Risto Moilanen

SOVELLUSALUEMALLINNUS OHJELMISTOTUOTANNON TUKENA

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

28.11.2006

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Risto Moilanen

Yhteystiedot: rismoila@gmail.com

Työn nimi: Sovellusaluemallinnus ohjelmistotuotannon tukena

Title in English: Domain-Specific Modeling in Support of Software Production

Työ: Pro gradu -tutkielma

Sivumäärä: 77 + 5

Linja: Ohjelmistotekniikka.

Teettäjä: Jyväskylän yliopisto, Tietotekniikan laitos

Avainsanat: Sovellusaluemallinnus, sovellusaluekielet, Software Factories

Keywords: Domain-Specific Modeling, Domain-Specific Languages, Software Factories

Tiivistelmä: Tässä pro gradu – tutkielmassa tutkitaan sovellusaluemallinnuksen käyttöä ohjelmistokehityksen automatisoinnissa. Esimerkkikonseptina esitellään Microsoftin kehittämä Software Factories. Työn käytännön osuudessa kehitetään sovellusaluelähtöisen tietomallinnuksen mahdollistava sovellusaluekieli EML, tätä tukeva mallinnustyökalu sekä kehitettyjä malleja hyödyntäviä generaattoreita.

Abstract: This pro gradu thesis investigates the use of domain-specific modeling in automatization of software development. As an example concept, Software Factories by Microsoft is introduced. The empirical part of the study describes the development of a domain-specific data modeling language called EML. In addition, a modeling tool that supports EML, and generators that utilize the developed models, will be introduced.

Esipuhe

Kiitos SysOpen Digia Financial Software Oy:lle erinomaisesta tuesta graduprojektin aikana. Kiitokset Risto Airaksiselle sekä muille työkavereille antoisista keskusteluista gradun aiheisiin liittyen sekä avunannosta työn eri vaiheissa.

Kiitokset Vesa Lappalaiselle asiantuntevasta ohjauksesta sekä mielenkiinnosta työtäni kohtaan.

Erityinen kiitos vaimolleni Jennille sekä lapsilleni Viljamille ja Pinjalle kärsivällisyydestä ja tuesta niinä lukemattomina iltoina kun *isi oli vain koneella*.

Termiluettelo

.NET	Microsoftin kehittämä sovellusalusta erityisesti hajautettujen monikerrossovellusten kehittämiseen
C#	Microsoftin kehittämä .NET-ohjelmointikieli
CORBA	Common Object Request Broker Architecture OMG:n standardi hajautettujen olio-ohjelmistojen väliohjelmistoksi
DSL	Domain-Specific Language Sovellusaluekieli
DSL Tools	Microsoftin sovellusaluekielten kehitysympäristö
DSM	Domain-Specific Modeling Sovellusaluemallinnus
IIS	Internet Information Services Microsoftin www-palvelinohjelmisto
MDA	Model-Driven Architecture OMG:n malliperustainen ohjelmistonkehityskonsepti
OMG	Object Management Group Suurten yritysten muodostama konsortio, joka kehittää standardeja ohjelmistokehityksen tueksi
SOAP	Simple Object Access Protocol Etäproseduurikutsujen mahdollistava tietoliikenneprotokolla
SQL	Structured Query Language Yleinen tietokantojen kyselykieli

UML	Unified Modeling Language OMG:n standardoima, useita kaaviotyyppejä sisältävä mallinnuskieli
Visual Basic.NET	Microsoftin .NET-ohjelmointikieli
Win32 API	32-bittisen Windows-käyttöjärjestelmän ohjelmointirajapinta
WSDL	Web Services Description Language Web Services -palvelujen kuvauskieli
XML	Extensible Markup Language Metakieli, jolla voidaan määritellä toisia kieliä

Sisältö

1 Johdanto

Perinteisessä teollisuudessa on jo aikoja sitten siirrytty käsityöstä automaattiseen tuotantoon. Ohjelmistotuotanto on kuitenkin ala, jolla merkittävää tuottavuuden kasvua ei ole saavutettu oikeastaan sen jälkeen, kun siirryttiin assembler-tasoisesta ohjelmoinnista korkean tason ohjelmointikielten käyttöön. Käsityöläisestä luonteestaan johtuen ohjelmistotuotanto on erittäin paljon aikaa, rahaa ja resursseja kuluttavaa. Ohjelmistoteollisuudessa on myös yleistä, että ohjelmistoprojekti joko myöhästyy tai tulee suunniteltua kalliimmaksi. Keskimäärin vain joka kuudes ohjelmistoprojekti valmistuu aikataulussaan ja budjetissaan pysyen. Pelkästään Yhdysvalloissa kulutettiin vuonna 1995 81 miljardia dollaria keskeytettyihin ohjelmistoprojekteihin ja ohjelmistoprojektien budjetteja ylitettiin 59 miljardilla dollarilla [May98].

Vaikka ohjelmistoteollisuuden tuotteiden avulla voidaan automatisoida mitä erilaisimpia prosesseja tavaroiden valmistuksesta asiakaspalveluun on merkillepantavaa, että itse ohjelmistotuotannossa ei automatisointia kuitenkaan merkittävässä mittakaavassa esiinny. Tämä on hämmästyttävää myös siksi, että usein ohjelmistoyritykset toimivat tietyllä kapealla toimialalla ja tuottavat monia toistensa kanssa yhteisiä piirteitä sisältäviä ohjelmistoja, jotka käsittelevät toimialan käsitteitä sen sääntöjen puitteissa. Ohjelmistokehityksessä käytettävät geneeriset työkalut ja prosessit eivät useinkaan vastaa tällaisen yksittäisen sovellusalueen erityistarpeisiin ja tästä syystä ohjelmistot kehitetään usein ”tyhjästä” eikä merkittävää uudelleenkäyttöä esiinny.

Ohjelmistoteollisuudessa on eräitä merkittäviä eroja verrattuna perinteiseen teollisuuteen ja nämä osaltaan selittävät tuottavuusongelmaa ja ohjelmistojen kehittämisen haastavuutta. Tärkeimmät ohjelmistoteollisuuden haasteet liittyvät siihen, että ohjelmistot ovat usein erittäin monimutkaisia, luonteeltaan uniikkeja ja kaiken lisäksi jatkuvan muutokset kohteina [Gre04a]. Ohjelmistojen vaatimukset ja toimintaympäristöt muuttuvat varsinkin toteutusaikana sekä usein vielä toimituksen jälkeen ja ohjelmistojen tulee voida sopeutua näihin muutoksiin. Ohjelmistojen ylläpidon arvellaan aiheuttavan 50–75 prosenttia kaikista ohjelmistokehityksen kustannuksista [Kos04].

Eräs tärkeä keino abstrahoida toteutuksen yksityiskohtia, automatisoida ohjelmistokehityksen rutiininomaisimpia toimintoja ja täten auttaa sovelluksen monimutkaisuuden hallinnassa on mallintaa sovelluksen toimintaa sovellusalueen käsittein. *Sovellusaluemallinnuksella* tarkoitetaan jonkin *sovellusalueen* keskeisten käsitteiden ja toimintojen mallintamista sekä kehitettyjen mallien hyödyntämistä esimerkiksi ohjelmakoodin generoinnissa. Sovellusaluemallinnukseen liittyy olennaisesti *sovellusaluekielten* (engl. *Domain-Specific Languages*) kehittäminen ja käyttäminen tähän tarkoitukseen kehitetyillä työkaluilla. Esimerkkejä sovellusaluekielten kehittämiseen tarkoitetuista työkaluista ovat MetaEdit+[1] ja Microsoft DSL Tools[2]. Sovellusaluekielen avulla voidaan kuvata järjestelmää sovellusalueen näkökulmasta formaalisti. Sovellusaluekielet voivat olla joko vertikaalisia, tietylle toimialalle spesifioituja tai horisontaalisia, usealle toimialalle soveltuvia. Esimerkki vertikaalisesta sovellusaluekielestä voisi olla esimerkiksi rahoitusalan liiketoimintaprosessien kuvaamiseksi kehitetty kieli ja horisontaalisesta taas vaikkapa SQL.

Sovellusaluekielten formaalius tarjoaa sen oleellisen hyödyn kehitysprosessille, että sovellusaluekielillä esitetyt mallit ovat koneellisesti käsiteltävissä. Sovellusaluemalleja voidaankin käyttää generoimaan esimerkiksi ajettavaa ohjelmakoodia, yksikkötestejä, tietokantaskriptejä, konfiguraatiotiedostoja, simulaatiodataa, HTML-sivuja, dokumentaatiota sekä erilaisia raportteja mallin perusteella [Tol06b]. Parhaimmillaan mallin perusteella on mahdollista generoida suoraan täydellisesti ajettava ohjelma. Suurin hyöty sovellusaluemallinnuksesta saadaankin silloin, kun sovellusaluemallia voidaan käsitellä korkean tason toteutuksena [Ise05].

Sovellusaluemallinnuksen käyttöönottaminen sekä sovellusaluekielten luominen vaativat sen, että kehittäjäorganisaatiolla on jo ennestään riittävä ymmärrys sovelluksen kohdealueesta. Tämän lisäksi sovellusaluekieliä hyödyntävien generaattoreiden tehokas toteuttaminen vaatii tuekseen kohdekehyksen, joka voi olla syntynyt esimerkiksi aiemmin samalle

[1] http://www.metacase.com/mep/

[2] http://msdn.microsoft.com/vstudio/dsltools/

kohdealueelle tehtyjen sovellusten kehittämisen myötä [Poh02] (ks. luku 3.4). Näin ollen sovellusaluemallinnusta ei nähdäkään hopealuotina kaikkeen sovelluskehitykseen, vaan ensisijaisesti tapana tehostaa jo aiemmin kohdealueella kokemusta saaneen kehittäjäorganisaation tuotekehitystä.

Microsoftin kehittämässä *Software Factories* -konseptissa on sovellusaluemallinnuksella keskeinen rooli. Termillä *ohjelmistotehdas* tarkoitetaan tässä tapauksessa tietyn tuoteperheen sovellusten toteuttamiseksi kehitettyä tuotelinjaa, jonka avulla tuoteperheeseen voidaan toteuttaa uusia osia mahdollisimman helposti. Konseptissa pyritään parantamaan tuotekehityksen välineitä, työtapoja sekä prosesseja muun muassa systemaattisen uudelleenkäytön, tuotantolinjamaisen ja malliperustaisen kehityksen sekä prosessikehysten avulla. Software Factories -konseptin keskeisimpänä ajatuksena on käyttää sovellusaluekielillä kehitettyjä malleja koko kehityksen lähtökohtana [Gre04a].

Kuten luvussa 3.6 havaitaan, on sovellusaluekielten kehittäminen usein iteratiivista, ts. kehitettävät sovellusaluekielet ja kohdekehykset vaativat muokkausta ja hiomista koko kehitysprosessin ajan. Sovellusaluemallinnukseen perustuvissa konsepteissa kehittäjät on ikään kuin jaettu kahteen rooliin: sovellusaluekielten ja työkalujen kehittäjät sekä varsinaiset sovelluskehittäjät. Luvussa 4.1 on esitelty Software Factories -konseptin systemaattisen uudelleenkäytön periaate, joka myös perustuu kehittäjien kahtiajakoon edellä mainitulla tavalla.

1.1 Tutkimusongelma

Tässä tutkielmassa tutkitaan sovellusaluemallinnuksen ja sovellusaluekielten käyttöä ohjelmistotuotannon automatisoinnin tukena erityisesti hajautetun kolmikerrosarkkitehtuurilla toteutetun Enterprise-tason sovellustuoteperheen sovelluksien kehittämisessä. Koska sovellustuoteperheen sovellukset ovat toteutettu Microsoft .NET- teknologialla Visual Studio-ohjelmistonkehitystyökalua käyttäen, on esimerkiksi sovellusaluemallinnusta hyödyntävästä konseptista valittu Microsoftin kehittämä Software Factories ja toteutustyökaluksi Microsoft DSL Tools.

Teoriaosuudessa ensisijainen tavoite on esitellä sovellusaluemallinnuksen ja sovellusaluekielten yleisiä periaatteita ja vaatimuksia mitä se sovelluskehitykselle asettaa. Lisäksi esitellään sovellusaluemallien muuntaminen ohjelmakoodiksi ja muiksi artefakteiksi tähän tarkoitukseen kehitettyjen generaattoreiden avulla.

Tutkielman käytännön osuuden tavoitteena on kehittää sovellusaluekieli ja tätä tukeva mallinnustyökalu, jolla voidaan mallintaa rahoitusalalle toteutettavien laajojen ohjelmistojen tietomallia tarkemmalla tasolla kuin perinteistä ER-mallinnusta käyttäen. Mallinnustyökalun lisäksi kehitetään generaattoreita, jotka generoivat sovellusaluemallien perusteella erilaisia artefakteja, kuten ohjelmakoodia ja SQL-skriptejä. Työkalun tarkoituksena on parantaa sovelluskehityksen tuottavuutta ja laatua rahoitusalalle toteutettavien sovellusten kehittämisessä. Tässä tutkielmassa ei integroida työkalua varsinaiseen sovelluskehitysympäristöön. Työkalu on luonteeltaan prototyypinomainen ja tämän tutkielman on tarkoitus arvioida alustavasti, minkälaisin edellytyksin sitä voitaisiin hyväksikäyttää ohjelmistojen kehityksessä.

1.2 Tutkielman rakenne

Luvussa 2 esitellään ohjelmistotuotannon yleisesti havaittuja haasteita ja ongelmia, joihin sovellusaluemallinnus yrittää antaa vastauksen.

Luvussa 3 esitellään yleisesti sovellusaluemallinnuksen sekä sovellusaluekielten periaatteita, sekä vaatimuksia mitä sovellusaluemallinnuksen käyttöönotto kehitysprosessille asettaa.

Luvussa 4 esitellään Microsoftin sovellusaluemallinnukseen vahvasti nojautuva konsepti, Software Factories.

Luvussa 5 käsitellään ohjelmakoodin ja muiden tuotosten generointia sovellusaluemallien perusteella.

Luvussa 6 kuvataan tutkielman käytännön osuudessa hyödynnettävä kolmitasoarkkitehtuurin mukainen sovellusalusta.

Luvussa 7 esitellään tutkielman käytännön osuudessa kehitetty Entity Modeling Language-sovellusaluekieli, jota voidaan käyttää sovellusten tietomallin mallintamiseen rahoitusalan erityispiirteet huomioiden. Luvussa esitellään myös Visual Studioon integroitu työkalu, jonka avulla tietomalleja voidaan kehittää.

2 Taustaa

Ohjelmistokehitys ei ole vielä niin kypsässä tilassa, jotta se pystyisi vastaamaan sille asetettuihin vaatimuksiin tarpeeksi hyvin. Ohjelmistoprojektit eivät valmistu useinkaan asetetun aikataulun puitteissa ja eikä toteutus läheskään aina vastaa asiakkaiden vaatimuksiin. Lisäksi muun muassa virheiden määrä ja yhteensopimattomat konfiguraatiot aiheuttavat suuria käytännön ongelmia.

Software Productivity Research-tutkimusorganisaation mukaan keskimääräinen tuottavuus Java-kehityksessä on vain noin 20 prosenttia parempi kuin BASIC-ohjelmointikieltä käytettäessä. Tutkimusten mukaan Smalltalkia lukuun ottamatta mikään yleisessä käytössä oleva ohjelmointikieli ei kykene oleellisesti parempaan tuottavuuteen kuin BASIC [Kel05].

Myöskään olio-ohjelmoinnin menetelmät eivät ole pystyneet siihen mitä lupasivat, eli tuottamaan olioita, joita voitaisiin uudelleenkäyttää suuressa mittakaavassa laajemminkin. Tämän vuoksi tavallisesti ohjelmistot rakennetaan vieläkin lähes tyhjästä. Käytännössä tämä näkyy muun muassa siinä, että ei ole syntynyt kaupallisesti menestyneitä komponenttimarkkinoita lukuun ottamatta käyttöliittymäkomponentteja (kontrolleja).

2.1 Ohjelmistonkehityksen haasteet

Ohjelmistonkehityksen suurimmat erot verrattuna perinteiseen teollisuuteen ovat ohjelmistojen monimutkaisuus sekä jatkuvat muutokset.

2.1.1 Monimutkaisuus

Ohjelmistoteollisuus eroaa muista teollisuudenaloista sillä, että sen tuotteet ovat lähes poikkeuksetta erittäin monimutkaisia. Ohjelmiston monimutkaisuus voi johtua suoraan ratkaistavan ongelman monimutkaisuudesta tai se voi olla seurausta toteutetusta ratkaisusta. Ohjelmistoon päätyy useimmiten enemmän ominaisuuksia ja varioitavuutta, kuin mitä minimissään ongelman ratkaisemiseksi vaadittaisiin ja tämä on osaltaan lisäämässä ohjelmiston monimutkaisuutta. Ratkaistavan ongelman monimutkaisuus päätyy aina vähintään sellaisenaan kehitettävään ohjelmistoon ja sen voidaan tästä syystä katsoa olevan *välttämätöntä* [Gre04a].

Toteutuksesta johtuva monimutkaisuus riippuu muun muassa toteutusteknologiasta sekä ohjelmistolle asetetuista ei-toiminnallisista vaatimuksista ja on täten tahatonta verrattuna ongelmasta johtuvaan monimutkaisuuteen. Ohjelmisto sisältää välttämättä toteutukseen liittyvää monimutkaisuutta, mutta sen määrä ei välttämättä ole riippuvainen ratkaistavan ongelman luonteesta.

Varhaiset oliosuuntautuneet tutkijat esittivät, että jokainen ongelma-alueen olio voitaisiin esittää yhtenä oliona myös toteutuksessa [Cop00]. Tämä ei kuitenkaan ole käytännössä mahdollista kuin aivan yksinkertaisimmissa sovelluksissa. Jos ajatellaan esimerkiksi oliota *Tili* ja siihen pysyvyystoiminnallisuutta perinteisessä Web Services -tiedonvälitystä hyödyntävässä monikerrossovelluksessa, sen huomataan jakautuneen varsin monen olion kesken:

1. Asiakassovelluksen hakunäyttö
2. Asiakassovelluksen muokkausnäyttö,
3. Asiakassovelluksen liiketoimintaolio[3] sekä sen kokoelmaolio
4. Web Services -serialisaatiosta vastaava olio
5. Palvelinsovelluksen liiketoimintaolio sekä sen kokoelmaolio

Edellä mainitun kaltaista ilmiötä kutsutaan *ominaisuuksien hajautumiseksi* [Gre04a]. Ominaisuuksien hajautuminen johtaa kahden ongelman syntymiseen: *Jäljitettävyysongelma* ja *uudelleenkonstruointiongelma*.

Jäljittävyyden häviäminen tulee ilmi silloin, kun ohjelmiston vaatimukset muuttuvat. Enää ei tiedetä tarkkaan mihin kaikkialle tulee tehdä muutoksia toiminnallisuuden muuttuessa ja on lisäksi työlästä varmistaa, että kaikki muutokset tulevat oikein tehtyä. Uudelleenkonst-

[3] Liiketoimintaolio on oliosanaston (http://www.cs.helsinki.fi/u/laine/oliosanasto) mukaan ”Olio, joka kuvaa liiketoiminnan kohdetta tai osapuolta.”

ruointiongelma nousee esiin esimerkiksi silloin, kun ohjelmistoa ryhtyy ylläpitämään henkilö, joka ei tiedä ohjelmiston suunnitteluperiaatteita. Enää ei yksinkertaisesti tiedetä miten ohjelmistoa tulisi muuttaa rikkomatta sitä täysin eikä ylläpitäjällä ole käsitystä esimerkiksi siitä, milloin hän voi poistaa kokonaan käyttämättömiä sovelluksen vanhentuneita osia.

2.1.2 Jatkuvat muutokset

Foote ja Yoder jakavat ohjelmistot kolmeen osaan sen perusteella, millä tavoin ne suhtautuvat muutokseen [Foo96]:

1. Jotkut ohjelmistot kertakäyttöisiä, sillä ne ovat yksinkertaisia ja halpoja kehittää. Kun tällaiseen ohjelmistoon vaaditaan muutosta, on helpompaa ja halvempaa hylätä se ja rakentaa kokonaan uusi sen sijaan että olemassa olevaa ohjelmistoa muutettaisiin.

2. Jotkut ohjelmistot (esimerkiksi niin sanotut perinnejärjestelmät) ovat niin kalliita hylätä ja muokata, että ne ovat käytännössä muuttumattomia. Niiden käyttämä teknologia saattaa olla esimerkiksi niin vanhaa, että on järkevämpää käyttää järjestelmää sellaisenaan niin pitkään kuin mahdollista ja yrittää vastata muuttuviin vaatimuksiin jollain muulla tavalla.

3. Ohjelmistot, joille on taloudellisesti mielekästä tehdä muutoksia, ovat muuttuvia. Nämä ohjelmistot kokevat elinkaarensa aikana *evoluutiota*.

Koska suurin osa ohjelmistoista kuuluu yllämainittuun kolmanteen ryhmään, on ohjelmistojen evoluutio tärkeätä ottaa huomioon niiden kehityksessä. Ohjelmistoille asetetut vaatimukset muuttuvat lähes poikkeuksetta ohjelmiston kehitysaikana sekä sen jälkeen, jolloin niihin täytyy tehdä muutoksia. Muutokset useimmiten heikentävät ohjelmiston laatua sitä enemmän mitä isommasta ja monimutkaisemmasta järjestelmästä on kyse. Ohjelmiston laadun heikkeneminen voi johtua esimerkiksi sellaisesta ohjelmiston muutoksesta, jossa muutoksen tekijä ei ymmärrä alkuperäistä suunnitelmaa ja ohjelmiston rakennetta.

Ohjelmiston evoluutio saa aikaan neljänlaisia ongelmia: Ohjelmiston *pysähtyneisyys*, ohjelmiston *väsyminen*, ohjelmiston *hauraus* ja ohjelmiston *redundanttisuus* [Par94]

[Gre04a]. Pysähtyneisyys ilmenee sillä, että ohjelmistoon ei ole enää järkevää tehdä muutoksia suuriksi kohoavien kustannusten tähden. Väsyminen taas on ikään kuin ohjelmiston laatua heikentävä eroosio, jota ohjelmisto kohtaa elinkaarensa aikana. Väsyminen johtuu siitä, että ohjelmistoon tehdään muutoksia, jotka ovat vastoin sen alkuperäisiä suunnitteluperiaatteita. Hauraus on ikään kuin mittari sille, kuinka paljon asiakaskomponentteja joudutaan muuttamaan, mikäli yksittäiseen komponenttiin tehdään sisäinen muutos. Ohjelmiston hauraus kasvaa normaalisti sen elinkaaren aikana. Ohjelmistojen redundanttisuudesta voisi olla hyvä analogia monien yritysten tapa ylläpitää useita asiakastietojärjestelmiä, kukin hieman eri tarkoituksiin ja hieman erilaisilla yksityiskohdilla, mutta kuitenkin suurimmaksi osaksi samaa dataa sisältäen. Redundanttisuudesta aiheutuu ylimääräistä työtä ja epävarmuustekijöitä ohjelmiston ylläpitoon.

Muutosten syitä on sekä ongelma-avaruudessa tapahtuvat muutokset, että toteutusavaruudessa tapahtuvat muutokset. Ongelma-alueen muutokset voivat johtua joko käyttäjästä ja sen vaatimusten muuttumisesta tai ohjelmiston käyttäjäorganisaation liiketoimintaprosessien muutoksista. Toteutusavaruuden muutokset voivat liittyä esimerkiksi toteutusteknologiassa tapahtuviin muutoksiin.

2.2 Ohjelmistokehityksen krooniset ongelmat

Syitä ohjelmistoprojektien epäonnistumiseen on tutkittu paljon. Erinomaisia lähteitä ovat muiden muassa [May98] ja [Bro95]. Software Factories -konseptin kehittäjien mukaan ohjelmistotuotannossa on havaittavissa erityisesti seuraavat yksilöitävät ongelmat joihin se pyrkii vastaamaan: Monoliittinen konstruointi, Liiallinen yleistäminen, ohjelmistojen uniikki kehittäminen ja prosessien kypsymättömyys.

2.2.1 Monoliittinen konstruointi

Ohjelmistoalan pioneerien tavoitteena on jo vuosikymmeniä ollut saattaa käytännöt ja prosessit sellaisiksi, että ohjelmia voitaisiin tehdä pelkästään olemassa olevia komponentteja kokoonpanemalla. Erityisesti olio-ohjelmointi sekä komponenttipohjainen kehitys ovat pyrkineet tähän kuitenkin siinä onnistumatta. Software Factories -konseptin kehittäjien mukaan tähän on johtanut muun muassa seuraavat syyt:

- Kommunikointiprotokollien tiukka sitominen komponenttien toteutusteknologiaan. Tästä on esimerkkinä Java RMI ja .NET Remoting, joita on erittäin hankala saada kommunikoimaan keskenään.

- Heikot komponenttien määrittely- ja koontiteknologiat, jotka asettavat oletuksia esimerkiksi komponenttien käyttäjän arkkitehtuurista [Gar95b].

- Ohjelmistoteollisuuden tuottaja-kuluttaja-suhteet eivät ole riittävän kypsiä.

- *Not Invented Here* - Syndrooma: Turhan usein päätetään kehittää tiettyyn tarkoitukseen sopiva komponentti itse, vaikka uudelleenkäytettäviä komponentteja olisikin saatavilla. On havaittu, että pitkällä tähtäimellä on edullisempaa uudelleenkäyttää hieman puutteitakin sisältäviä olemassa olevia komponentteja kuin kehittää juuri sopivat komponentit täysin tyhjästä.

