Ohjelmistoarkkitehtuurin suunnittelua avustavat tekniikat

Tomi Kirjavainen

13.06.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu –tutkielma

Tiivistelmä

Ohjelmistoarkkitehtuurin suunnittelu on olennainen osa ohjelmistokehitysprosessia. Ohjelmistoarkkitehtuurilla tarkoitetaan järjestelmän perusrakennetta, joka kuvaa järjestelmän jakamisen osiin, näiden osien suhteet toisiinsa ja ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän kehitystä. Arkkitehtuuriin vaikuttavat toiminnalliset ja laadulliset vaatimukset.

Arkkitehtuurin suunnittelu lähtee yleensä liikkeelle toiminnallisten vaatimusten pohjalta, joita sitten arvioidaan laadullisia vaatimuksia vasten. Kun kaikki vaatimukset on saatu täytettyä järjestelmän perusarkkitehtuuri on valmis. Monimutkaisuus on yleisin ongelma, jota pyritään ratkaisemaan erilaisilla tekniikoilla. Modulaarisuus on yksi keino vähentää ja hallita järjestelmän monimutkaisuutta. Arkkitehtuuristen kaavojen ja kehyksien soveltaminen on olennainen osa monimutkaisen arkkitehtuurin suunnitteluprosessia. Kaavojen avulla järjestelmää ja sen alijärjestelmiä pystytään kuvaamaan tyyleillä ja malleilla, jotka auttavat hahmottamaan järjestelmän rakennetta. Kehykset puolestaan muodostavat rungon, jonka ympärille arkkitehtuuria voidaan alkaa kehittämään.

Tässä tutkielmassa tutustutaan aluksi ohjelmistoarkkitehtuurin suunnitteluprosessiin, jonka jälkeen pyritään selvittämään tärkeimpiä arkkitehtuurin suunnittelua avustavia tekniikoita. Lisäksi esitellään kuinka palvelukeskeinen arkkitehtuuri voidaan toteuttaa käytännössä.

ACM-luokat (ACM Computing Classification System, 1998 version): D.2.11
Avainsanat: ohjelmistoarkkitehtuuri, arkkitehtuuriset kaavat, arkkitehtuuriset kehykset

Sisältö

1 Johdanto

Ohjelmistoarkkitehti on vastuussa suunnittelusta, dokumentoinnista ja johtamisesta, jotta tuloksena syntyisi asiakasta tyydyttävä järjestelmä (Rozanski & al., 2005). Yleisesti ottaen järjestelmän luomisprosessin tulee olla kokonaisuudessaan arkkitehdin hallussa. Yksi vastuista on luoda ja ylläpitää korkean tason näkymää pääelementeistä. Tämän lisäksi on tärkeää osata valita tarkoitukseen sopiva arkkitehtuurinen ratkaisu; yksi ja sama ratkaisu, joka toimi edellisessä projektissa ei välttämättä sovellu uuteen.

Suunnittelijat, asiakkaat ja johtajat katsovat arkkitehtiin, kun kyse on järjestelmätason suunnittelusta (McBride, 2007). Ohjelmistoprojektin alkaessa arkkitehdin täytyy ennaltaehkäisevästi valvoa ja hallita ohjelmiston rakentaminen etenkin suurissa järjestelmissä. Ohjelmistoratkaisun suunnittelu koostuu toiminnallisten ja laadullisten (ei-toiminnallisten) vaatimusten hallinnasta. Ohjelmistoarkkitehdin täytyy pystyä hallitsemaan ohjelmistoprojekteille ominaista monimutkaisuutta kommunikoijana ja johtajana sekä saada matalamman tason komponentin suunnitteluun ja rakentamiseen liittyvät taidot käyttöön.

Arkkitehtuureihin liittyvä terminologia on lainattu muilta alueilta ja on laajalti käytetty erilaisissa tilanteissa. Esimerkiksi termiä arkkitehtuuri käytetään puhuttaessa mikroprosessoreiden sisäisestä rakenteesta, koneiden sisäisestä rakenteesta, tietoverkkojärjestelmistä, ohjelmistojen rakenteesta ja monilla muilla alueilla. Kun yritämme ymmärtää tietokonejärjestelmää olemme kiinnostuneita mitä sen yksittäiset osat tekevät, kuinka ne toimivat keskenään ja kuinka ne kommunikoivat ympäröivän maailman kanssa; toisin sanoen olemme kiinnostuneita sen arkkitehtuurista. SEI:n (Software Engineering Institute) Software Architecture Group (SEI, 2008) määrittelee ohjelmistoarkkitehtuurin seuraavalla tavalla:

Ohjelmistoarkkitehtuuri on järjestelmän rakenne tai rakenteet, jotka koostuvat ohjelmistoelementeistä, niiden ulkoisesti näkyvistä ominaisuuksista ja elementtien välisistä suhteista.

Olennaista määritelmässä on, että arkkitehtuuri ei kata pelkästään järjestelmän jakamista osiin, vaan myös näiden osien välisiä suhteita ja niiden kehittymistä. Koska suhteet ovat usein luonteeltaan ajonaikaiseen käyttäytymiseen liittyviä, arkkitehtuuri koskee paitsi rakennetta myös käyttäytymistä. Toisaalta arkkitehtuuri ei koske ainoastaan ohjelmiston koodin rakennetta, vaan myös ohjelman suorituksen aikaisia rakenteita, kuten dynaamisia oliorakenteita.

Arkkitehtuuriin vaikuttavat keskeisimmät toiminnalliset vaatimukset, mutta myös laadulliset (ei-toiminnalliset) vaatimukset. Arkkitehtuurin ensimmäinen versio tehdään yleensä toiminnallisten vaatimusten pohjalta, jota arvioidaan sitten laadullisiin vaatimuksiin. Tarvittaessa arkkitehtuuria muokataan niin, että myös laadulliset vaatimukset täyttyvät.

Tässä tutkielmassa perehdytään ohjelmistoarkkitehtuuriin ja erityisesti sen suunnittelua avustaviin tekniikoihin. Luvussa 2 käydään ensin läpi arkkitehtuurin suunnitteluprosessi. Luvussa 3 keskitytään arkkitehtuurin suunnittelua avustaviin tekniikoihin. Luku 4 on yhteenveto käsitellyistä asioista.

2 Ohjelmistoarkkitehtuurin suunnitteluprosessi

Ohjelmistoarkkitehtuurin suunnittelu keskittyy järjestelmän jakamiseen elementteihin ja näiden väliseen vuorovaikutukseen toiminnallisten ja ei-toiminnallisten vaatimusten toteuttamiseksi. Ohjelmistojärjestelmä voidaan kuvata suunnittelupäätösten hierarkiaksi (hierarchy of design decisions). Hierarkian kullakin tasolla on joukko suunnittelusääntöjä, jotka jollain tavalla sitovat tai liittävät elementit kyseiselle tasolle. Tässä luvussa kuvataan arkkitehtuurin suunnitteluprosessia. Tämän prosessin ensisijainen tulos on *arkkitehtuurinen kuvaus* (architectural description).

2.1 Prosessin yleiskuvaus

Osana yleistä ohjelmistokehitysprosessia *ohjelmistokehityksen arkkitehtuurinen näkymä* keskittyy sovelluksen tai järjestelmän suunnitteluun ja kuinka suunnittelu ohjaa kehitystä. Sewell (2002) kuvaa tätä arkkitehtuurivetoiseksi ohjelmiston rakentamiseksi, joka koostuu *esisuunnitteluvaiheesta* (predesign phase), *kohdealueen analysointivaiheesta* (domain analysis phase), *skemaattisesta suunnitteluvaiheesta* (schematic design phase) ja *suunnitelman kehitysvaiheesta* (design development phase). Vaiheet suoritetaan peräkkäin edellä esitetyssä järjestyksessä, mutta kaikkia vaiheita ei välttämättä suoriteta jokaisen iteraation aikana.

Tavallisesti arkkitehti osallistuu ohjelmistokehitysprosessiin jo esisuunnitteluvaiheessa. Arkkitehdillä voi olla toimialueen asiantuntemusta varsinkin yrityksen sisäisissä ohjelmistokehitysprosesseissa. Vaikka luotava järjestelmä tehtäisiinkin ulkoisen yrityksen toimesta, arkkitehdin on syytä olla mukana alusta lähtien, jotta projektin laajuus osataan ottaa huomioon järjestelmän alustavissa suunnitelmissa, kuten budjetissa ja aikataulussa. Arkkitehdin täytyy kuunnella tilaajaa ja tutustua toimialueeseen, johon järjestelmä toteutetaan. Yrityksen olemassa olevat IT-järjestelmät, kuten palvelimet, tietokannanhallinta- ja käyttöjärjestelmät tulee ottaa huomioon ennen kuin luotavan järjestelmän

arkkitehtuurisuunnittelu aloitetaan. Olemassa olevasta teknologia-alustasta tulee osa kontekstia, jossa vaatimukset muotoillaan, eikä osa ratkaisua.

Toimialueen analysointivaiheessa arkkitehti pyrkii ymmärtämään ja dokumentoimaan järjestelmän tilaajan ja käyttäjien tarpeet mahdollisimman tarkasti. Tietoa toimialueesta voi saada toimialueasiantuntijoilta, kirjallisuudesta ja aikaisempien tai vastaavien järjestelmien vaatimusmäärittelyistä. Tämä vaihe vastaa ohjelmistotuotantovaiheista läheisesti vaatimusten analysointia ja määrittelyä. Albinin (2003) mielestä toimialueen analysointivaihe on yksi tärkeimmistä ohjelmistoarkkitehturoinnin vaiheista, sillä hyvin tehtynä tämä vaihe vaikuttaa merkittävästi yrityksen ohjelmistokehitysprosessin onnistumiseen. Arkkitehdin tehtävänä on varmistaa, että muut projektin työntekijät ymmärtävät toimialueen analysointivaiheen tulokset, koska ne esittävät ongelma-alueiden semantiikan. Väärien ongelmien ratkaisu voi johtaa ajan ja rahan tuhlaamiseen ja jopa kokonaisten projektien hylkäämiseen.

Skemaattisen suunnitteluvaiheen aikana arkkitehti valmistelee arkkitehtuuritason suunnitelmaa, joka määritellään arkkitehtuurikuvauksessa. Suunnitelma kuvataan korkean tason malleilla, jotka esittävät järjestelmän toimintaa, siihen syötettyä ja käsiteltyä tietoa, käyttöliittymää, ratkaisun modulaarista rakennetta sekä sovelluksen toteuttamiseen tarvittavaa teknologiaa tai erilaisten suunnittelu- tai teknologiapäätösten perusteluita.

Suunnitelman kehitysvaiheessa arkkitehtuurikuvausta hiotaan ja laaditaan vaihtoehtoisia suunnitelmia. Arkkitehtuurikuvausta viimeistellään, kunnes tarkat aikataulut voidaan luoda. Skemaattista suunnittelu- ja suunnitelman kehitysvaihetta voidaan toistaa useaan kertaan lopullisen suunnitelman saavuttamiseksi. Raja näiden kahden vaiheen välillä muodostuu usein epämääräiseksi. Kun suunnitelma on saatu tarpeeksi yksityiskohtaiseksi riskiarviointia varten, voidaan tehdä päätös kehitysprosessin jatkamisesta.

Arkkitehtuurin luomisprosessi on yleisen kehitysprosessin laajennus, jota periaatteessa voidaan soveltaa missä tahansa kehitysprosessivaiheessa, mutta saavutettavan hyödyn

kannalta mahdollisimman aikaisin. Tämä luomisprosessi voidaan muodostaa seuraavista askelista (Albin, 2003):

1. Ongelman ymmärtäminen
2. Suunnitteluelementtien ja niiden välisten suhteiden tunnistaminen
3. Arkkitehtuurisuunnitelman evaluointi
4. Arkkitehtuurisuunnitelman muuntaminen

Ensimmäinen askel on kiistämättä tärkein, koska se vaikuttaa sitä seuraavan suunnitelman laatuun. Ilman ongelman selkeää ymmärrystä ei ole mahdollista luoda toimivaa ratkaisua.

Toisessa askeleessa tunnistetaan suunnitteluelementit ja niiden keskinäiset riippuvuudet. Suunnitteluprosessin alkuvaiheissa muodostetaan sovelluksen yksinkertainen toiminnallinen rakenne, joka muodostaa lähtökohdan seuraaville suunnittelutehtäville ja suunnitelman muuntamiselle.

Kolmannessa askeleessa punnitaan arkkitehtuurin vastaavuutta laatuattribuuttien (ks. kohta 3.3) vaatimuksiin. Sovelluksen toiminnallista käyttäytymistä ei pystytä testaamaan parhaiten arkkitehtuurisesta koostumuksesta. Kuitenkin monet muut laatuattribuutit voidaan arvioida tutkimalla suunnitelmaa tai toteuttamalla prototyyppejä arkkitehtuurisesti merkittävien komponenttien välisestä toiminnasta.

Neljäs askel käsittää suunnitteluoperaatioiden soveltamista arkkitehtuurisuunnitelman muuntamisesta uudeksi suunnitelmaksi, joka osoittaa laatuattribuuttien vaatimuksia paremmin kuin edellinen suunnitelma. Vaihe voidaan toistaa useita kertoja ja jopa suorittaa rekursiivisesti.

2.2 Arkkitehtuurin elementit

Arkkitehtuuri koostuu osista joita kutsutaan *arkkitehtuurielementeiksi* tai lyhyemmin vain elementeiksi. Arkkitehtuuri käsittelee elementtejä ja niiden välisiä suhteita, mutta ei arkkitehtuuriin kuuluvien elementtien sisäistä rakennetta. Tämä muodostaa eron arkkitehtuurisuunnittelun ja yksityiskohtaisen suunnittelun välille (Koskimies & Mikkonen, 2005).

