Arkkitehtuuriratkaisujen uudelleenkäyttö olioperustaisessa ohjelmistokehityksessä

Janne Kauppila

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu –tutkielma
Toukokuu 2005

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

Tietojenkäsittelyoppi

KAUPPILA, JANNE: Arkkitehtuuriratkaisujen uudelleenkäyttö olioperustaisessa ohjelmistokehityksessä

Pro gradu –tutkielma, 57 sivua

Toukokuu 2005

Tässä tutkielmassa tarkastellaan suunnittelun uudelleenkäyttöä olioperustaisessa ohjelmistokehityksessä. Olio-ohjelmoinnin yhtenä suurimmista hyödyistä pidetään sen mukanaan tuomaa uudelleenkäytettävyyttä. Pelkkä oliomenetelmien käyttö ei kuitenkaan johda ilman muuta uudelleenkäytettävään koodiin [Koskimies 2000]. Oliotekniikassa uudelleenkäytettävyys perustuu osittain uudelleenkäytettävien komponenttien abstraktiotason nousuun koodista luokan tasolle. Painotettaessa suunnittelua keskeisenä uudelleenkäytettävänä elementtinä, voidaan abstraktiotasoa edelleen nostaa luokan tasolta. Tunnetuimpia olioperustaisen suunnittelun uudelleenkäyttöä tukevia mekanismeja ovat suunnittelumallit (design patterns) ja sovelluskehykset (application frameworks). Oliosuunnittelun uudelleenkäyttö toteutuu siis hyväksi havaittuja ohjelmistoarkkitehtuuriratkaisuja uudelleenkäyttämällä (suunnittelumallit) tai kokonaista valmista arkkitehtuuria erikoistamalla (sovelluskehykset). Näistä ratkaisutyypeistä suunnittelumallit ovat tässä työssä suuremmassa osassa kuin sovelluskehykset.

Sisällys

1. Johdanto

Olen aloittanut tämän tutkielman kirjoittamisen vuonna 1995, jolloin suunnittelumalleja yleisesti pidettiin alan viimeisimpänä ”kuumana juttuna”. Tietojenkäsittelytiede kohtaa säännöllisin väliajoin uusia konsepteja, joiden ympärille nousee valtava kohu ja usein myös yliarvostus. Uusien paradigmojen autuaaksitekevyyteen uskotaan toisaalla vahvasti, toisaalla taas mikä tahansa kohuttu tuomitaan herkästi vain tämän vuoden uutena vouhotuksena, pelkkänä ”hypenä”. Mitä tapahtuikaan virtuaalitodellisuudelle? Missä ovat elektroniset kirjat? Miksei puheentunnistus vieläkään toimi? Nämä karrikoidut esimerkit ovat ehkä kaukana tietojenkäsittelytieteen ytimestä, mutta tieteellisesti merkittävämpienkin aiheiden ympärillä on ollut runsaasti ylimääräistä kiehuntaa. Toisaalta myös kaikki viime vuosien kiistatta merkittävimmät uudistukset ohjelmistokehityksessä (esimerkiksi olioperustaisuus, UML, Java) ovat olleet vuorollaan alan kohuttuja kuumia aiheita.

Tietojenkäsittelytiede on tieteenalana vastasyntynyt suhteutettuna esimerkiksi filosofiaan tai matematiikkaan. Alan tutkimusta on toki tehty jo kymmeniä vuosia, mutta tieteenalana tietojenkäsittelytiede on varsinaisesti kehittynyt vain hieman edellä tietokoneen kehitystä. Valtaosa edelleen relevantista tutkimuksesta on tehty alkaen vasta 1970-luvulla. Esimerkiksi filosofian historian kohdalla puhutaan herkemmin sataluvuista tai jopa tuhatluvuista kuin kymmenluvuista. Erityisesti teollisen ohjelmistokehityksen kohdalla suurin osa keskeisestä käsitteistöstä, käytettävistä menetelmistä ja menestyneimmistä paradigmoista on peräisin 1980- tai 1990-luvuilta. Kuten alalla toimivat ihmiset yleisestikin, luonnollisesti myös koko tiedeyhteisö on edelleenkin suhteellisen nuorta. Tätä hyvin nuorta historiaa vasten tarkasteltuna onkin ehkä helppo ymmärtää uusista asioista innostuminen ja kenties myös jonkinlainen perspektiivin puuttuminen.

Tutkielmani teko jäi kesken aloitettuani työt vuonna 1997. Jatkaessani keskeneräistä tutkielmaa nyt lähes kahdeksan vuoden tauon jälkeen, minulla on mahdollisuus käyttää sekä aikaperspektiiviä että työssäni saamaani kokemusta arvioimaan uudelleen tämän tutkielmani johtoajatusta suunnittelun uudelleenkäytöstä sekä suunnittelumallien ja sovelluskehysten roolista siinä.

Suurimpia tekijöitä olioperustaisuuden läpilyönnille ohjelmistokehityksessä on ohjelmistokomponenttien parempi uudelleenkäytettävyys. Uudelleenkäytettävyys kulkee käsi kädessä sekä ylläpidettävyyden että ohjelmiston ymmärrettävyyden kanssa. Näiden arvojen arvioidaan johtavan laadukkaampiin ohjelmistoihin sekä pienentävän tuotanto- ja ylläpitokustannuksia. Askel koodin uudelleenkäytöstä suunnittelun uudelleenkäyttöön on uusi askel samaan suuntaan, johon olioperustaisuuden mukanaan tuoma uudelleenkäytettävyys on johtanut teollista ohjelmistokehitystä.

Tässä tutkielmassa tarkastellaan ajatusta suunnittelun uudelleenkäytöstä sekä esitellään tunnetuimpia suunnittelun uudelleenkäyttöä tukevia mekanismeja: suunnittelumalleja (engl. design patterns) ja sovelluskehyksiä (engl. application frameworks).

Luku 2 johdattaa olioperustaisuuteen ja sen peruskäsitteistöön. Luku 3 esittelee varsinaista ohjelmistokehitystä, määrittelee ohjelmiston laadun kriteereitä ohjelmistokehityksen tavoitteina ja mittareina. Olioperustaista ohjelmistokehitystä kuvataan hieman tarkemmin ja valotetaan niitä syitä, miksi nykyaikaisessa ohjelmistokehityksessä ollaan siirrytty niin voimakkaasti olioperustaisuuteen. Syistä keskeisin, uudelleenkäyttö, esitellään tarkemmin luvussa 4. Uudelleenkäytössä keskitytään erityisesti tämän tutkielman keskeiseen ajatukseen, suunnittelun uudelleenkäyttöön ohjelmakoodin uudelleenkäytön sijasta. Seuraavissa luvuissa esitellään olioperustaisen ohjelmistokehityksen tunnetuimmat suunnittelun uudelleenkäyttöä tukevat mekanismit: suunnittelumallit luvussa 5 ja

sovelluskehykset luvussa 6. Luvussa 7 luodaan katsaus suunnittelumallien ja sovelluskehysten nykypäivään ja arvioidaan, ovatko ne todella murtautuneet osaksi nykyaikaista ohjelmistokehitystä. Luku 8 sisältää yhteenvedon.

2. Olioperustaisuudesta

Tämä luku sisältää lyhyen johdannon olioperustaisuuteen kohdassa 2.1 sekä olioparadigman keskeiseen peruskäsitteistöön kohdassa 2.2 alakohtineen. Sekä olioperustaisuuden että keskeisen käsitteistön esittelyssä olen yrittänyt pitää taustalla tämän työn keskeisen lähtökohdan, arkkitehtuuriratkaisujen uudelleenkäytön.

2.1 Johdatus olioperustaisuuteen

Olioperustaisella ohjelmistokehityksellä tarkoitetaan ohjelmistojen rakentamistapaa, joka perustuu ohjelmistojen kuvaamiseen keskenään kommunikoivina olioina [Koskimies 2000]. Olioperustaisuudelle olennaista on sovellusalueen käsitteellinen mallintaminen, jonka seurauksena ohjelmiston käsitteistö vastaa sovellusalueen käsitteistöä. Sovellusalueen osat kuvataan olioina ja niiden välinen vuorovaikutus kuvataan palveluina, joita oliot tarjoavat toisilleen. Syntyneellä mallilla kuvataan siis sekä sovellusalueen rakenne että toiminta.

Tämä käsitteellinen vastaavuus sovellusalueen kanssa helpottaa ohjelman ymmärtämistä, edellyttäen tietenkin että sovellusalueen käsitteistö on helposti ymmärrettävää. Ei ole itsestään selvää, että sovellusalue koostuu konkreettisista käsitteistä, jotka ovat suoraviivaisesti mallinnettavissa. Kun ajattelen puhelinohjelmiston arkkitehtuuria, jonka kanssa työskentelen päivittäin, minulle ei tule mieleen kovinkaan konkreettisia reaalimaailman vastineita tämän ohjelmiston keskeisille ohjelmistokomponenteille.

Käsitteellisestä vastaavuudesta seuraava helpompi ymmärrettävyys johtaa niihin varsinaisiin hyötyihin, joiden seurauksena olioparadigman voidaan katsoa tehneen lopullisen läpimurtonsa systemaattiseen ohjelmistokehitykseen. Helpommin ymmärrettävä ohjelmisto on helpommin

ylläpidettävää. Ylläpitokustannukset ovat merkittävä osa ohjelmistokehityksen kokonaiskustannuksia. *Uudelleenkäytettävyys* on toinen merkittävä hyöty, joka tyypillisesti lasketaan olioperustaisuuden mukanaan tuomaksi. Uudelleenkäytettävyyden voidaan ajatella olevan seurausta juuri ohjelmiston paremmasta ymmärrettävyydestä, mutta myös ohjelmiston *jatkuvuudesta*: pienten muutosvaatimusten vaikutukset ohjelmistoon ovat myös pieniä [Koskimies 2000]. Tämä on suoraa seurausta ohjelmiston käsitteellisestä vastaavuudesta sovellusalueen käsitteistön kanssa.

2.2 Olio-ohjelmoinnin peruskäsitteet

2.2.1 Olio

[Rumbaugh et al. 1991] määrittelevät *olion* "käsitteeksi, abstraktioksi tai asiaksi jolla on selkeät rajat ja jokin merkitys käsiteltävän ongelman suhteen." Tämä on kohtalaisen väljä määritelmä, mutta se esittelee meille sopivan pintapuolisesti olio-ohjelmoinnin perusyksikön. Olio on se perusyksikkö, joihin sovellusalueen käsitteellinen malli perustuu. [Koskimies 2000] määrittelee olion ympäristöstään erottuvaksi kokonaisuudeksi: sillä on oma identiteettinsä, sisäinen rakenteensa sekä suhteet tiettyyn ympäristöön. Olioilla on [Rumbaugh et al. 1991]:n mukaan kaksi tarkoitusta: ne auttavat reaalimaailman ymmärtämistä ja tarjoavat käytännöllisen pohjan tietokonetoteutukselle.

Oliolla on tila, käyttäytyminen ja identiteetti; samanlaisten olioiden rakenteen ja käyttäytymisen määrittää niiden yhteinen luokka; termeillä (luokan) ilmentymä ja olio on sama merkitys. [Booch 1991]

Oliolla on seuraavat oleelliset ominaisuudet [Koskimies 2000]:

- Olio pystyy pyydettäessä suorittamaan tietyt tälle oliolle ominaiset toiminnot. Kutsumme näitä toimintoja *operaatioiksi*. Kullakin

operaatiolla on nimi, mahdollisesti parametreja, sekä toiminnan määrittelevä runko. Operaatio voi olla arvon palauttava funktio.

- Olio pystyy tallettamaan tietoa olion *attribuutteihin*, nimettyihin tietokenttiin. Attibuuttien arvojen yhdistelmiä kutsutaan olion *tiloiksi*: operaation suoritus voi muuttaa olion tilaa. Kutsumme olion attribuutteja ja operaatioita yhteisesti olion *piirteiksi*.

- Jokaisella oliolla on sen identifioiva tunniste, *viite*. Kahdella eri oliolla on eri viite, vaikka ne olisivat toistensa kopioita.

- Olio on *suojattu* kokonaisuus, jonka käyttö on rajattu tiettyihin muotoihin. Yleensä vain tietyt olion piirteet ovat käytettävissä olion ulkopuolelta.

[Shlaer 1988] määrittelee olion reaalimaailman tietyn käsitejoukon abstraktioksi siten, että kaikilla joukkoon kuuluvilla käsitteillä (ilmentymillä) on samat ominaisuudet ja nämä ilmentymät ovat säännönmukaisia ja näistä säännöistä riippuvaisia.

Olion suojauksella tarkoitetaan sellaisten yksityiskohtien kätkemistä, jotka eivät kuulu olion peruspiirteisiin. [Booch 1991] Oliolle on määritelty tietyt piirteet, joihin muut oliot pääsevät käsiksi, mutta suunnittelussa on olennaista suojata osa piirteistä, joita on järkevintä käsitellä vain olion sisäisesti.

Konkreettisena esimerkkinä edellä määritellyistä olioiden ominaisuuksista voidaan tarkastella oliota, joka toteuttaa graafisen käyttöliittymän painikkeen. Attribuuteista voisi olla esimerkkinä painikkeen sisältämä teksti, painikkeen väri ja muoto. Painike-olion operaatio voisi olla esimerkiksi `piirrä(x,y)`, joka piirtää painikkeen koordinaatteihin x,y. Nämä ovat siis kyseisen olion piirteitä. Operaation `piirrä(x,y)` kutsu muuttaa olion tilaa piirtämällä sen eri kohtaan. Kun luodaan käyttöliittymään ikkuna, jossa on ”OK”- ja ”Cancel”-painikkeet, niillä on omat viitteensä ikkuna-oliossa. Painike-olion piirteistä voidaan suojata esimerkiksi teksti-attribuutin arvojen mielivaltainen

asettaminen määrittelemällä oliolle operaatio, joka asettaa attribuutin arvon sovitun rajapinnan mukaisesti. Itse teksti-attribuutti ei kuitenkaan ole suoraan käytettävissä olion ulkopuolelta, se on siis suojattu. Tekstiä voidaan muuttaa ainoastaan käyttämällä asetusfunktiota.

2.2.2 Luokka

Luokka määrää olion rakenteen ja käyttäytymisen; oliot ovat luokkien ilmentymiä. [Meyer 1988] korostaa luokan ja olion eroa seuraavasti: Oliot ovat ajonaikaisia elementtejä, jotka luodaan ohjelman käynnistyessä; luokat ovat puhtaasti staattinen kuvaus mahdollisten olioiden joukosta — luokkien ilmentymistä.