2.2.2 Liiallinen yleistäminen

Nykyiset ohjelmistokehityksen menetelmät ja käytännöt tarjoavat kehittäjille monesti enemmän vapautta, kuin mitä on tarpeen. Jos ajatellaan esimerkiksi liiketoimintaa tukevia sovelluksia, huomataan, että ne koostuvat usein muutamista perusmalleista: Ne lukevat tietoa tietokannasta, käsittelevät sitä sovellusalueelle kuuluvin säännöin, näyttävät sen käyttäjälle, antavat käyttäjän muokata tietoa tiettyjen sääntöjen puitteissa ja viimein tallentavat tiedot takaisin tietokantaan. Tämä on luonnollisesti yksinkertaistus, useinhan järjestelmiin liittyy myös muita vaatimuksia, kuten esimerkiksi liittymistä perinnejärjestelmiin, normaalista poikkeavia käyttöliittymiä kuten mobiilikäyttöliittymiä, suuria datamääriä, monia yhtäaikaisia käyttäjiä tai vasteaikavaatimuksia ja näin ollen edellä mainittujen perusmallien sovittaminen saattaa olla haastavaa. Kuitenkin, vaikka jokaisessa projektissa on omat uniikit yksityiskohtansa, on työssä usein havaittavissa samankaltaisia piirteitä projektista toiseen. Täten menetelmien, käytäntöjen ja työkalujen olisi järkevää tarjota kehittäjille tukea näiden yhteisten piirteiden osalta.

2.2.3 Uniikit ohjelmistoprojektit

Ohjelmistoteollisuudessa ei olla onnistuttu hyödyntämään uudelleenkäytön tarjoamia mahdollisuuksia. Suurin osa ohjelmistoista kehitetään yksittäisinä ohjelmistoina eristyksissä muista ohjelmistoista, vaikka ohjelmistoissa olisikin enemmän yhtäläisyyksiä kuin eroavaisuuksia. Tuottavuus on luonnollisesti parempi kuin saman tuotteen eri variaatiot otetaan huomioon jo ennen kehitystyöhön ryhtymistä. Harvemmin kuitenkaan tehdään merkittäviä investointeja sen edistämiseksi, että uudelleenkäytettäviä yksiköitä tunnistettaisiin, hiottaisin ja pakattaisiin uudelleenkäytettävään muotoon. Esiintyvä uudelleenkäyttö on ennemminkin satunnaista kuin systemaattista. Tämä onkin luonnollista, sillä uudelleenkäyttöä harvemmin esiintyy, ellei sitä olla varta vasten suunniteltu etukäteen.

2.2.4 Prosessien kypsymättömyys

Ohjelmistojen tekeminen budjetissa ja aikataulussa pysyen on harvinaista, vain joka kuudes ohjelmistoprojekti valmistuu aikataulussaan ja budjetissaan pysyen [May98]. Tämä kertoo siitä, että ohjelmistonkehitysprosessit ovat epäkypsiä.

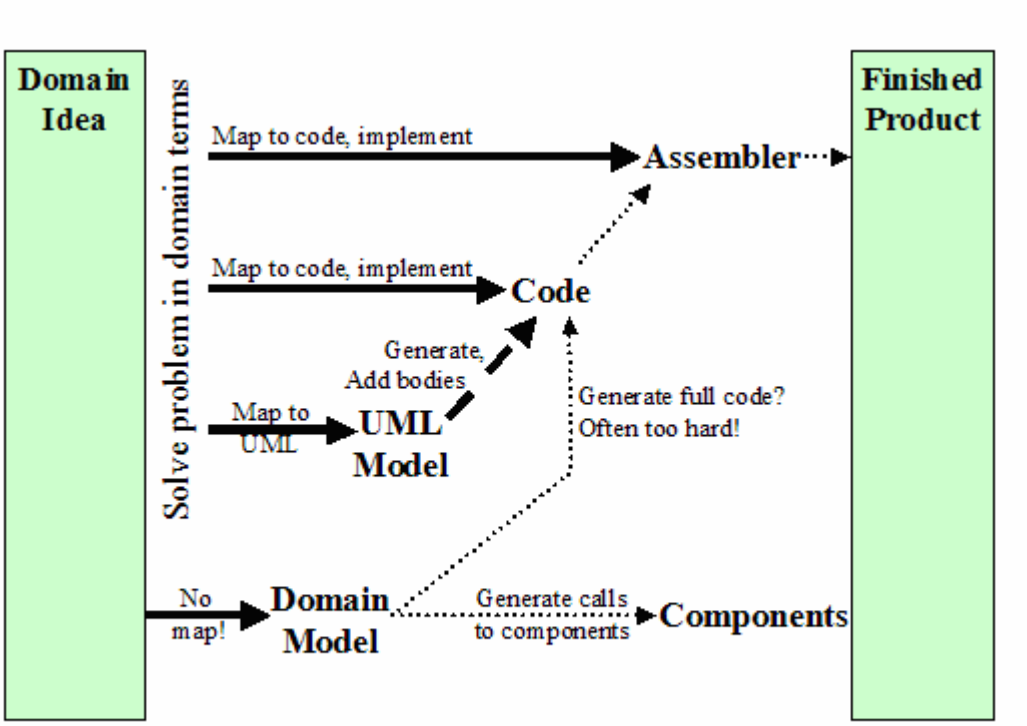
Hyvä kuvaus ohjelmistoprosessien epäkypsyydestä on Fred Brooksin[4] kirjassa *The Mythical Man-Month* [Bro95]. Brooksin mukaan yksi pahimmista virheistä myöhässä olevalle ohjelmistoprojektille on lisätä siihen uutta henkilöstöä sillä ajatuksella, että aikataulua saataisiin kurottua täten umpeen. Tästä Brooks esittää analogian lapsen odottamiseen: vaikka raskaana olevan naisen tukena olisi kuinka monta ihmistä, nähdään lopputulos kuitenkin vasta yhdeksän kuukauden kuluttua. Projektin estimoiminen ja kurinalainen seuraaminen alusta asti on ainut keino välttyä ikäviltä yllätyksiltä. Mielenkiintoista on lisäksi se, että Brooks kehottaa käyttämään erikoistuneita työkaluja, kuten koodigeneraattoreita aina kun mahdollista [Bro95].

[4] Kirjan kirjoittaja on tullut tutuksi muun muassa lanseeraamastaan käsitteestä *No Silver Bullet* samannimisessä esseessään.

Ohjelmistonkehitysprosessimallit voidaan jakaa karkeasti ottaen kahteen ryhmään: On muodollisia prosessimalleja, jotka keskittyvät monimutkaisuuden hallintaan ollen samalla kankeita muutokselle ja epämuodollisia, jotka keskittyvät muutoksen hallintaan, jolloin kokonaisuudesta voi helposti tulla vaikeasti ylläpidettävä. Kypsä ohjelmistoprosessi on ehdoton edellytys ohjelmistotuotannon automatisoinnille, koska työkalut eivät luonnollisesti voi automatisoida sitä mistä ihmisetkään eivät suoriudu hyvin. Jopa tärkeämpää kuin mikä prosessimalli valitaan, on se, että valittua prosessimallia käytetään systemaattisesti ja johdonmukaisesti.

3 Sovellusaluemallinnus

Ohjelmiston ongelma- ja toteutusavaruudet ovat kaksi aivan eri maailmaa. Molemmissa on omat asiantuntijansa, kielensä ja tapansa ratkoa ongelmia [Tol00a]. Perinteisesti lopullinen ohjelmisto on aina näiden kahden maailman, ongelma-avaruuden ja ohjelmakoodin leikkaus ja ohjelmistokehittäjien haastava työ on rakentaa silta näiden kahden maailman välille.



Kuva 1, Sovellusalueen ideasta tuotteeksi [Tol00a]

Kuva 1 esittää neljä tapaa, jolla ohjelmistokehittäjä voi rakentaa sillan ongelma- ja toteutusavaruuden välille [Tol00a]. Kahdessa ylimmässä tavassa ohjelmistosuunnittelija ratkaisee ongelman ensin ongelma-avaruuden käsittein ja sen jälkeen siirtää sen toteutusavaruuteen rakentamalla toteutuksen. Siirryttäessä Assembler-ohjelmoinnista ensimmäisten korkean tason ohjelmointikielten käyttöön, saavutettiin merkittävä abstraktiotason kasvu ja tuottavuus nousi noin 400 prosenttia [Kel05]. Näin ollen kääntäjätekniikan kehittyminen sekä korkean tason ohjelmointikielten käyttöönotto pienensi idea-tuote-kuilua kuvassa oikealta käsin, mutta silti yleiskäyttöisiä ohjelmointikieliä käytettäessä ongelma täytyy rat-

kaista kaksi kertaa. UML:n ja muiden mallinnuskielten käyttöönotto ei juuri helpottanut tätä tilannetta: ongelma täytyy yhä ratkaista ensin ongelma-avaruudessa ilman työkalutukea ja sen jälkeen muuntaa se UML-kielen käsitteiksi, josta voidaan generoida pieni osa ohjelmakoodia. Ohjelmistokehittäjän tulee sen jälkeen kirjoittaa varsinaisen logiikan toteuttava ohjelmakoodi. Kuvan kolmatta nuolta seuraamalla täytyy ongelma ratkaista oikeastaan kolme kertaa: Ensin ongelma-avaruudessa (usein ohjelmoijan omassa päässä!), sen jälkeen muunnettaessa ratkaisu UML-malliksi ja viimeisenä toteutusvaiheessa kirjoitettaessa luokkarunkojen sisään varsinainen sovelluslogiikan toteuttava ohjelmakoodi.

Sovellusaluemallinnuksella tarkoitetaan ohjelmiston toiminnallisuuden mallintamista suoraan sovelluksen ongelma-avaruuden käsittein tarkoituksenmukaisia työkaluja käyttäen. Sen sijaan, että ohjelmisto jouduttaisiin mallintamaan samalla abstraktiotasolla, kuin millä se tullaan aikanaan toteuttamaan, nostetaan mallinnuksen abstraktiotasoa lähemmäs ohjelmiston sovellusaluetta ja näin vältetään virhealttiit muunnokset kuvauksesta toiseen sekä kaksin- tai jopa kolminkertainen työ ongelman ratkaisussa[5]. Täten sovellusaluemallinnus eroaa oleellisesti esimerkiksi UML:llä mallintamisesta, siinähän käsiteltävät asiat kuten oliot ja luokat sekä näiden väliset suhteet ovat abstraktiotasoltaan varsin lähellä toteutusavaruutta [Tol00a]. Kuva 1 esittää sovellusaluemallinnusta hyödyntävän menetelmän alimmalla rivillään.

Ohjelmistokehityksen kannalta ongelmia aiheuttaa aina tilanne, jossa samansisältöistä informaatiota joudutaan säilömään useassa paikassa. Näinhän on esimerkiksi UML-mallinnusta käytettäessä. Sama, käsin ylläpidettävä informaatio on sekä UML-malleissa, että ohjelmakoodissa. Redundanttisuus aiheuttaa ennen pitkää toissijaisen mallin jälkeenjäämisen.

Booch ja kumppanit ovat sanoneet MDA Manifestissaan [Boo04]: ”*Täysi teho sovellusaluelähtöisestä ohjelmistokehityksestä saadaan vain silloin, kun mallinnettavat konsep-*

[5] Tämä on ollut tavoitteena myös tämän tutkielman puitteissa kehitettävän Entity Modeling Language-sovellusaluekielen kehittämisessä (ks. luku 7).

tit kuvautuvat suoraan sovellusalueen konsepteiksi, sen sijaan, että niillä esitettäisiin toteutusavaruuden, ohjelmointitekniikkaan kuuluvia konsepteja." Esimerkiksi mallinnettaessa matkapuhelinten toimintaa, olisi hyvä että voitaisiin mallintaa suoraan käsitteillä kuten "Navi-näppäin", "tekstiviesti" ja "soittoääni".

Sovellusaluemallia voidaan parhaissa tapauksissa pitää eräänlaisena korkean tason toteutuksena. Tällöin sovelluksen toiminnallisuus mallinnetaan vain kerran ja suoraan sovellusalueen käsittein. Esimerkiksi UML-mallinnusta hyödyntävässä ohjelmistokehityksessä sovellus joudutaan ikään kuin toteuttamaan useaan otteeseen (ongelman ratkaisu-> UML-mallit -> ohjelmakoodi). Jokainen vaihe käsittää suuren määrän käsin tehtävää työtä ja tämä heikentää tuottavuutta sekä lopputuotteen laatua.

Ideaalitilanteessa lopulliset toteutukset generoidaan suoraan korkean tason sovellusaluemallien perusteella. On raportoitu jopa kymmenkertaisesta tuottavuuden kasvusta käytettäessä sovellusaluemallinnusta verrattuna perinteiseen ohjelmistonkehitykseen [Kel05] [Met05a]. Näin merkittävä tuottavuuden kasvu on mahdollista sen ansiosta, että sovellusaluemallinnukseen perustuvat menetelmät rakennetaan aina suoraan kyseiselle sovellusalueelle ja usein suoraan juuri kyseisen yrityksen tarpeisiin.

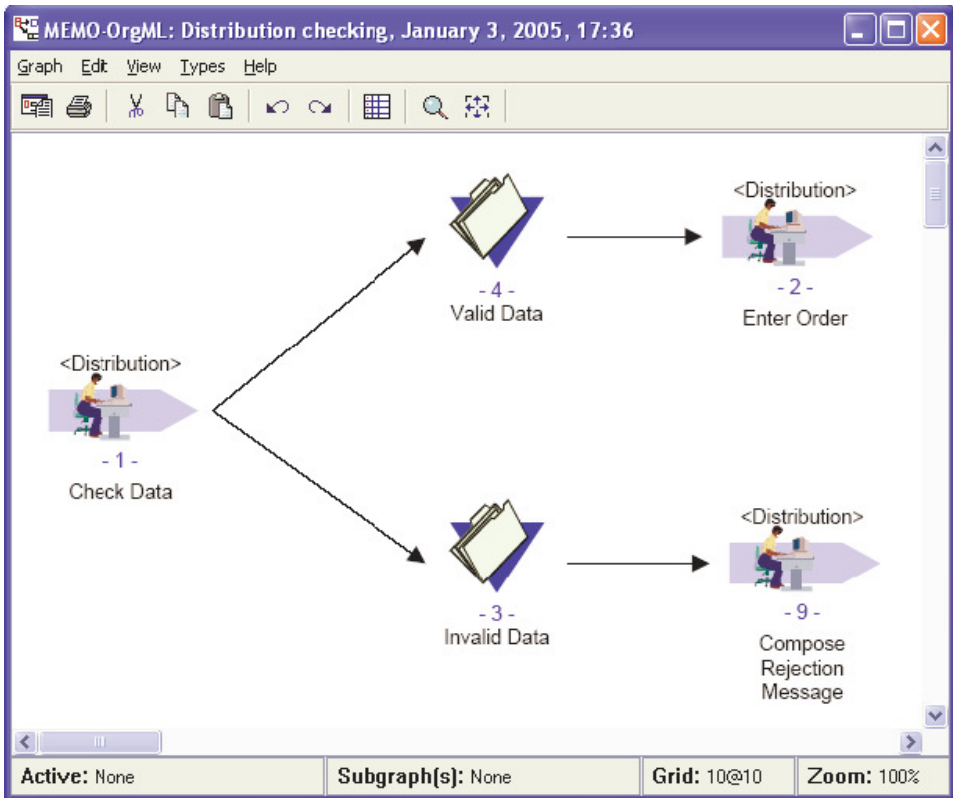
3.1 Visuaalinen mallinnus

Johtuen ohjelmistotuotannon nykyisestä käsityöläisestä luonteesta suurin osa abstraktiotasoltaan ohjelmakoodin yläpuolella olevasta informaatiosta katoaa ohjelmiston kehityksen aikana (ks. luku 2.1.1 ja *ominaisuuksien hajautuminen*). Tämä jopa siitä huolimatta, että se kertoisi parhaiten ohjelmistosta ja siitä miksi se on rakennettu juuri senkaltaiseksi kuin on [Gre04a]. Kun ohjelmoija kirjoittaa koodia, hänellä on päässään jonkinlainen aie siitä, minkälainen ohjelmiston tulisi olla ja miten sen tulisi toimia. Perinteinen ohjelmointi on tämän aikeen kuvausta jotain yleiskäyttöistä ohjelmointikieltä käyttäen suoraan toteutukseksi. Kirjoitettu ohjelmakoodi ei kuitenkaan esitä ohjelmoijan mielessä ollutta aietta sellaisessa muodossa, että sen avulla voitaisiin kommunikoida ongelma-avaruuden asioista muiden ihmisten kanssa.

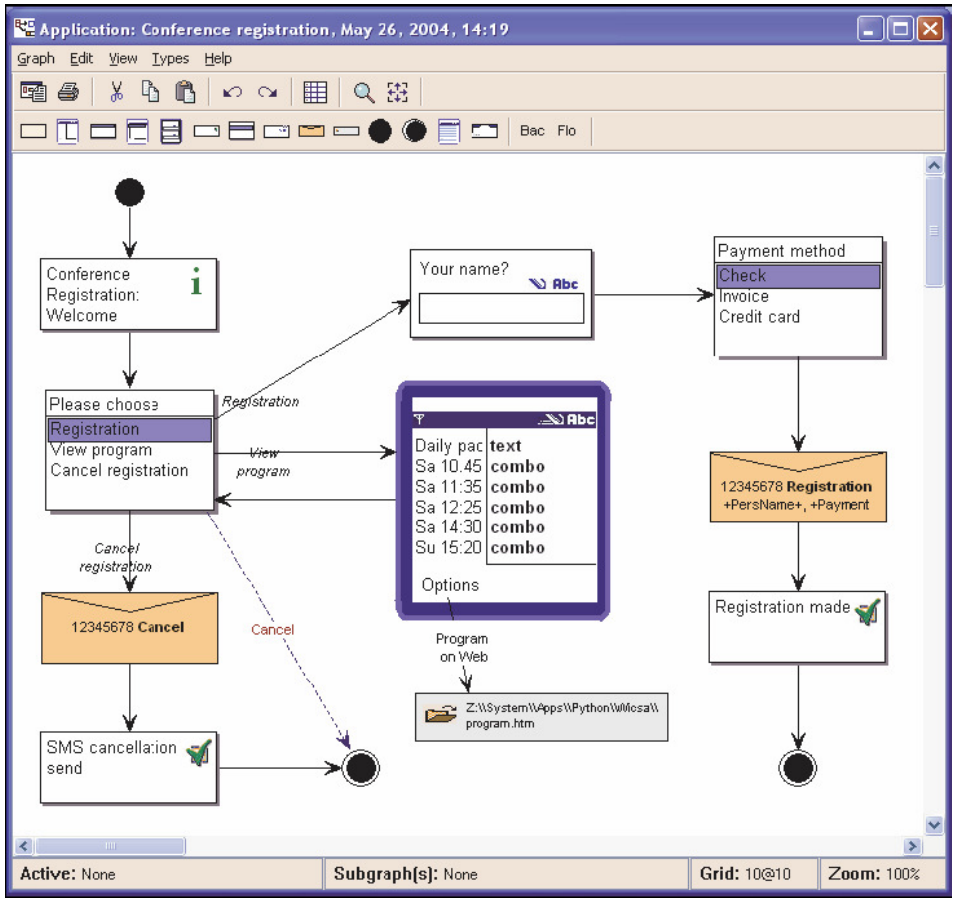
Dokumentaatio, jota kirjoitetaan ohjelmiston suunnittelun myötä, tulee useimmiten vanhentumaan ohjelmiston elinkaaren aikana, koska sen ylläpitokustannukset ylittävät dokumentaatiosta saatavan hyödyn. Useimmiten ohjelmistokehityksessä käy niin, että ainoa ajan tasalla oleva suunnitteluinformaatio on huonosti kommentoitu ohjelmakoodi.

Sovellusaluemallinnukseen perustuvat ohjelmistonkehitysmenetelmät, kuten luvussa 4 esiteltävä Software Factories -konsepti lähestyvät ongelmaa käyttämällä ohjelmistokehityksen tukena työkaluin käsiteltävissä olevia visuaalisia mallinnuskieliä. Näillä tehtyjä sovellusaluemalleja pyritään nostamaan samaan asemaan kuin ohjelmakoodi on tällä hetkellä, kehityksen keskeisimmiksi artefakteiksi [Gre04a]. Täysi hyöty visuaalisista mallinnuskielistä saadaan, silloin kun ne ovat juuri tietylle sovellusalueelle tarkoitettuja, sen sijaan että tarjoaisivat vain syntaktista sokeria ohjelmakoodin kuvaamiseksi tai käsin ylläpidettävän tavan dokumentoida muualla tehtyjä ratkaisuja.

Kuvissa 2 ja 3 on kaksi esimerkkiä visuaalisista mallinnuskielistä [Met05a]:



Kuva 2, Liiketoimintaprosessia kuvaava visuaalinen mallinnuskieli [Met05a]



Kuva 3, Konferenssiin rekisteröitymistä kuvaava mallinnuskieli [Met05a]

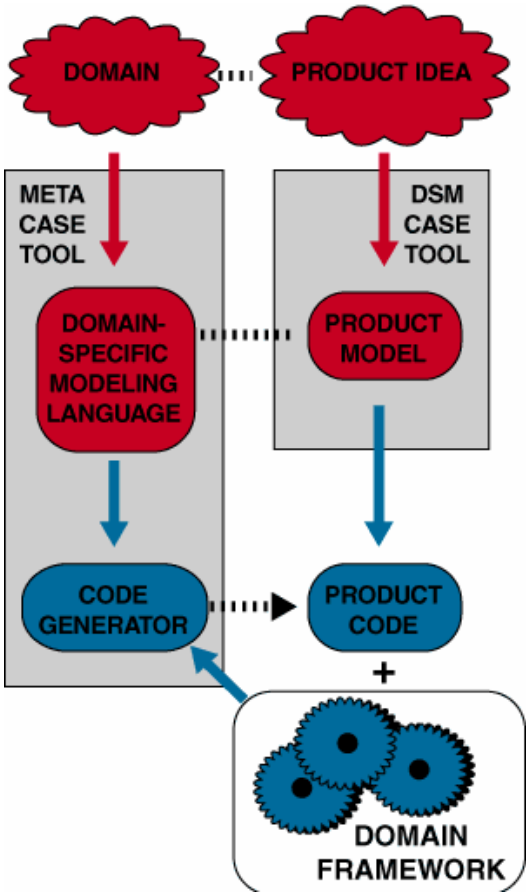
3.2 Sovellusaluemallinnusta tukeva ympäristö

Sovellusaluemallinnusta tukevaan ympäristöön kuuluu oleellisesti seuraavat kolme asiaa [Poh02]

- Sovellusaluekieli ja CASE-työkalu, jossa sitä voidaan käyttää
- *Kohdekehys*, jonka avulla kyseiselle sovellusalueelle voidaan kehittää sovelluksia tehokkaasti. Tässä kohdekehyksen tuoma hyöty ulottuu pelkän ohjelmakoodin luettavuuden ja ylläpidettävyyden kasvun lisäksi yksinkertaisimpiin ja tehokkaampiin generaattoreihin.

- Muuntimia eli generaattoreita, jotka osaavat generoida kohdekehystä hyödyntävää ohjelmakoodia sovellusaluekielillä esitettyjen mallien perusteella. Muuntimien toimintaperiaatteita on esitelty tarkemmin luvussa 5.

Seuraavassa kuvassa 4 on esitetty sovellusaluemallinnusta tukevan ympäristön eri osat:



Kuva 4, Sovellusaluemallinnusta tukevan ympäristön osat [Poh02].

Yllä olevan kuvan vasemmassa reunassa on esitetty Metacase-työkalu, jonka avulla sovellusaluekieliä sekä näitä hyödyntäviä CASE-välineitä voidaan kehittää sekä koodigeneraattori. Oikeassa laidassa on varsinaiseen mallintamiseen käytettävä CASE-työkalu sekä kohdekehys.

3.3 Sovellusaluekielet

Useissa insinööritieteissä voidaan erottaa spesifit ja yleiset ratkaisumallit toisistaan [Due00]. Yleinen lähestymistapa tarjoaa osittaisen ratkaisun moniin ongelmiin siinä missä spesifi lähestymistapa tarjoaa paremman ratkaisun pienempään määrään ongelmia. Tietojenkäsittelytieteissä tämä ero näkyy selvästi kun tutkitaan *yleiskäyttöisiä ohjelmointikieliä* ja *sovellusaluekieliä*.

Vanhimmat ohjelmointikielet kuten COBOL, Fortran ja Lisp kehitettiin nimenomaan ratkaisemaan tiettyjen sovellusalueiden ongelmia (liiketoimintaprosessit, numeerinen laskenta ja symbolien käsittely) [6]. Mainittujen ohjelmointikielten sovellusaluekohtainen luonne tulee ilmi jo niiden alkuperäisissä nimissä:"**Co**mmon **B**usiness **O**riented **L**anguage", "The IBM Mathematical **For**mula **Tran**slating System" ja ("**Lis**t **P**rocessing"). Loppujen lopuksi niistä on kehitetty yleiskäyttöisiä ohjelmointikieliä ja tiettyjen sovellusalueiden ongelmien ratkaisemiseksi on täytynyt kehittää muita ratkaisuja. Muun muassa seuraavia ratkaisuja sovellusaluekohtaisen ohjelmistotuotannon tukemiseksi on kokeiltu [Due00]:

- *Aliohjelmakirjastot* tarjoavat aliohjelmia tietyn sovellusalueen ohjelmien käytettäväksi. Tällaisia sovellusalueita voivat olla esimerkiksi differentiaalilaskenta, grafiikka, käyttöliittymät ja tietokantaoperaatiot.

- *Olioperustaiset ohjelmistokehykset* laajentavat aliohjelmakirjastojen ideaa. Aliohjelmakirjastot toimivat siten, että sovellukset kutsuvat aliohjelmakirjastojen palveluja. Ohjelmistokehykset taas toimivat usein siten, että kehyksellä on kontrolli sovelluksen toiminnasta ja se kutsuu varsinaista sovelluskohtaista koodia [Foo88] [Fay97].

