Suunnittelumallien systemaattinen yhdistäminen

Jussi Lehikoinen

23.5.2007

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Suunnittelumallit esittävät aiemmin hyväksi havaittuja ratkaisuja ohjelmistosuunnittelun ongelmiin. Suunnittelumallien yksittäisen ja sattumanvaraisen käytön sijaan tulisi pyrkiä kuitenkin niiden tehokkaampaan hyödyntämiseen. Systemaattinen yhdistäminen vie ohjelmistosuunnittelun vielä askeleen eteenpäin kohti entistä uudelleenkäytettävämpiä oliopohjaisia ohjelmistoja. Itseasiassa ohjelmistot voisi jo alusta pitäen suunnitella käyttäen suunnittelumalleja. Suunnittelumallien yhdistämiseksi ei ole kuitenkaan vielä olemassa yhtä standardia prosessia. Täten tutkielmassa tarkastellaan eri yhdistämistekniikoita ja UML-mallikielen tukea suunnittelumalleille, pyrkien samalla antamaan yleiskuva tutkimuksesta tällä alalla. Lopuksi yhdistämisestä esitetään lyhyt esimerkki POAD-tekniikkaa käyttäen ja tutustutaan samalla suunnittelumallien käyttöä tukevaan mallinnustyökaluun, Enterprise Architectiin.

ACM-luokat (ACM Computing Classification System, 1998 version): D.1.5, D.2.2, D.2.11

Avainsanat: Mallinnustyökalu, POAD, suunnittelumallit, suunnittelumallien systemaattinen yhdistäminen, UML.

Sisältö

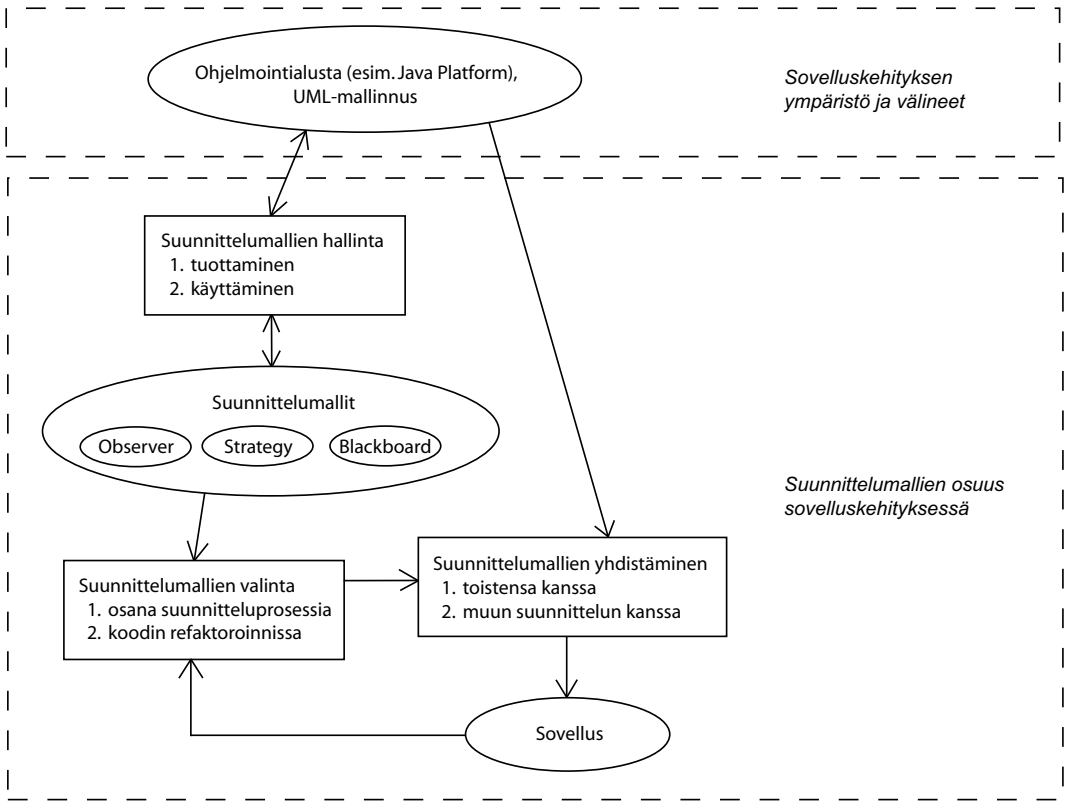
1 Johdanto

Sovelluskehityksen vaikein vaihe ei ole koodaaminen vaan suunnittelu, sillä sen päätökset vaikuttavat koko ohjelmiston elinkaaren ajan. Suunnitteluvaiheen merkitys korostuu, kun pyritään uudelleenkäytettävyyteen, laiteriippumattomuuteen ja siirrettävyyteen. Eräs viimeisimmistä olio-ohjelmoinnin ratkaisuista suunnitteluvaiheen ongelmiin ovat *suunnittelumallit* (design patterns), jotka on muotoiltu ratkaisemaan tietyissä konteksteissa esiintyviä yleisiä suunnitteluongelmia. Suunnittelumallit auttavat valitsemaan vaihtoehtoja, jotka tekevät ohjelmistosta uudelleenkäytettävän ja välttämään ratkaisuja, jotka vähentävät uudelleenkäytettävyyttä. Uudelleenkäytettävyys varmistaa sen, ettei tehdä tarpeetonta suunnittelu- ja toteutustyötä (Gamma & al., 1995).

Jo pelkkä yksittäisten suunnittelumallien satunnainen käyttö ohjelmiston suunnitteluvaiheessa voi parantaa ohjelmiston laatua. Suunnittelumallien yhdistäminen mahdollistaa korkeamman tason uudelleenkäytettävyyden kuin vain yksittäisten suunnittelumallien käyttö. Suunnittelumallien ja niiden yhdistelmien mallinnus ja esittäminen perustuu yleensä olio-ohjelmoinnin mallinnustekniikoihin, jotka käyttävät graafista kuvauskieltä, kuten *UML* (Unified Modeling Language) (Dong, 2003). UML-standardi tarjoaa erityisesti tähän tehtävään soveltuvan *yhteistyönotaation* (collaboration) (OMG, 2007).

Kuva 1 havainnollistaa suunnittelumallien käyttöä ja niiden yhdistämistä osana sovelluskehitystä. Sovelluskehityksen apuna voidaan käyttää erilaisia välineitä ja tekniikoita, jotka tarjoavat mallinnustekniikan ja tuen suunnittelumalleille. Sovelluskehitysympäristö saattaa jo itsessään sisältää suunnittelumallikirjaston, josta kehittäjä voi ottaa ne käyttöön, olipa kyse sitten ohjelmiston suunnitteluvaiheesta tai koodin *refaktoroinnista* (refactoring). Suunnittelua kehitetään sitten valitun metodologian mukaisesti suunnittelumalleja toisiinsa ja muuhun suunnitteluun yhdistäen.

Suunnittelumallien systemaattiseksi yhdistämiseksi on ainakin kaksi lähestymistapaa. *Mallikielet* (pattern languages) kuvaavat tietyn ongelma-alueen suunnittelumallien lisäksi niiden väliset suhteet. Niitä käyttäen voidaan ratkaista joukko suunnitteluongelmia. Toinen mahdollisuus on kehitysprosessi, joka määrittelee yhdistämisen lähestymistavan, analyysin ja suunnittelun vaiheet ja parhaassa tapauksessa myös mallit ja työkalut automatisoimaan kehitysprosessi. Systemaattisen yhdistämisen tueksi pitää määritellä yhdistämistekniikat ja niitä tukevat mallinnuskielet ja näkymät, jotka tuke-



Kuva 1: Suunnittelumallien yhdistäminen osana sovelluskehitystä.

vat UML:ää (Yacoub & Ammar, 2004).

Tämän tutkielman tarkoituksena on erityisesti käsitellä suunnittelumallien systemaattista yhdistämistä. Tutkielmassa oletetaan, että yleiset ohjelmistotuotannon asiat, kuten esimerkiksi UML-kieli (OMG, 2005; OMG, 2007) ja ohjelmistoprojektin eri vaiheet, tunnetaan. Lisäksi oletetaan tunnetuiksi perustiedot olio-ohjelmoinnista. Tutkielmassa ei oleteta suunnittelumallien ja niiden käytön tuntemusta. Täten luvussa 2 kerrotaan suunnittelumalleista oleelliset tiedot, jotta lukija voisi ymmärtää tässä tutkielmassa tarkasteltavia suunnittelumallien yhdistämistekniikoita. Luku 3 esittelee nämä yhdistämistekniikat niiden luonteen mukaisesti neljässä osassa: mallikielet, käytökselliset yhdistämistekniikat, rakenteelliset yhdistämistekniikat ja UML 2.1.1 -tuki suunnittelumalleille. Luvussa 4 tutustutaan lyhyesti Enterprise Architect -mallinnustyökaluun, joka tukee suunnittelumallien käyttöä ja yhdistämistä tarjoamalla suunnittelumalleista valmiita UML-muotoisia luokkakaavioita. Luku 5 tekee yhteenvedon esitellyistä yhdistämistekniikoista ja pohtii suunnittemallien yhdistämisen tulevaisuutta.

2 Suunnittelumallit

Mallit (patterns) sijoittuvat yleisellä tasolla esittämänsä suunnitteluratkaisun rakeisuuden mukaan ohjelmistokehysten ja luokkakirjastojen välimaastoon (Yacoub & Ammar, 2004). Yleisesti malli kuvaa suunnittelussa ja toteutuksessa esiintyvän ongelman ja sen ratkaisun. Malleja voidaan luokitella ohjelmiston kehitysvaiheen mukaan, missä niitä käytetään. *Analyysimalleja* (analysis patterns) (Fowler, 1997) käytetään nimensä mukaisesti vaatimusmäärittelyn analysointiin, *arkkitehtuurimalleja* (architecture patterns) (Buschmann & al., 1996) suunniteltaessa ohjelmiston perusrakennetta ja *idiomeja* (idioms) (Coplien, 1992) toteutusvaiheessa kuvaamaan mallien ratkaisuja ohjelmointikielikohtaisesti. Suunnittelumallien käyttö sijoittuu pääasiassa suunnitteluvaiheeseen mutta niitä voidaan hyödyntää myös toteutusvaiheessa. Näin varsinkin Unified-prosessissa (Larman, 2002) ja muissa sovelluskehitysprosesseissa, joissa suunnittelua kehitetään myös toteutuksen aikana. Myös refaktoroinnissa voidaan käyttää suunnittelumalleja apuna (Kerievsky, 2005).

Suunnittelumallit[1] kuvaavat yksinkertaisia ja elegantteja ratkaisuja tiettyihin oliopohjaisen ohjelmistosuunnittelun ongelmiin. Suunnittelumalleissa kuvataan ratkaisuja, jotka on kehitetty ja jalostettu pitkän ajan kuluessa. Täten ne eivät ole ratkaisuja, joita ihmisillä on tapana ensimmäiseksi kokeilla. Suunnittelumallit heijastavat lukemattomia uudelleensuunnittelu- ja uudelleenkoodauskierroksia, joilla kehittäjät ovat pyrkineet parantamaan ohjelmistojensa uudelleenkäytettävyyttä ja joustavuutta. Suunnittelumalleihin on vangittu nämä ratkaisut ytimekkäässä ja helposti sovellettavassa muodossa (Gamma & al., 1995). Idea suunnittelumallista on lainattu ohjelmistotuotantoon Alexanderilta & al. (1977): ”Jokainen ratkaisumalli kuvaa ongelman, joka toistuu jatkuvasti ympäristössämme ja määrittelee ongelmalle ratkaisuperiaatteen, jota voidaan soveltaa miljoonia kertoja aina uudella tavalla”. Alexander & al. tarkoittivat rakennusten ja kaupunkien suunnittelussa käytettäviä malleja, mutta heidän määritelmänsä sopii myös oliopohjaisiin suunnittelumalleihin, jolloin käsitellään olioita ja rajapintoja seinien ja ovien sijasta. Molempien suunnittelumallien ideana on kuitenkin esittää ratkaisu yleiseen ongelmaan. Tässä luvussa esittelen mitä suunnittelumallit ovat ja miksi ne ovat olemassa. Lopuksi esitetään lyhyesti suunnittelumallien yleisiä käyttöperiaatteita.

[1] Jatkossa käytetään termiä suunnittelumalli englanninkielisistä vastineista pattern ja design pattern, jotta voidaan tehdä ero yleiseen termiin malli, jonka englanninkielinen vastine on model.

2.1 Suunnittelumallin kuvaaminen

Gamman & al. (1995) mukaan suunnittelumalli koostuu yleisellä tasolla neljästä keskeisestä osasta:

1. *Suunnittelumallin nimi* kuvaa muutamalla sanalla suunnittelun kohteena olevan ongelman, sen ratkaisun ja ratkaisun seuraamukset.

2. *Ongelma* ja ympäristö, jossa se esiintyy, kuvaavat suunnittelumallin soveltamiskohteita.

3. *Ratkaisussa* kuvataan elementit, joista suunnitteluratkaisu koostuu, niiden vastuut, niiden väliset suhteet ja niiden keskinäinen yhteistyö.

4. *Seuraukset* kuvaavat suunnittelumallin soveltamisen tuloksia ja sen hyötyjä sekä haittoja.

Mikä tahansa malli ei käy suunnittelumalliksi. Suunnittelumalleiksi ei lueta sellaisia malleja, jotka voidaan koodata luokiksi ja joita voidaan uudelleenkäyttää sellaisenaan. Myöskään monimutkaiset toimialakohtaiset, kokonaisia sovelluksia tai alijärjestelmiä koskevat suunnitteluratkaisut eivät ole suunnittelumalleja. Suunnittelumallit ovat kuvauksia keskenään vuorovaikutuksessa olevista olioista ja luokista, jotka on muotoiltu ratkaisemaan tietyssä yhteydessä esiintyvä yleinen suunnitteluongelma (Gamma & al., 1995). Suunnittelumalli nimeää, abstrahoi ja määrittää yleisen suunnittelurakenteen avainkohdat, jotka hyödyttävät uudelleenkäytettävän oliopohjaisen suunnitteluratkaisun luomista. Suunnittelumalli määrittää osallistuvat luokat ja ilmentymät, niiden roolit ja yhteistyön sekä vastuiden jakaantumisen. Kukin suunnittelumalli keskittyy tiettyyn oliopohjaisen suunnittelun ongelmaan tai kohteeseen. Suunnittelumalli kertoo missä tilanteissa se on sovellettavissa ja voidaanko sitä soveltaa muiden suunnittelurajoitteiden yhteydessä sekä mitkä ovat sen seuraukset ja hyvät ja huonot puolet. Koska ratkaisu on jossain vaiheessa myös toteutettava, suunnittelumalli sisältää lisäksi toteutustapaa havainnollistavan koodiesimerkin.

Vaikka suunnittelumallit kuvaavat oliopohjaisia suunnitteluratkaisuja, ne perustuvat käytännön ratkaisuihin, joita on lähinnä toteutettu yleisimmillä olio-ohjelmointikielillä kuten C++ ja Java. Ohjelmointikielen valinnalla on vaikutusta, koska se ohjaa suunnittelijan näkökulmaa ongelman ratkaisussa. Gamma & al. mukaan riippuu kielestä millainen suunnittelumalli toimii parhaiten, sillä kielen ominaisuudet määräävät mitä

suunnittelumallia käytetään. Jotkin suunnittelumalleista on helpompi ilmaista toisella kielellä kuin toisella. Tässä tutkielmassa käsitellään pääasiassa Java-ohjelmoinnin suunnittelumalleja.

Gamma & al. (1995) esittivät suunnittelumallien kuvaamista erityisen yhtenäisen formaatin avulla, jossa suunnittelumalli jaetaan osiin tietyn rungon mukaisesti. Alla on esitelty eräs 13 kohtaa sisältävä yleisen tason kuvausta tarkentava esimerkkirunko Gammaa & al. mukaillen:

- *Suunnittelumallin nimi ja luokka:* Suunnittelumallin nimi kertoo sen olemuksen ytimekkäästi. Hyvä nimi on erittäin tärkeä, sillä siitä tulee osa suunnittelusanastoa. Koska suunnittelumalleja on paljon, ne on jäsenneltävä jotenkin. Yhteenkuuluvat suunnittelumallit voidaan ryhmitellä niiden käyttötarkoituksesta ja -tavasta riippuen omiksi malliperheikseen: *luontimallit*, *rakennemallit* ja *käytösmallit*.
- *Tarkoitus:* Lyhyt lause, joka vastaa seuraaviin kysymyksiin: Mitä suunnittelumalli tekee? Mikä on sen perusajatus ja tarkoitus? Mihin tiettyyn suunnittelukohteeseen tai -ongelmaan se soveltuu?
- *Alias:* Suunnittelumallin muita yleisesti tunnettuja nimiä, jos sellaisia on.
- *Perustelut:* Esimerkkitilanne, jossa kuvataan jotain suunnitteluongelmaa ja se, miten tarkasteltavan suunnittelumallin luokka- ja oliorakenteet ongelman ratkaisevat. Esimerkki auttaa ymmärtämään rungon seuraavissa kohdissa esiteltävän abstraktimman kuvauksen.
- *Soveltuvuus:* Mihin tilanteisiin suunnittelumallia voidaan soveltaa? Esimerkkejä huonoista suunnitteluratkaisuista, joita suunnittelumallilla voidaan korjata, ja siitä, miten nämä tilanteet tunnistetaan.
- *Rakenne:* Suunnittelumallin luokkien esitys graafisella kuvauskielellä, kuten UML.
- *Osallistujat:* Suunnittelumalliin liittyvät luokat ja/tai oliot sekä niiden vastuut.
- *Yhteistyösuhteet:* Miten osallistujat toimivat yhteistyössä toteuttaakseen vastuunsa.

- *Seuraukset:* Miten hyvin suunnittelumalli tukee tavoitteitaan? Mitä etuja suunnittelumallin käytöllä saavutetaan ja mitä menetetään? Mitä systeemirakenteita suunnittelumalli sallii muutettavan riippumattomasti?

- *Toteutus:* Mitkä sudenkuopat, vinkit ja tekniikat on tiedettävä suunnittelumallia toteutettaessa. Onko eri ohjelmointikielillä tehtyjen toteutusten välillä ohjelmointikielistä johtuvia eroja?

- *Mallikoodia:* Koodiesimerkkejä suunnittelumallin toteutuksesta jollain ohjelmointikielellä, kuten Java tai C++.

- *Tunnettuja käyttökohteita:* Esimerkkejä suunnittelumallin käytöstä todellisissa järjestelmissä.

- *Läheiset suunnittelumallit:* Mitkä suunnittelumallit liittyvät läheisesti tähän suunnittelumalliin? Mitkä ovat tärkeimmät erot? Minkä muiden suunnittelumallien kanssa tätä suunnittelumallia tulisi käyttää? Suunnittelumallien yhdistämisen kannalta tämä on erityisen tärkeä kohta.

Mallien kuvaukset voivat olla erilaisia kirjallisuuslähteestä riippuen, jolloin asia löytyy mahdollisesti eri nimisten kohtien alta. Esimerkki suunnittelumallin kuvauksesta liitteessä 1 esittää *Rekursiokoosteen* (Composite) edellä esitellyn yleisen tason neljän kohdan rungon (mallin nimi, ongelma, ratkaisu, seuraukset) mukaisesti täydennettynä läheisillä suunnittelumalleilla.

Suunnittelumalleja on paljon erilaisia ja niistä on koottu useita erilaisia kokoelmia, joista eräs varhaisimmista on Gamman & al. (1995) esittämä. Myös ohjelmointikielikohtaisia kokoelmia löytyy useimmille oliokielille, esimerkiksi Javalle (Grand, 2002). Erilaisten suunnittelumallien käyttöä tukevien sovelluskehitysympäristöjen mukana tulee usein omat mallikirjastot, josta eräs esimerkki esitellään luvussa 4. Näiden lisäksi on sovellusaluekohtaisia mallikieliä, jotka määrittelevät suunnittelumallien väliset yhteydet tavallisia hakemistoja tarkemmin. Mallikielistä kerrotaan tarkemmin seuraavan luvun kohdassa 3.2. Yleensä nämä erilaiset mallikokoelmat ovat toisistaan erillään, mikä vaikeuttaa suunnittelumallien löytämistä. Olemassaolevien suunnittelumallien indeksoinnin käynnistyminen 2000-luvun alussa tähtääkin yhtenäisen mallien tietokannan kokoamiseen (Yacoub & Ammar, 2004).

2.2 Suunnittelumallien käyttötarkoitus

Suunnittelumallien kautta löytyy ratkaisu moniin ongelmiin, joita järjestelmäsuunnittelijat kohtaavat jokapäiväisessä työssään. Seuraavaksi esitellään Gamman & al. (1995) tunnistamia tavallisimpia ongelmia ja ratkaisuja, joita suunnittelumallit tarjoavat näihin ongelmiin. Lopuksi tarkastellaan suunnittelumallien roolia kolmen laajan ohjelmistoryhmän yhteydessä. Nämä ryhmät ovat sovellusohjelmat, työkalupakit ja ohjelmistokehykset.

2.2.1 Sopivien olioiden löytäminen

Oliopohjaisen suunnittelun vaikein tehtävä on järjestelmän pilkkominen olioiksi. Tehtävä on vaikea, koska siihen vaikuttaa monta tekijää, joista tärkeimpinä kapselointi, karkeustaso, riippuvuudet, muunneltavuus, suorituskyky, systeemin muutokset ja uudelleenkäyttö. Nämä kaikki vaikuttavat oliojakoon, usein ristiriitaisella tavalla. Oliopohjaiset suunnittelumenetelmät suosivat useita erilaisia lähestymistapoja. Monet suunnitteluratkaisun oliot löytyvät analyysivaiheessa, mutta oliopohjaisiin ratkaisuihin sisältyy usein myös luokkia, joilla ei ole vastaavuutta reaalimaailmassa. Esimerkiksi olioita, jotka esittävät prosesseja tai algoritmeja, ei esiinny luonnossa, mutta silti ne ovat välttämätön osa joustavaa suunnitteluratkaisua. Tällaiset oliot löytyvät harvoin analyysivaiheessa tai edes toteutusratkaisun alkuvaiheessa. Ne ilmaantuvat vasta kun suunnitteluratkaisua työstetään joustavammaksi ja paremmin uudelleenkäytettäväksi.

Tarkka pitäytyminen reaalimaailmaan tuottaa järjestelmän, joka kuvaa tämän päivän realiteetteja, mutta ei välttämättä ota huomioon huomisen tarpeita. Suunnitteluvaiheessa löytyvät abstraktiot ovat avainasemassa, kun ratkaisuun pyritään saamaan joustavuutta. Suunnittelumallit auttavat löytämään vähemmän ilmeisiä abstraktioita ja olioita, joilla ne voidaan esittää.

2.2.2 Olion karkeustason määrittely

Olioiden koko ja lukumäärä voivat vaihdella paljon. Olioilla voidaan esittää mikä tahansa laitteistokomponenteista kokonaisiin sovelluksiin. Suunnittelumallit auttavat tekemään valintoja olioesityksen suhteen. Gamman & al. (1995) esittämät suunnittelumallit tukevat tätä seuraavasti. *Julkisivu* (Facade) määrittelee miten kokonainen

alisysteemi kuvataan oliona. *Hiutale* (Flyweight) näyttää miten käsitellään suurta joukkoa hyvin hienojakoisia olioita. Jotkut suunnittelumallit määrittelevät erityisiä tapoja jakaa olio pienemmiksi olioiksi. *Abstrakti tehdas* (Abstract Factory) ja *Rakentaja* (Builder) tuottavat olioita, joiden ainoana tehtävänä on luoda toisia olioita. *Vierailija* (Visitor) ja *Komento* (Command) luovat olioita, joiden ainoana tehtävänä on toteuttaa toiseen olioon tai olioryhmään kohdistuva pyyntö.

