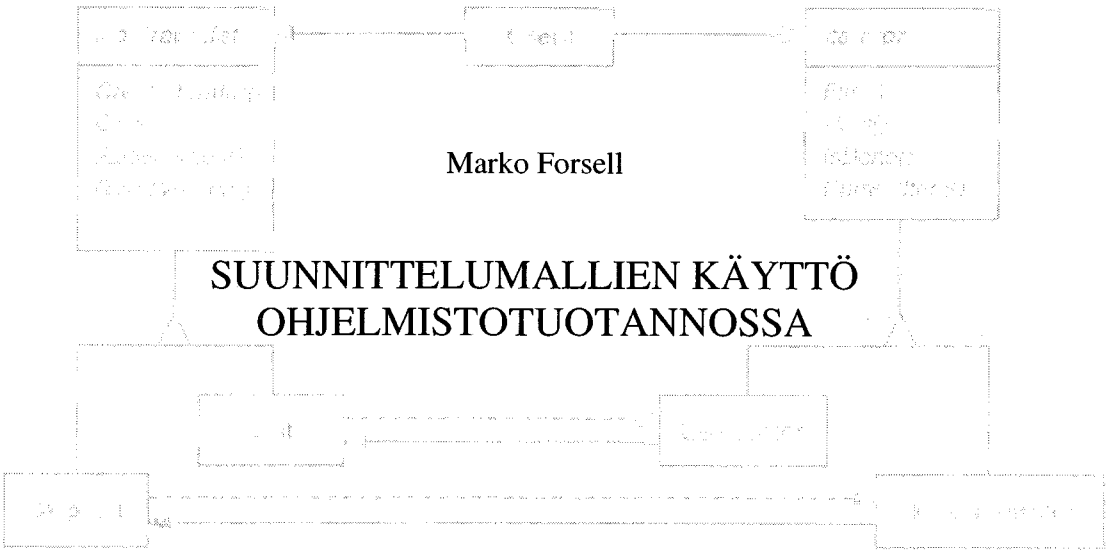
1142



Marko Forsell

SUUNNITTELUMALLIEN KÄYTTÖ OHJELMISTOTUOTANNOSSA

Tietojärjestelmätieteen
pro gradu -tutkielma
26.10.1998

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Forsell, Marko

Suunnittelumallien käyttö ohjelmistotuotannossa/Marko Forsell.

Jyväskylä: Jyväskylän yliopisto, 1998.

98 s.

Tutkielma

Suunnittelumalleja käytetään ohjelmistotuotannossa laadukkaampien ja joustavampien ohjelmistojen tuottamiseksi. Dokumentoituihin suunnittelumalleihin on kerätty alan eksperttien tietotaito, jota kokemattomammat suunnittelijat voivat käyttää. Suunnittelumallit tarjoavat nimettyjä ratkaisuja toistuviin ongelmiin tietyssä kontekstissa.

Tutkielmassa tarkastellaan tieteellisten julkaisujen avulla oliopohjaisten suunnittelumallien luokittelua ja käyttöä sekä niiden tarjoamaa tukea oppimiselle. Suunnittelumallien käyttöä varten arvioidaan ja vertaillaan luokituksia sekä valitaan ohjelmistotuotantoon sopiva luokittelu, joka auttaa käyttäjäänsä valitsemaan oikean suunnittelumallin oikeaan tarkoitukseen. Suunnittelumallien käyttöä katsotaan uudelleenkäytön näkökulmasta, samalla kuvataan raportoituja kokemuksia suunnittelumallien käytöstä todellisissa projekteissa.

Tutkielmassa osoitetaan kognitiivisen tieteen skeema- ja skriptiteorioiden avulla, että ekspertit voivat tallentaa tietämystään myöhempiä tilanteita varten suunnittelumalleilla; suunnittelumallit ovat valmiita ja dokumentoituja ratkaisuja ongelmiin; suunnittelija voi kokeilla kilpailevia suunnittelumalleja tiettyyn ongelmaan; ja suunnittelumallien avulla voidaan dokumentoida eksperttien tietämystä ja siirtää sitä noviisien käyttöön.

Tutkielman lopuksi esitetään olio-oppimisen jako viiteen tasoon. Kullekin tasolle asetetaan oliokeskeisyyden käsitteitä. Tutkielmassa osoitetaan, kuinka suunnittelumallien avulla voidaan siirtyä nopeammin oppimistasolta seuraavalle.

AVAINSANAT: olio, oliokeskeisyys, suunnittelumallit, olio-oppiminen

SISÄLLYS

1 JOHDANTO

Ohjelmistotuotannossa ei ole oikeastaan tapahtunut mitään uutta 30 vuoteen. Edelleen ohjelmistotuotanto on kriisissä. Edelleen keksitään joka vuosi uusi muotitermi. Edelleen eletään työvoimapulassa. Edelleen ammattitaidottomia ihmisiä on ohjelmistonkehityksessä. Laitteistojen suorituskyvyn ja hinnan suhde noudattaa edelleen Mooren lakia. Paljon tehdään työtä, jotta saadaan ratkaistua ohjelmistotuotannon peruskysymys: kuinka tuottaa parempia ohjelmia nopeammin ja tehokkaammin?

Ohjelmistotuotannossa keksitään pyörä uudelleen joka ohjelmistotuotantoprojektissa. Samoja komponentteja, samoja periaatteita ja samoja käsitteitä rakennetaan yhä uudelleen. Kokeneet suunnittelijat kehittyvätkin ajan kanssa. Heille kertyy tietoja ja taitoja ratkaistuista ongelmista, joka näin jää heille käyttöpääomaksi seuraavaa projektia varten. Jokaisen uuden projektin myötä kokemus lisääntyy ja löytyy uusia käyttökelpoisia suunnitelmia ja ratkaisuja.

Kokemattomat ohjelmistokehittäjät toistavat usein samat virheet kuin kokeneetkin. Tälläkin hetkellä joku ohjelmoi lineaarista etsintää, ja senkin väärin. Suunnittelumallit ovat yksi tapa tallentaa kokeneiden suunnittelijoiden tietoa ohjelmistotuotannosta. Suunnittelumalli on ratkaisu ohjelmistotuotannossa toistuvasti esiintyvään ongelmaan. Suunnittelumallit auttavat kehittäjiä jakamaan kokemustaan ensinnäkin välittämällä tietoa onnistuneista ratkaisuista, toiseksi kertomalla uuden paradigman tai lähestymistavan omaksumisesta ja ominaisuuksista sekä kolmanneksi kertomalla epäonnistuneista ratkaisuista ja sudenkuopista, jotka normaalisti jokainen ohjelmistokehittäjä kokee itse (Coplien & Schmidt, 1995).

Suunnittelumallit voidaan ymmärtää paremmin käyttämällä analogiaa näytelmän kirjoittamiseen. Näytelmäkirjailijat harvoin keksivät juonen alusta lähtien. Aluksi kirjailijalla on mielessä perusjuoni kuten 'traaginen sankari' (Macbeth, Hamlet, ym.) tai 'romanttinen novelli' (mm. Harlekiinisarja). Samalla tavoin suunnittelija voi käyttää hyväkseen suunnittelumalleja: 'luodaan käyttöliittymä MVC-mallilla' taikka 'esitetään olion tilat State-mallilla'. (Gamma ym. 1995.)

Suunnittelumallit ovat hyödyllisiä dokumentoitaessa ohjelmiston arkkitehtuuri- ja suunnitteluratkaisuja. Nämä ratkaisut eivät suoraan käy toteutuksen kopiointiin, mutta niillä on vaikutusta ohjelmistotuotantoon (Schmidt 1995, Beck ym. 1996). Dokumentoituja

suunnittelumalleja on olemassa jo useita satoja. Jo yksistään viidessä tunnetuimmassa teoksessa (Gamma ym. 1995, Buschmann ym. 1997, Coplien & Schmidt, 1995, Vlissides ym. 1995, Martin ym. 1997) esitellään yli kaksisataa erilaista suunnittelumallia. Nämä suunnittelumallit tunnistavat, dokumentoivat ja luokittelevat onnistuneita ratkaisuja usein esiintyviin ohjelmistotuotannon ongelmiin.

Suunnittelumallit eivät ole ohjelmointitemppuja, joiden avulla voi näytellä ammattitaitoista hakkeria. Suunnittelumallit voidaan tehdä tavallisilla ohjelmointitavoilla, mutta ne vaativat hieman enemmän ajattelua ja perehtymistä kuin suoraviivaisemmat ratkaisut. Mallien tarkoituksena on tehdä ohjelmistosta joustava ja uudelleenkäytettävä. Kun suunnittelumallin sisältämä ajatus ymmärretään, pääsee ilmoille ajatus 'ahaa' taikka 'hahaa', eikä pelkästään 'huh'. Tällaisen kokemuksen jälkeen oliolähestymistapaan ei enää suhtaudu samalla tavalla. Vaikka lopulta kävisi niin, ettei suunnittelumalleista ole hyötyä, ja vaikka lopulta kävisi niin, etteivät suunnittelumallit paranna ohjelmistoa, kuitenkin tuo keksimisen tunne ja ilahdus antaa suunnittelumalleille oikeuden olemassa oloon.

Ongelmana tällä hetkellä on, että suunnittelumallien tutkimisessa keskitytään uusien sovellutusalojen löytymiseen ja uusien suunnittelumallien tekemiseen. Ensinnäkin kunnollista arviointia ei ole tehty erilaisille suunnittelumallien luokittelutavoille, eikä näin ollen tiedetä, mikä luokittelu sopii ohjelmistotuotannon tarpeisiin. Toiseksi ei ole tarpeeksi saatavissa tietoa, kuinka käyttää suunnittelumalleja ohjelmistotuotannossa. Kolmanneksi ei esitetä, missä järjestyksessä suunnittelumalleja kannattaa opetella.

Tässä tutkielmassa keskitytään suunnittelumalleihin, ja sillä on kolme keskeistä tavoitetta. Ensinnäkin tutkielman tavoitteena on selventää ja helpottaa suunnittelumallien käyttöä. Tämä tehdään antamalla suunnittelumallien käyttöön kolme erilaista näkökulmaa: 1) esittää suunnittelumallien käyttöä helpottava luokittelu ohjelmistotuotannossa, 2) esittää, kuinka suunnittelumalleja käytetään ohjelmistotuotannossa ja 3) kuinka käyttää suunnittelumalleja oliokeskeisyyden oppimiseen.

Toinen keskeinen tavoite on antaa lukijalle mahdollisuus ymmärtää ne keskeiset periaatteet, joille oliokeskeiset suunnittelumallit perustuvat. Varsinkin kolmannen ja neljännen luvun tarkoitus on selkeyttää suunnittelumallien yhteydessä käytettävää termistöä.

Kolmas ja tärkein tavoite on antaa lukijalle mahdollisuus käyttää suunnittelumalleja omaksi hyödykseen. Viidennessä luvussa esitetään, kuinka suunnittelumallien avulla voidaan parantaa omaa oliosuunnittelutaitoa. Tässä annetaan mahdollisuus niin opiske-

lijalle kuin opettajallekin nähdä, miten käyttää suunnittelumalleja hyväkseen tavoitellessaan eksperttiyttä oliokeskeisessä ohjelmistotuotannossa.

Suunnittelumallit ovat vielä varsin uusi ja tuore tutkimuksen kohde. Suunnittelumalliajatusta sovelletaan jatkuvasti uusiin kohteisiin. Tällaisia ovat mm. analyysimallit (Fowler 1997), prosessimallit (mm. Coplien 1995, Foote & Opdyke 1995, Kerth 1995, Cockburn 1996), organisaatiomallit (mm. Harrison 1996, Coplien 1998) ja koulutusmallit (mm. Anthony 1996, Coram 1996). Tässä tutkielmassa näkökulma rajataan koskemaan ainoastaan oliokeskeisen ohjelmistotuotannon suunnittelumalleja.

Suunnittelumallien uutuudesta johtuen myöskään sanastoa ei ole ehtinyt syntyä suomen kielellä. Suunnittelumallien osalta on tutkielmassa käytetty keskeisenä sanastona Kai Koskimiehen Pientä oliokirjaa (1997). Suunnittelumalleja ei ole lähdetty suomentamaan kahdesta syystä. Ensinnäkin suunnittelumalleista kirjoitetaan lähes ainoastaan englanniksi, eikä tämän tutkielman tavoitteen kannalta suunnittelumallien nimiä kannattanut suomentaa. Toiseksi suomentaminen olisi häivyttänyt suunnittelumallien ytimen, koska suunnittelumallin nimi on olennainen osa itse suunnittelumallia.

Tutkielman aineiston muodostavat kirjoittajan oma kokemus ja kirjallisuus. Kirjallisuutena on käytetty pääasiallisesti tieteellisiä artikkeleita ja julkaisuja. Tutkimusote on käsitteellisteoreettinen ja tulokset ovat tämän mukaisia. Kuudennessa luvussa esitetään tutkielman keskeinen anti. Luvussa kuvataan kuinka olio-oppimista voidaan nopeuttaa valitsemalla oikeita suunnittelumalleja opiskeltaviksi ja tutkittaviksi.

Tutkielma jakaantuu seitsemään lukuun. Johdannon jälkeen esitellään toisessa luvussa ohjelmistotuotanto ja sen ongelmia sekä käsitellään, kuinka oliokeskeisyys pyrkii ratkaisemaan näitä ongelmia. Luvussa 3 esitellään oliokeskeisyyttä tarkemmin ja käydään läpi suunnittelumallien ymmärtämiseksi tarvittavaa oliokäsitteistöä. Luvussa 4 sovelletaan ensimmäistä näkökulmaa suunnittelumalleihin ja esitellään niiden valintaa ja käyttöä helpottava luokittelu. Luvussa 5 sovelletaan toista näkökulmaa, tarkoituksena katsoa suunnittelumalleja uudelleenkäytettävyyden kannalta. Tämä tehdään tutkimalla suunnittelumallien käyttöä ohjelmistotuotantoprosessissa. Kolmas ja viimeinen näkökulma suunnittelumalleihin otetaan esille kuudennessa luvussa, jossa nivotaan suunnittelumallien käyttö koko oliokeskeisyyden oppimiseen ja esitellään, kuinka suunnittelumalleja voidaan käyttää siirryttäessä olio-oppimistasolta seuraavalle. Tutkielman päättää luvussa seitsemän esitetty yhteenveto.

2 OHJELMISTOTUOTANTO

Vapaasti tulkiten ohjelmistotuotanto (Software engineering, SE) tarkoittaa ohjelmistotyötä, jonka tuloksena syntyvät järjestelmät täyttävät käyttäjiensä kohtuulliset toiveet ja odotukset ja joka lisäksi valmistuu laadittujen aikataulujen ja kustannusarvioiden mukaisesti. Termi käsittää siis kaikki ohjelmiston tuotantoprosessiin liittyvät osa-alueet: määrittelyn, suunnittelun, toteutuksen, testauksen, käyttöönoton, ylläpidon, laatujärjestelmän, projektinhallinnan, dokumentoinnin, tuotehallinnan ja laadunvarmistuksen (Haikala & Märijärvi, 1997). *Ohjelmistotuotannon* klassisin ja yksi eniten lainattuja määritelmiä on Fritz Bauerin esittämä määritelmä: "Luotettavien tekniikoiden luominen ja käyttäminen niin, että voidaan tuottaa taloudellisesti ohjelmistoja, jotka ovat luotettavia ja toimivat tehokkaasti todellisissa koneissa (Naur & Randell, 1969)."

Tässä luvussa kuvataan ohjelmistotuotantoa. Aluksi kerrotaan, mistä näkökulmasta tässä on ohjelmistoja kuvattu. Tämän jälkeen kuvataan lyhyesti ohjelmistotuotannon eri vaiheet ja minkälaisia elinkaarimalleja on kehitetty tukemaan ohjelmistotuotantoa. Ohjelmistotuotannon oleellisia ongelmia sekä näiden ratkaisuyrityksiä kuvataan seuraavaksi ja lopuksi esitetään, kuinka oliokeskeisyys pyrkii ratkaisemaan esitettyjä ongelmia.

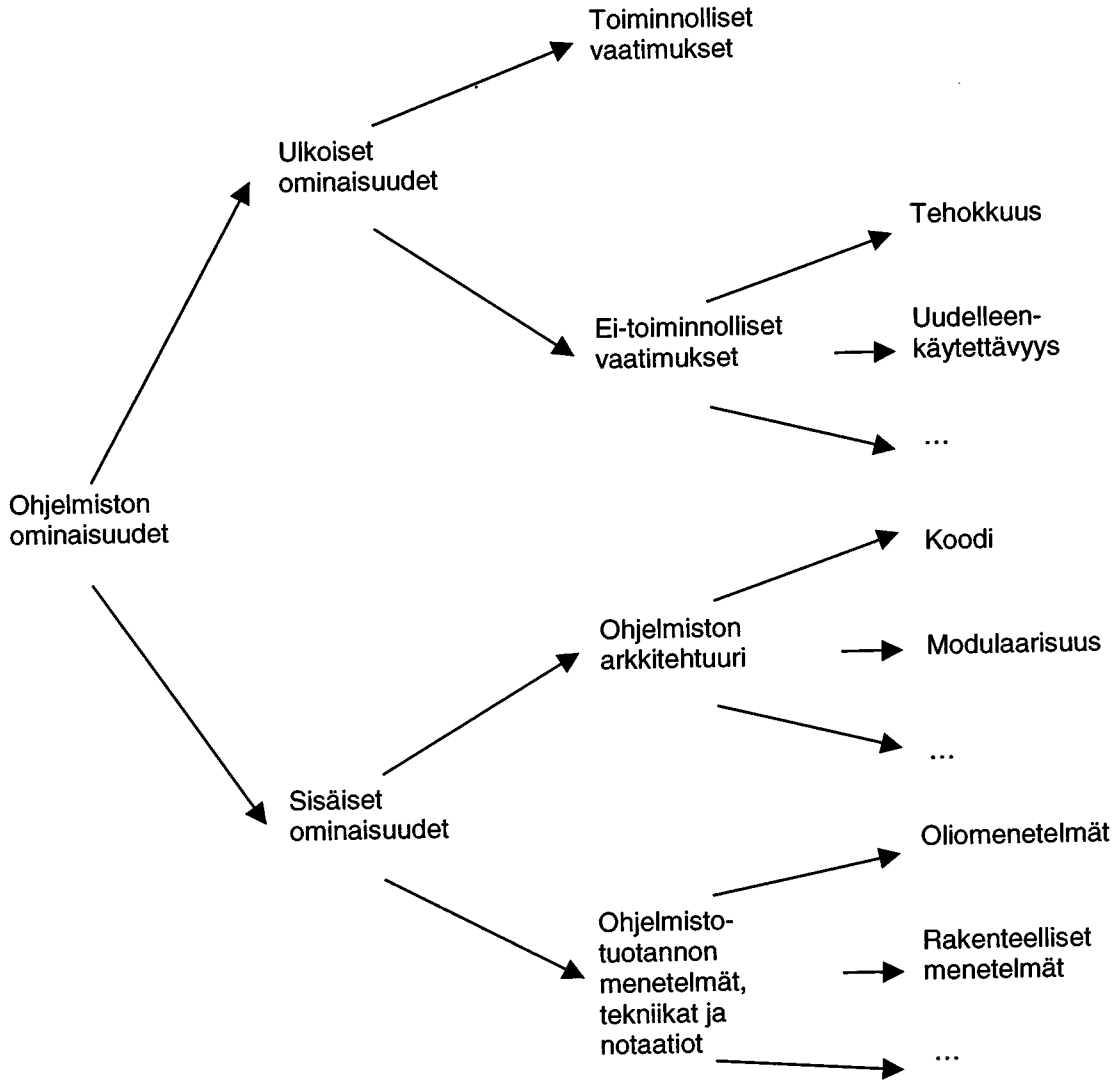
2.1 Ohjelmiston ominaisuudet

Ohjelmistotuotanto koostuu kahdesta sanasta: ohjelmisto ja tuotanto. *Ohjelmisto* tarkoittaa sovellusta sekä siihen liittyvää dokumentaatiota, joka kertoo, kuinka asentaa, käyttää ja ylläpitää sovellusta (Sommerville 1992). *Tuotanto* taas tarkoittaa edellä kuvatun ohjelmiston tekemistä ja tekemiseen liittyviä toimenpiteitä.

Perinteisesti ohjelmistoja tutkitaan sen mukaan, miten hyvin algoritmit toimivat ja kuinka nopeasti ohjelmisto selviytyy sille annetuista tehtävistä. Vaikka nämä ovatkin tietyissä ympäristöissä tärkeitä kriteereitä, eivät ne koske kaikkia ohjelmistoja. Todellisuudessa ohjelmistoja rakentavat ihmiset ihmisille. Jotta voimme ymmärtää ohjelmistoja, täytyy meidän nähdä ohjelmiston todellinen luonne, se olennaisuus, jonka jokainen ohjelma sisältää. Platonin sanoin, meidän on löydettävä ohjelmiston idea.

Kuviossa 1 esitetään ohjelmiston ominaisuudet. Tässä kuviossa ohjelmistoon otetaan kaksi eri näkökulmaa: sisäinen ja ulkoinen. Ensimmäinen näkökulma, ulkoiset ominai-

suudet, kertoo kuinka ohjelmisto näkyy käyttäjälle. Loppujen lopuksi käyttäjää ei kiinnosta, kuinka ohjelmisto on rakennettu, millä ohjelmointikielellä se on toteutettu tai minkälaisia tietokantoja siinä on käytetty. Käyttäjä haluaa, että ohjelmisto tekee ne tehtävät, jotka sille on annettu ja että ohjelmisto tekee nämä tehtävät hyvin. (Meyer 1988.)



Kuvio 1. Ohjelmiston ominaisuudet

Ohjelmiston käyttäjä on ihminen, jonka jokapäiväiseen työhön ohjelmisto vaikuttaa. Käyttäjiksi täytyy laskea myös sellaiset henkilöt, jotka eivät suoranaisesti käytä ohjelmistoa, mutta joiden työhön se vaikuttaa. Tällaisia henkilöitä ovat esimerkiksi käyttäjien esimiehet, joiden on huolehdittava siitä, että tarpeellinen laitteisto on alaisten käytettävissä.

Käyttäjän vaatimukset jakaantuvat edelleen kahteen eri luokkaan: toiminnalliset ja ei-toiminnalliset vaatimukset (Sommerville 1992). Toiminnalliset vaatimukset kertovat, mitä ohjelmiston on tehtävä. Ohjelmiston on esimerkiksi tulostettava yhteenvetotiedot työntekijöiden palkoista tai ohjelmiston on laskettava lentokoneen arvoitu saapumisaika vastatuulen nopeus huomioiden. Käyttäjää ei kiinnosta, kuinka nämä palvelut toteutetaan, joten ohjelmistonkehittäjän ei tulisi esittää mitään toteutukseen liittyviä käsitteitä kuvatessaan toiminnallisia vaatimuksia.

Ei-toiminnalliset vaatimukset kertovat, kuinka hyvin, missä rajoissa ja millä ehdoilla ohjelmiston on toimintansa suoritettava. Esimerkiksi ohjelmiston on vastattava kaikkiin käyttäjän tekemiin toimintoihin kahden sekunnin sisällä ja ellei näin tapahdu, on ajan kulumisesta informoitava käyttäjää. Ei-toiminnallisiksi vaatimuksiksi luetaan mm. suoritusteho, luotettavuus, siirrettävyys ja turvallisuus.

Toinen näkökulma ohjelmistoon on sisäinen. Se kertoo, kuinka ohjelmisto on sisäisesti rakentunut. Sisäisiä ominaisuuksia ovat mm. modulaarisuus ja koodin luettavuus. Nämä ominaisuudet näkyvät vain ohjelmistotuotannon ammattilaisille. Sisäiset ominaisuudet kuitenkin määräävät, kuinka hyvin ohjelmisto lopulta vastaa käyttäjän vaatimuksiin, eli ulkoisiin ominaisuuksiin. (Meyer 1988.)

Sisäiset ominaisuudet voidaan jälleen jakaa kahteen eri luokkaan: a) arkkitehtuuri ja b) ohjelmistotuotannon notaatiot, tekniikat ja menetelmät. Arkkitehtuuri kuvaa ja määrää ohjelmiston sisäisen rakenteen. Niillä ratkaisuilla, joita arkkitehtuuria määriteltäessä ja toteutettaessa tehdään, vaikutetaan ohjelmiston uudelleenkäytettävyyteen, laajennettavuuteen ja yhteensopivuuteen.

Ohjelmistotuotannon notaatiot, tekniikat ja menetelmät määrittelevät puolestaan, kuinka ohjelmisto on rakennettu. Osaltaan tähän voidaan katsoa myös sellaisia tekijöitä, kuten projektinhallinta, henkilöstö yms. seikat, jotka liittyvät koko ohjelmistotuotantoprosessiin. *Notaatiolla* tarkoitetaan niitä kuvaustapoja, joita käytetään ohjelmistoa määriteltäessä, suunniteltaessa ja toteutettaessa. Ne antavat tekniset ohjeet, kuinka rakentaa ohjelmisto. Erilaisia notaatioita löytyy kaikkiin ohjelmistotuotannon elinkaaren vaiheisiin mm. projektin suunnitteluun ja arviointiin, ohjelmiston vaatimusanalyysiin sekä ylläpitoon. Ohjelmistotuotannon notaatiot sisältävät usein erityisen ohjelmointikielen tai graafisen notaation ja esittelevät kriteeristön ohjelmiston laadunvalvonnalle. (Pressman 1992, Booch 1994.)

Ohjelmistotuotannon *tekniikat* antavat säännöt, kuinka rakentaa malleja järjestelmästä. Tekniikoiden avulla voidaan löytää notaatioilla tehdyistä kuvauksista virheet ja epäjohdonmukaisuudet. Ohjelmistotuotannon *menetelmät* toimivat yhdistävinä tekijöinä notaatioiden ja tekniikoiden välillä ja ne mahdollistavat järkevän ja oikea-aikaisen kehityksen ohjelmistoille. Menetelmät määrittelevät notaatioiden käyttöjärjestyksen, vaadittavat tuotokset (dokumentit, raportit, kaavakkeet, jne.), valvontamekanismit laadunvarmistukseen ja muutoksenhallintaan sekä tarkistuspisteet, joilla voidaan valvoa edistymistä.

Ohjelmisto on siis tarkoitettu täyttämään käyttäjänsä vaatimukset hyviä insinööritaitoja tehokkaasti käyttäen. Kuitenkin koko termi ohjelmistotuotanto esitettiin samassa NATO:n 1968 järjestämässä kokouksessa, jossa ensimmäisen kerran todettiin, että ohjelmistotuotanto on kriisissä. Nyt 30 vuoden jälkeen ohjelmistotuotanto on edelleen kriisissä. Tässä vaiheessa voidaan kyllä todeta, että kyseessä on krooninen sairaus, josta ohjelmistotuotanto on kärsinyt koko aikuisikänsä. (Pressman 1992.)

2.2 *Ohjelmistotuotannon elinkaarimallit*

Ohjelmistotuotanto toteutuu jonkin elinkaarimallin mukaan. *Elinkaarimalli* esittää, mitä ohjelmistotuotannon vaiheita suoritetaan missäkin järjestyksessä. Esimerkiksi perinteisessä vesiputousmallissa ohjelmistotuotannon vaiheet suoritetaan peräkkäisessä järjestyksessä (Royce 1970). Erilaisia elinkaarimalleja ovat esitelleet Roycen (1970) lisäksi mm. Boehm (1986), Gomaa & Scott (1981) ja Gilb (1985). Vaikka edellä mainitut elinkaarimallit eroavat huomattavastikin toisistaan, niin löytyy niistä merkittäviä yhtäläisyyksiäkin. Jokainen elinkaarimalli sisältää seuraavat vaiheet (vrt. Davis ym. 1988, Sommerville 1996): vaatimusmäärittely, analyysi, suunnittelu, toteutus, testaus ja ylläpito.

Sommerville (1996) jakaa elinkaarimallit määrittelypohjaisiin ja evolutionaarisiin malleihin. Määrittelypohjaiset elinkaarimallit pohjautuvat vesiputousmalliin ja luottavat siihen, että edellisen vaiheen tuotos on täydellinen. Tällaisia malleja ovat mm. perinteinen vesiputousmalli, kasvattavat mallit (incremental models) (Mills ym 1980) ja ns. puhdashuone lähestymistapa (cleanroom approach) (Mills ym 1987).

Evolutionäärinen lähestymistapa sitoo toisiinsa määrittelyn, suunnittelun ja toteutuksen. Näitä suoritetaan tarpeelliseksi katsotussa järjestyksessä ja niiden oletetaankin vaikutta-

van toisiinsa. Tällaisia lähestymistapoja ovat mm. prototyyppilähestymistapa (Gomaa & Scott 1981) ja Boehmin spiraalimalli (1986).

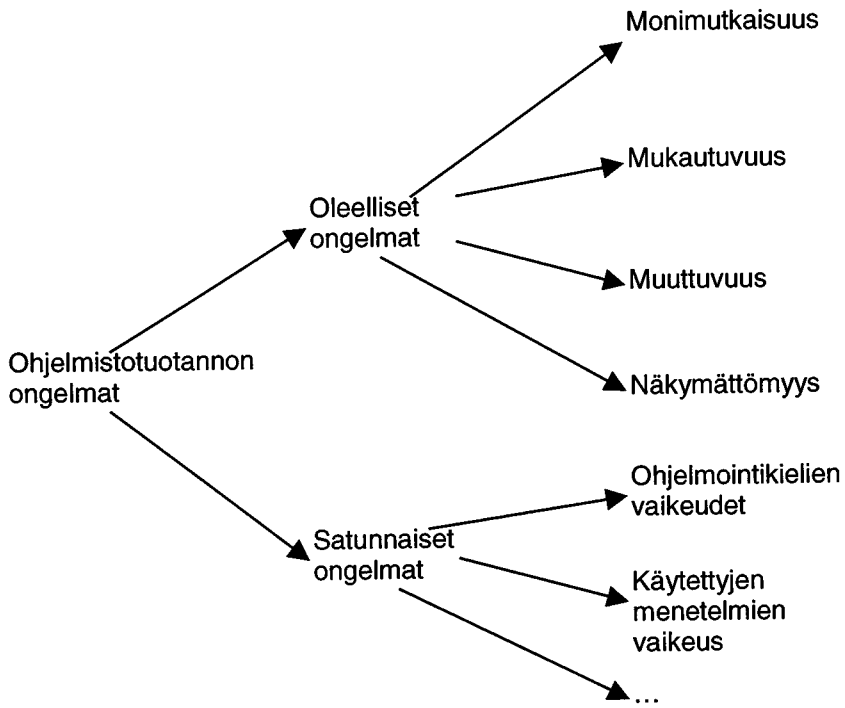
2.3 *Ongelmia 30 vuoden ajalta*

Ohjelmistotuotanto on siis "kroonikkopotilas". Ongelman ytimenä ovat jatkuvasti kasvavat ja monimutkaistuvat ohjelmistot. Tämän mahdollistavat kasvavat laitteistotehot, jotka ovat tuhatkertaistuneet tietokoneen alkuajoista lähtien. Silloin kun tietokoneita ei ollut, ei ohjelmistotuotanto ollut vaikeaa. Kun tietokoneiden käsittelemät ohjelmat olivat pieniä, olivat ohjelmistotuotannon ongelmat pieniä. Kun tietokoneista on tullut valtavan tehokkaita, ovat ohjelmistotuotannon ongelmat yhtä valtavia (Dijkstra, 1972). Ohjelmistotuotannon kehitys ei ole ollut epätavallisen hidasta, tietokoneiden tehon kasvu on ollut epänormaalin nopeaa.

Hyviä kuvauksia ohjelmistotuotannon ongelmista ovat antaneet mm. Booch (1994), Brooks (1975; 1987), Jackson (1994) sekä Parnas ja Clements (1986). Jackson kiinnittää huomionsa siihen, että ohjelmistosuunnittelijat keskittyvät liiaksi ratkaisuihin, kun pitäisi keskittyä ongelmiin. Ohjelmistotuotannon notaatiot, tekniikat ja menetelmät ovat kaikki ratkaisuvetoisia. Esimerkiksi ohjelmointikielet ja -ympäristöt tarjoavat käytettäväksi sellaisia käsitteitä kuin proseduuri, prosessi, tietorakenne, funktio, tyyppi, viesti ja tiedosto. Kaikki nämä auttavat kuvaamaan ratkaisua, kun kuvaamme ohjelmistoa. Toisaalta on mainittu, että projektit epäonnistuvat, vaikka käytössä olisikin uusinta teknologiaa. Tämä sen vuoksi, että ohjelmistokehittäjillä ei ole ratkaisuja tavanomaisiin ongelmiin (Coplien & Schmidt 1995). Suunnittelumallit toimivat yhtenä tapana tallentaa tietoa tavanomaisten ongelmien ratkaisusta.

Jo klassikoksi muodostuneessaan artikkelissa "No silver bullet – the essence and accidents of software engineering" Brooks (1987) ei näe hopealuotia, joka moninkertaistaisi ohjelmistokehityksen tuottavuuden, luotettavuuden ja yksinkertaisuuden. Artikkelissaan Brooks jakaa ohjelmistotuotannon ongelmat kahteen päätyyppiin: olennaisiin ja satunnaisiin[1] (kuvio 2). Oleelliset ongelmat ovat olemassa ohjelmistotuotannon luonteen vuoksi. Niitä voidaan yrittää hallita, mutta ne eivät koskaan poistu. Satunnaiset ongelmat taas eivät ole välttämättömiä. Satunnaisia ongelmia ovat mm. käytettyjen ohjelmointikielien ja menetelmien aiheuttamat käyttövaikeudet.

[1] Termit on suomennettu Esa Saarisen (1985) Aristoteleen yhteydessä käyttämien termien 'olennainen' ja 'satunnainen' mukaan.



Kuvio 2. Ohjelmistotuotannon ongelmat Brooksin (1987) jaottelun mukaan

Oleellisia ongelmia ovat ohjelmistojen luontainen monimutkaisuus, mukautuvuus, muuttuvuus ja näkymättömyys. Ensimmäinen oleellinen ongelma on monimutkaisuus. Ohjelmistot ovat isoja ja ratkaisevat laaja-alaisia ongelmia. Kokoonsa nähden ohjelmistot ovat luultavasti mutkikkaimpia ihmisen luomuksia, eikä niissä ole täsmälleen samanlaisia osia. Kun ohjelmistoja laajennetaan ja niihin lisätään erilaisia osia, osien väliset vuorovaikutussuhteet kasvavat nopeammin kuin lineaarisesti. Monimutkaisuutta ei voida poistaa samalla tavoin kuin luonnontieteissä, eli yksinkertaistamalla ja tekemällä approksimaatioita, koska ohjelmistot ovat epäjatkuvia.

Monet ohjelmistotuotannon ongelmat aiheutuvat suoraan mutkikkuudesta. Projektiryhmän sisäinen kommunikointi on vaikeaa (vrt. Brooks 1975). Ihmisen on mahdotonta ymmärtää ohjelman kaikki mahdolliset tilat. Toimintojen mutkikkuus tekee ymmärtämisen vieläkin vaikeammaksi. Uusien toimintojen ja ominaisuuksien lisääminen ohjelmistoon aiheuttaa sivuvaikutuksia ympäri ohjelmistoa.

Toinen oleellinen ongelma on mukautuvuus. Ohjelmistojen taustalla ei ole vaikuttamassa luonnonlakeja, joten mutkikkuus on mielivaltaista. Ohjelmiston täytyy mukautua mitä moninaisimpiin käyttöliittymiin ja toimintatapoihin. Ohjelmiston täytyy mukautua olemassa oleviin järjestelmiin, koska ohjelmisto on viimeisin tulokas. Toisaalta oletetaan, että ohjelmistoa on myös helpoin muuttaa.

Kolmas ohjelmistotuotannon oleellinen ongelma on muuttuvuus. Ohjelmistoon tulee jatkuvasti muutoksia senkin jälkeen, kun se on toimitettu asiakkaalle. Onnistunuttakin ohjelmistoa täytyy muuttaa, koska käyttäjät keksivät ohjelmistolle uusia käyttötarkoituksia, ja toisaalta, mitä parempi ohjelma, sitä kauemmin käyttäjät haluavat sitä käytää ja muokata sitä tarpeisiinsa. Vaikka käyttäjä ei muutoksia vaatisikaan, niin laitteistojen kehitys aiheuttaa, että ohjelmistoa on muutettava vastaamaan uusia laitteistokokoonpanoja.

