T A R T U   Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Tarkvarasüsteemide õppetool
Informaatika eriala

Andres Vilgota

Mudelipõhine tarkvaraarendus

Bakalaureusetöö

Juhendaja: Jaak Vilo, PhD

Autor:.. “.....“ 2004
Juhendaja:..................................... “.....“ 2004
Õppetooli juhataja:......................... “.....“ 2004

TARTU 2004

Sisukord

LÜHENDID **4**

SISSEJUHATUS **5**

ÜLESANDEPÜSTITUS **6**

1. TRADITSIOONILINE TARKVARAARENDUS **7**

1.1. DOKUMENTATSIOONI PROBLEEM 7

1.2. PRODUKTIIVSUSE PROBLEEM 8

1.3. PLATVORMIDE ARENEMISE PROBLEEM (*PORTABILITY PROBLEM*) 8

1.4. PLATVORMIDEVAHELINE PROBLEEM (*INTEROPERABILITY PROBLEM)* 9

1.5. REKODEERIMISE PROBLEEM (*REFACTORING PROBLEM*) 10

1.6. LIHTSATE MUUDATUSTE PROBLEEM 10

2. NÄIDE: MUDELI PÕHJAL INFOSÜSTEEMI GENEREERIMINE **12**

2.1. OBJEKT-RELATSIOONILINE ANDMEBAASISÜSTEEM 12

2.1.1 Objekt-relatsioonilised andmeteisendajad *13*

2.1.2. Umbrello UML Modeller *16*

2.1.3. Infosüsteemi loomine *16*

2.2. PROGRAMM U-CODER 18

2.2.1. Klassidiagrammist andmebaasiskeemini *20*

2.2.2. Transformatsiooni parameetrid. *21*

3. MUDELIL PÕHINEV ARHITEKTUUR **22**

3.1. MDA PÕHIALUSED 24

3.1.1. Mudel *24*

3.1.2. Platvorm *24*

3.1.3. PIM – platvormisõltumatu mudel *25*

3.1.4. PSM – platvormispetsiifiline mudel *25*

3.1.5. Lähtekood *25*

3.1.6. Automaatsed mudelite transformatsioonid *26*

3.2. MDA EELISED 28

3.2.1. Produktiivsus *28*

3.2.2. Platvormide arenemine *28*

3.3.3. Platvormidevaheline sõltuvus *29*

3.2.4. Abstraktsus *29*

3.3. MDA PUUDUSED: UML KUI PIM KEEL 31

4. OMG STANDARDID **33**

4.1. UML (*UNIFIED MODELING LANGUAGE*) 33

4.2. OBJEKTITÕKKEKEEL (*OBJECT CONSTRAINT LANGUAGE, OCL*) 33

4.2.1. OCL kasutamine ... *34*
4.3. CWM (*Common Warehouse Metamodel*) ... 36
4.4. MOF (*Meta Object Facility*) ... 36
4.5. XMI (*XML Metadata Interchange*) ... 36
4.6. Metamodelleerimine ... 37
5. MUDELITE TRANSFORMATSIOONID ... 39
5.1. Transformatsiooni keeled ... 40
5.1.1. (QVT) Query/Views/Transformations ... *41*
5.2. Transformatsioonid ... 42
5.3. Transformatsioonide taaskasutamine ... 42
5.4. Transformatsioonide järk-järguline rakendamine. ... 43
5.5. MDA arengusuunad ... 44
KOKKUVÕTTEKS ... 46
MODEL-DRIVEN SOFTWARE DEVELOPMENT ... 47
VIITED ... 48
LISAD ... 50
Lisa 1. Näide U-Coderi tüübiteisenduse XML failist ... 50

Lühendid

AS	(UML) Action Semantics
AOP	Aspect Oriented Programming
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JDO	Java Data Objects
MDA	Model Driven Architecture
MDD	Model Driven Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OQL	Object Query Language
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/Views/Transformations
RTE	Round-Trip Engineering
SQL	Structured Query Language
TXL	Tree Transformation Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XPath	XML Path Language
XSL	Extensible Stylesheet Language

Sissejuhatus

Traditsioonilise tarkvaraarenduse fookuses on koodi kirjutamine erinevates programmeerimiskeeltes. Ühe ärilahenduse loomisel kirjutatakse koodi mitme tehnoloogilise platvormi jaoks korraga. Näiteks tüüpilise infosüsteemi loomine kaasab nii XML, XSL kui ka HTML tehnoloogiat. Erinevatel platvormidel ja tehnoloogiatel on süsteemi töös spetsiifilised rollid, kuid neile kirjutatud kood kirjeldab ikkagi samu kontseptsioone.

Traditsioonilisel tarkvaraarendusel on mitmeid puudusi ning paljud projektid ebaõnnestuvad. Üha enam hakkab populaarsust koguma *mudelil põhinev tarkvaraarendus* (ingl. k. *Model Driven Development*, edaspidises lühendatult MDD), mis lahendab mitmed traditsioonilise tarkvaraarenduse probleemid. Selleks, et soodustada kasulike ja koostöövõimeliste MDD tööriistade kasutamist, on OMG (Object Managenment Group) arendamas vastavat standardite komplekti koondnimetusega MDA (*Model Driven Architecture*) [MDA]. Lihtsustatult võib öelda, et MDA eesmärgiks on kodeerimise asendamine kõrgema abstraktsustasemega mudelite koostamise ja transformeerimisega.

MDA arendusprotsessi fookus on suunatud süsteemi funktsionaalsele ja käitumuslikule aspektile, jättes teisejärguliseks lahenduse tehnoloogilised küljed. Selle asemel, et tegeleda äriloogika sobitamisega konkreetsesse keele süntaksisse arendatakse MDA protsessi korral võimalikult palju loogikat platvormist sõltumatusse mudelisse (ingl. k. *Platform Independent Model*, edaspidises PIM).

Kui me suudame kogu süsteemi, st tema klassid, seosed klasside vahel ja loogika, mis süsteemi elemente ühendab kirja panna platvormist sõltumatul kujul, siis selle põhjal saab luua selle süsteemi kirjelduse mistahes platvormi jaoks (Java, andmebaasiesitus, XML-esitus jne). Vastavat konkreetse platvormi jaoks loodud esitust nimetatakse platvormispetsiifiliseks mudeliks (ingl. k. *Platform Specific Model*, PSM), mis on praktiliselt üks-üheselt teisendatav allasuva platvormi koodiks (Java mudel lähtekoodiks, andmebaasimudel SQL-lauseteks jne).

Traditsiooniliselt tehakse mudelite teisendused käsitsi, kontseptuaalse mudeli põhjal luuakse andmemudel ja Java klassimudel. MDA raamistikus on mudelite teisendused maksimaalselt automatiseeritud.

Ülesandepüstitus

Käesoleva töö eesmärgiks on uurida mudelipõhist tarkvaraarendust, anda lugejale ülevaade vastavatest kontseptsioonidest ja demonstreerida mudelipõhise tarkvaraarenduse reaalset rakendamist. Töös tahame anda ülevaate MDD eelistest võrreldes traditsioonilise tarkvaraarendusega. Konkreetse näiterakenduse varal näitame, milliseid eelisi annab mudelipõhine tarkvaraarendus ning millised probleemid selle kasutamisega võivad tekkida.

Praktilisemast küljest on eesmärgiks tutvustada lugejale MDD jaoks OMG (Object Management Group) poolt loodud standardeid ning anda ülevaade nende poolt arendatavast MDD teostusest: MDA protsessist. Et olemasolevatest standarditest töö kirjutamise hetkel veel ei piisa mudelitel põhineva arhitektuuri täielikuks rakendamiseks, siis anname ka ülevaate MDA puuduvatest lülidest, eelkõige mudelite transformatsioonide alal.

Töö motivatsiooniks on saavutada praktilised teadmised infosüsteemide mudelipõhisest genereerimisest ning mudelite transformatsioonidest OMG standardite eeldatud tasemel.

1. Traditsiooniline tarkvaraarendus

Käesolev peatükk kirjeldab traditsioonilist tarkvaraarendust ja sellega kaasnevaid probleeme, mis on tuttavad pea igale programmeerijale.

Tüüpiline arendusprotsess tellimusest kuni tellija vajadusi rahuldava rakenduse valmimiseni sisaldab endas järgmisi etappe:

1. Kontseptuaalne analüüs ja nõuete analüüs
2. Analüüs ja funktsionaalsuse kirjeldamine
3. Disain
4. Kodeerimine
5. Testimine
6. Juurutamine

Traditsioonilise tarkvaraarenduse fookuses on disain ja kodeerimine [KWB03]. Lähenemisi on mitmeid – inkrementaalseid ja iteratiivsed meetodid, koskmudel ja ekstreemprogrammeerimine, jne. Kõik nad sisaldavad endas suuremal või vähemal määral eelpoolnimetatud etappe.

Projektid erinevad ajaliste nõuete, mahu, kvaliteedinõuete jne poolest. Erinevate projektide jaoks on sobilik kasutada erinevaid meetodeid – ühe projekti jaoks sobilik meetod ei pruugi olla efektiivne teise projekti jaoks, sest meetodid on orienteeritud erinevatele tulemustele. Seega võib ühte meetodit eelistada teistele, kõik sõltub vajadustest. Samas paistab, et edukus ühes valdkonnas tähendab paratamatult kehvemat tulemust teises valdkonnas. Millised aga on tarkvaraarenduse probleemid üldisemalt?

1.1. Dokumentatsiooni probleem

Sõna „tarkvara“ võib tähistada sadu tuhandeid ridu koodi. Dokumentatsioon on enamasti kohustuslik, nii teiste programmeerijate kui ka tellijate jaoks. Isegi kui arenduse kestel on arendajate grupil programmist selge ülevaade, siis hiljem ununeb palju ja kiiresti. Kuigi süsteemi kontseptuaalne mudel on programmeerijate peades olemas, jäetakse dokumentatsiooni kirjutamine ja mudelite ning jooniste taasjoonistamine ajapuuduse tõttu tihtipeale viimaseks etapiks. Programmeerijad tunnevad, et nende

peamine ülesanne on kodeerimine, dokumentatsiooni kirjutamine on teisejärguline. Ometigi on dokumentatsiooni kirjutamine vajalik, kasvõi hilisemate muudatuste sisseviimiseks. Dokumentatsiooni probleemi vältimiseks on loodud võimalusi koodi tasemel dokumentatsiooni kirjutamiseks – näiteks Java-s saab kirjutada javadoc-stiilis kommentaare. Ometigi ei ole selline lahendus täielik, keeruliste süsteemide juures on kindlasti tarvis ka kõrgema abstraktsioonitasemega dokumentatsiooni, mis oleks sobilik esitada ka tellijatele, juhtidele ja eelkõige neile, kes peavad hakkama programmi haldama. Selline dokumentatsioon tuleb eraldi lisaks kirjutada.

1.2. Produktiivsuse probleem

Enne kodeerimist, analüüsi ja disaini etappide käigus luuakse süsteemist palju kirjeldusi, tekib nägemus tarkvarast ja selle tööst visandite, tekstide ja ka UML diagrammide näol. Mudeleid muudetakse: parandatakse, täiendatakse. Vahel eemaldatakse suur osa ja alustatakse taas algusest – niiviisi võib tekkida aukartustäratav hulk kasutuid mudeleid ja diagramme, aegunud dokumentatsiooni.

Mis peamine, tihtipeale jäävadki mudelid niisama visanditeks, neid ei kasutata muuks kui programmeerijale tema töö kättenäitamiseks. Sel juhul aga jäävad diagrammid kodeerimise arenedes lihtsalt pildikesteks, mis kunagi võibolla isegi olid koodiga kooskõlas, kuid hiljem enam mitte. Kuna iga muudatuse tegemine koodis tähendab potentsiaalset muudatust dokumentatsioonis ja mudelites, siis programmeerija ei taha ega suudagi dokumentatsiooni kirjutada.

Iseenesest ei ole midagi halba selles, kui programmeerija on produktiivne ja kirjutab palju koodi. Probleemiks on see, et koostatud mudelid oleks heaks lisaks dokumentatsioonile, aga olles fokuseeritud koodile jääb see võimalus kasutamata.

1.3. Platvormide arenemise probleem (*portability problem*)

Üha kiiremini luuakse uusi tehnoloogiaid ja platvorme (näiteks XML, Java, SOAP, J2EE, .NET, Flash, jne). Selleks, et säilitada konkurentsivõime ja edestada konkurente tuleb uusi tehnoloogiaid kiiresti õppida ja varakult kasutama hakata. Vahel tuleb uusi

tehnoloogiad hakata kasutama ka seepärast, et vanadele ei leidu enam piisavalt toetust, näiteks on objektmodelleerimise tehnika asendunud modelleerimisega UML keeles.