Arkkitehtuurielementeistä käytetään myös nimiä komponentti ja moduuli. Tässä tutkielmassa käytetään kuitenkin termiä elementti, koska komponentilla viitataan usein ohjelmointitason komponenttimallin käyttöön (kuten J2EE tai .NET) ja moduulilla tarkoitetaan usein ohjelmointikielen rakennetta.

Rozanski ja Woods (2005) esittelevät seuraavat avainattribuutit elementille:

- Selvästi määritelty joukko *vastuita* (responsibilities).
- Selvästi määritelty *raja* (boundary).
- Joukko selvästi määriteltyjä *rajapintoja* (interfaces), jotka määrittelevät *palvelut* (services), joita elementti tarjoaa muille elementeille.

Arkkitehtuurielementin luonne riippuu tarkasteltavan järjestelmän tyypistä ja tarkastelun asiayhteydestä. Ohjelmointikirjastot, alijärjestelmät, jaeltavat eli käyttöönotettavat ohjelmistoyksiköt, uudelleenkäytettävät ohjelmistotuotteet (kuten tietokannan hallintajärjestelmät) tai kokonaiset sovellukset voivat muodostaa arkkitehtuurielementtejä riippuen rakennettavasta järjestelmästä.

Elementin kokoa ei ole siis rajoitettu: elementti voi olla pieni yksinkertaisia palveluita tarjoava olion kaltainen yksikkö tai suuri sovelluksen kokoinen yksikkö. Koskimies ja Mikkonen (2005) esittävät kuitenkin kokoa rajoittavaksi nyrkkisäännöksi, että elementin tulisi olla yhden henkilön hallittavissa.

2.3 Ongelman ymmärtäminen

Monet ohjelmistoprojektit ja tuotteet ovat epäonnistuneita, koska ne eivät ratkaisseet haluttua liiketoimintaongelmaa tai eivät olleet sijoituksensa arvoisia. Ohjelmistokehittäjät, joille ei ole annettu selkeää suuntaa ongelman ratkaisuun, voivat hukata resurssejaan teknisiin ongelmiin ja alkuperäinen ongelma jää ratkaisematta. Albin (2003) kutsuu tätä *toteutusansaksi* (implementation trap). Ansan välttääkseen pitäisi pystyä kysymään itseltään minkä ongelman suunnittelupäätös ratkaisee. Jos vastaus on teknisen tai toteutusongelman ratkaisu, niin ongelman kysymys tulisi tehdä uudelleen. Kysymyksen toistaminen tulisi lopulta johtaa alkuperäiseen liiketoimintaongelmaan. Jos kysymykset eivät johda alkuperäiseen ongelmaan, niin tämä on merkki siitä, että kyseisen suunnittelupäätöksen polku johtaa toteutusansaan.

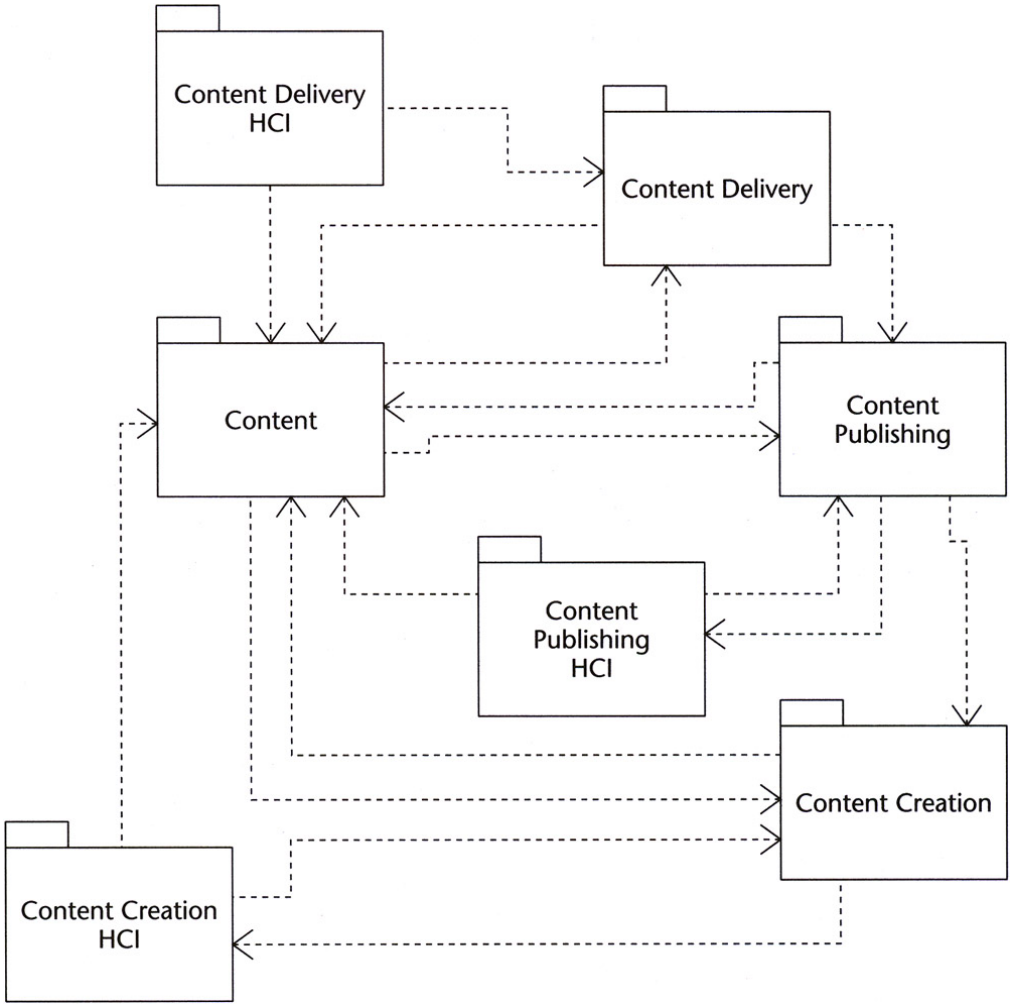
Ohjelmistokehitysprojekteissa ongelman muodostaa usein se, että kehitysryhmässä on niitä, jotka pitävät suunnittelusta ja niitä, jotka pitävät ohjelmoinnista (Garland & Anthony, 2003). Ohjelmistoarkkitehdin täytyy pitää huoli siitä, että toteuttamista edeltää keskustelua ehdotetusta suunnitelmasta. Prototyyppiä kehitettäessä suunnitelmien ja niiden arvioinnin ei tarvitse olla muodollisia. Muille kehitysvaiheille Garland ja Anthony ehdottavat, että teknisiä tapaamisia, vertaisarviointeja, suunnitelmien arviointeja ja tarkistuksia suoritettaisiin riittävä määrä ennen varsinaista teknistä toteutusta. Varsin usein kehitysprojekteissa arkkitehdille kerrotaan, ettei tarkastuksessa huomattua suunnitteluvirhettä voida enää korjata, koska ohjelmakoodi on jo kirjoitettu. Arkkitehdin tulisi pitää säännöllisiä kokouksia teknisen tiimin kanssa, jotta ohjelmointi ei pääsisi suunnittelun edelle.

2.4 Suunnitteluelementtien ja niiden suhteiden tunnistaminen

Tässä vaiheessa muodostetaan alustava osiinjako, jossa järjestelmä jaetaan toiminnallisten vaatimusten perusteella (Albin, 2003). Nykyiset ohjelmointikielet eivät tarjoa johdonmukaista rakennejoukkoa, jossa rakenteita voitaisiin pitää arkkitehtuurisina

elementteinä. Siksi sovelluksen fyysinen arkkitehtuurinäkymä on tyypillisesti looginen esitys, jota ei voida kääntää. Tutkijat ovat kuitenkin kehittämässä *arkkitehtuurisia kuvauskieliä* (ADL, architectural description language), joilla tämä olisi mahdollista. Arkkitehtuuriset kuvauskielet ovat formaaleja kieliä ohjelmistoarkkitehtuurien kuvaamiseen, analysoimiseen ja vertailuun (Garland & Anthony, 2003).

Esittämällä arkkitehtuuri suunnitteluelementtien suhteen saadaan malli, joka auttaa löytämään moduuleja kytkevät vuorovaikutussuhteet ja saamaan aikaan suunnittelusäännöt, jotka määrittävät moduulien väliset rajapinnat. UML-pakettikaaviota (ISO, 2005) (kuva 2.1) on mahdollista käyttää, mutta suunnitteluelementtien lukumäärän kasvaessa kaavioista tulee vaikeasti tulkittava.



Kuva 2.1: UML-pakettikaavio (Albin, 2003)

Suurempien informaatiomäärien kuvaamiseen matriisi on parempi vaihtoehto. Osiinjakaminen voidaan mallintaa käyttämällä *suunnittelurakennematriisia* (design structure matrix), joka esittää suunnitteluelementtien riippuvuudet kuvaamatta elementtien rakeisuutta. Tämän mallin luonnos voidaan luoda esisuunnitteluvaiheessa ja parantaa myöhemmissä vaiheissa. Taulukossa 2.1 on esitetty kuvan 2.1 pakettikaaviota vastaava esitys suunnittelurakennematriisina.

Taulukko 2.1: Suunnittelurakennematriisi (Albin, 2003)

	Content	**Content Creation**	**Content Creation HCI**	**Content Publishing**	**Content Publishing HCI**	**Content Delivery**	**Content Delivery HCI**
Content	**O**	**X**		**X**		**X**	
Content Creation	**X**	**O**	**X**				
Content Creation HCI	**X**	**X**	**O**				
Content Publishing	**X**	**X**		**O**	**X**		
Content Publishing HCI	**X**			**X**	**O**		
Content Delivery	**X**			**X**		**O**	**X**
Content Delivery HCI	**X**					**X**	**O**

Suunnittelurakennematriisia tulkitaan seuraavalla tavalla. Elementtien riippuvuuksia merkitään X:llä. Matriisin diagonaali on merkitty O:lla pelkästään tarkastelun helpottamiseksi, sillä on tarpeetonta todeta että elementti riippuu itsestään. Alla olevassa esimerkissä (taulukko 2.1) Content-elementti on riippuvainen elementeistä Content

Creation, Content Publishing ja Content Delivery. Jokainen elementti riippuu Content-elementistä. Siis jokainen Content-elementtiin vaikuttava suunnittelupäätös vaikuttaa jokaiseen muuhun elementtiin.

2.5 Evaluointi

Arkkitehtuurisuunnitelman evaluoinnissa päätetään toteuttaako suunnitelma arkkitehtuuriset vaatimukset (Albin, 2003). Suunnitelma arvioidaan, jotta saadaan kvalitatiivisia mittoja ja kvantitatiivista dataa, joista molemmat ovat metriikoita. Kutakin metriikkaa verrataan laatuattribuuttien vaatimuksiin. Jos mitattu tai havainnoitu laatuattribuutti ei täytä vaatimuksia, uusi suunnitelma on laadittava. Arkkitehtuurisuunnitelmia voidaan evaluoida usealla eri tavalla. Kuitenkin, mitä vähemmän muodollinen arkkitehtuurisuunnitelmamalli on, sitä vähemmän tarkkoja tulokset ovat. Arkkitehtuurisuunnitelmaa arvioidakseen laatuattribuuttivaatimukset on oltava selvästi kuvattu. ”Järjestelmän täytyy olla nopea” tai ”järjestelmän tulee olla helposti muunneltavissa ja sopeutua asiakkaan tarpeisiin” eivät ole riittävän selkeästi sanottu. Tällaisia toteamuksia voidaan käyttää alustavissa vaatimusluetteloissa, mutta ne eivät ole tarpeeksi tarkkoja suunnitelman evaluointiin.

Vaatimukset arkkitehtuurin arvioinniksi täytyy määritellä tarkasti laatuattribuutteina ja niiden hyväksyttävinä arvoina. Useimmat laatuattribuuteista eivät ole luonteeltaan kvantitatiivisia, joten niiden arvot eivät ole numeerisia. Suorituskyky (performance) -laatuattribuutti esitetään yleensä jonain numeerisena alueena tai joukkona ihannearvoja. Muunneltavuus (modifiability) -vaatimusta voidaan käsitellä skenaariona nimeltä muutostapaus. Muutostapaus kuvaa tarkalleen mikä on muuttumassa ja millä tavalla. Tämän attribuutin hyväksyttävä arvo voi olla, että muutos on mahdollinen ilman useamman kuin yhden moduulin uudelleen tekemistä.

Arkkitehtuurin arviointi voidaan aloittaa heti kun ensimmäinen luonnos arkkitehtuurista on valmis (Koskimies & Mikkonen, 2005). Tässä vaiheessa arviointiin ei kannata uhrata

paljon resursseja, sillä luonnos on tavallisesti suhteellisen epätarkka. Tavoitteena onkin analysoida vaatimuksia arkkitehtuurin kannalta. Tulosten perusteella vaatimuksia voidaan priorisoida ja mahdottomista luopua kokonaan. Kun arkkitehtuurin suunnittelu on saatu valmiiksi, voidaan suorittaa täydellisempi arviointi. Arvioinnin tuloksena arkkitehtuuri saattaa muuttua, joten arkkitehtuurista riippuvien ratkaisujen toteuttaminen voidaan aloittaa vasta arvioinnin päätyttyä. Mikäli aikataulu pakottaa aloittamaan toteuttamisen, voidaan arkkitehtuuria arvioida myös valmiista järjestelmästä. Tällä menettelyllä voidaan ainakin lisätä organisaation ymmärrystä järjestelmästä.