Luokka voidaan rinnastaa abstraktin tietotyypin käsitteeseen; olio sen ilmentymään. Abstrakti tietotyyppi on sellainen tyyppi, joka määrittelee käyttäjilleen vain ilmentymiinsä sovellettavat operaatiot (kutsumuodot ja merkitys) mutta ei näiden toteutustapaa eikä tarvittavia tietorakenteita [Koskimies 2000]. Tämä toteutetaan juuri suojauksen avulla; esimerkiksi vain luokan operaatiot voivat olla käytettävissä luokan ulkopuolelta, attribuutit taas ovat suojattuja.

Luokka määrittelee siis olion. Esimerkkinä voimme ajatella ihmisluokkaa, jonka ilmentymiä kaikki olemme. Kaikilla ihmisluokan olioilla ei luonnollisestikaan ole samoja piirteitä, mutta jos luokka on riittävän karkealla tasolla määritelty, sen voidaan ajatella sisältävän kaikkien ihmisolioiden yhteiset ominaisuudet. Tällainen yhteinen attribuutti ihmisolioille on esimerkiksi ikä. Ihmisluokka sisältää ikä-attribuutin määrittelyn; ihmisolioilla on ikä-attribuutti.

Luokalla voi myös olla operaatioita, joiden toteutusta ei ole annettu. Tällaista luokkaa kutsutaan *abstraktiksi luokaksi*, eikä siitä voi suoraan luoda oliota. Toteuttamattomia operaatioita kutsutaan avoimiksi virtuaalioperaatioiksi. Näiden toteutus annetaan abstraktin luokan konkreettisissa aliluokissa, joista

puolestaan voidaan luoda olioita. Abstraktin luokan käsite on hyödyllinen täsmällistä luokkahierarkiaa muodostettaessa. Oikeammin voidaan myös ajatella edellisen kappaleen ihmisluokan olevan abstrakti luokka, jolla on konkreettiset aliluokat mies ja nainen. Ihmisluokalle voidaan määritellä sukupuolen palauttava funktio ja määritellä se vasta aliluokissa.

Mitä eroa luokalla ja luokan oliolla on? Luokka on tavallaan kuin piparkakkumuotti, olio on sillä tehty piparkakku.

2.2.3 Periytyminen

Periytyminen (inheritance) tarkoittaa luokkien välistä suhdetta, jonka avulla voidaan toteuttaa olemassa olevaan luokkaan (tai *moniperiytymisen* kohdalla useampaan luokkaan) perustuva uusi luokka. Kun luokka B perii luokan A, sisältää luokka B kaikki samat attribuutit ja operaatiot kuin luokka A. Luokkaa A sanotaan tässä tapauksessa B:n *yliluokaksi* ja B:tä vastaavasti A:n *aliluokaksi*. Luokka B voi muuttaa perimäänsä käyttäytymistä tarvitsematta kopioida luokan A koodia tai kajoamatta luokkaan A.

Ensinnäkin luokka voi muuttaa perimiensä operaatioiden toteutusta. Tätä kutsutaan operaation *uudelleenmäärittelyksi* (overriding). Operaation kutsurajapintaa (operaation nimi ja parametrit) luokka ei kuitenkaan voi muuttaa. Toiseksi luokka voi määritellä omia attribuutteja ja operaatioita perimiensä piirteiden lisäksi.

Kun luokkaa uudelleenkäytetään, se usein ei istu sellaisenaan uuteen ympäristöön, vaan tarvitsee muutoksia. Periytymisen avulla luokkaa voidaan laajentaa ja sovittaa uusia tarpeita vastaavaksi. Periytyminen mahdollistaa näin ollen yksittäisten ohjelmistokomponenttien uudelleenkäytön.

Sivuhuomautuksena todettakoon tosin, että komponentin laajentaminen periytymisen avulla on usein hieman ongelmallista ja saattaa johtaa ohjelmiston käsitemallin sirpaloitumiseen ja tehdä siitä epäintuitiivisen. Käytännössä olen törmännyt muutaman kerran luokkiin, jotka on nimetty

`LuokanNimiV2`:ksi, joka jo kielii siitä, ettei käsitteellinen vastaavuus sovellusalueen kanssa voi enää olla täysin kohdallaan. Tällaiseen menettelyyn joudutaan usein pragmaattisista syistä, jos ei koko käsitemallin huolelliseen uusimiseen ole resursseja.

Olioparadigman useista ohjelmiston laatuun positiivisesti vaikuttavista tekijöistä periytyminen kohentaa siis ainakin ohjelmiston uudelleenkäytettävyyttä ja toisaalta laajennettavuutta. Periytymismekanismi kannustaa etsimään luokista yhteisiä piirteitä ja toteuttamaan nämä piirteet abstraktina yliluokkana. Abstrakti yliluokka taas helpottaa sellaistenkin uusien piirteiden lisäystä ohjelmistoon, joita ohjelmoija ei ole tullut alunperin edes ajatelleeksi. Nämä periytymismekanismin positiiviset vaikutukset luokkahierarkiaan parantavat myös ohjelmiston ymmärrettävyyttä. Uudet piirteet voidaan parhaassa tapauksessa tuoda huolellisesti suunniteltuun luokkahierarkiaan uusina aliluokkina.

Käsitteellisesti periytyminen vastaa erikoistamisen käsitettä: aliluokka on yliluokan erikoistapaus. Abstraktin yliluokan muodostaminen taas on yleistämistä. Edellisen kohdan ihmisluokka-esimerkkiä jatkaen, ihmisluokan aliluokkia ovat miesluokka ja naisluokka.

Periytymistä on periaatteessa kahta lajia: yksittäisperiytymistä ja moniperiytymistä. Yksittäisperiytymisellä tarkoitetaan sitä periytymistä, jota tässä kohdassa on esitelty: luokalla on vain yksi yliluokka. Moniperiytymisessä luokka perii useamman kuin yhden yliluokan piirteet. Tämän on todettu mutkistavan ohjelmiston käsitemallia – sitä ei voida enää kuvata puuna, vaan se täytyy kuvata verkkona. Muita moniperiytymisen ongelmia ovat nimikonfliktit (samannimisen operaation periminen eri yliluokilta) ja toistuva periytyminen (saman luokan periminen kahta eri kautta) [Koskimies 2000]. Näistä syistä koko moniperiytymisen käytöstä on päätetty luopua nykyisen työnantajani ohjelmointikäytännöissä.

2.2.4 Monimuotoisuus ja myöhäinen sidonta

Monimuotoisuus tarkoittaa käytännössä sitä, että viitemuuttujan sisältö saattaa vaihdella. Monimuotoisella viitteellä on sekä staattinen että dynaaminen tyyppi. Viitteen staattinen tyyppi on annettu ohjelmakoodissa ja tästä staattisesta tyypistä voidaan johtaa viitteen mahdollisten dynaamisten tyyppien joukko, johon kuuluvat kaikki staattisen tyypin jälkeläistyypit. Viitteen dynaaminen tyyppi saattaa vaihtua ajoaikana.

Myöhäisellä sidonnalla tarkoitetaan kutsuttavan operaation määräytymistä vasta ajoaikana. Suoritettavan operaation koodi määräytyy ajoaikana ja riippuu vastaanottajan dynaamisesta tyypistä.

Periytyminen yhdessä myöhäisen sidonnan kanssa mahdollistaa myös uudelleenkäytettävät sovelluskehykset [Pree 1994], joihin palataan myöhemmin.

Edelleen ihmisluokka-esimerkkiä soveltaen, voidaan määritellä muuttuja, jonka staattinen tyyppi on ihminen. Dynaamisesti muuttujan arvoksi voidaan kuitenkin asettaa miestyypin olio. Tällöin viitemuuttujan dynaaminen tyyppi on eri kuin sen staattinen tyyppi. Esimerkki myöhäisestä sidonnasta samassa esimerkissä on ihmisluokan operaatio `pukeudu()`, joka on uudelleenmääritelty mies-aliluokassa. Kutsuttaessa ihmistyyppiä olevan viitemuuttujan `pukeudu`-operaatiota, suoritetaankin ajoaikana dynaamisen tyypin mies ylimääritelty versio operaatiosta.

3. Ohjelmistokehitys

Ohjelmistokehitys on vaikeaa. Suurten ohjelmistojen kehityksen vaikeutta kutsuttiin jo 1970-luvulla ohjelmistokriisiksi. Ohjelmistotekniikan ongelmia on yritetty ratkoa siis jo kymmeniä vuosia, mutta samaan aikaan ohjelmistojen kompleksisuus on kasvanut paljon nopeammin. Ohjelmistojen koon on todettu kaksinkertaistuvan muutaman vuoden välein, kun ohjelmistotyön tuottavuuden kasvu on ollut vuosittain vain noin neljän prosentin luokkaa [Haikala et al. 2000].

Ohjelmistokehityksen tavoitteiksi ja mittareiksi on määritelty useita laadun kriteereitä, joiden avulla pyritään ymmärtämään mihin suuntaan ohjelmistokehitystä tulisi viedä ja mikä tekee ohjelmistosta laadukkaamman. Näitä kriteereitä esitellään kohdassa 3.1. Kohta 3.2 esittelee systemaattista ohjelmistokehitystä, joka on nähty välttämättömyytenä pyrittäessä hallitsemaan ohjelmistokehityksen mutkikasta ongelmakenttää. Olioperustainen ohjelmistokehitys, johon paneudutaan kohdassa 3.3, on vakiintunein systemaattisen ohjelmistokehityksen muoto.

3.1 Ohjelmiston laatu

Ohjelmiston laatua on tunnetusti hankala mitata. [Pree 1994] esittää ohjelmiston laadulle muutamia attribuutteja:

- *Oikeellisuus, korrektius*: ohjelmistotuote toteuttaa suunnitellut vaatimukset eikä sisällä virheitä. (Jos puhutaan korrektiudesta, täytyisi määritelmän mukaan tietysti sanoa, ettei ohjelmisto sisällä lainkaan virheitä, mutta käytännössä puhutaan kuitenkin siedettävästä määrästä virheitä tai ettei ohjelmisto sisällä kriittisiä virheitä.) Tämä on tietenkin tärkein, jopa pakollinen laatuvaatimus ohjelmistolle. Ohjelmiston testaustuloksien avulla voidaan arvioida ohjelmiston

oikeellisuuden astetta ja binäärinen korrektius voidaan määritellä vaikka ohjelmistotuotteen valmiutena kaupalliseen julkistukseen.

- *Helppokäyttöisyys*: ohjelmistotuotteen käyttäminen ja käytön oppiminen ovat helppoja. Tämä kriteeri varmaankin esiintyy useimmissa vaatimusmäärittelyissä, mutta lienee silti yksi vaikeimmin mitattavista. Pree tarkoittaa tässä yhteydessä tuotteen helppokäyttöisyyttä loppukäyttäjälle.

- *Uudelleenkäytettävyys*: mahdollisuus käyttää järjestelmää tai sen osia toisissa ohjelmistotuotteissa. Uudelleenkäytöllä voidaan saavuttaa merkittäviä säästöjä ohjelmistotuotannossa ja uudelleenkäytettäessä laadukkaiksi todettuja ohjelmistokomponentteja ohjelmiston laatu paranee. Tämän kohdan alle voidaan laskea myös ohjelmiston ylläpidettävyys, joka on tavallaan uudelleenkäytön muoto.

- *Luettavuus*: ohjelmistotuotteen ymmärtämiseksi tarvittava ponnistelu. Saman ohjelmiston parissa työskentelee tyypillisesti iso joukko suunnittelijoita ja mitä pidempi on tuotteen elinkaari, sitä useamman uuden suunnittelijan täytyy perehtyä ohjelmistoon. Ohjelmiston helppo luettavuus/ymmärrettävyys on tärkeä tekijä sen uudelleenkäytettävyydessä ja ylläpidettävyydessä.

- *Tehokkuus*: ohjelmisto- ja laitteistoresurssien optimaalinen käyttö. Resursseja on harvoin tuhlattavaksi asti, hyvänä esimerkkinä matkapuhelinteollisuus, jossa laitteiden on oltava kustannustehokkaita, eikä laitteistoon panosteta ylimääräistä, että hinta saadaan pidettyä kohtuullisena. Tämä laatuvaatimus on saattanut jäädä nykyään ehkä vähemmälle huomiolle laitteistoresurssien yleisen paranemisen myötä.

- *Siirrettävyys*: ohjelmiston muunnettavuus toisille laitteisto- tai ohjelmistoalustoille. Kaikille ohjelmistotuotteille siirrettävyys ei välttämättä ole kovinkaan tärkeää, mutta tavoiteltaessa mahdollisimman yleistä tai monikäyttöistä ohjelmistotuotetta, sen on

oltava helposti siirrettävä. Kokemukseni lisensoitavista matkapuhelinohjelmistoista osoittavat, että tämä on suuri haaste toteutettaessa ohjelmistokomponentteja, jotka ovat ohjelmistossa lähellä laitespesifisiä osia. Osa ohjelmistosta jätetään abstraktiksi ja lisensoija toteuttaa ohjelmiston konkreettiset, laitespesifiset osat.

[Meyer 1988] näkee keskeisinä laatuvaatimuksina myös seuraavat tekijät:

- *Tukevuus (robustness)*: ohjelmiston kyky toimia oikein myös epänormaaleissa tilanteissa. Tämä vaatimus on tavallaan jatke ohjelmiston korrektiusvaatimukselle; korrektius voidaan yleisemmin nähdä ohjelmiston kykynä toimia oikein kaikissa tilanteissa. Toiminnallisuuteen, joka esiintyy vain 20 %:ssa tapauksista käytetään helposti 80 % ohjelmointiajasta. Silti tyypillisesti ohjelmistovirheet löytyvät pääasiassa epänormaaleissa käyttötilanteissa.

- *Laajennettavuus:* ohjelmiston muunnettavuus muuttuneiden vaatimusten mukaiseksi. Laajennettavuus tai joustavuus on tärkeää erityisesti suurissa ohjelmistoprojekteissa. Laajennettavuus voidaan nähdä myös samana asiana kuin uudelleenkäytettävyys ja ylläpidettävyys.

- *Yhteensopivuus:* ohjelmistotuotteiden yhdisteltävyys toistensa kanssa. Standardirajapintojen käyttö ja yleisten ohjelmointikäytäntöjen noudattaminen tukee tätä laatukriteeriä.

[Meyer 1988] jaottelee kolme viimeksi mainittua sekä korrektiuden ja uudelleenkäytettävyyden laatutekijöiksi, joita voidaan parantaa nimenomaan olioperustaisella suunnittelulla.