- *Sovellusaluekielet* ovat pieniä, useimmiten deklaratiivisia kieliä. Ne tarjoavat ilmaisuvoimaa tietylle kapealle sovellusalueelle. Useimmiten sovellusaluekielillä esitetyt

---

[6] Katso lisätietoa esimerkiksi Wikipediasta: http://en.wikipedia.org/wiki/COBOL, http://en.wikipedia.org/wiki/Fortran ja http://en.wikipedia.org/wiki/Lisp_programming_language

mallit muunnetaan ohjelmakoodiksi, joka hyödyntää esimerkiksi olemassa olevan aliohjelmakirjaston koodia. Näin ollen se piilottaa ko. aliohjelmakirjaston yksityiskohtia tarjoten korkeamman abstraktiotason näkymän ko. kirjastoon.

Sovellusaluekielet (engl. Domain-Specific Languages) ovat siis tiettyjen tehtävien ratkaisemiseksi kehitettyjä ohjelmointikieliä, vastakohtana yleiskäyttöisille ohjelmointikielille [Fow05a]. Sovellusaluekielet tarjoavat sisäänrakennettuja abstraktioita ja notaatioita kyseiselle sovellusalueelle. Ne ovat täten vähemmän ilmaisuvoimaisia kuin yleiskäyttöiset ohjelmointikielet. Toisaalta sovellusaluekielten avulla voidaan ilmaista paljon tiiviimmin ja ilmaisuvoimaisemmin kyseisen sovellusalueen asioita.

Sovellusaluekielet voivat olla joko *vertikaalisia*, tietylle toimialalle tarkoitettuja (esimerkiksi rahoitusala, vakuutusala) tai *horisontaalisia*, usealle toimialalle tarkoitettuja kuten esimerkiksi relaatiotietokantojen kyselykieli SQL ja käyttöliittymien kuvauskielet. Hyviä esimerkkejä vertikaalisista sovellusaluekielistä on esimerkiksi musiikki- ja äänisyntetisaattorien toiminnan kuvaamiseksi kehitetty CSound[7] sekä ohjelmointikielten kielioppien jäsentäjien kuvaamiseksi kehitetty Yacc [Joh06]. Martin Fowler korostaa, että sovellusaluekielten kuvaama sovellusalue voi olla mikä tahansa sovellusalue. Sen ei siis tarvitse olla välttämättä kehitettävän sovelluksen ongelma-avaruudesta, vaan sovellusaluekielillä voidaan luonnollisesti kuvata myös sovelluksen toteutusavaruuden käsitteitä [Fow05a]. Näinhän on muun muassa ohjelmistoarkkitehtuureja kuvaavien sovellusaluekielten tapauksessa [Med97].

Sovelluksen ongelma-avaruutta kuvaavat sovellusaluekielet ovat useimmiten ennemminkin *deklaratiivisia* kuin *imperatiivisia*. Sovellusaluekielten tehokas käyttäminen vaatii, että kehittäjäorganisaatiolla on ennestään riittävä tietämys sovellusalueesta ja siitä, kuinka kyseiselle sovellusalueelle kehitetään laadukkaita ohjelmia. Tämä tuntuu loogiselta myös käytettäessä aiemminkin esillä ollutta analogiaa perinteiseen teollisuuteen: Ei ole mitään

[7] http://www.csounds.com/

järkeä ryhtyä kehittämään automaattista tuotantolinjaa jollekin tuotteelle, mikäli ko. tuotteen ominaisuudet eivät ole selvillä.

Greenfield ja kumppanit esittävät sovellusaluekielille muutamia vaatimuksia [Gre04a]:

- Mallinnuskielen tarkoitus tulee ilmaista täsmällisesti siten, että sovellusalueen asiantuntija voi helposti sanoa, onko kieli riittävän ilmaisuvoimainen kyseiseen käyttöön.
- Käsitteet, joita sovellusaluekielessä esiintyy, tulee olla relevantteja sovellusalueelleen.
- Mallinnuskielen käsitteillä tulee olla käyttäjilleen tutut, järkevät nimet.
- Mallinnuskielen käyttämä notaatio tulee olla johdonmukaista.
- Mallinnuskielellä tulee olla selvästi määritellyt säännöt, jota kutsutaan kieliopiksi. Kielioppi helpottaa työkaluin tehtävää validointia sekä käyttäjien mallinnustyötä.
- Jokaisen ilmauksen merkitys tulee olla tarkkaan määritelty, jotta käyttäjät voivat kehittää malleja, joita muut ymmärtävät ja joiden perusteella työkalut voivat generoida valideja toteutuksia.

Martin Fowlerin mukaan sovellusaluekielen generoimista ohjelmakoodiksi voisi ajatella jopa ohjelmiston *konfiguroinniksi* käännösaikana. Siinä missä perinteinen ohjelma huomioi konfiguraatioinformaation ajonaikana, voidaan sovellusaluekieltä joissain tapauksissa ajatella ohjelmiston konfiguraationa, joka ohjaa ohjelmiston käyttäytymistä käännösaikana[Fow05b].

3.3.1 Esitystavat

Sovellusaluekielet voidaan esittää joko tekstuaalisina (kuten edellä mainittu Yacc ja Csound), diagrammeina, lomakkeina tai graafisina. Modernit, metamallinnusta tukevat työvälineet kuten MetaEdit+ ja Microsoft DSL Tools (ks. luku 4.5) tarjoavat ympäristön graafisten sovellusaluekielten kehittämiseen samoin kuin valmiin ympäristön sovellusalue-

kielellä mallintamiseen piirtopintoineen ja työkalupakkeineen. Tämän tutkielman käytännön osuudessa kehitettävä sovellusaluekieli on graafinen.

3.3.2 Graafisen sovellusaluekielen kehittäminen

Sovellusaluekielten kehittämistä pidetään yleisesti vaikeana asiana. Näin on luonnollisesti silloin, kun yritetään kehittää sovellusaluekieltä jonkin muun kuin kehittäjäyrityksen omaan käyttöön tai kun yritetään kehittää sovellusaluekieltä, joka sopisi kaikkien käyttöön. Kuinka olisikaan mahdollista kehittää käyttökelpoista sovellusaluekieltä, mikäli sen käyttötarkoitus, käyttäjät sekä sovellusalue olisivat vieraita? Hyvänä esimerkkinä tällaisesta on UML:n määrittely[8] ja sen lähes olematon tuki automatisoinnille ja todelliselle malliperustaiselle kehitykselle.

Kielen kehittämisen lähtökohtana tulee olla sen käyttäminen nimenomaan yhdessä yrityksessä ja täten myös yhdellä toimialalla. Kun näin on, kielen perustana olevat käsitteet ovat tuttuja yrityksen aiemmasta toiminnasta, koska niitä on käytetty yrityksen tuotteen/tuotteiden kehittämisessä.

Sovellusaluemallinnuksen tueksi kehittävään kieleen tulee sisällyttää ja formalisoida ne konseptit, jotka ovat kielen perustana olevalle sovellusalueelle ja kieltä hyödyntäville toimijoille relevantteja. Nämä konseptit ovat olleet yrityksessä aktiivisessa käytössä jo aiemminkin, joskin implisiittisesti ja mahdollisesti epäformaalisti. On luonnollista, että kehitettävän kielen vaatimukset muuttuvat aikaa myöten ja täten sen kehitys voi olla luonteeltaan jopa iteratiivista [Tol06a]. Kieleen valituille sovellusalueen konsepteille määritellään lisäksi sen ominaisuudet, jotka voivat olla joko sovellusalueeseen liittyviä tai toteutukseen liittyviä, esimerkiksi generaattoreita tukevaa informaatiota [Met05a]. Tämän tutkielman käytännön osuudessa kehitettävän Entity Modeling Language -sovellusaluekielen elementeissä on sekä varsinaiseen tietomallinnukseen liittyviä ominaisuuksia sekä generaattoreita tukevia elementtejä (ks. luvut 7.3, 7.4 ja 7.5).

---

[8] http://www.uml.org/

Kun kehitetään graafista sovellusaluekieltä, tulee valittujen konseptien lisäksi määritellä käytettävä notaatio. Notaation symbolit esittävät sovellusaluekielen käsitteitä yksikäsitteiselle ja selkeällä tavalla. Sovellusaluekielen konseptien ja symbolien lisäksi sovellusaluekieliin liittyy useimmiten sovellusalueen *sääntöjä* opastamaan ja rajoittamaan sovellusaluekielellä mallintamista [Poh02] [Met05a]. Sääntöjen avulla voidaan esimerkiksi määritellä lailliset suhteet eri symbolien välillä ja se kuinka eri konsepteja voidaan käyttää mallinnuksessa. Kun sovellusaluekieleen liittyvät säännöt on määritelty, varmistutaan siitä, että kaikki kieltä käyttävät kehittäjät seuraavat samoja sääntöjä.

3.4 Kohdekehys

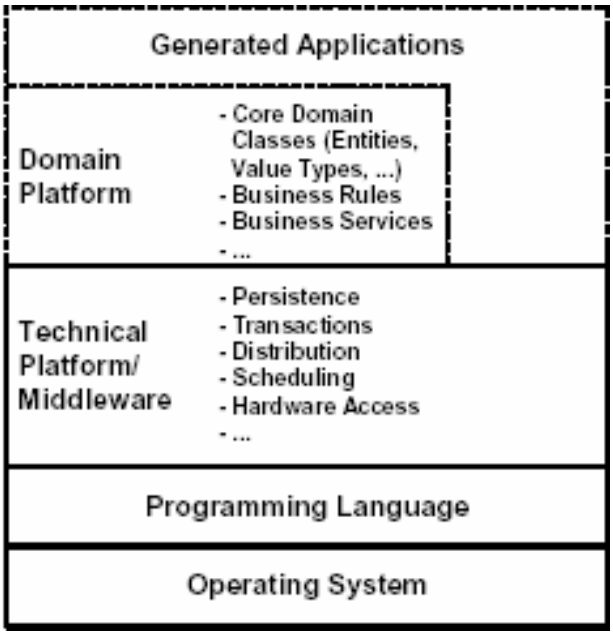
Kuten todettua, sovellusaluemallien muuntimet generoivat kohdekehystä hyväksikäyttävää ohjelmakoodia. Tämä korostaa kohdekehyksen arkkitehtuurin roolia menestyksekkään sovellusaluemallinnusta tukevan ympäristön kehittämisessä [Poh02]. Kuten ohjelmistoarkkitehtuurissa yleensä, myös kohdekehyksen kehittämisessä päätavoitteena on nostaa abstraktiotasoa piilottamaan yleisesti esiintyvien ongelmien monimutkaisuutta kehittäjiltä hyviä ohjelmistosuunnittelun periaatteita sekä toisaalta sovellusaluetietämystä hyväksikäyttäen. Kohdekehyksen avulla voidaan nostaa abstraktiotasoa periaatteessa kolmella tasolla [Poh02]:

- Matalimmalla tasolla kohdekehys tarjoaa primitiivipalveluja suoraan valitun ohjelmointikielen päälle. Hyvänä esimerkkinä tästä on kohdekehyksen tarjoama aliohjelmakirjasto.

- Toisella tasolla kohdekehys voi käyttää näitä primitiivipalveluja tarjotessaan ”peruspalikoita” sovelluksille tai tuotteille. Tämän tason tavoitteena on toteuttaa käytettävän sovellusaluekielen vastineet siten, että muuntimet voivat suoraviivaisesti tuottaa näitä sovellusaluemallien perusteella.

- Korkeimmalla tasolla kohdekehys voi tarjoa konfiguraatiomekanismin, joka sallii toisen tason peruspalikoiden yhdistelemisen suoraan kokonaisiksi ohjelmiksi tai tuotteiksi.

Sovellusaluemallinnuksen Pattern-kokoelmassaan *Patterns for Model-Driven Software Development* Markus Völter ja Jorn Bettin esittelevät kohdekehyksen yhtenä tärkeimmistä sovellusaluemallinnusta edistävistä asioista [Völ04]. Kohdekehystä kuvaavan patternin nimi on *Rich domain-specific platform* ja siinä kohdekehys sisältää muun muassa kirjasto-, ohjelmistokehyksiä, kantaluokkia ja tulkkeja. Tällaisen juuri tietylle sovellusalueelle kehitetyn kohdekehyksen etuina mainitaan generaattoreiden kompleksisuuden väheneminen ja sovellusaluekielten suoraviivaisempi kuvautuminen kohdekehyksen komponentteihin. Kuvassa 5 on esitetty kohdekehyksen korkeampi abstraktiotaso ohjelmointikieleen ja tekniseen alustaan nähden.



Kuva 5, kohdekehyksen (kuvassa *Domain Platform*) korkeampi abstraktiotaso tekniseen alustaan (kuten esimerkiksi Microsoft .NET) ja ohjelmointikieliin nähden [Völ04]

Tutkielman käytännön osuudessa kehitettävä mallinnustyökalu hyödyntää olemassa olevaa kohdekehystä, joka tarjoaa peruspalveluja Enterprise-tason kolmikerrossovellusten kehittämisen tueksi yllä olevan kuvan mukaisesti. Tutkielman käytännön osuudessa käytetystä kohdekehyksestä on kerrottu tarkemmin luvussa 6.

3.5 Malliperustainen kehitys

Sovellusaluemallinnusta voidaan hyödyntää kehitysprosessissa eri tavoin. Sovellusaluekielillä voidaan joko vain tukea tuotekehitystä tai sitten tuotantoprosessi voi olla kokonaan *malliperustainen* (engl. *model-driven*). Sovellusaluemallinnukseen perustuvasta kehityksestä käytetään myös termiä *Domain-Specific Modeling*, DSM[9]. DSM-lähestymistapa korostaa nimenomaan malliperustaista kehitystä, jossa ohjelmat määritellään korkean tason sovellusaluekielten avulla ja varsinainen toteutus generoidaan automaattisesti näiden sovellusaluemallien perusteella. Seuraavassa on koottuna kriteerejä malliperustaiselle kehitykselle:

- Versiointi koskee vain malleja, ei ohjelmakoodia. Koska ohjelmakoodi voidaan generoida milloin vain, ei ole tarpeen versioida itse koodia. Analogiana voidaan ajatella perinteistä ohjelmistokehitystä: Emme versioi kääntäjän tekemiä objektitiedostoja vaan C-koodia.

- Generoituun ohjelmakoodiin ei tarvitse koskea. Mikäli generoitu koodi kaipaa parannusta, teemme muutoksen joko sen tuottavaan generaattoriin, mallinnuskieleen tai kohdekehyksen komponentteihin. Jälleen analogiana perinteiseen kehitykseen: Ei ole järkeä hakkeroida käännettyä assembler-koodia vaan ennemminkin kannattaa hankkia parempi kääntäjä, mikäli kääntäjän lopputulos ei täytä sille asetettuja vaatimuksia.

- Generoitu koodi täyttää kaikki ohjelmistolle asetetut laatuvaatimukset.

- Testausta kyetään suorittamaan jo mallinnuskielen tasolla. Hyvä mallinnuskieli estää laittomien mallien kehittämisen ja täten se edesauttaa sovelluksen toiminnallisuuden testaamisen jo suunnitteluvaiheessa.

[9] Katso http://www.dsmforum.org

- Mallinnuksessa hyödynnetään asiakkaita. Kun kielessä käsitellään suoraan sovellusalueen käsitteitä, on ohjelman toiminta helpompi lukea, ymmärtää, muistaa ja varmentaa kuin käytettäessä perinteistä ohjelmointikieltä.

Malliperustaiseen kehitykseen siirtyminen edesauttaa tuottavuuden, luotettavuuden, uudelleenkäytettävyyden ja ylläpidettävyyden kasvua. Ohjelmiston toimintaa on tällöin myös mahdollista verifioida jo sovellusaluemallin tasolla.

Sovellusaluekielen ja sitä hyödyntävän koodigeneraattorin kehittäminen vaatii oleellisesti seuraavat asiat onnistuakseen:

- Hyvin määritellyn ko. sovellusalueelle sopivan kohdekehyksen, joka tarjoaa sovellusalueen abstraktioita ohjelmistosuunnittelijoiden käyttöön.

- Valmiita sovelluksia, jotka hyödyntävät kohdekehyksen komponentteja. Valmista ohjelmakoodia voidaan hyödyntää kehitettäessä muuntimia, jotka osaavat tuottaa hyvien käytäntöjen ja ohjelmointitapojen mukaista ohjelmakoodia.

- Sovellusalueen asiantuntija, jolla on riittävästi aikaa käytettävissä.

- Hyvä metamallinnustyökalu.

3.6 Sovellusaluekielen kehitysprosessi

Sovellusaluekielen kehittämisessä sekä kehitetyn sovellusaluekielen käyttöönotossa on havaittavissa usein seuraavat pääpiirteet: *Ongelman havaitseminen*, *metamallin kehittäminen*, *työkalujen kehittäminen* sekä *kohdekehyksen muuttaminen*.

Seuraavassa on kuvattu tiivistetysti tämän tutkielman käytännön osuudessa, luvussa 7 esitettävän Entity Modeling Language-sovellusaluekielen kehitysprosessin pääpiirteet.

1. *Ongelman havaitseminen*. Sovellusalueen huomioivan tietomallinnustyökalun kehittämien lähti havainnosta, että hajautettuun kolmikerrosarkkitehtuuriin perustuvan sovelluksen tietomallin muuttaminen on työlästä käsin tehtäväksi. Esimerkkinä monimutkaisuudesta on se, että lisättäessä attribuutti sovelluksessa esiintyvälle tie-

tueelle, joudutaan muutos tekemään noin 15 paikkaan ympäri ohjelmistoa (mukaan lukien ohjelmakoodi, erinäiset Excel-tiedostot, resurssitiedostot, SQL-skriptit, käsin ylläpidettävät ER-mallit sekä muu dokumentaatio).

2. *Metamallin kehittäminen*. Tietomallin kuvaamiseksi kehitettiin sovellusaluekieli, joka mahdollistaa pelkän ER-mallinnuksen lisäksi joidenkin sovellusalueeseen liittyvien semanttisten piirteiden mallintamisen suoraan tietomalliin.

3. *Työkalujen kehittäminen*. Metamalli implementoitiin Visual Studio-kehitysympäristöön integroituvaksi piirtotyökaluksi, joka mahdollistaa sovelluksen tietomallinnuksen graafisesti. Lisäksi metamallia hyödyntäviä generaattoreita kehitettiin tuottamaan muun muassa ohjelmakoodia, sql-skriptejä sekä dokumentaatiota.

4. *Kohdekehyksen muuttaminen*. Kohdekehystä joudutaan lähes aina muuttamaan, jotta kehitetyistä työkaluista saataisiin kaikki teho irti. Esimerkiksi generoidun ja käsin kirjoitettavan ohjelmakoodin eriyttäminen saattaa vaatia joitain muutoksia sovellusalustan arkkitehtuuriin ja ohjelmointikäytäntöihin.

Sovellusaluemallinnusta hyödyntävien työkalujen kehittäminen on usein iteratiivista, edellä esitetyssä listassa saattavat vaiheet 2-4 toistua useaankin kertaan. Edellä mainitut vaiheet saattavat lisäksi asettaa vaatimuksia muissa vaiheissa toteutettaville operaatioille, lisäten prosessin iteratiivista luonnetta.

Kuten myös aiemmin totesimme, on sovellusaluekielten iteratiivisen kehityksen vuoksi usein järkevää nimetä joukko edistyneitä kehittäjiä kehittämään sovellusaluekieliä, generaattoreita sekä työkaluja, joita muu kehitysorganisaatio voi työssään käyttää. Näin ollen kehittäjät ikään kuin jaettaisiin kahteen leiriin (vertaa perinteinen arkkitehti-ohjelmoija-jako).

3.7 Konsepteja ja työkaluja

Tässä esitellään muutama sovellusaluemallinnukseen perustuva konsepti.

3.7.1 MetaEdit+

MetaEdit+ on MetaCasen[10] kehittämä sovellusaluemallinnusta tukeva ympäristö. Se tarjoaa GOPPRR-metamallinnuskielen mukaisen metamallinnusympäristön omien CASE-työkalujen kehittämiseksi. GOPPRR tulee sanoista *Graph*, *Object*, *Property*, *Port*, *Relationship* ja *Role*, jotka ovat kyseisen metamallinnuskielen elementit. GOPPRR metamallinnuskielessä graafit koostuvat olioista, oliot ominaisuuksista ja oliot voidaan liittää toisiinsa suhteiden ja porttien avulla. Lisäksi suhteisiin liittyy aina yhtä monta roolia kuin on liittyviä olioitakin.

MetaEdit+-työkalussa on oma (ohjelmointi)kieli, *Report Definition Language* generaattoreiden kehittämiseen. Kieli on alun perin tarkoitettu nimensä mukaisesti raporttien generoimiseen ja kyselyjen tekemiseen malleista, mutta se on viime aikoina saanut enemmän ohjelmointikielimäisiä piirteitä, jotka ovatkin välttämättömiä vähänkin monimutkaisemman generaattorin kehittämisessä.

3.7.2 MDA

MDA on Object Management Group-organisaation[11] vuodesta 2001 lähtien kehittämä formaali malliperustainen konsepti, joka korostaa alustariippumattomuutta. MDA tulee sanoista *Model-Driven Architecture*. MDA pohjautuu sovellusaluekielten käyttämisen lisäksi muun muassa UML-mallinnuskieleen. MDA:ssa ohjelmisto mallinnetaan ensiksi alustariippumattomasti (*Platform Independent Model, PIM*) ja vasta sen jälkeen alustakohtaisesti (*Platform Specific Model, PSM*). MDA-yhteensopivia työkaluja on varsin niukasti ja sen konkretisoituminen käytettävään muotoon onkin ollut vähäisempää.

MDA-konseptia arvostellaan sen liian yleisestä luonteesta. Todellisessa ohjelmistonkehityksessä on varsin harvoin tarpeen huolehtia siitä, että järjestelmä toimii kaikilla mahdollisilla alustoilla.

---

[10] www.metacase.com

[11] www.omg.org

3.7.3 Software Factories

Software Factories on Microsoftin kehittämä konsepti, jonka keskeisimpänä lähtökohtana on sovellusaluekielten ja niitä hyödyntävien työkalujen kehittäminen sovelluskehityksen tueksi. Konseptin perusperiaatteet esitellään seuraavassa luvussa 4.

4 Software Factories -konseptin erityispiirteitä

Software Factories on Microsoftin kehittämä konsepti, jonka tavoitteena on ohjelmistotuotannon automatisointiasteen radikaali parantaminen. Konseptin pääarkkitehteina ovat samannimisen kirjan (*Software Factories: Assembling application with patterns, models, frameworks and tools*) [Gre04a] vuonna 2004 kirjoittaneet *Jack Greenfield* ja *Keith Short* sekä kirjan kirjoittamiseen osallistuneet *Steve Cook* ja *Stuart Kent*.

Software Factories – vision lähtökohtana ovat olleet ensimmäisessä luvussa esitellyt ohjelmistokehityksen haasteet sekä nykyisen ohjelmistoteollisuuden krooniset ongelmat. Konsepti pyrkii tehostamaan ohjelmistotuotantoa nostamalla ohjelmistokehityksen abstraktiotasoa malliperustaisen ohjelmistokehityksen avulla. Se pyrkii nimensä mukaisesti saattamaan ohjelmistokehityksen teolliselle tasolle samaan tapaan kuin perinteisillä aloilla tapahtui teollisen vallankumouksen aikaan [Gre04b]. Tällä hetkellähän ohjelmistokehitys on pitkälti kallista käsityötä, eikä merkittävää automatisointia ole saavutettu juuri ollenkaan. Tärkeimpänä tekijänä tuottavuuden kasvattamisessa Greenfield ja Short näkevät sovellusaluekielet ja niillä esitettyjen mallien automaattinen muuntaminen lähdekoodiksi.

Yhtenä hyvänä esimerkkinä automatisoinnista ohjelmistotuotannossa Greenfield ja Short esittävät graafisten käyttöliittymien (ns. WYSIWYG-) suunnittelutyökalut. Tutkiessaan syitä, miksi tietyillä ohjelmistotuotannon osa-alueilla kuten juuri graafisten käyttöliittymien kehityksessä on saavutettu niin merkittävää abstraktiotason ja tuottavuuden nousua he löysivät mallin (ns.*Software Factories Pattern*), jossa esiintyy seuraavat elementit:

1. Kehitetään uudelleenkäytettäviä elementtejä kuten *ohjelmistokehyksiä*, *sovellusaluekohtaisia suunnittelumalleja* ja *prosesseja*. Tätä kokonaisuuttahan päätettiin jo aiemmin kutsua kohdekehykseksi. Kohdekehyksen komponentit rakennetaan esimerkiksi jotain tunnettua arkkitehtuurityyliä noudattaen, jolloin sen käyttäminen ja ymmärtäminen helpottuu. Kohdekehystä voidaan hyödyntää kehitettäessä sovelluksia tai niiden osia tietylle sovellusalueelle. Kohdekehystä käytetään mukauttamalla, konfiguroimalla ja kokoonpanemalla kohdekehyksen komponentteja. Kohdekehys voi olla yksinkertaisimmillaan esimerkiksi aliohjelmakirjasto. Olio-ohjelmointia tukeva kohdekehys voi esimerkiksi sisältää joukon sovellusaluekoh-

taisia suunnittelumalleja, joita käyttäen kehitettävistä sovelluksista saadaan laadukkaampia, tehokkaampia, turvallisempia ja ennen kaikkea ylläpidettävämpiä [Gam95].

2. Kohdekehysten komponenttien kokoonpanoa varten kehitetään sovellusaluekieliä, joiden avulla komponenttien kokoonpano, mukauttaminen ja konfigurointi voidaan suorittaa formaalilla tavalla.

3. Sovellusaluekielten tueksi kehitetään työkaluja, joiden avulla sovellusaluekielen mukaisten mallien kehittäminen mahdollistetaan ja joiden avulla sovellusta voidaan täten mukauttaa nopeasti muuttuviin vaatimuksiin. Työkalut osaavat tuottaa kohdekehyksen komponentteja hyväksikäyttävää ohjelmakoodia suoraan sovellusaluemallien perusteella siten että sovelluksesta on koko ajan toimiva versio olemassa. Työkalujen käytön tukena voidaan käyttää kehittäjien lisäksi liiketoiminta-alueen asiantuntijoita kuten esimerkiksi asiakkaan edustajia.

