KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Tomas Škultinas

MDA PANAUDOJIMO PROGRAMINĖS ĮRANGOS KŪRIMUI TYRIMAS

Magistro darbas

Darbo vadovas
doc. V. Pilkauskas

Kaunas, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
prof. habil. dr. H. Pranevičius
2005 05

MDA PANAUDOJIMO PROGRAMINĖS ĮRANGOS KŪRIMUI TYRIMAS

Informatikos mokslo magistro baigiamasis darbas

Kalbos konsultantė
Lietuvių katedros lekt.
dr. Jurgita Mikelionienė
2005 05

Vadovas
doc. dr. V. Pilkauskas
2005 05

Recenzentas
doc. dr. D. Rubliauskas
2005 05

Atliko
IFM-9/1 gr. stud.
T. Škultinas
2005 05

Kaunas, 2005

KVALIFIKACINĖ KOMISIJA

Pirmininkas: Laimutis Telksnys, akademikas

Sekretorius: Stasys Maciulevičius, docentas

Nariai:

Rimantas Barauskas, profesorius

Raimundas Jasinevičius, profesorius

Jonas Kazimieras Matickas, docentas

Jonas Mockus, akademikas

Rimantas Plėštys, docentas

Henrikas Pranevičius, profesorius

SANTRAUKA

Informacinių technologijų industrija nuolatos ieško būdų, kaip padidinti programinės įrangos kūrimo produktyvumą, kokybę ir ilgaamžiškumą. OMG konsorciumo vystoma modeliais pagrista architektūra (MDA) yra programinės įrangos metodika, pateikianti visiškai naują požiūrį į programinės įrangos kūrimo procesą.

Remiantis OMG specifikacijų ir literatūros šaltiniais, darbe analizuojami pagrindiniai MDA architektūros elementai ir jų funkcijos - tai sistemos probleminės srities modeliavimas ir sukurtų modelių transformavimas. Šio darbo tikslas yra įvertinti MDA aplinkos panaudojimo galimybes programinės įrangos kūrimui. Naudojantis MDA architektūros analizės rezultatais yra realizuota MDA aplinka. Eksperimentinio MDA aplinkos naudojimo metu yra vertinamas programinės įrangos kūrimo produktyvumas, pasikartojančių užduočių automatizavimas ir techninių įgūdžių reikalavimai programinės įrangos kūrėjams.

SUMMARY

IT industry all time is looking for ways to improve software development productivity as well as the quality and longevity of the software that it creates. OMG announced Model Driven Architecture as its strategic direction. It is software development methodology that provides new viewpoint in software development process.

The modeling of problem domain and model transformation are key elements of MDA architecture and they are analyzed in this work using OMG specifications and other resources. The purpose of this work is to evaluate benefits of MDA framework in software development process. The new MDA framework is developed according to the results of MDA architecture analysis. Experimental usage of new MDA framework concentrates on productivity of software development process, automation of repeated tasks and required skill set of application developers.

TURINYS

LENTELĖS

PAVEIKSLAI

1 ĮVADAS

Programinės įrangos kūrimo efektyvumas dažnai yra lyginamas su technines įrangos vystymusi. Paskutinius dvidešimt metų kompiuterių procesorių našumas didėja eksponentiškai. Mūro dėsnis (*Moore's Law*) taikomas aparatinės įrangos pramonei ir pasireiškiantis eksponentiniu procesorių, atmintinių ir duomenų saugyklų našumo didėjimu, vis dar galioja. Lyginant su didžiuliu progresu techninės įrangos srityje, programinės įrangos kūrimo vystymosi tempai yra kuklesni. Informacinių technologijų (IT) industrija nuolatos ieško būdų, kaip padidinti programinės įrangos produktyvumą, kokybę ir ilgaamžiškumą. Tai objektinis (*Object-orientied*) programavimas, komponentais paremtas (*Component-based*) programavimas, projektavimo šablonai (*Design patterns*), paskirstytojo skaičiavimo (*Distributed computing*) infrastruktūros. Buvo pastebėta, kad skirtingi programinės įrangos abstrakcijos lygiai pasižymi nevienoda kaitos sparta. Kaip taisyklė: kuo didesnis abstrakcijos lygis – tuo mažesnis pasikeitimų skaičius. IT specialistai nuolatos rekomenduoja projektuoti programinės įrangos dalis prieš rašant išeities kodą. Naudinga atlikti aukštesnio lygio programines įrangos architektūrinį modeliavimą prieš projektuojant atskiras programinės įrangos dalis. Tokios objektinės, ketvirtos kartos programavimo kalbos (4GL), tokios kaip C++, C# ar Java yra nuolatos vystomos ir baigia visiškai išstumti funkcinio programavimo kalbas, nes pasižymi daug didesniu abstrakcijos lygiu ir turi galingas išeities kodo derinimo programas (*Source-level debugger*). Taip pat dideliu tempu vystomos tokios įmonės lygio (*Enterprise*) platformos, kaip J2EE (*Java2 Enterprise Edition*) ar .NET. Programinės įrangos struktūrizavimas ir modeliavimas tapo pastovesnis, kai atsirado standartizuotos modeliavimo kalbos, pavyzdžiui OMG (*Object Management Group*) konsorciumo vieninga modeliavimo kalba UML (*Unified Modeling Language*). Kartu naudojamas UML, racionalus vieningas procesas (RUP) (*Rational Unified Process* ir įmonės lygio programavimo platformos, programinės įrangos kūrimą iškėlė į aukštesnį architektūrinio vientisumo lygį.

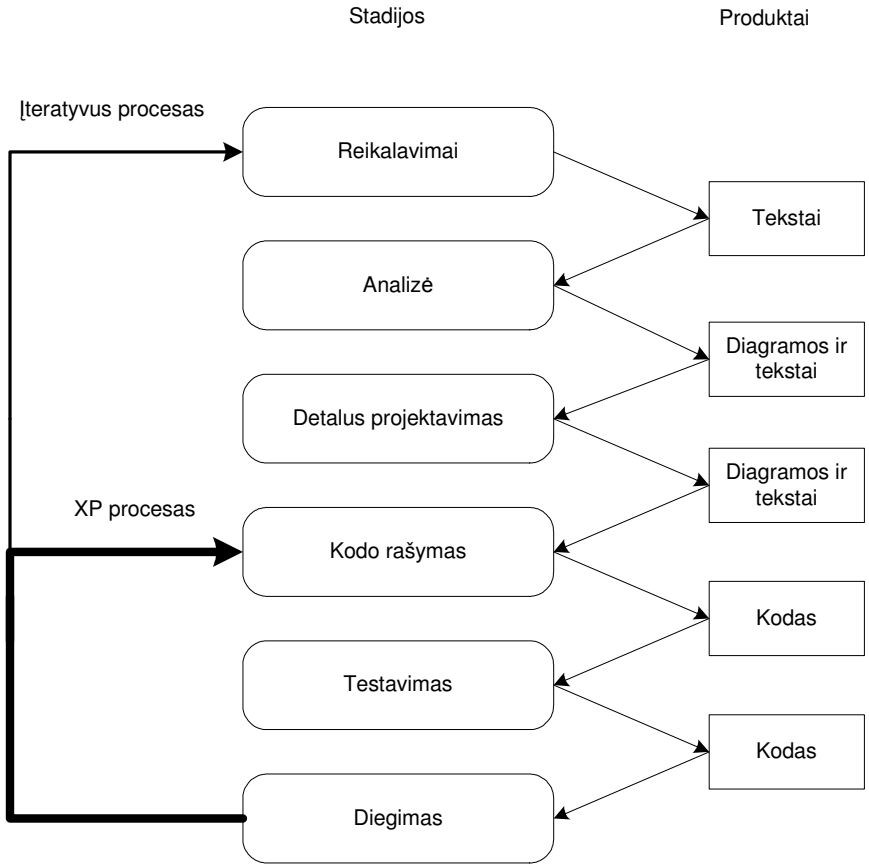
2002 m. OMG pristatė modeliais pagristą architektūrą (MDA) (*Model Driven Architecture*), kaip strateginę savo kryptį. MDA nėra dar viena nauja technologija, tai nauja programinės įrangos kūrimo strategija. Ji akcentuoja modelių svarba programinės įrangos kūrimo procese. Modeliavimo kalbos čia naudojamos kaip programavimo kalbos, o ne kaip projektavimo. MDA architektūros taikymas ateityje turėtų pakeisti dabartinį programinės įrangos procesą. Paanalizuokime svarbiausias programinės įrangos kūrimo proceso problemas.

Pirmoji problema - programinės įrangos kūrimo produktyvumas. Tipiškas šiuolaikinis programinės įrangos procesas [3], paremtas detaliu projektavimu ir kodo rašymu, pereina tokias fazes:

1. Reikalavimu surinkimas
2. Analizė ir funkcinis aprašymas
3. Projektavimas
4. Kodo rašymas
5. Testavimas
6. Diegimas

Naudojant šį procesą dokumentai ir diagramos yra sukuriami 1 – 3 fazės metu. Tai reikalavimų specifikacijos, įvairios UML diagramos ir kiti dokumentai. Šios dokumentacijos rašymas užima nemažai laiko, tačiau pradėjus kodo rašymo fazę, dokumentų vertė labai sumažėja. Dažnai realus išeities kodas nevisiškai atitinka pradines projektavimo diagramas. Tam turi įtakos nenumatytos situacijos analizės ir projektavimo fazių metu, sistemos atnaujinimai, besikeičiantys reikalavimai. Dėl laiko stokos pakeitimai būna padaromi tik išeities kodo lygyje, o diagramos ir dokumentai lieka neatnaujinti.

Šią problemą bando spręsti ekstremalaus programavimo (XP) (*Extreme programming*) metodika. Ji teigia, kad programinės įrangos kūrimo metu realiai produktyvios yra kodo rašymo ir testavimo fazės (žr. 1 pav.). Tačiau, tai išsprendžia tik dalį problemos, nes programinės įrangos palaikymas, atnaujinimas darosi sudėtingas pasikeitus komandos sudėčiai ir paskiriant problemos sprendimą naujam žmogui. Turint tik tūkstančius išeities kodo eilučių ir testus yra ganėtinai sudėtinga įsigilinti ir išsiaiškinti sistemos veikimo problemas. Produktyvumo atžvilgiu reikia padaryti sprendimą ar laikas bus gaištamas rašant dokumentaciją pirmose trijose programinės įrangos kūrimo proceso fazėse, ar palaikymo fazėje, besiaiškinant programinės įrangos realizacija. Vis tik dokumentacija dažnai atspindi programinės įrangos kūrimo projekto brandą.



1 pav. Tradicinės programų kūrimo fazės

Kita problema yra programinės įrangos pernešamumas ir technologijų kaita. Programinės įrangos industrija kiekvienais metais ar net dažniau pristato naujas technologijas padedančias spręsti šias problemas. Tai XML, HTML, SOAP, J2EE, .NET, ASP ir kitos populiarios technologijos. Šios technologijos ne tik duoda apčiuopiamą naudą, bet reikalauja laiko sąnaudų, kol yra įsisavinamos. Prieš keičiant senesnę technologiją naujesne, reikia atlikti išsamų tyrimą ar nauja technologija tikrai atitiks jai keliamus reikalavimus ar į sistemą įdiegus naują technologiją ji bus tarpusavyje suderinta (*Backward-compatible*) su ankstesniu funkcionalumu, nes priešingu atveju nauja technologija gali padaryti daugiau žalos nei jos pritaikymas duos naudos. Taip pat egzistuojančios technologijos yra nuolatos vystomos, todėl reikalingos laiko sąnaudos įsisavinant naują funkcionalumą ar pakeitimus. Skirtingų versijų ta pati technologija dažnai gali būti ir tarpusavyje nesuderinta. Dažniausiai dauguma bibliotekų ir programavimo priemonių tiekėjų palaiko tik tris paskutines versijas, todėl neatnaujinus sistemos galima likti be palaikymo iš tiekėjų pusės.

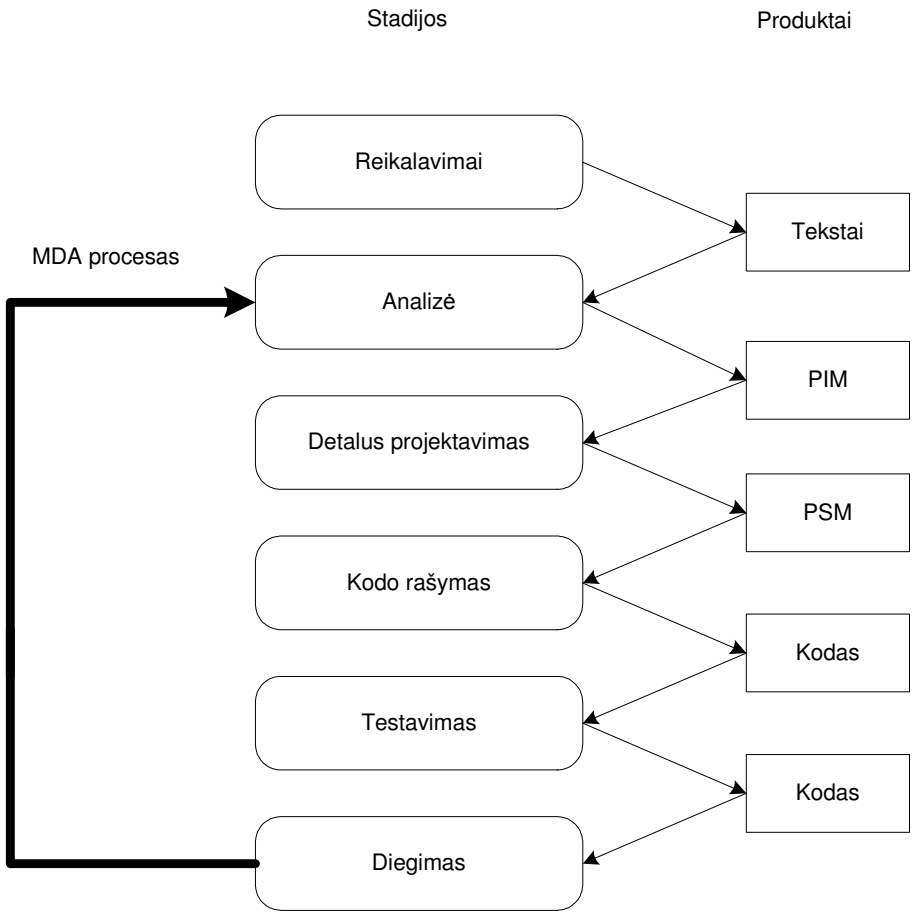
Programinė įranga retai funkcionuoja nepriklausomai. Dažnai ji yra integruota ar turi komunikuoti su kitomis egzistuojančiomis sistemomis, kas iššaukia sąveikavimo ir integravimo problemas (*interoperability problem*). Sena sistema turi veikti su naujai sukurta, kuriant naują sistemą reikia atsižvelgti ar ji sėkmingai veiks su senąja. Todėl projektuojant naują programinę įranga ar darant egzistuojančios atnaujinimus reikia atlikti išsamia analizę prieš pasirenkant vieną ar kitą technologiją. Dažniausiai nauja programinė įranga ar programinės įrangos atnaujinimai naudoja keletą technologijų, bibliotekų ir programavimo aplinkų (*frameworks*), ir naujai sukurtų ir jau seniai naudojamų.