Neljäs oleellinen ongelma on ohjelmiston näkymättömyys. Ohjelmistoja ei pystytä kuvaamaan tarkasti perinteisten kuvaustapojen avulla. Ohjelmistoille ei ole vieläkään löydetty luonnollista kuvaustapaa. Erilaiset kartat ja kaaviot ovat riittämättömiä, ja useiden suunnattujen verkkojen avulla kuvattu ohjelmisto on vaikeasti hahmotettavissa. Visualisoimattomuus estää joidenkin vahvojen ajatteluvälineiden käyttöä. Yhdelle ihmiselle koko ohjelmiston suunnittelu on vaikeaa, ja toisaalta kommunikointi muiden kanssa on vaikeaa.

2.4 ... ja ratkaisuyrityksiä

Sekä Brooks (1987) että Jackson (1994) esittävät oman näkemyksensä siitä, kuinka nämä perustavaa laatua olevat ongelmat voidaan ratkaista. Jackson lähtee siitä, että ongelma-avaruus on kuvattava ongelmalähtöisesti. Hänen mielestään mm. oliokeskeinen lähestymistapa on liian yleinen antaakseen tarkan kuvan siitä, mitä olioita ongelman ratkaisemiseksi tarvittaisiin taikka mitä operaatioita ja tietorakenteita olisi tehtävä. Hän mainitsee uusista lähestymistavoista suunnittelumallit yhtenä mahdollisena osaratkaisuna ongelmien kuvaamiseen.

Brooks (1987) mainitsee lupaavina lähestymistapoina lisätä ohjelmistokehityksen tuottavuutta, luotettavuutta ja yksinkertaisuutta seuraavilla neljällä tavalla:

1. Ostaminen vastaan rakentaminen: kaikista radikaalein ratkaisu ohjelmistotuotannolle on olla tuottamatta ohjelmistoa itse. Mikäli ohjelmistoa myydään n kappaletta, on sen tuottaneiden ihmisten tuottavuus n kertaa suurempi, kuin jos ohjelmistoa olisi myyty vain yksi kappale.
2. Vaatimusten tarkentaminen ja nopea protoilu: ohjelmiston vaatimusten tarkka määrittely on koko prosessin vaikein ja kriittisin tehtävä. Asiakas ei yleensä tarkasti tiedä, mitä hän haluaa, mutta tietää ehdottomasti, mitä hän ei halua. Kun voidaan esit-

tää valmistettavasta ohjelmistosta prototyyppi, saadaan asiakkaalta nopeasti palaute siitä, onko ohjelmisto sellainen kuin asiakas on toivonut.

3. Vähittäinen kehittäminen – kasvata, älä rakenna ohjelmistoa: ohjelmistoja kasvattamalla voidaan hallita monimutkaisempien ohjelmistojen tekeminen kuin rakentamalla. Kehittäjät ja asiakkaat saavat nopeammin tehtyä jotain näkyvää, ja tämä vaikuttaa positiivisesti koko kehitystyöhön.
4. Hyvät suunnittelijat: keskeinen tekijä ohjelmistotuotannossa on ihminen. Tutkimus toisensa perään osoittaa, että hyvät suunnittelijat tuottavat ratkaisuja, jotka ovat nopeampia, pienempiä, yksinkertaisempia, puhtaampia ja joita tuotetaan pienemmällä vaivalla kuin huonojen tai keskinkertaisten suunnittelijoiden ratkaisut.

Booch (1994) keskittyi kuvaamaan Brooksin innoittamana järjestelmien monimutkaisuutta. Hänen kuvaamansa monimutkaisuus on hyvin samankaltaista kuin Brooksilla. Booch tarjoaa ratkaisuiksi hajottamisen, abstraktion ja hierarkian. Hajottaminen perustuu periaatteeseen "divide et impera", hajota ja hallitse. Tässä yhteydessä hajottamisella ymmärretään osiin jakamista.

Abstraktio on ihmiselle ominaista asioiden yksinkertaistamista. Asioita luokitellaan ja asetetaan järjestykseen ilman, että huomioitaisiin kaikkia yksityiskohtia. Ihminen pystyy hallitsemaan laajempia kokonaisuuksia, kun hänen tarvitsee ottaa huomioon vain osa ilmiöön tai tapahtumaan vaikuttavista tekijöistä. Ohjelmistotuotannossa esiintyviä abstraktioita ovat mm. aliohjelmat, moduulit ja oliot (Budd 1991). Oliokeskeisessä lähestymistavassa abstraktio voidaan toteuttaa perinnällä.

Millerin (1956) mukaan ihminen ei kykene käsittelemään kerrallaan kuin 7 ± 2 yksikköä. Yksiköllä tarkoitetaan käsitettä, hallittavaa ajatuskokonaisuutta[2]. Jotta ihminen pystyy hallitsemaan suurempia kokonaisuuksia, on hänen tehtävä käsitteistä ajatusryppäitä, kuitenkin aina 7 ± 2 säännön puitteissa. Esimerkiksi ohjelmistoissa keskenään keskustelevat oliot järjestetään luokiksi. Luokka kuvaa kaikkia olioita yhdenmukaisella tavalla eikä tarvitse ymmärtää kaikkien olioiden keskinäisiä viestejä.

Hierarkia on kolmas tapa tuoda järjestystä monimutkaisiin järjestelmiin. Noudattamalla hierarkkista järjestystä saadaan monimutkaisiin järjestelmiin samankaltaisuutta. Kuten jo Brooks (1987) mainitsi, niin kaikki luonnon näennäisesti yksinkertaiset rakenteet

[2] Monimutkaisuus johtuu siis ihmisen kyvystä käsitellä tietoa. Dijkstra (1989) on todennutkin: "Yritys saada koneet matkimaan ihmistä on aina tuntunut minusta aika typerältä. Minä mieluummin yrittäisin saada ne matkimaan jotakin parempaa." Koneissahan on kuitenkin äärellinen määrä tiloja ja mikäli pystyisimme hallitsemaan tätä äärellisyyttä, ei ohjelmiston tekeminen tuntuisi niin monimutkaiselta.

muodostuvat hierarkioista. Esimerkiksi kasvit muodostuvat useista eri soluyksiköistä. Ja nämä soluyksiköt muodostavat kasvin rakenteen: juuret, varren ja lehdet. Nämä eri soluyksiköt toimivat yhdessä muodostaen monimutkaista käyttäytymistä, kuten fotosynteesiä ja haihtumista. Oliokeskeisissä ohjelmissa hierarkiaa kuvaa koostamissuhde.

Jotta edellä mainituista ratkaisuista olisi jotain hyötyä ohjelmistojen teossa, täytyy niille luoda yhteinen viitekehys, eli menetelmä niiden käyttöön. Yksi nykyään suuressa suosiossa oleva lähestymistapa ohjelmistotuotantoon on oliokeskeinen lähestymistapa. Seuraavassa kohdassa keskitytään kuvaamaan, kuinka oliokeskeisyys lähestyy monimutkaisuutta ja kuinka se pyrkii hallitsemaan sitä.

2.5 Ratkaisuna oliokeskeinen lähestymistapa

Syy siihen, miksi oliokeskeisyyteen siirrytään yhä enemmän, perustuu ohjelmistokustannusten hallintaan. Ohjelmistokustannuksia syntyy niin kehitysvaiheessa kuin ylläpitovaiheessakin. Mikäli onnistutaan tekemään hyvä ohjelmisto, ihmiset haluavat käyttää sitä pitempään ja näin ylläpitokustannukset nousevat edelleen. Aina kun ylläpitotyötä syntyy, aiheuttaa se muutoksia ohjelman määrittelyyn. Oliokeskeisyyden avulla pystytään paikallistamaan muutoksen kohteet paremmin kuin perinteisissä lähestymistavoissa. Oliokeskeisyys mahdollistaa myös uudelleenkäytön paremmin kuin perinteiset menetelmät. (Koskimies 1997.)

Oliokeskeisyyden idea on, että simuloidaan sovellukseen liittyvää maailmaa. Sovelluksen edustama maailma jaetaan sitä edustaviin komponentteihin, eli olioihin. Näillä olioilla on samat ominaisuudet ja käyttäytyminen kuin maailmassa esiintyvillä komponenteilla. Näin ollen ohjelmisto on samankaltainen kuin käsitemaailma, joka liittyy ohjelmistoon. (Koskimies 1997.)

Koskimiehen (1997) mukaan oliokeskeisyydellä saavutetaan seuraavat edut:

1. Sovellukseen tehtävät muutokset ovat suoraan verrannaisia sovelluksen kuvaaman maailman muutoksiin. Näin ollen tehtävät muutokset on helpompi paikallistaa ohjelmistosta.
2. Ohjelmiston muutoksen suuruus on jotakuinkin yhtä suuri kuin käsitemaailmaan tuleva muutos.
3. Sovelluksen rakentamisesta tulee systemaattisempi ja hallittavampi.

4. Uudelleenkäytöstä tulee helpompaa, koska oliot kuvaavat maailmassa esiintyviä käsitteitä sekä niiden ominaisuuksia ja toimintaa.

Brooksin (1987) esittämiin oleellisiin ongelmiin oliokeskeisyys hyökkää seuraavin tavoin:

1. Monimutkaisuus: Oliokeskeinen lähestymistapa mallintaa maailmaa olioin, jotka kuvaavat mahdollisimman hyvin maailman käsitteitä. Näin ei tarvitse kuluttaa ylimääräistä kapasiteettia muuttamalla maailman käsitteistöä vastaamaan ohjelmassa esiintyvää käsitteistöä. Monimutkaisuutta voidaan hallita myös abstrahoimalla. Oliokeskeisyydessä abstrahointi voidaan toteuttaa perinnän avulla, eli samankaltaisille käsitteille voidaan luoda yliluokka (vrt. 3. luku).
2. Mukautuvuus: Oliokeskeisiä ohjelmistoja on helpompi mukauttaa vastaamaan käyttäjien vaatimuksia ja toiveita.
3. Muuttuvuus: Ohjelmistoon tulevat muutokset on helpompi paikallistaa ja jäävät helpommin paikallisiksi, ilman sivuvaikutuksia koko ohjelmistoon. Muutosten kokoa voidaan myös arvioida paremmin.
4. Näkymättömyys: Näkyvyyttä lisätään ennen kaikkea oliomenetelmillä, mutta myös olioiden käsite on käyttäjälle helpompi kuin proseduuri tai aliohjelma (vrt. mm. Rumbaugh ym. 1991). Käyttäjät voivat antaa kommentteja mallinnettavasta järjestelmästä helpommin kuin rakenteellisilla menetelmillä työskenneltäessä.

2.6 Yhteenveto

Tässä luvussa käytiin läpi ohjelmistotuotantoon keskeisesti liittyviä seikkoja. Aluksi määriteltiin ohjelmiston ominaisuudet. Ne jaettiin ulkoisiin ja sisäisiin ominaisuuksiin. Käyttäjää kiinnostaa lähinnä ohjelmiston ulkoiset ominaisuudet eli se, kuinka ohjelmisto näkyy ja toimii käyttäjälle. Sisäiset ominaisuudet ovat lähinnä ohjelmistotuotannon ammattilaisille näkyviä asioita. Tämän jälkeen esiteltiin lyhyesti ohjelmistotuotannon elinkaarimallit. Seuraavaksi tarkasteltiin ohjelmistotuotantoon liittyviä keskeisiä ongelmia. Näiksi tunnistettiin ohjelmistojen monimutkaisuus, joustavuus, muuttuvuus ja näkymättömyys. Ratkaisuna ongelmiin nähtiin mm. ohjelmistojen ostaminen, vaatimusten tarkentaminen ja protoilu, ohjelmistojen kasvattaminen sekä parempien suunnittelijoiden kouluttaminen. Yhdeksi lupaavaksi lähestymistavaksi ohjelmistotuotannon kehittämiseen nähtiin oliokeskeinen lähestymistapa. Oliokeskeisyyden avulla voidaan parantaa tunnistettuja ongelmia erityisesti oliokeskeisellä mallinnuksella ja perinnällä.

3 OLIOKESKEISYYS

Oliokeskeisyys toistuu ohjelmistotuotannon yhteydessä nykyään usein. Ohjelmistotuotannon tutkimuksessa ja myös teollisessa ohjelmistotuotannossa on oliokeskeisyys saanut merkittävän osan. Oliokeskeisyyden kaksi merkittävintä myyntiargumenttia ovat olleet uudelleenkäytettävyys ja mallintaminen (Taivalsaari 1993).

Tämän luvun tarkoituksena on esittää suunnittelumallien ymmärtämisen kannalta keskeinen oliokäsitteistö. Tämän jälkeen luvussa esitetään lyhyesti erilaisia lähestymistapoja oliokeskeiseen ohjelmistotuotantoon ja esitellään oliomenetelmiä. Lopuksi esitetään oliosuunnittelun keskeisiä käsitteitä sekä millaisia lähestymistapoja suunnittelumalleissa on käytetty ratkaistaessa ohjelmistotuotannon ongelmia.

3.1 Oliokäsitteistöä

Wegner (1990) määrittelee olion kokoelmaksi operaatioita, joilla on yhteinen tila. Tässä työssä oliosta käytetään Taivalsaaren (1993) käyttämää määritelmää, jonka mukaan:

olio = identiteeti + tila + käyttäytyminen, tai
olio = identiteetti + attribuutit + operaatiot

Olion *attribuutit* kuvaavat tietoja ja *operaatiot* kuvaavat olion toimintaa. Yhdessä olion attribuutteja ja operaatioita kutsutaan olion *piirteiksi*. Oliot ovat yksittäisiä entiteettejä, jonka johdosta kullakin oliolla on oltava identiteetti. *Identiteetin* avulla kaksi toistensa kopioita olevaa oliota voidaan erottaa toisistaan. Edellä mainittujen seikkojen lisäksi voidaan vielä todeta, että olio on *suojattu* kokonaisuus (Koskimies 1997). Suojaus merkitsee, että oliota voidaan käyttää vain etukäteen määritetyllä tavalla.

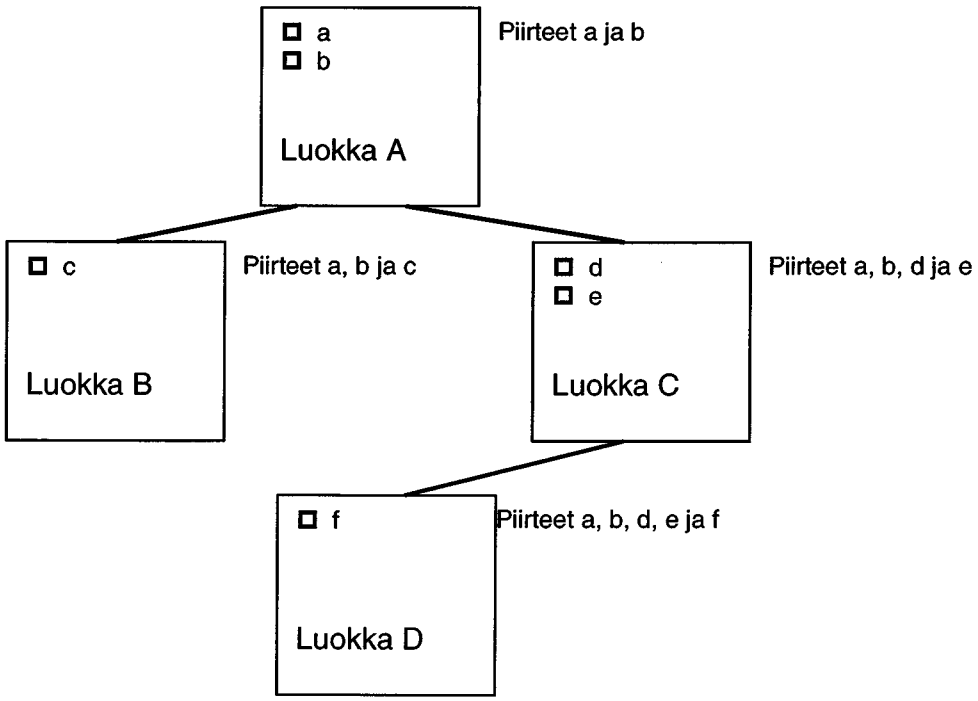
Luokat toimivat olioiden malleina tai "piparkakkumuotteina", joiden mukaan varsinaiset oliot voidaan luoda (Taivalsaari 1993). Oliota kutsutaankin luokan *ilmentymäksi*. Voidaankin sanoa, että luokat ovat yleisiä kuvauksia olioista, joiden mukaiseksi itsenäinen olio luodaan. Luokka sisältää tiedon siitä, minkälaisia attribuutteja ja operaatioita olioon liittyy. Olion attribuuttien arvot ilmaisevat olion tilan. Oliolla on myös yksilöllinen identiteetti. Luokan kuvausta luokkana kutsutaan *metaluokaksi* ja metaluokasta luotua

oliota *metaolioksi* (Koskimies 1997). Metaolio huolehtii luokasta tehtävien olioilmentymien luomisesta sekä sisältää ilmentymien tarvitsemien palvelujen antamisesta.

Olion operaation määritelty sisältää nimen, parametrit ja palautusarvon. Tämä tunnetaan nimellä operaation *kutsumuoto* (signature). Kaikki kutsumuodot yhdessä muodostavat olion *rajapinnan*. Olion rajapinta kuvaa kaiken sen toiminnan, jonka olio tarjoaa asiakkailleen. Olio pystyy toteuttamaan kaikki rajapintansa mukaiset pyynnöt.

Laajasti katsoen *perintä* tarkoittaa ominaisuuksien tai piirteiden vastaanottamista toiselta, normaalisti tämä johtuu erityisestä suhteesta vastaanottajan ja antajan välillä (Taivalsaari 1993). Perinnän avulla uusia luokkia voidaan määritellä olemassa olevien avulla. Toisin sanoen vastaanottava luokka saa antajan sisältämät piirteet. Perittyjen piirteiden lisäksi luokka voi esitellä omia piirteitä.

Kuviossa 3 näemme, kuinka luokan A piirteet a ja b periytyvät luokalle C. Luokka C lisää kuvaukseen itselleen ominaisia piirteitä d ja e. Näin ollen luokan C piirteitä ovat nyt a, b, d ja e. Luokkaa A kutsutaan luokan C *yliluokaksi* ja luokkaa C luokan A *aliluokaksi*. Käytössä ovat myös termit *esi-isäluokka* ja *jälkeläisluokka*, joita voidaan käyttää silloin, kun luokkien välille asettuu muita luokkia. Kuvion 3 esimerkissä luokka D on A luokan jälkeläinen ja vastaavasti luokka A on D luokan esi-isä.



Kuvio 3. Luokkahierarkia (Koskimies 1997)

Edellä kuvatussa tapauksessa puhutaan yksittäisperiytymisestä. *Yksittäisperiytymisessä* kullakin luokalla voi olla vain yksi yliluokka. Mikäli luokka D perisi sekä B- että C-luokan piirteitä, kutsuttaisiin tätä *moniperiytymiseksi*. Vaikka moniperiytymisellä saavutetaan huomattavasti enemmän mahdollisuuksia luokan piirteiden lisäämiseksi, niin se lisää monimutkaisuutta ohjelmiston rakentamiseen (Taivalsaari 1993).

Perinnän yhteydessä on hyvä muistaa, että perintää voidaan katsoa hyvin monesta näkökulmasta. Erilaisia luokitteluja perinnälle löytyy esimerkiksi Meyerilta (1996), Sakkiselta (1992) ja Taivalsaarelta (1993). Perintä voidaan jakaa oleelliseen ja satunnaiseen (Sakkinen 1992, Taivalsaari 1993). Oleellisessa perinnässä olio perii esi-isiensä piirteet ja pyrkii toimimaan näiden piirteiden mukaisesti. Satunnaisessa perinnässä keskitytään lähinnä toteutuksen tai koodin uudelleenkäyttöön (Sakkinen 1992).

Olennaisella perimisellä tarkoitetaan, että jälkeläisluokalla on on-suhde esi-isään. Esimerkiksi Auto on Kulkuväline –tapauksessa, Auto voi periä kulkuvälineelle ominaisia piirteitä, kuten nopeus ja paino. Satunnaisen perinnän tapauksessa Auto voisi periä Eläimeltä paino-piirteen, mutta ilmeisistä syistä johtuen tällainen ratkaisu ei ole toimiva.

Yksi olennaiseen perintään liittyvä keskeinen seikka on rajapinnan periminen. *Rajapinnan periminen* tarkoittaa sitä, että luokka perii itselleen rajapinnan, jonka kautta luokan piirteitä voidaan käyttää. Rajapinnan perinnässä luokka perii esi-isiltään vain kuvauksen siitä, minkälaisia palveluita luokan on toteutettava. Yleensä näitä palveluja ei ole toteutettu esi-isäluokissa, joten esi-isäluokkia kutsutaan *abstrakteiksi* luokiksi. Rajapinnan perimistä ei tule sekoittaa luokkaperintään. Luokkaperintä määrittää olion toteutuksen, kun taas rajapinnan perintä (tai alityypitys) määrittää, milloin oliota voidaan käyttää toisen olion tilalla.

Alityypitys tarkoittaa, että jokaiseen kohtaan, missä esiintyy tietyn luokan ilmentymä, voidaan sijoittaa myös tämän luokan jälkeläisluokan ilmentymä (Koskimies 1997). Esimerkiksi kuvion 3 D-luokkaa, jonka esi-isänä on A-luokka, voidaan käyttää samoissa paikoissa kuin A-luokkaakin. Tämä tietysti edellyttää sen, että luokka D toteuttaa samat ominaisuudet kuin luokka A eikä ole muuttanut piirteitä niin, että niiden käyttö aiheuttaisi ristiriitoja ohjelmassa. Olio voi siis olla useaa eri tyyppiä, ja eri luokat voivat olla samaa tyyppiä.

Alityypittäminen liittyy läheisesti monimuotoisuuteen. *Monimuotoisuutta* ilmenee, kun ohjelmassa esiintyvä nimetty kohde voi tarkoittaa useita erilaisia asioita (Koskimies

1997). Esimerkiksi kuvion 3 tapauksessa A-luokan paikan voi ottaa joko B-, C- tai D-luokka. Jokainen näistä luokista voi toteuttaa A-luokan operaation omalla tavallaan. Kun ohjelmassa kutsutaan A-luokan tyyppistä oliota, voi tämä luokka olla joko A-, B-, C- tai D-luokka. Näin ollen etukäteen ei voida sanoa, minkä luokan operaatio toteuttaa tehdyn kutsun.

Koska olio voi periä esi-isäluokiltaan piirteitä ja näin ollen myös operaatioita, ei aina voida tietää tarkasti, mihin luokkaan operaatiokutsu viittaa. Näin ollen vasta ohjelman suorituksen aikana voidaan tarkasti tietää, mihin luokkaan kutsu viittaa. Tätä kutsutaan *myöhäiseksi sidonnaksi*. Esimerkiksi kuviossa 3 olevalle D-luokalle voidaan antaa operaatiokutsu a, joka on peritty A-luokalta. On mahdollista, että D-luokka itse ei ole operaatiota määritellyt, jolloin ohjelma tarkistaa, onko se toteutettu joissakin sen esi-isäluokista.

Tavallisesti perinnän tarkoituksena on periä abstrakteimmilta tai yleisemmiltä esi-isäluokilta. Tällaisessa tapauksessa esi-isäluokat voivat olla abstraktejakin. Moniperintä mahdollistaa myös ns. sekoitusperiytymisen (Sakkinen 1992, Taivalsaari 1993, Bracha ym. 1990). *Sekoitusperiytymisessä* (mixin-inheritance) erityinen *sekoiteluokka* (mixin-class) sisältää jälkeläisluokkaan tehtävän muutoksen. Syntaktisesti sekoiteluokka ei eroa tavallisesta luokasta, mutta sen käyttötapa on erilainen. Tällainen luokka luodaan ainoastaan siinä tarkoituksessa, että se lisätään jonkun muun luokan yhteyteen. Sekoiteluokasta ei koskaan tehdä itsenäistä oliota.

Oliokeskeisen lähestymistavan keskeisiä käsitteitä ovat siis olio, luokka, attribuutti, operaatio, piirteet, perintä ja monimuotoisuus. Näiden lisäksi ja näille rakentuvia käsitteitä ovat:

1. Abstrahointi: Järjestelmässä esiintyviä komponentteja ja niiden ominaisuuksia kuvataan niiden yhteisten piirteiden avulla, eli piirteitä *luokitellaan*. Luokittelun avulla järjestelmää voidaan kuvata ilman, että kaikkia yksityiskohtia tunnettaisiin. Luokkia voidaan järjestää erilaisiin hierarkioihin, jotka kuvaavat järjestelmän käsitteiden kokonaisuuksia.
2. Kapselointi: Luokkien sisäinen käyttäytyminen piilotetaan sitä käyttäviltä luokilta. Luokka itse tietää oman toteutuksensa, mutta ulospäin tämä toteutus ei näy. Muut luokat joutuvat käyttämään niitä palveluja, jotka luokka tarjoaa. Näitä tarjottavia palveluja kutsutaan myös luokan rajapinnaksi. Kapseloinnilla erotetaan toisistaan luokan käyttäytyminen ja toteutus.
3. Tiedon ja käyttäytymisen yhdistäminen: Luokkaan kuuluu tietorakenne sekä tämän tietorakenteen käsittely. Tällaista kokonaisuutta kutsutaan olioksi.

4. Tiukka sidonta ja löyhä kytkentä: Luokkien sisäiseen rakenteeseen kuuluu periaate tiukasta sidonnasta. Tämä tarkoittaa, että luokkaan tulisi kuulua vain keskenään sidoksissa olevia käsitteitä. Löyhä kytkentä -periaate tarkoittaa, että luokkien välillä tulisi olla mahdollisimman vähän kutsusuhteita.

Edellä mainittu näkemys ei ole ainoa, joka on tuotettu olioiden eri ominaisuuksista, mutta listassa esitetyt asiat tulevat useimmiten vastaan oliokirjallisuutta luettaessa. Näiden asioiden toteutus löytyy jokaisesta oliokielestä, taikka niiden on otettava näihin seikkoihin kantaa tavalla tai toisella.

Olioiden piirteiden kuvaamiseen on olemassa useita notaatioita. On myös olemassa tekniikkoja näiden piirteiden kuvaamiseen ja huomioon ottamiseen. Kuitenkin vielä tarvitaan menetelmä, joka sitoo kaikki nämä notaatiot ja tekniikat ja saa ne toimimaan yhdessä

3.2 Oliomenetelmät

Menetelmän tarkoituksena on sitoa yhteen notaatiot ja tekniikat. Oliomenetelmien tehtävänä on siis sitoa yhteen olioihin liittyvät notaatiot ja tekniikat. Erilaisia oliomallintamismenetelmiä on olemassa kymmeniä (mm. Jacobson ym. 1992, Rumbaugh ym. 1991, Booch 1994, Jaaksi 1997, Coleman ym. 1994 ja Wirfs-Brock ym. 1990). Jokainen näistä esittää oman notaationsa, tekniikkansa ja menetelmänsä.

Oliosuunnittelumenetelmät ja -tekniikat olettavat, että maailma on tehty olioista, jotka puolestaan sisältävät attribuutteja ja toimintaa, eli piirteitä. Järjestelmän suunnittelussa pyritäänkin ymmärtämään ongelma-alue sekä löytämään siinä esiintyvät oliot. Toinen keskeinen tehtävä suunnittelussa on löytää olioiden väliset suhteet. Olioiden avulla voidaan kuvata todellista toimintaympäristöä paremmin (vrt. Meyer 1988, Booch 1994, Rumbaugh ym. 1991). Perusteluina em. teoksissa käytetään vain, että ihminen havainnoi ympäristöään olioina. Tämä ei kuitenkaan riitä tässä tapauksessa, mikäli haluamme rakentaa viitekehystä olioiden oppimiselle.

Viime vuosina on tapahtunut yhdentymistä eri notaatioiden kesken. Yksi tällainen on Unified Modeling Language (UML) (UML Semantics 1997, UML Notation Guide 1997, UML Summary 1997). Toinen merkittävä kuvaustapa on OPEN (Object-oriented Process, Environment and Notation) (OPEN 1998).

Maailmanlaajuista elinkaarimallia ei kuitenkaan ole odotettavissa, koska ohjelmistonkehitystarpeet ovat tällä hetkellä erittäin erilaisia. Vaikka UML:n kehittäjät luovat yhteistä Objectory-menetelmää, ei se tule käymään kaikkiin ohjelmistonkehitystarpeisiin.

Oliomenetelmien keskeisenä piirteenä on koko ohjelmistotuotantoprosessin iteratiivisuus. Iteratiivisuus näkyy vaiheen sisällä sekä vaiheiden välillä. Iteratiivisuuden avulla voidaan lisätä vähitellen suunnittelijoiden ja käyttäjien ymmärrystä toteutettavasta järjestelmästä. Koko ohjelmaa ei ole tarkoitus toteuttaa yhtenä suurena kokonaisuutena vaan sitä pyritään toteuttamaan pala kerrallaan. Näin saavutetaan Brooksin (1987) mainitsema ohjelmiston kasvattaminen.

Iteratiivisuus vaiheen sisällä tarkoittaa, että esimerkiksi analyysivaiheessa voidaan keskittyä mallintamaan vain keskeisiä seikkoja. Ei haittaa, vaikka jotakin jäisi mallintamatta, koska aina voidaan palata takaisin kesken jääneeseen kohtaan. Iteratiivisuus vaiheiden välillä tarkoittaa, että osa ohjelmistosta voidaan toteuttaa kerrallaan. Eli ensin voidaan analysoida, suunnitella, toteuttaa, testata ja ottaa käyttöön vain osa ohjelmasta, vaikka muita osia ei vielä olekaan analysoitu loppuun saakka. Tällaisella vähittäisellä ohjelmiston 'kasvatuksella' saavutetaan parempi ymmärrys koko ohjelmistosta, eikä kaikkea tarvitse päättää ohjelmistotuotantoprosessin alussa.

Vaikka Meyer (1988) väittää, että oliot vain tunnistetaan mallinnettavasta kohteesta ja määritellään, niin asia ei suinkaan ole näin yksioikoinen. Nykyään tunnustetaan, että järjestelmän jakaminen mielekkäisiin olioihin on yksi vaikeimmista tehtävistä oliopohjaisessa suunnittelussa. Varsinkin kun tämä halutaan tehdä hyvin ja hyviä periaatteita noudattaen.

Eri lähestymistavat oliopohjaiseen mallintamiseen käyttävät erilaisia tekniikoita olioiden löytämiseen järjestelmästä. Kohteena olevasta järjestelmästä voidaan kirjoittaa ongelmakuvaukset, etsiä substantiivit ja verbit sekä luoda vastaavat luokat ja operaatiot (Rumbaugh ym. 1991). Kohdejärjestelmästä voidaan kirjoittaa käyttötilanteita (use-cases), joiden pohjalta saadaan toiminnan ydin, jota tarkennetaan suunnitteluprosessin aikana (Jacobson ym. 1992, Jacobson ym. 1995). Erilaisia lähestymistapoja on, ja aina tullaan käymään kiistaa siitä, mikä niistä on paras.

Oliomenetelmillä syntyy järjestelmästä erilaisia kuvauksia, *suunnitelmia*. Keskeisiä tuotettuja kuvauksia ovat luokka- ja dynaamiset mallit (vrt. Rumbaugh ym. 1991, Jacobson ym. 1992, Booch 1994). *Luokkamallit* kuvaavat, kuinka järjestelmän oliot ovat

suhteessa toisiinsa. Luokkien välillä kuvataan yleensä kolmentyylisiä suhteita: perintä-, koostumis- ja yhteistyösuhteita (vrt. Rumbaugh 1991, Koskimies 1997). *Dynaamiset mallit* kuvaavat, kuinka oliot kutsuvat toisiaan taikka keskustelevat keskenään. Dynaamisia malleja ovat mm. tilakoneet ja herätekaaviot.

Monet järjestelmän oliot löytyvät analyysivaiheen luokkamallista (mm. Jaaksi 1997). Analyysivaiheen luokkamallista ei kuitenkaan voida siirtyä suoraan toteutukseen. Analyysivaiheen mallissa on useita olioita, joita ei toteutettavassa järjestelmässä joko ole tai niitä ei siihen toteuteta. Toisaalta analyysivaiheen suunnittelumallista puuttuu toteutusläheisiä luokkia kuten esimerkiksi säiliöluokat[3] sekä muut alemman tason suunnittelumallit (vrt. Zimmerin luokittelu kohta 3.3). Mikäli tosimaailmaa pyritään mallintamaan liian tarkasti, on mahdollisuutena päätyä malliin, joka vanhenee nopeasti. Mallintamisen aikana löydetyt abstraktiot antavat hyvän lähtökohdan luoda pitkäaikaisen mallin, jota voidaan uudelleenkäyttää muissa sovelluksissa. Abstraktiot luovat suunnitelmalle joustavuutta.

Suunnittelumallien yksi keskeinen tavoite on antaa suunnittelijalle mahdollisuus löytää ja kuvata sellaiset abstraktiot, jotka eivät ole itsestään selviä (Gamma ym. 1995). Esimerkiksi oliot, jotka esittävät prosessia tai algoritmia, eivät esiinny luonnossa, mutta ne ovat keskeinen osa joustavaa suunnitelmaa. Yleensä tämänkaltaisia olioita ei löydetä analyysivaiheessa, ei edes suunnitteluvaiheen alussa, vaan ne löydetään vasta, kun suunnitelmasta yritetään tehdä joustavaa. Suunnittelumallit kuvaavat tämänkaltaisia ratkaisuja, joten ne mahdollistavat lyhyemmät iteraatiokierrokset ja antavat suunnitelmille jo käyttökelpoisiksi todettuja ratkaisuja.

3.3 Oliosuunnittelussa huomioon otettavia seikkoja

Rajapinnat ovat oliopohjaisen suunnittelun peruselementtejä. Oliot näkyvät ulospäin ainoastaan rajapintojensa kautta. Oliota ei voi pyytää suorittamaan mitään ilman, että se tehdään määritellyn rajapinnan kautta. Rajapinta ei kuvaa mitään olion rakenteesta taikka siitä, kuinka olio suorittaa asiakkaalta tulevan pyynnön, vaan se määrittelee pyynnön muodon ja sen, mitä pyyntö palauttaa. Oliot, joilla on samanlainen rajapinta, voivat toteuttaa tulevat pyynnöt eri tavoin (vrt. dynaaminen sidonta ja monimuotoisuus). Olioiden rajapinnan kuvaus tehdään luokassa, eli luokka kuvaa olioiden rajapinnan.

[3] Säiliö on olioterminologiassa yleinen nimitys tietorakenteelle, joka säilyttää useita elementtejä. Esimerkkeinä säiliöistä ovat mm. listat, pinot ja jonot. (Buschmann ym. 1997.)

Suunnittelumallit auttavat määrittelemään rajapintoja tunnistamalla niiden tärkeimmät elementit sekä sen, minkälaista tietoa rajapinnan kautta voidaan välittää. Suunnittelumallit määrittävät myös rajapintojen välisiä suhteita. Ne vaativat toisinaan tiettyjen luokkien välisten rajapintojen samankaltaisuutta tai ne asettavat rajoitteita rajapinnan toteutukselle. Näin voidaan esimerkiksi toiminnallisuutta kuvata yliluokissa, eikä asiakkaiden tarvitse tuntea aliluokkien rajapintoja, vaan voivat käyttää aliluokkia yliluokkien määrittelemien rajapintojen kautta.

Abstraktin luokan päätarkoitus on määritellä yleinen rajapinta sen aliluokille (Gamma ym. 1995). Abstrakti luokka voi jättää yhden, osan tai kaikkien operaatioiden toteuttamisen aliluokilleen. Näin ollen abstraktista luokasta ei voi luoda oliota. Operaatio, jonka abstrakti luokka määrittelee muttei toteuta, kutsutaan abstraktiksi operaatioksi. Abstraktin luokan vastakohta on konkreetti luokka.

Aliluokat voivat tarkentaa tai määritellä uudelleen yliluokan käyttäytymistä. Tällainen *korvaus* (overriding) antaa aliluokalle mahdollisuuden toteuttaa pyyntö omalla tavallaan niin, ettei sen tarvitse välittää yliluokan vastaavasta toteutuksesta. Luokkaperintä antaa mahdollisuuden määritellä luokkia laajentamalla muita luokkia. Tämä tekee helpoksi määritellä luokkaperheitä, joilla on samankaltaista toimintaa.

Kuitenkin toteutuksen perintä mahdollistaa vain osan uudelleenkäytöstä. Rajapinnan perintä on toinen kokonaisuus uudelleenkäytössä. Monimuotoisuus perustuu juuri tähän periytymisen tyyliin. Vaikka olioilla olisi samankaltainen rajapinta ja ne olisivat samaa tyyppiä, niin niiden operaatioiden toteutus voi olla täysin erilainen. Kun perintää käytetään huolellisesti, kaikki abstraktista luokasta johdetut luokat jakavat abstraktin luokan rajapinnan.

