SAIMAAN AMMATTIKORKEAKOULU
Tekniikka Lappeenranta
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Juha-Matti Seppänen

OHJELMISTOARKKITEHTUURIT

Opinnäytetyö 2010

TIIVISTELMÄ

Juha-Matti Seppänen
Ohjelmistoarkkitehtuurit, 40 sivua, 2 liitettä
Saimaan ammattikorkeakoulu
Tekniikan yksikkö, tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Opinnäytetyö 2010
Ohjaaja: Lehtori Ylä-Jussila Martti

Tutkimuksen tavoitteena on tutkia nykyaikaisia ohjelmistoarkkitehtuureja sekä niiden suunnittelu- ja toteutusmenetelmiä sekä tuottaa tutkimustuloksista ammattikorkeakoulun opetukseen soveltuvaa opetusmateriaalia.

Tutkimuksen lähtökohtana on ollut tietojärjestelmien arkkitehtuurisuunnittelun merkityksen kasvu yritysten tietojärjestelmäsuunnittelussa ja ohjelmistotuotannossa. Syinä tähän kasvuun ovat mm. yritysten sisäisten tietojärjestelmien ja ohjelmistojen kasvu ja monimutkaistuminen sekä eri yritysten toiminnan ja tietojärjestelmien integroituminen, ohjelmistotuotannon globalisoituminen ja komponenttiohjelmointi. Ohjelmistotuotannon globalisoitumisen seurauksena standardien ohjelmakomponenttien tuotanto on siirretty halvan työvoiman maihin, jolloin kalliisiin ja kehittyneisiin teollisuusmaihin on jäänyt vain tietojärjestelmien suunnittelu ja sovittaminen yritysten ja yhteiskunnan toimintaan.

Tutkimus on tehty lukemalla alan tieteellisiä julkaisuja, kirjallisuutta sekä Internet-lähteitä ja tutustumalla yliopistojen ja ammattikorkeakoulujen tarjoamaan materiaaliin aiheesta.

Tutkimuksen tuloksista luotiin opintomateriaali asiakkaan järjestämälle Ohjelmistoarkkitehtuurit -opintojaksolle. Yksityiskohtaiset tutkimustulokset ovat tämän raportin liitteessä 1, joka on opiskelijoille tarkoitettu opiskelumateriaali.

Asiasanat: ohjelmistoarkkitehtuuri, ohjelmistotuotanto, ohjelmistosuunnittelu

ABSTRACT

Juha-Matti Seppänen
Software architectures, 40 pages, 2 appendices
Saimaa University of Applied Sciences
Technology, Information Technology
Software Engineering
Final year Project 2010
Instructor: Ylä-Jussila Martti

The purpose of this research is to study modern software architectures and the techniques of designing and developing software architectures. The results of the research were used to make teaching material for the school.

The starting point for this research has been the ever growing need for better software architectures. Reasons for this need are integration of systems between different companies, making software from all ready existing components, making these components in countries with low labor costs, which means that more expensive countries are stuck with only designing and fitting systems to companies and in overall the fact, that the modern society is more and more dependent on informationsystems.

The Research was conducted by studying articles, literature, internet sources and also studying materials of other schools about software architectures.

The results of the research were used to make teaching materials to the client's course of "Ohjelmistoarkkitehtuurit". More specific results are found in the first appendix, which is the material meant for students.

Keywords: software architecture, software production, software design

SISÄLTÖ

TERMIT JA LYHENTEET

Abstrakti	Abstraktinen, käsitteellinen (vastakohtana konkreettiselle), ei suoranaisesti havaittavissa.
Alijärjestelmä	Osajärjestelmä, pienempi kokonaisuus järjestelmässä.
API	Application programming interface. Ohjelmointirajapinta, jolla eri ohjelmat voivat tehdä pyyntöjä ja vaihtaa tietoja eli keskustella keskenään. Esimerkiksi ohjelmat tarvitsevat käyttöliittymältä luvan käyttää keskusmuistia ja tiedostoja.
Arkkitehtuuriviipale	Järjestelmän pääositusperusteen kanssa eriävä mutta jonkin kriteerin perusteella loogisesti yhteenkuuluva rakenne.
Artefakti	Ihmisen valmistama esine, aine, rakenne tai tekotuote. Ohjelmistotuotannossa artefaktilla tarkoitetaan esim. dokumentteja, luokkakaavioita, yms.
Komponentti	Itsenäisesti toimiva ohjelmistoarkkitehtuurin osa, jolla on hyvin määritellyt rajapinnat.
Käyttötapaus	Jokin ominaisuus/tapahtuma, joka järjestelmän pitää toteuttaa. Esim. sisäänkirjautuminen tai asiakastietojen hallinta.
Sidosryhmä	Toimintaympäristössä toisiinsa sidoksissa oleva henkilöryhmä. Esimerkiksi ohjelmistoa kehittäessä

	sidosryhmiä ovat asiakas, loppukäyttäjät ja projektiryhmä.
Suunnittelumalli	Ratkaisu yleiseen suunnitteluongelmaan.
Periyttäminen	Periyttäminen on luokkien välinen suhde, jossa aliluokka perii kantaluokan ominaisuudet. Periyttämisen ansiosta voidaan käyttää uudestaan jo tehtyä ohjelmakoodia uusien luokkien pohjana.
Rajapinta	Komponenttien kohdalle määrittelee kuinka ja kuka voi käyttää sen palveluja ja mitä palveluja komponentti tarvitsee. Olemassa myös roolirajapintoja, jotka toimivat eri rooleissa.

1 JOHDANTO

Ohjelmistojen koot kasvavat ja niiden rakenteet monimutkaistuvat jatkuvasti. Vaikka suurien ohjelmistojen suunnittelu ja toteuttaminen on vaikeaa itsessään, vaaditaan ohjelmistoilta lisäksi uudelleenkäytettävyyttä, helppoa ylläpitoa sekä muunneltavuutta. Dokumentointi on tärkeä osa ohjelmistoja kehitettäessä. Kun ohjelmiston arkkitehtuuri on dokumentoitu, se auttaa ohjelmiston suunnittelijoita ja toteuttajia ymmärtämään toisiaan ja lisäksi se helpottaa kaikkien ohjelmiston ympärillä toimivien sidosryhmien välistä kommunikointia. Lisäksi varsinkin suurissa ohjelmistoissa työn jakaminen voi osoittautua vaikeaksi, jos ohjelmistoa ei ole jaettu osajärjestelmiksi järkevästi. Jotta ohjelmistot täyttäisivät edellä mainitut vaatimukset, täytyy ohjelmistoille suunnitella toimivat arkkitehtuurit.

Tämän tutkimuksen tavoitteena on selvittää, mitä ohjelmistoarkkitehtuuri tarkoittaa, millaisia ohjelmistoarkkitehtuureja on olemassa ja mitkä asiat ovat niille ominaista, millaisia ohjelmistoarkkitehtuurien suunnittelu- ja toteutusmenetelmiä on ja mikä on ohjelmistoarkkitehtuurin merkitys yleensä ja erityisesti ohjelmistotuotannossa.

Asiakkaan tavoitteena oli saada käyttökelpoista luku- ja opetusmateriaalia Ohjelmistoarkkitehtuurit-opintojaksolle, jonka tavoitteena on oppia ohjelmistoarkkitehtuurien sekä ohjelmistoarkkitehtuurisuunnittelun peruskäsitteet ja menetelmät.

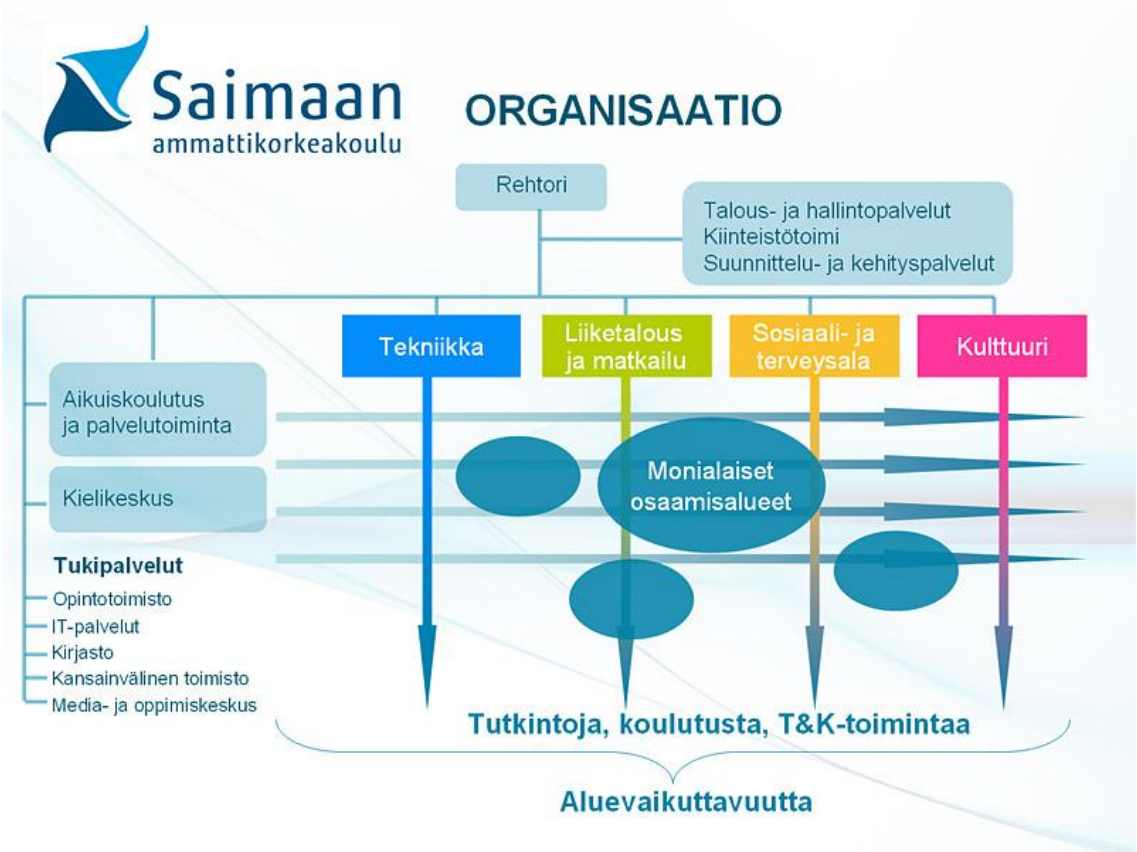
Tässä opinnäytetyöraportissa käsitellään vain tiivistetysti selvitystyön tuloksia, koska yksityiskohtaiset tulokset ovat tuotetussa opintomateriaalissa(liite1). Opinnäytetyöraportin luvussa 1 käsitellään työn lähtökohtia ja tavoitteita. Luvussa 2 kuvataan Saimaan ammattikorkeakoulun toimintaa ja tietotekniikan koulutusohjelman opetussuunnitelman muutoksia, jotka ovat synnyttäneet tarpeen tämän tutkimuksen tekemiseen. Luvussa 3 on käsitelty tiivistetysti

tutkimustuloksista tehdyn opintomateriaalin asiat. Luvussa 4 on kuvattu työprosessi ja luvussa 5 on pohdintaa opinnäytetyön tuloksista.

2 ASIAKAS

Asiakkaana oli Saimaan ammattikorkeakoulun tietotekniikan koulutusohjelma, jonka yhteyshenkilönä toimi Martti Ylä-Jussila.

Saimaan ammattikorkeakoululla on viisi koulutusalaa: tekniikka, sosiaali- ja terveysala, liiketalous, matkailu- ja ravitsemispalvelut sekä kulttuuri (kuva 2.1). Koulutuskampuksia on neljä, joista kaksi sijaitsee Lappeenrannassa ja toiset kaksi Imatralla. Opiskelijoita oli yhteensä noin 3000 ja henkilöstöä noin 300 vuonna 2009.



Kuva 2.1 Saimaan ammattikorkeakoulun koulutusalakaavio (Saimaan AMK)

2.1 Tietotekniikan koulutusohjelma

Tietotekniikan koulutusohjelman tavoitteena on antaa opiskelijoille valmiudet suunnitella ja toteuttaa tietoteknisiä sovelluksia, tietojärjestelmiä ja ohjelmistoja teollisuuden, kaupan ja hallinnon tarpeisiin. Tämän tavoitteen mukaisesti koulutusohjelmassa

opiskellaan monipuolisesti tietotekniikan sovellusten periaatteita, suunnittelua, toteutusta, testausta ja käyttöönottoa. Kahtena viimeisenä vuotena opiskellaan työskentelyä käytännön projekteissa sekä erityyppisten tietojärjestelmien suunnittelua ja hallintaa Tietojärjestelmäklinikalla. Suuntautumisvaihtoehtoja on kolme, ja ne ovat ohjelmistotekniikka, viestintätekniikka ja liiketoiminnan tietojärjestelmät. Tietotekniikan koulutusohjelmasta valmistuneet insinöörit työskentelevät esim. ohjelmistosuunnittelijoina, järjestelmäpäälliköinä, tietotekniikkakonsultteina, systeeminsuunnittelijoina, tietotekniikkakouluttajina tai tietotekniikan asiantuntijoina. (Saimaan AMK, opinto-opas 2008)

Tietotekniikan koulutusohjelman opinnot (Saimaan AMK, opinto-opas 2008):

Tekniikan ammattikorkeakoulututkinto, insinööri (AMK):

Perusopinnot	58 op
– yhteiset perusopinnot	11 op
– koulutusohjelman perusopinnot	47 op
Ammattiopinnot	122 op
– yhteiset ammattiopinnot	78 op
– suuntautumisvaihtoehdon opinnot	44 op
Vapaasti valittavat opinnot	15 op
Harjoittelu	30 op
Opinnäytetyö	15 op
Yhteensä	240 op

2.2 Tietotekniikan koulutusohjelman opetussuunnitelman uudistaminen

Ohjelmointityön siirtyminen Suomesta Itä-Eurooppaan ja Intiaan on vähentänyt ohjelmoijien koulutustarvetta Suomessa. Saimaan ammattikorkeakoulun tietotekniikan koulutusohjelman opetuksen painopistettä on siirretty ohjelmointitekniikasta tietojärjestelmien suunnitteluun. Lisäksi uudessa opetussuunnitelmassa erikoistuminen johonkin tietotekniikan osa-alueeseen on suuressa osassa. Opetustapa on myös muuttunut erikoistumisien vuoksi enemmän projektimaisemmaksi ja PBL(Problem Based Learning), eli ongelmakeskeiseen oppimiseen (Suomen virtuaaliyliopisto, Ongelmakeskeinen oppiminen).

Seuraavassa on verrattu vuonna 2005 ja 2008 tietotekniikan opetusohjelmassa aloittaneiden opetussuunnitelmia ohjelmistotekniikkaan erikoistuvien kurssien suhteen.

2005 ops

OHJELMISTOTEKNIIKAN SUUNTAUTUMISVAIHTOEHTO	32 op
KTE1320 Systeemisuunnittelu II	5 op
KTE1326 Olio-ohjelmointi II	4 op
KTE1152 Olio-ohjelmoinnin suunnittelumallit	3 op
KTE1327 Hajautetut sovellukset	4 op
KTE1328 Ohjelmistotekniikan erikoiskurssi	3 op
KTE1330 Tietorakenteet ja algoritmit II	3 op
KTE1332 Projektityö	10 op

2008 ops

OHJELMISTOTEKNIIKAN SUUNTAUTUMISVAIHTOEHTO	44 op
KTI0080 Tiedonhallinnan jatko	3 op
KTI0081 Olio-pohjaisen ohjelmistosuunnittelun jatko	4 op
KTI0082 Ohjelmistoarkkitehtuurit ja suunnittelumallit	4 op
KTI0083 Käytettävyys ja käyttöliittymäsuunnittelu	4 op
KTI0084 Ihminen vuorovaikutteisessa teknologiassa	3 op

Erikoistuminen johonkin seuraavista: 16 op

- Yhteisölliset järjestelmät
- Avoimen lähdekoodin järjestelmät
- Mobiilijärjestelmät
- Järjestelmäkehitys
- Projektinhallinta
- Laaja projektityö erikoistumisalueesta 10 op

Tietotekniikan opetussuunnitelman 2006–2007 Ohjelmistoarkkitehtuurit-opintojakso on kolmen opintopisteen opintojakso ja sen sisältöön kuuluu vain ohjelmistoarkkitehtuuriin liittyvät asiat. Tämän kurssin jälkeen ei ole jatkokurssia ohjelmistoarkkitehtuurien opiskeluun, paitsi aihetta sivuava Olio-ohjelmoinnin suunnittelumallit –opintojakso, joka on myös kolmen opintopisteen opintojakso.

Tietotekniikan opetussuunnitelmassa 2008–2009 Ohjelmistoarkkitehtuurit- ja Olio-ohjelmoinnin suunnittelumallit –opintojaksot on yhdistetty neljän opintopisteen Ohjelmistoarkkitehtuurit ja suunnittelumallit –opintojaksoksi. Tämän opintojakson jälkeen voi erikoistua järjestelmäkehitykseen, jossa ohjelmistoarkkitehtuurien suunnittelu on merkittävässä asemassa.

Saimaan ammattikorkeakoulu tilasi tutkimuksen nykyisin käytössä olevista ohjelmistoarkkitehtuureiden suunnittelu- ja toteutusmenetelmistä. Tutkimuksen yhtenä tavoitteena oli tuottaa opiskelumateriaalia tekniikan koulutusalan tietotekniikan opetusohjelmaan Ohjelmistoarkkitehtuurit-opintojaksolle, joka myöhemmin laajennetaan Ohjelmistoarkkitehtuurit ja suunnittelumallit -opintojaksoksi.

3 OHJELMISTOARKKITEHTUURIT

Ohjelmistojen kasvaessa yhä vain suuremmiksi ja monimutkaisemmiksi on ohjelmistojen arkkitehtuurien suunnittelu noussut tärkeään asemaan. Suuren ohjelmiston hallitsemisen on vaikeaa, jos sitä ei ole suunniteltu järkevästi. Kun ohjelmiston arkkitehtuuri on hyvin suunniteltu ja dokumentoitu ja ohjelmisto jaettu itsenäisiin osiin, voidaan ohjelmointityökin jakaa osiin ja ohjelmiston testaaminen, ylläpito ja päivitys helpottuvat. Arkkitehtuurin dokumentointi luo yhteisen ymmärryksen ohjelmistosta ja sen rakenteesta kehittäjien välille sekä helpottaa ohjelmiston ympärillä toimivien sidosryhmien välistä kommunikointia. Kun suunnitteluvaiheessa otetaan huomioon uudelleenkäytettävyys ja muunneltavuusvaatimukset saadaan työhön lisää tehokkuutta, koska valmista ohjelmakoodia voidaan käyttää myöhemmin uudelleen. Ohjelmiston arkkitehtuuri toimii järjestelmän yleissuunnitelmana, se kertoo pääpiirteittäin, miten asioiden tulee toimia.

3.1 Ohjelmistoarkkitehtuurin määritelmä

Ohjelmistoarkkitehtuuri on abstrakti käsite ja sille löytyy eri lähteistä hieman toisistaan eroavia määritelmiä, jotka kuitenkin ovat tiettyjen keskeisten käsitysten suhteen yhteneväisiä.

IEEE:n arkkitehtuurien kuvaamista koskevassa standardissa ohjelmistoarkkitehtuuri määritellään järjestelmän perusorganisaatioksi, joka sisältää järjestelmän osat, niiden väliset suhteet ja niiden suhteet ympäristöön, sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua. (IEEE 1471-2000)

”Tietojärjestelmän arkkitehtuuri kuvaa kohdealueensa rakenneosat, niiden ulospäin näkyvät ominaisuudet ja niiden väliset yhteydet ja riippuvuudet. Arkkitehtuuri muodostaa rungon järjestelmän suunnittelulle ja toteutukselle sekä ohjaa järjestelmän rakenteen kehittämistä järjestelmän elinkaaren ajan. Se toimii myös keskusteluvälineenä järjestelmän kehittämisen ja ylläpitämisen

sidosryhmien (organisaation johto, käyttäjät, suunnittelijat, toteuttajat) välillä.” (Wikipedia, Tietojärjestelmäarkkitehtuuri)

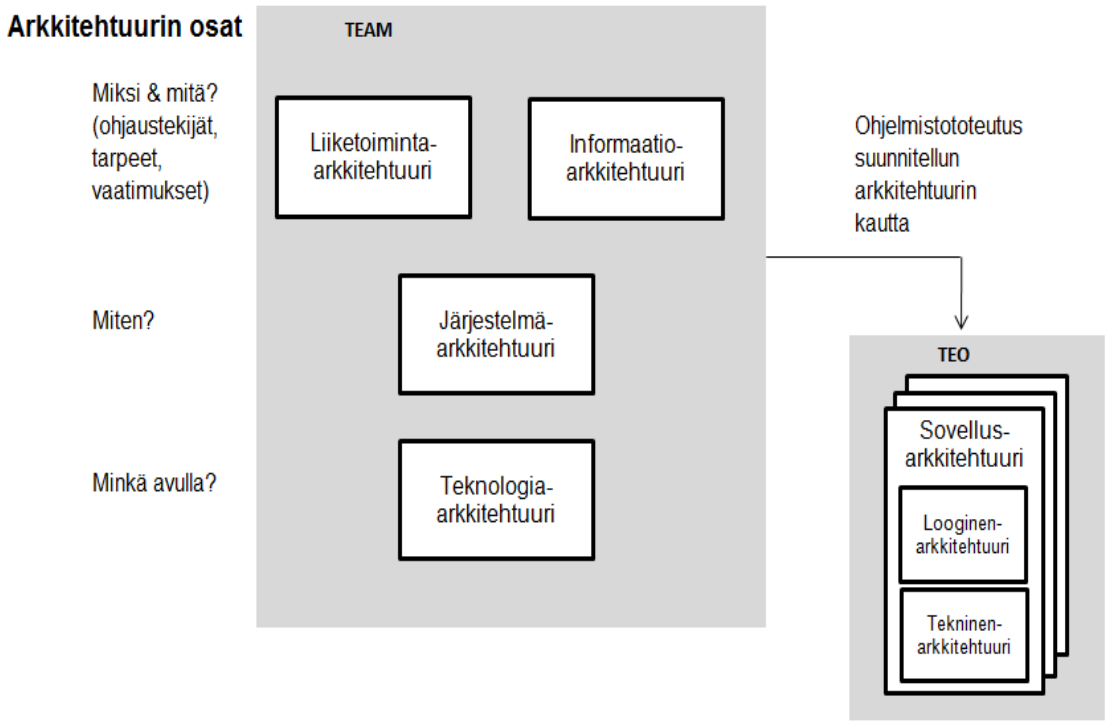
”Voidaankin ajatella, että arkkitehtuuri on järjestelmän perustuslaki. Sitä on noudatettava järjestelmää rakennettaessa, ja sitä voidaan muuttaa vain erittäin painavilla perusteilla..... Toisin sanoen arkkitehtuuri määrittelee rajat, joiden puitteissa järjestelmää on rakennettava ja ylläpidettävä.” (Koskimies & Mikkonen, 2005)

Vaikka ohjelmistoarkkitehtuurilla on monenlaisia määritelmiä, niin seuraavissa keskeisissä asioissa määritelmät ovat yhteneväisiä: Ohjelmistoarkkitehtuuri käsittää järjestelmän jaon osiin ja näiden osien väliset suhteet mutta ei osien sisäisiä asioita. Arkkitehtuuri tarjoaa järjestelmällä pohjan, jonka päälle rakentaa ohjelmisto.

3.2 Yritysarkkitehtuurit

Yritysarkkitehtuurit ohjaavat suuresti ohjelmistojen toteutusta. Yrityksen liiketoiminta- ja informaatioarkkitehtuurit määrittelee mitä pitäisi tehdä ja miksi, ja järjestelmäarkkitehtuuri kertoo tavan tehdä sovelluksen ja teknologia -arkkitehtuuri kertoo, millä teknologialla ohjelmisto pitää tehdä (kuva 3.1).

Yritysarkkitehtuuriajattelumalli ei sovellu ainoastaan liiketoimintaan, vaan myös mille tahansa organisaatiolle, jonka toimintaa ohjaavat yhteiset tavoitteet ja päämäärät. Yritysarkkitehtuuri muuttuu nimensä mukaisesti IT-keskeisestä ajattelutavasta kokonaisvaltaiseen ja prosessiperusteiseen viitekehykseen, jonka tavoitteena on luoda joustava ja tehokas hallintaväline niin liiketoiminta kuin IT-muutoksia varten. (TietoEnator)



Kuva 3.1 Arkkitehtuurin osat (TietoEnator)

Lähes kaikista yritysarkkitehtuureista on tunnistettavissa tavalla tai toisella seuraavat kokonaisuudet (TietoEnator):

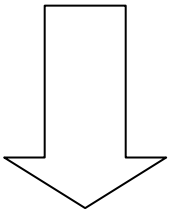
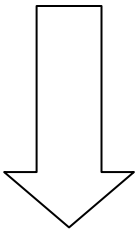
- Liiketoiminta-arkkitehtuuri (business architecture)
- Informaatioarkkitehtuuri (information architecture)
- Järjestelmäarkkitehtuuri (system architecture)
- Teknologia-arkkitehtuuri (technology architecture)

Taulukko 3.1 Yritysarkkitehtuurin osa-alueet (TietoEnator)

Liiketoiminta-arkkitehtuuri	**Informaatio-arkkitehtuuri**
Tavoitteet Palvelut Prosessit	Tietotarpeet Tietovarannot Tietovirrat
Teknologia-arkkitehtuuri	**Järjestelmä-arkkitehtuuri**
Standardit Teknologiat Ratkaisut	Järjestelmäsalkku Stardardit Menettelyt

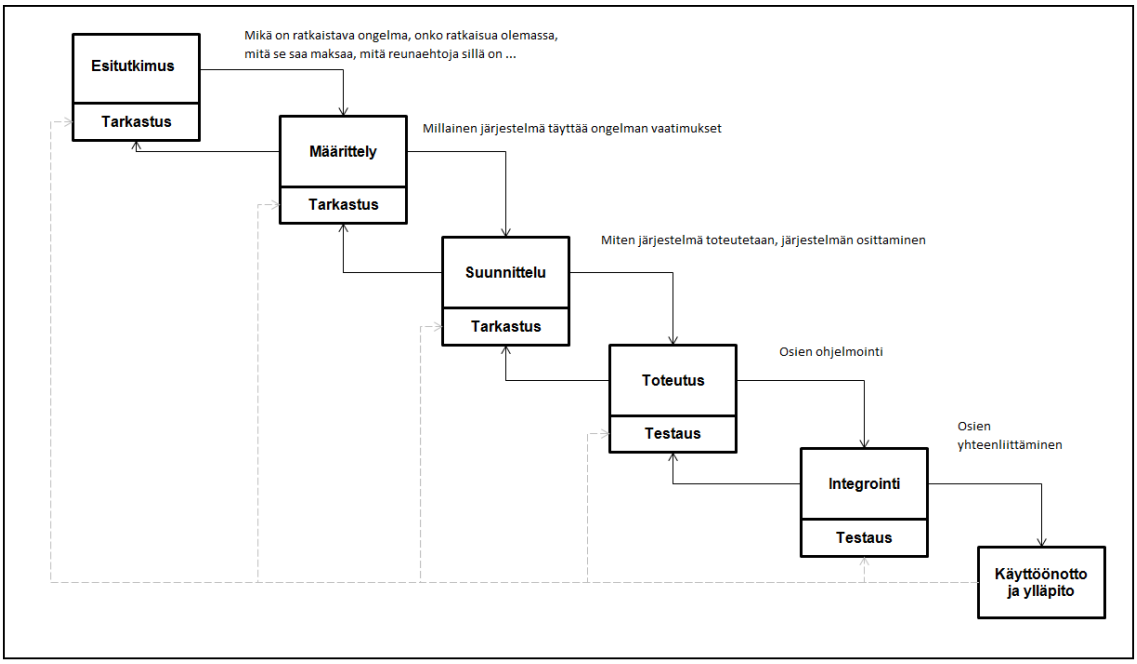
Liiketoiminta-arkkitehtuuri kuvaa muun muassa organisaation tavoitteet, palvelut ja tuotteet sekä liiketoimintaprosessit, joissa palveluja ja tuotteita tuotetaan. Informaatioarkkitehtuuri kuvaa organisaation toiminnassaan tarvitsemat tietotarpeet, tietovarastot ja tietojen väliset suhteet. Järjestelmäarkkitehtuuri kuvaa järjestelmät ja sovellukset, joiden avulla informaatioarkkitehtuurin sisältämiä tietoja hallinnoidaan liiketoiminta-arkkitehtuurin edellyttämillä tavoilla. Teknologia-arkkitehtuuri kuvaa organisaation IT-järjestelmien ja -sovellusten kehittämisessä ja hallinnoinnissa käytettävät teknologiset ratkaisut ja standardit. (TietoEnator)

Taulukko 3.2 Yritysarkkitehtuurin vaikutus systeemityöprojekteihin (TietoEnator)

<table>
<tr><td rowspan="2">Ohjaa ohjelmisto-arkkitehtiä
  </td><td>Yritysarkkitehtuuri/järjestelmä ja teknologia-arkkitehtuuri
- Arkkitehtuurin visio, periaatteet, tyylit, avainkonseptit ja –mekanismit
- Fokus: korkean tason päätöksiä, jotka vaikuttavat järjestelmän rakenteeseen; rajaa joitakin rakenteellisia ratkaisuja pois ja ohjaa hyväksyttyjen rakenteiden käyttöön ja vertailuun.</td></tr>
<tr><td>Ohjelmistoarkkitehtuuri
- Rakenne ja suhteet, staattiset ja dynaamiset näkymät, oletukset ja perustelut
- Fokus: osiin jako ja vastuiden asettaminen, rajapintojen suunnittelu, kohdentaminen prosesseille</td></tr>
<tr><td>Ohjaa suunnittelijaa
  </td><td>Arkkitehtuuriohjeet ja politiikat
- Malleja ja ohjeita; ratkaisumalleja, kehikkoja, infrastruktuuria ja stadardeja
- Fokus: Ohjaa suunnittelijaa luomaan sellaisia suunnitteluratkaisuja, jotka ylläpitävät arkkitehtuurin eheyttä.</td></tr>
</table>

3.3 Ohjelmistoarkkitehtuurisuunnittelu perinteisessä ohjelmistonkehitysprosessissa

Ohjelmiston elinkaarella tarkoitetaan aikaväliä ohjelmiston kehittämisen aloittamisesta sen poistamiseen käytöstä. Vaihejakomallilla tarkoitetaan tapaa, jolla ohjelmiston kehitystyö tai koko elinkaari jaetaan vaiheisiin. Tavallisin vaihejakomalli on ns. vesiputousmalli, jonka eräs versio on kuvassa 3.2. Mallista on erilaisia muunnelmia, mutta yleensä niistä voidaan erottaa ainakin määrittely-, suunnittelu- ja toteutusvaiheet. Määrittelyvaihetta edeltää usein esitutkimusvaihe. (Haikala & Märijärvi, 2004)



Kuva 3.2 Esimerkki vesiputousmallista (Haikala & Märijärvi, 2004)

Kaikkiin vaiheisiin liittyy laadunvarmistustoimenpiteitä, kuten tarkastuksia, katselmuksia ja testausta, joilla pyritään kitkemään virheet järjestelmästä mahdollisimman varhaisessa vaiheessa. Katselmuksia pidetään yleensä vaiheiden päätteeksi. (Haikala & Märijärvi, 2004)

Esitutkimuksen aikana otetaan selville asiakkaan tarpeet ja selvitetään, miksi ja mitä järjestelmän täytyy kyetä tekemään, ottamatta kantaa siihen, millainen järjestelmästä tulee. Esitutkimus vastaa kysymykseen miksi eikä miten.

Esitutkimusvaihe on tärkeä vaihe siinä mielessä, että vääristä vaatimuksista ei voi päätyä hyvään järjestelmään. Onkin tärkeää ymmärtää, mitä asiakas tarvitsee. (Haikala & Märijärvi, 2004)

Määrittelyvaiheessa asiakasvaatimuksia analysoidaan ja niistä johdetaan ohjelmistovaatimukset, jotka määrittelevät toteutettavan järjestelmän. Asiakasvaatimuksia kutsutaan myös nimellä järjestelmävaatimukset, toiminnalliset vaatimukset ja ominaisuudet. Määrittelyn tuloksena saadaan dokumentti nimeltään toiminnallinen määrittely. (Haikala & Märijärvi, 2004)

Toiminnallisessa määrittelyssä kuvataan ohjelmiston toiminnot, toteutukselle asetettavat ei-toiminnalliset vaatimukset sekä rajoitukset. Toimintojen yhteydessä määritellään ohjelmistolla toteutettavat ominaisuudet, käyttöliittymä ja kommunikointi muiden järjestelmien kanssa. (Haikala & Märijärvi, 2004)

Suunnitteluvaiheessa suunnitellaan määrittelyn kuvaamien toimintojen toteutus. Suunnitteluvaihe jaetaan usein kahteen (tai useampaan) tasoon. Aluksi järjestelmä jaetaan mahdollisimman itsenäisiin, toisistaan riippumattomiin osiin, moduuleihin. Tätä vaihetta kutsutaan arkkitehtuurisuunnitteluksi. Arkkitehtuurisuunnittelun tuloksena saadaan tekninen määrittely. Arkkitehtuurisuunnittelua seuraa moduulisuunnitteluvaihe, jossa jokaisen moduulin sisäinen rakenne suunnitellaan. (Haikala & Märijärvi, 2004)

Toteutusvaiheessa ohjelmaa ohjelmoidaan suunnitelmien mukaan. Käytännössä ohjelmoijat suorittavat tässä vaiheessa moduulitestauksen omille moduuleilleen.

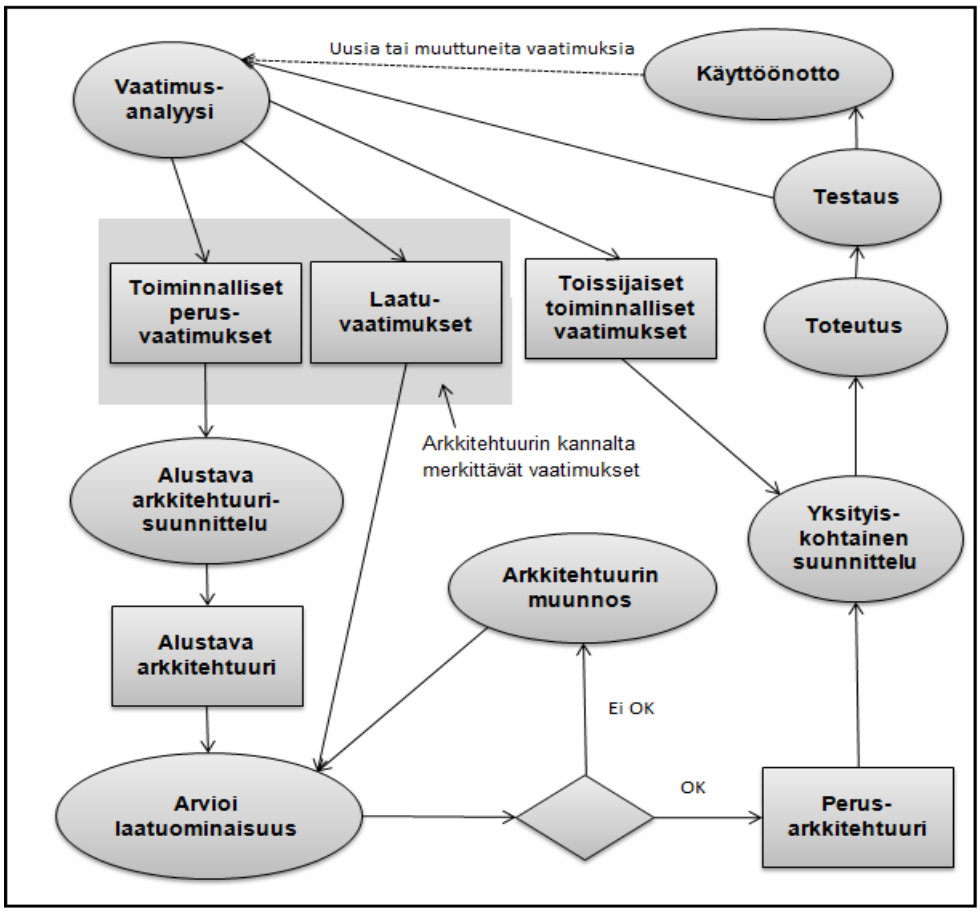
Testausvaiheessa ohjelmistoa testataan ja yritetään etsiä virheitä. Testaus tapahtuu usein monella tasolla ns. V-mallin mukaisesti. V-mallissa testaus jaetaan moduulitestaukseen, integrointitestaukseen ja järjestelmätestaukseen. (Haikala & Märijärvi, 2004)

Ylläpito on asiakkaan ongelmien ratkomista, virheiden korjaamista, ohjelman muuttamista vaatimusten muuttuessa sekä uusien piirteiden lisäämistä. (Haikala & Märijärvi, 2004)

3.4 Arkkitehtuuripainotteinen ohjelmistonkehitysprosessi

Arkkitehtuuripainotteisessa ohjelmistonkehitysprosessissa korostetaan arkkitehtuurin suunnittelua ja arviointia ennen siirtymistä yksityiskohtaiseen suunnitteluun ja toteutukseen. Arkkitehtuuri pyritään luomaan iteratiivisesti arkkitehtuurin kannalta tärkeistä vaatimuksista. Arkkitehtuuriin vaikuttavat toiminnalliset sekä laadulliset vaatimukset.

Prosessi etenee tyypillisesti (Kuva 3.3) siten, että ensimmäinen versio arkkitehtuurista tehdään toiminnallisten vaatimusten pohjalta; tätä versiota verrataan laadullisiin vaatimuksiin. Mikäli huomataan tarvetta muutoksille, niin arkkitehtuuria muutetaan siten, että myös laadulliset vaatimukset täyttyvät. Kun myös laadulliset vaatimukset täyttyvät, on järjestelmän perusarkkitehtuuri valmis. Tästä edetään yksityiskohtaisempaan suunnitteluun, toteutukseen ja testaukseen, josta saadaan ensimmäinen, keskeisimmät toiminnalliset vaatimukset täyttävä versio. Sitten tarkastellaan sekundaarisia vaatimuksia ja toteutetaan ne perusarkkitehtuurin pohjalle. Tämän jälkeen versiota tehdään inkrementaalisti vaatimus kerrallaan. Jos sovelluksen käyttöönoton jälkeen havaitaan tarve uusille vaatimuksille, tai jotain vaatimusta täytyy muuttaa, joudutaan koko prosessi periaatteessa käymään uudestaan läpi. Jos vaadittavat muutokset ovat suuria, voidaan arkkitehtuuria joutua muuttamaan, tai ainakin arvioimaan uudestaan.



Kuva 3.3 Arkkitehtuuripainotteinen ohjelmistokehitysprosessi (Koskimies, Mikkonen, 2005)

3.5 Ohjelmistoarkkitehtuurien dokumentointi

Jos järjestelmän arkkitehtuuria ei ole dokumentoitu, on seurauksena ennen pitkää järjestelmän rapautuminen. Arkkitehtuurista keskusteltaessa eri ihmiset tekevät omat päätelmänsä arkkitehtuurista, joten tarvitaan jokin konkreettinen dokumentti selittämään asia. Periaatteessa mitään ohjelmistoprojektia ei pitäisi päästää etenemään, jos arkkitehtuuria ei ole selkeästi dokumentoitu.

3.5.1 Ohjelmistoarkkitehtuurin kuvaus

Arkkitehtuurikuvaus antaa järjestelmästä tietoa, jota ei muualta saa. Jos järjestelmän toteutus on ristiriidassa arkkitehtuurikuvauksen kanssa, on

järjestelmä toteutettu väärin. On järkevää ajatella, että arkkitehtuuri realisoituu arkkitehtuurikuvauksessa, ei järjestelmässä.

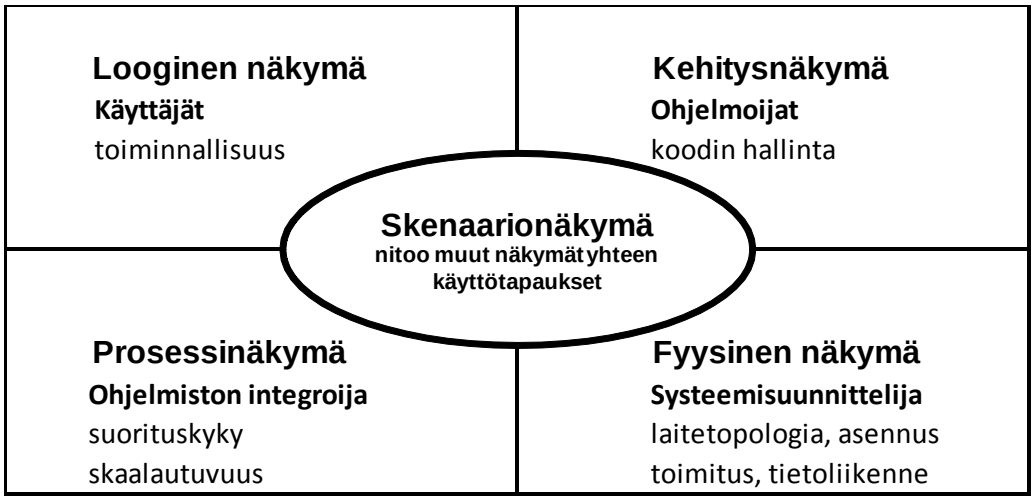
Ohjelmiston arkkitehtuuri on tärkein ohjelmistoa luonnehtiva informaatio. Tästä syystä useimmat ohjelmistoon liittyvien sidosryhmien kysymykset koskevat arkkitehtuuria. On siis tärkeää, että ohjelmistolla on kattava ja selkeä kuvaus sen arkkitehtuurista, jotta kaikilla on sama käsitys järjestelmästä. Arkkitehtuurilla on tärkeä merkitys järkevän kommunikaation mahdollistavana ohjelmistoartefaktina. Arkkitehtuuri ei ainoastaan kuvaa sitä, miten keskeiset tehtävät ratkaistaan, vaan tarjoaa myös järjestelmää koskevan käsitteistön ja sanaston, jolloin järjestelmästä voidaan puhua oikein termein.

Arkkitehtuurikuvaus on tärkeä osa arkkitehdin ja toteutusryhmän välistä kommunikointia. Jos arkkitehtuuri on vain arkkitehdin päässä, on toteutusryhmän vaikea olla täysin perillä siitä, millainen arkkitehtuurin tarkasti ottaen pitäisi olla. On siis tärkeää, että arkkitehti tarjoaa ryhmälle tarkan kuvauksen arkkitehtuurista mahdollisimman pian, koska ilman kuvausta, toteutusryhmä luultavammin tekee omia ratkaisujaan.

3.5.2 Arkkitehtuurin kuvauksessa käytetyt näkökulmat

Ohjelmistoarkkitehtuurin on kuvattava, millaisia komponentteja järjestelmässä on ja mitä suhteita niiden välillä on. Näitä suhteita on vaikea selvittää yleisen määritelmän perusteella, joten tarvitaan eri näkökulmia arkkitehtuurin tutkimiseen.

Yleinen tapa kuvata ohjelmistoarkkitehtuuria eri näkökulmista on nk. 4 + 1 – malli (Wikipedia, Kruchten)(Kuva 3.4).



Kuva 3.4 4 + 1 –malli (Wikipedia, Kruchten)

Skenaarionäkymässä tarkastellaan järjestelmän vuorovaikutusta ulkopuolisiin käyttäjiin ja järjestelmiin, se siis kuvaa järjestelmän rajapinnat ympäristöönsä. Skenaarionäkymässä käsitellään yleensä järjestelmän keskeisiä toimintoja, joten se toimii lähtökohtana muiden näkymien muodostamiseen. Muita näkymiä voidaan myös arvioida skenaarionäkymää vasten, jotta nähdään miten näkymässä esitetyt ratkaisut tukevat kyseistä käyttöskenaariota. Skenaarionäkymä on olennainen lähes kaikille järjestelmille.

Loogisessa näkymässä kuvataan, kuinka järjestelmän toiminnallisuus on jaettu eri osien kesken ja mitä vastuita eri osilla on. Looginen näkymä kuvaa skenaarionäkymässä esitetyn käyttötapauksen eri osien välisenä yhteistyönä. Loogista näkymää käytetään yksityiskohtaisen suunnittelun pohjana, ja se on tärkeä näkymä lähes kaikille järjestelmille.

Prosessinäkymässä kuvataan, kuinka järjestelmän toiminnot on jaettu loogisiin prosesseihin ja prosessien kommunikointiin. Prosessinäkymää tarvitaan varsinkin järjestelmiin, joissa vaaditaan rinnakkaisuutta. Prosessinäkymää käytetään arvioimaan suorituskykyä ja skaalautuvuutta.

Kehitysnäkymä kuvaa, kuinka järjestelmä on jaettu osiin, jotka voidaan toteuttaa erillisinä yksikköinä. Kehitysnäkymää tarvitaan varsinkin suurissa

ohjelmistoissa, ja sitä käytetään projektisuunnittelussa, kustannusarvioissa ja projektinhallinnassa.

Fyysisessä näkymässä kuvataan järjestelmässä tarvittavat fyysiset artefaktit, millaisista osista järjestelmä koostuu, miten artefaktit ovat yhteydessä toisiinsa, ja mihin prosessointiyksikköihin mitäkin sijoitetaan.

Edellä mainittujen näkökulmien lisäksi, varsinkin tuoterunkoarkkitehtuurien yhteydessä, on tarvetta vielä yhdelle näkökulmalle, variaationäkökulmalle.

Variaationäkökulma kuvaa, millaista muuntelua järjestelmä tukee. Muuntelunäkökulma kuvaa näin järjestelmän variaatiopisteet (so. kohdat järjestelmässä, joissa muuntelu tapahtuu), niiden käytön muunnelmien toteuttamiseen. Muuntelunäkökulma kuvaa siis järjestelmän toteutusalustana sovelluksille, määritellen järjestelmän laajennusrajapinnan. Muuntelunäkökulma on olennainen tuoterunkoarkkitehtuurien kohdalla, mutta se voi olla hyödyllinen esimerkiksi ylläpidon kannalta mille hyvänsä ohjelmistolle. Muuntelunäkymää tarvitaan myös mm. järjestelmän uudelleenkäytettävyyden arvioimiseksi. (Koskimies & Mikkonen, 2005)

3.5.3 Arkkitehtuuriviipale

Todellisuudessa hiemankin monimutkaisempi järjestelmä koostuu useasta loogisesta kokonaisuudesta, joista osa on järkevä esittää yksittäisinä komponentteina, mutta monet osat taas koostuvat useista komponenteista.

Ohjelmistotekniikassa on alettu ymmärtää, että järjestelmää osiin jaettaessa ei voida kaikkia loogisia kokonaisuuksia tehdä erillisinä ohjelmayksikköinä. Tällaistä järjestelmän pääositusperusteen kanssa eriävää mutta jonkin kriteerin perusteella loogisesti yhteenkuuluvaa rakennetta kutsutaan (arkkitehtuuri)viipaleeksi.

Tyypillisiä viipaleita ovat suunnittelumallien ilmentymät, järjestelmän ulospäin näkyvät piirteet ja aspektit. Jälkimmäiset ovat jonkin koko järjestelmää tai sen suurta osaa koskevan ominaisuuden tai tarpeen toteutuksia. Aspektina voi olla vaikkapa virheenkäsittely, lokitulostus, hajautus, olioiden pysyvyys ja tietoturvallisuus. Kaikki nämä ovat tarpeita, joiden toteutus tavallisesti jakautuu useiden komponenttien sisään. Viipaleet voivat olla myöskin keskenään osittain päällekkäisiä, esimerkiksi sama komponentti tai luokka voi esiintyä useassa suunnittelumallissa ja samalla osallistua monen piirteen toteutukseen. (Koskimies & Mikkonen, 2005)

On tärkeää olla selvillä arkkitehtuuriviipaleiden olemassa olosta ja niiden luonteesta. Jos esimerkiksi jotakin järjestelmän piirrettä tai olioiden pysyvyyttä halutaan muuttaa, täytyy piirteitä vastaavat viipaleet löytää järjestelmästä. Kun näitä viipaleita ei edusta mitkään yksittäiset komponentit, voi olla vaikeaa löytää ne, jos viipaleiden olemassaoloa ei ole dokumentoitu etukäteen.

3.5.4 Arkkitehtuuridokumenttityypit

Yleisempiä arkkitehtuuridokumenttityyppejä jotka ovat yleisesti käytössä teollisuudessa, ovat alustava-, järjestelmä-, alijärjestelmä-, tuoterunko- ja tuotearkkitehtuuridokumentti sekä rajapintadokumentti.

Alustava arkkitehtuuridokumentti kuvaa tärkeimmät ratkaisut arkkitehtuurissa ja niiden perustelut. Myös vaihtoehtoiset ratkaisut ja niiden hyvät ja huonot puolet voidaan käsitellä alustavassa arkkitehtuuridokumentissa. Dokumenttia käytetään liiketoimintapäätösten, projektisuunnittelun, työmääräarvioinnin, alustavan arkkitehtuurin arvioinnin sekä tarkemman arkkitehtuurisuunnittelun pohjana. Dokumentissa kuvataan tyypillisesti konkreettinen arkkitehtuuri, mutta siihen voi sisältyä viittauksia referenssiarkkitehtuureihin.

Järjestelmäarkkitehtuuridokumentti kuvaa järjestelmän arkkitehtuurin ylimmältä tasolta. Kuvaa järjestelmän sidokset ympäristöönsä, alijärjestelmien ulkoiset ominaisuudet (esim. rajapinnat) ja niiden väliset vuorovaikutukset.

Vuorovaikutus voidaan kuvata vaikka siten, kuinka käyttötapaukset toteutuvat alijärjestelmien vuorovaikutuksena. Dokumenttia käytetään arkkitehtuurisuunnitteluun, arkkitehtuurin arvioinnissa, tarkennettujen työmääräarvioiden tekemiseen, projektisuunnitteluun ja järjestelmätestauksen suunnitteluun. Dokumentissa kuvataan tyypillisesti konkreettinen arkkitehtuuri, mutta siihen voi sisältyä kuvauksia esim. alijärjestelmien meta-arkkitehtuureista.

Alijärjestelmäarkkitehtuuridokumentti kuvaa alijärjestelmän sisältämien komponenttien ulkoiset ominaisuudet (esim. rajapinnat) ja niiden välisen vuorovaikutuksen. Vuorovaikutus voidaan kuvata vaikkapa tarkentamalla käyttötapauksien suoritus komponenttien tasolle. Dokumenttia käytetään pohjana alijärjestelmien ja komponenttien yksityiskohtaiseen suunnitteluun ja toteutukseen, työnjakoon sekä yksikkötestauksen suunnitteluun. Dokumentti kuvaa konkreettisen arkkitehtuurin.

Tuoterunkoarkkitehtuuridokumentti kuvaa, kuinka rungon tarjoamaa arkkitehtuurin varianssia voidaan käyttää hyödyksi muiden sovellusten suunnittelussa. Hyvä tapa tähän on esittää esimerkkejä rungon soveltamisesta. Tämä dokumentti kertoo myös mitä sääntöjä sovellusten tulee noudattaa omissa arkkitehtuureissaan.