Süsteemi üleviimine uuele tehnoloogiale võib olla lihtne, aga võib olla ka vägagi keerukas. Platvormide paljusus nõuab sügavaid teadmisi kõigist kasutatavatest keeltest ja protokollidest. Halvimal juhul tuleb uue platvormi jaoks kogu funktsionaalsus uuesti kirja panna, kui aga esialgset programmi koostades on kogu tähelepanu olnud kodeerimisel, siis tekivad küsimused – milline funktsionaalsus on universaalne ning milline osa koodist on pelgalt selle funktsionaalsuse teostus vastava programmeerimiskeele vahenditega?

1.4. Platvormidevaheline probleem (*interoperability problem)*

Tarkvarasüsteemid ehitatakse peaaegu alati kasutades paljusid vahendeid korraga – näiteks ühe lihtsa veebiprojekti jaoks kasutatakse HTML tehnoloogiat kasutajaliidese veebibrauseris näitamiseks, XML ja XSL tehnoloogiat andmete esitamise hõlbustamiseks ning relatsioonilist andmebaasi andmete salvestamiseks. Tervikliku süsteemi tekkimiseks on tarvis erinevate aspektidega tegelevate platvormide vahele ehitada sillad. Tüüpiliseks näiteks sellisest sillast on andmete esitluskihi ja salvestuskihi vahel asuv objekt-relatsiooniline andmeteisendaja. Selline sild peab olema teadlik süsteemi esitusest mõlemas platvormis, seega tuleb konkreetsete platvormide vahele loodud sillad ühe või teise platvormi muutudes ümber kodeerida.

Lisaks sildade tundlikkusele süsteemi funktsionaalsuse muutuse suhtes ühes või teises platvormis tuleb sama arvestada ka platvormide arenemisega. Üks rakendus hõlmab paratamatult mitmeid tehnoloogiaid, mis võivad aja jooksul areneda (näiteks HTML standardid) või asenduda hoopis uuega (näiteks JDBC abil süsteemi sisseehitatud andmebaasiühendus objekt-relatsioonilise andmeteisendajaga). Tarkvara töö jätkamiseks uute tehnoloogiatega tuleb konstrueerida uued sillad, mis on teadlikud mõlemast platvormist. Näiteks peab XML dokumente teisendav XSLT skript teadma nii lähte- kui ka tulemusdokumendi täpset formaati.

1.5. Rekodeerimise probleem (*refactoring problem*)

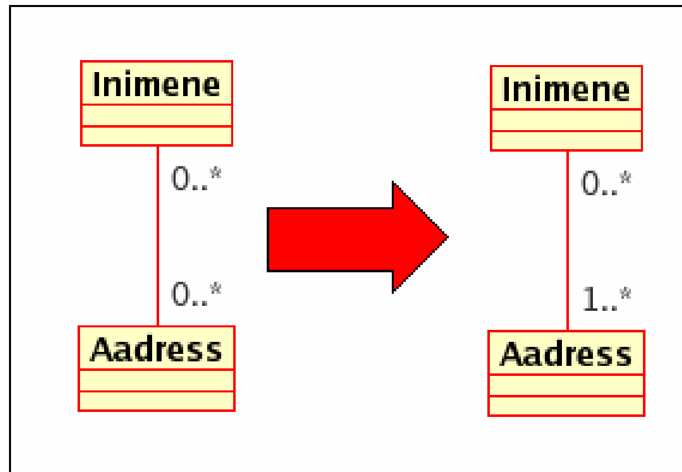
Rekodeerimise eesmärk on koodi parandamine sellisel viisil, et funktsionaalsus säiliks aga koodi kvaliteet tõuseks. Hea kvaliteediga koodi on funktsionaalsuse lisamiseks lihtne muuta, samuti on vigade leidmine ja parandamine võimalikult valutu.

Rekodeerimise omaduseks on tegelikult väljendumine rohkem disaini kui koodi tasemel, näiteks klassi ümbernimetamiseks disaini tasemel on vaja teha üks pisike liigutus, koodi tasemel aga tähendab see klassi nime muutmist kõikjal, kus seda klassi kasutatakse. Rekodeerimise tegemiseks on olemas mitmeid häid abivahendeid (RefactorIT, Eclipse, jt.), ometigi võib nende kasutamine suurte projektide korral osutuda küllaltki raskeks, sest nad tegutsevad koodi tasemel: näiteks klassi nime muutmist suudavad nad automaatselt teostada vaid Java lähtekoodis, kui aga sama kontseptsioon esineb ka andmebaasitabelites, dokumentatsioonis, XML schema-des ja mujal, siis on nendest tõusev tulu suhteliselt väike.

1.6. Lihtsate muudatuste probleem

Tarkvaraarenduse ainuke muutumatu vajadus on pideva muutmise vajadus („The only constant in software develoment is change“). Muudatuste põhjuseks võivad olla planeeritud arendustööd, ootamatute probleemide lahendamine või programmi struktuuri parandamise vajadus (rekodeerimine). Nägime, et rekodeerimise eesmärgiks on ilma funktsionaalsust muutmata saavutada olukord, kus funktsionaalsust oleks lihtne lisada ning muuta.

Oletame, et oleme oma süsteemi piisavalt korralikult rekodeerinud ning soovime seda nüüd muuta. Näiteks tahame muuta seose klassiga *Aadress* kohustuslikuks iga *Inimene* jaoks (joonis 1.1).



Joonis 1.1. Lihtne muudatus disainis on suurem koodis.

Säärane muudatus peab kajastuma kõikjal süsteemis – andmebaasis *not null* kitsenduse näol, veebiliideses sisestusvormi täitmisel veateate ilmumisega jne. Seega lihtne muudatus disaini tasemel tähendab suurt hulka spetsialistide tööd.

Analoogseid näiteid võib tuua veelgi, näiteks infosüsteemi mitmekeelseks tegemine on kasutaja poolt vaadatuna triviaalne – kasutaja tahaks lihtsalt klassidiagrammis iga atribuudi juures määrata, kas vastav atribuut on keeletundlik või mitte. Teisalt, vastava süsteemi reaalne loomine tähendab päris mitut tegevust. Mõelgem kasvõi ainult andmebaasipäringutele: iga selline klass on andmebaasis esitatud juba kahe tabelina, kus lisatabelis hoitakse keelesõltuvaid väljasid koos keele identifikaatoriga. Objekt-relatsioonilise andmeteisendaja kasutamisel peab eriline olema ka teisenduskirjeldus.

Kokkuvõtteks võib öelda, et tarkvara arendamise käigus tehakse disainis palju pealtnäha lihtsaid muudatusi, millede realiseerimine süsteemis on märkimisväärselt keerukam. Paneme tähele, et tegelikult on ju nn lihtsatele probleemidele olemas tuntud ja tunnustatud lahendused, aga meil ei ole vahendeid nende lahenduste automaatseks realiseerimiseks. Seega peab programmeerija tegema tegelikult automatiseeritavat tööd.

Eelnevast nägime, et traditsioonilisel tarkvaraarendusel on mitmeid puudusi. Loomulikult on lugejale juba selge, et eelnevas „traditsiooniliseks“ nimetatud arendusmeetodi kõrval peab olema alternatiive. Selliseks alternatiiviks on mudelipõhine tarkvaraarendus, mille abil on loomulikul viisil võimalik lahendada enamik traditsioonilise tarkvaraarenduse probleeme.

2. Näide: Mudeli põhjal infosüsteemi genereerimine

Käesolevas peatükis näitame, et mudeli põhjal süsteemi genereerimine võib meid säästa suurest hulgast tööst. Tutvustame võimalust infosüsteemi kiireks loomiseks ühe praktilise näite varal: vaatleme klassimudeli põhjal relatsioonilisele andmebaasile objektivaate loomist. Näide põhineb sügisel 2003 Andres Vilgota poolt koostatud programmil U-Coder [Vil03]. Esmalt tutvustame lugejat objekt-relatsioonilise andmebaasiga, seejärel näitame kuidas saab selle loomist automatiseerida.

2.1. Objekt-relatsiooniline andmebaasisüsteem

Infosüsteemi üheks oluliseks osaks on andmebaas. Andmete hoidmiseks on kasutatud ja kasutatakse ka tulevikus *relatsioonilisi* andmebaase, eelkõige nende laialdase leviku, usaldusväärsuse ja efektiivse relatsioonialgebra tõttu.

Kui infosüsteemi juhtimisloogika on kirjutatud mõnes objekt-orienteeritud keeles (Java-s), siis andmetega töötamise jaoks tuleb paratamatult luua vastavad andmeobjektid, olgu algandmed mistahes vormingus. Seega objekt-orienteeritud rakenduse korral oleks mugav kasutada *objekt-orienteeritud* andmebaasi, kui aga soovitakse kindlaks jääda relatsioonilisele andmebaasile, siis on mõistlik kasutada *objekt-relatsioonilist* andmebaasi.

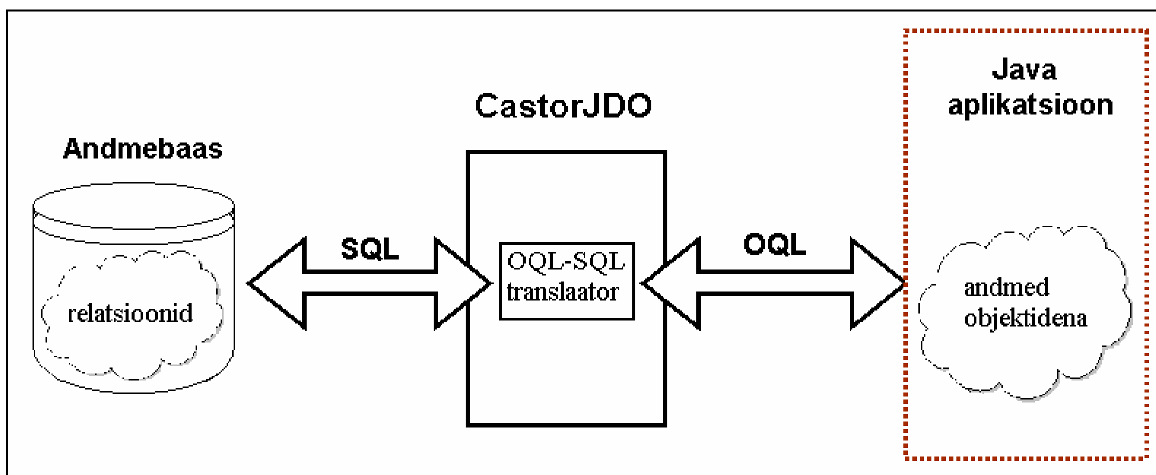
Objekt-relatsioonilise andmebaasi korral hoitakse andmeid relatsioonides milledele on ehitatud objektivaade. On palju tootjaid, kelle andmebaasisüsteemid ongi objekt-relatsioonilised, näiteks PostgreSQL, Oracle9i, Caché, FirstSQL, jne. [SOA]. Nendes on relatsioonid ja objektivaate kiht ühendatud kinnisesse süsteemi, ning väljastpoolt vaadatuna on tegu objekt-orienteeritud andmebaasiga.

Alternatiivina kinnistele lahendustele võib objektivaate saamiseks kasutada spetsiaalset objekt-relatsioonilist andmeteisendajat. Sisuliselt saame tulemuseks samuti objekt-orienteeritud andmebaasi. Relatsioonilise baasi eraldihoidmine objektivaate loomise kihist annab paindlikumad võimalused andmetega töötamiseks – vajadusel saame kasutada kogu SQL võimekust (kiire), üldjuhul aga saame kasutada objekte (aeglasem, kuid mugavam).

2.1.1 Objekt-relatsioonilised andmeteisendajad

Objekt-orienteeritud mõttemaailm ja relatsiooniline maailm ei ole kuigi lähedased. Teisendused objektivaate ja relatsioonilise vaate vahel pole lihtsad, näiteks tuleb relatsioonilises baasis kuidagi võimaldada objekt-orienteeritud maailmas nii tavapärast nähtust nagu pärimine. Ometigi on sellised teisendused olemas, teisenduste eest vastutab objekt-relatsiooniline andmeteisendaja, näiteks Castor.

