Antti Kiiskinen

Sovellusaluekeskeinen ohjelmistokehitys ja visuaalinen sovellusaluekieli rahaston arvonlaskentaprosessin mallintamiseksi

Tietojärjestelmätieteen
pro gradu -tutkielma
9.6.2008

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Kiiskinen, Jyri Antti
Sovellusaluekeskeinen ohjelmistokehitys ja visuaalinen sovellusaluekieli laskentakaavan esittämiseksi / Antti Kiiskinen
Jyväskylä: Jyväskylän yliopisto, 2008.
91 s.
Pro gradu -tutkielma

Suurimpina ohjelmistokehitystä vaivaavina ongelmina mainitaan usein mm. kehityksen kalleus ja sovellusten heikko laatu. Eräs ratkaisuehdotus näihin ongelmiin on sovellusaluekeskeinen ohjelmistokehitys, jonka tärkein piirre on abstraktiotason nosto perinteisiin menetelmiin verrattuna. Tähän tavoitteeseen pyritään käyttämällä sovellusaluemallinnusta ja sovellusaluekieltä, jolloin mallinnus voidaan tehdä sovellusalueen omilla termeillä.

Tutkielman tarkoituksena on selvittää, mitä tarkoitetaan sovellusaluemallinnuksella ja sovellusaluekielellä, sekä miten näitä voidaan soveltaa käytännön tilanteessa. Mallinnusta yleisesti ja sovellusaluemallinnusta erityisesti koskevan käsitteellisen osuuden jälkeen kuvataan tutkielmaa varten kehitetty visuaalinen sovellusaluekieli, jolla voidaan mallintaa yksinkertainen sijoitusrahaston arvonlaskentaprosessi ja luoda tämän mallin pohjalta sovellus. Sovellusaluekielen toteuttamisessa käytetään Microsoftin DSL Tools -työkalua. Kielen tavoitteena on toimia välineenä, jolla testataan sovellusaluekeskeisen lähestymistavan toimivuutta valitulla sovellusalueella, ja pohjana jatkokehitystä varten. Lopullisena tavoitteena on luoda mallinnuskieli, jota voidaan käyttää sijoitusrahaston arvonlaskentakaavan esittämiseen ja jonka avulla loppukäyttäjä voi muokata kaavaa mahdollisimman itsenäisesti.

AVAINSANAT: sovellusaluekeskeinen ohjelmistokehitys, sovellusaluemallinnus, sovellusaluekieli, DSL Tools

SISÄLLYSLUETTELO

KUVIOT

TAULUKOT

LIITTEET

KIITOKSET

Suuri kiitos kuuluu ohjaajalleni, KTT Mauri Leppäselle. Hänen neuvonsa olivat varsinkin kirjoitustyön alussa korvaamattomia. Ei myöskään voida vähätellä hänen myöhempää panostaan tekstin rakenteen ja sisällön selkeyttämiseksi.

Kiitos myös monivuotiselle työnantajalleni, Digia Financial Softwarelle mahdollisuudesta tehdä tutkielma yrityksen aiheesta, samoin kuin tuesta ja joustavuudesta kirjoitustyön aikana.

FM Risto Moilanen toimi Digia Fincancial Softwaren puolelta ohjaajanani. Häntä haluan kiittää paitsi keskustelujemme kautta saamistani ideoista ja neuvoista, myös ylipäänsä aihealueen tuomisesta organisaatioomme.

1 JOHDANTO

Ohjelmistokehitys on nykyään pitkälti käsityötä. Tästä seuraa ongelmia, jotka ovat tuttuja muilta teollisuudenaloilta niiden historian alkuvaiheesta. Silmiinpistävin niistä on heikko työn tuottavuus: käsityö on hidasta ja kallista. Lisäksi monessa tapauksessa vain parhaiden ammattilaisten työn jälki on yhtä hyvää kuin automatisoitujen koneiden tuottama standarditavara. Yksi virheetön käsityönä tehty tuote ei takaa, etteikö seuraavasta kappaleesta tulisi viallinen. Perinteisemmillä teollisuudenaloilla näihin ongelmiin on löydetty tehokkaat lääkkeet, ja massatuotannolla pystytään tuottamaan laadukasta tavaraa kustannustehokkaasti. Ohjelmistokehitys on edennyt huimaavan nopeasti muutamien vuosikymmenten olemassaolonsa aikana, mutta tasalaatuisia sovelluksia ei vielä tule liukuhihnalta kuluttajan kukkarolle sopivaan hintaan.

Tehokkuus- ja laatuongelmien ratkaisemiseksi etsitään jatkuvasti keinoja, ja niitä on myös löydetty. Työkalut ovat kehittyneet alkuaikojen reikäkorttikoneista konekielen ja assemblerin kautta korkeamman tason ohjelmointikieliin, ja erilaisia tapoja ohjelmistokehityksen menetelmien virtaviivaistamiseksi on kehitetty. Suurimmat edistysaskeleet tehokkuuden parantamisessa on saatu, kun on pystytty nostamaan abstraktiotasoa, jolla kehitystyö tehdään (Kelly & Tolvanen 2008, 3). Esimerkiksi siirtyminen konekielisestä ohjelmoinnista C-kieleen moninkertaisti ohjelmoijien tehokkuuden. Seuraava hyppy abstraktiotason nousussa on ainakin laajassa mittakaavassa tekemättä, mutta muutamia ehdokkaita tämän loikan tekijäksi on esitetty.

Komponenttipohjainen ohjelmistokehitys on eräs keino tehostaa kehitystyötä piilottamalla ohjelmistokehittäjälle tarpeeton tieto omaan komponenttiinsa, jonka palveluja kehitettävä sovellus käyttää. Tärkeä osa komponenttpohjaista lähestymistapaa on myös pyrkimys komponenttien uudelleenkäyttöön (Crnkovic 2003, 9). Komponenttien käyttö ei kuitenkaan ole osoittautunut kattavaksi ratkaisuksi, vaikka se joissain tiukasti rajatuissa tapauksissa varteenotettava ja

käyttökelpoinen tapa onkin. Komponenttipohjaisen ohjelmistokehityksen ongelmia ovat mm. puutteet komponenttien paketoinnissa ja usein esille nouseva, komponentin sisäisen toiminnan näkymättömyydestä kumpuava epäluulo komponentin toimintaa kohtaan (ks. Greenfield & Short 2004, 109-115). Malliperustainen ohjelmistokehitys taas on Object Management Groupin (OMG) ehdotus, jonka tarkoituksena on automatisoida ohjelmakoodin tuottaminen mallin pohjalta (OMG 2008). Ongelmana tässä lähestymistavassa on mm. Steven Kellyn (2005) mukaan kuitenkin mallinnukseen käytettävän UML-kielen yleisluontoisuus, jolloin abstraktiotasoa ei voida nostaa riittävän korkealle, vaan joudutaan edelleen toimimaan sovelluskehityksen termistöllä.

Tässä tutkielmassa käsiteltävä keino abstraktiotason nostoon on sovellusaluekeskeinen ohjelmistokehitys. Sovellusaluekeskeinen ohjelmistokehitys pyrkii häivyttämään mallinnuksen toteuttajalta ohjelmistotekniset yksityiskohdat ja keskittymään kohteena olevan sovellusalueen hahmottamiseen (Kelly & Tolvanen 2008, 15-17). Tähän tavoitteeseen pyrittäessä tärkeänä tekijänä on se, että mallinnus tehdään sovellusalueen omilla käsitteillä. Erittäin oleellinen seikka on myös sovelluksen automaattinen generointi. Tältä osin sovellusaluemallinnus onkin läheistä sukua malliperustaiselle ohjelmistokehitykselle, kuten voidaan huomata vertaamalla vaikkapa artikkeleita Völter ja Bettin (2004) sekä Tolvanen (2005) koodin generointia käsitteleviltä osiltaan. Sovellusaluemallinnus kuitenkin poikkeaa malliperustaisesta lähestymistavasta mm. sovellusalueelle räätälöidyn mallinnuskielen käytöllä (Kelly & Tolvanen 2008, 6).

Sovellusaluemallinnukseen käytettävää työkalua kehittävän MetaCase Consulting -yrityksen kautta aihepiiristä kokemuksia keränneet Kelly ja Tolvanen toteavat tuoreessa teoksessaan lähestymistavalla saavutetun monissa tilanteissa 3-10-kertaisen tehokkuuden lisäyksen (Kelly & Tolvanen 2008, 22). Myös Kieburtz, McKinney, Bell, Hook, Kotov, Lewis, Oliva, Sheard, Smith ja Walton (1996) havaitsivat sovellusaluemallinnuksessa käytettävän sovellusaluekielen tehokkuuden vertaillessaan sovellusaluekielen käyttöä mallinteiden (*template*)

avulla tehtyyn suunnitteluun. Mikäli sovellusaluekeskeinen ohjelmistokehitys on otettavissa tulevaisuudessa käyttöön yhä useammilla sovellusalueilla ja näistä pääosassa päästäisiin vastaaviin tuloksiin, voidaan tätä lähestymistapaa pitää uutena hyppäyksenä ohjelmistokehityksen tuottavuudessa. Laajaa tietoa sovellusaluekeskeisen lähestymistavan käytettävyydestä ei kuitenkaan ole olemassa, koska tutkimusala on erittäin nuori. Tästä syystä aihetta on tutkittava lisää ja eräs osa tätä jatkotutkimusta on lähestymistavan testaaminen erilaisissa käytännön ohjelmistokehityksen tilanteissa.

Tämän tutkielman tutkimusongelmana on selvittää, miten rahaston arvonlaskentaan tarkoitettu sovellus voidaan kuvata loppukäyttäjän ymmärtämällä sovellusaluekielellä. Sovellusalueena on rahoitusala, tarkemmin rahaston arvonlaskentaan käytettävä sovellus. Sovellusaluekielen toteuttamisen työkaluna käytetään Microsoftin DSL Toolsia (ks. Cook, Jones, Kent & Wills 2007).

Tutkimusongelma jakautuu seuraaviin neljään tutkimuskysymykseen:

- Mitä sovellusaluemallinnus ja sovellusaluekielet ovat?
- Miten sovellusaluekieltä kehitetään?
- Millainen sovellusaluekieli sopii rahaston arvonlaskentaprosessin kuvaamiseen?
- Miten Microsoftin DSL Tools -työkalua voidaan käyttää tämän kielen toteutuksessa?

Tutkielman tavoitteena on tuottaa tiivis yleisesitys sovellusaluekeskeisestä ohjelmistokehityksestä ja DSL Tools -työkalusta sekä toteuttaa prototyyppi sovellusaluekielestä, jolla voidaan esittää rahaston arvonlaskennan laskentakaava puurakenteena. Prototyypissä pyritään mahdollistamaan kaavan lähtöarvojen sekä tietolähteiden mallintaminen.

Tutkielma jakautuu neljään lukuun. Luvussa 2 esitellään sovellusaluemallinnukseen liittyvät termit ja kerrotaan laajemmin lähestymistavan osa-alueista ja suositelluista työskentelytavoista. Osuudessa käsitellään myös sovellusaluekeskeisen lähestymistavan käyttöönoton edellytyksiä ja käydään läpi sovellusaluekielten hyötyjä ja haasteita verrattuna yleiskäyttöisiin mallinnuskieliin. Luvussa 3 esitellään Microsoftin DSL Tools -työkalu. Luvussa kuvataan työkalun eri osat ja sen käytön pääperiaatteet. Luku 4 sisältää tutkielman käytännön osuuden. Siinä esitellään sovellusaluekielen prototyyppi, joka on pohjana myöhemmälle jatkokehitykselle. Tutkielma päättyy yhteenvetoon.

2 SOVELLUSALUEKESKEINEN OHJELMISTOKEHITYS

Yleiskielessä mallinnuksella tarkoitetaan yleensä jonkin esineen tai asian kuvaamista siten, että tuloksena syntynyttä mallia tarkastelemalla voidaan päätellä jotain mallin kuvaamasta kohteesta. Sovellusaluemallinnus on eräs mallinnuksen esitysmuoto. Tässä luvussa määritellään aluksi yleiseen mallintamiseen liittyvät käsitteet siltä osin kuin tämän tutkielman kannalta on tarpeen. Tämän jälkeen laajennetaan käsitteistöä sovellusaluekeskeisen ohjelmistokehityksen omilla, yleisestä termistöstä erilaistetuilla käsitteillä, sekä määritellään lähestymistavan omat käsitteet.

2.1 Malli ja kieli

Mallinnuksella tarkoitetaan sarjaa toimenpiteitä, jotka sisältävät kohdealueen tarkastelun, alueen sisäistämisen pohjalta tehtävän mallin luomisen ja mallin esittämisen (Falkenberg, Hesse, Lindgreen, Nilsson, Oei, Rolland, Stamper, Van Assche, Verrjin-Stuart & Voss 1998, 56). Malli-termin eri määritelmät poikkeavat toisistaan suuresti sen mukaan, minkä tieteenalan edustaja määritelmän on tehnyt. Yhteisenä, käytännössä kaikkien tutkijoiden hyväksymänä osana määritelmää on se, että *malli* on keino saada tietoa ongelmaan liittyvistä relevanteista asioista. (Leppänen 2005, 279)

Malli on siis jonkin asian, esineen, käsitteen tai niiden muodostaman kokonaisuuden kuvaus. Falkenberg ym. (1995, 31) määrittelevät mallin tarkoituksenmukaisesti abstrahoiduksi, selkeäksi, tarkaksi ja yksiselitteiseksi käsitykseksi kohteestaan. *Mallin esitys* taas tarkoittaa mallin tarkkaa ja yksikäsitteistä esitystä soveltuvalla kielellä (Falkenberg ym., 1995, 55). Mallin esitys voidaan käsittää myös mallin osaksi, jolloin siis malli-käsitteeseen kuuluu myös sen esitystapa. Tässä tutkielmassa käytetään malli-termiä yleisesti mallista ja esityksestä yhdessä. Milloin on syytä eritellä nämä käsitteet, käytetään mallin esityksestä omaa termiään.

Kieli on väline, jolla malli voidaan esittää. Kieli koostuu syntaksista ja semantiikasta. *Syntaksi* määrittelee kielen elementit, näiden elementtien väliset suhteet ja säännöt näiden suhteiden luontiin. *Abstrakti syntaksi* jäsentää kielen kuvaamia todellisen maailman käsitteitä ja niiden välisiä suhteita. Abstrakti syntaksi ei ota kantaa käsitteiden ja suhteiden esitystapaan, vaan se toimii käsitemallin tasolla. *Konkreetti syntaksi* taas sisältää abstraktin syntaksin sisältämien käsitteiden ja suhteiden esitystavan, sekä näiden käsittelysäännöt kyseisellä kielellä. (Leppänen 2005, 96; Langois, Jitia, & Jouenne 2007, 30) Esimerkiksi käsite ”pankkiautomaatti” voi olla osa pankkijärjestelmää kuvaavan mallin luontiin käytetyn kielen abstraktia syntaksia, ja pankkiautomaattia tarkoittava symboli osa kielen konkreettia syntaksia. Kielellä voi olla useita konkreetteja syntakseja (Langois ym. 2007, 30). Esimerkiksi käyttöliittymän ikkuna voidaan esittää C#:ssa paitsi ohjelmointiympäristön ikkunaeditorissa, myös ohjelmakoodin muodossa, ja tiedostoon kopiointiin kulunut aika voidaan esittää numeromuodossa sekunteina tai graafisesti pylväänä.

Semantiikka tarkoittaa syntaksin sisältämien symbolien suhdetta niiden kuvaamiin kohteisiin (Leppänen 2005, 96). Semantiikka siis sisältää määritelmät, jotka antavat tarkoituksen syntaksin mukaisesti esitetyille kielen lauseille tai niiden osille (Sánchez-Ruiz, Saeki, Langlois & Paiano 2007, 4). Suppeammasta, tietotekniikan näkökulmasta ajateltuna Tucker (1985) määrittelee semantiikan tarkoittavan ohjelman ajonaikaista käyttäytymistä, eli sitä, mitä tapahtuu, kun ohjelma suoritetaan. Tästä syystä hän mainitseekin semanttisten tavoitteiden saavuttamisen olevan syntaksin pääasiallinen tavoite, ja siten syntaksin suunnittelun alkuperäinen motivaatio. (Tucker 1985, 250)

Kielet voidaan jaotella formaaleihin, puoliformaaleihin ja ei-formaaleihin eli luonnollisiin kieliin. *Formaalit* kielet ovat kieliä, joilla on tarkasti määritetty syntaksi ja semantiikka. *Puoliformaaleissa* kielissä on määritelty vain syntaksi. (Leppänen 2005, 97) *Luonnolliset* kielet ovat esimerkiksi puhuttuja kieliä, kuten suomi tai englanti.

Mallin kuvaamiseksi käytettävä kielikin pitää suunnitella, ja suunnitteluun tarvitaan kielestä tehtävää mallia. Tätä mallia kutsutaan *metamalliksi*, eli mallin malliksi. Metamalli kuvaa kielen käsitteet ja ne säännöt, jotka määrittävät, millaisia malleja kielellä voidaan luoda. (Falkenberg ym. 1998, 58) Metamallin esittämiseen käytetään *metakieltä*, jonka suhde metamalliin vastaa kielen suhdetta malliin. Metakieli on siis kieli, jolla kuvataan toista kieltä.

Metatasojen hierarkiassa seuraavalla tasolla ovat *meta-metamalli* ja vastaavasti *meta-metakieli*. Näitä käytetään kuvaamaan metamallia ja metakieltä (Falkenberg ym. 1998, 58) Hierarkiaa voidaan jatkaa loputtomiin, mutta kirjallisuudessa ei yleensä edetä tämän pidemmälle. Syynä lienee se, että meta-metamallin tasolta ylemmäs jatkettaessa kyse on edelleen ”mallin kuvaamiseen tarkoitetun mallin kuvaamisesta”, eli meta-metamallista, joten useampien meta-etuliitteiden käyttö ei ole mielekästä. Metatasojen hierarkian huipulla on yleinen kuvauskieli, joka pystyy kuvaamaan myös itsensä (Falkenberg ym. 1998, 85). Tällaisesta kielestä käy esimerkkinä Backus-Naur –muoto (Backus Naur Form, BNF). Sánchez-Ruiz ym. (2007, 5) mainitsevat meta-metamallista esimerkkinä mm. predikaattilogiikan.

2.2 Sovellusaluemallinnus ja sovellusaluekieli

Sovellusaluemallinnuksella tarkoitetaan prosessia, joka tuottaa mallin, joka kuvaa tiettyä sovellusaluetta sen omalla termistöllä. Kyse on siis määrätyn tyyppisestä mallintamisesta, jonka jäsentämisessä on luontevaa käyttää mallintamisessa käytettyjä käsitteitä ja termejä (Sánchez-Ruiz ym. 2007, 6). Sovellusaluemallinnuksen käsitteistöön kuuluu lisäksi muutamia sille ominaisia termejä, joita ei perinteisessä mallintamisessa käytetä.

Ohjelmistokehityksen tuottamista käsitteellisistä kokonaisuuksista tutuin lienee *sovellus.* Sovelluksella tarkoitetaan arkikielessä usein yksittäistä ohjelmaa, riippumatta sen laajuudesta tai suhteista muihin ohjelmiston komponentteihin. Tällainen määritelmä ei kaikessa epämääräisyydessään ole tutkielman kannalta

käyttökelpoinen. Firesmithin ja Henderson-Sellersin (1999, 31) määritelmä sen sijaan on parempi, ja se kuuluu vapaasti suomennettuna seuraavasti: ”sovellus on kaikilta osiltaan toimiva tuote, joka luovutetaan käyttäjäorganisaatiolle”. Sovellus ei siis tämän määritelmän mukaan ole synonyymi tietokoneohjelmalle, vaan se voi koostua useamman ohjelman tai minkä tahansa muun ohjelmistotuotteeseen kuuluvan osan joukosta. Tärkeä osa määritelmää on tuotteen toimivuus, mikä ei nykyisellä ohjelmistotuotannon kypsyystasolla ole aina itsestään selvää.

Sovellusalue tarkoittaa kohdealuetta, jonka ongelmia halutaan ratkaista (Sánchez-Ruiz ym. 2007, 5). Sovellusalueen rajaama osa maailmaa on siis se alue, josta käsitemallin käsitteet löytyvät. Sovellusalue voi olla laajuudeltaan ja monimutkaisuudeltaan hyvinkin vaihteleva: esimerkkejä voidaan etsiä yksittäisen pikkufirman taloushallinnosta, lentoliikenteen ohjaamisesta ja oikeastaan mistä tahansa kuviteltavissa ja rajattavissa olevasta kokonaisuudesta. Sovellusalueet voivat olla myös hierarkkisesti sisäkkäisiä, esimerkiksi reskontra-sovellusalue voi olla osa taloushallinnon sovellusaluetta. (Sánchez-Ruiz ym. 2007, 3)

Sovellusaluemalli on malli, joka on syntynyt sovellusaluemallinnuksen tuloksena. Sovellusaluemalli ei välttämättä poikkea käsitteistöltään tai esitykseltään tavallisesta mallista lainkaan, koska myös perinteisen mallinnuksen tulos voi olla tietyn sovellusalueen kuvaus. Ero on mallin luontitavassa ja sen käyttötarkoituksessa sovellusaluekielen tuottaman sovelluksen lähtökohtana. Sovellusaluemalli-termiä ei käytetä kirjallisuudessa läheskään aina, vaan usein puhutaan vain mallista (esim. Sánchez-Ruiz ym. 2007, 5). Cook ym. (2007, 87) taas käyttävät tätä termiä. Tässä työssä käytetään pääasiassa sovellusaluemalli-termiä, mutta yhteyksissä, joissa sekaannuksen vaaraa ei ole, voidaan käyttää myös malli-termiä sen lyhyyden vuoksi.

Sovellusaluekieli on työkalu, jolla sovellusaluemallinnus tehdään. Ohjelmistokehityksessä kielellä tarkoitetaan yleensä tekstimuotoista ohjelmointikieltä, kuten esimerkiksi Javaa tai C++:aa. Aiemmin kohdassa 2.1. esitetty kielen määritelmä on kuitenkin laajempi sisältäen kaikki mallin esittämiseen käytetyt tavat. Sovellusaluekielen osalta taas yleisesti käytössä oleva määritelmäkin on hyvin laaja: siihen kuuluu esimerkiksi erilaisia graafisia mallinnuskieliä (esim. UML), kaavioita (esim. tilakaaviot) ja kokonaisia kysely- tai muunlaisia kieliä (esim. XML, SQL). (Cook ym., 2007)

Sovellusaluekielelle esiintyy kirjallisuudessa useita määritelmiä. Van Deursen, Klint ja Visser (2000, 26) esittävät seuraavan määritelmän:

> Sovellusaluekieli on ohjelmointi- tai määrityskieli, joka tarjoaa soveltuvien notaatioiden ja abstraktioiden avulla ilmaisuvoimaa, joka on kohdistettu ja yleensä myös rajattu tietylle ongelma-alueelle.

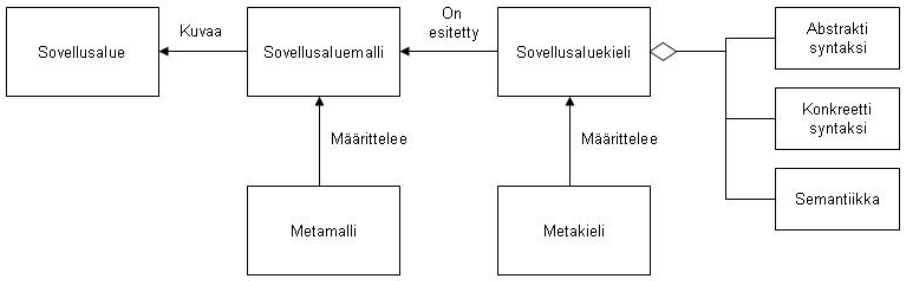
Cook ym. (2007, 11) taas määrittelevät sovellusaluekielen hieman toisesta näkökulmasta:

> Sovellusaluekieli on räätälöity kieli, joka kohdistuu pieneen ongelma-alueeseen, jonka se kuvaa ja jolla tehdyn kuvauksen oikeellisuuden se tarkistaa alueen omia käsitteitä käyttäen.

Sovellusaluekielet eivät ole uusi keksintö. Esimerkiksi jo pitkään käytetyt COBOL ja FORTRAN ovat tällaisia: molemmat kuvaavat oman alueensa kattavasti, mutta eivät sovellu yleiskäyttöisiksi kieliksi (Bentley, 1986). Sovellusaluekielten juuret ovat löydettävissä tietotekniikan tutkimusalan nuoruuteen nähden suorastaan antiikkisesta artikkelista (vrt. Landin 1966). Samoin Jean E. Sammet aavistelee tietyille sovellusalueille rajattujen kielten esiinmarssia (Sammet 1972, 609). Myös sovellusaluemallinnuksen ja sovellusalueanalyysin ajatus on ollut olemassa jo jonkin aikaa. Berard (1993) esittää ajatuksen teoksessaan, tosin oliokeskeisen lähestymistavan näkökulmasta. Samassa teoksessa esitetään myös lyhyt katsaus sovellusalueanalyysin historiaan, ja tässä yhteydessä mainitaan

ensimmäisten aiheeseen liittyvien, merkittävämpien julkaisujen sijoittuvan 1980-luvun alkupuolelle (Berard 1993, 185).

Sovellusaluemallinnuksen pohjalta sovellusaluekieltä käyttäen toteutettua ohjelmistokehitysprosessia kutsutaan *sovellusaluekeskeiseksi ohjelmistokehitykseksi* (*Domain-Specific Software Development, DSSD*). Sovellusaluekeskeiseen ohjelmistokehitykseen kuuluvat mallinnusvaiheessa tuotetut mallit ja metamallit, sovellusaluekieli sekä sovellusaluekielellä tehdyn valmiin mallin käsittelyyn käytetyt komponentit. (Sánchez-Ruiz ym. 2007, 6-7) Lähestymistapaan kuuluvat käsitteet ja niiden suhteet on esitetty yleisellä tasolla kuviossa 1. Eräänä erona perinteiseen ohjelmistokehitykseen on Sánchez-Ruizin ym. (2007, 6) mukaan se, että kehitystyön tekee loppukäyttäjä eikä ohjelmistosuunnittelija. Tämä rajoite ei kuitenkaan vaikuta järkevältä, koska sovellusaluekeskeistä ohjelmistokehitystä voidaan tehdä myös ohjelmistotoimittajan sisäisenä prosessina, kuten esimerkiksi tässä tutkielmassa on asian laita.