2.6 Muuntaminen

Muuntamisvaihe suoritetaan evaluoinnin tai vapaamuotoisen arvioinnin jälkeen (Albin, 2003). Jos arkkitehtuurisuunnitelma ei täytä sen laatuattribuuttien vaatimuksia, suunnitelmaa täytyy muuttaa, kunnes vaatimukset täyttyvät. Olemassaolevaa suunnitelmaa muutetaan *suunnittelutoimintojen* (design operators) mukaan. Uusi suunnitelma arvioidaan ja prosessi jatkuu näin, kunnes hyväksyttävä suunnitelma on saatu aikaan. Joskus tehdyt muutokset voi joutua perumaan ja jatkaa uutta suunnittelupolkua pitkin. On myös mahdollista luoda useita vaihtoehtoisia suunnitelmia samaan aikaan.

Arkkitehtuurisuunnitelmaa muunnetaan käyttämällä suunnittelutoimintoja, *tyylejä* (styles) tai *kaavoja* (patterns). Suunnittelutoimintoja on kahdenlaisia: niitä, jotka vaikuttavat moduuliarkkitehtuuriin ja niitä, jotka vaikuttavat komponenttiarkkitehtuuriin. Moduulitoimintoja on kuusi erilaista (Albin, 2003):

- *Jakaminen* (splitting): suunnitelma jaetaan kahteen tai useampaan moduuliin.
- *Korvaaminen* (substituting): korvataan yksi suunnittelumoduuli toisella.
- *Laajentaminen* (augmenting): laajennetaan järjestelmää lisäämällä uusi moduuli.
- *Poistaminen* (excluding): poistetaan moduuli järjestelmästä.
- *Invertointi* (inverting): moduulin kapseloinnin purkaminen uusien rajapintojen luomiseksi.

- *Liittäminen* (porting): liitetään moduuli toiseen järjestelmään.

Vastaavat komponenttiarkkitehtuuriin sovellettavat suunnittelutoiminnot ovat (Albin, 2003):

- *Jakaminen* (decomposition): jaetaan järjestelmä komponentteihin.
- Komponenttien *replikointi* (replication) luotettavuudeen parantamiseksi
- *Pakkaaminen* (compression): yhdistetään kaksi tai useampi komponentti yhdeksi suorituskyvyn parantamiseksi.
- Komponentin *abstrahointi* (abstraction) muunneltavuuden (modifiability) ja soveltuvuuden (adaptability) parantamiseksi.
- *Resurssien jakaminen* (resource sharing), tai yhden komponentti-instanssin jakaminen toisten komponenttien kanssa integroitavuuden (integrability), siirrettävyyden (portability) ja muunneltavuuden (modifiability) parantamiseksi.

Moduuli- ja suunnittelutoiminnot ovat hyvin samanlaisia. Moduulitoiminnot liittyvät järjestelmän moduulinäkymään: ne ovat suunnittelun ja kehityksen yksiköitä, jotka ensisijassa vaikuttavat ei-toiminnallisiin ominaisuuksiin. Suunnittelutoiminnot liittyvät järjestelmän ajonaikaiseen komponenttinäkymään, jotka ovat niitä suorituksen yksiköitä, jotka ensisijassa vaikuttavat toiminnallisiin ominaisuuksiin (Albin, 2003).

3. Suunnittelua avustavat tekniikat

Arkkitehtuurin suunnittelu on luova ongelmanratkaisuprosessi, joka sisältää ratkaisujen löytämisen tai luomisen joukkoon ongelmia. Tätä prosessia voidaan helpottaa joukolla tekniikoita, jotka auttavat ratkaisemaan ohjelmistoprojekteissa toistuvia ongelmakohtia. Tässä luvussa on pyritty käsittelemään keskeisimpiä tekniikoita ja luvun loppupuolella kerrotaan kuinka SOA-arkkitehtuurikehys voidaan toteuttaa käytännössä.

3.1 Monimutkaisuuden hallinta

Monimutkaisuus on yksi suurimmista ongelmista, jotka yritämme ratkaista ohjelmiston kehitystyökaluilla ja menetelmillä (Albin, 2003). Jos monimutkaisuutta ei hallita, tuotettava ohjelmisto voi myöhästyä aikataulustaan, ylittää budjetin, olla heikkolaatuinen tai peruuntua kokonaan. Monimutkaisuuden poistamiseen ei ole helppoja ratkaisuja, mutta sitä voidaan pyrkiä vähentämään tai hallitsemaan. Monimutkaisuutta voidaan mitata elementtien riippuvuuksien määrän suhteen. Mitä enemmän elementtien välisiä riippuvuuksia, sitä monimutkaisempi järjestelmä on.

3.1.1 Monimutkaisuuden ymmärtäminen

Ohjelmistosuunnitelmaa pidetään tavallisesti monimutkaisena, jos se vaatii pitkän kuvauksen (Albin, 2003; Phleeger & Atlee, 2006). Tämä ei välttämättä tarkoita tekstimuotoista kuvausta, vaan pikemminkin abstraktia konseptia, kuten rakenteellisten elementtien välisten suhteiden kuvausta. Mitä enemmän elementtien välisiä riippuvuuksia on, sitä pitempi kuvaus on. On tavallista lisätä ylimääräisiä objektitasoja järjestelmäsuunnitelmaan, jotta tietyt laatuattribuutit saavutetaan. Muunneltavuuden parantamiseksi muutokset täytyy eristää koskemaan mahdollisimman harvoja moduuleita, ihanteellisesti vain yhtä. Soveltuvuuden (adaptability) parantamiseksi lisätään abstraktioita, jotka mahdollistavat moduulien korvaamisen muilla moduuleilla. Uusien elementtien lisääminen voi

aiheuttaa lisää riippuvuuksia elementtien välille. Kuitenkin huolellisella suunnittelulla ja periaatteiden, kuten abstrahointi, noudattamisella elementtejä voidaan lisätä suunnitelmaan ja samalla riippuvuuksien suhteellista määrää voidaan vähentää. Tämä tarkoittaa, että riippuvuuksien määrä voi nousta, mutta suhteessa elementtien määrään se vähenee.

Uuden ongelman tarkastelu voi näyttää aluksi lannistavalta sen laajuuden ja monimutkaisuuden vuoksi (Pfleeger & Atlee, 2006). Abstrahoinnin avulla pystytään keskittymään ongelman avainalueisiin hukkumatta yksityiskohtiin. Tavallisesti abstrahoinnissa tunnistetaan objektien luokkia, jolloin voidaan yhdistellä asioita helpommin ymmärrettäviin kokonaisuuksiin. Tällä tavoin suunnittelijat voivat keskittyä eri abstraktioiden tasolle kerrallaan sotkeutumatta yksityiskohtiin tai hukkumatta koko järjestelmän laajuuteen. Hyvin valitut abstraktiot lisäävät järjestelmän ymmärrettävyyttä.

Kun arkkitehtuurin monimutkaisuus nousee, yksittäisten moduulien monimutkaisuus voi vähentyä (Albin, 2003). Järjestelmän jakaminen osiin hierarkkisesti mahdollistaa monimutkaisuuden jakamisen useisiin komponentteihin jakamalla suunnittelutyö ja vapauttamalla yksittäisten moduulien suunnittelijat tekemään enemmän suunnittelupäätöksiä vähemmillä riippuvuussuhteilla. Valitsemalla sopiva kuvauskieli järjestelmän näkyvää monimutkaisuutta voidaan hillitä. Järjestelmän monimutkaisuuden vähentämisessä on onnistuttu, mikäli kuvauksen pituus pienenee sisällön merkityksen muuttumatta.

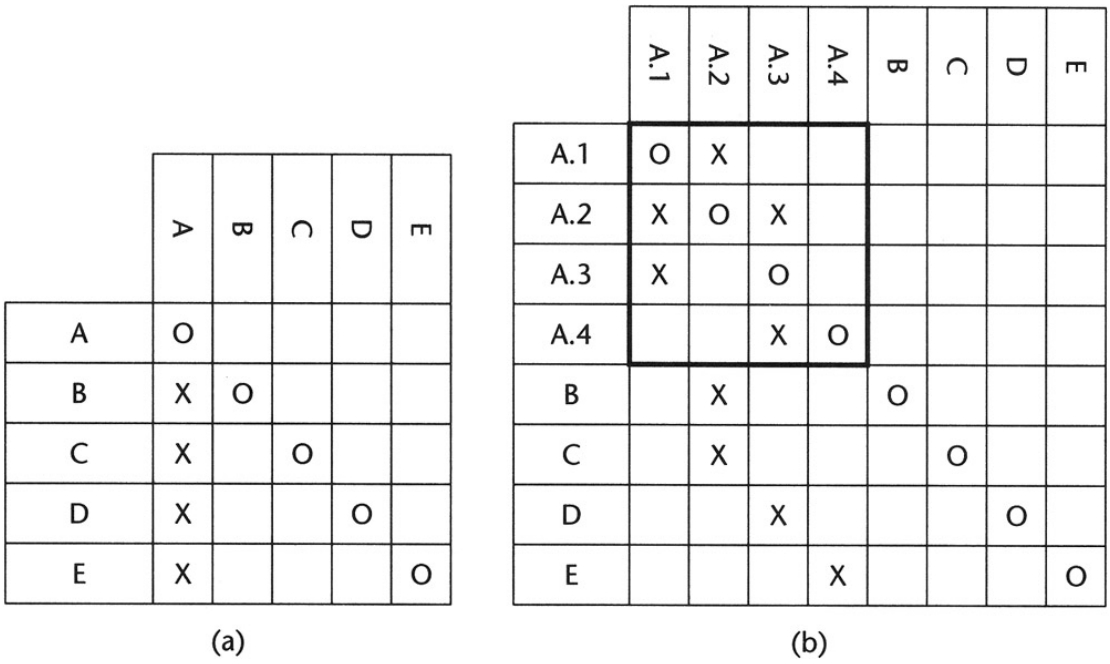
Luonnollisessa kielessä asioiden monimutkaisuutta vähennetään käyttämällä standardia terminologiaa. Yksittäinen sana tai suunnittelumalli (ks. kohta 3.4.2) voi sisältää paljon informaatiota. Suunnittelumalli on tiivis kuvaus yleisesti esiintyvästä suunnittelun ongelma-konteksti-ratkaisu kolmikosta. Suunnitelmien kuvaaminen malleilla auttaa vähentämään näkyvää monimutkaisuutta, koska se lyhentää suunnitelman kuvausta. Näkyvän monimutkaisuuden vähentämistä sanotaan myös järjestelmän ymmärrettävyyden parantamiseksi. Vaikka järjestelmän sisäinen rakenne voi yhä olla monimutkainen, ihmiselle järjestelmä on helpompi ymmärtää.

3.1.2 Rakeisuus ja konteksti

Suunnitelman rakeisuus vaikuttaa monimutkaisuuteen (Albin, 2003). Yksityiskohtien eri tasoilla lukijalle voi olla vaikea hahmottaa kokonaisuus. Tämä pätee myös UML-kaavioihin, sillä kuten mikä tahansa malli, ne jättävät osan yksityiskohdista esittämättä. Komponenttikaavio jättää esittämättä yksityiskohtia yksittäisistä komponenteista ja niiden attribuuteista, metodeista ja toiminnoista muiden komponenttien välillä. Sekvenssikaavio kuvaa olioiden välistä toimintaa, mutta jättää näyttämättä yksityiskohtia olioiden attribuuteista ja joistakin metodeista. Vain välttämättömät metodit toimintojen kuvaamiseen esitetään. Tämä kertoo kuinka joukko olioita kommunikoi, mutta ei kerro muista olioista, jotka eivät liity kuvattuun skenaarioon. Järjestelmän arkkitehtuurin ymmärtäminen katsomalla matalan tason suunnittelumalleja tai lähdekoodia on kuin yrittäisi päättää, mitä kuva esittää katsomalla yksittäisiä kuvapisteitä.

Suunnittelurakennematriisi on järjestelmän sisäisen rakenteen kuvaamiseen tehokas työkalu, jota voidaan käyttää kuvaamaan ohjelmistoarkkitehtuureja, organisaatioita ja liiketoimintaprosesseja. Kuvassa 3.1 (a) on suunnittelurakennematriisi, jossa elementit A, B, C, D ja E kaikki riippuvat elementistä A. Kuvassa 3.1 (b) elementin A sisäinen rakenne on laajennuttu näkyviin. Elementti A koostuu neljästä alielementistä A.1 – A.4, joilla kullakin on omat riippuvuutensa toisistaan. Kuvasta 3.1 (b) selviää myös elementtien B, C, D ja E riippuvuudet elementin A eri osiin ja kaikki muutokset elementtiin A eivät välttämättä vaikuta kaikkiin muihin elementteihin. Mallin rakeisuuden vaihtaminen vaikuttaa siis näkyvään monimutkaisuuteen.

Järjestelmän monimutkaisuus riippuu siis valitusta kontekstista ja tarkastelun kohteesta. Jotta suunnittelijat pystyisivät ymmärtämään järjestelmää paremmin, sen esittämiseen tulee valita järkevä rakeisuus. Osa monimutkaisuuden hallintaa on järjestelmän ymmärrettävyyden parantaminen, vaikka tämä ei varsinaisesti vähennä järjestelmän monimutkaisuutta.