Tuotearkkitehtuuridokumentti kuvaa, kuinka tuoterunkoarkkitehtuuria on sovellettu ja mitä tuoterungosta riippumattomia ratkaisuja on tehty. Sitä käytetään tuotteen yksityiskohtaiseen suunnitteluun, testaukseen ja ylläpitoon. Dokumentissa kuvataan konkreettinen arkkitehtuuri.

Rajapintadokumentti täydentää muita dokumentteja kuvaamalla järjestelmän, alijärjestelmän tai tuoterungon tarjoaman ohjelmointirajapinnan (API). Sitä käytetään komponenttien suunnittelun ja toteutuksen perustana.

3.6 Komponentit ja rajapinnat

Järjestelmät koostuvat komponenteista ja rajapinnat kuvaavat näiden liittymät. Molemmat ovat keskeisiä käsitteitä arkkitehtuuria suunnitellessa.

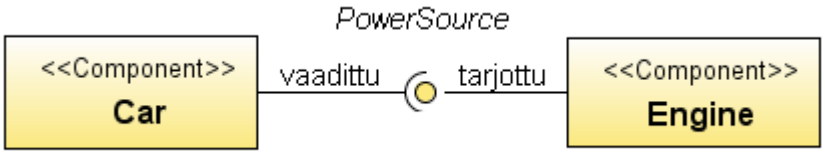
3.6.1 Komponentit

Ohjelmistokomponentille on kirjallisuudessa monia erilaisia määritelmiä, jotka eivät sovi yhteen nykyisen komponenttikäsityksen kanssa. Nykyisin ohjelmistokomponentti mielletään itsenäiseksi ohjelmistoyksiköksi, joka tarjoaa palvelujaan hyvin määriteltyjen rajapintojen kautta.

3.6.2 Rajapinnat

Yksi tärkeimmistä periaatteista ohjelmistotekniikassa on pyrkiä erottamaan toisistaan se, mitä halutaan saada aikaan ja se, miten tämä tapahtuu. Komponenttien kohdalla tämä tarkoittaa sitä, että palvelun toteutus on erotettava palvelusta abstraktiona: palvelun käyttäjä ei saisi olla riippuvainen tietystä palvelun tuottajasta, komponentista, vaan palvelusta itsestään abstraktina käsitteenä. Tämä abstrakti käsite esitellään rajapintana, jonka yksi tai useampi konkreettinen komponentti toteuttaa. (Koskimies & Mikkonen, 2005) Rajapinnat määräävät, miten komponentit kommunikoivat keskenään, tästä syystä rajapinnat ovat tärkeä osa ohjelmistoarkkitehtuuria. Rajapintojen järkevä suunnittelu helpottaa itse työn jakamista ja myöhemmin ohjelmiston testausta ja ylläpitoa.

Komponentin ja rajapinnan välillä voi olla kaksi erilaista suhdetta (kuva 3.5), rajapinta voi olla tarjottu tai vaadittu komponentille. Sama rajapinta voi olla tarjottu jollekin komponentille ja toiselle vaadittu.

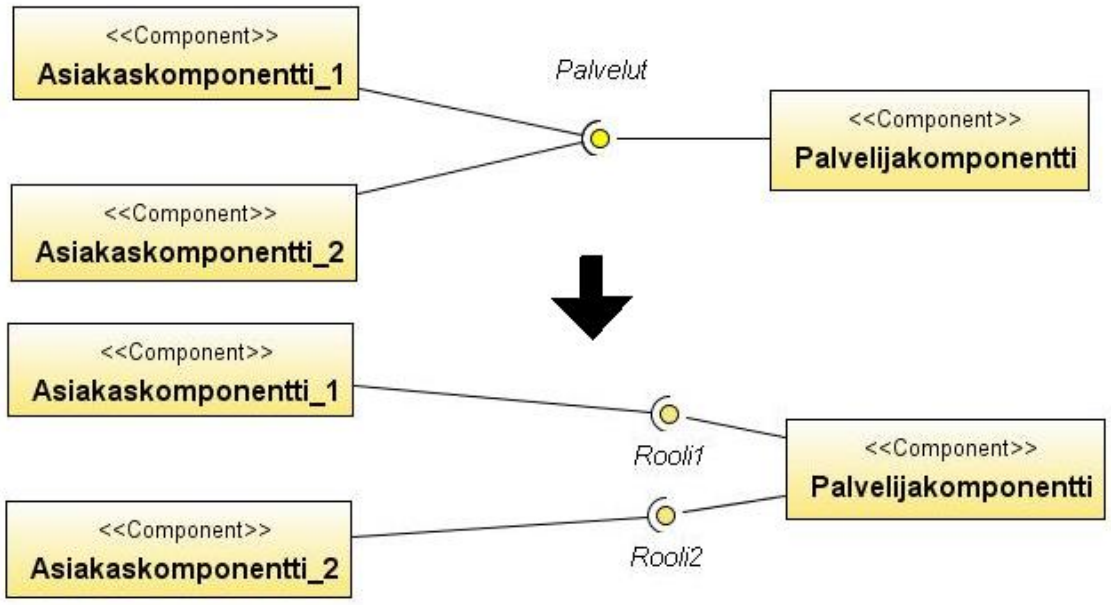


Kuva 3.5 Tarjotut ja vaaditut rajapinnat UML:ssä (Koskimies & Mikkonen, 2005)

Car-komponentti vaatii rajapinnan PowerSource (Auto tarvitsee voimanlähteen). Engine-komponentti tarjoaa (toteuttaa) tämän rajapinnan.

Tarkoituksenmukaisempi ja hienojakoisempi rajapinta saadaan aikaan tarkastelemalla tarkemmin sitä, miten komponenttien toiminnallisuus voidaan toteuttaa eri rooleissa olevien palvelun tarjoajien avulla. Komponentit tarjoavat toisilleen palveluja aina jossain roolissa, mutta tämä rooli ei välttämättä kata kaikkia palveluja, jotka kyseinen komponentti pystyy toteuttamaan. Tarkoituksena olisi antaa erillinen rajapinta kaikille identifioiduille rooleille, joissa komponentti palvelee toista komponenttia. Tällaistä rajapintaa kutsutaan roolirajapinnaksi. Komponentti on usein eri palvelijarooleissa muihin komponentteihin nähden, joten se voi toteuttaa useita roolirajapintoja, ja roolirajapinnan palveluiden tarjoajiin voi kuulua useampi komponentti.

Roolirajapinnat (kuva 3.6) helpottavat järjestelmän ylläpitoa, koska jos muutoksia täytyy tehdä vaikkapa jonkin palvelun kutsumuotoon, se heijastuu ainoastaan komponenteissa, jotka todella käyttävät palvelua, eikä kaikissa komponenteissa, jotka käyttävät palvelun tarjoavaa komponenttia. Toisaalta taas roolirajapinnat lisäävät rajapintojen lukumäärää ja monimutkaistavat järjestelmän arkkitehtuuria.



Kuva 3.6 Roolirajapintojen käyttö (Koskimies & Mikkonen, 2005)

3.6.3 Komponenttien räätälöinti

Komponenttia suunnitellessa pitäisi ottaa huomioon sen uudelleenkäytettävyys. Jotta komponentti olisi uudelleenkäytettävä, sitä pitää pystyä käyttämään erilaisissa yhteyksissä, joilla saattaa olla eri odotuksia komponentille. Jos komponentti tarjoaa palvelunsa aina samalla tavalla, sitä on vaikea hyödyntää. Kun komponenttia voi räätälöidä eri ympäristöihin, sen uudelleenkäytettävyysaste nousee.

Yleisimmät tavat, joilla komponentit saadaan tarjoamaan varianssia, ovat komponentin tilan muuttaminen, vaadittujen rajapintojen toteutus ja periyttäminen.

3.6.4 Komponenttien väliset vuorovaikutustekniikat

Ohjelmistoarkkitehtuurisuunnittelu on suureksi osaksi komponenttien välisten suhteiden määrittelyä. Tämä johtuu siitä, että yleisempiä ohjelmistolle asetettavia vaatimuksia ovat uudelleenkäytettävyys, ylläpidettävyys ja työn hajautettavuus ja näihin kaikkiin vaikuttavat ohjelmiston komponenttien suhteet

ja riippuvuudet. On tärkeää vähentää ja selkeyttää näitä riippuvuuksia. Näin saadaan täytettyä ohjelmiston laadullisia vaatimuksia. Käytännössä tämä toteutetaan esimerkiksi rajapinnoilla ja välittäjäluokilla.

Yleisimmät vuorovaikutustekniikat ovat välittäjän käyttö, kutsun siirtäminen, edustajakomponentin käyttö, takaisinkutsun käyttö, tapahtumiin perustuva, vuorovaikutus, sovittimen käyttö ja tehtaan käyttö.

3.7 Suunnittelumallit

Mestareiden töiden kopiointi on ikivanha keksintö, ja juuri siitä on kyse suunnittelumalleissa. Edellä mainittu toimii hyvin varsinkin ohjelmistotekniikan alalla, koska alalla on paljon kokemattomia ohjelmoijia ja toisaalta taas paljon ammattitaitoisia ohjelmistoarkkitehteja, jotka voivat jakaa tietämystään suunnittelumalleilla, joilla ratkaistaan yleisiä ohjelmistonsuunnitteluun liittyviä ongelmia. Ohjelmistotekniikassa suunnittelumalli tarkoittaa siis yleistä tapaa jonkin usein esiintyvän ongelman ratkaisemiseksi. Kirjassa ”Olio-ohjelmointi Suunnittelumallit, Gamma E. et al, 2001” ja alkuperäisessä ”Design Patterns – Elements of Reusable Object-Oriented Software, Gamma E. et al, 1995” on esitelty 23 yleistä suunnittelumallia.

3.8 Arkkitehtuurityylit

Arkkitehtuurityylit ovat yleisiä malleja, jotka määräävät, kuinka järjestelmä organisoidaan ylimmällä abstraktiotasolla ja kertovat järjestelmän teknisen luonteen.

Arkkitehtuurityyleillä ja suunnittelumalleilla on paljon yhteistä, mutta rajanvetoina voidaan pitää kahta asiaa: Suunnittelumallista on useita instansseja järjestelmässä, ja järjestelmässä voi olla useita suunnittelumalleja käytössä. Arkkitehtuurityyli määrää järjestelmän kokonaisrakenteen.

Yleisimmät arkkitehtuurityylit ovat kerrosarkkitehtuurit, tietovuoarkkitehtuurit, asiakas-palvelin-arkkitehtuuri, viestinvälitysarkkitehtuurit, malli-näkymä-ohjain-arkkitehtuurit, tietovarastoarkkitehtuurit ja tulkkipohjaiset arkkitehtuurit.

3.9 Tuoterunkoarkkitehtuurit

Ohjelmistotuotannossa on tärkeää pystyä uudelleenkäyttämään tehtyjen ohjelmistojen osia muissakin ohjelmistoissa. Kun ohjelmistot suunnitellaan alusta lähtien uudelleenkäytettäväksi, se nopeuttaa tulevien saman sovellusalueen ohjelmistojen toteutusta. Tuoterunkoarkkitehtuurit toimivat jonkin tietyn sovellusalueen tuotteiden pohjana, jonka päälle halutut tuotteet tehdään.

Tuoterunkoarkkitehtuureihin liittyviä käsitteitä ovat tuoteperhe, tuotelinja, tuoterunko ja tuotealusta sekä tuoterunkoarkkitehtuuri ja tuoteperhearkkitehtuuri.

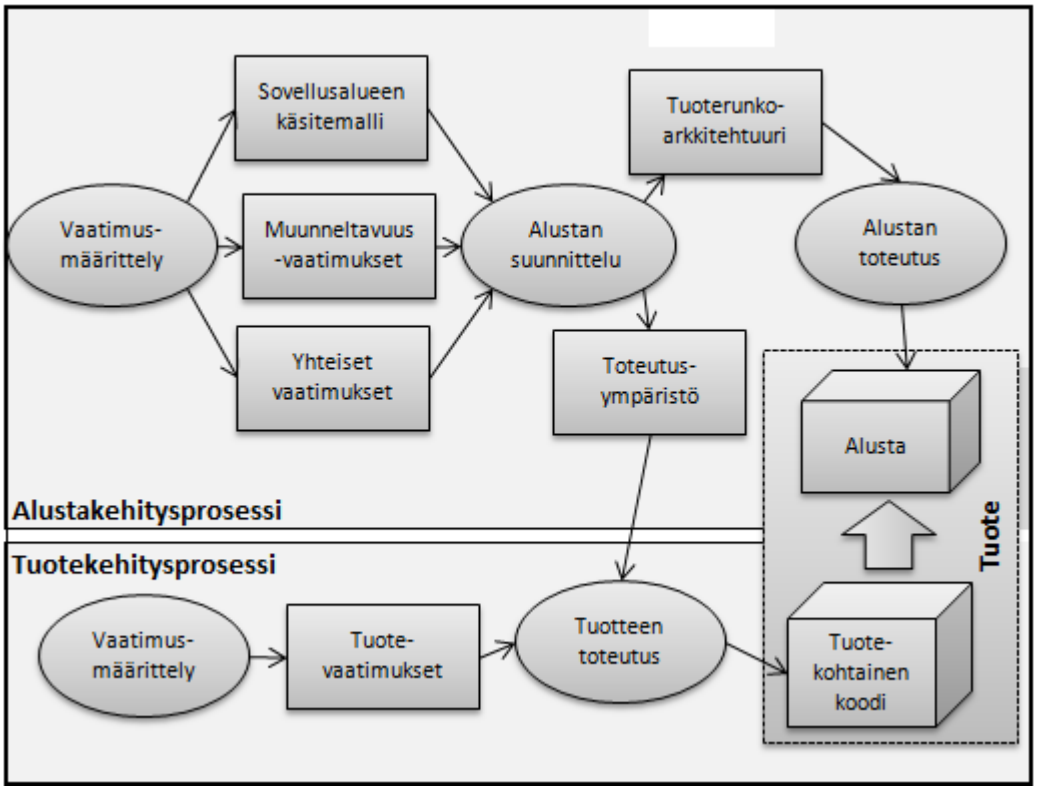
Tuoteperhe on joukko koordinoidusti kehitettyjä ohjelmistotuotteita joilla on samankaltainen rakenne ja toiminta. Tuotelinjaan kuuluvat kaikki artefaktit, välineet ja prosessit, jotka tukevat tuoteperheen jäsenten kehittämistä ja ylläpitoa. Tuoterunko ja tuotealusta tarkoittavat tuoteperheen yhteistä ohjelmistoalustaa. Tuoterunkoarkkitehtuuri ja tuoteperhearkkitehtuuri tarkoittavat tuoteperheen yhteistä arkkitehtuuria.

3.9.1 Tuoterunkopohjainen ohjelmistokehitysprosessi

Jotta tuoterunkojen käytöstä saataisi kaikki edut irti ohjelmiston kehittämisen ja ylläpidon suhteen, täytyy tekniikan lisäksi ajatella toiminnan organisointia ja käytettyjä prosesseja.

Tuoterunkoon pohjautuva ohjelmistonkehitysprosessi jakautuu kahteen osaan (Kuva 3.7): alustakehitysprosessiin, jossa tuloksena tuoterunko, ja tuotekehitysprosessiin, jossa tuloksena on tuoterunkoon pohjautuva yksittäinen

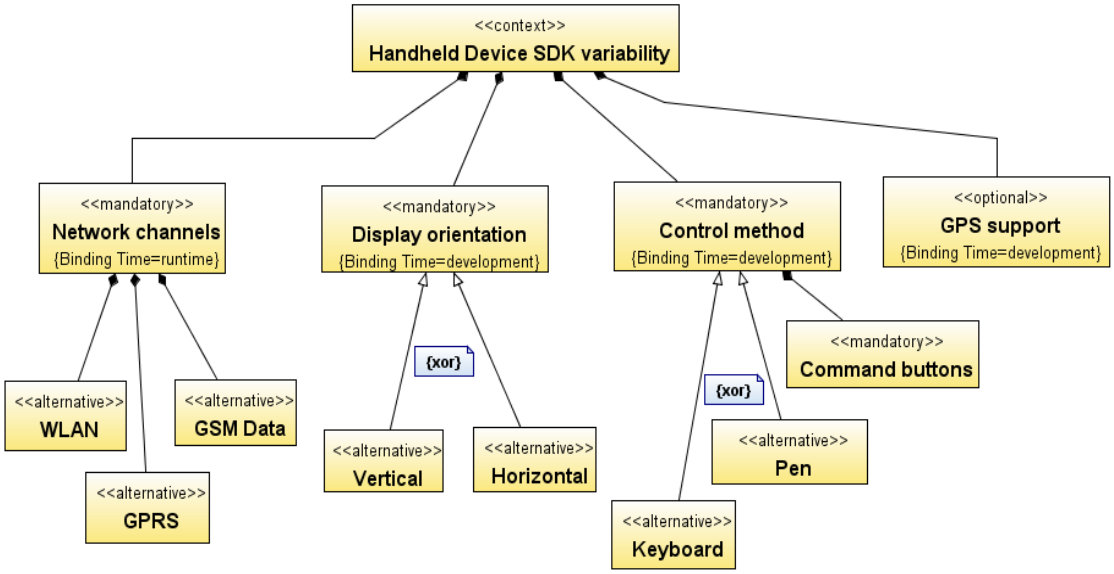
tuote. Näitä edeltää esitutkimusvaihe, jossa selvitetään, onko mitään järkeä kehittää tuoterunkoa. Tähän asiaan vaikuttaa suuresti arvioitu tuoteperheen jäsenten määrä. Jos tullaan siihen tulokseen, että tuoterungon päälle ei tulisi tehtyä montaakaan eri tuotetta, niin on parasta tehdä tuotteet jokainen erikseen.



Kuva 3.7 Tuoterungon ohjelmistokehitysprosessi (Koskimies & Mikkonen, 2005)

3.9.2 Tuoterungon muunneltavuus vaatimustasolla

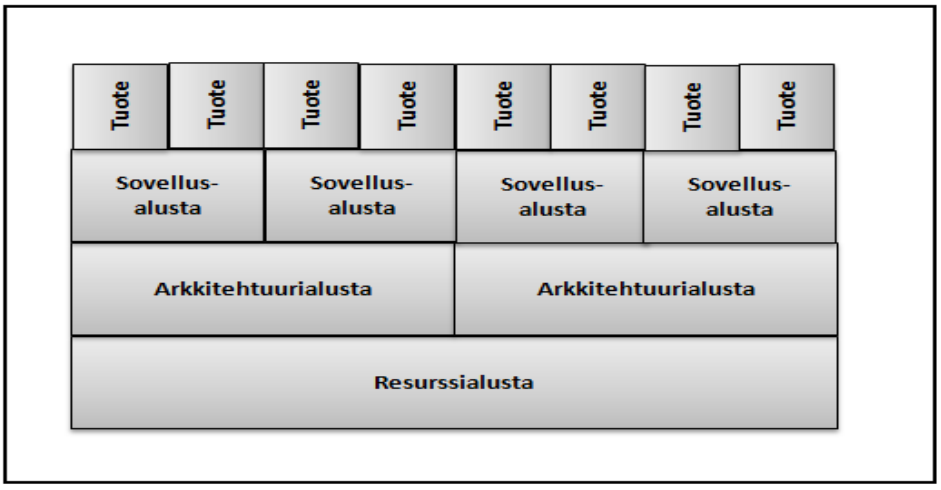
Täsmällisesti tuotteiden yhteisiä ja eroavia ominaisuuksia voidaan esittää nk. piirremallin avulla (Kuva 3.8). Piirremallissa kuvataan, mitkä ominaisuudet ovat tuotteelle pakollisia, mitä valinnaisia ja vaihtoehtoisia ominaisuuksia tuotteella on ja milloin nämä ominaisuudet kiinnitetään tuotteeseen. Kuvan 3.8 PDA-laitteen piirremallista voi huomata, että esimerkiksi ohjauslaitteena voi olla kynä tai näppäimistö mutta ei molempia (xor-rajoite) ja ominaisuus kiinnitetään tuotteeseen kehitysaikana (Binding Time = development).



Kuva 3.8 Muunneltavuuden kuvaus piirremallina käyttäen laajennettua UML-luokkakaavioesitystä (Koskimies & Mikkonen, 2005)

3.9.3 Tuoterunkoarkkitehtuurin kerrosmalli

Tuoterunkojen ongelmana on muunneltavuuden hallinta. Muunteluun liittyvät huolet voidaan eriyttää jakamalla tuoterunkoarkkitehtuuri neljään kerrokseen kerrosarkkitehtuurin idean mukaisesti (Kuva 3.9). Kerrokset ovat resurssi-, arkkitehtuuri- ja sovellusalusta sekä tuotekerros, joista kaikki tarjoavat eri osa-alueisiin varianssia.



Kuva 3.9 Tuoterunkoarkkitehtuuri kerrosmallina (TuTY-kalvot 2008)

3.10 Ohjelmistokehykset

Ohjelmistokehys on ohjelmistorunko, jota voidaan täydentää eri tavoin eri tarkoituksia varten. Sitä ei ole siis tehty valmiiksi, vaan siihen on jätetty aukkoja, joita täydennetään halutulla tavalla. Kehykset ovat suosittu tapa tuoterunkoarkkitehtuurien toteuttamiseen varsinkin oliomaailmassa. Sen tarkoitus on sama kuin tuoterunkoarkkitehtuureilla, uudelleenkäytettävyyden parantaminen.

Kehyksen aukkoja voidaan kutsua myös laajennuskohdiksi (hot spots). Kehystä tehdessä, nämä kohdat täytetään uudella tuotekohtaisella koodilla. Kutsut voivat kulkea molempiin suuntiin kehyksen ja uusien osien välillä. Kehys määrää, mitä vaatimuksia tuotekohtaisen koodin pitää täyttää missäkin aukossa. Kehyksen laajennuskohtia ja niihin liittyviä vaatimuksia tuotekohtaiselle koodille sanotaan erikoistamisrajapinnaksi. Kehyksen erikoistamisrajapinnassa voidaan käyttää erilaisia mekanismeja tuotekohtaisen koodin sitomiseen kehyksen koodiin. Yleisimmät mekanismit ovat periyttäminen, rajapintojen toteuttaminen, parametrointimekanismit, geneeristen ohjelmarakenteiden käyttö, refleksiivisyysominaisuuksien käyttö ja sovelluksen tarjoama muuntelu.

Kehykset luokitellaan yleisesti sen mukaan, mikä on kehyksen pääasiallinen erikoistamismekanismi. Tämä luokittelu ei ole tarkka, koska käytännössä kehykseen voidaan soveltaa useita erikoistamismekanismeja. Yleisempiä kehystyyppejä ovat abstraktit kehykset, muunneltavat kehykset, plugin-kehykset ja koottavat kehykset.

3.11 Ohjelmistoarkkitehtuurien arviointi

Arkkitehtuurin arvioinnissa arvioidaan komponenttien ja alijärjestelmien suhteita ja niiden ominaisuuksia. Lisäksi arkkitehtuurin arvioinnissa tutkitaan, pystyykö järjestelmä täyttämään sille asetetut vaatimukset ja pidemmän tähtäimen

tavoitteet, kuten laajennettavuuden, muunneltavuuden ja skaalautuvuuden ilman, että suorituskyky tai muistinkulutus kärsivät liikaa.

Arviointimenetelmät tarjoavat yleensä vastauksia seuraaviin kysymyksiin:

- Sopiiko suunniteltu arkkitehtuuri järjestelmälle?
- Mikä vaihtoehtoisista arkkitehtuureista soveltuu parhaiten järjestelmälle ja miksi?
- Miten hyvä tulee olemaan järjestelmän jokin tietty laadullinen ominaisuus?

Useat arkkitehtuurin arviointimenetelmät perustuvat skenaariotekniikoihin. Tällöin esitetään konkreettisia esimerkkitilanteita, joissa laatuominaisuudet tulevat esiin. Tämän jälkeen tutkitaan miten arkkitehtuuri soveltuu kyseisiin skenaarioihin. Skenaarioiden etuna on niiden helppo löytäminen, konkreettisuus ja ymmärrettävyys. Skeenariopohjaisia arviointimenetelmiä ovat mm. SAAM, ATAM, MPM ja CBAM.

SAAM (Software Architecture Analysis Method) on kehitetty SEI:ssä (Software Engineering Institute, Carnegie-Mellon University). Se keskittyy erityisesti muunneltavuuteen, siirrettävyyteen, ylläpidettävyyteen ja perustuu evoluutioaikaisiin skenaarioihin.

ATAM (Architecture Tradeoff Analysis Method) soveltuu kaikille laatuominaisuuksille ja on kehitetty SEI:ssä ja se on johdettu SAAM:sta.

MPM (Maintenance Prediction Method) keskittyy ylläpidettävyyteen ja pyrkii löytämään suhteellisen tarkat kustannusarviot ylläpidolle. Sen kehitti Jan Bosch ja se perustuu ylläpitoskenaarioihin.

CBAM (Cost Benefit Analysis Method)on johdettu ATAM:sta ja on tarkoitettu suuremmille järjestelmille. Sen avulla arvioidaan kuinka kalliiksi järjestelmä tulee ja mitä hyötyjä sen tekemisestä on.

Muita arviointimenetelmiä ovat esimerkiksi tarkistuslistat ja kysymyslomakkeet, joissa kysytään arkkitehtuurin tai sen suunnitteluprosessiin liittyviä asioita (esim. ”Onko käyttöliittymä erotettu sovelluslogiikasta”). Näitä voidaan yhdistellä skenaariopohjaisten menetelmien kanssa.

4 TYÖPROSESSIN KUVAUS

Opinnäytetyöni noudatti Saimaan ammattikorkeakoulun laatimaa opinnäytetyöprosessia. Opinnäytetyöprosessi koostuu viidestä vaiheesta, jotka ovat aiheen valinta, suunnittelu, toteutus, raportointi ja työn viimeistely.

Opinnäytetyön aiheen sain Saimaan ammattikorkeakoululta, joka tarvitsi tuoretta tietoa Ohjelmistoarkkitehtuurit ja suunnittelumallit –opintojaksolle.

Kun opinnäytetyön aihe oli valittu, pidettiin aloituspalaveri ja laadittiin projektisuunnitelma ja aikataulu sekä sovittiin tulevat palaverit.

Opinnäytetyöni toteutettiin kirjallisuustutkimuksena ja ensimmäisenä tehtävänä oli tutustua aiheeseen, koska minulla ei ollut mitään erityistä tietoa aihealueesta ennestään. Tätä varten tutkin aiheeseen liittyviä tieteellisiä julkaisuja, kirjoja sekä yliopistojen ja ammattikorkeakoulujen tarjoamia materiaaleja. Muiden koulujen materiaaleja tutkimalla selvitin myös, että mihin asioihin kannattaa keskittyä. Keskeisempiä lähteitä olivat ”Koskimies, K. & Mikkonen T., P. 2005. ”Ohjelmistoarkkitehtuurit” -kirja, Helsingin Yliopiston syksyn 2008, Turun Teknillisen Yliopiston 2008 sekä Tampereen Teknillisen Yliopiston 2006 Ohjelmistoarkkitehtuurit-kurssin materiaalit.

Opintomateriaalin valmistettua aloin tekemään opinnäytetyöraporttia. Raportti sisältää tiivistelmän opintomateriaalista sekä Saimaan ammattikorkeakoulun laatiman opinnäytetyöohjeen mukaiset asiat.

Viimeistelyvaiheeseen kuului pohdinnan kirjoittaminen sekä korjausten tekeminen ohjaavan opettajan korjausehdotusten perusteella.

5 POHDINTA

Alunperin oli tarkoituksena tehdä opetusmateriaalia vain ohjelmistoarkkitehtuureista, mutta opetusmateriaaliin päätettiin lisätä myös osuus suunnittelumalleista, jotta opetusmateriaalia voidaan käyttää myös Ohjelmistoarkkitehtuurit ja suunnittelumallit –opintojaksolla. Oppimateriaali on laajuudeltaan ja tasoltaan samankaltainen muiden yliopistojen ja ammattikorkeakoulujen vastaavan opintojakson opiskelumateriaalin kanssa. Suunnittelumallien osalta materiaali on suppea, koska opintojakson laajuus on vain neljä opintopistettä, joten suunnittelumallien laajempaan käsittelyyn ei luultavasti ole aikaa. Kurssia ei ole tarkoitettu tekemään valmiita arkkitehtejä, vaan tarjoaa perusteet ohjelmistoarkkitehtuureista ja niiden suunnittelusta sekä suunnittelumalleista,

Opintomateriaalin ymmärtäminen vaatii perusymmärrystä ohjelmointimenetelmistä sekä ohjelmistoprojektien etenemisestä. Tämä ei luultavammin tuota ongelmia, koska Saimaan ammattikorkeakoulun opetussuunnitelmassa 2008-2009 tietotekniikan opiskelijat ovat opiskelleet ennen Ohjelmistoarkkitehtuurit ja suunnittelumallit –opintojaksoa 16 opintopisteen edestä ohjelmointia ja yhdeksän opintopisteen edestä ohjelmistotekniikkaa.

Opintomateriaalin avulla saa käsityksen yleisistä ohjelmistoarkkitehtuurien suunnittelumenetelmistä ja yleisistä ohjelmistoarkkitehtuureista. Opintomateriaalista puuttuvat harjoitukset ja niiden ratkaisut. Harjoitukset olisivat hyvä menetelmä syventää oppimista, joten ne olisi hyvä lisätä materiaaliin.

KUVAT

TAULUKOT

LÄHTEET

Haikala, I & Märijärvi J., P. 2004. Ohjelmistotuotanto. Hämeenlinna: Talentum Media Oy.

IEEE 1471-2000, Recommended Practice for Architectural Description for Software- Intensive Systems, http://standards.ieee.org/cgi-bin/status?1471-2000 (luettu 10.07.2009)

Koskimies, K. & Mikkonen T., P. 2005. Ohjelmistoarkkitehtuurit. Jyväskylä: Talentum Media Oy.

Saimaan AMK, Lappeenrannan tietotekniikan opetusohjelma 2005-2006, http://www.saimia.fi/fi-FI/images/docs/opetusohjelmat/lv2005-2006/ops-05-tietotekniikka-lpr.PDF (luettu 14.12.2009)

Saimaan AMK, Lappeenrannan tietotekniikan opetusohjelma 2006-2007, http://www.saimia.fi/fi-FI/images/docs/opetusohjelmat/lv2006-2007/ops-06-tietotekniikka-lpr.PDF (luettu 14.12.2009)

Saimaan AMK, Lappeenrannan tietotekniikan opetusohjelma 2007-2008, http://www.saimia.fi/fi-FI/images/docs/opetusohjelmat/lv2007-2008/ops-07-tietotekniikka.PDF (luettu 14.12.2009)

Saimaan AMK, Lappeenrannan tietotekniikan opetusohjelma 2008-2009, http://www.saimia.fi/fi-FI/images/docs/opiskelu/ops2008/ops-08-tietotekniikka.pdf (luettu 14.12.2009)

Saimaan AMK, Opinto-opas 2008, http://www.saimia.fi/fi-FI/images/docs/opiskelu/oo2008.pdf (luettu 22.12.2009)

Saimaan AMK, organisaatiokaavio, http://www.saimia.fi/fi-FI/images/saimaan_amk_organisaatio.jpg (luettu 05.01.2010)

Suomen virtuaaliyliopisto, Ongelmakeskeinen oppiminen, http://tievie.oulu.fi/verkkopedagogiikka/luku_6/ongelmakeskeinen.htm (luettu 22.12.2009)

TietoEnator, Yritysarkkitehtuuri – Hypeä vai asiaa?, http://www.pcuf.fi/sytyke/syysseminaarit/SS_2005/Yritysarkkitehtuuri_TE.pdf (Luettu 14.12.2009)

TuTY-kalvot, Turun Teknillisen Yliopiston 2008 ohjelmistoarkkitehtuurit -kurssin materiaalit, http://www.cs.tut.fi/kurssit/OHJ-3200/luennot/.

Wikipedia, Kruchten 4 + 1 –malli,
http://en.wikipedia.org/wiki/4%2B1_Architectural_View_Model
(luettu 05.01.2010)

Wikipedia,Tietojärjestelmäarkkitehtuuri,
http://fi.wikipedia.org/wiki/Tietojärjestelmäarkkitehtuuri (Luettu 10.07.2009)

SAIMAAN AMMATTIKORKEAKOULU
Tekniikka, Lappeenranta
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

LIITE 1

Juha-Matti Seppänen

OHJELMISTOARKKITEHTUURIT

ALKUSANAT

Nykyisin ohjelmistot monimutkaistuvat ja laajenevat ja ohjelmistojen uudelleenkäytettävyys on tärkeässä asemassa. Kaikki edellämainittu vaatii huolellista ohjelmistoarkkitehtuurin suunnittelua. Tämä dokumentti selittää alusta lähtien mitä ohjelmistoarkkitehtuurit ovat ja mitä siihen liittyvät käsitteet tarkoittavat. Tämä dokumentti on tarkoitettu tietotekniikan opiskelijoille ja ei vaadi mitään erikoista tietoa tai osaamista ohjelmistoarkkitehtuurien osalta. Tässä dokumentissa käydään läpi mitä ohjelmistoarkkitehtuurit ovat ja miten ne dokumentoidaan. Lisäksi käydään läpi komponentit, rajapinnat, suunnittelumallit sekä erilaisia arkkitehtuurityylejä ja lopulta kuinka ohjelmistoarkkitehtuureja voidaan arvioida. Kun olet sisäistänyt tämän dokumentin asiat, on sinulla perusymmärrys ohjelmistoarkkitehtuureista.

SISÄLLYSLUETTELO

TERMIT JA LYHENTEET

prototyyppi	Ensimmäinen testiversio, jolla voidaan testata yleistä suunnitelman toimivuutta.
sidosryhmä	Esimerkiksi ohjelmistoa kehittäessä sidosryhmiä ovat mm. asiakas, loppukäyttäjät, projektiryhmä. Jokaisella ryhmällä on omia tavotteitansa.
inkrementaali	Tarkoittaa että jotakin tehdään askel kerrallaan.
komponentti	Itsenäisesti toimiva ohjelmistoarkkitehtuurin osa, jolla on hyvin määritellyt rajapinnat.
rajapinta	Komponenttien kohdalle määrittelee kuinka ja kuka voi käyttää sen palveluja ja mitä palveluja komponentti tarvitsee.
artefakti	Ohjelmiston teossa syntyvä tuotos, esimerkiksi dokumentit, luokkakaaviot yms.
arkkitehtuuriviipale	Järjestelmän pääositusperusteen kanssa eriävä mutta jonkin kriteerin perusteella loogisesti yhteenkuuluva rakenne.
periyttäminen	Periyttämisen ansiosta voidaan käyttää uudestaan jo tehtyä ohjelmakoodia uusien luokkien pohjana. Periyttäminen mahdollistaa abstraktien perusluokkien käytön, joista periytetään useita luokkia, joita kaikkia voidaan käyttää samannimisillä funktioilla ja samoilla parametreilla kuin abstraktia perusluokkaa.

algoritmi	Tarkasti määritelty vaihesarja, jota seuraamalla voidaan ratkaista tietty ongelma.
pseudokoodi	Ohjelmointikielen tapaista koodia, jossa ei oteta huomioon syntaksia vaan halutaan näyttää pelkkä algoritmi.
abstrakti	Abstrahoimalla saatu yleiskäsite.
abstrahointi	Yleiskäsitteen muodostaminen pelkistämällä.
käyttöskenaario	Jokin tapahtuma järjestelmässä, esim. käyttäjän tietojen muokkaaminen.
alijärjestelmä	Pienempi kokonaisuus järjestelmässä.
roolirajapinta	Rajapinta, joka toimii eri rooleissa.
delegointi	Tehtävän siirtäminen toisen vastuulle.
suunnittelumalli	Ratkaisu yleiseen suunnitteluongelmaan.
antisuunnittelumalli	Huono ratkaisu suunnitteluongelmaan.
EJB	(Enterprise JavaBean) Javan komponenttipohjainen arkkitehtuuri.
CORBA	(Common Object Request Broker Architecture) CORBA perustuu asiakas-palvelin -arkkitehtuuriin, jossa palvelinoliot tarjoavat palveluita asiakasolioille.

MVC	(Model-View-Controller) on arkkitehtuurityyli, jonka tarkoituksena on erottaa käyttöliittymä sovelluslogiikasta.
SQL	(Structured Query Language) standardoitu kyselykieli, jolla voi tehdä erilaisia hakuja, muutoksia ja lisäyksiä tietokantaan.
API	(Application programming interface) Ohjelmointirajapinta on käyttöliittymä, jolla eri ohjelmat voivat tehdä pyyntöjä ja vaihtaa tietoja eli keskustella keskenään. Esimerkiksi ohjelmat tarvitsevat käyttöliittymältä luvan käyttää keskusmuistia ja tiedostoja.
C++	Ohjelmointikieli
Java	Ohjelmointikieli
Swing	Javan graafisen käyttöliittymän luontiin tarkoitettu kirjasto.
AWT	Javan graafisen käyttöliittymän luontiin tarkoitettu kirjasto.
XML	(Extensible Markup Language) on standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan. XML-kieltä käytetään sekä formaattina tiedonvälitykseen järjestelmien välillä että formaattina dokumenttien tallentamiseen. XML-kieli on rakenteellinen kuvauskieli, joka auttaa jäsentämään laajoja tietomassoja selkeämmin.

1. JOHDANTO

Ohjelmistoarkkitehtuurit ovat muodostuneet omaksi ohjelmistotekniikan alueekseen vasta 1990-luvun loppupuolella. Ennen sitä ohjelmistoarkkitehtuureja pidettiin lähinnä korkeamman tason suunnitteluna. Nykyisin ohjelmistoarkkitehtuurien merkitys ohjelmistotekniikan alueena kasvaa jatkuvasti, koska nykyisessä tietoyhteiskunnassa lukematon määrä palveluita toteutetaan tietojärjestelminä, jotka kasvavat ja monimutkaistuvat jatkuvasti. Tästä syystä tietojärjestelmien suunnittelussa täytyy kiinnittää huomiota hallittavuuteen, ylläpidettävyyteen sekä uudelleenkäytettävyyteen. Tämän vuoksi tarvitaan arkkitehtuurisuunnittelua.

Arkkitehtuurisuunnittelun osa ohjelmistokehityksessä

Varsinkin laajojen ohjelmistojen yhteydessä arkkitehtuurin suunnittelu on todella tärkeää. Kun järjestelmä saadaan suunnitteluvaiheessa jaettua itsenäisiin osiin, voidaan myös työnjako tehdä tämän pohjalta. Itsenäiset osat tarkoittavat myös helpompaa testausta ja ylläpitoa, koska kaikki osat voidaan testata/päivittää erikseen. Arkkitehtuuri antaa korkean abstraktiotason näkymän ohjelmistoon, mikä mahdollistaa aikaisempaa monimutkaisempien järjestelmien tarkastelun ja eri sidosryhmien välisen kommunikoinnin havaitsemisen järjestelmässä. Etuna arkkitehtuurin olemassaolosta on myöskin se, että arkkitehtuurin kunnollinen määrittely sekä dokumentointi mahdollistaa järjestelmän arvioinnin joidenkin kriittisten tekijöiden osalta jo varhaisessa vaiheessa. Jos tällöin huomataan että järjestelmää täytyy muuttaa, niin muutostyö on vielä verrattain halpaa.

Ohjelmistoarkkitehtuuri

1.1.1 Arkkitehtuurin määritelmä

Ohjelmistoarkkitehtuurille on olemassa monia määritelmiä:

IEEE:n arkkitehtuurien kuvaamista koskevassa standardissa ohjelmistoarkkitehtuuri määritellään järjestelmän perusorganisaatioksi, joka sisältää järjestelmän osat, niiden väliset suhteet ja niiden suhteet ympäristöön, sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua. (IEEE 1471-2000)

”Tietojärjestelmän arkkitehtuuri kuvaa kohdealueensa rakenneosat, niiden ulospäin näkyvät ominaisuudet ja niiden väliset yhteydet ja riippuvuudet. Arkkitehtuuri muodostaa rungon järjestelmän suunnittelulle ja toteutukselle sekä ohjaa järjestelmän rakenteen kehittämistä järjestelmän elinkaaren ajan. Se toimii myös keskusteluvälineenä järjestelmän kehittämisen ja ylläpitämisen sidosryhmien (organisaation johto, käyttäjät, suunnittelijat, toteuttajat) välillä.” (Wikipedia, Tietojärjestelmäarkkitehtuuri)

”Voidaankin ajatella, että arkkitehtuuri on järjestelmän perustuslaki. Sitä on noudatettava järjestelmää rakennettaessa, ja sitä voidaan muuttaa vain erittäin painavilla perusteilla..... Toisin sanoen arkkitehtuuri määrittelee rajat, joiden puitteissa järjestelmää on rakennettava ja ylläpidettävä.” (Koskimies, Mikkonen, 2005)

Yhteistä määritelmissä on se, että arkkitehtuuri kattaa järjestelmän jakamisen osiin, osien väliset yhteydet, mutta ei osien sisäisiä asioita. Arkkitehtuuri määrittelee järjestelmän ”ytimen”, perustan, sen täytyy pysyä olennaisesti samana kehityksen ja ylläpidon aikana. Osat, jotka eivät kuulu ”ytimeen”, ovat vapaasti muunneltavissa milloin mihinkin tarkoitukseen. Tämä ajatusmaailma sopii nk. tuoterunkoarkkitehtuureihin, tällöin vaihtuvat osat vastaavat sovelluskohtaisia osia.

1.1.2 Arkkitehtuurin tehtävät

Arkkitehtuurin tehtävänä on ottaa kantaa ohjelmiston keskeisiin ratkaisuihin, kuten:

- Miten järjestelmä jaetaan osiin?
- Miten osat kommunikoivat keskenään?
- Mitä prosesseja järjestelmässä on ja mitä yhteyksiä niiden välillä on?
- Kuinka tiedot saadaan?
- Mitä vastuita kullakin osalla on?

- Järjestelmän fyysinen rakenne?
- Järjestelmän tehokkuus?
- Päivitys mahdollisuudet?
- Uudelleenkäytettävyys?
- Käytettävä teknologia?

Arkkitehtuuri siis toimii yleissuunnitelmana, joka helpottaa monimutkaisen järjestelmän hallitsemista. Arkkitehtuurissa jaetaan järjestelmä osiin ja selvitetään osien väliset suhteet ja tehtävät.

Monia arkkitehtuurissa käytettäviä ratkaisuja voi olla vaikea arvioida etukäteen, joten arkkitehtuurin pohjalta on hyvä tehdä alussa prototyyppi, jossa testataan järjestelmältä vaadittavat ydintoiminnot. Tällöin on mahdollista huomata, jos järjestelmälle suunnitellussa arkkitehtuurissa on vakavia puutteita.

1.1.3 Arkkitehtuurin dokumentointi

”Jos arkkitehtuuria ei ole selkeästi dokumentoitu, sitä ei ole oikeastaan olemassa: arkkitehtuuri materialisoituu kuvauksena.” (Koskimies, Mikkonen, 2005)

Jos järjestelmän arkkitehtuuria ei ole dokumentoitu, seuraksena ennen pitkää on se, että järjestelmän alkuperäinen, hyvä, selkeä, toimiva arkkitehtuuri, tulee rapautumaan. Myöskin arkkitehtuurista keskusteltaessa eri ihmiset voivat tehdä eri päätelmiä, arkkitehtuurikuvaukset selventävät asian ja näin kaikilla on sama käsitys arkkitehtuurista. Periaatteessa mitään ohjelmistoprojektia ei pitäisi päästää etenemään, jos arkkitehtuuria ei ole selkeästi kuvattu.

Arkkitehtuuridokumentteja ovat

- alustava arkkitehtuuridokumentti
- järjestelmäarkkitehtuuridokumentti
- alijärjestelmäarkkitehtuuridokumentti
- tuoterunkoarkkitehtuuridokumentti
- tuotearkkitehtuuridokumentti

- rajapintadokumentti

Arkkitehtuuridokumentteja käsitellään tarkemmin luvussa 2.

Huono arkkitehtuurisuunnittelu ja sen seuraukset

Epäonnistuneesta järjestelmästä voidaan usein syyttää arkkitehtuuria. Huono arkkitehtuuri aiheuttaa tyypillisesti mm. aikatauluongelmia, vaatimusten täyttämättömyyttä, huonon suorituskyvyn sekä testaus- , ylläpito- ja uudelleenkäyttöongelmia.

Aikatauluongelmat syntyvät esim. jos joudutaan muuttamaan eri osien vastuualueita kesken työn. Tästä seuraa uutta suunnittelua ja mahdollisesti viivästyksiä, kun muut osat joutuvat odottamaan osaa, joka täytyy suunnitella ja toteuttaa uusiksi. Myöskin liian monimutkainen tai joustava arkkitehtuuri ja väärät olettamukset teknologian suhteen aiheuttavat ongelmia aikataululle.

Viimeistään testausvaiheessa voidaan havaita, että huonon arkkitehtuurin omaava järjestelmä täyttää huonosti asiakkaan keskeiset vaatimukset tai pahimmassa tapauksessa järjestelmä ei täytä niitä ollenkaan. Syynä tähän on yleensä se, että ei ole täysin ymmärretty määrittelyvaiheessa, mitä asiakas haluaa, eikä suunnitteluvaiheessa sitä, että mitkä ohjelmiston osat oikein toteuttavat kyseiset vaatimukset. Kehno arkkitehtuuri voi aiheuttaa sen, että järjestelmä ei pysty suoriutumaan kuin vain pienistä tieto- tai kävijämääristä, tai tekee järjestelmästä niin hitaan, että sitä ei ole järkevää käyttää.

Järjestelmän testaus ja ylläpito voi olla kallista ja vaikeaa, jos järjestelmä ei tue eri osien yksittäistä testausta tai muutostöitä. Jos arkkitehtuurin suunnittelussa ei oteta huomioon tulevia muutos- ja uudelleenkäyttötarpeita, niin järjestelmän päivittäminen tai uuden version tekeminen järjestelmästä vaatii paljon koodin uudelleenkirjoittamista ja johtaa raskaaseen rinnakkaisten toteutusten ylläpitoon.

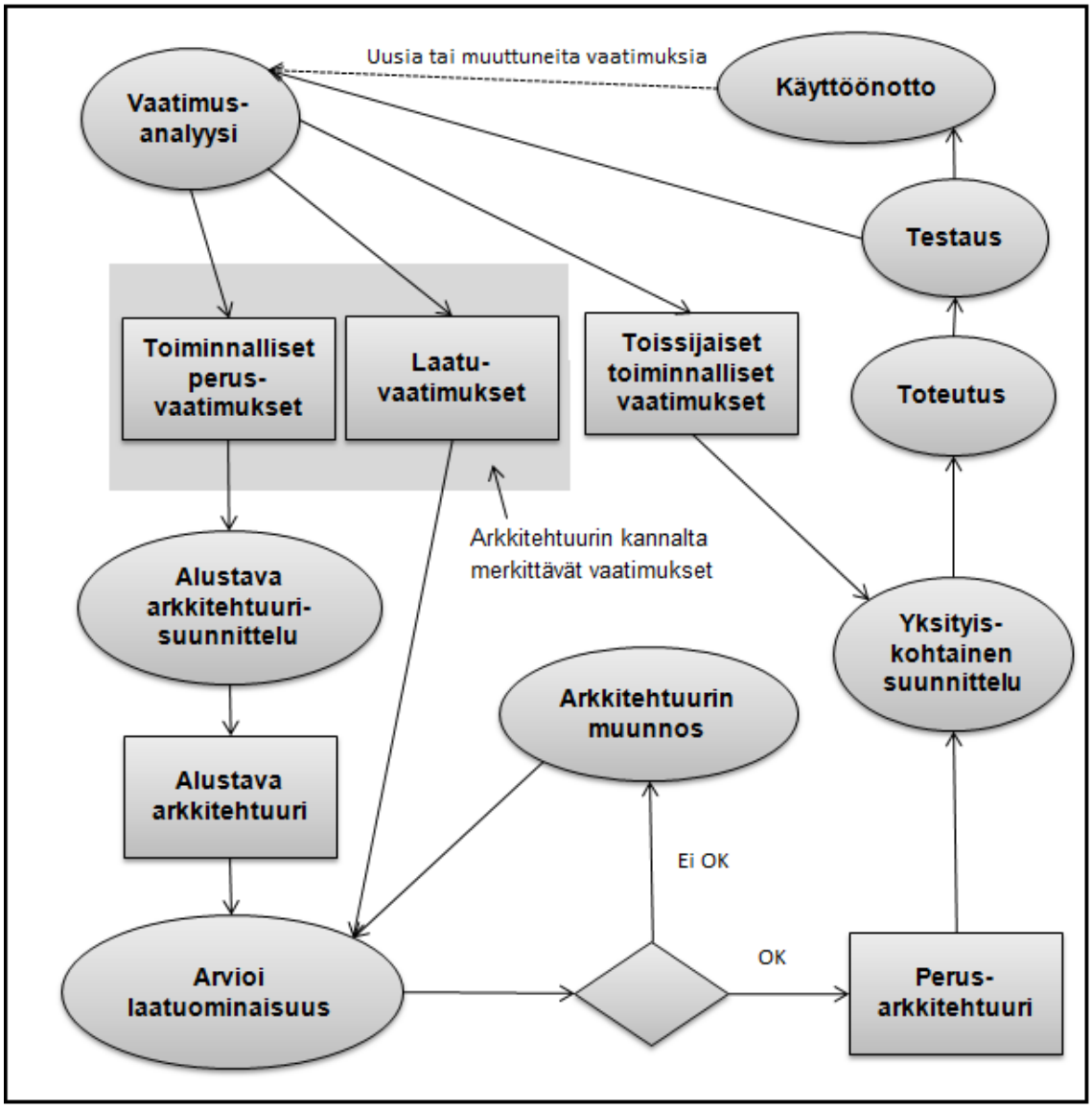
Arkkitehtuuripainotteinen ohjelmistokehitysprosessi

Arkkitehtuuripainoitteisessa ohjelmistonkehitysprosessissa korostetaan arkkitehtuurin suunnittelua ja arviointia keskeisenä vaiheena ennen siirtymistä yksityiskohtaiseen suunnitteluun ja toteutukseen. Arkkitehtuuri pyritään luomaan iteratiivisesti arkkitehtuurin kannalta tärkeistä vaatimuksista.

Arkkitehtuuriin vaikuttavat toiminnalliset sekä laadulliset vaatimukset. Yleisesti ottaen prosessi etenee tyypillisesti siten, että ensimmäinen versio arkkitehtuurista tehdään toiminnallisten vaatimusten pohjalta, tätä versiota verrataan laadullisia vaatimuksia vasten. Jos tarvetta tulee, arkkitehtuuria muutetaan niin, että myös laadulliset vaatimukset täyttyvät.

Kun myös laadulliset vaatimukset on saatu täytettyä, järjestelmällä on perusarkkitehtuuri. Tästä sitten edetään yksityiskohtaisempaan suunnitteluun, toteutukseen ja testaukseen, josta saadaan ensimmäinen, keskeisimmät toiminnalliset vaatimukset täyttävä versio. Tämän jälkeen tarkastellaan sekundaarisia vaatimuksia ja toteutetaan ne perusarkkitehtuurin pohjalle. Tässä vaiheessakin muutoksen tarve voidaan havaita.