Modeliais pagrista architektūra yra intensyviai vystoma, naujos kartos programinės įrangos kūrimo strategija, siūlanti į modelį orientuotą (*Model-orientied*) arba modeliais paremta (*Model-driven*) programinės įrangos būdą. Vis tik ši architektūra yra palyginti jauna, todėl ji daugiau orientuota į naują programinės įrangos kūrimo metodiką ir griežtai nespecifikuoja aplinkos realizacijos sprendimų variantų. Šio darbo tikslas yra susipažinti su pagrindiniais MDA architektūros elementais ir jų funkcijomis, realizuoti MDA programinės įrangos kūrimo aplinką, bei įvertinti šios aplinkos panaudojimo galimybes keliant tokius tikslus:

- Padidinti programinės įrangos kūrimo produktyvumą.
- Supaprastinti programinės įrangos kūrimą.
- Automatizuoti pasikartojančias užduotis.
- Sumažinti techninių įgūdžių reikalavimus programinės įrangos kūrėjams.
- Tobulinti techninius sprendimus ar keisti sistemą realizuojančias technologijas nedarant įtakos veiklos logikai (*Business logic*).
- Pagerinti lygiagretaus programinės įrangos kūrimo galimybes.

2 MDA PROGRAMAVIMO APLINKOS ANALIZĖ

MDA – tai programinės įrangos kūrimo aplinka pasiūlyta OMG konsorciumo. Kitoks požiūris į programinės įrangos kūrimą padeda spęsti anksčiau paminėtas programinės įrangos kūrimo proceso problemas. MDA programinės įrangos kūrimo fazės yra labai panašios į klasikinio proceso fazes, tačiau šių fazių produktai skiriasi (žr. 2 pav.) [3]. Antros ir trečios fazės produktai yra formalūs programinės sistemos modeliai. Šie modeliai ne tik suprantami žmonėms, bet gali būti apdoroti kompiuteriu. MDA programų kūrimas yra pagrįstas modeliais, nes modeliai valdo sistemos kūrimą, modifikavimą ir diegimą. Nuo platformos nepriklausomas modelis (PIM) (*Platform Independent Model*), tai abstraktus modelis nepriklausomas nuo platformos realizacijos detalių. Kitas MDA modelis yra platformai specifinis modelis (PSM) (*Platform Specific Model*). Šis modelis aprašo sistemą specifiniais terminais priklausančiais nuo to kaip bus realizuota programinė įranga. Naudojant MDA metodologiją, analizės fazės metu sukurtas PIM modelis yra transformuojamas į vieną ar daugiau PSM modelių. Paskutinis žingsnis, naudojant MDA architektūrą, yra PSM transformavimas į išeities kodą. Kadangi PSM modelis yra specifinis tam tikrai technologijai, tai kodo generavimas yra santykinai paprastas. MDA apibrėžia ne tik PIM, PSM ir išeities kodą, bet ir jų tarpusavio ryšius. Sudėtingesnis atvejis MDA programinės įrangos kūrimo procese yra tada, kai vienas PSM turi būti transformuotas į keletą PSM.



2 pav. MDA programų kūrimo fazės

MDA architektūros branduolį sudaro sistemos probleminės srities modeliavimas ir sukurtų modelių transformavimas. Šios dalys detaliau nagrinėjamos 2.1 ir 2.2 skyriuose.

2.1 Modeliai MDA aplinkoje

Modeliai yra pagrindinis MDA architektūros elementas, aprašantis kuriamą sistemą. Kitose šio skyriaus dalyse analizuojama modelio samprata, jam keliami reikalavimai ir realizavimo būdai.

2.1.1 Modelio samprata

Pats MDA architektūros pavadinimas akcentuoja, kad modeliai yra esminis dalykas MDA programavimo aplinkoje. Modeliai yra tiesiogiai susiję su kuriama programinę įranga ir juos galima apibūdinti šiais teiginiais:

- Modelis yra abstrakcija to, kas egzistuoja realybėje.

- Modelis skiriasi nuo to, kas yra projektuojama detalumo lygiu. Į modelį įtraukiamos tik esminės objekto detales, o į nereikšmingas neatsižvelgiama.
- Modelis gali būti naudojamas kaip pavyzdys norit pagaminti tai, kas egzistuoja realybėje.

Reikia pažymėti, kad teiginiuose naudojama sąvoka „kas egzistuoja realybėje“ neapima pačių modelių, nes egzistuoja ir modelių modeliai. Modeliai turi turėti ryšį su tam tikru programinės įrangos kontekstu dar vadinamu sistema (*System*). Dažniausiai sistema reiškia tam tikrą programinės įrangos sistemą, tačiau veiklos modelio (*Business model*) atveju pats veiklos modelis yra sistema.

Modelis aprašo sistema taip, kad juo pasinaudojus panaši sistema galėtų būti realizuota. Tačiau nauja sistema gali skirtis nuo senos sistemos, nes modelis yra abstraktus ir neįtraukia visų sistemos detalių. Nors visos detalės nėra įtrauktos į modelį, tačiau jis apibrėžia esminius sistemos bruožus, todėl naujoje ir senoje sistemoje šie esminiai bruožai išliks. Kuo didesnis sistemos detalumo lygis modelyje, tuo iš modelio realizuotos sistemos bus panašesnės.

Modelis yra aprašomas tam tikra kalba, nesvarbu ar tai UML, ar natūrali, ar programavimo kalba. Norint turėti galimybę automatizuotai transformuoti modelius, reikia modelius apriboti iki MDA modelių, kurie yra rašomi aiškiai apibrėžta (*Well defined*) kalba. Aiškiai apibrėžta kalba turi aiškiai apibrėžta formą, kuri gali būti interpretuojama automatizuotai. Natūralios kalbos nėra aiškiai apibrėžtos, todėl jos netinka automatizuotam transformavimui. Didžioji dauguma modelių yra rašomi naudojant UML, bet pati MDA architektūra tokio apribojimo neturi.

Projektuojama sistema dažnai yra sudaryta iš keleto modelių, kurie skiriasi jų aprašomomis detalėmis. Jei sistema turi keletą modelių, tai natūralu, kad šie modeliai gali turėti tarpusavyje ryšį. Modelių ryšiai yra įvairių tipų. Pavyzdžiui, vienas modelis aprašo vieną sistemos dalį, o kitas modelis aprašo kitą, galbūt šiek tiek persidengiančią, sistemos dalį. Vienas modelis gali aprašyti sistemą detaliau nei kitas ar aprašyti sistemą iš visiškai kitos perspektyvos.

2.1.2 PIM ir PSM modeliai

MDA standartas apibrėžia terminus PIM ir PSM. PIM modelis pasižymi dideliu abstrakcijos lygiu ir yra nepriklausomas nuo realizacijos technologijų. Jis aprašo programinės įrangos sistemą, kuri palaiko tam tikrą veiklą, o sistema yra projektuojama iš tos perspektyvos, kurioje ta veikla yra geriausiai matoma. Sekančiame žingsnyje PIM yra transformuojamas į vieną

ar daugiau PSM modelių. Kiekvienai specifinei technologijai yra generuojamas atskiras PSM modelis, todėl PSM turi specifinės technologijų detales. Pavyzdžiui, jei PSM aprašo reliacinę duomenų bazę tai natūralu, modelis turės tokius elementus, kaip "lentelė", "stulpelis", "pirminis raktas" ir taip toliau. Jei sistema yra paremta EJB (*Enterprise Java Beans*) technologija, tai PSM turės tokius elementus, kaip "esybės komponentas" (*Entity Bean*), "sesijos komponentas" (*Session Bean*) ir taip toliau. Akivaizdu, kad PSM modelis suteiks ne daug informacijos programinės įrangos kūrėjui, jei jis nėra susipažinęs su tam tikra specifine technologija. PIM ir PSM yra naudojami skirtingose MDA proceso fazėse ir apibrėžia sistemą skirtingais abstrakcijos lygiais. Galimybė transformuoti mažo detalumo PIM modelį į PSM modelį leidžia programinės įrangos kūrėjui dirbti su mažesniu informacijos kiekiu, todėl didelės ir sudėtingos sistemos yra realizuojamos daug mažesnėmis pastangomis.

2.1.3 UML panaudojimas PIM aprašymui

PIM modelis turi būti išbaigtas, nuoseklus ir nedviprasmiškas, nes priešingu atveju nebus įmanoma sugeneruoti PSM modelio. Panagrinėkime UML tinkamumą PIM modelio aprašymui. UML yra plačiai naudojamas objektinio modeliavimo standartas. Stiprioji jo pusė yra struktūrinis sistemos projektavimas. Dažniausiai PSM modelio generavimui yra naudojamas kasių modelis, turintis visas reikiamas struktūrines savybes. Tačiau UML turi ir silpnų vietų. Viena iš silpniausių vietų yra elgsenos sritis. UML turi daug skirtingų diagramų elgsenai aprašyti, tačiau jos nėra pakankamai formalios ar išbaigtos, kad būtų lengvai naudojamos PSM modelio generavimui. Pavyzdžiui, yra pakankamai sudėtinga generuoti PSM modelį, naudojant sąveikos diagramą (*Interaction diagram*) ar panaudojimo atvejų diagramą (*Use case diagram*). Elgsenos problemas bando spręsti vykdomasis UML (*Executable UML*). Tai UML kartu su elgsenos savybėmis, paimtomis iš veiksmų semantikos (*Action semantics*) (AS). Vykdomasis UML turi visas stipriąsias grynojo UML struktūrinio projektavimo savybes, o elgsenos problemos sprendžiamos būsenų mašinomis (*Sate machine*). Konkreti sintaksė, naudojama vykdomajame UML, nėra standartizuota.

RUP yra formalizuota programinės įrangos kūrimo metodologija, akcentuojanti reikalavimų surinkimo, analizės ir projektavimo veiklų svarbą programinės įrangos kūrimo procese, prieš pradedant rašyti išeities kodą. Tai sudėtingas, didelis ir struktūrizuotas procesas, dažnai naudojamas didelių projektų vykdymo kontrolei užtikrinti. UML vaidina svarbų vaidmenį RUP procese. Dauguma RUP proceso elementų yra atvaizduojami tam tikra UML modelio

forma. Jei šie RUP modeliai būtų aprašomi MDA stiliumi, tai juos galima būtų naudoti taip pat ir PSM modelio generavimui. RUP proceso naudojimas MDA projekte užtikrintų suprojektuoto modelio atitikimą reikalavimams. Integravus MDA architektūrą į RUP procesą, biurokratinės modelių projektavimo laiko sąnaudos virstų realios programinės įrangos kūrimo dalimi, neprarandant projekto vykdymo kontrolės.

OMG konsorciumas apibrėžė XMI, kaip standartinį modelių ir metamodelių saugojimo būdą [10]. Kadangi MDA aplinkos realizavimui pasirinkta Java programavimo kalba, tai modelių nuskaitymui ir išsaugojimui į XMI formato bylas galima pasirinkti vieną iš dviejų, šiuo metu populiariausių programavimų aplinkų: NSUML (http://nsuml.sourceforge.net/) ir EMF (http://www.eclipse.org/emf). MDA aplinkos realizacijos pradžioje EMF aplinka dar nebuvo išleista, todėl darbui su modeliai buvo pasirinkta NSUML aplinka.

Kadangi realizuojama MDA aplinka dirbs su standartizuotu XMI formatu, tai PIM modeliams projektuoti galima naudoti trečios šalies UML projektavimo įrankius. Buvo palyginti populiariausi UML projektavimo įrankiai: MagicDraw UML ir RationalRose. Atsižvelgiant į išvardintus RationalRose trūkumus, PIM modeliams projektuoti buvo pasirinktas MagicDraw UML projektavimo įrankis:

- Palaiko tik UML 1.3 standartą.
- UML elementas gali turėti tik vieną stereotipą.
- Nepalaiko modelių, išskyrus pagrindinį modelį.
- XMI lygyje nepalaiko daugelio failų. Jie gali būti surišti, naudojant tik specifinį RationalRose formatą.

2.1.4 UML metamodeliai

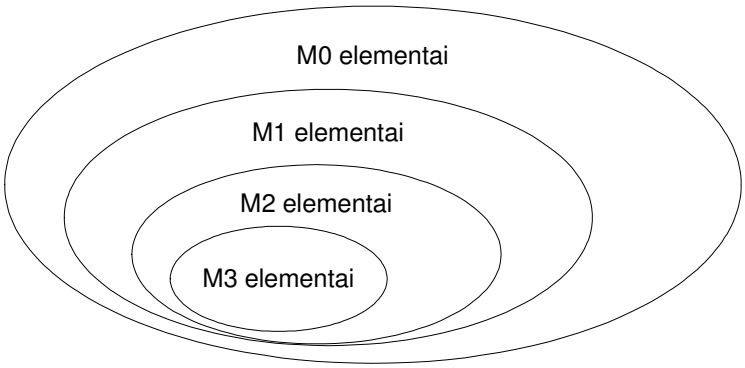
Ankstesniuose skyriuose paminėta, kad modelis aprašo sistemą aiškiai apibrėžta kalba, o aiškiai apibrėžta kalba tinka automatizuotam interpretavimui. Tačiau iškyla problema, kaip apibrėžti aiškiai apibrėžtą kalbą. Modelyje aprašomos tam tikros elementų klasės, kurių egzemplioriai egzistuos sistemoje. Kalba aprašo elementus, kurie egzistuos modelyje (pvz. “klasė”, “būsena”, “katalogas” ir t.t.). Atsižvelgiant į šiuos panašumus, kalba gali būti apibrėžta modeliu, kuris apibrėžia kalboje naudojamus elementus. Kiekvienas elementas, naudojamas projektuojant modelį, yra apibrėžtas kalbos metamodeliu. UML naudojamos klasės, atributai, asociacijos, būsenos ir taip toliau yra aprašyti UML metamodelyje. Pavyzdžiui, jei sąsajos (*Interface*) metaklasė nebūtų įtraukta į UML metamodelį, tai šis elementas negalėtų būti

naudojamas UML modelyje. Metamodelis pilnai aprašo kalba, todėl nenaudinga įvesti skirtumo tarp kalbos ir metamodelio, kuris aprašo kalbą ir iš praktinių sumentimų jie yra traktuojami ekvivalentiškai. Kadangi metamodelis yra taip pat modelis, tai jis turi būti aprašytas aiškiai apibrėžta kalba. Ši kalba vadinama metakalba. Savo ruožtu, metakalba yra taip pat kalba, kuri gali būti apibrėžta metamodeliu, aprašytu dar kita metakalba. Tokiems formaliems metaduomenų ir modeliavimo kalbų aprašymams kurti OMG konsorciumas sukūrė MOF (Meta Object Facility) kalbą [7]. Formalių modeliavimo kalbų aprašymų specifikavimas, vadinamas metamodeliavimu. MOF kalba ypatinga tuo, kad ji yra save aprašanti, tai yra MOF kalba galima aprašyti ne tik kitų kalbų metamodelius, bet ir jos pačios modelį. MOF gali būti naudojama ne tik naujų modeliavimo kalbų specifikavimui, bet ir, kaip universalus metaduomenų aprašymo ir saugojimo standartas [1]. Teoriškai galėtų būti begalinis skaičius modelio-kalbos-metakalbos sluoksnių, tačiau OMG standartas apibrėžia keturis sluoksnius: M0, M1, M2 ir M3 (žr. 1 lent.).

1 lent. Keturi metaduomenų architektūros sluoksniai

Sluoksnis	**Aprašymas**
M0	Modelių panaudojimas. Šito lygmens elementai yra M1 lygmens elementų egzemplioriai. Jei M1 lygmenyje apibrėžiama klasė `Customer`, tai M0 lygmenyje klasės egzemplioriai bus `Customer` tipo objektai.
M1	Modeliai, kurių elementai M2 lygmens modelio egzemplioriai. Tai UML modelių lygmuo.
M2	Metamodeliai, kurių elementai M3 lygmens modelio elementų egzemplioriai (*instances*) ir M2 lygmens metaklasės. Šitas lygmuo atitinka UML kalbos gramatiką – UML metamodelį.
M3	Metametamodeliai, kurių elementai susieti statiniais ryšiais. MOF modelio lygmuo.

Keturių metaduomenų architektūros sluoksnių poaibių ryšiai pavaizduoti 3 pav.



