Sumean olio-ohjelmoinnin teoriaa ja sovellusta

Teemu Hiltunen

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Martti Juhola
Toukokuu 2008

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Teemu Hiltunen: Sumean olio-ohjelmoinnin teoriaa ja sovellusta
Pro gradu -tutkielma, 55 sivua, 1 liitesivu
Toukokuu 2008

Sumeuden tutkimus on kypsynyt ja levinnyt viime vuosikymmeninä mm. olioperustaisen tietomallin alueelle. Sumeuden tarjoamat menetelmät ja työkalut auttavat monissa tapauksissa mallintamaan oliotietomallin kohdealuetta luonnollisemmin ja aidommin kuin perinteiset menetelmät. Sumeuden ja olio-ohjelmoinnin yhteensovittaminen tarvitsisi sovelluskehittäjän kannalta helppokäyttöiset työkalut sumeuden toteuttamiseen, mutta mitään yhtenäistä saati standardinmukaista viitekehystä sumeaan olio-ohjelmointiin ei vielä ole olemassa. Tässä työssä esitellään olioparadigman peruskäsitteitä, ja sitä kuinka sumeutta on tutkittu olioperustaisen tietomallin yhteydessä sekä viime aikoina myös olio-ohjelmoinnissa nykyisillä olio-ohjelmointikielillä. Työn tarkoitus on innostaa ohjelmoivaa lukijaa tutustumaan sumeuteen ja alkamaan hyödyntää sitä omassa ohjelmointityössään. Kirjallisuudesta löytyvien sumeiden oliotietomallien ja olio-ohjelmointiviitekehysten esittelyn lisäksi työssä esitellään myös oma ehdotus sumeaksi ohjelmointiviitekehykseksi Java-ohjelmointikielellä. Ehdotus on hyvin suppea, mutta tarjoaa perusidean jatkokehitystä ajatellen.

CR-luokat: D.1.5 [**Programming Techniques**]: Object-oriented Programming.; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving – *Uncertainty, "fuzzy" and probabilistic reasoning*.

Avainsanat ja -sanonnat: sumeus, oliperustaisuus, olio-ohjelmointi.

Sisällys

1. Johdanto 1
2. Olio-ohjelmointi 2
2.1. Olio 2
2.2. Luokka 4
2.3. Metaluokat, refleksiivisyys ja geneeriset luokat 7
2.4. Periytyminen 8
2.5. Myöhäinen sidonta ja monimuotoisuus 12
3. Sumeus 12
3.1. Yleistä 13
3.1.1. Tiedon epätäydellisyydestä 13
3.2. Sumeat joukot 13
3.2.1. Kielellinen muuttuja 15
3.2.1.1. Kielellinen aita 17
3.2.2. Normalisoitu sumea joukko 17
3.2.3. Alfa-taso 17
3.3. Mahdollisuusteoria 17
3.3.1. Mahdollisuusjakauma 18
3.4. Sumea luku 18
3.5. Samankaltaisuusrelaatio 19
3.6. Sumea logiikka 19
3.6.1. Sumea päättely 20
3.6.2. Sumea implikaatio 20
3.7. Pehmeä laskenta 20
4. Sumea oliotietomalli 21
4.1. Taustaa 21
4.2. Yleistä 22
4.2.1. Sumea attribuutti 22
4.2.2. Sumea instanssi 23
4.2.3. Sumea periytyvyys 23
4.2.4. Sumea käyttäytyminen 24
4.3. Rossazza *et al*.: sumeiden luokkien hierarkkinen malli 24
4.4. George *et al*. ja Yazici *et al*.: samankaltaisuus sumeassa oliotietomallissa 27
4.5. Bordogna *et al*.: sumea graafiperustainen oliotietomalli 29
4.5.1. Epämääräiset attribuuttien arvot 30
4.5.2. Olioiden epävarmat ominaisuudet 30
4.5.3. Olioiden voimistetut ominaisuudet 31

4.5.4. Sumeat luokat 31
4.5.5. Sumeiden luokkahierarkioiden määritelmä 31
4.6. Van Gyseghem *et al*.: epävarmuus ja sumeus oliotietomallissa 32
4.6.1. "Sumeusosio" UFO-mallissa 32
4.6.1.1. Moniarvoisten attribuuttien sumeat arvot 32
4.6.1.2. Pehmeä mallinnus ja sumeat ominaisuudet 33
4.6.1.3. Sumeat instanssit, sumeat jäsenet, sumeat luokat ja sumea erikoistaminen 33
4.6.1.4. Sumea periytyvyys 34
4.6.2. "Epävarmuusosio" UFO-mallissa 35
4.6.2.1. Epätarkat ja epävarmat attribuuttien arvot 35
4.6.2.2. Epävarmat rooli ja roolioliot 35
4.6.2.3. Ominaisuuksien epävarma soveltuvuus 36
4.6.2.4. Epävarmat instanssit 36
4.7. Marín *et al*.: sumeat tyypit 37
4.7.1. Sumean tyypin määritelmä 37
4.7.2. Sumeiden tyyppien ilmentyminen ja periytyminen 38
4.7.3. Sumean tyypin esittäminen terävässä oliomallissa 39
5. Sumea olio-ohjelmointi 39
5.1. Pereira: sumea oliomalli laajennetussa Java-kielessä 40
5.2. Berzal *et al*.: sumeuden ohjelmointi moderneissa olio-ohjelmointikielissä 43
6. Sumea ohjelmointirajapinta Java-kielellä 46
6.1. Sumeat annotaatiot 47
6.1.1. @FuzzyObject 47
6.2. Sumeuden hallintakomponentti 49
7. Yhteenveto 50
Viiteluettelo 50
Liite 1: UML-notaatio 56

1. Johdanto

Olioperustainen ohjelmointi – lyhyemmin olio-ohjelmointi – on nykyään vallitseva ohjelmointiparadigma. Syynä tähän ovat sen tarjoamat yksinkertaiset, mutta tehokkaat tavat mallintaa kohdealuetta sekä lisätä ohjelmistojen uudelleenkäytettävyyttä ja ylläpidettävyyttä. Näitä tapoja ovat mm. tiedon kapseloiminen ja hierarkkiset olioiden luokkarelaatiot kuten periytyminen.

Sumeat joukot ja sumea logiikka ovat menetelmiä, joilla käsitellään epätarkkuutta ja epävarmuutta joukko-opissa ja loogisessa päättelyssä. Sumeus tarjoaa tapoja ja menetelmiä käsitellä reaalimaailman olioita ja tapahtumia binaarista kaksiarvologiikkaa luonnollisemmin ja monasti myös tehokkaammin.

Sumeuden käyttäminen olio-ohjelmoinnissa tarjoaa keinon kohdealueella toimivien olioiden mahdollisen – ja joskus hyvin merkittävänkin – epätarkkuuden ja epävarmuuden huomioon ottamiseen.

Tässä työssä on tarkoitus käsitellä sumeuden hyödyntämistä oliotietomallissa ja etenkin olio-ohjelmoinnissa. Työ on tarkoitettu ensisijaisesti antamaan ohjelmoijalle käsitys siitä, mitä sumeus on ja kuinka sitä voisi käyttää omassa työssä ilman syvällistä matemaattista tuntemusta. Työssä ei siis syvennytä sumeiden teorioiden matemaattisiin malleihin, vaan keskitytään nimenomaan siihen, miten ohjelmoija voisi helposti ottaa käyttöönsä sumeuden tarjoamia työkaluja kohdealueen mallintamiseen ja loogiseen päättelyyn. Sumeuden käytön helppous olio-ohjelmoinnissa riippuu hyvin pitkälle siitä, millaisia valmiita ohjelmointirajapintoja ja -kirjastoja sitä varten on olemassa, koska sumeuden ohjelmointi perinteiseen kaksiarvologiikkaan perustuvilla ohjelmointikielillä vaatii aina lisätyötä. Myös nykyisten tietokoneiden sisäinen, binaarinen arkkitehtuuri asettaa sekin jo kynnyksen sumeuden yksinkertaiselle käytölle. Työssä käydään läpi aiheesta tehtyjä tutkimuksia ja niissä esiteltyjä ratkaisuja, ja tarkastellaan esiteltyjen ratkaisujen soveltuvuutta yleiseen ohjelmointityöhön. Lisäksi pyritään konstruoimaan aiempien tutkimustulosten pohjalta uutta, helppokäyttöisempää, sumeaa ohjelmointirajapintaa Java™-kielellä[1]. Tällainen yleiskäyttöinen rajapinta voisi tarjota ohjelmoijalle aiempaa matalamman kynnyksen käyttää sumeutta omassa työssään.

Työ on jaettu siten, että ensin käydään lyhyesti läpi olio-ohjelmoinnin peruskäsitteitä. Tämän jälkeen käsitellään sumeutta ja sen teorioita suppeasti ja yleisellä tasolla. Kun molempien aihepiirien perusasiat on saatu riittävissä määrin käsiteltyä, kerrotaan, kuinka sumeuden tutkimus on yhdistynyt oliotietomallien tutkimukseen viime vuosikymmeninä. Tämä luo hieman teoreettista lisäperustaa työn varsinaiselle aiheelle eli sumeuden olio-ohjelmoinnille, johon paneudutaan seuraavaksi kahden kirjallisuudessa ehdotetun tutkimuksen esittelyllä. Oma ehdotus sumeaksi olio-

[1] Java™ on rekisteröity tavaramerkki, jonka omistaa Sun Microsystems, Inc. http://java.sun.com.

ohjelmointiviitekehykseksi esitellään tämän jälkeen ja lopuksi luodaan yhteenveto työn tuloksista.

2. Olio-ohjelmointi

Kai Koskimies [1993] kuvailee olioperustaisuutta seuraavasti:

> Olioperustaisuus on yksi ohjelmointiparadigma, tapa kuvata erilaisten järjestelmien rakennetta ja toimintaa. Muita yleisiä ohjelmointiparadigmoja ovat mm. proseduraalinen ohjelmointi, funktionaalinen ohjelmointi, logiikkaohjelmointi ja rajoiteohjelmointi. Verrattuna kolmeen viimeksi mainittuun olioperustaisuuden sovellusalue on laajempi: olioperustaisuus sopii käytännöllisesti katsoen kaikkii sovelluksiin. Olioperustaisuus vastaa hyvin ihmisen tapaa hahmottaa maailmaa, ja on siksi perustavampaa laatua kuin tiettyihin laskentamalleihin pohjautuvat paradigmat. Voidaan sanoa, että olioperustaisuus on kaikkien sovellusten suurin yhteinen tekijä, ja siinä mielessä (ainakin nykytietämyksen mukaan) käsitetasoltaan optimaalinen yleiskäyttöinen ohjelmistojen kuvausmenetelmä.

Tässä luvussa kerrotaan lyhyesti olio-ohjelmoinnin peruskäsitteistä kuten luokista, olioista, periytyvyydestä ja tiedon kapseloinnista. Tässä työssä oletetaan, että olio-ohjelmoinnin perusteet ovat jo jossain määrin ennestään tuttuja lukijalle, ja siksi niitä käsitellään vain pintapuolisesti.

Kuten Koskimies [1993] toteaa, vastaa olioperustaisuus hyvin ihmisen tapaa hahmottaa maailmaa ja sille on luonteenomaista, että ohjelmiston rakenne seuraa sovelluksen käsitemaailman rakennetta. Kun tähän yhdistetään sumeuden tuoma epävarmuuden ja epätäsmällisyyden kuvaaminen, niin päästään entistä lähemmäksi reaalimaailman luonnollista kuvaamista ohjelmoinnissa. Tässä luvussa käsiteltyjä peruskäsitteitä tullaan luvussa 4 laajentamaan tukemaan ja sisältämään sumeutta.

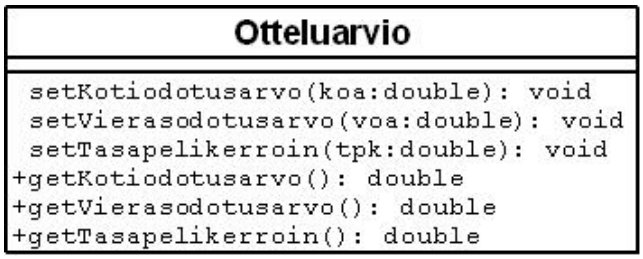
2.1. Olio

Olioperustainen tietomalli esittää kohdealueen kokoelmana vuorovaikutteisia *olioita* (object). Koskimiehen [1993] määritelmää seuraten voidaan oliolle määritellä alla olevat neljä oleellista ominaisuutta. Ominaisuuksien kuvausten yhteydessä on käytetty UML®-luokkakaavioita[2] (Unified Modeling Language) antamaan selventäviä esimerkkejä, vaikka kyseessä onkin olioiden eikä luokkien kuvaaminen. UML on yleisesti käytössä oleva mallinnuskieli, jolla kuvataan ohjelmistojen rakennetta ja suunnittelua. Liitteessä 1 on esitelty UML-luokkakaavion notaatio.

---

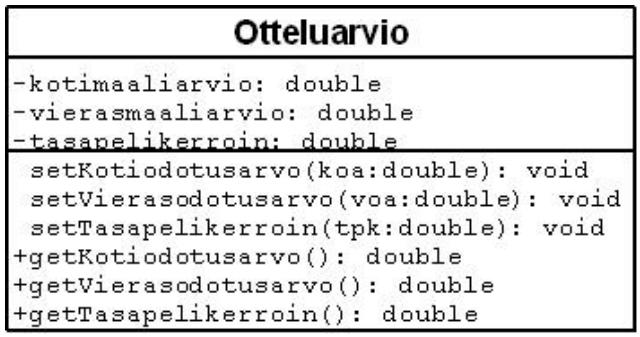
[2] UML® on rekisteröity tavaramerkki, jonka omistaa Object Management Group, Inc. http://www.omg.org ja http://www.uml.org.

1. Olio pystyy pyydettäessä suorittamaan tietyt tälle oliolle ominaiset toiminnot. Näitä toimintoja kutsutaan ohjelmointikielestä riippuen esimerkiksi metodeiksi, operaatioiksi, jäsenfunktioiksi tai rutiineiksi. Tässä työssä käytämme termiä metodi. Kuva 1 esittää urheiluvedonlyöntiin liittyvän esimerkkiluokan `Otteluarvio`, jolla on kolme asetusmetodia `setKotiodotusarvo`, `setVierasodotusarvo` ja `setTasapelikerroin` sekä kolme lukumetodia `getKotiodotusarvo`, `getVierasodotusarvo` ja `getTasapelikerroin`.



Kuva 1. Otteluarvio-luokan metodit.

2. Olio pystyy tallentamaan tietoa *attribuutteihin* eli nimettyihin tietokenttiin. Olion kulloinenkin *tila* määräytyy sen attribuuttien arvojen yhdistelmien mukaisesti. Metodin suoritus voi muuttaa olion tilaa muuttamalla sen attribuuttien arvoja. Olion attribuutteja ja metodeja kutsutaan yhteisesti olion *piirteiksi* tai *ominaisuuksiksi*. Olion attribuutit voivat olla joko yksinkertaisia tai yhdistettyjä eli kokoelmia. Yksinkertainen attribuutti voi olla esimerkiksi perustietotyyppinen, kuten kokonaisluku, reaaliluku tai merkki, tai suhde (association) toiseen olioon. Useimmiten suhde toiseen olioon toteutetaan kielen sisäisesti vittauksella tai osoittimella. Tällöin samaan olioon voidaan viitata useasta muusta oliosta, toisin kuin perustietotyyppien kohdalla. Yhdistetty attribuutti voi koostua edelleen joko yksinkertaisista tai yhdistetyistä attribuuteista. Kuva 2 esittää aiemman `Otteluarvio`-luokan hieman laajemmin sisältäen kolme attribuuttia: `kotimaaliarvio`, `vierasmaaliarvio` ja `tasapelikerroin`. Kaikkien kolmen attribuutin arvot ovat liukulukutyyppiä (Java-kielen `double`).



Kuva 2. Otteluarvio-luokka attribuutteineen.

3. Olio luodaan dynaamisesti ali ajoaikana. Luonnin yhteydessä syntyy olion identifioiva tunniste, *viite*. Kahdella eri oliolla on eri viite vaikka ne muutoin olisivat identtisiä. Samaan olioon voi kohdistua useita viittauksia. Identtisyys voidaan määritellä monella tavalla, mutta yleensä se tarkoittaa sitä, että oliot kuuluvat samaan luokkaan ja niiden attribuuttien arvot ovat identtisiä. Olion identiteetin voidaan myös sanoa tarkoittavan sitä, että oliot erotetaan toisistaan niiden luontaisen olemassaolon eikä kuvaavien ominaisuuksiensa kautta [Rumbaugh *et al*. 1991].
4. Olio on *suojattu* kokonaisuus, jonka käyttö on rajattu tiettyihin muotoihin. Yleensä vain tietyt olion piirteet ovat käytettävissä olion ulkopuolella. Tätä ominaisuutta kutsutaan *tiedon kapseloinniksi* (encapsulation).

Olioiden elinkaari alkaa niiden luomisesta ja päättyy niiden tuhoamiseen. Olioiden tuhoaminen tarkoittaa yleensä sitä, että olioon kohdistuvat viittaukset hävitetään. Tällöin oliota ei enää voida kutsua ohjelmassa ja se lakkaa näin ollen – ohjelmoijan näkökulmasta katsoen – olemasta, vaikkakin muistirakenne, mihin olio on tallentunut, ei samalla hetkellä vapautuisikaan. Monissa järjestelmissä oliot tuhoutuvat lopullisesti (so. niiden viemä muisti vapautuu) automaattisen "*roskienkeruun*" (garbage collection) myötä. Olioita voi olla järjestelmästä riippuen mahdollista myös tallentaa pysyvämmin kuin vain ohjelma-ajon ajaksi ns. *pysyvyysmekanismeilla* (persistence mechanisms). Tämä tarkoittaa sitä, että olion tila tallennetaan esimerkiksi tiedostoon tai tietokantaan, josta se on mahdollista palauttaa ja luoda alkuperäisessä tilassa ollut olio uuden ohjelma-ajon aikana tai jopa kokonaan toisessa järjestelmässä.

2.2. Luokka

Koskimies [1993] määrittelee *luokan* (class) olion malliksi, "kaavaimeksi", jonka avulla uusia olioita voidaan luoda. Jokaisella oliolla on yksikäsitteinen luokka[3], jonka mukaan olio on luotu. Oliot ovat siis luokan *instansseja*, *ilmentymiä*. Luokka määrittelee, mitä attribuutteja ja metodeja siihen kuuluvilla olioilla on, kuten edellä olioiden yhteydessä kuvissa 1 ja 2 esitettiin.

Toisin kuin oliot, luokat eivät yleensä ole dynaamisia, ajoaikana luotavia rakenteita. Luokka on staattinen, käännösaikainen käsite, jonka ohjelmoija suunnittelee ja toteuttaa valitun ohjelmointikielen syntaksin mukaisesti. Esimerkissä 1 on kuvattu Java-kielellä kuvan 2 luokka `commons.betting.Otteluarvio`[4].

Aiemmin olioiden kohdalla mainittu tiedon kapselointi tarkoittaa `Otteluarvio`-luokan kohdalla sitä, että vain sen kolme *julkiseksi* (public) määriteltyä lukumetodia

---

[3] Oletetaan, että ko. kieli sisältää luokan käsitteen. On myös olemassa olio-ohjelmointikieliä, joissa ei ole lainkaan luokan käsitettä, kuten esimerkiksi Self-kieli [Koskimies 1993].

[4] Java-kielessä luokkien koko nimi ilmoitetaan pakkausnimi mukaanlukien. Tässä työssä seurataan tätä tapaa ja ilmoitetaan luokasta ensimmäistä kertaa puhuttaessa sen koko nimi pakkaus mukaanlukien.

ovat käytettävissä kaikkialla luokan ulkopuolelta. Luokan kaikki attribuutit on määritelty *yksityiseksi* (private), jolloin niitä voidaan käyttää vain luokan sisäisesti. Tiedon suojaamiseen on monissa olio-ohjelmointikielissä, esim. Javassa olemassa myös määre *suojattu* (protected), joka tarkoittaa, että attribuutti tai metodi on käytettävissä vain kyseisessä luokassa tai siitä periytyvässä aliluokassa. Java-kielessä on lisäksi vielä neljäskin suojaustason määre, jolla määritellään luokan piirre *pakkauskohtaisesti suojatuksi* (package-level protection). Se tarkoittaa sitä, että piirre on julkinen ainoastaan saman pakkauksen sisällä. Esimerkkiluokka `Otteluarvio` kuuluu pakkaukseen `commons.betting` ja sen kaikki kolme asetusmetodia on julkisia vain saman pakkauksen sisällä. Tällä tavoin määriteltynä luokan instanssit voivat saada attribuuttiensa arvot vain saman pakkauksen sisällä olevalta toiselta luokalta asetusmetodien kautta, mutta mikä tahansa olio, mistä tahansa pakkauksesta, voi kysyä samaisia arvoja julkisilla lukumetodeilla. Näin luokan käyttö ja sen instanssien tilojen muutokset kyseisen pakkauksen ulkopuolelta on paremmin suojattu.

