MDA – malliperustainen arkkitehtuuri

Hannu Kavakka

8.12.2004

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

// Tiivistelmä

Ohjelmistojen laadukkaaseen kehittämiseen tähdätään erilaisten ohjelmistotuotantoprosessien avulla. Perinteiset ohjelmistotuotantoprosessit sisältävät kuitenkin muutamia ongelmia, joita on pyritty ratkaisemaan useiden eri menetelmien avulla. Yksi tällainen menetelmä on malliperustainen arkkitehtuuri, MDA, joka käyttää ohjelmistokehityksessä apuna sovellusalustariippumattomia UML-malleja sekä koodingenerointia. Tässä tutkielmassa selvitetään malliperustaisen arkkitehtuurin peruskäsitteet ja kerrotaan kuinka MDA:n avulla voidaan ratkaista perinteisen ohjelmistotuotannon ongelmia. Lopuksi havainnollistetaan malliperustaisen arkkitehtuurin käyttöä pienimuotoisen esimerkin avulla erityisesti notkean MDA:n näkökulmasta.

ACM-luokat (ACM Computing Classification System, 1998 version): D.2.2, D.2.11, D.2.13

Avainsanat: MDA, malliperustainen arkkitehtuuri, koodingenerointi, UML

Sisältö

1 Johdanto **1**

1.1 Tutkielman rakenne . . . 5

2 MDA-kehys **6**

2.1 Mallit . . . 6

2.1.1 Liiketoiminta- ja ohjelmistomalli . . . 7

2.1.2 PIM ja PSM . . . 9

2.2 Transformaatio . . . 11

2.3 Standardit . . . 13

2.3.1 Meta Object Facility . . . 15

2.3.2 Unified Modeling Language . . . 17

2.3.3 Common Warehouse Metamodel . . . 22

2.3.4 XML Metadata Interchange . . . 24

3 Ohjelmistotuotannon kehitysprosessit **28**

3.1 Perinteiset kehitysprosessit ja niiden ongelmat . . . 28

3.2 Malliperustainen kehitysprosessi . . . 35

3.2.1 Notkea MDA . . . 39

4 Koodingenerointi **42**

5 MDA käytännössä **45**

5.1 Apuvälineet . . . 45

5.1.1 Enterprise Architect . . . 45

5.1.2 iQgen . . . 46

5.1.3 JBuilderX . . . 49

5.2 Esimerkkisovellus . . . 50

6 Yhteenveto **54**

Viitteet **56**

Liite 1: Esimerkin luokkakaavio **61**

Liite 2: Näyte XMI-tiedostosta **62**

Liite 3: Kaavaimet **65**

Liite 4: Näyte generoiduista SQL-lauseista **75**

Liite 5: Näyte lähdekoodista **76**

1 Johdanto

Ennen *malliperustaisen arkkitehtuurin* (Model Driven Architecture, MDA) tärkeimpien periaatteiden käsittelemistä, on syytä perehtyä hieman ohjelmointikielten kehitykseen ja siten muun muassa *abstraktiotason nostamiseen*. Kleppe & al. (2003) kuvaavat ohjelmointikielten kehityksen historian taulukon 1 mukaisina jaksoina, joista käy ilmi jokaiselle jaksolle tyypillinen ohjelmointisuuntaus. 1950–luvun alussa oli tyypillistä pelkän konekielen, ja hieman myöhemmin symbolisten konekielten käyttö. Frankel (2003) kutsuu tätä vuoteen 1965 asti kestävää aikakautta *konekeskeiseksi*, koska ohjelmoijat kirjoittivat ohjelmakoodinsa suoraan, tai ainakin melkein suoraan, konekielelle. Tämä mahdollisti muun muassa ohjelmien muistin- ja prosessorinkäytön korkean optimoinnin, mutta itse ohjelmoijien työskentely oli vielä erittäin tehotonta verrattuna nykyhetkeen. Symboliset konekielet auttoivat tehottomuuteen mahdollistamalla esimerkiksi näytölle tulostamiseksi vaadittujen käskyjen tiivistämisen monien rivien sijasta yhteen käskyyn.

Taulukko 1: Ohjelmoinnin historia (Kleppe & al., 2003).

Vuosi	*Ohjelmointikieli*
1950	Pelkkä konekieli.
1950–1965	Symboliset konekielet.
1965–1985	Proseduraaliset ohjelmointikielet.
1985–nykyhetki	Olio-ohjelmointikielet.
Nykyhetki–...	Mitä seuraavaksi?

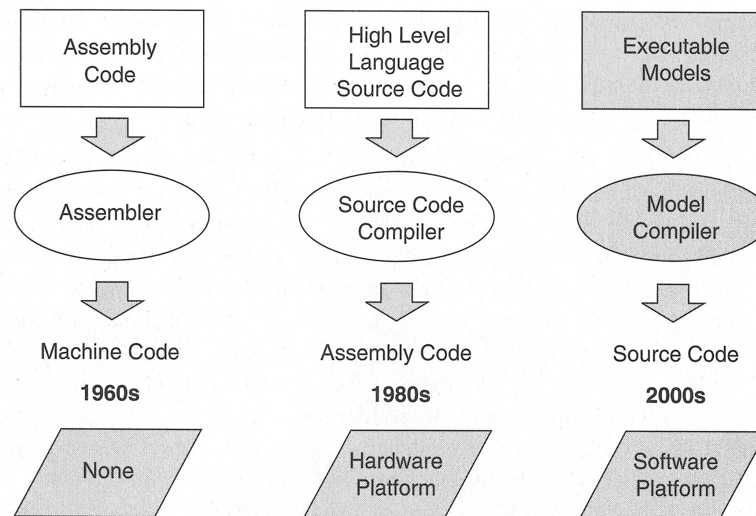
Vuonna 1965 oli proseduraalisten ohjelmointikielten esiintulon aika, josta alkoi myös *sovelluskeskeinen aikakausi* (Kleppe & al., 2003; Frankel, 2003). Nämä kolmannen sukupolven ohjelmointikielet kehitettiin symbolisten konekielten päälle ja ominaista niille on se, että ohjelmat käännetään ennen ajoa konekielisiksi kääntäjien avulla tai ajonaikana tulkeilla. Kleppen & al. (2003) mukaan kaikki ohjelmoijat eivät suinkaan olleet samaa mieltä kääntäjien mukanaan tuomista eduista, koska heidän mielestään koodin tehokas optimointi oli tärkeää. Ensimmäiset kääntäjät eivät tehneetkään optimointia niin hyvin kuin ohjelmoijat olisivat itse pystyneet tekemään. Ajan kuluessa ja kääntäjien kehittyessä tehokkuuden kasvu oli kuitenkin merkittävämpi hyöty kuin optimoinnin hienosäätö, joten kääntäjät olivat tulleet pysyäkseen. Kääntäjien ja tulkkien kehittäminen yhtäaikaisesti ohjelmointikielten kanssa oli tärkeää, jotta voitiin turva-

ta jatkuva kehitys ja sen myötä automaatio abstraktiotasojen välisissä muunnoksissa. Oleellisinta proseduraalissa ja ylipäätänsä kolmannen sukupolven ohjelmointikielissä on se, että niistä pyrittiin tekemään ohjelmoijalle helpommin omaksuttavia ja ilmaisuvoimaisempia kuin edeltäjänsä, mikä oli taas luonnollinen seuraus Mellorin & Balcerin (2002) esittelemästä abstraktiotason nostamisesta — kehityksen suunta oli selvä.

Sovelluskeskeinen aikakausi jatkui edelleen vuonna 1985, jolloin katsottiin alkaneen nykypäivään saakka kestävä *olio-ohjelmointikielten aikakausi* (Kleppe & al., 2003). Olio-ohjelmointikielet saivat proseduraalisten kielten hyväksi havaittujen ominaisuuksien lisäksi uudenlaisen rakenteen ja kokonaan uuden ajattelutavan. Frankelin (2003) mielestä tämä oli seurausta siitä, että vaatimukset yhä tehokkaammasta ja samalla tuottavammasta ohjelmoinnista sekä monimutkaisemmista ominaisuuksista kasvoivat. Perinteinen ohjelmistotuotanto kohtasi ongelmia, joiden ratkaisemiseksi tarvittiin uusia tekniikoita. Näistä ongelmista sekä malliperustaisen arkkitehtuurin tarjoamista ratkaisuista kerrotaan lisää luvussa 3.

Kuten aiemmin todettiin, siirtyminen eri historiallisten jaksojen ohjelmointikielestä toiseen toteutettiin abstraktiotason nostamisella. Mellorin & Balcerin (2002) mukaan uusin ohjelmointikieli muunnetaan aina edellisen tason kielen mukaiseksi yleensä automaattisesti *kuvausten*[1] (mappings) avulla. Jotta menetelmää voidaan ryhtyä käyttämään, joudutaan uusin ohjelmointi- tai mallinnuskieli aina ensin esittelemään käsitteiden avulla käyttäjilleen. Näin ollen alemman tason yksityiskohdat jäävät suurimmalta osalta käyttäjistä piiloon — kuten on tarkoituskin. Ajan kuluessa käsitteet formalisoituvat ja apuvälineet kehittyvät, jolloin ne otetaan yleisesti käyttöön. Erilaisten ohjelmointikielten kehitys aiheutti abstraktioketjujen syntymisen, mikä osaltaan mahdollisti siirtymisen malliperustaiselle tasolle. Mellor & al. (2004) havainnollistavat kuvan 1 mukaisesti kuinka abstraktiotason nostaminen on vaikuttanut kehitykseen ja mahdollistanut uudelle tasolle siirtymisen. *Laitteistoriippumattomilla* ohjelmointikielillä, kuten C ja Java, voidaan ohjelmoida järjestelmiä, jotka toimivat nimensä mukaisesti erilaisilla laitealustoilla. Samalla tavalla *ohjelmistoriippumattomia* järjestelmiä voidaan rakentaa käyttäen hyväksi *suorituskelpoisia malleja* (executable models), joista voidaan generoida lähdekoodia mallikääntäjien avulla halutuille ohjelmistoalustoille, mallia muuttamatta. Automaattisen koodingeneroinnin eri tapoja ja tasoja selvitetään luvussa 4 sekä myös kohdassa 5.2 käytännön esimerkin avulla.

[1] Kuvaukset ovat funktiota, joilla kaavan alkiot voidaan muuntaa vastaaman toisen kaavan alkioita.

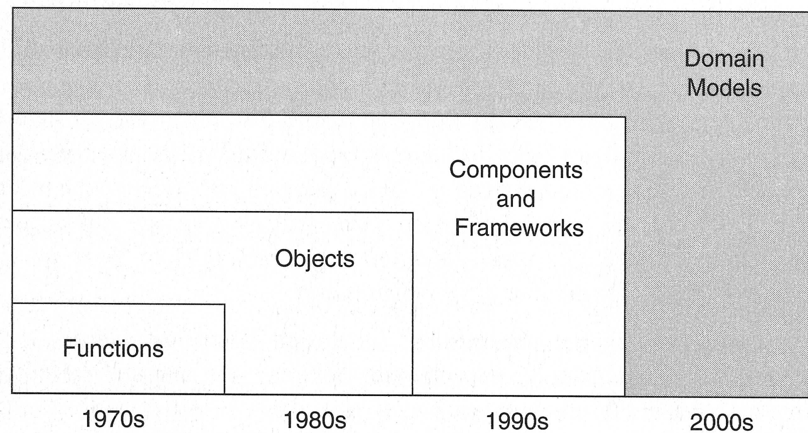


Kuva 1: Abstraktiotason nostaminen (Mellor & al., 2004).

Mellorin & al. (2004) mukaan *uudelleenkäytön lisääminen* on abstraktiotason nostamisen ohella toinen tärkeä esille tuotava seikka tarkasteltaessa ajatusta malliperustaiseen arkkitehtuuriin siirtymisestä. Ensimmäisissä systeemeissä koodin uudelleenkäyttöön kiinnitettiin paljon huomiota, koska tietokoneissa oli vähän muistia ja kaiken lisäksi se oli erittäin kallista. Muistin säästämiseksi sekä uudelleenkäytön tehostamiseksi kehitettiin funktiot, jotka mahdollistivat kirjastojen kokoamisen. Funktioiden hyöty perustuu siihen, että tietyt funktiot, yleensä matemaattiset, palauttavat aina saman arvon syötteen ollessa sama. Oliot, jotka pitävät sisällään aliohjelmia sekä tietorakenteita, edesauttoivat uudelleenkäytön lisäämistä esittelemällä muun muassa periyttämisen. Tämä ei kuitenkaan vielä riittänyt, vaan tarvittiin jotain muuta.

Seuraava askel eteenpäin toi mukanaan komponentit ja siten Frankelin (2003) kuvaaman *komponenttikeskeisen kehityksen*. Komponenttikeskeinen kehitys nopeuttaa ohjelmistojen toteutusta ja suunnittelua sekä parantaa laatua, koska eri projekteissa voidaan käyttää valmiita komponentteja uudelleenkeksimisen sijasta ja näin ollen aikaa jää enemmän muihin tärkeisiin asioihin. Komponentit sisältävät läheisesti toisiinsa liittyvät oliot paketoituna yhteen, mikä mahdollistaa siis koodin korkean uudelleenkäytön. Tämäkään ratkaisu ei edellisen tapaan ole ongelmaton, joten Mellor & al. (2004) suosittelevat, että ohjelmistoprojekteissa työ jaettaisiin horisontaalisiin osiin — *toimialuemalleihin* (domain models). Nämä mallit voivat olla sellaisia kuten pankki, käyttäjäntunnistus ja käyttöliittymä, joille määritellään tietyt säännöt ja rajoitteet. Kuva 2

havainnollistaa kuinka uudelleenkäytön kehitys on edennyt historian kuluessa.



Kuva 2: Uudelleenkäytön lisääminen (Mellor & al., 2004).

Muutama keskeisin seikka malliperustaisuuden syntyyn johtaneista asioista on nyt käsitelty, joten voidaan palata vuoteen 1990 ja siihen kuinka itse MDA syntyi. Millerin & Mukerjin (2001) mukaan kyseisestä vuodesta lähtien OMG[2] otti selkeän päämäärän ohjelmistoarkkitehtuuristandardien kehityksessä — sovellusten välisen integraatio-ongelman ratkaisemisen. Tähän ongelmaan kehitettiin aikanaan arkkitehtuuri, OMA (Object Management Architecture), joka toimi suunnannäyttäjänä vuosien ajan laajentuen lopuksi malliperustaiseksi arkkitehtuuriksi (OMG, 2004c).

CORBA (Common Object Request Broker Architecture) oli tärkeä merkkipaalu OMG:n historiassa, koska moni jälkeenpäin kehitetty standardi sekä määritys perustui nimenomaan siihen. OMA ja CORBA keskittyivät kuitenkin pelkästään hajautettujen järjestelmien toteutukseen, joten tilaa oli uudelle teknologialle. Vuonna 1997 OMG esitteli monia uusia standardeja, jotka eivät olleet CORBA -pohjaisia (Miller & Mukerji, 2001). Esimerkkeinä näistä kohdassa 2.3 esiteltävistä standardeista voisi mainita UML:n (Unified Modeling Language) sekä hieman myöhemmin tulleen XMI:n (XML Metadata Interchange). Molemmat standardit ovat olennainen osa malliperustaista arkkitehtuuria, joka sai alkunsa vuonna 2001 OMG:n toimesta. Se oli suoraa seurausta kehityksestä, jossa pyrittiin koko ajan korkeammalle abstraktiotasolle niin ohjelmoinnissa kuin mallinnuksessakin. Luvussa 2 pyritään selvittämään malliperustaisen arkkitehtuurin keskeisimmät käsitteet ja kerrotaan lyhyesti joistakin OMG:n standardeista.

[2]Object Management Group (www.omg.org) on voittoa tavoittelematon konsortio, joka tuottaa ja ylläpitää tietotekniikan standardeja sekä spesifikaatioita. MDA:n lisäksi muita OMG:n määrityksiä ovat esimerkiksi CORBA, MOF, UML ja XMI.

1.1 Tutkielman rakenne

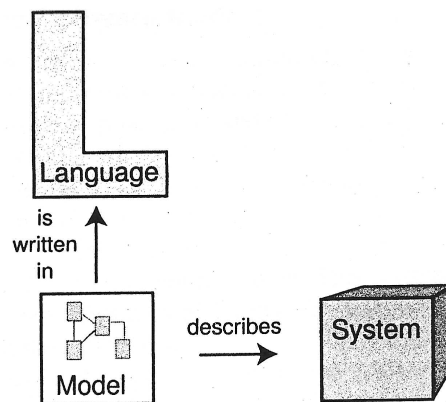
Abstraktiotason nostaminen ja uudelleenkäytön lisääminen ovat olleet luonnollisena lähtökohtana malliperustaiselle arkkitehtuurille. Nämä kaikki edellyttävät alustaksi erilaisia standardeja, joita tarkastellaan luvussa 2. Ohjelmistotuotannon kehitysprosesseista niin perinteisten menetelmien kuin malliperustaisenkin prosessin osalta kerrotaan luvussa 3. Perinteisten kehitysprosessien yhteydessä kuvataan myös ongelmat, jotka omalta osaltaan edesauttoivat malliperustaisen arkkitehtuurin syntymistä. Malliperustaisesta kehitysprosessista kuvataan pääpiirteet sekä MDA:n mukanaan tuomat ratkaisut aiemmin mainittuihin perinteisen ohjelmistotuotannon ongelmiin. Luku 4 käsittelee koodingenerointia esittelemällä muutamia erilaisia koodingeneroinnin muotoja sekä hyötyjä, joita automaattisella lähdekoodin muodostamisella voidaan saavuttaa. Luvussa 5 kootaan yhteen aiemmin esitelty teoria käytännön esimerkin avulla. Tutkielman päättää luku 6, jossa esitetään tutkielman perusteella tehdyt johtopäätökset sekä pohditaan hieman malliperustaisen arkkitehtuurin tulevaisuudennäkymiä.

2 MDA-kehys

Tässä luvussa kuvataan MDA-kehystä ja sen tärkeimpiä käsitteitä, jotta voidaan myöhemmin ymmärtää paremmin kohdassa 3.2 esitettävä MDA-kehitysprosessi. Luvun loppupuolella kerrotaan malliperustaiseen arkkitehtuuriin läheisesti liittyvistä standardeista ja niiden merkityksestä.

2.1 Mallit

Mallit ovat nimensä mukaisesti keskeisessä roolissa tarkasteltaessa malliperustaista arkkitehtuuria. Sanalla malli on monia merkityksiä, mutta tässä tapauksessa keskitytään malleihin, jotka liittyvät ohjelmistojen kehittämiseen. Mallit voivat olla hyvin erilaisia niin luonteeltaan kuin tarkkuudeltaankin riippuen tietenkin käyttötarkoituksesta.



Kuva 3: Mallinnuskielen, mallin ja systeemin suhde (Kleppe & al., 2003).

Kleppe & al. (2003) käyttävät sanaa *systeemi* kuvaamaan mallien konkreettista ilmentymää (kuva 3). Systeemi tarkoittaa siis ohjelmistomallin kohdalla itse ohjelmistoa ja liiketoimintamallin kohdalla liiketoimintaa. Luotujen mallien avulla voidaan tuottaa toinen samalla tavalla toimiva systeemi, mutta sen ei välttämättä tarvitse olla toteutukseltaan identtinen. Mallit kuvaavat systeemiä pääpiirteittäin, joten yksityiskohdat voivat vaihdella. Mitä tarkemmin mallien kuvaukset on tehty, sitä identtisemmät lopulliset systeemit tulevat olemaan toteutukseltaan. Mellorin & al. (2004) mukaan seuraavat neljä ominaisuutta muodostavat hyvän mallin perustan:

1. *Yksityiskohtaisin informaatio* on jätetty pois havainnoijan käsityksen selkiyttä-

miseksi, joten malli ei ole täsmälleen samanlainen kaikilta osin kuin oikea mallinnettava asia.

2. Hyvä malli kuvaa oikean käsityksen saamiseksi *riittävän tarkasti* todellista, abstraktia tai hypoteettista todellisuutta, vaikka siitä on jätetty pois informaatiota edellisen kohdan perusteella.

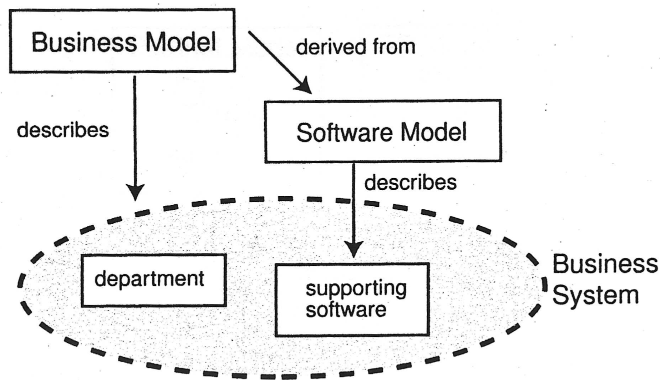
3. Mallin on ehdottomasti oltava *halvempi rakentaa* kuin oikea asia. Tässä tapauksessa halvin ei tarkoita pelkästään rahaa, vaan myös aikaa ja työpanosta.

4. Malli palvelee kommunikaatiota muodostamalla katsojalleen helpommin ja nopeammin *uusia ideoita* sekä ehkäisee tehokkaasti *väärinkäsitysten* syntymistä ihmisten ja tietokoneiden välillä.

Jotta mallien hyödyt saadaan esille automaattisten muunnosten avulla, tarvitaan niiden tekemiseen jokin tietokoneen ymmärtämä, sekä syntaksisesti että semanttisesti hyvin määritelty mallinnuskieli. Tällainen mallinnuskieli voi olla esimerkiksi yleisesti käytetty, kohdassa 2.3.2 tarkasteltava UML. MDA ei kuitenkaan rajoita mallinnuskielen käyttöä pelkästään UML:ään, vaan se voi toki olla mikä tahansa edellä mainitut ehdot täyttävä kieli (Kleppe & al., 2003).

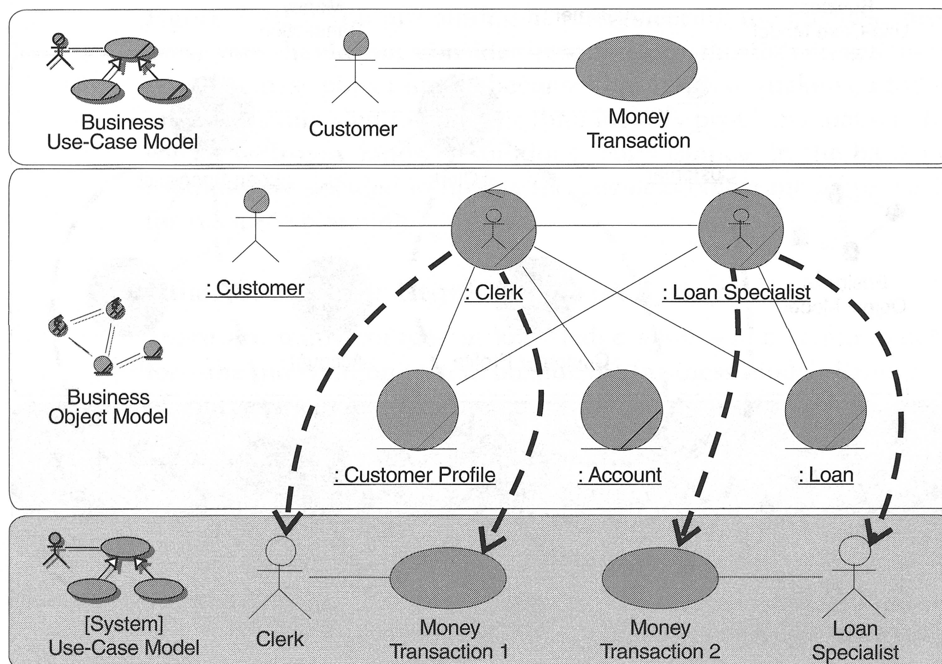
2.1.1 Liiketoiminta- ja ohjelmistomalli

Liiketoimintamallien (business models) avulla voidaan Kleppen & al. (2003) mukaan kuvata yrityksen liiketoimintaa tai jotakin osaa siitä. Liiketoimintamalli ei välttämättä ota kantaa yrityksen sisäisiin ohjelmistoratkaisuihin, minkä vuoksi sitä voidaan kutsua *tietokoneriippumattomaksi malliksi* (Computational Independent Model, CIM). CIM on riippumaton käytettävästä alustasta, mutta se voi sisältää osia, jotka on määritelty jollekin tietylle ohjelmistoalustalle (Miller & Mukerji, 2001). *Ohjelmistomallit* (software models) muodostetaan liiketoimintamallien perusteella siten, että ohjelmistomallin vaatimukset periytyvät kuvan 4 mukaisesti liiketoimintamallista, tai sen osasta, jota ohjelmistomallin täytyy tukea. Kleppe & al. (2003) huomauttavat, että useimmiten yksi ohjelmistomalli ei riitä, vaan tarvitaan useita ohjelmistosysteemejä varustettuna erilaisilla malleilla mallintamaan tiettyä liiketoiminnan osaa. Liiketoiminta- ja ohjelmistomallien erot havainnollistetaan kuvan 5 käyttötapauskaavioiden avulla.



Kuva 4: Ohjelmisto- ja liiketoimintamallin suhde (Kleppe & al., 2003).

Liiketoiminnan mallintamisella pyritään Kruchtenin (2004) mukaan organisaation rakenteen, dynamiikan sekä mahdollisten ongelmien ymmärtämiseen. Sillä varmistetaan myös se, että asiakkaat, loppukäyttäjät ja systeemin kehittäjät ymmärtävät rakennettavan järjestelmän kohdealueen samalla tavalla. Yksi tärkeimmistä syistä on kuitenkin, jo aiemmin mainittu, ohjelmistosysteemin vaatimusten periyttäminen organisaation liiketoimintaa kuvaavista malleista.

