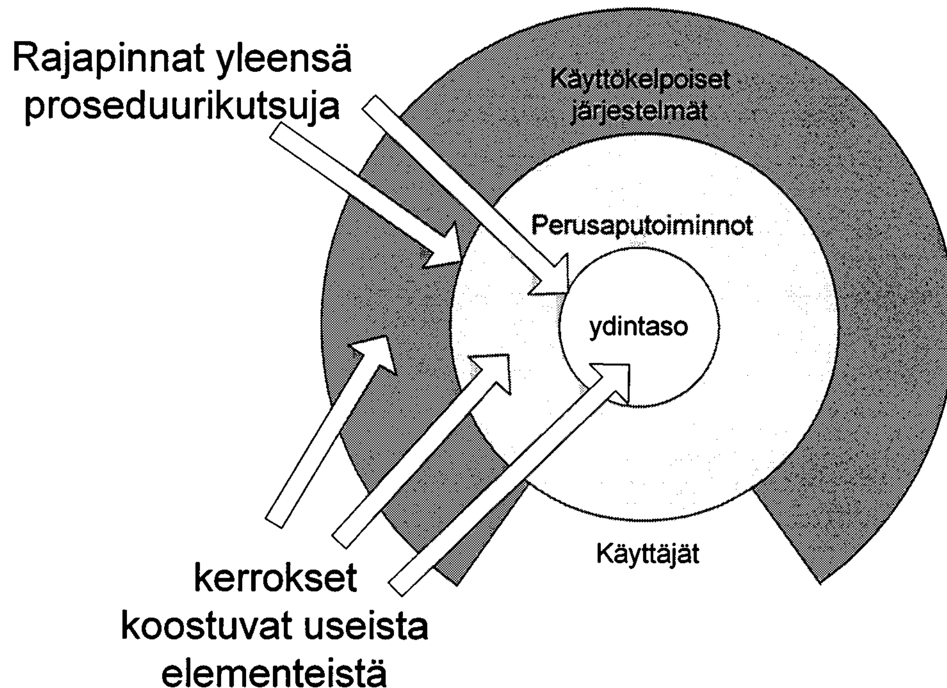
```
for (I=0; I<20000; I++) {

  /* printf ("%n",I); */ ← silmukka unohdettu poistaa rivin kommentoinnin jälkeen

}
```

Optimoinnin jälkeen optimointimoduuli kirjoittaa levylle valmiin suoritettavan koodin.

Kääntäjän moduulirakenteen keskeisenä ajatuksena voidaa pitää sitä, että jokainen yksittäinen moduuli suorittaa vain yhden tehtävän. Leksikaalisen analyysin suorittava moduuli osaa lukea tekstitiedostoa ja etsiä sieltä tiettyjä avainsanoja. Moduuli ei kuitenkaan ymmärrä, mitä näillä avainsanoilla tarkoitetaan. Jäsenninmoduuli taas ei tiedä mitään tekstitiedostojen lukemisesta, mutta se ymmärtää, mitä esim. pääohjelmalla ja funktiokutsuilla tarkoitetaan. Jäsennin kertoo koodin generointimoduulille esim. kun pääohjelma on kokonaan tulkittu, jolloin koodingenerointimoduuli tuottaa siitä suoritettavan konekielisen koodin jäsentimen antamien ohjeiden mukaisesti. Koodin generointimoduuli ei kuitenkaan enää ymmärrä koodin semantiikkaa – koodin generoinnilla tarkoitetaan absoluuttisten konekielisten peräkkäisten käskyjen muodostamista kuten hyppykäskyjä tiettyyn muistiosoitteeseen, rekisterien arvojen kasvattamista jne. Optimointimoduuli osaa esim. ainoastaan poistaa tarpeettomaksi havaitsemiaan rakenteita.

4.3.2. Kerroksittaiset järjestelmät

Kerroksittaiset järjestelmät (*layered systems* tai *abstract machine model*) muodostuvat hierarkkisista tasoista, niin että jokainen taso tarjoaa palveluita yläpuolella oleville kerroksille. Jokainen taso toimii myös alla olevan kerroksen asiakkaana eli taso käyttää hyväkseen alemman kerroksen tarjoamia palveluita. Joissakin kerroksittaisissa järjestelmissä sisempien kerrosten palvelut on piilotettu niin, että vain viereiset kerrokset voivat käyttää kerroksen tarjoamia palveluita [Garlan & Shaw 96].



Kuva 11: Kerroksittaiset järjestelmät [Garlan & Shaw 96]

Käytännön tyylin ilmentyminä voidaan pitää useita kommunikointiprotokollia, joista esimerkkinä ISO:n OSI-malli [McClain 91] [Haikala & Märijärvi 98]. OSI-mallin "protokollapinon" jokainen kerros abstrahoi sisäänsä jonkin toiminnallisuuden. Matalan tason kerrokset tarjoavat tyypillisesti verkkokorttien ja muun laitteiston tarjoamia palveluita yläpuolella oleville kerroksille, jolloin protokollapinon korkeamman tason kerrosten ei ole tarpeellista tietää alla olevista fyysisistä verkkoratkaisuista, esim. verkkokaapeleiden nollia ja ykkösiä kuvaavista jännitevaihteluista [Garlan & Shaw 96].

Kerroksittaisilla järjestelmillä voidaan havaita kolme pääasiallista hyvää ominaisuutta: 1) Kerrostamisen avulla saadaan aikaan järjestelmän abstraktiotason nouseminen. Jokainen kerros piilottaa sisäänsä kerroksen toteutukseen liittyvää yksityiskohtaista tietoa, jota järjestelmän kehittämisessä ei aina ole tarpeellista tietää. 2) Järjestelmää voidaan muunnella helpommin kerrostamisen avulla. Jokaisella kerroksella on määritelty rajapinta, jota voidaan käyttää uusien järjestelmien ja palveluiden rakentamisessa. 3) Uudelleenkäytön mahdollisuus paranee. Samoin kuin abstraktit tietotyypit, kerrokset antavat mahdollisuuden vaihtaa jonkin tietyn kerroksen (tai sen osan) toteutuksen toiseksi.

Ainoana edellytyksenä muuntamiselle tai koko kerroksen toteutuksen vaihtamiselle on, että sovitut rajapinnat eivät muutu. Sovittujen rajapintojen avulla voidaan rakentaa uusia palveluita, jotka käyttävät järjestelmän muodostavia eri kerroksia [Garlan & Shaw 96].

Kerroksittaisen mallin huonoina puolina voidaan pitää mm. joidenkin järjestelmien loogisesti vaikeaa mieltämistä kerroksittaiseksi järjestelmäksi sekä kerrostamisen mahdollisesti aiheuttamaa liian heikkoa suorituskykyä. Joissain tapauksissa pääseminen käsiksi ydintason operaatioihin saattaa olla välttämätöntä riittävän suorituskyvyn aikaansaamiseksi [Sommerville 95]. Lisäksi kerrosten oikeiden (tai sopivien) abstraktiotasojen määritteleminen voi osoittautua vaikeaksi [Garlan & Shaw 96].

4.3.3. Liitutaulumenetelmä

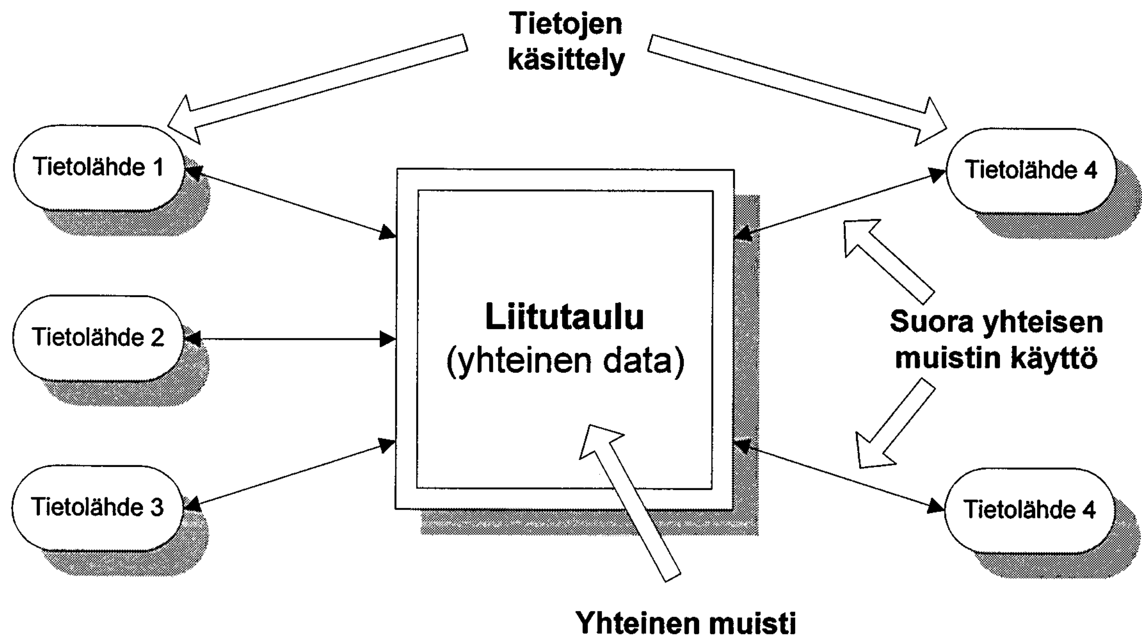
Alijärjestelmät, joiden täytyy toimiakseen vaihtaa useasti informaatiota keskenään, voivat perustua kahteen eri tyyppiseen ratkaisumalliin. Vaihtoehdot ovat:

a) Jokainen alijärjestelmä tallentaa tiedot omaan käyttöönsä
b) Tieto tallennetaan keskitetysti yhteen paikkaan, josta alijärjestelmät käyttävät sitä yhteisesti

Jälkimmäistä, yhteiseen tiedontallennukseen perustuvaa ratkaisua käyttävää vaihtoehtoa kutsutaan yleisesti liitutaulumenetelmäksi (*blackboard method* tai *repository model*) [Sommerville 95][Garlan & Shaw 96].

Liitutaulumenetelmä koostuu kahdenlaisista komponenteista. Keskeisin komponentti on yhteinen tietovarasto (liitutaulu), joka kuvaa järjestelmän nykyistä tilaa. Liitutaulukomponentin lisäksi menetelmään kuuluu joukko tietojenkäsittelyä suorittavia komponentteja. Tietojen siirtomenetelmän toteutus liitutaulun ja tietoja käsittelevien komponenttien välillä voi vaihdella järjestelmästä riippuen; jos järjestelmän toiminnan laukaisee tietojen sijoittaminen jaettuun muistiin, yhteinen muisti voidaan toteuttaa perinteisenä tietokantana. Jos taas järjestelmän toiminnan laukaiseminen perustuu jaetun da-

tan eri tiloihin, tieto voidaan varastoida liitutaulua muistuttavaan kaikille komponenteille yhteiseen rakenteeseen, esimerkiksi jaettuun muistialueeseen [Garlan & Shaw 96].



Kuva 12: Liitutaulumenetelmä [Garlan & Shaw 96]

Liitutaulumallin esitetään yleensä koostuvan kolmesta eri osasta: itsenäisistä tietolähteistä, jotka kommunikoivat keskenään ainoastaan yhteisen datan kautta, ongelman ratkaisuun käytettävästä liitutaulun tietorakenteesta sekä kontrollimallista, joka kuvaa miten järjestelmän ajonaikainen suoritus etenee. Suorituksen eteneminen riippuu täysin liitutaulun tilasta, ja tietojenkäsittelyn suorittavat komponentit eli tietolähteet vastaavat vain liitutaulun tilassa tapahtuneisiin muutoksiin. [Garlan & Shaw 96].

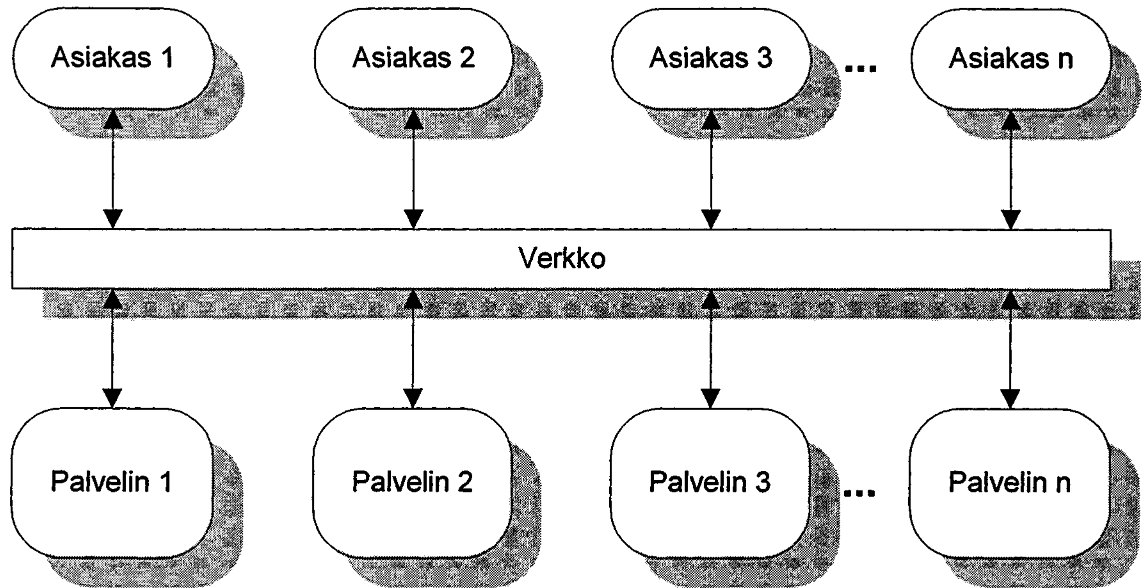
Mikäli järjestelmän toiminnan kannalta on tarpeellista jakaa suuri määrä tietoa eri tietojenkäsittelyä suoritettavien komponenttien välillä, liitutaulumenetelmää pidetään hyvänä lähestymistapa. Toisena menetelmän hyvänä puolena pidetään sitä, että tietoja ei myöskään tarvitse säilyttää useassa paikassa. Tämän seurauksena tietojen suojaus ja varmuuskopiointi voidaan järjestää toteutettavaksi kohtuullisen helposti – varsinkin mikäli järjestelmän yhteisiä tietoja säilytetään perinteisessä tietokannassa. Järjestelmän tietojenkäsittelyä suorittavien alijärjestelmien ei myöskään tarvitse tietää muiden alijärjestelmien toiminnasta.

Menetelmän huonoina puolina voidaan pitää sitä, että kaikkien alijärjestelmien on tiedettävä tietojen täsmällinen tallennusformaatti yhteisessä liitutaulussa. Alijärjestelmät saattaisivat myöskin haluta asettaa tiedolle eri suojaustasoja, mutta yhteisen muistialueen käyttö pakottaa järjestelmän vain yhden yhteisen suojaustason käyttöön. Lisäksi yhteisen tietovaraston jakaminen usean eri koneen välille on vaikeaa tietojen redundanssin ja epävakauden ongelmien takia [Sommerville 95].

Liitutauluja käytetään tyypillisesti monimutkaisissa signaaliprosessointiin liittyvissä sovelluksissa, kuten puheen- ja hahmontunnistuksessa [Nii 86] [Garlan & Shaw 96] [Sommerville 95].

4.3.4. Asiakas-palvelin -arkkitehtuuri

Asiakas-palvelin -arkkitehtuuri kuvaa järjestelmän tietojen ja niiden käsittelyn rakenteellisen hajauttamisen eri prosessoreille ja mahdollisesti eri koneille. Järjestelmän toimintaa kuvaava malli koostuu palvelimista, niiden tarjoamia palveluita käyttävistä asiakasohjelmista sekä niitä yhdistävästä verkosta (kuva 13). Teoriassa verkkoa ei pidetä pakollisena, koska palvelimet voivat fyysisesti toimia samalla koneella ja prosessorilla kuin asiakasohjelmatkin [Sommerville 95].



Kuva 13: Asiakas-palvelin -arkkitehtuuri. Muunnettu [Sommerville 95]:stä

Asiakas-palvelin -arkkitehtuurin ongelmana pidetään sitä, että asiakasohjelmien on tiedettävä tarkalleen käytössä olevien palvelimien verkko-osoitteet sekä palvelut, joita palvelin tarjoaa. Toisaalta palvelimen ei ole välttämätöntä tietää asiakasohjelmien lukumäärää tai niiden fyysistä verkkosijaintia [Sommerville 95].

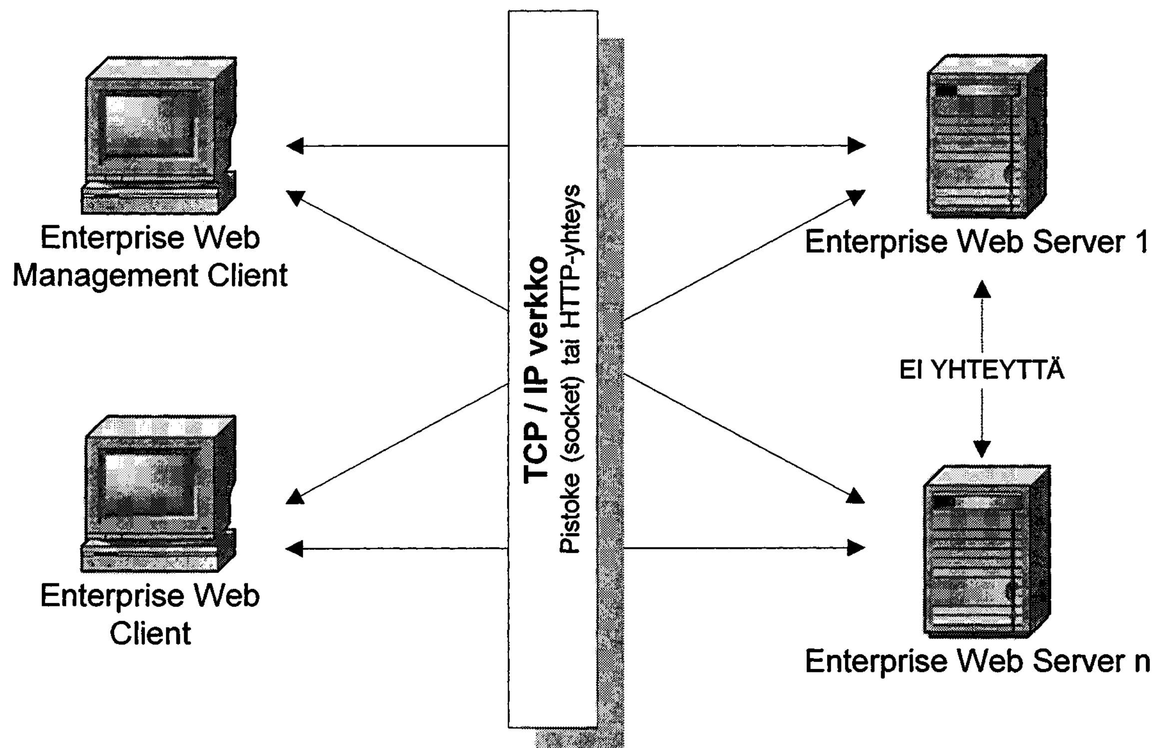
Teoriassa asiakas-palvelin -arkkitehtuureita käyttävien järjestelmien suoritustehoa voidaan lisätä yksinkertaisesti kasvattamalla tarjolla olevien palvelinten määrää. Käytännössä palvelinten lisäämistä vaikeuttaa kuitenkin palvelinten välisen yhteisen tietomallin puuttuminen. Jaetun tietomallin puute voi johtaa tietokantojen tietoylimäärän ja tiedon eheyden ongelmiin. Tietojen varmuuskopiointi ja tietosuojaukseen liittyvät ongelmat on ratkaistava siis palvelinkohtaisesti (vrt. edellä kuvatun liitutaulumenetelmän hyvät ja huonot puolet) [Sommerville 95].

