VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Modeliais grįsto programų kūrimo metodo taikymas dokumentų valdymo sistemų kūrime

Using Model Driven Development in the creation of Document Management Systems

Magistro baigiamasis darbas

Atliko: Laurius Vaitkevičius
(parašas)

Darbo vadovas: asist. Viktoras Golubevas
(parašas)

Recenzentas: asist. Tomas Gžegožas Lipnevičius
(parašas)

Vilnius – 2008

Santrauka

Modeliais grįstas programų kūrimas yra vienas naujausių abstrakcijos pakėlimo būdų programų kūrimo procese. Jis kai kuriais atvejais leidžia pagreitinti programų kūrimo procesą net iki 10 kartų. Šiame darbe analizuojamas dviejų modeliais grįstų metodų taikymas dokumentų valdymo sistemų kūrime – tai apibendrinta modeliais grįsta architektūra (MDA) ir dalykinei sričiai specifinis modeliavimas (DSM). Darbe rasite tiek teorinius šios temos aspektus, tiek ir praktinius bandymus. Teorinėje dalyje apibrėžta dokumentų valdymo sistema – jos pagrindinės funkcijos bei savybės. Taip pat išskirti pagrindiniai modeliais grįstų metodų privalumai ir trūkumai. Praktinių bandymų metu atsiskleidžia tikroji metodams skirtų palaikyti įrankių vertė. MDA įrankių atveju realiai pamatysite, kokios pagalbos galima laukti iš jų spartinant programų kūrimą. Tuo tarpu analizuojant DSM galimybes buvo sukurtas pilnai funkcionuojantis įrankis skirtas dokumentų valdymo sistemoms kurti, kuris leidžia sumodeliuoti dokumentų valdymo sistemų duomenų sluoksnį, sugeneruoti jo kodą bei duomenų bazių schemą. DSL kalbos kūrimo žingsnių aprašymas gali būti naudojamas kaip pagrindas kitų dalykinių sričių kalbų kūrimui, o sukurta kalba gali būti praplėsta iki tokio lygio, kokio reikia, ir naudojama dokumentų valdymo sistemų kūrime.

Raktiniai žodžiai: dokumentų valdymo sistema, modeliais grįstas programų kūrimas, modeliais grįsta architektūra, dalykinei sričiai specifinis modeliavimas.

Summary

Model Driven Development is one of the newest ways to increase level of abstraction in software development process. In some cases it allows to increase productivity up to 10 times. This work analyzes usage of two model driven methods in the creation of document management systems. They are, unified Model Driven Architecture (MDA) and Domain Specific Modeling (DSM). You will find both, theoretical and practical aspects of this topic. Theoretical part consists of a definition of document management system, its functions and characteristics. It also contains main advantages and drawbacks of model driven methods. During practical experiments the real value of method supporting tools is revealed. In case of MDA tools, you will see what help can you expect while increasing productivity. Whereas while analyzing DSM possibilities the fully functional tool for modeling document systems was created. It allows creating a model of a data layer, generating its source code and database scheme. The description of DSL creation steps can be used as a base for other domains, and created language can be extended as much as needed and used to create document management systems.

Keywords: Document Management System, Model Driven Development, Model Driven Architecture, Domain Specific Modeling.

Turinys

Įvadas....6
1. Programinės įrangos pakartotinis panaudojimas....8
2. Modeliais grįstas programų kūrimas....9
3. Dokumentų valdymo sistema....10
  3.1. Saugykla....10
  3.2. Meta duomenys....10
  3.3. Versijų palaikymas....11
  3.4. Indeksavimas....11
  3.5. Radimas....11
  3.6. Apsauga....12
  3.7. Darbų seka....12
  3.8. Bendras darbas....12
  3.9. Dokumentų valdymo sistemų modeliavimas....12
4. Modeliais grįsta architektūra (MDA)....13
  4.1. Bendrų meta duomenų svarba....14
  4.2. Meta duomenų integracija: CWM, UML, MOF ir XMI....16
    4.2.1. CWM....16
    4.2.2. UML....17
    4.2.3. MOF....18
    4.2.4. XMI....19
  4.3. MDA apibendrinimas....20
5. MDA praktinis naudojimas....20
  5.1. Sistemos modeliavimas su MDA....21
  5.2. MDA įrankių apibendrinimas....22
6. Specifinis sričiai modeliavimas....22
  6.1. Specifinė dalykinės srities kalba....22
  6.2. Dalykinės srities analizė....23
  6.3. Specifinės dalykinei sričiai modeliavimo kalbos pranašumai....23
  6.4. Kodo generavimas su DSM....25
  6.5. Paprasti generatoriai....26
  6.6. Modelis į kodą ar modelis į modelį....27
  6.7. Gero kodo generavimo savybės....28
  6.8. Gerosios DSM praktikos....28
  6.9. DSM apibendrinimas....29
7. DSM praktinis panaudojimas....31
  7.1. Dalykinės srities modelio aprašymas....31
  7.2. DSL kūrimas su DSL Tools....32
  7.3. Dalykinės srities modelio kūrimas....33
  7.4. DSL grafinė notacija....37
  7.5. Testinė dokumentinė sistema su DSL Tools....40
  7.6. Kodo generavimas....40
    7.6.1. Technologijos....41
    7.6.2. Generuojamų objektų identifikavimas....41
    7.6.3. Kodo generatorius....42
  7.7. DSL naudojimo apibendrinimas....43
Rezultatai ir išvados....44
Naudota literatūra....46
Priedas 1. MDA nuo platformos nepriklausomas modelis....48
Priedas 2. MDA specifinis platformai modelis....49
Priedas 3. MDA sugeneruotas kodas....50

Priedas 4. Dokumentų valdymo sistemos dalykinės srities modelis ... 62
Priedas 5. Dalykinės srities redaktorius ... 64
Priedas 6. C# kodo šablonai ... 65
A) Dokumento šablonas ... 65
B) Redakcijos šablonas ... 67
C) Klasifikatoriaus šablonas ... 70
D) Papildomos informacijos šablonas ... 71
Priedas 7. SQL šablonai ... 74
A) Dokumento šablonas ... 74
B) Redakcijos šablonas ... 75
C) Klasifikatoriaus šablonas ... 77
D) Papildomos informacijos šablonas ... 78
Priedas 8. NHibernate XML konfigūracijos šablonai ... 80
A) Dokumento šablonas ... 80
B) Redakcijos šablonas ... 82
C) Klasifikatoriaus šablonas ... 84
D) Papildomos informacijos šablonas ... 85
Priedas 9. DSL sugeneruotas C# kodas ... 87
Priedas 10. DSL sugeneruotas SQL kodas ... 91
Priedas 11. DSL sugeneruota NHibernate XML konfigūracija ... 93

Įvadas

Vienas iš didžiausių iššūkių, su kuriuo susiduria programų sistemų inžinerijos industrija yra pakartotinis programų arba jų dalių panaudojimas. Problema yra ta, kad laikas, žmonių darbas, pinigai ir kiti svarbūs resursai dažnai yra viršijami netgi tais atvejais, kuomet naujai kuriamos sistemos yra panašios funkcionalumu ir problemine sritimi į anksčiau sukurtas. Programų pakartotinio panaudojimo idėja yra ta, kad kuriant naujas programas būtų panaudojami jau sukurtų produktų privalumai. [DKV00]

Abstrakcijos lygio pakėlimas yra viena iš fundamentalių technikų programų kūrime. Iš tiesų, dauguma kūrėjų daro tai kiekvieną dieną, bet dažnai nesusimasto apie šį terminą. Abstrakcijos lygio pakėlimas yra būdas pakartotinai panaudoti žinias, kaip spręsti pasikartojančias problemas tam tikroje srityje, iš anksto sukuriant abstrakcijas ir vėliau jas pritaikant. [Gre04]

Vienas iš neseniai atsiradusių būdų abstrakcijos lygiui pakelti programų kūrimo procese yra modeliais grįstas programų kūrimas (angl. Model Driven Development, toliau MDD). Pasak kai kurių literatūros šaltinių, šis būdas yra pirmas tikras abstrakcijos lygio pakėlimas nuo trečios kartos programavimo kalbų pasirodymo. Jis leidžia pakartotinai panaudoti jau nebe kažkokios konkrečios programavimo kalbos komponentus, bet visą programos logiką, kuri gali būti visiškai nepriklausoma nuo realizacijos platformos. Šiuo metu yra du vyraujantys modeliavimo metodai – tai apibendrinta modeliais grįsta architektūra (angl. Model Driven Architecture, toliau MDA) naudojanti Object Management Group (toliau OMG) sukurtą apibendrintą modeliavimo kalbą (angl. Unified Modeling Language, toliau UML) ir dalykinei sričiai specifinis modeliavimas (angl. Domain Specific Modeling, toliau DSM) naudojantis dalykinei sričiai specifinę kalbą (angl. Domain Specific Language, toliau DSL).

Nors abiejų metodų pagrindinė idėja yra panaši – programos kuriamos naudojant modelius, tačiau ideologija skiriasi. Vienas metodas orientuojasi į apibendrintus dalykus, kitas į specifinius tam tikram uždaviniui. Dėl šios priežasties skiriasi ir metodų panaudojimo galimybės. Pagrindinė darbo mintis – palyginti šiuos du modeliavimo metodus. Vieno iš metodų esmė yra specializacija, todėl norint atlikti tyrimą, reikėjo pasirinkti konkrečią dalykinę sritį. Tam tikslui buvo pasirinkta sritis, kurioje turiu pakankamai kompetencijos ir žinių – tai dokumentų valdymo sistemos. Tuomet jau buvo galima iškelti ir darbo tikslus.

Pagrindinis darbo tikslas – ištirti modeliais grįstų metodų panaudojimo galimybes kuriant dokumentų valdymo sistemas. Kad pasiekti šį tikslą, darbe keliami tokie uždaviniai:

- apsibrėžti dokumentų valdymo sistemą – dalykinę sritį, kuriai bus tiriamos modeliais grįstų metodų panaudojimo galimybės;

- nustatyti dalykinės srities aspektus, kuriuose modeliais grįsti metodai duotų didžiausią efektą;
- ištirti teorines modeliais grįstų programų kūrimo metodų galimybes;
- ištirti MDA ir DSL įrankių pasiūlą, jų ypatybes bei panaudojimo galimybes kuriant dokumentų valdymo sistemas.

Kaip darbo rezultatai, bus pateiktos rekomendacijos ir galimi metodų, bei tiems metodams įgyvendinti skirtų įrankių panaudojimo scenarijai.

Siekiant užsibrėžtų darbo tikslų bei įgyvendinant iškeltus uždavinius darbe derinama informacija rasta įvairiuose informacijos šaltiniuose, bei praktinis konkretiems metodams skirtų įrankių išbandymas.

Visų pirma apžvelgiami teoriniai nagrinėjamos temos aspektai, tokie kaip pakartotinis panaudojimas, modeliais grįstas programų kūrimas, bei apibrėžiama nagrinėjama dalykinė sritis – dokumentų valdymo sistemos. Tuomet nagrinėjamos atskirų modeliais grįstų metodų ypatybės ir jų panaudojimo konkrečioje dalykinėje srityje galimybės, privalumai bei trūkumai.

Dauguma darbe naudojamų informacijos šaltinių, kaip ir patys metodai, yra gana nauji – nesenesni nei penkių metų senumo. Bet taip pat nevengiami ir senesni šaltiniai, kuriuose analizuojami fundamentalūs dalykai. Praktiniams metodų bandymams naudojami įrankiai, kurių buvo ieškoma specializuotuose interneto svetainėse tokiose kaip www.objectsbydesign.com ir www.dsmforum.org. Pabaigoje pateikiami rezultatai, išvados bei rekomendacijos.

1. Programinės įrangos pakartotinis panaudojimas

Programinės įrangos pakartotinis panaudojimas yra procesas, kuomet egzistuojanti programinė įranga naudojama kuriant naują programinę įrangą. Pakartotinio naudojimo tikslas – sumažinti poreikį kurti naują programinę įrangą nuo nulio, jei tinkama programinė įranga jau egzistuoja. Dauguma šiuo metu atliekamų tyrimų orientuojasi į pakartotinio panaudojimo kliūtis ir į metodų kūrimą, kurie padėtų išvengti šių kliūčių. [DKV00]

Labai svarbi pakartotinio panaudojimo dalis yra abstrakcija. Sunku panaudoti programinius artefaktus, jei nėra pakankamo abstrakcijos lygio. Nesvarbu, ar tai būtų funkcijos, klasės, ar komponentai, visada yra rizika, kad jie bus per daug specifiniai. Abstrakcija turi du lygius: abstrakti specifikacija ir abstrakti realizacija. Kuomet programinė įranga yra sluoksniuota, tuomet techninė įranga yra žemiausiame sluoksnyje, o kiekvienas kitas sluoksnis turi apatinį sluoksnį – realizaciją, ir viršutinį sluoksnį – specifikaciją. Taigi, vieno sluoksnio realizacija yra kito žemesnio sluoksnio specifikacija. Ši specifikacija apibūdina, ką sluoksnis daro, tuo tarpu realizacija apibūdina, kaip tai turi būti padaryta. [Kru92]

Pakartotinai panaudojamas artefaktas privalo turėti glaustą abstrakciją, kas reiškia, kad artefakto panaudojimas gali būti lengvai rastas ir identifikuotas. Abstrakcija taip pat turi turėti kintamąjį, kuris yra keičiamas pakartotinio panaudojimo metu. Artefakto interfeisas taip pat yra abstrahuotas taip, kad vidinės artefakto detalės būtų nesvarbios procesui, kuris integruoja panaudojamą komponentą naujoje sistemoje. [RB05]

Kita svarbi pakartotinio panaudojimo proceso dalis yra parinkimas. Tai yra proceso dalis, kurioje iš tiesų pasirenkami pakartotinai panaudojami artefaktai. Artefakto pasirinkimas priklauso nuo turimos abstrakcijos. Aukšto lygio kalboje, artefakto specifika dažniausiai bus aukšta, o tai reiškia, kad yra žemesnio lygio abstrakcija ir todėl yra mažiau pasirinkimo variantų. Žemo lygio kalboje yra priešingai – keletas artefaktų tinka tam pačiam pakartotiniam panaudojimui. Kūrėjas yra labiausiai atsakingas pasirinkimo procese, kadangi jam reikia nuspręsti, ar kodo gabalas gali būti keičiamas ir panaudojamas. [Kru92]

Po pasirinkimo eina specializacija. Šiame žingsnyje reikia apibendrintos kalbos konstrukcijas užpildyti specializuotomis konstrukcijomis, kad būtų sukurta labiau specifinė programa. Naudojamos kalbos semantika diktuoja specializacijos žingsnius. Kadangi kiekviena kalba turi savo semantiką ir taisykles, specializacijos žingsnis turi skirtingas veiklas skirtingoms kalboms. Veiksmai specializacijos žingsnyje turės įtakos veiksmas integracijos žingsnyje. [RB05]

Integracijos metu, didelio mąsto kalbos konstrukcijos yra sukomponuojamos, kad suformuoti galutinę sistemą. Šias konstrukcijas sudaro procedūros, paketai ir moduliai. Naujai

sukurto komponento integravimas į sistema ir pakartotinai panaudojamo komponento integravimas sudaro tiek pat darbo.

Pagrindiniai efektyvaus programų pakartotinio panaudojimo reikalavimai yra tokie:

- jis turi sumažini kognityvinį atstumą tarp sistemos koncepcijos ir realizacijos;
- turi būti lengviau panaudoti egzistuojančius artefaktus, nei juos sukurti iš naujo;
- kad panaudoti artefaktą, jūs turite žinoti, ką jis daro;
- yra keletas pakartotinio panaudojimo metodų, ir geriausias turi būti pasirinktas atsižvelgiant į programos prigimtį ir jos sudėtingumo lygį. [Kru92]

2. Modeliais grįstas programų kūrimas

Iš esmės, nuo trečios kartos programavimo kalbų pristatymo vėlyvaisiais 1950-aisiais metais, programavimo technologijos esmė praktiškai nepasikeitė. Nors nuo tada buvo sukurta keletas naujų programavimo paradigmų, tokių kaip struktūrinis ir objektinis programavimas, ir buvo nemažai nuveikta, kad apšlifuoti detales, abstrakcijos lygmuo išliko praktiškai tas pats. „If" arba „Loop" sakinys moderniose programavimo kalbose, tokiose kaip Java ar C++, praktiškai niekuo nėra pranašesnis už tuos pačius sakinius ankstyvojoje Fortran kalboje. Netgi daug žadantys mechanizmai, tokie kaip klasės ir paveldėjimas, kurie turi galimybę sukurti aukštesnes abstrakcijos formas, išlieka beveik neišnaudojami. Objektai, pavyzdžiui, yra nužeminti iki reliatyviai paprastos abstrakcijos priklausančios vienai adresų erdvei suderinamai tik su kalbos smulkumu ir abstrakcijos lygiu, kuriai jie priklauso. [Sel03]

Dauguma praktikų jau prarado viltį, kad bus koks nors esminis šuolis programavimo technologijose. Vietoje to, jie deda visas viltis į proceso gerinimą. Tai iš dalies paaiškina didelį susidomėjimą tokiais metodais, kaip Extreme Programming (XP) ir Rational Unified Process (RUP). Nors gero proceso laikymasis yra kritinis norint pasiekti didelės sėkmės, tačiau jis greitai išnaudos visas šiandienos technologijų galimybes. Tačiau programų kūrimas susideda, visų pirma, iš minčių išreiškimo, kas reiškia, jog mūsų gebėjimas panaudoti tinkamas priemones yra praktiškai apribotas mūsų vaizduotės, o ne fizikos dėsnių. Šios galimybės išnaudojimas yra viena esminių modeliais grįsto kūrimo idėjų ir viena iš priežasčių, kodėl šis būdas yra pirmas tikras kartos pakeitimas programavimo technologijose nuo kompiliatorių atsiradimo. [Sel03]

Modeliais grįsta inžinerija (angl. Model Driven Engineering, toliau MDE) remiasi sistematiniu modelių naudojimu kaip pirminiu inžineriniu artefaktu per visą inžinerijos gyvavimo ciklą. MDE gali būti pritaikyta programų, sistemų ir duomenų inžinerijai. Modeliai laikomi pirmos klasės esybės. Žinomiausia MDE iniciatyva yra OMG sukurta modeliais grįsta architektūra (MDA). [WK06]

Kas liečia programų kūrimą, modeliais grįstas kūrimas remiasi kūrimo metodais paremtais programų modeliavimu kaip pirmine išraiškos priemone. Kartais modeliai konstruojami iki tam tikro detalumo lygio, ir tuomet kodas rašomas rankiniu būdu kaip atskiras žingsnis. Kartais išbaigti modeliai yra sukuriami kartu su vykdymo veiksmais. Kodas sugeneruojamas iš modelių gali būti tik griaučiai, arba pilni, paruošti platinimui produktai. Pasirodžius UML, MDD tapo labai populiarus. Jį daug kas naudoja, sukurta daug įrankių. Dar labiau pažengę MDD tipai išsiplėtė iki industrijos standartų, kurie įgalina kurti vieningas programas ir rezultatus. Besivystanti MDD padidino susidomėjimą architektūra ir automatika. [WK06]

MDD technologijos su didesniu susidomėjimu architektūra ir susijusiu automatizavimu lėmė aukštesnius abstrakcijos lygius programų kūrime. Ši abstrakcija iškelia paprastesnius modelius su didesniu dėmesiu probleminei sričiai. Sujungus su vykdymo semantika tai pakelia bendrą automatizavimo galimybės lygį. [WK06]

Šiuo metu yra du vyraujantys modeliavimo metodai – tai jau minėta apibendrinta modeliais grįsta architektūra (MDA) naudojanti apibendrintą modeliavimo kalbą – UML, ir dalykinei sričiai specifinis modeliavimas (DSM) naudojantis dalykinei sričiai specifinę kalbą (DSL).

Vienas modeliais grįstų metodų orientuojasi į specifinius dalykinės srities aspektus, todėl prieš pradedant metodų panaudojimo galimybių tyrimą reikia apsibrėžti konkrečią dalykinę sritį.

3. Dokumentų valdymo sistema

Dokumentų valdymo sistemos yra programos, padedančios organizacijoms greitai ir patogiai valdyti savo dokumentų srautus elektroniniu pavidalu. Dokumentų valdymo sistemos suteikia daug privalumų, kurie pagerina organizacijos darbo efektyvumą. Šios sistemos paprastai pateikia saugyklos, meta duomenų, versijų palaikymo, apsaugos, o taip pat ir indeksavimo bei paieškos galimybes. Kai kurios dokumentinės sistemos gali turėti ir kitų naudingų funkcijų tokių, kaip darbų sekos ar bendro darbo užtikrinimas. Toliau apie kiekvieną iš svarbesnių funkcijų detaliau. [WK07]

3.1. Saugykla

Saugyklos arba dokumentų saugojimo funkcija dažnai apima tokių dalykų valdymą, kaip dokumentų saugojimo vietą, būdą, trukmę. Taip pat, jei reikia, užtikrina dokumentų migravimą iš vienos fizinės saugyklos į kitą, dokumentų atsarginių kopijų darymą ir atstatymą iš jų, bei dokumentų sunaikinimą. [WK07]

3.2. Meta duomenys

Paprastai prie kiekvieno dokumento be jo paties turinio saugoma ir tam tikra papildoma informacija palengvinanti dokumento paiešką dokumentinėje sistemoje, dar vadinama meta

duomenimis. Meta duomenyse gali būti dokumento data, numeris ar naudotojo sukūrusio dokumentą identifikatorius. DVS taip pat gali mokėti išrinkti meta duomenis iš dokumento automatiškai arba paprašyti naudotojo juos įvesti. Kai kurios sistemos taip pat naudoja optinį simbolių atpažinimą skenuotiems atvaizdams apdoroti, arba atlieka teksto ištraukimą iš elektroninių dokumentų. Gautas tekstas gali būti naudojamas palengvinant dokumento paiešką nurodant raktinius žodžius arba ieškant tam tikrų frazių tekste. Ištrauktas tekstas taip pat gali būti naudojamas kaip meta duomenų dalis patalpinta kartu su atvaizdu, arba kaip atskiras šaltinis. [WK07]

3.3. Versijų palaikymas

Versijų palaikymas yra procesas, kurio metu dokumentai yra paimami arba atiduodami į dokumentinę sistemą, leidžiant naudotojams gauti ankstesnes versijas ir tęsti darbą nuo pasirinkto taško. Dokumento versijos, arba kitaip dokumento redakcijos, yra naudingos dokumentams, kurie bėgant laikui reikalauja atnaujinimo, bet gali prireikti grįžti į ankstesnes versijas. [WK07]

3.4. Indeksavimas

Indeksavimas skirtas dokumentų paieškai. Indeksavimas gali būti ganėtinai paprastas, toks kaip unikalių dokumento identifikatorių sekimas, bet dažnai jis įgauna daug sudėtingesnę formą, suteikdamas klasifikavimą per dokumento meta duomenis ar net per žodžių indeksus paimtus iš dokumento turinio. Indeksavimas egzistuoja tam, kad palaikyti paieškos galimybę. Vienas iš kritinių greito radimo aspektų yra naudojama indeksavimo technologija. [WK07]

3.5. Radimas

Radimas – tai dokumento paėmimas iš saugyklos. Nors konkretaus dokumento radimo sąvoka yra paprasta, radimas elektroniniam kontekste gali būti gana sudėtingas ir galingas. Paprastas tam tikrų dokumentų radimas gali būti palaikomas leidžiant naudotojui nurodyti unikalų dokumento numerį, ir leisti sistemai naudoti paprastą indeksą, tam kad rasti dokumentą. Labiau lanksti paieška leidžia naudotojui nurodyti dalinius paieškos kriterijus tikėtinus meta duomenyse. Tai paprastai grąžins sąrašą dokumentų, kurie tenkina naudotojo paieškos kriterijus. Kai kurios sistemos suteikia galimybę nurodyti logines išraiškas turinčias keletą raktinių žodžių arba pavyzdinių frazių, kurių tikimasi tekste. Tokio tipo paieška gali būti palaikoma anksčiau sukurtų indeksų, arba gali atlikti daugiau laiko reikalaujančią paiešką per dokumento turinį, kad grąžintų potencialiai reikalingus dokumentus. [WK07]

3.6. Apsauga

Dokumentų apsauga yra labai svarbi daugumoje dokumentų valdymo sistemų. Prieigos reikalavimai tam tikriems dokumentams gali būti ganėtinai sudėtingi, priklausomai nuo dokumentų tipo. Kai kurios dokumentų valdymo sistemos turi teisių valdymo modulius, kurie leidžia administratoriams suteikti prieigos teises prie tam tikrų dokumentų tam tikriems asmenims arba asmenų grupėms. Pavyzdžiui, vieni asmenys gali dokumentus kurti, kiti patvirtinti, o treti tik peržiūrėti. [WK07]

3.7. Darbų seka

Darbų seka yra sudėtinga problema ir kai kurios dokumentinės sistemos turi integruotus darbų sekos modulius. Yra skirtingi darbų sekos tipai. Naudojimas priklauso nuo aplinkos, kurioje naudojama DVS. Rankinė darbų seka reikalauja, kad naudotojas peržiūrėtų dokumentą ir nuspręstų, kam jį persiųsti. Taisyklėmis grįsta darbų seka leidžia administratoriui sukurti taisykles, kurios nurodo dokumento kelią per organizaciją. Pavyzdžiui, sąskaita praeina pro patvirtinimo procesą ir tuomet persiunčiama sąskaitas apmokančiam departamentui. Dinaminės taisyklės leidžia sukurti šakas darbų sekoje. Paprastas to pavyzdys galėtų būti toks – jei sąskaitos suma yra žemesnė nei nustatyta, ji keliauja kitu keliu per organizaciją. [WK07]

3.8. Bendras darbas

Kiekviena, bent kiek rimtesnė dokumentų valdymo sistema, privalo turėti įdiegtą bendro darbo su dokumentais funkcionalumą. Autorizuotas naudotojas turėtų galėti pasiimti dokumentą ir su juo dirbti. Prieiga prie tokio dokumento turėtų būti uždrausta kitiems naudotojams, kol su juo yra atliekami tam tikri veiksmai. [WK07]

3.9. Dokumentų valdymo sistemų modeliavimas

Dokumentų valdymo sistemų naudą suvokia vis daugiau ir daugiau organizacijų pradedant mažomis bendrovėmis, baigiant didelėmis valstybinėmis įstaigomis. Todėl kuriama vis daugiau įvairių dokumentų valdymo sistemų. Dauguma jų turi nemažai vienodų ar panašių savybių, tokių kaip išvardintos anksčiau, tačiau priklausomai nuo organizacijos poreikių, sistemos šiek tiek gali skirtis. Vieniems gali reikėti vienokių meta duomenų, kitiems kitokių. Gali skirtis dokumentų klasifikavimas ar teksto saugojimo forma. Taip pat gali būti skirtingi apsaugos ar darbų sekos reikalavimai.

