Markku Finnilä

PROSEDURAALISEN TIETOKONEOHJELMAN MUUTTAMINEN OLIOSUUNTAUTUNEEKSI

Opinnäytetyö
KESKI-POHJANMAAN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Kesäkuu 2012

KESKI-POHJANMAAN AMMATTIKORKEAKOULU
MELLERSTA ÖSTERBOTTENS YRKESHÖGSKOLA

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Yksikkö Tekniikan ja liiketalouden yksikkö, Kokkola	**Aika** Kesäkuu 2012	**Tekijä/tekijät** Markku Finnilä
Koulutusohjelma Tietotekniikan koulutusohjelma		
Työn nimi Proseduraalisen tietokoneohjelman muuttaminen oliosuuntautuneeksi		
Työn ohjaaja Sakari Männistö		**Sivumäärä** 33
Työelämäohjaaja Raimo Kunnari		

Sain opinnäytetyön aiheen Sofor Oy:ltä. Sofor on Kauhavalla 1991 perustettu, viidellä paikkakunnalla toimiva ohjelmistoyritys, joka toimittaa paitsi valmiita tuotteita, myös asiakkaan tarpeisiin räätälöityjä sovelluksia.

Opinnäytetyössä selvitettiin, millaiset vaiheet tulee ottaa huomioon, kun muutetaan proseduraalisella ohjelmointitavalla tehty ohjelma olioparadigmaa noudattavaksi. Toisinaan järjestelmän täydellinen uudelleen rakentaminen olisi liioiteltua, mutta ohjelman yhteistoiminta oliosuuntautuneiden järjestelmien kanssa on tärkeää. Selvitin siis työssäni myös erilaisia kevyempiä tapoja muutoksen tekemiseen.

Erityishuomiota kiinnitin Lotus Domino -ohjelmien ohjelmointiin ja erityispiirteisiin. Sovelsin ohjelman oliosuuntautuneeksi muuttamisesta havaittuja seikkoja tyypilliseen Domino-sovellukseen.

Asiasanat
atk-ohjelmat, dokumentointi, Lotus Notes, ohjelmistokehitys, ohjelmointi, ohjelmointikielet, oliokeskeisyys, työryhmäohjelmistot

CENTRAL OSTROBOTHNIA
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Unit Business and Technology, Kokkola-Pietarsaari	**Date** June 2012	**Author** Markku Finnilä
Degree programme **Information Technology**		
Name of thesis Transforming procedural software into object oriented		
Instructor Sakari Männistö		**Pages** 33
Supervisor Raimo Kunnari		

Sofor Oy offered me this subject for thesis. Sofor is a software company founded in Kauhava in 1991, which now operates in five cities across Finland and offers their customers ready-made products as well as tailored software to fit their specific needs.

In this thesis different stages were examined that should be taken into consideration when transforming a program written in procedural code, into one following the object oriented paradigm. Sometimes complete restructuring of a program would be exaggerating, however the co-operation of the program with object oriented systems is important. In this thesis, I also examined the different lighter ways of completing the transformation.

Special attention was paid in the special characteristics and programming of Lotus Domino applications. The perceived matters from transforming the program into object oriented were utilized in a typical Domino application.

Key words
computer programs, documentation, Lotus Notes, object orientation (Computer science), programming, programming languages, software development, workgroup software

KÄSITTEET JA LYHENTEET

Abstraktio	ohjelmasta eri tarkkuuksisilla tasoilla tuotettava graafinen tai tekstimuotoinen esitys, jonka korkeimpana tasona ovat käyttäjän vaatimukset ja matalimpana ohjelmakoodin taso
API	Application Programming Interface; ohjelmointirajapinta, jonka kutsujen avulla voidaan hyödyntää toisia ohjelmia tai järjestelmiä
CGI	Common Gateway Interface; internetselaimen ja palvelimen välinen standardinmukainen tiedonvälitysmenetelmä, jossa käytetyt CGI-ympäristömuuttujat sisältävät tietoja selaimen ja palvelimen ominaisuuksista
HTML	HyperText Markup Language on Www-sivujen kuvauskieli
Ohjelmointiparadigma	ohjelmointikielen taustalla oleva tapa hahmottaa ja järjestää eri osat, jotka muodostavat toimivan ohjelman
Pseudokoodi	pseudokoodi sisältää sekä luonnollista kieltä että ohjelmointikielen komentoja, ja auttaa ymmärtämään ohjelmointirakenteita paremmin.
UML	Unified Modeling Language; standardinmukainen kokoelma piirrossuosituksia, joita käyttämällä voidaan rakentaa kaavioita esittämään käyttöliittymän tai ohjelman osien välisiä suhteita.
XML	eXtensible Markup Language; teollisuusstandardin aseman saavuttanut rakenteellinen kuvauskieli, jota käytetään järjestelmien väliseen kommunikaatioon tai tiedon tallennusmuotona.

TIIVISTELMÄ
ABSTRACT
KÄSITTEET JA LYHENTEET
SISÄLLYS

1 JOHDANTO 1

2 ERILAISET OHJELMOINTITAVAT 3
2.1 Spagettikoodi 3
2.2 Proseduraalinen ohjelmointitapa 3
2.3 Oliokeskeinen ohjelmointi 5

3 TAKAISINMALLINNUS 8
3.1 Ohjelman ymmärtäminen 9
3.2 Uudelleendokumentointi 10
3.3 Suunnittelumallien tunnistaminen 10

4 OLIOINTI 12
4.1 Oliopiirteiden tunnistaminen proseduraalisesta koodista 12
4.2 Kääriminen 12

5 LUOKKAKIRJASTOT 14

6 MUUTOSTARPEEN ARVIOINTI 16

7 DOMINO-SOVELLUKSEN OHJELMOINTI 18
7.1 Ohjelmoitavia kohteita Domino-sovelluksessa 18
7.2 Ohjelmointikielet 19
7.2.1 JavaScript 19
7.2.2 Formula-kieli 19
7.2.3 LotusScript 21
7.2.4 Java 22
7.2.5 Muita kieliä 22
7.3 Koodikirjastot 23
7.4 Automaattinen dokumentaatio 24

8 DOMINO-SOVELLUS OLIOSUUNTAUTUNEEKSI 26
8.1 Käyttöliittymätapahtumat 26
8.2 Agentit 29

9 YHTEENVETO 31

LÄHTEET 32

KUVIOT

KUVIO 1. Spagettikoodi 3
KUVIO 2. Proseduraalisen koodin rakenne 4
KUVIO 3. Oliokeskeinen ohjelmointi 5
KUVIO 4. Luokkakaavio 6
KUVIO 5. Takaisinmallinnus ja ohjelmistokehitys 8
KUVIO 6. Aliohjelmakutsut ohjelmakirjastossa ja kehyksessä 15
KUVIO 7. Päätösmatriisi 16
KUVIO 8. Formula-koodi yhteen kirjoitettuna 20
KUVIO 9. Formula-koodi, jossa on kommentointi ja rivinvaihdot 20
KUVIO 10. Sivu LotusScript.doc-kuvauksesta 25
KUVIO 11. Kenttätarkistus 27
KUVIO 12. Tyhjän arvon tarkistaminen ja virheen näyttö 28
KUVIO 13. Tarkistus-luokan käyttö 29

1 JOHDANTO

Tässä opinnäytetyössä tutkin erilaisia menetelmiä, joita voi käyttää hyödyksi kirjoitettaessa tietokoneohjelmaa uudelleen käyttäen uudenlaista ohjelmointitapaa. Oliokeskeinen ajattelutapa on nykykäsityksen mukaan lähellä tapaa, jolla aivot hahmottavat asioita, ja oliokeskeinen suunnittelu- ja ohjelmointitapa kerääkin yhä enemmän suosiota. Yrityksissä on kuitenkin yhä aktiivisessa käytössä proseduraalisella ohjelmointitavalla aikoinaan kirjoitettuja ohjelmistoja, jotka ovat vuosien saatossa läpikäyneet valtavan määrän päivityskierroksia. Kun jälleen kerran tulee tarve ohjelmamuutokselle, voivat muutoksen tekemiseen kuluva aika ja vaiva ylittää saavutetun hyödyn, jos ohjelmakoodin hahmottaminen on hankalaa ja kokonaiskuvaa koko ohjelmistosta ei ole tiedossa kenelläkään alkuperäisen ohjelmoijankin jo mahdollisesti siirryttyä toisaalle.