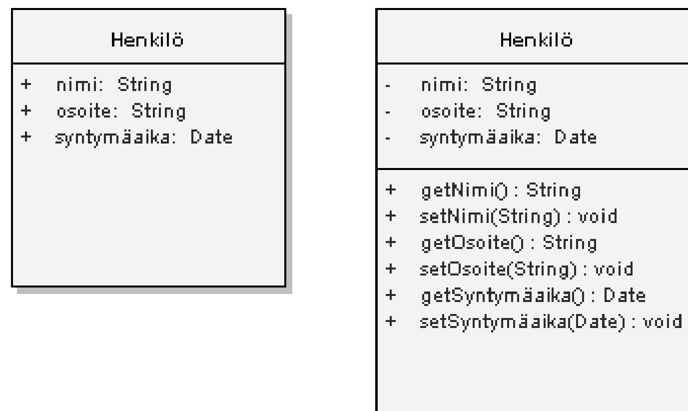


Kuva 5: Pankin liiketoiminnan kuvaamisesta systeemin kuvaamiseen (Kruchten, 2004).

2.1.2 PIM ja PSM

MDA-dokumentaatio (Miller & Mukerji, 2001) esittelee todennäköisesti malliperustaisuuden kaksi keskeisintä mallia: *sovellusalustariippumattoman mallin* (Platform Independent Model, PIM) ja *sovellusalustariippuvaisen mallin* (Platform Specific Model, PSM). PIM mahdollistaa rakenteiden formaalin määrittelyn sekä systeemin toiminnallisuuden kuvaamisen siten, että kaikki tekniset yksityiskohdat ovat piilotettuina. PIM-malleja ei voida Kleppen & al. (2003) mukaan kuitenkaan automaattisesti johtaa kohdassa 2.1.1 esitellyistä tietokoneriippumattomista malleista, koska ohjelmistosysteemin tukemat osat on aina valittava ihmisen toimesta.

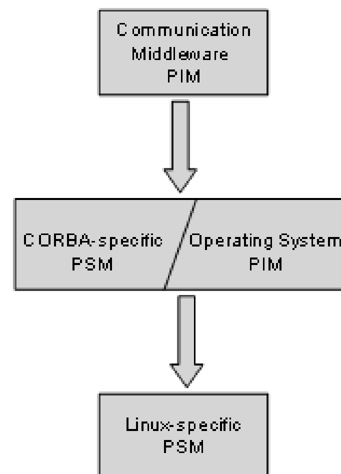
Sovellusalustariippuvaisen mallin avulla voidaan realisoida sovellusalustariippumattoman mallin toiminnallisuus tietylle alustalle kohdassa 2.2 kuvattavan *transformaation* kautta, jolloin myös samalla mahdollistetaan koodingenerointi (Mellor & Balcer, 2002). Kuvassa 6 vasemmalla oleva luokka on sovellusalustariippumaton, kun taas oikealla oleva luokka on kohdennettu Javalle. PIM-mallissa attribuutit merkitään julkisiksi, koska tällä tavalla halutaan ilmaista, että niiden arvo voi muuttua ajan kuluessa. PSM-mallin attribuutit ovat olioparadigman mukaisesti yksityisiä, jolloin niiden arvoihin päästään käsiksi vain kuhunkin tarkoitukseen soveltuvalla funktiolla.



Kuva 6: PIM ja Javalle kohdennettu PSM.

Kleppen & al. (2003) mukaan malli on aina joko PIM tai PSM, mutta käytännössä niiden erottaminen voi olla vaikeaa. Kuvassa 7 Brown (2004) esittelee hyvän esimerkin siitä kuinka jonkun henkilön PIM voi olla toiselle PSM, mikä vahvistaa käsitystä siitä, että käsitteet PIM ja PSM ovat erittäin suhteellisia. Malli voi olla PIM sillä perusteella,

että kommunikointiväliohjelmisto on valinnainen, eli kyseinen malli ei ennalta määrää käytettäväksi tiettyä teknologiaa. Kun päätös väliohjelmistosta (esimerkiksi CORBA:sta) tehdään, muunnetaan malli CORBA:n mukaiseksi PSM-malliksi. Tämä uusi malli voi silti olla PIM siten, että käyttöjärjestelmän sekä laitteiston lisäksi myös ORB voi olla toteutukseltaan valinnanvarainen.



Kuva 7: Mallien suhteellisuus (Brown, 2004).

Wang & Zhang (2003) luettelevat muutamia merkityksellisiä etuja, joita teknisesti tarkan rakenteen kuvaamisen abstrahointi sovellusalustariippumattomaksi malliksi tuo esille:

- Mallien validointi helpottuu, koska sovellusalustan semantiikka ei vaikuta. PSM-mallit joutuvat ottamaan kantaa esimerkiksi virheenkäsittelyyn ja muuttujien tyyppeihin.

- Toteutusten tuottaminen liiketoimintamalleista erilaisille sovellusalustoille helpottuu, koska rakenteet ja systeemin käyttäytyminen ovat helposti mukautettavissa.

- Systeemien välinen integraatio ja *yhteentoimivuus* (interoperability) voidaan määritellä selkeämmin sovellusalustariippumattomin termein ja vasta sen jälkeen muuntaa jokaisen sovellusalustan omilla mekanismeilla niille sopiviksi.

Edellä mainittujen etujen saavuttamiseksi käytetään usein UML-mallinnuskieltä, jolla sovellusalustariippumattomat mallit voidaan kuvata paremmin kuin esimerkiksi IDL-

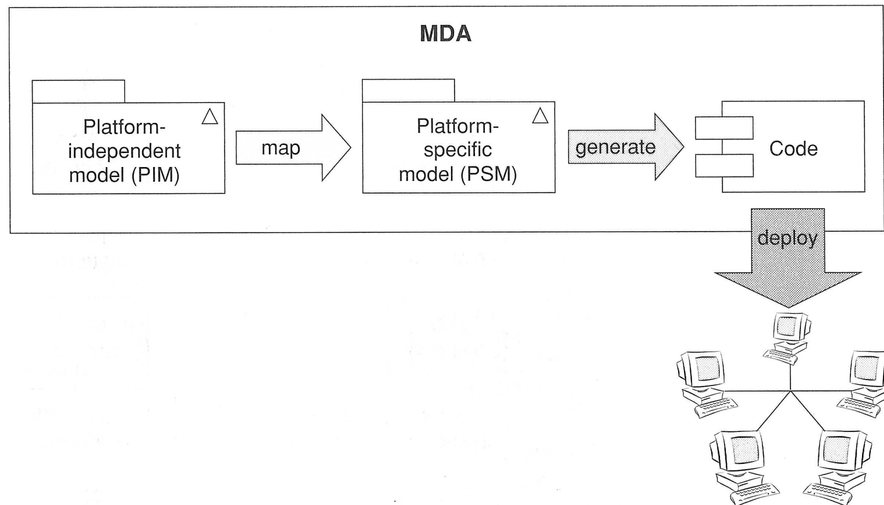
pohjaisilla (Interface Definition Language) ratkaisuilla. Tämä paremmuus johtuu Millerin & Mukerjin (2001) mukaan siitä, että UML perustuu tarkasti kuvattuun metamalliin. UML-mallit voidaan myös esittää sekä graafisessa- että tekstimuodossa[3]. UML-mallit ovat semanttisesti paljon laajempia kuin IDL-perustaiset mallit, joilla voidaan ilmaista hyvin syntaksia, mutta heikosti mallien rajoitteita sekä käyttäytymistä. Siegel (2001) lisää etuihin myös sen, että PIM-mallit ovat todellakin sovellusalustariippumattomia, jolloin liiketoiminnan ammattilaiset voivat rauhassa suunnitella mallin toiminnallisuuden ilman teknologian mukanaan tuomia rajoitteita. Tämä mahdollistaa myös mallien kritisoinnin ja korjaamisen ennen toteutusvaiheen alkamista. Myöhemmin ohjelmoinnin aikana tai sen jälkeen tapahtuvat muutokset ovat moninkertaisesti kalliimpia tehdä kuin mallinnusvaiheessa havaitut, joten etu on merkittävä.

2.2 Transformaatio

Transformaatiolla tarkoitetaan MDA:n yhteydessä mallien välisiä muunnoksia, jolloin malleina toimivat aiemmin esitellyt PIM ja PSM sekä ohjelmakoodi. Kleppe & al. (2003) kuvailevat transformaatiota prosessiksi, jossa alkuperäisestä mallista generoidaan automaattisesti kohdemalli *transformaatiomääritysten* avulla. Transformaatiomääritykset taas koostuvat joukosta *transformaatiosääntöjä*, jotka yhdessä kuvaavat kuinka alkuperäinen malli voidaan muuntaa kohdemallin kielelle. Näin ollen säännöt ovat kuvauksia siitä kuinka yksittäisen rakenteen tai monien rakenteiden muunnos alkuperäiseltä kieleltä toiselle tehdään. Kuvassa 8 havainnollistetaan mallien välisten muunnosten eteneminen sovellusalustariippumattomasta mallista sovellusalustariippuvaisen mallin kautta lopulliseksi käännöskelpoiseksi ohjelmakoodiksi.

Transformaatioissa käytetään apuvälineinä erilaisia *muuntotyökaluja* (transformation tool), jotka voidaan valita esimerkiksi niiden sisältämien ominaisuuksien perusteella. Muuntotyökalu toimii siten, että se ottaa ensin syötteenä PIM-mallin ja muuntaa sen transformaatiomääritysten mukaisesti PSM-malliksi. Tämän jälkeen PSM voi mennä syötteenä jollekin toiselle työkalulle tai kuten yleensä, sama työkalu huolehtii myös muunnoksen koodiksi, jolloin muunnosten tekeminen helpottuu entisestään. Useiden työkalujen kanssa PSM jää näkymättömäksi käyttäjälle, koska mallit muunnetaan suoraan ohjelmakoodiksi. Muunnoksia voidaan suorittaa myös muiden kuin pelkästään

[3]UML-mallit voidaan esittää tekstimuodossa käyttäen hyväksi esimerkiksi OMG:n XMI:tä, jolloin niitä voidaan vaihtaa eri CASE (Computer Aided Systems Engineer) -välineiden välillä.



Kuva 8: Transformaatio (Arlow & Neustadt, 2003).

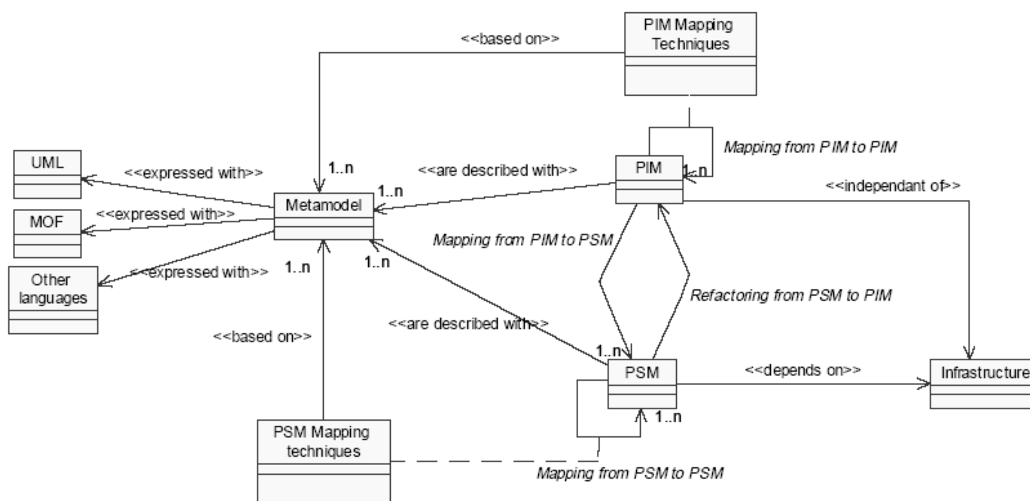
aiemmin mainittujen mallien välillä. Malliperustaisen arkkitehtuurin metamallia esittävästä kuvasta 9 voidaan huomata muun muassa Millerin & Mukerjin (2001) luettelemat neljä erilaista mallien välistä muunnosta sekä tilanteita, joissa muunnosta voidaan käyttää:

1. **PIM – PIM.** Kahden sovellusalustariippumattoman mallin välistä muunnosta käytetään, kun malleja tarvitaan parantaa, suodattaa tai erikoistaa kehitysprosessin aikana ilman, että sovellusalustariippuvaista informaatiota täytyisi lisätä. Kleppe & al. (2003) mainitsevat oliosuhdemallin normalisoinnin esimerkkinä tavanomaisesta käytöstä.

2. **PIM – PSM.** Tätä muunnosta käytetään, kun halutaan muuntaa sovellusalustariippumaton malli halutulle sovellusalustalle sopivaksi malliksi.

3. **PSM – PSM.** Tätä muunnosta tarvitaan, kun kyseessä ovat komponenttien realisointi ja sijoittelu. PSM – PSM muunnokset liittyvät yleensä sovellusalustariippuvaisten mallien kehittämiseen sekä parantamiseen.

4. **PSM – PIM.** Mallien abstrahointi olemassa olevista toteutuksista vaatii PSM – PIM muunnosta, jota kutsutaan myös *käänteistekniikaksi* (reverse engineering) (Cooper & al., 2004). Tämä tapahtuma vaatii paljon suodatusta, joten sitä on vaikea automatisoida kokonaan. Ihannetapauksessa tulos vastaa täysin PIM – PSM muunnosta.



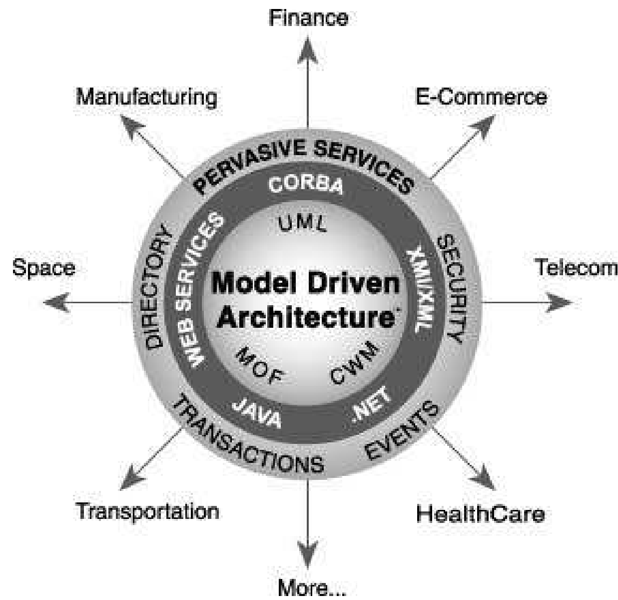
Kuva 9: Malliperustaisen arkkitehtuurin metamalli (Miller & Mukerji, 2001).

UML:llä mallinnettujen sovellusalustariippumattoman sekä sovellusalustariippuvaisen mallin välinen transformaatio voidaan suorittaa Millerin & Mukerjin (2001) mukaan monilla tavoilla. Sovelluskehittäjä voi suorittaa transformaation käsin rakentamalla sovellusalustariippuvaisen mallin sovellusalustariippumattoman mallin perusteella. Apuna muuntamisessa hänellä voi olla tunnettuja *muunnoskaavoja* (refinement patterns), jolloin työ hieman helpottuu. Sovelluskehittäjä voi käyttää transformaatiossa myös algoritmeja, jotka muodostavat sovellusalustariippumattomasta mallista *ohjelmarunkoja* (skeleton), joita sitten täydennetään käsin mahdollisten muunnoskaavojen avulla. Algoritmi voi myös luoda automaattisesti täydellisen sovellusalustariippuvaisen mallin täydellisestä sovellusalustariippumattomasta mallista muiden työkalujen avulla.

2.3 Standardit

MDA-dokumentaation (Miller & Mukerji, 2001) mukaan malliperustainen arkkitehtuuri koostuu useista eri osista. Kuvasta 10 voidaan erottaa MDA:n suhteet OMG:n muihin standardeihin, eri sovellusalustoihin, *jokapaikanpalveluihin* (pervasive services) sekä *toimialamäärityksiin* (domain specifications).

Malliperustaisen arkkitehtuurin ydin muodostuu OMG:n mallinnusstandardeista, joita ovat Unified Modeling Language (UML), Meta Object Facility (MOF) ja Common Warehouse Metamodel (CWM). Standardien lisäksi olennaisena osana ydintä ovat myös UML-profiilit, jotka kuvaavat tietylle sovellusalustalle, kuten CORBA:lle



Kuva 10: OMG:n malliperustainen arkkitehtuuri (OMG, 2004b).

tai web-palvelutekniikoille, käytettävän merkintätavan. Uloimman kehän kuvassa 10 muodostavat jokapaikanpalvelut, joiden toteutus vaihtelee sovellusalustamäärityksestä riippuen. Tällaisia palveluita voivat olla muun muassa hakemistopalvelut, tapahtumankäsittely, pysyvyyden takaaminen, transaktiot sekä tietoturva. Jokapaikanpalvelut kuvataan sovellusalustariippumattomassa mallissa UML-mallinnuskielellä ottamatta tietenkään kantaa käytettävään sovellusalustaan. Myöhemmin muunnettaessa mallia sopivaksi tietylle sovellusalustalle, voidaan käytetyn sovellusalustan mukaiset palveluiden kutsut generoida automaattisesti. Kuvan 10 reunoilla ovat toimialamääritykset. Ne pyrkivät kuvaamaan jokaiselle toimialalle oman UML-merkintätavan, jonka avulla voidaan tuottaa PIM ja PSM -mallit. Tällä hetkellä toimialamäärityksiä on tehty OMG:n toimesta yhteensä jo 34 (OMG, 2004a). Jotta ymmärrettäisiin paremmin malliperustaisen arkkitehtuurin sisältöä, on syytä perehtyä hieman tarkemmin OMG:n määrittelemiin standardeihin. Seuraavaksi esitellään MDA:han läheisesti liittyvät määritykset MOF, UML, CWM ja XMI.

2.3.1 Meta Object Facility

MOF[4] on OMG:n vuonna 1997 ratifioima standardi teknologiasta riippumattomien metamallien määrittelyyn, rakentamiseen ja hallinnoimiseen (OMG, 2002). MOF on ylin kerros taulukon 2 mukaisessa nelikerrosarkkitehtuurissa, jonka muodostavat seuraavat tasot: *meta-metamalli* (M3), *metamalli* (M2), *malli* (M1) sekä *data* (M0).

Taulukko 2: MOF-metadata-arkkitehtuuri (Damm & al., 2000; OMG, 2002).

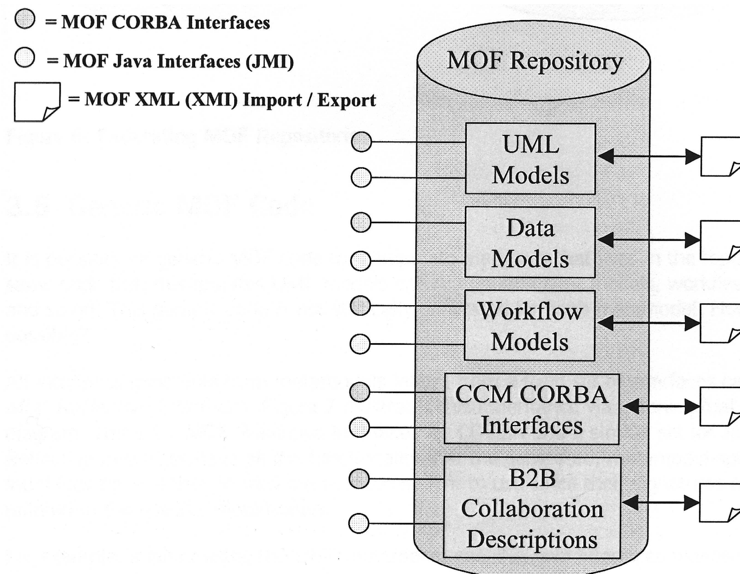
Metataso	*MOF-termi*	*Kuvaus*	*Esimerkki*
M3	meta-metamalli	Tällä tasolla mallinnetaan metamallit eli taso kuvaa kielen, jolla metamallit määritellään.	MOF-malli
M2	metamalli, meta-metadata	Meta-metamallin ilmentymä eli kuvaa säännöt, joilla malli voidaan määritellä. CASE-välineiden rakentajat työskentelevät tällä tasolla.	UML-metamalli
M1	malli, metadata	Metamallin ilmentymä, jolla kuvataan mallinnettavana olevan kohteen informaatio. Systeeminsuunnittelijat toimivat tällä tasolla.	UML-malli pankista
M0	data	Mallin ilmentymät eli oliot.	Konkreettisen pankkitilin objektit

Esimerkkeinä MOF:in avulla rakennetuista metamalleista, eli *MOF-metamalleista*, voidaan mainita muun muassa UML, CWM ja IDL. MOF:in kaltaista standardia tarvittiin, koska huomattiin, että ohjelmistoteollisuuden keskuudessa käytettiin keskenään erilaisia keinoja mallinnusrakenteiden kuvaamiseksi (Frankel, 2002). MOF:in suunnitelleet arkkitehdit eivät edes yrittäneet yhdistää näitä monia tekniikoita yhdeksi, vaan ryhtyivät kehittelemään universaalia tapaa kuvailla erilaiset mallinnusrakenteet.

MOF käyttää hyväksi UML:n olioperustaisia luokkarakenteita, minkä vuoksi MOF-

[4] Tässä tutkielmassa keskitytään kuvaamaan MOF 1.4:n ominaisuuksia. MOF 2.0 on tällä hetkellä viimeistelyn kohteena, joten sen lopullinen versio ilmestynee piakkoin.

metamallit näyttävät UML:n luokkakaavioilta. Tästä johtuen niitä voidaan rakentaa tavallisilla UML:ää tukevilla CASE-välineillä. MOF-standardi (OMG, 2002) kuvailee myös kehyksen metadataa sisältävien *kuvauskantojen* (repository) toteuttamiseksi. Kehyksessä muunnetaan MOF-metamallit metadatan ohjelmointirajapinnoiksi, mikä mahdollistaa yhtenäisen ja yhteentoimivan metadatakuvauskannan käytön eri laitevalmistajien sekä toteutustekniikoiden keskuudessa. Kuvassa 11 havainnollistetaan MOF-kuvauskannan käyttöä ja sen sisältämiä rajapintoja.



Kuva 11: MOF-kuvauskanta (Frankel, 2002).

Kuvauskanta voi pitää sisällään esimerkiksi MOF-CORBA, MOF-Java ja MOF-XML väliset kuvaukset, joiden avulla esimerkiksi UML-malleja voidaan käsitellä halutulla tavalla. Esimerkiksi *Java Metadata Interface* (JMI) mahdollistaa kuvauskannassa olevien mallien tutkimisen Javan avulla (JCP, 2002). Tätä tekniikkaa hyödyntää muun muassa tämän tutkielman esimerkissä käytetty koodingeneroija, josta on kerrottu lisää kohdassa 5.1. Esimerkkinä Javan avulla suoritetusta mallin tutkimisesta voisi toimia luokan abstraktisuuden selvittäminen:

```
boolean isAbstract = myClass.getIsAbstract();
```

Malliperustaiseen arkkitehtuuriin MOF liittyy siten, että se käyttää MDA:n periaatteita toiminnassaan. Frankelin (2002) mielestä MOF on yksi puhtaimmista olemassa olevista MDA-sovelluksista. MOF:in avulla muodostetut metamallit ovat sovellusalustariip-

pumattomia malleja, jotka ovat kuvauksien avulla muunnettu jokaiselle sovellusalustalle sopivaksi, mikä on MDA:n keskeinen lähtökohta. Näin ollen hyöty MOF:sta koetaan metadatan hallintaan tarkoitettujen ohjelmistojen kehityksessä, jossa sovellusalustariippumattomuus sekä lisääntynyt automaatio ovat keskeisessä roolissa.

2.3.2 Unified Modeling Language

UML on OMG:n kehittämä graafinen mallinnuskieli, joka on suunniteltu erilaisten ohjelmistojen määrittelemiseksi, rakentamiseksi, havainnollistamiseksi sekä dokumentoimiseksi (OMG, 2003b). Malliperustaisessa arkkitehtuurissa UML:ää käytetään sekä sovellusalustariippumattoman että sovellusalustariippuvaisen mallin kuvaamiseen. UML:n versio 1.1 hyväksyttiin viralliseksi standardiksi marraskuussa 1997, jonka jälkeen se on kehittynyt merkittävästi useiden versioidensa ansiosta (Kobryn, 1999). Lukuisat yritykset ovat edesauttaneet UML:n kehitystä tukemalla OMG:n toimintaa sekä ennen kaikkea hyödyntämällä mallinnuskieltä omissa projekteissaan. Näin ollaan päästy nykyiseen viralliseen versioon UML 1.5[5].

Suunnitellessaan UML-mallinnuskieltä OMG on halunnut keskittyä erityisesti seuraavien tavoitteiden toteuttamiseen (OMG, 2003b). UML:n tulisi tarjota käyttäjille valmis, ilmaisuvoimainen mallinnuskieli, jonka avulla voidaan rakentaa ja vaihtaa malleja. Sen tulisi antaa mahdollisuus käyttää erilaisia mekanismeja laajentamaan mallinnuksen peruskäsitteitä. UML:n tulisi myös tukea tietystä ohjelmointikielestä sekä ohjelmistoprosessista riippumattomia määrityksiä, millä pyritään kasvattamaan eri tekniikoiden yhteensopivuutta. Tuki korkean tason käsitteille, kuten komponenteille, yhteistoiminnalle, kehyksille ja suunnittelukaavoille auttavat tässä hankkeessa.

Tavallisesti UML:ää käytetään ohjelmistojärjestelmien mallinnuksessa, mutta se soveltuu erinomaisesti myös moniin muihin järjestelmiin. Kannattaa huomata, että useimmat järjestelmät eivät kuulu pelkästään yhteen ryhmään, vaan useampien yhdistelmään. Eriksson & Penker (2000) luettelevat esimerkkejä erilaisista järjestelmistä, joita voidaan mallintaa UML:n avulla:

[5]Tässä tutkielmassa käsitellään erityisesti UML 1.5:n sisältämiä ominaisuuksia, koska seuraava OMG:n kehittelemä versio UML 2.0 ei ole vielä tämän tutkielman kirjoittamisen aikana lopullisesti valmis. Versiota UML 2.0 käsitellään kuitenkin muutamien malliperustaiseen arkkitehtuuriin liittyvien ominaisuuksien osalta.