Kaip tik tokioms sistemoms, kurios turi daug bendrų savybių, bet kartu ir šiek tiek skiriasi modeliavimo metodai gali labai pasitarnauti didinant kūrimo efektyvumą. Peržiūrėjus dokumentų valdymo sistemų savybes galima teigti, kad modeliavimo metodus įmanoma pritaikyti praktiškai visoms išvardintoms funkcijoms. Paprasčiausia tai iliustruoti pateikiant

funkcijas ir UML diagramų tipus, kuriomis galima tas funkcijas sumodeliuot. Galima modeliuoti saugyklos schemas dislokavimo diagramomis, dokumentų meta duomenų aprašus – klasių diagramomis, darbų sekas – sekų diagramomis, bendrą darbą ar apsaugą būsenų diagramomis ir panašiai. Tačiau tyrimui tai būtų per daug plati sritis, todėl bus apsiribota tik baziniu funkcionalumu, be kurio negali apsieiti bet kokia dokumentų valdymo sistema. Tai bus dokumentų meta duomenys, jų atvaizdas saugykloje, dokumentų paieška, bei labai dažnai naudojamas versijų palaikymas.

Kad būtų dar paprasčiau, galima apsiriboti labai minimalia teisės aktų saugojimui skirta dokumentų valdymo sistemas, kurioje dokumentas gali turėti vieną arba daugiau redakcijų, prie kurios reikia saugoti jos numerį, datą ir tekstą. Taip pat turi būti saugoma dokumento publikavimo informacija, kurią sudaro leidinys, jo numeris bei data, ir dokumento tipas.

Nors modeliavimo vienas iš privalumų turėtų būtų nepriklausomumas nuo platformos, tačiau realybėje galutinis produktas vis tiek turės būti realizuotas tam tikroje aplinkoje. Todėl šiame darbe bus orientuojamasi į platformą, kurioje turiu didžiausią kompetenciją – tai Microsoft .NET 2.0, bei į Microsoft SQL Server 2005 duomenų bazę.

Kuomet žinomas uždavinys, galima pradėti jo sprendimo analizę skirtingais modeliavimo metodais. Pirma bus išanalizuota modeliais grįsta architektūra, tuomet DSL metodas, bei galiausiai pateiktos išvados. Labiausiai bus akcentuojama galutinio produkto generavimo galimybė.

4. Modeliais grįsta architektūra (MDA)

Neseniai OMG pristatė MDA iniciatyvą kaip sistemų specifikavimo ir bendradarbiavimo būdą pagrįstą formalių modelių naudojimui. Nuo platformos nepriklausomi modeliai (angl. Platform Independent Model, toliau PIM) išreiškiami apibendrinta modeliavimo kalba – UML. Nuo platformos nepriklausomi modeliai tuomet yra nuosekliai transformuojami į platformai specifinius modelius (angl. Platform Specific Model, toliau PSM) susiejat PIM su tam tikra realizacijos kalba arba platforma (pavyzdžiui Java arba .NET) naudojant formalias taisykles. [MDA01]

MDA koncepcijos branduolys yra keletas svarbių OMG standartų: The Unified Modeling Language (UML), Meta Object Facility (MOF), XML Meta Interchange (XMI) ir Common Warehouse Metamodel (CWM). Šie standartai apibūdina esminę MDA infrastruktūrą, ir ženkliai pasitarnavo dabartiniam sistemų modeliavimui. [MDA01]

MDA atstovauja evoliucinį žingsnį kelyje, kuriuo OMG apibrėžia bendradarbiavimo standartus. Labai ilgai bendradarbiavimas daugiausiai buvo paremtas CORBA standartais ir servisais. Įvairialypės programų sistemos bendradarbiauja per standartinių komponentų lygmenį.

MDA procesas, padeda formalius sistemos modelius į bendradarbiavimo problemos šaknis. Kas dar svarbiau dėl šio būdo, tai sistemos specifikacijos nepriklausomybė nuo realizacijos technologijos ar platformos. Sistemos apibrėžimas egzistuoja nepriklausomai nuo bet kokio realizacijos modelio ir turi formalius atitikmenis daugelyje galimų platformų (pavyzdžiui Java, XML, SOAP). [Poo01]

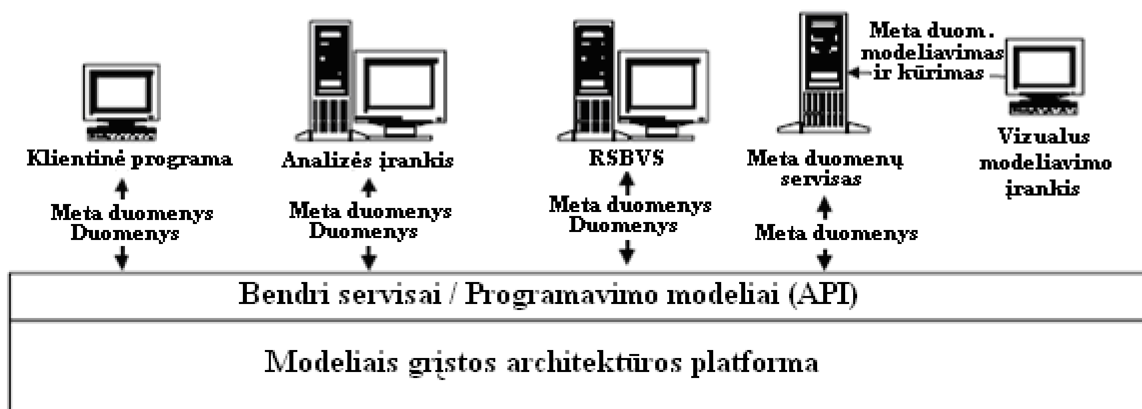
MDA turi reikšmingą užuominą į meta modeliavimo ir Adaptyvaus Objektų Modelių (angl. Adaptive Object Model, toliau AOM) discipliną. Meta modeliavimas yra pirminė specifikavimo veikla, kitaip meta duomenų modeliavimas. Bendradarbiavimas daugialypėse aplinkose yra daugiausiai pasiekiamas per bendrus meta duomenis, ir visa dalinimosi ir meta duomenų supratimo strategija susideda iš modelių automatinio kūrimo, platinimo, valdymo ir interpretavimo. AOM technologija suteikia dinaminę sistemos elgseną pagrįstą tokių modelių interpretavimu vykdymo metu. Architektūros paremtos AOM labiausiai galinčios bendradarbiauti, lengvai plečiamos vykdymo metu, ir visiškai dinamiškos bendros elgsenos specifikacijos terminais, t.y., jų funkcionalumas neapribotas iš anksto suprogramuota logika. [Poo01]

Pagrindiniai MDA standartai (UML, MOF, XMI, CWM) sudaro pagrindą nuoseklių schemų sudarymui, skirtų modelių kūrimui, platinimui ir valdymui modeliais grįstos architektūros viduje. Taip pat yra labai pagirtinas siekis industrijoje realizuoti šiuos standartus Java platformai (pavyzdžiui, PIM standartų atitikmenys PSM, kuri konkrečiai būtų Java platforma). Tai protinga realizacijos strategija, kadangi kūrimas ir integracija yra labai susiję per bendrus platformų servisus ir programavimo modelius (interfeisus arba API), pateikiamus kartu su Java platforma. Java 2 Enterprice Edition (J2EE) tapo vienu pirmaujančių industrijos standartų realizuojant ir platinant komponentais pagrįstas, išskirstytas programas daugiasluoksnėse į Internetą orientuotose aplinkose. Dabartinės Java Community Process pastangos sukurti grynai Java programavimo modelius realizuojančius OMG standartus J2EE standarto API (pavyzdžiui, JMI, JOLAP ir JDM) dar labiau išplečia meta duomenimis grįstą bendradarbiavimą tarp išskirstytų programų. [Poo01]

4.1. Bendrų meta duomenų svarba

Meta duomenys yra kritiniai visiems bendradarbiavimo aspektams bent kokioje daugialypėje aplinkoje. Iš tiesų, meta duomenys yra pirminės priemonės, kuriomis bendradarbiavimas yra pasiekiamas. Bendradarbiavimas yra labiausiai paruoštas per standartinį API, bet labiausiai reikalauja bendrų meta duomenų kaip sistemos semantikos ir gebėjimų apibrėžimo. Bet kuri MDA pagrįsta sistema privalo turėti galimybę saugoti, valdyti ir platinti tiek programos, tiek ir sistemos lygio meta duomenis (taip pat ir pačios aplinkos apibūdinimus).

Programos, įrankiai, duomenų bazės ir kiti komponentai prisijungia prie aplinkos ir atranda meta duomenų aprašymus esančius aplinkoje. Panašiai ir komponentas arba produktas įdėtas į aplinką, taip pat gali išplatinti savo meta duomenis likusiai aplinkos daliai. Šis scenarijus pavaizduotas paveiksle 1. [Poo01]



1 pav. MDA realizacijos pavyzdys [Poo01]

Turint daug bendrų aprašančių meta duomenų, galima programų bendradarbiavimą organizuoti labai specifiniais būdais:

- Duomenų apsikeitimas, transformacija ir tipų atitikmenų radimas tarp skirtingų duomenų resursų gali būti grindžiamas formaliais, nuo produkto nepriklausomais meta duomenų transformacijos, duomenų tipų ir tipų atitikmenų sistemos aprašymais.
- Schemų generacija gali būti paremta bendraisiais meta duomenų schemų aprašymais. Pavyzdžiui, tiek reliacinė duomenų bazė, tiek ir OLAP serveris gali konstruoti savo vidinius modelių pavaizdavimus, pagal standartą, meta duomenimis paremtas dimensijos aprašymas yra išplatinamas į aplinką. Turint bendrus meta duomenų aprašymus, įmanoma apsikeisti duomenimis tarp posistemių, kadangi jos turi bendrą supratimą, ką reiškia duomenys.
- Verslo logikos ir pavaizdavimo funkcijos gali panaudoti meta duomenis apdorojant ir formuojant duomenis analizei ir pavaizdavimui. Meta duomenų aprašai suteikia „aukštesnį reikšmingumo lygį“ duomenų elementams, kurių reikia analizuojantiems ir pranešantiems naudotojams, tam kad suvokti į ką duomenys rodo ir ką reiškia.
- Programiniai komponentai be jokių išankstinių žinių apie vienas kito galimybes, interfeisus ir duomenų pavaizdavimus gali bendradarbiauti, kaip tik jie apsikeis savo meta duomenimis, kuriuose kiekvienas paplatins savo galimybes ir tai gaus iš kitų. Šis žinių apsikeitimas ne visuomet turi būti pilnas, bet tiek kad komponentai suvoktų apie vienas kito gebėjimus. Trūkstant specifinių žinių komponentai gali pasikliauti

standartinėmis reikšmėmis, arba gali kreiptis į kokį nors kitą informacijos šaltinį, kad užpildyti žinių trūkumus. [Poo01]

MDA paremtos sistemos nereikalauja, kad vidiniai meta duomenų pavaizdavimai programose, įrankiuose ir duomenų bazėse būtų modifikuojami, kad atitiktų bendrus apibrėžimus. Produktui specifiniai vidiniai ir programavimo modeliai lieka tokie, kokie yra. Bendrus meta duomenis sudaro pritaikyti išorei aprašymai, kuriais keičiamasi tarp dalyvaujančių komponentų. Šie išoriniai aprašymai yra pilnai suprantami kitų komponentų, kurie sutinka su meta modelį aprašančiais meta duomenimis (pavyzdžiui, CWM). Išoriniai aprašymai yra labai bendri, bet taip pat laiko pakankamai semantikos baigtumo (atsižvelgiant į probleminę sritį), ir taip pat yra suprantami plataus dalyvių rato. Produktui specifiniai duomenys, kurie netelpa į bendrą modelį yra apdorojami per išplėtimo mechanizmus, kurie aprašomi kaip dalis bendrinio modelio (pavyzdžiui, UML išplėtimo mechanizmai, tokie, kaip stereotipai, apribojimai ir panašiai). [Poo01]

Kad užtikrinti meta duomenų supratimą visuose komponentuose, MDA paremta sistema privalo standartizuoti savo komponentus pagal šiuos kriterijus:

- Formali kalba (sintaksė ir semantika) meta duomenų vaizdavimui.
- Apsikeitimo formatas meta duomenų apsikeitimui ir viešinimui.
- Meta duomenų prieigos ir atradimo programavimo modelis. Jis turi įtraukti bendrinius programavimo gebėjimus susitvarkyti su nežinomos prigimties meta duomenimis.
- Anksčiau paminėtų aspektų išplėtimo mechanizmai.
- Papildomas meta duomenų servisas, kur guli paviešinti meta duomenys. [Poo01]

4.2. Meta duomenų integracija: CWM, UML, MOF ir XMI

Sėkmingos integracijos ir bendradarbiavimo raktas slypi protingame meta duomenų naudojime ir valdyme tarp visų programų , platformų, įrankių ir duomenų bazių. Meta duomenų valdymas ir integracija gali būti realizuota OMG MDA standartais: CWM, MOF, UML ir XMI. [Poo01]

4.2.1. CWM

Common Warehouse Metamodel apibrėžia meta modelį (duomenų modelio modelį) į kurį įeina tiek verslo, tiek ir techniniai meta duomenys, kurie dažniausiai sutinkami duomenų saugojime ir verslo analizės srityse. Jis naudojamas kaip pagrindas keičiantis meta duomenų atvejais tarp daugialypių, skirtingų tiekėjų programų sistemų (pavyzdžiui, integruojant duomenų talpyklų ir verslo analizės informacijos tiekimo grandinę). Sistemos, kurios supranta CWM meta modelį keičiasi meta duomenimis, kurie yra vieningi su meta modeliu.

CWM iš tiesų susideda iš daugelio vieningų meta modelių atstovaujančių duomenų išteklius, analizę, saugojimo valdymą ir tipinių duomenų saugojimo ir verslo intelektinės aplinkos komponentų pagrindą. Duomenų išteklių meta modeliai palaiko galimybę modeliuoti išliekamuosius ir neišliekamuosius duomenų išteklius, įskaitant ir reliacines duomenų bazes, į įrašus orientuotas duomenų bazes, XML bei objektais paremtus duomenų resursus. CWM analizės sluoksnis apibrėžia duomenų transformacijos meta modelius, OLAP, informacijos vaizdavimą/ataskaitų generavimą, verslo nomenklatūrą ir duomenų apdorojimą. Saugyklų valdymo sluoksnis susideda iš meta modelių atitinkančių standartinius saugyklų procesus, veiklų sekimą ir grafikų sudarymą (pavyzdžiui, kasdieniniai pakrovimai). Galiausiai, pamatinis meta modelis palaiko įvairių bendrų elementų ir servisų specifikavimą, tokių kaip duomenų tipai, tipų sistemų atitinkamumai, abstraktūs raktai ir indeksai, išraiškos, verslo informacija, ir komponentais grįstą programų kūrimą. [CWM]

CWM atstovauja modeliais grįstą duomenų apsikeitimo būdą tarp programų sistemų. Meta duomenys, kuriais dalinasi skirtingi produktai, yra formuluojami duomenų modelio terminais, kuris yra vieningas su vienu ar daugiau CWM meta modelių. Produktais eksportuoja meta duomenis suformuluodamas savo vidinių meta duomenų struktūrų modelį CWM nurodytu formatu. Panašiai, produktas importuoja meta duomenis naudodamas su CWM suderintą modelį ir susiedamas jį su savo vidiniais meta duomenimis. [Tol00]

Meta modelių kolekcija, kurią pateikia CWM yra pakankamai visapusiška, kad būtų galima modeliuoti visą duomenų saugyklą. Naudojant CWM įrankius, duomenų saugyklų atvejis gali būti sugeneruotas tiesiai iš saugyklos modelio. Kiekvienas skirtingas įrankis naudoja tas modelio dalis, kurios jam yra naudingos. Pavyzdžiui, reliacinės duomenų bazės serveris naudos reliacinę modelio dalį. Panašiai, OLAP serveris ieškos OLAP meta duomenų modelio, ir naudos jį apibrėždamas daugiamatę schemą. [Tol00]

CWM modeliai turėtų būti labai bendriniai išoriniai bendrų meta duomenų pavaizdavimai. Meta duomenys, kurie nevisai atitinka CWM formatą (pavyzdžiui, įrankiui specifiniai meta duomenys, kuriais reikia keistis) yra apdorojami arba per standartinį išplėtimo mechanizmą pateikiamą CWM, arba per produktui specifines standartines reikšmes, naudotojo įvedimą, arba kitą platinimo apibrėžtą logiką. [Tol00]

4.2.2. UML

CWM išreiškiamas apibendrinta modeliavimo kalba UML – OMG standartine kalba skirta diskrečių sistemų modeliavimui. UML yra CWM apibrėžimo pavaizdavimo pagrindas, bet CWM taip pat išplečia UML meta modelį duomenų saugojimo ir verslo analizės srities elementais. [RJB98]

Konstruojant duomenų saugyklų modelius paremtus CWM, rekomenduojami vizualaus modeliavimo įrankiai (palaikantys UML arba kitą ekvivalenčią formalią notaciją), kadangi vizualūs sudėtingų meta duomenų struktūrų modeliai yra žmonių lengviau valdomi ir suvokiami nei jie būtų atvaizduoti kitu formatu, pavyzdžiui tekstiniu. Kita vertus, kadangi UML kalba turi tikslų apibrėžimą, vizualūs UML modeliai gali būti automatiškai išversti į kitą kalbą – tiek vizualią, tiek ir ne. Tai leidžia apsikeisti CWM modeliais įvairiose nuo platformos ir nuo įrankių nepriklausomuose formatuose, pavyzdžiui XML, o taip pat ir įrankiams specifinių meta duomenų konstravimą iš CWM modelių, pavyzdžiui CWM reliacinio modelio vertimas į SQL DDL sakinius, kurie iš tiesų sukuria duomenų bazės schemą. [RJB98]

4.2.3. MOF

Meta Object Facility yra OMG standartas apibrėžiantis bendrą abstrakčią kalbą meta modelių specifikavimui. MOF yra meta-meta modelio pavyzdys, arba meta modelio modelis, kartais vadinamas ontologija. [MOF]

MOF yra išskirtinai objektiškai orientuotas. Jis apibrėžia esminius meta modelių elementus, sintaksę ir struktūrą, kurie yra naudojami konstruojant objektiškai orientuotus modelius diskrečioms sistemoms. MOF tarnauja kaip bendras modelis tiek CWM, tiek ir UML meta modeliams. Papildomai MOF specifikacija suteikia:

- Abstraktų bendrinių MOF objektų modelį ir jų asociacijas.
- Taisyklių rinkinį skirtą susieti bet kurį MOF grįstą meta modelį su nuo kalbos nepriklausomais interfeisais. Šių interfeisų realizacija duotajam meta modeliui būtų naudojama prieiti ir modifikuoti bet kurį modelį paremtą šiuo meta modeliu.
- Taisykles apibrėžiančias MOF grįstų meta modelių gyvavimo ciklą, kompoziciją ir semantiką.
- Interfeisų hierarchiją, kurie apibrėžia bendrines operacijas ieškant ir manipuliuojant modeliais paremtais su MOF suderinamais meta modeliais, bet kurių atitikimo interfeisai nežinomi. [MOF]

MOF galia yra tame, kad jis leidžia skirtingus meta modelius, atstovaujančius skirtingas sritis, naudoti bendradarbiaujant. MOF prisilaikančios programos gali nieko nežinoti apie kai kurių dalykinės srities modelių specifinius interfeisus, bet vis tiek gali skaityti ir atnaujinti tą modelį naudodamos bendrines operacijas. [Poo01]

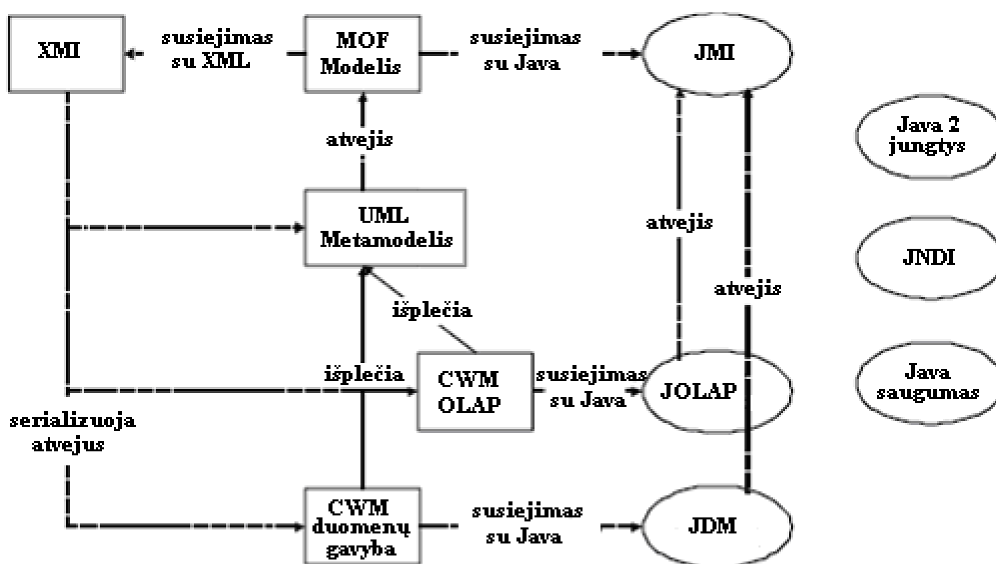
MOF semantika apibrėžia meta duomenų saugyklos servisus, kurie palaiko modelių konstravimą, radimą, apdorojimą ir atnaujinimą, kur modeliai suprantami kaip tam tikro meta modelio atvejai. MOF modelio gyvenimo ciklo semantikos palaikymas reiškia, kad MOF realizacija suteikia efektyvų meta duomenų kūrimo ir platinimo įrankį, kuomet yra apjungiamas

su vizualiu modeliavimu. Pavyzdžiui, naujai sukurti meta modeliai gali būti išsaugomi MOF saugykloje ir apjungiami su egzistuojančiais meta modeliais pagal MOF gyvavimo ciklo ir kompozicijos semantiką (paveldėjimą, skaidymą, įdėjimą ir panašiai). Modelio interfeisai ir standartinės realizacijos yra toliau tobulinamos įtraukiant papildomai suprogramuotos logikos, kuri gali būti parašyta ranka arba sugeneruota kokio nors įrankio, pavyzdžiui OCL apribojimų realizacija. Pilnai su MOF suderinama saugykla suteikia žymų meta duomenų servisų skaičių, kurie gerokai išeina už meta duomenų konstravimo ir aptarnavimo ribų (pavyzdžiui, saugojimas, versijų palaikymas, katalogavimo servisai). [Poo01]

4.2.4. XMI

XML Metadata Interchange yra OMG standartas, kuris suriša MOF su W3C XML kalba. XMI apibrėžia, kaip XML žymos yra naudojamos atvaizduoti su MOF suderinamus modelius XML formate. MOF paremti meta modeliai išverčiami į XML Dokumentų Tipų Apibrėžimus (angl. Document Type Definition, toliau DTD), o modeliai išverčiami į XML dokumentus, kurie yra suderinti su juos atitinkančiais DTD. [XMI]

XMI išsprendžia daugelį sudėtingų problemų, kurios kyla bandant naudot žymėm grįstas kalbas, kad atvaizduoti objektus ir jų ryšius. Be to, tas faktas, kad XMI yra paremtas XML, reiškia tai, kad meta duomenys (žymės) ir atvejai, kuriuos jie apibrėžia (elementų turinys) gali būti supakuoti kartu tame pačiame dokumente, įgalinant programas pilnai suprasti atvejus per jų meta duomenis. Turinio komunikacija yra save apibrėžianti ir paveldimai asinchroninė. Todėl XMI paremtas apsikeitimas yra labai svarbus išskirstytose, daugialypėse aplinkose. [XMI]



2 pav. Ryšiai tarp MDA standartų [Poo01]

4.3. MDA apibendrinimas

Iš pirmo žvilgsnio MDA yra daug žadantis metodas, ypatingai tinkantis šiame darbe nagrinėjamai dalykinei sričiai – dokumentų valdymo sistemoms. Dokumentinės sistemos gali būti labai įvairaus sudėtingumo – nuo paprasčiausių leidžiančių patogiai kaupti bei vėliau rasti dokumentus, iki labai sudėtingų su įvairia duomenų analize, ataskaitomis ir kitais mechanizmais.