4. Suunnitteluinformaatio jalostetaan sellaiseen muotoon, että ajettavien sovellusten generoiminen suoraan sovellusaluemallien perusteella mahdollistuu.

Itse ohjelmistotehtaalla tarkoitetaan räätälöityä ohjelmistotuotelinjaa, jonka avulla on tehokasta tuottaa sovelluksia tietylle sovellusalueelle. Se sisältää sovellusaluekohtaisen kohdekehyksen, sovellusaluekieliä, sovellusalueen suunnittelumalleja, prosesseja sekä näitä hyödyntäviä työkaluja [Gre04a]. Työkalujen avulla ohjelmistokehityksessä voidaan automatisoida rutiininomaiset ja täten ohjelmistokehittäjän näkökulmasta vähäpätöiset työtehtävät. Tämänkaltaisen ohjelmistotehtaan avulla kehitystiimi voi tuottaa tehokkaasti yksilöllisiä vaatimuksia sisältäviä ohjelmistoja tietylle sovellusalueelle. Prosessien, arkkitehtuureiden sekä pakkaustekniikoiden standardisoimisten avulla voidaan edesauttaa tuotantoketjujen syntymistä [Gre04c].

Microsoft pyrkii toteuttamaan Visual Studio - kehitysympäristön ympärille työkaluja, joiden avulla siitä saataisiin tällainen tehokkaasti tietylle sovellusalueelle ohjelmistoja tuottava tuotelinja.

Tärkein työkalu Software Factories -vision toteuttamiseksi on DSL Tools, jonka avulla sovellusaluekielten kehittäminen ja käyttäminen mahdollistetaan Visual Studio – kehitysympäristössä. DSL Tools-työkalua on esitelty tarkemmin luvussa 4.5.

Software Factories -konsepti korostaa neljää erillistä periaatetta, joita se käyttää pyrkiessään automatisoimaan ohjelmistokehitystä. Ne ovat *systemaattinen uudelleenkäyttö*, *malliperustainen kehitys*, *sovellusten kokoonpano* sekä *prosessikehykset*.

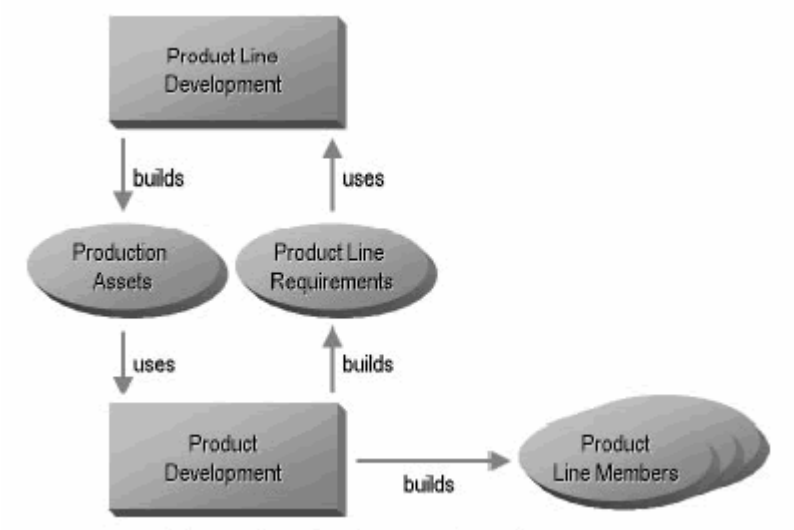
4.1 Systemaattinen uudelleenkäyttö

Sovellustuoteperheellä (*Software Product Family*) tai – *linjalla* (*Software Product Line*) tarkoitetaan tietylle sovellusalueelle toteutettujen, toiminnaltaan ja rakenteeltaan samankaltaisten, osittain samoista osista koostuvien ja samalla tuotantomenetelmällä tai työkaluilla toteutettujen ohjelmistotuotteiden joukkoa [Sei06]. Parnasin mukaan sovellustuoteperhe tarjoaa kontekstin, jossa eri sovelluksille yhteiset ongelmat voidaan ratkaista kootusti [SF04]. Tuoteperheen sisällä samantapaiset ongelmat pyritään ratkaisemaan tuoteperheen yhteisten sääntöjen ja mallien mukaisesti mahdollisimman paljon uudelleenkäytettäviä ohjelmistokomponentteja hyväksikäyttäen.

Uudelleenkäyttö on ohjelmistotuotannossa keskimäärin varsin vähäistä, varsinkin eri tuotteiden kesken. Saman ohjelmiston sisällä samaa koodia vielä pystytään uudelleenkäyttämään, mutta eri sovellusten kesken koodin uudelleenkäyttö on verrattain pientä. Kun komponentteja tehdään tiettyä kontekstia varten ja tiettyyn tuoteperheeseen, voidaan tuotoksia uudelleenkäyttää suuremmalla todennäköisyydellä. Myös toteutusalustan kiinnittämisellä saavutetaan luonnollisesti suurempi uudelleenkäyttöaste. Esimerkiksi Microsoft .NET – alustalle tehtyjen käyttöliittymäkontrollien kehittäminen uudelleenkäytettäviksi on helpompaa verrattuna tilanteeseen, jossa sovellusalustaa ei ole kiinnitetty. Tutkielman käytännön osuus hyödyntää kehitettyä sovellusalustaa, joka tarjoaa palveluja nimenomaan tietynlaisten sovellusten kehittämiseksi Microsoft .NET-alustalle (ks. luku 6).

Systemaattisen uudelleenkäytön ideana on jakaa ohjelmistokehittäjät kahteen kategoriaan: Niihin jotka tuottavat uudelleenkäytettäviä komponentteja ja niitä jotka hyödyntävät kehitettyjä komponentteja varsinaisessa sovelluskehityksessä. Systemaattisen uudelleenkäytön

yhteydessä puhutaan *sovellustuotelinjoista*. Sovellustuotelinja on kokonaisuus, jonka avulla tiettyyn sovellustuoteperheeseen kuuluvia sovelluksia voidaan tuottaa tehokkaasti. Tuotelinjan kehittäjät pyrkivät löytämään yhteisiä ominaisuuksia esimerkiksi useista yksittäisistä projekteista ja paketoimaan ne uudelleenkäytettävään muotoon. Käytännössä tämä voi tarkoittaa esimerkiksi tuotelinjan alustana olevan sovelluskehyksen toteuttamista. Tuotelinjaa kehitetään myös parantamalla itse kehitysprosessia ja työkaluja. Kehitysprosessin parantaminen voi tarkoittaa esimerkiksi sovelluskehittäjien ja uudelleenkäytettäviä komponenttien kehittäjien välisen kommunikoinnin parantamista. Kuvassa 6 on esitetty systemaattista uudelleenkäyttöä hyödyntävän tuotelinjamaisen kehityksen prosessi:



Kuva 6, Systemaattista uudelleenkäyttöä hyödyntävä tuotelinjamainen kehitysprosessi

Tuotelinjan kehittäjät kehittävät tuotteiden kehittäjien tueksi *tuotantoelementtejä*, jotka voivat joko *toteutuselementtejä* tai *prosessielementtejä*. Tuotantoelementtejä ovat esimerkiksi sovellusaluekielet, sovellusalueen suunnittelumallit, ohjelmistokehykset ja komponentit. Prosessielementtejä ovat esimerkiksi mikroprosessit (ks. luku 4.4), jotka sisältävät opastusta tietyn tuotantoelementin käyttöön. Tuotteiden taas kehittäjät antavat vaatimuksia tuotelinjan kehittäjille kehittäessään tuoteperheeseen kuuluvia tuotteita.

Uudelleenkäyttö ei varsinaisesti ole mikään uusi idea ohjelmistokehityksessä, mutta esitetynkaltaisen, systemaattisesti uudelleenkäytettäviä komponentteja kehittävän organisaation

käyttäminen ohjelmistotuotteiden kehittämisen tukena on jokseenkin ennakkoluuloton idea.

4.2 Malliperustainen kehitys

Kuten todettu, keskeisimpänä asiana Software Factories -konseptissa on sovellusaluekielillä esitettyjen korkean tason sovellusaluemallien käyttäminen ohjelmistotuotannon perustana. Sen sijaan, että ohjelmistokehitystä tehtäisiin kaikkein yksityiskohtaisimmalla tasolla, voidaan malliperustaisen kehityksen avulla vähentää kehityksessä kohdattavaa monimutkaisuutta. Usein ohjelmistokehityksessä on paljon rutiininomaisia ja varsin vähän luovuutta vaativia toimintoja, joita ohjelmistokehittäjien on tehtävä toteuttaessaan sovelluksia. Malliperustaisen kehityksen avulla nämä rutiinit voidaan antaa työkalujen tehtäväksi.

Analogiana abstraktiotason nostosta voitaisiin esittää Microsoft SQL Server-tietokantapalvelimen hallintasovellus Enterprise Manager. Sen avulla voidaan hallita kaikkia tietokantapalvelimen olioita: luoda tietokantoja, lisätä näihin tauluja ja näkymiä, muokata taulujen rakenteita ja käsitellä niiden sisältöä. Kaikki tämä tehdään graafisen käyttöliittymän kautta. Tällä tavalla on mahdollista nostaa tietokannan hallinnan abstraktiotasoa, perinteisestihän tietokantoja hallitaan SQL-kielen avulla suoraan tekstimuotoisia skriptejä käsittelemällä. Työkalulla voidaan tehdä helpon ja graafisen näkymän kautta kaikki samat määritykset kuin SQL:n kautta. Työkalun tarjoama näkymä antaa tietokannoista hyvin jäsennellyn ja organisoidun kuvan, jonka kautta tietokantojen rakenteet ovat helposti ymmärrettävissä. SQL:n tarjoama abstraktiotaso on siis nostettu helpommin ymmärrettävälle tasolle ja visuaalisen mallin avulla voidaan tehdä kaikki tietokannan määritykset tuntematta lainkaan SQL:ää.

Mallit, jotka pystyvät tarjoamaan kaikki samat mahdollisuudet matalamman tason esityksen kanssa, ovat kaikista käyttökelpoisimpia. Edellä mainitun esimerkin valossa tämä tarkoittaa sitä, että kaikki tietokannan määritykset on voitava tehdä Enterprise Manager –työkalun kautta. Jos osa määrityksistä jouduttaisiin tekemään suoraan tekstimuotoisen SQL:n kautta, rajoittaisi tämä työkalun käyttömahdollisuuksia ja tekisi toiminnasta tehottomampaa.

Abstraktiotasoa voidaan nostaa oleellisesti kahdella tavalla: korkean tason sovellusaluekielten avulla sekä tarjoamalla korkean tason komponenttikirjastoja ohjelmistokehityksen tueksi. Oleellinen ero näiden kahden välillä on se, että sovellusaluekieliin perustuvat abstraktiot ovat koneellisesti käsiteltävissä ja lisäksi sovellusaluekielillä esitettyjen mallien perusteella voidaan generoida mitä tahansa metadataa ohjelmistokehityksen tueksi. Vaikka modernit komponenttikirjastot tarjoavat korkean tason toimintoja ohjelmistokehittäjien tueksi, sovellusaluekielten puuttumisen vuoksi niitä on käytettävä yhä matalan tason yleiskäyttöisistä ohjelmointikielistä käsin.

Software Factories -konsepti korostaa sitä, että mallien tulisi olla sovelluskehityksen keskeisimpiä artefakteja, sen sijaan että ne olisivat vain osa dokumentaatiota odottaen vanhentumistaan [Ms05]. Sovellusaluemalleilla tulee olla tarkasti määritelty syntaksi ja semantiikka, joka kertoo millä tavalla ne kuvautuvat toteutusartefakteiksi kuten esimerkiksi koodiksi, projektistruktuureiksi, konfiguraatiotiedostoiksi ja tietokantaskripteiksi. Tällä tavalla mallit ovat ikään kuin lähdekooditiedostoja perinteisessä ohjelmistokehityksessä ja menetelmät joilla mallit muunnetaan toteutusartefakteiksi muistuttavat kääntäjien toimintaa kuten huomaamme luvussa 5.

4.3 Sovellusten kokoonpaneminen

Kokoonpanoajattelun tuominen ohjelmistokehitykseen ei ole itsessään mikään uusi idea. Muun muassa olio-ohjelmointi ja komponenttiperustainen kehitys ovat osaltaan pyrkineet saattamaan ohjelmistokehitystä sille tasolle, että kehittäjäorganisaatioiden ei tarvitsisi parhaassa tapauksessa kuin valita sopivat komponentit valmiiden komponenttien joukosta ja vain koota ohjelmisto näitä valmiita komponentteja hyväksikäyttäen. Kuten todettua, mikään yksittäinen tekijä ei ole tähän päivään mennessä mahdollistanut toimivien komponenttimarkkinoiden syntymistä, ainoastaan käyttöliittymäkomponenttien (-kontrollien) tapauksessa on saavutettu jonkinlaisia toimittajarajat ylittävää uudelleenkäyttöä. Esimerkkinä komponenttien kauppapaikasta on Component Source[12], jossa volyymiltaan suurin

[12] www.componentsource.com

kauppanimeke on käyttöliittymäkontrollit ja toiseksi suurin Internet-kommunikaatioon liittyvät komponentit.

Sovellusten kehittäminen kokoonpanemalla on Software Factories -konseptin laajemman mittakaavan visio. Sen toteutuminen ei välttämättä vaikuttaisi yksittäisen ohjelmistoyrityksen sisäiseen kehitysprosessiin, vaan se pyrkii mahdollistamaan nykyistä kehittyneempien *tuotantoketjujen* syntymisen rohkaisemalla käyttämään kokoonpanoajatteluun erityisen hyvin soveltuvia tekniikoita. Tällöin esimerkiksi yksittäinen ohjelmistoyritys voisi keskittyä vain yhdentyyppisten sovellusten osien kehittämiseen kokonaisten sovellusten sijaan ansaintalogiikkanaan myydä kehitettyjä komponentteja muiden ohjelmistoyrityksien käyttöön.

Perinteisessä teollisuudessa on siirrytty jo kauan sitten käsityöstä tuotantoketjujen käyttöön. Tuotantolinjojen avulla tuotevariantteja kyetään kokoamaan nopeasti useiden toimittajien toimittamista osista sen sijaan, että tuotteet valmistettaisiin alusta lähtien käsin. Tuotantolinjoihin siirtyminen vaatii tuotteiden osien, tuotantomenetelmien, pakkauksien sekä muun muassa prosessien tarkkaa standardoimista onnistuakseen [Gre04b]. Tuotantolinjoihin laitetun panostuksen vastapainoksi saavutetaan muun muassa eri toimittajien kesken hajautettu riski sekä parantunut taloudellisuus.

Millä tavalla ohjelmistotuotannon teollistuminen ja tuotantolinjojen syntyminen tulee näkymään? Vaikka sitä ei voida sanoa ennen kuin se tapahtuu, on mahdollista esittää valistuneita arvauksia perustuen ohjelmistoteollisuuden tähänastiseen evoluutioon sekä perinteisten teollisuudenalojen kehitykseen. Seuraavassa käydään läpi tekniikoita, joiden Software Factories -konseptin kehittäjät uskovat sopivan erityisen hyvin tuotantolinjamaiseen ohjelmistokehitykseen [Gre04a].

4.3.1 Alustariippumattomat protokollat

Alustariippumattomilla protokollilla tarkoitetaan erityisesti Web Services -hajautustekniikkaa. Web-palvelut hyödyntävät useita avoimia standardeja sekä protokollia, minkä vuoksi niiden avulla on mahdollista integroida erilaisten sovellusten toiminnalli-

suutta. Oleellista Web Services -tekniikassa on se, että se piilottaa varsinaisen toteutustekniikan palvelun käyttäjiltä.

Web-palveluissa käytetään XML-kieltä (Extensible Markup Language) kaikessa tietojen esittämisessä. XML-muotoista dataa siirretään sovellusten välillä yleisten protokollien avulla. Näitä ovat muun muassa HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol) ja SMTP (Simple Mail Transfer Protocol). Web-palveluiden julkinen rajapinta kuvataan WSDL-kielen (Web Services Description Language) avulla. Se kuvaa XML-muodossa kuinka web-palvelun kanssa tulisi kommunikoida. SOAP (Simple Object Access Protocol) -standardin mukaisia XML-pohjaisia viestejä käytetään web-palveluiden kutsuissa ja vastauksissa.

On sanottu että Web Services -teknologia onnistui siinä missä aiemmat komponenttiteknologiat epäonnistuivat erottaessaan palvelun kuvaustekniikan ja palvelun toteutustekniikan toisistaan. Alustariippumaton olioiden hajautustekniikka CORBA[13] pyrki samaan lopputulokseen eriyttäessään palvelujen kuvaustekniikan varsinaisesta toteutustekniikasta, mutta siinä oli pääasiallisena ongelma raskas käyttöönotto ja loppujen lopuksi varsin vähäiset implementaatiot eri alustoille.

4.3.2 Itseäänkuvaavat komponentit

Metadatan avulla ajettaviin tiedostoihin voidaan liittää informaatiota komponentin käytöstä. Itseään kuvaavat komponentit vähentävät arkkitehtuurien yhteensopimattomuutta tehden komponenttien vaatimuksista, riippuvuuksista, käyttäytymisestä, resurssienkulutuksesta ja sertifioinneista eksplisiittisiä. Komponenttien metadataa voidaan käyttää moneen tarkoitukseen, muun muassa niiden etsimiseen, valintaan, lisensointiin, hankintaan, asennukseen, testaukseen, konfigurointiin, monitorointiin, versiointiin ja hallintaan.

[13] http://www.omg.org/corba

Microsoft .NET-arkkitehtuuri ja sen suoritusympäristö käyttää ajettavina tiedostoina itseäänkuvaavia *koonteja* (engl. *Assemblies*) ja samankaltaista tekniikkaa käytetään myös esimerkiksi Java-ajoympäristössä.

4.3.3 Orkestrointi

Orkestrointi liittyy ensisijaisesti palveluorientoituneiden, hajautettujen järjestelmien kehitykseen. Se tarkoittaa sitä, että erityisen *orkestrointimoottorin* avulla voidaan hallita monimutkaisia Web Services -rajapintoihin liittyviä sopimuksia [Gre04a]. Tällöin eri Web Palveluja tarjoavien instanssien välillä ei tarvitse olla riippuvuuksia keskenään, vaan näiden välinen yhteistoiminta hallitaan orkestroimalla ne työkalun avulla. Esimerkkinä orkestrointimoottorista on Microsoft BizTalk Server.

Orkestroinnin avulla voidaan saattaa jopa eri organisaatioissa toimivat Web palvelut hyötymään toisistaan. Orkestroinnilla on tässä samankaltainen rooli kuin on *välittäjällä* samannimisessä suunnittelumallissa [Gam95]. Välittäjähän mahdollistaa muuten toisilleen vieraiden komponenttien yhteistyön hoitaen tarvittavat tietomuunnokset ja suodatukset komponenttien väliltä. Välittäjä voi myös monitoroida viestiliikennettä, hallita monimutkaisten kutsusekvenssien tilaa, suorittaa lokitusta sekä harjoittaa tietoturvapolitiikkaa.

4.3.4 Arkkitehtuuriperustainen ohjelmistokehitys

Ohjelmistoarkkitehtuuri määrittää ohjelmiston komponentit, niiden väliset vuorovaikutukset ja mahdolliset muodot, joissa ne voivat yhdessä esiintyä [Gar94]. Se on korkean tason esitys piilottaen suuren määrän ohjelmiston toteutuksen yksityiskohdista. Ohjelmistoarkkitehtuuri tarjoaa myös yleisiä suosituksia ja rajoituksia komponenttien suunnitteluun ja kehittämiseen.

Ohjelmistoarkkitehtuurin kuvauksen perusteella voidaan tehdä päätelmiä ohjelmiston laadullisten vaatimusten toteuttamisesta. Laadullisia vaatimuksia ovat esimerkiksi turvallisuus, suorituskyky, luotettavuus ja ylläpidettävyys. IEEE:n standardi 1471 ohjelmistoarkkitehtuureista tarjoaa joitain ohjeita arkkitehtuurin kuvaamiseen. Kuvaukseen liittyy aina *osakas* (engl. Stakeholder), jolla on mielenkiintoa järjestelmän tiettyyn aspektiin. Arkkiteh-

tuuri voidaan kuvata tarjoamalla eri osakkaille soveltuvia *näkymiä* järjestelmään. Eri näkymiä voi olla esimerkiksi:

- Toiminnallinen näkymä,
- Koodinäkymä,
- Rakenteellinen näkymä,
- Fyysinen näkymä ja
- Käyttötapaus/palautenäkymä.

Arkkitehtuurityyli on toistuvasti esiintyvä malli ohjelmistoarkkitehtuurista. Se siis ryhmittelee ohjelmistoarkkitehtuurit eri ryhmiksi sen perusteella, minkälaisia yhteisiä ominaisuuksia niissä on. Eri arkkitehtuurityylejä ovat esimerkiksi oliotyyli, putket ja suodattimet, kerrosrakenne ja tietovarasto [Gar94]. Eri laadulliset vaatimukset toteutuvat eri arkkitehtuurityyleillä eri tavoin ja tätä tietoa voidaan käyttää hyväksi suunniteltaessa kehitettävänä olevan ohjelmiston arkkitehtuuria.

4.4 Prosessikehykset

Ketteryyden säilyttämisestä on tullut tavoiteltava asia monille uusille prosessimalleille, silloinkin kun projektin koko ja maantieteellinen hajautuminen ovat suuria. Software Factories -konseptissa esitelty prosessikehyksen käsite ei ole poikkeus. Prosessikehyksellä tarkoitetaan rakennetta, jolla sovelluskehityksen osatoiminnot organisoidaan, jotta tuoteperheen mukaisia tuotoksia voidaan tuottaa tehokkaasti. Prosessikehyksellä pyritään tehostamaan nimenomaan ohjelmistotuoteperheiden kehittämistä.

Prosessikehys tarjoaa kontekstinmukaista opastusta ja asettaa rajoituksia sen sijaan että se pakottamalla pakottaisi kehittäjiä tiettyyn prosessiin. Prosessikehys hajottaa ohjelmistoprosessin *mikroprosesseihin*, jotka liittyvät järjestelmästä luotuihin näkymiin samaan tapaan kuin arkkitehtuurin kuvauksessa tehtiin (ks. edellinen luku 4.3.4). Jokainen mikroprosessi

kuvaa prosessin, jolla tietyn näkymän mukaiset artefaktit konstruoidaan. Prosessi asettaa sekä esi- että jälkiehtoja artefaktien kehittämiseen.

Prosessikehys tarjoaa siis joukon mahdollisia prosesseja, joita tietyn sovellustuoteperheeseen liittyvän sovelluksen kehityksen aikana käytetään, sen sijaan että se kuvaisi universaalin prosessin, jota käytetään kaikkialla.

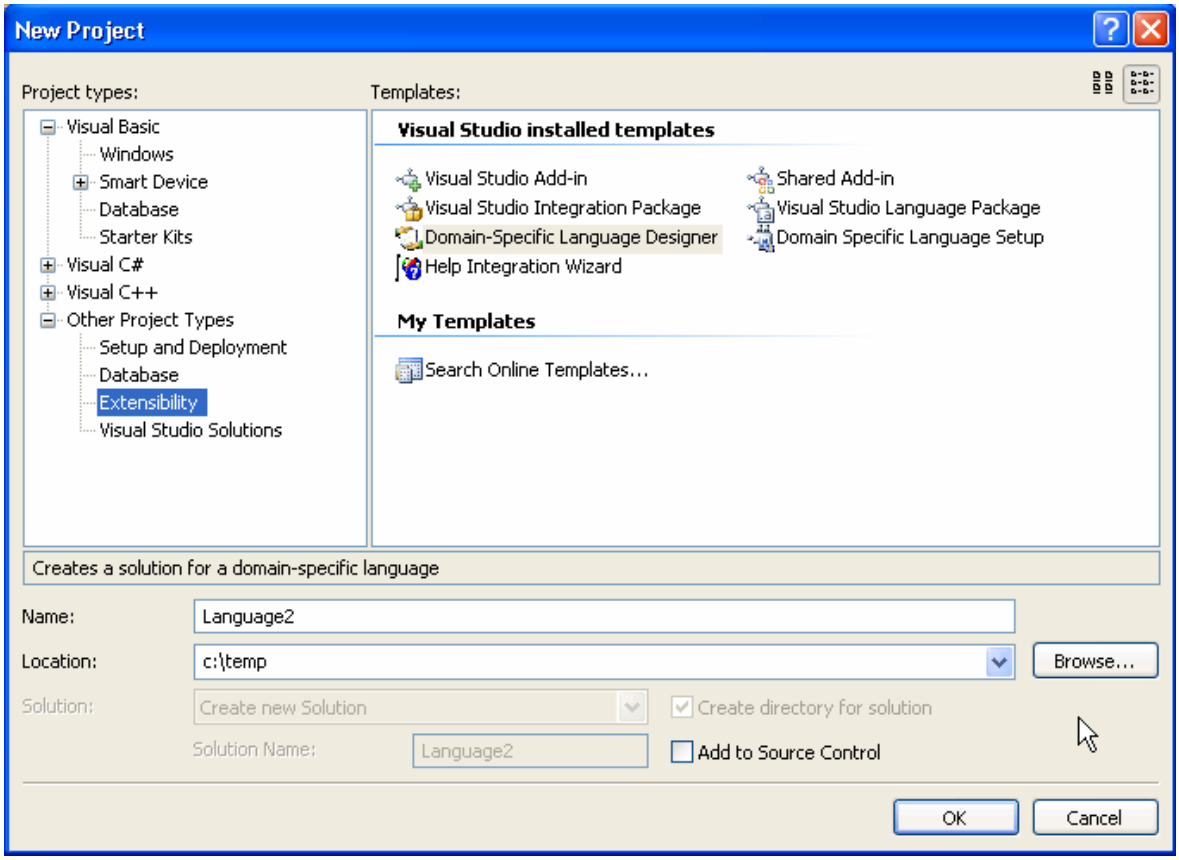
Prosessikehys voi olla esimerkiksi kolmitasoarkkitehtuuria noudattavan tuoteperheen osalta sellainen, että se sisältää kolme osaprosessia (mikroprosessia): käyttöliittymän toteuttaminen, liiketoimintalogiikan toteuttaminen ja tietovarastojen käsittelyn toteuttaminen. Todellisuudessa mikroprosesseja voi olla paljon useampia ja kukin niistä voi jakautua taas useaan pienempään osaprosessiin. Tietovarastojen käsittelyn osaprosessi voi jakautua esimerkiksi loogisen ja fyysisen tietomallin toteuttamiseen, optimointiin sekä hallintaan ja ylläpitoon.