- Suuria tietomääriä käsittelevät, relaatio- tai oliotietokantoihin tallentavat *tietojärjestelmät*, joilla on monimutkaisia yhteyksiä, voidaan mallintaa UML:n avulla. Tällaisten tietojärjestelmien tehtäviä ovat yleensä tiedon tallennus, hakeminen, muuttaminen ja käyttäjälle esittäminen.

- *Tekniset järjestelmät* koostuvat teknisten laitteiden, kuten teollisuusprosessien käsittelystä ja hallinnasta. Olennaista näille usein tosiaikaisille järjestelmille on erilaisten laitteiden liittymien ja epästandardien ohjelmistojen käsittely.

- *Sulautetut tosiaikaiset järjestelmät* toimivat nimensä mukaisesti sulautettuna muihin laitteistoihin, kuten matkapuhelimiin tai autoihin. Nämä usein ilman näyttölaitetta olevat järjestelmät vaativat matalan tason ohjelmointia sekä tosiaikaista tapahtumien käsittelyä.

- UML:ää voidaan käyttää myös *hajautettujen järjestelmien* mallinnuksessa. Ne ovat useisiin laitteisiin jaettuja järjestelmiä, joiden välillä vaihdetaan tietoa. Hajautetut järjestelmät ovat yleensä oliojärjestelmien, kuten CORBA:n, varaan rakennettuja.

- *Varusohjelmistot* määrittävät tekniset rakennelmat, joita muut ohjelmistot käyttävät. Esimerkkeinä varusohjelmistoista toimivat käyttöjärjestelmät ja tietokannanhallintajärjestelmät.

- Malliperustaisen arkkitehtuurin kannalta erityisen hyödyllistä on *liiketoimintajärjestelmän* kuvaamisen mahdollisuus. Kuten aiemmin todettiin kohdassa 2.1.1, liiketoimintamallit kuvaavat liiketoiminnan tavoitteet, resurssit, säännöt ja toiminnan.

UML-arkkitehtuuri koostuu neljästä kerroksesta: meta-metamallista, metamallista, mallista ja ilmentymistä (Rumbaugh & al., 1999). MOF:in yhteydessä, taulukossa 2, on selvitetty näiden neljän tason tehtävät sekä annettu jokaisesta selventävä esimerkki.

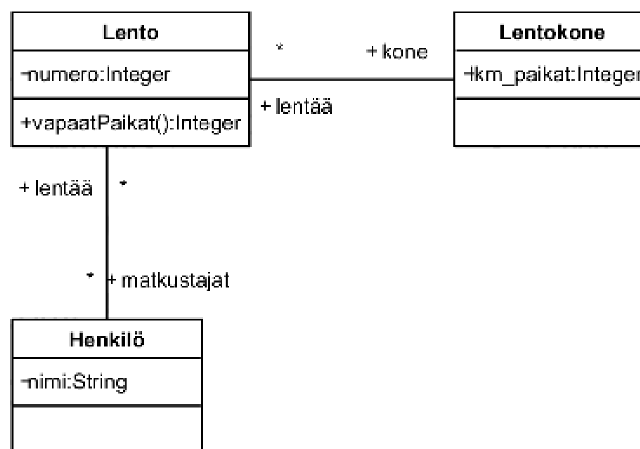
UML-dokumentaation (OMG, 2003b) mukaan on olemassa yhteensä kaksitoista erilaista kaaviota, joilla voidaan kuvata järjestelmän staattista rakennetta, dynaamista käyttäytymistä tai hallinnoida sovelluksen moduuleita. Staattisen rakenteen kuvaamiseksi on kehitetty seuraavat kaaviot: luokkakaavio (class diagram), oliokaavio (object diagram), komponenttikaavio (component diagram) ja sijoittelukaavio (deployment diagram). Järjestelmän dynaamista käyttäytymistä kuvaavat kaaviot ovat: käyttö-

tapauskaavio (use-case diagram), sekvenssikaavio (sequence diagram), yhteistyökaavio (collaboration diagram), tilakaavio (statechart diagram) ja toimintokaavio (activity diagram). Sovelluksen moduulien organisoimiseksi ja hallinnoimisen helpottamiseksi on kehitetty paketit (packages), alijärjestelmät (subsystems) ja mallit (models).

Kaikkea monimutkaisen järjestelmän sisältämää informaatiota ei voida ilmaista pelkästään yhdellä edellä luetelluista kaavioista johtuen järjestelmän erilaisista piirteistä (Eriksson & Penker, 2000). Toiminnalliset piirteet sisältävät järjestelmän pysyvän rakenteen ja muuttuvat vuorovaikutukset. Laadullisia piirteitä taas ovat esimerkiksi rinnakkaisuus, luotettavuus ja asennus. Muun muassa näistä piirteistä johtuen järjestelmän kuvaamiseen tarvitaan useita Rumbaughin & al. (1999) luettelemia *näkymiä* (view), joista jokainen edustaa tiettyä piirrettä. Jokainen näkymä muodostuu useista kaavioista, jotka kuvaavat järjestelmän piirteeseen liittyviä tietoja. Kaaviot voivat kuulua myös useampaan näkymään, eli kaavioiden rajaus näkymien perusteella ei ole tarkka:

- *Käyttötapausnäkymä* (use-case view) kuvaa järjestelmän toimintaa ulkopuolisen toimijan, kuten ihmisen tai toisen järjestelmän, kannalta. Näkymä määritellään käyttötapauskaavioilla ja joskus myös toimintokaavioilla.
- *Looginen näkymä* (logical view) ilmaisee järjestelmän sisäisen toiminnallisuuden rakenteiden ja muuttuvien toimintojen kannalta, eli käytännössä kuinka järjestelmän toiminnot toteutetaan. Pysyvä rakenne kuvataan luokka- ja oliokaavioilla, dynaaminen mallinnus taas suoritetaan tila-, sekvenssi-, yhteistyö- ja toimintokaavioilla.
- *Komponenttinäkymällä* (component view) muodostetaan kuva lähdekoodimoduulien rakenteesta. Se kuvaa järjestelmän kehittäjille moduulien toteutuksen ja niiden väliset riippuvuudet.
- *Samanaikaisuusnäkymä* (concurrency view) havainnollistaa järjestelmän yhteistyötoiminnan sekä synkronoinnin muun muassa prosessien ja säikeiden osalta. Samanaikaisuusnäkymässä järjestelmän kehittäjät ja kokoajat käyttävät dynaamisia kaavioita (tila-, sekvenssi, yhteistyö- ja toimintokaaviot) sekä toteutuskaavioita (komponentti- ja sijoittelukaaviot).
- *Käyttöönottonäkymä* (deployment view) kuvaa järjestelmän fyysisen sijoittelun eli tietokoneet ja oheislaitteet sekä niiden väliset yhteydet.

UML tarjoaa yleisen ja käyttökelpoisen visuaalisen notaation sovellusten mallinnukseen, mutta siinä on myös joitakin ongelmia malliperustaisen arkkitehtuurin kannalta. Thomasin (2004) mielestä OMG ei ole panostanut UML:n kohdalla tarpeeksi erilaisten apuvälineiden yhteistoiminnan parantamiseen, mikä taas omalta osaltaan voi johtaa MDA:n edellyttämien CASE-välineiden hitaaseen kehittymiseen. UML:n artefaktien syntaksinen rakenne ei myöskään ole tarpeeksi selkeä eikä mallien semanttinen rakenne tarkka. Transformaation ja koodingeneroinnin kannalta mallien rakenteen tulisi olla tarkka, jotta toiminnan tulos olisi mahdollisimman laadukas. Kobryn (2002) lisää UML 1.x:n[6] puutteiksi myös sen laajuuden, rajoitetun muokattavuuden sekä riittämättömän tuen komponenttiperustaiselle ohjelmistokehitykselle. Näiden puutteiden vuoksi järjestelmien kehittäjistä voi tuntua liian hankalalta siirtyä käyttämään malliperustaista arkkitehtuuria. OMG pyrkii kuitenkin ratkaisemaan syntyneet ongelmat uudessa UML:n versiossa.



Kuva 12: Osa kuvitteellisesta lentojenvarausjärjestelmästä esitettynä luokkakaavion avulla (Warmer & Kleppe, 2003).

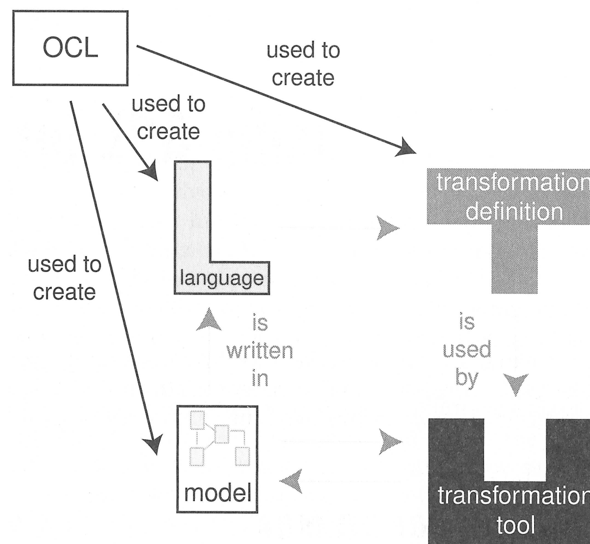
UML sisältää *Object Constraint Language* -rajoitekielen (OCL), jonka avulla voidaan määritellä mallinnettavia järjestelmiä koskevia rajoitteita. Tällä hetkellä uusin versio on viimeistelyä vailla oleva OCL 2.0, joka on tarkoitettu käytettäväksi UML 2.0:n ja MOF 2.0:n yhteydessä (OMG, 2003d). Rajoitteita tarvitaan, koska kaaviot eivät pysty ilmaisemaan kaikkea tarvittavaa informaatiota. Kun rajoitteet on kirjoitettu tarkalla matematiikkaan perustuvalla kielellä, kuten OCL:llä, voidaan esimerkiksi automaattisilla työkaluilla tarkastaa mallien toimivuus. OCL tukee myös kohdassa 2.3.4 esiteltävää XMI:tä, joten CASE-välineet voivat vaihtaa OCL-perustaista tietoa keskenään.

[6]Merkintä 1.x tarkoittaa kaikkia versiota 2.0 edeltäviä UML:n määrityksiä.

Erityisesti nämä ominaisuudet ovat eduksi malliperustaiselle arkkitehtuurille, koska mallien määritysten tarkentuessa myös koodingenerointi tehostuu. Warmer & Kleppe (2003) toteavat, että pelkkien rajoitteiden itsenäinen käyttäminen vaikeuttaisi mallien ymmärtämistä, koska silloin ne viittaisivat olemattomiin mallin osiin. Pelkkä UML:n käyttö taas tekee malleista epätarkkoja. Yhdessä OCL:n kanssa käytettynä UML mahdollistaa monipuolisen ja tarkan mallinnuksen.

Kuvan 12 yksinkertaisessa esimerkissä on mallinnettu luokkakaaviolla kuinka esimerkiksi lentojenvarausjärjestelmässä lentokoneiden, lentojen ja henkilöiden suhteet toisiinsa esitettäisiin. Kyseisessä järjestelmässä olisi luonnollisesti rajoite, jonka mukaan tietylle lennolle ei voitaisi ottaa enempää matkustajia kuin mitä lentokoneessa on istumapaikkoja. Tämä rajoite esitettäisiin OCL-kielellä seuraavasti:

```
context Lento
inv: matkustajat->size() <= kone.lkm_paikat
```



Kuva 13: OCL:n merkitys malliperustaisessa arkkitehtuurissa (Warmer & Kleppe, 2003).

Malliperustaisessa arkkitehtuurissa OCL näyttelee suhteellisen merkittävää roolia, kuten kuva 13 osoittaa. Transformaatiomääritysten luomisessa täytyy käyttää formaalia ja tarkkaa kieltä, jotta automaattiset työkalut ymmärtäisivät niitä mahdollisimman hyvin. Myös mallinnuskielten määrittelyssä vaaditaan käytettävältä kieleltä edellä mainittuja

ominaisuuksia, koska mallinnuskielten täytyy toimia MDA-kehyksen sisällä. Itse mallien täytyy olla tarkkoja saavuttaakseen vaadittavan kypsyystason, joten myös niiden luomisessa tarvitaan OCL:n kaltaista rajoitekieltä.

2.3.3 Common Warehouse Metamodel

Tietovarastointi (data warehousing) tarjoaa CWM-dokumentaation (OMG, 2003a) mukaan erinomaisen mahdollisuuden muuntaa data luotettavaan ja käyttökelpoiseen muotoon, jotta se tukisi mahdollisimman hyvin liiketoiminnan päätöksentekoa. Näiden tietojen hallinnassa ja analysoinnissa käytettävien työkalujen määrän räjähdysmäinen kasvu on tuonut mukanaan ongelman, metadatan esitystavan sekä käsittelyn vaihtelevuuden. Koska jokainen työkalu vaatii käytännössä toimiakseen erilaista metadataa sekä erilaisia metamalleja, ei ole mahdollista käyttää täsmälleen yhtä kuvauskantaa, joka toteuttaisi yhden metamallin kaikelle yrityksen metadatalle. Tähän ongelmaan ratkaisuna on OMG:n vuonna 2000 määrittelemä CWM[7], jonka tärkeimpänä tehtävänä on metadatan ja liiketoimintainformaation helppo siirtäminen erilaisten tietovarastointityökalujen, tietovarastointialustojen sekä tietovarastointikuvauskantojen välillä heterogeenisissä hajautetuissa ympäristöissä.

CWM on Millerin & Mukerjin (2001) mukaan MOF:in ohella käytännön esimerkki malliperustaisen arkkitehtuurin käytöstä. Se kattaa koko tietovarastointisovellusten suunnittelun, rakentamisen ja hallinnoimisen elinkaaren. CWM perustuu kolmeen OMG:n mallinnusstandardiin: UML:ään, MOF:iin ja XMI:hin, jotka näin ollen yhdessä muodostavat metadatan varastointiarkkitehtuurin. UML:n avulla luodaan CWM-metamallit, MOF tarjoaa ohjelmointirajapinnat metadatan muokkaamiseksi ja XMI:n rooli on toimia standardina metadatan siirrossa eri työkalujen välillä (Chang, 2001).

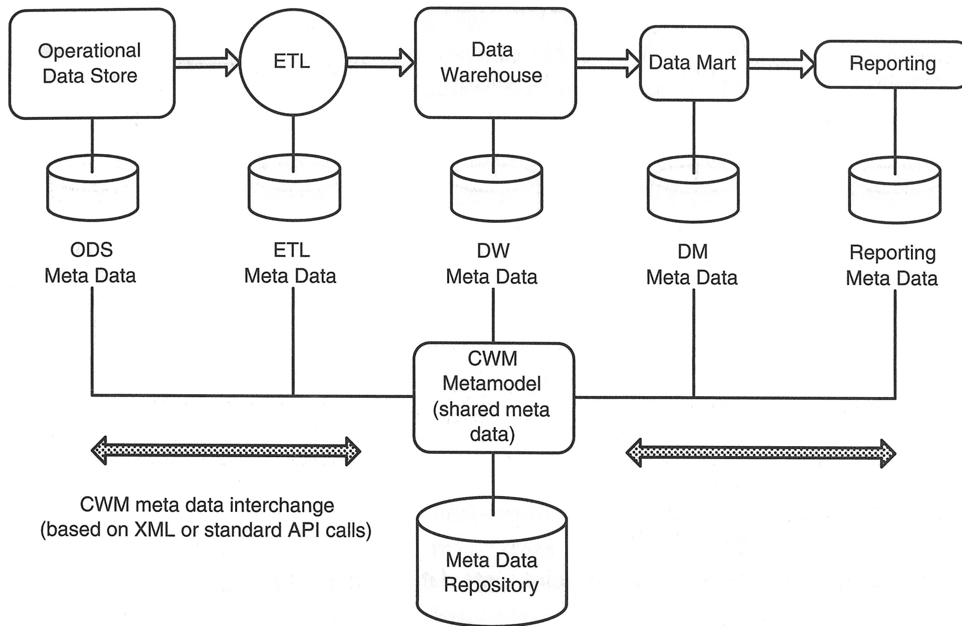
CWM-metamalli sisältää monia yhtäläisyyksiä UML-metamallin kanssa, mutta siitä on myös poistettu joitakin tarpeettomia osia, kuten käyttäytymistä kuvaavat kaaviot. Toisaalta siihen on lisätty erityisiä metaluokkia, joita tarvitaan esimerkiksi relaatiotietokantojen mallintamisessa (Kleppe & al., 2003). CWM-metamalli koostuu kuvan 14 mukaisesti paketeista, jotka on ryhmitelty viiteen tasoon roolien perusteella. Medinan & Trujillon (2002) mukaan yksi CWM:n perusperiaatteista on se, että samalla tasolla olevat paketit ovat riippuvaisia vain alemman tason paketeista. Tällä pyritään välttä-

[7] Tämän tutkielman tarkastelu perustuu CWM:n versioon 1.1. Seuraava työn alla oleva versio on 1.2.

Kuva 14: CWM-metamalli (OMG, 2003a).

määän turhaa saman tai ylemmän tason pakettien kytkentää toisiinsa. *Tietoresurssit* (data resources) sisältävät metamalleja, jotka esittävät tietoresurssien rakenteen. Tietoa voi olla esimerkiksi relaatio- ja XML-muodossa. *Tiedon analysointi* (data analysis) sisältää metamallit muun muassa tiedonlouhinnalle (data mining) sekä transformaatiolle. *Varastonhallinta* (warehouse management) tarjoaa päivittäiset operaatiot varaston prosessien sekä tietovirtojen hallinnoimiseksi. *Perusta* (foundation) tarjoaa CWM-kohtaisia palveluja muille metamallissa ylemmillä tasoilla sijaitseville paketeille. Se pitää sisällään esimerkiksi liiketoimintainformaation sekä tietotyypit. *Oliomalli* (object model) on UML:n osajoukko ja se tarjoaa perusrakenteet CWM-mallien luomiseksi.

Kuvassa 15 Poole & al. (2002) havainnollistavat esimerkin avulla metadata-arkkitehtuuria ja sen toimintaa. Ylimpänä kuvassa esitetään sovellukset (tietovaraston lataus operatiivisista tietokannoista sekä paikallisten tietovarastojen muodostaminen ja raportointi niiden perusteella), joiden käyttämä metadata perustuu jokaisen omiin metamalleihin. Ne voivat käyttää toisten sovellusten kanssa kuvan alalaidassa sijaitsevassa yhteisessä kuvauskannassa olevaa jaettua metadataa siten, että CWM toimii eräänlaisena adapterina sovellusten ja kuvauskannan välillä. CWM suorittaa metadatan muunnoksen yhteisen kuvauskannan sekä sovellusten omien mallien mukaiseksi. Metadatan siirtäminen suoritetaan XMI-muodossa. Koska CWM-metamallit mallinnetaan käyttäen MOF:ia, niitä voidaan käyttää MDA-muunnosten lähteinä tai kohteina.



Kuva 15: Metadata-arkkitehtuuri (Poole & al., 2002).

2.3.4 XML Metadata Interchange

XMI on standardi, joka mahdollistaa olioiden kuvailemisen WWW:ssä (World Wide Web) yleisesti käytetyn XML:n (Extensible Markup Language) avulla. Sen tarkoitus on helpottaa tiedon vaihtoa UML-metamallia sekä MOF-perustaisia kuvauskantoja tukevien mallinnustyökalujen välillä (Pender, 2003). XMI liittyy siis läheisesti erilaisiin mallinnusstandardeihin sekä malliperustaiseen arkkitehtuuriin, minkä vuoksi se esitellään tässäkin tutkielmassa. XMI:n toiminta perustuu siihen, että sen avulla voidaan tehdä muunnos kohdassa 2.3.1 esitellyn MOF:in ja XML:n välillä (Frankel, 2002).

Tutkielman kirjoitushetkellä uusin standardi XMI:stä on versio 2.0, joka uutena asiana edellisiin versioihin määrittelee muun muassa kuinka *XML-skeemat* (schema) muodostetaan malleista (OMG, 2003e). Ne toimivat edellisten XMI-versioiden määrittelemien *XML-dokumenttimääritysten* (XML Document Type Definition) seuraajina XML-dokumenttien sisällön kuvaamisessa. Skeemojen ja XML-dokumenttimääritysten avulla voidaan tarkastaa esimerkiksi dokumenttien oikeellisuus, mutta standardi ei velvoita käyttämään skeemoja tai dokumenttimäärityksiä, vaikka ne olisivatkin olemassa. XMI-standardin käyttö tarjoaa monia mahdollisuuksia, joista Grose & al. (2002) tiivistävät joitakin pääkohtia:

- tarjoaa standardiesitystavan olioille XML-muodossa, jolloin voidaan toteuttaa tehokasta olioiden siirtoa esimerkiksi muuten yhteensopimattomien sovellusten välillä
- määrittelee kuinka XML-skeemoja muodostetaan malleista
- mahdollistaa yksinkertaisten XML-dokumenttien luomisen siten, että niiden muokattavuus säilyy sovellusten kehittyessä

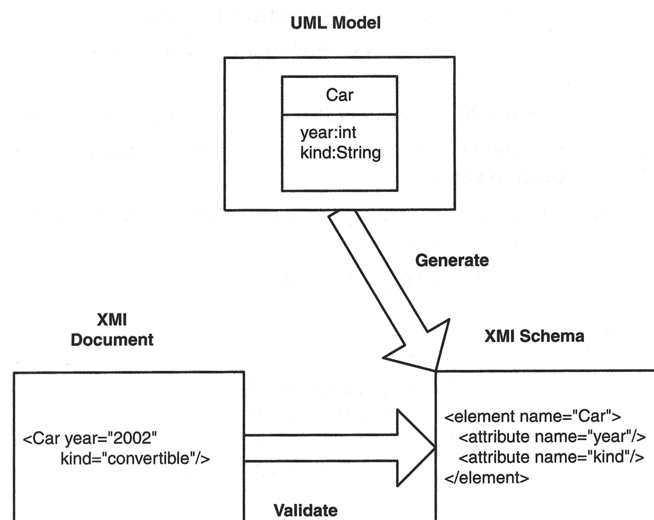
Koska moni nykyajan ohjelmoinnissa käytetty ohjelmointikieli on olioperustainen, vaaditaan muun muassa hajautettujen sovellusten yhteydessä olioiden siirtoa, joka toteutetaan XML-muotoisten dokumenttien avulla. XML:n suosio perustuu Grosen & al. (2002) mukaan siihen, että sillä voidaan tallentaa tietoa niin, että kaikki XML:ää tukevat sovellukset riippumatta ohjelmointikielestä tai sovellusalustasta pystyvät tulkitsemaan oikein tiedostoihin tallennetun datan. XML ei ole kuitenkaan olioperustainen, vaan se kuvaa pelkästään elementit sekä attribuutit, ei olioita. Skeemat voivat sisältää määriteltyjä tyyppejä, mutta niistäkin puuttuvat tärkeät olioihin liittyvät käsitteet kuten esimerkiksi moniperintä. Tämän lisäksi tarvittaisiin vielä skeeman ominaisuuksien ja olioiden välisiä muunnoksia. XMI:llä voidaan ratkaista tämä olioiden kuvaamiseen liittyvä ongelma, koska se tarjoaa välineet UML:llä luotujen olioiden muuntamiseksi XML-formaattiin.

Malliperustaiseen arkkitehtuuriin XMI liittyy siten, että UML:llä luodut sovellusalustariippumattomat mallit voidaan muuntaa CASE-välineen toimintojen kautta XML-dokumenteiksi, jonka jälkeen ne voidaan siirtää vaikkapa toiseen CASE-välineeseen tai koodingeneroijasovellukseen. Tämä edellyttää luonnollisesti käytettävien välineiden XMI-tukea, joka onkin toteutettu suurimpaan osaan nykyisistä välineistä. XMI helpottaa mallien muuntamista XML-dokumenteiksi, koska XML mahdollistaa saman tiedon esittämisen useassa eri muodossa, jolloin erilaiset työkalut voisivat periaatteessa muodostaa rakenteeltaan erilaisia dokumentteja (Grose & al., 2002). Näin ollen XML-dokumentit eivät olisi siirrettävissä sovelluksesta toiseen. Dokumenttien siirtämisessä välineiden välillä täytyy myös ottaa huomioon, että XMI:stä on olemassa eri versioita. Molempien osapuolten on siis tuettava käytettävää versiota, mikä ei kuitenkaan yleensä ole ongelma.

Tiedon vaihtoon liittyvänä ongelmana voidaan pitää CASE-välineiden UML-määrityksen vajavaista tukea (Pender, 2003). Suurin osa UML-mallinnusvälineistä ei

tue täysin UML-spesifikaatiota, johtuen sen laajuudesta sekä jatkuvasta kehittymisestä. Tämän vuoksi monimutkaisten mallien vaihtaminen voi olla hankalaa tai jopa mahdotonta. Frankel (2002) lisää, että UML-metamallin versio 1.x ei tue UML:n graafista notaatiota. Tämä tarkoittaa sitä, että ei ole olemassa sellaisia käsitteitä kuten laatikko, viiva tai koordinaatti. Näin ollen eri sovellusten välillä ei voida vielä vaihtaa graafista tietoa. Kuitenkaan esimerkiksi koodingeneroijat eivät tarvitse graafista tietoa pystyäkseen suoriutumaan tehtävästään. Tähän ongelmaan on olemassa ratkaisu nimeltään *UML Diagram Interchange*, jonka avulla voidaan säilyttää mallinnettujen kaavioiden ulkoasu tiedonsiirron jälkeen (OMG, 2003c).