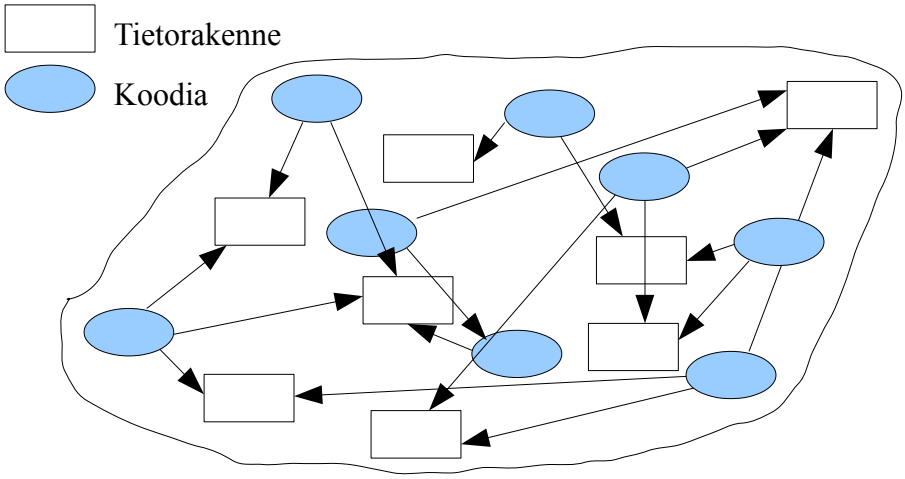
Proseduraalisella ohjelmoinnilla tarkoitan tässä ohjelmointitapaa, jossa ohjelman toiminnallisuus kirjoitetaan pääohjelmalohkoon ja aliohjelmiin. Ohjelmiston elinkaaren aikana useat ohjelmoijat tekevät muutoksia, jolloin laadultaan vaihtelevaa ohjelmakoodia kertyy suuria määriä. Jos vielä dokumentaatioiden ylläpito laiminlyödään, tulevat muutokset ovat yhä hankalampia. Oliokeskeisessä koodissa toki pystytään tekemään samat virheet, mutta tämä ohjelmointitapa tarjoaa myös keinoja välttää useita karikkoja. Oliokeskeinen ohjelmisto jakautuu tarkasti rajattua toimintaa toteuttaviin luokkiin ja näiden metodeihin. Ohjelmisto rakennetaan käyttämällä hyväksi olioita, jotka ovat luokkien ilmentymiä ja joita käytetään rajapintojen kautta, eli metodin sisäinen toteutus on piilossa. Luokkien ja itse metodien ohjelmakoodin määrä pidetään minimissään, jolloin niiden toiminta selviää nopeasti ja korjausten sekä uusien piirteiden lisäämisen vaikutukset voidaan kohtuullisella tarkkuudella ennakoida.

Ohjelmiston uudistamisprojektissa keskeiseen osaan nousee suunnittelutyö, jossa selvitetään ohjelmassa alun perin käytettyjä ratkaisuja sekä pyritään tunnistamaan oliorakenteita proseduraalisessa lähdekoodissa. Tulen esittelemään suositeltuja malleja, joilla ohjelmiston uudistaminen viedään läpi. Selvitän, millaisiin ongelmiin kannattaa varautua ryhdyttäessä ohjelmiston uudistamisprojektiin, jossa on tavoitteena helpompi ylläpidettävyys. Kaikissa tilanteissa ohjelmiston täydellinen uudelleenrakentaminen ei ole pakollista, vaan jo kevyemmät uudistukset tuovat hyötyjä ylläpitoon ja uudistusten tekoon. Tutkin opinnäytetyössäni myös näitä menetelmiä.

2 ERILAISET OHJELMOINTITAVAT

2.1 Spagettikoodi

Vuosikymmeniä sitten kirjoitetuissa ohjelmissa olennaisinta oli pienitehoisten prosessorien tehokas käyttö sekä mahdollisimman pienikokoiset ohjelmat, koska myös laitteiden muisti oli rajallista. Ohjelmointi tapahtui Assembly-kielellä, jolla muutetaan suoraan muistiosoitteiden tilaa. Aliohjelmaan päästiin siirtymään hyppykäskyllä, kun siihen ensin oli tallennettu paluuosoite. Kuvio 1 esittää, miten edestakaisin menevät viittaukset johtivat siihen, että koodin rakennetta pystyi parhaiten kuvaamaan sanalla spagetti. (Haikala & Märijärvi 2002, 304.)



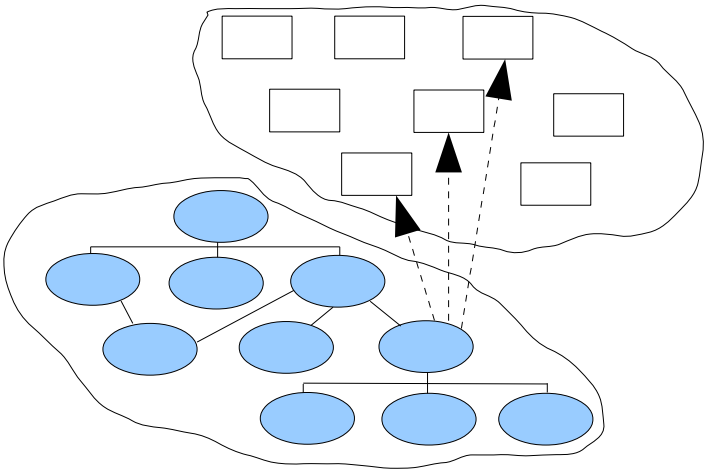
KUVIO 1. Spagettikoodi (mukaillen Haikala & Märijärvi 2002, 305).

2.2 Proseduraalinen ohjelmointitapa

1970-luvulle tultaessa muotiin tuli rakenteellinen eli strukturoitu (proseduraalinen) ohjelmointi. Tässä ohjelmointitavassa ositetaan ohjelman toiminnot aliohjelmahierarkioihin sen mukaan, mihin pienimpiin yhteisiin tekijöihin ne voidaan jakaa. Aikaisemmin yleisessä käytössä olleet hyppykäskyt saatettiin kieltää koko-

naan. Mikäli ohjelman tietorakenteet ovat globaaleja, niihin voidaan viitata mistä ohjelman osasta tahansa. Niinpä kun tietorakenteeseen tulee muutos, täytyy vastaava muutos tehdä myös jokaiseen viittaavaan kohtaan. (Haikala & Märijärvi 2002, 305–306.)

Tavallisimpia rakenteita proseduraalisessa ohjelmassa ovat pääohjelmalohko ja joukko aliohjelmia, joihin pääohjelmasta viitataan (KUVIO 2). Aliohjelmat toteuttavat ohjelman toimintalogiikan. Lisäksi esimerkiksi C-kielessä käytetään tietojen säilyttämiseen tietue (struct) -tyyppistä rakennetta.

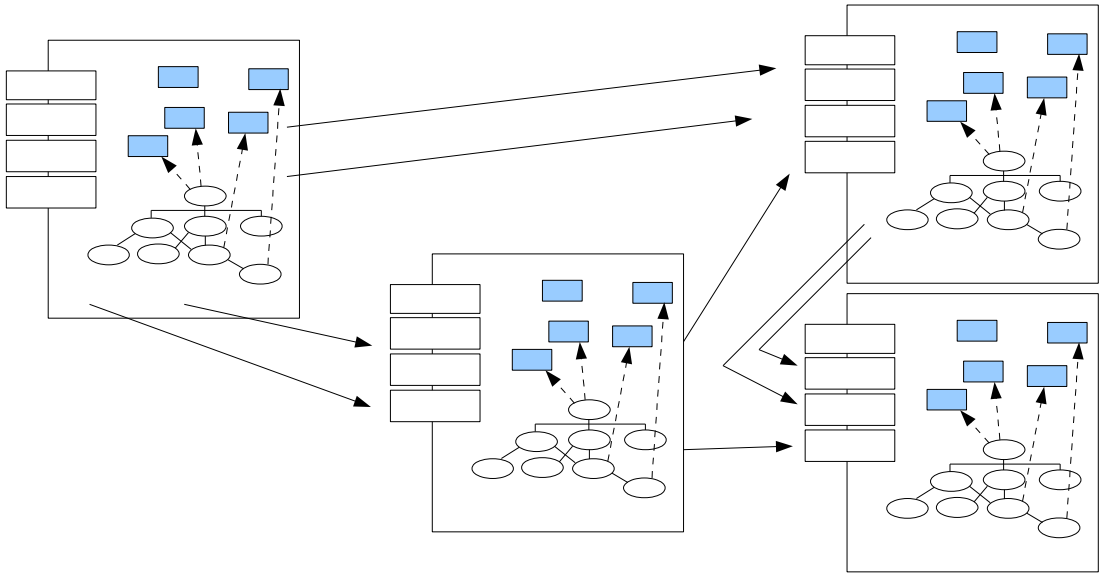


KUVIO 2. Proseduraalisen koodin rakenne (mukaillen Haikala & Märijärvi 2002, 305)

Ohjelmointitavassa on edellytykset toimintojen nopeaan muuttamiseen ja lisäämiseen. Aliohjelmat voivat kuitenkin paisua suuriksikin ja rakenne evoluutiosyklien myötä huonontua niin, että ylläpito käy yhä hankalammaksi. Globaaleja tietorakenteita voidaan muuttaa ja lukea useissa aliohjelmissa, minkä vuoksi muutosten hallinta on vaikeaa.

2.3 Oliokeskeinen ohjelmointi

Oliokeskeisessä ohjelmoinnissa luokkamoduuliin kapseloidaan tietorakenne ja toiminnallisuus. Näille piirteille on olio-ohjelmoinnissa erityiset nimityksensä: määrättyä luokkaa ilmentävässä oliossa tieto varastoidaan attribuutteihin jäsenmuuttujien avulla ja toiminta toteutetaan käyttäen metodien kutsuja parametreineen. (Haikala & Märijärvi 2002, 348–349.) Kuvio 3 esittää, miten oliosuuntautuneessa ohjelmassa toiminta rakentuukin luokkien välisiä rajapintoja käyttäen.