Käsitellään asiakas-palvelin -arkkitehtuurin hyviä ja huonoja puolia vielä järjestelmäesimerkin avulla. InfoManager Enterprise Web-tuotteen (Eweb) hajautusmalli käyttää hyväkseen asiakas-palvelin -arkkitehtuuria (kuva 14). Järjestelmä muodostuu kahdesta erilaisesta asiakasohjelmasta (Eweb Client ja Eweb Management Client), palvelimesta (Eweb Server) sekä niitä yhdistävästä TCP/IP-verkosta.

Asiakasohjelman tarkoituksenmukainen käyttö vaatii täsmällisen tiedon Eweb Serverin TCP/IP verkko-osoitteesta. Toisaalta Eweb Server (järjestelmän palvelin) pitää asiakkaan fyysistä verkkosijaintia tai asiakkaiden lukumäärää järjestelmän toiminnan kannalta merkityksettömänä. Eweb Server ja Client-ohjelmia voidaan ajaa myös samalla koneella ja prosessorilla ilman fyysistä verkkoyhteyttä.

Jos järjestelmän palvelinohjelmaan lisätään tuotteen jatkokehityksessä uusia palveluita, myös asiakasohjelmia joudutaan päivittämään uusien palveluiden hyödyntämiseksi. Asiakasohjelmat eivät siis automaattisesti tiedä, mitä palveluita järjestelmän palvelin tarjoaa.

Järjestelmän suorituskykyä voidaan teoriassa nostaa lisäämällä Eweb-palvelinten määrää. Palvelimet eivät kuitenkaan toistaiseksi jaa yhteistä tietomallia – yhden palvelimen tietoja ei voida jakaa tarjottavaksi toisen palvelimen kautta. Palvelimet eivät välitä tietoja toistensa kanssa, ja toistaiseksi jokaisen palvelimen on itse huolehdittava tietojen varmuuskopioinnista ja suojauksesta.



Kuva 14: Asiakkaan ja palvelimen välinen kommunikointi. Muunnettu [InfoManager 99]:stä

Arkkitehtonisten tyylien määritelmien osuudessa todettiin, että järjestelmää voi pyrkiä analysoimaan sen kehittämisessä käytettyjen tyylien perusteella (kappale 4.4. [Garlan ym. 94]). Edellisen esimerkin valossa voimme olettaa tämän pitävän paikkansa. Enterprise Web -tuotteesta on havaittavissa jokainen Sommervillen [1995] luettelema asiakas-palvelin -arkkitehtuurin sekä hyvä että huono puoli. Järjestelmän kehityksessä käytettyjen arkkitehtonisten tyylien analysoinnilla voidaan siis arvioida järjestelmän tulevia laadullisia heikkouksia ja vahvuuksia jo ennen järjestelmän varsinaista kehittämistä.

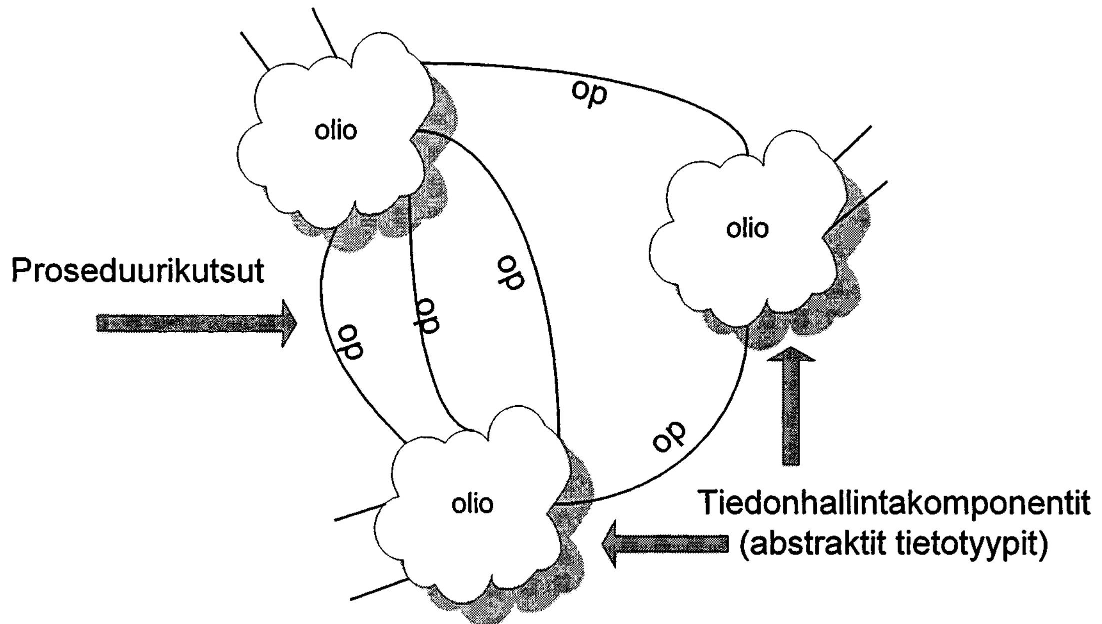
Enterprise Web -järjestelmää voitaisiin kuvata myös järjestelmänä, joka on toteutettu käyttäen seuraavaksi käsiteltävää oliopohjaista tyyliä käyttäen.

4.3.5. Tietoabstraktiot ja oliopohjaiset järjestelmät

Oliomenetelmien ja tietoabstraktioiden käyttöä pidetään myös järjestelmän rakenteellisen arkkitehtuurin muodostamisen tyylinä. Olioita (tai tietoabstraktioita) pidetään tiedon hallintaan perustuvina komponentteina, koska niiden keskeinen ajatus on säilyttää oma identiteettinsä ja tilansa [Garlan & Shaw 96].

Tyylin keskeisinä piirteinä voidaan pitää sekä olion vastuuta oman tilansa ylläpitämisestä että tiedon esityksen piilottamisesta muilta olioilta. Menetelmän hyvänä puolena mainitaan rajapintojen avulla järjestetty toteutuksen piilottamisen avulla saatava mahdollisuus moduulin (tai olion) sisäisen toteutuksen muuttamiseen, ilman että muita luokkia tarvitsee muuttaa [Garlan & Shaw 94].

Oliomenetelmän huonoina puolina voidaan pitää olioiden tarvetta tietää sekä toistensa olemassaolo että täsmällinen rajapintametodi vuorovaikutuksen mahdollistamiseksi. Vastaesimerkkinä voidaan käyttää aikaisempaa putki ja suodatin -menetelmää, jossa yksittäisen suodattimen ei tarvitse tietää mitään ympäröivästä maailmasta – suodatin toimittaa tulosteensa mitään kyselemättä sille osoitettuun putkeen [Garlan & Shaw 96].

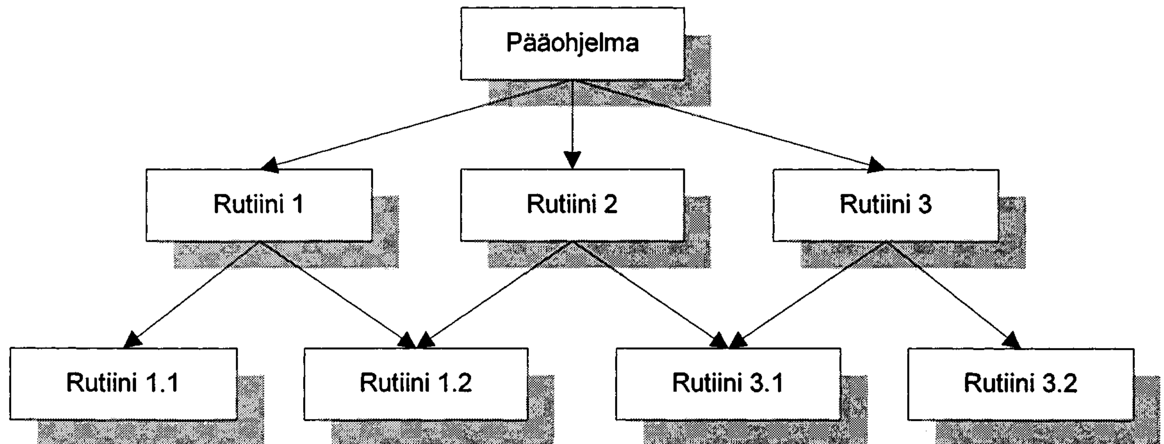


Kuva 15: Abstraktit tietotyypit ja oliot. Yksinkertaistettu [Garlan & Shaw 96]:sta

4.3.6. Keskitetyn ohjauksen mallit

Edellä esitetyt arkkitehtonisten tyylien esimerkit ovat olleet järjestelmän rakenteellisen jaottelun malleja. Keskitetyn ohjauksen malli on ensimmäinen esimerkki, joka käsittelee järjestelmän suoritusta ohjaavia malleja. Ohjausmalleilla tarkoitetaan niitä eri tyylejä, joilla järjestelmä valitsee keskinäisten suoritusvaiheiden järjestyksen (*control flow management*).

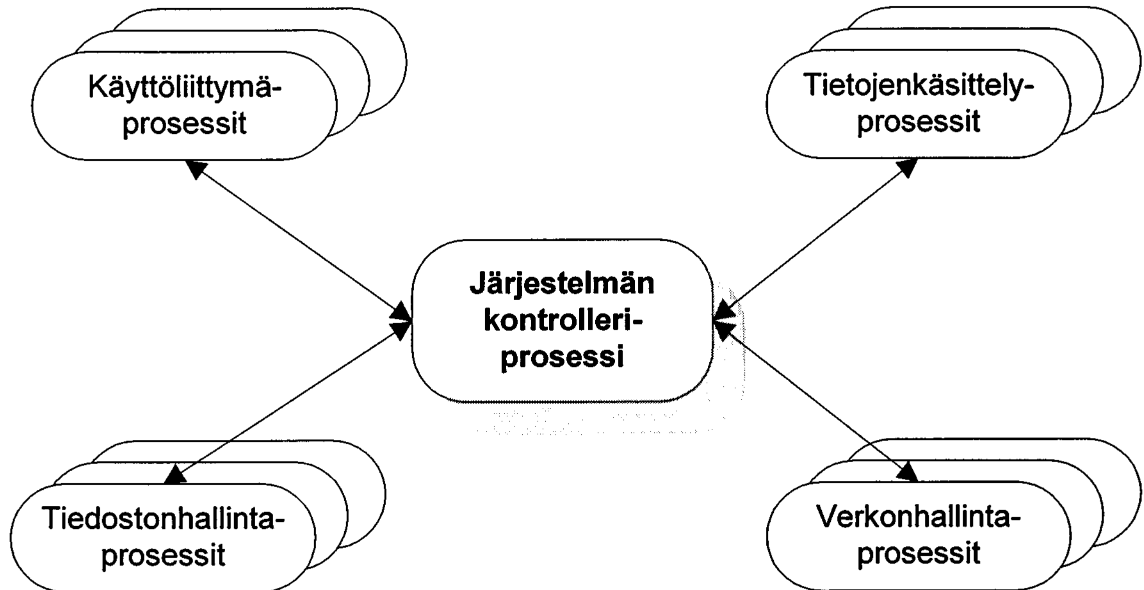
Keskitetyn ohjauksen mallissa yksi alijärjestelmä (tai komponentti) on vastuussa järjestelmän sisäisen suorituksen ohjauksesta. Keskitetyn ohjauksen malleja on Sommervillen [1995] mukaan kaksi: kutsu ja paluu -malli, joka soveltuu yksisäikeisten ohjelmistojen ohjausmalliksi, sekä kontrollerimalli (*manager model*), jota voidaan käyttää monisäikeisten ohjelmistojen suorituksen ohjaukseen [Sommerville 95].



Kuva 16: Kutsu-paluu malli järjestelmän kontrollirakenteena [Sommerville 95]

Kutsu ja paluu-malli (*call-return model*) soveltuu yksisäikeisten, peräkkäiseen suoritukseen perustuvien järjestelmien ohjausmalliksi (*sequential systems*). Peräkkäisellä mallilla tarkoitetaan esimerkiksi rakenteisen ohjelmoinnin järjestelmän abstraktiotason ylhäältä alaspäin tarkentuvaa jaottelua, jossa toiminta alkaa järjestelmän moduulijaon korkeimmalta abstraktiotasolta ja etenee hierarakkiassa alaspäin kohti yksityiskohtaisempaa tasoa. Toiminnan suorittamisen jälkeen kontrolli palaa matalan tason moduulilta takaisin päätason moduulille, esimerkiksi C-kielen `return`-lauseiden kautta.

Toinen keskitetyn ohjauksen malli – kontrollerimalli – soveltuu monisäikeisten reaaliaikaisten järjestelmien suorituksen ohjaamiseen. Monisäikeisissä järjestelmissä komponenttien välinen toiminta voi tapahtua (ainakin näennäisesti) samanaikaisesti. Kontrollerimallissa yksi järjestelmän komponentti on määritelty säikeidenvälisten suoritusvuorojen eksplisiittiseksi jakajaksi. Kontrollerikomponentti on vastuussa järjestelmän käynnistämisestä, pysäyttämistä sekä järjestelmän prosessien vuorottelemisesta ja priorisoinnista.



Kuva 17: Reaaliaikaisten järjestelmien keskitetyn kontrollin malli. Muunnettu [Sommerville 95]:stä

Esimerkiksi Windows NT -käyttöjärjestelmä on toteutettu loogisesti yhtäaikaiseen prosessien suoritukseen perustuen. Käyttöjärjestelmätasoinen kontrolleriprosessi hoitaa prosessien perustamista, tuhoamista sekä sisäistä säikeiden prioriteetteihin perustuvaa vuoronjakoa. Käyttöliittymänä toimii Taskmanager-ohjelma, jonka avulla voi seurata käynnissä olevien prosessien muistinkulutusta sekä tarpeen vaatiessa käskeä käyttöjärjestelmää lopettamaan jonkun tietyn prosessin suorituksen.

Keskitetyn ohjauksen mallien yleispiirteenä voidaan Sommervillen [1995] mukaan pitää sitä, että järjestelmän suoritusta ohjaavat päätökset tehdään käyttäen apuna joitakin järjestelmän sisäisiä tilamuuttujia (prosessien prioriteetit, samanarvoisten prosessien tasapuolinen vuorottelu, kutsu- ja paluumallin kooditasolla määrätty suoritusjärjestys).

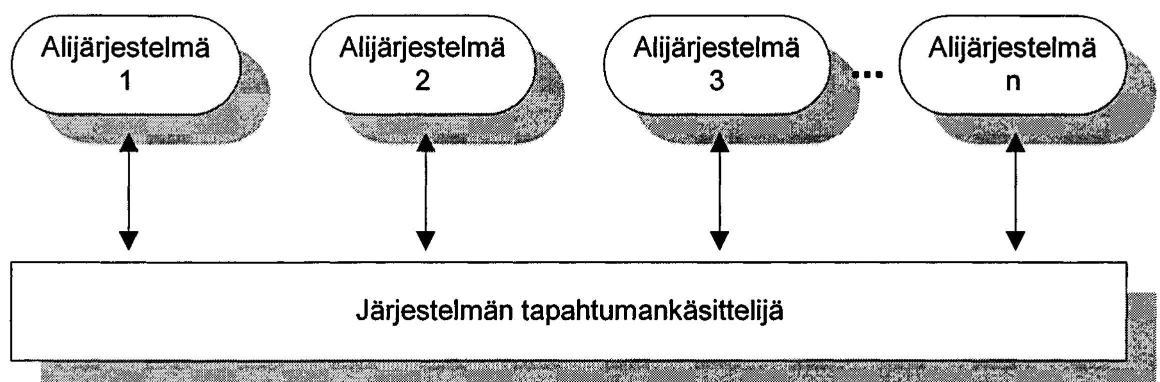
Seuraavaksi käsiteltävä tapahtumapohjainen järjestelmän ohjausmalli antaa hieman toisenlaista näkökulmaa järjestelmän suorituksen ohjaamisen mahdollisuuksiin. Mallia voidaan pitää eräänlaisena vastakohtana tässä kappaleessa esitetyille keskitetyn ohjauksen malleille.

4.3.7. Tapahtumapohjaiset järjestelmät

Komponenteista ja niiden määritellyistä rajapinnoista koostuvat järjestelmät ovat tyypillisesti toimineet suorien rajapintakutsujen avulla. Palvelua tarvitseva komponentti kutsuu suoraan palvelun tarjoavan moduulin rajapintametodia (esim. oliopohjaiset järjestelmät, kappale 4.3.5.).

Garlanin ja Shaw'n [1994] mukaan viime aikoina on yleistynyt järjestelmän ohjausmalli, jossa suoran rajapintakutsun sijaan puhutaan implisiittisestä kutsumisesta (*implicit invocation*), reaktiivisesta yhdistämisestä (*reactive integration*), tai valikoivasta lähettämisestä (*selective broadcast*). Tyyliä kutsutaan yleensä nimellä tapahtumapohjainen järjestelmä (*event based system*).

Komponentti voi suoran metodikutsun sijaan lähettää tapahtuman (*event*) järjestelmän tapahtumankäsittelijälle. Lähetetyn tapahtuman ottavat vastaan kaikki ne komponentit jotka ovat rekisteröityneet tapahtumatyypin kuuntelijaksi (*listener*). Tapahtumatyypin kuuntelijaksi rekisteröidytään kertomalla järjestelmän tapahtumankäsittelijälle, minkä tyyppisiä tapahtumia halutaan vastaanottaa. Esimerkkinä edellisestä on Javan tapahtumankäsittelymalli [Sun 99]. Tapahtuman lähettäminen kutsuu siis muita komponentteja implisiittisesti järjestelmän tapahtumanvälitysmekanismin kautta [Garlan & Shaw 94].



Kuva 18: Valikoivaan tapahtumien lähettämiseen perustuva kontrollimalli [Sommerville 95]

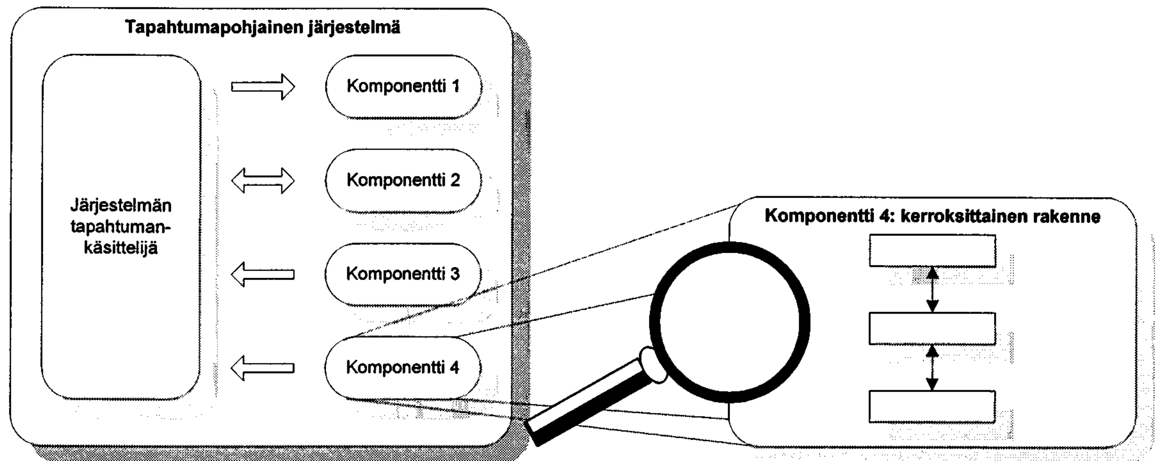
Tapahtumapohjaisen ohjausmallin huonona puolena voidaan pitää tietynlaista järjestelmällisyyden katoamista – järjestelmän koodista ja suunnittelumalleista on vaikea havaita toisaalta, mikä komponentti kuuntelee mitäkin tapahtumatyyppiä, ja toisaalta, mitä tapahtumia mikäkin komponentti lähettää. Tapahtumapohjaisen komponentin toiminnan seuraaminen debuggerilla on vaikeaa – komponentti saa herätteensä jostain näennäisen satunnaisesta osoitteesta, ilman että kutsuja on tiedossa.

