Tuomas Kommeri

# ASPEKTIKESKEINEN OHJELMISTOKEHITYS – JOHDATUS ELINKAAREN VAIHEISIIN

Tietojärjestelmätieteen
kandidaatintutkielma
25.2.2009

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

# TIIVISTELMÄ

Kommeri, Tuomas Pekka Viljami
Tietojärjestelmätieteen kandidaatintutkielma
Jyväskylä: Jyväskylän yliopisto, 2009, 31 s.

Tämän tutkielman aihepiirinä on aspektikeskeinen ohjelmistojen kehittäminen. Tutkielmassa kuvaillaan, mitä hyötyä kyseisen lähestymistavan mukaisesta ohjelmistokehityksestä on, ja millä tavalla sen tarjoamilla ratkaisuilla on mahdollista parantaa ohjelmistojen laatua.

Tutkielma antaa yleiskuvan aspektikeskeisen ohjelmistokehityksen periaatteista ja esittelee elinkaaren eri vaiheisiin kehitettyjä lähestymistapoja. Yksityiskohtaisia tai monimutkaisia ohjelmistokehitysprosessin rakenteita ei tämän tutkielman yhteydessä esitellä. Koska tutkimusala on vielä nuori ja suhteellisen tuntematon, perusteet ja lähtökohdat on pyritty selittämään useita esimerkkejä hyödyntäen. Tutkimus suoritetaan kirjallisuuskatsauksena.

Tutkimustuloksena esitellään kartoitus aspektikeskeisen ohjelmiston vaatimusmäärittelyyn, analyysiin ja suunnitteluun liittyvistä lähestymistavoista. Lisäksi aspektiohjelman toteutusta ja sen rakenteita on havainnollistettu ohjelmakooditasolle viedyillä esimerkeillä.

AVAINSANAT: tutkielma, aspektikeskeinen, aspekti, menetelmä, lähestymistapa, tietojärjestelmä, ohjelmistokehitys, järjestelmäkehitys

Ohjaaja: Pekka Makkonen

Tietojenkäsittelytieteiden laitos

Jyväskylän yliopisto

Tarkastaja: Pekka Makkonen

Tietojenkäsittelytieteiden laitos

Jyväskylän yliopisto

# SISÄLLYSLUETTELO

# 1 JOHDANTO

Järjestelmiä suunniteltaessa on otettava huomioon useita eri tekijöitä sekä tasapainoiltava monien hankalien kysymysten kanssa: kuinka järjestelmässä toteutetaan haluttu toiminnallisuus, millä tavalla saavutetaan suorituskyky ja luotettavuus, kuinka käsitellään ohjelmistoalustaan liittyvät kysymykset jne. Saattaa olla, että luokkien, operaatioiden tai proseduurien täytyy suorittaa useita toimintoja, mikä voi johtaa ”spagettikoodiin”. Tämä kielii heikosta ohjelman suunnittelusta, joten siihen on panostettava enemmän. Tämä tapahtuu edistämällä ohjelman modulaarisuutta ja hankkimalla parempi kohteiden erillisyys (engl. *separation of concerns*). Jokaisen komponentin, luokan ja operaation tulee olla kohdistettuna tiettyyn tarkoitukseensa. (Jacobson & Ng 2005).

Olemassa olevilla tekniikoilla on kuitenkin rajoituksensa. Vaikka suunnittelussa mentäisiin kuinka pitkälle tahansa, useassa järjestelmän osassa esiintyy ohjelmakoodin pirstoutumista. Mm. lokien kirjoitus, todentaminen, tiedon pysyvyys, perkaus (engl. *debugging*), jäljitys, hajautus ja virheiden käsittely ovat asioita, joilla on tekemistä koodin pirstoutumisen kanssa. Toisinaan suurehkolla osuudella operaatiosta tai luokasta ei ole mitään tekemistä sen kanssa, mitä sen pitäisi tehdä. (Jacobson & Ng 2005).

Myös Kiczalesin ym. (1997) mukaan useiden ohjelmiston suunnitteluvaiheessa tehtävien päätösten toteuttamiseen liittyy ongelmia, joita proseduraalisella tai oliolähestymistavalla ei pystytä selkeästi ratkaisemaan. Tällaisia suunnittelupäätöksiä toteutettaessa ohjelmakoodi hajautuu ohjelmiston eri osiin, johtaen sekavaan koodiin, jota on kohtuuttoman hankalaa kehittää sekä ylläpitää. Edellä mainitun kaltaisten päätösten aiheita voidaan kutsua aspekteiksi. Niiden hankala ilmaiseminen ohjelmakoodin tasolla johtuu siitä, että ne läpileikkaavat (engl. *cross-cut*) ohjelman perustoiminnallisuuden. Aspektikeskeisellä (engl. *aspect-oriented*) ohjelmoinnilla on mahdollista ilmaista

aspekteja sisältävät ohjelmat selkeästi. Aspektikeskeinen lähestymistapa tarjoaa mekanismin muodostaa läpileikkaava toiminta halutuiksi operaatioiksi ja luokiksi ohjelman kääntämisen - tai jopa suorittamisen aikana.

Shakerin ja Petersin (2005) mukaan moderni ohjelmistokehitys on enemmän kuin ohjelmakoodin kirjoittamista; se on iteratiivinen prosessi, joka koostuu useista vaiheista. Menetelmät ohjaavat vaiheita ja ohjelmointikielet ovat ainoastaan kehitysprosessin työkaluja. Kuitenkin perinteisesti suurimmat edistysaskeleet ovat saaneet alkunsa tältä tasolta ja laajentuneet koskemaan koko kehitysprosessia. Aspektikeskeinen teknologia ei ole poikkeus; aspektikeskeisten ohjelmointikielten kehityksen myötä aspektikeskeisyyden idea on laajentunut koskemaan koko kehitysprosessia, muodostaen *aspektikeskeisen ohjelmistojen kehityksen* (engl. *aspect-oriented software development*) tutkimusalan. Jacobsonin ja Ngn (2005) mukaan aspektikeskeisen ohjelmistokehityksen tarkoitus on parantaa koko ohjelmiston modulaarisuutta. Tämä koskee niin toiminnallisia kuin ei-toiminnallisiakin vaatimuksia, alustan ominaisuuksia jne. Kohteiden ilmaiseminen toisistaan erillään luo mahdollisuuden rakentaa järjestelmiä, joissa on ymmärrettävämpi rakenne ja jotka ovat helpommin laajennettavissa uusia vaatimuksia vastaaviksi.

Tutkimuksen tavoite on selvittää, mitä aspektikeskeisellä ohjelmistokehityksellä tarkoitetaan ja kartoittaa aspektikeskeisen ohjelmistokehitysprosessin eri lähestymistapoja. Tavoitteena on antaa yleiskuva lähestymistavoista sekä esitellä myös aspektiohjelmointia. Tutkimusongelma voidaan muotoilla seuraavasti: Mitä tarkoitetaan aspektikeskeisellä ohjelmistokehityksellä ja mitä lähestymistapoja sen elinkaaren eri vaiheisiin on kehitetty?