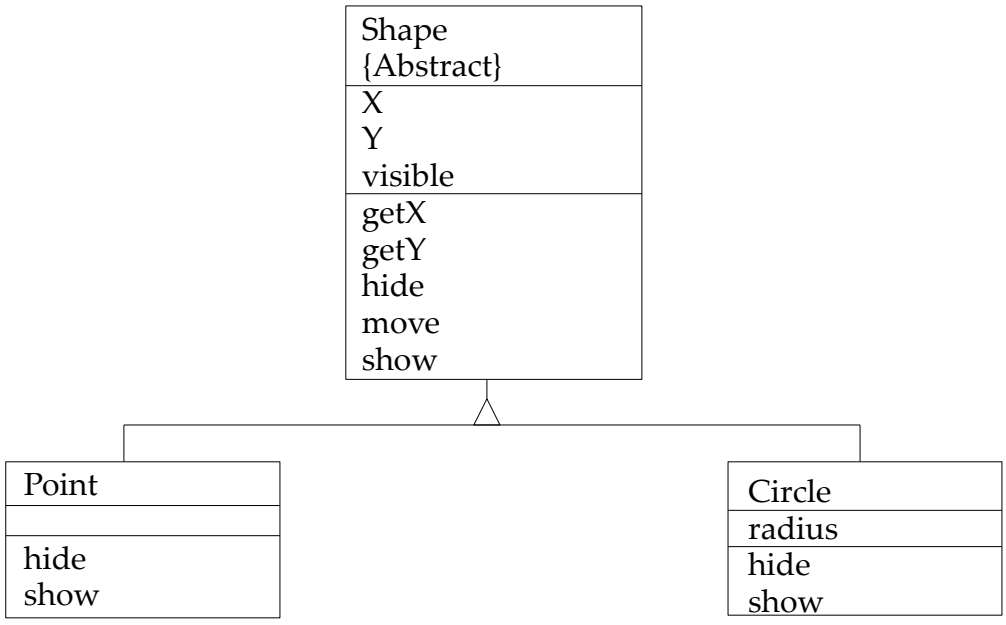
KUVIO 3. Oliokeskeinen ohjelmointi (Haikala & Märijärvi 2002, 305.)

Erityisiä piirteitä oliokeskeiselle ohjelmointitavalle ovat muun muassa luokan metodien periytyminen, metodien ylikuormittaminen sekä monimuotoisuus. Kun uusi luokka aloitetaan, se voidaan rakentaa aiemman luokan perustalta. Näin on tehty kuvion 4 erilaisia muotoja kuvaavassa luokkakaaviossa. Nyt luokka, josta perittiin, on yliluokka ja uusi luokka on sen aliluokka. Aliluokka on perinyt kaikki yliluokalla olevat piirteet, ja niiden toimintaa voidaan muuttaa tai lisätä uusia. (Haikala & Märijärvi 2002, 350). Kun luokalla on saman nimisiä metodeja, joiden toiminta on erilainen parametrien tyyppien tai lukumäärän mukaan, puhutaan ylikuormitetuista metodeista (Kendal 2009). Monimuotoisuus liittyy läheisesti periy-

tymiseen: Kun muuttujan arvona on luokka, se voidaan korvata joko tätä alkuperäistä luokkaa ilmentävällä uudella muuttujalla tai sitten millä tahansa alkuperäistä aliluokkaa ilmentävällä oliolla (Koskimies 2000, 96).

Perusteellinen suunnittelutyö on perusta onnistuneelle olio-ohjelmalle. Monesti analyysi- ja suunnitteluvaiheet kestävät kauemmin kuin itse ohjelmointi. Osaltaan tähän vaikuttaa koodin uudelleenkäyttömahdollisuuksien huomiointi. Toisaalta perusteellinen suunnittelutyö myös vaikuttaa laadukkaamman koodin syntymiseen, sillä lopulta käytettäväksi valitut ratkaisut ovat perusteltuja. Tähän sopiikin sanonta "hyvin suunniteltu on puoliksi tehty".

Hyviä työvälineitä oliosuunnitteluun ovat erilaiset UML-kaaviot, kuten kuvion 4 erilaisia muotoja esittävä luokkahierarkia. Pää- ja aliluokkien hierarkkista rakennetta kuvaavassa luokkakaaviossa (Class diagram) kutakin luokkaa esittävä laatikko sisältää luokan nimen, sen attribuutit ja metodit. UML-standardi on hyvin joustava, ja luokkakaaviossa pakollista tietoa onkin ainoastaan luokan nimi; muita tietoja voi yksinkertaisen esityksen saavuttaakseen jättää kaaviosta pois. (Kendal



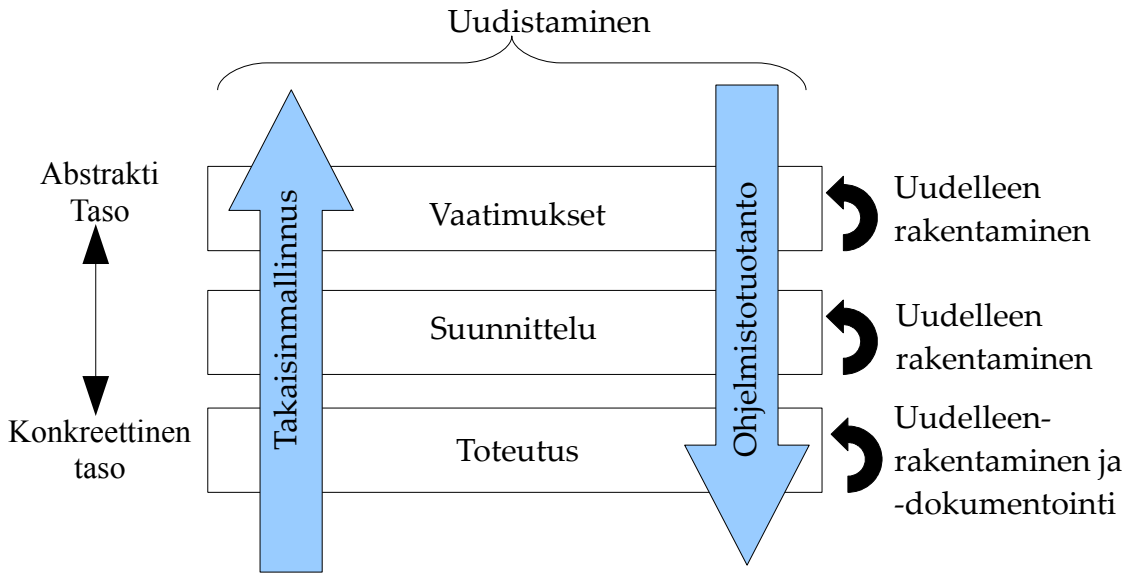
KUVIO 4. Luokkakaavio (Haikala & Märijärvi 2002, 350)

2009.) Luokan voi merkkiarvolla {Abstract} ilmoittaa soveltuvan ainoastaan pääluokaksi, jonka perusteella luodaan aliluokkia (Haikala & Märijärvi 2002, 350). Pääluokasta periytyminen esitetään kuvion 4 alaluokista pääluokkaan osoittavalla tyhjällä nuolella.

3 TAKAISINMALLINNUS

Takaisinmallinnusta (reverse engineering) käyttämällä muodostetaan käsitys siitä, miten ohjelman osat liittyvät toisiinsa ja millaisia suunnitteluratkaisuja ohjelmassa on käytetty. Takaisinmallinnuksessa ei kirjoiteta uutta ohjelmaa alkuperäisen pohjalta eikä muuteta vanhaa, vaan se toimii apuvälineenä ohjelman ymmärtämisprosessissa paljastamalla vaiheittain, miten ohjelmoija on päätynyt käyttämiinsä ratkaisuihin. Tässä prosessissa pystytään täydentämään ohjelman dokumentaatiota ja ennen kaikkea se saadaan vastaamaan ohjelman nykytilaa. (Harsu 2003, 120.)

Terminä takaisinmallinnus pohjautuu laitteistopuolelle, jossa saatetaan purkaa kilpailijan tuotteita osiin tavoitteena niiden toimintalogiikan selvittäminen, kun tuotteen suunnittelu- ja valmistusdokumentteja ei ole saatavilla. Tietokoneohjelmistojen ollessa kyseessä voidaan takaisinmallinnuksen ajatella olevan käänteistä ohjelmistokehitystä. Kuvio 5 havainnollistaa ohjelmistokehityksen ja takaisinmallinnuksen välistä samankaltaisuutta. Ohjelmistokehityksessä tyypillisesti laaditaan vaatimusmäärittely, käyttötapaukset, dokumentoidaan eri vaiheissa tehdyt muu-



KUVIO 5. Takaisinmallinnus ja ohjelmistokehitys (Harsu 2003, 23)

tokset ja niin edelleen, kunnes ohjelma on saatu valmiiksi. Nyt työ alkaakin valmiista ohjelmasta, jota analysoidaan selvittäen suunnitteluratkaisut ja täydentäen dokumentaatiota. (Harsu 2003, 22–23.)

3.1 Ohjelman ymmärtäminen

Jotta ohjelmoija tietäisi, millaisen ohjelman kanssa hän on tekemisissä, hänen täytyy selvittää sen toimintaa korkeammalla abstraktiotasolla. Hän hyödyntää tietämystä käytetystä ohjelmointikielestä, sovellusaluetietämystä sekä ymmärtämisstrategioita, joilla saadaan käsitys ohjelmasta. Selvityksen tekoon on kolme menetelmää: kokoava malli, osittava malli sekä tilanteen mukainen malli.

- Kokoavassa mallissa tietämys kootaan analysoimalla lähdekoodista koodirakenteita, kuten silmukka- ja ehtolauseita sekä algoritmeja, jotka yhdessä muodostavat koodikokonaisuuksia.
- Osittavassa mallissa hyödynnetään sovellusalueen tietämystä sekä etsitään ohjelmakoodista tunnettuja rakenteita ja esimerkiksi aliohjelman nimistä päätellen muodostetaan yhä tarkempi käsitys ohjelmasta. Ohjelmoija tarkentaa muodostamaansa alkuolettamusta, jonka on saanut esimerkiksi dokumentaatiota tutkimalla.
- Tilanteen mukaisessa mallissa yhdistetään piirteitä kokoavasta ja osittavasta mallista. (Harsu 2003, 106–108.)

Ohjelman ymmärretyksi tulemisessa vähintään yhtä tärkeitä kuin ohjelmakoodin osat ovat kommenttirivit sekä muuttujien ja aliohjelmien nimeäminen. Näiden merkityksen huomaa varsinkin, kun tutkittavaksi tulee ohjelma, josta puuttuvat kommentit täysin ja nimeämiskäytäntökin on epäjohdonmukaista. (Harsu 2003, 113–114.)