2.2.3 Olion rajapintojen määrittely

Suunnittelumalleissa kuvataan rajapintojen tärkeimmät osat ja se, millaista tietoa rajapinnalle lähetetään. Suunnittelumalli voi myös kertoa, mitä rajapintaan ei kannata laittaa. *Muisto* (Memento) (Gamma & al., 1995) on tästä hyvä esimerkki. Siinä kuvataan, miten olion sisäinen tila kapseloidaan ja talletetaan siten, että olion voi myöhemmin palauttaa aikaisempaan tilaansa. Suunnittelumalli vaatii, että Muisto-olioilla on oltava kaksi rajapintaa: julkinen rajapinta, joka sallii asiakkaiden kopioida ja käyttää muistoja, sekä yksityinen rajapinta, jota vain alkuperäinen olio voi käyttää tilan tallentamiseen ja hakemiseen muistosta. Suunnittelumalleissa käsitellään myös rajapintojen välisiä suhteita. Suunnittelumalli saattaa vaatia, että joillain luokilla on samanlaiset rajapinnat, tai asettaa luokkien rajapinnoille rajoituksia. Esimerkiksi *Kuorruttaja* (Decorator) ja *Edustaja* (Proxy) vaativat, että Kuorruttaja- ja Edustaja-olioiden rajapinnat ovat identtiset kuorrutettujen ja edustettujen olioiden kanssa. Vierailija-rajapinnan puolestaan täytyy edustaa kaikkia olioluokkia, joissa vierailija voi vierailla.

2.2.4 Muunneltavuuden huomioiminen suunnittelussa

Uusien vaatimusten ja nykyvaatimuksiin kohdistuvien muutosten ennakoiminen ja sovelluksen suunnittelu mahdollisia muutoksia ajatellen on tärkeää uudelleenkäytettävyyden takia. Jotta suunnitteluratkaisusta tulisi muutoksia kestävä, on mietittävä, mitä muutostarpeita järjestelmään saattaa kohdistua sen elinaikana.

Suunnittelumallien avulla uudelleensuunnittelun tarvetta voidaan välttää. Suunnittelumallien käyttö varmistaa sen, että sovellus voi muuttua määrätyillä tavoilla. Jokainen suunnittelumalli mahdollistaa joidenkin systeemirakenteiden muuttamisen toisista riippumatta, mikä tekee systeemin immuuniksi vastaaville muutoksille.

2.2.5 Sovellusohjelmat

Rakennettaessa *sovellusohjelmia* (application programs), esimerkiksi tekstieditoreja tai taulukkolaskentaohjelmistoja, ovat ohjelmien *sisäinen* uudelleenkäyttö, ylläpidettävyys ja laajennettavuus tärkeitä tavoitteita (Gamma & al., 1995). Sisäinen uudelleenkäyttö varmistaa sen, ettei tehdä tarpeetonta suunnittelu- ja toteutustyötä. Suunnittelumallit, jotka vähentävät riippuvuuksia, lisäävät sisäistä uudelleenkäyttöä. Löyhä sidonta parantaa mahdollisuuksia käyttää yhtä luokkaa yhteistyössä useiden muiden kanssa. Kun esimerkiksi operaatiojoukon välisiä riippuvuuksia eliminoidaan eristämällä ja kapseloimalla kukin operaatio, operaatioiden uudelleenkäyttö erilaisissa yhteyksissä helpottuu. Myös algoritmi- ja esitysmuotoriippuvuuksien eliminointi on hyödyksi. Suunnittelumalleja voidaan käyttää myös alustariippuvuuksien rajoittamiseen ja sovelluksen kerrostamiseen. Mallit parantavat laajennettavuutta tarjoamalla keinoja luokkahierarkioiden laajentamiseen ja olioiden koostamiseen (Gamma & al., 1995).

2.2.6 Työkalupakit

Usein sovellukseen sisällytetään luokkia valmiiksi määritellyistä luokkakirjastoista eli *työkalupakeista* (toolkits) (Gamma & al., 1995). Työkalupakki on joukko yhteen kuuluvia, uudelleenkäytettäviä luokkia, joiden toiminnallisuus on yleiskäyttöistä. Esimerkki työkalupakista on luokkakokoelma listojen, pinojen yms. rakenteiden käsittelyyn. Työkalupakit tarjoavat toiminnallisuutta, joka auttaa sovellusta toteuttamaan tehtävänsä. Niitä käyttämällä ohjelmoija välttyy tavallisten funktioiden uudelleenkoodaukselta. Työkalupakit korostavat uudelleenkäyttöä. Työkalupakin suunnittelu on selvästi vaikeampaa kuin sovelluksen suunnittelu, koska työkalupakin täytyy toimia monissa eri sovelluksissa ollakseen hyödyllinen. Työkalupakin tekijä ei voi tietää, mitä sovellukset tekevät ja mitkä ovat niiden erityistarpeet. Täten on ensiarvoisen tärkeää välttää oletuksia ja riippuvuuksia, jotka voisivat rajoittaa työkalupakin joustavuutta, sovellettavuutta ja tehokkuutta. Suunnittelumallit helpottavat tätä tehtävää.

2.2.7 Ohjelmistokehykset

Ohjelmistokehys (framework) on joukko yhteistyössä toimivia luokkia, jotka muodostavat uudelleenkäytettävän suunnitteluratkaisun tietyntyyppisen sovellusalueen tarpeisiin (Deutsch, 1989). Ohjelmistokehys voi olla suunniteltu esimerkiksi graafisten käyttöliittymien laatimiseen. Kehyksestä tuotetaan yksittäinen sovellus luomalla kehyksen abstrakteista luokista sovelluskohtaisia aliluokkia. Kehys sanelee sovelluksen arkkitehtuurin määräämällä kokonaisrakenteen, sen jaon luokkiin ja olioihin, näiden päävastuut ja yhteistyötavan ja kontrollin kulun. Kehys määrittelee nämä suunnitteluparametrit valmiiksi ja suunnittelija voi keskittyä sovelluskohtaisiin kysymyksiin. Ohjelmistokehys kokoaa oman suunnittelualueensa yleiset suunnitteluratkaisut.

Ohjelmistokehykset korostavat täten enemmänkin suunnittelun kuin koodin uudelleenkäyttöä, vaikka kehykset sisältävätkin myös konkreettisia aliluokkia, joita voidaan käyttää sellaisenaan. Suunnitteluratkaisun uudelleenkäyttö kääntää sovelluksen ja ohjelmiston välisen kontrollin suunnan. Ohjelmistokehystä käytettäessä uudelleenkäytetään runkoa ja koodataan itse se osuus, jota runko kutsuu. Ohjelmoijan on annettava operaatioille ennalta sovitut nimet ja kutsumallit, mutta tämä vähentää osaltaan tehtävien suunnittelupäätösten määrää. Kehyksiä käyttämällä sovellusten rakentaminen nopeutuu ja sovellusten rakenteista tulee yhdenmukaisia. Sovellukset ovat helpommin ylläpidettäviä ja näyttävät käyttäjille päin yhdenmukaisemmilta. Sovelluksiin ja työkalupakkeihin verrattuna ohjelmistokehykset ovat kaikkein vaikeimpia suunniteltavia. Kehyksen suunnittelija yrittää arvioida, millainen arkkitehtuuri sopisi kaikille ko. sovellusalueen sovelluksille. Koska sovellukset ovat täysin riippuvaisia kehyksestä, ne ovat myös erittäin herkkiä kaikille kehyksen rajapinnoissa tapahtuville muutoksille. Kun kehystä muutetaan, sovellusta on muutettava vastaavasti. Löyhä sidonta on näin ollen välttämätöntä, koska muuten pienimmätkin kehykseen tehdyt muutokset aiheuttavat mittavia johdannaisseuraamuksia.

Suunnittelumalleista koostuvan kehyksen suunnittelun ja koodin uudelleenkäytettävyys on todennäköisesti parempi kuin kehyksen, joka ei käytä suunnittelumalleja. Suunnittelumalleja käytetään sekä kehyksen suunnitteluun että sen rakennuspalikoina, jolloin kehys voi koostua hyvinkin pitkälti suunnittelumalleista. Suunnittelumallit tekevät kehyksen arkkitehtuurin paremmin useisiin sovelluksiin soveltuvaksi. Lisäksi suunnittelumallien käyttö kehysten suunnittelussa parantaa kehysten ymmärrettävyyttä ja dokumentaatiota (Yacoub & Ammar, 2004). Tällä tavoin henkilöt, jotka tuntevat

suunnittelumallit, saavat kehyksen toiminnasta nopeasti tarkan kuvan. Myös suunnittelumalleja tuntemattomat henkilöt hyötyvät rakenteesta, jossa sunnittelun avainkohdat esitetään eksplisiittisesti.

Suunnittelumalleilla ja ohjelmistokehyksillä on selviä yhtäläisyyksiä. Ne eroavat toisistaan kolmella tärkeällä tavalla (Gamma & al., 1995):

1. *Suunnittelumallit ovat abstraktimpia kuin kehykset.* Kehykset voidaan esittää ohjelmakoodina, mutta suunnittelumalleista voidaan antaa vain esimerkkejä ohjelmakoodin avulla. Kehysten vahvuutena on se, että ne voidaan kirjoittaa valmiiksi ohjelmointikielellä ja niitä voidaan paitsi opiskella myös suoraan ajaa ja uudelleenkäyttää. Suunnittelumallit on kuitenkin koodattava joka kerta kun niitä käytetään. Suunnittelumallit toisaalta selittävät suunnitteluratkaisujen tarkoituksen, seuraukset ja niiden hyvät ja huonot puolet.

2. *Suunnittelumallit ovat pienempiä arkkitehtuurinosia kuin kehykset.* Tyypillinen kehys sisältää useita suunnittelumalleja, mutta suunnittelumalli ei voi sisältää kehystä.

3. *Suunnittelumallit ovat vähemmän erikoistuneita kuin kehykset.* Kehykset liittyvät aina johonkin tiettyyn sovellusalueeseen. Graafisen editorin kehystä saatetaan käyttää tehdassimuloinnissa, mutta sitä ei voida pitää simulointikehyksenä. Sitä vastoin suunnittelumalleja voidaan käyttää lähes kaikissa sovelluksissa.

2.3 Suunnittelumallin valinta

Voi olla vaikeaa löytää tiettyä ongelmaa vastaava suunnittelumalli monien muiden joukosta etenkin, jos suunnittelumallit eivät ole entuudestaan tuttuja. Seuraavassa esitetään muutamia erilaisia lähestymistapoja sopivan suunnittelumallin löytämiseen Gammaa & al. (1995) mukaillen.

- *Mietitään miten suunnittelumallit ratkaisevat suunnitteluongelmia.* Kuten kohdassa 2.2 esitettiin, suunnittelumallit auttavat löytämään ratkaisuja suunnitteluongelmiin. Niiden kuvausten ja ratkaisujen tutkiminen voi auttaa sopivan suunnittelumallin etsimisessä.

- *Käydään läpi Tarkoitus-kohdat.* Suunnittelumallien kuvauksissa on Tarkoitus-kohta, jossa kerrotaan mitä suunnittelumalli tekee, mikä on sen perusajatus ja tar-

koitus ja mihin tiettyyn suunnittelukohteeseen tai -ongelmaan se soveltuu. Kannattaa siis lukea Tarkoitus-kohta ja selvittää, mikä kuulostaa oleelliselta ratkaistavan suunnitteluongelman suhteen. Etsintää voi yrittää myös rajata kiinnittämällä huomiota suunnittelumallien luokitteluun.

- *Tutkitaan miten suunnittelumallit liittyvät toisiinsa.* Suunnittelumallien välisiä suhteita voi selvittää niiden kuvausten Läheiset suunnittelumallit -kohdan avulla (esimerkki Rekursiokoosteen kuvauksessa liitteen 1 lopussa) ja esimerkiksi tarkastelemalla graafisia suhdekarttoja. Liite 2 esittää erään esimerkin suhdekartasta Gamman & al. (1995) mukaisesti.

- *Tutkitaan samaan ryhmään kuuluvia suunnittelumalleja.* Suunnittelumallien hakemistoissa suunnittelumallit on yleensä luokiteltu ryhmiin, kuten luontimalleihin, rakennemalleihin ja käyttäytymismalleihin (Gamma & al., 1995). Voidaan siis vertailla ryhmän suunnittelumallien eroja ja samankaltaisuuksia keskenään suhteessa ratkaistavaan ongelmaan.

- *Tarkastellaan uudelleensuunnitteluun johtavia syitä.* Tarkastelemalla uudelleensuunnittelua aiheuttavia syitä, kuten laite- ja alustariippuvuus, luokkien välinen tiukka sidonta ja heikko muunneltavuus. Tutkimalla syihin pureutuvia suunnittelumalleja, voidaan selvittää ongelman ratkaiseva suunnittelumalli.

- *Mietitään mitkä asiat halutaan pitää muunneltavina omassa ratkaisussa.* Valittaessa suunnittelumallia keskitytään seikkoihin/kohteisiin, joiden halutaan olevan muunneltavia ilman tarvetta uudelleensuunnitteluun myöhemmin. Jotkin suunnittelumallit nimittäin mahdollistavat tiettyjen muunneltavien asioiden kapseloinnit siten, etteivät muutokset aiheuta uudelleensuunnittelutarvetta.

Tietyn suunnittelumallin valintaan voidaan päätyä jo sovelluskehitysprosessin suunnitteluvaiheessa tai sitten vasta koodin refaktoroinnissa (kuva 1). Suunnitteluvaiheessa edellä esiteltyjen kohtien soveltaminen ja mallien valinta perustuu yleensä edeltävään toiminnalliseen määrittelyyn. Tällöin valintaa ohjaa lähinnä pyrkimys jo valmiiksi uudelleenkäytettävään ja laadukkaaseen sovelluksen suunnitteluun. Refaktoroinnissa taas tarve tietyn suunnittelumallin käyttöön tulee jo olemassa olevan ohjelmakoodin laadun kehittämisestä. Tällöin valinnan syynä voi olla vaikkapa pyrkimys vähentää saman koodin toistoa siirtämällä se tilanteeseen parhaan ratkaisun tarjoavan suunnittelumallin osoittamaan luokkaan (esimerkiksi Rekursiokooste-malli ja *Adapteri*-malli (Adapter)) (Kerievsky, 2005).

2.4 Suunnittelumallin käyttö

Suunnittelumalli valitaan tiettyä käyttöä varten, ratkaisemaan suunnitteluongelma tai esittämään aiempi ratkaisu aiempaa paremmin (refaktorointi). Riippumatta sovelluskehitysprosessin vaiheesta pitää suunnittelumallien soveltamisessa ottaa huomioon tietyt asiat. Ohessa on vaiheittainen Gamman & al. (1995) esittämä lähestymistapa, jossa otetaan huomioon kohdan 2.1 esittelemä suunnittelumallille tarkoitettu kuvausrunko:

1. *On varmistettava, että suunnittelumalli sopii käsillä olevaan ongelmaan.* Tähän liittyvät erityisesti suunnittelumallin kuvauksen kohdat Soveltuvuus ja Seuraukset.

2. *On varmistettava, että ymmärretään suunnittelumalliin sisältyvät luokat ja oliot ja niiden väliset suhteet.* Kohdat Rakenne, Osallistujat ja Yhteistyösuhteet liittyvät tähän.

3. *On osattava toteuttaa suunnittelumalli.* Apua saa katsomalla Mallikoodia-kohdassa esitettyjä konkreettisia esimerkkejä suunnittelumallista ohjelmakoodina.

4. *Valitaan suunnittelumalliin osallistuville luokille nimet, jotka liittyvät sovellusalueeseen.* Suunnittelumalleissa käytetyt luokkien nimet ovat yleensä liian abstrakteja suoraan omassa sovelluksessa käytettäväksi. Siitä huolimatta on hyödyllistä sisällyttää luokan nimi sovelluksessa esiintyvään nimeen. Se auttaa suunnittelumallia erottumaan paremmin toteutuksessa. Jos esimerkiksi rivitysalgoritmissa käytetään *Strategia*-mallia (Strategy), voi luokan nimeksi antaa SimpleLayoutStrategy tai TeXLayoutStrategy.

5. *Tehdään luokkien määrittely.* Määritellään luokkien rajapinnat, niiden perimissuhteet ja ilmentymämuuttujat, jotka edustavat tietoja ja olioviittauksia. Tunnistetaan sovelluksesta ne luokat, joihin suunnittelumalli vaikuttaa, ja muutetaan niitä vastaavasti.

6. *Annataan suunnittelumallin operaatioille sovellusaluekohtaiset nimet.* Nimet riippuvat yleensä sovellusalueesta. Käytetään operaatioon liittyvien vastuiden ja yhteistyösuhteiden nimiä oppaana. Pidetään kiinni yhtenäisestä nimeämiskäytännöstä.

7. *Toteutetaan operaatiot suunnittelumallin vastuiden ja yhteistyösuhteiden aikaansaamiseksi.* Toteutus-kohdassa on annettu vinkkejä, jotka auttavat toteutuksessa. Myös Mallikoodia-kohdan esimerkeistä on apua.

Lisäksi suunnittelumallien käyttöön on olemassa erityisiä sovelluskohtaisia ohjeistuk-

sia, jotka kuvaavat eri suunnittelumallien käyttöä juuri tietyn ongelman ratkaisemiseksi. Tällaisia ovat esimerkiksi kohdassa 3.2 tarkasteltavat mallikielet, jotka tarjoavat suunnittelumallit ja sen lisäksi näiden suunnittelumallien väliset suhteet ja tiedon siitä, kuinka suunnittelumalleja tulisi käytännössä soveltaa. Suunnittelumalleja refaktoroinnissa käyttävistä ohjeistuksista eräs esimerkki on Kerievskyn (2005) Refactoring to Patterns.

Suunnittelumallien käytön etuna voidaan pitää myös sitä, että ne noudattavat olio-ohjelmoinnin yleisiä ”hyviä tapoja” (Gamma & al., 1995).

Sen lisäksi, että tiedetään kuinka suunnittelumalleja käytetään, on tiedettävä miten niitä *ei* pidä käyttää. Suunnittelumalleja ei pidä käyttää kritiikittömästi, sillä usein joustavuus ja muunneltavuus lisäävät epäsuoraa ohjausta, mikä voi puolestaan monimutkaistaa suunnitteluratkaisua ja/tai vähentää suoritustehoa (Kerievsky, 2005). Suunnittelumallia tulee käyttää vain silloin, kun sen tarjoamaa joustavuutta todella tarvitaan. Suunnittelumallin kuvaukseen liittyvä Seuraukset-kohta on hyödyllinen arvioitaessa sen etuja ja haittoja (Gamma & al., 1995).

Suunnittelumallien käytön kehittyneempi aste on suunnittelumallien systemaattinen yhdistäminen. Parhaimmillaan tällöin on käytössä selkeä prosessi, joka tarjoaa vaiheet ja askeleet suunnittelun luomiseksi. Tarkemmin erilaisiin yhdistämistekniikoihin perehdytään luvussa 3. Myös systemaattisessa suunnittelumallien yhdistämisessä kannattaa hyödyntää edellä esitettyjä suunnittelumallien käytön periaatteita.

3 Suunnittelumallien yhdistäminen

Edellisessä luvussa todettiin suunnittelumallien antavan jo aiemmin hyväksi havaittuja ratkaisuja oliopohjaisen ohjelmistosuunnittelun ongelmiin ja parantavan ohjelmistojen uudelleenkäytettävyyttä. Yksittäinen suunnittelumalli ei kuitenkaan tee ihmeitä ohjelmistojen kokojen kasvaessa, vaan suunnittelussa tarvitaan monia, usein toisiinsa jotenkin yhteydessä olevia suunnittelumalleja. Alexander & al. (1977) sanovatkin: ”Yksikään suunnittelumalli ei ole eristetty kokonaisuus. Jokainen suunnittelumalli voi olla olemassa vain siihen rajaan saakka kuin muut suunnittelumallit sitä tukevat: sitä suuremmat suunnittelumallit, joihin se on sulautettu, sen kanssa samansuuruiset ympäröivät ja pienemmät, siihen sulautetut suunnittelumallit.”

Ohjelmistoa suunniteltaessa suunnittelumalleja käytetään tarpeen mukaan. Haluttu suunnittelumalli voidaan valita joko oman harkinnan tuloksena tai jonkin valmiin ohjeistuksen mukaisesti ja liittää se sitten muuhun suunnitteluun. Kehittyneessä suunnittelumallien hyödyntämisessä halutaan myös sovittaa suunnittelumalleja yhteen kuvan 1 esittämän periaatteen mukaisesti. Joskus malleja voidaan jopa yhdistää rakennuspalikoiden tapaan, muodostaen yhdistettyjä suunnittelumalleja, olio-ohjelmoinnin sovelluksia tai ohjelmistokehyksiä.

Tämän luvun alussa käydään läpi mihin suunnittelumallien systemaattista yhdistämistä tarvitaan ja mitä se tarkoittaa esittämällä lyhyt yhdistämistapojen luokittelu. Sen jälkeen esitellään luokittelun mukaisesti eri tapoja suunnittelumallien yhdistämiseksi. Luvun lopussa tehdään vielä lyhyt yhteenveto UML:n määrittelemästä yhdistelmämallista ja tarkastellaan muiden yhdistämistekniikoiden suhdetta siihen.

3.1 Mallien yhdistämisen luokittelua

Sovellusten suunnittelu käyttämällä suunnittelumalleja ei ole itsestään selvä prosessi. Suunnittelijan pitää tietää mitä suunnittelumalleja halutun toiminnallisuuden aikaansaamiseksi tarvitaan, kuinka suunnittelumallit yhdistetään muuhun suunnitteluun ja mitä hyötyjä ja haittoja tehtävistä valinnoista seuraa (Gamma & al., 1995).

Kun rakennettavien ohjelmistojen koot kasvavat, monimutkaistuu niiden suunnittelukin. Suunnittelun hallintaan tarvitaan selkeä prosessi, jossa on selkeät vaiheet ja askeleet suunnittelun luomiseksi (Yacoub & Ammar, 2004). Lisäksi tarvitaan suun-

nittelun ongelmiin ratkaisut ja työkalut ratkaisujen yhdistämiseksi valmiiksi suunnitteluksi. Suunnittelumallit ovat näitä ratkaisuja ja niitä on kehitetty yleisimpiin ongelmatilanteisiin. Koska suunnittelumallit ovat hyväksi havaittuja ja testattuja ratkaisuja, yritetään mahdollisimman moni ongelma ratkaista käyttämällä siihen käyvää suunnittelumallia. Suunnittelumallien käytön lisääntyessä on tullut tarve kehittää tekniikoita, joilla ne yhdistetään muuhun suunnitteluun ja täten myös muihin suunnittelumalleihin. Näitä yhdistämistekniikoita tarvitaan myös muissa ohjelmiston kehityksen vaiheissa, kuten varsinaisessa toteutusvaiheessa, kun aiempaa suunnittelua parannetaan ja ylläpitovaiheessa, jossa halutaan lisätä ohjelmistoon uutta toiminnallisuutta (Yacoub & Ammar, 2004).

