Olioiden pysyvyyteen ja käyttäytymiseen liittyviä suunnittelumalleja uudelleenkäytettävyyden näkökulmasta

Timo Väänänen

13.6.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu-tutkielma

Tiivistelmä

Ohjelmistosuunnitteluprosessien kehittämiseen kiinnitetään nykyisin entistä enemmän huomiota. Oliosuuntautuneen suunnittelun nähdään olevan yhä enenevissä määrin iteratiivinen prosessi, jossa kaikki vaiheet toteutetaan vähitellen pienissä osissa. Esimerkiksi analyysi- tai suunnitteluvaiheesta voidaan palata taaksepäin vaatimusmäärittelyyn tarkentamaan jotakin vaatimusta. Oleellista on se, ettei vaiheen tarvitse olla valmis ennen kuin voidaan edetä seuraavaan vaiheeseen. Tämä näkökanta eroaa 1970-luvulta adoptoidusta mallista, jossa ohjelmistosuunnittelun nähtiin etenevän suoraviivaisesti vaiheesta toiseen ilman mahdollisuutta vaiheiden väliseen ja aikaiseen iteraatioon. Tämä vesiputousmalliksi nimetty ohjelmiston elinkaarimalli on joutunut kovan kritiikin kohteeksi, koska se ei edistä vaiheiden välistä iteraatiota. Ohjelmistosuunnitteluprosessien kehittymisen myötä myös suunnitteluratkaisujen dokumentointiin on kiinnitetty huomiota. Suunnittelumallit ovat abstrakteja kuvauksia tällaisista ratkaisuista ja ne tarjoavat ohjelmistonsuunnittelijalle hyvän toteutuskieliriippumattoman apuvälineen suunnittelun aikana eteen tulevien abstraktioiden hahmottamiseen.

Tässä tutkielmassa selvitetään lyhyesti kuinka ohjelmistosuunnittelun ala on muuttunut 1950-luvulta 2000-luvulle tultaessa siirryttäessä rakenteisesta analyysista ja suunnittelusta olioparadigmaan. Tarkemmin tutustutaan suunnittelumallien tarjoamiin abstraktioihin ja niiden käyttöön olioiden käyttäytymiseen ja tiedon pysyvyyteen liittyvien suunnitteluongelmien yhteydessä kapselointia, periytymistä ja koostamista hyödyntämällä uudelleenkäytettävyyden lisäämiseksi.

ACM-luokat (ACM Computing Classification System, 1998 version) D.3.2, D.2.2

Avainsanat: suunnittelumalli, rakenteinen analyysi ja suunnittelu, olioparadigma, kapselointi, periytyminen, koostaminen

Sisällysluettelo

1. Johdanto 1

2. Suunnittelumallit 3

  2.1 Ohjelmistosuunnittelun kehittyminen 3

  2.2 Suunnittelumallien luokittelu 14

  2.3 Suunnittelumallit ohjelmistojen uudelleenkäytössä 19

3. Sisällönhallintasovellus 29

  3.1 Sovelluksen yleiskuvaus 29

  3.2 Domain Object Factory 32

  3.3 Data Access Object 37

  3.4 Chain of Responsibility 40

4. Yhteenveto 45

Viitteet 46

Liite 1. Roycen ohjelmistojen elinkaarimalli 49

Liite 2. Esimerkki olioiden koostamisesta 50

Liite 3. Product.php 51

Liite 4. DBDao.php 54

Liite 5. DataAccessObject-rajapinta 58

1. Johdanto

Ohjelmistonsuunnittelun sisältämiä prosesseja ollaan pyritty kehittämään 1970-luvulta lähtien. W.W Roycen 1970-luvulla esittämä elinkaarimalli tulkittiin tahattomasti tai tahallisesti vesiputousmalliksi, jossa ohjelmiston suunnittelu ja toteuttaminen koostui selkeistä vaiheista vaatimusmäärittelystä toteutukseen ja ylläpitoon. Mallissa oletettiin, että esimerkiksi ohjelmiston vaatimat määritykset saadaan kerättyä asiakkaalta vaivattomasti eikä niihin tarvinnut palata enää elinkaaren myöhemmissä vaiheissa. Frederick Brooks osoitti mallin toimimattomuuden vuonna 1986 julkaisemassaan esseessä: *No Silver Bullet: Essence and Accidents of Software Engineering*. Brooks kuvasi esseessä ohjelmistosuunnittelijan suurimpana haasteena olevan vaatimusten keräämisen asiakkaalta (Brooks, 1986). Tämän vaiheen tulisi sisältää Brooksin mukaan asiakkaan ja ohjelmistosuunnittelijan välisen läheisen yhteistyön, jonka aikana vaatimuksia tarkennetaan ja niiden pohjalta tehdään alustavia prototyyppejä. Brooks pitää melkein mahdottomana ajatusta, että asiakas pystyisi täydellisesti määrittelemään halutut vaatimukset ilman konkreettista kuvaa järjestelmän toiminnasta.

1980-luvulla syntyi uusi ohjelmistometodologia, joka käytti hyväkseen 1970-luvulla syntyneitä konsepteja tiedon käsittelemisestä. Oliosuuntautuneen paradigman odotettiin tuovan ratkaisun ohjelmistotuotantoalaa vaivanneeseen tuottavuuskriisiin. Rakenteellisessa analyysissa ja suunnittelussa käytetyt ideat ohjelmien vaiheittaisesta jakamisesta pienemmiksi ja hallittavimmiksi kokonaisuuksiksi sekä tiedon piilottamisesta siirrettiin funktioista ja proseduureista olioiden vastuulle. Rakenteellisessa paradigmassa sovelluksen tarvitsema data ja sen käsittelyyn tarvittavat operaatiot olivat kaksi eri käsitettä. Dataa käsiteltiin syötteinä ja sen perusteella saatiin tulosteita. Oliosuuntautuneessa paradigmassa data ja sen käsittelyyn tarvittavat operaatiot siirrettiin olioiden vastuulle. Näin saatiin luotua ja määriteltyä ominaisuuksia ja käyttäytymistä sisältäviä kokonaisuuksia, joita pystyttiin mallintamaan ympäröivästä reaalimaailmasta. Ohjelmistojen uudelleenkäytön näkökulmasta oliot toivat mullistavan uutuuden. Huomattiin, että olioiden keskinäisiä suhteita voitiin mallintaa reaalimaailman tavoin hierarkkisilla periytymissuhteilla, joissa yliluokan sisältämä toiminnallisuus voitiin uudelleen käyttää aliluokassa. Oliosuunnittelun näkökulmasta suunnittelumallit tarjoavat uudelleenkäytettävän abstraktion suunnitteluongelmien ratkaisemiseksi. Suunnittelumalleja on käytetty hyväksi ohjelmistonsuunnittelussa aina. Tästä tosiasiasta huolimatta, vasta kehittyneet mallinnuskielet kuten *UML* (Unified Modelling Language) (OMG, 2008) tarjosivat mahdollisuuden olioiden välisten suhteiden kuvaamisen

graafisesti luokka- ja oliosuhdekaavioilla, jolloin dokumentoinnista tuli helpompaa. Suunnittelumalleilla pystyttiin myös helposti dokumentoimaan toteutettavan järjestelmän sisältämät yksityiskohdat (Fraser, 1995). Suunnittelumalleja voidaan käyttää esimerkiksi kuvaamaan olioiden käyttäytymistä tai niillä voidaan kuvata olioiden luontiin liittyviä suunnitteluongelmia. Yksi esimerkki olioiden käyttäytymiseen liittyvistä suunnittelumalleista on Chain of Responsibility -malli, jossa oliolle lähetetyn pyynnön käsittelyyn voi osallistua useita olioita ohjelman suorituksen aikana. Pyyntö välitetään oliolta toiselle, kunnes joku olioista suostuu käsittelemään pyynnön (Gamma et al., 2000). Myös tiedon pysyvyyteen liittyvät suunnittelumallit ovat aiheuttaneet kasvavaa kiinnostusta, sillä kaikissa tosielämän sovelluksissa käsitellään jossakin muodossa tiedon tallentamista tietokantaan ja vastaavasti tiedon hakemista tietokannasta.

Tämän opinnäytetyön sisältö on jaettu seuraavasti. Luvussa 2 pyritään luomaan yleiskatsaus suunnittelumalleihin. Luvussa 3 kuvataan lyhyesti syksyllä 2007 Ina Finland Oy:lle toteutettu sisällönhallintasovellus, jossa myöhemmissä luvuissa kuvattuja suunnittelumalleja käytettiin ratkaisemaan suunnittelun aikana eteen tulleita ongelmia. Yhteenveto ja johtopäätökset tehdään lopuksi luvussa 4.

2. Suunnittelumallit

Kohdassa 2.1 tutustutaan ohjelmistosuunnitteluprosesseihin historiallisesta näkökulmasta, painotuksen ollessa vahvasti rakenteellisessa analyysissä ja suunnittelussa sekä oliosuunnittelussa. Kohdassa 2.2 pyritään selvittämään tarkemmin mitä suunnittelumallit ovat ja miksi niitä kannattaisi käyttää. Tämän lisäksi tutustutaan Erich Gamman luomaan suunnittelumallien luokitteluun, jossa suunnittelumallit on jaettu erillisiin luokkiin sen mukaan mikä tarkoitus ja kohde suunnittelumallilla on (Gamma et al., 2000). Kohdassa 2.3 tarkennetaan analyysiä ja tarkastellaan suunnittelumalleja ennen kaikkea ohjelmistojen uudelleenkäytön näkökulmasta. Kohdassa 2.3 tutustutaan sekä organisaatiotasoiseen uudelleenkäyttöön sekä siihen, kuinka yksittäinen ohjelmistosuunnittelija voi suunnitella ja toteuttaa uudelleenkäytettäviä komponentteja suunnittelumallien ja oliomekanismien avulla.

2.1 Ohjelmistosuunnittelun kehittyminen

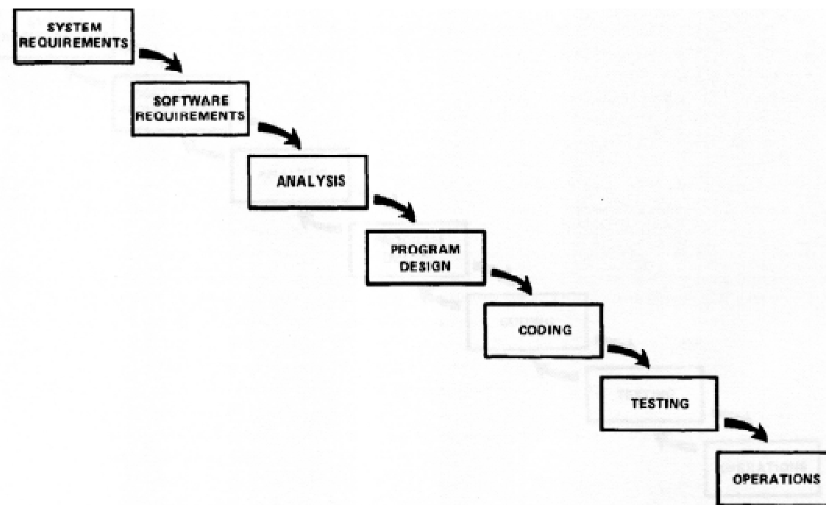
Tarkasteltaessa ohjelmistosuunnittelua historialliselta näkökannalta, voidaan nostaa esiin kaksi keskeistä yhä edelleen 2000-luvulla vaikuttavaa paradigmaa. 1950-luvulla ohjelmistosuunnitteluun tai sen sisältämiin prosesseihin ei kiinnitetty juuri lainkaan huomiota, vaan ohjelmistosuunnittelun tarkoituksena nähtiin olevan ainoastaan kalliiden tietokonelaitteistojen toimintakyvyn turvaaminen (Boehm, 2006). Boehm esittääkin 1950-luvulta adaptoidun näkökannan, jossa ohjelmistosuunnittelun rooli voitiin nähdä selkeästi:

Software engineering = Hardware engineering

Sotavarustelun kiihtyminen sekä sen myötä puolustusmenojen kasvava osuus kansantalouksien budjeteista oli saanut aikaan sotateollisuutta, jossa matemaatikot ja tietokonelaiteinsinöörit rakensivat mm. monimutkaisia tutka- ja ohjusjärjestelmiä. Järjestelmät määriteltiin, suunniteltiin sekä toteutettiin prosessien mukaan, jotka muistuttivat kaukaisesti myöhemmin ohjelmistosuunnittelussa laajemmin käyttöönotettua vesiputousmallia. 1960-luvulla havahduttiin ohjelmistosuunnitteluprosessien alalle tuomiin etuihin: huomattiin, että ohjelmistoja oli paljon helpompi muokata kuin laitteistoja. Sen sijaan, että jokaisen yksittäisen tietokoneen kokoonpanoa muutettaisiin joka kerta vastaamaan tehtyä muutosta, helpompaa oli monistaa muutoksessa tarvittava ohjelmakoodi. Huomattiin myös että toisin kuin tietokonelaitteistot, ohjelmistot eivät kuluneet loppuun. 1970-luvulla huomio ohjelmistosuunnittelun prosesseihin sai aikaan panostusta ohjelmiston elinkaaren

vaiheisiin. Suunnittelu nähtiin selkeänä omana elinkaaren osanaan, jota edelsi vaatimusmäärittely ja joka toimi syötteenä ohjelmointivaiheelle. W.W.Royce esitti vuonna 1970 liitteessä 1 kuvatun alkuperäisen mallinsa iteratiivisesta ohjelmiston elinkaaresta, jossa vaiheet seuraavat toisiaan peräkkäisessä järjestyksessä (sequential order), mutta jossa vaiheista voidaan palata taaksepäin saamalla näin aikaan iteratiivinen ohjelmiston elinkaari, joka kannustaa pieniin muutoksiin ja virheiden nopeaan löytämiseen.

Valitettavasti Roycen alkuperäinen idea tulkittiin kuitenkin elinkaarimalliksi, jossa vaiheet seuraavat toisiaan järjestyksessä eikä vaiheista ole mahdollisuutta palata taaksepäin edelliseen vaiheeseen (Boehm, 2006). Kuvassa 1 on esitettynä Roycen alkuperäisestä elinkaarimallista muunneltu versio, joka on ollut kovan kritiikin alaisena julkistamisestaan lähtien.



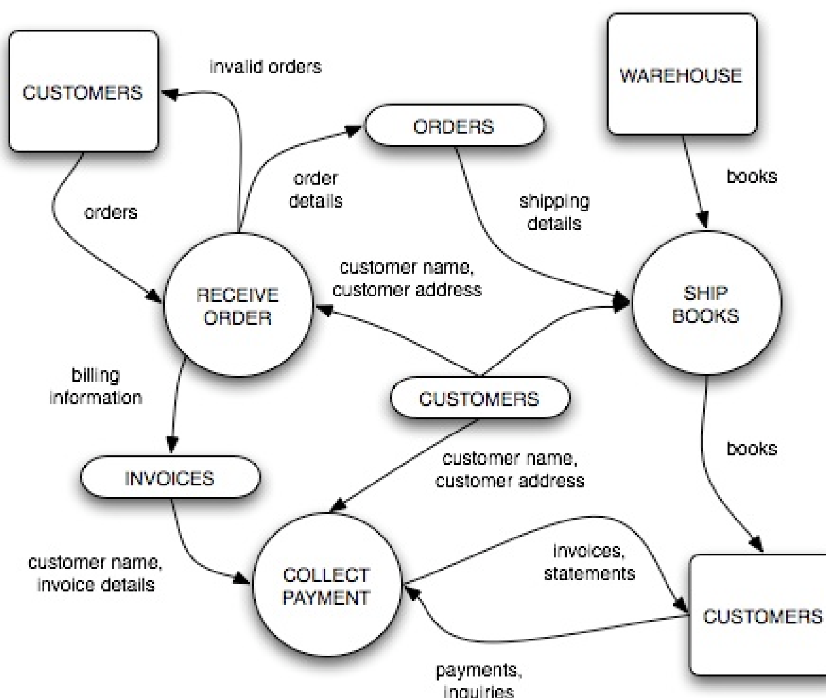
Kuva 1. Vesiputousmalli (Fincher M, 2006).

Kritiikin on nähty kohdistuneen vesiputousmallin joustamattomuuteen. Davis mm. kritisoi vesiputousmallia siitä, ettei sen käyttö anna suurten ohjelmistoprojektien johtajille välttämättä oikeaa kuvaa siitä missä ohjelmiston kehitysvaiheessa projektiorganisaatio kulloinkin on (Davis, 1994). Korson puolestaan kritisoi vesiputousmallia siitä, että se ei tue elinkaarivaiheiden yhdistämistä ja että siinä ei painoteta tarpeeksi ohjelmistojen uudelleenkäyttöä (Korson, 1990).

Krish Pillai mainitsee vesiputousmallista puuttuvan mahdollisuuden *protoiluun* (prototyping), jossa ohjelmisto kehitetään pienissä iteratiivisissa vaiheissa jolloin jokaisessa vaiheessa syntyy jokin konkreettinen tulos (Pillai, 1996).

Ohjelmiston elinkaarimallien lisäksi 1970-luvulla panostettiin erilaisiin ohjelmistomenetelmiin. *Menetelmillä* (methodology) tarkoitetaan ohjelmistojen yhteydessä tapaa, jolla tietty ongelma voidaan hahmottaa sekä myöhemmin suunnitella ongelman ratkaiseminen ohjelmistotuotannon keinoilla. *Rakenteellinen suunnittelu ja analyysi* (Structured Analysis and Design) keskittyy elinkaaren analyysi- ja suunnitteluvaiheisiin. Sille on ominaista käsillä olevan ongelman pilkkominen pienimmiksi ja samalla hallittavimmiksi osakokonaisuuksiksi. Nämä osakokonaisuudet jaetaan funktioihin/proseduureihin, joiden välillä tarvittava tieto kulkee.

Rakenteellisen analyysin tarkoituksena on selventää mitä ollaan tekemässä. Vaiheen lopputuloksena saadaan *määritystietoja* (data dictionaries), tarkentavia tekstimuotoisia kuvauksia sekä graafisia *tietovuokaavioita* (data flow diagram), jotka sisältävät kuvauksia järjestelmän vaatimasta datasta ja siitä kuinka data kulkee järjestelmän sisältämien osien välillä. Kuvassa 2 on esitettynä yksi esimerkki tällaisesta tietovuokaaviosta.



Kuva 2. Tietovuokaavio (Yourdon, 2008).

Kuvassa 2 esitetystä tietovuokaaviosta on erotettavissa neljä keskeistä osaa, joista toteutettava järjestelmä koostuu. Seuraavassa jokainen näistä osista käydään lyhyesti läpi ja pyritään samalla tarkentamaan niiden merkitystä rakenteellisen analyysin näkökulmasta.

