Ohjelmistoarkkitehtuuri

Jonne Itkonen
Jyväskylän yliopisto, tietotekniikan laitos

20. marraskuuta 2007

1 Johdanto

Tämän luennon tarkoituksena on hieman selventää ohjelmistoarkkitehtuurin monisyistä käsitettä. En lähde tekemään mitään määritelmiä tai luomaan teorioita siitä mitä ohjelmistoarkkitehtuuri on, tai kuinka ohjelmiston arkkitehtuuri tulisi luoda. Annan esimerkkejä parista arkkitehtuurityylistä ja esittelen erään, hieman kyseenalaisen menetelmän arkkitehtuurin luomiseen.

Alussa esitellään hieman ohjelmistotekniikan ja ohjelmistoarkkitehtuurin historiaa. Myös raapaisen (rakennus-)arkkitehtuurin kahden eri teorian pintaa. Näiden kahden osion tulisi antaa näkemys siitä, kuinka laajasta ja monisyisestä, jopa taiteellisesta asiasta on kyse. Alun jälkeen on esitelty hieman erästä tapaa nähdä ohjelmistoarkkitehtuuri. Lopussa on vielä hieman omaa pohdintaani ohjelmistoarkkitehtuurista, sen luonteesta ja merkityksestä ohjelmistotuotannolle.

Yksi alue, joka jää lähes kokonaan pois tutkiskelusta, on ohjelmistoarkkitehtuurien liiketoiminnallinen puoli. Pyydän anteeksi, sillä pidän itseäni enemmän tekevänä ja toteuttavana yksilönä. Uskon siksi, että pelkästään arkkitehtuurien tuoma varmuus tuotteen laadusta ja mittarit sille (!) ovat jo tarpeeksi suuri kontribuutio yhtiön liiketoiminnalliselle yksikölle tavoitteiden saavutettavuuden ja resurssien arviointiin. Myöskään en aio puhua olemassaolevan arkkitehtuurin löytämisestä tai arkkitehtuurikielistä.

Mainittakoon vielä, että vuonna 2005 on ilmestynyt suomenkielinen ohjelmistoarkkitehtuureja käsittelevä kirja '*Ohjelmistoarkkitehtuurit*', jonka ovat kirjoittaneet Tampereen teknisen yliopiston professorit Kai Koskimies ja Tommi Mikkonen [8]. Kun tässä kirjoituksessa vielä arkkitehtuurityyleillä on suuri merkitys, on tuohon kirjaan mennessä tyyleistä tullut lähinnä arkkitehtuurimalleja. Tätä muutosta ei ole vielä huomioitu tässä kirjoituksessa.

2 Historia

Harvemmin kuulee missään käytettävän termiä *ohjelmistotuotanto* englanninkielisessä muodossa *software production* , vaan englanniksi termi on sama kuin suomen *ohjelmistotekniikka*, *software engineering* . Käsite ohjelmistotekniikka on peräisin vuodelta 1968[1], jolloin NATO:n sponsoroimana järjestettiin Saksan Garmischissa konferenssi, jonka nimeksi pitäjät antoivat *Working Conference on Software Engineering* . Nimi oli provokatiivinen, kuten konferenssin raportista [9, s. 8] voidaan lukea:

> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

Tämä konferenssi oli menestys, mutta menestys jäi lyhyeksi. Seuraavana vuonna Italian Roomassa järjestetty jatkokonferenssi ei enää yltänyt ensimmäisen tasolle, ja jäikin sarjan viimeiseksi. Valitettavasti termi *software engineering* vakiintui tarkoittamaan sen hetken vallalla olevia käytäntöjä, eikä siten täyttänyt provokatiivista tarkoitustaan herättää ihmiset tajuamaan moisten käytäntöjen puute.

Konferenssin keskustelujen kuvauksen [9, s. 22] alkupuolelta voimme lukea Peter Naurin kommentin:

> . . . software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention: Christopher Alexander[2]: *Notes on the Synthesis of Form* (Harvard Univ. Press, 1964)

Vuoden 1969 sisältää kylläkin paljon hyvää materiaalia myös, muunmuassa seuraavan herra Sharpin puheenvuoron [12, s. 9]:

> I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focused on it. This is the subject of software architecture. Architecture is different from engineering.
> . . .
> What happens is that specifications of software are regarded as functional specifications. We only talk about what it is we want the program to do. It is my belief that anybody who is responsible for the implementation of a piece of software must specify more than this. He must specify the design, the form; and within that framework programmers or engineers must create something. No engineer or programmer, no programming tools, are going to help us, or help the software business, to make up for a lousy design.
> . . .
> Probably a lot of people have experience of seeing good software, an individual piece of software which is good. And if you examine why it is good, you will probably find that the designer, who may or may not have been the implementer as well, fully understood what he wanted to do and he created the shape.

1. Sana ohjelmisto on vain kolme vuotta vanhempi, mutta jo konferenssin vuonna 1968 puhuttiin *ohjelmistokriisistä*, joskaan kaikki eivät uskoneet sen olemassaoloon.