KUVIO 1: Sovellusaluekeskeisen lähestymistavan käsitekartta

Edellä esitelty termistö vaikuttaa joiltain osin vielä vakiintumattomalta. Parhaiten tämän voi huomata siitä, että sovellusaluemallinnus-termiä ei käytetä kirjallisuudessa läheskään yhtä laajalti kuin sovellusaluekieli-termiä. Esimerkiksi Mernik, Heering ja Sloane (2005) käsittelevät kyllä sovellusaluekieliä ja koodigeneraattoreita erillisinä käsitteinä, mutta eivät esitä näille yhteistä yläkäsitettä. Sen sijaan esimerkiksi ACM:n OOPSLA-konferenssien (Object-Oriented Programming, Systems, Languages & Applications) julkaisuissa sovellusaluemallin-

nus-termi on hyvin yleisesti käytetty, ja sitä käytetään myös konferenssissa järjestettävän, sovellusaluemallinnukseen keskittyvän työpajan nimessä. Ero johtunee osittain artikkelien painotuseroista, mutta tutkimusalueen termistökään ei ole vielä vakiintunutta.

2.3 Sovellusaluekeskeinen lähestymistapa ohjelmistokehitykseen

Yleiskäyttöisiä mallinnuskieliä voidaan käyttää useilla eri sovellusalueilla. Tästä on kiistatonta hyötyä: UML:ää ja Javaa voidaan soveltaa yhtä hyvin vaikkapa rahoitusalalla kuin teollisuuden toiminnanohjauksessakin. Kun sovelluksen kehittäjän työtehtävät vaihtuvat toiselle sovellusalueelle, hänen ei tarvitse opetella uutta mallinnus- tai ohjelmointikieltä. Steven Kelly toteaa Dr. Dobb's Journalin haastattelussa (Blake, 2007), että vaikka UML on erittäin hyvä työkalu eri sovellusalueilla toimivien ohjelmistokehittäjien väliseen kommunikaatioon, sitä kuitenkin harvoin tällaiseen käytetään. Yleensä kyse on tietyn sovellusalueen ohjelmoijien keskinäisestä kommunikaatiosta tai kyseisen sovellusalueen ohjelmiston suunnittelijoiden ja heidän asiakkaidensa välisestä keskustelusta. Tällaista kommunikaatiota varten olisi usein jo olemassa oma, sovellusalueelleen yhteinen termistö. (Blake, 2007) Tällöin on siis perusteltua ottaa käyttöön sovellusaluekeskeinen ohjelmistokehitysprosessi.

Sovellusaluekeskeinen ohjelmistokehitys ei kilpaile perinteisempien lähestymistapojen kanssa yleiskäyttöisyydessä. Sovellusaluemallinnuksen perimmäinen idea on päinvastoin se, että se keskittyy kuvaamaan yhden alueen hyvin, mutta jättää muut alueet huomioimatta. Tällöin mallinnuskieli voidaan rajata koskemaan vain kohdealuetta. Tästä taas seuraa, että myös mallinnettavaa käsiteavaruutta voidaan rajata, jolloin sen mallintaminen tarkasti on oleellisesti helpompaa. Sovellusalue on usein rajattavissa hyvinkin kapeaksi, esimerkiksi yhden yrityksen tai jopa tietyn ohjelmistoperheen käyttöön. (Pohjonen & Kelly 2002) Koska sovellusaluemalli on tietylle sovellusalueelle räätälöity, on se tähän alu-

eeseen nähden ilmaisuvoimaisempi verrattuna yleiskäyttöisiin mallinnusmenetelmiin.

Sovellusaluemallinnuksessa pyritään siis siirtämään mallinnus *toteutusalueelta* (solution domain) *ongelma-alueelle* (problem domain). Ongelma-alue tarkoittaa sitä sovellusaluetta, johon kuuluvaa ongelmaa ollaan ratkaisemassa. Haikalan ja Märijärven esittämän jaottelun mukaisesta vaihejakomallista ainoastaan esitutkimus ja määrittelyvaiheestakin vain osa kuuluvat ongelma-alueeseen (Haikala & Märijärvi 1998, 25-29). Mallin osista myös ylläpito voidaan ehkä ajatella ohjelmiston toiminnan semanttisen oikeellisuuden osalta kuuluvaksi ongelma-alueeseen, mutta lopulta kyse on kuitenkin ratkaisun toimivuuden ylläpidosta. Ylläpitokin siis kuuluu toteutusalueeseen, joka on se alue, jolla ongelman ratkaisemiseen käytettävät työkalut sijaitsevat, eli esimerkiksi ohjelmointikieli tai mallinnustekniikka (Langlois ym., 2007).

Mallinnuksen siirtäminen ongelma-alueelle mahdollistaa kyseisen sovellusalueen termistön, prosessien ja ajattelutavan käyttämisen mallintamisessa. Samalla mallinnuksen abstraktiotaso nousee huomattavasti perinteistä ohjelmistokehitystä korkeammalle. Tärkeimpänä hyötynä tästä on se, että syvällistä ohjelmistokehityksen asiantuntemusta ei tarvita, joten sovellusalueen asiantuntijoiden on huomattavasti helpompaa olla mukana sovelluskehityksessä myös määrittelyvaiheen jälkeen (esim. Sánchez-Ruiz ym., 2007, 7, Cook ym., 2007, 10). Sovellusaluemallinnuksessa ei tarvitse osata mallintaa esimerkiksi UML:llä tai ymmärtää periytymistä, vaan voidaan vaikkapa matkapuhelimen käyttöliittymää suunniteltaessa puhua tekstiviestin lähettämisestä, valikosta ja virtanapista (Kelly 2005).

Mallinnuskielen muuttuminen sovellusalueen asiantuntijoiden käyttöön soveltuvaksi on siis suuri edistysaskel. Sovellusaluekeskeinen ohjelmistokehitys tarjoaa tämän ohella toisenkin tärkeän edun: sovellusaluemallista voidaan tuottaa ohjelmakoodia automaattisesti. Mallin kuvaaman käsitteistön ollessa rajattu on

mahdollista kehittää koodigeneraattoreita, jotka osaavat luoda koodia suoraan mallin pohjalta. Koska muunnos mallista toiseen tapahtuu automaattisesti, jää tämä työvaihe ainakin osittain pois. (Tolvanen 2005, 18) Tolvanen (2005, 14) vertaakin sovellusaluemallinnuksen tarjoaman abstraktiotason nousun tuomaa etua tuottavuudessa ja laadussa jopa hyppäykseen assemblerin ja C-kielen välillä.

Jälkimmäinen luvun 2 alussa esitetyistä sovellusaluekielen määritelmistä (Cook ym. 2007, 11) sisältää maininnan oikeellisuustarkistuksista. Nämä automatisoidut tarkistukset vähentävät mallinnusvirheitä huomattavasti. Perinteisten mallinnus- ja ohjelmointimenetelmien oikeellisuustarkistukset rajoittuvat pakostakin lähinnä syntaktisen oikeellisuuden tarkistukseen, koska menetelmien yleiskäyttöisyyden vuoksi ei voida työkalun kehittämisen yhteydessä tietää, mihin kaikkiin tarkoituksiin sitä tullaan käyttämään. Sovellusalueen omaan työkaluun taas voidaan toteuttaa myös semanttisen oikeellisuuden tarkistus, ainakin johonkin rajaan asti. Tällä tavoin virheet huomataan jo aikaisessa vaiheessa, jolloin korjauskustannukset jäävät matalammiksi.

2.4 Generaattorit

Tärkeä osa sovellusaluemallinnusta on pyrkimys minimoida käsityönä tehtävien muunnosten määrä mallista toiseen. Perinteisessä mallinnuksessa tehdään ensin kuvaus sovellusalueen prosesseista ja käsitteistä, jonka jälkeen tämä malli muunnetaan esimerkiksi UML-kielelle (ks. Jacobson, Booch & Rumbaugh 1999, 421-433). UML:n mukaisen mallin pohjalta taas kirjoitetaan ohjelmakoodi, joka on saman asian esitys, mutta jälleen eri kielellä. Kaikki nämä muunnokset tehdään käsityönä, joka on paitsi työlästä, myös aiheuttaa useita mahdollisuuksia virheiden syntymiseen. (Pohjonen & Kelly, 2002)

Sovellusaluemallinnuksessa pyritään siihen, että kun mallinnus on tehty sovellusalueen käsitteistöllä, ei ole enää tarvetta siirtää mallia ongelma-alueelta malliksi, joka kuvaa sovellusta toteutusalueen termistöllä. Sen sijaan sovellusalue-

kielellä luodusta mallista voidaan generoida suoraan haluttu lopputulos, joka on yleensä jollain yleiskäyttöisellä kielellä esitetty, suoritettavaksi käännettävissä oleva ohjelma. Mikään ei myöskään estä käyttämästä useampaa peräkkäistä mallia, mikäli se katsotaan tarpeelliseksi. Joissain tilanteissa voi olla järkevää muuntaa ensimmäinen malli toiseksi malliksi, esimerkiksi useamman sovellusaluekielen tulosten yhdistämiseksi. Tällaisessakin tapauksessa ylemmän tason mallista tuotettava uusi malli kuitenkin generoidaan mahdollisimman automatisoidusti.

Ohjelmointivirheiden määrä vähenee oleellisesti, kun muunnokset mallista sovellukseksi tapahtuvat generaattoreiden avulla automaattisesti. Tämä antaa kehittäjälle mahdollisuuden keskittyä mallin hiomiseen ja vähentää myös tarvetta käännetyn sovelluksen virheiden korjaukseen ja uusintatestauksiin (Cleaveland 1988, 26; Tolvanen 2005, 22-23). Kuten mallinnustyökaluun, myös generaattoreihin voidaan kehittää hyvinkin kattavia oikeellisuustarkistuksia, jotka huomaavat sellaiset käyttäjän tekemät virheet, joita ei pystytä havaitsemaan mallinnusvaiheessa. Lisäksi mekaaniset ja toisteiset, ja siksi ohjelmoijan kannalta suorastaan tappavan tylsät työvaiheet voidaan usein hoitaa generaattoreilla. Koska käsin tehtäviä muunnoksia ei tarvita, ei niihin myöskään kulu aikaa. Generaattorit ovat yleensä taitavimpien ohjelmistokehittäjien rakentamia, ja siten heidän ratkaisunsa teknisesti monimutkaisiin ongelmiin ovat myös kokemattomampien kehittäjien käytettävissä (Tolvanen 2005, 23).

Generaattori toimii siten, että se käy läpi koko sovellusaluekielellä luodun mallin ja tuottaa siitä halutun tuloksen. Mallin eri komponenteille tehdään käsittelysäännöt, joiden pohjalta generaattori tuottaa kunkin säännön mukaisen tuloksen. Generaattori voi esimerkiksi luoda mallinnetusta liiketoimintaluokasta vastaavan luokan kohdekielelle, tehdä näiden mallin luokkien ilmentymistä vastaavia kohdekielen olioita ja muodostaa suhteita olioiden välille. (Kelly 2006)

Ohjelmakoodin automaattinen generointi soveltuu DSM-lähestymistapaan erittäin hyvin. Generaattorin tarvitsee osata käsitellä vain tietyllä sovellusaluekielellä toteutettua mallia, mikä ei tee generaattorin kehittämisestä helppoa, mutta sentään vähemmän vaikeaa. Kelly (2006) pitää myös yleiskäyttöisten mallinnusmenetelmien pohjalta toimivia generaattoreita hankalina niiden nopean monimutkaistumisen takia.

Koodigeneraattorin kehittämisen tavoitteena tulisi aina olla se, että generaattorin tuottama koodi on valmista käännettäväksi. Tätä varten generaattorin tulisi olla yksinkertainen ja mahdollisimman suoraviivaisesti toimiva. Kaikenlainen monimutkaisempi vaihtelevuus on pyrittävä toteuttamaan joko kohdealuemalliin tai sovelluskehykseen. (Pohjonen & Tolvanen 2002)

Generoidun koodin tulisi olla selkeää. Vaikka generoituun koodiin ei saakaan tehdä muutoksia käsin, pitää se kuitenkin olla ohjelmoijan luettavissa. Tähän on syynä se, että koodia joudutaan varsinkin ohjelmoijan yleensä suorittamassa moduulien integrointitestauksessa käymään läpi debug-toiminnolla. Samoin mahdollisten generaattoreihin tai niiden konfiguraatioihin tehtyjen muutosten testaaminen tarkoittaa yleensä myös generoidun koodin lukemista. (Völter ym. 2004, 39) On siis syytä pyrkiä generoimaan mahdollisimman hyvin yleisiä ohjelmointikäytäntöjä kunnioittavaa ohjelmakoodia, milloin se vain suinkin on mahdollista. Kommenttien käyttö on erityisen tarpeellista. Joissain tapauksissa voidaan joutua uhraamaan luettavuutta esimerkiksi tehokkuuden vuoksi, mutta tällaisten ratkaisujen tulisi olla poikkeuksia (Völter ym. 2004, 39-40). Generoidun koodin ihmisen luettavaksi kelpaamaton maine on Völterin ym. (2004, 39) mukaan joskus syynä automaattisen koodin generoinnin käyttämättä jättämiseen, ja siksikin selkeä ja hyvin kommentoitu koodi tulisi olla tavoitteena.

Vaikka täysin valmista koodia ei generaattorilla saataisi tuotettua, se ei saa silloinkaan vaatia käsin muokkaamista. Tämä tarkoittaa sitä, että mahdolliset mallista generoidun koodin ulkopuoliset osat tulee pitää erillään. Jos koodia pitää

muokata joka käännöksen jälkeen, tuotantoprosessin riskit lisääntyvät huomattavasti. Ongelmia voi syntyä esimerkiksi versioinnissa ja käsin tehtyjen muutosten säilymisessä, kun automaattisesti syntyvä koodi generoidaan uudelleen (Völter ym. 2004, 30-31). Generoidun koodin eräs etu onkin se, että koodi voidaan koska tahansa tuottaa uudelleen, kuitenkaan menettämättä aiemmassa ajossa syntyneitä ominaisuuksia. Jos mallia muutetaan, tarvitsee vain ajaa generaattorit uudestaan ja muutokset tulevat mukaan tuotteeseen. Muuttumattomat osat kuitenkin pysyvät edelleen entisellään. Mikäli generoitua koodia on muokattu manuaalisesti, tällainen koodin uudelleengenerointi ei enää ole mahdollista.

2.5 Sovelluskehys

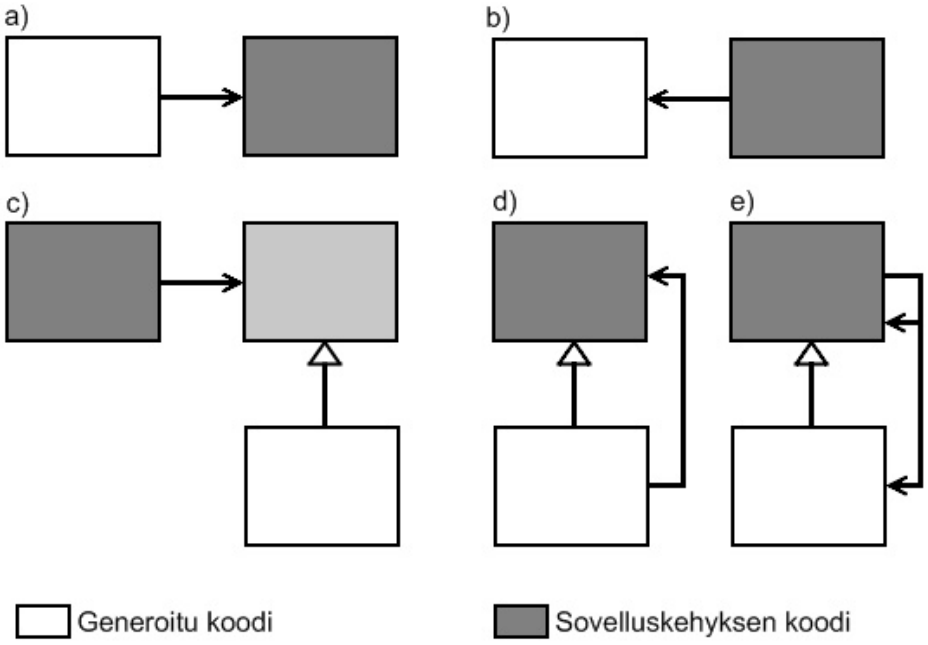
Generaattoreiden kohdealuemallista tuottama koodi ei yleensä ole kääntämistä vaille valmis sovellus. Sovelluksessa on aina osia, joita ei voida tai joita ei kannata generoida, vaan ne on luotava perinteisesti käsin. Esimerkiksi generaattorilla luodun ikkunakomponentin kentät on syytä asemoida käsin, koska niiden automaattinen asettelu näytölle käyttäjäystävällisesti on käytännössä mahdotonta. Tällaisten ihmisen harkintaa vaativien toimenpiteiden lisäksi myös ohjelmakoodin osia voidaan tuottaa muuten kuin generoimalla.

Usein on järkevää luoda sovellusaluekeskeistä kehitystä varten sovelluskehys. Tämä kehys on kokoelma yleisiä toimintoja, joka toimii yhdessä generaattoreiden tuottaman ohjelmakoodin kanssa. Esimerkiksi jakolaskutoimituksen suorittavan luokan nollalla jaon estävää tarkistusta ei välttämättä kannata luoda generaattorin avulla, vaan se voidaan toteuttaa valmiiseen sovelluskehykseen. Tällöin generaattorin tuottama koodimäärä pienenee, ja sitä kautta mallin muuntaminen ohjelmakoodiksi nopeutuu.

Sovelluskehyksen avulla voidaan myös piilottaa käyttöjärjestelmässä tai laiteympäristössä mahdollisesti tapahtuvat muutokset siten, että sovellusaluekieli ja generaattorit pysyvät muuttumattomina. Generoitua koodia voidaan käyttää

esimerkiksi eri käyttöjärjestelmiin käännettynä, kun tarvittavat muutokset tehdään kehykseen siten, että rajapinta generoidun koodin ja kehyksen välillä säilyy muuttumattomana. Samoin laitteistomuutosten aiheuttamat muutostarpeet vähenevät. (Kelly 2005, 4)

Kehyksen ja generaattoreiden tuottaman koodin riippuvuussuhde voidaan määritellä usealla eri tavalla. Tässä esitellyt tavat ovat peräisin Völterin ym. (2004) tutkimuksesta. Kuviossa 2 esitetään kaaviomuotoisena eräitä näistä tavoista.



KUVIO 2. Eri tapoja generoidun ja valmiin ohjelmakoodin integroimiseksi (muokattu artikkelista Völter ym. (2004, 32)):

a) Generoitu koodi kutsuu sovelluskehystä
b) Sovelluskehys kutsuu generoitua koodia
c) Sovelluskehys kutsuu rajapintametodeja, jotka generoitu koodi toteuttaa
d) Generoitu koodi kutsuu perimänsä abstraktin luokan metodeja
e) Sovelluskehyksessä oleva yläluokka kutsuu omia abstrakteja metodejaan, jotka generoitu koodi toteuttaa

Useimmille ensimmäisenä mieleen tuleva tapa yhdistää generoitu ja olemassa oleva ohjelmakoodi lienee se, että generoitu koodi kutsuu valmiin kehyksen tar-

joamia funktioita. Tässä ratkaisussa on siis kyse perinteistä kirjastokomponenttien käytöstä, mutta se on vain yksi, eikä läheskään aina paras mahdollinen vaihtoehto.

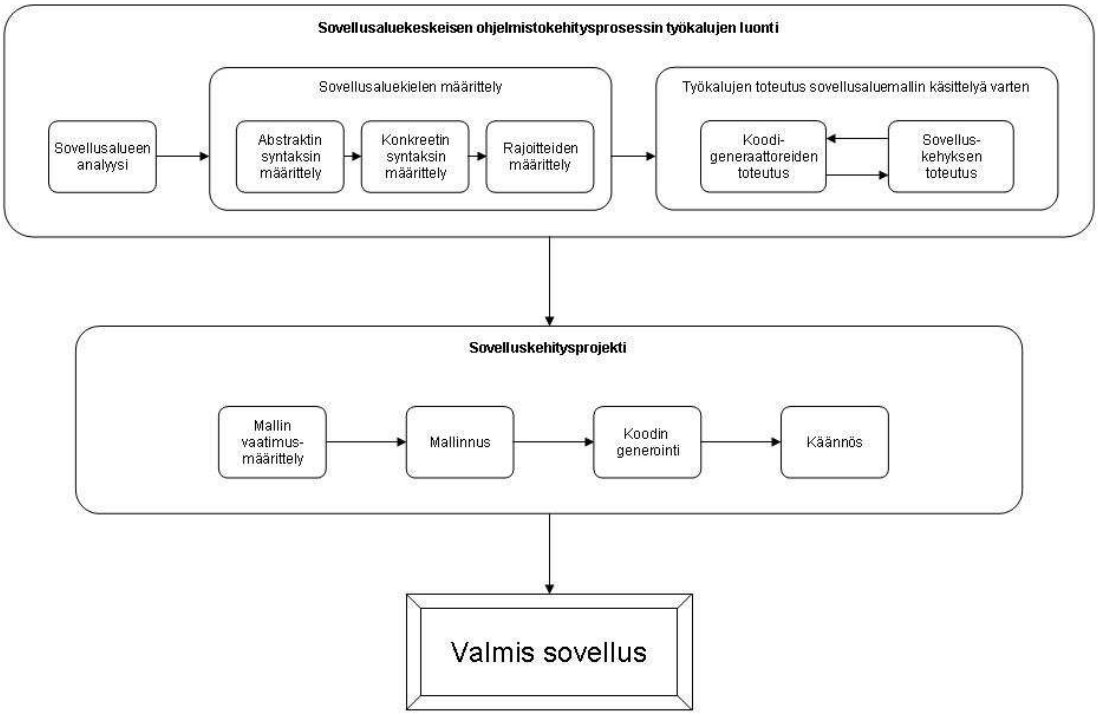
Samantyyppinen, mutta olio-ohjelmointiin paremmin soveltuva tapa on luoda generaattorilla ns. partial-luokkia. Tämä tarkoittaa sitä, että luokan toteutus on jaettu useampaan tiedostoon. C++:ssa tämä on ollut mahdollista jo pitkään, ja partial-luokat ovat nykyään käytössä myös C#:ssa ja VB.NET:ssä (MSDN, 2005). Yleiskäyttöiset ja muuttumattomat luokan osat voidaan siis toteuttaa kehykseen ja generoitu luokka sisällyttää puuttuvat osansa kehyksestä.

Vastaavantyyppinen rakenne voidaan toteuttaa myös perimällä kehykseen toteutettuja abstrakteja luokkia generoidussa koodissa. Tämä mahdollistaa monimutkaisenkin ohjelmalogiikan toteutuksen kehykseen: kehyksessä oleva ohjelmakoodi voi kutsua abstrakteja metodeita, jotka voidaan toteuttaa generoidussa koodissa.

Kehyksen ja generoidun koodin yhteys voi olla myös päinvastainen. Kehyksestä voidaan kutsua ennalta määriteltyjen rajapintojen kautta generoitua koodia. Tällöin esimerkiksi uuden ikkunan avaaminen voidaan tehdä kehyksestä, joka kutsuu generoidun ikkunaluokan metodeja. Tällainen metodi voisi olla esimerkiksi IWindow::Activate(), jossa IWindow on rajapinta (*interface*), joka kaikkien generoitujen ikkunakomponenttien on toteutettava.

2.6 Sovellusaluekeskeisen ohjelmistokehityksen prosessi

Sovellusaluekeskeinen ohjelmistokehitys eroaa perinteisistä menetelmistä olennaisesti prosessinsa osalta. Kehitysprosessi jakautuu kahteen osaan, sovellusaluekielen, sitä käsittelevien generaattoreiden ja sovelluskehyksen määrittelyyn sekä näitä työkaluja hyväksi käyttäen tehtyyn sovelluskehitykseen. Prosessi on esitetty karkealla tasolla kuviossa 3.



KUVIO 3. Sovellusaluekeskeisen ohjelmistokehityksen prosessi

Luoma, Kelly ja Tolvanen (2007) esittävät neljä erilaista näkökulmaa sovellusaluekeskeisen kehitysprosessin käyttöönottoon. Nämä näkökulmat ovat *sovellusalueen asiantuntijan käsitteistö*, *koodigeneraattorin tuottama tuloskoodi*, *tuotetun järjestelmän ilmiasu* ja *sovelluksien välillä esiintyvän vaihtelun käsittely*. Artikkelissa jaotellaan 20 sovellusaluekeskeisen lähestymistavan avulla läpivietyä ohjelmistoprojektia näiden neljän näkökulman mukaan. Näkökulmat eivät sulje toisiaan pois, vaan niistä voidaan ja on usein syytäkin huomioida useampia. Artikkelin loppupäätelmissä todetaan, että soveltuvin näkökulma vaihtelee suuresti sen mukaan, millainen mallinnettava kohde ja tuloksena saatava sovellus ovat, mutta kahden viimeisen näkökulman, eli järjestelmän ilmiasun ja vaihtelun käsittelyn, yhdistäminen näyttäisi olevan usein paras vaihtoehto.

Sovellusaluekielen määrittely alkaa sovellusalueen analyysista. Tämän analyysin tarkoituksena on löytää ne sovellusalueen käsitteet ja suhteet, jotka on mallinnettava sovellusaluekieleen. Samoin tässä vaiheessa on tunnistettava erilaiset

rajoitteet, jotka ohjaavat käsitteiden ja suhteiden muodostamista. Kaikkia sovellusalueen käsitteitä ei ole tarpeen mallintaa, vaan niistä on pystyttävä valitsemaan mallintamisen kannalta merkitykselliset (Kelly & Tolvanen 2008, 228).