3.2 Uudelleendokumentointi

Uudelleendokumentoinnissa kirjoitetaan tai täydennetään keskeisiä ohjelman toimintaan ja ylläpitoon liittyviä dokumentteja käyttäen pohjana ohjelman nykytilaa. Aina ohjelmiston nykytilaa kuvaavaa dokumentaatiota ei ole saatavilla. Vaikka sellainen vaikuttaisi löytyvänkin, kannattaa dokumentaatio käydä tarkasti läpi, sillä yleisimmin ohjelmiston dokumentaation ylläpito on jossain elinkaaren vaiheessa laiminlyöty. Uudelleendokumentointia on jo kommenttien lisääminen ohjelmakoodin lomaan tai pseudokoodin tuottaminen ohjelmakoodista. Ohjelmaa analysoitaessa voidaan esimerkiksi tuottaa graafinen esitys aliohjelmien suhteista toisiinsa. (Harsu 2003, 122–123.)

3.3 Suunnittelumallien tunnistaminen

Ohjelmasta pystytään tunnistamaan ohjelmointiympäristölle tai asiakasrajapinnalle tyypillisiä ohjelmointiratkaisuja ja suunnittelumalleja, joita voidaan jopa sellaisenaan käyttää osana oliokeskeistä ohjelmaa. Ohjelmoija on saattanut tietämättäänkin käyttää jotakin suunnittelumallia tekemissään valinnoissa, ja takaisinmallinnuksen suorittajan työ helpottuu, kun hänellä on tietämystä eri tilanteisiin liittyvistä suunnittelumalleista. Kokenut ohjelmoija osaa jo suunnitteluaineistoon tutustuttuaan ennakoida, millaisia ratkaisuja ohjelmakoodista todennäköisesti löytyy. Siten hän osaakin etsiä haluamaansa tietoa esimerkiksi aliohjelmien nimistä ja valintarakenteista. (Harsu 2003, 160–161.)

Suunnittelumallit ovat ohjelmointikielestä riippumattomia yleiskäyttöisiä, hyväksi havaittuja ratkaisuja yleisiin ohjelmistoteknisiin ongelmiin. Olio-ohjelmoinnissa suunnittelumalli käsittää yleensä muutamia luokkia ja niiden välisiä suhteita. (Koskimies 2000, 245.) Monesti ohjelmoijalla löytyy mallit tyypillisimpiin esiin tu-

leviin tilanteisiin ulkomuistista, mutta hyödynnettävissä olevaa lähdekirjallisuutta löytyy myös runsaasti sekä painettuna että sähköisessä muodossa. Myös tietämys vastamalleista eli ei-suositelluista ratkaisuista sekä näiden löytäminen dokumenteista tai ohjelmakoodista auttavat ennustamaan koko toteutuksen laatua. (Harsu 2003, 162; Koskimies 2000, 247.)

Kun suunnittelumallia käytetään ja nimetään siihen kuuluvat osat vakiomuotoon, muut ohjelmoijat ymmärtävät tämän yhteisen kielen. Näin muutostyöhön ryhtyminen on suoraviivaisempaa silloinkin, kun itse ohjelma on muutosta tekevälle ennestään tuntematon. (Koskimies 2000, 246.)

4 OLIOINTI

4.1 Oliopiirteiden tunnistaminen proseduraalisesta koodista

Oliopiirteitä voi tunnistaa koodista globaalien muuttujien perusteella turvautuen oletukseen, että jokaiseen globaaliin muuttujaan viitataan yhdessä tai useammassa aliohjelmassa, mutta ei kuitenkaan kaikissa. Muuttujista ja niitä käyttäneistä aliohjelmista muodostetaan verkko, jossa aliohjelmien yhteydet merkitään viivoilla ja samoihin globaaleihin muuttujiin viittaavat aliohjelmat merkitään kaarilla. Jokainen globaalin muuttujan ja siihen liittyvät aliohjelmat sisältävä solmu vastaa luokkaa ja sen metodilistaa. (Harsu 2003, 145–148.)

Joskus aliohjelmat ovat niin tiiviitä, että globaaleihin muuttujiin perustuvalla olioinnilla saadaan vain yksi iso rakenne. Tällöin aliohjelmien välillä todennäköisesti on ei-toivottuja yhteyksiä eli esimerkiksi aliohjelmilla oleva yhteinen alustusohjelma. Verkkokuvauksessa aliohjelmia siis yhdistetään kaarella. Jos aliohjelmille olisi rakennettu erilliset alustusohjelmat, ne olisi kuvauksessa esitetty erillisinä komponentteina. Aliohjelmia voidaan jakaa osiin, kunnes ei-toivottujen yhteyksien määrä pienenee ja voidaan muodostaa sopivan kokoisia komponentteja (Harsu 2003, 145–148). Osittaminen voidaan tehdä vaikkapa viipalointia käyttämällä, jolloin koodista tarkastellaan vain sellaisia rivejä, jotka vaikuttavat haluttujen muuttujien arvoon (Harsu 2003, 47).

4.2 Kääriminen

Kevyt uudistamistapa on uudistaa pelkkiä ohjelmointirajapintoja. Käärimisen kautta saadaan järkevästi määritellyn rajapinnan avulla käyttöön sellaiset aliohjel-

mat, joita ei luontevasti voida olioita tunnistamalla muuttaa oliokielisiksi. Käärimistä käytettäessä ohjelman rakenteellinen laatu pysyy ennallaan, mutta metodien kutsut pystytään muuttamaan oliopohjaisiksi. Koska kääritty ohjelmakoodin ylläpito on monesti hankalaa, toimii kääriminen parhaiten väliaikaisena ratkaisuna sekä ohjelman elinkaaren loppuvaiheessa tuottamassa yhteyksiä oliopohjaisten ohjelmistojen kanssa. (Harsu 2003, 232–233.)

Käärittäväksi voidaan valita aliohjelmia tai suurempia kokonaisuuksia, kuten ohjelmia, alijärjestelmiä tai kokonaisia järjestelmiä. Aliohjelmaa pienempien osien kääriminen sen sijaan ei ole järkevää. Aliohjelman käärimisessä voidaan muodostaa luokka, jonka ainoa metodi tuottaa aliohjelman toiminnan. Mahdollista on myös aliohjelman käyttö osana jonkin luokan metodin toimintaa. Kokonaisen ohjelman käärimisessä tulee kiinnittää huomiota siihen, että syöttö- ja tulostustoiminnot ovat käytettävissä myös käärimisen jälkeen. Laitteet voidaan mallintaa olioiksi, joita kääritty ohjelma käyttää hyödykseen. Vaihtoehtoisesti käyttöliittymän voi muuttaa graafiseksi ja rakentaa syöttötoiminnot uuden käyttöliittymän olioiksi. Alijärjestelmän käärimisessä syöttö- ja tulostusrajapinnat otetaan mukaan samalla tavalla, mutta lisäksi mukaan kääritään alijärjestelmän käyttämät tietovarastot. Mikäli muut alijärjestelmät käyttävät samoja tietovarastoja kääritty alijärjestelmän kanssa, tulee niihin muuttaa uuden rajapinnan mukaiset kutsut. (Harsu 2003, 234–238.)

5 LUOKKAKIRJASTOT

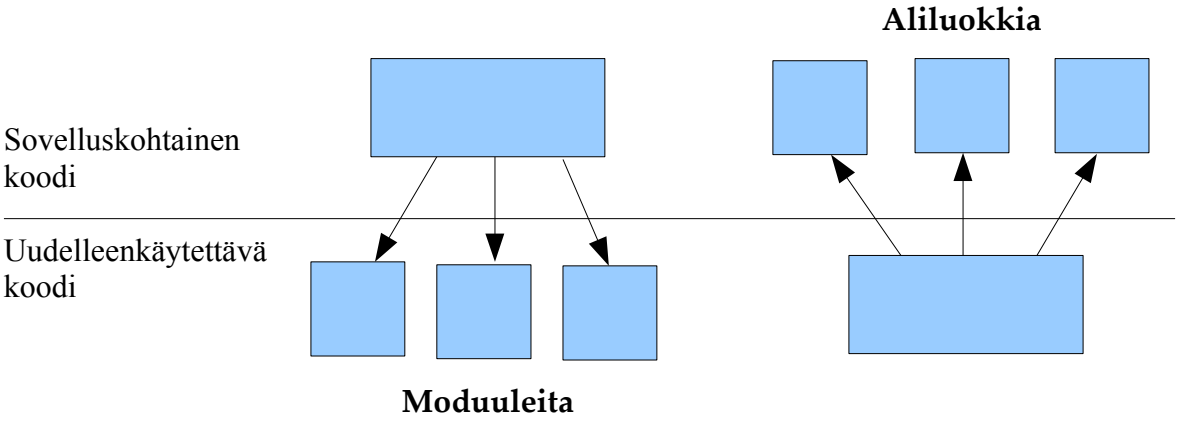
Proseduraalisessa ohjelmoinnissa tavallinen uudelleenkäytettävän koodin muoto on käyttää aliohjelmakirjastoja sovelluksen rutiinien suorittamiseen. Aliohjelmia käyttämällä ohjelmien koko saadaan pienemmäksi, mikä on ollut tärkeää varsinkin aiemmin laitteiden vähäisen muistin ja tehon vuoksi. (Koskimies 2000, 260.)

Tiiviisti yhteen kuuluvat luokkahierarkiat voi koota luokkakirjastoon, joka ideana muistuttaa paljon aliohjelmakirjastoa. Luokkakirjastoon otettaan mukaan hyvin testattua, uudelleenkäytettävää koodia. Käyttäjä ilmentää näiden luokkien pohjalta oliot ja käyttää luokkien metodeja toimintojen suorittamiseen. (Koskimies 2000, 260.)