2. Herra Alexanderin nimen tulisi olla tuttu; häneen törmäämme vielä myöhemminkin.

Siinäpä tuo oli, mitä ohjelmistoarkkitehtuuri on, kaikkinaisuudessaan, lyhykäisyydessään. Nykyisen määritelmän ohjelmistoarkkitehtuurille on antanut Bass et al. lähteessä [2]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Ohjelmistotekniikan alkuajoista kiinnostuneita kehoitan tutustumaan lähteeseen [11], jossa Randell valoittaa vuoden 1968 tilannetta tietotekniikan tutkimuksen ja sovelluksen kannalta, sekä itse konferenssin tunnelmia.

Ajalta ennen ohjelmistoarkkitehtuurin viimeaikaista nousua mainitsen pari paperia: Dijkstran THE-käyttöjärjestelmäpaperin [5] ja Parnasin paperin järjestelmien modularisoinnista [10]. Näistä Dijkstran paperi kuvaa heidän tekemänsä THE-käyttöjärjestelmän rakenteen, joka on nähtävissä vielä monissa nykyisissä käyttöjärjestelmissä. THE:n rakenne on tuttu kerrosrakenne, missä jokainen kerros toimii itsenäisesti laajentaen aina alemman kerroksen tarjoamaa toiminnallisuutta.

Parnasin paperi taas kertoo kahdesta eri modularisointitavasta pienelle ohjelmalle. Ensimmäinen näistä modularisoi ohjelman syötön muotoilun eri vaiheiden mukaan, jälkimmäinen eri toiminnallisten vastuiden mukaan, hierarkkisesti. Vaikka jälkimmäinen tapa voitaneen nähdä hyvin oliomaisena, on olioarkkitehtuuri vanhempaa perua. Olioiden historian katsotaan alkavan vuonna 1967 julkaistusta Simula-67 ohjelmointikielestä, joskin olioita ja oliomaisia tapoja ohjelman kirjoittamiseen on varmaankin ollut olemassa jo ennen sitä.

Nykyinen kiinnostus arkkitehtuuriin vahvistui 1990-luvun alkupuolella. Vuonna 1996 Mary Shaw ja David Garlan julkaisuvat kirjansa '*Software Architecture – Perspectives on an Emerging Discipline* ' [14], joka kerää mielestäni hyvin yhteen tuon ajan näkemyksen. Nykyistä näkemystä pääsee parhaiten tutkailemaan lähteistä [2] ja [3].

Arkkitehtuuri on myös kiinnostanut ohjelmistoalan ihmisiä muilla kuin varsinaisen ohjelmistoarkkitehtuurin saralla. Tästä on hyvänä esimerkkinä nykyinen suunnittelumallivillitys, joka on lähtöisin *neljän koplan* (*Gang of four* , lyhyemmin *GoF*) kirjasta '*Design Patterns – Elements of Reusable Object-Oriented Software* ' [6]. Tämä kirja on saanut innoituksensa arkkitehti Christopher Alexanderin ja kumppaneiden kirjasta '*A Pattern Language* ' [1].

3 Sananen arkkitehtuurista

Arkkitehtuurin tuotoksia saamme katsoa joka päivä ympärillämme, joitakin jopa ihailla. Ymmärrämme, että tarvitsemme arkkitehteja, jotta rakennuksemme näyttävät kauniilta ja ovat käyttökelpoisia – ja rakennusarkkitehteja, jotta ne pysyvät pystyssä. Meitä ohjelmistoarkkitehteina kuitenkin kiinnostaa enemmän se, miten arkkitehdit suunnittelevat ja oppivat suunnittelemaan rakennuksia. Tätä voidaan lähestyä arkkitehtuurin teorian kautta.

Arkkitehtuurin teoriasuuntauksia näyttäisi olevan ainakin kolme. Näistä *teemallisia* ja *synteettisiä* teorioita käsitellään esimerkiksi Pentti Routionin Taideteollisen korkeakoulun verkkopalvelimelta löytyvillä arteologian, taito-opin, sivuilla [13], tarkemmin osoitteessa `http://www2.uiah.fi/projects/metodi/035.htm#begin`. Kolmannelle teorian muodolle en osaa nimeä sanoa, mutta siitä löytyy erittäin hieno oppikirja, Stenrosin ja Auran *Arkkitehtuurin muoto ja sisältö* [15].

Routio käy sivuillaan läpi arkkitehtuurin historiaa ja sen aikana vallinneita tyylisuuntia. Opimme Vitruviuksen keisari Augustuksen aikana aloittaneen teorian kerryttämisen jo sisältäneen pitkälle meidän aikaan säilyneet kolme rakentamisen käytännön tavoitetta: kestävyys, käytännöllisyys ja miellyttävyys. Mielenkiintoinen kannaltamme on esimerkiksi Eugène Viollet-le-Ducin lausahdus vuoden 1863 tietymiltä:

> [Rakennustaiteessa] hyvä maku on yleensä yhtä kuin alitajuisesti aistittu järkevyys.

Hänen luomansa muotokieli on Routionin mukaan ensimmäinen sitten antiikin päivien.