Castor [Castor] on vabavaraline andmete sidumise raamistik (ingl. k. *data binding framework*) Java jaoks. Castorit arendatakse kahes suunas: CastorJDO tegeleb relatsioonilise andmebaasi ja Java vahelise sidumisega, CastorXML aga Java objektide ja XML vahelise sidumisega. Joonisel 2.1 on kujutatud objekt-relatsiooniline andmebaasisüsteem kasutades objekt-relatsioonilist andmeteisendajat Castor:

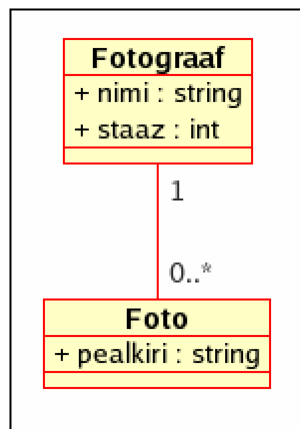


Joonis 2.1. O-R andmeteisendaja Castor põhimõtteskeem.

Kasutades objekt-relatsioonilist andmeteisendajat ei pea enam ise kirjutama SQL päringuid, mis relatsioonidest andmeobjekte looksid. Selle asemel saame kasutada palju mugavamat objektpäringukeelt – OQL-i (ingl. k. *Object Query Language*). Castor vastab OQL päringule objektide või objektivõrkude tagastamisega..

Teisenduste tegemisel relatsioonide ja objektide vahel kasutab Castor XML-põhist teisenduskirjelduse faili. Ühes ja samas teisenduskirjelduse failis saab määrata objektide relatsioonilise kuju (CastorJDO jaoks) ja ka XML-kuju (CastorXML jaoks). Castori eripäraks tulekski lugeda asjaolu, et ta suudab ühe teisenduskirjelduse faili põhjal Java objekte esitada nii relatsioonilises andmebaasis kui ka XML dokumentidena, paljud teised lahendused tegelevad ainult ühega neist korraga.

Vaatleme näidet fotokonkursside andmebaasist, joonisel 2.2 on kujutatud väike osa suuremast klassidiagrammist, joonis kujutab üks-mitmele seosest klasside Fotograaf ja Foto vahel:



Joonis 2.2. Lihtne näide klassidiagrammi osast.

Sellisele mudelile vastab kahe tabeliga andmebaas, kus tabelis `Fotograaf` on väljad `id`, `nimi` ning `staaz`, tabelis `Foto` on väljad `id`, `pealkiri` ning `fotograaf_id`. Paneme tähele, et kontseptuaalses mudelis ei ole kujutatud andmebaasi võtmeväljasid (`id` väljad ja `fotograaf_id väli`). ID-väljad on objektide andmebaasiidentifikaatorid, neid kasutatakse klassidevaheliste seoste kujutamiseks andmebaasis. Ka igas Java objektis peab vastav väli eksisteerima.

Joonisel 2.2 kujutatud klasside ja vastava andmebaasi vahel on järgmine objekt-relatsiooniline andmeteisendus:

```
<mapping>
  <class name="Fotograaf" identity="id" key-generator="MAX">
    <map-to table="fotograaf"/>
    <field name="id" type="integer">
      <sql name="id" type="integer"/>
    </field>
    <field name="nimi" type="string">
      <sql name="nimi" type="varchar"/>
    </field>
    <field name="staaz" type="integer">
      <sql name="staaz" type="integer"/>
    </field>
    <field name="fotod" type="Foto" collection="vector">
      <sql name="id" many-key="fotograaf_id" many-table="foto"/>
    </field>
  </class>
  <class name="Foto" identity="id" key-generator="MAX">
    <map-to table="foto"/>
    <field name="id" type="integer">
      <sql name="id" type="integer"/>
    </field>
    <field name="pealkiri" type="string">
      <sql name="pealkiri" type="varchar"/>
    </field>
    <field name="fotograaf" type="Fotograaf">
      <sql name="fotograaf_id"/>
    </field>
  </class>
</mapping>
```

Selline teisenduskirjeldus on CastorJDO jaoks piisav, et võimaldada andmete objektkuju teisendamist relatsioonilisele kujule ja vastupidi. Teisenduskirjelduses oleva info põhjal tõlgib Castor kasutajapoolsed objektpäringukeele (OQL) päringud SQL päringuteks.

Loomulikult ei ole Castor ainuke objekt-relatsiooniline andmeteisendaja, näiteks Hibernate [Hibernate] on samuti vabavaraline, populaarne ja põhjalik tööriist relatsioonilise andmebaasi ja Java sidumiseks. Ka Hibernate kasutab analoogilist XML-formaadis andmeteisendusfaili.

2.1.2. *Umbrello UML Modeller*

Käesolevas töös olevad mudelid on joonistatud kasutades modelleerimisvahendit Umbrello UML Modeller. Umbrello on vabavaraline ja avatud lähtekoodiga UML diagrammide joonistamise programm KDE keskkonna jaoks. Alates KDE versioonist 3.2 kuulub ta ka KDE standardpaketti. Umbrello on lihtsasti kasutatav, ei nõua arvutilt palju ressurssi (mida teevad mitmed teised, nt Poseidon, ArgoUML) ja mis peamine, ei nõua ressurssi rahakotis (Rational Rose vahendite hinnad algavad tuhandetest dollaritest).

Umbrello kasutab mudelite salvestamiseks XML-põhist XMI formaati (XMI formaadist räägime lähemalt peatükis 4.5), eelnevas toodud näitemudel fotode ja fotograafidega (joonis 2.2) näeb XMI dokumendina lihtsustatud kujul välja järgmine:

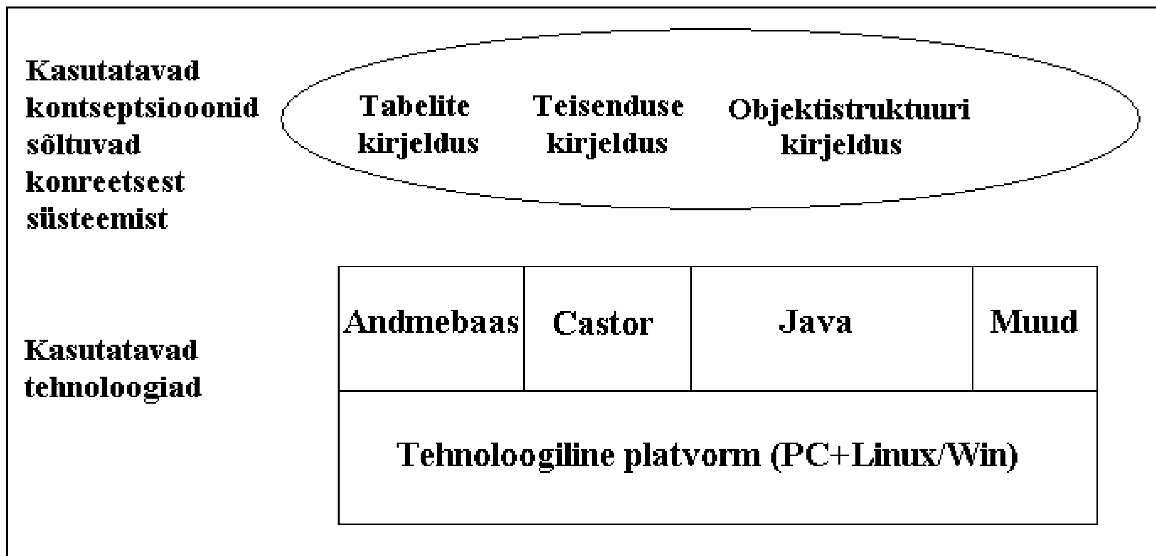
```
<?xml version="1.0" encoding="UTF-8"?>
<UML:Model>
  <UML:Class xmi.id="1" name="Fotograaf" >
    <UML:Attribute xmi.id="3" type="string" name="nimi" />
    <UML:Attribute xmi.id="4" type="int" name="staaz"/>
   </UML:Class>
   <UML:Class xmi.id="2" name="Foto">
    <UML:Attribute xmi.id="5" type="string" name="pealkiri"/>
   </UML:Class>
   <UML:Association rolea="1" roleb="2" multia="1" multib="0..*"/>
</UML:Model>
```

XMI on standardne formaat mudelite vahendamiseks UML tööriistade vahel. Tuleb mainida, et nimetatud standard ei ole hästi järgitav, selles on teatavaid võimalusi mitmetimõistmiseks. Seetõttu pole tihtipeale ühe modelleerimisvahendiga koostatud mudelid teisega avatavad.

2.1.3. Infosüsteemi loomine

Infosüsteemi loomine ei ole lihtne. Hea süsteemi saamiseks konsulteeritakse erialaspetsialistidega, näiteks perearsti infosüsteemi loomiseks tehakse tihedat koostööd perearstidega. Arendajate peaeesmärgiks on luua kliendi vajadusi rahuldav süsteem, mis muuhulgas tähendab konkreetse valdkonna kontseptsioone esitavate andmeobjektide loomist. Vastavad kontseptsioonid on läbivaks jooneks terves infosüsteemis.

Iga andmete esitusviisi loomiseks on tarvis spetsiifilisi oskusi. Nii loob andmebaasi just andmebaasiprogrammeerija ning veebiliidese veebimeister. Ometigi on selge, et erinevate spetsialistide eesmärgiks on ühtede ja samade kontseptsioonide esitamine. Infosüsteemi spetsiifilised osad, mis traditsiooniliselt tuleb iga ülesande korral käsitsi luua on toodud joonisel 2.3.

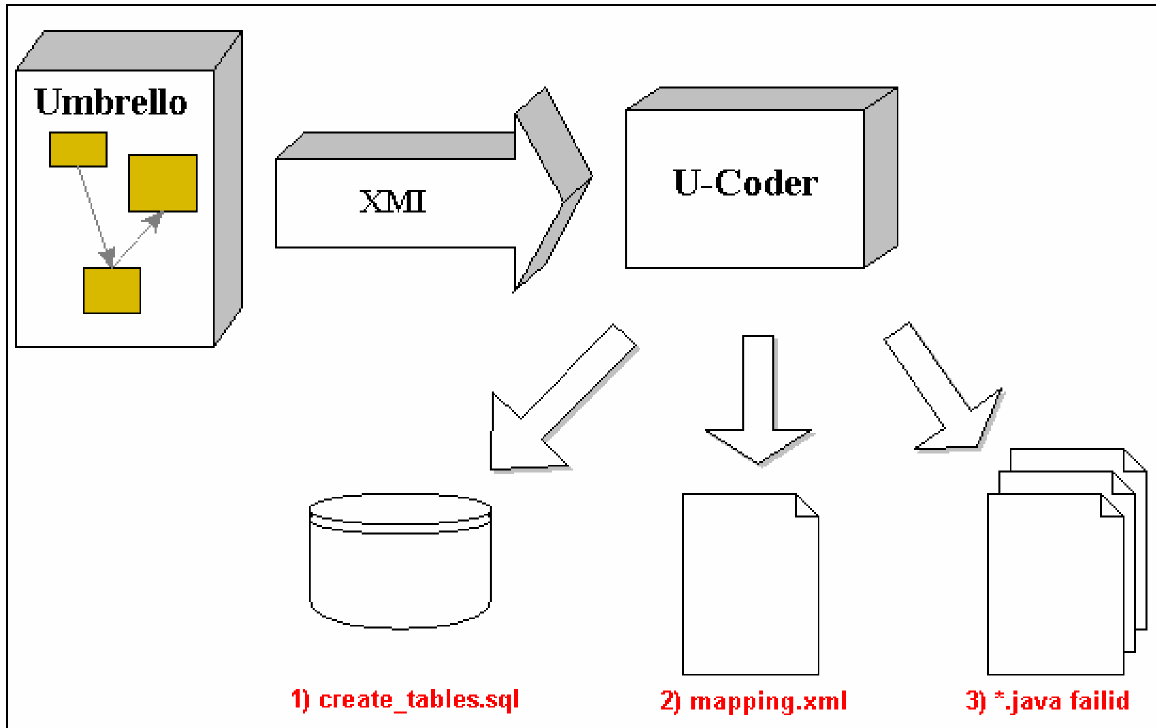


Joonis 2.3. Ühed ja samad andmed ning kontseptsioonid on läbivaks terves infosüsteemis.

Kui me suudaksime mudelitega piisavalt hästi kirja panna, milliseid reaalse maailma objekte meie andmeobjektid ja andmebaasid peavad esitama ning millised on kitsendused ja nõuded neile, siis sellise kontseptuaalse mudeli põhjal peaks olema võimalik luua mistahes andmete esitusviise.