Tutkimuksen tuloksista on hyötyä mm. ohjelmistokehityksen parissa työskenteleville henkilöille. Se tarjoaa yleiskuvan uudenlaisesta ohjelmistokehityksen lähestymistavasta, jolla pyritään parantamaan mm. ohjelmistojen ylläpidettävyyttä, laajennettavuutta ja uudelleenkäyttöä. Näin

ollen myös ohjelmistojen kehittämisen ja ylläpidon kustannuksia on mahdollista vähentää.

Johdannon jälkeisessä luvussa esitellään keskeiset käsitteet sekä aspektikeskeisen ohjelmistokehityksen perusperiaatteet. Kolmannessa luvussa esitellään elinkaaren eri vaiheisiin kehitettyjä lähestymistapoja. Neljännessä luvussa tehdään yhteenveto tutkielman sisällöstä.

# 2 ASPEKTIKESKEINEN OHJELMISTOJEN KEHITTÄMINEN

## 2.1 Käsitteet

Foxin (2005) mukaan sanan *aspekti* merkitys voidaan määritellä tietyksi näkökulmaksi, josta jotakin asiaa tarkastellaan. Sana on peräisin latinankielisestä sanasta *aspectus*, joka tarkoittaa *katsella jotakin*. Näin ollen sanan aspekti valintaa voidaan pitää johdonmukaisena, kun ajatellaan ohjelmiston koostuvan eri osista tai näkökulmista ja osat on mahdollista erotella kiinnostuksen *kohteiden* mukaan.

Jacobson (2003) määrittelee aspektin läpileikkaavan toteutuksen käsittäväksi modulaariseksi yksiköksi. Viegan ja Voasin (2000) mukaan aspektit ovat eräällä tavalla vastakohta perityille luokille. Periytymisessä luokat valitsevat, mitä toiminnallisuuksia liitetään muista objekteista. Aspektit sen sijaan valitsevat, mitä toiminnallisuuksia muihin objekteihin liitetään. Luokkien tapaan aspektit sisältävät toiminnallisuutta, mutta tärkein ero luokkiin on siinä, että aspektien tarkoitus on ottaa haltuun ohjelman läpileikkaavat kohteet (engl. *cross-cutting concerns)*.

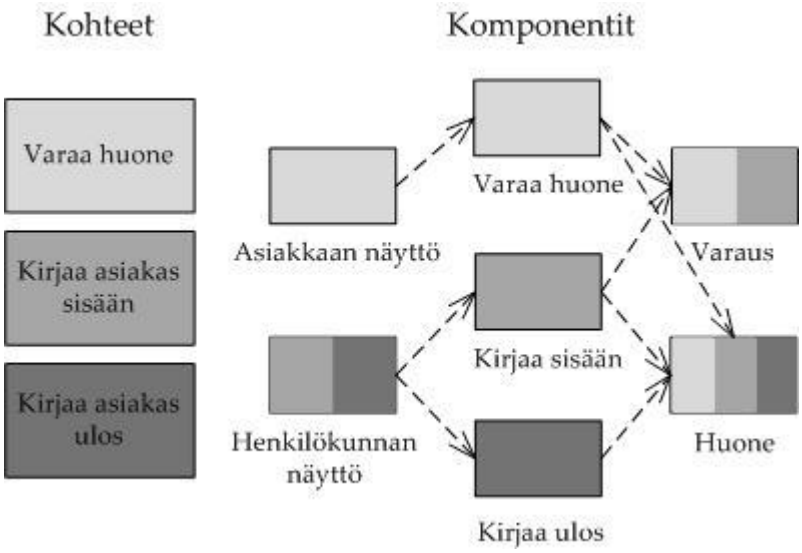
Hilliard (1999) määrittelee sanan *concern* seuraavasti: "A concern expresses a specific interest in some topic pertaining to a particular system of interest (or other subject matter)". Käytän tässä tutkielmassa suomenkielisenä vastineena sanaa *kohde*. Myös Tietäväinen (2006) on päätynyt tähän sanaan, joskaan sitä ei voi pitää kattavana suomennoksena. Jacobsonin ja Ngn (2005) mukaan kohde on mitä tahansa, joka on joidenkin ohjelmistoprojektiin osallisten tahojen (esim. loppukäyttäjän, projektin sponsorin tai jopa kehittäjän) mielenkiinnon kohteena. Kohde voi olla esimerkiksi toiminnallinen vaatimus, ei-toiminnallinen vaatimus tai järjestelmän suunnittelurajoitus. Se voi myös olla enemmän kuin järjestelmän vaatimus tai niinkin tarkoin määrätty asia kuin

välimuisti tai puskurointi. Tarkkaa ja yksiselitteistä määritelmää kohteelle on vaikeaa keksiä.

Dijkstran (1982) mukaan kohteiden ilmaiseminen erillään on ainoa tehokas tekniikka käsitysten järjestämiseen, vaikkei se olisikaan täydellisesti mahdollista. Tällä hän tarkoittaa tietyn tahon huomion keskittymistä johonkin aspektiin. Se ei tarkoita muiden aspektien huomioimatta jättämistä vaan, että huomion kohteena olevan aspektin näkökulmasta kaikki muu on epäolennaista. Jacobsonin ja Ngn (2005) mukaan ideaalitilanteessa erilaiset kohteet eroteltaisiin moduuleihin ja jokaista kehitettäisiin erillään, yksi kerrallaan. Tämän jälkeen moduuleista koottaisiin valmis järjestelmä. Näin ollen kohteet jaotellaan moduuleihin ja jokainen moduuli ratkaisee tai toteuttaa erillisen osan kohteista. Vaikka jotkut kohteet on mahdollista toteuttaa erillisinä ja erotella komponentteihin, on olemassa useita kohteita, jotka vaikuttavat moniin komponentteihin. Tällöin on kysymys läpileikkaavasta kohteesta.

## 2.2 Komponenttien rajoitteet kohteiden toteuttamisessa

Jacobson ja Ng (2005) esittelevät pelkistetyn komponenttipohjaisen esimerkin, jossa kohteita ei pystytä ilmaisemaan toisistaan erillisinä (kuva 1). Esimerkissä on kuvattu hotellin varausjärjestelmä, jossa seitsemän komponentin avulla toteutetaan kolme haluttua kohdetta. Esimerkiksi Huone-komponentti on osallisena jokaisen kohteen toteutuksessa. Ohjelmakoodin sekoittumisen (engl. *tangling*) lisäksi voidaan havaita, että tietty kohde on pirstaloituneena (engl. *scattered*) useaan eri komponenttiin. Esimerkiksi kohde Kirjaa asiakas sisään käyttää hyväkseen neljää eri komponenttia. Näin ollen, jos kyseisen kohteen vaatimukset muuttuvat, eri luokat on myös päivitettävä. Tällä tavalla ohjelmiston ylläpito muuttuu hankalammaksi.




*Kuva 1 Sekoittuminen ja pirstaloituminen kohteita toteutettaessa (Jacobson & Ng, 2005)*

### 2.3 Kohteiden toteuttaminen aspektien avulla

Kiczalesin ym. (1997) mukaan järjestelmään toteutettava ominaisuus on komponentti, jos se voidaan puhtaasti kapseloida esimerkiksi olioksi, metodiksi, proseduuriksi tai sovellusliittymäksi. Puhtaalla kapseloinnilla tarkoitetaan sitä, että ominaisuus on hyvin paikallistettava (engl. *well-localized*), siihen pääseminen käsiksi on helppoa (engl. *easily accessed)* ja se on helposti muodostettavissa (engl. *composed*). Jos taas ominaisuutta ei voida tällä tavalla puhtaasti kapseloida, se tulee toteuttaa aspektina.