Sivuhuomautus: Edsger W. Dijkstra on kirjoituksessaan [4] todennut seuraavasti:

> After more than 45 years in the field, I am still convinced that in computing, elegance is not a dispensable luxury but a quality that decides between success and failure; in this connection I gratefully quote from The Concise Oxford Dictionary a definition of "elegant", viz. "ingeniously simple and effective". Amen.

Pikakelaus eteenpäin ja toimivan, käytettävän rakentamisen ylilyöntiin, funktionalismiin. Routio siteeraa tässä osassa Aaltoa seuraavasti:

> Aalto kirjoitti 1940 The Technology Review - lehdessä:
> Kuluneen vuosikymmenen aikana moderni arkkitehtuuri on ollut funktionaalista pääasiassa tekniseltä kannalta painopisteen ollessa etupäässä rakennustoiminnan taloudellisella puolella... Mutta koska arkkitehtuuri kattaa koko ihmiselämän alan, todellisen funktionalistisen arkkitehtuurin tulee olla funtionaalista pääasiassa inhimilliseltä kannalta. ... Tekniikka on vain apuneuvo... Tekninen funktionalismi on oikeassa vain jos se laajennetaan käsittämään myös psykofyysistä aluetta. Se on ainoa tapa inhimillistää arkkitehtuuria... (Aalto 1972, siv. 49, 51).

Jotain opittavaa komponenttiohjelmoijille löytynee seuraavasta Routionin kommentista:

> . . . useatkin funkiksen pioneerit yrittivät saada käyntiin myös ihmisten psyykkisten tarpeiden selvittelyä, mutta se käynnistyi perin hitaasti, ehkä siksi, että funktionalistit ymmärsivät tuon tehtävän ylivoimaisen laajuuden. Vasta aivan viime aikoina on alettu ymmärtää, että käyttäjien toiveiden huomioonottamiseksi on muitakin keinoja kuin kysellä niitä laajoilta populaatioilta ja sitten työläästi muokata tuloksista teoreettisia standardeja. Nimenomaan rakennusosien esivalmistus tarjoaa suoremman keinon asiakkaiden kuulemiseen: heille voidaan antaa mahdollisuus osallistua rakennuksen suunnitteluun valitsemalla siihen parhaaksi katsomansa osat esivalmistetusta valikoimasta (ks. Yhteissuunnittelu).. . .

Myöhemmin, arkkitehtonisen synteesin teorian yhteydessä, Routio viittaa yhteissuunnittelun yhteydessä myös laajaan käytön ja viihtyvyystekijöiden tutkimuksen kautta

syntyneeseen suunnitteluoppaaseen [1], joka onkin jo meille tuttu.

Stenros ja Aura taas puhuvat kirjassaan [15] arkkitehtuurista mielestäni enemmän esteettisestä ja rakennusteoreettisesta näkökulmasta. He kertovat, mitä ovat tila, massa ja pinta arkkitehtuurin peruskäsitteinä, kuinka me koemme ne, mikä vaikuttaa niihin ja kuinka ne vaikuttavat lopputulokseen.

4 Ohjelmistoarkkitehtuuri

Monet näkevät ohjelmistoarkkitehtuurin identtisenä ohjelmiston suunnittelun kanssa, mutta olen tästä varsin eri mieltä. Suunnittelu on yksityiskohtaisempaa, toteutuksen suunnittelua. Ehkä parempi suomennos suunnitteluvaihetta tarkoittavalle englannin sanalle *design* olisi *muotoilu*. Arkkitehtuuri taas keskittyy kuvaamaan ohjelmiston ydinrakenteet. Tämä sinänsä abstrakti käsite saattaa olla perin hankala niin kuvailla kuin ymmärtää, jopa niin hankala, etteivät kaikki siihen helposti edes kykene.

Ohjelmistoarkkitehtuuri kuvataan yleensä elementtien (*element,component*) ja näiden välisten liittimien (*connector*) avulla. Koska komponentti-käsite sotkeutui helposti ohjelmistokomponenttiin, tai vielä yleisempään (ja erittäin häilyvään) komponentin käsitteeseen alallamme, on lähteen [2] mukaan siirrytty käyttämään ohjelmistoarkkitehtuurin komponenteista nimitystä elementti.

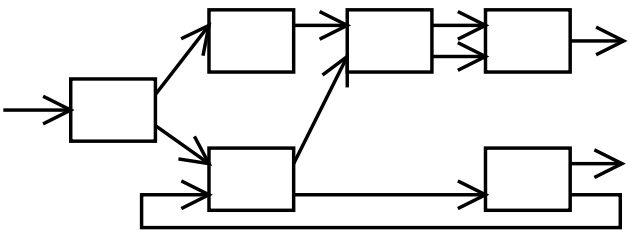
Nämä elementit ovat arkkitehtuurin osia, jotka saattava olla toteutuksessa usean ohjelmistokomponentin toiminnan yhdiste. Esimerkkejä mahdollisista elementeistä ovat olio, asiakas, palvelin, suodin, kerros ja tietokanta. Liittimistä voitaisiin esimerkkinä mainita aliohjelmakutsut, tapahtuman (*event*) välitys, tietokantaprotokollat ja putket.