Pastarosioms kaip tik ir būtų naudingiausia MDA architektūra. Apsirašius dokumentų meta duomenis, juos galima platinti daugialypėje aplinkoje, kur kiekvienas servisas ar programa sugeba atpažinti duomenis ir juos savaip interpretuoti. Tai gali būti duomenų įvedimo ar peržiūros klientinės programos, sudėtingi analizės servisai, ataskaitų formos ir panašiai. Kuriam nors servisui ar programai prireikus papildomos informacijos apie dokumentus, modeliavimo įrankiais sudaromas naujas duomenų meta modelis, kuris nesutrikdant kitų programų veikimo gali būti išplatinamas visoje infrastruktūroje.

Teoriškai viskas iš ties atrodo labai gražiai, tačiau kaip yra iš tiesų, kokie įrankiai yra pateikiami rinkoje, palaikantys šį metodą, ir kokią realią naudą jie gali duoti? Tai galima sužinoti tik praktiškai viską išbandžius.

5. MDA praktinis naudojimas

Naudojant MDA įrankius jokio pradinio pasiruošimo nereikia arba jis yra minimalus. Čia keliamas tik vienas esminis reikalavimas – UML kalbos išmanymas. Tai yra palyginti nedaug, nes šiuo metu dauguma su programų kūrimų susijusių asmenų dar besimokydami bent jau susipažįsta su šia jau standartu tapusia kalba. O net jos ir nemokant, ją perprasti nėra sudėtinga, kadangi ji tiesiog atvaizduoja objektinį programų modelį. Bet kuriuo atveju, nieko naujo išradinėti nereikės. Toliau belieka išsirinkti labiausiai tinkantį ir patinkantį MDA įrankį, jei reikia jį nusipirkti ir pradėti naudoti. Pasirinkimą čia įtakoja keli faktoriai – naudojama platforma, įrankio teikiamų galimybių ir kainos santykis ir panašiai. Daugiausia ir labiausiai išvystytų MDA įrankių skirta Java platformai, todėl čia pasirinkimas yra ganėtinai platus. Tačiau jei dirbate su Microsoft .NET platforma, situacija nėra labai puiki. Nors pastaruoju metu atsiranda vis didesnis MDA įrankių skirtų .NET platformai pasirinkimas, tačiau jų galimybės yra gana ribotos.

Galimybių tyrimui Interneto svetainėje www.objectsbydesign.com, kuri specializuojasi modeliavimo naujienų ir įrankių apžvalgomis buvo atrinkti įrankiai palaikantys Microsoft .NET platformą ir pamėginta išskirti esmines funkcijas kurių galima tikėtis iš šių įrankių. Toliau pateiktos dažniausiai pasitaikiusios savybės bei funkcijos:

- Daugumos UML standarto diagramų palaikymas;
- Integruotas reikalavimų valdymas;

- Integruoti projektų valdymo įrankiai apimantys resursus, metrikas bei testavimą;
- Dokumentavimo palaikymas;
- Testavimo atvejų generavimas pagal NUnit;
- Klasių generavimas iš UML klasių diagramų;
- Duomenų bazių schemų generavimas.

5.1. Sistemos modeliavimas su MDA

Kad paprasčiau būtų įsivaizduoti, kokios pagalbos galima tikėtis iš MDA įrankių kuriant programas, o tiksliau generuojant jas, buvo atliktas praktinis eksperimentas. Pasirinktas vienas įrankis, kuris bent jau pagal aprašymą, turėtų geriausiai atlikti kodo generavimo iš modelio uždavinį, bei pamėginta sumodeliuoti anksčiau darbe aprašyta testinė dokumentų valdymo sistema.

Eksperimentui pasirinktas įrankis objectiF, kuris neblogai atspindi visas MDA įrankių savybes išvardintas anksčiau. Įrankis buvo mėginamas pagal MDA siūlomą praktiką – visų pirma sukuriamas nuo platformos nepriklausomas modelis, kuris transformuojamas į platformai specifinį modelį, ir galiausiai iš jo sugeneruojamas kodas.

Testinės dokumentinės sistemos duomenų sluoksniui buvo sudaryta klasių diagrama, kurioje buvo penkios klasės – „Dokumentas“, „Redakcija“, „DokumentoTipas“, „PublikavimoSaltinis“ ir „Publikavimas“. Klasė „Dokumentas“ aprašo pagrindinę dokumento meta informaciją – pavadinimą, datą ir numerį. Ši agregacijos ryšiu sujungta su redakciją atitinkančia klase, kurioje saugomi redakcijos numeris, data bei tekstas. Dokumento publikavimo klasėje laikoma publikacijos šaltinio numeris, data bei nuoroda į patį publikacijos šaltinį. Taip pat buvo sukurta pora klasifikatorių – dokumento tipo bei publikavimo šaltinio, kurie sujungti atitinkamais ryšiais su klasifikuojama informacija. Nuo platformos nepriklausomas modelis pavaizduotas priede 1. Šis modelis įrankio pagalba, be jokių papildomų darbų, buvo konvertuotas į specifinį .NET platformai modelį, kurį galima pamatyti priede 2. Ir galiausiai, taip pat neįdedant jokių papildomų pastangų, tik pasirinkus reikiamą meniu punktą, platformai specifinis modelis buvo konvertuotas į kodą, kurį galima peržiūrėti priede 3. Sugeneruotas kodas buvo pilnai funkcionuojantis duomenų sluoksnis. Jame buvo aprašytos klasės paruoštos darbui su esybių valdymo moduliu, kuris užtikrina visą duomenų gyvavimo ciklą – sukūrimą, paiešką, atnaujinimą bei trynimą. Kitos UML diagramos kodo generavime nedalyvauja. Jos naudingos tik reikalavimų rinkimo fazėje bei reikalavimų trasavime kode, kadangi įrankis leidžia susieti diagramų elementus su kodu.

5.2. MDA įrankių apibendrinimas

Kaip ir buvo galima tikėtis, MDA įrankiai yra labai apibendrinti (tą pasako ir raidė „U“ angliškoje santrumpoje UML – „unified“, arba apibendrintas). Vieni įrankiai turi daugiau pagalbinių savybių, tokių kaip projekto valdymo ar testavimo palaikymas, kiti mažiau, tačiau žiūrint į kodo generavimo funkciją, visi jie yra panašūs – geriausiu atveju galima tikėtis sugeneruotų klasių ir pilnai veikiančio duomenų sluoksnio. Jei įmonė užsiima daugelio ir skirtingų sprendimų kūrimu, o dokumentinė sistema yra tik vienas iš jų, tuomet pasirinkus tinkamą MDA įrankį jo gali pilnai pakakti. Jis gerokai palengvins ir standartizuos reikalavimų rinkimą bei analizę, reikalavimų trasavimą, bei duos pagrindą tolesniam programavimui. Tačiau pilnai veikiančio programos kodo tikėtis neverta.

6. Specifinis sričiai modeliavimas

Dalykinei sričiai specifinis modeliavimas pakelia abstrakcijos lygį virš programavimo specifikuodamas sprendimą tiesiogiai naudodamas srities terminus. Galutiniai produktai sugeneruojami iš šių aukšto lygmens specifikacijų. Automatizavimas yra įmanomas, kadangi tiek kalba tiek generatoriai turi atitikti vienintelės kompanijos ir dalykinės srities reikalavimus. Ekspertai apibrėžia jas, o kūrėjai naudoja. [DSM]

Industrijos patirtis rodo, kad DSM pagreitina programų kūrimą 5 – 10 kartų palyginus su tradiciniais būdais, tame tarpe ir UML pagrįstomis MDA realizacijomis. Pilna MDA vertė pasiekiama tik tuomet, kai modeliavimo konceptai tiesiogiai atitinka dalykinės srities konceptus, o ne kompiuterinės technologijos konceptus. Pavyzdžiui, DSM mobiliojo telefono programinei įrangai turėtų konceptus kaip „Programinis klavišas“, „SMS“, „Skambėjimo tonas“, o generatoriai sukurtų atitinkamų kodo komponentų kvietimus. DSM išpildo modeliais grįsto kūrimo pažadus. Kadangi kodo generatorius apibrėžia įmonės ekspertas konkrečiai įmonės sričiai ir komponentams, sugeneruotas kodas yra geresnis, nei dauguma programuotojų parašytų ranka. [DSM]

6.1. Specifinė dalykinės srities kalba

Specifinė dalykinės srities kalba (DSL) – tai kalba sudaryta kurti specifinės probleminės srities programas. Labai dažnai DSL realizuojama kaip biblioteka egzistuojančiai kalbai. Pagrindinė specifinės kalbos charakteristika yra jos išraiškingumo galia. Dauguma egzistuojančių DSL yra deklaratyvios iš prigimties, tai reiškia, kad programuotojai mažiau kontroliuoja sakinius parašytus ta kalba. Deklaratyvios kalbos programa yra teiginių rinkinys, kuris remiasi matematine logika, ir todėl suteikia projektavimo karkasą, kuris turi intuityvias savybes, kurių nėra procedūrinėse kalbose. [DKV00]

Specifinė dalykinės srities kalba paprastai yra aukšto abstrakcijos lygio ir atitinka konceptus randamus probleminėje srityje, naudodama generatorius ir komponentus. DSL naudojimo privalumai yra tokie: padidintas produktyvumas, sumažintas kognityvinis atstumas, sumažintas kūrimo ir apmokymo laikas, bei padidintas komponentų pakartotinis panaudojimas. [RB05]

6.2. Dalykinės srities analizė

Tam, kad sukurti specifinę dalykinės srities kalbą, kūrėjas turi surinkti informacijos apie probleminę sritį, kuriai bus naudojama kalba. Dalykinės srities ir visų veiklų identifikavimo procesas vis dar yra labai kintantis. Ir nors programų pakartotinio panaudojimo bendruomenė teikia daug svarbos dalykinių sričių identifikavimui, tam naudojami metodai yra labai skirtingi. Yra keletas sprendimų priėmėjų dalykinės srities analizės procese. [Cle01]

Viena iš pagrindinių dalykinės srities analizės problemų yra spraga, kuri egzistuoja tarp dalykinės srities identifikavimo ir informacijos apie dalykinę sritį, kuri bus naudojama programose, ištraukimas. Pirma dalis vadinama „koncepcine analize“, o antroji „konstruktyviąja analize“. Vienas iš dalykinės srities analizės produktų yra dalykinės srities modelis, kuris yra informacinis artefaktas, leidžiantis kūrėjams pasiekti reikiamą abstrakcijos lygį realizuojant DSL. [RB05]

Probleminės srities analizė yra susitarimo reikalas, kadangi sritis paprastai turi daugiau negu vieną autoritetingą asmenį ar instituciją. Geriau yra laikyti juos kaip bendruomenę ir ištraukinėti informaciją apie ryšius ir galios lygį bendruomenės viduje. Tai yra dalis realaus pasaulio informacijos ištraukimo, kas reiškia, kad bet kokia informacija susijusi su sritimi yra renkama, nepaisant galutinio jos panaudojimo sukurtoje kalboje. Dalykinės srities analizėje svarbu išskirti objektus, ryšius ir operacijas susijusias su sritimi. [RB05]

Viena iš pagrindinių problemų atliekat dalykinės srities analizę, viena iš tų, kuri leidžia labai lengvai paveikti procesą daugeliu būdų, yra tai, kad dalykinės sritys beveik visada yra natūralios. Tai reiškia, kad ryšiai, operacijos ir objektai srityje nuolatos evoliucionuoja, ir nėra stipraus projektavimo principo kuriant dalykinės srities modelius. Tai dalykinės srities analizei suteikia tas pačias problemas su kuriomis susiduria natūraliųjų mokslų mokslininkai: nustatyti natūraliai vykstančius fenomenus ir teisingai juos sudėlioti į formalų karkasą. [CC94]

6.3. Specifinės dalykinei sričiai modeliavimo kalbos pranašumai

Visų pirma, ko gero nėra tokio dalyko, kaip universaliai suprantama notacija. Raidė „U“ UML standartuose reiškia „Apibendrinta“ (angl. „Unified“), o ne „Universali“ modeliavimo kalba. UML diagramos tikrai nėra suprantamos daugeliui klientų. Nors dauguma IT kūrėjų atpažins pagrindinius diagramų tipus, jie nebūtinai turės visiškai vienodą jų suokimą. Ir, žinoma,

yra daug procedūrinio Perl arba integruoto C programuotojų, kuriems objektiškai orientuotas modeliavimas niekada nebuvo labai prasmingas, kad jį mokytis. [Bla07]

UML yra puikus būdas OO programuotojams kalbant apie kodą. Ji gali būti naudojama nepaisant dalykinės srities. Nesvarbu ar jūs kuriate kosminius laivus, duomenų bazių programas, ar žaidimus. Įdomu yra tai, kad diskusijose dažnai neišnaudojama dalykinės srities terminijos nauda. Pavyzdžiui, žaidimų kūrėjai kalba su kitais žaidimų kūrėjais ir klientais apie žaidimus. Tam tikslui jie jau turi tam tikrą savo kalbą, savo terminiją šiai dalykiniai sričiai. [Bla07]

Specifinė dalykinės srities kalba apibendrina tą terminiją, kartu su taisyklėmis apie tai, kokia informacija turi būti surinkta apie konkretų elementą, ir kaip elementai turi būti sujungti. Tai suteikia kūrėjams ir klientams tikslią, aukšto abstrakcijos lygmens bendrą kalbą diskutuojant, mąstant ir projektuojant programas. Tikslus dalykinės srities atitikimas, aukšto lygio abstrakcija, tikslumas ir galimybė naudoti kalbą bendraujant su klientais yra aiškios sritys, kuriose DSL kalbos lenkia bendrines kalbas, tokias kaip UML. [Bla07]

Patyręs konkrečios dalykinės srities kūrėjas gali padaryti veikiančią programą tiesiai iš šios aukšto lygio specifikacijos. Kas dar svarbiau, paprastai patyręs kūrėjas gali specifikuoti kaip programa turi būti sukurta iš tokios specifikacijos bendru atveju. Kitaip sakant, patyręs kūrėjas gali specifikuoti kas konkretų elementą ar struktūrą modeliavimo kalboje atitinka programavimo kalboje ir karkase, kurį jis naudoja. Bet koks modelis sukurtas modeliavimo kalba gali būti automatiškai transformuotas į kodą. Kitaip nei ankstesni kodo generatoriai, kuriuos pateikia išoriniai kūrėjai nieko nenumanantys apie jūsų dalykinę sritį, šis yra parašytas jūsų patyrusio kūrėjo taip, kad sukurtų tokį kodą, kuris būtų lyg rašytas jo paties rankiniu būdu. [Bla07]

Naudojant specifines dalykinės srities kalbas ir kodo generatorius industrijoje produktyvumas paprastai išauga nuo 5 iki 10 kartų. Tuomet iškyla klausimas, kiek reikia pastangų sukurti savo modeliavimo kalbą ir generatorių, ir kiek kartų jūs jį naudosite. Jei nuolatos dirbate su ta pačia problemine sritimi, atsipirkimas garantuotas. Net jeigu kuriate vieną produktą, tą produktą gali sudaryti daug dalių, kurios gali būti išreikštos ta pačia modeliavimo kalba. [Bla07]

Specifinės dalykinės srities kalbos atsirado daug anksčiau nei UML. Žmonės kūrė savo pačių tekstines DSL nuo kompiuterijos industrijos pražios, ir vargu ar tai liausis. Bet kurioje dalykinėje srityje, gera DSL visada nurungs bendrinę kalbą. [Bla07]

Grafinių nuosavų modeliavimo kalbų kūrimas yra kažkas naujo. Čia yra daug daugiau darbo palaikant grafinį modeliavimą palyginus su tekstiniu. Kurti savo grafines modeliavimo kalbas realiai tapo įmanoma tik nuo 1990 metų, kuomet pirmieji metaCASE įrankiai buvo išleisti. Tokie įrankiai kaip MetaEdit+, neseniai išleisti Microsoft DSL Tools ir Eclipse EMF/GMF automatizuoja visą arba beveik visą grafinio modeliavimo palaikymą. Tačiau netgi

geriausiose specifinėse modeliavimo kalbose ir įrankiuose bus dalykų, kurie išeis iš dalykinės srities ribų. Šioms dalims mums vis dar reiks tradicinio programavimo ir bendrinio modeliavimo. [Bla07]

6.4. Kodo generavimas su DSM

Geriausi programų kūrėjai visada yra tinginiai. Jeigu jie kažką gali automatizuoti, jie tą būtinai padarys. Geras šios savybės poveikis yra tai, kad tuomet programuotojai gali kažką daryti aukštesniame lygyje, negaišdami laiko žemo lygio detalėms. Abstrakcijos lygio pakėlimas yra kelias pakelti produktyvumą. Praeities patirtis išmokė programuotojus į kodo generatorius žiūrėti su įtarimu. Kaip išorinio tiekėjo paruošta programa gali sukurti kažką panašaus ir gražaus į tai ką patys programuotojai rašo rankomis. Požiūris keičiasi, kai jų paklausi, ar jie pasitikėtų savo paties kurtais generatoriais. [Kel06]

Abstrakcijos lygio pakėlimas ir savo paties kodo generatoriai – būtent apie tai ir kalba DSL. Kas būtų, jei jūsų komanda galėtų kurti programas su modeliavimo kalbomis, kurios iš tiesų padeda sukurti geriausią projektą, rodydamos programuotojams geriausią kelią į unikalią architektūrą ir projektavimo taisykles, kurios yra svarbios jūsų dalykinei sričiai. Ir kuomet ateitų eilė kodo generavimui, jūs žinotumėt, kad kodas bus beveik toks pats, kokį jūs parašytumėt patys. Galutinis rezultatas yra toks pats, kaip ir būtų jei visi jūsų komandoje turėtų vienodai aukštą patirtį ir kvalifikaciją, išskyrus tai, kad viskas vyksta 5-10 kartų greičiau. [Kel06]

Kad modeliais grįstas programų kūrimas veiktų tokiu būdu, negalima naudoti bendros paskirties modeliavimo kalbų, tokių kaip UML ir iš anksto paruoštų kodo generatorių. Žmonės kurie kūrė UML neprojektavo jos, kad apibrėžti programas jūsų dalykinei sričiai, ar kad generuotų kitą kodą, nei griaučiai. Nepaisant daugelio pastangų padaryti ją tinkamą generavimui, niekam iki šiol nepavyko, ir vargu ar pavyks padaryti ją pakankamai gudrią ir patogią. Programos veikimas ir taisyklės, kurių ji turi laikytis yra specifinės dalykinei sričiai. Kad tai daryti efektyviai ir išbaigtai grafiniu projektavimu, jums reikia dalykinei sričiai specifinės modeliavimo kalbos. [Kel06]

DSL neturi tenkinti noro panaudoti ją kitiems tikslams. Kai kurie įrankių kūrėjai gali norėti padidinti savo rinkos pozicijas turint modeliavimo kalbą, kurią gali naudoti bet kas. Iš tiesų bet kuriai organizacijai iš tiesų reikia kalbos tik jų pačių naudojimui. UML nepakelia abstrakcijos lygio aukščiau nei kodas. UML gali pateikti gražią programos apžvalgą, bet tai nėra įrankis automatiniam kūrimui. Jei mes norime pakelti abstrakcijos lygį aukščiau nei kodas ir turėti galimybę sugeneruoti veikiantį kodą, specifinė dalykinės srities kalba yra vienintelė išeitis. [Kel06]

Kai ateina laikas kodo generavimo rezultatams, daugelis mūsų matėme ką padaro trečių šalių generatoriai. Statinio deklaratyvaus aprašymo sukūrimas iš interfeisų ar duomenų bazių schemų jau yra realybė nuo senų laikų. Situacija keičiasi kai kalba pasisuka apie funkcionalumo generavimą. Nepaisant daugelio būdų, kurias galima parašyti kodą konkrečiam funkcionalumui, paprastai pasirenkamas tik vienas iš jų, kuris vargu ar panašus į idealų kandidatą jūsų specifinei programai. Reikia atsižvelgti į daugelį faktorių: programavimo kalba, programavimo modelis ir atminties naudojimas yra tik nedaugelis iš jų. Be to, šie faktoriai bėgant laikui kinta, reikalaudami, kad ir kodo generavimas keistųsi. Trečiųjų šalių generatoriai dažnai nežino pakankamai apie specifinius organizacijos reikalavimus, tam kad sugeneruot idealų kodą, todėl nenuostabu, kad daugelis sugeneruotą kodą laiko nepatenkinamu. Kadangi generuoto kodo modifikavimas paprastai nėra realus pasirinkimas, organizacijos dažnai generuotą kodą tiesiog išmeta. Generuoto kodo nauda sumažinama iki reikalavimų rinkimo ir prototipų kūrimo – gal tai ir naudingą, bet galėtų būti ir geriau. [Kel06]

6.5. Paprasti generatoriai

Generatoriaus kūrimas yra vienas iš dviejų pagrindinių užduočių kuriant DSM sprendimą. Dauguma programuotojų rašė bent jau skriptus arba mažas programėles, kad sugeneruoti pasikartojantį kodą. DSM generatorius turi nueiti kur kas toliau, kadangi jis skirtas naudoti ir kitiems žmonėms, nei jo kūrėjas, ir jo rezultatas paprastai nebus koreguojamas, ar netgi tikrinamas rankiniu būdu. Nors DSM generatorius turi būti kur kas kokybiškesnis nei minėtieji skriptai ar programėlės, tačiau problema DSM generatoriuje yra mažiau skausminga negu klaida fiksuotame generatoriuje pateiktame su modeliavimo įrankiais. Kitaip nei su šiais supakuotais generatoriais, jūs kontroliuojate situaciją, ir taip pat galite ištaisyti problemą tuomet, kai jums to prireikė. [Kel06]

Dalykas, kuris daro DSM patrauklų iš kodo generavimo perspektyvos yra tai, kad jis leis išlaikyti kodo generatorių paprastą. Kitaip nei daugelis atvejų, kuomet generatorius turi užtikrinti, kad įvestis, kurią jis gauna, yra teisinga, su DSM tai padaro taisyklės, kurias jūs užtikrinate modeliavimo kalba. Aprašant generatorius, jūs dažnai pastebėsite pasikartojančius šablonus sugeneruotame kode. Tai leis jums atskirti šias dalis į programinius komponentus ar generatoriaus karkaso kodą, kurį generatorius galės kviesti. Pagrindinė mintis – stengtis išlaikyti generatorių kiek įmanoma paprastesnį. Kuo paprastesnis jis yra pradedant darbą, tuo paprasčiau bus atlikti pakeitimus. [Kel06]

DSM generatoriaus kūrimas yra modelio elementų susiejimas su kodu ar kita išvestimi. Paprastu atveju, kiekvienas modelio simbolis sukurs fiksuotą kodą, kuris turės reikšmes įvestas kaip simbolio argumentus. Generatorius taip pat gali įvertinti ryšius su kitais modelio elementais

ar kitą modelio informaciją, tokią kaip įdėtiniai modeliai, modeliai sukurti kitomis kalbomis, ar jau egzistuojančios bibliotekos. Būdas, kuriuo generatorius paima informaciją iš modelių ir paverčia ją kodu, priklauso nuo to, kaip sugeneruota išvestis turi atrodyti. Kitaip sakant, jums reikia geros nuorodų realizacijos. Svarbu, kad ekspertas parašytų tikslinį kodą geru standartiniu kodu, kurį būtų galima apibendrinti visoms dalykinės srities programoms. Standartinio kodo generavimas suteikia gerą įspūdį, kad jis yra pažįstamas, laikosi reikiamo programavimo modelio, įtraukia atitinkamus komentavimo ir kodo standartus. Tai daro sugeneruotą kodą lengviau priimtiną net jei vėliau būtumėte priversti atsisakyti kodo generavimo. [Kel06]

Kodo generatoriaus apibrėžimą geriausia pradėti apsirašant tikslinį kodą ir dirbti atvirkščia tvarka. Tuomet jūs galite pritaikyti tikslines realizacijas ir jų modelius kaip testavimo atvejus viso kodo generatoriaus apibrėžimo fazių metu. Paprastai jūs pradėsite nuo kelių tipinių struktūrų ir tuomet po truputį plėsite generatorių. [Kel06]

6.6. Modelis į kodą ar modelis į modelį

Žmonės, kurie išbando MetaEdit+ kartais klausia apie „modelis į modelis“ transformacijas. Nors įrankis tai palaiko, tačiau to kaip DSM sprendimo daryti nerekomenduojama, kadangi yra keletas būdų, kuriais žmonės gali nueiti klaidingu keliu. Patirtis rodo, kad geriau yra generuoti iš karto kodą, negu sukurti kažkokius tarpinius modelius, kuriuos vėliau reikėtų išplėsti kūrimo procese. [Kel006]