Kuvassa 2 ympyrällä kuvatut prosessit/funktiot saavat *syötteenä* (input) dataa. Prosessit käsittelevät syötteet itsenäisesti ja saavat aikaan *tulosteita* (output), jotka palautetaan järjestelmän muiden osien käyttöön. Prosessin nimen tulisi kuvata mahdollisimman selkeästi

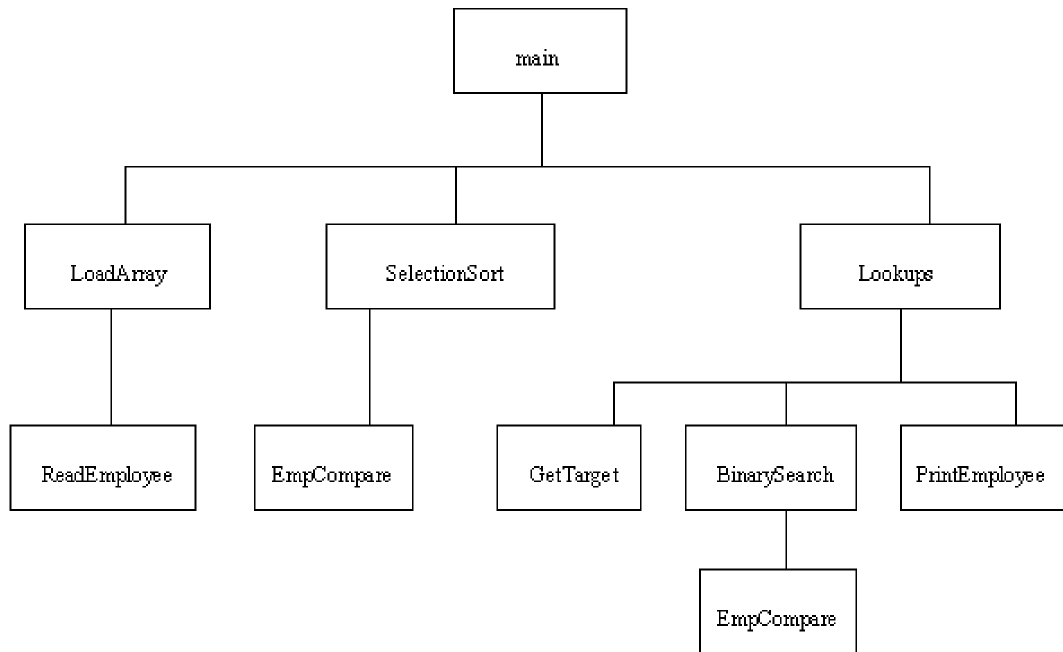
prosessin tarkoitus ts. mitä prosessi tekee. Kuvassa 2 esitetyt nuolet kuvaavat järjestelmän sisältämän datan virtausta prosesseihin tai prosesseista pois. Yleensä nuolen yhteyteen liitetään vielä tekstimuotoinen selvennys datan virtauksen tarkoituksesta. Huomioitavaa on myös se, että datan virtaaminen voi olla myös kaksisuuntaista.

Kuvassa 2 esitetty pyöristetty suorakaide kuvaa puolestaan tietovarastoja, joihin järjestelmän sisältämää dataa tallennetaan. Esimerkkejä tällaisista tietovarastoista voivat olla mm. tietokannat ja tekstitiedostot.

Järjestelmään liitetyt ulkoiset entiteetit (external entities) voidaan kuvata puolestaan pyöristetyillä neliöillä. Ulkoisten entiteettien voidaan ymmärtää olevan järjestelmään liittyviä ihmisiä, organisaatioita tai muita järjestelmiä. On kuitenkin huomioitava, että nämä entiteetit ovat mallinnettavan järjestelmän ulkopuolisia kohteita.

Entiteettien ja prosessien välisillä datavirtauksilla pyritään kuvaamaan ainoastaan sisäisen järjestelmän ja ulkopuolisen järjestelmän välisiä rajapintoja. Tämän lisäksi tietovuokaavion ei ole tarkoitus kuvata entiteettien välisiä suhteita, koska nämä mahdolliset suhteet eivät vaikuta itse toteutettavaan järjestelmään. Rakenteellisen analyysivaiheen tärkeimpänä osana voidaan pitää järjestelmän sisältämien osien jakamista pienempiin, itsenäisiin kokonaisuuksiin, joita kutsutaan *moduuleiksi* (modules). Nämä moduulit ovat ikään kuin suurempia järjestelmän sisältämiä osakokonaisuuksia.

Rakenteellinen suunnittelu seuraa analyysivaihetta ja sen aikana pyritään ottamaan kantaa siihen kuinka analyysivaiheessa rakennetut moduulijaot voidaan jakaa vielä pienempiin osiin, joita kutsutaan funktioiksi/proseduureiksi. *Rakennekaavio* (structure chart) tarjoaa keinon järjestelmän sisältämien funktioiden hierarkian graafiseen kuvaamiseen. Kuvassa 3 on esitetty esimerkki tällaisesta rakennekaaviosta. Rakenteellisen suunnittelun aikana järjestelmän sisältämien osien jakaminen pienempiin, helpommin hallittaviin osiin tapahtuu noudattamalla osittavaa tapahtuvaa suunnittelua (top-down design).



Kuva 3. Rakennekaavio (Carlson, 2006).

Kuvassa 4 on esitetty esimerkki yksinkertaisesta funktiosta, jonka *kutsumallista* (signature) voidaan nähdä funktioihin ja samalla rakenteelliseen analyysiin ja suunnitteluun liittyvä perusajatus pyrkimyksestä *tiedon piilottamiseen* (information hiding). Parnasin, vuonna 1972 esittämällä mallilla pyritään siihen, että yksittäisen järjestelmän osakokonaisuuden, moduulin sisäinen rakenne ei näy moduulin ulkopuolelle (Sullivan et al., 2001). Kuvassa 4 esitetty funktio saa syötteenä osoittimen merkkijonoon. Funktiossa lasketaan kuinka monta kirjainta merkkijonossa esiintyy. Funktio palauttaa kutsuvalle ohjelmanosalle kirjaimien lukumäärän.

```
char strlength (char *str)
{
    int length = 0;
    for(; *str != ’\0’; ++str)
        ++length;
    return length;
}
```

Kuva 4. C-kielinen funktio.

Oliosuuntautuneessa paradigmassa tiedon piilottamista tuetaan toteutuksen kannalta luokan tietojäseniin liitettyjen näkyvyysmääreiden kautta (Learning&Development, 2008). Esimerkiksi php5-skriptikieli tukee luokan tietojäsenien määrittelemistä *yksityisiksi* (private), *suojatuiksi* (protected) tai *julkisiksi* (public). Kuvassa 5 esitetään luokka, jonka sisältämä tieto on suunniteltu yksityiseksi. Luokkaan määritellään tarvittaessa erityiset saanti ja asetus-metodit, joiden kautta olion tilaan on mahdollista vaikuttaa.

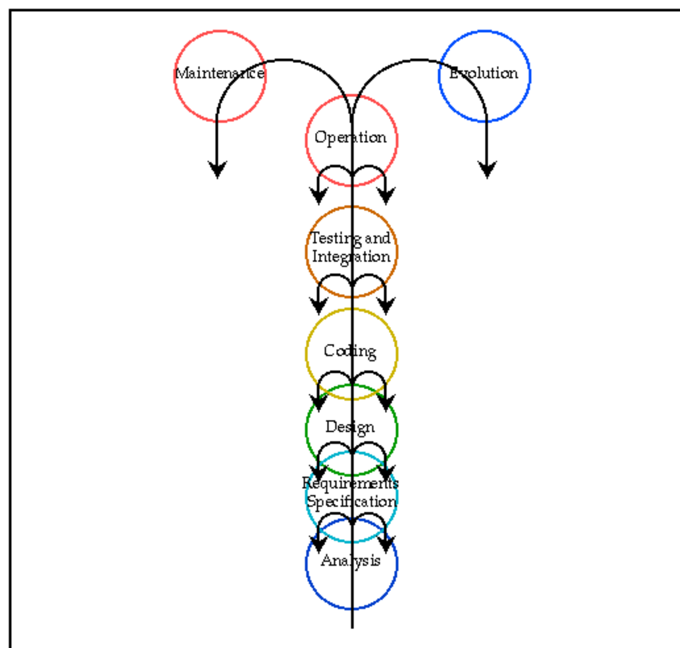
```
class Alias
{
        /* luokan sisältämät yksityiset tietojäsenet */
        private $id;
        …
        /* saantimetodit, palauttavat tiedon olion tilasta */
        public function getId()
        {
                return $this->id;
        }
        …
        /* asetusmetodit, muuttavat olion tilaa */
        function setId($_id)
        {
                $this->id = $_id;
        }
}
```

Kuva 5. Tiedon piilottaminen oliosuuntautuneessa paradigmassa.

Luiz Fernando Capretz mainitsee oliosuuntautuneen paradigman historiaa käsittelevässä artikkelissaan syitä siihen miksi korkean tason olio-ohjelmointikielet sekä *oliosuuntautunut* (object oriented) paradigma aiheuttivat kiinnostusta 1980-luvulta lähtien (Capretz, 2003). Yhtenä tällaisena syynä Capretz mainitsee ohjelmistojen kasvaneen monimutkaisuuden. Ohjelmistot olivat sisältäneet kompleksisia ominaisuuksia jo 1970-luvulla, mutta 80-luvulla ohjelmistojen yhä monimutkaistuneet piirteet sisälsivät ominaisuuksia, jotka vaativat täysin uusien ohjelmointimenetelmien kuten rinnakkaisuuden tai tekoälyn tuntemusta. Tietokoneiden leviäminen yhä suuremmalle osalle loppukäyttäjiä sai aikaan panostusta järjestelmien ja ohjelmien käyttäjäystävällisyyten. Havaittiin, että graafisten käyttöliittymien sisältämät ikkunoinnit, menuvalikot ja ikonit oli helpompi toteuttaa käyttämällä oliosuuntautuneita elinkaarimalleja sekä korkeantason olio-ohjelmointikieliä. 1980-luvulla panostettiin myös uudelleenkäytettävyyteen, jonka nähtiin olevan ratkaisu ohjelmistoalaa vaivanneeseen tuottavuuden heikkouteen. Capretz määrittelee uudelleenkäytettävyyden lyhyesti tavaksi jolla yleiskäyttöisiä ohjelmistokomponentteja voitiin liittää uusiin ohjelmistoprojekteihin sen sijaan, että komponentit olisi jouduttu suunnittelemaan ja toteuttamaan aina uudelleen alusta asti. Oliosuuntautunut paradigma tarjosi välineet, joilla uudelleenkäyttöä voitiin suorittaa. Periytyminen nähtiin yhtenä tällaisena välineenä. Oliosuuntautuneeseen elinkaarimalliin sisällytettiin samat vaiheet kuin 1970-luvulla esitetyssä vesiputousmallissakin oli, mutta oliosuuntauneen elinkaarimallin pääpaino oli näiden vaiheiden yhdistämisessä.

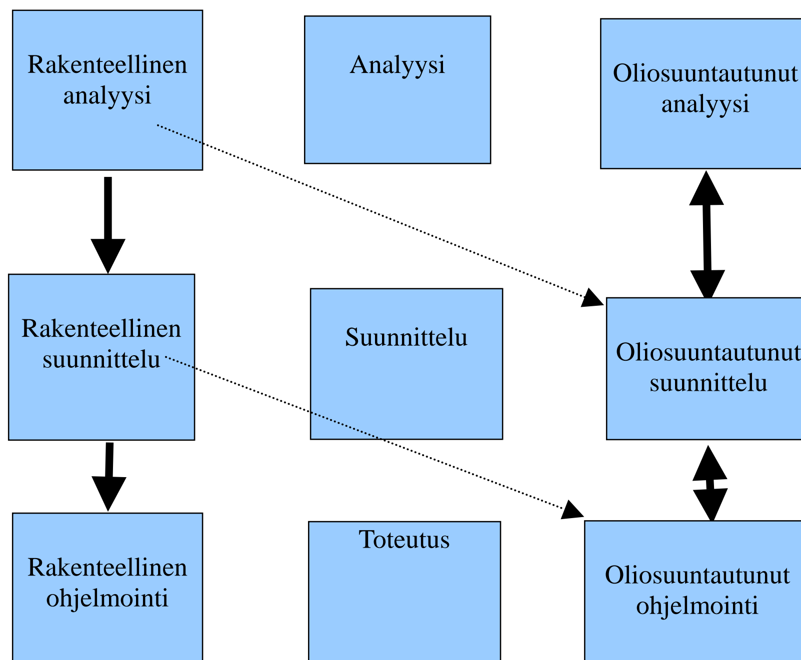
Vaiheiden yhdistämisen mahdollisti Korsonin mukaan se, että jo varhaisessa analyysivaiheessa kohdealueelta pyrittiin tunnistamaan siellä vaikuttavat osat ja mallintamaan ne olioiksi ja olioiden välisiksi suhteiksi (Korson, 1990). Suunnitteluvaiheessa näiden olioiden ominaisuuksia ja käyttäytymistä tarkennettiin. Toteutusvaiheessa ohjelmistosuunnittelija tarkensi suunnitteluvaiheessa mallinnettuja olioita ja niiden välisiä suhteita sekä toteutti ne välineillä, jotka tukivat oliosuuntautuneisuutta. Tällä tavalla oliosuunnittelussa elinkaarenvaiheet saatiin limittymään osaksi toisiaan ja lopputuloksena saatiin vaihejaoltaan yhtenäisempi elinkaarimalli, jossa käytetyt määritelmät ja termit pysyivät samoina siirryttäessä vaiheesta toiseen.

Kuvassa 6 on esitettynä oliopohjaisen elinkaarimallin kuvaamisessa käytetty suihkulähdemalli (fountain model), jonka avulla voi olla helpompaa tehdä vertailuja traditionaalisen elinkaarimallin etuihin ja haittoihin.



Kuva 6. Oliopohjaisen ohjelmiston elinkaarimalli (McCormack et al., 2005).

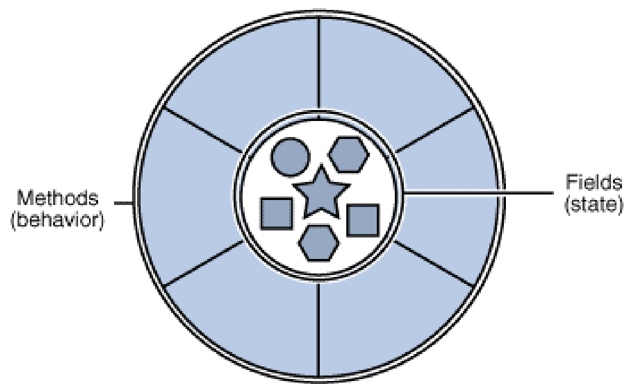
Vaikka oliosuuntautuneen paradigman odotettiinkin ratkaisevan ohjelmistotuotannossa havaitut ongelmat, huomattiin, että vuosikymmen aiemmin käyttöönotettu rakenteellinen lähestymistapa sisälsi piirteitä, joita voitiin käyttää hyväksi myös suunniteltaessa oliosuuntautuneita ohjelmistoja. Rakenteellisen ja oliosuuntautuneen lähestymistavan väliset erot ja samankaltaisuudet voidaan havaita kuvasta 7.



Kuva 7. Ohjelmistosuunnittelu metodologien välisiä eroja (Capretz, 2003).

Uusi oliosuuntautunut paradigma sisälsi myös vaikeasti ymmärrettäviä määritelmiä, joiden sisäistäminen oli kuitenkin tarpeellista, jotta niitä pystyttiin käyttämään oikeassa kontekstissaan. Korson mainitsee viisi peruskäsitettä, jotka muodostavat oliosuuntautuneen paradigman perusteet: *olio* (object), *luokka* (class), *periytyminen* (inheritance), *polymorfismi* (polymorphism) sekä *dynaaminen sidonta* (dynamic binding). Seuraavassa kukin näistä peruskäsitteistä käydään läpi tarkemmin.

Toisin kuin rakenteellisessa suunnittelussa, jossa toteutettavan järjestelmän tarvitsema data järjestettiin vaiheittaisesti moduuleihin ja niiden sisältämiin funktioihin, oliosuunnittelussa järjestelmän tarvitsema data ja sen käsittely jaetaan olioiden vastuulle. Olio sisältää itse datan sekä datan käsittelyyn tarvittavat funktiot, joita kutsutaan olioterminologialla metodeiksi tai operaatioiksi. Kuvassa 8, graafisesti esitetyllä oliolla pyritään selventämään olion *tilaa* (state) ja *käyttäytymistä* (behaviour).

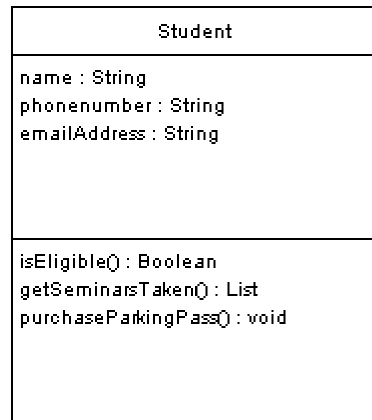


Kuva 8. Olion tila ja käyttäytyminen (The Java Tutorials, 2008).

Yksinkertaisimman mallin mukaan olio suorittaa operaation, kun se saa *pyynnön* (request) *asiakkaalta* (client) (Gamma et al., 2000). Pyynnöt ovat ainoa tapa, jolla olio saadaan suorittamaan jokin operaatio ja samalla operaatiot tarjoavat ainoan tavan, jolla olion sisäisiä tietoja voidaan muuttaa. Oliosuuntautuneen paradigman mukaan olion tilaa täytyy suojella tahattomilta muutoksilta kapseloimalla se, jolloin sen esitysmuoto ei näy muille olioille. Näin jo 1970-luvulla rakenteellisessa analyysissa ja suunnittelussa esitettyä ideaa tiedon piilottamisesta voidaan käyttää hyväksi oliosuuntautuneiden ohjelmistojen suunnittelussa sekä toteuttamisessa. Oliosuunnittelun näkökulmasta olioilla pyritään mallintamaan kohdealueella vaikuttavat entiteetit. Tätä mallintamista Gamma pitää oliosuunnittelun vaikeimpana vaiheena, koska siihen vaikuttaa monta tekijää. Samanaikaisesti on osattava ottaa huomioon olioiden väliset riippuvuudet sekä valittava suunnittelumenetelmät, jotka parantavat tai mahdollistavat järjestelmän tai sen osien uudelleenkäytön.

Olion toteutuksen määrittelee sen luokka. Luokka kokoaa yhteen joukon samankaltaisia olioita, joiden tila voi muuttua itsenäisesti, mutta jotka kaikki tarjoavat samat operaatiot muiden olioiden käyttöön. Olio luodaan luomalla luokasta *ilmentymä* (instance). Olion sanotaan tällöin olevan kyseisen luokan ilmentymä. Ohjelmoinnin näkökulmasta tämä tarkoittaa sitä, että ilmentymän luonnin yhteydessä varataan muistitilaa olion sisäisille tiedoille. Oliosuunnittelussa luokat voidaan mallintaa usealla eri tavalla. Tyypillinen tapa käsittää jonkin graafisen mallinnuskielen (UML, OMT) käytön. Kuvassa 9 esitetyn luokkakaavion luokka noudattaa laajasti käytössä olevaa UML-notaatiota (OMG, 2008). Luokkakaavion luokka kuvataan UML-notaatiolla suorakaiteen muotoisella laatikolla, joka on jaettu kolmeen osaan: ylimmässä osassa on mainittu luokan nimi. Valitsemalla havainnolliset ja kuvaavat luokkien nimet, pyritään helpottamaan luokan toiminnan ymmärtämistä. Luokkakaavion luokan keskimmäiseen osaan, voidaan kirjata luokan ilmentymien sisältämät tiedot. Olioterminologialla ilmaistuna tämä tarkoittaa tiedon mallintamista attribuuteiksi tai jäsenmuuttujiksi. Myös attribuuttien tietotyypit voidaan liittää tarvittaessa mukaan

luokkakaavioon luokkaan. Luokkakaavion luokan alimmassa osassa mainitaan luokan tarjoamat palvelut eli metodit tai jäsenfunktiot. Luokkakaavion luokassa voidaan mainita metodien nimet, paluuarvot sekä metodien tarvitsemat parametrit.