Näiden kaikkien laatutekijöiden saavuttamiseksi sekä [Meyer 1988] että [Pree 1994] näkevät ohjelmiston arkkitehtuurin huolellisen suunnittelun, joka tuottaa erillisistä moduuleista koostuvan joustavan rakenteen. Täsmällisiin vaatimusmäärittelyihin perustuva systemaattinen ohjelmistokehitys [Meyer

1988] sekä ohjelmistokomponenttien uudelleenkäyttö [Pree 1994] nähdään myös avaintekijöinä laadukkaaseen ohjelmistotuotteeseen.

Puhuttaessa ohjelmistosta ja sen laadusta keskitytään usein ainoastaan kehitysvaiheeseen. Ohjelmistokustannuksista on kuitenkin arvioitu jopa 70 % kuluvan ylläpitoon. Edellä mainitut tekijät helpottavat olennaisesti myös ohjelmiston ylläpitoa.

Olioperustainen ohjelmistokehitys antaa eväät kaikkiin edellä mainittuihin tekniikoihin. Palaan tähän vielä myöhemmin kohdassa, joka käsittelee olioperustaista ohjelmistokehitystä.

3.2 Perinteinen ohjelmistokehitys

Systemaattinen ohjelmistokehitys on perinteisesti jaettu kolmeen vaiheeseen: analyysiin, suunnitteluun ja toteutukseen. [de Champeaux et al. 1992] määrittelevät nämä vaiheet karkeasti näin:

- *Analyysivaiheen* tehtävä on tuottaa täsmällinen kuvaus siitä, mitä ohjelmiston on tarkoitus tehdä. Tämä kuvaus voi toimia asiakkaan ja toimittajan välisen sopimuksen pohjana sekä suunnittelijan työn lähtökohtana.

- *Suunnitteluvaiheen* tehtävä on tuottaa kuvaus ohjelmistosta, joka täyttää analyysin vaatimukset. Tämä kuvaus puolestaan toimii ohjelmoijan työn lähtökohtana.

- *Toteutusvaiheen* tehtävä on tuottaa suunnittelua vastaava ilmaisu kohdeympäristöön; so. ohjelmia tietyillä ohjelmointikielillä, tiettyjä välineitä ja järjestelmiä käyttäen, tietyille konfiguraatioille.

Puhuttaessa ohjelmistokehityksen systemaattisuudesta, on hyvä tiedostaa että todellisuudessa erikokoisten yritysten erikokoiset ohjelmistoprojektit ovat systemaattisuudeltaan jotain hätäisen koodaamisen ja puhdasoppisen

tieteellisen lähestymistavan väliltä. Täydellisimmilläänkin prosessi on melko varmasti iteratiivinen, eli aikaisempiin vaiheisiin on syytä palata. Vähemmän täydellisessä tapauksessa eri vaiheissa joudutaan käytännössä tekemään aikaisemman vaiheen työ uudelleen – suunnittelu tarkoittaa uudelleen analysointia ja toteutus uudelleen suunnittelua. Olen ollut mukana myös projekteissa, joissa suunnitteluvaiheen dokumentit on tehty viimeisenä, eikä tämä valitettavasti liene kovinkaan epätavallista.

Tämän työn kannalta keskeisimmät vaiheet ovat suunnittelu- ja toteutusvaihe. Arkkitehtuuriratkaisujen uudelleenkäyttö on enimmäkseen suunnitteluvaiheen työn uudelleenkäyttöä, mutta usein suunnittelua vastaava toteutuskin on tällöin käytettävissä. Varsinaisia malleja (patterns) on tarjolla kaikkiin vaiheisiin, myös analyysiin. Tässä työssä keskitytään kuitenkin ainoastaan suunnitteluvaiheen malleihin.

3.3 Olioperustainen ohjelmistokehitys

Olioperustaisella ohjelmistokehityksellä tarkoitetaan systemaattista, koko ohjelmiston elinkaaren läpi käyvää prosessia, joka perustuu kaikissa vaiheissaan ohjelmistojen kuvaamiseen joukkona keskinäisessä vuorovaikutuksessa olevia olioita [Koskimies 2000]. Koska olioperustaiset käsitteet ovat mukana kaikissa ohjelmistokehityksen vaiheissa, pidetään olioperustaista ohjelmistokehitystä usein “saumattomana”. Eri vaiheiden välillä ei siis tarvitse siirtyä kuvaustekniikasta toiseen, kuten aikaisemmin perinteisemmän ohjelmistokehityksen kohdalla.

Tunnetuin systemaattinen menetelmä olioperustaiseen analyysiin ja suunnitteluun on Rumbaugh et al.:n [1991] kehittämä Object Modeling technique (OMT). Menetelmä koostuu sovellusalueen mallintamisesta ja tämän mallin tarkentamista toteutusteknisin yksityiskohdin kehityksen suunnitteluvaiheessa. Olioperustaisen ohjelmistokehityksen ydin on pikemminkin sovelluksen kohdealueen käsitteistön identifioiminen ja

organisoiminen, kuin niiden lopullinen esittäminen olioperustaisella tai perinteisellä ohjelmointikielellä. [Rumbaugh et al. 1991]. OMT-tekniikka voidaan jakaa kuten perinteisempikin ohjelmistokehitys *analyysiin*, *suunnitteluun* ja *toteutukseen*. Näistä vielä suunnitteluvaihe on jaettu *systeeminsuunnitteluun* ja *oliosuunnitteluun*. Seuraavissa alakohdissa esitellään näitä vaiheita hieman tarkemmin.

Kun verrataan näitä olioperustaisen kehityksen vaiheita perinteisen systemaattisen ohjelmistokehityksen vaiheisiin, näyttää siltä ettei ero ole niin radikaali kuin ehkä joskus on annettu ymmärtää. Olioperustaisuus on vain yksi tapa systemaattiseen ohjelmistokehitykseen, joskin tietysti se vakiintunein. Systemaattisen ohjelmistokehityksen kulmakivet on kuitenkin muurattu jo ennen olioita – oliotekniikka tuo apua erityisesti sovellusalueen käsitteistön identifiointiin ja organisointiin.

Verrattaessa olioperustaista ohjelmistokehitystä perinteiseen ohjelmistokehitykseen, ei enää ole kyse kahdesta kilpailevasta menetelmästä. Olioparadigmasta on tullut standardi ja voisi jopa ajatella, ettei paluuta perinteiseen top-down -suunnitteluun ole. Osittava, asteittain tarkentuva suunnittelu tuskin kuitenkaan katoaa ohjelmistokehityksestä, koska se on luonnollinen tapa hahmottaa kokonaisuuksia. Tällaisella ajattelutavalla on paikkansa myös olioperustaisessa ohjelmistokehityksessä.

3.3.1 Olioperustainen analyysi

Olioperustaiseen analyysiin on kehitetty useitakin systemaattisia menetelmiä, joita karkealla tasolla yhdistävät tietyt piirteet. Periaatteessa kaikki olioperustaiset analyysimenetelmät perustuvat sovellusalueen mallintamiseen.

Analyysin tarkoitus on ymmärtää käyttäjän asettamat vaatimukset sovellukselle ja kyetä kuvaamaan ne. Olioperustaisessa analyysissä tämä kuvaaminen tapahtuu muodostamalla sovelluksen käsitemaailmasta malli,

joka kuvaa käsitteiden keskeiset ominaisuudet. Tämän verran yhteistä lienee kaikilla olioperustaisilla analyysimenetelmillä; menetelmät eroavat toisistaan siinä, miten ja millaisia malleja sovelluksen käsitemaailmasta muodostetaan.

[Rumbaugh et al. 1991]:n määritelmä analyysivaiheesta OMT-tekniikassa:

- *Analyysissä* lähtökohta on vaatimusmäärittelyssä; sovelluksen käsitemaailmasta muodostetaan malli, joka kuvaa käsitteiden keskeiset ominaisuudet. Mallia muodostettaessa ollaan edelleen asiakkaan kanssa yhteistyössä, koska vaatimusmäärittely on usein epätäydellinen tai virheellinen. Analyysimalli on suppea mutta täsmällinen kuvaus siitä *mitä* järjestelmän on tehtävä, ei siitä *miten* se tehdään. Mallin olioiden täytyy tässä vaiheessa vielä olla sovelluksen käsitteitä, eikä toteutusteknisiä käsitteitä kuten tietorakenteita. Hyvä malli on sellainen, jonka ohjelmointitaidoton sovellusalueen tuntija ymmärtää.

Analyysissä tärkeää on siis ymmärtää kohdealue ja rakennettavan järjestelmän vastuut sen suhteen. Tätä tarkoitusta varten analyysiä voidaan laajentaa sovellusalueanalyysillä, jossa järjestelmän toimintaympäristön käsitteistö täsmennetään [Koskimies 2000].

Analyysivaiheessa on tärkeää pitäytyä pelkässä analyysissä menemättä asioiden edelle suunnitteluvaiheeseen, toteutusvaiheesta puhumattakaan. Analyysivaiheessa tyypillisesti työskennellään sovellusalueen asiantuntijoiden kanssa, jotka eivät ole suunnittelun tai toteutuksen asiantuntijoita. Tulokseksi saatavan analyysin on oltava sellainen, jonka sovellusalueen asiantuntijat ymmärtävät täydellisesti ja johon he voivat sitoutua projektissa. Tuomalla toteutusteknisiä yksityiskohtia mukaan analyysivaiheeseen varsinainen käsitemalli voi hämärtyä sovellusalueen asiantuntijoilta. Tällöin analyysi ei ole paras mahdollinen, josta seuraa että kehityksen myöhemmätkin vaiheet vaikeutuvat.

3.3.2 Olioperustainen suunnittelu

Vasta suunnitteluvaiheessa on tarkoitus vastata kysymykseen *miten* järjestelmä toimii. Suunnitteluvaihe voidaan jakaa kahteen vaiheeseen: korkean tason systeeminsuunnitteluun (kutsutaan myös arkkitehtuurisuunnitteluksi) ja matalamman tason yksityiskohtaisempaan oliosuunnitteluun.

3.3.2.1 Systeeminsuunnittelu

[Rumbaugh et al. 1991]:n määritelmä systeeminsuunnitteluvaiheesta OMT-tekniikassa:

- *Systeeminsuunnittelussa* tehdään korkean tason päätökset sovelluksen yleisestä arkkitehtuurista. Sovellus jaetaan osajärjestelmiin esitetyn arkkitehtuurin sekä sovelluksen analysoidun rakenteen perusteella. Systeeminsuunnittelijan on tehtävä päätökset sovelluksen korkean tason teknisistä ratkaisuista.

Systeeminsuunnittelussa abstraktiotaso on riittävän korkea, että koko systeemi voidaan vielä nähdä hallittavana käsitteellisenä kokonaisuutena. Isoissa ohjelmistoprojekteissa tämän vaiheen keskeinen toteuttaja on systeemiarkkitehti.

3.3.2.2 Oliosuunnittelu

[Rumbaugh et al. 1991]:n määritelmä oliosuunnitteluvaiheesta OMT-tekniikassa:

- *Oliosuunnittelussa* rakennetaan analyysimalliin pohjautuva malli, joka sisältää jo toteutusteknisiäkin yksityiskohtia. Mallia tarkennetaan systeeminsuunnittelussa hahmotellun strategian mukaisesti, keskittyen luokkien toteuttamisessa tarvittavien tietorakenteiden ja algoritmien suunnitteluun. Sovellusalueen käsitteitä vastaavat oliot analyysimallissa ja toteutusteknisemmät oliot suunnitteluvaiheen

mallissa kuvataan samalla notaatiolla ja samojen käsitteiden avulla, vaikka ne sijaitsevatkin eri käsitetasoilla.

Suunnitteluvaiheen abstraktiotason madaltuessa arkkitehtuurin tasolta oliotasolle siirtyy tyypillisesti myös toteutusvastuu arkkitehdiltä suunnittelijalle. Oliosuunnitteluvaiheessa tarkennetaan systeeminsuunnittelun korkean tason suunnitelmaa, mutta myös analyysivaiheen tuottamaa käsitteellistä mallia täydennetään esimerkiksi uusilla operaatioilla ja uusilla abstraktiotasoilla mallin luokkahierarkiassa.

3.3.3 Toteutusvaihe

[Rumbaugh et al. 1991]:n määritelmä toteutusvaiheesta OMT-tekniikassa:

- *Toteutusvaiheessa* oliosuunnittelussa syntynyt luokkarakenne muunnetaan ohjelmointikielelle. Ohjelmointivaiheen tulisi ainakin teoriassa olla hyvin pieni ja mekaaninen osa ohjelmistokehitystä; kaikki ratkaisevat suunnittelupäätöksethän on tehty jo edellisessä vaiheessa. Valittu ohjelmointikieli tosin saattaa vaikuttaa suunnitteluun jonkin verran, mutta suunnittelu ei silti saisi olla riippuvainen ohjelmointikielen yksityiskohdista.

Erityisesti jos halutaan jostain syystä pitää toteutusvaihe täysin erillään suunnitteluvaiheesta, esimerkiksi ei haluta sitoutua tiettyyn ohjelmointikieleen, on syytä välttää toteutusteknisiä yksityiskohtia suunnitteluvaiheessa. Tyypillisesti ohjelmointikieli kuitenkin on tiedossa suunnitteluvaiheessa ja toteutuksesta voidaan saada tehokkaampikin, kun suunnittelussa on ajateltu toteuttavan ohjelmointikielen erityispiirteitä.

Olioparadigman tehdessä läpimurtoaan oli tarve käyttää hybriditekniikoita, joissa esimerkiksi analyysi ja/tai suunnittelu toteutettiin oliomenetelmin, mutta mahdollisesti toteutus jollain ei-oliokielellä, etteivät vanhempiin tekniikoihin tehdyt investoinnit olisi menneet kokonaan hukkaan. Tällaisissa

tapauksissa on tietysti jouduttu luopumaan joistakin olioparadigman tuomista hyödyistä.

4. Uudelleenkäyttö olioperustaisessa ohjelmistokehityksessä

Ohjelmistotekniikan (software engineering) katsotaan saaneen alkunsa vuonna 1968 pidetyssä NATO Software Engineering konferenssissa, jossa ryhdyttiin etsimään ratkaisuja ns. ohjelmistokriisiin. Kriisillä tarkoitettiin suurten ohjelmistojen laatimisen ongelmallisuutta, johon ensimmäisen sukupolven ohjelmointikielet eivät tarjonneet tukea. Suuria ohjelmistoja oli hankala tuottaa hallitusti ja tehokkaasti ja niistä oli vaikea rakentaa luotettavia. [Krueger 1992]

Ohjelmistojen rakentamisesta oli tulossa teollisuutta, mutta sitä vaivasivat edelleen lastentaudit, jollaisista jo vakiintuneet teollisuuden alat ovat päässeet eroon. Ohjelmistokehityksen tehottomuus ja epäluotettavuus nähtiin paljolti seurauksena *uudelleenkäytön* puutteesta. Eihän rakennusteollisuudessakaan kehitetä ja tuoteta keskeisiä rakennusteollisuuden elementtejä jokaisessa projektissa uudelleen.