Normaliu atveju, „modelis į modelį“ transformacijos mintis yra ta, kad kiekvienas duomenų elementas aukštesnio lygmens kalboje transformuojamas į daugiau nei vieną žemesnio lygmens kalbos elementą. Tai yra nieko tokio, jei jūs niekada nežiūrėsite į žemesnio lygio kalbą, ir niekuomet jos netaisysite. Bet jei nuspręsite pataisyti, turėsite dirbti su dviem duomenų elementais, bet akivaizdu kad jie nėra visiškai nepriklausomi, kadangi galėjo būti sukurti iš vieno elemento. Viso to rezultatas yra tai, kad pakeitimai aukštesniame lygmenyje reikalaus pakeitimų ir žemesniame. [Kel06]

Dažnai žmonės nori turėti galimybę pakeisti aukštesnio lygmens modelius, ir kad tie pakeitimai atsispindėtų žemesnio lygmens modelyje. Tai reiškia, kad jūs dirbate su 1+2 informacijos elementais, be to dar reikia sugalvoti būdą kaip perkelti pakeitimus į žemesnį lygmenį teisingai. Teisingai reiškia – nesunaikinant informacijos rankiniu būdu pridėtos žemesnio lygmens modelyje, atnaujinant automatiškai sugeneruotas dalis; sukuriant naujas dalis ir atnaujinant rankiniu būdu pridėtas dalis taip, kad jos atitiktų aukštesnio lygmens pakeitimus. Be to, dar yra noras pakeisti žemesnio lygmens modelius ir kad pakeitimai būtų perkelti į aukštesnį lygmenį. O tai padaryti dar sudėtingiau. Tai iš tiesų įmanoma tik tuo atveju, jeigu nėra dviejų lygmenų, o tik du skirtingi pavaizdavimai tame pačiame lygmenyje. [Kel06]

6.7. Gero kodo generavimo savybės

Jeigu jūs turite gerą generatorių, kuomet jis sugeneruoja išbaigtą ir veikiantį kodą, tokį, kurio vėliau nereikia rankiniu būdu perrašinėt ar kažką pridėti. Žinoma, kartais gali iškilti poreikis optimizuoti mažą kodo dalį rankiniu būdu, tačiau tai turėtų būti išimtis iš taisyklės. Sugeneruoto kodo taisymas yra analogiškas mašininio kodo taisymui po C kompiliavimo. Jūs turėtumėte laikyti sugeneruotą kodą kaip tarpinį produktui. Tai buvo sėkmės receptas kompiliatoriams, ir kodo generatoriai galėtų pasiekti tą patį. [Kel06]

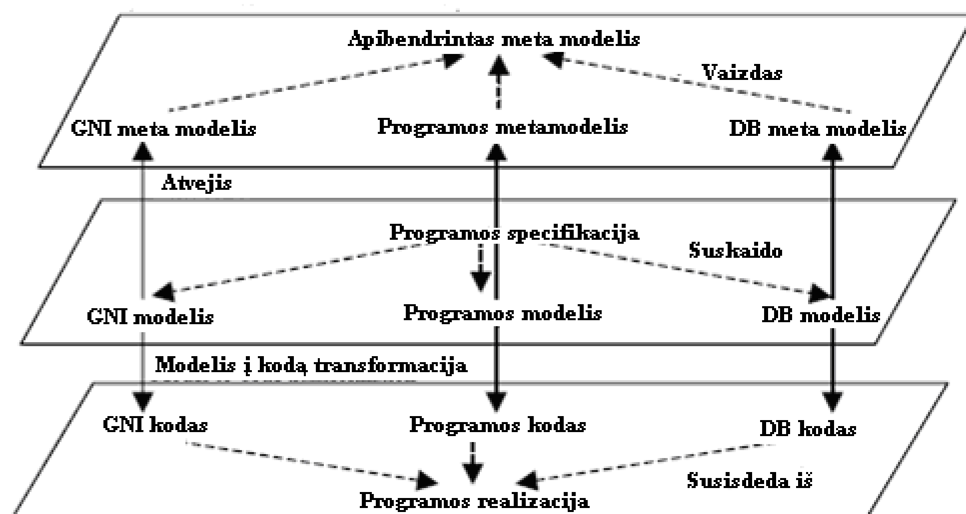
Kodo generavimas yra ne vienintelė automatizavimo vieta. Modelio galia išauga dar labiau jeigu jūs galite sugeneruoti tokius dalykus, kaip konfigūracijos duomenys, testavimo atvejai, simuliacijos medžiagą, dokumentaciją, automatinį surinkimo procesą, ir taip toliau. Turint vieną šaltinį ir keletą rezultatų gali būti labai naudinga darant pakeitimus. Tai tenka padaryti tik vienoje vietoje.

Paskutiniu metu, atviros ir pritaikomos technologijos išsivystė tiek, kad leidžia kūrėjams pakeisti tiek projektavimo kalbas ir/arba kodo generatorius, kad pasiektų savo tikslus. Tačiau patyrę kūrėjai kompanijoje gali pritaikyti projektavimo kalbas ir generatorius specifinei dalykinei sričiai, ir tuomet modeliuoti realius produktus. Galiausiai, ekspertui modeliais grįstas kodo generavimas nėra tik įdomus iššūkis – tai taip pat labai linksma. [Kel06]

6.8. Gerosios DSM praktikos

Programos kūrimas prasideda nuo abstrakčios specifikacijos, kuri turi būti transformuota į konkrečią realizaciją tam tikroje tikslinėje architektūroje. Tikslinė architektūra dažniausiai būna suskirstyta į atskirus sluoksnius, vaizduojančius tam tikrą aspektą, pavyzdžiui, Grafinio Naudotojo Interfeiso (GNI) sluoksnis, programos logikos sluoksnis ir duomenų bazės sluoksnis. Modeliavimas sukonstruoja Programos specifikaciją naudodamas skirtingus abstrakčius vaizdus – GNI sluoksnio modelis, programos logikos sluoksnio modelis ir DB sluoksnio modelis. Kiekvienas iš jų apibūdina tam tikras savybes atitinkančias sluoksnį, kuriam ji priklauso, kaip parodyta paveiksle 3. Šias specifikacijas atitinka trys meta modeliai – GNI sluoksnio meta modelis, logikos sluoksnio meta modelis, DB sluoksnio meta modelis, kurie yra vieno apibendrinto meta modelio vaizdai. Turint vieną meta modelį galima specifikuoti apribojimų integralumą, kuris būtų patenkintas susijusiuose modelio elementuose visuose sluoksniuose. Tai leidžia nepriklausomas GNI sluoksnio modelio, logikos sluoksnio modelio ir DB sluoksnio modelio transformacijas į atitinkamas realizacijas, t.y. GNI sluoksnio kodo, logikos sluoksnio kodo ir DB sluoksnio kodo. Šios transformacijos gali būti atliktos rankiniu arba automatiniu būdu. Transformacijos apibrėžiamos meta modelio lygmenyje ir yra pritaikomos visiems modelio atvejams. Jei kiekviena individuali transformacija realizuoja atitinkamą specifikaciją ir

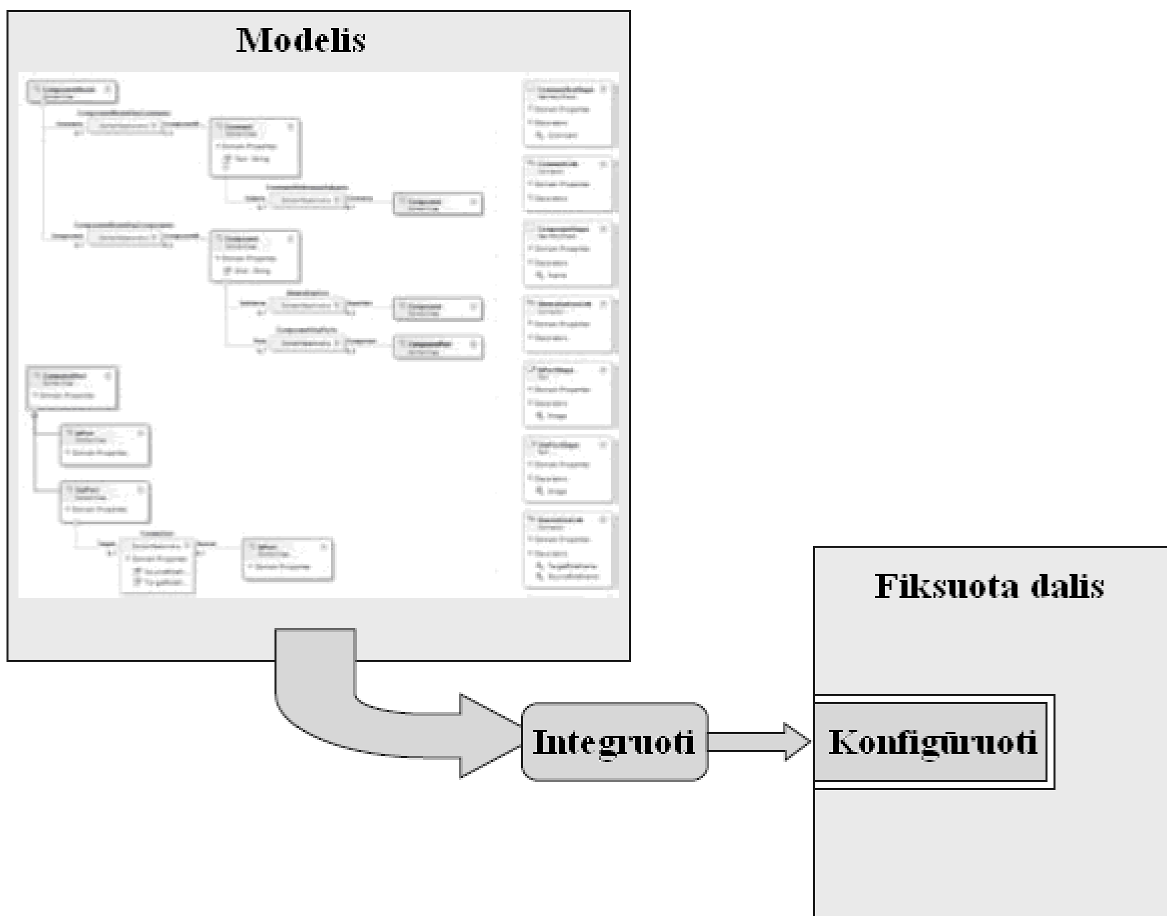
jos ryšiai su kitomis specifikacijomis yra teisingi, tuomet galutinės realizacijos susiklijuos drauge, duodamos vieningą specifikacijos realizaciją. Modeliai gali būti laikomi nepriklausomi nuo realizacijos technologijos ir programos specifikacijos gali būti perkeltos į keletą technologinių platformų per modeliais grįstą kodo generavimą. Programos specifikacijos konstravimas nepriklausomais modeliais leidžia skaldyti ir valdyti. Automatinis kodo generavimas leis pasiekti aukštesnį produktyvumą ir geresnę kokybę. Modeliavimas padeda rasti klaidas daug anksčiau visame programos kūrimo cikle. Su kiekvienu modeliu susiejamas rinkinys taisyklių ir apribojimų, kurie apibrėžia jo atvejų teisingumą. Šios taisyklės ir apribojimai gali įtrauki tipų tikrinimo taisykles ir vientisumą tarp specifikacijos skirtingų sluoksnių. [ARM01]



3 pav. Modeliais grįstas kūrimas [ARM01]

6.9. DSM apibendrinimas

Specifinis dalykinei sričiai programų kūrimas yra būdas spręsti tam tikras besikartojančias problemas. Kiekvienas problemos atvejis turi daug vienodų aspektų, ir šios problemos gali būti išspręstos vieną kartą ir visiems laikams. Problemos kintantys aspektai gali būti aprašyti specialia kalba. Kiekvienas problemos atvejis gali būti išspręstas sukuriant modelį ar išraišką specialia tam skirta kalba. Po to šis modelis būtų įdedamas į fiksuotą sprendimo dalį, kaip pavaizduota 4 paveiksle.



4 pav. Programų kūrimas su DSM [CJK07]

Jeigu įmonė užsiima išskirtinai vieno tipo sprendimų kūrimu, šiuo atveju dokumentinių sistemų, kiekvieną kartą kurti naują dokumentų valdymo sistemą nuo nulio būtų neefektyvu. Čia pasitarnauti gali specifinis dalykinei sričiai modeliavimas.

Šis programų kūrimo būdas yra labai tinkamas dokumentinėms sistemoms. Visos dokumentinės sistemos iš esmės yra labai panašios. Pagrindinis jų uždavinys – efektyviai kaupti dokumentus, kad vėliau būtų patogu juos surasti, keisti, analizuoti bei kitaip apdoroti. Skiriasi tik skirtingų organizacijų poreikiai. Vieniems reikia darbų sekos ar kažkokios ypatingos apsaugos palaikymo, kietiems – ne. Vieni saugo vienokius dokumentų meta duomenis, kiti – kitokius, ir panašiai. Taigi iš esmės turime fiksuotą pagrindinių funkcijų ir savybių rinkinį iš kurio galime konstruoti reikiamą dokumentų valdymo sistemą. Taigi čia visą savo grožį ir galią turėtų parodyti dalykinei sričiai specifinis modeliavimas.

Toliau šiame darbe bus mėginama sukurti specifinę dokumentinių sistemų kūrimui skirtą kalbą, kuri būtų suprantama dokumentinių sistemų kūrėjams ir jų užsakovams, bei įrankius skirtus naudoti šią kalbą, kurie leistų sudaryti modelius, bei iš jų sugeneruoti veikiantį programos, ar bent jau jos dalių kodą.

7. DSM praktinis panaudojimas

Kaip darbo priemonė buvo pasirinktas Microsoft paketas DSL Tools, leidžiantis gana patogiai kurti grafines modeliavimo kalbas, kurias galima integruoti į programavimo aplinką Microsoft Visual Studio 2005. DSL Tools leidžia sukurti ne tik grafinę modeliavimo kalbą, bet jos pagrindu sugeneruoja ir įrankį, labai panašų į Visual Studio, kuriuo galima kurti modelius sukurta kalba, bei generuoti iš sukurto modelio kodą.

7.1. Dalykinės srities modelio aprašymas

Kiekvienos DSL branduolys yra dalykinės srities modelis. Jis apibrėžia kalbos vaizduojamus konceptus, jų savybes, bei tarpusavio ryšius. Visi DSL naudotojai turi būti susipažinę su šiuo modeliu bent jau iki tam tikro lygio, kadangi kiekvienas elementas, kurį jie sukuria ir su kuriuo manipuliuoja naudodami DSL yra apibrėžti srities modeliu. Srities modelis yra kaip DSL gramatika; ji apibrėžia elementus, kurie sudaro modelį ir taisykles – kaip šitie elementai turi būti sujungti tarpusavyje.

Taigi pirma užduotis – išsiaiškinti pagrindinius bendrus dokumentų valdymo sistemų elementus bei ryšius tarp jų, t.y. sudaryti dokumentų valdymo sistemos dalykinės srities modelį. Šis modelis bus orientuotas į teisinio pobūdžio dokumentines sistemas, tokias kaip aprašyta testinė dokumentų valdymo sistema.

Kiekvienos dokumentinės sistemos pagrindinis objektas yra dokumentas, tai yra informacija kokioje nors laikmenoje – elektroninės dokumentų valdymo sistemos atveju dokumentai laikomi elektroninėje laikmenoje. Pagal informacijos pobūdį dokumentai gali būti tekstiniai, grafiniai arba garsiniai. Paprastai elektroninės dokumentinės sistemos, ypač orientuotos į teisinius dokumentus, apima tik tekstinius ir grafinius dokumentus. Dokumentuose išdėstoma tam tikram subjektui – asmeniui, organizacijai ar valstybei, svarbi informacija, kurią svarbu ne tik kaupti, bet vėliau ir rasti bei panaudoti. Saugant tik patį dokumentą – tekstą arba kažkokią grafinę informaciją, jo paieška gali būti komplikuota. Jei teksto atveju dar įmanoma rasti reikiamą dokumentą ieškant pagal kažkokią tekste esančią frazę, nors tokia paieška ir nėra labai efektyvi ir greita, tai grafinio dokumento atveju informacijos paieška praktiškai neįmanoma. Todėl be paties dokumento būtina saugoti ir kažkokią papildomą informaciją apie dokumentą, dar vadinamą meta informacija. Meta informacijoje gali būti saugomas unikalus dokumento numeris, data, pavadinimas, bei kiti dokumento paiešką palengvinantys atributai. Todėl projektuojant dokumentinę sistemą vienas svarbiausių aspektų yra numatyti, kokia meta informacija turi būti saugoma ir kaip ji grupuojama.

Informacijos grupavimui naudojami klasifikatoriai, kurie suskirsto dokumentus į tam tikras grupes, kurios gali susiaurinti paiešką. Pavyzdžiui, dokumentai gali būti grupuojami pagal rūšį –

įstatymas, įsakymas, nutarimas ir panašiai, arba pagal būseną – galiojantis, negaliojantis, sustabdytas, projektas, bei kitus požymius. Todėl kalba skirta projektuoti dokumentines sistemas turi apimti ir tokius atributus.

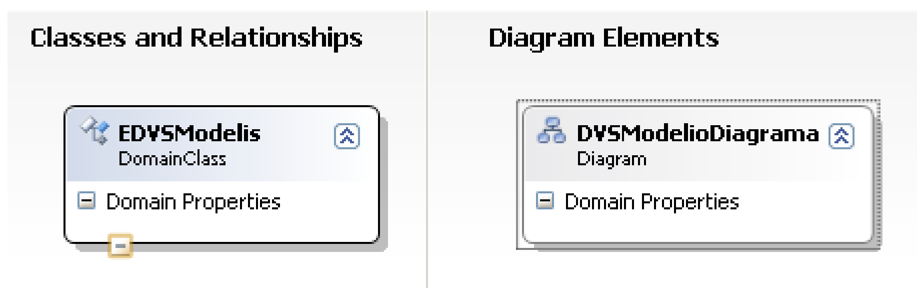
Kitas labai svarbus aspektas dokumentinėse sistemose yra dokumentų versijos, arba kitaip vadinamos dokumentų redakcijos. Ypač tai aktualu teisinio pobūdžio dokumentinėse sistemose, kadangi čia dokumentų pokyčiai ypatingai dažni ir būtina išsaugoti visas jų redakcijas, nes vykstant teisiniams procesams, pavyzdžiui teismams, vadovaujamasi konkrečiu periodu galiojusiomis dokumentų redakcijomis. Jei dokumentas turi daugiau nei vieną redakciją, tuomet ji būtinai turės mažiausiai du skirtingus nuo pagrindinio dokumento atributus – tai redakcijos numeris ir data. Dėl šios priežasties kuriant dalykinės srities modelį redakcija bus išskirta kaip atskiras objektas.

Be dokumento ir redakcijos dokumentinėse sistemose taip pat labai svarbu tarpusavio ryšiai tarp šių esybių, tai yra ryšiai tarp pačių dokumentų, bei tarp dokumentų ir redakcijų. Ryšiai tarp dokumentų atsiranda tuomet, kai vienas dokumentas cituoja kitą dokumentą arba juo remiasi. Taip pat dokumentai gali vienas kitą keisti, naikinti, stabdyti ir kitaip įtakoti. Šiems ryšiams nurodyti dokumentinės sistemos dalykinės srities kalboje taip pat turi būti numatytos priemonės.

7.2. DSL kūrimas su DSL Tools

Dabar jau aišku, kokie pagrindiniai elementai turi būti kalboje aprašančioje dokumentines sistemas, todėl galima pradėti sudarinėti konkretų kalbos modelį, tai yra dalykinės srities artefaktus reikia perkelti į dalykinės srities kalbos klases.

Modelio kūrimą pradėsim nuo paties mažiausio teisingo modelio sukūrimo, kuris turi tik vieną dalykinės srities klasę susietą su viena diagrama.



5 pav. Mažiausias teisingas DSL modelis

Dalykinės srities klasė „EDVSModelis“ yra šakninė dalykinė srities klasė (angl. Root Domain Class). Srities modelyje gali būti tik viena tokia klasė. Klasė „DVSModelioDiagrama“ yra pagrindinė vizualaus modelio klasė. Ši klasė sugeneruotame DSL įrankyje vaizduos pačią diagramą, kurioje bus sudarinėjamas modelis. Įrankis DSL Tools leidžia atskirti dalykinės srities

modelį nuo jo pavaizdavimo. Todėl pirma bus sukurtas dalykinės srities modelis, o vėliau bus sukurta jo grafinė notacija.

Prieš pradedant kurti patį modelį reikia susipažinti su tam tikrais terminais, kurie bus naudojami aprašant modelį.

Kas yra dalykinės srities klasė (angl. Domain class) detalaus paaiškinimo nereikia. Intuityviai galima suprasti, kad tai yra tiesiog dalykinės srities objektų atitikmuo dalykinės srities kalboje. Kaip priimta objektiniame modeliavime, čia klasės taip pat gali dalyvauti paveldėjimo hierarchijoje. Grafinė notacija vaizduoti paveldėjimui paimta iš UML kalbos. Paveldėjimo prasmė taip pat tokia, kokios ir tikimasi – paveldėta dalykinės srities klasė paveldi ir visas dalykinės srities savybes iš tėvinės klasės. Dalykinė klasė taip pat gali būti pažymėta kaip abstrakti (angl. Abstract) arba galutinė (angl. Sealed). Abstrakčios klasės negali būti tiesiogiai sukuriami objektai – ji turi turėt paveldėtą klasę. Iš galutinės klasės negalimas paveldėjimas.

Dalykinės srities klasės gali turėti tarpusavio ryšius. Kiekvienas ryšys turi kryptį iš kairės į dešinę. Kairys galas yra šaltinis, o dešinys ryšio tikslas. Yra du ryšių tipai – įdėtasis (angl. Embedding) ir nuorodos (angl. Reference). Šie tipai turi ganėtinai skirtingas charakteristikas, ypatingai įtakojančias kalbos vizualizavimą ir saugojimą.

Įdėtiniai ryšiai suteikia galimybę po modelį vaikščioti kaip per medį, tai yra, kaip po struktūrą, kurioje kiekvienas elementas (išskyrus šakninį) turi lygiai vieną tėvinį elementą. Tai yra svarbu dėl kelių priežasčių. Pirma, modeliai paprastai saugomi kaip XML dokumentai, kas yra išreikštinai medžio struktūra. Todėl įdėtiniai ryšiai nurodyti modelyje apibrėžia XML struktūrą. Antra, įdėtiniai ryšiai nusako modelio struktūrą, kadangi ji taip pat yra sudaryta kaip medis. Bei trečia, šio tipo ryšiai nusako standartinę trynimo ir kopijavimo modelyje elgseną. Pagal nutylėjimą, trinant įdėtinio ryšio tėvinį elementą ištrinami ir visi jo vaikai.

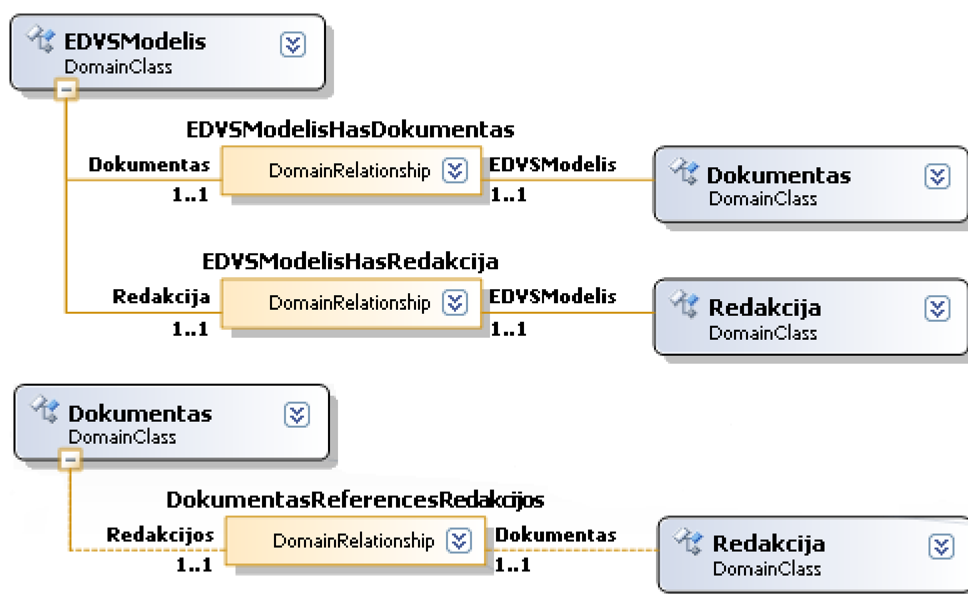
Nuorodos tipo ryšiai nėra apriboti kaip įdėtiniai. Bendru atveju, nuorodos tipo ryšiai gali sujungti bet kuriuos du kalbos elementus. Galima netgi sukurti ryšius tarp ryšių, tačiau nereikia pamiršti, kad tai įtakos kalbos sudėtingumą. O vienas iš svarbesnių tikslų kuriant DSL yra jos paprastumas ir aiškumas.