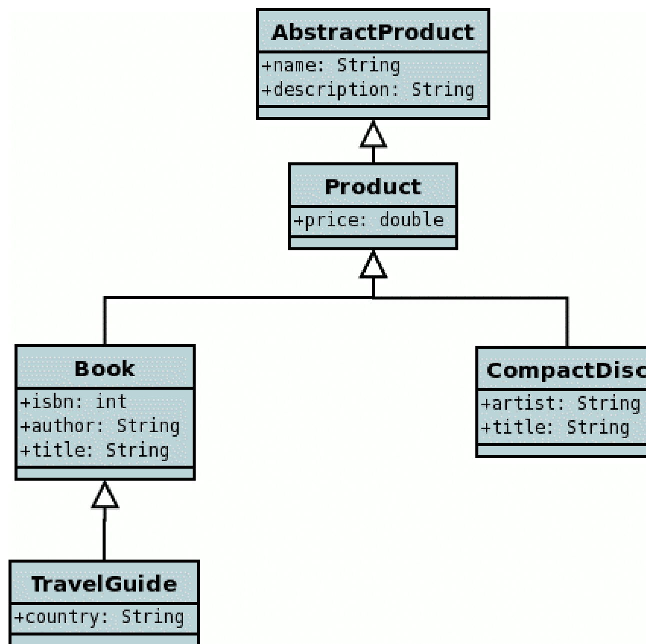


Kuva 9. UML-notaation mukainen luokka.

Oliosuunnittelussa pelkkä olioiden luokkiin jakaminen ei useinkaan riitä. Luokkien väliset suhteet on myös usein tarpeellista mallintaa oliosuuntautuneen suunnittelun aikana. Korsonin mielestä erilaisten periytymistapojen käyttämisellä pystytään olioparadigmassa parhaiten kuvaamaan uudelleenkäytön etuja (Korson, 1990). Periytymisen kuvaamisessa luokat järjestetään hierarkkiseen muotoon, jossa *yliluokka* (super class, parent class) tarjoaa käyttäytymisen sekä joskus datan *aliluokan* (child class, subclass) käyttöön. Kuva 10 pyrkii selventämään kuinka periytymishierarkia voidaan mallintaa käyttämällä UML-notaatiota.

Kun oliolle lähetetään pyyntö, riippuu pyynnön toteutus sekä pyynnöstä, että pyynnön vastaanottaneesta oliosta. Samanlaisia pyyntöjä vastaanottavat oliot voivat käyttää erilaisia toteutuksia pyynnöt täyttävissä operaatioissa. Ohjelman suorituksen aikana tehtävää pyynnön yhdistämistä olioon ja johonkin sen operaatioon kutsutaan dynaamiseksi sidonnaksi. Dynaaminen sidonta tarkoittaa Gamman mielestä sitä, että pyyntöä ei sidota operaation mihinkään tiettyyn toteutukseen ennen kuin ohjelman suorituksen aikana. Dynaamista sidontaa käyttämällä voidaan toteuttaa ohjelmia, jotka nojautuvat tietyn rajapinnan omaavan olion olemassaoloon tietäen, että mikä tahansa olio, jolla on oikea rajapinta, vastaanottaa pyynnön (Gamma et al., 2000). Tämän lisäksi dynaamisella sidonnalla voidaan korvata identtiset rajapinnat omaavia olioita toisillaan ohjelman suorituksen aikana. Tätä vaihdettavuutta kutsutaan monimuotoisuudeksi. Monimuotoisuuden vuoksi asiakasolion ei tarvitse edellyttää muilta olioilta juuri mitään muuta kuin että ne toteuttavat tietyn rajapinnan. Monimuotoisuus yksinkertaistaa asiakasolioiden määrittelyä, eristää oliot toisistaan ja sallii

olioiden varioida keskinäisiä suhteitaan ohjelman suorituksen aikana.



Kuva 10. Periytymishierarkia UML-notaatiolla kuvattuna (Java Persistent Objects, 2008).

Analyysi- ja suunnitteluvaiheita on pidetty syystäkin haastavimpina oliosuunnittelun vaiheina. Analyysivaiheessa kohdealueella vaikuttavat entiteetit on pystyttävä mallintamaan olioiksi, mutta joskus saatetaan joutua tilanteeseen, jossa huomataan, ettei niille löydy vastaavuutta reaalimaailmasta. Toisaalta toteutettavan järjestelmän sisältävien olioiden koko ja lukumäärä voivat myös vaihdella paljon. Olioilla voidaan esittää mitä tahansa laitteistokomponenteista kokonaisiin sovelluksiin.

Oliosuunnittelun aikana täytyy pystyä tekemään ratkaisu siitä mitkä kohteet ylipäätään tulisi esittää olioina. *Suunnittelumallit* (Design Patterns) on kehitetty ratkaisemaan tällaisia oliosuunnittelun aikana eteentulevia suunnitteluongelmia. Suunnittelumallit eivät esitä toteutusta siihen kuinka tietty ongelma voidaan ratkaista ohjelmoinnin näkökannalta, vaan ne tarjoavat toteutuskieli riippumattoman abstraktin kuvauksen ongelman ratkaisuun liittyvistä olioista ja niiden välisistä suhteista. Suunnittelumallit tarjoavat esimerkiksi mahdollisuuden olion sisäisen tilan kapselointiin ja tallentamiseen siten, että olion voi myöhemmin palauttaa aikaisempaan tilaansa. Tämä Memento (Muisto) -suunnittelumalli tarjoaa vain yhden esimerkin Erich Gamman esittämästä 23 suunnittelumallista.

Judith Bishop mainitsee artikkelissaan *Language Features Meet Design Patterns: Raising the Abstraction Bar* 10 vuotta sitten käydyn intensiivisen keskustelun ohjelmistosuunnittelun alalla siitä pitäisikö uusien korkeantason ohjelmointikielien sisältää piirteitä, joilla

pystyttäisiin suoraan tukemaan suunnittelumallien käyttöä (Bishop, 2008).

Bishop ihmettelee, että vaikka uudet ohjelmointikielet kuten C# tarjoavat nykyisin korkeantason piirteitä aina geneerisyydestä iteraattoreihin, suunnittelumallien toteutukset kirjallisuudessa nojautuvat kuitenkin vielä vahvasti periytymiseen sekä koostamiseen eikä niiden rakenteita ole täten laajennettu tai päivitetty vastaamaan nykyisiin korkean tason ohjelmointikielien piirteisiin.

Eden ja Hirschfeld kritisoivat myös suunnittelumallien dokumentointia alan kirjallisuudessa kahdelta näkökannalta: ensiksikin suunnittelumallien kuvaukset ovat liian epämuodollisia, jotta niillä pystyttäisiin kuvaamaan tarkasti mallin käyttökelpoisuutta. Kirjallisuudessa esitetyt mallien kuvaukset ovat sekavia ja näin aiheuttavat suunnittelijoiden keskuudessa entistä suurempaa sekaannusta. Edes mallien suunnittelijoiden keskuudessa ei vallitse täyttä yhteisymmärrystä siitä mihin tarkoitukseen tiettyä mallia oikeastaan voidaan käyttää. Toiseksi suunnittelumalleihin liittyväksi ongelmaksi Eden ja Hirschfeld mainitsevat jäsentelemättömän tiedon, joka on aiheutunut suunnittelumallien kasvaneesta lukumäärästä. Valtava malleihin liittyvä informaatio vaatisi heidän mukaansa tehokkaan tavan sen jäsentelyyn (Eden ja Hirschfeld., 2001).

2.2 Suunnittelumallien luokittelu

Ohjelmistosuunnittelijat kohtaavat usein vaikeuksia oliopohjaisten ja nykypäivän ohjelmistovaatimukset täyttävien sovelluksien suunnittelussa ja toteuttamisessa. Ohjelmistot ovat yhä enenevissä määrin kompleksisia kokonaisuuksia, joiden täytyy monimutkaisten kohdealue vaatimusten lisäksi olla käyttäjäystävällisiä, joustavia sekä ylläpidettäviä.

Ohjelmiston ylläpidettävyys on ohjelmiston elinkaaren vaihe, missä ohjelmistoon tehdään muutoksia esimerkiksi silloin kun havaitaan, että ohjelmistoon on saatava liitettyä uusi toiminnallisuus. Suurin osa ohjelmiston elinkaaren kustannuksista syntyy tässä vaiheessa. Esimerkiksi Macario Polo mainitsee ylläpitovaiheessa suoritettavien toimenpiteiden aiheuttavan joissain tapauksissa 67-90% ohjelmiston kokonaiskustannuksista (Polo et al., 1999). Scott Edgerton mainitsee toisaalta artikkelissaan tapoja, joilla näitä kustannuksia voidaan saada huomattavasti alennettua. Yksi näistä tavoista liittyy ohjelmistoyrityksen toimintakulttuuriin. Jos yrityksessä kannustetaan johdonmukaiseen suunnitteluun noudattamalla yleisesti sovittuja toimintatapoja, ylläpitoon liittyviä kustannuksia saadaan madallettua. Yhtenä esimerkkinä tällaisista toimintatavoista Edgerton mainitsee *suunnittelumallien* (design patterns) systemaattisen käytön (Edgerton et al., 1999).

Erich Gamma määrittelee suunnittelumalleihin liittyvässä klassikkoteoksessaan *Design Patterns -Elements of Reusable Object-Oriented Software* suunnittelumallien olevan abstrakteja kuvauksia keskenään vuorovaikutuksessa olevista olioista ja luokista joita voidaan käyttää useasti toistuvien suunnitteluongelmien ratkaisemiseen (Gamma at al., 2000). Erich Gamman alkuperäisen kirjan nimi paljastaa kaksi oleellista suunnittelumalleihin liittyvää ominaisuutta: *ohjelmistojen uudelleenkäyttö* (software reuse) sekä *olio-ohjelmointi* (object oriented programming).

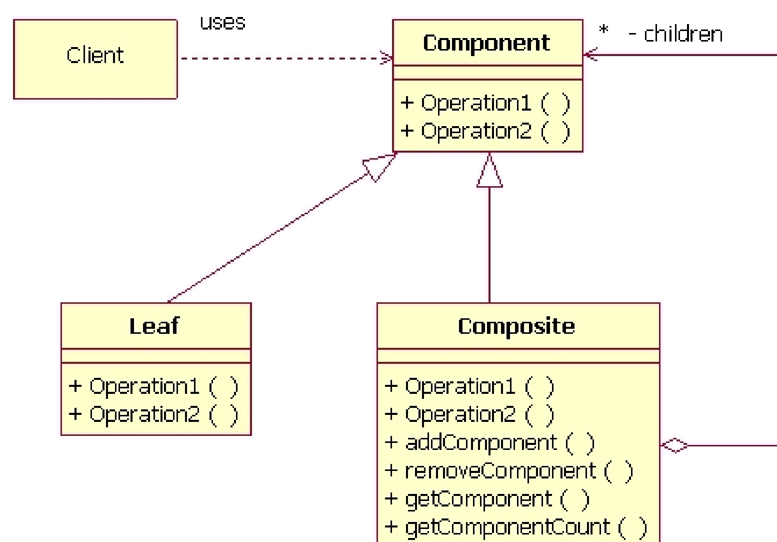
Uudelleenkäytettävyys liittyy suunnittelumalleihin siten, että suunnittelumallit tarjoavat suunnittelun avuksi hyviksi havaittuja, valmiiksi testattuja ratkaisuja, jolloin suunnittelutyötä ei tarvitse aloittaa aina alusta. Ohjelmistosuunnittelija voi tutustua hyvin dokumentoituihin suunnittelumalleihin ja valita niistä tarkoitukseensa parhaiten sopivan.

Suunniteltaessa oliopohjaisia ohjelmistoja, on ohjelmiston normaalin elinkaaren aikana tehtävän tarkan vaatimusmäärittelyn lisäksi osattava löytää kohdealueella vaikuttavat oliot. Nämä oliot on osattava koota luokkiin sekä määriteltävä kuinka nämä yksittäiset oliot toimivat tarvittaessa yhteistössä muiden olioiden kanssa. Gamman mukaan suunnittelumalli sisältää yleensä neljä keskeistä osaa:

1. **Mallin nimellä** pyritään kuvaamaan muutamalla sanalla suunnittelun kohteena oleva ongelma. Esimerkiksi suunnittelumallin nimi Iterator kuvaa abstraktia tapaa, jota voidaan käyttää kun on tarpeen käydä läpi erilaisia tietorakenteita tai kokoelmia. Iterator-nimi ei paljasta yksityiskohtia toteutuksesta, vaan se tarjoaa ohjelmistosuunnittelijoille mm. tavan keskustella suunnitteluratkaisujen hyvistä ja huonoista puolista.
2. **Ongelma** ja ympäristö, jossa se esiintyy, kuvaa mallin soveltamiskohteita. Se voi kuvata erityisiä suunnitteluongelmia kuten miten esittää tietty algoritmi olioina. Joskus ongelma sisältää listan ehtoja, joiden on täytyttävä, jotta mallin soveltaminen olisi järkevää.
3. **Ratkaisussa** kuvataan elementit, joista suunnitteluratkaisu koostuu. Ratkaisussa kuvataan myös näiden elementtien vastuut, niiden väliset suhteet sekä keskinäinen yhteistyö. Huomattava on Gamman mielestä kuitenkin se, ettei ratkaisu kuvaa mitään yksittäistä konkreettista toteutusta, vaan malli on ikään kuin runko, jota voidaan soveltaa usealla eri tavalla. Ohjelmistosuunnittelijan vastuulle jää esimerkiksi se, millä ohjelmointikielellä malli lopulta toteutetaan.

4. **Seuraukset** kuvaavat mallin soveltamisen tuloksia ja sen hyötyjä sekä haittoja. Ohjelmistoista puhuttaessa seuraukset ovat usein muistiin ja suorituskykyyn liittyviä. Gamma muistuttaa uudelleenkäytön olevan yksi oliopohjaisen suunnittelun tärkeimmistä tavoitteista. Tämän vuoksi suunnittelumallien sisältämissä seurauksissa selostetaan mallin käytön vaikutus ratkaisun muunneltavuuteen, laajennettavuuteen sekä siirrettävyyteen.

Suunnittelumallit voidaan kuvata usealla tavalla. Yleisin käytössä oleva tapa on graafinen, ohjelmistotuotannossa standardin kuvauskielen asemassa olevan UML-notaation mukainen esitystapaa, jossa oliot kuvataan luokkakaavioilla ja periytymissuhteet luokkakaavioiden välisillä nuolilla. Gamma käyttää puolestaan kirjassaan graafisten kuvausten esittämiseen Object Modeling Technique (OMT) notaatiota (Derr, 1995). Kuvassa 11 on esimerkki suunnittelumallin kuvauksesta UML-notaatiolla ilmaistuna.



Kuva 11. Uml-notaation mukainen suunnittelumallin kuvaus (Java World, 2008a).

Graafisten kuvausten etuna on se, ettei kuvaus sido suunnittelumallin toteutusta mihinkään tiettyyn ohjelmointikieleen. Ohjelmointikielen valinta on kuitenkin tärkeää, sillä valittavan ohjelmointikielen tulisi sisältää ominaisuuksia, joiden avulla suunnittelumallien toteuttaminen on tarkoituksen mukaista. Kuvassa 11 esitetyn Composite-mallin graafisesta kuvauksesta voidaan huomata, että suunnittelumallit on suunniteltu lähinnä oliopohjaisten suunnittelu-ongelmien ratkaisemiseen sen sijaan, että käytettävissä olisi jokin proseduraalinen ohjelmointikieli kuten Pascal tai C. Näissä ohjelmointikielissä ei ole mekanismeja, joiden avulla esimerkiksi periytyminen tai kapselointi voitaisiin helposti toteuttaa.

Gamma kuitenkin huomauttaa, että graafiset suunnittelumallien kuvaukset eivät ole yksistään riittäviä. Ne tallentavat ainoastaan suunnitteluprosessin lopputuotteen olioiden ja luokkien välisinä suhteina. Jotta suunnitteluratkaisua voitaisiin uudelleenkäyttää, on Gamman mielestä kirjattava ylös myös ratkaisuun johtaneet päätökset, suunnittelun aikana eteen tulleet vaihtoehdot sekä valitun ratkaisun hyvät ja huonot puolet (Gamma et al., 2000).

Gamma esittääkin suunnittelumallien esittämiseen/kuvaamiseen yhtenäistä formaattia, jonka avulla suunnittelumallit saadaan helpommin ymmärrettäviksi ja käytettäviksi. Tässä yhteydessä on syytä mainita luettelon omaisesti ainoastaan muutama tärkeimmistä osista:

- **Mallin nimi**, kertoo ytimekkäästi mallin olemuksen. Hyvän ja kuvaavan nimen valinta on tärkeää, sillä siitä tulee osa suunnittelusanastoa.
- **Mallin tarkoitus**. Tarkoituksen tulisi vastata esimerkiksi seuraaviin kysymyksiin: Mitä suunnittelumalli tekee? Mihin tiettyyn suunnittelukohteeseen tai ongelmaan se soveltuu?
- Suunnittelumallin käyttämisen **perusteluissa** kuvataan suunnitteluongelma ja se, miten mallin luokka ja oliorakenteet ongelman ratkaisevat.
- Mallin **soveltuvuus** kuvaa tilanteet, joissa suunnittelumallia voidaan soveltaa.
- **Rakenteella** suunnittelumalli kuvataan graafisesti. Kuvauskielenä voidaan käyttää joko UML- tai OMT-notaatiota.
- **Osallistujilla** esitetään malliin liittyvät luokat ja/tai oliot sekä niiden vastuut.
- **Yhteistyösuhteilla** pyritään esittämään kuinka osallistujat toimivat yhteistyössä toteuttaakseen vastuunsa.
- **Mallikoodin** avulla kuvataan kuinka suunnittelumallin toteuttaminen onnistuu käyttämällä valitun ohjelmointikielen ominaisuuksia.

Gamma kuvaa kirjassaan 23 suunnittelumallia, joiden voidaan nähdä muodostavan pohjan suunnittelumalleille. Nämä suunnittelumallit sisältävät yleisimmät ratkaisut oliopohjaisten ohjelmistojen suunnittelussa eteen tuleviin ongelmiin. Näiden suunnittelumallien lisäksi on suunniteltu useita muitakin suunnittelumalleja eri käyttötarkoituksiin. Esimerkiksi Clifton Nock kuvaa kirjassaan *Data Access Patterns: Database Interactions in Object-Oriented Applications* olioiden pysyvyyteen liittyviä suunnittelumalleja (Nock, 2004). Olioiden pysyvyydellä tarkoitetaan tässä yhteydessä tiedon tallentamista tietokantaan ja vastaavasti tiedon muuntamista tietokannasta sovelluksen ymmärtämään muotoon.