Rajapinnan perinnällä saadaan kaksi merkittävää etua (Gamma ym. 1995).

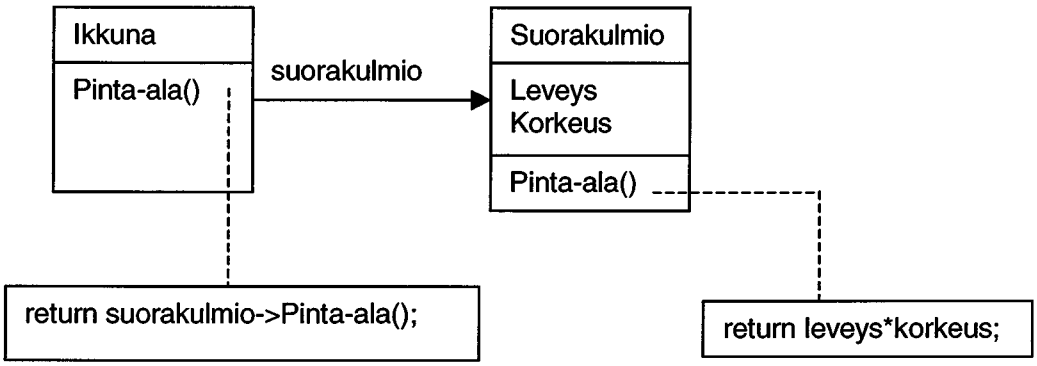
1. Asiakkaat eivät tiedä käyttämänsä olion tarkkaa tyyppiä, mutta tällä ei ole merkitystä, koska oliot toteuttavat asiakkaan odotusten mukaisen rajapinnan.
2. Asiakkaat eivät tiedä, mitkä luokat toteuttavat nämä oliot. Asiakkaat tietävät vain abstraktin luokan määrittelemän rajapinnan.

Perinnän lisäksi oliosuunnittelussa voidaan käyttää delegointia. *Delegoinnissa* kaksi oliota ovat mukana pyynnön toteuttamisessa: pyynnön vastaanottava olio delegoi operaation toteuttajalleen. Delegoinnilla voidaan uudelleenkäyttöä tehdä koostamisella yhtä hyvin kuin perinnällä. Tämä on analogista sen kanssa, että aliluokka välittää yliluokal-

leen sellaisen pyynnön, jota ei osaa suorittaa. (Koskimies 1997, Gamma 1995). Tietyssä teknisessä mielessä delegointi voidaan toteuttaa myös periytymisen avulla (Stein 1987).

Kuviossa 4 on esimerkki, kuinka Ikkuna-luokka delegoi pinta-alansa laskemisen Suorakulmio-luokalle. Suorakulmio-luokka palauttaa lasketun pinta-alan. Jos joku haluaisi tehdä pyöreitä ikkunoita, silloin Ikkuna-luokan tarvitsee vain delegoida operaatio esimerkiksi Ympyrä-luokalle. Nuolenpää kuviossa 4 tarkoittaa, että luokka pitää viitettä toisen luokan esiintymään. Viitteellä on tässä nimi 'suorakulmio'.

Suurin etu delegoinnista on, että sen avulla voidaan ajon aikana helposti koostaa käyttäytymistä ja muuttaa koosteolioita. Delegoinnin heikkoutena on, että se tekee ohjelmiston vaikeammaksi ymmärtää kuin staattisesti määritelty ohjelmisto. Ohjelmiston suoritusnopeus laskee myös. Delegointi on hyvä vaihtoehto silloin, kun se ei ainakaan merkittävästi monimutkaista ohjelmiston rakennetta. (vrt. Koskimies 1997, Gamma ym. 1995.)



Kuvio 4. Delegointiesimerkki (Gamma ym. 1995)

Oliokeskeisyyteen liittyy muitakin periaatteita kuin edellä mainitut rajapinnan perintä ja delegointi. Tällaisia ovat mm. Preen (1995) esittämät seuraavat säännöt:

1. Luokkien ja olioiden välinen heikko kytkentä. Tämä periaate rajoittaa olioiden välistä kommunikaatiota. Olioiden tulisi vaihtaa tietoa niin vähän kuin mahdollista. Luokkien välinen heikko kytkentä vähentää koko järjestelmän kompleksisuutta.

2. Vahva sidonta luokkien sisällä. Tämä tarkoittaa, että operaatioiden ja tiedon täytyy olla tiukasti kytkettyjä. Luokan sisäinen voimakas koheesio saavutetaan usein määrittelemällä yhtä luokkaa koskevat operaatiot käsittelemään vain yhden abstraktiotason att-

ribuutteja. Luokat, joilla on heikko sisäinen kytkentä, tulisi jakaa riippumattomiin luokkiin.

3. Minimaaliset ja käytännölliset luokkarajapinnat. Samankaltaisia operaatioita, jotka toteuttavat saman palvelun, tulisi välttää.

4. Testattavuus. Luokan oikeellisuus tulisi olla testattavissa ilman että tiedetään, missä kontekstissa luokkaa tullaan käyttämään. Heikko kytkentä luokkien välillä ja minimaaliset rajapinnat luokkaan yksinkertaistavat testaamista.

Yksi vielä mainittava sääntö olio-ohjelmoinnille on ns. Demeterin laki (Lieberherr ym. 1988, Holland 1992). Demeterin laki määrittelee hyvän olio-ohjelmoinnin tavat. Sakkinen (1992) on tiivistänyt lain seuraavasti: "Luokan operaatioiden ei tulisi riippua millään tavoin muiden luokkien rakenteesta, vain oman luokan välittömästä rakenteesta. Lisäksi, jokaisen operaation tulisi lähettää viestejä vain hyvin rajoitettuun määrään luokkia." Demeterin lain tekijöiden mukaan ohjelmoijien kannattaa lisäksi minimoida koodin kopiointi, operaation argumenttien määrä sekä luokan operaatioiden määrä (Sakkinen 1992).

3.4 Yhteenveto

Tässä luvussa käytiin läpi oliokeskeisyyteen liittyvää sanastoa. Oliokeskeisyyden 'uusiksi' asioiksi todettiin perintä ja siihen liittyvät käsitteet monimuotoisuus ja myöhäinen sidonta. Perinnällä on kaksi keskeistä muotoa: luokka- ja rajapintaperintä. Luokkaperinnällä tarkoitetaan toteutuksen perintää. Rajapintaperinnän avulla voidaan uudelleenkäyttötapaa muuttaa, tällöin asiakkaiden ei tarvitse tuntea käyttämänsä olion tarkkaa tyyppiä, eivätkä asiakkaat tiedä, mitkä luokat toteuttavat heidän pyyntönsä. Asiakkaat tietävät vain, minkä rajapinnan mukaisesti kutsuttu olio toimii.

Perinnän lisäksi oliokeskeisyydessä käytetään delegointia. Delegoinnin tarkoituksena on mahdollistaa olioiden ajon aikainen muuttaminen. Näin saavutetaan olioille dynaamisesti tapahtuva toiminnan muuttuminen.

4 SUUNNITTELUMALLIT JA NIIDEN LUOKITTELU

Edellisessä luvussa määriteltiin oliokeskeiseen lähestymistapaan liittyviä käsitteitä. Näihin käsitteisiin tulee jokaisen oliosuuntautuneen ohjelmointikielen ottaa kantaa. Oliomenetelmien avulla suunniteltu järjestelmä voidaan sitten toteuttaa olio-ohjelmointikielillä. Pelkästään oliokeskeisyys ei ole tarpeeksi. Perinnän ja uudelleenkäytön edut oliolähestymistavassa aiheuttavat ristiriitaisia tunteita (vrt. Taivalsaari 1993). Tarvitaan jotain oliokeskeisyyden syntaksin yläpuolelle menevää. Yksi tällainen yläpuolelle menevä käsite on suunnittelumallit.

Tässä luvussa esitetään suunnittelumallit ja arvioidaan niiden luokitteluja sekä valitaan ohjelmistotuotannon kannalta toimivin luokittelu. Aluksi kerrotaan suunnittelumallien taustaa sekä ajattelutapaa, joka liittyy niihin. Tämän jälkeen esitetään esimerkkinä yksi Gamman ym. (1995) suunnittelumalleista. Tätä seuraa seitsemän erilaista luokittelua suunnittelumalleille, joista valitaan ohjelmistotuotannon kannalta sopivin luokittelu.

4.1 Suunnittelumallit

Tosiasia on, että kokeneet oliosuunnittelijat tekevät parempia suunnitelmia kuin kokemattomat oliosuunnittelijat. Kokemattomat suunnittelijat kohtaavat jatkuvasti uusia ongelmia, eivätkä tiedä, millaisia ratkaisuja tulisi käyttää missäkin tilanteessa. Kokemuksen kautta huomataan, että tietyt ongelmat ovat toistuvia ja näihin toistuviin ongelmiin käyvät samat ratkaisut. Näitä ratkaisuja kutsutaan suunnittelumalleiksi.

Suunnittelumallit (design patterns) ovat nimettyjä ratkaisuja toistuviin ongelmiin tietyssä kontekstissa (vrt. Gamma ym. 1995, Vlissides 1997). Tässä määritelmässä yhdistyy useita suunnittelumallille oleellisia piirteitä. Ensinnäkin suunnittelumalli on nimetty ratkaisu. Nimeäminen antaa meille käsitteen ja termin ja näitä voidaan käyttää kuvaamaan suurempia kokonaisuuksia. Kun pystymme käyttämään nimeä tietystä ratkaisusta ja siihen liittyvistä ongelmista, saavutamme korkeamman abstraktiotason puhuessamme suunnitelmista.

Toiseksi, suunnittelumalli on ratkaisu toistuviin ongelmiin. Ei ole hyötyä nimetä ja kuvata ratkaisua, joka käy yhteen tai vain muutamaan tapaukseen. Ongelman on oltava toistuva ja se on toistuvasti ratkaistavissa samalla tavalla. Tähän liittyy myös usein

suunnittelumalleihin kytketty ns. kolmen esiintymän sääntö, eli suunnittelumallia on täytynyt käyttää kolmessa eri sovelluksessa ratkaisemaan sama ongelma.

Kolmanneksi, suunnittelumalli liittyy aina tiettyyn kontekstiin. Kontekstilla ymmärretään, että suunnittelumalliin vaikuttavat ympäristön erilaiset vaatimukset, joista suunnittelumallien yhteydessä käytetään nimitystä voimat. *Voimat* (forces) kuvaavat, minkälaisia vaikutuksia suunnittelumalleilla on järjestelmään ja minkälaisia vaihtokauppoja suunnittelijan on tehtävä, mikäli hän haluaa saavuttaa tietyn ratkaisun.

Jokaisesta suunnittelumallin kuvauksesta täytyy löytyä seuraavat osat: 1) suunnittelumallin nimi, 2) konteksti eli suunnittelutilanne, jossa ongelma esiintyy, 3) ongelman kuvaus, jossa esitetään ongelmassa esiintyvät voimat, 4) ratkaisu, jossa esitetään voimien käyttäytyminen ja mahdolliset vaihtoehtoiset ratkaisutavat, sekä ratkaisun rakenne ja käyttäytyminen.

Suunnittelumallien kuvaamiseksi on vakiintumassa ns. Gamma-muoto. Tämä muoto tulee Gamman ym. (1995) kirjasta, jossa kirjoittajat esittelevät suunnittelumalleja. Suunnittelumallien formaali kuvaaminen tekee mahdolliseksi tallentaa sitä tietoa, jota ohjelmistojen suunnittelijat keräävät vuosien varrella. Gamman ym. kirjassa suunnittelumallien kuvaamisessa on käytetty 12 kohtaa. Kaikkien suunnittelumallien kuvaaminen samalla tavalla auttaa käyttäjää lukemaan suunnittelumalleja. Kuvaustapa auttaa löytämään saman tiedon samasta paikasta ja näin helpottaa halutun tiedon löytymistä.

Yksittäinen suunnittelumalli ei ole eristyksissä ympäristöstään. Jokaisella suunnittelumallilla on yhteyksiä muihin malleihin. Suunnittelumalli on riippuvainen niistä toisista suunnittelumalleista, joista se rakentuu, joiden kanssa se kommunikoi ja joissa se on osana kokonaisuutta (vrt. Alexander ym. 1977, Alexander 1979). Suunnittelumalleista pitäisikin pyrkiä tekemään mallijärjestelmiä (Buschmann ym. 1997). *Mallijärjestelmässä* on suunnittelumallit järjestetty niin, että niiden avulla voidaan systemaattisesti suunnitella jokin kokonaisuus (Koskimies 1997).

Gamma ym. (1995) mainitsevat, että heidän suunnittelumallinsa ovat ns. mikrosovelluskehyksiä (vrt. myös Pree 1995). *Sovellukehys* (application framework) tarkoittaa olio-ohjelmoinnin yhteydessä kiinteästi toisiinsa liittyvien luokkien kokoelmaa, josta sopivasti täydentämällä ja kokoamalla saadaan valmis sovellus tai sen merkittävä osa (Koskimies 1997). Sovelluskehys määrittelee sovelluksen käyttöliittymän, käyttäytymisen, ja toimintaympäristön niin, että ohjelmoijan tarvitsee määritellä vain sovelluskohtaiset asiat (Weinand ym. 1989). Lewis (1995) mainitsee, että sovelluskehykset saavat

merkittävästi huomiota oliokeskeisessä suunnittelussa ja ohjelmoinnissa lähivuosien aikana.

4.2 Esimerkki suunnittelumalli

Seuraava esimerkki on otettu Koskimiehen (1997) Pienestä oliokirjasta (s. 161 – 162). Alun perin suunnittelumalli esitettiin Gamman ym. (1995) kirjassa (s. 163 –173). Ainoana muutoksena Koskimiehen esimerkkiin on rakennekuvauksen suomennos. Koskimiehen esimerkistä puuttuvat sellaiset oleelliset kohdat kuin esimerkkitoteutuksen vaiheet ja esimerkkikoodi. Kooste (Composite) käyttää ratkaisussaan rajapintaluokaa, joka tarjoaa yhtenäisen rajapinnan koostetta käyttävälle oliolle. Käyttäjän ei tarvitse välittää siitä, onko kyseessä koosteolio vai primitiiviolio.

Kooste (Composite)
Rakennemalli

Tarkoitus

Mallin avulla voidaan olio esittää hierarkkisesti toisista olioista koostuvana siten, että koosteolioita ja niiden osaolioita voidaan käsitellä samalla tavalla.

Motivointi

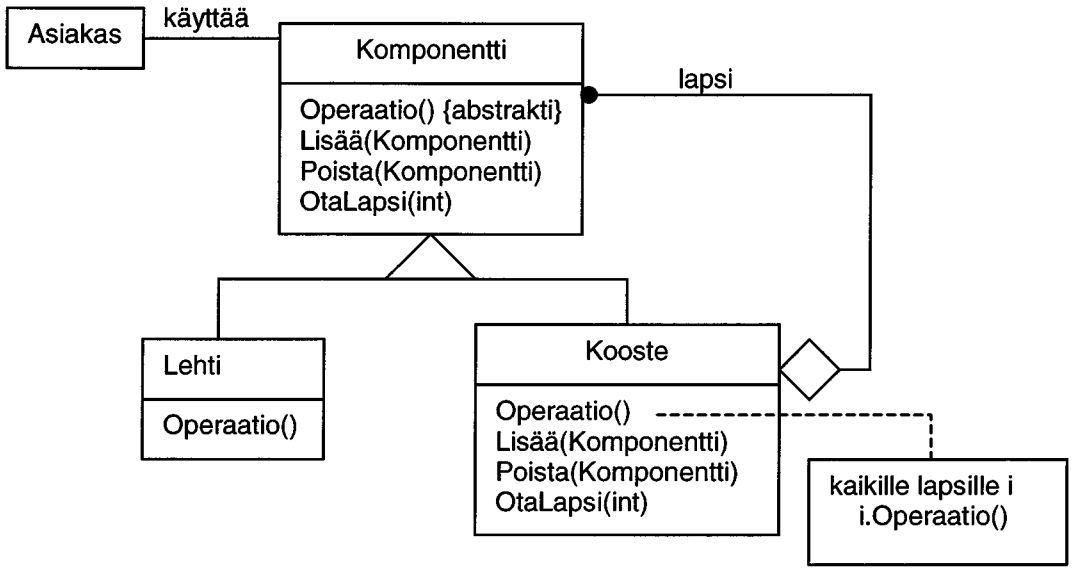
Monissa sovelluksissa tarvitaan olioita, jotka muodostuvat hierarkkisesti ja rekursiivisesti toisista olioista (ts. olion komponenttina voi olla samanlainen olio). Tyypillinen sovellus on piirtoväline, jonka avulla voidaan koota kuvioita pienemmistä kuvioista. Suoraviivainen ratkaisu olisi antaa kullekin primitiivioliolajille oma luokka ja näiden lisäksi sopiva säiliöluokka, joka kuvaa koosteoliot: säiliöolio sisältää komponenttiolionsa. Tällainen ratkaisu on kuitenkin hankala, koska silloin säiliöolioita ja primitiivioliota täytyy käsitellä eri tavoin. Mallin olisi sallittava rekursiivisen oliorakenteen käsittely ilman, että tällaista eroa täytyy tehdä. Ratkaisuna on antaa abstrakti luokka, joka kuvaa sekä koosteoliot että primitiivioliot.

Soveltuvuus

Mallia voidaan käyttää, kun

a) halutaan esittää hierarkkisesti osaolioista koostuvia olioita;
b) halutaan, että hierarkkista oliorakennetta käyttävien asiakkaiden ei tarvitse välittää primitiivisten olioiden ja koosteolioiden erosta.

Rakenne



Osallistujat

Komponentti: määrittelee yleisen liittymän kaikkiin hierarkkisen rakenteen olioihin; antaa oletustoteutuksen kaikkien näiden olioiden operaatioille; määrittelee liittymän operaatioille, joilla päästään käsiksi osaolioihin

Lehti: kuvaa primitiivioliot (so. oliot, joilla ei ole osaolioita)

Kooste: kuvaa koosteoliot (so. oliot, joilla on osaolioita); tallettaa viitteet osaolioihin; antaa toteutuksen osaolioihin liittyville luokan Komponentti operaatioille.

Asiakas: käsittelee oliorakennetta luokan Komponentti tarjoaman liittymän kautta.

Seuraukset

Missä tahansa asiakas edellyttää primitiiviolioita, se pystyy käsittelemään myös koosteolioita.

Asiakkaan koodi yksinkertaistuu, koska erottelua primitiiviolion ja koosteolion välillä ei tarvitse tehdä.

Helpottaa uudentyyppisten kooste- ja primitiiviolioiden lisäämistä.

Saattaa yleistää liiaksi: malli ei tue esimerkiksi rakennetta, jossa edellytetään, että koosteolio koostuu vain tietyistä tai tietyntyyppisistä olioista.

4.3 Suunnittelumallien luokittelu

Suunnittelumalleja on olemassa jo satoja. Jotta voitaisiin löytää oikea malli oikeaan paikkaan, on suunnittelumalleja luokiteltava. Luokittelun avulla voidaan löytää myös uusia ja puuttuvia suunnittelumalleja sekä helpottaa suunnittelumallien oppimista (Gamma ym. 1995) (vrt. Zimmer 1995a). Mikäli ohjelmistokehittäjien täytyisi lukea, ymmärtää ja muistaa jokainen suunnittelumalli, kävisi koko suunnittelumallien idea liian raskaaksi (Buschmann ym. 1996). Kukaan ei pysty hallitsemaan satoja suunnittelumalleja, vaan tarvitaan valintaa ohjaava ja tukeva luokittelu. Luokittelulle voidaan asettaa seuraavia kriteerejä:

1. Luokittelun on oltava yksinkertainen ja helppo oppia.
2. Luokittelu rakentuu muutamien keskeisten luokittelukriteerien varaan.
3. Jokaisen luokittelukriteerin tulee heijastaa suunnittelumallin luonnollisia ominaisuuksia, esimerkiksi ratkaistavia ongelmia, eikä kriteerejä siitä, mikä malli kuuluu luokittelun piiriin.
4. Luokittelun tulee antaa tiekartta, joka johtaa käyttäjät löytämään potentiaaliset suunnittelumallit, eikä jäykkää ajo-ohjetta, joka antaa yksityiskohtaiset ohjeet oikean mallin löytämiselle.
5. Luokittelun tulee olla avoin uusille suunnittelumalleille ilman, että olemassa olevia kriteerejä pitää muuttaa.
6. Luokituksen tulee tukea ja olla hyödyllinen käyttötarkoitukseensa.

Suunnittelumallit vaihtelevat tarkoitukseltaan ja abstraktiotasoltaan. Niitä käytetään suunnittelun eri vaiheissa ratkaisemaan toistuvia ongelmia. Suunnittelumalleilla on myös erilaisia suhteita keskenään. Jotkin mallit käyttävät toisia malleja rakennuspalikoinaan ja toiset käyttävät muita malleja apunaan. Yhdessä suunnittelumallit ovat suurena apuna ohjelmistotuotannossa ja hyvällä luokittelulla ne saadaan auttamaan koko ohjelmistontuotanto prosessia.

Seuraavaksi esitellään seitsemän erilaista luokittelua ja lopuksi arvioidaan luokittelujen soveltuvuutta ohjelmistotuotantoon. Keskeisinä kriteereinä soveltuvuudesta ovat, kuinka luokittelu tukee ohjelmistotuotannon eri vaiheita ja kuinka helposti suunnittelumalli voidaan löytää ohjelmistotuotannossa eteen tulevien ongelmien ratkaisuun.

4.4 Gamman luokittelu

Gamma ym. (1995) ovat luokitelleet omat suunnittelumallinsa tarkoituksen (purpose) ja kohteen (scope) perusteella (taulukko 1). Tarkoituksen mukaan suunnittelumallit voidaan jakaa kolmeen ryhmään: luontimallit, rakennemallit ja käyttäytymismallit. Kohde tarkoittaa, koskeeko suunnittelumalli oliota vai luokkaa.

Tarkoituskriteeri kertoo, mitä varten suunnittelumalli on olemassa. Luontimallit käsittelevät olion luomisprosessia. Rakennemallit käsittelevät luokkien tai olioiden rakennetta. Käyttäytymismallit kertovat, kuinka luokat tai oliot käyttäytyvät keskenään ja kuinka niiden kesken jaetaan vastuuta. Gamman ym. (1995) tarkoituksen mukainen luokittelu on tehty karkealla tasolla. Suunnittelumallimäärien kasvaessa voi oikean suunnittelumallin löytäminen osoittautua hankalaksi. Tämän kriteerin mukaan tapahtuva luokittelu on suunnittelumallin toiminta-alan mukaan tapahtuvaa luokittelua.

Kohdekriteeri kertoo, vaikuttaako suunnittelumalli olioihin vai luokkiin. Luokkakohtaiset mallit käsittelevät enimmäkseen luokkien ja aliluokkien välisiä suhteita. Nämä suhteet toteutetaan perinnän avulla, joten ne ovat luonteeltaan staattisia, ja suhteet luodaan käännöksen aikana. Oliokohtaiset mallit käsittelevät olioiden suhteita, joita voidaan vaihtaa suorituksen aikana, ja ne ovat näin ollen dynaamisempia. Lähes kaikki suunnittelumallit käyttävät perintää jossakin määrin. Niinpä ainoastaan luokkakohtaiset suunnittelumallit keskittyvät luokkien välisiin suhteisiin. Suurin osa Gamman ym. (1995) suunnittelumalleista koskee olioita.

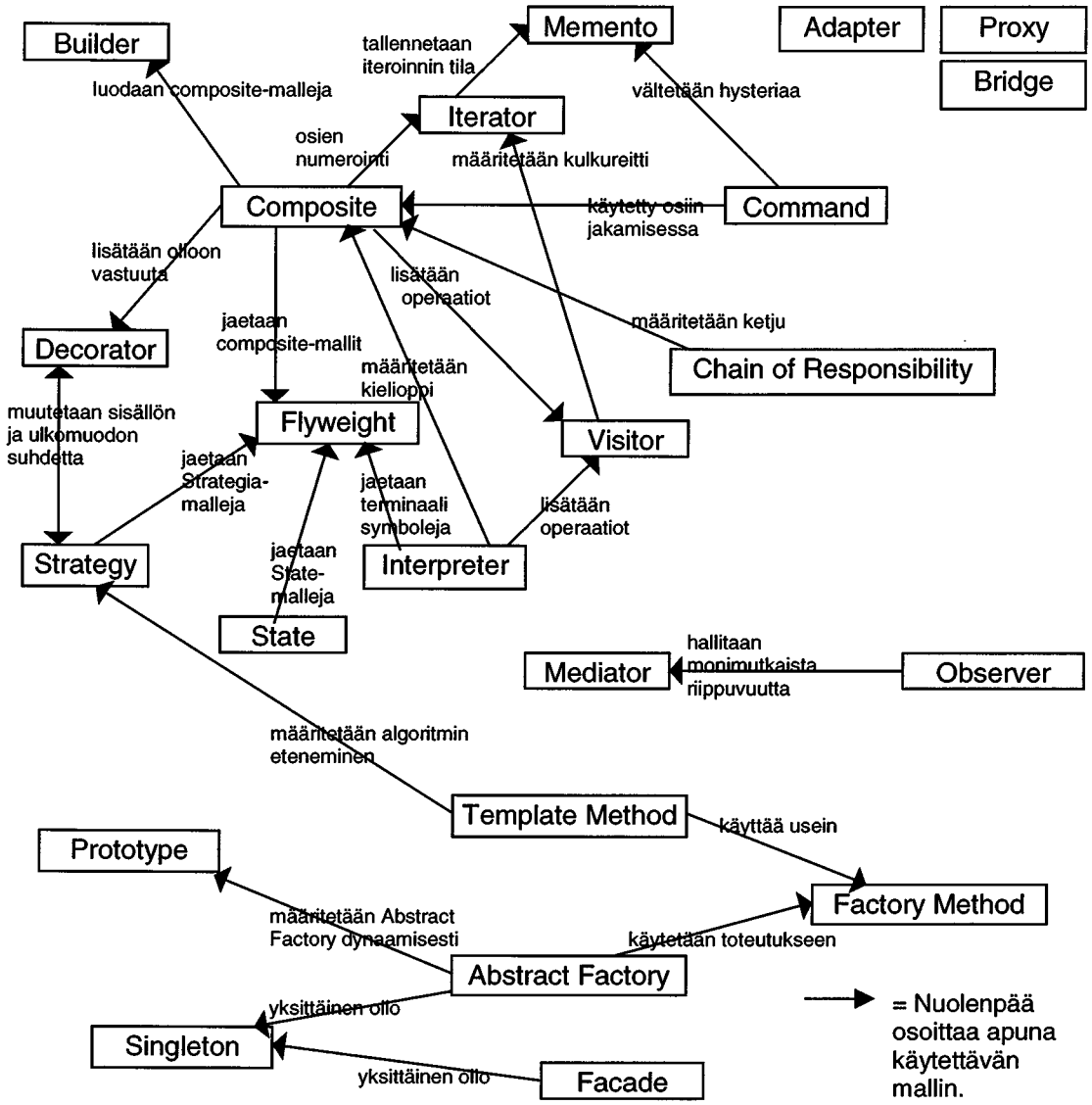
Tarkoitus / Kohde	Luontimalli	Rakennemalli	Käyttäytymismalli
Luokka	Factory Method	Adapter (class)	Interpreter Template Method
Olio	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Taulukko 1. Gamman ym. (1995) suunnittelumallien luokittelu.

Gamman ym. luokittelu on huono. ECOOP'97 konferenssin puheessaan Erich Gamma totesi tämän myös itse. Luokittelu on huono, koska se liittyy liikaa olioparadigman mukaiseen jaotteluun. Kohdekriteeri on suoraan olioparadigmaan liittyvä jaottelu. Toisaalta voidaan nähdä, että myös tarkoituskriteeri on olioparadigmaan liittyvä eli se kertoo, kuinka olio luodaan, kuinka olio rakentuu ja kuinka olio käyttäytyy. Tarkoituskriteeri kertoo kuitenkin, milloin suunnittelumallia käytetään ja millaiseen tarkoitukseen sitä käytetään. Näin ollen tarkoituskriteeri kuitenkin luokittelee enemmän suunnittelumalleja niiden käyttökohteen mukaan.

Toinen huono puoli Gamman ym. luokittelussa on luokittelun karkeus. Mallit jaetaan käytännössä kolmeen osaan: luonti-, rakenne- ja käyttäytymismalleihin, ja osat kertovat vain millaisessa olioparadigman mukaisessa tilanteessa mallia voidaan käyttää. Tällainen jaottelu onnistuu, kun mallit tunnetaan lähes ulkoa ja niiden määrä pysyy muutamissa kymmenissä, mutta määrän kasvaessa ei käyttökelpoisia malleja enää löydetä.

Gamma ym. kirjoittavat, että oliomalleja voidaan jakaa myös sen mukaan, kuinka suunnittelumalleja käytetään keskenään tai ovatko ne vaihtoehtoisia (kts. kuvio 5). Suunnittelumallien välisten suhteiden voidaan katsoa olevan myös tarkoitusta koskeva luokittelua.

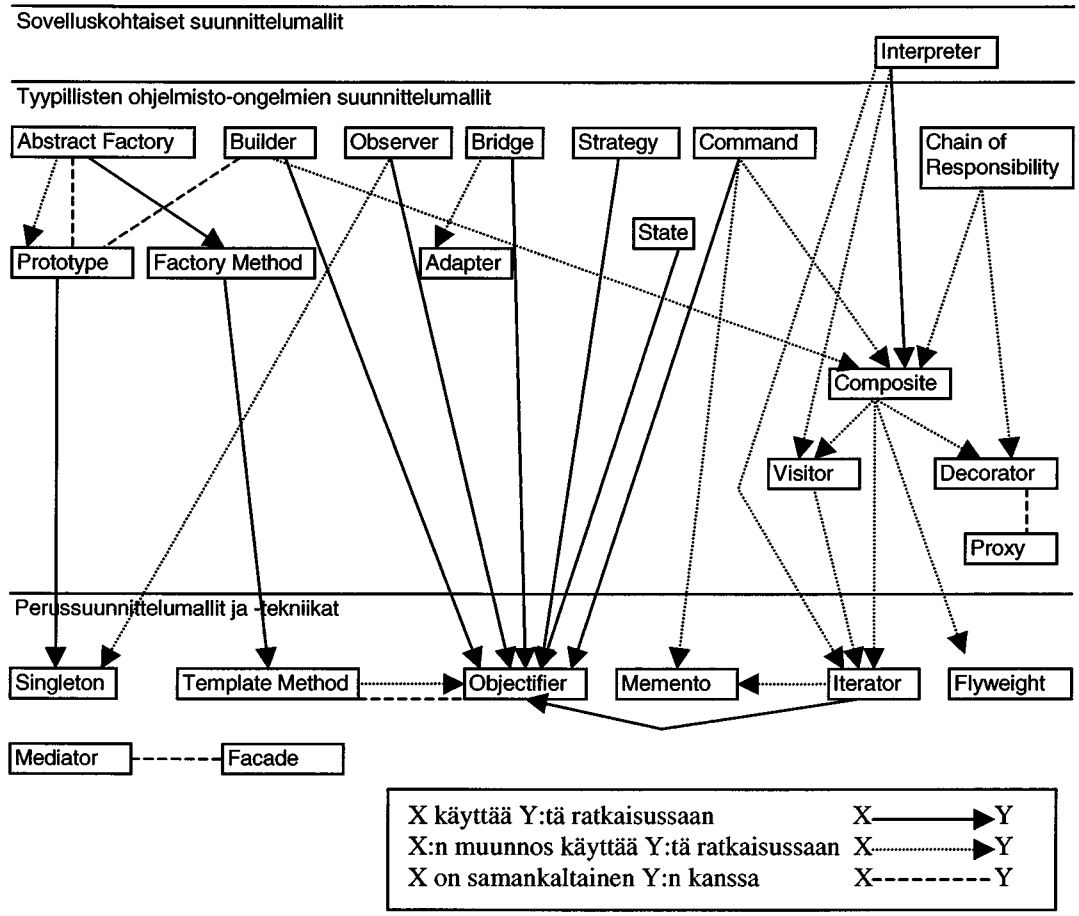


Kuvio 5. Suunnittelumallien väliset suhteet (Gamma ym. 1995)

4.5 Zimmerin luokittelu

Zimmer (1995a) on tehnyt luokittelunsa Gamman ym. (1995) suhteisiin perustuvan luokittelun (kuvio 5) pohjalta. Zimmer selvitti, kuinka suunnittelumallit käyttävät toisiaan, ja tätä kautta niiden kutsu- ja käyttösuhteita. Hän huomasi, että suunnittelumallit ovat luonteeltaan hierarkkisia: alemman tason suunnittelumalleja käyttävät hyväkseen hierarkiassa korkeammalla olevat. Toinen merkittävä Zimmerin huomio on, että Gamman ym. suunnittelumallien perusteella oli tunnistettavissa yksi suunnittelumalli lisää, Objectifier.

Zimmerin luokittelu jakaa suunnittelumallit kolmeen tasoon: perussuunnittelumallit ja -tekniikat, tyypillisten ohjelmisto-ongelmien suunnittelumallit ja sovelluskohtaiset suunnittelumallit (kuvio 6). Perussuunnittelumallit ja -tekniikat esiintyvät yhä uudelleen oliopohjaisissa järjestelmissä. Ohjelmiston rakentamisessa näitä suunnittelumalleja voisi pitää jopa suunnittelutekniikkoina eikä suunnittelumalleina. Nämä suunnittelumallit käyvät laaja-alaisesti erilaisiin ja erityyppisiin sovelluksiin.



Kuvio 6. Zimmerin (1995a) luokittelu

Ohjelmisto-ongelmiin liittyvät suunnittelumallit ovat konkreettisempia ratkaisuja yleisesti esiintyviin suunnitteluongelmiin. Esimerkiksi Builder, Prototype ja Abstract Factory (Gamma ym. 1995) käsittelevät olioiden luontiin liittyviä ongelmia. Kuitenkaan tämän tason mallit eivät ole sovelluskohtaisia vaan liittyvät pikemminkin jokaisessa sovelluksessa vastaan tulevien perusongelmien ratkaisuun.

Sovelluskohtaiset suunnittelumallit ovat kaikkein sovellussidonnaisimpia. Ne käyvät yleensä yhteen tai muutamaan sovelluskohteeseen. Interpreter-malli (Gamma ym. 1995) kertoo, kuinka jokin tunnettu kielioppi voidaan jäsentää. Nämä suunnittelumallit ovat

lähes sovelluskehyksiä, ja ne eroavat sovelluskehyksistä siinä, ettei näitä suunnittelumalleja ole toteutettu käytännössä. Tälle tasolle suunnittelumalleilla ja mallijärjestelmillä tulee pyrkiä.

Zimmerin luokittelu on yksiulotteinen, eikä se ole riittävä, mikäli suunnittelumalleja on enemmän. Tässä luokittelussa tulee eteen sama ongelma kuin Gamman ym. luokittelussa: suunnittelumallimäärän lisääntyessä ei ole enää mahdollista löytää käyttökelpoisia suunnittelumalleja ja näin ollen koko suunnittelumallien idea vesittyy. Zimmer luokittelee suunnittelumallit semanttisen tason mukaan ja näin ollen suunnittelumalleja voidaan jakaa ohjelmistotuotannon vaiheiden mukaan.

4.6 Buschmannin luokittelu 1996

Buschmannin ym. (1996) tekemä luokittelu, josta tässä käytetään nimeä Buschmannin luokittelu 1996, jakaantuu kahteen osaan: ohjelmistotuotannon vaihe ja ongelmaluokka (vrt. taulukko 3). Ohjelmistotuotannon vaihe on heidän mielestään perustavin luokittelukriteeri. Tämän kriteerin mukaan jaettuna suunnittelumallit voidaan jakaa arkkitehtuurimalleihin, suunnitteluvaiheen malleihin ja idiomeihin. Tämä luokittelutapa on jako ohjelmistotyön vaiheiden mukaan.

Arkkitehtuurimalleja voidaan käyttää kehitystyön alussa suurilinjaisempaan suunnitteluun. Tällöin määritetään kehitettävän ohjelmiston perusrakenne. Suunnitteluvaiheen mallit ovat käyttökelpoisia, kun tarkennetaan ja laajennetaan ohjelmiston perusarkkitehtuuria, esimerkiksi päätettäessä alijärjestelmien välisiä kommunikointimekanismeja. Suunnitteluvaiheen mallit ovat hyviä myös yksityiskohtaisessa suunnitteluvaiheessa, kun tehdään toteutukseen liittyviä suunnittelupäätöksiä. Idiomeja käytetään toteutusvaiheessa, kun ohjelmistoarkkitehtuuri muunnetaan ohjelmistoksi määrätyllä kielellä.