Suunnittelumalleja voidaan yhdistää luokka- tai oliotasolla. *Luokkamallit* (class models) tuovat esille suunnittelumallin toteutus- ja ylläpitonäkökulmat, kun taas oliomallit paljastavat ajonaikaiset, käyttäytymis- ja rooliaspektit. Useat tutkijat ja käytännön soveltajat kuten Reenskaug (1996) ja Riehle (1997) käyttävät suunnittelumallien yhdistämiseen rooli- ja vastuumallinnusta. Vähemmän huomiota on kiinnitetty suunnittelumallien yhdistämiseen luokkina (Yacoub & Ammar, 2004).

Tässä luvussa tarkastellaan lyhyesti useita tekniikoita suunnittelumallien yhdistämiseksi. Ensimmäisenä kohdassa 3.2 esitellään mallikielet. Muut tekniikat voidaan luokitella Yacoubin ja Ammarin (2004) mukaan kahteen ryhmään:

- käytökselliset yhdistämistekniikat
- rakenteelliset yhdistämistekniikat

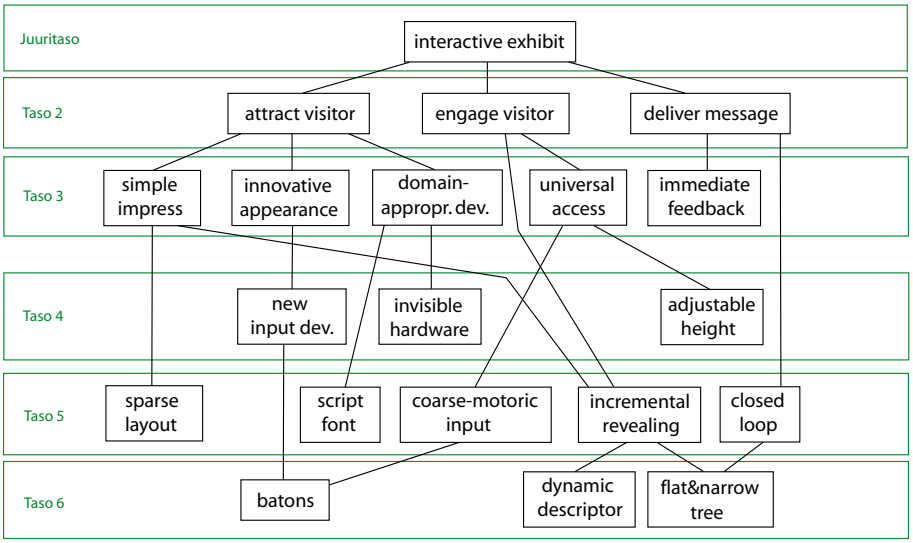
Käytökselliset tekniikat perustuvat olioiden vuorovaikutusmääritelmiin, eli siihen kuinka oliot ovat vuorovaikutuksessa keskenään, näyttäen kuinka suunnittelumallien ilmentymät voidaan yhdistää. Rakenteelliset tekniikat perustuvat muuttumattomiin arkkitehtuurisiin määritelmiin, eli siihen kuinka yhdistettyjen suunnittelumallien ilmentymät esitetään luokkakaavioita käyttäen (Yacoub & Ammar, 2004).

3.2 Mallikielet

Alexanderin & al. (1977) rakentamiseen tarkoittamat mallikielet ovat olleet esikuvana ohjelmistotuotannon mallikielille. Mallikielet ovat eräänlaisia kehyksiä. Ne sisältävät

tietyn sovellusalueen ongelmiin ratkaisuja tarjoavat suunnittelumallit ja sen lisäksi näiden mallien väliset suhteet ja tiedon siitä, kuinka suunnittelumalleja tulisi käytännössä soveltaa. Mallikielet ohjaavat kehitysprosessia (Fincher, 2000).

Mallikieli on sovellusaluekohtainen ja joskus jopa toteutuskielikohtainen. Mallikielen esittämät suunnittelumallit esittävät täten yleisiä parhaita ratkaisuja juuri tuon sovellusalueen kehittämisongelmiin. Käytön helpottamiseksi toisiinsa liittyvät suunnittelumallit on järjestetty hierarkkisesti. Salingaros (2000) mainitsee luontaiset rakenteet ja suhteet. Hänen mielestään mallikielet perustuvat suunnittelumallien välisiin yhteyksiin ja yhteydet määritteleviin sääntöihin yhdistäen suunnittelumallit järjestäytyneeksi kehykseksi. Näiden yhteyksien avulla voidaan yksittäisiä suunnittelumalleja yhdistämällä saada laajempaa toiminnallisuutta tarjoavia suunnittelumalleja (Todd & al., 2004). Suunnittelumallien hierarkia voi olla esimerkiksi kuvan 2 puumallin kaltainen. Siinä ylin suunnittelumalli on juurena, joka voi kuvata esimerkiksi sovellusalueen. Alemmat mallit eri tasoilla kohdistuvat aina ylempää yksityiskohtaisempiin ongelmiin. Ylemmät tasot tarjoavat kontekstin, johon alemman tason suunnittelumalleja käytetään.



Kuva 2: Mallikielen puumainen mallihierarkia (Borchers, 2000).

Varsinaista suunnittelumallien yhdistämistekniikkaa mallikielet eivät tarjoa, vaan tärkeintä on suunnittelumallien kuvauksen lisäksi niiden toteutusesimerkkien esittäminen ja käytön opastaminen. Vaikka yhdistämisen toteutus jääkin suunnittelumallien käyttäjän vastuulle, antavat mallikielet ainakin neuvoja mitä suunnittelumalleja yhdistää ja mihin sillä pyritään.

Kaikki suunnittelumallikokoelmat eivät ole mallikieliä, vaan juuri edellä esitellyt ominaisuudet tekevät pelkästä suunnittelumallien kokoelmasta mallikielen (Salingaros, 2000). Parhaimmillaan mallikieltä voidaan käyttää ohjaamaan koko sovelluksen kehitysprosessia. Tämän vuoksi suurin osa suunnittelumallikokoelmista, kuten esimerkiksi Gamman & al., (1995) esittelemät suunnittelumallit, eivät ole mallikieliä. Ne eivät kerro varsinaista suunnittelumallien käyttöjärjestystä eivätkä yksinään riitä valmiin sovelluksen rakentamiseen.

Ohjelmistotuotannon alalla tuskin päästään samankaltaiseen yleiseen ja yhtä täydelliseen mallikieleen, kuten Alexander & al. (1977) rakennusalalla. Siihen vaadittaisiin suunnittelumallien lisäksi myös muita suunnittelumalleja ja suurempia kokonaisuuksia, kuten kehyksiä (Gamma & al., 1995). Tutkimus ja kehitys mallikielten parissa jatkuu kuitenkin edelleen sovellusaluekohtaisempana ja laatuun keskittyen, kuten käyttöliittymien mallikielten laatututkimuksen (Todd & al., 2004) tapauksessa.

3.3 Käytökselliset yhdistämistekniikat

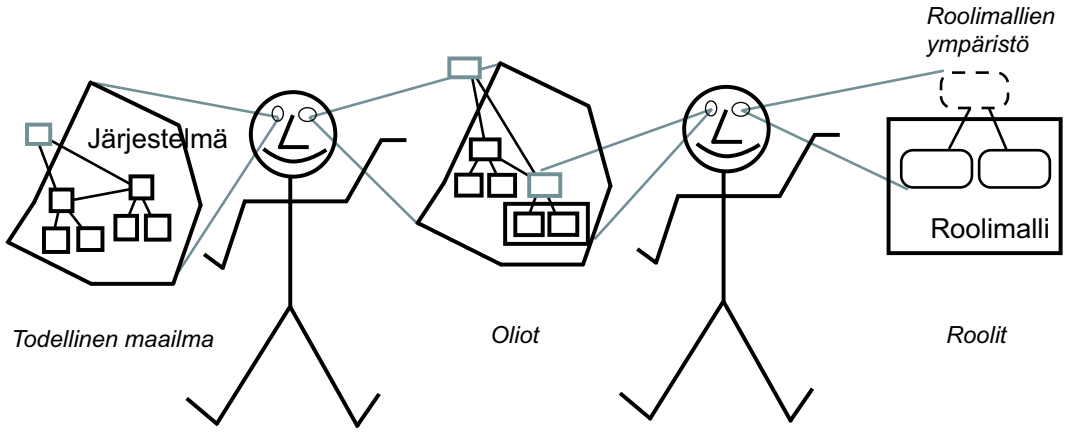
Käytökselliset yhdistämistekniikat pitävät olioita elementteinä, joilla on useita rooleja siten, että kukin rooli on osa erillistä mallia. Käytökselliset yhdistämistekniikat tunnetaan oliokirjallisuudessa myös vuorovaikutussuuntautuneena tai vastuulähtöisenä yhdistämisenä (Wirfs-Brock & Wilkerson, 1989).

Tässä kohdassa käydään läpi käytöksellisiä yhdistämistekniikoita yhteenvetona alan tutkimustyöstä ja analysoidaan niiden etuja ja haittoja. Yksittäisten suunnittelumallien käyttäytymisen määrittelyn formalisointi on tärkeää käyttäytymisen merkitysopin selventämiseksi ja määrittelyn hyödyntämiseksi suunnittelumallien yhdistämisessä. Ensimmäisenä tarkastellaan Reenskaugin (1996) esittämää tapaa roolien mallintamiseen ja olio-ohjelmoinnin rooli-analyysimetodia käyttävää synteesiä. Sitten kuvaillaan Riehlen (1997) työ, mikä soveltaa Reenskaugin ehdottamia roolimallien käsitteitä suunnittelumallien yhdistämiseen. Lopuksi tehdään lyhyt yhteenveto Boschin (1998b) superimpositio-tavasta, mikä käyttää suunnittelumalleja ja kehyksiä arkkitehtuurisina palasina ja yhdistelee rooleja ja komponentteja tuottamaan sovelluksia.

3.3.1 Oliosuuntautunut roolianalyysi ja ohjelmistosynteesi

Reenskaug (1996) on kehittänyt oliosuuntautuneen *roolianalyysin* ja *ohjelmistosynteesin*, *OOram-metodin* (Object Oriented Role Analysis Method). OOram-metodi osoittaa erityisesti kaksi kehitysprosessia: *mallinnusprosessin*, jonka aikana roolimallit luodaan ja *synteesiprosessin*, jonka aikana roolimallit yhdistetään. Seuraavaksi tarkastellaan näitä kahta prosessia.

OOram-mallinnusprosessissa suunnittelija aloittaa analysoimalla todellisen maailman järjestelmää ja tunnistamalla olioita ja niiden välisiä vuorovaikutuksia. Jokaisen olion esittämä rooli tunnistetaan olioiden välisen vuorovaikutuksen perusteella. Sitten luodaan *roolimalli*, joka kuvaa tietyn vuorovaikutuksen aiheen, olioiden välisen yhteyden ja olioiden viestinvaihdon. Kuva 3 kuvaa mallinnusprosessia. Yksittäinen olio voi esittää monia rooleja ja jokainen olion esittämä rooli kuuluu johonkin yhteistoimintaan tai roolimalliin. Esimerkiksi yrityksen työntekijä voi esittää useita rooleja ollen matkustaja matkakertomusmallissa tai projektipäällikkö palkkalistamallissa. Jokainen roolimalli kuvaa rajoitettua näkymää ongelmasta. Roolimallinnusta tehtäessä päädytään täten useisiin roolimalleihin (Reenskaug, 1996).

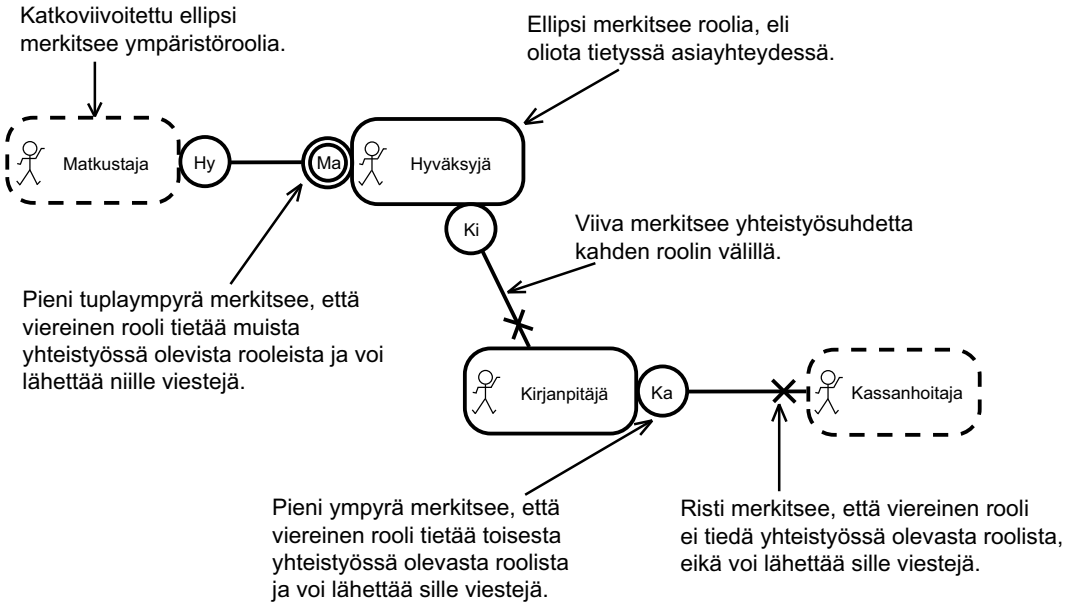


Kuva 3: Roolin mallinnusprosessi (Reenskaug, 1996).

OOram:issa roolit ovat luokkien ja olioiden merkinnän abstraktio. Kuten luokka, rooli on kuvaus joukosta olioita. Siinä missä luokka kuvaa joukon olioita yhteisellä käyttäytymisellä ja ominaisuuksilla, kuvaa rooli useiden käyttäytymisten ja ominaisuuksien yhteisvaikutusta. Lisäksi rooleilla on samankaltaisia ominaisuuksia kuin olioilla. Roolimallissa voidaan määritellä roolien välillä vaihdettavia viestejä ja kuinka roolit reagoivat näihin viesteihin (Reenskaug, 1996).

Syy roolimallien käyttöön sekä mallin staattisten että dynaamisten puolien löytämiseksi on se, että mallinnuksen aikana halutaan nähdä niin mallielementtien kyvyt kuin myös mallielementtien käyttäytyminen ja niiden välinen viestinvaihto (Reenskaug, 1996).

Roolimallikaavioilla on mallinnusta varten erilaisia rakenteita: *järjestelmärooli* (system role) ja sen attribuutit, *ympäristörooli* (environment role), *viestipolku* (message path) yhteistyössä olevien roolien välillä, *portteja* (ports) joiden kautta viestit voidaan jakaa muille rooleille, keinoja yksiselitteisesti mallintaa roolin tiedot toisesta yhteistyössä olevasta roolista ja keinot mallintaa suuret määrät rooleja ja portteja (Reenskaug, 1996). Esimerkki roolimallinäkymästä on kuvassa 4, jossa esillä on osa matkakustannusjärjestelmää, missä useat roolit toimivat yhteistyössä hoitaen matkakustannusten raportoinnin ja maksun.



Kuva 4: Esimerkki roolimallinäkymästä matkakustannusjärjestelmän tapauksessa (Reenskaug, 1996).

Kuva 4 kertoo, että matkakustannusraportin käsittelyyn osallistuvat esittävät matkustajan, hyväksyjän, kirjanpitäjän ja kassanhoitajan rooleja. Matkustaja ja hyväksyjä tietävät toisistaan ja vaihtavat viestejä. Kassanhoitaja ei tiedä kirjanpitäjästä, eikä voi lähettää tälle viestejä, samoin kirjanpitäjä ei tiedä hyväksyjästä. Kaksi roolia on merkattu kuuluvaksi ympäristöön (rooleja, joista tarvitsee kuvata vain tarpeellinen): matkustaja, koska se käynnistää mallin toiminnan lähettämällä ensimmäisen, pyytämättömän viestin ja kassanhoitaja, koska se on valmiin kuluraportin viimeinen

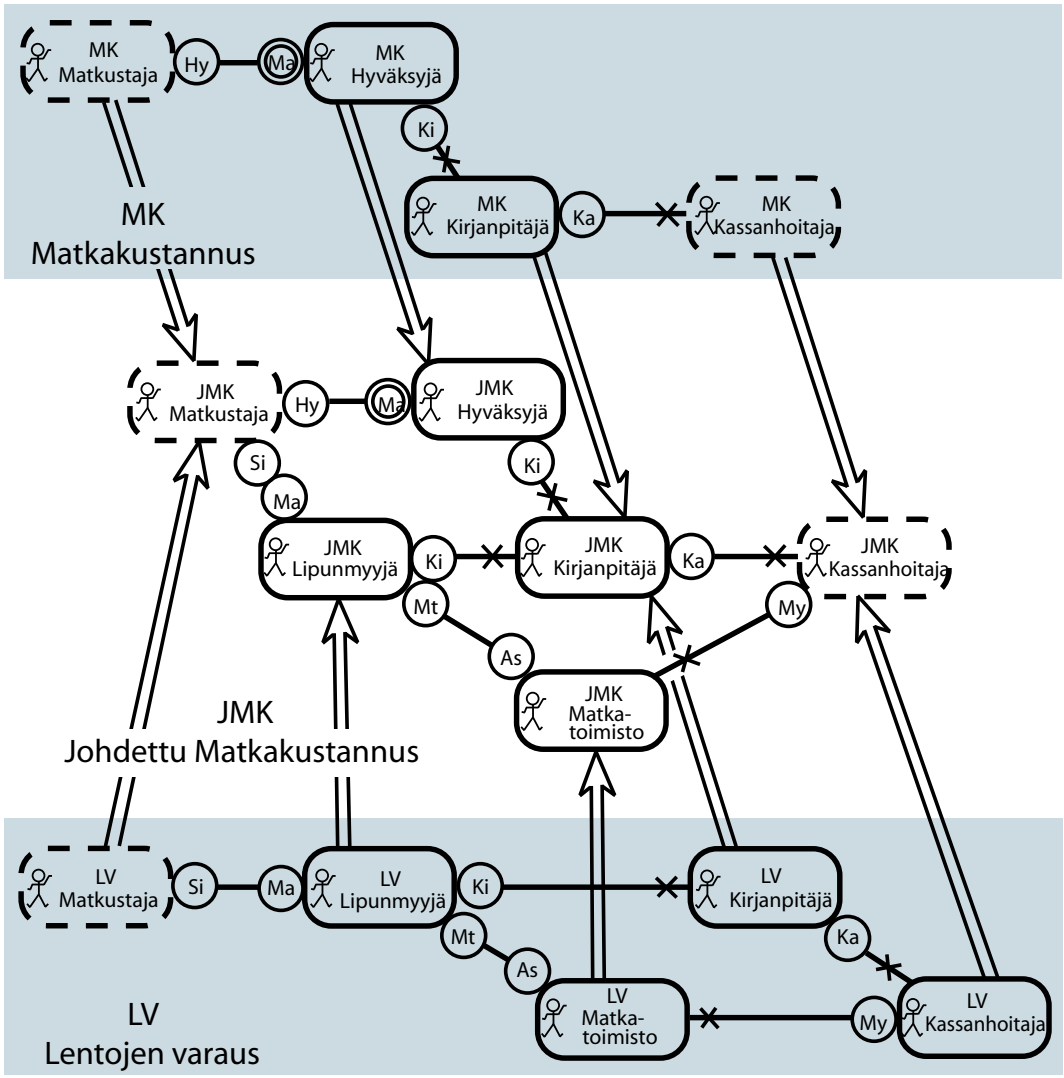
vastaanottaja. Kaksi muuta ovat järjestelmärooleja (osat järjestelmästä, jotka täytyy kuvata täydellisesti). Matkustajaroolista lähtevään viivaan yhdistettynä on portti, pyöreä ympyrä, joka esittää viestejä, joita matkustajarooli voi lähettää hyväksyjälle. Toiset pienet ympyrät tulkitaan vastaavasti.

Toinen OOram-metodin prosessi on synteesiprosessi. Synteesiprosessi integroi eri mallinäkymät yhdeksi mallinäkymäksi. Perusroolimallien synteesi tuottaa yhtenäisen (johdetun) roolimallin. Johdettu roolimalli koostuu uusista rooleista ja niiden yhteistoiminnasta. Yhdistelmärooli johdetussa roolimallissa esittää siten erilaisia yksinkertaisia rooleja muista perusroolimalleista. Kaikki perusroolimallien roolit, attribuutit ja yhteistyöt tulee sovittaa lopulliseen malliin. Reenskaugin mukaan synteesiprosessin esittävät näkymät ovat monimutkaisia jopa yksinkertaisten roolimallien tapauksessa. Kuva 5 esittää kahden roolimallin synteesin (kuvan keskiosa): Toinen on matkakustannus-roolimalli (kuvan yläosa) ja toinen lentojen varaus -roolimalli (kuvan alaosa).

OOram löytää roolimalleja, jotka sopivat mallintamaan suunnittelumalleja, joissa mallin ratkaisuun kuuluu vuorovaikutuksessa olevien olioiden yhteistyötä. Perusroolimalli voidaan luoda esittämään suunnittelumallin tarjoamaa abstraktiota. Tätä malliroolimallia (pattern role model) voidaan käyttää synteesissä, jolloin suunnittelumalleja yhdistetään niitä mallintavia roolimalleja käyttäen (Reenskaug, 1996).

Reenskaugin suunnittelumallien mallintamiseen käyttämät roolimallit eroavat tyypillisistä oliorakenteista, joita käytetään suunnittelumallien kuvaamiseen. Reenskaugin käyttämä mallinnus käyttää yhtä mallia roolin sekä staattisten että dynaamisten puolten esittämiseen. UML välttää tällaista yhdistämistä. Staattisia malleja käytetään yhdistämis- ja kytkeytymistarkoituksiin, kun taas dynaamisia malleja käytetään käyttäytymisen analysointi- ja vuorovaikutuksen esittämistarkoituksiin. Siksi niille on parempi pitää aina erilliset mallit; niiden suhteet ja johdonmukaisuuden kuitenkin säilyttäen (Yacoub & Ammar, 2004).

Reenskaug ei ole jatkanut tutkimustaan roolimallien käytöstä suunnittelumallien yhdistämiseksi tämän pidemmälle. Roolimallit ovat olleet kuitenkin osa suunnittelumallien käytöksellisen yhdistämisen tutkimusta, kuten Riehlen (1997) roolikaavioissa. Reenskaugin roolimallit ovat vaikuttaneet myös laajemmin, sillä UML:n yhteistyönotaatio pohjautuu osittain OOram-metodiin (Reenskaug, 2000). Reenskaug onkin osallistunut UML:n kehitykseen ja siirtynyt sittemmin itsekin tutkimuksissaan UML:n käyttöön.



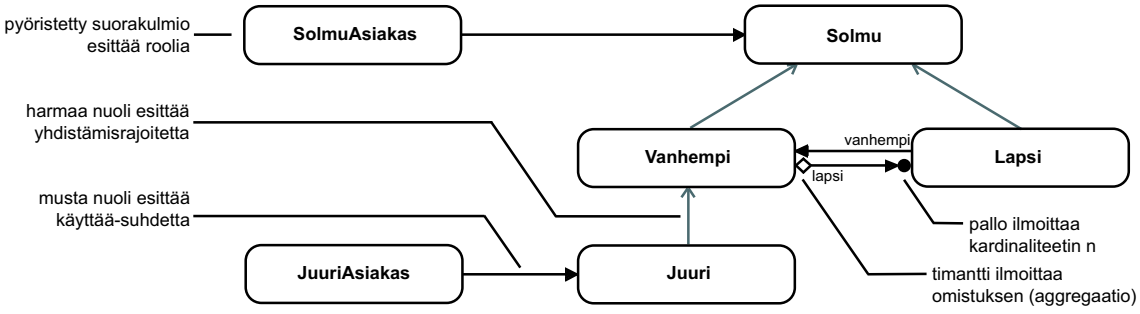
Kuva 5: Roolimallien synteesi (Reenskaug, 1996).