3 pav. M0, M1, M2 ir M3 sluoksnių poaibių ryšiai

2.1.5 UML profilių vaidmuo MDA architektūroje

Profilis (*Profile*) yra UML dalis ir apibrėžia tam tikrą UML panaudojimo būdą. Pavyzdžiui, JAVA profilis apibrėžia, kaip UML modeliuose projektuoti JAVA išeities kodą. Profilis yra sudarytas iš aibės stereotipų (*Stereotype*), aibės apribojimų (*Constraint*) ir aibės žymėtų reikšmių (*Tagged value*). Stereotipas apibrėžiamas vardu ir yra pridedamas prie UML metamodelio elementų. Apribojimai pridedami prie stereotipų apibrėžimo. Jie išreiškiami, naudojant objektų apribojimo kalbą (*Object constraint language*) (OCL), ir aprašo modelio elementų egzempliorių, kuriems taikomas stereotipas, apribojimus. Žymėta reikšmė yra papildomas metaatributas, kuris UML metamodelyje pridedamas prie UML metaklasės. Ji turi vardą, tipą ir tam tikrą reikšmę iš modelio. Profilis apibrėžia specializuotą metamodelį, kuris yra UML metamodelio poaibis. Faktiškai profilis apibrėžia naują kalbą pakartotinai panaudojęs UML metamodelį. Dažniausiai dauguma profilių yra naudojami apibrėžti kalbą, taikomą tam tikrai platformai, pavyzdžiui Java ar EJB profilis. Šios profilio savybės puikiai tinka aprašyti PIM ir PSM modelius.

2.1.6 Modelių susiejimo strategijos

Norint įvykdyti išeities modelio transformaciją į tikslo modelį, reikia aprašyti tų modelių elementų loginius sąryšius. Toks modelių elementų sąryšių specifikavimas vadinamas modelių susiejimu (*model mapping*). OMG mini žemiau išvardintas [4] išeities ir tikslo modelių susiejimo strategijas:

1. Modelių tipų (metamodelių) susiejimas – tai toks modelių susiejimas, kai susiejami PIM modeliavimo kalbos aprašomi tipai su PSM modeliavimo kalbos aprašomais tipais. Šiam

modelių susiejimui yra svarbus metamodeliavimas, kurio vaidmuo buvo nagrinėtas [2.1.4] skyriuje, nes susiejami vienos modeliavimo kalbos metamodelio elementai su kitos modeliavimo kalbos metamodelio elementais.

2. Modelių egzempliorių susiejimas – tai susiejimas modelių egzempliorių lygyje, kai apibrėžiama kokias sąlygas atitinkantis vieno modelio elementas susiejamas su kitu antrojo modelio elementu.

Modelių tipų (metamodelių) susiejimas yra griežtai formalus, nes susiejami elementai nurodomi metamodelio lygyje. Toks griežtai formalus susiejimas neleidžia modeliavimo kalbos vartotojui sukurti nekorektiškus modelius.

Modeliavimo kalbos požiūriu, modelių egzempliorių susiejimas nėra formalus ir griežtas, nes susiejami ne metamodelio elementai, kurie yra modeliavimo kalbos formalizmo dalis, o jų egzemplioriai. Pavyzdžiui, UML kalba leidžia klasėm priskirti stereotipus, tačiau šitos kalbos formalizmas nenustato, kokios klasės gali turėti stereotipą, o kokios ne.

MDA aplinkos realizacijoje buvo pasirinkta modelių egzempliorių susiejimo strategija, nes ji suteikia daugiau lankstumo realizuojant transformacija:

- Lanksti navigacija tarp aibės susietų modelio elementų
- Darant pakeitimus transformacijos realizacijoje pakanka susieti naujus modelio elementus, išvengiant pakeitimų metamodelio lygyje

2.2 Transformacijos MDA aplinkoje

Transformacijos yra dar vienas svarbus MDA architektūros elementas. Kitose šio skyriaus dalyse apžvelgiamos transformacijos, jų specifikavimas ir realizavimo būdai.

2.2.1 Transformacijų automatizavimas

MDA procesas gali pasirodyti panašus į tradicinį programinės įrangos kūrimą, tačiau yra vienas esminis skirtumas. Tradiciškai transformacijos iš modelio į modelį, ar iš modelio į išeities kodą yra atliekamos rankomis. Yra daug įrankių, kurie gali generuoti kodą iš modelio, bet dažniausiai yra apsiribojama šabloninio kodo generavimu, kuris vis tiek turi būti papildytas rankomis. MDA transformacijos yra vykdomos automatizuotai (žr. 4 pav.), įskaitant ne tik transformaciją iš PSM į kodą, bet ir PIM transformaciją į PSM. MDA architektūra yra ganėtinai jauna, todėl dabartiniai transformavimo įrankiai nėra prakankamai sudėtingi, kad galėtų atlikti

šimtaprocentines transformacijas. Programinės įrangos kūrėjas turi rankomis papildyti transformuotą PSM ar kodo modelį. Vis tik dabartinės priemonės, naudodamos PIM, pajėgios sugeneruoti veikiančią sistemą, turinčią pagrindinį funkcionalumą. Keisdamas PIM modelį, programinės įrangos kūrėjas gali operatyviai reaguoti į veikiančios sistemos problemas, nes veikiančią sistemą galima nedelsiant sugeneruoti iš naujo.

4 pav. Trys pagrindiniai MDA proceso žingsniai

2.2.2 Transformacijos specifikavimas

Transformacijos realizacijoje galime išskirti dvi pagrindines dalis: transformavimo įrankį ir transformacijos specifikaciją (*transformation definition*) (žr. 5 pav.). Priklausomai nuo pasirinktos realizacijos architektūros, šios dalys gali būti atskiros arba apjungtos į vieną įrankį, su įprogramuotu transformacijos šablonu (*transformation pattern*). Transformavimo įrankis ima PIM ir transformuoja jį į PSM. Kitas ar tas pats transformavimo įrankis transformuoja PSM į išeities kodą. Šios transformacijos yra MDA programinės įrangos kūrimo proceso pagrindas. Kiekvienai transformacijai ir visiems įėjimo modeliams, transformavimo įrankis naudoja tą pačią transformacijos specifikacija.

5 pav. Transformacijos specifikacija transformavimo įrankyje

Transformacijos specifikacija sudaryta iš transformacijos taisyklių rinkinio, kuris nedviprasmiškai aprašo būdą, kaip vienas modelis ar jo dalis turi būti panaudotas kito modelio ar jo dalies sukūrimui. [3] pateikia šiuos transformacijos, transformacijos specifikacijos ir transformacijos taisyklių apibrėžimus:

- Transformacija yra automatizuotas tikslo (*Target*) modelio generavimas iš išeities (*Source*) modelio, naudojant transformacijos specifikaciją.

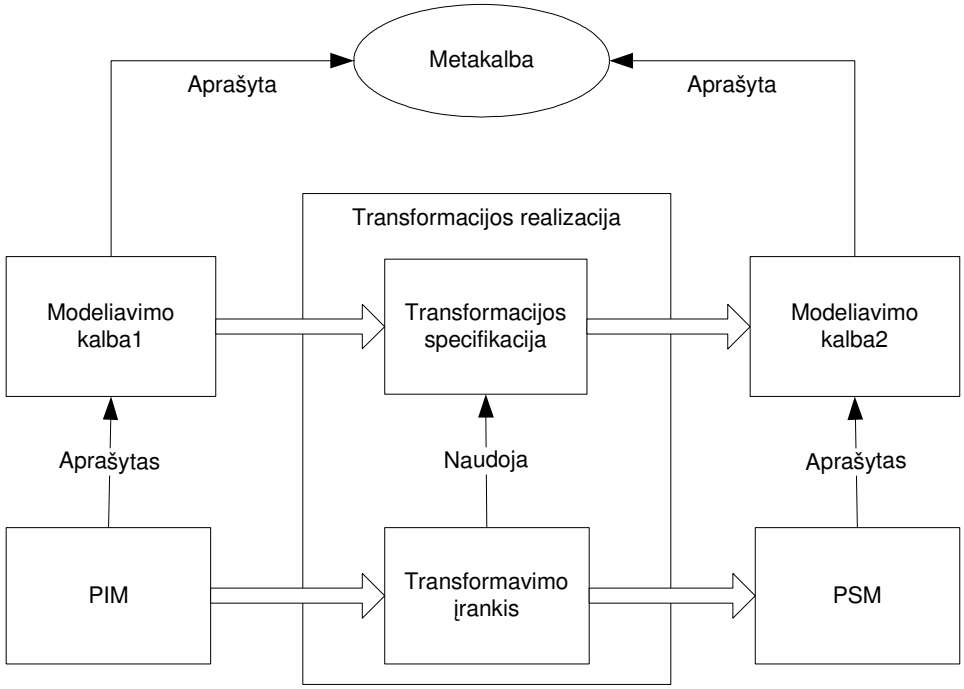
- Transformacijos specifikacija yra transformacijos taisyklių rinkinys, kuris aprašo kaip modelis, apibrėžtas išeities kalba, turi būti transformuotas į modelį, apibrėžtą tikslo kalba.
- Transformacijos taisyklė yra apibrėžimas, kaip viena ar daugiau išeities kalbos išraiškų turi būti transformuota į vieną ar daugiau tikslo kalbos išraiškų.

2.2.3 Metamodelio reikšmė transformacijose

Literatūroje [3] minimos dvi priežastys dėl kurių metamodeliavimas labai svarbus modelių transformavimui ir MDA:

1. Metamodeliavimas, tai modeliavimo kalbų specifikavimo priemonė MDA architektūroje. Metakalbos pagrindu, apibrėžę modeliavimo kalbos metamodelį, galime specifikuoti PIM ar PSM kalbą.
2. Transformacijų taisyklės (*rules*) aprašo kokie pradinio modelio elementai turi būti transformuoti į atitinkamus tikslo modelio elementus.

6 pav. demonstruoja, kaip metamodeliavimas siejamas su PIM – PSM modelių transformacijomis.



6 pav. Metamodeliavimo ir modelių transformacijų ryšis

Pradiniame transformacijų realizavimo etape reikia specifikuoti formalią PIM modeliavimo kalbą. Tai pasiekiama aprašant PIM metamodelį. PIM metamodelio pagrindu sukuriamas PIM modelio egzempliorius, kuris atitinka modeliuojamą programinę sistemą. Norint gauti PSM modelį, specifinį tam tikrai programavimo platformai, taip pat reikia aprašyti PSM metamodelį. Transformavimo įrankis, naudodamas transformacijos specifikaciją, transformuos mūsų PIM į PSM, todėl transformacijų realizacija yra neatskiriama nuo metamodeliavimo. Vadinasi, prieš realizuojant transformaciją, reikia apibrėžti išeities PIM ir tikslo PSM modeliavimo kalbas.

2.2.4 Transformacijų realizavimo būdai

Priklausomai nuo transformacijos strategijos aprašymo metodo, transformacijos gali būti realizuotos keliais būdais. Pirmasis yra algoritminis transformacijų realizavimas. Tai tokia modelių susiejimo ir jų transformacijų realizacija, kuri naudoja jau egzistuojančias algoritmines programavimo ar transformavimo kalbas. Galima išskirti šiuos kalbų tipus, naudojamus transformacijos realizavimui:

- Algoritminės kalbos. Transformacijos realizacija fiksuotai įprogramuojama į transformavimo įrankį arba transformuojanti dalis užkraunama dinamiškai (*plug-in*).
- Interpretuojamos kalbos . Jos naudojamos pačių transformavimo taisyklių aprašymui.
- Šablonų kalbos (pvz. XSL, Velocity [14]).

Kitas transformacijos realizavimo būdas yra metamodeliavimas. Tai modelių susiejimo ir transformacijų specifikavimo būdas, kuris pagrįstas metamodeliu ir jų programavimo sąsajų (*API*) panaudojimu. OMG konsorciumas paskelbė užklausą pasiūlymams RFP (*Request for Proposal*) dėl MOF 2.0 užklausų, atvaizdų ir transformacijų (*MOF-QVT – MOF Model / Views /Transformations*) [8] ir tikisi sukurti naują standartizuotą transformacijų modeliavimo kalbą. Transformacijų modeliavimo kalba leistų metamodelių lygyje, naudojant modeliavimo priemones, susieti modelius ir specifikuoti transformaciją, iš vieno modelio į kitą. OMG jau gavo atsakymų į MOF-QVT RFP, tačiau šita MOF metakalbos dalis iki šiol nėra standartizuota.

MDA aplinkos realizacijoje PIM į PSM modelių transformacijos specifikacijai aprašyti pasirinkta algoritminės kalbos strategija. Šios transformacijos metu yra atliekama daug iteracijų ir sukuriama daug naujų objektų., todėl transformacijos vykdymo greitis ir naudojamos operatyviosios atminties sąnaudos yra ypatingai svarbūs. Interpretuojamos ar šablonų kalbos

atveju, papildomai yra naudojami resursai pačios transformacijos specifikacijos interpretacijai. Pavyzdžiui, eksperimentinė didelio PIM modelio į PSM modelį transformacija, realizuota XSL kalba, užtrukdavo beveik 3 kartus ilgiau ir naudojo beveik 2 kartus daugiau operatyvios atminties nei, transformacija, realizuota algoritmine Java kalba. Kadangi PSM modelis turi visas specifines, sistemą realizuojančios technologijos detales, ir gali būti tiesiogiai transformuotas į išeities kodą, tai šiai transformacijai atlikti buvo pasirinktas Velocity šablonų variklis. Tai galinga ir paprasta naudoti šablonų kalba, puikiai tinkanti aprašyti ir esant būtinybei nesunkiai koreguoti, generuojamo kodo transformacijos specifikaciją.

2.3 MDA aplinkos apibendrinimas

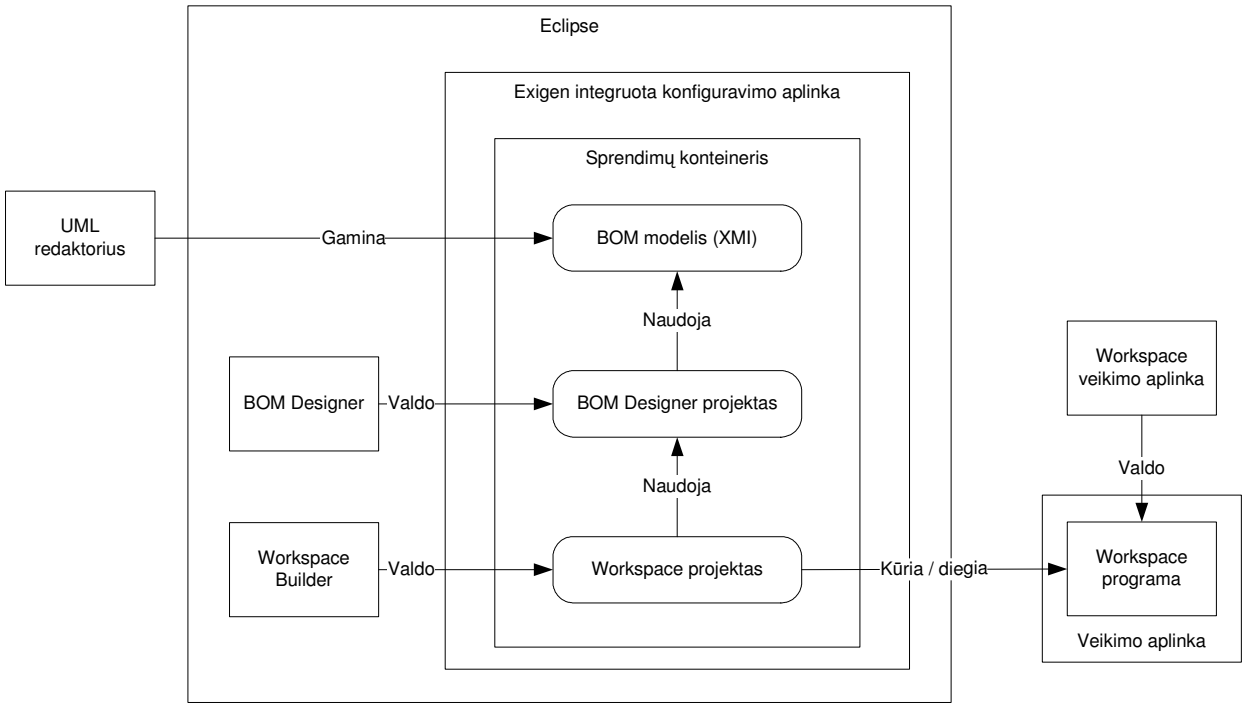
Ankstesniuose skyriuose yra aprašyti pagrindiniai MDA aplinkos elementai: modeliai, PIM, PSM, modeliavimo kalbos, transformacijos, transformacijų specifikacijos ir transformavimo įrankiai. Žemiau yra apibendrinti MDA aplinkos elementų terminai ir juo vaidmuo:

- *Modelis* yra sistemos aprašymas.
  - *PIM* yra nuo platformos nepriklausomas modelis, kuris sistema aprašo abstrakčiai, be specifinių jos realizacijos detalių.
  - *PSM* yra platformai specifinis modelis, kuris sistema aprašo specifiniais terminais, apibūdinančias, kaip sistema bus realizuota.
- Modeliai yra rašomi aiškiai apibrėžta *kalba*.
- *Transformacijos specifikacija* aprašo, kaip modelis aprašytas išeities kalba gali būti transformuotas į modelį aprašyta tikslo kalba.
- *Transformavimo įrankis* vykdo specifinių modelių transformavimą, atsižvelgdamas į transformacijos specifikaciją.