Kuten mallinnusmenetelmien tapauksessa, myös prosessin osalta Software Factories -konsepti korostaa tiettyyn sovellusalueeseen ja – arkkitehtuuriin räätälöityä prosessia. Esimerkiksi räätälöimätöntä Unified Process -prosessimallia kritisoidaan muun muassa siitä, että sen tarjoama ohjaus on usein itsestään selvää kokeneelle käyttäjälle eikä tarpeeksi konkreettista noviisille.

4.5 DSL Tools

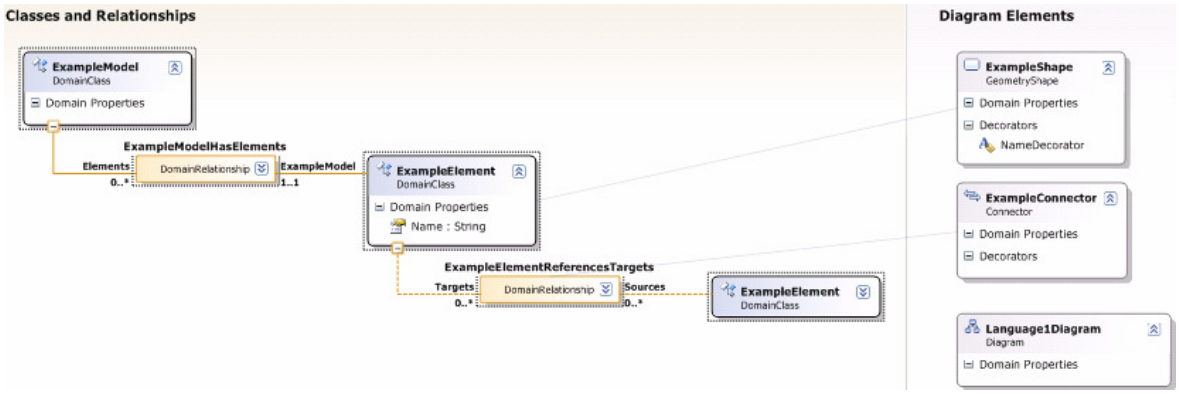
DSL Tools on tällä hetkellä tärkein Software Factories -konseptin mukainen olemassa oleva työkalu. Sen avulla voidaan kehittää Visual Studio 2005-kehitysympäristöön integroituja sovellusaluekieliä sekä mallinnustyökaluja. DSL Tools on osa Visual Studio SDK:ta ja tällä hetkellä (syyskuu 2006) on juuri julkaistu sen ensimmäinen ei-beeta versio 1.0. Tämän tutkielman käytännön osuuden mallinnustyökalu toteutetaan DSL Tools-työkalulla (ks. luku 7.6).

DSL Tools tuo Visual Studio 2005-kehitysympäristöön uuden projektitemplaten, jonka avulla uudet sovellusaluekielet kehitetään (ks. kuva 7).



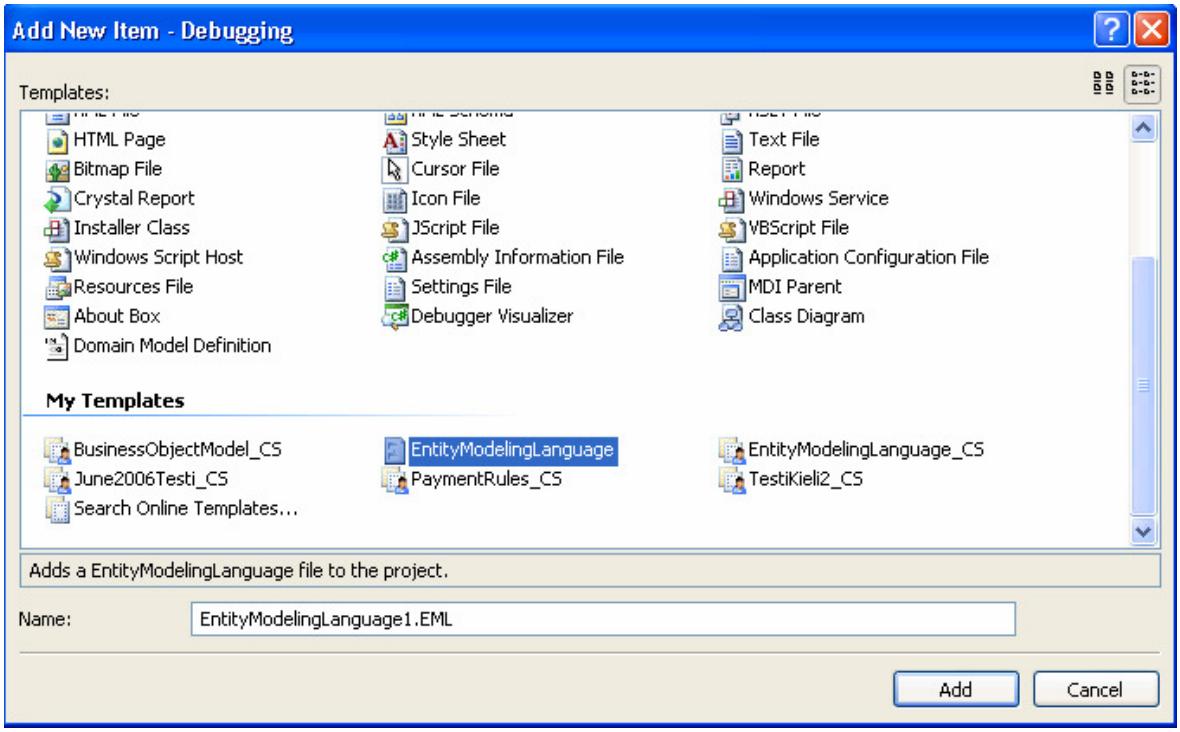
Kuva 7, DSL Tools-projektitemplate

Metamallinnus DSL Tools-työkalulla suoritetaan rakentamalla *Domain Class* -elementeistä puu luomalla työkalun piirtopinnalla näiden välille suhteita. DSL Tools metamallinnuskielen Domain Class -elementtien väliset erityyppiset suhteet ovat *Reference*, *Embedding* ja *Inheritance*. Graafinen piirtotyökalu määritellään linkkaamalla mallinnuskielen vielä abstraktit elementit konkreettisiksi kuvioiksi (*Shape*) sekä yhdistimiksi (*Connector*). Kuvioilla ja yhdistimillä voidaan luoda graafisia malleja (*Diagrams*) lopullisella työkalulla. Kuvassa 8 on minimaalinen sovellusaluekieli ja siihen liittyvät diagrammielementit.



Kuva 8, DSL Tools työkalulla kehitetty metamalli

DSL Tools -työkalulla kehitetty sovellusaluekieli käännetään sovellusaluekielipaketiksi, joka rekisteröidään omaksi tiedostotyypikseen. Näin ollen mihin tahansa olemassa olevaan projektirakenteeseen voidaan lisätä DSL Tools -työkalulla kehitettyjä designereitä, kuten kuvassa 9 näkyy.



Kuva 9, Rekisteröity tiedostotyyppi EML

DSL Tools mahdollistaa sovellusaluekielen ja mallinnustyökalun kehittämisen lisäksi generaattoreiden kehittämisen template-tekniikalla. DSL Toolsin generaattoriteknologiasta on kerrottu tarkemmin luvussa 5.2.

4.6 Software Factories ja UML

UML-mallinnusmenetelmän käyttöönotto ei ole oleellisesti muuttanut ohjelmistokehityksen työtapoja saati parantanut ohjelmistokehityksen tuottavuutta [Ms05]. Itse asiassa varsin harva kehittäjä käyttää UML-mallinnusta varsinaisen kehitystyönsä tukena ja suurin osa näistä käyttää luokkakaaviota. Luokkakaavioiden käyttö ohjelmakoodin generointiin on niin ikään vähäistä. Pääasiallisin käyttö UML-kaavioille on erilaiset hahmotelmat sekä kommunikaation tuki.

UML:ssä esiintyvät konseptit ovat abstraktiotasoltaan lähellä ohjelmiston toteutusta. Voidaan ajatella, että esimerkiksi luokkakaavion generointi ohjelmakoodiksi ei tarjoa erityistä lisäarvoa kehittäjille, koska muunnoksessa luokkakaavio kuvautuu lähes yksi-yhteen ohjelmakoodiksi. UML-mallinnusta käytetäänkin ensisijaistesti ohjelmiston teknisen dokumentaation tukena, luonnoksissa sekä ohjelmiston teknisen arkkitehtuurin hahmotelmissa [Ms05].

Sovellusaluekielten käyttäminen soveltuu paremmin tilanteisiin, jossa mallilta odotetaan muutakin kuin dokumentaatioarvoa. Johtuen Microsoftin vahvasta panostuksesta sovellusaluemallinnukseen on UML-mallinnustyökalu eriytetty kokonaan Visual Studio-kehitysympäristöstä omaksi erilliseksi tuotteekseen (Microsoft Visio[14]).

4.7 Software Factories -konseptin arviointia

Software Factories -konsepti tuntuu hieman hajanaiselta kokoelmalta ideoita ja teknologioita lukuun ottamatta sovellusaluemallinnukseen liittyvää osuutta. Varsinkin tämän luvun päälähteenä käytetty [Gre04a] on paikoin hieman jäsentymätön ja vaikealukuinen kirja

[14] www.microsoft.com/office/visio/

eikä tätä auta erittäin suuri sivumäärä. Tärkeää on myös huomata, ettei konseptin mukaisia toteutuksia ole juuri vielä olemassa.

Software Factories on Microsoftin kehittämä visio. Tämän huomaa erityisesti siitä, että siinä korostetut asiat ovat varsin sopusoinnussa Microsoftin sovelluskehitystuotteiden strategiaan. Konsepti tukeutuu vahvasti .NET-teknologiaan, kolmitasoarkkitehtuuriin sekä Web Services -hajautustekniikkaan. Software Factories -konseptin tuomista ideoista (samoin kuin Visual Studio 2005:n uusista ominaisuuksista) hyötyykin eniten ohjelmistotalo, joka kehittää Enterprise-tason kolmitasosovelluksia Microsoft .NET -teknologialla.

Software Factories -konsepti ei (sekään) näin ollen esittele kaikenkattavaa hopealuotia, jolla ohjelmistokehityksen tuottavuus ja laatu saataisiin universaalisti kohenemaan. Kuten luvussa 3 totesimme, on sovellusaluemallinnuksen tehokkaalle käyttöönotolle edellytyksenä se, että organisaatiolla on riittävä kokemus kohdealueesta sekä tietämys siitä, miten kohdealueelle kannattaa laadukkaita ohjelmistoja kehittää ja tämä osaltaan vähentää konseptin universaalia luonnetta.

5 Sovellusaluemallien muuntaminen

Sovellusaluekielten käyttöön liittyy oleellisena osana *muuntimet* eli *generaattorit*, ts. automaattiset työkalut, joiden avulla sovellusaluemallin perusteella voidaan generoida tuotoksia, kuten ohjelmakoodia, SQL-lauseita, dokumentaatiota, testitapauksia, yksikkötestejä ja niin edelleen. Sovellusaluemallin perusteella on mahdollista generoida periaatteessa mitä tahansa, ainoa rajoitus generoitaville tuotoksille on sovellusaluekielen ilmaisuvoima. Yleisin generoitava tuotos on luonnollisesti ohjelmakoodi.

Tämän tutkielman käytännön osuudessa toteutettavan Entity Modeling Language-mallinnuskielen yhteydessä kehitetään seuraavia artefakteja tuottavia generaattoreita:

- Visual Basic .NET – ohjelmakoodia
- SQL skriptejä tietokannan rakenteeseen ja sisältöön liittyen
- Resurssitiedostoja monikielisyyden tukemiseen
- Dokumentaatiota HTML-muodossa

Jack Herringtonin mukaan automaattisen ohjelmakoodin generoinnin ansiosta saavutetaan seuraavia oleellisia hyötyjä [Her03]:

- *Tuottavuutta* voidaan parantaa huomattavasti, koska koodingeneroijat voivat luoda satoja luokkia sekunneissa. Myös tehdyt muutokset leviävät generoijien avulla nopeasti koko järjestelmään. Näin saavutetaan tuottavuuden taso, johon ei päästä käsin ohjelmoimalla. Tutkielman käytännön työn taustana on huomio siitä, että esimerkiksi lisättäessä yhdelle liiketoimintaluokalle attribuuttia, on muutos toteutettava käsin kirjoittaen keskimäärin 15 paikkaan ohjelmakoodissa.
- *Laatu* on generoijien muodostaman koodin osalta tasainen. Jos järjestelmästä löydetään virhe, voidaan korjaava muutos suorittaa koko järjestelmän osalta yhdellä koodingeneroijan tekemällä kierroksella. Koodingenerointi tukee myös yksikkötestausta mahdollistamalla automaattisten testitapausten muodostamisen.

- Generoijalla luotu koodi on *yhtenäistä*, koska esimerkiksi muuttujien ja funktioiden nimeäminen on johdonmukaista. Tämä mahdollistaa jälkeenpäin järjestelmän helpomman ymmärtämisen sekä suoraviivaisen kehittämisen.

Steven Kelly korostaa generaattoreiden kehittämisessä olemassa olevan koodin merkitystä [Kel06]. Generaattorin kehittäminen kannattaa hänen mukaansa aloittaa takaperoisesti tarkastelemalla valmista, mahdollisesti tuotantokäytössä olevaa ohjelmakoodia. Käytettäessä olemassa olevaa ohjelmakoodia referenssinä voidaan varmistua siitä, että generoitu ohjelmakoodi on vähintään yhtä laadukasta kuin käsinkirjoitettu. Kuten esimerkiksi luvussa 4 kuvattu *Software factories -pattern* esittää, ennen kuin sovellusaluemallinnusta hyödyntävää ympäristöä voidaan lähteä rakentamaan, tulee olla valmiita tuotteita, joiden perusteella automatisointia voidaan kehittää. Näin ollen generaattoreiden kehittämisen tueksi on aina referenssikoodia saatavilla. Tämän tutkielman käytännön osuuden generaattorien kehitystä helpottaa se, että kohdekehystä hyödyntävää, tuotantokäytössä olevaa ohjelmakoodia on saatavilla.

Kellyn mukaan koodigeneraattorit toimivat tehokkaimmin, mikäli ne generoivat suoraan ajettavaa ohjelmakoodia. Vaikka nykyiset metamallinnustyökalut tukevatkin osittain mallien muuntamista toisiksi malleiksi, ei se useinkaan loppujen lopuksi ole järkevää redundanttisuuden aiheuttaman ylläpito-ongelman vuoksi [Kel06]. Koodigeneraattorit toimivat aina yhteen suuntaan, ja näin ollen mallien välinen synkronointi tulee suureksi ongelmaksi. Tämä onkin Kellyn mukaan juuri yksi suurimmista ongelmista MDA-lähestymistavassa.

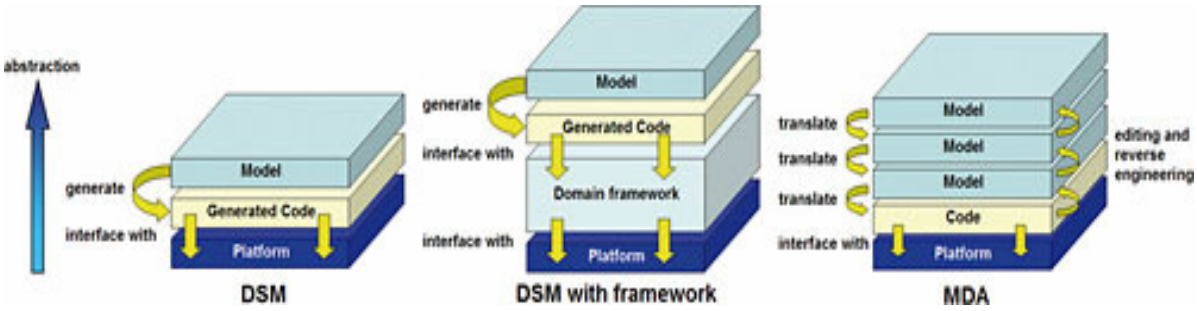
MDD – pattern nimeltä "*Produce nice-looking code…wherever possible*" kehottaa kehittämään generaattorit aina sitä silmällä pitäen, että niiden generoima koodi olisi mahdollisimman luettavaa ja hyvien ohjelmointikäytäntöjen mukaista [Völ04]. Generoitua koodia silmäillään kuitenkin ennen pitkää kehittäjien toimesta, joten hyvien kommentointi- ja nimeämiskäytäntöjen noudattaminen helpottaa elämää myös generoidun koodin tapauksessa.

5.1 Generoitavan koodin abstraktiotaso

Kuten luvussa 3.2 todettiin, perinteisesti sovellusaluemallinnusta tukevaan ympäristöön kuuluu kohdekehys, jota hyödyntävää ohjelmakoodia sovellusaluemallien perusteella gene-

roidaan. Näin on myös tämän tutkielman käytännön osuudessa kehitettävän Entity Modeling Language -sovellusaluekielen tapauksessa. Kohdekehyksen olemassaolo ei tietysti ole pohjimmiltaan välttämätöntä, generaattorithan voidaan rakentaa tuottamaan mitä tahansa.

Martijn Iseger onkin esittänyt kuvassa 10 näkyvän karkean jaon liittyen generoitavan koodin abstraktiotasoon [Ise05].



Kuva 10, Generoitavan koodin abstraktiotasot [Ise05]

5.1.1 Pelkkä DSM

Generaattorit voivat tuottaa ohjelmakoodia, joka kutsuu suoraan käytettävän ohjelmointikielen tarjoamien kirjastojen palveluja ilman varsinaista kohdekehystä. Tällöin generaattori sisältää varsin suuren määrän informaatiota ja sen kehittäminen on varsin haastavaa sovellusalueeseen liittyvän kohdekehyksen puuttuessa.

5.1.2 DSM ja kohdekehys

Perinteisessä mallissa generaattorit tuottavat ohjelmakoodia, joka kutsuu erityisen sovellusaluekohtaisen kohdekehyksen tarjoamia palveluja. Tämä on varmasti yleisin lähestymistapa hyödynnettäessä sovellusaluemallinnusta todellisessa ohjelmistokehityksessä. Kuten luvussa 4.1 todettiin, myös Software Factories -konseptissa on keskeisenä ajatuksena uudelleenkäytettävien komponenttien aktiivinen kehittäminen ja hyödyntäminen. Tästä syystä kehitettäessä sovellustuoteperhettä, on lähes aina käytettävissä jonkinlainen kohdekehys, joka tarjoaa mahdollisesti sovellusaluekohtaisia tai pelkästään teknisiä palveluja sovellustuoteperheen sovellusten kehittämisen tueksi.

5.1.3 MDA

OMG:n MDA- lähestymistavassa on erityispiirteenä mallien väliset muunnokset. MDA:ssa pyritään myös kunnianhimoisesti alustariippumattomuuteen. UML-mallinnuskieltä hyödyntävässä MDA:ssa ylimmän abstraktiotason mallina toimii alustariippumaton *Platform Independent Model* (PIM) ja tästä generoidaan automaattisen muunnoksen avulla alustariippuvainen *Platform Specific Model* (PSM) [Muk03]. Vasta PSM:n pohjalta generoidaan varsinainen ohjelmakoodi, joka voi tukeutua sovellusaluespesifiin kohdekehykseen tai sitten ei. Mallien muuntamisen ja serialisoinnin tueksi OMG on kehittänyt oman standardinsa nimeltään *Meta Object Facility*.

5.2 Template-pohjainen generointi

Microsoft DSL Tools (ks. luku 4.5) tukee generaattorien kehitystä template-menetelmällä. Siinä generaattori kirjoitetaan yhdistelemällä kiinteitä generoitavia lohkoja ja ohjelmointikielellä (esimerkiksi C# tai VB.NET) kirjoitettuja skriptejä. Tästä menetelmästä tekee erittäin tehokkaan se, että yleiskäyttöisen ohjelmointikielen ilmaisuvoima on lähes rajaton kirjoitettaessa generaattoria. DSL Tools tarjoaa generaattorin skriptaukseen mahdollisuuden navigoida sovellusaluemallia saman ohjelmointikielen puitteissa. MetaEdit+ -työkalussa (ks. luku 3.7.1) generaattorien kehittämisen suurimmaksi pullonkaulaksi olen havainnut juuri generaattorien kehityskielen ilmaisuvoiman heikkouden.

Seuraava esimerkki selventää template-pohjaisen generoinnin filosofiaa. Siinä esitellään osa tutkielman käytännön osuudessa kehitettävän mallinnustyökalun SQL-skriptigeneraattorista. Esimerkissä näkyy se, että TemplatingEngine, joka on DSL Toolsin generointiteknologian taustalla, tulkitsee ”<# #>” - tagien sisällä olevan tekstin ohjelmakoodiksi TransformText-metodin sisällä. ”<#+ #>” - tagien sisällä oleva teksti tulkitaan ko. metodin ulkopuoliseksi ohjelmakoodiksi ja näin mahdollistetaan muun muassa omien funktioiden ja metodien määrittely kuten lopussa oleva WriteFieldDefinition-metodi.

```
<#@ import NameSpace="Riston.Gradu.EntityModelingLanguage"#>
<#@ import NameSpace="Riston.Gradu"#>
<#
Dim entity as Entity
For Each entity in Me.EntityModel.Entities
#>
--<#= entity.Description #>
CREATE TABLE TABLE_<#=entity.DBTableName#> {
<#
For i As Integer  = 0 To entity.Attributes.Count - 1
    Dim attrib As EntityModelingLanguage.Attribute
    attrib = Ctype(entity.Attributes(i), EntityModelingLanguage.Attribute)
    Dim typeName As string = attrib.Type

    WriteFieldDefinition(attrib, entity, i)

Next
#>


<#+
Private Sub WriteFieldDefinition( _
       attrib As EntityModelingLanguage.Attribute, _
       entity As EntityModelingLanguage.Entity, _
       i As Integer)

    Write("" +attrib.DBFieldName)
    If attrib.DBFieldName.Length < 8 Then Write(" ")
    write("    " + GetDBType(attrib))
    If Not attrib.IsNullable Then
        Write("NOT NULL")
    Else
        Write("NULL")
    End If
    'If i < entity.Attributes.Count - 1 Then
    Write(",")
    'End If
    Write(Microsoft.VisualBasic.VBCrLf)
End Sub
#>
```

Tämänkaltaisessa skriptauksessa mahdollistetaan lähestulkoon kaikki ohjelmointikielen sallimat menetelmät. Tosin DSL Tools –työkalussa on joitain rajoituksia muun muassa rekursioon liittyen.

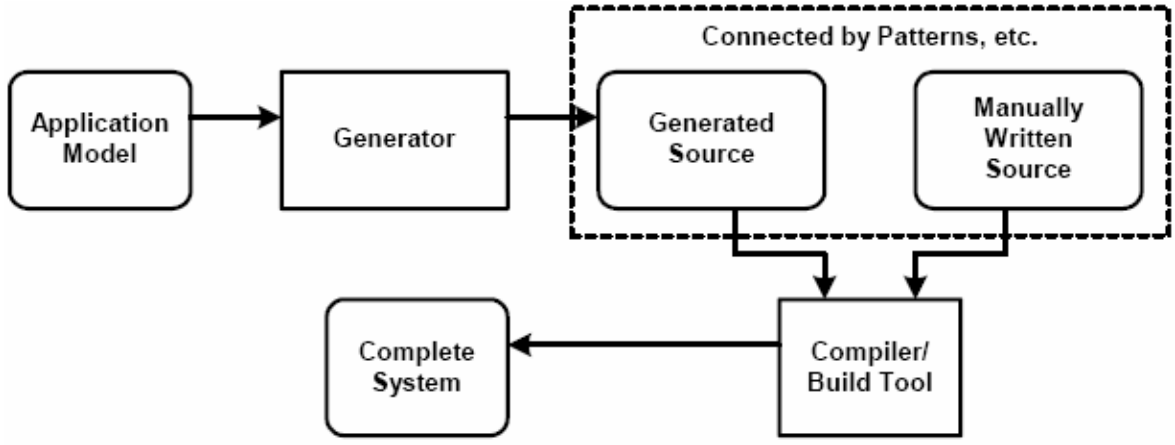
5.3 Generoidun koodin laajentaminen

Koodigeneraattorin voidaan ajatella toimivan periaatteessa samoin kuin ohjelmakoodin kääntäjä, joka taas tuottaa ajoympäristön ymmärtämää varsin matalan tason koodia. Nykyisten ohjelmointikielten kääntäjien lopputulokseen ei tarvitse oikeastaan ikinä koskea ja samaa periaatetta pyritään toteuttamaan sovellusaluekielten generaattoreissakin. Mikäli käytössä havaitaan, että generoitua koodia on pakko paikkailla käsin, voidaan muuntaa joko sovellusaluekieltä lähemmäksi ongelma-avaruutta tai kehittää generaattoria [Kel06].