Kuvassa 2 esitetään edellisessä kohdassa esitellyn varausjärjestelmän kohteet ja niiden toteutuksen hajautuminen luokkien kesken. Huoneen varaamiseksi on ensin tarkistettava, onko se vapaana ja jos on, niin tämän jälkeen voidaan luoda uusi varaus. Asiakkaan kirjaamiseksi sisään hänet sijoitetaan huoneeseen ja poistetaan varaus. Samanaikaisesti luodaan Maksu-luokassa uusi lasku. Asiakkaan kirjaamiseksi ulos veloitetaan maksu, jonka jälkeen asiakas poistetaan huoneesta. (Jacobson & Ng 2005). Kuvasta 2 nähdään, kuinka

toiminnot läpileikkaavat järjestelmän luokat. Kohteet eivät tässä tapauksessa siis kapseloidu erillisiin komponentteihinsa, joten ne toteutetaan aspekteina.

| | Huone | Varaus | Maksu |
|---|---|---|---|
| Varaa huone | tarkistaSaatavuus() | luo() | |
| Kirjaa asiakas sisään | sijoitaAsiakas() | poista() | luoLasku() |
| Kirjaa asiakas ulos | poistaAsiakas() | | maksaLasku() |

*Kuva 2 Kohteista muodostetut luokat (Jacobson & Ng, 2005)*

Kuvassa 3 esitetään yksinkertaistettu esimerkki Kirjaa asiakas sisään –kohteen toteutuksesta AspectJ-kielellä (Java-pohjainen ohjelmointikieli aspektiohjelmointiin). Ensimmäisellä rivillä kohde julistetaan aspektiksi aspect-sanalla. Tämän jälkeen toiminnallisuudet lisätään haluttuihin luokkiin. Tämä tapahtuu kirjoittamalla ensin olemassa olevan luokan nimi ja tämän jälkeen operaatio, joka luokkaan halutaan lisätä. Esimerkiksi rivillä 3 Huone-luokkaan lisätään operaatio sijoitaAsiakas(). Vaikka operaatio on siis osa Huone-luokkaa, se määritellään muualla kuin luokassa itsessään. Tämä antaa mahdollisuuden organisoida ohjelmakoodia niin, että kohteet on mahdollista toteuttaa toisistaan erillään. (Jacobson & Ng 2005). Aspektiohjelmointi tarjoaa muitakin mahdollisuuksia, kuin vain operaatioiden lisäämisen olemassa oleviin luokkiin. Myöhemmin esitellään, mitä ovat ns. liitoskohdat ja kuinka niihin lisätään toiminnallisuutta.

```
1.      public aspect KirjaaAsiakasSisaan {
2.              ...
3.              public void Huone.sijoitaAsiakas()
4.              {
5.                      // koodia asiakkaan kirjaamiseksi
6.              }
7.              public void Varaus.poista()
8.              {
9.                      // koodia varauksen poistamiseksi
10.             }
11.             public void Maksu.luoLasku()
12.             {
13.                     // koodia laskun luomiseksi
14.             }
15.             ...
16.     }
```

*Kuva 3 Kohteen toteutus aspektiohjelmoinnilla (Jacobson & Ng, 2005)*

# 3 ELINKAAREN VAIHEET

Tässä kappaleessa esitellään aspektikeskeisen ohjelmistokehityksen eri vaiheita vaatimusmäärittelystä ohjelmointiin. Tutkielman laajuus huomioon ottaen, kaikkia elinkaaren vaiheita tai lähestymistapoja ei kuitenkaan ole mahdollista esitellä. Esimerkiksi aspektikeskeisen arkkitehtuurin suunnitteluun on kehitetty useita lähestymistapoja, joita ei tämän tutkielman puitteissa esitellä. Mm. Chitchyan ym. (2005) ovat esitelleet näitä lähestymistapoja tutkimuksessaan.

Aspektikeskeisen ohjelmistokehityksen tarkoitus on tarjota teknologia, jolla läpileikkaavat kohteet saadaan pidettyä erillään vaatimusmäärittelystä lähtien. Kohteiden pitämiseksi erillään ne on mallinnettava ja jäsennettävä. Erillisyyden säilyttämiseksi suunnittelusta toteutukseen asti, tarvitaan siihen soveltuvia tekniikoita. Lisäksi tarvitaan mekanismi, jolla kohteet saadaan muodostettua yhtenäiseksi kokonaisuudeksi. (Jacobson & Ng 2005). Hannemanin ym. (2003) mukaan iteratiivinen ohjelmistokehitysprosessi soveltuu hyvin aspektikeskeiseen ohjelmistokehitykseen. Kehitysprosessin alkuvaiheessa on olemassa vain rajoitetut keinot kaikkien merkityksellisten läpileikkaavien kohteiden tunnistamiseksi. Tunnistettaessa uusi läpileikkaava kohde myöhemmissä kehitysvaiheissa, ei-iteratiivinen prosessi vaatisi mm. kauttaaltaan muutoksia tehtyihin suunnitelurakennelmiin.

## 3.1 Aspektikeskeinen vaatimusmäärittely

Monet nykyiset vaatimusmäärittelyn mallit tarjoavat tuen erityyppisten vaatimusten tunnistamiseen ja käsittelyyn, mutta ne eivät keskity läpileikkaaviin vaatimuksiin. Tällaiset mallit ovat kuitenkin usein pohjana aspektikeskeisen vaatimusmäärittelyn (engl. *aspect-oriented requirements engineering, AORE*) malleille. Voidaankin sanoa, että aspektikeskeiset vaatimusmäärittelyn mallit täydentävät aiemmin kehitettyjä malleja tarjoamalla systemaattisen tavan käsitellä läpileikkaavia kohteita. Aspektikeskeinen

vaatimusmäärittely ei kuitenkaan tarjoa ainoastaan edistynyttä tukea läpileikkaavien toiminnallisten ja ei-toiminnallisten ominaisuuksien erottamiselle. Se tarjoaa myös paremmat keinot tunnistaa ja käsitellä tällaisten ominaisuuksien sekoittuneisiin kuvauksiin liittyvät ristiriidat. (Chitchyan ym. 2005).

Taulukossa 1 esitellään aspektikeskeisen vaatimusmäärittelyn lähestymistapoja Chitchyanin ym. (2005) tutkimukseen perustuen. Siinä tärkeimmät lähestymistavat on jaettu eri ryhmiin niiden luonteen mukaan. Kuten edellisessä kappaleessa mainittiin, useimmat näistä lähestymistavoista perustuvat aiemmin kehitettyihin malleihin, joita ei siis ole suunniteltu aspektikeskeistä vaatimusmäärittelyä varten. Esimerkiksi näkökulmiin perustuva aspektilähestymistapa AORE with Arcade sisältää yhtäläisyyksiä suhteellisen tunnetun PREview-mallin kanssa. Lisäksi mm. perinteisen käyttötapauslähtöisen vaatimusmäärittelyn prosessi on hyvin samankaltainen kuin aspektikeskeisessä vastineessa (AOSD with Use Cases). Tärkeimpänä erona näiden mallien välillä on se, että aspektikeskeisessä versiossa on erilliset käyttötapaukset ei-toiminnallisille vaatimuksille (joita yleisesti pidetään läpileikkaavina).