2.2. Programm U-Coder

U-Coder (vt joonis 2.3) on programm, mis Umbrello klassidiagrammi põhjal loob automaatselt Java klassistruktuuri, tabelistruktuuri relatsioonilise andmebaasi jaoks ning objekt-relatsioonilise teisenduskirjelduse Castor jaoks.



Joonis 2.3. U-Coder genereerib klassimudeli põhjal objekt-relatsioonilise andmebaasi.

Loodud struktuurid on omavahel kooskõlalised ning võimaldavad pärast (näite)andmete sisestamist koheselt alustada tööd infosüsteemi dünaamilise osa koostamiseks.

U-Coder on realiseeritud Java keeles, sisendiks olev Umbrello mudel (XMI fail) loetakse vastavasse objektistruktuuri ning vastavalt teatud reeglitele luuakse selle põhjal objektistruktuurid, mis esitavad väljundeid. Kuna sisendiks oleva faili struktuur on XMI lihtsustus, siis on mõeldav mistahes modelleerimisvahendi XMI failide kohandamine sisendiks sobivaks (kasutades nt XSL tehnoloogiat).

UML klassidiagrammi põhjal Java klasside genereerimine ei ole keeruline, kuid andmebaasistruktuuri loomine on mõnevõrra keerulisem. Järgmises peatükis vaatlemegi reegleid, mille põhjal U-Coder genereerib andmebaasitabelite kirjeldusi.

2.2.1. Klassidiagrammist andmebaasiskeemini

MDD paremaks mõistmiseks vaatleme näitena reegleid, mille alusel U-Coder loob andmebaasitabeleid[1].

Tüüpilise UML assotsiatsiooni põhjal klasside A ja B vahel saame genereerida andmebaasitabelid järgmiste reeglite alusel:

iga UML klassi kohta luua teda esitav tabel, igasse tabelisse luua 'integer' tüüpi väli id -- andebaasiidentifikaator.

iga UML atribuudi kohta luua väli vastava klassi tabelisse

iga UML assotsiatsiooni jaoks:
kui multi(A)>1 ja multi(B)>1, siis
luua seosetabel ja
luua võõrvõti klassi A tabelisse mis viitab seosetabelile ja
luua võõrvõti klassi B tabelisse mis viitab seosetabelile
muidu
kui multi(A)<=1 ja multi(B)>1, siis
luua võõrvõti klassi A tabelisse, mis viitab klassi B tabelile
muidu //üks-ühele seose korral
luua võõrvõti ühte tabelitest, mis viitab teisele tabelile

UML andmetüüp 'string' tuleb teisendada SQL 'varchar'-ks

UML andmetüüp 'int' tuleb teisendada SQL 'integer'-ks

UML-is on ühe seosetüübina võimalik kasutada agregeerimist, mis on seos väljendamaks objektide eluigade sõltuvust. Kuna aga relatsioonilises andmebaasis pole otseselt võimalik väljendada ei agregeerimist (ega ka koostamist ehk komposiit-agregeerimist), siis kasutame agregeerimise väljendamiseks samu meetodeid, kui tavalise UML assotsiatsiooni korral. Siiski, kompositsiooni korral saame kasutada Castori võimalust määrata üks klass sõltuvaks teisest klassist (selleks on andmeteisenduses atribuut `depend`), niisiis toimib kompositsioon Castori tasemel. See tähendab, et kui liitobjekt eemaldatakse, siis kustutatakse andmebaasist ka temaga seotud osa-poolsed

[1] On veel palju reegleid mida siinkohal ei ole mõttekas näidata (reeglid tabelitele ja tabeli väljadele unikaalsete nimede valimiseks, andmetüüpide teisendamiseks, reeglid tabelite loomise järjekorra kohta jne.)

objektid. Ka vastupidine toimib, st Castor ei luba andmebaasi luua osa-poolset objekti, kui ta ei ole seotud tervik-poolse objektiga.

Pärimisseose väljendamiseks kasutame niinimetatud relatsioonilist pärimist, kus alamklass kajastub andmebaasis kahe tabelina: eraldi tabelis hoiame lisainfot alamklassi objektide kohta ja ülemklassi info on üldisemas tabelis.

2.2.2. Transformatsiooni parameetrid.

Kontseptuaalses mudelis ei ole mõttekas kasutada atribuutide tüüpidena andmebaasi andmetüüpe (`varchar`, `blob`, `timestamp`, jne). Samas ei ole ka Java tüüpide kasutamine põhjendatud (kuidas me sel juhul andmebaasitüübid saaksime?). Niisiis tuleb kasutada muud lahendust. Me ei tea andmebaasitabelite genereerimisel, kas mudelis märgitud `string` on mõeldud hoidma inimese eesnime (tüüpiliselt alla 20 tähemärgi) või näiteks 5MB suurust XML faili. MySQL andmetüübid, mis selliseid väärtusi hoida suudavad, on aga täiesti erinevad. Seega täisautomaatse tabelite tegemise puhul tuleb arvestada, et genereeritud kood ei pruugi kõige täpsemini vajadusi rahuldada. Programmis U-Coder on andmetüüpide tõlkimiseks kasutusele võetud eraldiseisev tüübiteisenduste fail, mis sisuliselt annab võimaluse defineerida transformatsiooni parameetreid (vt Lisa 1).

Kasutades tüübiteisenduse faili saab kasutaja ise kirjeldada endale sobilikke tüübiteisendusi. Näiteks võib ta defineerida andmetüübi `nimi`, mis kujutatakse maksimaalselt 20 tähemärgiga andmebaasis (`varchar(20)`) ja `String` tüüpi muutujana Javas.

Kokkuvõtteks võime öelda, et U-Coderi abiga saame kiiresti luua infosüsteemi prototüübi, täisautomaatne genereerimine säästab meid suurest hulgast käsi- ning mõttetööst, liiatigi kui kasutaja on objekt-relatsiooniliste teisenduste suhtes eelteadmisteta. Sellise infosüsteemi loomiseks piisas kasutajal mudelite joonistamisest ning tüübiteisenduste defineerimisest, ülejäänud töö oli võimalik teostada täisautomaatselt.

3. Mudelil põhinev arhitektuur

Programmi U-Coder esimene versioon on loodud semestritöö raames Andres Vilgota poolt sügisel 2003. U-Coderi eripära võrreldes teiste koodigeneraatoritega seisneb kooskõlaliselt mitmele platvormile korraga koodi genereerimises [Vil03]. Samas suunas töötab ka Object Management Group, kelle mudelipõhise tarkvaraarenduse raamistikku (MDA-d) selles ja järgnevates peatükkides tutvustame ja uurimegi.

Mudelil põhinev arhitektuur on OMG poolt defineeritud raamistik tarkvara arendamiseks. MDA seisneb mudelite koostamises otse koodi kirjutamise asemel. See tähendab, et kogu tarkvara loomine on modelleerimine, mudeleid tuleb lihtsalt koostada piisava täpsusega ja selliselt, et nad oleksid loetavad ka arvutile, mitte ainult programmeerijale. MDA püstitatud eesmärgiks on tulevikus saavutada olukord, kus kogu tarkvara on kirjeldatud ainult mudelitega, milledest saab automaatselt genereerida platvormile sobiliku koodi. MDA kasutuselevõttu peetakse evolutsiooni järgmiseks sammuks tarkvaraarenduses. Paralleele tõmmates on heaks näiteks tänapäevaste kõrgtaseme keelte võrdlus madala taseme assemblerkeelega.

Tänapäeva programmeerija ei kasutata assemblerkeelt infosüsteemi ehitamiseks, kõrgtaseme keele tõlkimine assemblerisse on nii loomulik, et assembleri kasutamine on programmeerija jaoks nähtamatu. MDA eesmärgiks on saavutada samasugune nähtamatus koodi tasemel – mõtleme ainult mudelites. Paneme tähele, et mudel võib siinkohal tähendada mistahes kirjeldust süsteemist, süsteemi abstraktsiooni. Me ei sea piiranguid mudeli esitusviisile – peamisteks tingimusteks on arusaadavus ja üheseltmõistetavus arvutile. Seega võib ka programmi koodi vaadelda mudeli eriliigina. MDA kontekstis aga ei tavatseta koodi mudeliks pidada, sest temaga töötamiseks tuleb ta lahti parsida.

Mudeli põhjal reaalse töötava süsteemi saamiseks kasutatakse mudelite transformatsioone. MDA realiseerimise juures on mudelite transformatsioonid ehk teisendused tähtsaimal kohal. Teisendused ei ole lihtsad ning nõuavad head arusaamist kogu MDA arendusprotsessist. Teisenduse rakendamine koosneb teisendusreeglite defineerimisest ja nende (pool-)automaatsest rakendamisest. Õnneks ei pea iga kasutaja hakkama iga platvormi jaoks reegleid defineerima, enamlevinud transformatsioonireeglid

peaks tulevikus olema vabalt kättesaadavad kas siis vastavate platvormide loojate poolt või vabavaraliselt huviliste poolt arendatuna.

Mudelipõhise arhitektuuri nurgakiviks on UML mudelid („*models drive everything*"). Mudelid jagunevad ärimudeliks (ingl. k. *Computationally Independent Model*) ja tarkvaramudeliks (näiteks platvormisõltumatu mudel). CIM koosneb sellistest reaalses elus kehtivatest reeglitest, millede kasutamiseks ja kehtimiseks ei ole tarvis tarkvarasüsteeme. Lühidalt öeldes esitab CIM probleemide sfääri ning PIM juba lahendust. MDA eesmärgiks on võimalikult palju infost, mis käib CIM kohta kirjutada PIM mudelisse. Järgnev loetelu peaks iseloomustama MDA uuenduslikkust ja püstitatud eesmärkide ulatust:

- protsess on mudel
- platvorm on mudel
- transformatsioon on mudel
- mudeli kirjeldus on mudel
- mudeli element on mudel
- programm on mudel
- test on mudel
- muster on mudel
- jne

Kui objekt-orienteeritud programmeerimine andis programmeerijale võimaluse mõelda objektides, siis MDA annab meile võimaluse mõelda veelgi üldisemalt, võimaluse mõelda vaid mudelites. Objekt-orienteeritud programmeerimise põhiteesiks olev lause „kõik on objekt", kõlab MDA raamistikus lausena „kõik on mudel".

Muster on idee, mis on olnud kasulik rakendada ühe probleemi lahendamisel ja mis ilmselt on kasulik rakendada ka mujal [Fow02]. MDA kontekstis tuleks väidet „muster on mudel" mõista nii, et kui me suudame konkreetse mustri formaalselt kirja panna, näidates millises olukorras seda tasub rakendada ja kuidas seda rakendada, siis edaspidi saame konkreetses olukorras mustri rakendamise jätta arvuti ülesandeks.

3.1. MDA põhialused

Alljärgnev on ülevaade MDA mõistmiseks vajalikest kontseptsioonidest: mudelitest, platvormidest ja mudelite transformatsioonidest.

3.1.1. Mudel

Mudel on süsteemi mõne osa, näiteks funktsiooni, struktuuri või käitumise esitus, enamasti kõrgema abstraktsioonitasemega kui seda on süsteem ise. Kõrgema abstraktsioonitaseme all mõistame siinjuures kirjelduse detailsuse vähenemist. Madalaima abstraktsioonitasemega mudel on platvormisõltuv mudel, mis on sisuliselt koodi üks-ühene kirjeldus.

Selleks, et mudelite põhjal genereerida töötav süsteem, peab mudelite täpsusaste olema piisavalt suur. Martin Fowler jaotab UML mudelid täpsusastme ja otstarbe põhjal visanditeks (UMLAsSketch), plaanideks (UMLAsBlueprint), ning veelgi täpsemateks, iseseisvalt käivitatavateks mudeliteks (UMLAsProgrammingLanguage) [Fow03]. MDA jaoks ei ole sobilikud mudelid, mis on mitmeti mõistetavad või sisaldavad liiga vähe infot.

3.1.2. Platvorm

Platvormiks nimetatakse mudeli käivitamiseks tarvilikku keskkonda. Lihtsaim näide keskkonnast, kus midagi käivitada saab on Java platvorm. Platvormid võivad üksteisele kuhjuda – Java kood käivitub Java 2 platvormil, mis käivitub Java Virtual Machine-s, mis käivitub Linux platvormil, mis omakorda käivitub PC platvormil.