Martin Fowler on puolestaan koonnut yhteen suunnittelumalleja ja niiden ratkaisuja kirjaansa *Analysis Patterns: Reusable Object Models* (Fowler, 1997).

Suunnittelumallien suuresta lukumäärästä johtuen Gamma on nähnyt tarpeelliseksi niiden luokittelun kahden kriteerin mukaan. Ensimmäinen kriteeri **tarkoitus**, kuvaa mallin toimintaa. Malli voi kohdistua joko luontiin, rakenteeseen tai käyttäytymiseen. Luontimallit käsittelevät nimensä mukaisesti olioiden luontiprosessia.

Rakennemallit koskevat luokkien ja olioiden koosteita. Käyttäytymismallit puolestaan käsittelevät tapoja, joilla luokat ja oliot ovat vuorovaikutuksessa ja joilla ne jakavat vastuita. Toinen kriteeri, **kohde**, kertoo liittyykö malli ensisijaisesti luokkiin vai olioihin. Luokkamallit käsittelevät luokkien ja niiden aliluokkien välisiä suhteita. Nämä suhteet kiinnitetään periytymisen kautta, joten ne ovat staattisia eli käännösaikaisia. Oliomallit sen sijaan käsittelevät olioiden välisiä suhteita, joita voidaan muuttaa ajonaikaisesti ja jotka ovat dynaamisempia (Gamma et al., 2000). Kuvassa 12 esitetty taulukkomuotoinen luokittelu pyrkii selventämään suunnittelumallien välistä luokittelua.

		Tarkoitus	
Kohde	**Luonti**	**Rakenne**	**Käyttäytyminen**
Luokka	Factory method	Adapter	Interpreter Template method
Olio	Abstract factory Builder Prototype Singleton	Adapter Bridge Recursion Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Kuva 12. Suunnittelumallien luokittelu (Gamma et al., 2000).

2.3 Suunnittelumallit ohjelmistojen uudelleenkäytössä

Charles Krueger määrittelee ohjelmistojen uudelleenkäytön tarkoittavan jo olemassa olevien ohjelmistoartefaktien käyttämistä uuden ohjelmiston suunnittelussa ja toteuttamisessa (Krueger, 1992). Kimberly M. Jordan määrittelee uudelleenkäytön samalla tavalla, mutta huomauttaa, että vaikka ohjelmistojen uudelleenkäytöllä usein tarkoitetaan ohjelmakoodin uudelleenkäyttöä, ohjelmistosuunnittelu sisältää lukuisia muitakin uudelleenkäytettäviä artefakteja. Esimerkkeinä näistä Jordan mainitsee ohjelmiston suunnittelun sekä toteuttamisen aikana syntyneet kirjalliset tuotokset kuten: vaatimusmäärittely- ja suunnitteludokumentit (Jordan, 1997). Frakes mainitsee 10 potentiaalista uudelleenkäytön kohdetta ohjelmistosuunnittelun elinkaaren aikana:

- arkkitehtuurit
- lähdekoodi
- data
- rakenteet
- dokumentaatio
- estimaatit (kaavaimet)
- käyttöliittymät
- suunnitelmat
- vaatimukset
- testitapaukset

Frakesin esittämissä uudelleenkäytettävissä artefakteissa painotus on koko ohjelmiston elinkaaren aikana toteutettavissa artefakteissa (Frakes, 1996). Tässä tutkielmassa uudelleenkäyttöä tarkastellaan kuitenkin ohjelmakoodia sisältävien *ohjelmistokomponenttien* (software component) näkökulmasta. Ohjelmistokomponentilla voidaan tarkoittaa yksinkertaisesti ohjelmiston sisältämää elementtiä, joka tarjoaa ennalta määritettyjä palveluita ja pystyy kommunikoimaan muiden komponenttien kanssa. Oliosuuntautuneessa paradigmassa ohjelmistokomponentin nähdään koostuvan yhdestä tai useammasta oliosta.

Clemens Szyperski ja David Messerschmitt tarjoavat tarkentavan määritelmän ja mainitsevat ohjelmistokomponentilla olevan seuraavat ominaisuudet (Szyperski ja Messerschmitt, 2003):

- konstekstiriippumattomuus, komponenttia voidaan käyttää useassa yhteydessä
- monikäyttöisyys
- kapselointi, komponentti ei paljasta sisäistä rakennettaan
- komponenttia voidaan käyttää muiden komponenttien yhteydessä

Uudelleenkäytettävän ohjelmistokomponentin (reusable component) tulisi puolestaan Ramachandranin mukaan esittää ainoastaan yhtä abstraktiota sekä sen tulisi olla täysin riippumaton muista abstraktioista (Ramachandran, 2005). Nämä kaksi vaatimusta sisältävät kaksi tärkeää, erityisesti oliosuuntautuneessa paradigmassa, käytössä olevaa määritelmää: *koheesion* (cohesion) ja *kytkennän* (coupling). Tässä yhteydessä on kuitenkin syytä mainita, että vaikka kyseiset termit ovat laajasti käytettyjä oliosuuntautuneessa paradigmassa, niiden voidaan nähdä syntyneen jo yli 30 vuotta sitten. Tästä näkökulmasta katsoen oliosuuntautuneisuus ei ole tuonut mitään uutta ohjelmistosuunnittelun alalle. Käytettävä konteksti on vain muuttunut. Seuraavassa koheesio ja kytkentä määritellään lyhyesti tarkemmin oliosuuntautuneen paradigman näkökulmasta.

Koheesiolla mitataan kuinka pitkälle moduulissa tai luokassa oleva ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen. Koheesion määrä ilmaistaan yleensä vain toteamalla luokalla olevan joko korkea koheesio tai matala koheesio. Oliosuuntautuneessa paradigmassa komponentit pyritään suunnittelemaan ja toteuttamaan suunnittelumenetelmillä ja ohjelmointikielen mekanismeilla, jotka edesauttavat korkeaa koheesiota. Luokat, jotka omaavat matalan koheesion on yleisesti pidetty vaikeina ylläpitää ja testata. Tähän on nähty vaikuttavan erityisesti luokan tarjoamien palveluiden/vastuiden jakaantuminen toisistaan riippumattomiin toimintoihin. Oliosuunnittelun ja ohjelmoinnin kannalta tällä tarkoitetaan luokan sisältämien metodien suunnittelemista ja toteuttamista siten, että luokka sisältää vain ja ainoastaan kiinteästi toisiinsa liittyviä metodeita.

Oliosuuntautuneessa paradigmassa kytkennän nähdään usein olevan yhteydessä koheesioon. Kytkennällä tarkoitetaan luokan riippumattomuutta tai riippuvuutta toisista luokista. Alhainen kytkentä tarkoittaa sitä, että toimiakseen yhteistyössä muiden luokkien kanssa, luokan ei tarvitse tietää näiden sisäistä rakennetta. Tämä ominaisuus perustuu suoraan kohdassa 2.1 mainittuun tiedon piilottamiseen. Kytkennän määrä ilmaistaan myös yksinkertaisesti toteamalla luokalla olevan vahva tai heikko kytkentä. Oliosuunnittelun ja ohjelmoinnin

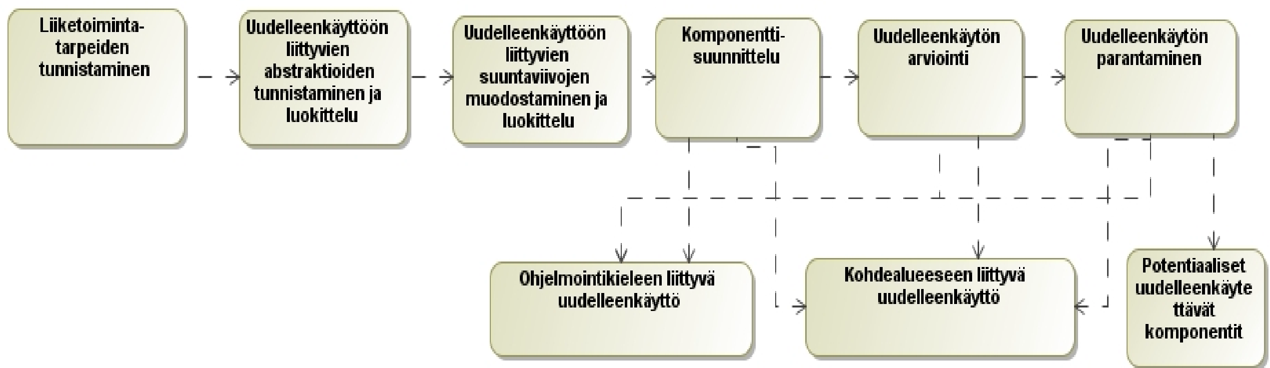
kannalta kahden luokan A ja B välinen kytkentä voi kasvaa jos:

- A:lla on attribuutti/tietojäsen, joka viittaa B-luokkaan
- A:lla on metodi, joka viittaa B-luokkaan, joko metodin paluuarvon tai parametrin kautta
- A perii B-luokan tai toteuttaa sen tarjoaman rajapinnan.

Koheesio ja kytkentä on otettava huomioon suunniteltaessa ja toteutettaessa mahdollisimman uudellenkäytettäviä ja ylläpidettäviä luokkia/moduuleita. Oliosuunnittelun näkökulmasta suunnittelumallien käytöllä saadaan aikaan suunnitteluratkaisuja, jotka edesauttavat esimerkiksi korkeaa koheesiota ja matalaa kytkentää.

Uudelleenkäytöllä on havaittu olevan ilmeinen vaikutus tuottavuuden parantamisessa sekä ohjelmistokustannuksien alentamisessa (Jordan, 1997). Organisaatio ei kuitenkaan hyödy uudelleenkäytön tarjoamista eduista, jollei uudelleenkäyttö ole systemaattista ja tarkkaan harkittua (Frakes, 1996). Organisaation täytyy myös pystyä mittaamaan edistymistään uudelleenkäyttötapojen omaksumisessa sekä tunnistamaan tehokkaimmat uudelleenkäyttö-strategiat. Frakes esittelee artikkelissaan tähän tarkoitukseen suunniteltuja mittareita ja malleja. Frakes esittää myös kypsyysmallin, jonka mukaan organisaatio pystyy hahmottamaan millä tasolla se on systemaattisten uudelleenkäyttö tapojen harjoittamisessa ja kuinka se pystyy kehittämään näitä tapoja, jotta uudelleenkäyttöä harjoitettaisiin mahdollisimman systemaattisesti jokaisella organisaation tasolla. Ramachandranin mukaan organisaatio hyötyy uudelleenkäytettävyyden hyödyistä parhaiten omaksumalla toimintaperiaatteen, jossa kiinnitetään huomio kehitystyöhön, joka tähtää uudelleenkäytettävyyteen (development for reuse) (Ramachandran, 2005). Ramachandran kuvaa tätä toimintaperiaatetta prosessiksi, jonka aikana pyritään tuottamaan potentiaalisesti uudelleenkäytettäviä ohjelmistokomponentteja.

Ramachandran tekee tarkoituksellisen eron tämän prosessin sekä normaalin ohjelmisto-kehitystyön välillä, jossa kehitystyö tapahtuu uudelleenkäytettäviä ohjelmistokomponentteja käyttämällä. Uudelleenkäytettävyyteen tähtäävän kehitystyön prosessivaiheet on esitetty tarkemmin Ramachandrania mukaillen kuvassa 13.



Kuva 13. Uudelleenkäytettävyyteen tähtäävä suunnitteluprosessi.

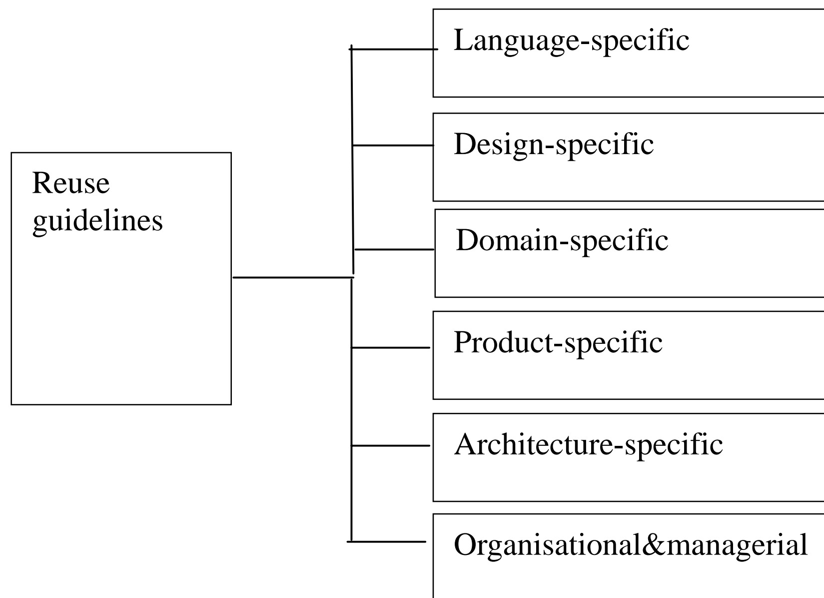
Uudelleenkäytettävyyteen tähtäävä kehitystyö sisältää useampia suuntaviivoja, jotka on luokiteltu kuvassa 14. Ramachandran keskittyy kuvaamaan artikkelissaan näistä kahta:

- ohjelmointikielen valintaan ja sen käyttöön liittyvät suuntaviivat
- kohdealueeseen liittyvät suuntaviivat.

Useat modernit ohjelmointikielet sisältävät piirteitä, jotka tukevat uudelleenkäyttöä. Tästä huolimatta pelkkä ohjelmakoodin kirjoittaminen näillä ohjelmointikielillä ei edistä uudelleenkäyttöä. Ohjelmistokomponentit täytyvät olla uudelleenkäyttöä varten suunniteltuja. Tämä suunnittelutyö toteutetaan käyttämällä hyväksi ohjelmointikielien tukea uudelleenkäytölle.

Kuvassa 13 esitetyssä mallissa prosessi alkaa kohdealueen ja liiketoimintatavoitteiden tunnistamisella. Kohdealueanalyysiä pidetään välttämättömänä vaiheena, jotta uudelleenkäyttö olisi mahdollisimman tehokasta. Uudelleenkäytettävien abstraktioiden tunnistaminen ja luokitteleminen pohjautuu edellisessä vaiheessa tehtyyn kohdealue- analyysiin. On osattava erottaa tärkeät kohdealueella vaikuttavat abstraktiot ja yritettävä selvittää kuinka usein näitä abstraktioita käytetään toteutettavan järjestelmän kannalta. Tämä tieto saadaan usein haastattelemalla ihmisiä, jotka tuntevat kyseessä olevan kohdealueen. Tietoa voidaan myös hankkia tutustumalla kohdealueeseen liittyvään kirjallisuuteen tai olemassa oleviin järjestelmiin. Ramachandranin mukaan seuraavassa vaiheessa tulisi pyrkiä tunnistamaan ne suunnittelun/ohjelmointikielten tarjoamat rakenteet, jotka tukevat uudelleenkäyttöä.

Ohjelmointikielen valinta on erittäin tärkeää, sillä uudelleenkäyttöön liittyvät suuntaviivat tulisi pystyä esittämään valitun kielen tukemilla mekanismeilla. Uudelleenkäytön arviointi on automatisoitu prosessin vaihe, missä suunniteltuja ja toteutettuja ohjelmistokomponentteja arvioidaan sen perusteella, kuinka hyvin ne noudattavat laadittuja suuntaviivoja.



Kuva 14. Uudelleenkäytön suuntaviivojen luokittelu (Ramachandran, 2005).

Tämän vaiheen lopputuloksena voidaan laatia erillinen arviointiraportti, johon voidaan palata, kun suuntaviivoja tarvitsee tarkentaa tai muuttaa. Viimeisessä vaiheessa uudelleenkäyttöön liittyviä suuntaviivoja muokataan tarvittaessa. Tässä vaiheessa voidaan nojata uudelleen-käytön arviointivaiheessa luotuun arviointiraporttiin. Tarkoituksena on kuitenkin se, että luotuja suuntaviivoja noudatetaan ja luotuja menetelmiä yritetään kehittää edelleen.

Uudelleenkäyttömekanismien hyödyntäminen suunnittelumallien näkökulmasta tarjoaa mah-dollisuuksia joustavien, ylläpidettävien ja uudelleenkäytettävien suunnitteluratkaisujen käyttämiseen. Gamma mainitsee kirjassaan seuraavat uudelleenkäyttömekanismit (Gamma, 2000):

- periytymis- ja koostamissuhteet
- pyyntöjen delegointi olioille
- muunneltavuuden ennakoiminen suunnittelun aikana.

Oliopohjaisissa järjestelmissä eniten käytettyjä tekniikoita toiminnallisuuden uudelleenkäyttämiseksi ovat *periytyminen* (inheritance) ja *koostaminen* (object compositon). Kohdassa 2.1 mainittiin, että periytymisessä *yliluokka* (super class, parent class) tarjoaa käyttäytymisen sekä joskus datan *aliluokan* (child class, subclass) käyttöön. Periytymiseen perustuvaa uudelleenkäyttöä kutsutaan usein **white box** -tyyppiseksi uudelleenkäytöksi. Termi white box viittaa läpinäkyvyyteen, periytymisessä yliluokan sisäinen rakenne usein paljastetaan tarkoituksella aliluokalle. Periytyminen määritellään staattisesti käännösaikana ja

sen käyttö on suoraviivaista, koska ohjelmointikielet tukevat sitä suoraan. Php-skriptikielessä voidaan esimerkiksi kuvassa 10 kuvattu Book-aliluokan ja Product-yliluokan välinen suhde mainita yksinkertaisesti seuraavalla kuvassa 15 esitetyllä tavalla.

Book-luokka voi käyttää yliluokassa määriteltyä ja toteutettua getPrice()-metodia suoraan tai jos Book-luokka tarvitsee getPrice()-metodista erilaisen toteutuksen, se voi ylikirjoittaa sen (override) haluamallaan tavalla. Gamma huomauttaa, että periytymisellä on huonojakin puolia. Varsinkin staattisten (vahvasti tyypitettyjen) ohjelmointikielien tapauksessa yliluokalta perittyä toteutusta ei voi muuttaa ohjelman suorituksen aikana, koska periytyminen kiinnitetään ohjelman käännösaikana. Toisena periytymisen haittapuolena Gamma pitää sitä, että periytyminen rikkoo kapseloinnin, koska yliluokan toteutuksen yksityiskohdat paljastuvat aliluokalle. Aliluokan toteutus on niin riippuvainen yliluokan toteutuksesta, että vähäinenkin muutos yliluokan toteutuksessa pakottaa muuttamaan myös aliluokkaa. Tämä aiheuttaa ongelmia, kun aliluokkaa yritetään uudelleenkäyttää. Mikäli perityn toteutuksen jokin yksityiskohta ei sovi uuteen ongelmakenttään, yliluokka täytyy toteuttaa uudelleen tai vaihtaa se johonkin sopivampaan luokkaan. Tällainen riippuvuus rajoittaa muunneltavuutta ja viime kädessä myös uudelleenkäytettävyyttä (Gamma et al., 2000). Yhtenä mahdollisena ratkaisuna tähän Gamma ehdottaakin periytymisen käyttämistä vain abstrakteista luokista, koska abstrakti luokka tarjoaa vain vähän tai ei ollenkaan toteutusta. Kuvassa 10 esitetty periytymishierarkia kuvaa periytymistä abstraktista yliluokasta.