Tämän tutkielman käytännön osuudessa on tavoitteena kehittää generaattoreita, joiden lopputulokseen ei tarvitse koskea. Vaikka generaattorit eivät tuottaisikaan kaikkea ohjelman lähdekoodia, on tärkeää, että niiden tuottama koodi pysyy koskemattomana ja käsin laajentaminen mahdollistetaan kohdekehyksen muiden laajentamismekanismien avulla.

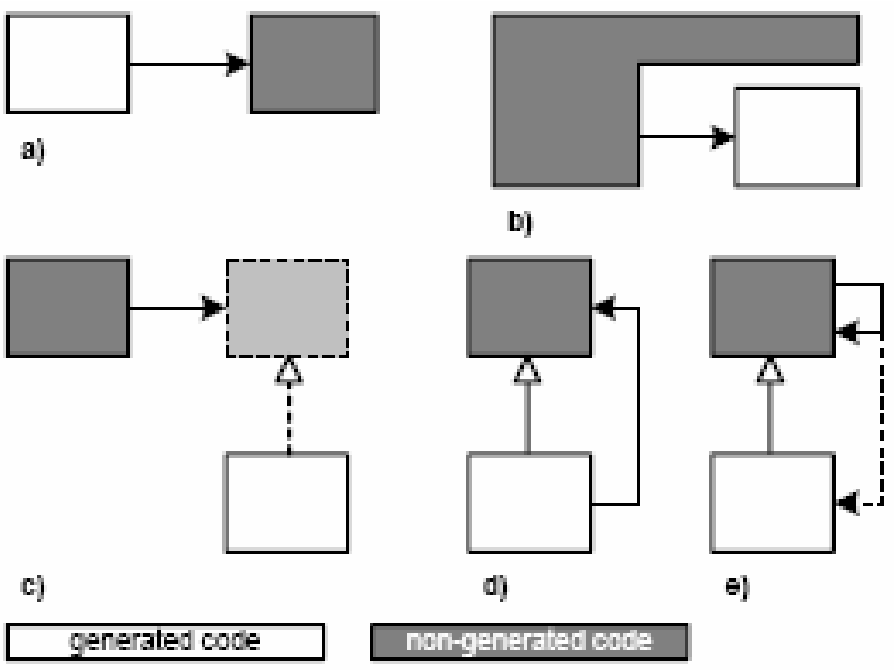
Microsoft .NET Framework 2.0 mahdollistaa luokkien määrittelemisen useassa paikassa, jopa useassa lähdekooditiedostossa erityisen *partial*-avainsanan avulla. Tämän mekanismin avulla on helppo erottaa generoitu ja käsikirjoitettu ohjelmakoodi. Muun muassa Visual Studio 2005:n näytönpiirtotyökalu osaa piilottaa generoidun koodin eri tiedostoon kehittäjältä piiloon ja sallii kehittäjän laajentaa generoituja luokkia juuri tällaisilla osittain määritellyillä luokilla. Koska tutkielman käytännön osuudessa kehitettävä mallinnustyökalu tuottaa Microsoft .NET Framework 2.0:aa hyödyntävää ohjelmakoodia, on generaattoreiden kehittämisessä hyödynnetty *partial*-määriteltyjä luokkia.

Markus Völter ja Jorn Bettin ovat määritelleet generaattoreiden tuottaman ja käsinkirjoitetun ohjelmakoodin erottamisen omaksi MDD-suunnittelumallikseen *"Separate generated and non-generated code"* [Völ04]. Tämän saavuttamiseksi modernit olio-ohjelmointikielet tarjoavat keinoja käsinkirjoitetun koodin erottamiseksi generoidusta edellä mainitun partial-luokkien lisäksi rajapintojen ja muun muassa *tehdas*, *silta* ja *tehdasmetodi*-suunnittelumallien avulla [Gam95]. Kuvassa 11 on havainnollistettu generoidun ja käsinkirjoitetun koodin sijoittumista sovellusaluemallinnusta hyödyntävässä ympäristössä.



Kuva 11, Generoidun ja käsinkirjoitetun koodin erottaminen [Völ04]

Kuvassa 12 esitetään kuinka generoitu ja käsinkirjoitettu koodi voidaan erottaa käyttäen muun muassa edellä mainittuja suunnittelumalleja.



Kuva 12, Generoidun koodin laajennusmekanismit [Völ04]

Generoitu koodi voi kutsua erityisissä kirjastoissa olevaa käsinkirjoitettua koodia (a). Tämä on erityisen hyödyllinen mekanismi silloin, kun halutaan hyödyntää mahdollisimman paljon aiemmin kehitettyjä ohjelmakomponentteja. Vastakkainen tapa on luonnollisesti myös mahdollinen (b), silloin käsinkirjoitettu ohjelmistokehys hyödyntää generoituja komponentteja. Tämän helpottamiseksi voidaan käsinkirjoitettu ohjelmakoodi kirjoittaa abstrakteja luokkia ja rajapintoja vastaan, jotka taas generoitu koodi toteuttaa (c).

Generoidut luokat voivat myös periytyä käsinkirjoitetuista, jolloin generoitu koodi voi kutsua käsinkirjoitettuja geneerisiä metodeja (d). Kantaluokka voi myös sisältää abstrakteja metodeja, joita generoidut luokat toteuttavat siten kuin *tehdasmetodi*-suunnittelumalli esittää (e) [Gam95].

6 Hyödynnettävä kohdekehys

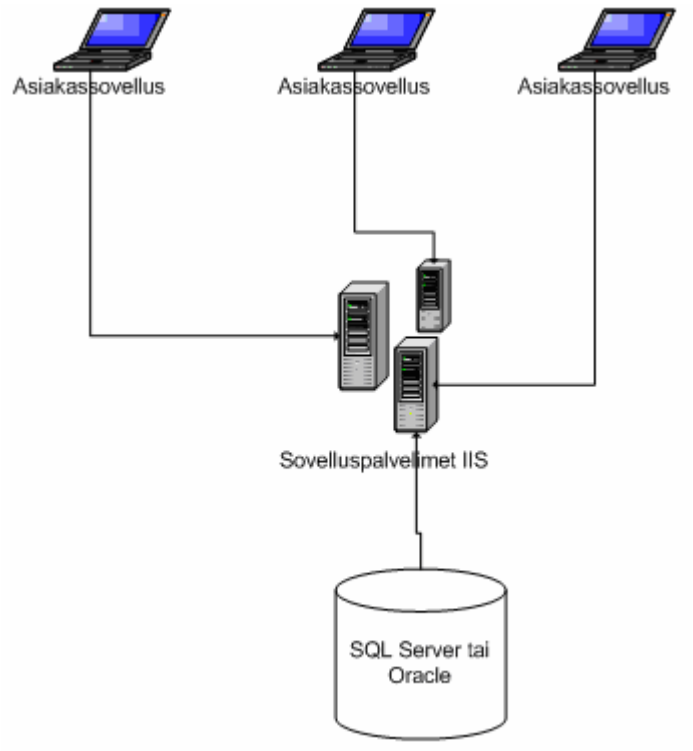
Tämän tutkielman käytännön osuuden kohdekehyksenä toimii Enterprise-tason kolmikerrossovellusten kehittämisen tueksi kehitetty sovellusalusta. Kehitettävä EML-sovellusaluekieli (ks. luku 7) sekä kehitettävät generaattorit osaavat hyödyntää sovellusalustan tarjoamia palveluja.

Sovellusalustaa hyödyntävä sovellustuoteperhe koostuu finanssialan Backoffice-sovelluksista[15]. Kohdekehyksen teknologialla on toteutettu muun muassa tilinhoitajayhteisöjärjestelmä, rahastoyhtiöiden osuusrekisterin hallintajärjestelmä sekä arvopaperikaupan Backoffice- prosessien hallintajärjestelmä. Niiden avulla käyttäjäorganisaatiot kuten tilinhoitajayhteisöt, rahastoyhtiöt sekä pankkiiriliikkeet voivat hallita omia liiketoimintaprosessejaan ja asiakkuuksiaan.

6.1 Sovellusalusta

Kohdekehyksenä toimiva sovellusalusta on kehitetty Microsoft .NET – teknologialla ja se tarjoaa joukon palveluja finanssialan sovellustuoteperheen tuotteiden kehittämiseen. Kehitettävät sovellukset koostuvat asiakassovelluksista, sovelluspalvelimesta sekä tietokannasta (ks. kuva 13).

[15] Samstock tuoteperhe: http://www.samstock.com/index.php?node_id=70



Kuva 13, Kohdekehyksen kolmitasoarkkitehtuuri

Kohdekehyksenä toimiva sovellusalusta sekä sitä hyödyntävät sovellukset ovat toteutettu Visual Basic.NET-ohjelmointikielellä ja se toimii tällä hetkellä Microsoft .NET Framework versio 1.1:n päällä ja tullaan siirtämään .NET Framework versioon 2.0.

Kolmitasoarkkitehtuurin käytön etuja verrattuna normaaliin kaksitasosovellukseen on lueteltu seuraavassa:

- Tietokantayhteyksien hallinta: ASP.NET- sovelluspalvelin ylläpitää tietokantayhteyksiä ns. *Connection Poolin* avulla ja käyttää niitä tarvittaessa asiakassovellusten pyyntöjen toteuttamiseen. Koska asiakassovellukset eivät suorita koko aikaa tietokantaintensiivisiä operaatioita, voidaan tällä keinolla kierrättää yhteyksiä ja säästää tietokantaresursseja verrattuna siihen, että asiakassovellukset pitäisivät tietokantayhteyttä auki koko sovelluksen päälläoloajan.

- Skaalautuvuus: Sovelluspalvelin voidaan monistaa samaan tapaan, kuin www-palvelimet yleensä, jolloin järjestelmä voidaan liittää enemmän asiakassovelluksia.

- Monitoroitavuus: Nykyiset www-palvelimet tarjoavat joukon työkaluja, joiden avulla voidaan monitoroida sekä analysoida sovelluspalvelimen kuormitusta ja suorituskykyä.

- Erilaiset asiakassovellukset: Sovelluspalvelin tarjoaa ulospäin Web Services -rajapinnan, jota voidaan hyödyntää niin Windows-sovelluksista kuin esimerkiksi www-sovelluksista selainohjelmista käsin.

- Integroitavuus: Sovelluspalvelin voi tarjota dokumentoidun ja julkisen rajapinnan järjestelmään muiden, ulkoisten järjestelmien käyttöön.

Taulukko 1 esittää sovellusalustan avulla kehitettyihin sovelluksiin liittyvät abstraktiotasot.

Sovellus
Sovellusalusta
Microsoft .NET Framework
Win32 API
Windows 2000/XP/2003
Laitteisto

Taulukko 1, Sovellusalustaan liittyvät abstraktiotasot

Hyödynnettävä sovellusalusta ei tarjoa varsinaisesti finanssialan sovellusaluekohtaisia palveluja. Tässä mielessä se ei ole sovellusaluekohtainen kohdekehys siten kuin lähteissä [Poh2], [Gre04a] ja [Met05a] esitetään. Sen sijaan sovellusalustaa voi ajatella yleisemmän tason sovellusaluekohtaisena kohdekehyksenä, missä sovellusalueena on Enterprise-tason tietojärjestelmä, jossa käsitellään relaatiomuotoista dataa.

6.2 Asiakassovellukset

Kehitettävät asiakassovellukset ovat ns. rikkaita Windows-sovelluksia. Niiden tehtävänä on tarjota käyttäjälle pääsy järjestelmään ja välittää käyttäjän pyynnöt sovelluspalvelimel-

le. Sovelluksen näytöt noudattavat yhteistä käyttöliittymästandardia, ja näin ollen käyttäjälle tarjotaan yhtenäinen käyttökokemus riippumatta siitä mitä sovellustuoteperheen sovellusta hän käyttää. Esimerkiksi haku- ja muokkausnäytöillä on yhtenäinen ulkoasu, mikä helpottaa sovellusten intuitiivista käyttöä. Asiakassovellukset sisältävät myös jonkin verran liiketoimintalogiikkaa, muun muassa tietojen validointiin sekä tarkistuksiin liittyen.

Hyödynnettävä sovellusalusta tarjoaa muun muassa seuraavia palveluja asiakassovellusten kehittämiseen:

- Konfiguraationhallinta: sovellusalusta sisältää konfiguraatiotiedoston hallintatoiminnot, joilla asiakassovellukset voivat lukea ja kirjoittaa niitä kiinnostavia konfiguraatioasetuksia.

- Monikielisyys: Sovellusalusta osaa käyttäjän tietojen perusteella esittää käyttöliittymän joko englanniksi, ruotsiksi tai suomeksi.

- Tietojen haku- ja muokkausnäytöt: Sovellusalusta tarjoaa valmiita komponentteja haettujen tietueiden näyttämiseen, muokkaamiseen ja poistamiseen liittyen.

- Sessionhallinta: Asiakassovellukset liittyvät sovelluspalvelimeen autentikoidusti ja sovellusalusta huolehtii muun muassa sovelluspalvelimen Web Services -edustaolioiden luomisesta.

- Liiketoimintaolioiden kantaluokat: kohdekehyksessä kukin liiketoimintaolio kuten esimerkiksi *Asiakas* periytyy yhteisestä kantaluokasta. Niin ikään liiketoimintaolioiden kokoelmaluokille tarjotaan yhteinen kantaluokka. Tämä mahdollistaa yhtenäisen tavan käsitellä liiketoimintaolioille yhtenäisiä piirteitä.

6.3 Sovelluspalvelin

Sovelluspalvelin on Microsoftin Internet Information Services- www-palvelimessa[16] suoritettava tilaton ASP.NET-sovellus. Se tarjoaa Web Services -rajapinnan asiakassovellusten sekä ulkoisten liittymien käyttöön. Sovelluspalvelimen tehtävänä on suorittaa varsinainen liiketoimintalogiikka, asiakassovellusten ja ulkoisten liittymien pyyntöjen perusteella. Sovelluspalvelin on järjestelmän ainoa komponentti, joka kommunikoi tietokantapalvelimen kanssa.

Sovellusalusta tarjoaa sovelluspalvelinten kehittämiseen seuraavia palveluja:

- Konfiguraationhallinta: Sovelluspalvelimia voidaan konfiguroida erillisellä konfiguraatiotiedostolla, ja sovellusalusta tarjoaa palveluja konfiguraatiodatan käsittelyyn.

- Monikielisyys: Sovelluspalvelimelta tulevat tekstit kuten esimerkiksi virheilmoitukset osataan esittää joko englanniksi, ruotsiksi tai suomeksi käyttäjän kielestä riippuen.

- Tietokantayhteys: Sovellusalusta tarjoaa ohjelmoijille helpon tavan päästä käsiksi järjestelmän tietokantaan. Tietojen lisäämiset, poistamiset ja muuttamiset sekä tietokantakyselyt suoritetaan sovellusalustan avulla, eikä ohjelmoijan tarvitse välittää esimerkiksi siitä onko käytettävä tietokanta Oracle vai SQL Server.

- Lokitus: Sovelluspalvelimen ohjelmakoodista on helppoa kirjoittaa sovelluspalvelimen lokitiedostoon sovellusalustan tarjoamien toimintojen avulla.

- Liiketoimintaolioiden kantaluokat: Myös sovelluspalvelimella kukin liiketoimintaolio periytyy yhteisestä kantaluokasta. Niin ikään liiketoimintaolioiden kokoelmat periytyvät yhteisestä kantaluokasta.

---

[16] http://www.iis.net/

- Pysyvyyspalvelut: Sovelluspalvelimen liiketoimintaoliot huolehtivat itse tilansa kuvaamisesta relaatiotietokantaan (*load*, *save* ja *delete* – toiminnot). Näissä käytetään sovellusalustan tarjoamia palveluja. Lisäksi sovellusalusta tarjoaa kokoelmaolioille palveluja, joiden avulla on kokoelmiin on helppo toteuttaa hakutoiminnallisuuksia erilaisin hakukriteerein.

6.4 Tietokanta

Kehitettävien sovellusten tietokantana toimii joko Oracle tai SQL Server. Tietokannan tyyppi on otettava huomioon järjestelmää kehitettäessä. Eräs eroavaisuus on se, että Oracle-tietokannassa *identity*-tyyppinen pääavain tulee hoitaa erillisen sekvenssin avulla, kun taas SQL Serverissä sama asia voidaan antaa tietokannan hoidettavaksi. Tämä on luonnollisesti otettava huomioon tutkielman käytännön osuudessa kehitettävien tietokantalauseita tuottavien generaattorien kehityksessä (ks. luku 7.7).

7 Entity Modeling Language

Tässä luvussa esitellään sovellusaluelähtöisen tietomallinnuksen mahdollistava sovellusaluekieli EML (*Entity Modeling Language*) sekä sen kehitysprosessi. Kielen avulla voidaan rakentaa rahoitusalalle kehitettävien sovellusten tietomalleja. Sen sijaan, että sovelluksen tietomalli esitettäisiin geneerisellä tavalla esimerkiksi ER-mallinnusta käyttäen, voidaan EML-kielellä esittää sovelluksen tietomallia osittain sovellusalueen erityispiirteet huomioiden. EML-kieli hyväksikäyttää sekä sovellusalueen tietämystä, että sovellusten alustana olevan kohdekehyksen arkkitehtuurin ominaisuuksien tuntemista. Näin ollen sitä voidaan pitää sovellusaluekielenä siten kuin luvussa 3.3 määriteltiin.

ER-mallinnusta tarkemman tietomallinnuksen lähtökohtana on havainto siitä, että rahoitusalan ohjelmistoissa on monia entiteettejä, joilla on samoja piirteitä ja joita käytetään samantyyppisesti. EML-kielen periaatteena on ER-mallinnuksessa esiintyvien elementtien erikoistaminen. Tällä mahdollistetaan se, että EML-mallin perusteella on mahdollista generoida esimerkiksi ohjelmakoodia tarkemmalla tasolla kuin mitä pelkän ER-mallin perusteella voidaan tehdä. EML-kielen syntyprosessista on kerrottu luvussa 7.2.

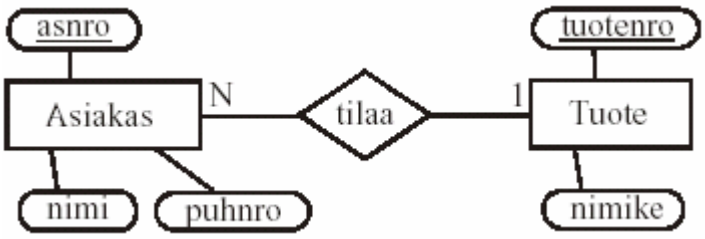
7.1 Taustana ER-mallinnus

ER-mallinnusta käytetään tietokantojen, erityisesti relaatiotietokantojen suunnittelutyökaluna. ER-mallissa kuvataan tietokannan tieto-objektit ja niiden väliset yhteydet. Mallin perusteella määritellään tietokannan taulut ja taulujen perus- sekä viiteavaimet (ja mahdolliset muut mallista näkyvät rajoitukset) [Ket02].

Er-mallinnuksen peruselementtejä ovat *kohteet*, *attribuutit* ja *yhteydet* [Tha00]. Kohteet eli *käsitteet* (*entity*) ovat perusobjekteja, joista tietoja kerätään. *Ominaisuudet* eli *attribuutit* (*property*, *attribute*) kuvaavat kohteen ominaisuuksia. Esimerkiksi kohteen Asiakas attribuutteja voisivat olla asiakasnumero, nimi ja puhelinnumero. Kohteen ilmentymät konkretisoituvat tiettyinä attribuuttien arvoina, esimerkiksi tietty asiakas konkretisoituu tiettynä asiakasnumerona, nimenä ja puhelinnumerona. Avainattribuutti yksilöi kohteen ilmentymän, esimerkiksi kohteen Asiakas luonnollinen avainattribuutti olisi asiakasnumero.

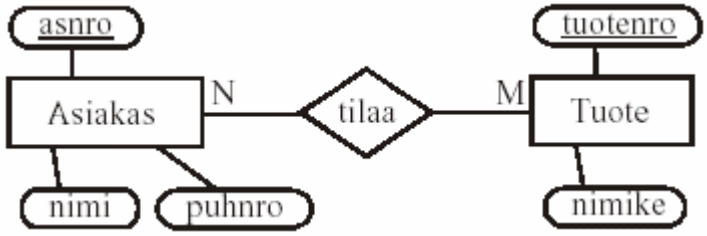
Avainattribuutteja voi olla myös monta, jotka yhdessä yksilöivät kohteen ilmentymän. Relaatiotietokannan taulussa attribuutit vastaavat sarakkeita ja avainattribuutit perusavainta.

Kohteiden väliset *yhteydet* (*relationships*) kuvaavat, miten kohteet liittyvät toisiinsa. Yhteydet piirretään tavallisesti vinoneliöinä, joiden sisään kirjoitetaan yhteyden nimi, ja yhdistetään viivoin kohteisiin. Viivojen yhteyteen merkitään 1 tai N (tai muu kirjain) kertomaan yhteystyyppi. Esimerkiksi kohteet Asiakas ja Tuote voivat olla yhteydessä sitä kautta, että asiakas tilaa tuotteita. Jos tilausyhteyden kautta yhteen asiakkaaseen eli yhteen Asiakas-kohteen ilmentymään voi liittyä vain yksi tuote (voi tilata vain yhtä tuotetta kerrallaan), ja yhteen tuotteeseen monta asiakasta (useampi asiakas saa tilata samaa tuotetta), kyseessä on *yhden-suhde-moneen* tyyppinen yhteys *(one-to-many*, ks. kuva 14*)*.



Kuva 14, Yhden-suhde-moneen – yhteys [Ket02]

Jos yhteen asiakkaaseen voi liittyä monta tuotetta tilauksen kautta, kyseessä on *monen-suhde-moneen* tyyppinen yhteys (*many-to-many*, ks. kuva 15).

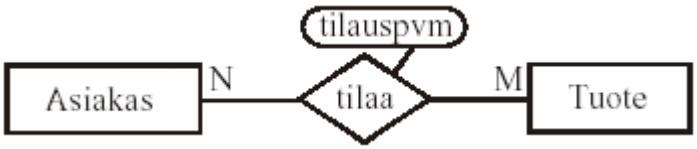


Kuva 15, Monen-suhde-moneen-yhteys [Ket02]

Kolmas yhteystyyppi on *yhden-suhde-yhteen* (*one-to-one*, ks. kuva 16). Esimerkiksi kohteiden Osasto ja Työntekijä välillä voisi olla johtaa-niminen *yhden-suhde-yhteen*-yhteys: yhteen osastoon liittyy johtamismielessä yksi työntekijä, yksi työntekijä johtaa vain yhtä osastoa.

Kuva 16, Yhden-suhde-yhteen-yhteys [Ket02]

Monen-suhde-moneen-yhteytellä voi olla myös attribuutteja. Esimerkiksi tietyn asiakkaan tietyn tuotteen tilaukseen liittyy tilauspvm-attribuutti (ks. kuva 17).



Kuva 17, Attribuutti monen-suhde-moneen-yhteydessä [Ket02]

7.2 ER-mallinnuksen käsitteiden erikoistaminen

Kuten edellä havaitsimme, ER-mallinnuksessa järjestelmän tietomallia mallinnetaan erittäin yleisellä tasolla. EML-kielen kehittämisen lähtökohtana on havainto siitä, että rahoitusalan tietojärjestelmissä on loppujen lopuksi varsin vähän aidosti erityyppisiä entiteettejä. Seuraavissa luvuissa 7.3-7.5 esitetyt luokittelut kehitettiin siten, että olemassa olevan tietojärjestelmän tietokantataulut sekä näiden väliset suhteet luokiteltiin niiden käyttötarkoituksen ja käyttäytymisen mukaan.

Liitteessä 3. on esitelty erittäin yksinkertainen esimerkki, jossa järjestelmän tietomalli on seuraava:

- Asiakkaalla (*Customer*) voi olla nolla tai useampia salkkuja (*Portfolio*). Salkulla voi olla nolla tai useampia tilejä (*Account*). Näillä kaikilla on esimerkin yksinkertaisuuden vuoksi vain vähän attribuutteja.

- Asiakas asuu jossain maassa (*Country*), josta tiedetään muun muassa valuutta ja henkilöveroprosentti. Asiakas edustaa jotain asiakastyyppiä (*CustomerType*) ja voi kuulua yhteen tai useampaan asiakasluokkaan (*CustomerClass*).

- Tili edustaa jotain tilityyppiä (*AccountType*), joita esimerkissä ovat kirjanpitotili ja pankkitili. Tilillä voi olla tilitapahtumia (*AccountTransaction*). Tilitapahtuma voi olla jossain seuraavista tiloista (*AccountTransactionStatus*): Avoin, kohdistettu tai poistettu.

Vaikka yllä esitellyssä esimerkissä on useita entiteettejä ja useita suhteita, on näissä havaittavissa selkeästi yhteisiä piirteitä. Asiakas, Salkku ja Tili edustavat eräänlaista *perustietoa*, joka kuvautuu suoraan reaalimaailman käsitteiksi. Asiakastyyppi ja tilityyppi lisäävät *tyyppitietoutta* vastaaville perustieto-entiteeteille. Maa-entiteetti taas edustaa järjestelmässä olevaa *parametritietoutta*, jolla voidaan lisätä tarkennusta tässä tapauksessa esimerkiksi asiakas-entiteetille.

Asiakasluokka taas edustaa oman tyyppistä entiteettiä, sillä sen avulla voidaan *luokitella* muita, tässä tapauksessa asiakasentiteettejä. Samankaltainen luokittelu olisi relevanttia myös muille perustietoentiteeteille. Tilitapahtuma edustaa taas nimensä mukaisesti jotain järjestelmässä kuvattavaa *tapahtumatietoutta*. Tilitapahtumaentiteetille on oleellista se, että se on jossain *tilassa* ja tätä edustaa niin ikään oma entiteettinsä, Tilitapahtuman tila. Erityyppiset entiteetit on esitelty tarkemmin seuraavassa luvussa 7.3.