3.3.2 Suunnittelumallien yhdistäminen käyttäen rooleja

Riehle (1997) käyttää roolimallien (Reenskaug, 1996) pohjalta kehittämiään roolikaavioita suunnittelumallien yhdistämiseen. *Roolikaaviot* ovat laajennoksia roolimalleihin ja määrittelevät joukon yhdistämisrajoitteita:

- Olio, joka esittää jotain roolia A, esittää aina jotain toista roolia B samassa yhteistyössä. Rooli A implikoi siis roolin B.
- Roolia A esittävä olio ei koskaan esitä roolia B.
- Rooli A ei välitä roolista B. Ne sekoittuvat mielivaltaisesti, eikä niistä voi sanoa mitään.

Esimerkki roolikaaviosta Rekursiokoosteen (kuvattu tarkemmin liitteessä 1) tapauksessa (juuriolioiden merkinnällä laajennettuna) on esitetty kuvassa 6. Siinä rajoitetta ”rooli A implikoi roolin B” esittää parit (Vanhempi, Solmu) ja (Lapsi, Solmu). Pari (Lapsi, Juuri) on esimerkki ”rooli A ei koskaan esitä roolia B” -rajoitteesta. Roolikaaviot yhdistetään olioihin sovelluksen oliokaavion kehittämiseksi (Riehle, 1997).



Kuva 6: Roolikaavio Rekursiokoosteen tapauksessa (Riehle, 1997).

Riehle tutki tiettyä yhdistämisongelmaa, joka liittyy *yhdistelmäsuunnittelumalleihin* (composite design patterns), kuten hän niitä kutsui. Yhdistelmäsuunnittelumallit eroavat GoF:n (Gamma & al., 1995) Rekursiokooste-mallista (Composite pattern). Rekursiokooste on suunnitteluratkaisu hierarkian esittämiseksi oliosovelluksissa, kun taas yhdistelmäsuunnittelumalli on yhdistelmä suunnittelumalleja, joiden integrointi tuottaa alkuperäisiä laajemman kokonaisuuden. Täten yhdistelmäsuunnittelumalli on yksinkertaisesti joukko suunnittelumalleja integroituna ja yhdistettynä toisiinsa niin,

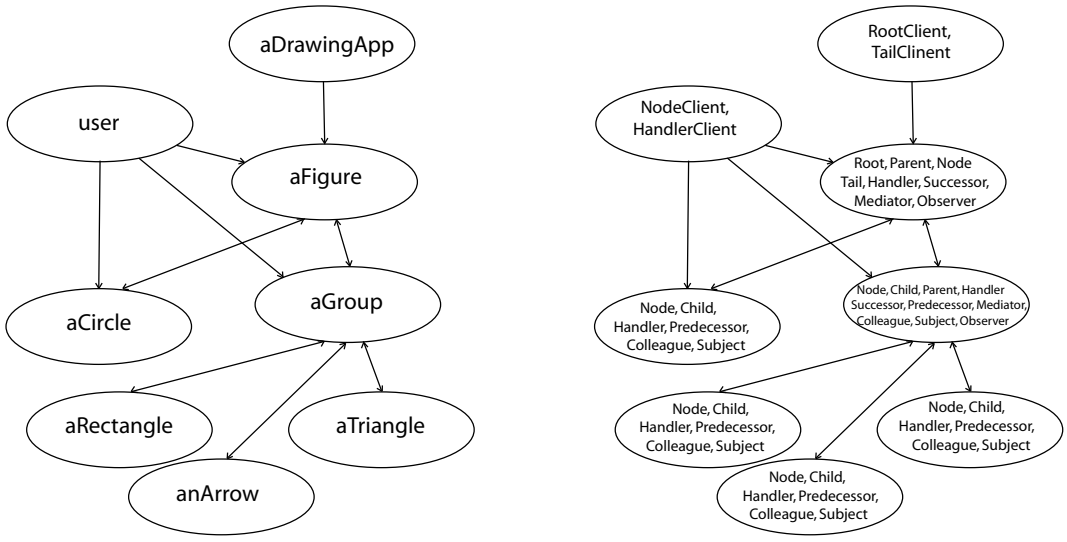
että integraatio täyttää suunnittelumallina olon ehdon, eli ratkaisee jonkin usein esiintyvän suunnitteluongelman (Riehle, 1997).

Yhdistelmämallien luomiseksi Riehle (1997) seuraa yksinkertaista proseduuria:

1. Mallinnetaan kaikki olemassaolevat suunnittelumallit käyttäen roolikaavioita.

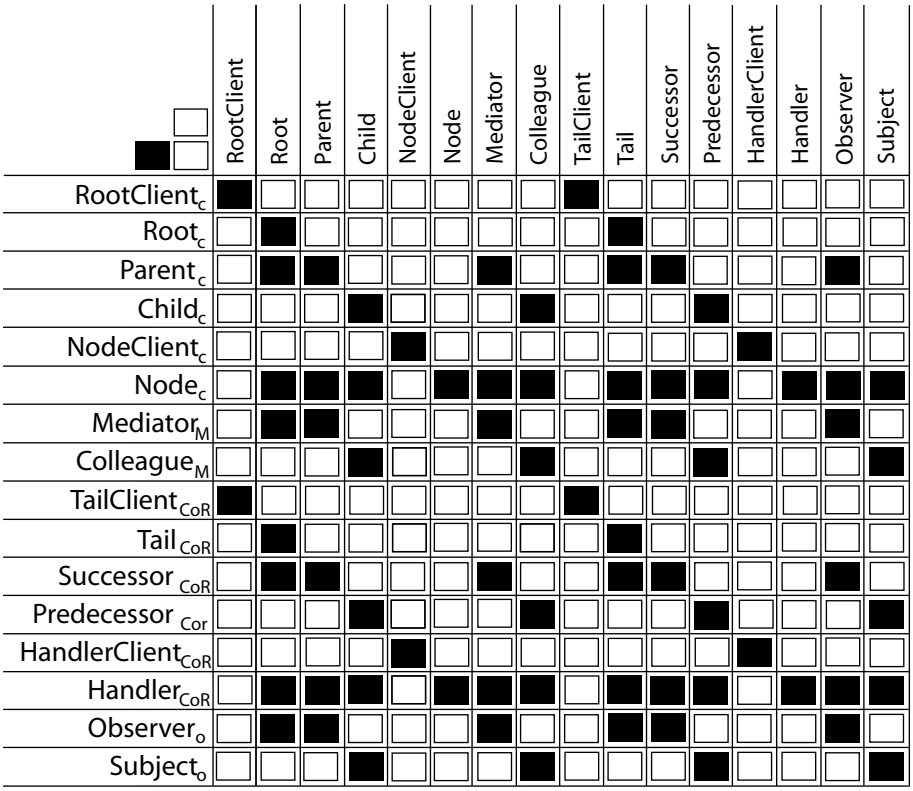
2. Suunnittelu käyttää prototyypin omaista mallisovellusta johtamaan yhdistelmäsuunnittelumallin. Prototyyppisovellus on konkreettinen sovellus, joka esitetään yleensä jonkinlaisella olioiden hierarkian kuvaavalla kaaviolla, oliokaaviolla (kuva 7(a)).

3. Käyttäen kuvausta sovelluksen olioiden välisestä yhteistyöstä, suunnittelija alkaa valita suunnittelumalleja, joita voidaan käyttää sovelluksessa. Hän käyttää jokaisen suunnittelumallin roolikaaviota ja määrää siitä roolit sovelluksen oliokaavion olioille. Tämän prosessin lopussa jokaiselle oliolle on määrätty useita rooleja useista suunnittelumalleista (kuva 7(b)).



Kuva 7: (a) Prototyyppisovelluksen oliokaavio ja (b) olioille määrätyt roolit (Riehle, 1997).

4. Käyttäen edellisessä vaiheessa luotua sovelluksen oliokaaviota, suunnittelija luo roolien *suhdematriisin* (kuva 8). Roolien suhdematriisia käytetään analysoitaessa roolien keskinäistä läheisyyttä yhdistämisrajoitteiden suhteen. Tämän analyysin tarkoitus on löytää suunnittelumallien vuorovaikutuksen yhteisvaikutus ja määritellä rooleiltaan vastaavat joukot, eli matriisin yhtenevät sarakkeet, jotka muodostavat *yhdistelmäroolit* (kuva 9). Roolien suhdematriisi sievennetään lopulliseen muotoonsa, joka sisältää enää jäljelle jääneet yhdistelmäroolit.



Kuva 8: Prototyyppisovelluksen suhdematriisi (Riehle, 1997).

```
DirectorClient_B = {  RootClient_C , TailClient_CoR  }
Director_B       = {  Root_C , Tail_CoR  }
Manager_B        = {  Parent_C , Mediator_M , Successor_CoR, Observer_O  }
Subordinate_B    = {  Child_C , Colleague_M , Predecessor_CoR, Subject_O  }
ClerkClient_B    = {  NodeClient_C , HandlerClient_CoR  }
Clerk_B          = {  Node_C , Handler_CoR  }
```

Kuva 9: Yhdistelmäroolit (Riehle, 1997).

5. Lopullista roolien suhdematriisia käytetään luomaan roolikaavio yhdistelmäsuunnittelumallia varten.

Yhdistelmäsuunnittelumallien mallintamisessa roolikaavioiden yhdistelminä on Riehle (1997) tunnistanut joitakin haittapuolia:

- Roolikaavioiden valitseminen pääasialliseksi suunnittelumallin kuvauskeinoksi jättää huomiotta luokkaperintään perustuvat suunnittelumallit, joita varten pitää kehittää kehittyneempiä tekniikoita.
- Jotkin suunnittelumallit ovat monimutkaisempia roolikaavioiden esittäminä. Erityisesti rekursiivisia rakenteita sisältävät suunnittelumallit, kuten Rekursiokooste tai *Vastuuketju* (Chain of Responsibility), ovat tällaisia, koska rajaehtojen täyttäminen lisää roolien ja yhdistämisrajoitteiden määrää.
- Roolikaavioiden yhdistämisprosessi on jälkikäteen tapahtuvaa toteuttamista. Luova suunnittelumallin työstämisprosessi ei edistynyt suoraviivaisesti, kuten otetut askeleet antoivat ymmärtää.

Päähyöty roolikaavioiden käyttämisestä suunnittelumallien mallintamiseen on, että se antaa korkeamman abstraktiotason kuin luokkakaaviot (Riehle, 1997). Roolimalli abstrahoi suunnittelumallin pääidean, kun taas suunnittelumallin luokkakaavio antaa toteutuspohjan, jolle voi olla useita eri toteutuksia.

3.3.3 Arkkitehtuurifragmentit ja superimpositio

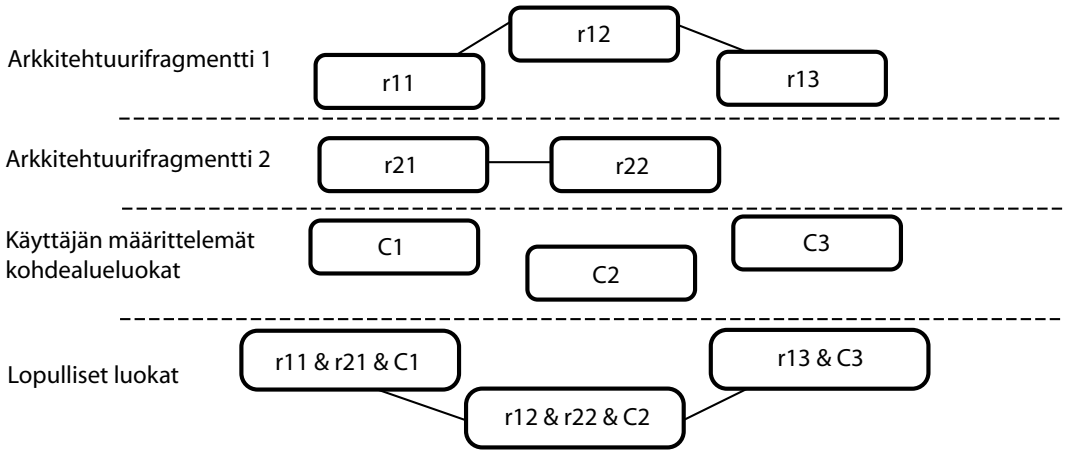
Arkkitehtuurifragmentit kehittyvät roolimallien käsitteitä pidemmälle. Bosch (1998b) käyttää suunnittelumalleja ja kehyksiä arkkitehtuurisina fragmentteina ja yhdistää niitä liittämällä yhteen niiden rooleja ja uudelleenkäytettäviä komponentteja. Bosch esittelee arkkitehtuuriset fragmentit keinona esittää suunnittelumalleja uudelleenkäytettävinä kokonaisuuksina. Kokonaisuus kuvaa suunnittelumalliin tai kehykseen liittyvän komponentin käyttäytymisen ja komponenttien välisen vuorovaikutuksen. Koska arkkitehtuurit ovat suunnittelutason kokonaisuuksia, parantaa saman käsitteistön käyttö toteutustasolla näiden kahden tason välistä jäljitettävyyttä ja käsitteellistä eheyttä.

Joukossa luokkia jokainen luokka esittää roolia tietyssä käyttäytymisessä. Arkkitehtuurifragmentti kuvaa kustakin noista luokista osat, jotka ovat ominaisia fragmentin

kuvaamalle käyttäytymiselle. Fragmentti koostuu täten joukosta rooleja. Lisäksi siihen kuuluu alustuskoodi, mikä suoritetaan kun fragmentti toteutetaan. Arkkitehtuurifragmentti määrittelee roolien osittaisen toteutuksen. Kun valitaan luokkaa esittämään roolia toteutetussa arkkitehtuurissa, luokan ja roolin käyttäytyminen yhdistetään. Jossain tapauksissa tämä vaatii luokan käyttäytymisen ylikirjoittamista tai laajentamista (extending) roolin käyttäytymisellä. Se saadaan tehtyä *superimpositiolla* (Bosch, 1998b). Arkkitehtuurisen fragmentin, roolin ja superimposition merkintä määritellään *kerrostetussa oliomallissa* (LayOM, layered object model) (Bosch, 1998a), jonka avulla voidaan mallintaa oliosuunnittelun erilaisia rakenteita, kuten suunnittelumalleja ja olioiden välisiä suhteita. LayOM:ssä olio koostuu muuttujista, metodeista, tiloista, kategorioista ja kerroksista. Kerrokset järjestetään luokiksi, jotka esittävät esimerkiksi vaikka juuri suunnittelumallia.

Tietyn tyyppiset roolin toiminnallisuudet täytyy yhdistää komponentin käyttäytymisen kanssa. Nämä roolin toiminnallisuudet voivat olla läpileikkaavia komponentin rakenteisten osien kanssa ja aiheuttavat tarvetta esimerkiksi useille metodeille. Suunnittelijan pitäisikin määrätä komponentille tietty käyttäytyminen tavalla, joka vaikuttaa suoraan komponentin koko toiminnallisuuteen. Bosch (1998b) määrittelee superimposition S komponentille O lisäävänä, O:n käyttäytymisen ylikirjoittavana käyttäytymisenä B. Sihman ja Katz (2003) havainnollistavat superimpositiota aspektikokoelmalla, joka sidotaan alkuperäiseen ohjelmaan. Sitomista kutsutaan yleisesti *punomiseksi* (weaving).

Suunnittelumallien yhdistämisen tapauksessa arkkitehtuurifragmenttitapaa käyttäen, jokainen suunnittelumalli esitetään fragmenttina. Yhdistäminen voi tapahtua myös sovellusaluekohtaisen suunnittelun luokkien kanssa. Kun suunnittelumalleille ja kohdealueluokille on annettu fragmenttimuotoinen esitys, käytetään superimpositiotekniikkaa yhdistämään nuo fragmentit yhteen kuvan 10 esittämällä tavalla (Yacoub & Ammar, 2004). Kuva näyttää kaksi fragmenttia, joista ensimmäisellä on kolme roolia ja toisella kaksi roolia. Siinä on myös kolme kohdealueluokkaa. Yhdistämisen tulos on joukko sovellusluokkia, jotka esittävät useita rooleja. Esimerkiksi sovellusluokka voi esittää kolmea roolia: fragmentin 1 roolia, fragmentin 2 roolia ja sovellusluokkaa (kuvassa vasen alakulma).



Kuva 10: Fragmenttien ja superimposition käyttö sovellusluokkien kehittämiseksi.

3.4 Rakenteelliset yhdistämistekniikat

Rakenteelliset yhdistämistekniikat rakentavat suunnittelun liimaamalla yhteen suunnittelumallin rakenteita, jotka mallinnetaan luokkakaavioina. Rakenteellinen yhdistäminen keskittyy enemmän suunnittelun varsinaiseen toteuttamiseen kuin abstrahointiin, käyttäen eri tyyppisiä malleja, kuten roolimalleja.

Seuraavissa alaluvuissa käydään läpi neljä rakenteellista yhdistämistekniikkaa. Ensimmäisenä Kellerin ja Sametingerin (2003) työ, joka näyttää kuinka suunnittelukomponentit kehittyvät sovellusalueen toteutukseksi. Tämä tapa takaa, että suunnittelun laatu siirtyy koodin laaduksi.

Toisena tarkasteltava tekniikka käyttää suunnittelumalleja komponenttipohjaisten kehysten kehittämiseen. Larsenin (1999) työssä kehitetään hierarkkinen arkkitehtuurinen suunnittelu, jossa komponentti voi toteuttaa monia suunnittelumalleja ja kehyksiä ja suunnittelumalli tai kehys voidaan toteuttaa monen komponentin yli. Suunnittelumallien yhdistämiseen käytetään rakenteellista tapaa ja suunnittelumallien rajapintoja mallinnetaan UML:n rajapintaluokilla. UML:n *komponenttikaavioita* käytetään sitten esittämään kehys yhdistelmänä fyysisiä komponentteja, jotka eivät kuitenkaan täysin vastaa suunnittelumalleja.

POAD, eli mallisuuntautunut analyysi ja suunnittelu (Yacoub & Ammar, 2004) esittää suunnittelumallien rakenteellisen yhdistämistavan suunnittelumallien luokkakaavioilla. POAD perustuu rajapinnan määrittelemiseen jokaiselle toteutetulle suunnittelumallille. Yhdistämiseen käytetään hierarkkisia loogisia malleja, jotka esittävät suunnitte-

lumalleja eri abstraktiotasoilla eri suunnittelutasoilla.

Lopuksi esitellään lyhyesti yhdistelmämallien ja subjektiperustaisen suunnittelun konseptit. Nämä konseptit näyttävät aspektiohjelmoinnin konseptiin pohjautuvan kehitysteknologian.

3.4.1 Ohjelmiston laatiminen suunnittelukomponentteja käyttäen

Keller ja Sametinger (2003) ehdottavat ohjelmiston suunnittelun tekemistä suunnittelukomponentteja, roolimalleja ja *roolirajoitteita* (role constraints) käyttäen.

Toteutuksen ja ylläpidon aikana suunnittelumallien erottaminen muusta sovelluksesta käy hankalaksi, ellei niitä erikseen dokumentoida. Suunnittelukomponentit tarjoavat järjestelmällisen tavan määritellä, toteuttaa ja jäljittää suunnittelumalleja konkretisoimalla suunnittelumallit (Keller& Schauer, 1998). Suunnittelukomponentit antavat ratkaisut Reenskaugin (1996) roolimalleja käyttäen esitettyihin suunnitteluongelmiin. Suunnittelukomponentteja on tarkoitus käyttää yhdistelemällä niitä toisiinsa ja muuhun suunnitteluun (Keller & Sametinger, 2003).

Suunnittelukomponentin rakenne mallinnetaan kolmella abstraktiotasolla käyttäen kullekin tasolle kuvan 11 kaltaista tekstimuotoista esitystä:

- *Komponentin kuvaus* -taso (description level) kertoo syyt valita tietty suunnittelu. Suunnittelumallia kuvatessaan se kertoo siitä oleellisimman tiedon, kuten nimen, luokittelun ja kaaviot. Esimerkki komponentin kuvauksesta on esitelty kuvassa 11.

```
description {
    component: Visitor Pattern
    author: Gamma, Helm, Johnson, Vlissides
    date: 1995
    version: v1.0
    short: Design component based on Visitor Pattern...
    intent: Represent an operation to be performed on...
    motivation: Consider a compiler that represents programs...
    applicability: An object structure contains many classes of...
    consequence: Visitor makes adding new operations easy. ...
    knownuse: The Smalltalk-80 compiler has a Visitor class ...
}
```

Kuva 11: Komponentin kuvaus Vierailija-mallista (Keller & Sametinger, 2003).

- *Komponentin roolimalli* -taso (role model level) kertoo kuinka toteuttaa suunnittelu toteutuskielestä riippumatta. Roolimallilla pyritään dokumentoimaan suunnittelukomponentin roolit yksiselitteisesti. Toteutusvaiheessa (instantiation) tämä tieto säilytetään asettamalla roolit luokkiin. Tällöin eri luokat esittävät suunnittelukomponentin rooleja noudattaen komponentin esittämää suunnittelua.

- *Komponentin toteutus* -taso (implementation level) kertoo kuinka sovittaa roolimalli ohjelmointikieleen ja tukee erilaisia toteutusalustoja. Toteutus voi perustua esimerkiksi luokkakirjastoon tai ohjelmistokehykseen.

Suunnittelukomponentin neljänneksi tasoksi luetaan sen varsinainen toteutus, mikä määrittelee mitä roolimallin rooleja mitkäkin luokat esittävät (Keller & Sametinger, 2003). Yleensä eri suunnittelukomponenttien toteutukset ovat vuorovaikutuksessa keskenään, kun luokat esittävät niiden rooleja. Tällä tasolla toteutuksessa on jo sovelluskohtaista toiminnallisuutta ja alkuperäisten suunnittelukomponenttien tieto säilytetään kommentoinnissa, kuten *Javadoc* (Sun Microsystems, 2006). Keller ja Sametinger (2003) käyttävät `@pattern` -tagia kertomaan suunnittelukomponentin nimen, esitetyn roolin ja toteutuksen nimen.

Suunnittelukomponenttien yhdistäminen tarkoittaa yhdistettävien komponenttien roolien liittämistä uusiin ja olemassaoleviin luokkiin tai rajapintoihin. Aina suunnittelukomponenttien rooleja ei kannata yhdistää keskenään, joten sitä varten käytetään suunnitteluraj oitteita. Suunnitteluraj oitteet ovat samankaltaisia kuin kohdassa 3.3.2 esitetyt roolikaavioiden yhdistämisraj oitteet. Keller ja Sametinger esittävät niiden lisäksi kuitenkin kaksi muuta rajoitetta:

- ali (sub): roolia A esittävän luokan pitää olla joko roolia B esittävän luokan aliluokka tai roolia B esittävän rajapinnan toteuttava luokka

- ylä (super): roolia A esittävän luokan pitää olla joko roolia B esittävän luokan yläluokka tai roolia B esittävän luokan toteuttama rajapinta.

Suunnitteluraj oitteiden esittämiseen Keller ja Sametinger käyttävät UML:n *oliorajoitekieltä* (OCL).