Toisena kriteerinä Buschmann ym. käyttävät ongelmaluokkaa. Tämä kriteeri luokittelee suunnittelumallit niiden ratkaisemien ongelmien mukaan. Ongelmat nimetään siten, että vastaavanlainen ongelmatilanne voidaan tunnistaa ohjelmistotuotannossa. Näin ollen ongelmaluokittelulla parannetaan suunnittelijoiden mahdollisuutta löytää haluamansa suunnittelumalli nopeasti. Tämän vuoksi he katsovat, että ongelmaluokkaan perustuva luokittelu on erittäin tehokas työkalu luokiteltaessa suunnittelumalleja. Ongelmaluokan perusteella tapahtuva luokittelu on suunnittelumallin tarkoituksen mukaan tapahtuvaa luokittelua. He löytävät omasta luokittelustaan taulukon 2 kaltaisia ongelmaluokkia.

Ongelmaluokka	Selitys
Mudasta rakenteeseen	Suunnittelumallit, jotka auttavat jakamaan koko järjestelmän yhdessä työskenteleviin alitehtäviin.
Hajautetut järjestelmät	Suunnittelumallit, jotka antavat infrastruktuurin järjestelmille, joiden osat on sijoitettu eri prosessoreille tai useisiin alijärjestelmiin ja komponentteihin.
Vuorovaikutteiset järjestelmät	Suunnittelumallit, jotka auttavat rakentamaan ihmisen kanssa kommunikoivia järjestelmiä.
Mukautuvat järjestelmät	Suunnittelumallit, jotka antavat infrastruktuurin kehittyville ja muuttuville sovelluksille, jotka tarvitsevat laajennuksia ja mukautumista.
Luominen	Suunnittelumallit, jotka auttavat luomaan olioita ja rekursiivisia oliorakenteita.
Rakenteellinen jakaminen	Suunnittelumallit, jotka auttavat jakamaan alijärjestelmät ja monimutkaiset komponentit sopiviksi keskenään työskenteleviksi osiksi.
Työn organisointi	Suunnittelumallit, jotka määrittelevät monimutkaisia palveluja tarjoavien komponenttien työskentelyn keskenään.
Käytön kontrolli	Suunnittelumallit, jotka vartioivat ja kontrolloivat komponenttien palvelujen käyttöä.
Palvelun muuntaminen	Suunnittelumallit, jotka auttavat muuttamaan olion tai komponentin käyttäytymistä.
Palvelun laajentaminen	Suunnittelumallit, jotka auttavat lisäämään uusia palveluja dynaamisesti olioon tai oliorakenteeseen.
Hallinta	Suunnittelumallit, jotka käsittelevät kokonaisuutena homogeenisten olioiden varastoja, palveluja ja komponentteja.
Mukautuminen	Suunnittelumallit, jotka auttavat muuntamaan rajapintaa ja tietoa.
Kommunikointi	Suunnittelumallit, jotka auttavat organisoimaan komponenttien välisen kommunikoinnin.
Resurssien hallinta	Suunnittelumallit, jotka auttavat hallitsemaan jaettuja komponentteja ja olioita.

Taulukko 2. Buschmannin luokittelun 1996 ongelmakriteeriluokat.

Ohjelmistokehitystyön vaiheen ja ongelman mukaan luokitellut suunnittelumallit muodostavat kaksiulotteisen taulukon, joka esitetään taulukossa 3.

Buschmannin luokittelu 1996 perustuu toiminta-alan mukaiseen luokitteluun ongelman luokkien mukaisesti. Toinen ulottuvuus, ohjelmistotuotannon vaihe, on tietenkin ohjelmistotuotannon vaiheen mukainen jaottelu. Buschmannin luokittelu 1996 kuvaa hyvin

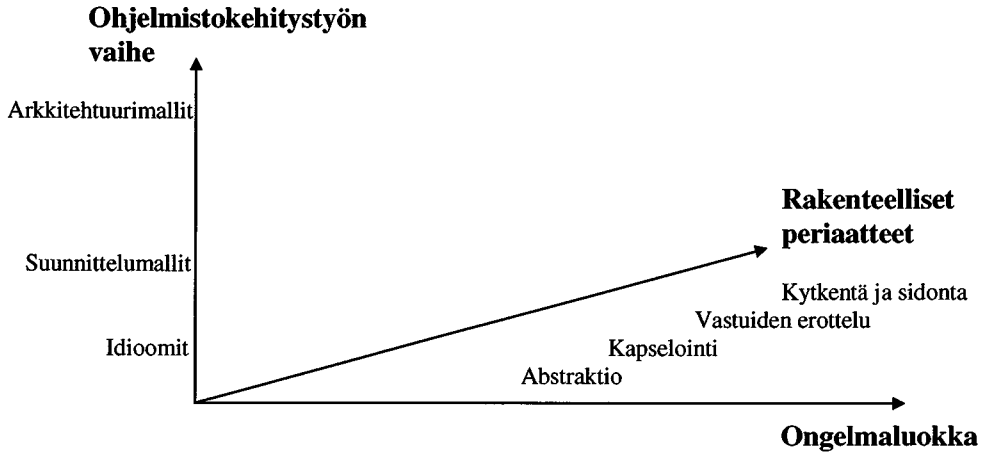
suunnittelumallien ominaisuuksia sekä helpottaa mallien löytymistä eri ohjelmistotuotannon vaiheissa. Vaikka suunnittelumallien määrä kasvaisi, voidaan aina tarkentaa ja lisätä ratkaistavia ongelmia (vrt. Buschmann ym. 1996, s. 379).

	Ohjelmistotuotannon vaihe		
Ongelmaluokka	Arkkitehtoniset mallit	Suunnittelu mallit	Idiomit
Mudasta rakenteeseen	Layers Pipes and Filters Blackboard	Interpreter	
Hajautetut järjestelmät	Broker Pipes and Filters Microkernel		
Vuorovaikutteiset järjestelmät	Model-View-Controller Presentation-Abstraction-Control		
Mukautuvat järjestelmät	Microkernel Reflection		
Luominen		Abstract Factory Prototype, Builder	Singleton Factory Method
Rakenteellinen jakaminen		Whole-Part Composite	
Työn organisointi		Master-Slave Chain of Responsibility Command, Mediator	
Käytön kontrolli		Proxy. Facade. Iterator	
Palvelun muuntaminen		Bridge, Strategy, State	Template Method
Palvelun laajentaminen		Decorator Visitor	
Hallinta		Command Processor View Handler, Memento	
Adaptointi		Adapter	
Kommunikointi		Publisher-Subscriber Forwarder-Receiver Client-Dispatcher-Server	
Resurssien hallinta		Flyweight	Counted Pointer

Taulukko 3. Buschmannin luokittelun 1996 mukaan järjestetyt suunnittelumallit (Buschmann ym. 1996)

4.7 Buschmannin luokittelu 1995

Edellä esitetyn Buschmannin luokittelun 1996 pohjana on käytetty Buschmannin ja Meunierin vuonna 1995 esittämä tapaa luokitella suunnittelumallit (kuvio 7). Perusidea on sama kuin Buschmannin luokittelussa 1996, mutta heillä on mukana kolmaskin ulottuvuus: rakenteelliset periaatteet.



Kuvio 7. Buschmannin luokittelun 1995 ulottuvuudet (Buschmann & Meunier 1995)

Kuvion 7 esittämää asetelmaa Buschmann ja Meunier (1995) perustelevat sillä, että suunnittelumallien toiminnallista puolta ei voi muuten ymmärtää. Suunnittelumallithan nojaavat tiettyyn oliokeskeiseen periaatteeseen. Tämä luokittelutapa on luonnollisestikin olioparadigmaan perustuva luokittelu. Nämä periaatteet muodostavat kolmannen ulottuvuuden, joka rakentuu seuraavista osista:

1. Abstraktio: Suunnittelumalli antaa abstraktin tai yleistetyn kuvan tietystä, usein monimutkaisesta kokonaisuudesta tai tehtävästä ohjelmistossa.
2. Kapselointi: Suunnittelumalli kapseloi tietyt yksityiskohdat olioista, komponenteista tai palveluista ja siirtää riippuvuudet asiakkaille, taikka suojaa nämä yksityiskohdat niin, ettei niihin pääse käsiksi.
3. Vastuiden erottelu: Suunnittelumalli siirtää tietyt vastuut erillisille olioille tai komponenteille, kun se ratkaisee tiettyä tehtävää tai tukee tiettyä palvelua.
4. Kytkentä ja sidonta: Suunnittelumalli siirtää tai vähentää rakenteellisia tai viestinnällisiä suhteita ja riippuvuuksia muuten voimakkaasti sidottujen olioiden väliltä.

Buschmann ym. ovat huomanneet tämän luokittelun heikkouden itsekin: "Kolmas ulottuvuus 'rakenteelliset periaatteet' kuvaa suunnittelumallien pohjalla olevia teknisiä ratkaisuja. ... Kuitenkaan rakenteellinen periaate, samalla tavalla kuin Gamman ym. kohdekriteeri, ei ole merkittävä valittaessa suunnittelumallia – joten jätimme tämän kriteerin pois määritellessämme uutta luokittelua" (Buschmann ym. 1996, 368). Buschmannin ym. (1996) mielestä Gamman ym. (1995) kohdekriteeri ja Buschmannin ja Meunierin (1995) rakenteellinen periaate, eivät ole keskeisiä kriteerejä valittaessa suunnittelumallia. Tämä sen vuoksi, että suunnittelija tuskin ajattelee, kuinka suunnittelumalli ratkaisee ongelman, vaan pikemminkin sen mukaan, kuinka laadukas ratkaisu on.

4.8 Martinin luokittelu

Robert Martin (1995) on tehnyt oman luokituksensa perustuen kokemuksiin, joita hänen projektiryhmänsä sai useiden sovellusten rakentamisesta. Projektin tarkoituksena oli rakentaa ohjelma, joka analysoi ja arvioi tietokoneella tehtyjen rakennusten arkkitehtuuriratkaisuja. Koska projektiin kuului useiden eri sovellusten tekeminen, he yrittivät luoda ohjelmille yhteisen sovelluskehyksen. Tämän sovelluskehyksen rakentamisen yhteydessä he löysivät ohjelmista toistuvia ratkaisuja, joita heidän mielestään voi kutsua suunnittelumalleiksi.

Martin toteaa, että mikäli he olisivat tunteneet suunnittelumallit etukäteen, he olisivat säästyneet suurelta työltä. He löysivät omista sovelluksistaan kaikki Gamman ym. (1995) kirjassa esiintyvät suunnittelumallit, muodossa tai toisessa. Martin jakaa ryhmänsä löytämät suunnittelumallit seuraaviin luokkiin

1. säiliösuunnittelumallit
2. korkean tason suunnittelumallit
3. alemman tason suunnittelumallit
4. C++:aan sidotut suunnittelumallit

Säiliösuunnittelumallit koskevat säiliöluokkia ja iteraattoreita[4]. Säiliöt muodostavat heidän tietomallinsa perustan, joten ei ole yllättävää, että useat suunnittelumallit olivat tekemisissä säiliöiden kanssa. Nämä suunnittelumallit koskevat monimuotoisten rajapin-

[4] Iteraattorit käyvät läpi jonkin säiliön sisältämät tietoelementit tietyssä järjestyksessä ja suorittavat kunkin olion kohdalla jonkin toiminnon (Koskimies 1997).

tojen luomista säiliöihin, jotta säiliön asiakkaiden ei tarvitse tietää, mitä säiliötä ne käyttävät.

Korkean tason suunnittelumalleilla on perustavaa laatua oleva vaikutus heidän sovelluksensa korkean tason suunnitteluun. Nämä suunnittelumallit toistuvat muodossa tai toisessa läpi koko suunnitelman. Ne käsittelevät lähdekoodin riippuvuuksien hallintaa eri alijärjestelmien välillä.

Alemman tason suunnittelumalleja he löysivät järjestelmästään paljon. Ne esiintyvät yhä uudelleen erilaisina tekniikoina ja temppuina koodissa. Ne käsittelevät paikallisen koodin uudelleenkäyttöä, kaksinkertaisten hierarkioiden hallintaa, äärellisiä tilakoneita, asiakas/palvelin-rajapintoja sekä kyselyiden ja valintojen tekemistä säiliöihin.

Useat suunnittelumallit koskivat C++ kieltä. Muissa kielissä nämä suunnittelumallit eivät ehkä olisi esiintyneet tai olisivat olleet toisen muotoisia. Martinin mielestä nämä kielisidonnaiset suunnittelumallit saattavat olla erittäin tärkeitä koko suunnittelumalliajattelussa. Näiden suunnittelumallien löytäminen kesti pitkän aikaa ja sisälsi paljon kokeiluja ja virheitä. Nämä suunnittelumallit helpottivat tekemään oliosuuntautunutta suunnittelua niissä tilanteissa, joita tulee vastaan vain C++-kielessä. Martin toivookin, että niiden kirjaaminen ja selittäminen auttaa toisia suunnittelijoita vastaavissa tilanteissa.

Martinin käyttämä jako perustuu toiminta-alan mukaiseen jakoon ja on yksiulotteinen. Säiliöluokka ei kuitenkaan kuulu toiminta-alan mukaiseen jakoon. Tämä luokittelu johtaa pitemmällä tähtäimellä samaan kuin Gamman ja Zimmerin luokittelut, eli jokainen kategoria tulee sisältämään liikaa suunnittelumalleja. Tämä aiheuttaa vaikeuksia löytää ongelmatilanteeseen sopiva suunnittelumalli.

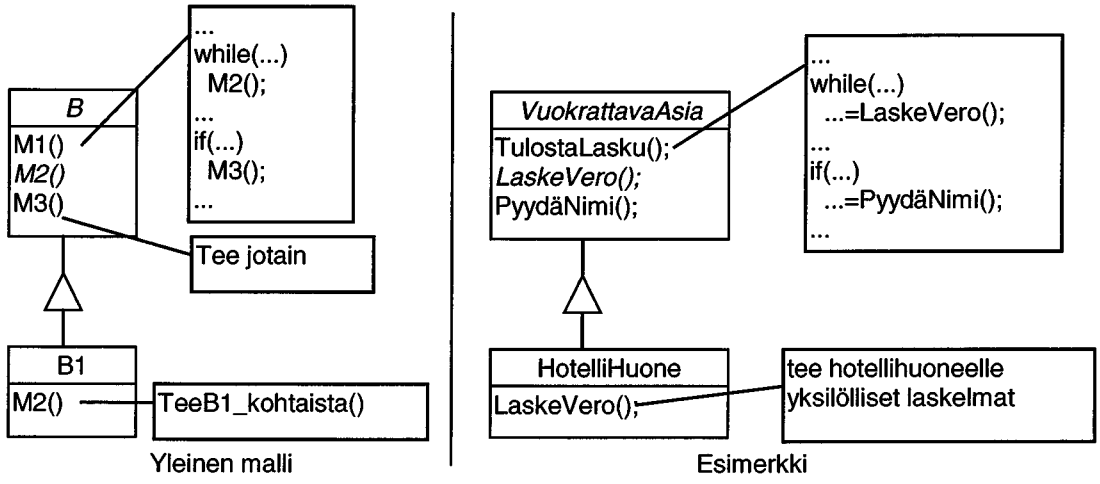
4.9 Preen luokittelu

Pree (1995) on luokitellut suunnittelumallit seitsemään ns. metasuunnittelumalliin. Luokittelussa tarkastellaan luokkien välisiä suhteita sekä vastuukysymyksiä. Metamallien avulla on pyritty löytämään Gamman ym. esittämien suunnittelumallien perusratkaisut. Nämä perusratkaisut ovat Preen mielestä ns. metasuunnittelumalleja, joiden avulla kaikki suunnittelumallit ovat rakennettavissa.

Metasuunnittelumallit ovat suunnittelumallien rakenteellisia ratkaisuja ja on tehty sovelluskehysten rakentamista silmälläpitäen (Buschmann ym. 1996). Näin ollen metamallit ovat enemmänkin rakenteellisia periaatteita kuin ratkaisuja tiettyyn ongelmaan tietyssä kontekstissa. Luokittelu on vahvasti olioparadigmaan perustuva ja käyttää hyväkseen perintää, monimuotoisuutta ja myöhäistä sidontaa. Luokkien välisten suhteiden voidaan katsoa heijastavat myös suunnittelumallien toiminta-alan mukaista luokittelua.

Metamalleissa luokan tai olion käyttäytyminen on jaettu ns. kuumiin ja kylmiin pisteisiin. *Kylmät pisteet* ovat paikkoja, joissa sovellus toimii aina samalla tavoin. *Kuumilla pisteillä* saadaan ohjelman laajennettavuutta ja mukautuvuutta lisättyä. Kuumat pisteet ovat ns. *hook-operaatioita* ja kylmät pisteet *template-operaatioita*[5]. Template-operaatiot kuvaavat toiminnan yleisen logiikan, jota voidaan sitten muunnella hook-operaatioiden avulla.

Kuviossa 8 nähdään esimerkki, kuinka Preen esittelemät template- ja hook-operaatiot toimivat. B-luokan operaatiot toteuttavat kylmän pisteen ja B1-luokan operaatiot toteuttavat kuumat pisteet sovelluksessa. Tietyssä mielessä voidaan sanoa, että monimutkaisia operaatioita kutsutaan template-operaatioiksi ja ne voidaan toteuttaa yksinkertaisemmilla hook-operaatioilla. Template-operaatiot määrittelevät yleisemmän ohjausrakenteen, jota voidaan muuntaa hook-operaatioiden avulla. Kuvassa B1-luokka perii toiminnan ohjausrakenteen B-luokalta eli M1()-operaation. Kuvan esimerkissä B-luokka toteuttaa kylmän pisteen sovellukselle operaatiolla M1(), jota käytetään yleistapauksena. B1-luokka toteuttaa itselleen ominaisen toiminnan korvaamalla B-luokan M2()-operaation.



Kuvio 8. Template- ja Hook-operaatioiden toiminta (Pree 1995)

[5] Pree käyttää termejä hook ja template päinvastoin kuin esimerkiksi Gamma ym. (1995).

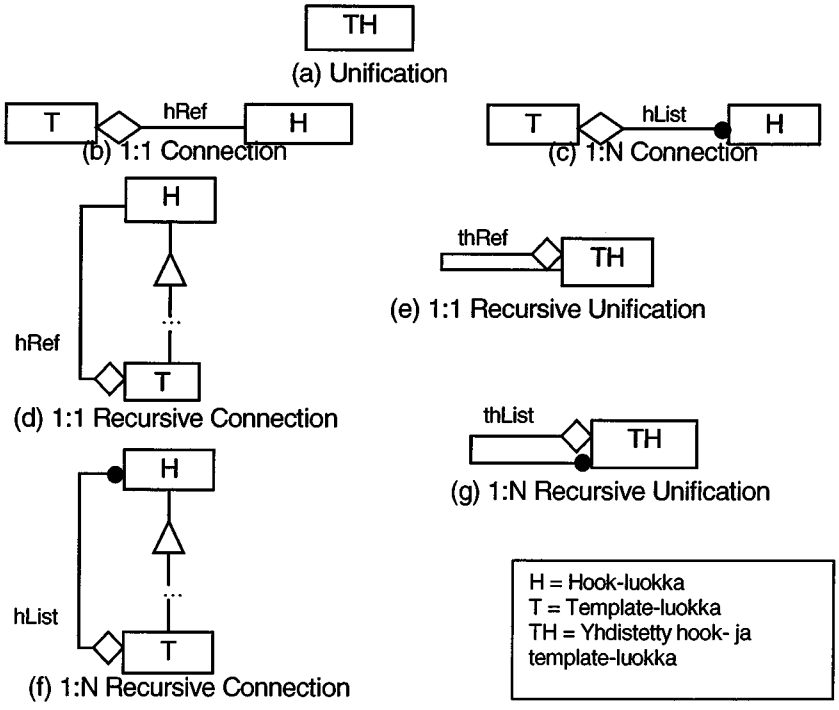
Template- ja hook-operaatiot voidaan selvyyden ja hallittavuuden vuoksi sijoittaa omiin luokkiinsa. Template-operaatioita sisältävät luokat ovat *Template-luokkia* ja Hook-operaatioita sisältävät luokat *Hook-luokkia.*

Näiden ajatusten pohjalta Pree jakaa mahdolliset luokkarakenteet kolmen kysymysten perusteella.

1. Voiko template-luokan olio viitata ainoastaan yhteen olioon vastaavassa hook-luokassa vai yhteen tai useampaan olioon hook-luokissa?
2. Onko template-luokka jälkeläinen hook-luokalle?
3. Onko molemmat luokat yhdistetty?

Ja vastaukseksi saadaan kuvion 9 esittämät seitsemän metasuunnittelumallia. Kuviossa 8 esitetty esimerkki on Unification-metasuunnittelumalli, koska Template- ja Hook-operaatiot ovat samassa luokassa.

Template-luokka ei voi olla hook-luokan esi-isä, koska template-luokka on aina konkreettisempi kuin hook-luokka. Hook-luokan metodit voivat olla abstrakteja metodeita tai konkreetteja metodeja, eli tavallisia tai template-metodeja. Molemmat tarjoavat konkreetin, tarkoituksenmukaisen oletustoteutuksen. Näin ollen hook-luokat voivat olla abstrakteja tai konkreetteja luokkia. (Pree 1995.)



Kuvio 9. Preen (1995) seitsemän metasuunnittelumallia

Preen mielestä kaikki suunnittelumallit löytyvät muodossa tai toisessa metasuunnittelumalleista. Kuitenkaan Preen luokittelusta ei ole hyötyä ohjelmistotuotantoon, sillä kun vastaan tulee konkreettinen ongelma, johon tulisi löytää vastaus, on metasuunnittelumalleista varsin vähän hyötyä.

4.10 Irvingin ja Eichmannin luokittelu

Irving ja Eichmann (1996) pyrkivät artikkelissaan luomaan suunnittelumalleille jaottelun suunnittelumallien mukautuvuuden perusteella. Mukautuvuudella he tarkoittavat, kuinka hyvin ohjelman suunnitelmaa voidaan muuttaa (usein rajoitetusti), jotta se käy spesifioituun kontekstiin. Heidän mukaansa mukautuvuus on tärkeää uudelleenkäyttävässä suunnittelussa, koska uudelleenkäytettäväksi tarkoitettujen suunnitelmien tulee olla helposti muutettavissa. Heidän luokittelunsa perustuu voimakkaasti olioparadigman mukaiseen luokitteluun. Mutta samoin kuin Preen metamalleissa, on heidänkin luokittelussaan nähtävissä suunnittelumallin toiminta-alaa koskevaa luokitusta.

Irving ja Eichmann (1996) ovat jakaneet suunnittelumalleja kuumien ja kylmien pisteiden perusteella. Perusajatuksen tähän luokitteluun he ovat saaneet Preen metamalleista, joiden avulla voidaan suunnitella sovelluskehyksiä. Nämä metamallit on jaoteltu ns. kuumien ja kylmien pisteiden avulla seuraavasti:

- Kuumia pisteitä ovat ne alueet sovelluskehyksestä, jotka on tarkoitettu muutettaviksi ja joiden on jäätävä joustaviksi.
- Kylmiä pisteitä ovat ne alueet sovelluksesta, jotka eivät muutu, ts. muodostavat sovelluskehyksen ytimen, jonka varaan muuttuvat osat rakennetaan.

Irving ja Eichmann tarkastelevat, kuinka mallien tyyppiyhteensopivuus, semantiikka, koostaminen ja protokolla jakaantuvat kuumiin ja kylmiin pisteisiin. Semantiikka muodostuu olion tai luokan omista piirteistä sekä kaikkien sen esi-isä luokkien piirteistä. Protokollalla he tarkoittavat, kuinka luokat tai oliot kommunikoivat toistensa kanssa. Mukautuvat arkkitehtuurit voidaan suunnitella niin, että sovelluksen aliluokat voivat muuttaa esi-isiltä perittyä semantiikkaa. Tällä tavoin suunnittelija voi halutessaan luoda kuumia pisteitä arkkitehtuuriin.

Irving ja Eichmann sanovat, että teknisestä näkökulmasta Gamman ym. käyttävät kahta eri tapaa luodessaan mukautuvia suunnittelumalleja. Ensinnäkin Gamma ym. käyttävät

abstrakteja luokkia luodakseen tyyppiyhteensopivia luokkia, joilla on samanlainen protokolla. Toisaalta he käyttävät kylmiä pisteitä tuottamaan tyyppiyhteensopivuutta ja protokollaa, jotta voidaan sallia suunnittelumallin osallistujien kesken eroavaa käyttäytymistä, eli kuumia pisteitä.

Irvingin ja Eichmannin luokittelussa on Preen ajatusta viety askeleen verran pidemmälle, mutta heidänkin luokittelunsa nojaa liian voimakkaasti olioparadigman mukaiseen jaotteluun. Heidänkään luokittelunsa perusteella ei voi helposti löytää mahdollisia suunnittelumalleja konkreettiseen ohjelmistotuotannon ongelmaan.

4.11 Yhteenveto ja arvio luokitteluista

Suunnittelumallien luokittelukriteerit voidaan jakaa kolmeen eri ryhmään: (1) ohjelmistotyön vaiheen mukainen jako, (2) toiminta-alan mukainen jako ja (3) olioparadigman mukainen jako (taulukko 4). Ohjelmistotyön vaihe jakaa suunnittelumallit työvaiheiden mukaan, kuten esimerkiksi analyysi- suunnittelu- ja toteutusvaiheisiin. Tähän luokitteluun nojaavat Zimmerin luokittelu (Zimmer 1995a), Buschmannin luokittelu 1996 (Buschmann ym. 1996), Buschmannin luokittelu 1995 (Buschmann & Meunier 1995) ja Martinin luokittelu (Martin 1995).

Toiminta-alan mukaan tapahtuva luokittelu jakaa suunnittelumallit sen mukaan, kuinka ne toimivat taikka minkälaisen ongelman ne ratkaisevat. Suunnittelumallien tarkoitus voi olla esimerkiksi olioiden luomiseen liittyvät seikat tai toiminnan vastuun jakaminen eri olioiden kesken. Tähän kategoriaan lasketaan myös rakenteellisten suhteiden kautta arvioitu toiminta, jota Preen (1995) luokittelu käyttää. Tähän kategoriaan voidaan katsoa kuuluvan myös Gamman luokittelun (Gamma ym. 1995), Buschmannin luokittelun 1996 (Buschmann ym. 1996) ja Buschmannin luokittelun 1995 (Buschmann & Meunier, 1995) yksi ulottuvuus. Myös Irvingin ja Eichmannin luokittelun (Irving & Eichmann 1996) voidaan katsoa sisältävän toiminta-alan mukaisen jaon suunnittelumalleille. Olioiden ja mallien piirteiden aiheuttama toiminnallisuus määrää luokan, mihin suunnittelumalli kuuluu.

Kolmantena luokittelukriteerina voidaan pitää olioparadigmaan kuuluvien seikkojen perusteella tapahtuvaa luokittelua. Tällaisia luokitteluja ovat Buschmannin luokitteluun 1995 (Buschmann & Meunier 1995) kuuluva kolmas ulottuvuus. Gamman (Gamma ym. 1995), Preen (1995), ja Irvingin ja Eichmannin (Irving ja Eichmann, 1996) luokittelut

perustuvat vahvasti tähän luokkaan. Heille keskeinen luokittelukriteeri on, kuinka oliot tai luokat käyttäytyvät ja ovat suhteessa toisiinsa sekä näin toteuttavat joustavia ja uudelleenkäytettäviä ratkaisuja suunnitelman luomiseen. Taulukossa 4 on yhteenveto luokittelutavoista.

Luokittelu	Ohjelmisto-tuotannon vaihe	Suunnittelumallin toiminta-ala	Oliokeskeisyyteen pohjautuva luokittelu
Gamman luokittelu		X	X
Zimmerin luokittelu	X		
Buschmannin 1996 luokittelu	X	X	
Buschmannin 1995 luokittelu	X	X	X
Martinin luokittelu	X		
Preen luokittelu		X (suhteiden muodossa)	X
Irvingin ja Eichmannin luokittelu		X (suhteiden muodossa)	X

Taulukko 4. Luokittelujen karkea jako.

Huomattavaa on, että Zimmer (1995a) on saanut luokittelussaan jaettua suunnittelumallit erilaisille semanttisille tasoille analysoimalla Gamman ym. (1995) esittelemiä suunnittelumalleja. Buschmann ym. (1996) ovat päätyneet jakamaan suunnittelumallit eri ohjelmistotuotannon vaiheiden mukaan omien tutkimusten ja kokemusten kautta, ja Martin (1995) on päätynyt eri ohjelmistotuotantovaiheiden mukaiseen jakoon analysoimalla olemassa olevaa koodia. Tästä voimme päätellä, että semanttisiin tasoihin jakaminen on oleellista, kun yritetään ymmärtää suunnittelumalleihin liittyvää kontekstia.

Suunnittelumallien toiminta-alan mukainen jako näkyy Gamman tarkoituskriteerin, Preen luokittelussa luokkien välisissä suhteissa, Irvingin ja Eichmannin luokkien välisissä suhteissa, Buschmannin luokittelun 1995 sekä Buschmannin luokittelun 1996 ongelmaluokan mukaan tapahtuvissa luokitteluissa.

Oliokeskeisyyteen perustuvat Preen (1995) metamallit, Irvingin ja Eichmannin (1997) sekä Buschmannin (Buschmann & Meunier 1995) luokittelu 1995 kolmas ulottuvuus. Se miten suunnittelumallit ovat rakentuneet, on implisiittisesti suunnittelumalleissa. Näiden rakenteiden ymmärtäminen on oleellista, jotta suunnittelumalleja voidaan käyttää oliokeskeisessä ohjelmistotuotannossa. Rakenteellisten periaatteiden ymmärtäminen on tärkeää, mutta se on jo sisäisesti suunnittelumallien ulkoasussa, kun ne kuvataan luokkakaavioina. Tässä yksi hyvä syy piirtää suunnittelumalleista kuva. Suunnittelu-

mallien ymmärtämiseksi ja käyttämiseksi on tärkeää osata hyvin oliokeskeisyyden periaatteet.

Buschmannin luokittelu 1996 (Buschmann ym. 1996) on ohjelmistotuotantoon käyttökelpoisin, koska se on selkeä ja jakaa suunnittelumallit ohjelmistotyön eri vaiheiden mukaan. Toisaalta heidän luokittelunsa jakaa mallit ratkaistavan ongelman mukaan. Heidän jaottelussaan tulevat hyvin esille suunnittelumallien perusominaisuudet: konteksti, ongelma ja ratkaisu. Kontekstina toimii suunnitteluvaihe. Ongelma on suoraan luokittelukriteerinä. Ratkaisu voidaan valita niiden suunnittelumallien kesken, joihin konteksti ja ongelma viittaavat.

Vaikka Buschmannin luokittelu 1996 osoittautui parhaaksi, on siinä yksi huomioitava seikka. Jaottelu on periaatteessa yksiulotteinen. Ohjelmistotuotannon vaiheen mukainen luokittelu ei jaa suunnittelumalleja kovin hyvin. Arkkitehtonisten mallien ratkaisemat ongelmat ovat hyvin erityylisiä kuin suunnittelu mallien.

4.12 Yhteenveto

Tässä luvussa ensiksi esiteltiin suunnittelumallit ja annettiin yksi esimerkki suunnittelumallista. Tämän jälkeen esiteltiin ja arvioitiin seitsemän luokittelutapaa suunnittelumalleillle. Näistä valittiin ohjelmistotuotantoon sopivin. Tällaiseksi osoittautui Buschmannin ym. (1996) esittelemä luokittelu. Se jakaa suunnittelumallit ohjelmistotuotannon vaiheiden mukaan ja tukee suunnittelumallien löytymistä ongelmatilanteen mukaan.

5 SUUNNITTELUMALLIEN KÄYTTÖ

Oliokeskeinen lähestymistapa ohjelmistotuotantoon on lyönyt itsensä läpi lähinnä kahden argumentin voimin: mallintaminen ja uudelleenkäyttö (Taivalsaari 1993). Oliokesekeisyyden avulla on helpompi mallintaa ohjelmistoja, koska se tarjoaa työkalut tukemaan ihmisen tapaa ajatella maailmaa. Uudelleenkäyttö liittyy tiivisti ajatukseen perinnästä. Olemassa olevia luokkia voidaan käyttää määrittelemään uusia.

Suunnittelumallit ovat yksi tapa toteuttaa uudelleenkäyttöä. Uudelleenkäytöllä ei ymmärretä pelkästään koodin ja toteutuksen uudelleenkäyttöä vaan sillä voidaan tarkoittaa myös suunnitelmien ja jopa dokumentaation uudelleenkäyttöä (Taivalsaari 1993, Sametinger 1997). Suunnittelumallien käyttö ohjelmistotuotannossa on kokeneiden ohjelmistosuunnittelijoiden tietotaidon uudelleenkäyttöä. Tämän vuoksi tässä luvussa keskitytään kuvaamaan mitä hyötyä suunnittelumalleista on ohjelmistotuotantoprosessissa uudelleenkäytön näkökulmasta. Luvun lopuksi keskitytään, kuinka uudelleenkäyttöä sovelletaan ohjelmistotuotannossa ja siihen, kuinka suunnittelumallit voivat olla apuna tässä.

5.1 Uudelleenkäyttö oliokeskeisessä ohjelmistotuotannossa

Uudelleenkäytöllä pyritään luonnollisesti saavuttamaan luotettavampia ohjelmistoja alhaisemmin kustannuksin. Taivalsaari (1993) on kerännyt eri lähteistä esiin tulleita uudelleenkäytön etuja. Uudelleenkäytöllä voidaan:

1. nopeuttaa ohjelmistotuotantoa,
2. vähentää toistuvia työvaiheita,
3. mahdollistaa monimutkaisemman ohjelmiston toteuttaminen,
4. helpottaa muutosten tekoa ja ylläpitoa,
5. nostaa ohjelmiston laatua,
6. parantaa ohjelmiston suoritustehoa,
7. mahdollistaa pienemmät ohjelmointiryhmät,
8. helpottaa hyvän suunnittelun oppimista,
9. rohkaista eksperttiyden jakamista,
10. helpottaa suunnittelijoiden keskeistä kommunikointia ja
11. helpottaa dokumentointia.

Vastaavasti Taivalsaari (1993) löytää myös esteitä uudelleenkäytölle. Hän jakaa keskeiset ongelmaryhmät neljään osaan: käsitteelliset, tekniset, organisatoriset ja taloudelliset sekä psykologiset ongelmat.

Kaksi yleisintä tekniikkaa uudelleenkäyttöön oliopohjaisissa järjestelmissä on luokkaperintä ja olioiden koostaminen. Luokkaperintää kutsutaan usein lasilaatikkouudelleenkäytöksi (white-box, glass-box reuse), koska perinnässä aliluokat tietävät, kuinka yliluokat on toteutettu.

Oliokoostaminen on vaihtoehtoinen lähestymistapa luokkaperinnälle. Koostamisessa toimintaa ylläpidetään kokoamalla tai koostamalla olioita, jotta saavutettaisiin monipuolisempaa käyttäytymistä. Koostaminen vaatii, että käytettävien olioiden rajapinnat on hyvin määritelty. Tämän kaltaista uudelleenkäyttöä kutsutaan mustalaatikkouudelleenkäytöksi (black-box reuse), koska olioiden sisäiset toteutustavat eivät ole muiden koostamisessa käytettyjen olioiden tiedossa.

Molemmilla yllämainituilla uudelleenkäyttötavoilla on omat etunsa ja haittansa. Luokkaperintä määritellään staattiseksi ohjelman käännösaikana ja sitä on suoraviivaista käyttää, koska sitä tuetaan suoraan ohjelmointikielitasolla. Luokkaperinnän avulla on myös helpompi muuttaa uudelleenkäytettävää toteutusta. Mikäli aliluokka korvaa yliluokkansa operaatioita, se voi vaikuttaa myös muuttamattomiin operaatioihin, mikäli muuttumattomat operaatiot käyttävät hyväkseen muutettuja operaatioita (Pree 1995).