Esimerkki 1: Otteluarvio-luokan toteutus Java-kielellä

Alla on Java-kielinen `Otteluarvio`-luokka, jonka attribuuttien ja metodien selitykset on annettu ohjelmakoodin kommenteissa `/*`- ja `*/`-merkkien välissä. Esimerkissä on kiinnitetty huomiota myös asetusmetodien parametrien arvojen oikeellisuuden tarkistukseen. Jos annettu arvo ei ole sallittu, niin heitetään ajoaikainen *poikkeus* (exception). Poikkeus on erityinen olio-ohjelmointikielten käsite, jonka avulla voidaan helpottaa virheiden käsittelyä ja etsimistä.

```
package commons.betting;

/**
 * Otteluarvio-luokka kuvaa yhden ottelun
 * (esim. jääkiekko tai jalkapallo)
 * arvion urheiluvedonlyöntiä varten.
 * Otteluarvioita käytetään optimaalisten
 * panosten laskennassa.
 *
 * @author teemu
 *
 */
public class Otteluarvio {

     private double tasapelikerroin;

     private double kotiodotusarvo;

     private double vierasodotusarvo;

     /**
      * Asettaa tasapelikertoimen.
      *
```

```
     * @param tasapelikerroin -
     *            oltava > 0
     * @throws IllegalArgumentException
     *             jos tasapelikerroin <= 0
     */
    public void setTasapelikerroin(
              double tasapelikerroin) {
         if (tasapelikerroin <= 0) {
              throw new IllegalArgumentException(
                        "tasapelikerroin oltava > 0");
         }
         this.tasapelikerroin = tasapelikerroin;
    }

    /**
     * Asettaa kotijoukkueen odotusarvon (esim. maaliluku).
     *
     * @param kotiodotusarvo -
     *            oltava >= 0
     * @throws IllegalArgumentException
     *             jos kotiodotusarvo <= 0
     */
    public void setKotiodotusarvio(
              double kotiodotusarvo) {
         if (kotiodotusarvo < 0) {
              throw new IllegalArgumentException(
                        "kotiodotusarvo oltava >= 0");
         }
         this.kotiodotusarvo = kotiodotusarvo;
    }

    /**
     * Asettaa vierasjoukkueen odotusarvon (esim. maaliluku).
     *
     * @param vierasodotusarvo -
     *            oltava >= 0
     * @throws IllegalArgumentException
     *             jos vierasodotusarvo <= 0
     */
    public void setVierasodotusarvio(
              double vierasodotusarvo) {
         if (vierasodotusarvo < 0) {
              throw new IllegalArgumentException(
                        "vierasodotusarvo oltava >= 0");
         }
         this.vierasodotusarvo = vierasodotusarvo;
    }

    /**
     *
     * @return tasapelikerroin
     */
    public double getTasapelikerroin() {
         return this.tasapelikerroin;
    }

    /**
     *
     * @return kotiodotusarvo
     */
    public double getKotiodotusarvo() {
         return this.kotiodotusarvo;
    }

    /**
```

```
     *
     * @return vierasodotusarvo
     */
    public double getVierasodotusarvo() {
        return this.vierasodotusarvo;
    }
}
```

On hyvin pitkälti sovelluskohtaista, kuinka olioiden luokat määritellään. Rumbaugh *et al*. [1991] antavat seuraavan esimerkin: vaikka hevosella ja ladolla on kummallakin ikä ja hinta, niin ne voivat silti olla eri luokkia. Jos kyseessä on sovellus, jossa hevosta ja latoa kumpaakin käsitellään puhtaasti taloudellisena omaisuutena, niin voi olla tarkoituksenmukaista käsitellä niitä kumpaakin samassa luokassa. Jos taas sovelluskehittäjän täytyy ottaa huomioon se, että henkilö voi maalata ladon, mutta ruokkia hevosen, niin ne luultavasti mallinnettaisiin eri luokkina.

Luokalle voidaan määritellä, paitsi em. instanssikohtaisia ominaisuuksia, myös staattisia, luokkakohtaisia attribuutteja ja metodeja. Java-kielessä staattisuus määritellään antamalla attribuutin tai metodin määrittelyssä suojaustason yhteydessä, ennen ominaisuuden tyyppimääritystä `static`-määre. Staattisilla metodeilla ei ole oikeutta käsitellä instanssikohtaisia ominaisuuksia, mutta instanssikohtaisissa metodeissa voidaan viitata staattisiin ominaisuuksiin. Staattisia metodeja käytetään usein ns. työkalumetodeina: ne tekevät jonkin toimenpiteen saamiensa parametrien pohjalta olematta riippuvaisia siitä luokasta ja sen instanssikohtaisista ominaisuuksista, jossa metodi on määritelty. Esimerkkinä triviaali laskutoimitusmetodi:

```
/**
 *
 * @param a
 * @param b
 * @return a+b
 */
public static int ynnaaLuvut(int a, int b) {
    return a+b;
}
```

2.3. Metaluokat, refleksiivisyys ja geneeriset luokat

Luokan kuvausta luokkana kutsutaan *metaluokaksi*. Kaikissa olio-ohjelmointikielissä ei tuoda tätä käsitettä ohjelmoijan ulottuville, mutta esimerkiksi Java-kielessä näin tehdään `java.lang.Class`[5]-luokan avulla. Luokan `Class` instanssit esittävät kaikki käynnissä olevan sovelluksen luokat ja rajapinnat, mukaanlukien taulukot ja

[5] Java-kielessä on sovittu niin, että pakkauksen `java.lang` luokkiin voidaan viitata ohjelmassa ilman pakkausmääritystä, mutta kaikkien muiden pakkausten käyttö vaatii pakkauksen tuomisen lähdekoodiin joko *import*-lauseella tai käyttäen luokan koko nimeä pakkausmääre mukaanlukien.

primitiiviset tietotyypit (boolean, byte, char, short, int, long, float ja double). Näin ohjelmoija voi tutkia ohjelman sisällä esimerkiksi jonkin tietyn olion luokan nimeä ja piirteitä, ja käyttää niitä *refleksiivisesti*. Refleksiivisyys on siis sitä, että kielen kautta päästään käsiksi kielen omiin mekanismeihin [Koskimies 1993]. Tästä voi olla väärin käytettynä haittaa, koska refleksiivisyydellä on mm. mahdollista ohittaa private-määritys, eli päästä käsiksi olion yksityisiin ominaisuuksiin. Tämän takia refleksiivisyyttä ei tulisi käyttää kevein perustein, vaan ainoastaan, kun sillä oikeasti saavutetaan hyötyä. Oikein ja soveltuvissa kohdin käytettynä on refleksiivisyys kuitenkin mielestäni oiva apuväline käytännön ohjelmointityössä – piirre, joka voi tarjota monissa ongelmatilanteissa elegantimman ratkaisun kuin perinteiset oliomenetelmät. Refleksiivisyydestä voi olla hyötyä esimerkiksi sumeuden ohjelmoinnissa, kuten Berzal *et al.*:n [2003] työssä on esitetty (ks. 5.2).

Geneerinen (eli *parametroitu*) *luokka* on luokka, joka parametrien avulla kiinnitetään johonkin toisiin luokkiin. Koskimies [1993] vertaa geneeristä luokkaa erikoistuneeseen makroon, josta parametrien korvauksella saadaan aikaan erilaisia todellisia luokkia. Geneerisyys mahdollistaa mm. yleiskäyttöisten algoritmien kirjoittamisen. Olio-ohjelmointikielet tukevat geneeristä ohjelmointia eri tavoin. Tunnetuin geneerisyyden mahdollistava mekanismi ovat C++-kielen kaavaimet. Java-kielessä[6] geneerisiä luokkia käytetään useimmiten tehtäessä yleiskäyttöisiä *säiliöluokkia* (container class) eli luokkia jotka toimivat kokoelmina. Tällaisia kokoelmia ovat Javassa esimerkiksi lista `java.util.List<E>` ja joukko `java.util.Set<E>`, jotka molemmat ovat ylemmän kokoelma-abstraktion `java.util.Collection<E>` erikoistuksia. Kokoelmien kohdalla geneerisyys tuo Java-kielessä huomattavaa helpotusta aiempaan tilanteeseen, jossa oltiin pakotettu käyttämään kokoelmien alkioiden luokkana yleistä, kaikkien luokkien juuriluokkaa `java.lang.Object`. Geneerisyyden myötä kokoelman alkioluokka voidaan kiinnittää parametrilla (`<E>` edellä), jolloin vältytään tyyppitarkistuksilta ja -muunnoksilta, joita aiemmin jouduttiin tekemään sen selville saamiseksi, mikä `Object`-luokan aliluokka oikeasti oli kyseessä.

Myös em. `Class`-luokka on geneerinen, `Class<T>`, missä `T` on sen luokan tyyppi, jota ko. luokka mallintaa. Esimerkiksi merkkijonoluokan `java.lang.String` metaluokka on `Class<String>`.

2.4. Periytyminen

Periytyminen (inheritance) on yksi olioparadigman merkittävimmistä piirteistä. Se tarkoittaa luokan (yliluokka) piirteiden siirtymistä jollekin toiselle luokalle (aliluokka). Koskimies [1993] esittelee periytymisen kaksi keskeistä tavoitetta seuraavasti: i) periytyminen johtaa *koodin uudelleen käyttöön*, koska aliluokkien ei tarvitse toistaa

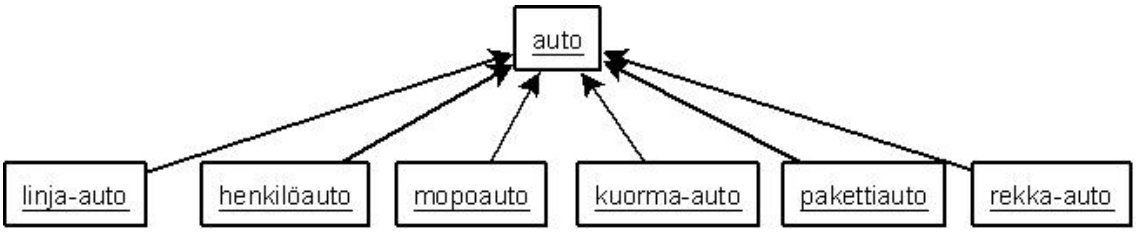
[6] Geneerisyys on tullut mukaan Java-kieleen versiosta 5 alkaen.

perittyjä piirteitä ja ii) periytyminen tukee käsitteelliseen mallintamiseen pohjautuvaa ohjelmakehitystä, koska se vastaa läheisesti *erikoistamisen* ja *yleistyksen* käsitteitä. Nämä kaksi tavoitetta eivät aina ole yhteneviä, vaan toisesta joudutaan monasti tinkimään toisen kustannuksella. Käytännönläheinen, pragmaattinen, ohjelmistokehitys suosii usein koodin uudelleenkäyttöä käsitteellisen "oikeaoppisuuden" sijaan. Usein käsitteellisen mallin suora siirto olioparadigmaan ei olekaan hyvä ajatus, vaan mallia joudutaan soveltamaan käytännön syistä esimerkiksi tehokkuuden parantamiseksi. Esimerkissä 2 on annettu käsitemalli erilaisista autoista ja mallin kaksi erilaista sovellusta olioparadigmassa.

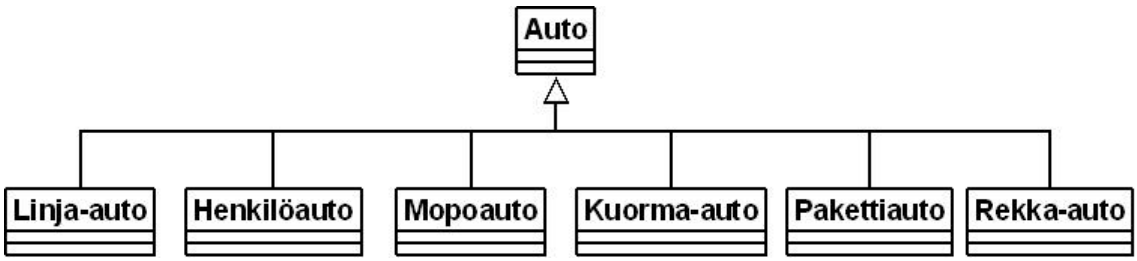
Esimerkki 2. Käsitemallin sovellus olioparadigmassa

Kuvassa 3 on kuvattu joidenkin erilaisten autojen käsitteet. Ylikäsitteestä *auto* on erikoistettu käsitteet *linja-auto*, *henkilöauto*, *mopoauto*, *kuorma-auto*, *pakettiauto* ja *rekka-auto*. *Auto* on siis näiden erikoistettujen käsitteiden yleistys. Kuvassa 4 on käsitemallin suora sovellus olioparadigmassa. Siinä periytymisellä on kuvattu käsitemallin erikoistuminen: `Auto` toimii yliluokkana kaikille eri autotyypeille. Kuvassa 5 on puolestaan toisenlainen oliosovellus samasta käsitemallista. Siinä auton tyyppi onkin kuvattu auto-luokan attribuuttina eikä periytymistä ole käytetty lainkaan.

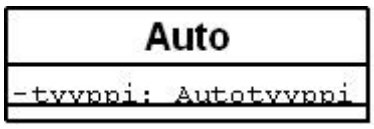
Se, käytetäänkö oliomallissa periytymistä tai ei, riippuu pitkälti sovelluksen toiminnallisuudesta; siitä miten olioita on tarkoitus käyttää, miten erikoistuneita metodeja tai attribuutteja eri tyypit tarvitsevat jne. Vaikka kohdealueen käsitteistön kerääminen ja käsitemallien luominen onkin tärkeä osa ohjelmiston suunnitteluprosessia, niin siitä saatua mallia ei aina voida – eikä kannatakaan – siirtää sellaisenaan oliomalliin.



Kuva 3. Autojen käsitemalli.



Kuva 4. Autojen luokkakaavio periytymistä käyttäen.



Kuva 5. Auton luokkakaavio ilman periytymistä.

Aiemmin kohdassa 2.2 käsiteltiin tiedon kapselointia. Periytyminen tuo tähän oman vaikutuksensa siten, että ohjelmointikielestä riippuen voidaan yliluokan joillekin piirteille määritellä erilaiset suojaustasot, jolloin kaikki piirteet eivät ole edes luokan aliluokkien käytettävissä. Tämä ei tarkoita, etteikö aliluokalla olisi ko. piirteitä, mutta niihin ei pystytä viittaamaan aliluokasta käsin.

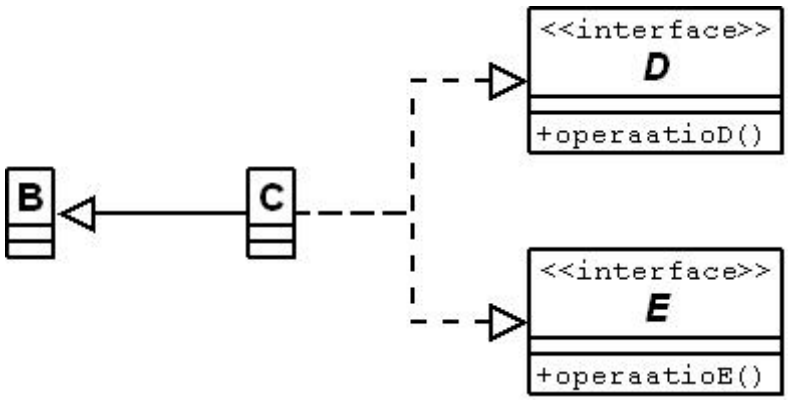
Periytyminen voi vaikuttaa toisellakin tavalla tiedon kapselointiin. Yliluokan julkiset piirteet ovat julkisia myös periytyvässä aliluokassa eikä aliluokalla ole mahdollisuutta niitä piilottaa. Toisinpäin taas on yleensä mahdollista tehdä eli aliluokassa voidaan määritellä metodi julkiseksi, vaikka se olisi yliluokassa määritelty suojatuksi.

Abstrakti luokka on luokka, josta ei voida luoda ilmentymiä [Koskimies, 1993]. Luokkahierarkian juuriluokat ovat usein määritelty abstrakteiksi luokiksi. Ne tarjoavat aliluokille yhteisiä attribuutteja ja metodeja, mutta eivät sellaisenaan mallinna mitään kohdealueen oliota.

Tähän mennessä on periytyminen kuvattu ns. yksittäisperiytymisenä eli kullakin luokalla on korkeintaan yksi välitön yliluokka. On myös mahdollista, että luokalla on useita välittömiä yliluokkia. Tätä kutsutaan *moniperiytymiseksi* (multiple inheritance) [Koskimies, 1993]. Eri ohjelmointikielissä moniperiytyminen on toteutettu hieman eri tavoin - sikäli kun sitä on toteutettu lainkaan. Java-kielessä jokaisella luokalla on vain yksi perittävä yliluokka (jos luokalle ei ole eksplisiittisesti määritelty yliluokkaa, niin se periytyy em. `java.lang.Object`-luokasta), mutta luokalla voi olla useita toteutettavia *rajapintoja* (interface). Rajapinnan voidaan katsoa olevan määritelmä metodeista, jotka luokan tulee toteuttaa. Rajapinta ei voi sisältää attribuutteja (paitsi staattisia luokkakohtaisia attribuutteja), eikä se voi periytyä luokasta vaan ainoastaan yhdestä tai useammasta muusta rajapinnasta. Rajapinta on siis eräällä tavalla abstrakti luokka, mutta ilman attribuutteja ja metodien toteutusta. Rajapinnasta ei voi luoda olioita, vaan sitä voi käyttää ainoastaan määrittelemään tietynlainen toiminnallisuus sen toteuttavalle luokalle. Esimerkissä 3 on kuvattu rajapinnan käyttöä moniperiytymisessä Java-kielessä.

Esimerkki 3: Moniperiytyminen Java-kielessä

Periytyköön luokka `C` luokasta `B`, ja toteuttakoon se rajapinnat `D` ja `E`. Tämä kuvataan UML-kaaviolla kuvan 6 mukaisesti.



Kuva 6. Moniperiytyminen.

Luokka `C` määriteltäisiin Java-kielellä seuraavasti:

```
/**
 * Luokka, joka periytyy luokasta B, ja
 * joka toteuttaa rajapinnat D ja E.
 *
 * @author teemu
 *
 */
public class C extends B implements D, E {

	/**
	 *
	 * @see commons.betting.D#operaatioD()
	 */
	@Override
	public void operaatioD() {
		// ...
	}

	/**
	 *
	 * @see commons.betting.E#operaatioE()
	 */
	@Override
	public void operaatioE() {
		// ...
	}
}
```

Van Gyseghem and De Caluwe [1998] tuovat esiin tärkeän erotuksen käsitteiden instanssi ja jäsen välille: olio on instanssi vain sen perusluokalle (so. erikoistetuin luokkahierarkian luokka, joka määrittelee oliota) ja jäsen sekä perusluokalle että kaikille tuon luokan yliluokille luokkahierarkiassa.

Esimerkiksi luokka `A` periytyy luokasta `B`, joka periytyy edelleen luokasta `C`. Olio `a` määritetään luokan `A` instanssiksi:

```
A a = new A();
```

Olio `a` on tällöin luokkahierarkian mukaisesti sekä luokan `A` että luokkien `B` ja `C` jäsen, mutta vain luokan `A` instanssi.

2.5. Myöhäinen sidonta ja monimuotoisuus

Myöhäinen eli dynaaminen sidonta tarkoittaa sitä, että kutsuttavan metodin toteutustapaa ei tiedetä aina käännösaikana eli staattisesti. Tämä johtuu siitä, että olio, jonka metodia kutsutaan ei välttämättä itse toteuta metodia, vaan toteutus on määritelty jossakin sen yliluokassa. Toisaalta taas olio, joka on staattisesti ohjelmakoodissa määritelty olemaan jonkin luokan ilmentymä, sidotaankin dynaamisesti ajonaikana johonkin tuon luokan aliluokkaan. Tällöin olion metodia kutsuttaessa toteutuu ohjelman ajonaikaisesti *monimuotoisuus* (eli polyformismi): metodin suoritustapa riippuu myöhäisen sidonnan ansiosta kyseisen olion perusluokasta (eikä siis muuttujan staattisesta luokasta). [Koskimies, 1993]

Esimerkissä 4 on kuvattu, mitä myöhäinen sidonta tarkoittaa havainnollistettuna Java-kielen avulla.

Esimerkki 4: Myöhäinen sidonta Java-kielessä

Periytyköön luokka `C` luokasta `B`, joka puolestaan periytyy luokasta `A`. Luokassa `A` on määritelty metodi `operoi`, joka on ylimääritetty aliluokissa `B` ja `C`. Metodin suorituksen toteutus selviää vasta ajoaikaisesti riippuen muusta ohjelman toiminnasta; syöteparametreista tms.

```
// viittaus luokan A instanssiin
A olioA = null;

// luodaan instanssi riippuen ajoaikaisesta tilanteesta
if (i > 5) {
     olioA = new B();
} else {
     olioA = new C();
}

// kutsutaan metodia, josta ei käännösaikana
// tiedetä missä luokassa sen toteutus
// on määritelty.
olioA.operoi();
```

3. Sumeus

Tässä luvussa kerrotaan lyhyesti sumeudesta, sen käsitteistä ja teoriasta. Sumeuden matematiikkaan ei syvennytä kovin tarkasti, vaan sitä käsitellään siinä määrin kuin on tarpeen myöhempää sumeuden ja olio-ohjelmoinnin yhteensovittamista varten.

3.1. Yleistä

Sumeus tarkoittaa yleisesti ottaen sitä, että asioiden totuusarvo tai mahdollisuus ei ole rajoittunut perinteiseen mustavalkoiseen joko–tai-ajatteluun, vaan totuudella on myös näiden ääripäiden välisiä "harmaita" arvoja. Sumeudesta puhuttaessa voidaan tilanteesta riippuen tarkoittaa monia erilaisia sumeita käsitteitä kuten sumeat joukot, sumea logiikka, sumea päättely, sumea laskenta ja sumea samankaltaisuus.

Sumeuden isänä[7] pidetään yleisesti Kalifornian yliopistossa, Berkeleyssä professorina toimivaa Lofti Zadehia, joka vuonna 1965 esitteli sumean joukon käsitteen avuksi epätäydellisen tiedon käsittelyyn [Zadeh, 1965].