Tyylin hyvänä puolena voidaan pitää sitä, että jos tapahtuma todella koskee useampaa kuin yhtä komponenttia, tapahtumankäsittelijä ilmoittaa tapahtuman kaikille siitä kiinnostuneille komponenteille. Lisäksi menetelmän käytön avulla voidaan vähentää komponenttien välisiä kytkentöjä (*coupling*), koska komponentit ovat yhteydessä toisiinsa vain tapahtumanvälitysmekanismin välityksellä.

4.3.8. Heterogeeniset tyylit

Toistaiseksi käsitellyt arkkitehtoniset tyylit ovat olleet ns. "puhtaita" tyylejä [Garlan & Shaw 96]. Vaikka järjestelmissä käytettyjen puhtaiden tyylien ymmärtäminen on tärkeää, harva järjestelmä todellisuudessa muodostetaan käyttämällä pelkästään yhtä tyyliä; järjestelmät ovat yleensä eri arkkitehtonisten tyylien yhdistelmiä [Garlan & Shaw 96] [Bass ym. 98].

Tyylejä voi yhdistellä usealla eri tavalla. Tyypillisimmillään tyylien yhdistäminen tapahtuu käyttämällä tyylejä hierarkkisesti. Järjestelmän muodostavat komponentit kommunikoivat keskenään esim. tapahtumapohjaista mallia käyttämällä, mutta komponentin sisäisessä toteutuksessa käytetään jotain toista mallia, esim. kerroksittaista rakennetta (kuva 19). Jopa komponenttien välisissä liittymissä voidaan käyttää arkkitehtonisia tyylejä. Esim. putki ja suodatin -menetelmän liittymät eli putket voidaan sisäisesti toteuttaa FIFO-jonona. [Garlan & Shaw 96].



Kuva 19: Heterogeeniset tyylit: tyylien hierarkkia. Muunneltu [Bass ym. 98]:sta

Tyylejä voi yhdistää myös sallimalla useiden tyylien esiintymisen saman komponentin toteutuksessa. Komponentti voi käyttää liitutaulumenetelmää tietojen varastointiin, mutta toteuttaa yhteydenpidon toisiin komponentteihin esim. putkien avulla [Garlan & Shaw 96].

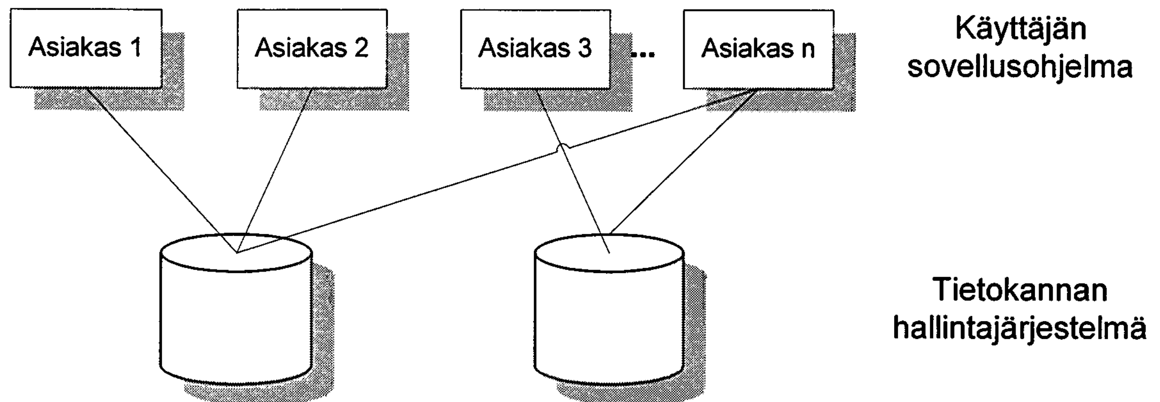
Kolmas tapa heterogeenisen arkkitehtuurin muodostamiseen on tyylien päällekkäinen käyttö [Garlan & Shaw 96]. Seuraavaksi esitettävän kolmitasoarkkitehtuurin mielenkiintoinen piirre on, että sen perusteella voidaan olettaa, että *tyylien päällekkäisen käytön yhteydessä muodostettuun malliin periytyy muodostamisessa käytettyjen mallien sekä hyvät että huonot puolet.*

4.3.9. Kolmitasoarkkitehtuuri esimerkkinä heterogeenisesta tyylistä

Tämä kappale tutkii tyylien päällekkäisen käytön yhteydessä muodostuvan heterogeenisen arkkitehtuurin ominaisuuksia. Voidaanko muodostuneessa arkkitehtuurissa havaita muodostuksessa käytettyjen alkuperäisten “puhtaiden” tyylien [Garland & Shaw 96] piirteitä?

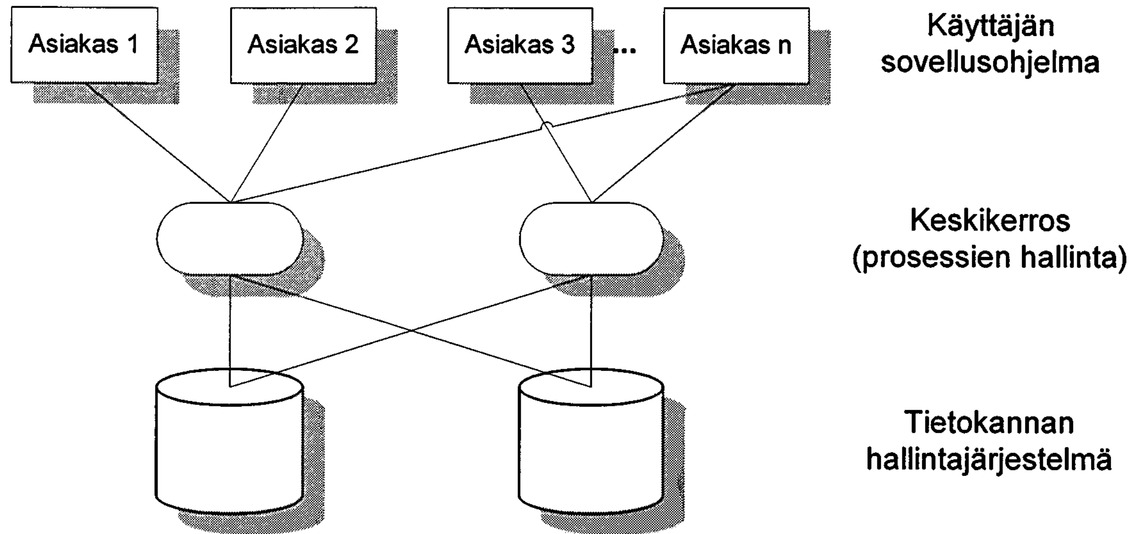
Kolmitasoarkkitehtuuria voidaan pitää esimerkkinä heterogeenisen tyylin muodostumisesta puhtaiden tyylien päällekkäisen käytön avulla. Tyylissä näyttäisi olevan havaittavissa piirteitä sekä asiakas-palvelin -arkkitehtuurista että kerroksittaisista järjestelmistä.

Tietokantoja käyttävät tietojärjestelmät oli ennen 90-lukua tyypillisesti rakennettu niin, että sovellusohjelma käytti suoraan tietokantaa sen hallintajärjestelmän kautta. Tätä suoraviivaista menetelmää kutsutaan kaksitasoarkkitehtuuriksi – asiakasohjelmat käyttävät dataa suoraan tietokannasta. Kun yhtäaikaisten tietokannan käyttäjien määrä myöhemmin kasvoi, ongelmaksi muodostui tietokannan hallintajärjestelmän suorituskyky – hallintajärjestelmää ei oltu tarkoitettu mahdollisesti useiden satojen yhtäaikaisten käyttäjien palvelemiseen [STR CS Architecture 99].



Kuva 20: Kaksitasoarkkitehtuuri

Kolmitasoarkkitehtuuri kehitettiin ratkaisemaan kaksitasoarkkitehtuurin suorituskykyongelmat. Kolmas kerros – keskikerros (*middle tier*) – lisättiin käyttäjän sovellusohjelman ja tietokannan hallintajärjestelmän välille. Keskikerroksen tehtävä oli hallita tietokantojen käytöön liittyviä prosesseja: säikeiden luominen, tehtävänanto, säikeiden monitorointi ja resursointi, tietojen lukitus sekä tietojen yhtenäisyyden ja replikoinnin varmistaminen [STR Three Tier 99].



Kuva 21: Kolmitasoarkkitehtuuri [STR Three Tier 99]

Kokonaisjärjestelmästä muodostui kerrostettu rakenne, jossa keskikerroksen tehtäväksi tuli abstrahoida tietokannan hallintaan ja kyselyiden tekemiseen liittyvät, käyttöliittymän kannalta merkityksettömät tehtävät.

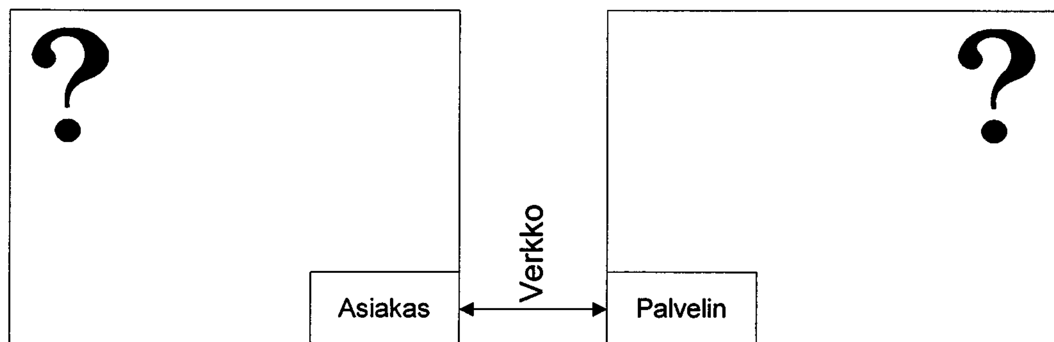
Muodostunut rakenne paransi myös uudelleenkäyttöä – jos keskikerroksessa määritelty rajapinta pysyy muuttumattomana, alla olevaa fyysistä tallennusmuotoa voidaan muuttaa asiakasohjelmien toiminnan siitä häiriintymättä. Keskikerroksen avulla voitiin rakentaa myös uusia asiakasohjelmia hyväksikäyttämällä määriteltyä rajapintaa – tietokantojen fyysisiä tietorakenteita ei ollut enää tarpeellista muuttaa jokaisen uuden asiakasohjelman takia. Kolmitasoarkkitehtuurin huonona puolena voidaan pitää kerroksittaisen järjestelmän mahdollista suorituskykyongelmaa.

Kolmitasoarkkitehtuurin keskikerroksen luonteessa on havaittavissa selkeitä asiakaspalvelin -arkkitehtuurin piirteitä. Keskikerros toimii palveluiden tarjoajana joukolle asiakasohjelmia. Keskikerroksen palvelinten toimintaan ei vaikuta asiakasohjelmien lukumäärä tai niiden fyysinen sijainti. Toisaalta asiakasohjelmien on tiedettävä, mitä palvelimia (keskikerroksen komponentteja) on olemassa ja mitä palveluita ne tarjoavat.

Voidaan olettaa, että muodostettuun heterogeeniseen tyyliin on siis todella *periytynyt* muodostuksessa käytettyjen puhtaiden arkkitehtonisten tyylien sekä hyvät että huonot puolet.

4.4. Järjestelmän kuvaaminen arkkitehtonisten tyylien avulla

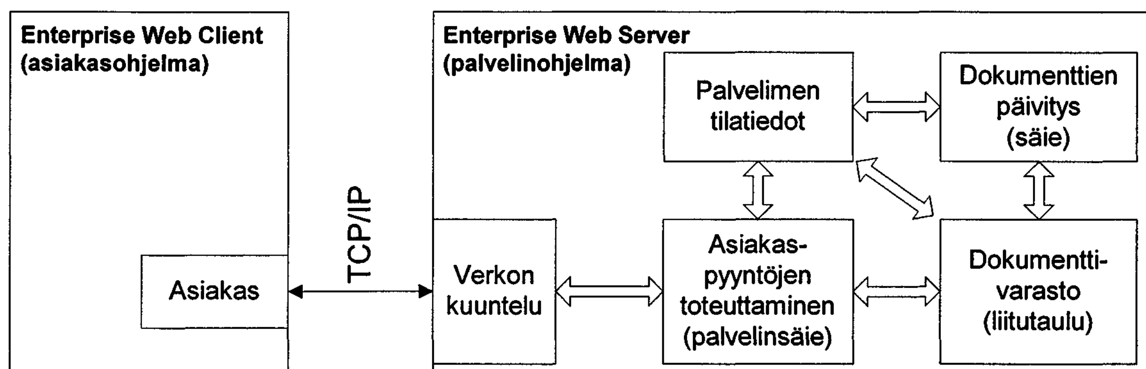
Arkkitehtonisia tyylejä voidaan käyttää järjestelmän eri piirteiden kuvaamiseen. Järjestelmän arkkitehtuurin kuvaamisella esim. asiakas-palvelin -arkkitehtuuriksi ei kuitenkaan saada kerrotuksi vielä kovinkaan paljoa varsinaisesta kohdejärjestelmästä (kuva 22).



Kuva 22: "Järjestelmä on toteutettu asiakas-palvelin -arkkitehtuurin avulla"

Mitä itse asiassa tiedämme järjestelmän arkkitehtuurista asiakas-palvelin kuvauksen jälkeen? Järjestelmä on luultavasti (mutta ei välttämättä) hajautettu usean koneen tai prosessorin välille. Järjestelmässä on ainakin yksi palvelin ja yksi asiakasohjelma, jotka ovat yhteydessä toisiinsa verkon välityksellä. Kuvaus ei vielä esim. kerro mitään kummankaan pään (asiakas tai palvelin) rakenteesta, ohjausmallista tai toiminnasta.

Vaikka kuvausta voidaankin pitää selkeästi puutteellisena, se ei kuitenkaan tee kuvausta hyödyttömäksi; järjestelmän arkkitehtuurin luonnehdinta asiakas-palvelin -arkkitehtuurina kuvaa järjestelmän yhden keskeisen piirteen. Tarvitaan kuitenkin vielä lisää samankaltaisia arkkitehtuurin luonnehdintoja, ennen kuin järjestelmän kokonaisrakenne on ymmärrettävissä.



Kuva 23: InfoManager Enterprise Web Serverin rakenteellinen arkkitehtuuri (epätäydellinen)

Kuvassa 23 on (epätäydellisesti) kuvattu InfoManager Oy Enterprise Web (Eweb) Serverin sisäinen arkkitehtuuri. Kuten kaaviosta huomataan, asiakas-palvelin -arkkitehtuuri edustaa vain murto-osaa järjestelmän arkkitehtuurista; Asiakas-palvelin -arkkitehtuuria palvelimen arkkitehtuurissa edustavat vain verkonkuuntelija-komponentti ja palvelinsäie. Palvelimen arkkitehtuuria kuvaavasta kaaviosta voidaan havaita myös liitutaulumenetelmän sovellus. Dokumenttivarasto-komponentti toimii alijärjestelmien (palvelinsäie, dokumenttien päivitys, palvelimen tilatiedot) jaettuna muistialueena.

Kuvan 23 arkkitehtuurin kuvaus ei vieläkään ole riittävä kuvaamaan koko järjestelmän toimintaa: kaavio ei vieläkään kerro mitään palvelimen suoritusta ohjaavasta rakenteesta – miten suoritus etenee eri komponenttien välillä? Onko järjestelmä tapahtumapohjainen vai kenties keskitetyn kontrollin mukainen järjestelmä? Jos siirryttäisiin vielä eteenpäin ja tarkasteltaisiin Eweb Serverin asiakaspyyntöjä toteuttavan palvelusäikeen sisäistä arkkitehtuuria, voitaisiin huomata että kyseisessä komponentissa tietoliikenneprotokolla on toteutettu käyttäen kerroksittaista tyyliä.

Järjestelmän arkkitehtuurin toteutuksessa voidaan sisäisesti käyttää useita hierarkkisia tai päällekkäisiä arkkitehtonisia tyylejä. Vaikka järjestelmän arkkitehtuurillisiin tyyleihin perustuvalla arkkitehtuurin luonnehdinnalla ei voida kuvata koko järjestelmän arkkitehtuurin muodostumista, tyyli kuvaa kuitenkin jonkin keskeisen piirteen järjestelmän toteutuksesta – luonnehdinta ei siis ole turha. Järjestelmän arkkitehtuurin kuvaus vaatii

useamman kuin yhden mallin käyttämistä. Arkkitehtonisten tyylien tärkein ominaisuus on kuitenkin niiden mahdollinen uudelleenkäyttö järjestelmien arkkitehtuureiden muodostamisessa.

4.5. Arkkitehtonisten tyylien yhteenveto

Kun tietyn ongelman hyväksi havaittu ratkaisumalli tulee yleiseen tietoisuuteen siitä muodostuu epämuodollinen "kansantaru". Näistä kansantaruista muodostuu teorioiden tieteellisen muodollistamisen kautta arkkitehtonisia tyylejä. Tyylit vaikuttavat alalla vallitseviin käytäntöihin ja sitä kautta edelleen uusien tyylien muodostumiseen.

Arkkitehtoniset tyylit jaotellaan yleensä neljään eri ryhmään: oliopohjaisiin arkkitehtuureihin, tilapohjaisiin arkkitehtuureihin, palaute- ja kontrollipohjaisiin arkkitehtuureihin sekä arkkitehtuureihin jotka korostavat järjestelmän reaaliaikaisia ominaisuuksia.

Heterogeenisilla tyyleillä tarkoitetaan arkkitehtuureita, jotka sisältävät piirteitä useammista kuin yhdestä "puhtaasta" arkkitehtuurista. Tutkielmassa esitetty kolmitasoarkkitehtuuri pitää sisällään piirteitä sekä asiakas-palvelin -arkkitehtuurista että kerroksittaisista järjestelmistä.

Arkkitehtonisilla tyyleillä ja niiden valinnoilla järjestelmän arkkitehtuurin muodostamisessa on siis edellisten esimerkkien valossa kauaskantoisia vaikutuksia. Yhdeksi arkkitehtuurisuunnittelun keskeiseksi ongelmaksi nouseekin: mikä arkkitehtoninen tyyli pitäisi valita? Tyylien valinnan ongelmaa käsitellään osana ohjelmiston arkkitehtuurin muodostamisen ongelmaa seuraavassa, arkkitehtuurisuunnittelua käsittelevässä luvussa.

5. ARKKITEHTUURISUUNNITTELU

Aikaisemmissa luvuissa on käsitelty järjestelmän arkkitehtuurin suhteita ympäröivään todellisuuteen: sidosryhmiin, järjestelmän vaatimuksiin, organisaatioon, eri kehitysvaiheisiin ja suunnittelumenetelmiin, sekä on todettu, että arkkitehtuuri vaikuttaa koko ohjelmiston elinkaaren ajan. Arkkitehtuuria on tutkittu sen kehittämisprosessin kautta.