Suunnittelukomponenttien käyttö ei edellytä tiettyä prosessia, vaan suunnittelija voi käyttää suunnittelukomponentteja vapaasti. Keller ja Sametinger (2003) suosittelevat

kuitenkin suunnittelun kehittämistä suunnittelukomponentteihin perustuen, sillä ne antavat sovelluksesta paremman yleiskuvan kuin perinteinen luokkakaavio. Vaikka luokkia tarvitaankin täydellisen esityksen tekoon, antavat suunnittelukomponentit hyvän esityksen näiden luokkien välisestä vuorovaikutuksesta. Suunnittelukomponenttiesitykseen on turvallisempaa tehdä muutoksia, kuten lisätä tai poistaa rooleja, toisia suunnittelukomponentteja tai luokkia. Myös koodin muutokset, kuten jonkin metodin poistamisen vaikutukset ovat selkeämmin nähtävissä suunnittelukomponenttiesityksestä (Keller & Sametinger, 2003).

3.4.2 Komponenttiperustaisten kehysten suunnittelu suunnittelumalleja käyttäen

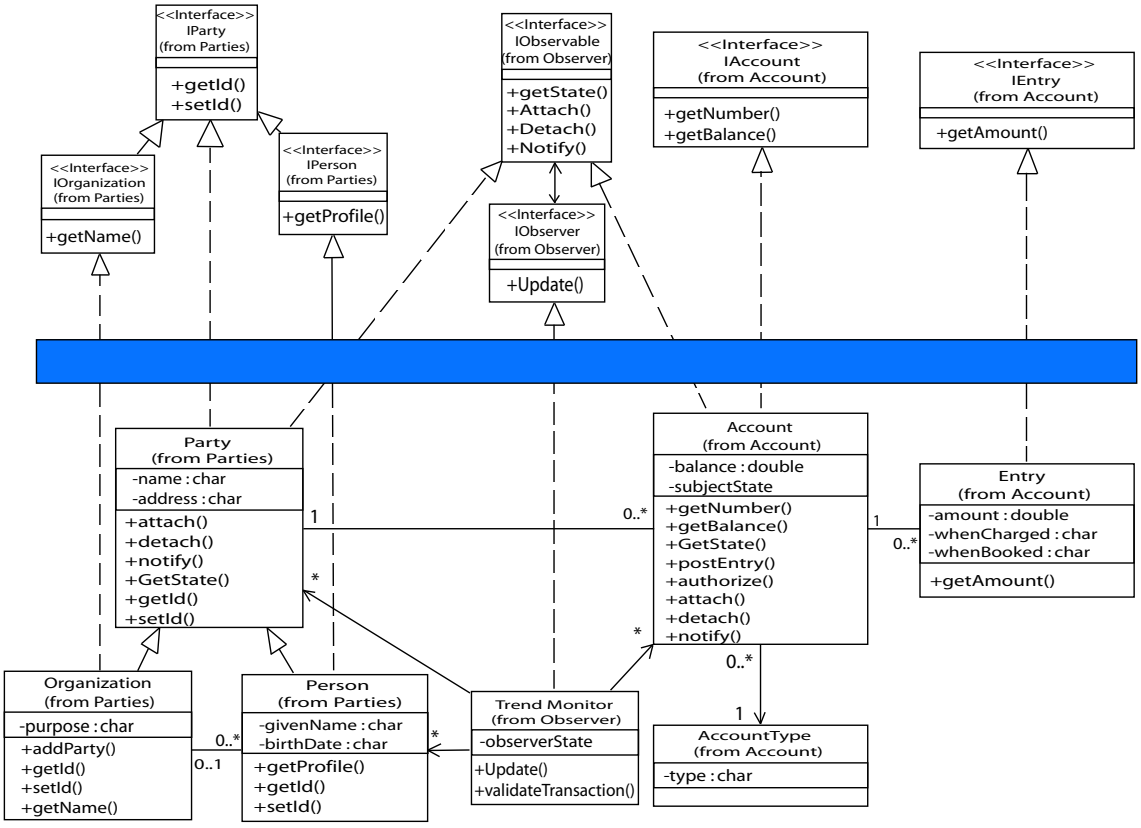
Larsenin (1999) rakenteellinen tapa yhdistää suunnittelumalleja osana ohjelmistokehysten suunnittelua. Suunnittelumallien yhdistämiseen Larsen käyttää UML-luokkakaaviota ja komponentteja. Komponentti on suunnittelumallin staattisen rakenteen toteuttava esitys ja se voi olla toteuttamassa monia suunnittelumalleja ja kehyksiä ja suunnittelumalli tai kehys voidaan toteuttaa monilla komponenteilla. Larsenin määritelmissä kehys sisältää useamman suunnittelumallin ja sillä on useita laajennuspisteitä (extension points) erilaisille sovelluksille.

Larsenin työ keskittyy kehysten suunnitteluun ja toteutukseen käyttäen suunnittelumalleja suunnitteluvaiheessa ja komponentteja toteutusvaiheessa. Suunnitteluvaiheessa käytetään UML:n rajapintaluokkia mallintamaan suunnittelumallien rajapintoja, jotka lisätään suunnittelumallin luokkakaavioesitykseen. Näin muodostettuja suunnittelumallien luokkakaavioita kutsutaan komponenteiksi. Suunnittelumallit yhdistetään toisiinsa yhdistämällä ne toteuttavat komponentit. Tämä tapahtuu sovittamalla suunnittelumallin rooleja yhteen muiden komponenttien vastaavien roolien kanssa. Käytännössä se tarkoittaa rajapintojen liittämistä yhteen muiden komponenttien rajapintojen kanssa tai toteuttamalla muiden komponenttien rajapintoja. Kehyksen viimeistellyssä versiossa Larsen käyttää UML:n komponenttikaavioita esittämään kehyksen fyysisten komponenttien yhdistelmänä.

Lyhyt esimerkki Larsenin tavasta näyttää kehyksen suunnittelun ja toteutuksen tärkeimmät toiminnot suunnittelumallien käytön kannalta.

Kun kehyksen sovellusalue, tuetut käyttötapaukset ja toimijat on selvitetty, valitaan

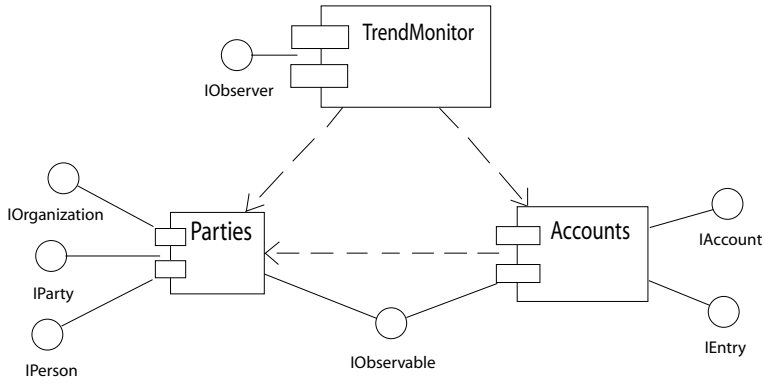
suunnittelussa käytettävät suunnittelumallit. Larsenin esimerkin tapauksessa käytetään tiedonmallinnusmallia Ryhmä (Party), analyysimallia Tili (Account) ja suunnittelumallia *Tarkkailija* (Observer). Suunnittelumalleista valitaan kehykselle tärkeimmät rajapinnat ja sijoitetaan niihin roolit ja toimijat. Esimerkissä valitaan Ryhmä-mallin rajapinnat IParty, IOrganization ja IPerson, sillä ne esittävät kehyksen kannalta tärkeitä rooleja. Samoilla perusteilla valitaan Tili-mallin IAccount ja IEntry ja Tarkkailija-mallin IObserver ja IObservable. Sitten valitut rajapinnat rooleineen ja roolit toteuttavat luokat sijoitetaan yhteen muun kehyksen luokkarakenteen kanssa. Lopputulos esimerkin tapauksessa on kuvan 12 luokkakaavio.



Kuva 12: Kehyksen rajapinnat (yläosassa) ja ne toteuttavat luokat (alaosassa) luokkakaaviossa.

Komponenttien rajapinnat ovat kuvan yläosassa ja toteutus alaosassa. Kehyksen komponenttiesitys (kuva 13) näyttää vain suunnittelumallit toteuttavat komponentit ja niiden rajapinnat.

Esimerkissä suunnittelumalleista valittiin kehyksen toiminnan kannalta tärkeät osat ja yhdistettiin ne muuhun kehyksen rakenteeseen. Mitään erityistä valinta- tai yhdistämisprosessia ei Larsenin tavassa esitellä.



Kuva 13: Kehyksen komponenttiesitys.

3.4.3 Mallisuuntautunut analyysi ja suunnittelu

Yacoubin ja Ammarin (2004) *mallisuuntautunut analyysi ja suunnittelu* (POAD, Pattern-Oriented Analysis and Design) esittää suunnittelumallien rakenteellisen yhdistämistavan luokkakaavioita käyttäen. POAD perustuu rajapinnan määrittelemiseen jokaiselle toteutetulle suunnittelumallille. Tämä rajapinta antaa määritelmän eri suunnittelumallien toteutusten olioiden välisille vuorovaikutuksille ja mahdollistaa käyttäytymisen analyysin teon. Luontimallit (constructional patterns), joissa suunnittelumallin rajapinta voidaan helposti määritellä, soveltuvat tälle rakenteiselle yhdistämistavalle parhaiten.

Yhdistämiseen POAD käyttää hierarkkisia loogisia malleja, jotka piilottavat yksityiskohdat, joita ei ole hyödynnetty suoraan tietyssä suunnittelutason abstraktiossa. POAD:issa käytetään suunnittelumalleille lisämalleja (additional models). Ensimmäisellä tasolla suunnittelumalleille ei esitetä rajapintoja tai sisäisiä yksityiskohtia, toisella tasolla näytetään rajapinnat ja niiden yhteydet toisiinsa ja lopuksi yksityiskohtainen taso näyttää suunnittelumallien yksityiskohdat ja rajapinnat. Täten POAD-malleissa voidaan niihin kuuluvia osia jäljittää ja tunnistaa muun suunnittelun joukosta myös eri suunnittelutasojen välillä (Yacoub & Ammar, 2004).

Suunnitteluvaiheen alussa tehdään järjestelmän toiminnallisiin vaatimuksiin pohjautuva analyysi, jossa pyritään tunnistamaan toiminnallisuuden toteuttamiseen tarvittavat komponentit ja niiden väliset suhteet. Tämän jälkeen komponenteille voidaan etsiä niiden vastuiden ja toiminnallisuuden nojalla parhaiten soveltuvat suunnittelumallit, jotka vastaavat kunkin komponentin esittämää ratkaistavana olevaa suunnitteluongelmaa. Esimerkkitapauksena suunnittelumallien yhdistämisestä esitetään Wamplerin

(2002) Wmvc-kehyksen rakentaminen. Esimerkin Tarkkailija-malli tarjoaa kuvauksensa mukaisesti ratkaisun mallin ja näkymän väliseen vuorovaikutukseen. Mallikomponentti voidaan ajatella Tarkkailijan subjektina (subject), joka informoi tilansa muutoksista näkymäänsä eli tarkkailijaa (observer). Kontrolleri-komponenttien toteutus voi olla hyvinkin erilainen, joten niille olisi tarjottava toteutuksesta riippumaton yhtäläinen rajapinta käyttäjän komentojen välittämiseen. Tämän rajapinnan tarjoaa Komento-suunnittelumalli, joka kapseloi pyynnön olioksi niin, ettei pyynnön lähettävän olion tarvitse tietää vastausta.

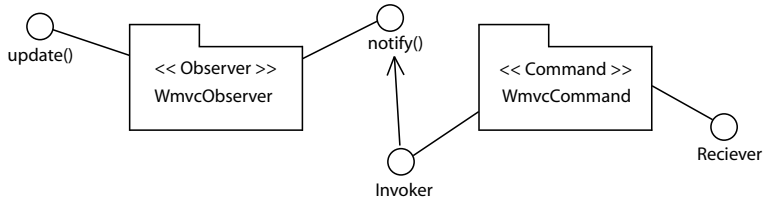
Kun komponenteille on valittu suunnittelumallit, on seuraavaksi liitettävä suunnittelumallit yhteen Wmvc-kehyksen kasaamiseksi. Suunnittelumallien yhteenliittäminen voidaan tehdä POAD-menetelmän mukaisesti kolmessa vaiheessa: rakentamalla karkea suunnittelumalli-tason UML-kaavio, täsmentämällä seuraavaksi se suunnittelumallien rajapinnat esittäväksi UML-kaavioksi, josta sitten lopulta tarkentaen yksityiskohtaiseksi suunnittelumallitason UML-kaavioksi. Ensimmäisessä vaiheessa tehdään aiemmin valittujen suunnittelumallien toteutus nimeämällä ne sovelluskohtaisesti. Esimerkin tapauksessa Tarkkailija nimetään WmvcObserveriksi ja Komento WmvcCommandiksi. Sitten suunnittelumallien tätä toteutusta ja suhteita käyttäen rakennetaan suunnittelumallitason UML-esitys pakettikaaviolla (kuva 14).

Kuva 14: Suunnittelumallitaso.

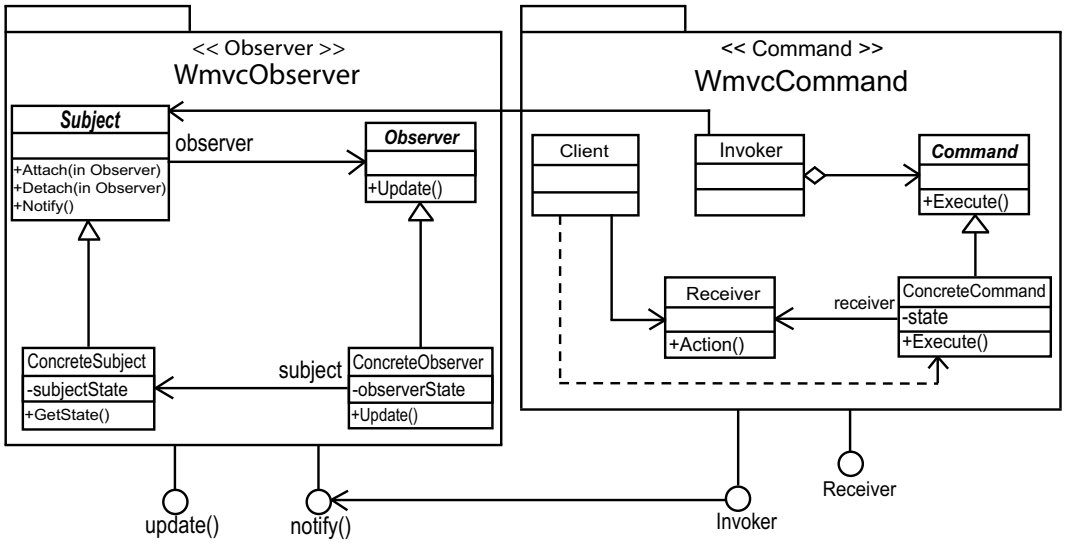
Toisessa vaiheessa analysoidaan suunnittelumallien edellisessä vaiheessa saatujen toteutusten väliset suhteet. Ensiksi määritellään toteutusten rajapinnat, jotka kertovat luokat ja niiden metodit, jotka osallistuvat suunnittelumallien väliseen vuorovaikutukseen. Näitä rajapintoja voi olla suunnittelumallista riippuen useitakin, kuten esimerkin tapauksessa Tarkkailijalla ja Komennolla molemmilla kaksi. Tarkkailijalla metodi notify() subjektissa ja update() tarkkailijassa. Komennolla on rajapintaluokkina Invoker ja Receiver. Sitten samaistetaan suunnittelumallien väliset suhteet niiden rajapintojen välisiksi suhteiksi, minkä tuloksena saadaan suunnittelumallitason UML-esitys rajapinnoilla täydennettynä (kuva 15).

Kolmannessa vaiheessa rakennetaan yksityiskohtainen suunnittelumallitason UML-esitys (kuva 16). Tämä saadaan lisäämällä jokaiseen edellisen vaiheen määrittelemään

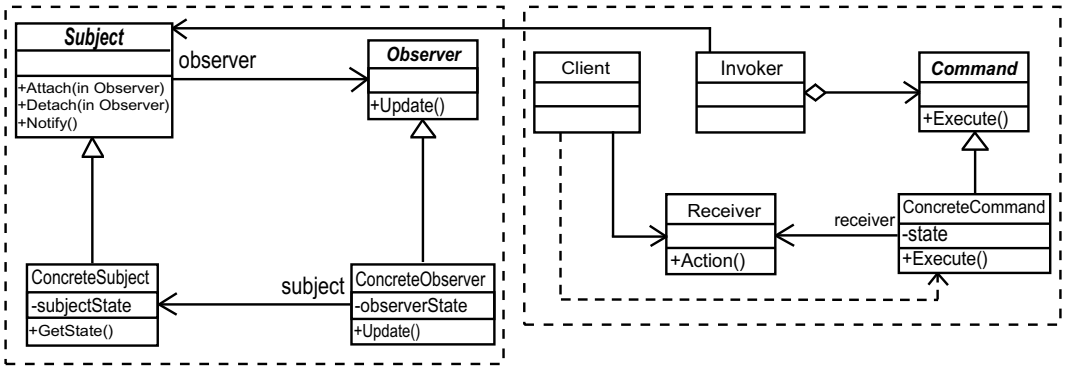
suunnittelumalliin sen sisäinen luokkarakenne GoF-kirjassa (Gamma & al., 1995) esitetyllä tavalla.



Kuva 15: Suunnittelumalli-taso täydennettynä rajapinnoilla.



Kuva 16: Yksityiskohtainen suunnittelumalli-taso.



Kuva 17: Alustava luokkakaavio.

POAD-menetelmässä suunnittelumallien yhdistämisen jälkeen saatu kehys hienosäädetään. Lisätään kehykseen sovelluskohtainen luonne nimeämällä suunnittelumallien luokat ja metodit sovelluksen kannalta mielekkäällä tavalla ja korvataan rajapinta-kohtainen esitys suunnittelumallien välisistä suhteista suoraan niiden

sisäisten luokkien välisiksi. Lisäksi kehyksen suunnittelua voitaisiin optimoida eliminoimalla ylimääräisiä abstrakteja luokkia ja ryhmittelemällä luokkia uudelleen niiden välisen vuorovaikutuksen ja vastuiden mukaan. Esimerkin tapauksessa aikaan saatu kehys jätetään kuitenkin tähän (kuva 17).

3.4.4 Yhdistelmämallit aspektien tuottamiseksi

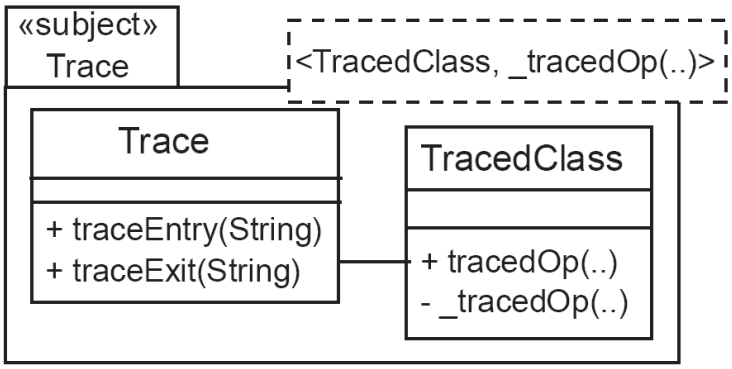
Clarke ja Walker (2001) esittelevät *yhdistelmämallien* konseptin tukemaan suunnitteluja, jotka on kehitetty *subjektiperustainen suunnittelu -mallia* (SOD, Subject-Oriented Design -model) (Clarke et al., 2000) käyttäen. SOD-malli on tullut subjektiohjelmoinnista, missä voidaan kehittää ja koodata subjekteja tukemaan erillisiä läpileikkaavia (cross-cutting) vaatimuksia, *aspekteja*. Esimerkkejä aspekteista ovat lokiinkirjoitus (logging) tai tapahtumien jäljitys, virheestä palautuminen ja tapahtumien samanaikaistaminen. Niillä on vaikutusta laajalle ja siksi myös niiden tuki on jakautunut useisiin luokkiin. Sellaiset vaatimukset antavat haastetta uudelleenkäytettävien ja ylläpidettävien suunnitteluiden kehittämiselle (Clarke & Walker, 2001).

SOD-malli keskittyy näihin aspekteihin hajoittamalla suunnittelun erilaisiksi suunnittelun malleiksi, *suunnittelusubjekteiksi* (design subjects). Subjekti on kokoelma luokkia tai luokkien osia, joista koostuva luokkahierarkia mallintaa tiettyä sovellusalaa omalla subjektiivisella tavallaan. Subjekti voi olla kokonainen sovellus tai pienempi kokonaisuus, joka pitää yhdistää muuhun sovellukseen. Kukin suunnittelusubjekti keskittyy aspektin suunnitteluun (Clarke & Walker, 2001).

Clarke ja Walker (2001) esittelevät yhdistelmämallit keinona kehittää uudelleenkäytettäviä suunnittelumalleja näille aspekteille. Yhdistelmäsuunnittelumalli määrittelee aspektin suunnittelun ja sen kuinka suunnittelua voidaan uudelleenkäyttää. Se perustuu UML:n *kaavaimiin* (templates) ja SOD-malliin.

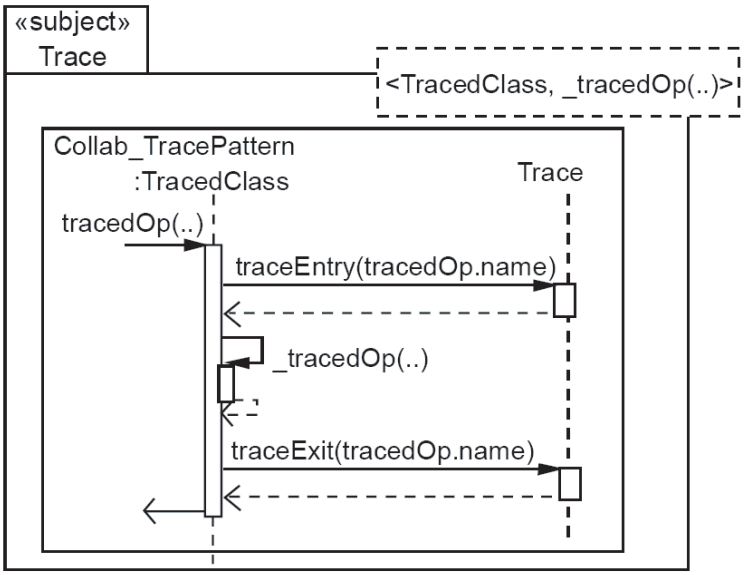
Yhdistelmäsuunnittelumallin rakentaminen alkaa käytettävän kaavaimen määrittelyllä. UML:n kaavainta käytetään perustana muiden mallielementtien generoimiseen ”Sidonta”-riippuvuussuhdetta käyttäen. Sidonta määrittelee kaavaimen mallielementtien kaavainparametrit korvaavat argumentit. Yhdistelmämallin luokkien kaavainparametrit esitetään subjektipaketissa katkoviivalla rajatussa suorakulmiossa. Esimerkki yhdistelmämallista on kuvassa 18. Siinä Trace-yhdistelmämalli esittää tavallisen luokan Trace, viittaukset malliluokkaan `TracedClass` ja kaavainparametriin

`_tracedOp()`.



Kuva 18: Kaavaimen määrittely Trace-yhdistelmämallissa (Clarke & Walker, 2001).