	A	B	C	D	E
A	O				
B	X	O			
C	X		O		
D	X			O	
E	X				O

(a)

	A.1	A.2	A.3	A.4	B	C	D	E
A.1	O	X						
A.2	X	O	X					
A.3	X		O					
A.4			X	O				
B		X			O			
C		X				O		
D			X				O	
E				X				O

(b)

Kuva 3.1: Mallin rakeisuuden vaihtaminen (Albin, 2003)

3.2 Modulaarisuus

Järjestelmän luomisessa täytyy päättää joukosta komponentteja ja komponenttien välisistä rajapinnoista, jotka yhdessä toteuttavat vaaditun joukon vaatimuksia (DeMarco, 1982). Jokainen suunnittelumenetelmä sisältää jonkinlaisen osiinjaon, joka alkaa järjestelmän korkean tason avainelementtien kuvauksesta tarkentuen matalamman tason näkymiin järjestelmän ominaisuuksien ja funktioiden yhteensovittamisesta. Osiinjako jakaa suunnitelman osiin joita kutsutaan moduuleiksi tai komponenteiksi (Pfleeger & Atlee, 2006). Järjestelmää sanotaan modulaariseksi, kun kukin järjestelmän toiminto suoritetaan täsmälleen yhdessä moduulissa ja kunkin moduulin syötteet ja tulosteet ovat tarkasti määriteltyjä. Moduuli on tarkasti määritelty, mikäli sen kaikki syötteet ovat oleellisia moduulin toiminnalle ja kaikki tulosteet ovat moduulin toiminnon tuottamia. Jos yksi syöte puuttuisi, moduuli ei pystyisi suorittamaan toimintaansa. Hyvin määriteltyyn sisältyy myös ehto, että moduuli ei saa turhia syötteitä.

Modulaarisuus on ensisijainen periaate, jolla hallitaan suunnitelmien monimutkaisuutta ja suunnittelutehtäviä tunnistamalla ja eristämällä ne yhteydet tai suhteet, jotka ovat monimutkaisimpia (Albin, 2003). Moduulien voidaan katsoa koostuvan yksityisestä piilotetusta tiedosta ja julkisesta näkyvästä tiedosta. Moduulit ottavat vastaan muiden moduulien julkaisemaa tietoa. Moduulien voidaan ajatella koostuvan suunnitteluelementeistä. Suunnitteluelementti on sekä joukko suunnittelupäätöksiä että vastaava joukko suunnittelutehtäviä. Suunnittelusäännöt ovat julkisia näkyviä suunnitteluelementtejä. Järjestelmän osiin jakaminen voidaan esittää suunnittelusääntöjen ja suunnitteluelementtien hierarkiana. Moduulitoiminnot ovat toimintoja, joita käytetään muuntamaan suunnitelma toiseksi suunnitelmaksi. Moduulitoimintojen lisääminen voi joko tuottaa uusia suunnittelusääntöjä tai poistaa niitä. Nämä toiminnot voivat auttaa yksinkertaisemman suunnitelman saavuttamisessa.

3.2.1 Modulaarisuus ja abstraktiotasot

Modulaarisuutta pidetään hyvän suunnitelman ominaisuutena (Pfleeger & Atlee, 2006). Jokaisen moduulin ollessa tarkasti rajattu on mahdollista tutkia kutakin moduulia erikseen, täyttääkö se halutut toiminnot. Lisäksi osiinjaossa moduulit on jaettu hierarkkisesti tasoihin, joten järjestelmää voidaan tarkastella myös taso kerrallaan. Näistä syistä ohjelmistot pyritään rakentamaan mahdollisimman modulaarisiksi.

Suunnitelman osiinjaossa yhden tason komponentit tarkentavat ylemmän tason komponentteja. Hierarkiassa alaspäin siirryttäessä komponenttien yksityiskohdat tarkentuvat, siis ylin taso on kaikkein abstraktein ja komponentit ovat järjestetty abstraktiotasoittain. Tarkastelemalla suunnitelmaa ylhäältä alas korkeamman tason ongelmat voidaan ratkaista ensin ja niiden ratkaisut laajentaa matalimmille tasoille sitä mukaa, kun ratkaisun yksityiskohdat selviävät. Komponentit piilottavat yksityiskohdat toiminnastaan muilta komponenteilta. Yksittäisen komponentin suunnittelupäätöksen muuttuessa suunnitelma kokonaisuudessaan säilyy koskemattomana, kun vain yhden komponentin sisäistä rakennetta muutetaan.

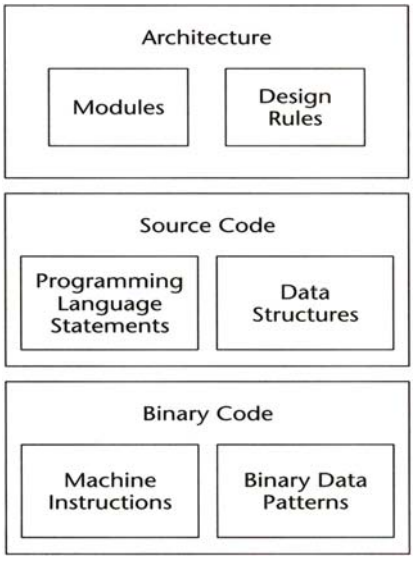
Modulaariset komponentit ja abstraktiotasot tarjoavat useita erilaisia näkymiä järjestelmästä. Korkeimman tason komponentit tarjoavat yleisnäkymän ratkaisusta, piilottaen yksityiskohdat, jotka voisivat vaikeuttaa ymmärtämistä. Esimerkiksi olio-pohjaisessa järjestelmässä tämä näkymä antaisi kuvan abstrakteista tyypeistä ja kuinka järjestelmän oliot ovat suhteessa toisiinsa ilman tarvetta tarkastella jokaista ilmentymää. Kun järjestelmästä tarvitaan yksityiskohtaisempia näkymiä, voidaan siirtyä hierarkian matalimmille tasoille.

Mahdollisuus siirtyä abstraktiotasojen välillä tarjoaa joustavuutta järjestelmän ymmärtämiseen, helppoutta tietovirtojen ja toimintojen jäljittämiseen sekä moni-mutkaisten osien selvittämiseen. Ongelman jakaminen osiin ei sinänsä tee moni-mutkaisesta ongelmasta yksinkertaista, mutta se mahdollistaa vaikeittein osien eristämisen ja käsittelemisen pienemmissä ja siten ymmärrettävimmissä osissa. Mod-ulaarisuuden etuna on myös mahdollisuus suunnitella eri komponentit eri tavoin.

3.2.2 Arkkitehtuuri ja moduulit

Albinin (2003) mukaan ohjelmistosovelluksen fyysistä näkymää voidaan tarkastella kor-keamman tason komponenttijoukkona eli ohjelmistomoduuleina. Nykyiset ohjel-mointikielet eivät tarjoa yhdenmukaisia rakennelmia, joita voitaisiin pitää arkkiteh-tuurisina komponentteina. Tästä syystä ohjelmistosovelluksen fyysinen arkkitehtuurinen näkymä on tyypillisesti looginen esitys, jota ei voida kääntää (kuva 3.2).

Järjestelmän suunnittelunäkökulmia on kahdenlaisia, jotka liittyvät toisiinsa. Ensim-mäinen on staattinen suunnittelutason näkymä järjestelmän hierarkkisesta rakenteesta moduuleineen. Toinen näkymä on sovelluksen dynaaminen eli ajonaikainen kom-ponenttinäkymä. Komponentilla tarkoitetaan tässä ajonaikaista entiteettiä, kuten oliota, oliokokoelmaa, tietokantapalvelinta tai web-palvelinta. Moduuli on diskreetti ohjel-mistopaketti, kuten esimerkiksi Java-kirjasto, relaatiotietokanta tai suoritettavat ja ajon-aikaiset kirjastot.



Kuva 3.2 Ohjelmisto komponenttihierarkiana (Albin, 2003)

Moduuli vie (export) ja ottaa vastaan eli tuo (import) tietoa. Moduuli voi tuoda vain mitä toinen moduuli vie. Nämä viennit ja tuonnit voidaan käsittää abstrakteina rajapintoina. Rajapinta moduuliin voi olla mitä tahansa, kuten tietomuoto (esimerkiksi XML DTD) tai proseduraalinen rajapinta (esimerkiksi joukko julkisia Java-luokkia ja -rajapintoja yhdessä paketissa). Mitä tahansa moduuli vie, muuttuu näkyväksi kaikille muille ulkopuolella oleville moduuleille.

3.2.3 Kytkentä ja koheesio

Kytkennän ja koheesion käsitteet ovat tärkeitä modulaarisen arkkitehtuurin ja järjestelmien monimutkaisuuden ymmärtämiseen (Albin, 2003; Rozanski & Woods, 2005). Kytkentä viittaa kahden moduulin välillä olevien liitäntöjen määrään. Löyhällä kytkennällä tarkoitetaan tilannetta jossa kahdella moduulilla on suhteellisen vähän riippuvuuksia, missä moduulin sisäinen toteutus ei vaikuta sen rajapintaan. Löyhä kytkentä saavutetaan kapseloinnilla eli piilottamalla mahdollisimman monta suunnitteluelementtiä. Löyhästi kytketyt järjestelmät ovat usein helpompia rakentaa ja ylläpitää.

Koheesiolla viitataan riippuvuuksien tiheyteen moduulin sisällä. Korkea koheesio on loogisesti järkevää ja johtaa usein yksinkertaisempaan, vähemmän virhealttiiseen suunnitelmaan.

3.3 Laatuattribuuttien soveltaminen

Järjestelmän laatu on suorassa suhteessa järjestelmän kykyyn tyydyttää sen toiminnalliset ja ei-toiminnalliset vaatimukset (Albin, 2003; Bass & al., 2005). Järjestelmällä on monia piirteitä, kuten toiminnallisuus, tehokkuus ja ylläpidettävyys. Kunkin piirteen laatu muodostaa yhdessä järjestelmän laadun. Kukin piirre voidaan määritellä järjestelmän attribuuttina. Mitattavia ja havainnoitavia ominaisuuksia kutsutaan *laatuattribuuteiksi*. Laatuattribuutit eivät ole toisensa poissulkevia; sen sijaan, ne voivat riippua toisistaan. Yhden laatuattribuutin arvo, kuten tehokkuus, voi riippua toisen laatuattribuutin arvosta, kuten muunneltavuudesta. Laatuattribuuttien standardia taksonomiaa kutsutaan *laatu-malliksi*.

Laatumalli toimii kehyksenä järjestelmän määrittelylle ja testaukselle (Albin, 2003). Laatumalli on luotavalta järjestelmältä vaadittujen ominaisuuksien spesifikaatio. Arkkitehdit voivat käyttää valmiiksi dokumentoituja laatumalleja pohjana ja johtaa oman laatumallin halutulle järjestelmälle. Joskus tietyn tyyppisen sovelluksen ei tarvitse toteuttaa kaikkia ominaisuuksia tai laatuattribuutteja, kuten valmiissa laatumallissa. Arkkitehdin tulisi osana vaatimusmäärittelyä valita tietty laatumalli, joka tunnistaa piirteet, laatuattribuutit näille piirteille, sekä metriikat laatuattribuuteille. Arkkitehti voi joutua laatimaan kullekin kehitystyön vaiheelle, kuten arkkitehtuurikuvaukselle tai valmiille suorituskelpoiselle järjestelmälle, oman laatumallinsa. Arkkitehtuurin suunnittelussa sovellettavat laatuattribuutit (Albin, 2003):

- toiminnallisuus
- tehokkuus

- muunneltavuus
- saatavuus ja luotettavuus
- käytettävyys
- siirrettävyys

Toiminnallisuus (functionality) on laatuattribuutti, joka kuvaa järjestelmän kykyä saavuttaa tarkoitus, johon se oli suunniteltu. Järjestelmän suunnitteluun kuuluva osiinjako suunnitteluelementteihin perustuu toiminnallisuus-laatuattribuuttiin. Toiminnallisuus on perusta, jonka mukaan muut laatuattribuutit määritetään, koska toiminnallisuus vaikuttaa osittain järjestelmän osiinjakoon. Toiminnallinen määrittely on dokumentti, jota voidaan käyttää kuvaamaan toiminnallisuuteen liittyviä vaatimuksia. Muut laatuattribuutit yleensä kärsivät, jos arkkitehtuuri perustuu pelkästään toiminnalliseen määrittelyyn.

Tehokkuus (performance) kuvaa järjestelmän nopeutta, jota voidaan mitata jonkin tapahtuman vasteaikana tai tapahtumien määränä tietyn aikajakson sisällä. Algoritmien ja tietorakenteiden valinta ei ole arkkitehtuurinen päätös, mutta suorituksen jakaminen komponenttien kesken ja komponenttien välisen tiedonvaihdon kaavat ovat. Komponenttien välinen kommunikointi on yleensä enemmän aikaa vievä kuin komponenttien sisäinen suoritus. Tehokkuus on osittain järjestelmän komponenttien välisen kommunikoinnin ja vuorovaikutuksen funktio. Albinin (2003) mukaan tehokkuus on ollut johtava laatuattribuutti arkkitehtuurisuunnittelussa muiden attribuuttien kustannuksella.

Muunneltavuus (modifiability) vaikuttaa arkkitehtuurin myöhempään käytettävyyteen. Muunneltavaan arkkitehtuuriin on helppo lisätä ominaisuuksia myöhemmin eli olemassaolevaa sovellusta on kustannuksiltaan halvempi muokata kuin tehdä kokonaan uusi. Muunneltavuutta kutsutaan joskus myös ylläpidettävyydeksi.

Saatavuus (availability) mittaa järjestelmän toiminta-aikaa yleensä mittaamalla aikaa, joka järjestelmällä menee virhetilanteista toipumiseen. *Luotettavuus* (reliability) liittyy

läheisesti saatavuuteen. Sillä mitataan järjestelmän kykyä toimia ilman virhetilanteita. Arkkitehtuuri vaikuttaa molempiin attribuutteihin.

Käytettävyys (usability) viittaa järjestelmän käytettävyyteen loppukäyttäjien, ylläpitäjien ja mahdollisten muiden käyttäjien kannalta. Loppukäyttäjät ovat yleensä kiinnostuneita toiminnallisuudesta, luotettavuudesta, käytettävyydestä ja tehokkuudesta. Ylläpitäjiä kiinnostaa ylläpidettävyys.