Kun sovellusalue on analysoitu, alkaa sovellusaluekielen määrittely. Analyysin tuloksena on syntynyt käsitys kohdealueen käsitteistä ja suhteista, joista muodostetaan sovellusaluekielen abstrakti syntaksi. Abstraktin syntaksin hahmottelun yhteydessä voidaan alkaa jo suunnitella myös konkreettia syntaksia, mutta järjestys on luonnollisesti aina abstraktin syntaksin käsitteistöstä lähtevä. Myös rajoitteita voidaan miettiä tässä vaiheessa, mutta ne voidaan hyvin jättää myös viimeiseksi sovellusaluekielen määrittelyn vaiheeksi, koska niiden testaaminen vaatii mahdollisuuden ainakin jonkinlaisen mallin toteuttamiseen. Lopulta valmista kieltä on vielä testattava laajemminkin. Tärkeimmiksi testattaviksi kohteiksi Kelly ja Tolvanen (2008, 262) esittävät kolmea mallin ominaisuutta, jotka ovat riittävän korkea abstraktiotaso, mallin johdonmukaisuus ja yhtenäisyys sekä mallin koodigeneraattoreille tarjoaman tiedon riittävyys.

Sovellusaluekielen määrittelyn jälkeen voidaan ryhtyä toteuttamaan koodigeneraattoreita ja sovelluskehystä. Generaattoreiden ja sovelluskehyksen toteutusta ei voida tehdä erillisinä prosesseina, vaan niiden yhteistoiminnan varmistamiseksi on syytä pyrkiä antamaan molempien toteutus saman ryhmän työksi. Tämä vaihe on myös iteratiivisin kaikista prosessin vaiheista, sillä kehityksen aikana toisteista ohjelmakoodia siirretään usein kehyksestä generaattoreihin, ja toisaalta toistuvasti samanlaista koodia tuottava osuus voidaan siirtää kehyksessä olevaan funktioon. Vaihe onkin paljolti tasapainon hakemista generaattoreiden ja sovelluskehyksen välille. (Kelly & Tolvanen 2008, 344-345)

Kun työkalut ovat valmiina, niitä käyttäen voidaan kehittää sovellus kyseiselle kohdealueelle. Sovelluksen vaatimusmäärittely voidaan kuitenkin tehdä tavanomaisia ohjelmistokehityksen keinoja käyttäen, jonka jälkeen sovellus mallinnetaan käyttäen sovellusaluekieltä. Mallinnusvaiheen jälkeen ei enää tarvita ma-

nuaalista työtä, koska muuttuva ohjelmakoodi generoidaan suoraan valmiista mallista. Koska sovellusaluekeskeisessä lähestymistavassa pyritään siirtämään kehitystyötä enemmän sovellusalueen asiantuntijoiden vastuulle, ei ideaalitilanteessa varsinaista ohjelmointia tehdä lainkaan. Kun malli on muunnettu generaattoreilla ohjelmakoodiksi, muunnoksen tulos yhdistetään sovelluskehykseen, jolloin kaikki sovellukseen kuuluva ohjelmakoodi on valmiina. Tämän jälkeen jäljellä on enää varsinaisen sovelluksen kääntäminen koodista.

Tämän tutkielman työstämisen aikana sen aiheesta keskusteltaessa useat työ- ja opiskelutoverit huomauttivat leikillisesti lähestymistavan laajemman käyttöönoton tekevän sekä tutkielman kirjoittajan että hänen kollegansa työttömiksi. Jos ohjelmakoodi kerran tuotetaan generaattoreilla, ohjelmistosuunnittelijathan muuttuvat tarpeettomiksi. Tällaista riskiä tuskin kuitenkaan on olemassa. Ensinnäkin lähestymistavassa käytettävien työkalujen toteutus on edelleen ohjelmointityötä, johon tarvitaan alan ammattilaisia. Samoin myös mallinnusvaiheessa tarvitaan asiantuntemusta työkalun käytöstä, jolloin ohjelmistoalan ammattilaisen työpanos on edelleen tarpeen. Kelly ja Tolvanen (2008, 410) esittävät herkullisen vertauksen automatisoidusta ompelimosta, jossa edelleen tarvitaan kyseisen alan ammattilaisia tekemään yhteistyötä asiakkaan kanssa tuotteen materiaaleja, värejä ja muita ominaisuuksia valittaessa. Eihän assemblerin laajamittaisesta käytöstä luopuminen vähentänyt tarvetta ohjelmistoalan työvoimalle, vaan kehitys on ollut aivan päinvastaista.

Edellä esitetty on sovellusaluekeskeisen ohjelmistokehityksen vaiheiden loogi-nen jaottelu. Tästä jaottelusta ei käy ilmi se tärkeä seikka, että varsinkin työkalujen kehittäminen tapahtuu usein iteratiivisesti. Vaikka järjestys pääpiirteissään onkin sovellusaluekielestä liikkeelle lähtevä, joudutaan yleensä myös siihen tekemään muutoksia kun generaattoreita ja sovelluskehystä luodaan. Samoin myös sovelluskehitystä tehtäessä voi nousta esiin seikkoja, jotka edellyttävät myös työkalujen muuttamista.

2.7 Sovellusaluekeskeisen lähestymistavan käyttöönotto ja soveltaminen

Sovellusaluekeskeisen lähestymistavan käyttöönotto ei ole aivan yksinkertainen prosessi. Prosessi koostuu lähestymistapaa varten tarvittavien työkalujen ja käytäntöjen suunnittelusta ja sovelluskehityksessä käytetyn prosessin suunnittelusta. Lähestymistavan käyttöönotto tuo mukanaan suuria muutoksia. Näistä uudistuksista ja muutoksista Wile (2005, 289) erittelee seuraavat viisi: mallinnuskohteet (*artifacts*), mallinnuskieli, työkalut, infrastruktuuri ja metodit. Kyseessä on siis usein hyvin radikaali muutos verrattuna vanhaan työskentelytapaan, ja se saattaa siksi aiheuttaa melkoista muutosvastarintaa (Kelly & Tolvanen 2008, 335).

Kelly ja Tolvanen (2008, 329-356)esittävät seitsemän vaihetta käyttöönotto- ja ylläpitoprosessin läpivienniksi. Nämä vaiheet ovat:

1. Ensimmäisen mallinnettavan sovellusalueen valinta
2. Lähestymistavan käyttöönotossa mukana olevan ryhmän valinta ja organisointi
3. Lähestymistavan käyttökelpoisuuden demonstrointiin käytetyn esittelyversion rakentaminen
4. Valitun sovellusalueen mallintaminen ja sovellusaluekeskeisen lähestymistavan vaatimien työkalujen rakentaminen
5. Pilottiprojekti
6. Prosessin hiominen ja laajempi käyttöönotto
7. Ylläpito ja evoluutio

Mallinnettavan sovellusalueen valinnassa kannattaa Kellyn ja Tolvasen mukaan pyrkiä etsimään alue, jonka mallintaminen on mahdollisimman yksinkertaista

ja joka on rajattavissa suhteellisen pieneksi. Tässä vaiheessa ei ole syytä pyrkiä tekemään kovin mullistavia muutoksia sovellusalueeseen tai kehitysorganisaatioon, vaan vain yrittää mallintaa yksi rajattu alue kunnolla. Useissa organisaatiossa tällaista sovellusalueen valintaa ei ole edes mahdollista tehdä, koska ohjelmistokehitys voi olla keskittynyt tiukasti vain yhdelle tietylle alueelle. (Kelly & Tolvanen 2008, 329-330)

Kun mallinnettava sovellusalue on valittu, täytyy lähestymistavan käyttöönottoa varten valita ryhmä, joka osallistuu prosessiin. Tähän projektiin kannattaa valita kokeneita ja päteviä kehittäjiä. Mukana tulisi olla sekä sovellusalueen että ohjelmistokehityksen asiantuntijoita. Eräs merkki kehittäjän soveltumisesta sovellusaluekeskeisen lähestymistavan toteuttajaksi on hänen pyrkimyksensä automatisoida toistuvia tehtäviä, jopa siten, että hänen kehittämänsä työkalut ovat levinneet muidenkin kehittäjien käyttöön. (Kelly & Tolvanen 2008, 333-335)

Vaikka sovellusaluekeskeisen lähestymistavan tarkoitus on helpottaa sovelluskehitystä, itse mallinnuskielen kehittäminen ja koko kehitysprosessin suunnittelu on vaikeaa ja se vaatii vahvaa asiantuntemusta sekä sovelluskehityksestä että mallinnettavasta alueesta. Kielen kehittäjän on tunnettava kaikki ne sovellusalueen osat, jotka on kieleen sisällytettävä. Tällaisia useaa, mahdollisesti hyvinkin erilaista alaa syvällisesti tuntevia asiantuntijoita on kuitenkin harvassa (Kelly 2005, Mernik ym., 2005), eikä heidän työpanoksensa ole ilmaista. Berard (1993, 187) toteaa sovellusalueen analysoijan olevan ”erään harvinaisimmista ohjelmistoalan ammattilaisten lajeista”. Yleensä edellä kuvattua asiantuntemusta löytyy organisaatioista, joissa on jo aiemmin tehty ratkaisuja kyseiselle sovellusalueelle perinteisillä menetelmillä. Tällaisesta tilanteesta Kelly (2005, 3) mainitsee esimerkkeinä mm. tuoteperheet ja jatkuvasti kehitettävät järjestelmät.

Paitsi tuntemusta sovellusalueesta, sovellusaluemallinnuksessa käytettävän työkalun kehittämisessä tarvitaan hyvää suunnittelu- ja ohjelmointitaitoa. Sekä koodigeneraattoreiden että oikeellisuustarkistusten toteuttaminen on joskus

hyvin hankalaa. Itse toiminnallisuuden sisäistäminen ja sen toteutus on usein jo sinällään vaikeaa, mutta näiden komponenttien tulisi vieläpä toimia tehokkaasti. Jos esimerkiksi uuden objektin lisääminen malliin laukaisee vaikkapa muita objekteja läpikäyvän oikeellisuustarkistusrutiinin, täytyy tämän rutiinin olla hyvin tarkkaan suunniteltu, jotta mallin kasvaessa tarkistus ei kestäisi suhteettoman kauan. (Cook ym. 2007, 284) Toisaalta myös generaattoreiden tuottaman, varsinaisen sovelluskoodin tehokkuus on tärkeää, mikä sekin voi olla haasteellinen tavoite. Sovellusaluemallinnuksen käyttöönottoon täytyy siis varata resursseja, jotka ovat kaikkein halutuimpia myös muun käynnissä olevan kehitystyön käyttöön. Tästä syystä sovellusaluemallinnuksen käyttöönottokustannukset ovat korkeat. Tästä kustannustekijästä huolimatta sovellusaluekeskeisen lähestymistavan tuoma kehitystyön tehostuminen yleensä korvaa alkuvaiheessa käytetyn työn moninkertaisesti. (Tolvanen 2005, 23)

Krahn, Rumpe ja Völkel (2006) ehdottavat sovellusaluekeskeisen kehitysprojektin jäsenten jaottelua neljään rooliin. Tämä jaottelu on tehty ketterän ohjelmistokehityksen näkökulmasta, mutta tästä huolimatta sitä voidaan käyttää ainakin eräänä vaihtoehtona pohdittaessa minkä tahansa sovellusaluekeskeisen ohjelmistoprojektin organisaatiota. Artikkelissa esitetyt roolit ovat kielen kehittäjä (*language developer*), työkalun kehittäjä (*tool developer*), kirjastojen (eli sovelluskehyksen) kehittäjä (*library developer*) sekä tuotteen kehittäjä (*product developer*). Kielen kehittäjä on vastuussa sovellusaluekielen kehittämisestä, ja tämän kehitystyön lähtökohtana ovat tuotteen kehittäjän tarpeet. Työkalun kehittäjä vastaa koodigeneraattoreista sekä generaattoreiden ja muiden komponenttien integroinnista. Kirjastojen kehittäjä taas vastaa sovelluskehyksestä. Tuotteen kehittäjä käyttää muiden tuottamia työkaluja ja vastaa mallinnustyöstä. (Krahn ym. 2006)

Aagedal ja Solheim (2004) käsittelevät artikkelissaan malliperustaisen ohjelmistokehityksen organisointia. Koska malliperustaisella ja sovellusaluekeskeisellä lähestymistavalla on useita yhtäläisyyksiä, voidaan heidän malliaan tarkastella

myös tämän tutkielman kohteen kannalta. Heidän ratkaisunsa on hyvä mainita myös siksi, että se ei ole erityisesti ketteriin kehitysmenetelmiin kohdennettu. Artikkelissa on jaettu malliperustaisen ohjelmistokehityksen prosessiin osallistuvat kahteen ryhmään, metaryhmään ja projektiryhmään. Näistä ensimmäinen vastaa kehitystyökaluista ja -metodeista, ja jälkimmäinen sovelluskehityksestä näillä työkaluilla. Projektiryhmä siis tulee mukaan prosessiin edellä esitetyn jaottelun viimeisessä vaiheessa.

Metaryhmän kokoonpano on melko samanlainen Krahnin ym (2006) esittämän kanssa, vaikkakin toimenkuvat onkin nimetty hieman eri tavalla ja joitain eroja vastuualueissakin on, mikä johtunee ainakin osittain siitä, että malliperustainen lähestymistapa jakautuu alustariippumattomaan ja alustakohtaiseen osaan (OMG 2008). Ryhmään kuuluvat metamallin kehityksessä sovellusalueen käsitteiden ja suhteiden mallintamisesta vastaava sovellusalueen asiantuntija (*domain expert*), sovellusalustan asiantuntija (*platform expert*), kielen kehittäjä (*language engineer*) ja muunnosten määrittelijä (*transformation specifier*). Näiden lisäksi mukana on koko metodista vastaava henkilö (*method engineer*), jonka tehtävänä on tunnistaa ja järjestellä tarvittavat työvaiheet, löytää mallinnettavat kokonaisuudet ja määrittää tarvittavat muunnokset. Hän vastaa myös prosessin organisoinnista. (Aagedal & Solheim 2004)

Kun kehitysryhmä on saatu kootuksi, seuraava vaihe on lähestymistavan toimivuuden osoittaminen. Tätä varten luodaan raakaversiot sovellusaluekeskeisen lähestymistavan käyttöönottoon tarvittavista työvälineistä. Tässä vaiheessa ei ole vielä tarkoitus toteuttaa kattavaa ratkaisua, vaan vain jokin sen osa-alue. Toteutuksen on kuitenkin oltava toimiva, jotta sitä voidaan esitellä organisaation johdolle ja muille sidosryhmille. Kehitys on syytä aloittaa mallinnuskielestä ja metamallista, koska siihen tehtävät muutokset aiheuttavat myöhemmässä vaiheessa muutoksia myös koodigeneraattoreihin. Jos kieli on kehitetty mahdollisimman kattavaksi jo ennen generaattoreiden toteutusta, työvaiheen läpivienti nopeutuu. (Kelly & Tolvanen 2008, 335-338) Tärkeä seikka on se, että

myöhemmässä vaiheessa tehtävät muutokset voivat työläytensä vuoksi jäädä jopa kokonaan tekemättä, jolloin mallinnuskieli ei ole niin kattava ja toimiva kuin se voisi olla (Kelly & Tolvanen 2008, 314).

Kun lähestymistavan toimivuus valitulla sovellusalueella on edellisten vaiheiden kautta saatu todistettua, jatketaan sovellusalueen mallinnusta ja työkalujen kehittämistä valmiiksi. Tässä vaiheessa tulee pyrkiä viemään prosessi loppuun valitun sovellusalueen osalta ennen muille sovellusalueille siirtymistä. Tällä tavoin myös mahdollisesti myöhemmin esiin tulevista tilanteista ja ongelmista saatuja kokemuksia voidaan hyödyntää muiden sovellusalueiden kohdalla. Näiden kokemusten perusteella voi olla tarpeen tehdä esittelyversioon suuriakin muutoksia, ja mahdollisesti jopa aloittaa koko prosessi alusta. (Kelly & Tolvanen 2008, 339-341)

Sovellusaluekielen, koodigeneraattoreiden ja sovelluskehyksen rakentaminen kannattaa Kellyn ja Tolvasen (2008, 341-345) mukaan järjestää siten, että ensin määritellään sovellusaluekieli. Kun kielellä pystytään mallintamaan sovellusalueen käsitteet, säännöt ja rajoitteet ja kun sille on määritelty alustava esitystapa, voidaan siirtyä toteuttamaan generaattoreita ja kehystä. Näiden kehitys on usein rinnakkaista. Tämä johtuu siitä, että kehityksen aikana ohjelmakoodin osia siirretään kehyksestä generoitavaksi ja päinvastoin. (Kelly & Tolvanen 2008, 344-345). Generaattoreiden kehitys on usein luontevaa tehdä siten, että toimiva koodin osa siirretään sellaisenaan generaattorin tulosteeksi, jonka jälkeen generaattoria lähdetään muokkaamaan lisäämällä kontrollilohkoja ja muita toisteisuutta vähentäviä rakenteita (Cook ym. 2006, 52; Kelly & Tolvanen 2008, 269).

Cook ym. (2007) näkevät sovellusaluekielen keinona parametrisoida ohjelmakoodia. Tällä he viittaavat siihen, että saman sovellusalueen eri sovellukset ovat monelta osiltaan samanlaisia, jolloin sovellusaluekielellä vain määritellään eroavat osat. Täten on luontevaa kehittää koodigeneraattorit ja sovelluskehys

siten, että vertaillaan eri sovelluksia ja niiden yhteiset osat jätetään kehykseen. Muuttuvat osat taas tuotetaan generaattoreilla, joita sovellusaluekieli siis ohjaa. (Cook ym. 2007, 44) He näkevät myös generaattoreista lähtevän kehitysprosessin mahdollisena, jolloin siis mallinnuskieli suunniteltaisiin myöhemmin, mutta pitävät parhaana molempien tapojen yhtäaikaista käyttöä. Syynä tähän on se, että mallin tasolta katsottuna voi olla hankalaa huomata generaattoreiden metamallille asettamat vaatimukset. Tuloksena voi olla metamalli, joka on liian monimutkainen ollakseen käyttökelpoinen. (Cook ym. 2007, 46-47) Koska kuitenkin tuntuisi järkevältä pyrkiä siihen, että metamalli kuvaa sovellusalueen mahdollisimman hyvin, on tämä näkökulma hieman outo: tällä tavoinhan rajoitettaisiin metamallin määrittelyä koodigeneraattoreiden tarpeiden perusteella, mikä ei vaikuta kovin kestävältä ratkaisulta.

Kun kehitystyökaluista on valmiina ensimmäinen kokonaisuudessaan toimiva versio, on aika valita pilottiprojekti. Tämän projektin tulisi olla mahdollisimman tavanomainen ja keskiverto kyseisen sovellusalueen projekti, jotta siitä saatavien kokemusten pohjalta tehtävät prosessin hienosäädöt hyödyttäisivät lähestymistavan kehitystä mahdollisimman paljon. Pilottiprojektin aikana on todennäköistä, että myös metamalliin tulee muutoksia. Kuten edellä mainittiin, tästä seuraa muutostarvetta myös koodigeneraattoreihin ja mahdollisesti myös sovelluskehykseen, mutta työläydestään huolimatta niitä on tässä vaiheessa usein mahdotonta välttää. Koska muutoksia todennäköisesti tulee suhteellisen paljon, on versiointiin kiinnitettävä erityistä huomiota. Paitsi kehitystyökalujen, myös niillä luodun mallin versio on oltava selvillä, jotta mallin käsittelyyn voidaan käyttää oikeita työkaluja. Pilottiprojektin jälkeen organisaatiolla pitäisi olla toimivat työkalut sovellusalueen mallintamiseen ja sovelluskehitykseen. (Kelly & Tolvanen 2008, 345-347)

Kun lähestymistavan tekninen toteutus on valmis, pitää sitä vielä hioa, jotta se olisi valmis laajempaan tuotantokäyttöön. Tähän kuuluvat mm. mallinnuskielen visuaalisen ilmeen viimeistely, kielessä käytettävien sääntöjen ja rajoitteiden

viimeistely sekä prosessin mahdollisimman laaja automatisointi. (Kelly & Tolvanen 2008, 349-350) Samassa yhteydessä on syytä kehittää myös prosessi kehitysympäristöjen päivittämiseksi sekä viimeistellä dokumentointi ja koulutusmateriaali (Kelly & Tolvanen 2008, 350-351).

Kun sovellusaluekeskeinen ohjelmistokehitysprosessi on otettu käyttöön, siirtyy menetelmä ylläpitovaiheeseen. Eri työkaluihin tehdään korjauksia, metamallia kehitetään ja laajennetaan, ja itse sovellusaluekin voi muuttua. Tästä syystä on tärkeää pyrkiä säilyttämään mahdollisimman moni kehitysvaiheessa mukana olleista henkilöistä kehitysryhmässä, jotta varmistetaan työkalujen tuntemus. (Kelly & Tolvanen 2008, 354) Sovellusaluekielten elinkaari on Kellyn ja Tolvasen (2008, 354-355) mukaan usein pidempi kuin yleiskäyttöisten mallinnuskielten, joten ylläpitovaihe voi olla hyvinkin pitkä.

Elinkaaren aikana metamalliin todennäköisesti tarvittavien muutosten kannalta hankaluutena on jo pilottiprojektin yhteydessä mainittu versioiden hallinta. Tietyn metamallin mukainen malli ei välttämättä vastaa enää muuttunutta, uudempaa metamallia, joten konfiguraation hallintaa vanhojen versioiden ylläpidon mahdollistamiseksi tarvitaan. Versionhallintaan voidaan hyvin käyttää nykyäänkin laajasti käytettyjä työkaluja, koska kyse on lopulta vain erilaisten tiedostoversioiden hallinnasta, kuten missä tahansa ohjelmistokehityksen menetelmässä. Toinen huomioitava osa-alue on uudelleenkäyttö, johon voidaan pyrkiä esimerkiksi jakamalla malli useisiin osamalleihin tai käyttämällä samoja elementtejä eri malleissa. (Kelly & Tolvanen 2008, 397-404) Kohdealueen kehittyessä ja uusien vaatimusten syntyessä sovellusaluekieltä on myös kehitettävä, mikä voi joissain tapauksissa olla työläs toimenpide. Tätä varten Cleenewerck (2003) esittää avainsanaperustaisen ohjelmoinnin (*keyword base programming, KBP*) keinona helpottaa sovellusaluekielen evoluutiota ylläpitovaiheessa. Tilanteessa, jossa vanha malli halutaan muuntaa uuden metamallin mukaiseksi, voi eräänä ratkaisuna olla de Geestin, Savelkoulin ja Alikosken (2007) esittämä sovellusaluekieli mallin muuntamiseksi.

Aiemmin esitettiin Aagedalin ja Solheimin (2004) ehdotus malliperustaisen prosessin tehtäväjaosta metaryhmän osalta. Heidän mallinsa mukaisesti ylläpitovaiheen projektiryhmään kuuluvat sovellussuunnittelija (*application designer*), joka käyttää metaryhmän tuottamia työkaluja sovellusten toteutukseen, järjestelmäanalyytikko (*system analyser*), joka vastaa järjestelmän toimivuudesta, ja testaaja (*system tester*). Nämä vastaavat perinteisiä ohjelmistokehityksen rooleja, mutta vaativat erityisosaamista lähestymistavan työkalujen ja toimintatapojen osalta. (Aagedal & Solheim 2004, 113) Tämä jaottelu ei ehkä sovellu sovellusaluekeskeisen lähestymistavan käyttöön aivan suoraan. Tavoitteenahan on usein nostaa abstraktiotasoa ja sovelluksen kehityksen automaatiota niin paljon, että mallinnuksen voisi tehdä jopa henkilö, joka ei ole ohjelmistokehityksen ammattilainen. Tällaisessa tilanteessa varsinainen sovelluskehitys vaatii vain mallintajan ja testaajan, koska mahdolliset mallin ulkopuoliset muutoksethan olisivat työkaluun kohdistuvia, ja siten metaryhmän tehtäviä.

Pitkänen ja Mikkonen (2006) esittelevät ehdotuksen lähestymistavan käyttöönottokynnyksen madaltamiseksi *kevennetyn sovellusaluemallinnuksen* avulla. Tällä tarkoitetaan eräiden sovellusaluemallinnuksen piirteiden yhdistämistä mm. malliperustaiseen ohjelmistokehitykseen siten, että mallinnusmenetelmä ei ole täysin sidottu sovellusalueeseen, vaan on yleisluontoisempi, eikä koko mallinnusprosessia tai kaikkien osa-alueiden mallinnusta viedä läpi tiukasti sovellusaluemallinnuksen periaatteiden mukaisesti. Näin voidaan laskea käyttöönottokustannuksia, mutta todennäköisesti menetetään osa tuottavuuden kasvusta ja muista sovellusaluemallinnuksen hyödyistä. (Pitkänen & Mikkonen 2006) Toinen tapa madaltaa sovellusaluekielen kehityksen ja lähestymistavan käyttöönoton aloituskynnystä on esitelty Bierhoffin, Liongosarin ja Swaminatan (2006) artikkelissa, jossa he kuvaavat prosessin sovellusaluekielen vähittäiseksi kehittämiseksi. Heidän ajatuksensa on lähteä liikkeelle yhdestä siemen- eli esimerkkisovelluksesta (*seed application*) toteuttaen sovellusaluekielen tämän yhden sovelluksen tuottamiseksi. Tämän jälkeen kieltä laajennetaan seuraavan to-

teutettavan sovelluksen tarpeiden pohjalta. Tällä tavoin kielen ylläpito ja kehitys tapahtuu pienin askelin ja on heidän kokemuksensa mukaan vieläpä helppoa.

2.8 Yhteenveto

Tässä luvussa esiteltiin mallinnuksen ja sovellusaluemallinnuksen käsitteistöä tärkeimmiltä osiltaan. Lisäksi käytiin läpi sovellusaluekeskeisen ohjelmistokehityksen periaatteita sekä esiteltiin kirjallisuuden pohjalta tämän lähestymistavan hyväksi havaittuja käytäntöjä ja hyötyjä verrattuna perinteisempiin lähestymistapoihin. Myös lähestymistavan käyttöönottoprosessia kuvattiin.

Edellä esitettyjen seikkojen pohjalta voidaan todeta sovellusaluekeskeisen ohjelmistokehityksen olevan ainakin teoreettiselta kannalta tarkasteltuna hyvin lupaava ratkaisuvaihtoehto ohjelmistokehityksen ikuisuusongelmiin. Sen perimmäinen idea, abstraktiotason nosto, vaikuttaisi toimivalta, varsinkin jos sen avulla voidaan todella ottaa kehitysprosessiin mukaan myös sovellusalueen asiantuntijoita.