Uudelleenkäyttö nähtiin siis ainakin yhtenä ratkaisuna ongelmaan jo silloin kun ongelma määriteltiin. Silti aihe ei lakkaa olemasta ajankohtainen – ohjelmistokehitys on edelleen ongelmallista. Siihen aikaan uusi uudelleenkäytön taso oli valmiiden funktiokirjastojen käyttö, joka oli jo iso askel uudelleenkäytettävyyteen; kaikkia rutiineja ei tarvinnutkaan kirjoittaa itse. Ohjelmistojen ollessa koko ajan suurempia ja kompleksisempia, tarvitaan myös korkeamman tason uudelleenkäyttöä.

Vaikka uudelleenkäyttö kiistatta nähdään positiivisena asiana ja ratkaisuna moneen ohjelmistotekniikan ongelmaan, sen käytölle on yhä esteitä. Seuraavaksi, kohdassa 4.1 tarkastelen tarkemmin syitä uudelleenkäytön ongelmallisuuteen. Kohdassa 4.2 esittelen uudelleenkäytön kohteita ja abstraktiotasoja. Uudelleenkäytön abstraktiotason kohottaminen on

keskeinen ajatus tämän työn kannalta. Kohta 4.3 luo katsauksen olioperustaisuuden uudelleenkäyttöä ja erityisesti suunnittelun uudelleenkäyttöä tukeviin mekanismeihin, joihin paneudutaan tarkemmin seuraavissa luvuissa.

4.1 Uudelleenkäytön ongelmallisuudesta

Syitä uudelleenkäyttöön on siis ollut yhtä kauan kuin laajamittaista ohjelmistokehitystäkin. Kun ohjelmistoja halutaan tuottaa tehokkaasti ja laadukkaasti, on järkevää käyttää uudelleen jo olemassa olevia laadukkaita elementtejä, oli tämän elementin abstraktiotaso mikä hyvänsä. Uudelleenkäyttö pienentää ohjelmiston kehitys- ja testauskustannuksia, sekä parantaa ohjelmiston laatua, joka puolestaan pienentää ohjelmiston ylläpitokustannuksia. [Taivalsaari 1993] esittää vakuuttavan listan uudelleenkäytön hyödyistä: uudelleenkäyttö nopeuttaa ohjelmistokehitystä, vähentää päällekkäisen työn määrää, sallii entistä monimutkaisempien ohjelmistojen valmistamisen, helpottaa ohjelman muuttamista ja ylläpitoa, nostaa ohjelmiston laatua, parantaa ohjelmiston suorituskykyä, sallii pienemmät ohjelmointiyhteisöt, helpottaa ohjelmistosuunnittelun oppimista, tukee osaamisen jakamista, palvelee ohjelmistosuunnittelijoiden kommunikaatiovälineenä ja helpottaa dokumentointia.

Näin ajateltuna voisi kuvitella, ettei uudelleenkäytölle ole mitään esteitä. Hyötyä siis saavutetaan, kun voidaan uudelleenkäyttää jotain aiemmin tuotettua ohjelmiston osaa. Toisaalta varauduttaessa uudelleenkäytettävyyteen voidaan joutua käyttämään suunnitteluun enemmän resursseja kuin unohdettaessa koko uudelleenkäytettävyysaspekti. Suuri osa ohjelmistoyritysten työstä voi olla kertaluonteista tai suunnitteluvaiheessa ei ehkä tiedetä, onko työllä tulevaisuudessa mitään käyttöä. Kaikkia uudelleenkäytettävyyteen tehtyjä panostuksia ei välttämättä voidakaan koskaan lunastaa.

Ohjelmistoprojektit ovat tunnetusti usein myöhässä. Uudelleenkäytön puutteen voidaan ajatella olevan ainakin osittain sekä tämän syy että seuraus. Jokainen ohjelmiston osa, joka ei ole uudelleenkäytetty, vie enemmän aikaa kehittää. Tiukka aikataulu puolestaan ei anna mahdollisuuksia keskittyä muuhun kuin täyttämään käynnissä olevan projektin vaatimukset, ehkä tulevienkin vaatimusten kustannuksella. Voi hyvinkin olla, että ohjelmistokehityksen lyhyen ja pitkän tähtäimen tavoitteet ovat aavistuksen ristiriidassa keskenään, erityisesti jos ohjelmistokehittäjällä ei ole kehitystyössään jatkuvasti uudelleenkäytettävyyden periaate mielessään luonnollisena tavoitteena.

Myös uudelleenkäytöllä on kustannuksensa. Vaikka uudelleenkäytettävä ohjelmistokomponentti olisikin omassa yhtiössä kehitetty ja ilmaiseksi käytettävissä, sen ymmärtäminen ja sovittaminen uudelleenkäyttöä varten vie aikaa. Mitä mutkikkaammasta ja suuremmasta uudelleenkäytettävästä kokonaisuudesta on kyse, sitä suurempi on uudelleenkäytön potentiaalinen hyöty, mutta vastaavasti myös kokonaisuuden käyttöönottaminen vie enemmän aikaa ja sisältää enemmän väärinymmärryksen riskejä.

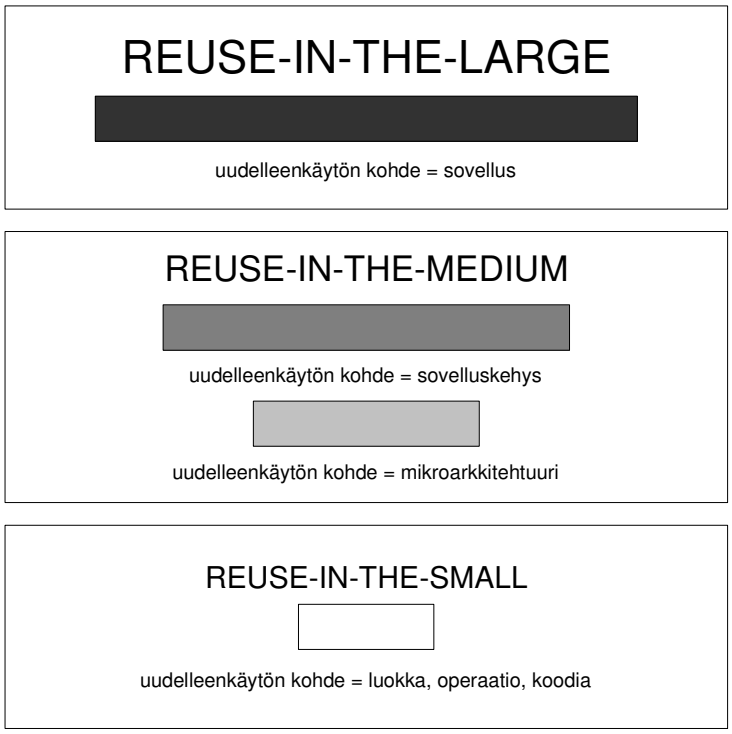
4.2 Mitä ja miten voidaan uudelleenkäyttää?

[Jones 1977] on määritellyt neljä kategoriaa uudelleenkäytön kohteiksi:

1. datan uudelleenkäyttö, jolla tarkoitettiin datan esitysmuotojen standardointia
2. arkkitehtuurin uudelleenkäyttö, jolla tarkoitettiin ohjelmiston loogisen rakenteen suunnittelu- ja ohjelmointikäytäntöjen standardointia
3. suunnittelun uudelleenkäyttö
4. ohjelmakoodin uudelleenkäyttö

Ohjelmistokehitys jaetaan usein eri luokkiin mittakaavansa mukaan. [DeRemer et al. 1976] ovat nimenneet käsitteet *programming-in-the-large, programming-in-the-middle* ja *programming-in-the-small*. [Lajoie et al. 1994] erottelevat uudelleenkäytöstä ja abstrahoinnista samaan tapaan kolme eri tasoa uudelleenkäytön kohteen mittakaavan mukaan. On huomioitavaa, että nämä korkeamman tason uudelleenkäyttötavat kohdistuvat nimenomaan olioperustaisen *suunnittelun* uudelleenkäyttöön *koodin* uudelleenkäytön sijasta.

[Lajoie et al. 1994] esittää seuraavia uudelleenkäytön tasoja:



Kuva 1: uudelleenkäytön tasot Lajoie et al.:n mukaan

Pienen mittakaavan uudelleenkäytössä (*reuse-in-the-small*) uudelleenkäytettävä yksikkö on luokka, sen operaatio ja/tai koodia. Esimerkiksi jo periminen sinänsä on pienen mittakaavan uudelleenkäyttöä. Tästä voidaan ehkä mennä vieläkin pienempään mittakaavaan käyttämällä uudelleen ohjelman lähdekoodin sijaan siitä käännettyä ajettavaa tiedostoa. Tämä on ns. black-box uudelleenkäyttöä.

Keskikokoisen mittakaavan uudelleenkäytöllä (*reuse-in-the-medium*) tarkoitetaan keskinäisessä suhteessa olevien luokkien muodostaman järjestelmän osan uudelleenkäyttöä. Tällaista järjestelmän osaa kutsutaan myös mikroarkkitehtuuriksi. Esimerkkejä näistä mikroarkkitehtuureista ovat tässä tutkielmassa keskeisessä osassa olevat suunnittelumallit. Uudelleenkäytön kohteena keskikokoisen mittakaavan uudelleenkäytössä on Lajoie et al.:n mukaan sekä mikroarkkitehtuurin suunnittelu ja sen toteuttava koodi. Näinhän ei välttämättä tarvitse olla, jos suunnittelu halutaan pitää toteutuksesta täysin erillään.

Keskikokoisen mittakaavan uudelleenkäytöksi Lajoie et al. laskevat myös sovelluskehykset. Tämä uudelleenkäyttö on abstraktiotasoltaan hivenen suunnittelumalleja korkeampaa; itse asiassa kun suunnittelumalli koostuu keskinäisessä vuorovaikutuksessa olevista luokista, koostuu sovelluskehys keskinäisessä vuorovaikutuksessa olevista mikroarkkitehtuureista.

Suurimman mittakaavan uudelleenkäyttö (*reuse-in-the-large*) on kokonaisten sovellusolioiden uudelleenkäyttöä. Nämä sovellusoliot ovat itsenäisiä järjestelmiä, joita voidaan uudelleenkäyttää sellaisenaan, ilman lisäyksiä tai laajennuksia.

Ohjelmistojen kasvaessa on koko ohjelmistokehityksen mittakaava kasvanut. Alkuvaiheessa (ja tietysti edelleenkin) on ollut luontevaa uudelleenkäyttää ohjelmistokehityksen tuloksena saatavaa näkyvää osaa, ohjelmakoodia tai jopa itse ajettavaa ohjelmaa. Ohjelmistokehityksen systematisoituessa suurempi osa kehityksen tuloksesta on muuta kuin koodia; myös analyysivaiheen ja suunnitteluvaiheen seurauksena on konkreettisia tuloksia. Analyysi- ja suunnitteluvaihe vievät myös paljon aikaa projektin kokonaiskestosta, suunnitteluvaihe jopa eniten.

Kun halutaan säästää aikaa ja rahaa uudessa projektissa, uudelleenkäytössä tavallaan käytetään kertaalleen aiemmin käytettyä aikaa ja rahaa. Jos sitä on käytetty eniten juuri suunnitteluun, siitä olisi luonnollisesti saatavissa suurin uudelleenkäytön hyöty. Suunnitteluvaiheen tulosta voidaan myös ajatella

ohjelmistoprojektin arvokkaimpana tuloksena; ohjelmakoodi puolestaan on tavallaan sen ilmentymä.

Haluttaessa ottaa uudelleenkäytöstä mahdollisimman paljon irti, on luonnollista yrittää uudelleenkäyttää niin laajoja käsitteellisiä kokonaisuuksia kuin mahdollista. Funktio tai luokka ovat vielä suhteellisen vaatimattoman tason käsitteellisiä kokonaisuuksia, mutta useamman luokan muodostama hyväksi havaittu rakenne keskinäisine vuorovaikutuksineen on jo sellainen kokonaisuus, jonka suunnittelun uudelleenkäytössä säästetään merkittävästi työtä. Jos mennään tätä laajempiin käsitteellisiin kokonaisuuksiin, rakenteet eivät enää ole niin geneerisiä, että niille olisi helppo löytää uudelleenkäytön mahdollisuuksia. On helppoa huomata muutaman luokan kokoisen käsitteellisen kokonaisuuden toistuminen useissa käsitteellisissä malleissa sovellusalueesta riippumatta, mutta esimerkiksi kymmentä luokkaa suuremmissa käsitteellisissä kokonaisuuksissa on jo todennäköisesti sovellusalueen ja kyseisen ohjelmiston spesifistä logiikkaa mukana – ainakin siinä määrin, että se hankaloittaa uudelleenkäyttöä.

4.3 Uudelleenkäyttöä tukevat oliomekanismit

Eräänä olioteknologian suurimmista eduista pidetään sitä, että se kohottaa ohjelmistokomponenttien abstraktiotason olion tasolle. Tämä abstraktiotason nousu tuo mukanaan useita etuja verrattuna aikaisempaan tilanteeseen, jossa korkeinta abstraktiotasoa edustivat ohjelman funktiot.

Puhuttaessa arkkitehtuuriratkaisuista ja niiden uudelleenkäytöstä, on kysymys ohjelmistokomponenttien abstraktiotason kohottamisesta edelleen yksittäistä oliota korkeammalle. Yksittäistä ohjelmakomponenttia edustaa tällöin useammista luokista koostuva laajempi rakenne, eräänlainen mikroarkkitehtuuri. Tällaisia mikroarkkitehtuureja ovat juuri tässä työssä keskeisessä osassa olevat suunnittelumallit. Edelleen askel samaan suuntaan ohjelmistokomponentin abstraktiotason kohottamisessa on sovelluskehysten

idea, jossa ohjelmistokomponentti koostuu puolestaan useasta mikroarkkitehtuurista.