Laajempien luokkakirjastojen kohdalla puhutaan sovelluskehyksestä, jos siitä erikoistamalla voidaan luoda uusia sovelluksia. Kehyskomponentista on kyse silloin, kun kehys tarjoaa tavan tuottaa komponentteja. Termin erikoisuus johtuu siitä, että komponenttikehys on jo vakiintunut termi. Komponenttikehys tarkoittaa sellaista kehystä, joka toteuttaa komponenttien tarvitseman infrastruktuurin. Tällaiset kirjastot on laadittu pitäen uudelleenkäytettävyys ja sovellusrajapinnat mielessä. Erikoistamisrajapinta eli tapa, jolla sovellus voidaan kehyksestä erikoistaa, ilmoitetaan kehyksen dokumentaatiossa. (Koskimies 2000, 260.)

Tavallinen muoto oliopohjaisen kehyksen arkkitehtuurissa on niin sanottu muunneltava kehys (white-box framework). Kehyksessä olevista luokista periyttämällä ja metodeja uudelleen määrittelemällä saadaan aikaan tila, jossa kehys kutsuukin tätä periyttäjän luomaa koodia Hollywood-periaatteella ”don't call us, we'll call

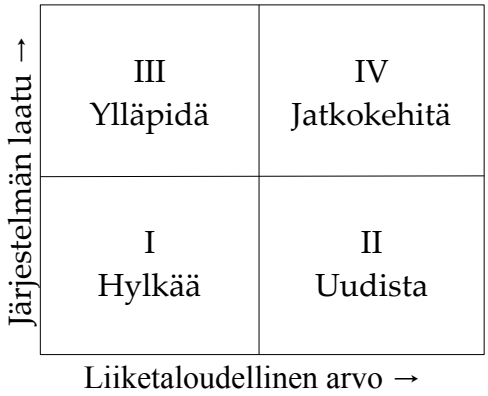
you”. (Koskimies 2000, 260.) Kuviossa 6 havaitaan ero aliohjelmakirjaston ja kehyksen välillä.



KUVIO 6. Aliohjelmakutsut ohjelmakirjastossa ja kehyksessä (Koskimies 2000, 261)

6 MUUTOSTARPEEN ARVIOINTI

Ohjelman täydelliselle uudelleenrakentamiselle voidaan löytää hyviä perusteluja, mutta joskus kevyemmätkin ratkaisut tulevat kysymykseen muun muassa sen mukaan, kuinka usein ohjelmaan on odotettavissa muutoksia ja kuinka keskeisessä osassa yrityksen liiketoimintaa ohjelma on. Toisissa tapauksissa saattaa jo muutamien aliohjelmien uudistaminen tehdä muutosten teon tulevaisuudessa joustavammaksi ja paremmin hallituksi. Joskus taas on järkevää aloittaa laajempi uudistamisprojekti, jolla ratkaistaan kerralla useita päivityksiin ja yleiseen käyttöön liittyviä ongelmia (KUVIO 7). (Harsu 2003, 74–75.)



KUVIO 7. Päätösmatriisi (Harsu 2003, 75)

Kuviossa 7 esitettävää päätösmatriisia voidaan hyödyntää arvioitaessa ohjelman muutostarvetta ja muutostyön laajuutta. Vasemmassa alakulmassa on ohjelmisto, joka on huonolaatuinen ja liiketaloudelliselta arvoltaan vähäinen. Tällainen ohjelmisto on viisainta hylätä. Oikeassa alakulmassa taas on yritykselle liiketaloudellisesti tärkeä ohjelma, joka kuitenkin on huonosti rakennettu. Tämä aiheuttaa lisäkustannuksia ylläpidossa, ja se onkin siksi järkevää uudistaa tai korvata pikimmin. Ylävasemmalla on hyviä menetelmiä noudattaen rakennettu helposti ylläpidettävä ohjelma, josta ei ole suuria liiketaloudellisia hyötyjä. Tätä järjestelmää ei kannata

korvata uudella, vaan sen ylläpitoa voi jatkaa tai sitten päättää luopua koko järjestelmästä. Viimeisessä neljänneksessä ovat helposti ylläpidettävät ohjelmat, jotka hyödyttävät liiketoimintaa. Näiden normaalia ylläpitoa kannattaa jatkaa, ja tulevaisuuden jatkokehitystarpeetkin sujuvat mutkattomasti. (Harsu 2003, 74–75.)

Liiketoiminnallisen arvon määrittämiseen saa tarvittavaa tietoa eri lähteistä. Mahdollisimman laaja-alaiset lähteet ovat tarpeen. Loppukäyttäjät voivat kertoa, mitä järjestelmän toimintoja he useimmin käyttävät ja mitä toimintoja he eivät käytä koskaan. He voivat myös kertoa, kuinka hyödyllisinä he eri toimintoja pitävät. Asiakkaat taas voidaan selvittää, tuottavatko jotkin järjestelmän piirteet hankaluuksia, kuten taukoja työskentelyyn. Yritysjohdolla on tieto järjestelmän aiheuttamista kustannuksista sekä siitä, miten helppoa on löytää henkilöstöä ylläpitämään järjestelmää. (Harsu 2003, 75–76.)

Järjestelmän laatua arvioitaessa painoarvoa on useilla tekijöillä.

- Järjestelmän ymmärrettävyyteen vaikuttavat muun muassa kuvaavat toimintojen nimet.
- Arkkitehtuuria tutkimalla voidaan selvittää järjestelmään muutosten tekemisen helppoutta ja vaikeutta.
- Dokumentaation ollessa saatavilla arvioidaan, miten hyvin se on ajan tasalla.
- Tietomallia tarkastelemalla havainnot siitä, onko käytetty yhtenäistä tietomallia vai ovatko tiedot hajallaan, tarjoavat hyödyllistä tietoa kokonaislaadusta.
- Järjestelmän suorituskyky ja mahdollisen tehottomuuden vaikutus käyttöön ovat merkille pantavia asioita.
- Käytetyn ohjelmointikielen osaaminen voi joskus olla hankalasti saatavissa.

Lisäksi on tärkeää selvittää, onko ohjelmoijia, joilla on kokonaiskäsitys ohjelmasta ja riittävästi tietämystä sen ylläpitämiseen. (Harsu 2003, 76.)

7 DOMINO-SOVELLUKSEN OHJELMOINTI

IBM Lotus Notes/Domino on työryhmäjärjestelmä, jossa eri sovellusten avulla voidaan suorittaa liiketoimintaa hyödyttäviä toimintoja. Sovellukset suoritetaan yleensä Domino-serveriltä, joskin replikoinnin avulla käyttö onnistuu myös ilman verkkoyhteyttä. Sovellusta voidaan suorittaa Lotus Notes -työpöytäohjelmalla, internetselaimen kautta tai mobiililaitteella. Itse ohjelmointi eri ympäristöille on hyvin samankaltaista, ja esitänkin ohjelmointinäytteet ainoastaan työpöytäohjelmistolle.

7.1 Ohjelmoitavia kohteita Domino-sovelluksessa

Sovelluksessa kaikki tieto tallennetaan dokumenteille, joita voidaan valita ja lajitella näkymillä. Tässä onkin merkittävä ero relaatiotietokantoihin, joissa puolestaan käsitellään tallennettujen tietojen keskinäisiä yhteyksiä. Näkymältä avattaessa dokumentti esitetään käyttäen sille määritettyä lomaketta, joka sisältää tietokenttiä, vakiotekstejä, alilomakkeita, taulukoita, kuvia, painikkeita ynnä muuta. Tyypillisessä sovelluksessa on useita lomakkeita, jotka eroavat toisistaan sisältämiensä kenttien puolesta, mutta myös sen mukaan, onko ne tarkoitettu näytettäväksi Notes-ohjelmistossa, www-sivuna vai mobiililaitteella. Muokkaustilassa lomakkeen tietokenttiin tallentuu käyttäjän syöttämä sekä ohjelmallisesti tuotettu tieto. Kenttien eri tilat voivat myös vaikuttaa siihen, mikä alilomake käyttäjälle milloinkin annetaan. (Tulisalo, Carlsen, Guirard, Hartikainen, McCarthy & Pecly 2002.)

Agentit ovat tärkeä lisä lähes kaikissa sovelluksissa. Niillä voidaan suorittaa suureen dokumenttimäärään yhtäaikaiset muutokset tai tehdä avattuun tai näkymällä valittuun dokumenttiin liittyviä toimia. Agentti voidaan myös kytkeä erilaisiin so-

velluksen tilatapahtumiin, jolloin agentti voidaan esimerkiksi suorittaa aina, kun tietokantaan on luotu uusia dokumentteja. Www-sivulla käytettynä agentit saavat käyttöönsä CGI-ympäristön muuttujat. Agentteja voidaan ohjelmoida niin formula- tai LotusScript-kielellä kuin Javaakin käyttämällä. (Tulisalo ym. 2002.)

7.2 Ohjelmointikielet

Notes-sovelluksen sovelluskehitykseen on tarjolla laaja valikoima ohjelmointikieliä. Eri ohjelmointikielet soveltuvat paremmin sovelluksen eri kohdissa käytettynä, kuten seuraavien lukujen kuvauksista havaitaan.

7.2.1 JavaScript

Javascript on www-maailmassa suosittu ohjelmointikieli. Domino-sovelluksessa Javascriptiä voidaan käyttää paitsi käyttöliittymätoimintoihin www-dokumentilla, myös esimerkiksi Notes-työpöytäohjelmiston dokumenttien kenttätarkistuksissa. Ainoastaan sivulla saatavilla oleva tieto on Javascript-ympäristön käytettävissä. Sen ulottumattomissa ovat siis esimerkiksi toisessa tietokannassa olevat näkymät.