Ilman käyttökelpoisia työkaluja sovellusaluekeskeinen ohjelmistokehitys olisi vain kokoelma hyviä ideoita, tai korkeintaan ohjelmistoalan keino nopeuttaa joitain sisäisiä prosessejaan. Työkaluja on viime vuosina kehitetty, ja osa niistä on jo suhteellisen kypsässä kehitysvaiheessa. Seuraavassa luvussa tutustutaan erääseen näistä työkaluista.

3 DSL TOOLS

Sovellusaluekielen kehittämiseen on olemassa useita työkaluja. Eräitä esimerkkejä näistä ovat MetaCasen kehittämä MetaEdit+ (MetaCase 2008) ja Eclipse-ympäristöön kehitetty Generic Eclipse Modeling System (GEMS) (The Eclipse Foundation, 2008). Tässä tutkielmassa keskitytään Microsoftin kehittämään meta-sovellusaluekieleen ja kehitysympäristöön, joka on nimeltään DSL Tools. Tässä luvussa esitellään DSL Tools siltä osin kuin se tutkielman kannalta on tarpeen. Työkalun esittely pohjautuu teokseen Cook ym. (2007).

3.1 Software Factories -konsepti

Microsoft on jo jonkin aikaa pyrkinyt kehittämään työkaluja ohjelmistokehityksen automatisoinniksi. Tähän tavoitteeseen Microsoft suuntaa Software Factories -konseptillaan. Konseptin nimi tarkoittaa ohjelmistotehdasta, ja se juontaa juurensa tavoitteeseen kehittää ohjelmistotuotantoa vanhempien teollisuudenalojen tapaiseksi, liukuhihnalta keskenään samanlaisia tuotteita kustannustehokkaasti ja nopeasti tuottavaksi, prosesseiltaan kypsäksi teollisuudenalaksi. Greenfield ym. (2004, 155) kutsuvat tätä kehitystä *ohjelmistokehityksen teollistamiseksi*.

Greenfield ym. (2004, 161) luettelevat Software Factories -konseptin kannalta kriittiset innovaatiot, joiden avulla ohjelmistokehityksen teollistaminen voidaan toteuttaa. Nämä innovaatiot ovat:

1. ohjelmiston uudelleenkäytön lisääminen ohjelmistotuoteperheiden kautta,

2. sovellusten kokoaminen itsensä metatiedolla kuvaavista komponenteista,

3. käsin kirjoitettavan ohjelmakoodin määrän vähentäminen sovellusaluekielten ja muiden työkalujen avulla ja

4. projektien kokoluokan kasvattaminen, maantieteellinen hajautus ja tuotteiden elinkaaren pidentäminen, kuitenkaan kehitysprosessin ketteryyttä menettämättä.

Tämän tutkielman kannalta näistä kiinnostavin on lähinnä kolmas kohta ja sen asettamien vaatimusten täyttämiseksi kehitetty työkalu, Microsoft DSL Tools.

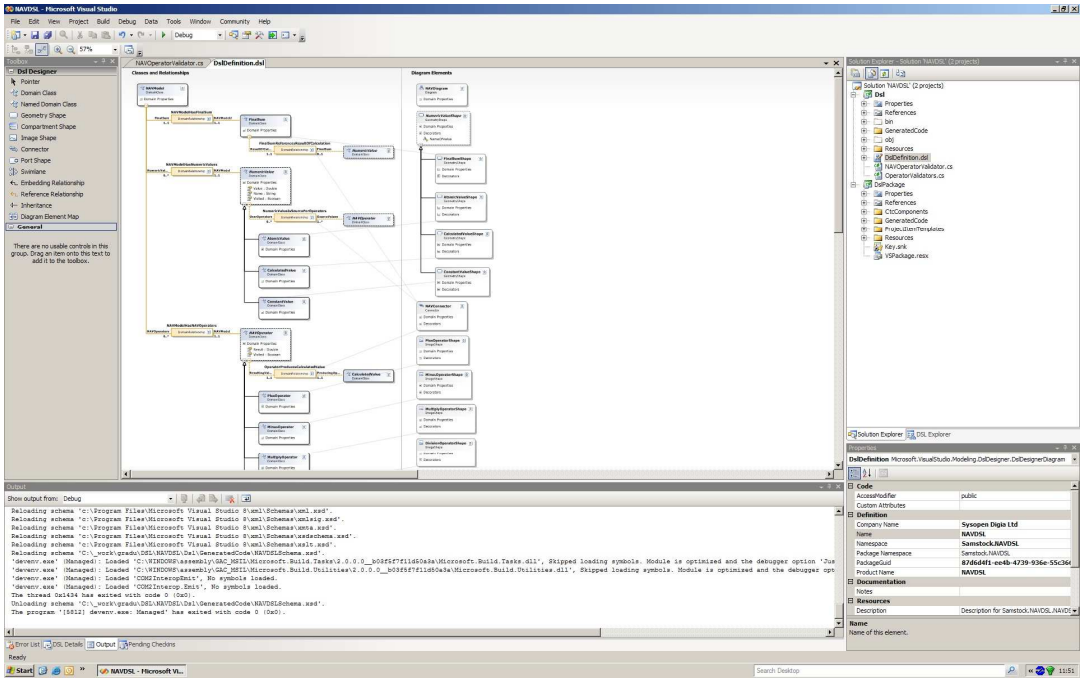
3.2 DSL Toolsin rakenne

DSL Tools on osa Visual Studion *SDK*:ta (*Software Development Kit*), ja se on vapaasti ladattavissa Microsoftin sivuilta. SDK:n käyttöönotto vaatii kuitenkin Visual Studiosta vähintään professional-tasoisen asennuksen. Tämän tutkielman käytännön osuudessa käytetään Visual Studio 2005:ä, joka toimii yhdessä .NET Frameworkin 2.0-version kanssa. Visual Studio SDK:sta on käytössä versio 4.0. Mainittavan arvoinen seikka on se, että DSL Toolsin kuvauskieli on paitsi metakieli, myös meta-metakieli, sillä kehitysympäristö on itsekin kehitetty DSL Toolsilla (Cook 2007, 27). Kehitysympäristö muistuttaa Visual Studion käyttöliittymää (vrt. kuvio 4).

Metamallin kehitysympäristö, eli metakieli ja sen graafinen käyttöliittymä, perustuu kahteen erilliseen projektitiedostoon. Nämä ovat DSL-projekti ja DSLPackage-projekti. Ensimmäinen sisältää sovellusaluekielen määritykset, eli kaiken sen mitä metamallia kehitettäessä on saatu aikaan. Projektissa ovat mukana myös mallin muunnoksessa syntynyt koodi. DSLPackage-projekti taas sisältää tarvittavat komponentit kehitetyn sovellusaluekielen integroimiseksi Visual Studioon. DSLPackage on siis DSL Toolsin generoiman koodin ja työkalun sovelluskehyksen yhdistävän ”liimakoodin” sisältävä komponentti.

Sovellusaluekielen kehitysprosessissa DSL Toolsin osuus alkaa uuden sovellusaluekielen syntaksin määrittelystä. Ennen tätä vaihetta mallinnettava sovellusalue on kartoitettu ja tarpeelliset analyysit on tehty. Sovellusalueen kuvaamiseen tarvittava syntaksi jakautuu luvussa 2 kuvatulla tavalla abstraktiin ja

konkreettiin syntaksiin. Tämä jako on nähtävissä myös syntaksin määritysnäytössä (Kuvio 4). Seuraavaksi kuvataan tarkemmin abstraktin ja konkreetin syntaksin määrittelyä.



KUVIO 4. DSL Toolsin käyttöliittymä

3.3 Abstrakti syntaksi

Metamallin määritysnäytön vasen puoli (Kuvio 4) edustaa sovellusalueen abstraktia syntaksia. Tässä osassa määritellään siis luokat, jotka vastaavat mallinnettavia käsitteitä ja niiden välisiä suhteita. Uuden luokan lisääminen tapahtuu raahaamalla vasemmalla olevasta työkalulaatikosta haluttua sovellusalueen luokkaa (*Domain Class*) edustava symboli suunnittelupinnalle. Tämän jälkeen luokan ominaisuuksia pystyy muuttamaan näytön oikeassa alakulmassa olevasta ominaisuuslaatikosta. Eräiden ominaisuuksien lisääminen onnistuu myös napsauttamalla hiiren oikeaa nappulaa luokan otsikon päällä.

Abstraktin syntaksin lähtökohtana toimii kehitettävää mallia edustava *juuriluokka* (*root class*). Kaikki muut mallinnettavat luokat liittyvät suoraan koos-

tesuhteella tai välillisesti muiden luokkien kautta tähän luokkaan. Tämä koostesuhde voi olla kardinaalisuudeltaan N:N, vaikkakin useimmiten suhde on 1:N tai 1:1. Yhteen malliin voi siis yleensä sisältyä yksi tai useampi sisällytetyn luokan ilmentymä, ja yksi tällainen ilmentymä kuuluu yhteen ja vain yhteen malliin.

Suhteita on kolmenlaisia: koostesuhde (*embedding relationship*), viitesuhde (*reference relationship*) ja periytymissuhde (*inheritance*). *Koostesuhteen* avulla voidaan määrittää jonkin luokan ilmentymien olevan toisen luokan ilmentymien osia. *Viitesuhteella* määritetään viite luokasta toiseen, kuitenkin niin että viitattava luokka on erillinen käsitteensä. *Periytymissuhteella* taas ilmaistaan luokan olevan toisen luokan erilaistettu alaluokka. Kullekin suhdetyypille syntyy automaattisesti *yhteysrakentaja* (*connection builder*), joka huolehtii yhteyden oikeellisuudesta (ks. kohta 3.5).

Luokan periytymiselle voidaan määritellä sääntöjä. Luokka voidaan määrittää *abstraktiksi* (*abstract*), jolloin siitä ei voida luoda ilmentymää, vaan siitä voidaan ainoastaan periä alaluokkia. Tässä tapauksessa siis luokasta täytyy olla periytymissuhde toiseen luokkaan, jotta se olisi semanttisesti mielekäs. Toinen oletustilasta poikkeava periytymissääntö on *sinetöity* (*sealed*), joka on päinvastainen suhteessa abstraktiin luokaan: siitä voidaan luoda ilmentymä, mutta sitä ei voi periä.

3.4 Konkreetti syntaksi

Määritysnäytön (vrt. Kuvio 4) oikealla puolella esitetään mallin konkreetti syntaksi. Konkreetin syntaksin tarkoitus on määrittää mallin esitys. Visuaalisen kielen tapauksessa tässä siis määritellään, minkälaisella graafisella muodolla esitetään halutut abstraktin syntaksin luokat ja yhteydet. Konkreetin syntaksin osaset ovat nekin luokkia, ja niiden välille voidaan määrittää samanlaisia suhteita kuin abstraktin syntaksin luokille. Suhteiden määrittämiselle on kuitenkin

eräitä rajoituksia, esimerkiksi koostesuhteen luominen ei ole mahdollista tiettyjen luokkien välille.

DSL Toolsin tarjoamat konkreetin syntaksin perusluokat on lueteltu alla olevassa listassa (Cook ym. 2007, 137, 147-154). Näistä geometrinen ja kuvaelementti sekä yhdistin ovat käytössä tämän tutkielman käytännön osassa.

- *kaavio* (*diagram*) kuvaa mallin esittämiseen käytettävää kaaviota. Tämän luokan ominaisuuksien kautta voidaan määritellä eräitä kaavion ominaisuuksia, kuten kaavion taustaväri.

- *geometrinen elementti* (*geometry shape*), joka on yksinkertainen, luokkaa esittävä muoto, joko nelikulmio tai ellipsi.

- *lokerikkoelementti* (*compartment shape*), joka sisältää yhteen tai useampaan osaan jaetun listan. Lista voi sisältää esimerkiksi ko. luokan ominaisuudet ryhmiteltynä kategorioittain.

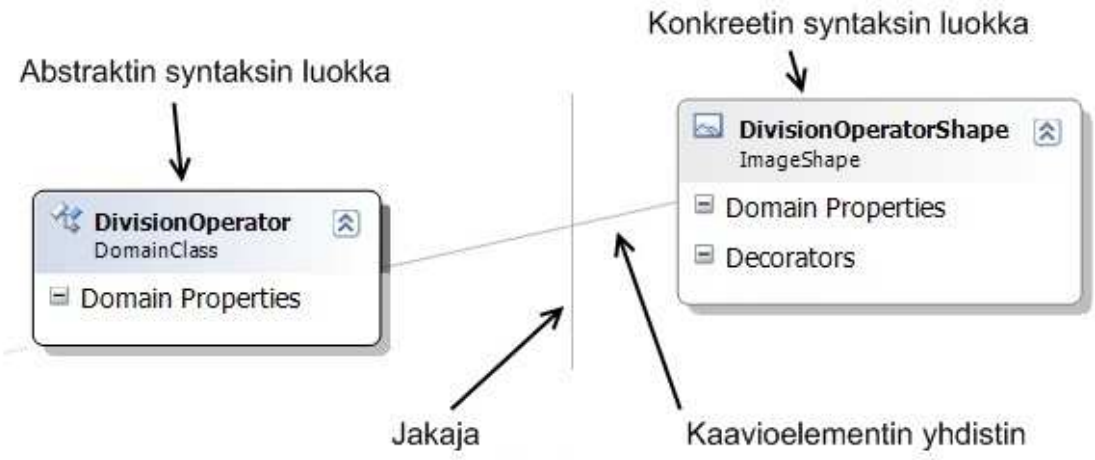
- *kuvaelementti* (*image shape*), jolle voidaan määrittää vapaavalintainen kuva graafiseksi ilmiasuksi.

- *yhdistin* (*connector*), jolla pystytään yhdistämään kaksi elementtiä abstraktin syntaksin määrittämällä tavalla. Graafisesti tämä luokka esitetään viivana, johon voidaan määritellä esimerkiksi nuoli yhteyden kohdepäähän.

- *portti* (*port*) on komponentti, joka voidaan liittää toisen komponentin ulkoreunaan. Usein tällä kuvataan isäntäkomponenttiin tulevaa tai siitä lähtevää suhdetta tai tietovirtaa.

- *mallin jakaja* (*swimlane*). Malli voidaan jakaa useaan eri loogiseen osaan. Esimerkkinä tästä on DSL Toolsin abstraktiin ja konkreettiin syntaksiin jaettu määritysnäyttö.

DSL Toolsin työkalulaatikosta löytyy vielä yksi yhdistin. Tällä *kaavioelementtien yhdistimellä* (*diagram element map*) yhdistetään metamallin abstraktin ja konkreetin syntaksin luokat toisiinsa. Esimerkki tästä on nähtävillä kuviossa 5, jossa abstraktin syntaksin luokka DivisionOperator on yhdistetty vastaavaan konkreetin syntaksin luokkaan DivisionOperatorShape, jakajan erottaessa abstraktin ja konkreetin syntaksin toisistaan. Sovellusaluekielen yhdistimien liittäminen vastaaviin suhdetyyppeihin tapahtuu samalla tavoin.



KUVIO 5. Abstraktin ja konkreetin syntaksin luokkien yhdistäminen

Määriteltyjen luokkien käyttöönottamiseksi sovellusaluekielessä on ne vielä lisättävä työkalulaatikkoon. Tämä tapahtuu luomalla uusi *yhteys*- tai *elementtityökalu* (*connection tool*, *element tool*). Kullekin työkalulle voidaan määritellä oma kuvakkeensa. Työkalu liitetään haluttuun abstraktin syntaksin luokkaan, jolloin kyseiseen luokkaan liitetty konkreetin syntaksin luokka tulee myös määritellyksi. Koska työkalu liitetään nimenomaan abstraktin syntaksin luokkaan, ei kielen logiikan kannalta synny sekaannusta, vaikka sama konkreetin syntaksin symboli kuvaisikin kahta eri abstraktin syntaksin käsitettä. Tällaista ratkaisua pohdittaessa on syytä kuitenkin ottaa huomioon, että se ei saa haitata mallin suunnittelua tai sen hahmottamista.

3.5 Oikeellisuustarkistukset

Tärkeä osa sovellusaluekieltä on sillä kehitetyn mallin oikeellisuuden tarkistaminen. Tarkistukset perustuvat *rajoitteisiin* (*constraint*), jotka on mahdollista tarkistaa ohjelmallisesti. Rajoitteita on kahta tyyppiä, *pehmeä* (*soft constraint*) ja *kova* rajoite (*hard constraint*). Näistä ensimmäinen tarkoittaa rajoitetta, jonka ei tarvitse täyttyä kaikissa tilanteissa, ja jälkimmäinen taas rajoitetta, jota ei saa rikkoa missään vaiheessa (Cook ym. 2007, 277). Rajoitteet määritellään metamalliin.

DSL Tools tarjoaa oikeellisuustarkistuksen tekoon kaksi pääasiallista tapaa. Näistä ensimmäinen, automaattisesti metamallin perusteella syntyvä tarkistus on eri elementtien välisten suhteiden oikeellisuuden tarkistus. Yhdistimiä ei ylipäänsä voi piirtää sellaisten elementtien välille, joille ei yhdistimellä kuvattavan kaltaista yhteyttä ole abstraktissa syntaksissa määritelty. Tämän lisäksi myös suhteen kardinaalisuus tarkistetaan, jolloin esimerkiksi 1:1-suhteen voi piirtää liittymään samaan elementtiin vain kerran. Tämä on esimerkki kovasta rajoitteesta, koska ylimääräisen suhteen piirtäminen ei ole ylipäänsä mahdollista.

Oikeellisuustarkistus tehdään myös mallin tallennus- ja muunnosvaiheessa, jolloin voidaan löytää esimerkiksi mallintamatta jääneet suhteet. Koska metamalli vaatii aina yhden juuriluokan, voidaan tässä vaiheessa tarkistaa, että kaikki juuriluokalle pakollisiksi määritetyt suhteet ovat olemassa, eli esimerkiksi tilakaavion ensimmäinen siirtymä alkupisteestä on määritetty. Samoin huomataan vastaavat puutteet muidenkin luokkien suhteissa. Tämä on esimerkki pehmeästä rajoitteesta, koska se sallii mallin olevan vaillinainen aina tallennus- tai muunnosvaiheeseen asti.

Oikeellisuustarkistuksia voidaan myös määrittää ohjelmallisesti metamalliin. Tämä tapahtuu asettamalla halutulle yhteysrakentajalle *räätälöity hyväksyntä* -ominaisuus (*custom accept*) päälle. Tämän ominaisuuden ollessa päällä DSL Toolsin generoima koodi ei kutsu oletusarvoisesti käytettävää tarkistusmetodia,

vaan metamallin suunnittelijan on luotava metodi käsin. Tällä tavoin voidaan tarkistuslogiikka luoda joustavasti vastaamaan kulloisenkin metamallin tarpeita. Kuten automaattisen tarkistuksen kautta asetetut rajoitteet, myös räätälöidyt tarkistukset voidaan määrittää koviksi tai pehmeiksi rajoitteiksi.

3.6 Mallin muunnokset

Kun DSL Toolsin piirtoikkunalla kehitetty metamalli on valmis, pitää se ottaa käyttöön. Koska DSL Tools on itsekin sovellusaluemallinnukseen tarkoitettu työkalu, se toimii kuten sellaisen on aiemmissa luvuissa esitetty toimivan. Aluksi metamalli pitää muuntaa ohjelmakoodiksi, ja se onnistuu yksinkertaisesti painamalla muunnosnappia metamallin selaimen yläreunassa (vrt. Kuvio 4). Kun muunnos on tehty onnistuneesti, pitää syntynyt ohjelmakoodi vielä kääntää, ja se tapahtuu samasta valikkotoiminnosta kuin Visual Studiossa yleensäkin. Käännöksen onnistuttua voidaan uuden sovellusaluekielen käyttöliittymä käynnistää. (Cook ym. 2007, 66)

Tässä vaiheessa uuden sovellusaluekielen mallinnustoiminnallisuus on valmis. Käynnistetyn käyttöliittymän kautta voidaan piirtää objekteja ja niiden välisiä suhteita, mutta niiden pohjalta ei synny sovellusta itsestään. Tätä varten pitää kehittää mallin muunnoksessa käytettävät koodigeneraattorit. Uuden generaattorin luominen tapahtuu lisäämällä uusi tiedosto projektiin ja määrittämällä se periytymään ModelingTextTransformation-luokasta. Lisäksi tiedostolle on asetettava CustomTool-ominaisuuden arvoksi TextTemplatingFileGenerator, jolloin DSL Toolsin sovelluskehys osaa käsitellä sen koodigeneraattorina.

Koodigeneraattori toteutetaan joko C#:lla tai VB.NET:illä. Valitun kielen ilmaisuvoima on kokonaan käytettävissä, ja se on kummassakin tapauksessa käytännössä yhtä laaja. Peruskielen lisäksi käytössä on muutamia avainmerkkiyhdistelmiä, joilla voidaan merkitä osia generaattoreista omiksi ohjelmakoodilohkoikseen. Kaikki teksti, jota ei ole erikseen merkitty ohjelmakoodiksi, tuloste-

taan tuloskoodiin sellaisenaan. Poikkeavaa käsittelyä ilmaisevat merkkiyhdistelmät on lueteltu ja kuvattu taulukossa 1

TAULUKKO 1: DSL Toolsin koodigeneraattorissa käytettävät ohjausmerkit (Cook ym.. 2007, 329-332)

Lohkon nimi	**Erotinmerkit**	**Selite**
Direktiivilohko (*directive*)	<#@ … #>	Direktiiveillä ohjataan DSL Toolsin generaattorikäsittelijän toimintaa. Esimerkiksi käytettävä ohjelmointikieli ja tuloskoodin tiedostopääte määritellään direktiiveillä.
Kontrollilohko (*control block*)	<# … #>	Kontrollilohkon teksti tulkitaan generaattorin ohjelmakoodiksi. Tämä koodi suoritetaan sellaisenaan. Kontrollilohkoa käytetään ohjaamaan generaattorin toimintaa, ja tyypillisimmillään kyse on silmukasta, jolla käydään läpi mallin elementtejä.
Lausekelohko (*expression block*)	<#= … #>	Lausekelohkon sisältämä teksti tulkitaan ohjelmakoodilla esitetyksi lausekkeeksi, joka evaluoidaan. Evaluoinnin tulos kirjoitetaan merkkijonona tuloskoodiin. Voidaan käyttää esimerkiksi luotaessa mallin elementtejä vastaavia olioita oliomuuttujien nimien tulostukseen.
Luokkaominaisuuslohko (*class feature block*)	<#+ … #>	Luokkaominaisuuslohkon sisältö tulkitaan ohjelmakoodiksi, joka laajentaa ModelingTextTemplate-luokkaa. Tällä tavoin voidaan määritellä generaattorille omia metodeja ja ominaisuuksia, aivan kuten valitussa ohjelmointikielessä yleensäkin.

Kun malli on valmis ja siitä halutaan generoida koodia, toimitaan kuten metamallin luonnin yhteydessä, eli painetaan muuntonappulaa, joka käynnistää muunnokset. Kukin malliin määritelty generaattori ajetaan, minkä jälkeen koodi on valmista. Tästä eteenpäin sovelluksen kehittäminen on sovelluskehyksen ja generoidun koodin yhteensovittamista, mikä ei enää ole DSL Toolsin sovellusaluetta.

3.7 Huomioita DSL Toolsista

DSL Toolsin parhaimpiin puoliin kuuluu mahdollisuus käyttää yleiskäyttöistä ohjelmointikieltä koodigeneraattoreiden luonnissa. Tällainen näkemys voi tun-

tua hieman omituiselta, kun huomioidaan tämän tutkielman aihe ja aiempi argumentointi yleiskäyttöisten kielten heikkouksista suppean ja rajatun sovellusalueen kuvaamisessa – sitähän automaattinen ohjelmakoodin generointi kiistatta on. Kelly ja Tolvanen (2008, 274) mainitsevatkin, että mikäli generaattorien luontiin on olemassa oma kielensä, voidaan siihen toteuttaa esimerkiksi mallin läpikäyntiä varten omia toimintojaan. Lisäksi he katsovat yleiskäyttöisen kielen käytön generaattorien luonnissa aiheuttavan ongelmia generaattorin koodin sisäistämisessä, varsinkin jos generaattorin tuottama koodi on samalla kielellä toteutettua. Tämä johtuu siitä, että koodin eri osien hahmottaminen on vaikeaa ja siitä voi seurata sekaannuksia. (Kelly & Tolvanen 2008, 380) Tämä ongelma tuli esille myös tämän työn käytännön osuudessa. Generaattorin ollessa yksinkertainen ei ongelmaa juuri ilmene, mutta kun generaattorin koodi monimutkaistuu, alkaa se muuttua nopeasti epäselvemmäksi.

Tästä huolimatta yleiskäyttöisen kielen käyttö generaattoreiden luomiseksi tuo myös eräitä etuja. Generaattoreiden kehittäminen kuuluu ohjelmistokehityksen ammattilaisen työtehtäviin eikä siksi vaadi käytettävältä työkalulta erityistä helppokäyttöisyyttä. Yleiskäyttöinen ohjelmointikieli, eli DSL Toolsin tapauksessa joko C# tai VB.NET, tarjoaa myös riittävät keinot esimerkiksi merkkijonojen manipulointiin. Koska koodin generointi on lopulta merkkijonojen tulostamista tekstitiedostoon, on merkkitiedon käsittely tärkeä osa generaattorin toimintaa. Tähän voidaan käyttää esimerkiksi .NET:in sovelluskehykseen toteutettua StringBuilder-luokkaa, joka tarjoaa tehokkaan tavan käsitellä pitkiäkin merkkijonoja, vaikkapa kokonaisia luokan metodeja. Myös tietorakenteiden käsittelyyn on valmiit keinot, mikä helpottaa mallin läpikäyntiä.

DSL Toolsin vahvuus on myös mahdollisuus luoda räätälöityjä toimintoja metamallin luonnin yhteydessä. Esimerkiksi kohdassa 3.5 esiteltyjen oikeellisuustarkistusten määrittelyssä tämä on erittäin käyttökelpoinen ominaisuus. Samoin mahdollisuus luoda itse logiikka elementtien jäsenmuuttujien käsittelyyn tuo vapautta metamallin suunnitteluun.