Realizuojant MDA aplinką, pagrindinis dėmesys bus skiriamas MDA architektūros metodikai, MDA programinės įrangos kūrimo procesui ir išnagrinėtomis esminių MDA elementų funkcijomis.

3 BOM DESIGNER REALIZACIJA

BOM (*Business Object Model*) Designer yra MDA aplinkos realizacija, skirta veiklos objektų modelio projektavimui ir veiklos logikos realizavimui. Jis suteikia galimybę naudoti duomenų (*Data*) ir veiklos (*Business*) objektus kitiems Exigen integruotos konfigūravimo aplinkos įrankiams, pavyzdžiui grafinės aplinkos projektavimo įrankiui *Workspace Builder* (žr. 7 pav.). BOM modelis yra saugomas XMI failuose, o Java klasės generuojamos iš UML klasių diagramų, aprašančių duomenų ir veiklos objektus. BOM modeliui projektuoti gali būti naudojami išoriniai UML redaktoriai arba vidinis BOM modelio redaktorius. BOM Designer yra realizuotas kaip Eclipse programavimo aplinkos papildomas komponentas (*Plug-in*).

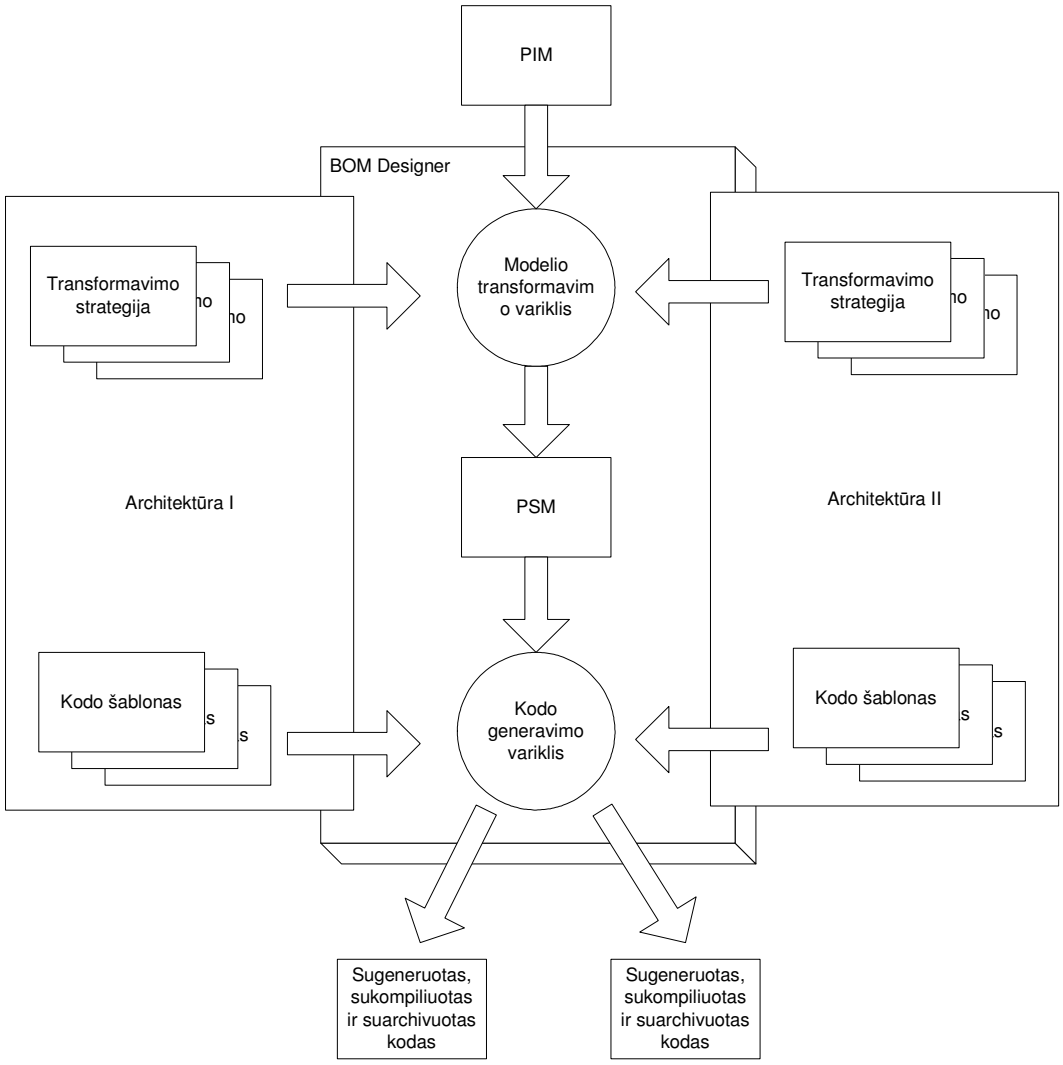


7 pav. BOM Designer vieta Exigen integruotoje konfigūravimo aplinkoje

3.1 BOM Designer architektūra

BOM Designer naudoja skirtingas transformacijų realizacijas PIM modeliui transformuoti į PSM modelį ir PSM modelio transformavimui į išeities kodą. PIM į PSM transformacijos realizavimui yra naudojama algoritminės kalbos strategija. Transformacijos specifikacija realizuota Java programavimo kalba. PSM į išeities kodą transformacijos realizacijai naudojama

šablonų kalbos strategija, o transformacijos specifikacija aprašyta Velocity šablonais. PMS modelis egzistuoja tik transformacijos vykdymo metu ir jai pasibaigus yra pašalinamas.



8 pav. BOM Designer architektūra

BOM designer architektūros elementų (žr. 8 pav.) ir jų vaidmens aprašymas:

- Transformavimo strategija yra transformacijos specifikacija, realizuota Java kalba.
- Kodo šablonas yra transformacijos specifikacija, aprašyta panaudojant Velocity šablonus.
- Architektūra I / II yra baigtinis transformavimo specifikacijų rinkinys (aibė transformavimo strategijų ir aibė kodo šablonų) tam tikrai sistemos realizacijai. BOM designer palaiko JDO (*Java Data Objects*), Hibernate, EJB ir žimatinklio paslaugų (*Web-Services*) sistemų realizavimo technologijas.

- Modelio transformavimo variklis, tai transformavimo įrankis, kuris naudodamas transformavimo strategijų aibę, transformuoja PIM modelį į PSM modelį.
- Kodo generavimo variklis, tai transformavimo įrankis, kuris naudodamas kodo šablonų aibę, transformuoja PSM modelį i išeities kodą.

3.2 BOM modelio profilis

BOM profilis yra skirtas PIM modelio aprašymui aiškiai apibrėžta kalba. Tai UML profilis, kuris apibrėžia, kaip veiklos ir duomenų objektai turi būti aprašyti naudojant UML klasių diagramas. BOM profilis palaiko UML modelius atitinkančius UML 1.4 ir XMI 1.1 specifikacijas. 2 lent. aprašyti pagrindiniai BOM profilio stereotipai.

2 lent. BOM profilio stereotipai

Stereotipas	**MOF elementas**	**Aprašymas**	**Apribojimai**
<<BOM>>	Model	BOM modelis yra konteineris, kuriame talpinami veiklos ir duomenų objektai	Veiklos duomenų objektai turi būti talpinami šiuo stereotipu pažymėtuose modeliuose
<<dataObject>>	Class	Tai paprastas duomenų objektas, naudojamas duomenims laikyti. Gali turėti įvairių pritaikymų, bet dažniausiai gyvuoja vienos iteracijos ribose.	
<<firstClass>>	Class	Tai duomenų objektas, turintis identifikaciją, ir kuris gali būti išsaugotas	Klasė, turinti <firstClass>> stereotipą, turi turėti atributą su <<primaryKey>> stereotipu
<<businessObject>>	Class	Tai veiklos objektas, kuris valdo sąveiką su duomenų objektais. Veiklos objektai nesaugo būsenos (*Stateless*)	
<<primaryKey>>	Attribute	Atributas su šiuo stereotipu duomenų objektuose naudojamas, kaip identifikatorius	Klasė, turinti atributą su stereotipu <<primaryKey>>, turi turėti stereotipą <<firstClass>>
<<calculated>>	Attribute	Šio atributo reikšmė apskaičiuojama programos veikimo metu, panaudojus kitus duomenų objekto atributus	
<<readOnly>>	Attribute	Šis atributas negali būti pakeistas programos veikimo metu	

3.3 Transformacijos BOM Designer aplinkoje

Išeities kodas BOM Designer aplinkoje generuojamas dviem žingsniais – sąsajų fazė ir realizacijų fazė. Sąsajų fazės metu generuojamos duomenų objektų sąsajos, abstrakčios klasės objektų kūrimui ir darbui su duomenų saugyklom (*Persitence storage*), veiklos objektų ir apskaičiuotų (*Calculated*) klasių realizacijos skeletai. To pilnai pakanka veiklos logikos rašymui, o programinės įrangos kūrėjui visiškai nebūtina žinoti kokios technologijos bus naudojamos sistemos realizacijoje. BOM modelis turi dviejų pagrindinių klasės tipų elementus. Tai duomenų objektai ir veiklos objektai. Duomenų objektai yra konteineriai duomenims laikyti ir savyje neturi jokios veiklos logikos. Veiklos objektai yra konteineriai veiklos logikai saugoti. 3 lent. išvardintos klasės sugeneruojamos iš BOM modelyje esančio duomenų objekto.

3 lent. Klasės sugeneruojamos iš BOM modelyje esančio duomenų objekto

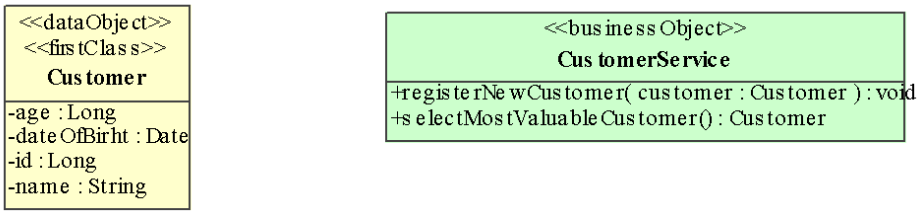
Sugeneruota klasė	Aprašymas
DO interface	Duomenų objekto sąsaja turinti getter/setter metodus (kolekcijų atributai turi tik getter metodus). Programinės įrangos kūrėjui pakanka šios sąsajos dirbant su duomenų objektu.
DO abstract factory	Abstrakti klasė naujam duomenų objekto egzemplioriui sukurti. Ji reikalinga, nes konkreti duomenų objekto realizacija bus sugeneruota tik realizacijų fazėje.
DO helper interface	Sąsaja darbui su duomenų saugykla. Turi metodus duomenų objekto paieškai, išsaugojimui ir pašalinimui.
DO helper abstract factory	Abstrakti klasė, naudojama sąsajos darbui su duomenų saugykla gavimui.
Calculated class	Generuojama, jei duomenų objektas turi apskaičiuojamų atributų. Sukuriami metodų skeletai šiems atributams

4 lent. išvardintos klasės sugeneruojamos iš BOM modelyje esančio veiklos objekto.

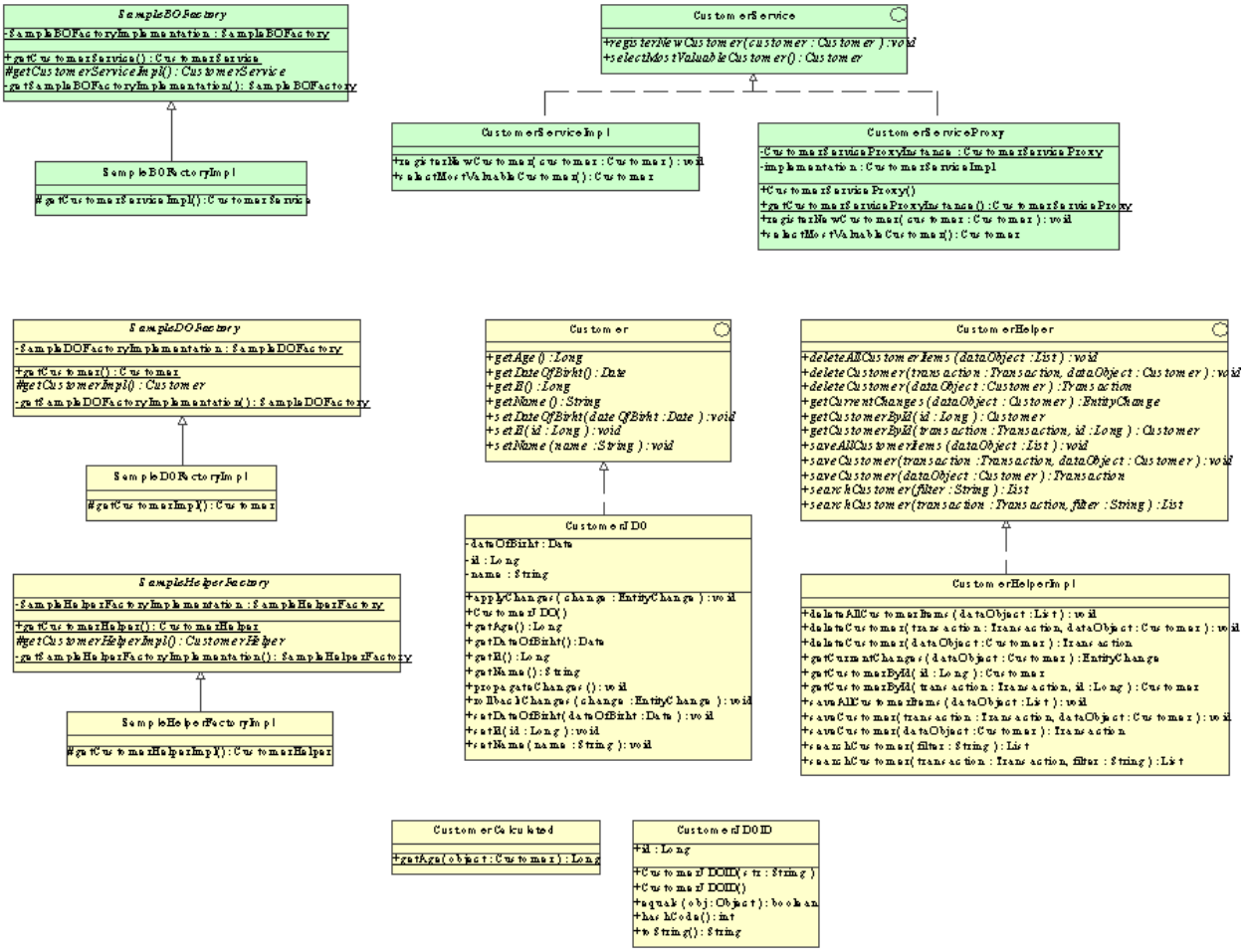
4 lent. Klasės sugeneruojamos iš BOM modelyje esančio veiklos objekto

Sugeneruota klasė	Aprašymas
BO interface	Veiklos objekto sąsaja turinti modelyje apibrėžtus veiklos objekto metodus.
BO abstract factory	Abstrakti klasė naudojama veiklos objekto sąsajai gauti. Reikalinga, nes veiklos objektas gali būti realizuotas, naudojant įvairias technologijas (pvz. paprasta Java klase, EJB ar Web-Services)
BO implementation class	Veiklos objekto realizacijos klasė, kurioje rašoma veiklos logika. Turi veiklos objekto metodų skeletus.