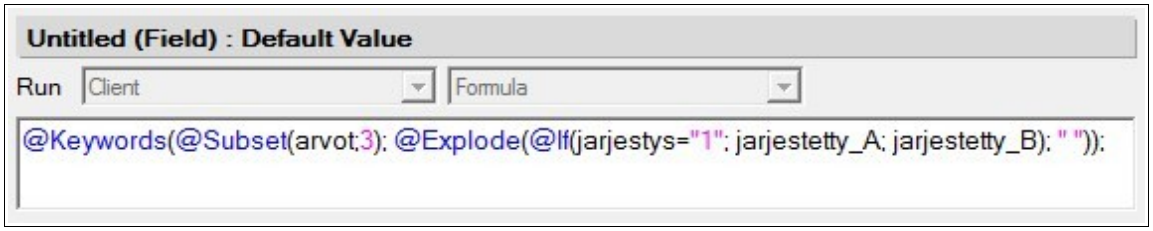
7.2.2 Formula-kieli

Formula-kielessä käytettävät @Function- ja @Command-komennot ovat tehokas keino lisätä pieniä käyttöliittymätoimintoja. Näitä komentoja käytetään tyypillisesti kentän oletusarvojen määrittämisessä, kenttäarvon tarkistuksessa tai painiketoiminnoissa. Myös näkymän sarakearvot voidaan tuottaa Formula-komennoilla. Komennot voivat käyttää kaikkia dokumentin kenttäarvoja toiminnoissaan. Formula-kielellä voidaan aikaansaada hyvin kompakteja komentoja. Kahdella

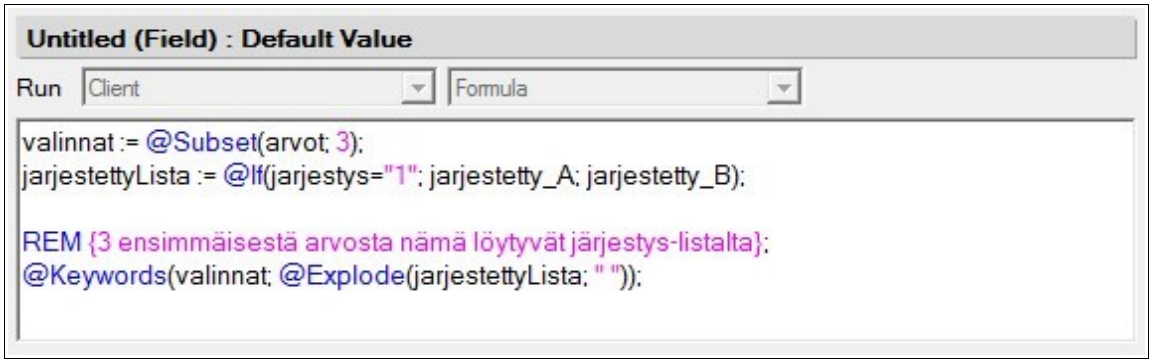
@Function-komennolla voidaan poistaa kahteen kertaan esiintyvät nimet listalta sekä korvata nimissä esiintyvät peräkkäiset välilyönnit yhdellä: @Rem {Dokumentin tallennus}; @Trim(@Unique(nimiLista)). Saman toiminnon suorittaminen LotusScript- tai Java-kielellä vaatisi useita komentorivejä koodia. (Tulisalo ym. 2002.)

@Command-komennot jäljittelevät Lotus Notes -järjestelmän valikkotoimintoja. Painiketoiminnossa @Command([FileSave]) suorittaa dokumentin tallennuksen. (Tulisalo ym. 2002.)

Komentoja voi käyttää sisäkkäin toisten komentojen arvoina. Näin toimimalla formula-kielellä kirjoitetut rakenteet voivat muodostua hyvinkin vaikeasti ymmärrettäviksi. Toiminnon luettavuus alkaakin kärsiä, ellei rivinvaihtoihin ja kommentteihin kiinnitä erityistä huomiota, kuten havaitaan kuvioiden 8 ja 9 välillä. Kommentointi tapahtuu REM-lauseilla.



KUVIO 8. Formula-koodi yhteen kirjoitettuna



KUVIO 9. Formula-koodi, jossa on kommentointi ja rivinvaihdot

Osa komennoista, esimerkiksi @UserRoles, on muita raskaampi suorittaa. Tällaisten komentojen kohdalla suositellaankin arvon tallentamista ainoastaan näytön aikana suoritettavaan kenttään (Computed for Display). Tarvittavissa paikoissa saadaan vastaava tieto nyt kentän nimellä, johon haku on kuitenkin suoritettu vain yhden kerran. (Tulisalo ym. 2002.)

7.2.3 LotusScript

LotusScript on Lotus Notes -järjestelmän kehitykseen käytettävä ohjelmointikieli, jossa on olio-ohjelmoinnin piirteitä. Syntaksiltaan kieli muistuttaa uudempia BASIC-pohjaisia kieliä kuten Visual Basic. (Tulisalo ym. 2002.)

LotusScript-kielessä on valmiudet olio-ohjelmointiin, mutta joitain keskeisiksi ajateltavia piirteitä kielestä myös puuttuu. Metodien ylikuormittaminen ei ole mahdollista, vaan metodinimestä voi esittää vain yhden version. (Perry 2001.) Ylikuormittamisella tarkoitetaan sitä, että saman niminen metodi saadaan käyttöön eri parametrimäärällä tai erilaisilla parametrin tyypeillä, jolloin metodin toiminta muuttuu näiden mukaisesti (Koskimies 2000, 337).

Muokattavana olevan dokumentin tallennus tapahtuu LotusScriptiä käyttämällä seuraavan ohjelmointiesimerkin mukaan.

```
Sub Querysave(Source As Notesuidocument, Continue As Variant)
    If ( source.FieldGetText( "Title" ) = "" ) Then
        Messagebox( "You must enter a title." )
        Call source.GotoField( "Title" )
        Continue = False
    End If
End Sub
```

(IBM Corporation 2007.)

Esimerkissä aliohjelmalla on käytössään käyttöliittymätason dokumentti (NotesUIDocument Source) ja tieto siitä, jatketaanko tallennusta tämän aliohjelman jälkeen (totuusarvo Continue). Mikäli arvoa Continue ei aliohjelmassa muuteta, lomakkeen muutokset tallentuvat dokumentille. Esimerkissä on myös kenttävalidointi otsikkokentälle Title. Jos siis tämä kenttä on tallennushetkellä tyhjä, käyttäjälle näytetään virheilmoitus eikä tallennusta suoriteta loppuun, ennen kuin tilanne on korjattu ja valittu tallennus uudelleen.

7.2.4 Java

Java on nykyään keskeisimmässä osassa oleva ohjelmointikieli, ja sitä voi käyttää myös Notes-sovelluksen ohjelmoinnissa. Javalle on tarjolla samankaltainen ohjelmointirajapinta kuin LotusScriptille – usein luokkien erona on ainoastaan alkuun lisättävä lotus.domino. (Tulisalo ym. 2002.) Javan käyttöön liittyy kuitenkin sen kaltaisia suoritusaikahaasteita, että sen käyttö on tyypillisesti suositeltavampaa esimerkiksi agenteissa. Toinen tyypillinen käyttökohde on WebService-rajapinnassa, jonka tyypillinen käyttökohde on kahden erilaisen järjestelmän välinen kommunikointi käyttäen standardin mukaisia XML-sanomia. (Tulisalo ym. 2002.)

7.2.5 Muita kieliä

C++-kielisille ohjelmille on saatavilla API, joka mahdollistaa samankaltaisten toimintojen rakentamisen kuin valmiina tulevaa Notes APIa käyttämällä. Näillä ohjelmilla voi tehdä käyttöliittymään uusia työkaluja tai vaikka uuden tallennustoiminnon tyypin eli tiedostopäätteen. (Tulisalo ym. 2002)

XML-kuvauskielellä tuotettuja dokumentteja käyttämällä voidaan viedä ja tallentaa mitä tahansa tietokannan elementtejä tai dokumentteja tietokannan sisällä. Nii-

tä voidaan myös käyttää toteuttamaan eri järjestelmien välistä kommunikaatiota. (Tulisalo ym. 2002.)

7.3 Koodikirjastot

Domino-sovelluksessa on hyvät mahdollisuudet koodin uudelleenkäytölle. Sovelluskehittimessä jaetun koodin voi sijoittaa kohtaan Shared Code – Script Library (koodikirjasto). Koodikirjasto voi olla kirjoitettu JavaScript-, LotusScript- tai Java-kielellä. Toisen koodikirjaston voi ottaa mukaan käyttämällä Use-lausetta, jolloin liitetyn ja kaikkien siihen aikaisemmin liitettyjen kirjastojen koodi on uuden kirjaston käytettävissä. LotusScript-kirjaston Options-osa näyttää seuraavalta, kun kirjasto on aloitettu toisen kirjaston pohjalta:

```
Option Declare
Use ”paaLuokat”
```

Esimerkissä oleva lause Option Declare tarkoittaa, että sovelluskehitin tarkistaa jo tallennusvaiheessa, että kaikki ohjelmassa käytetyt muuttujat on esitelty. Muussa tapauksessa käyttäjä saa varoituksen eikä tallennus onnistu. Option Declare -lauseen käyttö on suositeltavaa, koska sen avulla voi välttää hankalasti jäljitettävät ongelmat, jotka johtuvat muuttujan tyypistä. (Tulisalo ym. 2002.)

Koodikirjastossa voi olla perinteistä proseduraalista koodia, jolloin aliohjelmat kirjoitetaan Function- tai Sub-lauseita käyttäen. Myös oliosuuntautunut koodi on mahdollista – se taas kirjoitetaan kokonaisuudessaan kirjaston Declarations-osaan.