DSL Toolsin tätä tutkielmaa tehtäessä käytössä olleesta versiosta 2005 näkee kuitenkin selvästi sen olevan vielä kehitysvaiheessa. Käyttöliittymä on joissain tapauksissa kovin kankean oloinen. Esimerkiksi metamallin luokkien siirtely ei onnistu hiirellä raahaamalla, vaan valitsemalla hiiren oikealla napilla haluttu luokka ja valitsemalla ponnahdusvalikosta ”Siirrä ylös” tai ”Siirrä alas”, jolloin luokka siirtyy yhden askeleen haluttuun suuntaan. Tätä ei voida pitää missään nimessä käyttäjäystävällisenä ratkaisuna, varsinkaan jos luokkia on paljon. Syynä tähän ratkaisuun lienee se, että tällä tavoin voidaan pitää metamallin eri elementit helpommin loogisessa järjestyksessä ja sen ulkonäkö selkeänä. Sen sijaan on huomattavasti vaikeampi ymmärtää, miksi sama tilanne on myös työkalulaatikon järjestelyssä. Vaikka tässä työssä tuotetun kielen työkalujen määrä on suhteellisen vähäinen, ei niiden järjestelyyn esimerkiksi useampaan lokeroon riittänyt intoa enää ensimmäisen lokeron luonnin jälkeen. Tässä lisähankaluutena oli vielä se, että työkalua ei voi itse asiassa siirtää lokerosta toiseen, vaan se on poistettava kokonaan ja luotava uudelleen haluttuun lokeroon. Molemmat ongelmat voi kiertää editoimalla suoraan metamallin sisältävää XML-tiedostoa, mutta tällainen on lähinnä hätävararatkaisu toimivampaa versiota odoteltaessa.

Koodigeneraattorin luonnissa käytettävä työkalu on pelkkä tekstieditori. Editori ei tue mm. Visual Studiosta tuttua automaattista koodin täydennystä, eli intellisense-toimintoa. Tämä ei vielä ole suurikaan puute, vaikkakin nykyaikaisissa ohjelmointiympäristöissä toiminto on yleensä mukana. Hämmästyttävää, ja huomattavasti häiritsevämpää, sen sijaan on se, että työkalussa ei ole minkäänlaista tukea koodin havainnollistamiseen, mikä aiheuttaa koodin muuttumista nopeasti erittäin vaikeaksi luettavaksi, mistä toimii esimerkkinä liitteessä 2 esitetty esimerkkigeneraattori. Esimerkiksi kontrollimerkkien sisällä olevan tekstin erottamista generoituun koodin suoraan tulostettavasta tekstistä ei helpoteta millään lailla, vaikka tämän luulisi olevan yksinkertaisimmasta ja helpoimmasta päästä kokonaisen mallinnustyökalun kehitystehtäviä. Kyse on kuitenkin

vain tietyn tekstin osan merkitsemisestä esimerkiksi värittämällä, kontrollilohkothan joudutaan joka tapauksessa tunnistamaan, kun generaattorin koodi käännetään. Kolmansien osapuolten kehittämiä työkaluja väritystoimintoineen on tarjolla, mutta niidenkin käyttöön otossa on omat hankaluutensa. Eräs puuttuva ominaisuus on myös generaattorikoodin ajonaikainen debug-mahdollisuus. Tämä helpottaisi oleellisesti vianetsintää, mutta sen toteuttaminen lienee jo koodin väritystä hankalampaa.

DSL Toolsin dokumentaatio on täysin keskeneräistä. MSDN:stä löytyy kyllä kaikkien työkalun sovelluskehyksen luokkien rakenteelliset kuvaukset, mutta osassa näistä ei ole mukana minkäänlaista tekstimuotoista kuvausta luokan toiminnasta. Esimerkkejäkään ei juuri ole tarjolla. Eri verkkosivuilta, blogeista ja uutisryhmistä löytyy jonkin verran sekä käyttäjien että DSL Toolsin kehittäjien opastuksia aiheeseen, mutta paras tapa tuntuu edelleen olevan mallista generoidun koodin tutkiminen sekä yrityksen ja erehdyksen metodilla eteneminen.

3.8 Yhteenveto

Tässä luvussa esiteltiin DSL Tools -työkalun tärkeimmät ominaisuudet. Luvussa pyrittiin keskittymään niihin piirteisiin, jotka ovat myöhemmin esitettävän kielen kannalta oleellisia. Työkalu tarjoaa mahdollisuuden sovellusaluekielen abstraktin ja konkreetin syntaksin määrittelyyn sekä koodigeneraattoreiden luontiin. Lisäksi työkalulla on mahdollista määritellä erilaisia rajoitteita, joiden avulla sovellusaluekielellä luodun mallin oikeellisuus voidaan tarkistaa. Yleisesti ottaen DSL Tools on eräistä suuristakin puutteistaan huolimatta käyttökelpoinen väline sovellusaluekielen toteuttamiseen ja vaikuttaisi soveltuvan myös puumaisen laskentakaavan esittämiseen. Tätä soveltuvuutta testataan seuraavassa luvussa.

4 NAV-KIELI

Tutkielman käytännön osuutena tuotetaan sovellusaluekieli, jonka avulla pystytään kuvaamaan sijoitusrahaston arvonlaskentaprosessi. Tämän kielen nimi on NAV-kieli. Kielen kehittämisen tarkoituksena on hankkia kokemuksia ja esittää arvioita sovellusaluekeskeisen lähestymistavan käytännön soveltamisesta ja tutkia sen soveltuvuutta kohteena olevalle sovellusalueelle. Luvussa kuvataan ensin sovellusalue ja sen mallintamisen tavoitteet. Sen jälkeen määritellään NAV-kielen perustana oleva metamalli ja esitetään kehitetty koodigeneraattori sekä sovelluskehys. Lopuksi kuvataan kielen ja työkalujen määrittelyprosessia sekä arvioidaan kehitettyä kieltä.

4.1 Sovellusalue ja sen mallintamisen tavoitteet

Usein yksityisellä sijoittajalla ei ole riittävästi pääomaa, jotta olisi mahdollista sijoittaa se kovinkaan hajautetusti. Hajauttamalla pyritään jakamaan sijoitettu pääoma usealle osakkeelle, toimialalle tai jopa markkina-alueelle, ja siten vähentämään kuhunkin yksittäiseen osa-alueeseen liittyvää riskiä. Tällainen hajauttaminen on kuitenkin mahdollista myös pienemmällä pääomalla, mikäli useamman piensijoittajan sijoitukset kootaan yhteen, ja tämä pääoma sijoitetaan edelleen haluttuihin arvopapereihin. Tällaista toimintaa varten on perustettu *sijoitusrahastoja*.

Sijoitusrahastoja ja niiden toimintaa säätelee *sijoitusrahastolaki (SRL)*. Se määrittelee *sijoitusrahastotoiminnan* seuraavasti (SRL 1. luku, §2):

> *Sijoitusrahastotoiminnalla* [tarkoitetaan] varojen hankkimista yleisöltä yhteistä sijoittamista varten ja näiden varojen sijoittamista pääasiallisesti rahoitusvälineisiin tai kiinteistöihin ja kiinteistöarvopapereihin sekä sijoitusrahaston hallintoa.

Sijoitusrahastolla taas tarkoitetaan sijoitusrahastolain mukaan sijoitusrahastotoiminnassa kerättyjä ja sijoitettuja varoja sekä näistä seuraavia velvoitteita, ja

rahastoyhtiöllä sijoitusrahastotoimintaa harjoittavaa yritystä (SRL 1.luku, §2). Tässä tutkielmassa käytetään sijoitusrahastosta myös termiä *rahasto* lyhyyden vuoksi. Rahaston sijoituskohteet eivät ole kovinkaan merkityksellisiä tutkielman tavoitteena olevan sovellusaluekielen osalta, mutta esimerkkien ja käytännön yhtymäkohtien kannalta on hyvä mainita, että tässä työssä rahaston sijoituskohteina ajatellaan olevan lähinnä erilaiset arvopaperit ja rahamarkkinainstrumentit, kuten osakkeet, korkosopimukset ja joukkovelkakirjalainat.

Jotta rahastoilla voitaisiin käydä kauppaa ja niiden tuottoa vertailla, pitää niiden arvo laskea. Tätä laskentaprosessia kutsutaan *arvonlaskennaksi*. Rahaston arvo lasketaan vähentämällä rahaston varoista sitä koskevat velat. Yksittäisen rahasto-osuuden arvo saadaan tästä rahaston kokonaisarvosta jakamalla se liikkeessä olevien osuuksien lukumäärällä. (SRL 8.luku, §48) Tätä yksittäisen osuuden arvoa käytetään rahasto-osuuden merkintä- ja lunastushintana. Sijoitusrahastolaki ei määritä, miten rahaston varat ja sitä koskevat velat tarkkaan ottaen lasketaan. Tästä syystä Rahoitustarkastus on julkaissut kannanoton, joka valottaa laskentaperusteita hieman tarkemmin (Rahoitustarkastus 2007). Tämäkin kannanotto antaa ohjeita lähinnä suurista linjoista, jättäen suuren osan laskentaperusteista rahastoyhtiön itsensä päätettäväksi.

Rahastolla on oltava säännöt, jotka määräävät mm. rahaston sijoituspolitiikan, arvonlaskennassa käytettävien kurssien valintaperusteet ja arvonlaskennan kellonajan. Rahastolla voi olla useita arvonlaskenta-aikoja, joista kuitenkin vain yksi on virallinen, rahaston arvon määrittävä laskenta-aika. Muut laskenta-ajat voivat olla esimerkiksi vertailuarvoja, joiden avulla voidaan vertailla keskenään rahastoja, joiden viralliset arvot lasketaan eri aikaan. Näissä säännöissä tulee myös määritellä muita rahaston arvoon vaikuttavia tekijöitä, mm. arvopaperiomistusten arvostusperusteet, mikäli kurssitietoja ei jostain syystä saada tietyltä laskentapäivältä. (Rahoitustarkastus 2007)

Ohjelmistokehityksen näkökulmasta katsottuna arvonlaskennan perustana olevien rahaston omistusten ja velkojen tiedot pitää lukea arvonlaskennasta huolehtivaan sovellukseen ulkoisista järjestelmistä. Tässä tutkielmassa käsiteltävän, arvopapereihin sijoittavan rahaston osalta nämä tiedot löytyvät pääosin rahaston osuusrekisteristä ja sen salkunhallintajärjestelmästä. Osuusrekisterissä ylläpidetään tietoa siitä, kuka omistaa rahasto-osuuksia. Näiden omistuksien osalta järjestelmään tallennetaan mm. omistusten määrä, niiden hankinta-ajat sekä hankinta-arvot. Salkunhallintajärjestelmässä taas pidetään yllä tietoja rahaston arvopaperiomistuksista. Usein näihin järjestelmiin lähetetään myös arvonlaskennan pohjalta palautetietoja, esimerkiksi osuusrekisteriin viedään rahasto-osuuden arvo.

Rahaston arvonlaskenta on erittäin tärkeä osa sijoitusrahaston toimintaa. Tästä syystä sen suorittamiseen käytettävän sovelluksen täytyy toimia luotettavasti. Vaikka Rahoitustarkastus painottaa kannanotossaan (2007) arvonlaskennan tuloksen tarkistamisen tärkeyttä, ei tämä tietenkään vähennä laskennasta vastaavaan sovellukseen kohdistuvia luotettavuusvaatimuksia. Luotettavuuden lisäksi vaatimuksena on se, että sovelluksen käytön olisi syytä olla mahdollisimman automatisoitua, koska laskentaprosessin läpivienti on päivittäinen toimenpide. Mikäli prosessi siis on kömpelö ja vie tarpeettomasti aikaa, vaikuttaa se haitallisesti myös rahastoyhtiön toiminnan tehokkuuteen ja siten sen kulurakenteeseen.

Sovellusaluekielen kehittämisen kannalta sovellusalue ei ehkä ole kaikkein otollisin, koska täysin standardoitua laskentaprosessia ei ole olemassa. Tästä syystä sovellusalueen käsitteistössä voi olla vaihtelua eri rahastojen välillä. Kaksi kaikissa rahastoissa mukana olevaa toiminnallista kokonaisuutta voidaan kuitenkin löytää: laskentakaava ja syötetietojen käsittely. Näiden toimintojen mallintamisen helpottaminen on jo askel kohti helpompaa ja tehokkaampaa arvonlaskentasovelluksen luontia ja ylläpitoa.

Kehitettävälle sovellusaluekielelle on asetettu seuraavat tavoitteet. Kielellä tulee pystyä esittämään arvonlaskennan laskentakaava sekä kaavassa käytettävien arvojen tietolähteet. Laskentakaavan esittämistä varten tulee pystyä kuvaamaan peruslaskutoimitukset (yhteen-, vähennys-, kerto- ja jakolasku), keskeisimmät koostefunktiot (summa ja keskiarvo) sekä laskujärjestys.

4.2 Metamalli

Kehitettävän mallinnuskielen nimi on NAV-kieli, joka on lyhenne sanoista Net Asset Value, suomennettuna arvonlaskentakieli. Kielen peruselementtejä ovat tietolähteitä, näistä saatavia numeroarvoja ja näiden arvojen välisiä suhteita kuvaavat luokat. Suhteet kuvaavat joko arvon hakua tietolähteestä tai sen käyttöä osana laskuoperaatiota. Elementtien ja niiden välisten suhteiden joukko muodostaa NAV-kielen metamallin, ja ne on kuvattu seuraavissa alakohdissa. Malli jakautuu kahteen osaan, tietolähteisiin ja laskentapuuhun. Nämä osat on jaettu suunnittelunäytöllä omille swimlane-osioilleen. Metamalli on esitetty kuvana liitteessä 1.

4.2.1 Tietolähteet ja laskentapuu

Laskentaan käytettävien lähtöarvojen hakua varten tarvitaan malliin keino määrittää näiden arvojen tietolähteet. Tässä työssä lähteitä on kolmea tyyppiä: tietokantataulu, ini-muotoinen tiedosto sekä XML-tiedosto. Erilaisten tietolähteiden lisääminen kieleen on jatkossa suhteellisen yksinkertaista, koska niiden tarvitsee vain toteuttaa samanlainen hakurajapinta kuin alun perin mukana olleiden tietolähteiden, sisäisen toteutustavan ollessa täysin vapaa. On myös mahdollista ottaa käyttöön uusia tietolähteitä siten, että rakennetaan erillinen sovellus, joka vie tiedot uudesta formaatista esimerkiksi tietokantaan, jonka käsittelyä varten kielessä on jo olemassa keinot.

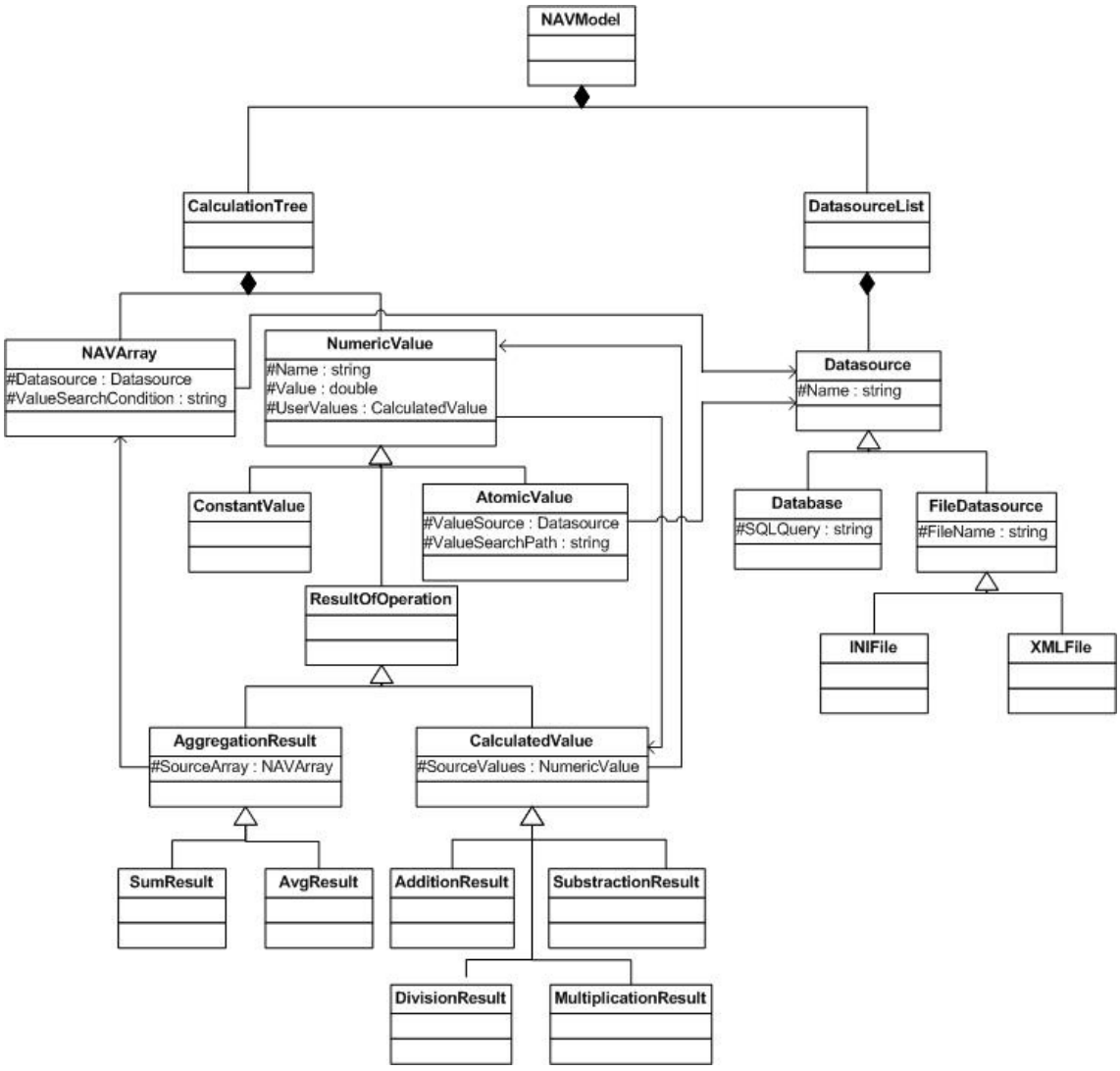
Tietolähteelle voidaan määritellä vain yksi ominaisuus: tietolähteen sijainti. Tiedostomuotoisessa tietolähteessä sijaintimääritteenä toimii hakemistopolku tiedostonimineen, tietokantalähteessä taas SQL-lause, jolla tulosjoukko haetaan. Muita tietoja ei suoraan tietolähteelle määritetä. Tietolähteestä haetaan tietoja yhdistämällä laskentapuun elementti tietolähteeseen erityisellä *tietolähteen yhdistimellä*. Tietolähteeseen voidaan yhdistää kahdenlaisia elementtejä: atomisia arvoja ja taulukoita.

Laskentapuu sisältää arvonlaskennan laskentalogiikan. Laskentalogiikan esittämiseksi puu sisältää joukon elementtejä ja yhdistimiä, joilla kuvataan erilaiset laskutoimitukset. Tästä puusta valitaan juurisolmuiksi mallin muunnosvaiheessa sellaiset arvot, jotka ei käytetä missään laskutoimituksessa. On syytä huomata, että kielen laskentapuu ei tarkkaan ottaen ole puurakenne, koska jokin arvo voi olla kahden eri välisumman tekijänä, ja nämä välisummat taas jonkin muun arvon tekijöinä. Kahden solmuna toimivan arvon välillä voi siis olla useita eri reittejä, mikä ei puurakenteessa ole sallittua. Ennemminkin kyse on suunnatusta verkosta, jossa ei voi olla silmukoita, eli solmusta ei voi palata mitään reittiä takaisin samaan solmuun, mutta solmusta toiseen voi olla useita reittejä. (ks. tarkemmat määritelmät esim. Savolainen 2001) Nimi kuitenkin kuvaa yleiskielen merkityksessään hyvin laskennan rakennetta, joka ikään kuin haaroittuu juurina olevista tulosarvoista kohti lehtisolmuina olevia lähtöarvoja.

4.2.2 Elementit

Mallinnuskieli sisältää muutamia elementtejä, joita käytetään kuvaamaan joko tietolähdettä tai arvoa. Elementit muodostavat luokkahierarkian, joka on esitetty kuviossa 6. Metamallin abstrakti syntaksi sisältää kaksi abstraktia perusluokkaa NumericValue ja Datasource, joista periytyvät varsinaiset konkreetit luokat. Lisäksi mallissa on mukana yksi konkreetti luokka NAVArray, joka ei periydy kummastakaan edellä mainitusta perusluokasta.

NumericValue on abstrakti yläluokka numeerisen tiedon esittämiseksi. Luokka sisältää eräitä jäsenmuuttujia, joista tärkeimmät ovat merkkijonomuuttuja *Name*, jossa on kulloisenkin ilmentymän edustaman arvon nimi, sekä double-tyyppinen jäsenmuuttuja *Value.* Tässä muuttujassa säilytetään luokan ilmentymän edustamaa numeroarvoa. Lisäksi luokalla on yksi viitemuuttujataulukko *UserValues*, joka sisältää viittauksen kaikkiin niihin arvoihin, jotka kyseistä NumericValue-oliota arvonsa laskemiseen käyttävät.



KUVIO 6. NAV-kielen abstrakti syntaksi luokkakaaviona. Kuvassa on luokkina esitetty vain elementtiluokat, suhdeluokat on kuvattu nuolilla.

NumericValue-luokasta periytyvät seuraavat luokat:

- *ConstantValue*: Luokka edustaa yksittäistä vakioarvoa, joka määritellään mallissa. Kyse voi olla esimerkiksi kiinteästä kertoimesta.

- *AtomicValue*: Luokka esittää arvoa, joka on atominen, eli sitä ei lasketa mallin perusteella. Arvo on siis mallin kannalta vakio, mutta poikkeaa ConstantValue-luokasta siten, että se saadaan jostain ulkoisesta lähteestä. Tällainen ulkoinen lähde voi olla esimerkiksi käyttöliittymän tekstikenttä tai XML-tiedoston tietty solmu. Luokalla on jäsenmuuttuja *ValueSource*, joka sisältää viitteen tietolähteeseen, josta luokan esittämä arvo haetaan. Jäsenmuuttuja *ValueSourcePath* taas sisältää merkkijonon, jonka perusteella mikä ValueSource-tietolähteen sisältämistä arvoista halutaan hakea.

- *ResultOfOperation*: Tämä abstrakti luokka kuvaa numeroarvoa, joka saadaan laskutoimituksen tuloksena. Laskutoimitus perustuu muihin NumericValue-luokan ilmentymiin tai koostefunktion tulokseen.

ResultOfOperation-luokasta periytyvät seuraavat luokat:

- *CalculatedValue*: Abstrakti luokka, joka kuvaa peruslaskutoimituksen tuloksena saatua arvoa. Luokasta periytyy neljä luokkaa, jotka vastaavat kutakin nelilaskimen laskutoimitusta. Luokka sisältää viitemuuttujan *SourceValues*, joka viittaa niihin NumericValue-luokan ilmentymiin, joiden arvojen perusteella laskutoimituksen tulos on laskettu. CalculatedValue-luokasta periytyvät seuraavat konkreetit luokat:

    - *AdditionResult*: Yhteenlasku

    - *SubstractionReult*: Vähennyslasku

    - *MultiplicationResult*: Kertolasku

    - *DivisionResult*: Jakolasku

- *AggregationResult*: Abstrakti luokka, joka kuvaa arvojoukkoon kohdistuvan koostefunktion tulosta. Luokalla on viitemuuttuja *SourceArray*, joka viittaa siihen taulukkoon, jonka sisältämän arvojoukon funktio saa syötteenä. Luokasta periytyvät seuraavat konkreetit luokat:
    - *SumResult*: Summa
    - *AvgResult*: Keskiarvo

Lisäksi mallissa on määriteltynä luokka *NAVArray*. Luokka kuvaa kokoelmaa, joka sisältää listan arvoja. NAVArray-oliolla on viitemuuttuja *Datasource*, joka viittaa tietolähteeseen, josta arvot haetaan. Lisäksi oliolla on merkkijonomuuttuja *ValueSearchCondition*, joka sisältää taulukkoon mukaan otettavien arvojen hakuehdon. Tämä hakuehto muodostetaan sen mukaan, minkälainen tietolähde on. XML-tiedostosta haettaessa hakuehto on arvo, joka voidaan antaa sellaisenaan .NET-kehyksen System.Xml.XmlDocument-olion SelectNodes-metodille. INI-tiedostoon tai tietokantakyselyn tulosjoukkoon kohdistuva hakuehto on merkkijono, joka täsmää joko INI-tiedoston lohkon tai yksittäisen tiedostosta tai kyselystä löytyvän arvon nimeen. Hakuehdossa voi olla alussa ja lopussa jokerimerkkinä tähti (*), joka mahdollistaa osittaisten osumien mukaan oton taulukkoon. NAVArray-olio siis viittaa aina ulkoiseen tietolähteeseen, josta se hakee kaikki ne arvot, jotka osuvat hakuehtoon.

Erilaisia tietolähteitä on kolme: INI-tiedosto (*INIFile*), XML-tiedosto (*XMLFile*) ja tietokantakysely (*Database*). Näiden kautta laskentapuu saa lähtöarvonsa. Tietolähteeseen voidaan yhdistää joko taulukko tai yksittäinen atominen arvo. Tietolähteillä on aina jäsenmuuttuja *Name* (tietolähteen nimi). Lisäksi tietokantakyselyllä on jäsenmuuttuja *Query*, joka sisältää käytettävän SQL-lauseen. Tiedostomuotoisilla tietolähteillä taas on jäsenmuuttuja *FileName*, joka kertoo tiedostonimen. Tietolähteet on kuvattu taulukossa 2.

Kuhunkin edellä kuvattuun abstraktin syntaksin konkreettiin luokkaan liittyy konkreetin syntaksin vastaava luokka. ResultOfOperation-luokasta periytyvien luokkien kohdalla käytetään yhteistä konkreetin syntaksin luokkaa, ja näiden ero ilmenee suoritettavaa laskutoimitusta kuvaavasta symbolista. Kunkin konkreetin syntaksin luokan nimi muodostuu vastaavan abstraktin syntaksin luokan nimestä siten, että nimen loppuun lisätään tarkenne Shape, eli esimerkiksi CalculatedValueShape.

TAULUKKO 2: Tietolähteet.