4.1 Arkkitehtuurityylit

Seuraavana on lueteltu joukko Shaw'n ja Garlanin kirjassaan [14] esittämiä arkkitehtuurityylejä. Tyylejä ei lähdetä tässä analysoimaan sen tarkemmin, tyydyn vain viittaamaan lähteeseen [7].

Putkia ja suodattimia

Elementillä on tiedon sisään- ja ulostulo. Elementti saa tiedon sisääntulosta (syöttö), käsittelee sen, ja laittaa vastauksen ulostuloon (tulostus). Tämä voi tapahtua joko käsitellen ja tulostaen yhden syötön osan kerrallaan, tai ottamalla ensin koko syötön, käsittelemällä tämän kokonaisuutena, ja tulostaen taas yhtenä kokonaisuutena. Tätä voitaneen kuitenkin pitää omana tyylinään. Sekamuotojakin löytyy. Kuva 1 esittää putki-suodatin-arkkitehtuuria.



Kuva 1: Putkia ja suodattimia.

Tätä ideologiaa kannattaa verrata alkuperäiseen Unixin ideaan, joka oli tehdä paljon pieniä ohjelmia, joita yhdistelemällä – toisen tuloksen käyttäminen toisen syöttönä – saadaan ongelma ratkaistuksi tai tieto käsitellyksi. Fyysikot taas muistavat LabView'n, jotkut ovat myös tulleet tutuiksi muiden visuaalisten ohjelmointikielten kanssa, esimerkiksi kuvankäsittelyn yhteydessä. Tuleva mahtisana, *komponenttiohjelmointi*, saa myös tämän arkkitehtuurimallin näyttämään tutulta.

Elementtinä tässä tyylissä toimii suodatin, liittymänä putki.

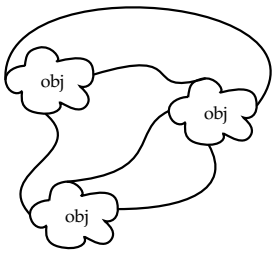
Lisää esimerkkejä: Perinteiset ohjelmointikielten kääntäjät, signaalinkäsittely, funktionaalinen ohjelmointi, hajautetut järjestelmät(?).

Lisää kursseja aiheesta: Unix systeemiohjelmointi C-kielellä, Käyttöjärjestelmät.

Tiedon abstrahointia ja olioita

Sitten tämän ajan lempilapsi – olio-ohjelmointi. Kun 60-luvulla Norjassa havaittiin, että simulointi on huomattavasti mukavampaa, kun simuloinnin kohdetta voi käsitellä kokonaisena yksilönä ohjelmakooditasolla, syntyi olio-ohjelmointi. Tieto ja sitä käsittelevät operaatiot kapseloitiin yhteen. Valaistuttuaan 70-luvun Amerikoissa olio-ohjelmointi saavutti nykyisen tilansa, jollaisena me sen tunnemme. Oliot, joilla on käytös, tila ja identiteetti, käsittelevät tiedon ja ratkaisevat ongelmat keskenään kommunikoimalla. Tärkeitä käsitteitä oli paitsi olion kykyjen siirto toiselle periyttämällä, myös olion tiedon abstrahointi. Tämä johtaa aivan erinlaiseen ajattelu- ja suunnittelutapaan kuin mihin on aiemmin esimerkiksi rakenteisessa ohjelmoinnissa totuttu. Valitettavasti tämä perustavaa laatua oleva erilaisuus ei ole kovin yleiseen tietouteen levinnyt, minkä takia jotkut ovat jo uskaltautuneet väittämään olio-ohjelmoinnin epäonnistuneen.

Elementtejä ovat nyt, yllätys, oliot, liittymiä niiden väliset suhteet. Kuva 2 hahmottaa tätä arkkitehtuurityyliä (nostalgisesti).



Kuva 2: Oliot tekevät yhteistyötä salaten tilansa mustasukkaisesti. Kuvassa käytetty notaatio muistuttaa etäisesti Boochin notaatiota.

Yleensä olio-ohjelmoinnista puhuttaessa kannattaa aina tähdentää siinä vallitsevaa kahtiajakoisuutta järjestelmän kuvauksesessa kehityksen versus ajon aikana. Monesti oliojärjestelmät nähdään vain staattisina luokkahierarkioina, kun monin verroin tärkeämpää olisi nähdä myös ajonaikainen, olioiden muodostama dynaaminen rakenne.

Lisää kursseja aiheesta: Olio-ohjelmointi.

Tapahtumat (implicit invocation)

Ennen Javaa ja oliokieliä varsin yleinen malli graafisten käyttöliittymien ohjelmoinnissa oli tapahtumakeskeinen malli. Kun nappulaa kuvaruudulla painettiin, lähti siitä viesti viestikanavaan, josta halukkaat sen kuulivat ja siihen reagoivat. Vaikka ovat päällisin puolin käyttöliittymäohjelmoinnista hävinneet, eivät tapahtumamallit silti ole

kokonaan kadonneet, pikemminkin päinvastoin! Nykyinen web-palvelujen huuma ja vanhat kunnon hajautetut järjestelmät ovat pitäneet, ja tulevat pitämään, tämän mallin hengissä vielä pitkään.