Tämän jälkeen versiota tehdään inkrementaalisti vaatimus kerrallaan. Jos sovelluksen käyttöönoton jälkeen havaitaan tarve uusille, tai jotain vaatimusta täytyy muuttaa, joudutaan koko prosessi periaatteessa käymään uudestaan läpi. Jos vaadittavat muutokset ovat suuria, voidaan joutua muuttamaan, tai ainakin arvioimaan koko arkkitehtuuria uusiksi.



Kuva 1.1 Arkkitehtuuripainotteinen ohjelmistokehitysprosessi (Koskimies, Mikkonen, 2005)

Arkkitehtuuri toteutusvälineenä

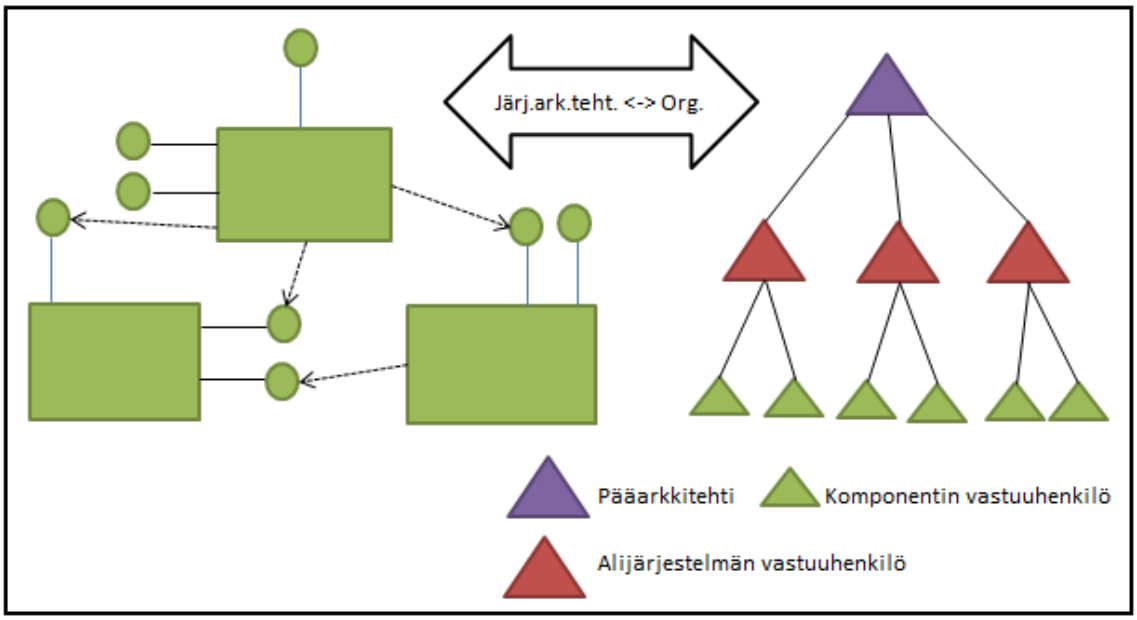
Yleisesti pyritään uudelleenkäyttämään ohjelmistoja, tästä seuraa se, että uudet ohjelmistot toteutetaan perustuen yhä useammin jo olemassaolevaan arkkitehtuuriin. Tämä ilmenee erilaisina ohjelmistoalustoina. Kun ohjelmistoalustalla on arkkitehtuuri, joka täytyy ottaa huomioon, arkkitehtuurille tulee toteutusvälineen rooli, joka on verrattavissa ohjelmointikielen rooliin. Arkkitehtuurin tarjoamat välineet ovat vain korkeammalla tasolla kuin ohjelmointikielen välineet. Sovelluskehittäjällä on täten tehtävänä ratkaista

miten sovelluksen vaatimukset täytetään alustan tarjoamalla arkkitehtuurilla, eikä se, miten vaatimukset toteutetaan tietyllä ohjelmointikielellä.

Arkkitehtuurin ja sitä kehittävän organisaation yhteys

Yleensä järjestelmän arkkitehtuuri ja sitä kehittävän ja ylläpitävän organisaation rakenteet alkavat muistuttaa toisiaan. Tämä johtuu siitä, että tyypillisesti arkkitehtuurin perusyksikkö, eli komponentti, annetaan yhden henkilön tehtäväksi. Toisiinsa liittyvät komponentit, jotka muodostavat jonkin alijärjestelmän, pyritään taas antamaan ryhmälle, joka muodostuu vaadittavien komponenttien kehittäjistä.

Tälläinen järjestelmän arkkitehtuurin ja organisaation rakenteen heijastuminen toisiinsa tulee selvästi näkyviin varsinkin tuoterunkoarkkitehtuurien kohdalla. Ohjelmistoalustasta vastaa yleensä oma yksikkö, kun taas tuotekohtaisista ohjelmistoista vastaa toinen yksikkö.



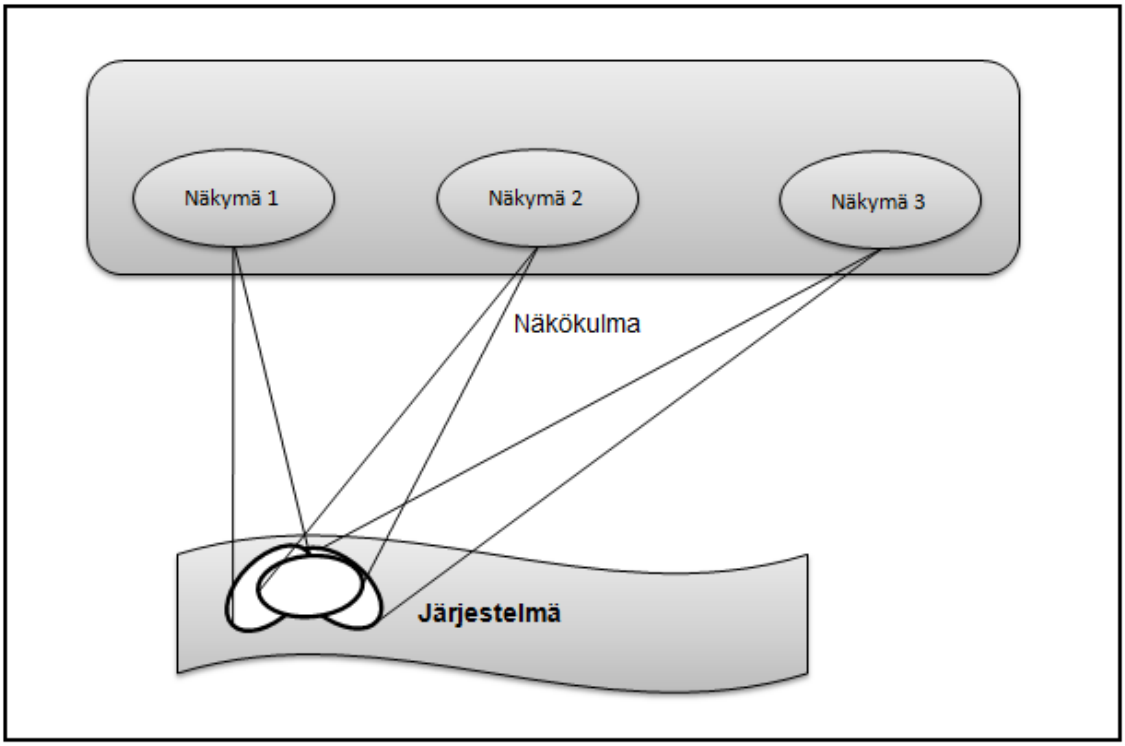
Kuva 1.2 Järjestelmän arkkitehtuurin ja organisaation yhteys (HY-kalvot 2008)

Ohjelmiston kehitykseen liittyy useita sidosryhmiä (kehittäjät, ylläpitäjät, testaajat, käyttäjät), joilla kaikilla on omat näkökulmansa ja vaatimuksensa ohjelmistoon. Nämä eri sidosryhmien vaatimukset ovat usein ristiriidassa

keskenään, ja arkkitehdin tehtäväksi jää määritellä ohjelmistolle arkkitehtuuri, joka tuottaa siedettävän kompromissin, jossa tarpeeksi monen sidosryhmän tarpeeksi moni vaatimus täyttyy.

Arkkitehtuurinäkymät

Arkkitehtuureja voidaan tarkastella eri näkökulmista, tällöin jokainen näkökulma tuottaa näkymän, ja kukin näkymä mallintaa jonkin asian järjestelmästä. Eri näkymillä on luonnollisesti paljon riippuvuuksia, ja jos jotain kohtaa muutetaan jossakin näkymässä, seuraa siitä muutoksia muihin näkymiin. Yhdessä nämä näkymät muodostavat kattavan kuvauksen järjestelmän arkkitehtuurista. Asiaa on havainnollistettu kuvassa 1.3.



Kuva 1.3 Arkkitehtuurinäkymiä (Koskimies, Mikkonen, 2005)

Yhteenveto luvusta

- Arkkitehtuuriin kuuluvat komponenttien väliset suhteet, mutta eivät komponenttien sisäiset asiat.

- Arkkitehtuurien ratkaisut pitää myös perustella.

- Arkkitehtuuri on suunniteltavan ohjelmiston perustuslaki, joka säätää rajat suunnitteluun ja toteutukseen.

- Jos arkkitehtuuria ei ole dokumentoitu, sitä ei ole oikeastaan olemassa.

- Arkkitehtuuria voidaan kuvata eri näkökulmista, jolloin kiinnitetään huomioita eri asioihin.

2. OHJELMISTOARKKITEHTUURIN KUVAUS

Ohjelmiston arkkitehtuurikuvaus on tärkeä artefakti, koska siinä arkkitehtuuri konkretisoituu ja saadaan yhteisymmärrys arkkitehdin ja toteuttajien välille. Lisäksi se tarjoaa järjestelmää koskevan käsitteistön ja sanaston, joka taas selkeyttää keskustelua järjestelmästä. Tässä luvussa käydään läpi arkkitehtuurikuvaus ja siihen liittyvät eri näkökulmat sekä käsitteet, eri arkkitehtuurikuvauksen tyypit, arkkitehtuuriviipaleet ja erilaisia ohjelmistoarkkitehtuuridokumentteja.

Arkkitehtuurikuvauksen merkitys

Arkkitehtuurikuvaus antaa järjestelmästä tietoa, jota ei muualta saa. Jos järjestelmän toteutus on ristiriidassa arkkitehtuurikuvauksen kanssa, on järjestelmä toteutettu väärin. On siis järkevää ajatella, että arkkitehtuuri realisoituu arkkitehtuurikuvauksessa, ei järjestelmässä.

Ohjelmiston arkkitehtuuri on tärkein ohjelmistoa luonnehtiva informaatio. Tästä syystä useimmat ohjelmistoon liittyvien sidosryhmien kysymykset koskevat arkkitehtuuria. On siis tärkeää että ohjelmistolla on kattava ja selkeä kuvaus sen arkkitehtuurista, jotta kaikilla on sama käsitys järjestelmästä. Arkkitehtuurilla on siis tärkeä merkitys järkevän kommunikaation mahdollistavana ohjelmistoartifaktina. Arkkitehtuuri ei ainoastaan kuvaa sitä, miten keskeiset tehtävät ratkaistaan, vaan tarjoaa myös järjestelmää koskevan käsitteistön ja sanaston, jolloin järjestelmästä voidaan puhua oikein termein. Teollisuudessa on nykyisin ymmärretty arkkitehtuurikuvauksen merkitys, ja siitä on tullut tärkeä dokumentti ohjelmistonkehitysprosessissa.

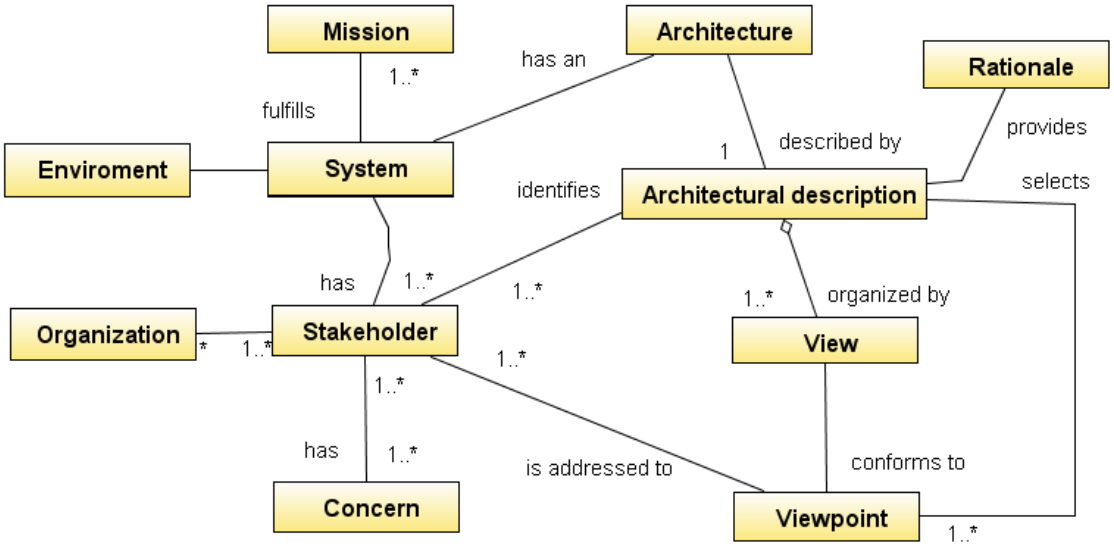
Arkkitehtuuri konkretisoituu arkkitehtuurikuvauksessa, joten jos kuvausta ei ole, ei arkkitehtuuriakaan ole olemassa, vaikka itse järjestelmä on olemassa. Tällöin

järjestelmän arkkitehtuurista on vaarallista keskustella, koska eri ihmiset voivat saada eri käsityksiä olemassaolevan järjestelmän perusteella, ja tehdä omia päätöksiään siitä, mikä on järjestelmän arkkitehtuuri ja mitkä asiat siihen kuuluvat. Varsinaisen rakenteen lisäksi arkkitehtuuri selittää itse järjestelmää.

Arkkitehtuurikuvaus on myöskin tärkeä osa arkkitehdin ja toteutusryhmän välistä kommunikointia. Jos arkkitehtuuri on vain arkkitehdin päässä, on toteutusryhmän vaikea olla täysin perillä siitä, millainen arkkitehtuurin tarkasti ottaen pitäisi olla. On siis tärkeää, että arkkitehti tarjoaa ryhmälle tarkan kuvauksen arkkitehtuurista mahdollisimman pian, koska ilman kuvausta, toteutusryhmä luultavammin tekee omia ratkaisujaan. Nämä ratkaisut taas saattavat sotia tätä arkkitehdin ajatellun arkkitehtuurin periaatteita vastaan, ja joudutaan tekemään muutoksia, jotka maksavat rahaa, ja kuten aikaisemmin todettu, mitä myöhemmin muutoksia tulee järjestelmään, sitä kalliimmaksi se käy.

Arkkitehtuurikuvaukseen liittyvät käsitteet

IEEE on julkaissut standardin (IEEE 1471-2000), jossa käsitellään arkkitehtuurinkuvausta. Standardissa esitetään arkkitehtuureja käsittelevä käsitekaavio, josta seuraavana (Kuva 2.1) on piirretty yksinkertaistettu versio.

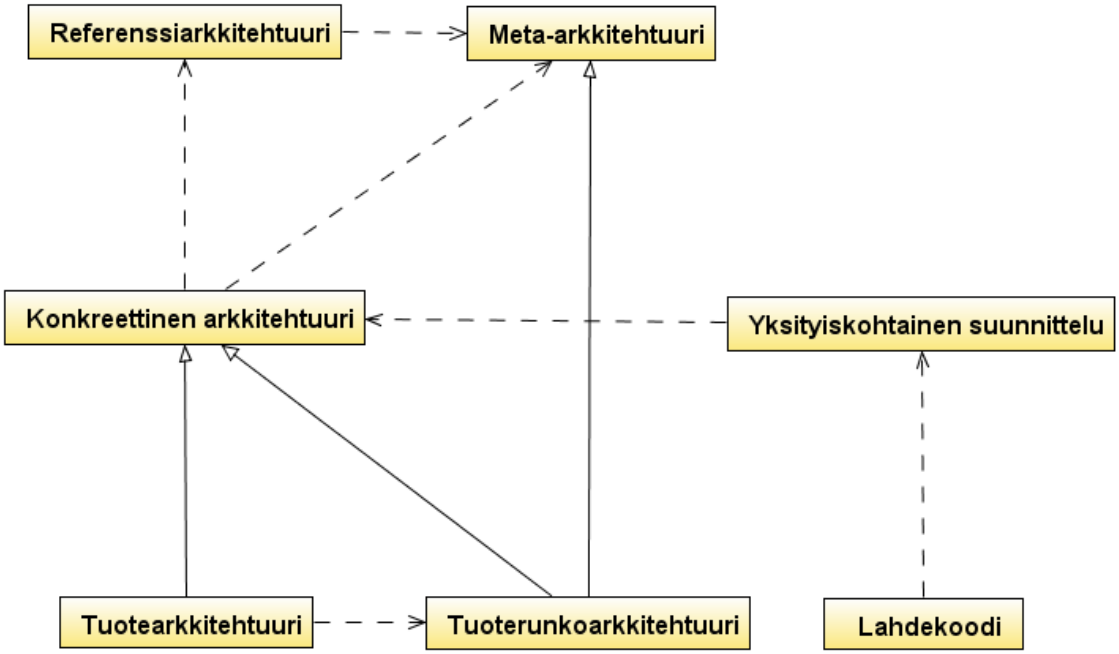


Kuva 2.1 IEEE:n käsitemalli arkkitehtuureille (Koskimies, Mikkonen, 2005)

Kuva lyhyesti selitettynä: Järjestelmä täyttää yhden tai useamman tehtävän toimintaympäristössään. Järjestelmällä on sidosryhmiä, joilla kullakin on omat tavoitteensa ja huolensa järjestelmästä. Järjestelmällä on arkkitehtuuri ja arkkitehtuurista on yksi kuvaus. Arkkitehtuurikuvaus tarjoaa perustelut ratkaisuille, ja muodostuu erilaisista näkymistä, jotka seuraavat tiettyjä näkökulmia.

Arkkitehtuurikuvauksen abstraktiotaso

Aivan kuten tietojenkäsittelytehtävä voidaan kuvata eri abstraktiotasoilla, esim. epäformaalina algoritmina, pseudokoodina tai suoritettavana ohjelmakoodina, voidaan ohjelmistoarkkitehtuuri antaa esim. yleisenä, abstraktina ratkaisumallina tietyn tyyppisille järjestelmille, jonkin järjestelmäperheen yhteisen arkkitehtuurin kuvauksena tai yksittäisen konkreettisen järjestelmän kuvauksena. Kun arkkitehtuurikuvausta tehdään, on syytä tehdä selväksi, minkä tyyppisestä järjestelmästä on kyse. Kuvassa 2.2 on esitetty eri arkkitehtuurityypit ja niiden väliset suhteet. Katkonuoli kuvassa tarkoittaa, että lähde noudattaa kohdetta.



Kuva 2.2 Arkkitehtuurityypit ja niiden väliset suhteet (Koskimies, Mikkonen, 2005)

2.1.1 Arkkitehtuurityypit

- Meta-arkkitehtuuri
- Referenssiarkkitehtuuri
- Konkreettinen arkkitehtuuri
- Tuoterunkoarkkitehtuuri
- Tuotearkkitehtuuri

Meta-arkkitehtuuri ei kuvaa ohjelmistoarkkitehtuuria itseään, vaan siihen liittyvät käsitteet ja mekanismit, joilla varsinaisia arkkitehtuurikuvauksia annetaan. Meta-arkkitehtuuri voi näin kuvata esim. komponenttikategorioita ja niiden välisiä suhteita.

Referenssiarkkitehtuuri ei myöskään kuvaa mitään konkreettista järjestelmää, vaan antaa yleisen ratkaisumallin tietyn tyyppisten järjestelmien tai niiden osien arkkitehtuureille. Toisin kuin meta-arkkitehtuuri, referenssiarkkitehtuuri kuvaa

arkkitehtuuriratkaisun, mutta abstraktilla tasolla liittämättä sitä mihinkään todelliseen järjestelmään.

Konkreettinen arkkitehtuuri esittää yksittäisen ohjelmiston arkkitehtuurin. Esimerkiksi tietyn sovelluksen arkkitehtuuridokumentti antaa näin aina konkreettisen arkkitehtuurin, mutta siinä voidaan viitata eri referenssi- ja meta-arkkitehtuureihin, joita konkreettinen arkkitehtuuri noudattaa.

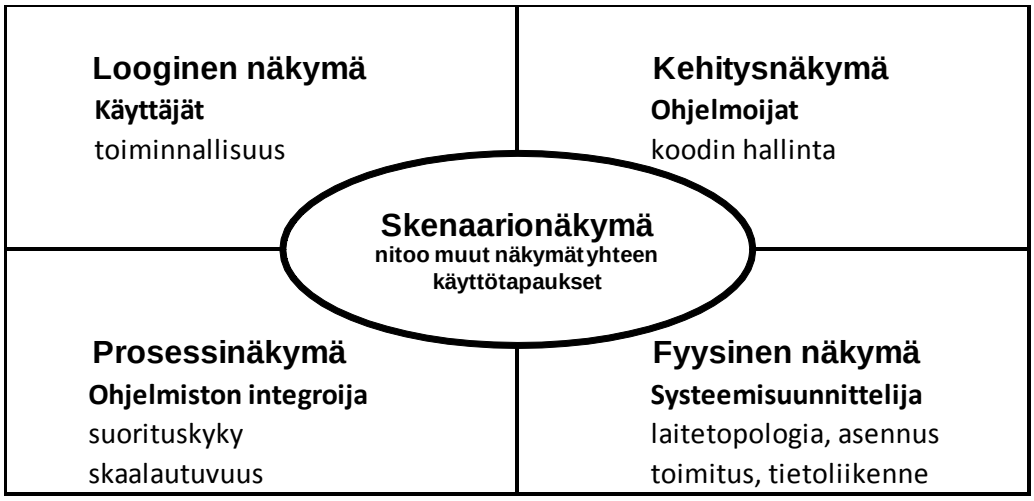
Tuoterunkoarkkitehtuuri kuvaa ohjelmistoalustan arkkitehtuurin ja antaa säännöt yksittäisten ohjelmistojen rakentamiseen alustan pohjalle.

Tuotearkkitehtuuri on jonkin tietyn ohjelmiston arkkitehtuuri. Tälläinen tuote voi olla rakennettu tuoterungon päälle, jolloin se nuodattaa rungon periaatteita, tai sitten se voi olla täysin oma itsenäinen sovellus.

Näkökulmat arkkitehtuurikuvauksessa

Kuten aikaisemmin sanottu, ohjelmistoarkkitehtuurin tulee kertoa, millaisia komponentteja järjestelmässä on ja mitä suhteita niiden välillä on. Näitä suhteita on vaikea selvittää yleisen määritelmän perusteella, joten tarvitaan eri näkökulmia arkkitehtuurin tutkimiseen.

Yleinen tapa kuvata ohjelmistoarkkitehtuuria eri näkökulmista on nk. 4 + 1 – malli (Kruchten 1995)(Kuva 2.3).



Kuva 2.3 4 + 1 -malli

Skenaarionäkymässä tarkastellaan järjestelmän vuorovaikutusta ulkopuolisiin käyttäjiin ja järjestelmiin, se siis kuvaa järjestelmän rajapinnat ympäristöönsä. Skenaarionäkymässä käsitellään yleensä järjestelmän keskeisiä toimintoja, joten se toimii lähtökohtana muiden näkymien muodostamiseen. Muita näkymiä voidaan myös arvioida skenaarionäkymää vasten, jotta nähdään miten näkymässä esitetyt ratkaisut tukevat kyseistä käyttöskenaariota. Skenaarionäkymä on olennainen lähes kaikille järjestelmille.

Loogisessa näkymässä kuvataan, kuinka järjestelmän toiminnallisuus on jaettu eri osien kesken ja mitä vastuita eri osilla on. Looginen näkymä kuvaa skenaarionäkymässä esitetyn käyttötapauksen eri osien välisenä yhteistyönä. Loogista näkymää käytetään yksityiskohtaisen suunnittelun pohjana, ja se on tärkeä näkymä lähes kaikille järjestelmille.

Prosessinäkymässä kuvataan kuinka järjestelmän toiminnot on jaettu loogisiin prosesseihin ja prosessien kommunikointiin. Prosessinäkymää tarvitaan varsinkin järjestelmiin, joissa vaaditaan rinnakkaisuutta. Prosessinäkymää käytetään arvioimaan suorituskykyä ja skaalautuvuutta.

Kehitysnäkymä kuvaa kuinka järjestelmä on jaettu osiin, jotka voidaan toteuttaa erillisinä yksikköinä. Kehitysnäkymää tarvitaan varsinkin suurissa

ohjelmistoissa, ja sitä käytetään projektisuunnittelussa, kustannusarvioissa, projektinhallinnassa, jne.

Fyysisessä näkymässä kuvataan järjestelmässä tarvittavat fyysiset artefaktit, millaisista osista järjestelmä koostuu, miten artefaktit ovat yhteydessä toisiinsa, ja mihin prosessointiyksikköihin mitäkin sijoitetaan.

Edellämainittujen näkökulmien lisäksi, varsinkin tuoterunkoarkkitehtuurien yhteydessä, on tarvetta vielä yhdelle näkökulmalle, variaationäkökulmalle.

Variaationäkökulma kuvaa, millaista muuntelua järjestelmä tukee. Muuntelunäkökulma kuvaa näin järjestelmän variaatiopisteet (so. kohdat järjestelmässä, joissa muuntelu tapahtuu), niiden käytön muunnelmien toteuttamiseen. Muuntelunäkökulma kuvaa siis järjestelmän toteutusalustana sovelluksille, määritellen järjestelmän laajennusrajapinnan. Muuntelunäkökulma on olennainen tuoterunkoarkkitehtuurien kohdalla, mutta se voi olla hyödyllinen esimerkiksi ylläpidon kannalta mille hyvänsä ohjelmistolle. Muuntelunäkymää tarvitaan myös mm. järjestelmän uudelleenkäytettävyyden arvioimiseksi. (Koskimies, Mikkonen, 2005)

Arkkitehtuurikuvauksen tyypit

Näkökulmasta riippumatta järjestelmän kuvaus voi painottua eri tavoilla. Kuvaukset voi luokitella kolmen ominaisuuden mukaan, eli käsitteleekö kuvaus:

- rakennetta vai käyttäytymistä
- järjestelmän staattisia vai dynaamisia piirteitä
- tietyn asian kokonaisuudessaan, vai onko se vain esimerkki

Tästä seuraa kuusi erilaista kuvaustyyppiä:

- rakennekuvaus
- käyttäytymiskuvaus
- staattinen kuvaus

- dynaaminen kuvaus
- esimerkkikuvaus
- määrittelykuvaus

Rakennekuvaus tarkastelee järjestelmän rakenteellisia suhteita (esim. viittaussuhteet, periytymissuhteet, jne.). Ajalla ei ole suurta merkitystä rakennekuvauksessa ja se voi koskea staattisia ja dynaamisia rakenteita.

Käyttäytymiskuvauksessa tarkastellaan ohjelmistoyksikköjen vuorovaikutusta. Toimintojen ajallisella järjestyksellä on tärkeä merkitys käyttäytymiskuvauksessa. Käyttäytymiskuvauksella ja rakennekuvauksella on yhteys. Jos käyttäytymiskuvauksessa komponentti1 pyytää palveluja komponentilta2, on rakennekuvauksessa oltava käyttösuhde näiden komponenttien välillä.

Staattisessa kuvauksessa tarkastellaan järjestelmän staattista esitystä (esim. lähdekoodi). Esimerkkinä oliojärjestelmässä staattinen rakennekuvaus esittää tyypillisesti luokkien väliset suhteet.

Dynaamisessa kuvauksessa tarkastellaan ohjelmiston ajoaikana ilmeneviä ominaisuuksia. Esimerkiksi jos kyseessä olisi taas oliojärjestelmä, niin dynaaminen rakennekuvaus kuvaisi olioiden välisiä suhteita.

Esimerkkikuvaus kuvaa erään mahdollisen ilmentymän kiinnostuksen kohteena olevasta ilmiöstä, eikä koko ilmiötä. Esimerkkinä dynaaminen rakenne-esimerkkikuvaus voi kuvata yhden mahdollisen oliokonfiguraation, joka saattaa esiintyä ajonaikana.

Määrittelykuvaus esittää kiinnostuksen kohteena olevasta ilmiöstä kokonaisuuden, ei pelkästään esimerkkiä. Esimerkiksi dynaaminen käyttäytymisen määrittelykuvaus voi määritellä tietyn luokan olioiden käyttäytymisen.

2.1.2 Kuvaustyypin valinta

Mitä kuvaustyyppiä sitten kannattaa käyttää milloinkin? Kuvaustyypin valinta riippuu teknisestä tarkoituksenmukaisuudesta sekä siitä, kuinka helppoa on antaa ja ymmärtää kuvaus. Esimerkkikuvaus on helpompi tehdä ja ymmärtää kuin määrittelykuvaus, mutta normaaleihin tilanteisiin keskittynyt esimerkkikuvaus ei välttämättä ota huomioon erikoistapauksia. Selkeyden vuoksi on parempi olla sotkematta kuvaukseen vastakkaisia tyyppejä. Kuvaus joka on yhtäaikaa staattinen ja dynaaminen ja esimerkinomainen ja määrittelevä, ei ole selkeä.

Arkkitehtuuriviipale

Jos ajatellaan järjestelmän arkkitehtuuri pelkästään komponenttien rakenteina ja komponenttien suhteina, niin kyseinen ajattelumalli ei yleensä pidä paikkaansa. Todellisuudessa hiemankin monimutkaisempi järjestelmä koostuu useasta loogisesta kokonaisuudesta, joista osa on järkevä esittää yksittäisinä komponentteina, mutta monet osat taas koostuvat useista komponenteista.

Ohjelmistotekniikassa on alettu ymmärtää, että järjestelmää osiin jakaessa ei voida kaikkia loogisia kokonaisuuksia tehdä erillisinä ohjelmayksikkönä. Tälläistä järjestelmän pääositusperusteen kanssa eriävää mutta jonkin kriteerin perusteella loogisesti yhteenkuuluvaa rakennetta kutsutaan (arkkitehtuuri)viipaleeksi.

Tyypillisiä viipaleita ovat suunnittelumallien ilmentymät, järjestelmän ulospäin näkyvät piirteet, ja aspektit. Jälkimmäiset ovat jonkin koko järjestelmää tai sen suurta osaa koskevan ominaisuuden tai tarpeen toteutuksia. Aspektina voi olla vaikkapa virheenkäsittely, lokitulostus, hajautus, olioiden pysyvyys, tieoturvallisuus, jne. Kaikki nämä ovat tarpeita, joiden toteutus tavallisesti jakautuu useiden komponenttien sisään. Viipaleet voivat olla myöskin keskenään osittain päällekkäisiä, esim. sama komponentti tai luokka voi

esiintyä useassa suunnittelumallissa ja samalla osallistua monen piirteen toteutukseen. (Koskimies, Mikkonen, 2005)

On tärkeää olla selvillä arkkitehtuuriviipaleiden olemassa olosta ja niiden luonteesta. Jos esimerkiksi jotakin järjestelmän piirrettä tai olioiden pysyvyyttä halutaan muuttaa, täytyy piirteitä vastaavat viipaleet löytää järjestelmästä. Kun näitä viipaleita ei edusta mitkään yksittäiset komponentit, voi olla vaikeaa löytää ne, jos viipaleiden olemassaoloa ei ole dokumentoitu etukäteen.

2.1.3 Arkkitehtuuriviipaleen kuvaus

Arkkitehturiiviipaleen kuvauksessa on erotettava kaksi asiaa: viipaleen merkitys ja viipaleen ilmeneminen ohjelmistossa. Koska viipaleet ovat osittain päällekkäisiä, viipaleeseen liittyvää merkitysinformaatiota ei voi suoraan liittää ohjelmayksiköihin, jotka esiintyvät viipaleessa. On myös huomattava, että tietty looginen viipale (esim. tietty suunnittelumalli) voi esiintyä useita kertoja samassa ohjelmistossa. Näin esimerkiksi koodin kommentoimiseen perustavat tavat viipaleiden merkkaamiseen ovat yleensä epätyydyttäviä. Viipaleen luonteen kannalta on olennaista, että viipale ja sen merkitys kuvataan yhtenä kokonaisuutena koodista erillisenä ohjelmistoartefaktina. (Koskimies, Mikkonen, 2005)

Viipale koostuu rooleista ja näiden välille määritellyistä suhteista. Kukin rooli sidotaan johonkin ohjelmaelementtiin (esim. komponentti, luokka, funktio, muuttuja, tms.), joka osallistuu viipaleeseen kyseisessä roolissa. Rooli määrittelee, millaiset elementit siihen voidaan sitoa. Toisaalta viipaleen tulee määritellä, mitä suhteita sen eri rooleihin sidotuilla ohjelmaelementeillä tulee olla keskenään. (Koskimies, Mikkonen, 2005)

Yksi suositeltava tapa dokumentoida viipale on seuraavanlainen:

- ***Nimi:*** viipale tulee nimetä yksiselitteisesti kuvaavalla tunnisteella (esim. Copy-and-paste).

- ***Laji:*** viipale tulee luokitella lajinsa perusteella (esim. piirreviipale)

- ***Tarkoitus:*** Viipaleen tarkoitus tulee kuvata lyhyesti (Esim. ”Viipale toteuttaa copy-and-paste-piirteeseen liittyvät toiminnot”).

- ***Sidosryhmät:*** minkä sidosryhmän tai –ryhmien kannalta viipale on olennainen (esim. ylläpitäjät).

- ***Roolit:*** luettelo viipaleeseen kuuluvista rooleista ja niiden merkityksestä viipaleen kannalta sekä rajoitteet, joita rooliin sidottavalla ohjelmistoelementillä on (esim. ”Rooli *TextBuffer* , sisältää kopioitavan tekstin, edellyttää luokan, joka toteuttaa rajapinnan *Text*).

- ***Roolien väliset suhteet:*** mitä suhteita viipale edellyttää ohjelmaelementeiltä, jotka on sidottu sen rooleihin (esim. ”Rooliin *X* sidotun luokan tulee periä rooliin *Y* sidottu luokka”); voidaan antaa esimerkiksi UML:n luokkakaaviona.

- ***Sidonnat:*** mitkä todelliset ohjelmaelementit on sidottu viipaleen rooleihin (esim. ”Rooliin *TextBuffer* on sidottu sovelluksen luokka *AppText*”); voidaan antaa esim. taulukkomuodossa.

Jos viipaleella voi olla useita esiintymiskohtia, riittää kuvata viipale viimeistä kohtaa lukuunottamatta vain kertaalleen, ja antaa kullekin esiintymiskohdalle erillinen sidontojen kuvaus. Viipaleen kuvausmuoto voi yleisestikin vaihdella tarpeen mukaan. Esimerkiksi suunnittelumallien tapauksessa noudatetaan yleensä hieman toisenlaista muotoa, johon palataan myöhemmin. (Koskimies, Mikkonen, 2005)

Arkkitehtuuridokumentit

Yrityksillä on omat käytäntönsä ja järjestelmät ovat erikokoisia ja niillä on eri tarkoitukset. Tästä syystä dokumentit, jotka kuvaavat ohjelmistoarkkitehtuuria,

ovat erilaisia muodoltaan ja sisällöltään. Mutta oli kyseessä suuri tai pieni järjestelmä, on arkkitehtuurikuvauksen oltava selkeänä osana dokumentointia ja sen on oltava helposti löydettävissä. Arkkitehtuuridokumentit kuuluvat ohjelmistokehitysprosessiin, ja niillä on yleensä voimakkaita riippuvuuksia muihin prosessin dokumentteihin. Esim. arkkitehtuurikuvaus viittaa toiminnalliseen määrittelyyn perustellessaan jotain ratkaisua, ja yksityiskohtaiset suunnittelu- , testi- ja muut tekniset dokumentit viittaavat arkkitehtuuridokumenttiin.

2.1.4 Arkkitehtuuridokumenttityypit

Seuraavana on luettelo ja kuvaus erilaisista arkkitehtuuridokumenttityypeistä, jotka ovat yleisesti käytössä teollisuudessa:

- ***Alustava arkkitehtuuridokumentti*** kuvaa tärkeimmät ratkaisut arkkitehtuurissa ja niiden perustelut. Myös vaihtoehtoiset ratkaisut ja niiden hyvät ja huonot puolet voidaan käsitellä alustavassa arkkitehtuuridokumentissa. Dokumenttia käytetään liiketoimintapäätösten, projektisuunnittelun, työmääräarvioinnin, alustavan arkkitehtuurin arvioinnin sekä tarkemman arkkitehtuurisuunnittelun pohjana. Dokumentissa kuvataan tyypillisesti konkreettinen arkkitehtuuri, mutta siihen voi sisältyä viittauksia referenssiarkkitehtuureihin.

- ***Järjestelmäarkkitehtuuridokumentti*** kuvaa järjestelmän arkkitehtuurin ylimmältä tasolta. Kuvaa järjestelmän sidokset ympäristöönsä, alijärjestelmien ulkoiset ominaisuudet (esim. rajapinnat) ja niiden väliset vuorovaikutukset. Vuorovaikutus voidaan kuvata vaikka siten, että kuinka käyttötapaukset toteutuvat alijärjestelmien vuorovaikutuksena. Dokumenttia käytetään arkkitehtuurisuunnitteluun, arkkitehtuurin arvioinnissa, tarkennettujen työmääräarvioiden tekemiseen, projektisuunnitteluun ja järjestelmätestauksen suunnitteluun. Dokumentissa kuvataan tyypillisesti konkreettinen arkkitehtuuri, mutta

siihen voi sisältyä kuvauksia esim. alijärjestelmien meta-arkkitehtuureista.

- ***Alijärjestelmäarkkitehtuuridokumentti*** kuvaa alijärjestelmän sisältämien komponenttien ulkoiset ominaisuudet (esim. rajapinnat) ja niiden välisen vuorovaikutuksen. Vuorovaikutus voidaan kuvata vaikkapa tarkentamalla käyttötapauksien suoritus komponenttien tasolle. Dokumenttia käytetään pohjana alijärjestelmien ja komponenttien yksityiskohtaiseen suunnitteluun ja toteutukseen, työnjakoon sekä yksikkötestauksen suunnitteluun. Dokumentti kuvaa konkreettisen arkkitehtuurin.

- ***Tuoterunkoarkkitehtuuridokumentti*** kuvaa kuinka rungon tarjoamaa arkkitehtuurin varianssia voidaan käyttää hyödyksi muiden sovellusten suunnittelussa. Hyvä tapa tähän on esittää esimerkkejä rungon soveltamisesta. Tämä dokumentti kertoo myöskin mitä sääntöjä sovellusten tulee noudattaa omissa arkkitehtuureissaan.

- ***Tuotearkkitehtuuridokumentti*** kuvaa kuinka tuoterunkoarkkitehtuuria on sovellettu ja mitä tuoterungosta riippumattomia ratkaisuja on tehty. Käytetään tuotteen yksityiskohtaiseen suunnitteluun, testaukseen ja ylläpitoon. Dokumentissa kuvataan konkreettinen arkkitehtuuri.

- ***Rajapintadokumentti*** täydentää muita dokumentteja kuvaamalla järjestelmän, alijärjestelmän tai tuoterungon tarjoaman ohjelmointirajapinnan(API). Käytetään komponenttien suunnittelun ja toteutuksen perustana.

2.1.5 Arkkitehtuuridokumentin rakenne ja sisältö

Arkkitehtuuridokumentin rakenne noudattaa yleensä yrityksen omia ohjeistuksia, jotka on luotu palvelemaan jonkin tyyppisiä järjestelmiä. Erittäin suuret järjestelmät voivat saada myös oman ohjeistuksensa. Rakenteesta

riippumatta, seuraavat asiat yleensä sisältyvät dokumenttiin (Koskimies, Mikkonen, 2005).

- ***Identifiointi:*** Ohjelmiston identifiointi, dokumentin versio sekä versiohistoria, ohjelmistoa kehittävä organisaatio, päiväys.

- ***Konteksti:*** Ohjelmistoon liittyvät liiketoimintatavoitteet ja ohjelmistoon liittyvät sidosryhmät.

- ***Vaatimukset:*** Arkkitehtuurin kannalta merkittävät vaatimukset, mahdolliset referenssiarkkitehtuurit ja tekniset rajoitteet.

- ***Ympäristö:*** Ohjelmiston tekninen toimintaympäristö ja vuorovaikutus sen kanssa.

- ***Näkymät:*** Valittujen näkökulmien mukaiset arkkitehtuurinäkymät.

- ***Arkkitehtuuriviipaleet:*** Tässä osiossa kuvataan järjestelmän arkkitehtuuriviipaleet.

- ***Analyysi:*** Kuinka hyvin arkkitehtuuri vastaa vaatimuksiin, ratkaisujen hyvät ja huonot puolet, suorituskyvyn arviointi sekä tunnettujen ongelmien ja ristiriitojen kuvaus.

- ***Perustelu:*** Perustelut arkkitehtuurin ratkaisuille.

Edellämainitut asia eivät välttämättä esiinny samassa järjestyksessä eikä niitä kaikkia edes tarvita kaikissa tapauksissa. Esim. alustavassa arkkitehtuurikuvauksessa ei ole vielä tarvetta kuvata arkkitehtuuriviipaleita.

HP:llä on mallipohja arkkitehtuurikuvaukseen perustuen 4 + 1 –malliin osoitteessa: http://www.cs.helsinki.fi/group/os3/HP_arch_template_vers13_withexamples.pdf (Luettu 23.07.2009)

Yhteenveto luvusta

- Arkkitehtuuria ei ole olemassa ilman arkkitehtuurikuvausta.

- Järjestelmän saa kuvattua tarkemmin, kun se kuvataan monesta näkökulmasta. Näkökulmia ovat skenaario-, looginen-, prosessi-, kehitys-, fyysinen- ja muuntelunäkökulma.

- Näkökulmasta riippumatta järjestelmän kuvaus voi painottua eri tavoilla. (rakenne tai käyttäytyminen, staattinen tai dynaaminen, esimerkinomainen tai määrittelevä)

- Arkkitehtuuriviipale kuvaa jonkin loogisen kokonaisuuden järjestelmästä.

- Arkkitehtuuridokumentti on tärkeä ohjelmistoartefakti, jolla on riippuvuuksia muihin artefakteihin.

3. KOMPONENTTI JA RAJAPINTA

Järjestelmät koostuvat komponenteista ja rajapinnat kuvaavat näiden liittymät. Molemmat ovat keskeisiä käsitteitä arkkitehtuuria suunnitellessa. Tässä luvussa käydään läpi mitä ovat komponentit ja rajapinnat sekä tutustutaan erilaisiin komponenttien välisiin vuorovaikutustekniikoihin.

Komponentti

Ohjelmistokomponentille on kirjallisuudessa monia erilaisia määritelmiä, jotka eivät sovi yhteen nykyisen komponenttikäsityksen kanssa. Nykyisin ohjelmistokomponentti mielletään itsenäiseksi ohjelmistoyksiköksi, joka tarjoaa palvelujaan hyvin määriteltyjen rajapintojen kautta.

Komponentin riippuvuudet:

- Olettaa yleensä tietyn infrastruktuurin olemassaolon (komponenttiteknologia, ohjelmistokehys, sovellus).
- Voi riippua toisten komponenttien tarjoamista palveluista rajapintojen kautta.
- Ei pitäisi kuitenkaan riippua suoraan toisesta komponentista.

Komponentin käyttöönotto:

- Voidaan ottaa käyttöön yhtenä yksikkönä, jos sen vaatimukset ympäristön suhteen on täytetty (saa vaatimansa palvelut rajapintojen kautta).
- Lähdekoodisena tai käännettynä binäärimuodossa.

Komponentin koko:

- Ei ole asetettu mitään yleistä rajoitusta, voi olla pieni muutaman palvelun tarjoava, tai sitten suuri yksikkö, joka tarjoaa valtavasti erikoistuneita palveluita.

- Hyvä sääntö komponentin koolle on se, että komponentin pitäisi olla yhden henkilön hallittavissa.

Rajapinta

Yksi tärkeimmistä periaatteista ohjelmistotekniikassa on pyrkiä erottamaan toisistaan se, mitä halutaan saada aikaan ja se, miten tämä tapahtuu. Komponenttien kohdalla tämä tarkoittaa sitä ,että palvelun toteutus on erotettava palvelusta abstraktiona: palvelun käyttäjän ei tulisi riippua tietystä palvelun tuottajasta, komponentista, vaan palvelusta itsestään abstraktina käsitteenä. Tämä abstrakti käsite esitellään rajapintana, jonka yksi tai useampi konkreettinen komponentti toteuttaa. (Koskimies, Mikkonen, 2005).

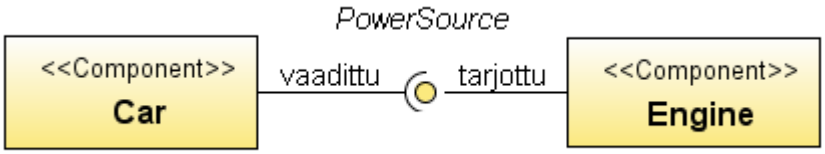
Minimimuodossaan rajapinta kertoo, miten palvelu otetaan käyttöön, ts. palvelun nimen, parametrit ja niiden tyypit sekä mahdollisen tuloksen tyypin. Tälle kokonaisuudelle käytetään nimitystä kutsumuoto (signature)(Koskimies, Mikkonen, 2005). Muita asioita, jotka tulisi rajapinnassa määritellä ovat: palvelun toiminnallinen määrittely, palvelun ei-toiminnalliset ominaisuudet, sivuvaikutukset (= vaikutukset rajapinnan tarjoavan komponentin ulkopuolelle), mahdolliset poikkeukset, jotka voivat aiheutua palvelunpyynnöstä, riippuvuudet globaaleista resursseista ja riippuvuudet palveluiden välillä. Tavallisia tapoja kuvata palveluiden merkitys ovat mm. tulo- ja jättöehdot (Muutetaan tuloehtojen kuvaamat olosuhteet jättöehtojen olosuhteiksi.) ja invariantit (Ilmaisevat ohjelmalta edelletettyä logiikkaa, invariantti on ehto, jonka täytyy olla tosi tietyissä tilanteissa ohjelman suorituksen aikana.).

Rajapinnat määräävät, miten komponentit kommunikoivat keskenään, tästä syystä rajapinnat ovat tärkeä osa ohjelmistoarkkitehtuuria. Rajapintojen järkevä suunnittelu helpottaa itse työn jakamista ja myöhemmin ohjelmiston testausta ja ylläpitoa. Rajapintojen suunnittelu alkaa tyypillisesti heti, kun keskeiset arkkitehtuuriratkaisut on tehty.

3.1.1 Tarjottu ja vaadittu rajapinta

Komponentin ja rajapinnan välillä voi olla kaksi erilaista suhdetta, rajapinta voi olla tarjottu tai vaadittu komponentille. Sama rajapinta voi olla tarjottu jollekin komponentille ja toiselle vaadittu.

Otetaanpa esimerkki tarjotuista ja vaadituista rajapinnoista.



Kuva 3.1 Tarjotut ja vaaditut rajapinnat UML:ssä (Koskimies, Mikkonen, 2005)

Car-komponentti vaatii rajapinnan PowerSource (Auto tarvitsee voimanlähteen). Komponentti Engine tarjoaa (toteuttaa) tämän rajapinnan.

Javalla kirjoitettuna edellä esitetty kaavio voisi olla esimerkiksi:

```
interface PowerSource {
    void start();
    int temperature();
   void stop();
}

class Car… {
    …
    private PowerSource eng;
    …
    public void setEng(PowerSource e) {
          eng = e;
    }
    public void run() {
        …
        eng.start();
        …
        eng.stop();
        …
    }
}
class Engine implements PowerSource… {
```

```
    …
    public void start() {…}
    public int temperature() {…}
    public void stop() {…}
}
```

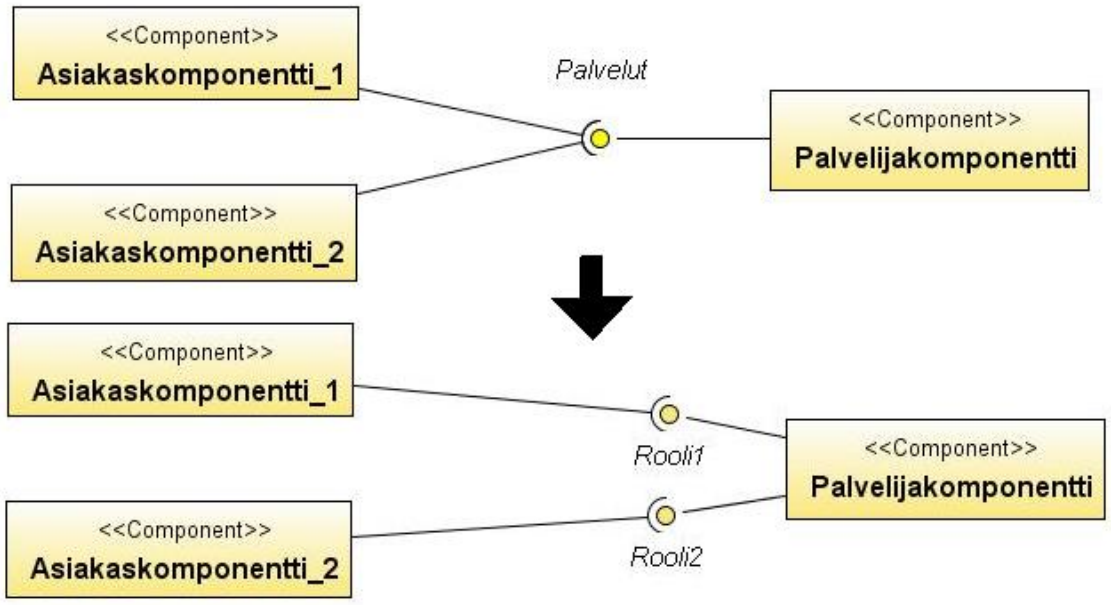
Esimerkki on lähteestä (Koskimies, Mikkonen, 2005).

3.1.2 Roolirajapinta

Tarkoituksenmukaisempi ja hienojakoisempi rajapinta saadaan aikaan tarkastelemalla tarkemmin sitä, miten komponenttien toiminnallisuus voidaan toteuttaa eri rooleissa olevien palvelun tarjoajien avulla. Komponentit tarjoavat toisilleen palveluja aina jossain roolissa, mutta tämä rooli ei välttämättä kata kaikkia palveluja, jotka kyseinen komponentti pystyy toteuttamaan. Tarkoituksena olisi antaa erillinen rajapinta kaikille identifioiduille rooleille, joissa komponentti palvelee toista komponenttia. Tälläistä rajapintaa kutsutaan roolirajapinnaksi. Komponentti on usein eri palvelijarooleissa muihin komponentteihin nähden, joten se voi toteuttaa useita roolirajapintoja, ja roolirajapinnan palveluiden tarjoajiin voi kuulua useampi komponentti.