Tässä luvussa käsitellään varsinaisia arkkitehtuurin muodostamisen menetelmiä ja ongelmia – miten arkkitehti muodostaa arkkitehtuurin järjestelmälle osoitettujen vaatimusten ja laadullisten tavoitteiden perusteella ja käyttää ongelmanratkaisussa mahdollisesti apunaan hyväksi havaittuja arkkitehtonisia tyylejä.

5.1. Arkkitehtuurisuunnittelun tavoitteita

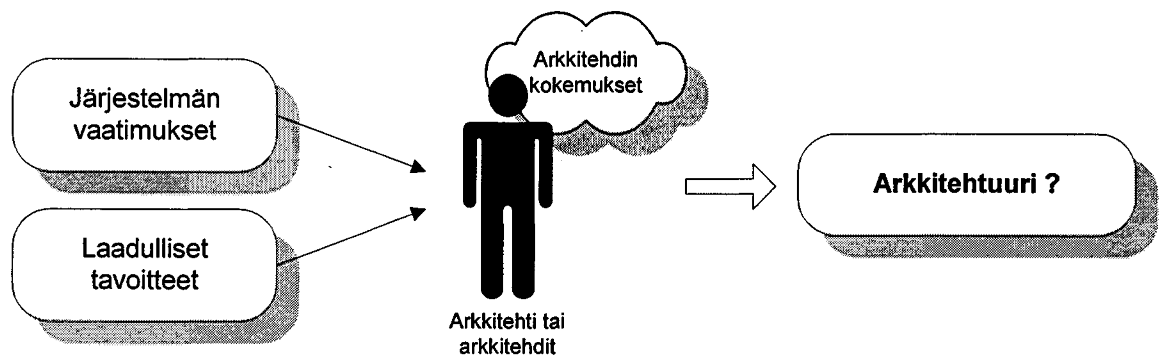
Jokaisella järjestelmällä on arkkitehtuuri, joka on heijastuma järjestelmälle asetetuista vaatimuksista, laadullisista tavoitteista sekä niistä kompromisseistä, joita on jouduttu tekemään edellä mainittujen saavuttamiseksi [Bass ym. 98] [Medvidovic 99]. Kompromisseillä tarkoitetaan sitä, että mitään järjestelmän yksittäistä laatuattribuuttia ei voida maksimoida, ilman että jostain toisesta ominaisuudesta jouduttaisiin tinkimään (*architectural trade-offs*). Laatuattribuutteja ovat mm.

- Järjestelmän suorituskyky
- Yhteensopivuus vanhojen järjestelmien kanssa
- Suunniteltu uudelleenkäyttö
- Ohjelmiston fyysinen jakeluprofiili
- Turvallisuus, käyttövarmuus, vikojensietokyky
- Järjestelmän jatkokehitettävyys ja muunneltavuus

Keskeiseksi kysymykseksi nousee: "Kuinka saadaan aikaan arkkitehtuuri, joka toteuttaa annetut toiminnalliset ja laadulliset vaatimukset?" [Medvidovic 99]

5.2. Mistä arkkitehtuurit tulevat?

Mitään yleisesti hyväksyttyä prosessimallia – tapaa miten arkkitehtuuri muodostetaan – ei ole olemassa [Sommerville 95] [Bass ym. 98]. Sommervillen [1995] mukaan arkkitehtuurisuunnittelussa joudutaan kuitenkin yleensä tekemään seuraavat tehtävät: järjestelmän jakaminen alijärjestelmiksi ja niiden välisen kommunikoinnin määritteleminen, järjestelmän kontrollimallin määritteleminen sekä järjestelmän jakaminen moduuleiksi.



Kuva 24: Arkkitehtuuriin vaikuttavat tekijät [Bass ym. 98]

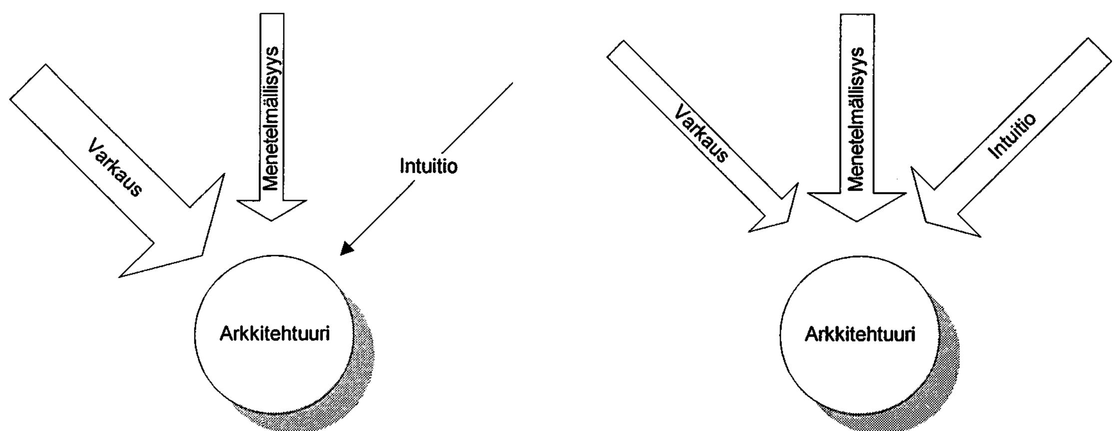
Arkkitehtuurin muodostaminen riippuu viime kädessä järjestelmälle asetetetuista vaatimuksista sekä laadullisista tavoitteista ja arkkitehdin (tai arkkitehtien) kokemuksesta ratkaista käsillä oleva ongelma. Arkkitehti voi käyttää mitä tahansa menetelmää muodostaessaan arkkitehtuuria – tarkoitus pyhittää keinot. Arkkitehtuuri on seuraus joukosta teknisiä, liiketoiminnallisia ja sosiaalisia syitä [Bass ym. 98].

Tehtävää vaikeuttaa osaltaan ohjelmiston arkkitehtuurin eksplisiittisen määritelmän puute (ks. luvun 3 kappale 1). Mikä arkkitehtonisen suunnittelun lopputulos itseasiassa on? Kaikilla arkkitehtuurin määritelmillä on ainakin yksi yhteinen tekijä; ohjelmiston arkkitehtuuri on kuvaus järjestelmän muodostavista komponenteista sekä niiden välisistä liittymistä. Arkkitehtuurin suunnittelun ensimmäinen tavoite onkin yleensä saada aikaan tämä järjestelmän rakenteellinen jaottelu. Miten jaotteluun päädytään, riippuu täysin arkkitehdin kokemusmaailmasta: ”varasta, käytä menetelmää tai luota intuiti-

oon." [Medvidovic 99] [Booch 99] Jos arkkitehti on kokematon, rakenne on pakko muodostaa menetelmällisesti omaan intuitioon luottaen (innovatiivinen ratkaisutapa). Jos arkkitehdillä on kokemusta järjestelmien kehittämisestä tai sovellusalueesta yleensä, rakenteellinen jako kannattaa muodostaa rutiininomaisesti varastamalla – ts. käyttäen hyväksi arkkitehtonisia tyylejä tai sopivia sovelluskehyksiä (application frameworks).

5.2.1. Varastamalla – menetelmällä – intuitiolla

Arkkitehtuurit muodostetaan joko varastamalla, menetelmällisyyden avulla tai arkkitehdin intuition avulla. Edellisten käytön suhde riippuu arkkitehdin kokemuksesta sekä itse rakennettavan järjestelmän hienostuneisuudesta [Medvidovic 99].



Kuva 25: Eri menetelmiä arkkitehtuurin saavuttamisessa [Medvidovic 99]

Arkkitehtuurin varkaudella tarkoitetaan arkkitehtonisen ratkaisun ottamista joko aikaisemmin rakennetusta järjestelmästä, alan kirjallisuudesta tai järjestelmään sopivasta sovelluskehyksestä (sovelluskehyksien käytöstä arkkitehtuurisuunnittelussa tarkemmin kappaleessa 5.2.3.). Menetelmällisyydellä tarkoitetaan järjestelmällisten ja tietoisten menetelmien (mahdollisesti heuristista) käyttöä arkkitehtuurin johtamisessa erilaisten siirtymien kautta järjestelmän vaatimuksista. Intuitiolla tarkoitetaan arkkitehdin kykyä päätyä toimiviin ratkaisuihin ilman varsinaista vaihtoehtojen syvällistä pohdintaa. Edellä mainittuja keinoja käyttäen päädytään joko rutiininomaisiin tai innovatiivisiin ratkaisuihin. [Medvidovic 99].

5.2.2. Rutiiniratkaisu vai innovaatio?

Rutiiniratkaisuiden käyttämiseen arkkitehtuurin muodostamiseksi liittyy tuttujen ongelmien ratkaiseminen käyttämällä uudelleen suuria osia aikaisemmista ratkaisuista [Garlan & Shaw 96]. Jos järjestelmän arkkitehdillä on hyviä kokemuksia tietyn arkkitehtonisen tyylin aikaisemmasta soveltamisesta käsillä olevaan ongelmaan, on hyvin todennäköistä, että arkkitehti yrittää käyttää samaa tyyliä uudelleen ongelman ratkaisuun – ja päin vastoin [Bass ym. 98]. Rutiiniratkaisujen käytössä menetelmällisyys on tärkeää: arkkitehtuuri, joka on rakennettu 50% varastamalla ja 50% innovaation avulla on Medvidovicin mukaan tuomittu epäonnistumaan [Medvidovic 99]. Arkkitehtuurin "varastaminen" aikaisemmista ratkaisuista on halvempaa kuin innovatiivisten ratkaisujen käyttö, mutta lähestymistavan ongelmana on mahdollinen liiallinen uudelleenkäyttö. Ratkaisu todennäköisesti ei myöskään ole yhtä optimaalinen, kuin innovatiivisella menetelmällä saataisiin tuotetuksi [Medvidovic 99].

Innovatiivisella ratkaisulla tarkoitetaan uusien hienostuneiden ratkaisujen löytämistä ongelmiin, joita arkkitehti ei ole aikaisemmin joutunut ratkaisemaan [Garlan & Shaw 96]. Innovatiiviset ratkaisut voidaan saada aikaan yksinkertaisesti keksimällä uusia tapoja, käyttämällä intuitiota tai pyrkimällä johtamaan uusia ratkaisuja joukosta abstrakteja periaatteita [Medvidovic 99]. Innovatiivisen ratkaisumallin käyttäminen tuottaa todennäköisesti optimaalisemman ratkaisun kuin rutiinimenetelmä, mutta innovatiivisten menetelmien käyttö on pääsääntöisesti kalliimpaa kuin rutiiniratkaisuiden. Potentiaalisena ongelmana innovaation käytössä on vaara myös "keksiä pyörä uudelleen" [Medvidovic 99].

5.2.3. Sovelluskehykset

Arkkitehtuurisuunnittelussa voidaan käyttää apuna erilaisia sovelluskehyksiä. Sovelluskehysten käyttöä arkkitehtuurisuunnittelun apuna voidaan pitää eräänlaisena tunnetun ongelman rutiininomaisena ratkaisutapana.

Sovelluskehys muodostuu Johnsonin [Johnson 97a] mukaan joukosta järjestelmän (tai sen osan) kuvaavia abstrakteja luokkia, sekä tavasta, miten näiden luokkien esiintymät ovat keskenään vuorovaikutuksessa. Toisen yleisen määritelmän mukaan sovelluskehystä pidetään järjestelmän "luurankona" (skeleton application), josta sovelluskehittäjä voi sopeuttamalla muodostaa uuden järjestelmän [Fayad & Schmidt 97]. Sovelluskehykset ovat siis eräänlaisia oliotekniikan avulla toteutettuja uudelleenkäytettäviä "puolivalmiita" ohjelmistoja tai niiden osia, joita voidaan sopeuttaa uusien sovellusten tuottamiseksi. Mitään yleisesti hyväksyttyä eksplisiittistä määritelmää sovelluskehyksille ei kuitenkaan ole olemassa [Johnson 97a].

Sovelluskehysten läheinen suhde arkkitehtuureihin voidaan havaita ensimmäisestä määritelmästä: arkkitehtuuri muodostuu komponenteista ja niiden välisistä vuorovaikutuksista, sovelluskehys luokista ja niiden esiintymien välisistä vuorovaikutuksista.

Sovelluskehykset on tyypillisesti rakennettu johonkin yksittäiseen tarkoitukseen kuten esim. käyttöliittymäohjelmointiin tai puhelinkeskuksen puhelunohjausta varten. Eräs yleisesti tunnettu sovelluskehys on Microsoftin MFC (Microsoft Foundation Classes), joka on tarkoitettu lähinnä liiketoimintaan liittyvien Windows -ohjelmien graafisten käyttöliittymien rakentamiseen [Fayad & Schmidt 97]. MFC sisältää sekä käyttöliittymäluokkia että suosituksia kuinka näitä luokkia tulisi yhdistää. Sovelluskehykset sisältävätkin komponenttien lisäksi yleensä myös ohjeet kuinka näiden luokkien avulla voidaan rakentaa tietyntyyppisiä sovelluksia [Johnson 97b].

Sovelluskehykset voidaan jakaa karkeasti kolmeen eri tyyppiin [Fayad & Schmidt 97]:

1. Käyttöjärjestelmien kehitykseen tarkoitetut infrastruktuuritason sovelluskehykset.
2. Hajautettujen järjestelmien muodostamisen apuna käytetyt sovelluskehykset (*middleware* – esim. CORBA ja ORB, Javan RMI, Microsoftin DCOM).
3. Rajatun sovellusalueen sovelluskehykset (*enterprise application frameworks* – esim. telekommunikaatio-, pankki- ja vakuutussovelluskehykset jne.).

Arkkitehtuurisuunnittelun näkökulmasta mielenkiintoisimpina voidaan pitää sekä hajautettujen järjestelmien muodostamiseen tarkoitettuja sovelluskehyksiä että tietyn sovellusalueen sovelluskehyksiä. Arkkitehtuurisuunnittelussa ja järjestelmän moduulijaossa voidaan käyttää apuna kehyksen määrittelemiä liiketoimintakomponentteja. Toisaalta järjestelmän hajauttamisessa (eli komponenttien välisten liittymien määrittelyssä) voidaan käyttää apuna jotain (tai joitakin) edellä mainituista hajauttamiseen tarkoitetuista sovelluskehyksistä.

Sovelluskehysten käytön huonona puolena voidaan pitää kehysten käytön vaikeutta. Tyypillisesti esim. MFC-kirjaston käytön tehokas hyödyntäminen vaatii noin puolen vuoden opiskelua [Fayad & Schmidt 97]. Lisäksi debuggerin avulla tapahtuva virheenjäljittäminen voi olla vaikeaa – virheitä on vaikea (tai jopa mahdotonta) jäljittää kehykseen kuuluvien valmiiden luokkien sisältä.

Sovelluskehykset ja arkkitehtuurisuunnittelu liittyvät läheisesti mm. kappaleessa 5.4.4. lyhyesti käsiteltyihin sovellusalueen viitearkkitehtuureihin, joissa tietyn ohjelmaperheen kaikki ohjelmat jakavat saman viitearkkitehtuurin.

5.3. Järjestelmän rakenteellisen arkkitehtuurin muodostaminen

Medvidovicin [1999] mukaan järjestelmän rakenteen muodostamisen ongelma on kaksijakoinen:

- Miten jakaa ohjelma kokonaisuuden muodostaviin osiin (decomposition)
- Miten muodostaa ohjelma mahdollisesti olemassa olevista osista (composition)

Vaihtoehtoisiksi ratkaisumalleiksi sopii kaksi lähestymistapaa: voidaan lähestyä ongelmaa joko ylhäältä alas (*top-down*) tai alhaalta ylös (*bottom-up*) [Medvidovic 99].

Ylhäältä alas -lähestymistavassa järjestelmä pyritään osittamaan ongelman ratkaiseviksi alijärjestelmiksi. Esimerkkinä voidaan pitää rakenteisen analyysin (*SA - Structured*

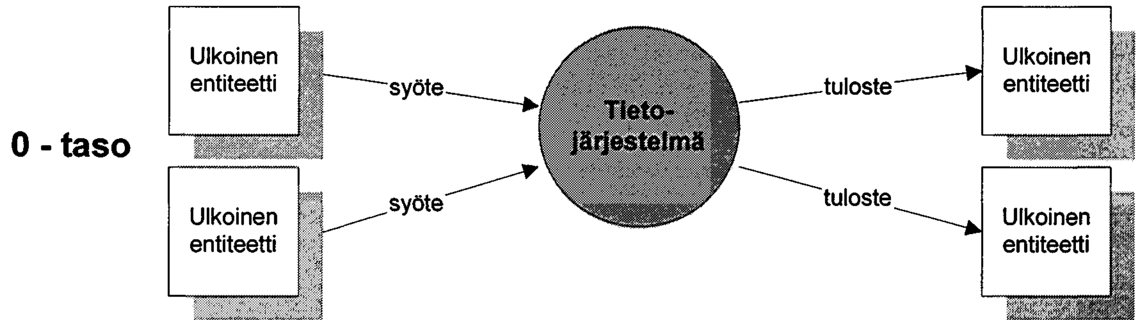
analysis) yhteydessä käytettyä järjestelmän tietovirtakaavion (*DFD – Dataflow Diagram*) muodostamista [Yourdon 89]. Kokonaisjärjestelmä muodostuu jakamisen jälkeen osaongelmat ratkaisevista komponenteista ja niiden välisistä liittymistä [Pressman 97][Medvidovic 99].

Alhaalta ylös -lähestymistavassa järjestelmä pyritään kokoamaan (mahdollisesti olemassaolevista) komponenteista, jotka yhdessä toteuttavat järjestelmälle asetetut toiminnalliset ja laadulliset tavoitteet. Esimerkkinä voidaan pitää komponenttipohjaisia ohjelmistotuotannon lähestymistapoja [Boehm & Scherlis 92] [Gacek 97] [Kuusi 99]. Järjestelmän kokoamista komponenteista vaikeuttaa, mistä löytää tarvittavat komponentit, mitä tarjolla olevat komponentit yleensäkään tekevät ja mistä tiedämme, että juuri näiden komponenttien käytöllä saadaan aikaan vaadittu järjestelmä [Medvidovic 99][Krueger 92].

Todellisuudessa arkkitehtuurin muodostamisen realistinen lähestymistapa vaatii sekä ylhäältä alas että alhaalta ylös -lähestymistapojen käyttöä [Medvidovic 99]. Em. lähestymistapojen lisäksi tässä kappaleessa esitetään arkkitehtuurin muodostaminen oliomenetelmän avulla, jota voidaan pitää jonkinlaisena edellisten vaihtoehtojen välimuotona.