Oliokoostaminen määräytyy vasta ajonaikaisesti, kun oliot viittaavat muihin olioihin. Koosteolioiden täytyy käyttää määriteltyjä rajapintoja luokkiin ja näin toteuttaa kapseloinnin periaatetta. Koska olioita käytetään vain niiden rajapintojen kautta, voidaan mikä tahansa olio korvata ajonaikaisesti toisella oliolla, mikäli se on vain samaa tyyppiä. Näin ollen, mikäli olion toteutus tehdään rajapintaa kunnioittaen, niin oliot eivät ole toisistaan toteutusriippuvaisia.

Oliokoostamisella on myös toinenkin vaikutus järjestelmän suunnitteluun. Mikäli koostamista käytetään luokkaperinnän sijasta, auttaa koostaminen pitämään jokaisen luokan kapseloituna, ja kukin luokka voi keskittyä omaan tehtäväänsä. Luokat ja luokkahierarkiat säilyvät pieninä, eivätkä kasva hallitsemattomiksi kokonaisuuksiksi. Toisaalta suunnittelu, joka perustuu olioiden koostamiseen, sisältää useampia olioita, vaikkakin vähemmän luokkia, ja järjestelmän käyttäytyminen riippuu olioiden välisestä käyttäytymisestä.

Ideaalimaailmassa ohjelmistot voitaisiin kostaa olemassa olevista komponenteista. Koko sovelluksen toiminta voitaisiin määritellä vain koostamalla tarpeellisia olioita, mutta todellisuudessa olioita ei koskaan ole tarpeeksi. Luokkaperinnän avulla voidaan nopeasti tehdä uusia komponentteja, joista sitten voidaan koostaa vaadittavaa toiminnallisuutta. Näin ollen luokkaperintä ja koostaminen ovat toisiaan tukevia tekniikoita.

5.2 *Suunnittelumallien käyttö ohjelmistotuotannossa*

Jaaksi (1997) on maininnut, että suunnittelumallien sitominen OMT++-menetelmään tapahtuu tulevaisuudessa. Vielä tätä kirjoitettaessa tällaista ei ole tapahtunut. Suunnittelumallien ja –menetelmien sitominen toisiinsa olisi kuitenkin hyödyllistä. Yhden tällaisen menetelmän ovat esittäneet Ram ym. (1997).

Ramin ym. (1997) esittelemä Pattern Oriented Technique (POT) –menetelmä käyttää suunnitelman lähtökohtana suunnittelumalleja. POT tunnistaa suunnittelumallit keskenään keskusteleviksi olioiksi, ja tässä käytetään hyödyksi käytettävissä olevaa suunnittelutietoutta. POTissa on kymmenen vaihetta:

1. tunnista luokat,
2. tunnista vastuut,
3. tunnista kommunikoivat luokat,
4. tunnista luokkaryhmät,
5. tunnista luokkaryhmien kommunikointi,
6. määrittele luokkaryhmien kommunikointi abstraktilla tasolla,
7. tunnista suunnittelumallit,
8. tee karkea suunnitelma,
9. määritä vaihtoehtoisten ratkaisujen edut ja haitat sekä
10. tee yksityiskohtainen suunnitelma.

Vaiheessa seitsemän otetaan käyttöön suunnittelumallit, jotka pyritään tunnistamaan aikaisempien vaiheiden löydösten perusteella. Vaiheessa kahdeksan jokainen tunnistettu suunnittelumalli lisätään suunnitelmaan ja katsotaan, kuinka se vaikuttaa suunnitelmaan. Vaiheessa yhdeksän valitaan kriteerien mukaan ongelmaan sopivin ratkaisu. Tässä vaiheessa käytetään hyväksi Ramin ym. (1996) esittelemää tapaa verrata suunnittelumallien tarjoamia ratkaisuja. Lopuksi vaiheessa 10 valitaan sopivimmat suunnittelumallit ratkaistavaan tehtävään.

Ramin ym. (1997) POT-lähestymistavalla on kuitenkin huomattavia puutteita. Ensinnäkin heidän lähestymistapansa lähtee siitä, että suunnittelija tuntee käytettävissä olevat suunnittelumallit lähes ulkoa. Suunnittelumallit tunnistetaan luokkien tai olioiden välisen kommunikaation perusteella, ei ongelma- tai tarvelähtöisesti. Toinen vakava puute on, ettei eri vaiheiden suunnittelumalleja oteta huomioon. Heidän esittelemässään mallissa on suunnittelijan tunnistettava kaikki olemassa olevat luokat ja niiden välinen kommunikointi, ja vasta tämän jälkeen voidaan tunnistaa suunnittelumallit. Tässä lähestymistavassa ei lainkaan käytetä hyväksi sitä tietämystä, joka on löydettävissä suunnittelumalleista.

5.3 Ohjelmistotuotanto suunnittelumalleja hyödyntäen

Hyvää menetelmää, jossa hyödynnettäisiin suunnittelumalleja, ei siis ole vielä olemassa. Tämän vuoksi otamme esimerkkiä komponenttipohjaisesta lähestymistavasta suunnitteluun. Komponenttien uudelleenkäyttö ei tarkoita ainoastaan koodin uudelleenkäyttöä. On mahdollista myös käyttää uudelleen määrityksiä ja suunnitelmia. Uudelleenkäyttö voidaan jakaa kahteen osaan: suunnittelu uudelleenkäyttöä hyödyntäen ja suunnittelu uudelleenkäyttöä varten. (vrt. Sommerville 1992)

Ohjelmistotuotannolla uudelleenkäyttöä hyödyntäen voidaan myös tarkoittaa suunnittelumallien hyväksikäyttöä. Ohjelmistoa suunniteltaessa otetaan huomioon, että on jo olemassa valmiiksi ratkaisuja usein toistuviin ongelmiin, ja näitä ratkaisuja käytetään hyväksi käyttökelpoisissa tilanteissa. Suunnittelu uudelleenkäyttöä varten ei myöskään poissulje suunnittelua uudelleenkäyttöä hyödyntäen, mutta usein kannattaa ainakin isommissa organisaatioissa tehtävää varten nimetä oma ryhmänsä. Uudelleenkäytettävien suunnittelumallien tekeminen ja löytäminen on aikaa vievää työtä. Usein työ ei tule edes yhden ohjelmistoprojektin aikana valmiiksi, vaan jatkuu useiden eri projektien rinnalla. Suunnittelumallien käytössä kannattaa aloittaa olemassa olevien suunnittelumallien ottamisella mukaan ohjelmistotuotantoon.

Ohjelmistotuotanto suunnittelumalleja hyödyntäen tarkoittaa, että uusia sovelluksia suunniteltaessa otetaan huomioon valmiiden suunnittelumallien olemassaolo ja niitä pyritään käyttämään hyväksi. Ideaalinen tilanne olisi, että suunniteltavasta järjestelmästä löytyisi sovelluskehys tai mallijärjestelmä, joka ohjaisi koko suunnitteluprosessia. Tällaisia on kuitenkin löydettävissä vain harvoille ja tarkasti määritellyille sovellusalueille. Keskeinen tavoite uudelleenkäytöllä on alentaa ohjelmistotuotantokustannuksia.

Ääriesimerkkinä Brooks (1987) mainitsee erääksi lupaavaksi hopealuodiksi ohjelmiston ostamisen. Kun ohjelmisto on tuotettu ja sitä käyttää yhden sijasta sata ihmistä, on ohjelmistokehittäjien tehokkuus satakertainen.

Vaikka kustannuskysymykset eivät ole ainoita merkittäviä seikkoja uudelleenkäytölle, ovat ne kuitenkin yksi tärkeimmistä syistä. Muita painavia seikkoja uudelleenkäytölle ovat seuraavat seikat (Sommerville 1992):

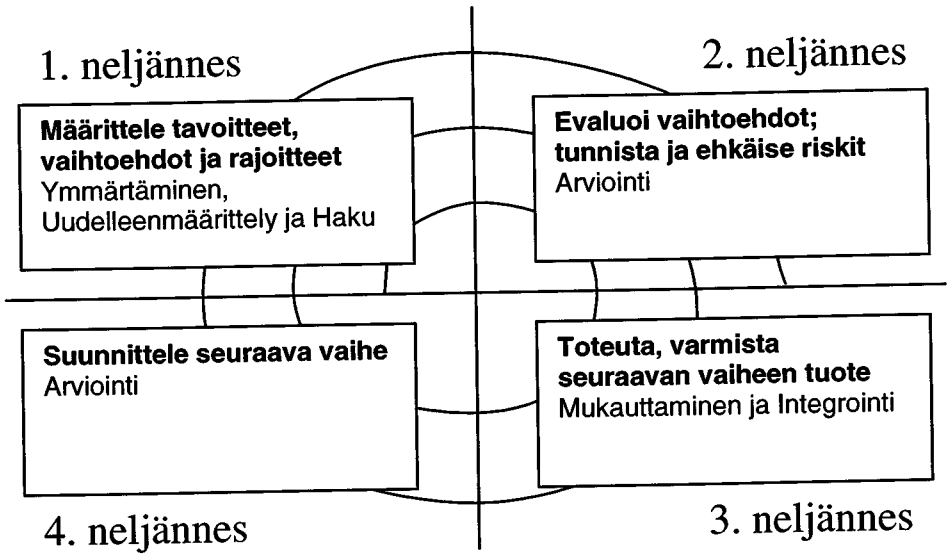
1. Järjestelmän luotettavuutta voidaan kohottaa. Voidaan väittää, että ainoastaan todellinen käyttö testaa käytettävät komponentit. Useammassa järjestelmässä käytetty komponentti on paremmin testattu kuin yhdessä järjestelmässä käytetyt komponentit.
2. Kokonaisriskiä voidaan vähentää. Valmiiden ratkaisujen kustannuksia voidaan arvioida tarkemmin. Projektin ohjaukselle tällainen tieto on kultaakin arvokkaampaa kustannusarviovaiheessa.
3. Asiantuntijoiden työpanosta voidaan ohjata tehokkaammin. Asiantuntijat voivat keskittyä tuottamaan uudelleenkäytettäviä komponentteja, koska heidän ei tarvitse toistaa samoja ratkaisuja projektista toiseen
4. Organisatorisia standardeja voidaan sitoa uudelleenkäytettävillä komponenteilla. Parhaita esimerkkejä tästä ovat valikkorakenteet ja muut graafisen käyttöliittymän standardit. Sitomalla tällaista tietoa komponentteihin saadaan kaikkien tuotettavien ohjelmistojen ulkoasu samankaltaiseksi. Tämä puolestaan vähentää oppimiskynnystä ja lisää käyttäjätyytyväisyyttä.
5. Ohjelmiston kehitysaikaa voidaan pienentää. Monesti ohjelmiston mahdollisimman nopea markkinoille tulo on paljon tärkeäpää kuin ohjelmiston kehityskustannukset. Uudelleenkäytöllä voidaan vähentää kehitys- ja testaustyötä.

Uudelleenkäyttöön perustuva ohjelmistotuotanto ei voi kulkea samalla tavalla kuin useimmat elinkaarimallit ehdottavat. Vieläkin suurin osa menetelmistä lähtee siitä oletuksesta, että ohjelmiston kehitys lähtee puhtaalta pöydältä. Todellisuudessa jokainen ohjelmistotuotantoa harrastava yritys on tehnyt komponentteja ja suunnitelmia, jotka ovat käyttökelpoisia tulevissa ohjelmistoissa. Jacobson ym. (1992) kirjoittavat, että jokainen tuotettava ohjelmisto on vanhan ohjelmiston muuttamista: täysin puhtaalta pöydältä lähtevä ohjelmistokehitys on erikoistapaus.

Sametinger (1997) toteaa, että uudelleenkäytön täytyy olla oleellinen osa ohjelmistotuotantoa. Ainoastaan näin voidaan uudelleenkäyttää muutakin kuin koodia. Uudelleenkäytön yhteydessä on ohjelmistotuotannon eri vaiheissa suoritettava seuraavia toimintoja (Sametinger 1997):

1. Ymmärtäminen: ongelman ymmärtäminen ja mahdollisen ratkaisun tunnistaminen, jotka perustuvat olemassa oleviin komponentteihin.
2. Uudelleenmäärittely: määrittely tehdään uudelleen niin, että voidaan käyttää olemassa olevia komponentteja.
3. Haku: hankitaan, arvioidaan ja käytetään olemassa olevia komponentteja.
4. Mukauttaminen: muutetaan ja mukautetaan komponentteja.
5. Integrointi: integroidaan komponentit tämän vaiheen tuotteeseen.
6. Arviointi: arvioidaan valmistettavien ja muutettujen komponentin uudelleenkäytettävyys.

Nämä vaiheet Sametinger (1997) on yhdistänyt Boehmin (1986) spiraalimalliin Kuvion 10 tavalla.



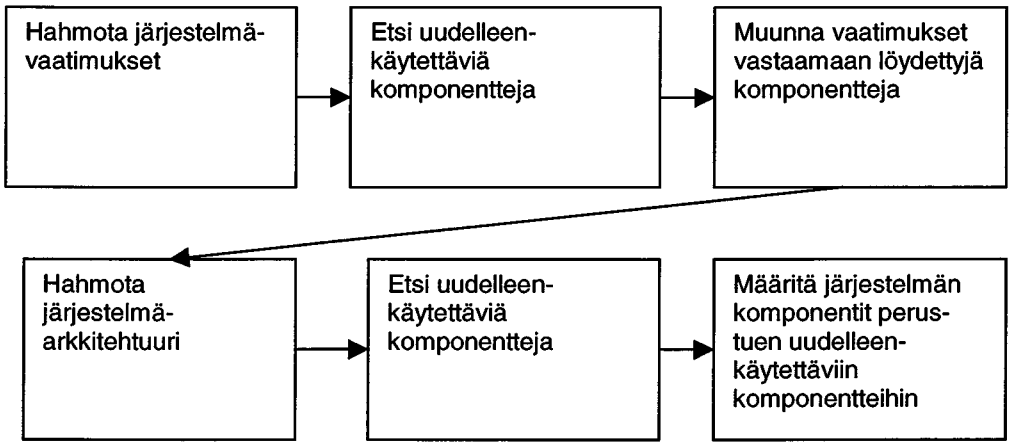
Kuvio 10. Uudelleenkäytön spiraali (Sametinger 1997)

Kuvion 10 ensimmäisessä neljänneksessä määritellään vaiheen tavoitteet, mahdolliset vaihtoehdot ja annetut rajoitteet. Uudelleenkäytön näkökulmasta meidän on ymmärrettävä ongelma ja etsittävä mahdolliset uudelleenkäytettävät komponentit. Löydettyjen komponenttien pohjalta on mahdollisesti tehtävä muutoksia määrittelyyn. Toisessa neljänneksessä evaluoidaan vaihtoehdot ja tunnistetaan sekä ehkäistään riskejä. Uudelleenkäytön kannalta tämä tarkoittaa, että on määriteltävä komponentteihin tehtävät muutokset, muutosten vaikutus ja riskit muutosten aiheuttamista ongelmista. Kolmannessa neljänneksessä toteutetaan käynnissä olevan vaiheen mukainen järjestelmän osa. Uudelleenkäytön näkökulmasta valitut komponentit muutetaan, mukautetaan ja integroidaan

järjestelmään. Neljännessä neljänneksessä suunnitellaan seuraava vaihe. Uudelleenkäytettävyyden kannalta tämä tarkoittaa, että spiraalin edellisessä vaiheessa tehtyjä ja muutettuja komponentteja arvioidaan, kuinka ne sopivat uudelleenkäytettäviksi myöhemmissä vaiheissa.

Kuviossa 10 esitetty Sametingerin (1997) uudelleenkäyttöön perustuva ohjelmistotuotanto tunnisti neljännessä neljänneksessään valmistettujen ja muutettujen komponenttien arvioinnin tulevaan uudelleenkäyttöön. Tunnistamiseen ei kuitenkaan annettu minkäänlaisia työkaluja.

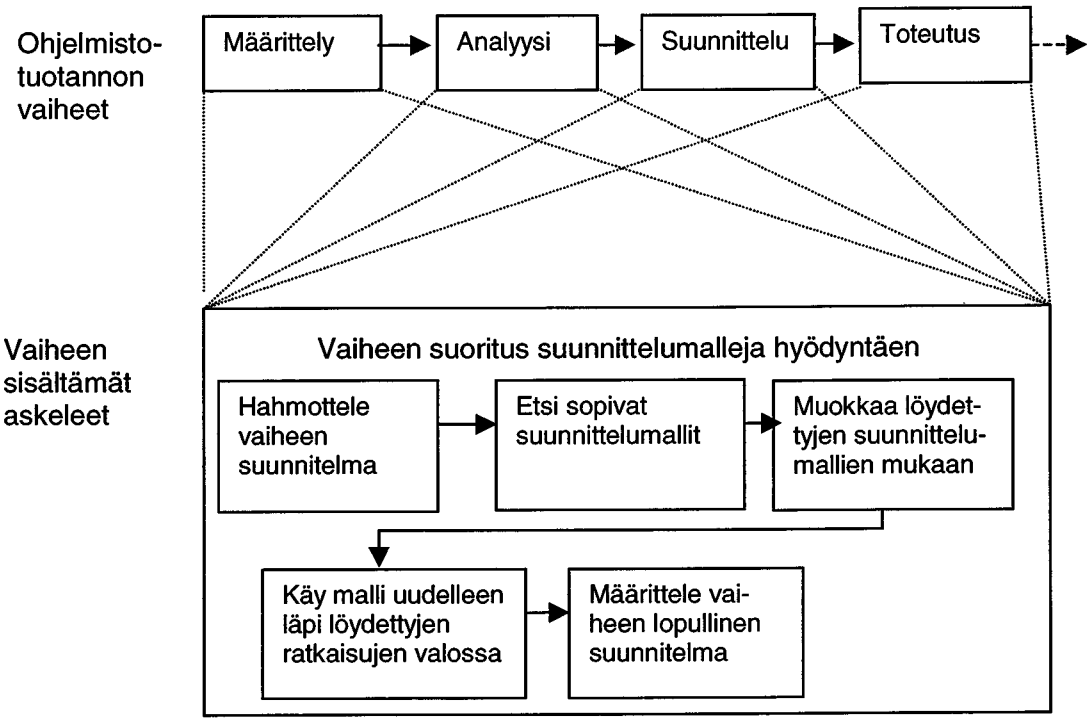
Sommervillen (1992) esittämä uudelleenkäyttöön sopiva elinkaarimalli näkyy kuviossa 11. Kuvion mukaan komponenttien käyttöön on kaksi kohtaa. Ensinnäkin hahmoteltaessa koko järjestelmän vaatimuksia ja toisaalta hahmoteltaessa järjestelmän arkkitehtuuria.



Kuvio 11. Uudelleenkäyttöön perustuva ohjelmistokehitys (Sommerville 1992)

Edellä esitetyt mallit on kuitenkin tarkoitettu lähinnä komponenttien uudelleenkäyttöön järjestelmässä ja ovat siksi epätäydellisiä suunnittelumallien uudelleenkäytölle. Huomattavaa kummassakin lähestymistavassa on se, että järjestelmälle asetettuja vaatimuksia on mukautettava sen mukaan, mitä komponentteja on saatavilla.

Edellisessä luvussa valittiin suunnittelumalleille luokittelu, joka sopii ohjelmistotuotantoon. Tässä luokittelussa suunnittelumallit jaetaan ohjelmistokehityksen eri vaiheiden mukaan. Niinpä jokaisen ohjelmistokehitysvaiheen täytyy sisältää jokin tapa käsitellä mahdollisia suunnittelumalleja. Seuraavassa kuviossa on esitetty ohjelmistotuotantovaiheet ja kutakin vaihetta koskeva suunnittelumallien käyttö.



Kuvio 12. Suunnittelumallien käyttö ohjelmistotuotannon eri vaiheissa.

Kuviossa 12 on esitetty kunkin ohjelmistotuotantovaiheen suorituksen sisältävän viisi erillistä vaihetta. Vaikka edellä kiinnitettiin huomiota siihen, että olemassa olevia vaatimuksia on muokattava olemassa olevien komponenttien mukaiseksi, ei se tarkoita samaa kuin kolmas askel vaiheen suorituksessa suunnittelumalleja hyödyntäen. Kun vaiheen suunnitelmaa muokataan löydettyjen suunnittelumallien perusteella, saavutetaan tällä suunnitelmiin suunnittelumallien antamia hyötyjä. Tästä näkökulmasta katsoen suunnitelmien muokkaamien tunnistettujen suunnittelumallien avulla auttaakin saavuttamaan parempilaatuisia suunnitelmia.

Kuviossa 12 esiintyvässä kohdassa 'Etsi sopiva suunnittelumalli' voidaan käyttää avuksi luvussa 3 tehtyä luokittelua. Luokittelun perusteella voidaan rajata käyttökelvottomat suunnittelumallit pois ja tämän jälkeen käyttää seuraavia askelia sopivan suunnittelumallin löytämiseksi. Askeleet suunnittelumallin löytämiseksi on muokattu Buschmannin ym. (1996) ja Gamman ym. (1995) mukaan:

1. Määrittele ongelma.
2. Määrittele konteksti.
3. Valitse sopivat suunnittelumallit.
4. Tutustu suunnittelumallien kuvauksiin.
5. Valitse sopivin.

6. Yritä ratkaista ongelma suunnittelumallin avulla.
7. Mikäli suunnittelumalli ei ratkaise ongelmaa, valitse seuraavaksi sopivin.
8. Mikäli mikään suunnittelumalli ei sopinut, kokeile ongelman uudelleenmäärittelyä.
9. Mikäli et löydä sopivaa suunnittelumallia, käytä oliomenetelmän antamia ohjeita ongelman ratkaisemiseksi.

On kuitenkin huomattava, etteivät suunnittelumallit ratkaise kaikkia ongelmia. Jokainen rakennettava järjestelmä on ainutkertainen omine ongelmineen ja omine ratkaisuineen. Ainakin tulisi olla niin, koska ei ole kovin järkevää rakentaa samanlaista järjestelmää kahta kertaa, sillä kopiointi on halpaa. Tavoitteena tulisikin olla, että ajan kuluessa jokaisessa yrityksessä aletaan tuottaa omia suunnittelumalleja, jotka ovat kunkin yrityksen ohjelmistoille ominaisia. Tämän vuoksi tarvitaan myös menetelmä suunnittelumallien tuottamiselle.

5.4 Ohjelmistotuotanto suunnittelumalleja löytäen

Systemaattinen uudelleenkäyttö vaatii hyödyllisesti tallennettuja ja dokumentoituja tietoja uudelleenkäytettävistä komponenteista. Yleinen harhaluulo on, että näitä komponentteja voidaan vain noukkia olemassa olevista järjestelmistä ja ainoa ongelma niiden käytölle olisi vain oikean komponentin löytäminen. Kuvitellaan myös, että uudelleenkäyttö on halpaa ja helppoa. (Sommerville 1992.)

Tosiasiassa uudelleenkäytettävien komponenttien, ja tässä tapauksessa suunnittelumallien löytäminen, soveltaminen ja käyttäminen on kaikkea muuta kuin helppoa ja halpaa. Uusien suunnittelumallien löytämiseksi käytetään termiä *mallinkaivaminen* (pattern-mining) (Buschmann ym. 1996). Mallinkaivamisen tarkoituksen on löytää ja kuvata mahdollisia kandidaatteja suunnittelumalleiksi. Suunnittelumallikandidaattien löytämiselle käytetään seuraavia askeleita:

1. Löydä ainakin kolme esimerkkiä, joissa toistuva ongelma on ratkaistu samalla suunnittelulla tai toteutuksella.
2. Tee ratkaisukaava. Abstrahoi konkreetit ratkaisut niin, että ratkaisun ydin voidaan ymmärtää.
3. Tee ratkaisusta suunnittelumallikandidaatti.
4. Järjestä kandidaatille kirjoituspaja (ks. sivu 56).
5. Käytä kandidaattia todellisessa ohjelmistoprojektissa todelliseen ongelmaan.
6. Totea kandidaatti suunnittelumalliksi, jos soveltaminen on menestyksellistä.

Mikäli kandidaatin käyttäminen epäonnistui, selvitä syy, miksi näin kävi. Paranna kandidaattia huomioimaan epäonnistuneet tilanteet, ja yritä uudelleen. Vaihtoehtoisesti hylkää kandidaatti ja yritä löytää parempi suunnittelumallikandidaatti ratkaisemaan ongelma. (Buschmann ym. 1996.)

Kirjallisuudessa esiintyy Buschmannin ym. (1996) lisäksi vähän muita esityksiä suunnittelumallien löytämiseksi. Vlissides (1995) ja Shull ym. (1996) ovat esittäneet omat lähtökohtansa suunnittelumallien löytämiseksi. Shullin ym. (1996) työ perustuu Vlissidesin (1995) esitykseen, ja Shull ym. ovat vieneet Vlissidesin ohjeellisia tapoja löytää suunnittelumalli hieman eteenpäin.

Shull ym. (1996) ovat tunnistaneet kuusi vaihetta suunnittelumallien löytämiseksi olemassa olevista ohjelmistoista. Nämä vaiheet ovat:

1. Lue ohjelmiston ongelmamäärittely ja suunnitteludokumentit. Koodia ei kannata tutkia ennen kuin on olemassa tarpeeksi tietoa siitä, mitä järjestelmän on tarkoitus tehdä. Ongelmamäärittelyn tutkimisella tunnistetaan järjestelmälle asetetut rajoitukset ja järjestelmän tukema toiminnallisuus.
2. Tee järjestelmästä alustava luokkamalli luokkamäärittelyjen perusteella. Järjestelmän luokkamäärittelyt tarjoavat selkeimmän kuvan käytetyistä luokista. Luokkamäärittelyjen perusteella on helppo havaita perintäsuhteet.
3. Tarkenna luokkamallia tarkastelemalla luokkien toteutusta. Luokkien toteutuksesta voidaan havaita muutkin kuin vain perintäsuhteet. Samalla voidaan luokkamalliin määrittää luokkien välisten suhteiden kardinaalisuus, eli onko kyseessä monesta moneen suhde tai jokin muu.
4. Tunnista mahdolliset suunnittelumallikandidaatit perinnän ja kommunikoinnin perusteella käyttäen hyväksi edellisessä vaiheessa tuotettua luokkamallia. Tässä vaiheessa pyritään tunnistamaan, toistuvatko tietyt rakenteet luokkamallissa. Seuraavat tapaukset ovat osoittautuneet hyödyllisiksi tunnistettaessa suunnittelumalleja:
   - Luokka, jolla on suhteita useisiin muihin luokkiin ja joka toimii kommunikointisuhteiden vastaanottavassa päässä, saattaa määritellä muiden luokkien käyttäytymistä.
   - Luokka, jolla on suhteita useisiin muihin luokkiin ja joka toimii kommunikointisuhteiden lähettävässä päässä, saattaa toimia rajapintana muille luokille.
   - Luokat, joilla on *paralleeliperintää* eli jokainen aliluokka on suhteessa jonkin toisen luokan aliluokkaan, työskentelevät todennäköisesti läheisessä yhteistyössä.

- Luokka, joka yhdistää kaksi "luokkarypästä", saattaa toimia kommunikoivassa roolissa luokkaryppäisiin nähden.

5. Analysoi suunnittelumallikandidaatit. Tähän vaiheeseen ei vielä ole luotu kunnollista kriteeristöä ja vaihe vaatiikin huomattavasti työtä. Osavastaus voidaan saada siitä, kuinka hyödyllisiä suunnittelumallit ovat tälle ongelma-alueelle ja onko suunnittelumallikandidaateista hyötyä myöhemmässä vaiheessa.
6. Haastattele järjestelmän toteuttaneita suunnittelijoita ja ohjelmoijia ja tarkenna suunnittelupäätöksiä. Järjestelmän alun perin toteuttaneet suunnittelijat ja ohjelmoijat tuntevat kuitenkin parhaiten toimintaympäristön ja toiminnalle asetetut rajoitteet.

Kandidaatin löydyttyä täytyy käyttää jotakin formaalia menettelytapaa, jolla kandidaatti 'korotetaan' suunnittelumalliksi. Yksi yleinen tapa tällaiseksi menettelytavaksi on kirjoituspaja (writer's workshop)[6] (Coplien & Schmidt 1995, Buschmann ym. 1996). *Kirjoituspajassa* jokainen suunnittelumallikandidaatti käy läpi seuraavanlaiset vaiheet (Buschmann ym. 1996):

1. Suunnittelumallista keskustellaan ryhmässä, johon osallistuvat suunnittelumallin kirjoittaja ja ryhmä suunnittelumallin taustan ja sovellusalueen tuntevia tarkastajia. Mukana on myös kirjoituspajaprosessin vaiheet tunteva ohjaaja.
2. Suunnittelumallin kirjoittaja lukee sellaisen kohdan mallin kuvauksesta, jonka osallistujat haluavat kuulla.
3. Kaksi tarkastajaa tekee yhteenvedon kuulemastaan, kuten he asian ymmärsivät.
4. Ensiksi keskustellaan suunnittelumallin vahvoista puolista ja seuraavaksi heikoista. Tämän jälkeen keskustellaan kaikista muista mahdollisista seikoista, jotka tulevat tarkastajille mieleen mallin kuvauksesta. Keskustelun aikana mallin kirjoittaja on kuuntelijana ja kirjoittaa keskustelun tuloksena syntyneet ajatukset. Keskustelijoiden tulee puhua suunnittelumallista, niin kuin sen kirjoittaja ei olisi paikalla.
5. Keskustelun jälkeen kirjoittaja voi kysyä keskustelijoilta tarkentavia kysymyksiä.
6. Kirjoittaja päättää keskustelun loppukommentilla.

Kun uusia suunnittelumalleja löytyy, täytyy ne luokitella olemassa olevan luokittelun mukaan. Uuden suunnittelumallin suhteet olemassa oleviin suunnittelumalleihin täytyy määritellä. Uusi suunnittelumalli on myöskin kuvattava jollakin tavoin, tähän voi käyttää mm. Meszaroksen ja Doblen (1998) esittelemää mallijärjestelmää.

[6] Tätä tapaa mm. Pattern Language of Program Design (PLoP) –konferenssissa. PLoP-konferenssi on erityisesti tarkoitettu suunnittelumalleja varten.

5.5 *Suunnittelumallien käyttökokemuksia*

Olio-ohjelmistoissa esiintyy oliokeskeisyydelle tyypillisiä perusrakenteita. Nämä perusrakenteet toimivat tarkoitukseensa lähes poikkeuksetta (vrt. Zimmerin löydöksiä kohdassa 3.3). Suunnittelumallien eräs tärkeimpiä tarkoituksia on, että ne antavat eksplisiittisesti ratkaisun, joka suunnitelmiin sisältyy implisiittisesti (Schmidt 1995). Suunnittelumallit tekevät suunnitelmista joustavia, elegantteja ja lopulta uudelleenkäytettäviksi sopivia (Helm 1995).

Suunnittelumallien käyttökokemuksista todellisissa ohjelmistoprojekteissa ovat raportoineet mm. Riehle ja Züllighoven (1996), Schmid (1995), Schmidt (1995), Brown (1998), Schmidt ja Stephenson (1995), Helm (1995), Beck ym. (1996), Johnson (1992), Zimmer (1995b) ja Prechelt ym. (1998). Taulukossa 5 on yhteenveto Beckin ym. (1996) kokemuksista suunnittelumallien käytöstä. Nämä kokemukset kuvaavat hyvin kaikkia edellä mainituissa lähteissä esiin tulleita asioita.

Suunnittelumallit ...	FCS	AT&T	Motorola	BNR	Siemens	IBM
ovat hyvä kommunikoinnin väline	X	X	X	X	X	X
löydetään olemassa olevista suunnitelmista	X	X	X	X	X	X
kuvaavat suunnitelman ytimen	X	X	X	X	X	X
auttavat jakamaan parhaat käytännöt	X		X	X	X	X
eivät ole välttämättä oliosuuntautuneita		X	X	X	X	
pitäisi esitellä tutoreiden avulla	X				X	X
ovat vaikeita/aikaa vieviä kirjoittaa			X	X	X	
eivät synny ilman harjoitusta					X	X

Taulukko 5. Suunnittelumallien käyttökokemukset (Beck ym. 1996)

Yksi suurimpia hyötyjä on kommunikoinnin paraneminen (mm. Beck ym. 1996, Helm 1995, Schmidt 1995, Schmidt & Stephenson 1995). Kommunikointi paranee niin projektiryhmän sisällä kuin vuorovaikutuksessa muiden projektiryhmien ja ulkoisten sidosryhmien kanssa. Sisäisestä kommunikoinnista tulee tehokkaampaa, koska suunnittelumallin nimellä voidaan tarkoittaa laajempia kokonaisuuksia. Projektiryhmän käyttämät termit ovat samoja ja näin vältytään väärinkäsityksiltä ja virheiltä. Mikäli projekti on iso ja järjestelmän toteuttaminen on jaettu usean projektiryhmän kesken, voidaan tietoa jakaa tehokkaammin eikä järjestelmän ulkopuolisista osista jää vain mustalaatikkokuva.

Toinen suuri hyöty on, että kokemusta suunnittelusta voidaan välittää kokemattomille suunnittelijoille (mm. Beck 1996, Gamma ym. 1995, Helm 1995, Brown 1998). Brooks

(1987) toteaa, että yksi keino nostaa ohjelmistotuotannon tuottavuutta on kouluttaa hyviä suunnittelijoita. Tutkimukset osoittavat, että ohjelmoijien välillä voi olla jopa kymmenkertaisia eroja tuottavuuden suhteen (Cambell ym. 1992). Suunnittelumallit luovat mahdollisuuden välittää hyvien suunnittelijoiden tietämystä kokemattomammalle suunnittelijalle. Mikäli kokematon suunnittelija voi oppia näistä kokemuksista ja parantaa näin suunnitelmiensa laatua, jää häneltä aikaa kehittää omia mallejaan ja oppia enemmän suunnittelusta lyhyemmässä ajassa. Lisäksi, koska suunnittelumallit ovat käytännössä toimivia ratkaisuja, voidaan huomattavasti vähentää virheiden määrää suunnitelmissa.

Kolmas huomattava etu suunnittelumallien käytössä on uudelleenkäyttö. Koko suunnittelumallien ajatus perustuu mallien uudelleenkäyttöön, vaikka on nähtävissä kovaakin kritiikkiä suunnittelumallien uudelleenkäytöstä (vrt. Menzies 1997). Kuitenkin moinen kritiikki voidaan jättää omaan arvoonsa, koska suunnittelumalleja on käytetty menestyksekkäästi ongelmien ratkaisuun useammassa eri ohjelmistossa, ennen kuin niitä nimitetään suunnittelumalleiksi. Jokaista suunnittelumallia on käytetty menestyksekkäästi hyväksi ainakin kolmen eri ohjelmiston suunnittelussa. Suunnittelumallit ovat abstraktimmalla tasolla kuin mallin konkreettinen toteutus, mutta juuri tämä abstrahointi antaa mahdollisuuden ymmärtää ongelmanratkaisun ydin, eikä näin ollen suunnittelumallin tarjoama ratkaisu jää pelkäksi toteutuksen kopioimiseksi. Suunnittelumallit uudelleenkäyttävät ideoita ja mentaalista työtä, näin ne tarjoavat mahdollisuuden tehdä yksinkertaisia ja tyylikkäitä suunnitelmia. Kun ongelmaan on löytynyt sopiva suunnittelumalli, , herää kysymys, kuinka tämän olisi muuten voinutkaan tehdä.

Suurimpana haittana suunnittelumallien käytössä nähdään yli-innokkuus mallien käyttöön. Kun sinulla on vasara, kaikki ongelmat näyttävät nauloilta. Suunnittelumallien käytössä ei ole olennaista se, kuinka useaa mallia olet käyttänyt vaan se, että eteen tulleet ongelmat ovat soveliaita ratkaistavaksi suunnittelumallien avulla. Täytyy muistaa, että mallit tarjoavat ratkaisun vain tietyssä kontekstissa. Pyrkimys käyttää suunnittelumalleja jokaisessa mahdollisessa tilanteessa luo ylimääräistä kompleksisuutta suunnittelussa. Yli-innokkuus saa suunnittelijan näkemään kohdejärjestelmän suunnittelumalleina, kun tarkoituksena olisi ratkaista ongelmia suunnittelumallien avulla.

Yli-innokkuus johtaa väistämättä myös siihen, että suunnittelutason suunnittelumallit nähdään ainoina oikeina suunnittelumalleina. Suunnittelumallien perimmäinen tarkoitus on vain tarjota nimetty ratkaisu toistuvaan ongelmaan tietyssä kontekstissa. Alexander käytti suunnittelumalleja koko kaupungin kuvaamiseen, aina tiestön suunnittelusta yksittäisen huoneen sisustukseen (Alexander ym. 1977, Alexander 1979). Mikäli suunnit-

telumalleista halutaan niiden suurin hyöty, tulee dokumentoida ja kuvata suunnittelumalleja, jotka koskevat koko ohjelmiston elinkaarta ja siihen liittyviä tehtäviä.