Roolirajapintoja voi löytää esimerkiksi tarkastelemalla käyttötapauksista johdettuja sekvenssikaavioita, joissa kuvataan komponenttien vuorovaikutusta käyttötapauksien toteuttamiseksi. Jos komponentti A (tai jokin sen palvelu) käyttää komponentin B palveluista vain jotain tiettyä osaa, on tämä osa mahdollinen roolirajapinta.

Roolirajapinnat helpottavat järjestelmän ylläpitoa, koska jos muutoksia täytyy tehdä vaikkapa jonkin palvelun kutsumuotoon, se heijastuu ainoastaan komponenteissa, jotka todella käyttävät palvelua, eikä kaikissa komponenteissa, jotka käyttävät palvelun tarjoavaa komponenttia. Toisaalta taas roolirajapinnat lisäävät rajapintojen lukumäärää ja monimutkaistavat järjestelmän arkkitehtuuria.



Kuva 3.2 Roolirajapintojen käyttö (Koskimies, Mikkonen, 2005)

Komponenttien räätälöinti

Kuten aiemmin todettu, komponenttia suunnitellessa pitäisi ottaa huomioon sen uudelleenkäytettävyys. Jotta komponentti olisi uudelleenkäytettävä, sitä pitäisi pystyä käyttämään hieman erilaisissa yhteyksissä, joilla saattaa olla eri odotuksia komponentille. Jos komponentti tarjoaa palvelunsa aina samalla tavalla, sitä on vaikea hyödyntää. Kun komponenttia voi räätälöidä eri ympäristöihin, sen uudelleenkäytettävyysaste nousee. Tälläistä räätälöitävää komponenttia voisi ajatella pienenä tuoterunkona.

Seuraavana on muutama tapa, jolla saadaan komponentit tarjoamaan hieman varianssia.

3.1.3 Komponentin tilan muuttaminen

Hyvin yksinkertainen tapa räätälöidä komponentti on asettaa se tiettyyn tilaan antamalla sen tietokentille sopivat arvot. Nämä arvot voidaan antaa komponentin ilmentymää luodessa (parametrinen muodostin) tai luonnin jälkeen metodilla, joka muuttaa ilmentymän tietoja (setteri).

Esimerkkinä olkoon vaikkapa painonappikomponentti, jonka ominaisuutena on sen päällä näkyvä teksti. Painonapin voi luoda käyttämällä oletusrakentajaa ja ominaisuuden asetusoperaatiota(setLabel) tai luoda suoraan käyttämällä parametrista muodostinta (Koskimies, Mikkonen, 2005).

```
class Button extends Component ... {
    ...
    String label;
    public Button() {…}
    public Button(String arg) {label = arg;}
    public setLabel(String arg) {label = arg;}
}
```

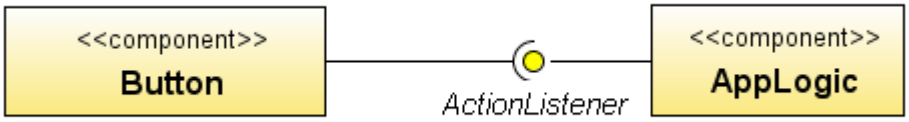
3.1.4 Vaadittujen rajapintojen toteutus

Ominaisuuksien asettaminen ei muuta komponentin koodia, vaan ainoastaan sen tietosisältöä vaikuttaen siten komponentin suorituspolkuun. Komponentin käyttäytymistä voidaan muokata voimakkaammin antamalla erilaisia toteutuksia komponentin vaatimille rajapinnoille. Tällöin palvelun suorittava koodi muuttuu, koska palvelun toteutus käyttää hyväkseen jonkin toisen komponentin koodia, ja jälkimmäinen komponentti vaihdetaan toiseksi. Koska vaaditun rajapinnan toteuttava komponentti voidaan vaihtaa milloin tahansa ajoaikana, komponentin käyttäytymistä voidaan varioida saman komponentti-ilmentymän yhteydessä. (Koskimies, Mikkonen, 2005)

Komponentin palvelut voidaan toteuttaa niin, että varsinainen palvelun suoritus delegoidaan toiselle komponentille. Tässä tapauksessa alkuperäisen komponentin tarjoama ja vaatima rajapinta on sama; alkuperäisen komponentin

tehtäväksi jää toimia välittäjänä palvelun pyytäjän ja tarjoajan kesken. Tämä ratkaisu on tarpeen kun halutaan tallettaa tietoa rajapinnan käytöstä ajonaikaiseen edustajaan rajapinnassa (esim. Edustaja-suunnittelumalli).

Esimerkkinä olkoon taas painonappikomponentti. Kun nappia painetaan, sen toiminto aktivoituu. Tätä varten komponentilla on operaatio, jolla sovelluskohtainen komponentti voidaan rekisteröidä kuuntelemaan klikkaustapahtumaa. Tämän komponentin tulee toteuttaa rajapinta ActionListener, johon sisältyy sovelluskohtaisen toiminnon aktivoiva operaatio. Kun painonappia klikataan, Button pyytää sille rekisteröityä sovelluskohtaista komponenttia suorittamaan tämän operaation. Muuttamalla tai lisäämällä kuuntelijakomponentteja painonappikomponentin käyttäytymistä klikkaustilanteessa voidaan muunnella halutulla tavalla. Kuvassa 3.3 on esitetty Button-komponentin räätälöinti antamalla vaaditun rajapinnan ActionListener toteuttava sovelluskohtainen komponentti AppLogic. (Koskimies, Mikkonen, 2005)

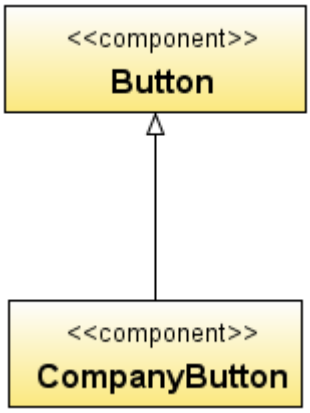


Kuva 3.3 Komponentin räätälöinti vaaditun rajapinnan toteutuksen avulla (Koskimies, Mikkonen, 2005)

3.1.5 Periytymisen avulla tehty räätälöinti

Vaadittuihin rajapintoihin perustuva räätälöinti muuttaa komponentin kutsumaa koodia, mutta ei komponentin omaa koodia. Joskus haluttua variointia ei pystytä luontevasti ilmaisemaan vaaditun rajapinnan erilaisten toteutusten avulla, vaan komponentin omaa koodia halutaan muuttaa. Jos komponentti on toteutettu luokan avulla, voidaan käyttää periyttämistä. Tällöin komponentin luokan perii toinen luokka, joka uudelleenmäärittelee jotkin sen operaatiot. Näin syntyy kokonaan uusi komponentti, joka on periytymissuhteessa alkuperäisen

komponentin kanssa. Uudella komponentilla on samat tarjotut rajapinnat kuin alkuperäisellä, mutta joillakin sen tarjoamilla palveluilla on täysin uusi toteutus. (Koskimies, Mikkonen, 2005)



Kuva 3.4 Periyttämällä tehty räätälöinti

Esimerkkinä katsotaan taas painonappikomponenttia. Otetaanpa vaikka yritys, joka haluaa, että kaikkien sen tuotteissa olevien painonappien tekstit esitetään isoilla kirjaimilla. Tätä varten yritys tekee oman painonappikomponentin, joka periytyy standardista komponentista ja muuttaa rakentajan ja setLabel-operaation toteutuksen.

```
class CompanyButton extends Button {
    public CompanyButton(String arg){
        super(arg);
        label = arg.toUpperCase();
    }

    public void setLabel(String arg) {
        label = arg.toUpperCase();
    }
}
```

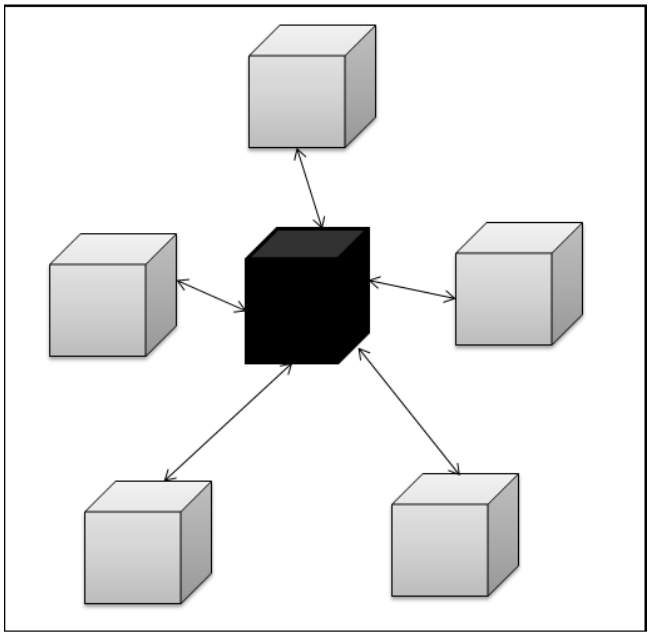
Tällä tavalla saadaan aikaiseksi se, että kaikissa paikoissa missä napin teksti asetetaan, kutsutaan samalla toUpperCase-funktiota, joka muuttaa tekstin isoiksi kirjaimiksi. Kun yritys ohjeistaa käyttämään vain tätä erikoistettua painonappia, niin kaikissa yrityksen sovelluksissa noudatetaan haluttua käytäntöä.

Komponenttien vuorovaikutustekniikoita

Ohjelmistoarkkitehtuurisuunnittelu on suureksi osaksi komponenttien välisten suhteiden määrittelyä. Tämä johtuu siitä, että yleisempiä ohjelmistolle asetettavia vaatimuksia ovat uudelleenkäytettävyys, ylläpidettävyys ja työn hajautettavuus ja näihin kaikkiin vaikuttavat ohjelmiston komponenttien suhteet ja riippuvuudet. On tärkeää vähentää ja selkeyttää näitä riippuvuuksia. Näin saadaan täytettyä ohjelmiston laadullisia vaatimuksia. Käytännössä tämä toteutetaan esimerkiksi rajapinnoilla ja välittäjäluokilla.

3.1.6 Välittäjän käyttö

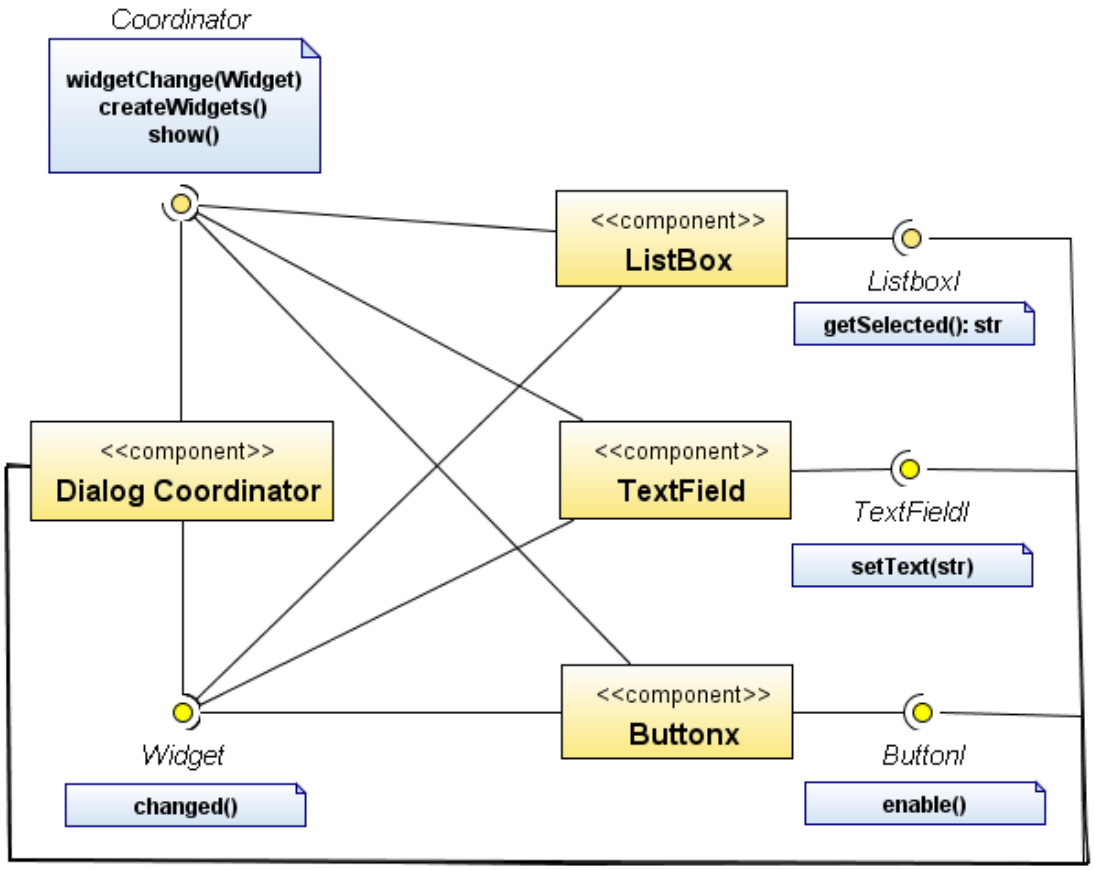
Kun kuvataan ohjelman toiminta komponenteilla, jotka tarjoavat tai pyytävät toisiltaan palveluita, tuo se ongelmia arkkitehtuuritasolle. Tämä johtuu siitä, että se sitoo komponentteja johonkin tiettyyn ympäristöön ja tekee komponenteista riippuvaisia, joka taas vaikeuttaa niiden uudelleenkäyttöä ja ylläpitoa. Myöskin tälläisen ohjelmiston toiminnallisuuden muuttaminen on vaikeaa, koska se yleensä tarkoittaisi usean komponentin muuttelua.



Kuva 3.5 Välittäjän käyttö komponenttien välisessä vuorovaikutuksessa (Koskimies, Mikkonen, 2005)

Tälläisen ongelman voi ratkaista käyttämällä välittäjää (Kuva 3.5). Välittäjä toteuttaa komponenttien vuorovaikutuksen ja purkaa komponenttien suorat riippuvuudet toisistaan ja tällöin kaikki osallistuvat komponentit kommunikoivat välittäjän kanssa eivätkä suoraan toistensa kanssa. Tästä seuraa se, että nyt jos haluttaisi muuttaa ohjelmiston toiminnallisuutta, tarvitsisi muuttaa vain välittäjää, ja osallistuvat komponentit ovat käytettävissä muuallakin, kunhan ne pystyvät kommunikoimaan toisen välittäjän kanssa.

Tyypillinen esimerkki välittäjän käytöstä on dialogi-ikkunan toteutus (Kuva 3.6). Dialogi koostuu erilaisista käyttöliittymäkomponenteista (”widget”), jotka toteutetaan komponenteilla. Dialogin käytön aikana nämä komponentit ovat vuorovaikutuksessa keskenään. Oletetaan esimerkiksi, että dialogissa näytetään alkiolista, tekstikenttä ja painonappi: kun käyttäjä valitsee alkion listasta, valittu alkio näytetään tekstikentässä ja painonappi aktivoidaan. Periaatteessa listakomponentti voisi kommunikoida suoraan tekstikenttä- ja painonappikomponenttien kanssa, mutta silloin kaikki komponentit riippuisivat juuri tästä dialogista ja sen toiminnallisuudesta. Tässä esimerkissä komponenttien keskinäiset riippuvuudet voidaan purkaa antamalla erillinen välittäjä, joka toteuttaa komponenttien välisen vuorovaikutuksen; kukin komponentti kommunikoi vain välittäjän kanssa. Välittäjän voi ajatella tässä olevan komponentti, joka edustaa dialogia. Samoja käyttöliitymäkomponentteja voidaan nyt käyttää useissa dialogeissa. (Koskimies, Mikkonen, 2005)



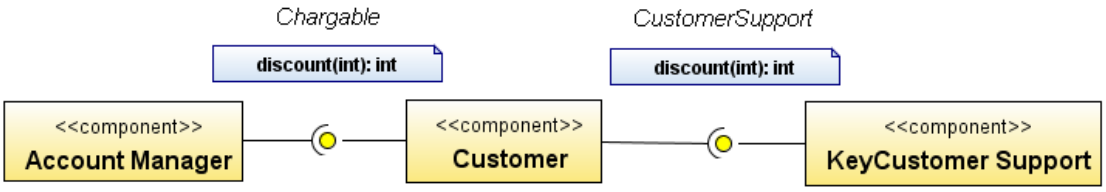
Kuva 3.6 Välittäjän käyttö dialogi-ikkunan komponenttien välisessä vuorovaikutuksessa (Koskimies, Mikkonen, 2005)

3.1.7 Kutsun siirto

Yksi perustekniikka erottaa palvelun tarvitseva komponentti ja palvelun tarjoava komponentti toisistaan on kutsun siirto (forwarding). Tämä tarkoittaa sitä, että komponentti joka saa palvelupyynnön ei itse suorita palvelua vaan välittää pyynnön jollekin toiselle komponentille, joka sitten toteuttaa palvelun. Palvelun alkuperäinen pyytäjä ei ole tietoinen välissä olevasta komponentista. Siirron yhteydessä palvelun välittämiseen liittyy siis aina jotain toiminnallisuutta. Siirtämistä käytetään useaan tarkoitukseen ja sitä sovelletaan suunnittelumalleissa.

Siirtämistä voi esimerkiksi käyttää palvelun toteutuksen dynaamiseen vaihtamiseen. Kuvan 3.7 esimerkissä Customer-komponentti tarjoaa rajapinnan Chargable, jossa on määritelty palvelu discount, joka palauttaa asiakasryhmä

kohtaisesti tietystä laskusummasta alennuksen. Asiakas voi olla tavallinen tai kanta-asiakas, ja vaihtaa tilaansa olemassaolonsa aikana. Tämä voidaan toteuttaa niin, että customer ei itse toteuta discount-palvelua vaan siirtää palvelupyynnön toiselle komponentille, joka sitten laskee alennuksen. Jälkimmäinen komponentti on sitten eri tavallisille ja kanta-asiakkaille, ja voi toteuttaa myös muita asiakasryhmästä riippuvia palveluita. Kun asiakas mahdollisesti vaihtaa tilaansa, tämä komponentti vaihdetaan toiseksi asiakasta edustavassa Customer-komponentissa, eikä komponentin käyttäjän tarvitse olla vaihdosta tietoinen. (Koskimies, Mikkonen, 2005)



Kuva 3.7 Siirtämisen käyttö palvelun muunteluun (Koskimies, Mikkonen, 2005)

3.1.8 Edustajakomponentti

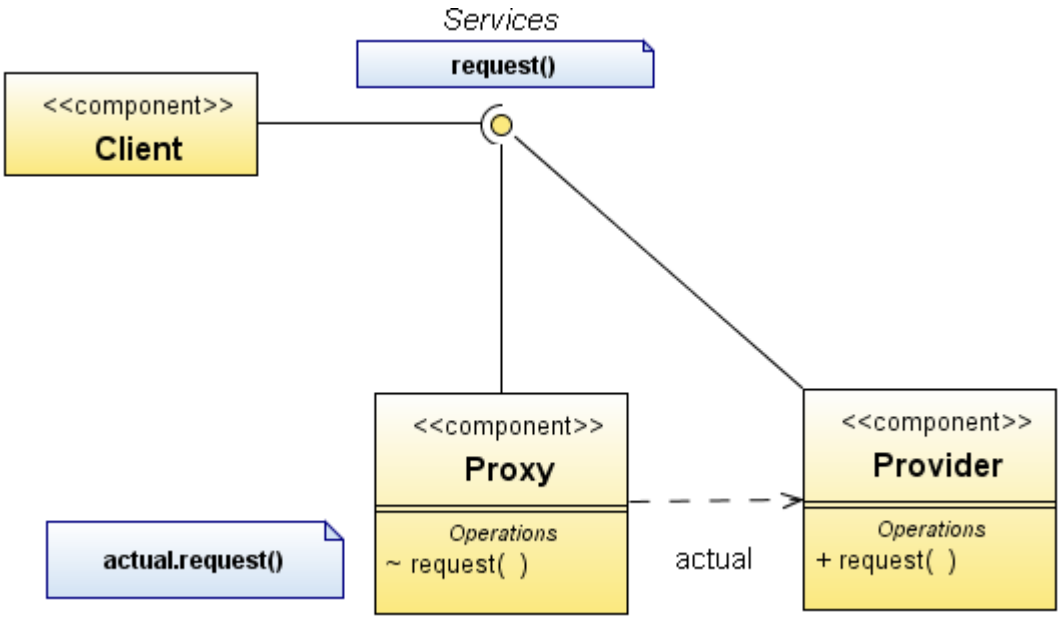
Hyvin suoraviivaisesti siirtämistä sovelletaan edustajan(proxy) yhteydessä. Erinäisistä syistä suora pääsy jonkin komponentin palveluihin halutaan estää, ja tarjota nämä palvelut epäsuorasti jonkin muun komponentin, edustajan, kautta.

Syitä tälläiseen ovat esimerkiksi:

- Komponentti on eri koneessa kuin sen palveluiden pyytäjä.
- Komponenti luonti on raskas toimenpide, jota halutaan lykätä niin pitkälle kuin mahdollista.
- Komponentin käyttöä halutaan valvoa.

Komponentti voidaan irrottaa käyttäjistään edustajan avulla, jolloin käyttäjät kommunikoivat komponentin kanssa edustajan kautta. Käyttäjä ei ole tietoinen välissä olevasta edustajasta vaan näkee edustajan rajapinnan kautta, joka on yhteinen edustajalle ja varsinaiselle komponentille, joka palvelut toteuttaa.

Tällöin palveluiden käyttäjä ei ole riippuvainen varsinaisesta palveluita tarjoavasta komponentista, mutta riippuvuus rajapintaan pysyy. Asiaa on kuvattu kuvassa 3.8. Provider tarjoaa varsinaisen palvelun mutta sitä käytetään proxyn kautta.



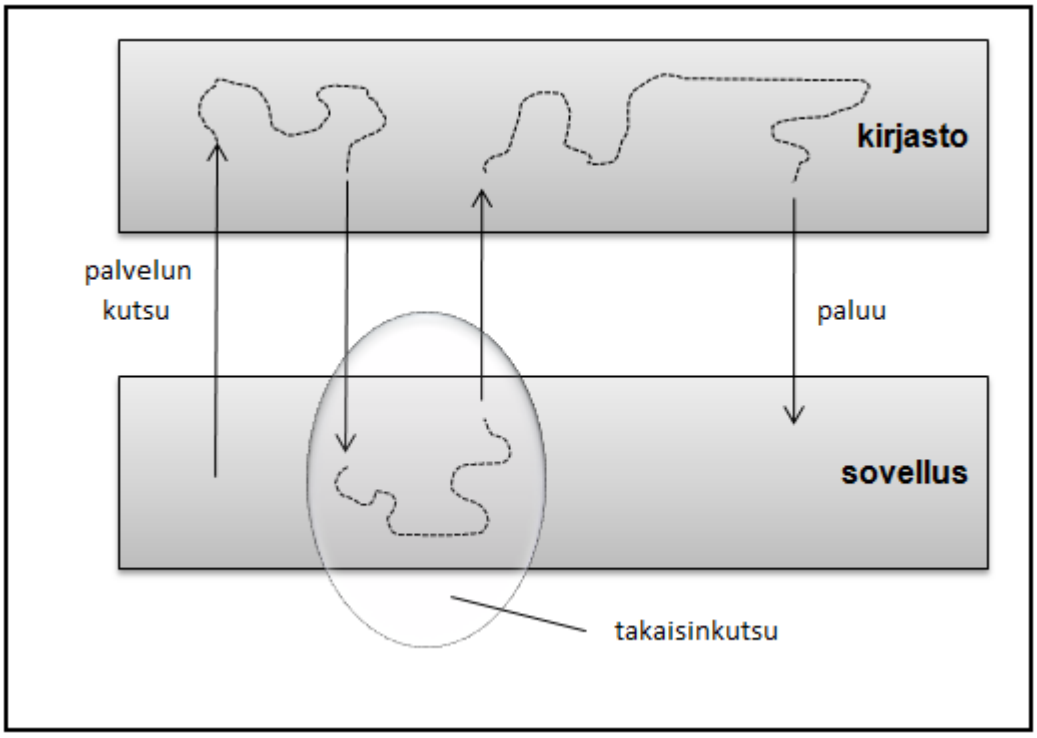
Kuva 3.8 Edustajan käyttö komponenttien välillä (Koskimies, Mikkonen, 2005)

Edustajaa käytetään tyypillisesti hajautetuissa komponenttijärjestelmissä, joissa asiakaskoneella toimivat sovellukset pääsevät palvelimella olevaan komponenttiin asiakaskoneella olevan edustajan kautta. Edustaja muokkaa palvelupyynnön verkossa välitettävään muotoon ja lähettää sen palvelimella olevalle edustajalle, joka puolestaan purkaa viestin ja pyytää palvelimella olevaa komponenttia suorittamaan pyydetyn palvelun. Edustajat myöskin palauttavat palvelun tulokset takaisin alkuperäiselle palvelunpyytäjälle. (Koskimies, Mikkonen, 2005)

3.1.9 Takaisinkutsu

Takaisinkutsu on mekanismi, jonka avulla palvelun kutsuja voi saada kontrollin palvelun aikana. Tällä tavalla palvelua kutsunut ohjelmayksikkö voi suorittaa omia toimenpiteitään tietyssä vaiheessa palvelua. Takaisinkutsua käytetään kun halutaan yleiskäyttöinen ohjelmayksikkö, joka antaa kontrollin takaisin

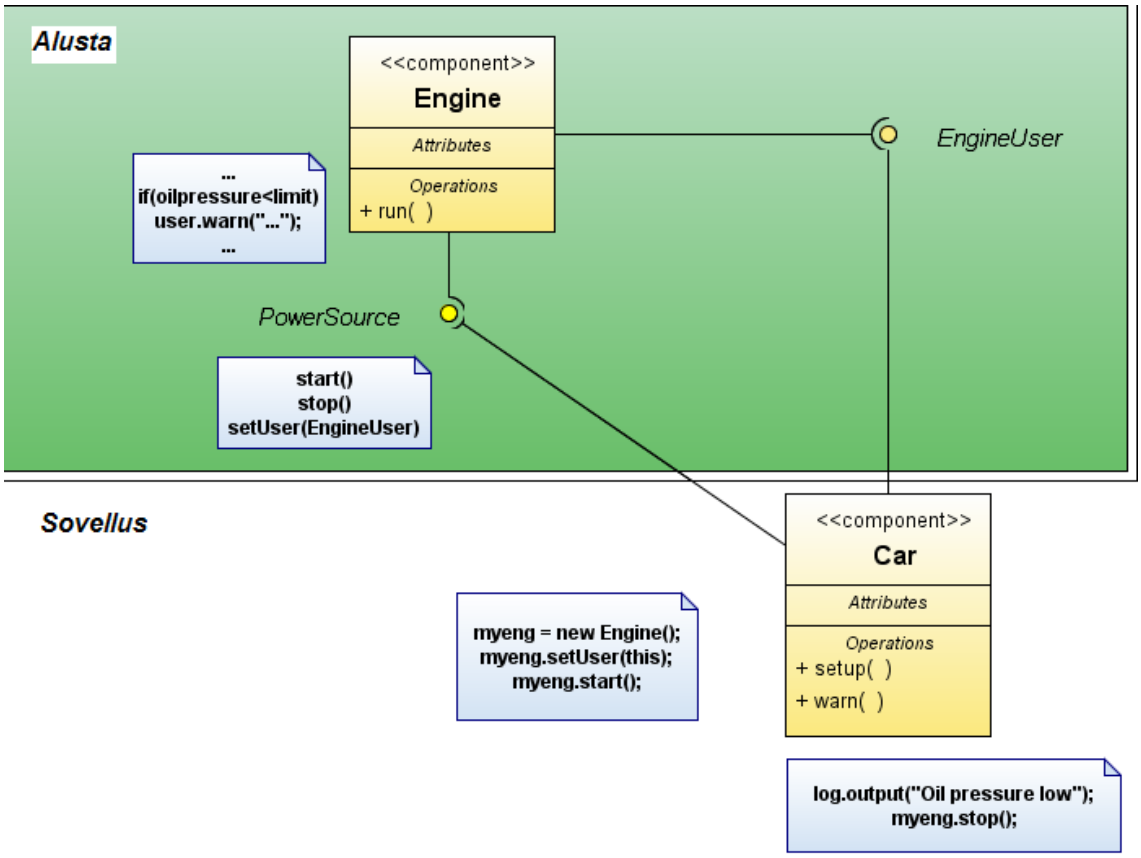
palvelun pyytäjälle suorituksen aikana. Takaisinkutsua on havainnollistettu kuvassa 3.9.



Kuva 3.9 Takaisinkutsu (Koskimies, Mikkonen, 2005)

Esimerkkinä olkoon Engine-komponentti (Kuva 3.10), josta halutaan yleistää yleiskäyttöinen kirjastokomponentti. Lisäksi halutaan että öljynpaineen laskiessa moottorin käyttäjällä on mahdollisuus omiin toimenpiteisiin. Engine-komponentin pitää siis tässä tilanteessa suorittaa takaisinkutsu käyttäjälleen.

Tässä esimerkissä kirjasto tarjoaa Engine-komponentin lisäksi rajapinnan EngineUser, joka kuvaa vaatimukset joita Engine odottaa käyttäjältään. Tässä tapauksessa vaatimuksena on, että käyttäjän on toteutettava warn-palvelu, jota kutsutaan öljynpaineen laskiessa. Komponentti Car toteuttaa tämän rajapinnan ja luo Engine-komponentista ilmentymän käyttöönsä asettaen viitteen itseesä (user). Engine-komponentti käyttää tätä viitettä kutsuessaan warn-operaatiota, jonka Car toteuttaa.



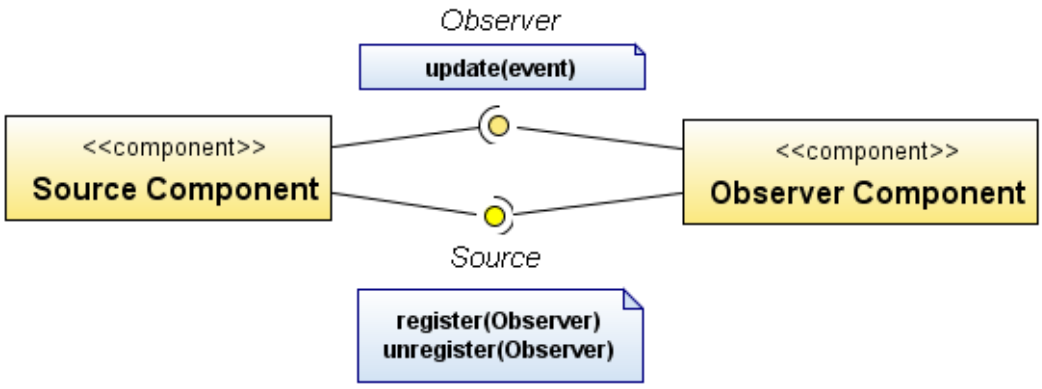
Kuva 3.10 Takaisinkutsun käyttö kirjastokomponentin yhteydessä. (Koskimies, Mikkonen, 2005)

3.1.10 Tapahtumiin perustuva vuorovaikutus

Palvelun pyytäjän ja tarjoajan välistä suhdetta voi heikentää käyttämällä tapahtumaa (event). Tapahtuma on tilanne, joka tapahtuu ohjelman suorituksen aikana ja jolla on tunnettu esitystapa sekä vaatii reagointia joiltakin ohjelman osilta. Tällä tavalla siis palvelun pyyntö tilanne on tapahtuma ja palvelun tarjoaminen on tapahtumaan reagointia.

Yleinen tapa toteuttaa tapahtumamekanismi on soveltaa takaisinkutsun ideaa. Tarkkailijat rekisteröityvät lähteelle (komponentti, joka pyytää palvelua), joka kutsuu niiden tapahtumankäsittelyoperaatiota tapahtuman sattuessa. Tämä operaatio kuuluu takaisinkutsurajapintaan, jonka toteuttavat tarkkailijat. Tapahtuman tiedot annetaan operaation parametreinä (esim. tapahtumaoliona). Kuvassa 3.11 on kuvattu tälläinen

tapahtumakäsittelyarkkitehtuuri. Source-rajapinta tarjoaa palvelut tarkkailijoiden rekisteröinnille ja sen peruuttamiseen. Observer-rajapinta tarjoaa palvelun update, joka on tapahtumien käsittelyoperaatio, jonka tarkkailijat toteuttavat. Kuvan 3.11 ratkaisu on esitetty myös suunnittelumallina Tarkkailija (Observer).

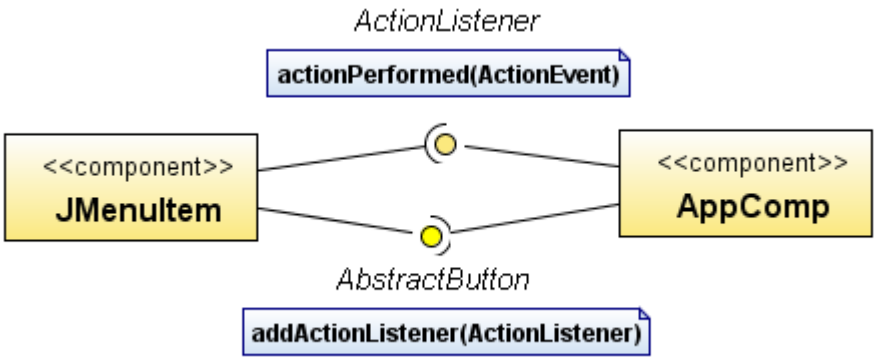


Kuva 3.11 Tarkkailija-suunnittelumallin mukainen tapahtumankäsittely (Koskimies, Mikkonen, 2005)

Tarkkailija-malliin perustuvaa tapahtumien käsittelyä käytetään yleensä vuorovaikutteisissa sovelluksissa välittämään kontrollia käyttöliittymän ja sovelluslogiikan välillä. Tapahtumat voivat kulkea molempaan suuntaan käyttöliittymän elementit voivat ilmoittaa käyttöliittymän tapahtumista sovelluslogiikalle haluttujen operaatioiden aktivoimiseksi ja sovelluslogiikka voi ilmoittaa tapahtumilla sovelluksen tilan muutoksista käyttöliittymäkomponenteille, joiden tulisi päivittää näyttö muutosten mukaan.

Esimerkkinä olkoon JavaBeans-arkkitehtuuri ja Swing-kirjasto, jotka soveltavat Tarkkailija-mallia yleisesti eri käyttöliittymälähtöisille tapahtumille. JavaBeans-terminologiassa tarkkailijaa kutsutaan kuuntelijaksi(Listener). Esimerkiksi valikossa oleva komento toteutettu komponentilla JmenuItem, joka toteuttaa rajapinnan AbstractButton. Tähän kuuluu palvelu addActionListener, jonka avulla ActionListener- rajapinnan toteuttava kuuntelija rekisteröidään valikkokomponentille. ActionListener- rajapinta sisältää tapahtuman käsittelyoperaation actionPerformed, jonka parametrina annetaan tapahtumaolio, johon on talletettu tarkemmat tiedot tapahtumasta. Kuvassa

3.12 on esitetty yksinkertaistettu komponenttikaavio, jossa AppComp edustaa komennon sovelluskohtaisen toiminnon toteuttavaa komponenttia. (Koskimies, Mikkonen, 2005)



Kuva 3.12 Esimerkki JavaBeans-tapahtumakäsittelystä (Koskimies, Mikkonen, 2005)

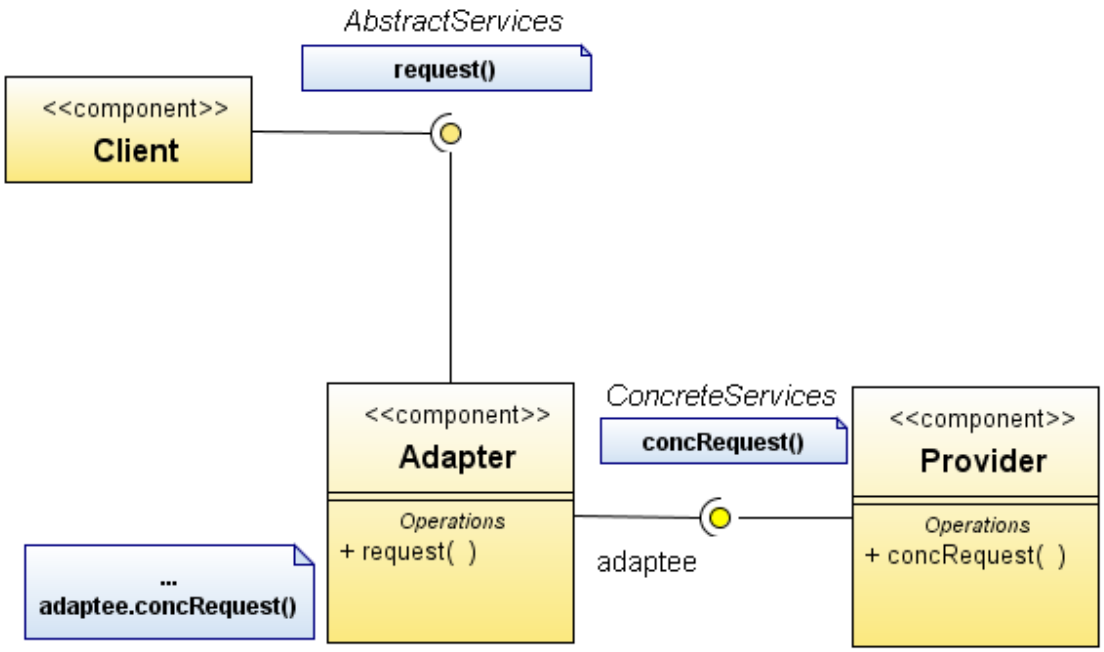
3.1.11 Sovitin

Joskus tulee tilanne kun komponentit eivät voi tuntea edes toistensa toteuttamia rajapinoja. Tälläinen tilanne voi syntyä kun halutaan tehdä uusi järjestelmä jo olemassa olevista komponenteista, jotka on tehty toisistaan riippumatta. Jos näitä komponentteja ei ole tehty jollain yleisellä rajapintastandardilla, nämä komponentit luultavasti toteuttavat toisilleen tuntemattomia rajapintoja.

Perusratkaisu rajapintariippuvuuksien poistoon on käyttää sovitinta(adapter). Sovittimen avulla komponentti voi kutsua toisen komponentin palveluita tuntematta sen rajapintaa. Sovitin muuttaa kutsujan palvelupyynnön vastaanottavan rajapinnan mukaiseksi. Täten saadaan mahdolliseksi vuorovaikutus näiden kahden toisistaan riippumattoman komponentin välille.

Sovittimen idea on kuvattu kuvassa 3.13. Client edustaa komponenttia, joka halutaan tehdä riippumattomaksi yksittäisen palveluntarjoajan antamasta rajapinnasta. Adapter toteuttaa AbstractServices-rajapinnan, jota vasten Client-komponentti on suunniteltu. Palvelun request toteutuksessa tehdään muutetun

rajapinnan vaatimia muutoksia, ja kutsutaan sen jälkeen palvelun varsinaista toteuttajaa(Provider) tämän rajapinnan ConcreteServicen kautta.



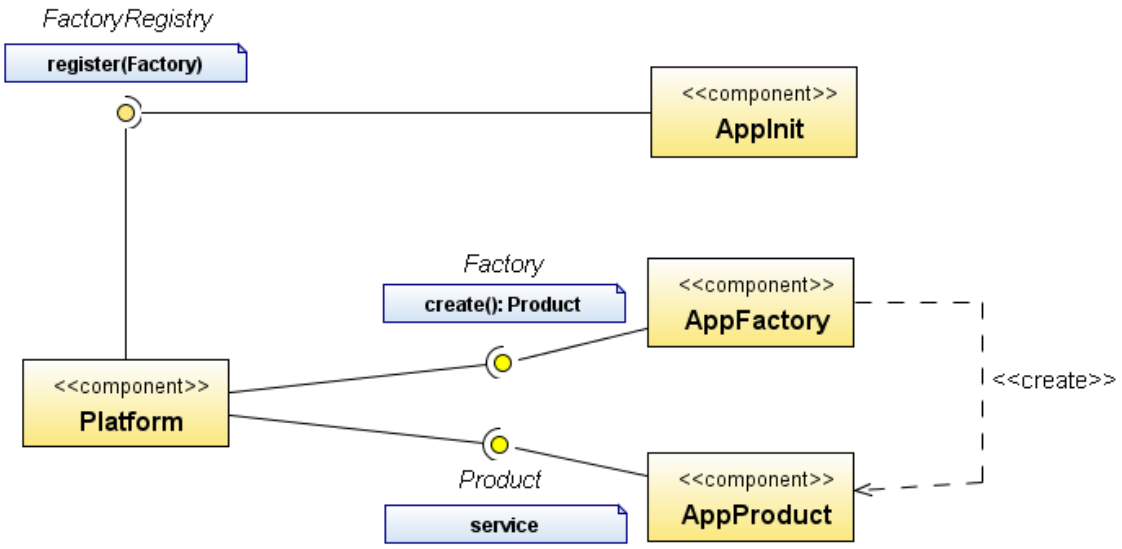
Kuva 3.13 Sovittimen käyttö (Koskimies, Mikkonen, 2005)

3.1.12 Tehdas

Suoraviivaisessa toteutuksessa komponentin luodessa ilmentymän toisesta komponentista, luovan luokan on tunnettava luotavan luokka, mitä ei usein toivota. Tällä tekniikalla tätä luontiriippuvuutta voi heikentää.

Usein tulee tilanne, jossa yleiskäyttöinen ohjelmisto, esimerkiksi tuoterunko, joutuu luomaan joukon sovelluskohtaisia komponentteja. Tämä johtuu siitä, että usein on tarpeen kuvata jokin sovelluskohtainen käsitekokonaisuus komponenttina, ja näiden komponenttien ilmentymien lukumäärä riippuu sovelluksen käytöstä. Jos sovellusten rakentamista tukee tuoterunko, tämän on pystyttävä luomaan mielivaltainen määrä sovelluskohtaisia komponentin ilmentymiä.

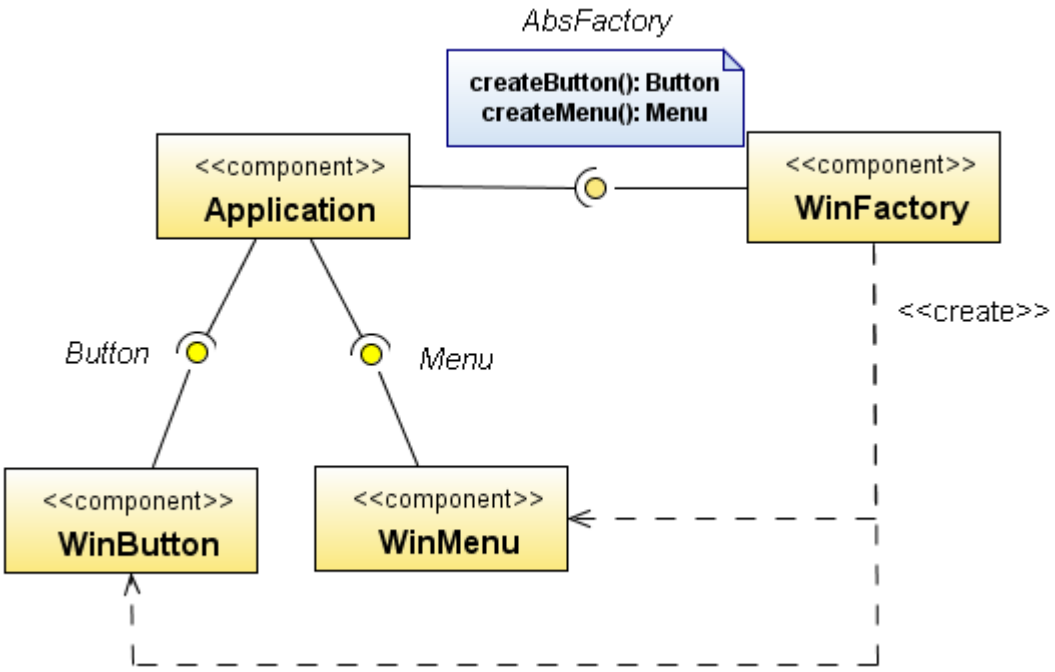
Tälläisen ongelman voi ratkaista ottamalla käyttöön rajapinnan luontia varten, sekä sitomalla luontioperaation kutsumuoto johonkin luotavien komponenttien yhteiseen rajapintaan, eikä mihinkään yksittäiseen komponenttityyppiin. Tuoterunko tuntee näin vain rajapinnat, eikä itse komponentteja. Sovellus luo luontirajapinnan toteuttavan komponentin, nk. tehdaskomponentin, joka antaa luontioperaation sellaisessa muodossa, että se luo tietyn sovelluskohtaisen komponentin ilmentymän. Kun sovellus rekisteröi tehdaskomponentin tuoterungolle, tämä voi kutsua tehdaskomponentin luontipalvelua, joka näin ollen palauttaa sovelluskohtaisen komponentin ilmentymän ilman, että tuoterunko tuntee komponentin tyyppiä. Tämä ratkaisu on esitetty kuvassa 3.15 komponenttikaaviona. (Koskimies, Mikkonen, 2005)



Kuva 3.14 Luontiriippuvuuden heikentäminen tehtaan avulla (Koskimies, Mikkonen, 2005)

Abstrakti tehdas –suunnittelumallissa tehdas tarjoaa luontioperaatiot tietylle joukolle erilaisia komponentteja, joita käytetään yhdessä. Tehdas luo komponentin aina tietyn variantin mukaisina. Esimerkkinä tälläisestä komponenttijoukosta voisivat olla vaikkapa käyttöliittymäelementit, jotka tehdas luo aina tietyn käyttöliittymäkirjaston mukaisina. Kuvassa 3.16 on esitetty tämän tehdasratkaisun sovellus tilanteessa, jossa halutaan tehdä tietystä käyttöliittymäkirjastosta riippumaton sovellus. Tässä kirjasto voidaan vaihta toiseksi antamalla tehdasrajapinnalle uusi toteutus, joka luo jonkin toisen

graafisen kirjaston mukaisia käyttöliittymäelementtejä. (Koskimies, Mikkonen, 2005)



Kuva 3.15 Abstraktin tehtaan soveltaminen käyttöliittymäarkkitehtuurissa (Koskimies, Mikkonen, 2005)

Yhteenveto luvusta

- Komponentti on itsenäinen ohjelmayksikkö, joka tarjoaa palveluita hyvin määriteltyjän rajapintojen kautta.

- Rajapinnan tulee antaa palvelusta kaikki olennainen tieto käyttäjälle.

- Komponentilla on tarjotut ja vaaditut rajapinnat.

- Rajapintojen suunnittelun pitäisi pohjautua rooleihin.

- Komponenttia voidaan räätälöidä tilaa muuttamalla, eri toteutuksilla rajapinnoissa ja periyttämällä.

- Ohjelmistoarkkitehtuurisuunnittelu perustuu komponenttien välisiin suhteisiin. Mitä riippumattomampia komponentit ovat, sitä helpompi järjestelmää on rakentaa ja ylläpito ja uudelleenkäyttö on helpompaa.

- Välittäjä poistaa keskenään kommunikoivien komponenttien riippuvuuden toisistaan.

- Kutsunsiirto tarkoittaa sitä, kun komponentti joka saa palvelupyynnön ei itse suorita palvelua vaan välittää pyynnön jollekkin toiselle komponentille.

- Komponentti voidaan irrottaa käyttäjistään edustajan avulla, jolloin käyttäjät kommunikoivat komponentin kanssa edustajan kautta.

- Takaisinkutsu avulla palvelun kutsuja voi saada kontrollin palvelun aikana. Tällä tavalla palvelua kutsunut ohjelmayksikkö voi suorittaa omia toimenpiteitään tietyssä vaiheessa palvelua.

- Palvelun pyytäjän ja tarjoajan välistä suhdetta voi heikentää käyttämällä tapahtumia.

- Sovittimen avulla komponentti voi kutsua toisen komponentin palveluita tuntematta sen rajapintaa.

- Tehtaalla voi heikentää luontiriippuvuutta.

4. SUUNNITTELUMALLIT

Suunnittelumallit ovat hyväksihavaittuja ratkaisuja yleisiin ohjelmistosuunnittelussa esiin tuleviin ongelmiin. Tässä luvussa käydään yleisellä tasolla läpi suunnittelumallit ja niiden dokumentointi.

Mikä on suunnittelumalli?

Mestareiden töiden kopiointi on ikivanha keksintö, ja juuri siitä on kyse suunnittelumalleissa. Edellämainittu toimii hyvin varsinkin ohjelmistotekniikan alalla, koska alalla on paljon kokemattomia ohjelmoijia ja toisaalta taas paljon ammattitaitoisia ohjelmistoarkkitehteja, jotka voivat jakaa tietämystään suunnittelumalleilla, joilla ratkaistaan yleisiä ohjelmistonsuunnitteluun liittyviä ongelmia. Ohjelmistotekniikassa suunnittelumalli tarkoittaa siis yleistä tapaa jonkin usein esiintyvän ongelman ratkaisemiseksi.

Osoitteessa http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html (Luettu 4.9.2009) on paljon lisätietoa suunnittelumalleista. Sivusto on ehkä vanha, mutta sisältö on vieläkin ajankohtaista.

Kirjassa ”Olio-ohjelmointi Suunnittelumallit, Gamma E. et al, 2001” ja alkuperäisessä ”Design Patterns – Elements of Reusable Object-Oriented Software, Gamma E. et al, 1995” on esitelty 23 yleistä suunnittelumallia.

Suunnittelumallin sisältö ja kuvaus

Suunnittelumallit esitetään dokumentteina, joilla on yhtenäinen rakenne. Dokumentin pitäisi antaa kaikki tarvittava tieto mallin soveltajalle. Esitysmuotoja on erilaisia, mutta seuraavat asiat tulisi ainakin kirjata suunnittelumallin kuvaukseen:

- **Mallin nimi:** Annetaan suunnittelumallille yksiselitteinen nimi.

- **Ongelmayhteys:** Kuvataan millaisissa tilanteissa mallia kannattaa käyttää. Voi sisältää myös listan ehdoista, joiden täytyy täyttyä, jotta mallin käyttäminen on järkevää.

- **Ratkaisu:** Kuvataan elementit joista ratkaisu koostuu. Elementeistä kuvataan myös niiden vastuut ja yhteydet toisiinsa. Ratkaisussa ei kuvata konkreettista toteutusta, vaan siinä kuvataan joku abstrakti kuvaus ongelmasta ja se, kuinka ja millaisilla elementeillä ja niiden yhteistyöllä ongelma ratkaistaan.

- **Seuraukset:** Kuvataan mallin soveltamisen tuloksia ja sen hyötyjä sekä haittoja.

GoF(Gang of Four)(Gamma et al. 2001) tarjoaa seuraavanlaisen kuvausformaatin:

- **Mallin nimi ja luokittelu:** Mallin nimi kertoo mallin olemuksen ytimekkäästi. Hyvä nimi on tärkeä, sillä siitä tulee osa suunnittelusanastoa.

- **Tarkoitus:** Kuvataan lyhyesti mitä suunnittelumalli tekee ja minkalaisten ongelmien ratkaisuun se soveltuu.