Yksi huomattava piirre tapahtumapohjaisissa järjestelmissä on, ettei tapahtuman aiheuttaja tiedä kuka reagoi tapahtumaan. Olio-ohjelmoinnissahan tämä on melkein päinvastoin, olion viite täytyy olla tiedossa, jotta jotain olion metodia voidaan kutsua.

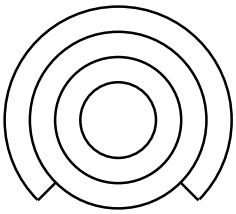
Elementtejä ovat nyt moduulit, joissa on kasapäin proseduureja ja tapahtumia.

Lisää kursseja aiheesta: (vanha?) Graafisten käyttöliittymien ohjelmointi.

Kerrosjärjestelmä

Kun kirjastojen tekeminen tuli muotiin, syntyivät kerrosjärjestelmät. Kerros A_1 hoitaa tehtävän alimmalla tasolla, kerros A_2 työvälineitä lisäten, ja niin ylöspäin iteroiden aina kerrokseen A_n, jota käyttääkin asiakas. Puhtaassa kerrosjärjestelmässä kommunikoivat vain välittömät naapurikerrokset, epäpuhtaampia toteutuksiakin on.

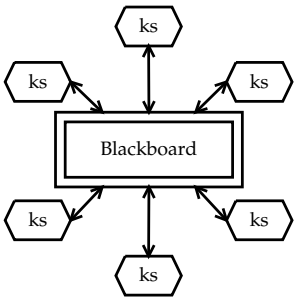
Kirjastojen lisäksi hyviä esimerkkejä kerrosjärjestelmistä ovat käyttöjärjestelmät [5] ja kaikkien suosikki Java-arkkitehtuuri, alimmalla tasollaan virtuaalikone, jonka päälle on useita luokkakirjastoja lastattu. Kuva 3 syöpyy helposti lähtemättömästi päähän.



Kuva 3: Kerroksittainen rakenne kuin sipulilla.

Lisää kursseja aiheesta: Käyttöjärjestelmät, Tietoliikenne.

Tietovarastot

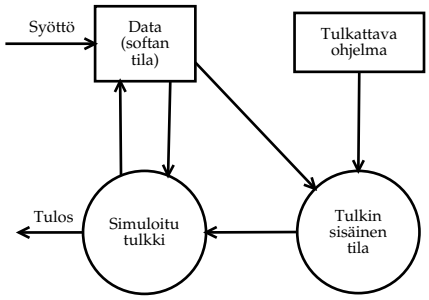


Kuva 4: Liitutauluko?

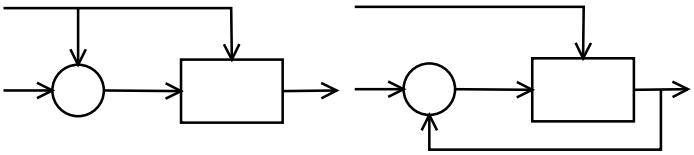
Tietovarastoissa (*repository*), kuten kuvasta 4 nähdään, on pääosassa kaksi elementtityyppiä: Tietovarastoelementti ja sitä käyttävät tietolähteet tai tiedon käsittelijät. Mallissa tietovarasto toimii yhteisenä muistina tietoa käsitteleville ja käyttäville elementeille. Taulun ja käsittelijöiden väliseen kommunikoinnin toteuttamiseen ei juurikaan oteta kantaa.

Tällaisen arkkitehtuurin järjestelmiä ovat esimerkiksi jotkut puheentunnistusjärjestelmät, tietokannat ja uudemmat ohjelmointikielten kääntäjät.

Lisää kursseja aiheesta: Tekoäly??, Tietokannat?.



Kuva 5: Tulkki



Kuva 6: Kontrolloitu prosessi (laatikko) ja kontrolloija (ympyrä).

Tulkit

Java, riittääkö? Rakennetaan tulkki, joka suorittaa eli tulkitsee ohjelmaa, ja jolla on oma sisäinen tila, sekä ohjelman tila. Java-teknologia on hyvä esimerkki tästä tyylistä, ja tämän hetken suosikkina varmaan monien mielessä. Wanhat parrat voivat muistella vanhoja Pascal-toteutuksia (UCSD-Pascal ja P-code).

Lisää kursseja aiheesta: Ohjelmointikielten periaatteet.

Prosessit ja kontrollointi

Teräksen jatkuvavalu lienee monille tuttu esimerkkinä prosesseista ja niiden kontrolloinnista. Ideana tässä arkkitehtuurityylissä on, että syöttötiedosta lasketaan ohjausparametrit, joilla elementtiä ohjataan. Myös takaisinkytkentä tuloksista ohjaukseen on mahdollinen.

Lisää kursseja aiheesta: Simuloinnin ja mallintamisen kurssit.

Muita tuttavuuksia

Muita tuttuja tyylejä ovat esimerkiksi hajautetut järjestelmät, kuten CORBA ja Javan RMI ja Jini, unohtamatta perinteistä asiakas-palvelin-mallia. Myös vieläkin tutumpi jako pääohjelmaan ja aliohjelmiin voidaan nähdä arkkitehtuurityylinä, vieläpä perin yleisenä. Tarjolla on myös sovellusalueelle räätälöityjä viitearkkitehtuureja, unohtamatta tilaperustaisia ratkaisuja (tilakoneet ja muut).