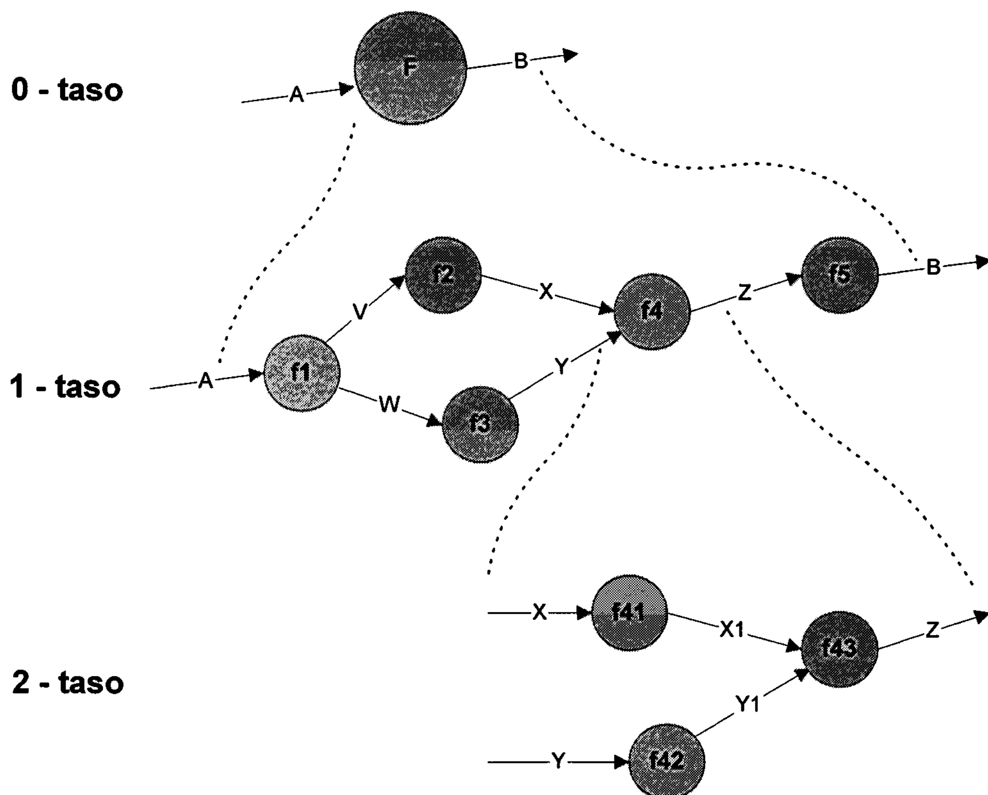
5.3.1. Ylhäältä alas: rakenteisen analyysin tietovirtakaaviot

Rakenteisessa analyysissä rakennettava järjestelmä pyritään mallintamaan ylhäältä alaspäin. Suunnittelu alkaa ns. nollatasolta, jossa koko järjestelmä on mallinnettu yhtenä prosessina, jolla on määrätyt syötteet ja joista järjestelmä muodostaa tietyt tulosteet. Järjestelmä nähdään siis joukkona syötteitä, joista on tarkoitus erilaisten muunnosten kautta saada aikaan halutunlaiset tulosteet (kuva 26).



Kuva 26: DFD nollataso: syötteet, järjestelmä, tulosteet [Pressman 97]

Menetelmän mukaisesti jokainen prosessi jaetaan aliprosesseihin, kunnes osittamiselle ei ole enää tarvetta – lopputuloksena saavutetaan toteutettavaksi soveltuva tarkkuustaso. Kuva 27 havainnollistaa ylhäältä alaspäin lähestyvää asteittaista tarkentamista.



Kuva 27: Ylhäältä alaspäin tarkentuva tietovirtakaavio. Muunneltu [Pressman 97]:sta.

5.3.2. Alhaalta ylös: komponenttipohjainen ohjelmistotuotanto

Komponenttipohjaista ohjelmistotuotantoa (*CBD – component based development*) voidaan pitää järjestelmän arkkitehtuurin muodostamista alhaalta ylöspäin lähestyvänä menetelmänä. CBD:llä tarkoitetaan ohjelmistojen tuottamista sopeuttamalla ja yhdistämällä komponentteja. Komponentit voivat olla joko edellisten projektien tuotoksia, ne voidaan ostaa markkinoilta (*COTS – components off the shelf*), tai ne voidaan tehdä itse [Kuusi 99].

CBD:n yhteydessä on hyvä muistaa, että menetelmä tyypillisesti muokkaa itse järjestelmälle asetettuja vaatimuksia ja laadullisia tavoitteita [Kuusi99] [Sommerville 95]. Tällä perusteella Gacek ym:n [1995] määritelmä ohjelmistoarkkitehtuurista, johon kiinteästi kuuluu järjestelmän vaatimukset ja tavoitteet, on hyvin ymmärrettävissä – komponenttien uudelleenkäytöllä voidaan saavuttaa merkittäviä säästöjä ohjelmiston elinkaaren aikana, joten niiden käyttöä jopa suorastaan vaaditaan järjestelmän rakentamisessa [Sommerville 95] [Kuusi 99] [Bass ym. 98].

Viime vuosikymmen on ollut komponenttipohjaisen ohjelmistotuotannon suuren nousun aikaa. Miksi järjestelmiä kuitenkin edelleen rakennetaan alusta asti itse, kun ohjelmien koostaminen valmiista komponenteista olisi niin paljon halvempaa, nopeampaa ja tehokkaampaa? Uudelleenkäytön yleisten ongelmien lisäksi [Krueger 92] vaikeuksia aiheuttaa komponenttien arkkitehtoninen yhteensopimattomuus (*architectural mismatch*). Arkkitehtonisella yhteensopimattomuudella tarkoitetaan niitä ongelmia, jotka johtuvat uudellenkäytettäväksi aiotun komponentin valmistajan oletuksista järjestelmää kohtaan, jossa komponenttia aiotaan käyttää [Garlan ym. 95] [Gacek 97].

Komponentteja tuottavat yritykset julistavat tuotteidensa yhteensopivuutta, koska komponentti noudattaa jotakin tiettyä standardia. Esim. "CORBA:n avulla kaikki komponenttien koostamisen yhteensopivuuden ongelmat saadaan katoamaan." Jos järjestelmää kuvataan arkkitehtoniselta tyyliltään CORBA-pohjaiseksi järjestelmäksi, itse asiassa tiedetään vain, miten komponentti kommunikoi toisten komponenttien kanssa (vertaa luku 4, kappale 4, järjestelmän kuvaaminen arkkitehtonisten tyylien avulla). Kompo-

nenttipohjaisen kehitysmenetelmän ongelmana ei kuitenkaan ole pelkästään komponenttien välinen kommunikointitapa, vaan ongelmat johtuvat muistakin komponenttivalmistajien tekemistä kohdejärjestelmää koskevista arkkitehtonisista oletuksista.

Arkkitehtonisen yhteensopimattomuuden ongelma voi ilmetä mm. järjestelmän luvattoman suurena koodikokona, huonona suorituskykynä tai tarpeena muuntaa järjestelmän muita komponentteja vastaamaan lisättävän uuden komponentin tarpeita. Lisäksi voidaan joutua tekemään jo järjestelmässä olemassa olevia toimintoja uudella tavalla komponentin olettaessa tietyntyyppistä rajapintaa tai tietojen tallennusrakennetta järjestelmän muissa komponenteissa. Järjestelmän kääntäminen saattaa myös monimutkaistua ja hidastua. Tämän seurauksena käännösprosessi saattaa olla altis virheille [Garlan ym. 95]. Uudelleenkäytettävä komponentti joudutaan *sopeuttamaan* kohdejärjestelmään. Sopeuttamisella tarkoitetaan niitä toimenpiteitä, joita komponentille joudutaan tekemään, jotta sitä voitaisiin käyttää osana järjestelmää [Kuusi 99].

Garlan ym. [1995] esittävät neljän kategorian taksonomisen kehyksen, jonka avulla voidaan ymmärtää syitä, joista arkkitehtoniset yhteensopimattomuuden ongelmat johtuvat:

1. Oletukset komponenttien luonteesta
2. Oletukset komponenttien välisten liittymien luonteesta
3. Oletukset järjestelmän arkkitehtuurin kokonaisrakenteesta
4. Oletukset komponenttien ajonaikaisesta käytöstä

Komponenttien luonteella tarkoitetaan oletuksia komponentin vaatimasta ja sen mukana toimitettavasta infrastruktuurista, komponentin sisäisestä suorituksen ajoittamisesta (komponentin kontrollimallista) ja siitä, miten komponentin ulkopuoliset järjestelmän osat käsittelevät komponentin sisältämiä tietoja (tietomalli) [Garlan ym. 95].

Jotta uudelleenkäytettäväksi tarkoitettu komponentti voi toimia aiotussa ympäristössään, komponentin täytyy kapseloida sisäänsä kaikki mahdolliset apukirjastot, joita komponentin sisäinen toiminta saattaa vaatia. Komponentti ei siis voi ennakkoon olettaa mitään kohdejärjestelmästä; mitä palveluita on olemassa jne. Tämä infrastruktuurin on-

gelma on helppo ymmärtää vertaamalla sitä vaikkapa nykyisiin tekstinkäsittelyohjelmiin: vaikka käyttäjä ei esimerkiksi koskaan käyttäisi tekstinkäsittelyohjelmaan integroidun piirto-ohjelman ominaisuuksia, piirto-ohjelman apukirjastot toimitetaan joka tapauksessa automaattisesti ohjelman mukana. Vaikka komponentin toiminnasta käytettäisiin vain pientä osaa, kirjastot toimitetaan komponentin mukana. Seurauksena on järjestelmän koon sekä resurssien tarpeen kasvaminen.

Yksi vaikeimmista komponentin luonteen yhteensopimattomuuden ongelmista muodostuu kohdejärjestelmän kontrollimallin oletuksista. Konflikti voi syntyä, jos järjestelmässä on useita komponentteja, jotka olettavat saavansa haltuunsa järjestelmän koko prosessinhallinnan. Mikä komponentti lopulta ohjaakaan järjestelmän toimintaa? Valitettavan tyypillinen esimerkki on vanhojen 16-bittisten Windows-käyttöjärjestelmien tapahtumankäsittely. Käyttöjärjestelmässä on yksi tapahtumajono, jota kaikki ohjelmat (mukaanlukien käyttöjärjestelmä) käyttävät yhteisesti. Jos rakennettava järjestelmä koostuu useasta komponentista, joka olettaa saavansa haltuunsa koko tapahtumajonon hallinnan, konflikti on valmis.

Komponentin luonteeseen liittyvien oletusten ongelmista viimeinen liittyy oletuksiin tietomallien käytöstä. Komponentti voi olettaa datan olevan jossain tietyssä formaatissa, jota muun järjestelmän on tuettava, tai komponenttia ei voi käyttää.

Liittymien luonteella tarkoitetaan komponenttien välisten liittymien vuorovaikutuksen malleja, kommunikointiprotokollia sekä sitä, minkälaista tietoa liittymän kautta välitetään. Komponentti saattaa vaatia liittymän olevan tietyn tyyppinen tapahtumapohjainen yhteys, suora metodikutsu tai jokin muu spesifi menetelmä, jota muun järjestelmän on toteutettava komponentin käyttämiseksi. Myös välitettävällä tiedolla on tyypillisesti tarkka formaatti – joka harvoin sopii suoraan yhteen muun järjestelmän kanssa.

Järjestelmän arkkitehtuurin yhteensopimattomuudella tarkoitetaan sitä ongelmaa, joka muodostuu komponentin olettaessa tiettyjä asioita järjestelmän arkkitehtuurilta, esimerkiksi tiettyjen kommunikointitapojen tai komponenttien olemassaoloa tai puuttumista. Esim. tietokantakomponentti voi olettaa, että tietoja yrittää käyttää vain yksi kutsuja- tai

asiakaskomponentti kerrallaan. Jos järjestelmä on ennakko-oletuksia vastaan kuitenkin monisäikeinen, ongelmaksi muodostuu tietojen lukitusmekanismien (*synchronization*) puuttuminen.

Komponentin ajonaikaiseen käyttöön liittyvillä oletuksilla tarkoitetaan vaatimuksia, missä järjestyksessä komponenttien ilmentymät on luotava [Garlan ym. 95]. Esimerkiksi tietokannan SQL-kyselykomponentin luomisen edeltävyysvaatimuksena on varsinaisen tietokantakomponentin instanssin luominen jne.

Miten näiltä yhteensopimattomuuden ongelmilta voitaisiin välttyä? Garlanin ym. [1995] mukaan ratkaisuna on sekä rakentaa itse uudelleenkäytettäväksi tarkoitetut komponentit paremmin että parantaa niitä mekanismeja, notaatioita ja työkaluja, joiden avulla komponettien suunnittelijat voivat tämän tehdä. Garlan ym. [1995] ehdottavat neljää pitkän tähtäimen kehitysehdotusta:

1. Uudelleenkäyttöön tarkoitettujen komponenttien kehityksen yhteydessä pitäisi eksplisiittisesti määritellä ne oletukset, jotka on tehty kohdejärjestelmän arkkitehtuurista. Nykyisin ongelmana on, että komponentin kohdejärjestelmälle asettamia arkkitehtonisia vaatimuksia ei juuri koskaan ole dokumentoitu.
2. Suuret järjestelmät pitäisi koota pienistä alijärjestelmistä. Arkkitehtuurioletukset saataisiin näin ollen kapseloiduksi yhden alijärjestelmän sisäisiksi sen sijaan, että ne levittäytyvät läpi koko järjestelmän.
3. Yhteensopivuusongelmien ratkaisuun tähtäävien tekniikoiden tuottaminen. Yhteensopimattomuuden ongelmat ratkaistaan sopeuttamalla [Kuusi 99] komponentti kohdejärjestelmään. Sopeuttaminen tapahtuu raa'alla voimalla – kirjoittamalla uudelleen matalan tason tiedonsiirtoa hoitavaa koodia, tekemällä erityisiä tiedon esitystavan muunnoskomponentteja jne. Parempi lähestymistapa olisi muodostaa potentiaalisesti ongelmalliselle komponentille erilaisia kapselointi- tai ympäröintimekanismeja (*wrapper*) [Kuusi 99], joiden avulla komponenttia voitaisiin käyttää luontevammalla tavalla.
4. Arkkitehtuureiden suunnitteluoppaiden tuottaminen. Oppaiden pitäisi mm. kertoa, mitkä arkkitehtoniset ratkaisut toimivat yhdessä ja mitkä taas eivät. Arkkitehtuu-

rioppaiden tarpeeseen liittyy komponenttipohjaisen ohjelmistotuotannon periaatteiden sekä sääntöjen muodostaminen ja kirjaaminen.

Vaikka komponenttipohjaisessa ohjelmistotuotannossa onkin vielä paljon ratkaistavia ongelmia, sitä voidaan pitää kuitenkin yhtenä lupaavimmista lähestymistavoista järjestelmien tehokkaaseen rakentamiseen. Komponenttiajattelu tukee myös hyvin arkkitehtuurin muodostamisen periaatteita: järjestelmän arkkitehtuuri muodostuu sen komponenteista ja komponenttien välisistä liittymistä.

5.3.3. Arkkitehtuurisuunnittelu ja oliolähestymistapa

Oliomenetelmä pyrkii järjestelmän muodostamiseen ongelmakentän mallintamisen avulla. Ongelmakenttä pyritään mallintamaan tunnistamalla ongelmakentästä joukko vuorovaikutteisia entiteettejä eli olioita. Järjestelmän arkkitehtuuri muodostuu olioiden sekä niiden välisten liittymien ohjelmallisesta toteutuksesta [Korson & McGregor 90].

Mallintaminen voidaan aloittaa useasta eri lähtökohdasta. Yksi tapa on pyrkiä tunnistamaan ongelmakentän muodostavia konkreettisia tai abstrakteja olioita. Konkreettisilla olioilla tarkoitetaan olioita, joilla on oikean elämän vastine: käyttäjä, liikennevalo, auto tms. Abstraktioilla tarkoitetaan sellaisia olioita kuin roolit, vuorovaikutukset jne. Olioiden tunnistamisen yhteydessä pyritään mallintamaan niiden suhteet toisiin olioihin (*relations*) [Korson & McGregor 90], jotka muodostavat arkkitehtuurissa lopulta komponenttien väliset liittymät.

Toinen lähtökohta ongelmakentän mallintamiseksi on pyrkiä tunnistamaan suoraan järjestelmän tulevia toiminnallisia kokonaisuuksia. Timothy Buddin [1997] mukaan olioiden tunnistaminen kannattaa aloittaa järjestelmän käyttäytymisen analyysista, koska järjestelmän käyttäytyminen on helpoin ymmärtää verrattuna mihinkään järjestelmän muuhun aspektiin.

Medvidovic [1999] luettelee lyhyesti oliomenetelmän järjestelmän suunnittelun keskeiset vaiheet:

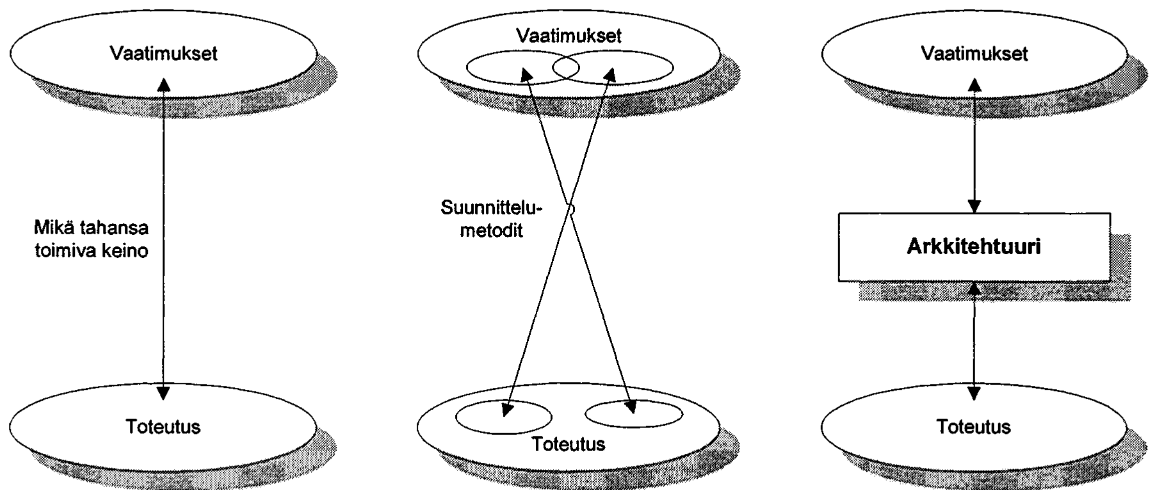
1. Tunnista ongelmakentän muodostavat oliot.
2. Määrittele jokaisen olion tila, käytös ja rajapinnat.
3. Tunnista olioiden yhteiset ominaisuudet (perintää varten).
4. Etsi ja uudelleenkäytä olemassaolevat luokat tai tee uudet luokat joihin oliot kuuluvat.
5. Muodosta olioista toimiva sovellus. Muodostamisen malli saadaan ongelmakentän muodostavien olioiden riippuvuuksista toisiinsa.

Edellä mainituista lähestymistavoista Buddin [1997] ehdottama menetelmä soveltuu luultavasti paremmin järjestelmän arkkitehtuurin muodostamiseen, koska ongelman kuvauksesta pyritään suoraan tunnistamaan järjestelmän keskeiset toiminnot, joilla taas on kohtuullisen suora yhteys järjestelmän muodostaviin komponentteihin. Oliomenetelmä ei ainakaan näennäisesti ota kantaa järjestelmän laadullisiin vaatimuksiin – suorituskykyyn, ylläpidettävyyteen tai muunneltavuuteen. Oliomenetelmän käyttö ohjelman arkkitehtuurin muodostamisessa voi johtaa tilanteeseen, jossa järjestelmälle asetetuista vaatimuksista johdetaan suoraan järjestelmän toteutus, ilman että varsinaista arkkitehtuuria koskaan tietoisesti muodostettaisiin tai analysoitaisiin (kts. kuva 28).

Vaikka sekä arkkitehtuuri että suunnittelumenetelmät tähtäävät samaan lopputulokseen – toteutuksen saavuttamiseen järjestelmän vaatimusten avulla – ne tekevät sen eri tavalla [Garlan & Perry 95].