- **Alias:** Mallin muut nimet, jos niitä on.

- **Perustelut:** Lyhyt esimerkki, jossa kuvataan suunnitteluongelma ja se, millä rakenteella ongelma ratkaistaan. Esimerkki helpottaa myöhemmän abstraktin kuvauksen ymmärtämistä.

- **Soveltavuus:** Mihin tilanteisiin suunnittelumallia voi soveltaa.

- **Rakenne:** Mallin graafinen esitys, esim. UML: n luokka- ja sekvenssikaaviot.

- **Osallistujat:** Malliin liittyvät osat ja niiden vastuut.

- **Yhteistyösuhteet:** Miten osallistujat toimivat yhteistyössä toteuttaakseen vastuunsa.

- **Seuraukset:** Mitä etuja mallin käytöllä saavutetaan mitä menetetään.

- **Toteutus:** Mitkä asiat on tiedettävä mallia toteutettaessa ja onko eroja eri ohjelmointikielillä tehdessä.

- **Mallikoodia:** Koodiesimerkkejä.

- **Tunnettuja käyttäkohteita:** Esimerkkejä missä mallia on käytetty oikeissa järjestelmissä.

- **Läheiset mallit:** Mitkä suunnittelumallit liittyvät läheisesti tähän malliin ja mitä tärkeitä eroja näiden välillä on sekä minkä muiden mallien kanssa tätä mallia voisi käyttää.

Suunnittelumallien luokittelu

Suunnittelumallien karkeus- ja abstraktiotaso vaihtelevat ja koska niitä on paljon on ne jäsenneltävä jotenkin. Taulukossa 4.1 on luokittelu, joka ryhmittelee läheisesti yhteen kuuluvat mallit perheiksi. Suunnittelumallit on luokiteltu kahden kriteerin mukaan.

Ensimmäisenä kriteenä on tarkoitus, joka kuvaa mallin toimintaa. Malli kohdistuu joko luontiin, rakenteeseen tai käyttäytymiseen. Luontimallit käsittelevät olioiden luontiprosessia, rakennemallit koskevat luokkien ja olioiden koosteita ja käyttätymismallit käsittelevät luokkien ja olioiden vuorovaikutusta ja

vastuita. Toisena kriiterinä on kohde, joka kertoo liittyykö malli ensisijaisesti luokkiin vai olioihin.

Taulukko 4.1 Suunnittelumallien luokittelu (Gamma et al. 2001)

		Tarkoitus		
Kohde		**Luonti**	**Rakenne**	**Käyttäytyminen**
	Luokka	Tehdasmetodi	Sovitin (luokka)	Tulkki Operaatiorunko
	Olio	Abstrakti tehdas Rakentaja Prototyyppi Ainokainen	Sovitin (olio) Silta Rekursiokooste Kuorruttaja Julkisivu Hiutale Edustaja	Vastuuketju Komento Iteraattori Välittäjä Muisto Tarkkailija Tila Strategia Vierailija

Suunnittelumallien hyödyt ja ongelmat

Kunkin yksittäisen suunnittelumallin kuvauksessa tulisi ilmetä kyseisen mallin edut ja haitat. Seuraavana listaus yleisellä tasolla mitä etuja ja haittoja suunnittelumallien käytöstä on.

\+ Kokemuksen siirto vanhoilta nuoremmille ohjelmoijille.

\+ Tarjoavat korkeamman tason rakenneabstraktioita, joita yhdistämällä ohjelmistoja voidaan rakentaa.

\+ Kun suunnittelijat ymmärtävät suunnittelumalleja, kommunikointi tehostuu.

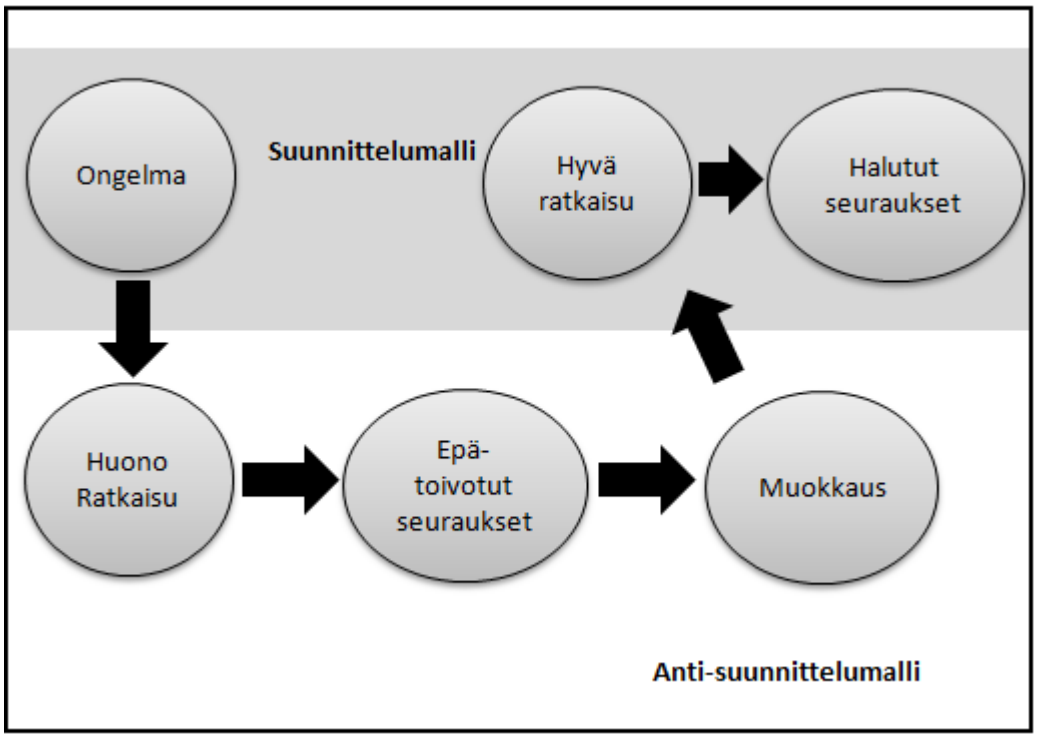
\--- Suunnittelumalli edistää jotain ohjelmiston laatuominaisuutta, mutta tavallisesti heikentää toista.

--- Monimutkaistavat järjestelmää lisäämällä luokkien, komponenttien ja rajapintojen määrää.

--- Suunnittelumalleilla ei ole omaa esitysmuotoa, joten ne pitää kuvata erikseen dokumenteissa tai koodin kommenteissa.

Antisuunnittelumallit

Antisuunnittelumallit ovat yleisesti esiintyviä ratkaisuja yleisiin ongelmiin mutta ratkaisut eivät ole mitenkään hyviä, vaan niissä on suuria ongelmia, jotka esiintyvät erilaisina epätoivottuina ominaisuuksina ohjelmiston elinkaaren aikana. Antisuunnittelumalliin voidaan myös liittää ohjeet, joiden avulla huono ratkaisu voidaan korvata hyvällä ratkaisulla.



Kuva 4.1 Antisuunnittelumallin käyttö (Koskimies, Mikkonen, 2005)

Huonon ratkaisun kuvaava antisuunnittelumalli voi olla aivan yhtä hyödyllinen kuin hyvän ratkaisun kuvaava suunnittelumalli. Antisuunnitelmamallien hyöty perustuu siihen, että niiden avulla voi löytää ohjelmasta kohtia jotka heikentävät ohjelmiston laatua. Jos tälläiseen antisuunnittelumalliin on liitetty oikean

suunnittelumallin käyttöön perustuvat ohjeet, antisuunnittelumalli voidaan nähdä suunnittelumallina, jossa ongelmayhteys on kuvattu huonolla ratkaisulla.

Yleisiä antisuunnittelumalleja ovat esimerkiksi:

- The blob
- Lava flow
- Golden Hammer
- Spaghetti Code
- Copy and Paste programming

Tyypilliä ominaisuuksia joita ohjelmistolla saattaa esiintyä, kun käytetään antisuunnittelumalleja ovat:

- Sama koodi toistuu uudestaan ja uudestaa eri paikoissa.
- Eri vaihtoehtojen staattinen kiinnittäminen ohjelmakoodissa periytymisen ja dynaamisen sidonnan käytön sijaan.
- Tiedon tyypin perusteella tapahtuva haarautuminen.
- Useiden komponenttien kautta kulkevat saantiketjut.
- Suuret ja monimutkaiset luokat.
- Epäselvät rajapinnat.

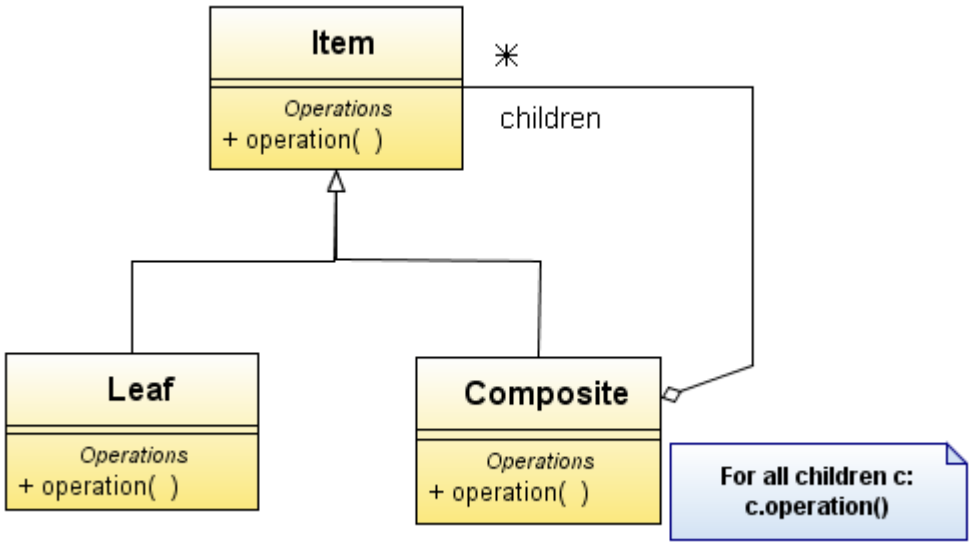
Antisuunnittelumalleja käytettäessä ohjelmisto saattaa olla vaikea ylläpitää, hankala uudelleenkäyttää, vaikea laajentaa, vaikea ymmärtää ja ohjelmisto voi olla yleisesti ottaen tehoton.

Esimerkki: Rekursiokooste-suunnittelumalli

Nimi: RekursioKooste

Ongelma: Miten organisoida hierarkinen oliokokoelma niin, että kokoelman käyttäjän ei tarvitse tuntea hierarkian rakennetta?

Ratkaisu:



Soveltuvuus:

Rekursiokoostetta kannattaa käyttää, kun

- haluat esittää periaatteessa mielivaltaisen syviä hierarkisia oliorakenteita, ja
- haluat, että rakenteen käyttäjälle koko rakenne näkyy yhtenä abstraktiona.

Seuraukset:

- Rakenteen käyttäjä ei tule riippuvaiseksi rakenteen eri osista.
- Rakenteen käyttäjän koodi yksinkertaistuu.
- Helppo lisätä uuden tyyppisiä rakenneosia.

- Ratkaisu ei suoraan tue tilannetta, jossa vain joidenkin lehtiluokkien ilmentymiä sallitaan tietyssä oliohierarkiassa.

Toteutusnäkökohtia:

- Tarvitaanko linkit myös lapsista koosteolioon?
- Voiko sama olio olla lapsena useammalle koosteelle?
- Määritelläänkö koosteluokan omat operaatiot (esim. lisää lapsi) yhteisessä rajapintaluokassa (Item)?
- Onko lapsille määritelty järjestys?
- Kuka luo ja hävittää hierarkian oliot?
- Millaista tietorakennetta käytetään lasten tallettamiseen?

Esimerkki lähteestä: (Koskimies, Mikkonen, 2005)

Yhteenveto kappaleesta

- Suunnittelumalli kuvaa yleistä tapaa jonkin usein esiintyvän ongelman ratkaisemiseksi.

- Suunnittelumalleilla saadaan siirrettyä kokeneiden suunnittelijoiden kokemus nuorempien käyttöön.

- Suunnittelumallit parantavat suunnittelijoiden keskinäistä kommunikointia antamalla tietyille suunnitteluratkaisuille kaikkien tunteman nimen.

- Aina ei ole tarpeen käyttää suunnittelumalleja, mutta oikein käytettyinä ne parantavat ohjelmiston laatua.

- Yleisiä ongelmia suunnittelumallien käytöstä ovat esimerkiksi suorituskykyongelmat ja suuri määrä luokkia ja monimutkaiset suhteet niiden välillä.

5. ARKKITEHTUURITYYLIT

Arkkitehtuurityylit ovat yleisiä malleja, jotka määräävät kuinka järjestelmä organisoidaan ylimmällä abstraktiotasolla ja kertovat järjestelmän teknisen luonteen.

Arkkitehtuurityyleillä ja suunnittelumalleilla on paljon yhteistä, mutta rajanvetoina voidaan pitää kahta asiaa:

- Suunnittelumallista on useita instansseja järjestelmässä, ja järjestelmässä voi olla useita suunnittelumalleja käytössä.

- Arkkitehtuurityyli määrää järjestelmän kokonaisrakenteen.

Ryhmittelyyn käytettävät arkkitehtuurityylit

Arkkitehtuurityyliä voidaan käyttää järjestelmän hahmotteluun. Tällöin arkkitehtuurityylin mukaisessa järjestelmässä joukko komponentteja on samassa roolissa tyylin kannalta. Yleisimmät ryhmittelyyn käytettävät arkkitehtuurityylit ovat kerrosarkkitehtuuri ja tietovuoarkkitehtuuri. Edellämainituista varsinkin kerrosarkkitehtuuria voidaan käyttää lähes minkä tahansa järjestelmän kuvaamiseen.

5.1.1 Kerrosarkkitehtuurit

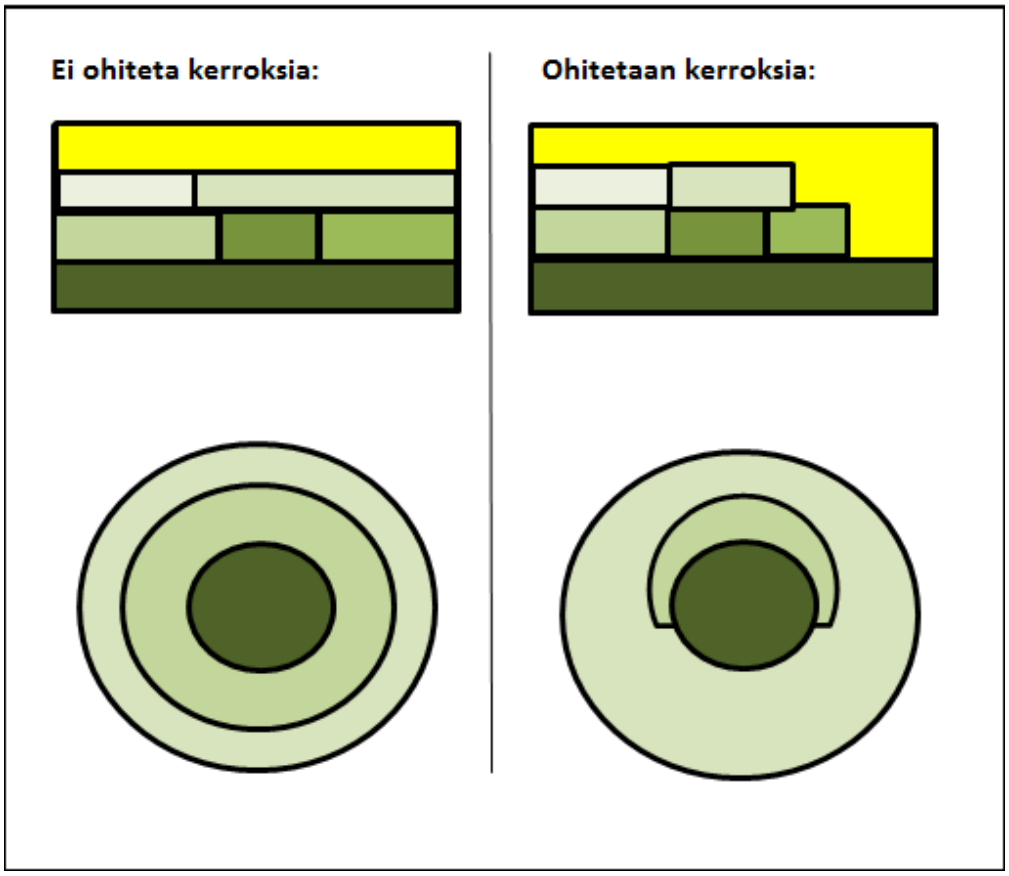
Kerrosarkkitehtuuri koostuu tasoista, jotka ovat järjestetty jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen. Tämä periaate yleisesti tarkoittaa akselia ihminen-laite. Laitepäässä on tarjolla primitiiviset käyttöjärjestelmää lähellä olevat toiminnot ja ihmispäässä on esimerkiksi graafiseen käyttöliittymään liittyvät palvelut. Käytännössä kerrokset voi olla

vaikea tunnistaa ja kerrostusperiaatteen ratkaisuna on se, että ylempi kerros käyttää alemman palveluja.

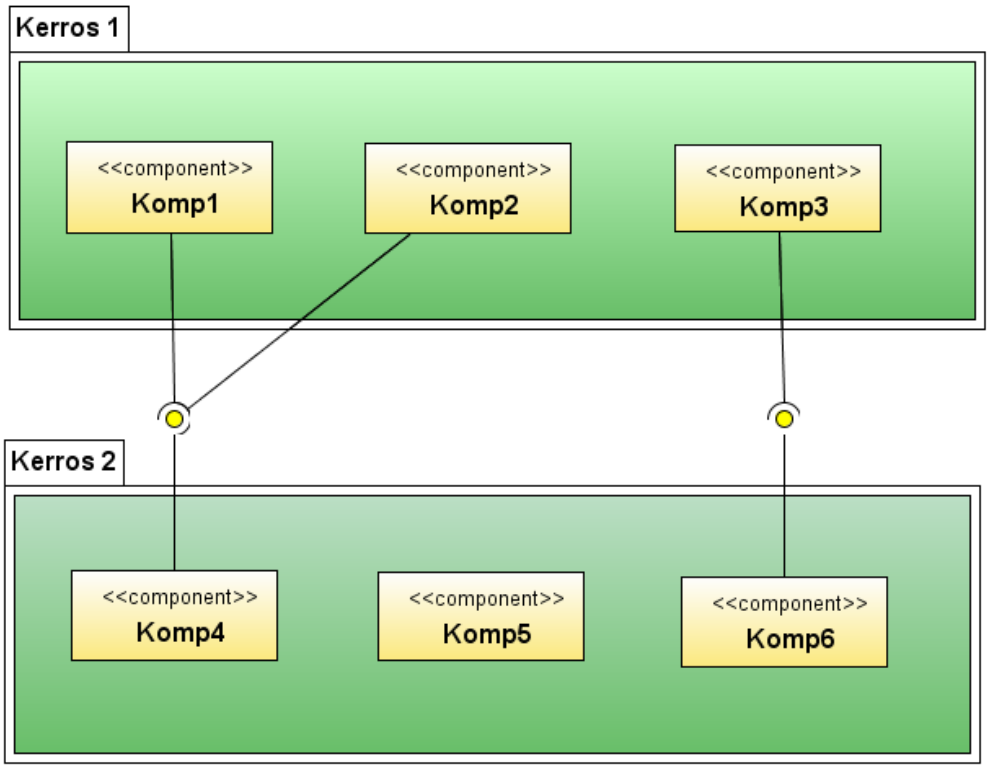
Yleistä

Kerrosarkkitehtuurin perusajatuksena on, että tietyllä tasolla oleva komponentti tai palvelu toteutetaan käyttämällä hyväksi alemman tason palveluita tai komponentteja. Tälläinen puhdas kerrosarkkitehtuuri tosin on hyvin harvinainen, koska usein esimerkiksi tehokkuussyistä on tehtävä poikkeamia. Poikkeamia on kahdenlaisia: palvelukutsu kulkeutuu alhaalta ylöspäin tai ylhäältä useamman kuin yhden kerroksen alaspäin. Poikkeama, jossa kutsu kulkeutuu alaspäin useamman kerroksen, ei sinänsä ole vakava rikkomus, mutta liikaa käytettynä se johtaa kerrosarkkitehtuurin idean heikentymiseen. Sen sijaan alhaalta ylöspäin kulkeva kutsu on vakavempi ongelma, jos alempi kerros tämän vuoksi on riippuvainen ylemmästä. Joskus tälläinen alhaalta ylöspäin kutsu on yksinkertaisesti tarpeen ja tälläinen hoidetaan takaisinkutsu periaatteella, jolloin saadaan alempi kerros riippumattomaksi ylemmästä.

Kerrosarkkitehtuureita voi kuvata useilla tavoilla, joista suosituin on esitetty kuvassa 5.1, jossa kuvataan kerrosten ohitukset porrastamalla. Kullakin kerroksella on rajapintansa, jotka se tarjoaa ja jotka se tarvitsee (Kuva 5.2). Ylemmän kerroksen tarvitsemien rajapintojen tulisi täsmätä alemman kerroksen tarjoamien rajapintojen kanssa. Jos näitä rajapintoja ei ole selkeästi määritelty, ne voidaan selvittää irrottamalla kerros järjestelmästä ja tutkimalla, mitkä palvelut jäävät ylemmältä kerrokselta toteuttamatta ja mitkä jäävät ilman toteuttajaa irrotetussa kerroksessa. Kun rajapinnat ovat selvillä, kerroksen toteutus voidaan vaihtaa toiseksi ilman vaikutusta muuhun järjestelmään.



Kuva 5.1 Kerrosarkkitehtuurin kuvaaminen (HY-kalvot 2008)



Kuva 5.2 Kerrosten välinen rajapinta

Hyvin tyypillinen esimerkki kerrosarkkitehtuurista on liiketoimintajärjestelmä, joka on jaettu neljään kerrokseen (Kuva 5.3). Alimmassa kerroksessa on yleinen infrastruktuuri, sen yläpuolella sovellusalueen logiikka ja käsitteet, tämän jälkeen on yksittäisen sovelluksen logiikan toteuttava kerros, ja ylimpänä on yksittäisen sovelluksen käyttöliittymän toteuttava kerros. Tälläisestä arkkitehtuurista voidaan helposti uudelleenkäyttää kahta alinta kerrosta, kun tehdään sovellusta liittyen samaan liiketoimintalaan.

Kuva 5.3 Yleinen esimerkki kerrosarkkitehtuurin käytöstä (Koskimies, Mikkonen, 2005)

Kerrosarkkitehtuuria suunnitellessa on hyvä analysoida kerrosten riippuvuuksia esimerkiksi taulukkojen 5.1 ja 5.2 kaltaisten matriisien avulla, varsinkin silloin, kun kerrosjako ja kerrosten väliset riippuvuudet eivät ole vielä täysin selvillä. Kun matriisin alkioon laitetaan risti, olettaen että rivikerros riippuu sarakekerroksesta, ristien pitäisi hyvässä kerrosarkkitehtuurissa olla samassa linjassa, ilman useamman kerroksen loikkia.

Taulukko 5.1 Kerrosarkkitehtuurin riippuvuusmatriisi, huono

rivikerros riippuu sarakekerroksesta	P1	P2	P3	P4	P5
P1		**x**	**x**		
P2				**x**	
P3		**x**		**x**	**x**
P4					**x**
P5					

Taulukko 5.2 Kerrosarkkitehtuurin riippuvuusmatriisi, hyvä

rivikerros riippuu sarakekerroksesta	P1	P2	P3	P4	P5
P1		**x**	**x**		
P2			**x**	**x**	
P3				**x**	**x**
P4					**x**
P5					

Soveltaminen

Kerrosarkkitehtuuri on hyvin yleinen malli, jota sovelletaan lähes kaikissa järjestelmissä pienemmässä tai suuremmassa mittakaavassa. Se jakaa järjestelmän korkealla tasolla karkeasti osiin ja helpottaa näin kokonaiskuvan saamista järjestelmästä. Kerrosarkkitehtuuria käytettäessä pyritään minimoimaan riippuvuudet tasojen välillä, josta seuraa se, että järjestelmää on helppo muuttaa ja ylläpitää. Kerrosarkkitehtuuri tukee ohjelmistojen uudelleenkäytettävyyttä siten, että alempien kerrosten päälle on helppo kasata uusia yksilöllisiä sovelluksia. Ongelmina kerrosarkkitehtuureissa mainittakoon tehokkuushäviö ja poikkeusten käsittely. Tehokkuushäviö aiheutuu yleisesti kun palvelua pitää hakea alemmalta kerrokselta, joka taas hakee sitä alemmalta kerrokselta ja tekee näin palvelun toteuttamisesta liian epäsuoraa. Myös

poikkeusten käsittely ongelma aiheutuu, kun palvelupyyntö lähtee ylimmältä kerrokselta alaspäin ja jollakin alemmalla kerroksella tapahtuu virhe, josta lähetetään viestiä ylöspäin jotta jokin kerros voisi käsitellä sen. Ongelma edellämainitussa on se, että poikkeus käsitellään ylemmällä tasolla kuin missä se on syntynyt, eikä sitä enää pystytäkkään korjata. Pahimmassa tapauksessa virheilmoitus etenee alimmalta kerrokselta aina käyttäjälle saakka.

Edut ja ongelmat

Etuja

- Melkein aina sovellettavissa
- Tukee järjestelmän ymmärtämistä ja hallintaa
- Vähentää riippuvuuksia (ylläpidettävyys, muunneltavuus)
- Helppo liittää yrityksen organisaatioon
- Tukee uudelleenkäyttöä (tuoterunkoarkkitehtuurin pohjana)

Ongelmia

- Suorituskyky voi heikentyä (epäsuoruus)
- Saattaa johtaa tarpeettomaan/moninkertaiseen laskentaan
- Poikkeusten käsittely

5.1.2 Tietovuoarkkitehtuurit

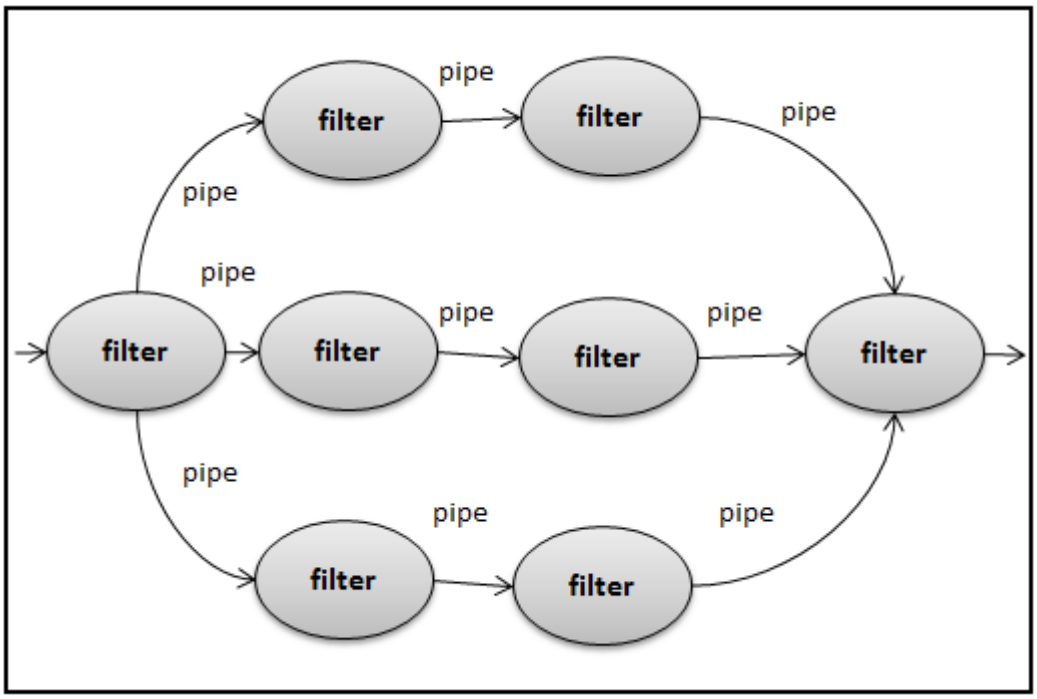
Tietovuoarkkitehtuuri sopii järjestelmiin, joissa jalostetaan ja prosessoidaan tietovirtoja.

Yleistä

Tietovuoarkkitehtuuri (Kuva 5.4) koostuu komponenteista (filter), jotka tuottavat ja kuluttavat tietoalkioita, ja tietoalkioita komponentilta toisille kuljettavista väylistä (pipe). Jotta tietovuoarkkitehtuuria voidaan soveltaa, täytyy kunkin komponentin toimia itsenäisesti lukemalla omaa syötevirtaansa ja tuottamalla omaa syötevirtaansa eikä komponentti saa olla riippuvainen muista

komponenteista. Näillä komponenteilla ei myöskään saa olla jaettua tilatietoa eikä niiden edes tarvitse tuntea toisiaan.

Tietovuoarkkitehtuurin yksinkertaisin ja yleisin muoto on liukuhihna-arkkitehtuuri, jossa tietovirta etenee ilman haarautumisia yhtä prosessointi linjaa pitkin. Tälläisessä mallissa tiedonvälitys voidaan toteuttaa kahdella tavalla, työntämällä tai vetämällä tietoa. Työntämisessä tiedon alkuperäinen tuottaja kutsuu ensimmäistä prosessointiyksikkö antamalla sille parametrina ensimmäisen tietoalkion. Prosessointiyksikkö tekee työnsä ja lähettää oman tuotoksensa seuraavalle yksikölle, joka tekee työnsä ja lähettää tuotoksensa eteenpäin ja tätä jatkuu kunnes viimeinen yksikkö kutsuu tietovirran lopullista käyttäjää ja antaa parametrina sen tuottaman tietoalkion. Tätä jatketaan kunnes koko tietovirta on käsitelty. Vetämisessä tehdään sama asia mutta aloitetaan toisesta päästä.



Kuva 5.4 Tietovuoarkkitehtuuri

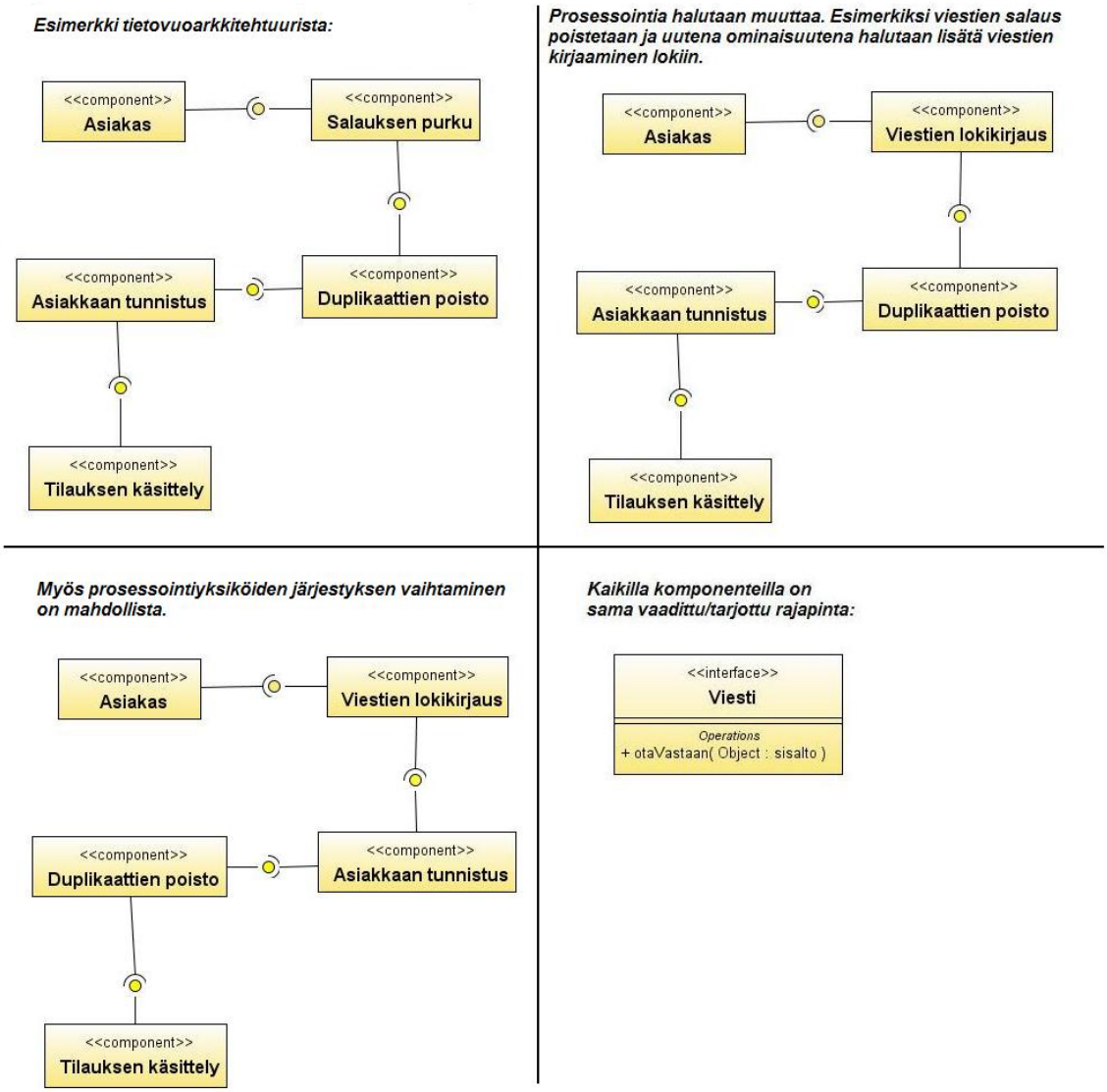
Tietovuoarkkitehtuuri tarjoaa tavan rinnakkaistaa järjestelmän, koska jokainen komponentti voi toimia itsenäisesti omana prosessinaan muiden kanssa. Tämä voi olla hyödyllistä laskennan hahmottamisessa, järjestelmän tehostamisessa tai ihan vain siksi, että komponentit voivat toimia eri koneissa verkon yli.

Rinnakkaistetun tietovuoarkkitehtuurin tehostaminen toteutetaan puskureiden avulla, jolloin peräkkäin toimivien komponenttien ei tarvitse toimia samaan tahtiin.

Soveltaminen

Tietovuoarkkitehtuurin avulla voidaan käsitellä mutkikas tieto asteittan, jalostamalla tietoa vähän kerrallaan. Tällä tavalla vaikeakin tietojenkäsittelytehtävä voidaan ymmärtää ja toteuttaa hallitusti. Tietovuoarkkitehtuurin käyttö mahdollistaa myös varianssia, koska jokainen komponentti toimii itsenäisesti. Olennaista on vain se, että komponentit ymmärtävät omaa syötevirtaansa. Tämä mahdollistaa komponenttien vaihtamisen toisiin, kunhan uudet komponentit ymmärtävät samaa syötevirtaa. Arkkitehtuuri tarjoaa myös tavan rinnakkaistaa järjestelmän. Kuvassa 5.5 on havannoillistettu tietovuoarkkitehtuurin tarjoamaa varianssia.

Tietovuoarkkitehtuuri ei sovi interaktiivisiin järjestelmiin, koska tietovuoarkkitehtuurin perustavassa järjestelmässä ei ole tarkoituskaan jakaa mitään globaalia tietoa osien kesken. Myöskin tiedon välittäminen voi joskus johtaa tarpeettomaan työhön. Poikkeusten käsittely on myöskin ongelma tietovuoarkkitehtuurissa. Jotta komponentit pystyvät jatkamaan työtänsä tietovirrassa ilmenneiden virheiden takia, tarvitaan erilaisia virheistätoipumistekniikoita. Nämä voivat perustua esimerkiksi turvallisten tietoalkioiden määrittelyyn, jolloin esimerkiksi kun tietovirrassa havaitaan virhe, komponentti ohittaa tietoalkioita niin kauan, että löytää seuraavan turvallisen tietoalkion ja jatkaa sitten työtänsä.



Kuva 5.5 Esimerkki tietovuoarkkitehtuurista (HY-kalvot 2008)

Edut ja ongelmat

Etuja

- Mutkikas tiedon käsittelyprosessi voidaan jakaa helpommin hallittaviin palasiin
- Tukee uudelleenkäyttöä: prosessointiyksiköitä voidaan kombinoida eri tavoin
- Tukee ylläpitoa: prosessointiyksikkö voidaan helposti vaihtaa
- Tukee rinnakkaisuutta

Ongelmia

- Ei sovi interaktiivisille järjestelmille
- Tiedon tulkinnasta tulee suorituskykyrasitetta
- Virhetilanteiden käsittely voi olla vaikeaa

Palveluperustaiset arkkitehtuurityylit

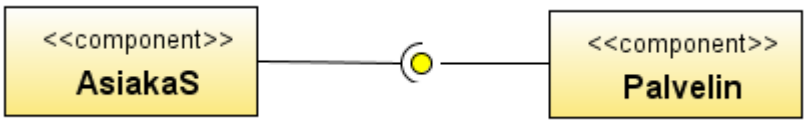
Palveluperustaiset arkkitehtuurityylit rakennetaan niin, että niissä on kahdenlaisia rooleja. On palveluntarjoajia ja niiden käyttäjiä. Nämä roolit eivät tosin ole tiukkoja, koska palvelun tarjoaja voi tarvittaessa käyttää jonkin muun palveluita. Palvelu perustuu usein johonkin resurssiin, jonka palveluita sen ympärille rakennettu komponentti tarjoaa ymäristölleen. Käytännössä tälläisellä komponentilla voidaan valvoa resurssin käyttöä, joka onkin usein syynä tälläisen arkkitehtuurityylin valitsemiseen.

5.1.3 Asiakas-palvelin- arkkitehtuurit

Asiakas-palvelin-arkkitehtuuri on ehkä yleisimmin käytetty arkkitehtuuriratkaisu. Perusajatuksena on että kapseloidaan tietyn arkkitehtuuritason resurssin hallinta siten, että resurssin käyttäjien ei tarvitse huolehtia resurssin käyttöön liittyvistä teknisistä ongelmista.

Yleistä

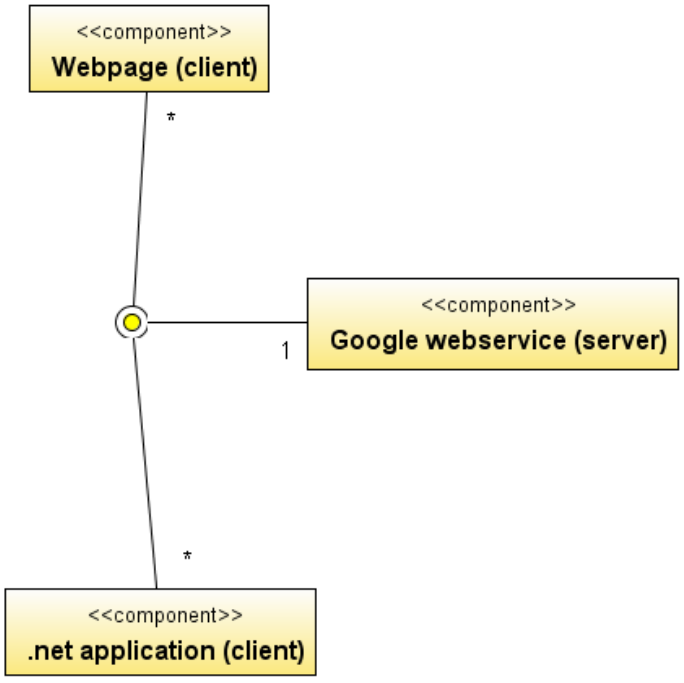
Tyypillisesti (mutta ei välttämättä) asiakkaan ja palvelimen välinen vuorovaikutus tapahtuu istunnon (session) puitteissa. Istunnon aikana suoritetaan asiakkaan pyytämiä palveluita, ja kun asiakas on saanut työnsä tehtyä se sulkee istunnon. Palvelimet odottavat yleensä passiivisena, kunnes asiakas ottaa niihin yhteyttä. Palvelin huolehtii myös istuntoon liittyvistä transaktioista ja toimii aina omassa säikeessään tai prosessissaan, mikä pitää sen erillään asiakkaista.



Kuva 5.6 Yksinkertainen asiakas-palvelin arkkitehtuuri

Esimerkkinä(HY-kalvot 2008)(Kuva 5.7) asiakas-palvelin-arkkitehtuurista on Googlen tarjoama verkkohakutoiminnon verkkopalvelu (web service)

- Erilaiset asiakkaat käyttävät SOAPin (= XML-perustainen HTTP:n päälle rakennettu protokolla) avulla palvelun rajapintaa
- Webpage-servlet tarjoaa hakutoiminnon osana ”omia sivujaan”
- .NET-sovellus hakee hintatietoja
- Eri asiakkailla erilaisia tarkoituksia, palvelin tarjoaa saman palvelun kaikille
- Järkevää pitää palvelin ja asiakkaat erillään



Kuva 5.7 Esimerkki asiakas-palvelin arkkitehtuurista (HY-kalvot 2008)

Soveltaminen

Asiakas-palvelin-arkkitehtuuri mahdollistaa selkeän työnjaon, joka voi toimia myös pohjana hajauttamiselle. Tästä tosin seuraa palvelun suorituksen hitaantuminen, koska etämetodikutsu on huomattavasti hitaampi kuin paikallisen palvelun kutsu. Hajautetuissa järjestelmissä useat tietovarastoa käyttävät järjestelmät perustuvat asiakas-palvelin-arkkitehtuuriin. Tällöin palvelin hallitsee tietovaraston käyttöä ja sitä voi käyttää vain palvelimen kautta. Myös erilaiset hajautetut liiketoimintajärjestelmät ja niille tarkoitetut ohjelmistoalustat (esim. EJB, CORBA) soveltavat asiakas-palvelin-arkkitehtuuria (Koskimies, Mikkonen, 2005). Syynä hajautettavuuteen on se, että yleensä asiakkaat ja palvelimet suunnitellaan toisistaan riippumattomiksi jo etukäteen ja ajatellaan että ne tulevat sijaitsemaan eri prosesseissa. Tällä tavalla saadaan myöskin eristettyä palvelimien ja asiakkaiden ongelmat. Esimerkiksi jos asiakas toimii virheellisesti, voi palvelin silti palvella muita asiakkaita. Asiakas-palvelin-arkkitehtuuria ajatellaan usein hajautettuna järjestelmänä mutta sitä ei ole sidottu hajautukseen vaan sitä voidaan käyttää myös missä tahansa ympäristössä eristämään jokin resurssi oman hallintayksikön valvontaan.

Edut ja ongelmat

Etuja

- Helpottaa yhteisen resurssin hallintaa (esim. tietoturva)
- Helpottaa ylläpitoa ja muunneltavuutta (esim. palvelimen vaihto)
- Hyvä teknologinen tuki

Ongelmia

- Verkkoliikenteestä johtuvat suorituskykyongelmat
- Palvelinkeskeisyys: kriittisen palvelimen vikaantuminen
- Poikkeusten käsittely

5.1.4 Viestinvälitysarkkitehtuurit

Oletetaan tilanne, jossa suunnitellaan järjestelmää, josta tiedätään vain, että siihen on tulossa joukko keskenään kommunikoivia komponentteja, mutta ei tiedetä montako, minkälaatuisia, eikä tiedetä tarkkaan millaista tietoa näiden komponenttien pitäisi pystyä käsittelemään. Tälläisessä tilanteessa on vaikeaa ja jopa riskialtista alkaa suunnittelemaan järjestelmää kiinteillä staattisilla rajapinnoilla, koska kaikkea ei voi vielä määritellä. Tällöin on hyvä käyttää viestinvälitysarkkitehtuuria.

Yleistä

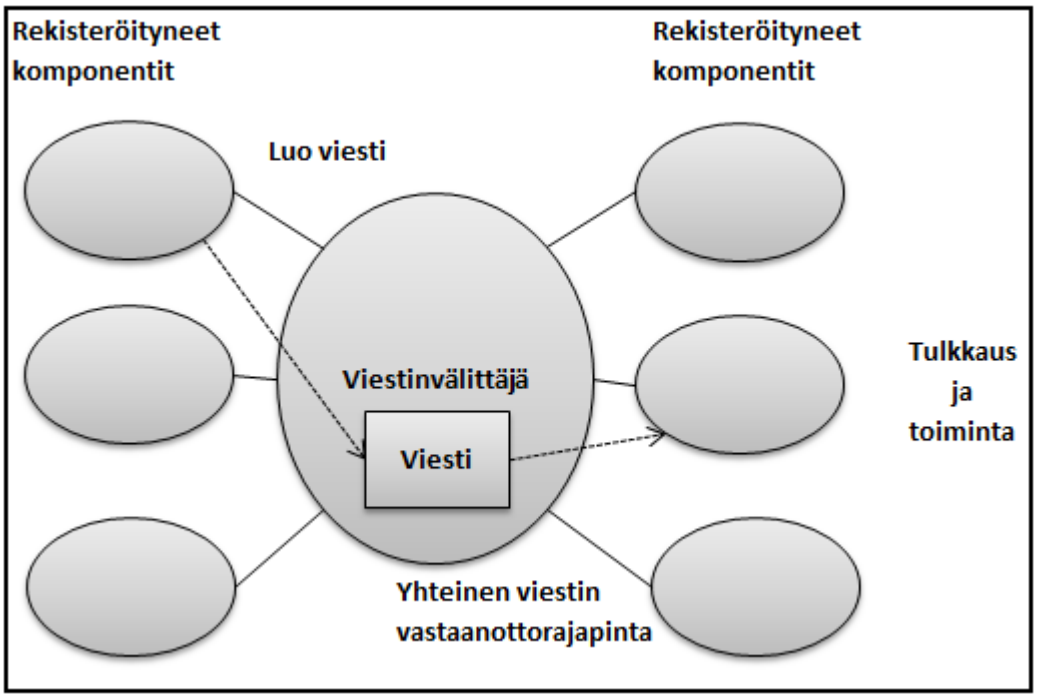
Viestinvälitysarkkitehtuurilla tarkoitetaan arkkitehtuuria, jossa komponentit kommunikoivat keskenään keskitetyn viestinvälittäjän kautta. Komponenteilla on yhteinen rajapinta, joka sisältää tarvittavat operaation viestien vastaanottamiseen. Viestit kertovat välittäjälle mitä sen pitäisi tehdä. Tälläinen toiminta tarkoittaa periaatteessa dynaamista rajapintaa, sillä sisällöltään uudenlainen viesti ei muuta järjestelmän staattista rakennetta, mutta uusi komponentti voi käsitellä sen ja tuoda uutta toiminnallisuutta järjestelmään. Toteutus voitaisiin tehdä esimerkiksi niin, että komponentit rekisteröityvät välittäjälle ilmoittaen olevansa kiinnostuneita tietyn tyylisistä viesteistä ja välittäjä puolestaan toimittaisi viestit komponenteille sitä mukaa kun niitä lähetetään, tai käyttämällä konfiguraatiotiedostoja, joiden kanssa voitaisiin tehdä viestien reititys. Viestinvälitys arkkitehtuurin toimintaperiaatetta on selvennetty kuvassa 5.8.

Viestinvälitysarkkitehtuurin määrittelevät seuraavat ominaisuudet:

- keskenään kommunikoivien komponenttien joukko
- viestit, joiden avulla komponentit kommunikoivat ilman, että viestin lähettäjä tietää minne viesti pitää toimittaa tai vastaanottaja mistä viesti on peräisin
- operaatiot, joilla komponentit reagoivat viesteihin
- säännöt, joiden avulla komponentit ja viestit rekisteröidään järjestelmälle

- säännöt, joiden avulla välittäjä tietää mille komponentille viesti on lähetettävä
- rinnakkaisuusmalli: missä määrin komponentit ja välittäjä toimivat rinnakkain (Koskimies, Mikkonen, 2005)

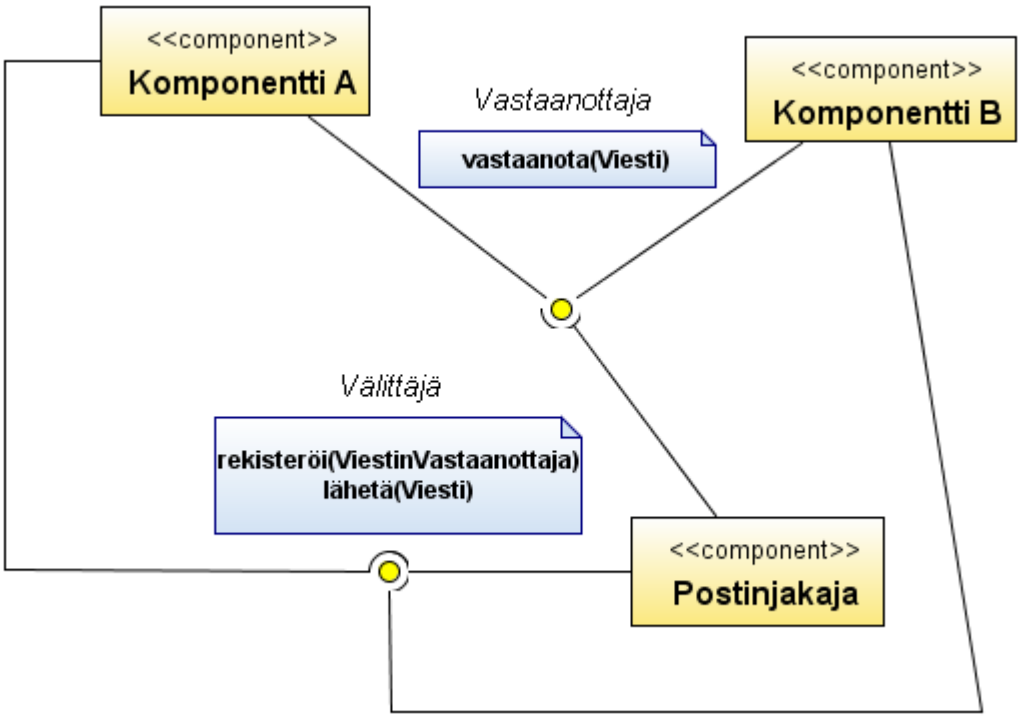
Nykyisin ohjelmistoja suunnitellessa joudutaan ottamaan huomioon ennaltatuntemattomia vaatimuksia tai yleisyyttä ja helppoa laajennattavuutta koskevia vaatimuksia. Tällöin on hyvä välttää staattisia rajapintoja. Tähän viestinvälitysarkkitehtuuri tarjoaa hyvän vaihtoehdon, sillä staattinen rajapinta koostuu yksinkertaisimmillaan komponenttien osalta viestin vastaanotosta ja viestinvälittimen osalta lähetyksestä ja rekisteröitymisestä. Varsinainen toiminta kootaan tämän perusrakenteen päälle käyttämälle viestejä laukaisemaan toimintoja eri komponenteissa.



Kuva 5.8 Viestinvälitysarkkitehtuuri (Koskimies, Mikkonen, 2005)

Soveltaminen

Viestinvälitysarkkitehtuurin etuna on uusien komponenttien lisäämisen ja poistamisen helppous muuttamatta järjestelmää itsessään. Tämä tekee järjestelmästä joustavan ja dynaamisesti muodostettavan. Viestinvälitysarkkitehtuuri tukee synkronista että asynkronista viestinvälitystä ja rinnakkaisuutta. Ongelmana arkkitehtuurissa on viestien muodostamisen ja tulkitsemisen aiheuttama potentiaalinen tehottomuus. Myöskin ylläpitäminen voi olla hankalaa, koska järjestelmässä ei ole selkeitä staattisia rajapintoja ja täytyy ymmärtää myös viestien ajoaikainen rakenne ja sisältö.