10 pav. vaizduojamos JDO sistemos realizacijai sugeneruotos klasės, naudojant PIM modelyje aprašytus (žr. 9 pav.) duomenų ir veiklos objektus.



9 pav. PIM modelio pavyzdys

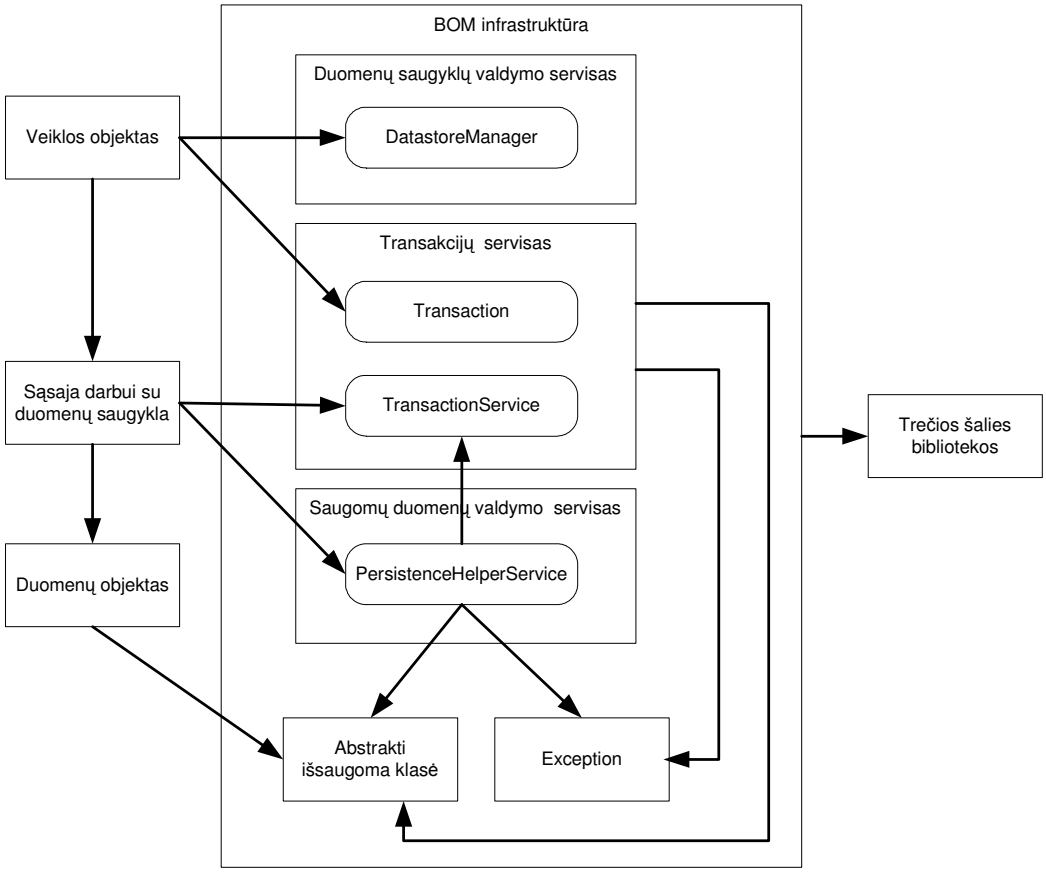


10 pav. PSM modelio, sugeneruoto iš PIM modelio, pavyzdys

Veiklos logikos aprašymo pavyzdį galima rasti priede [7.3].

3.4 BOM infrastruktūra

BOM infrastruktūra, tai aibė klasių ir bibliotekų, kurios naudojamos pagrindiniam sistemos funkcionalumui realizuoti. Ji suteikia galimybę abstrakčiai naudotis esminiu technologijų funkcionalumu, tokiu, kaip transakcijos ar duomenų išsaugojimas, paslepiant tikrąjį funkcionalumą, realizuojančią technologiją (pvz. JDO ar Hibernate). Generuojama išeities kodo dalis nuo infrastruktūrinės dalies atskiriama vadovaujantis paprastu principu – kintanti technologijos dalis arba specifinė modelio elemento dalis yra generuojama, o pastovioji technologijos dalis arba nepriklausoma nuo modelio elemento yra realizuojama infrastruktūroje. BOM Designer sugeneruotas kodas programos vykdymo metu naudoja BOM infrastruktūrą (žr. 11 pav.).



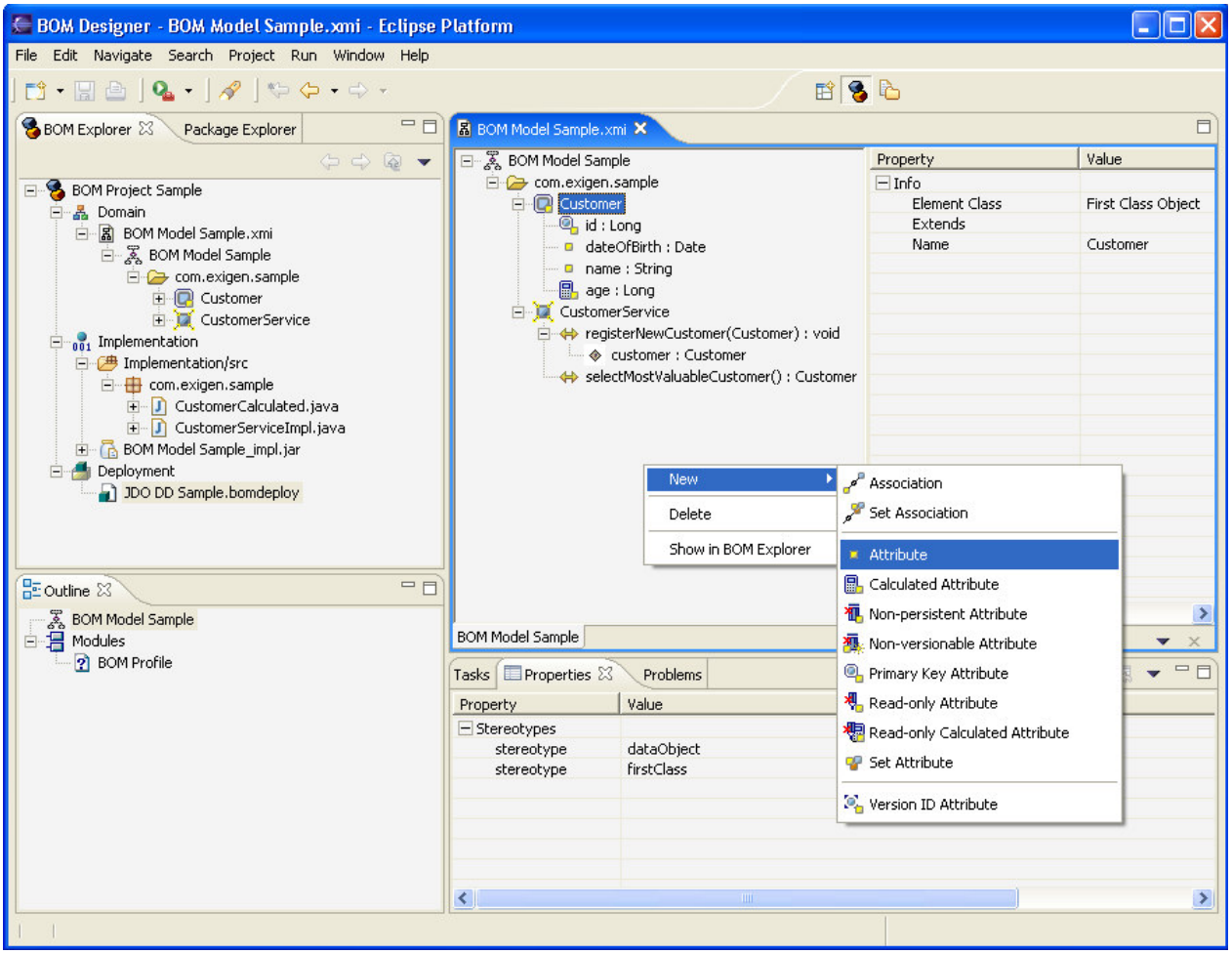
11 pav. BOM aplinkos infrastruktūra

3.5 BOM Designer grafinės aplinkos apžvalga

Kitose šio skyriaus dalyse apžvelgiami svarbiausi BOM Designer aplinkos grafiniai įrankiai ir integracija su grafinės aplinkos redaktoriumi.

3.5.1 BOM modelio redaktorius

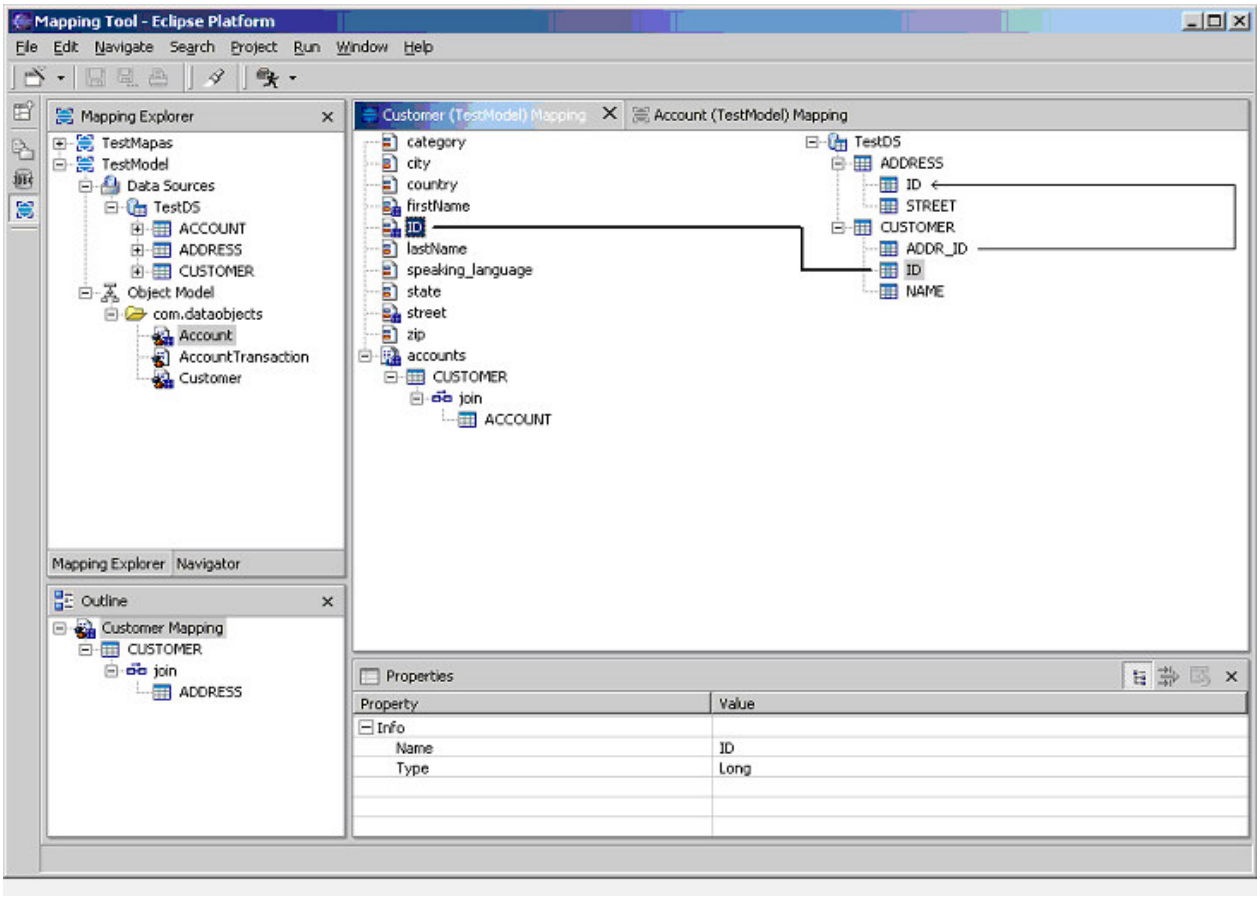
BOM aplinkos modeliai gali būti projektuojami su išoriniais UML redaktoriai. Tačiau, geri UML redaktoriai yra mokami, be to realizuojant sistemą reikėtų persijunginėti tarp skirtingų aplinkų, net ir atliekant minimalius modelio pakeitimus. Atsižvelgiant į šias problemas, buvo realizuotas BOM modelio redaktorius [žr. 12 pav.], kuris pilnai atlieka pagrindines modelio redagavimo funkcijas.



12 pav. BOM aplinkos modelio redaktorius

3.5.2 BOM objekto į duomenų bazę projekcijos įrankis

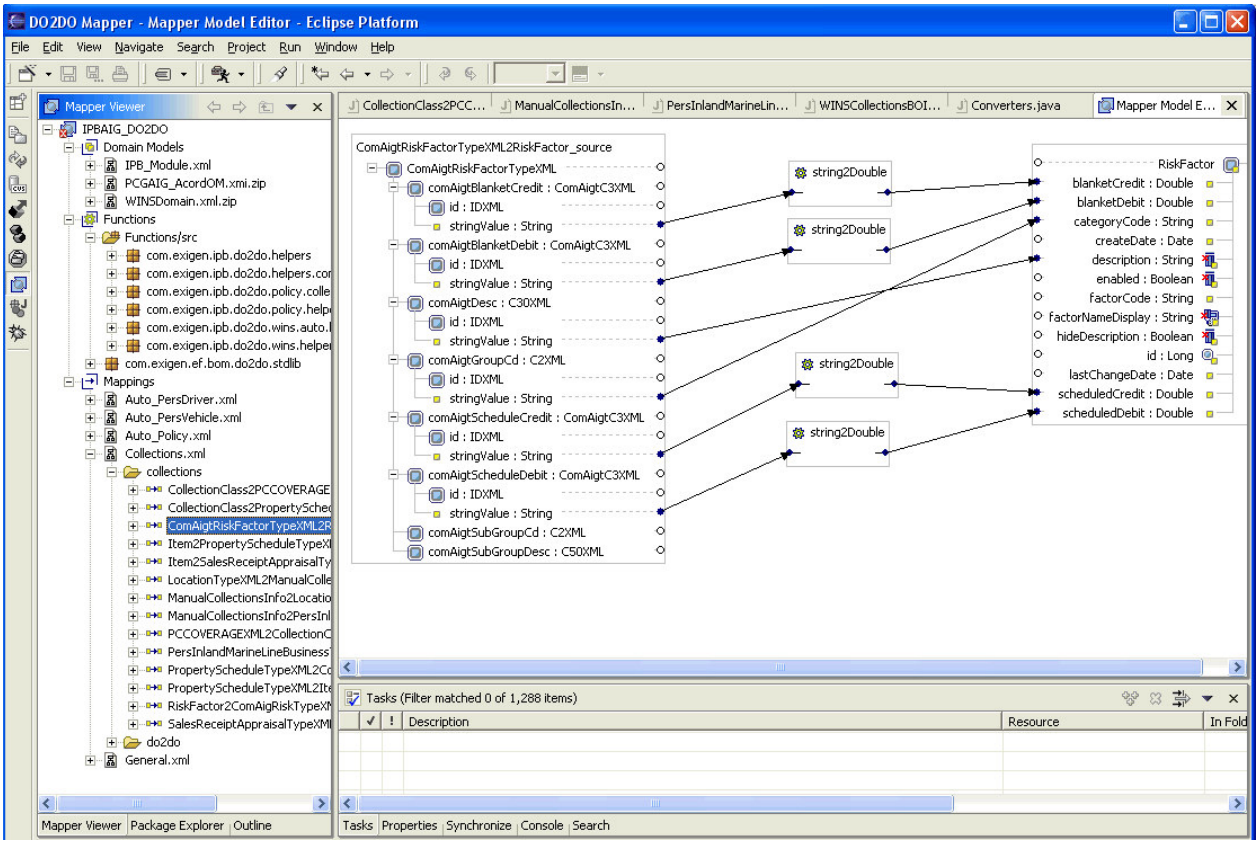
Dažnai realizuojant sudėtingas sistemas ar atnaujinant egzistuojančias, duomenų bazė jau egzistuoja arba ji būna atskirai projektuojama programos vykdymo greičio sumetimais. Todėl natūraliai iškilo poreikis atlikti objektų projekciją į egzistuojančią duomenų bazę, nes standartinė projekcija dažnai nesutampa su egzistuojančia duomenų bazės schema. Skiriasi lentelių vardai, stulpelių vardai ar net normalizacijos lygis. Atsižvelgiant į šias problemas, buvo realizuotas BOM objekto į duomenų bazę projekcijos įrankis [žr. 13 pav.]. Objekto projekcija į duomenų bazę yra saugoma UML modelyje, kuris apibrėžtas jam skirtu profiliu. Realizacijos generavimo fazėje, transformuojant PSM į duomenų bazės metaduomenis (pvz. JDO ar Hibernate metaduomenis) atsižvelgiama ne tik į PSM, bet ir į objektų projekcijos į duomenų bazę modelį. Priede [7.1] yra sugeneruotas JDO metaduomenų pavyzdys, o [7.2] yra sugeneruotas Hibernate metaduomenų pavyzdys.