Vaikka suunnittelumallien tai sovelluskehysten idea saisikin enemmänkin jalansijaa olioperustaisessa ohjelmistokehityksessä, niistä tuskin tulee samanlaista edistysaskelta kuin mitä olioparadigman käyttöönotto aikanaan oli. Puhtaasti olioperustaisessa ohjelmassahan koko ohjelman toiminnallisuus on olioissa. Olio on tällöin myös *pienin* mahdollinen ohjelmistokomponentti. Suunnittelumallien kirjasto täytyisi olla todella laaja, ennen kuin voitaisiin rakentaa ohjelmia, joissa pienin komponentti olisi suunnittelumalli. Sovelluskehysten tapauksessa voitaisiin ehkä ajatella ohjelman koostuvan kokonaisuudessaan sovelluskehyksestä (tai siitä erikoistamalla saadusta sovelluksesta), mutta tällöin ohjelmissa olisi vain yksi komponentti. Jos sovelluskehys toteuttaa järjestelmän osan, voisi ainakin periaatteessa koota useista sovelluskehyksistä koostuvia ohjelmia, joissa pienin komponentti olisi sovelluskehys.

Käytännössä ohjelmistot rakentuvat siis erikokoisista ja -tasoisista komponenteista. Uusien abstraktiotasojen ohjelmistokomponenttien tuominen mukaan olioperustaiseen ohjelmistokehitykseen ei edellytä mitään radikaalia ajattelutavan muutosta. Abstraktiotason nostaminen pikemminkin helpottaa kokonaisuuksien hahmottamista.

5. Suunnittelumallit

Puhuttaessa laajamittaisesta uudelleenkäytöstä, suunnittelun uudelleenkäyttö eli uudelleenkäytön kohteen abstraktiotason nostaminen ja laajentaminen ovat hedelmällisimpiä tapoja saada uudelleenkäytöstä eniten hyötyä irti. Tässä luvussa esitellään tarkemmin suunnittelumallien idea, jota voidaan mielestäni pitää tärkeimpänä lisänä olioperustaisuuden mukanaan tuomaan uudelleenkäytettävyyteen. Kohta 5.1 sisältää johdannon suunnittelumalleihin ja esittelee niiden olennaisimmat elementit. Kohdassa 5.2 esittelen suurimpia suunnittelumallien hyötyjä. Suunnittelumallien konkreettinen osa, on sen dokumentointi, joka esitellään kohdassa 5.3. Esimerkkinä suunnittelumallista esittelen kohdassa 5.4 Observer-suunnittelumallin. Kohdassa 5.5 pohdiskelen suunnittelumallien käyttöä osana ohjelmistokehitysprosessia.

5.1 Johdatus suunnittelumalleihin

Suunnittelumallien idea on peräisin rakennusarkkitehtuurista. Arkkitehti Christopher Alexander kehitti kuvaustekniikan, jonka avulla voidaan esittää rakennusten suunnitteluun ja toteutukseen liittyvää osaamista tietyn formaalin mallin muodossa. Alexanderin mallien mukaisesti ryhdyttiin OOPSLA-91 -konferenssissa soveltamaan suunnittelumallien ideaa olioteknologiaan.

Christopher Alexanderin mukaan malli kuvaa ympäristössämme toistuvasti esiintyvän ongelman ja esittää tähän ongelmaan ratkaisun ytimen siten, että ratkaisua voidaan käyttää uudelleen (tekemättä koskaan samalla tavalla) [Pree 1994]. Suluissa oleva määritelmän loppuosa on täysin ymmärrettävä rakennusarkkitehtuurissa; hyväksi todettua ratkaisua ei esteettisistä syistä kannata kopioida loputtomiin täsmälleen samalla tavalla, varsinkaan jos kyseessä on jokin arkkitehtuurin ulospäin näkyvä osa. Ohjelmistokehityksessä tällaisesta “aina samalla tavalla tekemisestä” ei ole

haittaa; niin ohjelman sisäisessä (toteutus) kuin ulkoisessakin (käyttöliittymä) rakenteessa on samankaltaisuus olemassa olevien, hyväksi havaittujen ohjelmien kanssa pelkästään hyväksi. Tämä seikka saattaa tehdä suunnittelumallit jopa käyttökelpoisemmaksi olioperustaisessa ohjelmistokehityksessä kuin rakennusarkkitehtuurissa. Käsittääkseni mallien idea ei koskaan saanutkaan kovin suurta jalansijaa arkkitehtuurissa — onhan arkkitehtuuri olennaisesti eri tavalla luovaa kuin ohjelmistokehitys.

Oliosuunnittelussa suunnittelumalli on tietyn yleisen suunnitteluongelman ratkaisun dokumentointi. Kyseessä on siis tietyllä tavalla oliosuunnitteluun liittyvä "niksi", jossa esitellään ongelma ja sen ratkaisu. Suunnittelumalli-käsitteellä voidaan viitata sekä tähän ratkaisuun että dokumentoituihin ohjeisiin ratkaisun toteuttamiseksi.

Yleensä monen käytännön taidon oppimisessa helpoin tapa on yrittää matkia kokenutta taitajaa, olipa kyse sitten kitaran soitosta, jääkiekosta, arkkitehtuurista tai oliosuunnittelusta. Suunnittelumallien tärkein anti onkin juuri suunnitteluosaamisen dokumentoiminen ja jakaminen. Oliosuunnittelussa (tai jääkiekossa, toisin kuin kitaran soitossa tai arkkitehtuurissa) suoranainen plagioiminenkaan ei välttämättä ole synti.

[Gamma et al. 1994] luettelevat suunnittelumallien neljä olennaista elementtiä:

1. Suunnittelumallin **nimen** avulla voidaan muutamalla sanalla kuvata suunnitteluongelma, sen ratkaisut ja ratkaisun seuraukset. Suunnittelumallin nimeäminen kasvattaa välittömästi suunnittelusanastoamme ja antaa meidän suunnitella korkeammalla abstraktiotasolla. Mallisanasto helpottaa olennaisesti suunnittelusta kommunikointia, sen dokumentointia ja jopa omaa hahmottamista. Hyvien nimien löytäminen osoittautui erääksi hankalimmista tehtävistä Gamma et al.:n laatiessa suunnittelumalliluetteloaan.

2. **Ongelman** kuvaus esittää milloin mallia voidaan soveltaa. Se selittää ongelman ja sen kontekstin. Se voi kuvata tietyn suunnitteluongelman, esimerkiksi "miten esittää algoritmeja olioina". Se voi kuvata tietyn olioiden rakenteen, joka on osoittautunut hyväksi suunnitteluksi. Ongelman kuvaus saattaa myös sisältää listan ehtoja, joiden tulee olla voimassa, jotta mallia voitaisiin järkevästi käyttää.

3. **Ratkaisu** kuvaa suunnittelun tekevät elementit, niiden väliset suhteet, vastuut ja yhteistyön. Ratkaisu ei kuvaa mitään tiettyä konkreettista suunnittelua tai toteutusta, koska malli on ohje, jota voidaan soveltaa erilaisiin tilanteisiin. Sen sijaan malli esittää suunnitteluongelman abstraktin kuvauksen ja tavan jolla yleinen elementtien (luokat ja oliot tässä tapauksessa) järjestely ratkaisee ongelman.

4. **Seuraukset** ovat mallin soveltamisen tulokset ja mahdolliset haitat. Vaikka seurauksia ei usein mainitakaan suunnittelupäätöksiä kuvattaessa, ne ovat tärkeitä arvioitaessa suunnitteluvaihtoehtoja ja mallin soveltamisen hyötyjä ja haittoja. Ohjelmiston seuraukset koskevat usein tila- ja aikahaittoja ja ne voivat kohdistua myös kieli- ja toteutusteknisiin seikkoihin. Uudelleenkäyttöä silmälläpitäen seuraukset sisältävät myös mallin vaikutukset ohjelmiston joustavuuteen, laajennettavuuteen ja siirrettävyyteen. Näiden seurausten eksplisiittinen luetteleminen auttaa niiden ymmärtämisessä ja arvioimisessa.

[Gamma et al. 1994] ovat koonneet aihealueen keskeisimpään kirjaan 23 suunnittelumallia. Kirjan tarkoitus ei varsinaisesti ole opettaa uusia ohjelmointitekniikoita; sehän ei ole keskeistä suunnittelumalleissa. Hyvälle suunnittelijalle suunnittelumalleissa ei välttämättä olekaan mitään uutta. Hyvä suunnittelu ei kuitenkaan aina ole välttämättä intuitiivista ja toisaalta hyvältäkin näyttävä suunnittelu saattaa sisältää petollisia karikoita. Itse

asiassa suunnittelumalleja onkin olemassa myös muotoa "älä suunnittele näin" (ns. anti-patterns); kokeneella suunnittelijalla on tietenkin kokemusta paitsi "usein toistuvan ongelman hyväksi osoittautuneesta ratkaisusta", myös vaarallisista, epäonnistuneista ratkaisuista.

5.2 Suunnittelumallien hyötyjä

[Gamma et al. 1994] ja [Buschmann et al. 1996] luettelevat suunnittelumallien tunnettuja hyötyjä ja keskeisiä ominaisuuksia:

- *Suunnittelumalli kuvaa tietyissä suunnittelutilanteissa toistuvan ongelman ja esittää siihen ratkaisun.*

  Tämä lause esitetään tyypillisesti suunnittelumallien määritelmänä. Esimerkki tunnetussa suunnittelumallissa Model-View-Controllerissa toistuva ongelma on käyttöliittymän vaihtelun heijastuminen ohjelmiston alempiin kerroksiin. Ratkaisu on irrottaa käyttöliittymän näkymä ohjelman sisäisestä toiminnallisuudesta.

- *Suunnittelumallien avulla voidaan dokumentoida olemassa olevia, hyväksi todettuja suunnittelukäytäntöjä.*

  Olennaista on, että suunnittelumallit eivät sinänsä tarjoa mitään uutta – pikemminkin ne tarjoavat vanhaa! Suunnittelumallien lähtökohta ei saa olla keinotekoinen, vaan suunnittelumallien on synnyttävä kokeneiden suunnittelijoiden suosimista käytännöistä oikeissa ohjelmistoprojekteissa.

- *Suunnittelumallit identifioivat ja spesifioivat yksittäisiä luokkia ja olioita laajempia abstraktiotasoja.*

  Tyypillisesti suunnittelumalli koostuu useasta komponentista, luokasta tai oliosta ja kuvaa niiden väliset suhteet, vuorovaikutuksen ja keskinäiset vastuut mallissa. Malliin kuuluvat komponentit ratkaisevat mallin kuvaaman ongelman yhdessä.

- *Suunnittelumallit edistävät suunnittelukäytäntöjen ymmärrystä ja laajentavat suunnittelun sanastoa.*

  Muistan innostuneeni suunnittelumallien ideasta ensimmäisessä työharjoittelupaikassani, kun kokenut suunnittelija selvitti minulle ohjelmiston luokkahierarkiaa osoittamalla erästä luokkaa ja toteamalla ”tässä on vaan wrapperi” (Gamma et al. muuttivat ”Wrapper”-suunnittelumallin nimen myöhemmin ”Adapteriksi”). Tuo sai minut vakuuttumaan ainakin eräästä suunnittelumallien hyödystä; olin oivaltanut ohjelmiston rakenteesta jo hyvän osan muutaman sanan kommentista, vaikka olin täysi noviisi suunnittelijana.

- *Suunnittelumallit ovat keino dokumentoida ohjelmistoarkkitehtuureja.*

  Kun toinen suunnittelija yrittää perehtyä ohjelmiston arkkitehtuuriin, hän tyypillisesti yrittää asettua alkuperäisen suunnittelijan asemaan ja ymmärtää mitä hän on todella ajanut takaa ratkaisuillaan. Ymmärtämällä arkkitehtuurissa käytetyn suunnittelumallin, suunnittelija voi ymmärtää arkkitehtuuria syvemmin ja pystyy pitämään arkkitehtuurin vision mielessään paremmin joutuessaan mahdollisesti laajentamaan arkkitehtuuria.

- *Suunnittelumallit tukevat spesifisten ohjelmistokomponenttien tuottamista.*

  Suunnittelumallit määrittelevät rungon tietylle toiminnallisuudelle, joka muodostuu osaksi kehitettävän ohjelmiston toiminnallisuutta. Suunnittelumalleja löytyy useisiin spesifisiin tarkoituksiin, kuten esimerkiksi prosessien väliseen kommunikointiin. Toiminnallisten vaatimusten tukemisen lisäksi suunnittelumallit tukevat useita ei-toiminnallisia vaatimuksia kuten muunneltavuutta, luotettavuutta, testattavuutta ja uudelleenkäytettävyyttä. Edellä mainittu Model-View-Controller tukee käyttöliittymän muunneltavuutta ja sen käyttöliittymästä riippumattoman osan uudelleenkäytettävyyttä.

- *Suunnittelumallit tukevat erilaisten kompleksisten arkkitehtuurien rakentamista.*

Suunnittelumallit toimivat rakennuspalikoina rakennettaessa kompleksisia arkkitehtuureja, parantaen ohjelmistokehityksen laatua ja tehokkuutta. Vaikka suunnittelumallin sisältämä ratkaisu ei olisikaan ehdottomasti paras juuri ratkaistavaan ongelmaan, vähimmilläänkin se tarjoaa yhden vaihtoehdon ratkaisuksi ja lisää näin perspektiiviä ongelman tarkasteluun. Suunnittelumallit eivät tyypillisesti ratkaise koko ongelmaa, vaan ongelman keskeisen, geneerisen osan.

- *Suunnittelumallit helpottavat ohjelmiston kompleksisuuden hallintaa.* Suunnittelumallit tarjoavat kuvaamaansa ongelmaan hyväksi todetun ratkaisun. Kun suunnittelija törmää vastaavaan suunnitteluongelmaan, hän voi luottaa mallin ratkaisevan ongelman geneerisen osan ja keskittyä toteuttamaan mallin ratkaisun.

5.3 Suunnittelumallien dokumentointi

Gamma et al. kuvaavat kirjassaan systemaattisen tavan esittää suunnittelumallit. Jo nimen tulisi jonkin verran kuvata mallin toimintaa, koska nimiä yleensä käytetään joka tapauksessa suunnittelusanastossa, ymmärsivät kaikki suunnittelijat niitä tai eivät. Tarkoitus-kenttä antaa lyhyen kuvauksen mallista suunnittelumalliin pikaisesti perehtyvälle. Suunnittelumallia käyttävälle suunnittelijalle on tarjolla runsaasti tietoa muissa kentissä. Seuraavassa käydään suunnittelumallien kentät lyhyesti läpi.

Nimi

Suunnittelumallin nimi kuvaa ytimekkäästi mallin perimmäisen olemuksen. Hyvä nimi on olennainen, koska siitä on tarkoitus tulla osa suunnittelussa käytettävää sanastoa.

Tarkoitus

Lyhyt kuvaus suunnittelumallin toiminnasta. Mitä malli tekee? Mikä on sen tarkoitus? Minkä ongelman malli ratkaisee?