Kuvassa 16 havainnollistetaan UML:llä luodun mallin, XMI-dokumentin sekä XMI-skeeman suhdetta toisiinsa. UML-muotoisesta sovellusalustariippumattomasta mallista voidaan generoida automaattisesti XMI-skeema, jota vasten XMI-dokumentteja voidaan tarvittaessa validoida. Samaa periaatetta voidaan soveltaa myös aikaisempien standardien määrittelemiin XML-dokumenttimäärityksiin.



Kuva 16: UML-mallin, XMI-dokumentin sekä XMI-skeeman suhde toisiinsa (Grose & al., 2002).

Penderin (2003) mielestä XMI:llä on hyvät mahdollisuudet menestyä tiedon vaihdon välineenä, koska UML:stä on tulemassa hyvää vauhtia systeemien mallinnuksen de facto -standardi. Tämä mahdollistaa sen, että on olemassa yksi yleisesti käytössä oleva ja laajalti hyväksytty tapa vaihtaa tietoa esimerkiksi CASE-välineiden välillä. UML perustuu tarkasti määriteltyyn metamalliin, jota voidaan tarvittaessa laajentaa. XMI:n etu on se, että se pystyy myös ottamaan huomioon nämä muutokset, mikä mahdol-

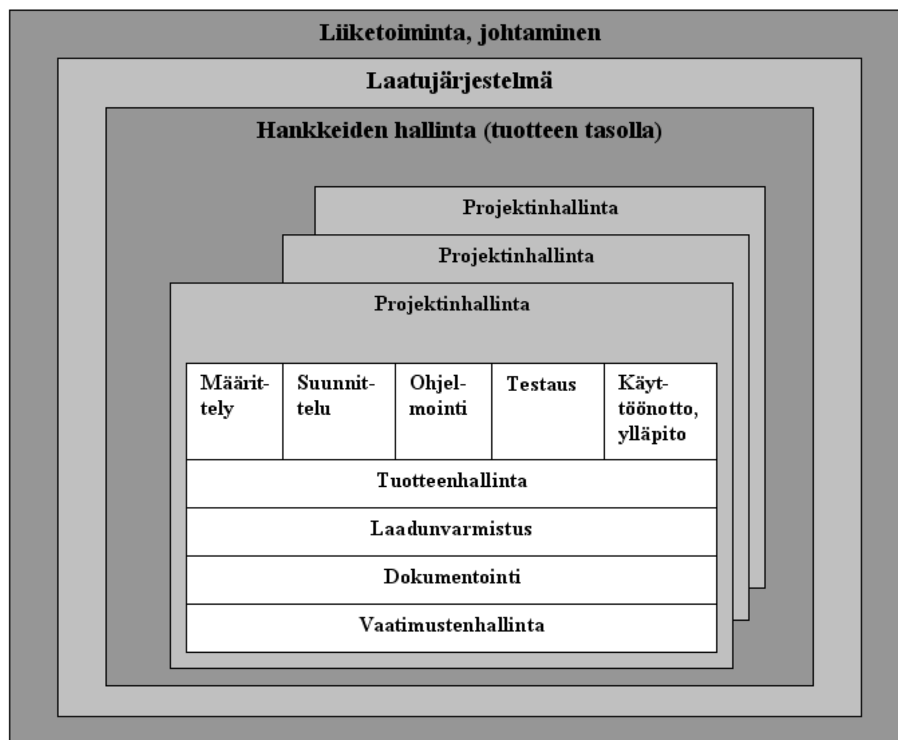
listaa uusien ominaisuuksien ja määritysten lisäämisen työkaluihin ilman, että tiedon siirtäminen toiseen työkaluun vaarantuu.

3 Ohjelmistotuotannon kehitysprosessit

Tässä luvussa tutustutaan perinteiseen ohjelmistotuotantoprosessiin ja sen ongelmiin sekä käydään läpi malliperustaisen kehitysprosessin vaiheet ja erot perinteiseen menetelmään verrattuna. Malliperustaisen prosessin yhteydessä esitellään myös notkea MDA.

3.1 Perinteiset kehitysprosessit ja niiden ongelmat

Ohjelmistotuotanto terminä tarkoittaa Haikalan & Märijärven (1998) mukaan ohjelmistotyötä, jonka tuloksena syntyvät järjestelmät täyttävät kohtuullisesti käyttäjien toiveet ja odotukset sekä myös valmistuvat laadittujen aikataulujen ja kustannusarvioiden puitteissa. Ohjelmistotuotanto voidaan jakaa kuvassa 17 Haikalaa & Märijärveä (1998) mukaillen tuotantoprosessiin liittyviin osa-alueisiin, joita ovat esimerkiksi laatujärjestelmä, projektinhallinta, dokumentointi, tuotteenhallinta, laadunvarmistus, määrittely, suunnittelu, testaus, käyttöönotto ja ylläpito.



Kuva 17: Ohjelmistotuotannon osa-alueet.

Ennen kuin perehdytään ohjelmistotuotantoprosessin eri vaiheisiin sekä joihinkin *elin-*

kaarimalleihin, on syytä esitellä SEI:n (Software Engineering Institute) määrittelemät ohjelmistotyön eri *kypsyystasot*, joiden avulla voidaan tiedostaa kehitystä vaativat osa-alueet sekä todeta ohjelmistotyön laadukkuus (Ferguson & al., 1996). Pressman & Ince (2000) kuvailevat kypsyystasoja seuraavasti:

1. *Kaoottisella* (initial) tasolla olevassa ohjelmistotyössä ei juurikaan hyödynnetä standardeja eikä muita yhdenmukaisia määrityksiä, vaan prosessin lopputulos on täysin riippuvainen yksilöiden panoksesta.

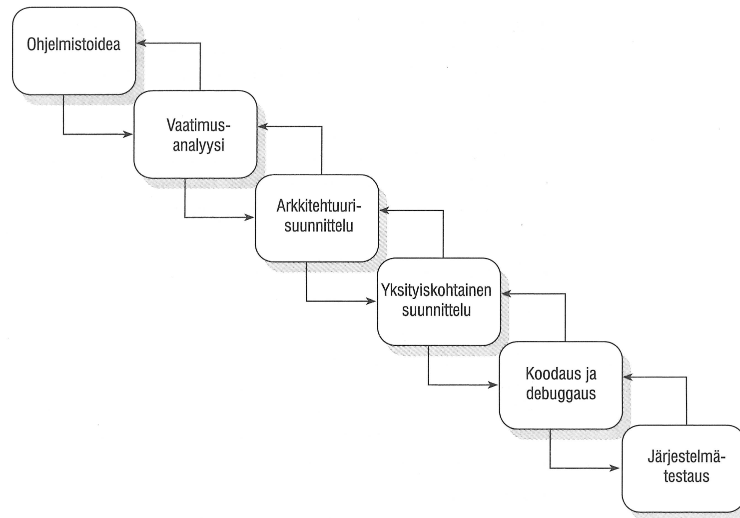
2. *Toistuvalla* (repeatable) tasolla projektinhallinnan perusosien osalta vallitsee yhteisymmärrys muun muassa kustannusten, aikataulun sekä toiminnallisuuden tarkkailemiseksi, jolloin samankaltaiset projektit voidaan viedä hallitusti loppuun.

3. *Määritelty* (defined) taso takaa sen, että ohjelmistotyö on sekä hallinnon että toteutuksen osalta dokumentoitu, standardisoitu ja integroitu organisaation muihin ohjelmistokehitysprosesseihin. Käytettävät menetelmät ovat selkeästi määriteltyjä, hyväksyttyjä ja ylläpidettyjä. Tämä taso sisältää myös edellisen tason vaatimukset.

4. *Hallitulla* (managed) tasolla olevassa projektissa kerätään yksityiskohtaisia tietoja ohjelmistotyön etenemisestä sekä laadusta erilaisten mittareiden avulla. Muut vaatimukset ovat samat kuin edellisellä tasolla.

5. *Optimoivalla* (optimizing) tasolla oleva projekti hyödyntää käytettyjen mittareiden antamaa kvantitatiivista tietoa ohjelmistotyön kehittämiseksi.

Riippuen hieman käytettävästä elinkaarimallista, ohjelmistotuotantoprosessista voidaan tavallisesti erottaa seuraavat vaiheet: määrittely, suunnittelu, ohjelmointi, testaus, käyttöönotto ja ylläpito (Haikala & Märijärvi, 1998). *Vesiputousmalli* on kaikkein tunnetuin ja perinteisin ohjelmistotuotannossa käytetty elinkaarimalli. Vesiputousmallia on käytetty pohjana muille, hieman kehittyneimmille elinkaarimalleille, joilla on korjailtu perinteisen vesiputousmallin sisältämiä ongelmia. McConnell (2002) esittää kuvassa 18 puhtaan vesiputousmallin mukaisen projektin sisältämät vaiheet. Puhtaassa mallissa projektin vaiheet eivät koskaan mene päällekkäin, toisin kuin joissakin kehittyneimmissä versioissa. Projektissa pidetään katselmus jokaisen vaiheen lopussa,



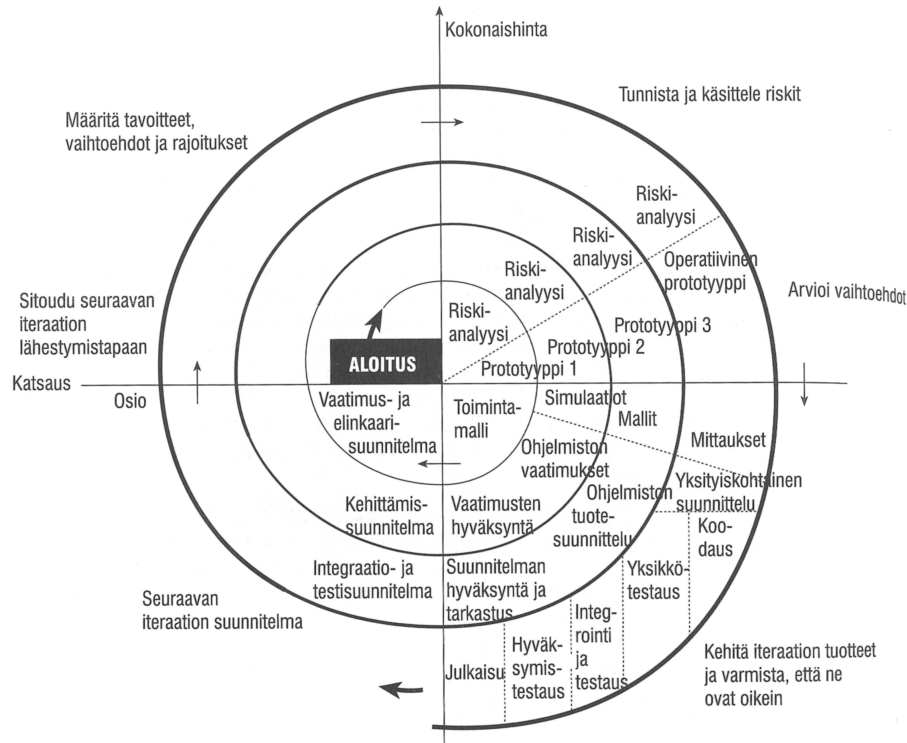
Kuva 18: Vesiputousmalli (McConnell, 2002).

jolloin päätetään jatketaanko mallin mukaisesti seuraavaan vaiheeseen vai pysytäänkö vielä nykyisessä vaiheessa.

McConnell (2002) kuvailee vesiputousmallia dokumenttiohjautuvaksi, eli jokaisen suoritetun vaiheen tuloksena syntyy dokumentti. Tämän vuoksi vesiputousmallin käyttäminen nopeata kehittämistä vaativissa projekteissa ei ole kannattavaa, koska dokumenttien ylläpitäminen voi viedä liikaa aikaa. Projektin eteneminen voi myös vaikuttaa hitaalta, koska varsinaista sovellusta ei ryhdytä rakentamaan ennen kuin projektin loppupuolella. Vesiputousmallia on kutsuttu joustamattomaksi, koska palaaminen edelliseen vaiheeseen on hankalaa — tosin ei mahdotonta. Vesiputousmallin mukaisesti vaatimusmäärittely tehdään heti projektin alussa, minkä vuoksi projektin kohteen täytyy olla teknisesti hyvin määriteltävissä. Hyvin tehtyjen määritelmien avulla vesiputousmalli auttaa löytämään virheet tarpeeksi ajoissa, jolloin niiden korjaaminen on halvempaa kuin myöhemmissä vaiheissa. Vesiputousmallin mukainen toiminta on eduksi etenkin monimutkaisissa projekteissa, joissa laatuvaatimukset ovat tärkeämpiä kuin aikataulu tai budjetti.

Paljon vesiputousmallia kehittyneempi elinkaarimalli on alun perin Boehmin (1988) kehittelemä *spiraalimalli*, joka ottaa huomioon muun muassa ohjelmistotuotantoprosessin sisältämät riskit. Spiraalimallin mukainen ohjelmistoprojekti sisältää monia *miniprojekteja*, joista jokainen keskittyy yhteen tai useampaan riskiin, kunnes kaikki pääriskit on käsitelty (McConnell, 2002). Riski voi tässä tapauksessa tarkoittaa monenlaisia asioita, kuten huonosti ymmärrettyjä vaatimuksia tai mahdollisia suorituskykyon-

gelmia. Pääriskien käsittelyn jälkeen spiraalimalli päättyy samalla tavalla kuin aiemmin esitelty vesiputousmallikin. Kuvassa 19 esitetään spiraalimallin sisältämät vaiheet ja projektin kulku. Spiraalimallin tulkinta aloitetaan ensimmäisestä *iteraatiosta* kuvan keskeltä, jossa tehdään riskianalyysi sekä riskienkäsittelysuunnitelma. Jokainen iteraatio, joka sisältää kuvan ulkoreunoilla vahvennettuina olevat toiminnot, vie projektia kohti suurempaa mittakaavaa spiraalinomaisesti.



Kuva 19: Spiraalimalli (McConnell, 2002).

Spiraalimallin hyviin puoliin kuuluu McConnellin (2002) mielestä etenkin sen joustavuus, jolloin siihen voidaan esimerkiksi yhdistää muita elinkaarimalleja. Kun projektissa olevat riskit on kartoitettu ja saatu sopivalle tasolle, voidaan siirtyä käyttämään jotain muuta riskienhallinnan sivuuttavaa elinkaarimallia. On myös mahdollista sisällyttää jokin elinkaarimalli spiraalimallin iteraatioksi. Yksi tärkeimmistä eduista on Boehmin (1988) mukaan se, että projektin kustannusten kasvaessa riskit pienenevät. Jos jossain vaiheessa huomataan, että havaittu riski on käymässä ylitsepääsemättömäksi, voidaan projekti keskeyttää kustannusten kannalta tarpeeksi aikaisessa vaiheessa. Mainittava haittapuoli spiraalimallissa on sen monimutkaisuus, joka vaatii projektin hallinnolta paljon aikaa sekä asiantuntevuutta koskien esimerkiksi riskien kartoitusta.

Elinkaarimallin valinta riippuu paljon ohjelmistoprojektin luonteesta, jonka perusteel-

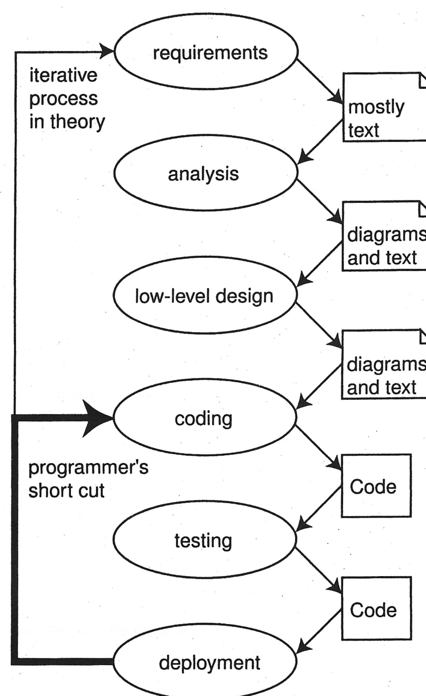
la paras vaihtoehto yleensä valitaan. Kaikkia elinkaarimalleja ei luonnollisesti tässä tutkielmassa voida käsitellä, mutta muutamia tavallisimpia vaihtoehtoja voidaan edellisten lisäksi luetella. Pressman & Ince (2000) esittelevät *prototyyppien* avulla suoritettavan kehittämisen yhtenä vaihtoehtona etenkin silloin, kun ollaan tekemisissä käyttöliittymien kanssa. Prototyyppilähestymistavalla tarkoitetaan sellaista kehittämistä, jossa tuotteen erilaisia piirteitä ja ominaisuuksia kokeillaan ennen varsinaisen sovelluksen rakentamista. *Inkrementaalinen* kehittäminen tarkoittaa Haikalan & Märijärven (1998) mukaan sovelluskehitystä, jossa lopullinen sovellus saadaan aikaan yhden projektin sisällä pienten *inkrementtien* avulla. Inkrementaalinen kehittäminen eroaa prototyyppien käytöstä siinä, että sen jokaisen iteraation lopputuloksena on toimiva sovellus. Jokaisen iteraation lopputulos on siis ominaisuuksiltaan karsittu, mutta silti toimiva versio lopullisesta tuotteesta.

Ohjelmistotuotantoa pidetään paljon aikaa vievänä työnä, jossa pitää koko ajan ottaa huomioon muun muassa uuden teknologian mukanaan tuomat haasteet ja kaiken aikaa muuttuvat ohjelmistoa koskevat määritykset. Kleppe & al. (2003) mainitsevat seuraavat ohjelmistotuotannon ongelmat:

1. tuottavuusongelma (productivity problem)
2. siirrettävyysongelma (portability problem)
3. yhteentoimivuusongelma (interoperability problem)
4. ylläpito- sekä dokumentaatio-ongelma (maintainance and documentation problem)

Perinteisen ohjelmistotuotantoprosessin mukaisesti suurin osa dokumenteista ja kaavioista syntyy ohjelmointia edeltävissä vaiheissa. Kuitenkin ohjelmoinnin alkaessa Kleppen & al. (2003) mukaan näiden dokumenttien sisältämät kaaviot saattavat usein menettää merkityksensä johtuen muun muassa niiden epätarkkuudesta. Väli koodin ja kaavioiden välillä voi kasvaa entisestään, jos sovellukseen tarvitaan tehdä muutoksia. Tämä johtuu siitä, että muutokset tehdään vain koodin tasolla eikä kaavioita näin ollen päivitetä ollenkaan. Kuva 20 havainnollistaa perinteisessä ohjelmistokehitysprosessissa käytetyn ”oikopolun” dokumenttien päivittämisen osalta. Ohjelmoijat eivät päivitä dokumentteja, koska heillä ei tavallisesti ole aikaa tai he kokevat päivittämisen turhaksi esimerkiksi kaavioiden epätarkkuuden vuoksi. Yleensä ongelmana on ajan puute,

koska nykypäivänä korkea tuottavuus on tärkeää, jolloin dokumenttien muokkaaminen ei edistä itse sovelluksen tekemistä. Näistä seikoista syntyy *tuottavuusongelma*, johon yhtenä ratkaisuna kehitettiin *XP-ohjelmointi* (Extreme Programming). Sen pääperiaate pitää vaatimusmäärittelyä sekä ohjelmointia ohjelmistotuotannon tärkeämpinä osina, jolloin ohjelmistotuotantoprosessia on saatu yksinkertaistettua mahdollisimman paljon (Succi & Marchesi, 2001). XP-ohjelmointi ei kuitenkaan ratkaise koko ongelmaa, vain osan siitä. Heti kun sovellusta rakentamassa ollut projektiryhmä hajoaa, tulee muiden henkilöiden suorittamasta ylläpidosta erittäin vaikeaa kunnollisen dokumentaation puuttuessa (Kleppe & al., 2003).

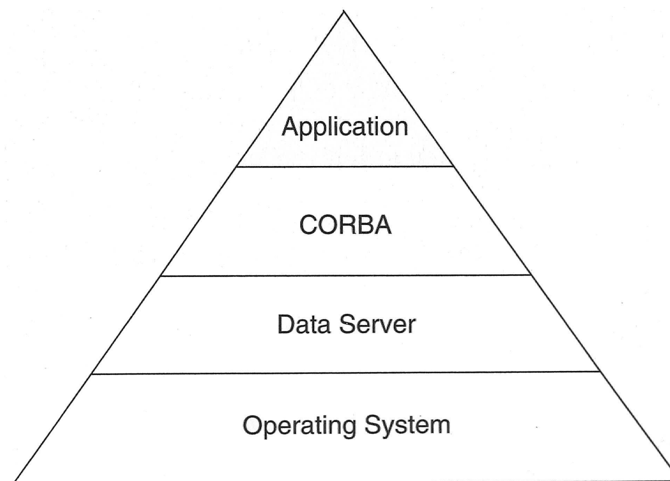


Kuva 20: Perinteinen ohjelmistotuotantoprosessi (Kleppe & al., 2003).

Vuosittain kehittyvät teknologiat pakottavat yritykset sopeutumaan uusien asioiden mukanaan tuomiin muutoksiin. Kleppen & al. (2003) mukaan yritykset eivät voi olla ottamatta käyttöön näitä teknologioita, koska asiakkaat voivat vaatia omiin järjestelmiinsä muutoksia. Uusi teknologia voi myös ratkaista joitakin kriittisiä ongelmia, joten muutos on välttämätön. Ei myöskään kannata unohtaa, että erilaisten työkalujen kehittäjät eivät tue vanhoja versioita loputtomiin. Luonnollisesti on mahdollista, että teknologiat itse muuttuvat, jolloin tuki vanhojen versioiden kanssa ei ole itsestäänselvyys. *Siirrettävyysongelman* vuoksi joudutaan miettimään muunnetaanko sovellus uudelle teknologialle vai uudelle versiolle olemassa olevasta teknologiasta. Kummas-

sakin tapauksessa sovelluksen täytyy pystyä myös kommunikoimaan uudella teknologialla tehtyjen sovellusten kanssa ongelmitta.

Yhteentoimivuusongelma koetaan Kleppen & al. (2003) mukaan, kun sovellusten pitäisi pystyä kommunikoimaan monilla eri teknologioilla rakennettujen järjestelmien ja komponenttien kanssa. Käytännön esimerkkinä tällaisista systeemeistä toimivat monikerroksiset web-palvelut, joissa käytetään tavallisesti lukuisia eri teknologioita, kuten Javaa sekä relaatiotietokantoja. Erilaiset *väliohjelmistot* (middleware) helpottavat heterogeenisten järjestelmien yhteensovittamista. Ne yhdistävät kaksi sovellusta tai erillistä tuotetta välittäen tietoa niiden välillä, mikä ei muutoin olisi mahdollista. Kuvassa 21 Mellor & al. (2004) kuvaavat sovellusten uudelleenkäytön vaikeutta pyramidina, jossa jokainen väliviiva esittää mahdollisuutta käytönaikaista yhteentoimivuutta tukevan standardin käyttöön. Väliohjelmistojen ansiosta jokainen pyramidin osa voidaan periaatteessa vaihtaa muiden osien sitä tietämättä. On kuitenkin mahdollista, että kaikki pyramidin osat eivät toimi kaikkien muiden vaadittujen osien kanssa tai ei haluta käyttää tietyn valmistajan osaa järjestelmässä — ongelma on edelleen olemassa.



Kuva 21: Sovellusten uudelleenkäytön vaikeus (Mellor & al., 2004).

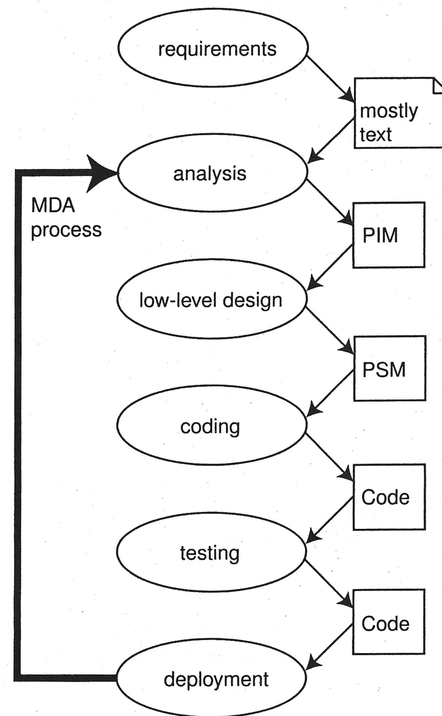
Tuottavuusongelman yhteydessä sivuttiin jo hieman *dokumentaatio-ongelmaa*, joka aiheuttaa merkittäviä vaikeuksia järjestelmien ylläpidossa — *ylläpito-ongelmia*. Kleppe & al. (2003) pitävät dokumentaation tuottamista ohjelmistotuotannon heikkona lenkkinä, joka näin ollen vaikuttaa koko prosessin laatuun. Dokumentaatio tuotetaan tavallisesti ohjelmoinnin jälkeen kiireellä, ja vain siksi, että sen pitää olla olemassa myöhempiä ohjelmistokehitysprosessin vaiheita varten. Dokumenttien huono laatu johtuu juuri tästä pakon tunteesta, jota koetaan sovelluskehittäjien keskuudessa, minkä vuoksi

myös dokumenttien huolellinen tarkastaminen jää usein tekemättä. Luonnollisesti olisi tärkeintä kehittää mahdollisimman helposti muutettavia ja ylläpidettäviä järjestelmiä. Tähän päästään osittain siten, että korvataan matalan tason dokumenttien käsin tekeminen tarvittavien dokumentin osien automaattisella generoinnilla koodin joukosta, jolloin dokumentaatio pysyy aina ajan tasalla. Tätä tekniikkaa tukevat muun muassa Java sekä Eiffel. Esimerkiksi Javalla tehtyjen sovellusten lähdekoodista voidaan muodostaa automaattisesti luokkien sisältöä kuvaavat dokumentit Javadoc-komennon avulla (Sun Microsystems, 2004). Dokumenttien tuottaminen edellyttää vain luokkien ja niiden sisällä olevien tietojen oikeaoppista kommentointia. Ratkaisu dokumentaatio-ongelmaan on vain osittainen, koska automaattinen generointi ei mahdollista elintärkeiden korkean tason dokumenttien ylläpitoa. Seuraavassa kohdassa esiteltävä malliperustainen kehitysprosessi tarjoaa ratkaisuja edellä mainittuihin perinteisen ohjelmistotuotannon ongelmiin.