Siirrettävyys (portability) on kyky käyttää komponenttia eri sovelluksissa tai käyttöympäristöissä, kuten laitteisto, käyttöjärjestelmät, tietokannat ja sovelluspalvelimet. Siirrettävyyttä voidaan pitää muunneltavuuden erikoistapauksena. Siirrettävyyden mittari perustuu siihen, kuinka paikallisia muutokset ovat. Ihannearvona on yksi komponentti. Kuten muunneltavuus, siirrettävyys on kustannusten mittari ajan ja rahan suhteen järjestelmän siirtämiseksi toiseen ympäristöön.

3.4 Arkkitehtuuristen kaavojen soveltaminen

Historiallisesti ohjelmistoteollisuus ei ole ollut hyvä oppimaan tekemisistään (Rozanski & Woods, 2005; Koskimies & Mikkonen, 2005). Ohjelmistosuunnittelijat eivät usein käytä olemassa olevia, toimiviksi todettuja ratkaisuja, vaan sen sijaan kehittävät omat ratkaisunsa monimutkaisiin ongelmiin. Samaa voidaan sanoa ohjelmistoarkkitehdeistä päätyessään luomaan uusia suunnitelmia järjestelmiin, jotka sisältävät jo entuudestaan tunnettuja haasteita.

Yksi syy tähän tilanteeseen oli helposti saatavien standardoitujen ohjelmistoarkkitehtuuri- ja suunnitteluongelmien ratkaisujen puuttuminen. 1990-luvulla tähän ongelmaan alettiin hakemaan muutosta. Ohjelmistokaavojen edelläkävijät alkoivat tunnistamaan ja tallentamaan laajasti käytettyjä ratkaisuja tyypillisiin suunnitteluongelmiin. Tämä työ on johtanut siihen, että nykyään on saatavilla kasvava joukko erilaisia kaavoja.

Rozanskin ja Woodsin (2005) mukaan arkkitehtuuriset kaavat jaetaan yleensä kolmeen ryhmään: *arkkitehtuuriset tyylit* (architectural styles), *suunnittelumallit* (design patterns) ja *ohjelmahahmot* (language idioms). Kaikki kolme kaavatyyppiä voivat olla hyödyllisiä, vaikka niitä käytetäänkin järjestelmän kehityksen eri vaiheissa.

3.4.1 Arkkitehtuuriset tyylit

Ensimmäinen näkymä luotavasta järjestelmästä on yleensä yksi komponentti, joka tuottaa joukon käyttötapauksia, jotka esittävät järjestelmän vaadittuja toimintoja ja ovat vuorovaikutuksessa muitten ulkoisten entiteettien kanssa, kuten käyttäjät ja muut ohjelmistojärjestelmät (Albin, 2003). Tämä komponentti jaetaan pienempiin osiin ominaisuudet säilyttäen, mutta lisäten rakennetta järjestelmään, jotta muut ei-toiminnalliset laatuattribuutit saavutetaan. Järjestelmä voidaan jakaa osiin suunnittelu-toimijoiden avulla, mutta prosessi voi olla pelkästään niitä käyttäen hidas. Suunnittelua voidaan nopeuttaa käyttämällä hyväksi ohjelmistotuotantoyhteisön jo olemassa olevaa kokemusta.

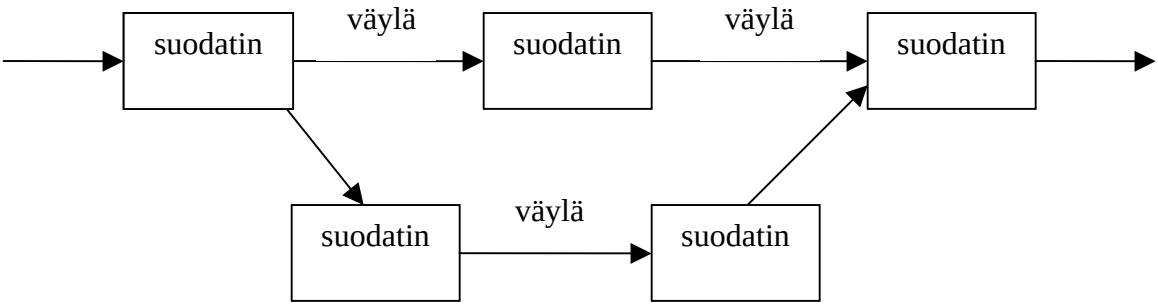
Arkkitehtuurisia tyylejä käytetään usein varsinaisen järjestelmän hahmottamisessa (Koskimies & Mikkonen, 2005). Arkkitehtuurinen tyyli määrittää joukon tietyn tyyppisiä arkkitehtuurisia elementtejä ja liittimiä sekä sääntöjä kuinka, elementit ja liittimet toimivat yhdessä. Samaa tyyliä voidaan soveltaa useissa eri ongelma-alueissa, kuten terveydenhuoltojärjestelmissä tai sisällön julkaisusovelluksissa (Albin, 2003). Samaa arkkitehtuurista tyyliä käyttävät sovellukset noudattavat samoja rakenteellisia periaatteita. Seuraavissa alakohdissa tarkastellaan tavallisimpia arkkitehtuurityylejä.

3.4.1.1 Tietovuoarkkitehtuuri

Tietovuoarkkitehtuurissa (pipes and filters) elementeillä on joukko syötteitä ja tulosteita (Garlan & Shaw, 1994). Elementti lukee tietovirtoja syötteinä ja tuottaa tietovirtoja tulosteina. Tietoa käsittelevät elementit käsittelevät syötteitään itsenäisesti ja aloittavat

tulostuksen ennen kuin syötevirta katkeaa; tästä johtuu elementin toimintaa kuvaava termi *suodatin*. Tietovirtoja suodattimesta toiseen johtavia liittimiä kutsutaan *väyliksi* (kuva 3.3).

Tietovuoarkkitehtuuri auttaa suunnittelijaa ymmärtämään järjestelmän käyttäytymistä syötteiden ja tulosteiden suhteen. Monimutkaisten järjestelmien tiedon prosessointi voidaan tehdä askel kerrallaan, lisäten toimintaa tarkentavia elementtejä asteittain. Koskimies ja Mikkonen (2005) mainitsevat esimerkkinä ohjelmointikielen kääntäjän: lähdekoodin kääntäminen suoraan konekieleksi olisi liian monimutkainen ymmärrettäväksi yhdessä askeleessa, joten käännösoperaatio suoritetaan useammassa vaiheessa. Tietovuoarkkitehtuurin etuna on myös helppo uudelleenkäytettävyys: suodattimia voidaan liittää toisiinsa sillä ehdolla, että suodattimet ymmärtävät toistensa syötevirtoja. Järjestelmää voidaan myös parantaa lisäämällä uusia suodattimia ja korvaamalla vanhoja kehittyneemmillä versioilla. Tietovuoarkkitehtuuri tukee myös rinnakkaista suoritusta, sillä suodatin voidaan toteuttaa toimimaan samanaikaisesti muiden suodattimien kanssa.



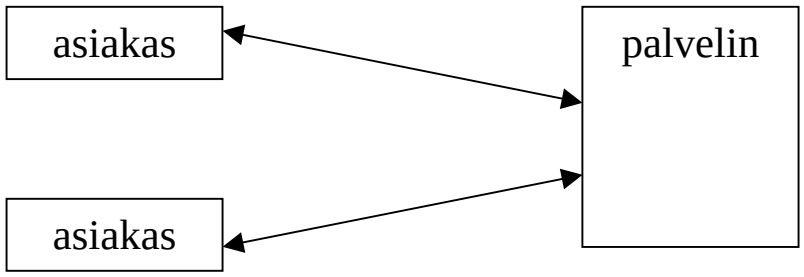
Kuva 3.3: Tietovuoarkkitehtuuri

3.4.1.2 Asiakas-palvelin-arkkitehtuuri

Asiakas-palvelin-arkkitehtuuri (client/server) on laajasti käytetty tyyli, joka määrittelee järjestelmän koostuvan kahdesta elementtityypistä (Rozanski & Woods, 2005): palvelin, joka tarjoaa yhden tai useamman palvelun tarkasti määritellyn rajapinnan kautta ja asiakas, joka käyttää näitä palveluja omien toimintojensa osana (kuva 3.4). Asiakas ja

palvelin tyypillisesti oletetaan sijaitsemaan verkossa eri tietokoneilla. Tämä ei kuitenkaan ole välttämätöntä, asiakas ja palvelin voivat sijaita samassa käyttöjärjestelmäprosessissa. Tämä tyyli on luultavasti tuttu yleisesti käytetyistä tekniikoista, kuten asiakas/palvelin-tietokannat.

Vuorovaikutus asiakkaan ja palvelimen välillä suoritetaan tyypillisesti istunnoissa (Koskimies & Mikkonen, 2005). Palvelin odottaa passiivisena asiakkaan yhteydenottoa. Asiakas esittää pyynnön palvelimelle, joka suorittaa halutut toiminnot ja lähettää tuloksen vastauksena takaisin asiakkaalle. Istunto jatkuu, kunnes asiakas lopettaa sen. Tavallisesti palvelin ottaa vastaan pyyntöjä usealta asiakkaalta samanaikaisesti, joka voi johtaa tyylille kriittisen elementin, palvelimen, kaatumiseen. Palvelimen toimintaa voidaan tehostaa käyttämällä palvelimen sisäisessä toteutuksessa monisäikeisyyttä tai moniprosessointia.



Kuva 3.4: Asiakas-palvelin-arkkitehtuuri

3.4.1.3 Tasoihin perustuva ratkaisu

Asiakas-palvelin-arkkitehtuurista kehitetty tasoihin perustuva ratkaisu (tiered computing) on laajalti käytössä yritystietojärjestelmissä (Rozanski & Woods, 2005). Tähän tyyliin perustuva järjestelmä koostuu useista laskennan tasoista, jotka yhdistyvät tarjoamaan palvelun käyttäjälle. Kukin taso toimii palvelimena kutsujalleen ja asiakkaana arkkitehtuurin seuraavalle tasolle. Tämän ratkaisun avainperiaate on, että taso voi

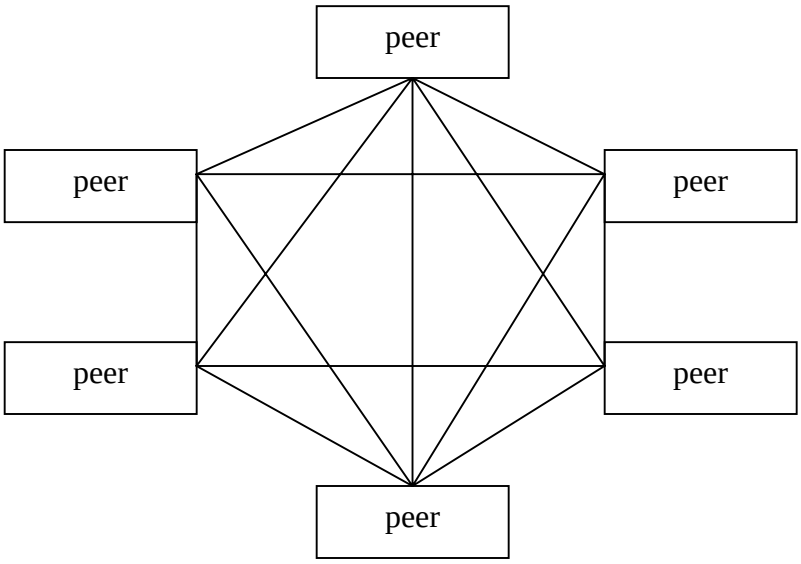
kommunikoida tällä tavalla vain sen vieressä olevien tasojen kanssa. Taso ei ole tietoinen muiden tasojen olemassaolosta naapureitaan lukuunottamatta. Yleensä yritystieto-järjestelmä sisältää seuraavat tasot:

- käyttöliittymä (user interface)
- liiketoiminnan esitys (business presentation)
- liiketoimintatapahtumat (business transactions)
- tiedonsaanti (data access)
- tiedonvarastointi (data storage)

Useat suuret yritystietojärjestelmät on organisoitu tasoihin ja myös muutama tavallinen sovelluskehitystekniikka, kuten J2EE ja .NET.

3.4.1.4 Vertais-arkkitehtuuri

Vertais-arkkitehtuurityyli (peer-to-peer) määrittelee yhden elementtityypin (peer) ja yhden liitäntätyypin (connector) (Rozanski & Woods, 2005). Liitännän ominaisuudet eivät ole tyylille tärkeitä. Tyyliä on käytetty useissa erityyppisissä tietoverkkoyhteyksissä. Järjestelmän keskeinen rakenteinen periaate on, että mikä tahansa elementti on vapaa kommunikoimaan minkä tahansa muun elementin kanssa. Tyypillisesti elementit paikantavat toisensa automaattisesti vaihtamalla listoja tunnetuista elementeistä. Kukin elementti kykenee toimimaan sekä asiakkaana (pyyntöjä tehdessä) että palvelimena (vastatessa pyyntöihin), usein molempina yhtäaikaa (kuva 3.5). Tunnettuja sovelluksia jotka käyttävät vertais-arkkitehtuuria ovat tiedostojenjako-ohjelmat tai hajautetut laskentajärjestelmät.