Elementti	**Selite**
Database	Tietokannasta saatava tietuejoukko. SQL-kysely muodostaa hakupolun.
XMLFile	XML-muotoinen tiedosto. Hakupolku määritetään XML-standardin mukaisesti.
INIFile	INI-muotoinen tiedosto, vastaa esimerkiksi WIN.INI-tiedoston rakennetta. Hakupolku määritetään muodossa ”LOHKO ARVO”. Esimerkiksi hakupolku ”PORTFOLIO VALUE” löytäisi seuraavan arvon: [PORTFOLIO] VALUE=13000

4.2.3 Yhdistimet

Kieleen kuuluu elementtiluokkien lisäksi myös joitakin suhdeluokkia. Osa suhteista on olemassa metamallin rakenteen vuoksi, eikä niillä ole semanttista merkitystä. Esimerkkeinä näistä suhdetyypeistä toimivat NAVModelHasCalculationTree ja NAVModelHasDatasourceList (ks. Liite 1). Suhteet siis ainoastaan määrittävät CalculationTree- ja DatasourceList -luokat kuuluviksi NAVModel-mallia esittävään juuriluokkaan (vrt. kohta 3.3). Samalla tavoin näihin mallin

osiin kuuluvat luokat on yhdistetty niihin koostesuhteella. Kun nämä suhdetyypit on määritelty, mallin kehitysvaiheessa mallin juuriluokasta alkaen luodaan vastaavat viitemuuttujat, joiden muodostamaa hierarkiapolkua seuraamalla päästään käsiksi malliin määriteltyihin objekteihin. Näille suhdetyypeille ei ole vastaavaa konkreetin syntaksin luokkaa, koska ne ovat automaattisesti mallin piirtovaiheessa syntyviä, eivätkä ole varsinaisen mallinnustyön kannalta oleellisia.

Varsinaisia mallintamisessa käytettäviä suhdetyyppejä on neljä (ks. Liite 1):

- AtomicValueReferencesDatasource, jolla yhdistetään tietolähteestä saatava atominen arvo tietolähteeseensä. Konkreetin syntaksin vastine on AtomicValueSourceConnector.

- CalculatedValueHasSourceValues, jolla yhdistetään laskentaoperaatiosta saatava arvo lähtöarvoihinsa. Konkreetin syntaksin vastine on CalculationSourceValueConnector.

- AggregationResultReferencesNAVArray, jolla yhdistetään koostefunktion tulos syötearvonsa sisältävään taulukkoon. Konkreetissa syntaksissa AggregationSourceConnector.

- NAVArrayReferencesDatasource, jolla yhdistetään taulukko tietolähteeseensä. Konkreetissa syntaksissa ArraySourceConnector

CalculationSourceValueConnector poikkeaa hieman muista yhdistimistä. Tällä yhdistimellä, tai oikeammin sen kuvaamalla abstraktin syntaksin luokalla on määriteltynä laskujärjestyksen määräävä järjestysnumero *CalculationOrderNumber*. Tämä numero on merkityksellinen vähennys- ja jakolaskuissa. Numerointi toimii oletusarvoisesti yhteyksien luontijärjestyksessä, mutta käyttäjä voi kasvattaa numeroa napsauttamalla hiirellä yhdistintä kahdesti. Tällöin kyseinen yhdistin vaihtaa järjestysnumeroa lähinnä suuremman yhdistimen kanssa, jolloin näiden keskinäinen järjestys siis vaihtuu päinvastaiseksi.

4.3 Generaattori

NAV-kielessä on vain yksi generaattori, CalculationTreeCreation, joka vastaa laskentalogiikkaa esittävän puurakenteen sekä tietolähteiden luomisesta. Generaattori on toteutettu VB.NET:illä, ja sen ohjelmakoodi on esitetty liitteessä 2. Generoitu koodi on myös VB.NET:iä, ja esimerkki yksinkertaisesta laskentapuusta tietolähteineen on esitetty liitteessä 3.

Generaattori toimii siten, että se käy läpi kaikki mallissa olevat elementit tuottaen niitä vastaavien olioiden luontikomennot. Nämä muuttujien esittelyt kirjoitetaan tulostiedostoon läpikäynnin aikana. Samassa yhteydessä rakennetaan StringBuilder-luokan avulla alustusmetodin ohjelmakoodi. Tämä metodi huolehtii olioiden välisten suhteiden luonnista ja se tulostetaan tiedostoon läpikäynnin jälkeen StringBuilder-oliosta. Tuloksena saatava metodi alustaa laskentaa kuvaavan tietorakenteen, joka on kahteen suuntaan linkitetty, suunnattu verkko.

Generaattorin tuottama koodi hoitaa vain mallin mukaisen laskentarakenteen, arvojen saantitaulukoiden sekä tietolähteiden luomisen. Muut ohjelmakoodin osat on toteutettu sovelluskehykseen, jolloin muuttuvan koodin määrä on pystytty pitämään mahdollisimman pienenä.

4.4 Sovelluskehys

Sovelluskehys sisältää ne sovelluksen komponentit, jotka pysyvät muuttumattomina, eikä niitä siten tarvitse generoida mallin pohjalta. Tässä työssä esiteltävässä kielessä kehykseen kuuluu pääosa varsinaisesta ohjelmalogiikasta, vain laskentapuun muodostamisen jäädessä koodigeneraattorin huoleksi.

Sovelluskehykseen on toteutettu pääosin samat luokat kuin NAV-kielen metamallin abstraktiin syntaksiinkin lukuun ottamatta mallia vastaavaa NAVModel-luokkaa sekä laskentapuun ja tietolähdelistan luokkia. Kehyksessä oleviin

luokkiin on myös toteutettu laskennan suorittamisen kannalta tarpeelliset metodit ja jäsenmuuttujat. Laskutoimitusten osalta sovelluskehyksessä luodaan erilliset laskuoperaatio-oliot kutakin CalculatedValue-oliota kohden. Tällä tavoin pystytään paremmin kapseloimaan laskentalogiikka omaan luokkaansa.

Sovelluskehyksessä on kaksi abstraktia yläluokkaa, joista muut laskentapuun luokat periytyvät. Yläluokat määrittävät muutaman rajapintametodin, joiden kautta voidaan laskenta suorittaa sekä käynnistää tietyn arvon muuttuessa siitä riippuvien arvojen päivitys. Yläluokat ovat NumericValue ja NAVOperator. Luokkien ylikirjoitettavat metodit on esitetty taulukossa 3. Taulukossa esitetään metodien nimet sekä niiden parametrit ja paluuarvot.

Myös tietolähteitä varten on määritelty muutamia rajapintamäärityksiä ja luokkia. Rajapintamääritysten tarkoituksena on määritellä tietyt toiminnot, jotka kaikkien ko. rajapintaa vastaavien tietolähteiden on toteutettava. Tässä vaiheessa rajapintoja on kaksi. Ensimmäinen näistä on *IDatasource*, jolla määritellään yleinen tietolähteen kuvaus ja joka sisältää vain hakumetodit. Toinen on *IFileDatasource*, joka määrittää tiedostomuotoisen tietolähteen käsittelymetodit.

Rajapintojen lisäksi on toteutettu kaksi luokkaa, jotka tarjoavat keinot tietolähteen käsittelyyn. Luokat ovat *ValueArray* ja *FileDatasource*. ValueArray käsittelee taulukkomuotoista tietoa ja toteuttaa IDatasource-rajapinnan. FileDatasource taas toteuttaa IFileDatasource-rajapinnan ja osaa lukea tiedoston sisällön omaan taulukkoonsa. Näiden rajapintojen ja luokkien metodit on esitelty taulukossa 4. Varsinaiset tietolähdettä vastaavat luokat käyttävät näiden luokkien palveluja. Esimerkiksi *INIFile*-luokka, joka käsittelee INI-tiedostomuotoista lähdeaineistoa, toteuttaa IFileDatasource ja IDatasource -rajapinnat ja käyttää sisäisesti koostesuhteella FileDatasource ja ValueArray -olioita näiden rajapintojen takana olevan logiikan toteutuksessa.

TAULUKKO 3: NAV-kielen sovelluskehyksen laskentapuun abstraktien luokkien metodit.

Metodi	**Selite**
NumericValue-luokka	
New(name as String)	Konstruktori, jolle on annettava pakollisena parametrina olion esittämän arvon nimi.
AddSourceOperator(opt as NAVOperator)	Metodilla lisätään viite olion esittämää arvoa käyttävään operaattoriin.
Recalculate()	Metodi kutsuu arvoa käyttävien operaattoreiden uudelleenlaskentametodia.
NAVOperator-luokka	
New()	Konstruktori
Calculate()	Metodin kutsu laukaisee operaattorin tulosarvon laskennan.
ToString() as String	Metodi palauttaa merkkijonomuuttujaan operaattorin toiminnallisuutta kuvaavan symbolin, esimerkiksi siis plus-merkin.
SignalSourceValueChanged()	Metodia käytetään laukaisemaan operaattorin arvon uudelleen laskenta jonkin lähtöarvon vaihtuessa.
AddSourceValue(value as NumericValue)	Metodi lisää operaattorin käyttämän arvon taulukkoon ja kutsuu kyseisen NumericValue-olion AddUserOperator()-metodia antaen parametrina viitteen operaattoriin.

Laskentapuun luominen tapahtuu generaattorin luomalla metodilla (vrt. kohta 4.3.) ja sovelluskehyksen tehtävänä on vain kutsua tätä luontimetodia. Lisäksi sovelluskehys huolehtii myös laskentapuun tallentamisesta tietokantaan. Tätä toiminnallisuutta ei kuitenkaan ole vielä toteutettu, vaan sen kehittäminen jää myöhemmäksi. Toteutus on perustoiminnaltaan triviaali, mutta siinä on otettava huomioon mahdolliset rahaston elinkaaren aikana sen laskentalogiikkaan ja siten myös malliin tehtävät muutokset.

TAULUKKO 4. NAV-kielen sovelluskehyksen tietolähteiden määritykseen liittyvien abstraktien luokkien sekä rajapintamääritysten metodit.

Metodi	**Selite**
IDatasource-rajapinta	
Find (itemName as string) as Decimal	Hakee yksittäisen arvon ko. arvon nimen perusteella.
FindSeveral (searchCondition as string) as HashTable	Hakee hakuehtoon täsmäävät arvot ja palauttaa ne .NET-kehyksestä löytyvässä HashTable-kokoelmassa.
IFileDatasource-rajapinta	
Load()	Lataa tietolähteen sisällön tietorakenteeseen.
Property FileName() as String	Tiedostonimi
FileDatasource-luokka	
Load()	Lataa tietolähteen sisällön tietorakenteeseen
Readonly Property IsLoaded() as Boolean	Kertoo onko tiedosto ladattu
Readonly Property Rows() as ArrayList	Tiedoston sisältö riveittäin
Property FileName() as String	Tiedoston nimi
ValueArray-luokka	
Find (itemName as string) as Decimal	Hakee yksittäisen arvon ko. arvon nimen perusteella.
FindSeveral (searchCondition as string) as HashTable	Hakee hakuehtoon täsmäävät arvot ja palauttaa ne .NET-kehyksestä löytyvässä HashTable-kokoelmassa.
Clear()	Tyhjentää arvot sisältävän taulukon
Add(key as Object, value as Object)	Lisää uuden arvon taulukkoon.

Kieleen on lisäksi toteutettu demonstraatiotarkoituksessa näyttö, joka luodaan laskentapuun perusteella. Tämä näyttö on kuitenkin tarkoitettu vain keinoksi esitellä sovellusaluekielen, koodigeneraattorin sekä sovelluskehyksen toimintaa ja se on siten tutkielman varsinaisen tavoitteen kannalta toisarvoinen. Kuva täs-

tä näytöstä on esitetty liitteessä 5. Liitteessä esitetty näyttö on generoitu liitteessä 2 esitetyn mallin pohjalta siten, että liitteen 3 koodigeneraattori on tuottanut liitteessä 4 esitetyn ohjelmakoodin. Näytön luontilogiikka käy läpi tämän tietorakenteen, ja muodostaa siitä yksinkertaisen näytön, jolla vain esitetään mallin sisältämät arvot.

4.5 Kielen määrittely ja työkalujen rakentaminen

Kielen ensimmäinen määrittelyvaihe oli kandidaatintutkielman osana toteutettu, nelilaskimen toiminnot sisältänyt testikieli. Ensimmäisessä vaiheessa toteutettavaksi valittu kielen osa rajattiin siten, että tutkielma ei kasvanut liian laajaksi. Rajauksesta huolimatta pyrittiin siihen, että tämän vaiheen tuloksia voitaisiin käyttää mahdollisimman suurelta osin myöhemmissäkin vaiheissa. Nelilaskimen toiminnot katsottiin riittävän yksinkertaisiksi toteuttaa. Nämä toiminnot ja niistä koostuva laskentapuu tarjosivat kuitenkin riittävästi sekä vaihtelevuutta että toisteisuutta, jotta oli mielekästä toteuttaa sovellusaluekeskeisen ohjelmistokehityksen kaikki osat, eli sovellusaluekieli, koodigeneraattori ja sovelluskehys. Toteutettavaksi valitut osat tarjosivat myös hyvät mahdollisuudet DSL Tools -työkaluun tutustumiseen.

Kieltä lähdettiin kehittämään pienin askelein ja iteratiivisesti. Kehitys alkoi laskentakaavan esittämiseen käytetystä kielen osasta, eli metamallista ja sen graafisesta esityksestä. Aluksi toteutettiin vain mahdollisuus piirtää suhde kahden eri elementin välille. Metamallin perusteella piirretyn mallin pohjalta luotiin generaattori, joka tulosti vain elementtien nimet suhteiden hierarkian määräämässä järjestyksessä tiedostoon. Tästä jatkettiin luomalla sovelluskehykseen vastaavat luokat, joiden nimimuuttujaan nimet tallennettiin ja joka osasi tulostaa nimet näytölle.

Kun ensimmäinen testiversio oli saatu onnistuneesti toimimaan, lähdettiin laajentamaan kutakin kielen osa-aluetta kohti nelilaskimen toimintoja. Tässä vaiheessa metamalliin luotiin erilliset luokat numeroarvoille ja laskuoperaatioille,

jolloin mallin esityksen toteutus oli helppoa. Tällä tavoin käytettävien elementtien määrä kasvoi melko suureksi, mutta niiden hallinta oli yksinkertaista. Samoin mallin läpikäyntiin tarvittava koodigeneraattori oli suhteellisen yksinkertainen rakentaa, koska kukin mallin elementti vastasi vain joko arvoa tai laskuoperaatiota. Sovelluskehykseen tehtiin täysin vastaava luokkarakenne, mikä helpotti myös osaltaan koodigeneraattorin rakentamista. Generaattori pystyi luomaan vastaavan olion jokaisesta mallista löytämästään elementistä. Generaattorin kehittämisessä haastavinta oli laskentapuun luomista varten tarvittu viitteiden asettaminen arvojen ja operaatioiden välille. Tämäkin oli kuitenkin suhteellisen yksinkertainen operaatio, koska metamallin suhteiden avulla oli helppoa luoda vastaavat suhteet myös generoituun koodiin.

Vaikka ensimmäisen kehitysvaiheen tuloksena syntynyt kieli oli sinällään toimiva, oli se kuitenkin hyvin yksinkertainen ja melko työläs käyttää. Yhden laskutoimituksen kuvaamiseksi käyttäjän piti luoda kolme eri arvoelementtiä (tulosarvo sekä kaksi lähtöarvoa), laskuoperaatiota kuvaava elementti sekä kolme yhdistintä laskuoperaation ja arvoelementtien välille. Tästä syystä kielen seuraavassa kehitysvaiheessa pyrittiin vähentämään tietyn toiminnon mallintamiseen tarvittavien elementtien määrää. Tähän pyrittiin poistamalla operaattorielementit mallista kokonaan ja muuttamalla CalculatedValue-elementti abstraktiksi, jolloin siitä perittiin oma luokkansa kullekin laskutoimitukselle. Laskutoimitukselle määritettiin suhde suoraan toiseen NumericValue-luokkaan. Tällä tavoin yhden laskutoimituksen kuvaamiseksi tarvittiin vain kolme arvoa kuvaavaa elementtiä sekä kaksi suhdetta kummankin lähtöarvon ja tulosarvon välille. Kehitystyön toisessa vaiheessa tuotiin metamalliin mukaan myös koostefunktiot sekä tietolähteet. Näiden avulla metamalli saatiin aiemmin tässä luvussa esitetylle tasolle.

Koodigeneraattoriin tarvittiin myös toisessa kehitysvaiheessa hieman muutoksia, vaikkakin ne olivat melko vähäisiä. Metamalliin tulleiden uusien komponenttien käsittelyn lisääminen oli melko yksinkertaista. Tietolähteiden lisäämi-

nen aiheutti generaattoriin vain niiden luontilauseen generoinnin. Suurin muutos oli operaattoreiden metamallista poistamisen aiheuttama tarve generoida CalculatedValue-olion perusteella sekä arvoa että sen luomaa operaatiota vastaava olio.

Vaikka kieli on tarkoitettu kaupallisen sovelluksen prototyypiksi, oli kehitysprosessin lähtökohta kuitenkin akateeminen. Tästä syystä kehitystyön aikana saatujen kokemuksien tarkastelu yritysmaailman prosessien kehittämisen kannalta ei ole kaikilta osiltaan järkevää. Ensinnäkin prosessiin osallistuneiden henkilöiden lukumäärä ja tehtäväjakauma poikkeaa käytännön kehitysprosessista. Samoin prosessin läpiviennin tavoitteet poikkesivat normaalista suuresti, koska tärkeimpänä tavoitteena oli tutkielman tekeminen, tuloksena syntyvän kielen ollessa toki tarpeellinen mutta siltikin vasta toissijainen tavoite. Tästä huolimatta edellä kuvattu prosessi sisältää useita Kellyn ja Tolvasen (2008) esittämiä ja aiemmin kohdassa 2.6 kuvattuja, yleisesti sovellusaluekeskeisen ohjelmistokehityksen käyttöönottoon liittyviä vaiheita.

Prosessi oli ensinnäkin vahvasti iteratiivinen ja inkrementaalinen. Tämä johtui osaltaan siitä, että tämä on kielen kehittäjälle ominainen tapa työskennellä, mutta myös siitä, että tapa oli kokonaan uuden ohjelmistoteknisen kokonaisuuden kehittämisessä selvästi toimivin. Yhden kapean osa-alueen toteuttaminen vähintään testiversioksi antoi mahdollisuuden opetella samalla työkalun käyttöä. Samoin pienen testimallin, siitä tehtävien muunnosten ja sen käyttöön luodun minimaalisen sovelluskehyksen toimivuuden toteaminen antoi viitteitä koko lähestymistavan toimivuudesta jo hyvin aikaisessa vaiheessa.

Toinen yhtäläisyys yleisten sovellusaluekeskeisen ohjelmistokehityksen prosessien kanssa on se, että ratkaisun kehittäjällä on aiempaa kokemusta tämän sovellusalueen ratkaisujen kehittämisestä. Tällä tavoin sovellusalue oli jo ennestään tuttu ja tarvittavat lähtötiedot sekä tavoitteena oleva lopputulos olivat suhteellisen selkeitä jo työtä aloitettaessa. Myös organisaatio, jota varten sovel-

lusaluekeskeistä ratkaisua kehitettiin (Digia Financial Software), on kyseiselle sovellusalueelle kehitettävien ohjelmistotuotteiden asiantuntijaorganisaatio. Vaikka mallinnuskielen kehittäjä itse tuntee sovellusaluetta vain päällisin puolin ja syvemmin vain sovelluskehityksen kannalta, organisaatiosta löytyy myös henkilöitä, jotka ovat erittäin kokeneita sovellusalueen asiantuntijoita.

Kielen kehityksen aikataulu oli sellainen, että pääosa kielestä oli jo valmiina, kun kohdassa 2.6. esitellyn kehitysprosessin pääasiallisena lähteenä käytetty Kellyn ja Tolvasen teos (2008) ilmestyi. Tästä huolimatta kielen kehitys kulki yllättävän luontevasti heidän esittämäänsä polkua, vaikkakin tässä tapauksessa sama henkilö täytti sekä asiasta innostuneen kehittäjän että aiheesta hiljalleen vakuuttuvan johtoportaan edustajan roolin: alkuinnostus vaihtui nopeasti epäilyyn lähestymistavan käyttökelpoisuudesta, kunnes taas kehityksen hiljalleen edetessä se alkoi näyttää soveltuvan ratkaisuksi kaikkiin mahdollisiin ongelmiin, edustivatpa nämä ongelmat mallinnettavaa sovellusaluetta tai jotain aivan muuta, edes etäisesti ohjelmistokehitystä sivuavaa aihetta. Tästä johtoportaan ja muun henkilöstön yli-innostumisestahan Kelly ja Tolvanen (2008, 339) prosessin kuvauksessaan juuri varoittavat.

Tämän tutkielman valmistuessa lähestymistavan käyttöönottoprosessi lienee kohdassa esitetyn 2.6. jaottelun mukaan jossain vaiheiden 3 ja 4 välimaastossa. Prosessin ensimmäiset askeleet otettiin jo ennen tutkielman teon aloittamista: mallinnettava sovellusalue oli toki jo valittuna, samoin yhden hengen projektiryhmä oli koottu sekä ryhmän ulkopuoliset tuki- ja ohjaustahot valittu. Tutkielman yhteydessä on valmistunut toimiva sovellusaluekieli koodigeneraattoreineen ja sovelluskehyksineen, ja tätä kieltä voidaan käyttää rahaston arvonlaskentaprosessin osittaiseen mallintamiseen. Kieli ei kuitenkaan ole vielä valmis, mutta sen potentiaali on kuitenkin pystytty osoittamaan.

4.6 Kielen käyttö ja arviointi

Mallinnuskielen suunnittelun tueksi on esitetty kirjallisuudessa periaatteita, joita noudattaen on helpompaa päästä hyvään lopputulokseen. Esimerkiksi Paige, Ostroff ja Brooke (2000) esittävät yhdeksän ominaisuutta, jotka mallinnuskielen tulisi toteuttaa. Taulukossa 5 on esitetty nämä periaatteet sekä arvioitu tutkielman yhteydessä kehitettyä kieltä kunkin periaatteen osalta.

Yleisesti voidaan sanoa, että kielen monet osa-alueet vaikuttavat toteuttavan Paigen ym. (2000) esittämät vaatimukset ainakin jossain määrin. Ongelmana arvioinnissa on se, että kieli on kehityksensä alkuvaiheessa, eikä sillä ole mielekästä mallintaa todellista arvonlaskentaprosessia, koska siitä puuttuvat kokonaan eräät tarpeelliset toiminnot. Esimerkkinä tällaisesta toiminnosta ovat taulukkomuotoisten arvojoukkojen käsittelyt (eli jonkinlainen .NET:in for-each -kontrollirakennetta vastaava käsittelytapa). Tästä syystä ei voida myöskään vielä arvioida kielen toimivuutta todellisessa tilanteessa.

Tutkielman yhteydessä kehitetyn kielen silmiinpistävin piirre on sen ulkonäkö, joka on suorastaan rujo. Tähän on syynä paitsi ulkonäön hiomiseen käytetyn ajan vähäisyys, myös se, että kielen toteuttajan kyvyt visuaalisen suunnittelun alalla ovat parhaimmillaankin vain välttävät. Lisäksi pääasiallinen kiinnostuksen kohde kielen kehityksessä oli sen tekninen toteutus, joten visuaalisen ilmeen suunnittelu jäi hyvin vähälle. Tämä puute on syytä korjata ennen kuin kieltä lähdetään viemään tuotantokäyttöön. Haugen ja Mohagheghi (2007, 17) katsovat visuaalisen syntaksin tärkeäksi ominaisuudeksi sen, että erilaiset käsitteet erottuvat toisistaan selkeästi, joten tämä osa kielestä tulee jatkossa ottaa erityisen huomion kohteeksi.

Kielen vahvuutena voidaan pitää sen suhteellisen helppoa omaksuttavuutta. Tämä lienee seurausta lähinnä kielen yksinkertaisuudesta, jolloin opeteltavia toimintoja ja elementtejä on rajallinen määrä. Toisaalta tämä tarkoittaa myös sitä, että sovellusalueelta mahdollisesti löytyviä suurempia kokonaisuuksia ei

ole mallinnettu, vaan käytettävät elementit ovat edelleen hyvin atomisella tasolla. Tämä taas aiheuttaa sen, että kaavaan pitää piirtää paljon toimintoja. Jos sovellusalueen analyysi olisi tehty syvällisemmin ja abstraktiotasoa olisi pystytty nostamaan korkeammaksi, voisivat monet laskutoimenpiteet itse asiassa sisältyä johonkin käsitteelliseen kokonaisuuteen, joka piilottaisi laskennan kokonaan mallintajalta.

Haugen ja Mohagheghi toteavat, että sovellusaluekielen käytön tulisi olla tehokkaampaa verrattuna yleiskäyttöisen kielen kääyttöön (Haugen & Mohagheghi 2007, 17). Tämä vaatimus toteutuu ainakin siltä osin kuin sitä kielen tässä kehitysvaiheessa voidaan arvioida. Kielellä pystytään mallintamaan laskentakaava suhteellisen nopeasti. Kielen kehityksen loppuvaiheessa mallinnettiin erittäin yksinkertainen laskentakaava tietolähteineen. Tämä onnistui noin tunnissa, vaikka samassa yhteydessä korjattiin eräitä työkaluista löytyneitä puutteita ja virheitä. Itse kaavan piirtäminen oli nopeaa. Tästä ei kuitenkaan voida tehdä päätelmää kielen yleisestä käytettävyydestä, koska testaajana toimi kielen kehittäjä itse. Samoin mallinnettu laskentaprosessi oli hyvin yksinkertainen. Tästä huolimatta laskentapuusta tuli melko suuri, joten mahdollisuuksia eri laskentavaiheiden kapselointiin suurempien kokonaisuuksien sisälle tulisi etsiä. Eräs tarpeellinen lisätoiminto olisi laskentapuun jonkin haaran piilottaminen hiiren oikean napin ponnahdusvalikon kautta. Tällä tavoin voitaisiin kaavaa tarkasteltaessa piilottaa sellaiset osat, jotka eivät ole sillä hetkellä mielenkiintoisia. Toinen vaihtoehto olisi se, että osa laskentapuusta voitaisiin piirtää omaan laatikkoonsa, jonka sisältö voitaisiin piilottaa ja näyttää tarpeen mukaan. Kyse olisi siis eräänlaisesta mustan laatikon periaatteesta. Tämä erillinen laatikko sisältäisi itse asiassa pienemmän mallin, joka on osa suurempaa, koko laskentapuun kuvaavaa mallia.