3.2 Malliperustainen kehitysprosessi

Neljännen sukupolven tekniikat (fourth generation techniques, 4GT) keskittyvät siihen, että ohjelmistokehittäjä voi määritellä järjestelmän piirteitä mahdollisimman korkealla tasolla, minkä jälkeen käytettävä työkalu generoi automaattisesti lähdekoodia tehtyjen määritysten perusteella (Pressman & Ince, 2000). Tämä on myös malliperustaisen kehitysprosessin perusidea. Malliperustainen kehitysprosessi ei nopeasti katsottuna eroa perinteisistä menetelmistä vaiheidensa osalta. Sen suurimpana erona on kuitenkin kehitysprosessin aikana syntyvien artefaktien luonne (Kleppe & al., 2003). Artefaktit ovat formaaleja malleja, jolloin niitä voidaan tulkita suoraan tietokoneen avulla. Kuvasta 22 selviää MDA-kehitysprosessin kulku sekä MDA:n perustan muodostavat kolme mallia: PIM, PSM ja koodi.

Kuten muutkin ohjelmistokehitysprosessit, MDA-kehitysprosessi alkaa vaatimusten keräämisellä. Vaatimukset voidaan määritellä esimerkiksi asiakkaan hyvin ymmärtämällä graafisella notaatiolla, jolloin saadaan luotua järjestelmän liiketoimintaa kuvaava PIM. PIM ei nimensä mukaisesti ota kantaa käytettävään sovellusalustaan. Prosessin seuraavassa vaiheessa PIM muunnetaan transformaation kautta yhdeksi tai useammaksi PSM-malliksi, jotka kuvaavat mallin valitun sovellusalustan mukaisella tavalla. Eri sovellusalustoille tehtyjä PSM-malleja voi olla useita, koska nykyään on tavallista, että järjestelmät koostuvat useista eri teknologioista. Muunnos PIM:n ja PSM:n välillä



Kuva 22: Malliperustainen ohjelmistotuotantoprosessi (Kleppe & al., 2003).

onkin koko malliperustaisen prosessin kannalta monimutkaisin vaihe. Lopuksi PSM-malleista generoidaan lähdekoodi, joka testataan muiden kehitysprosessien tapaan huolellisesti. Nämä kolme vaihetta muodostavat MDA-prosessin perusaskeleet. On tärkeää huomata, että jokainen prosessin transformaatio tehdään automaattisen työkalun avulla — toisin kuin perinteisessä kehityksessä.

Järjestelmien kehitysprosesseissa tarvitaan monien ihmisten työpanosta, joten on luonnollista, että heillä on erilaiset työnkuvat ja roolit. Malliperustaisen kehitysprosessin roolit poikkeavat hieman perinteisten kehitysprosessien vastaavista, joten on syytä esitellä Mellorin & Watsonin (2004) kuvaukset rooleista:

- *Vaatimusten määrittelijät* (Requirements Analysts) keräävät tietoa ja tekevät päätöksiä laadittavan järjestelmän vaatimusten suhteen. Heidän tehtäväänsä kuuluu vaatimusten muuntaminen tietokoneen ymmärtämiksi UML-malleiksi, joita voidaan myöhemmin prosessin edetessä testata ja verifioida. UML-mallien käyttö ehkäisee muodostettujen vaatimusten väärinymmärtämistä.
- *Atk-suunnittelijat* (Analyst/Designers) esittävät vaatimukset liiketoimintalogiikan osalta formaalilla tavalla käyttäen apuvälineenä esimerkiksi luokka- ja ti-

lakaavioita. Eniten aikaa käytetään PIM-mallien riittävän tarkkuuden saavuttamiseksi, koska niitä käytetään myöhemmin kehitysprosessin mukaisesti PSM-mallien ja koodin muodostamiseen.

- *Järjestelmäarkkitehdit* (Architects) tekevät päätökset koskien järjestelmän fyysistä rakennetta. MDA-prosessissa tämä tarkoittaa muun muassa mallien välisten kuvausten valintaa sekä järjestelmän suorituskyvyn parantamiseksi vaadittavaa hienosäätöä.

- *Ohjelmoijien* (Analyst/Programmers) työnkuva on sama kuin tavallisestikin joitakin poikkeuksia lukuun ottamatta. Näin ollen heidän tehtävänsä on muuntotyökalun koodingenerointiin käyttämien kaavojen luominen sekä hienosäätö. Tämä mahdollistaa ohjelmoijien sovellusalustakohtaisen tietämyksen paremman hyödyntämisen lähdekoodin käsin kirjoittamiseen verrattuna. Vain harvat koodingeneroijat muodostavat 100%:n toimivaa koodia, joten ohjelmoijat voivat myös keskittyä tärkeiden ja heidän kannalta kiinnostavien osien ohjelmoimiseen.

- *Testaajat* (Testers) voivat suorittaa järjestelmän testaamisen tehokkaammin, koska he pystyvät luomaan malleista automaattisia skriptejä, joiden avulla voidaan testata kattavasti sekä käsin ohjelmoidut että automaattisesti generoidut ohjelman osat.

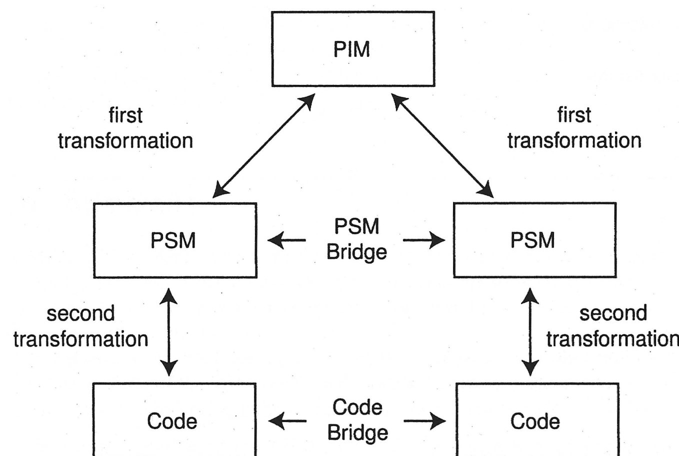
- *Ylläpitäjien* (Maintainers) työ keskittyy mallien sekä *suunnittelusääntöjen* (design rules), ei koodin ylläpitämiseen. Sovelluksen toimintaa koskevan muutoksen tekemiseksi suoritetaan ylläpitotoimet PIM-malliin. Jos taas koko sovellusalusta vaihtuu, korvataan vanhat kuvaukset uusilla.

- *Asiakkaat* (Customers).

Malliperustaisen arkkitehtuurin avulla on pyritty ratkaisemaan kohdassa 3.1 mainittuja perinteisten sovelluskehitysprosessien ongelmia. MDA:n perusperiaatteiden mukaisesti enemmistö sovelluskehittäjistä keskittyy suurimman osan ajastaan PIM-mallien tekemiseen. Kleppe & al. (2003) muistuttavat, että PIM-mallin muuntaminen PSM-malliksi vaatii tarkasti määriteltyjen transformaatiomääritysten kehittämisen, mikä on muutamien sovellusalusta-asiantuntijoiden tehtävä. Tuottavuus parantuu sekä kustannukset pienenevät, koska sovelluskehittäjät voivat keskittyä teknisistä yksityiskohdista riippumattomien PIM-mallien kehittämiseen (Mellor & al., 2004). Kleppe & al. (2003)

korostavat, että sovelluskehittäjillä on vähemmän työtä, koska käytettävään sovellusalustaan ei tarvitse ottaa kantaa, ja näin ollen he voivat käyttää enemmän aikaa liiketoimintaongelmien ratkaisuun. Tämän vuoksi loppukäyttäjien pitäisi saada paremmin suunniteltuja ja toteutettuja ohjelmia.

Keskittyminen sovellusalustariippumattomiin malleihin ratkaisee Kleppen & al. (2003) mielestä myös siirrettävyysongelman. PIM-mallit voidaan muuntaa minkä tahansa käytettävän sovellusalustan mukaisiksi PSM-malleiksi, jos vain kyseisen alustan transformaatiomääritykset on tehty. Transformaatiomäärityksiä ei tarvitse tehdä kuin kerran, jonka jälkeen niitä voidaan käyttää jokaisen alustalle rakennettavan sovelluksen kehityksessä. PIM-mallit toimivat myös järjestelmän korkean tason dokumentaationa. PIM-mallit pysyvät päivitettyinä mahdollisten järjestelmää koskevien muutosten tapahtuessa, koska muutosten jälkeen koodi generoidaan uudelleen päivitetyistä malleista. Näin ollen järjestelmästä on aina olemassa uusimmat korkean tason dokumentit, jotka voivat myös sisältää kaavioihin sisältymättömiä kommentteja esimerkiksi mallinnustyöskentelyn ajalta.



Kuva 23: Siltojen käyttö eri mallien välillä (Kleppe & al., 2003).

Yhteentoimivuusongelma liittyy eri sovellusalustoille kohdennettujen mallien välisiin suhteisiin. PIM-malleista generoiduilla PSM-malleilla voi olla suhteita toisiinsa, jolloin yhteentoimivuusongelma nousee esille. PSM-mallit on muodostettu eri sovellusalustoille sopiviksi, joten ne eivät voi suoraan keskustella toistensa kanssa. MDA tarjoaa ratkaisuna mallien väliset *sillat* (bridges), jotka välittävät tietoa saman tason elementtien kesken (Mellor & al., 2004). Siltojen muodostaminen voidaan Kleppen & al. (2003) mukaan suorittaa, jos tietoa on riittävästi muuntamaan PIM-malli vähintään kahdeksi eri PSM-malliksi. Eri laitealustojen välistä yhteentoimivuutta voidaan hyö-

dyntää, jos käytettävissä oleva työkalu tukee mallien välisten transformaatioiden lisäksi siltojen generointia. Siltojen käyttöä eri laitealustojen välisessä kommunikoinnissa havainnollistetaan kuvassa 23.

Notkeiden menetelmien käyttö on nykyään kasvussa, joten myös MDA:sta löytyy notkea versio. Seuraavassa kohdassa esitellään notkeiden menetelmien ominaisuuksia sekä erityisesti notkea MDA -prosessi.

3.2.1 Notkea MDA

Notkeat menetelmät (agile methods) ovat kevyitä, nopeita ja joustavia ohjelmistokehitysprosesseja, jotka soveltuvat erityisen hyvin nopeasti kehittyvien alojen, kuten Internet- ja mobiilisovellusten kehittämisprosesseiksi. Abrahamssonin & al. (2002) mielestä sovelluskehitysprosessi on notkea, jos kehitys on *inkrementaalista* (toimivia sovelluksia toimitetaan tietyin väliajoin), *yhteistoiminnallista* (asiakas ja järjestelmän kehittäjät työskentelevät yhdessä), *suoraviivaista* (käytettävä menetelmä on helppo omaksua, sitä voidaan muuttaa ja se on hyvin dokumentoitu) ja *mukautuvaa* (viime hetken muutokset eivät ole ongelma). Tehtäessä ohjelmistokehitystyötä notkeilla menetelmillä, taustalla vaikuttavat koko ajan Highsmithin & Cockburnin (2001) esittelemät arvot:

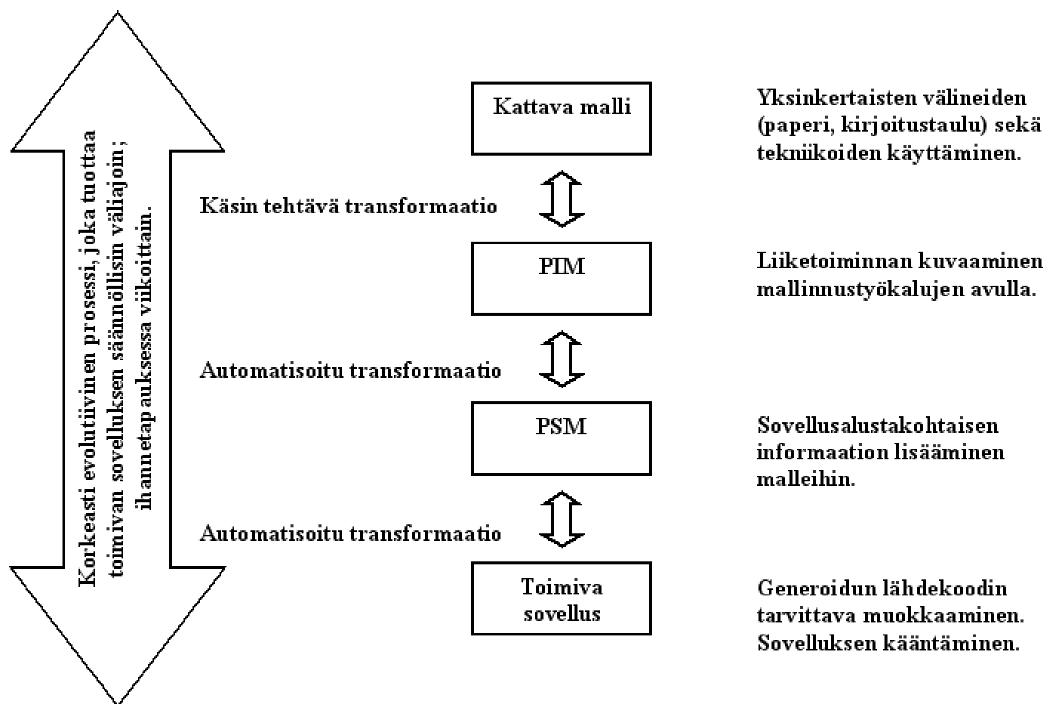
- *Yksilöt ja vuorovaikutus* ovat tärkeämpiä kuin prosessit ja työkalut.
- *Toimiva sovellus* kulkee täydellisen dokumentaation edellä.
- *Yhteistyö asiakkaiden kanssa* koetaan tärkeämpänä kuin sopimusneuvottelut.
- *Haasteisiin tartutaan* ennemmin kuin noudatetaan suunnitelmaa.

Notkea MDA perustuu Mellorin & al. (2004) mielestä faktaan, jonka mukaan lähdekoodi ja suorituskelpoiset mallit ovat käytännön tasolla sama asia. Tämä johtuu siitä, että suorituskelpoisia malleja voidaan perinteisen koodin tavoin rakentaa, ajaa, testata sekä muokata lyhyissä inkrementaalisissa sykleissä. Näin ollen edellä mainitut Agile Alliancen[8] tunnustamat periaatteet pätevät myös suorituskelpoisiin malleihin ja si-

[8] Agile Alliance on voittoa tavoittelematon organisaatio, joka tukee ohjelmistotuotannossa notkeita menetelmiä käyttäviä yrityksiä sekä yksityishenkilöitä saavuttamaan laadullisesti parempia tuloksia ohjelmistoprojekteissaan (Agile Alliance, 2004).

ten notkeaan MDA:han. Jotta Mellorin & al. (2004) esittämää vertausta lähdekoodin ja suorituskelpoisten mallien samankaltaisuudesta voidaan käyttää, täytyy mallien olla riittävän tarkasti rakennettuja.

Mellorin (2004) mukaan kuilu perinteisten dokumenttien ja niiden perusteella tuotetun lähdekoodin välillä (verification gap) syntyy, kun dokumenttien oikeellisuutta sekä toimivuutta ei päästä toteamaan riittävän aikaisin. Tämä voi pahimmassa tapauksessa johtaa siihen, että dokumenteista tulee käyttökelvottomia, jos esimerkiksi ongelmaa koskeva ratkaisutapa muuttuu täydellisesti. Notkeat menetelmät mahdollistavat nopean, pienissä osissa tapahtuvan kehityksen, jolloin muutoksiin malleissa voidaan sopeutua. Pienet toimivat järjestelmän osat voidaan näyttää asiakkaalle, jolloin hän pystyy kertomaan mahdollisista puutteista ja ongelmista ajoissa. Kun kehittämisessä käytetään pelkän lähdekoodin sijasta asiakkaan ymmärtämiä, riittävän tarkasti määriteltyjä suorituskelpoisia malleja, voidaan edellä esitetyn kuilun aiheuttamia ongelmia vähentää merkittävästi. MDA:ssa suorituskelpoisia malleja kutsutaan sovellusalustariippumattomiksi malleiksi.



Kuva 24: Notkea MDA.

Kuvassa 24 Ambleria (2004) mukaillen esitetty notkea MDA-prosessi alkaa liiketoiminnan hahmottelemisella. Tässä vaiheessa käytetään vain yksinkertaisia tekniikoita,

kuten kynää ja paperia, jotta liiketoiminta saadaan esitettyä mahdollisimman selkeästi. Näin ollen prosessin tähän vaiheeseen voivat osallistua aktiivisesti sovelluskehittäjän lisäksi myös kyseisen liiketoiminta-alan ammattilaiset. Vaiheen tuloksena syntyy kattava malli liiketoiminnasta tai sen osa-alueesta. Prosessin seuraavassa vaiheessa edellisen vaiheen malli muunnetaan mallinnustyökalujen avulla tietokoneen ymmärtämäksi sovellusalustariippumattomaksi malliksi. Tämän jälkeen PIM-malli muunnetaan automatisoidun transformaation kautta PSM-malliksi, joka lisää edelliseen malliin sovellusalustakohtaisen informaation. Notkean MDA-prosessin seuraava vaihe on lähdekoodin automaattinen generoiminen PSM-mallista tai suoraan PIM-mallista, riippuen käytettävästä työkalusta. Lähdekoodia voidaan muokata tarpeen mukaan ennen varsinaista sovelluksen kääntämistä. Oleellisinta notkeassa MDA-prosessi on se, että sovellusta voidaan rakentaa lyhyissä inkrementaalisissa sykleissä, jotka tuottavat aina tuloksena toimivan sovelluksen. Tätä työskentelytapaa on käytetty myös luvun 5 esimerkissä.

4 Koodingenerointi

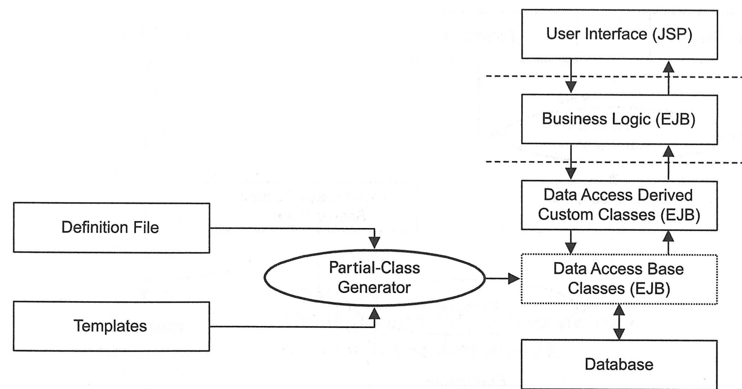
Koodingeneroinnilla tarkoitetaan erilaisten automaattisesti lähdekoodia tuottavien ohjelmien käyttöä apuvälineinä omia sovelluksia rakennettaessa (Herrington, 2004). Apuvälineiden koko voi vaihdella pienistä makroista suurempiin ja samalla paljon monimutkaisempiin ohjelmiin. Yhteistä niillä on kuitenkin se, että ne tuottavat koodia, jonka ohjelmoija joutuisi muuten kirjoittamaan käsin. Automaattista generointia voidaan käyttää hyväksi monia eri tarkoituksia silmällä pitäen. On mahdollista luoda luokkien runkoja, kokonaisia käyttöliittymiä, SQL-lauseita tai vaikka testitapauksia. Esimerkki ajonaikaisesta koodingeneroinnista on Java-virtuaalikoneen mahdollistama mukautuva optimointiteknologia (Sun Microsystems, 1999).

Herrington (2003) listaa muutamia hyötyjä, joita voidaan saavuttaa käyttämällä automaattista koodingenerointia:

- *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.
- *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.
- Järjestelmän kehittämisessä voidaan käyttää kohdassa 2.1 kuvattuja *korkean abstraktiotason* malleja, joiden avulla saavutetaan esimerkiksi siirrettävyys sekä lisätään yhteistyötä liiketoiminnan ammattilaisten sekä sovelluskehittäjien välillä.

Edellä mainittuja hyötyjä voidaan tavoitella monenlaisilla koodingenerointimuodoilla. Seuraavaksi käsitellään Herringtonin (2003) kuvaamat erilaiset tavat tuottaa automaattisesti koodia yksinkertaisimmasta tavasta alkaen. *Koodin muuntaminen* toiseen muotoon (code munging) on yksi sellainen. Siinä annetaan muuntajatyökalulle syötteenä

lähdekoodi, josta se poimii tärkeimmät ominaisuudet ja käyttää niitä uusien tiedostojen muodostamiseksi. Tätä tapaa voidaan käyttää esimerkiksi dokumentaation automaattiseen kokoamiseen lähdekoodin seasta. *Sisäisohjelman laajentaja* (inline-code expander) ottaa taas syötteenä lähdekoodin, josta se muodostaa käännöstä varten *tuotantokoodin* (production code). Tuotantokoodi voi muodostua siten, että sisäisohjelman laajentaja lisää lähdekoodiin tietyllä tavalla merkittyihin kohtiin vaikkapa SQL-lauseita. Tällaista menetelmää tarvitaan, kun sovelluskehittäjät haluavat esimerkiksi pitää sulautetut SQL-lauseet erillään järjestelmän muusta koodista. Yhteiskoodingeneroija (mixed-code generator) toimii samalla periaatteella kuin edellinen menetelmä, mutta sen ensimmäisellä suorituskerralla muodostama koodi toimii seuraavan suorituskerran aikana syötteenä. Tällä tavalla voidaan etsiä esimerkiksi tietynlaisia kommentteja, jotka korvataan joillakin lopulliseen kääntämiseen tarvittavilla koodin osilla.

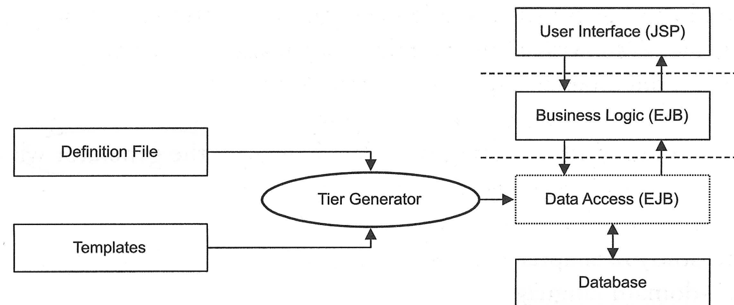


Kuva 25: Vaillinaisten luokkien generointi kolmikerrosarkkitehtuurissa (Herrington, 2003).

Vaillinaisten luokkien generoija (partial-class generator) luo määrittelytiedostosta sekä *kaavaimista* (templates) perusluokkakirjaston, jonka sisältämät luokat käännetään yhdessä ohjelmoijien kehittämien luokkien kanssa. Tämä tekniikka sopii erityisesti kolmikerrosarkkitehtuuriin, jossa vaillinaisten luokkien generoijalla voidaan muodostaa kuvan 25 mukaisesti tietokantayhteyden tarvitsemat perusluokat. Ohjelmoijat voivat tämän jälkeen täydentää kyseisen tason valmiiksi.

Tason generointi (tier generation) eroaa edellä esitetystä vaillinaisten luokkien muodostamisesta siten, että tässä mallissa generoidaan koko tason tarvitsema koodi — toisin kuin edellisessä. Esimerkkinä tason generoinnista toimii *malliperustainen generointi*. Kuva 26 havainnollistaa malliperustaisen generoinnin vaiheet. Ensin generoija saa syötteenä kaavaimet sekä yleensä XML-muotoiset määritykset, minkä jälkeen se

muodostaa määrityksistä kaavainten perusteella lähdekoodin koko tasolle.



Kuva 26: Malliperustainen generointi kolmikerrosarkkitehtuurissa (Herrington, 2003).

Koodingeneroinnilla on Herringtonin (2004) mielestä myös joitakin huonoja puolia, jotka saattavat aiheuttaa ylimääräistä työtä ellei päätöstä koodingeneroijan käytöstä ole tehty riittävän huolellisesti. Valitun koodingeneroijan käyttö ja toiminta on dokumentoitava huolellisesti, jotta sen ylläpito voidaan suorittaa tehokkaasti. Lisättäessä koodingeneroijan käyttö rutinoituun työnkulkuun, pitää ottaa huomioon sen mukanaan tuomat muutokset käytettyihin toimintatapoihin etenkin kehitysprosessin ja ylläpidon osalta. Muun muassa henkilökunnan koulutuksen määrän tarve riippuu pitkälti käytettävän generoijan ja rakennettavan sovelluksen monimutkaisuudesta. Koodingeneroinnin haittapuolia voidaan monesti karsia yksinkertaisesti ymmärtämällä milloin automaattisesti generoinnista on oikeasti hyötyä, ja milloin on taas helpompi ohjelmoida käsin. Luvussa 5 päästään tutustumaan malliperustaisessa arkkitehtuurissa ja koodingeneroinnissa käytettäviin apuvälineisiin sekä esimerkkisovellukseen.

5 MDA käytännössä

Tässä luvussa käsitellään malliperustaisen arkkitehtuurin teorian soveltamista käytännön tasolla. Ensimmäisenä esitellään työkaluja, joita voidaan käyttää hyväksi rakennettaessa MDA-sovellusta. Tämän jälkeen perehdytään esimerkkisovelluksen rakentamisen eri vaiheisiin sekä esitellään rakennettu esimerkki.

5.1 Apuvälineet

Malliperustaiseen arkkitehtuuriin perustuva sovellus vaatii rakennusvaiheessa joitakin työkaluja. Sovellusalustariippumattoman mallin tekemisessä käytetään CASE-välinettä, lähdekoodin generointiin tarvitaan koodingeneroijaa sekä generoidun koodin muokkaaminen toimivaksi on helpompaa käyttäen apuna jotakin *kehitysympäristöä* (Integrated Development Environment, IDE). Seuraavana esitellään varteenotettavat vaihtoehdot näistä välineistä.