Jos ohjelmaa rakennetaan ilman suunnittelumenetelmien – kuten tässä kappaleessa mainitun oliomenetelmien – tai arkkitehtuurisuunnittelun käyttöä, ratkaisu muodostuu käyttämällä mitä tahansa ad hoc -menetelmää. Suunnittelumenetelmät määrittelevät periaatteessa kaikki ne välivaiheet, joiden kautta järjestelmän suunnittelija etenee järjestelmän vaatimuksista valmiiseen toteutukseen asti. Menetelmän toimivuus riippuu siitä, kuinka hyvin menetelmä sattuu sopimaan käsillä olevan ongelman ratkaisemiseen; oliomenetelmää käyttäen saadaan tyypillisesti aikaan järjestelmä, joka muodostuu oli-

oista ja niiden välisistä suhteista [Korson & McGregor 90] [Budd 97], kun taas esim. rakenteisen analyysin ja suunnittelun avulla muodostettu ratkaisu perustuu mm. tietovirtojen käyttöön [Pressman 97]. Arkkitehtuurit ja suunnittelumenetelmät ovat toisiaan täydentäviä menetelmiä: useiden suunnittelumenetelmien taustalta voidaan löytää suosittuja arkkitehtonisia tyylejä ja eri arkkitehtonisten tyylien käyttö voi taas johtaa uusiin kehitysmenetelmiin [Garlan & Perry 95].



Kuva 28: Suunnittelumenetelmien ja arkkitehtuureiden ero [Garlan & Perry 95]

Arkkitehtuurisuunnittelussa ei välttämättä oteta suoranaisesti kantaa esim. käytettyyn toteutuskieleen ja suunnittelumenetelmään, vaan ne määräytyvät vasta komponentin oman arkkitehtuurisuunnittelun jälkeen.

5.4. Tyylien käyttö arkkitehtuurisuunnittelussa

Ohjelmiston arkkitehtonisella tyylillä tarkoitetaan joksenkin samaa asiaa kuin rakennusarkkitehtuurissakin. Tyyli sisältää muutamia avainpiirteitä sekä sääntöjä, miten näitä avainpiirteitä voidaan yhdistellä, jotta arkkitehtoninen yhtenäisyys säilyy [Bass ym. 98].

Tämä kappale käsittelee arkkitehtonisten tyylien käyttöä järjestelmien arkkitehtuurisuunnittelun apuna. Pohjaa tyylien hyväksikäytölle muodostetaan tunnistamalla tyylien jaotteluperusteita sekä tutkimalla uudelleenkäyttöä yleensä järjestelmien kehityksessä.

Kappaleessa 5.4.3. tutkitaan tyylin valintamenetelmiä ja lopulta käsitellään sitä, kuinka tyylin (tai tyylien) valinnan jälkeen muodostetaan varsinainen ohjelmiston arkkitehtuuri. Lisäksi tutkitaan lyhyesti, miten arkkitehtuurisuunnittelun jälkeen siirrytään järjestelmän kehityksessä eteenpäin.

5.4.1. Arkkitehtonisten tyylien jaottelusta

Arkkitehtonisia tyylejä voi löytää järjestelmän useilta eri tarkastelutasoilta. Tämän kappaleen tarkoituksena on tyylien jaottelun avulla tunnistaa näitä eri tarkastelutasoja ja näin ollen luoda pohjaa tyylien uudelleenkäytön mahdollisuuksien ymmärtämiselle.

Arkkitehtoniset tyylit voidaan jaotella karkeasti kahteen kategoriaan [Garlan & Shaw 96]:

1. Idiomit ja ratkaisumallit (idioms ja patterns)
2. Viitemallit (reference models)

Idiomeilla (*idiom*) tarkoitetaan koko järjestelmän toimintaa koskevia organisointimalleja, kuten aiemmin esitettyä putki ja suodatin –menetelmää sekä kerroksittaista mallia [Garlan ym. 94]. Muita esimerkkejä koko järjestelmää koskevista malleista ovat asiakas-palvelin -arkkitehtuuri ja liitutaulumenetelmä.

Ratkaisumalleilla (*pattern*) tarkoitetaan ohjelman paikallisesti käyttämiä organisointimalleja (esim. *MVC eli Model-View-Controller,* joka määrittää käyttöliittymien ja tiedon välisen suhteen [Krasner & Pope 88]) sekä useita oliomenetelmiin perustuvia ratkaisumalleja (suunnittelumallit, [Gamma ym. 94] ja Coadin oliopohjaiset mallit [Coad 92]) [Garlan & Shaw 96].

Ratkaisumalleja löytyy eri tarkastelutasoilta. Tästä seurauksena arkkitehtoninen tyyli voi käsitteellisesti olla hyvinkin pieni (mikroarkkitehtuuri), tai toisaalta koko järjestelmän toimintaa koskeva uudelleenkäytettävissä oleva rakenne (makroarkkitehtuuri) [Ko-

gut & Clements 99]. Ratkaisumalleja ovat mm. kappaleessa 5.2.3. esitetyt sovelluskehykset, suunnittelumallit ja koodimallit.

Käsitellään esimerkinomaisesti hieman arkkitehtonisten tyylien ja suunnittelumallien suhdetta toisiinsa. *Suunnittelumallit* (*design patterns*) ovat hyvin lähellä arkkitehtonisia tyylejä. Suunnittelumallien keskeinen ajatus on löytää oliomalleista toistuvia rakenteita. Näiden rakenteiden muodollistamisen ja eksplisiittiseen muotoon muuntamisen avulla niitä voidaan uudelleenkäyttää samankaltaisten ongelmien ratkaisemiseksi. Aikaisemmin on kuvattu, kuinka arkkitehtoniset tyylit muodostuvat samalla tavalla kuin suunnittelumallit – ongelman hyväksi havaittu ratkaisu muodollistetaan ja käytetään uudelleen.

Eroa voi pyrkiä ymmärtämään ajattelemalla arkkitehtonista tyyliä kehyksenä tai viitteenä, jonka avulla arkkitehti voi kehittää käyttökelpoisia suunnittelumalleja tietyn tyyppisten ongelmien rutiiniratkaisemiseksi. Ero on hyvin samankaltainen kuin esimerkiksi OMT-oliomenetelmällä ja olioilla – OMT-menetelmä tarjoaa viitemallin olioiden suunnitteluun [Monroe ym. 97].

Konkreettisempia eroja on mm. suunnittelumallin sitoutuminen oliototeutusmenetelmään, kun arkkitehtuurit periaatteessa ovat toteutusmenetelmästä riippumattomia. Suunnittelumallit ja arkkitehtoniset tyylit eroavat ongelmien tarkastelun tasossa; arkkitehtuurit tarkastelevat järjestelmää kokonaisvaltaisesti, suunnittelumallit taas keskittyvät pienempien osaongelmin ratkaisemiseen.

Kaikille kehyksiä käyttäville malleille (arkkitehtoniset tyylit, sovelluskehykset, suunnittelumallit, koodimallit) on kuitenkin yhteistä se, että malli edustaa joukkoa suunnittelua koskevia päätöksiä, jotka voidaan uudelleenkäyttää kokonaisuutenaan – "pakettina". Eri mallien pääasiallinen ero on, milloin uudelleenkäyttö tapahtuu ja missä mittakaavassa. Arkkitehti käyttää järjestelmämalleja ensimmäisten suunnittelupäätösten tekemiseen, suunnittelumalleja taas sovelletaan pääasiassa järjestelmän arkkitehtuurin suunnittelun jälkeen ohjelman tai komponentin suunnittelun aikana [Bass ym. 98].

Viitemallit pyrkivät määrittelemään, miten tietyn sovellusalueen elementit ja niiden väliset liittymät tulee järjestää [Garlan & Shaw 96] [Bass ym. 98]. Kappaleessa 4.3.1. esitetty kääntäjän rakenne on esimerkki viitemallista: viitemalli määrittelee kääntäjän perusmallin muodostuvan leksikaalisen analyysin suorittavasta osuudesta, jäsentimestä sekä koodin tuottamisen suorittavasta elementistä. Komponenttien välillä ei saa myöskään olla taaksepäin ketjussa suuntautuvaa palautetta; komponentti antaa tuotoksensa vain sen jälkeen tulevalle komponentille. Toinen yleisesti tunnettu viitemallin esimerkki on ISO:n seitsemäntasoinen OSI-malli [McClain 91] [Garlan & Shaw 96].

5.4.2. Tyylien uudelleenkäytöstä

Uudelleenkäytöllä tarkoitetaan uusien ohjelmien rakentamista käyttäen apuna jo olemassa olevia ohjelmistoja tai niiden osia. Uudelleenkäyttö ei vielä kuitenkaan ole saavuttanut täyttä potentiaaliaan ohjelmistojen kehityksessä. Uudelleenkäytön keskeisenä menestystekijänä voidaan pitää ratkaisumallin onnistunutta abstrahointia [Krueger 92].

Charles Krueger määrittelee artikkelissaan Software Reuse [Krueger 92] uudelleenkäytettävän ohjelmistoarkkitehtuurin korkean tason ohjelmistokehykseksi (*software framework*) ja alijärjestelmiksi (*subsystem*), jotka sisältävät ohjelmiston kokonaisrakenteen ja toiminnallisuuden mallin. Ohjelmiston arkkitehtuurin uudelleenkäytöllä voidaan saada merkittäviä hyötyjä ohjelmiston suunnittelun ja toteutuksen aikana.

Arkkitehtonisten tyylien käyttö arkkitehtuurisuunnittelun apuna tehostaa uudelleenkäyttöä [Garlan ym. 94]. Muodostuneita rutiiniratkaisuja voidaan käyttää tulevaisuudessa samankaltaisten ongelmien turvalliseen ratkaisemiseen. Tyylien käyttö parantaa myös koodin uudelleenkäytettävyyttä. Esimerkiksi järjestelmät, jotka käyttävät putki ja suodatin -menetelmää, voivat usein hyödyntää samoja UNIXin käyttöjärjestelmäprimitiivejä tehtävien ajoittamisessa, synkronoinnissa sekä kommunikoinnissa putkien välityksellä. Samoin asiakas-palvelin -mallissa voidaan koodissa usein käyttää samoja RPC-kutsuja (*Remote Procedure Call*). Lisäksi järjestelmän toimintaa ja koostumusta on helpompi ymmärtää, jos se on muodostettu käyttäen yleisesti tunnettuja (uudelleenkäytet-

tyjä) rakenteita. Suunnitteluavaruuden rajaaminen tyylien avulla helpottaa järjestelmän analysointia. Järjestelmälle voidaan siis suorittaa tyyliin perustuvia analyyseja. Tyylien uudelleenkäytön avulla järjestelmää on lisäksi helpompi kuvata visuaalisesti tyylille tyypillisellä esitystavalla [Garlan ym. 94].

Uudelleenkäytön mahdollistamiseksi uudelleenkäytettävän rakenteen (*reuse artifact*) on täytettävä seuraavat ehdot [Krueger 92]:

1. Uudelleenkäytettävän rakenteen on vähennettävä kognitiivista eroa järjestelmän alkuperäisen käsitteen ja uudelleenkäyttöä hyväksikäyttäneen toteutuksen välillä. Kognitiivisella erolla tarkoitetaan suunnittelijan ajattelutyön määrää, joka vaaditaan järjestelmän saattamiseksi nykyisestä kehitysvaiheesta seuraavaan.
2. Uudelleenkäytettävien rakenteiden on oltava helppokäyttöisempiä kuin koko rakenteen kirjoittaminen uudelleen.
3. Uudelleenkäytön onnistumiseksi on tiedettävä, mitä mikäkin uudelleenkäytettävä rakenne tekee.
4. Uudelleenkäytettävän rakenteen täytyy olla löydettävissä nopeammin, kuin mitä sellaisen rakentaminen kestäisi.

Uudelleenkäytettävyys ei synny automaattisesti, vaan sen aikaansaamiseksi on panostettava resursseja. [Haikala & Märijärvi 98] [Sametinger 97] [Kuusi 99]. Laadukkaiden uudelleenkäyttöön tarkoitettujen arkkitehtuurikirjastojen muodostaminen on erittäin vaikeaa [Neighbors 89] [Kruger 92]. Ongelma johtuu toisaalta sopivien uudelleenkäytettävien arkkitehtuureiden tunnistamisesta ja rakentamisesta, ja toisaalta uudelleenkäytettävien rakenteiden riittävän tehokkaiden kirjastointimenetelmien kehittämisestä.

Aikaisemmin on todettu, että potentiaalisesti uudelleenkäytettäviä ratkaisumalleja voidaan löytää monilta ohjelmiston eri tarkastelutasoilta [Garlan & Shaw 96]. Uudelleenkäytettävän tyylin sovellusalue (domain) voi olla hyvinkin pieni, kuten joukko pinojen ja jonojen muodostamiseen liittyviä rakenteita, tai hyvin suuri, kuten navigointi, prosessiohjaus tai tietojen käsittely. Arkkitehtuureita voidaan näin ollen käyttää joko rekursiivisesti uusien korkeamman tason arkkitehtuureiden muodostamiseen, tai sellaisenaan

sovelluskehittimen (application generator) omaisesti loppukäyttäjän sovelluksen tuottamiseksi [Krueger 92].

Tyylien uudelleenkäytöllä näyttäisi olevan merkittäviä hyötyjä, mutta kuinka valita järjestelmälle oikea tyyli, jonka avulla hyödyt olisivat saavutettavissa? Ongelmaa tutkitaan seuraavassa kappaleessa.

5.4.3. Arkkitehtonisen tyylin valinnan ongelma

Tämä kappale käsittelee arkkitehtonisen tyylin valinnan ongelmaa. Miten arkkitehti valitsee järjestelmän arkkitehtonisen tyylin? Tyylien uudelleenkäyttöä vaikeuttaa mm. "tyylioppaiden" puuttuminen. Mitä arkkitehtonisia tyylejä on olemassa? Ja vaikka arkkitehti tietäisikin useita eri tyylejä, miten valita se oikea, joka soveltuu juuri tämän ongelman ratkaisemiseen? Arkkitehtuureiden tutkimus tieteenalana ei ole vielä toistaiseksi tuottanut mitään kattavia ohjeita ongelmien ratkaisemiseksi [Garlan & Shaw 96]. Arkkitehtuureja tekevätkin Medvidovicin [1999] mukaan ihmiset, joilla on "arkkitehtuurillisia näkemyksiä" ja jotka käyttävät apunaan "mustan magian keinoja".

Medvidovicin kommentista voimmekin päätellä, että tyylivalintaan vaikuttaa pääasiassa arkkitehdin kokemus. Jos arkkitehti on kokematon, arkkitehtuuri on muodostettava luottaen omaan intuitioon. Sama koskee myös arkkitehtonisen tyylin valintaa. Jos arkkitehti ei ole koskaan joutunut ratkaisemaan käsillä olevaa ongelmaa eikä näinollen pysty tunnistamaan vaihtoehdoista sopivaa tyyliä, valinta on tehtävä menetelmällisesti luottaen omaan intuitioon sekä "raakaan voimaan" eli eri vaihtoehtojen kokeilemiseen ja arviointiin. Jos arkkitehdillä on kokemusta sovellusalueesta tai aikaisemmin onnistuneista arkkitehtonisista ratkaisuista, hän todennäköisesti yrittää käyttää samoja tai saman tyyppisiä ratkaisuja tyylivalinnoissa.

Tehtiinpä tyylin valinta millä tahansa keinolla, valinnassa tehdään kauaskantoisia ratkaisuja tulevan järjestelmän ominaisuuksista. Esimerkiksi järjestelmän suorituskyvylle ja muunneltavuudelle asetetut odotukset ovat toistensa osittain poissulkevia vaihtoehtoja

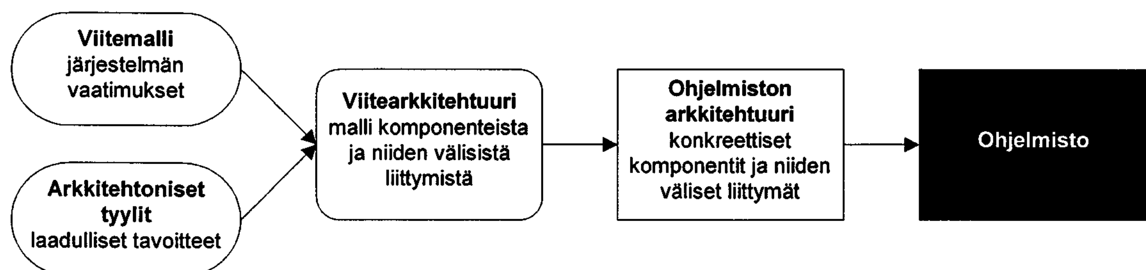
verratessa, toteutettaisiinko järjestelmä kaksi- vai kolmitasoarkkitehtuurin mukaisesti. Arkkitehtuurisuunnitteluhan ei ole pakollinen toimenpide: toimivan järjestelmän saa lopulta aikaiseksi käyttämällä mitä tahansa menetelmää – vaikkapa Haikalan ja Märijärven [1998] spagettikoodausta (jota myös voidaan pitää eräänlaisena järjestelmän arkkitehtonisen tyylin valintaratkaisuna – kaikilla järjestelmillähän on arkkitehtuuri [Garlan & Shaw 96] [Garlan & Perry 95]).

Kun arkkitehtuurin muodostamiseen käytettävät tyylit ja rakenteelliset ratkaisut saadaan valituksi, järjestelmän kehityksessä siirrytään itse arkkitehtuurin muodostamiseen.

5.4.4. Tyylistä ja vaatimuksista ohjelmiston arkkitehtuuriin

Kun järjestelmän laadullisiin tavoitteisiin perustuva tyylien valinta on suoritettu, siirrytään itse arkkitehtuurin muodostamiseen. Valittu arkkitehtoninen tyyli määrittelee järjestelmän komponenttityypit sekä niiden ajonaikaiset suhteet toisiinsa. Viitemalli heijastaa järjestelmän toiminnallisten vaatimusten jakautumista ja tietovirtoja niiden välillä. Siirtyminen itse ohjelmiston arkkitehtuurin konkreettiseen toteutukseen tapahtuu viitearkkitehtuurin kautta.

Viitearkkitehtuuri on viitemalli, jonka abstraktit elementit on määritelty ohjelmiston toteuttaviksi konkreettisiksi komponenteiksi, joiden yhteistoiminnan avulla saadaan toteutetuksi abstraktin viitemallin määrittelemä toiminnallisuus ja osien väliset liittymät [Bass ym. 98]. Viitearkkitehtuuri koostuu siis järjestelmän toiminnallisista vaatimuksista sekä tyyleistä, jotka on valittu toteuttamaan vaatimukset ja laadulliset tavoitteet.



Kuva 29: Arkkitehtuurin muodostaminen tyylien ja viitemallin avulla. Muunneltu [Bass ym. 98]:sta

On tärkeää huomata, että arkkitehtoninen tyyli, viitemalli tai viitearkkitehtuuri eivät ole ohjelmiston arkkitehtuuri, vaan käytännöllisiä välivaiheita arkkitehtuurin muodostamisessa. Jokainen niistä on tulos järjestelmän kehityksessä tehdyistä ensimmäisen vaiheen suunnittelupäätöksistä (kappale 3.2.2.), ja jokainen luo pohjaa tuleville kehitysvaiheille [Bass ym. 98].

Viitearkkitehtuuri voi toteuttaa tyyliin kuuluvat erilliset komponentit yhtenä konkreettisena komponenttina. Esim. kääntäjän koodin generointi- ja optimointimoduuli voivat viitearkkitehtuurissa olla yksi ja sama komponentti (kts. kappale 4.3.1.). Komponentti voi siis toteuttaa joko vain osan toiminnosta, tai vaihtoehtoisesti useamman viitemallissa määritellyn toiminnan.