3.1.1. Tiedon epätäydellisyydestä

Tieto voi olla *epätäydellistä* (imperfect). Bordogna *et al*. [1997] jakavat epätäydellisyyden kattamaan seuraavanlaiset tiedon vajavuudet: *epätarkkuus* (imprecision), *epämääräisyys* (vagueness), *epävarmuus* (uncertainty) ja *epäjohdonmukaisuus* (inconsistency). Epätarkkuus ja epämääräisyys liittyvät lauseen sisältöön. Esimerkiksi lauseet "noin 80 km/h" ja "välillä 70–100 km/h" ovat epätarkkoja ja epämääräisiä arvoja auton nopeudelle. Epävarmuus liittyy lauseen totuuteen. Ilmaisuja, kuten "todennäköisesti" ja "on mahdollista, että" voidaan käyttää määrittelemään tiedon osittaista tietämättömyyttä. On myös tilanteita, joissa tietoon vaikuttaa sekä epävarmuus että epätarkkuus tai epämääräisyys, kuten esimerkiksi lauseessa "John on mahdollisesti erittäin nuori". Epäjohdonmukaisuus puolestaan syntyy, kun samasta todellisuuden tilasta on yhtä aikaa ristiriitaista tietoa, kuten esimerkiksi lauseissa "John on alle 40-vuotias" ja "John on yli 45-vuotias". [Bordogna *et al*., 1997]

3.2. Sumeat joukot

Sumeat joukot ovat sumeuden perusta, jonka päälle sumeuden muut osa-alueet rakentuvat. Sumeiden joukkojen teoriat ovat laaja-alaisia ja osin monimutkaisiakin, mutta tässä työssä ei kaivauduta niihin kovinkaan paljon pintaa syvemmälle, vaan pitäydytään niissä perusasioissa, joita myöhemmissä kohdissa tullaan tarvitsemaan.

Olkoon X ei-tyhjä joukko. Sumea joukko A X:ssä määritellään *jäsenyysfunktion* (membership function) avulla seuraavasti:

$$\mu_A: X \rightarrow [0, 1],$$

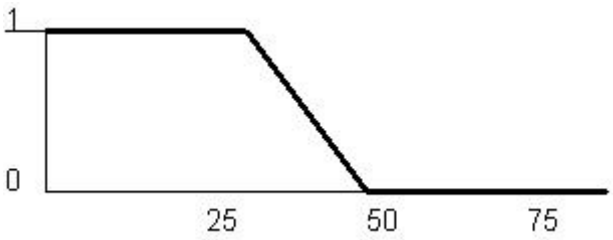
missä $\mu_A(x)$ on x:n *jäsenyysaste* (degree of membership) kaikille $x \in X$ sumeassa joukossa A. [Fullér, 1999]

---

[7] Max Black kehitti jo ennen Zadehia sumeuden käsitettä, mutta vasta Zadeh antoi sille formaa-lisemman määrityksen. On esitetty myös paljon kritiikkiä koko sumeuden käsitettä vastaan, mm. monien matemaatikoiden ja tilastotieteilijöiden toimesta, joiden mielestä "sumeus" on vain todennäköisyyslaskentaa verhottuna uutuudeksi. [Kosko, 1996]

Esimerkki 5: Sumea joukko

Kuvassa 7 on esimerkki sumeasta joukosta “nuori” joukossa Ikä. Kuvasta nähdään kuinka joukon jäsenyysfunktion arvo alkaa laskea 25 vuoden jälkeen asteittain kohti 50 ikävuotta. Näin ollen esimerkiksi 30-vuotias henkilö olisi tässä sumeassa joukossa “nuori” jäsenyysasteella 0,8 ja 60-vuotias jäsenyysasteella 0 eli ei lainkaan “nuori”.



Kuva 7. Sumea joukko “nuori”.

Sumean joukon *tuki* (support) määritellään:

$$^{0+}A = \{ \text{ x} \mid \mu_A(x) > 0 \}$$

eli kaikkien niiden pisteiden joukko, jossa jäsenyysaste on suurempi kuin 0. Esimerkiksi kuvan 7 tapauksessa $^{0+}A = \{ \text{ x} \mid 0 < \text{x} < 50 \}$.

Sumean joukon *ydin* (core) määritellään:

$$\text{core}(A) = \{ \text{ x} \mid \mu_A(x) = 1 \}$$

eli kaikkien niiden pisteiden joukko, jossa jäsenyysaste on 1. Esimerkiksi kuvan 7 tapauksessa $\text{core}(A) = \{ \text{ x} \mid 0 < \text{x} <= 25 \}$.

Sumean joukon sanotaan olevan *normaali*, jos

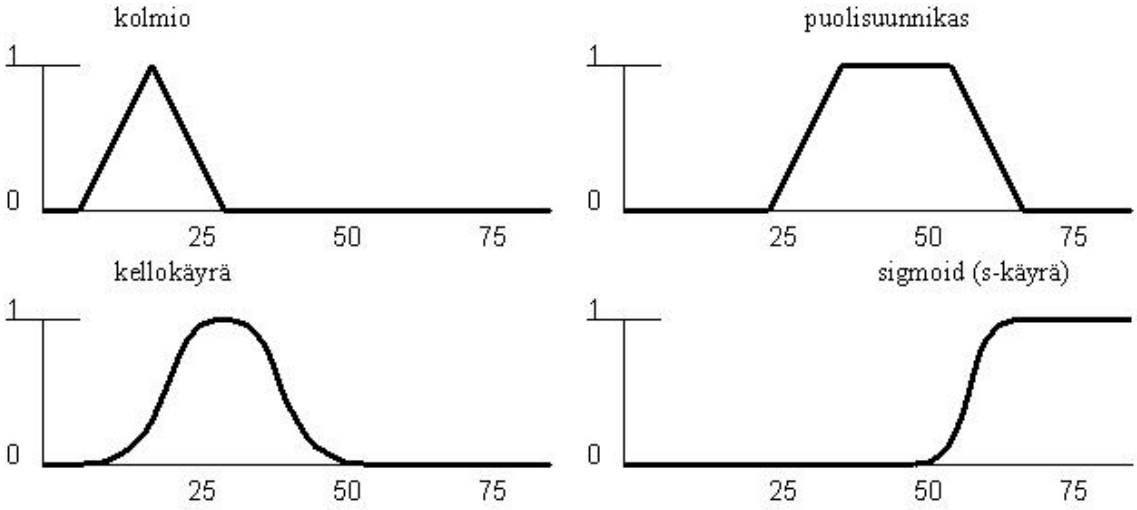
$$\exists \text{ x} \in \text{X}: \mu_A(x) = 1,$$

eli on olemassa ainakin yksi piste, missä jäsenyysfunktio saa arvon 1. Toisin sanoen sumea joukko on normaali, jos sen ydin ei ole tyhjä joukko.

Sumean joukon komplementti $-A$ määritellään

$$\mu_{-A} = 1 - \mu_A.$$

Sumeiden joukkojen jäsenyysfunktiot voivat saada erilaisia geometrisia muotoja riippuen funktiosta. Yleisimmin käytetyt jäsenyysfunktiot ovat kolmion muotoinen (triangular) ja puolisuunnikas (trapezoidal). Näitä tehokkaampia ilmaisuvoimaltaan – ja samalla monimutkaisempia – ovat kellonmuotoinen (bell-shaped) ja sigmoidi (sigmoidian). Kuvassa 8 on esimerkkejä erilaisista jäsenyysfunktioista.



Kuva 8. Tyypillisiä jäsenyysfunktioita.

Sumeiden joukkojen jäsenyysfunktioiden määrittely on usein hankalaa; miten valita funktiot siten, että malli vastaisi mahdollisimman hyvin todellista tilannetta? Eräs tapa voisi olla käyttää kohdealueen asiantuntijoita ja heidän kokemustaan ja tietämystään apuna. On myös mahdollista käyttää neuroverkkoja optimoimaan kielellisiä muuttujia ja jäsenyysfunktioita [Duch *et al*, 1999]. Tässä työssä ei oteta tämän enempää kantaa tähän asiaan, mutta se tuo oman lisäpiirteensä hyvän ja käyttökelpoisen sumean olio-ohjelmointirajapinnan toteutukseen ja jatkotutkimukseen.

3.2.1. Kielellinen muuttuja

Zadeh [1972, 1975] esitteli kielellisen muuttujan (linguistic variable), ja siihen liittyen kielellisen aidan (linguistic hedge) käsitteet jatkona sumeiden joukkojen tutkimukselleen. Aivan kuten numeeriset muuttujat saavat numeerisia arvoja, niin kielelliset muuttujat saavat kielellisiä arvoja, jotka ovat sanoja (kielellisiä termejä), joilla on tietty jäsenyysaste joukossa.

Kielellinen muuttuja määritellään formaalisti viisikkona

$$(x, T(x), U, G, M),$$

missä x on muuttujan nimi; $T(x)$ on x:n termijoukko eli niiden x:n kielellisten arvojen joukko, jossa jokainen arvo on sumea U:ssa määritelty luku. G on syntaktinen eli lauseopillinen sääntö siitä, miten x:n arvojen nimet määritellään. M on semanttinen eli merkitysopillinen sääntö siitä, miten jokainen arvo yhdistetään merkitykseensä. [Fullér, 1999]

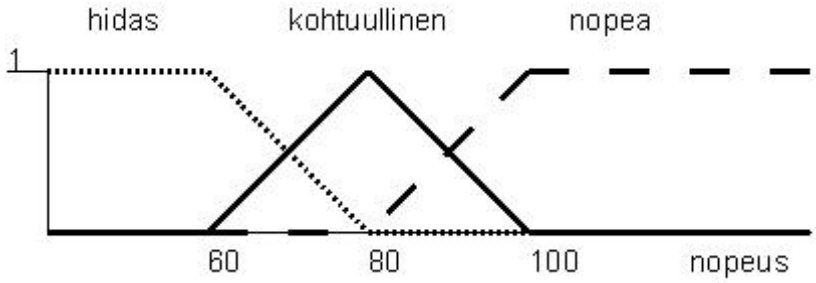
Esimerkki 6: Kielellinen muuttuja

Kielellisen muuttujan *nopeus* termijoukko *T*(*nopeus*) on

$$T = \{\ hidas, kohtuullinen, nopea\ \},$$

missä jokainen termi on kuvattu sumealla joukolla kohdealueella $U = [0, 160]$. Voimme tulkita *G*:n seuraavasti

- hidas on “nopeus, joka on alle n. 60 km/h”
- kohtuullinen on “nopeus, joka on lähellä 80 km/h:a”
- nopea on “nopeus, joka on yli n. 100 km/h”



Kuva 9. Kielellisen muuttujan *nopeus* arvot.

Nämä termit (ja viisikon *M*) voidaan kuvata sumeina joukkoina joiden jäsenyysfunktiot ovat

$$\text{hidas}(v) = \begin{cases} 1, \text{jos } v \le 60 \\ 1 - \dfrac{v - 60}{20}, \text{jos } 60 \le v \le 80 \\ 0 \text{ muuten} \end{cases}$$

$$\text{kohtuullinen}(v) = \begin{cases} 1 - \dfrac{|v - 80|}{20}, \text{jos } 60 \le v \le 100 \\ 0 \text{ muuten} \end{cases}$$

$$\text{nopea}(v) = \begin{cases} 1, \text{jos } v \ge 100 \\ \dfrac{v - 80}{20}, \text{jos } 80 \le v \le 100 \\ 0 \text{ muuten} \end{cases}$$

Kielellinen muuttuja voidaan määritellä myös ilman taustalla olevaan *terävää* (crisp) arvoaluetta, kuten nopeus tai ikä. Tällöin puhutaan muuttujan arvojen nimiöinnistä sumeilla termeillä. Esimerkiksi muuttuja *riskiluokitus*, joka voi saada arvot “matala”, “kohtuullinen” ja “suuri”, ei välttämättä tarvitse mitään numeerista tai prosentuaalista kohdealuetta, johon kielelliset arvot peilautuisivat. Jos sovelluksessa halutaan kuitenkin saada operoinnin lopputuloksena terävä arvo, esimerkiksi jonkin

laitteen säätöä varten, täytyy tuloksen muuttujille kuvata terävä kohdealue ja käyttää kiellisiä muuttujia sumeiden joukkojen kanssa.

3.2.1.1. Kielellinen aita

Kielelliseen muuttujaan voidaan liittää *kielellinen aita* (linguistic hedge), joka kasvattaa, lieventää tai rajoittaa jäsenyysastetta. Esimerkiksi termit “erittäin”, “enemmän tai vähemmän” ja “hieman” ovat kielellisiä aitoja, joista “erittäin” kasvattaa ja “enemmän tai vähemmän” sekä “hieman” lieventävät kuuluvuusastetta. Kielellinen aita voidaan tulkita yksipaikkaiseksi yksikkövälillä [0, 1] määritellyksi funktioksi. Esimerkiksi “todella” tulkitaan usein funktioksi $h(a) = a^2$ ja “melko” funktioksi $h(a) = \sqrt{a}$. Tällaista kielellisen aidan määrittelevää operaatiota kutsutaan sen *modifioijaksi* tai *määreeksi*. [Karvonen, 2000]

3.2.2. Normalisoitu sumea joukko

Sumean joukon normalisointi tarkoittaa sitä, että joukon jäsenyysfunktioon kohdistetaan muunnosfunktio, jonka avulla saadaan ainakin yksi piste, jossa jäsenyysfunktion arvo on 1 eli sumea joukko on normaali (ks. 3.2).

3.2.3. Alfa-taso

Alfa-taso (α-cut) on rajoite, jolla voidaan sumeasta joukosta rajata annettua parametria α suuremman tai yhtäsuuren jäsenyysasteen omaavat alkiot. Joukon A alfa-taso kaikille $\alpha \in (0,1]$ on terävä joukko ${}^{\alpha}A$

$$ {}^{\alpha}A = \{x \in X \mid \mu_A(x) \geq \alpha\}. $$

Alfa-tasoja käytetään mm. interpolaatiossa, joka puolestaan tarjoaa menetelmän funktion approksimointiin.

3.3. Mahdollisuusteoria

Sumeiden joukkojen teoria tarjoaa ensisijaisesti kalkyylin totuusarvoille, ei niinkään työkalua epävarmuudessa päättelyyn. Sumeita joukkoja voidaan käyttää epävarmuuden mallintamisessa silloin, kun jäsenyysfunktioita tulkitaan *mahdollisuusjakaumina* (possibility distribution). [Dubois and Prade, 2001]

Mahdollisuusteoria (possibility theory) on matemaattinen teoria epätäydellisen tiedon käsittelyyn. Zadeh [1978] esitteli sen vuonna 1978 sumeiden joukkojen teoriansa laajennuksena ja Dubois and Prade [2001] jatkoivat myöhemmin sen kehittelyä.

Epätarkka tai epämääräinen arvo voidaan määritellä attribuutin arvojen kohdealueessa määriteltynä sumeana joukkona, joka tulkitaan mahdollisuusjakaumana; jokaisen arvoalueen arvon jäsenyysaste esittää mahdollisuusastetta sille, että kyseinen arvo on juuri oikea arvo attribuutille. Mahdollisuusteoriassa sumealla joukolla on näin ollen disjunktiivinen tulkinta. [Bordogna *et al*., 1999]

3.3.1. Mahdollisuusjakauma

Olkoon joukko *K uskomuskanta* (belief base) eli joukko kaksiarvoisia lauseita, jotka uskotaan olevan tosia tai epätosia. *K* voi esimerkiksi mallintaa tekoälyn piirissä älykkään agentin tietämystä eli kaikkea sitä, minkä agentti tietää kohdealueesta, tai oikeammin sanottuna, kaikkea sitä, minkä agentti uskoo olevan totta kohdealueessa. Uskomuskanta *K* voidaan korvata joukolla *E*, jossa kaikki *K*:n lauseet on tulkittu tosiksi. *E* on *K*:n mallien joukko. *K*:n mallit edustavat mahdollisia asiantiloja, joista yksi on oikeasti voimassa. *E*:n karakteerista funktiota voidaan täten kutsua mahdollisuusjakaumaksi. Lause p on varmastikin totta, jos $E \subseteq [p]$ ja varmastikin epätotta, jos $E \subseteq [\neg p]$ ([*p*]:n komplementti) sekä täydellisesti epävarmaa, jos sekä $E \cap [p] \neq \varnothing$ että $E \cap [\neg p] \neq \varnothing$. [Dubois and Prade, 2001]

Mahdollisuusjakauma voidaan merkitä $\{ p / o, \ldots \}_\pi$, missä p on mahdollisuusaste joka on yhdistetty olioon (tai kohdealueen elementtiin) $o, \ldots$. Mahdollisuusjakaumaa ei pidä sekoittaa sumean joukon käsitteeseen. Sumea joukko mallintaa elementtien välistä eriasteisuutta, joka yhdessä voi muodostaa epätarkan käsitteen. Sillä on lähtökohtaisesti *konjunktiivinen* (conjunctive), moniarvoinen luonne sisältäen AND-semantiikan, kuten esimerkiksi sumeassa joukossa “nuori”, joka esittää kaikki iät, joiden voidaan sanoa kuuluvan nuoruuteen (tietyllä asteella). Mahdollisuusjakauma mallintaa mahdollisia vaihtoehtoja arvolle. Se esittää *disjunktiivista* (disjunctive) tietoa sisältäen XOR-semantiikan. Monissa tapauksissa mahdollisuusjakauma voidaan johtaa normalisoidusta sumeasta joukosta, kuten henkilön iän epätarkassa kuvaamisessa mahdollisuusjakaumana, joka on johdettu sumeasta joukosta “nuori”: vain yksi tämän mahdollisuusjakauman vaihtoehdoista vastaa henkilön ikää, mutta ei ole sanottu mikä niistä. [Van Gyseghem and De Caluwe, 1997]

3.4. Sumea luku

Siermala [2000] antaa sumean luvun määritelmäksi seuraavan: sumea joukko *A* on sumea luku, jos se täyttää seuraavat kolme ehtoa:

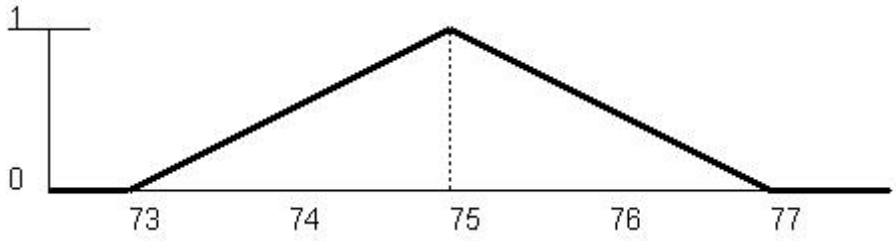
- *A*:n täytyy olla normalisoitu sumea joukko (ks. 3.2.2);
- ${}^{\alpha}A$ täytyy olla suljettu väli, kun α kuuluu välille (0,1];
- *A*:n tuki eli ${}^{0+}A$ täytyy olla rajoitettu.

Klir and Yuan [1995] esittävät lauseen, jolla voidaan kuvata jokainen sumea luku seuraavassa muodossa:

$$A(x) = \begin{cases} 1, \text{ kun } x \in [a,b] \\ l(x), \text{ kun } x \in (-\infty, a) \\ r(x), \text{ kun } x \in (b, \infty), \end{cases}$$

jossa funktio $l(x)$ on monotonisesti kasvava oikealta jatkuva funktio ja $r(x)$ vastaavasti monitonisesti vasemmalta vähenevä jatkuva funktio.

Mattila [1997] antaa sumean luvun merkintätavaksi yksinkertaisemman vaihtoehdon: sumea luku voidaan merkitä kolmiolukuna $A = (a, b, c)$, jossa luku a vastaa pistettä, jossa jäsenyysfunktio alkaa kasvamaan nollasta, luku b pistettä, jossa jäsenyysfunktio saa arvon 1 ja luku c pistettä, jossa jäsenyysfunktio laskee takaisin nollaan. Kuvassa 10 on esimerkki sumeasta luvusta "75", joka voidaan merkitä muodossa (73, 75, 77).



Kuva 10. Sumea luku "75".

3.5. Samankaltaisuusrelaatio

Sumeaa binaarirelaatiota, joka on refleksiivinen, symmetrinen ja transitiivinen, kutsutaan *similaarisuus-* eli *samankaltaisuusrelaatioksi* (similarity relation) [Kankainen, 2000]. Similaarisuusrelaatio voidaan tulkita kahdella tavalla: i) se ryhmittää alkioita teräviin joukkoihin, joiden jäsenet ovat "samankaltaisia" tietyllä vahvuudella ja ii) sen avulla voidaan tarkastella alkioiden samankaltaisuutta toisiin joukon alkioihin nähden. Sumean joukon tai kielellisen muuttujan samankaltaisuusrelaatio esitetään usein matriisina, kuten taulukossa 1 on havainnollistettu. Taulukon mukaan esimerkiksi "nuori" on samankaltainen kuin "keski-ikäinen" asteella 0,3, kun taas "nuori" ja "vanha" eivät ole lainkaan samankaltaisia.

	nuori	keski-ikäinen	vanha
nuori	1	0,3	0
keski-ikäinen	0,3	1	0,4
vanha	0	0,4	1

Taulukko 1.

3.6. Sumea logiikka

Klassisen logiikan perusperiaate, kaksiarvoisuus, määrää, että jokainen lause on joko tosi tai epätosi. Tämä kaksiarvoisuusperiaate, jonka muodosti ja jota puolusti antiikin Kreikan stoalaiset filosofi Khrysippoksen (279–207 eaa.) johdolla, kyseenalaistettiin alunperin jo epikurolaisen koulukunnan toimesta (koulukunta perustettiin n. 307 eaa.).

1900-luvun alussa puolalainen matemaatikko Lukasiewicz ehdotti kaksiarvoisuusperiaatteen tilalle yleisempää arvoisuusperiaatetta, joka määrää, että jokaisella lauseella on totuusarvo. Lukasiewiczin mukaan lauseilla voi olla totuusarvo myös toden ja epätoden väliltä [Dubois and Prade, 2001]. Sumeaa logiikkaa voidaan pitää sumeiden joukkojen sovelluksena logiikan ja päättelyn alalla.