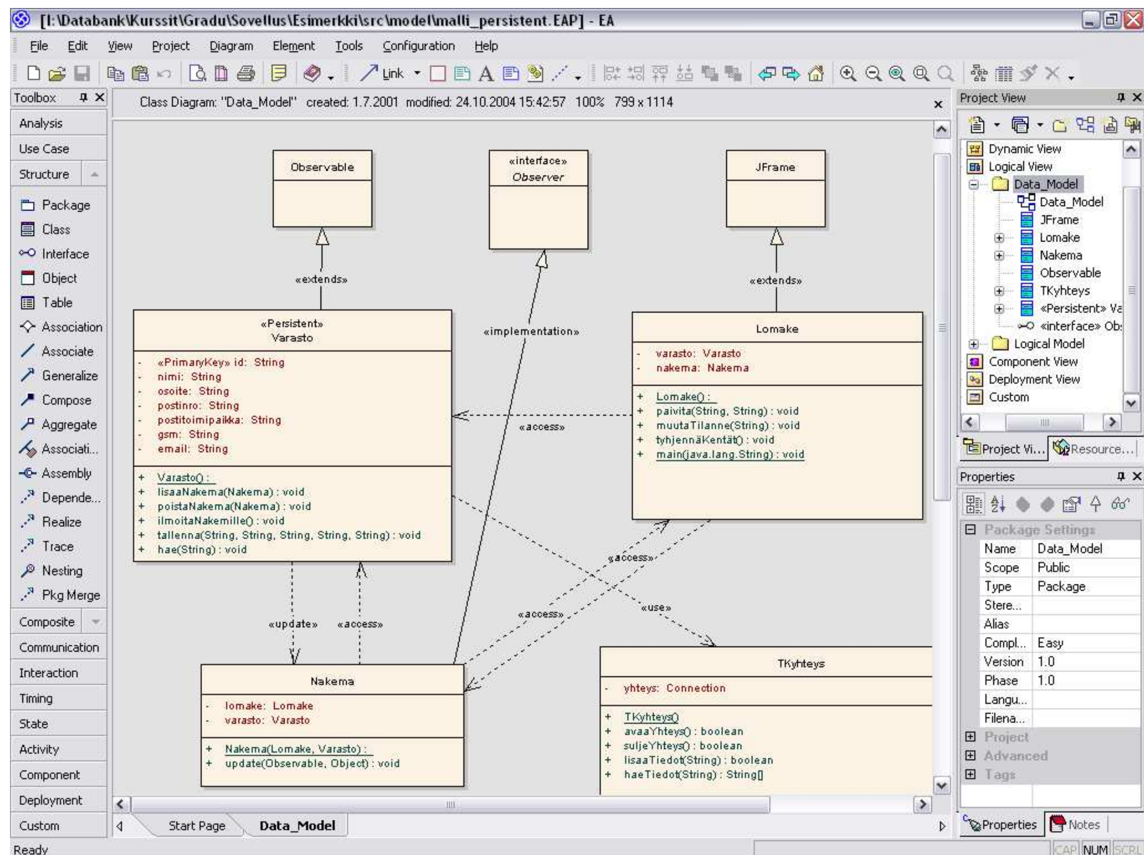
5.1.1 Enterprise Architect

Enterprise Architect[9] (EA) on CASE-väline, joka mahdollistaa sovelluksen koko elinkaaren mallintamisen UML:n avulla (Sparx Systems, 2004). EA:n ominaisuuksiin kuuluvat kaikkien UML-kaaviotyyppien tuki, koodingenerointi luoduista malleista eri ohjelmointikielille (Java, C++, C#, VB.Net, Delphi ja Visual Basic), mallien muodostaminen lähdekoodista eli käänteistekniikka sekä tuki XMI-standardille. Malliperustaisen arkkitehtuurin kannalta tärkeimmät ominaisuudet ovat kattavien UML-mallien muodostaminen sekä tuki mallien muuntamiseksi XMI-formaattiin, mikä mahdollistaa sovellusalustariippumattomien mallien luomisen ja siirtämisen CASE-välineestä koodingeneroijasovellukseen.

Kuvassa 27 EA:ta on käytetty kohdan 5.2 esimerkin PIM-mallin tekemiseen sekä sen muuntamiseen XMI-formaattiin. CASE-välineen avulla mallit saadaan muodostettua sellaisella tarkkuudella, että myöhemmin koodingeneroija pystyy tuottamaan lähdekoodin niiden perusteella. EA mahdollistaa kaavioiden rakenteiden tarkan määrittelemisen lisäksi myös kattavan dokumentoinnin. Esimerkiksi luokkakaavion yhteydessä

[9] Tämän tutkielman esimerkin tekemisessä on käytetty EA:n versiota 4.0.

jokaiseen mallinnettuun luokkaan, luokan attribuutteihin sekä operaatioihin voidaan kirjoittaa kuvaukset niiden merkityksestä ja toiminnasta. Malliperustaisen arkkitehtuurin kannalta EA:n ominaisuudet ovat erittäin kattavat. Ominaisuudet mahdollistavat laajojenkin järjestelmien mallintamisen niin rakenteen kuin toiminnallisuudenkin osalta, jolloin MDA:n hyödyntäminen ohjelmistoprosessissa on mallinnuksen osalta helppoa ja joustavaa.



Kuva 27: PIM-mallin rakentamista EA:n avulla.

5.1.2 iQgen

Koodingeneroijana tämän tutkielman esimerkissä toimii iQgen[10]. Se on täysin Java-pohjainen ohjelma, joka käyttää koodingeneroinnissa hyväkseen Java Server Pages[11]

[10]Tämän tutkielman esimerkissä on käytetty iQgen:in versiota 2.0.

[11]JSP on tekniikka, jonka avulla voidaan luoda muun muassa sovellusalustasta riippumattomia dynaamisia web-sivuja käyttäen hyväksi kaikkia Javan tarjoamia palveluja. JSP mahdollistaa web-sovelluksen sisällön tuottamisen erottamisen sovelluksen käyttöliittymästä, jolloin käyttöliittymän ulkoasua voidaan muokata vapaasti. (Sun Microsystems, 2004)

(JSP) -tekniikalla kirjoitettuja kaavaimia (innoQ, 2004). Kaavaimet vastaavat tutkielman teoreettisessa osassa, kohdassa 2.2 esiteltyjä transformaatiomäärityksiä, joiden perusteella voidaan suorittaa mallien välisiä muunnoksia. Kaavaimet siis kertovat mitä koodingeneroijan tulee tehdä muuntaakseen syötteenä saamansa, CASE-välineellä mallinnettu PIM-malli halutun ohjelmointikielen mukaiseksi lähdekoodiksi.

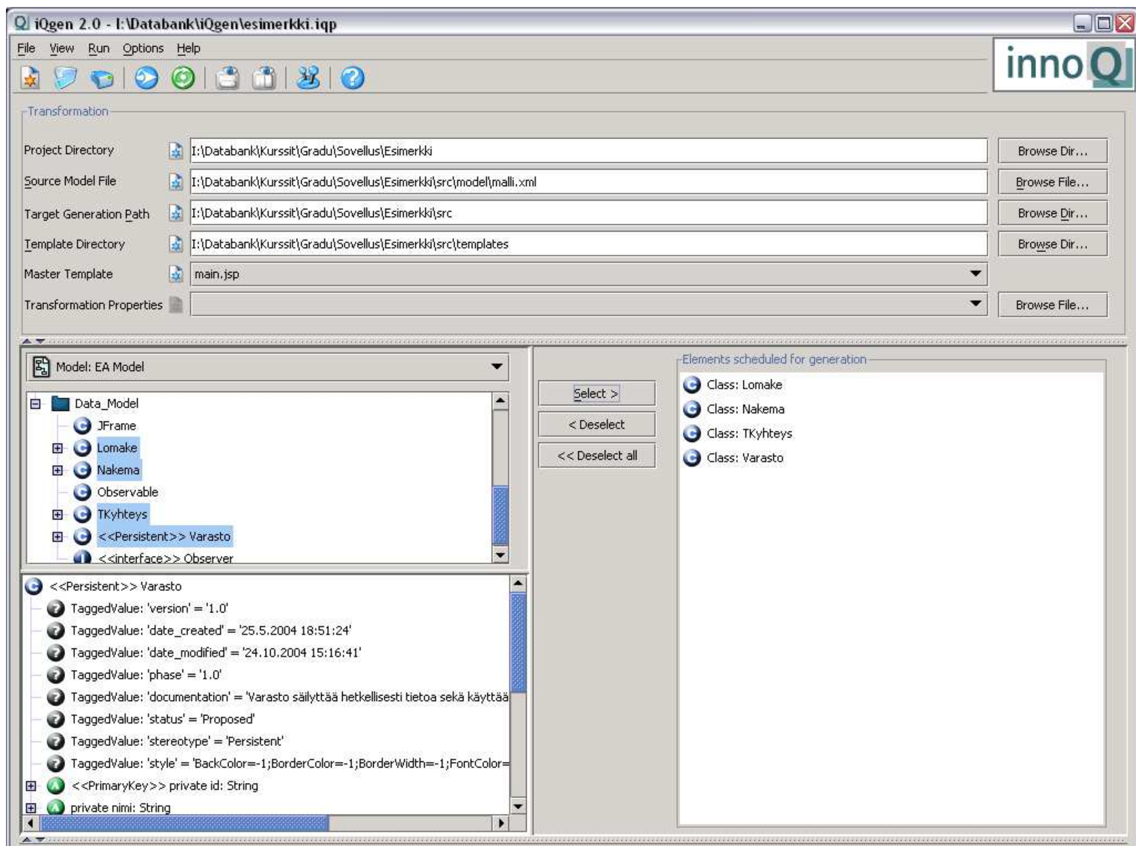
Taulukossa 3 kuvataan lyhyesti esimerkkisovelluksen teossa käytettyjen kaavaimien tehtävät. Kokonaisuudessaan kaavaimet löytyvät liitteestä 3. Kaavaimet ovat peräisin iQgen:in mukana tulleesta esimerkistä. Niistä on pyritty karsimaan joitakin monimutkaisimpia ja tarpeettomia osia, jotta kaavaimien toiminta tulisi mahdollisimman selkeästi esille. Kaavaimia on myös muokattu paremmin esimerkkisovellukseen soveltuviksi. Ne sisältävät silti muutamia esimerkin vuoksi säilytettyjä toiminnallisuuksia, kuten rajapintojen ja suhteiden muodostamisen, joita ei kuitenkaan ole tässä tapauksessa hyödynnetty.

Taulukko 3: Esimerkin generoinnissa käytetyt kaavaimet.

Kaavain	*Kuvaus*
main.jsp	Tarkastaa mallissa olevista luokista stereotyypin ja kutsuu sen perusteella oikeaa luokan generoivaa kaavainta tai kaavaimia.
drop_and_create.jsp	Generoi luokasta, jonka stereotyyppi on 'PERSISTENT', muun muassa tietokantataulun luovan SQL-lauseen.
interface.jsp	Generoi rajapintaluokan.
class.jsp	Generoi luokan rungon, attribuutit, operaatiot ja dokumentaation. Muodostaa luokan jokaiselle attribuutille get- ja set-operaatiot, jos luokan stereotyypiksi on määritelty 'PERSISTENT'. Käyttää apuna toiminnoissaan muita kaavaimia.
attributes.jsp	Muodostaa attribuutit ja tarvittaessa get- ja set-operaatiot.
operations.jsp	Muodostaa määritellyt operaatiot.
params.jsp	Muodostaa tarvittavat parametrit.
utilities.jsp	Sisältää hyödyllisiä operaatioita, kuten tyyppien tarkastamista sekä palautusarvon tutkimista.
copyright.jsp	Sisältää tekijänoikeudelliset tiedot.

Kaavaimien avulla voidaan toteuttaa monimutkaisiakin asioita koodingenerointiin liit-

tyen. Automaattisesti generoitavien ominaisuuksien lisäämisen suurimpana vaikeutena toimii kaavainten ohjelmointi, joka ei välttämättä ole helppoa. Sen vuoksi tämän tutkielman esimerkissä on käytetty vain yleisimpiä ja helpoimmin omaksuttavia ominaisuuksia, kuten luokkien, attribuuttien ja operaatioiden generointia. Kaavaimet mahdollistavat kuitenkin monien muidenkin asioiden, kuten luokkien välisten suhteiden, automaattisen generoinnin.



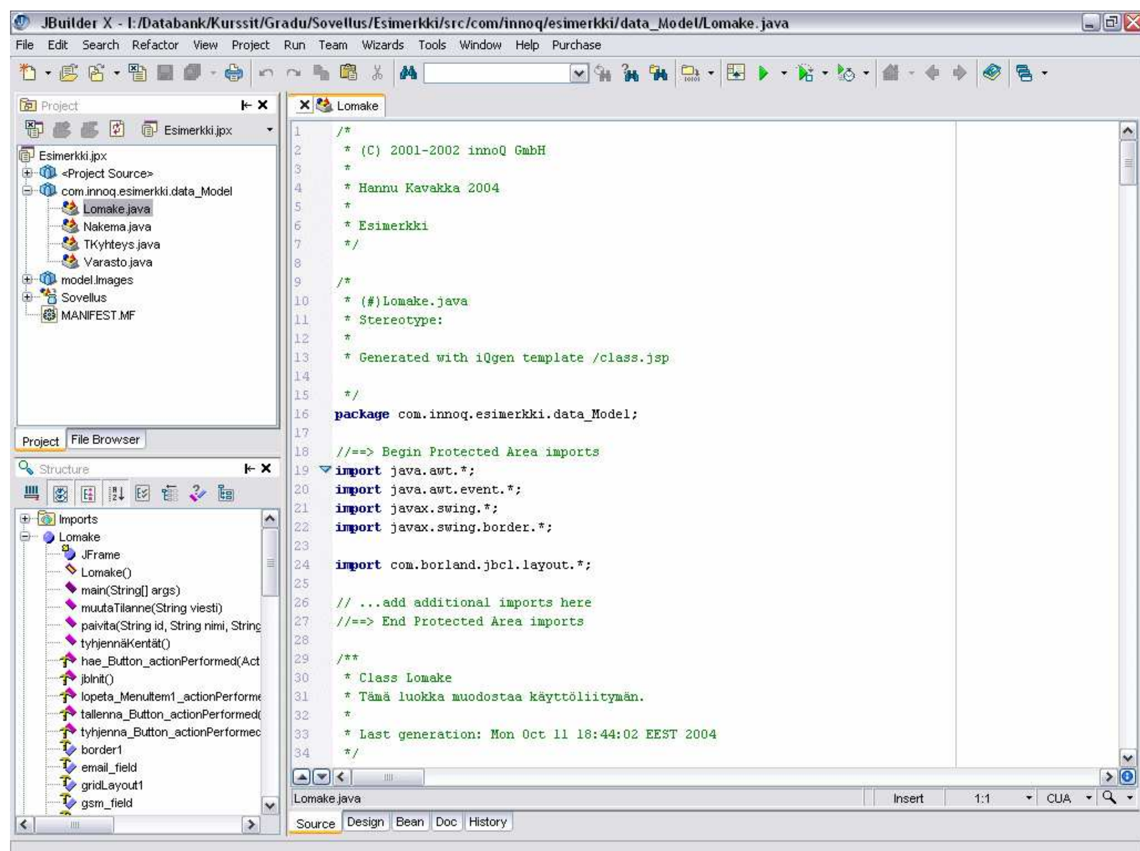
Kuva 28: iQgen.

Kuvassa 28 olevan iQgen:in käyttö alkaa valitsemalla projektille päähakemisto (Project Directory), joka sisältää koodingeneroinnin kannalta kaikki tarpeelliset tiedostot. Seuraavaksi määritetään kansio (Source Model File), josta löytyy XMI-muotoiseksi muunnettu PIM-malli. Jos malli on oikeaa XMI-formaattia (XMI 1.0), niin mallin sisältämät elementit ilmestyvät iQgen:in vasemman reunan laatikkoon valittaviksi. Tämän jälkeen tulee valita hakemisto (Target Generation Path) muodostettaville lähdekooditiedostoille. Viimeisenä on kerrottava polku kaavaimet sisältävään kansioon (Template Directory). Ennen generoitavien luokkien valitsemista, pitää määritellä vielä *pääkaavain* (Master Template), joka ohjaa koodingenerointia. Koodingenerointi tapahtuu siten, että ensin valitaan vasemman reunan laatikosta generoitavat luokat, mahdollises-

ti tietotyypit sekä muut elementit. Valitut elementit siirretään (Select) generoitavien listalle, jonka jälkeen valikosta (Run) voidaan suorittaa generointi (Generate). Kaikki tarpeellinen generoinnin yhteydessä syntyvä informaatio näytetään sovelluksen alareunaan ilmestyvässä laatikossa.

5.1.3 JBuilderX

Sovelluksia rakennettaessa on hyvä käyttää jotakin kehitysympäristöä helpottamaan ja nopeuttamaan ohjelmoimista. Tämän tutkielman esimerkin rakentamisessa on käytetty kuvan 29 Borland JBuilderX Foundation -nimistä Java:lle suunniteltua kehitysympäristöä (Borland, 2004). Sen avulla automaattisesti generoitua lähdekoodia voidaan muokata muun muassa toiminnallisuuden osalta. Kehitysympäristö osaa huomauttaa mahdollisista syntaksivirheistä ja se tarjoaa muutenkin apua esimerkiksi metodien käytössä.



Kuva 29: Borland JBuilderX Foundation.

Esimerkkisovellukseen on lisätty JBuilderX:n avulla toiminnallisuus ja varsinainen

käyttöliittymä, jonka rakentaminen oli helppoa graafisen rakennustilan avulla. Kaikkein tärkein kehitysympäristön ominaisuus on tietenkin projektin kääntäminen suorituskelpoisiksi tiedostoiksi. Ilman kehitysympäristöä lähdekoodin kääntäminen voi vaatia paljonkin vaivaa erilaisten polkujen ja ympäristömuuttujien määrittelemiseksi. JBuilderX:n avulla esimerkkisovellus saadaan suorituskelpoiseksi nopeasti ja siitä voidaan myös muodostaa pakattu JAR-tiedosto. JAR-tiedoston avulla sovellus voidaan siirtää tietokoneesta toiseen ilman hakemistorakenteiden tuntemista ja hankalien ympäristömuuttujien asettamista. Kaikki JAR-tiedoston muodostamiseen tarvittava tieto kuvataan MANIFEST.MF -tiedostossa.

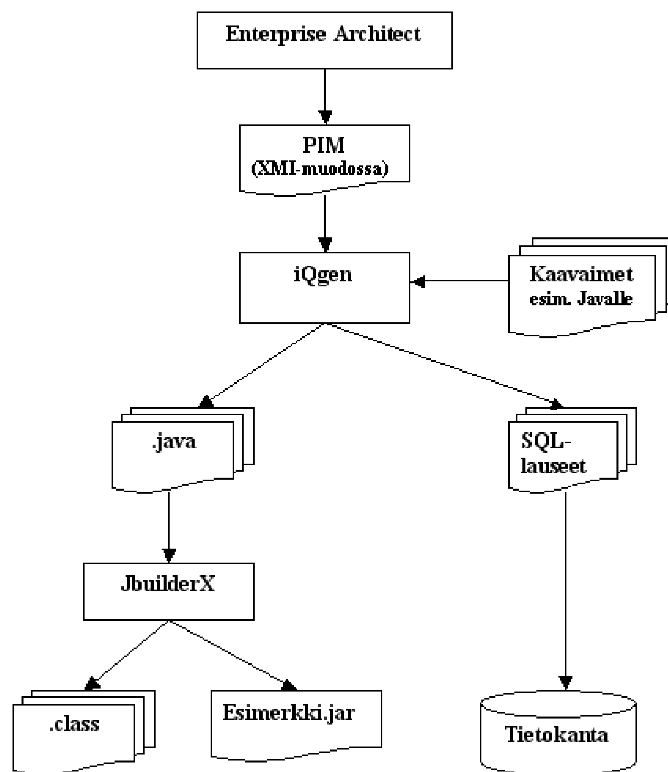
Koodingeneroijan kaavaimiin oli määritelty säännöt PIM-mallien sisältämän dokumentaation muuntamisesta Javadoc-muotoon. Lähdekoodiin joukkoon generoidut Javadoc-kommentit voidaan nyt JBuilderX:n avulla muuntaa HTML-sivuiksi, jotka ovat helposti luettavissa kaikilla web-selaimilla. Näin ollen sovelluksesta saadaan muodostettua matalan tason dokumentaatio puoliautomaattisesti.

5.2 Esimerkkisovellus

Seuraavaksi kuvataan pienimuotoisen esimerkkisovelluksen rakentamisen vaiheet. Esimerkki on erittäin yksinkertainen henkilötietoja MySQL[12]-tietokantaan tallentava Java-sovellus. Tallentamisen lisäksi sovelluksella voidaan myös hakea tietoja tietokannasta. Esimerkkisovelluksen rakentamisen tarkoitus on havainnollistaa malliperustaisen arkkitehtuurin käyttöä käytännön tasolla, eli sitä kuinka sovellus rakennetaan käyttäen aiemmin mainittuja MDA:n perusperiaatteita. Rakentamisen vaiheet selviävät parhaiten kuvasta 30, joka kuvaa käytettävät välineet sekä prosessissa syntyvät artefaktit. Esimerkkisovellusta rakennettaessa on noudatettu, kohdassa 3.2.1 kuvatun, notkean MDA:n mukaisia vaiheita.

Ensimmäisenä työvaiheena on vaatimusten keruu ja niiden analysointi, jonka aikana pyritään ymmärtämään mitä sovelluksen tulisi tehdä ja mitä käsitteitä siihen liittyy. Toisena työvaiheena on luokkakaavion, eli PIM-mallin, rakentaminen EA:n avulla aiemmin määritellyistä vaatimuksista. Luokkakaavio, joka löytyy kokonaisuudessaan liitteestä 1, koostuu taulukon 4 kuvailemista Java-luokista. Sen tarkoituksena on selvittää sovelluksen moduulit sekä niiden väliset suhteet.

[12]MySQL on suosittu avoimen lähdekoodin tietokanta (MySQL, 2004).



Kuva 30: Työn vaiheet esimerkin rakentamisessa.

Esimerkin luokat pohjautuvat Model-View-Controller -suunnittelukaavaan, jonka periaate on Darwinin (1999) ja GuangChunin & al. (2003) mukaan erottaa käyttöliittymällä varustetun sovelluksen toiminnallisuus kolmeen eri osaan: *malliin* (model), *näkymään* (view) ja *ohjaimeen* (controller). Malli pitää sisällään varsinaisen tiedon sekä siihen liittyvän toiminnallisuuden. Näkymä taas esittää mallin sisältämän tiedon tai näyttää käyttäjälle sovelluksen käyttöliittymän. Ohjain määrittää kuinka käyttöliittymä reagoi käyttäjän antamiin syötteisiin.

PIM-mallin tultua valmiiksi, se voidaan muuntaa EA:n omien toimintojen avulla XMI-formaattiin. Esimerkki XMI-muotoisesta mallista löytyy liitteestä 2, johon on koottu näyte luokasta, attribuuttista sekä operaatiosta. Tehdystä muunnoksesta johtuen koodingeneroija iQgen pystyy nyt vastaanottamaan mallin syötteenä ja tekemään kaavaimissa määrätyt toiminnot Java-koodin sekä SQL-lauseiden muodostamiseksi. Syntyneitä Java-tiedostoja, kuten liitteestä 5 löytyvää 'Varasto.java' -tiedostoa, voidaan muokata JBuilderX:n avulla. Kaikki lähdekoodiin tulevat lisäykset ja muutokset tehdään aina koodingeneroijan muodostamiin *suojattuihin alueisiin* (protected area). Tällä pyritään siihen, että sovelluksen kehittämisen yhteydessä notkean MDA:n mukaiset ite-

raatiot, eli koodin uudelleengeneroinnit, eivät hävittäisi jo käsin tehtyjä lisäyksiä.

Kun tarpeelliset lisäykset on tehty, voidaan sovellus kääntää JBuilderX:n avulla, jolloin muodostuvat suorituskelpoiset class-tiedostot sekä paketoitu tiedosto Esimerkki.jar. Tässä yhteydessä on huomattava, että Java:lle on kerrottava missä tietokantayhteyden Java:n sekä MySQL:n välille muodostava jdbc-ajuri sijaitsee. Ennen esimerkkisovelluksen käynnistämistä, täytyy vielä MySQL-tietokantaan tehdä seuraavat toimenpiteet: luodaan tarvittava tietokanta, johon liitetään käyttäjätunnus sekä salasana. Tämän jälkeen suoritetaan koodingeneroinnin yhteydessä muodostettu, liitteestä 4 löytyvä SQL-lause taulun 'Varasto' luomiseksi.

Kun esimerkkisovellus on käynnistetty, ilmestyy kuvan 31 mukainen käyttöliittymä. Jos sovellus on muodostanut tietokantayhteyden onnistuneesti, ilmestyy sovelluksen oikean alareunan laatikkoon teksti "Tietokantayhteys muodostettu.". Jos tietokantayhteyttä ei jostain syystä voitu muodostaa, ilmestyy samaan laatikkoon viesti "Tietokantayhteyttä ei voitu muodostaa." ja samalla kaikki sovelluksen painikkeet (*Hae id:llä*, *Tallenna*, *Tyhjennä*) on poistettu käytöstä. Vain sovelluksen päävalikosta (*Valikko*) löytyvä toiminto (*Lopeta*) on käytettävissä.

Taulukko 4: Esimerkkisovelluksen luokat ja niiden tehtävät.