Kuva 3.5: Vertais-arkkitehtuuri

3.4.1.5 Kerrostettu toteutus

Kerrostetussa arkkitehtuurityylissä (layered implementation) (Rozanski & Woods, 2005) tunnistetaan yksi järjestelmäelementti: kerros. Tyyli rakentaa järjestelmän toteutuksen kerroksien pinoon, jossa kukin kerros tarjoaa palvelun yläpuolellaan sijaitsevalle kerrokselle. Tasot järjestetään niiden esittämän abstraktiotason mukaan. Kaikkein abstraktein sijaitsee pinon päällä ja vähiten abstraktein pinon alimmaisena. Riippuen tyylin toteutuksesta tasot pystyvät kommunikoimaan joko kaikkien muiden alapuolella sijaitsevien kerroksien kanssa tai ainoastaan seuraavan yhden suoraan alapuolella sijaitsevan kerroksen kanssa. Ratkaisua voidaan verrata aiemmin esiteltyyn tasoihin perustavaan tyyliin. Kerrostetussa toteutuksessa kerrokset jaetaan niiden esittämän abstraktiotason mukaan, kun taas tasoihin perustuvassa ratkaisussa tasot jaetaan niiden tarjoaman palvelun mukaan. Tyypillinen esimerkki kerrostetusta toteutuksesta on OSI-viitemalli eli tiedonsiirtoprotokollien esitys seitsemässä kerroksessa (kuva 3.6).

Kuva 3.6: OSI-viitemalli (Bennett et al., 2006)

3.4.1.6 Julkaisija-tilaaja-arkkitehtuuri

Julkaisija-tilaaja (publisher/subscriber) -tyyli (Rozanski & Woods, 2005) kehitettiin, kun havaittiin, ettei asiakas-palvelin ratkaisu sovi kaikkiin hajautettujen järjestelmien ongelmiin. Tyyli määrittelee julkaisija-elementin, joka luo tietoa tilaaja-elementeille, jotka haluavat sitä käyttöönsä. Yhden tyyppinen liitäntä, tietoverkkolinkki, yhdistää julkaisijan tilaajiin. Tilaajat rekisteröivät haluamansa tiedon julkaisijalle, joka vuorostaan luo tai muuttaa sitä ja ilmoittaa muutoksista takaisin tilaajille. Riippuen tyylin toteutuksesta, ilmoitus voi sisältää uutta tai muutettua tietoa, tai se voi olla vain ilmoitus tehdyistä muutoksista jättäen tilaajat tekemään tarvittaessa uuden pyynnön varsinaisista muutetuista tiedoista. Julkaisija-tilaaja- tyyliä käytetään laajalti yrityksien viestijärjestelmissä. Suunnittelumallitasolla tarkkailijamalli (Maciaszek & Liong, 2005; Koskimies & Mikkonen, 2005) noudattaa julkaisija-tilaaja-arkkitehtuuria.

3.4.1.7 Epäsynkroninen tiedon replikointi

Epäsynkronista tiedon replikointia (asynchronous data replication) käytetään, kun halutaan pitää kaksi tietokantaa synkronoituna (Rozanski & Woods, 2005). Tyylillä on kolme elementtityyppiä: tietolähde, tiedon replika ja replikoija. Tietolähde on tietokanta, joka sisältää replikoinnin lähtötiedot, kun taas tietoreplika on erillinen tietokanta, joka sisältää kopioidun tiedon. Replikoija on elementti, joka on vastuussa muutoksien ja lisäyksien tunnistamisesta lähteessä ja synkronoinnin suorittamisesta replikatietokantaan. Tätä tyyliä käytetään yritysten tietokantoja hajautettaessa (Connolly & Begg, 2005).

3.4.1.8 Tietovarastoarkkitehtuuri

Tietovarastoarkkitehtuuri (tuple space) sallii hajautetun järjestelmän osien toimia yhdessä tiedonjakamisessa. Tyyliin kuuluu kaksi elementtityyppiä: asiakkaat ja tietovarasto. Nämä on liitetty toisiinsa asiakas-palvelin-tietoverkossa. Asiakkaat kommunikoivat tietovaraston kanssa joko lisäämällä sinne uutta tietoa tai hakemalla hakuehtoja vastaavaa tietoa. Tyypillisesti tietovarasto voi myös kutsua asiakkaita, kun niiden haluamat tiedot muuttuvat (Rozanski & Woods, 2005; Koskimies & Mikkonen, 2005).

3.4.2 Suunnittelumallit

Suunnittelumallit ovat keskitason arkkitehtuurisia kaavoja (Buschmann & al., 1996). Ne ovat suurempia kuin seuraavassa kohdassa esiteltävät ohjelmahahmot, mutta ne eivät vaikuta ohjelmistojärjestelmän perusrakenteeseen. Sen sijaan suunnittelumalleilla on suuri vaikutus alijärjestelmien arkkitehtuureihin.

Suunnittelumallien jako voidaan suorittaa kategorioihin (Buschmann & al., 1996):

- Rakenteellinen osiinjako (structural decomposition). Sisältää mallit, jotka tukevat alijärjestelmien osiinjakoa ja monimutkaisten komponenttien yhteistoimintaa.

- Työn organisointi (organization of work). Koostuu malleista, jotka määrittävät kuinka komponentit toimivat yhdessä monimutkaisten ongelmien ratkaisemiseksi.
- Pääsyn valvonta (access control). Mallit, jotka valvovat pääsyä palveluihin ja komponentteihin.
- Hallinta (management). Mallit, jotka käsittelevät objektien, palveluiden ja komponenttien homogeenisia kokoelmia kokonaisuudessaan.
- Kommunikointi (communication). Tämän kategorian mallit auttavat organisoimaan komponenttien välistä kommunikointia.

Suunnittelumallien tärkeä ominaisuus on, että ne eivät riipu sovelluksen toimialasta. Ne käsittelevät sovelluksen toiminnan rakennetta, eivät sen toteuttamista. Useimmat suunnittelumallit ovat riippumattomia ohjelmointiparadigmoista. Tavallisesti malleja voidaan toteuttaa helpoiten olio-pohjaisilla kielillä, mutta yleensä niitä on myös mahdollista soveltaa perinteisten, kuten proseduraalisten kielien, yhteydessä.

Suunnittelumallit tarjoavat suunnittelijoille yhteisen sanaston kommunikoimiseen, dokumentoimiseen ja parempien suunnitelmien etsimiseen (Gamma & al., 1995). Suunnittelumallit saavat järjestelmän näyttämään vähemmän monimutkaiselta, kun suunnittelijat voivat keskustella siitä korkeammalla abstraktiotasolla kuin suunnittelunotaatio tai ohjelmointikielet ovat. Suunnittelumallit nostavatkin suunnittelun ja keskustelun tasoa.

Suunnittelumallin keskeiset osat ovat (Koskimies & Mikkonen, 2005):

- Yleinen monenlaisissa järjestelmissä toistuva suunnitteluongelma, joka ei edellytä esimerkiksi tiettyä ohjelmointikieltä.
- Ongelmayhteys, jossa suunnittelumallia voidaan soveltaa ja yhteyteen liittyvät laatuattribuutit, joita ratkaisun tulee parantaa.
- Formaalilla menetelmällä kuvattava yleinen ratkaisu, joka täyttää laatuvaatimukset, mutta saattaa vaikuttaa joihinkin laatuattribuutteihin negatiivisesti.

Suunnittelumallien kuvaaminen tapahtuu dokumenttina, jonka tulisi periaatteessa sisältää kaikki se tieto, mitä mallin soveltaja tarvitsee (Koskimies & Mikkonen, 2005). Suunnittelumallin kuvaustyylit vaihtelevat riippuen lähteistä, mutta ainakin seuraavat asiat tulisi kirjata suunnittelumallin kuvaukseen: nimi, ongelma, ratkaisu, seuraukset, toteutusnäkökulmat ja esimerkit.

Suunnittelumallin *nimi* on olennainen osa kuvausta. Hyvästä nimestä tulee osa suunnittelijoiden käyttämää ammattisanastoa. Samalla suunnittelumallilla ei saisi olla useampaa nimeä, kuitenkin voi olla olemassa idealtaan samanlaisia malleja samoilla tai eri nimillä.

Ongelma, jonka suunnittelumalli ratkaisee, tulisi pyrkiä esittämään riittävän yleisessä muodossa, että se kattaa kaikki mahdolliset mallin sovellukset. *Ongelmayhteys* esitetään usein yhdessä ongelman kanssa. Esimerkkejä voidaan käyttää havainnollistamaan ongelmaa ja ongelmayhteyttä.

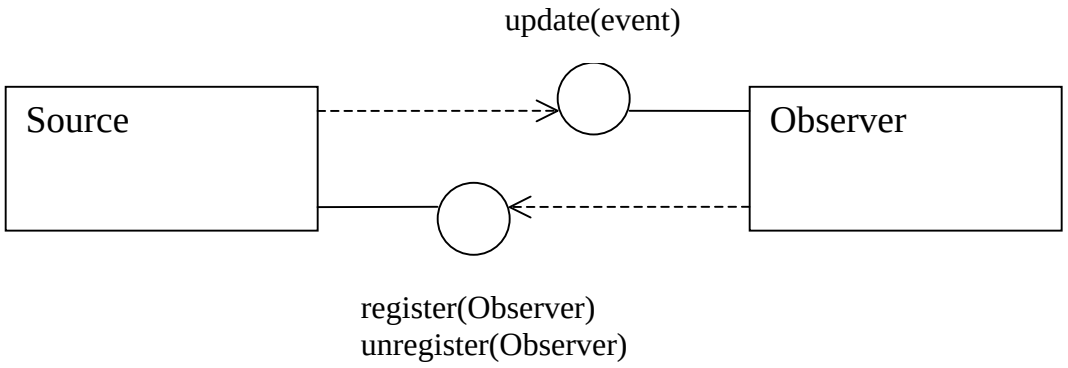
Suunnittelumallin ydin on itse *ratkaisu*, jonka esittämiseen voidaan käyttää esimerkiksi UML:n luokkakaaviota ja sekvenssikaaviota. Mikäli ratkaisussa on algoritmisesti jokin olennainen osa tai kun käyttäytymisen yksityiskohdat ovat syystä tai toisesta olennaisia, voidaan käyttää myös pseudokoodikuvausta.

Suunnittelumallit eivät tavallisesti tarjoa joka suhteessa optimaalista ratkaisua, joten *seurauksien* kuvaus on tarpeen. Suunnittelumallin kuvauksessa on syytä kertoa mihin laatuattribuutteihin malli vaikuttaa positiivisesti ja mihin negatiivisesti. Esimerkiksi muunneltavuus ja suorituskyky ovat vastakkaisia laatuattribuutteja, toisen vahvistuessa toinen heikentyy.

Suunnittelumallin kuvauksessa voidaan myös kertoa *toteutusnäkökulmista*. Mallin toteutus voi vaihdella eri kielissä, joten kuvaus voi sisältää kielikohtaisia vinkkejä toteutustavoista. Suunnittelumalli ei kuitenkaan ota kantaa ohjelmointikieliin, joten looginen ratkaisu ja toteutusasiat on syytä pitää erillään.

Kuvausdokumentissa voidaan antaa myös *esimerkkejä* suunnittelumallin soveltamisesta, jotka voivat olla ohjelmakoodia tai esimerkkejä missä järjestelmissä samaa mallia on käytetty aiemmin.

Usein järjestelmissä muodostuu ongelmaksi tilanne, jossa tieto muuttuu yhdessä osassa ja muut osat ovat riippuvaisia tästä tiedosta. Tarkkailija-suunnittelumalli (Koskimies & Mikkonen, 2005; Sommerville, 2004) soveltuu käytettäväksi juuri tällaisissa tilanteissa. Tämän mallin ratkaisussa lähteestä riippuvat oliot saavat automaattisesti tiedon muutoksista ilman osapuolien tiukkaa kytkentää. Synkroninen tapahtumankäsittelymalli on esitetty kuvassa 3.7 tarkkailija-suunnittelumallina. Tarkkailijat rekisteröityvät lähteelle (source), joka kutsuu tarkkailijoiden tapahtumankäsittelyoperaatiota tapahtuman (event) sattuessa. Tämä operaatio kuuluu takaisinkutsurajapintaan, jonka tarkkailijat toteuttavat. Tapahtuman tiedot voidaan antaa operaation parametrina, esimerkiksi tapahtumaoliona.



Kuva 3.7: Tarkkailija-suunnittelumallin periaate (Koskimies & Mikkonen, 2005)

3.4.3 Ohjelmahahmot

Ohjelmahahmo (language idiom) on matalan tason ohjelmointikielikohtainen kaava, joita kutsutaan myös ohjelmointikaavoiksi. Toisin kuin suunnittelumallit, jotka käsittelevät yleisiä rakenteellisia periaatteita, ohjelmahahmot kuvaavat, kuinka ratkaista toteutus-

ongelmia ohjelmointikielessä (Buschmann & al., 1996). Tiettyjen ohjelmointiongelmien ratkaisuun on usein olemassa erilaisia ratkaisuja. Osa voi käyttää paremmin kielen ominaisuuksia tai noudattaa parempaa ohjelmointityyliä. Ohjelmahahmo voi auttaa ratkaisemaan toistuvia ongelmia kuten muistinhallinta, olioiden luominen, metodien nimeäminen ja lähdekoodin muotoileminen luettavuuden parantamiseksi. Esimerkki ohjelmahahmosta Java-kielellä on esitetty kuvassa 3.1.

```
public synchronized void metodi (....) throws InterruptedException {

  while (odotellaan jotakin resurssia, jonka muut säikeet
         tähän koodiin päästessään voivat antaa)
    wait();

  tehdään kriittiset hommat häiriöttä

  notifyAll();
}
```

Kuva 3.7: Ohjelmahahmo (Wikla, 2006)

3.5 Arkkitehtuuristen kehysten käyttö

Arkkitehtuurinen kehys (architectural framework) on joukko näkökulmamääritelmiä ja niiden välisiä suhteita (Albin, 2003). Kehyksiä käytetään arkkitehtuurikuvauksen laatimisen pohjana. Tässä tarkasteltavia esimerkkejä arkkitehtuurisista kehyksistä ovat: arkkitehtuuriset näkökulmat ja SOA-arkkitehtuurikehys.