*Taulukko 1 Aspektikeskeisiä lähestymistapoja vaatimusmäärittelyyn*

| Lähestymistapa | Ryhmä |
|---|---|
| AORE with Arcade | Viewpoint-Based |
| ARGM | Goal-Oriented |
| AOSD with Use Cases | Use Case / Scenario -Based |
| Scenario Modelling with Aspects | Use Case / Scenario -Based |
| Aspectual Use Case Driven Approach | Use Case / Scenario -Based |
| Cosmos | Multi-Dimensional Separation of Concerns |

| | |
|---|---|
| CORE | Multi-Dimensional Separation of Concerns |
| AOCRE | Aspect-Oriented Component-Based |
| Theme/Doc | Other |

Useat edellä esiteltyjen lähestymistapojen tuottamat mallit ovat jäljitettävissä seuraavaan elinkaaren vaiheeseen. On siis olemassa lähestymistapoja, jotka soveltuvat käsittelemään tässä vaiheessa tuotettua materiaalia. Muutamat lähestymistavoista ovat osana menetelmiä, joiden tarkoituksena on tarjota elinkaaren vaiheet kattava ohjeistus vaatimusmäärittelystä toteutukseen asti. Tällaisista menetelmistä mainittakoon AOSD with Use Cases (Jacobson & Ng 2005) sekä Theme (Clarke & Baniassad 2005). Lisäksi ainakin Grundyn (1999) esittelemä komponenttitekniikkaan perustuva AOCRE ja AOCE (Grundy 2000) ovat yhteensopivia toistensa kanssa, jolloin nämä yhdessä muodostavat vaatimusmäärittelystä toteutukseen asti ulottuvan menetelmän. Seuraavaksi, analyysin ja suunnittelun yhteydessä, esitellään näitä sekä muutamia muita lähestymistapoja.

### 3.2 Aspektikeskeinen analyysi ja suunnittelu

Kirjallisuudesta löytyy useita lähestymistapoja, jotka tarjoavat tuen läpileikkaaville kohteille suunnittelutasolla. Useimmat tällaisista lähestymistavoista on kehitetty vahvistamaan oliokeskeisiä suunnitteluteknikoita ja UML:ää siten, että läpileikkaavien kohteiden modularisointi on mahdollista. Blairin ym. (2004) mukaan eräitä avainlähestymistavoista ovat koostemallit (engl. *Composition Patterns*) ja niiden seuraaja Theme/UML, aspektikeskeinen komponenttitekniikka (engl. *Aspect-Oriented Component Engineering, AOCE*), Hyperspaces, Architectural Views, sekä Suzuki & Yamamoton malli (UXF). Lisäksi mm. Chitchyan ym. (2005) esittelevät tutkimuksessaan useita lähestymistapoja, joista mainittakoon

aspektikeskeinen suunnittelumalli (engl. *Aspect-Oriented Design Model, AODM*), käyttötapauslähtöinen lähestymistapa (engl. *Aspect-Oriented Software Development with Use Cases*) sekä aspektikeskeinen arkkitehtuurimallinnus (engl. *Aspect-Oriented Architecture Modelling, AAM*). Myös Schauerhuber ym. (2006) ovat esitelleet näitä sekä muutamaa muuta lähestymistapaa tutkimuksessaan. Eräänä mainittakoon muutamassa käytännön projektissakin testattu *The JAC Design Notation*, joka on pääasiassa suunniteltu avoimen lähdekoodin JAC (Java Aspect Components) –kehystä varten. Lisäksi huomattakoon, että myös mm. ketteriä menetelmiä varten on kehitetty aspektikeskeinen lähestymistapa; eXtreme AOCE (Singh ym. 2005). Taulukossa 2 esitellään joitakin lähestymistapojen piirteitä Chitchyanin ym. (2005), Schauerhuberin ym. (2006) ja Blairin ym. (2004) mukaisesti.

Skaalautuvuudella tarkoitetaan lähestymistavan kykyä käsitellä niin laajojen, kuin pienempienkin järjestelmien suunnittelua. Ulkoisella jäljitettävyydellä tarkoitetaan kahden mallin välisen suhteen ominaisuutta, jolla mitataan läpinäkyvyyttä tai mallien jalostusprosessin selkeyttä. Esimerkiksi Theme-lähestymistapa tukee jäljitettävyyttä läpi kehitysprosessin vaatimusmäärittelystä toteutukseen (R->I). Se tarjoaa siis selkeän ohjeistuksen vaatimusten ottamiseen ja niiden järjestämiseen suunnittelua varten. Lisäksi lähestymistapa tarjoaa ohjeet, joilla vaatimukset tuotetaan suunnittelumalleiksi ja kuinka tällä tavalla tuotetut mallit saadaan toteutettua. (Chitchyan ym. 2005). Kypsyyden kriteereitä ovat mallinnusesimerkit, toteutetut sovellukset, lähestymistapojen esiintyminen julkaisuissa sekä viimeisen julkaisun vuosi. Työkalutuki osoittaa sen, tarjoaako lähestymistapa työkaluja mallinnukselle, muodostamiselle tai koodin tuottamiselle. Esimerkiksi JAC-kehys tarjoaa valmiin IDE-työkalun, joka tukee mallinnusta ja ohjelmakoodin tuottamista. (Schauerhuber ym. 2006).

*Taulukko 2 Aspektikeskeisten lähestymistapojen piirteitä*

| Lähestymistapa | Skaalautuvuus | Jäljitettävyys (ulkoinen) | Kypsyys | Työkalutuki |
|---|---|---|---|---|
| Theme | Hyvä, esimerkein osoitettu | R->I | Hyvä, ei varmuutta käytännön projekteista | Ei tuettu |
| AOCE | Hyvä, esimerkein osoitettu | R->I | Ei tiedossa | Ei tiedossa |
| Architectural Views | Hyvä | Ei tiedossa | Ei tiedossa | Ei tiedossa |
| Suzuki & Yamamoto | Ei tiedossa | D->I | Ei tiedossa | Ei tiedossa |
| The Motorola Weavr | Hyvä | Ei tuettu | Keskinkertai-nen, ollut tuotantokäytös-sä Motorola-yhtiössä | Työkalutuki mallinnukselle, muodostamiselle ja koodin tuottamiselle |
| AODM | Osittain tuettu | D->I | Keskinkertai-nen, ei käytän-nön projekteja | Työkalutuki mallinnukselle |
| AOSD with Use Cases | Hyvä, esimerkein osoitettu | R->I | Keskinkertai-nen, ei varmuutta käytännön projekteista | Ei tuettu |
| AAM | Hyvä | Ei tuettu | Hyvä, ei käytännön projekteja | Työkalutuki mallinnukselle ja muodostamiselle |
| JAC | Hyvä | D->I | Keskinkertai-nen, testattu muutamassa käytännön projektissa | Työkalutuki mallinnukselle ja koodin tuottamiselle |

Seuraavaksi esitellään muutamaa edellä mainituista lähestymistavoista lyhyesti. Koostemalleihin perustuva Theme-lähestymistapa sisältää erillisten