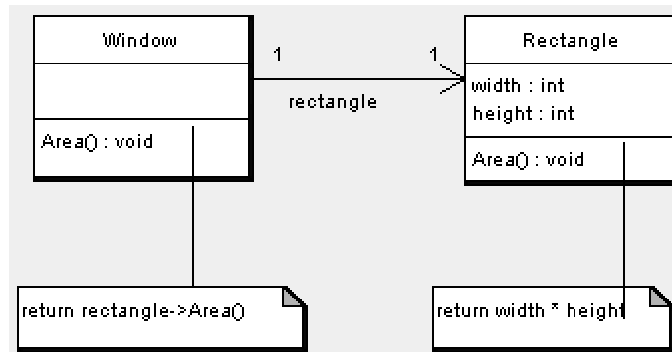
Olioiden koostamista pidetään vaihtoehtona luokkaperiytymiselle. Uutta toiminnallisuutta luodaan yhdistämällä eli koostamalla olioita monimutkaisemman toiminnallisuuden aikaansaamiseksi. Tämä tarkoittaa toteutuksen kannalta sitä, että koosteolio määritellään luokan attribuutiksi. Tällaista uudelleenkäyttötapaa kutsutaan **black box** -uudelleenkäytöksi, koska olion sisäiset yksityiskohdat eivät ole näkyvissä luokalle, joka sisältää tällaisen koosteolion. Koostaminen eroaa periytymismekanismeista siinä, että koostamisen yhteydessä koosteoliot muodostetaan dynaamisesti ohjelman suorituksen aikana kiinnittämällä viitteitä toisiin olioihin. Koostaminen ei riko kapselointia, koska koosteolioita käytetään vain sen oman rajapinnan kautta. Liitteessä 2 on kuvattuna yksinkertainen esimerkki koostamisesta Java-ohjelmointikielellä toteutettuna.

```
class Product
{
        /* luokan omat tietojäsenet, $price aliluokan käyttöön */
        ...
        protected $price;
        /* omat metodit, jotka aliluokka voi ylikirjoittaa */
        public function getPrice(){...}
        ...
}
class Book extends Product
{
        /* luokan omat tietojäsenet */
        private $isbn;
        private $title;
        private $author;
        /* price periytyy yliluokalta, ei tarvitse erikseen mainita */
        /* omat metodit */
        /* perityt metodit, voidaan käyttää suoraan */
        public function getPrice(){...}
}
```

Kuva 15. Periytyminen toteutuksen kannalta.

Suunnittelun aikana koostesuhteita voidaan käyttää hyväksi, jos huomataan, että tietty luokka voi sisältää toisen luokan olioita, ts. jos tietty luokka koostuu muista luokista/olioista. Tämä luokkien välinen suhde ilmaistaan yleensä englannin kielisessä kirjallisuudessa **has-a**-suhteeksi. Luokka A koostuu/sisältää luokan B olioita. Vastaavasti periytymissuhteessa aliluokan nähdään olevan erityistapaus yliluokasta, jolloin käytetään **is-a**-suhdetta kuvaamaan luokkien välistä suhdetta. Jos luokka A on luokan B yliluokka, luokka B on erityistapaus luokasta A. Tällainen luokkien välinen jako onnistuu, kun kohdealueella mallinnettavat abstraktiot ovat konkreettisia. Tilanne muuttuu vaikeammaksi, kun mallinnettavat kohteet muuttuvat abstrakteimmiksi. Suunnittelumallit tarjoavat tähän ongelmaan valmiita ratkaisuja. Useassa suunnittelumallissa luokkien välisissä suhteissa käytetään koostamista. Esimerkkinä näistä voidaan mainita Adapter- (Sovitin) ja Recursion- (Rekursio) -suunnittelumallit.

Delegointi (Delegation) tekee koostamisesta yhtä tehokkaan uudelleenkäytön keinon kuin periytymisestä (Gamma et al., 2000). Delegoinnissa pyynnön käsittelyyn osallistuu kaksi oliota: pyynnön vastaanottanut olio delegoi pyynnön käsittelyn edustajalleen (delegate). Ratkaisu on Gamman mukaan analoginen periytymiselle, missä aliluokat siirtävät pyyntöjä yliluokilleen. Kuvassa 16 on esitettynä Gammaa mukaillen esimerkki delegoinnista koostamisen avulla.



Kuva 16. Delegointi (Gamma et al., 2000).

Kuvassa 16 on esitetty kaksi luokkaa: Window ja Rectangle. Sen sijaan, että suunnittelun aikana näiden kahden luokan välinen suhde kuvattaisiin periytymisen avulla, Window-luokassa voidaan uudelleenkäyttää Rectangle-luokan toiminnallisuutta siten, että Window-luokka ylläpitää jäsenmuuttujan kautta viitettä Rectangle-olioon ja delegoi suorakulmio-tyyppisen käyttäytymisen sille. Window-luokan täytyy siirtää pyynnöt eksplisiittisesti Rectangle-oliolleen, sen sijaan, että se olisi periytymisen kautta perinyt suoraan nuo operaatiot. Kuvassa 16 kuvattu kiinteä nuoli osoittaa, että Window-luokka sisältää viittauksen Rectangle-luokan ilmentymään. Toteutuksen kannalta Window-luokkaan sisällytettäisiin Rectangle-luokan olio. Kuva 17 pyrkii havainnollistamaan asiaa.

```
class Window
{
        /*Window-luokan mahdolliset omat tietojäsenet */
        ...
        /* viittaus koostettavan luokan ilmentymään */
        Rectangle  rect;
}
public Window()
{
        rect = new Rectangle(25, 30);
        ...
}
public Rectangle Area()
{
        return rect.Area();
}
```

Kuva 17. Delegoinnin toteuttaminen koostamisen avulla.

Useissa suunnittelumalleissa käytetään delegointia. Esimerkiksi State- (Tila) ja Chain of Responsibility (Vastuuketju) -suunnittelumallit perustuvat delegoinnille. State-suunnittelumallissa olio delegoi pyynnöt senhetkistä tilaansa edustavalle State-oliolle. Chain of Responsibility-malli käsittelee puolestaan pyyntöjä siirtämällä niitä oliolta toiselle pitkin olioiden muodostamaa ketjua. Chain of Responsibility-suunnittelumalli käydään tarkemmin läpi kohdassa 3.4.

Avainasia uudelleenkäytön maksimoinnissa on Gamman mukaan uusien vaatimusten ja nykyvaatimuksiin kohdistuvien muutosten ennakoiminen ja sovelluksen suunnittelu siten, että se voi kehittyä vastaavasti (Gamma et al., 2000). Ohjelmistoon kohdistuvat muutokset voivat merkitä luokkien uudelleensuunnittelua, toteuttamista ja testaamista. Toteutetun ohjelmiston joustamaton rakenne voi pahimmassa tapauksessa tarkoittaa sitä, että ohjelmisto joudutaan suunnittelemaan ja/tai toteuttamaan alusta saakka uudelleen, sen sijaan, että muutokset voitaisiin joustavasti toteuttaa olemassa olevaan ohjelmistoon. Uudelleensuunnittelu vaikuttaa ohjelmistossa useaan kohtaan, joten on selvää, että tällaisessa tilanteessa pienenkin ohjelmiston tapauksessa muutoksesta voi aiheutua mittavat kustannukset. Suunnittelumallien systemaattisella käyttämisellä voidaan uudelleensuunnittelua välttää. Gamman mukaan suunnittelumallien käyttö varmistaa sen, että sovellus voi muuttua määrätyillä tavoilla. Jokainen suunnittelumalli mahdollistaa joidenkin järjestelmärakenteiden muuttamisen toisistaan riippumatta, mikä tekee järjestelmän immuuniksi vastaaville muutoksille.

Gamma mainitsee yleisimmiksi uudelleensuunnitelutarvetta aiheuttaviksi syiksi mm. algoritmiriippuvuuden sekä tiukan sidonnan. Algoritmeja laajennetaan, optimoidaan ja vaihdetaan suunnittelun kuluessa ja ohjelmakoodia uudelleenkäytettäessä. Olioita, jotka ovat riippuvaisia tietystä algoritmista, on muutettava, kun algoritmi muuttuu. Tämän vuoksi todennäköisesti muuttuvat algoritmit tulisi eristää jo suunnittelun aikana.

Iterator (Iteraattori) -suunnittelumallia voidaan esimerkiksi käyttää hyväksi silloin kun tietorakenteiden sisältämiä alkioita pitää käydä läpi eri tavoilla, mutta läpikäytävän tietorakenteen sisäinen rakenne ei saa näkyä ulospäin. Esimerkiksi C++-ohjelmointikielen sisältämä Template Library (STL) -vakiokirjasto tarjoaa viisi iteraattoriryhmää, joiden sisältämillä operaatioilla saadaan tietoja olioista ja tietorakenteista, joihin ne osoittavat. Kun tietorakenteen läpikäyntiin liittyvä algoritmi muuttuu, voi ohjelmoija toteuttaa tarvittavan algoritmin käyttämällä hyväkseen STS:ssä olevia iteraattoriperheitä tai toteuttaa oman spesifisimmän toteutuksen.

Vahvan kytkennän omaavia luokkia on vaikea uudelleenkäyttää yksittäin, koska ne ovat riippuvaisia toisistaan. Luokkien välinen vahva kytkentä johtaa Gamman mukaan nopeasti järjestelmiin, joissa luokkia ei voi muuttaa tai poistaa muuttamatta samalla muitakin luokkia.

Esimerkiksi Mediator (Välittäjä) -suunnittelumalli edistää heikkoa kytkentää estämällä olioita viittaamasta suoraan toisiinsa ja mahdollistaa olioiden välisen vuorovaikutuksen muuttamisen kaikkia yhteyksiä muuttamatta (Gamma et al., 2000).

3. Sisällönhallintasovellus

Kohdassa 3.1 kuvataan sovelluksen toimintaperiaate sekä sen vaatimat rajapinnat. Kohdassa 3.2 tutustutaan Domain Object Factory -suunnittelumalliin, jonka avulla sovelluksen tarvitsema data saadaan tallennettua tietokantaan ja haettua se takaisin tietokannasta sovelluksen ymmärtämään muotoon. Domain Object Factory -mallin kanssa yhteistyössä toimii kohdassa 3.3 esitetty Data Access Object -suunnittelumalli, jonka avulla useat sovellukset voivat käyttää MySQL-tietokantaa hyväkseen. Data Access Object suunnittelumallin avulla tietokantahaut ja lisäykset saadaan suoritettua yhden rajapinnan kautta.. Nämä kaksi suunnittelumallia toimivat yhdessä mahdollistaen joustavan datan käsittelyn. Kohdassa 3.4 käsitellään puolestaan Chain of Responsibility suunnittelumalli, jota käytettiin sovelluksessa WAP-hinnoittelussa käytettävän rajapinnan määrittelyssä.

3.1 Sovelluksen yleiskuvaus[1]

Ina Finland Oy on Tampereella toimiva ja vuonna 1989 perustettu puhelinvaihdejärjestelmän ja telemarkkinointipalveluita, audiopalveluita ja mobiilipalveluita suunnitteleva ja toteuttava yritys**.** Yrityksellä on liiketoimintaa useassa maassa ja sen liiketoiminta on vahvassa kasvussa. Ina Finland Oy:ssä toimii myös pieni tekninen osasto, jonka päävastuulla on uusien järjestelmien suunnitteleminen ja toteuttaminen sekä vanhojen järjestelmien ylläpito. Esimerkkeinä tällaisista järjestelmistä voidaan mainita Helsingin kaupungin liikennelaitokselle suunniteltu ja toteutettu järjestelmä, joka mahdollistaa matkalipun ostamisen tekstiviestillä. Lipuntarkastajilla on matkalipun tarkastamiseen käytössään automatisoitu apuväline, jolla matkalipun validisuus saadaan selvitettyä nopeasti. Puhelinvaihde järjestelmän uusimista voidaan pitää Ina Finland Oy:n historian mittavimpana teknisenä projektina. Projektin on määrä olla valmis vuoden 2008 syksyllä.

Ina Finland Oy:ssä toimiva kuluttajapalvelut-yksikkö suunnittelee ja toteuttaa viihdepalveluja yhteistyössä teknisen osaston kanssa. Tällaisista viihdepalveluista voidaan mainita erilaisten mobiilituotteiden myymisen. Mobiilituotteilla tarkoitetaan esimerkiksi videoita, taustakuvia ja soittoääniä. Loppukäyttäjä voi tilata tuotteen matkapuhelimeensa joko tekstiviestillä tai WAP-latauksella.

---

1 Ina Finland Oy on antanut luvan sisällönhallintasovelluksen osittaiseen käyttöön. Osittainen käyttö sisältää esimerkkejä sovelluksen sisältämästä ohjelmakoodista sekä suunnittelukuvauksista.

Sisällönhallintasovellus (MCM, Mobile Content Management) on syksyllä 2007 Ina Finland Oy:lle toteutettu sovellus, jota käyttämällä voidaan luoda ja hallita mobiilituotteita. Projektin vaatimuksista voidaan tässä yhteydessä mainita tärkeimmät:

1. Järjestelmän siirto Oracle-tietokannasta MySQL-tietokantaan
2. Mobiilituotteiden hallinnointi
3. Tuotteiden julkaiseminen web/wap-sivulla
4. WAP-hinnoittelun uusiminen
5. Tekijänoikeus/lisenssimaksujen raportointi yhteistyökumppaneille

Kuva 18 tarjoaa yleiskuvan sovelluksen toimintaympäristöstä ja rajapinnoista. Kuvassa 18 esitetyn järjestelmän toiminta voidaan kuvata karkeasti seuraavasti: järjestelmän ylläpitäjä lisää uusia mobiilituotteita yrityksen sisäverkossa sijaitsevaan MCM-järjestelmään (1). Tuotteita voidaan hakea myöhemmin tarkasteltaviksi ja tuotteet voidaan liittää myyntikategorioihin sekä erillisiin kategorioihin.

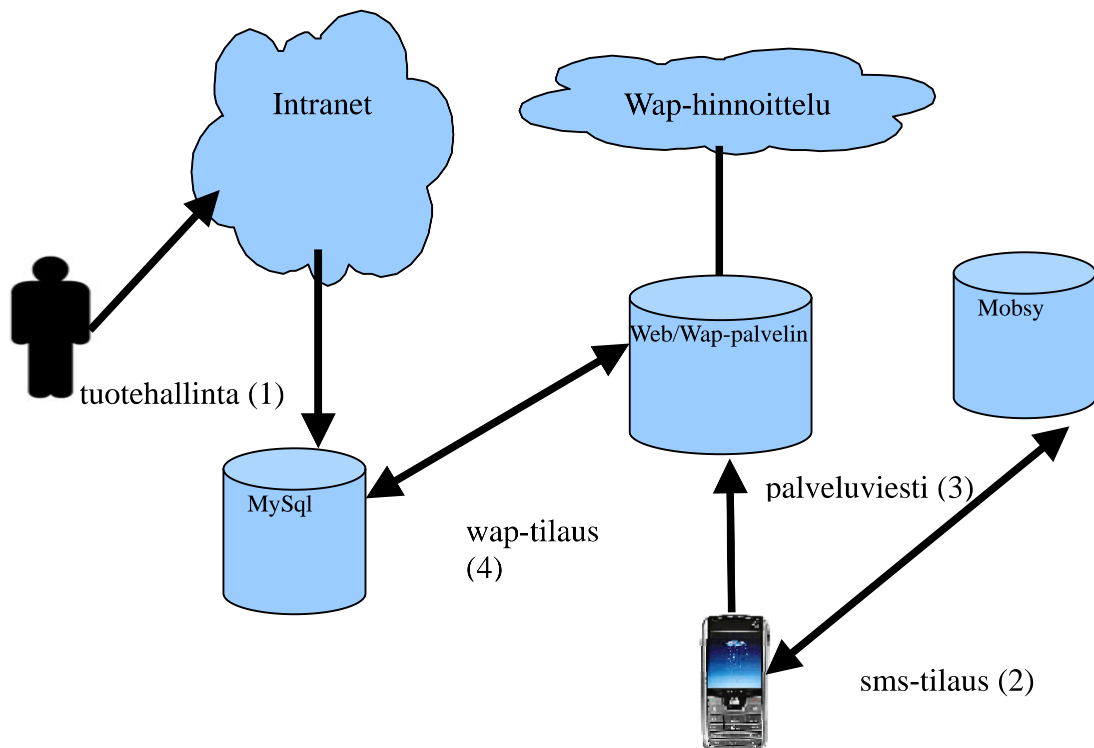
Tuotteet julkaistaan web/wap-palvelimella ja asiakas voi tilata tuotteen joko SMS-viestillä tai WAP-latauksella. SMS-tilauksen yhteydessä asiakas lähettää tekstiviestin Mobsy tekstiviestipalvelimelle (2). Tekstiviestipalvelimella oleva Java-palvelu osaa lähettää asiakkaalle sopivan palveluviestin[2] (service indicator), jonka avulla asiakas ohjautuu oikealle sivustolle ja saa ladattavakseen halutun tuotteen (3). WAP-tilauksessa asiakas siirtyy sivustolle wap-protokollaa käyttäen(4) (Wikipedia, 2008). Tuotteen hinnoittelu WAP-tilauksessa eroaa SMS-tilauksesta siinä, että wap-tilauksessa tuotetieto sekä tuotteen hinta on ilmoitettava jollakin tavalla matkapuhelin operaattorille. Nämä ns. header-tiedot ovat aikaisemmin olleet ”kovakoodattuina” tuotesivuilla. Karkeasti ottaen voidaan sanoa, että aiemmin jokaista myyntikategoriaa (video, taustakuva) kohti on ollut yksi hinnoittelusivusto. Tämä lähestymistapa ei ole ollut varsinkaan ylläpidettävyyden kannalta paras mahdollinen. Uuden MCM-järjestelmän yksi vaatimus oli järjestää tuotteiden WAP-hinnoittelu siten, että hinnoittelu saataisiin dynaamisemmaksi. Tämä tarkoitti suunnitteluvaiheessa tuotetiedon, hinnan sekä operaattoritiedon yhteen nivomista. Tässä tehtävässä käytettiin myöhemmin kohdassa 3.4 esitettyä Chain of Responsibility -suunnittelumallia. Näin saatiin joustava ja yleiskäyttöinen WAP-hinnoittelu rajapinta, jota voidaan käyttää myöhemmin kaikissa sovelluksissa, joissa tarvitaan WAP-hinnoittelua.