3.5.1 Arkkitehtuuriset näkökulmat

Näkökulma (viewpoint) tarkoittaa yleistä, järjestelmästä riippumatonta tapaa kuvata tiettyä arkkitehtuurin kannalta merkityksellistä ohjelmistojen ominaisuutta. *Näkymä* (view) on varsinainen järjestelmästä riippuva kuvaus, joka noudattaa jotakin näkökulmaa (Koskimies & Mikkonen, 2005). Kehykset sisältävät tavallisesti seuraavan tyyppiset näkökulmat (Albin, 2003):

- Suoritus (processing): esimerkiksi toiminnalliset vaatimukset ja käyttötapaukset.
- Informaatio (information): esimerkiksi oliomallit ja tietovirtakaaviot.
- Rakenne (structure): esimerkiksi asiakkaita, palvelimia, sovelluksia ja tietokantoja ja niiden välisiä suhteita kuvaavat komponenttikaaviot.

Arkkitehti päättää onko näkökulma tarpeellinen annetussa ohjelmistokehitysprojektissa. Rakenteellinen näkökulma ei välttämättä ole tarpeellinen, jos fyysinen rakenne on yksinkertainen eikä vaadi kuin lyhyen tekstimuotoisen kuvauksen. Monimutkaisia modulaarisia tai hajautettuja järjestelmiä varten rakenteellinen näkökulma tulee tarpeellisemmaksi.

Unified-prosessi on käyttötapausvetoinen, arkkitehtuurikeskeinen, iteratiivinen ja inkrementaalinen kehys ohjelmistotuotantoprosessiin (Kruchten, 2004). Arkkitehtuurinen 4+1-näkymämalli on Rational Software Corporation kehittämä, joka liitettiin myöhemmin RUP (Rational Unified Process) -kehykseen. Sen tavoitteena on tarjota moninäkökulmainen kehys olio-pohjaisten ohjelmistojärjestelmien määrittämiseen. Tämän 4+1-näkymämallin mukaiset näkökulmat ovat (Albin, 2003: Kruchten, 2004; Bennett & al., 2006):

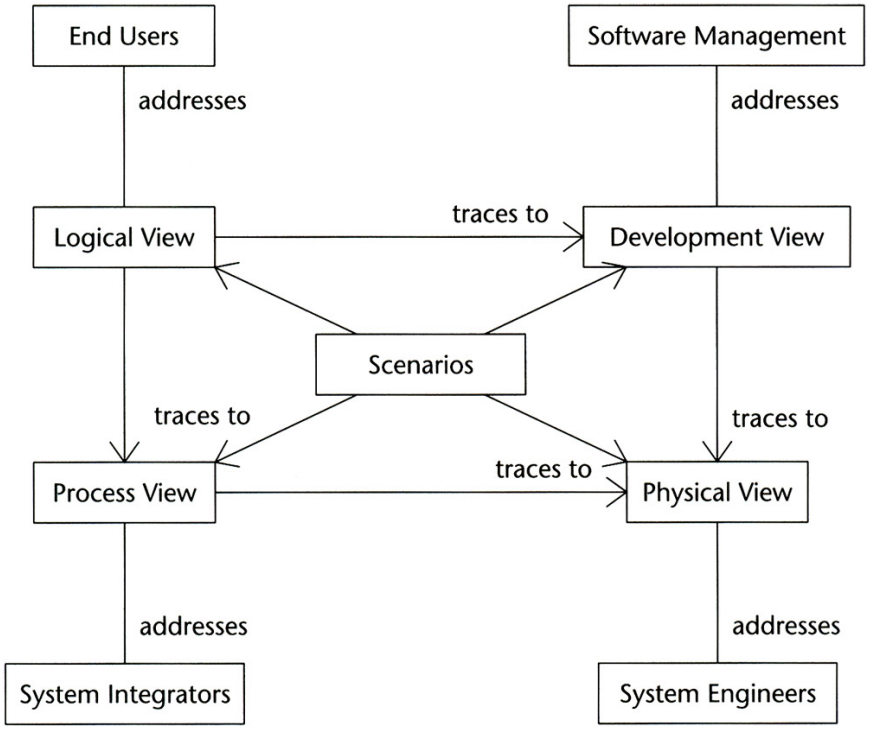
- looginen näkymä
- prosessinäkymä
- toteutusnäkymä
- fyysinen näkymä

- käyttötapausnäkymä

Looginen (logical, design) *näkymä* esittää järjestelmän tärkeät suunnitteluluokat ja rajapinnat UML-pakettirakenteena (ks. kuva 2.1). *Prosessinäkymä* (process view) esittää suunnitteluratkaisut joillekin ei-toiminnallisille vaatimuksille, kuten suorituskyky ja vikasietoisuus. Prosessinäkymä voi näyttää kuinka monta suoritettavaa prosessia järjestelmässä tulee olemaan ja onko samasta prosessista useita ilmentymiä. Prosessien välinen kommunikointitapa vaikuttaa kuinka tehokkaasti tietoa voidaan komponenttien kesken välittää. *Toteutusnäkymä* (development view, implementation view) kuvaa järjestelmän kokoonpanon, kuinka järjestelmä toteutetaan komponentteina. Suuri komponenttien tai alijärjestelmien määrä johtaa kommunikointiin tarvittavan tiedon kasvuun, joten toteutusnäkymän tulisi näyttää pieni määrän prosessiin tarvittavia komponentteja.

Fyysinen (physical, deployment) *näkymä* kuvaa tietoverkon fyysiset solmut komponentteineen ja kommunikointikanavineen. Tämä näkymä kertoo missä eri komponentit on otettu käyttöön, täytyykö tiedon kulkea koneesta toiseen koneeseen vai ovatko kaikki prosessit sijoitettuina samaan paikkaan.

Käyttötapausnäkymä (use case view) ilmaisee arkkitehtuurisesti merkittävän käyttäytymisen ja sitä voidaan käyttää tunnistamaan korkeaa suorituskykyä vaativat käyttötapaukset. Se tunnetaan myös skenaarionäkymänä, jonka tehtävänä on sitoa kaikki muut näkymät yhteen. Skenaariot ovat käyttötapauksien ilmentymiä. Tätä näkymää pidetään ylimääräisenä, josta johtuu ”+1” kehyksen nimessä. Kuvassa 3.8 on esitetty näkymien väliset suhteet.



Kuva 3.8: 4+1-näkymämalli (Albin, 2003)

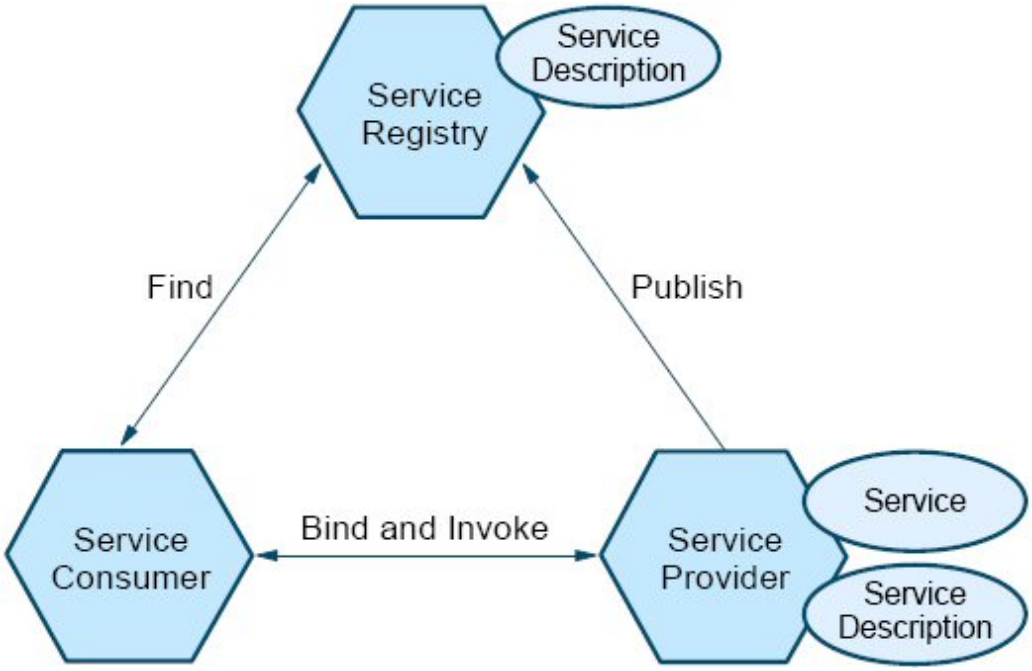
3.5.3 SOA-arkkitehtuurikehys

SOA (Service Oriented Architecture) eli palvelukeskeinen arkkitehtuuri on arkkitehtuurikehys, jossa eri tietojärjestelmien toiminnot ja prosessit on suunniteltu toimimaan itsenäisinä, avoimina ja joustavina palveluina (Erl, 2005; Wikipedia, 2008). Näitä palveluita käytetään avoimien standardien rajapintojen avulla, jolla pyritään aikaansaamaan erilaisten tietojärjestelmien joustava ja järjestelmäriippumaton vuorovaikutus.

SOA-arkkitehtuurissa palveluita voivat käyttää esimerkiksi sovellukset tai toiset palvelut. Toiminta tapahtuu yleensä tietoverkkojen, kuten Internetin, välityksellä avointen rajapintojen ja tekniikoiden avulla. Palvelukeskeisen arkkitehtuurin hyödyntämisellä pyritään saavuttamaan löyhällä kytkennällä entistä avoimempia ja helpommin integroitavia järjestelmiä. Uusien järjestelmien tuottamiseen kuluvaa aikaa ja kustannuksia voitaisiin näin

vähentää, kun uudet sovellukset pystyisivät kommunikoimaan vanhojen, jo olemassa olevien perinnesovellusten kanssa.

SOA-arkkitehtuuri sisältää kuvan 3.9 mukaiset kolme roolia (Endrei & al., 2004). *Palvelun käyttäjä* on sovellus, ohjelmistomoduuli tai toinen palvelu, joka lähettää haun haluamastaan palvelusta *palvelurekisteriin*. Mikäli hakukriteerejä vastaava palvelu löytyi, rekisteri palauttaa rajapinnan *palvelun tarjoajaan* käyttäjälle. Käyttäjä ottaa yhteyden tarjoajaan ja suorittaa haluamansa palvelun. Web-palveluita voidaan käyttää myös suoraan, ilman rekisterihakua, mikäli palvelun käyttäjä tietää tarjoajan web-palvelun osoitteen.



Kuva 3.9: SOA-arkkitehtuuri (Endrei & al., 2004)

3.6 Standardien soveltaminen

Yleisin tapa toteuttaa SOA-arkkitehtuurisovelluksia on web-palvelutekniikka, joka perustuu avoimiin teknologioihin kuten XML, SOAP, UDDI ja WSDL. Tässä kohdassa käsitellään näitä standardeja.

3.6.1 Web-palvelutekniikka

Web-palvelutekniikka (Web Services) muodostuu hajautetusta arkkitehtuurista, jossa useat tietojärjestelmät pyrkivät kommunikoimaan keskenään muodostaakseen yhden järjestelmän. Web-palvelutekniikka on täysin riippumatonta ohjelmointikielistä, käyttöjärjestelmistä ja laitteistoista (Endrei & al., 2004). Web-palvelutekniikka koostuu joukosta avoimia standardeja, joiden avulla sovelluskehittäjät voivat rakentaa web-palveluja. Web-palvelut käyttävät muun muassa seuraavia tekniikoita: XML SOAP, WSDL, UDDI. W3C:n Web Services Architecture Working Group määrittelee web-palvelun seuraavasti (W3C, 2004):

Web-palvelu on URI (Uniform Resource Identifier) -tunnistettava ohjelmisto-sovellus, jonka rajapinnat ja sidonnat voidaan määritellä, kuvata ja löytää XML-artifakteina. Web-palvelu tukee suoraa vuorovaikutusta muiden ohjelmistoagenttien kanssa käyttäen XML-pohjaisia viestejä, joita välitetään Internet-pohjaisten protokollien avulla.

Web-palvelutekniikka sopii hyvin palvelupohjaisten arkkitehtuurien toteuttamiseen. Web-palvelut ovat itsensä määritteleviä (self-describing) ja modulaaria sovelluksia, jotka mahdollistavat liiketoimintalogiikan palveluiksi, joita voidaan julkaista, etsiä ja kutsua Internetissä.

Web-palveluiden julkaisemiseen, paikantamiseen ja suorittamiseen tietoverkoissa käytetään seuraavia standardeja (Endrei & al., 2004):

- SOAP: XML-pohjainen etäproseduurikutsu- ja viestinvälitysprotokolla.
- WSDL: XML-pohjainen web-palvelujen kuvauskieli.
- UDDI: web-palvelurekisteri.