*teemojen* tunnistamisen ja suunnittelun, jotka myöhemmin yhdistetään kokonaiseksi järjestelmäksi. Teemat voidaan ajatella samankaltaisiksi ominaisuuksien tai kohteiden kanssa. Tätä lähestymistapaa hyödyntämällä jokainen järjestelmän ominaisuus voidaan suunnitella erillisenä, jonka jälkeen lähestymistapa tarjoaa keinot niiden yhdistämiseen. Lähestymistapa auttaa tunnistamaan teemoista ne, jotka ovat läpileikkaavia sekä ne, joilla ei ole tällaista luonnetta. Mikä tahansa kohde, oli se läpileikkaava tai ei, voidaan siis kapseloida yhden teeman sisään. Teema on eräänlainen paketti, jonka sisällä luokkakaavio kuvaa kohteen rakennetta ja sekvenssikaavio käyttäytymistä. Sekvenssikaaviota käytetään läpileikkaavissa teemoissa eli ns. *aspektiteemoissa* osoittamaan missä ja kuinka tällainen teema läpileikkaa toisen teeman käyttäytymistä. Lähestymistapa sisältää Theme/Doc –esityksen, jota käytetään vaatimusten analysointiin sekä Theme/UML:n, jota sovelletaan suunnitteluun. Lähestymistapaa sovellettaessa on suoritettava kolme päätoimintoa, jotka ovat analyysi, suunnittelu sekä muodostaminen. Analyysissä suoritetaan siis vaatimusten analysointi teemojen tunnistamiseksi. Suunnitteluvaiheessa teemat suunnitellaan toisistaan erillisinä ko. lähestymistavan mukaisella tekniikalla. Muodostamisessa määritetään, kuinka Theme/UML –mallit tulee yhdistää. (Clarke & Baniassad 2005).

Aspektikeskeinen komponenttitekniikka (AOCE) pyrkii tuottamaan uudelleenkäytettävämpiä, laajennettavempia sekä dynaamisesti joustavia ohjelmistokomponentteja. Periaatteena on perinteisestä komponenttitekniikasta poiketen hallita horisontaalinen, läpileikkaava toiminta kun tavallisesti komponentit on suunniteltu ja toteutettu hallitsemaan järjestelmän ”vertikaalit viipaleet”, jakaen järjestelmät datan ja sen operaatioiden mukaan ryhmitettyihin palveluihin. Aspektikeskeinen komponenttitekniikka keskittyy tunnistamaan järjestelmän erilaiset ”horisontaaliset viipaleet”, eli käytännössä aspektit, joille komponentti tarjoaa palveluita tai palvelut, jotka se vaatii muilta komponenteilta. Tavallisesti jokainen järjestelmän komponentti tarjoaa yhden

tai useamman aspektipalvelun (engl. *aspect-related service*) muiden komponenttien käytettäväksi ja vaatii yhden tai useamman aspektipalvelun muilta komponenteilta toimiakseen. (Grundy 2000).

Kehitettäessä järjestelmää käyttötapauslähtöisesti (AOSD with Use Cases), analyysivaiheessa saadaan rakennettua pohja luokkien erottamiselle niille kuuluviin kerroksiin sekä paketteihin ja lopulta viipaleisiin sekä moduuleihin. Käyttötapausten pitäminen erillään analyysistä toteutukseen asti koostuu seuraavista neljästä vaiheesta:

1. Tarkastele käyttötapausten määrittelyä ja päivitä, mikäli se on tarpeellista.

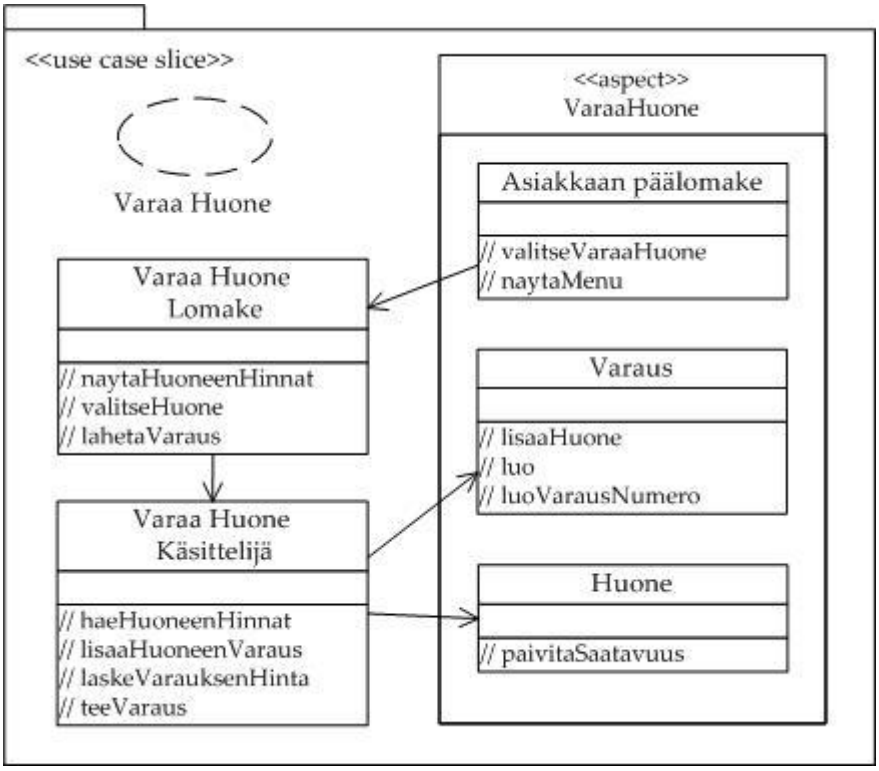
2. Analysoi käyttötapaus tunnistamalla ne analyysivaiheen luokat, jotka ovat osallisia kyseisen käyttötapauksen toteuttamisessa sekä kohdenna näiden käyttötapausten toiminta analyysivaiheen luokille.

3. Organisoi luokat kerroksiksi ja paketeiksi elementtirakenteessa ja sekä luokat, että luokkien laajennokset viipaleiksi käyttötapausrakenteessa.

4. Suunnittele käyttötapaus kuvaamalla analyysielementit suunnitteluelementeiksi, tunnistamalla rajapinnat ja täydentävät suunnitteluelementit ja tämän jälkeen jalostamalla näitä elementtejä.