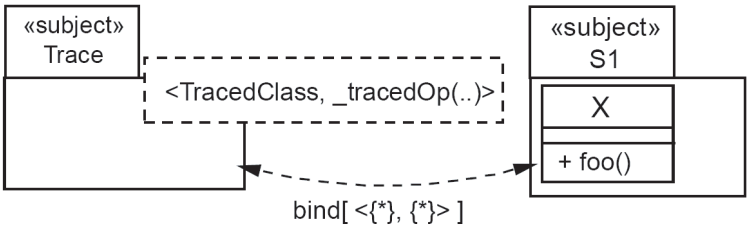
Seuraavaksi yhdistämisessä tehdään läpileikkaavan käyttäytymisen määrittely. SOD-malli tukee läpileikkaavan käyttäytymisen yhdistämistä sen kanssa samanlaiseen käyttäytymiseen. Tämä tarkoittaa eri suunnittelusubjektien toisiaan vastaavien operaatioiden yhdistämistä. Sitä voidaan hyödyntää läpileikkaavien käyttäytymisten suunnittelumallien määrittelyyn. Suunnittelija voi viitata alkuperäisiin ja yhdistettyihin operaatioihin erikseen, määrittelemällä alkuperäisen kaavainparametrinä ja viittaamalla yhdistettyyn operaatioon lisäämällä alaviivan kaavainnimeen. Luotuun tulosoperaatioon viitataan samalla nimellä ilman alaviivaa.



Kuva 19: Läpileikkaavan käyttäytymisen suunnittelumallien määrittely (Clarke & Walker, 2001).

Kuvassa 19 esimerkin malliluokan `TracedClass` käyttäytymisen tapauksessa operaation `_tracedOp(..)` korvaavan operaation suorittaminen aiheuttaa yhdistetyssä subjektissa operaation `traceEntry()` suorittamisen ennen korvaavaa operaatiota ja operaation `traceExit()` suorittamisen korvaavan jälkeen.

SOD-malli määrittelee yhdistelmäsuhteen tukemaan erilaisten subjektien integroimista yhdistellyksi tulokseksi ja UML määrittelee Sidonta-suhteen kaavainmääritelmien ja ne korvaavien elementtien välille. Yhdistelmämalli yhdistää nämä kaksi merkintätapaa `bind[]` -kiinnityksellä. Kiinnityksen aaltosulkeissa esitettävät parametrit ovat samassa järjestyksessä kuin vastaavat kaavainmallin parametrit. Kuvassa 20 kaikki `S1` luokat korvaavat `TracedClass` luokan ja luokan `X foo()` korvaa operaation `_tracedOp(..)`.

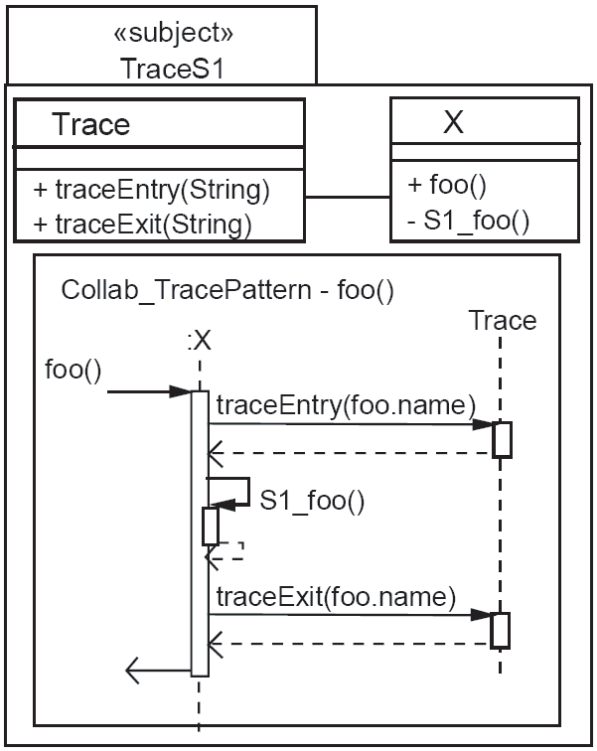


Kuva 20: Sidonnan määrittely yhdisteelle (Clarke & Walker, 2001).

Yhdisteen `bind[]` -kiinnitys voi määritellä malliluokille ja kaavainperaatioille useita korvaajia. Yhdistellyssä subjektissa malliluokan ominaisuudet lisätään kaikkiin korvaaviin luokkiin. Esimerkin tapauksessa luokalla `X` on luokan `TracedClass` ominaisuudet (kuva 21).

Kaavainmallin ulkopuoliset elementit lisätään kuhunkin lopputulokseen kerran. Esimerkiksi kuvan 21 `Trace`-luokka ei ole malliluokka ja esiintyy täten tulossubjektissa. Jokainen kaavainperaation korvaava operaatio nimetään uudelleen korvaamalla viittaus kaavainoperaatioon sopivalla operaation nimellä. Esimerkin tapauksessa `_tracedOp()` muuttuu `foo()`:ksi. Operaatioiden uudet vuorovaikutukset määritellään ja jokaisen operaation delegoinnit toteutetaan yhdistelmämallissa määritellyin yhteistöin.

Yhdistämisen lopputulokset riippuvat yhdistettävistä luokista. Esimerkiksi periytymisten yhdistäminen voi tuottaa useita periytymisiä yhdistettyyn subjektiin, vaikka yhdistetyissä subjekteissa oli alunperin vain yksi perintä.



Kuva 21: Yhdistämisen tulos (Clarke & Walker, 2001).

Subjektien yhdistäminen voidaan suorittaa jo ohjelmiston suunnitteluvaiheessa, mutta Clarke ja Walker (2001) suosivat sen tekemistä vasta toteutusvaiheen jälkeen jolloin kukin suunnittelusubjekti on jo toteutettu. Suunnitteluvaiheen yhdistämisen hyvinä puolina he mainitsevat kuitenkin aikaisen mahdollisuuden tarkistaa yhdistetyn subjektin merkityksen ja sen yhteyksien oikeellisuuden.

3.5 UML suunnittelumallien yhdistämisessä

UML on *OMG*:n (Object Management Group, Inc) kehittämä graafinen mallinnuskieli, joka on suunniteltu erilaisten ohjelmistojen määrittelemiseksi, rakentamiseksi, havainnollistamiseksi sekä dokumentoimiseksi (OMG, 2005). UML:n versio 1.4.2 on hyväksytty ISO-standardiksi, mutta UML:n uusin versio on 2.1.1. OMG:n tavoitteita ovat edistää olioteknologioiden kasvua ja vaikuttaa niiden suuntaan. Tähän pyritään *OMA*:lla (Object Management Architecture), joka tarjoaa käsitteellisen infrastruktuurin, jolle kaikki OMG:n määritelmät perustuvat. UML:n tavoitteena on tukea tietystä ohjelmointikielestä sekä ohjelmistoprosessista riippumattomia määrityksiä, millä pyritään kasvattamaan eri tekniikoiden yhteensopivuutta. Tätä hanketta tukevat korke-

an tason käsitteet, kuten komponentit, yhteistoiminnat, kehykset ja suunnittelumallit. Tässä luvussa esittelen lyhyesti UML-tuen suunnittelumalleille, yhdistelmämallin määritelmän ja teen yhteenvetoa edellä esitellyistä suunnittelumallien yhdistämistekniikoista UML 2.1.1:n määritelmiä ajatellen.

3.5.1 UML:n yhdistelmämalli

UML:n *EDOC-profiili* (EDOC, Enterprise Distributed Object Computing) sisältää profiilin suunnittelumalleille (OMG, 2004). Spesifikaatiossa määritellään UML:n ja *ECA*:n (Enterprise Computing Architecture) profiilin olennaisten osien käyttäminen oliomallien, kuten *liiketoimintaoliomallien* (BFOP, Business Function Object Patterns) esittämiseksi mallisovellusmekanismeja (pattern application mechanisms) käyttäen.

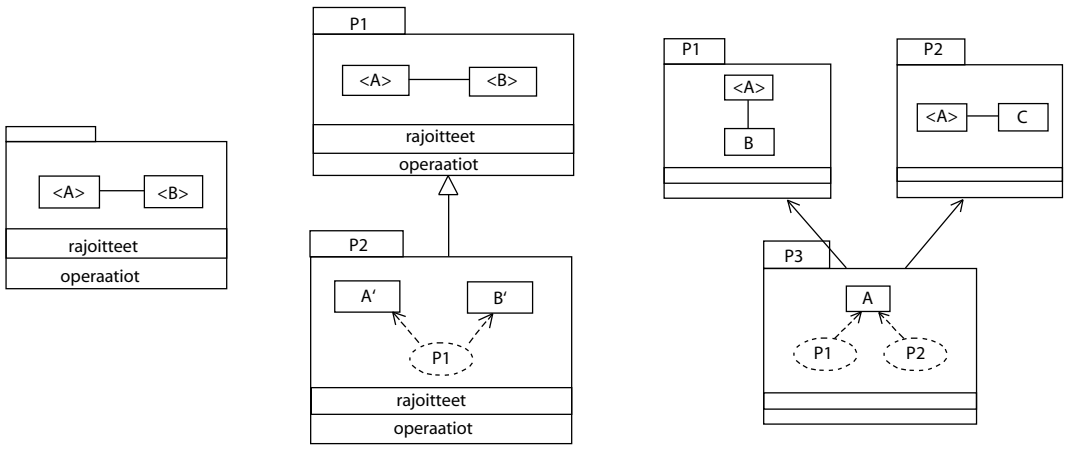
Tämä spesifikaatio keskittyy suunnittelumallien käytössä enemmän parantamaan oliomallien (object models) jaettavuutta ja uudelleenkäytettävyyttä kuin tukemaan ponnisteluja mallinnuksen parissa esittämällä siihen hyviä mallinnustekniikoita. Tähän pyritään tukemalla seuraavia ominaisuuksia:

- Mallin täytyy tarjota ennaltamäärättyjä normimallinnusrakenteita, ei vain mallinnustapoja ja -notaatiota.
- Ennaltamäärättyjen mallinnusrakenteiden pitäisi sisältää yleiset perusobjektit, kuten päiväys, valuutta ja maakoodi, joita voidaan sitten käyttää ilman selvitystä.
- Yleiset koosteoliot, kuten liiketoimintakokonaisuudet *asiakas* (Customer), *yhtiö* (Company) tai *tilaus* (Order), pitäisi myös määritellä normimallinnusrakenteina perusolioita käyttäen.
- Liiketoiminnan konseptit, kuten *kaupankäynti* (Trade), *laskutus* (Invoice) ja *sopimus* (Settlement), jotka esitetään yleensä olioiden välisinä suhteina, pitäisi määritellä yleisten perusyhdistelmäolioiden tai perusolioiden yhdistelminä (aggregations). Niidenkin tulee olla ennaltamäärättyjä normimallinnusrakenteita.
- Ne aggregaatiot (aggregations), jotka voidaan ennaltamäärätä käyttäen pohjana perusmalleja (basic and elementary patterns), voidaan määritellä oliomalleiksi (object patterns).

- Suunnittelumallit voivat esittää liiketoimintakonseptia, missä ne mahdollistavat yksinkertaisempien mallien yhdistämisen (aggregation). Täten yhdistämismekanismi (aggregaatio tai kompositio) on oleellinen osa suunnittelumallia.

Liiketoimintakonseptin (business concept) määräävät *liiketoimintasäännöt* (business rules) voidaan esittää rajoitteet sisältävällä suunnittelumallilla. Tämän takia suunnittelumalleille esitellään mekanismi rajoitteiden perintään.

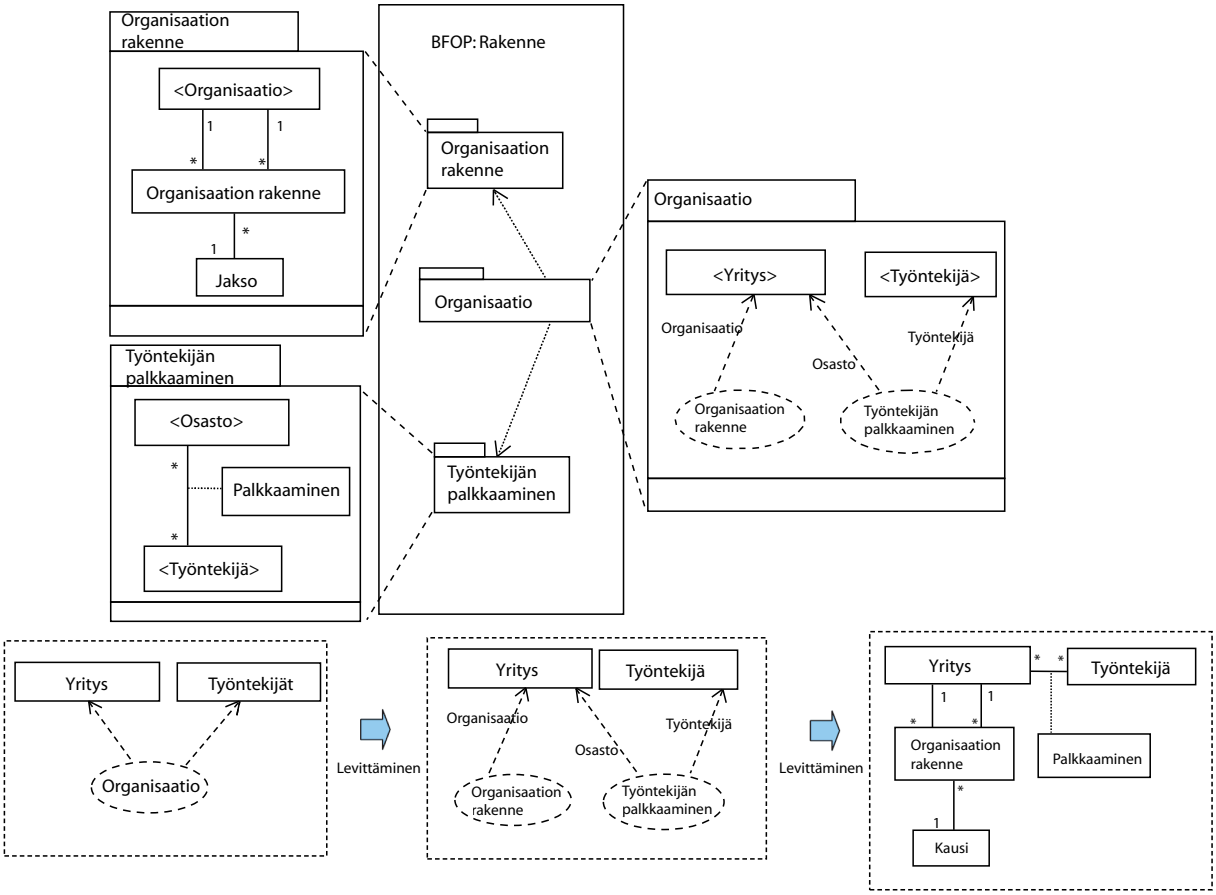
UML esittelee kolme perusmuotoa suunnittelumallin esittämiseksi. Ensimmäinen on *yksinkertainen suunnittelumalli* (kuva 22 (a)) (simple pattern), joka koostuu mallin muodostamiseen tarvittavasta minimaalisesta määrästä elementtejä. Toinen muoto on *periytetty suunnittelumalli* (kuva 22 (b)) (inherited pattern), mikä määritellään periyttämällä toisesta suunnittelumallista. Kolmas on *yhdistelmämalli* (composite pattern), mikä saadaan yhdistämällä useampia suunnittelumalleja. Se on laajennos periytettyyn malliin ja mahdollista monimutkaisempien suunnittelumallien rakentamisen. Kun yhdistetään kaksi suunnittelumallia kuvaamaan yhdistelmämalli, luodaan uusi tyyppi, eli looginen luokka (logical class), mikä jakaa alkuperäisten suunnittelumallien ominaisuudet (characteristics). Looginen luokka esitetään käyttäen UML:n parametrisoitua yhteistyömerkintää. Kuva 22 (c) esittää yhdistelmämallia.



Kuva 22: (a) Yksinkertainen suunnittelumalli, (b) Periytetty suunnittelumalli ja (c) Yhdistelmämalli.

Kuvan 23 yläosan kaavio esittää, kuinka BFOP-hierarkiassa *Organisaatio*-malli (Organization Pattern) saadaan yhdistämällä *Työntekijän palkkaaminen* -malli (Employee Assignment Pattern) ja *Organisaation rakenne* -malli (Organization Structure Pattern). Yhdistelmämalli saadaan toteutettua hierarkkisessa rakenteessa ratkaisemalla suunnit-

telumallin perintä ja yhteistyö. Tämä tapahtuu *levittämällä* (unfolding) yhdistelmämallin parametrisoitu yhteistyömerkintä *komponenttimalliksi* (component pattern) kuvan 23 alaosan esittämällä tavalla. Jos yhdistelmämalli on riittävän hienorakenteinen sisältämään toteutusyksityiskohtia ja sitä voidaan käyttää kuvaamaan komponenttikonsepti, kuten OMG:n *CCA* (Component Collaboration Architecture), jokainen mallipaketti (pattern package) voidaan toteuttaa todellisilla komponenteilla tarvitsematta levittää sitä. Ehdotettua mallikonseptia ja mekanismia voidaan soveltaa komponenttiperustaiseen kehitykseen, mitä EDOC:issa tarvitaan.



Kuva 23: Esimerkki BFOP-rakenteesta ja yhdistelmämallin levittäminen.

3.5.2 Yhdistämistekniikat ja UML 2.1.1

Suurin osa tässä luvussa aiemmin esitellyistä suunnittelumallien yhdistämistekniikoista on kehittynyt samoihin aikoihin UML:n kanssa. Niitä ei siis ole kehitetty tukemaan UML:n standardia. Toisaalta UML pohjautuu osittain samoihin aiemmin käytössä olleisiin mallinnustekniikoihin (OMT jne.), mitä monet yhdistämistekniikoista

kat käyttävät. Täten on mahdollista löytää yhteneväisyyksiä eri kaaviotyyppien ja UML:n välillä. Yhtenä esimerkkinä tästä on UML:n yhteistyönotaatio, mikä pohjautuu osittain OOram-metodiin (Reenskaug, 2000). Varsinkin uusimmat rakenteiset yhdistämistekniikat käyttävät UML-standardin tarjoamia tekniikoita, ja koska UML kehittyy jatkuvasti, sisältyy yhdistämistekniikoihin monesti myös esityksiä UML:ään tehtävistä laajennoksista. Laajennokset ovat yleensä joitakin merkintöjä kaavioissa, joilla pyritään lisäämään kaavion informaatio- tai käyttöarvoa suunnittelumallien yhdistämistarkoituksiin. Esimerkkinä tästä on Kellerin ja Schauerin (1998) ehdotus laajentaa UML:n pakettikaavio kuvaamaan suunnittelukomponentteja.

UML-dokumentaatio (OMG, 2005; OMG, 2007) esittää useita erilaisia kaaviota, joilla voidaan kuvata järjestelmän staattista rakennetta, dynaamista käyttäytymistä tai hallinnoida sovelluksen moduuleita. Staattisen rakenteen kuvaavista kaavioista suunnittelumallien kuvaamiseen yleisimmin käytettäviä ovat *luokkakaavio* (class diagram), *oliokaavio* (object diagram) ja *komponenttikaavio* (component diagram). Järjestelmän dynaamista käyttäytymistä kuvaavat kaaviot suunnittelumalleille ovat *sekvenssikaavio* (sequence diagram) ja *yhteistyökaavio* (collaboration diagram). Sovelluksen moduulien organisoimiseksi ja hallinnoimisen helpottamiseksi kehitetyistä kaavioista käyttävät suunnittelumallit *paketteja* (packages).

UML sisältää oliorajoitekielen (OCL), jonka avulla voidaan määritellä mm. suunnittelumallien rajoitteita. Tällä hetkellä uusin versio kielestä on OCL 2.0 (OMG, 2005b). Rajoitteita tarvitaan, koska kaaviot eivät pysty ilmaisemaan kaikkea tarvittavaa informaatiota, kuten sitä miten tiettyä suunnittelumallia voidaan käyttää yhdessä muiden suunnittelumallien kanssa. Yhdessä OCL:n kanssa käytettynä UML mahdollistaa monipuolisen ja tarkan mallinnuksen. Joissakin suunnittelumallien yhdistämistekniikoissa käytetäänkin OCL:ää, kuten esimerkiksi Keller ja Sametinger (2003), jotka esittävät sen avulla suunnittelukomponenttien roolien yhdistämisen suunnitterurajoitteet.

Staattisen rakenteen kuvaavista kaavioista luokkakaaviota käytetään yleisesti suunnittelumallien rakenteen kuvaamiseen ja täten se on käytössä myös useimmissa yhdistämistavoissa. Oliokaavion ohella yhdistettävien suunnittelumallien mallintamiseen käytetään komponenttikaavioita, kuten kohdassa 3.4.2.

Yhteistyökaavion päätarkoitus on selittää kuinka yhdessä kommunikoivien kokonaisuuksien järjestelmä suorittaa tietyn tehtävän tai joukon tehtäviä tarvitsematta esittää selitykselle epäoleellisia yksityiskohtia. Tämä on erityisen hyödyllistä esitettäessä

standardeja suunnittelumalleja (OMG, 2007).

Seuraavissa alaluvuissa tehdään lyhyt yhteenveto UML:n käytöstä mallikielissä sekä käytöksellisissä ja rakenteellisissa yhdistämistekniikoissa ja eroista UML 2.1.1 versioon.

3.5.3 Mallikielet

Mallikielten käyttämät mallinnustekniikat riippuvat yleensä siitä ajasta, jolloin mallikieli on kehitetty. Ennen UML:n standardoimista käytettiin OMT:tä. Standardoimisen jälkeen suunnittelumallien kuvaamiseen on kuitenkin yleisesti käytetty UML:n luokkakaaviota, oliokaavioita tai sekvenssikaaviota, joten ne ovat yleisiä uusimmissa mallikielissäkin. OMT:n ja UML:n ”sukulaisuuden” ansiosta vanhemmatkin mallikielet ovat kuitenkin esitettävissä UML:ää käyttäen.

3.5.4 Käytökselliset yhdistämistekniikat

Reenskaugin (1996) oliosuuntautunut roolianalyysi ja ohjelmistosynteesi eivät kumpikaan käytä UML:ää. UML:n yhteistyönotaatio kuitenkin pohjautuu osittain OOram-metodiin (Reenskaug, 2000), jonka roolimallikaavion rakenteet, roolit ja niiden väliset yhteydet, vastaavat UML:n yhteistyökaavion elementtejä ja niiden välisiä yhteyksiä.