Esimerkkimallissa on lisäksi entiteettien välisiä suhteita, joissa on havaittavissa selkeitä ryhmiä. Esimerkiksi perustietoentiteettien välillä on *hierarkkista suhdetta* edustava liittyvä kokoelma-viite ja perustietoentiteettien ja tyyppientiteettien välillä taas on luonnollisesti *tyyppiä osoittava* suhde ja niin edelleen. EML-kielen erilaiset suhdetyypit on esitelty luvussa 7.5.

EML-kielen metamalli on esitetty kokonaisuudessaan liitteessä 1. EML-kielen tärkeimmät elementit on esitelty liitteessä 2.

7.3 Entiteetit

EML:n entiteettejä on seitsemää eri lajia. Lajittelu perustuu entiteettien toiminnalliseen rooliin sovelluksessa. Entiteettien eri lajit ovat *perustieto-olio*, *tapahtumaolio*, *tilaolio, luokitteluolio*, *parametriolio*, *säännöstöolio* sekä *tyyppiä ilmaiseva olio*. Luokittelun on

tarkoitus olla niin kattava, että jokainen sovelluksessa esiintyvä entiteetti voidaan mahdollisimman yksikäsitteisesti sijoittaa johonkin ryhmään.

Liitteessä 3 on esimerkki EML-kielellä tehdystä metamallista. Mallissa on käytetty lähes kaikkia erityyppisiä entiteettejä sekä erityyppisiä suhteita (erityyppisistä suhteista on kerrottu luvussa 7.5).

Jokaisella mallinnettavalla entiteetillä on seuraavat, metamallissa entiteettien kantaluokalle *Entity* määritellyt ominaisuudet:

- Entiteetin nimi
- Tietokantataulun nimi, johon entiteetin ilmentymät tallennetaan tietokannassa
- Lyhyt kuvaus entiteetistä.
- Entiteetin selkokielinen nimi englanniksi, suomeksi ja ruotsiksi
- Kyseisen tyyppisten entiteettien selkokielinen kokoelman nimi englanniksi, suomeksi ja ruotsiksi
- Luontiaika
- Attribuuttikokoelma (ks. luku 7.4)
- Avainta edustavat attribuutit
- Käyttäytyminen poistettaessa (voidaanko poistaa deletoimalla tietokannasta vai merkitäänkö vain poistetuksi)

Seuraavassa esitellään eri entiteettityyppien ominaisuuksia.

7.3.1 Perustieto-olio

Perustieto-olio (*Basic Entity*) edustaa jotain sovellusalueeseen kuuluvaa reaalimaailman oliota. Esimerkkejä perustieto-olioista on *Asiakas*, *Arvopaperi* ja *Tili*. Perustieto-olioille on ominaista, että se sisältää usein paljon attribuutteja. Perustieto-oliot ovat myös erittäin usein mukana järjestelmässä tapahtuvissa aktiviteeteissa, ts. esimerkiksi jokainen tapahtumaolio (ks. luku 7.3.2) liittyy lähes poikkeuksetta yhteen tai useampaan perustieto-olioon. Perustieto-olioille tehdään kaikkia pysyvyyteen liittyviä toimenpiteitä: lisäyksiä, päivityk-

siä ja poistoja. Näiden lisäksi perustieto-olioita tulee voida hakea monipuolisin hakuehdoin järjestelmästä.

7.3.2 Tapahtumaolio

Tapahtumaolio (*Transaction Entity*) edustaa järjestelmässä tapahtuvaa tapahtumaa. Esimerkki tapahtumaoliosta on *Tilitapahtuma*. Tapahtumaoliot sisältävät usein oleellista tietoa järjestelmän liiketoimintalogiikasta. Tapahtumaoliolle on ominaista, että sen luomisoperaatio suhteellisen monimutkainen ja että siihen liittyy lähes aina luomisen vastakohtana peruminen. Tapahtumaolio sisältää myös jonkinlaisen tiedon olion tilasta, joten metamalliin on lisätty mahdollisuus listata tapahtumaolion mahdolliset tilat jo mallinnusaikana (ks. liite 1, EML-kielen metamalli).

7.3.3 Tilaolio

Tilaolio (*Status Entity*) lisää tapahtumaoliolle tilatietoutta. Esimerkki tilaoliosta on *Tilitapahtuman tila*. Tilaentiteetille on oleellista, että sen instanssit ovat tiedossa jo mallinnusaikana.

7.3.4 Luokitteluolio

Luokitteluolion (*Classification Entity*) avulla järjestelmän perustieto-olioita voidaan luokitella. Esimerkkinä luokitteluista olkoon *Asiakasluokka* ja *Arvopaperiluokka*. Luokitteluolion ja perustieto-olion välinen suhde on aina luonteeltaan monen-suhde-moneen (ks. luku 7.5.3).

7.3.5 Parametriolio

Parametriolio (*Parameter Entity*) kapseloi parametrinomaista tietoa järjestelmän muista entiteeteistä sekä sovellusalueesta. Esimerkkinä parametriolioista ovat *Maa* ja *Markkinapaikka*. Parametrioliot ovat suhteellisen staattisia, niihin kohdistuu harvoin lisäyksiä, poistoja ja päivityksiä. Parametrioliot alustetaan usein järjestelmän asennusvaiheessa. Niillä on usein varsin vähän, lähinnä selitetyyppisiä attribuutteja. Parametriolioiden kaksi yleisintä käyttötarkoitusta on

1. Lisätä selkokielisiä selitteitä järjestelmän muille entiteeteille, erityisesti tapahtumaolioille
2. Ohjata muiden entiteettien toimintaa

7.3.6 Säännöstöolio

Säännöstöoliot (*Definition Entity*) toimivat järjestelmän muita toimintoja ohjaavina säännöstöinä. Säännöstöjen avulla on mahdollista luoda eritasoisia, dynaamisia sääntöjä, jotka otetaan käyttöön tietyn toimenpiteen suorituksen aikana. Esimerkkejä säännöstöolioista on *Tiliöintisäännöt* ja *Palkkiosäännöt*. Säännöstöolioille tehdään sovelluksen käytönaikana kaikkia eri pysyvyyteen liittyviä toimenpiteitä: lisäyksiä, päivityksiä, poistoja sekä hakuja eri hakukriteerein.

7.3.7 Tyyppiä ilmaiseva olio

Tyyppiä ilmaiseva olio (*Type Definition Entity*) lisää toiselle entiteetille tyyppitietouden. Esimerkki tyyppiä ilmaisevasta oliosta on *Tili*-entiteettiin liittyvä *Tilityyppi*. Tyyppiä ilmaiseva olio erottuu parametrioliosta siinä, että sen instanssit ovat tiedossa jo mallinnusaikana ja niihin ei kohdistu pysyvyyteen liittyviä muutoksia juuri ollenkaan sovelluksen käytön aikana. Metamalliin on lisätty mahdollisuus listata tyyppiä ilmaisevan olion kaikki mahdolliset instanssit jo mallinnusaikana (ks. liite 1, EML-kielen metamalli).

Koska tyyppiä ilmaisevan olion avulla voidaan ikään kuin lisätä periytymistietoutta muihin entiteetteihin, on sen vaikutus ohjelmakoodiin merkittävä. Tavallisesti sen perusteella generoidaankin erityisiä luontitehtaita, jotta ohjelmakoodissa voidaan luoda oikean tyyppisiä liiketoimintaolioita ajonaikana.

7.4 Attribuutit

Kullakin entiteetillä on joukko attribuutteja (*Attribute*). EML-kielen attribuutti on samassa roolissa kuin attribuutti ER-mallinnuksessa: se lisää entiteetille tietoa. Attribuutti voi olla tyypiltään jokin seuraavista: merkkijono, kokonaisluku, liukuluku, päivämäärä, totuusarvo tai aikaleima.

Entiteeteillä olevista attribuuteista kerätään joukko attribuutteja edustamaan entiteetin *avainta* (*Primary Key*) samaan tapaan kuin ER-mallinnuksessa vahvoilla entiteeteillä. EML-kielellä ei voi mallintaa heikkoja entiteettejä, eli entiteettejä joilla ei olisi avainta.

Jokaiselle mallinnettavalle attribuutille voidaan asettaa seuraavat ominaisuudet:

- Attribuutin nimi
- Attribuutin tyyppi
- Tietokantasarakkeen nimi
- Tietokantasarakkeen pituus (tämä ei ole relevantti kaiken tyyppisille attribuuteille)
- Sallitaanko tyhjä-arvot
- Onko attribuutti identity-tyyppinen (tämä on relevantti erityisesti avainta edustaville attribuuteille)
- Selkokielinen nimi englanniksi, suomeksi ja ruotsiksi
- Halutaanko attribuutin perusteella voida hakea entiteettejä järjestelmästä
- Luontiaika

7.5 Suhteet

EML-kielessä on neljän tyyppisiä entiteettien välisiä suhteita. Eri suhdetyypit ovat *yksinkertainen viite*, *liittyvä kokoelma*, *luokittelu* ja *tyypinmäärittely*. Erityyppiset suhteet eroavat semantiikan lisäksi toiminnallisuudessa, eli käytännössä suhteen perusteella generoitavien tuotosten määrässä ja laadussa. Seuraavassa esitellään EML-kielen eri suhteet.

7.5.1 Yksinkertainen viite

Yksinkertainen viite (*Simple Reference*) on perusassosiaatio ja nimensä mukaisesti yksinkertainen viite kahden eri entiteetin välillä. Sen roolit ovat *viittaavat oliot* ja *viitatut oliot*.

Esimerkki yksinkertaisesta viitteestä EML-instanssissa on *Laskun (tapahtumaolio)* ja *Salkun (perustieto-olio)* välinen linkki. Metamallissa yksinkertainen viite on luonteeltaan monen-suhde-moneen-assosiaatio, toisin sanoen entiteettiin voi viitata useita entiteettejä yksinkertaisella viitteellä ja päinvastoin. Varsinaisessa EML-instanssissa, eli kehitteillä olevan sovelluksen tietomallissa yksinkertainen viite kuvaa yhden-suhde-moneen assosiaatiota (esimerkiksi yhteen *salkkuun* voi viitata monta *laskua*).

7.5.2 Liittyvä kokoelma

Liittyvä kokoelma (*Dependent collection*) mallintaa entiteettien perustieto-olioiden välistä hierarkkista suhdetta. Liitteessa 3 esitetyssä esimerkissä hierarkkisuutta kuvaa se, että *Asiakkaalla* voi olla monta *Salkkua*, joissa kussakin voi olla monta *Tiliä*. Liittyvän kokoelman roolit ovat nimeltään *liittyvät kokoelmat* ja *isäntäoliot*. Metamallissa liittyvä kokoelma on luonteeltaan monen-suhde-moneen-assosiaatio, toisin sanoen entiteetti voi olla usean entiteetin liittyvä kokoelma sekä toisaalta entiteetillä voi olla useita liittyviä kokoelmia. Lopullisessa sovelluksessa liittyvä kokoelma kuvautuu yhden-suhde-moneen-assosiaatioksi. Liittyvään kokoelmaan liittyy esimerkiksi sellaista toiminnallisuutta, että sovelluksen käyttöliittymällä tulee voida navigoida hierarkkista suhdetta edestakaisin.

7.5.3 Luokittelu

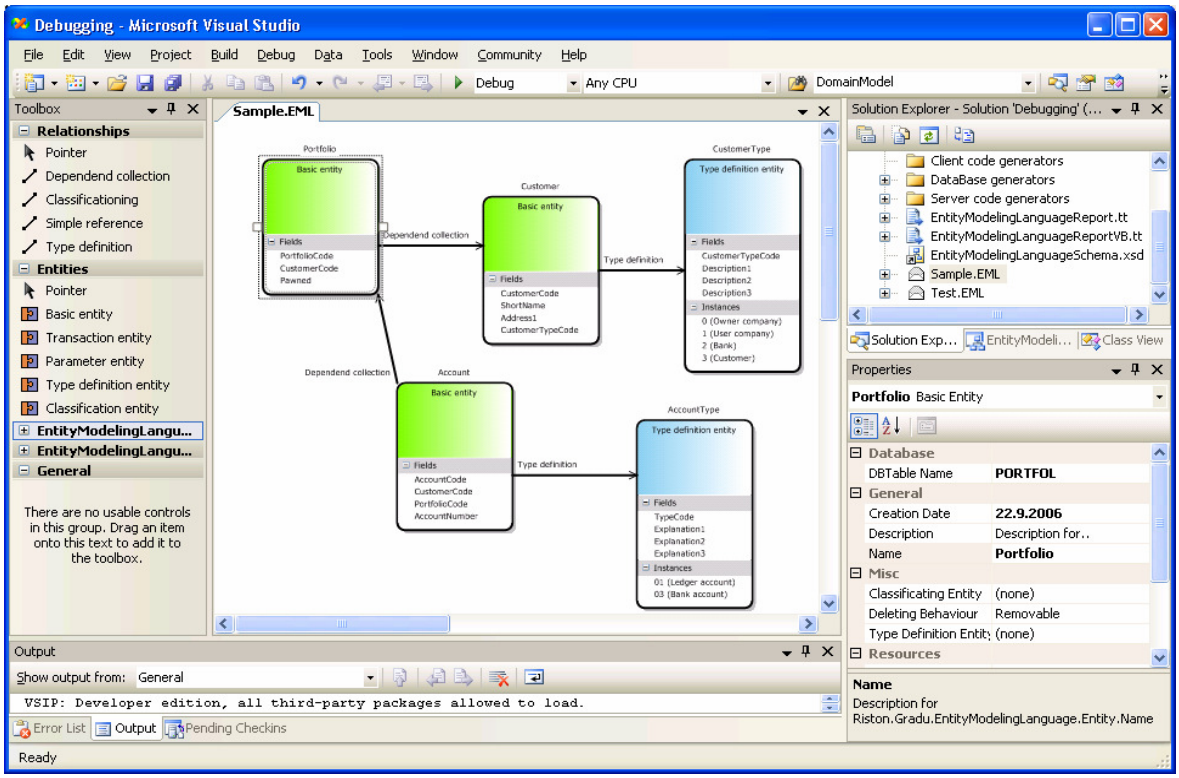
Luokittelu (*Classification*) merkitsee kehitettävässä sovelluksessa aitoa monen-suhde-moneen assosiaatiota, joskin tämä näkyy mallinnuskielessä implisiittisesti. Siihen voi sisältyä attribuutteja ER-mallinnuksen monen-suhde-moneen assosiaation tapaan. Tietokantatasolla monen-suhde-moneen-assosiaatio ilmenee aina omana välitaulunaan, joten assosiaatiolle lisätyt attribuutit kuvautuvat suoraan välitaulun sarakkeiksi suhteeseen osallistuvien entiteettien avainten lisäksi. Luokittelu-suhteen roolit ovat *luokitteleva entiteetti* ja *luokiteltu entiteetti*. Metamallissa luokittelu on yhden-suhde-yhteen-assosiaatio, nimittäin entiteettiin voi sisältyä korkeintaan yksi luokitteleva entiteetti ja luokitteleva entiteetti liittyy aina täsmälleen yhteen luokiteltuun entiteettiin. Esimerkki luokittelu-suhteesta on *Asiakas (perustieto-olio)*- ja *Asiakasluokka (luokitteluolio)*-entiteettien välinen suhde.

7.5.4 Tyypinmäärittely

Tyypin määrittely (*Type Definition*) liittyy läheisesti tyyppiä ilmaisevien olioiden mallintamiseen. Esimerkki Tyypinmäärittelystä on *Asiakkaan (perustieto-olio)* ja *Asiakastyypin (tyyppiä ilmaiseva olio)* välinen suhde. Suhteeseen liittyvät roolit ovat nimeltään *Tyypitetty olio* ja *isäntäolio*. Tyypinmäärittely on sekä metamallissa, että EML-mallin kuvaamassa tietomallissa yhden-suhde-yhteen-assosiaatio.

7.6 Mallinnustyökalu

EML-kieltä tukeva mallinnustyökalu on DSL Tools-työkalulla toteutettu Microsoft Visual Studio 2005 - kehitysympäristöön integroitu piirtotyökalu, josta löytyy Visual Studion käyttäjille tutut elementit *piirtopinta*, *työkalulaatikko* sekä *ominaisuudet*-ikkuna. Varsinainen mallinnus tapahtuu raahaamalla työkalupalkista mallinnuskielen elementtejä piirtopinnalle sekä sen jälkeen asettamalla elementtien ominaisuudet ominaisuudet-ikkunassa. Seuraavassa kuvassa 18 on kokonaisnäkymä mallinnustyökalusta käytössä.



Kuva 18, EML-mallinnustyökalu käytössä

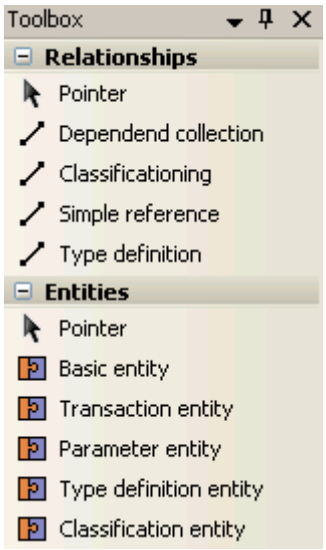
Yllä olevan kuvan keskellä näkyy työkalun piirtopinta, vasemmalla työkalulaatikko sekä oikealla alhaalla ominaisuudet-ikkuna.

7.6.1 Piirtopinta

Piirtopinta toimii EML-mallien graafisena suunnittelutyökaluna samaan tapaan kuin esimerkiksi Visual Studion käyttöliittymien suunnittelutyökalu. Piirtopinta tukee zoomausta ja mahdollistaa näin ollen isojenkin mallien kehittämisen. Mallinnustyökalu assosioi .EML-päätteiset tiedostot automaattisesti ja osaa avata ne piirtopinnalle (ks. kuva 18 ja sample.EML).

7.6.2 Työkalulaatikko

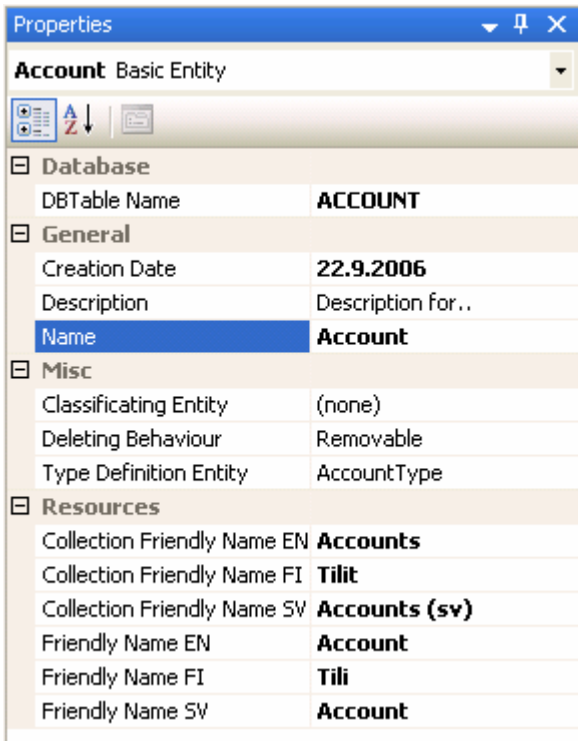
Mallinnustyökalun työkalupalkissa on EML-kielen elementit ryhmiteltynä suhteisiin ja entiteetteihin. Seuraavana on kuva työkalupalkista ja sen tarjoamista palveluista.



Kuva 19, EML-mallinnustyökalun työkalulaatikko

7.6.3 Ominaisuudet-ikkuna

Ominaisuudet-ikkunassa asetetaan mallinnuskielen elementtien ominaisuudet. Entiteeteille ja Attribuuteille asetettavat ominaisuudet esiteltiin luvuissa 7.3 ja 7.4. Seuraavassa kuvassa esitetään piirtopinnalla aktiivisena olevan Account-perustieto-olion ominaisuudet-ikkuna.



Kuva 20, Account-perustieto-olion ominaisuudet-ikkuna

7.7 EML-malleista generoitavat artefaktit

EML-kielellä esitetyt tietomallit sisältävät metadataa ohjelmiston kohdealueesta ja mallien perusteella voidaan generoida valitun kolmikerrosarkkitehtuurin mukaista ohjelmakoodia sekä muita artefakteja. EML-kielen elementeille annettu metadata konkretisoituu generaattorien toimesta hyödylliseksi informaatioksi, joka tukee järjestelmän toimintaa.

Seuraavassa on lueteltu artefakteja, joita EML-tietomallien perusteella on mahdollista generoida:

- Sovelluspalvelimen sekä asiakassovelluksen liiketoimintaluokat (esimerkiksi Account). Luokkiin toteutetaan pysyvyystoiminnallisuudet (Load, Save ja Delete). Sovelluspalvelimen liiketoimintaluokat keskustelevat suoraan tietokannan kanssa ja asiakassovelluksen luokat taas toimivat edustana ja keskustelevat sovelluspalvelimen kanssa Web Services -rajapinnan kautta.

- Sovelluspalvelimen sekä asiakassovelluksen liiketoimintaluokkia vastaavat kokoelmat (esimerkiksi Accounts). Kokoelmiin toteutetaan hakufunktiot haluttujen kenttien perusteella sekä pääavainten perusteella (LoadByFields ja LoadByPrimaryKeys).

- SOAP-yhteensopivat serialisoituvat liiketoimintaluokat (esimerkiksi AccountData). Lisäksi sovelluspalvelimelle toteutetaan toiminnot, joilla varsinaisen liiketoimintaolion perusteella luodaan Web Services –rajapinnan kautta serialisoituva olio.

- Sovelluspalvelimen Web Services -metodien toteutukset edellä mainituille toiminnoille.

- Asiakassovellukset haku- ja muokkausnäyttöjen rungot liiketoimintaolioille (AccountSearchControl ja AccountEditControl).

- Liiketoimintaluokkia vastaavien tietokantataulujen luontilauseet SQL-kielellä. Tämä sisältää muun muassa Oracle-tietokannan vaatimat sekvenssien luontilauseet. Lisäksi kohdekehyksen vaatimat sisältölauseet mm. roolien hallintaa varten voidaan generoida tietomallin perusteella. Tietokannan suorituskykyyn vaikuttavia tekijöitä, kuten esimerkiksi indeksejä viiteavaimien perusteella on mahdollista generoida mallissa olevan metadatan avulla.

- Resurssi-dll:t, joita voidaan käyttää järjestelmän monikielisyyden tukemiseksi.

- Enumeraatiot, joilla edellä mainittuihin resurssitietoihin päästään käsiksi.

- Yksikkötestejä muun muassa olioiden tietokantatoimintojen testaamiseen. Liitteessä 4. on esitelty pysyvyystoimintojen yksikkötestit tuottava generaattori ja liitteessä 5. taas osa generoidusta yksikkötestikoodista.

7.8 EML-työkalun käyttö sovelluskehityksessä

EML-työkaluun kehitetyt generaattorit eivät generoi kaikkea sovellukseen kuuluvaa lähdekoodia. EML-kielellä mallinnettavat asiat ovat luonteeltaan staattisia ja näin ollen työkalun

avulla kehitettävän sovelluksen varsinainen dynaaminen osuus, ns. *liiketoimintalogiikka* tulee ohjelmoida perinteisellä tavalla. Lisäksi sovelluksen näyttöjen ulkoasu tulee toteuttaa perinteisellä tavalla Visual Studion käyttöliittymädesigneriä käyttäen. Generaattorien kehityksessä on pyritty huomioimaan manuaalisesti tehtävät laajennukset, muun muassa käyttämällä .NET Frameworkin 2.0:n tarjoamaa *Partial*-avainsanaa luokkien määrityksissä. Näin voidaan täyttää vaatimus siitä, ettei generoituun koodiin tarvitse koskea käsin. Muita hyväksihavaittuja keinoja käsinkirjoitetun ja generoiden ohjelmakoodin erottamiseksi esiteltiin luvussa 5.3.

EML-työkalun mahdollinen käyttöönotto sovelluskehityksen tueksi ei tule muuttamaan oleellisesti käyttäjäorganisaation sovelluskehityksen rooleja. Ainoastaan EML-metamallin sekä kehitettyjen generaattorien ylläpidosta tulee uusi tehtävä siihen nimety(i)lle henkilö(i)lle. Itse EML-työkalun käyttäminen on mahdollista kaikille siihen koulutetuille sovelluskehittäjille.

8 Yhteenveto

Sovellusaluemallinnuksen käyttöönotolla voidaan saavuttaa merkittäviä parannuksia ohjelmistokehityksen tuottavuudessa ja laadussa. Sovellusaluemallinnuksen käyttöönottaminen ohjelmistokehityksen tueksi vaatii kuitenkin sen, että ohjelmistokehitysorganisaatiolla on ennestään riittävä kohdealueen tuntemus, olemassa olevia toteutuksia sekä kohdekehys, jonka avulla kohdealueelle voidaan kehittää ohjelmistoja tehokkaasti ja laadukkaasti.

Sovellusaluekielet ovat tietyn organisaation ja tietyn sovellusalueen tarpeisiin kehitettyjä, tehokkaita ja sovellusalueellaan ilmaisuvoimaisia kieliä toisin kuin esimerkiksi UML. Sovellusaluekielten kehittämisellä pyritään mahdollistamaan valitun sovellusalueen käsitteiden mallintaminen formaalisti. Koska sovellusaluekielet ovat formaaleja kieliä, on niitä mahdollista käsitellä koneellisesti esimerkiksi generoitaessa ohjelmakoodia sovellusaluekielellä esitetyn mallin perusteella.

Software Factories on Microsoftin kehittämä konsepti, joka perustuu vahvasti sovellusaluemallinnuksen käyttöönottoon ohjelmistokehityksen tueksi. Sen eräänä ylevänä tavoitteena on tehdä sama ohjelmistoteollisuudelle, kuin mitä tapahtui perinteiselle teollisuudelle teollisen vallankumouksen aikana. Ylevästä tavoitteesta huolimatta konsepti ei ole kaikenkattava hopealuoti ohjelmistoteollisuuden ongelmiin, mutta voi toki parhaimmillaan tarjota yksittäiselle organisaatiolle merkittäviä parannuksia tuottavuudessa kunhan vain edellä mainitut vaatimukset sovellusaluemallinnuksen menestyksekkäästä käyttöönotosta täyttyvät.