Suunnittelumallit saattavat antaa suunnittelijalle kuvan, että he ymmärtävät itse ratkaisusta enemmän, kuin he tietävätkään (Schmidt 1995). Kuitenkaan suunnittelumallit eivät sisällä muuta kuin ongelman ratkaisun ytimen. Yhä edelleen saattaa jäädä toteutuskielikohtaisia ongelmia, joihin suunnittelumallit eivät ota kantaa. Vaikka mallit sisältävät esimerkkiratkaisun, niin lopullisen toteutuksen määrästä suunnittelumallin antama koodi on pieni osa (Helm 1995). Suunnittelumallin sisältämä koodi keskittyy ratkaisemaan olioiden välisiä kommunikaatiosuhteita, ja kuinka nämä kommunikaatiosuhteet ratkaisevat ongelman. Sovelluskohtaisen toiminnan koodin tulee kuitenkin perustua keskeisiltä osiltaan suunnittelumallin yhteydessä esitettyyn koodiin.

Etujen ja haittojen lisäksi on käyttökokemuksista löydettävissä myös hyödyllisiä vinkkejä, nyrkkisääntöjä, jotka kannattaa pitää mielessä, kun käytetään suunnittelumalleja. Näitä vinkkejä ovat seuraavat (Schmidt 1995):

1. Suunnittelumallien kuvausten tulisi sisältää konkreettisia esimerkkejä.
2. Suunnittelumallien nimien tulee olla tarkasti valittuja ja niitä tulee käyttää johdonmukaisesti.
3. Keskity kehittämään suunnittelumalleja, jotka ovat strategisia ongelma-alueelle, ja uudelleenkäytä olemassa olevia taktisia suunnittelumalleja.
4. Vakiinnuta suunnittelumallien palkkiosysteemi.
5. Käytännölliset suunnittelumallit tulevat käytännön kokemuksen kautta.
6. Suunnittelumallit helpottavat olioteknologian käyttöönottoa.
7. Suunnittelumallit varmennetaan kokemuksen, ei testauksen, kautta.
8. Suunnittelumallien kehittäjien tulee olla suorassa yhteydessä sovelluksen kehittäjien ja ongelma-alueen asiantuntijoiden kanssa.
9. Suunnittelumallien integrointi ohjelmistonkehitykseen on ihmistyötä vaativa toiminto.
10. Suunnittelumallit kuvaavat eksplisiittisesti voimat ja minkälaisia vaihtokauppoja joudutaan tekemään, mikäli valitaan tietty ratkaisu.
11. Dokumentoi tarkasti kontekstit, joissa suunnittelumalli toimii ja joissa se ei toimi.
12. Onnistunut suunnittelumallin kuvaus kattaa sekä rakenteen että käyttäytymisen.
13. Suunnittelumallit helpottavat uusien kehittäjien koulutusta.
14. Suunnittelumallien tehokas toteutus vaatii kielen ominaisuuksien tarkan valinnan.
15. Suunnittelumallit helpottavat pääsemään yli ohjelmointikielikeskeisestä ajattelusta.
16. Realistiset odotukset ovat oikea lähtökohta.

Lähempää tarkastelua vaativat kohdat 3 ja 6. Kohdassa 3 todetaan, että kannattaa keskittyä strategisiin, ei taktisiin suunnittelumalleihin. *Taktisilla* suunnittelumalleilla tarkoitetaan esimerkiksi Gamman ym. (1995) ja Buschmannin ym. (1996) suunnittelumalleja. Näillä suunnittelumalleilla on tarkoitus ratkaista lähinnä paikallisia ongelmia (vrt. Iterator, Gamma ym. 1995). *Strategiset* suunnittelumallit ovat yrityksen kannalta kilpailutekijä, eikä niitä julkaista.

Luvussa neljä esitelty luokittelu jaotteli suunnittelumallit kolmeen eri tasoon: idiomeihin, suunnitteluvaiheen malleihin ja arkkitehtuurimalleihin. Jokaiselta tasolta on periaatteessa löydettävissä strategisen tason suunnittelumalleja, mutta varsinkin idiomit ovat sen luonteisia, ettei niistä ainakaan helposti ole tehtävissä strategisen tason malleja. Kahden muun tason malleilla voidaankin sitoa sellaista pysyvää pääomaa suunnitelmiin, jota muiden on vaikea kopioida omissa tuotteissaan. Suunnitteluvaiheen malleja voidaan käyttää poikkeavalla tavalla, jolloin voidaan luoda omalle tuotteelle ylivoimaisia ominaisuuksia tietyllä alueella.

Resursseja ei kannata kuluttaa taktisten suunnittelumallien tekemiseen, vaan kannattaa keskittyä oman sovellusalueen keskeisiin ongelmiin. Uudelleen käyttämällä näitä hyvin tunnettuja ja dokumentoituja taktisia suunnittelumalleja jää enemmän aikaa keskittyä oman sovelluksen ydinongelmien ratkaisuun. Toisaalta myös keskittyminen itselle tärkeisiin suunnittelumalleihin vähentää suunnittelumallien lukumäärän kasvamista (vrt. Schmidt 1995).

Kohdassa 6 todettiin, että suunnittelumallit helpottavat oliolähestymistavan käyttöönottoa. Moneen kertaan on jo todettu, kuinka suunnittelumallit helpottavat ymmärtämään oliosuunnittelua. Helpompi opettelu saavutetaan useasta eri syystä: (a) suunnittelumallit dokumentoivat eksplisiittisesti ongelman ratkaisun ytimen, (b) suunnittelumallit dokumentoivat toimivia ratkaisuja, jotka eivät ole ilmiselviä, ja (c) suunnittelumallit helpottavat kommunikointia. Suunnittelumalleissa ei ole tärkeää se, mitä ne sisältävät, vaan kaikki se tietotaito ja suunnittelukokemus, joka on saatu kerättyä niihin. Kun olioteknologia otetaan käyttöön, niin suunnittelumallit helpottavat aluksi ymmärtämään, miksi tietyt ratkaisut ovat parempia kuin toiset. Hieman pidemmälle ehtineet alkavat nähdä suunnittelumalleissa toistuvia ratkaisuja, joita ovat mm. delegointi ja abstrahointi, joiden avulla saavutetaan suunnitelmien joustavuus (vrt. kohta 3.3 ja Pree 1995). Suunnittelumalleista löytyy aina uusia puolia, kun niitä käytetään uudestaan[7] (Helm 1995).

[7] Christopher Alexander on kirjoittanut, että jokainen suunnittelumalli kuvaa ongelman, joka esiintyy yhä uudelleen ja uudelleen ympäristössämme, ja kuvaa ratkaisun ytimen ongelmaan niin, että ratkaisua voi käyttää miljoona kertaa ilman, että tekee toteutuksen kahdesti samalla tavalla (Alexander ym. 1977).

Käyttökokemuksista nousee esiin myös tarve suunnittelumallien ja oliomenetelmien yhteensovittamiseen. Suunnittelumallit perustuvat käytännön kokemukseen ja ovat todistettavasti toimivia ratkaisuja tiettyyn ongelmaan. Mallit eivät kuitenkaan kerro, mitä tehdään seuraavaksi. Menetelmät kuvaavat tarkasti sen, mitä pitää tehdä, missä järjestyksessä ja miten, mutta menetelmät eivät sisällä ratkaisuja ongelmiin. Pikemminkin menetelmät kuvaavat, kuinka ongelmia voidaan kuvata ja rajata. Suunnittelumalleja ja menetelmiä ei tämän vuoksi ole yhdistetty ja tarvitaan lisää tutkimustyötä, jotta voitaisiin tehdä menetelmä, joka eksplisiittisesti kuvaisi suunnittelumallien käytön ohjelmistotuotannossa. (vrt. Helm 1995, Schmidt 1995, Jaaksi 1997)

5.6 Yhteenveto

Tässä luvussa esiteltiin suunnittelumallien käyttäminen ohjelmistotuotannossa. Aluksi todettiin, ettei ole vielä olemassa hyvää ohjelmiston kehitysmenetelmää, johon suunnittelumallit olisi otettu mukaan. Tämän vuoksi tarkastelun lähtökohdaksi otettiin Sommervillen (1992) ja Sametingerin (1997) esittelemät yleiset lähestymistavat komponenttien uudelleenkäyttöön. Sommervillen lähestymistapaa muokattiin niin, että otettiin huomioon neljännessä luvussa valittu Buschmannin ym. (1996) luokittelu. Tässä mallissa jokaiseen ohjelmistotuotannon vaiheeseen kuuluu suunnittelumallien huomioiminen.

Toiseksi luvussa esitettiin, kuinka suunnittelumalleja voi löytää ohjelmistosuunnitelmista. Löytämisen apuna käytettiin suunnittelumallikirjallisuudesta eteen tulleita askelia taikka vaiheita. Suunnittelumallit ovat käytännössä hyväksi havaittuja ratkaisuja, ja kohdassa 5.4 esitettiin, mitä tutkimustuloksia suunnittelumallien käytöstä on tullut. Lopuksi selvitettiin vielä suunnittelumallien käyttökokemuksia

6 OLIOLÄHESTYMISTAVAN OPPIMINEN SUUNNITTELUMALLIEN AVULLA

Kuten edellisessä luvussa todettiin, suunnittelumallit sisältävät käytännön kokemusta ja kuvaavat toimivan ratkaisun todelliseen ongelmaan. Suunnittelumalleja opiskelemalla ja tutkimalla voidaan oppia kokeneiden suunnittelijoiden ratkaisuja. Näin voidaan siirtää kokeneempien ihmisten tietotaitoa vähemmän kokeneille ihmisille. Tässä luvussa selvitetään, kuinka olioparadigmassa eksperttien tietotaitoa voidaan helpommin siirtää noviiseille. Ensiksi selvitetään kognitiivisten tieteiden käsitystä siitä, kuinka ihmisen mieli rakentuu. Tähän perustuen selvitetään, miksi olioparadigma helpottaa järjestelmien rakentamista ja mitä tarkoittaa, että ihminen on ekspertti oliosuuntautuneisuudessa. Ennen kuin voidaan esittää, kuinka noviisit voivat siirtyä eri vaiheiden tai tasojen kautta eksperteiksi, täytyy oliosuuntautuneesta lähestymistavasta tunnistaa eri tasot ja kuinka näiden tasojen välillä voidaan liikkua. Lopuksi esitetään olioparadigman kokemuksellinen oppiminen.

6.1 Skeemat ja skriptit

Kognitiotiede on syntynyt monien tieteiden leikkauskohtaan. Filosofia, kielitiede, psykologia, tietojenkäsittelytiede, tekoälyn tutkimus ja neurotieteet ovat jo vakiinnuttaneet etupiirinsä kognitiotieteen sisällä. Kaikkia kognitiotieteen taustatieteitä yhdistää pyrkimys ymmärtää omalla alueellaan symbolista informaation käsittelyä, mikä liittyy joko suoraan tai epäsuorasti tietokoneiden käyttöönottoon näissä tieteissä. Kognitiotieteen peruskäsitteeksi onkin näin nousemassa systeemi, joka käsittelee symboleja. Tämä käsite on tietoinen sovellutus Turingin koneesta. (Saariluoma 1988a, Eysenck & Keane 1990.)

Kognitiotieteen esikuvana on ollut ihmisen oma mieli. Tarkkailemalla mielen toimintoja on luotu käsitys informaation esittämisestä ja prosessoinnista. Erityisen tärkeitä ovat muistin ja ajattelun tutkimus. Muistintutkimuksen problematiikassa tiedon esittäminen nousee tarkastelun kohteeksi. Immanuel Kant (1781, A 181) esitteli ajatteluprosessiin tärkeän uuden käsitteen, skeeman. Skeemateoria on suosittu modernissa psykologiassa ja tärkeä teoria ihmisen informaation prosessoinnin tutkimuksessa. (Saariluoma 1988a.)

Skeemalla tarkoitetaan abstraktia tietorakennetta, jossa muistettavan asian eri elementtien järjestys on kuvattu (Saariluoma 1988b). Skeemat ovat malleja maailmasta. Ne edustavat kohteiden, tilanteiden, tapahtumien ja toimintojen prototyyppejä (Rumelhart 1977). Skeemoja ovat mm. kasvot, eläin, häät tai urheilukilpailut. Esimerkiksi kasvojen skeema luodaan toisiinsa liittyvien silmien, nenän ja suun avulla, ja näiden väliseen suhteeseen liittyy vielä, ettei esimerkiksi suu voi olla silmien yläpuolella. Yksilöiden välillä saattaa olla eroja yksityiskohdissa, mutta kaikkien kasvot noudattavat edellä mainittuja pääsääntöjä. Urheilukilpailujen skeema sisältää kilparadan, urheilulajit ja urheilijan skeemat tietyissä suhteissa toisiinsa.

Skeemat siis edustavat tietoa, ja ne myös toimivat suodattimina tiedon palautukselle mieleen. Skeemat auttavat hallitsemaan tilanteiden ja tapahtumien suurta variaatiota yleistämällä ja luokittelemalla näitä tapahtumia. Ihmisen rajoitetun tiedonkäsittelykapasiteetin huomioonottaen skeemat auttavat ihmistä kiinnittämään huomionsa siihen, mikä on yleistyksestä poikkeavaa. Näin voidaan hallita erilaisia tapahtumia ja tehdä päätöksiä pienienkin eroavaisuuksien pohjalta. Skeeman ja maailman välisiä eroja verrataan muuttujien avulla. Jokainen skeema sisältää useita muuttujia. Muuttujien oletusarvo on tosi. Skeemaa verrataan maailmaan ja mikäli huomataan, että jokin skeeman muuttuja onkin epätosi, osataan keskittää huomio tähän poikkeavaan seikkaan. Skeemat voivat muuttujien lisäksi sisältää myös aliskeemoja. Mikäli aliskeema ei pidä paikkaansa, voidaan huomio kiinnittää tähän yksittäiseen aliskeemaan. (Eysenc & Keane 1990.)

Skeemat ovat luonteeltaan joko abstrakteja tai konkreetteja. Konkreetti skeema on esimerkiksi ihminen, ja abstrakti skeema suhde tai ystävyys. Skeeman ja sen sisältämän aliskeeman ei tarvitse olla samaa tyyppiä, joten aliskeema voi olla konkreetti, vaikka se kuuluisi abstraktiin skeemaan. Skeemojen koostaminen tehdään aina niin, ettei abstraktimpi skeema ole konkreettisemman skeeman osana. Sisällyttämällä skeemaan aliskeemoja voidaan muodostaa koostumissuhteita. Esimerkiksi ystävyys muodostuu kahden ihmisen välille. Ihminen voidaan tässä käsittää konkreetiksi skeemaksi ja ystävyys ja suhde abstrakteiksi skeemoiksi. Skeemat voivat koskea myös eri tasolla olevia käsitteitä. Näin voidaan skeemassa siirtyä yleisestä yksityiskohtaisempaan ja päinvastoin. (Eysenc & Keane 1990.) Skeemat aliskeemoineen muodostavat ihmisen muistin (Piaget & Inhelder 1977, Piaget 1988).

Skeeman käsite on hyvin keskeinen modernissa kognitiivisessa psykologiassa ja on osoittautunut hyvin selitysvoimaiseksi (Saariluoma 1988a). Tästä johtuen skeemakäsitteeseen on ladattu valtavasti erilaisia selityksiä ja skeeman käsite on näin ollen hyvin laaja (Eysenck & Keane 1990). Schankin ja Abelsonin (1977) skripti on erityinen skee-

marakenne, joka selittää, kuinka tapahtumasarjat ilmenevät. Skriptit omalla tavallaan ennustavat, mitä tapahtuu seuraavaksi. Voidaan yksinkertaistetusti sanoa, että skeemat esittävät asioiden ja tapahtumien suhteita, kun taas skriptit esittävät tapahtumien kulkua.

Skripti on rakenne, joka kuvaa tapahtumasarjan tietyssä kontekstissa (Schank & Abelson, 1977). Skripti on johdonmukainen tapahtumasarja, ja havainnoija odottaa, että tilanne menee oletetulla, skriptin kuvaamalla tavalla. Skripti muodostuu huokosista ja vaatimuksista, jotka voivat täyttää nuo huokoset. Rakenne on sidottu kokonaisuuteen, ja mikä on yhdessä huokosessa täytteenä, vaikuttaa siihen, mitä voi olla toisessa. Skriptit käsittelevät tyyliteltyjä jokapäiväisiä tilanteita. Ne eivät ole alttiita muutoksille, eivätkä ne muodosta kojeistoa täysin uudenlaisten tilanteiden käsittelemiseksi. Siten skriptit ovat ennalta määriteltyjä, stereotyyppisiä tapahtumasarjoja, jotka määrittelevät hyvin tunnettuja tilanteita. Skriptit sallivat uusia viittauksia asioihin tai käsitteisiin, aivan kuin nämä asiat tai käsitteet olisivat aikaisemmin mainittuja. Asiat skriptissä voivat ottaa määritellyn muodon ilman eksplisiittistä määrittelyä, koska skripti itsessään implisiittisesti on jo esitellyt asiat. (vrt. Abelson 1976.)

Seuraavaksi on kaksi esimerkkiä skeemoista:

1. Mika meni ravintolaan. Hän pyysi tarjoilijalta viinikukon. Mika maksoi laskun ja lähti.
2. Mika meni puistoon. Hän pyysi kääpiöltä hiiren. Mika poimi kukan ja lähti.

Ensimmäinen esimerkki on ymmärrettävä, koska se sisältää skriptin ravintolassa syömisestä. Mutta toiselle esimerkille ei ole olemassa skriptiä, joka yhdistäisi puiston, kääpiön, hiiren ja kukan.

Huolelliset skeema- ja skriptirakenteet voivat esittää monimutkaisia käsitteitä, tapahtumia ja suhteita johdonmukaisella ja tehokkaalla tavalla. Taulukkoon 6 on kerätty yhteen skriptien ja skeemojen ominaisuuksia.

Oliomallien staattisia ja dynaamisia puolia voidaan kuvata kognitiivisen tieteen käsitteillä skeemat ja skriptit. Skeemat ovat staattisia, ja skriptit ovat dynaamisia kuvauksia maailmasta. Liitäntä oliomallien ja kognitiivisen tieteen teorioiden välillä osoittaa, että oliomallit ovat lähellä kognitiivisen tieteen käsitystä maailmasta. (Villeneuve ja Fedorowicz 1997.)

Oliosuunnittelu tehdään staattisesta ja dynaamisesta näkökulmasta. Eri suunnittelumenetelmistä tämä näkyy kenties parhaiten OMT++-menetelmän staattisena ja dynaamise-

na polkuna (Jaaksi 1997). Olioiden välisiä suhteita ja olioiden ominaisuuksia kuvataan staattisella luokkamallilla. Olioiden sisäistä ja olioiden välistä käyttäytymistä kuvataan dynaamisin mallein, mm. herätekaaviolla ja tilakaaviolla.

Kognitiivinen rakenne	Ominaisuudet
Skeema	Hierarkkisesti järjestetty Attribuutin sisältämän arvon muoto Attribuuteilla on lista mahdollisista arvoista (arvoalue) Attribuuteilla on oletusarvo Attribuuttien arvot voivat olla riippuvaisia toisistaan Attribuuteilla on painoarvoa määrittämässä niiden suhteellista arvoa Skeemalla on viite siihen luokkaan, johon se kuuluu Skeema sisältää tiedon kontekstista Skeema sisältämä tieto suodatetaan skeeman luontivaiheessa Skeema esittää tietämystä kaikilla abstraktiotasoilla Skeemat voivat sisältää toisia skeemoja (aliskeemoja) Abstraktit skeemat esiintyvät hierarkian yläpäässä ja niillä on suppeampi attribuuttilista
Skripti	Toimintasuuntautunut tietämys rakenteista Skriptit järjestävät mentaalisen prosessin askeleet Jotkut ovat puhtaasti toimintaskriptejä, jotka automatisoivat motorisen käyttäytymisen Päätarkoitus on tapahtumien järjestäminen Tarkoituksena on saada skeema toimimaan Skriptit vaativat luontivaiheessaan suuresti huomiota Skriptit helpottavat valitsemaan sopivan skeeman Skriptit helpottavat varmistamaan, että oikea skeema on valittu

Taulukko 6. Yhteenveto skeeman ja skriptin rakenteista (Villeneuve & Fedorowicz 1997)

Oliot ovat samankaltaisia kuin skeemat (Villeneuve & Fedorowicz 1997). Molemmat tunnistavat luokat ja luokan ilmentymät (oliot). Jokaisella oliolla on määrittäviä attribuutteja. Olioluokat voidaan linkittää toisiin luokkiin assosiaatioiden avulla, jotka kuvaavat määrättyä suhdetta, suhteen tarkoitusta ja määräävät, kuinka monta oliota suhteen kummallekin puolelle kuluu. Yleistys on erityinen assosiaatio, joka mahdollistaa attribuuttien ja toiminnan periytymisen hierarkiassa. Alemmat hierarkiatasot sisältävät yksityiskohtaisempaa tietoa käsitteistä. Toinen suhdetyyppi, koostaminen, tarjoaa mahdollisuuden rakentaa olio toisista olioista. Olioiden koostaminen ja skeemateorian ra-

kentumiskäsite ovat samankaltaisia. Oliomalli sisältää luokkia ja luokkien välisiä suhteita, jotka heijastavat nykyistä ymmärrystä mallin esittämästä maailmasta. Taulukoissa 7 ja 8 on esitetty skeemojen ja skriptien suhdetta staattiseen ja dynaamiseen oliomalliin.

Oliokeskeisyys tunnistaa, että esiintyvät tapahtumat muuttavat arvoja olioinstansseissa. Dynaaminen malli kuvaa odotetut tapahtumat ja tapahtumien jälkeen tulevan tilan. Dynaaminen malli perustuu sarjalle skenaarioita, jotka puolestaan kuvaavat yksityiskohtaisen tapahtumien etenemisen ja tulokset, joita saadaan ajan mukana. Skenaariot ovat hyvin samankaltaisia skriptien kanssa. Dynaaminen malli kuvaa, kuinka skenaariot sitoutuvat toisiinsa.

Käsite	Skeema	Oliot
Yksittäinen ilmentymä	Skeema	Instanssi
Samankaltaisten ilmentymien ryhmä	Luokka	Luokka
Erilaisia ominaisuuksia sisältävien ilmentymien ryhmä	Kompositio	Kooste (aggregation)
Kuvaava ominaisuus	Attribuutti	Attribuutti
Perintäsuhde	Perintä	Yleistäminen
Olioiden väliset yhteydet	Toisiinsa liittyvät skeemat	Assosiaatio

Taulukko 7. Skeemojen ja oliomallin vertailu (Villeneuve & Fedorowicz 1997)

Käsite	Skripti	Dynaaminen malli
Yksittäinen tapahtumasarja	Skripti	Skenaario
Tapahtumasarjojen ryhmä	Skriptit	Dynaaminen malli
Aloituksen heräte	Tapahtuma	Tapahtuma
Tarkoitus	Saada skeema toimimaan	Kuvata olion ajasta riippuvat tilat

Taulukko 8. Skriptien ja dynaamisen mallin vertailu (Villeneuve & Fedorowicz 1997)

Skeemat ja skriptit selittävät, kuinka ihminen havainnoi maailmaa. Kuitenkaan skeemojen ja skriptien olemassaolo ei sellaisenaan selitä suorituseroja eksperttien ja noviisien välillä. Voitaisiin ajatella, että mitä suurempi määrä skeemoja on olemassa, sitä vaikeampi on valita oikea skeema käyttöön. Näin ei kuitenkaan ole. Kokemus auttaa tarkentamaan tietämystä, ja uusien kokemusten jälkeen voidaan aina parantaa skeemoja. Kun tarpeeksi moni skeema sisältää saman rakenteen, voidaan näistä tehdä korkeamman tason skeema, joka yleistää kaikki sen alapuolelle kuuluvat aliskeemat. (Villeneuve & Fedorowicz, 1997.)

6.2 Ekspertit ja noviisit

Kognitiivisissa tieteissä on tutkittu paljon ihmisten taitoja mm. ongelman ratkaisussa ja päätöksenteossa. Kuitenkin varsin vähän on saanut huomiota eksperttiys eri aloilla. Cooke (1992) on summannut yhteen eri tutkimuksissa paljastuneita eroja eksperttien ja noviisien välillä seuraavasti:

1. ekspertit ovat ylivoimaisia vain omalla alallaan,
2. ekspertit muodostavat laajoja ja merkityksellisiä malleja erikoisalastaan,
3. ekspertit ovat nopeita; he suorittavat noviiseja nopeammin oman erikoisalansa tehtäviä ja ratkaisevat ongelmia vähemmin virhein,
4. eksperteillä on parempi lyhyen ja pitkän keston muisti erikoisalallaan,
5. ekspertit havaitsevat ja esittävät oman erikoisalansa ongelmat tarkemmin ja yksityiskohtaisemmin kuin noviisit; noviisit esittävät ongelmat ylimalkaisesti,
6. ekspertit käyttävät pitkän ajan analysoimalla ongelmia kvalitatiivisin perustein ja
7. eksperteillä on hyvä kyky arvioida omaa suoritustaan.

Ekspertit ovat vuosien saatossa sisäistäneet erikoisalaansa liittyviä käsitteitä, joten he voivat suoraan keskittyä itse ongelmaan. Noviiseilla suuri osa kapasiteettista menee erikoisalan ymmärtämiseen ja vain osa voidaan käyttää ymmärtämään itse ongelmaa. Esimerkiksi tutkimustyötä tekevän professorin ei tarvitse miettiä tutkimusmenetelmiä ja tutkimukseen liittyviä eri vaiheita niin paljoa kuin graduaan kirjoittavan opiskelijan. Siinä missä gradun kirjoittaja joutuu jatkuvasti miettimään tieteellisen työn tekemistä ja kuinka edetä tutkimustyössä vaiheesta seuraavaan, ovat nämä professorilla sisäistettyjä asioita, eikä hänen tarvitse keskittyä niihin juuri lainkaan[8]. *Ekspertti* onkin ihminen, joka on usean vuoden kokemuksen kautta saavuttanut korkean suoritustehon omalla alallaan (Foley & Hart 1992). Henkilön eksperttiys voidaan määrittää käyttämällä mittareita siitä, kuinka paljon aikaa hän kuluttaa ongelman ymmärtämiseen ja esittämiseen ja kuinka kauan häneltä menee aikaa tehtävän suorittamiseen sekä minkälainen on ratkaisun laatu.

Dreyfusin ja Dreyfusin (1986) mukaan yksilö etenee noviisista ekspertiksi minkä tahansa taidon alueella kokemuksen karttuessa viiden tason kautta. Nämä tasot ovat:

1. Noviisi
2. Edistynyt aloittelija
3. Pätevä

[8] Tämä esimerkki tietenkin olettaa, että professori käyttää tuttua ja turvallista tutkimusmenetelmää.

4. Kyvykäs
5. Ekspertti

Kun noviisi yrittää päästä oikeaan lopputulokseen analyyttisesti harkiten, todellinen ekspertti toimii intuitiivisesti, eläytyy tilanteeseen ja nojaa kokemuksen tuomiin analogioihin ratkaisussaan. Asiantuntijan taito on muuttunut siinä määrin osaksi häntä, että hänen ei tarvitse olla siitä sen enempää tietoinen kuin hän on omasta kehostaan (Dreyfys & Dreyfys 1986, 30). Ero aloittelevan noviisin ja intuitiivisesti toimivan ekspertin välillä on huomattava. Ekspertit ovat luoneet vuosien opiskelun ja kokemuksen kautta sellaisen määrän yhteen nivoutunutta tietoa, että he pystyvät varsin lyhyessä ajassa valitsemaan oikean strategian ongelman ratkaisuun.

Eksperttien ja noviisien välisiä eroja voidaan ymmärtää paremmin eri abstraktiotason skeemojen olemassaololla. Mitä enemmän ihmisellä on tietoa ja mitä enemmän tiedot liittyvät toisiinsa abstrahoinnin kautta, sitä paremmin voidaan nähdä syy-seurausyhteyksiä eri kokonaisuuksien kesken. Eksperttiyttä voidaan kohottaa järjestelemällä skeemoja uudelleen ja lisäämällä kykyä käyttää skriptejä. Vastaavasti skeemat ja skriptit auttavat määrittämään päättelystrategioita eli strategioita muistista palauttamiselle, ongelman ymmärtämiselle ja ratkaisulle. (Villeneuve & Fedorowicz 1997.)

Ekspertit käyttävät skriptejä eri tavoin kun noviisit. Noviiseille skriptit ovat yksityiskohtaisia ohjeita, kuinka suorittaa tietty vaihe. Eksperteille skriptitkin muodostavat sisäkkäisiä hierarkioita, ja kunkin skriptin tietyssä vaiheessa voidaan soveltaa tilanteen vaatimaa menettelytapaa.

Eksperttien on helpompi liikkua yleisestä yksityiseen ja päinvastoin. He pystyvät näkemään, kuinka eri abstraktiotasot liittyvät toisiinsa ja kuinka muutokset tietyllä abstraktiotasolla vaikuttavat muihin abstraktiotasoihin. Tätä muutosta esimerkiksi koodin ja oliomallin välillä kuvaavat erilaiset suunnittelumallit.

6.3 Oliot ja ekspettiys

Ekspertit luovat hienojakoisia suhteita eri skeemojen välille. Eksperttitason muistisuoritukset perustuvat tuttujen mallien löytämiseen ongelma-alueesta, ja näin luokittelemalla tietoa. Ekspertit käyttävät erityisiä mielleyhtymiä, jotka parantavat skeemojen sopivuutta ja laatua kognitiivisiin prosesseihin. Näin vältetään umpimähkäiset kokeilut

eri skeemoilla. Olioparadigma toimii hyvin, kun käytetään assosiaatioita, yleistystä ja koostamista tehokkaasti. Lattea kokoelma olioita on vain pieni parannus, jos lainkaan, relaatiotietokantoihin nähden. Olioiden suhteet ja operaatiot olioiden välillä antavat ekspertin kontribuution, kontekstuaalisen syvyyden oliomallille, minkä perusteella voidaan erottaa ekspertin ja noviisin oliomallit toisistaan. (Villeneuve & Fedorowicz 1997.)

Oliokeskeisyys antaa myös välineet selittää monimutkaisen järjestelmän rakennetta. Noviisit pyrkivät usein mallintamaan koko järjestelmää konkreeteilla, alemman tason malleilla kuin ekspertit. Ekspertit pystyvät paremmin esittämään koko järjestelmän abstraktilla tasolla ja vasta myöhemmin keskittyvät kuvaamaan osajärjestelmiä tarkemmalla tasolla. Kuten edellä on mainittu, abstraktiohierarkiaa voidaan esittää oliomallilla ja dynaamisilla malleilla. Eksperteillä on kyky navigoida eri abstraktiotasojen välillä ilman, että he menisivät sekaisin tästä.

Ekspertit käyttävät ongelmiin toisenlaisia ratkaisumalleja kuin noviisit. Ekspertit käyttävät enemmän aikaa ymmärtääkseen ongelman. Tämä ymmärtäminen muodostaa analyysivaiheen perustan. Oliosuuntautunut järjestelmän kehitys auttaa näin löytämään virheet mahdollisimman aikaisin, koska se kiinnittää huomion ongelma-alueen ymmärtämiseen analyysivaiheessa. Näin olioanalyysi auttaa havaitsemaan virheet aikaisemmin ja säästää rahaa.

Ekspertit käyttävät myös koottua tietoa ratkaistessaan ongelmaa. Käytännössä he käyttävät skriptejä ratkaisumekanismeina. Eksperteillä on skriptihierarkioita, jotka käyvät erikoistapauksiin tai vaihtoehtoisiin tapauksiin, samalla tavoin kuin dynaamiset mallit oliosuuntautuneisuudessa. (Villeneuve & Fedorowicz 1997.)

Oliosuuntautuneisuus pystyy auttamaan eksperttejä selittämään monimutkaisia päättelyjä antamalla hyvän välineen dokumentointiin, jossa näkyvät hienosyiset suhteet olioiden välillä. Noviisit tekevät konkreetteja, alemman tason malleja, kun taas ekspertit pystyvät päättämään kuinka ja milloin navigoida eri hierarkiatasojen välillä. Oliomallin tai dynaamisten mallien eksplisiittinen esitys auttaa tällaisessa prosessissa. (Villeneuve & Fedorowicz 1997.)

Ekspertin esittämän ratkaisun laadun ennustettavuus heijastuu oliosuuntautuneissa menetelmissä. Oliosuuntautuneisuus johtaa ideoiden ja suhteiden uudelleen käyttöön, koska se antaa voimakkaan esitystavan olioiden ja niiden käyttäytymisen esittämiseen. Se antaa myös olioille riippumattomuutta niin, että yksittäistä oliota voidaan muuttaa ilman, että se vaikuttaisi koko malliin. (Villeneuve & Fedorowicz 1997.)

6.4 Skeemat ja skriptit oliokeskeisyydessä

Eksperttianalysoijat käyttävät ajan kanssa luomaansa skeemahierarkiaa eteentulevien ongelmien ratkaisemiseen. Ekspertit muuntavat skeemaa mielessään, kunnes se vastaa ongelman ratkaisua. Ekspertit käyttävät skriptejä varmistaakseen valitun skeeman oikeellisuuden ja vakuuttuakseen, että heidän tilannetulkintansa on oikea. Vasta kun skeema ja todellisuus ovat hyvin lähellä toisiaan, siirrytään kuvaamaan ongelman ratkaisua. (Villeneuve & Fedorowicz 1997.)

Käytännössä ekspertti luo oliomallin ja dynaamisen mallin ongelman ratkaisusta ennen kuin alkaa miettiä implementointiin liittyviä seikkoja. Oliomenetelmät tarjoavat ekspertille työkaluja, joiden mukaan hän voi kuvata ja ymmärtää ongelman niin, ettei ratkaisutapaa tarvitse määrittää. Ekspertti järjestelee olioiden attribuutteja, operaatioita ja olioiden välisiä suhteita, kunnes nämä vastaavat hänen mielestään todellisuutta tarpeeksi hyvin. Oliomenetelmät eivät myöskään pakota suunnittelijaa käyttämään tietyntyyppisiä assosiaatioita. Lisäksi, koska maailma on tehty kompleksisten olioiden suhteista ja yhdistelmistä, oliosuuntautuneisuus antaa mahdollisuuden käyttää koosteita, ja näin suunnittelijat voivat tuottaa tarkempia ja hienojakoisempia malleja todellisuudesta. (Villeneuve & Fedorowicz 1997.)

Toinen oliokeskeisyyden antama hyöty on, että se antaa suunnittelijoille joustavuutta. Se tuottaa todennäköisemmin korkeatasoisia suunnitelmia. Ekspertit navigoivat toistuvasti tietovarastonsa läpi, kun yrittävät ymmärtää ongelmanratkaisutasoaan prosessissa. *Sen vuoksi he luonnostaan luovat kilpailevia skeemoja, jotka ohjaavat heitä kysymään lisää kysymyksiä. Näin he luovat tarkemman kuvan ongelmasta.* Kysymällä enemmän kysymyksiä he voivat laukaista lisäperusteluita ja tulkintoja käyttäjiltä, joilta he keräävät vaatimuksia, ja jälleen auttavat näitä tarkentamaan omaa ymmärrystään ongelmasta ja mahdollista ratkaisua. Yksi piirre eksperteissä on, että he luovat kokonaisvaltaisemman näkymän koko ongelmasta ja luovat korkeatasoisemman ratkaisun kuin noviisit, jotka luovat konkreetin ja rajoittavan näkemyksen ongelmaan. (Villeneuve & Fedorowicz 1997.)

Eksperttianalysoija voi tuottaa oliomallin, jota myöhemmin käytetään ratkaisuna samankaltaiseen ongelmaan. Oliokeskeisyyden puoltajat esittävät, että oliomenetelmät edistävät uudelleenkäytettävyyttä. Oliokeskeisyys auttaa eksperttiyden saavuttamista,

koska sillä voidaan dokumentoida ja jakaa mallinnustavat, jotka ekspertillä on ongelma-alueella. Noviisit voivat käyttää eksperttien tietoa omaksumalla olio- ja suhdemäärityksiä. Näin voidaan nopeuttaa noviisien oppimista, samoin kuin eksperttisysteemit nopeuttavat käyttäjiensä oppimista. (Villeneuve & Fedorowicz, 1997.)