Synonyymit

Muita suunnittelumallista mahdollisesti käytettäviä nimiä.

Motivointi

Skenaario, joka kuvaa suunnitteluongelman sekä miten suunnittelumalli ratkaisee ongelman.

Sovellettavuus

Missä tilanteissa mallia voidaan soveltaa? Minkälaisen huonon suunnittelun tämä malli voi korvata?

Rakenne

Suunnittelumallin luokkien graafinen esitys OMT:lla tai vastaavalla kuvaustekniikalla.

Osalliset

Suunnittelumalliin kuuluvat luokat ja/tai oliot sekä niiden vastuut mallissa.

Yhteistoiminta

Miten suunnittelumallin osalliset toimivat yhdessä?

Seuraukset

Miten suunnittelumalli tukee päämääräänsä? Mitä tuloksia tai kompromisseja mallin soveltamisesta seuraa?

Toteutus

Mistä tekniikoista, vinkeistä tai vaaroista on syytä olla tietoinen suunnittelumallia sovellettaessa? Onko mallin toteutuksessa otettava huomioon joitakin kieliriippuvaisia seikkoja?

Esimerkkikoodi

Esimerkkikoodia, miten suunnittelumalli voitaisiin toteuttaa C++:lla tai Smalltalkilla.

Tunnetut käyttötapaukset

Esimerkkejä suunnittelumallin käytöstä oikeissa ohjelmistoissa.

Liittyvät suunnittelumallit

Mitkä suunnittelumallit liittyvät läheisesti tähän suunnittelumalliin? Mitä eroja näillä malleilla on? Minkä suunnittelumallien kanssa tätä mallia tulisi käyttää?

5.4 Esimerkki suunnittelumallista: Observer

Käyn nyt läpi yhden Gamma et al.:n suunnittelumalleista esitelläkseni mallien ideaa laajemmin. Kyseessä on malli nimeltä Observer, jonka nimi (suomeksi tarkkailija tai havainnoija) jo antaa jonkinasteisen vihjeen siitä, mikä on tämän suunnittelumallin tehtävä.

5.4.1 Nimi: Observer

Suunnittelumalli on nimetty Observeriksi, joka tarkoittaa tarkkailijaa tai havainnoijaa. Mallin nimen yhteydessä puhuttiin nimen tärkeydestä suunnittelijoiden sanavaraston kasvattamisessa. Tietotekniikkayhteisö on epäilemättä tuottanut Kielitoimistolle lukuisia harmaita hiuksia, eikä tämäkään nimeämiskäytäntö taida tehdä poikkeusta. On tarkoituksenmukaisempaa käyttää alkuperäistä englanninkielistä nimeä kuin käännöstä. Alkuperäinen nimi tulee tutuksi useista englanninkielisistä yhteyksistä, mutta on hankalaa vakiinnuttaa suomenkielistä nimeä niin, että se palvelisi tarkoitustaan ja kertoisi suunnittelijalle heti, mistä suunnittelumallista on kysymys.

5.4.2 Tarkoitus

Observerin käyttötarkoitus on määritellä olioiden välinen yhden-suhde-moneen riippuvuus siten, että yhden olion muuttaessa tilaansa, kaikki siitä riippuvat oliot saavat tiedon ja osaavat päivittää tilansa automaattisesti.

5.4.3 Synonyymit

Observer-suunnittelumallista käytetään myös nimiä Dependants (suomeksi ”huollettavat”) ja Publish-Subscribe (suomeksi ”julkaise-tilaa”).

5.4.4 Motivointi

Kun järjestelmä pilkotaan kokoelmaksi keskinäisessä vuorovaikutuksessa olevia luokkia, nousee esiin ongelma ylläpitää toisiinsa liittyvien olioiden konsistenssia. Jos luokat kytketään liian läheisesti yhteen, kokonaisuudesta tulee vaikeammin uudelleenkäytettävä.

Esimerkki Observerin käytöstä on datan irrottaminen sen graafisesta esitysmuodosta. Data voidaan esittää numeerisena taulukkona, pylväsdiagrammina sekä ympyrädiagrammina ja muutos mihin tahansa kolmesta esitysmuodosta voidaan välittömästi heijastaa kahteen muuhun esitysmuotoon.

Keskeinen Observer-mallin tarjoama anti on näiden toisistaan riippuvien olioiden suhteiden määrittämisessä. Mallin rakenne koostuu *subjektista*, sekä tästä riippuvista *havainnoijista*. Subjekti tiedottaa tilansa muutoksista havainnoijilleen. Havainnoija voi myös kysyä subjektin tilaa päivittääkseen oman tilansa.

5.4.5 Sovellettavuus

Observer-suunnittelumallia voidaan käyttää seuraavanlaisissa tilanteissa:

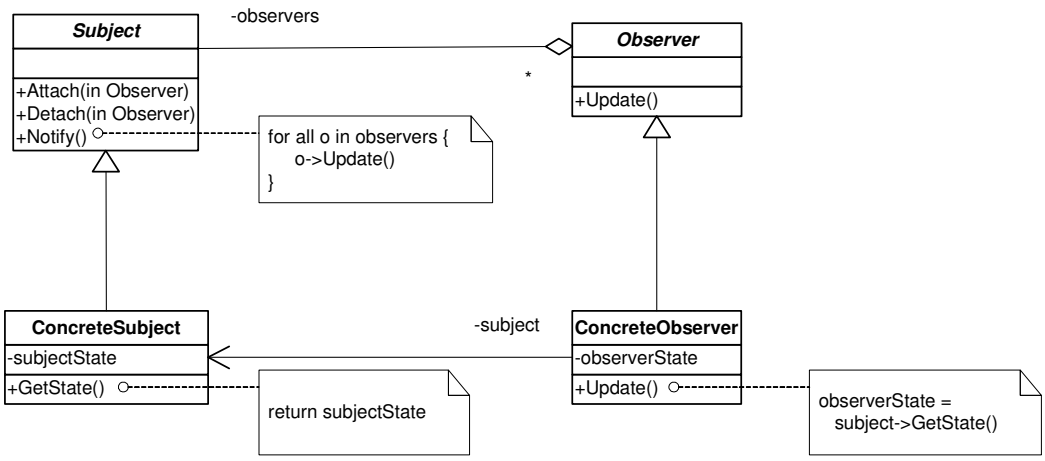
- Kun abstraktiolla on kaksi puolta, jotka ovat riippuvaisia toisistaan. Näiden puolten kapseloiminen omiin olioihinsa antaa paremmat mahdollisuudet muokata ja uudelleenkäyttää abstraktioita.

- Kun yhden olion tilan muutoksen pitäisi heijastua toisten olioiden tiloihin, eikä voida olla varmoja siitä, montako näitä riippuvia olioita on.

- Kun olion pitäisi tiedottaa muutoksesta toisille olioille tekemättä mitään oletuksia näistä olioista tai niiden luokasta, eli kun tiukkaa kytkentää halutaan välttää.

5.4.6 Rakenne

Observer-mallin rakenne voidaan esittää esimerkiksi seuraavan kaltaisella luokkakaaviolla:



Kuva 2: Observer-suunnittelumallin rakenne.

5.4.7 Osalliset

- **Subject**

    - tuntee Observerinsa. Yhtä Subjectia voi havainnoida rajoittamaton määrä Observereita.

- tarjoaa rajapinnan Observerin lisäämiseen ja poistamiseen.

- **Observer**
  - määrittelee rajapinnan, jonka avulla Subject voi kertoa muuttuneesta tilastaan.

- **ConcreteSubject**
  - tallettaa tilaa, josta havainnoivat ConcreteObserverit ovat kiinnostuneita
  - lähettää notifikaation havainnoiville ConcreteObservereille tilansa muuttuessa.

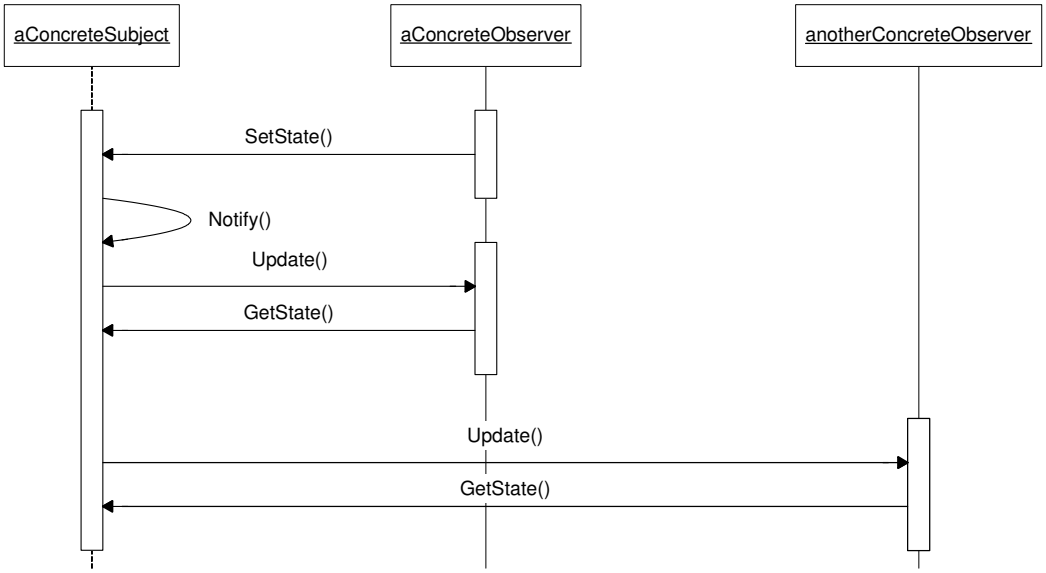
- **ConcreteObserver**
  - ylläpitää viitettä ConcreteSubject-olioon.
  - tallettaa tilaa, jonka pitäisi pysyä identtisenä havainnoimansa kohteen tilan kanssa.
  - toteuttaa Observerin määrittelemän rajapinnan tilansa päivittämiseksi.

5.4.8 Yhteistoiminta

- ConcreteSubject lähettää notifikaation havainnoijilleen aina tilansa muuttuessa. Havainnoijien tilatieto ei välttämättä tällöin ole ajan tasalla.

- Saatuaan tiedon ConcreteSubjectin muuttuneesta tilasta, ConcreteObserver voi kysyä uutta tilaa ConcreteSubjectilta ja päivittää näin oman tilansa vastaavasti.

- Oheinen interaktiokaavio esittää yhden ConcreteSubjectin ja kahden ConcreteObserverin yhteistoimintaa:



Kuva 3: Observer-suunnittelumallin interaktiokaavio.

5.4.9 Seuraukset

Käytettäessä Observer-suunnittelumallia, voidaan mallissa käytettäviä subjekti- ja havainnoija-olioita varioida riippumatta mallista. Subjekti-oliot ovat uudelleenkäytettävissä ilman havainnoija-olioita ja päinvastoin. Malli on myös laajennettavissa uusilla havainnoijilla, eikä olemassa oleviin subjekteihin tai havainnoijiin tarvitse tehdä muutoksia.

Muita Observer-suunnittelumallin käytön seurauksia – etuja ja haittoja – ovat:

1. *Subjektin ja havainnoijan löyhä kytkentä.*

   Subjektin ja sen havainnoijien yhteys muodostuu ainoastaan abstraktissa Observer-yliluokassa määritellyn rajapinnan kautta. Subjekti näkee vain listan havainnoijia, jotka käyttävät rajapintaa, mutta ei esimerkiksi havainnoijan konkreettista luokkaa. Subjektin ja havainnoijien välinen kytkentä on siis minimaalinen. Tämä löyhä kytkentä mahdollistaa ohjelmiston joustavamman pilkkomisen käsitteellisiin kerroksiin, joka parantaa ohjelmiston uudelleenkäytettävyyttä.

2. *Kommunikointimallin tuoma vapaus.*

   Subjektin kommunikointi havainnoijiensa kanssa muistuttaa radiolähetystä; kaikki kiinnostuneet voivat vastaanottaa tietoa, eikä lähettäjän tarvitse välittää vastaanottajien yksityiskohdista tai lukumäärästä. Näin havainnoijia voidaan tarpeen tullen lisätä ja poistaa vapaasti.

3. *Odottamattomat päivitykset.*

   Subjektin havainnoijat eivät ole tietoisia muista havainnoijista, eivätkä näin ollen voi tietää subjektin tilan muutoksen seurauksista. Subjektin päivityspyyntö yhdeltä havainnoijalta voi johtaa esimerkiksi satoihin notifikaatioihin muille havainnoijille ja näistä riippuville olioille, joka kokonaisuutena voi sisältää huomattavan paljon turhaa informaatiota. Välttyäkseen turhilta päivityksiltä tai turhilta reagoimisilta päivityksiin, havainnoija saattaa tarvita tarkempaa tietoa siitä, mikä subjektissa on muuttunut.

5.4.10 Toteutus

Seuraavassa käydään läpi muutamia Observer-mallin toteutukseen liittyviä ongelmakohtia:

1. *Subjektien linkittäminen havainnoijiinsa.*

   Subjektin täytyy ylläpitää tietoa havainnoijistaan. Yksinkertaisin tapa toteuttaa tämä on tallettaa viitteet havainnoijiin subjektissa. Jos subjekteja on paljon ja havainnoijia vähän, tämä voi osoittautua tehottomaksi ratkaisuksi. Tällöin yksi varteenotettava ratkaisu voisi olla esimerkiksi taulukko, johon talletetaan kaikki subjekti-havainnoija –linkit.

2. *Usean subjektin havainnoiminen.*

   Jos havainnoijan on tarve kuunnella useampaa kuin yhtä subjektia, voidaan päivitysrajapintaa joutua laajentamaan. Havainnoijan täytyy

tietää, miltä subjektilta notifikaatio tuli. Subjekti voi käyttää esimerkiksi viitettä itseensä päivityksen yhteydessä, että havainnoija tietää, mitä subjektia tutkia.

3. *Kuka laukaisee päivityksen?*
   Subjektin ja sen havainnoijien välinen konsistenssi perustuu notifikaatiomekanismiin. Kenen pitäisi kutsua subjektin Notify()-operaatiota, että päivitykset lähtisivät havainnoijille?

   a. Subjektin tilaa vaihtavien operaatioiden päätteeksi kutsutaan Notify()-operaatiota. Hyötynä on, että subjektin tilaa päivittävien olioiden ei tarvitse huolehtia subjektin notifikaatiotarpeesta. Haittapuolena voi olla tehottomuus, jos tilaa päivittäviä operaatiokutsuja on useampia peräkkäin.

   b. Velvoitetaan subjektin tilaa päivittävät oliot kutsumaan subjektin Notify()-operaatiota tarvittaessa. Näin voidaan välttyä tarpeettomilta notifikaatioilta, kun subjektin ulkopuolella on tiedossa useasta tilamuutoksesta koostuva kokonaisuus, jonka jälkeen riittää yksi Notify()-operaation kutsu. Haittapuolena ratkaisussa on suurempi virhealttius; päivittävä olio voi jättääkin kutsumatta Notify()-operaatiota.