Lisää kursseja aiheesta: Hajautetut järjestelmät, Automaatit ja kieliopit.

Heterogeeniset arkkitehtuurit

Arkkitehtuureita voi yhdistellä monella tavalla. Yhden tyylin elementit voidaan esimerkiksi toteuttaa toisen tyylin avulla, vaikkapa Unixissa ohjelmat, joita putkitetaan, voivat olla oliosuuntautuneita. Tämä ei ole ainoastaan elementtien yksinoikeus, vaan liitokset voivat myös olla toisen arkkitehtuurityylin tuotteita.

Toinen tapa on sallia elementille eri tyylien mukaisia liitoksia. Kolmas on tehdä arkkitehtuurikuvauksen eri tasot eri tyyleillä.

4.2 Arkkitehtuurin suunnittelu

Yleensä sovelluksia tehdessä ei laadullisiin vaatimuksiin juurikaan kiinnitetä huomiota. Ne täyttyvät joko sattumalta, muun osaamisen johdosta, tai ne ovat sovellusalueelle niin olennaisia, että ne täyttyvät huomaamatta.

On myös erityisiä tutkimusaloja, jotka ovat keskittyneet laadullisiin vaatimuksiin, valitettavasti usein vain yhteen kerrallaan. Sovelluksen toimintaa pyritään tehostamaan esimerkiksi suorituskyvyn tai varmuuden kannalta. Näiden vaatimusten yhdistämistä ei auta se, että usein vaatimukset ovat toisilleen epäsuotuisia.

Laadullisia vaatimuksia koetetaan kyllä arvioida hieman sovelluksen valmistuttua, mutta tällöin muutosten tekeminen, varsinkin, jos ne vaikuttavat vahvasti sovelluksen arkkitehtuuriin, on perin kallista. Tämän takia olisi myös hyvä kiinnittää huomiota laadullisiin vaatimuksiin kehityksen alkuvaiheessa, jolloin muutoskustannukset ovat yleisesti pienempiä.

Tässä esitelty metodi perustuu kolmeen toteutettuun järjestelmään. Järjestelmät on paremmin esitelty lähteessä [3, Luku kolme eteenpäin].

Vaikka tämä arkkitehtuurin suunnitteluprosessi voidaan nähdä funktiona, jonka syötteinä ovat vaatimukset ja tuloksena arkkitehtuurikuvaus, ei prosessi suinkaan ole automaattinen, vaan vaatii ohjelmistoarkkitehdeilta panostusta ja luovuutta. Metodin kolme vaihetta ovat: arkkitehtuurin suunnittelu toiminnallisuuden perusteella, laatuatribuuttien varmistus ja arkkitehtuurin muuntaminen tarvittaessa.

Ensimmäisen suunnitellun arkkitehtuurin avulla testataan, täyttyvätkö laatuvaatimukset. Jos ne eivät täyty, muokataan arkkitehtuuria ja testataan laatuvaatimukset uudestaan. Ei ole mielekästä testata kaikkia vaatimuksia kerrallaan, vaan vaatimukset on hyvä jakaa mielekkäiksi osajoukoiksi.

Arkkitehtuurin suunnittelu toiminnallisuuden perusteella

Prosessi, joka on esitelty kuvassa 7 sivulla 6 alkaa arkkitehtuurin laatimisella toiminnallisista vaatimuksista. Tämän ensimmäisen vaiheen tarkoituksena on löytää järjestelmän ydinkäsitteet, joiden mukaan järjestelmä rakentuu. Näitä yleistyksen kohteita ei useinkaan löydy suoraan sovellusalueesta, vaan niiden löytymiseen tarvitaan hieman luovuutta ja analysointia. Kun yleistykset ovat löytyneet, tarkennetaan niiden välisten toimintojen kuvauksia.

Laatuatribuuttien varmistus

Arkkitehtuurin ensimmäisen version suunnittelun – ja jokaisen arkkitehtuurin muuntamisen – jälkeen arvioidaan täyttyvätkö kaikki laadulliset vaatimukset, jotka ovat kyseisellä kierroksella tarkastelun kohteena. Jokainen vaadeatribuutti arvioidaan kvalitatiivisesti tai kvantitatiivisesti, ja näitä arvioita verrataan vaatimuksissa esitettyihin arvoihin. Jos kaikki arviot ovat yhtähyviä tai parempia kuin vaatimukset, siirrytään tutkimaan seuraavaa vaatimusjoukkoa.

Laatuatribuuttien varmistamiseen on tarjolla neljä eri vaihtoehtoa. Voidaan käyttää skenaarioiden arviointia, simulaatiota, matemaattista mallintamista tai järkeilyä.

Skenaarioita tarvitaan kaksi, toinen suunnittelua ja toinen arviointia varten. Ajamalla näitä skenaarioita läpi saadaan arvio laatuatribuuttien täyttymisestä. Esimerkiksi jos muutosskenaariot aiheuttavat suuria muutoksia arkkitehtuuriin, on järjestelmän ylläpidettävyys heikkoa.