Tässä kohdassa esitettiin Villeneuven ja Fedorowiczin (1997) löytämiä yhtäläisyyksiä skeemojen, skriptien ja oliokeskeisyyden välillä. Merkittävimmät huomiot ovat

1. Ekspertit luovat skeemat vastaamaan mahdollisimman hyvin todellisuutta.
2. Oliokeskeisyys antaa mahdollisuuden kuvata näitä skeemoja.
3. Ekspertit luovat keskenään kilpailevia skeemoja.
4. Ekspertit voivat käyttää olemassa olevia skeemoja ratkaistessaan vastaavia ongelmia.
5. Oliokeskeisyys voi tallentaa näitä ratkaisuja.
6. Noviisit voivat omaksua eksperttien tietoa omaksumalla olio- ja suhdemäärityksiä.

Suunnittelumalleja voidaan käyttää juuri edellä mainittuihin tilanteisiin: (1) ekspertit voivat tallentaa tietämystään myöhempiä tilanteita varten suunnittelumalleilla, koska suunnittelumallien ratkaisu dokumentoidaan; (2) suunnittelumallit ovat valmiita ja dokumentoituja ratkaisuja ongelmiin, koska ovat ratkaisseet onnistuneesti samanlaisia ongelmia; (3) suunnittelija voi kokeilla kilpailevia suunnittelumalleja tiettyyn ongelmaan, koska on useita dokumentoituja ratkaisuja, jotka ratkaisevat ongelman erilaisin vaihtokaupoin; ja (4) suunnittelumallien avulla voidaan dokumentoida eksperttien tietämystä ja siirtää sitä noviisien käyttöön, koska noviisit voivat lukea eksperttien kirjoittamia suunnittelumalleja.

6.5 Olio-ohjelmoinnin oppimistasot

Edellisessä kohdassa todettiin, kuinka noviisi voi nopeammin saavuttaa eksperttiyden suunnittelumallien avulla. Seuraavaksi tunnistetaan olio-oppimisen tasot, ja tätä lähestytään olio-ohjelmoinnin oppimisen kautta.

Cambell ja Brown (1992) tunnistivat Smalltalk-kielen oppimisessa seitsemän eri tasoa:

1. Visuaalisen käyttöliittymän oppiminen
2. Kielen syntaksin ja arvojärjestyksen oppiminen
3. Luokkien ja metodien tunnistaminen luokkahierarkiasta
4. Luokan ja instanssin erottaminen

5. Model-Pane-Dispatcher[9] yhdistelmän oppiminen
6. Oliosuuntautunut suunnittelu
7. (Suurmestari taso).

Tasot 1-4 voidaan yhdistää ja kutsua niitä aloittamiseksi. Aloittamisvaiheessa ohjelmoija oppii käyttöympäristön ja ohjelmointikielen syntaksin. Hän oppii vähitellen käyttämään Smalltalkin tarjoamaa laajaa luokkakirjastoa sekä viimeisessä vaiheessa ymmärtää mitä eroa on luokalla ja luokan ilmentymällä, oliolla. Tyypillisesti tämä vaihe kestää 1-9 kuukautta.

Taso 5 käsittää Model-Pane-Dispatcher-yhdistelmän (MPD-yhdistelmän) hallitsemisen. MPD:n avulla hallitaan ikkunoita ja menuja sekä kuinka syöttö- ja tulostetietoja käsitellään. MPD-yhdistelmän hallitseminen tarkoittaa, että ohjelmoija pystyy tekemään Smalltalkilla lähes kaiken, mitä kielellä pystytään toteuttamaan. Tyypillisesti tämä vaihe kestää 3 kuukautta - 3 vuotta.

Vaikka ohjelmoija hallitsisi kielen syntaksin täysin ja osaisi käyttää Smalltalkin luokkia, ei se kuitenkaan tarkoita, että ohjelmoija osaisi suunnitella oliosuuntautuneesti. Tason 6 ja 5 erottaa toisistaan se, ajatteleeko ohjelmoija oliotermein ja olioiden kommunikoinnin kautta. Tason 6 ohjelmoijat tekevät selvän eron suunnittelun ja ohjelmoinnin välillä ja tekevät suunnitelmansa oliotermein, mutta tekevät sen selvästi tietoisina siitä. Tason 6 ohjelmoijat kuvittelevat, että tason 7 ohjelmoijat tekevät oliosuunnittelun alitajuisesti ja että heille tulee luonnostaan mieleen järjestelmän oliot. Yleensä taso 6 kestää kolmesta vuodesta kuuteen vuoteen. Seitsemännen tason ohjelmoijia Cambelin ym. tutkimuksessa ei ollut.

Olioiden kommunikaation ja käyttäytymisen ymmärtäminen vaatii pidemmän opettelujakson, jota on kuvattu mm. Hodgsonin (1994) toimesta. Hodgson esittelee myös C++-kielen oppimiskäyrän (kuvio 13).

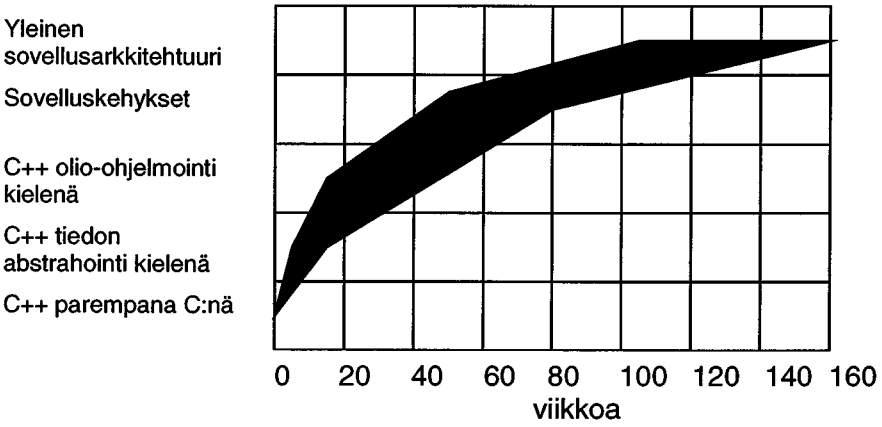
Hodgson ei esitä tämän tarkemmin tasojaan, vaan toteaa, että teollisuudesta tulleiden raporttien perusteella voidaan kuvan osoittama yhteenveto tehdä. Yritettäessä löytää jotakin informaatiota kuviosta 13 voimme tehdä seuraavia johtopäätöksiä:

1. Abstraktiotaso nousee kehityksen myötä: alussa kieltä käytetään parempana ohjelmointityökaluna. Tämän jälkeen aletaan käyttää kielen ominaisuuksia tiedon abstra-

[9] Model-Pane-Dispatcher on sama kuin Model-View-Controller, joka on esitetty mm. Buschmann ym. (1997) kirjassa.

hointiin. Lopuksi kieltä käytetään sovellusten abstrahointiin, eli suunnitellaan sovellus- ja arkkitehtuurikehyksiä.

2. Kuviosta on myös tunnistettavissa olio-oppimisen kehittyminen. Aluksi opitaan kielen syntaksi. Toiseksi opitaan käyttämään oliokäsitteitä tiedon kapselointiin ja tiedon abstrahointiin. Kolmanneksi opitaan käyttämään kieltä oliosuuntautuneena kielenä. Neljänneksi opitaan luomaan laajempia abstraktiotasoja.



Kuvio 13. C++-kielen oppimiskäyrä (Hodgson, 1994)

6.6 Olio-oppimisen tasot

Olio-oppimisessa voidaan intuitiivisesti tunnistaa kaksi eri tasoa: oliokielen oppiminen ja oliokeskeisyyden oppiminen. Tämä jaottelu ei kuitenkaan pakota opettelemaan oliokieltä. Oliokäsitteistä saadaan kuitenkin konkreettista kokemusta tekemällä olio-ohjelmia. Taulukossa 9 esitetään intuitiivinen jako oliokeskeisyyden oppimiselle. Siinä esitetään, millaisia käsitteitä edellä mainittuihin tasoihin voi sisältyä.

Taulukossa 9 esitetty jako on usein kirjallisuudessa esiintyvä. Jollakin mystisellä tavalla oliokeskeisyydessä siirrytään ylemmälle tasolle, jossa oliokeskeisyys ja kaikki siihen liittyvät asiat ovat selkeitä ja helppoja. Tällainen jako ei ilmeisistä syistä ole hyvä.

Taso	Tunnusmerkkejä
Oliokielen oppiminen	Hallitaan oliokeskeisyyden peruskäsitteet: perintä, myöhäinen sidonta ja monimuotoisuus sekä luokat. Hallitaan oliokielen syntaksi.
Oliokeskeisyyden oppiminen	Ymmärretään luokkien ja olioiden ero, olioiden välinen viestintä ja nähdään koko järjestelmä oliotermein. Hallitaan oliokielen semantiikka.

Taulukko 9. Olio-oppimisen intuitiivinen jako

Parempi jako oppimiselle saadaan, kun jaetaan olio-oppiminen useampaan ja selkeästi määriteltyyn tasoon. Tasot saadaan jakamalla oppijan hallitsemat kokonaisuudet eri tasoille. Seuraava lista on tehty Smalltalk-kielen oppimisen ja Hodgsonin C++-kielen oppimistasojen pohjalta, ja siinä esitellään viisitasoinen jako olio-oppimiselle.

1. Aloittelija
2. Oliokäsitteet
3. Oliosuhteet
4. Abstrahointi
5. Järjestelmätaso

Ensimmäinen taso on luonnollisesti aloittelija. Aloittelijaksi käsitetään ihminen, joka tunnistaa, millaisia käsitteitä liittyy oliokeskeisyyteen. Luultavasti aloittelija on saanut muutamasta tunnista muutamaan päivään kestävän koulutuksen. Oliokeskeisyyden käsitteistä tunnetaan luokka, olio, attribuutti ja operaatio, mutta nämä eivät muodosta yhtenäistä kokonaisuutta: käsitteet ymmärretään yksittäisinä saarekkeina. Esimerkiksi attribuutti on tietoelementti, joka liittyy luokkaan; kuitenkaan luokan ja attribuutin välistä yhteyttä ei nähdä. Samalla tavalla ymmärretään, että assosiaatio esittää suhteen kahden luokan välillä; kuitenkaan tämän assosiaation aiheuttamaa yhteyttä luokkien välillä ei nähdä. Aloittelijan kyvyttömyys nähdä kahden eri käsitteen välisiä yhteyksiä aiheuttaa myös sen, ettei esimerkiksi perinnän merkitystä pystytä oivaltamaan.

Tämän luokituksen aloittelija vastaa lähes samaa kuin Smalltalk-kielen oppimisen tasot 1-2 ja Hodgsonin C++ parempana C:nä. Tällä tasolla opitaan oliokeskeisistä käsitteistä luokka, attribuutti, operaatio, assosiaatio ja koostaminen. Olio-ohjelmointikielestä opitaan syntaksi. Aloittelijavaihe kestää noin kolmesta päivästä kahteen viikkoon[10].

Toinen taso on oliokäsitteet. Oppija ymmärtää, että luokkaan liittyy attribuutit ja operaatiot ja nämä kuvaavat luokan piirteet. Tätä kautta oivalletaan abstraktit tietotyypit. Ymmärretään tietoabstrahoinnin tarkoitus ja kuinka se toteutetaan oliokeskeisyydessä. Oppija ymmärtää myös luokan ja olion eron. Myös perinnän ymmärtäminen käy mahdolliseksi. Oppija huomaa, että tietoabstrahointi mahdollistaa olion piirteiden uudelleenkäytön jälkeläisluokissa. Voidaan ymmärtää, mitä perintään tiivisti liittyvät käsitteet, monimuotoisuus ja myöhäinen sidonta, tarkoittavat. Tässä vaiheessa voidaan alkaa

[10] Arvioidut kestoajat on otettu Cambellin ym. (1992) ja Hodgsonin (1994) tutkimuksista.

piirtää luokkamallia ja suunnitella, kuinka oliokeskeisiä järjestelmiä rakennetaan. Yksi merkittävimpiä oivalluksia on olion identiteetti.

Oliokäsitteet-taso vastaa Smalltalk-kielen oppimisessa tasoa 3 ja Hodgsonin C++ kielen luokituksessa C++ tietoabstrahointikielenä -tasoa. Tällä tasolla opitaan oliokeskeisyyden luokka- ja oliokäsitteet sekä näihin liittyvät seikkojen merkitys. Ymmärretään luokka kokonaisuutena ja ymmärretään, että olio on luokan instanssi. Osataan tehdä ja käyttää hyväksi luokkahierarkioita. Oliokäsitteet-taso saavutetaan kahdesta viikosta kuuteen kuukauteen.

Kolmas oppimisen taso on oliosuhteet. Tällä tasolla oppija oivaltaa oliokokonaisuuksia. Ymmärretään yksityiskohtaisemmin luokkien ja olioiden eroja; oivalletaan metaluokkien merkitys ja ymmärretään, milloin esimerkiksi attribuutti tai operaatio on luokkakohtainen. Suurin oivallus on ymmärtää olioiden välinen dynamiikka. Oliokäsitteet-tasolla pystytään ymmärtämään rakennettavaa järjestelmää staattisesti, nyt voidaan nähdä sama rakenne dynaamisena. Oppija kykenee näkemään keskenään vaikutuksissa olevat oliot sekä pystyy tunnistamaan yksinkertaisia taktisen tason suunnittelumalleja. Voidaan jakaa vastuuta eri olioiden kesken ja ymmärretään, mitä tällainen vastuunjako ohjelmassa tekee. Oppija pystyy suojaamaan paremmin ohjelman olioiden käyttäytymisen ja ymmärtää rajapintojen merkityksen.

Oliosuhteet-taso vastaa Smalltalk-kielen oppimisessa tasoa 4 ja Hodgsonin jaottelussa tasoa C++ oliokeskeisenä kielenä. Oliokeskeisyyden käsitteistä ymmärretään sidonta ja kytkentä, olioiden keskeinen vastuunjako, delegointi ja kapselointi. Kaikki em. käsitteet eivät koske ainoastaan oliokeskeisyyttä, mutta ymmärretään ja nähdään, kuinka nämä käsitteet toimivat oliokeskeisissä järjestelmissä. Tämän tason keskeisin oivallus on dynamiikan näkeminen olioiden välillä. Tämä taso voidaan saavuttaa muutamasta kuukaudesta kahdeksaan kuukauteen.

Abstrahointi on neljäs oppimisen taso. Tällä tasolla kyetään näkemään eri abstraktiotasojen oliomalleja. Analyysivaiheessa tehtävä luokkamalli voi erota suunnittelutason oliomallista huomattavasti, ja silti tällä tasolla pystytään näkemään, kuinka analyysimalliin tehtävät muutokset vaikuttavat suunnittelutason malliin ja jopa kooditasolle saakka. Tällä tasolla voidaan jo erottaa suunnittelumallien väliset yhteydet sekä kuinka suunnittelumalli voi koostua useista pienemmistä suunnittelumalleista. Oppija oivaltaa esimerkiksi, että Model-View-Controller suunnittelumalli rakennetaan usean muun suunnittelumallin avulla (Buschmann ym. 1996). Näitä muita suunnittelumalleja ovat

Observer, Composite ja Strategy. Oletuskontrolleri luodaan Factory Methodilla, ja Decoratoria käytetään luomaan mm. vierityspalkit.

Abstrahointikyky antaa suunnittelijalle mahdollisuuden nähdä paremmin yleisen ja yksityisen ero. Näin ollen suunnittelumallien ja sovelluskehysten suunnitteleminen käy mahdolliseksi. Oppija alkaakin nähdä ja tehdä omasta erikoisalastaan suunnittelumalleja, jotka ovat strategisia suunnittelumalleja. Tavoitteena on luoda omalle erikoisalalleen yleiskäyttöisiä sovelluskehyksiä.

Neljäs oppimisen taso vastaa Smalltalk-oppimisen tasoa 5 ja Hodgsonin mallissa C++ sovelluskehyskielenä. Oliokeskeisyyden käsitteistä on sisäistetty abstrahointi, rajapintaluokat (vrt. Jaaksi 1997) ja ns. template- ja hook-luokat. Nähdään myös koko oliokeskeisyyden eri osa-alueiden luoma synergia (vrt. Rumbaugh ym. 1991) ja järjestelmän dynamiikka helpommin: ei enää rajoituta esimerkiksi suunnittelumallin dynamiikkaan. Tähän vaiheeseen pääsy kestää yleensä kolmesta kuukaudesta kahteen ja puoleen vuoteen.

Tämän viitekehyksen viides ja viimeinen taso on järjestelmätaso. Oliot ja olioiden muodostamat järjestelmät ovat tällä tasolla itsestään selviä. Oliomaisuus, olioiden käyttäytyminen ja olioiden dynamiikka nähdään vaistomaisesti sovelluksessa. Olioista on tullut toinen luonto. Cambellin ym. (1992) tutkimuksessa seitsemännen tason ihmiset ajattelevat tiedostamattaan oliomaisesti, ja Dreyfus ja Dreyfus (1986) sanovat, että ekspertille tieto ja taito ovat menneet selkäytimeen. Näin on viidennen tasonkin kanssa, oliot ja oliomaisuus ovat itsestään selviä. Kehittäjä pystyy näkemään lopullisen ratkaisun jo pitkälti arkkitehtuurin perusteella ja pystyy antamaan arvion lopullisen järjestelmän hyvistä ja huonoista puolista luotettavasti.

Viides taso muodostuu Smalltalk-oppimisen tasoista 6 ja 7 sekä Hodgsonin mallin tasosta C++ yleisenä sovelluskielenä. Tälle tasolle voidaan päästä noin 3 – 5 vuodessa, mikäli oppija yrittää aktiivisesti päästä eteenpäin.

Kaikkien edellä mainittujen tasojen yhteinen nimittäjä on, että kokemuksen myötä saavutetaan yhä parempi näkemys omalta erikoisalalta. Mikäli ihminen on intensiivisesti mukana kehitystyössä ja kokee useiden järjestelmien rakentamisen ja valmistumisen, on itsestään selvää, että vuosien kokemuksen jälkeen nähdään jo alkuvaiheen ratkaisuista, minkälainen järjestelmä on tulossa. Oppija saa jatkuvasti kattavampia skeemoja ja voi näin hallita laajempia kokonaisuuksia. Kuten kohdassa 2.2 mainittiin, ihminen ei hallit-

se kerralla enempää kuin 7 ± 2 ajatuskokonaisuutta. Mutta kun ajatuskokonaisuus kasvaa, voidaan hallita suurempia kokonaisuuksia.

Taulukoon 10 on tehty yhteenveto edellä esitetyistä seikoista. Taulukosta näkyvät olio-oppimisen tasot, eri tasoihin kuuluvat keskeisimmät opittavat käsitteet sekä ajatuskokonaisuudet eli skeemakokonaisuudet kullakin tasolla.

Taso	Oliokeskeiset käsitteet	Ajatuskokonaisuus
Aloittelija	Peruskäsitteet: olio, luokka, attribuutit, kooste, operaatiot, perintä.	Luokan attribuutti tai operaatio.
Oliokäsitteet	Olion ja luokan erottaminen. Olio- ja luokkakohtaiset piirteet. Olioiden välinen viestintä. Myöhäinen sidonta ja monimuotoisuus.	luokka, olio, kahden olion välinen suhde
Oliosuhteet	Olioiden väliset suhteet. Dynamiikan näkeminen mallissa.	olioiden muodostamat kokonaisuudet
Abstrahointi	Eri abstraktiotasojen näkeminen. Siirtyminen eri abstraktiotasojen välillä. Erotetaan analyysi- ja suunnitteluvaiheiden luokkamallit. Olioiden koostumissuhteet.	Suunnittelumallien väliset yhteydet. Usean suunnittelumallin muodostama kokonaisuus, esimerkiksi Model-Pane-Dispatcher.
Järjestelmätaso	Oliot ovat toinen luonto. Järjestelmät ajatellaan olioiden muodossa.	Järjestelmä, ohjelmisto, ohjelmistojen ja järjestelmien väliset suhteet.

Taulukko 10. Olio-oppimisen viisi tasoa.

Taulukoon 11 on eritelty oliokäsitteistöä ja termejä jaettuna viidelle olio-oppimisen tasolle. Oliokäsitteistö on kerätty Boochin (1994), Jacobsonin ym. (1992), Rumbaughin ym. (1991), Koskimiehen (1997), Preen (1995) ja Taivalsaaren (1993) teoksista. Näissä teoksissa esiintyneet termit on jaettu ajatuskokonaisuuksien perusteella kullekin oppimistasolle.

Villeneuven ja Fedorowiczin (1997) esittämä jako skeemoihin ja skripteihin selittää, kuinka ihminen mieltää maailman paremmin olioiden avulla. Skeemat ja skriptit auttavat ymmärtämään oliokeskeisyyttä. Skeemateorian mukaan oppiminen on abstrahointia. Kuten ylempänä on mainittu, voidaan koko olio-oppiminen nähdä yhä suurempien kokonaisuuksien hallintana. Tasoihin jako ei vielä kuitenkaan selitä, kuinka eri tasot saavutetaan tai kuinka alemmalta tasolta päästään siirtymään ylemmälle tasolle.

Aloittelija	Oliokäsitteet	Oliosuhteet	Abstrahointikyky	Järjestelmätaso
- luokat - metodit - attribuutit - koostaminen - oliokielen syntaksi - periytyminen	- monimuotoisuus - myöhäinen sidonta - luokittelu - olio - identiteetti - suojaus - luokkahierarkiat	- sidonta ja kytkentä - vastuunjako - delegointi - kapselointi (luokkien välillä; suunnittelumallin sisällä) - metaluokat	- takaisinkutsumekanismit - template- ja hook-luokat - metasuunnittelumallit - synergia - abstrahointi	- sovelluskehykset - ohjelma - järjestelmä

Taulukko 11. Olio-oppimisen eri tasot ja tasoihin liittyvää käsitteistöä

Seuraavaksi esitetään yksityiskohtaisemmin, miten siirtyminen eri tasojen välillä tapahtuu. Siirtyminen selitetään Kolbin kokemuksellisen oppimisteorian avulla. Kokemusoppiminen käsittelee, kuinka oppiminen voi tapahtua kokemuksen kautta. Koska suunnittelumallit dokumentoivat eksplisiittisesti kokemusta, nämä kaksi yhdistämällä voidaan saada toimiva malli oliokeskeisyyden nopeammalle oppimiselle.

6.7 Kokemusoppiminen

Engeströmin (1995) mukaan oppimisteoriat ovat perinteisesti rakentuneet kokemuksen käsitteelle. Oppiminen mielletään kokemuksen karttumiseksi ja työstämiseksi. Karttumista ja työstämistä on kuvattu erilaisilla mekanismeilla. Sen sijaan itse kokemus on yleensä otettu jotakuinkin itsestään selvänä ilmiönä, joka ei vaadi suurempaa teoreettista erittelyä.

Tieto ja käsitteet on perinteisesti ymmärretty ilmiöitä ja olioita kuvaavina määritelminä. Tämän perinteisen ajattelutavan mukaan käsite sisältää kyseisen ilmiön tai olion tuntomerkit, joiden avulla ko. ilmiö tai olio voidaan tunnistaa ja erottaa muista. Tällaiset empiiriset käsitteet irrottavat ilmiöt ja oliot toiminnallisista ja historiallisista yhteyksistään, erillisiksi fragmenteiksi. Tällainen tieto johtaa ilmiöiden staattiseen kuvailuun ja luokitteluun ulkoisten yhtäläisyyksiensä perusteella. Se ei auta ymmärtämään ilmiöiden muutosta ja kehitystä. Muutoksen ja kehityksen ymmärtämiseen tarvitaan toisenlaista, teoreettista tietoa. (Engeström 1995.)

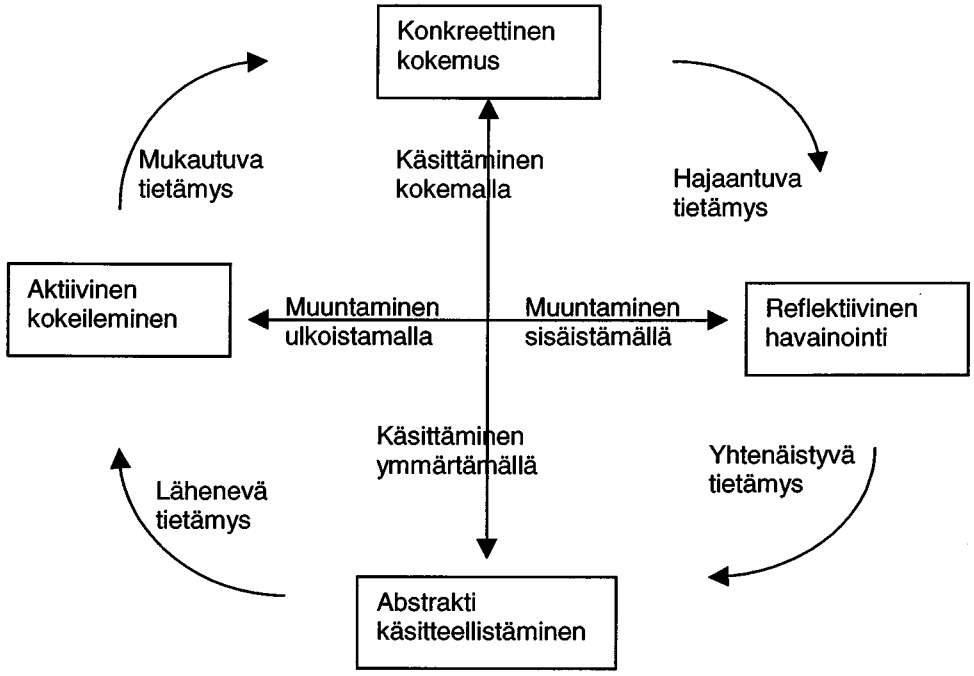
Kolbin (1984) kokemusoppimisen kehämalli muodostuu neljästä askelesta. Ensimmäinen askel on konkreettinen kokemus; toisena seuraa havainnointi ja harkinta; kolmas askel on abstraktien käsitteiden ja yleistysten muodostaminen; neljäs askel on käsitteiden seuraamusten koettelu uusissa tilanteissa, mikä puolestaan johtaa alkupisteeseen eli uusiin konkreetteihin kokemuksiin. Kolb myöntää abstraktien käsitteiden ja yleistysten olemassaolon. Mallin mukaan käsitteet ja yleistykset syntyvät, kun yksilö pohtii eli reflektoi kokemuksiaan. (Engestrom 1995.)

Kolb on kehittänyt teoriansa Piagetin, Lewinin ja Deweyn tutkimusten ja teorioiden pohjalta. Näistä teorioista Kolb (1986) löysi seuraavia keskeisiä yhtäläisyyksiä:

1. Oppimista on parasta tarkastella prosessina, ei saavutettuina tuloksina.
2. Oppiminen on kokemukselle pohjautuva jatkuva prosessi.
3. Oppimisprosessi vaatii erottelua dialektisten vastakkainasettelujen aiheuttamista konflikteista, kun yritetään ymmärtää maailmaa.
4. Oppiminen on holistinen prosessi.
5. Oppimisessa ihmisen ja ympäristön välillä on kanssakäymistä.
6. Oppiminen on tietämyksen luontiprosessi.

Näistä yllämainituista havainnoista Kolb määrittelee *oppimisen* prosessiksi, jossa tietämys luodaan muuntamalla kokemusta. Tässä tutkielmassa oppimisesta käytetään Kolbin määritelmää. Tämä määrittely painottaa useita oppimiseen liittyviä käsitteitä. Ensinnäkin se painottaa oppimisen ja mukautumisen prosessimaista luonnetta, eikä painota opitun sisältöä tai tuloksia. Toiseksi, tietämys on muuntautumisprosessi, jota jatkuvasti uudelleen luodaan. Näin ollen oppiminen ei ole yksilöstä irrotettava kokonaisuus, jota voitaisiin siirtää toiselle henkilölle sellaisenaan. Kolmanneksi, oppiminen muuntaa kokemuksen sekä objektiivisiin että subjektiivisiin muotoihin. Neljänneksi, jotta voitaisiin ymmärtää oppimista, meidän täytyy ymmärtää tietämyksen luonne ja päinvastoin. (Kolb 1986.) Kannattaa huomioida, mitä Rummelhart (1977, 269) on todennut tästä asiasta: "Käsittämisprosessi on oletettava identtiseksi käsitteellisen skeeman valinta- ja varmistusprosessin kanssa tilanteessa (tai tekstissä), joka pitää ymmärtää." (vrt. kohta 6.4)

Edellä mainittujen luokittelujen ja teorioiden pohjalta Kolb esittää oman mallinsa oppimisesta ja tietämyksen hankkimisesta (kuvio 14).



Kuvio 14. Oppimisprosessin rakenteelliset ulottuvuudet, jotka muodostavat tietämyksen perusmuodot (Kolb 1986)

Oppimista täytyy katsoa kahdesta ulottuvuudesta, jotka ovat käsittäminen ja muuntaminen. *Käsittäminen* tarkoittaa kokemuksen ottamista, joko olemalla itse vangitsemassa tapahtumaa taikka muodostamalla ymmärrystä tapahtumasta. Tapahtuman *vangitseminen* on suora ja välitön konkreettinen kokemus. Tapahtuman *ymmärtäminen* on epäsuora kokemus, jossa ymmärretään kokemus symbolisten kuvausten kautta, ymmärtämällä erilaisia symboleja ja näiden yhteyksiä.

Muuntaminen tarkoittaa kokemuksen muuntamista. Tämä tapahtuu sisäistämällä tai ulkoistamalla. Käyttäessään *sisäistystä* ihminen pyrkii havainnoimaan tilannetta ja yrittää luoda siitä ymmärrettävän kokonaisuuden. *Ulkoistamisen* avulla ihminen yrittää käyttää oppimaansa aktiivisesti käytäntöön ja hän pyrkii selvittämään, kuinka opittu toimii käytännössä.

Edellä mainitut kaksi ulottuvuutta, käsittäminen ja muuntaminen, eivät ole yhtenäinen jatkumo. Pikemminkin ne esittävät vastakkain asettelua kahden itsenäisen, mutta toisiaan täydentävien käsitteiden kesken. Vangitseminen ja ymmärtäminen ovat kumpikin itsenäisiä muotoja kokemuksen saamiseksi. Sisäistäminen ja ulkoistaminen ovat itsenäisiä muotoja kokemuksen muuntamiseksi. Vangitseminen ja ymmärtäminen käsittämis-

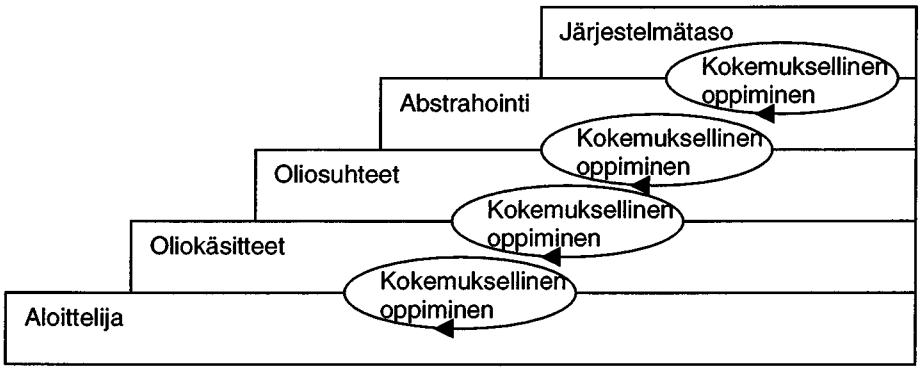
prosessina sekä sisäistäminen ja ulkoistaminen muuntamisprosessina ovat yhtä tärkeitä koko oppimisprosessille. (Kolb 1986.)

Oppiminen on prosessi, jossa tietämystä luodaan muuntamalla kokemusta. Tietämys muodostuu yhdistelmästä, jossa kokemusta hankitaan ja muunnetaan. Koska on kaksi vastakkaista muotoa käsittämisestä sekä vastaavasti on kaksi vastakkaista muotoa muuntaa tätä käsitystä, niin saadaan neljä erilaista tietämysmuotoa. Kokemus, joka hankitaan vangitsemalla ja muunnetaan harkitsemalla, on hajaantuvaa (divergent) tietämystä. Kokemus, joka hankitaan ymmärtämällä ja muunnetaan harkitsemalla, on yhtenäistävää (assimilative) tietämystä. Kun kokemus hankitaan ymmärtämällä ja muunnetaan ulkoistamalla, tuloksena on lähenevää (convergent) tietämystä. Lopuksi, kun kokemus saadaan vangitsemalla ja se muunnetaan ulkoistamalla, tuloksena on mukautuvaa (accomodative) tietämystä. (Kolb 1986.)

6.8 Oliokeskeisyyden kokemusoppiminen

Kuviossa 15 on yhdistetty olio-oppimisen tasot ja Kolbin kokemuksellisen oppimisen prosessikuvaus. Oppiminen sinänsä on kokonaisvaltainen prosessi ja se ei koskaan pysähdy. Näin ollen ei koskaan voida sanoa, että oltaisiin jollakin tietyllä tasolla, vaan aina ollaan siirtymässä seuraavaan tasoon. Oppija on oppiessaan aina tasojen välillä. Varmasti voidaan vain sanoa, koska tietty taso on ohitettu.

Kuviossa 15 esitetyn mallin avulla voidaan testata, millä tasolla oppija on. Kysymällä eri tasoille asetetut käsitteet oppijalta voi ulkopuolinen selvittää, ymmärtääkö testattava käsitteet. Selvittämällä minkä tason käsitteet testattava hallitsee, voidaan päätellä, mille tasolle hän kuuluu.



Kuvio 15. Olio-oppimisen eri tasot ja siirtyminen niiden välillä

Tasot eivät ole mitenkään rajoittavia. Tiettyjä asioita voidaan hallita eri tasoilta, mutta merkitykselliseksi kokonaisuuden kannalta ylemmän tason tieto tulee vasta, kun kyseistä tasoa varten aletaan käydä läpi kokemusoppimisen kehää.

Seuraavaksi esitetään, miten siirrytään eri tasojen välillä. Siirtyminen seuraavalle tasolle tapahtuu saamalla opittavasta asiasta konkreettinen kokemus, josta oppija saa sykäyksen tutkia kokemustaan eli siirtyy reflektiiviseen havainnointiin. Tämän jälkeen oppija abstrahoi havaintojaan ja muodostaa niistä abstraktin kokemuksen. Tätä tietoa kokeillaan aktiivisesti käytäntöön ja sitä kautta saadaan uuttaa konkreettista tietoa ja voidaan siirtyä seuraavan tason oppimiskehän alkuun.

Oppiminen on iteratiivinen prosessi, ja tässä pyritään tunnistamaan vain keskeisimmät käsitteet ja abstrahoinnin kohteet, jotta voidaan luoda perusrunko oliokeskeisyyden kokemusoppimiselle suunnittelumallien avulla.

Aloittelija on ensimmäinen oppimistaso. Ihminen tulee oliokeskeisyydessä aloittelijaksi, kun hän saa ensimmäisen kokemuksen olioteknologiasta. Oppija saa tietoonsa, että on olemassa luokkia, olioita ja näiden välisiä suhteita. Tiedetään peruskäsitteet sekä niiden selitykset. Kuitenkaan ei voida olettaa, että oppijalla olisi muuta kuin ulkoa opittua tietoa. Ulkoa opittu tieto on muotoa: "olio on luokan instanssi" (eli "hauki on kala"). Tästä tilanteesta voidaan lähteä tekemään kokemuksellista oppimista, jotta voitaisiin siirtyä seuraavalle oppimistasolle.

Aloittelijasta oliokäsitteiden hallintaan

Aloittelija osaa peruskäsitteet lähinnä ulkoa. Konkreettinen kokemus onkin, ettei aloittelijan tietomäärällä voi tehdä mitään. Konkreettinen kokemus on, etteivät yksittäiset käsitteet anna sitä, mitä oliokeskeisyyden oletettiin antavan. Tilannetta ja käsitteitä aletaan havainnoida ja kerätään käsitteistä lisää tietoa. Tietoa aletaan abstrahoida ja viimein kyetään muodostamaan yhteinen viitekehys käsitteille ja tätä viitekehystä aletaan kokeilemaan käytännössä. Tässä on ensimmäinen kokemusoppimisen sykli.

Ensimmäisen syklin keskeisin opittu asia on ero luokan ja olion välille. Tätä kautta kyetään ymmärtämään oliokeskeisyyden keskeisten periaatteiden, identiteetin, perinnän, monimuotoisuuden ja myöhäisen sidonnan merkitys ja keskinäinen yhteys. Hallitessaan oliokeskeisyyden peruskäsitteet oppija havaitsee myös abstraktien tietotyyppien merkityksen ja kuinka niitä voidaan toteuttaa ja käyttää oliokeskeisyydessä.