MDA üheks oluliseks eesmärgiks on saavutada platvormisõltumatus. Osutub, et selle saavutamiseks tuleb võtta sootuks teine lähenemine kui senises tarkvaraarenduses. Traditsiooniliselt arendatakse programme niiöelda alt üles, see tähendab allasuvate platvormi võimalusi ja kitsendusi silmas pidades: esmalt otsustatakse, kas kirjutada programm Windowsi või Linuxi jaoks, seejärel valitakse implementatsiooniks kas C või Java keel. MDA visiooni kohaselt käib arendus vastassuunas – ülevalt alla. Selleks kirjeldatakse esmalt aplikatsioon ja selle loogika sõltumatult mistahes platvormist (UML mudelites), seejärel teisendatakse seesama loogika sobiva platvormi keelde (C, Java,

muu). Sellise lähenemisega saavutataksegi platvormisõltumatus – iga keele jaoks defineeritakse kujutus platvormisõltumatust mudelist selle platvormi keelde. Ülalt-all lähenemisega saavutatakse ka kindlus selle suhtus, et vajaduse korral on süsteem võimalik tööle panna mistahes tulevikus tekkida võivates *„the next best thing“* platvormides.

3.1.3. PIM – platvormisõltumatu mudel

PIM (*Platform Independent Model*) on tarkvarasüsteemi selline mudel, mis on piisavalt sõltumatu võimalikest implementeerimiseks kasutatavatest tehnoloogiatest. PIM koostamisel on peaeesmärk keskenduda süsteemi loogiliste komponentide omavahelistele suhetele, jättes kirjeldamata kõik sellised omadused, mis on seotud juba implementatsiooniga. Näiteks ei määrata PIM-is, kuidas ja millisesse andmebaasi klassi objekte salvestatakse, ei määrata isegi seda, et objekt on salvestatav. UML stereotüübi `«persistent»` määramine koormaks platvormisõltumatut mudelit liigselt. Selle asemel määratakse vastav omadus transformatsiooni käigus.

3.1.4. PSM – platvormispetsiifiline mudel

MDA mudelite hierarhias asuvad PIM järel platvormispetsiifilised mudelid, PSM-id (*Platform Specific Model*). PSM on mudel, mis kirjeldab süsteemi elemente implementatsiooniks kasutatava platvormi mõistetega. Näiteks relatsioonilise andmebaasi kirjeldamiseks kasutatakse mõisteid tabel, väli, võti, jne.

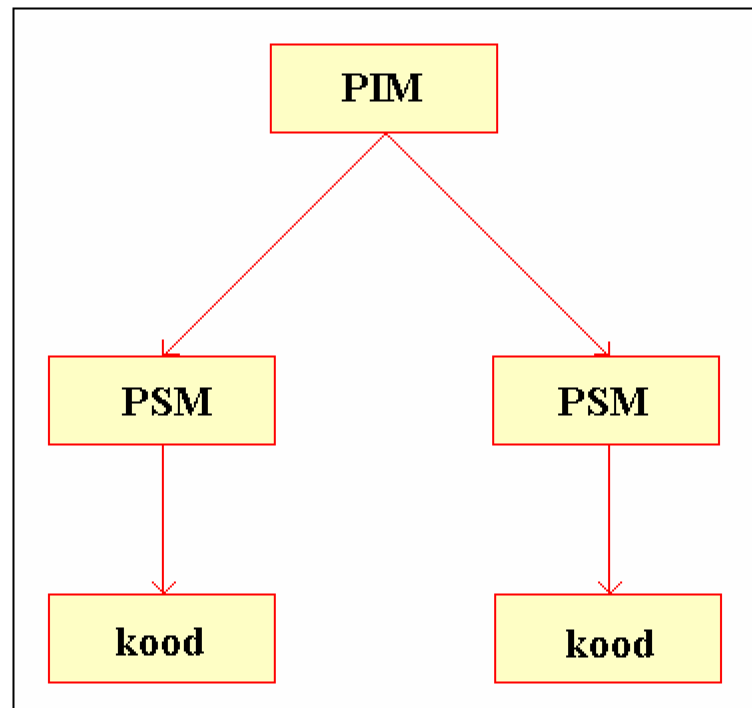
3.1.5. Lähtekood

Kolmas ja viimane samm arendusprotsessis on platvormispetsiifilise mudeli põhjal koodi kirjutamine. Platvormispetsiifiline mudel kirjeldab vastavat platvormi sisuliselt üks-üheselt, seega PSM põhjal koodi genereerimine on küllaltki lihtne. Enamik tänastest UML koodigeneraatoritest niiviisi töötavadki – selle erinevusega MDA-st, et puudub selgepiiriline eristus PIM ja PSM vahel, vahepealset platvormispetsiifilist mudelit, kasutajale ei näidata.

3.1.6. Automaatsed mudelite transformatsioonid

Traditsioonilises tarkvaraarenduses on palju tööriistu (näiteks U-Coder), mis suudavad mudelist koodi genereerida. Ühest mudelist teise mudeli saamine on aga puhas käsitöö. Kontseptuaalsest mudelist andmebaasiskeemi ja samast mudelist Java klassimudeli tegemine ning vastavate mudelite kooskõlalisena hoidmine ei ole lihtne.

MDA eesmärgiks on saavutada (pool)automaatne mudelite transformeerimine, kus kasutajalt küsitakse transformatsiooniks vaid mõningaid tarvilikke parameetreid. Transformatsioonidest rääkides kasutatakse termineid PIM ja PSM, enamasti on eesmärgiks PIM teisendamine üheks või mitmeks PSM-ks. Platvormisõltumatu mudel ja platvormisõltuv mudel on tegelikult relatiivsed terminid, st MDA transformatsioonis transformeeritakse alati PIM PSM-ks, aga saadud PSM võib olla järgmise transformatsiooni sisendiks, seega järgmise transformatsiooni jaoks on ta PIM. Näiteks kontseptuaalne klassidiagramm on kujuteldava UML→Relational transformatsiooni jaoks PIM, genereeritud PSM on aga Relational→MySQL jaoks PIM.



Joonis 3.1. Platvormisõltumatu mudeli saame automaatselt teisendada mitmele platvormile sobivaks.

Transformatsioonid peaksid olema maksimaalselt automatiseeritud. See tähendab, et kui teisenduse sooritamiseks on tarvis parameetreid ja konfiguratsioone, siis kasutaja peab saama neid küll määrata, kuid transformaator peaks juba määratud seadistused meelde jätma. Vastasel korral küsitaks parameetreid mudeli igakordsel teisendusel uuesti.

3.2. MDA eelised

Traditsioonilise tarkvaraarendusega võrreldes on MDA-l nii miinuseid kui ka plusse. Järgnevalt vaatleme, kuidas MDA lahendab esimese peatükis kirjeldatud traditsioonilise tarkvaraarenduse probleemid.

3.2.1. Produktiivsus

MDA protsessi korral on arendaja ülesandeks võimalikult hea PIM loomine. Kuna PIM transformeeritakse automaatselt PSM-iks ja koodiks, siis ei pea modelleerija mõtlema mismoodi tema mudeleid koodis kujutatakse. Selle asemel saab PIM looja keskenduda konkreetse projekti probleemidele vaadeldes süsteemi lahusolevalt allasuvatest tehnoloogiatest [KWB03]. PIM mudelis kirjeldatakse võimalikult palju süsteemi loogikat, sealhulgas kitsendusi, algoritme, üldist struktuuri.

PIM on mudel, mis kirjeldab süsteemi abstraktsemalt kui kood, seega ta on sobilik dokumentatsiooniks. PIM annab hea ülevaate nii tulevastele programmeerijatele, kes peavad hakkama süsteemi muutma, kui ka juhtidele ja tellijatele. Traditsioonilise tarkvaraarenduse korral oleks tulnud PIM-le sarnanev kontseptuaalne dokumentatsioon spetsiaalselt lisaks luua, MDA korral tekkis see loomulikul teel ise.

3.2.2. Platvormide arenemine

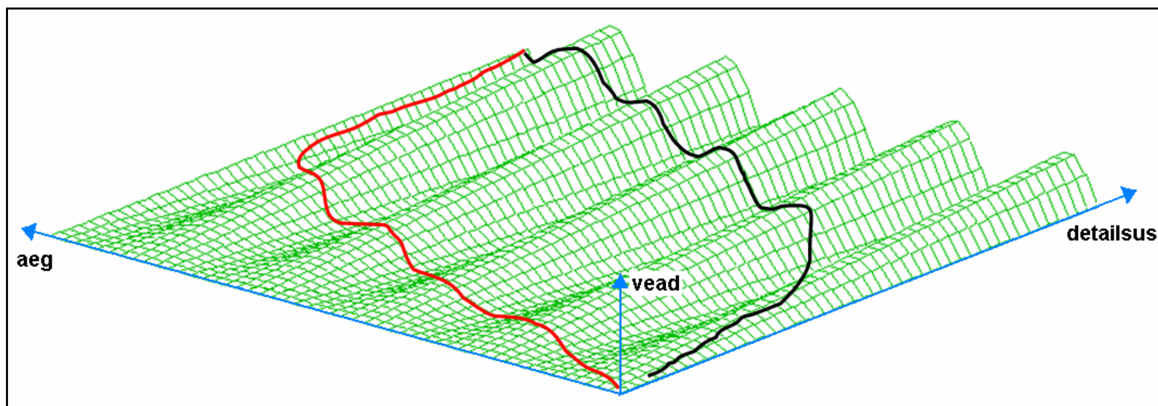
Kuna enamik arendustööst on fokuseeritud PIM arendamisele ja PIM→PSM transformatsioon on automatiseeritud, siis ei valmista platvormi vahetamine mingeid probleeme. Kogu PIM tasemel defineeritud loogika (seal peaks olema kirjas enamik projekti loogikast) on täielikult taaskasutatav mistahes teise platvormi jaoks. Paneme tähele, et traditsioonilise tarkvaraarenduse korral see nii ei ole: kuigi süsteem võib olla võimalikult hästi kodeeritud, siis loogika eraldamine tehnoloogilistest detailidest on ilma välise kirjelduseta küllaltki keeruline. Kui vana süsteem on tarvis üle viia uuele platvormile (keelele), siis tuhandetele koodiridadele otsa vaadates tekib küsimus – kustkohast alustada?

3.3.3. Platvormidevaheline sõltuvus

Erinevatel ühest ja samast PIM-st loodud mudelitel on tarvidus omavahel suhelda, näiteks objekti esitusest relatsioonilises andmebaasis peab olema võimalik saada objekti esitus XML kujul. Seega on PSM-id omavahel kooskõlaliselt suhestatud ja nende vahele on tarvis sildasid. Aga ka need sillad saab PIM-st genereerida, sest me teame kuidas oleme genereerinud vastavad PSM-id. Silla näiteks ongi objekt-relatsiooniline andmeteisendus, U-Coder genereeris teisenduskirjelduse automaatselt.

3.2.4. Abstraktsus

Korralikult modelleerides ning mudeleid vähese vaevaga (automaatselt) koodiks teisendades tuleb kokkuvõttes teha vähem tööd kui otse koodi kallal töötades. Abstraktsus tarkvara arendamise käigus annab võimaluse vähema vaevaga parandada sisulisi vigu [IO02], suure detailsuse korral tuleb vigade ilmnemisel teha palju parandustööd (joonis 3.2).



Joonis 3.2. Abstraktsus annab põhimõtteliste vigade parandamisel eelise.

Kui meil on võimalik mitte ainult mõelda, vaid ka tegutseda kõrgema abstraktsustasemega (joonisel vasakpoolne, laugem joon), siis „lihtsad muudatused“ (vt peatükk 1.6) on võimalik sisse viia palju vähema vaevaga. Transformatsioonide

loomisega näevad programmeerijad vaeva iseenda töö lihtsustamiseks, lisaks on suure tõenäosusega samad transformatsioonid taaskasutatavad ka teiste projektide juures. Kui kohe arendusprotsessi alguses süveneda detailidesse, siis tuleb sisuliste otsuste muutumisel teha palju parandusi käsitsi.

3.3. MDA puudused: UML kui PIM keel

Modelleerimise tunnustatuima, UML keele tugevaimaks küljeks on süsteemi struktuursete aspektide kirjeldamine. UML nõrgaks küljeks on aga süsteemi dünaamika kirjeldamine, sest kasutatavad diagrammid ei ole piisavalt formaalsed ega täielikud, võimaldamaks täisfunktsionaalse programmi genereerimist. Klassidiagrammis loetakse meetodite sisud rakendusest sõltuvaks ja jäetakse täiesti kirjeldamata, niisiis ei sobitu UML diagrammid MDA konteksti ideaalselt.