3.6.1. Sumea päättely

Sumea päättely (*fuzzy reasoning*) perustuu klassisen ei-sumean logiikan *modus ponens* -päättelystäännön laajennukseen:

havainto: $x = A'$

tiedetään: jos $x = A$, niin $y = B$

johtopäätös: $y = B'$,

missä A' ja B' ovat sumeita joukkoja terävissä joukoissa A ja B.

Propositio jos $x = A'$, niin $y = B'$ voitaisiin esittää myös muodossa

$$\langle x, y\rangle \text{ on } R,$$

missä R on joukossa $X \times Y$ sopivasti määritelty sumea joukko. Joukko R määritellään käyttäen sumeaa implikaatiota.

3.6.2. Sumea implikaatio

Sumea implikaatio $A \rightarrow B$ on sumea relaatio $X \times Y$:ssä, joka määritellään yhtälöllä

$$(A \rightarrow B)(u,v) = I\ldots(A(u),B(v)),$$

missä I...: $[0,1] \times [0,1] \rightarrow [0,1]$.

Alla on lueteltu joitain sumeita implikaatio-operaatioita:

$I_{ZAD}(x,y) = \max(\min(x,y),1 - x)$ (Zadeh)

$I_{MAM}(x,y) = \min(x,y)$ (Mamdani)

$I_{LUK}(x,y) = \min(1,1 - x + y)$ (Lukasiewicz)

$I_{GÖD}(x,y) = \begin{cases} 1, \text{ jos } x \le y \\ y, \text{ jos x > y} \end{cases}$ (Gödel)

3.7. Pehmeä laskenta

Pehmeä laskenta (*soft computing*) on termi, joka on otettu käyttöön puhuttaessa sumeasta logiikasta, neuroverkoista ja geneettisistä algoritmeista. Nämä käsitteet sisältävät laskenta- ja päättelytapoja, jotka eroavat ns. perinteisestä "kovasta" laskennasta siten, että ne tarjoavat luonnollisemman ja lähempänä ihmisen ajattelua olevan tavan käsitellä tietoa. Tämä luonnollisuus mielletään pehmeytenä siten, että se ei aseta kovia rajoja tiedon totuutta tai attribuuttien arvoja koskien. Pehmeän laskennan työkaluja käytetään etenkin tekoälytutkimuksessa juuri siitä syystä, että ne tarjoavat

älykkyydessä mielletyksi olevan “harmauden” perinteisen mustavalkoisen päättelyn sijaan. Pehmeää laskentaa voitaisiin hyödyntää myös olio-ohjelmoinnissa muutenkin kuin tässä työssä käsitellyn sumeuden osalta.

4. Sumea oliotietomalli

Tässä luvussa paneudutaan sumean oliotietomallin teorioihin pohjautuen edellisissä luvuissa käsiteltyihin perusasioihin. Ensin kerrotaan yleisesti sumeiden oliotietomallien tutkimuksen taustoista ja siitä, millä eri tavoin sumeutta voidaan oliomalliin tuoda. Tämän jälkeen esitellään lyhyesti ja pintapuolisesti joitakin aiempia tutkimuksia, joihin kirjallisuudessa on usein viitattu. Tässä luvussa ei mennä vielä käytännön ohjelmoinnin tasolle, vaan pitäydytään teoreettisessa käsittelyssä.

4.1. Taustaa

Sumeuden ja olioparadigman yhteensovittamista on tutkittu verrattain vähän aikaa alkaen varsinaisesti vasta 1990-luvun alusta. Sittemmin aiheesta on julkaistu useita artikkeleita ja oman ehdotuksensa koskien epätarkkuuden, epävarmuuden ja sumeuden käyttöä oliotietomalleissa ovat esittäneet monet tutkijat, joista alla esitellään muutama lyhyesti ja seuraavissa kohdissa osa hieman perusteellisemmin. Tarkemman katsauksen eri tutkimuksiin sumeuden käsittelystä oliotietomallissa tarjoaa mm. Lee *et al*. [2001].

Zicari [1990] esittelee epätäydellisen tiedon käyttöä olioissa “null”-arvojen avulla.

Inoue *et al*. [1991] antavat ehdotuksen sumeiden joukkojen luokasta sekä listan useista mahdollisista tavoista toteuttaa sumeita joukkoja olioina päämääränään laitteiston ja ohjelmistojen kehittämisen sumealle tietokonejärjestelmälle.

Rossazza [1990], Dubois *et al*., [1991] ja Rossazza *et al*. [1997] esittelevät mm. sumeiden luokkien hierarkkisen mallin mahdollisuusteorian avulla.

Tanaka *et al*. [1991] pyrkivät esittelemään epätarkkuutta, epävarmuutta ja sumeutta oliotietomallin eri tasoilla ottaen huomioon sekä staattisen että dynaamisen näkökulman.

George *et al*. [1993, 1996, 1997], Aksoy *et al*., [1996] ja Yazici *et al*. [1998] ovat käyttäneet samankaltaisuusrelaatiota perustana epävarmuuden ja epätarkkuuden kuvaamiseen oliotietomallissa.

Bordogna *et al*. [1994, 1997, 1999] ja Bordogna and Pasi [2001] ovat määritelleet sumean graafiperustaisen oliotietomallin, jossa attribuutit voivat saada kielellisiä arvoja.

Van Gyseghem *et al*. [1993] sekä Van Gyseghem and de Caluwe [1994, 1995, 1996, 1997, 1998] ovat kehittäneet oliotietomallin, joka perustuu sumeisiin joukkoihin ja yleistettyihin sumeisiin joukkoihin, joiden avulla sumeus ja epävarmuus vastaavasti määritellään.

Marín *et al*. [2000, 2001] esittelevät sumean tyypin käsitteen ratkaisemaan ongelmia, joita perinteisessä terävässä oliomallinnuksessa voi esiintyä.

Edellä mainitusta aikaisempien tutkimusten ja tutkijoiden lyhyestäkin luettelosta voidaan päätellä, että sumeuden ja oliotietomallin yhteensovittaminen ei ole toistaiseksi yhdenmukaista ja vakiintuneen tavan saanutta saati standardoitua. Erilaiset lähestymis- ja tulkintatavat sekä lukuisat erilaiset sumeuden laskentamallit voivat aiheuttaa sumean oliomallin toteuttajalle valinnan vaikeuksia. Alalla on ollut yritystä saada aikaiseksi yhtenevää terminologiaa sekä yhtenäinen formaali malli sumealle olioperusteisuudelle mm. Cross *et al*.:n [1997] toimesta. Erilaiset laskentatavat ja -mallit eivät kuitenkaan sulje toisiaan pois, vaan on mahdollista käyttää tietyssä tilanteessa toista mallia ja toisessa tilanteessa toista, ja jopa yhdistellä eri malleista eri asioita. Omassa ehdotuksessani sumeaksi ohjelmointiviitekehykseksi ei sitouduta mihinkään tiettyyn malliin, vaan tarkoitus on jättää kehys mahdollisimman avoimeksi erilaisille toteutustavoille.

4.2. Yleistä

Sumeutta voidaan yleisesti ottaen tarkastella neljällä eri tasolla oliotietomallissa [Marín *et al*., 2000]:

- Attribuuttitaso (attribute level): luokan attribuuttien arvoalue voi olla sumea.
- Instanssitaso (instance relationship level): olion jäsenyys luokassa voi olla sumeasti määritelty jäsenyysasteella [0, 1].
- Periytymistaso (inheritance relationship level): luokan periytyminen yliluokasta voidaan määritellä sumeasti.
- Käyttäytymistaso (behaviour level): olioiden metodit voivat sisältää sumeutta tehden olion käyttäytymisestä sumeaa.

Seuraavissa kohdissa käsitellään lyhyesti näitä eri tasoja ja sitä, mitä ne tarkoittavat.

4.2.1. Sumea attribuutti

Olion attribuutin arvot voidaan jakaa seuraaviin kategorioihin [Berzal *et al*., 2005a]:

- Tarkat arvot: perinteiset arvoluokat, kuten numeeriset ja merkkijonoluokka.
- Epätarkat arvot: eri tyyppisiä kielellisiä termejä tai nimiöitä arvon semantiikkaan perustuen.
- Oliot: viittaukset toisiin olioihin (kompleksinen olio). Viitattava olio voi olla sumea.
- Kokoelmat: joukko olioita. Joukko voi olla sumea ja/tai sen sisältämät oliot voivat olla sumeita.

Tämän määritelmän mukaan olion katsotaan olevan sumea, jos i) sillä on attribuutteja, jotka saavat epätarkkoja arvoja tai ii) sillä on viittauksia sumeisiin olioihin tai iii) sillä on kokoelmia, jotka ovat sumeita ja/tai sisältävät sumeita olioita.

Attribuutille voidaan määritellä jäsenyysaste, joka kertoo sen, millä asteella kyseinen attribuutti kuuluu luokan ominaisuuksiin. Voidaan puhua myös tärkeysasteesta eli asteesta, joka kuvaa attribuutin tärkeyttä luokan kuvaamassa käsitteessä. Esimerkiksi luokalla `Lintu` voi olla attribuutit `lentokyky` ja `uimiskyky`. Sovelluskohteesta riippuen näille voidaan antaa erilaiset tärkeys- tai jäsenyysasteet vaikka siten, että `lentokyky` saa tärkeysasteekseen 0,8 ja `uimiskyky` 0,5. Tällöin lentotaito katsottaisiin uimataitoa tärkeämmäksi lintua kuvaavaksi piirteeksi.

4.2.2. Sumea instanssi

Ma *et al*. [2004] luettelevat neljä erilaista tapausta siitä, miten olion ja luokan suhde voi rakentua:

a) Terävä luokka ja terävä olio. Tämä on perinteinen tilanne, jossa olio joko kuuluu luokkaan täysin tai ei lainkaan.

b) Terävä luokka ja sumea olio. Vaikka luokka olisikin tarkasti määritelty ja sillä olisi täsmälliset rajat, niin siihen kuuluvat olio voi olla sumea, koska sen attribuutin tai attribuuttien arvo voi olla sumea. Tässä tapauksessa olio on suhteessa luokkaan tietyllä asteella [0, 1].

c) Sumea luokka ja terävä olio. Samoin kuin kohdassa b), olio voi kuulua luokkaan jäsenyysasteella [0, 1].

d) Sumea luokka ja sumea olio. Myös tässä tapauksessa olio kuuluu luokkaan jäsenyysasteella [0, 1].

Kohdissa b)–d) luetellut relaatiot ovat sumeita olio–luokka-relaatioita. Kohta a) voidaan myös tulkita sumean olio–luokka-relaation erikoistapaukseksi, jossa olion jäsenyysaste luokassa on 1.

4.2.3. Sumea periytyvyys

Luokan jäsenyys sen yliluokassa riippuu luokan attribuuttien semantiikasta sekä niiden arvoalueiden sisältyvyydestä yliluokan vastaaviin. Myös käsitteellinen välimatka luokan ja sen yliluokan välillä tulee ottaa huomioon. [George *et al*., 1997]

Teoreettisen ehdotuksen sumean periytyvyyden määritelmäksi ovat esittäneet monet tutkijat [esim. George *et al*., 1993, Rossazza *et al*., 1997, Dubois *et al*., 1991 ja Cross, 2002]. Käytännössä sumea periytyvyys voi tarkoittaa yksinkertaisesti eksplisiittisen jäsenyysasteen, tai periytymisen painoarvon, määräämistä luokalle sen yliluokkaan nähden. Tällöin ei järjestelmän sisäistä attribuuttien arvoalueiden tai käsitteellisten välimatkojen huomioon ottamista välttämättä tarvita, mutta käyttäjän, ohjelmoijan täytyisi itse huomioida nuo seikat jäsenyysasteen määrittelyssä. Tätä tapaa käytetään mm. Pereiran [2006] työssä (ks. 5.1).

Sumea periytyvyys vaikuttaa, paitsi itse luokkahierarkiaan, myös olioiden luokkaan kuuluvuuteen siten, että oliolla voi olla alempi jäsenyysaste yliluokassa kuin aliluokassa [Van Gyseghem and De Caluwe, 1997].

4.2.4. Sumea käyttäytyminen

Olion käyttäytyminen voi olla sumeaa, jos luokassa on määritelty metodeille jotain seuraavista:

a) Metodille on määritelty tyypillisyysjäsenyysaste (samaan tapaan kuin attribuutillekin, ks. 4.2.1), joka tarkoittaa, että kyseinen metodi on jossain määrin epätyypillinen luokalle. Jäsenyysastetta voidaan kutsua myös tärkeysasteeksi tai mahdollisuusasteeksi tilanteesta ja toteutuksesta riippuen. On myös mahdollista määritellä metodille useita erilaisia ja eri asiaa tarkoittavia jäsenyysasteita. Esimerkiksi luokan `Lintu` aliluokalla `Kanalintu` voi olla metodi `lennä`, jonka tyypillisyysaste on 0,6, tärkeysaste 0,4 ja mahdollisuusaste 0,7. Toteutuksessa voidaan käsitellä eri asteita eri tavoin niiden merkityksestä riippuen.

b) Metodi saa parametriksi sumean arvon tai sumeita arvoja, jolloin sen käytöksenkin voidaan sanoa olevan sumeaa.

c) Metodi palauttaa sumean arvon.

4.3. Rossazza *et al.*: sumeiden luokkien hierarkkinen malli

Rossazza *et al.*:n [1997] ehdotuksessa sumeaksi oliotietomalliksi avainasemassa on luokan ja *prototyypin* käsitteiden erottaminen toisistaan. Toisin kuin perinteisessä oliomallissa, heidän ehdotuksessaan luokat nähdään yleisesti ottaen kokoelmana instansseja eikä niinkään instanssien prototyyppeinä. Ehdotuksessa esitellään prototyypin käsitteen tilalle käsitettä *tyypillinen luokka* (typical class). [Rossazza *et al.*, 1997]

Olio voidaan kuvata sen attribuuttien joukolla. Attribuutilla on määritelty kohdealue, joka sisältää kaikki mahdolliset arvot, joita attribuutti voi saada. Kohdealue ei ole riippuvainen luokasta, jossa attribuutti on. Koska reaalimaailman mallinnuksessa tarvitaan usein erilaisia olioiden ominaisuuksia, on ehdotuksessa tehty ero olion *tyypillisten* (typical) ja *välttämättömien* (necessary) ominaisuuksien sekä niiden arvoalueiden välillä. Luokan `C` (välttämättömän) attribuutin `a` arvoalue R(`a`, `C`) on niiden arvojen joukko, jotka luokan `C` jäsen voi saada attribuutille `a`. Luokan attribuutin arvoalue voi olla sumea, koska jotkin arvot voivat olla *epätyypillisiä* (atypical). Tällöin epätyypillisille arvoille annetaan jäsenyysaste [0, 1).

Luokan `C` attribuutin `a` tyypillinen arvoalue T(`a`, `C`) on niiden arvojen joukko, jotka ovat enemmän tai vähemmän tyypillisiä luokan `C` jäsenelle. Attribuutin tyypillinen arvoalue on joukko, joka sisältää arvoalueen kaikki ne arvot, joiden jäsenyysaste on 1.

Tyypillinen arvoalue voi myös olla sumea joukko, mutta käytännössä siihen ei välttämättä ole tarvetta. [Rossazza *et al*., 1997]

Esimerkki 7. Attribuutin arvoalue ja tyypillinen arvoalue [Rossazza et al., 1997]

Luokalla `Linnut` on attribuutti `liikkumismuoto` (oikeammin pääliikkumismuoto, koska kyseessä on yksiarvoinen attribuutti). Attribuutin kohdealue on { `lentäminen`, `käveleminen`, `uiminen`, `ryömiminen`, … }. Arvoalue puolestaan on { `1,0/lentäminen`, `0,6/käveleminen`, `0,2/uiminen` }. Tästä saadaan tyypilliseksi arvoalueeksi {`lentäminen`}, koska sen jäsenyysaste arvoalueella on 1. Muut liikkumismuodot on katsottu luokalle jossain määrin epätyypillisiksi ja siksi niille on annettu jäsenyysaste, joka on pienempi kuin 1.

Luokan instanssi kuvataan mallissa, samoin kuten luokkakin, sen attribuuttien kautta. Instanssista `x` käytettävissä oleva tieto voi olla epätäydellistä koskien attribuuttia `a`. Tämä esitetään *mahdollisen* (possible) arvoalueen *P*(`a`, `x`) ja *uskottavan* (credible) arvoalueen *Cred*(`a`, `x`) avulla. Toisin kuin luokan arvoalue, *P*(`a`, `x`) (tai *Cred*(`a`, `x`)) on joukko toisensa poissulkevia arvoja. *P*(`a`, `x`) on (sumea) joukko, joka sisältää kaikki enemmän tai vähemmän mahdolliset attribuutin `a` arvot instanssille `x`. Arvojen oletetaan olevan varmoja, koska ne on annettu metodin avulla tai saatu periytymisen kautta. *Cred*(`a`, `x`) määrittelee arvoalueen epävarmalla tavalla (esim. vakuuttavan tuntuisen päättelytekniikan avulla, jonka validiteettia ei ole taattu). [Rossazza *et al*., 1997]

Esitetyssä mallissa on kahdenlaisia hierarkkisia linkkejä olioiden välillä. Nämä linkit vastaavat klassisia linkkejä “is a kind of” (kahden luokan välillä) ja “is a” (luokan ja olion välillä). Malli esittää “is a kind of”-linkin eli periytyvyyden, samoin kuin se klassisessa olioparadigmassa esitetään eli (attribuuttien) arvoalueiden erikoistamisena (so. aliluokan arvoalueiden täytyy sisältyä yliluokan vastaaviin arvoalueisiin). Sumeiden arvoalueiden sisältyvyyden laskennassa käytetään mm. sumeita implikaatioita (ks. 3.6.2). Mallissa esitetään myös laskentakaavat “is a”-linkin eli olion luokkaan kuulumisen määrittelemiseen olion määritykseen perustuen. [Rossazza *et al*., 1997]

Rossazza *et al*. [1997] kuvaa työssään, mitä ongelmia voi syntyä eksplisiittisesti painotetun periytyvyysmekanismin käytöstä (ks. 4.2.3):

- Painotuksen subjektiivisuus: käyttäjä ei voi helposti perustella arviotaan.
- Painotuksen validiteetti: annetun arvon täytyy loppujen lopuksi olla yhtenevä kaikkien periytyvyydessä mukana olevien luokkien kuvausten kanssa, etenkin jos uusissa aliluokissa määritellään uusia ominaisuuksia.

Tästä syystä mallissa kuvatussa ehdotuksessa ei tueta eksplisiittisesti painotettua periytyvyyttä. Sen sijaan mukana on kolme muunlaista periytyvyystyyppiä: tyypillinen periytyvyys, epätyypillinen periytyvyys ja “normaali” periytyvyys.

Määritellään luokka C, jonka tiedetään olevan jonkin verran erikoistuneempi kuin luokan D. Tämä voidaan tehdä kolmella tavalla ja sanoa:

- C:t ovat tyypillisiä D:eitä: C:n mahdolliset arvoalueet periytyvät D:n tyypillisistä arvoalueista.
- C:t ovat D:eitä: C:n mahdolliset arvoalueet periytyvät D:n mahdollisista arvoalueista ja C:n tyypilliset arvoalueet periytyvät D:n tyypillisistä arvoalueista. Tämä on "normaali" tapa.
- C:t ovat epätyypillisiä D:eitä: C:n mahdolliset arvoalueet periytyvät joko a) D:n mahdollisten arvoalueiden leikkauksesta D:n tyypillisten arvoalueiden komplementin (ks. 3.2) kanssa, jos jälkimmäinen tiedetään, tai b) D:n mahdollisista arvoalueista, jos D:n tyypillisiä arvoalueita ei tiedetä.

C:n kuvaaminen D:n epätyypilliseksi aliluokaksi on hieman mutkikkaampaa kuin kaksi muuta tapausta. Se tarkoittaa sitä, että C:n attribuuteista ainakin yksi – mutta ei välttämättä kaikki – on epätyypillinen. Täten epätyypillinen periytyvyys vaatii, että epätyypilliset attribuutit on määrätty. Kaikille näille attribuuteille C:n mahdolliset arvoalueet perivät D:n mahdollisen arvoalueen ja D:n tyypillisen arvoalueen komplementin leikkauksen. Kaikki muut attribuutit periytyvät "normaaliin" tapaan. [Rossazza *et al*., 1997]

Moniperiytymisen ongelmaan mallissa ehdotetaan seuraavaa yksinkertaista ratkaisua: aliluokan arvoalueet perivät yliluokkien arvoalueiden leikkauksen. [Rossazza *et al*., 1997]

Olio–luokka -suhteeseen malli tarjoaa samankaltaisen ehdotuksen kuin periytymiseenkin eli on kolme erilaista tapaa olion x olla luokan D instanssi [Rossazza *et al*., 1997]:

- x on tyypillinen D: x:n mahdolliset arvoalueet perivät D:n tyypilliset arvoalueet.
- x on D: x:n mahdolliset arvoalueet perivät D:n mahdolliset arvoalueet ja x:n uskottavat arvoalueet perivät D:n tyypilliset arvoalueet.
- x on epätyypillinen D: x:n mahdolliset arvoalueet perivät D:n mahdolliset arvoalueet, jos liittyviä tyypillisiä arvoalueita ei tiedetä. Muussa tapauksessa x:n mahdolliset arvoalueet perivät D:n mahdollisten arvoalueiden ja niihin liittyvien tyypillisten arvoalueiden leikkauksen.