7.3. Dalykinės srities modelio kūrimas

Turint modelio pagrindą. toliau į jį reikia įdėti pagrindinius dalykinės srities elementus. Bet kurioje dokumentinėje sistemoje pagrindiniai yra du elementai – dokumentas ir jo redakcijos. Dokumentas čia suprantamas, kaip objektas, kuriame saugoma pagrindinė dokumento informacija, dar vadinama dokumento meta informacija, pavyzdžiui, dokumento numeris, priėmimo data, tipas ir panašiai. Dokumento redakcija yra informacija apie atskiras dokumento

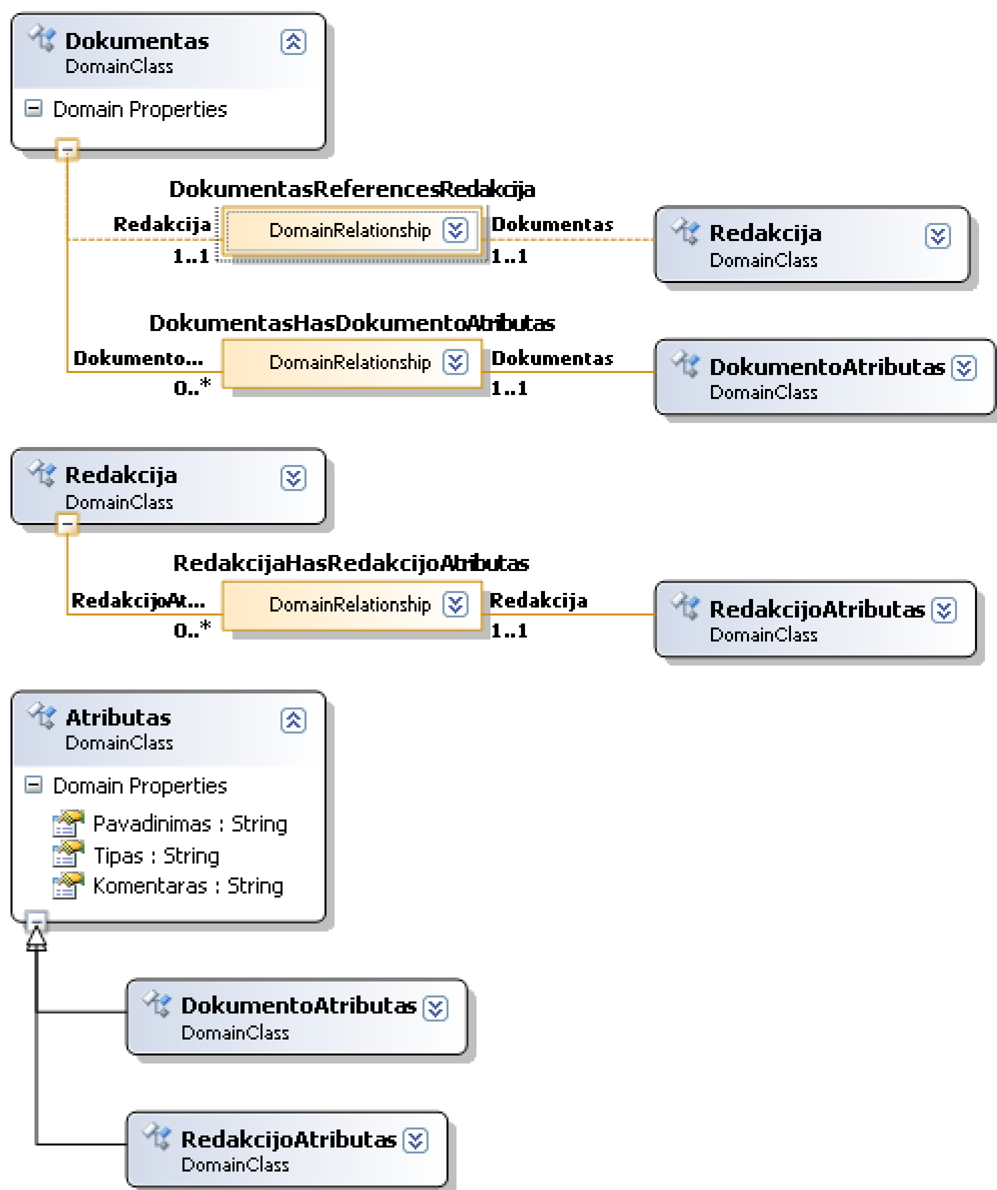
versijas. Redakcijos pagrindiniai atributai yra redakcijos tekstas, priėmimo data, įsigaliojimo data ir panašiai. Šie du elementai turi būti sujungti ryšiu „dokumento redakcija“.

Dokumentui ir redakcijai bus sukurtos dvi dalykinės srities klasės „Dokumentas“ ir „Redakcija“. Šios dvi klasės bus sujungtos ryšiu „DokumentasTuriRedakcijas“. Modelyje dokumentas ir redakcija gali būti tik po vieną kartą. Tai nurodoma sujungiant dalykinės srities klases „Dokumentas“ ir „Redakcija“ su klase „EDVSModelis“ įdėtuoju ryšiu (angl. Embedding relationship) ir nurodant pasikartojimą 1..1 (6 pav.).



6 pav. Pagrindinės dalykinės srities klasės

Turint pagrindines dalykinės srities klases, dabar reikia jas padaryti pritaikomas skirtingiems poreikiams. Pirmasis dalykas, kas daro šias klases konfigūruojamomis, tai jų savybės. Tiek dokumentas, tiek redakcija gali turėti pačių įvairiausių savybių, arba dar kitaip atributų. Pavyzdžiui, dokumento numeris, priėmimo data, pavadinimas, įvairūs kiti identifikavimo požymiai. Panašiai ir redakcija gali turėti įvairių savybių – datą, kelių skirtingų formatų tekstai ir panašiai. Tam tikslui sukursim ir į dalykinės srities modelį įtrauksim klasę „Atributas“. Ši klasė turės tris savybes – pavadinimą, duomenų tipą bei komentarą, kuriame bus galima trumpai aprašyti, kam atributas skirtas. Kadangi atributai turi priklausyti ne visam modeliui, o konkrečiai dokumentui arba redakcijai, todėl reikia sukurti dvi atskiras klases „DokumentoAtributas“ ir „RedakcijosAtributas“, paveldėtas iš klasės „Atributas“, ir jas surišti įdėtumo ryšiu su atitinkamomis klasėmis (7 pav.).

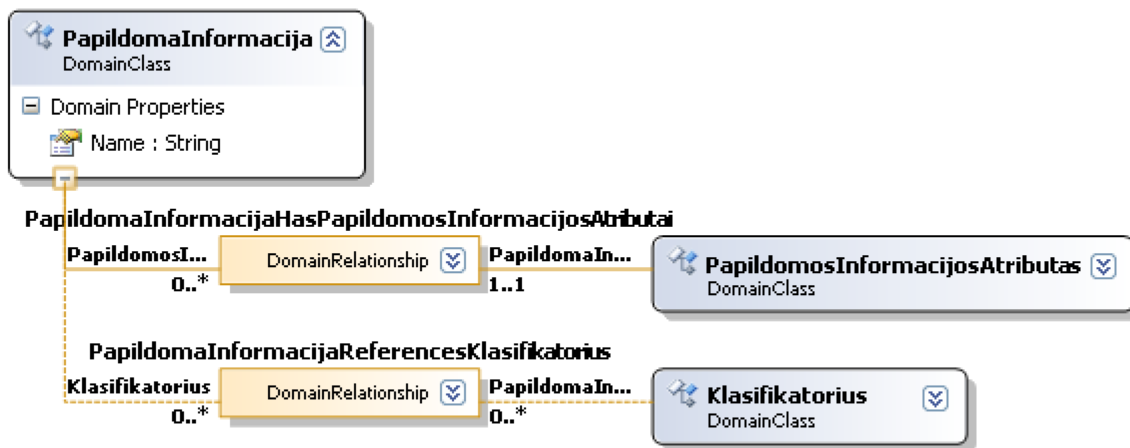


7 pav. Dokumento ir redakcijos atributai

Kitas svarbus dalykas dokumentinėse sistemose – klasifikatoriai. Tai yra tam tikros grupės, į kurias skirstomi duomenys, kad jų paieška taptų efektyvesnė. Dokumentinėse sistemose tokių klasifikatorių pavyzdžiais galėtų būti dokumentų rūšys, būsenos ir panašiai. Klasifikatorius turėti gali tiek dokumentas, tiek ir jo redakcijos, todėl dalykinės srities klasė „Klasifikatorius" bus įdėta į modelį ir susieta įdėtiniu ryšiu su šaknine dalykinės srities klase „EDVSModelis", o su klasėmis „Dokumentas" ir „Redakcija" bus susieta nuorodos ryšiais. Be to, klasifikatorius taip pat kaip ir kitos dalykinės srities klasės turi tam tikrus atributus, pavyzdžiui, klasifikatoriaus pavadinimas, pavadinimo sutrumpinimas, rūšiavimo ir kiti. Todėl reikės sukurti dar vieną atributo klasę paveldėtą iš klasės „Atributas" – „KlasifikatoriausAtributas", bei susieti jį įdėjimo ryšiu su klasifikatoriaus klase.

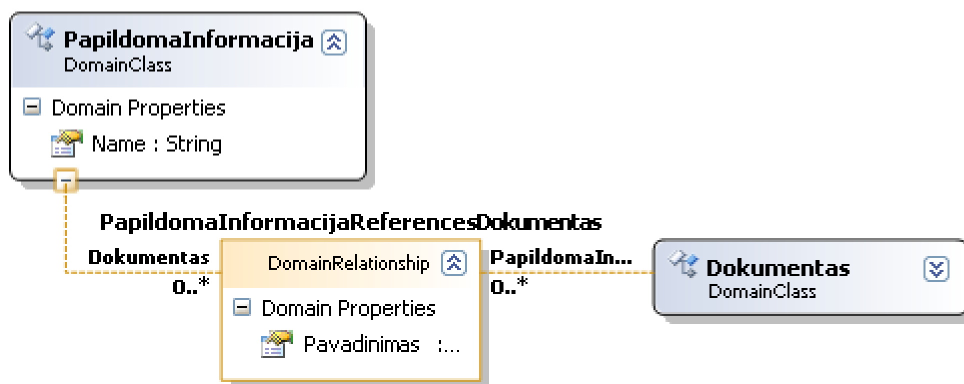
Be jau išvardintų, dokumentai bei redakcijos gali turėti ir papildomos informacijos, tokios kaip publikavimas, įvairios chronologijos datos ir panašiai. Šios informacijos negalima pridėti į

pagrindinius dokumento atributus, kadangi vieni dokumentai tokių savybių gali turė daug, kiti iš vis neturėti. Todėl šioms savybėms reikia sukurti atskirą dalykinės dalies klasę susietą su dokumentu ir redakcija, kurią pavadinsime „PapildomaInformacija". Papildomai informacijai aprašyti taip pat reikės atributų. Tam sukursime dar vieną klasę „PapildomosInformacijosAtributas" paveldėtą iš klasės atributas ir susiesime su naujai sukurta. Be to, informacija, taip pat gali būti klasifikuojama, todėl ji turi būti susieta ir su klasifikatoriaus klase (8 pav.).

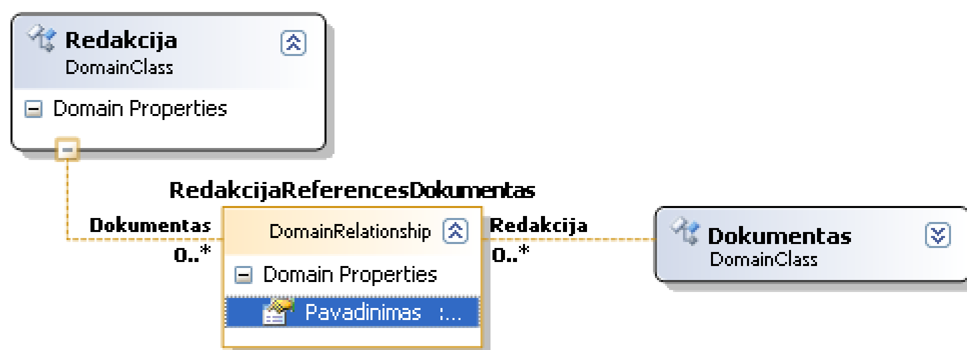


8 pav. Papildoma informacija

Kartais tam tikrose dokumentinėse sistemose reikia ryšio jungiančio redakciją arba papildomą informaciją su dokumentu. Pavyzdžiui, redakcijoje norime įdėti nuorodą į dokumentą, kuris tą redakciją sąlygojo, arba norime aprašyti papildomą informaciją, kurioje nusakomi logiškai susiję dokumentai. Tai turėtų būti nuorodos tipo ryšys turintis pavadinimą (9 ir 10 pav.).

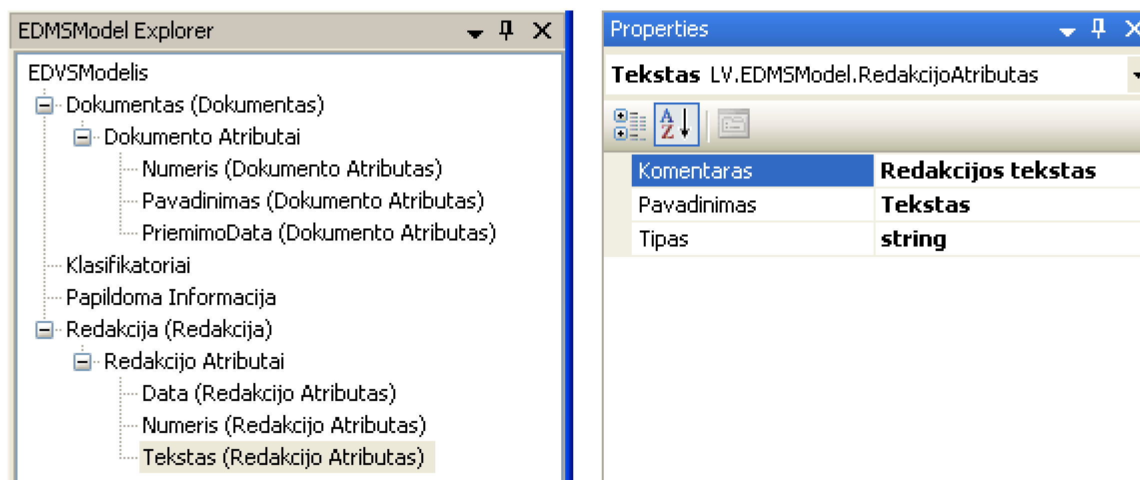


9 pav. Papildomos informacijos ryšys su dokumentu



10 pav. Papildomos informacijos ryšys su redakcija

Turint iki šiol aprašytą dalykinės srities modelį jau galima sugeneruoti veikiantį dokumentinės sistemos dizainerį ir juo naudotis. Tiesa, tai nebus labai patogi programa. Mes turėsime tuščią diagramą ir tuščią įrankių juostą, kadangi šių dalių dar neaprašėme, tai bus padaryta kitame skyriuje. Tačiau mes turime pilnai veikiančius tyrinėjimo (angl. Explorer) ir savybių langus. 11 paveikslėlyje matome vien tyrinėjimo lango pagalba sukurtą dokumentinės sistemos modelį, bei redakcijos atributo tekstas savybes. Tačiau ši programa nėra pilnai funkcionali iš dalykinės srities modelio perspektyvos, kadangi neįmanoma tyrinėjimo ar savybių lango pagalba sukurti ryšių tarp atskirų elementų. Be to ji nėra labai patogi ar informatyvi nepatyrusiam naudotojui. Kad užpildyti šias spragas reikia sukurti kalbos grafinį vaizdą.



11 pav. Modelis sukurtas be gravinės notacijos

7.4. DSL grafinė notacija

Ką tik buvo aprašytas dokumentinėms sistemoms skirtos kurti kalbos branduolys – dalykinis modelis. Tačiau vien jo nepakanka. Taip pat reikia aprašyti ir pavaizdavimo aspektus, tai yra, kaip informacija užkoduota modelio elementuose turi būti pavaizduota grafiniame naudotojo interfeise (GNI). Yra trys GNI langai, kuriuose vaizduojama informacija DSL Tools įrankyje: dizainerio erdvė, modelio tyrinėtojas ir savybių langas. Vaizdavimo aspekto apibrėžimas apima grafinės notacijos aprašymą, kurią naudoją dizaineris, tyrinėtojo išvaizdos

pritaikymas ir savybių lango išvaizdos pritaikymas. Svarbiausias, ko gero, yra grafinės notacijos aprašymas.

Grafinis dizaineris vaizduoja kai kuriuos modelio elementus dizainerio erdvėje per grafinę notaciją, kuri naudoja formas (angl. shapes) ir jungtis (angl. connectors). Formos atitinka dalykinės srities klases, o jungtys – ryšius tarp jų. Formos ir jungtys DSL apibrėžime turi įvairių savybių, kurios leidžia prisitaikyti spalvą ir stilių, pradinį sukurtos formos dydį, geometrinį formos pavidalą, ir panašiai.

Kad galima būtų modelio elementus pavaizduoti diagramoje, reikia apibrėžti:

- formos tipą ir išvaizda, kuri bus naudojama tiems elementams vaizduoti,
- sąryšį tarp formos apibrėžimo ir dalykinės srities klasės, kuris padiktuoja tiek išsidėstymo elgseną, tiek ir tai, kaip formos išvaizda priklauso nuo duomenų.

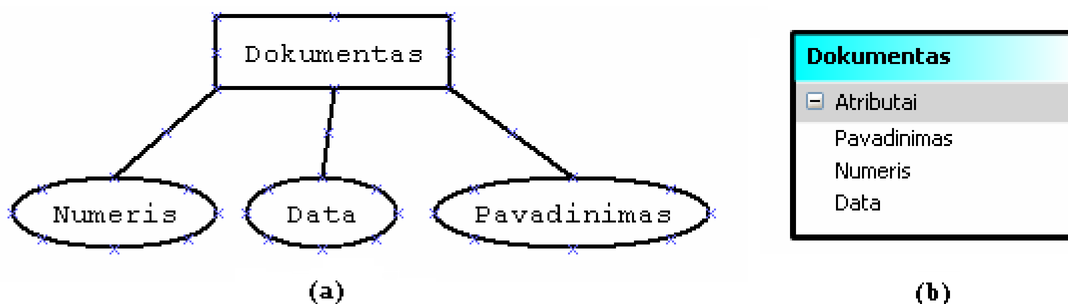
Kad pavaizduoti ryšius, reikia apibrėžti:

- jungties išvaizdą,
- jungties ir dalykinės srities ryšio atitikmenis, kurie nurodo jungties elgseną ir išvaizdą.

DSL Tools pateikia penkias skirtingas formų rūšis skirtas kalbos elementams vaizduoti: geometrinės formos (angl. geometry shapes), formos su sekcijomis (angl. compartment shapes), atvaizdų formos (angl. image shapes), „uostai“ (angl. ports) ir „plaukimo takeliai“ (angl. swim lanes).

Geometrinė forma yra paprasčiausias dalykinės srities klasių vaizdavimo būdas. Tai yra tiesiog tam tikra geometrinė forma (stačiakampis, apskritimas ir panašiai) turinti tam tikras savybes – dydį, spalvą, tekstą ir kitas. Formos su sekcijomis yra geometrinė forma turinti sekcijas. Sekcija naudojama pavaizduoti elementų, susijusių su vaizduojamu elementu, sąrašą. Atvaizdų forma paprasčiausiai vaizduoja paveiksliuką. „Uostas“ yra forma prikabinta prie kitos formos, ir gali būti judinama tik aplink tą formą. „Plaukimo takeliai“ naudojami padalinti diagramą į eilutes arba stulpelius.

Dokumentinės sistemos dalykiniame modelyje yra keturios pagrindinės klasės – „Dokumentas“, „Redakcija“, „Klasifikatorius“ ir „PapildomaInformacija“, ir visos jos savyje turi sąrašą susijusių atributų. Čia galimi du pavaizdavimo būdai. Pirmasis, tai būtų panašus į Esybių-Ryšių (ER) diagramų vaizdavimą, kur kiekvienos esybės atributai vaizduojami kaip atskiras mazgas, kaip pavaizduota pav. 12 (a).



12 pav. Klasės atributų vaizdavimas

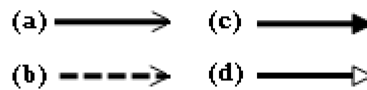
Tai, galbūt, patogu nedidelėms sistemoms, tačiau jei sistema sudėtinga, taip vaizduojama diagrama gali tapti perkrauta ir sunkiai suprantama. Dažniausiai tokiais atvejais naudojamas kompaktiškesnis vaizdavimo būdas, kuomet atributai išvardinami tame pačiame vaizdavimo elemente, kaip ir klasė. Tai būtų formos su sekcijomis atitikmuo. Todėl patogiausia dokumentinės sistemos modelio klases patogiausia būtų vaizduoti formomis su sekcijomis, kur sekcijose būtų sudėti atributai, kaip pavaizduota pav. 12 (b). Šis vaizdavimo būdas ir buvo pasirinktas visų anksčiau išvardintų klasių vaizdavimui.

Sukurti dalykinės srities klasių grafinę notaciją naudojant DSL Tools yra ganėtinai paprasta – reikia sukurti formą, nustatyti jos išvaizdą, bei formą sujungti su atitinkama dalykinės srities klase. Kitos dalykinės srities klasės – atributai, išreikštinės grafinės notacijos neturės. Jie bus kaip įrašai pagrindinių klasių sekcijose. Kaip tai atrodo, pamatysime pavyzdyje.

Taip kaip formos apibrėžia mazgus grafinėje notacijoje, taip jungtys (angl. Connector) apibrėžia ryšių išvaizdą. Ir lygiai taip pat kaip formos yra sujungiamos su dalykinės srities klasėmis, taip ryšiai sujungiami su jungtimis. Grafiškai jungtys yra paprasčiausios linijos jungiančios dvi formas. Jungtys yra kryptinės, tai yra, turi pradžią ir pabaigą. DSL Tools leidžia pasirinkti, kaip atrodo pati linija, kaip ji elgiasi, bei kaip atrodo jos galai.

Kuriamame dalykinės srities modelyje pavaizduoti reikia aštuonis ryšius: dokumento – redakcijos, dokumento – klasifikatoriaus, redakcijos – klasifikatoriaus, papildomos informacijos – klasifikatoriaus, dokumento – papildomos informacijos, redakcijos – papildomos informacijos, redakcijos – dokumento, papildomos informacijos – dokumento. Kad grafiškai neapkrauti būsimų diagramų, visas jungtis buvo nuspręsta apjungti į keturias grupes:

- dokumento ryšys su redakcija bus vaizduojamas ištisine rodykle pav. 13 (a);
- atgaliniai ryšiai į dokumentą iš redakcijos ir papildomos informacijos vaizduojami punktyrine linija pav. 13 (b);
- objektų ryšiai su klasifikatoriais vaizduojami ištisine linija su užpildyta rodykle pav. 13 (c);
- objektų ryšiai su papildoma informacija vaizduojami ištisine linija su tuščia rodykle pav. 13 (d).



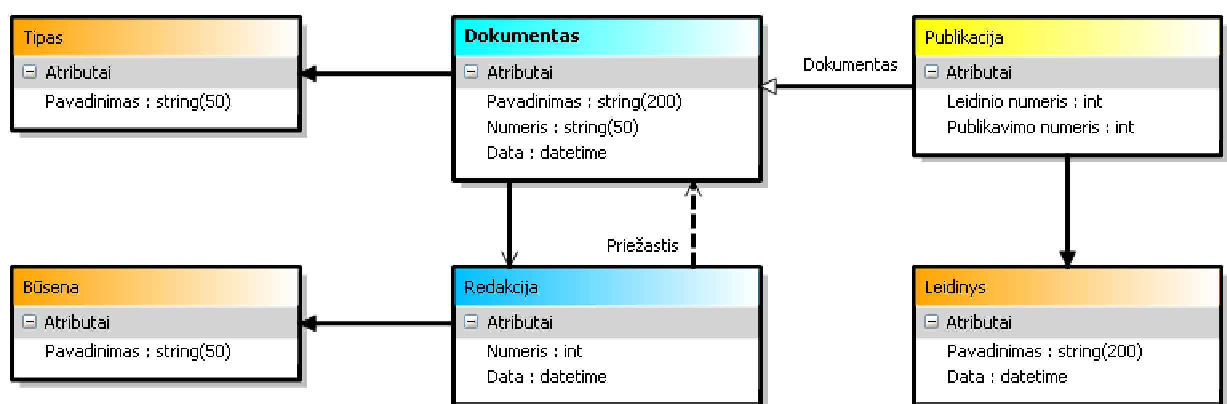
13 pav. Ryšių grafinė notacija

Dabar jau turime pilnai funkcionuojančią dalykinės srities kalbą – turime dalykinės srities modelį bei jo grafinę notaciją. Pilnas dalykinės srities modelio grafinis pavaizdavimas priede 4. Iki pilnavertės ir galinčios atnešti realios naudos kalbos dar trūksta nemažai – reikia aprašyti apribojimus, artefaktų (kodo, SQL ir panašiai) generatorius ir kitus dalykus, tačiau jau dabar galima pamėginti sudaryti testinį modelį testinei dokumentinei sistemai.

7.5. Testinė dokumentinė sistema su DSL Tools

Dokumentinės sistemos modelio kūrimui mums reikės DSL Tools pagalba sugeneruoti dalykinės srities redaktorių. Jeigu programų kūrimui naudojate įrankį Visual Studio, vaizdas bus labai pažįstamas. Programa turi įrankių juostą, dizainerio erdvę, modelio tyrinėjimo bei savybių langus (priedas 5).

Šio įrankio pagalba, panašiai kaip ir kuriant Windows aplikacijas su Visual Studio, arba kaip ir bet kuriuo UML redaktoriumi, iš įrankių juostos pasirenkame reikiamus dalykinės objektus ir jais modeliuojame sistemą. Skirtumas nuo UML tik toks, kad čia nereikia galvoti, kokį elementą pasirinkti, kokio tipo ryšį pasirinkti ir panašiai. Čia visos sąvokos yra gerai pažįstamos bet kuriam, bent kiek susidūrusiam su dokumentų valdymu. Modelis aprašytai testinei dokumentinei sistemai sudarytas naujai sukurtu įrankiu pavaizduotas pav. 14.