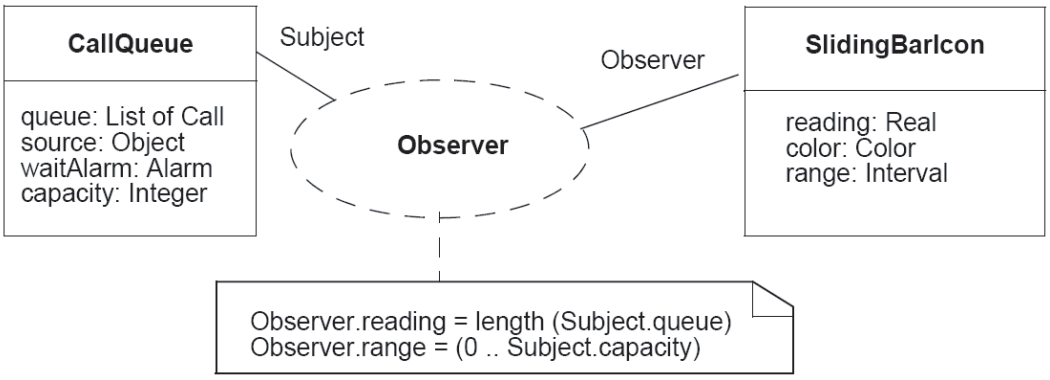
Suurimmat erot UML:ään ovat roolimallin ja UML-oliomallin erot. Roolien merkintätapa keskittyy olion vastuisiin koko olioryhmän sisällä ja rooli kuvaa useiden käyttäytymisten ja ominaisuuksien yhteisvaikutusta. Luokkien merkintätapa taas keskittyy luokista luotujen olioiden mahdollisuuksiin kuvaten joukon olioita yhteisellä käyttäytymisellä ja ominaisuuksilla. Roolimalli kuvaa sekä dynaamisen, että staattisen puolen mallista yhdessä ja samassa kaaviossa, mutta UML:ssä staattiset ja dynaamiset mallit ovat erikseen. Yhteistyökaavio tosin esittää myös staattisia ominaisuuksia yhteistyöhön osallistuvista elementeistä siltä osin kuin se on oleellista niiden osallistumisen kannalta.

Roolianalyysissä ja ohjelmistosynteesissä voisi mahdollisesti hyödyntää UML:ää pyrkimällä kuvaamaan tekniikan käyttämiä rakenteita ja kaavioita korvaavilla UML-kaavioilla. Esimerkiksi UML:n yhteistyökaaviolla voisi esittää ainakin roolimallin

olioiden vuorovaikutuksen.

Riehlen (1997) työ, suunnittelumallien yhdistäminen käyttäen rooleja, soveltaa Reenskaugin ehdottamia roolimallien käsitteitä mallien yhdistämiseen. Yhdistämiseen tekniikka käyttää roolien suhdematriisia, joka luodaan eräänlaisen oliokaavion pohjalta. Käytetyn oliokaavion tarkoitus on esittää vain yhdistämisessä käytettävän esimerkkisovelluksen olioiden hierarkia ilman lisäinformaatiota, joten tehtävään voitaisiin käyttää myös UML:n oliokaaviota.

Kuten Reenskaugin roolimallien tapauksessa, myös Riehlen roolikaavioissa on paljon vastaavuuksia UML:ssä yhteistyökaavioon ja sen yhteistöihin osallistuvien olioiden rooleihin. Roolikaavioiden käyttämiä yhdistämisrajoitteita suoraan vastaavaa tekniikkaa UML:ssä ei yhteistyökaaviolle ole. Toisaalta yhteistyökaavion roolit ja niiden tyypit määräävät luokat määrittelevät osallistuvien olioiden ominaisuudet ja rajoittavat rooleja esittäviä olioita (kuva 24). Kuvassa subjektiroolia esittää CallQueue ja tarkkailijaroolia SlidingBarIcon. Rajapinnat mahdollistavat olioiden ulkoisesti havaittavien ominaisuuksien määrittelyn tarvitsematta päättää olion luokkaa. Tämän vuoksi yhteistyönotaation roolit esitetään usein rajapinnoilla.



Kuva 24: Roolit UML:n yhteistyökaaviossa (OMG, 2007).

Boschin (1998b) superimpositio-tapa käyttää suunnittelumalleja ja kehyksiä arkkitehtuurisina palasina ja yhdistelee rooleja ja komponentteja tuottamaan sovelluksia. Tekniikka käyttää arkkitehtuurifragmentin ja superimposition esittämiseen LayOM:n rakenteita, jotka eivät vastaa UML:ää. Joitakin samaistuksia oliokaavioon voidaan tehdä, mutta esimerkiksi kerrosten esittämiseen ei UML:stä löydy suoraa keinoa.

3.5.5 Rakenteelliset yhdistämistekniikat

Kellerin ja Sametingerin (2003) työ, ohjelmiston laatiminen suunnittelukomponentteja käyttäen, näyttää kuinka suunnittelukomponentit kehittyvät sovellusalueen toteutukseksi. Tekniikka käyttää UML:n oliorajoitekieltä, OCL:ää suunnittelukomponenttien roolien yhdistämisrajoitteiden esittämiseen. Roolimallit ovat Reenskaugin (1996) roolimalleja, joten myös ne ovat periaatteessa esitettävissä UML:n oliokaavion roolien avulla. Suunnittelukomponentin rakenteen mallinnuksen komponentin kuvaus -taso kertoo syyt valita tietty suunnittelu ja osa sitä on suunnittelumallien kuvaus. Siihen voidaankin käyttää UML:n luokka- tai oliokaaviota.

Larsen (1999) käyttää suunnittelumalleja komponenttipohjaisten kehysten kehittämiseen UML:n luokkakaavioita käyttäen. Työssä kehitetään hierarkkinen arkkitehtuurinen suunnittelu, jossa komponentti voi toteuttaa monia suunnittelumalleja ja kehyksiä ja suunnittelumalli tai kehys voidaan toteuttaa monen komponentin yli. Suunnittelumallien yhdistämiseen käytettäviä rajapintoja mallinnetaan UML:n rajapintaluokilla. UML:n komponenttikaavioita käytetään esittämään kehys yhdistelmänä fyysisiä komponentteja.

POAD eli mallisuuntautunut analyysi ja suunnittelu (Yacoub & Ammar, 2004) esittää suunnittelumallien rakenteellisen yhdistämistavan käyttäen suunnittelumallien UML-luokkakaavioita. Suunnittelumallien rajapinnat käyttävät UML:n rajapinta-merkintää. Tekniikassa UML:n pakettikaaviota, luokkakaaviota ja rajapintoja yhdistetään esitettäessä suunnittelumalleja eri abstraktiotasoilla. Suunnittelumallien yhdistämisen tulos on luokkakaavio, jonka hiominen tehokkaammin sovelluskohteeseen sopivaksi tehdään UML:n sekvenssikaavion avulla.

Viimeisenä yhdistämistekniikkana esiteltiin aspektiohjelmoinnin konseptiin pohjautuva kehitysteknologia, jossa käytetään yhdistelmämalleja aspektien tuottamiseksi. Tekniikka käyttää UML:n luokkakaaviota ja kaavaimia yhdistelmämallien määrittelyyn. Kyseessä on luokkakaavion parametrisoitu paketti. Subjekti esitetään luokkien tai niiden osien kokoelmana. Yhdistämistekniikassa käytetyssä UML 1.4:ssä mistä tahansa mallielementistä saattoi tehdä kaavaimen. UML 2.1.1:ssä kaavainten käyttöä on rajoitettu ja vain ne mallielementit (model element), joille on tarkoituksellista olla kaavainparametreja, voivat enää olla kaavaimena (OMG, 2007 s. 617). Muita muutoksia kaavaimiin ei kuitenkaan ole tullut.

4 Suunnittelumallien yhdistäminen Enterprise Architect -ympäristössä

Tässä luvussa esittelen erään markkinoilla olevista suunnittelumalleja tukevista mallinnustyökaluista, Sparxsystemsin Enterprise Architectin. Muita vastaavia ohjelmistoja ovat muun muassa IBM:n Rational Software Architect, Borlandin Together ja TECHNOLOGIC ARTS:in Pattern Weaver. Esimerkissä esittelen kohdassa 3.4.3 tarkastellun POAD-yhdistämistekniikan soveltamisen käytännössä Enterprise Architectia käyttäen. Lopuksi pohdin hieman tarkastelemani ohjelmiston ominaisuuksia ja kuinka se suoriutui esimerkkitapauksesta .

Esimerkissä ei tehdä valmista ohjelmaa, vaan suoritetaan palautteenhallintakehyksen (Feedback Control Framework) suunnittelu ja mallinnus POAD-prosessin mukaisesti. Palautteenhallintajärjestelmiä käytetään valvomaan joko ulkoista ympäristöä tai muita järjestelmän komponentteja. Esimerkissä palautteenhallintajärjestelmille luodaan uudelleenkäytettävä, suunnittelumalleihin perustuva kehys. Rajoitumme Enterprise Architectin osalta vain suunnittelumallien käyttöä tukeviin ominaisuuksiin. POAD-prosessi sisältää kolme vaihetta: analyysin, suunnittelun ja suunnittelun hienosäätämisen. Analyysissä selvitetään järjestelmän toiminta ja osallistujat, sekä valitaan suunnitteluongelman parhaiten ratkaisevat suunnittelumallit.

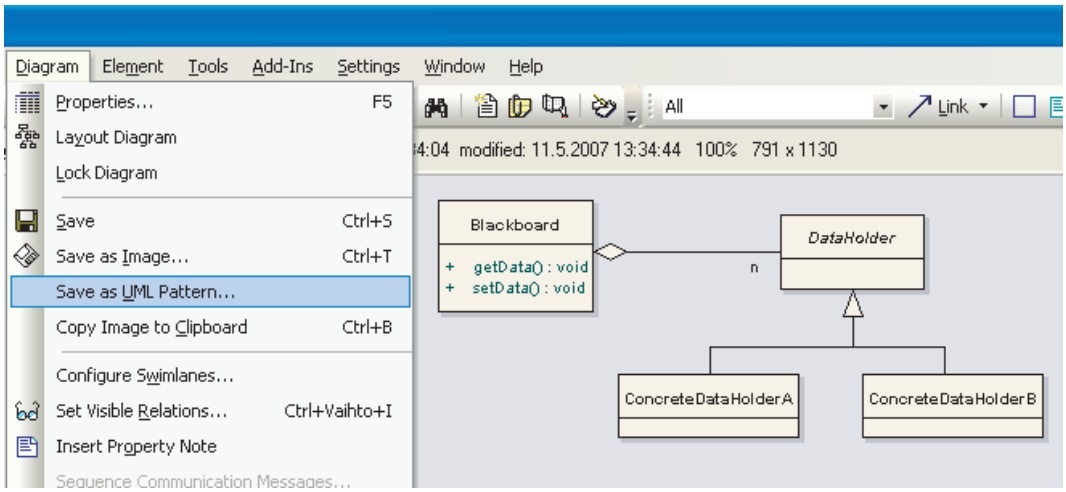
Esimerkin tapauksessa aloitetaan tekemällä käyttötapauskaavio, joka onnistuu Enterprise Architectin käyttötapaustyökalulla. Kaavion avulla tunnistetaan esimerkkikehykseen tarvittaviksi komponenteiksi feedforward-komponentti käsittelemään virhetilanteet ja valvomaan järjestelmän kohdetta, feedback-komponentti mittaamaan kohdetta, käsittelemään saamansa tiedon ja antamaan palautetta muulle järjestelmälle, sekä error calculation -komponentti vertaamaan järjestelmän syöte- ja palautetietoja ja tuottamaan sen perusteella saadun virheen. Tunnistettujen komponenttien tehtävänkuvauksen perusteella päädytään käyttämään Strategia-, Tarkkailija- ja Blackboard-malleja.

Analyysivaiheen jälkeen aloitetaan kehyksen suunnittelu. Ensimmäisessä suunnitteluvaiheessa kehyksestä tehdään suunnittelumallitason kaavio, jossa esitetään valitut suunnittelumallit ja niiden suhteet toisiinsa pakettikaavioita käyttäen. Suunnittelumallit nimetään sovellusaluekohtaisesti. Seuraavaksi suunnittelumallitason kaaviota tarkennetaan suunnittelumallien välisten rajapintojen välisillä suhteilla. Luotavassa suunnittelumalli-tason rajapinnoilla täydennetyssä kaaviossa rajapinnat esitetään UML:n

rajapintamerkinnällä, viivalla pakettiin yhdistetyllä ympyrällä. Enterprise Architect mahdollistaa pakettikaavioiden laatimisen, joten suunnittelumallitason kaavion tekeminen esimerkkikehyksen ensimmäisessä suunnitteluvaiheessa onnistuu. Toisessa vaiheessa tehtävä rajapintojen lisääminen POAD-prosessin esittämällä UML-merkinnällä tapahtuu rajapintaelementillä, joka luokkakaavion työkaluvalikoimasta valittuna tekee ensin luokan, jonka ominaisuuksista sitten voidaan vaihtaa ulkoasuksi ympyrä. Jos rajapintaelementin valitsee komponenttikaavion valikoimasta, tulee se oletusarvoisesti ympyrämerkinnällä. Kolmannessa vaiheessa rakennetaan yksityiskohtainen suunnittelumalli-tason UML-esitys, jossa edellisen vaiheen kaavion pakettien sisään lisätään suunnittelumallien luokkakaaviot. Jokaisen suunnitteluvaiheen kaaviot kannattaa tehdä erikseen niiden muokattavuuden säilyttämiseksi ja POAD-prosessin periaatteen mukaisesti suunnittelumallien jäljitettävyyden takia. Enterprise Architectissa kaaviosta voidaan tehdä kopio, jolloin suunnittelumalli-tason kaavion kopion päälle on helppo tehdä sen rajapinnoilla täydennetty versio, jonka kopion päälle sitten saadaan vuorostaan yksityiskohtainen suunnittelumalli -tason kaavio.

Enterprise Architect helpottaa POAD-prosessin kolmannen suunnitteluvaiheen työtä tarjoamalla suunnittelumallien lisäys -toiminnon, jossa muualta tuotuja, XMI-muodossa määriteltyjä suunnittelumalleja voidaan tuoda osaksi muuta suunnittelua. Lisäksi käyttäjät voivat luoda uusia suunnittelumalleja. Suunnittelumallin luominen tapahtuu rakentamalla suunnittelumallia kuvaava luokkakaavio. Luotavaksi suunnittelumalliksi valittiin BlackBoard-malli, sillä sitä ei ollut Enterprise Architectin valikoimassa valmiina. Malli BlackBoardiin (muunneltu versio liitutaulumalleista (blackboard patterns) (Rogers 1997; Buschmann et al. 1996)) tulee Yacoubilta ja Ammarilta (2004). Työkaluina kaavion rakentamiseen käytetään Enterprise Architectin luokkakaavion rakennustyökaluja, kuten luokkaelementtiä luokkien ja niiden metodien tekemiseen ja aggregaattielementtiä Blackboardin ja DataHolderin välisen aggregaation tekemiseen. DataHolderista yleistetyt luokat ErrorData, MeasuredData ja FeedbackData yhdistetään yleistyssuhteella DataHolderiin (Generalize-elementtiä). Valmiin Blackboard-mallin kaavio tallennetaan UML-mallina (Diagram valikko->Save as UML pattern...). Kuvassa 25 Blackboard-malli on tehty omaan kaavioonsa, sillä UML-mallina tallennus koskee koko kaaviota, eikä täten voida tehdä suunnittelumallia suoraan yksityiskohtainen suunnittelumalli -tason kaavioon, josta sen olisi sitten voinut valita ja tallentaa.

Suunnittelumallin tallentamisessa suunnittelumallille annetaan nimi ja XML-tiedosto, johon suunnittelumalli tallennetaan. Lisäksi suunnittelumallille voidaan määritellä li-



Kuva 25: Kuva valmiista Blackboard-mallista.

sätietoja, kuten versionumero, lisähuomautuksia ja muokata mallin elementtien nimiä. Tärkein toiminto tallentamisen yhteydessä on kuitenkin suunnittelumallin elementtien toimintojen valitseminen myöhempää käyttöä varten. Valittavat toiminnot ovat:

- Luonti (Create): Luo suunnittelumallin elementin suoraan ilman muokkausta.
- Limitys (Merge): Limittää suunnittelumallin elementin olemassaolevan elementin kanssa, sallien olemassaolevan elementin ottaa valitun suunnittelumallin elementin rooli.
- Ilmentymä (Instance): Luo suunnittelumallin elementin olemassaolevan elementin ilmentymänä (valinta mahdollinen vain, jos suunnittelumallin elementti tukee tätä toimintoa).
- Tyyppi (Type): Luo suunnittelumallin elementin tyypit olemassaolevana elementtinä (valinta mahdollinen, jos suunnittelumallin elementti tukee tätä toimintoa).

Esimerkin tapauksessa Blackboardin elementeille valittiin toiminnoiksi luonti ja limitys. Ilmentymää eikä tyyppiä voitu valita, sillä Blackboardin elementit eivät tukeneet kyseisiä toimintoja.

Tallentamisen jälkeen suunnittelumalli voidaan tuoda resurssi-ikkunaan (Resources tab), josta sitä voidaan käyttää. Malli tuodaan nykyiseen UML-malliin valitsemalla projekti-ikkunasta resurssinäkymä ja valitsemalla UML-malli (UML Pattern) -

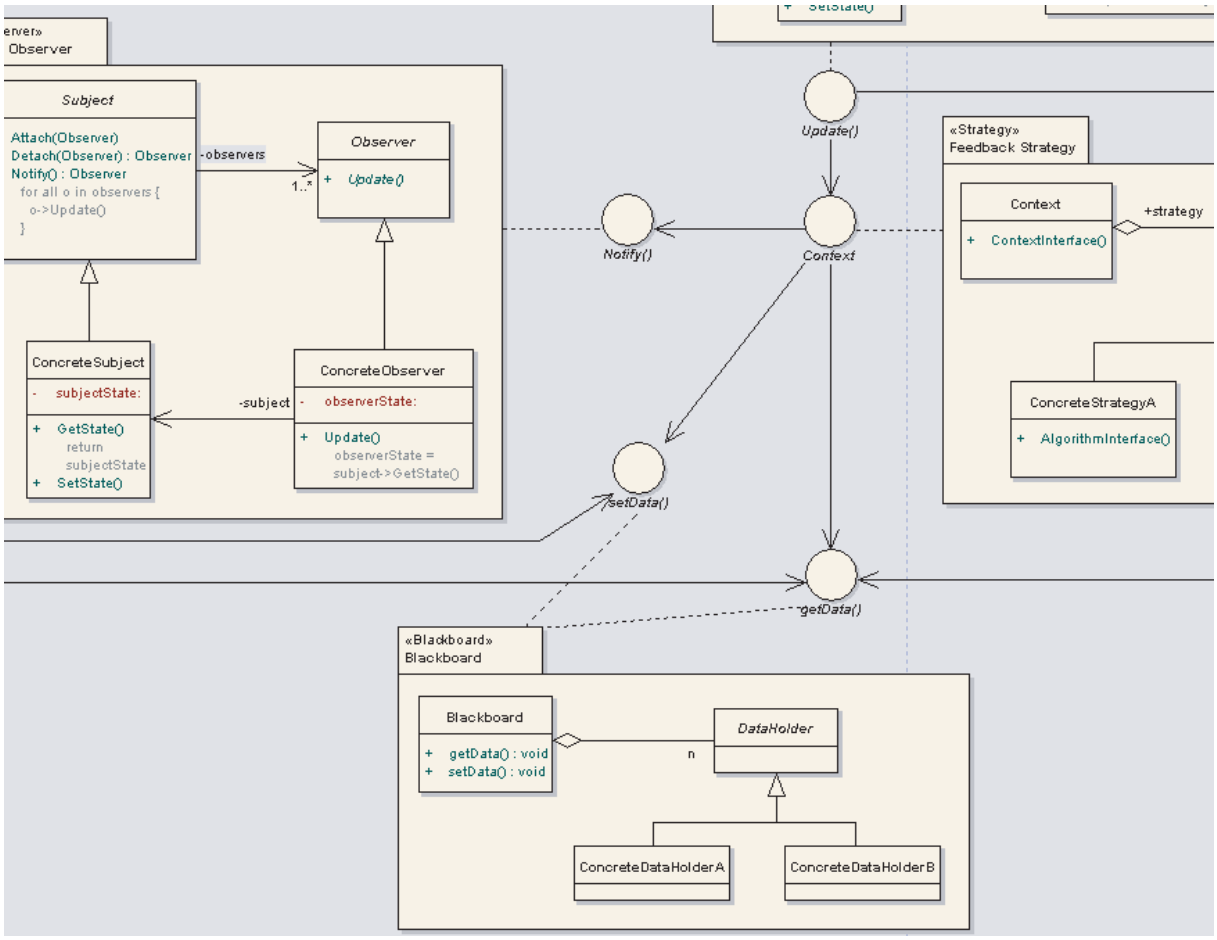
kohta hiiren oikealla painikkeella. Avautuvasta valikosta valitaan toiminto UML-mallin tuominen (Import UML Pattern), joka avaa tiedostoikkunan tuotavan suunnittelumallin XML-tiedoston hakemiseksi. Tuotu suunnittelumalli sijoittuu sille XML-tiedostossa määrättyyn kategoriaan. Jos kategoriaa ei ollut UML-mallissa valmiina, luodaan se sinne. Esimerkissä luodulle Blackboard-mallille oli tallennusvaiheessa määritelty tiedostonimeksi blackboard.xml, joten samanniminen tiedosto tulee valita tuotavaksikin.

Jotta resurssinäkymään tuotua suunnittelumallia voitaisiin käyttää, täytyy valittuna olla kaavio, johon suunnittelumalli halutaan lisätä. Esimerkissä rakennettavaan kehykseen on jo lisätty suunnittelumallit Observer ja Strategy, joiden lisäksi halutaan äsken luotu Blackboard-malli. Suunnittelumalli lisätään kaavioon avaamalla projektiselaimesta (Project Browser) resurssinäkymä ja valitsemalla haluttu suunnittelumalli UML Pattern -kansiosta hiiren oikealla painikkeella. Avautuvasta valikosta valitaan suunnittelumallin lisäystoiminto (Add Pattern to Diagram) tai vedetään ja tiputetaan (drag and drop) suunnittelumalli resurssinäkymästä kaavioon.

Lisäystoiminto avaa suunnittelumallin lisäysikkunan (Add Pattern dialog), jossa voidaan tarkastella suunnittelumallin tietoja, kuten sen kuvaus ja esikatselukuva kaaviosta, sekä tehdä lisämäärityksiä suunnittelumallin elementtien käytöstä kohdassa mallin elementit (Pattern Elements). Kyseinen kohta näyttää suunnittelumallin sisältämät elementit ja niille mahdolliset toiminnot (Create, Merge, Instance or Type) siten kuin ne suunnittelumallin tallennusvaiheessa on niille määritelty. Suunnittelumallin elementin nimeä voidaan muuttaa ja limitetyn elementin nimiavaruuden voi valita, mikäli niitä on aiemmin projektissa määritelty.