Ehkä pisimmälle viety ajatus viitearkkitehtuureista ovat sovellusaluekohtaiset arkkitehtuurit (DSSA- *domain specific software architecture*). DSSA:n ajatus perustuu Parnasin [1976] esittämiin ohjelmaperheisiin, joissa kaikki ohjelmaperheen (tai tuotelinjan) jäsenet jakavat yhteisen viitearkkitehtuurin. Viitearkkitehtuurin avulla on tarkoitus pystyä nopeasti ja luotettavasti tuottamaan samankaltaisia samaan ohjelmaperheeseen kuuluvia ohjelmia. Viitearkkitehtuuri määrittelee ohjelmaperheen yhteneväisyydet ja eroavaisuudet [Clements & Northrop 96]. Edellä mainitun arkkitehtuurin muodostamisen välivaiheet kuvaavan kaavion mukaan viitearkkitehtuurit ovat viimeinen välivaihe ennen ohjelmiston arkkitehtuurin muodostumista. Tämän perusteella voisi päätellä, että sovellusaluekohtaisten arkkitehtuureiden avulla voitaisiin saada aikaan merkittäviä hyötyjä. DSSA:ta ei kuitenkaan tämän tutkielman piirissä tarkemmin käsitellä aihepiirin laajuuden vuoksi.

5.4.5. Yhteenveto tyylien käytöstä arkkitehtuurisuunnittelun apuna

Arkkitehtoniset tyylit voidaan jaotella idiomeiksi ja ratkaisumalleiksi sekä viitemalleiksi. Tyyli voi olla joko hyvin pieni (mikroarkkitehtuuri) tai hyvin suuri uudelleenkäytettävä kokonaisuus (makroarkkitehtuuri). Tyylien uudelleenkäyttöä koskevat samat ehdot kuin uudelleenkäyttöä yleensäkin: uudelleenkäytettävän rakenteen on helpotettava suunnittelutyötä, uudelleenkäytön on oltava helpompaa kuin rakenteen kirjoittaminen uudelleen, on tiedettävä, mitä mikäkin uudelleenkäytettävä rakenne tekee, sekä uudelleenkäytettävän rakenteen on oltava löydettävissä nopeammin kuin sen rakentaminen itse kestäisi.

Tyylien valinnassa vaikutetaan järjestelmän tuleviin laadullisiin ominaisuuksiin. Valintaan ei ole mitään yleispäteviä ohjeita: jos arkkitehti on kokematon, eri vaihtoehtoja on punnittava menetelmällisesti. Kokenut arkkitehti todennäköisesti soveltaa aikaisempia kokemuksiaan ongelman ratkaisemiseksi.

5.5. Millainen on hyvä arkkitehtuuri?

Vaikka useat tässä kappaleessa mainituista yleisistä arkkitehtuurin mittareista käsitellään tarkemmin tutkielman muissa jaksoissa, katson kuitenkin tarpeelliseksi koota tärkeimmät arkkitehtuurin piirteet yhdeksi omaksi kappaleekseen. Lista ei ole kattava, mutta antaa jonkinlaisen yleiskäsityksen hyvän arkkitehtuurin eri piirteistä.

Arkkitehtuuria voidaan mitata sen mukaan, kuinka hyvin se toteuttaa sidosryhmien sille asettamat vaatimukset ja laadulliset tavoitteet [Boehm 98]. Yleisiä hyviä arkkitehtuurin ominaisuuksia ovat sen muunneltavuus ja kyky palautua tehtyjen muutosten jälkeen takaisin muotoonsa [Booch 99] [Bass ym. 98]. Arkkitehtuurin tulisi olla tasapainossa vaatimusten ja laadullisten tavoitteiden näkökulmasta sekä taloudellisten että teknisten näkökulmien välillä [Booch 99].

Hyvä arkkitehtuuri on rakenteeltaan yksinkertainen ja selkeä, jolloin arkkitehtuuri on helposti lähestyttävissä. Selkeys saadaan aikaan eri piirteiden selkeällä erottelulla ja vastuualueiden tasaisella jakamisella arkkitehtuurin muodostavien komponenttien välillä (ei vain yksi suuri kaiken suorittava komponentti, vaan useampi pieni) [Booch 99].

Bass ym. [1998] ehdottavat, että arkkitehtuurin olisi hyvä olla yhden arkkitehdin tuotos. Jos arkkitehtuurin muodostamisessa on mukana useampi arkkitehti, joukolla pitäisi olla nimettynä pääarkkitehti. Arkkitehdillä tulisi olla käytössään järjestelmän tekniset vaatimukset sekä laadulliset tavoitteet arkkitehtuurin muodostamisprosessin aikana [Bass ym. 98]. Vain yhden arkkitehdin käyttöä ei perustella tarkemmin, mutta kysymys lienee vanhasta toteamksesta "mitä useampi kokki sitä huonompi soppa" – muodostamisprosessi on helpompi hallita kun tekijöitä (tai vastuuhenkilöitä) on vain yksi.

Arkkitehtuurin kuvausta tulisi kierrättää arvioitavana sidosryhmien keskuudessa [Bass ym. 98]. Kierrätyksen avulla arvokkaan palautteen saamisen lisäksi sidosryhmät sitoutetaan paremmin järjestelmän kehittämiseen. Käytännössä arkkitehti voi kierrätyksen avulla turvata myös omaa selustaansa – jos sidosryhmät eivät osallistu arkkitehtuurin arvointiin sen muodostamisvaiheessa, on myöhemmässä vaiheessa epäonnistumisista turha syyttää vain arkkitehtia.

5.6. Arkkitehtuurista järjestelmän toteutukseen

Tässä kappaleessa käsitellään, mitä arkkitehtuurisuunnittelun loppuvaiheessa tapahtuu, ja miten arkkitehtuurisuunnittelusta siirrytään järjestelmän suunnittelu- ja toteutusvaiheeseen. Kun arkkitehtuuri on pääpiirteittäin muodostettu, suositellaan rakennettavaksi arkkitehtuurin miniversio [Bass ym. 98]. Tätä järjestelmän miniversiota voidaan käyttää sekä suunnittelun apuna että arkkitehtuurin sopivuuden varmistamisessa – toteuttaako arkkitehtuuri sille asetetut vaatimukset ja laadulliset tavoitteet.

5.6.1. Minijärjestelmän rakentaminen ja arviointi

Minijärjestelmän (*skeletal system*) toteuttamisen ajatuksena on muodostaa järjestelmän pienin mahdollinen versio, joka heijastaa muodostetun arkkitehtuurin käyttäytymistä. Jokaisesta arkkitehtuurin määritelmässä esiintyvästä komponentista rakennetaan versio, jonka avulla komponentin käyttäytymistä voidaan tutkia. Kun jokaisen komponentin "luuranko" ja sen rajapinnat on määritelty ja toteutettu, järjestelmästä saadaan aikaan ensimmäinen ajettava versio, jota voidaan käyttää arkkitehtuurin analysointiin [Bass ym. 98].

Minijärjestelmän jokainen komponentti on osa toimivaa prototyyppiä, vaikka rajapintametodien toteutus olisikin vielä joko simuloitua tai toteutusta ei ole ollenkaan olemassa. Minijärjestelmän ansiosta järjestelmästä on koko ajan olemassa ajettava (ja testattavissa oleva) versio. Ajettavan version olemassaololla on suuri merkitys kehitysryhmän motivaatioon – työn tulokset näkyvät jatkuvasti konkreettisesti ajettavana ohjelmana. Minijärjestelmän kehittämisellä saadaan aikaan rakenne, joka kestää projektin loppuun asti; ainoastaan simuloidut ratkaisut muuttuvat iteraatioiden myötä todellisiksi ratkaisuiksi [Bass ym. 98].

Minijärjestelmää voidaan jatkuvasti käyttää järjestelmän arviointiin: toteuttaako arkkitehtuuri sille asetetut vaatimukset ja tavoitteet. Minijärjestelmän ajettavan version avulla voidaan keskittyä heti alusta asti järjestelmän toteutuksen kannalta vaikeiksi oletettuihin kohtiin. Minijärjestelmä alentaa todennäköisesti myös järjestelmän integrointivaiheen kustannuksia – miniarkkitehtuurin muodostaminenhan on itsessään järjestelmän eri osien integroinnin testaamista [Bass ym. 98].

Bass ym:n [1998] mukaan järjestelmän miniversiolla on hyvin paljon yhteistä järjestelmän protoilulähestymistavan kanssa – molemmissa on tarkoituksena iteraatioiden avulla saada aikaan kokonaisuudessaan vaatimukset ja laadulliset ominaisuudet toteuttava järjestelmä. Ero muodostunee siitä konkretisoinnista, että minijärjestelmä muodostetaan nimenomaan arkkitehtuurisuunnitelman perusteella – ei esim. käyttöliittymäprototyyppien perusteella.

Kirjan arkkitehtuuripohjainen iteratiivinen kehitysmenetelmä [Bass ym. 98] jatkuu minijärjestelmän muodostamisen jälkeen yksittäisten komponenttien arkkitehtuurin suunnittelulla käyttäen hyväksi arkkitehtonisia tyylejä, suunnittelu- sekä koodimalleja.

5.7. Arkkitehtuurisuunnittelun yhteenveto

Arkkitehtuurin muodostamiseksi ei ole mitään yleisesti hyväksyttyä prosessimallia. Lähtökohtana on järjestelmälle asetetut vaatimukset sekä laadulliset tavoitteet, joista arkkitehti muodostaa arkkitehtuurin käyttäen apuna arkkitehtonisia tyylejä, suunnittelumenetelmiä sekä puhdasta intuitiota. Ongelmaa voidaan lähestyä joko alhaalta ylös eli muodostaa järjestelmä koostamalla se (mahdollisesti olemassaolevista) komponenteista, tai ylhäältä alas suunnittelumenetelmien käytön avulla. Arkkitehtuurin muodostamisessa kannattaa pyrkiä mahdollisimman suureen uudelleenkäyttöön joko komponentteja, arkkitehtonisia tyylejä, viite- tai suunnittelumalleja taikka sovelluskehyksiä uudelleenkäyttämällä. Uudelleenkäytön yhteydessä täytyy ottaa huomioon mahdolliset arkkitehtuureiden yhteensopivuuden ongelmat.

Ohjelmiston arkkitehtuuri muodostetaan viitearkkitehtuurin kautta käyttäen hyväksi järjestelmän toiminnallisuuden määrittelemää viitemallia sekä arkkitehtonisia tyylejä, jotka heijastavat järjestelmän laadullisia tavoitteita. Järjestelmän rakenteellisen arkkitehtuurin valmistuttua arkkitehtuuria voidaan testata rakentamalla minijärjestelmä, joka toteuttaa minimaalisesti järjestelmän kaikki komponentit ja niiden väliset liittymät.

6. YHTEENVETO

Tutkielman tavoitteena on ollut antaa lukijalle kokonaisnäkemys siitä, mitä ohjelmistoarkkitehtuureilla tarkoitetaan, sekä kuvata sen keskeisimpiä sovelluskohteita.

Luvussa 3 aihetta käsiteltiin ohjelmistoarkkitehtuurin määritelmiä niiden eri piirteiden vertailun ja analyysin kautta. Ohjelmistoarkkitehtuurin kirjallisuudessa esitettyjä määritelmiä voidaan tyypillisesti pitää joko suoraan kolmen luvussa esitetyn perusmääritelmän kopioina, tai ne voidaan palauttaa johonkin niiden yhdistelmään. Luvussa käsiteltiin myös sitä, miksi ohjelmiston arkkitehtuurin muodostamista pidetään tärkeänä järjestelmien kehittämisen välivaiheen tuotoksena. Ohjelmiston arkkitehtuuria ja sen kuvausta käytetään sidosryhmien välisen kommunikoinnin mahdollistavana tekijänä. Lisäksi arkkitehtuuri sisältää järjestelmän ensimmäisen vaiheen suunnittelupäätökset, joiden avulla järjestelmän ominaisuuksia voidaan analysoida jo ennen varsinaista tuotantovaihetta. Arkkitehtuuri toimii myös potentiaalisesti uudelleenkäytettävänä järjestelmän abstraktiona. Arkkitehtuurin vaikutusten todettiin olevan kaksisuuntaisia; arkkitehtuuri vaikuttaa takaisin sen muodostumiseen alunperin vaikuttaneisiin tekijöihin.

Luvussa 4 käsiteltiin arkkitehtonisten tyylien eri määritelmiä sekä sitä, miten tyylit muodostuvat järjestelmien kehittämisen ongelmien hyviksi havaituista ratkaisumalleista. Luvussa esitettiin useita esimerkkejä arkkitehtonisista tyyleistä, joita voidaan käyttää järjestelmän arkkitehtuurisuunnittelun apuna. Järjestelmän muodostamisessa voidaan käyttää useita päällekkäisiä tai hierarkkisia tyylejä. Lisäksi luvussa käsiteltiin, kuinka järjestelmiä voidaan analysoida niiden arkkitehtuureissa käytettyjen tyylien perusteella.

Luku 5 käsitteli arkkitehtuurisuunnittelua. Ohjelmiston arkkitehtuurin muodostamiselle esitettiin kolme vaihtoehtoista lähestymistapaa: arkkitehtuurin varastaminen jostain aikaisemmasta järjestelmästä tai kirjallisuudessa esitetystä mallista, menetelmällisen lähestymistavan käyttäminen, tai intuition avulla. Edellisten käytön suhde vaihtelee riippuen arkkitehdin (tai arkkitehtien) kokemuksesta. Järjestelmän rakenteellisen arkkitehtuurin muodostamiseksi esitettiin kaksi lähestymistapaa: ongelman lähestyminen yl-

häältä alas- tai alhaalta ylöspäin. Ylhäältä alaspäin -menetelmässä arkkitehtuuri pyritään muodostamaan osittamalla järjestelmä sen muodostaviksi komponenteiksi. Alhaalta ylöspäin lähestyvässä menetelmässä arkkitehtuuri muodostetaan kokoamalla ohjelma (mahdollisesti olemassaolevista) komponenteista. Lähestymistavan esimerkkinä voidaan pitää komponenttipohjaista ohjelmistotuotantoa. Realistisen lähestymistavan esitetään vaativan kuitenkin molempien edellisten lähestymistapojen yhteiskäyttöä.

Tutkielman kontribuutioina voidaan siis pitää seuraavia:

- Arkkitehtuurin eri määritelmien kirjallisuustutkimus, vertailu sekä määritelmien puutteiden toteaminen.
- Esitys arkkitehtuurin suhteista järjestelmien kehittämisen yleisiin prosessimalleihin (vesiputous ja spiraalimalli), suunnittelumenetelmiin sekä ohjelmiston elinkaareen.
- Laajahko katsaus yleisimmin käytettyihin arkkitehtonisiin tyyleihin sekä niiden luokittelu ja vertailu sekä tyypillisten sovelluskohteiden toteaminen.
- Arkkitehtonisten tyylien uudelleenkäytön kuvaaminen osana arkkitehtuurisuunnitteluprosessia ja suhteet muihin uudelleenkäytön menetelmiin (suunnittelumallit jne.).

LÄHDELUETTELO

[AWG 99]

The IEEE Architecture Working Group (AWG): IEEE's Recommended Practice for Architectural Description [online www-kalvoesitys]. [viitattu 5.19.99]. Saatavilla www-muodossa: <URL: http://www.pithecanthropus.com/~awg/intro/sld001.htm>

[Bass ym. 98]

Bass Len, Clements Paul, Kazman Rick: Software Architecture in Practice (kolmas painos). Addison Wesley, 1998

[Boehm 95]

Boehm, Barry: Anchoring the Software Process. USC-CSE-95-507, USC Center for Software Engineering Technical Report, 1995

[Boehm & Port 98]

Boehm Barry ja Port Dan: Escaping the Software Tar Pit: Model Clashes and How to Avoid Them. USC-CSE-98-517, USC Center for Software Engineering Technical Report, 1998

[Boehm & Scherlis 92]

Boehm Barry ja Scherlis W. L.: Megaprogramming. Proceedings of the DARPA Software Technology Conference, April 1992

[Boehm ym. 98]

Boehm Barry, Port Dan, Egyed Alexander ja Abi-Antoun Marwan: The MBASE Life Cycle Architecture Milestone Package: No Architecture Is An Island. USC-CSE-98-510, USC Center for Software Engineering Technical Report, 1998

[Booch 99]

Booch Grady: Software Architecture and the UML [online MS PowerPoint kalvoesitys]. Rational Software, 1999. [viitattu 5.19.99] Saatavilla www-muodossta: <URL: http://www.rational.com/uml/img/arch.zip>

[Brooks 87]

Brooks Frederic Jr: No Silver Bullet – Essence and Accidents of Softwre Engineering. IEEE Computer, Vol.20, No.4, 1987

[Budd 97]

Budd Timothy: An Introduction to Object-Oriented Programming, 2nd Edition. Addison Wesley Longman, 1997

[Callon 96]

Callon Jack D.: Competitive advantage through information technology. McGraw-Hill 1996.

[Clements & Northrop 96]

Clements Paul C. ja Northrop Linda M.: Software Architecture: An Executive Overview. CMU/SEI-96-TR-003, CMU Software Engineering Institute Technical Report, February 1996

[Clements 96]

Clements Paul C.: Coming Attractions in Software Architecture. CMU/SEI-96-TR-008, CMU Software Engineering Institute Technical Report, January 1996

[Coad 92]

Coad Peter: Object-Oriented Patterns. Communications of the ACM, 1992, Vol 35, No 9, sivut 153-159

[Dijkstra 68].

Dijkstra Edsger: The Structure of the "THE" - Multiprogramming System. Communications of the ACM, 1968, Vol 11, No 5, sivut 341-347

[Earl 89]

Earl Michael J.: Management Strategies for Information Techology. Prentice Hall 1989

[Fayad & Schmidt 97]

Fayad Mohamed ja Schmidt Douglas: Object-oriented application frameworks. Communications of the ACM, October 1997, Vol 40, No 10, sivut 32-28

[Gacek 94]

Gacek Cristina: Software Architecture, the Architecting Process, and Examples of Architecting Infrastructures. Teoksessa: Knowledge Summary USC Center for Software Engineering Focused Workshop on Software Architectures, June 1994

[Gacek 97]

Gacek Cristina: Detecting Architectural Mismatches During System Composition. USC-CSE-97-TR-506, USC Center for Software Engineering Technical Report, July 1997

[Gacek ym. 95]

Gacek Cristina, Abd-Allah Ahmed, Clark Bradford, Boehm Barry: On the Definition of Software System Architecture. USC-CSE-95-TR-500, USC Center for Software Engineering Technical Report, April 1995

[Gamma ym. 94]

Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John: Design Patterns: Micro-Architectures for Reusable Object-Oriented Design. Addison-Wesley, 1994

[Garlan ym. 94]

Garlan David, Allen Robert ja Ockerbloom John: Exploiting style in architectural design environments. Teoksessa: SIGSOFT '94, Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering, 1994, sivut 175-188

[Garlan ym. 95]

Garlan David, Allen Robert ja Ockerbloom John: Architectural mismatch or why it's hard to build systems out of existing parts. Teoksessa: ICSE '95, Proceedings of the 17th international conference on Software engineering, 1995, sivut 179-185

[Garlan & Perry 95]

Garlan David ja Perry Dewayne E.: Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, April 1995, Vol 21, No 4, sivut 269-274

[Garlan & Shaw 94]

Garlan David ja Shaw Mary: An Introduction to Software Architecture. CMU/ SEI-94-TR-21, ESC-TR-94-21, CMU Software Engineering Institute Technical Report, 1994

[Garlan & Shaw 96]

Garlan David ja Shaw Mary: Software Architecture – Perspectives on an Emerging Culture, Prentice-Hall 1996

[Haikala & Märijärvi 98]

Haikala Ilkka ja Märijärvi Jukka: Ohjelmistotuotanto, 6. Painos, Suomen ATK-kustannus 1998

[InfoManager 99]

Enterprise Web Training Material, InfoManager Oy, April 1999

[Johnson 97a]

Ralph E. Johnson: Components, Frameworks, Patterns (extended abstract). ACM SIGSOFT Software Engineering Notes, May 1997, Vol 22, No 3, Sivut 10-17

[Johnson 97b]

Ralph E. Johnson: Components, Frameworks, Patterns (extended abstract). Communications of the ACM, October 1997, Vol 40, No 10, Sivut 39-42

[Kogut & Clements 99]

Kogut Paul ja Clements Paul: The Software Architecture Renaissance [online]. Carnegie Mellon University Software Engineering Institute, 1999 [viitattu 1.6.99]. Saatavilla WWW-muodossa: <URL: ftp://ftp.sei.cmu.edu/pub/sati/Papers_and_Abstracts/SW_Arch_Renaissance.ps >

[Korson & McGregor 90]

Korson Tim ja McGregor John D.: Understanding Object Oriented: A Unifying Paradigm, Communications of the ACM, Vol 33, No 9, September 1990

[Krasner & Pope 88]

Krasner, G.E. ja Pope S.T.: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object Oriented Programming, September 1988, Vol 1, No 3, sivut 26-49.