14 pav. Testinės dokumentinės sistemos modelis

7.6. Kodo generavimas

Toks DSL įrankis koks yra sukurtas iki šiol, kol kas realios naudos ar produktyvumo pagerinimo tikrai neatneštų, nes jis leidžia tik vizualiai projektuoti dokumentines sistemas. Kad įrankis būtų tikrai naudingas, reikia aprašyti kodo generatorius, kurie iš modelio sugeneruotų

bent jau tam tikras sistemos dalis. Ar tai bus tik karkasas ar pilnai veikianti sistema, čia jau priklauso nuo generatoriaus kūrėjo.

Šiame darbe iki šiol sukurta dokumentinių sistemų modeliavimo kalba kol kas apsiriboja tik duomenų sluoksnio modeliavimu, todėl generatorius taip pat sukurs tik duomenų sluoksnį. Tačiau to turėtų pakakti, kad susidaryti bendrą įspūdį ir suvokimą, kas yra kodo generatoriaus kūrimas.

7.6.1. Technologijos

Kodo generavimui pasirinkta Microsoft C# 2.0 programavimo kalba. Ši kalba pasirinkta visų pirma dėl unikalios jos savybės – dalinių klasių (angl. Partial Classes). Ši savybė leidžia tą pačią klasę aprašyti keliuose failuose, o tai ypač naudinga kuomet norima sugeneruotą kodą išplėsti papildomu funkcionalumu. Tuomet tiesiog generuotas kodas nekliudomas, o sukuriamas atskiras failas, kuriame ir realizuojamos papildomos savybės.

Duomenų saugojimui bus naudojama reliacinė duomenų bazė, todėl reikės sugeneruoti SQL kalbos sakinius, kurie sukurtų reikiamus objektus. Taip pat bus naudojamas esybių valdymo komponentas NHibernate, kuris susies C# klases su duomenų baze. NHibernate konfigūravimas vyksta per XML failus, kuriuose aprašoma, kokia klasė atitinką kokią lentelę duomenų bazėje, kokie atributai atitinka kokius lentelės stulpelius, įvairūs ryšiai ir panašiai. Taigi kuriamas generatorius sukurs ir šiuos konfigūracijos XML failus.

7.6.2. Generuojamų objektų identifikavimas

Toliau reikia identifikuoti į kokius artefaktus bus generuojami objektai esantys modelyje.

Visų pirma turime keturias pagrindines modelio klases – „Dokumentas“, „Redakcija“, „Papildoma Informacija“ bei „Klasifikatorius“, kurios bus transformuotos į C# klases. Klasės paveldėtos iš „Atributas“ bus transformuotos į atitinkamų C# klasių atributus.

„Dokumento“, „Redakcijos“ bei „Papildomos Informacijos“ ryšiai su „Klasifikatorium“, o taip pat ir „Dokumento“ ryšys su „Redakcija“ bus transformuoti į agregacijos ryšį. Tuo tarpu „Redakcijos“ ir „Papildomos Informacijos“ ryšys su „Dokumentu“, bei „Papildomos Informacijos“ ryšys su „Redakcija“ bus transformuoti į asociacijos tipo ryšius

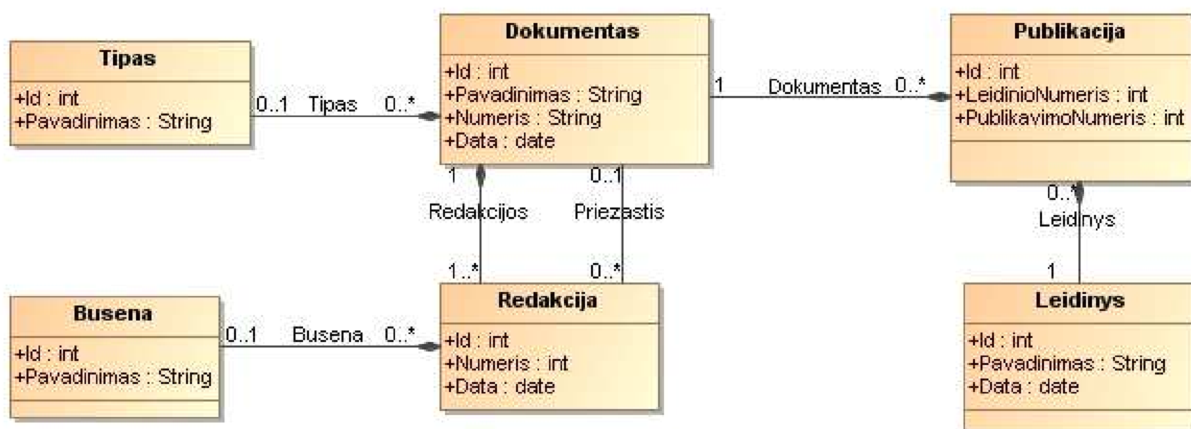
Duomenų bazės lygyje bus generuojamos dvi pagrindinės lentelės modelio klasėms „Dokumentas“ ir „Redakcija“, bei lentelės kiekvienam klasių „Klasifikatorius“ bei „Papildoma Informacija“ objektui. „Atributas“ klasės tipo objektai bus transformuojami į atitinkamų lentelių stulpelius. Taip pat bus sukuriami atitinkami ryšiai.

7.6.3. Kodo generatorius

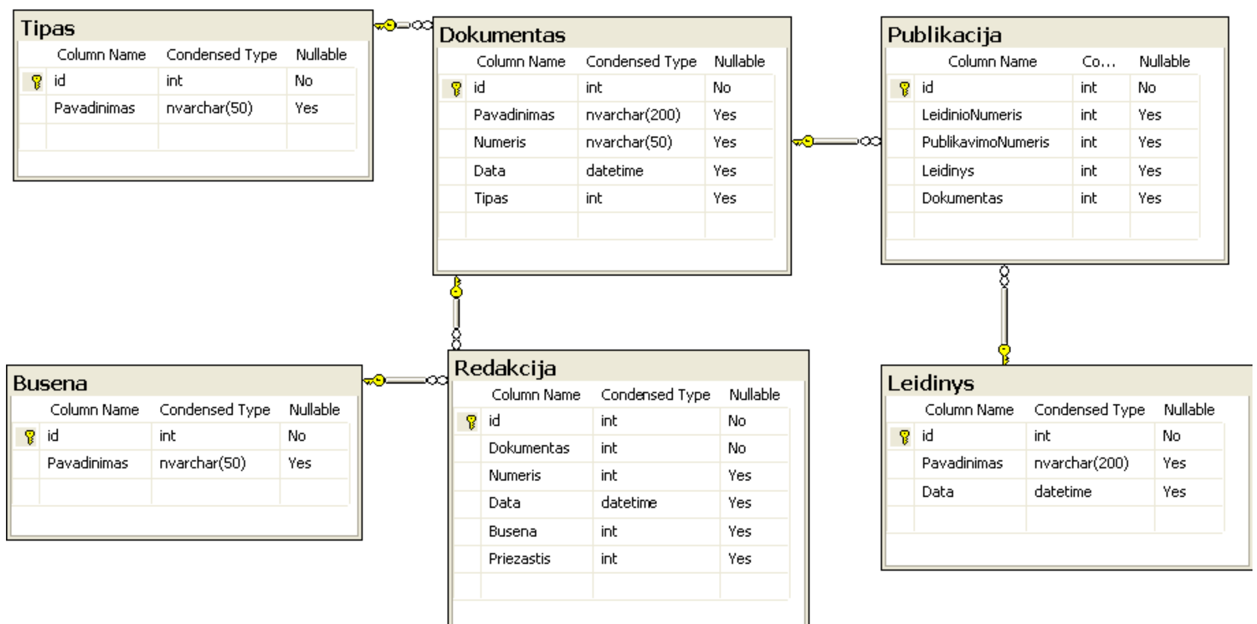
Kodo generavimui DSL Tools siūlo keletą variantų. Vienas iš jų, tai paprasčiausia XSLT transformacija, kadangi modeliai paprastai saugomi XML formatu. Šis metodas turi vieną didelį privalumą – tai greitis. Tačiau trūkumų yra kur kas daugiau ir jie yra sunkiai sprendžiami. Visų pirma, kodo generavime reikia daug dirbti su tekstine informacija, pavyzdžiui, paversti pavadinimą mažosiomis raidėmis, ką XSLT transformacijos metu padaryti nėra labai paprasta. O jei DSL pasiskirsto į keletą failų, XSLT kodas greitai tampa gana sunkiai sukuriamas ir palaikomas.

Kitas būdas, naudoti kartu su DSL modeliu sukuriamu kalbos API. Šiuo atveju, mes turime modelį kaip objektų rinkinį, kuriuo galime manipuliuoti savo sukurtoje programoje. Tačiau ir šis metodas turi vieną rimtą trūkumą – tai skaitomumas. Tikrasis generuojamas kodas pradingsta tarp kodo, kuris jį kuria. Todėl perskaityti tokį kodo generatorių bei jį palaikyti tampa gana sudėtinga.

Todėl buvo pasirinktas trečiasis būdas – šablonai. Naudojant šį būdą sukuriami šablonai – generuojamo kodo elementai, kuriuose specialios sintaksės pagalba įterpiami kintami modelio elementai. Šiame darbe kuriamos dokumentinės sistemos DSL įrankiui reikėjo sukurti šablonus visoms keturioms pagrindinėms modelio klasėms – dokumentui, redakcijai, klasifikatoriui bei papildomai informacijai. Kuriami šablonai buvo trijų tipų – .NET klases ir SQL sakinius generuojantys šablonai, bei esybių valdymo komponento XML konfigūracijos failams skirti generuoti šablonai. Kaip atrodo šablonai, galima pamatyti 6 – 8 prieduose. Paskui šie šablonai kartu su modeliu paduodami šablonų apdorojimo servisui, kuris sugeneruoja galutinius produktus – C# klases, duomenų bazės apibrėžimo SQL sakinius, bei NHibernate konfigūracijos XML failus. Pagal 14 paveiksle pateiktą dokumentinės sistemos modelį sugeneruoto C# kodo klasių diagramą galima pamatyti 15 paveiksle, o duomenų bazių schemą 16 paveiksle. O galutinius sugeneruotus produktus galima pažiūrėti 9 – 11 prieduose.



15 pav. Sugeneruoto kodo klasių diagrama



16 pav. Sugeneruotos duomenų bazės schema

7.7. DSL naudojimo apibendrinimas

Praktiškai mėginant DSM metodą buvo sukurtas realiai veikiantis dokumentinėms sistemoms, o tiksliau jų duomenų sluoksniui, skirtas kurti įrankis. Tam reikėjo apsibrėžti dalykinės srities modelį, sukonstruoti jį specialia programine įranga, parengti kodo generavimo šablonus bei visa tai apjungti į vien įranki. Kad tai įgyvendinti, visų pirma, būtinas dalykinę sritį išmanantis asmuo bei jo laikas, sugaištas bandant perprasti DSL kūrimui skirtą programinę įrangą, ir paskui kuriant ir patį DSL įrankį. Testinio DSL įrankio atveju, darbo įdėta daug, o rezultatas gautas toks pats, kaip ir MDA atveju be jokio pasiruošimo. Taigi galima teigti, kad DSL įrankių naudojimą galima svarstyti tik tuo atveju, jei įmonė užsiima išskirtinai vieno tipo produktų kūrimu ir yra pasiryžusi investuoti į DSL kūrimą, kuris apimtų daugiau nei vien duomenų sluoksnį. Net tuo atveju jei būtų nuspręsta naudoti šį metodą, nėra labai gera praktika pusmečiui dedikuoti žmogų DSL kūrimui ir paskui žiūrėti kas gavosi. DSL ir yra unikalus tuo, kad leidžia integruotis į kūrimo procesą iteracijomis. Pradžioje, gal būt, užteks sukurti duomenų sluoksnio generatorių, koks buvo sukurtas šio darbo metu. Vėliau pridėti saugumo, darbų sekos ir kitus aspektus, taip po truputį auginant įrankio vertę.

Rezultatai ir išvados

Šiame darbe buvo analizuojamos modeliais grįstų programų kūrimo metodų panaudojimo galimybės kuriant dokumentų valdymo sistemas. Šiam tikslui pasiekti pradžioje buvo iškelti uždaviniai, kuriuos pilnai pavyko įgyvendinti darbo metu.

Visų pirma buvo ištirtos bendrinės modeliais grįstų metodų MDA ir DSL stipriosios ir silpnosios savybės. Didžiausias MDA privalumas, bet kartu ir trūkumas – tai jos universalumas. Viena vertus, norint pradėti naudoti šį metodą praktiškai nereikia jokio pradinio pasiruošimo arba jis yra minimalus. Yra nemažai įrankių suderinamų su MDA, UML kalbos standartas yra plačiai žinomas ir pripažįstamas. Tačiau bendraujant su tam tikros srities specialistais tenka naudoti ne tos srities, o bendrinius, objektiškai orientuoto programavimo terminus, kas tam tikrais atvejais gali ganėtinai apsunkinti komunikaciją. Tuo tarpu DSL atvirkščiai – bendravimo barjerų nėra. Čia kalbant apie problemas naudojami tos srities terminai. Tačiau norint sukurti programinę įrangą, palaikančią tokį bendravimą, tenka investuoti nemažai resursų. Taigi pirmas faktorius renkantis modeliais grįstą metodą yra įmonės veiklos sritis. Jei įmonė kuria daug skirtingų produktų kurie neturi daug bendro, ar atlieka daug vienas su kitu nelabai susijusių projektų, tuomet pasirinkimas yra tik vienas – MDA. DSL paruošimo kaštai būtų per dideli, kad duotų naudos. Tačiau jei įmonė specializuojasi vienoje srityje, tuomet verta pagalvoti ir apie MDA.

Toliau darbe buvo akcentuojamas dokumentų valdymo sistemų kūrimas. Buvo apsibrėžta, kas tai yra dokumentų valdymo sistema, kokį funkcionalumą ir kokias savybes ji paprastai turi, ir pabandyta nustatyti, ką iš to rinkinio patogiausia ir naudingiausia yra modeliuoti. Paaiškėjo, kad praktiškai visi dokumentinių sistemų aspektai pakankamai lengvai pasiduoda modeliuojami, tačiau tolesniam tyrimui buvo apsiribota baziniu funkcionalumu.

Išrinktas pagrindinis funkcionalumas visų pirma buvo praktiškai mėginamas sumodeliuoti MDA metodui skirtais įrankiais. Paaiškėjo, kad šią dieną siūlomi MDA įrankiai, bent jau tie, kurie palaiko Microsoft .NET platformą, yra ganėtinai riboti. Jie geba generuoti kodą tik iš statinių UML diagramų, tokių kaip klasių ar komponentų diagramos. Sugeneruotas kodas paprastai bus tik duomenų sluoksnio karkasas, ne visada net ir veikiantis. Visos kitos diagramos naudingos tik dokumentavimui. Tačiau kartais to gali pilnai pakakti. Jei įmonė užsiima tik dokumentų valdymo sistemų kūrimu ir nėra linkusi investuoti į dalykus, kurie nežinia kada atsipirks, tačiau nori formalizuoti ir standartizuoti programų kūrimo procesą, MDA įrankiai gali būti puikus pasirinkimas.

Jei dokumentų valdymo sistemų kūrimu besispecializuojanti įmonė vis tik nuspręstų, kad verta investuoti į savos DSL kūrimą, darbe pateikiamas pavyzdys kaip tą galima būtų daryti.

Darbo metu buvo sukurta bazinė tačiau pilnai funkcionuojanti DSL, kuri geba sugeneruoti pilnai veikiantį dokumentų valdymo sistemos duomenų sluoksnį kartu su duomenų bazės schema. Šią dokumentų valdymo sistemų DSL galima imti kaip pagrindą, išplėsti ją tiek kiek reikia ir naudoti ją savo tikslams.

Naudota literatūra

[ARM01] Sreenivas A, Venkatesh R and Joseph M. Meta-modelling for Formal Software Development in Proceedings of Computing: the Australian Theory Symposium. Gold Coast, Australia, 2001.

[Bla07] Deirdre Blake. Domain-Specific Languages versus Generic Modeling Languages. 2007.
[žiūrėta 2007-04-03]. Prieiga per Internetą:
http://www.ddj.com/dept/architect/199500627

[CC94] de Cima, Werner, and Cerqueira. The Design of Object-Oriented Software with Domain Architecture Reuse. Third International Conference on Software Reuse: Advances in Software Reusability, 1994.

[CJK07] Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.

[Cle01] J. Craig Cleveland. Program Generators with XML and Java. Upper Saddle River: Prentice Hall, 2001.

[CWM] Object Management Group. The Common Warehouse Metamodel.
[žiūrėta 2007-04-19]. Prieiga per Internetą:
http://www.cwmforum.org

[DKV00] van Deursen, Klint and Visser. Domain-Specific Languages: An Annotated Bibliography. 2000.
[žiūrėta 2007-04-15]. Prieiga per Internetą:
http://homepages.cwi.nl/~arie/papers/dslbib/dslbib.html

[DSM] DSM Forum. What is Domain-Specific Modeling?.
[žiūrėta 2007-04-02]. Prieiga per Internetą:
http://www.dsmforum.org

[Gre04] Keith Jack Greenfield. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Ohn Wiley & Sons, 2004.

[Kel06] Steven Kelly. Generating Code with DSM. MetaCase, 2006.
[žiūrėta 2007-04-10]. Prieiga per Internetą:
http://www.codegeneration.net/tiki-read_article.php?articleId=81

[KR05] Vinay Kulkarni, Sreedhar Reddy. Gererating enterprise applications From models – experiece and best practices. Tata Research Development and Design Centre, 2005.
[žiūrėta 2007-04-10]. Prieiga per Internetą:
http://www.softmetaware.com/oopsla2005/kulkarni.pdf

[Kru92] Charles W. Krueger. Software Reuse. ACM Computing Surveys, 1992.

[MDA01] OMG Architecture Board MDA Drafting Team. Model-Driven Architecture: A Technical Perspective. 2001.
[žiūrėta 2007-04-17]. Prieiga per Internetą:
ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf

[MOF] OMG Meta Object Facility Specification, Version 1.3. September, 1999.
[žiūrėta 2007-04-25]. Prieiga per Internetą:
http://www.omg.org

[Poo01] John D. Poole. Model-Driven Architecture: Vision, Standards And Emerging Technologies. Hyperion Solutions Corporation, April 2001 .

[RB05] Evgeny Rahman, Jarrod Bellmore. Domain Specific Language Development Process. Worcester Polytechnic Institute, 2005.

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. IBM Rational Software, 2003.
[žiūrėta 2007-04-20]. Prieiga per Internetą:
www.inf.ed.ac.uk/teaching/courses/seoc/2006_2007/resources/Mod_MDD2.pdf

[Tol00] D. Tolbert. CWM: A Model-Based Architecture for Data Warehouse Interchange. Workshop on Evaluating Software Architectural Solutions 2000, University of California at Irvine, May, 2000
[žiūrėta 2007-04-13]. Prieiga per Internetą:
http://www.cwmforum.org/uciwesas2000.htm

[WK06] Model-driven architecture. Wikipedia, 2006.
[žiūrėta 2007-04-05]. Prieiga per Internetą:
http://en.wikipedia.org/wiki/Model-driven_architecture

[WK07] Document management system. Wikipedia, 2007.
[žiūrėta 2008-01-05]. Prieiga per Internetą:
http://en.wikipedia.org/wiki/Document_Management

[XMI] Object Management Group. XML Metadata Interchange Specification, Version 1.1.
[žiūrėta 2008-01-05]. Prieiga per Internetą:
http://www.omg.org

Priedas 1. MDA nuo platformos nepriklausomas modelis

Priedas 2. MDA specifinis platformai modelis

Priedas 3. MDA sugeneruotas kodas

```
//..begin "File Description"
/*--------------------------------------------------------------------------------*
   Filename:  Dokumentas.cs
   Tool:      objectiF, CSharpSSvr V7.0.155
 *--------------------------------------------------------------------------------*/
//..end "File Description"

using System;
using System.Collections;
using Gentle.Framework;
using Microtool.PersistenceNet.SessionManager;


namespace MyCompany.MyApplication.Entities
{
    [TableName("Dokumentas")]
    public class Dokumentas
    : Microtool.PersistenceNet.SessionManager.PersistentBase
    {
        public int Version
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.version_;
            }
        }

        [TableColumn("Version", NotNull = true), Concurrency()]
        private int version_;

        public int Id
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.id_;
            }
        }

        [TableColumn("Id", NotNull = true), PrimaryKey(AutoGenerated = true)]
        private int id_;

        public static Dokumentas RetrieveInstance(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Dokumentas dokumentas = (Dokumentas)RetrieveInstance(typeof(Dokumentas), key);
            return dokumentas;
        }

        public static Dokumentas[] RetrieveList(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            IList list = RetrieveList(typeof(Dokumentas), key);
            Dokumentas[] result = new Dokumentas[list.Count];
            for (int i = 0; i < list.Count; i++)
            {
                result[i] = (Dokumentas)list[i];
            }
            return result;
        }

        public static Dokumentas Retrieve(int id)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Dokumentas), true, "Id", id);
            return RetrieveInstance(typeof(Dokumentas), key) as Dokumentas;
        }

        public static Dokumentas[] RetrieveAll()
```

```
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Dokumentas), false);
            return Dokumentas.RetrieveList(key);
        }

        public static Dokumentas Create(DokumentoTipas dokumentoTipas)
        {
            // Begin of generated code. Do not edit. Modifications will be lost.

            Dokumentas newDokumentas = new Dokumentas(dokumentoTipas);

            // End of generated code.

            //..begin "Body"

            // TODO: remove warning statement after manual code review.
#warning [OPTIONAL] Manual review of generated code is recommended.

            //..end "Body"
            // Begin of generated code. Do not edit. Modifications will be lost.

            newDokumentas.Persist();
            return newDokumentas;

        }

        public DokumentoTipas DokumentoTipas
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                return DokumentoTipas.Retrieve(this.dokumentoTipasId_);
            }

            set
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                this.dokumentoTipasId_ = (value == null) ? 0 : value.Id;
                this.PersistAtCommit();
            }
        }

        [TableColumn("DokumentoTipasId", NotNull = true), ForeignKey("DokumentoTipas", "Id")]
        private int dokumentoTipasId_;

        public Redakcija[] RedakcijosList
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                Key key = new Key("Redakcija", null, true, "Dokumentas", this.Id);
                return Redakcija.RetrieveList(key);
            }

            set
            {
// Generated code. Editing should be unnecessary. Modifications will be preserved.

                foreach (Redakcija redakcija in value)
                {
                    redakcija.Dokumentas = this;
                }
            }
        }

        public Publikavimas[] PublikavimasList
        {
            get
            {
```

```
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                Key key = new Key("Publikavimas", null, true, "Dokumentas", this.Id);
                return Publikavimas.RetrieveList(key);
            }
        }

        internal Dokumentas()
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.id_ = 0;
            this.version_ = 0;
        }

        internal Dokumentas(DokumentoTipas dokumentoTipas)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.dokumentoTipasId_ = (dokumentoTipas == null) ? 0 : dokumentoTipas.Id;
        }

        internal Dokumentas(int version, int id, int dokumentoTipasId)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.version_ = version;
            this.id_ = id;
            this.dokumentoTipasId_ = dokumentoTipasId;
        }
    }
}
```

```
//..begin "File Description"
/*--------------------------------------------------------------------------------*
   Filename:  Redakcija.cs
   Tool:      objectiF, CSharpSSvr V7.0.155
 *--------------------------------------------------------------------------------*/
//..end "File Description"

using System;
using System.Collections;
using Gentle.Framework;
using Microtool.PersistenceNet.SessionManager;


namespace MyCompany.MyApplication.Entities
{
    [TableName("Redakcija")]
    public class Redakcija
    : Microtool.PersistenceNet.SessionManager.PersistentBase
    {
        internal Dokumentas Dokumentas
        {
            set
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                this.dokumentasId_ = (value == null) ? 0 : value.Id;
                this.PersistAtCommit();
            }
        }

        [TableColumn("DokumentasId", NotNull = true), ForeignKey("Dokumentas", "Id")]
        private int dokumentasId_;

        public int Version
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.version_;
            }
        }

        [TableColumn("Version", NotNull = true), Concurrency()]
        private int version_;

        public int Id
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.id_;
            }
        }

        [TableColumn("Id", NotNull = true), PrimaryKey(AutoGenerated = true)]
        private int id_;

        public static Redakcija RetrieveInstance(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Redakcija redakcija = (Redakcija)RetrieveInstance(typeof(Redakcija), key);
            return redakcija;
        }

        public static Redakcija[] RetrieveList(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            IList list = RetrieveList(typeof(Redakcija), key);
            Redakcija[] result = new Redakcija[list.Count];
            for (int i = 0; i < list.Count; i++)
            {
                result[i] = (Redakcija)list[i];
```

```
            }
            return result;
        }

        public static Redakcija Retrieve(int id)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Redakcija), true, "Id", id);
            return RetrieveInstance(typeof(Redakcija), key) as Redakcija;
        }

        public static Redakcija[] RetrieveAll()
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Redakcija), false);
            return Redakcija.RetrieveList(key);
        }

        public static Redakcija Create(Dokumentas dokumentas)
        {
            // Begin of generated code. Do not edit. Modifications will be lost.

            Redakcija newRedakcija = new Redakcija(dokumentas);

            // End of generated code.

            //..begin "Body"

            // TODO: remove warning statement after manual code review.
#warning [OPTIONAL] Manual review of generated code is recommended.

            //..end "Body"
            // Begin of generated code. Do not edit. Modifications will be lost.

            newRedakcija.Persist();
            return newRedakcija;

        }

        internal Redakcija()
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.id_ = 0;
            this.version_ = 0;
        }

        internal Redakcija(Dokumentas dokumentas)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.dokumentasId_ = (dokumentas == null) ? 0 : dokumentas.Id;
        }

        internal Redakcija(int dokumentasId, int version, int id)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.dokumentasId_ = dokumentasId;
            this.version_ = version;
            this.id_ = id;
        }
    }
}
```

```
//..begin "File Description"
/*--------------------------------------------------------------------------------*
   Filename:  DokumentoTipas.cs
   Tool:      objectiF, CSharpSSvr V7.0.155
 *--------------------------------------------------------------------------------*/
//..end "File Description"

using System;
using System.Collections;
using Gentle.Framework;
using Microtool.PersistenceNet.SessionManager;


namespace MyCompany.MyApplication.Entities
{
    [TableName("DokumentoTipas")]
    public class DokumentoTipas
    : Microtool.PersistenceNet.SessionManager.PersistentBase
    {
        public Dokumentas[] DokumentaiList
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                Key key = new Key("Dokumentas", null, true, "DokumentoTipas", this.Id);
                return Dokumentas.RetrieveList(key);
            }
        }

        public int Version
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.version_;
            }
        }

        [TableColumn("Version", NotNull = true), Concurrency()]
        private int version_;

        public int Id
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.id_;
            }
        }

        [TableColumn("Id", NotNull = true), PrimaryKey(AutoGenerated = true)]
        private int id_;

        public static DokumentoTipas RetrieveInstance(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            DokumentoTipas dokumentoTipas =
(DokumentoTipas)RetrieveInstance(typeof(DokumentoTipas), key);
            return dokumentoTipas;
        }

        public static DokumentoTipas[] RetrieveList(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            IList list = RetrieveList(typeof(DokumentoTipas), key);
            DokumentoTipas[] result = new DokumentoTipas[list.Count];
            for (int i = 0; i < list.Count; i++)
            {
                result[i] = (DokumentoTipas)list[i];
            }
            return result;
        }
```

```
        public static DokumentoTipas Retrieve(int id)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(DokumentoTipas), true, "Id", id);
            return RetrieveInstance(typeof(DokumentoTipas), key) as DokumentoTipas;
        }

        public static DokumentoTipas[] RetrieveAll()
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(DokumentoTipas), false);
            return DokumentoTipas.RetrieveList(key);
        }

        public static DokumentoTipas Create()
        {
            // Begin of generated code. Do not edit. Modifications will be lost.

            DokumentoTipas newDokumentoTipas = new DokumentoTipas();

            // End of generated code.

            //..begin "Body"

            // TODO: remove warning statement after manual code review.
#warning [OPTIONAL] Manual review of generated code is recommended.

            //..end "Body"
            // Begin of generated code. Do not edit. Modifications will be lost.

            newDokumentoTipas.Persist();
            return newDokumentoTipas;

        }

        internal DokumentoTipas()
        {
            // Fully generated code. Do not edit. Modifications will be lost.

        }

        internal DokumentoTipas(int version, int id)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.version_ = version;
            this.id_ = id;
        }
    }
}
```

```
//..begin "File Description"
/*--------------------------------------------------------------------------------*
   Filename:  Publikavimas.cs
   Tool:      objectiF, CSharpSSvr V7.0.155
 *--------------------------------------------------------------------------------*/
//..end "File Description"

using System;
using System.Collections;
using Gentle.Framework;
using Microtool.PersistenceNet.SessionManager;


namespace MyCompany.MyApplication.Entities
{
    [TableName("Publikavimas")]
    public class Publikavimas
    : Microtool.PersistenceNet.SessionManager.PersistentBase
    {
        public Dokumentas Dokumentas
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                return Dokumentas.Retrieve(this.dokumentasId_);
            }

            set
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                this.dokumentasId_ = (value == null) ? 0 : value.Id;
                this.PersistAtCommit();
            }
        }

        [TableColumn("DokumentasId", NotNull = true), ForeignKey("Dokumentas", "Id")]
        private int dokumentasId_;

        public PublikavimoSaltinis PublikavimoSaltinis
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                return PublikavimoSaltinis.Retrieve(this.publikavimoSaltinisId_);
            }

            set
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                this.RefreshOnDemand();
                this.publikavimoSaltinisId_ = (value == null) ? 0 : value.Id;
                this.PersistAtCommit();
            }
        }

        [TableColumn("PublikavimoSaltinisId", NotNull = true), ForeignKey("PublikavimoSaltinis",
"Id")]
        private int publikavimoSaltinisId_;

        public int Version
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.version_;
            }
```