MCM-järjestelmän vaatimus oli myös luoda yleiskäyttöinen tietokantarajapinta, jota voitaisiin

---

2 Palveluviesti on tekstiviestipalvelimella sijaitsevan Java-palvelun asiakkaan matkapuhelimeen lähettämä määrämuotoinen viesti, jossa on ilmoitettuna url-muotoinen osoite esim.http://wap.palvelu.fi/sms_content.php?id=1, josta halutun tuotteen lataus onnistuu.

käyttää hyväksi myöhemmissä sovelluksissa, jotka käyttävät MySQL-tietokantaa. Ina Finland Oy:llä on ollut aikaisemmin käytössään oliopohjainen ratkaisu tietokantayhteyksien luomiseen ja tiedon lisäämiseen ja päivittämiseen. Kaikki Oracle-tietokantaa käyttävät sovellukset ovat voineet käyttää hyväkseen tätä rajapintaa. MCM-sovelluksen suunnittelun aikana jouduttiin ratkaisemaan, kuinka sovelluksen tarvitsema data saataisiin tallennettua tietokantaan ja minkälaisen rajapinnan tietokannan tulisi tarjota. Tiedon pysyvyyteen liittyvässä kirjallisuudessa on mainittu useita suunnittelumalleja, jotka tarjoavat oliopohjaisia ratkaisuja tiedon tallentamiseen ja hakemiseen. Tähän liittyy kohdissa 3.2 ja 3.3 kuvatut Domain Object Factory ja Data Access Object -suunnittelumallit.



Kuva 18. MCM-sovelluksen rajapinnat.

MCM-sovelluksen toteutuskieleksi valittiin php, koska toteuttajaksi saatiin riittävän php-kokemuksen omaava henkilö. Php:n on myös osoitettu toimivan erittäin hyvin MySQL -tietokannan kanssa. Kolmanneksi php5 mahdollistaa nykyaikaisten oliopiirteiden hyväksikäytön. Tässä yhteydessä ei ole tarkoitus käydä läpi näitä ominaisuuksia. Väänänen (2005) esimerkiksi on kuvannut kandidaatintutkielmassaan php5-version sisältämiä oliopiirteitä laajemmin.

3.2 Domain Object Factory

Jingwen Cheng määrittelee *tiedon pysyvyyden* (data persistence) olevan konsepti, jossa tieto tallennetaan uudelleenkäytettävässä tilassa tietokonejärjestelmään (Cheng, 1993). Tällaisia tallennusmekanismeja ovat esimerkiksi tiedostot ja tietokannat. Nykyisin yhä enenevissä määrin tietokannan hallintajärjestelmien vastuulla on tiedon tehokas tallentaminen, hakeminen ja muokkaaminen. Mihalis Yannakakis mainitsee tietokanta-alan valtavan kasvun viimeisten 25 vuoden aikana. Yannakakiksen mukaan tietokanta-ala loi 7 miljardin dollarin voitot pelkästään vuonna 1994 ja sen odotetun kasvuvauhdin on estimoitu kasvavan 35% vuodessa (Yannakakis, 1995).

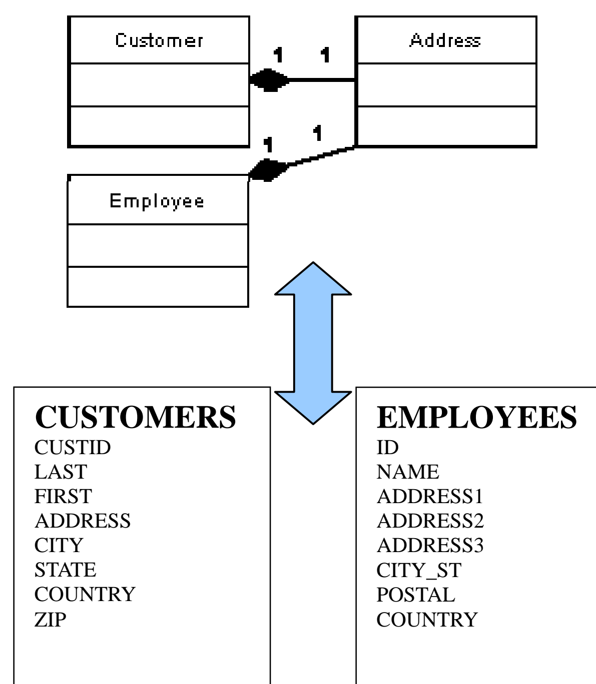
Alalla vallitsevat kasvunäkymät ovat saaneet aikaan kiinnostusta relaatiotietokantojen edelleen kehitykseen ja uusien tietokantajärjestelmien luomiseen ja kehittämiseen. Esimerkkinä tällaisesta suhteellisen uudesta tietokantajärjestelmästä voidaan mainita oliosuuntautuneet tietokannat.

Ohjelmistosuunnittelijat joutuvat ottamaan huomioon tiedon pysyvyyteen liittyvät aspektit jokapäiväisessä työssään. Suunnittelutyössä kohdataan ongelmia, jossa esimerkiksi sovelluksen sisältämä data pitää saada tallennettua tietokantaan, haettua se tehokkaasti ja muunnettava se vielä sovelluksen ymmärtämään muotoon. Suunniteltujen sovellusten tulee toimia tämän lisäksi usein yhdessä useamman tietokantatuotteen kanssa. Käyttöliittymien tulee piilottaa tietokantaan liittyvät operaatiot käyttäjältä ja samalla tukea useiden käyttäjien samanaikaista tiedonsaantia. Tiedonsaantisuunnittelumallit (data access pattern) on kehitetty ratkaisemaan tällaisia tietokantojen yhteydessä useasti toistuvia suunnitteluongelmia. Samalla tavoin kuin luvussa 2 kuvatut suunnittelumallit, tiedonsaanti-suunnittelumallit on kehitetty tarjoamaan ohjelmistosuunnittelijoille abstrakteja kuvauksia tiedonsaantiin liittyvistä suunnitteluongelmista (Nock, 2004).

Tässä yhteydessä näitä suunnittelumalleja ei luetteloida tarkemmin, vaan keskitytään kuvaamaan kahta mallia, joita käytettiin hyväksi sisällönhallinta sovelluksen yhteydessä. Tässä luvussa tarkastellaan lähemmin Domain Object Factory -suunnittelumallia, jolla sovelluksen tarvitsema data saadaan joustavasti siirrettyä tietokannan ja sovelluksen välillä. Tämä siirtäminen sisältää datan tallentamisen relaatiotietokantaan sekä sen muuttamisen tarvittaessa takaisin sovelluksen ymmärtämään muotoon. Kohdassa 3.3 puolestaan tutustutaan Data-Access Object -suunnittelumalliin, jonka avulla saadaan luotua yksi yhtenäinen rajapinta, jonka avulla sovellukset voivat muodostaa tietokantayhteyksiä, tallentaa dataa tai poistaa dataa MySQL-relaatiotietokannasta.

Kuvassa 19 pyritään kuvaamaan sovelluksen tarvitseman datan ja relaatiotietokannan välinen suhde.

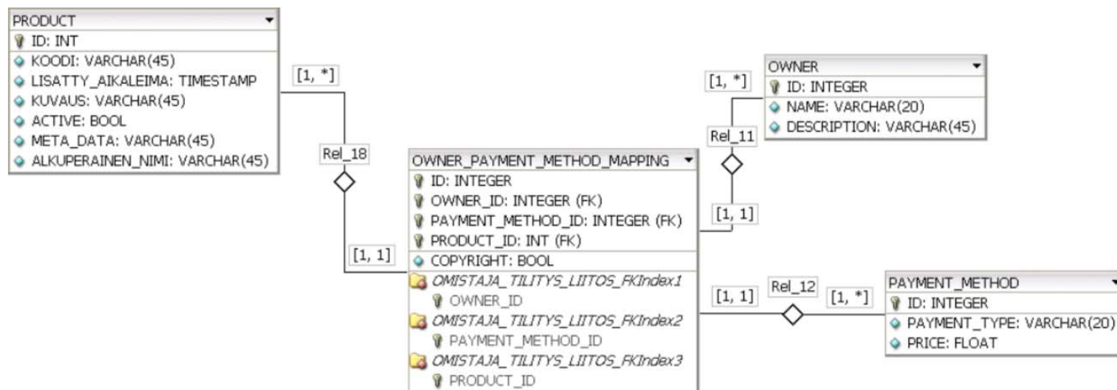
Oliosuunnittelun aikana muodostuneet luokat muunnettiin tietokantamuotoon, jossa karkeasti ottaen jokaista luokkaa vastaa yksi fyysinen tietokannan taulu. Kuvassa 19 luokkien ja tietokantataulujen välinen nuoli kuvaa kaksisuuntaista tiedonmuuntamista. Haettaessa tietoa tietokannasta, yhdellä tulosjoukon rivillä alustetaan Domain Object -olio. Kyseinen Domain Object on yksinkertainen luokka, joka toimii yhteistyössä Domain Object Factory -mallin kanssa. Domain Object sisältää pääsääntöisesti ainoastaan asetus- ja saantimetodeita, joilla olion tilaa pystytään muuttamaan ja joilla vastaavasti saadaan tarvittaessa tietoa olion tilasta.



Kuva 19. Datan muuntaminen tietokantamuotoon (Nock, 2004).

Sisällönhallintasovelluksessa käyttäjä luo web-käyttöliittymän kautta myytäviä tuotteita ja tuotealiaksia, liittää aliaksia erilaisiin kategorioihin sekä liittää aliaksiin tarvittaessa esikatselutiedostoja. Sovelluksen suunnitteluvaiheessa kohdealueelta tunnistettiin itsenäiset kokonaisuudet, niistä muodostettiin tietokantataulut sekä tietokantataulujen välille liitettiin tarvittavat yhteydet. Suunnitteluvaiheessa jokaista tietokantataulua vastasi yksi Domain Object -olio. Mallinnettaessa monta-moneen- yhteyksiä, liitostaulusta ei luotu omaa Domain Object -oliota, sillä luokkien lukumäärä haluttiin pitää suhteellisen pienenä. Liitostaulun sisältämät kentät siirrettiin sellaisen luokan vastuulle, jolle ne suunnittelun ja toteutuksen näkökulmasta parhaiten sopivat. Kuvassa 20 esitetyssä kuvassa pyritään havainnollistamaan

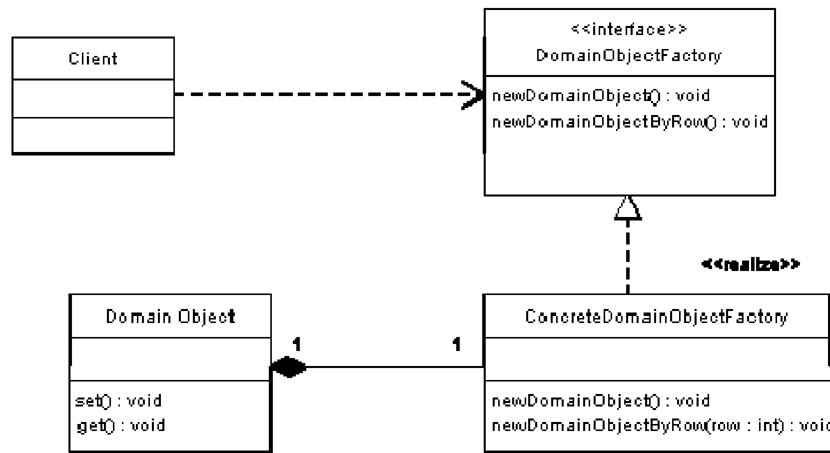
tällaista monta-moneen-yhteyttä.



Kuva 20. Monta-moneen yhteydet.

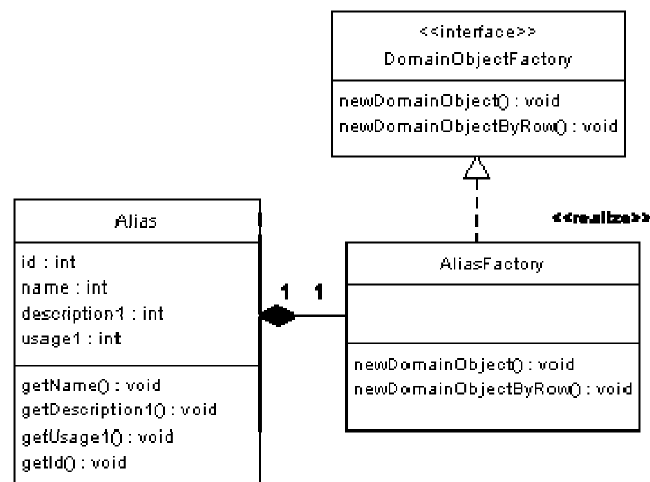
Kuvassa 20 tietokantakuvauksen osassa Product-taulu kuvaa myytävää tuotetta. Owner-taulu kuvaa Ina Finland Oy:n yhteistyökumppania, joka omistaa tuotteen. Ina Finland Oy myy tuotteen ja tilittää jokaisesta SMS/WAP-tilauksesta tietyn summan yhteistyökumppanille. Payment_method-taulu kuvaa tätä tilitystapaa, joka voi olla esimerkiksi soittoäänien yhteydessä kiinteä 0,20 euron suuruinen summa. Oleellista on tässä yhteydessä kuitenkin se, että käyttäjän on kyettävä määrittämään tilityksen suuruus jokaisen tuotteen yhteydessä. Omistaja ja tilitystapa liittyvät läheisesti tuotteeseen, joten on johdonmukaista siirtää liitostaulun kentät Product-luokkan vastuulle. Liitteessä 3 on esitettynä Product-luokan sisältämä ohjelmakoodi kokonaisuudessaan.

Kuvassa 21 on esitetty MCM-sovelluksessa käytetyn Domain Object Factory -suunnittelumallin rakenne Nockia mukaellen. DomainObjectFactory on rajapinta, joka tarjoaa kaksi metodia: *newDomainObject*-metodilla sovelluksen vaatima data saadaan mallinnettua olioiksi ja vastaavasti *newDomainObjectByRow*-metodissa tietokantakyselyn tulosjoukon rivin avulla saadaan luotua tarvittava olio. ConcreteDomainObjectFactory-käyttää olioiden luomiseen Domain Object -oliota. Kuvasta 21 poiketen, Domain Object -luokat sisältävät tietojäseniä ja paikoitellen runsaasti metodeita. Ne on jätetty pois kuvasta tilanpuutteen vuoksi.



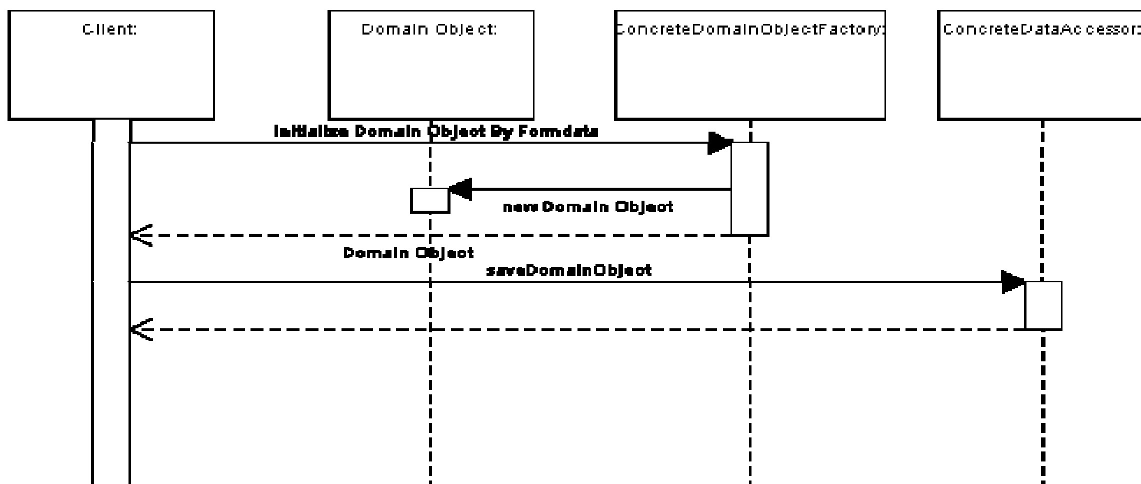
Kuva 21. Domain Object Factory-suunnittelumallin rakenne (Nock, 2004).

Kuvassa 22 on esitetty esimerkki luokkahierarkiasta MCM-sovelluksen näkökulmasta.



Kuva 22. Domain Object Factory-malli MCM-sovelluksen näkökulmasta.

Kuvassa 23 esitetyllä sekvenssikaaviolla pyritään kuvaamaan sisällönhallintasovelluksen sisältämien olioiden välistä vuorovaikutusta. Käyttäjä hallinnoi tuotteita esimerkiksi täyttämällä html-lomakkeella olevaa tietoa. Lomakkeen sisältämä data tarkistetaan ja jos annetut tiedot ovat valideja, tietojen perusteella luodaan tarvittava olio. Olion luonnin suorittavat kulloinkin vastuussa oleva Domain Object Factory-olio yhteistyössä vastaavan Domain Object-olion kanssa. Luotu olio palautetaan sovelluksen kutsuvalle osalle, joka voi tallentaa olion tilan tilapäisesti myöhempää tietokantaan tallentamista varten. Kuvassa 24 on esitetty puolestaan DomainObjectFactory –rajapinnan toteutus ja kuvassa 25 kuinka sisällönhallintasovellus voi kutsua rajapinnan toteuttavaa luokkaa.



Kuva 23. Olioiden välinen vuorovaikutus.

```
<?php
public interface DomainObjectFactory
{
        function newDomainObject();
        function newDomainObjectByRow($row);

}
?>
```

Kuva 24. DomainObjectFactory-mallin tarjoama rajapinta.

```
<?php
...
$o = new AliasFactory();
$_SESSION['alias'] = $o->newDomainObject();

?>
```

Kuva 25. Esimerkki sisällöhallintasovelluksen Client -osasta.

Kuvassa 26 on esitettynä AliasFactory-luokan toteutus. Ohjelmoinnin kannalta huomattava on se, että DomainObjectFactory-rajapinta ja tietokantataulua vastaava alias-luokka sisällytetään toteutettavan luokan käyttöön include_once-lauseella.

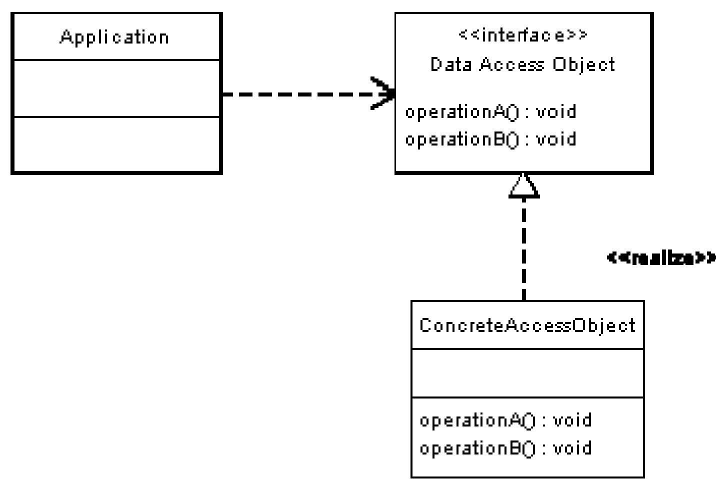
```php
<?php
include_once("domain_object_factory.php");
include_once("alias.php");

class AliasFactory implements DomainObjectFactory
{
        /* initializes new domain object with database table row */
        function newDomainObjectByRow($row)
        {
                $alias = new Alias();
                $alias->setId($row[0]);
                $alias->setName($row[1]);
                $alias->setDescription1($row[2]);
                $alias->setUsage1($row[3]);
                $alias->setDescription2($row[4]);
                $alias->setUsage2($row[5]);
                $alias->setPublTime($row[6]);

                return $alias;
        }
        /* initializes new domain object with valid form information which is
           given by the user */
        function newDomainObject()
        {
                $alias = new Alias();
                $alias->setId(-1);
                $alias->setName($_SESSION['aliasfrm']['name']);
                $alias->setDescription1($_SESSION['aliasfrm']['description1']);
                $alias->setDescription2($_SESSION['aliasfrm']['description2']);
                $alias->setUsage1($_SESSION['aliasfrm']['usage1']);
                $alias->setUsage2($_SESSION['aliasfrm']['usage2']);
                $alias->setPublTime($_SESSION['aliasfrm']['publishingtime']);
                $alias->setProductId(-1);
                $alias->setActive(true);
                $alias->setWebPreviewId(0);
                $alias->setWapPreviewId(0);

                return $alias;
        }
}
?>
```

Kuva 26. AliasFactory-luokan toteutus.