TAULUKKO 5. Mallinnuskielen tärkeät ominaisuudet (Paige ym. 2000) ja NAV-kielen arviointi kunkin ominaisuuden osalta.

Ominaisuus	**NAV-kieli**
Yksinkertaisuus: Kielessä ei ole tarpeetonta monimutkaisuutta. Näin kieli on myös pieni, ja se on helppo oppia.	Vaatimus toteutuu. Syynä tosin on pääasiassa se, että kielessä on vähän elementtejä, jolloin niistä koostuvat kieli on luonnostaan suhteellisen yksinkertainen.
Yksiselitteisyys: Kielessä ei ole useita elementtejä, jotka kuvaavat samaa semanttista käsitettä. Kieli siis tarjoaa yhden ja vain yhden hyvän tavan kunkin asian kuvaamiseen.	Vaatimus toteutuu ainakin siltä osin, että saman asian toteuttamiseen ei ole useaa keinoa. Toteutustapojen hionta on kuitenkin kesken. Tähänkin vaikuttaa kielen elementtien vähäinen määrä, samoin kuin alhainen abstraktiotaso.
Yhtenäisyys: Kielellä esitetyn mallin eri osien tai eri mallien tulee olla yhtäpitäviä, ja niiden tulee toimia yhdessä kielen tavoitteiden saavuttamiseksi.	Vaatimus toteutuu. Kielen elementit toimivat yhdessä, ja niillä pystytään toteuttamaan laskentaprosessin malli. Tästä mallista voidaan generoida toimiva sovellus.
Saumattomuus: Kielen käsitteet ja abstraktiotaso pysyvät samoina koko sillä toteutetun kehitysprosessin ajan.	Vaatimus toteutuu. Tämä vaatimus on itse asiassa sovellusaluekeskeisen lähestymistavan oleellinen osa: mallinnus tehdään sovellusalueen abstraktiotasolla ja ohjelmakoodi generoidaan suoraan tästä mallista.
Muutosten palautettavuus: Mihin tahansa kehitysprosessin vaiheeseen tehty muutos viedään automaattisesti myös muihin vaiheisiin, myös siten että toteutukseen tehty muutos vyörytetään myös malliin.	Vaatimus ei toteudu. Kielen eri osat ovat irrallisia, eikä niiden välillä ole automaattista ylläpitoa. Tämä tosin on valitussa lähestymistavassa tarpeetonta, koska prosessin tulisi lähteä aina liikkeelle metamallista.
Skaalautuvuus: Kielen pitäisi olla käyttökelpoinen riippumatta mallinnettavan järjestelmän koosta.	Vaikea arvioida tässä vaiheessa. Pieneen laskentakaavaan kieli soveltuu, mutta suurempien mallintaminen voi osoittautua epäkäytännölliseksi.
Tuettavuus: Kielelle pitäisi olla mahdollista toteuttaa sen käyttöä tukevia työkaluja, kuten mallinnustyökaluja ja koodigeneraattoreita.	Vaatimus toteutuu. Mallinnustyökalu on DSL Toolsin tarjoama, ja koodigeneraattorit ovat olennainen osa lähestymistapaa.
Luotettavuus: Kielen pitäisi toimia määritystensä mukaisesti ja reagoida virheisiin oikein. Sen pitäisi tukea toimivien sovellusten kehittämistä.	Toteutuu jossain määrin. Kieli toimii kuten sen on määritelty toimivan. Oikeellisuustarkistukset ovat sen sijaan riittämättömät, mutta niiden toteuttaminen jatkossa on mahdollista.
Tilankäytön taloudellisuus: Kielellä tehtyjen mallien pitäisi viedä mahdollisimman vähän tilaa.	Ei toteudu. Kielessä ei ole mahdollisuuksia muokata mallin osien tarkkuustasoa, joten laskentapuu kasvaa nopeasti.

Mallinnuskieli on siis periaatteessa käyttökelpoinen jo nyt. Se vaatii kuitenkin vielä lisäkehitystä ennen kuin sitä voidaan lähteä viemään tuotantokäyttöön. Luultavasti on syytä pyrkiä järjestämään pilottiprojekti, jossa uusi sovellus viedään yhden asiakkaan käyttöön, ja tästä saatujen kokemusten pohjalta päätetään kielen jatkokehityksestä.

4.7 Yhteenveto

Luvussa esiteltiin DSL Toolsilla toteutettu sovellusaluekieli. Kielellä voidaan mallintaa sijoitusrahaston arvonlaskentaprosessi. Mallilla voidaan esittää arvonlaskennan laskukaava sekä laskennan perustana käytettävien lähtöarvojen tietolähteet. Sovellusaluekielellä toteutetutta mallia käyttävä koodigeneraattori sekä tämän generoimaa koodia käyttävä sovelluskehys käytiin myös läpi. Kielen jatkokehitystarpeista tärkeimmät ovat visuaalisen ilmeen huomattava parantaminen, sovellusalueen kattavampi analysointi sekä tämän pohjalta tehtävä kielen abstraktiotason nostaminen.

5 YHTEENVETO

Tämän tutkielman tavoitteena oli selvittää, mitä sovellusaluekeskeisellä ohjelmistokehityksellä, sovellusaluemallinnuksella ja sovellusaluekielellä tarkoitetaan. Lisäksi tavoitteena oli kehittää sovellusaluekieli sijoitusrahaston arvonlaskentaprosessin mallintamiseksi ja saada tällä tavoin kokemuksia siitä, miten sovellusaluemallinnusta tehdään käytännössä.

Sovellusaluekeskeisen ohjelmistokehityksen pyrkimyksenä on nostaa kohdealueen mallinnuksen abstraktiotaso niin korkealle, että mallinnus tehdään sovellusalueen omilla käsitteillä ja termeillä. Tällaista mallinnuksen lähestymistapaa kutsutaan sovellusaluemallinnukseksi. Sovellusaluemallinnusta varten tarvitaan sovellusaluekieli, jonka syntaksi on räätälöity vastaamaan kohdealueen termistöä. Jotta sovelluskehityksessä ei tarvittaisi syvällistä ohjelmistokehityksen tuntemusta, sovellusaluekielellä tehdystä sovellusaluemallista generoidaan ohjelmakoodi automaattisesti. Eri sovellusten yhteiset, muuttumattomina pysyvät osat pyritään toteuttamaan erilliseen sovelluskehykseen.

Sovellusaluemallinnusta varten on kehitetty erilaisia työkaluja. Tässä tutkielmassa käytettiin Microsoftin DSL Tools -työkalua, ja se esiteltiin tarpeellisilta osiltaan. DSL Tools on osa Software Factories -konseptia, jolla Microsoft pyrkii teollistamaan ohjelmistokehitystä. DSL Tools on vielä kehitysvaiheessa, mutta puutteistaan huolimatta se on käyttökelpoinen työkalu sovellusaluekielen toteutukseen. Työkalussa on eräitä erittäin hyviä ominaisuuksia, joten puutteiden tulevaisuudessa korjaantuessa voi DSL Tools olla tehokas ja kohtalaisen helppokäyttöinen apuväline sovelluskehityksen automatisoinnissa.

Tutkielman käytännön osuuden sovellusalueena oli sijoitusrahaston arvonlaskentaprosessi, jonka mallintamista varten määritettiin sovellusaluekieli. Kielelle asetettiin erityisesti vaatimukseksi se, että se olisi rahoitusalan käyttöön tarkoitetun sovelluksen loppukäyttäjän ymmärrettävissä ja käytettävissä. Kieli ei siis

saanut vaatia syvällistä sovelluskehityksen tuntemusta. Toteutettu testikieli sisältää nelilaskimen toiminnot, eräitä matemaattisia funktioita, taulukkomuotoisen tietorakenteen sekä kolme erilaista tietolähdettä. Näiden elementtien sekä niitä kuvaavien symboleiden toteuttamiseen tarvittiin hieman yli 30 luokkaa sekä kymmenkunta niiden välistä suhdetta, yksi koodigeneraattori sekä hyvin suppea sovelluskehys. Kieli toteutettiin DSL Toolsilla. Kielen kehittämisestä saatuna kokemuksena voidaan todeta, että DSL Toolsilla on mahdollista, ja jopa suhteellisen helppoa, kehittää sovellusaluekieli ja mallintaa sen avulla ainakin yksinkertaisia sovelluksia. Jatkossa on kuitenkin parhaan vaihtoehdon löytämiseksi syytä tutustua myös muihin työkaluihin.

Kehitetyn sovellusaluekielen voidaan arvioida vastaavan sille asetettuja tavoitteita. Kielen käyttö on melko yksinkertaista, kunhan käyttäjä on ensin oppinut kielen peruslogiikan. Sovellusaluemallista generoitava ohjelmakoodi sisältää vain tarvittavien olioiden luonnin ja näiden olioiden välisten suhteiden määrittelyn, minkä ansiosta generaattori on hyvin yksinkertainen. Varsinainen laskentalogiikka on toteutettu sovelluskehykseen, joka sekään ei ole kovin monimutkainen.

Mallinnuskieli vaikuttaisi soveltuvan pohjaksi jatkokehitykseen. Kieltä pitää toki laajentaa huomattavasti ennen kuin sillä on mahdollista toteuttaa tuotantokäyttöön tarkoitettu sovellus. Tärkeitä jatkokehittämisen kohteita ovat ainakin seuraavat:

1. Kielen visuaalinen ilme. Tällä hetkellä kieli ei ole ulkoisesti riittävän havainnollinen, jotta sitä voitaisiin käyttää tuotantoympäristössä. Lisäksi kielen ulkonäön on syytä olla käyttäjänsä silmää miellyttävä; tämänhetkisen version tila on kaukana tästä tavoitteesta.

2. Mallin tarkkuustason vaihto mallinnuksen aikana. Laskentapuun kasvaessa kohti todellista laajuuttaan se muuttuu samalla melko hankalaksi hahmottaa. Tätä ongelmaa voitaisiin helpottaa luomalla keino poistaa

näkyvistä vähemmän olennaiset osat. Poistaminen voitaisiin tehdä esimerkiksi siten, että puu romahdutettaisiin (collapse) jostain elementistä alkaen. Toinen vaihtoehto olisi lisätä metamalliin jonkinlainen ryhmittelyelementti, jonka sisälle osa laskentapuusta voitaisiin mallintaa. Elementin sisältö voitaisiin tämän jälkeen piilottaa ja tarvittaessa avata tämä eräänlainen ”musta laatikko” uudelleen näkyville.

3. Laskentapuun tallennus tietokantaan. Paitsi mallin tallennuksen, myös sen arvojen tallennuksen toteutus on jätetty tämän työn ulkopuolelle, mutta ilman niitä kielen käyttö on mahdotonta. Sijoitusrahaston arvonlaskennan oleellinen osa-alue on sen arvojen vertailu edellisen laskennan arvoihin, mikä on myös toteutettava. Tässä yhteydessä on otettava huomioon myös mahdolliset malliin sen käytön aikana tehtävät muutokset, jotka eivät saa rikkoa laskennan jatkumoa. Tietokantakäsittelyn toteutuksen yhteydessä on syytä tutkia mahdollisuutta käyttää kantarakenteen suunnittelussa ja liiketoimintaluokkien luonnissa Moilasen (2006) aiemmin kehittämää EML-mallinnuskieltä.

4. Mallin oikeellisuustarkistusten lisääminen. Tällä hetkellä malliin on toteutettu vain muutamia sääntöjä ja rajoitteita. Kuitenkin mallinnusvaiheessa näistä olisi suurta hyötyä, ja joiltain osin ne ovat suorastaan elintärkeitä. Tämä on vain yksi osa mallin käsittelyn lisäkehitystarpeita: esimerkiksi mallin elementtien nimiä ei tällä hetkellä tarkisteta lainkaan, minkä seurauksena vaikkapa välilyönnin sisältävä nimi aiheuttaa generoidun koodin rikkoutumisen.

Onko lopulta edes järkevää tehdä mallinnusta visuaalisella työkalulla? Olisiko tehokkaampaa luoda tekstimuotoinen kaavakieli, jolla mallinnus tehdään? Tällaisenkaan kielen opettelu tuskin olisi mahdoton tehtävä sovellusalueen asiantuntijalle, joskin se voisi aluksi olla hieman hankalampaa. Toisaalta laskentapuun hahmottaminen on helpompaa, kun se on esitetty graafisesti. Ehkä olisi

mahdollista löytää jonkinlainen välimuoto, mahdollisesti siten, että molempia tapoja voitaisiin tukea: kaava voitaisiin kirjoittaa tekstimuotoisena, mutta muuntaa se myöhemmin graafiseksi. Kysehän olisi vain mallin esitystavan vaihtamisesta, mikä ei ole mahdotonta. Työlästä se toki voi olla.

Eräs mielenkiintoinen tutkimuskohde on prosessin ja sovellusalueen pidemmälle viety analyysi. Tällä tavoin saattaisi olla mahdollista nostaa kielen abstraktiotasoa vieläkin korkeammalle. Tällä hetkellähän kieli toimii laskukaavan tasolla, eikä siinä ole mallinnettu suurempia käsitteellisiä kokonaisuuksia lainkaan. Tästä seuraa se, että sovellusaluekeskeisen ohjelmistokehityksen tärkeintä hyötyä ei päästä käyttämään juuri lainkaan. Sijoitusrahastojen arvonlaskentaprosessista on varmasti löydettävissä tällaisia kokonaisuuksia, ja niiden avulla laskentapuuta voitaisiin yksinkertaistaa huomattavasti. Näitä tuloksia olisi mahdollista käyttää jatkossa hyväksi myös muissa sijoituspalvelu- ja rahoitusalan sovelluksissa.

LÄHDELUETTELO

Aagedal, J. Ø. & Solheim, I., 2004. New roles in model-driven development. Teoksessa Akehurst, D. H. (toim.): Computer Science at Kent. Second European Workshop on Model Driven Acrchitercture. Proceedings. Technical Report No. 17-04. Kent, UK. 109-115.

Bentley, J., 1986. Programming pearls: little languages. Communications of the ACM, Vol. 29, Issue 8, 711-721.

Berard, E. V., 1993. Essays on object-oriented software engineering. Englewood Cliffs, New Jersey, USA.

Bierhoff, K., Liongosari, E. S. & Swaminathan, K. S., 2006. Incremental development of a domain-specific language that supports multiple application styles. Teoksessa Gray, J., Tolvanen, J.-P. & Sprinkle, J. (toim.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06). Computer Science and Information Systems Reports, Technical Reports, TR-37, University of Jyväskylä. 67-78.

Blake, D., 2007. Domain-specific languages versus generic modeling languages. Weighing-in on the DSL versus standardized language debate. Steven Kellyn haastattelu. Dr. Dobb's Journal, 10.5.2007. Saatavilla www-muodossa: <http://www.ddj.com/architect/199500627> [viitattu 9.1.2008].

Cleaveland, J. G., 1988. Building application generators. IEEE Software, Volume 5, Issue 4, 25-33.

Cleenewerck, T., 2003. Component-based DSL development. Teoksessa Pfenning, F. & Smaragdakis, Y. (toim.) Generative programming and component engineering. Second International Conference (GPCE 2003), Proceedings. Saksa. 245-264.

Cook, S., Jones, G., Kent, S & Wills, A. C., 2007. Domain-specific development with Visual Studio DSL Tools.

Crnkovic, I., 2003. Component-based software engineering - new challenges in software development. Teoksessa Budin, L., Stiffler-Lužar, V., Bekić, Z. & Dobrić, V. H. (toim.) Proceedings of the 25th Internation Conference on Information Technology Interfaces. University of Zagreb. Kroatia. 9-18.

van Deursen, A., Klint, P. & Visser J., 2000. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices, Vol. 35, Issue 6, 26-36.

Eclipse Foundation, The, 2008. About GEMS. [online] [viitattu 26.1.2008] Saatavilla www-muodossa: <http://www.eclipse.org/gmt/gems/about.php>.

Falkenberg, E. D., Hesse, W., Lindgreen, P., Nilsson, B. E., Oei, J. L. H., Rolland, C., Stamper, R. K., Van Assche, F. J. M., Verrjin-Stuart, A. A. & Voss, K., 1998. A framework of information system concepts, The FRISCO Report, IFIP.

Firesmith D. & Henderson-Sellers B., 1999. Improvements to the OPEN process metamodel. Journal of Object-Oriented Programming, November/December.

de Geest, G., Savelkoul, A. & Alikoski, A., 2007. Building a framework to support domains specific language evolution using Microsoft DSL Tools. Teoksessa Sprinkle, J., Gray, J., Rossi, M. & Tolvanen, J.-P. (toim.) Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). Computer Science and Information Systems Reports, Technical Reports, TR-38, University of Jyväskylä. 60-71.

Greenfield, J. & Short, K., 2004. Software Factories. Assembling applications with patterns, models, frameworks and tools. Indianapolis, USA.

Haikala, I. & Märijärvi, J., 1998. Ohjelmistotuotanto. Jyväskylä.

Haugen, Ø. & Mohagheghi, P., 2007. A multi-dimensional framework for characterizinf domain specific languages. Teoksessa Sprinkle, J., Gray, J., Rossi, M. & Tolvanen, J.-P. (toim.) Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). Computer Science and Information Systems Reports, Technical Reports, TR-38, University of Jyväskylä. 10-19.

Jacobson, I., Booch, G. & Rumbaugh, J., 1999. The unified software development process. USA.

Kahn, H., Rumpe, B. & Völkel, S., 2006. Roles in software development using domain specific modelling languages. Teoksessa Gray, J., Tolvanen, J.-P. & Sprinkle, J. (toim.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06). Computer Science and Information Systems Reports, Technical Reports, TR-37, University of Jyväskylä. 150-158.

Kelly, S., 2005. Improving developer productivity with domain-specific modeling languages. [online] [viitattu 26.1.2008] Saatavilla pdf-muodossa: <http://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Kelly_DomainModeling.pdf>.

Kelly, S., 2006. Generating code with DSM. Saatavilla www-muodossa: http://www.codegeneration.net/tiki-read_article.php?articleId=81.

Kelly, S. & Tolvanen, J.-P., 2008. Domain-specific modeling. Enabling full code generation. Hoboken, NJ, USA.

Kieburtz, R.B., McKinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I. & Walton, L., 1996. A software engineering experiment in software component creation. IEEE Software Engineering,

Proceedings of the 5-618th International Conference on 25-30 March 1996. 542-552.

Landin, P.J., 1966. The next 700 programming lanuages. Communications of the ACM, Vol. 9, Issue 3, 157-166.

Langois, B., Jitia, C-E. & Jouenne, E., 2007. DSL classification. Teoksessa Sprinkle, J., Gray, J., Rossi, M. & Tolvanen, J.-P. (toim.) Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). Computer Science and Information Systems Reports, Technical Reports, TR-38, University of Jyväskylä. 28-38.

Leppänen, M., 2005. An ontological framework and a methodical skeleton for method engineering - a contextual approach. Jyväskylä Studies in Computing 52. Jyväskylän yliopisto, Tietojenkäsittelytieteen väitöskirja.

Luoma, J., Kelly, S. & Tolvanen, J.-P., 2004. Defining domain-specific modeling languages: collected experiences. Teoksessa Tolvanen, J.-P., Sprinkle, J, & Rossi, M. (toim.) Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04). Computer Science and Information System Reports, Technical Reports, TR-33, University of Jyväskylä, 1-10.

Mernik, M., Heering, J. & Sloane, A.M., 2005. When and how to develop domain-specific languages. ACM Computing Surveys, Vol. 37, Issue 4, 316-344.

Moilanen, R., 2006. Sovellusaluemallinnus ohjelmistotuotannon tukena. Jyväskylän yliopisto, Tietotekniikan pro gradu -tutkielma.

MSDN (Microsoft Developer Network). C# programming guide. Partial class definitions. [online] [viitattu 9.1.2008] Saatavilla www-muodossa: <http://msdn2.microsoft.com/en-us/library/wa80x488(VS.80).aspx>.

OMG (Object Management Group), 2008. Model driven architecture. [online] [viitattu 10.1.2008] Saatavilla www-muodossa: <http://www.omg.org/mda/>.

Paige, R. F., Ostroff, J. S. & Brooke, B. J., 2000. Principles for modeling language design. Information and Software Technology, vol. 42, issue 10. 665-675.

Pitkänen, R. & Mikkonen, T., 2006. Lightweight domain-specific modeling and model-driven development. Teoksessa Gray, J., Tolvanen, J.-P. & Sprinkle, J. (toim.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06). Computer Science and Information Systems Reports, Technical Reports, TR-37, University of Jyväskylä. 159-168.

Pohjonen, R. & Kelly, S., 2002. Domain specific modeling. Improving productivity and time to market. Dr. Dobb's Journal, August 2002. Saatavilla www-muodossa: <http://www.ddj.com/architect/184405121?pgno=1>.

Pohjonen, R. & Tolvanen, J.-P., 2002. Automated production of family members: lessons learned. Teoksessa Schmid, K. & Geppert, B. (toim.) Proceedings of the PLEES'02. International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling and Managing. IESE-Report No. 056.02/E. Fraunhofer Institut Experimentelles Software Engineering. 49-58.

Rahoitustarkastus, 2007. Rahoitustarkastuksen tulkinta rahastoyhtiön menettelytavoista sijoitusrahastojen arvonlaskennassa. Dnro 11/125/2007. Saatavilla www-muodossa <http://www.rahoitustarkastus.fi/Fin/Saantely/Tulkinnat_ja_kannanotot/2007/4_2007.htm> [viitattu 25.3.2008].

Sammet, J. E., 1972. Programming languages: history and future. Communications of the ACM. Vol. 15, issue 7. 601-610.

Sánchez-Ruiz, A. J., Saeki, M., Langlois, B. & Paiano, R., 2007. Domain-specific software development terminology: do we all speak the same language? Teoksessa Sprinkle, J., Gray, J., Rossi, M. & Tolvanen, J.-P. (toim.) Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). Computer Science and Information Systems Reports, Technical Reports, TR-38, University of Jyväskylä. 1-10.

Savolainen, V., 2001. Verkkoteoria. Porvoo.

Sijoitusrahastolaki 29.1.1999/48. Saatavilla www-muodossa: <http://www.finlex.fi/fi/laki/ajantasa/1999/19990048> [viitattu 25.3.2008].

Tolvanen, J.-P., 2005. Domain-specific modeling for full code generation. Methods & Tools, Fall 2005. 14-23. Saatavilla pdf-muodossa: <http://www.methodsandtools.com/PDF/mt200503.pdf>.

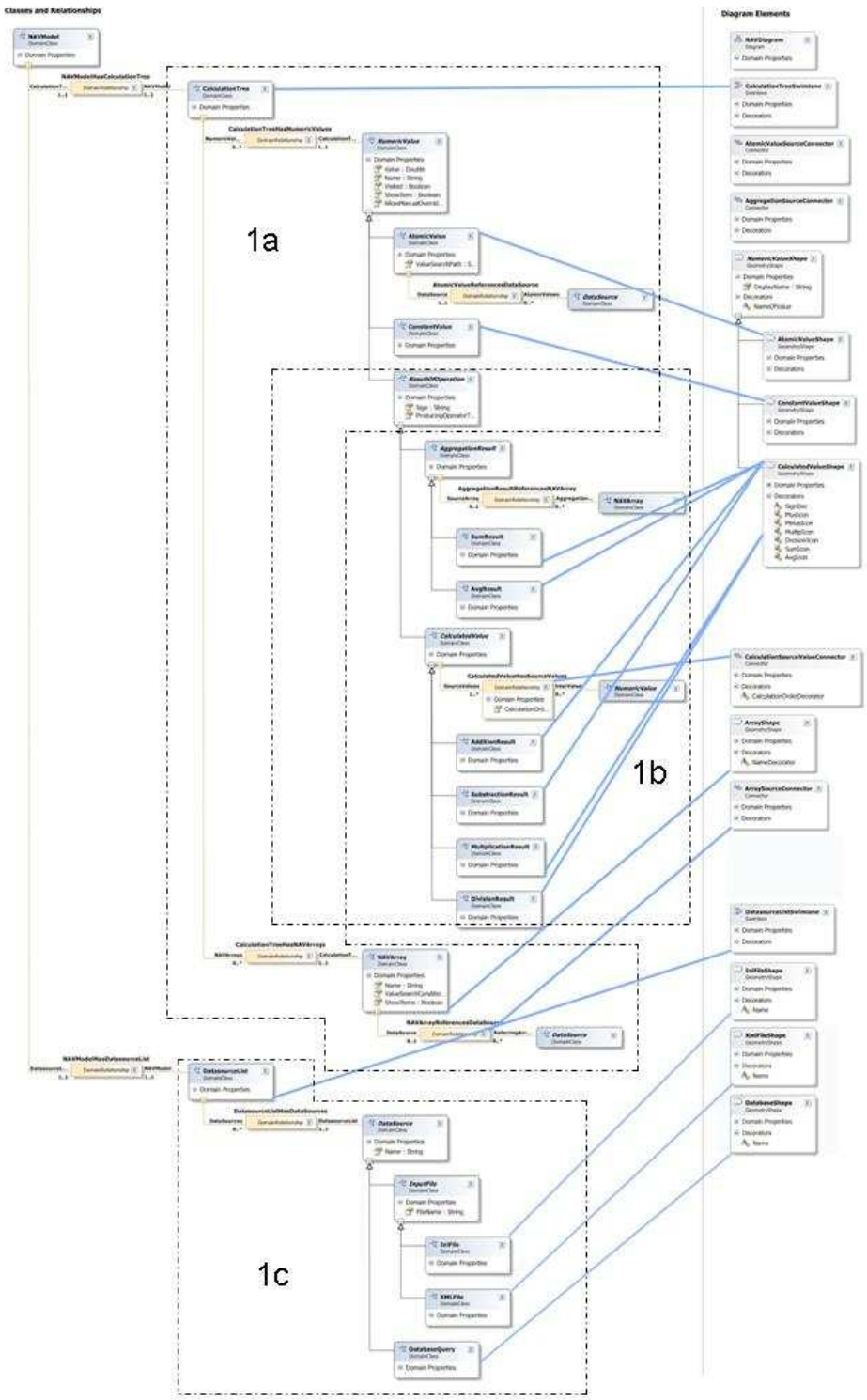
Tucker, A.B., 1985. Programming languages. International student edition. McGraw-Hill Computer Science Series. Singapore.