Kuva 5.9 Esimerkki viestinvälitysarkkitehtuurista (HY-kalvot 2008)

Kuvassa 5.9 on luokkakaavio esimerkki viestinvälitysarkkitehtuurista. Kommunikoivilla komponenteilla on yhteinen rajapinta viestien vastaanottamiseen (Vastaanottaja). Viestien välittämisen hoitaa erityinen komponentti (Postinjakaja). Välittäjäkomponentti tarjoaa rajapinnan, jonka kautta voidaan (1) rekisteröityä viestin vastaanottajaksi, (2) lähettää viestejä.

Viesti sisältää informaation, joka kertoo, mitä vastaanottavan komponentin tulee tehdä

Edut ja ongelmat

Etuja

- Helppo muuttaa, lisätä ja poistaa komponentteja tai sovelluksia
- Vikasietoinen (esim. jos viestillä ei ole vastaanottajaa), voidaan esim. toistaa viestin lähettämistä
- Joustava järjestelmäkonfiguraatio
- Sallii heterogeeniset järjestelmät, sovellusintegraation
- Sallii sekäsynkronisen ettäasynkronisen kommunikoinnin

Ongelmia

- Tehokkuus: viestien kirjoittaminen ja lukeminen
- Vaikeampi toteuttaa, testata ja ymmärtääkuin perinteinen
- Jotkut ”tavalliset” asiat vaativat erityistukea (esim. peräkkäisyys, synkronisuus)
- Olemassaoleva infratuki teknologiariippuvaista
- Syntyy helposti implisiittisiä riippuvuuksia yksiköiden välille

Sovellusaluekohtaiset arkkitehtuurityylit

Joillekin sovelluksille on vakiintunut tietynlainen toteutustapa. Toteutusteknisesti sovellusaluekohtaista arkkitehtuurityyliä noudattava järjestelmä voidaan rakentaa palveluiden ja jonkinlaisen kerrostamisen varaan. (Koskimies, Mikkonen, 2005)

5.1.5 Malli-näkymä-ohjain arkkitehtuurit

Malli-näkymä-ohjain-arkkitehtuurin (MVC, model-view-controller) ajatuksena on erottaa käyttöliittymä sovelluslogiikasta. Tämän vuoksi käyttöliittymää on helppo muokata ja siirtää toisille graafisille alustoille. Lisäksi käyttöliittymässä pystytään

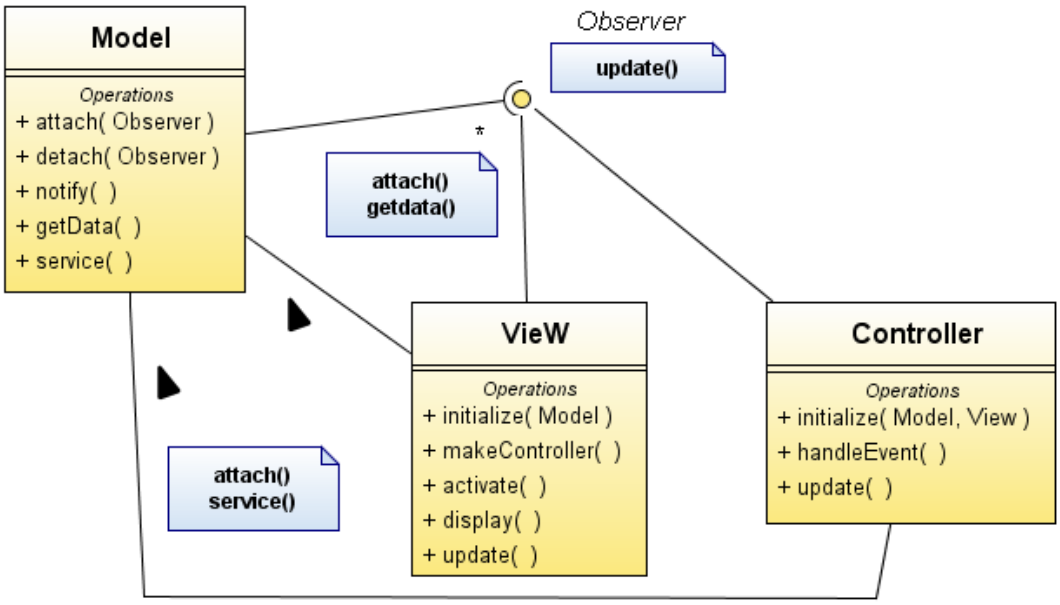
näyttämään sovelluksessa käytettävät tiedot eri näkymissä aina oikeina ja päivitettyinä.

Yleistä

Tätä arkkitehtuuria käyttävä järjestelmä jaetaan kolmenlaisiin osiin:

- Malleihin (Model), jotka edustavat jotakin osaa sovellusdatasta tai loogisesta sovelluksen tilasta.
- Näkymiin (View), jotka edustavat jotakin osaa käyttöliittymästä.
- Ohjaimiin (Controller), jotka toimivat sovittimina näkymien ja mallien välillä ja vastaavat että näkymien ja mallien tiedot vastaavat.

Tästä arkkitehtuurista on olemassa erilaisia variaatioita, mutta tässä tutustutaan kuvassa 5.10 näkyvään malli-näkymä-ohjain arkkitehtuurin, jossa käytetään hyväksi tarkkailija-suunnittelumallia. View ja controller toteuttavat Observer-rajapinnan, jonka kautta ne voivat rekisteröityä haluamilleen malleille tarkkailemaan niiden muutoksia. Jos muutoksia tapahtuu, mallit ja näkymät voivat kysyä mallilta sen muuttunutta dataa. Malli tarjoaa operaatiot, joilla kiinnostuneet voivat ilmottautua tarkkailemaan sen muutoksia, ja ilmoittaa näille kun muutoksia on tapahtunut. Näkymä huolehtii, että näyttö päivittyy vastaamaan mallin tilaan. Ohjain ottaa vastaan käyttäjän komentoja ja muuntaa ne loogisiksi sovellustoiminnoiksi.



Kuva 5.10 Malli-näkymä-ohjain-arkkitehtuuri (Koskimies, Mikkonen, 2005)

Soveltaminen

Malli-näkymä-ohjain-arkkitehtuuri mahdollistaa mallin uudelleenkäytön useissa tilanteissa ja onkin usein luonnollinen valinta graafisen käyttöliittymän toteuttamiseksi. Arkkitehtuurilla on tosin muutamia ongelmia. Tyypillisesti se monimutkaistaa järjestelmää kasvattamalla luokkien määrää ja tarkkailija-suunnittelumallin soveltaminen tarkoittaa enemmän päivityskutsuja. Lisäksi näkymä- ja ohjainluokat liittyvät toisiinsa ja niitä on vaikea uudelleenkäyttää toisistaan irrallaan muissa yhteyksissä.

Edut ja ongelmat

Etuja

- Helppo toteuttaa useita näkymiä samaan tietoon.
- Kaikki näkymät ovat automaattisesti synkronoituja.
- Uusia näkymiä voidaan ajoaikana liittää järjestelmään.
- Käyttöliittymän ulkoasu suhteellisen helposti vaihdettavissa.

Ongelmia

- Mahdollisesti turhia näkymien päivityskutsuja.
- Mallidatan kyselyt voivat lisätä suoritusaikaa.

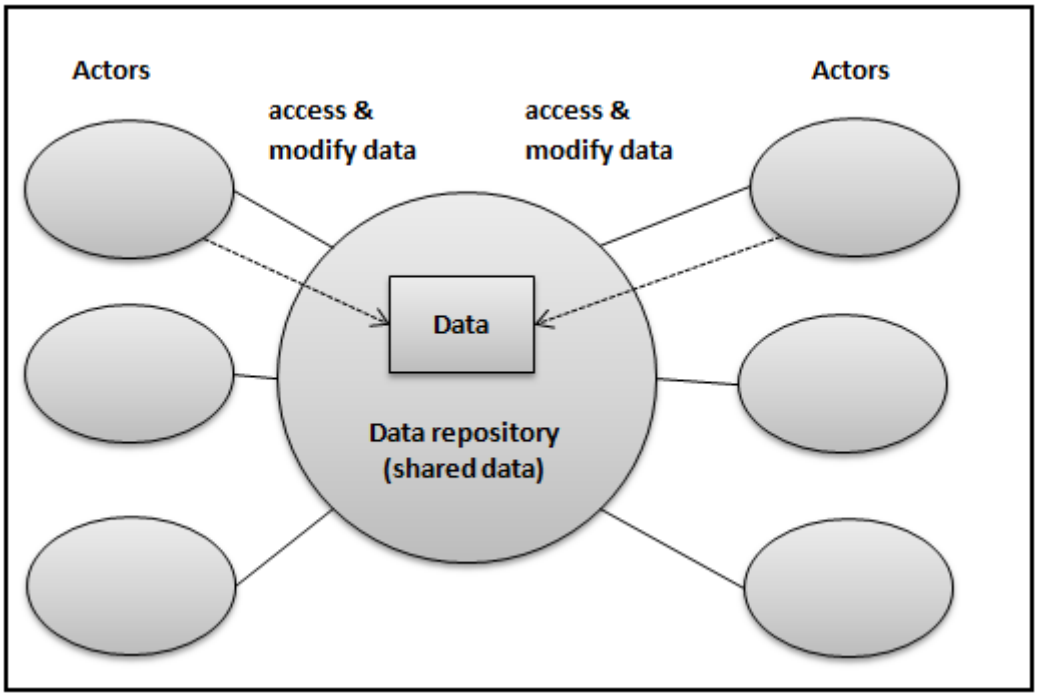
5.1.6 Tietovarastoarkkitehtuurit

Tietovarastoarkkitehtuurissa joukko järjestelmiä tai komponentteja ylläpitää yhteistä tilaa tietovarastossa. Arkkitehtuurista on olemassa erilaisia muunnoksia riippuen tietovaraston aktiivisuudesta.

Yleistä

Tässä arkkitehtuurissa ydin on jaettu tietovarasto, jota kaikki järjestelmän komponentit voivat tutkia ja muuttaa. Komponentit eivät kommunikoi suoraan vaan tietovaraston kautta. Jotta tiedoille ei tule ristiriitaisuutta, täytyy tietovaraston yleensä tukea transaktiota, eli jos tehdään tapahtumaa, jossa tietoa muutetaan useissa paikoissa, tehdään se aina kokonaan tai sitten ei ollenkaan. Kuvassa 5.11 on kuvattu tietovarastoarkkitehtuuri.

Esimerkkinä tietovarastoarkkitehtuurin käytöstä mainittakoon ohjelmointiympäristöt, joissa yhteinen tietovarasto koostuu ohjelman sisäisestä esityksestä, jota erilaiset työkalut käsittelevät (Koskimies, Mikkonen, 2005).



Kuva 5.11 Tietovarastoarkkitehtuuri (HY-kalvot 2008)

Soveltaminen

Tietovarastoarkkitehtuurilla voidaan toteuttaa luontevasti järjestelmän rinnakkaistaminen. Tietovarastoon kiinnittyviä komponentteja on helppo lisätä ja poistaa, koska ne eivät riipu toisistaan. Arkkitehtuurin avulla kaikilla osajärjestelmillä on sama tieto. Arkkitehtuurilla on myös ongelmansa. Esimerkiksi jatkuva tiedon tulkintaa sisältävä kommunikointi tietovaraston kanssa aiheuttaa tehottumuutta, varsinkin, jos kommunikointi tapahtuu verkon yli. Lisäksi tehottomuutta aiheutuu, jos joku sovellus lukitsee suuren osan tietovarastosta ja muut joutuvat odottamaan. Tietovaraston rajapinnan suunnittelu on vaikea tehtävä, koska rajapinnasta olisi tehtävä tarpeeksi yleinen, jotta useat sovellukset voivat sitä käyttää ja tietovaraston pitää silti säilyttää tehokkuutensa.

Edut ja ongelmat

Etuja

- Mahdollistaa rinnakkaistamisen – rinnakkaiset toimijat

- Keskitetty tieto takaa tiedon eheyden
- Toimijoiden riippumattomuus toisistaan – mahdollisuus lisätä ja poistaa

Ongelmia

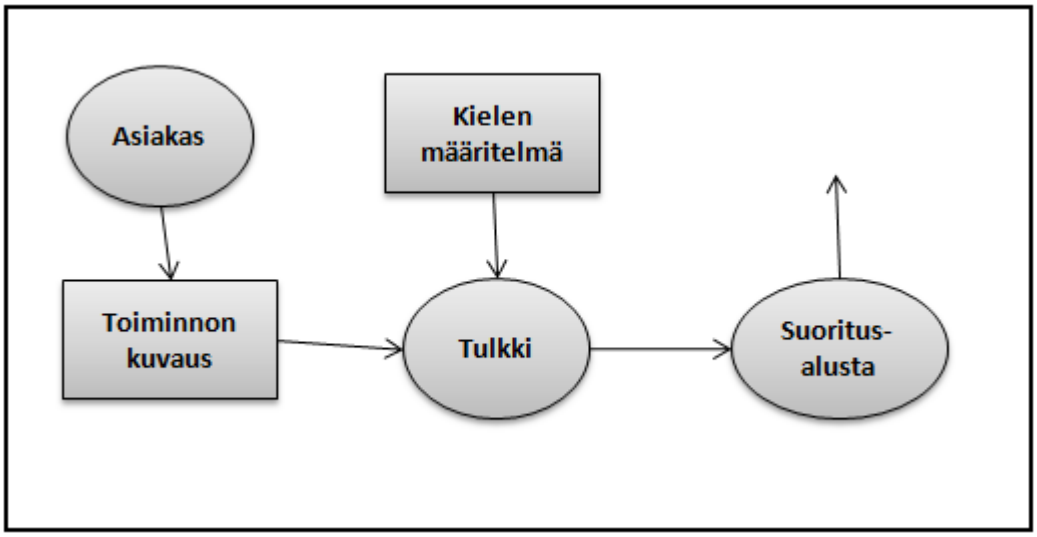
- Suorituskyky - erityisesti verkon yli kommunikoitaessa
- Tietovaraston rajapinnan muuttaminen vaikeaa
- Rajapinnan sopivuus kaikille toimijoille ei välttämättä optimaalinen

5.1.7 Tulkkipohjaiset arkkitehtuurit

Usein on tarpeen pystyä antamaan järjestelmälle syötteenä toiminnallisia kuvauksia. Tilanne voisi olla esimerkiksi sellainen, että järjestelmä tarjoaa peruspalveluita, mutta vasta ajoaikana tiedetään miten niitä on yhdisteltävä. Toinen tilanne voisi olla sellainen, että jollakin sovellusalueella on tietty abstrakti tapa kuvata toiminnallisuutta tällä alueella ja sovellusten halutaan toimivan useilla erilaisilla alustoilla, joiden avulla abstrakti toiminnallisuus voidaan toteuttaa. Tällöin eri alustoille halutaan toteuttaa järjestelmiä, jotka pystyvät suorittamaan abstrakteja toiminnallisia kuvauksia. Näiden suunnittelu johtaa usein tulkkipohjaiseen arkkitehtuuriin. (Koskimies, Mikkonen, 2005)

Yleistä

Perusajatuksena tulkkipohjaisessa arkkitehtuurissa on, että tulkki lukee askeleittain (käsky käskyltä) toiminnallista kuvausta ja kutsuu vastaavia alustan tarjoamia palveluita. Tätä toimintaa on havannoillistettu kuvassa 5.12. Esimerkkinä tulkkiarkkitehtuurin käytöstä mainittakoon SQL:n käyttö tietokantakyselyissä.



Kuva 5.12 Tulkkiperustainen arkkitehtuuri (HY-kalvot 2008)

Soveltaminen

Tulkkiarkkitehtuuri tekee suoritettavasta koodista ajoaikaisen tietorakenteen, joka on täysin järjestelmän hallinnassa, mikä mahdollistaa sen tutkimisen ja muuttamisen suorituksen aikana. Tulkittavaa kieltä on helppo muuttaa, varsinkin jos on käytetty Tulkki-suunnittelumallia. Kielen laajentaminen ei tee vanhoja kielellä kirjoitettu toimintakuvauksia epäkelvoiksi. Mutta toteutuksen muuttaminen tai rakenteita vastaavien luokkien poistaminen sen voi tehdä. Jos halutaan muuttaa pelkästään kielen tulkintaa mutta ei sen rakennetta, riittää että muutetaan tulkintaoperaation toteutusta. Tämä voidaan tehdä aliluokittamalla, joka helpottaa myös toteutusalustan vaihtoa.

Huonoa tässä arkkitehtuurissa on se, että suorituskyky kärsii, joka johtuu tulkattavan esityksen muodostamisesta ja epäsuorasta tulkitsevasta suorituksesta. Lisäksi suoritettavan olioesityksen tilavaatimus voi olla paljon suurempi kuin vastaavan merkkijonoesityksen tai konekoodin.

Edut ja ongelmat

Edut

- Ajoaikainen suoritusympäristö omassa hallinnassa
- Tulkittavaa kieltä suhteellisen helppo muuttaa
- Kielen merkitystä suhteellisen helppo muuttaa
- Suoritusympäristö voidaan helposti vaihtaa

Ongelmat

- Suorituskyky (epäsuora, ei-natiivi suoritus; sisäisen esityksen muodostaminen)
- Tilankäyttö (ohjelman sisäinen esitys voi vaatia paljon tilaa)

Yhteenveto luvusta

- Arkkitehtuurityylejä käyttämällä on mahdollista perustaa suunnittelu koeteltuihin ja hyväksi havaittuihin ratkaisuihin arkkitehtuuritasolla.

- Arkkitehtuurityylit esiintyvät harvoin puhtaina, vaan samassa järjestelmässä voidaan käyttää useita tyylejä eri tarkoituksiin.

- Uusia ominaisuuksia varten ei kuitenkaan automaattisesti kannata valita uutta tyyliä, vaan mukailla järjestelmään jo toteutettuja suunnitteluratkaisuja, sillä tämä helpottaa ylläpitoa jatkossa.

6. TUOTERUNKOARKKITEHTUURIT

Ohjelmistotuotannossa on tärkeää pystyä uudelleenkäyttämään tehtyjen ohjelmistojen osia muissakin ohjelmistoissa. Kun ohjelmistot suunnitellaan alusta lähtien uudelleenkäytettäväksi, se nopeuttaa tulevien saman sovellusalueen ohjelmistojen toteutusta. Tuoterunkoarkkitehtuurit toimivat jonkin tietyn sovellusalueen tuotteiden pohjana, jonka päälle halutut tuotteet tehdään.

Arkkitehtuurin rooli ohjelmistokehityksessä

Arkkitehtuurin voidaan nähdä toimivan kolmessa erilaisessa roolissa ohjelmistokehitystä ajatellen:

- Arkkitehtuuri selittää järjestelmää
- Arkkitehtuuri ohjaa järjestelmän rakentamista
- Arkkitehtuuri mahdollistaa järjestelmiä

Perinteisesti arkkitehtuuri selittää ohjelmiston rakennetta ja luonnetta. Tässä tapauksessa arkkitehtuurindokumentointi saatetaan valitettavasti hoitaa vasta, kun ohjelmisto on jo tehty.

Merkittävämpi rooli arkkitehtuurilla on, kun sitä ajatellaan järjestelmän rakentamista ohjaavana artifaktina. Tällöin arkkitehtuuridokumentointi tehdään jo varhaisessa vaiheessa ja sitä noudatetaan tarkasti. Tälläisen arkkitehtuurin täytyy myöskin olla riittävän tarkka ja kattava, jotta se mahdollistaa yksityiskohtaisemman suunnittelun ja toteutuksen ja sen on tuettava myös ylläpitoa. Ohjaavaa arkkitehtuuria voidaan käyttää myös selittävänä arkkitehtuurina mutta ei toisin päin. Ohjaava arkkitehtuuri myöskin johtaa arkkitehtuuripainoitteiseen ohjelmistokehitysprosessiin (luku 1.4).

Nykyisin ohjelmistotuotteet tehdään useammin ja useammin olemassa olevien ohjelmistojen pohjalle. Tämä johtuu siitä, että yrityksillä on kokoajan suuremmat paineet tehdä nopeammin laadukasta tuotekehitystä. Tällöin tarvitaan uudellenkäytettävää ohjelmistoalustaa, jolla on hyvin määritelty arkkitehtuuri, joka mahdollistaa korkean tason käsitteet ja mekanismit, joiden avulle se toteuttaa vaatimukset mahdollisimman helposti. Tällöin arkkitehtuurin rooli on mahdollistaa järjestelmien tekoa. Mahdollistavaa arkkitehtuuria voidaan käyttää yksittäisen tuotteen kohdalla ohjaavana ja selittävänä arkkitehtuurina, mutta ei toisinpäin.

Tuoterunkoarkkitehtuureihin liittyvät käsitteet

- **Tuoteperhe** on joukko koordinoidusti kehitettyjä ohjelmistotuotteita, joilla on samankaltainen rakenne ja toiminta.
- **Tuotelinjaan** kuuluvat kaikki artifaktit, välineet ja prosessit, jotka tukevat tuoteperheen jäsenten kehittämistä ja ylläpitoa.
- **Tuoterunko** ja **tuotealusta** tarkoittavat tuoteperheen yhteistä ohjelmistoalustaa.
- **Tuoterunkoarkkitehtuuri** ja **tuoteperhearkkitehtuuri** tarkoittavat tuoteperheen yhteistä arkkitehtuuria.

Tuoterungon käyttöönotto

Hyödyt

- Taloudelliset näkökohdat
- Uudelleenkäytön mahdollistuminen
- Markkinoille pääsyn nopeutuminen
- Laadun paraneminen

Edellytykset

- Tuoteperhe tunnistettavissa: yhteisiä piirteitä ja tunnistettu muunneltavuuden tarve

Ongelmia

- Ratkaisuja jäykistävä, ohjaileva
- Innovatiivisten (tuotekohtaisten) ratkaisuiden estyminen
- Investointi on riski – hyödyt myöhemmin; muutokset ympäristössä

Lähteenä: (TaTY -kalvot, 2006)

Tuoterungon rakentaminen ja evoluutio

Tuoterunkoa hyödyntävään ohjelmistokehitykseen voidaan siirtyä monella tavalla. Yleisesti ottaen ainakin seuraavat tavat ovat mahdollisia:

- Olemassa olevan toteutuksen komponentit otetaan tuoterungon perustaksi, ja niiden varaan toteutetut tuotteet tuoteperheen jäseniksi.
- Uusi tuoterunkoon perustuva tuotelinja voidaan rakentaa inkrementaalisti, eli ensimmäisessä versiossa on vain muutamia ominaisuuksia, mutta kehittyy kunnolliseksi tuoterungoksi.
- Uusia tuotteita varten toteutettu tuoterunko rakennetaan valmiiksi, jolloin sen varaa rakennetut tuotteet voivat hyödyntää sitä maksimaalisesti.
- Ensimmäinen uudentyyppinen tuote otetaan perustaksi, joka muokataan tuoterungoksi tekemällä siitä paremmin muunneltava.

Tuoterungon evoluution voi liittyä erityispiirteitä, kuten:

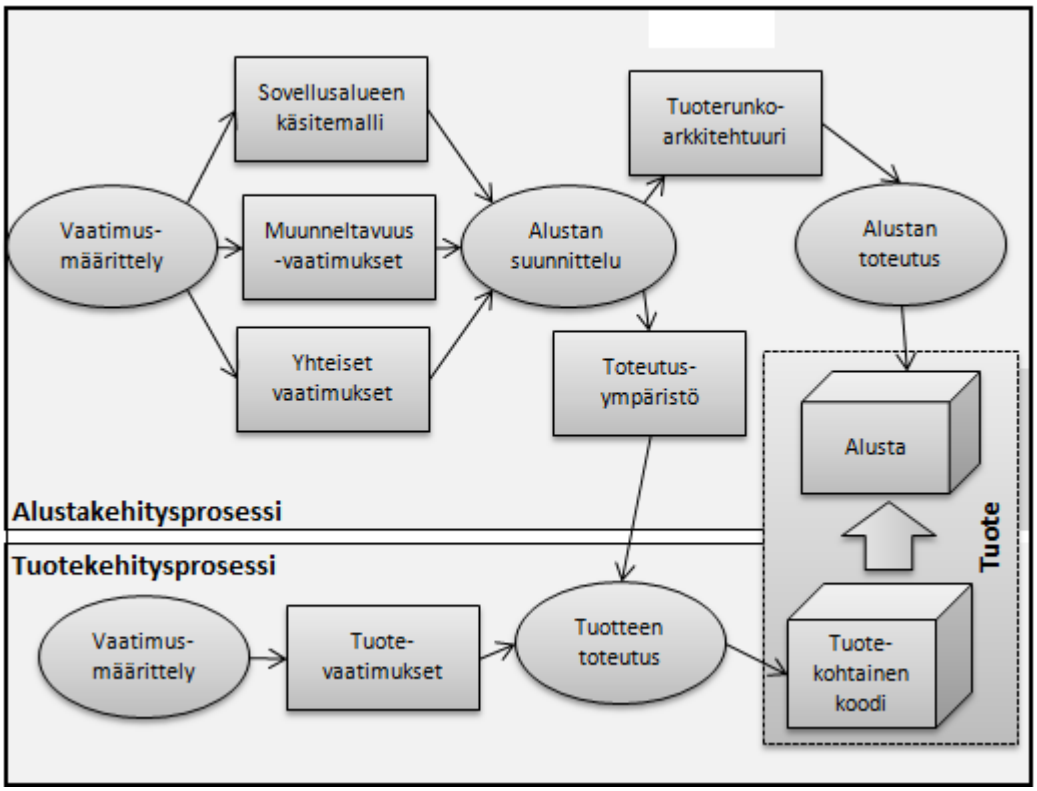
- Tuoterunko voi haarautua, jolloin syntyy erityyppisiä tuotteita palvelevia kokonaisuuksia.
- Uusien tuotteiden toteuttaminen tuoterunkoon perustuen on välttämätöntä jatkohyödyntämisen kannalta.
- Tuoterunkoon saatetaan lisätä uusia ominaisuuksia. Usein jotain yksittäistä tuotetta varten tehdyt ominaisuudet siirretään tuoterunkoon.
- Laadullisten ominaisuuksien parantaminen mahdollistaa tuoterungon käytön uudenlaisiin tuotteisiin.

Tuoterunko pohjainen ohjelmistonkehitysprosessi

Jotta tuoterunkojen käytöstä saataisi kaikki edut irti ohjelmiston kehittämisen ja ylläpidon suhteen, täytyy tekniikan lisäksi ajatella toiminnan organisointia ja käytettyjä prosesseja.

6.1.1 Prosessin yleiskuvaus

Tuoterunkoon pohjautuva ohjelmistonkehitysprosessi jakautuu kahteen osaan (Kuva 6.1), alustakehitysprosessiin, jossa tuloksena tuoterunko, ja tuotekehitysprosessiin, jossa tuloksena on tuoterunkoon pohjautuva yksittäinen tuote. Näitä edeltää esitutkimusvaihe, jossa selvitetään onko mitään järkeä kehittää tuoterunkoa. Tähän asiaan vaikuttaa suuresti arvioitu tuoteperheen jäsenten määrä. Jos tullaan siihen tulokseen, että tuoterungon päälle ei tulisi tehtyä montaakaan eri tuotetta, niin on parasta tehdä tuotteet jokainen erikseen.



Kuva 6.1 Tuoterungon ohjelmistokehitysprosessi (Koskimies, Mikkonen, 2005)

6.1.2 Alustakehitysprosessi

Tuoterungon kehitys aloitetaan tekemällä vaatimusmäärittely, josta saadaan sovellusalueen käsitemalli, muunneltavuusvaatimukset sekä yhteiset vaatimukset.

Sovellusalueen käsitemalli määrittelee sen toiminta-alueen käsitteet, jolle tuotteet on tarkoitettu. Esimerkiksi jos sovellusalueena on pankkitoiminta, niin sovellusalueen käsitemalli määrittelee pankkitoiminnan yleiset käsitteet ja niiden väliset suhteet. Käsitemallilla voidaan myös varmistaa, että kaikki osapuolet ymmärtävät sovellusalueen samalla tavalla ja tarjoaa sanaston, jonka avulla vaatimuksia voidaan esittää tuoterungon suhteen.

Muunneltavuusvaatimukset määrittelevät mitkä ominaisuudet voivat vaihdella tuotteissa.

Yhteisissä vaatimuksissa määritellään kaikille tuotteille yhteiset vaatimukset.

Vaatimusmäärittelyn tulosten jälkeen voidaan aloittaa tuoterungon arkkitehtuurin suunnittelu. Lähtökohdiksi kelpaavat esimerkiksi esitutkimuksessa vaiheessa ajatellut arkkitehtuurityylit tai sovellusalueen käsitemalli perinteisen oliosuunnittelun mukaisesti tai ihan vaan joku puhtaasti tuotteiden yhteisiin vaatimuksiin perustuva alustava ratkaisu. Lähtökohdasta riippumatta tuoterunkoarkkitehtuurin suunnittelu on iteratiivista toimintaa, jossa muunneltavuus on keskeinen laatutavoite ja siinä voidaan soveltaa arkkitehtuuripainotteista ohjelmistokehitysprosessia (Luku 1.4).

Kun vaatimukset täyttävä arkkitehtuuri on toteutettu, se voidaan vielä testata käyttäen arkkitehtuurin arviointimenetelmiä (Luku 8).

Rinnan tuoterunkoarkkitehtuurin kanssa täytyy suunnitella se toteutusympäristö, jonka avulla kehitetään uusia tuotteita. Pelkkä tuoterunkoarkkitehtuuri järjestelmän rakenteen kuvauksena ja sen toteuttava ohjelmistoalusta ei yleensä riitä, koska tuotteiden kehittäjille halutaan tarjota

helposti ymmärrettävä, selkeä toteutusvälineistö. Mitä paremmin kehittäjä hallitsee tämän valineistön, sen helpompi hänen on esittää vaatimukset yksittäisten tuotteiden suhteen tuoterungon avulla. Yksinkertaisimmillaan tälläinen toteutusympäristö voi olla hyvin määritellyistä rajapinnoista koostuva API, joka piilottaa tuoterunkoarkkitehtuurin ja ohjelmistoalustan toteutuksen tuotteiden kehittäjiltä. Usein pitää kuitenkin paljastaa jotain tuoterunkoarkkitehtuurista tuotteiden kehittäjille, jotta haluttua muunneltavuutta saadaan aikaan.

6.1.3 Tuotekehitysprosessi

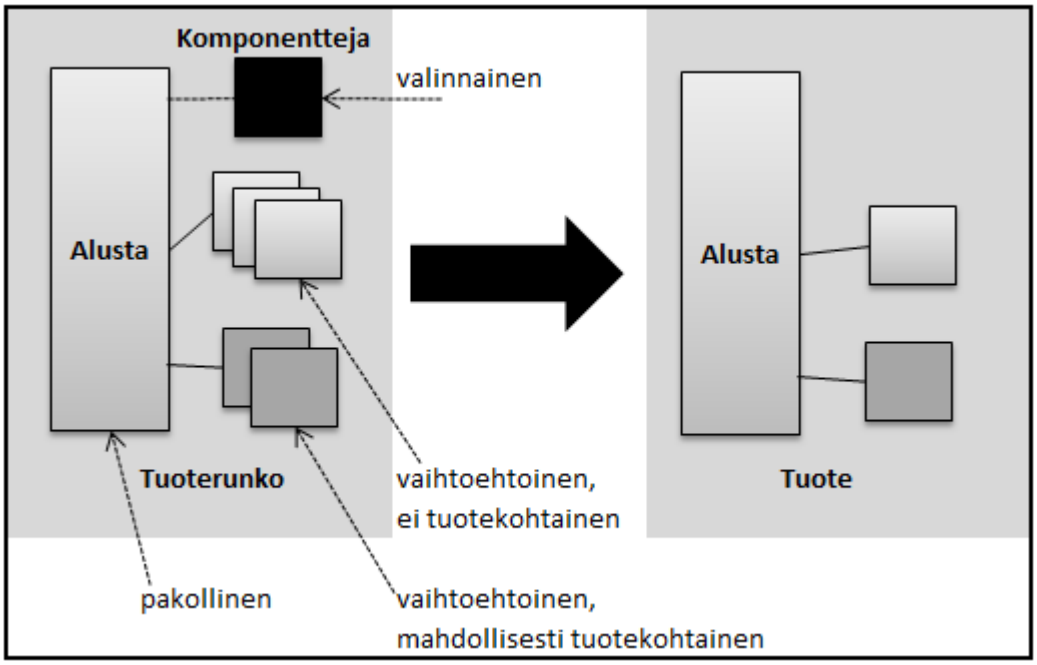
Tuoteprojekti alkaa kuten yleensä, määritellään tuotteen vaatimukset analyyseillä ja selvitetään asiakkaalta mitä tarpeita heillä on tuotteen suhteen. Tuoterunkoja käyttäessä on olennaista kertoa asiakkaalle mitä on helppo toteuttaa rungon päälle, mitä vaikea toteuttaa, ja mitä ei pystytä toteuttamaan.

Tuoterungosta ja toteutusympäristöstä riippuen tuotteen suunnittelu ja toteutus voi olla hyvin erilainen tilanne. Jos ympäristö tarjoaa esimerkiksi oman erikoiskielen tuotteen kuvaamiseen, vaatimukset täyttävä tuote voidaan kuvata suoraan tällä kielellä ja generoida automaattisesti koodi tuotetta varten. Jos tuoterunko koostuu joukosta komponentteja, joiden rajapinnat mouodostavat toteutusympäristön, tuote saattaa vaatia oman tuotekohtaisen arkkitehtuurinsa, joka pitää suunnitella erikseen.

Tyypillisesti tuoterungot ovat näiden ääripäiden välillä. Jos tuotteiden varianssi on pientä, ne tarvitsevat vain vähän tai ei ollenkaan tuotekohtaista arkkitehtuuria, jolloin vaatimukset voidaan toteuttaa lähes suoraan tuoterungon päälle. Jos varianssi on suuri, tuoterungon arkkitehtuuri tarjoaa vain perusrakenteen, ja tuotteelle täytyy suunnitella omaa arkkitehtuuria.

Muunneltavuus tuoterungossa

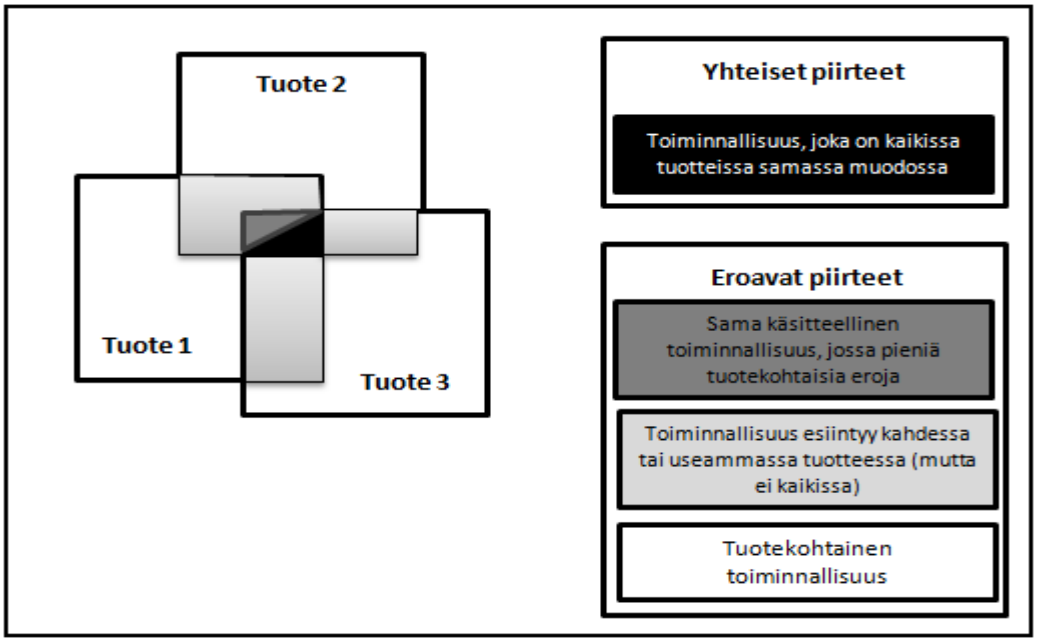
Muunneltavuuden hallinta on ongelma tuoterunkoihin perustuvassa ohjelmistokehityksessä. Tuoterunkoihin kuuluvat tyypillisesti ohjelmistoalusta, joka sisältyy kaikkiin tuotteisiin ja komponentteja joista jotkut ovat valinnaisia ja jotkut vaihtoehtoisia. Tuotetta varten voidaan tehdä myös tuotekohtaisia komponentteja. Tilannetta on havainnoillistettu kuvassa 6.2. Tässä luvussa käydään läpi muunneltavuuden hallintaan liittyviä näkökulmia prosessin eri vaiheissa.



Kuva 6.2 Tuoterunko ja yksittäinen tuote (Koskimies, Mikkonen, 2005)

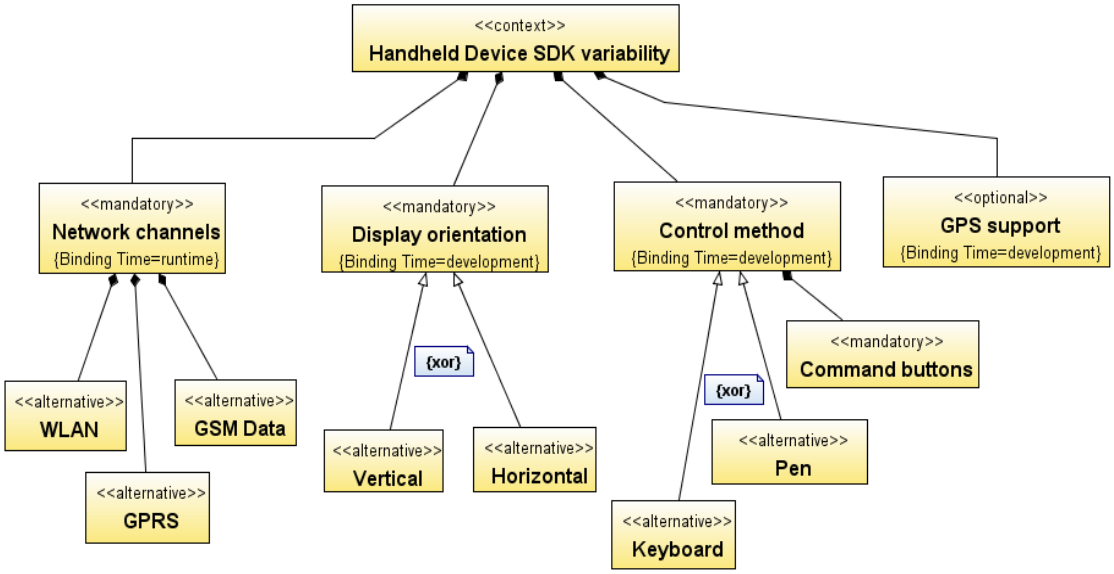
6.1.4 Muunneltavuus vaatimustasolla

Kuten muutkin vaatimukset, myös varianssi rajataan vaatimusmäärittelyssä. Tuotteiden yhteisiä ja eroavia piirteitä on havainnoillistettu kuvassa 6.3.



Kuva 6.3 Ohjelmistotuotteiden yhdistävät ja erottavat piirteet (Koskimies, Mikkonen, 2005)

Täsmällisesti tuotteiden yhteisiä ja eroavia ominaisuuksia voidaan esittää nk. piirremallin avulla (Kuva 6.4). Piirremallissa kuvataan mitkä ominaisuudet ovat tuotteelle pakollisia, mitä valinnaisia ja vaihtoehtoisia ominaisuuksia tuotteella on ja milloin nämä ominaisuudet kiinnitetään tuotteeseen. Kuvan 6.4 PDA-laitteen piirremallista voi huomata, että esimerkiksi ohjauslaitteena voi olla kynä tai näppäimistö mutta ei molempia (xor-rajoite) ja ominaisuus kiinnitetään tuotteeseen kehitysaikana(Binding Time = development).



Kuva 6.4 Muunneltavuuden kuvaus piirremallina käyttäen laajennettua UML-luokkakaavioesitystä (Koskimies, Mikkonen, 2005)

6.1.5 Muunneltavuus suunnittelussa ja toteutuksessa

Piirrekartoituksen jälkeen tiedetään millä laajuudella tuoterunkoarkkitehtuurin pitää tukea erilaisia tuotemuunnelmia. Jokaisen muunneltavuusvaatimuksen kohdalla on päätettävä tapa, jolla tätä muunneltavuutta tuetaan tuoterungossa. Muunneltava ominaisuus on eristettävä niin, että se voidaan poistaa (valinnainen ominaisuus) tai korvata (vaihtoehtoinen ominaisuus) jollakin muunnelmalla. Jos tämä muunnelma kiinnitetään vasta ajoaikana, täytyy tuoterungon omata mekanismi, jolla tuote voidaan konfiguroida ajoaikana. Vaikka tuoterunkoarkkitehtuurin rooli korostuu muunneltavuuden mahdollistajana, kaikkea muunneltavuutta ei oteta vielä huomioon arkkitehtuuritasolla, koska jotkut muunneltavuusvaatimukset on luontevampaa huomioida vasta yksityiskohtaisessa suunnittelussa tai toteutusvaiheessa.

Suunnittelutason muunneltavuutta tukeville ratkaisuille ei ole mitään yleisiä ohjeita, yleisesti käytetään esimerkiksi komponenttiteknologiaa (esim. rajapintojen erilaiset toteutukset), ohjelmointikielen geneerisiä ominaisuuksia (esim. C++:n template), arkkitehtuurityylien muunneltavuutta tukevia

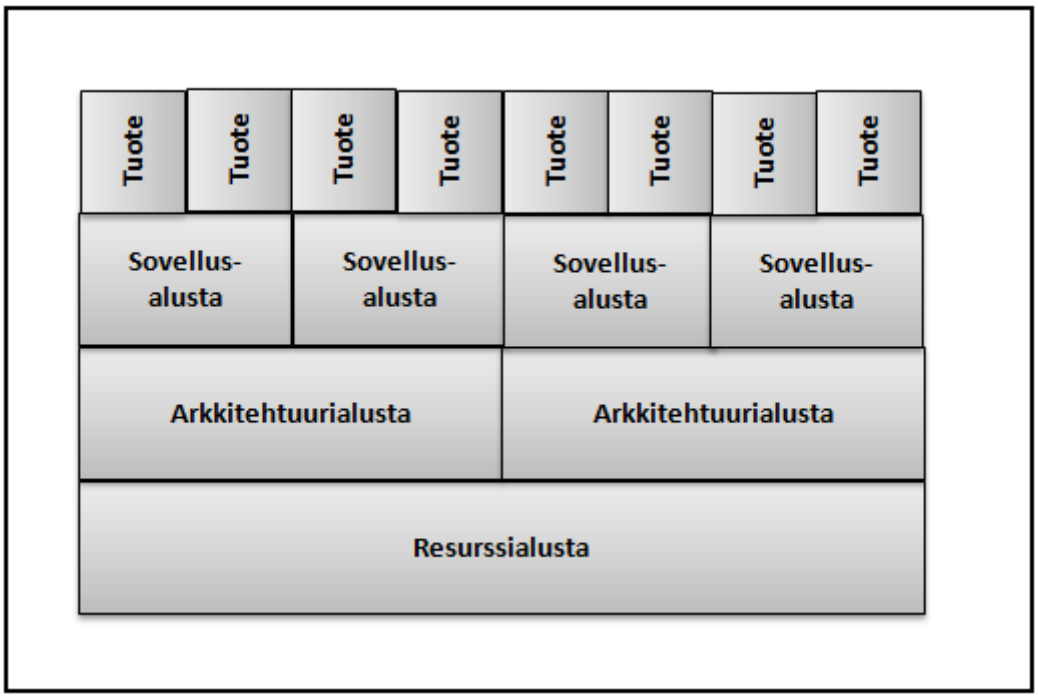
ominaisuuksia ja suunnittelumalleja. On tärkeää pystyä liittämään suunnittelussa käytetyt ratkaisut muunneltavuusvaatimuksiin. Tälläistä muunneltavuusvaatimuksen ja sitä tukevaa suunnitteluratkaisun kokonaisuutta kutsutaan tuoterungossa variaatiopisteeksi.

6.1.6 Muunneltavuus testauksessa

Tuoterunkojen heikkoutena on testattavuus. Variaatiopisteiden avulla on helppo määritellä ja toteuttaa uusia konfiguraatioita, mutta nämä kaikki täytyy testata kuin yksittäin tehdyt sovellukset. Syynä tälläiseen on se, että tuoterunko itsessää ei välttämättä ole suoritettavissa, vaan se voi sisältää toteutusta vaativia rajapintoja. Vaikka tuoterunko olisi suoritettavissa, on vaikeaa määritellä testitapaukset, joilla tuoterunko voitaisiin testata omana kokonaisuutena. Näin tuoterungon muunneltavuus ja sen edut eivät suoraan siirry testausvaiheeseen. Jokainen tuoterungon päälle kasattu ohjelmisto joudutaan testaamaan erikseen kokonaisuudessaan.

Tuoterunkoarkkitehtuurin kerrosmalli

Tuoterunkojen ongelmana on muunneltavuuden hallinta. Muunteluun liittyvät huolet voidaan eriyttää jakamalla tuoterunkoarkkitehtuuri neljään kerrokseen kerrosarkkitehtuurin idean mukaisesti (Kuva 6.5). Kerrokset ovat nimeltään resurssialusta, arkkitehtuurialusta, sovellusalusta ja tuotekerros, joista kaikki tarjoavat eri osa-alueisiin varianssia.



Kuva 6.5 Tuoterunkoarkkitehtuuri kerrosmallina (TuTY-kalvot 2008)

6.1.7 Resurssialusta

Alin kerros, resurssialusta, tarjoaa useimmiten API-rajapinnan, jonka kautta järjestelmä voi käyttää ympäristön yleisiä peruspalveluita ja resursseja. Järjestelmästä riippuen, nämä palvelut voivat olla esimerkiksi kommunikkaation liittyviä hajautetuissa järjestelmissä, prosessien hallintaan liittyviä rinnakkaisissa järjestelmissä tai tiedon talletuksiin liittyviä tietokannoissa.

Resurssialustan palvelut eivät riipu järjestelmän arkkitehtuurista eikä sovellusalueesta eikä palvelujen kutsujan tarvitse tietää resurssialustan arkkitehtuurista. Resurssialusta kiinnittää järjestelmät tiettyyn ulkoiseen (esim. laite-, verkko-, grafiikka- tms) ympäristöön.

Esimerkkinä resurssialustasta voisi olla vaikkapa CORBA-toteutus, tietokantapalvelut abstrahoiva kerros, sulautetun järjestelmän käyttöjärjestelmä tai graafisen ympäristön primitiivipalvelut.

6.1.8 Arkkitehtuurialusta

Tässä kerroksessa määritellään järjestelmien yleinen arkkitehtuurityyli ja tarjotaan kiinnityskohdat sovellusaluekohtaisille kerroksille. Tässä kerroksessa ei oteta vielä kantaa sovellusalueeseen. Arkkitehtuurialustan suunnitteluun vaikuttavat lähinnä siihen pohjautuvien sovellusten ei-toiminnalliset vaatimukset (esim. hajautettavuus, muunneltavuus, suorituskyky).

Esimerkkinä arkkitehtuurialustasta voisi olla vaikka graafinen käyttöliittymäalusta (Esim. MVC)

6.1.9 Sovellusalusta

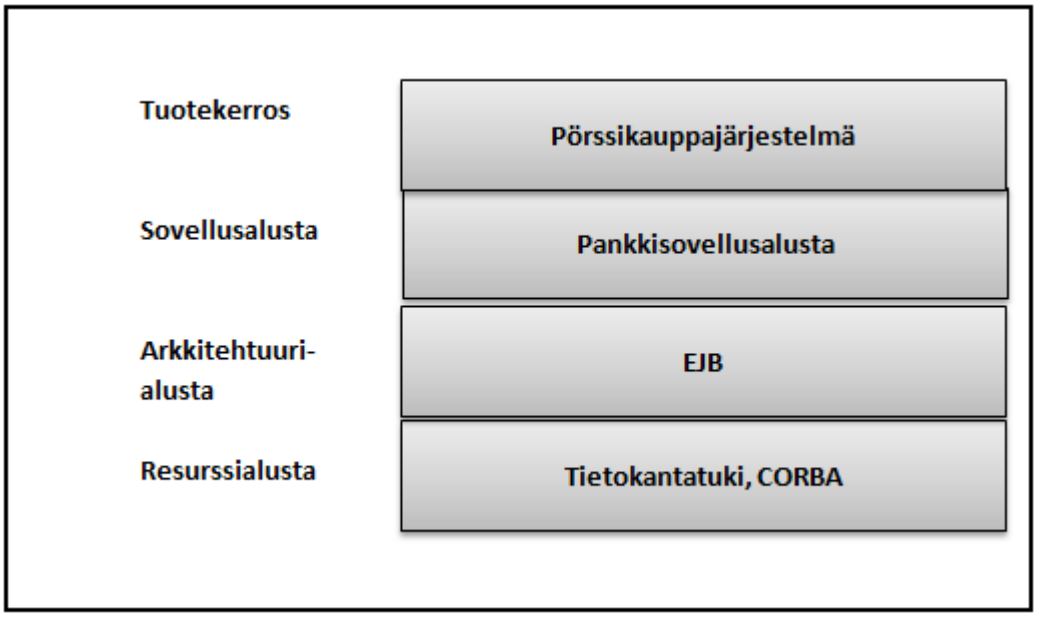
Tämä kerros rakentuu arkkitehtuurialustan määrittelemälle yleiselle arkkitehtuurimallille. Sovellusalusta toteuttaa sovellusten rungon jollekin tietylle sovellusalueelle. Tämä runko voi sisältää sovellusaluekohtaisia arkkitehtuuriratkaisuja, jotka kuitenkin pohjautuvat alemman kerroksen yleiseen tyyliin. Sovellusalusta tarjoaa erikoistamisrajapinnan, jonka avulla toteutetaan sovelluksen toiminnallisia vaatimuksia.

Sovellusalustan suunnitteluun vaikuttavat sovellusten laadulliset sekä toiminnalliset vaatimukset. Eräs toteutusmuoto sovellusalustalle on sovelluskehys ja sitä täydentävät komponentit. Kukin näistä komponenteista sisältyy vähintään kahteen eri sovellukseen.

Esimerkkinä sovellusalustasta on EJB:n pohjautuva vakuutusjärjestelmäkehys, Swing:iin perustuva verkonhallintasovelluskehys tai Symbian-ympäristöön toteutettu mobiilisovelluskehys, jonka avulla voidaan sama sovellus toteuttaa eri mobiililaitteille.

6.1.10 Sovelluskerros

Tässä kerroksessa toteutetaan lopulta sovelluskohtaiset toiminnalliset vaatimukset. Tämän kerroksen osat kirjoitetaan vasten sovellusalustan tarjoamaa erikoistamisrajapintaa.