3.3 Data Access Object

Data Access Object -suunnittelumallin avulla fyysiset tietokantaoperaatiot voidaan eristää sovelluksen sisältämästä ohjelmakoodista. Clifton Nockin mukaan Data Access Object -suunnittelumalli auttaa piilottamaan tietokantaan liittyvät fyysiset toimenpiteet sovelluksen

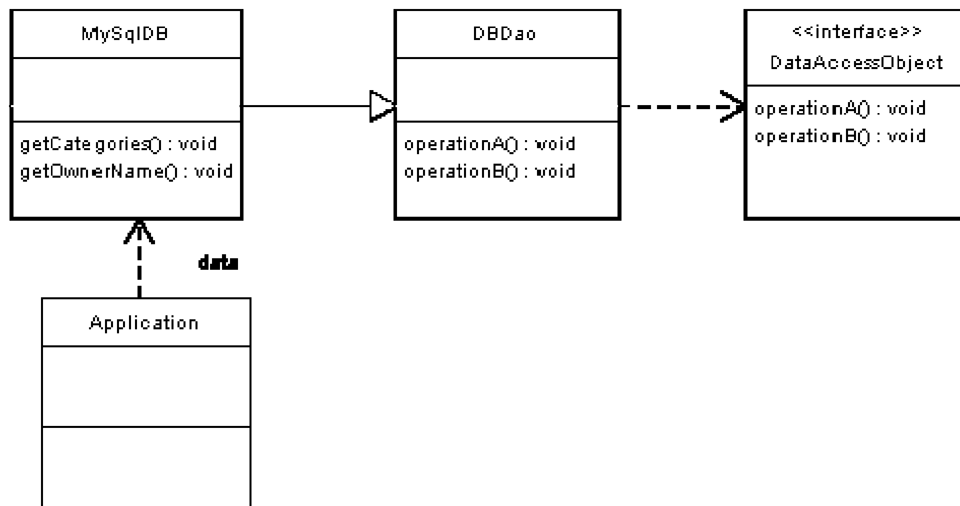
muilta osilta ja näin kannustaa julkistamaan ainoastaan tarvittavat loogiset operaatiot (Nock, 2004). Tietokantaoperaatiolla tarkoitetaan tietokantaan kohdistuvia perustoimintoja kuten: tietokantayhteyksien luomista, tietojen lisäämistä, päivittämistä sekä poistamista. Sovelluksen ja sen sisältämien tietokantaoperaatioiden erottamisella toisistaan pyritään uudelleenkäytettävyyteen ja ylläpidettävyyteen. Muutokset tietokantaan liittyvässä ohjelmakoodissa eivät vaikuta itse sovellukseen ja näin tarvittavat muutokset ovat helpompia suorittaa kuin jos tietokannan fyysiset operaatiot olisivat suoraan käytössä sovelluksessa. Kuva 27 kuvaa Data Access Object -mallin rakennetta Clifton Nockia mukaellen.



Kuva 27. Data Access Object -mallin rakenne (Nock, 2004).

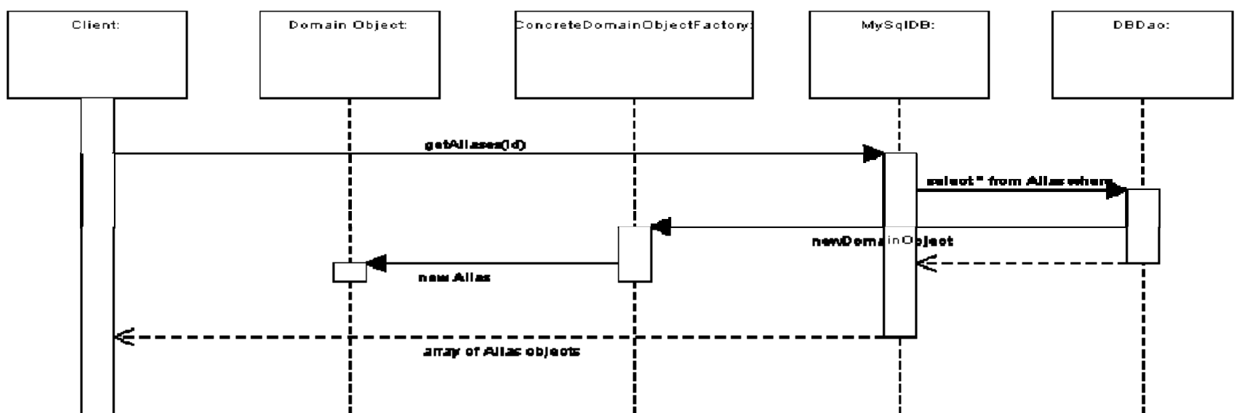
Kuvassa 27 esitetyssä luokkarakenteessa on kuvattuna Data Access Object -suunnittelumallin perusrakenne. Data Access Object on rajapinta, joka tarjoaa toteutettavaksi tietokantaan liittyviä fyysisiä operaatioita. ConcreteAccessObject toteuttaa rajapinnan ja sen kautta fyysiset tietokantaoperaatiot voidaan suorittaa. Kuvassa 27 sovellus (application) käyttää Data Access Object -mallin tarjoamaa rajapintaa suoraan. MCM-sovelluksessa itse sovelluksen ja Data Access Object-mallin väliin toteutettiin luokka, jonka kautta varsinaiset SQL-lauseet muodostetaan ja välitetään Data Access Object -mallille. Kuvassa 28 on esitetty esimerkki kyseisestä luokkahierarkiasta.

Kuvassa 28 esitetty MySqlDB-luokka toimii sovelluksen ja Data Access Object -mallin välissä ja sisältää pelkästään SQL-lauseita, joilla dataa varsinaisesti haetaan tietokannasta tai tallennetaan tietokantaan. Tällä tavoin itse sovelluksen sisältämä ohjelmakoodi on täysin riippumaton siitä, kuinka SQL-lauseet muodostetaan ja kuinka varsinainen data saadaan takaisin tietokannasta sovelluksen ymmärtämään muotoon. Kuvassa 28 esitetyn DataAccessObject-luokan tarjoama rajapinta on liitteessä 5. DBDao-luokka toteuttaa rajapinnan ja sen toteutus on vastaavasti liitteessä 4.



Kuva 28. Data Access Object -malli MCM-sovelluksen näkökulmasta.

Kuvassa 29 pyritään havainnollistamaan asiaa esittämällä olioiden välinen vuorovaikutus.



Kuva 29. Olioiden välinen vuorovaikutus.

Kuvassa 30 on esitetty esimerkin omaisesti osa update_product.php -skriptistä, jossa käyttäjä voi muokata jo tallennettuja tuotteita. Update_product.php saa GET-parametreina muokattavan tuotteen id:n sekä tiedon ollaanko tuotteita muokkaamassa tai lisäämässä. Tuotteisiin liitetyt aliakset saadaan noudettua tietokannasta käyttämällä MySqlDB -luokkaan määriteltyä getAliasData()-metodia. Kyseisen metodin toteutus on esitetty kuvassa 31.

```
<?php
   ...
   include_once("mysqldb.php");
   if($mode == "update")
   {
      $prod_id = $_GET['id'];
      $aliases = array();
     //alias olio/oliot palautetaan
      $aliases = $mysql->getAliasData($prod_id);
      for($i = 0; $i < count($aliases); $i++)
      {
         $_SESSION['aliases'][$i] = $aliases[$i];
       //haetaan aliakselle kuuluvat kategoriat
         $_SESSION['aliases'][$i]->fillCategories($mysql->getCategoriesFromDB(1,$i),1);
       //haetaan aliakselle kuuluvat myyntikategoriat
         $_SESSION['aliases'][$i]->fillCategories($mysql->getCategoriesFromDB(2,$i),2);

       }
    }
    …
    …
 ?>
```

Kuva 30. update_product.php

```
<?php
include_once("mysqldb.php");
function getAliasData($prod_id)
{

       $columns = array("id", "name", "description1", "usage1", "description2",
                        "usage2",   "publ_time");
       $filter = array("product_id = '$prod_id' and active = 1");
       $alias = new Alias();
       return $this->read($alias, $columns, $filter);
}
?>
```

Kuva 31. getAliasData()-metodin toteutus.

3.4 Chain of Responsibility

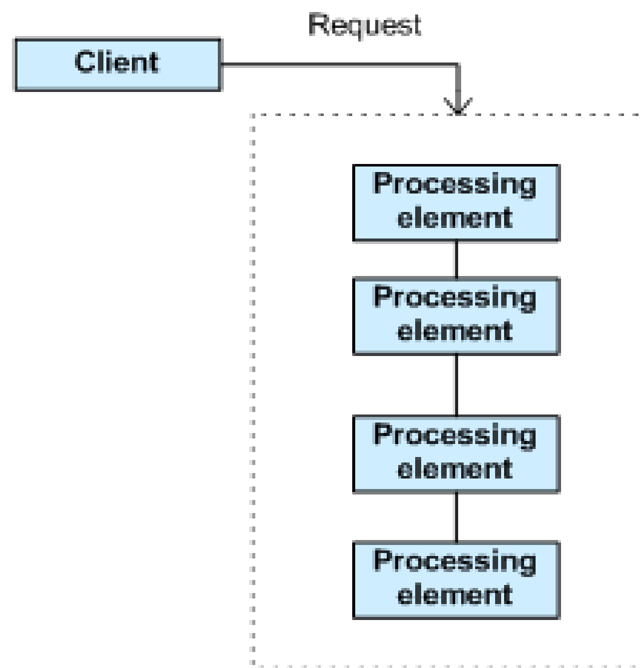
WAP-palveluissa asiakkaan tekemä ostos on pystyttävä laskuttamaan matkapuhelinoperaattorin tarjoaman rajapinnan kautta. SMS-tilauksessa asiakas lähettää tekstiviestin, jonka mukaan tekstiviestipalvelimella oleva palvelu osaa laskuttaa asiakasta palvelimelle valmiiksi konfiguroitujen hinta-asetusten mukaan. WAP-laskutus perustuu suori-

tettuihin transaktioihin. WAP-laskutusparametrit välitetään matkapuhelinoperaattorin WAP-gatewayn ja palveluntarjoajan palvelimen välillä HTTP-otsikkotiedoissa. Ina Finland Oy:n WAP-palveluiden hinnoittelu oli aikaisemmin järjestetty siten, että jokaiselle tuoteryhmälle (video, taustakuva, soittoääni) on useampi laskutustiedosto. Tällä hetkellä Suomessa on neljä suurempaa matkapuhelinoperaattoria. Tämä tarkoittaa WAP-laskutuksen yhteydessä sitä, että jokaiselle tuoteryhmälle on laadittu 4 laskutustiedostoa. Jos tuotteen hinta muuttuu, on muistettava päivittää kaikkia näitä laskutustiedostoja. Otettaessa käyttöön uusi tuoteryhmä, on sille luotava neljä laskutustiedostoa, yksi kutakin matkapuhelin- operaattoria kohti. Tämä lähestymistapa ei ole ollut kovinkaan joustava tai helposti ylläpidettävä. Uuden MCM sovelluksen yhtenä vaatimuksena olikin luoda uusi WAP-laskutusrajapinta, jonka kautta tarvittavat laskutustiedot saataisiin muodostettua ja lähetettyä dynaamisesti. Lisäksi vaatimuksena oli se, että toteutettava WAP-laskutus rajapinta olisi tulevaisuudessa mahdollisimman helposti käyttöönotettavissa kaikissa muissakin WAP-laskutusta vaativissa sovelluksissa.

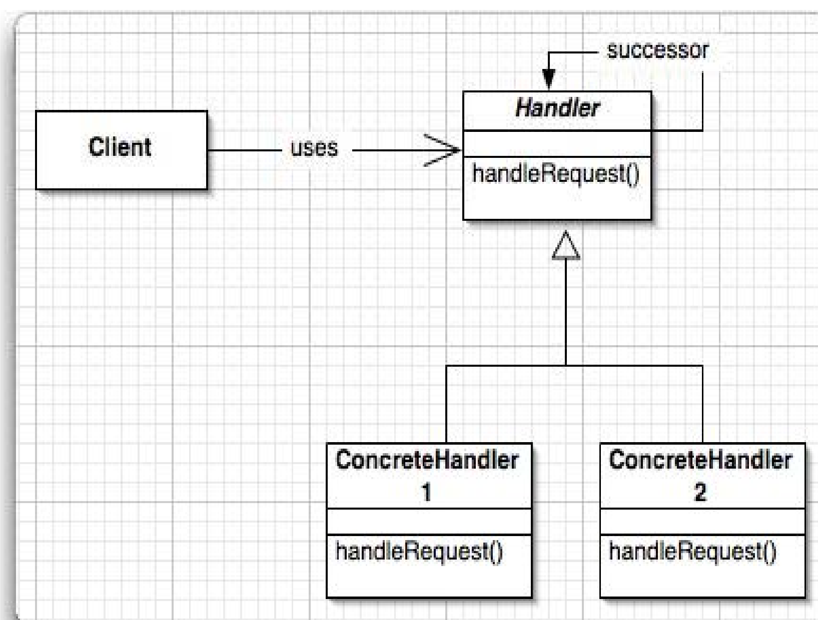
Vastuuketju, Chain of Responsibility -suunnittelumalli (COR) kuuluu Gamman tekemän luokituksen mukaisesti olioiden käyttäytymiseen liittyviin suunnittelumalleihin. COR-suunnittelumallia voidaan käyttää hyväksi pyynnön delegoimisessa joukolle vastaanottajia. Ajonaikana ei välttämättä ole tiedossa vastaanottajien lukumäärä eikä se mikä vastaanottajista lopulta käsittelee pyynnön. Gamma käyttää näistä vastaanottajista nimitystä *implisiittinen vastaanottaja* (implicit receiver). COR -suunnittelumallissa pyynnön vastaanottavista olioista muodostetaan ketju, ja pyyntöä siirretään ketjussa, kunnes joku olioista käsittelee sen. COR -suunnittelumallissa on myös otettava huomioon virhetapauksen mahdollisuus jos mikään vastaanottavista olioista ei pysty käsittelemään pyyntöä (Gamma et al., 2000). Kuva 32 pyrkii selventämään COR -suunnittelumallin ideaa. Gamman mukaan COR -suunnittelumalli soveltuu hyvin käytettäväksi kun

- useampi kuin yksi olio voi käsitellä pyynnön ja käsittelijää ei tiedetä etukäteen *a priori.* Käsittelijän valinnan on siis oltava automaattinen.
- pyyntö halutaan lähettää jollekin useista mahdollisista olioista määrittelemättä vastaanottajaa eksplisiittisesti.
- pyynnön käsittelevien olioiden joukko halutaan määritellä dynaamisesti.

Kuvassa 33 on esitetty COR -suunnittelumallin rakenne Gamman käyttämän luokkakaavionotaation mukaisesti:



Kuva 32. Pyynnön eteneminen vastuuketjussa (SourceMaking, 2008).

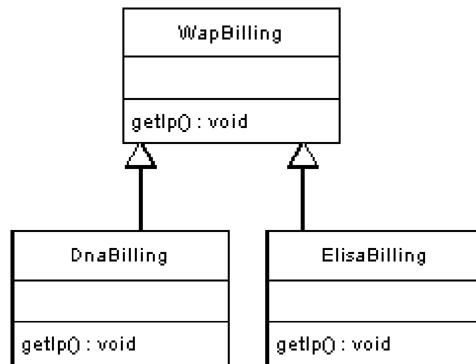


Kuva 33. COR -suunnittelumalliin osallistuvat oliot (Geary, 2003).

- **Handler** määrittelee rajapinnan, jolla pyyntöjä käsitellään. Handler voi myös toteuttaa valinnaisesti seuraaja-linkin.
- **ConcreteHandler** käsittelee ne pyynnöt, joista se itse vastaa. ConcreteHandler pystyy myös osoittamaan seuraajansa ja käsittelemään pyynnön. Muussa tapauksessa se siirtää pyynnön eteenpäin seuraajalleen.

- **Client** käynnistää pyynnön käsittelyn lähettämällä sen ketjussa olevalle ConcreteHandler-oliolle.

Kuvassa 34 on esitetty osa WAP-laskutuksessa käytetystä luokkahierakiasta. Muut operaattori-luokat periytyvät WapBilling-yliluokasta kuvassa 34 esitetyllä tavalla.



Kuva 34. WAP-laskutuksen luokkahierarkia.

WAP-tilaamisen askeleet askeleet ovat seuraavat:

1. Asiakas tulee sivustolle WAP -protokollaa käyttäen.
2. Asiakkaan ip:n mukaan selvitetään operaattori.
3. Operaattori-tieto tallennetaan sessio-muuttujaan.
4. Asiakkaan msisdn (puhelinnumero) otetaan tarvittaessa talteen.
5. Tuotteen maksamisen yhteydessä käytetään sessio-muuttujassa olevaa operaattori-oliota laskutustiedon lähettämisessä.

```
header("x-wap-serviceid: 33616");

header("x-wap-cpid: 131");

header("x-wap-tc: 31”);
```

Kuva 35. Esimerkki Dna:n laskutustiedoista.

Kuvassa 35 on esimerkki laskutustiedoista, jotka välitetään asiakkaan matkapuhelin-operaattorille tuloutusta varten. Tarvittavat laskutustiedot vaihtelevat operaattoreittain. Esimerkiksi Soneran tapauksessa riittää pelkästään yksi laskutustietorivi. Laskutuskentän arvo koostuu palvelun nimestä sekä hintaluokasta. Kuvassa 35 ylin rivi ilmoittaa käytettävän palvelun. Palvelu (service) voidaan ilmoittaa numerona, kuten kuvassa 35 Dna:n tapauksessa, tai merkkijonona, kuten Soneran tapauksessa kuvassa 36.

```
header("x-up-billing-info: SN=MC_TRUE&RC=317);
```

Kuva 36. Soneran WAP-laskutustieto.

Kuvassa 35 alimmalla rivillä kuvattu tc-arvo pitää sisällään tuotteen hinnan. Tuotteen hinta koostuu tc/hinta-pareista ja nämä on listattu operaattorin ja WAP-palvelutarjoajan välisessä palvelusopimuksessa.