Rossazza *et al*.:n [1997] mallissa kiinnitetään erityistä huomiota siihen, että luokkahierarkian muutokset päivittävät em. laskennallisia periytymis- ja instanssilinkkejä. Samoin uusien attribuuttien lisääminen saattaa muuttaa periytyvyys- ja luokkaankuulumissääntöjä. Malli tarjoaa algoritmit näiden muutosten laskemiseen. Olio-ohjelmoinnin näkökulmasta tällainen oliomalliin implisiittisesti kuuluva

hierarkiasuhteiden laskennallisuus tuo omat haasteensa. Nykyiset ohjelmointikielet eivät tarjoa tällaisia mekanismeja valmiina. Jos itse kieleen ei haluttaisi puuttua, olisi mahdollista luoda ohjelmointiympäristö, joka osaisi ottaa huomioon tällaiset seikat ja ylläpitää sumeaa hierarkiamallia varsinaisen oliomallin "taustalla". Toinen tapa olisi ulkoistaa hierarkian sumeus pois itse oliomallista omaan erilliseen komponenttiinsa, jonka avulla sumeutta käsiteltäisiin eksplisiittisesti.

4.4. George *et al*. ja Yazici *et al*.: samankaltaisuus sumeassa oliotietomallissa

Samankaltaisuusrelaatio (ks. 3.5) muodostaa George *et al*.:n [1997] ehdotuksessa sumean oliotietomallin perustan korvaten terävän yhtäsuuruusrelaation.

Mallissa yleistetään olio–luokka-suhteen samoin kuin luokka–aliluokka-suhteen määritelmät siten, että perinteinen terävä näkökulma on eräs erikoistapaus. Tähän ratkaisuun annetaan kaksi perustelua: ensinnäkin se sallii aiempaa tarkemman tietämyksen esittämisen, ja toiseksi se mahdollistaa entistä tehokkaamman hakumekanismin rakentamisen. [George *et al*., 1997]

George *et al*.:n [1997] työssä korostetaan sitä, että määriteltävä tietomalli säilyttää sen alla olevan loogisen paradigman sisällyttäen samalla epävarmuutta sen hierarkkisiin suhteisiin. Mallissa myös erotetaan epävarmuus ja epätarkkuus. Epätarkkuus on tiedon ominaisuus, jonka samankaltaisuusrelaatio pyrkii mallintamaan. Epävarmuus ilmenee luokkatasolla attribuuttien arvojen epätarkkuuden *koosteena* (aggregation). Kooste on sovelluksesta riippuvainen, joten mallissa huomioidaan attribuuttien merkitys sumeassa luokassa samoin kuin käsitteellinen (tai semanttinen) etäisyys luokan ja sen instanssien (tai aliluokkien) välillä. [George *et al*., 1997]

Sumea luokkahierarkia eroaa klassisesta terävästä hierarkiasta siinä, että sen luokkien mallintamien käsitteiden rajat ovat epävarmoja. Tämä epävarmuus johtuu siitä, että luokan attribuuttien arvot ovat epätarkkoja. Lisäksi on mahdollista, että tosiasialliset attribuutin arvot luokassa eroavatkin formaaleista arvoista. Koska samankaltaisuusrelaatio linkittää kohdealueen attribuutin arvot transitiivisuuden kautta, on mahdollista, että olio kuuluu luokkaan, vaikka sen attribuutin arvot eroavat formaalin arvoalueen elementeistä. Se, missä määrin nämä arvot eroavat, vaikuttaa olion jäsenyyteen luokassa ja on osa jäsenyysarvojen muodostusta. Jäsenyysarvojen määrityksessä täytyy myös huomioida em. attribuutin merkitys mallinnettavassa käsitteessä sekä käsitteellinen etäisyys yliluokan ja luokan, tai luokan ja olion, välillä. Sumeissa hierarkioissa on tärkeää ajatella käsitteellistä etäisyyttä linkkien välillä. Luokka ei välttämättä sisälly yliluokkaan, mutta voi olla vain likimääräinen aliluokka. Voi esiintyä tilanteita, joissa luokan jäsenyys sen yliluokissa lisääntyy, kun hierarkiassa noustaan ylemmäksi (vahva "is-a"). On myös mahdollista, että jäsenyys vähenee (heikko "is-a"). [George *et al*., 1997]

George *et al.*:n [1997] mallissa määritellään olion o_j jäsenyys luokassa C seuraavasti:

$$\mu_C(o_j) = \mathbf{g}[\,\mathbf{f}(RLV(a_i, C), INC(rng_C(a_i) / o_j(a_i)))\,],$$

missä $RLV(a_i, C)$ on attribuutin a_i merkitys (relevance) käsitteelle C ja $INC(rng_C(a_i) / o_j(a_i))$ on o_j:n attribuuttiarvojen sisältymisaste (degree of inclusion) formaalilla arvoalueella a_i luokassa C. Sisältymisaste määrittelee samankaltaisuusasteen nimittäjän arvon (tai arvojoukon) ja osoittajan arvon (tai arvojoukon) välillä. Funktio **f** esittää luokan attribuuttien keräymän ja **g** semanttisen linkin luonteen instanssin (olion) ja luokan/yliluokan välillä. Merkitysarvo $RLV(a_i, C)$ voidaan antaa eksplisiittisesti käyttäjän toimesta tai se voidaan laskea. Sisältymisasteen $INC(rng_C(a_i) / o_j(a_i))$ laskemiseksi annetaan useita erilaisia tapoja, joita ei tässä tarkastella lähempää. [George *et al.*, 1997]

Yazici *et al.* [1998], jotka jatkoivat George *et al.*:n [1997] työtä, esittävät, että sumeassa olioperustaisessa mallissa voi attribuutti saada useita arvoja yhtäaikaisesti ja nämä arvot voivat olla yhdistettyjä loogisilla operaattoreilla AND: ⟨…⟩, OR: {…} tai XOR: […]. Samankaltaisuusmatriisilla määritetään samankaltaisuus jokaisen kohdealueen elementin välille. [Yazici *et al.*, 1998]

Esimerkki 8. Attribuuttien määrittely [Yazici et al., 1998].

Attribuutille `ikä` on määritelty kohdealueeksi joukko { erittäin vanha, vanha, nuori, erittäin nuori, pikkulapsi } sekä taulukon 2 mukainen samankaltaisuusmatriisi.

Luokan `Lehdenjakaja` `ikä`-attribuutin arvoalue on annettu OR-semantiikalla rng(`ikä`) = { nuori, erittäin nuori }. Tällä mallinnetaan tilannetta, jossa lehdenjakajat ovat yleensä nuoria tai erittäin nuoria (tai molempia). Määritellään olio `poika` ja sille attribuutin arvo `poika.ikä` = { erittäin nuori, pikkulapsi } eli pojan ikä on joko erittäin nuori tai pikkulapsi tai kumpaakin. Nyt voidaan erityisellä sisältymisastekaavalla saada laskettua pojan sisältymisasteeksi 0,3.

	Erittäin vanha	Vanha	Nuori	Erittäin nuori	Pikkulapsi
Erittäin vanha	1	0,7	0	0	0
Vanha	0,7	1	0	0	0
Nuori	0	0	1	0,8	0,1
Erittäin nuori	0	0	0,8	1	0,3
Pikkulapsi	0	0	0,1	0,3	1

Taulukko 2. Attribuutin `ikä` samankaltaisuusmatriisi.

Taulukossa 3 on kuvattu joidenkin esimerkkiarvojen sisältymisarvot OR-semantiikalla. Vastaavasti voidaan käsitellä AND- ja XOR-semantiikan mukaisia attribuuttiarvoja niille sopivilla sisältymiskaavoilla.

Attribuutin arvo	INC
{nuori}	1
{erittäin nuori, pikkulapsi}	0,3
{erittäin nuori, erittäin vanha}	0
{nuori, vanha}	0
{vanha, pikkulapsi}	0

Taulukko 3. Attribuutin `ikä` sisältymisarvoja OR-semantiikalla.

Luokan yliluokkaan sisältymisen asteen laskennassa riittää, että luokka ja sen yliluokka jakavat ainakin yhden attribuuttiarvon keskenään. Tätä kutsutaan *likimääräiseksi* (approximate) lähestymistavaksi. Esimerkiksi luokalla `Kulkuneuvo` on attribuutti `osat`, jossa arvot ⟨ runko, moottori, istuin ⟩ (AND-semantiikalla). `Puu`, jolla myös on runko, ei ole kuitenkaan `Kulkuneuvo`. Samankaltaisuuteen perustuvassa mallissa oletetaan, että luokka–aliluokka -relaatiot on rakennettu alusta alkaen loogisesti, eikä tällaista relaatiota puun ja kulkuneuvon välille tehtäisi. On kuitenkin parempi, että malli osaa ottaa huomioon myös mielivaltaisia mallinnuksia. Esitelty malli pyrkii tähän käyttämällä em. attribuuttien samankaltaisuuksia sekä merkitys- ja sisältymisasteita. [Yazici *et al*., 1998]

Samankaltaisuudelle perustuva sumea oliomalli sellaisena, kuin George *et al*. [1997] ja Yazici *et al*. [1998] sen tutkimuksissaan esittävät, ottaa huomioon sekä sumeuden että epävarmuuden luokkahierarkioissa. Tutkimuksissa tuodaan esiin se, että luokka voi olla toisen luokan aliluokka jollakin asteella. Tämän jäsenyysasteen laskentaa varten esitellään samankaltaisuuslaskentakaavoja, joilla voidaan mitata attribuuttien arvojen läheisyys yli- ja aliluokissa. Lisäksi tutkimuksissa käytetään hyväksi luokkien ja olioiden sumeiden arvojen arvoalueita, kun lasketaan sisältymis- ja jäsenyysasteita. [Lee *et al*., 2001]

4.5. Bordogna *et al*.: sumea graafiperustainen oliotietomalli

Bordogna *et al*. [1997] ehdottavat tutkimuksissaan sumeaa olioperustaista mallia, jolla voidaan ylläpitää sekä terävää että sumeaa tietoa. Sumea tieto tarkoittaa heidän tutkimuksissaan sekä epätarkkaa että epävarmaa tietoa (ks. 3.1.1). Heidän yleistetty graafiperustainen mallinsa mahdollistaa epätarkkuuden ja epävarmuuden hallinnan olioiden attribuuttien tasolla sekä olion suhteissa muihin olioihin. [Lee *et al*., 2001]

Graafiperustaiset tietomallit tarjoavat luonnollisen tavan käsitellä tietoa sellaisissa sovelluksissa kuten hyperteksti- tai multimediatietojärjestelmät. Graafiperustaisessa oliomallissa sekä tietokantaskeema että instanssit voidaan nähdä graafeina. [Bordogna *et al*., 1997]

Graafiperustainen oliomalli on suunniteltu alunperin käsittelemään vain terävää tietoa [Bordogna *et al*., 1997]:

1. tarkkojat arvoja
2. olioiden teräviä ominaisuuksia
3. olioiden tarkkaa luokitusta
4. terävää luokkahierarkiaa

Sumeassa graafiperustaisessa oliomallissa laajennetaan alkuperäistä mallia seuraavilla tasoilla [Bordogna *et al*., 1997]:

1. epämääräisen attribuuttiarvon määritelmä
2. epävarman ominaisuuden ja linkkisuhteen määritelmä
3. voimistetun ominaisuuden ja linkkisuhteen määritelmä
4. sumean luokan määritelmä
5. sumean luokkahierarkian määritelmä

Seuraavissa kohdissa käydään nämä tasot läpi kuten Bordogna *et al*. [1997] ovat esittäneet syventymättä kuitenkaan yksityiskohtiin. Bordogna *et al*. [1997, 1999] antavat töissään myös formaalin määritelmän ja graafisen kuvaustavan mallilleen sekä kiinnittävät huomiota kyselyiden tekemiseen siinä.

4.5.1. Epämääräiset attribuuttien arvot

Attribuuttitasolla esitetään, että epämääräisiä tyyppejä luonnehditaan epämääräisillä tai epätarkoilla arvoilla, jotka esitetään mahdollisuusjakaumina. Numeerisen kohdealueen omaavat attribuutit voivat saada epämääräisen arvon kielellisen termin avulla. Esimerkiksi kielellinen muuttuja *Ikä*: *T*(*Ikä*) = { erittäin nuori, nuori, keski-ikäinen, vanha, erittäin vanha }. Kun attribuutin kohdealue ei ole numeerinen, epämääräisyys esitetään silti mahdollisuusjakaumana, mutta ei kielellisenä terminä, vaan nimiöintinä (ks. 3.2.1). Esimerkiksi epämääräinen arvo muuttujalle *Asiantuntemus* voidaan antaa mahdollisuusjakaumana { 1 / hyvä, 0,7 / ei erityisen hyvä }, jolla mallinnetaan tilannetta, jossa asiantuntemus on ennemmin hyvällä tasolla kuin ei niin hyvällä. [Bordogna *et al*., 1997]

4.5.2. Olioiden epävarmat ominaisuudet

Epävarma ominaisuus tarkoittaa sitä, että attribuutilla voi olla muitakin mahdollisia arvoja kuin linkkisuhteessa määritelty. Jos epävamuusaste $\mu = 1$, tarkoittaa se, että mikä

tahansa kohdealueen arvo on mahdollinen eli epävarmuus on täydellinen. Näin on silloin, kun varsinainen arvo on tuntematon. Jos taas epävarmuusaste esimerkiksi henkilön ominaisuudelle *ikä* on 0,3 ja sen arvo on "keski-ikäinen", niin se tarkoittaa, että kyseinen henkilö arvioidaan keski-ikäiseksi 30 %:n epävarmuudella eli on melko varmaa, että ko. henkilö on keski-ikäinen, mutta ikä voi olla jokin muukin. Jos *iän* epävarmuusaste olisikin $\mu = 0$, niin olisi täysin varmaa, että henkilö on keski-ikäinen. [Bordogna *et al*., 1997]

4.5.3. Olioiden voimistetut ominaisuudet

Bordogna *et al*.:n [1997] ehdotuksessa esitetään myös voimistettujen ominaisuuksien määritelmä. Ominaisuuden voimistamiseksi on määritelty oma, mallin sisäinen kielellinen muuttuja *Voimakkuus*: *T*(*Voimakkuus*) = { ei yhtään, erittäin alhainen, alhainen, korkea, erittäin korkea, täysi }. Esimerkiksi luokan `Päättötyö` instanssin ominaisuuteen `määräaika` liitetty voimakkuus "täysi" tarkoittaa sitä, että määräaikaa ei voida siirtää myöhemmäksi, vaan se on ehdoton. Voimakkuusasteesta voidaan puhua myös tärkeysasteena. [Bordogna *et al*., 1997]

4.5.4. Sumeat luokat

Bordogna *et al*.:n [1997] sumeassa oliotietomallissa olioiden sumea luokittelu määritellään sumeiden luokkien avulla, joita luonnehtii oliot, joiden jäsenyys luokassa on asteittaista. Koska tämä asteisuus kuvaa voimakkuuden instanssisuhteessa, käytetään sitä esittämään samaa kielellistä termiä *Voimakkuus*, jota käytettiin edellä linkkisuhteen asteisuutta kuvatessa. Instanssisuhde määritellään toisen asteen sumeana relaationa eli se saa arvoikseen sumeita joukkoja tulkittuina mahdollisuusjakaumiksi joukossa [0, 1]. Esimerkiksi tutkimusjulkaisu "Sumean olio-ohjelmoinnin teoriaa ja sovellusta" on sumean luokan `TeoreettinenJulkaisu` jäsen voimakkuusasteella *korkea* ja samalla sumean luokan `SoveltavaJulkaisu` jäsen myöskin voimakkuusasteella *korkea*. [Bordogna *et al*., 1997]

4.5.5. Sumeiden luokkahierarkioiden määritelmä

Bordogna *et al*.:n [1997] mallissa sumean luokkahierarkian määritelmä nähdään käyttökelpoisena tapana esittää epämääräisyyttä hierarkiassa, joka on tehty luokittelutarpeisiin. Sumean yliluokan esittämät epämääräiset käsitteet voidaan erikoistaa tai yleistää sumean aliluokan esittämissä epämääräisissä käsitteissä käyttäen kielellisiä määreitä (ks. 3.2.1.1), kuten "erittäin" tai "enemmän tai vähemmän". Bordogna *et al*. [1999] huomauttavat myöhemmässä tutkimuksessa, että luokan erikoistamisen käsite pitää sisällään sen, että aliluokka sisältyy yliluokkaansa: olio, joka on aliluokan instanssi jäsenyysasteella m, on myös yliluokan instanssi jäsenyysasteella $m1 > m$. Sama pätee toisinpäin luokan yleistämiseen. Esimerkiksi sumean yliluokan `TärkeäProjekti` sumea aliluokka `SeuraavaProjekti` (so. projekti, jossa seurataan

jonkin asian, esim. toisen projektin, etenemistä) erikoistaa tärkeän projektin käsitteen modifoijalla “erittäin”: `SeuraavaProjekti` on erittäin `TärkeäProjekti`. [Bordogna *et al*., 1999]

4.6. Van Gyseghem *et al*.: epävarmuus ja sumeus oliotietomallissa

Van Gyseghem *et al*. [1993] esittelevät *UFO*-mallin (Uncertainty and Fuzziness in an Object-oriented database model), jota mm. Berzal *et al*. [2005a] kehuvat yhdeksi täydellisimmistä sumean oliomallin ehdotuksista, mitä kirjallisuudessa löytyy.

UFO-mallissa lähtökohtana on olioperustaisen tietokantamallin laajentaminen tukemaan sumeutta mahdollisimman täydellisesti jokaisella mallin kerroksella käsittäen sekä staattisen että dynaamisen näkökulman. [Van Gyseghem and De Caluwe, 1997]

Mahdollisuusjakauma (ks. 3.3.1) mallintaa mahdollisia vaihtoehtoja arvolle. Se esittää disjunktiivista tietoa XOR-semantiikalla. UFO-mallissa käsitellään vain disjunktiivista tietoa. [Van Gyseghem and De Caluwe, 1998]

Seuraavissa kohdissa seurataan Van Gyseghemin ja De Caluwen [1997] artikkelin rakennetta ja esitellään ensin UFO-mallin “sumeusosio” ja sen jälkeen vastaavasti “epävarmuusosio”.

4.6.1. “Sumeusosio” UFO-mallissa

UFO-mallissa olioperustaisen tietokantamallin “sumeuttaminen” tapahtuu yksinkertaisesti kuvattuna siten, että kaikkialla missä perinteisessä oliotietomallissa on käytetty käsitettä “joukko”, korvataan tuo käsite käsitteellä “sumea joukko”, mutta tietenkin vain, jos korvaaminen nähdään tarkoituksenmukaiseksi. Mallissa löydetään seuraavat sumeat osa-alueet: i) moniarvoisten attribuuttien sumeat arvot, ii) pehmeä mallinnus (soft modeling) ja sumeat ominaisuudet, iii) sumeat instanssit, jäsenet ja luokat ja sumea erikoistaminen sekä iv) sumea periytyvyys. [Van Gyseghem and De Caluwe, 1997]

Seuraavissa alakohdissa käydään lyhyesti näitä osa-alueita läpi.

4.6.1.1. Moniarvoisten attribuuttien sumeat arvot

Moniarvoinen attribuutti mallintaa 1–N (yhden suhde moneen) tai M–N (monen suhdetta moneen) –suhdetta, joka voidaan määritellä esimerkiksi `Joukko`-luokan instanssina.

Sumea tai asteittainen 1–N tai M–N -suhde, kuten esimerkiksi henkilön ystävät tai hänen puhumansa kielet, mallinnetaan myös moniarvoisella attribuutilla, jonka arvo on kuitenkin sumea joukko. Sumean moniarvoisen attribuutin mallintamiseksi UFO-mallissa esitellään geneerinen luokka `FuzzySet`, josta johdetaan parametriluokan `Cl` avulla varsinainen luokka `FuzzySet<Cl>`. `FuzzySet<Cl>` mallintaa `Cl`-olioiden sumeaa joukkoa, jossa joukon jäsenillä on annettu tietty jäsenyysaste. [Van Gyseghem and De Caluwe, 1997]

`FuzzySet`-luokka on määritelty yleisen geneerisen kokoelmaluokan `Collection` aliluokaksi ja geneerisen, terävän kokoelmaluokan `Set` yliluokaksi, koska jokainen terävä joukko on sumean joukon erikoistapaus, jossa jäsenyysaste on joukosta { 0, 1 } [Van Gyseghem and De Caluwe, 1997].

UFO-mallissa ei sallita yksiarvoisen attribuutin arvon sumeutta, vaan sen sijaan esitellään käsite *sumea ominaisuus* [Van Gyseghem and De Caluwe, 1997].

4.6.1.2. Pehmeä mallinnus ja sumeat ominaisuudet

Sumea tieto koskien ominaisuuksia tulkitaan UFO-mallissa olion ominaisuuksien soveltuvuusasteisuutena. Pehmeä mallinnus ilmenee mallissa mahdollisuutena määritellä luokille sekä välttämättömiä että valinnaisia ominaisuuksia. Valinnaiset ominaisuudet ovat epätäydellisen erikoistamisen sijaisia. [Van Gyseghem and De Caluwe, 1997]

Esimerkiksi, vaikka ominaisuus `ui` kuuluu useimmille `Henkilö`-olioille, se ei kuitenkaan ole välttämätön, koska kaikki ihmiset eivät osaa uida. Tämä valinnainen ominaisuus voidaan kuitenkin määritellä ja toteuttaa `Henkilö`-luokassa ilman, että tarvitsisi määritellä oma aliluokka sekä henkilöille, jotka osaavat uida, että henkilöille, jotka eivät osaa. [Van Gyseghem and De Caluwe, 1997]

UFO-mallissa jokainen olio ilmoittaa luontivaiheessaan, mitkä valinnaiset ominaisuudet on määritelty sille. Valinnainen ominaisuus ei välttämättä periydy aliluokille; jos se periytyy, niin se voi periytyä joko välttämättömänä tai valinnaisena ominaisuutena. [Van Gyseghem and De Caluwe, 1995]

Sumea ominaisuus on ominaisuus, joka ei välttämättä täysin sovellu kaikille luokan jäsenille tai soveltuu niille eri asteisesti. Jokainen olio liittää *soveltuvuusasteen* (degree of applicability) jokaiseen sumeaan ominaisuuteen. Esimerkiksi `Henkilö`-olioilla on eri asteita kyvyssään toteuttaa `ui`-metodi, tai `Henkilö`:llä voi olla yksiarvoinen attribuutti `parasYstävä`, joka mallinnetaan sumeana ominaisuutena. [Van Gyseghem and De Caluwe, 1997]

Jokainen luokka määrittelee mahdolliset soveltuvuusasteet sumeille ominaisuuksilleen joko i) liittämällä pienimmän mahdollisen soveltuvuusasteen ilmaisevan kynnysarvon vaadittuun tai valinnaiseen sumeaan ominaisuuteen tai ii) eksiplisiittisesti luettelemalla kaikki mahdolliset soveltuvuusasteet sumealle ominaisuudelle [Van Gyseghem and De Caluwe, 1997].