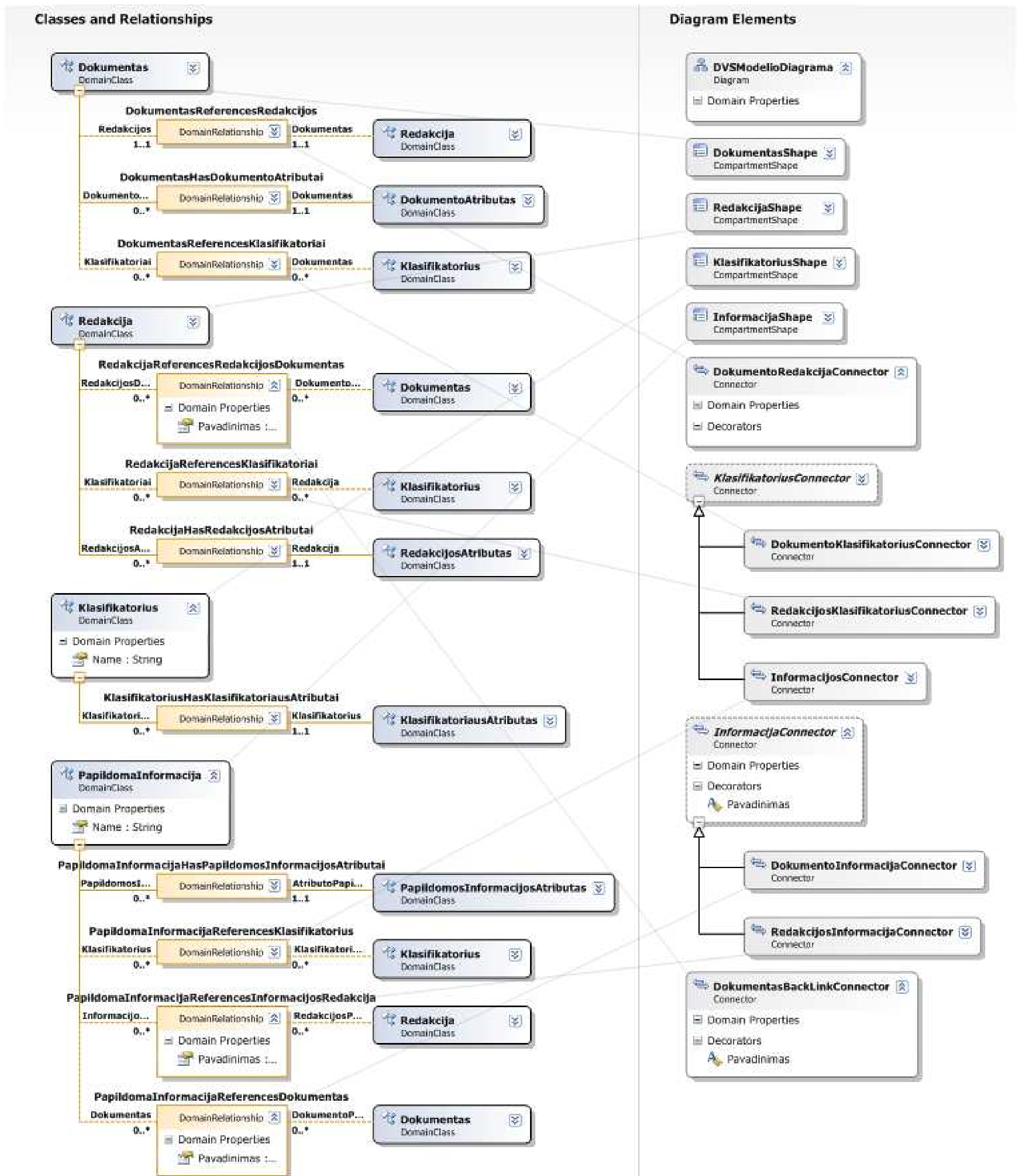
```
        }

        [TableColumn("Version", NotNull = true), Concurrency()]
        private int version_;

        public int Id
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.id_;
            }
        }

        [TableColumn("Id", NotNull = true), PrimaryKey(AutoGenerated = true)]
        private int id_;

        public static Publikavimas RetrieveInstance(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Publikavimas publikavimas = (Publikavimas)RetrieveInstance(typeof(Publikavimas),
key);
            return publikavimas;
        }

        public static Publikavimas[] RetrieveList(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            IList list = RetrieveList(typeof(Publikavimas), key);
            Publikavimas[] result = new Publikavimas[list.Count];
            for (int i = 0; i < list.Count; i++)
            {
                result[i] = (Publikavimas)list[i];
            }
            return result;
        }

        public static Publikavimas Retrieve(int id)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Publikavimas), true, "Id", id);
            return RetrieveInstance(typeof(Publikavimas), key) as Publikavimas;
        }

        public static Publikavimas[] RetrieveAll()
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(Publikavimas), false);
            return Publikavimas.RetrieveList(key);
        }

        public static Publikavimas Create(Dokumentas dokumentas, PublikavimoSaltinis
publikavimoSaltinis)
        {
            // Begin of generated code. Do not edit. Modifications will be lost.

            Publikavimas newPublikavimas = new Publikavimas(dokumentas, publikavimoSaltinis);

            // End of generated code.

            //..begin "Body"

            // TODO: remove warning statement after manual code review.
#warning [OPTIONAL] Manual review of generated code is recommended.

            //..end "Body"
            // Begin of generated code. Do not edit. Modifications will be lost.

            newPublikavimas.Persist();
            return newPublikavimas;

        }

        internal Publikavimas()
        {
```

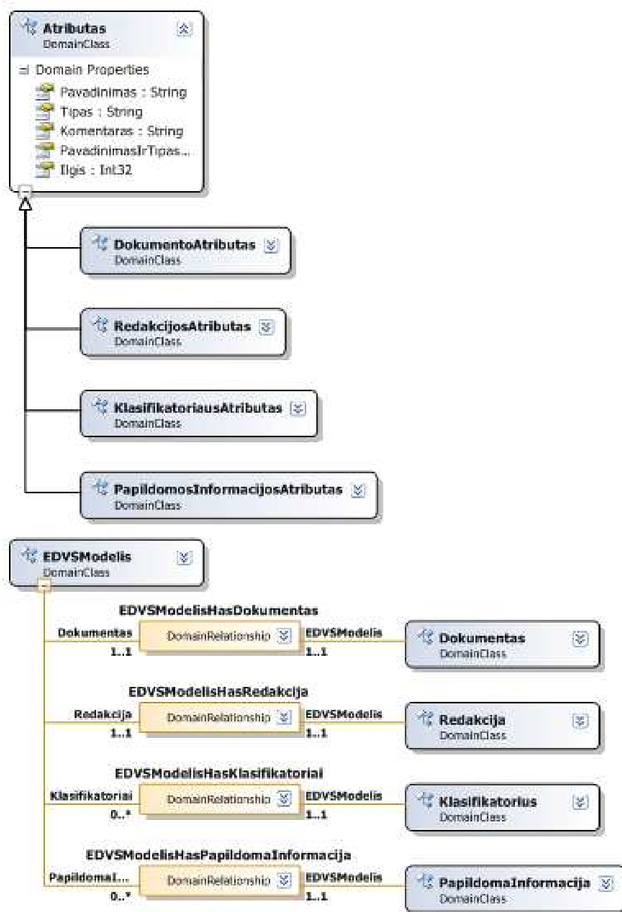
```
            // Fully generated code. Do not edit. Modifications will be lost.

            this.id_ = 0;
            this.version_ = 0;
        }

        internal Publikavimas(Dokumentas dokumentas, PublikavimoSaltinis publikavimoSaltinis)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.dokumentasId_ = (dokumentas == null) ? 0 : dokumentas.Id;
            this.publikavimoSaltinisId_ = (publikavimoSaltinis == null) ? 0 :
publikavimoSaltinis.Id;
        }

        internal Publikavimas(int dokumentasId, int publikavimoSaltinisId, int version, int id)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.dokumentasId_ = dokumentasId;
            this.publikavimoSaltinisId_ = publikavimoSaltinisId;
            this.version_ = version;
            this.id_ = id;
        }
    }
}
```

```
//..begin "File Description"
/*--------------------------------------------------------------------------------*
   Filename:  PublikavimoSaltinis.cs
   Tool:      objectiF, CSharpSSvr V7.0.155
 *--------------------------------------------------------------------------------*/
//..end "File Description"

using System;
using System.Collections;
using Gentle.Framework;
using Microtool.PersistenceNet.SessionManager;


namespace MyCompany.MyApplication.Entities
{
    [TableName("PublikavimoSaltinis")]
    public class PublikavimoSaltinis
    : Microtool.PersistenceNet.SessionManager.PersistentBase
    {
        public Publikavimas[] PublikavimasList
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.

                Key key = new Key("Publikavimas", null, true, "PublikavimoSaltinis", this.Id);
                return Publikavimas.RetrieveList(key);
            }
        }

        public int Version
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.version_;
            }
        }

        [TableColumn("Version", NotNull = true), Concurrency()]
        private int version_;

        public int Id
        {
            get
            {
                // Generated code. Editing should be unnecessary. Modifications will be
preserved.
                return this.id_;
            }
        }

        [TableColumn("Id", NotNull = true), PrimaryKey(AutoGenerated = true)]
        private int id_;

        public static PublikavimoSaltinis RetrieveInstance(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            PublikavimoSaltinis publikavimoSaltinis =
(PublikavimoSaltinis)RetrieveInstance(typeof(PublikavimoSaltinis), key);
            return publikavimoSaltinis;
        }

        public static PublikavimoSaltinis[] RetrieveList(Key key)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            IList list = RetrieveList(typeof(PublikavimoSaltinis), key);
            PublikavimoSaltinis[] result = new PublikavimoSaltinis[list.Count];
            for (int i = 0; i < list.Count; i++)
            {
                result[i] = (PublikavimoSaltinis)list[i];
            }
            return result;
        }
```

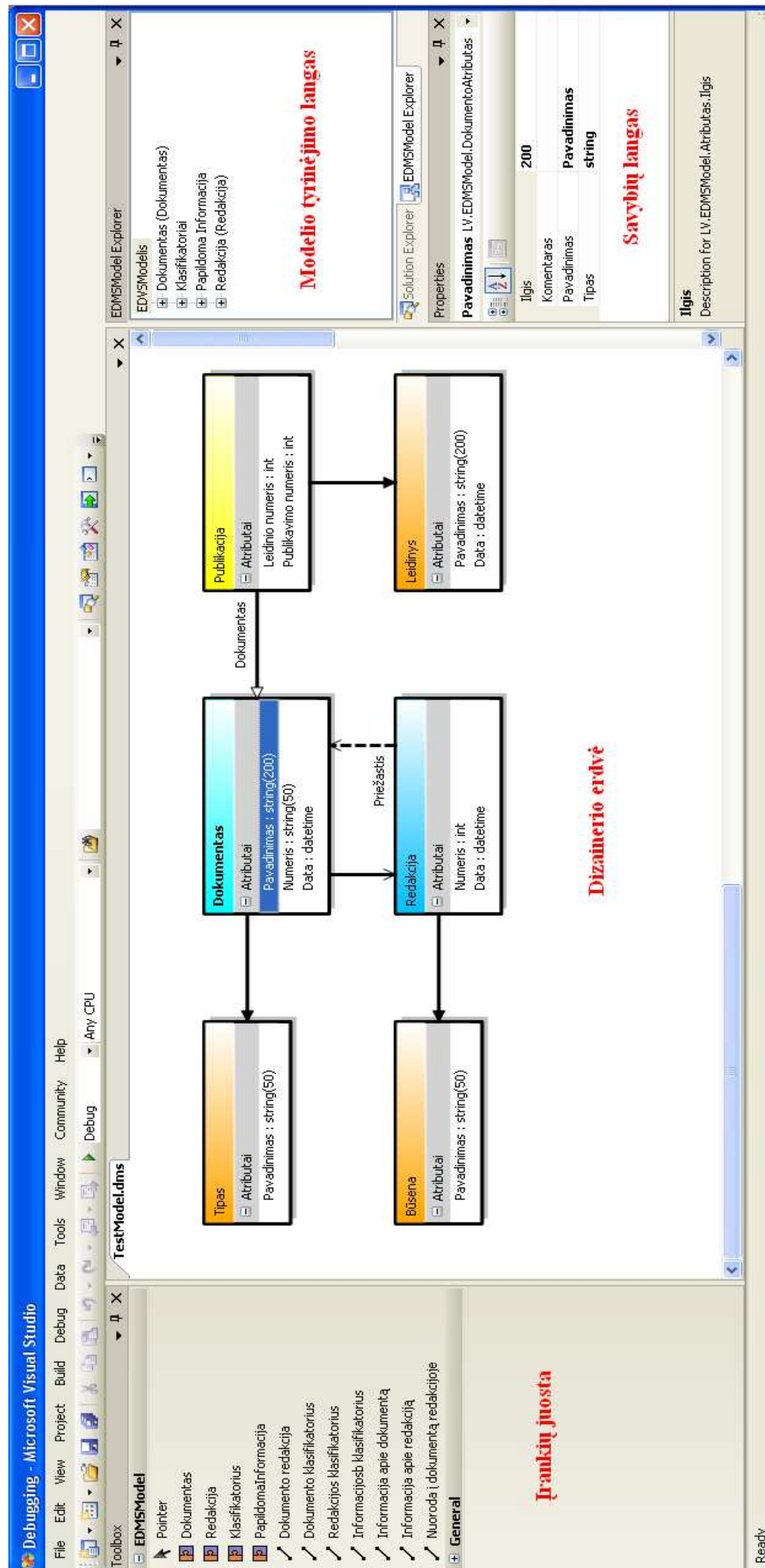
```
        public static PublikavimoSaltinis Retrieve(int id)
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(PublikavimoSaltinis), true, "Id", id);
            return RetrieveInstance(typeof(PublikavimoSaltinis), key) as PublikavimoSaltinis;
        }

        public static PublikavimoSaltinis[] RetrieveAll()
        {
            // Generated code. Editing should be unnecessary. Modifications will be preserved.

            Key key = new Key(typeof(PublikavimoSaltinis), false);
            return PublikavimoSaltinis.RetrieveList(key);
        }

        public static PublikavimoSaltinis Create()
        {
            // Begin of generated code. Do not edit. Modifications will be lost.

            PublikavimoSaltinis newPublikavimoSaltinis = new PublikavimoSaltinis();

            // End of generated code.

            //..begin "Body"

            // TODO: remove warning statement after manual code review.
#warning [OPTIONAL] Manual review of generated code is recommended.

            //..end "Body"
            // Begin of generated code. Do not edit. Modifications will be lost.

            newPublikavimoSaltinis.Persist();
            return newPublikavimoSaltinis;

        }

        internal PublikavimoSaltinis()
        {
            // Fully generated code. Do not edit. Modifications will be lost.

        }

        internal PublikavimoSaltinis(int version, int id)
        {
            // Fully generated code. Do not edit. Modifications will be lost.

            this.version_ = version;
            this.id_ = id;
        }
    }
}
```

Priedas 4. Dokumentų valdymo sistemos dalykinės srities modelis




Atributas
DomainClass
Domain Properties
Pavadinimas : String
Tipas : String
Komentaras : String
PavadinimasIrTipas...
Ilgis : Int32
DokumentoAtributas
DomainClass
RedakcijosAtributas
DomainClass
KlasifikatoriausAtributas
DomainClass
PapildomosInformacijosAtributas
DomainClass
EDVSModelis
DomainClass
EDVSModelisHasDokumentas
Dokumentas
1..1
DomainRelationship
EDVSModelis
1..1
Dokumentas
DomainClass
EDVSModelisHasRedakcija
Redakcija
1..1
DomainRelationship
EDVSModelis
1..1
Redakcija
DomainClass
EDVSModelisHasKlasifikatoriai
Klasifikatoriai
0..*
DomainRelationship
EDVSModelis
1..1
Klasifikatorius
DomainClass
EDVSModelisHasPapildomaInformacija
PapildomaI...
0..*
DomainRelationship
EDVSModelis
1..1
PapildomaInformacija
DomainClass

Priedas 5. Dalykinės srities redaktorius



Priedas 6. C# kodo šablonai

A) Dokumento šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>
using System;
using System.Collections;
using System.Text;

namespace EDMSModel.Entities
{

	public partial class Dokumentas
	{
		private int id;
<#
foreach (DokumentoAtributas attr in this.Dokumentas.DokumentoAtributai)
{
#>
		private <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, true) #>;
<#
}
#>
<#
foreach (Klasifikatorius c in this.Dokumentas.Klasifikatoriai)
{
#>
		private <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, true) #>;
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesDokumentas> piDoc =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesDokume
ntas>();
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
		private IList <#= NetGetValidName(pi.PapildomaInformacija.Name,
true) #>;
<#
}
#>

		private IList redakcijos;

		public Dokumentas()
		{
		}

		public int Id
		{
			get { return this.id; }
			set { this.id = value; }
```

```
		}

<#
foreach (DokumentoAtributas attr in this.Dokumentas.DokumentoAtributai)
{
#>
		public <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, false) #>
		{
			get { return this.<#= NetGetValidName(attr.Pavadinimas, true)
#>; }
			set { this.<#= NetGetValidName(attr.Pavadinimas, true) #> =
value; }
		}

<#
}
#>
<#
foreach (Klasifikatorius c in this.Dokumentas.Klasifikatoriai)
{
#>
		public <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, false) #>
		{
			get { return this.<#= NetGetValidName(c.Name, true) #>; }
			set { this.<#= NetGetValidName(c.Name, true) #> = value; }
		}

<#
}
#>
<#
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
		public IList <#= NetGetValidName(pi.PapildomaInformacija.Name,
false) #>
		{
			get { return this.<#=
NetGetValidName(pi.PapildomaInformacija.Name, true) #>; }
			set { this.<#= NetGetValidName(pi.PapildomaInformacija.Name,
true) #> = value; }
		}
<#
}
#>
		public IList Redakcijos
		{
			get { return this.redakcijos; }
			set { this.redakcijos = value; }
		}
	}
}

@@NETHELPER@@
```

B) Redakcijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>
using System;
using System.Collections;
using System.Text;

namespace EDMSModel.Entities
{
     public partial class Redakcija
     {
          private int id;
          private Dokumentas dokumentas;
<#
foreach (RedakcijosAtributas attr in this.Redakcija.RedakcijosAtributai)
{
#>
          private <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, true) #>;
<#
}
#>
<#
foreach (Klasifikatorius c in this.Redakcija.Klasifikatoriai)
{
#>
          private <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, true) #>;
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesInformacijosRedakcija> piRed =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesInform
acijosRedakcija>();
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
          private IList <#= NetGetValidName(pi.PapildomaInformacija.Name,
true) #>;
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<RedakcijaReferencesRedakcij
osDokumentas> rdList =
this.Store.ElementDirectory.FindElements<RedakcijaReferencesRedakcijosDokumen
tas>();
foreach (RedakcijaReferencesRedakcijosDokumentas rd in rdList)
{
#>
          private Dokumentas <#= NetGetValidName(rd.Pavadinimas, true) #>;
<#
}
#>
```

```
          public Redakcija()
          {
          }

          public int Id
          {
               get { return this.id; }
               set { this.id = value; }
          }

          public Dokumentas Dokumentas
          {
               get { return this.dokumentas; }
               set { this.dokumentas = value; }
          }

<#
foreach (RedakcijosAtributas attr in this.Redakcija.RedakcijosAtributai)
{
#>
          public <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, false) #>
          {
               get { return this.<#= NetGetValidName(attr.Pavadinimas, true)
#>; }
               set { this.<#= NetGetValidName(attr.Pavadinimas, true) #> =
value; }
          }

<#
}
#>
<#
foreach (Klasifikatorius c in this.Redakcija.Klasifikatoriai)
{
#>
          public <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, false) #>
          {
               get { return this.<#= NetGetValidName(c.Name, true) #>; }
               set { this.<#= NetGetValidName(c.Name, true) #> = value; }
          }

<#
}
#>
<#
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
          public IList <#= NetGetValidName(pi.PapildomaInformacija.Name,
false) #>
          {
               get { return this.<#=
NetGetValidName(pi.PapildomaInformacija.Name, true) #>; }
               set { this.<#= NetGetValidName(pi.PapildomaInformacija.Name,
true) #> = value; }
          }
<#
}
#>
<#
foreach (RedakcijaReferencesRedakcijosDokumentas rd in rdList)
{
```

```
#>
          public Dokumentas <#= NetGetValidName(rd.Pavadinimas, false) #>
          {
               get { return this.<#= NetGetValidName(rd.Pavadinimas, true) #>;
}
               set { this.<#= NetGetValidName(rd.Pavadinimas, true) #> =
value; }
          }
<#
}
#>
     }
}

@@NETHELPER@@
```