3.6.2 SOAP

SOAP (Simple Object Access Protocol) on XML-pohjaisten viestien välitykseen tarkoitettu protokolla. SOAP koostuu neljästä osasta (W3C, 2001; Snell et al., 2001):

1. SOAP-sanoman kuvaus
2. sovellusten käyttämien tietotyyppien esitysmuotojen koodaussäännöt
3. sopimus, miten etäproseduurikutsut ja etäproseduurikutsujen vastaukset esitetään
4. sopimus, miten SOAP-sanomat sidotaan alla olevaan protokollaan

SOAP-sanoma koostuu kolmesta loogisesta osasta. Varsinainen sanoma on aina pakattu kuoreen (envelope). Viestillä voi lisäksi olla otsikko (header) ja sillä on aina runko-elementti (body). Eri osat tunnistetaan alku- ja lopputunnisteesta puhtaaseen XML-tyyliin. Kuvissa 3.10 ja 3.11 on havainnollistettu SOAP-viestejä.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Kuva 3.10: SOAP-pyyntö (W3Schools, 2008)

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>
</soap:Envelope>
```

Kuva 3.11: SOAP-vastaus (W3Schools, 2008)

3.6.3 WSDL

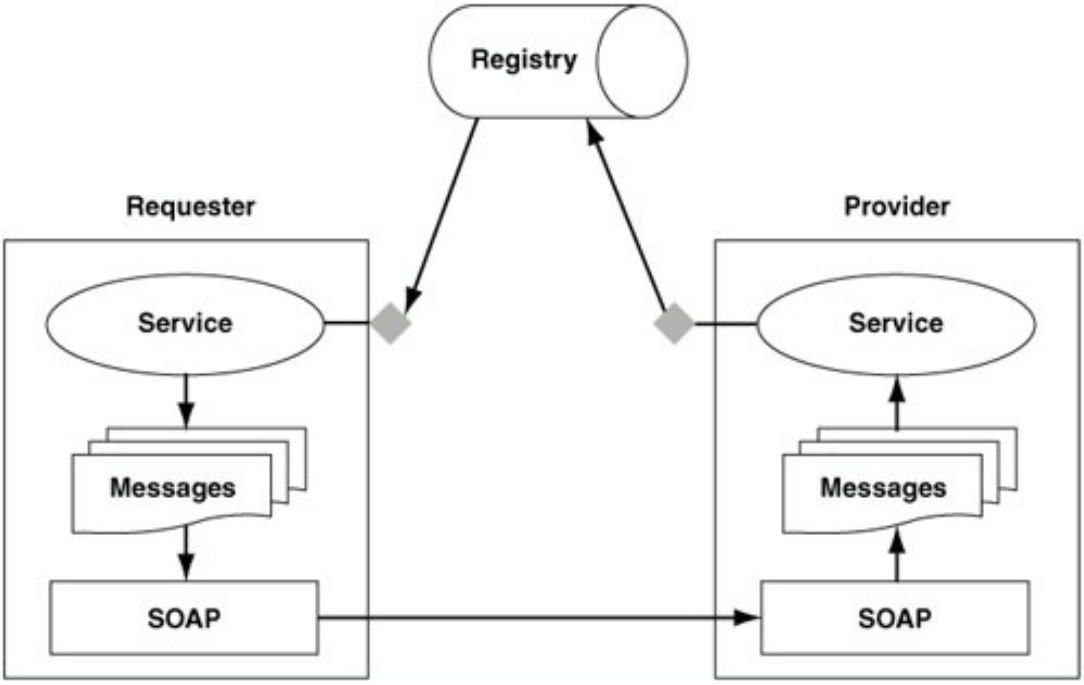
WSDL (Web Services Description Language) on XML-pohjainen web-palvelujen kuvauskieli. Web-palvelun WSDL-määritys koostuu seuraavista komponenteista (W3C, 2001):

- *message*, kuvaa web-palvelun lähettämän tai vastaanottaman sanoman.
- *portType*, ryhmittelee web-palvelun sanomat loogisiksi operaatioiksi (operation). Operaatio koostuu useista, toisiinsa liittyvistä sanomista. PortType-komponentti kuvaa web-palvelun todellisen rajapinnan käyttäjään päin.
- *binding*, kuvaa web-palvelun rajapinnan sidonnan käytettävään tietoliikenneprotokollaan ja tiedon esitystapaan. Siis määrittää sanomien siirrossa käytettävän protokollan ja tiedon formaatin.
- *service*, määrittää web-palveluun kuuluvat päätepisteet (endpoints) eli portit (port), joita voi olla yksi tai useampia. Portti tarkoittaa palveluportin verkko-osoitetta ja palvelun käyttämää tietoliikenneprotokollaa ja tiedon siirtoformaattia.
- *types*, määrittää sanomien käyttämät tietotyypit XML Schema -kielellä ilmaistuna.

Liitteessä 1 on kuvattuna esimerkkinä WSDL-dokumentti.

3.6.4 UDDI

UDDI (Universal Description, Discovery and Integration) on web-palveluiden kuvaus- ja hakemistopalvelu (Newcomer & Lomow, 2005). Tätä rekisteriä voidaan käyttää web-palveluiden tarjoajien ja käyttäjien välisenä varastomekanismina (kuva 3.12). Tarjoaja julkaisee rekisteriin kuvauksen tarjoamastaan palvelusta. Palveluiden hakija voi puolestaan suorittaa haun rekisteriin tallennetun metadatan perusteella. Web-palveluiden metadatan tallentamiseen on olemassa myös monia muita rekistereitä, kuten LDAP-hakemistopalvelut ja CORBA-nimipalvelu.



Kuva 3.12: UDDI-rekisteri (Newcomer & Lomow, 2005)

Alla olevassa kuvan 3.13 esimerkissä kuvataan SOAP-viesti, jolla pyydetään UDDI-rekisteröintiä XYZ.com nimisen yrityksen tiedoille.

```
POST /save_business HTTP/1.1
Host: www.XYZ.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "save_business"
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
      <Body>
         <save_business generic="2.0" xmlns="urn:uddi-org:api_v2">
         <businessKey="">
         </businessKey>
         <name>
         XYZ, Pvt Ltd.
         </name>
         <description>
             Company is involved in giving Stat-of-the-art....
         </description>
         <identifierBag> ... </identifierBag>
         ...
         </save_business>
      </Body>
</Envelope>
```

Kuva 3.13: UDDI-rekisteröinti (TutorialsPoint.com, 2008)

Seuraavassa kuvan 3.14 esimerkissä esitetään SOAP-pyyntö kuvassa 3.12 tallennetun yrityksen tietojen hakemiselle UDDI-rekisteristä.

```
POST /get_businessDetail HTTP/1.1
Host: www.XYZ.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "get_businessDetail"
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
   <Body>
      <get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
          <businessKey="C90D731D-772HSH-4130-9DE3-5303371170C2">
      </businessKey>
      </get_businessDetail>
   </Body>
</Envelope>
```

Kuva 3.14: Tiedon haku UDDI-rekisteristä (TutorialsPoint.com, 2008)

4. Yhteenveto

Ohjelmistoarkkitehtuuri voidaan määritellä järjestelmän perusrakenteena, joka koostuu järjestelmän elementeistä, niiden suhteista toisiinsa ja ympäristöön sekä periaatteista, joiden mukaan järjestelmä tulisi suunnitella. Arkkitehtuurin suunnittelussa järjestelmä jaetaan elementteihin toiminnallisten ja laadullisten vaatimusten toteuttamiseksi. Elementti voi olla yksinkertaisia palveluita tarjoava pieni olion kaltainen yksikkö tai suuri sovelluksen kokoinen yksikkö. Järjestelmän osiinjaon lisäksi suunnitteluprosessi kattaa myös elementtien väliset suhteet ja niiden kehittymisen.

Monimutkaisuus on yksi suurimmista ohjelmistokehityksen ongelmista, jonka seurauksena voi olla kehityskustannusten nousu, projektin viivästyminen aikataulustaan tai jopa peruuntuminen kokonaan. Monimutkaisuutta ei voida poistaa kokonaan, mutta sitä voidaan pyrkiä vähentämään ja hallitsemaan huolellisella suunnittelulla ja periaatteiden, kuten abstrahointi, noudattamisella. Monimutkaisuutta voidaan mitata tarkastelemalla elementtien riippuvuuksien määrää toisistaan.

Ohjelmistoarkkitehtuurin suunnittelu sisältää aina jonkinlaisen järjestelmän osiinjaon. Järjestelmää sanotaan modulaariseksi, kun kukin järjestelmän toiminto suoritetaan täsmälleen yhdessä moduulissa ja kunkin moduulin syötteet ja tulosteet ovat tarkasti määriteltyjä. Modulaarisuus on ensisijainen periaate, jolla hallitaan suunnitelman monimutkaisuutta.

Järjestelmän laatu on suorassa suhteessa sen kykyyn saavuttaa toiminnalliset ja laadulliset vaatimukset. Osa laatuattribuuteista riippuu toisistaan, joten järjestelmää suunniteltaessa täytyy sopia hyväksyttävät arvot kullekin attribuutille, jotka ovat mahdollista saavuttaa. Arkkitehtuuria arvioidessa voidaan soveltaa laatumalleja, jotka toimivat kehyksenä järjestelmän määrittelylle ja testaukselle. Arkkitehti voi joutua kehittämään oman laatumallin kullekin kehitystyön vaiheelle.

Arkkitehtuuristen kaavojen soveltaminen on tekniikka, jolla järjestelmän rakennetta voidaan hahmottaa. Nämä kaavat voidaan jakaa kolmeen kategoriaan: arkkitehtuuriset tyylit, suunnittelumallit ja ohjelmahahmot. Arkkitehtuurisia tyylejä voidaan käyttää hahmottamaan koko järjestelmän rakennetta. Suunnittelumalleja sovelletaan tavallisesti alijärjestelmien kuvaamiseen ja ohjelmahahmot ovat matalan tason ohjelmointikieli-kohtaisia kaavoja.

Arkkitehtuuriset kehykset muodostavat rungon, jonka pohjalta kehystä soveltamalla ja täydentämällä järjestelmä voidaan rakentaa. Kehykset tarjoavat joukon näkökulmamääritelmiä ja niiden välisiä suhteita. Järjestelmän tarkasteleminen tietystä näkökulmasta tarjoaa näkymän, joka mallintaa jonkin arkkitehtuurisesti merkittävän asian järjestelmästä. Useista eri näkökulmista otetut näkymät tarjoavat kattavan, osittain päällekkäisen kuvauksen järjestelmän arkkitehtuurista.

Viitteet

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

Bass, L., Clements, P., Kazman, R. (2005) *Software Architecture in Practice*. Addison-Wesley, USA.

Bennett, S., McRobb, S., Farmer, R. (2006) *Object-Oriented Systems Analysis and Design Using UML*. McGraw-Hill Educations, England.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, England.

Connolly, T.M., Begg, C.E. (2005) *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley, USA.

DeMarco, T. (1982) *Controlling Software Projects*. Yourdon Press, USA.

Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P. Krogdahl, P., Luo, M., Newling, T. (2004) *Patterns: Service-Oriented Architecture and Web Services.* IBM Redbooks, USA.

Erl, T. (2005) *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall, USA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA.

Garlan, D., Shaw, M. (1994) *An Introduction to Software Architecture*. http://www.eelke.com/files/cs709/garlan_introSA.pdf (10.06.2008).

Garland, J., Anthony, R. (2003) *Large-Scale Software Architecture.* John Wiley & Sons, England.

ISO (2005) *Unified Modeling Language Specification, version 1.4.2.* ISO/IEC 19501:2005(E), http://www.omg.org/technology/documents/formal/uml.htm (10.6.2008).

Koskimies, K., Mikkonen, T. (2005) *Ohjelmistoarkkitehtuurit.* Talentum, Suomi.

Kruchten, P. (2004) *The Rational unified process: an introduction*. Addison-Wesley, USA.

Maciaszek, L.A., Liong, B.L. (2005) *Practical Software Engineering: A Case Study Approach*. Addison-Wesley, USA.

McBride, M., (2007) The Software Architect. *Communications of the ACM*, **50**(5), 75-81.

Newcomer, E., Lomow, G. (2005) *Understanding SOA with web-services*. Addison-Wesley, USA.

Pfleeger, S.L., Atlee, J.M. (2006) *Software Engineering: Theory and Practice*. Prentice Hall, USA.

Rozanski, N., Woods, E. (2005) *Software Systems Architecture.* Addison-Wesley, USA.

SEI (2008) *Software Architecture for Software-Intensive Systems, Published Software Architecture Definitions*. Software Engineering Institute, http://www.sei.cmu.edu/architecture/published_definitions.html, (10.06.2008).

Sewell, M., Sewell, L. (2002) *The Software Architect's Profession: An introduction*. Prentice Hall, USA.

Snell, J., Tidwell, D., Kulchenko, P. (2001) *Programming Web Services with SOAP*. O'Reilly, USA.

Sommerville, I. (2004) *Software Engineering*. Addison-Wesley, USA.

TutorialsPoint.com (2008) *UDDI Usage Example*. WWW-sivusto, http://www.tutorialspoint.com/uddi/uddi_usage_example.htm (10.06.2008).

Wikla, A. (2008) *Ohjelmointitekniikka (Java): rinnakkaisuudesta*. WWW-sivusto, http://www.cs.helsinki.fi/u/wikla/OTJ/K06/Sisalto/Saikeet (10.06.2008).

Wikipedia (2008) *Palvelukeskeinen arkkitehtuuri*. WWW-sivusto, http://en.wikipedia.org/wiki/Service-oriented_architecture (10.06.2008).

W3C (2001) *Simple Object Access Protocol (SOAP) 1.1*. WWW-sivusto, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ (10.06.2008).

W3C (2001) Web *Services Description Language (WSDL) 1.1*. WWW-sivusto, http://www.w3.org/TR/wsdl (10.06.2008).

W3C (2004) *Web Services Architecture*. WWW-sivusto, http://www.w3.org/TR/ws-arch/ (10.06.2008).

W3 Schools (2008) *SOAP Example*. WWW-sivusto, http://www.w3schools.com/soap/soap_example.asp (10.06.2008).

Liite 1: WSDL-dokumentti

Lähde: W3C (2001)

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           <element name="TradePrice">
              <complexType>
                  <all>
                      <element name="price" type="float"/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
```

```
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
           <soap:operation
soapAction="http://example.com/GetLastTradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>
```