[Kruchten 95]

Kruchten Philippe B.: The 4+1 View Model of Architecture. IEEE Software, November 1995, Vol 12, No 6, sivut 42-50

[Krueger 92]

Krueger Charles W.: Software Reuse. ACM Computing Surveys, Vol 24, No 2, June 1992

[Kuusi 99]

Kuusi Sampo: Uudelleenkäytettävät komponentit ohjelmistotuotannossa. Pro gradu-tutkielma, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto, Jyväskylä, 1999

[McClain 91]

McClain Gary R., toimittaja: Open Systems Interconnection Handbook. Intertext Publications McGraw-Hill Book Company, New York, NY, 1991

[Medvidovic 99]

Medvidovic Nenad: Arriving at An Architecture, CS 612 Software Architectures Lecture Notes. [online luentokalvot pdf-muodossa]. USC Computer Science Department Spring 1999. [viitattu 31.5.1999]. Saatavilla WWW-muodossa <URL: http://sunset.usc.edu/~neno/teaching/s99/January28.pdf> (Kurssin kotisivu osoitteessa <URL: http://sunset.usc.edu/~neno/teaching/s99/cs612.html>)

[Medvidovic & Taylor 98]

Medvidovic Nenad ja Taylor Richard N.: Separating Fact from Fiction in Software Architecture. Teoksessa: ISAW '98. Proceedings of the third international workshop on Software architecture, 1998, sivut 105-108

[Monroe ym. 97]

Monroe Robert D., Kompanek Andrew, Melton Ralph ja Garlan David: Architectural Styles, Design Patterns, and Objects, IEEE Software, January-February 1997, Vol 14, No 1, sivut 43-52

[Neighbors 89]

Neighbors J. M.: Draco: A method for engineering reusable software systems. Frontier Series: Software Reusability: Volume I – Concepts and Models, ACM Press, New York 1989, sivut 295-319

[Nii 86]

Nii Penny H.: Blackboard systems. AI Magazine, 1986, Vol 7, No 3: sivut 38-53 ja Vol 7, No 4 sivut 82-107

[Parnas 76]

Parnas David: On the Design and Development of Program Families. IEEE Transactions on Software Engineering, March 1976,, VolSE-2, No 1, sivut 1-9

[Perry & Wolf 92]

Perry Dewayne E., Wolf Alexander L.: Foundations for the Study of Software Architecture. ACM SIGSOFT, Software Engineering Notes, Vol 17, No 4, October 1992, sivut 40-52

[Porter & Millar 85]

Porter M. E. ja Millar V.E.: How Information gives you competitive advantage. Harvard Business Review, Vol 63, No 4, July-August 1985, sivut 149-160

[Pressman 97]

Pressman Roger S.,: Software Engineering – A Practitioners Approach, Fourth Edition (European Adaptation), McGraw-Hill 1997

[Rational 99]

Rational Rose: Visual Modeling, UML, Object-Oriented, Component Based Development. [online] Rational Software 1999. [viitattu 5.19.99]. Saatavilla WWW-muodossa <URL: http://www.rational.com/products/rose/ >

[Sametinger 97]

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

[Shaw 95]

Shaw Mary: Comparing Architectural Design Styles. IEEE Software, November 1995, Vol 12, No 6, sivut 27-41

[Sommerville 95]

Sommerville Ian: Software Engineering, Fifth Edition. Addison-Wesley 1995

[STR CS Architecture 99]

Client/Server Software Architectures--An Overview. [online] Carnegie Mellon University Software Engineering Institute, 1999 [viitattu 8.6.1999]. Saatavilla WWW-muodossa: <URL: http://www.sei.cmu.edu/str/descriptions/clientserver.html >.

[STR Three Tier 99]

Three Tier Software Architectures. [online] Carnegie Mellon University Software Engineering Institute, 1999 [viitattu 8.6.1999]. Saatavilla WWW-muodossa: <URL: http://www.sei.cmu.edu/str/descriptions/threetier.html >.

[Sun 99]

Java® 2 Platform API Specification. [online]. Sun Microsystems Inc., 1999 [viitattu 9.6.1999]. Saatavilla WWW-muodossa: <URL: http://java.sun.com/products/jdk/1.2/docs/api/index.html >

[Webster 99]

WWWebster Dictionary. [online tietokanta] Merriam-Webster, Incorporated 1999 [viitattu 5.19.99]. Saatavissa WWW-muodossa : <URL: http://www.m-w.com >

[Yourdon 89]

Yourdon Edward: Modern Structured Analysis, Prentice-Hall International 1989

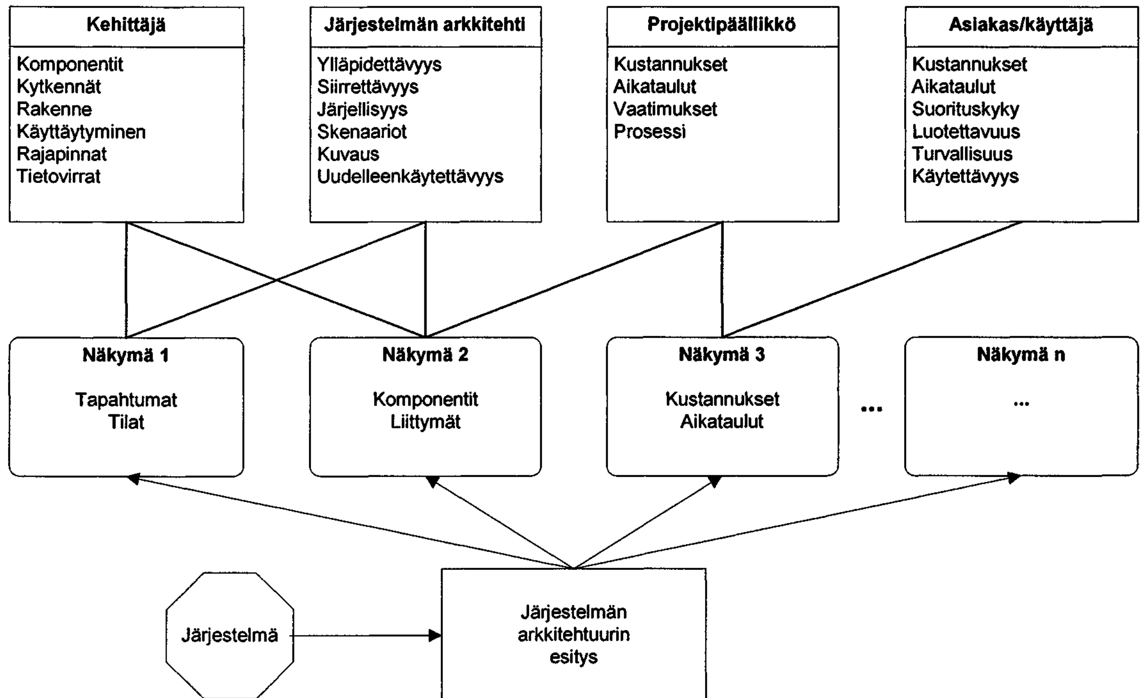
LIITTEET

Liite 1: Arkkitehtuurin kuvauskielten soveltuvuus kuvaamaan järjestelmän arkkitehtuurin eri piirteitä

	MetaH	ControlH	DICAM	WRIGHT	UNICON	LEAP	Rapide	UNAS(Z)
Staattinen rakenne (topologia)	▒	▒	■	■	■	■	■	■
Dynaaminen rakenne (käyttäytyminen)	■	■	▒	■	▒	▒	■	■
Tietovirta		■	▒				■	■
Sovellusaluekohtaiset tiedot	■	■	▒					
Sovellusaluekohtaisuus			■	■	■	■	■	■
Toteutusriippuvaiset tiedot	■	■	▒		▒			▒
Soveltuvuus formaaleille analyyseille	■	■	▒	■	▒	▒	■	▒
Valmis ohjelma	■	■	▒		▒	▒	▒	■
Soveltuvuus luotettavuusanalysointiin		■		■			▒	■
Kustannukset								
Ei-toiminnallisten tietojen formalisointi								

Kuva 30: Kuvauskielten soveltuvuus arkkitehtuurin eri piirteiden kuvaamiseen [Gacek ym. 95]

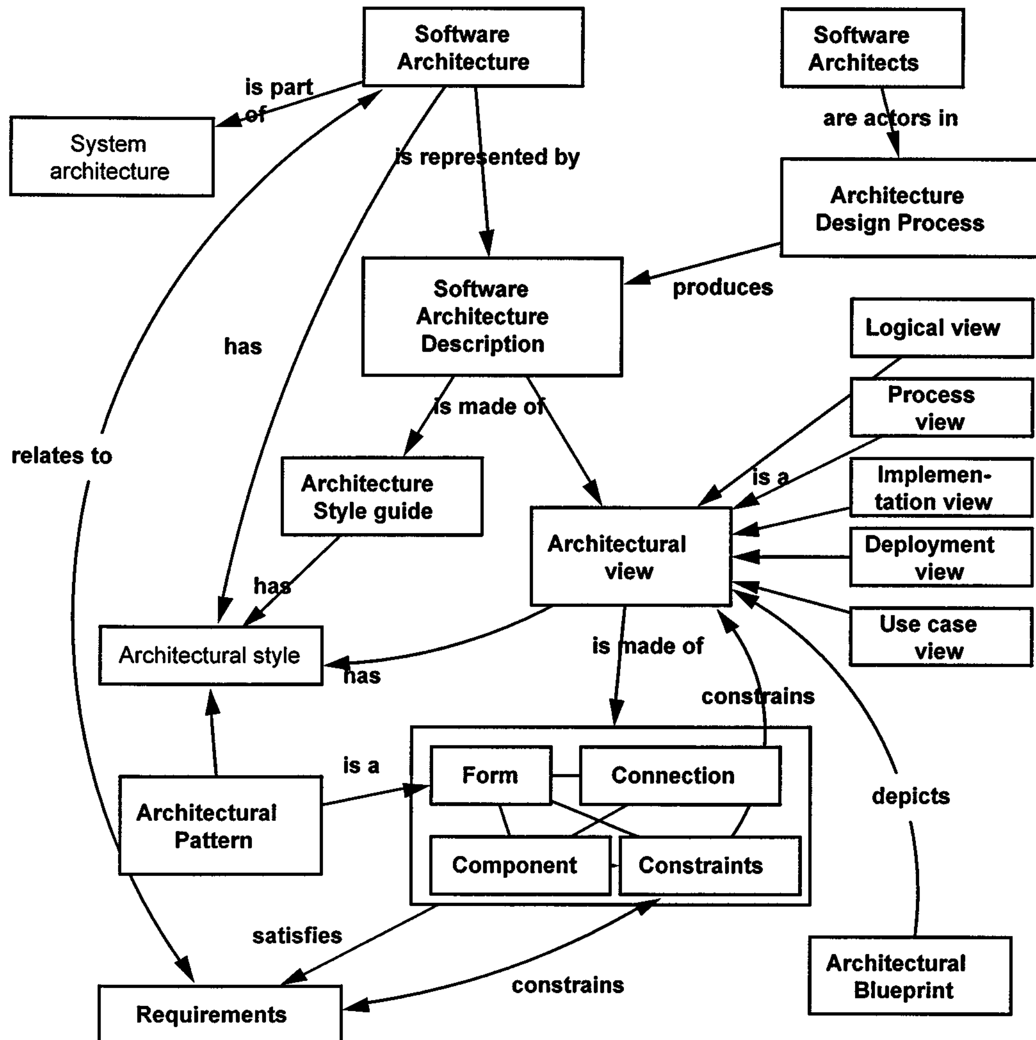
Mitä tummempi taulukon solu, sitä paremmin kuvauskieli soveltuu järjestelmän piirteen kuvaamiseen.



Kuva 31: Tarve arkkitehtuurin eri näkymille [Gacek 94]

Järjestelmän kehitysvaiheessa arkkitehtuurista on tarpeellista olla useita erilaisia näkymiä eri sidosryhmien tarpeita varten. Yhden kuvauksen tai kaavion ei katsota voivan kattaa kaikkien kehitystyöhön osallistuvien osapuolten tarpeita.

Liite 2: Arkkitehtuurin metamalli



Kuva 32: Arkkitehtuurin metamalli [Booch 99]

Boochin arkkitehtuurin metamalli havainnollistaa arkkitehtuuriin liittyvien eri käsittei-den välisiä suhteita.

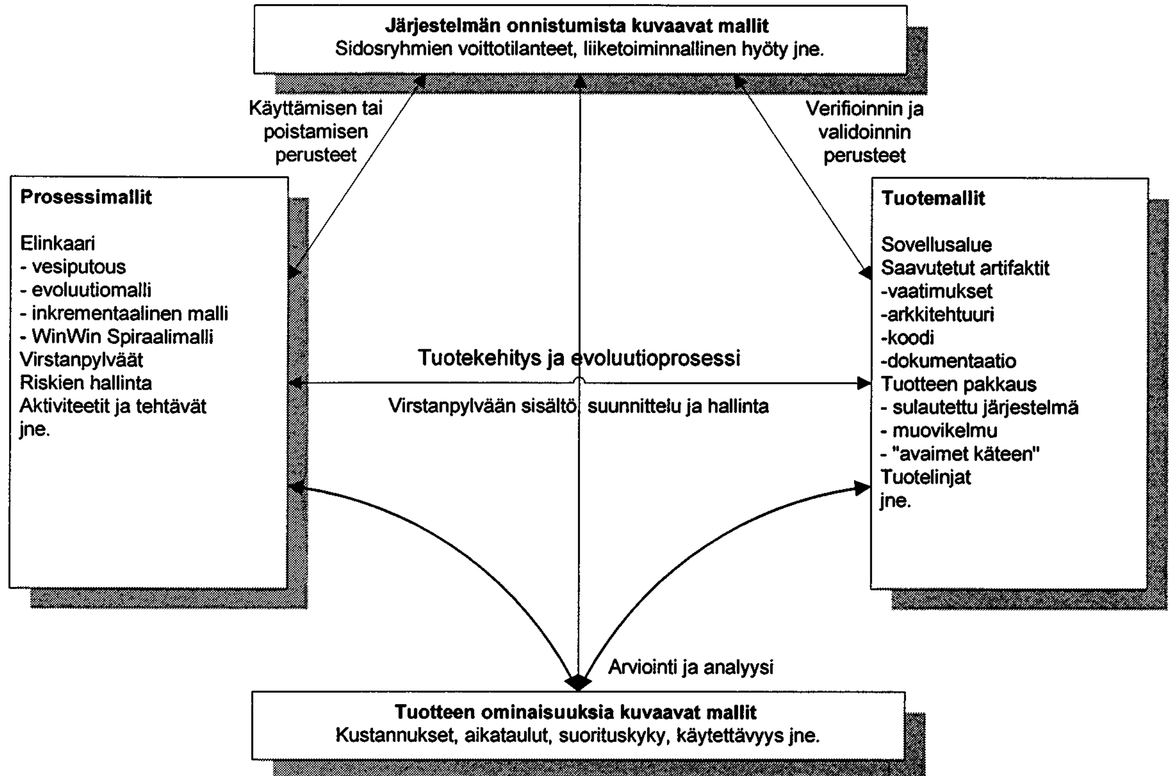
Liite 3: MBASE - lähestymistapa

MBASE (Model Based Architecting and Software Engineering) on Boehmilaisen koulukunnan kehittämä lähestymistapa, jonka avulla pyritään välttämään ohjelmistokehitysprosessissa tapahtuvia mallien yhteentörmäyksiä (model clash).

Mallien yhteentörmäyksellä tarkoitetaan järjestelmän kehityksessä käytettyjen eri mallien välisiä yhteensopivuusongelmia. Prosessimallit käsittelevät järjestelmän kehitysprosessia ja ajan suhteen tapahtuvaa ohjelmiston evoluutiota, järjestelmän ominaisuuksia kuvaavat mallit (*property model*) käsittelevät järjestelmän hintaa, suorituskykyä ja luotettavuutta, järjestelmän onnistumista kuvaavat mallit (*success model*) kuvaavat järjestelmän toimintaa liiketoimintaympäristössä sekä sidosryhmien järjestelmälle asettamia ehtoja. Mallien yhteentörmäys on kehityksessä käytettyjen mallien taustalla olevien oletusten konflikti [Boehm & Port 98].

Ohjelmistokehityksessä käytetyt mallit ovat peräisin useista eri lähteistä. Joitakin malleja saadaan erilaisten säädösten kautta (esim. kirjanpitosäädökset), toisia taas opitaan käyttämään yrityksen ja erehdyksen kautta (IKIWISI – I'll Know It When I See It, "tiedän kun näen sen" käyttöliittymien suunnittelussa käytetty prototyyppilähestymistapa). Joitakin malleja saadaan suoraan kasvatuksesta – käytöksen kultainen sääntöhän on "tee toisille niinkuin haluaisit itsellesi tehtävän". Kultaisen säännön ongelma on että malli olettaa toisten olevan samanlaisia kuin sinä itse!

MBASE-lähestymistapa pyrkii varmistamaan että järjestelmän kehityksessä käytetyt tuotemallit (arkkitehtuuri, vaatimukset, koodi) , prosessimallit (tehtävät, aktiviteetit, virstanpylväät), järjestelmän ominaisuuksia kuvaavat mallit (kustannukset, aikataulut, suorituskyky, luotettavuus) ja järjestelmän onnistumista mittavaat mallit (sidosryhmien voittotilanteet, business case) ovat yhtenäisiä ja vahvistavat toisiaan [Boehm ym. 98].



Kuva 33: MBASE – mallien integrointi [Boehm & Port 98]

Kuvassa 33 on esitetty järjestelmän kehitykseen vaikuttavien eri mallien vuorovaikutuksen suhteet. Mallin toimintaa voidaan esim. kuvata tilanteella, jossa ”kehitettävä järjestelmä on oltava valmis ja esitteillä yhdeksän kuukauden päästä järjestettävillä messuilla.” Jos tuote ei ole valmis aikarajaan mennessä, tuote menettää uutuusarvonsa kuluttajien silmissä. Tämä liiketoiminnallinen järjestelmän onnistumista kuvaava rajoite vaikuttaa järjestelmän muiden mallien sisältöön. Tuotemallista on toistaiseksi jätettävä pois ominaisuuksia (toteutetaan myöhemmin), prosessimallin näkökulmasta tuote on saatava aikaan yhdeksässä kuukaudessa, käytettävyydestä joudutaan tinkimään aikataulujen saavuttamiseksi jne. [Boehm ym. 98]. MBASE-lähestymistavan eri malleja käytetään projektin onnistumisen, tuotteen sisällön, prosessien ja tuotteen ominaisuuksien yhteneväisyyden varmistamiseksi. [Boehm ym. 98].