Joskus on tarpeen piilottaa esimerkiksi liikesalaisuuden perusteella osa koodista. Koodikirjaston koodin voi piilottaa tallentamalla valmiin kirjaston levylle ja korvaamalla kirjoitetun koodin viittauksella tallennettuun tiedostoon Options-osassa:

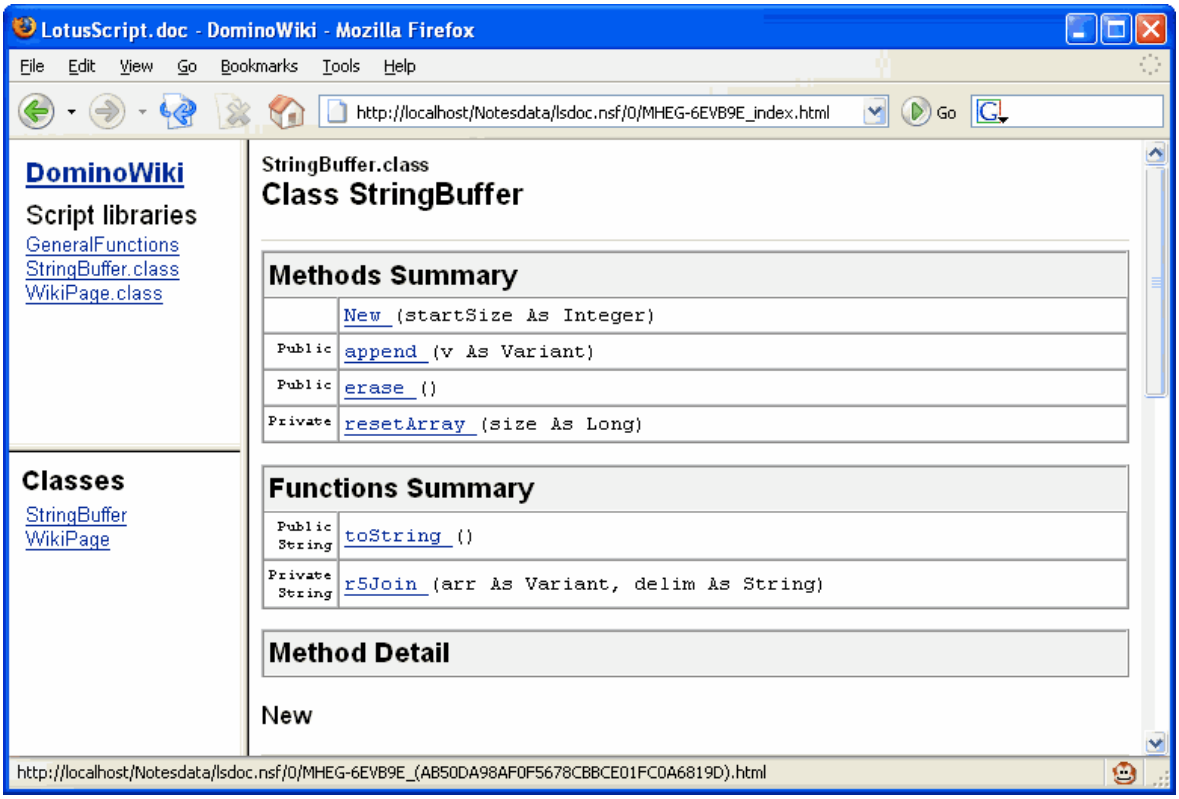
```
%include ”C:\lss-example\aliLuokat.lss”
```

Jatkossa muutokset tehdäänkin tähän .lss-tiedostoon ja tallennetaan koodikirjasto uudelleen, jotta muuttunut koodi tulee käyttöön kirjastossa. Tallennettujen koodikirjastojen käytössä on oltava tarkkana siitä, että jokaisella ohjelmoijalla on samat versiot käytetyistä kirjastoista ja että ne löytyvät samasta kansiorakenteesta.

7.4 Automaattinen dokumentaatio

Tyhjästä aloitetun luokkakirjaston hyödyntämisen osaa perusteellisesti ainakin ohjelmoija itse. Valmista luokkakirjastoa käytettäessä puolestaan on tiedettävä, mihin käyttötarkoituksiin se parhaiten soveltuu. Tämän voi selvittää muun muassa oliodokumentaatiosta.

Luokkahierarkioista voi monesti tuottaa automaattisen dokumentaation, josta selviävät metodit sekä kommentoidut kuvaukset luokkien toiminnasta. Java-kielessä tällainen dokumentaatio on JavaDoc. LotusScript-kielellä ohjelmoiduista luokista voi tuottaa HTML-kuvauksen LotusScript.doc, jonka rakentava työkalu hyödyntää luokkien määrämuotoon kirjoitettuja kommenttirivejä. Rivit sisältävät alkuperäisen ohjelmoijan sekä vähintään viimeksi muutoksia tehneen nimet sekä vastaavat päivämäärät. Kommenttiriveillä on myös tiivis kuvaus luokan toiminnasta ja yleisestä käyttötarkoituksesta. Metodien kuvaukset on hyvä ilmoittaa näissä kommenteissa. Jopa ilman kommentteja LotusScript.doc tuottaa esityksen, josta selviää luokkien väliset suhteet ja metodien käyttö. (Flindt Heisterberg 2009.) Kuvan 10 HTML-sivulla, joka on luotu ilman ensimmäistäkään kommenttia, näytetään lista koodikirjastoista ja luokista sekä vielä yksityiskohtainen lista luokkien metodeista. Metodit näyttävässä taulukossa esitetään metodin näkyvyysasetus ja metodin käyttämät parametrit.



KUVIO 10. Sivu LotusScript.doc-kuvauksesta (Flindt Heisterberg 2009)

8 DOMINO-SOVELLUS OLIOSUUNTAUTUNEEKSI

Graafisessa sovelluskehittimessä eri kohteita ohjelmoidaan käyttäen eri ohjelmointikieliä. Formula-kieltä käytetään näkymissä, kenttäarvojen yhteydessä sekä yksinkertaisissa painiketoiminnoissa. LotusScript on käytössä käyttöliittymätapahtumissa, agenteissa ja painikkeissa, Java pääasiassa agenteissa. Oliosuuntautunut ohjelmointitapa on mahdollinen LotusScript- ja Java-kielissä – Javahan on puhdas oliokieli, joten muuta vaihtoehtoa ei olekaan. Oliosuuntautuneeksi muuttaminen rajoittuu siis lähinnä elementteihin, joita voi ohjelmoida LotusScript-kieltä käyttäen.

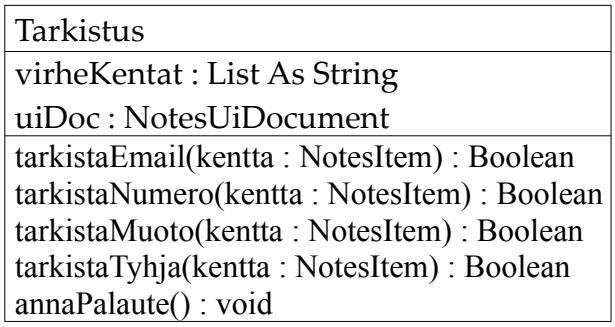
Koska arkkitehtuurissa elementin koodi on kiinteästi mukana itse elementissä, sovellus- tai komponenttikehyksen sijaan kannattaa käyttää luokkakirjastoa koodin uudelleenkäytettävyyttä tuomaan. Käyttöliittymäelementtien kohdalla niin kuin agenteissakin kirjastot otetaan käyttöön kuten toisissa koodikirjastoissa eli Use-lausetta käyttämällä lomakkeen tai agentin Options-osassa.

8.1 Käyttöliittymätapahtumat

Sovelluksessa saattaa lomakkeen tallennuksessa olla samanlainen kenttätarkistus kuin luvun 7.2.3 lainauksessa sivulla 21. Jos toisellakin lomakkeella on tarve kenttätarkistukselle, on sielläkin tallennusvaiheessa tämä sama tai vastaava koodi. Kun mukaan vielä tulee useampia tarkistettavia kenttiä, täytyy pohtia tapoja tehdä tarkistus joustavammin eri tilanteissa.

Oliosuunnittelussa kenttien tarkistukseen soveltuu luokka, jonka metodit toteuttavat erilaisia kentän arvoon kohdistuvia tarkistuksia. Tyypillisiä tarkistuksia ovat

tyhjän arvon (joka voi joskus tarkoittaa myös pelkkiä välilyöntejä sisältävää merkkijonoa) lisäksi tarkistus siitä, onko arvona päivämäärä tai numero, tai sen tarkistaminen, vastaako arvo määritettyä merkkijonomallia. Esimerkki viimeksi mainitusta on äärimmilleen yksinkertaistettu sähköpostiosoitteen tarkistus, jossa varmistetaan arvosta löytyvän ”@”-merkin sekä ainakin yhden pisteen. Luokan on syytä myös tarjota koottu lista puutteellisista kentistä, kun kaikki kentät on tarkastettu. Kuviossa 11 on UML-standardin mukainen esitys kenttätarkistuksen toteuttavasta luokasta.



KUVIO 11. Kenttätarkistus

Tarkistuksen toteuttavassa luokassa voidaan hyödyntää alkuperäistä koodia. Viittaukset Title-kentän nimeen korvataan muuttujalla ja virheilmoituksen näyttö siirretään tarkistuksen tuloksen esittävään metodiin. Virheen ilmoittamista muutetaan niin, että ilmoituksessa näytetäänkin kaikki kentät, jotka eivät läpäisseet jotain tarkastuksista.