Luokka	*Kuvaus*
Lomake.java	Muodostaa sovelluksen käyttöliittymän ja huolehtii kutsuttaessa lomakkeen kenttien päivityksestä.
Varasto.java	Tämän luokan perusteella muodostetaan tietokantataulu. Varasto huolehtii tietojen tallentamisesta sekä niiden hakemisesta tietokannasta (kutsuu muita metodeja). Ilmoittaa tarkkailijalle (Nakema) tapahtuneista muutoksista, jotka pitäisi välittää käyttöliittymälle (Lomake).
Nakema.java	Tarkkailee mallia (Varasto) mahdollisten käyttöliittymälle (Lomake) välitettävien muutosten varalta. Muutosten tapahtuessa informoi käyttöliittymää.
TKyhteys.java	Hoitaa sovelluksen ja tietokannan välisen liikenteen. Avaa ja sulkee tietokantayhteyden sekä suorittaa tiedon lisäämisen ja hakemisen.

Kuva 31: Sovelluksen normaali käyttöliittymä.

Sovelluksen toimintojen käyttäminen tapahtuu siten, että ensin täydennetään lomakkeelle tiedot, jonka jälkeen painetaan halutun toiminnon painiketta. Tietojen tallentamisen yhteydessä lomakkeelle voidaan syöttää henkilön *id*, *nimi*, *osoite*, *postinumero*, *postitoimipaikka*, *GSM*-numero sekä *e-mail* -osoite. Jos syötetyt tiedot on tallennettu onnistuneesti tietokantaan, ilmestyy sovelluksen oikean alareunan laatikkoon teksti ”Tallennettu *id*:n tiedot.”. Haettaessa henkilön tietoja tietokannasta, täydennettävänä tietona on vain id. Jos haettavan henkilön tiedot löytyvät onnistuneesti tietokannasta, ilmoitetaan siitä jälleen *Tapahtumat* -laatikossa. Tyhjennä -painike tyhjentää nimensä mukaisesti lomakkeen kentät uuden henkilön tietojen lisäämistä tai etsimistä varten. Sovellus voidaan lopettaa päävalikon toiminnosta ’Lopeta’. Esimerkkisovelluksen luonteen vuoksi siihen ei ole toteutettu sen suurempia virheiden tai syötteiden tarkastuksia – tietenkään sovelluksen ei tule kaatua virhetilanteiden sattuessa.

6 Yhteenveto

Ohjelmistokehitysprosessien jatkuva parantaminen on luonnollinen seuraus koko ajan kasvavista laatu- ja tehokkuusvaatimuksista. Työskentelyn pitäisi olla tehokkaampaa sekä laatua parantavaa. Teknologioiden jatkuva kehittyminen on omalta osaltaan pakottanut ohjelmistokehitysprosesseja mukautumaan jatkuviin muutoksiin. Teknologioissa tapahtuvien muutosten vuoksi järjestelmät pitäisi rakentaa siten, että niitä voitaisiin käyttää pidemmänkin ajan kuluttua. Tämän tavoitteen saavuttaminen edellyttää nykypäivänä, ja etenkin tulevaisuudessa, muun muassa heterogeenisten järjestelmien saumatonta yhteentoimivuutta sekä tarkkaa, ajantasaista dokumentaatiota.

Erilaisten teknologioiden määrän huomioiminen on aloitettava jo ohjelmistokehitysprosessin tasolla. Järjestelmän mallintaminen jokaiselle halutulle sovellusalustalle erikseen ei ole järkevää, jos on mahdollista työskennellä toisin. Malliperustaisen arkkitehtuurin avulla järjestelmät voidaan mallintaa sovellusalustariippumattomin termein, PIM-malleina, jolloin säästetään niin aikaa kuin rahaakin ja voidaan keskittyä olennaiseen, eli järjestelmän liiketoiminnan kuvaamiseen. PIM-mallien käytöstä seuraa myös se, että järjestelmien siirtäminen alustalta toiselle on helppoa transformaatiomääritysten avulla. Näin malleista voidaan transformaation kautta generoida minkä tahansa sovellusalustan mukaista lähdekoodia. Lähdekoodin joukkoon voidaan generoida myös määrätyllä tavalla muotoiltuja kommentteja, joista voidaan myöhemmin muodostaa järjestelmän matalan tasan dokumentaatio esimerkiksi Javadoc:in avulla. PIM-mallit toimivat myös järjestelmän korkean tason dokumentaationa. Ne pysyvät aina ajan tasalla, koska MDA-prosessin luonteen mukaisesti kaikki muutokset tehdään ensin malleihin, joista lähdekoodi automaattisesti generoidaan. Tässä kohdassa on otettava huomioon kuitenkin se, että järjestelmän korkean tason dokumenteissa kuvataan toki myös toiminnallisuutta. Malliperustaisen arkkitehtuurin tulevaisuuden haasteisiin kuuluukin se, kuinka mahdollistetaan mallien käyttö ja niistä lähdekoodin generoiminen järjestelmän toiminnalliset vaatimukset huomioon ottaen.

Mielestäni malliperustainen arkkitehtuuri on yksi varteenotettavista vaihtoehdoista mietittäessä tulevaisuudessa käytettäviä ohjelmistokehitysprosesseja. MDA:n käytön yleistyminen edellyttää vielä tarvittavien työkalujen määrän lisääntymistä, ominaisuuksien kehittymistä sekä myös keskinäistä yhdistymistä yhdeksi käytettäväksi työkaluksi, jolloin MDA:n käyttöönottamisen kynnys madaltuisi merkittävästi. MDA ei välttämättä sovellu jokaiseen ohjelmistoprojektiin, koska mallien rakentaminen voi

viedä kohtuuttomasti aikaa niiden tarjoamiin hyötyihin nähden. Ajan käytön merkitys on kuitenkin aina mietittävä projektikohtaisesti, eikä siihen voi antaa yleispätevää ratkaisua. Mielestäni suuret sekä keskisuuret ylläpidettävät projektit hyötyisivät eniten MDA:n käytöstä, koska niissä mallien tuomat edut ovat selkeästi havaittavissa. Mallit selkeyttävät monimutkaisen järjestelmän toimintaa. Niistä generoitava lähdekoodi mahdollistaa ohjelmoijien kallisarvoisen työpanoksen siirtämisen mielenkiintoisempien ohjelmointitöiden yhteyteen. Pienemmät ohjelmistoprojektit, joita ei välttämättä ole tarkoitus ylläpitää, voidaan rakentaa jollakin vaihtoehtoisella menetelmällä, kuten XP-ohjelmoinnilla. Toisaalta pieniin projekteihin soveltuu myös notkea MDA, joka on erinomainen vaihtoehto XP-ohjelmoinnin ohella valittaessa projektiin soveltuvaa ohjelmistokehitysprosessia. Kaiken kaikkiaan MDA on mielenkiintoinen ja kehityskelpoinen menetelmä, joka kannattaa ottaa huomioon valittaessa kehitysprosessia vaativiin ohjelmistoprojekteihin.

Viitteet

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. (2002) *Agile software development methods: Review and analysis.* VTT, Espoo (Saatavana myös: `http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf`, 5.9.2004).

Agile Alliance (2004) *AgileAlliance.* WWW-sivusto, `http://www.aanpo.org/` (9.9.2004).

Ambler, S. W. (2004) *A Roadmap for Agile MDA.* `http://www.agilemodeling.com/essays/agileMDA.htm` (11.9.2004).

Arlow, J., Neustadt, I. (2003) *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Pearson Education Inc., Boston.

Boehm, B. W. (1988) A Spiral Model of Software Development and Enhancement. *Computer* **21**(5), 61–72.

Borland (2004) *Borland Nordic.* WWW-sivusto, `http://www.borland.se/` (7.11.2004).

Brown, A. (2004) *An introduction to Model Driven Architecture Part I: MDA and today's systems.* `http://www-106.ibm.com/developerworks/rational/library/3100.html` (10.6.2004).

Chang, D. T. (2001) *CWM Enablement Showcase: Warehouse Metadata Interchange Made Easy Using CWM.* IBM (Saatavana myös: `http://www-900.ibm.com/cn/software/db2/products/download/whitepaper/cwm.pdf`, 16.8.2004).

Cooper, D., Khoo, B., von Konsky, B. R., Robey, M. (2004) Java Implementation Verification Using Reverse Engineering. *Proceedings of the 27th conference on Australasian computer science - Volume 26*, Australian Computer Society Inc, Darlinghurst, Australia, 203–211.

Damm, C., Hansen, K., Thomsen, M., Tyrsted, M. (2000) Tool integration: experiences and issues in using XMI and component technology. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*, IEEE, Mont-Saint-Michel, France, 94-107.

Darwin, I. (1999) GUI Development with Java. *Linux Journal* **1999**(61).

Eriksson, H-E., Penker, M. (2000) *UML.* Oy Edita Ab, Jyväskylä.

Ferguson, J., Cooper, J., Falat, M., Fischer, M., Guido, A., Marciniak, J., Matejceck, J., Webster, R. (1996) *Software Acquisition Capability Maturity Model (SA-CMM).* CMU/SEI-96-TR-020, Software Engineering Institute, Pittsburgh (Saatavana myös: `http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr020.96.pdf`, 31.8.2004).

Frankel, D. (2002) *Using Model-Driven Architecture to Manage Metadata.* IONA Technologies PLC (Saatavana myös: `http://www.iona.com/whitepapers/Model-Driven-Metadata-Management-v01-02.pdf`, 10.6.2004).

Frankel, D. S. (2003) *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, Indianapolis, Indiana.

Grose, T. J., Doney, G. C., Brodsky, S. A. (2002) *Mastering XMI: Java Programming with XMI, XML, and UML.* John Wiley & Sons, New York.

GuangChun, L., Lu, W., Hanghong, X. (2003) A Novel Web Application Frame Developed by MVC. *ACM SIGSOFT Software Engineering Notes* **28**(2), 7.

Haikala, I., Märijärvi, J. (1998) *Ohjelmistotuotanto.* Suomen Atk-kustannus Oy, Jyväskylä.

Herrington, J. (2003) *Code Generation in Action.* Manning Publications, USA.

Herrington, J. D. (2004) *Code Generation Network.* WWW-sivusto, `http://www.codegeneration.net/` (16.9.2004).

Highsmith, J., Cockburn, A. (2001) Agile Software Development: The Business of Innovation. *Computer* **34**(9), 120–122.

innoQ (2004) *innoQ iQgen – The Model Driven Software Generator.* WWW-sivusto, `http://www.innoq.com/iqgen/` (1.11.2004).

JCP (2002) *Java Metadata Interface(JMI) Specification.* JSR 040, Java Community Process (Saatavana myös: `http://jcp.org/aboutJava/communityprocess/final/jsr040/`, 14.8.2004).

Kleppe, A., Warmer, J., Bast, W. (2003) *MDA Explained: The Model Driven Architecture: Practice and Promise.* Pearson Education Inc., Boston.

Kobryn, C. (1999) UML 2001: A Standardization Odyssey. *Communications of the ACM* **42**(10), 29–37.

Kobryn, C. (2002) Will UML 2.0 Be Agile or Awkward?. *Communications of the ACM* **45**(1), 107–110.

Kruchten, P. (2004) *The Rational Unified Process: An Introduction.* Pearson Education Inc., Boston.

McConnell, S. (2002) *Ohjelmistotuotannon hallinta.* Edita Publishing Oy, Helsinki.

Medina, E., Trujillo, J. (2002) A Standard for Representing Multidimensional Properties: The Common Warehouse Metamodel (CWM). *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, Springer-Verlag, London, UK, 232–247.

Mellor, S. J., Balcer, M. J. (2002) *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, Boston.

Mellor, S. J. (2004) *Agile MDA.* Project Technology, Inc (Saatavana myös: `http://www.omg.org/mda/mda_files/Agile_MDA.pdf`, 9.9.2004).

Mellor, S. J., Scott, K., Uhl, A., Weise, D. (2004) *MDA Distilled: Principles of Model-Driven Architecture.* Pearson Education Inc., Boston.

Mellor, S. J., Watson, A. (2004) *Roles in the MDA Process.* Object Management Group (Saatavana myös: `http://www.omg.org/registration/Roles_in_MDA1.pdf`, 9.6.2004).

Miller, J., Mukerji, J. (2001) *Model Driven Architecture - A Technical Perspective.* ormsc/2001-07-01, Object Management Group (Saatavana myös: `http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf`, 27.4.2004).

Miller, J., Mukerji, J. (2003) *MDA Guide.* omg/2003-06-01, Object Management Group (Saatavana myös: `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`, 27.4.2004).

MySQL (2004) *MySQL: The World's Most Popular Open Source Database.* WWW-sivusto, http://www.mysql.com/ (7.11.2004).

OMG (2002) *Meta Object Facility (MOF) Specification.* Version 1.4, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/doc?formal/2002-04-03, 13.7.2004).

OMG (2003a) *Common Warehouse Metamodel (CWM) Specification.* Version 1.1, formal/03-03-02, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?formal/03-03-02.pdf, 13.8.2004).

OMG (2003b) *OMG Unified Modeling Language Specification.* Version 1.5, formal/03-03-01, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf, 6.7.2004).

OMG (2003c) *UML 2.0 Diagram Interchange Specification.* ptc/03-09-01, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-01.pdf, 4.8.2004).

OMG (2003d) *UML 2.0 Object Constraint Language Specification.* ptc/03-10-14, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf, 2.8.2004).

OMG (2003e) *XML Metadata Interchange (XMI) Specification.* Version 2.0, formal/03-05-02, Object Management Group (Saatavana myös: http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf, 2.8.2004).

OMG (2004a) *Catalog of Domain Specifications.* http://www.omg.org/technology/documents/domain_spec_catalog.htm (5.8.2004).

OMG (2004b) *Model Driven Architecture.* WWW-sivusto, http://www.omg.org/mda/ (30.06.2004).

OMG (2004c) *Object Management Architecture.* WWW-sivusto, http://www.omg.org/oma/ (30.06.2004).

Pender, T. (2003) *UML Bible.* Wiley Publishing, Indianapolis, Indiana.

Poole, J., Chang, D., Tolbert, D., Mellor, D. (2002) *Common Warehouse Metamodel: An Introduction to the Standard for Data Warehousing Integration.* John Wiley & Sons,

New York.

Pressman. R. S., Ince, D. (2000) *Software Engineering: A Practitioner's Approach: European Adaptation.* McGraw-Hill International, UK.

Rumbaugh, J., Jacobson, I., Booch, G. (1999) *The Unified Modeling Language Reference Manual.* Addison Wesley Longman Inc., Reading, MA.

Siegel, J. (2001) *Developing in OMG's Model-Driven Architecture.* Object Management Group (Saatavana myös: `ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf`, 9.7.2004).

Sparx Systems (2004) *Enterprise Architect – Full Lifecycle UML Modeling Software.* WWW-sivusto, `http://www.sparxsystems.com.au/` (1.11.2004).

Succi, G., Marchesi, M. (2001) *Extreme Programming Examined.* Addison-Wesley, Boston.

Sun Microsystems (1999) *The Java HotSpot Performance Engine Architecture.* `http://java.sun.com/products/hotspot/whitepaper.html` (5.12.2004).

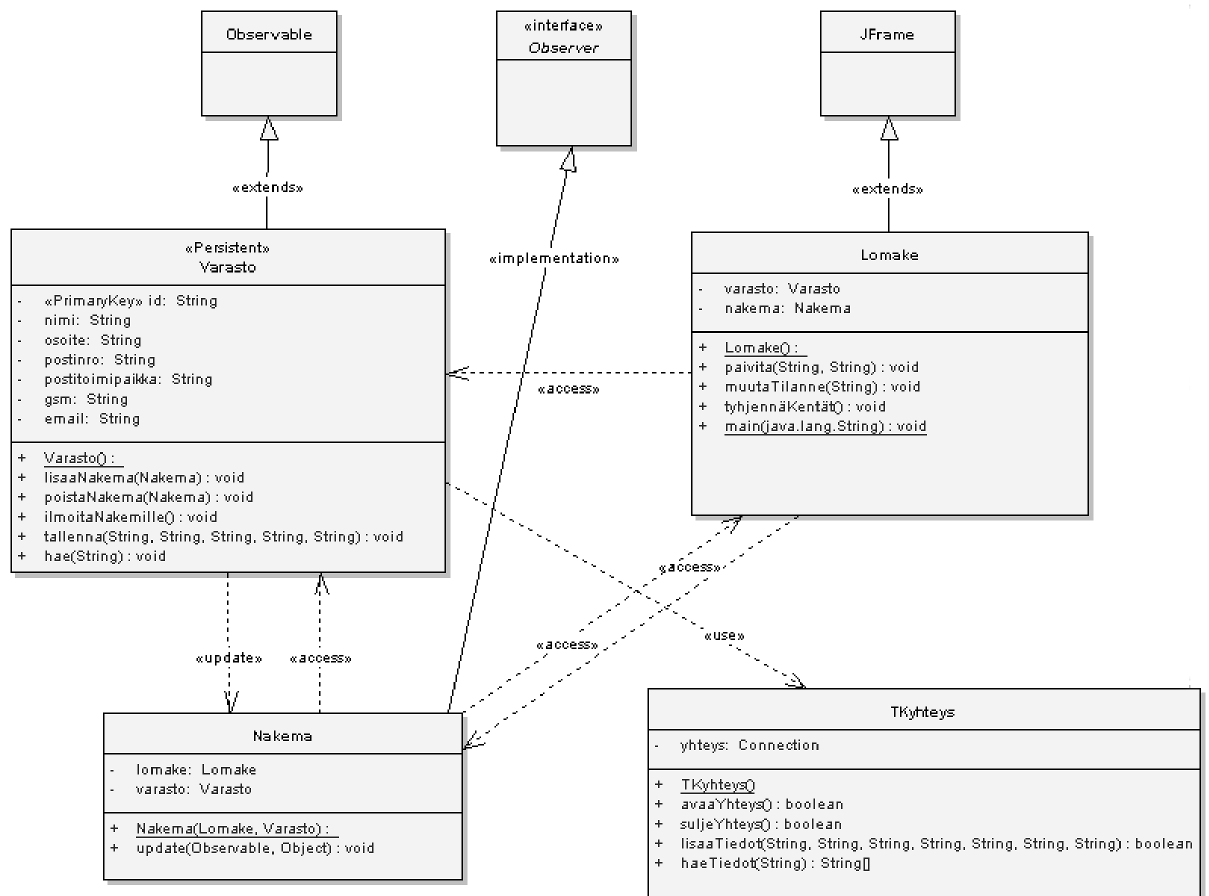
Sun Microsystems (2004) *Java Technology.* WWW-sivusto, `http://java.sun.com/` (6.11.2004).

Thomas, D. (2004) MDA: Revenge of the Modelers or UML Utopia?. *IEEE Software* **21**(3), 22–24.

Wang, H., Zhang, D. (2003) MDA-based Development of E-Learning System. *International Computer Software and Applications Conference (COMPSAC'03)*, IEEE Computer Society.

Warmer, J., Kleppe, A. (2003) *The Object Constraint Language Second Edition: Getting Your Models Ready for MDA.* Pearson Education Inc., Boston.

Liite 1: Esimerkin luokkakaavio



Liite 2: Näyte XMI-tiedostosta

Varasto -luokka

```
<Foundation.Core.Class xmi.id="EAID_BA111692_41BF_404a_BB44_665B16A91247">
<Foundation.Core.ModelElement.name>Varasto</Foundation.Core.ModelElement.name>
<Foundation.Core.ModelElement.visibility xmi.value="public"/>
<Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
<Foundation.Core.Class.isActive xmi.value="false"/>
-<Foundation.Core.ModelElement.namespace>
  <Model_Management.Package xmi.idref="EAPK_F0F1B9A0_3F79_4a82_8D1A_205E33BCB323"/>
</Foundation.Core.ModelElement.namespace>
-<Foundation.Core.GeneralizableElement.generalization>
  <Foundation.Core.Generalization xmi.idref="EAID_944D8281_4DC6_4cb4_8CBF_CF6A219B9878"/>
</Foundation.Core.GeneralizableElement.generalization>
-<Foundation.Core.ModelElement.clientDependency>
  <Foundation.Core.Dependency xmi.idref="EAID_5CB27CD3_4674_4516_B281_7D0AE3722014"/>
  <Foundation.Core.Dependency xmi.idref="EAID_4E25E439_15D5_4491_947B_545E67A50D48"/>
</Foundation.Core.ModelElement.clientDependency>
-<Foundation.Core.ModelElement.supplierDependency>
  <Foundation.Core.Dependency xmi.idref="EAID_B863281F_4807_4e57_9F0D_2700025DFC99"/>
  <Foundation.Core.Dependency xmi.idref="EAID_F2E07F0A_BAEF_41e1_9E3E_E2DEB194B82B"/>
</Foundation.Core.ModelElement.supplierDependency>
-<Foundation.Core.Classifier.feature>
+ <Foundation.Core.Attribute xmi.id="eaxmiid71">
+ <Foundation.Core.Attribute xmi.id="eaxmiid73">
+ <Foundation.Core.Attribute xmi.id="eaxmiid74">
+ <Foundation.Core.Attribute xmi.id="eaxmiid75">
+ <Foundation.Core.Attribute xmi.id="eaxmiid76">
+ <Foundation.Core.Attribute xmi.id="eaxmiid77">
+ <Foundation.Core.Attribute xmi.id="eaxmiid78">
+ <Foundation.Core.Operation xmi.id="eaxmiid79">
+ <Foundation.Core.Operation xmi.id="eaxmiid82">
+ <Foundation.Core.Operation xmi.id="eaxmiid86">
+ <Foundation.Core.Operation xmi.id="eaxmiid90">
+ <Foundation.Core.Operation xmi.id="eaxmiid93">
+ <Foundation.Core.Operation xmi.id="eaxmiid103">
</Foundation.Core.Classifier.feature>
</Foundation.Core.Class>
```

Varasto -luokan attribuutti ’id’

```
<Foundation.Core.Attribute xmi.id="eaxmiid71">
  <Foundation.Core.ModelElement.name>id</Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.visibility xmi.value="private"/>
  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
  <Foundation.Core.Feature.ownerScope xmi.value="classifier"/>
  <Foundation.Core.StructuralFeature.multiplicity>
    <Foundation.Data_Types.Multiplicity>
      <Foundation.Data_Types.Multiplicity.range>
        <Foundation.Data_Types.MultiplicityRange>
          <Foundation.Data_Types.MultiplicityRange.lower>1
          </Foundation.Data_Types.MultiplicityRange.lower>
          <Foundation.Data_Types.MultiplicityRange.upper>1
          </Foundation.Data_Types.MultiplicityRange.upper>
        </Foundation.Data_Types.MultiplicityRange>
      </Foundation.Data_Types.Multiplicity.range>
    </Foundation.Data_Types.Multiplicity>
  </Foundation.Core.StructuralFeature.multiplicity>
  <Foundation.Core.StructuralFeature.changeability xmi.value="changeable"/>
  <Foundation.Core.StructuralFeature.targetScope xmi.value="instance"/>
  <Foundation.Core.Attribute.initialValue>
    <Foundation.Data_Types.Expression>
      <Foundation.Data_Types.Expression.language/>
      <Foundation.Data_Types.Expression.body/>
    </Foundation.Data_Types.Expression>
  </Foundation.Core.Attribute.initialValue>
  <Foundation.Core.StructuralFeature.type>
    <Foundation.Core.DataType xmi.idref="eaxmiid15"/>
  </Foundation.Core.StructuralFeature.type>
</Foundation.Core.Attribute>
```

Varasto -luokan operaatio ’lisaaNakema’

```
<Foundation.Core.Operation xmi.id="eaxmiid82">
  <Foundation.Core.ModelElement.name>lisaaNakema</Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.visibility xmi.value="public"/>
  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
  <Foundation.Core.Feature.ownerScope xmi.value="instance"/>
  <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
  <Foundation.Core.Operation.concurrency xmi.value="sequential"/>
  <Foundation.Core.Operation.isRoot xmi.value="false"/>
  <Foundation.Core.Operation.isLeaf xmi.value="false"/>
  <Foundation.Core.Operation.isAbstract xmi.value="false"/>
  <Foundation.Core.Operation.specification/>
  <Foundation.Core.BehavioralFeature.parameter>
    <Foundation.Core.Parameter xmi.id="eaxmiid84">
      <Foundation.Core.ModelElement.name/>
      <Foundation.Core.ModelElement.visibility xmi.value="private"/>
      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
      <Foundation.Core.Parameter.defaultValue>
        <Foundation.Data_Types.Expression>
          <Foundation.Data_Types.Expression.language/>
          <Foundation.Data_Types.Expression.body/>
        </Foundation.Data_Types.Expression>
      </Foundation.Core.Parameter.defaultValue>
      <Foundation.Core.Parameter.kind xmi.value="return"/>
      <Foundation.Core.Parameter.type>
        <Foundation.Core.DataType xmi.idref="eaxmiid36"/>
      </Foundation.Core.Parameter.type>
    </Foundation.Core.Parameter>
    <Foundation.Core.Parameter xmi.id="eaxmiid85">
      <Foundation.Core.ModelElement.name>view</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility xmi.value="private"/>
      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
      <Foundation.Core.Parameter.defaultValue>
        <Foundation.Data_Types.Expression>
          <Foundation.Data_Types.Expression.language/>
          <Foundation.Data_Types.Expression.body/>
        </Foundation.Data_Types.Expression>
      </Foundation.Core.Parameter.defaultValue>
      <Foundation.Core.Parameter.kind xmi.value="in"/>
      <Foundation.Core.Parameter.type>
        <Foundation.Core.Classifier xmi.idref="EAID_CC296019_4D4B_4c0e_A379_04A1B3FBFD90"/>
      </Foundation.Core.Parameter.type>
    </Foundation.Core.Parameter>
  </Foundation.Core.BehavioralFeature.parameter>
</Foundation.Core.Operation>
```