Völter, M. & Bettin, J., 2004. Patterns for model-driven software development. Saatavilla pdf-muodossa <http://www.voelter.de/data/pub/MDDPatterns.pdf. EuroPLoP 2004>.

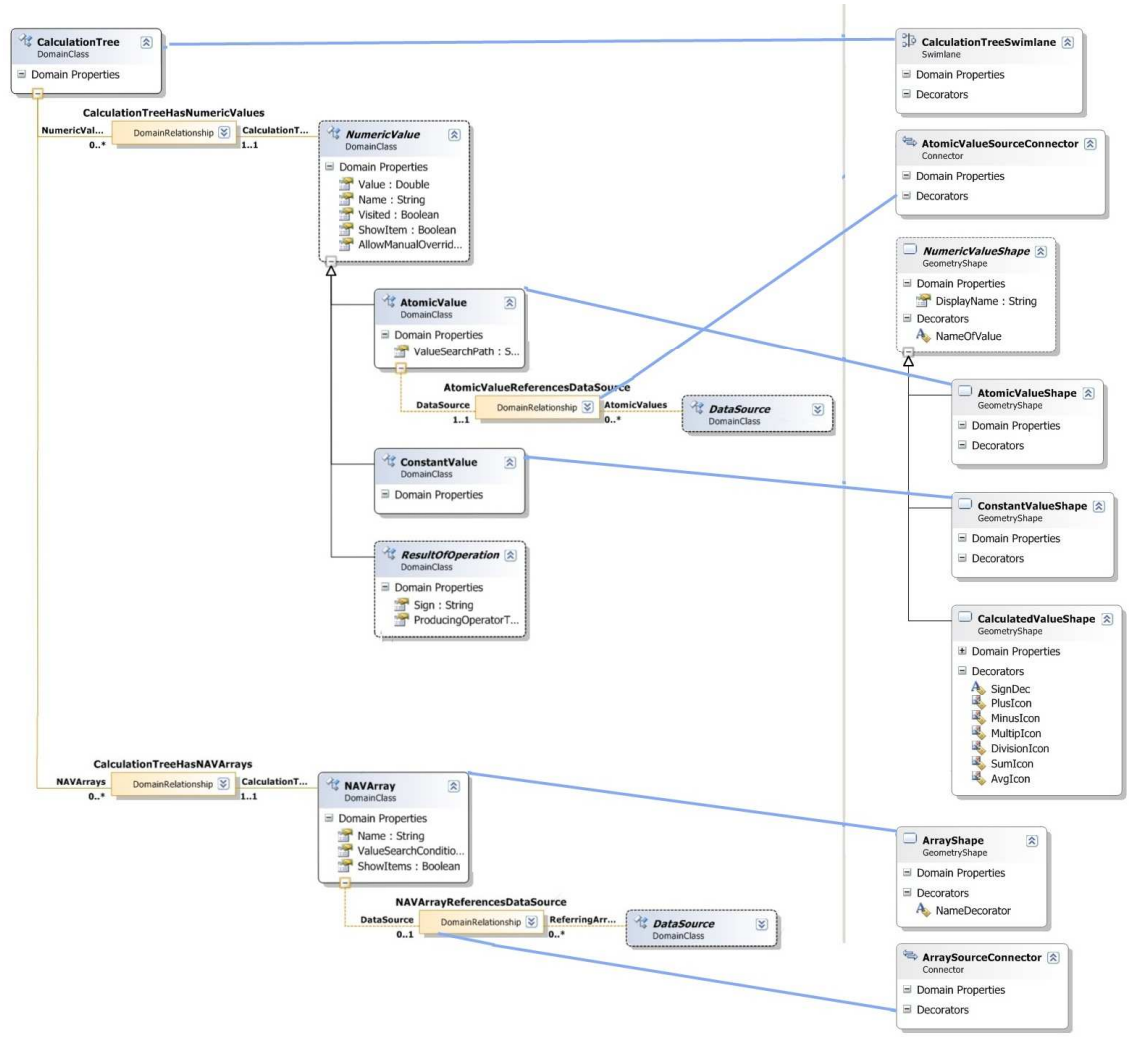
Wile, D., 2004. Lessons learned from real DSL experiments. Science of Computer Programming, vol. 51, issue 3, June 2004. 265-290.

LIITTEET

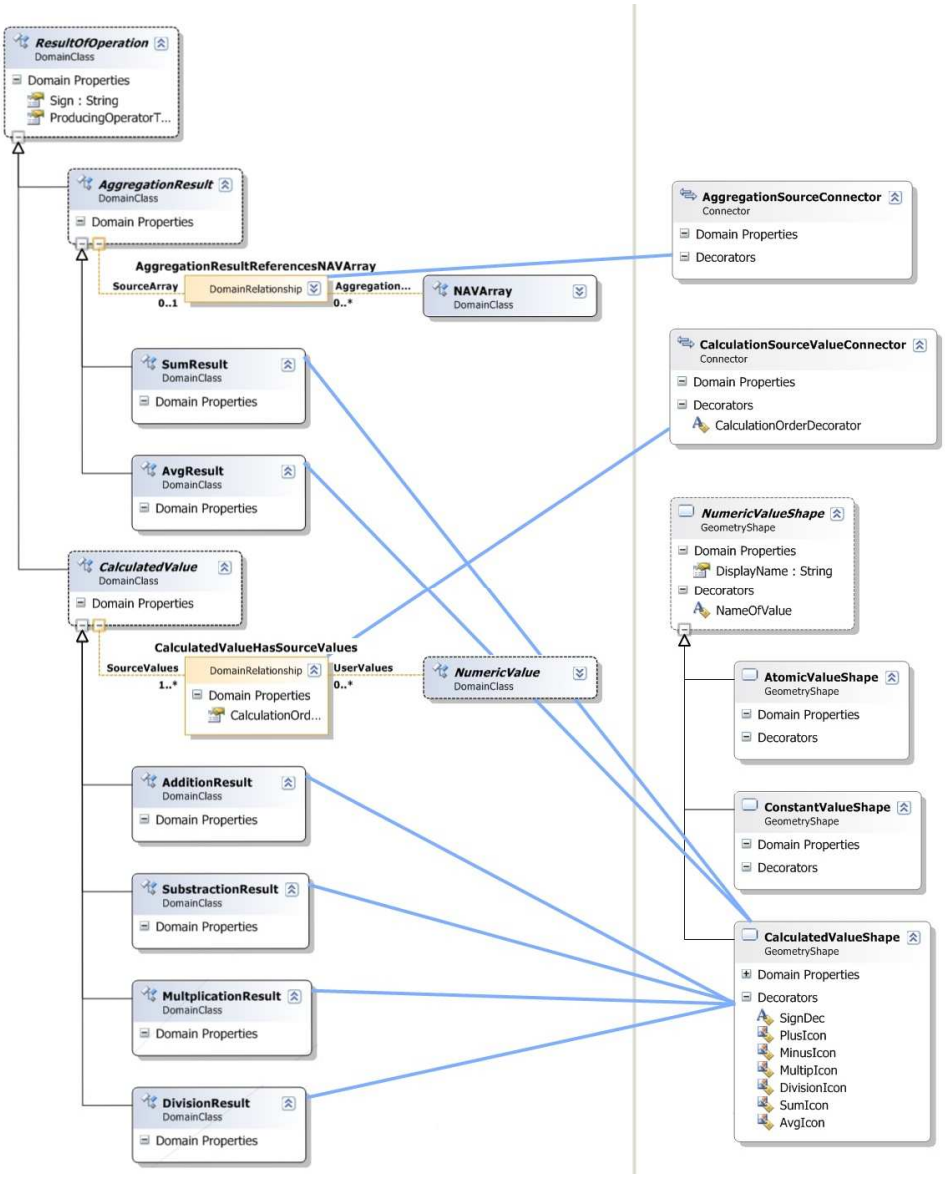
Liite 1: Yleiskuva NAV-kielen metamallista DSL Toolsin suunnittelunäytöllä. Tarkemmat kuvat liitteissä 1a - 1c.



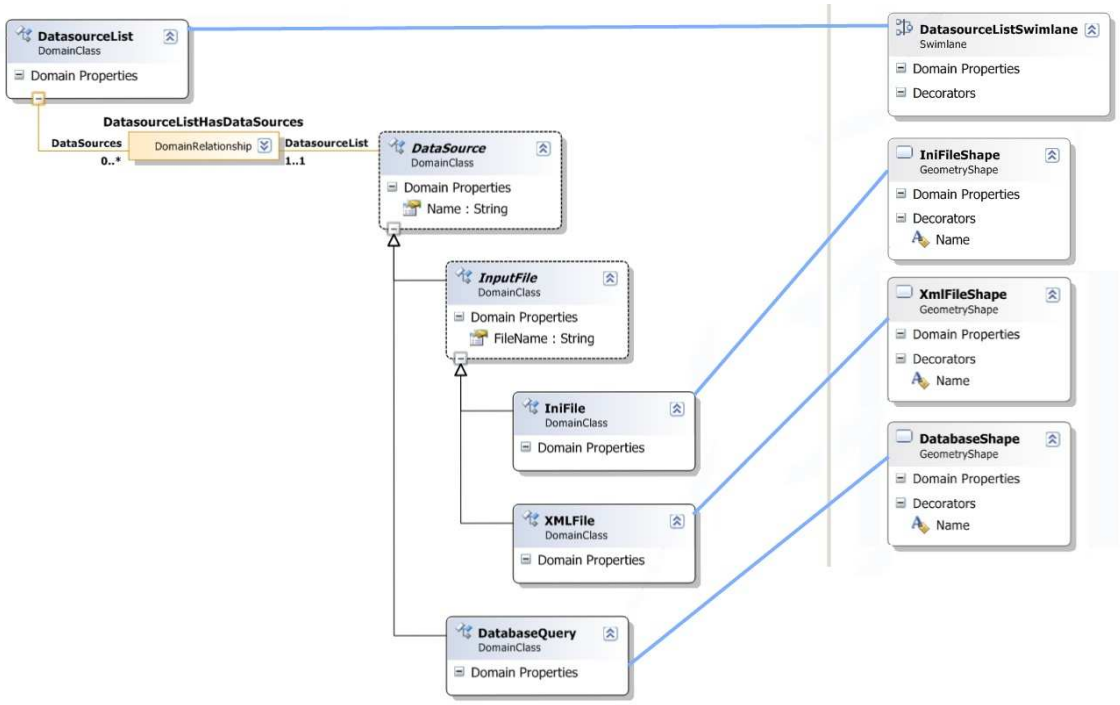
Liite 1a: Osa NAV-kielen metamallista / Laskentapuu



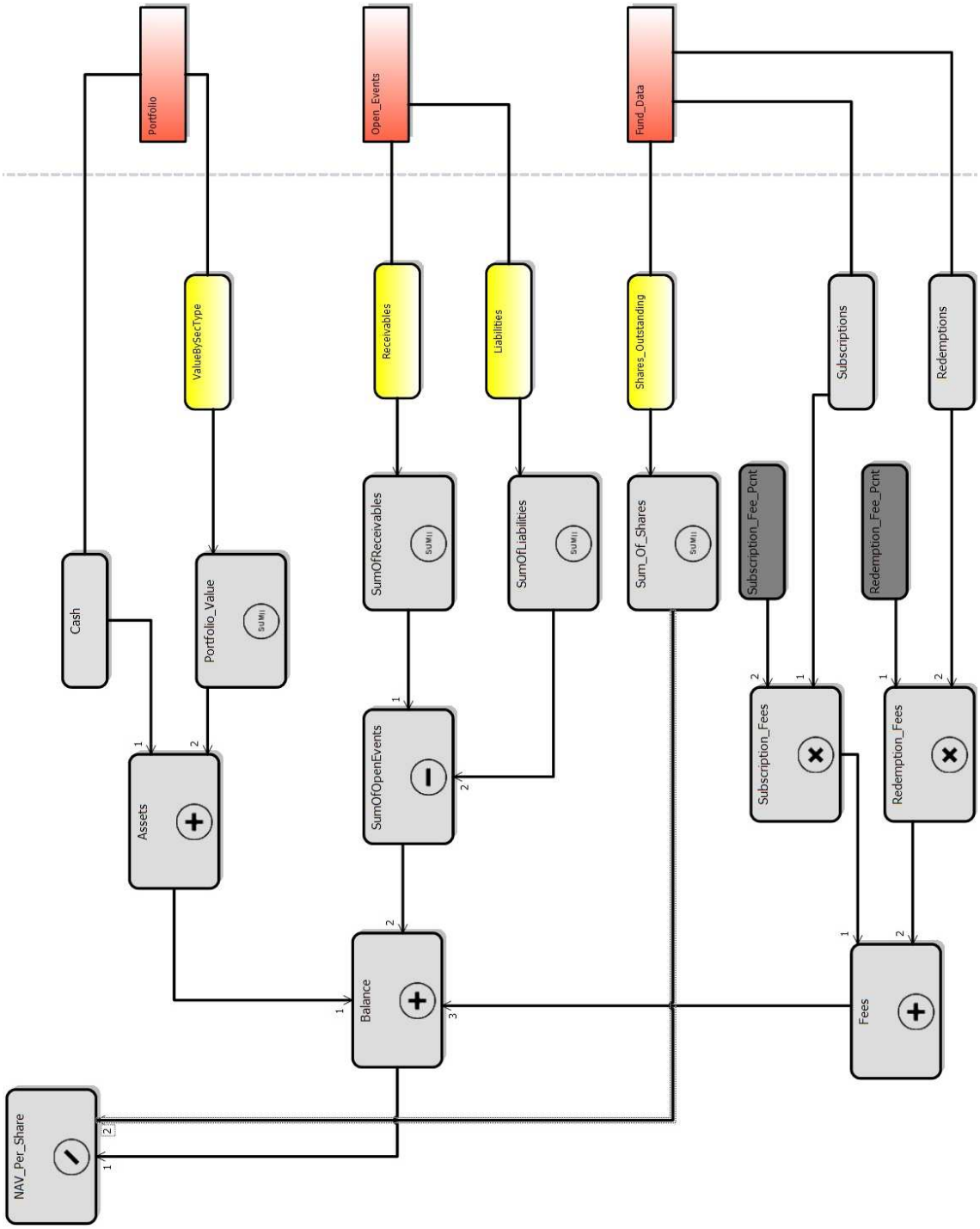
Liite 1b: Osa NAV-kielen metamallista / Numeroarvot



Liite 1c: Osa NAV-kielen metamallista / Tietolähteet



Liite 2: NAV-kielellä toteutettu malli



Liite 3: Koodigeneraattori.

Generaattorin ohjauslohkojen sisältämä koodi on merkitty lihavoinnilla, normaalilla kirjasimella esitetty teksti tulostuu tulostiedostoon sellaisenaan. Kommentit on molemmissa tapauksissa merkitty kursiivilla.

```
<#
' ****************************************************************************
' *** Generator creates objects representing the calculation tree as defined
' *** in the model
' *** AKi 5.2.2008
' ****************************************************************************

' ****************************************************************************
' *** Directives controlling the generator
' ****************************************************************************
#>
<#@ template
        inherits=
        "Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation"
        language="VB" debug="true" imports="System.Text" #>
<#@ output extension=".vb" #>
<#@ NAVDSL processor="NAVDSLDirectiveProcessor" requires="fileName='Test.NAV'" #>

<#@ import namespace="System.Text" #>
<#@ import namespace="MicroSoft.VisualBasic" #>

<#
' ****************************************************************************
' *** Code generation logic begins
' ****************************************************************************
#>
'******************************************************************************
'Generated code from NAVDSL. DO NOT MODIFY MANUALLY!!!
'******************************************************************************

Public Partial Class CalculationTree : Implements ICalculationTree

#Region " Calculation objects "

'****************************************
'Objects representing the data sources
'****************************************

<#
' ****************************************************************************
' *** Iterate through data sources in the model and create corresponding
' *** objects.
' ****************************************************************************
for each ds as DataSource in Me.NAVModel.DatasourceList.DataSources
#>
public shared _<#=ds.Name#> as New <#=ds.GetType.Name#><#
        if TypeOf(ds) is InputFile then
                #>("<#=CType(ds, InputFile).FileName#>")
<#
        end if
#>
<#
next ds
#>


'****************************************
'Objects representing the calculation tree
'****************************************
```

```
<#
' *** Helper objects
dim calculationTreeCreation as new system.text.stringbuilder
dim calculatedValues as new system.collections.arraylist
dim rootValues as new system.collections.arraylist

' ************************************************************************
' *** Iterate through arrays in the model and create corresponding
' *** objects
' ************************************************************************
for each arr as NAVArray in me.NAVModel.CalculationTree.NAVArrays
       #>protected _<#= arr.Name #>_Array as new NAVArray ( "<#= arr.Name #>", _<#=
arr.Datasource.Name #>, "<#= arr.ValueSearchCondition #>", <#= arr.ShowItems.ToString()
#>)
<#
next arr
#>


'***************************************
'Value objects
'***************************************
<#
' ************************************************************************
' *** Iterate through the NumericValues in the model and create corresponding
' *** objects
' ************************************************************************
       for each val as NumericValue in Me.NAVModel.CalculationTree.NumericValues#>
<#
'      *** Class names in the framework match with the ones in the metamodel
'      *** -> use val's TypeName as type of the object to be created
'      *** All classes in the framework have a constructor that accepts
'      *** the name of the instance as a parameter
#>
protected _<#= val.Name #> as new <#
       if typeof( val ) is CalculatedValue then
              #>CalculatedValue<#
       else
       #><#=val.GetType.Name
       #><#
       end if
       #> ("<#= val.Name #>"<#
              if typeof ( val ) is ConstantValue then
' *** Constructor for ConstantValue also requires the value of the object
                     #>, <#= val.Value #><#
              end if
              if typeof ( val ) is AtomicValue then
' *** Constructor for AtomicValue also requires the path to the value

                     #>, _<#= CType(val, AtomicValue).DataSource.Name #>, "<#= CType (
val, AtomicValue ).ValueSearchPath #>"<#

              end if
              if typeof ( val ) is AggregationResult then
' *** Constructor for AggregationResults also requires the data source
                     #>, _<#= ctype(val, AggregationResult).SourceArray.Name #>_Array<#
              end if
              #>, <#= (not val.AllowManualOverride).ToString #>)<#

              if typeof ( val ) is CalculatedValue then
                     ' *** Add value in the array so that the operators can be
                     ‘ *** created later
                     calculatedValues.Add ( val )
              end if

              if val.UserValues.Count = 0 then
                     ' *** No connectors from this value -> Handle as a root value
                     rootValues.Add ( val )
              end if

       next val


#>
```

```
'**************************************
'Operator objects
'**************************************
<#
' ****************************************************************************
' *** Iterate through the CalculatedValues in the model and generate creation
' *** logic for operator objects
' ****************************************************************************
       dim elemDir as IElementDirectory = me.NAVModel.partition.ElementDirectory
       dim src as NumericValue

       for each val as CalculatedValue in calculatedValues#>
<#
' *** Use the name of the operator's resulting value
' *** as base of the operator object's name
#>protected _<#= val.Name #>_Producer as new <#= val.ProducingOperatorType #><#

              ' *** Get the links from and to this object

              dim links as System.Collections.ObjectModel.ReadOnlyCollection _
                     ( Of ElementLink ) = _
                     DomainRoleInfo.GetAllElementLinks ( _
                                                  elemDir.FindElement( val.ID ) )


              for each link as ElementLink in links
                     if link.GetDomainClass.Name = _
                                   "CalculatedValueHasSourceValues" then
                            ' *** This is the source connector for this value, use it
                            ' *** for creating the operator object
                            src = DomainRoleInfo.GetTargetRolePlayer ( link )

                            if not src is val then
                            ' *** Don't add links leading out of the value object

                                   calculationTreeCreation.Append _
                                          ( vbCrLf + vbTab + "_" _
                                          + val.Name + "_Producer.AddSourceValue(_" _
                                          + src.Name + ", " + _
                                                 CType ( link, _
                                                 CalculatedValueHasSourceValues) _
                                          .CalculationOrderNumber.ToString + ")" )

                            end if
                     end if
              next link



              ' *** Add link between the value and the operator producing it
              calculationTreeCreation.Append ( vbCrLf + vbTab + "_" _
                     + val.Name + ".ProducingOperator = _" + val.Name + "_Producer")

       next val
#>


#End Region

<#
' ****************************************************************************
' *** Generate the calculation tree structure initialization logic
' ****************************************************************************
#>

#Region " Calculation logic "

'Method creates the calculation tree structure
public sub New

       'Set the links between values and operators

<#= calculationTreeCreation.ToString #>
```

```
<#
for each val as NumericValue in rootValues
#>
        'Set the root values
        _rootValues.Add (_<#= val.Name #>)
<#
next val
#>
end sub

#End Region

End Class
```

Liite 4: Esimerkki koodigeneraattorin tuottamasta ohjelmakoodista. Rivinvaihdot on lisätty käsin luettavuuden parantamiseksi.

```
'*************************************************************************
'Generated code from NAVDSL. DO NOT MODIFY MANUALLY!!!
'*************************************************************************

Public Partial Class CalculationTree : Implements ICalculationTree

#Region " Calculation objects "

'***************************************
'Objects representing the data sources
'***************************************

public shared _Portfolio as New IniFile("C:\_work\gradu\DSL\Aineistot\Portfolio.dat")
public shared _Open_Events _
              as New IniFile("C:\_work\gradu\DSL\Aineistot\Open_Events.dat")
public shared _Fund_Data as New IniFile("C:\_work\gradu\DSL\Aineistot\Fund_Data.dat")


'***************************************
'Objects representing the calculation tree
'***************************************

protected _ValueBySecType_Array as new NAVArray ( "ValueBySecType", _Portfolio, _
                                                  "*BK_VALUE", False)
protected _OpenEvents_Array_Array as new NAVArray ( "OpenEvents_Array", _
                                                  _Open_Events, "", False)
protected _Receivables_Array as new NAVArray ( "Receivables", _Open_Events, _
                                                  "RECEIVABLES*", False)
protected _Liabilities_Array as new NAVArray ( "Liabilities", _Open_Events, _
                                                  "LIABILITIES*", False)
protected _Shares_Outstanding_Array as new NAVArray ( "Shares_Outstanding", _
                                                  _Fund_Data, _
                                                  "*SHARES_OUTSTANDING", False)


'***************************************
'Value objects
'***************************************

protected _Assets as new CalculatedValue ("Assets", False)
protected _Cash as new AtomicValue ("Cash", _Portfolio, "CASH VALUE", False)
protected _Portfolio_Value as new SumResult ("Portfolio_Value", _
                                  _ValueBySecType_Array, True)
protected _SumOfReceivables as new SumResult ("SumOfReceivables", _
                                  _Receivables_Array, True)
protected _SumOfLiabilities as new SumResult ("SumOfLiabilities", _
                                  _Liabilities_Array, True)
protected _SumOfOpenEvents as new CalculatedValue ("SumOfOpenEvents", True)
protected _Balance as new CalculatedValue ("Balance", True)
protected _Sum_Of_Shares as new SumResult ("Sum_Of_Shares", _
                                  _Shares_Outstanding_Array, True)
protected _NAV_Per_Share as new CalculatedValue ("NAV_Per_Share", True)
protected _Subscriptions as new AtomicValue ("Subscriptions", _Fund_Data, _
                                  "SUBSRIPTION_AMOUNT", True)
protected _Redemptions as new AtomicValue ("Redemptions", _Fund_Data, _
                                  "REDEMPTION_AMOUNT", True)
protected _Subscription_Fee_Pcnt as new ConstantValue ("Subscription_Fee_Pcnt", 0, True)
protected _Redemption_Fee_Pcnt as new ConstantValue ("Redemption_Fee_Pcnt", 0.02, True)
protected _Subscription_Fees as new CalculatedValue ("Subscription_Fees", True)
protected _Redemption_Fees as new CalculatedValue ("Redemption_Fees", True)
protected _Fees as new CalculatedValue ("Fees", True)


'***************************************
'Operator objects
'***************************************
```

```
protected _Assets_Producer as new PlusOperator
protected _SumOfOpenEvents_Producer as new MinusOperator
protected _Balance_Producer as new PlusOperator
protected _NAV_Per_Share_Producer as new DivisionOperator
protected _Subscription_Fees_Producer as new MultiplyOperator
protected _Redemption_Fees_Producer as new MultiplyOperator
protected _Fees_Producer as new PlusOperator

#End Region


#Region " Calculation logic "

'Method creates the calculation tree structure
public sub New

        'Set the links between values and operators


        _Assets_Producer.AddSourceValue(_Cash, 1)
        _Assets_Producer.AddSourceValue(_Portfolio_Value, 2)
        _Assets.ProducingOperator = _Assets_Producer
        _SumOfOpenEvents_Producer.AddSourceValue(_SumOfReceivables, 1)
        _SumOfOpenEvents_Producer.AddSourceValue(_SumOfLiabilities, 2)
        _SumOfOpenEvents.ProducingOperator = _SumOfOpenEvents_Producer
        _Balance_Producer.AddSourceValue(_Assets, 1)
        _Balance_Producer.AddSourceValue(_SumOfOpenEvents, 2)
        _Balance_Producer.AddSourceValue(_Fees, 3)
        _Balance.ProducingOperator = _Balance_Producer
        _NAV_Per_Share_Producer.AddSourceValue(_Balance, 1)
        _NAV_Per_Share_Producer.AddSourceValue(_Sum_Of_Shares, 2)
        _NAV_Per_Share.ProducingOperator = _NAV_Per_Share_Producer
        _Subscription_Fees_Producer.AddSourceValue(_Subscriptions, 1)
        _Subscription_Fees_Producer.AddSourceValue(_Subscription_Fee_Pcnt, 2)
        _Subscription_Fees.ProducingOperator = _Subscription_Fees_Producer
        _Redemption_Fees_Producer.AddSourceValue(_Redemption_Fee_Pcnt, 1)
        _Redemption_Fees_Producer.AddSourceValue(_Redemptions, 2)
        _Redemption_Fees.ProducingOperator = _Redemption_Fees_Producer
        _Fees_Producer.AddSourceValue(_Subscription_Fees, 1)
        _Fees_Producer.AddSourceValue(_Redemption_Fees, 2)
        _Fees.ProducingOperator = _Fees_Producer

        'Set the root values
        _rootValues.Add (_NAV_Per_Share)
end sub

#End Region

End Class
```

Liite 5: Näyttö, joka on luotu liitteen 2 esittämän mallin pohjalta