4.6.1.3. Sumeat instanssit, sumeat jäsenet, sumeat luokat ja sumea erikoistaminen

Jotkin luokat ovat semantiikaltaan sumeita, mikä tarkoittaa, että jotkin oliot ovat tällaisen luokan instansseja tai jäseniä vain tietyllä asteella. Esimerkiksi `Henkilö`-olio

Jane on osa-aikainen opiskelija ja määriteltynä siten Henkilö-luokan täydeksi instanssiksi ja Opiskelija-luokan osittaiseksi instanssiksi:

Henkilö.ι(Jane) = 1,

Opiskelija.ι(Jane) = 0,5,

Jane.IMM-CLASS = { 1 / Henkilö, 0,5 / Opiskelija }, missä IMM-CLASS tarkoittaa olion välitöntä luokkaa (ks. 4.6.2.4). [Van Gyseghem and De Caluwe, 1997]

Aliluokka, joka määritellään määräämällä ominaisuudelle sumea rajoite yliluokkaan nähden, määrittelee yliluokan sumean erikoistamisen. Täydellinen sumea erikoistaminen määritellään sumean abstraktin yliluokan kautta eli luokan, jonka kaikki instanssit ovat myös jonkin sen aliluokan sumeita instansseja. Esimerkiksi sumeat luokat NuoretIhmiset, KeskiIkäisetIhmiset ja VanhatIhmiset määrittelevät sumean abstraktin luokan ElävätIhmiset täydellisen sumean erikoistamisen: kaikki ElävätIhmiset-oliot ovat myös ainakin yhden aliluokan jäsen tai sumea jäsen. Attribuutin Ikä kohdealue on rajoitettu aliluokissa niitä vastaavasti: nuori(Iät), keski-ikäinen(Iät) ja vanha(Iät). [Van Gyseghem and De Caluwe, 1997]

Sumealla luokalla voi olla sekä sumeita että teräviä aliluokkia [Van Gyseghem and De Caluwe, 1997].

4.6.1.4. Sumea periytyvyys

Sumean periytyvyyden yksinkertaisimmassa tapauksessa aliluokka määrittelee laajennetun (mahdollisesti sumean) raja-alueen terävälle käsitteelle, yliluokalle. Raja-aluetta kutsutaan laajennetuksi, koska se ei välttämättä sisälly terävään käsitteeseen. Sumea periytyvyysaste määrittelee ylärajan kaikkien aliluokan olioiden jäsenyysasteelle yliluokkaan nähden estäen näin aliluokkaa luomasta yliluokan täysiä jäseniä. Kaikki aliluokan oliot mukautuvat vain osittain yliluokan olioiden määritykseen. Tästä syystä tällaista periytyvyyttä kutsutaan myös *osittaiseksi periytyvyydeksi* (partial inheritance). Osittainen periytyvyys tarkoittaa myös sitä, että vain yliluokan välttämättömät “ydinominaisuudet” periytyvät aliluokan välttämättömiksi ominaisuuksiksi. Muut ominaisuudet periytyvät valinnaisiksi ominaisuuksiksi aliluokassa. Jos peritty ominaisuus määritellään uudelleen aliluokassa, niin sitä käsitellään kuitenkin samalla tavoin kuin erityisesti aliluokassa määriteltyä ominaisuutta; sumea periytyvyysaste ei vaikuta siihen onko ominaisuus soveltuva aliluokalle vai ei. [Van Gyseghem and De Caluwe, 1997]

Kuten kaikki yliluokassa välttämättömät piirteet eivät ole välttämättömiä aliluokassa, eivät myöskään kaikki yliluokassa määritellyt piirteiden rajoitteet ole voimassa aliluokilla [Van Gyseghem and De Caluwe, 1996].

Yksinkertaisimman sumean periytyvyyden tapauksessa käsitellään vain terävän yliluokan osittaista perimistä. Kun yliluokka on sumea, tai kun aliluokan olioiden jäsenyysasteen määritteleminen yliluokkaan nähden on monimutkaisempaa, on myös

metodi, jolla periytyvyysaste lasketaan, monimutkaisempi. [Van Gyseghem and De Caluwe, 1997]

4.6.2. "Epävarmuusosio" UFO-mallissa

Epätarkkaa ja epävarmaa tietoa voi löytää kaikilta oliotietomallin kerroksilta. UFO-mallissa pyritään käsittelemään epätarkkuutta ja epävarmuutta implisiittisesti ja käyttäjän kannalta näkymättömästi ilman, että tietokantaskeemaan täytyisi sitä varten tehdä muutoksia. Epätarkkuuden ja epävarmuuden käsittelyyn tarvittavat laajennukset käydään seuraavissa alakohdissa lyhyesti läpi. [Van Gyseghem and De Caluwe, 1997]

4.6.2.1. Epätarkat ja epävarmat attribuuttien arvot

UFO-mallissa epätarkka ja epävarma attribuuttiarvo mallinnetaan samaan tapaan kuin Bordogna *et al.*:n [1997] mallissa (ks. 4.5). Esimerkiksi luokalle `Henkilö` on määritelty attribuutit `ikä`, jolla primitiivinen kohdealue `Numero`, ja `puoliso`, jolla ei-primitiivinen kohdealue `Henkilö`. `Henkilö`-olio `John` voi saada seuraavat epätarkat attribuuttien arvot:

`John.ikä = ″noin 33″` (`″noin 33″` on kielellinen termi, joka ilmaisee mahdollisuusjakaumaa, joka mallintaa ikää "noin 33 vuotta vanha";

`John.puoliso = { 1 / Jane, 0,5 / Susan, 0,4 / Lucy }`$_{\pi}$. [Van Gyseghem and De Caluwe, 1998].

Van Gyseghem ja De Caluwe [1998] esittävät, että koska attribuutin on aina mahdollista saada epätarkka arvo, niin mahdollisuusjakauman käsitettä ei voida mallintaa vain yhdellä erillisellä geneerisellä luokalla `POSS(CLASS C)` (tässä ei syvennytä väitteen perusteluihin sen tarkemmin). Sen sijaan tällaisesta geneerisestä luokasta johdetut luokat `POSS(CL)` esitetään omina luokkinaan, jotka periytyvät paremetriluokasta `CL`. Luokan `POSS(CL)` instanssit ovat näin ollen myös luokan `CL` jäseniä. [Van Gyseghem and De Caluwe, 1998]

4.6.2.2. Epävarmat rooli ja roolioliot

Epätarkan ja epävarman tiedon mallinnukseen UFO-malli tarjoaa käsitettä *rooliolio* (role object). Oliolla on usein monia rooleja. Terävässä oliotietomallissa olion rooli esitetään joko (terävänä tai varmana) suhteena toiseeen olioon, kuten "Jane on Johnin puoliso": `John.puoliso = Jane`, tai suhteena olioiden luokkaan, kuten "John on opiskelija": `John` on luokan `Opiskelija` instanssi.

Olio voi esiintyä epävarmassa roolissa joko sen johdosta, että sen jollain attribuutilla on epätarkka tai epävarma arvo, tai olion epätarkan tai epävarman luokittelun takia. Epävarma rooli UFO-mallissa on mallinnettu olioon liitetyillä rooliolioilla, joissa olion tilan epävarmat muutokset tapahtuvat. Jälkeenpäin roolioliot

voidaan sulauttaa varsinaiseen olioon tai poistaa riippuen siitä, ovatko epävarmat roolit enää todellisuutta vastaavia. [Van Gyseghem and De Caluwe, 1998]

Jokaista viittausta varten mahdollisuusjakaumaoliosta `pdo` olioon `o` luodaan rooliolio `ro`. Sen rakenne ja käyttäytyminen on sama kuin `o`:lla samoin kuin alkutila, mutta kaikki epävarmat muutokset joita `pdo` kutsuu, toteutetaan vain `ro`:ssä (eikä `o`:ssa). Kutsu, joka lähetetään `pdo`:lle ja jossa pyydetään tilan muutosta, ohjataan välittömästi `ro`:lle. Mahdollisuusjakaumaolio `pdo` ei täten viittaa "pääolioon" `o` itseensä, vaan vastaavaan rooliolioon `ro`. Jokainen varma muutos, joka pyydetään suoraan `o`:lta, tehdään myös jokaiselle `o`:hon liitetylle roolioliolle. Olioihin liitetyn rooliolion käsite, kuten se on UFO-mallissa esitetty, vastaa reaalimaailman varjon käsitettä [Van Gyseghem and De Caluwe, 1998]:

Reaalimaailma: Valon lähteet, esineet, varjot.

UFO: Mahdollisuusjakaumaoliot, oliot, roolioliot.

Esimerkiksi mahdollisuusjakaumaolio, joka esittää arvon `John.puoliso = { 1 / Jane, 0,5 / Susan, 0,4 / Lucy }`$_{\pi}$, luo uudet roolioliot `Jane-1`, `Susan-1` ja `Lucy-1` ja yhdistää ne `Henkilö`-olioihin `Jane`, `Susan` ja `Lucy`. Näillä rooliolioilla on alkuaan sama sisäinen tila, kuin pääolioilla. Kun lähetetään viesti "vähennä viisi vuotta Janen iästä", niin sen tuloksena muutetaan sekä olion `Jane` että olion `Jane-1` attribuutin `ikä` arvoa. Kun lähetetään viesti "vähennä viisi vuotta `John.puoliso`:n iästä", niin tuloksena muutetaan rooliolioiden `Jane-1`, `Susan-1` ja `Lucy-1` attribuutin `ikä` arvoa. [Van Gyseghem and De Caluwe, 1998]

4.6.2.3. Ominaisuuksien epävarma soveltuvuus

Kaikki luokassa määritellyt välttämättömät ominaisuudet ovat aina välttämättömiä kaikille luokan instansseille. Voi kuitenkin olla epävarmaa, soveltuuko jokin (tai mikään) valinnaisista luokan ominaisuuksista jollekin instanssille. Valinnaisen ominaisuuden soveltuvuuden epävarmuus esitetään epävarmana totuusarvona, joka on mahdollisuusjakauma joukossa { 0, 1 } (mikä on sama kuin { epätosi, tosi }). Olion, josta ei olla varmoja, soveltuuko jokin valinnainen ominaisuus sille vai ei, mallintaminen voi hyötyä edellä kuvatusta mekanismista, jossa rooliolioita luodaan mallintamaan olion epävarmoja rooleja. [Van Gyseghem and De Caluwe, 1998]

4.6.2.4. Epävarmat instanssit

Voi olla tilanne, että jotakin oliota koskeva epävarma tieto estää olion tarkan luokituksen, eikä yhden luokan asettaminen olion välittömäksi luokaksi ole mahdollista. Tällaista oliota kutsutaan UFO-mallissa *epävarmaksi instanssiksi* (uncertain instance) [Van Gyseghem and De Caluwe, 1998].

Oliotietomallissa jokaisen olion täytyy kuulua ainakin yhteen luokkaan ja on oltava jonkin luokan instanssi. Tätä varten jokaiselle epävarmalle instanssille täytyy osoittaa

ainakin yksi luokka, jonka on täysin mahdollista olla olion välitön luokka. Tätä tilannetta varten UFO-mallissa määritellään mahdollisuusjakauma mahdollista välitöntä luokkaa varten. Mallissa määritellään edelleen olion instanssi- ja jäsenyysasteet mahdollisen välittömän luokan suhteen epävarmoina totuusarvoina. Esimerkiksi, jos on epävarmaa, onko John opiskelija vai assistentti, niin olio `John` mallinnetaan epävarmana instanssina johon liitetään instanssiasteet:

$$\text{Opiskelija}.\iota(\text{John}) = \{\ 1\ /\ 1,\ 0{,}5\ /\ 0\ \}_{\pi}$$

$$\text{Assistentti}.\iota(\text{John}) = \{\ 0{,}5\ /\ 1,\ 1\ /\ 0\ \}_{\pi}$$

ja välittömäksi luokaksi saadaan $\text{John.IMM-CLASS} = \{\ 1\ /\ \text{Opiskelija},\ 0{,}5\ /\ \text{Assistentti}\ \}_{\pi}$. [Van Gyseghem and De Caluwe, 1998]

4.7. Marín *et al*.: sumeat tyypit

Marín *et al*. [2000, 2001] tarjoavat uuden näkökulman sumean luokkaan liitetyn tyypin esitykseen. He määrittelevät *sumean tyypin* (fuzzy type) käsitteen parina rakenteellisia ja käytöksellisiä komponentteja. *Rakenteellinen komponentti* on sumea joukko, joka sisältää kaikki mahdolliset tietomallin attribuutit. Sumean joukon jäsenyysfunktion antaa ohjelmoija, joka määrittelee tyypin. *Käytöksellinen komponentti* on puolestaan sumea joukko, joka sisältää kaikki mahdolliset tietomallin metodit. Sen sumean joukon jäsenyysfunktio saadaan laskennallisesti metodien sisältämistä viittauksista. Sekä periytymisen että ilmentymisen mekanismit on muunnettu hyödyntämään sumeita tyyppejä. Ilmentymisen mekanismin voidaan sallia valitsevan tyypin attribuuttien alfa-tason (ks. 3.2.3) ilmentääkseen oliota. Periytymisen mekanismin voidaan sallia rakenteen ja käytöksen perimisen aliluokissa. Periytymiselle määritellään kaksi tyyppiä: periytyminen ilman epämääräisyyden lisääntymistä ja periytyminen epämääräisyyden lisääntymisen kanssa. [Lee *et al*., 2001]

4.7.1. Sumean tyypin määritelmä

Sumea rakenne on siis sumea joukko kaikista mahdollisista mallin attribuuteista. Sumea tyyppi on tyyppi, jonka rakenteellinen osa *S* on sumea rakenne. Niiden attribuuttien joukko, joita voidaan käyttää kuvaamaan tyyppiä minä ajan hetkenä tahansa, on tyyppiin liitetyn sumean joukon *tukijoukko* (support set). *Ydinjoukko* (kernel set) sisältää tyyppiä kuvaavat perusattribuutit. Tarkkuusaste, jolla tyyppiä voidaan tarkastella, määritellään alfa-tasoilla. On huomattava, että alfa-tason konkreettinen arvo ei välttämättä ole tärkeä, kun halutaan ainoastaan organisoida rakenne tiettyyn määrään tarkkuustasoja jakaen. Alfa-tasoa voidaan toisaalta käyttää myös ilmaisemaan tyypin jokaisen attribuutin merkitystaso, ja tällöin tason tarkkuusarvolla, sekä sillä miten tarkkuusarvot jakautuvat, on merkitystä. [Marín *et al*., 2001]

Esimerkki 9. Sumean tyypin rakenne [Marín et al., 2001].

Tarkastellaan digitaalisen kuvan käsitettä kolmella tarkkuustasolla:

- Minimiominaisuudet: teema, tiedosto, formaatti ja versio.
- Ensimmäinen tarkkuustaso: vaakaresoluutio, pystyresoluutio ja väripaletti.
- Toinen tarkkuustaso: histogrammi, rajat, kaistat ja kierteisyys.

Tämä rakenne voidaan esittää seuraavalla sumealla joukolla:

- S = { 1 / teema, 1 / tiedosto, 1 / versio, 0,9 / pystyres., 0,9 / vaakares., 0,9 / väripaletti, 0,8 / histogrammi, 0,8 / rajat, 0,8 / kaistat, 0,8 / kierteisyys }.

Näin saadaan seuraavat asiaankuuluvat alfa-tasot:

- S_1 = { teema, tiedosto, versio }.
- $S_{0,9}$ = { teema, tiedosto, versio, vaakares., pystyres., väripaletti }.
- $S_{0,8}$ = { teema, tiedosto, versio, vaakares., pystyres., väripaletti, histogrammi, rajat, kaistat, kierteisyys }.

Kun käyttäjä haluaa luoda luokan `Kuva` instanssin, hän voi tehdä sen sisältämään joko S_1:n, $S_{0,9}$:n tai $S_{0,8}$:n attribuutit riippuen siitä, minkä tarkkuustason luotava olio vaatii.

Attribuuttien epämääräisyyden määritys vaikuttaa myös luokan käyttäytymisen määritykseen. Samoin, kuten attribuuteille, täytyy myös jokaiselle metodille määritellä tarkkuusaste. Täytyy myös määritellä pienin tarkkuusaste, joka instanssilla täytyy olla, jotta tietty metodi sisältyisi sen käyttäytymiseen. Sumean tyypin tai luokan käytöksellinen komponentti on sumea joukko kaikkien mallissa määriteltyjen metodien joukosta. On tärkeää huomata, että käytöksellinen komponentti ei lisää mitään uutta tarkkuuden astetta. Se vain täydentää rakenteellisen komponentin aiemmin määrittelemiä tarkkuusasteita. On kuitenkin mahdollista antaa tyypin ohjelmoijan asettaa metodin tarkkuusaste alemmaksi, kuin mitä sen toiminta edellyttäisi. Tämä voi olla hyödyllistä tilanteessa, jossa ohjelmoijan tarkoituksena on esimerkiksi estää joiltakin instansseilta kyseisen metodin käyttö. [Marín *et al*., 2001]

4.7.2. Sumeiden tyyppien ilmentyminen ja periytyminen

Marín *et al*.:n [2000, 2001] sumean tyypin ehdotuksessa todetaan, että muutos tyypin käsitteessä sisältää myös muutoksen ilmentymisen ja perytymisen käsitteissä. Ilmentymismekaniikan tulee sallia kaikkia uusia olioita varten ominaisuuksien alfa-tason valinta. Periytymismekanismin tulee sallia luokan rakenteen ja käyttäytymisen osittainen periytyminen aliluokissa. Esitetään kaksi erilaista perytymistapaa [Marín *et al*., 2001]:

- Periytyminen ilman epämääräisyyden lisääntymistä: perityt attribuutit ja metodit sisällytetään aliluokan rakenteelliseen ja käytökselliseen ydinkomponenttiin. Tällä tavoin eliminoidaan perittyjen ominaisuuksien sumeus. Voitaisiin esimerkiksi luoda em. `Kuva`-luokan aliluokka `Kaavio`, johon sisältyisi `Kuva`-luokan kahden ensimmäisen tason ominaisuudet, mutta nyt tarkkuusasteella 1.
- Periytyminen epämääräisyyden lisääntymisen kanssa: säilytetään sumeus perimällä sekä attribuutit että metodit vastaavan jäsenyysasteen mukaisesti. Voidaan esimerkiksi luoda biologisia kuvia varten luokka `BioImage` asteella 0,8. Luokka voisi olla yliluokkansa tapaan organisoitu kolmeen tarkkuustasoon, mutta lisäksi sillä voisi olla juuri kyseistä luokkaa kuvaavia uusia ominaisuuksia jokaisella perityllä tasolla.

4.7.3. Sumean tyypin esittäminen terävässä oliomallissa

Marín *et al*.:n [2001] ehdottama sumea tyyppi voidaan esittää perinteisessä olioperustaisessa mallissa käyttäen luokkahierarkiaa, joka ei haaraudu eli jossa jokaisella luokalla on korkeintaan yksi aliluokka. Hierarkian luokat vastaavat eri sumean tyypin tarkkuustasoja. Jokaista relevanttia alfa-tasoa kohti luodaan hierarkiaan terävä luokka. Tällä luokalla on sitä vastaavan sumean tyypin tarkkuusasteen mukainen rakenne ja käyttäytyminen. Hierarkian juuriluokalla on ydinominaisuudet. Sen aliluokka perii ominaisuudet aivan kuten perinteisessä oliomallissa ja lisää niiden joukkoon omat, kyseiseen tarkkuustasoon liittyvät, ominaisuudet. [Marín *et al*., 2001]

Tällä tavoin rakennettu malli luo graafin, jossa on kaksi kerrosta: i) sumea kerros, jossa ovat sumean rakenteen omaavat luokat, jotka esittävät mallinnettavan ongelman käsitehierarkiaa ja ii) terävä kerros, joka on sumean kerroksen “alla” sisältäen terävät luokat, joilla esitetään sumean kerroksen luokkien eri tasot. Terävän tason tulisi olla käyttäjälle läpinäkyvä. [Marín *et al*., 2001]

Marín *et al*. [2001] esittävät myös ehdotuksen sumean mallin toteutuksesta ilmentymis- ja periytymisalgoritmeineen.

5. Sumea olio-ohjelmointi

Varsinaista sumeaa olio-ohjelmointia on alettu tutkia vasta hiljattain sumeiden oliotietomallien tutkimuksen myötä. Sumean oliotietomallin ohjelmointia nykyisillä olio-ohjelmointikielillä ovat tutkineet mm. Pereira [2006] ja Berzal *et al*. [2003, 2005a, 2005b, 2007a, 2007b]. Seuraavissa kohdissa tarkastellaan hieman tarkemmin näiden kahden tutkimusten tuloksia.