Kuva 6.6 Nelikerrosmalliin perustuva esimerkkisovellus. (Koskimies, Mikkonen, 2005)

Tuoterunkoarkkitehtuurien hyödyt ja ongelmat

6.1.11 Tuoterungon edut

Tuoterunkoa hyödynnättäessä toteutuksen perusrakenteita voidaan uudelleenkäyttää kaikissa tuotteissa, mikä puolestaan

- parantaa laatua, sillä käytettävä koodi on testattu useassa aikaisemmassa tuotteessa useassa eri konfiguraatiossa,
- nopeuttaa ohjelmistokehitystä, sillä suurin osa koodista on jo olemassa uutta tuotetta kehitettäessä,
- helpottaa projektin hallintaa, sillä tuoteprojekteissa voidaan käyttää samankaltaisia prosessimalleja,

- helpottaa siirtymistä projektista toiseen, sillä ympäristö ja käytettävät työkalut ovat jo tuttuja,
- standardoi tuotteita, sillä yhteiset asiat toimivat samalla tavalla eri tuotteissa (esim. käyttöliittymässä),
- tehostaa toimintaa, sillä paljon panostusta vaativa arkkitehtuurisuunnittelu on jo etukäteen ainakin suureksi osaksi tehty.

Lähteenä: (TaTY, 2006)

6.1.12 Tuoterunkojen ongelmia

- Henkilöstöön liittyvät ongelmat:
  - vaihtuvuus heikentää sisäistämistä
  - motivoituneisuus
  - avainosaajien merkitys
  - konfliktit rungon ja tuotteen kehittäjien välillä
- Yksittäisen tuotteen ylivertaiset ominaisuus vs. rungon rajoitteet
- Ensimmäisen tuotteen julkistamisen viivästyminen
- Huojuminen: tuoterungon virittyminen yksittäisten tuotesukupolvien syntymisen kautta
- Byrokratian lisääntyminen: johtaminen, kommunikointi, koordinointi
- Version- ja konfiguraationhallinnan monimutkaistuminen
- Testauksen ongelmat

Lähteenä: (TaTY, 2006)

Yhteenveto luvusta

- Tuoterunkojen käyttö on paljon sovellettua tekniikkaa, jossa arkkitehtuurin avulla pyritään systemaattiseen koodin uudelleenkäyttöön useissa ohjelmistotuotteissa.
- Tuoterungon kannattavuus riippuu yhtäältä sen varaan toteutettavien tuotteiden määrästä ja toisaalta tuotekohtaisen räätälöinnin määrästä.

- Tuoterunkojen ydinkysymys on muunneltavuuden hallinta: miten suunnitella arkkitehtuuri niin, että se sallii tuotteille halutun muunneltavuuden, ja miten tukea tuotteen toteuttajaa tämän muunneltavuuden hyödyntämisessä.

- Tuoterunkoon perustuvassa lähestymistavassa tuoterunkoa hallitaan kokonaisuutena, ja tätä kokonaisuutta arvostetaan enemmän kuin sen yksittäisiä ilmentymiä tuotteissa.

- Tuoterunkoarkkitehtuurin nelikerrosmalli sisältää resurssialustan, arkkitehtuurialustan, sovellusalustan ja tuotekohtaisen kerroksen.

- Tuoterungon käyttö vaikuttaa usein myös organisaatiorakenteeseen ja ohjelmistotyössä käytettyihin toimintatapoihin.

7. OHJELMISTOKEHYKSET

Ohjelmistokehys on ohjelmistorunko, jota voidaan täydentää eri tavoin eri tarkoituksia varten. Sitä ei ole siis tehty valmiiksi, vaan siihen on jätetty aukkoja, joita täydennetään halutulla tavalla. Kehykset ovat suosittu tapa tuoterunkoarkkitehtuurien toteuttamiseen varsinkin oliomaailmassa. Sen tarkoitus on sama kuin tuoterunkoarkkitehtuureilla, uudelleenkäytettävyyden parantaminen.

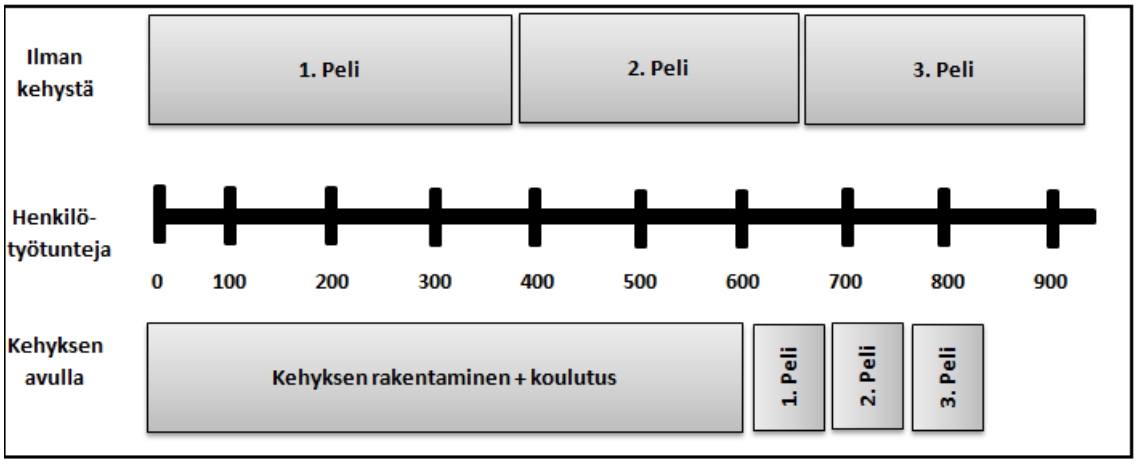
Yleistä ohjelmistokehyksistä

Ohjelmistokehys on luokka-, komponentti- ja/tai rajapintakokoelma, joka toteuttaa jonkin ohjelmistojoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden. Tälläinen ohjelmistojoukko voisi olla esimerkiksi jollekkin tietylle sovellusalueelle tarkoitettu samantyyppisten sovellusten joukko (Esimerkiksi simulointikehykset), tietyn perusrakenteen omaavien sovellusten joukko (Esimerkiksi hajautetut liiketoimintasovellukset), tietyntyyppisen graafisen käyttöliittymän omaavien sovellusten joukko tai jonkin komponentin eri variaatioiden joukko. (Koskimies, Mikkonen, 2005)

Kehys on siis ohjelmistorunko, jossa on aukkoja. Haluttu ohjelmisto saadaan kehyksen avulla, kun täytetään nämä aukot uudella koodilla, joka toteuttaa kyseisen ohjelmiston erityistoiminnallisuuden muuttamalla kehyksen arkkitehtuuria. Tätä sanotaan kehyksen erikoistamiseksi. Jos erikoistamisen tuloksena on uusi itsenäinen sovellus, sitä sanotaan sovelluskehykseksi. Jos taas erikoistamisen tulos on uusi komponentti, sitä sanotaan komponenttikehykseksi.

Kuvassa 7.1 on verrattu tarvittavia työtunteja tietokonepelejä tehdessä, kun pelit tehdään ilman kehystä ja kehyksen kanssa. Kuvasta näkyy, että vaikka kehyksen rakentamiseen ja sen käytön opetteluun kuluu rutkasti aikaa, niin

pelien tekeminen sen jälkeen on huomattavasti nopeampaa kuin yksittäin tehtynä (Vaikka yksittäin tehtynäkin pelien valmistuminen hiukan nopeutuu, koska toteuttajat tietävät ensimmäisen pelin jälkeen paremmin kuinka tehdä se.).

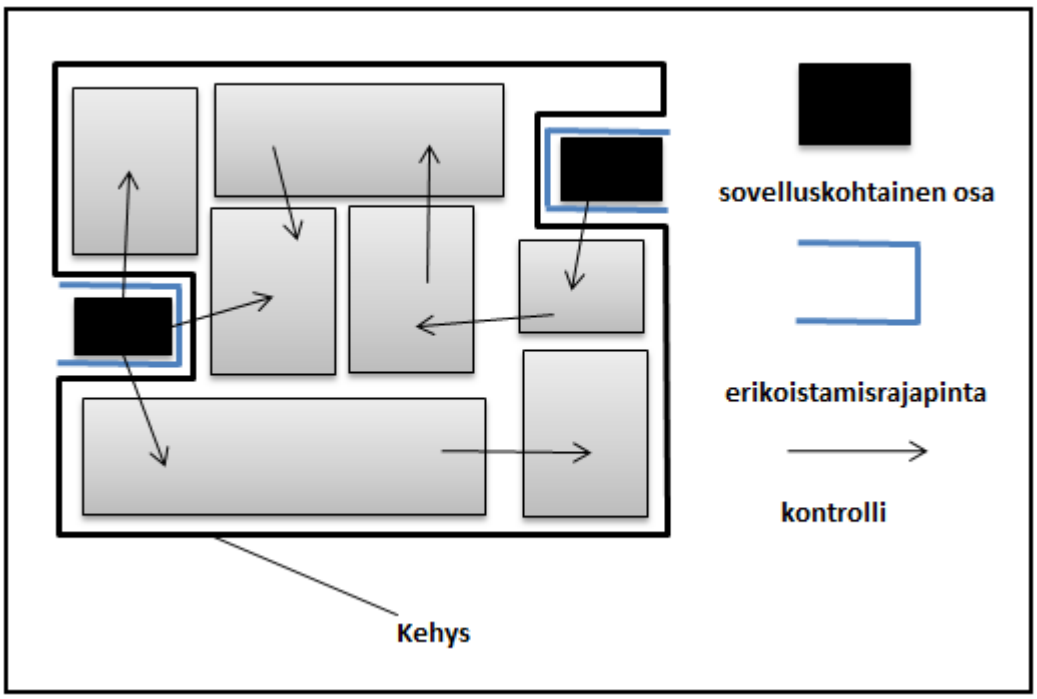


Kuva 7.1 Työtuntimäärien vertailu pelisovellusten tapauksessa, kehyksellä ja ilman. (Koskimies, Mikkonen, 2005)

Kehystekniikka on sovellettu menestykkäästi monilla sovellusalueilla. Tunnetuimpina ehkä graafisten käyttöliittymien rakentamiseen tarkoitetut kehykset kuten Javan Swing. Myöskin pelisovelluksissa, pankkisovelluksissa, vakuutussovelluksissa, verkonhallintasovelluksissa ja erilaisissa graafisissa editoreissa kehysten käyttö on ollut menestyksellistä.

7.1.1 Erikoistamisrajapinta

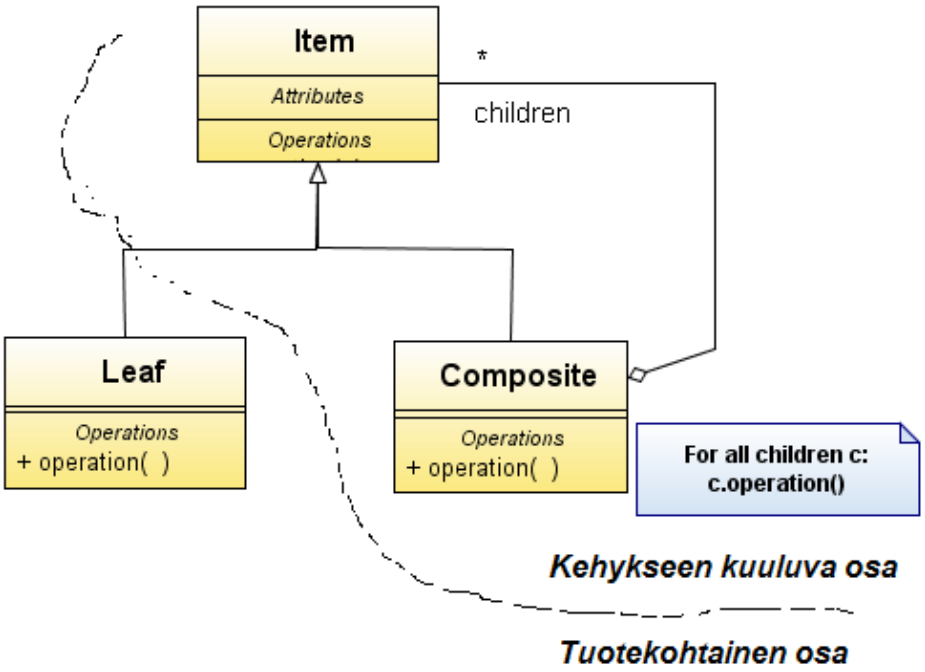
Kehyksen aukkoja voidaan kutsua myös laajennoskohdiksi (hot spots). Kehystä tehdessä, nämä kohdat täytetään uudella tuotekohtaisella koodilla (Kuva 7.2). Kutsut voivat kulkea molempiin suuntiin kehyksen ja uusien osien välillä. Kehys määrää mitä vaatimuksia tuotekohtaisen koodin pitää täyttää missäkin aukossa. Kehyksen laajennoskohtia ja niihin liittyviä vaatimuksia tuotekohtaiselle koodille sanotaan erikoistamisrajapinnaksi.



Kuva 7.2 Ohjelmistokehys

7.1.2 Kehykset ja suunnittelumallit

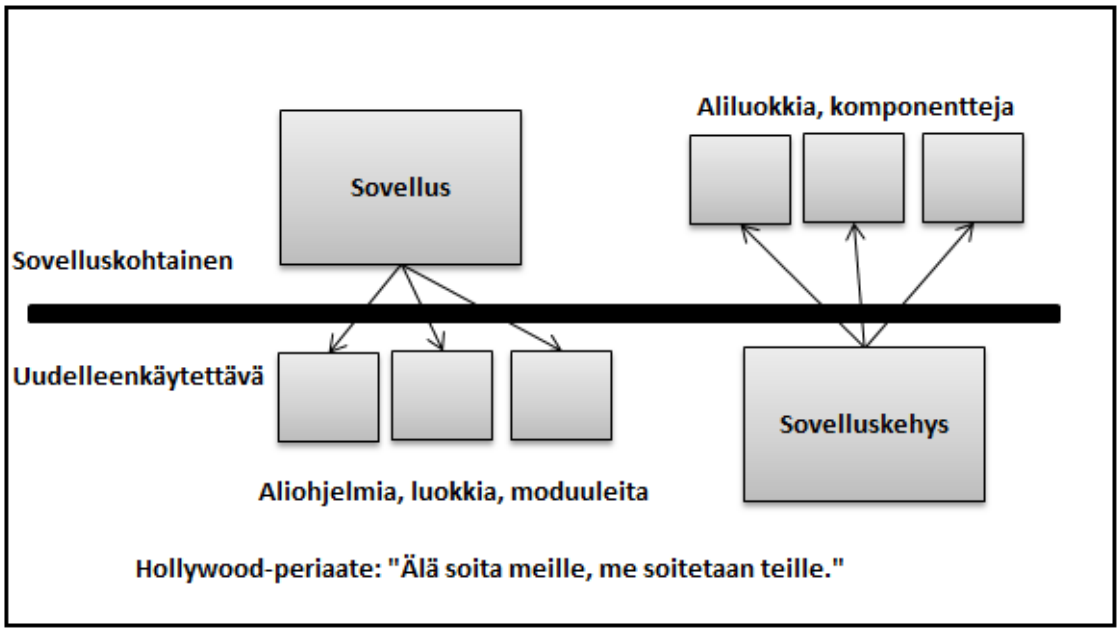
Kehysten arkkitehtuureissa suunnittelumalleilla on keskeinen asema. Suunnittelumallien tavoitteena on parantaa järjestelmän joustavuutta ja muunneltavuutta johon pyritään myös kehysten arkkitehtuurissa. Kehysten arkkitehtuuri voidaan usein pitkälti kuvata esittämällä mitä suunnittelumalleja on käytetty ja miksi. Kun suunnittelumallilla pyritään muunneltavuuteen, on luonnollista että sitä sovelletaan kehyksen ja tuotekohdaisen koodin rajapinnalla. Yleisesti ottaen suunnittelumalleissa on nähtävissä yleinen, kehykseen kuuluva osa ja vaihtuva tuotekohtainen osa. Tätä on havainnoillistettu kuvassa 7.3. Kuvassa on rekursiokooste-suunnittelumalli, jonka puurakenne sisältyy kehykseen ja lehtioliot erikoistetaan tuotekohtaisesti.



Kuva 7.3 Rekursiokooste-suunnittelumalli kehyksen erikoistamisrajapinnan osana (Koskimies, Mikkonen, 2005)

7.1.3 Hollywood-periaate

Perinteisessä kirjaston uudelleenkäytössä sovelluksella on toiminnan pääkontrolli ja se kutsuu tarvittaessa tietyissä kohdissa yleisiä kirjastopalveluja. Kehysten tapauksessa tilanne on päinvastoin. Mahdollisen sovelluskohtaisen alustuksen jälkeen kontrolli siirtyy kehykselle, joka pitää sen koko toiminnan ajan, mutta voi kuitenkin kutsua sovelluskohtaista koodia takaisinkutsujen avulla. Perinteissä kirjastossa sovellus kutsuu uudellenkäytettävää koodia ja kehyksen tapauksessa uudelleenkäytettävä kutsuu sovelluskohtaista koodia. Tätä sanotaan Hollywood-periaatteeksi (”don’t call us, we’ll call you”). Asiaa on havainnollistettu kuvassa 7.4.



Kuva 7.4 Hollywood-periaate (TuTY- kalvot 2008)

7.1.4 Erikoistamismekanismit

Kehyksen erikoistamisrajapinnassa voidaan käyttää erilaisia mekanismeja tuotekohtaisen koodin sitomiseen kehyksen koodiin. Tälläisiä mekanismeja ovat

- periyttäminen
- rajapintojen toteuttaminen
- parametrointimekanismit
- geneeristen ohjelmarakenteiden käyttö
- refleksiivisyysominaisuuksien käyttö
- sovelluksen tarjoama muuntelu

Periyttämisessä kehys tarjoaa kantaluokan, jonka tuotekohtainen luokka perii ja antaa joillekin sen operaatioille uuden toteutuksen. Tehdasta käyttämällä kehys voi luoda tuotekohtaisen aliluokan ilmentymiä tuntematta tätä luokkaa. Kun kehys kutsuu näiden olioiden operaatioita, kontrolli siirtyy dynaamisen sidonnan kautta tuotekohtaiseen operaation toteutukseen. Kun operaatio on suoritettu, kontrolli palaa jälleen kehykselle.

Rajapintojen toteutuksessa jollekin kehyksen sisältämälle rajapinnalle annetaan tuotekohtainen toteutus luokkana tai komponenttina. Kehys näkee tällöin tuotekohtaisen luokan ilmentymän tai komponentin rajapinnan kautta, ja pyytäessään jotakin rajapintaan kuuluvaa palvelua tulee kutsuneeksi tuotekohtaista koodia.

Parametrointimekanismeja käytettäessä tuotekohtainen alustuskomponentti voi käyttää esimerkiksi rakentajien parametreja luodessaan kehyksen luokista erilaisia ilmentymiä tai jotakin kehyksen luokan operaatiota kutsuessaan antaa parametrien avulla kehykselle tietoa halutusta toimintamuunnelmasta.

Ohjelmointikielet kuten C++ ja uusimmat Java-versiot tarjoavat mahdollisuuden geneeristen ohjelmarakenteiden käyttöön. Esimerkiksi luokkakaavain, josta ei sellaisenaan voi luoda ilmentymiä vaan joka pitää ensin konkretisoida kiinnittämällä geneeriset parametrit. Tyypillisesti geneerisen luokan tapauksessa parametroidaan jokin tietotyyppi, jota luokka käsittelee. Kehysten yhteydessä sovelluskohtainen koodi voi erikoistaa kehyksen tarjoamia geneerisiä luokkia kiinnittämällä näiden todellisiksi parametreiksi joitakin sovelluksen omia luokkia.

Kieli on refleksiivinen, jos se sallii ohjelman tutkia ja mahdollisesti jopa muuttaa itseään. Refleksiivisyyttä hyödyntämällä kehys voi kysyä ajoaikana tietoa sovelluskohtaisesta koodista, ja ohjata omaa toimintaansa tämän tiedon mukaisesti (esimerkiksi sovelluskohtaisen olion julkisten attribuuttien nimet, ja käyttää niitä hyväksi käyttöliittymässä)

Viime kädessä myös itse sovellus voi tarjota loppukäyttäjälle toimintoja, jotka voidaan tulkita sovelluksen ajoaikaiseksi erikoistamiseksi. Käyttäjä voi esimerkiksi räätälöidä käyttöliittymän mieleisekseen (esim. tekstinkäsittelysovellukset), tai kuvata erityisellä skriptikielellä tietyssä tilanteessa aktivoituvia toimintoja (esim. taulukkolaskentasovellukset).

Kehyslajit

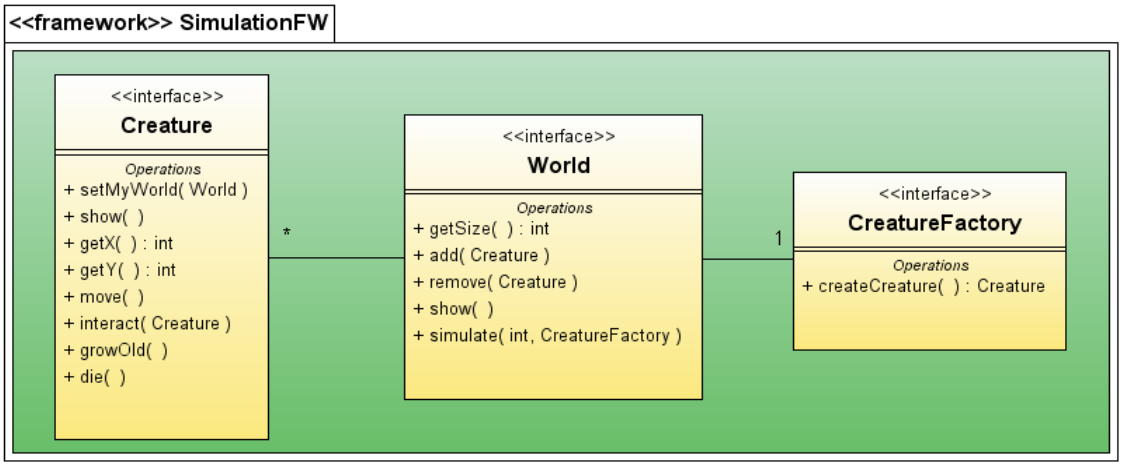
Kehykset luokitellaan yleisesti sen mukaan, mikä on kehyksen pääasiallinen erikoistamismekanismi. Tämä luokittelu ei ole tarkka, koska käytännössä kehykseen voidaan soveltaa useita erikoistamismekanismeja. Seuraavaksi käydään läpi seuraavanlaiset kehykset:

- Abstraktit kehykset
- Muunneltavat kehykset
- Plugin-kehykset
- Koottavat kehykset

Esimerkkinä toimii eliöpopulaatioiden käyttäytymistä simuloiva kehys.

7.1.5 Abstraktit kehykset

Abstrakti kehys ei sisällä lainkaan suoritettavaa koodia, vaan ainoastaan rajapintoja. Se ei myöskään anna simulointisovellusten yhteistä perustoiminnallisuutta vaan määrittelee rajapintojen avulla sovelluksen luokkien tai komponenttien palvelut. Esimerkissä (kuva 7.5) kehys sisältää simulointimaailman (World), eliöiden (Creature) sekä eliötehtaan (CreatureFactory) rajapinnat.



Kuva 7.5 Abstrakti kehys (Koskimies, Mikkonen, 2005)

Abstrakti kehys ei pahemmin rajoita sovelluskehittäjää, koska kehys edellyttää toteutukselta ainoastaan rajapintojen noudattamista. Mallin muu informaatio yhdessä luokkien ja operaatioiden nimien (ja parametrityyppien) kanssa antaa vihjeitä siitä, miten toteutus on ajateltu tehtävän. Abstraktia kehystä on luontevinta käyttää kerroksittaisessa kehyksessä (seuraava ratkaisumalli) yleisimmän kerroksen rajapintana. Abstrakti kehys erikoistetaan antamalla konkreettinen kehys, joka noudattaa abstraktin kehyksen rajapintoja.

7.1.6 Muunneltavat kehykset

Muunneltava kehys (white-box framework) tarjoaa erikoistamisrajapinnassaan luokkia, jotka toimivat kantaluokkina sovelluskohtaisille aliluokille. Aliluokat antavat toteutuksen yhdelle tai useammalle kantaluokan operaatiolle. Sovelluskehittäjä tyypillisesti antaa sovelluskohtaisen alustuskoodin, joka luo näiden aliluokkien ilmentymiä ja rekisteröi ne kehykselle. Alustuksen jälkeen pääkontrolli siirretään kehykselle, joka kutsuu sovelluskohtaisten olioiden palveluja Hollywood-periaatteen mukaisesti. Muunneltavassa kehyksessä pääasiallisena erikoistamismekanismina on periyttäminen.

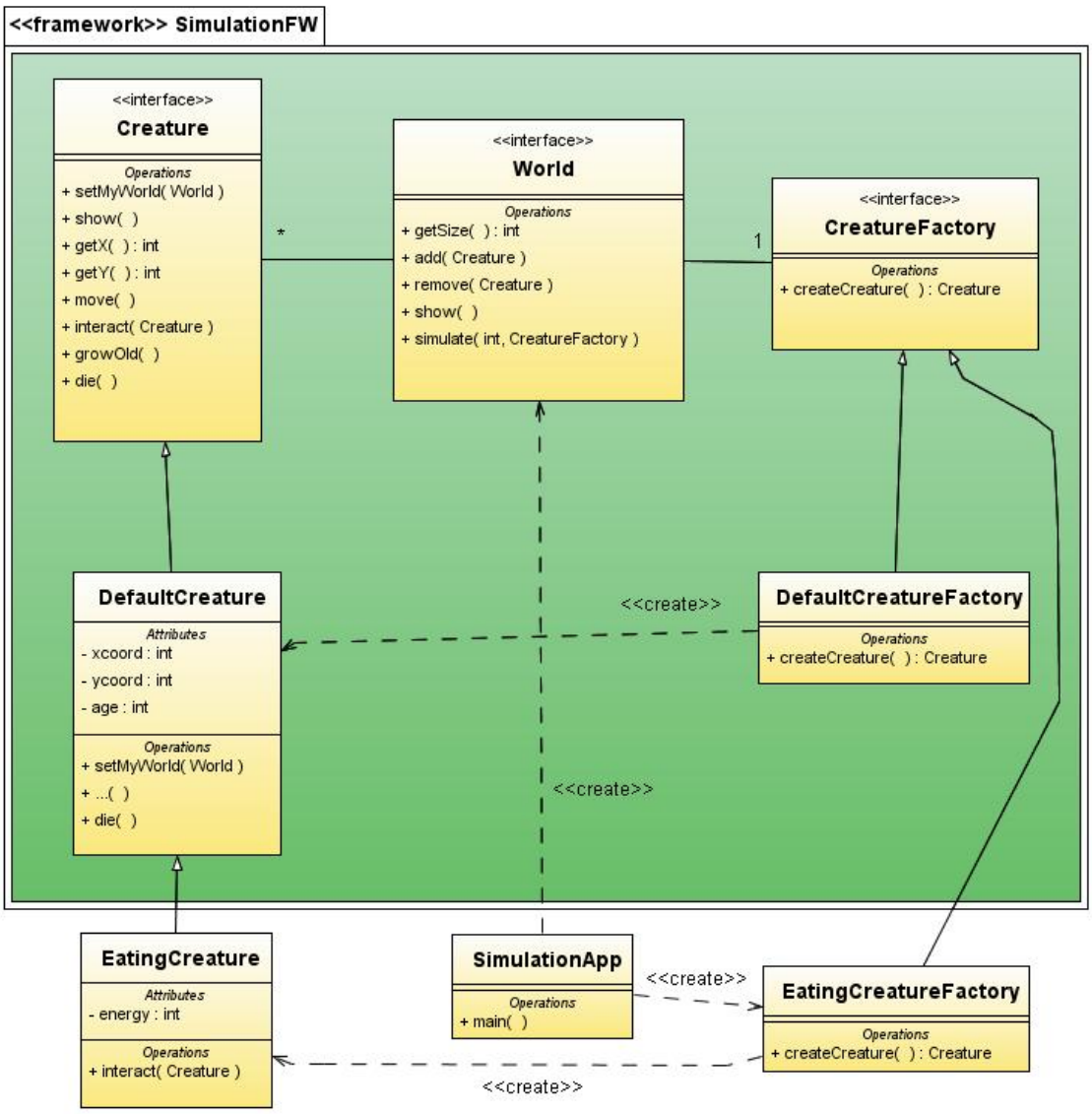
Esimerkkikehyksessä (kuva 7.6) kehys koostuu eliö- ja eliötehdasrajapintojen lisäksi näiden oletustoteutuksista (kuvan luokat DefaultCreature ja DefaultCreatureFactory), jotka määrittelevät peruskäyttäytymisen. Simulointimaailman toteuttava luokka (*World*) käytetään sellaisenaan sovelluksessa.

Kehyspakkauksen ulkopuolinen erikoistus:

- Kehystä on käytetty toisiaan syövien eliöiden muodostaman populaatio simulointiin.
- Eliöiden käyttäytymistä kuvaava luokka (*EatingCreature*): annettu uusi toteutus eliöiden vuorovaikutukselle (operaatio *interact*), joka tässä tapauksessa sisältää syömistoiminnon.
- Sitä vastaavan olioita luova tehdasluokka (*EatingCreatureFactory*).

Alustuskoodissa (*SimulationApp*):

- Luodaan World-olio sekä *EatingCreatureFactory*-olio.
- Luodaan syöjäeliöitä ja liitetään ne maailmaan *add*-operaatiolla.
- Lopuksi siirretään pääkontrolli kehykselle kutsumalla maailman simulate-operaatiota (parametrina annetaan simulointiajan lisäksi tehdasolio, jotta kehys pystyisi myös itse tarvittaessa luomaan syöjäeliöitä).



Kuva 7.6 Muunneltava kehys (Koskimies, Mikkonen, 2005)

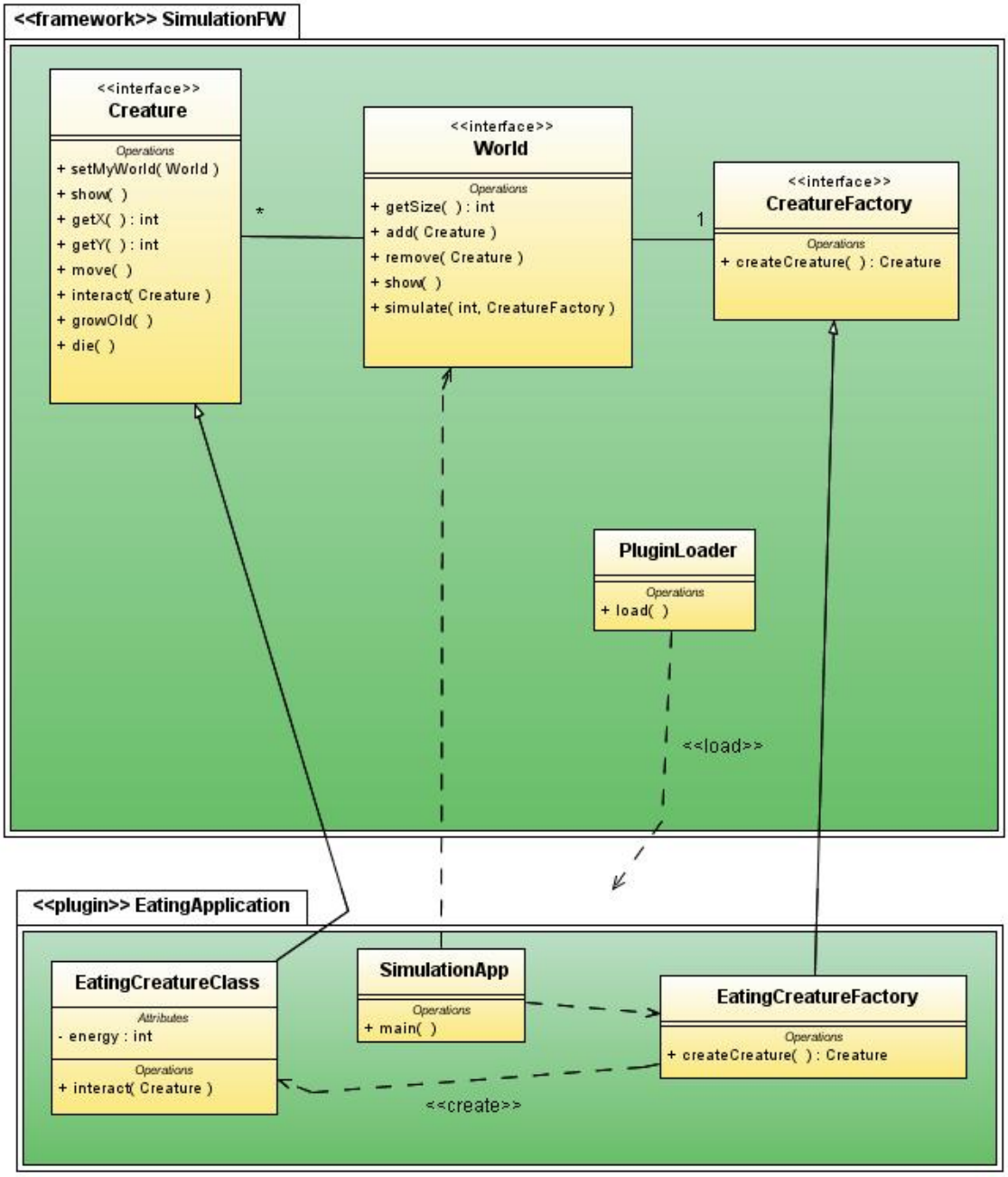
Menestyksellinen käyttäminen edellyttää, että sovelluskehittäjä tuntee hyvin kehyksen erikoistamisrajapinnan: mitkä luokat on tarkoitettu periytettäviksi, mille operaatioille voidaan antaa tai on annettava sovelluskohtainen toteutus, ja miten se tarkkaan ottaen tulisi tehdä. Hyvin suunniteltuna, dokumentoituna ja

esimerkkierikoistuksilla varustettuna muunneltava kehys tarjoaa voimakkaan välineen sovelluskehitykseen. Sellaisessa ympäristössä, jossa kehyksen suunnittelijat ja käyttäjät ovat samassa organisaatiossa (esimerkiksi yrityksessä), muunneltavan kehyksen edut tulevat hyvin esiin. Suurin riski muunneltavan kehyksen yhteydessä on sen kasvaminen suureksi, monoliittiseksi ohjelmistoksi, jonka erikoistamisrajapintaa kukaan ei enää hallitse.

7.1.7 Plugin-kehykset

Plugin-kehyksen (plug-in framework) pääasiallinen erikoistamismekanismi on rajapintojen toteutus. Kehys erikoistetaan toteuttamalla kehyksen sisältämiä rajapintoja sovelluskohtaisilla laajennosyksiköillä (plug-in), ja rekisteröimällä toteutukset kehykselle. Laajennosyksikkö koostuu tyypillisesti yhdestä tai useammasta luokasta tai komponentista. Kehys rekisteröi laajennosyksiköt automaattisesti lataamalla ne sovitusta hakemistosta - kehys voidaan erikoistaa sijoittamalla halutut toteutukset laajennoshakemistoon ennen järjestelmän käynnistystä. Esimerkki plugin-kehyksestä on Eclipse (erilaisten ohjelmistokehitysympäristöjen tekemiseen tarkoitettu Java ohjelmistoalusta). Laajennosyksikkö voi määritellä edelleen omia laajennoskohtiaan (extension point), joita muut yksiköt laajentavat. Laajennoskohta on tavallisesti rajapinta, jonka laajentava yksikkö toteuttaa. Pientä ydintä lukuun ottamatta kaikki uuden ympäristön toiminnallisuus tulee määriteltyä rajapintoja toteuttavissa laajennosyksiköissä.

Esimerkissä (kuva 7.7) sovelluksen laajennosyksikkö antaa uuden eliötyypin toteutuksen, sen tarvitseman tehdasluokan sekä uuden sovelluksen pääohjelman. Kehys aktivoi käynnistyessään laajennosyksiköiden lataamisen (luokka PluginLoader), PluginLoader käynnistyttyään aktivoi ladatun sovelluksen pääohjelman (main-operaatio SimulationApp-luokassa).



Kuva 7.7 Plugin-kehys (Koskimies, Mikkonen, 2005)

Etuja verrattuna perinteiseen muunneltavaan kehykseen:

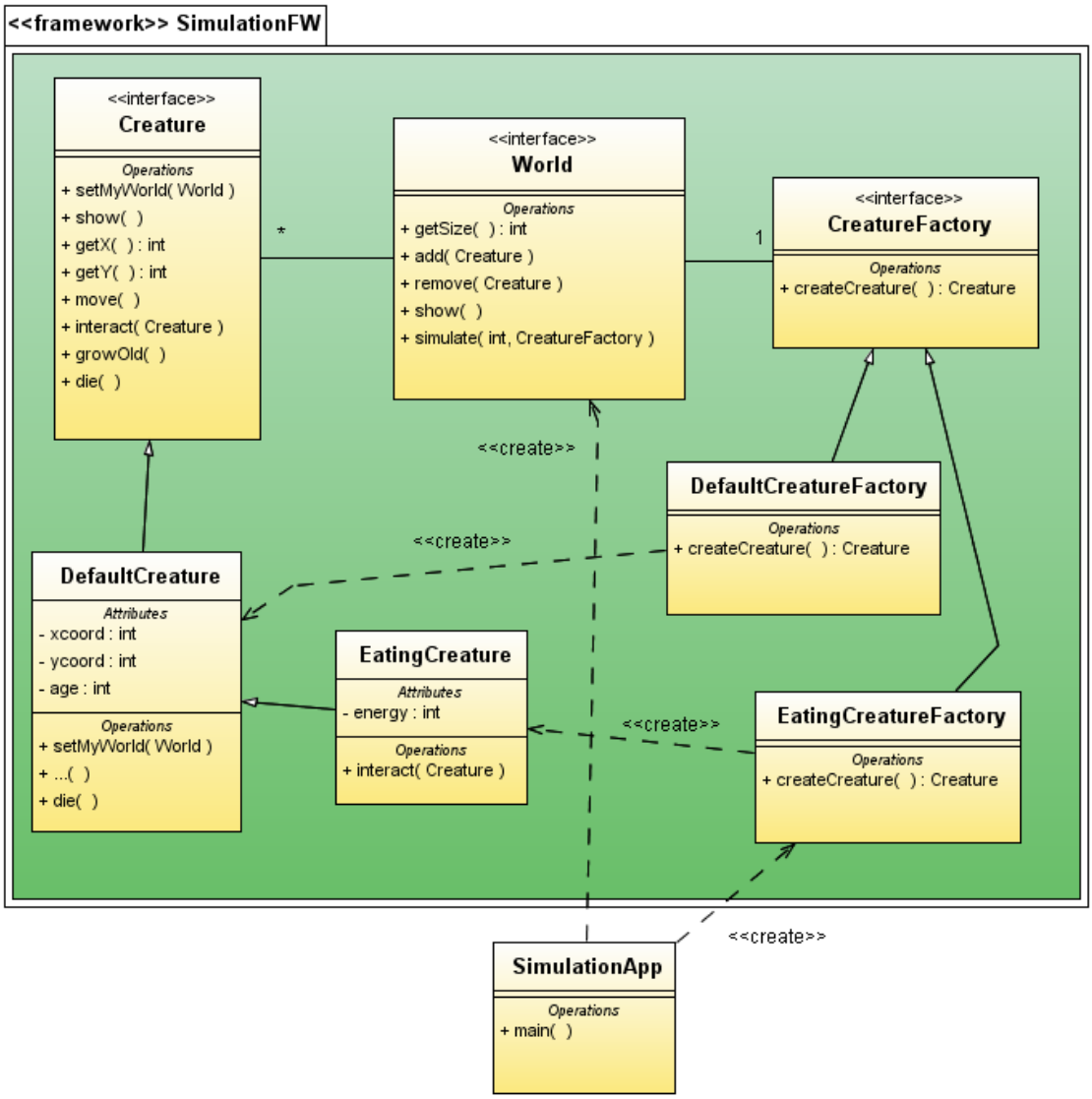
- Plugin-kehyksen erikoistamisrajapinta on helpommin nähtävissä, koska se koostuu rajapintojen toteutuksista.
- Laajennosyksiköt muodostavat luonnollisen tavan strukturoida erikoistus ja liittää se kehykseen.
- Periaatteessa plugin-kehys sallii jopa sovelluksen ajoaikaisen erikoistamisen.

7.1.8 Koottavat kehykset

Koottava kehys (black-box framework) on perinteisen uudelleenkäytettävän kirjaston ja kehyksen välimuoto. Pääkontrolli on kehyksellä, mutta kehys ei (takaisin)kutsu sovelluskohtaista koodia. Tarvittava muunneltavuus saadaan aikaan luomalla kehyksen luokkien ilmentymiä sopivilla alustusparametreilla, muodostamalla näistä ilmentymistä haluttuja konfiguraatioita, ja kutsumalla näiden palveluja sopivilla parametreilla ennen kontrollin siirtämistä kehykselle. Koottavaa kehystä pidetään usein muunneltavan kehyksen evoluution lopputuloksena. Kun sovellusalueella tarvittava muunneltavuus ymmärretään riittävästi, kehyksen luokat ovat muokkaantuneet tukemaan tätä muunneltavuutta niin hyvin, että niitä ei enää tarvitse periyttää. Näin kehyksen käyttö yksinkertaistuu huomattavasti ja tulee lähemmäksi tavanomaista kirjastoa. Koottavan kehyksen tapauksessa sovelluskehittäjän täytyy ymmärtää vain kehyksen luokkien palvelurajapinnat, mutta ei luokkien toteutustapaa.

Esimerkissä (Kuva 7.8) päätetään, että tulevissa simulointisovelluksissa tarvitaan vain tiettyjä peruseliötyyppejä. Kunkin perustyypin sisällä tarvittava muunneltavuus saadaan aikaan rakentajien ja muiden operaatioiden parametroinnilla. Siirretään EatingCreature-luokan tehtaineen kehyksen puolelle yhdessä muiden vastaavien peruseliötyyppejä kuvaavien luokkien kanssa. Tehtaita kannattaa edelleen käyttää eliöiden luontiin, koska näin saadaan muu osa kehyksestä riippumattomaksi valitusta eliötyypistä. Näin saadun koottavan kehyksen erikoistuskoodiksi riittää alustus (SimulationApp-luokan main-operaatio), joka luo halutun alkutilanteen simulointia varten käyttäen suoraan kehyksen luokkia.

Yleisesti koottavan kehyksen etuna on käytön helppous. Toisaalta koottava kehys tukee vain hyvin rajattua muunneltavuutta, koska alustuskoodia lukuun ottamatta soveluskehittäjällä ei ole mahdollista määritellä täysin uutta käyttäytymistä. Ratkaisu ei siten toimi sovellusalueilla, joilla variaatio on ainakin yksityiskohdissaan vaikeasti ennustettavaa.



Kuva 7.8 Koottava kehys (Koskimies, Mikkonen, 2005)

7.1.9 Yhteenvetoa kehysten luokittelusta

- Kehys ei juuri koskaan kuulu pelkästään yhteen edellä mainituista kategorioista.

- Lähes kaikissa kehyksissä on koottavan kehyksen piirteitä, vaikka niiden perusluonne olisikin jokin muu. Tämä johtuu siitä, että eri asioiden varioituvuus tietyllä sovellusalueella voi olla hyvinkin erilaajuista: joissakin asioissa varioituvuus saattaa olla rajattu muutamiin

vaihtoehtoihin, kun taas toisissa asioissa pitää voida sallia eri sovelluksissa lähes mielivaltainen käyttäytyminen.

- Tarvittavat erikoistamismekanismit vaihtelevat samassakin kehyksessä.

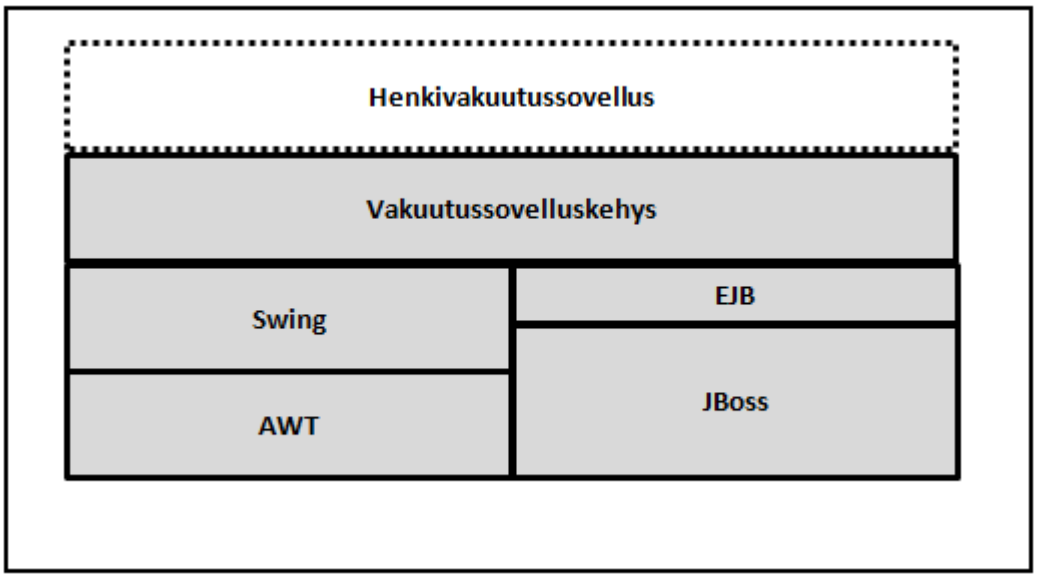
Kehysten strukturointi

Suuri riski kehysten käytössä on, että kehys kasvaa monoliittiseksi hallitsemattomaksi kokoelmaksi luokkia mutkikkaine yhteyksineen, joita kukaan ei hallitse. Kehysohjelmistojakin tehdessä on syytä muistaa modularisointi. Seuraavaksi käsitellään kolme tapaa strukturoida kehykset:

- Kerroksittaisen kehykset
- Hierarkiset kehykset
- Komponenttikehysten käyttö

7.1.10 Kerroksittaiset kehykset

Kerroksittaisen kehykset tarjoavat tavan jakaa kehysohjelmisto osiin eri abstraktiotasoille. Kehysten tapauksessa kerrokset jaotellaan niiden tarjoaman tuen yleisyyden mukaan. Mitä erikoistuneempi kehys sitä parempi tuki sovellusalueen kapenemisen hinnalla. Yleisesti alin kerros on yleinen ydinkehys, sen yläpuolella on vähän enemmän erikoistunut kehys, sitten tulee taas hieman erikoistuneempi, jne. Kunnes ylimpänä on sitten yksittäinen sovellus. Esimerkissä (kuva 7.9) kehyksen pohjalla on käyttöliittymän toteuttavat AWT ja sen päälle tehty Swing, sekä sovelluksen logiikkaa tukeva EJB ja sen toteuttava Jboss. Näiden yläpuolella on vakuutussovellusalueelle tarkoitettu kehys, ja sen yläpuolelle on rakennettu itsenäinen henkivakuutussovellus.



Kuva 7.9 Kerroksittainen EJB-pohjainen kehys (Koskimies, Mikkonen, 2005)

Edut:

- Kerrosrakenne jakaa muuten monoliittisen kehyksen helpommin ymmärrettäviin osiin, joista kullakin on oma abstraktiotasonsa, erikoistamisrajapintansa ja sovellusalueensa.

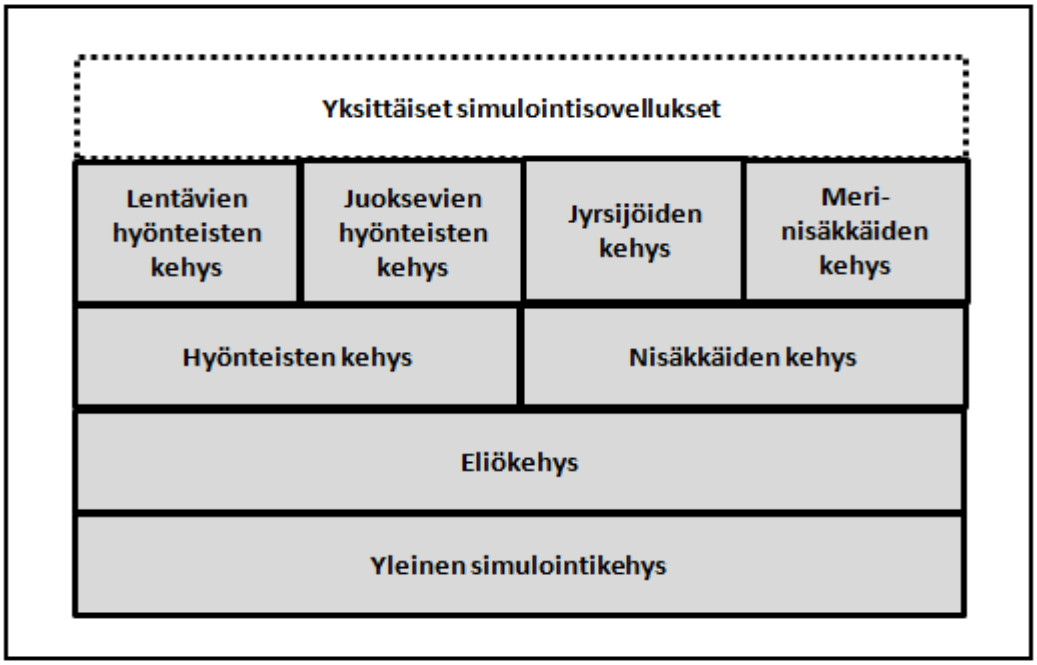
- Kerroksia voidaan ylläpitää omina kokonaisuuksinaan, ja tiettyä kerrosta voidaan käyttää useiden erikoistuneempien kehysten pohjana.

- Kerroksittainen kehys tarjoaa useita eri abstraktiotasoilla olevia erikoistamisrajapintoja, joista sovelluskehittäjä voi valita parhaiten sopivan.