C) Klasifikatoriaus šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".cs" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>
using System;
using System.Collections;
using System.Text;

namespace EDMSModel.Entities
{

	public partial class <#= NetGetValidName(this.Klasifikatorius.Name,
false) #>
	{
		private int id;
<#
foreach (KlasifikatoriausAtributas attr in
this.Klasifikatorius.KlasifikatoriausAtributai)
{
#>
		private <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, true) #>;
<#
}
#>

		public <#= NetGetValidName(this.Klasifikatorius.Name, false) #>()
		{
		}

		public int Id
		{
			get { return this.id; }
			set { this.id = value; }
		}

<#
foreach (KlasifikatoriausAtributas attr in
this.Klasifikatorius.KlasifikatoriausAtributai)
{
#>
		public <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, false) #>
		{
			get { return this.<#= NetGetValidName(attr.Pavadinimas, true)
#>; }
			set { this.<#= NetGetValidName(attr.Pavadinimas, true) #> =
value; }
		}

<#
}
#>
	}
}

@@NETHELPER@@
```

D) Papildomos informacijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>
using System;
using System.Collections;
using System.Text;

namespace EDMSModel.Entities
{
     public partial class <#= NetGetValidName(this.Informacija.Name, false) #>
     {
          private int id;
<#
foreach (PapildomosInformacijosAtributas attr in
this.Informacija.PapildomosInformacijosAtributai)
{
#>
          private <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, true) #>;
<#
}
#>
<#
foreach (Klasifikatorius c in this.Informacija.Klasifikatorius)
{
#>
          private <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, true) #>;
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesDokumentas> piDoc =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesDokume
ntas>();
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
          private Dokumentas <#= NetGetValidName(pi.Pavadinimas, true) #>;
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesInformacijosRedakcija> piRed =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesInform
acijosRedakcija>();
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
          private Redakcija <#= NetGetValidName(pi.Pavadinimas, true) #>;
<#
}
#>

          public <#= NetGetValidName(this.Informacija.Name, false) #>()
```

```
        {
        }

        public int Id
        {
            get { return this.id; }
            set { this.id = value; }
        }

<#
foreach (PapildomosInformacijosAtributas attr in
this.Informacija.PapildomosInformacijosAtributai)
{
#>
        public <#= NetGetType(attr.Tipas) #> <#=
NetGetValidName(attr.Pavadinimas, false) #>
        {
            get { return this.<#= NetGetValidName(attr.Pavadinimas, true)
#>; }
            set { this.<#= NetGetValidName(attr.Pavadinimas, true) #> =
value; }
        }

<#
}
#>
<#
foreach (Klasifikatorius c in this.Informacija.Klasifikatorius)
{
#>
        public <#= NetGetValidName(c.Name, false) #> <#=
NetGetValidName(c.Name, false) #>
        {
            get { return this.<#= NetGetValidName(c.Name, true) #>; }
            set { this.<#= NetGetValidName(c.Name, true) #> = value; }
        }
<#
}
#>
<#
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
        public Dokumentas <#= NetGetValidName(pi.Pavadinimas, false) #>
        {
            get { return this.<#= NetGetValidName(pi.Pavadinimas, true) #>;
}
            set { this.<#= NetGetValidName(pi.Pavadinimas, true) #> =
value; }
        }
<#
}
#>
<#
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
        public Redakcija <#= NetGetValidName(pi.Pavadinimas, false) #>
        {
            get { return this.<#= NetGetValidName(pi.Pavadinimas, true) #>;
}
            set { this.<#= NetGetValidName(pi.Pavadinimas, true) #> =
value; }
        }
```

```
<#
}
#>
        }
}
```

@@NETHELPER@@

Priedas 7. SQL šablonai

A) Dokumento šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>
CREATE TABLE dbo.[Dokumentas]
(
     [id] int IDENTITY(1,1) NOT NULL,
<#
foreach (DokumentoAtributas attr in this.Dokumentas.DokumentoAtributai)
{
#>
     [<#= SqlGetValidName(attr.Pavadinimas) #>] <#= SqlGetType(attr.Tipas)
#><#= SqlGetLength(attr.Ilgis) #>,
<#
}
#>
<#
foreach (Klasifikatorius c in this.Dokumentas.Klasifikatoriai)
{
#>
     [<#= SqlGetValidName(c.Name) #>] int,
<#
}
#>
     CONSTRAINT [PK_Dokumentas] PRIMARY KEY
     (
          [id] ASC
     )
)

<#
foreach (Klasifikatorius c in this.Dokumentas.Klasifikatoriai)
{
#>
ALTER TABLE [dbo].[Dokumentas] WITH CHECK ADD CONSTRAINT [FK_Dokumentas_<#=
SqlGetValidName(c.Name) #>] FOREIGN KEY([<#= SqlGetValidName(c.Name) #>])
REFERENCES [dbo].[<#= SqlGetValidName(c.Name) #>] ([id])

<#
}
#>

@@SQLHELPER@@
```

B) Redakcijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>

CREATE TABLE dbo.[Redakcija]
(
     [id] int IDENTITY(1,1) NOT NULL,
     [Dokumentas] int NOT NULL,
<#
foreach (RedakcijosAtributas attr in this.Redakcija.RedakcijosAtributai)
{
#>
     [<#= SqlGetValidName(attr.Pavadinimas) #>] <#= SqlGetType(attr.Tipas)
#><#= SqlGetLength(attr.Ilgis) #>,
<#
}
#>
<#
foreach (Klasifikatorius c in this.Redakcija.Klasifikatoriai)
{
#>
     [<#= SqlGetValidName(c.Name) #>] int,
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<RedakcijaReferencesRedakcij
osDokumentas> rdList =
this.Store.ElementDirectory.FindElements<RedakcijaReferencesRedakcijosDokumen
tas>();
foreach (RedakcijaReferencesRedakcijosDokumentas rd in rdList)
{
#>
     [<#= SqlGetValidName(rd.Pavadinimas) #>] int,
<#
}
#>
     CONSTRAINT [PK_Redakcija] PRIMARY KEY
     (
          [id] ASC
     )
)

ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT
[FK_Redakcija_Dokumentas] FOREIGN KEY([Dokumentas])
REFERENCES [dbo].[Dokumentas] ([id])

<#
foreach (Klasifikatorius c in this.Redakcija.Klasifikatoriai)
{
#>
ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT [FK_Redakcija_<#=
SqlGetValidName(c.Name) #>] FOREIGN KEY([<#= SqlGetValidName(c.Name) #>])
REFERENCES [dbo].[<#= SqlGetValidName(c.Name) #>] ([id])

<#
}
```

```
#>

<#
foreach (RedakcijaReferencesRedakcijosDokumentas rd in rdList)
{
#>
ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT
[FK_Redakcija_Dokumentas_<#= SqlGetValidName(rd.Pavadinimas) #>] FOREIGN
KEY([<#= SqlGetValidName(rd.Pavadinimas) #>])
REFERENCES [dbo].[Dokumentas] ([id])

<#
}
#>


@@SQLHELPER@@
```

C) Klasifikatoriaus šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>

CREATE TABLE dbo.[<#= SqlGetValidName(this.Klasifikatorius.Name) #>]
(
     [id] int IDENTITY(1,1) NOT NULL,
<#
foreach (KlasifikatoriausAtributas attr in
this.Klasifikatorius.KlasifikatoriausAtributai)
{
#>
     [<#= SqlGetValidName(attr.Pavadinimas) #>] <#= SqlGetType(attr.Tipas)
#><#= SqlGetLength(attr.Ilgis) #>,
<#
}
#>
     CONSTRAINT [PK_<#= SqlGetValidName(this.Klasifikatorius.Name) #>] PRIMARY
KEY
     (
          [id] ASC
     )
)

@@SQLHELPER@@
```

D) Papildomos informacijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>

CREATE TABLE dbo.[<#= SqlGetValidName(this.Informacija.Name) #>]
(
     [id] int IDENTITY(1,1) NOT NULL,
<#
foreach (PapildomosInformacijosAtributas attr in
this.Informacija.PapildomosInformacijosAtributai)
{
#>
     [<#= SqlGetValidName(attr.Pavadinimas) #>] <#= SqlGetType(attr.Tipas)
#><#= SqlGetLength(attr.Ilgis) #>,
<#
}
#>
<#
foreach (Klasifikatorius c in this.Informacija.Klasifikatorius)
{
#>
     [<#= SqlGetValidName(c.Name) #>] int,
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesDokumentas> piDoc =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesDokume
ntas>();
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
     [<#= SqlGetValidName(pi.Pavadinimas) #>] int,
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesInformacijosRedakcija> piRed =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesInform
acijosRedakcija>();
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
     [<#= SqlGetValidName(pi.Pavadinimas) #>] int,
<#
}
#>
     CONSTRAINT [PK_<#= SqlGetValidName(this.Informacija.Name) #>] PRIMARY KEY
     (
          [id] ASC
     )
)

<#
foreach (Klasifikatorius c in this.Informacija.Klasifikatorius)
```

```
{
#>
ALTER TABLE [dbo].[<#= SqlGetValidName(this.Informacija.Name) #>] WITH CHECK
ADD CONSTRAINT [FK_<#= SqlGetValidName(this.Informacija.Name) #>_<#=
SqlGetValidName(c.Name) #>] FOREIGN KEY([<#= SqlGetValidName(c.Name) #>])
REFERENCES [dbo].[<#= SqlGetValidName(c.Name) #>] ([id])

<#
}
#>

<#
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
ALTER TABLE [dbo].[<#= SqlGetValidName(this.Informacija.Name) #>] WITH CHECK
ADD CONSTRAINT [FK_<#= SqlGetValidName(this.Informacija.Name)
#>_Dokumentas_<#= SqlGetValidName(pi.Pavadinimas) #>] FOREIGN KEY([<#=
SqlGetValidName(pi.Pavadinimas) #>])
REFERENCES [dbo].[Dokumentas] ([id])

<#
}
#>

<#
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
ALTER TABLE [dbo].[<#= SqlGetValidName(this.Informacija.Name) #>] WITH CHECK
ADD CONSTRAINT [FK_<#= SqlGetValidName(this.Informacija.Name)
#>_Redakcija_<#= SqlGetValidName(pi.Pavadinimas) #>] FOREIGN KEY([<#=
SqlGetValidName(pi.Pavadinimas) #>])
REFERENCES [dbo].[Redakcija] ([id])

<#
}
#>

@@SQLHELPER@@
```

Priedas 8. NHibernate XML konfigūracijos šablonai

A) Dokumento šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="Dokumentas" table="Dokumentas">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
<#
foreach (DokumentoAtributas attr in this.Dokumentas.DokumentoAtributai)
{
#>
          <property name="<#= NetGetValidName(attr.Pavadinimas, false) #>"
                    column="<#= SqlGetValidName(attr.Pavadinimas) #>" />
<#
}
#>
          <set
               name="Redakcijos"
               inverse="true"
               lazy="true"
               cascade="all">
               <key column="Dokumentas"/>
               <one-to-many class="Redakcija"/>
          </set>
<#
foreach (Klasifikatorius c in this.Dokumentas.Klasifikatoriai)
{
#>
          <one-to-one name="<#= NetGetValidName(c.Name, false) #>" column="<#=
SqlGetValidName(c.Name) #>" not-null="false"/>
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesDokumentas> piDoc =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesDokume
ntas>();
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
          <set
               name="<#= NetGetValidName(pi.PapildomaInformacija.Name, false)
#>"
               inverse="true"
               lazy="true"
               cascade="all">
               <key column="<#= SqlGetValidName(pi.Pavadinimas) #>"/>
               <one-to-many class="<#=
NetGetValidName(pi.PapildomaInformacija.Name, false) #>"/>
```

```
          </set>
<#
}
#>
     </class>
</hibernate-mapping>

@@SQLHELPER@@
@@NETHELPER@@
```

B) Redakcijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".sql" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@'"  #>
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="Redakcija" table="Redakcija">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
<#
foreach (RedakcijosAtributas attr in this.Redakcija.RedakcijosAtributai)
{
#>
          <property name="<#= NetGetValidName(attr.Pavadinimas, false) #>">
               <column name="<#= SqlGetValidName(attr.Pavadinimas) #>" />
          </property>
<#
}
#>
          <many-to-one name="Dokumentas" column="Dokumentas" not-null="true"/>
<#
foreach (Klasifikatorius c in this.Redakcija.Klasifikatoriai)
{
#>
          <one-to-one name="<#= NetGetValidName(c.Name, false) #>" column="<#=
SqlGetValidName(c.Name) #>" not-null="false"/>
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesInformacijosRedakcija> piRed =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesInform
acijosRedakcija>();
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
          <set
               name="<#= NetGetValidName(pi.PapildomaInformacija.Name, false)
#>"
               inverse="true"
               lazy="true"
               cascade="all">
               <key column="<#= SqlGetValidName(pi.Pavadinimas) #>"/>
               <one-to-many class="<#=
NetGetValidName(pi.PapildomaInformacija.Name, false) #>"/>
          </set>
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<RedakcijaReferencesRedakcij
osDokumentas> rdList =
```

```
this.Store.ElementDirectory.FindElements<RedakcijaReferencesRedakcijosDokumen
tas>();
foreach (RedakcijaReferencesRedakcijosDokumentas rd in rdList)
{
#>
          <one-to-one name="<#= NetGetValidName(rd.Pavadinimas, false) #>"
column="<#= SqlGetValidName(rd.Pavadinimas) #>" not-null="false"
class="Dokumentas" />
<#
}
#>
     </class>
</hibernate-mapping>

@@SQLHELPER@@
@@NETHELPER@@
```

C) Klasifikatoriaus šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".xml" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="<#= NetGetValidName(this.Klasifikatorius.Name, false) #>"
table="<#= SqlGetValidName(this.Klasifikatorius.Name) #>">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
<#
foreach (KlasifikatoriausAtributas attr in
this.Klasifikatorius.KlasifikatoriausAtributai)
{
#>
          <property name="<#= NetGetValidName(attr.Pavadinimas, false) #>">
               <column name="<#= SqlGetValidName(attr.Pavadinimas) #>" />
          </property>
<#
}
#>
     </class>
</hibernate-mapping>

@@SQLHELPER@@
@@NETHELPER@@
```

D) Papildomos informacijos šablonas

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransforma
tion" debug="true"#>
<#@ output extension=".xml" #>
<#@ EDMSModel processor="EDMSModelDirectiveProcessor"
requires="fileName='@@MODELFILE@@';name='@@CLASSNAME@@'"  #>
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="<#= NetGetValidName(this.Informacija.Name, false) #>"
table="<#= SqlGetValidName(this.Informacija.Name) #>">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
<#
foreach (PapildomosInformacijosAtributas attr in
this.Informacija.PapildomosInformacijosAtributai)
{
#>
          <property name="<#= NetGetValidName(attr.Pavadinimas, false) #>">
               <column name="<#= SqlGetValidName(attr.Pavadinimas) #>" />
          </property>
<#
}
#>
<#
foreach (Klasifikatorius c in this.Informacija.Klasifikatorius)
{
#>
          <one-to-one name="<#= NetGetValidName(c.Name, false) #>" column="<#=
SqlGetValidName(c.Name) #>" not-null="false"/>

<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesDokumentas> piDoc =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesDokume
ntas>();
foreach (PapildomaInformacijaReferencesDokumentas pi in piDoc)
{
#>
          <many-to-one name="<#= NetGetValidName(pi.Pavadinimas, false) #>"
column="<#= SqlGetValidName(pi.Pavadinimas) #>" not-null="true"/>
<#
}
#>
<#
System.Collections.ObjectModel.ReadOnlyCollection<PapildomaInformacijaReferen
cesInformacijosRedakcija> piRed =
this.Store.ElementDirectory.FindElements<PapildomaInformacijaReferencesInform
acijosRedakcija>();
foreach (PapildomaInformacijaReferencesInformacijosRedakcija pi in piRed)
{
#>
          <many-to-one name="<#= NetGetValidName(pi.Pavadinimas, false) #>"
column="<#= SqlGetValidName(pi.Pavadinimas) #>" not-null="true"/>
```

```
<#
}
#>
     </class>
</hibernate-mapping>

@@SQLHELPER@@
@@NETHELPER@@
```

Priedas 9. DSL sugeneruotas C# kodas

```
public partial class Dokumentas
{
    private int id;
    private string pavadinimas;
    private string numeris;
    private DateTime data;
    private Tipas tipas;
    private IList publikacija;

    private IList redakcijos;

    public Dokumentas()
    {
    }

    public int Id
    {
        get { return this.id; }
        set { this.id = value; }
    }

    public string Pavadinimas
    {
        get { return this.pavadinimas; }
        set { this.pavadinimas = value; }
    }

    public string Numeris
    {
        get { return this.numeris; }
        set { this.numeris = value; }
    }

    public DateTime Data
    {
        get { return this.data; }
        set { this.data = value; }
    }


    public Tipas Tipas
    {
        get { return this.tipas; }
        set { this.tipas = value; }
    }

    public IList Publikacija
    {
        get { return this.publikacija; }
        set { this.publikacija = value; }
    }
    public IList Redakcijos
    {
        get { return this.redakcijos; }
        set { this.redakcijos = value; }
    }
}
```

```
public partial class Redakcija
{
    private int id;
    private Dokumentas dokumentas;
    private int numeris;
    private DateTime data;
    private Busena busena;
    private Dokumentas priezastis;

    public Redakcija()
    {
    }

    public int Id
    {
        get { return this.id; }
        set { this.id = value; }
    }

    public Dokumentas Dokumentas
    {
        get { return this.dokumentas; }
        set { this.dokumentas = value; }
    }

    public int Numeris
    {
        get { return this.numeris; }
        set { this.numeris = value; }
    }

    public DateTime Data
    {
        get { return this.data; }
        set { this.data = value; }
    }


    public Busena Busena
    {
        get { return this.busena; }
        set { this.busena = value; }
    }

    public Dokumentas Priezastis
    {
        get { return this.priezastis; }
        set { this.priezastis = value; }
    }
}
```

```
public partial class Busena
{
	private int id;
	private string pavadinimas;

	public Busena()
	{
	}

	public int Id
	{
		get { return this.id; }
		set { this.id = value; }
	}

	public string Pavadinimas
	{
		get { return this.pavadinimas; }
		set { this.pavadinimas = value; }
	}

}

public partial class Leidinys
{
	private int id;
	private string pavadinimas;
	private DateTime data;

	public Leidinys()
	{
	}

	public int Id
	{
		get { return this.id; }
		set { this.id = value; }
	}

	public string Pavadinimas
	{
		get { return this.pavadinimas; }
		set { this.pavadinimas = value; }
	}

	public DateTime Data
	{
		get { return this.data; }
		set { this.data = value; }
	}

}
```

```
public partial class Publikacija
{
    private int id;
    private int leidinioNumeris;
    private int publikavimoNumeris;
    private Leidinys leidinys;
    private Dokumentas dokumentas;

    public Publikacija()
    {
    }

    public int Id
    {
        get { return this.id; }
        set { this.id = value; }
    }

    public int LeidinioNumeris
    {
        get { return this.leidinioNumeris; }
        set { this.leidinioNumeris = value; }
    }

    public int PublikavimoNumeris
    {
        get { return this.publikavimoNumeris; }
        set { this.publikavimoNumeris = value; }
    }

    public Leidinys Leidinys
    {
        get { return this.leidinys; }
        set { this.leidinys = value; }
    }
    public Dokumentas Dokumentas
    {
        get { return this.dokumentas; }
        set { this.dokumentas = value; }
    }
}

public partial class Tipas
{
    private int id;
    private string pavadinimas;

    public Tipas()
    {
    }

    public int Id
    {
        get { return this.id; }
        set { this.id = value; }
    }

    public string Pavadinimas
    {
        get { return this.pavadinimas; }
        set { this.pavadinimas = value; }
    }

}
```

Priedas 10. DSL sugeneruotas SQL kodas

```
CREATE TABLE dbo.[Tipas]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[Pavadinimas] nvarchar(50),
	CONSTRAINT [PK_Tipas] PRIMARY KEY
	(
		[id] ASC
	)
)


CREATE TABLE dbo.[Dokumentas]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[Pavadinimas] nvarchar(200),
	[Numeris] nvarchar(50),
	[Data] datetime,
	[Tipas] int,
	CONSTRAINT [PK_Dokumentas] PRIMARY KEY
	(
		[id] ASC
	)
)

ALTER TABLE [dbo].[Dokumentas] WITH CHECK ADD CONSTRAINT
[FK_Dokumentas_Tipas] FOREIGN KEY([Tipas])
REFERENCES [dbo].[Tipas] ([id])

CREATE TABLE dbo.[Busena]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[Pavadinimas] nvarchar(50),
	CONSTRAINT [PK_Busena] PRIMARY KEY
	(
		[id] ASC
	)
)

CREATE TABLE dbo.[Redakcija]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[Dokumentas] int NOT NULL,
	[Numeris] int,
	[Data] datetime,
	[Busena] int,
	[Priezastis] int,
	CONSTRAINT [PK_Redakcija] PRIMARY KEY
	(
		[id] ASC
	)
)

ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT
[FK_Redakcija_Dokumentas] FOREIGN KEY([Dokumentas])
REFERENCES [dbo].[Dokumentas] ([id])

ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT [FK_Redakcija_Busena]
FOREIGN KEY([Busena])
REFERENCES [dbo].[Busena] ([id])
```

```
ALTER TABLE [dbo].[Redakcija] WITH CHECK ADD CONSTRAINT
[FK_Redakcija_Dokumentas_Priezastis] FOREIGN KEY([Priezastis])
REFERENCES [dbo].[Dokumentas] ([id])


CREATE TABLE dbo.[Leidinys]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[Pavadinimas] nvarchar(200),
	[Data] datetime,
	CONSTRAINT [PK_Leidinys] PRIMARY KEY
	(
		[id] ASC
	)
)

CREATE TABLE dbo.[Publikacija]
(
	[id] int IDENTITY(1,1) NOT NULL,
	[LeidinioNumeris] int,
	[PublikavimoNumeris] int,
	[Leidinys] int,
	[Dokumentas] int,
	CONSTRAINT [PK_Publikacija] PRIMARY KEY
	(
		[id] ASC
	)
)

ALTER TABLE [dbo].[Publikacija] WITH CHECK ADD CONSTRAINT
[FK_Publikacija_Leidinys] FOREIGN KEY([Leidinys])
REFERENCES [dbo].[Leidinys] ([id])


ALTER TABLE [dbo].[Publikacija] WITH CHECK ADD CONSTRAINT
[FK_Publikacija_Dokumentas_Dokumentas] FOREIGN KEY([Dokumentas])
REFERENCES [dbo].[Dokumentas] ([id])
```

Priedas 11. DSL sugeneruota NHibernate XML konfigūracija

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="Dokumentas" table="Dokumentas">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
          <property name="Pavadinimas"
                    column="Pavadinimas" />
          <property name="Numeris"
                    column="Numeris" />
          <property name="Data"
                    column="Data" />
          <set
               name="Redakcijos"
               inverse="true"
               lazy="true"
               cascade="all">
               <key column="Dokumentas"/>
               <one-to-many class="Redakcija"/>
          </set>
          <one-to-one name="Tipas" column="Tipas" not-null="false"/>
          <set
               name="Publikacija"
               inverse="true"
               lazy="true"
               cascade="all">
               <key column="Dokumentas"/>
               <one-to-many class="Publikacija"/>
          </set>

     </class>
</hibernate-mapping>


<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="Redakcija" table="Redakcija">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
          <property name="Numeris">
               <column name="Numeris" />
          </property>
          <property name="Data">
               <column name="Data" />
          </property>
          <many-to-one name="Dokumentas" column="Dokumentas" not-null="true"/>
          <one-to-one name="Busena" column="Busena" not-null="false"/>
          <one-to-one name="Priezastis" column="Priezastis" not-null="false"
class="Dokumentas" />
     </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

    <class name="Busena" table="Busena">
        <id name="Id">
            <column name="Id" sql-type="int" not-null="true"/>
            <generator class="identity" />
        </id>
        <property name="Pavadinimas">
            <column name="Pavadinimas" />
        </property>
    </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

    <class name="Leidinys" table="Leidinys">
        <id name="Id">
            <column name="Id" sql-type="int" not-null="true"/>
            <generator class="identity" />
        </id>
        <property name="Pavadinimas">
            <column name="Pavadinimas" />
        </property>
        <property name="Data">
            <column name="Data" />
        </property>
    </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

    <class name="Publikacija" table="Publikacija">
        <id name="Id">
            <column name="Id" sql-type="int" not-null="true"/>
            <generator class="identity" />
        </id>
        <property name="LeidinioNumeris">
            <column name="LeidinioNumeris" />
        </property>
        <property name="PublikavimoNumeris">
            <column name="PublikavimoNumeris" />
        </property>
        <one-to-one name="Leidinys" column="Leidinys" not-null="false"/>

        <many-to-one name="Dokumentas" column="Dokumentas" not-null="true"/>

    </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
     namespace="EDMSModel.Entities" assembly="EDMSModel.Entities">

     <class name="Tipas" table="Tipas">
          <id name="Id">
               <column name="Id" sql-type="int" not-null="true"/>
               <generator class="identity" />
          </id>
          <property name="Pavadinimas">
               <column name="Pavadinimas" />
          </property>
     </class>
</hibernate-mapping>
```