5.1. Pereira: sumea oliomalli laajennetussa Java-kielessä

Pereira [2006] esittää tavan, jolla laajennettua Java-kieltä voitaisiin käyttää yhdessä NRC FuzzyJ[8] –kirjaston kanssa ohjelmoitaessa sumeita sovelluksia. Java-kielen laajennuksella mallinnetaan sumeaa periytymistä ja FuzzyJ –kirjaston avulla saadaan tuotettua epämääräisyyttä (sumeutta) ja epävarmuutta luokan attribuutteihin. Kirjasto sisältää seuraavat tärkeimmät luokat sumeuden käsittelyyn [Pereira, 2006]:

`FuzzyVariable`: Sumean käsitteen, kuten lämpötila tai paine, esittämistä varten oleva luokka. Luokan instanssin luomista varten täytyy antaa seuraavat parametrit: nimi, arvoalueen ala- ja yläraja sekä arvojen mittayksikkö. Esimerkiksi:

```
FuzzyVariable temp = new FuzzyVariable(
          "temperature", 0, 100, "C");
```

`FuzzySet`: Tämän luokan oliot mahdollistavat sumeiden joukkojen kuvaamisen. `FuzzyVariable`-luokalla on metodi `addTerm`, jolla lisätään sumea termi arvoalueelle. Esimerkiksi:

```
temp.addTerm("Cold",
          new TrapezoidFuzzySet(0.0, 0.0,
                    5.0, 15.0));
```

`FuzzyValue`: luokka jonka instanssi liittää `FuzzySet`-olion `FuzzyVariable`-olioon. `FuzzyValue` tarjoaa yleensä, muttei aina, kielellisen ja merkityksellisen ilmentymän `FuzzySet`:lle. Esimerkiksi:

```
Room room = new Room();
room.setTemperature(new FuzzyValue(temp,
          "cold"));
```

Attribuuttitason epävarmuuteen terävälle attribuutille Pereira [2006] ehdottaa esimerkin 10 mukaista tapaa. Sumeiden attribuuttien kohdalla epävarmuusaste muodostuu toisin. Niillä aste annetaan *lambda-puolisuunnikkaan* (λ-trapezium) [Buisson *et al*., 1987] avulla. Esimerkki 11 kuvaa tätä tapausta.

Esimerkki 10. Terävän attribuutin epävarmuus [Pereira, 2006].

Oletetaan, että terävälle attribuutille halutaan määritellä epävarmuus- tai mahdollisuusaste. Esimerkiksi `Henkilö`-luokan oliolla `wilmer` on attribuutti `secondLanguage`, jolla on arvona `English`. Tälle halutaan määritellä epävarmuus-/mahdollisuusasteeksi 0,6. Tämä tarkoittaa sitä, että mahdollisuus sille, että Wilmerin

[8] National Research Council Canada FuzzyJ Toolkit for Java(tm) Platform. http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJDocs/index.html

toisena kielenä on englanti, on 60 %. Laajennetulla Java-kielellä tämä tapahtuisi seuraavasti:

```
(unc 0.6) wilmer.secondLanguage = English;
```

Vastaavasti epävarmuusasteen kysymiseksi oliolta kutsutaan metodia `unc` seuraavasti:

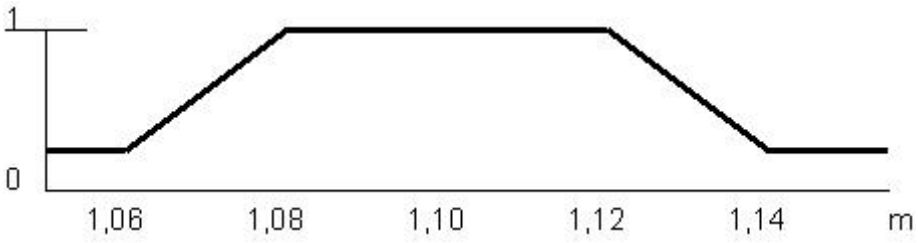
```
unc(wilmer.secondLanguage);
```

Epävarmuusaste 1,0 tarkoittaa sitä, että attribuutin arvon mahdollisuus on 100 %. Tällöin merkintä `unc` voidaan jättää pois.

Esimerkki 11. Sumean attribuutin epävarmuus [Pereira, 2006].

Olkoon `Henkilö`-olio `nathalie`, jonka `height`-attribuutti saa arvoksi kielellisen arvon "noin 1 m 10 cm". Suunnittelija antaa tälle 30 %:n mahdollisuuden. Toisin sanoen, suunnittelija haluaa mallintaa tilannetta, jossa on 30 %:n mahdollisuus sille, että Nathalien pituus on noin 1 m 10 cm. Puolisuunnikas, joka kuvaa arvon "noin 1 m 10 cm", on kuvan 11 mukainen. Kuvassa 12 on puolestaan arvon 30 %:n mahdollisuuden lambda-puolisuunnikas.

Kuva 11. Sumean arvon "noin 1 m 10 cm" puolisuunnikas.



Kuva 12. Sumean arvon "noin 1 m 10 cm" lambda-puolisuunnikas.

Laajennetussa Java-kielessä Nathalien pituuden epävarmuuden kuvaus tehdään seuraavasti:

```
Person nathalie = new Person();
FuzzyValue h = null;
FuzzyVariable height = new FuzzyVariable(
          "height", 0.0, 230.0, "cm");
height.addTerm("about_110",
```

```
            new TrapezoidFuzzySet(106.0,
                        108.0, 112.0, 114.0));
(iunc 0.3) h = new FuzzyValue(height,
            "about_110");
nathalie.setHeight(h);
```

Erotuksena tarkkojen attribuuttien `(unc)`-merkinnälle käytetään sumeiden arvojen kohdalla `(iunc)`-merkintää, koska tarkan attribuutin oletusarvo epävarmuusasteelle on 1,0 (jolloin merkinnä `(unc)` voi siis jättää pois), kun taas sumean attribuutin oletusarvo epävarmuusasteella on 0. Tämä on seurausta siitä miten lambda-puolisuunnikas lasketaan.

Sumean periytyvyyden Pereira [2006] kuvaa laajennetussa Java-kielessä siten, että luokalle voidaan määritellä epävarmuusaste, jolla se perii yliluokan. Moniperiytymisen sumeus voidaan määritellä vastaavasti antamalla jokaiselle toteutettavalle rajapintaluokalle (Javassahan ei ollut tukea moniperiytymiselle tavallisista luokista) periytymisaste, jolla aliluokka rajapinnat toteuttaa. Esimerkiksi luokka `RoundedSquare` periytyy yliluokasta `Shape` ja toteuttaa sekä `Circle` että `Square` rajapinnat. Tällä mallinnetaan tilannetta, jossa olisi kulmistaan pyöristettyjen neliöiden luokka, joka on ensinnäkin “muoto”-luokan aliluokka, ja toisekseen sekä ympyrä että neliö. Pyöristettyjen neliöiden luokka kuuluu “ympyrä”-luokkaan jäsenyysasteella 0,8 ja “neliö”-luokkaan asteella 0,6. Laajennetussa Java-kielessä tämä toteutettaisiin seuraavasti:

```
public class RoundedSquare extends Shape (imem 1.0) implements Circle
(imem 0.8), Square (imem 0.6) {…}
```

Jäsenyysaste 1,0 tarkoittaa sitä, että aliluokka periytyy täysin eli terävästi yliluokasta. Tällöin periytymisen epävarmuusastetta `(imem)` ei tarvitse merkitä.

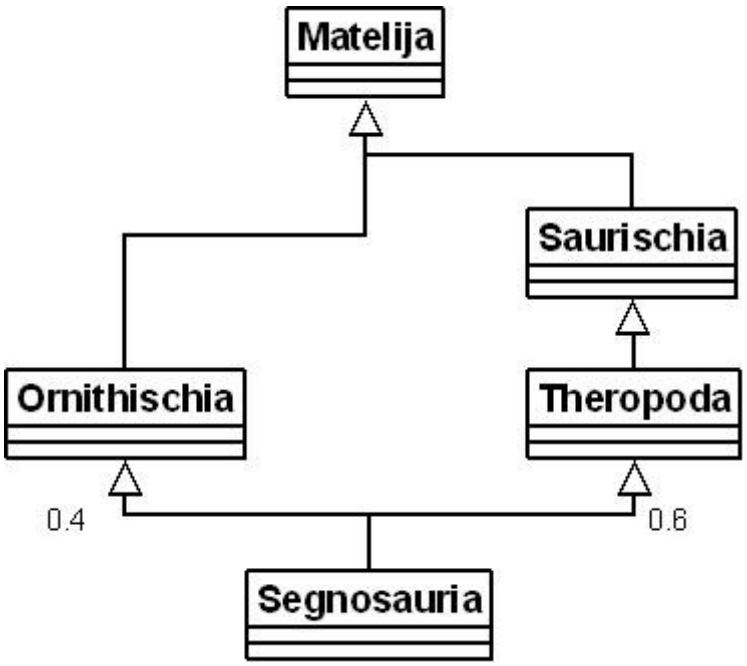
Pereiralla [2006] on myöskin ehdotus sumean luokkaan kuulumisen mallintamiseen. Tämä tapahtuu antamalla olion luonnin yhteydessä jäsenyys- tai mahdollisuusaste, jolla olio kuuluu luokkaan:

```
Dinosaur dino = new Dinosaur() (omem 0.5);
```

Hierarkkisen periytymisen ja moniperiytymisen tuomat olion monet eri yliluokat ja olion niihin kuuluminen voidaan laskea Pereiran [2006] ehdotuksessa myös automaattisesti. Olkoon esimerkiksi kuvan 12 mukainen luokkahierarkia, joka kuvaa osaa dinosaurusten lajihierarkiasta sen mukaisesti, mitä fossiileista on saatu selville. Luokan `Segnosauria` instanssin `sigy` uskotaan olevan segnosaurus 50 %:n todennäköisyydellä:

```
Segnosauria sigy = new Segnosauria() (omem 0.5);
```

Pereira [2006] ehdottaa laskentatapaa, jolla saadaan mitattua `sigy`:n mahdollisuus kuulua ornithischia- ja theropoda-lahkoihin. Laskennassa otetaan huomioon sekä olion itsensä että sen luokan periytymissuhteen mahdollisuusasteet. Laskennan tuloksena saataisiin tässä esimerkissä `sigy`:n mahdollisuusasteeksi luokassa `Ornithischia` 0,4 ja luokassa `Theropoda` 0,5. Tässä ei tarkemmin syvennytä käytettyihin laskentakaavoihin.



Kuva 13. Ote dinosaurusten luokitteluhierarkiasta.

Pereiran [2006] ehdotuksessa mallinnetaan attribuuttien ja metodien epävarmuutta luokan instansseissa siten, että ohjelmoija antaa luokan ominaisuudelle kynnysarvon, jonka mukaisesti olio joko saa tai ei saa käyttää kyseistä ominaisuutta. Ehdotuksessa käytetään alfa-tasoja tämän laskentaan [Pereira, 2006].

5.2. Berzal *et al.*: sumeuden ohjelmointi moderneissa olio-ohjelmointikielissä

Tämän työn kannalta ehkä kiinnostavimmat tutkimusaiheet ja -tulokset löytyvät Berzal *et al.*:lta [2003, 2005a, 2005b, 2007a, 2007b]. He ovat tutkineet runsaasti sumeuden ohjelmointia nykyisissä olio-ohjelmointikielissä. Tutkimuksissa on kiinnitetty erityistä huomiota modernien olio-ohjelmointikielten tarjoamiin mekanismeihin, kuten reflektiivisyyteen (ks. 2.3) ja metadatan käyttöön. Metadatalla tarkoitetaan sellaista tietoa, joka kuvaa jotain toista tietoa. Esimerkiksi luokan määritelmä on luokan metadataa. Ohjelmointikielissä, kuten C#[9] ja Java, on mekanismeja, joilla ohjelmoija voi kuvata ohjelmakoodissa luokille ja luokan piirteille erilaista metadataa.

Berzal *et al*. [2003] kuvaavat kirjoituksissaan yleisen viitekehyksen, jonka avulla voidaan suunnitella sumea oliotietomalli käyttäen perinteisiä olio-ohjelmointikieliä.

[9] http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx

Avainasemassa tässä on Marín *et al*.:n [2000, 2001] aiemmin esittelemä käsite *sumea tyyppi* (ks. 4.7).

Sumea tyyppi mallinnetaan Berzal *et al*.:n [2003] viitekehyksessä abstraktina perusluokkana `FuzzyObject`, josta kaikki sumeita ominaisuuksia tarvitsevat luokat periytyvät. `FuzzyObject`:n eräs ominaisuus on metodi `fuzzyEquals`, jolla voidaan verrata luokan instanssia toiseen olioon samaan tapaan kuin Javan perusluokan `java.lang.Object`:n `equals`-metodi, joka palauttaa normaalin terävän boolean-arvon `true`/`false`. Marín *et al*. [2003] esittelivät aiemmin ehdotuksen kompleksisten olioiden vertailuun sumeassa kontekstissa ja tätä ehdotusta Berzal *et al*. [2003] hyödyntävät `fuzzyEquals`-metodin toteutuksessa. Pääpiirteissään toteutus toimii siten, että samankaltaisuus lasketaan ko. luokan attribuuttien perusteella käyttäen hyväksi refleksiivisyyttä. Tällainen ratkaisu mahdollistaa sen, ettei viitekehystä käyttävän ohjelmoijan tarvitse itse ohjelmoida samankaltaisuuden laskentaa. Riittää, että hän toteuttaa oman oliomallinsa sumeat luokat periyttämällä ne `FuzzyObject`-luokasta tai `FuzzyObject`:n eri tyyppisiä epätarkkoja kohdealueita mallintavista abstrakteista aliluokista. Menetelmä on mitä mainioin tilanteissa, joissa oliomalli voidaan toteuttaa periyttämällä viitekehyksen luokista, mutta ei sovellu sellaisiin tilanteisiin, joissa oliomallin tulisi olla ns. puhdas eli ilman periytyvyyssidoksia mallin ulkopuolisiin luokkiin. Lisäksi olemassaolevaa oliomallia on – mallin monimutkaisuudesta riippuen – hankalaa muuttaa jälkikäteen lisäämällä siihen `FuzzyObject`-periytyvyyksiä.

Berzal *et al*. [2003] lisää viitekehykseensä myös rakenteen luokan attribuuttien tärkeyden käsittelemiseksi. Tärkeysarvoa käytetään `fuzzyEquals`-metodissa samankaltaisuuden laskemiseksi. Tärkeysarvojen esittäminen tapahtuu viitekehyksen ensimmäisissä versioissa [Berzal *et al*., 2003] staattisella metodilla, joka saa parametrikeen attribuutin nimen ja palauttaa attribuutin tärkeysarvon, luvun välillä [0, 1]. Myöhemmin viitekehyksen kehittyessä, esitellään tärkeysarvon kuvaamiseksi C#-kielen metadatarakennetta [Berzal *et al*., 2005b]. Tällöin tärkeysarvot voidaan asettaa elegantimmin ilman ylimääräistä metodia.

Esimerkki 12. `FuzzyObject` *[Berzal et al., 2005b].*

Berzal *et al*. käyttävät artikkeleissaan samaa opiskeluaiheista esimerkkiä, jossa on `Room`-luokka ja `Student`-luokka, jotka molemmat periytyvät `FuzzyObject`-luokasta. Alla on kuvattu luokat attribuutteineen Java-kielellä toteutettuina (muu toteutus on jätetty pois). Luokkien attribuuttien tärkeysaste on kuvattu metadatalla, Java-kielen *annotaatiolla* (annotation) `@FuzzyImportance`. Alkuperäisessä esimerkissä [Berzal *et al*., 2005b] on käytetty C#-kieltä, joka sisältää vastaavan ominaisuuden.

```java
/**
 * Sumea luokka joka kuvaa luokkahuoneen, jolla on attribuutit
 * laatu, tila (tai ala), kerros ja opiskelijat,
 * jotka opiskelevat ko. luokkahuoneessa.<br>
 * Attribuuttien tärkeysaste on annettu
 * {@link FuzzyImportance}-annotaatiolla.
 *
 * @author teemu
 *
 */
public class Room extends FuzzyObject {

    /**
     * laatu
     */
    @FuzzyImportance(0.5f)
    private Quality quality;

    /**
     * tila
     */
    @FuzzyImportance(0.8f)
    private Extension extension;

    /**
     * kerros
     */
    @FuzzyImportance(1.0f)
    private Floor floor;

    /**
     * opiskelijat
     */
    @FuzzyImportance(1.0f)
    private StudentCollection students;

}
```

```java
/**
 * Sumea luokka, joka kuvaa opiskelijaa,
 * jolla on attribuutit nimi, ikä ja pituus.<br>
 * Attribuuttien tärkeysaste on annettu
 * {@link FuzzyImportance}-annotaatiolla.
 *
 * @author teemu
 *
 */
public class Student extends FuzzyObject {

    /**
     * nimi
     */
    @FuzzyImportance(1.0f)
    private String name;

    /**
     * ikä
     */
    @FuzzyImportance(0.75f)
    private Age age;
```

```
    /**
     * pituus
     */
    @FuzzyImportance(0.75f)
    private Height height;

}
```

Berzal *et al.*:n töissä ei ole kiinnitetty juurikaan huomiota periytymiseen tai olioiden välisiin suhteisiin muuten kuin em. samankaltaisuuden laskennan kannalta. Näiltä osin heidän mallinsa onkin hieman vajavainen.

6. Sumea ohjelmointirajapinta Java-kielellä

Edellisissä kohdissa hyvin suppeasti esiteltyihin tutkimuksiin pohjautuen laaditaan nyt oma ehdotus sumeaksi ohjelmointiviitekehykseksi Java-kielellä. Viitekehyksen kuvaamisessa ei oteta tässä huomioon sen sisäistä toiminnallisuutta eli varsinaista laskentaa, vaan esitellään vain viitekehyksen käyttäjälle, ohjelmoijalle näkyvät osat. Toiminnallisuuden suunnittelu ja rakentaminen sekä ehdotuksen laajentaminen yleensäkin voisivat olla jatkotutkimuksen aiheita.

Sen sijaan, että käytettäisiin Berzal *et al.*:n [2003] ehdottamaa `FuzzyObject` (ks. 5.2) -tyylistä luokkarakennetta, on omassa ehdotuksessani lähtökohtana oliomallin pitäminen mahdollisimman puhtaana periytyvyysriippuvuuksista viitekehykseen nähden. Lisäksi ehdotuksessa pyritään puhtauteen ja käyttäjän kannalta mahdollisimman suureen läpinäkyvyyteen *ulkoistamalla* sumeuden käsittely varsinaisesta oliomallista. Tällä tarkoitetaan sitä, että oliomalli voidaan laatia perinteiseen tapaan terävänä luokkahierarkiana ja luokkien välisten suhteiden hierarkiana ilman, että ohjelmoijan tarvitsee huomioida sumean viitekehyksen olemassaoloa lainkaan. Sumeus voidaan “liimata” oliomallin päälle käyttäen Java-kielen tarjoamia mekanismeja, kuten metatietoa annotaatioiden muodossa.

Oliomallin puhtauteen ja toiminnallisuuden “päälle liimaamiseen” perustuvia muita ohjelmointiviikehyksiä ovat mm. pysyvyydenhallintaan tarkoitettu *Java Persistence API* (JPA)[10] sekä XML-muotoisen tiedon käsittelyyn tarkoitettu *Java Architecture for XML Binding* (JAXB) 2.0[11]. Ne kumpikin nojaavat vahvasti metatietoannotaatioiden käyttöön ja toiminnallisuuden ulkoistamiseen kohteena olevasta oliomallista.

On mielestäni suuri etu ohjelmoijan kannalta, jos oliomalli voidaan rakentaa perinteisellä tavalla ilman sumeita yliluokkia tms. Sen lisäksi, että ohjelmoijan ei tarvitse suunnitella uutta oliomallia ottaen huomioon periytyminen sumean viitekehyksen tarjoamista abstrakteista yliluokista, ei ohjelmoijan tarvitse myöskään suunnitella olemassaolevaa oliomallia uudestaan siinä tapauksessa, että siihen halutaan lisätä sumeita ominaisuuksia jälkikäteen.

[10] http://java.sun.com/javaee/technologies/persistence.jsp
[11] https://jaxb.dev.java.net/

Ehdotus sumeaksi olio-ohjelmointiviitekehykseksi jakautuu kahteen osaan: i) sumeat annotaatiot, joilla perinteinen terävä oliomalli “sumennetaan” läpinäkyvästi ja ii) sumeuden hallintakomponentit, jotka käsittelevät sumeiden annotaatioiden metatiedon ja tarjoavat käyttäjälle oliomallin sumeat palvelut.

6.1. Sumeat annotaatiot

Kuten edellä on mainittu, on metatiedon käyttöä sumeuden esittämisen apuna käyttänyt mm. Berzal *et al.* [2005b]. Berzal *et al.*:n ehdotuksessa annotaatioiden rooli on kuitenkin lähinnä avustava sumean luokkahierarkian ja sen juuriluokan `FuzzyObject` rinnalla (ks. 5.2). Myöhemmässä tutkimuksessaan Berzal *et al.* [2007b] lisäävät jonkin verran metatiedon käyttöä lisäämällä sumean samankaltaisuuden laskentaa varten kaksi uutta, laskentatapaa ohjaavaa annotaatiota, mutta sumeuden käyttö yleensä on edelleen sidottu `FuzzyObject`-luokan käyttöön.

Java-kielessä on mahdollista käyttää annotaatioita useassa paikassa: pakkauksen, tyypin (luokan, rajapinnan, luetellun tyypin (enumeration) tai annotoinnin), attribuutin, metodin, metodin parametrin, rakentimen tai paikallisen muuttujan yhteydessä. Annotointi voi olla parametroitu, kuten esimerkiksi `@FuzzyObject`-annotaatio, joka saa parametrikseen periytymistyypin::

```
/**
 * Erityinen huoneluokka, joka periytyy yleisestä huoneluokasta.
 * Periytyminen tapahtuu epätyypillisesti.
 *
 *
 * @author teemu
 *
 */
@FuzzyObject(inheritance=InheritanceType.ATYPICAL)
public class SpecialRoom extends Room {

    // ...
}
```

Annotointi voi olla myös parametroimatonta. Parametroimatonta annotointia kutsutaan merkintäannotaatioksi (marker annotation).