13 pav. BOM aplinkos objekto į duomenų bazę projekcijos įrankis

3.5.3 BOM objekto į objektą projekcijos įrankis

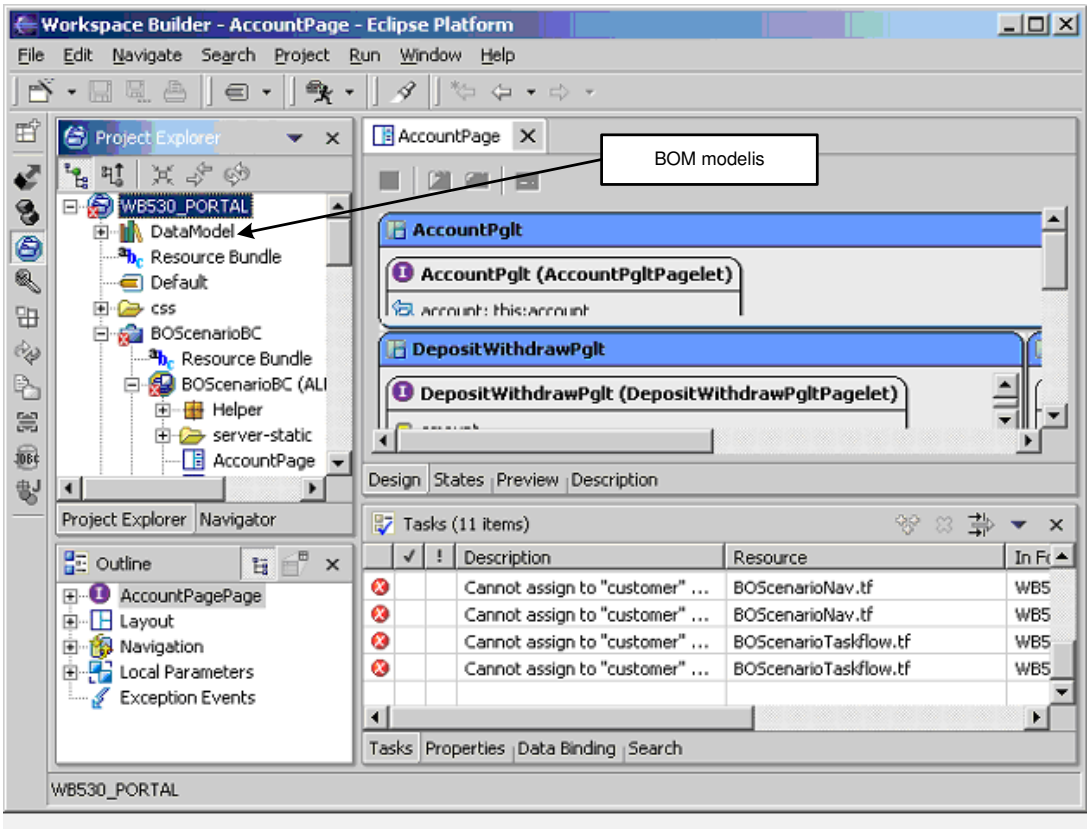
Dažnai projektuojama didelė ir sudėtinga sistema yra skaidoma į mažesnes logines dalis, kurios aprašomos atskirais modeliais. Taip pat sistemos turi komunikuoti su kitomis sistemomis, kurioms aprašyti yra kuriami nauji modeliai. Tokių sistemų realizavimo metu dažnai tenka rankomis realizuoti vienos objektų hierarchijos projekciją į kitos objektų hierarchiją. Siekiant supaprastinti ir pagreitinti šį procesą, buvo realizuotas BOM objekto į objektą projekcijos įrankis (žr. 14 pav.). Jis leidžia grafiškai projektuoti projekcijas, dirbant su duomenų objektais aprašytais PIM modeliuose. Sugeneruotas kodas naudoja tik duomenų objektų sąsajas, todėl šios projekcijos visiškai nepriklauso nuo sistemą realizuojančių technologijų. Objekto į objektą projekcija yra saugoma UML modelyje, aprašytame jam skirtu profiliu.



14 pav. – BOM aplinkos objekto į objektą projekcijos įrankis

3.5.4 BOM Designer integracija su grafinės aplinkos redaktoriumi WorkspaceBuilder

Programinės įrangos kūrimas neapsiriboja veiklos logikos realizacija. Taip pat reikalinga ir sistemos grafinė aplinka. Atsižvelgiant į tai buvo realizuota BOM Designer integracija su grafinės aplinkos redaktoriumi WorkspaceBuilder (15 pav.). WorkspaceBuilder naudodamas BOM API (*Application Programming Interface*) nuskaito PIM modelyje suprojektuotus duomenų ir veiklos objektus, ir susieja juos su grafiniais komponentas. Tokiu būdu programinės įrangos kūrėjai gali projektuoti sistemą, rašyti veiklos logiką ir realizuoti grafinę vartotojo sąsają (GUI), naudojant vienos aplinkos įrankius.



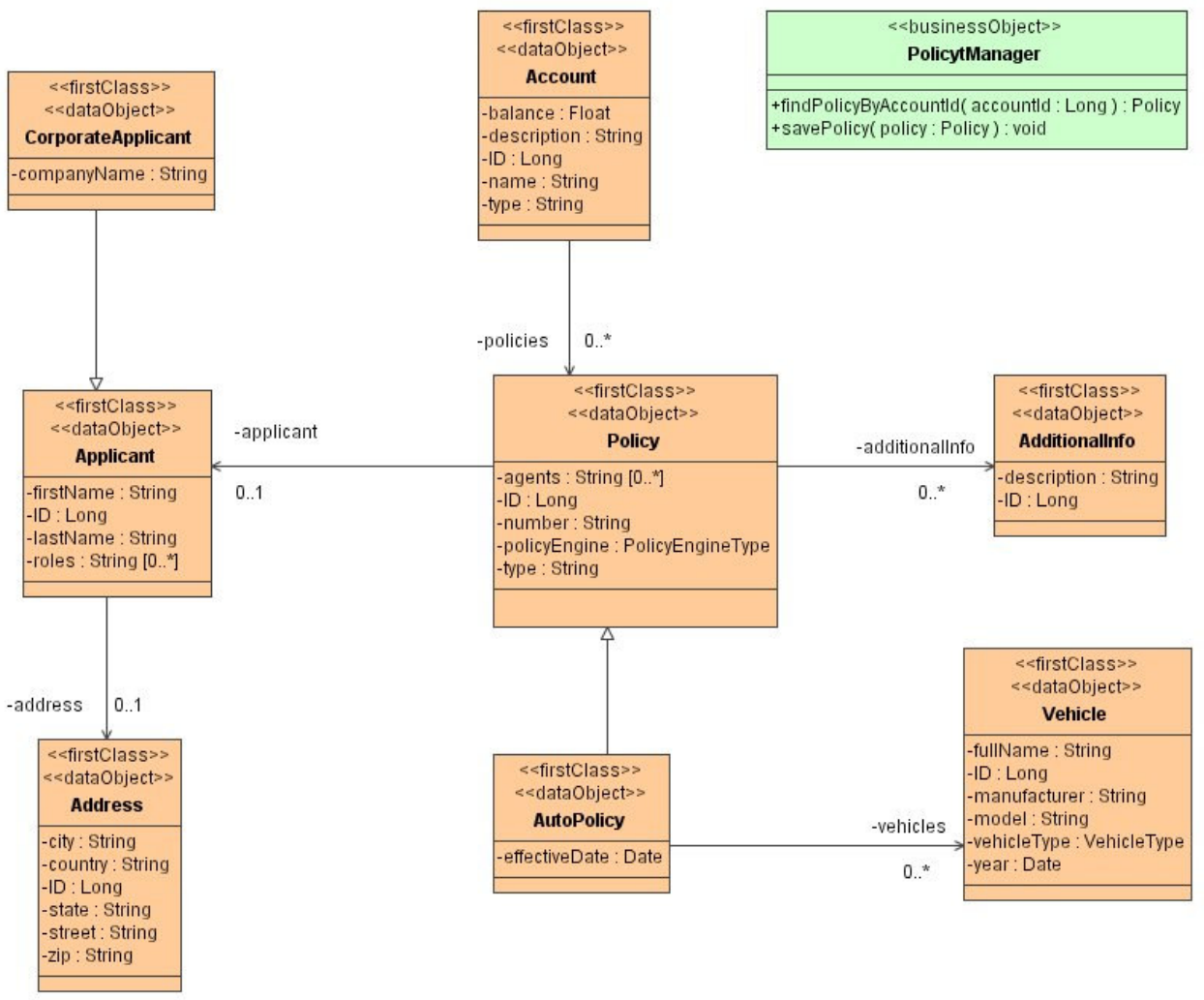
15 pav. – BOM Designer integracija su grafinės aplinkos redaktoriumi WorkspaceBuilder

4 EKSPERMENTINIS MDA APLINKOS NAUDOJIMAS

Šiame skyriuje pateikiami eksperimentinio MDA aplinkos naudojimo rezultatai ir įvertinimai.

4.1 Eksperimentinis išeities kodo generavimas

Eksperimentiniam išeities kodo generavimui naudojamas PIM modelis turintis 8 duomenų objektus ir 1 veiklos objektą yra pavaizduotas (žr. 16 pav.).



16 pav. Eksperimentinis PIM modelis

Eksperimentinio kodo generavimo rezultatai pateikti 5 lent. .

5 lent. Eksperimentinio išeities kodo generavimo rezultatai

Vertinamas elementas	Kiekis
PIM modelio elementų skaičius	9
Sugeneruotų klasių skaičius	28
Sugeneruotų duomenų bazės metaduomenų failų skaičius	8
Sugeneruotų išeities kodo eilučių skaičius	10951
Sugeneruotų duomenų bazės metaduomenų eilučių skaičius	336

4.2 Eksperimentinis veiklos logikos rašymas

Rašant eksperimentinę veiklos logiką naudojamas iš eksperimentinio modelio, aprašyto skyriuje 4.1, sugeneruotas kodas.

Veiklos objektų sąsajos gavimas:

```
PolicyManager policyManager = DataobjectsBOFactory.getPolicytManager();
```

Naujo duomenų objekto sukūrimas:

```
AutoPolicy autoPolicy = DataobjectsDOFactory.createAutoPolicy();
```

Sąsajos darbui su duomenų saugykla gavimas:

```
AutoPolicyHelper helper =
      DataobjectsHelperFactory.getAutoPolicyHelper();
```

Duomenų objekto išsaugojimas:

```
helper.saveAutoPolicy(autoPolicy);
```

Duomenų objekto pašalinimas:

```
helper.deleteAutoPolicy(autoPolicy);
```

Išsaugoto duomenų objekto gavimas:

```
Long policyID = new Long(1);
AutoPolicy autoPolicy = helper.getAutoPolicyById(policyID);
```

Duomenų objekto paieška naudojant filtrą:

```
String policyTypeToSearch = "DRC";
String filter = "policy.type == "+policyTypeToSearch;
List results = helper.searchAutoPolicy(filter);
```

Duomenų objekto paieška, naudojant užklausą pagal kriterijus (*Query by criteria*):

```
String policyTypeToSearch = "DRC";
Criteria criteria = helper.createAutoPolicySearchCriteria();
criteria.add(ExpressionHelper.eq("type", policyTypeToSearch));
List results = helper.searchAutoPolicyByCriteria(criteria);
```

Duomenų objekto paieška, naudojant užklausą pagal pavyzdį (*Query by example*):

```
Criteria criteria = helper.createAutoPolicySearchCriteria();
AutoPolicy autoPolicy = DataobjectsDOFactory.createAutoPolicy();
autoPolicy.setNumber("NR-1111");
autoPolicy.setType("DRC");
Example ex = Example.create(autoPolicy);
ex.excludeProperty("number");
criteria.add(ex);
List results = helper.searchAutoPolicyByCriteria(criteria);
```

Išeities kodo pavyzdžiuose matome, kad veiklos logika rašoma abstrakčiai, nesigilinant į sistemą realizuojančias technologijas. Programinės įrangos kūrėjui pakanka susipažinti su sąsajų fazėje generuojamais objektų tipais, jų paskirtimi ir BOM infrastruktūra. Atisižvelgiant į šias generuojamo kodo savybes, darbui su MDA aplinka nereikia aukštos kvalifikacijos programinės įrangos kūrėjų, nes visi technologiniai sprendimai yra realizuoti, aprašant transformacijų specifikacijas. Esant poreikiui, programinės įrangos kūrimo proceso metu, į projektą lengvai galima įtraukti naujus resursus, be laiko sąnaudų reikalingų naujų technologijų įsisavinimui.

Taikant dviejų žingsnių kodo generavimo strategiją, veiklos logika yra aprašoma naudojant sąsajų fazėje sugeneruota išeities kodą. Sistemos realizacijos technologija yra lengvai keičiama realizacijos fazės konfigūracijoje, pakeičiant naudojamą architektūrą. Taip gaunama nuo technologijos nepriklausoma veiklos logika, kuri yra lengvai pernešama į skirtingas technologijas ar platformas, pergeneravus išeities kodą realizacijos fazėje. Išeities kodo pavyzdžiuose, paieška naudojant filtrą yra dalinai susijusi su sistemą realizuojančia technologija, nes paieškos filtrų sintaksė nežymiai skiriasi priklausomai nuo technologijos. Ši problema yra

sprendžiama veiklos logikos realizacijoje, naudojant užklausas pagal kriterijus arba užklausas pagal pavyzdžius.