Tutkielman käytännön osuudessa kehitetty EML-sovellusaluekieli mahdollistaa sovelluskehityksen rutiininomaisimpien vaiheiden automatisoinnin. EML-sovellusaluekielen avulla voidaan mallintaa rahoitusalalle toteutettavien Enterprise-tason sovellusten tietomallia kohdealueen erityispiirteet huomioonottaen. EML-kieli on puhdasoppinen sovellusaluekieli siinä mielessä, että se on kehitetty juuri tietylle sovellusalueelle sekä tietyn kehittäjäorganisaation tarpeisiin olemassa olevien sovellusten pohjalta.

EML-kielellä esitetyn tietomallin perusteella on mahdollista generoida ohjelmakoodia ja muita artefakteja automaattisesti. EML-työkalu käyttöönottaminen lisää näin ollen ohjel-

mistonkehityksen tuottavuutta sekä laatua, sillä ilman EML-työkalun mahdollistamaa ohjelmiston osien automaattista generointia esimerkiksi attribuutin lisääminen liiketoimintaluokalle vaatii muutoksen toteuttamisen käsin moneen paikkaan ollen samalla sekä hidasta, että virhealtista.

8.1 Jatkokehitys

Tämän tutkielman käytännön osuudessa kehitetty EML-työkalu on prototyyppi, eikä sitä ole vielä integroitu varsinaiseen sovelluskehitykseen. Eräs mielenkiintoinen tutkimusaihe olisikin se, miten kehitetty työkalu voitaisiin liittää olemassa olevaan Visual Studion solution/projektirakenteeseen. Kun mallinnuskielellä lähdettäisiin mallintamaan todellista järjestelmää, saattaisi herätä myös itse mallinnuskieltä koskevia jatkokehitysideoita.

Työkalua kehitettäessä on herännyt myös joitakin kehitysideoita luvussa 6 esiteltyä kohdekehystä kohtaan. Eräänä tällaisena on järjestelmän parempi monikielisyyden tuki. Tällä hetkellähän on jo mallinnusaikana tiedettävä käytettävät kielet sekä termien käännökset. Monikielisyyden tukemista olisi syytä tutkia tavoitteena se, että tuettuja kieliä voitaisiin lisätä jo käännettyyn ja asennettuun ohjelmistoon ilman tarvetta ohjelmiston uudelleenkääntämiselle tai järjestelmän tietokannan rakenteen muuttamiselle.

Lähteet

[Boo04] Booch Grady, Brown Alan, Lyengar Sridhar, Rumbaugh James ja Selic Bran, *An MDA Manifesto***,** ”MDA Journal“, Toukokuu 2004, s. 2-9

[Bro95] Brooks Frederick P., *The Mythical Man-Month*: Essays on Software Engineering, Addison-Wesley Professional, Boston, 1995

[Cop00] Coplien James O., *Multi-paradigm design*, kirjassa ”Proceedings of the GCSE '99, First International Symposium on Generative and Component-Based Software Engineering”, Springer, 1999

[Due00] van Duersen Arie, Klint Paul ja Visser Joost, ”Domain-Specific Languages:An Annotated Bibliography”, saatavilla www-muodossa <URL: http://homepages.cwi.nl/~arie/papers/dslbib>, 9.2.2000

[Fay97] Fayad Mohamed ja Schmidt Douglas C., *Object-oriented application frameworks*, kirjassa ”Communications of the ACM volume 40”, ACM Press, s. 32-38, 1997

[Foo88] Foote Brian ja Johnson Ralph E., *Desinging reusable classes*, ”Journal of Object-Oriented Programming”, 2/1988, s. 22-35

[Foo96] Foote Brian ja Yoder Joseph, *Evolution, Architecture and Metamorphosis*, kirjassa ”Pattern Languages of Program Design 2”, Addison Wesley, Luku 13, 1996

[Fow05a] Fowlen Martin, ”Language Workbenches: The Killer-App for Domain Specific Languages? ”, saatavilla www-muodossa <URL: http://martinfowler.com/articles/languageWorkbench.html>, 12.6.2005

[Fow05b] Fowler Martin, ”Generating code for DSLs”, saatavilla www-muodossa <URL: http://martinfowler.com/articles/codeGenDsl.html> , 12.6.2005

[Gam95] Gamma Erich, Helm Richard, Vlissides John, ”Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional, Boston, 1995

[Gar94] Garlan David ja Shaw Mary, *An Introduction to Software Architecture*, kirjassa ”Advances in Software Engineering and Knowledge Engineering”, World Scientific Publishing Company, s. 1-39, 1993

[Gar95b] Garlan David, Allen Robert ja Ockerbloom John, *Architectural mismatch, why reuse is so hard*, IEEE Software, 6/1995, s.17-26

[Gre04a] Greenfield Jack ja Short Keith, “Software Factories, Assembling applications with Patterns, Frameworks and Tools”, Wiley, Indiapolis USA, 2004

[Gre04b] Greenfield Jack ja Short Keith, ”Moving to Software Factories”, saatavilla www-muodossa <URL: http://blogs.msdn.com/askburton/articles/232021.aspx>, 20.9.2004

[Gre04c] Greenfield Jack, ”The Case for software factories”, saatavilla www-muodossa <URL: http://www.itarchitect.co.uk/articles/display.asp?id=96>, 24.8.2004

[Her03] Herrington Jack, ”Code Generation in Action”, Manning Publications, New York USA, 2003

[Ise05] Iseger Martijn, ”Domain-specific modeling for generative software development”, saatavilla www-muodossa <URL: http://www.itarchitect.co.uk/articles/display.asp?id=161>, 17.4.2005

[Joh06] Johnson Stephen C., ”Yacc: Yet Another Compiler-Compiler”, saatavilla www-muodossa <URL: http://dinosaur.compilertools.net/yacc/index.html>, viitattu 15.11.2006

[Kel05] Kelly Steven, ”Improving Developer Productivity With Domain-Specific Modeling Languages”, saatavilla www-muodossa <URL: http://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Kelly_DomainModeling.pdf>, 3.7.2005

[Kel06] Kelly Steve, ”Code Generation”, saatavilla www-muodossa <URL: http://www.codegeneration.net/tiki-read_article.php?articleId=81>, 10.6.2006

[Ket02] Kettunen Eero, ER-mallinnuksen perusteet, saatavilla www-muodossa <URL: http://www.lpt.fi/it/opetus/tietokantasuunnittelu/erperusteet.pdf>, 2002

[Kos04] Koskinen Jussi, ”Ohjelmistojen ylläpidon seminaari – 2003”, Jyväskylän Yliopistopaino, Jyväskylä, 2003

[May98] May Lorin J., *Major Causes of Software Project Failures*, *CROSSTALK* The Journal of Defence Software Engineering, 7/1998, s. 9-12

[Med97] Medvidovic Nenad ja Rosenblum David S., *Domains of Concern in Software Architectures and Architecture Description Languages*, Kirjassa ”Proceedings of the USENIX Conference on Domain-Specific Languages”, Santa Barbara, CA, 1997

[Met05a] MetaCase, ”Domain-Specific Modeling with Metaedit+: 10 times faster than UML”, saatavilla www-muodossa <URL: http://www.metacase.com/papers/Domain-specific_modeling_10X_faster_than_UML.pdf>, 2005

[Ms05] Microsoft, ”Visual Studio 2005 Team System Modeling Strategy and FAQ”, saatavilla www-muodossa <URL: http://msdn.microsoft.com/vstudio/dsltools/default.aspx?pull=/library/en-us/dnvs05/html/vstsmodel.asp>, toukokuu 2005

[Muk03] Mukerji Jishnu ja Miller Joaquin, ”Model Driven Architecture”, saatavilla www-muodossa <URL: http://www.omg.org/docs/omg/03-06-01.pdf>, 12.6.2003

[Par94] Parnas David Lorge, *Software Aging*, kirjassa ”Proceedings of the 16th international conference on Software engineering”, IEEE Computer Society Press, s. 279-287, 1994

[Poh02] Pohjonen Risto ja Kelly Steven, “Domain-Specific Modeling; Improving productivity and time to market”, saatavilla WWW-muodossa <URL: http://www.metacase.com/papers/DrDobbs_Domain-Specific_Modeling.html> elokuu 2002

[Sei06] Carnegie Mellon University: Software Engineering Institute, *Software Product Lines*, saatavilla www-muodossa <URL: http://www.sei.cmu.edu/productlines/>, viitattu 15.11.2006

[Tha00] Thalheim Bernard, ”Entity-Relationship Modeling”, Springer, New York, 2000

[Tol00a] Tolvanen Juha-Pekka ja Kelly Steven, *Visual domain-specific modelling: Benefits and experiences of using metaCASE tools*, kirjassa ”International workshop on Model Engineering”, ECOOP, 2000

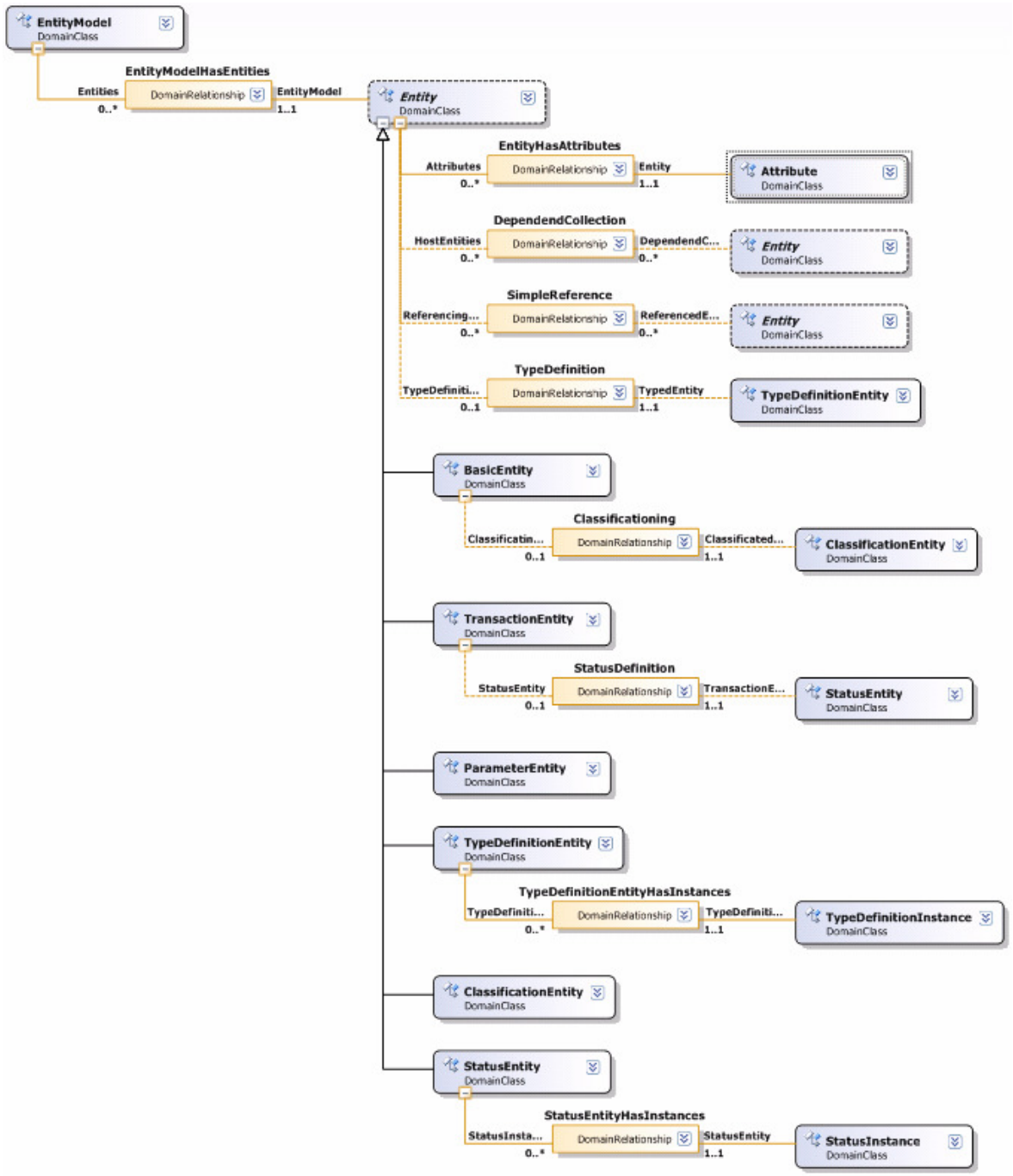
[Tol06a] Tolvanen Juha-Pekka, ”Language creation difficult?”, saatavilla www-muodossa <URL: http://www.metacase.com/blogs/jpt/blogView?showComments=true&entry=3321189439> , 30.3.2006

[Tol06b] Tolvanen Juha-Pekka, ”Domain-specific Modeling: Making Code Generation Complete”, saatavilla www-muodossa <URL: http://www.devx.com/architect/Article/31351>, 27.4.2006

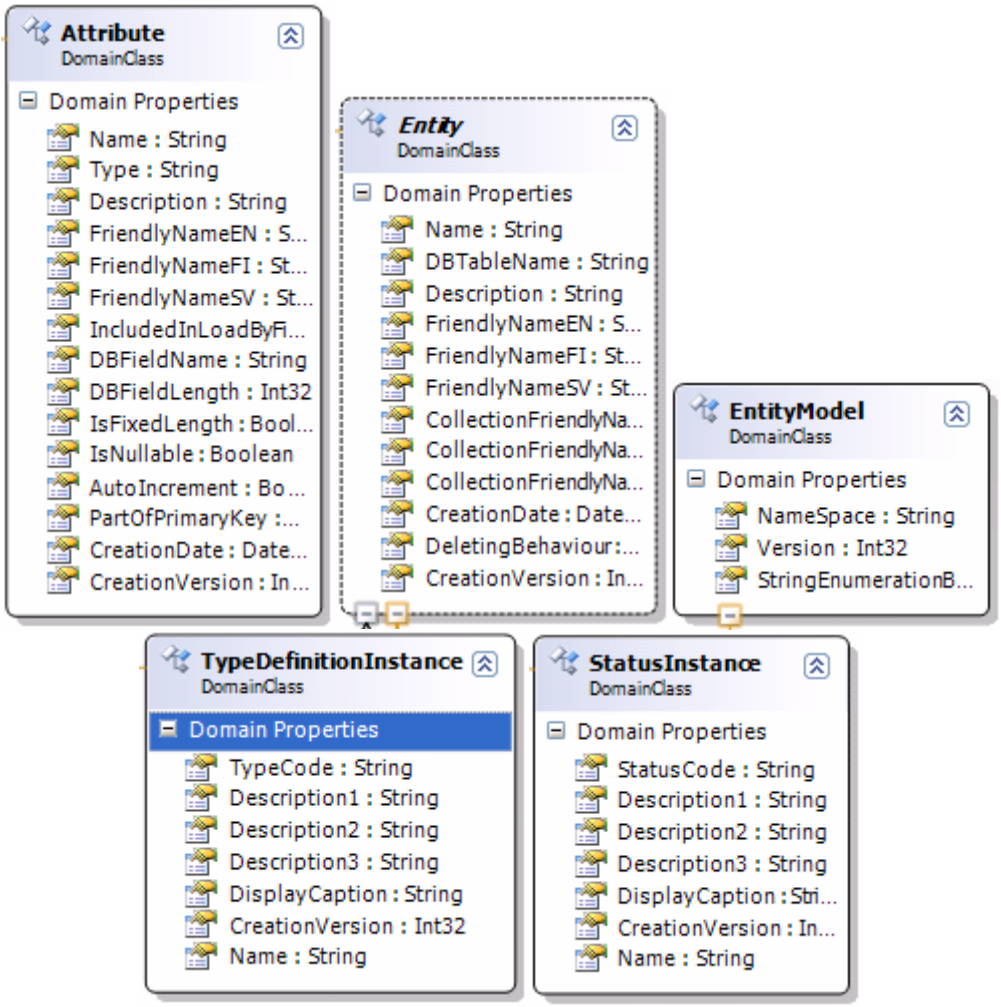
[Völ04] Völter Markus ja Bettin Jorn, ”Patterns for Model-Driven Software Development”, saatavilla www-muodossa <URL: http://www.voelter.de/data/pub/MDDPatterns.pdf>, 10.5.2004

Liitteet

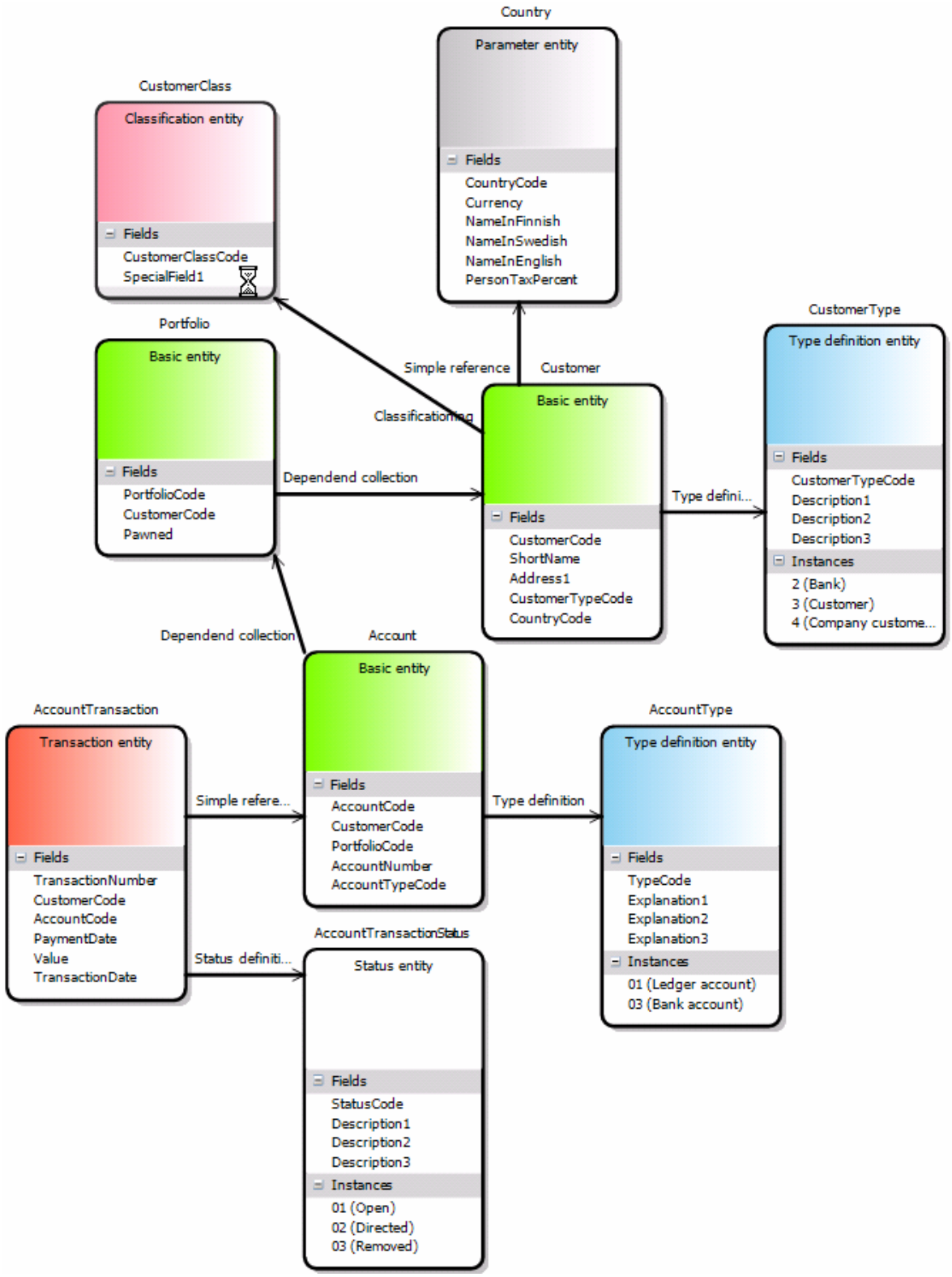
Liite 1. EML-metamalli



Liite 2. EML-elementit, joilla ominaisuuksia



Liite 3. Esimerkki EML-kielen käytöstä



Liite 4. Pysyvyystoimintojen yksikkötestien generaattori

```
<#@ template inhe-
rits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" langu-
age="VB" debug="true"#>
<#@ output extension=".vb" #>
<#@ EntityModelingLanguage processor="EntityModelingLanguageDirectiveProcessor" re-
quires="fileName='..\Sample.EML'" #>
<#@ import NameSpace="Riston.Gradu.EntityModelingLanguage"#>
<#@ import NameSpace="Riston.Gradu"#>
<#@ import NameSpace="System.Collections"#>
Imports Samstock.Server.Common
Imports Samstock.Server.<#=Me.EntityModel.NameSpace#>
Imports nUnit.Framework

<TestFixture()> _
Public Class <#=Me.EntityModel.NameSpace#>UnitTests

    <SetUp()> _
    Public Sub SetUp
    End Sub

    <TearDown()> _
    Public Sub TearDown
    End Sub

<#
        Dim entity as Entity
        For Each entity in Me.EntityModel.Entities
            Dim ClassName As String = entity.Name
            Dim DataClassName As String = entity.Name + "Data"
            Dim primaryKeys As ArrayList = GetPrimaryKeys(entity)

#>
    <Test()> _
    Public Sub <#=ClassName#>PersistenceTest
        Dim obj As New <#=ClassName#>

<#      For Each attrib As EntityModelingLanguage.Attribute In entity.Attributes #>
        obj.<#=attrib.Name#> = <#=GetRandomValue(attrib)#>
<#      Next#>

        obj.Save()

        Dim obj2 As New <#=ClassName#>
        obj2.SetStructure(obj.GetStructure)

<#      For Each attrib As EntityModelingLanguage.Attribute In entity.Attributes #>
        obj.<#=attrib.Name#> = Nothing
<#      Next#>

        obj.Load(<#WriteParameterList("obj2",primaryKeys)#>)

        'This test tells if there is something gone wrong with Save/Load
        Assert.Equals(obj1, obj2)

        Try
            obj.Delete()
            obj.Load(<#WriteParameterList("obj",primaryKeys)#>)
            Throw New Exception("Object should not be loadable after deletion!")
        Catch onf As ObjectNotFoundException
            'This is normal way
        End Try

        Try
            obj2.Delete()
            obj2.Load(<#WriteParameterList("obj2",primaryKeys)#>)
            Throw New Exception("Object should not be loadable after deletion!")
        Catch onf As ObjectNotFoundException
```

```
            'This is normal way
        End Try

    End Sub

<#
        Next
#>
End Class

<#
 '********************************** Helping functions
*****************************************
#>

<#+

Private Function GetPrimaryKeys(entity As EntityModelingLanguage.Entity) As ArrayList
    Dim arr As new ArrayList
    For Each attrib As EntityModelingLanguage.Attribute in entity.Attributes
        if attrib.PartOfPrimaryKey Then
            arr.Add(attrib)
        End If
    Next
    Return arr
End Function


Private Function IsLast(ByVal o As Object, ByVal i As ArrayList) As Boolean
     Return i.IndexOf(o) = i.Count - 1
End Function

Private Sub WriteParameterList(objName As String, attributes As ArrayList)
    For Each attrib As EntityModelingLanguage.Attribute in attributes
        Write(objName + "." + attrib.Name)
        If Not IsLast(attrib, attributes) Then
            Write(", ")
        End If
    Next
End Sub

Private Function toPrivateFieldName(s As String) As String
    If s.Length = 0 Then Return ""
    If s.Length = 1 Then Return s.toLower
    Return "_" + s(0).ToString.ToLower + s.SubString(1)
End Function

Private Function GetRandomValue(attrib As EntityModelingLanguage.Attribute) As String
    Static val As Integer
    val += 1
    Select Case attrib.Type
        Case "String"
            Return """Value" + val.toString() + """"
        Case "Integer"
            Return val.ToString
        Case "Boolean"
            Return True.ToString()
        Case "Date"
            Return "Date.Now"
        Case Else
            Return """Default"+ """"
    End Select
End Function
#>
```

Liite 5. Customer-olion pysyvyystesti

```
Imports Samstock.Server.Common
Imports Samstock.Server.BaseData
Imports nUnit.Framework

<TestFixture()> _
Public Class BaseDataUnitTests

    <SetUp()> _
    Public Sub SetUp
    'TODO:Initialization
    End Sub

    <TearDown()> _
    Public Sub TearDown
    'TODO:Finalization
    End Sub


    <Test()> _
    Public Sub CustomerPersistenceTest
        Dim obj As New Customer

        obj.CustomerCode = "Value1"
        obj.ShortName = "Value2"
        obj.Address1 = "Value3"
        obj.CustomerTypeCode = "Default"
        obj.CountryCode = "Value5"

        obj.Save()

        Dim obj2 As New Customer
        obj2.SetStructure(obj.GetStructure)

        obj.CustomerCode = Nothing
        obj.ShortName = Nothing
        obj.Address1 = Nothing
        obj.CustomerTypeCode = Nothing
        obj.CountryCode = Nothing

        obj.Load(obj2.CustomerCode)

        'This test tells if there is something gone wrong with Save/Load
        Assert.Equals(obj1, obj2)

        Try
            obj.Delete()
            obj.Load(obj.CustomerCode)
             Throw New Exception("Cannot load object after deletion!")
        Catch onf As ObjectNotFoundException
            'This is normal way
        End Try

        Try
            obj2.Delete()
            obj2.Load(obj2.CustomerCode)
             Throw New Exception("Cannot load object after deletion!")
         Catch onf As ObjectNotFoundException
             'This is normal way
        End Try

    End Sub
...
```