Dünaamika modelleerimiseks UML vahenditega on siiski võimalusi:

1. ***käivitatav UML*** (ingl. k. *Executable UML*, pole standardiseeritud) defineerib olekumasinad, milledega saab käitumist modelleerida. Käivitatava UML-i miinuseks on asjaolu, et kasutatav AS keel (ingl. k. *Action Semantics*) keel pole standardiseeritud, ning selle kasutamine ei anna abstraktsuse näol sugugi eeliseid otse koodi kirjutamise ees. Kõrgema taseme mudelis tuleb kirjutada ikkagi samapalju koodi, küll aga saavutame niimoodi platvormist sõltumatuse.

Teiseks võimaluseks on täpsustada mudeli elemente (klasse, atribuute, seoseid, meetodeid) spetsiaalse keele abil. UML standardi spetsifikatsioon sisaldab endas objektitõkkekeelt [LL00]. Objektitõkkekeel (ingl. k. *Object Constraint Language, OCL)* on lähema vaatluse all peatükis 4.2.

2. **UML ja objektitõkkekeele kombinatsioon** teeb klassimudeli oluliselt täpsemaks, seades näiteks kitsendusi atribuutide (alg)väärtustele või kohustuslikkusele [WK03]. Kirjeldades meetodite eel- ja järeltingimused saame OCL abi kasutada isegi dünaamika kirjeldamiseks. Nimelt on tihtipeale võimalik järeltingimus(t)e põhjal genereerida ka vastav meetodi sisu. Keerulisematel juhtudel aga ei ole see võimalik ja meetodite sisud tuleb kirjutada käsitsi. Sel juhul aitavad eel- ja järeltingimused veenduda, et kirjutatud meetodi sisu vastaks mudelis määratud nõuetele (ingl. k. *Design by contract*). OCL ja UML kombinatsiooni peetakse seni parimaks võimaluseks süsteemi dünaamika kirjeldamiseks.

Tuleb nentida, et praeguses situatsioonis pole meil mugavat võimalust dünaamika kirjeldamiseks mistahes abstraktsioonitasemega mudelites, tehniliselt kindlaimaks ja ilmselt ka kiireimaks võimaluseks on siiski kodeerimine koodi tasemel. Selline olukord ei soosi MDA kasutamist, sest kasutajal (programmeerijal) tekib tahtmine hüljata modelleerimine ning kasutada otseteed – programmeerida kogu süsteem koodis. Niiviisi aga kaob suur osa MDA mõttest, kaob platvormist sõltumatus ja kooskõla mudelite vahel. Olukord ilmselt paraneb lihtsate ja visuaalsete keelte tekkimisel tulevikus.

Olukord, kus mudeli põhjal ei saa genereerida täisväärtuslikku süsteemi, st tulemus nõuab kasutajapoolset lisatööd meetodite muutmise või lisamise näol, pidurdab MDA kasutamist. See on ka peamiseks põhjuseks miks MDA ei ole senini oma kasutajaskonda leidnud.

4. OMG standardid

Standardite loomine on äärmiselt tänuväärne nii mitmeid aspekte hõlmavas valdkonnas nagu seda on mudelipõhine tarkvaraarendus. Tarvis on tegeleda mudelihulkade haldamisega, mudelite teisendamisega, nende versioonide haldamisega jne. Kuna MDA koosneb päris mitmest omavahel seotud standardist, milledest osad on loodud defineerimaks teisi, siis anname siinkohal ülevaate erinevatest MDA standarditest. Object Management Group poolt on loodud ka MDA võtmestandard: UML ehk unifitseeritud modelleerimise keel. Järgnevalt vaatame ka teisi MDA jaoks tarvilikke standardeid.

4.1. UML (*Unified Modeling Language*)

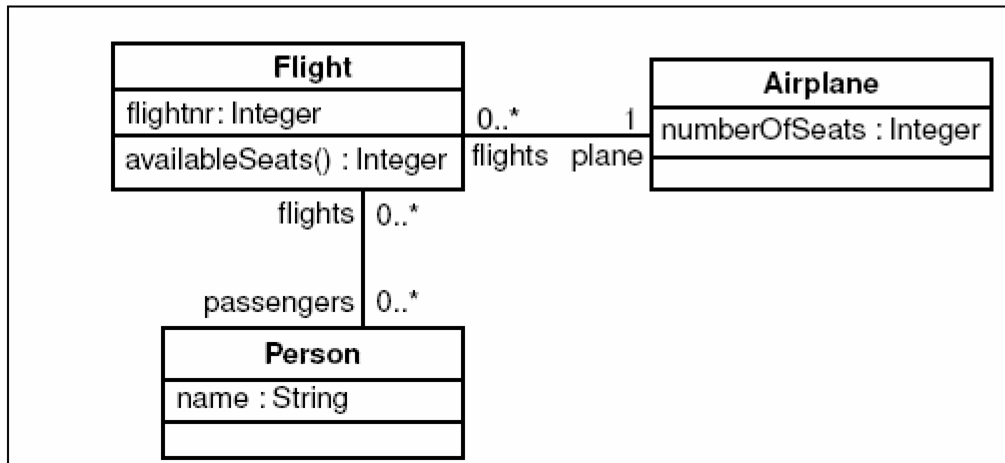
UML on OMG kõige tuntum standard, mille eesmärgiks on modelleerimise standardiseerimine. Praegusel hetkel on käsil standardi versiooni 2.0 loomine, sisuliselt on viimane versioon välja töötatud just MDA toetamiseks.

4.2. Objektitõkkekeel (*Object Constraint Language, OCL*)

Objektitõkkekeel on tekstipõhine keel, mida saab kasutada avaldiste koostamisel navigeerimiseks, tõkete ja eeltingimuste formuleerimiseks UML mudelitel [LL00]. OCL UML osana annab meile formaalse keele kõrvalmõjudest vabade tõkete esitamiseks invariantide, eeltingimuste ja järeltingimuste kirjeldamise keel. Nimest tulenevalt tuleks arvata, et OCL-i saab kasutada ainult kitsenduste määramiseks, tegelikult on olukord siiski veidi parem. OCL ei ole ainult kitsenduste määramiseks, ta on deklaratiivne, kõrvalefektideta ja platvormist sõltumatu keel väljendamaks ka päringuid ja avaldisi. Deklaratiivsus tähendab, et kasutatavad avaldised ei kirjelda samme tulemuse saavutamiseks (keel väljendab „kuidas?“ asemel „mida?“). Kuna ühegi OCL avaldise arvutamine ei muuda süsteemi, siis saab juba mudelite koostamise etapis ilma implementatsiooni teadmata mudelit testida.

4.2.1. OCL kasutamine

Klassidiagramm inimese jaoks ei ole sama, mis klassidiagramm arvuti jaoks. Inimene suudab tihtipeale modelleeritava süsteemi klassidiagrammi põhjal luua kujutluspildi reaalsest elust (joonis 4.2).



Joonis 4.2. Tüüpiliselt on klassidiagrammis inimese jaoks rohkem informatsiooni kui arvuti jaoks [WK03].

Mõte lennukist ja reisijatest toob meile kohe silme ette pildi inimestest, kes istuvad mugavates istmetes. Arvuti seda ei suuda ning tema jaoks saab mõiste „Lend“ (ehk ka klass Lend) oluliselt selgemaks, kui me ütleme, et reisijate arv ei ole suurem lennuki istekohtade arvust.

Näide 1. Kitsendus reisijate arvu kohta lennukis:

```
context Flight
inv: passengers -> size() <= plane.numberOfSeats
```

Eeltoodu ongi lihtne kitsendus arvutile arusaadavas keeles. Iga kitsendus töötab teatud kontekstis, kitsendusega saab määrata tingimused mis peavad kehtima alati (inv:), eeltingimused (pre:), järeltingimused (post:) või ka meetodi sisu (body:).

Näide 2. Objektitõkke keelega saab spetsifitseerida ka meetodite sisu:

```
context Flight::availableSeats() : Integer
body: plane.numberOfSeats – passengers -> size()
```

Näide 3. OCL kasutamine UML keele kirjeldamiseks:

```
context Class
inv: attributes -> isUnique(name)
```

Sellega ütleme UML keele kohta, et mõiste „klass“ (ehk iga Class isendi) kohta peab alati kehtima invariant: tema atribuutide hulgas ei tohi olla kaht sama nimega atribuuti.

Näide 4. Kitsendus UML Java profiili kirjeldamiseks:

```
context Class
inv: generalizations -> size() <= 1
```

Java profiilis modelleerides peab alati kehtima, et iga klassi ülemklasside arv peab olema null või üks. Sisuliselt ütleme, sellega, et Java-s ei ole mitmest pärimist.

4.3. CWM (*Common Warehouse Metamodel*)

CWM on OMG arendatud ja juunis 2000 vastu võetud standard võimaldamaks meta-andmete vahendamist andmelaondusega tegelevate keskkondade vahel. CWM annab võimaluse modelleerida metaandmeid relatsiooniliste, mitte-relatsiooniliste, mitmedimensionaalsete ja muude andmebaaside jaoks [CWM]. CWM eesmärgiks on ühendada erinevates allikates asuvad andmed ühtsesse andmekaevanduse jaoks sobilikku raamistikku, sealjuures võimaldades tagasiulatuvalt uurida, kust on andmed pärit ja millal nad tekkisid (ingl. k. *lineage tracing*) [Poo03]. CWM mudelite vahendamiseks kasutatakse XMI dokumente.

4.4. MOF (*Meta Object Facility*)

MOF [MOF] on standard, mis defineerib ühe abstraktse keele metamudelite (nagu UML ja CWM) kirjeldamiseks. UML kui modelleerimise keel on defineeritud kasutades MOF keelt, MOF-ga on määratud ka kitsendused ja kõik UML kooskõlalisuse (ingl. k. *well-formedness*) reeglid, näiteks „ühes klassis ei tohi olla kaht sama nimega atribuuti“. MOF on loodud UML kõige üldisema osa põhjal. MOF on UML-i ja CWM ühiseks mudeliks, võimaldades seega UML, CWM ja mistahes teiste MOF-põhiste mudelite ühtset kasutamist, salvestamist ja nendega manipuleerimist. MOF eesmärgiks ongi võimaldada erinevate domeenide metamudelite ühtset manipuleerimist, lisaks annab MOF ühise meetodi töötamaks välja uusi modelleerimise keeli.

4.5. XMI (*XML Metadata Interchange*)

XML põhine metaandmete vahendamine on kasulik mudelite salvestamisel ühtsesse formaati. Mudeliks võib siinkohal olla nii UML mudel kui ka CWM mudel, teisisõnu mistahes MOF abil defineeritud mudel. XMI on levinud eelkõige just UML mudelite salvestamiseks, kuid tegelikult on XMI eesmärk võimaldada mistahes metainfo salvestamist XML dokumendina, kaasa arvatud ka UML mudelite salvestamist.

Tegelik olukord mudelite püsiesituse standardiseerimisel ei ole kõige parem. Juba praegu on XMI standardist mitu versiooni, mis ei ole ühilduvad. Lisaks ei soosi XMI formaat ka mudelite versioonide muutumist: UML standardist on mitmeid versioone

(viimane neist UML 2.0), kuid erinevate versioonide XMI esitused ei ole omavahel ühilduvad ja tööriistadel võib tekkida raskusi mudelitega töötamisel [Fra04]. Seda vaatamata asjaolule, et suur osa UML kontseptsioonidest on läbi aegade jäänud muutumatuks.

4.6. Metamodelleerimine

MDA protsess sisaldab endas mitmete mudelitega töötamist. Kuna mudeleid on palju, siis arusaam mudelite omavahelistest suhetest on MDA mõistmiseks hädavajalik.

MDA realiseerimiseks kasutatakse neljakihilist standardite arhitektuuri, nendeks kihtideks on modelleeritava süsteemi kirjeldused järjestikustel abstraktsioonitasemetel. Järgnevas loetelus ongi vastavad abstraktsioonitasemed kõrvuti näidetega oma instantsidest.

M0 tase: andmed, inimene Andres Vilgota

M1 tase: mudel, Inimene: eesnimi, perekonnanimi

M2 tase: metamudel, UML Klass, UML Atribuut, UML Seos

M3 tase: meta-metamudel, MOF: (Meta)Klass, (Meta)Atribuut, (Meta)Seos.

Kogu abstraktsioonitasemete eraldamise mõtteks on, et iga alama taseme mudel on kõrgema taseme mudeli instants.

Madalaimal tasemel on tegelikud objektid, järgmisel tasemel on keel kirjeldamaks M0 tasemel asuvaid andmeid (kõige tavalisem UML).