4.3 Eksperimentinė produktyvumo analizė

BOM designer produktyvumui įvertinti buvo realizuota demonstracinė informacinė sistema (žr. priede [8.4]). Sistemos realizacijai buvo naudota JDO permatomo duomenų išsaugojimo technologija. Sistemos veiklos logikos realizacija sudaro:

- 359 BOM modelio elementai:
  - 35 veiklos objektai
  - 324 duomenų objektai
- 1397 veikimo aplinkos (*runtime*) klasės
- 14% (~11000 eilutės) rankomis rašyto kodo
- 86% (~66500 eilutės) sugeneruoto kodo
  - 65000 eilutės Java išeities kodo
  - 1500 JDO metaduomenų

Statistinės realizuotos demonstracinės IS realizacijos laiko sąnaudos yra pateiktos 6 lent. Veiklos logikos rašymas, visų sistemos realizacijos laiko sąnaudų atžvilgiu, užima tik 10%. Tai byloja apie didelį MDA aplinkos panaudojimo produktyvumą ir leidžia programinės įrangos kūrėjams daugiau laiko skirti kitų sistemos dalių, tokių kaip grafinė vartotojo sąsaja, realizacijai.

6 lent. Demonstracinės IS realizacijos laiko sąnaudos

Sistemos realizavimo darbai	Vieno žmogaus mėnesiai sistemos realizavimui	Procentiškai sugaištas laikas
GUI	30	60%
BOM	5	10%
Kiti	15	30%

7 lent. pateikti statistiniai generuojamo kodo efektyvumo rodikliai, nurodomi literatūroje [5] .

7 lent. Programos kodo generavimo efektyvumas (ištrauka iš [5])

	Neautomatizuotas kodo rašymas	**Paprastas UML modeliavimas**	**MDA**
Rankiniu būdu parašytų kodo eilučių skaičius	126	71	20
Sugeneruotų eilučių skaičius	0	55	106
Visa kodo apimtis eilutėmis	126	126	126
Rankomis parašytų eilučių kiekis [%]	100	56	16

Realizuojant demonstracinę IS sistemą, sugeneruoto kodo santykis 86% yra labai artimas statistiniam generuojamo kodo santykiui 85.71%. Naudojant Exigen kompanijos metrika, užsakomosios paslaugos (*Outsourcing*) projektų realizavimo laikui įvertinti, apskaičiuota, kad naudojant įprastą programinės įrangos kūrimo procesą, komanda, sudaryta iš 10 aukštos kvalifikacijos programinės įrangos kūrėjų, demonstracinė IS realizacijai užtruktų apie 13 mėnesių. Matome, kad sudėtingos sistemos, naudojant BOM Designer aplinką, realizuojamos apytiksliai 2.6 karto greičiau, nei naudojant įprasta programinės įrangos kūrimo procesą.

4.4 Pasikartojančių klaidų įvertinimas

Realizuojant didelių sistemų veiklos logiką, programinės įrangos kūrėjai turi atlikti aibę pasikartojančių užduočių. Dažniausiai tai yra išeities kodas duomenų objekto išsaugojimui, nuskaitymui, šalinimui ir paieškai. Programinės įrangos kūrimo proceso metu dažnai keičiasi ir reikalavimai programinei įrangai. Todėl, jau realizuotos, pasikartojančios užduotys turi būti atnaujintos ir realizuotos naujos. Realizuojant pasikartojančias užduotis, natūraliai yra daromos pasikartojančios klaidos. Naudojant BOM Designer, pasikartojančios užduotis yra sugeneruojamos automatiškai, naudojant jau kokybės testais išbandytas transformacijas, todėl iš programinės įrangos kūrėjo yra atimama galimybė daryti pasikartojančias klaidas. Testuojant sistemas, aptinkamos tik loginio tipo klaidos veiklos objektų realizacijose.

4.5 Lygiagretaus programinės įrangos kūrimo galimybių įvertinimas

BOM designer suteikia puikias lygiagretaus programinės įrangos kūrimo galimybes. BOM modeliai gali būti skaidomi į mažesnes dalis ir projektuojami nepriklausomai. BOM objekto į duomenų bazę ir BOM objekto į objektą projekcijos įrankiai gali būti naudojami lygiagrečiai, nes projekcijų projektavimui naudojamas tik BOM modelis. Grafinės vartotojo aplinkos projektavimui ir realizacijai taip pat pakanka BOM modelio, o sugeneruotas kodas, naudojamas tik sistemos diegimo fazėje. BOM modelis į išeities kodą transformuojamas dviem žingsniais. Tai sąsajų ir realizacijos fazės. Atlikus sąsajų transformaciją, jau galima lygiagrečiai realizuoti veiklos logiką, nesigilinant į tikrąsias sistemą realizuosiančias technologijas. Šios BOM Designer savybės leidžia optimaliai ir efektyviai paskirstyti užduotis programinės įrangos kūrėjams.

4.6 Pakartotino panaudojimo įvertinimas

Naudojant BOM Designer, realizuojamos sistemos dalys gali būti aprašytos aibe BOM modelių. Anksčiau suprojektuotą BOM modelį galima susieti su naujai projektuojamais modeliais ir jam pakartotinai sugeneruoti išeities kodą. Taip pat galima susieti BOM modelius iš skirtingų projektų, todėl jau realizuota sistemos dalis gali būti panaudota realizuojant naujas sistemas. Kuriant programinę įrangą, orientuotą į tam tikrą paslaugų sektorių (*Vertical applications*), galima iš pradžių realizuoti šabloninę sistemą. Ši šabloninė realizacija, naudojant BOM Designer modelio transformavimo ir kodo generavimo galimybes, lengvai gali būti modifikuota, atsižvelgiant į naujus sistemos reikalavimus, išsaugant didžiąją dalį veiklos logikos, nes to paties segmento veiklos procesai dažniausiai neturi esminių skirtumų.

4.7 Bendras MDA aplinkos panaudojimo įvertinimas

Bendras MDA aplinkos panaudojimo įvertinimas yra pateiktas 8 lent.

8 lent. MDA aplinkos panaudojimo įvertinimas

Įvertinamas kriterijus	l.aukštas	aukštas	vidutiniškas	žemas	l. žemas
Produktyvumas	√				
Naudojimo sudėtingumas			√		
Pasikartojančių užduočių automatizavimas	√				
Techninių įgūdžių reikalavimai programinės įrangos kūrėjams				√	
Lygiagretaus darbo galimybės		√			
Pernešamumas	√				
Pakartotino panaudojimo galimybės		√			

5 IŠVADOS

1. Modeliais pagrista architektūra (MDA) yra intensyviai vystoma, naujos kartos programinės įrangos kūrimo strategija. Ši architektūra yra palyginti jauna, todėl ji daugiau orientuota į programinės įrangos kūrimo metodiką ir griežtai nespecifikuoja aplinkos realizacijos sprendimų variantų.
2. Remiantis MDA programinės įrangos kūrimo proceso apžvalga ir MDA aplinkos esminių elementų ir jų funkcijų analize, pasirinktos modeliavimo ir transformacijų realizavimo strategijos.
3. BOM Designer yra, tyrimo metu sukurta, MDA aplinkos realizacija, kurią sudaro:
    - BOM modelio profilis
    - PIM modelio į PSM modelį transformacijos
    - PSM modelio į išeities kodą transformacijos
    - BOM modelio redaktorius
    - BOM objekto į duomenų bazę projekcijos įrankis
    - BOM objekto į objektą projekcijos įrankis
    - Integracija su grafinės aplinkos redaktoriumi WorkspaceBuilder
4. Eksperimentinio BOM Designer aplinkos naudojimo metu, pastebėta, kad MDA aplinkos naudojimas padeda išvengti pasikartojančių klaidų, automatizavus pasikartojančias užduotis, reikalauja mažesnių techninių įgūdžių programinės įrangos kūrėjams, padidina programinės įrangos pernešamumą ir pakartotino panaudojimo galimybes.
5. BOM Designer produktyvumui įvertinti realizuota demonstracinė IS:
    - Demonstracinės IS realizacijoje, sugeneruoto kodo santykis 86% yra labai artimas statistiniam generuojamo kodo santykiui 85.71% [5].
    - Veiklos logikos rašymas, visų sistemos realizacijos laiko sąnaudų atžvilgiu, užima tik 10%.
    - Naudojant Exigen kompanijos metriką, apskaičiuota, kad demonstracinė IS realizuota 2.6 karto greičiau, nei naudojant įprastą programinės įrangos kūrimo procesą.

6 LITERATŪRA

1. Meta Object Facility (MOF) Specification. OMG, 2002 [žiūrėta 2003-04-20]. Prieiga per internetą: http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf.
2. Model Driven Architekture (MDA). OMG, 2001 [žiūrėta 2003-02-04]. Prieiga per internetą: http://www.omg.org/docs/ormsc/01-07-01.pdf.
3. Kleppe A., Warmer J., Bast W. MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003.
4. MDA Guide Version 1.0.1. OMG, 2003 [žiūrėta 2003-10-11]. Prieiga per internetą: http://www.omg.org/docs/omg/03-06-01.pdf.
5. Bettin J. Model-Driven Architecture – Implementation and Metrics. Version 1.1. 2003 [žiūrėt 2004-01-29]. Prieiga per internetą: http://www.softmetaware.com/mda-implementationandmetrics.pdf.
6. Frankel D. S. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003
7. Meta Object Facility (MOF) 2.0 Core Proposal. OMG, 2003 [žiūrėta 2003-05-05]. Prieiga per internetą: http://www.omg.org/docs/ad/03-04-07.pdf.
8. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. OMG, 2002 [žiūrėta 2003-08-01]. Prieiga per internetą: http://www.omg.org/docs/ad/02-04-10.pdf.
9. Judson S. R., France R. B., Carver D. L. Specifying Model Transformations at the Metamodel Level. [žiūrėta 2003-04-07] Prieiga per internetą: http://www.metamodel.com/wisme-2003/19.pdf.
10. XML Metadata Interchange (XMI) Specification. OMG, 2003 [žiūrėta 2003-10-16]. Prieiga per internetą: http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf.
11. Budinsky F., Steinberg D., Merks E., Ellersick R., Grose T. Eclipse modeling framework. Addison-Wesley, 2004.
12. Cheesman J., Daniels J. UML Components. Addison-Wesley, 2001.
13. Hubert R. Convergent Architecture. Addison-Wesley, 2002.
14. Velocity svetainė. Jakarta [žiūrėta 2003-05-05]. Prieiga per internetą: http://jakarta.apache.org/velocity/index.html.

7 TERMINŲ IR SANTRUMPŲ ŽODYNAS

9 lent. Terminai

Lietuviškai	**Angliškai**	**Paaiškinimas**
Abstrakcija	Abstraction	Esybės ar proceso aprašymas atmetant detales kurios nagrinėjimo metu laikomos neesminėmis.
Detalizavimas	Refinement	Detalizavimas – tai detalesnis sistemos aprašymas kuris vienareikšmiškai atitinka abstraktesnį sistemos aprašymą.
Infrastruktūra	Infrastructure	Infrastruktūra (taip pat vadinama skaičiavimo (*computing*) infrastruktūra) yra programinės ar aparatinės sistemos dalys kurios laikomos duotomis taikomosios programos realizavimo metu.
Modelis	Model	Modelis yra sistemos dalies (funkcijos, struktūros ir/arba elgesio) atvaizdavimas kokioje nors sintaksiškai formalioje kalboje (pvz. UML, MOF).
Nuo platformos nepriklausantys modeliai	Platform independent models (PIM)	PIM pateikia formalią sistemos struktūros ir funkcijų specifikaciją abstrahuotą nuo techninių realizacijos detalių.
Nuo skaičiavimų nepriklausomas modelis	Computationally Independent Model	CIM yra sistemos atvaizdavimas iš nuo skaičiavimų nepriklausomo požiūrio. CIM neparodo programos struktūros detalių. CIM kartais vadinamas probleminės srities modeliu.
Objektiškai orientuotas	Object Oriented	Programavimo būdas paremtas objektais
Platforma	Platform	Platforma yra programinė infrastruktūra

		realizuota tam tikra technologija (pvz.: Unix platforma, CORBA platforma, Windows platforma, .NET platforma, J2EE platforma)
Platformai specifiniai modeliai	Platform specific models (PSM)	PSM yra išreikštas tikslinės platformos modelio terminais ir pateikia formalią sistemos specifikaciją tam tikroje tikslinėje platformoje.
Projekcija	Mapping	Taisyklių ir metodų aibė naudojama modelio modifikavimui norint gauti kitą modelį. Sukuriant modelių projekcijas į tam tikras platformas vėliau galima atlikti automatinį ar rankinį modelio transformavimą į tikslinę platformą.
Vykdymo aplinka	Execution environment	Vykdymo aplinka priklauso nuo programinės ir aparatinės infrastruktūros ir yra realizuota vienos ar kelių platformų.

10 lent. Santrumpos

Santrumpa	**Pilnas pavadinimas**	**Paaiškinimas/Vertimas**
IT	Informacinės technologijos	-
IS	Informacinė sistema	
BOM	Business Object Model	Veiklos objektų modelis, naudojamas MDA aplinkos realizacijoje
CWM	Common Warehouse Metamodel	Duomenų saugyklų metamodelio standartas.
DTD	Document Type Definition	Kalba naudojama dokumento struktūrai apibrėžti.
EJB	Enterprise Java Beans	J2EE paskirstytųjų komponentų architektūra sukurta kompanijos Sun Microsystems Inc.
GUI	Graphic User Interface	Grafinė vartotojo sąsaja
JDO	Java Data Objects	Permatoma (*transparent*) duomenų išsaugojimo

		technologija sukurta kompanijos Sun Microsystems Inc.
J2EE	Java 2 Enterprise Edition	Java 2 verslo sistemų programavimo platforma.
M0	Metalevel-0 (Runtime)	0-is metalygis – atitinka konkretų taikomosios programos vykdymą.
M1	Metalevel-1 (Application Model)	1-as metalygis – atitinka konkrečios programos modelį (pvz.: objektinės programos specifikacija UML, IDL sąsajų aprašai, Java programos tekstas).
M2	Metalevel-2 (Application Metamodel)	2-as metalygis – atitinka programos modelio gramatiką.
M3	Metalevel-3 (Application Meta-Metamodel)	3-ias metalygis – atitinka gramatiką, leidžiančią apibrėžti naujas modeliavimo ir gramatikų apibrėžimo gramatikas (pvz.: MOF gali apibrėžti save ir kitas modeliavimo kalbas kaip UML).
MDA	Model Driven Architecture	Modeliais pagrįsta architektūra.
MOF	Meta-Object Facility	Metaobjektų infrastruktūra – metamodeliavimo ir metaduomenų saugyklų standartas.
OCL	Object Constraint Language	OMG objektų apribojimų specifikavimo kalba. Šiuo metu kūrimo stadijoje.
OMG	Object Management Group	Kompanijos pavadinimas kuri sukūrė UML, MOF, CWM, XMI, CORBA ir daugybe kitų specifikacijų.
PIM	Platform Independent Model	Nuo platformos nepriklausomas modelis.
PSM	Platform Specific Model	Platformai specifinis modelis.
RFP	Request for Proposal	Užklausa pasiūlymams
RUP	Rational Unified Process	Formalizuota programinės įrangos kūrimo metodologija, akcentuojanti reikalavimų surinkimo, analizės ir projektavimo veiklų svarbą programinės įrangos kūrimo procese, prieš pradedant rašyti išeities kodą
UML	Unified Modeling Language	Universali, OMG konsorciumo sukurta,

		modeliavimo kalba skirta aprašyti objektinės programinės įrangos sudėtiniams elementams ir sistemoms.
XMI	XML Metadata Interchange	OMG metaduomenų apsikeitimo standartas pagrįstas XML dokumentais.
XML	Extensible Markup Language	W3C konsorciumo standartas, išplečiama kalba, naudojama duomenų struktūros sužymėjimui, saugojimui ir perdavimui.
XSL	Extensible Stylesheet Language	W3C konsorciumo standartas, leidžiantis transformuoti viena XML į kitą
XP	Extreme programming	Programinės įrangos kūrimo metodika, orientuota į išeities kodo rašymą ir testavimą
4GL	4 Generation Languages	Tai objektiškai orientuotos kalbos, tokios kaip Java, C++ ar C#