Oliokäsitteiden avulla oppija kykenee nyt myös mallintamaan staattisesti, eli luokkamallin avulla. Keskeistä tällä tasolla on nähdä, kuinka luokkamallin avulla voidaan kuvata ongelma-aluetta.

Mikäli tässä vaiheessa yritetään oppia myös olio-ohjelmointia, on idiomeista hyötyä. Aloittelija näkee alusta pitäen kuinka käyttää kieltä oikein, eikä hänen tarvitse myöhemmin opetella pois huonoista tavoista. Olio-ohjelmoinnin avulla oppijan on helpompi mieltää myös edellä mainittuja keskeisiä käsitteitä. Olion identiteetin, perinnän, myöhäisen sidonnan ja monimuotoisuuden oppiminen on huomattavasti helpompaa konkreettisten olio-ohjelmien kautta kuin kirjoista lukemalla.

Oliokäsitteistä olioiden välisten suhteiden hallintaan

Peruskäsitteiden hallinnalla on oppija oivaltanut suhteiden jakaantuvan kolmeen eri luokkaan: perintä-, koostumis- ja yhteistyösuhde. Nyt havaitaan, että olioiden käyttäytyminen vaikuttaa laajemmallekin alueelle kuin pelkästään kahden olion välillä. Konkreettinen kokemus on, että olio-ohjelma syntyy olioiden välisestä kommunikaatiosta. Olion välisiin suhteisiin vaikuttaa kaikkien olioiden suhteet yhdessä. Havaitaan tarve suhteiden muodostamien kokonaisuuksien hallintaan. Luodaan käsitys sidonnan ja kytkennän merkityksestä sekä siitä, kuinka vastuuta voidaan jakaa olioiden kesken. Olioiden välinen dynamiikka käy tärkeäksi ja oppija kokeilee näitä malleja käytännössä.

Oliosuhteet-tason tärkein oivallus on nähdä olioiden välinen dynamiikka. Samalla oppija kykenee ymmärtämään tärkeitä periaatteita järjestelmän rakentamisesta, kuten esimerkiksi sidonnan ja kytkennän, kapseloinnin ja näihin liittyvät oliokeskeiset käsitteet, kuten metaluokat ja virtuaalioperaatiot.

Tässä vaiheessa suureksi avuksi tulevat taktiset suunnittelumallit, joissa on valmiiksi dokumentoitu olioiden välistä vastuunjakoa. Tutkimalla taktisia suunnittelumalleja oppija kykenee näkemään, kuinka todellisuudessa on ratkaistu niitä ongelmia, joita hän pohtii.

Olioiden välisistä suhteista abstrahointiin

Oliodynamiikan havaitseminen oliojärjestelmissä auttaa oppijaa näkemään, kuinka monimutkainen lopullinen järjestelmä on. Konkreettinen kokemus on, että havaitaan tiettyjen rakenteiden ja suhteiden toistuvan suunnitelmissa. Havainnoimalla tehtyjä dynaa-

misia ja staattisia malleja sekä katsomalla taktisia suunnittelumalleja voidaan järjestelmistä luoda korkeampia abstraktiotasoja. Esimerkiksi Model-View-Controller mallin näkeminen ja huomaaminen, että se muodostuu Observer-, Composite-, Strategy-, Factory Method - ja Decorator-suunnittelumalleista, auttaa oppijaa muodostamaan konkreettisista luokista korkeamman tason malleja. Käytännössä huomataankin, kuinka suunnittelumallit itsessään voivat sisältää toisia suunnittelumalleja ja esimerkiksi arkkitehtoniset suunnittelumallit määräävät koko sovelluksen perusrakenteen.

Merkittävin muutos tämän tason ja edellisen tason välillä on oppijan kyky siirtyä yleisestä yksityiseen ja päinvastoin. Tällä tasolla oppija kykenee näkemään, kuinka abstraktin tason muutokset vaikuttavat konkreetille tasolle.

Tällä tasolla voidaan käyttää kaikkia suunnittelumalleja hyväksi ja varsinkin monimutkaisemmat suunnittelumallit auttavat mieltämään, kuinka voidaan abstrahoida järjestelmiä. Nyt oppija omaa kyvyn tehdä omiakin suunnittelumalleja ja näitä kannattaakin luoda omalle erikoisalalle. Lisää oppia ja näkemystä saadaan tutkimalla valmiita sovelluskehyksiä.

Abstrahoinnista järjestelmätasolle

Korkeamman abstraktiotason avulla ja tehtyjen strategisten suunnittelumallien avulla oppija havaitsee kykenevänsä kuvamaan erittäin tarkasti omaa erikoisalaansa. Konkreettinen kokemus on, että järjestelmät rakennetaan muutamia yleisiä periaatteita varioimalla. Abstrahoimalla lisää sovelluksia ja käyttämällä oppimaansa oppija kykenee luomaan sovelluskehyksiä omalle erikoisalalleen. Kokeilemalla sovelluskehyksiä käytännössä voidaan niitä hiljalleen parantaa ja saada niistä yhä edelleen parempia ja yleiskäyttöisempiä. Oppija on vihdoinkin saavuttanut tason, jolla hän kykenee vaivattomasti näkemään eri järjestelmien väliset suhteet. Hän on viimein saavuttanut tason, jossa häntä voi nimittää erikoisalansa ekspertiksi.

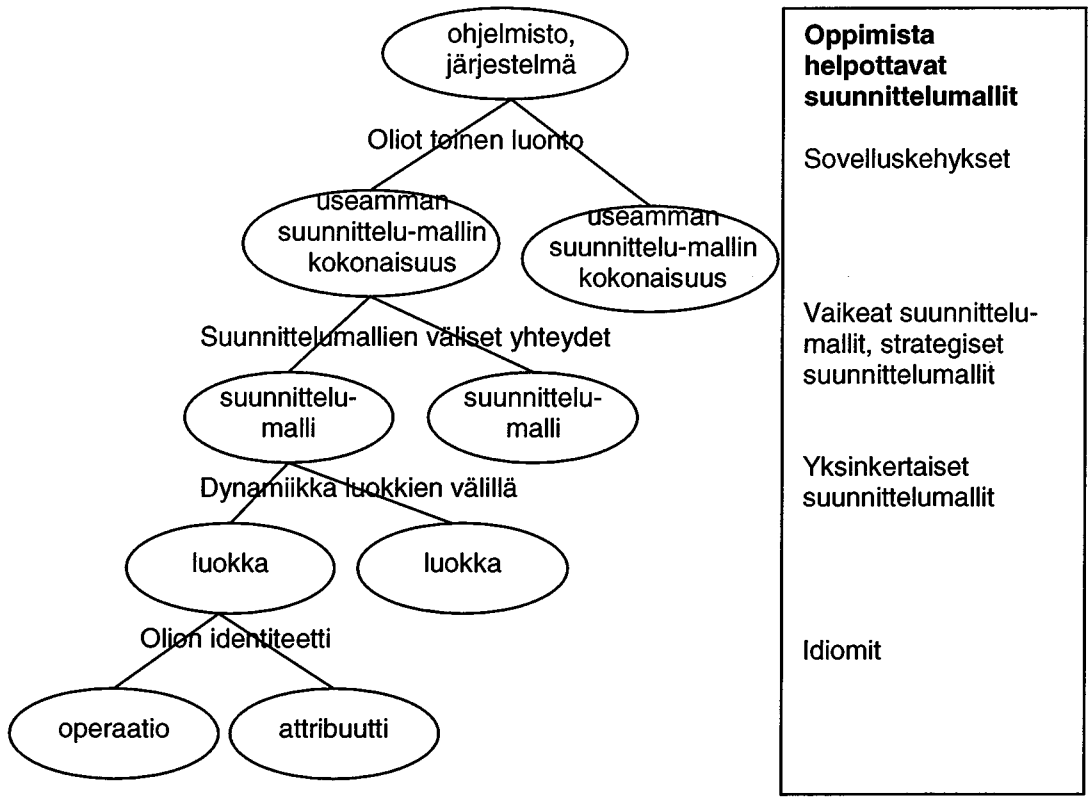
Yhteenveto olio-oppimisen tasoista

Taulukossa 12 esitetään, kuinka eri tasoilta voidaan siirtyä seuraavalle tasolle. Jokaisella tasolla on hyödyllistä käyttää erilaisia suunnittelumalleja, koska niiden avulla voidaan nopeuttaa seuraavan tason saavuttamista. Aluksi käytetään suunnittelumalleja, jotka ovat yksityiskohtaisempia ja suppeampia ja joiden omaksuminen on helpompaa. Kun perusasiat on hallussa, voidaan pyrkiä ymmärtämään syvemmälle meneviä asioita, jotka perustuvat näille peruskäsitteille.

Kuinka voidaan nopeuttaa havainnointia, auttaa käsitteellistämistä ja nopeuttaa aktiivista kokeilua			
Aloittelijasta oliokäsitteiden haltijaksi	Oliokäsitteistä olioiden välisten suhteiden ymmärtämiseen	Olioiden välisistä suhteista eri abstraktiotason olioiden ymmärtämiseen	Olioabstraktioista järjestelmätason taitajaksi
- Idiomit	- Yksinkertaisimmat ja yleisimmät taktiset suunnittelumallit: Iterator, Composite, Observer, Stategy, Decorator, Abstract Factory, Factory Method, Template Method, Adapter	- Vaikeammat ja harvemmin käytettävät taktiset suunnittelumallit mm. Chain of Responsibility, Command, Builder, Flyweight, Visitor ja Memento. - Strategiset suunnittelumallit - Suunnittelumallien väliset yhteydet, esimerkiksi Model-View-Controller muodostetaan Observeristä, Compositestä ja Strategysta, sekä oletus kontrolleri luodaan Factory Methodilla ja Decoratoria käytetään luomaan mm. vierityspalkki.	- Suunnittelumallit + komponentit = sovelluskehykset - ET++, CORBA, Java AWT, OLE, OpenDoc - Mallijärjestelmät

Taulukko 12. Olio-oppimista nopeuttavat suunnittelumallit

Kuviossa 16 on tehty yhteenveto olio-oppimisen tasoista ja oppimista nopeuttavista suunnittelumalleista. Kuviossa ovaalit kuvaavat kunkin tason käsitekokonaisuuksia. Käsitteet on yhdistetty viivoilla, ja viivojen teksti kuvaa keskeisintä oppimista, joka tapahtuu tasojen välillä. Oppimista nopeuttavat suunnittelumallit on esitetty kuvioin oikean reunan suorakaiteessa.



Kuvio 16. Oliokeskeisyyden oppiminen ja oppimista helpottavat suunnittelumallit

6.9 Yhteenveto

Tässä luvussa esiteltiin oliolähestymistavan oppiminen suunnittelumallien avulla. Aluksi esiteltiin kognitiivisen tieteen käsitystapa ihmisen mielen rakenteesta. Tämän teorian mukaan ihmisen ajattelua ja muistamista voidaan selittää skeema- ja skriptiteorioiden kautta. Skeemat ja skriptit kuvaavat ihmisen tapaa hahmottaa maailmaa. Skeemat ja skriptit muistuttavat myös läheisesti oliokeskeisessä lähestymistavassa käytettyjä käsitteitä ja tapaa mallintaa maailmaa. Näiden teorioiden mukaan ihmisen tieto kasvaa, kun hän pystyy luomaan korkeamman abstraktiotason skeemoja ja skriptejä, jotka sisältävät erikoistapauksia.

Seuraavaksi käsiteltiin eksperttien ja noviisien välisiä eroja ja todettiin, että ekspertti on henkilö, joka hallitsee erikoisalansa paremmin kuin noviisi. Paremmuutta voidaan mitata nopeuden ja työn laadun perusteella. Seuraavassa kohdassa käsiteltiin, kuinka eksperttiys näkyy oliokeskeisyydessä. Ekspertit lähestyvät ongelmia eri tavoin kuin noviisit sekä käyttävät enemmän aikaa ongelman määrittelyyn. Oliokeskeisyys tarjoaa eksperteille myös työkalut mallintaa päättelyprosessiaan.

Tämän jälkeen luvussa jaettiin olio-oppiminen viiteen tasoon sen mukaan, kuinka oliokeskeisyydessä aloitteleva ihminen saavuttaa eksperttitason. Ensiksi esiteltiin, kuinka oliokielissä on tunnistettu eri tasoja ja näiden mukaan jaettiin olio-oppiminen viiteen eri tasoon. Vaikka tasojako on spekulatiivinen, niin sitä tukee kuitenkin kolme seikkaa. Ensinnäkin se pohjautuu Smalltalk-kielen oppimisen tasoihin sekä Hodgsonin esittelemään C++-kielen oppimiseen. Toiseksi se pohjautuu perusjakoon, joka lähtee konkreettisista ja yksinkertaisista asioista ja päättyy abstrakteihin ja vaikeisiin kokonaisuuksiin, kuten skeema- ja skriptiteoriat osoittavat. Kolmanneksi siinä näkyy, kuinka oppiminen lähtee rajatuista käsitteistä ja kokonaisuuksista ja vasta tämän jälkeen voidaan hallita laajempia kokonaisuuksia.

Luvun lopuksi esitettiin Kolbin kokemusoppiminen. Kokemusoppimisen avulla esitettiin, kuinka suunnittelumalleja voidaan käyttää siirryttäessä olio-oppimisen tasolta toiselle. Suunnittelumallit on jaettavissa yksinkertaisiin ja monimutkaisiin ja näitä malleja kannattaa opetella olio-oppimistason mukaisessa järjestyksessä.

7 YHTEENVETO

Suunnittelumalli on nimetty ratkaisu toistuvaan ongelmaan tietyssä kontekstissa. Ongelmana suunnittelumallien käytölle on ollut ensinnäkin, ettei olemassa olevia luokitteluja ole arvioitu, eikä ole luotu yhtenäistä viitekehystä niiden arvioimiseksi. Toiseksi suunnittelumallien käyttöä ohjelmistotuotannon eri vaiheissa ei ole selvitetty. Kolmanneksi ei ole esitetty, kuinka suunnittelumalleja kannattaa opetella.

Tämän tutkielman tarkoituksena on ollut tutkia suunnittelumallien käyttöä ohjelmistotuotannossa. Tutkielman tavoitteena on ollut ensinnäkin selventää ja helpottaa suunnittelumallien käyttöä. Tätä varten on luotu yhtenäinen viitekehys suunnittelumallien arviointiin ja valitsemalla suunnittelumallien käyttöä helpottava luokittelu. Toiseksi tutkielmassa on esitetty, kuinka suunnittelumalleja voidaan käyttää ohjelmistotuotannossa. Tätä varten suunnittelumallien käyttötapa on jaettu kahteen eri luokkaan: suunnittelumallien hyväksikäyttö ohjelmistotuotannossa ja ohjelmistotuotanto suunnittelumalleja varten. Kolmanneksi on esitetty olio-oppimisen eri tasot ja kuinka suunnittelumallien avulla voidaan siirtyä tasolta seuraavalle. Tätä varten on olio-oppiminen jaettu viiteen tasoon ja selvitetty, kuinka suunnittelumalleja voidaan hyödyntää siirryttäessä tasolta toiselle.

Tutkielman keskeiset kontribuutiot ovat: 1) suunnittelumalliluokitusten arviointi ja vertailu, 2) tiivis esitys siitä, millä tavalla suunnittelumalleja voidaan käyttää ja kehittää ohjelmistotuotannossa, sekä niiden käyttökokemuksista, 3) olio-oppimisen jakaminen tasoihin ja näiden tasojen esittely, sekä 4) kuinka suunnittelumallien avulla voidaan edistää olio-oppimista.

Käytössä olevat suunnittelumalliluokitukset perustuvat kolmeen ulottuvuuteen. Nämä ovat luokittelu ohjelmistotuotannon vaiheen mukaan, suunnittelumallin toiminta-alan mukaan sekä oliokeskeisyyden käsitteiden mukaan. Ohjelmistotuotannon näkökulmasta toimiva luokittelu on sellainen, joka tukee ohjelmistotuotannon eri vaiheita sekä auttaa käyttämään luokittelua ohjelmistotuotannon ongelmien mukaan. Tällaiseksi luokitteluksi osoittautui Buschmannin ym. (1996) esittämä luokittelu.

Suunnittelumallien käyttöä ohjelmistotuotannossa on työssä tarkasteltu komponenttilähestymistapaan perustuen. Ohjelmistotuotannon eri vaiheisiin tarkoitettuja suunnittelumalleja kannattaa käyttää kunkin vaiheen yhteydessä. Tätä varten esiteltiin, kuinka kun-

kin eri ohjelmistotuotannon vaiheen sisällä otetaan huomioon mahdollisten suunnittelumallien käyttö.

Olio-oppimisen on esitetty etenevän viiden tason mukaan. Kullekin tasolle on ominaista oppijan hallitsema ajatuskokonaisuus. Työssä on näytetty perustuen kognitiivisten tieteen teorioihin, kuinka oliokeskeisyys ja suunnittelumallit tukevat ajatuskokonaisuuksien hahmottamista ja hallintaa. Lopuksi on esitetty, kuinka suunnittelumalleja voidaan käyttää hyödyksi siirryttäessä olio-oppimisen tasolta toiselle.

Suunnittelumallien suurimmaksi hyödyksi on kirjallisuudessa todettu yhteinen sanasto, joka antaa korkeamman abstraktiotason keskusteluihin ja suunnitteluun. Tätä on toistettu ja siteerattu niin usein, että se on alettu ottamaan totuutena. Kuitenkaan yhteinen sanasto ei ole merkittävin suunnittelumallien antama hyöty. Suurimmat suunnittelumallien antamat hyödyt ovat: 1) ekspertit voivat tallentaa tietämystään myöhempiä tilanteita varten suunnittelumalleilla; 2) suunnittelumallit ovat valmiita ja dokumentoituja ratkaisuja ongelmiin; 3) suunnittelija voi kokeilla kilpailevia suunnittelumalleja tiettyyn ongelmaan; ja 4) suunnittelumallien avulla voidaan dokumentoida eksperttien tietämystä ja siirtää sitä noviisien käyttöön.

Tämän tutkimuksen tuloksia voidaan käyttää hyödyksi aloitettaessa suunnittelumalleihin perehtyminen. Tutkielmassa esitetään kaikki keskeiset käsitteet, joita tarvitaan suunnittelumallien yhteydessä. Tutkimuksen tuloksia voidaan käyttää myös, kun suunnittelumalleja halutaan käyttää ohjelmistotuotannossa. Tutkielmasta löytyy malli, kuinka suunnittelumalleja voidaan käyttää ohjelmistotuotannossa hyväksi. Mallin yhteydessä esitetään myös ne eri vaiheet, jotka täytyy käydä läpi, kun suunnittelumalleja halutaan löytää.

Olio-oppimisen yhteydessä löydettyjä tasoja voidaan käyttää hyväksi suunniteltaessa oliokeskeisyyden käyttöönottoa yrityksessä. Oliokeskeisyyden opettaminen voidaan tehdä tutkimuksessa esitetyn tasojaon mukaan. Tutkielmassa on myös esitetty, kuinka suunnittelumalleja voidaan käyttää hyväksi siirryttäessä olio-oppimistasolta seuraavalle.

Suunnittelumallien käyttöönoton ohjelmistotuotannossa voi tehdä luvussa viisi esitettyjen periaatteiden mukaan. Aluksi ohjelmistotuotannossa otetaan käyttöön taktiset suunnittelumallit. Kun taktisten suunnittelumallien käyttö hallitaan, voidaan suunnittelumalleja etsiä omista ohjelmistoista. Näin saadaan vähitellen omia strategisia suunnittelumalleja ja voidaan kehittää omalle toimialalle mallijärjestelmiä.

Mikäli yrityksessä päätetään aloittaa suunnittelumallien hyväksikäyttö, niin tätä varten tarvitaan jokin tapa luokitella malleja. Ohjelmistotuotantoon sopiva luokittelu valittiin neljännessä luvussa. Käyttämällä hyväksi tätä luokittelua, voivat yritykset helpommin löytää tarvitsemansa luokittelumallit. Valittu luokittelu auttaa yrityksiä myös lisäämään omat suunnittelumallit luokittelun piiriin, koska se on helposti laajennettavissa sisältämään uusia ongelmaluokkia.

Tutkielmassa esitetyt tulokset ovat syntyneet oliokeskeisestä näkökulmasta. Tulokset eivät tämän vuoksi ole käyttökelpoisia muiden lähestymistapojen yhteydessä. Suunnittelumallien tutkimus on vielä nuorta, eikä kriittistä suhtautumista niihin vielä ole esiintynyt laajemmassa mittakaavassa. Koska tutkielma on perustunut tieteellisiin artikkeleihin ja koska näissä ei ole esiintynyt laajaa kritiikkiä, eivät tutkielman tulokset välttämättä ole niin koeteltuja, kun toivoisi.

Hedelmällistä aineistoa antaisi jatkotutkimus, jossa testattaisiin tässä tutkielmassa esiteltyjä oppimistasoja. Empiiristen tulosten pohjalta tämän tutkielman tietoja voisi tarkentaa edelleen ja luoda jopa hienojakoisemman mallin. Toinen jatkotutkimuksen aihe voisi sisältää kunkin vaiheen sisällä tapahtuvan oppimisen tunnistamisen. Jokaisen vaiheen sisällä voidaan tunnistaa varmasti pienempiä oppimiskehiä. Kolmas jatkotutkimuksen aihe olisi tutkia empiirisesti suunnittelumallien hyväksikäyttöä siirryttäessä oppimistasolta seuraavalle. Neljäs mielenkiintoinen tutkimuksen kohde olisi tarkentaa suunnittelumallien käyttöä eri ohjelmistotuotannon vaiheissa. Tutkimuksen tavoitteena olisi tuottaa yhtenäinen mallikieli samaan tapaan kuin Zimmer (1995a) on esittänyt luokittelunsa. Tutkimuksessa pitäisi tunnistaa, mistä erilaisista malleista voidaan rakentaa tietyn tyyppisiä sovelluksia.

LÄHTEET

Abelson R., Script processing in attitude formation and decision making. Teoksessa Carrol J., Payne J. (toim.), Cognition and Social Behavior, Lawrence Erlbaum Associates, 1976, 33 – 45.

Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., A Pattern Language. Oxford University Press, New York, 1977.

Alexander C., The Timeless Way of Building. Oxford University Press, 1979.

Anthony D., Patterns for classroom education. Teoksessa Vlissides J., Coplien J., Kerth N. (toim.), Pattern Languages of Program Design 2. Addison-Wesley, 1995, 391 – 406.

Beck K., Crocker R., Coplien J., Dominick L., Meszaros G., Paulisch F., Vlissides J., Industrial experience with design patterns. 18th International Conference on Software Engineering, Berlin, Berlin - Heidelberg - New York, Springer, March 1996, 103-114.

Boehm B., A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, August 1986, 14 – 24.

Booch G., Object-Oriented Analysis and Design with Applications 2nd ed. The Benjamin/Cummings Publishing Company, Inc., 1994.

Bracha G., Cook W., Mixin-based inheritance. ECOOP/OOPSLA '90 Proceedings, October 1990, 303 – 311.

Brooks F., The Mythical Man Month. Addison-Wesley, 1975.

Brooks F., No silver bullet: Essence and accidents of software engineering. IEEE Computer, Vol. 20, No. 4, 1987, 12-22.

Brown K., Using patterns in order management systems: A design patterns experience report. http://www.ksccary.com/PtrnJrnl.htm, 10.4.1998.

Budd T., An Introduction to Object-Oriented Programming. Addison-Wesley, 1991.

Buschmann F., Meunier R., A system of patterns. Teoksessa Coplien J., Schmidt D., Pattern Language of Program Design. Addison-Wesley, 1995, 325-344.

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons, New York, 1996.

Cambell R., Brown N., DiBello L., The programmer's burden: Developing expertise in programmin. Teoksessa Hoffman R. (toim.), The Psychology of Expertice: Congnitive Research and Empirical AI. Springer-Verlag, 1992, 269-294.

Cockburn A., Pioritizing forces in software design. Teoksessa Vlissides J., Coplien J., Kerth N. (toim.), Pattern Languages of Program Design 2. Addison-Wesley, 1995, 319 – 334.

Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F., Jeremaes P., Object-Oriented Development: The Fusion Method. Prentice Hall, 1994.

Cooke N., Modelling human expertice in expert systems. Teoksessa Hoffman R. (toim.), The Psychology of Expertise. Springer-Verlag, 1992, 29 – 60.

Coplien J., Schmidt D. (toim.), Pattern Languages of Program Design. Addison-Wesley, 1995.

Coplien J., A generative development-process pattern language. Teoksessa Coplien J., Schmidt D. (toim.), Pattern Languages of Program Design. Addison-Wesley, 1995, 183 – 238.

Coplien J., Organizational patterns. http://c2.com/cgi/wiki?OrganizationalPatterns, 21.9.1998.

Coram T., Demo prep: A pattern language for the preparation of software demonstrations. Teoksessa Vlissides J., Coplien J., Kerth N. (toim.), Pattern Languages of Program Design 2. Addison-Wesley, 1995, 407 – 416.

Davis A., Bersoff E., Comer E., A Strategy for comparing alternative software development life cycle models. IEEE Transactions on Software Engineering, Vol. 14, No. 10, October 1988, 1452 – 1461.

Dijkstra E., The humble programmer. Communications of theACM, Vol. 15, No. 10, October 1972, 859 – 866.

Dijkstra E., A debate on teaching computing science. Communications of theACM, Vol. 32, No. 12, December 1989, 1397 – 1404.

Dreyfus H., Dreyfus S., Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. Basil Blackwell Ltd., 1986.

Engeström Y., Kehittävä työntutkimus. Painatuskeskus Helsinki, 1995.

Eysenck M., Keane M., Cognitive Psychology: A Students Handbook. Lawrence Erlbaum Associates Ltd., 1990.

Foley M., Hart A., Expert-novice diffrences and knowledge elicitation. Teoksessa Hoffman R. The Psychology of Expertice: Congnitive Research and Empirical AI. Springer-Verlag, 1992, 233-244.

Foote B., Opdyke W., Lifecycle and refactoring patterns that support evolution and reuse. Teoksessa Coplien J., Schmidt D. (toim.), Pattern Languages of Program Design. Addison-Wesley, 1995, 259 – 292.

Fowler M., Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.

Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.

Gilb T., Evolutionary delivery versus the 'Waterfall Model'. Software Engienering Notes, Vol. 10, No. 3, 1985, 49 – 62.

Gomaa H., Scott D., Prototyping as a tool in the specification of user requirements. Proceedings of 5th IEEE International Conference on Software Engineering, March 1981, 333 – 342.

Haikala I., Märijärvi J., Ohjelmistotuotanto. Suomen ATK-kustannus 1995.

Harrison N., Organizational patterns for teams. Teoksessa Vlissides J., Coplien J., Kerth N. (toim.), Pattern Languages of Program Design 2. Addison-Wesley, 1995, 345 – 352.

Helm R., Patterns in practice. Proceedings of OOPSLA'95, SIGPLAN Notices, Vol. 30, No. 10, October 1995, 337 – 341.

Hodgson R., Adopting Object-oriented software engineering. Teoksessa Carmichael A. (toim.), Object Development Methods. SIGS Books Inc., 1994, 13-29.

Holland I., Representation of Object-Oriented Components. Ph.D. Thesis, Northeastern University, 1992. http://www.ccs.neu.edu/home/lieber/thesis-index.html, 21.9.1998.

Irving C., Eichmann D., Patterns and design adaptability. Esitetty 3rd Pattern Language of Program Design -konferensissa, Allerton Park, Illinois, September 4-6, 1996. http://www.cs.wustl.edu/~schmidt/PLoP-96/irving2.ps.gz 16.7.1998.

Jaaksi A., Object-Oriented Development of Interactive Systems. Ph.D. Thesis, University of Tampere, 1997.

Jackson M., Problems, methods and specialisation. Special Issue of SE Journal on Software Engineering in the Year 2001, 1994.

Jacobson I., Christerson M., Jonsson P., Övergaard G., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

Jacobson I., Ericson M., Jacobson A., The Object Advantage: Business Process Reengineering with Object Technology. Addison-Wesley, 1995.

Johnson R., Documenting frameworks using patterns. Proceedings of OOPSLA'92, SIGPLAN Notices, Vol. 27, No. 10, October 1992, 63 – 76.

Kant I., Kritik der reinen Vernunft. 1781. Englannin kielinen käännös Pluhar W., Critique of Pure Reason, Hacket Publishing Company, Inc., 1996.

Kerth N., Caterpillar's fate: A pattern language for the transformation from analysis to design. Teoksessa Coplien J., Schmidt D. (toim.), Pattern Languages of Program Design. Addison-Wesley, 1995, 293 – 320.

Kolb D., Experiental Learning. Prentice-Hall, New Jersey, 1984.

Koskimies K., Pieni oliokirja. Gummerus Kirjapaino Oy, Jyväskylä, 1997.

Lewis T. (toim.), Object-Oriented Application Frameworks. Manning Publications Co., 1995.

Lieberherr K., Holland I., Gar-lin L., Riel A., An objective sense of style. IEEE Computer, Vol. 21, No. 6, June 1988, 79 – 81.

Martin R., Discovering patterns in existing applications. Teoksessa Coplien J., Schmidt D., Pattern Language of Program Design. Addison-Wesley, 1995, 365-394.

Martin C., Riehle D., Buschmann F., Pattern Languages of Program Design 3. Addison-Wesley, 1997.

Menzies T., Object-oriented pattenrs: Lessons from expert systems. Software-Practice and Experience, Vol. 1, No. 1, December 1997.

Meszaros G., Doble J., A pattern language for pattern writing. http://hillside.net/patterns/Writing/patterns.html, 22.9.1998.

Miller G., The magical number seven, plus or minus two: Some limits on our capacity for processing information. The Psychological Review, Vol. 63, No. 2, 81-97.
Löytyy myös http://www.well.com/user/smalin/miller.html, 18.8.1998.

Meyer B., Object-Oriented Software Construction. Prentice Hall, 1988.

Meyer B., The many faces of inheritance: A taxonomy of taxonomy. IEEE Computer, May 1996.

Mills H., Dyer M., Linger R., Cleanroom software engineering. IEEE Software, Vol. 4, No. 5, 19 – 25.

Mills H., O'Neill D., Linger R., Dyer M., The management of software engineering. IBM Systems Journal, Vol. 24, No. 2, 414 – 477.

Naur P., Randell B. (toim.), Software engineering: A report on a conference sponsored by the NATO Science Committee. NATO, 1969.

OPEN, http://www.csse.swin.edu.au/cotar/OPEN/OPEN.html, 17.9.1998.

Parnas D., Clements P., A rational design process: How and why to fake it. IEEE Transactions on Software Engineering, Vol. 12, No. 2, February 1986, 251 – 257.

Piaget J., Inhelder B., Lapsen psykologia. K. J. Gummerus Oy, 1977.

Piaget J., Lapsi maailmansa rakentajana. WSOY, 1988.

Prechelt L., Unger B., Tichy W., Brössler P., A controlled experiment in maintenance comparing design patterns to simpler solutions. http://wwwipd.ira.uka.de/~exp/Biblio/patmain.ps.gz, 10.3.1998.

Pree W., Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

Pressman R., Software Engineering: A Practitioner's Approach 3rd ed. McGraw-Hill, 1992.

Ram J., Raman A., Guruprasad K., A Pattern Oriented Technique for Software Design. Software Engineering Notes, Vol. 22, No. 4, July 1997, 70 – 73.

Ram J., Raman A., Guruprasad K., Raman S., A Methodology for Constructing a Design Handbook for Object Oriented Systems. Teoksessa Proceedings of the Third Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 1996, Vol. 9, Chapter 5, Section 9.5.1. Löytyy myös http://siesta.cs.wustl.edu/~schmidt/PLoP-96/program.zip, 20.9.1998.

Riehle D., Züllighoven H., Understandign and using patterns in software development. Theory and Practice of Object Systems, Vol. 2, No. 1, 1996.

Royce W., Managing the development of large software systems: concepts and techniques. Proceedings VESCON, elokuu, 1970.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modeling and Design, Prentice Hall, 1991.

Rumelhart D., Understanding and summarizing brief stories. Teoksessa Laberge D., Samuels S. (toim.), Basic Processes in Reading: Perception and Comprehension. Lawrence Erlbaum Associates, 1977, 265 – 303.

Saariluoma P., Historiallinen johdatus kognitiotieteeseen. Teoksessa Hautamäki A. (toim.), Kognitiotiede. Oy Gaudeamus Ab, 1988a, 15 – 42.

Saariluoma P., Ihmisen muisti. Teoksessa Hautamäki A. (toim.), Kognitiotiede. Oy Gaudeamus Ab, 1988b, 71 – 99.

Saarinen E., Länsimaisen filosofian historia huipulta huipulle Sokrateesta Marxiin. WSOY, 1985.

Sakkinen M., Inheritance and Other Main Principles of C++ and Other Object-oriented Languages. Jyväskylän yliopisto, 1992.

Sametinger J., Software Engineering with Reusable Components. Springer-Verlag, 1997.

Schank R., Abelson R., Scripts Plans Goals and Understandign: An Inquiry into Human Knowledge Structures. Lawrence Erlbaum Associates Inc. 1977.

Schmid H., Creating the architecture of a manufacturing framework by design patterns. Proceedings OOPSLA'95, SIGPLAN Notices, Vol 30, No. 10, October 1995, 370 – 384.

Schmidt D., Experience using design patterns to develop reusable object-oriented communication software. Communications of ACM, Vol. 38, No. 10, October 1995.

Schmidt D., Stephenson P., Experience using design patterns to evolve communication software across diverse OS platforms. Proceedings of the 9th ECOOP'95, 1995.

Shull F., Melo W., Basili V., An inductive method for discovering design patterns from object-oriented software systems. Technical Report, University of Maryland, 1996, http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/OOPD_VAL.DOC.pdf, 20.9.1998.

Sommerville I., Software Engineering Fourth Edition, Addison-Wesley, 1992.

Sommerville I., Software process models, ACM Computing Surveys, Vol. 28, No. 1, March 1996, 269 – 271.

Stein L., Delegation is inheritance, OOPSLA '87 Proceedings, October 4-8 1987, 138 – 146.

Taivalsaari A., A Critical View of Inheritance and Reusability in Object-oriented Programming, Jyväskylän yliopistopaino ja Sisäsuomi Oy, 1993.

UML Summary, version 1.1, 1 September 1997. http://www.rational.com/uml, 18.8.1998.

UML Notation Guide, version 1.1, 1 September 1997. http://www.rational.com/uml, 18.8.1998.

UML Semantics, version 1.1, 1 Septempber 1997. http://www.rational.com/uml, 18.8.1998.

Villeneuve A., Fedorowicz J., Understanding expertise in information systems design, or, What's all the fuss about objects? Decision Support Systems, Vol. 21, 1997, 111-131.

Vlissides J., Reverse Architecture. Position paper for Software Architectures Seminar, Schloss Dagstuhl, Germany, 1995. ftp://st.cs.uiuc.edu/pub/patterns/papers/revarch.ps, 20.9.1998.

Vlissides J., Patterns: The top ten misconceptions. Object Magazine, Vol. 7, No. 1, maaliskuu, 1997, 30-33.
Löytyy myös http://www.sigs.com/publications/docs/objm/9703/9703.vlissides.html, 11.2.1998.

Vlissides J., Coplien J., Kerth N. (toim.), Pattern Languages of Program Design 2. Addison-Wesley, 1995.

Wegner P., Concepts and paradigms of object-oriented programming. ACM OOPS Messenger, Vol. 1, No. 1, August 1990, 7 – 87.

Weinand A., Gamma E., Marty R., Design and implementation of ET++, a seamless object-oriented application framework. Structured Programming, Vol. 10, No. 2, 1989, 63-87. Löytyy myös ftp://ftp.ubilab.ubs.ch/pub/ET++/paper/ETStrucProg89.ps.gz, 22.9.1998.

Wirfs-Brock R., Wilkerson B., Wiener L., Designing Object-Oriented Software. Prentice-Hall, 1990.

Zimmer W., Relationships between design patterns. Teoksessa Coplien J., Schmidt D., Pattern Language of Program Design. Addison-Wesley, 1995, 345-364.

Zimmer W., Experiences using Design Patterns to Reorganize an Object-Oriented Application. Teoksessa Casais E. (toim.), Architectures and Processes for Systematic Software Construction. FZI-Publication 1/95, Forschungszentrum Informatik Karlsruhe, 1995. Löytyy myös ftp://ftp.fzi.de/pub/PROST/papers/dp_experiences.ps.Z, 16.7.1998.