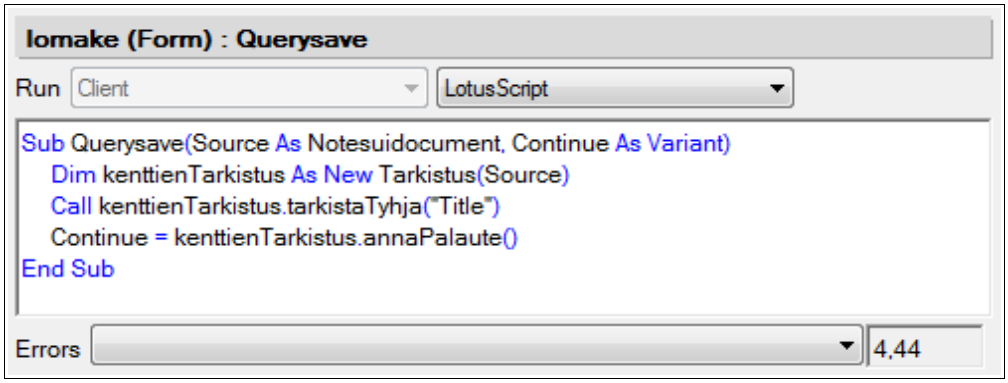
Muuttujien nimeämisessä on hyvä tapa aloittaa luokan attribuutit merkinnällä ”m_”, jolloin ne erottuvat paikallisista muuttujista koodin keskelläkin. Moniosainen muuttujan nimi kirjoitetaan ilman välilyöntejä niin sanotulla CamelCase-merkintätavalla, jossa ensimmäistä kirjainta lukuun ottamatta sanat alkavat isolla alkukirjaimella.

Kuviossa 12 esitetään LotusScript-kielellä kirjoitettu luokka, joka toteuttaa tyhjän arvon tarkistamisen sekä virheellisistä kentistä kootun listan näyttämisen. Käyttäjän hyväksyttyä palauteilmoituksen kohdistin vielä siirretään ensimmäiseen virheellisesti täytettyyn kenttään kuten aiemmin esitetyssä yhden kentän tarkistuksessakin. Metodi annaPalaute palauttaa totuusarvon False, jos missä tahansa kentässä on ilmennyt tarkistuksen aikana virhe. Tätä arvoa käytetään parametrin Continue arvona, jolloin lomakkeen tallennus ei onnistu ennen virheellisten kenttäarvojen korjaamista.

```
Class Tarkistus
    Private m_virheKentat List As String
    Private m_uidoc As NotesUIDocument

    Public Function tarkistaTyhja(kentta As Variant) As Boolean
        If ( m_uidoc.FieldGetText(kentta) = "" ) Then
            m_virheKentat(kentta) = kentta
            tarkistaTyhja = False
        End If
    End Function

    Public Function annaPalaute As Boolean
        annaPalaute = True
        Dim ensimmainenVirhe As String
        Dim msg As String

        ensimmainenVirhe = ""
        msg = ""

        ForAll vk In m_virheKentat
            msg = msg + vk + " "
            If (ensimmainenVirhe = "") Then ensimmainenVirhe = vk
        End ForAll

        If (msg <> "") Then
            MessageBox("Virhe kentässä " + msg)
            Call m_uidoc.GotoField(ensimmainenVirhe)
            annaPalaute = False
        End If
    End Function

    Sub New(src As NotesUIDocument)
        Erase m_virheKentat
        Set m_uidoc = src
    End Sub

End Class
```

KUVIO 12. Tyhjän arvon tarkistaminen ja virheen näyttö

Kuviossa 13 hyödynnetään uutta luokkaa lomakkeen tallennustapahtuman kenttätarkistuksessa. Luokka on tallennettu koodikirjastoon ja otettu käyttöön Options-osassa. Uudelleen käytettävän koodin ansiosta itse käyttöliittymätapahtumaan kirjoitettavan uuden koodin määrä on nyt pienempi. Jos tyhjän arvon tarkistavaa luokkaa vielä laajennettaisiin käsittämään myös välilyönneistä koostuva arvo, voitaisiin muutos toteuttaa koodikirjastoon sen sijaan, että jouduttaisiin muuttamaan jokaista kenttätarkistuksen sisältävää lomaketta erikseen.



KUVIO 13. Tarkistus-luokan käyttö

8.2 Agentit

Agentti voi olla hyvin laaja, useita aliohjelmia hyödyntävä kokonaisuus, mutta myös pienet yksinkertaisen toiminnon suorittavat agentit ovat tavallisia. Viimeksi mainittujen oliopohjaiseksi muuttaminen ei ehkä ole kannattavaa, kunhan agentti vain toimii luotettavasti. Poikkeuksena tähän on kohtalaisen tavanomainen tilanne, jossa eri agentit tekevät suunnilleen samankaltaista toimintaa. Tällöin kysymykseen tulee luokkakirjasto, jonka rajapinnan kautta käytetään kussakin agentissa tarvittavia toimintoja. Formula-kielellä toteutettuja toimintoja voi jäljitellä LotusScript-kielisellä ohjelmalla, mutta Formula-agentit ovat usein kooltaan siinä määrin pienehköjä, ettei niidenkään muuttaminen välttämättä kannata. Formula-agenteissa on myös se etu, että ainoastaan niissä voi automaattisesti käsitellä samalla koodilla valintakriteerien mukaisten dokumenttien koko joukkoa.

Valittaessa oliosuuntautuneeksi muutettavia agentteja on pääpaino agentin käyttöasteella sekä laajuudella eli tiedostokoolla. Aktiivisessa käytössä olevat monimutkaisemmasta päästä olevat agentit, joilla on runsaasti muutoksia joko ennakoitavissa tai viime aikoina suoritettuna, ovat sopiva valinta olioitavaksi. Kuvion 7 päätösmatriisia voidaan soveltaa myös agenttien muutostarvetta arvioitaessa. Globaaleihin muuttujiin perustuvaa oliointimenetelmää voidaan soveltaa proseduraalisesti ohjelmoituun agenttiin. Globaalit muuttujat on tyypillisesti määritetty

agentin Declarations-osiossa, ja varsinainen ohjelmakoodi sijaitsee Initialize-osiossa sekä aliohjelmissa.

9 YHTEENVETO

Alussa asetin tavoitteekseni selvittää suositeltuja hyviä tapoja proseduraalisen ohjelmiston muuttamiseen oliosuuntautuneeksi. Päätin myös ottaa selvää, milloin koko ohjelmiston muuttamiseen ei kannata ryhtyä, vaan kevyemmälläkin tavalla saadaan hyötyjä ylläpidettävyyden muodossa. Esittelin ohjelmointiparadigmojen pääpiirteet ja selvitin oliosuuntautuneeksi muuttamisen vaiheet takaisinmallinnuksesta ja uudelleendokumentoinnista lähtien aina siihen, kun muodostetut luokat on koottu luokkakirjastoon.

Globaaleja muuttujia hyödyntämällä saadaan muodostettua luokkarakenne. Tätä tapaa käytettäessä voi tosin vastaan tulla tilanne, jossa alkuperäisen ohjelmiston rakenteet ovat niin tiiviitä, ettei globaalien muuttujien olioinnilla saadakaan kuin yksi suuri luokka. Silloin luokat löydetään tutkimalla pienempiä osasia. Muutostarpeen arvioinnissa apua saatiin päätösmatriisista, jossa arvioidaan ohjelmiston liiketaloudellista arvoa sekä teknistä laatua.

Domino-sovelluksessa useissa kohteissa voidaan käyttää menetelmiä, joilla proseduraalinen ohjelmisto muutetaan oliosuuntautuneeksi. Luokkakirjastoja kannattaa käyttää uudelleenkäytettävän koodin varastona. Soveltuvat osat alkuperäisestä koodista voidaan hyödyntää oliosuuntautuneessa ratkaisussa. Agenttien muuttamisessa kannattaa jättää ennalleen kaikkein pienimmät agentit ja pääosa Formula-kielellä kirjoitetuista agenteista. Jäljelle jäävistä arvioidaan, kuinka aktiivisessa käytössä ne ovat. Tässäkin voidaan soveltaa päätösmatriisia.

LÄHTEET

Flindt Heisterberg, M. 2009. LotusScript.doc. Www-dokumentti. Saatavissa: http://blog.lsdoc.org/. Luettu: 17.4.2012.

Haikala, I. & Märijärvi, J. 2002. Ohjelmistotuotanto. 8., uudistettu painos. Helsinki: Talentum Media Oy.

Harsu, M. 2003. Ohjelmien ylläpito ja uudistaminen. Helsinki: Talentum Media Oy ja Maarit Harsu.

IBM Corporation 2007. Lotus Domino Designer 8 Help. Www-dokumentti. Saatavissa: http://publib-b.boulder.ibm.com/lotus/c2359850.nsf/Main?OpenFrameSet. Luettu: 28.3.2012.

Kendal, S. 2009. Object Oriented Programming using Java. Www-dokumentti. Saatavissa: http://bookboon.com/fi/student/it/object-oriented-programming-using-java. Luettu: 26.3.2012.

Koskimies, K. 2000. Oliokirja. 2. painos. Helsinki: Satku – Kauppakaari.
Perry, P. 2001. Using the object-oriented features of LotusScript. Www-dokumentti. Saatavissa: http://www.ibm.com/developerworks/lotus/library/ls-object_ oriented_ LotusScript/. Luettu: 22.3.2012.

Tulisalo, T., Carlsen, R., Guirard, A., Hartikainen, P., McCarthy, G. & Pecly, G. 2002. Domino Designer 6: A Developer's Handbook. Www-dokumentti. Saatavissa: http://www.redbooks.ibm.com/abstracts/sg246854.html?Open. Luettu: 18.11.2011.