8 PRIEDAI

8.1 Sugeneruotų JDO metaduomenų pavyzdys

```
<?xml version="1.0" encoding="UTF-8"?>
<jdo>
    <package name="dataobjects.infrastructure">
        <class      objectid-class="PolicyJDOID"      name="PolicyJDO"      identity-
type="application">
            <extension vendor-name="kodo" value="base" key="jdbc-class-map">
                <extension vendor-name="kodo" value="POLICYJDOX" key="table"/>
            </extension>
            <extension  vendor-name="kodo"  value="version-number"  key="jdbc-version-
ind">
                <extension vendor-name="kodo" value="JDOLOCKX" key="column"/>
            </extension>
            <extension vendor-name="kodo" value="in-class-name" key="jdbc-class-ind">
                <extension vendor-name="kodo" value="JDOCLASSX" key="column"/>
            </extension>
            <field name="ID" primary-key="true">
                <extension vendor-name="kodo" value="value" key="jdbc-field-map">
                    <extension vendor-name="kodo" value="IDX" key="column"/>
                </extension>
            </field>
            <field name="number">
                <extension vendor-name="kodo" value="value" key="jdbc-field-map">
                    <extension vendor-name="kodo" value="NUMBERX" key="column"/>
                </extension>
            </field>
            <field name="policyEngine">
                <extension vendor-name="kodo" value="value" key="jdbc-field-map">
                    <extension vendor-name="kodo" value="POLICYENGINEX" key="column"/>
                </extension>
            </field>
            <field name="applicant">
                <extension vendor-name="kodo" value="one-one" key="jdbc-field-map">
                    <extension         vendor-name="kodo"         value="ID_APPLICANTX"
key="column.IDX"/>
                </extension>
            </field>
            <field name="additionalInfo">
                <collection                                                element-
```

```
type="dataobjects.infrastructure.AdditionalInfoJDO"/>
                <extension vendor-name="kodo" value="many-many" key="jdbc-field-map">
                    <extension vendor-name="kodo"
                        value="ID_ADDITIONALINFOX" key="element-column.IDX"/>
                    <extension    vendor-name="kodo"    value="ID_JDOIDX"    key="ref-
column.IDX"/>
                    <extension vendor-name="kodo"
                        value="POLICYJDO_ADDITIONALINFOX" key="table"/>
                </extension>
            </field>
        </class>
    </package>
</jdo>
```

8.2 Sugeneruotų Hibernate metaduomenų pavyzdys

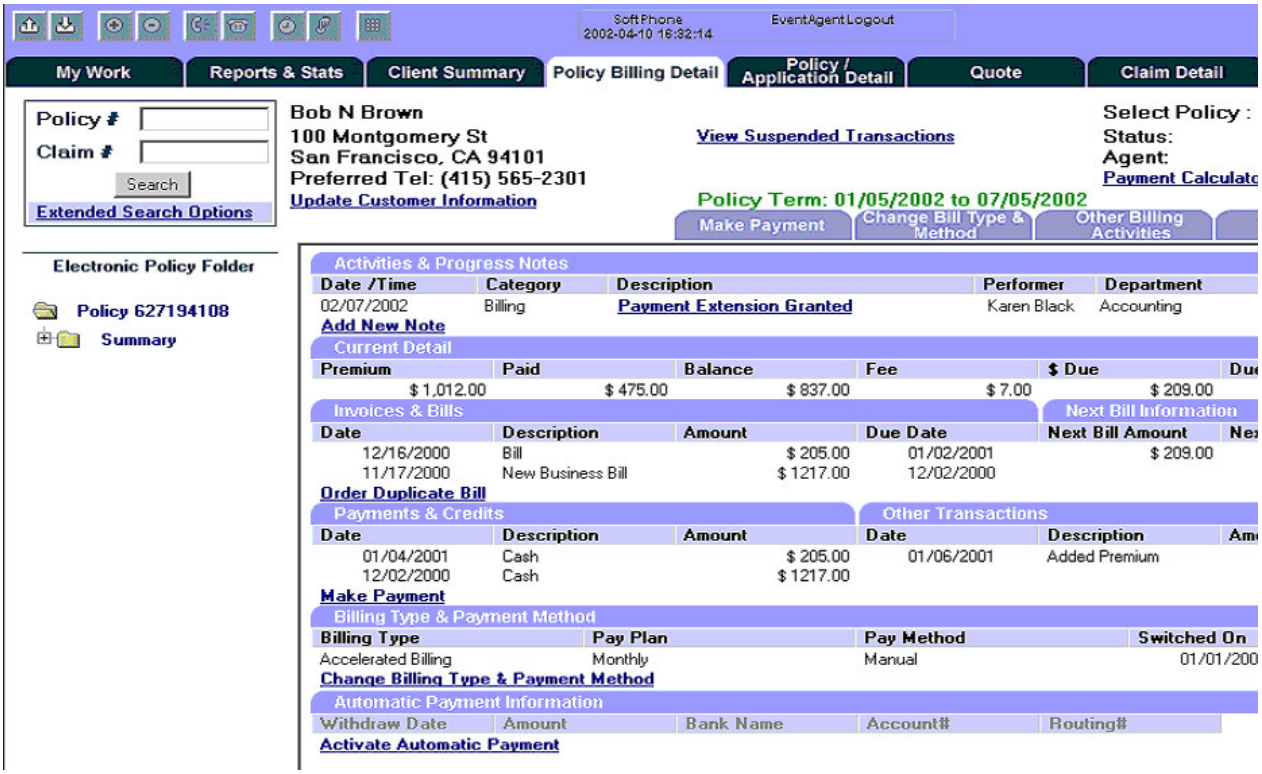
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE  hibernate-mapping  PUBLIC  "-//Hibernate/Hibernate  Mapping  DTD  2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping package="dataobjects.infrastructure" default-cascade="save-update">
<class  name="PolicyHibernate"  discriminator-value="Policy.dataobjects"  optimistic-
lock="version" table="POLICY">
    <composite-id name="id__" class="PolicyHibernateID" access="field">
        <key-property name="ID" column="ID" access="field"/>
    </composite-id>
    <discriminator column="class"/>
    <version name="version__" column="version__" access="field"/>
    <property name="number" column="NUMBER0" access="field"/>
    <bag name="additionalInfo" lazy="true" access="field">
        <key column="ADDITIONALINFO_ID"/>
        <one-to-many class="dataobjects.infrastructure.AdditionalInfoHibernate"/>
    </bag>
    <many-to-one name="applicant" class="dataobjects.infrastructure.ApplicantHibernate"
access="field" column="APPLICANT_ID"/>
</class>
</hibernate-mapping>
```

8.3 BOM veiklos logikos realizacijos pavyzdys

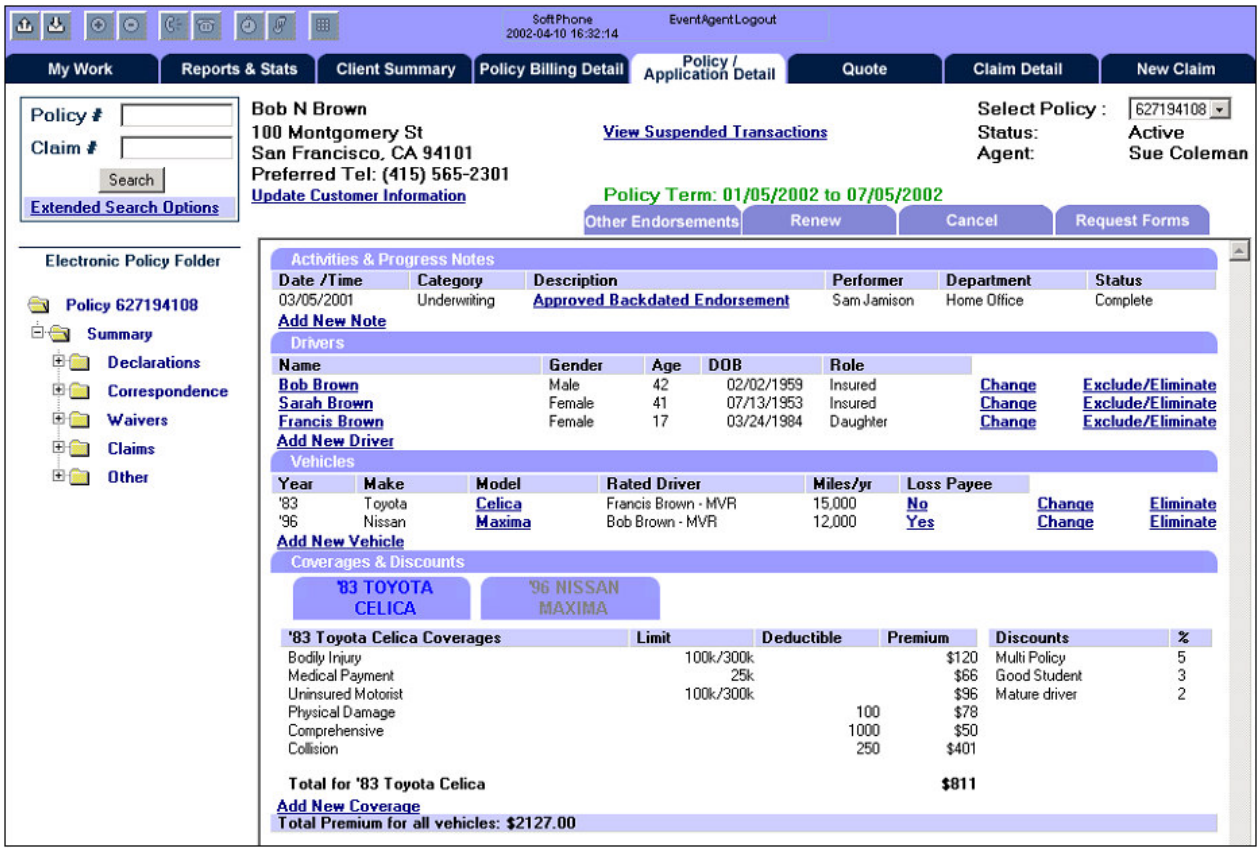
```
package demo.examples;

import java.util.List;
import com.exigen.ef.persistence.PersistenceService;
import com.exigen.ef.persistence.util.transaction.Transaction;
import demo.helper.ParentChildPkgHelperFactory;
public class Example{

    public void sampleScenario() {
        // creating Customer with 2 Accounts
        int idCustomer = 1;
        int idAccount = 0;
        // creating customer
        Customer customer = ExamplesDOFactory.createCustomer();
        parent.setId(new Integer(idCustomer));
        parent.setName(idCustomer + " customer");
        List accounts = customer.getAccounts();
        // first account
        {
            ++idAccount;
            Account account = ExamplesDOFactory.createAccount();
            account.setId(new Integer(idAccount));
            account.setDescription(idAccount + " account");
            account.setCustomer(customer);
            accounts.add(account);
        }
        // second account
        {
            ++idAccount;
            Account account = ExamplesDOFactory.createAccount();
            account.setId(new Integer(idAccount));
            account.setDescription(idAccount + " account");
            account.setCustomer(customer);
            accounts.add(account);
        }

        int saveMethod = 0;
        // saving customer using current transaction
        if (saveMethod == 0) {
            ExamplesHelperFactory.getCustomerHelper().saveCustomer(customer);
        }
```

```
        // if you need to do several updates in current transaction
        if (saveMethod == 1) {
            Transaction tx1 =
            PersistenceService.getTransactionService().getCurrentTransaction();
            tx1.beginTransaction();
            ExamplesHelperFactory.getCustomerHelper().saveCustomer(tx1,customer);
            tx1.commitTransaction();
        }

        // saving customer using explicit (new, isolated) transaction
        if (saveMethod == 2) {
            Transaction tx2 =
            PersistenceService.getTransactionService().createNewTransaction();
            ExamplesHelperFactory.getCustomerHelper().saveCustomer(tx2,customer);
        }

        // if you need to do several updates in explicit transaction
        if (saveMethod == 3) {
            Transaction tx3 =
            PersistenceService.getTransactionService().createNewTransaction();
            tx3.beginTransaction();
            ExamplesHelperFactory.getCustomerHelper().saveCustomer(tx3,customer);
            tx3.commitTransaction();
        }
    }
}
```

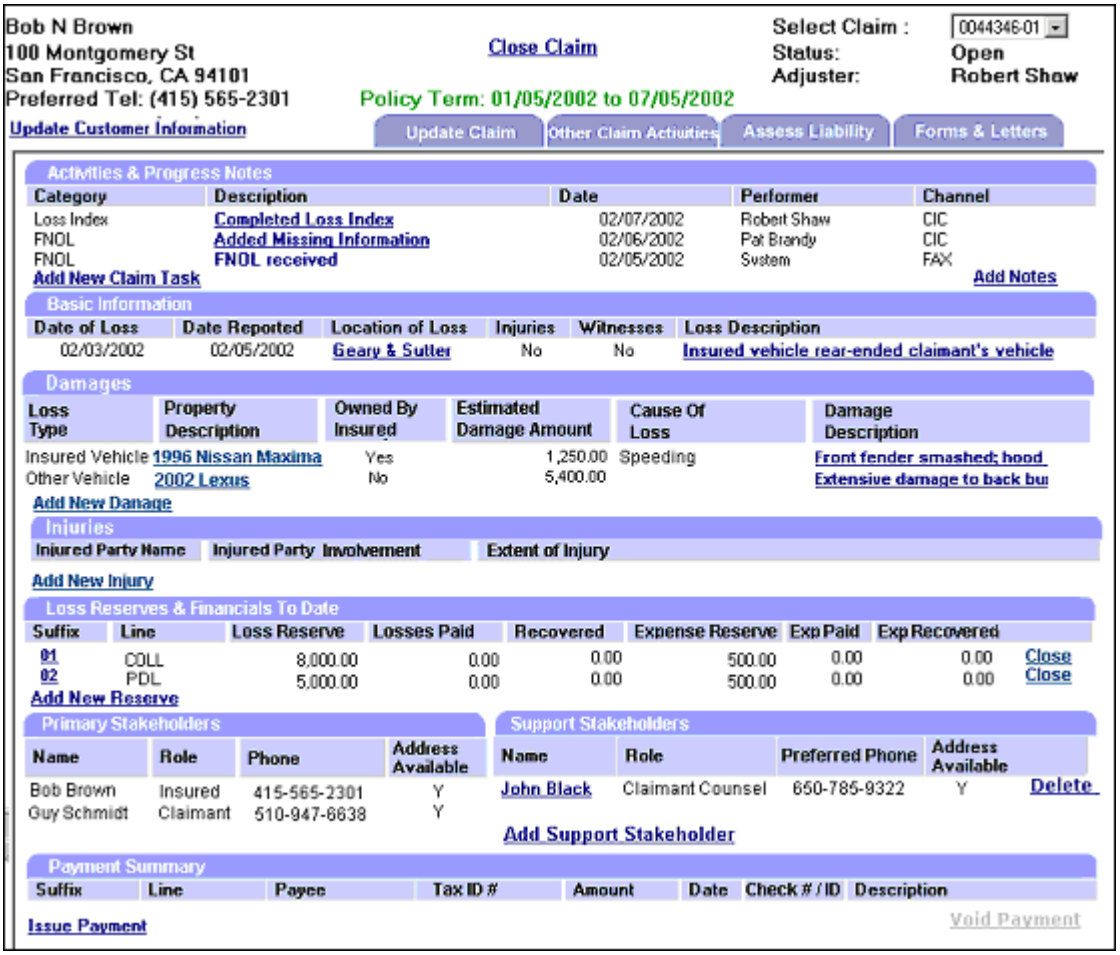
8.4 Demonstracinės IS vartotojo sąsaja



17 pav. Demonstracinės IS vartotojo sąsaja 1



18 pav. Demonstracinės IS vartotojo sąsaja 2



19 pav. Demonstracinės IS vartotojo sąsaja 3