7.1.11 Hierarkiset kehykset

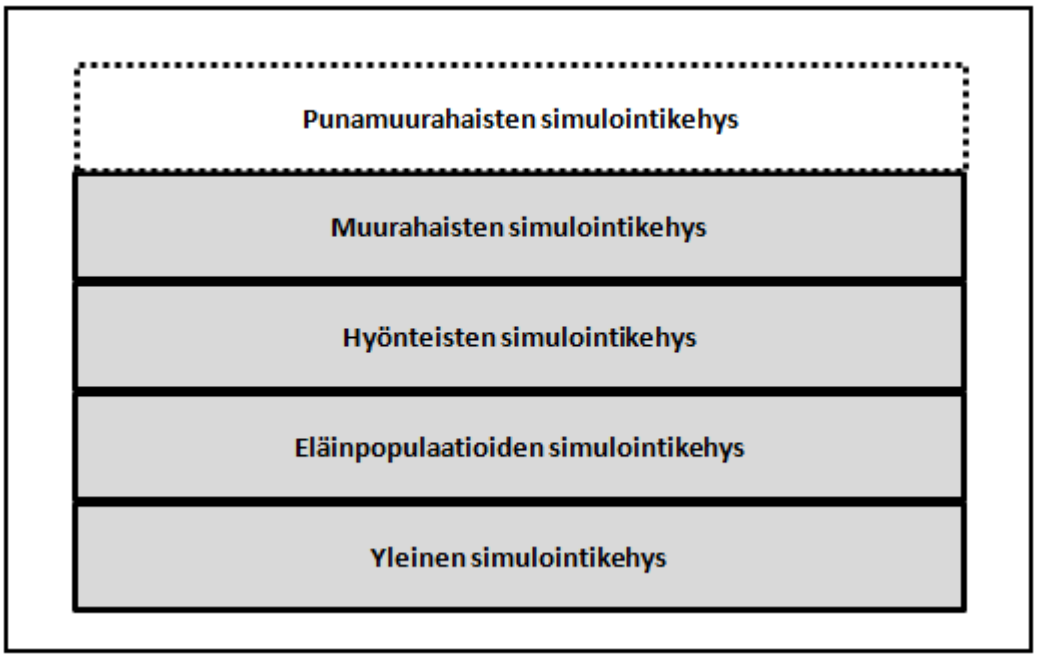
Kerroksittaisen kehyksen ideaa voidaan suoraviivaisesti yleistää haarauttamalla erikoistamispolut. Tällöin pohjalla oleva kehys voidaan erikoistaa kahdeksi tai useammaksi kapeampaa sovellusaluetta tukevaksi kehykseksi. Esimerkiksi kuvassa 7.10, simulointikehyksen tapauksessa hierarkkinen kehys saadaan luontevasti eliöluokittelujen mukaan. Alimmassa kerroksessa on tällöin eliölajeista riippumaton simulointituki, seuraavaksi ylemmässä kerroksessa

kaikille eliöille yhteinen oletuskäyttäytyminen. Tämä kerros erikoistetaan hyönteisille ja nisäkkäille. Tämän jälkeen hyönteisten ja nisäkkäiden kehykset erikoistetaan vielä kapea-alaisemmiksi kehyksiksi.



Kuva 7.10 Hierarkinen simulointikehys (Koskimies, Mikkonen, 2005)

Kuvassa 7.11 on haarautumaton hierarkinen kehys.



Kuva 7.11 Hierarkinen simulointikehys (TuTY –kalvot 2008)

Puhdas hierarkkinen kehys on käytännössä harvinainen, koska se edellyttää, että kehyksen sovellusalue voidaan jäsentää luontevasti sekä vertikaalisessa suunnassa eri abstraktiotasoihin että horisontaalisessa suunnassa eri segmentteihin.

Edut:

- Hierarkkisen kehyksen olennainen etu kerroksittaiseen kehykseen nähden on erikoistamisrajapintojen suurempi valikoima.
- Hierarkkinen kehys tarjoaa paitsi eri yleisyystasoilla olevia erikoistamisrajapintoja myös samalla yleisyystasolla olevia mutta eri sovellusalueen segmenteille tarkoitettuja erikoistamisrajapintoja.
- Sovelluskehittäjän on helpompi löytää juuri hänen tarpeisiinsa sopiva erikoistamisrajapinta, eikä hänen tarvitse ymmärtää kuin pieni osa koko kehysjärjestelmästä.

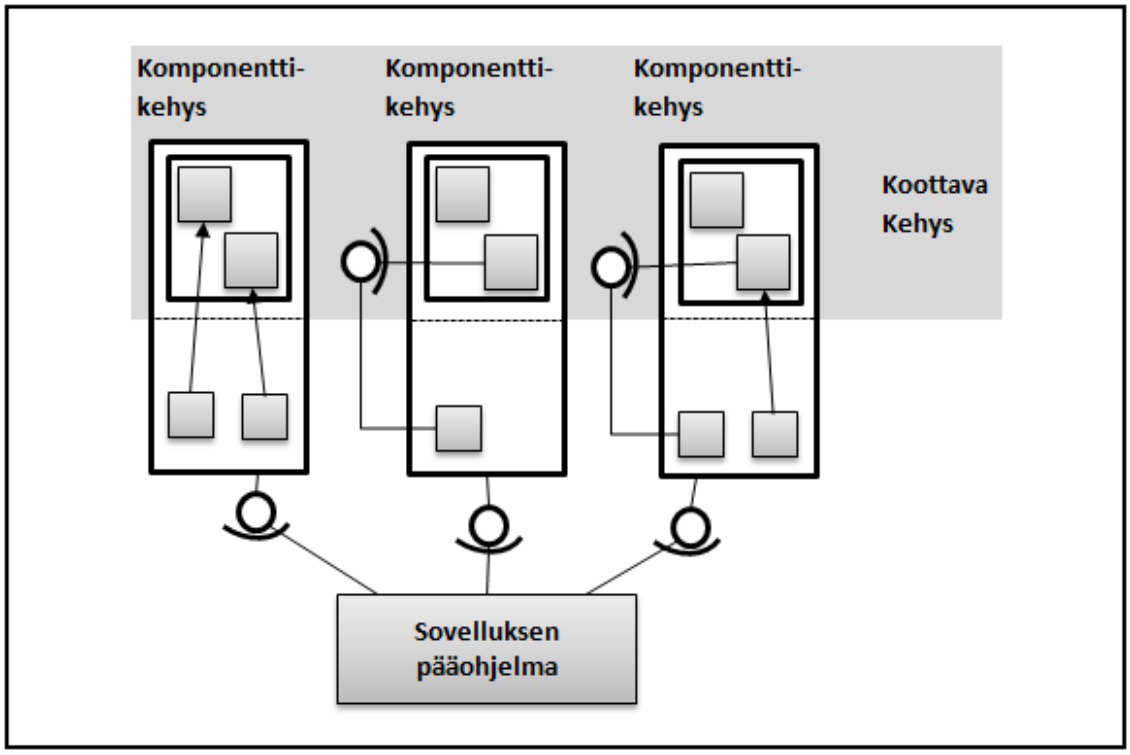
7.1.12 Komponenttikehysten käyttö

Koottavan kehyksen tapauksessa on mahdollista organisoida kehys joukoksi komponentteja ja rajapintoja siten, että haluttu variaatio saadaan aikaan yhdistelemällä kehykseen kuuluvia komponentteja sopivalla tavalla. Komponenttipohjaista rakennetta voidaan soveltaa myös muunneltavan kehyksen tapauksessa; tällöin muunneltavuus ei voi perustua pelkästään komponenttien valitsemiseen ja yhdistelyyn. Koska komponentin tulee olla itsenäinen ohjelmayksikkö, kullakin komponentilla tulisi olla myös oma erikoistamisrajapintansa. Näin kukin komponentti voi toimia ikään kuin pienenä kehyksenä, komponenttikehyksenä (framelet). Kun tällainen komponenttikehyksistä koostuva kehys halutaan erikoistaa, erikoistetaan valitut komponenttikehykset (periyttämällä tai vaadittujen rajapintojen toteutuksella), ja näin saadut varsinaiset konkreettiset komponentit liitetään sen jälkeen yhteen niin, että ne muodostavat halutun sovelluksen.

Komponenttikehyksiin perustuva kehys yhdistää koottavien ja muunneltavien kehysten hyvät puolet. Se sallii voimakkaat erikoistamismekanismit,

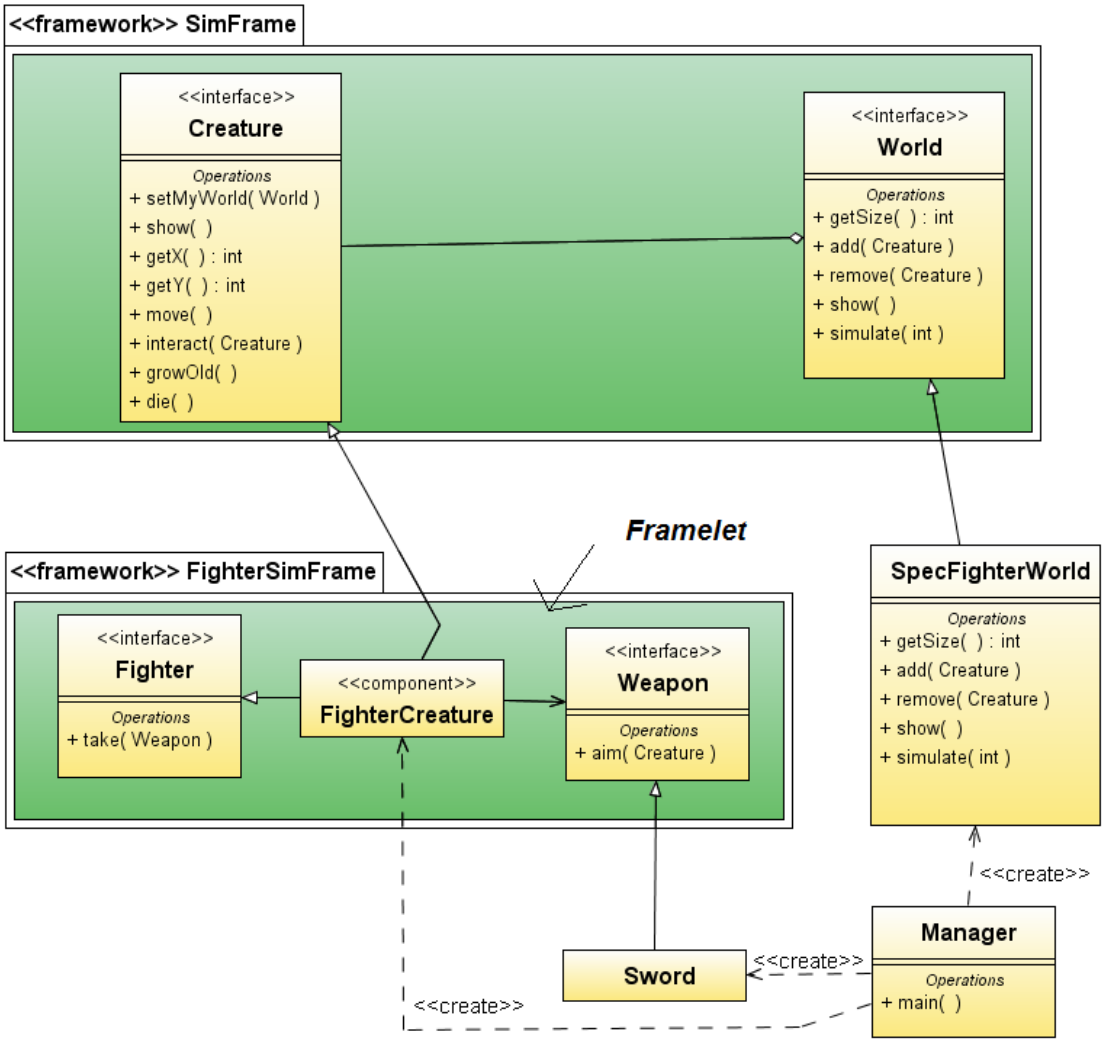
erikoistamisrajapinta pysyy hyvin hallittuna ja ymmärrettävänä, koska se koskee aina vain yhtä komponenttia. Korkeimmalla tasolla sovellus on koottu tavalliseen tapaan komponenteista hyvien modularisointiperiaatteiden mukaisesti.

Kuvassa 7.12 on kuvattu komponenttikehyksiin perustuva kehys. Tässä sovelluskehittäjä voi itse koostaa sovelluksen käyttämällä halumiaan erikoistettuja komponentteja. Komponentit on erikoistettu periyttämällä ja vaadittujen rajapintojen toteuttamisella.



Kuva 7.12 Komponenttikehyksiä käyttävä kehys (Koskimies, Mikkonen, 2005)

Kuvassa 7.13 on aikaisemmin esimerkkinä olleen simulointikehyksen jatkoa. Kuvassa on yhdistetty komponenttikehyksiä niin, että maailmaan kuuluu taistelevia otuksia, jotka voivat käyttää aseita, ja käyttää niitä toisia otuksia vastaan.



Kuva 7.13 Komponenttikehys simulointiesimerkin mukaan luokkakaaviona (TaTY –kalvot 2006)

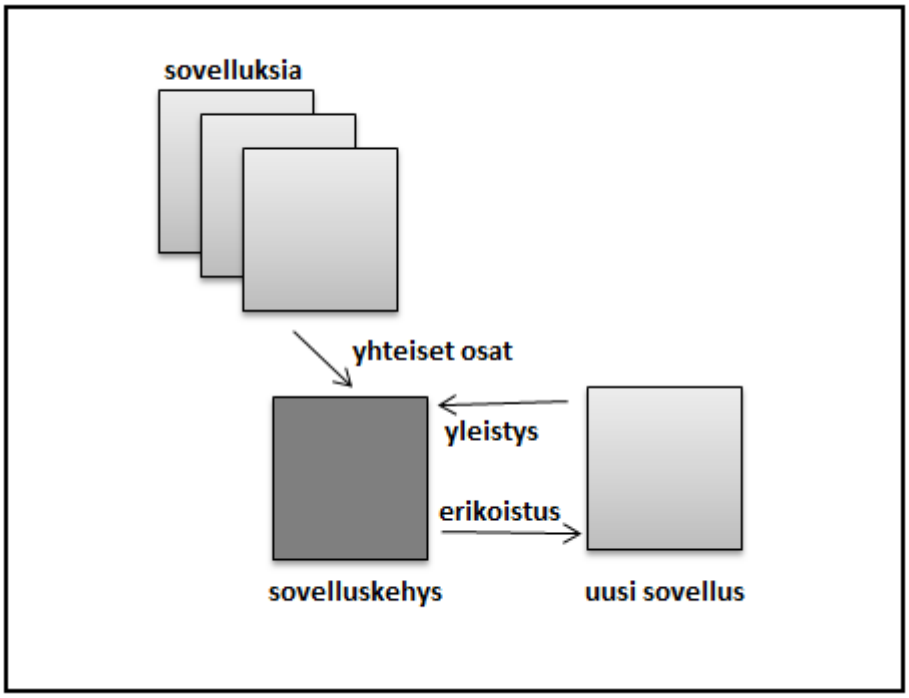
Kehysten suunnittelu

Kehysohjelmistoja suunnitellessa voidaan soveltaa tuoterunkoihin liittyviä malleja ja tekniikoita, koska kehysohjelmistot ovat olioperustainen tapa toteuttaa tuoterunkoarkkitehtuureja ja niitä tukevia ohjelmistoalustoja.

7.1.13 Iteratiivinen rakentamisprosessi

Sovelluskehysten suunnittelu lähtee yleensä liikkeelle siitä, että jollakin sovellusalueella on useampia samanlaisia sovelluksia. Tätä sovellusaluetta

sitten lähdetään tukemaan ohjelmistokehyksellä, jota varten analysoidaan jo olemassaolevat sovellukset, jotta löydetään kaikkien yhteiset osat. Näin saadaan aikaiseksi ensimmäinen versio kehyksestä, jota sitten käytetään seuraavan sovelluksen tekemiseen. Usein tämä ei vielä sovi tarkoitukseensa ilman yleistämistä, jota tehdään kunnes kehys on tarpeeksi joustava ja yleinen. Prosessia on kuvattu kuvassa 7.14.



Kuva 7.14 Kehyksen iteratiivinen rakennusprosessi (Koskimies, Mikkonen, 2005)

7.1.14 Suunnittelu

Vaatimusanalyysin tuloksena saadaan sovellusalueen malli, sovellusten yhteiset vaatimukset ja sovellusten muunneltavuusvaatimukset. Lisäksi vaatimuksiin voidaan sisällyttää yksi tai useampi tyypillisen esimerkkisovelluksen kuvaus omine vaatimuksineen ja käyttötapauksineen.

Alustava arkkitehtuurisuunnittelu tehdään käyttäen hyväksi sovellusalueen mallia ja mahdollisia esimerkkisovellusten vaatimuksia. Tässä vaiheessa ei vielä kiinnitetä huomiota muunneltavuuteen, vaan pyritään löytämään kyseiselle

alueelle sopiva yleinen arkkitehtuuri. Keskeinen työskentelytapa tässä vaiheessa on yleistäminen: arkkitehdin tulisi löytää sovellusalueen olennaiset käsitteet ja mekanismit yleistämällä esimerkkisovellusten vaatimuksia ja analysoimalla sovellusalueen mallia. Jos kehys on tarkoitettu tukemaan myös käyttöliittymää, sitä koskevat vaatimukset on samoin yleistettävä sopivan käyttöliittymäarkkitehtuurin löytämiseksi. Vaiheen tuloksena saadaan kehyksen alustava arkkitehtuuri.

Arkkitehtuurisuunnittelussa otetaan tarkasteluun yksi muunneltavuusvaatimus kerrallaan ja analysoidaan, tarvitaanko kyseisen muunneltavuuden tukemiseen arkkitehtuuritason ratkaisua vai riittääkö siihen yksityiskohtaisen suunnittelutason ratkaisu. Jos muunneltavuutta on tuettava arkkitehtuuritasolla, muokataan arkkitehtuuria halutun muunneltavuuden saavuttamiseksi. Ratkaisussa voidaan hyödyntää suunnittelumalleja. Ratkaisu dokumentoidaan nk. erikoistamismallina, jossa kuvataan tähän muunneltavuuskohtaan liittyvä arkkitehtuuriratkaisu sekä sen hyödyntäminen sovelluksen tekemisen yhteydessä. Vaiheen tuloksena saadaan kehyksen arkkitehtuurikuvaus yhdessä arkkitehtuuritason erikoistamismallien kanssa sekä luettelo myöhempiin vaiheisiin siirretyistä muunneltavuusvaatimuksista.

Yksityiskohtaisessa suunnittelussa käydään läpi yksityiskohtaisen suunnittelun tasolle siirretyt muunneltavuusvaatimukset ja annetaan ne toteuttavat suunnitteluratkaisut. Tässäkin voidaan ratkaisun perustana usein käyttää suunnittelumalleja. Ratkaisut sekä niiden käyttö erikoistuksen yhteydessä kuvataan erikoistamismalleina. Vaiheen tuloksena saadaan kehyksen yksityiskohtainen suunnitteludokumentti suunnittelutason erikoistamismalleineen.

Arkkitehtuurin arvioinnissa analysoidaan, miten hyvin saatu arkkitehtuuri tukee annettuja muunneltavuusvaatimuksia. Tämä voidaan tehdä käyttäen yleisiä arkkitehtuurin arviointimenetelmiä tai näyttämällä, miten vaatimuksiin sisällytetyt esimerkkisovellukset saadaan aikaan kehyksen erikoistamismalleja soveltamalla. Mikäli kehyksen joustavuudessa havaitaan puutteita, palataan

aikaisempiin vaiheisiin. Tässä vaiheessa voidaan löytää myös uusia muunneltavuusvaatimuksia.

Toteutuksessa kehys ohjelmoidaan halutulla ohjelmointikielellä. Ohjelmoinnissa on otettava huomioon, että pienin muunneltava ohjelmayksikkö on funktio (metodi), joka voidaan uudelleenmääritellä aliluokissa. Riittävän hienojakoisen muunneltavuuden saavuttamiseksi kannattaa kaikki mahdolliset erikseen muunneltavat toiminnallisuudet ohjelmoida erillisiksi funktioiksi.

Testauksessa pyritään varmistumaan kehyskoodin virheettömyydestä. Ongelmana on, että kehyksen tulisi toimia yhdessä kaikkien mahdollisten erikoistuksien ja muunnelmakombinaatioiden kanssa. Usein tyydytään testamaan kehys yhdessä muutamien edustavien esimerkkierikoistuksien kanssa, jotka kattavat mahdollisimman hyvin kehyksen koodin. Testaamista helpottaa, jos kehys on jaettu komponenttikehyksiksi, jolloin kukin komponenttikehys voidaan testata erikseen.

7.1.15 Erikoistaminen

Vaatimusanalyysissä varmistetaan, että sovelluksen vaatimukset mahtuvat kehyksen tarjoaman muunneltavuuden sisään: vaatimusten on toteuduttava joko suoraan kehyksen ominaisuuksien kautta tai kehyksen tukeman muunneltavuuden kautta. Analyysi voidaan tehdä vertaamalla kehyksen vaatimuksia sovelluksen vaatimuksiin.

Suunnittelussa käydään läpi ne vaatimukset, jotka edellyttävät kehyksen erikoistamisrajapinnan käyttöä, ja kuvataan erikoistamismalleja hyödyntämällä, kuinka kukin tällainen vaatimus toteutetaan erikoistamisrajapinnan avulla. Vaiheen tuloksena saadaan dokumentti, jossa erikoistamisrajapinnan käyttö tämän sovelluksen tapauksessa on kuvattu suunnittelutasolla. Tällainen kuvaus voidaan antaa esimerkiksi UML-luokkakaaviona, jossa näkyvät sovelluskohtaisten luokkien lisäksi ne kehyksen luokat, jotka liittyvät näihin (esimerkiksi kantaluokat).

Toteutuksessa annetaan sovelluskohtainen koodi edellisen vaiheen suunnitteludokumentin mukaisesti.

Testaus suoritetaan kuten minkä hyvänsä järjestelmän testaus. Testitapaukset tulisi suunnitella niin, että ne kattavat sovelluskohtaisen koodin. Testauksessa on tärkeätä, että myös kehyksen lähdekoodi on saatavilla, jotta virheet pystytään jäljittämään. Sovelluksen testaus voi paljastaa vikoja myös kehyksessä.

7.1.16 Kerroksittaisen kehyksen suunnittelu

Määritellään sopivan esimerkkisovelluksen vaatimukset. Yleistetään näitä vaatimuksia korvaamalla sovelluskohtaiset käsitteet yleisemmillä. Käsitteen yleistys johtaa samalla myös uuteen muunneltavuusvaatimukseen tämän käsitteen kohdalla. Kun saadaan ensimmäisen tason yleistetty käsitemalli ja vaatimukset, jatketaan prosessia pyrkimällä yleistämään nämä käsitteet vastaavalla tavalla. Tätä toistetaan niin kauan kuin löytyy luonnollisia, yleisempiä käsitetasoja. Tuloksena syntyy joukko asteittain yleistyviä kehysvaatimuuksia. Tyypillisesti tasoja voisi olla 2–3. Yleisin vaatimusmäärittely on pohjana ydinkehyksen toteutukselle. Tätä erikoistamalla saadaan toteutus seuraavan tason kehykselle jne. Kunnes lopulta voidaan toteuttaa esimerkkisovelluksen vaatimukset erikoistamalla viimeinen kehystaso. Lopputuloksena saadaan kerroksittainen kehys.

7.1.17 Erikoistamismallit

Erikoistamismalli sitoo muunneltavuusvaatimukset kehyksen erikoistamisrajapintaan kuvaamalla, miten tietty muunneltavuus saadaan aikaan hyödyntämällä kehyksen arkkitehtuuri- ja suunnitteluratkaisuja. Erikoistamismallin esitysmuodossa voidaan soveltaa suunnittelumallien esitysmuotoa: erikoistamismalli voi hyvin olla suunnittelumallin sovellus.

Mallissa ilmoitetaan muunneltavuusvaatimus, johon malli liittyy, sekä muunnelman kiinnitysaika. Muunneltavuuteen liittyvä suunnitteluratkaisu kuvataan UML:n luokkakaaviolla, jossa on varjostettu kehykseen kuuluvat osat. Ratkaisuun liittyvät elementit ja niiden rooli ratkaisussa selitetään. Lisäksi kuvataan rajoitukset ja erityisvaatimukset. Lopuksi voidaan antaa vielä esimerkkikoodi erikoistuksesta.

7.1.18 Esimerkki erikoistamismallista

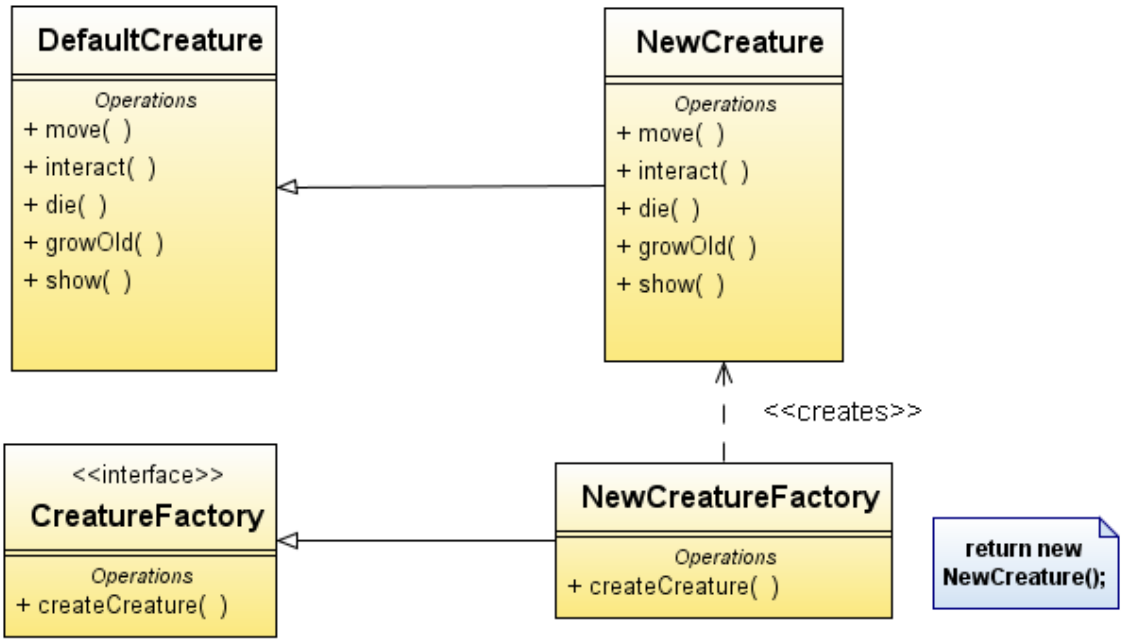
Muunneltavuusvaatimus: M23

Muunneltavuusvaatimuksen kuvaus:

eliön käyttäytyminen (liikkuminen, vuorovaikutus, vanheminen ja kuolema) on voitava määritellä mielivaltaisesti

Muunneltavuuden kiinnitysaika: käännösaika

Rakenne:



Selitykset

DefaultCreature: oletustoteutus eliölle

NewCreature: uusi sovelluskohtainen eliötyyppi

CreatureFactory: eliöiden tehdasluokan rajapinta
NewCreatureFactory: sovelluskohtaisten eliöiden tehdasluokka
move: yhden aikayksikön aikana tapahtuva eliön liike
interact: vuorovaikutus kahden eliön välillä
die: eliön kuolemiseen liittyvät toimet
growOld: eliön vanheminen
show: eliön ulkoasun tuottaminen näytölle

Rajoitteet

NewCreature: rakentajan täytyy kutsua yliluokan rakentajaa
NewCreatureFactory/createCreature: täytyy palauttaa NewCreature-olio
move: täytyy kutsua “show”-operaatiota, muuttaa yleensä x- ja y-koordinaatteja
die: täytyy poistaa eliöolio maailmasta
growOld: täytyy lisätä age-attribuutin arvoa

Esimerkki

```
class NewCreature extends DefaultCreature
 {
        int energy;
        public NewCreature(int x, int y, int e)
         {
            super(x, y);
            energy = e;
        }
        public void move()
        {
            xcoord = (xcoord+1)%myWorld.getSize();
            show();
        }
        public void show() {...}
        public void interact(AbstractCreature c)
        {
            if(c != this && c instanceof NewCreature)
```

```
            {
                if(((NewCreature)c).energy < energy)
                {
                    c.die();
                }
            }
            }
}
```

Esimerkki lähteestä (Koskimies, Mikkonen, 2005)

Kehysten käytön ongelmia

- Tekninen vaativuus ja monimutkaisuus.
    - Korkeat osaamisvaateet: sovellusalue + oliotekniikat + suunnittelumallit
    - Tulevaisuuden tarpeiden ennustaminen
- Kehysten yhdistely
    - Kehysten yhdistelytilanteeseen liittyvät vastuut, millä kehyksellä päävastuu?
- Monoliittisuus
    - Hallitsematon kasvu, ylläpito vaikeutuu.
- Laadullinen varianssi
    - Varianssi koskee tavallisesti toiminnallisia ominaisuuksia.
    - Järjestelmän laatuvaatimusten variointi vaikeaa.
- Dokumentointi
    - Yleisten dokumentointikäytäntöjen puute

Yhteenveto luvusta

- Ohjelmistokehykset ovat olioperustainen tuoterunkojen toteutustekniikka.

- Kehys erikoistetaan ohjelmistotuotteeksi täydentämällä siinä olevat aukot tuotekohtaisella koodilla.

- Kehysten avulla ohjelmoija uudelleenkäyttää jo toteutettua perusarkkitehtuuria kokonaisuutena.

- Kehyksiä on useita eri tyyppejä perustuen siihen, mitä erikoistamisen tuloksena syntyy ja mitä mekanismeja käytetään kehyksen erikoistamisessa.

- Kehysohjelmisto voidaan strukturoida kerrosten tai muunneltavien komponenttien avulla.

- Kehysten erikoistamisrajapinta voidaan kuvata erikoistamismalleilla, jotka liittävät muunneltavuusvaatimukset suunnitteluratkaisuihin.

- Kehysten käytön ongelmia ovat niiden suunnittelun ja käytön tekninen vaativuus ja heikko menetelmätuki.

8. ARKKITEHTUURIEN ARVIOINTI

Arkkitehtuurihan ei käsittele osien sisäisiä rakenteita, mutta arkkitehtuurin arvioinnissa arvioidaan komponenttien ja alijärjestelmien suhteita ja niiden ominaisuuksia. Arkkitehtuurin arvioinnissa tutkitaan myöskin pystyykö järjestelmä täyttämään sille asetetut vaatimukset, sekä pidemmän tähtäimen tavoitteet, kuten laajennettavuuden, muunneltavuuden ja skaalautuvuuden ilman että suorituskyky tai muistinkulutus kärsivät liikaa.

Arvioinnin perusteet

Yleensä arkkitehtuuri ratkaisee sen, kuinka hyvin ohjelmisto tulee täyttämään sille asetetut laadulliset vaatimukset. Jotta nämä laadulliset ominaisuudet saadaan arvioitua kattavasti on tärkeää, että arkkitehtuuri sisältää kaikki laadulliseen ominaisuuksiin vaikuttavat ratkaisut. Jos jotakin laadullista ominaisuutta ei voida arvioida arkkitehtuurin perusteella, on arkkitehtuuri puuttellinen.

Yleisiä laatuvaatimuksia ovat esimerkiksi:

- suorituskyky
- luotettavuus
- saatavuus
- turvallisuus
- muunneltavuus
- siirrettävyys
- varioitavuus

Nämä laatuvaatimukset eivät sellaisinaan ole yksiselitteisiä. Tälläisiä yleisiä laatuvaatimuksia ei pitäisi olla järjestelmän vaatimuksissa ilman täsmennyksiä.

Myös toiminnallisia vaatimuksia voidaan arvioida arkkitehtuurista. Tällöin yksinkertaisesti tutkitaan pystyykö arkkitehtuuri toteuttamaan halutut toiminnot. Tälläinen on suhteellisen helppo arvioida vaikkapa osoittamalla sekvenssikaaviolla, että tiettyyn toimintoon tarvittava tuki löytyy arkkitehtuurista.

Arkkitehtuurin arviointi helpottaa suunnittelijoita ymmärtämään järjestelmää paremmin ja tutkimaan sitä joka kantilta systemaattisesti eri ongelmakohtien löytämiseksi.

Arvioinnin sidosryhmiä

Arkkitehtuurin arviointiin liittyy muitakin sidosryhmiä kuin pelkästään suoraan ohjelmiston kanssa työskentelevät, kuten arkkitehdit, toteuttajat, testaajat ja ylläpitäjät ja näillä kaikilla on omat odotuksensa järjestelmän suhteen.

Markkinointi esimerkiksi voi olla kiinnostunut järjestelmän laadullisten vaatimusten suhteen. Jos tuote esimerkiksi on helposti muunneltavissa, helpottaa se neuvottelua asiakkaiden kanssa. Myöskin katselmoinnissa opittu terminologia helpottaa keskustelua tekniikan puolen ihmisten kanssa.

Kun arkkitehtuuri ymmärretään investointina, kuten tuoterunkoarkkitehtuurin kohdalla, myös liiketoimintajohto voi olla kiinnostunut valvomaan sijoituksen arvoa ja kehitystä.

Myöskin samantyyppisten järjestelmien toteuttajat ja yrityksen tutkimus- ja työkalukehitysorganisaatiot voivat olla kiinnostuneita ainakin arkkitehtuurissa käytetyistä tekniikoista.

Arkkitehtuurin arviointi osana ohjelmistokehitysprosessia

Arkkitehtuurin arvioinnin pitäisi kuulua rutiininomaisesti ohjelmistokehitysprosessiin. Arkkitehtuuridokumentti on ensimmäinen

dokumentti, joka mahdollistaa järjestelmän arkkitehtuurin arviointia ja viathan on paras löytää mahdollisimman aikaisessa vaiheessa, koska tällöin korjaukset eivät vielä tule maksamaan hirveästi. Lisäksi arkkitehtuuri sisältää kriittisimmät suunnitteluratkaisut, joten niidenkin arviointi on tärkeää. Voisi ajatella, että arkkitehtuurin arviointi on järjestelmän perusratkaisujen testausta ennenkuin siirrytään edemmäs yksityiskohtaiseen suunnitteluun. Perusratkaisujen testaamiseen voi käyttää myös prototyyppilähestymistapaa, mutta sen käyttäminen on harvemmin mahdollista aikataulu- ja kustannusyistä. Kalleintahan olisi testata arkkitehtuuri normaalin järjestelmätestauksen kanssa.

Arkkitehtuuria voi arvioida periaatteessa minkä tahansa arkkitehtuurikuvauksen perusteella. Tällöin arvioinnin tulokset ovat sitä luotettavampia kuin kuvaukset ovat tarkkoja. On olemassa ainakin kolme erityyppistä arviointia jotka ovat kiistattomasti hyödyllisiä:

- Arvioidaan heti ensimmäisen arkkitehtuurin luonnoksen jälkeen
- Arvioidaan valmis arkkitehtuurikuvaus
- Arvioidaan valmiin järjestelmän arkkitehtuuri

Kun arvioidaan arkkitehtuurin ensimmäistä luonnosta, kannattaa se tehdä pienimuotoisena, koska luonnoksessa arkkitehtuuri on vielä suhteellisen epätarkasti kuvattu ja siihen ei kannata tuhlata paljoa resursseja. Tärkeintä on analysoida vaatimuksia arkkitehtuurin kannalta. Kun arvioidaan arkkitehtuuria ennenkuin vaatimukset on lyöty lukkoon, voidaan vielä irrottautua mahdottomista vaatimuksista. Tuloksena saadaan realistiset vaatimukset ja alustava arkkitehtuuriratkaisu näiden täyttämiseen.

Kun arvioidaan valmista arkkitehtuurikuvausta, siihen kannattaa käyttää resursseja. Tämä arviointi on hyvä tehdä ennenkuin toteutus aloitetaan, koska arkkitehtuuri voi muuttua tämän arvioinnin tuloksena. Tässä ongelmana on se, että yleensä toteuttaminen on aloitettava jo arkkitehtuurisuunnittelun aikana resurssien varaamisen ja aikataulun takia. Valmiin arkkitehtuurin arviointi on hyödyllistä silloin, kun halutaan lisätä organisaation ymmärrystä järjestelmästä erityisesti laatuattribuuttien suhteen. Esimerkiksi että onko järjestelmä siirrettävissä uuteen ympäristöön ja jos on niin kuinka helposti.

Arviointimenetelmät

Arviointimenetelmät tarjoavat yleensä vastauksia seuraaviin kysymyksiin:

- Sopiiko suunniteltu arkkitehtuuri järjestelmälle?
- Mikä vaihtoehtoisista arkkitehtuureista soveltuu parhaiten järjestelmälle ja miksi ?
- Miten hyvä tulee olemaan järjestelmän jokin tietty laadullinen ominaisuus?

Useat arkkitehtuurin arviointimenetelmät perustuvat skenaariotekniikoihin. Tällöin esitetään konkreettisia esimerkkitilanteita, joissa laatuominaisuudet tulevat esiin. Tämän jälkeen tutkitaan miten arkkitehtuuri soveltuu kyseisiin skenaarioihin. Skenaarioiden etuna on niiden helppo löytäminen, konkreettisuus ja ymmärrettävyys.

Skeenariopohjaisia arviointimenetelmiä ovat mm. :

SAAM (Software Architecture Analysis Method)

- keskittyy erityisesti muunneltavuuteen, siirrettävyyteen, ylläpidettävyyteen
- kehitetty SEI (Software Engineering Institute, Carnegie-Mellon University) :ssä
- perustuu evoluutioaikaisiin skenaarioihin

ATAM (Architecture Tradeoff Analysis Method)

- soveltuu kaikille laatuominaisuuksille
- kehitetty SEI:ssä
- johdettu SAAM:ista

MPM (Maintenance Prediction Method)

- keskittyy ylläpidettävyyteen
- pyrkii löytämään suhteellisen tarkat kustannusarviot ylläpidolle

- Jan Bosch:in kehittämä
- perustuu ylläpitoskenaarioihin

CBAM (Cost Benefit Analysis Method)

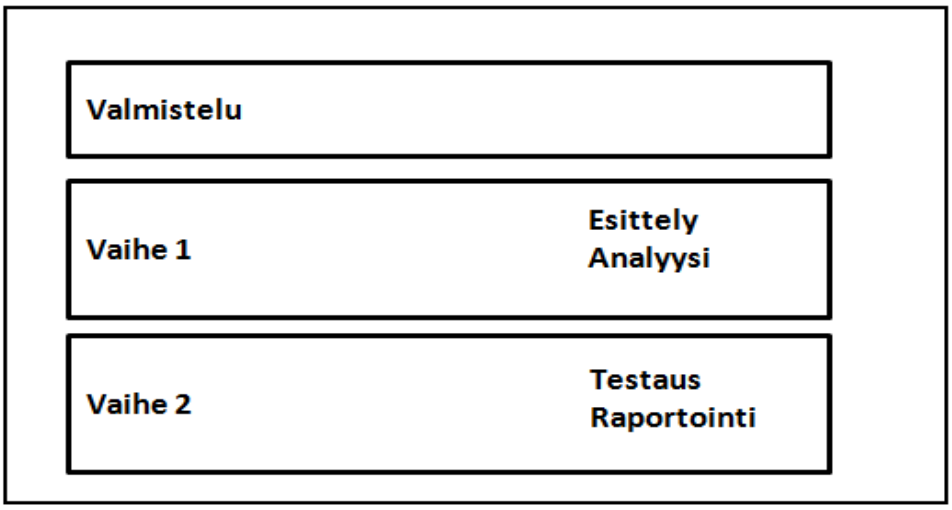
- suuremmille järjestelmille
- johdettu ATAM:ista
- arvioidaan kuinka kalliiksi järjestelmä tulee ja mitä hyötyjä sen tekemisestä on

Muita arviointimenetelmiä ovat esimerkiksi tarkistuslistat ja kysymyslomakkeet, joissa kysytään arkkitehtuurin tai sen suunnitteluprosessiin liittyviä asioita (esim. ”Onko käyttöliittymä erotettu sovelluslogiikasta”). Näitä voidaan yhdistellä skenaariopohjaisten menetelmien kanssa.

ATAM

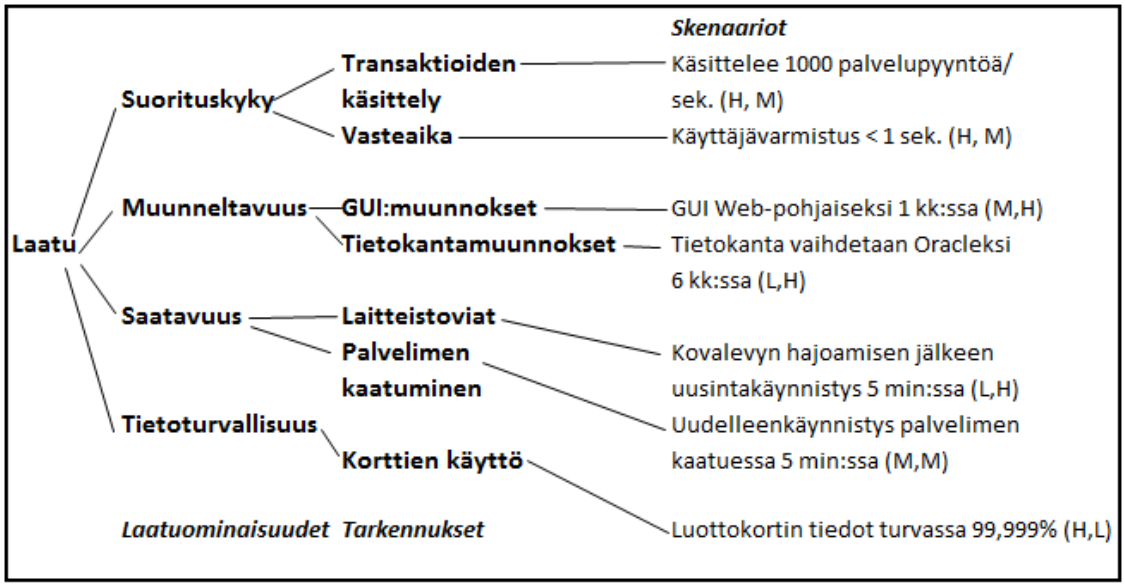
ATAM on arkkitehtuurin arviointiin tarkoitettu menetelmä, jolla voidaan arvioida yleisimpiä laatuominaisuuksia. ATAM jakautuu neljään osioon (Kuva 8.1): esittelyyn, analyysiin, testaukseen ja raportointiin.

Esittelyosiossa kerrotaan kaikille arviointiin osallistuville, mikä ATAM-menetelmä on, mikä rooli kenelläkin on, mitä tekniikoita käytetään ja minkälaisessa muodossa tulokset halutaan. Tämän jälkeen käydään läpi mitkä ovat järjestelmän vaatimukset ja tavoitteet. Tavoitteena on tämän jälkeen, että jokainen ymmärtää järjestelmän toimintaympäristön. Lopuksi kuvataan arkkitehtuuri, tekninen toimintaympäristö ja sen rajapinnan muihin järjestelmiin.



Kuva 8.1 ATAM: in vaiheet (TuTY –kalvot 2008)

Analyysiosiossa tunnistetaan ensimmäisenä arkkitehtuuriratkaisut, joilla on pyritty tiettyjen laatuominaisuuksien täyttämiseen. Jokaiseen ratkaisuun tehdään selvitys siitä, millä tavoin ratkaisu tukee laatuominaisuutta. Seuraavaksi laaditaan laatupuu (Kuva 8.2), jossa täsmennetään laatuominaisuuksia ja liitetään siihen esimerkkitilanteita, joissa kyseinen laatuominaisuus tulee esille. Kukin skenaario painotetaan kahdella parametrilla, kuinka tärkeä skenaario on järjestelmän kannalta, ja toteuttamisen vaikeusaste. Skaalana voi käyttää (L)=low, (M)=medium, (H)=high. Puun perusteella voidaan identifioida riskit, turvalliset ratkaisut, herkkyyskohdat ja tasapainokohdat.



Kuva 8.2 Laatupuu (TuTY –kalvot 2008)

Seuraavana tarkennetaan mitä tarkasti ottaen ovat puun perusteella tehdyt löydökset.

Riski:

Riski = potentiaalisesti ongelmallinen arkkitehtuuriratkaisu, joka voi heikentää jotain laatuominaisuutta

Riski = ratkaisu/fakta + laatuseuraamus + perustelu

Esimerkki:
Kriteerit ja säännöt keskimmäisen kerroksen komponenttien tekemiselle ovat epäselvät (ratkaisu/fakta). Tästä voi seurata toiminnallisuuden replikointia eri kerroksissa (perustelu), mikä heikentää ylläpidettävyyttä(laatuseuraamus).

Turvallinen ratkaisu:

Turvallinen ratkaisu= arkkitehtuuriratkaisu, jolla on tiedossa (lähinnä) vain hyviä laatuseuraamuksia.

Turvallinen ratkaisu = oletus + ratkaisu + laatuseuraamus + perustelu

Esimerkki:
Olettaen, että komponentit eivät joudu tutkimaan toistensa tilaa (oletus), Tarkkailija-suunnittelumallin käyttö komponenttien välisessä kommunikoinnissa (ratkaisu) parantaa muunneltavuutta (laatuseuraamus), koska komponenttien ei tarvitse tietää toisistaan mitään muuta kuin takaisinkutsu- ja rekisteröintirajapinnat (perustelu).

Herkkyyskohta:

Herkkyyskohta = arkkitehtuuriratkaisu, joka on kriittinen jonkin laatuominaisuuden saavuttamisen kannalta.

Esimerkkejä:
MVC-mallin käyttö GUI arkkitehtuurissa on olennaista järjestelmän siirrettävyyden kannalta.

Abstraktin tehtaan käyttö erilaisten ajurien luomiseen on olennaista järjestelmän muunneltavuuden kannalta.

Tasapainokohta:

Tasapainokohta = herkkyyskohta, joka koskee useaa laatuominaisuutta (yleensä vastakkaisiin suuntiin)

Esimerkkejä:
Tila-suunnittelumallin käyttötilakoneen arkkitehtuurissa parantaa muunneltavuutta mutta heikentää suorituskykyä verrattuna suoraan haarautumalauseeseen pohjautuvaan toteutukseen.

XML:n käyttö syöttöformaattina parantaa järjestelmän integroitavuutta mutta heikentää sen suorituskykyä.

Testausosiossa täydennetään ja testataan analyysin tuloksia käyttämällä muiden sidosryhmien tuottamia skenaarioita. Tarkoituksena on herättää kommunikointia eri sidosryhmien välille, jotta saadaan yhteisymmärrys järjestelmän tärkeistä osista ja laatuominaisuuksista. Uudet skenaariot priorisoidaan ja liitetään laatupuuhun ja vanhat skenaariot vahvistetaan. Tämän jälkeen tehdään uusinta-analyysi kierros, jossa tärkeimmät skenaariot tarkistetaan vasten arkkitehtuuria ja mahdolliset uudet riskit tunnistetaan.

Raportointiosiossa tulokset esitellään kaikille jotka osallistuivat arviointiin. Tulokset esitellään myös arviointiraporttina. ATAM –menetelmän tärkeimpänä tehtävänä on identifioida arkkitehtuurin laatuoiminaisuuksiin vaikuttavat ratkaisut käyttäen skenaarioita. Tuloksena saadaan myöskin tietoa, joka helpottaa arkkitehtuurin ymmärtämistä sekä helpottaa riskien hallitsemista. Tuloksena voi olla jopa parannuksia arkkitehtuurin, mutta se ei ole ATAM:n tavoite. Tulokset voidaan koota skenaarioittain siten, että kuhunkin skenaarioon liitetään sitä tukevat arkkitehtuuriratkaisut sekä näihin liittyvät riskikohdat, turvalliset ratkaisut, herkkyyskohdat ja tasapainokohdat listattuna ja numeroituna. Lisäksi skenaarioihin liitetään laatuominaisuudet joita se testaa sekä kuvaus olosuhteista, joissa skenaario tapahtuu.

Arvioinnin hyötyjä ja ongelmia

8.1.1 Hyötyjä

- Taloudelliset: kun riskit tunnistetaan ajoissa tai huomataan, että järjestelmä ei pystykkään siihen mihin on tarkoitus, on halvempaa tehdä korjaukset mahdollisimman alussa.
- Pakotetaan valmistautumaan esittelemään järjestelmän arkkitehtuuri: kun tiedetään, että järjestelmän arkkitehtuuria tullaan arvioimaan, täytyy arkkitehtuurin dokumentointi tehdä.
- Arvioinnin aikana selvitetään perinpohjin, mitä vaatimuksia järjestelmän arkkitehtuurilla on.
- Arkkitehtuurin laadun paraneminen.

Lähteenä: (Bass et al, 2003)

8.1.2 Ongelmia

- Sopivan vaiheen löytäminen, alussa tiedetään liian vähän, toteutusvaiheessa nähdään uhkana aikataululle, toteutuksen jälkeen arviointia pidetään turhana.

Yhteenveto luvusta

- Arviointimenetelmät antavat mahdollisuuden kommunikointiin, mikä pitäisi joka tapauksessa jossain vaiheessa tehdä eri sidosryhmien välillä.

- Arkkitehtuurien arviointia voidaan pitää tavallista syvällisempänä ja systemaattisempana arkkitehtuurin katselmointina.

- Arkkitehtuurin arviointimenetelmät eivät ole kovin täsmällisiä, ne voidaan (ja on syytäkin) räätälöidä yrityskohtaisesti. Esim. ATAM vaatii perusmuodossaan paljon resursseja, mitkä eivät ole aina saatavilla.

KUVAT

TAULUKOT

LÄHTEET

Koskimies, K. & Mikkonen T., P. 2005. Ohjelmistoarkkitehtuurit. Jyväskylä: Talentum Media Oy.

Haikala, I & Märijärvi J., P. 2004. Ohjelmistotuotanto. Hämeenlinna: Talentum Media Oy.

Gamma, E. & Helm, R. & Johnson, R. & Viissides, J., P. 2001. Olio-ohjelmointi – Suunnittelumallit. Helsinki: Oy Edita Ab.

Bass, L. & Clements, P. & Kazman, R. P. 2003. Software Architecture in Practice. Boston: Addison-Wesley.

Pressman R.S. (Adapted by Darrel Ince), P. 2000. Software engineering: a practitioner's approach: European Adaption. London: McGraw-Hill.

HY-kalvot 2008, Helsingin Yliopiston syksy 2008 ohjelmistoarkkitehtuurit -kurssin materiaalit, http://www.cs.helsinki.fi/u/gustafss/arkkiS08/index.html .

TuTY-kalvot, Turun Teknillisen Yliopiston 2008 ohjelmistoarkkitehtuurit -kurssin materiaalit, http://www.cs.tut.fi/kurssit/OHJ-3200/luennot/ .

TaTY –kalvot, Tampereen Teknillisen Yliopiston 2006 ohjelmistoarkkitehtuurit –kurssin materiaalit, http://www.pori.tut.fi/~hj/for-guests/public/ohs/ .

Wikipedia,Tietojärjestelmäarkkitehtuuri,
http://fi.wikipedia.org/wiki/Tietojärjestelmäarkkitehtuuri (Luettu 10.07.2009)

IEEE 1471-2000, Recommended Practice for Architectural Description for Software- Intensive Systems, http://standards.ieee.org/cgi-bin/status?1471-2000 (Luettu 10.07.2009)

Kruchten 1995, 4 + 1 –malli,
http://en.wikipedia.org/wiki/4%2B1 (Luettu 20.07.2009)

API, Ohjelmointirajapinta,
http://fi.wikipedia.org/wiki/Ohjelmointirajapinta (Luettu 23.07.2009)

OMG, Object Management Group,
http://www.omg.org/ (Luettu 23.07.2009)

MDA, Model-driven architecture,
http://www.omg.org/mda/ (Luettu 23.07.2009)

ATAM, Architecture Tradeoff Analysis Method,
http://www.sei.cmu.edu/architecture/ata_method.html (Luettu 27.08.2009)

SAAM, Software Architecture Analysis Method,
http://www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html
(Luettu 27.08.2009)

CBAM, Cost Benefit Analysis Method,
http://www.sei.cmu.edu/architecture/products_services/cbam.html
(Luettu 27.08.2009)