Käyttötapaukseen osallistuvien luokkien tunnistamisen jälkeen on kuvailta kuinka niiden ilmentymät toimivat yhdessä. Tämä tapahtuu käymällä läpi käyttötapauksen määrityksessä kuvatut vaiheet ja kuvailemalla, kuinka osallistuvien luokkien ilmentymät vaikuttavat toisiinsa kussakin vaiheessa. Vuorovaikutus on mahdollista kuvata esimerkiksi UML:n kommunikaatio-, vuorovaikutus- ja luokkakaavioiden avulla. Seuraavassa vaiheessa luokat lajitellaan paketteihin niiden käyttäytymisen ja ominaisuuksien mukaan. Lisäksi käyttötapaukset tulee sijoittaa kukin omaan käyttötapausviipaleeseensa (engl. *Use Case Slice*), joka sisältää kyseisen käyttötapauksen toteuttamiseen

liittyvät luokat sekä niiden laajennokset. Käyttötapausviipaleet kokoavat siis osia luokista, operaatioita jne. Tämän jälkeen käyttötapausviipale sisältää ainoastaan ne asiat, jotka ovat tietylle käyttötapaukselle ominaisia. Käyttötapausrakenne sisältää myös viipaleita, jotka eivät ole käyttötapaus-spesifejä (engl. *non-use-case-specific slices*). Kuvassa 5 esitellään toisessa luvussa esiteltyyn hotellin varausjärjestelmään liittyvä käyttötapausviipale, jonka tehtävänä on pitää Varaa Huone –käyttötapaus erillisenä. Se on siis eräänlainen paketti, joka sisältää Varaa Huone –käyttötapaukselle ominaiset luokat VaraaHuoneLomake ja VaraaHuoneKasittelija. Lisäksi viipaleeseen kuuluvan aspektin sisällä on muissa viipaleissa määritellyt luokat, joista ko. käyttötapaus on riippuvainen. VaraaHuone-aspektin kautta käyttötapaukseen saadaan siis liitettyä tarvittava toiminnallisuus. (Jacobson & Ng 2005). Perinteisesti ilman tällaisiin viipaleisiin jakamista, käyttötapaukset hajaantuisivat eri luokkien kesken. Tällä tavalla esimerkiksi uuden käyttötapauksen lisääminen tehtyihin rakennelmiin olisi hankalaa.




*Kuva 4 Käyttötapausviipale (Jacobson & Ng, 2005)*

## 3.3 Aspektikeskeinen ohjelmointi

Aspektikeskeisestä ohjelmointia käsitteleviä tutkimuksia löytyy nuoresta tutkimusalasta huolimatta suhteellisen hyvin. Tällä hetkellä Javaan perustuva AspectJ-ohjelmointikieli on laajimmin tunnettu. Muita aspektikieliä ovat mm. Aspect.NET sekä AspectC++. Tässä yhteydessä ei kuitenkaan käsitellä eri kielten ominaisuuksia, vaan tarkoitus on esittää aspektiohjelmoinnin perusrakenteet yleisellä tasolla. Rakenteiden esittäminen perustuu Kiczalesin ym. (2001) AspectJ-kieltä käsittelevään teokseen, joten on huomioitava, että muissa kielissä voi olla eroavuuksia mm. nimeämisten suhteen. Aspektiohjelmoinnin tarkoituksena on siis toteuttaa elinkaaren aikana erillään pidetyt kohteet siten, että läpileikkaavuus hallitaan aspekteilla erinäisiä ohjelmointikielen rakenteita hyödyntäen. Tällöin ohjelmakoodi ei hajaudu ohjelmiston eri osiin ja sitä on helppoa ylläpitää ja laajentaa.

### 3.3.1 Liitoskohtamalli

Liitoskohtamalli (engl. *join point model*) on minkä tahansa aspektikielen mekanismin suunnittelun kriittinen elementti. Malli tarjoaa mahdollisuuden suorittaa ohjelman aspektikoodi ja muu ohjelmakoodi koordinoidusti. Liitoskohdat (engl. *join points*) ovat hyvin määriteltyjä kohtia ohjelman suoritusvuossa, joiden kautta olio vastaanottaa metodikutsun ja kentän, johon kyseinen olio viittaa. Metodikutsun lisäksi voidaan mm. käsitellä muuttujia sekä alustaa luokka. Liitoskohtien kautta lisätoiminnallisuus on siis mahdollista liittää ohjelmaan.

Liitoskohtamäärittely (engl. *pointcut*) kokoaa yhteen liitoskohdat, joiden avulla aspektit laajentavat ohjelman olemassa olevaa toiminnallisuutta. Ohjelmakoodissa liitoskohtamäärittelyitä voidaan yhdistää esimerkiksi loogisilla operattoreilla.

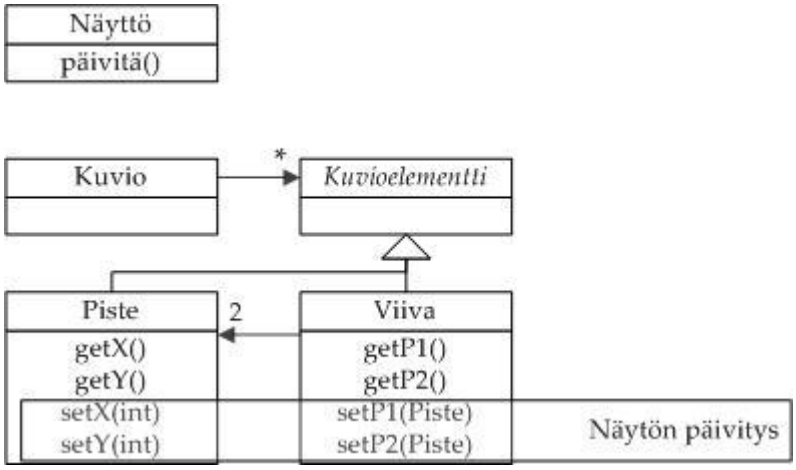
### 3.3.2 Kehote

Kehotteet (engl. *advice*) ovat metodin kaltaisia mekanismeja, joidan avulla liitoskohtiin voidaan lisätä toiminnallisuutta (ohjelmakoodia). Kehotteet voidaan jakaa kolmeen päätyyppiin, jotka ovat *before*, *after* sekä *around*. Before-kehote suoritetaan nimensä mukaisesti juuri ennen liitoskohdan suorittamista. Vastaavasti after suoritetaan heti liitoskohdan jälkeen. Around suoritetaan liitoskohdan ympärillä ja sen avulla voidaan kontrolloida sitä, tullaanko liitoskohdan ohjelmakoodi suorittamaan.