Simuloitaessa toteutetaan ensin arkkitehtuurin pääelementit, jäljelle jääneitä elementtejä simuloidaan ohjelmallisesti. Toinen vaihtoehto on tehdä prototyyppi, jolloin arkkitehtuuri toteutetaan osittain, ja tätä osittaista toteutusta ajetaan oikeassa ympäristössä.

Vaihtoehtona simuloinnille on tarjolla järjestelmän *matemaattinen mallintaminen.* Tätä käytetään varsinkin toiminnollisten laatuatribuuttien arviointiin. Matemaattista mallintamista voi myös käyttää simuloinnin apuna, sillä täysin toisensa poissulkevia ne eivät ole.

Järkeilyä ei tule aliarvioidan yhtenä laatuatribuuttien arviointimenetelmänä, vaikka se on kovin subjektiivinen tapa arvioida. Kokemus on valttia näissäkin asioissa.

$$f_i(t) = a(x), x \in \forall [1, 2, \ldots n] \quad (1)$$

Arkkitehtuurin muuntaminen

Jos jokin arvio edellä jää alle vaaditun tason, muunnetaan arkkitehtuuria ja arvioidaan vaadeatribuutit uudelleen. Tätä toistetaan, kunnes kaikki laatuvaatimukset on täytetty, tai tehtävä huomataan mahdottomaksi, jolloin vaatimuksista täytyy neuvotella uudestaan asiakkaan kanssa.

Arkkitehtuuri voi parantaa neljällä eri tapaa: sovittamalla arkkitehtuuri johonkin tunnettuu arkkitehtuurityyliin, sovittamalla arkkitehtuuriin jokin arkkitehtuurimalli, sovittamalla osaan arkkitehtuuria jokin suunnittelumalli tai muokkaamalla vaatimuksia.

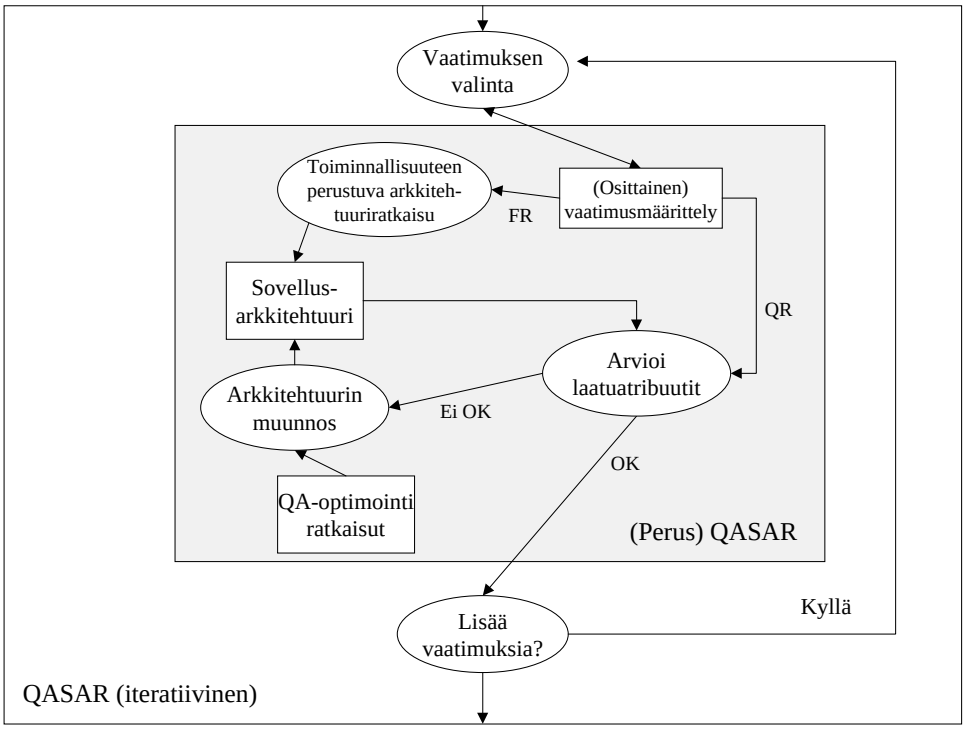
Arkkitehtuurityylit ovat koko arkkitehtuuria hallitsevia, joskin niitä voi yhdistellä hieman. Arkkitehtuurimallit ovat keskenään ja arkkitehtuurityylin kanssa ortogonaalisia, ja siten pikemminkin verrattavissa aspektisuuntautuneeseen ohjelmointitapaan kuin suunnittelumalleihin. Arkkitehtuurimallit muokkaavat koko arkkitehtuuria tai suurta osaa siitä, kun taas haluttaessa muokata vain pientä osaa arkkitehtuurista voidaan käyttää suunnittelumalleja.

Laatuvaatimuksen voi myös sopivissa tilanteissa muuntaa toiminnalliseksi, josta esimerkkinä Bosch mainitsee poikkeutukset vikasietoisuuden parantajina. Viimeinen keino on vanha tuttu: hajoita ja hallitse. Tällöin vaatimus hajoitetaan useammaksi laadulliseksi vaatimukseksi järjestelmän elementeille, tai kahdeksi tai useammaksi toiminnalliseksi vaatimukseksi.