M2 tasemel on keel, millega saab kirjeldada andmete kirjeldusi ehk M1 taset. Kui M1 tasemel asuvad mudelid, siis M2 taset võib nimetada metamudelite tasemeks. Metamudelid on mudelid, mille instantsideks on mudelid.

M3 tasemel asub MOF, keel, millega kirjeldatakse M2 taseme mudelite kirjeldamise võimalused. MOF keel on defineeritud piisavalt üldiselt, et kirjeldada ka iseennast.

Tänu MOF keelele saavad mudelitega tegelevad tööriistad lihtsasti tegeleda mistahes mudelitega, eeldades, et nad on teadlikud vastavate mudelite kirjeldusest. Näiteks selleks, et tööriist saaks töötada UML1.4 mudelitega, piisab kui ta teab UML1.4 metamudelit (milles on liidesed *UMLClass, Attribute, Operation* jne). Kui nüüd tööriist teab UML1.4

metamudelit (mille ta saab laadida XMI failist), siis saab ta töötada mistahes UML1.4 mudeliga. Siinkohal ei mõista me mudelitega töötamise oskuse all nende visualiseerimise, kasutajaga dialoogide pidamise oskust jne, vaid mudelielementide lugemisoskust, elementide filtreerimise, teisendamise ja XMI formaati salvestamise oskust. Sellised oskused on igal MOF-teadlikul tööriistal – ta saab ilma mudelielementide semantikat teadmata lugeda sisse näiteks andmebaasi mudeleid, mille metamudelis on liidesed *Table, Column,* jne.

5. Mudelite transformatsioonid

Sisuliselt ei ole meil piiranguid keelte ja meetodite valikule millega mudeleid transformeerida, ometigi on ühed lahendused paremad kui teised.

Teame, et levinuim viis mudelite esitamiseks on XML-põhine, täpsemalt XMI formaadis. Transformatsioonide teostamiseks võiksime kirjutada vastavaid XSL transformatsiooniskripte ja kasutada selleks XSLT protsessorit [XSLT], kuid XML töötlemiseks mõeldud XSL keel ei ole piisavalt spetsiaalne mudelite teisendamiseks. XSL on mudelite teisendamise seisukohast nn madala taseme keel. Me ei saa XSL abil mugavalt kirja panna formaalseid teisendusreegleid, me võime kirja panna vaid XML dokumentide teisendusi. See aga tähendab, et tegelik kirjapandud reegli sisu ei ole hästi jälgitav.

Teiseks võimaluseks on spetsiifiliste tööriistade arendamine (U-Coder). Selline tööriist täidab oma eesmärki täiesti rahuldavalt, kuid tal on üks suur miinus – teisenduste loogika ei ole lahus teisenduste rakendamise mehhanismist. Programmi U-Coder koodis ei ole selgepiirilisi kohti transformatsioonireeglite kirjutamiseks, andmetüüpide teisendamiseks, nimede määramiseks (ingl. k. *naming conventions*) jne. Üldiselt tähendab see, et teisendusreeglite muutmine on raske ning madalal tasemel kirjutatud transformatsioonid ei ole hallatavad.

Mõningad loomulikud nõuded transformatsioonidele on järgmised:

- Konfigureeritavus – näiteks veebiliidese genereerimisel peaks saama määrata, millist komponenti kasutada tõeväärtustüüpi atribuudi kuvamiseks. Variantideks võiks olla checkbox, drop-down, radiobutton jne.
- Transformatsiooni kulgemise jälgitavus (ingl. k. *traceability*) – oleks hea, kui saaksime genereeritud mudeli elementide kohta selgitada, milliste algmudeli elementide põhjal ta genereeriti. See nõue teenib nii transformatsiooni parandamise (ingl. k. *debugging*) kui ka tagasivõtmise (ingl. k. *undo*) eesmärki.
- Inkrementaalne kooskõlalisus (ingl. k. *incremental consistency*) – sihtmudelisse

lisatud informatsioon peaks ka sihtmudeli teistkordsel genereerimisel säilima. Oleks hea, kui väikeste muudatuste sisseviimisel algmudelisse ei genereeritaks uuesti mittemuutunud süsteemi osi. Näiteks kui meil on juba töötav süsteem reaalsete andmetega ja me soovime 1..* seose muuta 0..* seoseks – andmebaasi tabelitestruktuuri täielik taasloomine ei ole siinkohal mõttekas.

Selleks, et probleemidesse takerdumise asemel täita eeltoodud nõudeid, tuleb transformatsioonide loogika eraldada nende täideviijast. Transformatsioonide loogika väljendamiseks tuleb luua/valida sobilik transformatsioonikeel ning tööriistad, mis suudavad selles keeles väljendatud reegleid mudelitele rakendada. Käesoleva töö kirjutamise ajaks ei ole transformatsioonide kirjeldamiseks mõeldud keeli standardiseeritud ning iga tööriist kasutab reeglite väljendamiseks ja rakendamiseks spetsiaalselt väljatöötatud lahendusi. Ainult keel(t)e standardiseerimisel saab võimalikuks erinevate tööriistade koostöö.

5.1. Transformatsiooni keeled

Mudelite teisendamiseks on tarvis teisendusreegleid ja tööriista, mis oskab neid reegleid mudelitele rakendada. Käesolevas peatükis vaatlemegi mudelite teisendusreeglite väljendamise viise. Näitlikustame keelte eripärasid ja sobivust erinevate MDA nõuete rahuldamiseks.

Võimalike teisenduskeelte valimisel on kaks peamist lähenemist[1]:

- Deklaratiivne lähenemine – lähte- ja sihtmudeli vahelised seosed kirjeldatakse kasutades vastavaid kitsendusi ja seoseid. Deklaratiivse lähenemise korral kirjeldatakse ainult transformatsiooni soovitav tulemus, mitte aga algoritm tulemuse saamiseks.
- Imperatiivne lähenemine – kasutades mingit formaalset keelt kirjeldatakse algoritm, kuidas saada lähtemudelist sihtmudel. Transformatsioon teostub järjestikuseid algoritmi samme täites.

[1] Lisaks on veel mõeldavad ka hübriidkeeled deklaratiivsest ja imperatiivsest lähenemistest

5.1.1. (QVT) Query/Views/Transformations

Käesoleval aastal peaks sündima QVT-nimeline standard, millega sünnib transformatsioonikeel kõige üldisema teisenduse, ühest mudelist teise mudeli saamise jaoks. Standardi loojaks on Object Management Group.

QVT standardi [QVT] nõueteks ja eesmärkideks on:

- defineerida päringud mudelitel. Tingitud vajadusest mudeli elementide filtreerimiseks ja valimiseks. Võrreldav XSLT päringukeelega XPath [XPATH].
- defineerida keel esitamaks vaateid mudelitel. Sisuliselt nimetatakse vaateks mistahes algmudelist tuletatud mudelit.
- defineerida deklaratiivne keel transformatsioonidefinitsioonide kirjeldamiseks MOF metamudelite vahel.
- anda abstraktne süntaks sellise transformatsioonikeele jaoks MDA metamudelina (anda transformatsioonimudeli kirjeldus).

Võimalikud lisanõuded transformatsiooni keelele (QVT standardi mittekohustuslik osa):

- Kahesuunalisus – algmudelile transformatsiooni rakendamine võiks olla pööratav, st sihtmudeli põhjal võiks saada luua esialgset mudelit. See nõue on seotud ka transformatsiooni jälgitavuse nõudega, kuid täiel määral kahesuunalisust sisse seada ei saa, sest lähtemudeli ja sihtmudeli keelte väljendusvõimekused ei pruugi olla samaväärsed.
- Transformatsioonireeglite taaskasutatavus. Nii nagu objekt-orienteeritud programmeerimiskeeltes on võimalik juba kirjutatut taaskasutada, peaks saama taaskasutada ka defineeritud transformatsioone. Ka siin on märksõnadeks modulaarsus ja kapseldatus.
- Arvestada lisainformatsiooni lisamise vajadusega, eelkõige MDA tänases

olukorras, kus meetodite sisud tuleb tihipeale ise kirjutada.

- Teisendusparameetrite määramise võimalus.
- Võimalus määrata tingimuslikke teisendusi.

5.2. Transformatsioonid

Transformatsioonikeelest ja mudelitest üksi ei piisa, et mudeleid teisendada. Tarvis on ka tööriista, mis suudab mudeleid ja teisendusdefinitsioone lugeda, mõista, rakendada. QVT keele nõuetes on praktilise vajadusena mainitud ka interpretaatori ehitamise lihtsust. Paneme tähele, et tegelikult ei pruugi interpretaatori loomiseks tarvis olla enamat kui vaid transformatsiooni, mis teisendaks QVT-keelsed transformatsioonid mingisse madalama taseme keelde. Näiteks võib tegeliku transformeeriva keelena kasutada TXL keelt [PR03], mis on kasutust leidnud peamiselt lähtekoodi teisendamisel ühest keelest teise ja on seega end selles valdkonnas juba tõestanud.

Olekuta transformatsioon (ingl. k. *stateless transformation*) on selline transformatsioon, mida rakendatakse lihtsate sammude kaupa ja mis ei nõua lisateadmisi transformatsiooni kulgemisest väljaspool seda sammu.

Olekuga transformatsioonireegel (ingl. k. *stateful transformation*) on selline, mis kasutab ka välisest kontekstist pärit infot, sellise transformatsiooni näiteks on UML klassidiagrammi salvestamine XMI formaati. XMI formaadis salvestatakse iga mudeli elemendi juurde lisaatribuut atribuut *xmi.id,* mis on sisuliselt transformeerimise käigus loodud sisemine loendaja. See atribuut on tarvilik näiteks klassidevaheliste seoste taastamiseks klasside vahel (XML on puukujuline struktuur, mudel enamasti mitte).

5.3. Transformatsioonide taaskasutamine

Transformatsioonide taaskasutatavus on oluline nõue tuleviku tööriistadele. Taaskasutamiseks on kaks peamist võimalust: transformatsiooni spetsifitseerimine ning transformatsioonide kompositsioon. Spetsifitseerimine on analoogne Java pärimisega, spetsifitseerides saame transformatsiooni-definitsioonist osad reeglid üle defineerida. Näiteks võime kirjutada, et kui lähtemudelis on klassi atribuudi tüübiks mõni teine klass, siis tulemusena tuleb hoopis rakendada ühesuunalist seost selle klassiga (selline reegel

võimaldaks ise defineeritavate andmetüüpide kasutamist seoste taaskasutamise abil). Komposiit-transformatsioon koosneb mitmetest transformatsioonidest, mis on omavahel seotud loogiliste tehetega (*ja, või*).

5.4. Transformatsioonide järk-järguline rakendamine.

Transformatsioonide jälgitavus (ingl. k. *traceability*) on vajalik mudelite automaatseks sünkroniseerimiseks. Ühekordsest teisendusest saab ainult ühekordset kasu. Selleks, et programmeerija ei hülgaks modelleerimist on tarvis jälgida mudelite järk-järgulisi muudatusi.

Transformatsiooniks võivad olla teisenduskirjeldused (ingl. k. *mappings*) ja seosed (ingl. k. *relations*) [ACRTV03]. Seosed on deklaratiivsed, kahesuunalised, tüüpiliselt väljendatud sarnaselt OCL tõketega. Teisenduskirjeldused on tihtipeale ühesuunalised, kirja pandud mõnes imperatiivses keeles (näiteks UML Action Semantics).

Kasutaja ei taha teada kuidas transformatsioonid toimuvad, ta tahab näha tulemusi. Ülim eesmärk mida MDA peaks saavutama on kõikide seotud mudelite sünkroonishoidmine, isegi kui kasutaja modifitseerib koodi. Selle jaoks on esiteks tarvilik kahesuunaliste transformatsioonireeglite olemasolu. Lisaks peab transformatsioonitööriist olema suuteline tuvastama pisikesi muudatusi, ning seejärel kajastama need ja ainult need muudatused ka teistes mudelites.

Tööriista võimalikud reageeringud mudeli väikesele muutmisele oleksid: teha mitte midagi (mudelid väljuvad sünkroonsest seisust), paluda kasutaja sekkumist (paluda tal muuta ka teised mudelid) ning transformeerida seotud mudelid automaatselt. Viimatinimetatu teostamiseks on pakutud delta-transformatsioonide idee [TC03]. Sisuliselt on tegu täpselt samasuguse transformatsioonireegliga kui iga teisegagi, lihtsalt sellise reegli sisuks on info mingi väikese muudatuse kohta. Delta-transformatsioon on võrreldav *diff* käsu kasutamisega (mudeli muutuse väljaselgitamine) ning tulemuse tõlgendamise ja rakendamisega (muudatuste sisseviimine). Näiteks klassidiagrammile atribuudi lisamine peaks relatsioonilises mudelis kajastuma välja lisandumisega vastavasse andmemudelisse ning see omakorda SQL lause *add column* rakendamisena reaalses süsteemis.