### 3.3.3 Aspekti

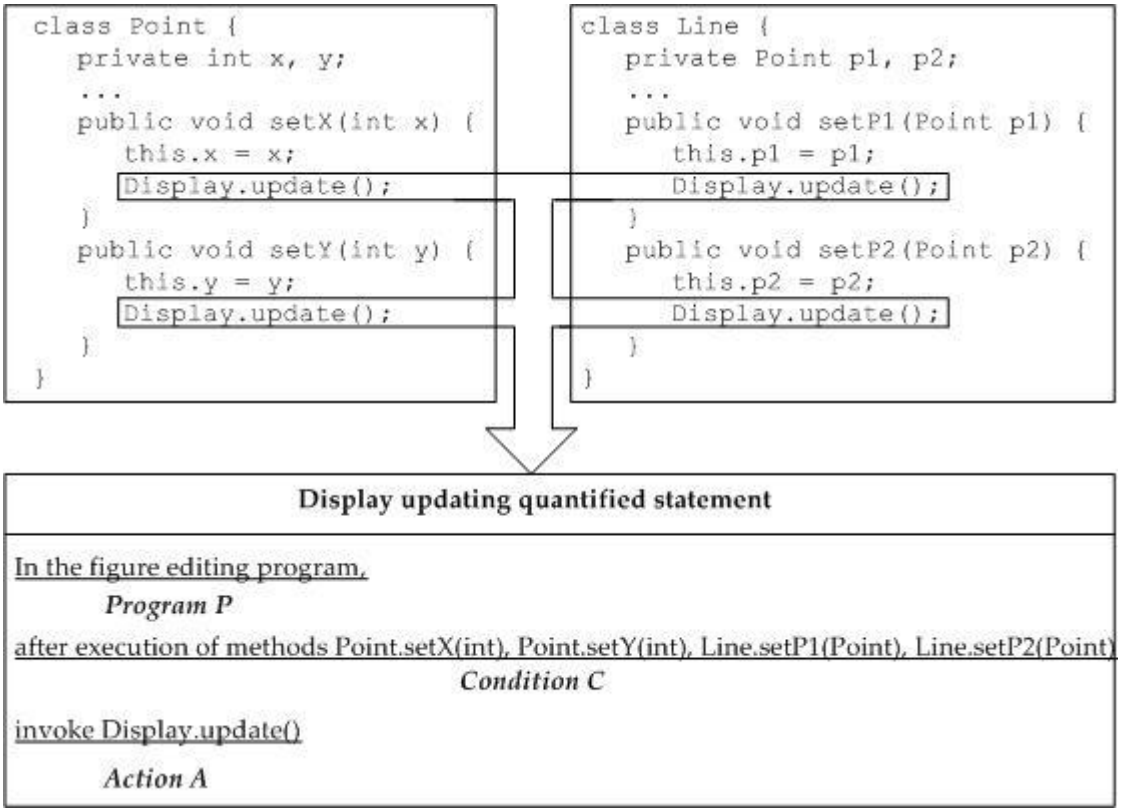
Aspektit ovat läpileikkaavan toteutuksen käsittäviä modulaarisia yksiköitä. Ne määritellään aspektin esittelyssä luokkien esittelyn tapaisesti. Aspektin esittely voi sisältää liitoskohtamäärittelyn ja kehotteiden esittelyjä sekä luokille tyypillisiä esittelyjä.

Elrad ym. (2001) esittelevät tunnetun esimerkin yksinkertaistetusta kuvionmuokkausohjelmasta, jossa näytön päivitysoperaatio läpileikkaa Piste- ja Viiva-luokan. Kuvassa 5 esitellään tyypillinen oliomalli kyseiselle ohjelmalle. Siinä kohdetta Näytön päivitys, joka päivittää näytön aina kun pisteet tai viivat liikkuvat, ei voida kapseloida yksittäisen moduulin sisään. Mallissa voisi kokeilla erilaista modularisoimista, joka paikallistaisi näytön päivityksen. Tällöin kuitenkin muut kohteet uudessa mallissa hajaantuisivat ristikkäin toistensa kanssa.




*Kuva 5 Läpileikkaavuus oliomallissa (Elrad ym., 2001)*

Filman ym. (2000) antavat aspektiohjelmoinnille seuraavanlaisen määritelmän: ”AOP (Aspect-Oriented Programming) can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers”. Tällaiset määritetyt lauseet voidaan kirjoittaa muotoon “In programs P, whenever condition C arises, perform action A”. Edellä esitellyssä esimerkissä määritetty lause voisi olla seuraavan kaltainen: ”Aina kun jokin liikkuu (suorittaa liikkumismetodinsa), näyttö täytyy päivittää”. Kuvassa 6 Shaker ja Peters (2005) valaisevat tällaisen lauseen toimintaa edellä esitellyn kuvionmuokkausohjelman tapauksessa. Kuvasta nähdään, kuinka kohde näytön päivitys saadaan paikallistettua määritetyn lauseen avulla, eikä sitä enää tarvitse toteuttaa hajautuneena luokkien sisälle. Lauseessa määritetään, että kuvionmuokkausohjelmassa tiettyjen metodien suorittamisen jälkeen, herätetään näytön päivitysoperaatio. Samalla havaitaan, että luokkien ohjelmoijat voivat olla Filmanin ym. (2000) määritelmän mukaan ”tietämättömiä” näytön päivityksestä, sillä se tapahtuu luokkien ulkopuolella.




*Kuva 6 Määritettyjen lauseiden käyttö kuvanmuokkausohjelmassa (Shaker & Peters, 2005)*

Kuvassa 7 näytön päivitys toteutetaan aspektina, joka sisältää liitoskohtamäärittelyn sekä kehotteen. Kuvasta nähdään, kuinka aspektin sisällä määritellään liitoskohta move(), jonka kautta aspekti laajentaa kuvionmuokkausohjelman toiminnallisuutta. Lisäksi aspekti sisältää kehotteen, joka after-komennolla pannaan täytäntöön sen *jälkeen*, kun liitoskohdan ehdot toteutuvat. Esimerkiksi sen jälkeen kun pisteelle annetaan uusi x-koordinaatti (suoritetaan funktio Point.setX(int)), suoritetaan kehotteessa oleva näytön päivitysfunktio. Tällä tavalla alunperin neljään eri funktioon kirjoitettu päivitysfunktio voidaan kätevästi hallita yhdessä paikassa aspektin avulla.

```
aspect DisplayUpdating {
   pointcut move() :
      execution(public void Point.setX(int)) ||
      execution(public void Point.setY(int)) ||
      execution(public void Line.setP1(Point)) ||
      execution(public void Line.setP2(Point));

   after(): move() {
      Display.update();
   }
}
```

Liitoskohtamäärittely

Kehote

*Kuva 7 AspectJ:llä toteutettu näytön päivitys (Shaker & Peters, 2005)*

# 4 YHTEENVETO

Tutkielmassa on esitelty aspektikeskeisen ohjelmistokehityksen perusperiaatteita sekä lähestymistapoja, joita aspektikeskeisen ohjelmiston elinkaaren vaiheisiin on kehitetty. Tutkielmassa on myös kuvailtu, mitä etuja aspektilähestymistapa tarjoaa oliolähestymistapaan nähden. Lisäksi aspektiohjelmointia on esitelty esimerkkien avulla.

Aspektikeskeisyyden ideana on vangita ohjelman läpileikkaavat ominaisuudet siten, että ne eivät hajaudu luokkien kesken. Tällaiset ominaisuudet on mahdollista toteuttaa aspektien avulla, joilla voidaan siis lisätä toiminnallisuutta olemassa oleviin luokkiin niiden ulkopuolelta. Lisäksi aspektiohjelmointi tarjoaa keinot kontrolloida sitä, missä vaiheessa lisätyt toiminnallisuudet suoritetaan. Tutkielmassa osoitettiin esimerkein, kuinka komponenteilla toteutettaessa läpileikkaavuutta ei pystytä hallitsemaan erillisinä yksiköinä, jolloin ohjelman ylläpidettävyys ja uusien toimintojen lisääminen hankaloituu. Toisessa luvussa esitelty esimerkki hotellin varausjärjestelmästä oli yksinkertainen, eikä näin ollen välttämättä tuo esiin todellista tarvetta aspektien käyttämiseen. Kuitenkin laajemmissa ja monimutkaisemmissa järjestelmissä modulaarisuuden merkitys korostuu ja kohteiden hajaantuminen saattaa olla yllättävänkin suuri ongelma.