4. *Irralliset viitteet tuhottuihin subjekteihin*
   Subjektin tuhoaminen ei saisi aiheuttaa sen havainnoijiin irrallisia osoittimia. Subjekti voi lähettää notifikaation havainnoijilleen ennen tuhoutumistaan, jotta nämä voivat poistaa osoittimensa tuhottuun subjektiin.

5. *Subjektin tilan on oltava konsistentti ennen notifikaatiota.*
   Notifikaatiomekanismin tarkoitus on pitää subjekti ja sen havainnoijat konsistentissa tilassa. Tällöin on tärkeätä, ettei subjekti lähetä notifikaatiota silloin, kun sen oman tilan päivitys on vielä keskeneräinen. Tähän ongelmaan törmätään helposti tilanteessa, jossa

kantaluokan operaatiosta lähetetään notifikaatio, mutta sen perivässä luokassa ylimääritellään vastaava operaatio, jossa kutsutaan ensin kantaluokan operaatiota (notifikaatio lähtee) ja tämän jälkeen vielä jatketaan olion tilan päivitystä. Ongelmaan voidaan varautua määrittelemällä operaatiossa ns. templaattioperaatio (toinen suunnittelumalli Gamma et al.:n kirjassa), joka voidaan perivässä luokassa ylimääritellä. Tämä ratkaisu kylläkin tarkoittaa, että jo kantaluokkaa kirjoitettaessa olisi tiedettävä mahdollisista perimistarpeista notifikaatioita lähettäville operaatioille.

6. *Vältä havainnoijaspesifisten päivitysprotokollien käyttöä (esim. push- ja pull-mallit)*

   Observer-suunnittelumallin notifikaatiomekanismia voidaan usein laajentaa lisäämällä notifikaatioiden yhteydessä lähetettävää informaatiota. Toisessa ääripäässä (pull-malli) subjekti lähettää päivityksestään vain minimaalisen informaation ja havainnoijan tehtävä on selvittää kaikki muutoksen yksityiskohdat notifikaation saatuaan. Vastaavasti push-mallissa subjekti voi lähettää kaiken havainnoijan tarvitseman informaation jo notifikaation mukana, oli havainnoija siitä kiinnostunut tai ei. Push-malli voi tehdä havainnoijista vaikeammin uudelleenkäytettäviä, koska iso osa toiminnallisuudesta sysätään subjektin harteille. Pull-mallin haittana puolestaan voi olla tehottomuus, kun tietoa välitetään tarpeettomasti joka notifikaatiossa.

7. *Muutosten eritteleminen kiinnostuksen kohteiden mukaan*

   Notifikaatiomekanismista saadaan tehokkaampi, kun subjektin tilamuutoksista tehdään hienojakoisempi ja havainnoijat voivat rekisteröityä olemaan kiinnostuneita vain tietyntyyppisistä muutoksista subjektissa. Kun tietyntyyppinen muutos tapahtuu subjektin tilassa, lähtee notifikaatio vain niille havainnoijille, jotka ovat ilmoittaneet olevansa kiinnostuneita ko. muutoksista.

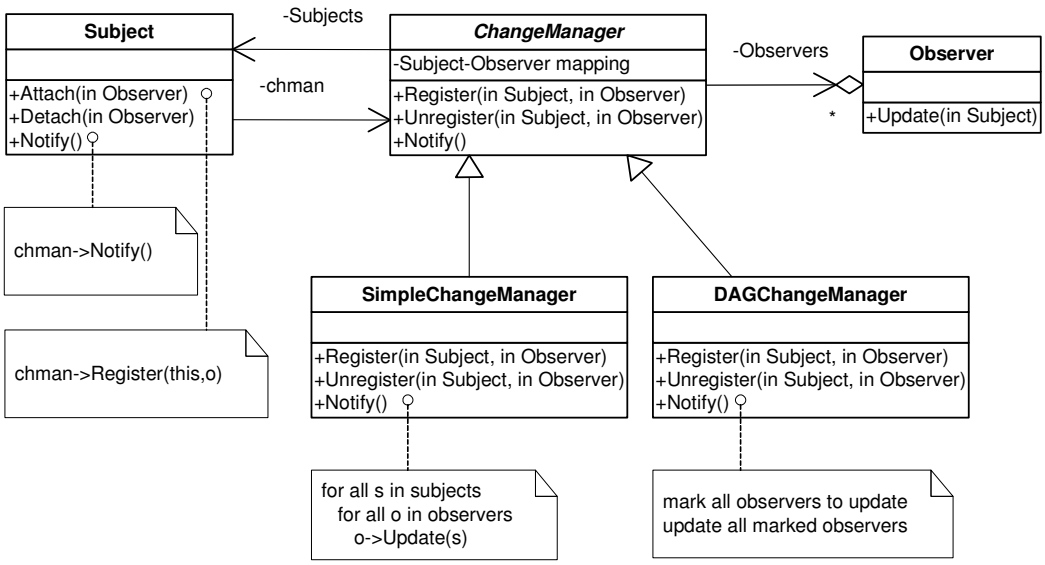
8. *Mutkikkaampien päivitysmekanismien kapselointi*

   Subjektien ja havainnoijien verkosto voi olla mutkikkaampi kuin perustapaus, jossa on yksi subjekti ja useampi havainnoija. Useampien havainnoijien havainnoidessa useampaa subjektia voi olla tarpeen käyttää erillistä ChangeManager-oliota, jonka tehtävä on minimoida havainnoijien päivitystarvetta. Ohjelmistossa voi tapahtua useaa subjektia päivittävä muutos ja notifikaatio niistä olisi syytä lähettää vasta, kun kaikki muutokset ovat tapahtuneet. ChangeManagerilla on kolme vastuuta:

   a. Se linkittää subjektin havainnoijiinsa ja tarjoaa rajapinnan tämän linkityksen ylläpitoon.

   b. Se määrittelee strategian päivitysnotifikaatioiden lähettämiseen.

   c. Subjektin vaatiessa se päivittää kaikki riippuvat havainnoijat.



Kuva 4: Observer-suunnittelumallin laajennettu rakenne.

Oheinen luokkakaavio esittelee ChangeManager-oliolla laajennetun Observer-mallin. Siinä SimpleChangeManager toteuttaa yksinkertaisimman mahdollisen päivitysmekanismin, jossa kaikkien

subjektien havainnoijat päivitetään. DAGChangeManager puolestaan hallitsee subjektien ja havainnoijien välisiä suunnattuja asyklisiä riippuvuuksien verkkoja, jolloin useamman subjektin samalle havainnoijalle aiheuttama päivitysten määrä voidaan pienentää yhteen. Jos useampi päivitys ei ole ongelma, on SimpleChangeManagerkin toimiva vaihtoehto.

9. *Subjektin ja havainnoijan yhdistäminen*
   Ohjelmointikielen erityispiirteet voivat tarjota vaihtoehdon, jossa subjekti ja havainnoija on yhdistetty yhteen luokkaan. Esimerkiksi Smalltalkissa voidaan samalle luokalle määritellä useita eri rajapintoja. Tällöin voidaan määritellä luokka, joka toteuttaa molemmat rajapinnat ja toimii sekä subjektina että havainnoijana.

5.4.11 Esimerkkikoodi

Observerin rajapinnan määrittelee abstrakti luokka:

```
class Subject;

class Observer {
public:
     virtual ~Observer();
     virtual void Update(Subject* theChangedSubject) = 0;
protected:
     Observer();
};
```

Tämä toteutus tukee useita subjekteja yhdelle havainnoijalle. Update-funktion parametrina välitetään muuttunut subjekti silloin kun tarkkailtavia subjekteja on useampia.

Myös subjektin rajapinta määritellään abstraktilla luokalla:

```
class Subject {
public:
     virtual ~Subject();

     virtual void Attach(Observer*);
```

```
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

ClockTimer on konkreettinen subjekti, joka ylläpitää kellonaikaa. Se lähettää havainnoijilleen notifikaation kerran sekunnissa. ClockTimer tarjoaa rajapinnan tuntien, minuuttien ja sekuntien hakemiseen.

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

Järjestelmän sisäinen kello kutsuu Tick() –operaatiota, jonka avulla ClockTimer pitää kellonaikansa oikeana. Tick() –operaation lopuksi kutsutaan Notify() –operaatiota, jonka avulla kerrotaan havainnoijille muuttuneesta kellonajasta.

```
void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
```

```
    Notify();
}
```

Seuraavaksi määritellään konkreettinen havainnoija nimeltään DigitalClock, joka näyttää kellonajan. Graafisen toiminnallisuutensa se perii käyttöliittymäkirjaston tarjoamalta Widget-luokalta, Havainnoija-rajapinnan puolestaan Observer-luokalta.

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        //overrides Observer operation

    virtual void Draw():
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

Ennen kuin Update-operaatio piirtää kellon, se varmistaa että kellon päivittämiseen tullut notifikaatio on tullut oikealta subjektilta:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw() {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.
```

```
    // draw the digital clock
}
```

Samalla lailla voidaan määritellä myös AnalogClock:

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

Seuraavassa luodaan sekä AnalogClock- että DigitalClock-oliot, jotka näyttävät aina samaa aikaa:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Nyt timerin tikittäessä molemmat kellot päivittyvät ja näyttävät oikeaa kellonaikaa.

5.4.12 Tunnetut käyttötapaukset

Ensimmäinen ja epäilemättä tunnetuin esimerkki Observer-suunnittelumallin käytöstä on Model/View/Controller (MVC), Smalltalk-ympäristön käyttöliittymäkehys. MVC:n tapauksessa Model edustaa Observer-mallin Subjektia ja View puolestaan Havainnoijien kantaluokkaa. Smalltalkissa, ET++:ssa ja THINK-luokkakirjastossa Observer-mallin konsepti on niin sisäänrakennettu, että subjekti- ja havainnoija-rajapinnat löytyvät valmiina järjestelmän kaikkien luokkien kantaluokasta.

Muita tunnettuja Observer-mallia hyödyntäviä käyttöliittymäkirjastoja ovat mm. InterViews, the Andrew Toolkit sekä Unidraw.

5.4.13 Liittyvät suunnittelumallit

Observer-malliin läheisesti liittyviä suunnittelumalleja ovat:

Mediator: ChangeManager-olio toimii Mediator-mallin mukaisena välittäjänä Subjektien ja Havainnoijien välillä.

Singleton: ChangeManager-olio voi käyttää Singleton-mallia muodostaakseen yhden, yleisesti käytettävissä olevan ChangeManager-olion.

5.5 Suunnittelumallien käyttö

Suunnittelumallien käyttöön ei varsinaisesti ole vakiintunutta käytäntöä, miten ja missä vaiheessa suunnittelumallit pitäisi sisällyttää ohjelmistokehitysprosessiin. Tämän vaiheen systematisoinnissa törmätään samaan problematiikkaan, joka vallitsee uudelleenkäytössä laajemminkin: uudelleenkäytön sisällyttäminen systemaattiseen ohjelmistokehitysprosessiin on yhtälailla ratkaisematta, vaikka suunnittelumallien (tai sovelluskehysten) tarjoaman korkeamman abstraktiotason uudelleenkäytön sijaan puhuttaisiinkin perinteisemmästä uudelleenkäytöstä, koodin tai edes luokkien uudelleenkäytöstä.

[Taivalsaari 1993] on jakanut uudelleenkäyttöprosessin viideksi itsenäiseksi osa-alueeksi. Vertailen seuraavaksi suunnittelumallin uudelleenkäyttöä luokkien uudelleenkäyttöön näiden osa-alueiden valossa :

1. Uudelleenkäytettävän ohjelmistokomponentin löytäminen.
2. Komponentin ymmärtäminen.
3. Komponentin sovittaminen.
4. Uuden komponentin määrittely.
5. Uuden komponentin liittäminen kokoelmaan.

Komponentin löytäminen entuudestaan tuntemattomasta laajasta luokkakokoelmasta voi olla suuritöinen tai jopa mahdotonkin tehtävä, jos ei ohjelmointiympäristö tarjoa siihen tukea. Parhaassa tapauksessa löydettävä komponentti vastaa haettua lähinnä sovellusalueensa puitteissa, ei niinkään välttämättä toiminnallisuutensa tai käytettävyytensä puolesta. Suunnittelumallien tapauksessa löytäminen lie yhtä vaikeaa tai vaikeampaakin – käytettävissä on hyvin vähän ”johtolankoja”, kun ei puhuta sovellusaluekohtaisesta ongelmasta (tai tarjolla ei ole sovellusaluekohtaisia suunnittelumalleja). Käytännössä vähintäänkin mallien perusperiaatteiden tulisi olla suunnittelijalle tuttuja jo ennen varsinaiseen suunnitteluongelmaan törmäämistä. Suunnittelumallien potentiaali uudelleenkäytettävänä komponenttina on kuitenkin mielestäni suurempi kuin abstraktiotasoltaan pienemmässä yksikössä, luokassa. Uudelleenkäytettäessä suunnittelua, luokkien välisen vuorovaikutuksen sisältävää rakennetta, ratkaistaan todennäköisesti perustavampaa laatua olevia ongelmia, kuin käytettäessä yhden luokan sisältämää ratkaisua johonkin todennäköisesti sovellusaluekohtaiseen ongelmaan. Yksittäinen luokkakin voi toki sisältää merkittävän ratkaisun kapseloituna sisäänsä, mutta usein luokkien vuorovaikutuksen sisältävä rakenne ratkaisee yleisempiä ja yleisemmän tason ongelmia. Uudelleenkäytettävien suunnittelumallien joukko ei ole kovin suuri, voidaan esimerkiksi keskittyä vain Gamma et al.:n peruskokoelmaan; siinä mielessä suunnittelumallien löytämistä voisi pitää helpompana.

Komponentin ymmärtämisen suhteen tilanne on mielestäni aika tasoissa verrattaessa suunnittelumallin ja yksittäisen luokan uudelleenkäyttöä. Suunnittelumallit ovat tyypillisesti yleisempiä, sovellusalueriippumattomia verrattuna yksittäisiin luokkiin.

Komponentin sovittaminen on ilman muuta suuritöisempää suunnittelumallien tapauksessa johtuen mallien yleisestä luonteesta. Suunnittelumallit eivät tarjoa ratkaisussaan käytettäväksi valmiita, konkreettisia luokkia, vaan ne vaativat aina sovittamista. Yksittäisten luokkien kohdalla sovittamistyö on ideaalitapauksessa minimaalinen.