Seuraavissa kohdissa on kuvattu tähän ehdotukseen toistaiseksi ainoana kuuluva annotaatio `@FuzzyObject`.

6.1.1. @FuzzyObject

Samaan tapaan kuin Berzal *et al.*:n [2003] ehdotuksessa, jossa sumea malli rakennettiin `FuzzyObject`-luokan perustalle (ks. 5.2), on tässä ehdotuksessa suurin merkitys sumeuden esiintuomisessa oliomallissa `@FuzzyObject`-annotaatiolla. Annotaatiolla merkitään luokka sumeaksi viitekehyksen sisällä. Sumeuden hallintakomponentti käsittelee vain sellaisia luokkia, joilla on `@FuzzyObject`-annotaatio. Annotaation määritys on seuraava:

```
/**
 * Annotaatio @FuzzyObject, jolla merkitään luokka sumeaksi.
 *
 * @author teemu
 *
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FuzzyObject {
     /**
      *
      * @return periytymistyyppi, joka voi olla oletusarvoinen,
      * tyypillinen tai epätyypillinen.
      */
     InheritanceType inheritance() default InheritanceType.DEFAULT;
}
```

`@FuzzyObject` voi saada parametriksi periytymistyypin, joka tulee luetellusta tyypistä `InheritanceType`:

```
/**
 * Lueteltu tyyppi kaikista periytymistyypeistä, joita
 * tässä mallissa tuetaan.<br>
 * Tällä hetkellä mallissa on tuettu vain
 * Rossazza et al.:n [1997] ehdotuksen kolmea eri
 * tyyppistä periytymistä: tyypillistä, oletusarvoista ja
 * epätyypillistä.
 *
 * @author teemu
 *
 */
public enum InheritanceType {
     /**
      * Tyypillinen
      */
     TYPICAL,
     /**
      * Oletusarvoinen
      */
     DEFAULT,
     /**
      * Epätyypillinen
      */
     ATYPICAL;

}
```

Määritellään esimerkiksi luokka `Room` seuraavasti:

```
/**
 * @author teemu
 *
 */
@FuzzyObject
public class Room {

     // muu toteutus jätetty pois
}
```

Tämä esimerkki vastaa Berzal *et al.*:n [2003] `Room`-luokan esimerkkiä (ks. 5.2) sillä erotuksella, että omassa ehdotuksessani annotaatio `@FuzzyObject` korvaa Berzal *et al.*:n

ehdotuksessa olevan periytymisen `FuzzyObject`-luokasta. Näin oliomallin luokkahierarkiasta saadaan viitekehyksen luokista riippumaton.

6.2. Sumeuden hallintakomponentti

Edellä kuvattu metatietoannotaatio `@FuzzyObject` ei sellaisenaan muuta oliomallia sumeaksi, vaan oliomalli toimii edelleen terävänä terävässä kontekstissa. Jotta oliomalliin määritelty sumeus saataisiin esiin, täytyy luoda ympäristö, joka osaa tulkita ja käsitellä sumeita annotaatioita. Komponentin tarjoamat palvelut on hyvä määritellä erillisessa rajapinnassa, jonka varsinainen komponenttiluokka toteuttaa. Rajapinnan käyttö antaa mahdollisuuden piilottaa varsinainen toteutus rajapinnan käyttäjältä. Erilaisiin tilanteisiin ja laskentatapoihin voidaan laatia erilaisia toteutuksia ilman, että rajapintaa käyttävää osaa tarvitsee muuttaa. Ehdotus komponentin rajapinnaksi alkuvaiheessa on seuraava:

```
/**
 * Sumeuden hallintakomponentin rajapintaluokka.
 * Rajapinnassa on määritelty komponentin tarjoamat
 * sumeat palvelut.
 *
 * @author teemu
 * @param <E> sumean asteen (esim. yhtäsuuruusaste) tyyppi,
 * joka määritellään toteuttavassa luokassa.
 */
public interface FuzzinessManager<E extends Object> {

    /**
     * Vertaa kahta oliota toisiinsa ja palauttaa arvon,
     * joka on paremtrilla {@link E} määritettyä tyyppiä.
     *
     * @param o1
     * @param o2
     * @return
     */
    public E compare(Object o1, Object o2);

    /**
     * Laskee olion <code>o</code> ilmentymisasteen
     * luokassa <code>c</code>.
     *
     * @param o
     * @param c
     * @return
     */
    public E measureInstanceDegree(Object o, Class<?> c);
}
```

Rajapintaa voidaan laajentaa lisäämällä siihen palveluita mm. periytyvyysasteen laskemista varten.

7. Yhteenveto

Tässä työssä on käyty lyhyesti läpi olio-ohjelmoinnin ja sumeuden perusteita ja näiden pohjalta esitelty, mitä sumeat oliotietomallit ovat ja miten niitä on kirjallisuudessa käsitelty. Tämän jälkeen on kuvattu, kuinka sumeuden käyttöä olio-ohjelmoinnissa on tutkittu. Esiteltyihin aiempiin tutkimuksiin nojautuen on konstruoitu oma, hyvin suppea ehdotus sumeaksi ohjelmointiviitekehykseksi Java-kielellä.

Tutkimusaihe on erittäin mielenkiintoinen; sumeuden teoriat tarjoavat lukuisia erilaisia, toisiaan täydentäviäkin, tulkintoja ja toteutustapoja sumeuden käsittelyyn olio-ohjelmoinnissa. Tässä työssä ehdotetty viitekehys ei ole suinkaan kaikenkattava ja kauttaaltaan valmis konstruktio, vaan se vaatii runsaasti lisäkehitystä kypsyäkseen laajaan käyttöön soveltuvaksi sumean ohjelmoinnin työkaluksi. Järjestelmän tulisi sisältää mekanismit jäsenyysasteiden automaattiseen laskentaan. Jäsenyysfunktioiden ja samankaltaisuusrelaatioiden automaattisessa laskennassa voitaisiin hyödyntää esimerkiksi neuroverkkoja. Toteutuksessa voisi olla hyötyä *aspektiperustaisesta ohjelmoinnista* (aspect oriented programming), jossa erityinen aspektiluokan instanssi kytkeytyisi järjestelmään ja sieppaisi tietyt metodikutsut ja pystyisi muuttamaan niiden palauttamia arvoja sumeiksi.

Työn tekeminen on ollut hyvin antoisaa ja tarjonnut laajan näkökulman alan nykyiseen tutkimukseen. Tutkimuksissa ei ole noussut esiin yhtä ainoaa oikeaa tapaa käyttää sumeutta oliperustaisessa mallinnuksessa ja ohjelmoinnissa. Erilaisia teorioita ja laskentatapoja on useita ja niistä voidaan valita kulloiseenkin tilanteeseen parhaiten sopiva. Luotavan ohjelmointiviitekehyksen tulisi ottaa tämä huomioon, eikä tarjota tukea vain yhdelle tavalle. Kohdassa 6 konstruoitu yksinkertainen ehdotus ottaa huomioon laskentatapojen moninaisuuden siten, että sumeus ja sumeuden laskenta ovat hyvin löyhästi sidottu varsinaiseen oliotietomalliin rajapintojen ja metadatan avulla.

Viiteluettelo

[Aksoy *et al*., 1996] Demet Aksoy, Adnan Yazici and Roy George, Extending the similarity-based fuzzy object-oriented data model. In: *Proceedings of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU)*, Granada, Spain, July 1996, 1177-1182.

[Berzal *et al*., 2003] Fernando Berzal, Nicolás Marín, Olga Pons and María-Amparo Vila, Using classical object-oriented features to build a foodbs. In: J. Lee (ed.), *Software Engineering With Computational Intelligence*. Springer Verlag, 2003.

[Berzal *et al*., 2005a] Fernando Berzal, Nicolás Marín, Olga Pons and María-Amparo Vila, Development of Applications with Fuzzy Objects in Modern Programming Platforms. *International Journal of Intelligent Systems* **20** (2005), 1117-1136.

[Berzal *et al*., 2005b] Fernando Berzal, Juan Carlos Cubero, Nicolás Marín and Olga Pons, Fuzzy Object-Oriented Modeling with Metadata Attributes in C#. *Advances in Soft Computing* **2** (2005), 253-262.

[Berzal *et al*., 2007a] Fernando Berzal, Nicolás Marín, Olga Pons and María-Amparo Vila, Managing Fuzziness on Conventional Object-Oriented Platforms. *International Journal of Intelligent Systems* **22** (2007), 781-803.

[Berzal *et al*., 2007b] Fernando Berzal, Juan Carlos Cubero, Nicolás Marín, María-Amparo Vila, Janusz Kacprzyk and Slawomir Zadrozny, A General Framework for Computing with Words in Object-Oriented Programming. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **15** (2007), 111-131.

[Bordogna *et al*., 1994] Gloria Bordogna, Dario Lucarella and Gabriella Pasi, A fuzzy object-oriented data model. In: *Proc. of the IEEE 3rd Internation Conference on Fuzzy Systems* (1994), 313-318.

[Bordogna *et al*., 1997] Gloria Bordogna, Dario Lucarella and Gabriella Pasi, Extending a graph-based data model to manage fuzzy and uncertain information. In: Rita De Caluwe (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Consepts and Models*. *Advances in Fuzzy Systems* **13** (1997), World Scientific, 97-122.

[Bordogna *et al*., 1999] Gloria Bordogna, Dario Lucarella and Gabriella Pasi, A fuzzy object-oriented data model managing vague and uncertain information. *International Journal of Intelligent Systems* **14**,7 (1999), 623-651.

[Bordogna and Pasi, 2001] Gloria Bordogna and Gabriella Pasi, Graph-based interaction in a fuzzy object oriented database. *International Journal of Intelligent Systems* **16**,7 (2001), 821-841.

[Buisson *et al*., 1987] Jean-Christophe Buisson, Henri Farrehy and Henri Prade, Dealing with imprecision and uncertainty in expert system DIABETO-III. In: *Innovation and Technology in Biology and Medicine* **8**,2 (1987).

[Cross *et al*., 1997] Valerie Cross, Rita de Caluwe and Nancy Van Gyseghem, A perspective from the Fuzzy Object Data Management Group (FODMG). In: *Proceedings of the 6th International Conference on Fuzzy Systems* **2** (1997), 721-728.

[Cross, 2002] Valerie Cross, Fuzzy inheritance in fuzzy object models. In: *Proceedings of the 2002 IEEE International Conference on Fuzzy Systems* **2** (2002), 843-848.

[de Tré and de Caluwe, 2004] Guy de Tré and Rita de Caluwe, A Constraint Based Fuzzy Object Oriented Database Model. In: Zongmin Ma (ed.), *Advances In Fuzzy Object-oriented Databases: Modeling And Applications*, Idea Group Publishing, 2005, 1-45.

[Dubois *et al*., 1991] Didier Dubois, Henri Prade and Jean-Paul Rossazza, Vagueness, typicality and uncertainty in class hierarchies. *International Journal of Intelligent Systems* **6** (1991), 167-183.

[Dubois and Prade, 2001] Didier Dubois and Henri Prade, Possibility theory, probability theory and multiple-valued logics: a clarification. In: *Annals of Mathematics and Artificial Intelligence* **32** (2001), 35-66.

[Duch *et al*., 1999] Wlodzislaw Duch, Rafal Adamczak and Krzysztof Grabczewski, Neural optimization of linguistic variables and membership functions. In: *Proceedings of the ICONIP '99, 6th International Conference on Neural Information Processing* (1999), 616-621.

[Fullér, 1999] Robert Fullér, On fuzzy reasoning schemes. In: C. Carlsson (ed.), *The State of the Art of Information Systems in 2007*, TUCS General Publications, No. 16, Turku Centre for Computer Science, Åbo, 1999, 85-112.

[George *et al*., 1993] Roy George, Fred E. Petry and Bill P. Buckles, Modeling class hierarchies in the fuzzy object oriented data model. *Fuzzy Sets and Systems* **60**, 3 (1993), 259-272.

[George *et al*., 1996] Roy George, Radhakrishnan Srikanth, Fred E. Petry and Bill P. Buckles, Uncertainty management issues in the object-oriented data model. *IEEE Transactions on Fuzzy Systems* **4**, 2 (May 1996), 179-192.

[George *et al*., 1997] Roy George, Adnan Yazici, Fred. E. Petry and Bill P. Buckles, Modeling impreciseness and uncertainty in the object-oriented data model – a similarity-based approach. In: Rita De Caluwe (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Consepts and Models*. *Advances in Fuzzy Systems* **13** (1997), World Scientific, 63-95.

[Inoue *et al*., 1991] Y. Inoue, S. Yamamoto and S. Yasunobu, Fuzzy set object: fuzzy set as first-class object. In: *Proceedings of the International Fuzzy Systems Association Conference, IFSA '91*, Brussels, July 1991.

[Kankainen, 2000] Mikko Kankainen, Sumeat relaatiot. Teoksessa: Marko Junkkari (toim.), *Sumean teoriaa ja sovelluksia*, Tampereen yliopisto, tietojenkäsittelytieteiden laitos, julkaisusarja **B-2000-1**, toukokuu 2000.

[Karvonen, 2000] Suvi Karvonen, Sumeaa logiikkaa. Teoksessa: Marko Junkkari (toim.), *Sumean teoriaa ja sovelluksia*, Tampereen yliopisto, tietojenkäsittelytieteiden laitos, julkaisusarja **B-2000-1**, toukokuu 2000.

[Klir and Yuan, 1995] George Klir and Bo Yuan, *Fuzzy Sets and Fuzzy Logic: Theory and Applications*, Prentice Hall, 1995.

[Koskimies, 1993] Kai Koskimies, *Olioperustainen ohjelmistokehitys, luvut 1-2*, Tampereen yliopisto, tietojenkäsittelyopin laitos, julkaisusarja **C-1993-3**, elokuu 1993.

[Kosko, 1996] Bart Kosko, *Sumea logiikka*. Gummerus Kirjapaino Oy, Jyväskylä, 1996.

[Lee *et al*., 2001] Jonathan Lee, Jong-Yih Kuo and Nien-Lin Xue, A note on current approaches to extending fuzzy logic to object-oriented modeling. *International Journal of Intelligent Systems* **16**, 7 (2001), 807-820.

[López-Ortega, 2006] Omar López-Ortega, Java Fuzzy Kit (JFK): A shell to build fuzzy inference systems according to the generalized principle of extension. *Expert Systems with Applications* **34** (2008), 796-804.

[Ma *et al*., 2004] Zongmin Ma, Wenjun Zhang and Weiyin Ma, Extending object-oriented databases for fuzzy information modeling. *Information Systems* **29** (2004), 421-435.

[Mattila, 1997] Jorma Mattila, *Sumean logiikan oppikirja: johdatusta sumeaan matematiikkaan*, Art House, 1997.

[Marín *et al*., 2000] Nicolás Marín, María-Amparo Vila and Olga Pons, Fuzzy types: A new concept of type for managing vague structures. *International Journal of Intelligent Systems* **15**, 11 (2000), 1061-1085.

[Marín *et al*., 2001] Nicolás Marín, Olga Pons and María-Amparo Vila, A strategy for adding fuzzy types to an object-oriented database system. *International Journal of Intelligent Systems* **16**, 7 (2001), 863-880.

[Niskanen, 2003] Vesa A. Niskanen, *Sumea logiikka*. WSOY, Vantaa, 2003.

[Pereira, 2006] Wilmer Pereira, Proposal of fuzzy object oriented model in extended Java. In: *Professional Practice in Artificial Intelligence, IFIP International Federation for Information Processing* **218** (2006), Springer, 191-200.

[Rossazza, 1990] Jean-Paul Rossazza, *Utilisation de hiérarchies de classes floues pour la représentation de connaissances imprécises et sujettes à exceptions : le système SORCIER*. Thèse de l'université Paul Sabatier de Toulouse soutenue le 15 mai 1990.

[Rossazza *et al*., 1997] Jean-Paul Rossazza, Didier Dubois and Henri Prade, A Hierarchical Model of Fuzzy Classes. In: Rita De Caluwe (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Consepts and Models*. *Advances in Fuzzy Systems* **13** (1997), World Scientific, 21-61.

[Rumbaugh *et al*., 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, *Object-oriented Modeling and Design*. Prentice-Hall, 1991.

[Siermala, 2000] Markku Siermala, Sumea aritmetiikka. Teoksessa: Marko Junkkari (toim.), *Sumean teoriaa ja sovelluksia*, Tampereen yliopisto, tietojenkäsittelytieteiden laitos, julkaisusarja **B-2000-1**, toukokuu 2000.

[Tanaka *et al*., 1991] Katsumi Tanaka, Susumu Kobayashi and Tomomi Sakanoue, Uncertainty Management in Object-Oriented Database Systems. In: *Proceedings*

of the International Conference on Database and Expert Systems Applications (DEXA91), Springer-Verlag, 1991, 251-256.

[Van Gyseghem *et al*., 1993] Nancy Van Gyseghem, Rita de Caluwe and R. Vandenberghe, UFO: Uncertainty and fuzziness in an object-oriented data model. In: *Proceedings of FUZZ-IEEE '93*, San Francisco, USA, March-April 1993, 773-778.

[Van Gyseghem and de Caluwe, 1994] Nancy Van Gyseghem and Rita de Caluwe, Fuzzy object-oriented databases: Some behavioural issues. In: *Proceedings of the Second European Congress on Intelligent Techniques and Soft Computing*, Aachen, Germany, September 20-23, 1994, 361-365.

[Van Gyseghem and de Caluwe, 1995] Nancy Van Gyseghem and Rita De Caluwe, Fuzzy behaviour and relationships in a fuzzy OODB-model. In: *Proceedings of the Tenth Annual ACM Symposium on Applied Computing*, Nashville, TN, USA, February 26-28, 1995, 503-507.

[Van Gyseghem and de Caluwe, 1996] Nancy Van Gyseghem and Rita de Caluwe, Overview of the UFO database model. In: *Proceedings of the Fourth European Congress on Intelligent Techniques and Soft Computing*, Aachen, Germany, September 2-5, 1996.

[Van Gyseghem and de Caluwe, 1997] Nancy Van Gyseghem and Rita de Caluwe, The UFO database model: dealing with imperfect information. In: Rita De Caluwe (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Consepts and Models*. *Advances in Fuzzy Systems* **13** (1997), World Scientific, 123-185.

[Van Gyseghem and de Caluwe, 1998] Nancy Van Gyseghem and Rita de Caluwe, Imprecision and uncertainty in the UFO database model. *Journal of the American Society for Information Science* **49**, 3 (March 1998), 236-252.

[Wong and Chun, 1999] Gary Yat Chung Wong and Hon Wai Chun, Modelling Fuzzy Sets Using Object-Oriented Techniques. In: *Multiple Approaches to Intelligent Systems, Lecture Notes in Computer Science* **1611/2004** (1999), Springer, 23-32.

[Yazici *et al*., 1998] Adnan Yazici, Roy George and Demet Aksoy, Design and implementation issues in the fuzzy object-oriented data model. *Journal of Information Sciences* **108** (1998), 241-260.

[Zadeh, 1965] Lofti A. Zadeh, Fuzzy Sets. *Information and Control* **8** (1965), 338-353.

[Zadeh, 1972] Lofti A. Zadeh, Fuzzy-set-theoretic interpretation of linguistic hedges. *Journal of Cybernetics **2*** (1972) 4-34.

[Zadeh, 1975] Lofti A. Zadeh, The concept of linguistic variable and its application to approximate reasoning. *Information Sciences* I: **8** (1975) 199-249; II: **8** (1975) 310-357; III: **9** (1975) 43-80.

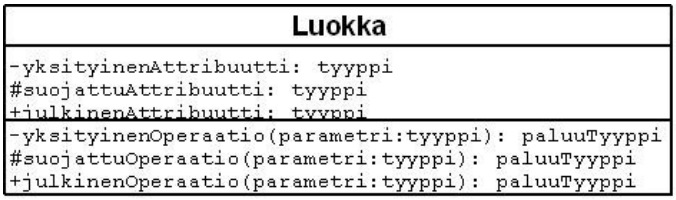
[Zadeh, 1978] Lofti A. Zadeh, Fuzzy sets as the basis for a theory of possibility. *Fuzzy Sets and Systems* **1** (1978), 3-28. (Reprinted in Fuzzy Sets Syst, 100 (Supplement) (1999), 9-34.

[Zicari, 1990] Robert Zicari, Incomplete Information in Object-Oriented Databases. *ACM SIGMOD Record* **19,** 3 (1990) 5-16.

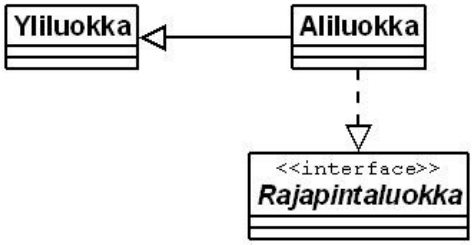
Liite 1

UML-notaatio

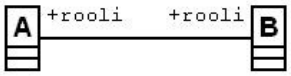
Alla on yksinkertainen UML-luokkakaavioiden notaatioselitys siltä osin, kuin sitä on tässä työssä käytetty. Katso tarkemmin UML-spesifikaatiosta[12].



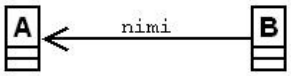
Luokka attribuutteineen ja operaatioineen



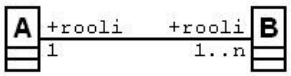
Periytyminen ja rajapinnan toteuttaminen



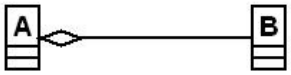
Assosiaatio, kahden luokan välinen yhteys



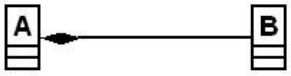
Suunnattu yksisuuntainen assosiaatio luokasta B luokkaan A



Assosiaatio jossa osallistujina yksi A ja 1-n kpl:a B



Kooste (aggregaatio), B on osa A:a



Aito kooste (kompositio)

[12] http://www.omg.org/spec/UML/Current/