Hotellin varausjärjestelmän yhteydessä esiteltiin ohjelmointiesimerkki, jossa olemassa oleviin luokkiin lisättiin tiettyä kohdetta koskevat operaatiot. Tällä tavalla olemassa oleviin luokkiin voidaan lisätä muitakin uusia ominaisuuksia (mm. attribuutteja). Tämä ei ole kuitenkaan ainoa aspektiohjelmoinnin tarjoama ominaisuus, vaan myös olemassa oleviin operaatioihin on mahdollista lisätä toiminnallisuutta. Kuvionmuokkausohjelmasta havaittiin, kuinka toiminnallisuuden lisääminen tapahtui liitoskohtien ja kehotteiden avulla. Lisäksi ohjelmasta havaittiin, kuinka kehotteen avulla on mahdollista määrittää, missä vaiheessa lisättävä ohjelmakoodi tullaan suorittamaan.

Aspektiohjelmoinnin muodostusmekanismi, jolla käyttäytymistä saadaan siis määriteltyä luokan ulkopuolelta käsin, on oikein käytettynä kätevä keino rakentaa modulaarisempia ohjelmistoja. Oikeanlainen käyttäminen tarkoittaa käytännössä sitä, että ohjelmiston kehitystä ohjaa tietty menetelmä, jolla läpileikkaavat kohteet saadaan pidettyä erillään jo ohjelmistoa suunniteltaessa. Mikäli minkäänlaisia menetelmiä tai lähestymistapoja ei käytetä, aspektiohjelmointi johtaa helposti järjestelmään, jossa aspektit laukaisevat toimintoja hallitsemattomasti. Tällaista järjestelmää on vaikeaa ylläpitää ja laajentaa.

Aspektikeskeiseen ohjelmistokehitysprosessiin kehitetyt lähestymistavat pohjautuvat usein oliokeskeisiin lähestymistapoihin. Monia lähestymistavoista voidaankin pitää eräänlaisina laajennoksina aikaisemmin kehitetyille malleille. Näin ollen malleissa on havaittavissa myös useita yhtäläisyyksiä toistensa kanssa. Esimerkiksi aiemmin esitellyn käyttötapauslähtöisen lähestymistavan tapa paketoida läpileikkaava kohde viipaleeseen muistuttaa monilta osin Theme-lähestymistavan tapaa kapseloida kohteet ns. teemoihin.

Joka tapauksessa lähestymistavoista on vaikeaa löytää sellaista, joka olisi selkeästi muita parempi. Lähestymistavan valinta riippuu myös siitä, millaista projektia ollaan tekemässä. Mikäli kysymys on esimerkiksi televiestinnän puolelle kehitettävästä ohjelmistosta, on selvää, että kannattaa tutustua aiemmin tällaisissa projekteissa käytettyyn The Motorola Weavr – lähestymistapaan. Mallien elinkaaren vaiheet kattavien menetelmien puolesta Theme ja aspektikeskeinen käyttötapauslähestymistapa ovat hyviä ehdokkaita. Lisäksi vahva tuki kirjallisuuden puolelta tukee näiden lähestymistapojen käyttämistä. Mallien kypsyyden puolesta Theme ja AAM ovat vahvoilla. Lisäksi AAM-lähestymistavan tarjoama työkalutuki puoltaa sen käyttämistä. Tässä yhteydessä on syytä myös huomioida, että arviointi perustuu vuosina 2005 ja 2006 julkaistuihin teoksiin. Näin ollen mm. arviot mallien kypsyydestä saattavat todellisuudessa vaihdella, sillä osa lähestymistavoista tai niitä

koskevasta kirjallisuudesta on julkaistu melko lähellä näitä ajankohtia. Eräs huomionarvoinen asia on lähestymistapojen soveltaminen tuotantokäytössä. Tutkimuksessa esitetyistä lähestymistavoista ainoastaan kahta oli varmuudella käytetty ”oikeissa” projekteissa. Lähestymistapoihin liittyvästä kirjallisuudesta löytyy tosin useita esimerkkisovelluksia, mutta niiden käyttämisestä erilaisissa projekteissa kaivattaisiin lisätutkimusta.

# LÄHTEET

Blair G., Blair L., Rashid A., Moreira A., Araújo J., Chitchyan R. 2004. Engineering Aspect-Oriented Systems. In Filman R., Elrad T., Clarke S., Aksit M., editors, Aspect-Oriented Software Development. Addison-Wesley, 2004, pp. 379-406.

Chitchyan R., Rashid A., Sawyer P., Garcia A., Alarcon M., Bakker J., Tekinerdogan B., Clarke S., Jackson A. 2005. Survey of Aspect-Oriented Analysis and Design Approaches. Technical Report, AOSD-Europe, 2005.

Clarke S., Baniassad E. 2005. Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, 2005.

Dijkstra E. 1982. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, 1982, pp. 60-66.

Elrad T., Aksit M., Kiczales G., Lieberherr K., Ossher H. 2001. Discussing aspects of AOP. Communications of the ACM vol. 44, no. 10, October 2001, pp. 33-38.

Filman R., Elrad T., Clarke S., Aksit M. 2004. Aspect-Oriented Software Development. Addison-Wesley, 2004.

Filman R., Friedman D. 2000. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns (OOPSLA), 2000.

Fox J. 2005. A Formal Foundation for Aspect-Oriented Software Development. Research on Computing Science vol. 14, 2005, pp. 241-251.

Grundy J. 1999. Aspect-Oriented Requirements Engineering for Component-based Software Systems. 4th IEEE International Symposium on Requirements Engineering, June 1999.

Grundy J. 2000. Multi-Perspective Specification, Design and Implementation of Software Components using Aspects. International Journal of Software Engineering and Knowledge Engineering vol. 10, no. 6, 2000.

Hanneman J., Chitchyan R., Rashid A. 2003. Report on the Workshop on Analysis of Aspect-Oriented Software. European Conference on Object-Oriented Programming (ECOOP), July 2003. Springer-Verlag, 2004.

Hilliard R. 1999. Aspects, Concerns, Subjects, Views. In Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.

Jacobson I. 2003. Use Cases and Aspects – Working Seamlessly Together. Journal of Object Technology vol. 2, no. 4, July-August 2003, pp. 7-28.

Jacobson I., Ng P-W. 2005. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2005.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. 2001. An Overview of AspectJ. European Conference on Object-Oriented Programming (ECOOP), 2001, pp. 327-353.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J-M., Irwin J. 1997. Aspect-oriented programming. European Conference on Object-Oriented Programming (ECOOP), June 1997.

Schauerhuber A., Schwinger W., Kapsammer E., Retschitzegger W., Wimmer M., Kappel G. 2006. A Survey on Aspect-Oriented Modeling Approaches. Vienna University of Technology & Johannes Kepler University, Technical Report, 2006.

Shaker P., Peters D. 2005. An Introduction to Aspect-Oriented Software Development. Newfoundland Electrical and Computer Engineering Conference, 2005.

Singh S., Chen H-C., Hunter O., Grundy J., Hosking J. 2005. Improving Agile Software Development using eXtreme AOCE and Aspect-Oriented CVS. Asia-Pacific Software Engineering Conference (APSEC), 2005.

Tietäväinen O. 2006. Aspektipohjaisuus ohjelmistokehityksessä. Jyväskylän yliopisto. Tietotekniikan pro gradu –tutkielma.

Viega J., Voas J. 2000. Can Aspect-Oriented Programming Lead to More Reliable Software? IEEE Software, November-December 2000, pp. 19-21.