Liite 3: Kaavaimet

Main.jsp

```
<%--
Main template for esimerkki
$Id: main.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page extends="com.innoq.generator.jsp.JspMain"
    import="ru.novosoft.uml.MBase,java.util.Collection" %>
<%!
    private static final String[] PERSISTENT =  {"class.jsp", "drop_and_create.jsp"};
    private static final String[] INTERFACE  =  {"interface.jsp"};
    private static final String[] DEFAULT    =  {"class.jsp"};
    private static final String[] POST       =  {"build.jsp"};
%>

<%!
    public String[] getTemplates(MBase element)  {
        String s = getMetaModel().getStereotype(element);
        String name = getMetaModel().getName(element);
        if ("Persistent".equals(s)) {
            return PERSISTENT;
        } else if (getMetaModel().isInterface(element)) {
            return INTERFACE;
        } else {
            return DEFAULT;
        }
    }

    public String getName() {
        return "esimerkki";
    }

    public String[] getPostprocessTemplates()  {
        return POST;
    }
%>
```

drop_and_create.jsp

```
<%--
Simple SQL script.
$Id: drop_and_create.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page extends="com.innoq.generator.jsp.JspBase"
    import="java.io.BufferedWriter,
        java.util.Date,
        java.util.Iterator,
        java.util.Collection,
        java.io.File,
```

```
        com.innoq.generator.*,
        ru.novosoft.uml.foundation.core.MAssociationEnd,
        ru.novosoft.uml.MBase"
%><%@ taglib uri="iqgen" prefix="iqgen" %>
<%@include file="params.jsp" %>
<%@include file="utilities.jsp" %>
--
-- SQL-script of <%=classname%>
-- Last generation: <%=new java.util.Date()%>
--
-- Generated with iQgen template <iqgen:jspfilename/>
<%!
    public String getFilename() {
        return getDirectory()+File.separatorChar+"sql"+File.separatorChar
                +getMetaModel().getName(element)+".sql";
    }
/* Modified for MySQL by HK 27.9.2004 */
    public String getDBType(MBase pAttribute) {
        String type = getMetaModel().getTypeName(pAttribute);
        if (type.equals("Date")) {
            return "DATE";
        } else if (type.equals("int")) {
            return "TINYINT";
        } else if (type.equals("long") || type.equals("Integer")) {
            return "INT";
        } else if (type.equals("boolean")) {
            return "BOOLEAN";
        } else {
            return "VARCHAR("+getLength(pAttribute)+")";
        }
    }
%>
drop table <%=classname%>;

create table <%=classname%> (<%
    Collection attr = getMetaModel().getAttributes(getElement());
    Iterator attr_it = attr.iterator();
    while (attr_it.hasNext()) {
        MBase attribute = (MBase)attr_it.next();
        String attrName = getMetaModel().getName(attribute);
        if (attrName.length()==0) {
            break;
        }
%> <%=attrName%> <%=getDBType(attribute)%>
   <%= isPrimaryKey(attribute)?"PRIMARY KEY ":""%><%=attr_it.hasNext()?", ":""%><%
    }
    Collection ass = getMetaModel().getAssociations(getElement());
    Iterator ass_ita = ass.iterator();
    while(ass_ita.hasNext()) {
        MAssociationEnd here = (MAssociationEnd)ass_ita.next();
        MAssociationEnd opp = here.getOppositeEnd();
        String oppName = StringHelper.fd(getMetaModel().getName(opp));
        String st = getMetaModel().getStereotype(opp.getType());
        if (oppName.length()==0) {
```

```
            oppName = StringHelper.fd(getMetaModel().getName(here));
            st = getMetaModel().getStereotype(here.getType());
        }
        int hereHigh = getMetaModel().getMultUpper(here.getMultiplicity());
        int high = getMetaModel().getMultUpper(opp.getMultiplicity());

        if ("Aufzaehlungstyp".equals(st) || "DynamischerAufzaehlungstyp".equals(st)) {
            out.print(", " + oppName+"Key INTEGER ");
        } else if (hereHigh==-1 && high==-1) {
            continue;
        } else if (hereHigh==-1 && high==1) {
            out.print(", " + oppName + " VARCHAR(50) ");
        } else if (hereHigh==1 && high==-1) {
            continue;
        } else {
            out.print(", " + oppName + " VARCHAR(50) ");
        } // end if
    } // end while
%>);
```

interface.jsp

```
<%@include file="copyright.jsp" %>
<%--

Declares all methods defined an interface.

$Id: interface.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page import="ru.novosoft.uml.MBase,java.util.Collection,java.util.Iterator"
extends="com.innoq.generator.jsp.JspBase" %>
<%@ taglib uri="iqgen" prefix="iqgen" %>
<%!
    // Helper methods

    private String getClassname() {
        return getMetaModel().getName(getElement());
    }

    private String getExtendsString() {
        Collection superclasses = getMetaModel().getAllSuperclasses(getElement());
        if (superclasses.isEmpty()) return "";
        StringBuffer sb = new StringBuffer(" extends ");
        for (Iterator i=superclasses.iterator(); i.hasNext();) {
            sb.append(getMetaModel().getName((MBase)i.next()));
            if (i.hasNext()) sb.append(", ");
        }
        return sb.toString();
    }

    public String getPrefix(char pSeparator, MBase m)  {
        return "com"+pSeparator+"innoq"+pSeparator+"esimerkki"+pSeparator;
    }
```

```
%>
package <%=getPath()%>;

<iqgen:import prefix="<%=getPrefix()%>" collectiontype="java.util.List"/>

/**
 * Interface <%=getClassname()%>
 *
 * Last generation: <%=new java.util.Date()%>
 *
 * Generated with iQgen template <iqgen:jspfilename/>
 */
public interface <%=getClassname()%><%=getExtendsString()%> {

    // methods
<iqgen:foreach group="<%=getMetaModel().getOperations(getElement())%>" item="method">
<%
    String methodName = getMetaModel().getName(method);
    Object retType = getMetaModel().getReturnType(method);
    String retTypeName = getMetaModel().getReturnTypeName(method);
    String parList = getMetaModel().getParameterListAsString(method);
%>
    /**
     * Method <%=methodName%>
     * <iqgen:foreach group="<%=getMetaModel().getParameterList(method)%>" item="parameter">
     * @param <%=getMetaModel().getName(parameter)%>
              <%=getMetaModel().getDocumentation(parameter)%>
      </iqgen:foreach><iqgen:if expr="<%=retType!=null%>">
     * @return <%=retTypeName%></iqgen:if>
     */
    public <%=retTypeName%> <%=methodName%>(<%=parList%>)
           <%=getMetaModel().getExceptionListAsString(method)%>;
</iqgen:foreach>
}
```

class.jsp

```
<%@include file="copyright.jsp" %>
<%--

Writes the class declarations and delegates
to included jsp for attributes, methods, and
associations.

$Id: class.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page import="java.util.Collection,
                 java.util.Iterator,
                 ru.novosoft.uml.MBase" extends="com.innoq.generator.jsp.JspBase" %>
<%@taglib uri="iqgen" prefix="iqgen" %>
<%@include file="params.jsp" %>
<%@include file="utilities.jsp" %>
/*
```

```
 * (#)<%=classname%>.java
 * Stereotype: <%=getMetaModel().getStereotype(getElement())%>
 *
 * Generated with iQgen template <iqgen:jspfilename/>
 */
package <%=getPath()%>;

<%-- imports --%>
<%--<iqgen:import prefix="<%=getPrefix()%>" collectiontype="java.util.Collection"/>--%>

<%-- protected area for user defined imports --%>
<iqgen:usercode id="imports">
// ...add additional imports here
</iqgen:usercode>

<%-- class declaration --%>
/**
 * Class <%=classname%>
 * <%=getMetaModel().getDocumentation(getElement())%>
 *
 * Last generation: <%=new java.util.Date()%>
 */
public class <%=classname%><iqgen:if expr="<%=superclazz!=null%>"> extends
                           <%=supername%> </iqgen:if><%
    Collection interfaces = getMetaModel().getInterfaces(getElement());
    String preOut = " implements ";
%><iqgen:foreach group="<%=interfaces%>" item="_interface">
<%=preOut+getMetaModel().getName(_interface)%><%
    preOut = ",";
%>
</iqgen:foreach> {

<%-- protected area for user defined attributes and methods --%>
    <iqgen:usercode id="global">
    // ...add your own attributes and methods here
    </iqgen:usercode>

<%-- attribute declarations --%>
<%@include file="attributes.jsp" %>
<%-- method declarations --%>
<%@include file="operations.jsp" %>
<%-- associations --%>
<%@include file="associations.jsp" %>
}

<%-- protected area for user defined classes --%>
<iqgen:usercode id="classes">
// ...add additional classes here
</iqgen:usercode>
```

attributes.jsp

```
<%--
```

```
Writes all attribute declarations and the
corresponding getter and setter methods.
$Id: attributes.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page import="java.util.Collection,ru.novosoft.uml.MBase,
                 com.innoq.generator.StringHelper"%>
<%
    Collection attr = getMetaModel().getAttributes(getElement());
%>
<%-- declare attributes --%>
<iqgen:foreach group="<%=attr.iterator()%>" item="attribute"><%
    String attrName = getMetaModel().getName(attribute);
    String attrType = getMetaModel().getTypeName(attribute);
%><iqgen:if expr="<%=attrName.length()!=0%>">
    // Membervariable of attribute <%=attrName%>
    private <%=attrType%> <%=StringHelper.fd(attrName)%>;
</iqgen:if>
</iqgen:foreach>
<% /* HK => */
/* Get ja set -metodit luodaan vain, jos luokan stereotyyppi on 'Persistent' */
String stereotyyppi = getMetaModel().getStereotype(getElement());
if (stereotyyppi.equals("Persistent") == true) {
/* <= HK */
%>
<%-- setter and getter for the attributes --%>
<iqgen:foreach group="<%=attr.iterator()%>" item="attribute"><%
    String attrName = getMetaModel().getName(attribute);
    String attrNameFU = StringHelper.fu(attrName);
    String attrType = getMetaModel().getTypeName(attribute);
    String instName = StringHelper.fd(attrName);
%><iqgen:if expr="<%=attrName.length()!=0%>">
    /**
     * Returns the attribute <%=attrName%>
     * <%=getMetaModel().getDocumentation(attribute)%>
     * @return Value of <%=attrName%>
     */
    public <%=attrType%> get<%=attrNameFU%>()  {
        <iqgen:usercode id="<%=\"get\"+attrNameFU%>">
        return <%=instName%>;
        </iqgen:usercode>
    }
<iqgen:if expr="<%=attrName.charAt(0)!='/'%>">
    /**
     * Sets the attribute <%=attrName%>
     * <%=getMetaModel().getDocumentation(attribute)%>
     * @param p<%=attrNameFU%> New value of attribute <%=attrName%>
     */
    public void set<%=attrNameFU%>(<%=attrType%> p<%=attrNameFU%>)  {
        <iqgen:usercode id="<%=\"set\"+attrNameFU%>">
        this.<%=instName%> = p<%=attrNameFU%>;
        </iqgen:usercode>
    }
</iqgen:if>
</iqgen:if>
```

```
</iqgen:foreach>
<%
} // getter ja setter metodien generoinnin loppu
%>
```

operations.jsp

```
<%--
Writes all declared methods with empty bodies.
$Id: operations.jsp,v 1.1.1.1 2002/12/11 08:10:52 ls Exp $
--%>
<%@ page import="ru.novosoft.uml.MBase,java.util.*,java.lang.*"%>
<iqgen:foreach group="<%=getMetaModel().getOperations(getElement())%>" item="method">
<%
    String retTypeName = getMetaModel().getReturnTypeName(method);
    MBase retType = getMetaModel().getReturnType(method);
    String parList = getMetaModel().getParameterListAsString(method);
    Collection pars = getMetaModel().getParameterList(method);
    String methodName = getMetaModel().getName(method);
    String visibility = getMetaModel().getVisibility(method);
    String documentation = "".equals(getMetaModel().getDocumentation(method)) ?
    "Method " + methodName : getMetaModel().getDocumentation(method);
%>
    /**
    * <%=documentation%>
    * <iqgen:foreach group="<%=getMetaModel().getParameterList(method)%>" item="parameter">
    * @param <%=getMetaModel().getName(parameter)%>
             <%=getMetaModel().getDocumentation(parameter)%>
    </iqgen:foreach><iqgen:if expr="<%=retType!=null%>">
    * @return <%=retTypeName%></iqgen:if>
    */
    <%=visibility%> <%=retTypeName%> <%=methodName%>(<%=parList%>)
    <%=getMetaModel().getExceptionListAsString(method)%>{
    <iqgen:usercode id="<%=methodName%>">

        // ...add your own code for method <%=methodName%> here

    </iqgen:usercode>
    } // <%=retTypeName%> <%=methodName%>()
</iqgen:foreach>
```

params.jsp

```
<%@page import="ru.novosoft.uml.MBase"%><%
    MBase clazz = getElement();
    String classname = getMetaModel().getName(clazz);
    String fqclassname = getMetaModel().getFQName(clazz);
    String packagename = getMetaModel().getNamespaceName(clazz);
    MBase superclazz =  getMetaModel().getSuperclass(clazz);
    String stereotype = getMetaModel().getStereotype(clazz);
    String supername = null;
```

```
    if (superclazz==null) {
        supername ="java.lang.Object";
    } else {
        supername = getMetaModel().getName(superclazz);
    }
%><%!
    public static String COLLECTION_TYPE="Collection";
    public static String COLLECTION_IMPL_TYPE="ArrayList";

    public String getPrefix(char pSeparator, MBase m)  {
        return "com"+pSeparator+"innoq"+pSeparator+"esimerkki"+pSeparator;
    }
%>
```

utilities.jsp

```
<%@page import="ru.novosoft.uml.MBase,
                ru.novosoft.uml.foundation.core.MParameter,java.util.*"%><%!
    public boolean isPrimitiveType(MBase element) {
        return isPrimitiveType(getMetaModel().getFQName(getMetaModel().getType(element)));
    }

    public boolean isPrimitiveType(String pTypename) {
        String s = pTypename;
        int p = s.lastIndexOf(".");
        if (p != -1) {
            s = s.substring(p + 1, s.length());
        }
        return (s.equals("int") ||
                s.equals("long") ||
                s.equals("char") ||
                s.equals("boolean") ||
                s.equals("double") ||
                s.equals("float") ||
                s.equals("byte") ||
                s.equals("short") ||
                s.equals("String") ||
                s.equals("NULL") || // used for"void"
                s.equals("void"));
    }

    public boolean isJavaUtilType(String pTypename) {
        String s = pTypename;
        int p = s.lastIndexOf(".");
        if (p != -1) {
            s = s.substring(p + 1, s.length());
        }
        return (s.equals("Collection") ||
                s.equals("Vector") ||
                s.equals("Enumeration") ||
                s.equals("List") ||
                s.equals("Set") ||
                s.equals("Map") ||
```

```
            s.equals("Hashtable") ||
            s.equals("HashMap") ||
            s.equals("Iterator"));
}

public String getReturnValue(MBase element) {
    String s = getName(element);
    if (isPrimitiveType(element)) {
        if (s.equals("int")) {
            return "0";
        } else if (s.equals("long")) {
            return "0L";
        } else if (s.equals("char")) {
            return "'_'";
        } else if (s.equals("boolean")) {
            return "true";
        } else if (s.equals("double")) {
            return "0.0";
        } else if (s.equals("float")) {
            return "0.0";
        } else if (s.equals("byte")) {
            return "0xff";
        } else if (s.equals("short")) {
            return "0";
        } else if (s.equals("String")) {
            return "\"\"";
        } else if (s.equals("NULL")) { // void
            return "/* " + s + " */";
        } else if (s.equals("VOID")) {
            return "/* " + s + " */";
        } else { // void
            return "null /* " + s + " */";
        }
    } else {
        return "null /* " + s + " */";
    }
} // end getReturnValue

private boolean isPrimaryKey(MBase m) {
    if (getMetaModel().isAttribute(m)) {
        String st = getMetaModel().getStereotype(m);
        return "PrimaryKey".equals(st);
    }
    return false;
}

private int getLength(MBase m) throws NumberFormatException {
    if (getMetaModel().isAttribute(m)) {
        String value = getMetaModel().getTaggedValueAsString
                            (m, "RationalRose$Java:Field length");
        try {
            return value == null ? 20 : Integer.parseInt(value);
        } catch (NumberFormatException nfe) {
            return 20;
```

```
            }
        }
        return 20;
    }

    public String createJavaModifier(MBase mo) {
        String ret = getMetaModel().getVisibility(mo);
        boolean isStatic = "True".equals(getMetaModel().getTaggedValueAsString
                                          (mo, "RationalRose$Java:Static"));
        boolean isFinal = "True".equals(getMetaModel().getTaggedValueAsString
                                          (mo, "RationalRose$Java:Final"));
        if (isStatic) {
            ret += " static";
        }
        if (isFinal) {
            ret += " final";
        }
        return ret;
    }
%>
```

copyright.jsp

```
/*
 * (C) 2001-2002 innoQ GmbH
 *
 * Hannu Kavakka 2004
 *
 * Esimerkki
 */
```

Liite 4: Näyte generoiduista SQL-lauseista

```
--
-- SQL-script of Varasto
-- Last generation: Mon Oct 11 18:44:02 EEST 2004
--
-- Generated with iQgen template /drop_and_create.jsp

drop table Varasto;

create table Varasto ( id VARCHAR(20) PRIMARY KEY ,
                       nimi VARCHAR(20) ,
                       osoite VARCHAR(20) ,
                       postinro VARCHAR(20) ,
                       postitoimipaikka VARCHAR(20) ,
                       gsm VARCHAR(20) ,
                       email VARCHAR(20) );
```

Liite 5: Näyte lähdekoodista

```
/*
 * (C) 2001-2002 innoQ GmbH
 *
 * Hannu Kavakka 2004
 * Esimerkki
 */

/*
 * (#)Varasto.java
 * Stereotype: Persistent
 *
 * Generated with iQgen template /class.jsp
 */

package com.innoq.esimerkki.data_Model;

//==> Begin Protected Area imports
import java.util.*;
//==> End Protected Area imports

/**
 * Class Varasto
 * Varasto säilyttää hetkellisesti tietoa sekä käyttää tietokantaa.
 *
 * Last generation: Mon Oct 11 18:44:02 EEST 2004
 */
public class Varasto extends Observable {

  //==> Begin Protected Area global
  private String tila;
  private TKyhteys tk;

  /**
   * Returns the attribute tila
   * @return Value of tila
   */
  public String getTila() {
    return tila;
  }

  /**
   * Sets the attribute tila
   * @param pTila New value of attribute tila
   */
  public void setTila(String pTila) {
    this.tila = pTila;
  }

  /**
   * Returns the attribute tk
   * @return Value of tk
```

```
 */
public TKyhteys getTk() {
  return tk;
}
//==> End Protected Area global

// Membervariable of attribute id
private String id;

// Membervariable of attribute nimi
private String nimi;

// Membervariable of attribute osoite
private String osoite;

// Membervariable of attribute postinro
private String postinro;

// Membervariable of attribute postitoimipaikka
private String postitoimipaikka;

// Membervariable of attribute gsm
private String gsm;

// Membervariable of attribute email
private String email;

/**
 * Returns the attribute id
 *
 * @return Value of id
 */
public String getId() {
  //==> Begin Protected Area getId
  return id;
  //==> End Protected Area getId
}

/**
 * Sets the attribute id
 *
 * @param pId New value of attribute id
 */
public void setId(String pId) {
  //==> Begin Protected Area setId
  this.id = pId;
  //==> End Protected Area setId
}

/**
 * Returns the attribute nimi
 *
 * @return Value of nimi
 */
```

```
public String getNimi() {
  //==> Begin Protected Area getNimi
  return nimi;
  //==> End Protected Area getNimi
}

/**
 * Sets the attribute nimi
 *
 * @param pNimi New value of attribute nimi
 */
public void setNimi(String pNimi) {
  //==> Begin Protected Area setNimi
  this.nimi = pNimi;
  //==> End Protected Area setNimi
}

/**
 * Returns the attribute osoite
 *
 * @return Value of osoite
 */
public String getOsoite() {
  //==> Begin Protected Area getOsoite
  return osoite;
  //==> End Protected Area getOsoite
}

/**
 * Sets the attribute osoite
 *
 * @param pOsoite New value of attribute osoite
 */
public void setOsoite(String pOsoite) {
  //==> Begin Protected Area setOsoite
  this.osoite = pOsoite;
  //==> End Protected Area setOsoite
}

/**
 * Returns the attribute postinro
 *
 * @return Value of postinro
 */
public String getPostinro() {
  //==> Begin Protected Area getPostinro
  return postinro;
  //==> End Protected Area getPostinro
}

/**
 * Sets the attribute postinro
 *
 * @param pPostinro New value of attribute postinro
```

```
 */
public void setPostinro(String pPostinro) {
  //==> Begin Protected Area setPostinro
  this.postinro = pPostinro;
  //==> End Protected Area setPostinro
}

/**
 * Returns the attribute postitoimipaikka
 *
 * @return Value of postitoimipaikka
 */
public String getPostitoimipaikka() {
  //==> Begin Protected Area getPostitoimipaikka
  return postitoimipaikka;
  //==> End Protected Area getPostitoimipaikka
}

/**
 * Sets the attribute postitoimipaikka
 *
 * @param pPostitoimipaikka New value of attribute postitoimipaikka
 */
public void setPostitoimipaikka(String pPostitoimipaikka) {
  //==> Begin Protected Area setPostitoimipaikka
  this.postitoimipaikka = pPostitoimipaikka;
  //==> End Protected Area setPostitoimipaikka
}

/**
 * Returns the attribute gsm
 *
 * @return Value of gsm
 */
public String getGsm() {
  //==> Begin Protected Area getGsm
  return gsm;
  //==> End Protected Area getGsm
}

/**
 * Sets the attribute gsm
 *
 * @param pGsm New value of attribute gsm
 */
public void setGsm(String pGsm) {
  //==> Begin Protected Area setGsm
  this.gsm = pGsm;
  //==> End Protected Area setGsm
}

/**
 * Returns the attribute email
 *
```

```
 * @return Value of email
 */
public String getEmail() {
  //==> Begin Protected Area getEmail
  return email;
  //==> End Protected Area getEmail
}

/**
 * Sets the attribute email
 *
 * @param pEmail New value of attribute email
 */
public void setEmail(String pEmail) {
  //==> Begin Protected Area setEmail
  this.email = pEmail;
  //==> End Protected Area setEmail
}

/**
 * Luokan rakentaja alustaa muuttujat sekä muodostaa tietokantayhteyden.
 */
public Varasto() {
  //==> Begin Protected Area Varasto
  id = "";
  nimi = "";
  osoite = "";
  postinro = "";
  postitoimipaikka = "";
  gsm = "";
  email = "";

  tk = new TKyhteys();
  boolean ok = tk.avaaYhteys();

  if (!ok) {
    tila = "Tietokantayhteyttä ei voitu muodostaa.";
  }
  else {
    tila = "Tietokantayhteys muodostettu.";
  }
  //==> End Protected Area Varasto
} //   Varasto()

/**
 * Lisää Näkemä -tyyppisen olion Varaston tarkkailijaksi.
 *
 * @param view Nakema
 */
public void lisaaNakema(Nakema view) {
  //==> Begin Protected Area lisaaNakema
  addObserver( (Observer) view);
  //==> End Protected Area lisaaNakema
} // void lisaaNakema()
```

```
/**
 * Poistaa Näkemä -tyyppisen olion tarkkailtavien joukosta.
 *
 * @param view Nakema
 */
public void poistaNakema(Nakema view) {
  //==> Begin Protected Area poistaNakema
  deleteObserver( (Observer) view);
  //==> End Protected Area poistaNakema
} // void poistaNakema()

/**
 * Ilmoittaa tarkkailijoille muutoksesta.
 */
public void ilmoitaNakemille() {
  //==> Begin Protected Area ilmoitaNakemille
  setChanged(); // Ilmoitetaan muutoksesta
  // Ilmoitetaan tarkkailijoille muutoksesta kutsumalla
  // niihin toteutettuja update -metodia
  notifyObservers();
  //==> End Protected Area ilmoitaNakemille
} // void ilmoitaNakemille()

/**
 * Pyrkii lisäämään tiedot tietokantaan.
 *
 * @param id String
 * @param nimi String
 * @param osoite String
 * @param postinro String
 * @param postitoimipaikka String
 * @param gsm String
 * @param email String
 */
public void tallenna(String id, String nimi, String osoite, String postinro,
                     String postitoimipaikka, String gsm, String email) {
  //==> Begin Protected Area tallenna
  // Yritetään tallentaa tiedot tietokantaan.
  boolean ok = tk.lisaaTiedot(id, nimi, osoite, postinro, postitoimipaikka,
                              gsm, email);

  if (ok) {
    this.id = id;
    this.nimi = nimi;
    this.osoite = osoite;
    this.postinro = postinro;
    this.postitoimipaikka = postitoimipaikka;
    this.gsm = gsm;
    this.email = email;
    tila = "Tallennettu";
  }
  else {
    this.id = "";
```

```
      this.nimi = "";
      this.osoite = "";
      this.postinro = "";
      this.postitoimipaikka = "";
      this.gsm = "";
      this.email = "";
      tila = "Virhe tallennettaessa!";
    }
    ilmoitaNakemille();
    //==> End Protected Area tallenna
  } // void tallenna()

  /**
   * Pyrkii hakemaan tiedon tietokannasta.
   *
   * @param id String
   */
  public void hae(String id) {
    //==> Begin Protected Area hae
    String[] tiedot = null;
    tiedot = tk.haeTiedot(id);

    if (tiedot != null) {
      this.id = tiedot[0];
      this.nimi = tiedot[1];
      this.osoite = tiedot[2];
      this.postinro = tiedot[3];
      this.postitoimipaikka = tiedot[4];
      this.gsm = tiedot[5];
      this.email = tiedot[6];
      tila = "Haettu";
    }
    else {
      this.id = "";
      this.nimi = "";
      this.osoite = "";
      this.postinro = "";
      this.postitoimipaikka = "";
      this.gsm = "";
      this.email = "";
      tila = "Ei löydy";
    }
    ilmoitaNakemille();
    //==> End Protected Area hae
  } // void hae()
}

//==> Begin Protected Area classes
// ...add additional classes here
//==> End Protected Area classes
```