4. Yhteenveto

Tässä tutkielmassa on pyritty kuvaamaan ohjelmistotuotannon alaan vaikuttaneita metodologeja 1950-luvulta 2000-luvulle. Ohjelmistotuotantoala on muuttunut tällä aikavälillä huomattavasti. Varmaa ei ole se, onko muutos ollut pelkästään positiivista, vai onko oliosuuntautunut paradigma jarruttanut kehitystä omalla tavallaan. 1970-luvulla havaittiin, että ohjelmistosuunnitteluun liittyvät ongelmat kannatti jakaa pienemmiksi ja samalla hallittavimmiksi osakokonaisuuksiksi. Nämä funktioiksi/proseduureiksi nimetyt osakokonaisuudet käsittelivät syötteitä ja palauttivat tietoa joko suoraan tai välillisesti syötteiden välityksellä. Hyvin nopeasti huomattiin, että funktioista kannattaa tehdä yleiskäyttöisiä, jotta aikaa säästyisi saman ongelman uudelleen ratkomiselta. Tämän toiminnan kautta syntyi pienen mittakaavan uudelleenkäyttöä. Mittakaava suureni, kun samaan toiminnallisuuteen liittyvät funktiot ymmärrettiin koota suuremmiksi kokonaisuuksiksi, moduuleiksi. Näitä moduuleja voitiin käyttää hyväksi jopa järjestelmän osakokonaisuuden uudelleenkäytössä. Oliosuuntautuneen paradigman perustavaa laatua oleva ajatus oli koota ennen irrallaan oleva data ja sen käsittelyrutiinit ja liittää ne olioiden ominaisuuksiksi. Olioiden voitiin käsittää olevan kuin ihmisiä, joilla on ominaisuuksia ja tietynlainen käyttäytyminen.

Olioparadigman ympärille muodostui nopeasti kukoistavaa yritystoimintaa, jossa kehitettiin kaupallisia ohjelmointikieliä, suunnittelumenetelmiä ja mallinnuskieliä. Oliosuunnittelussa kehitettiin uusia menetelmiä, joiden avulla entisestään monimutkaistuneita ohjelmistoja pyrittiin kehittämään entistä tuottavammin. Suunnittelumallit pyrkivät tuomaan apua kriittiseen suunnitteluvaiheeseen tarjoamalla testattuja, dokumentoituja ja toteutuskieliriippumattomia ratkaisuja olioiden rakenteesta ja niiden välisistä suhteista. Varsinkin Erich Gamman panostus suunnittelumallien luokittelussa on ollut merkittävä. Gamma on dokumentoinut luokittelemansa suunnittelumallit johdonmukaisesti. UML-notaation mukaiset rakennekuvaukset selvittävät kokeneelle ohjelmistosuunnittelijalle nopeasti mihin tarkoitukseen mallia voidaan käyttää, mutta mallien dokumentointi auttaa myös kokemattomampaa suunnittelijaa. Tulevaisuudessa haastetta riittääkin siinä, että valtavasta suunnittelumalleihin liittyvästä tiedosta osaa omaksua omaan tarkoitukseensa parhaan tiedon.

Viitteet

Bishop J. (2008) Language Features Meet Design Patterns: Raising the Abstraction Bar. *Proceedings of the 2nd international workshop on The role of abstraction in software engineering.*

Boehm, B (2006) A view of 20th and 21st Century Software Engineering. *Proceedings of the 28th international conference on Software engineering*.

F. P. Brooks. (1986) No silver bullet – essence and accidents of software engineering. *IEEE Computer*, **20**(4):10–19.

Capretz, L.F. (2003) A Brief History of the Object-Oriented Approach. *ACM SIGSOFT Software Engineering Notes*, **28**(2).

Carlson D. (2006) *Software Design Using C++*. Computing & Information Science Department, Saint Vicent College. http://cis.stvincent.edu/html/tutorials/swd/records/records.html (25.5.2008).

Cheng J. (1993) Improving the software reusability in object-oriented programming. *ACM SIGSOFT Software Engineering Notes*, 18(4).

Davis A.M., Sitaram P. (1994) A concurrent process model of software development. *ACM SIGSOFT Software Engineering Notes* **19**(2).

Derr K.W (1995) Applying OMT: a practical step-by-step guide to using the object modeling technique. SIGS Publications Inc., New York.

Eden A., Hirshfeld Y. (2001) Principles in formal specification of object oriented design and architecture. *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research* (3).

Edgerton S., Hulse C., Ubnoske M., Vazquez L. (1999) Reducing maintenance costs through the application of modern software architecture principles. *ACM SIGAda Ada Letters*, **XIX** (3).

Fincher M. (2006) *Updates on Software Engineering* http://www.fincher.org/tips/General/SoftwareEngineering/SoftwareEngineering2006.shtml (3.06.2008).

Fowler M. (1997) *Analysis Patterns: Reusable Object Models*. Addison-Wesley., Massachusetts.

Frakes W., Terry C. (1996) Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, **28**(2).

Fraser S., Booch G., Buschmann F., Coplien J., Kerth N., Jacobson I., Rosson M.B. (1995). *Patterns (Panel): Cult to Culture?.* Conference on Object Oriented Programming Systems Languages and Applications (85-88).

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

Geary D. (2003) *Follow the Chain of Responsibility* http://www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns.html (15.5.2008).

Graphical Development Process Assistant (GDPA) (2002) *Waterfall* http://www.informatik.uni-bremen.de/gdpa/def/def_w/WATERFALL.htm.

Java Persistent Objects (JPOX) (2008) http://www.jpox.org/docs/1_1/inheritance.html (14.5.2008).

The Java Tutorials (2008) http://java.sun.com/docs/books/tutorial/java/concepts/object.html (6.6.2008).

Java World (2008a) http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html (14.5.2008).

Java World (2008b) http://www.javaworld.com (14.5.2008).

Jordan K.M. (1997) *Software Reuse Term Paper* http://www.baz.com/kjordan/swse625/htm/tp-kj.htm#What (12.4.2008).

Korson T., McGregor J.D. (1990) Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, **33**(9).

Krueger C.W. (1992) Software Reuse. *ACM Computing Surveys (CSUR),* **24**(2).

Learning&Development (2008) *System Design Overview* http://elearning.tvm.tcs.co.in/SDO/SDO/3_1.htm (8.6.2008).

McCormack J., Conway D. (2005) *The Software Development Process.* http://www.csse.monash.edu.au/~jonmc/CSE2305/Topics/07.13.SWEng1/html/text.html (4.5.2008).

Messerschmitt D.G., Szyperski C. (2003) *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press., Massachusetts.

Nock C. (2004) *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Pearson Education Inc., Massachusetts.

Pillai K. (1996) The Fountain Model And Its Impact On Project Schedule. *ACM SIGSOFT Software Engineering Notes*, **21**(2).

Polo M., Piattini M., Ruiz F., Calero C. (1999) Roles in the maintenance process. *ACM SIGSOFT Software Engineering Notes*, **24**(4).

Ramachandran M. (2005) Software Reuse Guidelines. *ACM SIGSOFT Software Engineering Notes*, **30**(3).

Source Making http://sourcemaking.com/design_patterns/chain_of_responsibility (15.5.2008).

Sullivan K.J., Griswold W.G., Ben Hallen Y.C. (2001) *The structure and value of modularity in software design.* Proceedings of the 8th European software engineering conference.

Object Management Group (OMG) (2008) *Unified Modeling Language* http://www.uml.org/ (8.6.2008).

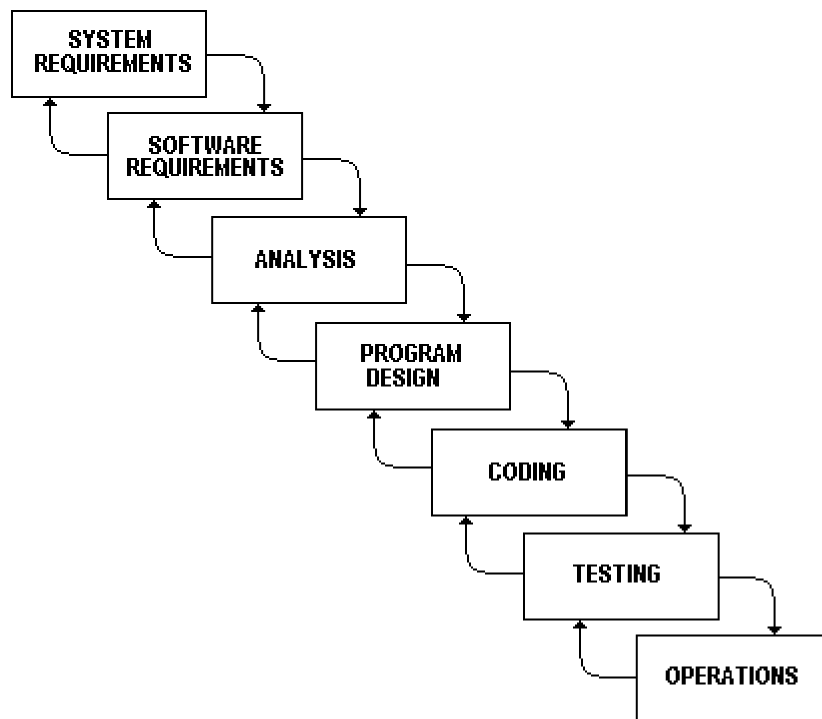
Väänänen T.A (2005) *Php:n oliopiirteet*. Kandidaatin tutkielma, Tietojenkäsittelytieteen laitos, Joensuun yliopisto.

Wikipedia (2008) Wireless Application Protocol http://en.wikipedia.org/wiki/Wireless_Application_Protocol (3.4.2008).

Yannakakis M. (1995) Perspectives on database theory. *ACM SIGACT News*, **27**(3).

Yourdon E. (2008) http://www.yourdon.com/strucanalysis/wiki/index.php?title=Chapter_9 (25.5.2008).

Liite 1. Roycen ohjelmistojen elinkaarimalli



Lähde: (GDPA, 2002).

Liite 2. Esimerkki olioiden koostamisesta

```
// CarDemo.java
class Engine
{
   private String type;
   Engine (String type)
   {
      this.type = type;
   }
   public String getType () { return type; }
}

class Car
{
   private String make;
   private String model;
   private String manufacturer;
   Car (String make, String model, String manufacturer)
   {
      this.make = make;
      this.model = model;
      this.manufacturer = manufacturer;
   }
   public String getMake () { return make; }
   public String getModel () { return model; }
   public String getManufacturer () { return manufacturer; }
}


class CarDemo
{
   public static void main (String [] args)
   {
      Engine e1 = new Engine ("3.8L V6");
      Car c1 = new Car ("Mustang", "Convertible", "Ford");
      Engine e2 = new Engine ("4.6L V8");
      Car c2 = new Car ("Mustang", "GT Coupe", "Ford");
      System.out.println (c1.getManufacturer () + " " +
                          c1.getMake () + " " +
                          c1.getModel () + " " + e1.getType ());
      System.out.println (c2.getManufacturer () + " " +
                          c2.getMake () + " " +
                          c2.getModel () + " " + e2.getType ());
   }
}
```

Lähde: (Java World, 2008b).

Liite 3. Product.php

```
<?php

class Product
{
   private $id;
   private $code;
   private $original_name;
   private $info;
   private $active;
   private $added_time;
   private $owner_id;
   private $payment_method_id;
   private $payment_type;
   private $payment;
   private $copyright;
   //private $salescategory;
   private $tblname = "PRODUCT";

   function __construct(){}

   function getId()
   {
          return $this->id;
   }



   function getCode()
   {
          return $this->code;
   }

   function getOriginalName()
   {
          return $this->original_name;

   }
   function getInfo()
   {
          return $this->info;

   }
   function isActive()
   {
          return $this->active;

   }
   function getAddedTime()
   {
          return $this->added_time;

   }

   function getOwnerId()
   {
          return $this->owner_id;

   }

   function getPaymentMethodId()
   {
          return $this->payment_method_id;

   }

   function getPaymentType()
   {
         return $this->payment_type;
   }

   function getPayment()
   {
         return $this->payment;

   }
```

```
function getCopyright()
{
        return $this->copyright;

}

/*function getSalesCategory()
{
        return $this->salescategory;

}*/


function getTableName()
{
        return $this->tblname;

}


function getColumns()
{
    $colrow = array("code","original_name", "added_time", "info", "active");
    return $colrow;
}

function setId($_id)
{
        $this->id = $_id;
}



function getValues()
{
    $values = array();
    $values[] = $this->code;
    $values[] = $this->original_name;
    $values[] = $this->added_time;
    $values[] = $this->info;
    $values[] = $this->active;

    /*$values[] = $this->owner_id;
    $values[] = $this->payment_method_id;*/

    return $values;

}

function setCode($_code)
{
        $this->code = $_code;
}

function setOriginalName($_name)
{
        $this->original_name = $_name;
}

function setInfo($_info)
{
        $this->info = $_info;

}
function setActive($_active)
{
        $this->active = $_active;

}

function setAddedTime($_time)
{
        $this->added_time = $_time;

}
function setOwnerId($_owner_id)
{
        $this->owner_id = $_owner_id;

}
```

```
    function setPaymentMethodId($_payment_id)
    {
            $this->payment_method_id = $_payment_id;

    }

    function setCopyright($_copyright)
    {
            $this->copyright = $_copyright;

    }

    function setPaymentType($_payment_type)
    {
           $this->payment_type = $_payment_type;
    }



    function setPayment($_payment)
    {
           $this->payment = $_payment;

    }

}

?>
```

Liite 4. DBDao.php

```
<?php
include_once("salescategory.php");
include_once("data_accessor.php");
include_once("dom_factory.php");
include_once("owner_factory.php");
include_once("size_factory.php");
include_once("salescategory_factory.php");
include_once("category_factory.php");
include_once("product_factory.php");
include_once("alias_factory.php");
include_once("blob_file_factory.php");

class DBDao implements DataAccessObject
{

    var $host;
    var $database;
    var $user;
    var $password;
    var $reshandler;
    var $dbConnectionData;


    function DBDao()
    {
        // täällä yhteyksien luominen tietokanta schemoihin

        $this->dbConnectionData = array (...)
    }

    function updateDB($sql)
    {
       $res = mysql_query($sql);
        if(!$res)
        {
            print "Update failed";
            print mysql_error();
            exit;
        }

    }

    function fsetDBconnection($conn)
    {
       foreach($this->dbConnectionData as $DBConnection)
       {

           if($conn == $DBConnection['CONNECTIONNAME'])
           {
               $this->host = $DBConnection['HOST'];
               //print "Host: ".$this->host;
                $this->user = $DBConnection['USERNAME'];
                //print "User: ".$this->user;
                $this->password = $DBConnection['PASSWORD'];
                //print "Pass: ".$this->password;
                $this->database = $DBConnection['DATABASE'];
                //print "Database: ".$this->database;
                $this->reshandler = false;
            }

        }
       return $this->fConnect2DB();
    }
```

```
function fConnect2DB()
{

    if(!$this->reshandler)
    {
        $this->reshandler = mysql_connect($this->host, $this->user, $this->password);
         if(!$this->reshandler)
             print "Opening database connection failed".mysql_error();
         else
         {
            if(!mysql_select_db($this->database, $this->reshandler))
                print "Selecting appropriate database table failed";
         }

    }

}

function fsetDBConn($_host, $_user, $_pass, $_database)
{
   $this->host = $_host;
    $this->user = $_user;
   $this->password = $_pass;
   $this->database = $_database;

   return $this->fConnect2DB();
}

function getColumnsToSelect($columns)
{
    for($i = 0; $i < count($columns); $i++)
    {
        if($i > 0)
            $retstr .= ",".$columns[$i];
        else
            $retstr = $columns[$i];
    }

    return $retstr;
}

function getDBFieldData($sql, $flag=true)
{
   $res = mysql_query($sql);
    if(!$res)
    {
        print "Query failed".mysql_error();
        exit;
    }
    if(!$flag)
    {
       if(mysql_num_rows($res) == 0)
          return 0;
        else
          return 1;

    }
    else
    {
        while($row = mysql_fetch_row($res))
           return $row[0];
   }
}
```

```
function getDBRowData($sql, $type=null)
{
    $res = mysql_query($sql);
     if(!$res)
     {
         print "Query failed";
         exit;
     }
     if($type == "MYSQL_ASSOC")
     {
         while($row = mysql_fetch_array($res, MYSQL_ASSOC))
             return $row;
    }
    else
    {
         while($row = mysql_fetch_array($res, MYSQL_NUM))
             return $row;
   }
}

function getDBArrayData($sql, $result_type=null)
{

      $res = mysql_query($sql);
      if(!$res)
      {
          print "Query failed";
          print mysql_error();
      }
     else
     {
          $arr = array();
          if($result_type == "MYSQL_ASSOC")
          {
              while($row_array = mysql_fetch_array($res, MYSQL_ASSOC))
              {
                  $arr[] = $row_array[0];
              }

          }
          else
          {
              while($row_array = mysql_fetch_array($res, MYSQL_NUM))
              {
                  $arr[] = $row_array[0];
              }
          }
          return $arr;
      }

}



function insert2DB($sql)
{
        $res = mysql_query($sql);
    if(!$res)
    {
        print "Insertion failed";
        print mysql_error();
        return 0;
    }
    return 1;

}
```

```
    function getDBObjectData($sql, $flag = true, $o = null)
    {
       //print "Sql: ".$sql;
       $result = mysql_query($sql);
       if(!$result)
       {
           print mysql_error();
           exit();
       }
       if(mysql_num_rows($result) == 0)
           return null;
       if(mysql_num_rows($result) == 1)
       {
               if(!$flag)
                   return 1;
               else
               {
                       $row = mysql_fetch_row($result);
                return $o->newDomainObjectByRow($row);
           }
       }
       else
       {
               if(!$flag)
                   return 0;
               else
               {
                while($row = mysql_fetch_array($result))
                {
                    $objects[] = $o->newDomainObjectByRow($row);
                }
                return $objects;           //returns an array filled with objects
           }
       }
    }

    function read($table, $columns, $filter_columns="", $obj = null, $sorting_str="")
    {
       //$objects = array();
       $sql_str = "SELECT ";
       if(count($columns) > 0)
       {
           $str = $this->getColumnsToSelect($columns);
           $sql_str .= $str;
       }
       else
           $sql_str .= "* ";
       if(!is_array($table))
           $sql_str .= " FROM ".$table;
       else
       {
          $sql_str .= " FROM ";
           for($i = 0; $i < count($table); $i++)
           {
              $sql_str .= $table[$i];
              if($i < count($table)-1)
                  $sql_str .= ",";

           }

       }
       if($filter_columns != "")
       {
          $sql_str .= " where ";

          $sql_str .= $filter_columns;
       }
       if($sorting_str != "")
           $sql_str .= $sorting_str;
       $sql_str .= ";";

       return $this->getDBObjectData($sql_str, 1, $obj);
   }


?>
```

Liite 5. DataAccessObject-rajapinta

```
public interface DataAccessObject
{
     function updateDB($sql);
     function fsetDBconnection($conn);
     function fConnect2DB();
     function fsetDBConn($_host, $_user, $_pass, $_database);
     function getColumnsToSelect($columns);
     function getDBFieldData($sql, $flag=true);
     function getDBArrayData($sql, $result_type=null);
     function getDBRowData($sql, $type=null);
     function read($table, $columns, $filter_columns="", $obj = null, $sorting_str="");
     function insert2DB($sql);
 }
```