Puhuttaessa uuden komponentin määrittelystä, on taustalla ajatus, että kaikki kehitystyö tähtää spesifisten ongelmien ratkaisemisen lisäksi uudelleenkäytettävien komponenttien tuottamiseen. Tämä ajatus pitää paikkansa lähinnä pienimuotoisemmassa uudelleenkäytössä kartutettaessa olemassa olevaa luokkakokoelmaa. Käyttökelpoisia suunnittelumalleja ei synny päivittäisessä ohjelmointityössä vastaavassa mittakaavassa; ratkaistavissa ongelmissa ei välttämättä ole nähtävissä mitään yleisempiä totuuksia takana. Suunnittelumallien synnyttämiseen vaaditaan myös huomattavasti kokemusta, että voidaan nähdä ongelman olevan toistuva ja että osataan löytää esimerkiksi mallin dokumentoinnissa tarvittavat seuraukset, tunnetut käyttötavat, sudenkuopat ja niin edelleen.

Komponentin liittäminen kokoelmaan liittyy olennaisesti edelliseen kohtaan, uusia suunnittelumalleja ei synny suunnittelumallikokoelmaan niin usein kuin luokkia luokkakokoelmaan. Sinänsä suunnittelumallin lisääminen olemassaolevaan kokoelmaan ei ole sen ongelmallisempaa kuin yksittäisen luokankaan, pikemminkin päinvastoin, sillä suunnittelumallikokoelman mallit ovat luonteeltaan irrallisempia toisistaan.

6. Sovelluskehykset

Sovelluskehys (*application framework*) tarkoittaa olio-ohjelmoinnin yhteydessä sovellussuuntautunutta luokkakokoelmaa, josta sopivasti täydentämällä saadaan valmis sovellus tai sen merkittävä osa [Koskimies 2000]. Sovelluskehys on kokoelma abstrakteja ja konkreettisia luokkia sekä niiden välisiä liittymiä. Sovelluskehys sisältää alijärjestelmän suunnittelun [Wirfs-Brock et al. 90]. Periaatteessa sovelluskehys-termillä tarkoitetaan täydellisen sovelluksen arkkitehtuurin sisältävää kehystä. [Koskimies 2000] käyttää erikseen termiä ohjelmistokehys, kun kehyksen sisältämä perusarkkitehtuuri ei välttämättä ole täydellinen. Ohjelmistokehys käsitteenä voisi kokonsa puolesta sijoittua suunnittelumallin ja sovelluskehyksen väliin. Toisaalta suunnittelumallinkaan kokoa ei varsinaisesti ole rajoitettu, joten mallin ja kehyksen välinen raja on muutenkin häilyvä.

[Gamma et al. 1994]:n määritelmä sovelluskehyksestä ei tee suurta eroa sovelluskehyksen ja suunnittelumallin välille: Sovelluskehys on joukko keskenään operoivia luokkia, jotka muodostavat tietyn sovellustyypin uudelleenkäytettävän suunnittelun.

Suunnittelumalli on tietyn yleisen ongelman arkkitehtuuriratkaisun dokumentointi, kun taas sovelluskehys on kokonaisen sovellusperheen yhteiset osat toteuttava ohjelmisto, joka ratkaisee erityisesti sovellusten yleisen arkkitehtuurin.

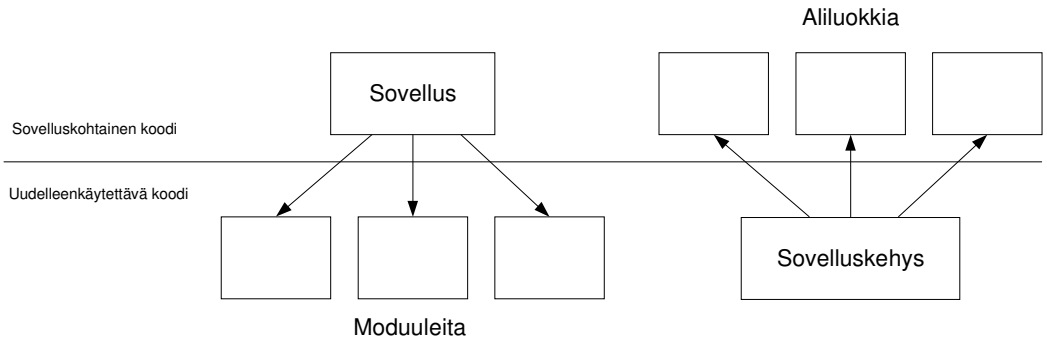
Ensimmäinen laajasti käytetty sovelluskehys oli Model/View/Controller (MVC), Smalltalk-80-järjestelmän käyttöliittymäkomponentti [Wirfs-Brock et al. 90]. MVC-malli on hyvä esimerkki sovelluskehysten ja suunnittelumallien välisestä läheisestä suhteesta. Kuten aiemmin todettiin, suunnittelumallia voidaan pitää uudelleenkäytettävänä mikroarkkitehtuurina, kun taas sovelluskehys toteuttaa kokonaisen ohjelman arkkitehtuurin. MVC-mallissa on kysymys arkkitehtuuriratkaisusta, joka asettuu kooltaan

mikroarkkitehtuurin ja makroarkkitehtuurin väliin; niinpä sitä pidetäänkin sekä suunnittelumallina että sovelluskehyksenä.

Sovelluskehys voidaan joissakin tapauksissa kuvata hyvin suunnittelumalleja apuna käyttäen. Suunnittelumallien ja sovelluskehysten ideathan sivuavat toisiaan; ne ovat uudelleenkäytettäviä ja muunneltavia. Sovelluskehykset voivat joissakin tapauksissa koostua kokonaan suunnittelumalleista. Suunnittelumallit ovat sovelluskehysten arkkitehtuurin perusta [Koskimies 2000].

Olio-ohjelmoinnin ehkä suurimpana etuna pidetään uudelleenkäytettävyyttä. [Wirfs-Brock et al. 90]:n mielestä tämä on suureksi osaksi sovelluskehysten ansiota. Kuten abstrakti luokka antaa konkreettisen luokan suunnittelun, on sovelluskehys alijärjestelmän suunnittelu.

[Koskimies 2000] vertaa mielenkiintoisella tavalla sovelluskehyksen kautta tapahtuvaa uudelleenkäyttöä perinteiseen luokkien uudelleenkäyttöön, jota oheinen kuva havainnollistaa. Perinteisessä yleiskäyttöisen luokkakirjaston uudelleenkäytössä sovellus kutsuu kirjaston yleiskäyttöisten luokkien palveluita, kun sovelluskehyksen tapauksessa uudelleenkäytettävä kehys kutsuu sovellusaluekohtaisten aliluokkien palveluita.



Kuva 5: Sovelluskehysten uudelleenkäyttö verrattuna perinteiseen uudelleenkäyttöön.

7. Suunnittelumallit ja sovelluskehykset tänään

Tein tutkimuksen aiheesta kattavan kirjallisuuskartoituksen aiheen ollessa polttavan ajankohtainen, eikä lähdemateriaalia tuntunut puuttuvan. Samanlaisen kartoituksen tekeminen vuosien kuluttua kohun jo laannuttua toivoakseni paljastaa aiheen todellisen merkityksen. On ymmärrettävää, että aiheen ollessa ajankohtainen siitä halutaan kirjoittaa ja julkaista useita artikkeleita. Jos kuitenkin ajankohtaisuuden hiivuttua aihe ei enää kiinnosta tutkijoita, on syytä kysyä oliko koko ajatuksessa sittenkään mitään kestävää ja alan kannalta oikeasti merkittävää.

Tehdessäni uutta kirjallisuuskartoitusta, huomasin että valtaosa olioperustaisen ohjelmistokehityksen kirjallisuudesta on peräisin 90-luvulta. Suunnittelumalleista ja sovelluskehyksistä julkaistujen kirjojen ja artikkelien julkaisuajankohdat sijoittuvat vielä tiiviimmin 90-luvun puolivälin paikkeille. Alkuperäisten perusteosten jälkeen on julkaistu lähinnä suunnittelumallikokoelmia. Suunnittelumallien ideahan on toisaalta varsin yksinkertainen ja siitä kirjoitettiin suhteellisen runsaasti aiheen ollessa ajankohtaisimmillaan. Uusien näkökulmien puute ei tietenkään tarkoita, etteikö alkuperäinen ajatus mekanismeista suunnittelun uudelleenkäyttöön olisi merkittävä. Muiden kohuttujen tutkimuskohteiden tapaan tämäkin aihe kuitenkin vaikuttaa olleen aikanaan ylimainostettu, jos aiheen tutkimus on alkuinnostuksen jälkeen lopahtanut täysin. Mitään merkittävää läpimurtoa käytännön työkaluihin ei myöskään ole tapahtunut.

[Harrison et al. 2000] arvioivat OOPSLA 2000 konferenssin oivallisesti nimetyssä ”Beyond the hype” paneelikeskustelussa suunnittelumallien herättämiä lupauksia ja niiden toteutumista. Keskustelun johdannossa pohdiskellaan edelleen lähtöruudun kysymyksiä; lisäävätkö mallit tuottavuutta ja käyttäjiensä asiantuntemusta, onko tämä edes realistinen

tavoite ja jos ei, mikä sitten pitäisi olla mallien tavoite? Mihin mallit ovat menossa tai mihin niiden pitäisi olla menossa. John Vlissides, yksi neljästä Design Patterns –kirjan kirjoittajasta asettuu puolustuskannalle esittäen ettei suunnittelumalleilta tulisikaan odottaa mitään muuta kuin sellaista tietämystä jonka ekspertit jo tietävät. Panelisteista David Ungar ja James Coplien esittävät erittäin kriittisen, osittain provosoivankin tuntuisen kannan. Ungar pitää ainoana merkittävänä asiana suunnittelumallien laatua, jota olisi tarkasteltava kriittisemmin. Coplien ei katso Design Patterns –kirjan mallien onnistuneen Alexanderin työn varsinaisen olemuksen saavuttamisessa. Frank Buschmann korostaa suunnittelumallien esimerkinomaisuutta.

Ehkä suunnittelumallien soveltaminen ei voikaan olla niin systemaattista, jos ja kun mallikokoelmasta ei kuitenkaan löydy patenttiratkaisua joka ongelmaan. Vaikka malleja aktiivisesti sovellettaisiinkin, on suuri osa sovelluksen logiikasta kuitenkin ratkottava tapauskohtaisesti. Suunnittelumallit ja sovelluskehykset antavat hyvän pohjan suunnittelutyölle ratkomalla toistuvia perusongelmia antaen suunnittelijan keskittyä olennaiseen. Konseptina suunnittelumallit jäävät kuitenkin elämään. On vaikea mitata, miten paljon suunnittelumallit ja sovelluskehykset ovat helpottaneet suunnittelijoiden päivittäistä työtä; sitä tuskin on mitään syytä aliarvioida.

8. Yhteenveto

Olemme käsitelleet arkkitehtuuriratkaisujen uudelleenkäyttöä olioperustaisessa ohjelmistokehityksessä. Vaikka uudelleenkäytön avulla saavutettavat edut ohjelmistokehityksessä ovat kiistattomat, uudelleenkäyttö on edelleen ongelmallista. Tässä tutkielmassa on käsitelty uudelleenkäytön parantamista suuntaamalla uudelleenkäytön kohde uudelleen. Suunnittelumallien ja sovelluskehysten tarjoama tapa uudelleenkäytön abstraktiotason nostamiseksi koodin tasolta ja luokan tasolta keskenään operoivien luokkien tasolle on merkittävä lisäarvo uudelleenkäyttöön olioperustaisessa ohjelmistokehityksessä.

Murtauduttuaan olio-ohjelmointiyhteisön tietoisuuteen viitisentoista vuotta sitten suunnittelumallit oli paljon lupauksia herättävä konsepti. Nyt nähdään, että mallit tarjoavat edelleen erinomaisen tavan oppia kokeneemmilta suunnittelijoilta ja ratkoa toistuvasti esiintyviä ongelmia – ehkä juuri sitä mitä Gamma et al. alan perusteoksessaan esittivät, ei enempää eikä vähempää. Systemaattiseen ohjelmistokehitysprosessiin suunnittelumallit eivät näytä murtautuneen. On yrityksestä tai jopa yksittäisestä suunnittelijasta kiinni, miten hän haluaa suunnitteluongelmansa ratkaista. Suunnittelumallien menestyksekäs soveltaminen edellyttää aktiivista hakeutumista mallien pariin, mitenkään automaattisesti ne eivät suunnittelijan ongelmia ratkaise.

Lähdeluettelo

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *1996. Pattern-oriented software architecture: a system of patterns. Wiley & Sons. New York.*

De Champeaux D., Lea D., Faure P. *1992. The Process of Object-Oriented Design. Proceedings of OOPSLA 1992.*

DeRemer F., Kron H.H. *1976. Programming-in-the-large versus programming-in-the-small. IEEE Transactions on Software Engineering, 2. 80-86.*

Gamma E., Helm R., Johnson R., Vlissides J. *1994. Design Patterns: Elements of Reusable Object-Oriented software. Addison-Wesley, Reading, Massachusetts.*

Haikala I., Märijärvi J. *2000. Ohjelmistotuotanto. Suomen ATK-kustannus Oy, Helsinki.*

Harrison N., Buschmann F., Coplien J., Ungar D., Vlissides J. *2000. Beyond the hype (panel session): sequel to the trial of the gang of four. Proceedings of OOPSLA 2000*

Jones T.C. *1984. Reusability in programming: A survey of the state of the art. IEEE Transactions on Software Engineering vol 10, no 5, 488-494.*

Koskimies K. *2000. Oliokirja. Suomen ATK-kustannus Oy.*

Krueger C.W. *1992. Software Reuse. ACM Computing surveys 24,2. 131-183.*

Lajoie R., Keller R.K. *1994. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. Proceedings of ACFAS.*

Meyer B. *1988. Object-Oriented Software Construction. Prentice Hall. Hemel Hempstead.*

Pree W. *1994. Design Patterns for Object-Oriented Software Development. Addison-Wesley, Reading, Massachusetts.*

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

Shlaer S. and Mellor S.J. *1988. Object-Oriented Systems Analysis: Modeling the World in Data. Yourdon Press Computing Series.*

Taivalsaari A. *1993. A critical view of inheritance and reusability in object-oriented programming. Jyväskylän yliopisto.*

Wirfs-Brock R., Johnson R.E. *1990. Surveying current research in object-oriented design. Communications of the ACM, 33(9):104-124.*