5 Yhteenveto

Ohjelmistoarkkitehtuurille on valtavirrassa käymässä niin kuin monelle muullekin hyvälle asialle on käynyt: se vaikuttaa samankaltaiselta jonkin vanhan, tutumman asian kanssa, joten se tähän samaistetaan. Näin on käynyt esimerkiksi olio-ohjelmoinnille ennen ohjelmistoarkkitehtuureja. Mielenkiintoista kyllä, esitelty Boschin arkkitehtuurinsuunnittelumenetelmä muistuttaa kovasti oliomenetelmiä, vaikkakin vastaavia menetelmiä löytyy jo 1960-luvun lähteistä.

Enemmän kuin oppimoniste, on tämän paperin tarkoitus olla ajatuksia herättävä ja varsinkin tiedonjanoisille lähteitä tarjoava. Tässä paperissa mainitut lähteet ovat minulle olleet tärkeitä, niin hyvässä kuin hieman huonommassakin



Kuva 7: Boschin QASAR-malli [3].

mielessä. Pentti Roution verkkojulkaisu on ollut minulle hyvin tärkeä tiedonlähde, ja olen huomannut monien sen taito-opin havaintojen soveltuvan hyvin ohjelmistojen tekemisen taidon ymmärtämiseen. Muistutanpa heti samaan hengenvetoon, etteivät nämä kaksi alaa silti ole välttämättä identtiset. Bosch ja Bass ovat kaksi lähdettä, joiden esitykseen en oikein osaa samaistua. Kuitenkin, Bosch tuo mielestäni hyvin esille, kuinka arkkitehtuureilla tuntuu olevan ohjelmiston laatua parantava vaikutus. Voisin jopa väittää, että ilman arkkitehtuuria ei ohjelmisto voi olla laadukas. Bass taas on mannaa heille, jotka haluavat yksityiskohtaista kuvausta metodeista arkkitehtuurien suunnitteluun ja dokumentointiin.

Toivon kuitenkin, että innostut tutustumaan näihin lähteisiin. Varsinkin NATO:n konferenssien paperit ovat erittäin tervettä luettavaa jokaiselle, joka luulee alan kehittyneen vasta 1980- ja 1990-luvulla. Nuo vuosikymmenet alkavat näyttää yhä pahemmalta pysähtymisen aikakaudelta.

Mitä tulee ohjelmistoarkkitehtuuriin, pitäkää silmät, korvat ja mieli avoimena. Ohjelmistoalan työntekijät tekevät sovelluksia useimmiten muille kuin omalle alalleen, joten muiden alojen tuntemus on välttämätöntä. Minkä toivon tällä kirjoituksellani ja luennollani osoittaneeni, ovat muut alat myös hyvin tärkeä lähde meille yrittäessämme ymmärtää omaa alaamme.

Viitteet

[1] C. Alexander et al.: *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[2] Len Bass, Paul Clements ja Rick Kazman: *Software architecture in practice*, Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN 0-321-15495-9.

[3] Jan Bosch: *Design and use of software architectures: adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., 2000, ISBN 0-201-67494-7.

[4] Edsger W. Dijkstra: "Computing science: Achievements and challenges", .
URL `http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1284.PDF`

[5] Edsger W. Dijkstra: "The structure of the "the"-multiprogramming system", teoksessa "Proceedings of the first ACM symposium on Operating System Principles", (ss. 10.1–10.6), ACM Press, 1967.

[6] Erich Gamma et al.: *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2.

[7] David Garlan ja Mary Shaw: *An Introduction to Software Architecture*, Tekninen Raportti CMU-CS-94-166, Carnegie Mellon University, January 1994.
URL `http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html`

[8] Kai Koskimies ja Tommi Mikkonen: *Ohjelmistoarkkitehtuurit*, Talentum Media Oy, 2005, ISBN 952-14-0862-6.

[9] P. Naur ja B. Randell (toim.): *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*, Scientific Affairs Division, NATO, Brussels, 1969.
URL `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html`

[10] D. L. Parnas: "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, 15(12), ss. 1053–1058, 1972, ISSN 0001-0782.
URL `http://doi.acm.org/10.1145/361598.361623`

[11] B. Randell: "Software engineering in 1968", teoksessa "Proceedings of the 4th international conference on Software engineering", (ss. 1–10), IEEE Press, 1979, ISBN none.

[12] B. Randell ja J.N. Buxton (toim.): *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969*, Scientific Affairs Division, NATO, Brussels, 1970.
URL `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html`

[13] Pentti Routio: *Tuote ja tieto*, Taideteollinen korkeakoulu, Helsinki, 1994, nykyään ylläpidetty verkkojulkai-

su.
URL http://www2.uiah.fi/projects/metodi/

[14] Mary Shaw ja David Garlan: *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, Inc., 1996, ISBN 0-13-182957-2.

[15] Helmer Stenros ja Seppo Aura: *Arkkitehtuurin muoto ja sisältö*, Rakennuskirja, Helsinki, 1984, ISBN 951-682-105-7.