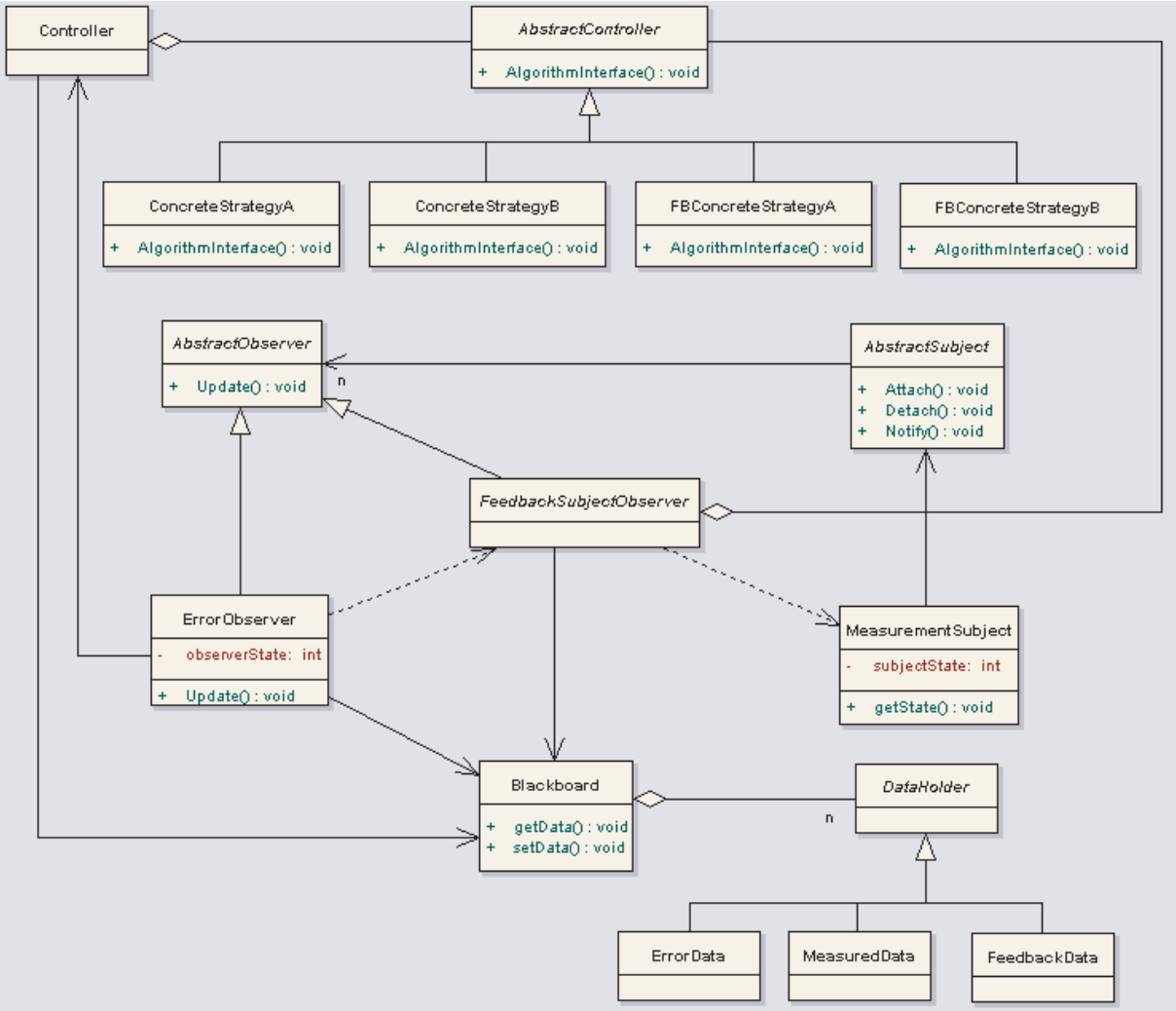
Kun halutut valinnat on tehty, painetaan OK-painiketta suunnittelumallin viemiseksi kaavioon. Esimerkin tapauksessa suunnittelumallit tuodaan yksityiskohtainen suunnittelumalli -tason kaavioon, jolloin mahdollisesti päällekkäisiä luokkia ei voida limittää, vaan niistä luodaan vain uudet ilmentymät. Kuva 26 esittää osaa yksityiskohtainen suunnittelumalli -tason kaaviosta, johon on lisätty esimerkissä aiemmin luotu Blackboard-malli.

POAD-prosessin kolmas ja viimeinen vaihe on suunnittelun hienosäätäminen, jolloin yksityiskohtainen suunnittelumalli -tason kaavioon lisätään sovellusaluekohtaisuus uudelleennimeämällä suunnittelumallien luokat ja niiden muuttujat ja metodit. Tämän jälkeen kaaviota kehitetään korvaamalla rajapintojen väliset yhteydet suoraan niitä



Kuva 26: Kuva suunnittelumallin lisäämisestä yksityiskohtainen suunnittelumalli -tason kaavioon.

vastaavien luokkien välisiksi yhteyksiksi käyttäen UML:n yhteyssuhdetta. Esimerkin tapauksessa nyt muodostettu hienosäädetty kaavio tallennetaan omana kaavionaan, jonka pohjalta voidaan tehdä uusi kaavio, jota voidaan hienosäätää edelleen kadottamatta näin aiemman vaiheen kaaviota. Edelleen tapahtuvaa hienosäätämistä voi olla kaavion monimutkaisuudesta riippuen tarvittava suunnittelun optimointi, kuten turhien luokkien poistaminen ja tehtäviltään ja vastuiltaan vastaavien luokkien yhdistäminen. Esimerkin tapauksessa esiintyy samoja abstrakteja luokkia useammin kuin kerran, joten niistä poistetaan ylimääräiset ja siirretään viittaukset jäljelle jätettyihin luokkiin. Lisäksi yhdistetään palautteen käsittelyyn läheisesti liittyviä luokkia yhdeksi abstraktiksi luokaksi FeedbackSubjectObserver. Lopputulos POAD-prosessista on esimerkin tapauksessa kuvan 27 kaltainen luokkakaavio.



Kuva 27: Kuva valmiin kehyksen lopullisesta kaaviosta.

Kokonaisuutena Enterprise Architect selviytyi hyvin POAD-prosessin mukaisesta

esimerkkikehyksen laatimisesta. Muutamia ongelmia tuli kuitenkin vastaan, kuten suunnittelumallien tallentaminen, mikä ei onnistu muuten kuin tallentamalla koko kaavio. Esimerkiksi suunnittelukaaviosta tiettyjen osien poimiminen ja tallentaminen suunnittelumalliksi ei onnistu, vaikka se voisikin olla monesti kätevä ominaisuus. Toinen ongelma koskee kaavion kopioimista pohjaksi seuraavan vaiheen kaaviolle. On oltava tarkkana muokatessaan uutta kaaviota, ettei poista elementtejä, jotka olivat jo kopioidussa kaaviossa, sillä poisto tapahtuu myös alkuperäisestä kaaviosta. Ongelman voi kiertää piilottamalla elementin poistamisen sijaan. Kummatkaan ongelmat eivät kuitenkaan ole varsinaisia ongelmia, mutta saattavat aiheuttaa turhaa työtä ja varomattomalle myös aiempien suunnitteluvaiheiden työn katoamista. Esimerkissä käytetty Enterprise Architectin versio 6.4 on suunnittelumalleille tarjoamansa tuen suhteen hyvin samanlainen kuin Borlandin vuoden 2005 Together-tuoteperheen mallinnustyökalut. Muuten ne ovat kuitenkin varsin erilaisia Togetherin tarjotessa toimintoja eri käyttäjille, suunnittelijalle, kehittäjälle ja arkkitehdille kohdistetusti omissa ohjelmistoissaan, kun Enterprise Architect sisältää kaikki työkalut samassa paketissa.

5 Yhteenveto

Suunnittelumallit kuvaavat yksinkertaisia ja elegantteja ratkaisuja tiettyihin oliopohjaisen ohjelmistosuunnittelun ongelmiin. Suunnittelumallien yksittäisen ja sattumanvaraisen käytön sijaan tulisi pyrkiä niiden tehokkaampaan hyödyntämiseen, systemaattiseen yhdistämiseen. Varsinkin laajat hajautetut ja sulautetut, reaaliaikaiset järjestelmät hyötyvät sovelluksen kehitystekniikoista, jotka tarjoavat systemaattisen, dokumentoidun prosessin suunnittelumalleja käyttäen.

Suunnittelumallien ja niiden yhdistelmien mallinnus ja esittäminen perustuu yleensä olio-ohjelmoinnin mallinnustekniikoihin, jotka käyttävät graafista kuvauskieltä, kuten UML. UML onkin nykyään tärkein jatkuvasti kehittyvä ohjelmistotuotannon mallinnuskieli ja sen uusimmat versiot tukevat suunnittelumallien esitystä ja yhdistämistä.

Suunnittelumallien systemaattiseksi yhdistämiseksi on kaksi lähtökohdiltaan erilaista lähestymistapaa: mallikielet, jotka kuvaavat tietyn sovellusalueen suunnittelumallit ja niiden suhteet, joita käyttäen voidaan ratkaista joukko suunnitteluongelmia ja sovellusalueriippumattomat kehitysprosessit, jotka määrittelevät yhdistämisessä käytettävän prosessin vaiheet ja tekniikat.

UML tarjoaa standardin tavan esittää suunnittelumallit ja tätä hyödyntävät monet sovelluskehitysympäristöt ja työkalut, jotka tukevat suunnittelumallien käyttöä ja yhdistämistä. Kehitys tällä saralla on kuitenkin vasta alussa ja suunnittelumallien käytössä on parantamisen varaa, kuten tutkielmassa tehdyssä esimerkkisuunnittelussa huomataan. Puutteistaan huolimatta työkalut tukevat ajatusta suunnittelumallien uudelleenkäytettävyydestä ja täten turhan suunnittelutyön unohtamisesta.

Suunnittelumallien systemaattiseksi yhdistämiseksi on paljon erilaisia lähestymistapoja ja jää nähtäväksi mitkä niistä tulevat jatkossa menestymään. UML:n ja kehitysympäristöjen tarjoama tuki saattavat tukea enemmän rakenteellista tapaa yleisessä käytössä, mutta sovellusalasta ja -kohteesta riippuen myös mallikielillä ja käytökselisillä tavoilla on käyttäjänsä. Ehkä näistä joskus yhdistetään yksi hybridikieli, joka hyödyntää eri tapojen parhaita puolia. Viimeisimmät tekniikat, kuten POAD ja subjektiohjelmoinnin yhdistelmämalli, edustavatkin jo omalla tavallaan tätä kehitystä.

Viitteet

Alexander, C., Inshikawa, S., Silverstein, M., Jacobson, M., Fiksdahlking, I., Angel, S. (1977) *A Pattern Language*. Oxford University Press, New York.

Anderson, E., Reenskaug, T. (1992) System Design by Composing Structures of Interacting Objects. *Proceedings of the 6th European Conference on Object Oriented Programming, ECOOP'92, Springer LNCS 615*, Utrecht, Netherlands, 133–152.

Borchers, J. O. (2000) A pattern approach to interaction design. *Proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques*, ACM Press, New York City, New York, United States, 369–378.

Bosch, J. (1998a) Design Patterns as Language Constructs. *Journal of Object Oriented Programming*. 11(2):18–22.

Bosch, J. (1998b) Specifying Frameworks and Design Patterns as Architecture Fragments. *Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS'98*. China.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture: A Pattern System*. Addison-Wesley, Boston.

Clarke, S., Harrison, W., Ossher, H., Tarr, P. (1999) Subject-Oriented Design. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application, OOPSLA, October 1999*, 34(10).

Clarke, S., Walker, R.J. (2001) Composition Patterns: An Approach to Designing Reusable Aspects. *Proceedings of the 23rd International Conference on Software Engineering, May 2001*.

Coplien, J. O. (1992) *Advanced C++ programming styles and idioms*. Addison-Wesley, Massachusetts.

Deutsch, L. P. (1989) Design reuse and frameworks in the Smalltalk-80 system. *Software Reusability, Volume II, Applications and Experience*, Addison-Wesley, MA, 57–71.

Dong, J. (2003) Representing the Applications and Compositions of Design Patterns in

UML. *Proceedings of the 2003 ACM symposium on Applied computing March 2003*. ACM Press, New York, 1092–1098.

Fincher, S. (2000) What is a Pattern Language. *Usability Pattern Language Workshop at Interact '99*.

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

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

Grand, M. (2002) *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Vol. 1*. Wiley, John & Sons, Inc., Canada.

Keller, R., Schauer, R. (1998) Design Components: Towards Software Composition at the Design Level. *Proceedings of 20th International Conference on Software Engineering, ICSE'98*. Kyoto, Japan, April 19-25, 302–311.

Kerievsky, J. (2005) *Refactoring of Patterns*. Addison Wesley, Boston.

Larman, G. (2002) *Applying UML and patterns: an introduction to object-oriented analysis and design and the Unified Process*. Prentice Hall PTR, New York.

Larsen, G. (1999) Designing Component-Based Frameworks using Patterns in the UML. *Communications of the ACM, October 1999*. 42(10):38–45.

OMG (2004) OMG UML Profile for Patterns Specification. Version 1.0, formal/04-02-04, Object Management Group (Saatavana myös: http://www.omg. org/cgi-bin/apps/doc?formal/04-02-04.pdf, 21.5.2006).

OMG (2005) OMG Unified Modeling Language Specification. Version 1.4.2, formal/05-04-01, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?formal/05-04-01.pdf, 14.5.2007).

OMG (2005b) OCL 2.0 Specification. ptc/2005-06-06, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?ptc/05-06-06.pdf, 14.5.2007).

OMG (2007) OMG Unified Modeling Language: Superstructure. Version 2.1.1, formal/07-02-03, Object Management Group (Saatavana myös:

http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf, 14.5.2007).

Reenskaug, T. (1996) *Working with Objects: The OOram Software Engineering Method*. Manning Publishing Co.

Reenskaug, T. (2000) *Modeling System Behavior: The What, the Why and the How of the UML Collaboration*. `http://heim.ifi.uio.no/~trygver/2000/UMLCollaboration/collab.html` (7.9.2006).

Riehle, D. (1997) Composite Design Patterns. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'97*. Atlanta Georgia USA, 218–228.

Salingaros, N. (2000) The Structure of Pattern Languages. *Architectural Research Quarterly*. Cambridge University Press, 4: 149–161.

Sametinger, J., Keller, R. (2003) Design Composition. *Journal of Computer Science & Technology*. 3(1): 27–33.

Sihman, M., Katz, S. (2003) Model Checking Applications of Aspects and Superimpositions. *Foundations of Aspect-Oriented Languages Workshop at AOSD 2003*. (toim. Leavens, G. T., Clifton, C.) Northeastern University, Boston, Massachusetts, 51-60.

Sun Microsystems (2006) Java Technology. WWW-sivusto, http://java.sun.com/ (21.5.2006).

Todd, E., Kemp, E., Phillips, C. (2004) What makes a good User Interface pattern language? *Proceedings of the fifth conference on Australasian user interface*. Dunedin, New Zealand, 91 - 100.

Wampler, B. E. (2002) *The Essence of Object-Oriented Programming with Java and UML*. Addison Wesley, Boston.

Wirfs-Brock, R., Wilkerson, B. (1989) Object-Oriented Design : A Responsibility-Driven Approach. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'89*. 71–75.

Yacoub, S. M., Ammar, H. H. (2004) *POAD Pattern-Oriented Analysis and Design*. Addison Wesley, Boston.

Liite 1: Rekursiokooste

Kohdat 1-4 kuvaavat suunnittelumallin yleisellä tasolla (Gamma & al., 1995). Yleisen tason kuvauksen lisäksi esitetään suunnittelumallien yhdistämiseen olennaisesti liittyvä läheiset mallit -kohta.

1. *Rekursiokooste-malli*

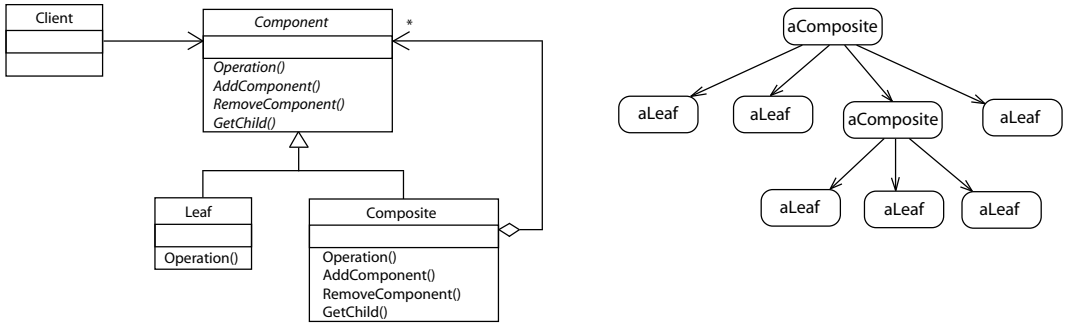
Suunnittelumalli esittää oliot rekursiivisesti koostettuna puurakenteena (part-whole hierarchy). Yksittäisiä olioita ja oliokoosteita voidaan käsitellä samalla tavalla.

2. *Ongelma*

Käytä suunnittelumallia, kun haluat esittää hierarkisia rekursiivisia oliorakenteita tai kun sovelluksessa ei haluta erotella primitiiviolioiden ja koosteolioiden käsittelytapaa, vaan sovellus haluaa käsitellä kaikkia rekursiokoosteen osia samalla tavalla.

3. *Ratkaisu*

Rakenne



Kuva 28: Esimerkki (a) Rekursiokoosteen kuvauksesta UML:n mukaisesti ja (b) suunnittelumallin tyypillinen oliorakenne (Gamma & al., 1995).

Osallistujat

- Component
  - määrittelee hierarkkisen rakenteen kaikkien solmujen yhteisen rajapinnan.
  - antaa yhteisille operaatioille oletustoteutuksen (jos mahdollista).
  - määrittelee operaatiot, joilla päästään käsiksi lapsisolmuihin.

  - (tarvittaessa) määrittelee operaation, jolla päästään käsiksi vanhempisolmuun ja toteuttaa sen mahdollisuuksien mukaan.

- Leaf
  - kuvaa koosterakenteen primitiiviolion eli lehden (primitiivioliolla ei ole lapsia).
  - toteuttaa primitiiviolion operaatiot.

- Composite
  - kuvaa koosteolion (koosteoliolla on lapsia).
  - tallentaa viitteet lapsisolmuihin.
  - toteuttaa Component-rajapinnan lapsisolmuja koskevat operaatiot.

- Client
  - käsittelee koosterakenteen olioita vain Component-luokan rajapinnan kautta.

4. *Seuraukset*

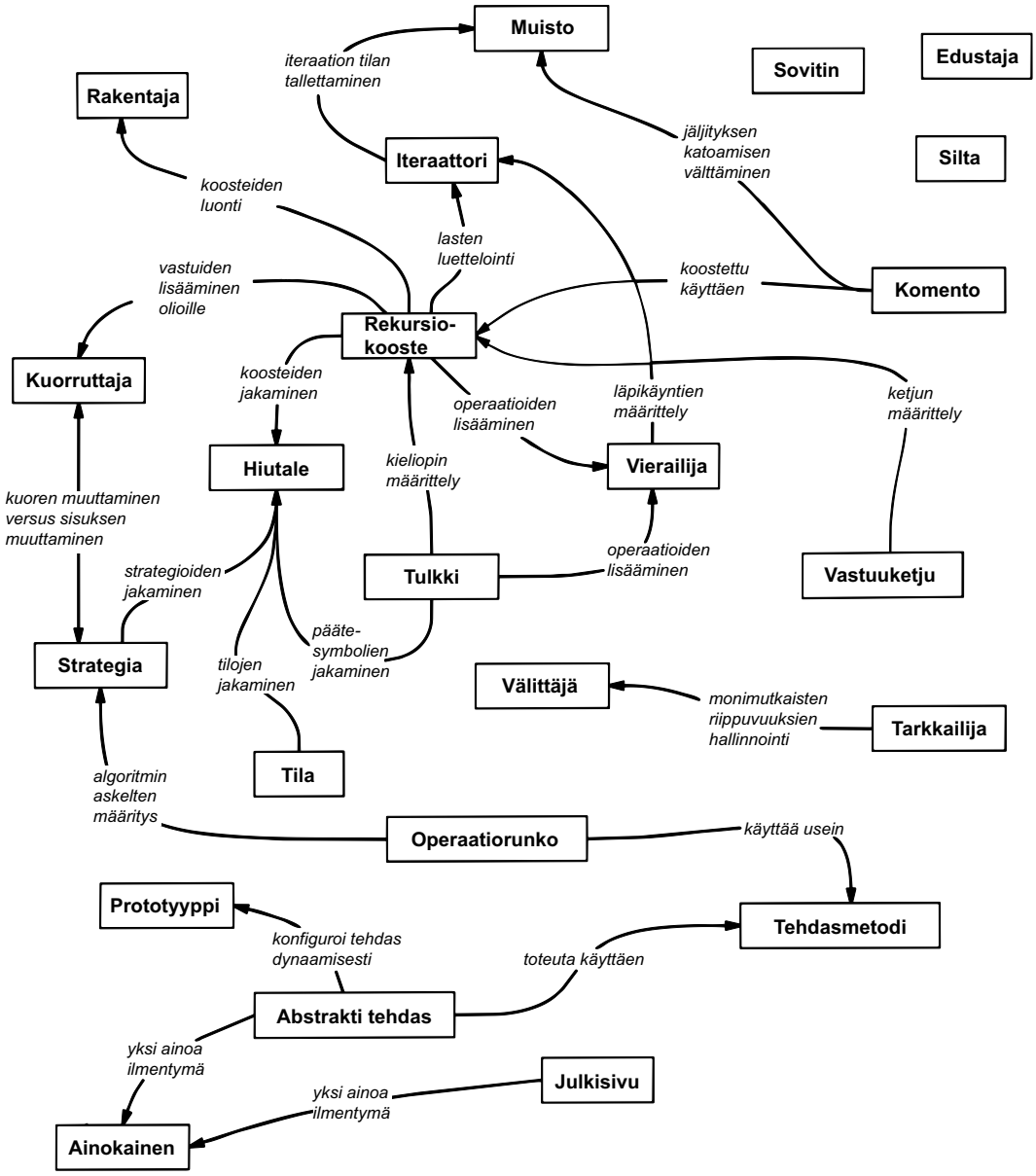
- Rekursiokooste
  - määrittelee luokkahierarkian, joka muodostuu koosteolioista ja primitiivioliosta. Primitiivioliota voidaan koota yhteen koosteolioiksi, joita voidaan edelleen koota yhteen rekursiivisesti. Jos sovellus osaa käsitellä primitiivioliota, se osaa käsitellä myös koosteoliota.
  - yksinkertaistaa asiakasta. Asiakas voi käsitellä primitiivioliota ja koosteolioita samalla tavalla. Yleensä asiakas ei edes tiedä (eikä sen tarvitsekaan tietää), kummanlaista oliota se käsittelee. Asiakaskoodi yksinkertaistuu, sillä siihen ei tarvitse kirjoittaa case-lausekkeita koosteen muodostavien luokkien käsittelemiseksi.
  - tekee uudenlaisten olioiden lisäämisen helpoksi. Uudet Composite- ja Leaf-aliluokat toimivat olemassa olevassa sovelluksessa automaattisesti. Asiakkaita ei tarvitse muuttaa, jos lisätään uusi Component-luokka.

– saattaa tehdä oliorakenteesta liian yleisen. Kun uusien komponenttien lisääminen on tehty helpoksi, on haittapuolena se, että koosteen komponenteille on hankala asettaa rajoituksia. Jos koosteolio saa koostua vain tietyntyyppisistä olioista, ei voida luottaa siihen, että järjestelmän tyyppitarkastukset valvoisivat rajoitusta, vaan sitä on valvottava ohjelmallisesti ajonaikana.

Läheiset suunnittelumallit

- Vastuuketjuna käytetään usein solmun ja sen vanhemman välistä linkkiä.
- Kuorruttajaa käytetään usein Rekursiokoosteen kanssa. Jos kuorruttajia ja koosteita käytetään yhdessä, niillä on yleensä yhteinen yliluokka. Kuorruttajien on siis sisällettävä Component-rajapinnan operaatiot Add, Remove ja GetChild.
- Hiutale mahdollistaa solmujen yhteiskäytön, mutta silloin ne eivät enää voi viitata vanhempaansa.
- Iteraattoria (Iterator) voidaan käyttää koosteiden läpikäymiseen.
- Vierailija kokoaa operaatiot ja käyttäytymisen, joka muuten olisi hajallaan Composite- ja Leaf-luokissa.

Liite 2: Suunnittelumallien suhdekartta



Kuva 29: GoF-kirjassa esitettyjen suunnittelumallien suhdekartta (Gamma & al., 1995).