5.5. MDA arengusuunad

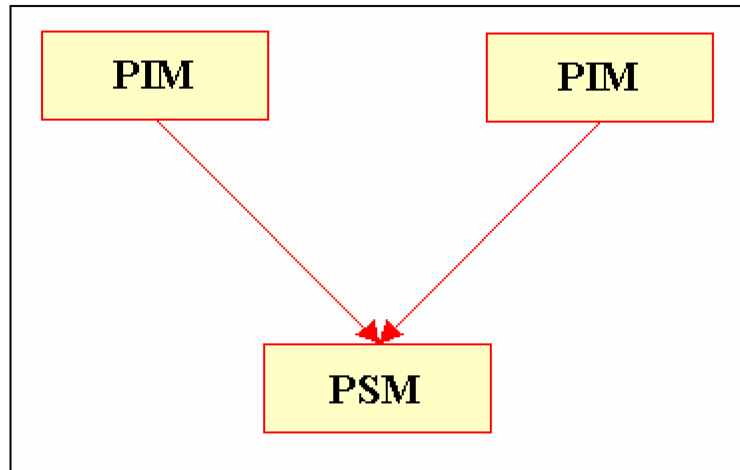
MDA ei ole praegusel hetkel valmis demonstreerima oma täit potentsiaali. Ometigi on olemas palju kasulikke lahendusi, mis oma olemuselt on MDA-le küllaltki lähedased.

MDA esmane lähendus reaalselt töötavate lahenduste vallas on RTE (ingl. k. *Round-trip Engineering*). Viimane on tarkvaraarenduse selline meetod, mis põhineb mudel-kood iteratsioonidel. Mudelite põhjal koostatakse võimalikult suur osa standardsest koodist, mida hiljem käsitsi täiendatakse. Vajadusel muudetakse taas mudelit. Spetsiaalsed UML tööriistad lubavad modelleerimiskeskkonnas loodud mudelitest nupuvajutusega genereerida meetodite päised, meetodite sisu jaoks jäetakse vastavad spetsiaalselt markeeritud kohad koodis, kuhu võib muudatusi teha. UML tööriist säilitab sellistesse kohtadesse kirjutatud koodi.

Järgmisel tasemel abivahend on selline UML tööriist, mis lihtsalt ei säilita meetodite sisusid, vaid suudab neid ka vajadusel muudatustega sünkroniseerida – näiteks kui meetodi sees kasutatakse mingit klassi, mille nime hiljem mudelis muudetakse, siis tööriist muudab nime ka koodis (RTE + rekodeerimine).

Selliste tööriistade puuduseks on töötamine konkreetse keele piirides: nad ei lahenda peatükis 1 tõstatatud lihtsate muudatuste probleemi, sest muudatusi oleks tarvis teha korraga mitmel platvormil. Nüüd peaks meile selge olema, et sääraste tööriistade tootjatel polegi tegelikult võimalik selliseid probleeme lahendada: neil pole vahendeid (keeli, infrastruktuuri) võimaldamaks praeguse tarkvaraarenduse parimate kontseptsioonide rakendamist mitme sihtplatvormiga korraga töötades.

Aspekt-orienteeritud programmeerimine (ingl. k. *Aspect-Oriented Programming*, AOP) on uus suund objekt-orienteeritud lähenemise kõrval. AOP on suunatud OOP piirangutele, peamiselt modulaarsuse kadumisele olukorras, kus on tarvis korraga silmas pidada paljusid teiseseid huvisid nagu logimine, turvalisus, vigade töötlus, jne. Selliste huvide rahuldamiseks kõikjal süsteemis on AOP ideaalne, sest AOP põhineb eri aspektide teineteisest sõltumatul arendamisel ja nende hilisemal kokkuühendamisel (ingl. k. *weaving*) [Wra03].



Joonis 5.1. Mudelite ühendamine

MDA ja AOP täiendavad teineteist, MDA vaatest on tegu eri aspektidega tegelevate mudelite kokkuühendamisega (ingl. k. *model merge*), Kui tüüpiline MDA arendusprotsess käib vertikaalses sihis (kõrgeimal kohal üks PIM), siis AOP annab meile selged piirid töötamaks horisontaalses sihis (vt joonis 5.1). Kõrvuti kontseptuaalse PIM mudeliga defineerime logimise PIM-i, turvalisuse PIM-i jne. Töötava süsteemi saame siis genereerida mudelite ühendamisel ja transformeerimisel PSM-deks.

Kokkuvõtteks

MDA on võimas vahend vähendamaks paljusid traditsioonilise tarkvaraarenduse probleeme, ta on tarkvaraarenduse raamistik, mille ideeks on modelleerimise tulemusena täisväärtusliku tarkvarasüsteemi genereerimine. MDA kasutuselevõtust räägitakse kui paradigmade vahetusest (ingl k. *paradigm shift)*, evolutsioonist tarkvaraarenduses. MDA raamistikus töötades on rõhuasetus täpsete mudelite koostamisel ja nende transformeerimisel koodiks, selle võimaldamiseks on tarvis allasuvate tööriistade hea koostöö.

Praeguseks on olemas mitmed standardid mudelitega manipuleerimiseks ning ka tööriistad mudeli põhjal koodi genereerimiseks, kuid loodud tarkvara ei saa enamasti koheselt kasutada täisväärtusliku programmina – tulemust tuleb käsitsi modifitseerida. Probleem on tingitud asjaolust, et me ei ole harjunud modelleerima piisavalt täpselt, samas ei ole meil ka mugavaid vahendeid mudelite teisendamiseks, ühendamiseks või valideerimiseks. Sellised tegevused saavad võimalikuks alles pärast ühtsete de-facto standardite kujunemist.

Kuigi MDA on alles oma arenemisejärgu algusetappides, on ta sellegipoolest paljutõotav. Juba praegu on näha selget kasu loodud standarditest ja infrastruktuurist – XMI formaat on laialt levinud ja huvilistel on võimalik seda kasutada programmeerija vaeva vähendamiseks. Selleks aga, et huviliste tehtud töö U-Coderi taoliste tööriistade näol ei jääks ainult ühekordseks projektiks, vaid oleks taaskasutatav ja kasulik kõigile huvilistele, tuleb luua vastavad ühtsed mudelitega manipuleerimise vahendid: keeled ning tööriistad. Alles siis avaldub MDA kogu potentsiaal.

Model-Driven Software Development

Andres Vilgota

Bachelor thesis

Abstract

The most common way of designing complex object-oriented applications is using UML as the basis of a software development process. Any kind of automated steps are welcome while taking the road from ideas and business rules to the delivery of working application. The concepts of problem domain are spread over variety of platforms and mismatches between different paradigms must be crossed. Model Driven Development (MDD), in particular OMG's Model Driven Architecture (MDA) is an emerging solution to the problems of traditional software development.

In this paper we demonstrate a practical MDD tool called U-Coder [Vil03]. U-Coder minimizes the effort needed when developing persistence-capable Java objects with relational database as the underlying data storage. Using *Umbrello UML Modeller*'s class diagrams as the basis, just a little effort is needed to generate consistent Java code for the business objects, table descriptions for the relational database and a mapping for object-relational mapping tool. U-Coder is especially useful for instant prototyping.

In this paper we also introduce a future process called Model Driven Architecture proposed by Object Management Group, the leading contributor to MDD. We demonstrate the essence of MDA, which solves many problems of traditional software development in a natural manner. We also give a brief overview of core components of MDA: standards called UML, XMI, OCL, CWM and MOF. These standards address the needs of next generation software development tools and are already commonly used.

The most important missing link of MDA today is a consistent way of transforming models, in the time of writing this thesis, a standard called QVT (Query/Views/Transformations) is still under development. Though QVT is not adopted and fully developed yet, a generic introduction to model transformations is also given.

Viited

[ACRTV03] B. K. Appukuttan, T. Clark, S. Reddy, L. Tratt, R. Venkatesh. „A model driven approach to model transformations“. (http://tratt.net/laurie/research/publications)

[Castor] ExoLab Groupi avatud lähtekoodil põhinev andmete sidumise raamistik Java jaoks. (http://castor.exolab.org)

[CWM] OMG Common Warehouse Metamodel. (http://www.omg.org/cwm)

[Fow02] Martin Fowler. „Refactoring. Improving the Design of Existing Code.“ Addison Wesley 2002.

[Fow03] Martin Fowler Wiki Pages. (http://martinfowler.com/bliki/UmlMode.html)

[Fra04] David Frankel. „Domain-Specific modeling and Model Driven Architecture.“, MDA Journal, January 2004.

[Hibernate] Hibernate - Relational Persistence for Idomatic Java. (http://www.hibernate.org)

[IO02] Interactive Objects Software. The Architectural IDE for MDA. (http://www.omg.org/mda/mda_files/iO_ArcStyler_OMG_MDA_USA_30_Min_04Apr02.ppt)

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. „MDA Explained.“ Addison-Wesley, 2003, ISBN 0-321-19442-X

[LL00] Mati Littover, Lauri Lubi. UML keele sõnastik. (http://www.cc.ioc.ee/uml/). 2000.

[MDA] OMG Model Driven Architecture. (http://www.omg.org/mda)

[MOF] OMG Meta Object Facility (http://www.omg.org/mof)

[Poo03] John Poole. „Model-Driven Data Warehousing." 2003. (http://www.cwmforum.org/paperpresent.htm)

[PR03] Richard Page, Alec Radjenovic. „Towards Model Transformation with TXL." 2003. (http://www-users.cs.york.ac.uk/~paige/Writing/m4mda.pdf)

[QVT] Request for Proposal: MOF 2.0 Query/Views/Transformations. (http://www.omg.org/docs/ad/02-04-10.pdf)

[SOA] Web Services and Service-Oriented Architectures: Object-relational database products vendors. (http://www.service-architecture.com/products/object-relational_databases.html)

[TC03] Laurence Tratt, Tony Clark. „Issues surrounding model consistency and QVT." 2003. (http://tratt.net/laurie/research/publications)

[Vil03] Andres Vilgota. „Klassimudelil põhineva objekt-relatsioonilise andmebaasiga infosüsteemi loomine." Semestritöö 2003. (http://www.ut.ee/~vilgota/)

[WK03] Jos Warmer, Anneke Kleppe. „The Object Constraint Language: Getting Your Models Ready for MDA." 2003. ISBN: 0321179366

[Wra03] Dean Wramper. „The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture." 2003. (http://www.aspectprogramming.com/papers/AOP and MDA.pdf)

[XPATH] W3C, XML Path Language (XPath). Version 1.0 1999. (http://www.w3.org/TR/xpath)

[XSLT] W3C, XSL Transformations (XSLT). Version 1.0. 1999. (http://www.w3.org/TR/xslt)

Lisad

Lisa 1. Näide U-Coderi tüübiteisenduse XML failist

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<dataTypes>
	<type UMLName="id">
		<asJava>java.lang.Integer</asJava>
		<asCastorJava>integer</asCastorJava>
		<asSQL>int(11)</asSQL>
		<asCastorSQL>integer</asCastorSQL>
	</type>
	<type UMLName="string">
		<asJava>String</asJava>
		<asCastorJava>string</asCastorJava>
		<asSQL>varchar(255)</asSQL>
		<asCastorSQL>varchar</asCastorSQL>
	</type>
	<type UMLName="bool">
		<asJava>Boolean</asJava>
		<asCastorJava>java.lang.Boolean</asCastorJava>
		<asSQL>char(1)</asSQL>
		<asCastorSQL>char[NY]</asCastorSQL>
	</type>
	<type UMLName="int">
		<asJava>java.lang.Integer</asJava>
		<asCastorJava>integer</asCastorJava>
		<asSQL>int(11)</asSQL>
		<asCastorSQL>integer</asCastorSQL>
	</type>
	<type UMLName="date">
		<asJava>java.util.Date</asJava>
		<asCastorJava>date</asCastorJava>
		<asSQL>timestamp(11)</asSQL>
		<asCastorSQL>timestamp</asCastorSQL>
	</type>
	<type UMLName="blob">
		<asJava>String</asJava>
		<asCastorJava>string</asCastorJava>
		<asSQL>mediumblob</asSQL>
		<asCastorSQL>varchar</asCastorSQL>
	</type>
</dataTypes>
```